

# Tutorial:

## *Model-Based Design Using Model Composer*

UG1259 (v2020.1) June 3, 2020

# Revision History

The following table shows the revision history for this document.

Section	Revision Summary
<b>06/03/2020 Version 2020.1</b>	
General Updates	<ul style="list-style-type: none"><li>• Vivado version</li><li>• Editorial updates</li></ul>

# Table of Contents

<b>Revision History</b> .....	<b>2</b>
<b>Model Composer Lab Overview</b> .....	<b>5</b>
Software Requirements.....	6
Launching Model Composer.....	6
Locating and Preparing the Tutorial Files.....	7
<b>Lab 1: Introduction to Model Composer</b> .....	<b>8</b>
Procedure.....	8
Step 1: Review the Model Composer Library.....	8
Step 2: Build Designs with Model Composer Blocks.....	9
Step 3: Work with Data Types.....	12
Conclusion.....	19
<b>Lab 2: Importing Code into Model Composer</b> .....	<b>20</b>
Step 1: Set up the Import Function Example.....	20
Step 2: Custom Blocks with Function Templates.....	22
Conclusion.....	27
<b>Lab 3: Debugging Imported C/C++-Code Using GDB Debugger</b> .....	<b>29</b>
Step 1: Set Up the Example to Debug the Import Function.....	29
Step 2: Debugging C/C++ Code Using GDB Debugger.....	32
Conclusion.....	35
<b>Lab 4: Debugging Imported C/C++-Code Using Visual Studio</b> .....	<b>36</b>
Step 1: Set Up the Example to Debug the Import Function.....	36
Step 2: Debugging C/C++ Code Using Visual Studio.....	39
Conclusion.....	44
<b>Lab 5: Automatic Code Generation</b> .....	<b>45</b>
Procedure.....	45
Step 1: Review Requirements for Generating Code.....	45
Step 2: Mapping Interfaces.....	47
Step 3: Generate IP from Model Composer Design.....	50

Step 4: Generate HLS Synthesizable Code.....	54
Step 5: Port a Model Composer Design to System Generator .....	57
Conclusion.....	62
<b>Appendix A: Additional Resources and Legal Notices.....</b>	<b>64</b>
Please Read: Important Legal Notices.....	64

# Model Composer Lab Overview

Xilinx<sup>®</sup> Model Composer is a model-based design tool that enables rapid design exploration within the Simulink<sup>®</sup> environment and accelerates the path to production on Xilinx programmable devices through automatic code generation.

Model Composer is designed as an add-on to Simulink and provides a library of performance-optimized blocks for design and implementation of algorithms on Xilinx FPGAs. The Model Composer library offers over 80 predefined blocks, including application-specific blocks for Computer Vision and Image Processing and functional blocks for Math, Linear Algebra, Logic, and Bitwise operations, among others.

You can focus on expressing algorithms using blocks from the Xilinx Model Composer library as well as custom user-imported blocks, without worrying about implementation specifics, and leverage all the capabilities of Simulink<sup>®</sup>'s graphical environment for algorithm design, simulation, and functional verification. Model Composer then transforms your algorithmic specifications to production-quality implementation using automatic optimizations that extend the Xilinx High Level Synthesis technology.

This tutorial introduces the end-to-end workflow for using Model Composer.

The included labs are as follows:

- Lab 1: Introduction to Model Composer
  - Introduction to Model Composer Library Blocks for design
  - Integration with native Simulink and Support for vectors and matrices
  - Working with data types
- Lab 2: Create Custom Blocks in Model Composer
  - Using the `xmcImportFunction` command to specify functions defined in source and header files to import into Model Composer and create Model Composer blocks or a block library.
  - Creating custom blocks with Function templates.
- Lab 3: Debugging Imported C/C++-Code Using GDB Debugger
  - Identifying the custom library block, created using the `xmcImportFunction` feature.
  - Debugging C/C++ code using the GDB tool.
- Lab 4: Debugging Imported C/C++-Code Using Visual Studio
  - Identifying the custom library block, created using the `xmcImportFunction` feature.

- Debugging C/C++ code using Visual Studio.
- Lab 5: Automatic Code Generation
- Requirements for Code Generation
- Mapping Interfaces
- Generate an IP for use in the Vivado® IP integrator
- Generate Vivado HLS Synthesizable Code
- Port a Model Composer Synthesized Design into System GeneratorSystem Generator for DSP

---

## Software Requirements

The lab exercises in this tutorial require that you have installed the following software:

- MATLAB®: The MATLAB releases and simulation tools supported in this release of Model Composer are described in the Compatible Third-Party Tools section of the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#))
- Vivado® Design Suite release: 2020.x (Includes Vivado HLS)
- Model Composer: 2020.x

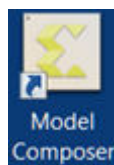
See the *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* ([UG973](#)) for a complete list and description of the system and software requirements.

---

## Launching Model Composer

To launch Model Composer:

- On Windows Systems
  - Select **Start** → **All Programs** → **Xilinx Design Tools** → **Model Composer 2020.x** → **Model Composer 2020.x**.
- OR
  - Double-click the Model Composer icon which was placed on your desktop after installation.



- On Linux Systems

You launch Model Composer under Linux using a shell script called `model_composer` located in the `<Model_composer_install_dir>/2020.x/bin` directory. Before launching this script, you must make sure the MATLAB executable can be found in your Linux system's `$PATH` environment variable for your Linux system. When you execute the `model_composer` script, it will launch the first MATLAB executable found in `$PATH` and attach Model Composer to that session of MATLAB. Also, the `model_composer` shell script supports all the options that MATLAB supports and all options can be passed as command line arguments to the `model_composer` script.

When Model Composer opens, you can confirm the version of MATLAB to which Model Composer is attached by entering the version command in the MATLAB Command Window.

```
>> version
ans =
'9.2.0.538062 (R2017a)'
```

---

## Locating and Preparing the Tutorial Files

There are separate project files and sources for each of the labs in this tutorial. You can find the design files for this tutorial on the [Xilinx](#) website.

1. Download the [Reference Design Files](#) from the Xilinx website.
2. Extract the ZIP file contents into any write-accessible location on your hard drive or network location.



**RECOMMENDED:** You will modify the tutorial design data while working through this tutorial. You should use a new copy of the `ModelComposer_Tutorial` directory extracted from `ug1259-model-composer-tutorial.zip` each time you start this tutorial.



**TIP:** This document assumes the tutorial files are stored at `C:\ModelComposer_Tutorial`. All path names and figures in this document refer to this path name. If you choose to store the tutorial in another location, adjust the path names accordingly.



**TIP:** Make sure to save the tutorial files in a folder structure with no spaces in them. There is a known limitation that does not support spaces in the directory structure for code generation.

# Introduction to Model Composer

This tutorial shows how you can use Model Composer for rapid algorithm design and simulation in the Simulink® environment.

## Procedure

This lab has the following steps:


- In Step 1, you examine the Model Composer Simulink library.
- In Step 2, you build a simple design using Model Composer blocks to see how Model Composer blocks integrate with native Simulink blocks and supported Signal Dimensions.
- In Step 3, you look at data types supported by Model Composer and the conversion between data types.

## Step 1: Review the Model Composer Library

In this step you see how Model Composer fits into the Simulink environment, and then review the categories of blocks available in the Model Composer library.

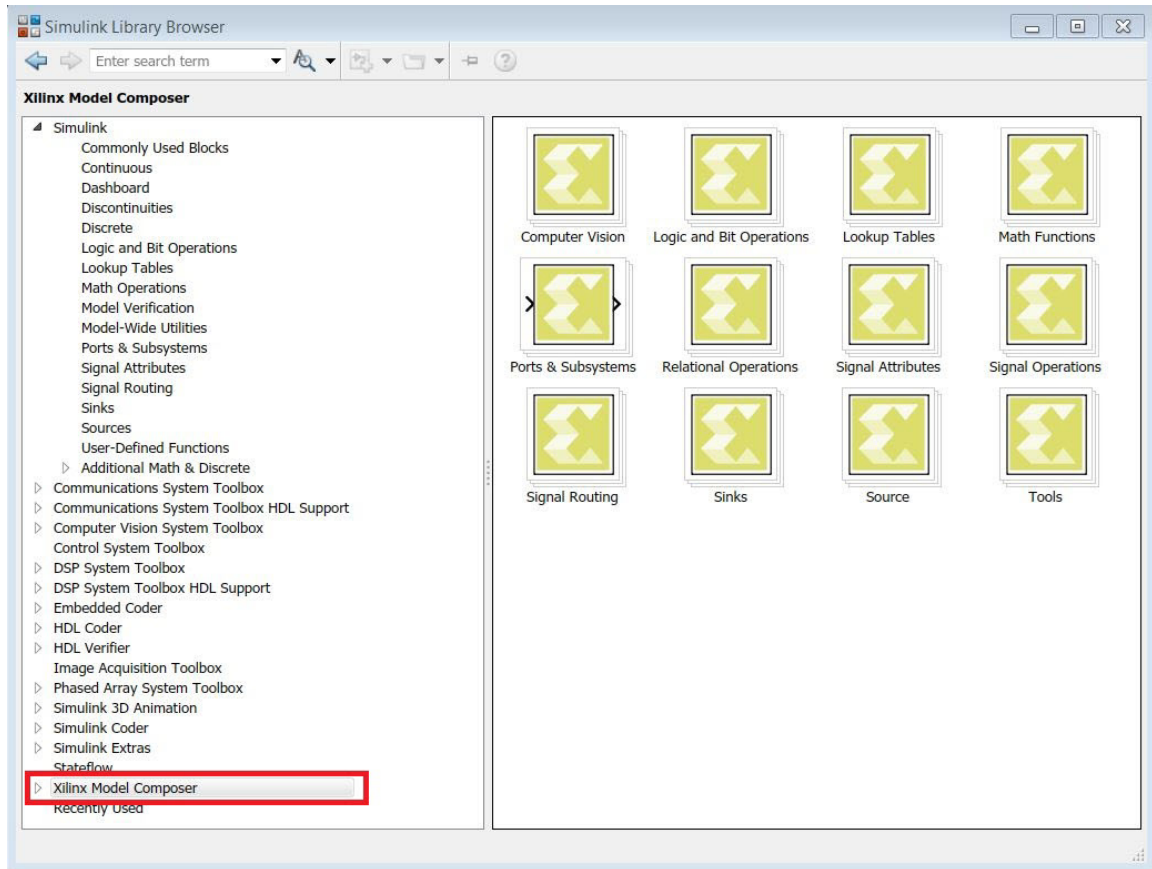
### Access Model Composer Library

Model Composer provides 80+ blocks for use within the Simulink environment. You can access these from within the Simulink Library Browser:

1. Use either of these techniques to open the Simulink Library Browser:
  - a. On the Home tab, click **Simulink**, and choose a model template. In the new model, click the **Library Browser** button. 
  - b. At the command prompt, type:

```
slLibraryBrowser
```

2. In the browser, navigate to the Xilinx Model Composer library.



The Model Composer blocks are organized into subcategories based on functionality. Spend a few minutes navigating through the sub-libraries and familiarizing yourself with the available blocks.

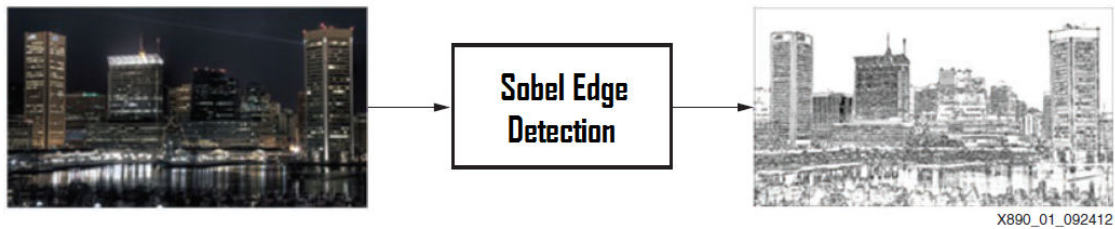
## Step 2: Build Designs with Model Composer Blocks

In this step, you build a simple design using the existing Model Composer blocks.

### Sobel Edge Detection: Algorithm Overview

Sobel edge detection is a classical algorithm in the field of image and video processing for the extraction of object edges. Edge detection using Sobel operators works on the premise of computing an estimate of the first derivative of an image to extract edge information.

Figure 1: Sobel Edge Detection



## Implementing Algorithm in Model Composer

1. In the MATLAB Current Folder, navigate to `ModelComposer_Tutorial\Lab1\Section1`.
2. Double-click the `Sobel_EdgeDetection_start.slx` model.

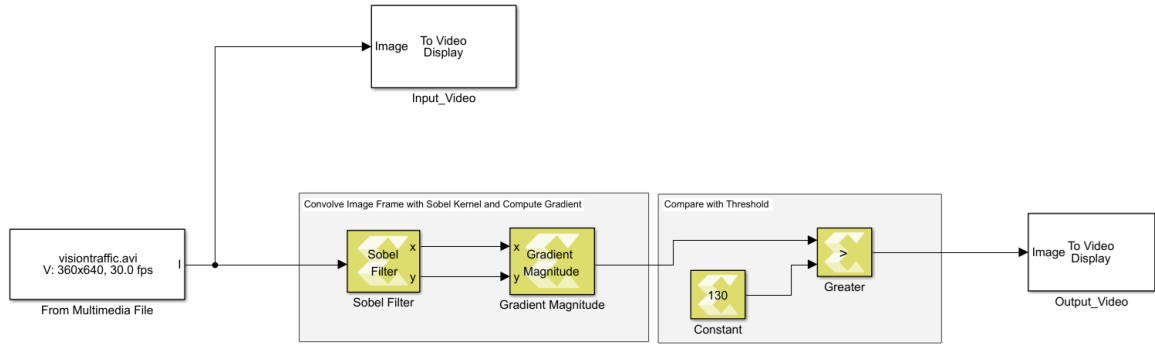
This model already contains source and sink blocks (from Simulink's Computer Vision System Toolbox), to stream video files as input directly into your algorithm and view the results. The model also contains some of the needed Model Composer blocks required for this section. Note the difference in appearance for the Model Composer blocks in the design versus the Simulink blocks.


3. From the Library Browser, select the **Sobel Filter** block from the Computer Vision sub-library of the Xilinx Model Composer library. Drag the block into the area labeled Convolve Image Frame with Sobel Kernel and Compute Gradient as shown in the following figure and connect the input of this block to the output of the From Multimedia File block.

**Note:** You can also add Model Composer blocks directly into your model by typing the block name onto the canvas (same as Simulink blocks).

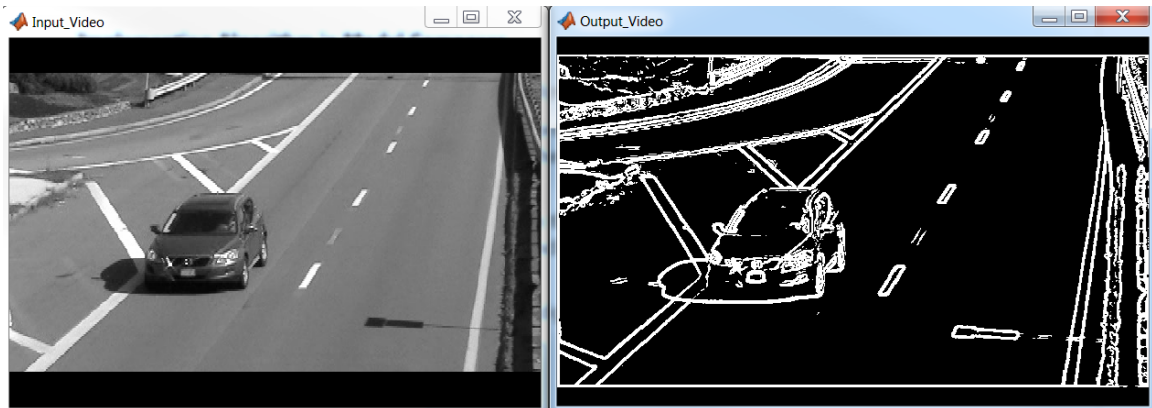


4. From the Library Browser, select the **Gradient Magnitude** block from the Xilinx Model Composer library (also found in the Computer Vision sub-library), drag it into the model, and connect the X and Y outputs of the Sobel Filter block to the input of this block.
5. Connect the rest of the blocks to complete the algorithm as shown in the following figure.



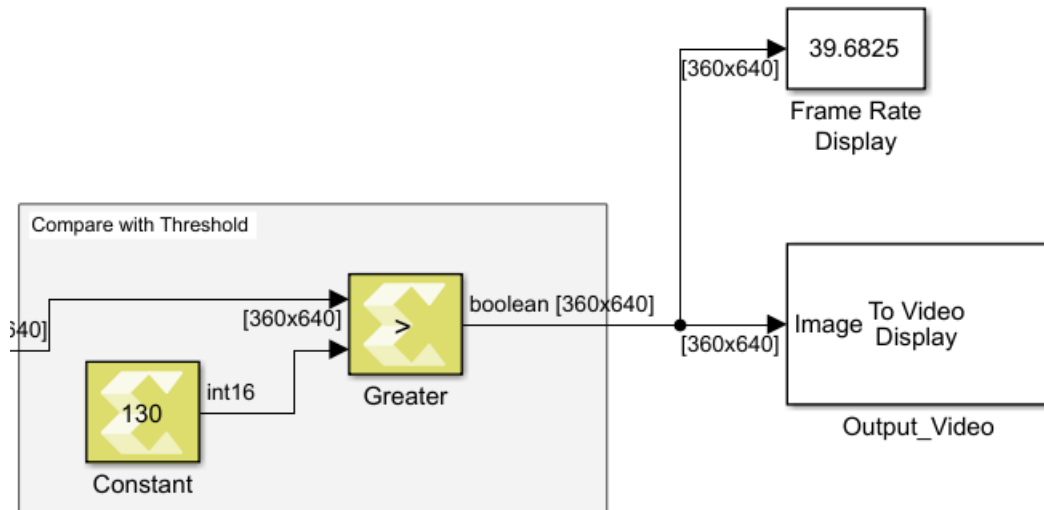
6. Select the **Simulation** → **Run** command or click the  button to simulate the model and view the results of the Sobel Edge Detection algorithm.

**Note:** The Model Composer blocks can operate on matrices (image frames in the following figure).



One way to assess the simulation performance of the algorithm is to check the video frame rate of the simulation. To do this:

7. Add the Frame Rate Display block from the Simulink Computer Vision System Toolbox (under the Sinks category) and connect it to the output of the algorithm as shown in the following figure.
8. Simulate the model again to see the number of video frames processed per second.



9. Try these things:

- Change the input video through the From Multimedia File block by double-clicking the block and changing the File Name field to select a different video. Notice that changing the video resolution in the Source block does not require any structural modifications to the algorithm itself.

**Note:** You must stop simulation before you can change the input file. Also, the .mp4 files in the MATLAB vision data tool box directory are not supported.

- Build any variations using other available blocks in the `Computer Vision` sub-library in Model Composer.

**Note:** You can find other smaller examples for reference in the folder `ModelComposer_Tutorial\Lab1\Section1\Examples`.

## Step 3: Work with Data Types

In this step, you become familiar with the supported Data Types for Model Composer and conversion from floating to fixed-point types.

This exercise has two primary parts, and one optional part:

- Review a simple floating-point algorithm using Model Composer.
- Look at Data Type Conversions in Model Composer designs.

### Work with Native Simulink Data Types

1. In the MATLAB Current Folder, navigate to the `ModelComposer_Tutorial\Lab1\Section2` folder.

2. Double-click **ColorSpace\_Conversion.slx** to open the design.

This is a Color Space conversion design, built with basic Model Composer blocks, that performs a RGB to YCbCr conversion.

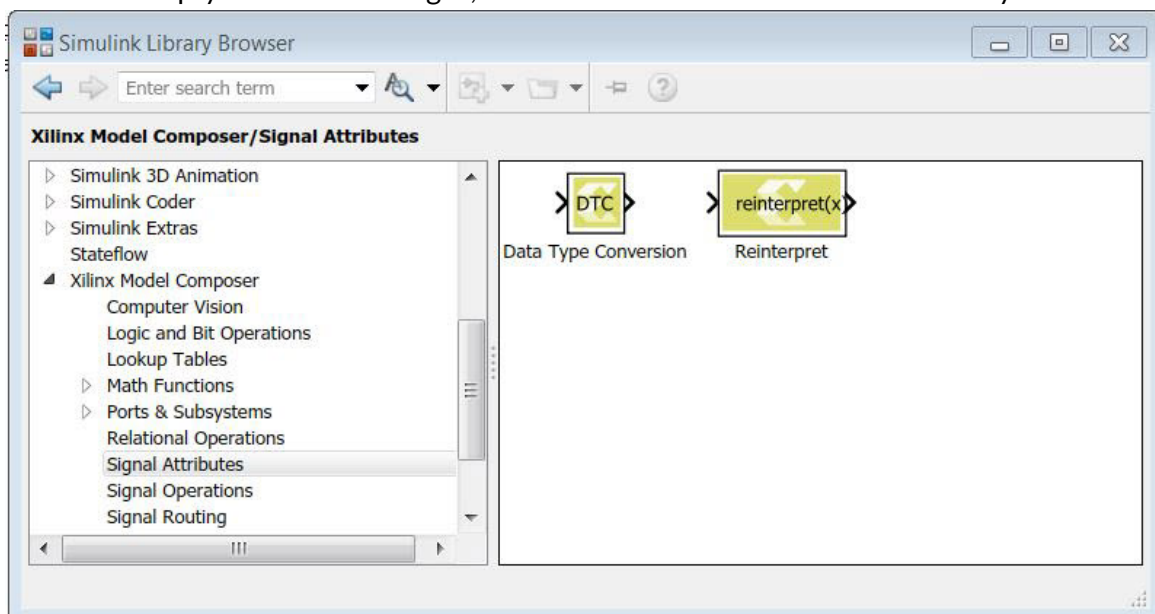
3. Update the model (**Ctrl+D**) and observe that the Data Types, Signal Dimensions and Sample Times from the Source blocks in Simulink all propagate through the Model Composer blocks. Note that the design uses single precision floating point data types.
4. Simulate the model and observe the results from simulation.

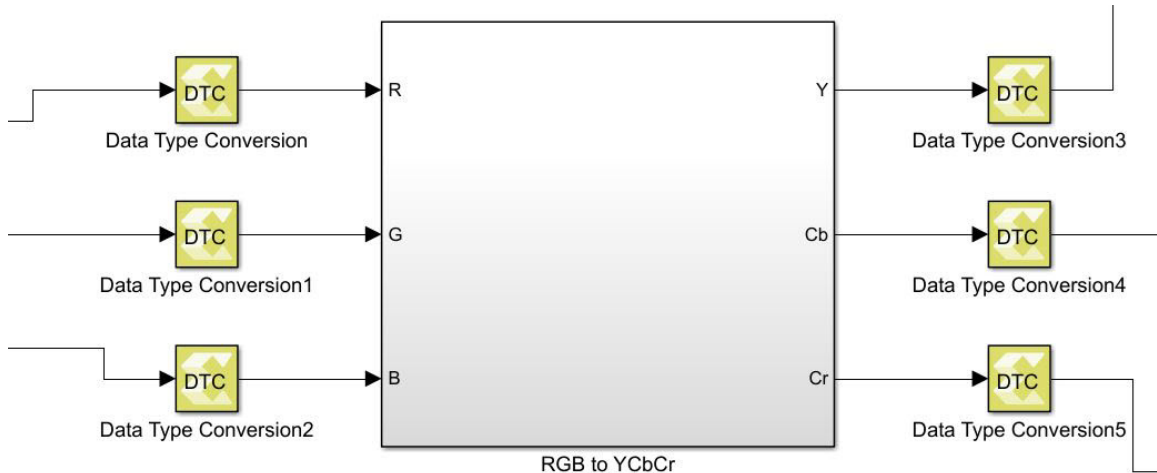
## Convert Data Types

To convert the previous design to use Xilinx Fixed Point types:

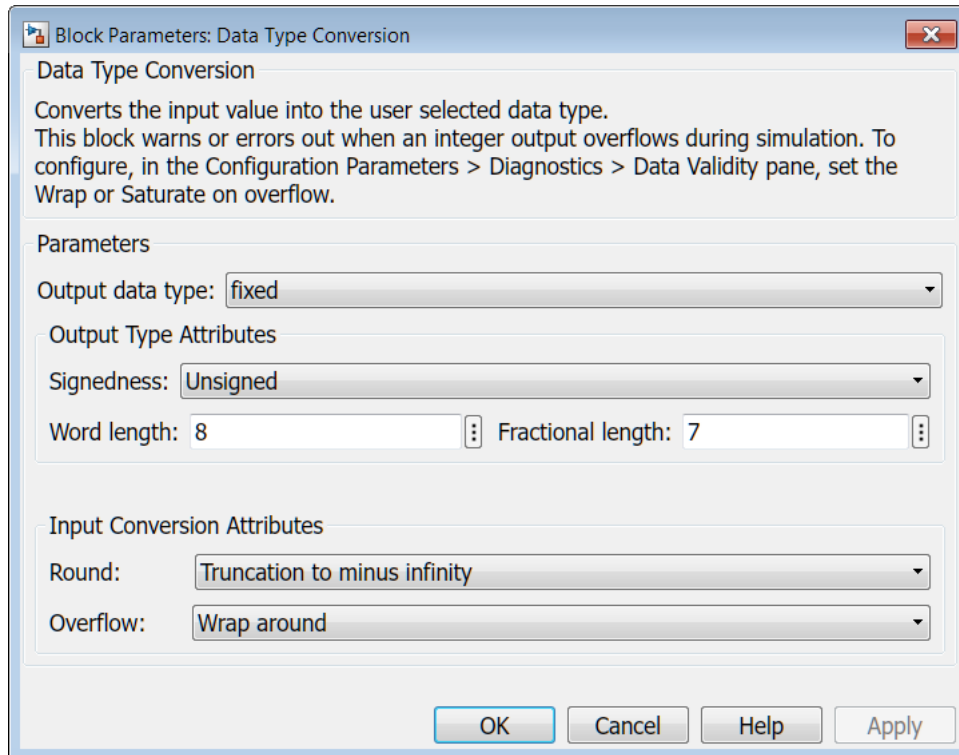
**Note:** Fixed point representation helps to achieve optimal resource usage and performance for a usually acceptable trade-off in precision, depending on the dataset/algorithm.

1. Double-click **ColorSpace\_Conversion\_fixed\_start.slx** in the Current Folder to open the design.
2. Open the **Xilinx Model Composer** library in the Simulink Library Browser.
3. Navigate to the Signal Attributes sub-library, select the **Data Type Conversion** block, and drag it into the empty slots in the designs, before and after the RGB to YCbCr subsystem.

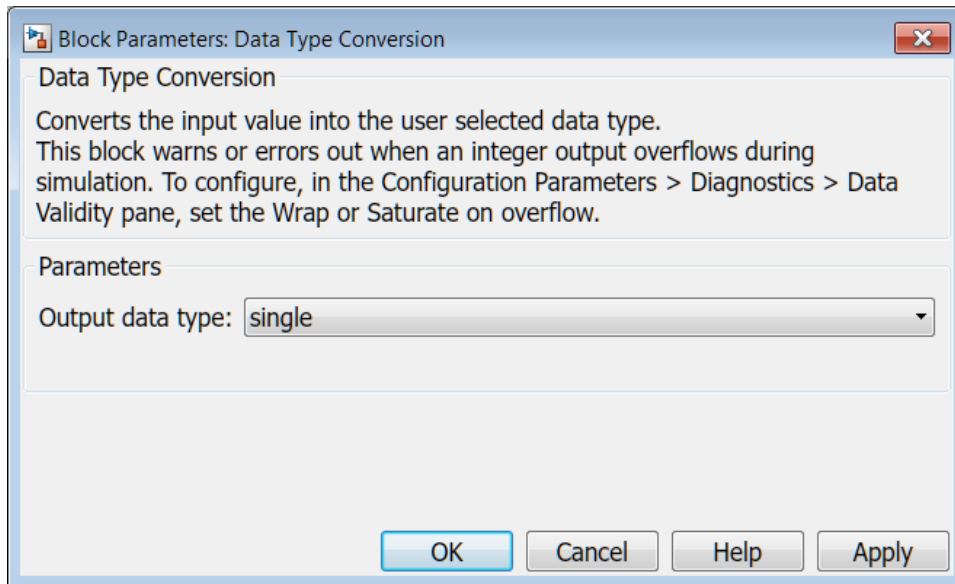




4. Open the Data Type Conversion blocks at the inputs of the RGB to YCbCr Subsystem, and do the following:
  - Change the **Output data type** parameter to **fixed**.
  - Set the **Signedness** to **Unsigned**.
  - Set the **Word length** to **8**.
  - Set **Fractional length** to **7**.
  - Click **Apply**, and close the dialog box.



5. Add the Data Type Conversion blocks at the output of the RGB to YCbCr Subsystem and set the **Output data type** parameter to **single**. This will enable connecting the output signals to the Video Viewer blocks for visualization.



6. Double-click the **RGB to YCbCr** subsystem to descend the hierarchy and open the model. Within the RGB to YCbCr subsystem, there are subsystems to calculate Y, Cb, and Cr components using Gain and Constant blocks.

You can control the fixed point types for the gain parameter in the Gain blocks and the value in the Constant blocks. You can do this by opening up the **Calculate\_Y**, **Calculate\_Cb**, and **Calculate\_Cr** blocks and setting the data types as follows.

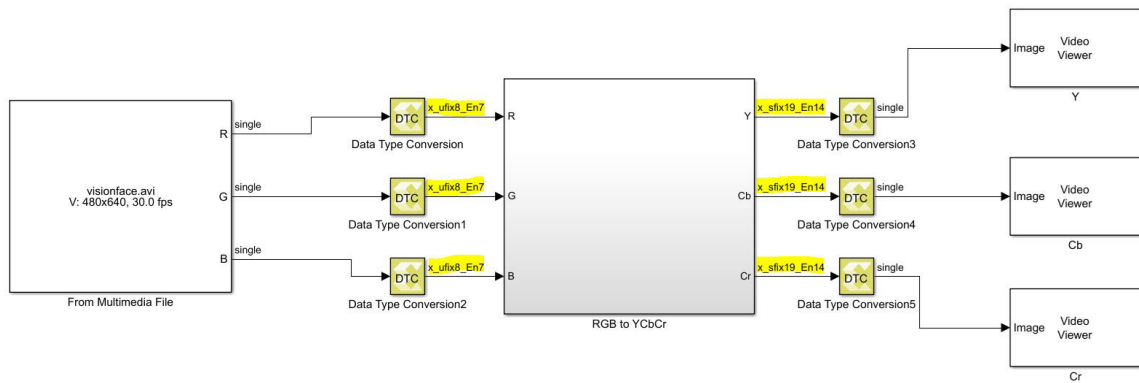
For Gain blocks, set the Gain data type to fixed. For Constant blocks, on the Data Types tab set the Output data type to fixed. The following options appear:

- **Signedness to Signed**
- **Gain data type to fixed**
- **Word length to 8**
- **Fractional length to 7**



**TIP:** You can use the **View → Property Inspector** command to open the Property Inspector window. When you select the different Gain or Constant blocks, you can see and modify the properties on the selected block.

Ensure you do this for all the Constant and Gain blocks in the design. Update the model (Ctrl +D) and observe the fixed point data types being propagated along with automatic bit growth in gain blocks and adder trees in the design as shown in the following figure:



The general format used to display the Xilinx fixed point data types is as follows:

`x_[u/s]fix[wl]_En[fl]`

- **u:** Unsigned
- **s:** Signed
- **wl:** Word Length
- **fl:** Fractional Length

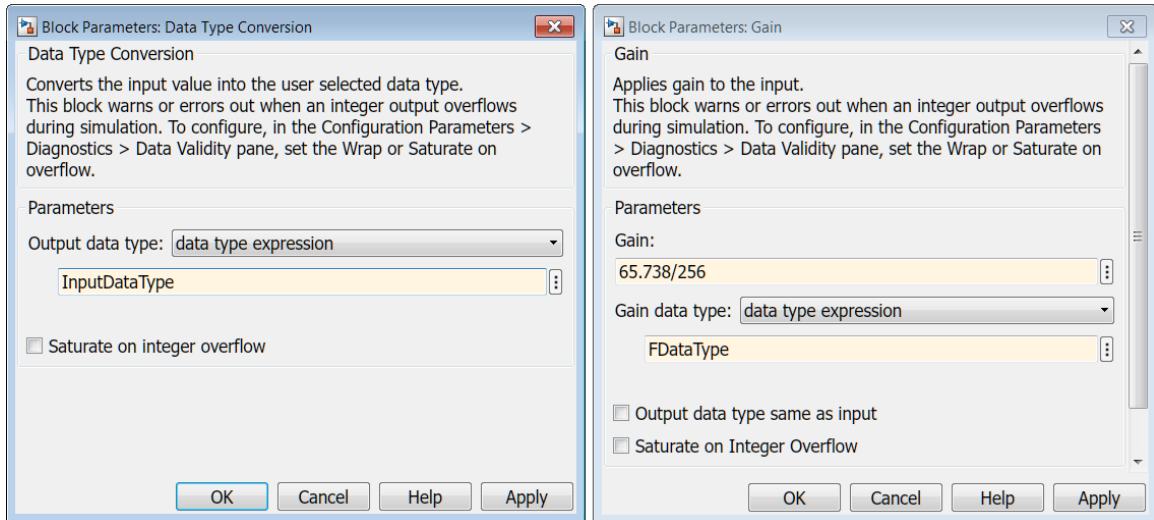
For example, `x_sfix16_En8` represents a signed fixed point number with Word Length=16 and Fractional Length=8.

You can view a completed version of the design here: `ModelComposer_Tutorial\Lab1\Section2\solution\Colorspace_Conversion_fixed.slx`

## Convert Data Types (Alternative)

Model Composer supports Data Type Expressions that make it easier to change data types and quickly explore the results from your design.

1. Double-click `ColorSpace_Conversion_Expression.slx` in the Current Folder to open the design.
2. Notice that the Data Type Conversion blocks at the Input of the RGB to YCbCr Subsystem, the Gain blocks and Constant blocks within the Subsystem have Output data type and Gain data type set to data type expression.



This enables Model Composer blocks to control the data types in the design using workspace variables, in this case `InputDataType` and `FDataType` that you can easily change from the MATLAB command prompt.

3. Update the model (**Ctrl+D**) and observe the fixed-point data types propagated through the blocks.

The other Model Composer blocks in the design will automatically take care of the bit-growth in the design. If you want more control over the fixed point data types at other intermediate portions of the design, you can insert Data Type Conversion blocks wherever necessary.

4. To change the fixed point types in the Gain, Constant, and DTC blocks, and Input data type in DTC blocks, type the following at the MATLAB command prompt:

```
>> FDataType = 'x_sfix8_En6'
>> InputDataType = 'x_ufix8_En6'
```

'`x_sfix8_En6`' represents a signed fixed point number with Word Length 8 and Fractional Length 6.

Now update the model (**Ctrl+D**) and observe how the fixed-point data types have changed in the design.

5. Simulate the model and observe the results from the design. Try further changing `InputDataType` and `FDataType` variables through command line and iterate through multiple word lengths and fractional lengths. See the Additional Details section below for information on specifying rounding and overflow modes.

## Additional Details

In the example above, we only specified the Word Length and Fractional Length of the fixed point data types using data type expressions. However, for greater control over the fixed point types in your design, you can also specify the Signedness, Rounding, and Overflow. In general the format used for specifying fixed point data types using the data type expression is

```
x_[u/s]fix[wl]_En[fl]_[r<round>w<overflow>]
```

- **u:** Unsigned
- **s:** Signed
- **wl:** Word length
- **fl:** Fractional length

**<round>:** Specify the corresponding index from the following table. This is optional. If not specified, the default value is 6 (Truncation to minus infinity). Note that for the rounding cases (1 to 5), the data is rounded to the nearest value that can be represented in the format. When there is a need for a tie breaker, these particular roundings behave as specified in the Meaning column.

**Table 1: Rounding Index**

Index	Meaning
1	Round to Plus Infinity
2	Round to Zero
3	Round to Minus Infinity
4	Round to Infinity
5	Convergent Rounding
6	Truncation to Minus Infinity
7	Truncation to Zero

**<overflow>:** Specify the corresponding index from table below. It's optional. If not specified, default value is 4 (Wrap around).

**Table 2: Overflow Index**

Index	Meaning
1	Saturation
2	Saturation to Zero
3	Symmetrical Saturation
4	Wrap Around
5	Sign-Magnitude Wrap Around

Example: `x_ufix8_En6_r6w4` represents a fixed point data type with:

- **Signedness:** Unsigned
- **Word Length:** 8
- **Fractional Length:** 6
- **Rounding Mode:** Truncation to Minus Infinity
- **Overflow Mode:** Wrap Around

---

## Conclusion

In this lab, you learned:

- How to connect Model Composer blocks directly to native Simulink blocks.
- How the Model Composer blocks support Vectors and Matrices, allowing you to process an entire frame of an image at a time without converting it from a frame to a stream of pixels at the input.
- How to work with different data types.
- How to use the Data Type Conversion block to control the conversion between data types, including floating-point to fixed-point data types.

**Note:** Model Composer Supports the same floating and integer data types as Simulink blocks. Model Composer also supports Xilinx fixed point data types.

The following solution directories contain the final Model Composer files for this lab:

- `C:\ModelComposer_Tutorial\Lab1\Section1\solution`
- `C:\ModelComposer_Tutorial\Lab1\Section2\solution`

## Lab 2

# Importing Code into Model Composer

Model Composer lets you import Vivado® HLS library functions and user C/C++ code as custom blocks to use in your algorithm for both simulation and code generation.

The Library Import feature is a MATLAB function, `xmcImportFunction`, which lets you specify the required source files and automatically creates an associated block that can be added into a model in Simulink®.

This lab primarily has two parts:

- In Step 1, you are introduced to the `xmcImportFunction` function, and walk through an example.
- In Step 2, you will learn about the Model Composer feature that enables you to create custom blocks with function templates

For more details and information about other Model Composer features, see the *Model Composer User Guide* ([UG1262](#)).

---

## Step 1: Set up the Import Function Example

In the MATLAB Current Folder panel, navigate to `Lab2\Section1` folder.

1. Double-click the `basic_array.cpp` and `basic_array.h` files to view the source code in the MATLAB Editor.

These are the source files for a simple `basic_array` function in C++, which calculates the sum of two arrays of size 4. You will import this function as a Model Composer block using the `xmcImportFunction` function.

The input and output ports for the generated block are determined by the signature of the source function. Model Composer identifies arguments specified with the `const` qualifier as inputs to the block, and all other arguments as outputs.

**Note:** For more details and other options for specifying the direction of the arguments, see the *Model Composer User Guide* ([UG1262](#)).




---

**IMPORTANT!** You can use the `const` qualifier in the function signature to identify the inputs to the block or use the pragma `IMPORT`.

---

In the case of the `basic_array` function, the `in1` and `in2` arguments are identified as inputs.

```
void basic_array(
    uint8_t out1[4],
    const uint8_t in1[4],
    const uint8_t in2[4])
```

- To learn how to use the `xmcImportFunction` function, type `help xmcImportFunction` at the MATLAB command prompt to view the help text and understand the function signature.
- Open the `import_function.m` MATLAB script, and fill in the required fields for the `xmcImportFunction` function in this way:

```
xmcImportFunction('basic_array_library', {'basic_array'},
    'basic_array.h', {'basic_array.cpp'}, {});
```

The information is defined as follows:

- Library Name:** `basic_array_library`. This is the name of the Simulink library that is created with the new block.
- Function Names:** `basic_array`. This is the name of the function that you want to import as a block.
- Header File:** `basic_array.h`. This is the header file for the function.
- Source Files:** `basic_array.cpp`. This is the source file for the imported function.
- Search Paths:** This argument is used to specify the search path(s) for header files. In this example, there are no additional search paths to specify and hence you can leave it as `{}` which indicates none.

**Note:** Look at `import_function_solution.m` in the solution folder for the completed version.

- Run the `import_function.m` script from the MATLAB command line:

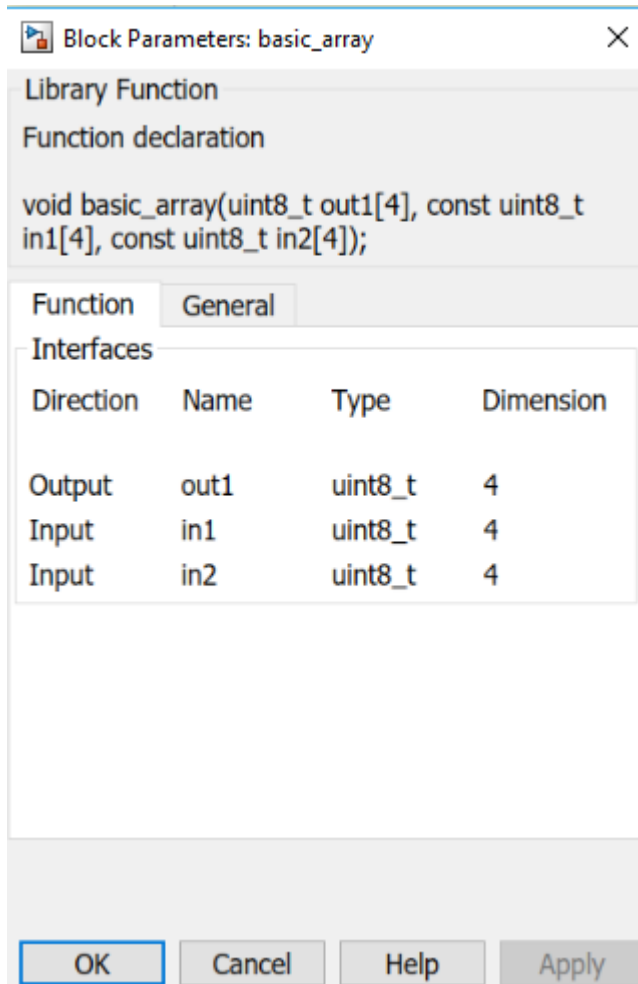
```
>>run('import_function.m')
```

Notice that a Simulink library model opens up with the generated block `basic_array`.

Save this Simulink library model.

- Double-click the **basic\_array** block, and look at the generated interface.

The following figure shows the Block Parameters dialog box for `basic_array`:



6. Open the `test_array.slx` model, which is just a skeleton to test the generated block.
7. Add the generated `basic_array` block into this model, then connect the source and sink blocks.
8. Simulate this model and observe the results in the display block.

---

## Step 2: Custom Blocks with Function Templates

In this step we will walk through an example to do the following:

- To create a custom block that supports inputs of different sizes.
- To create a custom block that accepts signals with different fixed-point lengths and fractional lengths.
- To perform simple arithmetic operations using template variables.

1. Navigate to the Lab2/section2 folder.
2. Double-click the **template\_design.h** file to view the source code in the MATLAB Editor. There are two functions: Demux and Mux. These two functions are a multiplexing and demultiplexing of inputs as shown in the following figure.

```
#pragma XMC INPORT vector_in
template<int NUMOFELEMENTS, int W, int I>
void Demux(ap_fixed<W,I> vector_in[NUMOFELEMENTS], ap_fixed<W,I> vector_out0[NUMOFELEMENTS/2],
           ap_fixed<W,I> vector_out1[NUMOFELEMENTS/2]) {
    for (int i = 0; i < NUMOFELEMENTS/2; i++) {
        vector_out0[i] = vector_in[i];
        vector_out1[i] = vector_in[i+NUMOFELEMENTS/2];
    }
}
```

3. In the piece of code, note the `#pragma XMC INPORT vector_in`. This is a way to manually specify port directions using pragmas. Here, we are specifying the function argument `vector_in` as the input port. Similarly, we can define `XMC OUTPORT` also.

**Note:** For additional information about specifying ports, see *Model Composer User Guide (UG1262)*.

4. Notice the use of template before the function declaration. To support the inputs of different sizes, `NUMOFELEMENTS` is declared as a parameter and used the same while defining an array `vector_in` as shown in the following figure. This allows you to connect signals of different sizes to the input port of the block.

```
template<int NUMOFELEMENTS, int W, int I>
void Demux(ap_fixed<W,I> vector_in[NUMOFELEMENTS], ap_fixed<W,I> vector_out0[NUMOFELEMENTS/2],
           ap_fixed<W,I> vector_out1[NUMOFELEMENTS/2]) {
```

5. Notice the template parameters `W` and `I` which are declared to accept signals with different word lengths and integer lengths.

```
template<int NUMOFELEMENTS, int W, int I>
void Demux(ap_fixed<W,I> vector_in[NUMOFELEMENTS], ap_fixed<W,I> vector_out0[NUMOFELEMENTS/2],
           ap_fixed<W,I> vector_out1[NUMOFELEMENTS/2]) {
```

**Note:** The same library is specified for both the functions.

6. Observe the arithmetic operations performed using template variables as shown below, indicating the output signal length is half of the input signal length.

```
void Demux(ap_fixed<W,I> vector_in[NUMOFELEMENTS], ap_fixed<W,I> vector_out0[NUMOFELEMENTS/2],
           ap_fixed<W,I> vector_out1[NUMOFELEMENTS/2]) {
```

7. Similar explanation follows for Mux function.

```
#pragma XMC INPORT vector_in0
#pragma XMC INPORT vector_in1
template<int NUMOFELEMENTS, int W, int I>
void Mux(ap_fixed<W,I> vector_in0[NUMOFELEMENTS], ap_fixed<W,I> vector_in1[NUMOFELEMENTS],
         ap_fixed<W,I> vector_out [NUMOFELEMENTS*2]) {
    for (int i = 0; i < NUMOFELEMENTS; i++) {
        vector_out[i] = vector_in0[i];
        vector_out[i+NUMOFELEMENTS] = vector_in1[i];
    }
}
```

Now create the library blocks for Mux and Demux functions using the `xmcImportFunction` command and complete the design below with custom blocks.



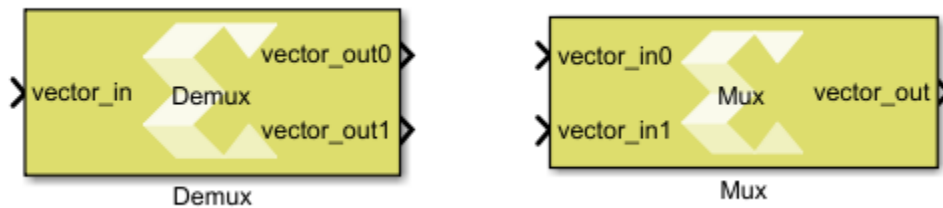
- Double-click the `import_function.m` script file in the MATLAB command window and observe the following commands that generate library blocks to embed into your actual design.

```
>>xmcImportFunction('design_lib',{'Demux'],'template_design.h',{},{},
{'$XILINX_VIVADO_HLS/include'},'override','unlock')
>>xmcImportFunction('design_lib',{'Mux'],'template_design.h',{},{},
{'$XILINX_VIVADO_HLS/include'},'override','unlock')
```

- Run the `import_function.m` script from the MATLAB command line:

```
>>run('import_function.m')
```

- Observe the generated library blocks in the `design_lib.slx` library model file and save it to working directory.



- Copy the Demux and Mux blocks and paste them in the `design.slx` file and connect them as shown in the following figure.



- Note the following after embedding the custom blocks:

- Double-click the Constant block and observe the vector input of type double. SSR is a workspace variable, initially set to 8 from the `initFcn` model callback.
- Using the Data Type Conversion (DTC) block, double type is converted to fixed type with 16-bit word length and 8-bit fractional length.

Input is configurable to any word length since the design is templated.

- Double-click the Demux block and observe the Template parameters section and Dimension column in the Interface section of the function tab.

Block Parameters: Demux

Library Function

Function declaration

```
template<int NUMOFELEMENTS, int W, int I> void
Demux(ap_fixed<W, I> vector_in[NUMOFELEMENTS],
ap_fixed<W, I> vector_out0[NUMOFELEMENTS / 2], ap_fixed<W,
I> vector_out1[NUMOFELEMENTS / 2]);
```

Function General

Template parameters

Name	Type
NUMOFELEMENTS	int
W	int
I	int

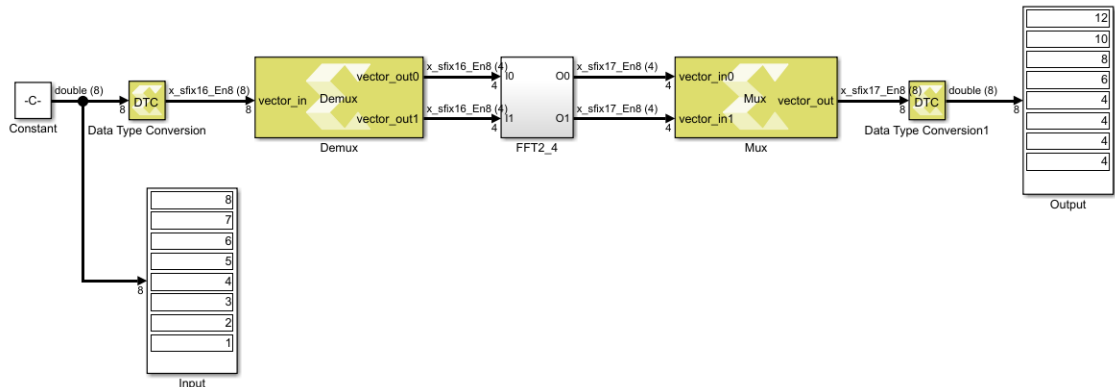
Interfaces

Direction	Name	Type	Dimension
Input	vector_in	ap_fixed<W, I>	NUMOFELEMENTS
Output	vector_out0	ap_fixed<W, I>	NUMOFELEMENTS / 2
Output	vector_out1	ap_fixed<W, I>	NUMOFELEMENTS / 2

OK Cancel Help Apply

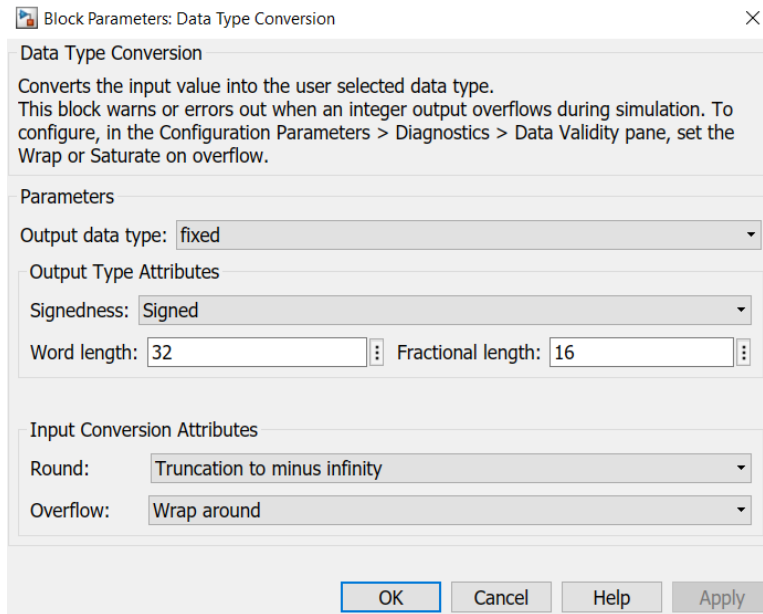
d. Next, double-click the Mux block and observe the Template parameters and Dimension.

13. Add a Display block at the input and output as shown in the following figure and simulate the model to observe the results.

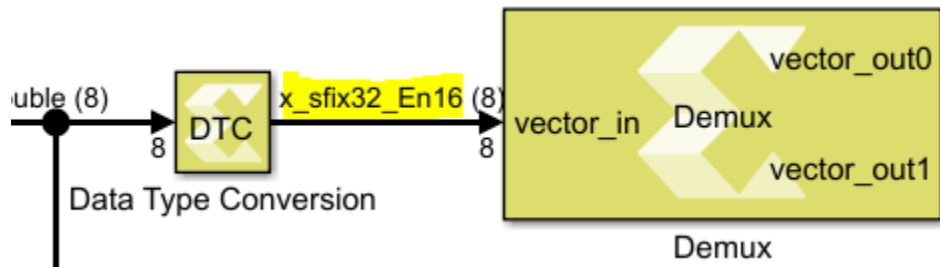


14. To understand how templated inputs add advantage and flexibility to your design, perform the following:

- a. Double-click the **DTC** block.
- b. In the Block Parameters dialog box, change the Word length from 16 to 32.
- c. Change the Fractional length from 8 to 16.



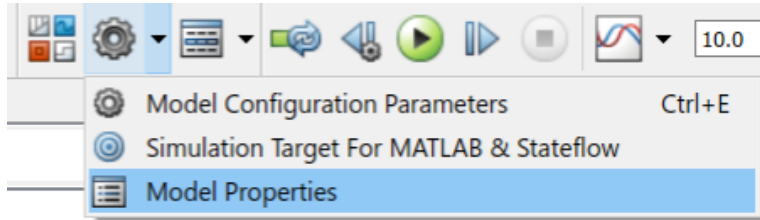
- d. Click **OK** and press **Ctrl+D**. Observe the signal dimensions in the design.



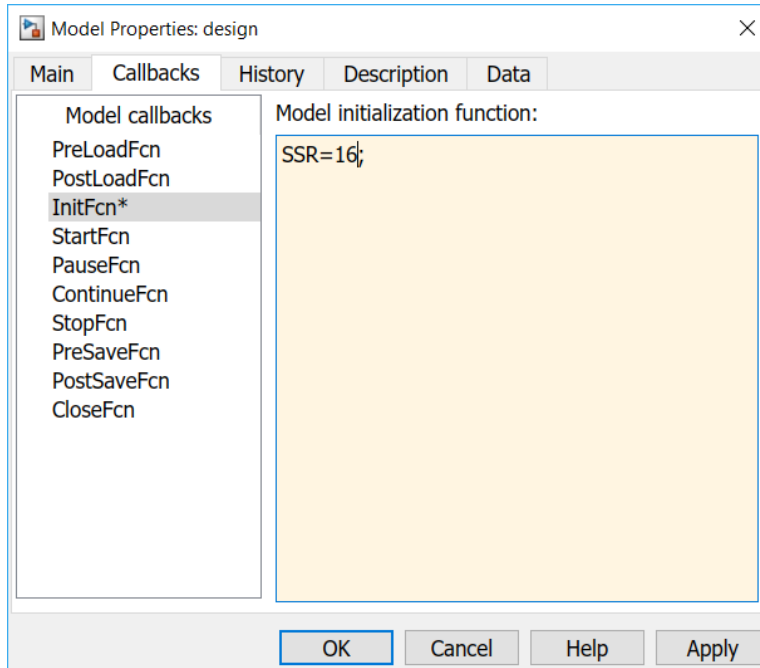
To make sure the output is correct, run the simulation and observe that the same block can still be used in a generic way for different values of Word length and Fractional length. This is possible only because we have templated the  $W$  and  $I$  values in our  $C$  design.

15. For an additional understanding of template parameters, perform the following:

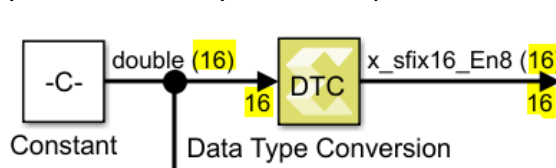
- a. Click the arrow mark beside the Model Configuration Parameters icon and select the **Model Properties** option.



- b. In the Model Properties window, go to the **Callbacks** tab and select **initFcn** and edit the SSR value from 8 to 16 as shown in the following figure.



- c. Click **OK** and press **Ctrl+D** to observe the change in the number of elements in the Constant block output vector. The bitwidth changes when we change the datatype on the input DTC. This is possible only because of the template parameter `NUMOFELEMENTS`.



- d. Run the simulation and validate the output according to the input values.

**Note:** For information about features such as function templates for data types and pragmas to specify which data type a template variable supports, see *Model Composer User Guide* ([UG1262](#)).

## Conclusion

In this lab, you learned:

- How to create a custom block using the `xmlImportFunction` in Model Composer.
- How to create a block that accepts signals with different fixed-point lengths and fractional lengths.
- How to use the syntax for using a function template that lets you create a block that accepts a variable signal size or data dimensions.
- How to perform simple arithmetic operations using template variables.

**Note:** Current feature support enables you to import code that uses:

- Vectors and 2D matrices
- Floating, integer, and Vivado® HLS fixed-point data types

The following solution directory contains the final Model Composer (\*.slx) files for this lab.

- C:\ModelComposer\_Tutorial\Lab2\section1\solution
- C:\ModelComposer\_Tutorial\Lab2\section2\solution

# Debugging Imported C/C++-Code Using GDB Debugger

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®.

The debug flow in Model Composer is as follows:

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.

This lab has two steps:

- Step 1 introduces you to the Optical Flow demo design example in Model Composer. It shows you how to identify the custom library block, created using the `xmcImportFunction` feature.
- Step 2 shows you how to debug C/C++ code using the GDB tool.

For more details and information about how to create custom blocks, follow this [link](#) in *Model Composer User Guide (UG1262)*.

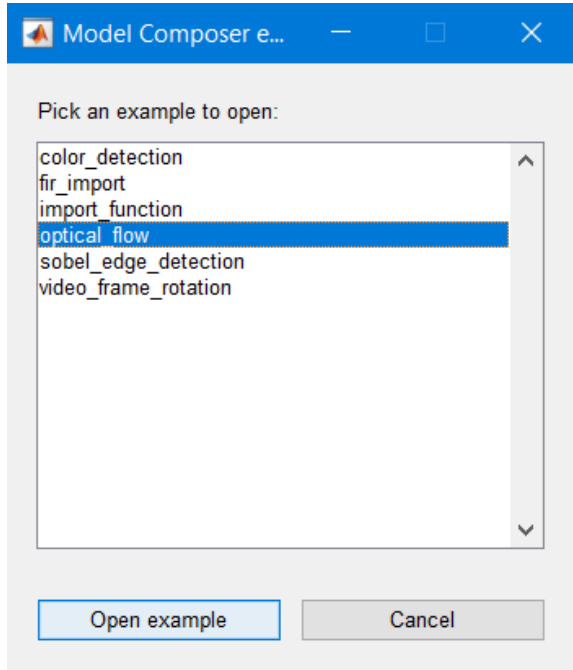
---

## Step 1: Set Up the Example to Debug the Import Function

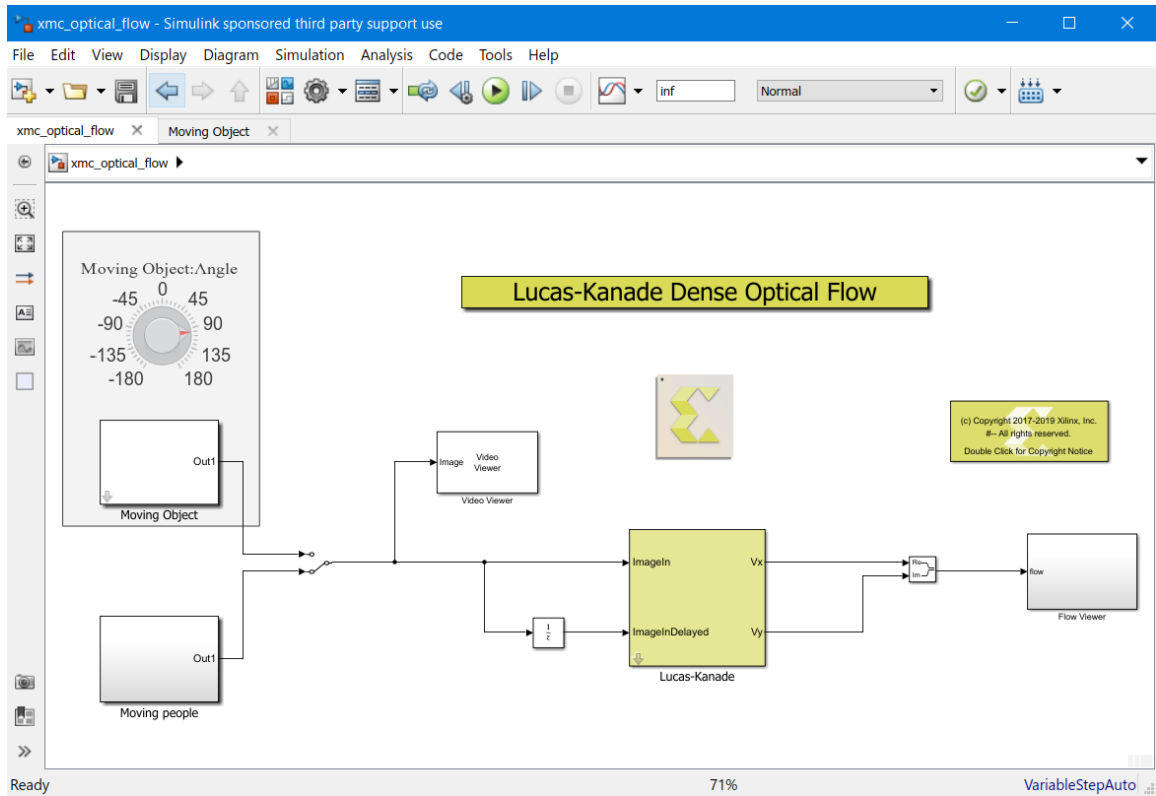
1. Type the following at the MATLAB command prompt:

```
>> xmcOpenExample
```

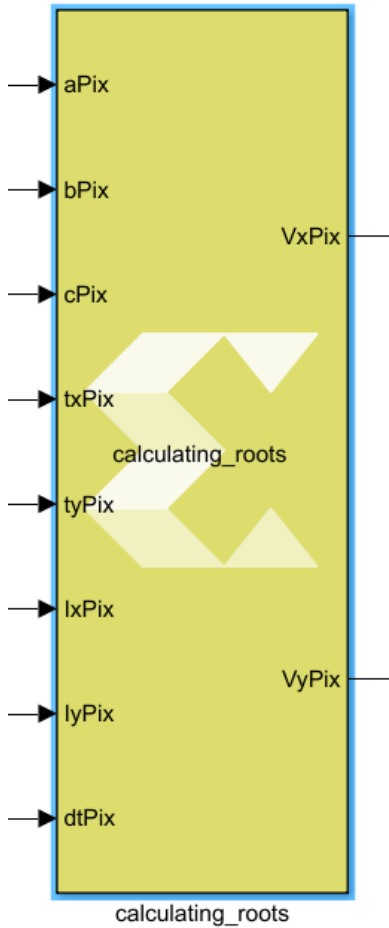
2. Press **Enter** to open the Model Composer examples dialog box.



3. In the Model Composer examples dialog box select **optical flow** and click **Open example**. This opens the example design.



4. Double click on the block labeled **Lucas-Kanade** and observe the `calculating_roots` block.

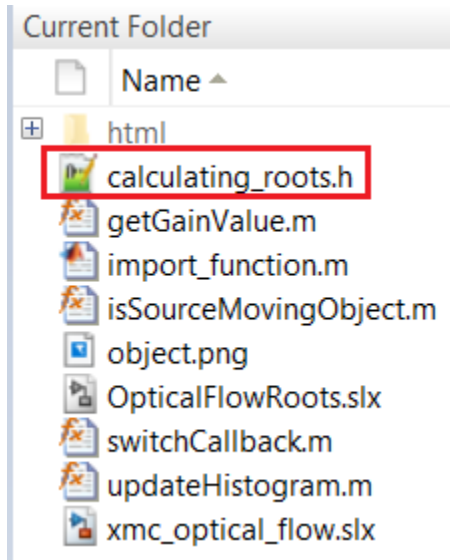


**Note:** This block has been generated using the `xmcImportFunction` feature. Its function declaration can be seen by double-clicking on the block.

```

Block Parameters: calculating_roots
Import Function
Function declaration
void calculating_roots(int16_t aPix, int16_t bPix, int16_t cPix,
int16_t txPix, int16_t tyPix, int16_t lxPix, int16_t lyPix, int16_t
dtPix, float & VxPix, float & VyPix);
    
```

- To view the function definition of `calculating_roots`, navigate to the current folder in the MATLAB window and double-click on `calculating_roots.h`.



The setup is now ready for you to debug your C/C++ code. In the next step, you will see how to debug the code using GDB tool debugger.

## Step 2: Debugging C/C++ Code Using GDB Debugger

1. Specify the debug tool using the `xmcImportFunctionSettings` command. At the MATLAB® command prompt, type the following command:

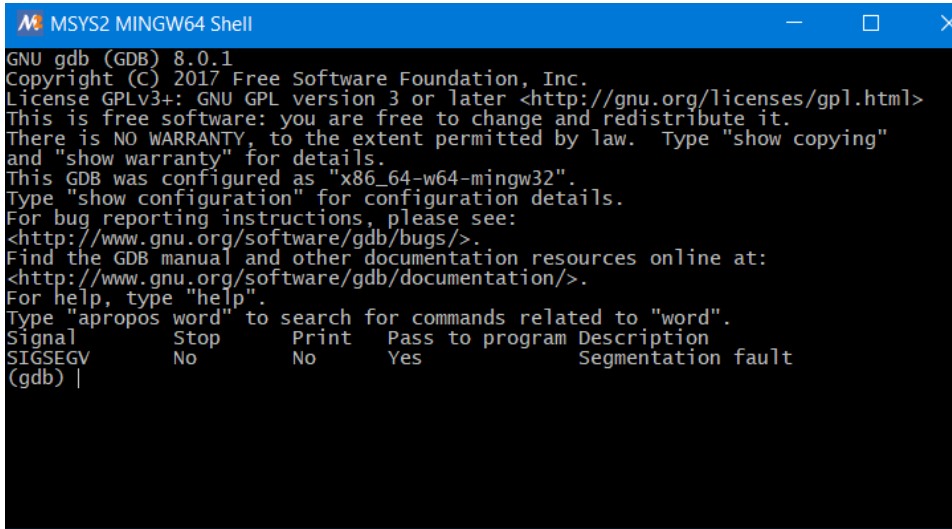
```
>> xmcImportFunctionSettings('build', 'debug');
```

2. Press **Enter** to see the applied settings in command window, as shown in the following figure.

```
>> xmcImportFunctionSettings('build','debug');
Current settings:
    'build' = 'debug'
    'compiler' = 'default'
Imported C/C++ code will be built with MingW compiler.
You can use gdb to debug your C/C++ code.
MATLAB process ID is 15972.
You can also get the process ID by typing "feature getpid" in the MATLAB command window.
```

Note the `gdb` link that you will use to invoke the debugger tool, and the MATLAB process ID that you will use to attach the process to the debugger.

3. Click on the `gdb` link, to invoke the Windows command prompt and launch `gdb`.



```

MSYS2 MINGW64 Shell
GNU gdb (GDB) 8.0.1
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Signal      Stop       Print      Pass to program  Description
SIGSEGV     No         No         Yes              Segmentation fault
(gdb) |
    
```

- At the Windows command prompt, use the following command to specify the breakpoint in the `calculating_roots.h` file where you want the code to stop executing. Press **Enter** to run the command.

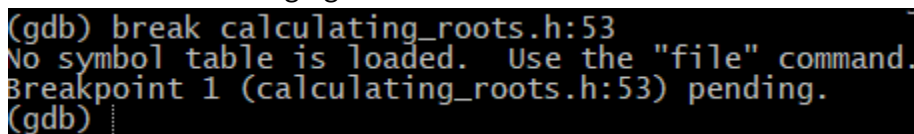
```
(gdb) break calculating_roots.h:53
```

**Note:** The “53” in the above command, tells the GDB debugger to stop the simulation at line 53 of your program.

```

50     int16_t r = aPix + cPix;
51     float s = hls::sqrtf(4 * bPix * bPix + (aPix - cPix) * (aPix - cPix));
52
53     int16_t eig1 = (r + s);
54     int16_t eig2 = (r - s);
    
```

- Once the command runs, you can see a pending breakpoint in the command window. This is shown in the following figure.



```

(gdb) break calculating_roots.h:53
No symbol table is loaded.  Use the "file" command.
Breakpoint 1 (calculating_roots.h:53) pending.
(gdb) |
    
```

If you see any questions from GDB, answer “yes” and press **Enter**.

- To attach the MATLAB process to the GDB debugger, type the following:

```
(gdb) attach <process_ID>
```

Enter the `<process ID>` you saw in step 2. For example “15972”.

As soon as the MATLAB process is attached, the MATLAB application gets frozen and becomes unresponsive.

```

MSYS2 MINGW64 Shell
[New Thread 15972.0x18e8]
[New Thread 15972.0x57e4]
[New Thread 15972.0x3120]
[New Thread 15972.0x3fcc]
[New Thread 15972.0x3f2c]
[New Thread 15972.0x2eb8]
[New Thread 15972.0x54bc]
[New Thread 15972.0xb04]
[New Thread 15972.0x4c80]
[New Thread 15972.0x149c]
[New Thread 15972.0x5a90]
[New Thread 15972.0x43e8]
[New Thread 15972.0x5c08]
[New Thread 15972.0x2a3c]
[New Thread 15972.0x4f90]
[New Thread 15972.0x3cb0]
[New Thread 15972.0x5be8]
[New Thread 15972.0x4b14]
[New Thread 15972.0x51d0]
[New Thread 15972.0x6798]
[New Thread 15972.0x3698]
Reading symbols from C:\Program Files\MATLAB\R2018a\bin\win64\MATLAB.exe...(no debugging symbols found)...done.
(gdb)

```

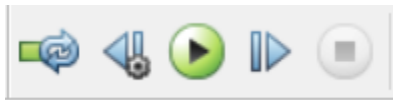
7. Type `cont` at the Windows command prompt.

```

MSYS2 MINGW64 Shell
[New Thread 15972.0x3cb0]
[New Thread 15972.0x5be8]
[New Thread 15972.0x4b14]
[New Thread 15972.0x51d0]
[New Thread 15972.0x6798]
[New Thread 15972.0x3698]
Reading symbols from C:\Program Files\MATLAB\R2018a\bin\win64\MATLAB.exe...(no debugging symbols found)...done.
(gdb) cont
Continuing.
[Thread 15972.0x3698 exited with code 0]
[New Thread 15972.0x666c]
[New Thread 15972.0x66a8]
[New Thread 15972.0x471c]
[New Thread 15972.0x337c]
[New Thread 15972.0x5c50]
[New Thread 15972.0x440c]
[New Thread 15972.0x6750]
[New Thread 15972.0x29dc]
[New Thread 15972.0x5788]
[New Thread 15972.0x2b98]
[New Thread 15972.0x3b44]
[New Thread 15972.0x5f30]

```

8. Now go to the Simulink® model and run the simulation by clicking the **Run** button.



9. The model takes some time to initialize. As the simulation starts, you see the simulation come to the breakpoint at line 53 in the Windows command prompt.

```

MSYS2 MINGW64 Shell
[New Thread 15972.0x308]
[New Thread 15972.0x50e0]
[Thread 15972.0x308 exited with code 0]
[Thread 15972.0x50e0 exited with code 0]
[New Thread 15972.0x56a8]
[Thread 15972.0x56a8 exited with code 0]
[New Thread 15972.0x6470]
[Thread 15972.0x6470 exited with code 0]
[Switching to Thread 15972.0x3098]

Thread 1 hit Breakpoint 1, calculating_roots (aPix=210, bPix=81, cPix=211,
txPix=<optimized out>, tyPix=tyPix@entry=346, IxPix=IxPix@entry=31,
IyPix=IyPix@entry=30, dtPix=dtPix@entry=39,
VxPix=@0x20e650490: 2.5743929e-036, VyPix=@0x20e6775a0: 2.5743929e-036)
at calculating_roots.h:53
53         int16_t eig1 = (r + s);
    
```

Now, type the command `list` to view the lines of code around line 53.

```
(gdb) list
```

10. Now, type command `step` to continue the simulation one line to the next step.

```
(gdb) step
```



**IMPORTANT!** *The following are some useful GDB commands for use in debugging:*

- (gdb) list
- (gdb) next (step over)
- (gdb) step (step in)
- (gdb) print <variable>
- (gdb) watch <variable>

11. Type `print r` to view the values of variables at that simulation step. This gives the result as shown in the following figure.

```
(gdb) print r
$1 = 534
```

12. You can try using more gdb commands to debug and once you are done, type `quit` to exit GDB, and observe that the Simulink model continues to run.

## Conclusion

In this lab, you learned:

- How to specify a third party debugger and control the debug mode using `xmcImportFunctionSettings`.
- How to debug source code associated with your custom blocks using the GDB debugger, while leveraging the stimulus vectors from Simulink.

# Debugging Imported C/C++-Code Using Visual Studio

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®.

The debug flow in Model Composer is as follows:

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.

This lab has two steps:

- Step 1 introduces you to the Color Detection design example in Model Composer, and shows you how to identify the custom library block created by the `xmcImportFunction` feature.
- Step 2 shows you the process used to debug the C/C++ code using Visual Studio.

For more details and information about how to create custom blocks, follow this [link](#) in *Model Composer User Guide (UG1262)*.

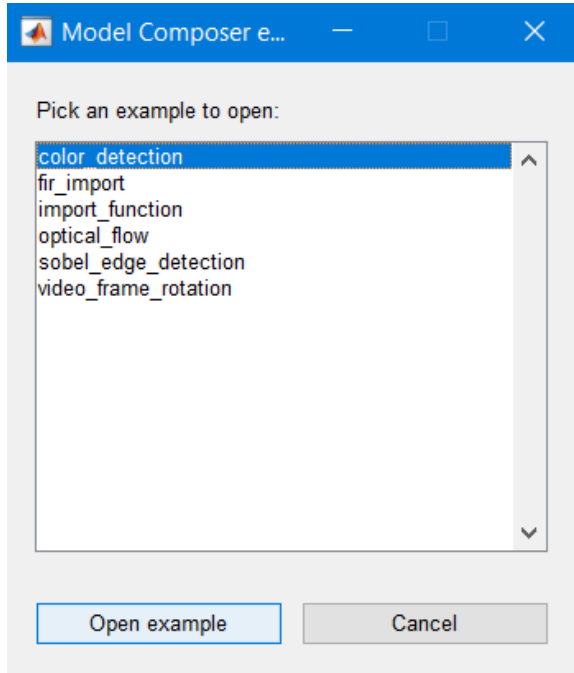
---

## Step 1: Set Up the Example to Debug the Import Function

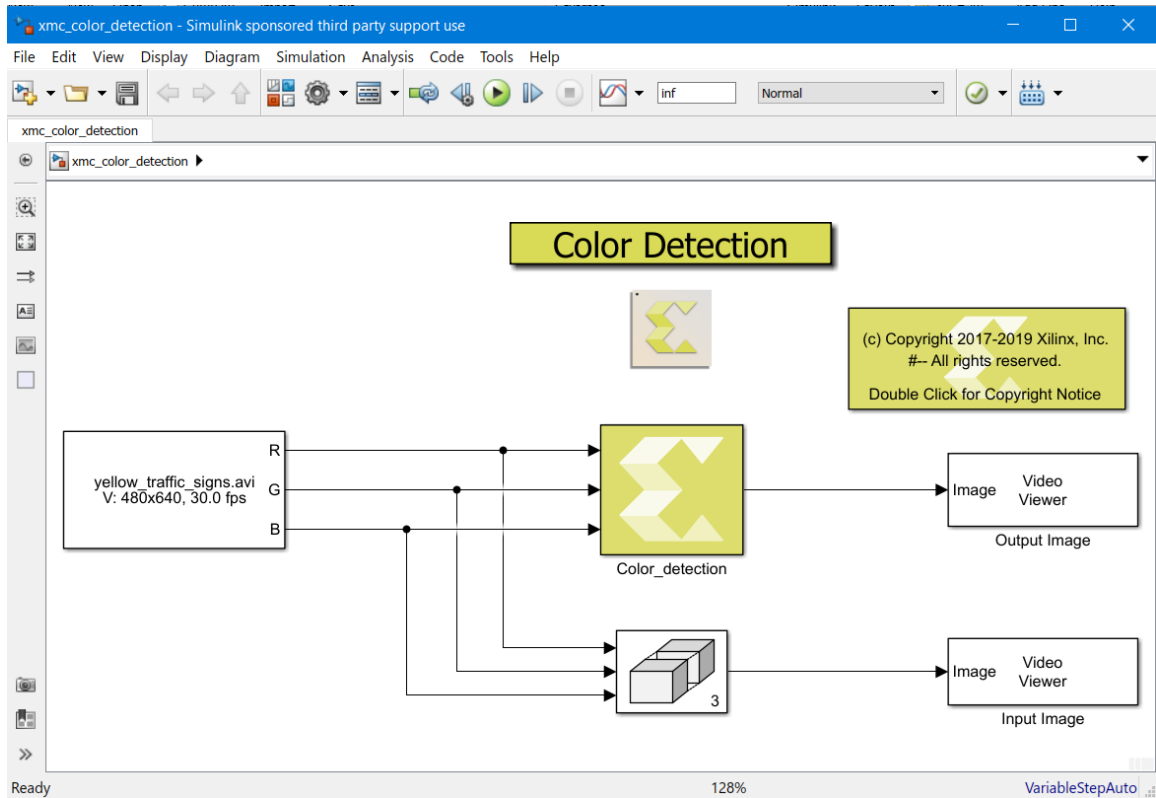
1. Type the following at the MATLAB® command prompt:

```
>> xmcOpenExample
```

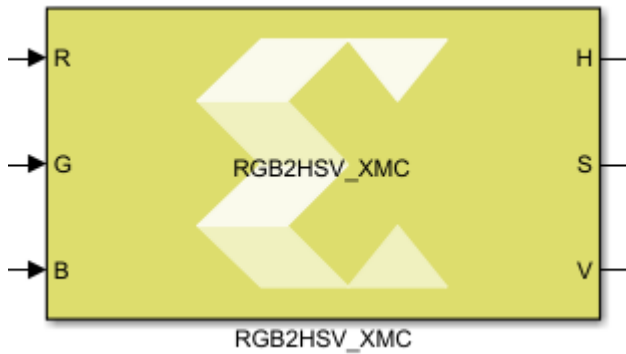
2. Press **Enter** to open the Model Composer examples dialog box:



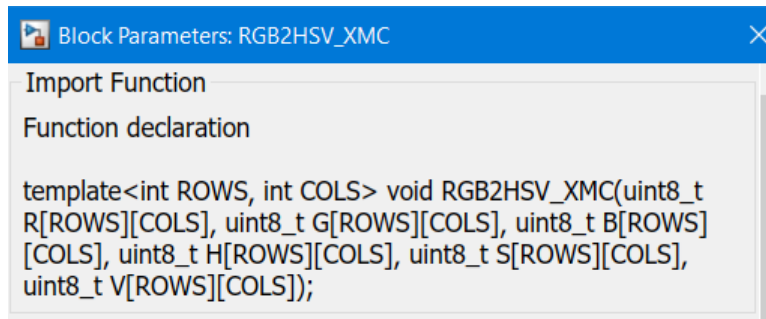
- From the above list, select **color\_detection** and click **Open example**, which opens the example design as shown in the following figure.



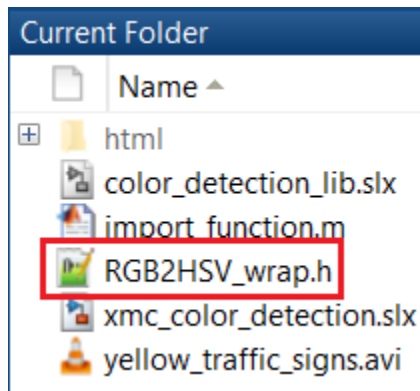
- Double click on the block labeled **Color\_detection** and observe the **RGB2HSV\_XMC** block.



**Note:** This block has been generated using the `xmcImportFunction` feature and the function declaration can be seen by double-clicking on the `RGB2HSV_XMC` block.



- To view the function definition of `RGB2HSV_XMC`, navigate to current folder in the MATLAB window and double click on `RGB2HSV_wrap.h`.



The setup is now ready for you to debug your C/C++ code. In next step, you will see how to debug the code using Visual Studio debugger.

## Step 2: Debugging C/C++ Code Using Visual Studio

1. Specify the debug tool using the `xmcImportFunctionSettings` command. In the MATLAB command prompt, type the following:

```
>> xmcImportFunctionSettings('build','debug','compiler','Visual Studio');
```

Press **Enter**.

This command picks the installed Visual Studio in your machine for debugging.

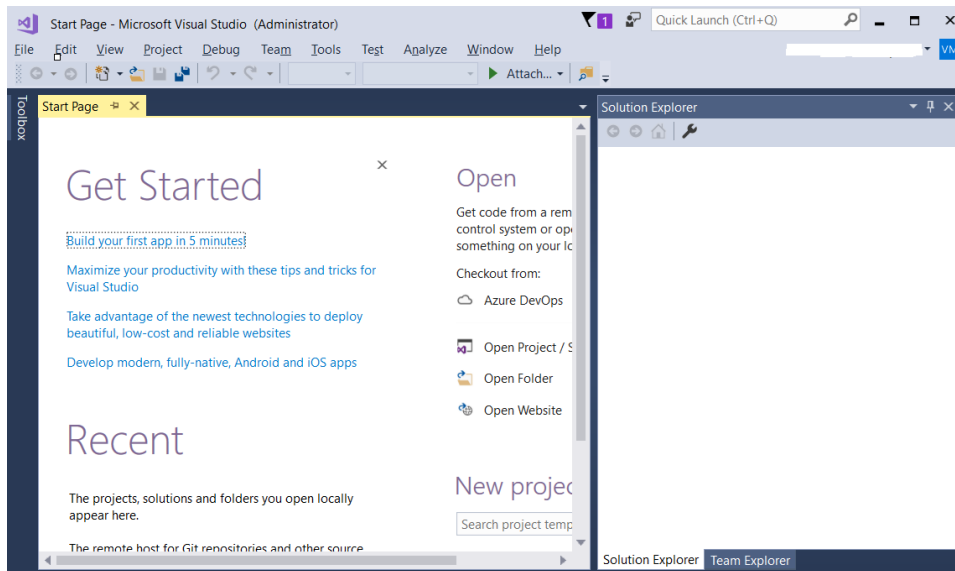
2. Observe the version and location of Visual Studio from the MATLAB console.

```
>> xmcImportFunctionSettings('build','debug','compiler','Visual Studio');
Current settings:
'build' = 'debug'
'compiler' = 'Visual Studio'
'Visual Studio location' = 'C:\Program Files (x86)\Microsoft Visual Studio\2017\Community'
'vcvars' = 'C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvarsal
Imported C/C++ code will be built with Visual C++ compiler found at 'C:\Program Files (x86)\Microsoft Vi
```

3. Type the following in the MATLAB console to get more information on this command and also to set different version of Visual studio.

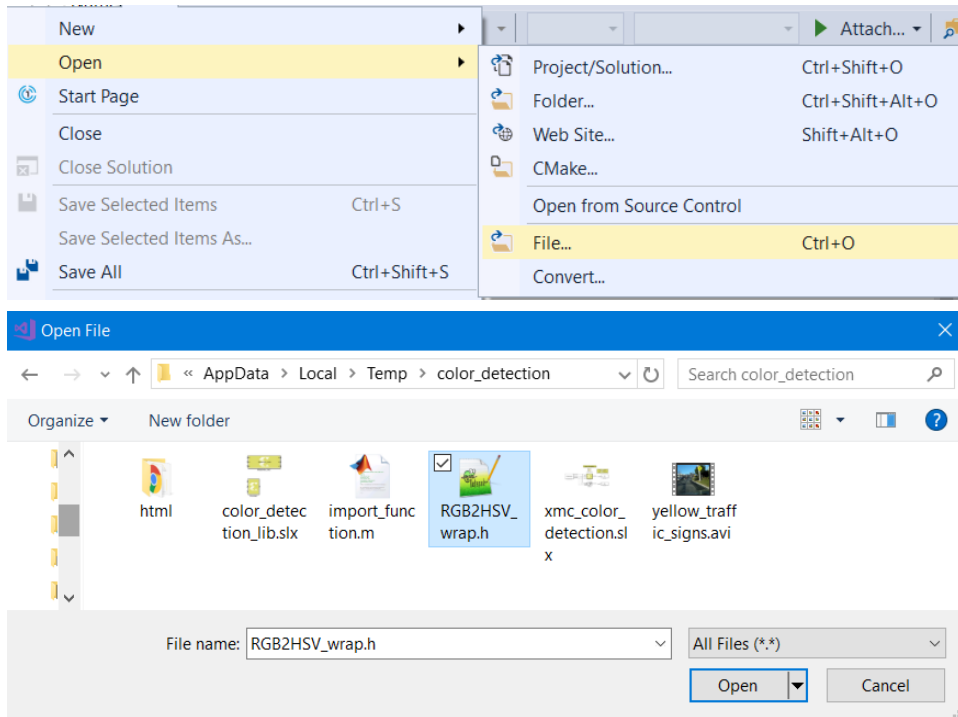
```
>>help xmcImportFunctionSettings
```

4. Invoke Visual Studio from your install directory to start debugging C/C++ code.



5. In the Visual Studio startup page, open the C/C++ file that you want to debug.

Click on **File** → **open** → **File** and browse to the location where the Color Detection example design resides. Select the file **RGB2HSV\_wrap.h** and click **Open**.



6. Observe that `RGB2HSV_wrap.h` opens in the Visual Studio as shown in the following figure.

```

1  #include "imgproc/xf_bgr2hsv.hpp"
2  #include "common/xf_common.h"
3
4  #include <stdint.h>
5  #include <ap_int.h>
6
7  #pragma XMC INPORT R,G,B
8  #pragma XMC OUTPORT H,S,V
9  #pragma XMC SUPPORTS_STREAMING
10
11  template<int ROWS,int COLS>
12  void RGB2HSV_XMC(
13      uint8_t R[ROWS][COLS],
14      uint8_t G[ROWS][COLS],
15      uint8_t B[ROWS][COLS],
16      uint8_t H[ROWS][COLS],
17      uint8_t S[ROWS][COLS],
18      uint8_t V[ROWS][COLS])
19  {
20      ap_uint<24> data_in;
21      ap_uint<24> data_out;
22
23      xf::Mat<XF_BUC3,ROWS,COLS,XF_NPPC1> src_obj;
    
```

7. Now set the break point in the `RGB2HSV_wrap.h`.

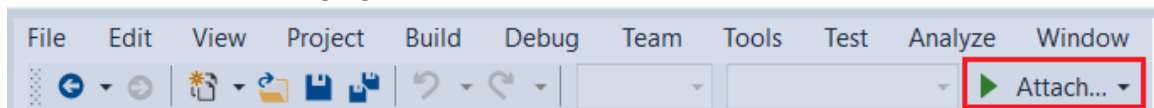
To set the break point, click on the grey color space at line number. A red dot appears there, as shown in the following figure. This indicates that the break point has been set at the corresponding line.

```

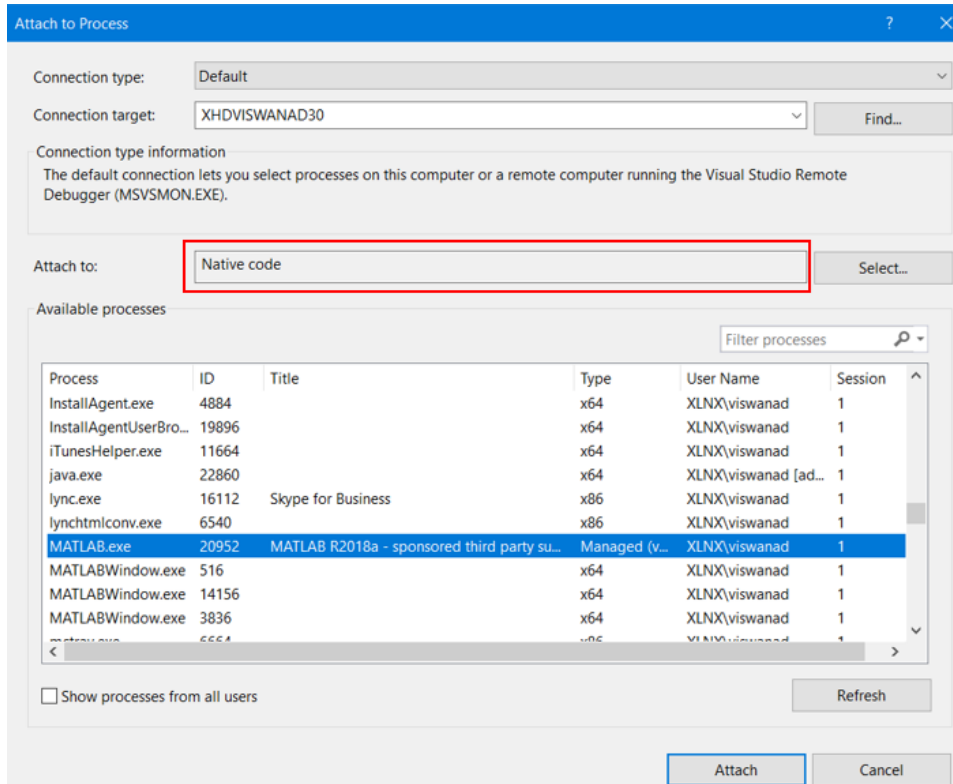
1  #include "imgproc/xf_bgr2hsv.hpp"
2  #include "common/xf_common.h"
3
4  #include <stdint.h>
5  #include <ap_int.h>
6
7  #pragma XMC IMPORT R,G,B
8  #pragma XMC OUTPUT H,S,V
9  #pragma XMC SUPPORTS_STREAMING
10
11 template<int ROWS,int COLS> <T>
12 void RGB2HSV_XMC(
13     uint8_t R[ROWS][COLS],
14     uint8_t G[ROWS][COLS],
15     uint8_t B[ROWS][COLS],
16     uint8_t H[ROWS][COLS],
17     uint8_t S[ROWS][COLS],
18     uint8_t V[ROWS][COLS])
19 {
20     ap_uint<24> data_in;
21     ap_uint<24> data_out;
22
23     xf::Mat<XF_8UC3,ROWS,COLS,XF_NPPC1> src_obj;

```

8. Next, attach the MATLAB process to the debugger by clicking the **Attach** icon in the tool bar as shown in the following figure.



9. In the Attach to Process dialog box, which is opened, search for the process MATLAB .exe.



When attaching Visual Studio to the MATLAB process, you must make sure to set the Attach to field to choose Native Code from the drop down menu as highlighted above.

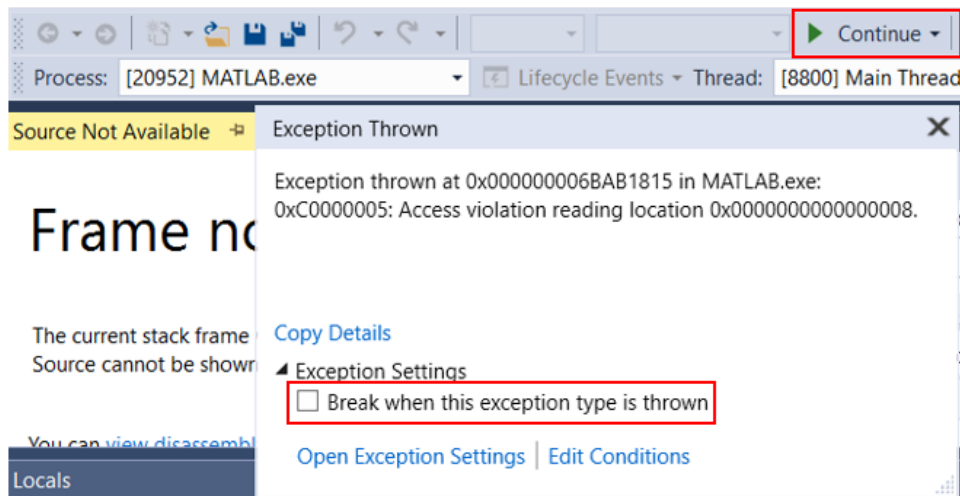
Click on **Attach** button. Your MATLAB process is attached to the debugger.

- Now, go back to the Simulink model and start simulating the `color_detection` design.



- The simulation process may take some time to initialize. Once this is done, switch back to the Visual Studio GUI to start debugging.

If any Exception is thrown by Visual Studio, simply uncheck **Break when this exception type is thrown** in the Exception settings and click **Continue**.

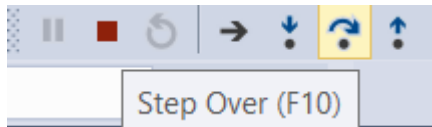


- You can now see the simulation hitting the break point in Visual Studio.

```

9  #pragma XMC OUTPORT H,S,V
10 #pragma XMC SUPPORTS_STREAMING
11
12 template<int ROWS,int COLS>
13 void RGB2HSV_XMC(
14     uint8_t R[ROWS][COLS],
15     uint8_t G[ROWS][COLS],
16     uint8_t B[ROWS][COLS],
17     uint8_t H[ROWS][COLS],
18     uint8_t S[ROWS][COLS],
19     uint8_t V[ROWS][COLS])
20 {
21     ap_uint<24> data_in;
22     ap_uint<24> data_out;
23
24     xf::Mat<XF_8UC3,ROWS,COLS,XF_NPPC1> src_obj;
25     #pragma HLS stream variable=src_obj dim=2 depth=1
26
27     xf::Mat<XF_8UC3,ROWS,COLS,XF_NPPC1> dst_obj;
28     #pragma HLS stream variable=dst_obj dim=2 depth=1
    
```

13. To debug, you can use the **Step Over** icon in the tool bar.

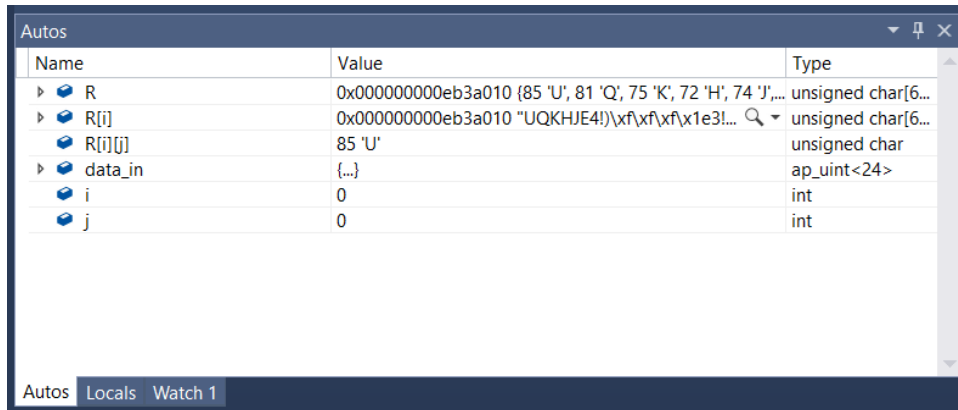


**IMPORTANT!**

Click the **Step Into** icon to execute the next line of the code. If the line involves a call to an operation, it steps into the operation's implementation and breaks the execution on the first action of that implementation.

Clicking the **Step Over** icon does not step into the line's implementation details, but steps to the next line of code after the call.

14. Observe the values of variables at each corresponding step, as you progress with debugging.



15. You can change the breakpoint to a different line, by removing the initial break point, and setting a new one for example at Line 22 and clicking the **Continue** button from tool bar.

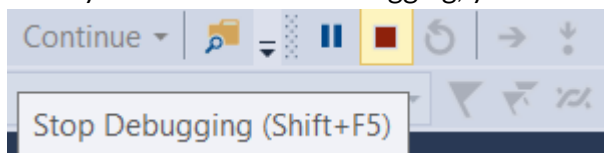
Now you can observe the break point hitting Line 22.

16. Now, remove all the break points and click the **Continue** button. You can see the model running.

**Note:** As the design is running in debug mode, the simulation may progress slowly.

**Note:** You can always set a break point as long, as the design is simulating.

17. Once you are done with debugging, you can the click **Stop Debugging** button in the tool bar.



18. To come out of debugging mode in MATLAB, type the following command in the MATLAB console and press **Enter**.

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

Now, you can run the imported C/C++ code in release mode.

## Conclusion

In this lab, you learned:

- How to specify a third party debugger and control the debug mode using `xmcImportFunctionSettings`.
- How to debug source code associated with your custom blocks using Microsoft® Visual Studio, while leveraging the stimulus vectors from Simulink.

# Automatic Code Generation

In this lab, you look at the flow for generating output from your Model Composer model and moving it into downstream tools like Vivado® HLS for RTL synthesis, or into System Generator, or the Vivado® Design Suite for implementation into a Xilinx device.

---

## Procedure

This lab has five steps:

- In Step 1, you will review the requirements for automatic code generation.
- In Step 2, you will look at how to map Interfaces in your design.
- In Step 3, you will look at the flow for generating an IP from your Model Composer design.
- In Step 4, you will look at the flow for generating HLS Synthesizable C++ code from the Model Composer design.
- In Step 5, you will look at the flow to port a Model Composer design back into System Generator for DSP as a block.

---

## Step 1: Review Requirements for Generating Code

In this step, you review the three requirements to move from your algorithm in Simulink to an implementation through automatic code generation.

1. In the MATLAB Current Folder, navigate to the `ModelComposer_Tutorial\Lab5` directory.
2. Double-click **CodeGen\_start.slx** to open the model.

To prepare for code generation, you will enclose your Model Composer design in a subsystem.

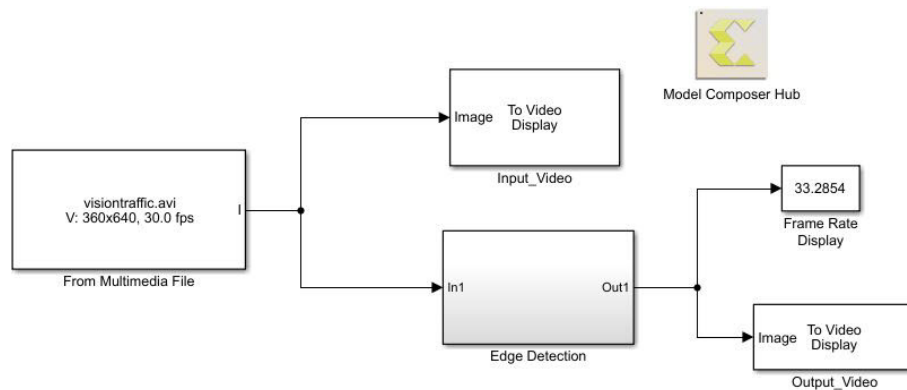
3. Right-click the **Edge Detection** area, and select **Create Subsystem from Area**.

**Note:** For code generation to work, all the blocks within the enclosed subsystem should only be from the Xilinx Model Composer library, with the exception of the Simulink blocks noted below. Subsystems with unsupported blocks will generate errors during code generation. The Simulink diagnostic viewer will contain error messages and links to the unsupported blocks in the subsystem.

**Note:** In addition to the base Model Composer blocks, a subset of native Simulink blocks such as From, Goto, Bus Creator, Bus Selector, If, and others, are supported. The supported Simulink blocks appear within the Xilinx Model Composer libraries as well.

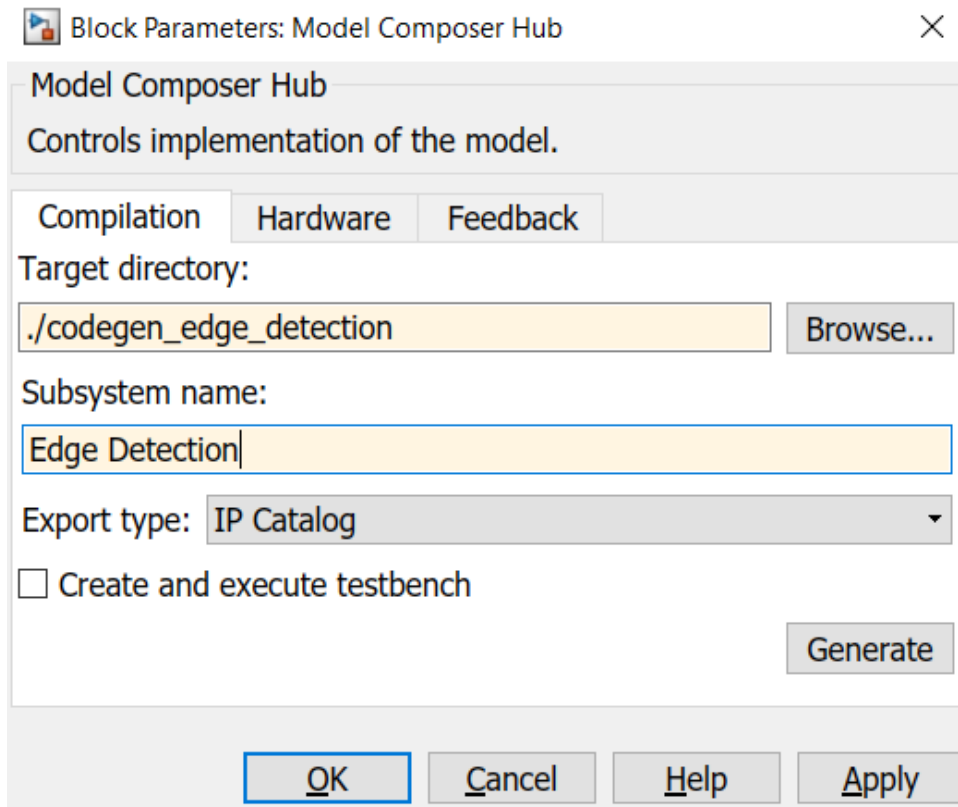
Next, you add the Model Composer Hub block at the top level of your design.

4. Open the Simulink Library Browser and navigate to **Xilinx Model Composer** → **Tools** sub-library.
5. Find the Model Composer Hub block, and add it into the design as shown in the following figure.



Next, you use the Model Composer Hub block to select the code generation options for the design.

6. Double-click the block to open the block interface and set up as shown in the following figure.



7. On the Compilation tab, you can set the following options as shown in the previous figure:
  - **Target directory:** In this case, use `./codegen_edge_detection` for the generating code.
  - **Subsystem name:** In this case, use the Edge Detection subsystem. You can have multiple subsystems at the top-level and use the Model Composer Hub block to select and individually compile the subsystem you want.
  - **Export Type:** This option determines what you want to convert your design into. In this case **IP Catalog** (default). You can select other compilation targets from drop down.
    - Vivado HLS Synthesizable C++ code
    - System Generator for DSP
8. On the Hardware tab, you can specify the target FPGA clock frequency in MHz. The default value is 200 MHz.

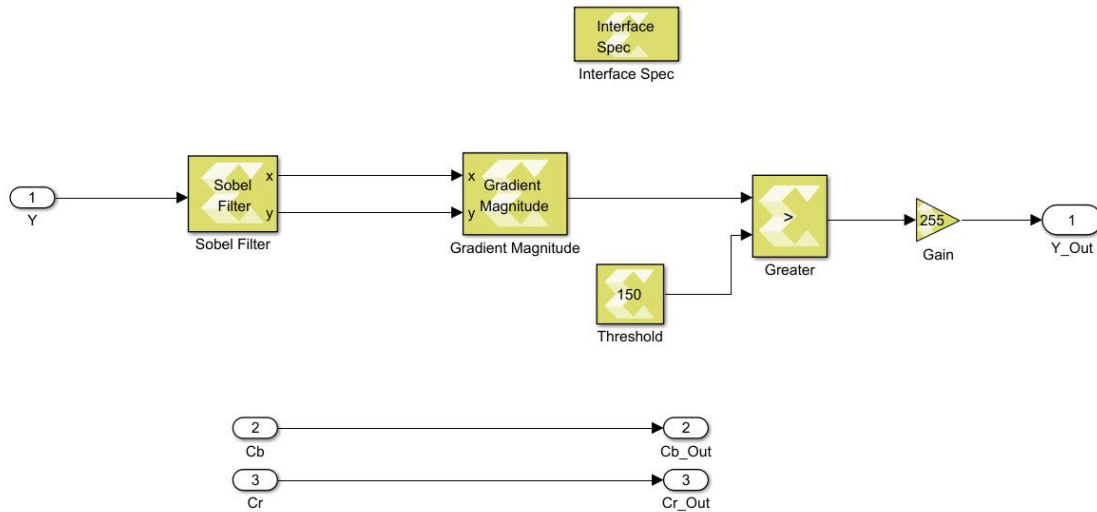
---

## Step 2: Mapping Interfaces

1. Double-click the `CodeGen_Interface.slx` model in your Current Folder to open the design for this lab section.

This is a slightly modified version of the Edge Detection algorithm that uses the YCbCr video format at the input and output.

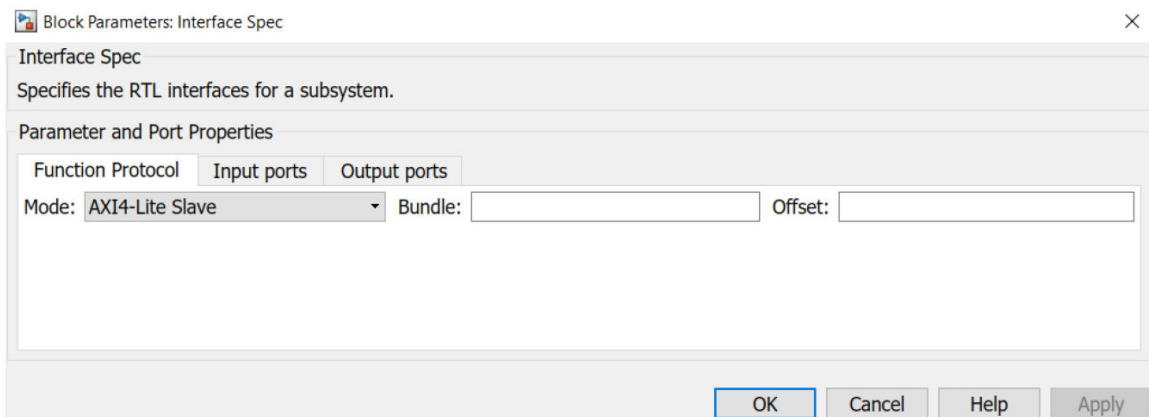
2. Simulate the model to see the results in the Video Viewer blocks.
3. Open the Simulink Library browser, navigate to the **Xilinx Model Composer** → **Tools** sub-library and add the Interface Spec block inside the Edge Detection subsystem as shown in the following figure.



4. Double-click the **Interface Spec** block to open the block interface.

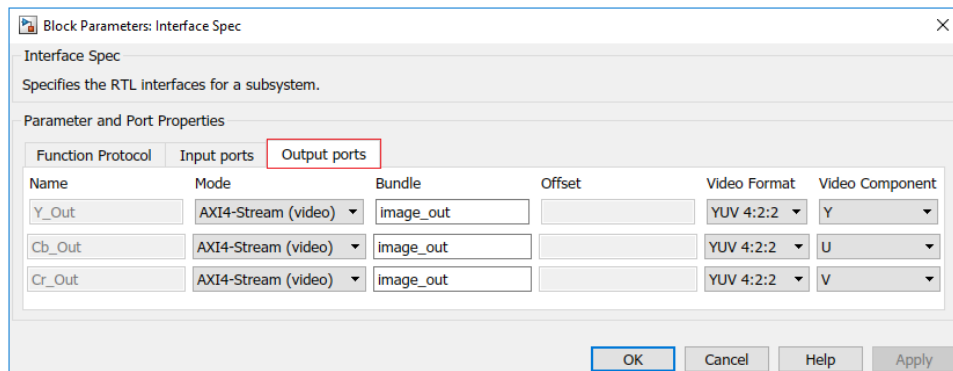
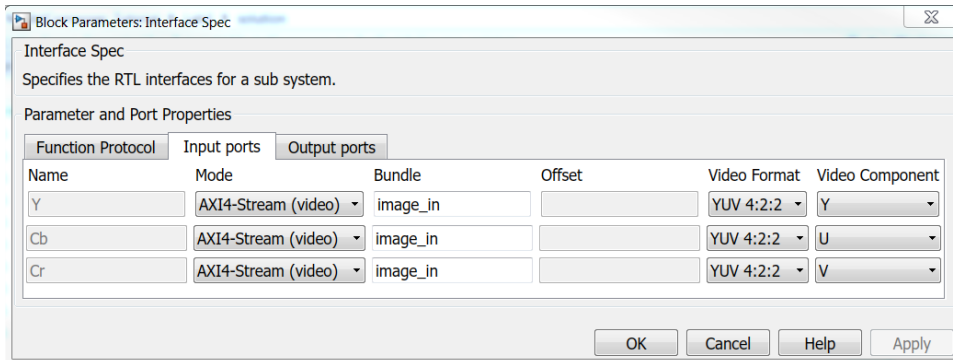
The Interface Spec block allows you to control what RTL interfaces should be synthesized for the ports of the subsystem in which the block is instantiated. This affects only code generation; it has no effect on Simulink simulation of your design.

The information gathered by the Interface Spec block consists of three parts (represented as three Tabs on the block).



- **Function Protocol:** This is the block-level Interface Protocol which tells 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:** Detects the Input ports in your subsystem automatically and allows specifying the port-level Interface Protocol for each input port of the subsystem.
  - **Output Ports:** Similar to the Input Ports tab, this tab detects the Output ports in the subsystem, and allows specifying the port-level Interface Protocol for each output port of the subsystem.
5. For this design, leave the Function Protocol mode at the default AXI4-Lite Slave and configure the Input ports and Output ports tabs as shown in the following figures.



- The **Bundle** parameter is used in conjunction with the AXI4-Lite or AXI4-Stream (video) interfaces to indicate that multiple ports should be grouped into the same interface. It lets you bundle multiple input/output signals with the same specified bundle name into a single interface port and assigns the corresponding name to the RTL port.

For example in this case, the specified settings on the Input ports tab result in the YCbCr inputs being mapped to AXI4-Stream (video) interfaces and bundled together as an `image_in` port in the generated IP while the YCbCr outputs are bundled together as an `image_out` port.

- The Video Format drop-down menu lets you select between the following formats:
  - YUV 4:2:2
  - YUV 4:4:4
  - RGB
  - Mono/Sensor

- The Video Component drop-down menu is used to subsequently select the right component: R, G, B, Y, U, V.

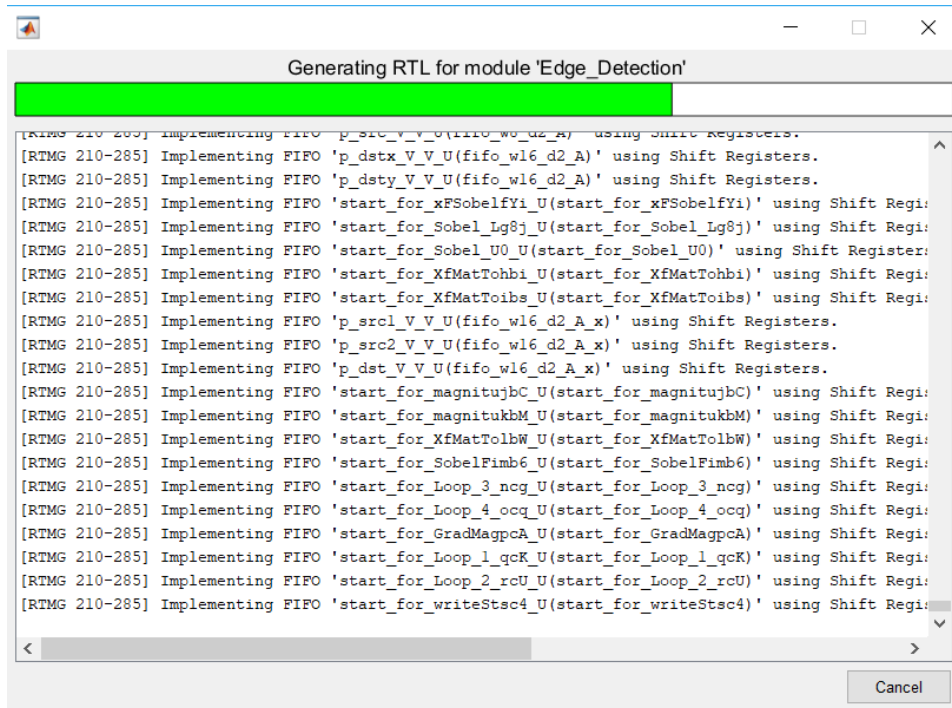
---

## Step 3: Generate IP from Model Composer Design

Using the same example, you will generate an IP from the Edge Detection algorithm.

1. Double-click the **CodeGen\_IP.slx** model in the Current Folder.
2. Double-click into the **Edge Detection** subsystem and review the settings on the Interface Spec block. Based on the previous lab, this block has already been set up to map the input and output ports to AXI4-Stream Video interface, and to use the YUV 4:2:2 video format.
3. Double-click the **Model Composer Hub** block, and set the following in the Block dialog box:
  - **Export Type:** IP Catalog (default)
  - **Target Directory:** ./codegen\_IP
  - **Subsystem name:** Edge Detection
4. To generate an IP from this design, click the **Apply** button in the Model Composer Hub block dialog box to save the settings. Then click the **Generate** button to start the code generation process.

Model Composer opens a progress window to show you the status. After completion, click **OK** and you will see the new `codegen_IP/Edge_Detection_prj` folder in the current folder, which contains the generated IP `solution1` folder.



At the end of the IP generation process, Model Composer opens the Performance Estimates and Utilization Estimates (from Vivado HLS Synthesis report) in the MATLAB Editor, as shown in the following figures.

```

=====
== Performance Estimates
=====
+ Timing (ns):
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target| Estimated| Uncertainty|
  +-----+-----+-----+-----+
  | lap_clk | 5.00| 4.03| 0.63|
  +-----+-----+-----+-----+

+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+-----+
  | Latency | Interval | Pipeline |
  | min | max | min | max | Type |
  +-----+-----+-----+-----+
  | 464048| 464048| 234284| 234284| dataflow |
  +-----+-----+-----+-----+
    
```

```

=====
== Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name      | BRAM_18K| DSP48E| FF  | LUT  |
+-----+-----+-----+-----+
| DSP       | -       | -     | -   | -   |
| Expression| -       | -     | 0   | 12  |
| FIFO      | 0       | -     | 0   | 10  |
| Instance  | 12      | -     | 2562| 2994|
| Memory    | -       | -     | -   | -   |
| Multiplexer| -       | -     | -   | -   |
| Register  | -       | -     | -   | -   |
+-----+-----+-----+-----+
| Total     | 12      | 0     | 2562| 3016|
+-----+-----+-----+-----+
| Available | 890     | 840   | 407600| 203800|
+-----+-----+-----+-----+
| Utilization (%) | 1      | 0     | ~0   | 1   |
+-----+-----+-----+-----+

```

You can also see a summary of the generated RTL ports and their associated protocols at the bottom of the report.

**Note:** The actual timing and resource utilization estimates may deviate from above mentioned values, based on the Vivado HLS build you choose.

```

=====
== Interface
=====
* Summary:
+-----+-----+-----+-----+-----+-----+
| RTL Ports | Dir | Bits | Protocol | Source Object | C Type |
+-----+-----+-----+-----+-----+-----+
| s_axi_AXILites_AWVALID | in | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_AWREADY | out | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_AWADDR | in | 4 | s_axi | AXILites | return void |
| s_axi_AXILites_WVALID | in | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_WREADY | out | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_WDATA | in | 32 | s_axi | AXILites | return void |
| s_axi_AXILites_WSTRB | in | 4 | s_axi | AXILites | return void |
| s_axi_AXILites_ARVALID | in | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_ARREADY | out | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_ARADDR | in | 4 | s_axi | AXILites | return void |
| s_axi_AXILites_RVALID | out | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_RREADY | in | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_RDATA | out | 32 | s_axi | AXILites | return void |
| s_axi_AXILites_RRESP | out | 2 | s_axi | AXILites | return void |
| s_axi_AXILites_BVALID | out | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_BREADY | in | 1 | s_axi | AXILites | return void |
| s_axi_AXILites_BRESP | out | 2 | s_axi | AXILites | return void |
| ap_clk | in | 1 | ap_ctrl_hs | Edge_Detection | return value |
| ap_rst_n | in | 1 | ap_ctrl_hs | Edge_Detection | return value |
| interrupt | out | 1 | ap_ctrl_hs | Edge_Detection | return value |

```

```

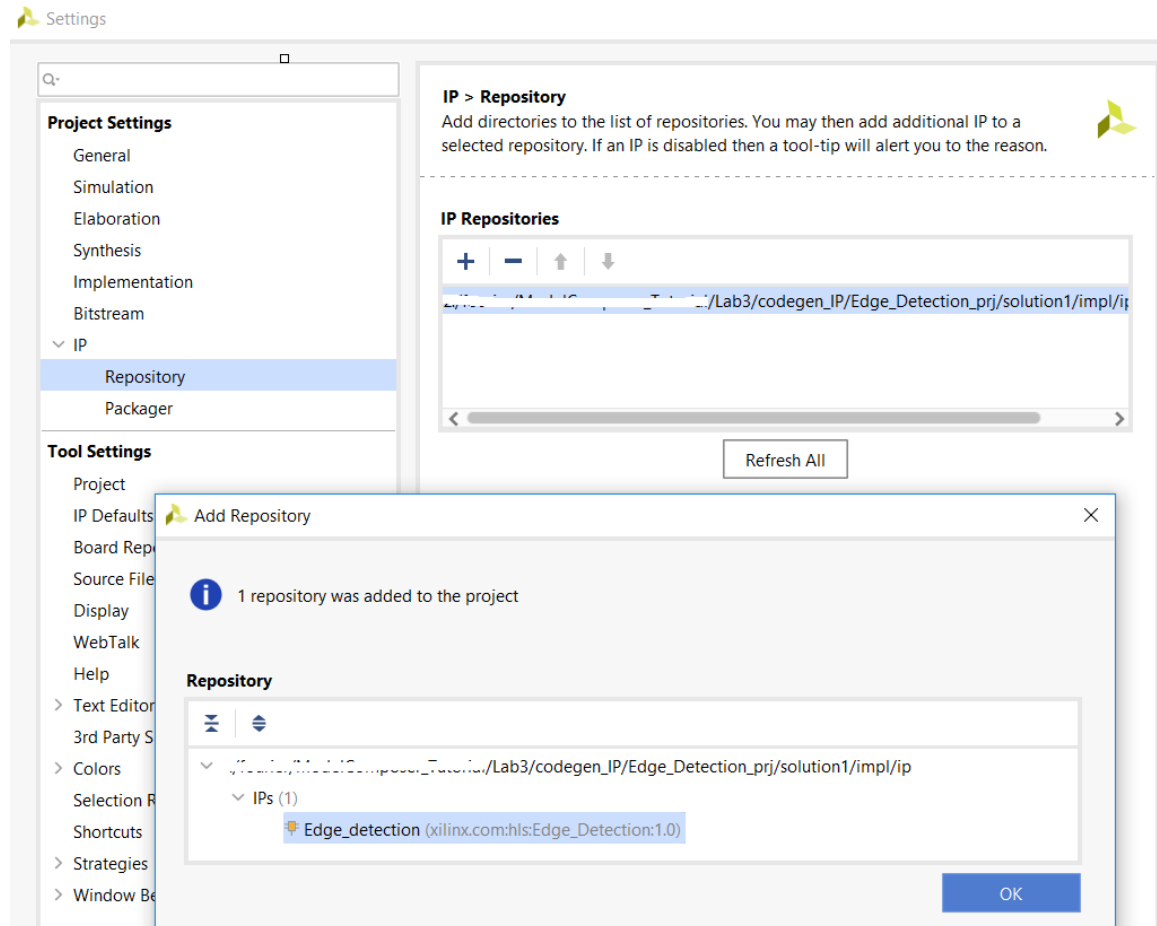
+-----+-----+-----+-----+-----+-----+
| Y_TDATA | in | 16 | axis | image_in_V_data_V | pointer |
| Y_TKEEP | in | 2 | axis | image_in_V_keep_V | pointer |
| Y_TSTRB | in | 1 | axis | image_in_V_strb_V | pointer |
| Y_TUSER | in | 1 | axis | image_in_V_user_V | pointer |
| Y_TLAST | in | 1 | axis | image_in_V_last_V | pointer |
| Y_TID | in | 1 | axis | image_in_V_id_V | pointer |
| Y_TDEST | in | 1 | axis | image_in_V_dest_V | pointer |
| Y_TVALID | in | 1 | axis | image_in_V_dest_V | pointer |
| Y_TREADY | out | 1 | axis | image_in_V_dest_V | pointer |
| Y_Out_TDATA | out | 16 | axis | image_out_V_data_V | pointer |
| Y_Out_TKEEP | out | 2 | axis | image_out_V_keep_V | pointer |
| Y_Out_TSTRB | out | 1 | axis | image_out_V_strb_V | pointer |
| Y_Out_TUSER | out | 1 | axis | image_out_V_user_V | pointer |
| Y_Out_TLAST | out | 1 | axis | image_out_V_last_V | pointer |
| Y_Out_TID | out | 1 | axis | image_out_V_id_V | pointer |
| Y_Out_TDEST | out | 1 | axis | image_out_V_dest_V | pointer |
| Y_Out_TVALID | out | 1 | axis | image_out_V_dest_V | pointer |
| Y_Out_TREADY | in | 1 | axis | image_out_V_dest_V | pointer |
+-----+-----+-----+-----+-----+-----+

```

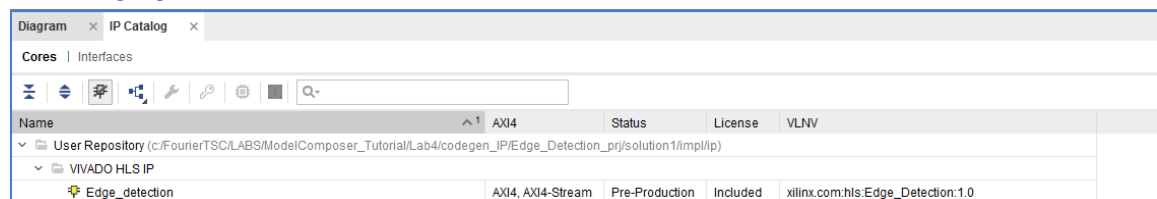
5. Launch Vivado IDE and perform the following steps to add the generated IP to the IP catalog.
6. Create a Vivado RTL project.

When you create the Vivado RTL project, specify the Board as **Kintex-7 KC705 Evaluation Platform** (which is the same as the default Board in the Model Composer Hub block).

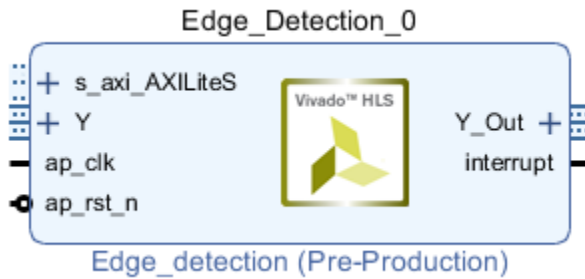
7. In the Project Manager area of the Flow Navigator pane, click **Settings**.
  - a. From **Project Settings** → **IP** → **Repository**, click the + button and browse to `codegen_IP\Edge_Detection_prj\solution1\impl\ip`.
  - b. Click **Select** and see the generated IP get added to the repository.
  - c. Click **OK**.



8. To view the generated Edge\_detection IP in the IP catalog, search for “Dut”. The generated Edge\_detection IP, now appears in the IP catalog under Vivado HLS IP as shown in the following figure.



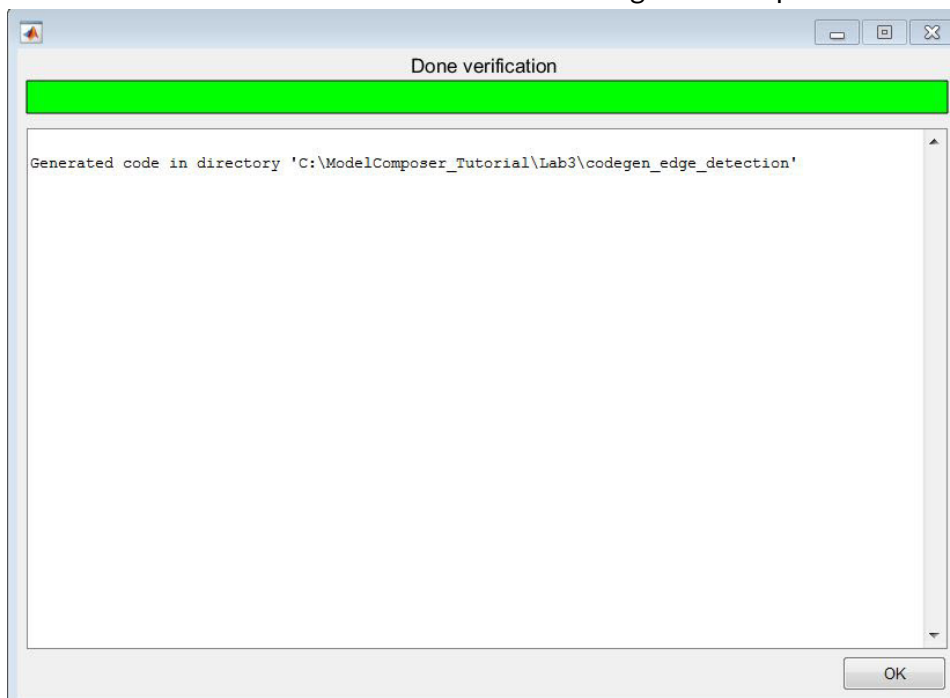
You can now add this IP into an IP integrator block diagram, as shown in the following figure.



## Step 4: Generate HLS Synthesizable Code

In this section you will generate HLS Synthesizable code from the original Edge Detection design. Use the `CodeGen_Cplus.slx` design for this lab. Simulate the model and ensure that algorithm is functionally correct and gives you the results you would expect.

1. Open the Model Composer Hub block dialog box, and set the following:
  - **Export Type:** C++ code
  - **Target Directory:** `./codegen_edge_detection`
  - **Subsystem name:** `Edge Detection`
2. Click the **Apply** button on the Model Composer Hub block dialog box to save the settings and then click the **Generate** button to start the code generation process.

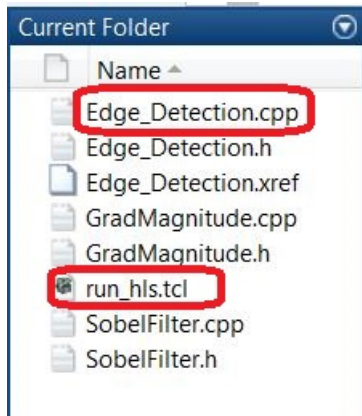


- At the end of code generation, observe the Current Folder in MATLAB.

You should now see a new folder: `codegen_edge_detection` in your **Current Folder**.

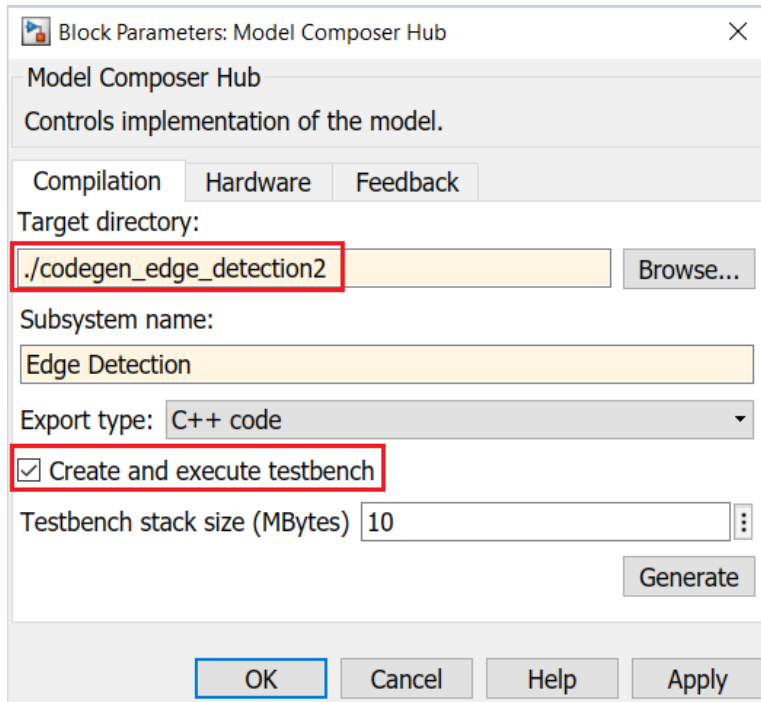
When you click **Generate** on the Model Composer Hub block, Model Composer first simulates the model, then generates the code and places the generated code files in the Target Directory folder. At the end of the code generation process, the window showing the progress of the code generation process tells you where to look for your generated code.

- Open the `codegen_edge_detection` folder and explore the generated code files highlighted in the following figure.



**Note:**

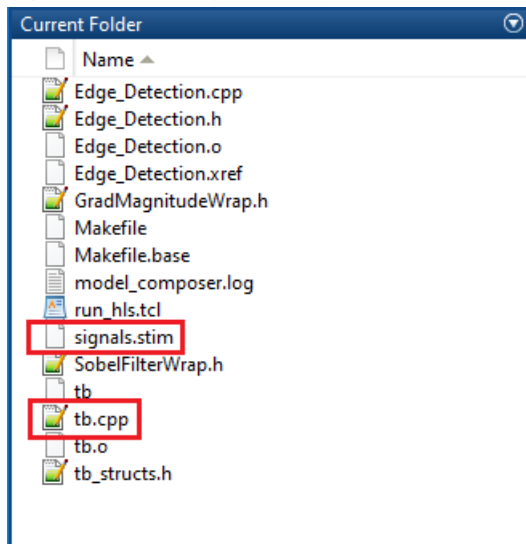
- `Edge_Detection.cpp` is the main file generated for the subsystem.
  - `run_hls.tcl` is the Tcl file needed to create the Vivado HLS project and synthesize the design.
- In the design, open the Model Composer Hub block dialog box, and modify the block settings, as shown in the following figure.
    - Check the **Create and execute testbench** checkbox.
    - Modify the **Target Directory** folder.



6. Click **Apply** and regenerate the code by clicking the **Generate** button. Click **OK** after you see Done Verification in the status bar.

You should now see a new folder, `codegen_edge_detection2`, in your `current` folder.

7. Open the `codegen_edge_detection2` folder and explore the generated code files.



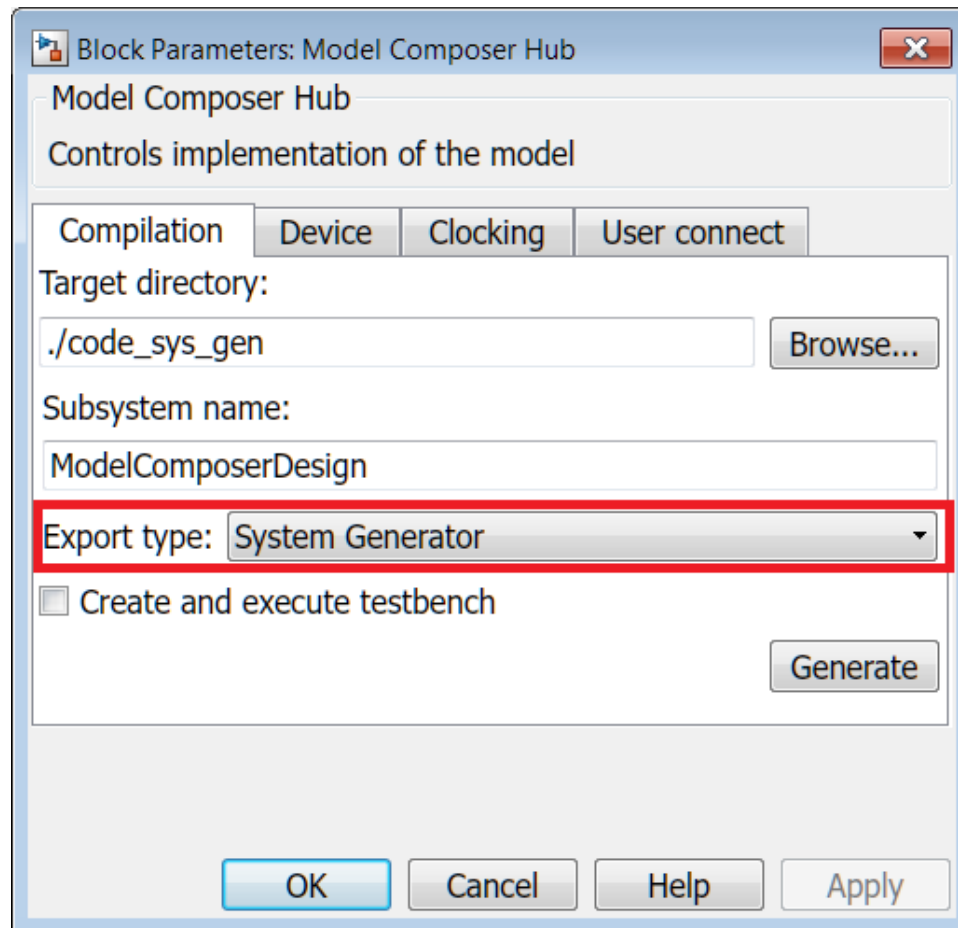
With the **Create and execute testbench** option selected on the Model Composer Hub block, Model Composer logs the inputs and outputs at the boundary of the Edge Detection subsystem and saves the logged stimulus signals in the `signals.stim` file. The `tb.cpp` file is the automatically-generated test bench that you can use for verification in Vivado HLS. At the end of the code generation process, Model Composer automatically verifies that the output from the generated code matches the output logged from the simulation and reports any errors.

---

## Step 5: Port a Model Composer Design to System Generator

Using Model Composer, you can package a model for integration into a System Generator model, which is especially useful if you are an existing System Generator for DSP user. This allows you to take advantage of both the high level of abstraction and simulation speed provided by Model Composer for portions of your design, and the more architecture-aware environment provided by System Generator.

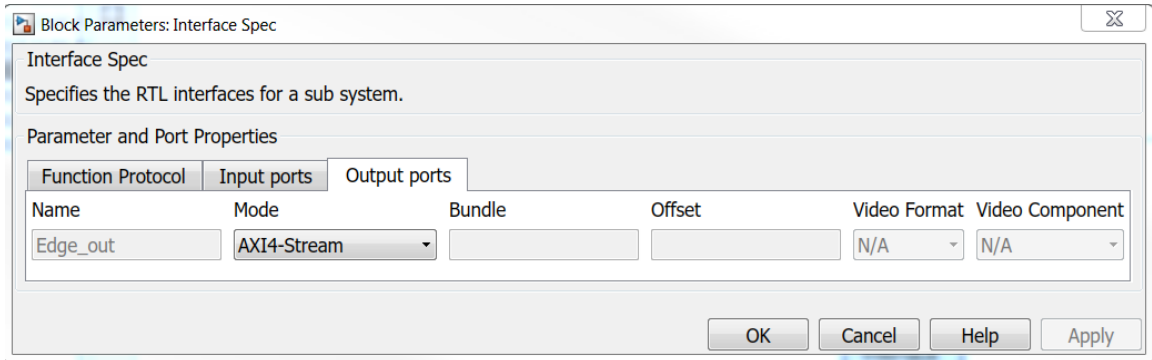
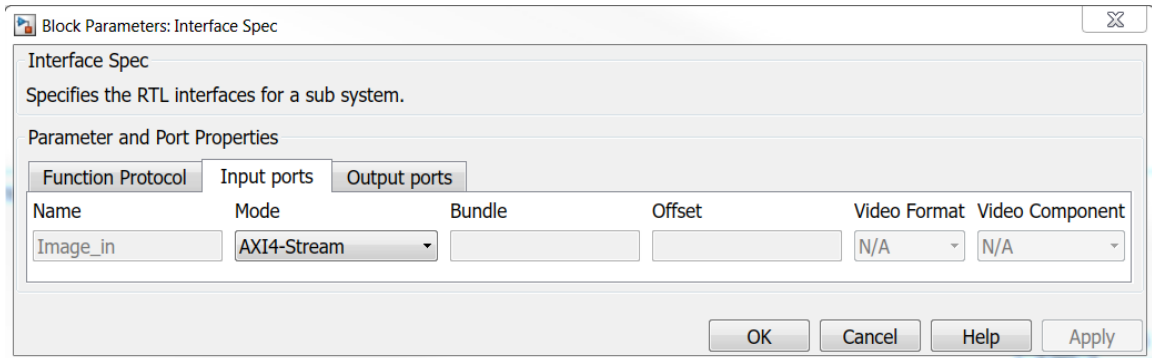
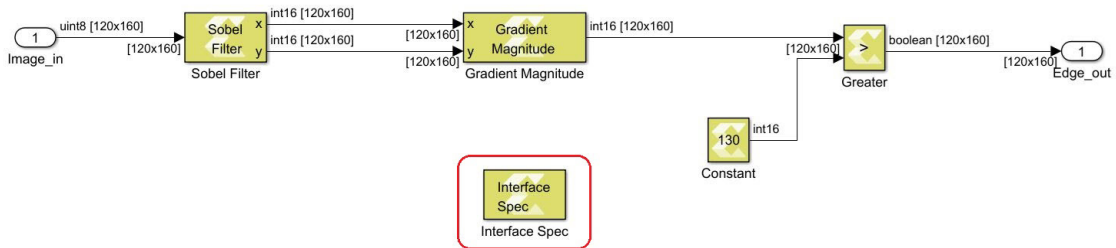
Figure 2: System Generator Export Type



Choosing **System Generator** as the Export type, and clicking **Generate**, creates a synthesized RTL block that you can directly add to a System Generator design using the Vivado HLS block in System Generator.

In this lab, you create an IP using Model Composer and then use the synthesized RTL as a block in a System Generator design.

1. In the `ModelComposer_Tutorial/Lab5/ModelComposer_to_SysGen` folder, double-click `MoC_design.slx` to see the Model Composer design. The design is configured to have AXI4-Stream interfaces at both the input and output. This is done through the Interface Spec block within the `ModelComposerDesign` subsystem. Note that there are no structural changes required at the Simulink level to change interfaces for the IP.



2. Open the **followme\_script.m** in MATLAB. This script will guide you through all the steps to import the Model Composer generated solution as a block in System Generator.
3. Read the comments at the start of each section (labeled Section 1 to Section 8) in the MATLAB script and execute each section one at a time (the start of each section is marked by a %% sign). You can click on **Run and Advance** to step through each section in the script. The sections are as follows:

a. Section 1: Set up

Open MATLAB for Model Composer and choose a video file as an input.

```
video_filename = 'vipmen.avi';

v = VideoReader(video_filename);
frame_height = v.Height;
frame_width = v.Width;
save video_handle v
```

b. Section 2: Creating a System Generator solution from a Model Composer design.

Model Composer allows you to export a design as a block into System Generator. The result of exporting a design from Model Composer to System Generator is a `solution` folder that you will import into the System Generator design using Vivado HLS block in System Generator.

```
open_system('MoC_design');
xmcGenerate('MoC_design');
```

#### c. Section 3: Serializing the input video

Serialize the input video which is required for use with the System Generator design which will do pixel-based processing.

```
stream_in = zeros(ceil(v.FrameRate*v.Duration*v.Height*v.Width),1);

i = 1;
while hasFrame(v)
    frame = rgb2gray(readFrame(v));
    a = reshape(frame',[],1);
    stream_in(i:i+length(a)-1) = a;
    i = i + length(a);
end

save stream_in stream_in
```

#### d. Section 4: Launch System Generator

Using System Generator currently requires launching a separate MATLAB session using the System Generator Launcher.

**Note:** Use a Windows or Linux command accordingly to change the path to point to your local version of System Generator in order to launch System Generator properly.

- Windows

```
system('C:\Xilinx\Vivado\2020.x\bin\sysgen.bat')
```

- Linux

```
system('<install directory>/Vivado/2020.x/bin/sysgen')
```

**Note:** Where 'x' in 2020.x denotes the latest release.

#### e. Section 5: Import the generated solution into System Generator

Set up the Vivado HLS block in the System Generator design to point to the correct solution folder generated in Section 2.

```
open_system('sys_gen_AXI');
```

#### f. Section 6: Simulate the System Generator Design

Simulate the System Generator design and save the outputs into a MAT file. Note that the simulation will be slower than the Model Composer design since we are simulating the generated RTL and are doing an element-by-element based processing.

```
sim('sys_gen_AXI');
```

g. Section 7: De-serializing the output of the System Generator design.

This is a post-processing step that creates a frame-based video for playback using the outputs logged from the System Generator simulation.

```
load stream_out
load video_handle

disp(['Length of input stream is ', num2str(length(stream_in))])
disp(['Length of output stream is ', num2str(length(stream_out))])

outputVideo = VideoWriter('stream_out.avi');
outputVideo.FrameRate = v.FrameRate;
open(outputVideo)

The output is Boolean. This is why we multiply the img by 255, so
that imshow shows the image.
for i = 1:length(stream_out)/v.Height/v.Width
    img = reshape(stream_out((i-1)*v.Height*v.Width
+1:i*v.Height*v.Width), v.Width, v.Height);
    writeVideo(outputVideo, 255*img')
end

close(outputVideo);
```

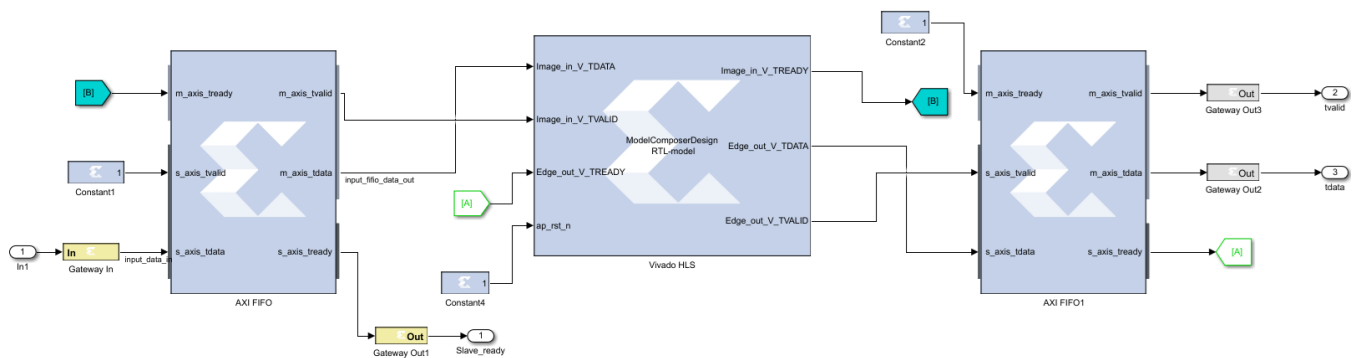
h. Section 8: Play the de-serialized output using `imshow`.

```
imshow('stream_out.avi')
```

4. The AXI4-Stream uses three signals, DATA, READY, and VALID. The READY signal is a back pressure signal from the slave side to the master side indicating whether the slave side can accept new data.

As you examine the System Generator model in Section 5, pay attention to the labels on blocks for each signal to help you understand how the model is designed. For example, whenever the IP can no longer accept input, the READY signal (top right of the Vivado HLS block) puts pressure on the master side of the input AXI FIFO by resetting the READY signal. Likewise, the input AXI FIFO pressures the input stream by resetting its READY signal.

**Note:** In Simulink all the inputs to a block are to one side of the block, and all the outputs are on the opposite side. As such, all the slave or master signals are not bundled together on one side of the block as you might expect.



## Conclusion

In this lab, you learned:

- About the Interface Spec block terminology and parameter names.
- How to specify interfaces and to map them directly from the Simulink environment using the Interface Spec block.
- How Model Composer enables push button IP creation from your design in Simulink with the necessary interfaces.
- How the Model Composer Hub block in Model Composer helps move from algorithm to implementation.
- How to generate code files from the Model Composer Hub block and read them.
- How to set compilation targets to C++ code, IP Catalog and System Generator.

Some additional notes about Model Composer:

- Model Composer takes care of mapping interfaces as part of the code generation process and you don't have to take care of interleaving and de-interleaving color channels and interface connections at the design level.
- An Interface Spec block must be placed within the subsystem for which you intend to generate code.
- For the C++ code compilation target, Model Composer generates everything you would need to further optimize and synthesize the design using Vivado HLS.
- Model Composer automatically generates the test vectors and test benches for C/RTL cosimulation in Vivado HLS.

- Model Composer provides an option to export a design back into System Generator through the Vivado HLS block
- When moving from a Model Composer design to System Generator, you move from an untimed C-based bit-true design to an RTL-based bit-true and cycle-accurate design.

The following `solution` directory contains the final Model Composer (\*.slx) files for this lab.

```
C:\ModelComposer_Tutorial\Lab5\solution
```

# Additional Resources and Legal Notices

---

## Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby **DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE**; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at <https://www.xilinx.com/legal.htm#tos>.

### **AUTOMOTIVE APPLICATIONS DISCLAIMER**

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING

OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

### **Copyright**

© Copyright 2017-2020 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.