Vitis Model Composer Tutorial

UG1498 (v2021.1) July 16, 2021

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/16/2021 Version 2021.1	
Document Title and Revision Summary	Title changed to ''Vitis Model Composer Tutorial''
General updates	Updated for release 2021.1

Locating the Tutorial Design Files

There are separate project files and sources for each of the labs in this tutorial. Download the reference design files for this tutorial from the Xilinx website, then 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 directory extracted from `ug1498-model-composer-sys-gen-tutorial.zip` *each time you start this tutorial.*

Table of Contents

XILINX

Chapter 1

HDL Library

Lab 1: Introduction to Model Composer HDL Library

In this lab, you will learn how to use Model Composer HDL library to specify a design in Simulink® and synthesize the design into an FPGA. This tutorial uses a standard FIR filter and demonstrates how Model Composer provides you the design options that allow you to control the fidelity of the final FPGA hardware.

Objectives

After completing this lab, you will be able to:

- Capture your design using the Model Composer HDL Blocksets.
- Capture your designs in either complex or discrete Blocksets.
- Synthesize your designs in an FPGA using the Vivado® Design Environment.

Procedure

This lab has four primary parts:

- **Step 1:** Review an existing Simulink design using the Xilinx® FIR Compiler block, and review the final gate level results in Vivado.
- **Step 2:** Use over-sampling to create a more efficient design.
- **Step 3:** Design the same filter using discrete blockset parts.
- **Step 4:** Understand how to work with Data Types such as Floating-point and Fixed-point.

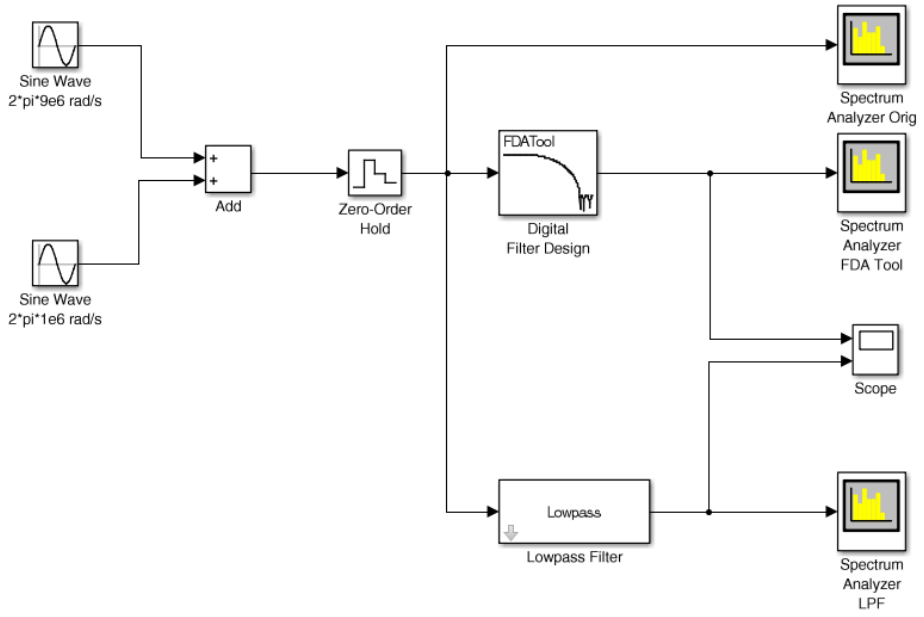
Step 1: Creating a Design in an FPGA

In this step, you learn the basic operation of Model Composer and how to synthesize a Simulink design into an FPGA.

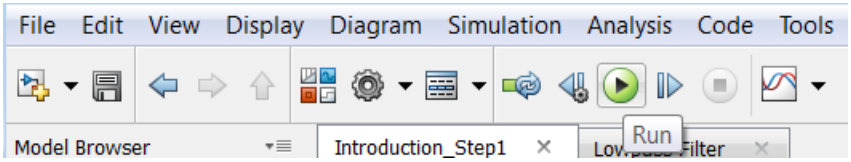
1. Invoke Vitis Model Composer.
   - On Windows systems, select **Windows → Xilinx Design Tools → Vitis Model Composer 202x.x**.
   - On Linux systems, type `model_composer` at the command prompt.
2. Navigate to the Lab1 folder: `\HDL_Library\Lab1`.

   You can view the directory contents in the MATLAB® Current Folder browser, or type `ls` at the command line prompt.
3. Open the Lab1_1 design as follows:
   a. At the MATLAB command prompt, type `open Lab1_1.slx` OR
   b. Double-click `Lab1_1.slx` in the Current Folder browser.

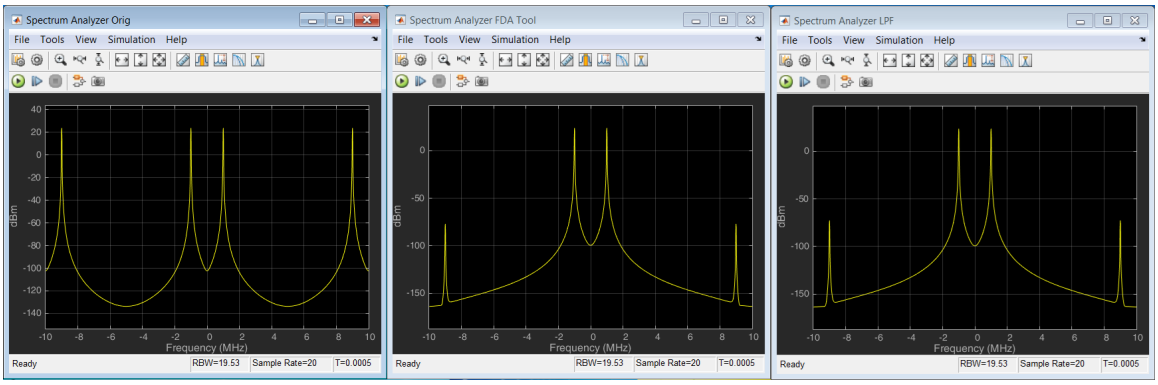
   The Lab1_1 design opens, showing two sine wave sources being added together and passed separately through two low-pass filters. This design highlights that a low-pass filter can be implemented using the Simulink FDATool or Lowpass Filter blocks.



4. From your Simulink project worksheet, select **Simulation → Run** or click the **Run** simulation button.

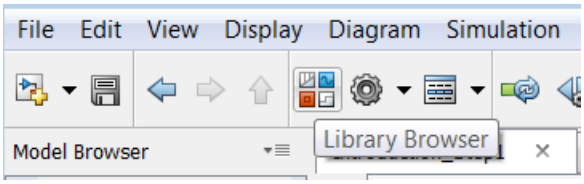


When simulation completes you can see the spectrum for the initial summed waveforms, showing a 1 MHz and 9 MHz component, and the results of both filters showing the attenuation of the 9 MHz signals.



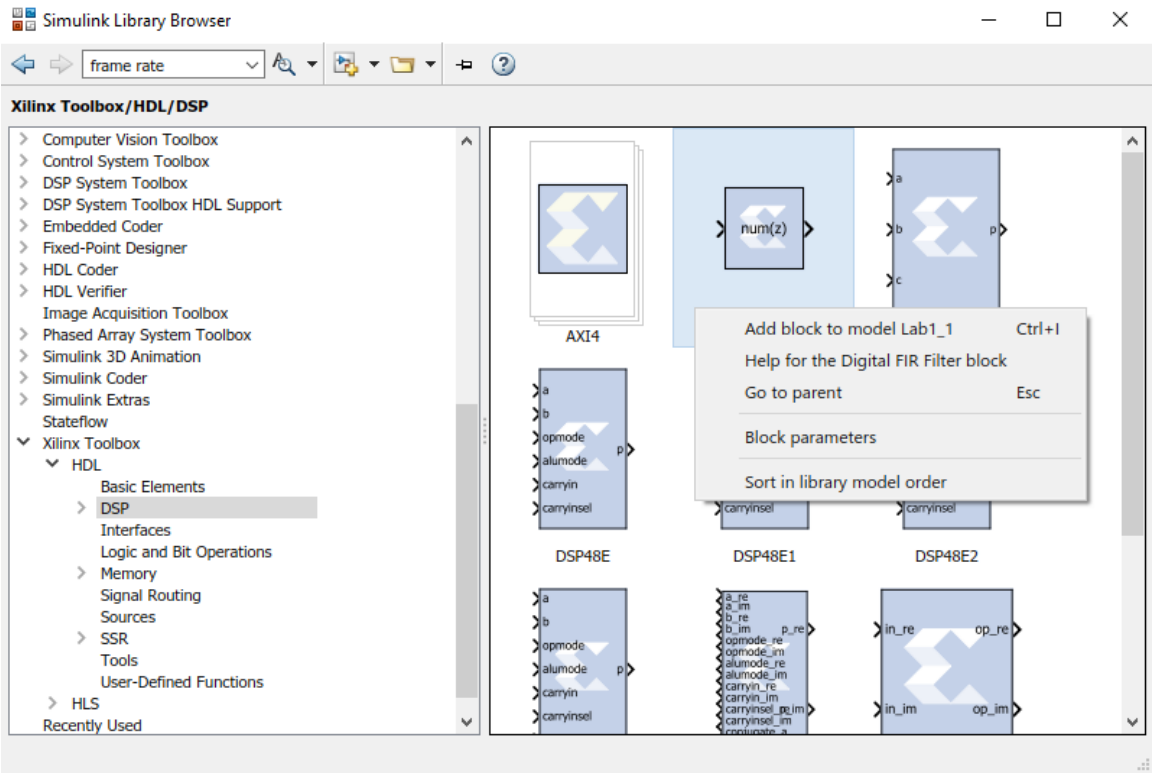
You will now create a version of this same filter using HDL blocks for implementation in an FPGA.

5. Click the **Library Browser** button in the Simulink toolbar to open the Simulink Library Browser.



When using Model Composer, the Simulink library includes specific blocks for implementing designs in an FPGA. You can find a complete description of the HDL library blocks provided by Model Composer in the *Vitis Model Composer Tutorial* (UG1498).

6. Expand the **Xilinx Toolbox → HDL** menu, select **DSP**, then select **Digital FIR Filter**.
7. Right-click the **Digital FIR Filter** block and select **Add block to model Lab1_1**.



You can define the filter coefficients for the Digital FIR Filter block by accessing the block attributes–double-click the **Digital FIR Filter** block to view these–or, as in this case, they can be defined using the FDATool.

8. From **Xilinx Toolbox → HDL → Tools**, select **FDATool** and add it to the Lab1_1 design.

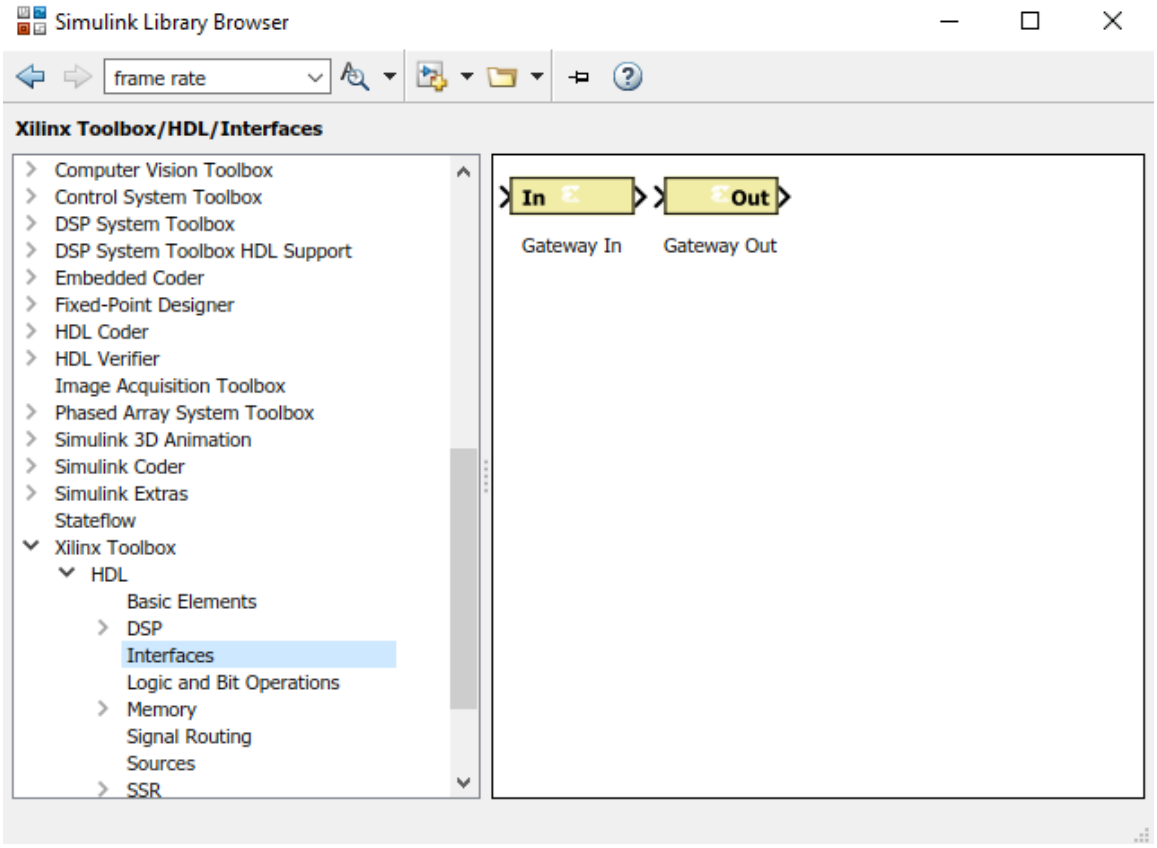
   An FPGA design requires three important aspects to be defined:

   - The input ports
   - The output ports
   - The FPGA technology

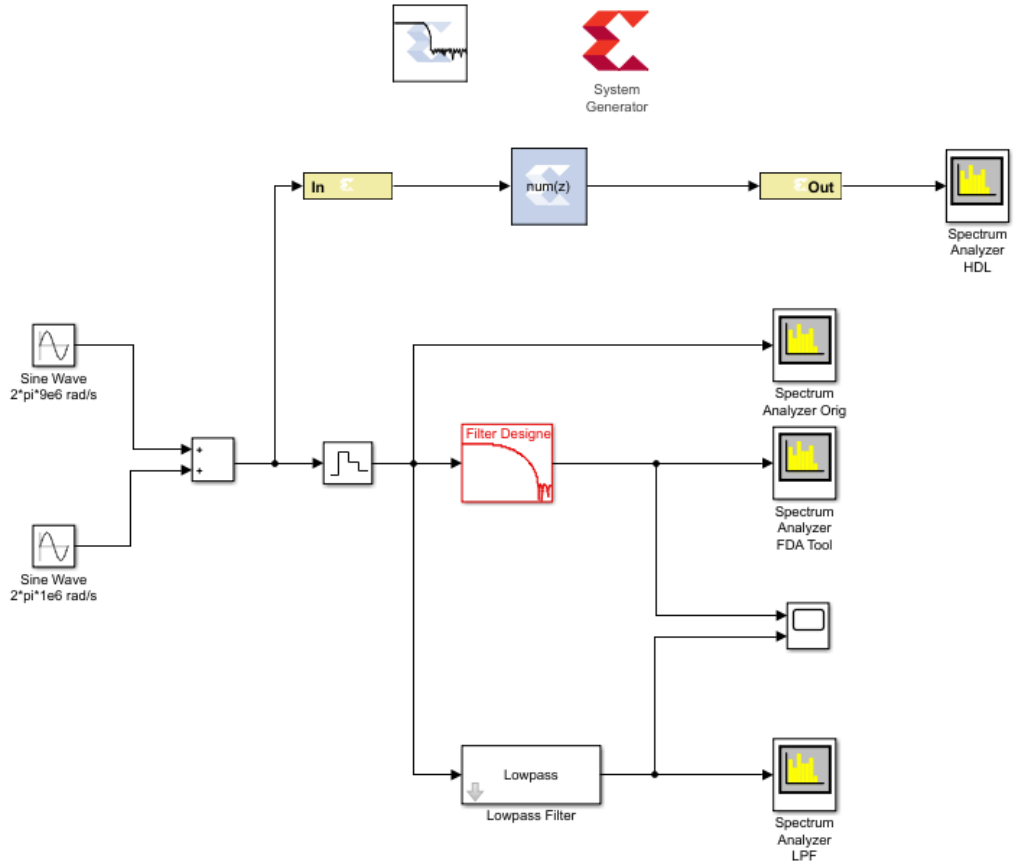
   The next three steps show how each of these attributes is added to your Simulink design.

   > **IMPORTANT!** *If you fail to correctly add these components to your design, it cannot be implemented in an FPGA. Subsequent labs will review in detail how these blocks are configured; however, they must be present in all Model Composer HDL designs.*

9. In the Interfaces menu, select **Gateway In**, and add it to the design.



10. Similarly, from the same menu, add a Gateway Out block to the design.
11. From the Tools menu, under the HDL menu, add the System Generator token used to define the FPGA technology.
12. Finally, make a copy of one of the existing Spectrum Analyzer blocks, and rename the instance to Spectrum Analyzer HDL by clicking the instance name label and editing the text.
13. Connect the blocks as shown in the following figure. Use the left-mouse key to make connections between ports and nets.



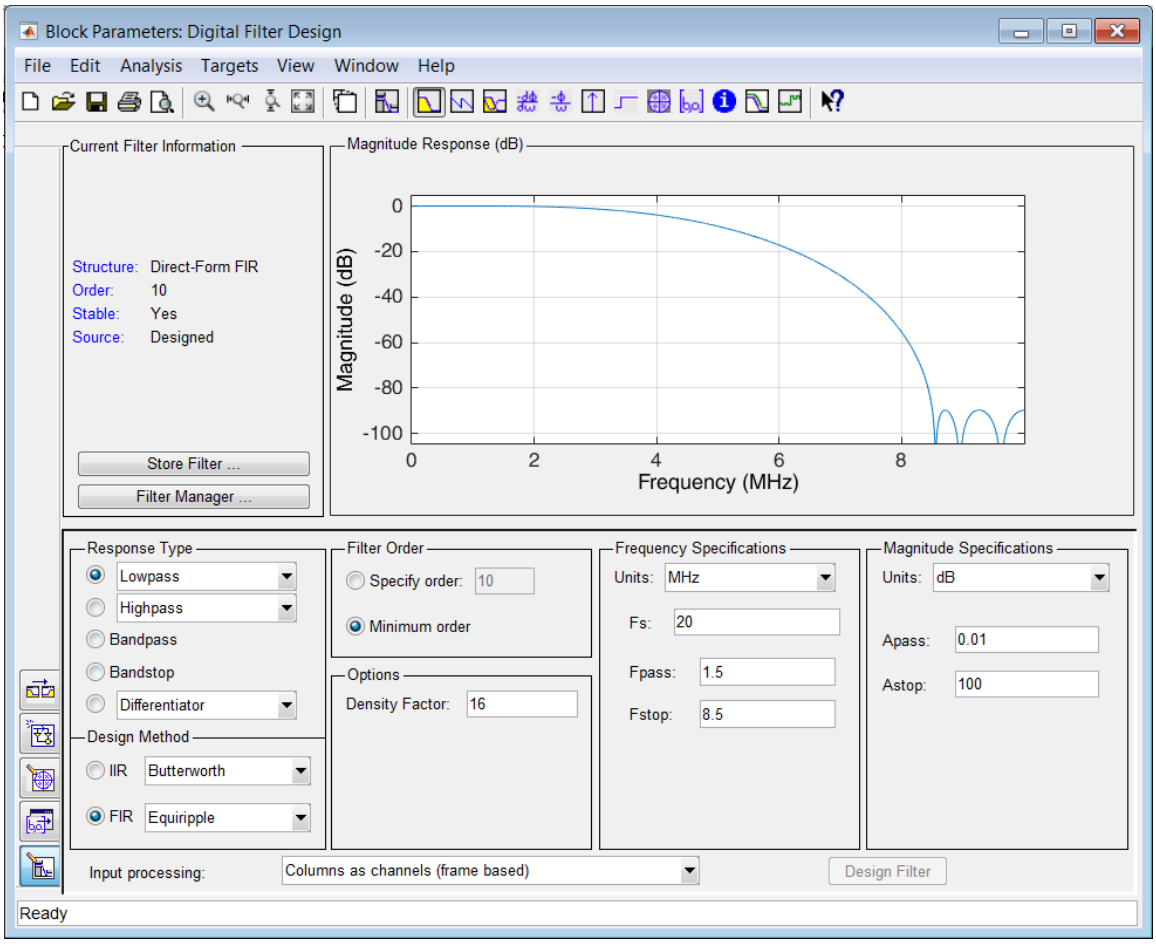
The next part of the design process is to configure the HDL blocks.

Configure the HDL Blocks

The first task is to define the coefficients of the new filter. For this task you will use the Xilinx block version of FDATool. If you open the existing FDATool block, you can review the existing Frequency and Magnitude specifications.

1. Double-click the **Digital Filter Design** instance to open the Properties Editor.

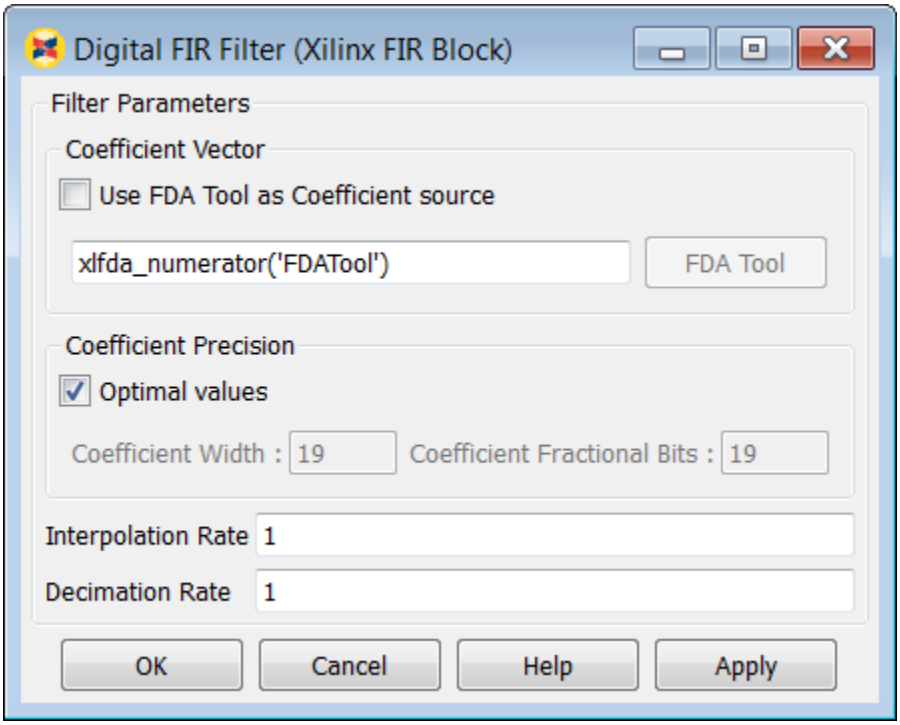
   This allows you to review the properties of the existing filter.



2. Close the Properties Editor for the Digital Filter Design instance.
3. Double-click the **FDATool** instance to open the Properties Editor.
4. Change the filter specifications to match the following values:
   - Frequency Specifications
     - Units = MHz
     - Fs = 20
     - Fpass = 1.5
     - Fstop = 8.5
   - Magnitude Specifications
     - Units = dB
     - Apass = 0.01
     - Astop = 100
5. Click the **Design Filter** button at the bottom and close the Properties Editor.

Now, associate the filter parameters of the FDATool instance with the Digital FIR Filter instance.

6. Double-click the **Digital FIR Filter** instance to open the Properties Editor.
7. In the Filter Parameters section, replace the existing coefficients (Coefficient Vector) with `xlfda_numerator('FDATool')` to use the coefficients defined by the FDATool instance.



8. Click **OK** to exit the Digital FIR Filter Properties Editor.

In an FPGA, the design operates at a specific clock rate and using a specific number of bits to represent the data values.

The transition between the continuous time used in the standard Simulink environment and the discrete time of the FPGA hardware environment is determined by defining the sample rate of the Gateway In blocks. This determines how often the continuous input waveform is sampled. This sample rate is automatically propagated to other blocks in the design by Model Composer. In a similar manner, the number of bits used to represent the data is defined in the Gateway In block and also propagated through the system.

Although not used in this tutorial, some HDL blocks enable rate changes and bit-width changes, up or down, as part of this automatic propagation. More details on these blocks are found in the *Vitis Model Composer User Guide* (UG1483).

Both of these attributes (rate and bit width) determine the degree of accuracy with which the continuous time signal is represented. Both of these attributes also have an impact on the size, performance, and hence cost of the final hardware.

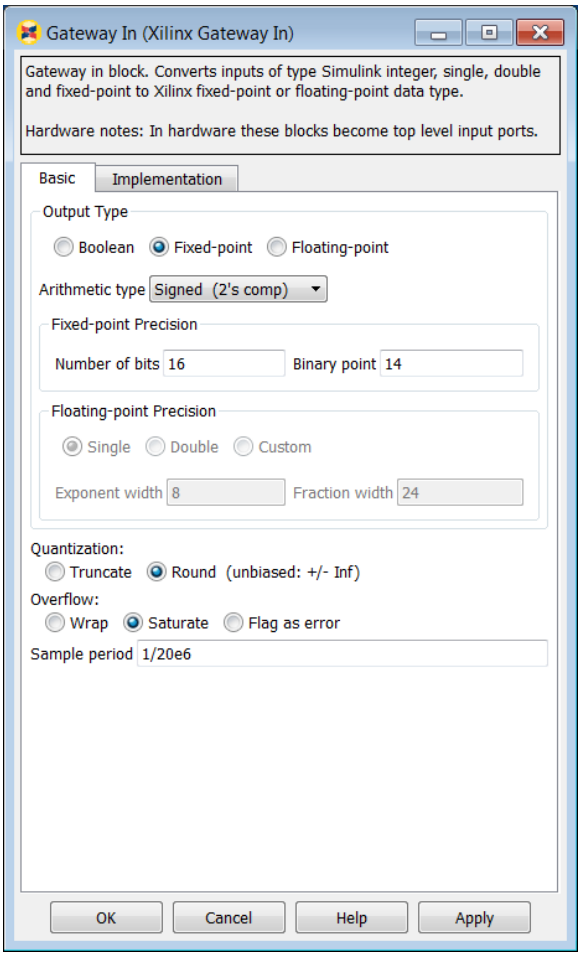
Model Composer allows you to use the Simulink environment to define, simulate, and review the impact of these attributes.

9. Double-click the **Gateway In** block to open the Properties Editor.

   Because the highest frequency sine wave in the design is 9 MHz, sampling theory dictates the sampling frequency of the input port must be at least 18 MHz. For this design, you will use 20 MHz.

10. At the bottom of the Properties Editor, set the Sample Period to 1/20e6.

11. For now, leave the bit width as the default fixed-point 2's complement 16-bits with 14-bits representing the data below the binary point. This allows us to express a range of -2.0 to 1.999, which exceeds the range required for the summation of the sine waves (both of amplitude 1).



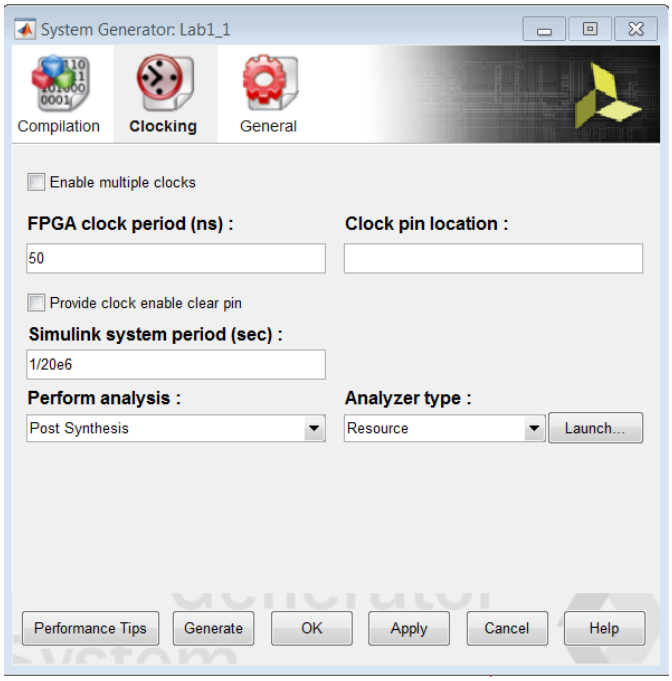
12. Click **OK** to close the Gateway In Properties Editor.

    This now allows us to use accurate sample rate and bit-widths to accurately verify the hardware.

13. Double-click the **System Generator** token to open the Properties Editor.

Because the input port is sampled at 20 MHz to adequately represent the data, you must define the clock rate of the FPGA and the Simulink sample period to be at least 20 MHz.

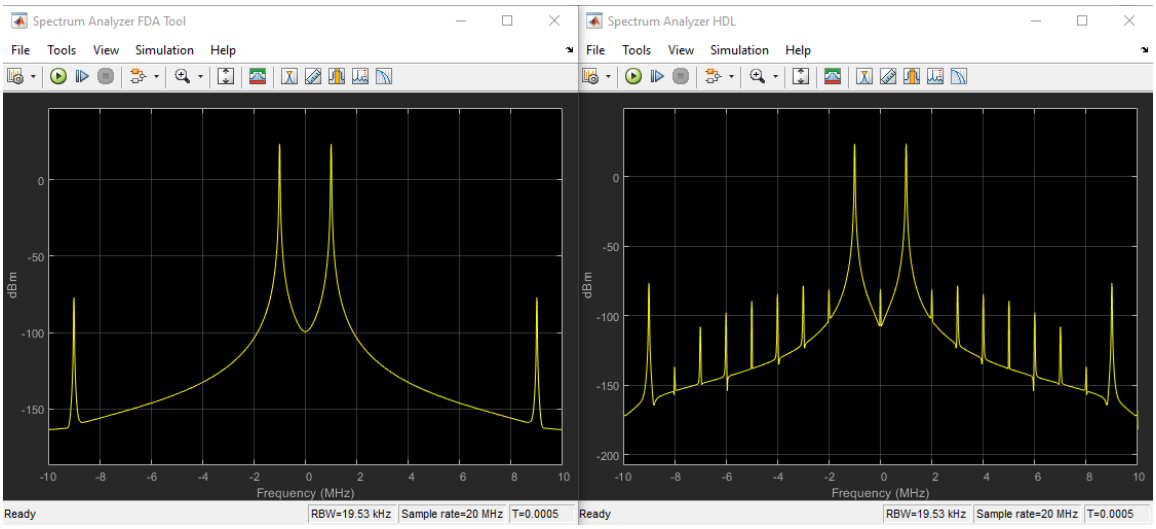
14. Select the Clocking tab.
    a. Specify an FPGA clock period of 50 ns (1/20 MHz).
    b. Specify a Simulink system period of 1/20e6 seconds.
    c. From the Perform analysis menu, select **Post Synthesis** and from the Analyzer type menu select **Resource** as shown in the following figure. This option gives the resource utilization details after completion.



15. Click **OK** to exit the System Generator token.

16. Click the Run simulation button  to simulate the design and view the results, as shown in the following figure.

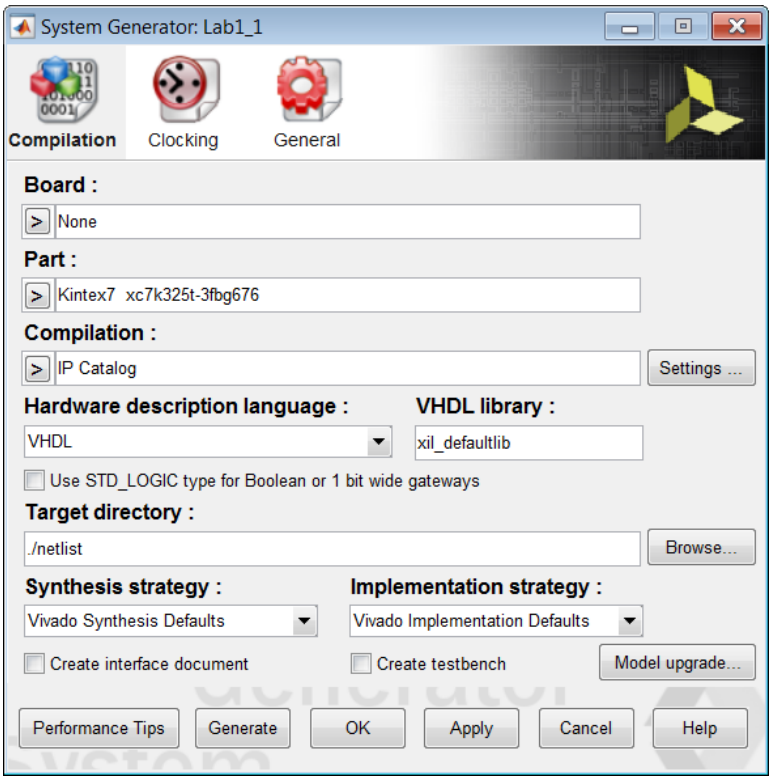
    Because the new design is cycle and bit accurate, simulation might take longer to complete than before.



The results are shown above, on the right hand side (in the Spectrum Analyzer HDL window), and differ slightly from the original design (shown on the left in the Spectrum Analyzer FDA Tool window). This is due to the quantization and sampling effect inherent when a continuous time system is described in discrete time hardware.

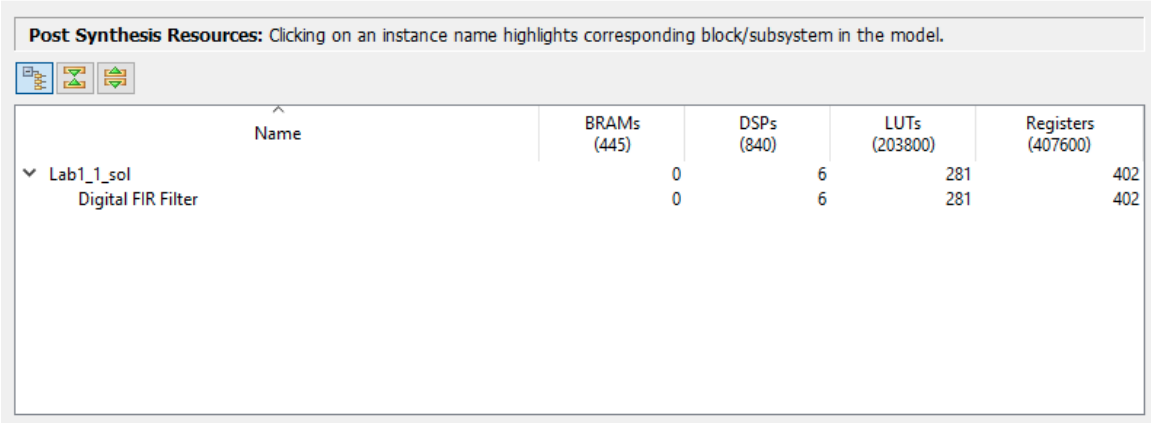
The final step is to implement this design in hardware. This process will synthesize everything contained between the Gateway In and Gateway Out blocks into a hardware description. This description of the design is output in the Verilog or VHDL Hardware Description Language (HDL). This process is controlled by the System Generator token.

17. Double-click the **System Generator** token to open the Properties Editor.
18. Select the **Compilation** tab to specify details on the device and design flow.
19. From the Compilation menu, select the IP catalog compilation target to ensure the output is in IP catalog format. The Part menu selects the FPGA device. For now, use the default device. Also, use the default Hardware description language, VHDL.

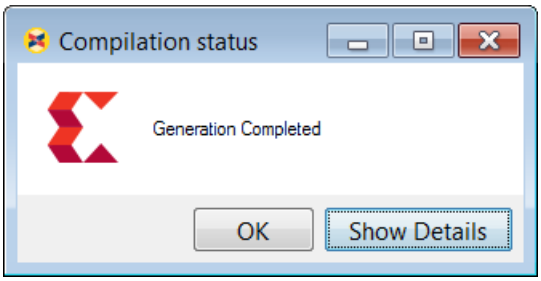


20. Click **Generate** to compile the design into hardware.

    The compilation process transforms the design captured in Simulink blocks into an industry standard Register Transfer Level (RTL) design description. The RTL design can be synthesized into a hardware design. A Resource Analyzer window appears when the hardware design description has been generated.



    The Compilation status dialog box also appears.



21. Click **OK** to dismiss the Compilation status dialog box.
22. Click **OK** to dismiss the Resource Analyzer window.
23. Click **OK** to dismiss the System Generator token.

The final step in the design process is to create the hardware and review the results.

Review the Results

The output from design compilation process is written to the `netlist` directory. This directory contains three subdirectories:

- **`sysgen`:** This contains the RTL design description written in the industry standard VHDL format. This is provided for users experienced in hardware design who wish to view the detailed results.
- **`ip`:** This directory contains the design IP, captured in Xilinx IP catalog format, which is used to transfer the design into the Xilinx Vivado. Lab 5: Using AXI Interfaces and IP Integrator, presented later in this document, explains in detail how to transfer your design IP into the Vivado for implementation in an FPGA
- **`ip_catalog`:** This directory contains an example Vivado project with the design IP already included. This project is provided only as a means of quick analysis.

The previous Resource Analyzer: Lab1_1 figure shows the summary of resources used after the design is synthesized. You can also review the results in hardware by using the example Vivado project in the `ip_catalog` directory.

IMPORTANT! *The Vivado project provided in the `ip_catalog` directory does not contain top-level I/O buffers. The results of synthesis provide a very good estimate of the final design results; however, the results from this project cannot be used to create the final FPGA.*

When you have reviewed the results, exit the `Lab1_1.slx` Simulink worksheet.

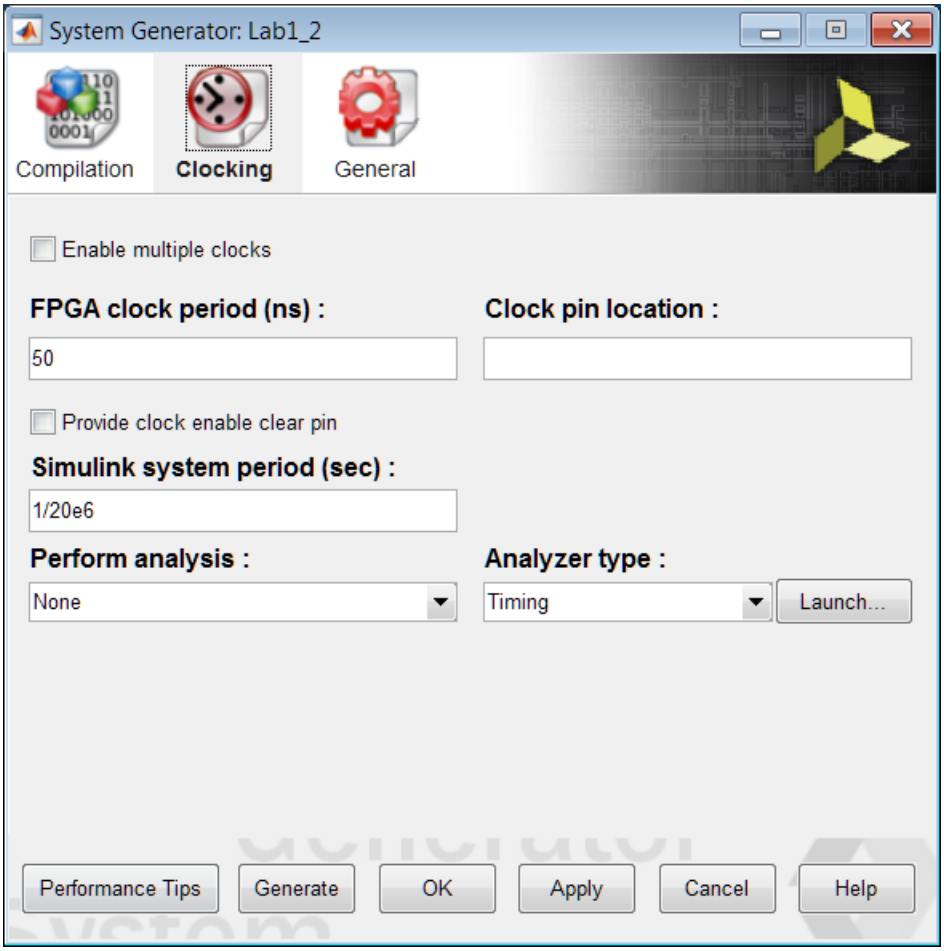
Step 2: Creating an Optimized Design in an FPGA

In this step you will see how an FPGA can be used to create a more optimized version of the same design used in Step 1, by oversampling. You will also learn about using workspace variables.

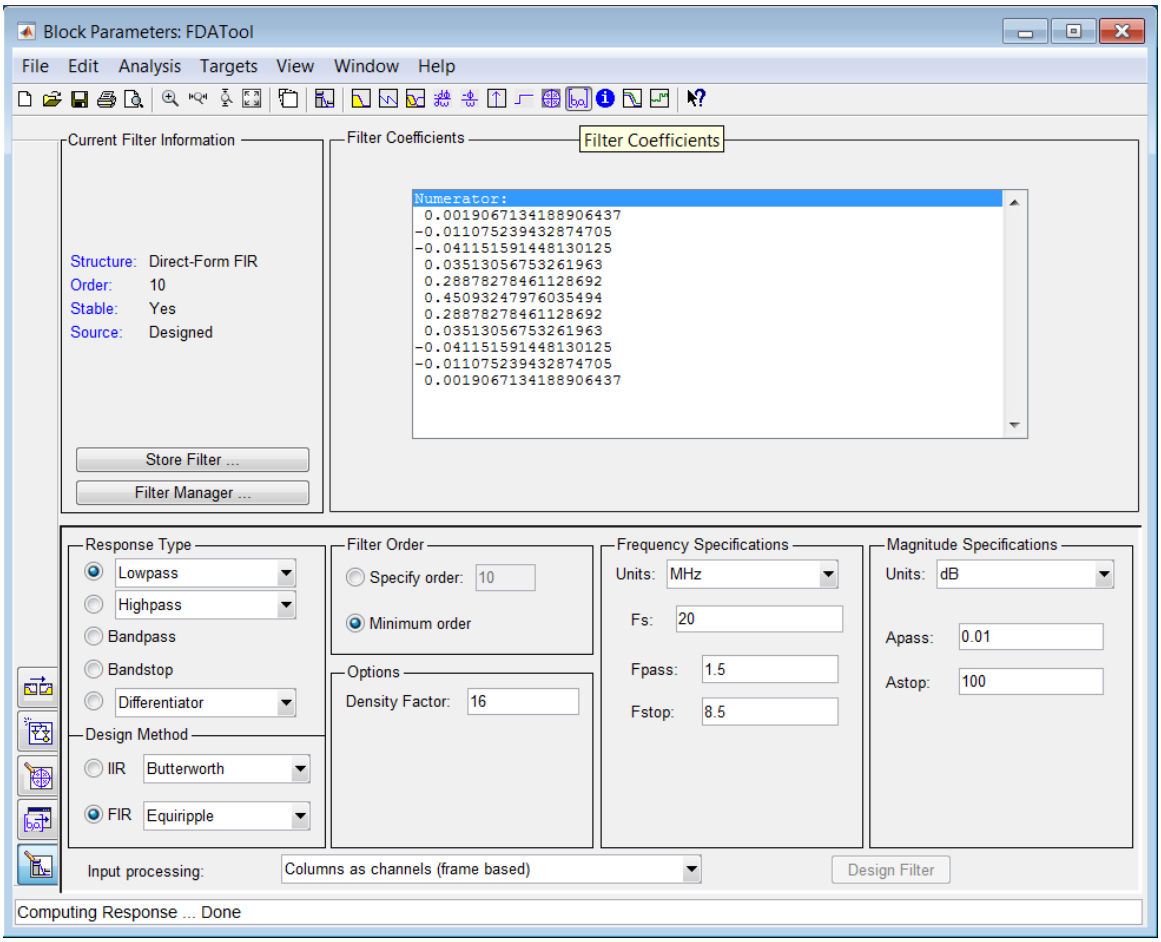
1. At the command prompt, type `open Lab1_2.slx`.

2. From your Simulink project worksheet, select **Simulation → Run** or click the Run simulation button to confirm this is the same design used in Step 1: Creating a Design in an FPGA.
3. Double-click the System Generator token to open the Properties Editor.

   As noted in Step 1, the design requires a minimum sample frequency of 18 MHz and it is currently set to 20 MHz (a 50 ns FPGA clock period).



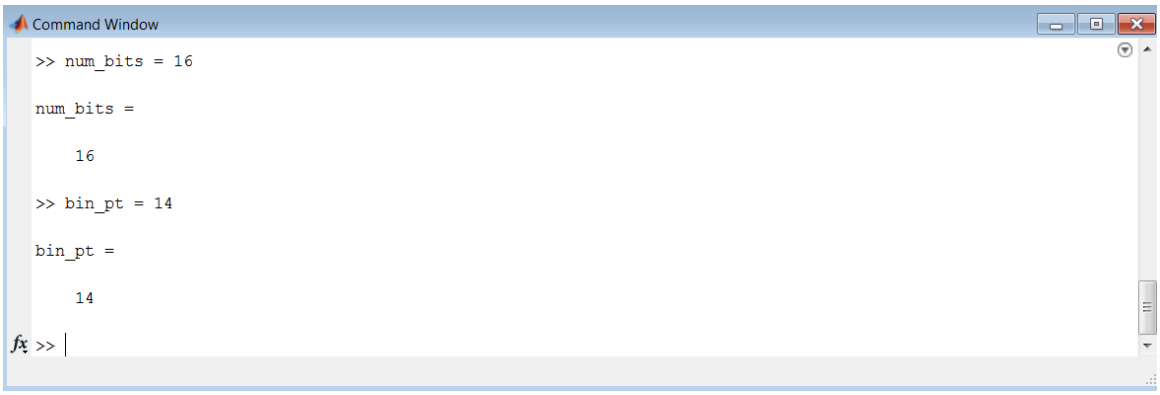
   The frequency at which an FPGA device can be clocked easily exceeds 20 MHz. Running the FPGA at a much higher clock frequency will allow Model Composer to use the same hardware resources to compute multiple intermediate results.
4. Double-click the **FDATool** instance to open the Properties Editor.
5. Click the **Filter Coefficients** button to view the filter coefficients.



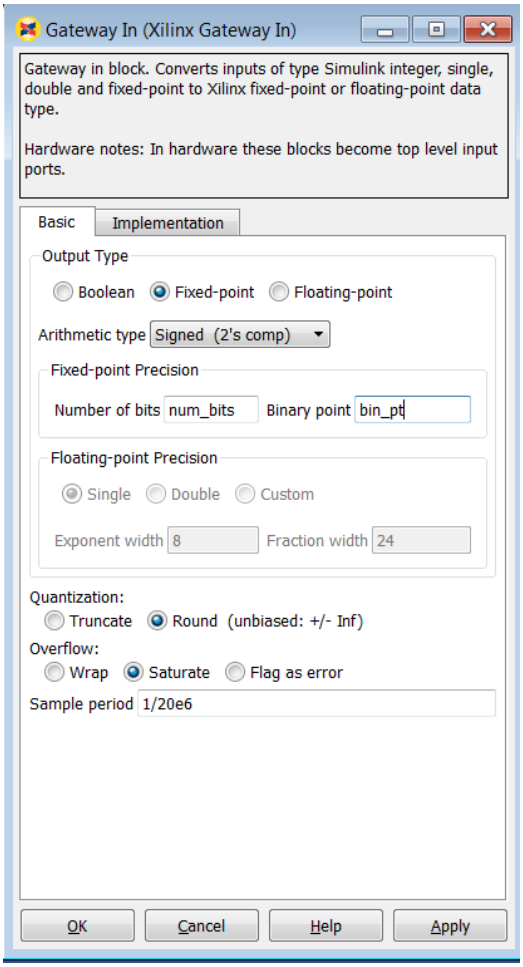
This shows the filter uses 11 symmetrical coefficients. This requires a minimum of six multiplications. This is indeed what is shown at the end of the HDL Blocks section where the final hardware is using six DSP48 components, the FPGA resource used to perform a multiplication.

The current design samples the input at a rate of 20 MHz. If the input is sampled at 6 times the current frequency, it is possible to perform all calculations using a single multiplier.

6. Close the FDATool Properties Editor.
7. You will now replace some of the attributes of this design with workspace variables. First, you need to define some workspace variables.
8. In the MATLAB Command Window:
   a. Enter `num_bits = 16`
   b. Enter `bin_pt = 14`



9. In design Lab1_2, double-click the **Gateway In** block to open the Properties Editor.
10. In the Fixed-Point Precision section, replace 16 with `num_bits` and replace 14 with `bin_pt`, as shown in the following figure.

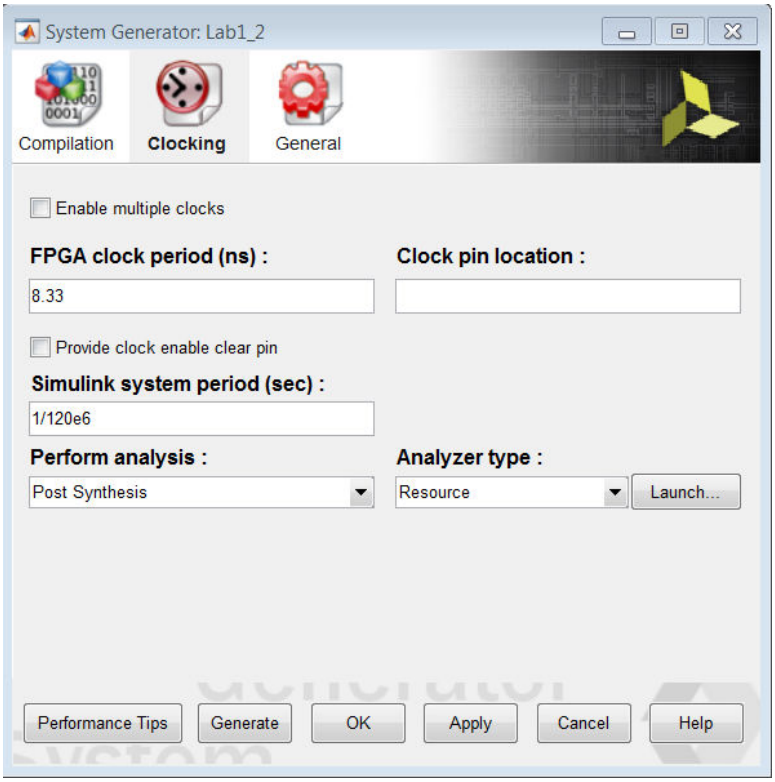


11. Click **OK** to save and exit the Properties Editor.

    In the System Generator token update the sampling frequency to 120 MHz (6 * 20 MHz) in this way:

1. Specify an FPGA clock period of 8.33 ns (1/120 MHz).
2. Specify a Simulink system period of 1/120e6 seconds.
3. From the Perform analysis menu, select **Post Synthesis** and from Analyzer type menu, select **Resource** as shown in the following figure. This option gives the resource utilization details after completion.

Note: In order to see accurate results from Resource Analyzer Window it is recommended to specify a new target directory rather than use the current working directory.



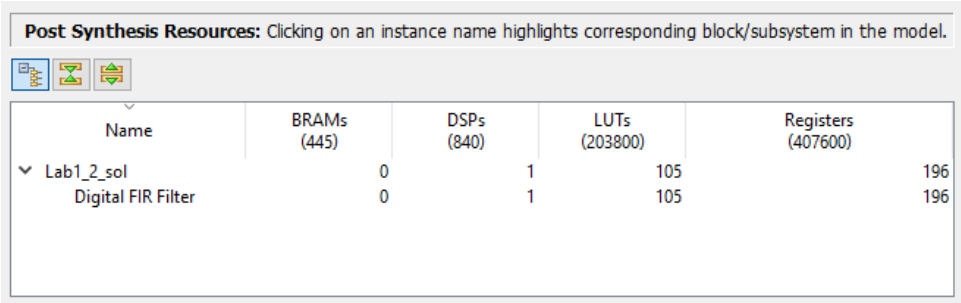
12. Click **Generate** to compile the design into a hardware description.

    In this case, the message appearing in the Diagnostic Viewer can be dismissed as you are purposely clocking the design above the sample rate to allow resource sharing and reduce resources. Close the Diagnostic Viewer window.

13. When generation completes, click **OK** to dismiss the Compilation status dialog box.

    The Resource Analyzer window opens when the generation completes, giving a good estimate of the final design results after synthesis as shown in the following figure.

    The hardware design now uses only a single DSP48 resource (a single multiplier) and compared to the results at the end of the Configure the HDL Blocks section, the resources used are significantly lower.



14. Click **OK** to dismiss the Resource Analyzer window.
15. Click **OK** to dismiss the System Generator token.

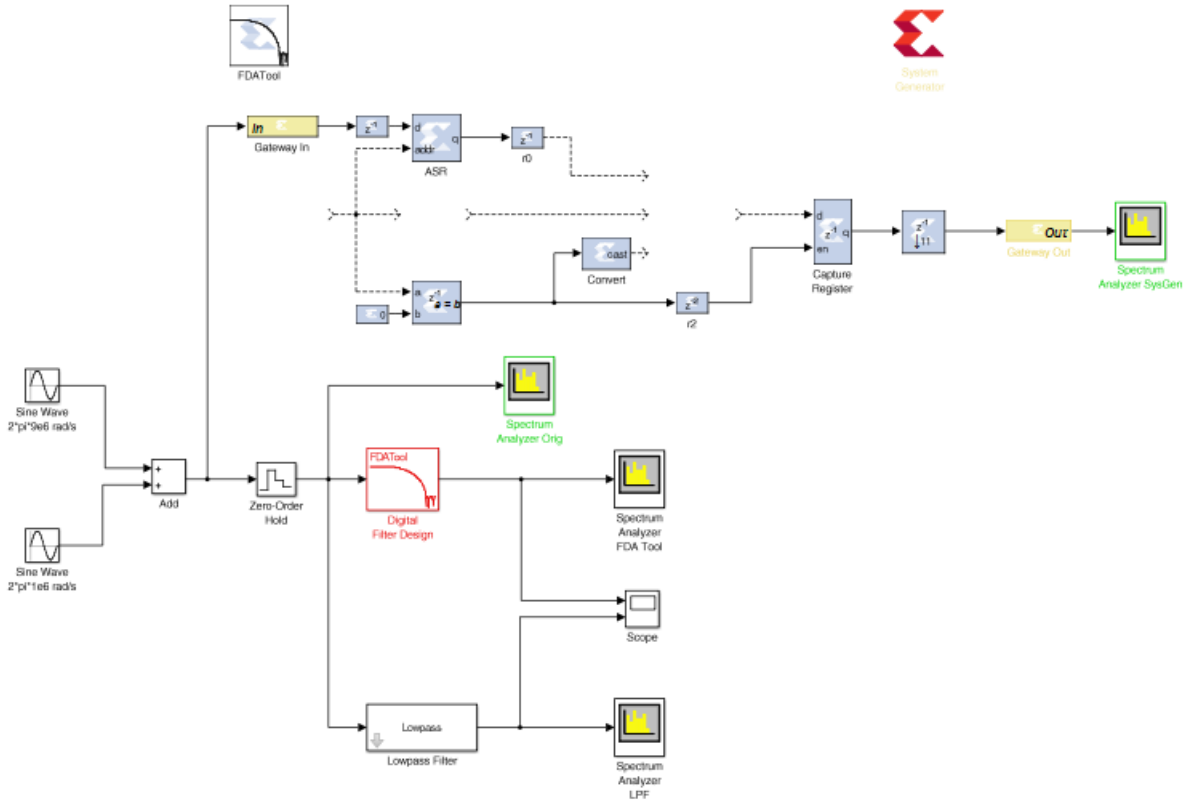
Exit the `Lab1_2.slx` Simulink worksheet.

Step 3: Creating a Design Using Discrete Components

In this step you will see how Model Composer can be used to build a design using discrete components to realize a very efficient hardware design.

1. At the command prompt, type `open Lab1_3.slx`.

   This opens the Simulink design shown in the following figure. This design is similar to the one in the previous two steps. However, this time the filter is designed with discrete components and is only partially complete. As part of this step, you will complete this design and learn how to add and configure discrete parts.

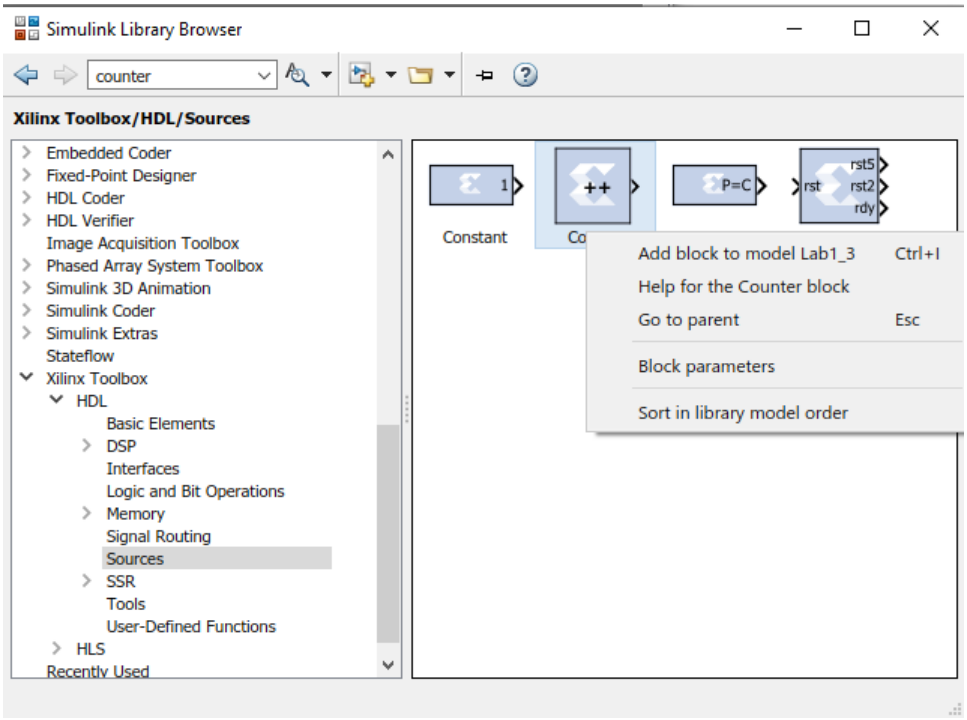


This discrete filter operates in this way:

- Samples arrive through port In and after a delay stored in a shift register (instance ASR).
- A ROM is required for the filter coefficients.
- A counter is required to select both the data and coefficient samples for calculation.
- A multiply accumulate unit is required to perform the calculations.
- The final down-sample unit selects an output every n^{th} cycle.

Start by adding the discrete components to the design.

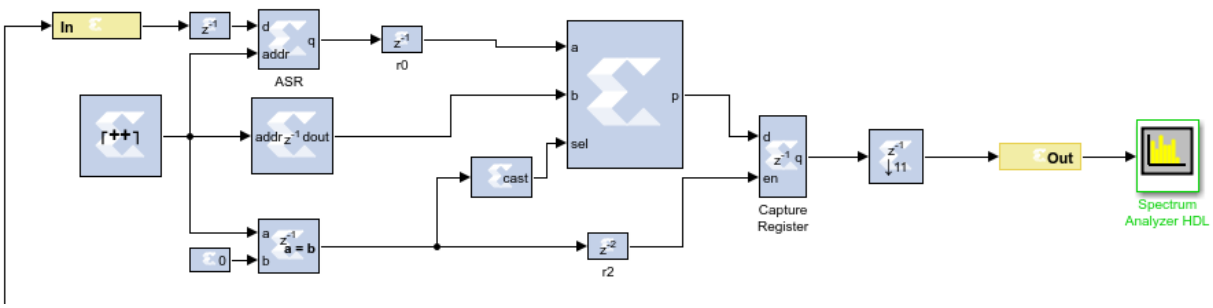
2. Click the Library Browser button in the Simulink toolbar to open the Simulink Library Browser.
   a. Expand the Xilinx Blockset menu.
   b. As shown in the following figure, select the **Sources** section in the HDL library, then right-click **Counter** to add this component to the design.



c. Select the **Memory** section (shown at the bottom left in the figure above) and add a ROM to the design.

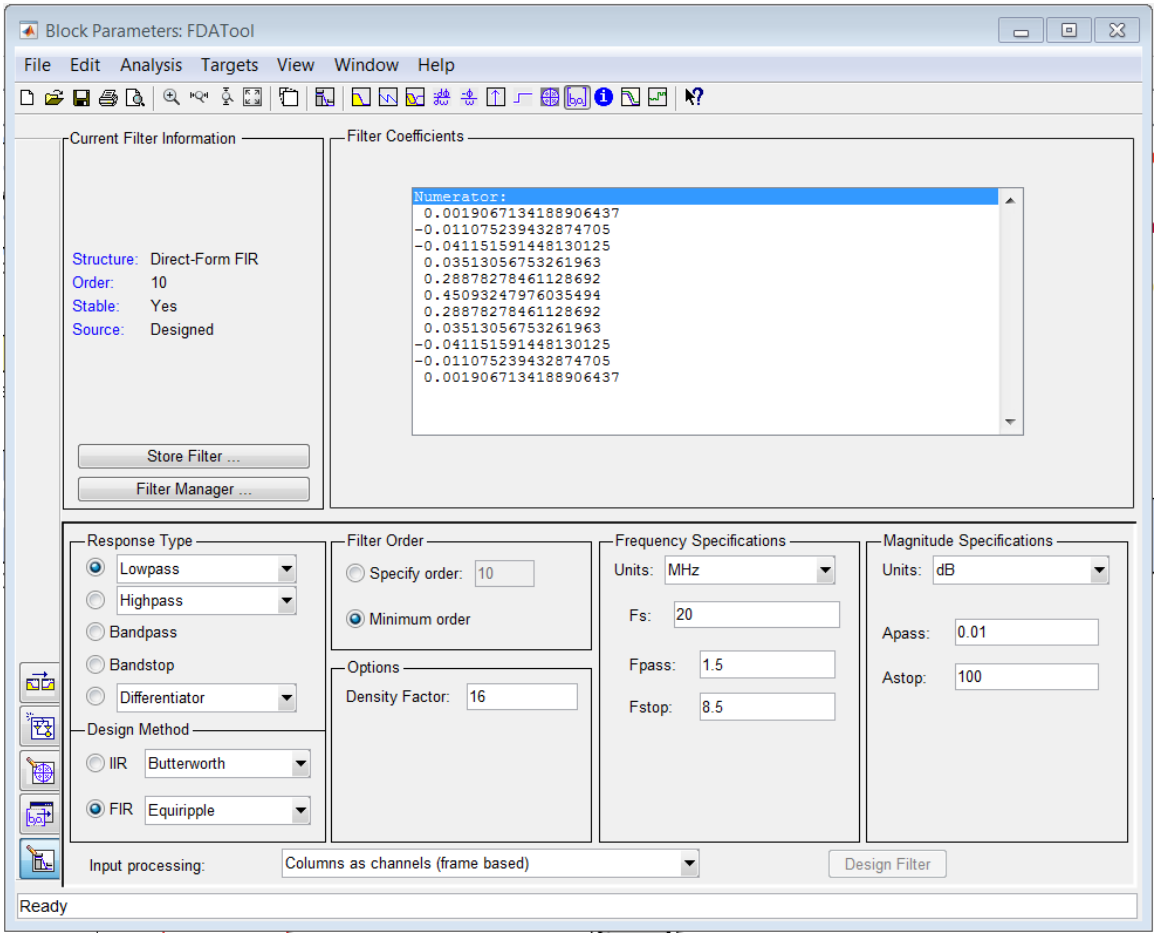
d. Finally, select the **DSP** section and add a DSP Macro 1.0 to the design.

3. Connect the three new instances to the rest of the design as shown in the following figure:



You will now configure the instances to correctly filter the data.

4. Double-click the **FDATool** instance and select Filter Coefficients from the toolbar to review the filter specifications.



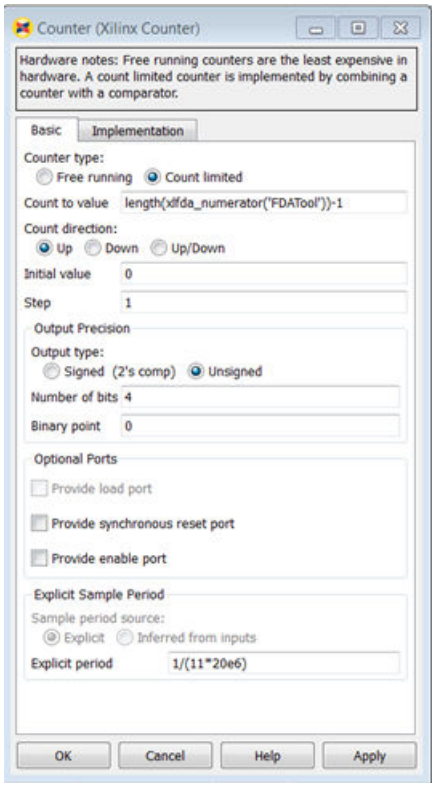
This shows the same specifications as the previous steps in Lab 1 and confirms there are 11 coefficients. You can also confirm, by double-clicking on the input Gateway In that the input sample rate is once again 20 MHz (Sample period = 1/20e6). With this information, you can now configure the discrete components.

5. Close the FDATool Properties Editor.
6. Double-click the **Counter** instance to open the Properties Editor.
   a. For the Counter type, select **Count limited** and enter this value for **Count to value**: `length(xlfda_numerator('FDATool'))-1`

      This will ensure the counter counts from 0 to 10 (11 coefficient and data addresses).

   b. For Output type, leave default value at Unsigned and in Number of Bits enter the value **4**. Only 4 binary address bits are required to count to 11.

   c. For the Explicit period, enter the value `1/(11*20e6)` to ensure the sample period is 11 times the input data rate. The filter must perform 11 calculations for each input sample.

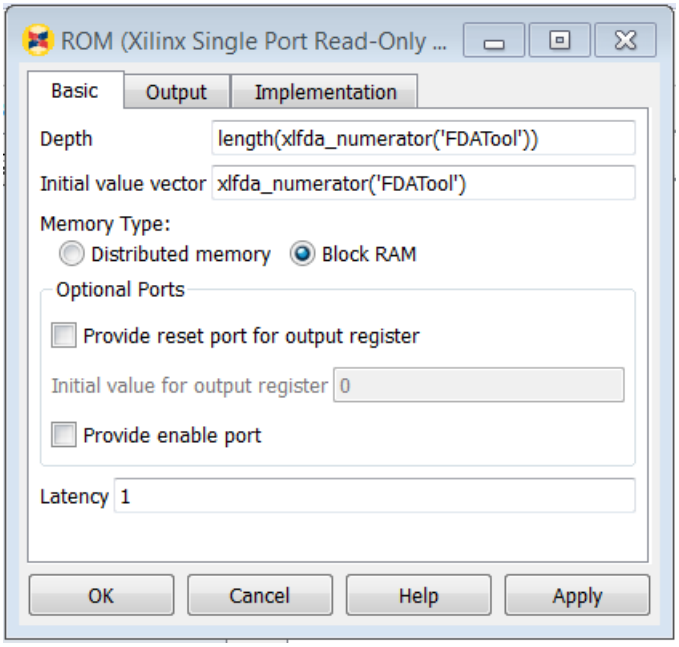


d. Click **OK** to exit the Properties Editor.

7. Double-click the **ROM** instance to open the Properties Editor.

   a. For the Depth, enter the value `length(xlfda_numerator('FDATool'))`. This will ensure the ROM has 11 elements.

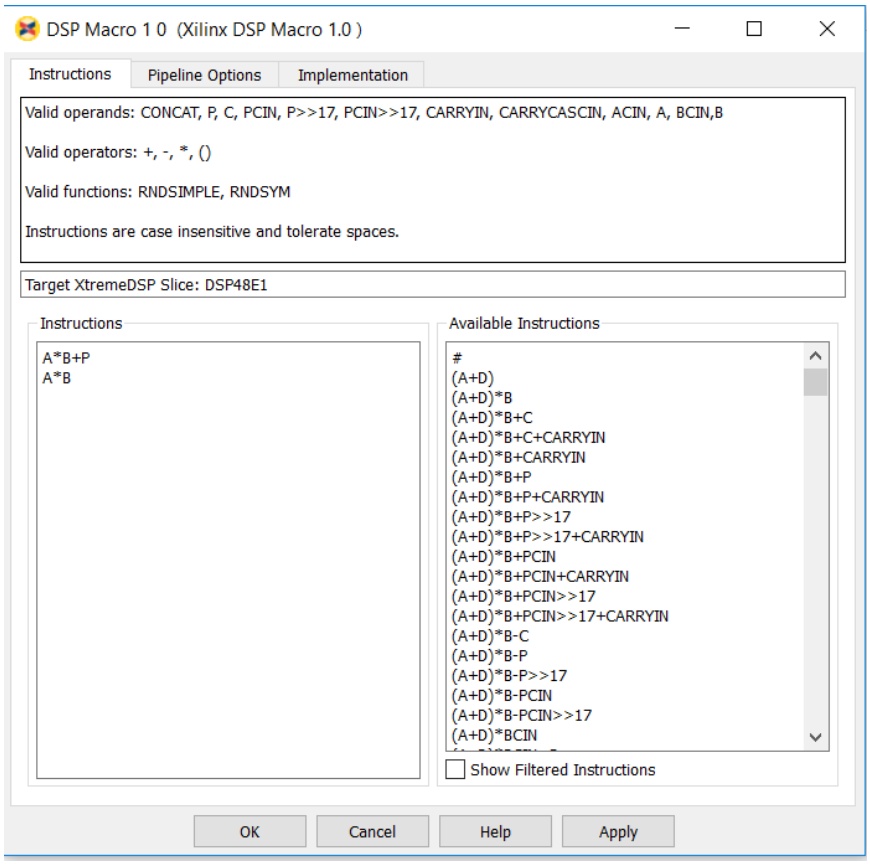
   b. For the Initial value vector, enter `xlfda_numerator('FDATool')`. The coefficient values will be provided by the FDATool instance.



c. Click **OK** to exit the Properties Editor.

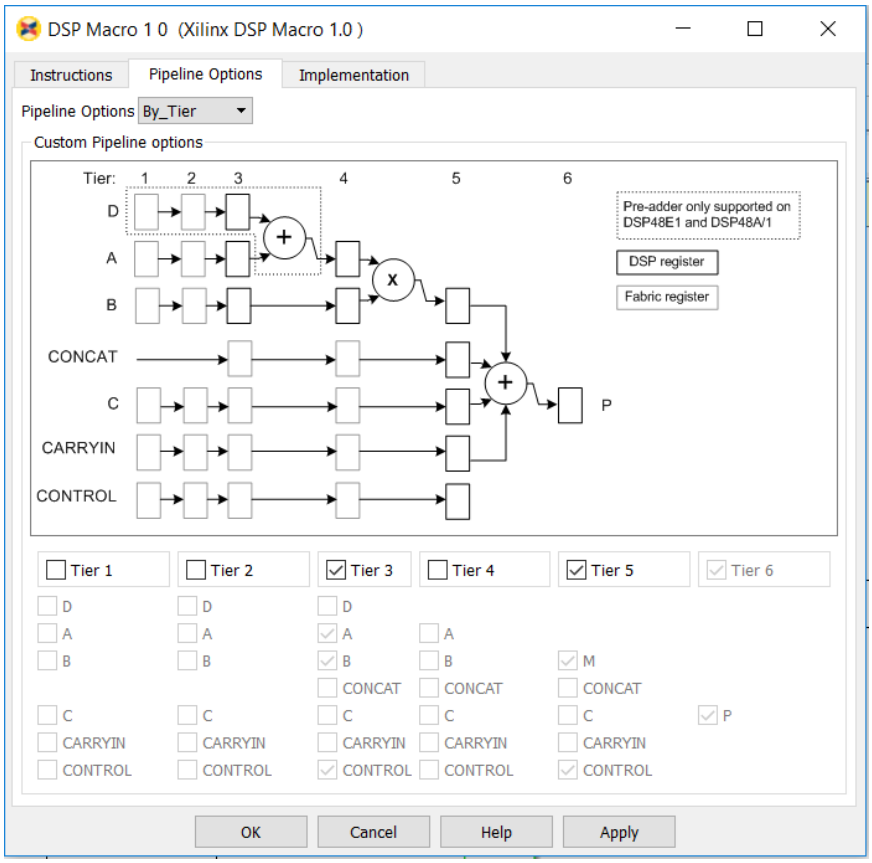
8. Double-click the **DSP Macro 1.0** instance to open the Properties Editor.

   a. In the Instructions tab, replace the existing Instructions with `A*B+P` and then add `A*B`. When the `sel` input is false the DSP will multiply and accumulate. When the `sel` input is true the DSP will simply multiply.



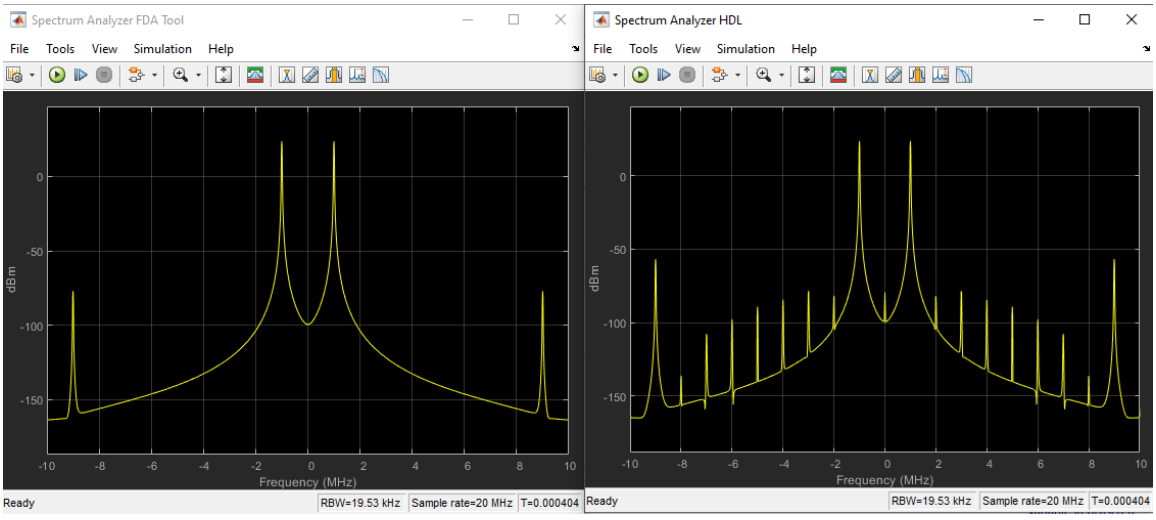
b. In the Pipeline Options tab, use the Pipeline Options drop-down menu to select **By_Tier**.

c. Select **Tier 3** and **Tier 5**. This will ensure registers are used at the inputs to A and B and between the multiply and accumulate operations.



d. Click **OK** to exit the Properties Editor.

9. Click **Save** to save the design.
10. Click the Run simulation button to simulate the design and view the results, as shown in the following figure.



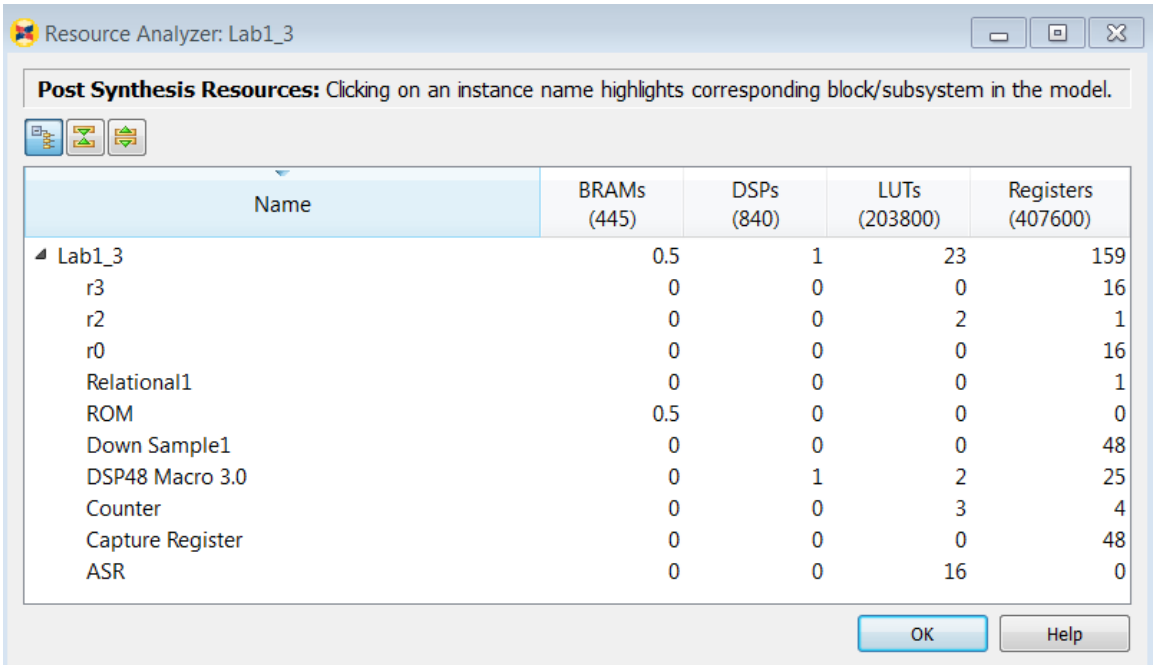
The final step is to compile the design into a hardware description and synthesize it.

11. Double-click the **System Generator** token to open the Properties Editor.

12. From the Compilation tab, make sure the Compilation target is IP catalog.
13. From the Clocking tab, under Perform analysis select **Post Synthesis** and for Analyzer type select **Resource**. This option gives the resource utilization details after completion.

    ***Note*:** In order to see accurate results from Resource Analyzer Window it is recommended to specify a new target directory rather than use the current working directory.

14. Click **Generate** to compile the design into a hardware description. After generation finishes, it displays the resource utilization in the Resource Analyzer window.



Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Lab1_3	0.5	1	23	159
r3	0	0	0	16
r2	0	0	2	1
r0	0	0	0	16
Relational1	0	0	0	1
ROM	0.5	0	0	0
Down Sample1	0	0	0	48
DSP48 Macro 3.0	0	1	2	25
Counter	0	0	3	4
Capture Register	0	0	0	48
ASR	0	0	16	0

    The design now uses fewer FPGA hardware resources than either of the versions designed with the Digital FIR Filter macro.

15. Click **OK** to dismiss the Resource Analyzer window.
16. Click **OK** to dismiss the Compilation status dialog box.
17. Click **OK** to dismiss the System Generator token.
18. Exit the `Lab1_3.slx` worksheet.

Step 4: Working with Data Types

In this step, you will learn how hardware-efficient fixed-point types can be used to create a design which meets the required specification but is more efficient in resources, and understand how to use Xilinx HDL Blocksets to analyze these systems.

This step has two primary parts:

- In Part 1, you will review and synthesize a design using floating-point data types.

- In Part 2, you will work with the same design, captured as a fixed-point implementation, and refine the data types to create a hardware-efficient design which meets the same requirements.

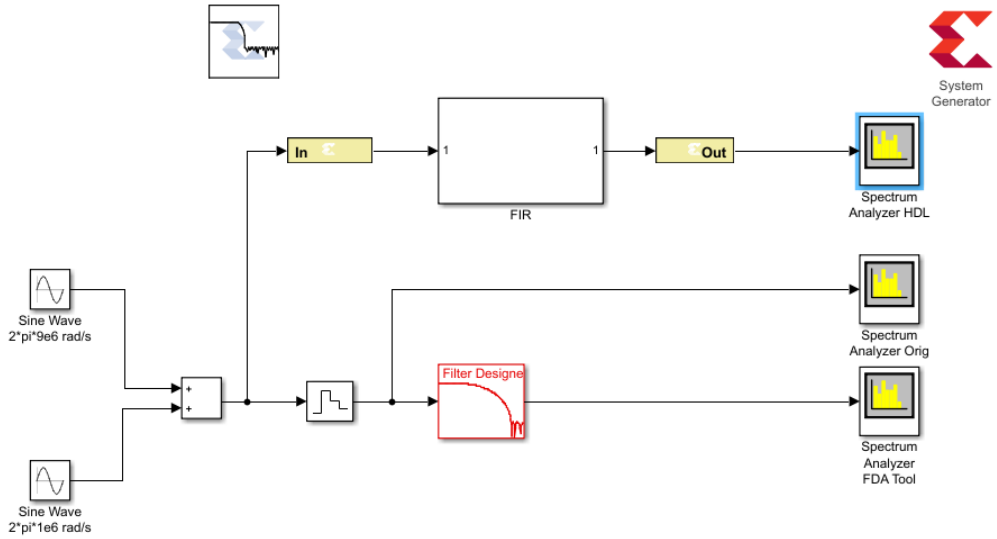
Part 1: Designing with Floating-Point Data Types

In this part you will review a design implemented with floating-point data types.

1. At the command prompt, type `open Lab1_4_1.slx`.

   This opens the Simulink design shown in the following figure. This design is similar to the design used in Lab 1_1, however this time the design is using float data types and the filter is implemented in sub-system FIR.

   First, you will review the attributes of the design, then simulate the design to review the performance, and finally synthesize the design.



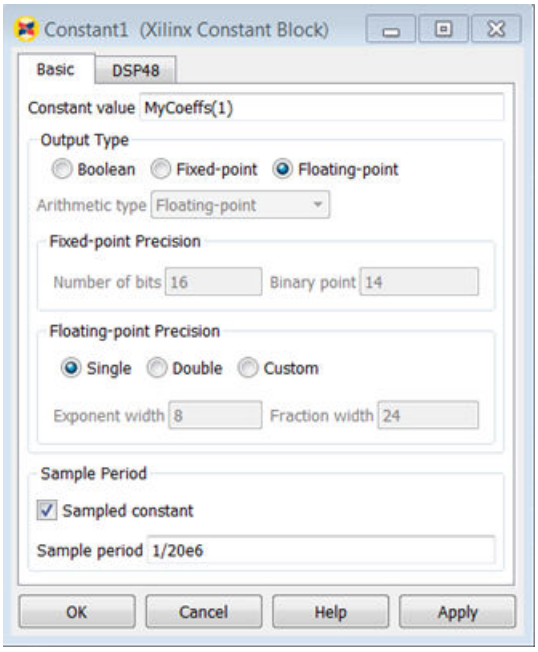
   In the previous figure, both the input and output of instance FIR are of type double.

2. In the MATLAB Command Window enter:

```
MyCoeffs = xlfda_numerator('FDATool')
```

3. Double-click the instance **FIR** to open the sub-system.
4. Double-click the instance **Constant1** to open the Properties Editor.

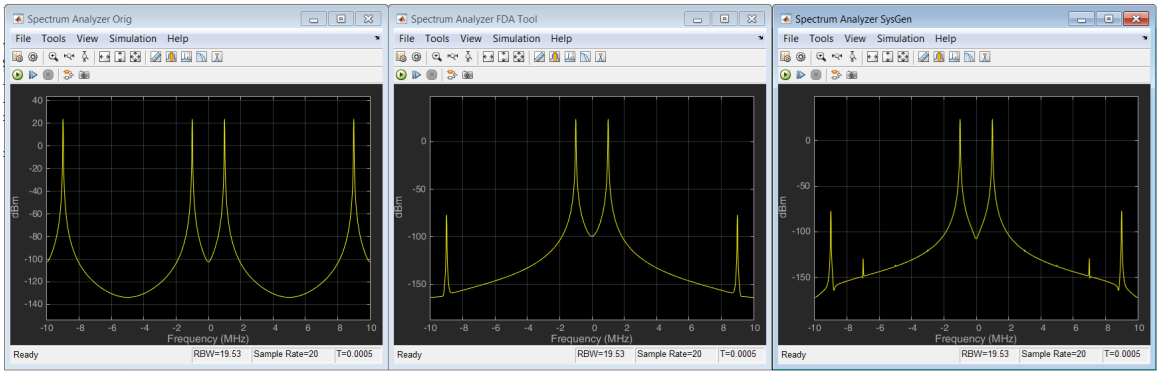
   This shows the Constant value is defined by `MyCoeffs(1)`.



5. Close the Constant1 Properties editor.
6. Return to the top-level design using the toolbar button Up To Parent , or click the tab labeled Lab1_4_1.

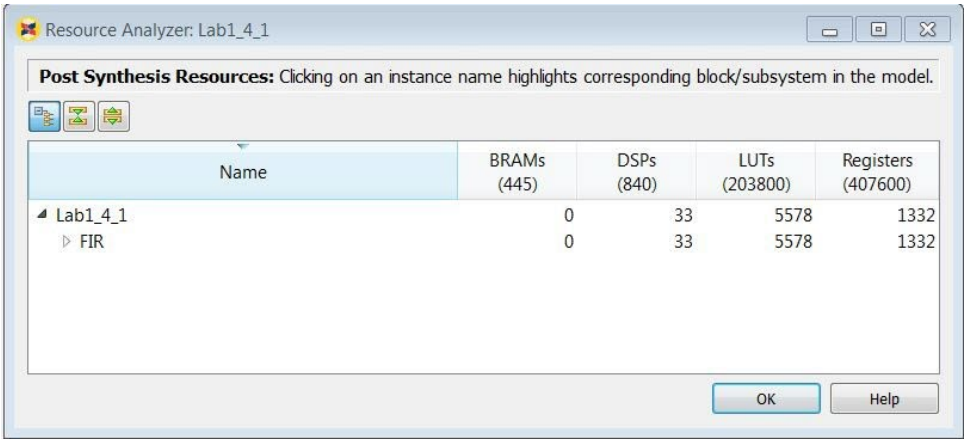
   The design is summing two sine waves, both of which are 9 MHz. The input gateway to the Model Composer must therefore sample at a rate of at least 18 MHz.
7. Double-click the **Gateway In1** instance to open the Properties Editor and confirm the input is sampling the data at a rate of 20 MHz (a Sample period of 1/20e6).
8. Close the Gateway In Properties editor.
9. Click the Run simulation button to simulate the design.

   The results shown in the following figure show the Model Composer HDL blockset produces results which are very close to the ideal case, shown in the center. The results are not identical because the Model Composer design must sample the continuous input waveform into discrete time values.



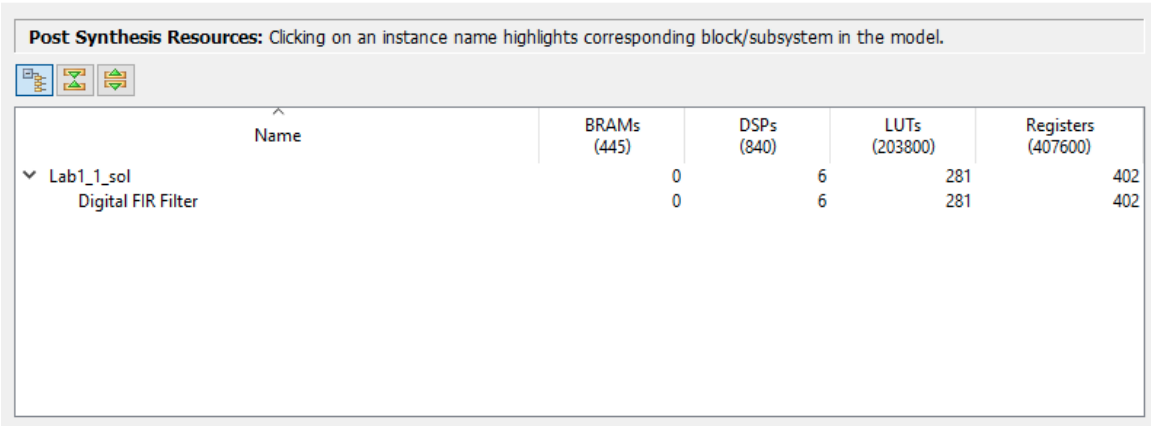
The final step is to synthesize this design into hardware.

10. Double-click the **System Generator** token to open the Properties Editor.
11. On the Compilation tab, make sure the Compilation target is IP Catalog.
12. On the Clocking tab, under Perform analysis select **Post Synthesis** and from Analyzer type menu select **Resource**. This option gives the resource utilization details after completion.
13. Click **Generate** to compile the design into a hardware description. After completion, it generates the resource utilization in Resource Analyzer window as shown in the following figure.



14. Click **OK** to dismiss the Compilation status dialog box.
15. Click **OK** to dismiss the System Generator token.

You implemented this same filter in Step 1 using fixed-point data types. When compared to the synthesis results from that implementation – the initial results from Step are shown in the following figure and you can see this current version of the design is using a large amount of registers (FF), LUTs, and DSP48 (DSP) resources (Xilinx dedicated multiplier/add units).



Maintaining the full accuracy of floating-point types is an ideal implementation but implementing full floating-point accuracy requires a significant amount of hardware.

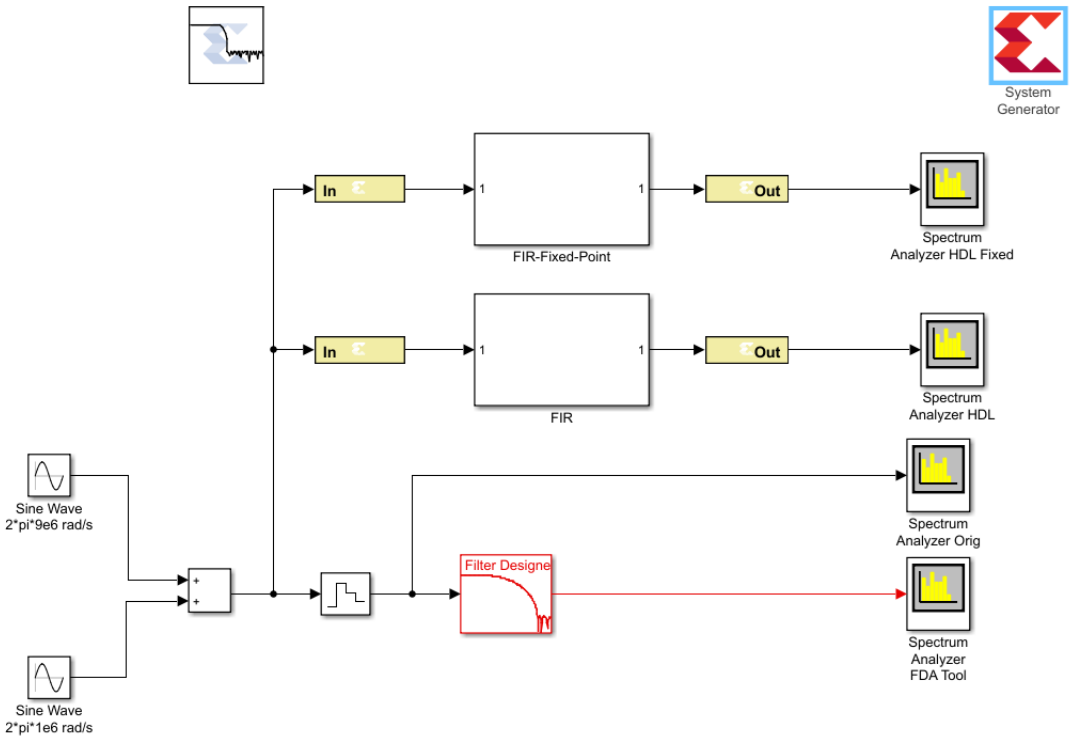
For this particular design, the entire range of the floating-point types is not required. The design is using considerably more resources than what is required. In the next part, you will learn how to compare designs with different data types inside the Simulink environment.

16. Exit the Vivado Design Suite.
17. Exit the `Lab1_4_1.slx` Simulink worksheet.

Part 2: Designing with Fixed-Point Data Types

In this part you will re-implement the design from Part 1: Designing with Floating-Point Data Types using fixed-point data types, and compare this new design with the original design. This exercise will demonstrate the advantages and disadvantages of using fixed-point types and how Model Composer allows you to easily compare the designs, allowing you to make trade-offs between accuracy and resources within the Simulink environment before committing to an FPGA implementation.

1. At the command prompt, type `open Lab1_4_2.slx` to open the design shown in the following figure.



2. In the MATLAB Command Window enter:

```
MyCoeffs = xlfda_numerator('FDATool')
```

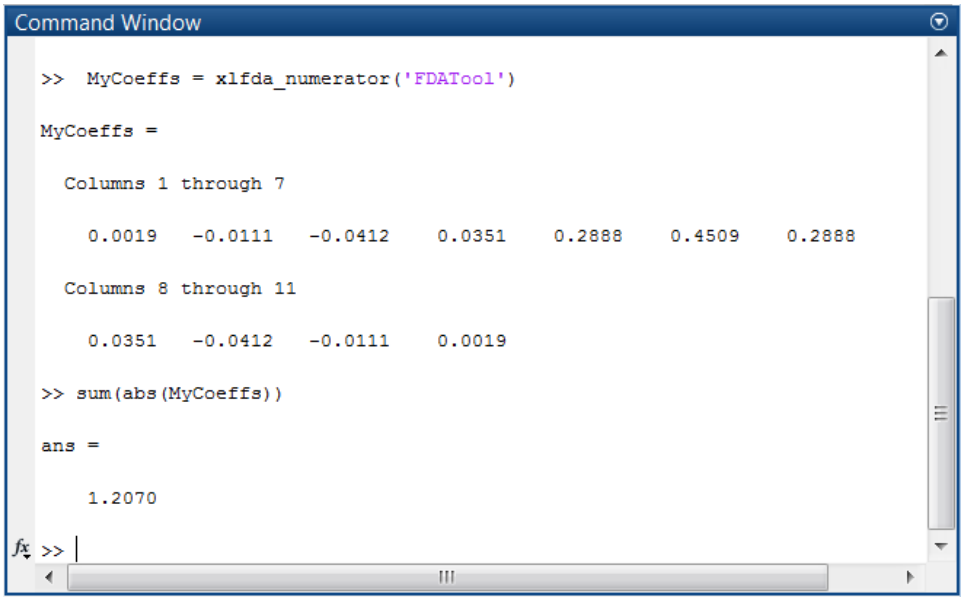
3. Double-click the instance **Gateway In2** to confirm the data is being sampled as 16-bit fixed-point value.
4. Click **Cancel** to exit the Properties Editor.

5. Click the Run simulation button to simulate the design and confirm instance Spectrum Analyzer HDL Fixed shows the filtered output.

   As you will see if you examine the output of instance FIR-Fixed-Point (shown in the previous figure) Model Composer has automatically propagated the input data type through the filter and determined the output must be 43-bit (with 28 binary bits) to maintain the resolution of the signal.

   This is based on the bit-growth through the filter and the fact that the filter coefficients (constants in instance FIR-Fixed-Point) are 16-bit.

6. In the MATLAB Command Window, enter `sum(abs(MyCoeffs))` to determine the absolute maximum gain using the current coefficients.

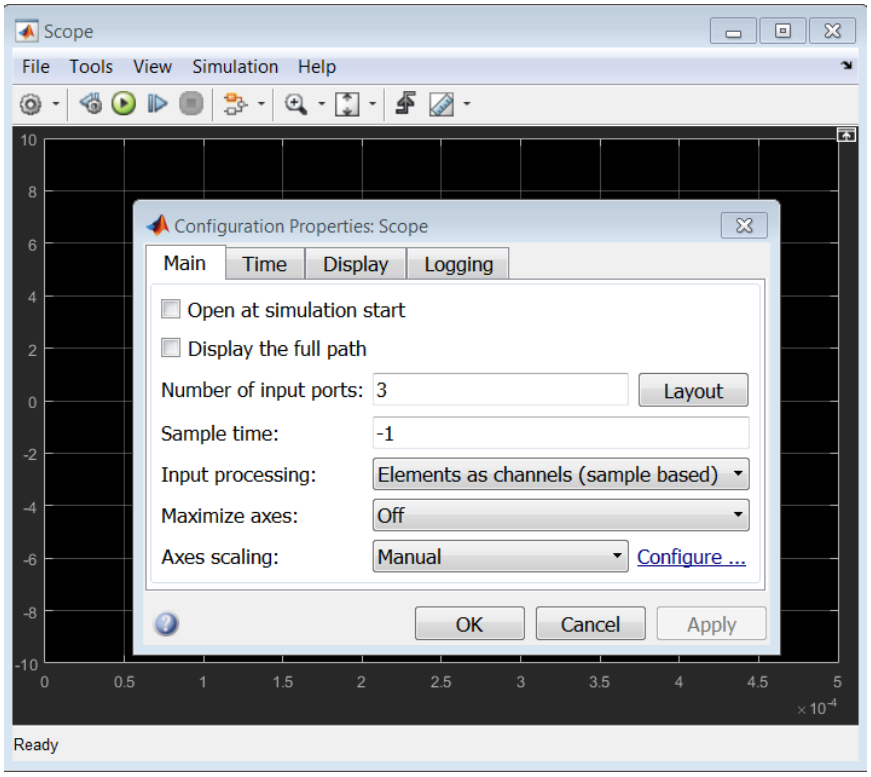


   Taking into account the positive and negative values of the coefficients the maximum gain possible is 1.2070 and the output signal should only ever be slightly smaller in magnitude than the input signal, which is a 16-bit signal. There is no need to have 15 bits (43-28) of data above the binary point.

   You will now use the Reinterpret and Convert blocks to manipulate the fixed-point data to be no greater than the width required for an accurate result and produce the most hardware efficient design.

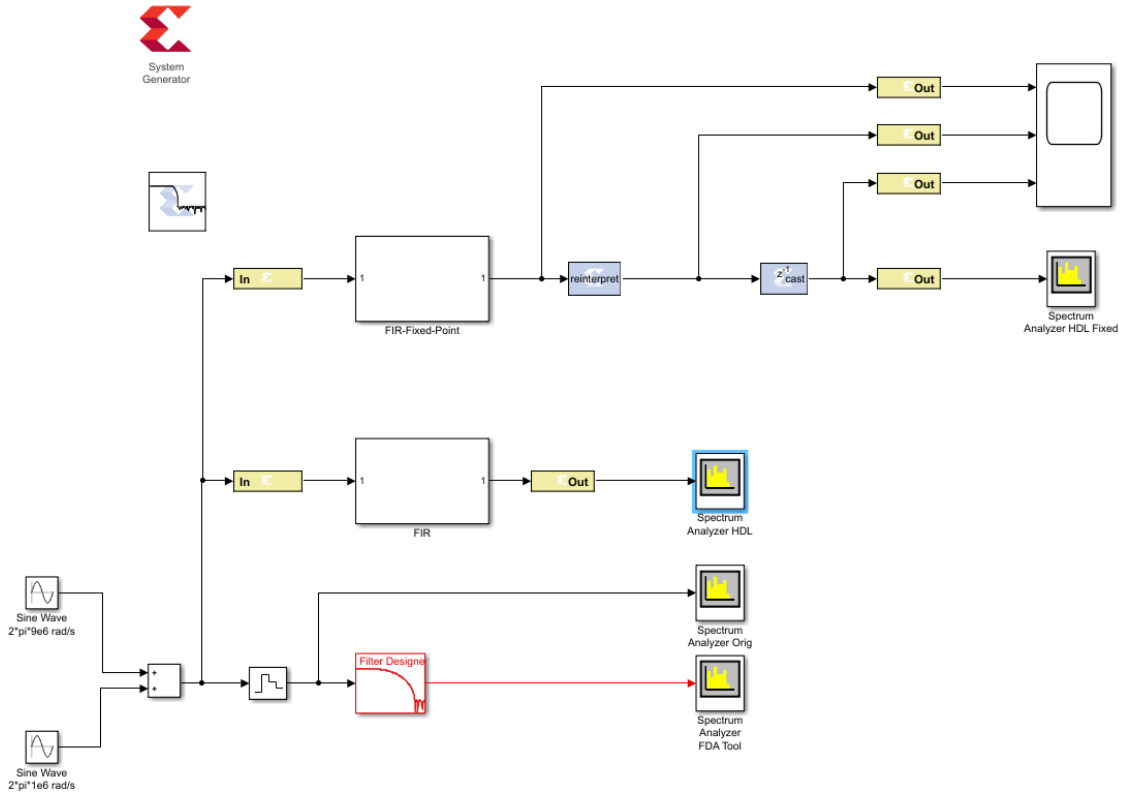
7. Right-click with the mouse anywhere in the canvas and select **Xilinx BlockAdd**.
8. In the Add Block entry box, type `Reinterpret`.
9. Double-click the **Reinterpret** component to add it to the design.
10. Repeat the previous three steps for these components:
    a. Convert
    b. Scope

11. In the design, select the **Gateway Out2** instance.
    a. Right-click and use Copy and Paste to create a new instance of the Gateway Out block.
    b. Paste twice again to create two more instances of the Gateway Out (for a total of three new instances).
12. Double-click the **Scope** component.
    a. In the Scope properties dialog box, select **File → Number of Inputs → 3**.
    b. Select **View → Configuration Properties** and confirm that the Number of input ports is 3.



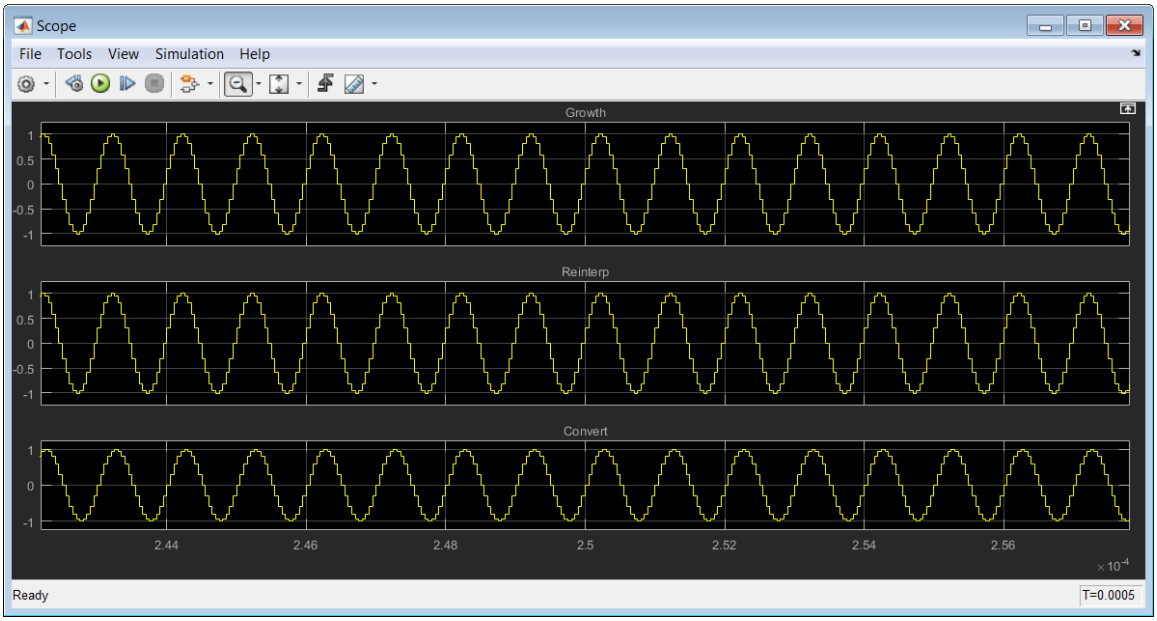
    c. Click **OK** to close the Configuration Properties dialog box.
    d. Select **File → Close** to close the Scope properties dialog box.
13. Connect the blocks as shown in the next figure.
14. Rename the signal names into the scope as shown in the following figure: Convert, Reinterpret and Growth.

    To rename a signal, click the existing name label and edit the text, or if there is no text double-click the wire and type the name.



15. Click the Run simulation button to simulate the design.
16. Double-click the **Scope** to examine the signals.

> **TIP:** *You might need to zoom in and adjust the scale in **View→Configuration Properties** to view the signals in detail.*



The Reinterpret and Convert blocks have not been configured at this point and so all three signals are identical.

The HDL Reinterpret block forces its output to a new type without any regard for retaining the numerical value represented by the input. The block allows for unsigned data to be reinterpreted as signed data, or, conversely, for signed data to be reinterpreted as unsigned. It also allows for the reinterpretation of the data's scaling, through the repositioning of the binary point within the data.

In this exercise you will scale the data by a factor of 2 to model the presence of additional design processing which might occur in a larger system. The Reinterpret block can also be used to scale down.

17. Double-click the **Reinterpret** block to open the Properties Editor.
18. Select **Force Binary Point**.
19. Enter the value `27` in the input field Output Binary Point and click **OK**.

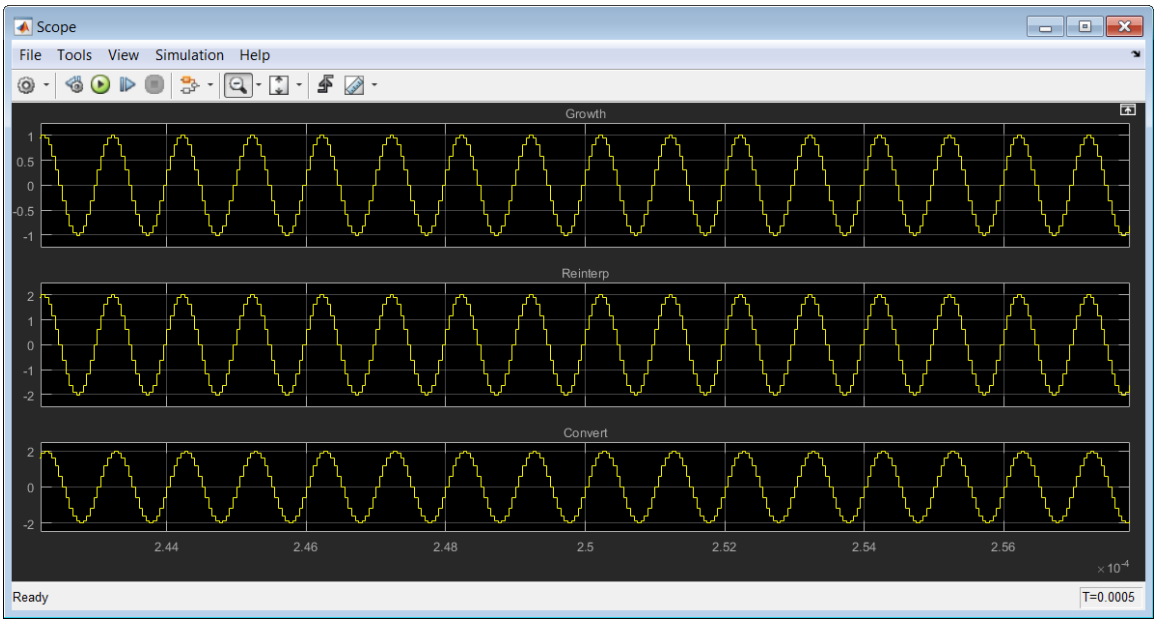
    The HDL Convert block converts each input sample to a number of a desired arithmetic type. For example, a number can be converted to a signed (two's complement) or unsigned value. It also allows the signal quantization to be truncated or rounded and the signal overflow to be wrapped, saturated, or to be flagged as an error.

    In this exercise, you will use the Convert block to reduce the size of the 43-bit word back to a 16-bit value. In this exercise the Reinterpret block has been used to model a more complex design and scaled the data by a factor of 2. You must therefore ensure the output has enough bits above the binary point to represent this increase.
20. Double-click the **Convert** block to open the Properties Editor.
21. In the Fixed-Point Precision section, enter `13` for the Binary Point and click **OK**.
22. Save the design.
23. Click the Run simulation button to simulate the design.
24. Double-click the **Scope** to examine the signals.

TIP: *You might need to zoom in and adjust the scale in **View→Configuration Properties** to view the signals in detail.*

In the following figure you can see the output from the filter (Growth) has values between plus and minus 1. The output from the Reinterpret block moves the data values to between plus and minus 2.

In this detailed view of the waveform, the final output (Convert) shows no difference in fidelity, when compared to the reinterpret results, but uses only 16 bits.

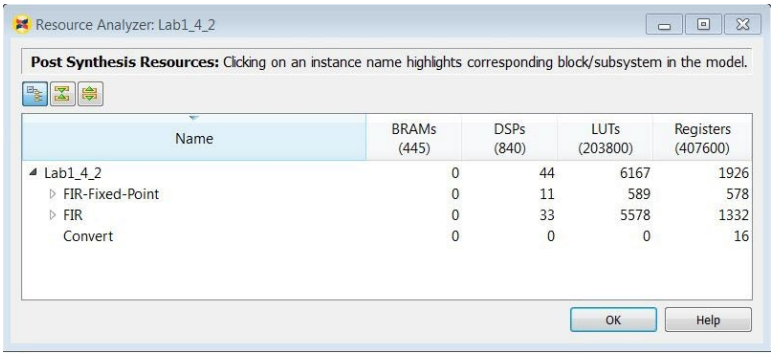


The final step is to synthesize this design into hardware.

25. Double-click the System Generator token to open the Properties Editor.
26. On the Compilation tab, ensure the Compilation target is IP catalog.
27. On the Clocking tab, under Perform analysis select **Post Synthesis** and from Analyzer type menu select **Resource**. This option gives the resource utilization details after completion.

    Note: In order to see accurate results from Resource Analyzer Window it is recommended to specify a new target directory rather than use the current working directory.

28. Click **Generate** to compile the design into a hardware description. After completion, it generates the resource utilization in Resource Analyzer window as shown in the following figure.



29. Click **OK** to dismiss the Compilation status dialog box.
30. Click **OK** to dismiss the System Generator token.

    Notice, as compared to the results in Step 1, these results show approximately:

    - 45% more Flip-Flops

- 20% more LUTs
- 30% more DSP48s

However, this design contains both the original floating-point filter and the new fixed-point version: the fixed-point version therefore uses approximately 75-50% fewer resources with the acceptable signal fidelity and design performance.

31. Exit Vivado.
32. Exit the `Lab1_4_2.slx` worksheet.

Summary

In this lab, you learned how to use the Model Composer HDL blockset to create a design in the Simulink environment and synthesize the design in hardware which can be implemented on a Xilinx FPGA. You learned the benefits of quickly creating your design using a Xilinx Digital FIR Filter block and how the design could be improved with the use of over-sampling.

You also learned how floating-point types provide a high degree of accuracy but cost many more resources to implement in an FPGA and how the Model Composer HDL blockset can be used to both implement a design using more efficient fixed-point data types and compensate for any loss of accuracy caused by using fixed-point types.

The Reinterpret and Convert blocks are powerful tools which allow you to optimize your design without needing to perform detailed bit-level optimizations. You can simply use these blocks to convert between different data types and quickly analyze the results.

Finally, you learned how you can take total control of the hardware implementation by using discrete primitives.

***Note*:** In this tutorial you learned how to add Model Composer HDL blocks to the design and then configure them. A useful productivity technique is to add and configure the System Generator token first. If the target device is set at the start, some complex IP blocks will be automatically configured for the device when they are added to the design.

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

```
/HDL_Library/Lab1/solution
```

Lab 2: Importing Code into Model Composer HDL Design

Objectives

After completing this lab, you will be able to:

- Create a Finite State Machine using the MCode block in Model Composer.
- Import an RTL HDL description into Model Composer.
- Configure the black box to ensure the design can be successfully simulated.
- Incorporate a design, synthesized from C, C++ or SystemC using Vitis HLS, as a block into your MATLAB design.

Step 1: Modeling Control with M-Code

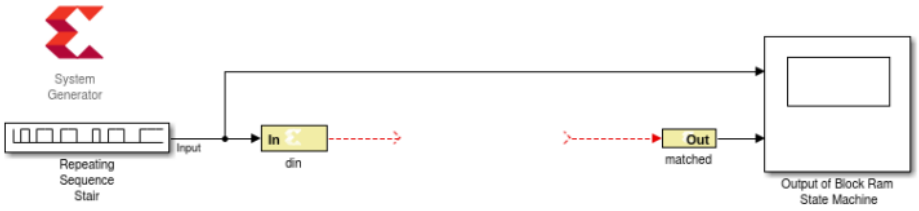
In this step you will be creating a simple Finite State Machine (FSM) using the MCode block to detect a sequence of binary values 1011. The FSM needs to be able to detect multiple transmissions as well, such as 10111011.

Procedure

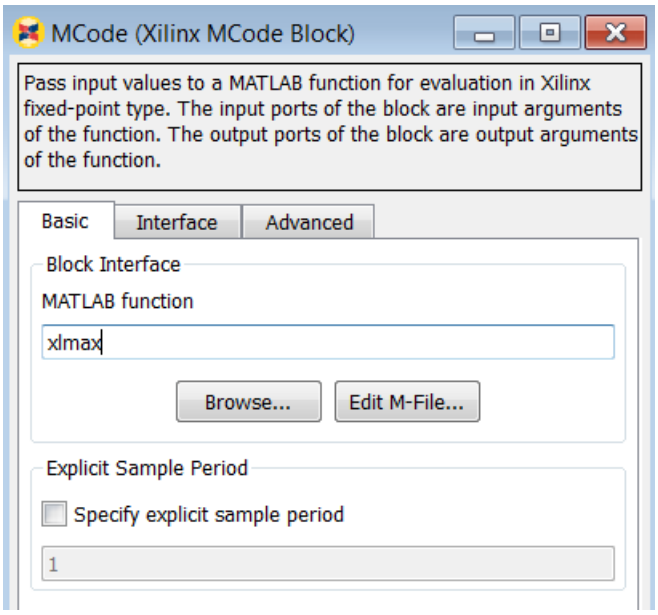
In this step you will create the control logic for a Finite State Machine using M-code. You will then simulate the final design to confirm the correct operation.

1. Launch Model Composer and change the working directory to: **\HDL_Library\Lab2\M_code**
2. Open the file `Lab2_1.slx`.

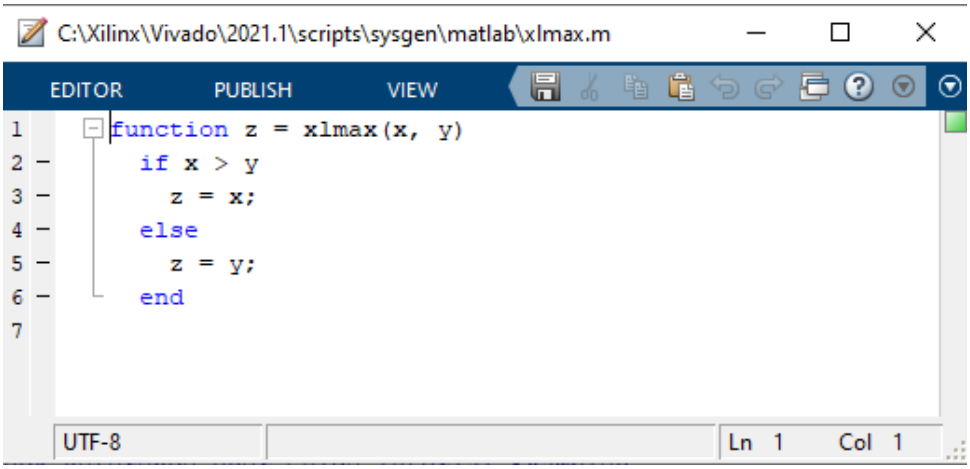
   You see the following incomplete diagram.



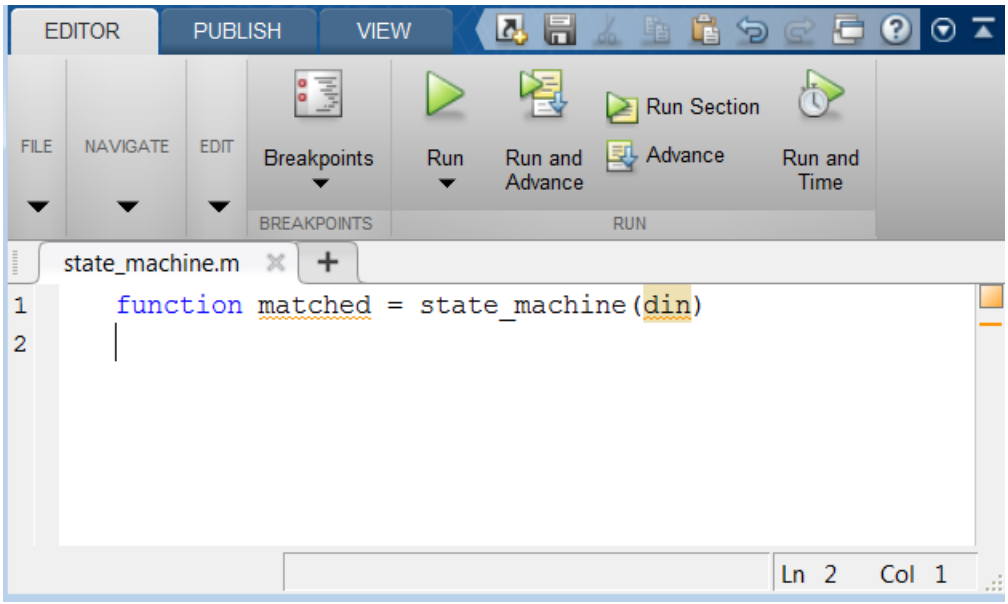
3. Add an MCode block from theXilinx Toolbox/HDL/User-Defined Functions library. Before wiring up the block, you need to edit the MATLAB® function to create the correct ports and function name.
4. Double-click the **MCode** block and click **Edit M-File**, as shown in the following figure.



The following figure shows the default M-code in the MATLAB text editor.

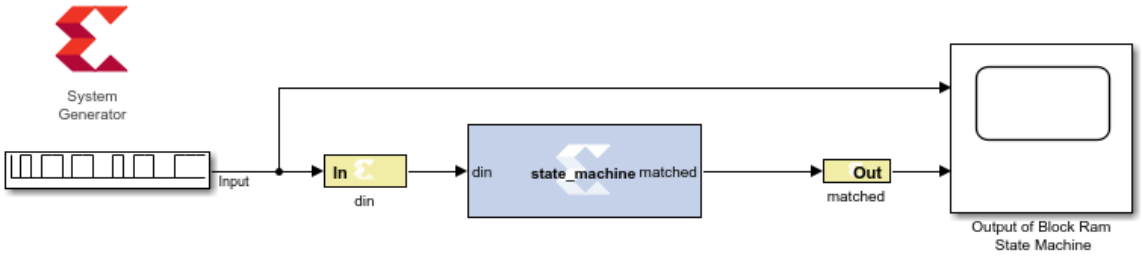


5. Edit the default MATLAB function to include the function name `state_machine` and the input `din` and output `matched`.
6. You can now delete the sample M-code.

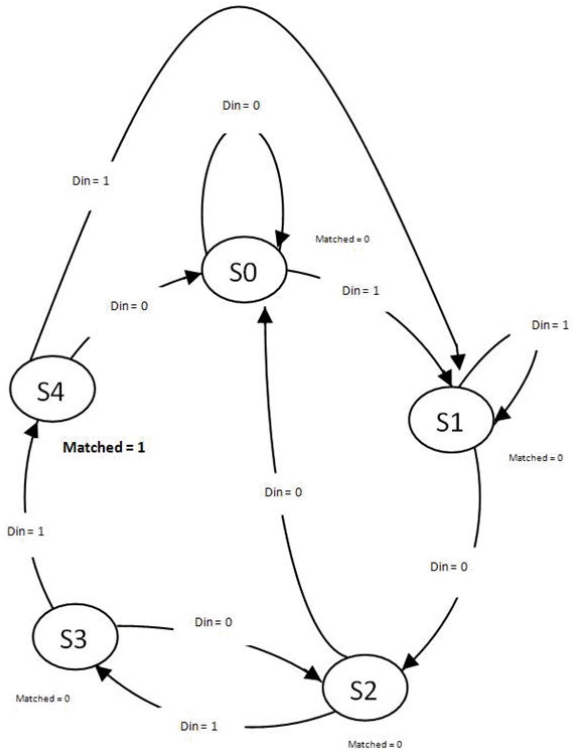


7. After you make the edits, use Save As to save the MATLAB file as `state_machine.m` to the `Lab2/M_code` folder.
   a. In the MCode Properties Editor, use the Browse button to ensure that the MCode block is referencing the local M-code file (`state_machine.m`).
8. In the MCode Properties Editor, click **OK**.

   You will see the MCode block assume the new ports and function name.
9. Now connect the MCode block to the diagram as shown in the following figure:



You are now ready to start coding the state machine. The bubble diagram for this state machine is shown in the following figure. This FSM has five states and is capable of detecting two sequences in succession.



10. Edit the M-code file, `state_machine.m`, and define the state variable using the Xilinx `xl_state` data type as shown in the following. This requires that you declare a variable as a persistent variable. The `xl_state` function requires two arguments: the initial condition and a fixed-point declaration.

    Because you need to count up to 4, you need 3 bits.

    ```
    persistent state, state = xl_state(0,{xlUnsigned, 3, 0});
    ```

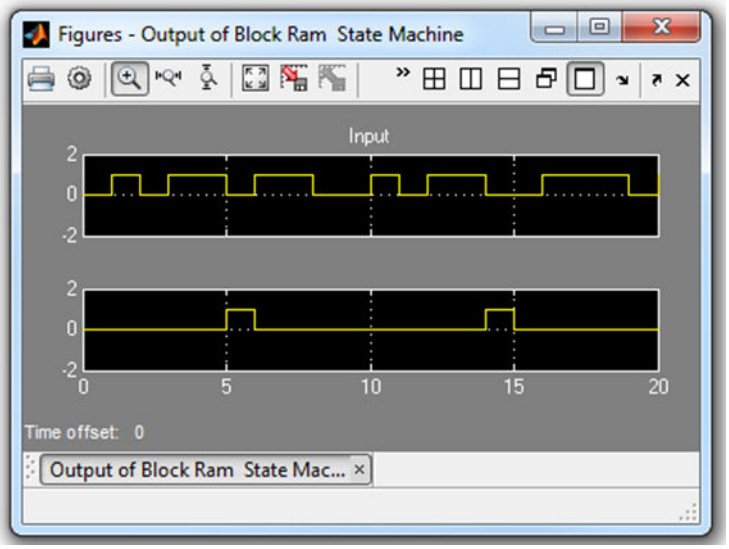
11. Use a switch-case statement to define the FSM states shown. A small sample is provided, shown as follows, to get you started.

    ***Note*:** You need an otherwise statement as your last case.

    ```
    switch state
       case 0
          if din == 1
              state = 1
          else
              state = 0
          end
          matched = 0;
    ```

12. Save the M-code file and run the simulation. The waveform should look like the following figure.

    You should notice two detections of the sequence.



Step 2: Modeling Blocks with HDL

In this step, you will import an RTL design into Model Composer as a black box.

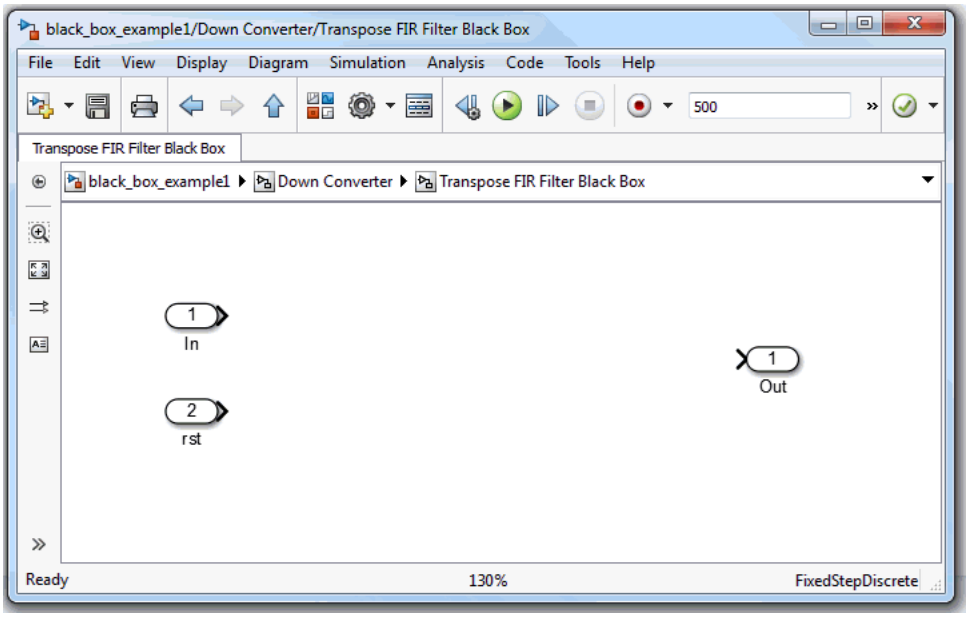
A black box allows the design to be imported into Model Composer even though the description is in Hardware Description Language (HDL) format.

1. Invoke Vitis Model Composer and from the MATLAB console, change the directory to: `\HDL_Library\Lab2\HDL`.

   The following files are located in this directory:

   - `Lab2_2.slx` - A Simulink model containing a black box example.
   - `transpose_fir.vhd` - Top-level VHDL for a transpose form FIR filter. This file is the VHDL that is associated with the black box.
   - `mac.vhd` – Multiply and adder component used to build the transpose FIR filter.
2. Type `open Lab2_2.slx`.
3. Open the subsystem named Down Converter.
4. Open the subsystem named Transpose FIR Filter Black Box.

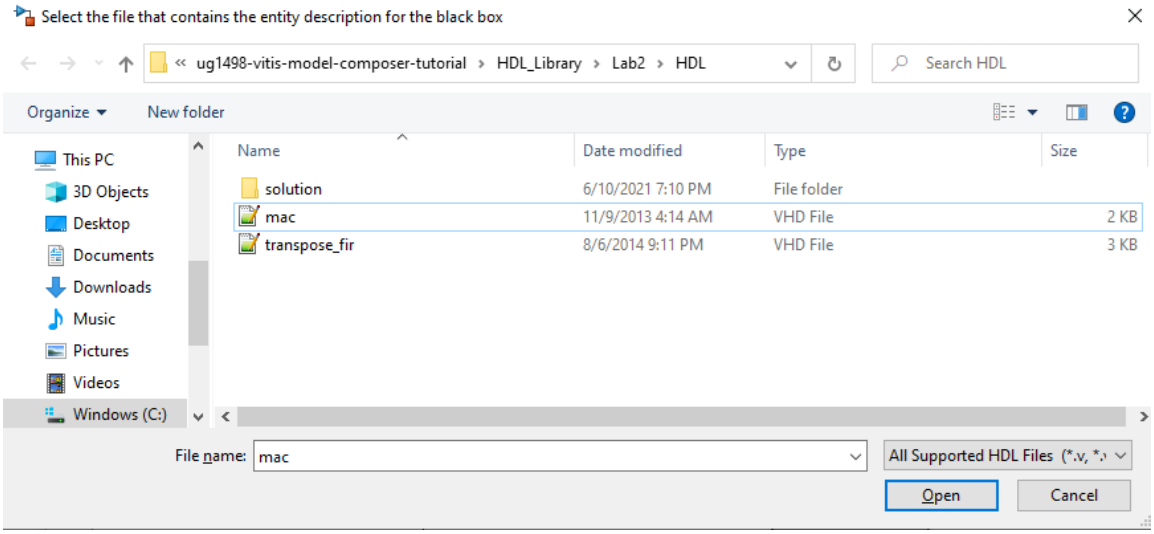
   At this point, the subsystem contains two input ports and one output port. You will add a black box to this subsystem:



5. Right-click the design canvas, select **Xilinx BlockAdd**, and add a Black Box block to this subsystem.

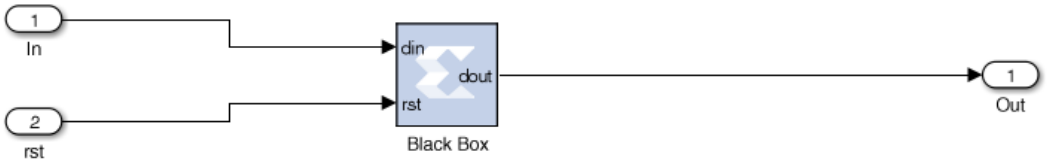
   A browser window opens, listing the VHDL source files that can be associated with the black box.

6. From this window, select the top-level VHDL file `transpose_fir.vhd`. This is illustrated in the following figure.

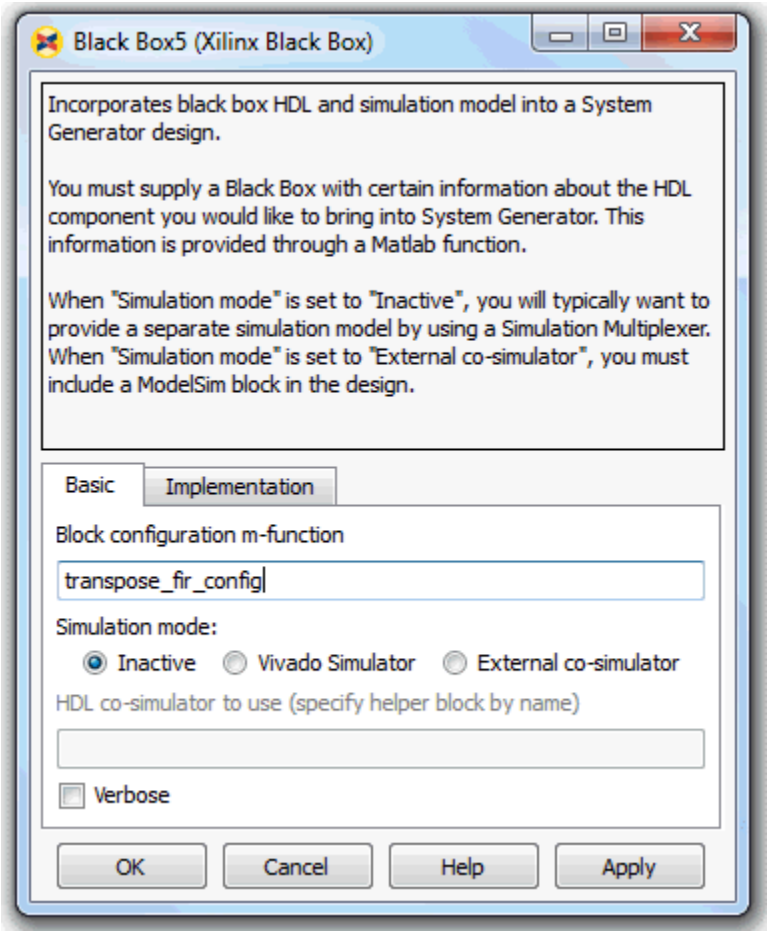


   The associated configuration M-code `transpose_fir_config.m` opens in an Editor for modifications.

7. Close the Editor.
8. Wire the ports of the black box to the corresponding subsystem ports and save the design.



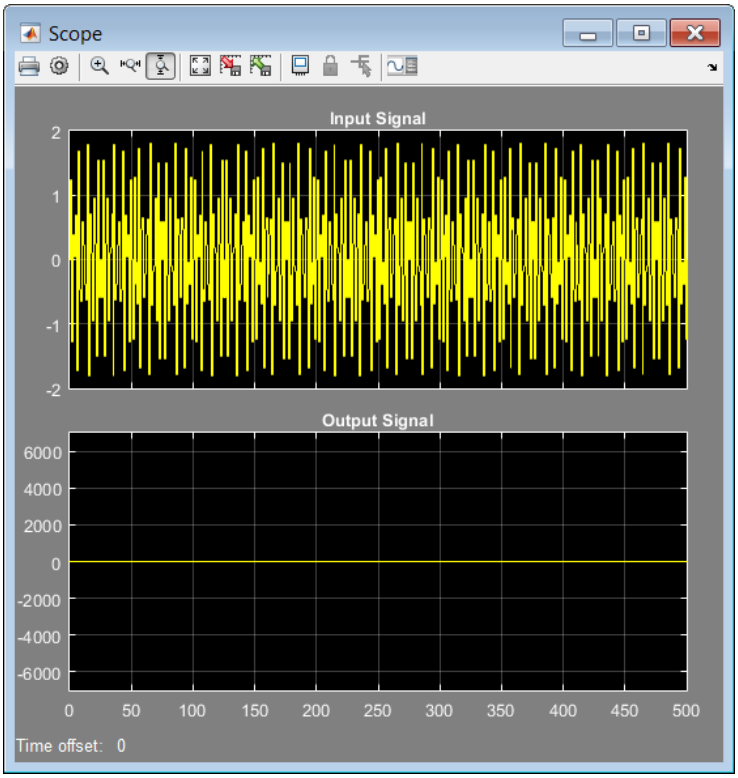
9. Double-click the Black Box block to open this dialog box:



The following are the fields in the dialog box:

- **Block configuration m-function:** This specifies the name of the configuration M-function for the black box. In this example, the field contains the name of the function that was generated by the Configuration Wizard. By default, the black box uses the function the wizard produces. You can however substitute one you create yourself.
- **Simulation mode:** There are three simulation modes:
  - **Inactive:** In this mode the black box participates in the simulation by ignoring its inputs and producing zeros. This setting is typically used when a separate simulation model is available for the black box, and the model is wired in parallel with the black box using a simulation multiplexer.
  - **Vivado Simulator:** In this mode simulation results for the black box are produced using co-simulation on the HDL associated with the black box.
  - **External co-simulator:** In this mode it is necessary to add a Questa HDL co-simulation block to the design, and to specify the name of the Questa block in the HDL co-simulator to use field. In this mode, the black box is simulated using HDL co-simulation.

10. Set the Simulation mode to Inactive and click **OK** to close the dialog box.
11. Move to the design top-level and run the simulation by clicking the Run simulation button  , then double-click the Scope block.
12. Notice the black box output shown in the Output Signal scope is zero. This is expected because the black box is configured to be Inactive during simulation.



13. From the Simulink Editor menu, select **Other Displays → Signals & Ports → Port Data Types** to display the port types for the black box.
14. Compile the model (Ctrl-D) to ensure the port data types are up to date.

    Notice that the black box port output type is `UFix_26_0`. This means it is unsigned, 26-bits wide, and has a binary point 0 positions to the left of the least significant bit.
15. Open the configuration M-function `transpose_fir_config.m` and change the output type from `UFix_26_0` to `Fix_26_12`. The modified line (line 26) should read:

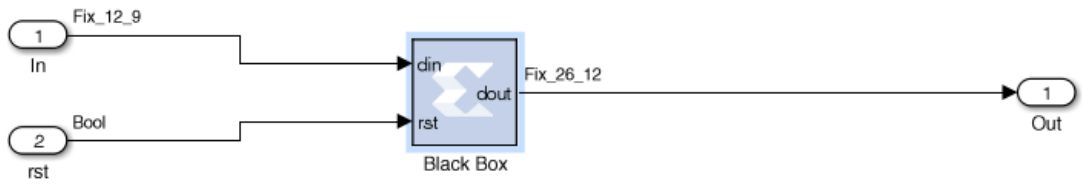
    ```
    dout_port.setType('Fix_26_12');
    ```

    Continue the following steps to edit the configuration M-function to associate an additional HDL file with the black box.
16. Locate line 65:

    ```
    this_block.addFile('transpose_fir.vhd');
    ```
17. Immediately above this line, add the following:

    ```
    this_block.addFile('mac.vhd');
    ```
18. Save the changes to the configuration M-function and close the file.
19. Click the design canvas and recompile the model (Ctrl-D).

    Your Transpose FIR Filter Black Box subsystem should display as follows:

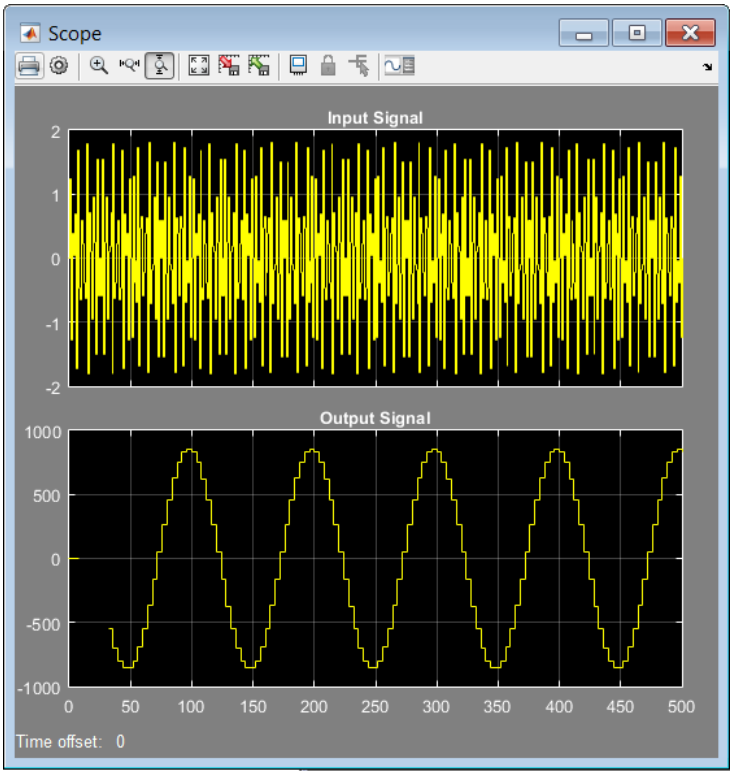


20. From the Black Box block parameter dialog box, change the Simulation mode field from **Inactive** to **Vivado Simulator** and then click **OK**.
21. Move to the top-level of the design and run the simulation.
22. Examine the scope output after the simulation has completed.

    Notice the waveform is no longer zero. When the Simulation Mode was Inactive, the Output Signal scope displayed constant zero. Now, the Output Signal shows a sine wave as the results from the Vivado Simulation.

23. Right-click the Output Signal display and select **Configuration Properties**. In the Main tab, set **Axis Scaling** to the **Auto** setting.

    You should see a display similar to that shown below.



Step 3: Modeling Blocks with C/C++ Code

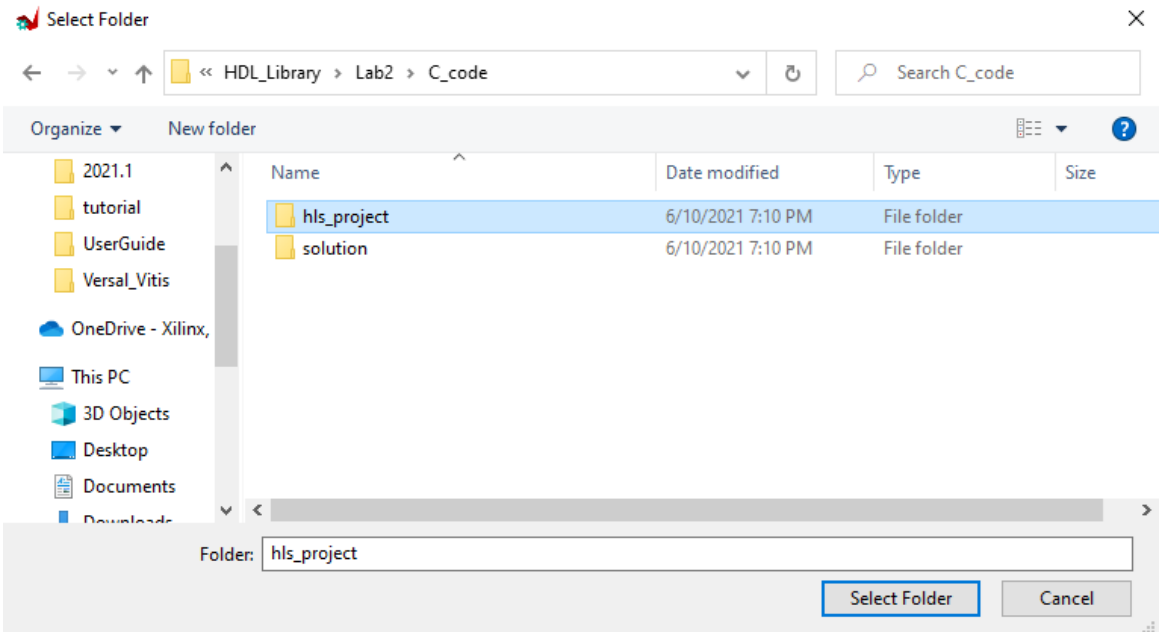
The Vitis HLS tool has the ability to transform C/C++ design sources into RTL. The Model Composer HDL library contains a Vitis HLS block in the HDL/User-Defined Functions library which enables you to bring in C/C++ source files into a Model Composer model.

Procedure

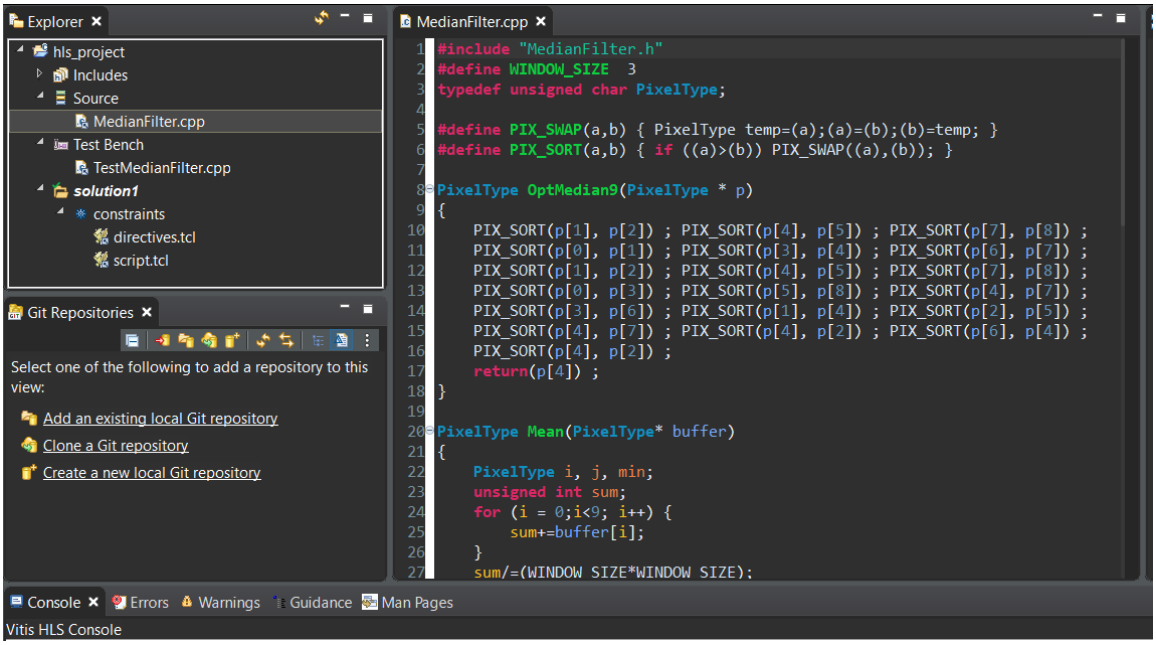
In this step you will first synthesize a C file using Vitis HLS. You will operate within a Vivado DSP design project, using a design file from MATLAB along with an associated HDL wrapper and constraint file. In Part 2, you incorporate the output from Vitis HLS into MATLAB and use the rich simulation features of MATLAB to verify that the C algorithm correctly filters an image.

Part 1: Creating a Vitis HLS Package

1. Invoke Vitis HLS: Click **Windows → Xilinx Design Tools → Vitis HLS 2021.1**.
2. Select **Open Project** in the welcome screen and navigate to the Vitis HLS project directory `\HDL_Library\Lab2\C_code\hls_project` as shown in the following figure.

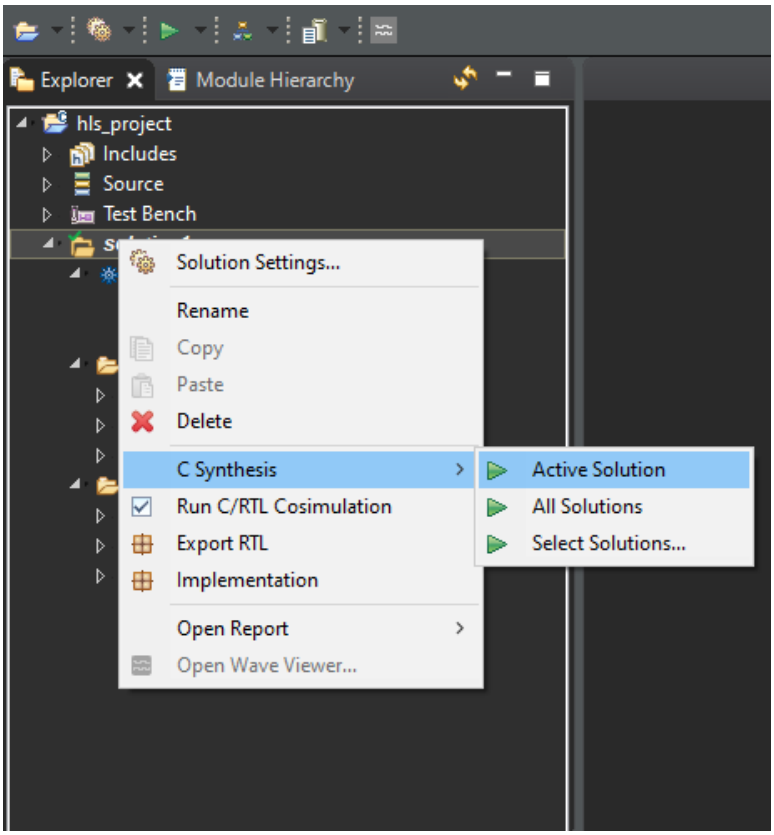


3. Click **Select Folder** to open the project.
4. Expand the Source folder in the Explorer pane (left-hand side) and double-click the file `MedianFilter.cpp` to view the contents of the C++ file as shown in the following figure.



This file implements a 2-Dimensional median filter on 3x3 window size.

5. Synthesize the source file by right-clicking on solution1 and selecting **C Synthesis → Active Solution** as shown in the following figure.

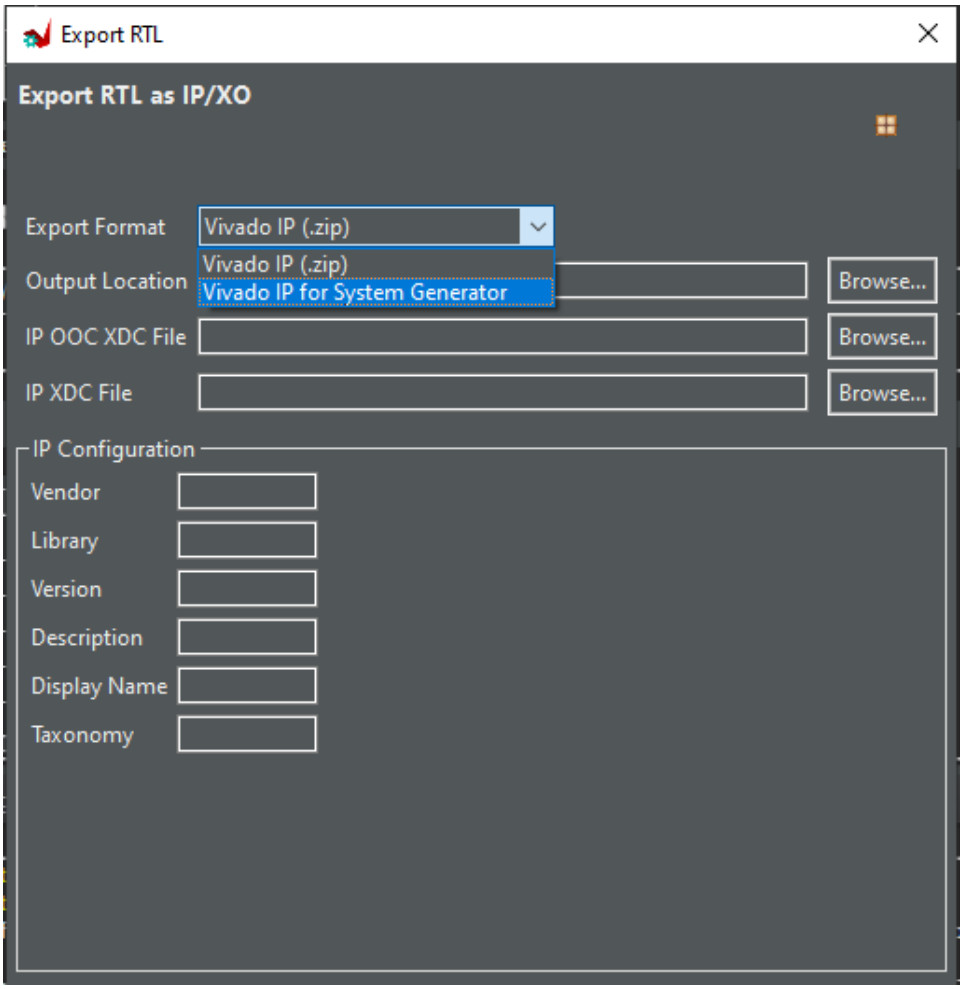


When the synthesis completes, Vitis HLS displays this message:

```
Finished C synthesis
```

Now you will package the source for use in Model Composer.

6. Right-click **solution1** and select **Export RTL**.
7. Set Format Selection to **Vivado IP for System Generator** as shown in the following figure and click **OK**.



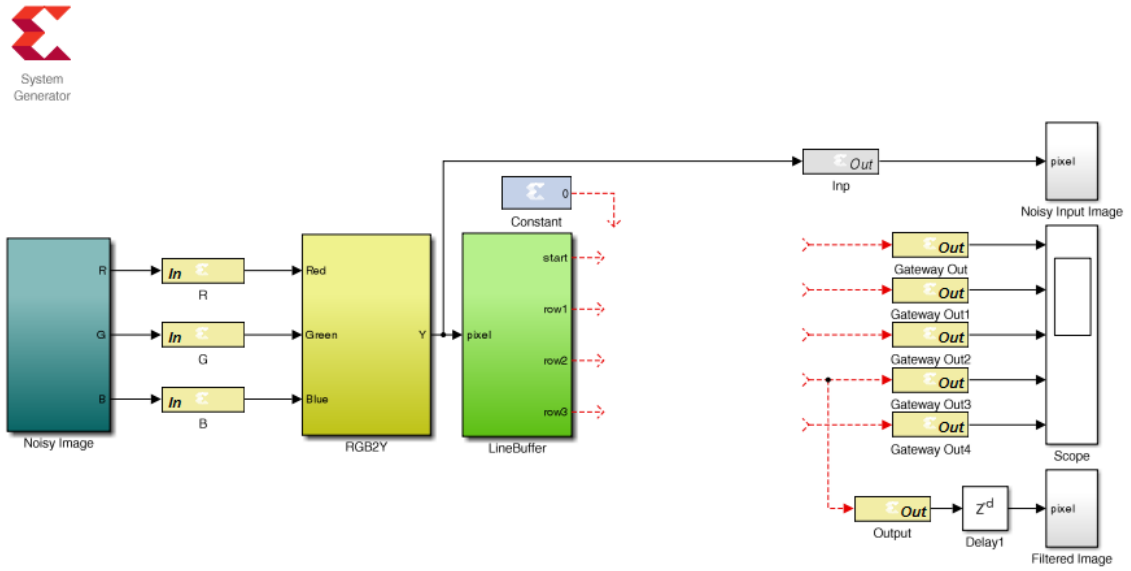
When the Export RTL process completes, Vitis HLS displays this message:

```
Finished export RTL
```

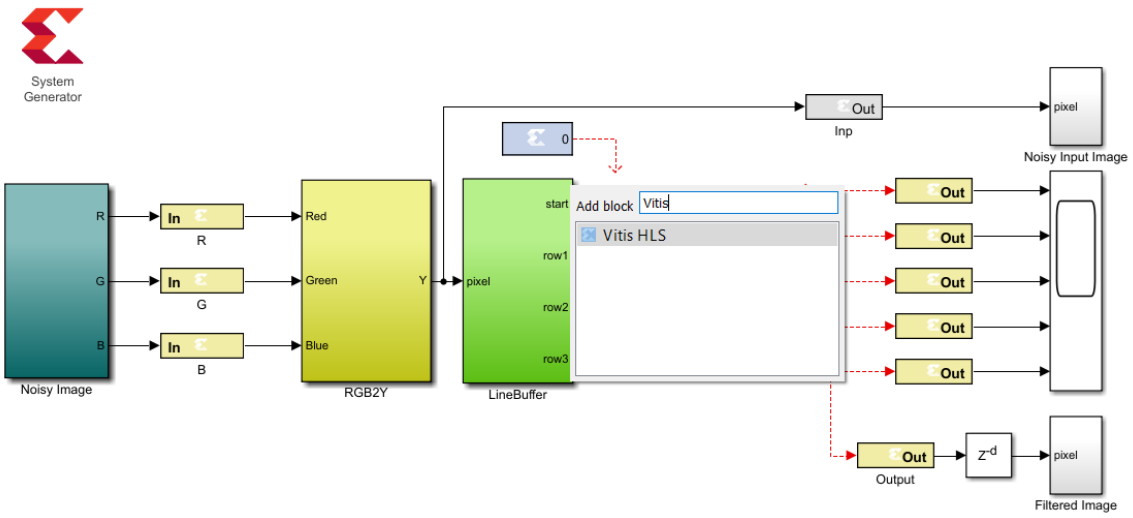
8. Exit Vitis HLS.

Part 2: Including a Vitis HLS Package in a Model Composer Design

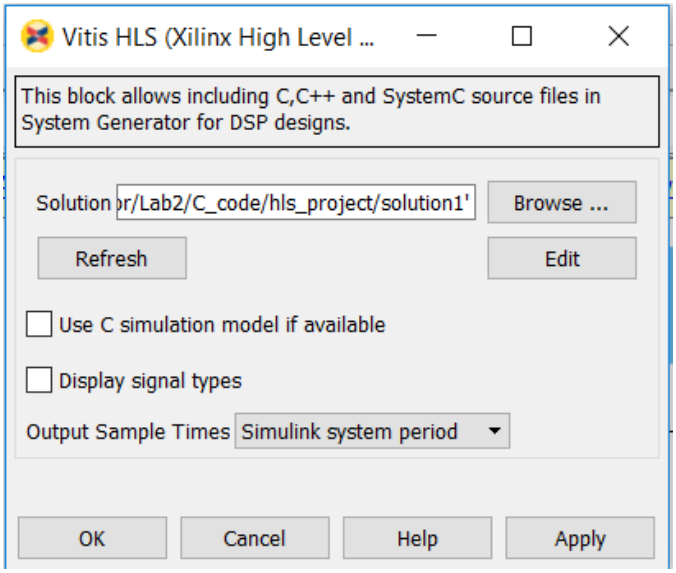
1. Launch Model Composer and open the `Lab2_3.slx` file in the `Lab2/C_code` folder. This should open the model as shown in the following figure.



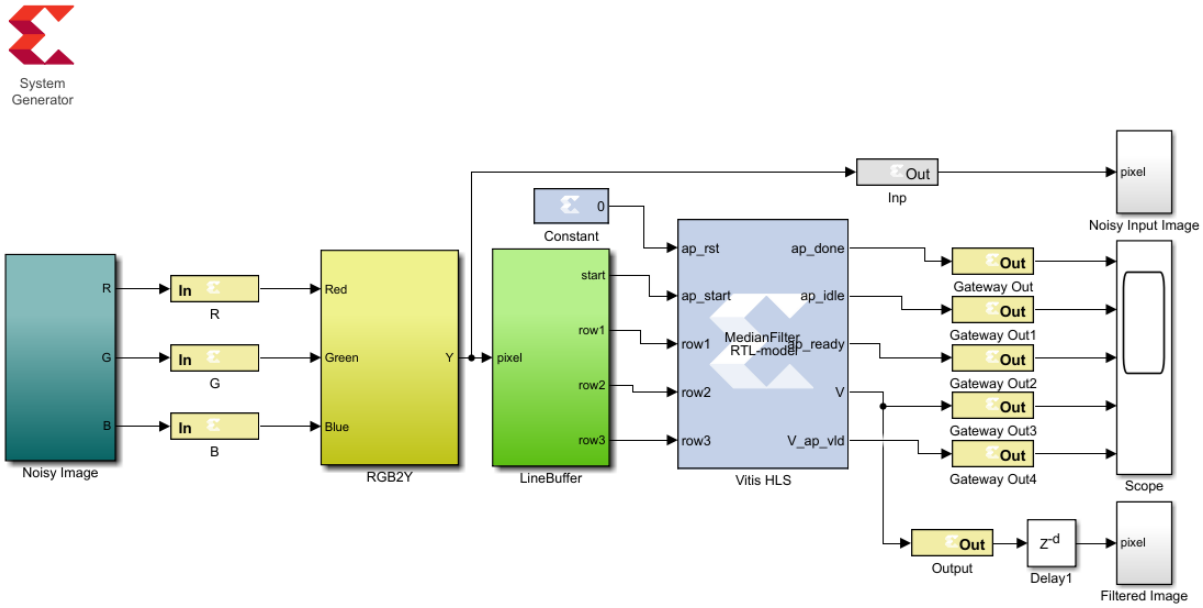
2. Add a Vitis HLS block:
   a. Right-click anywhere on the canvas workspace.
   b. Select **Xilinx BlockAdd**.
   c. Type `Vitis HLS` in the Add block dialog box.
   d. Select **Vitis HLS** as shown in the following figure.



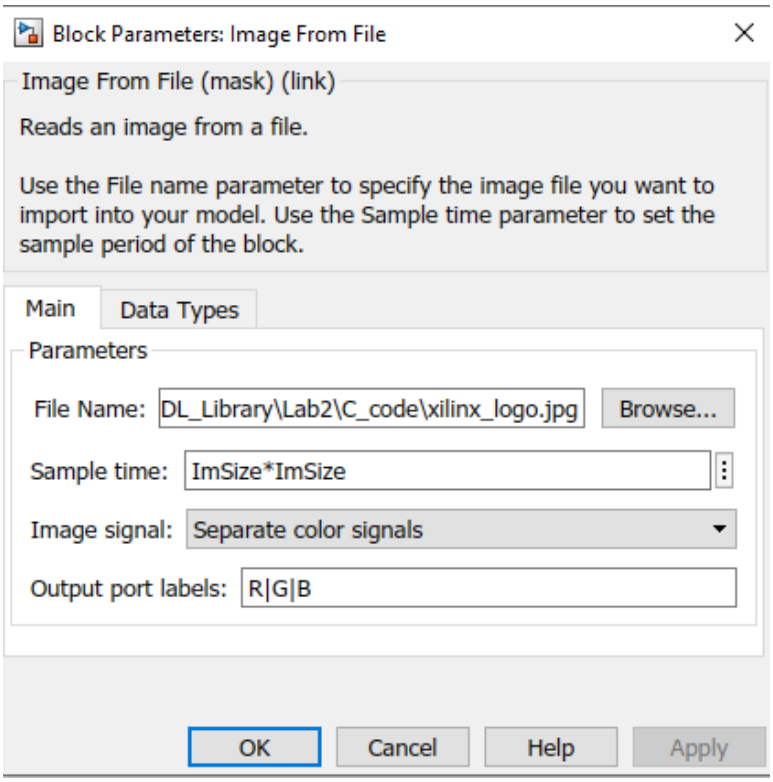
3. Double-click the **Vitis HLS** block to open the Properties Editor.
4. Use the Browse button to select the solution created by Vitis HLS in Step 1, at `\HDL_Library\Lab2\C_code\hls_project\solution1`, as shown in the following figure.
5. Click **OK** to import the Vitis HLS IP.



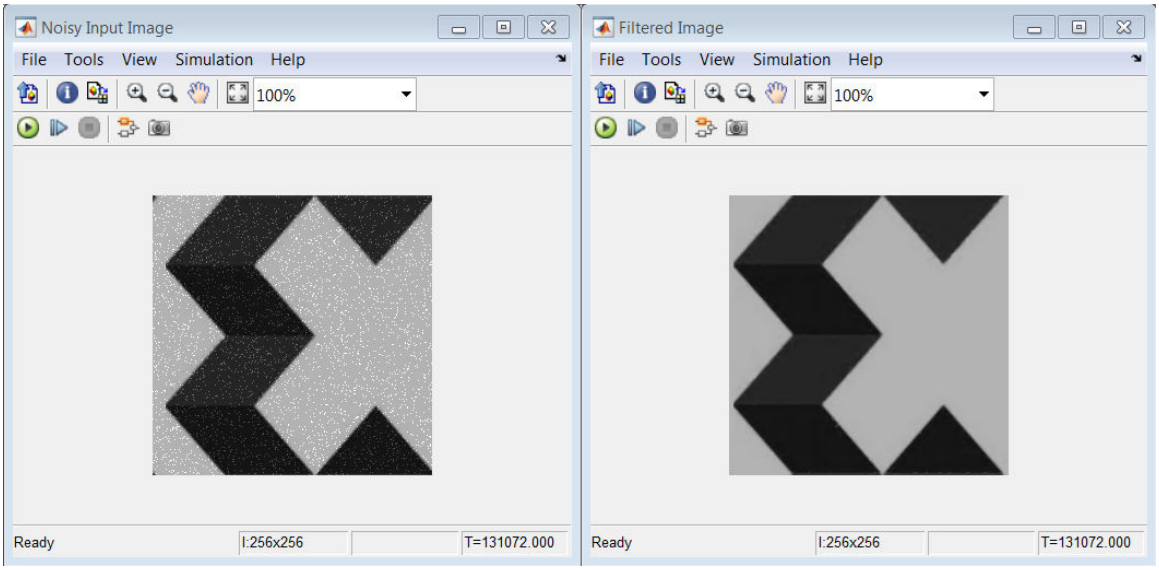
6. Connect the input and output ports of the block as shown in the following figure.



7. Navigate into the Noisy Image sub-system and double-click the **Image From File** block to open the Block Parameters dialog box.
8. Use the **Browse** button to ensure the file name correctly points to the file `xilinx_logo.jpg` as shown in the following figure.



9. Click **OK** to exit the Block Parameters dialog box.
10. Use the Up to Parent toolbar button to return to the top level.
11. Save the design.
12. Simulate the design and verify the image is filtered, as shown in the following figures.



Summary

In this lab you learned:

- How to create control logic using M-Code. The final design can be used to create an HDL netlist, in the same manner as designs created using the HDL Blocksets.
- How to model blocks in Model Composer using HDL by incorporating an existing VHDL RTL design and the importance of matching the data types of the Model Composer model with those of the RTL design and how the RTL design is simulated within Model Composer.
- How to take a filter written in C++, synthesize it with Vitis HLS and incorporate the design into MATLAB. This process allows you to use any C, C++ or SystemC design and create a custom block for use in your designs. This exercise showed you how to import the RTL design generated by Vitis HLS and use the design inside MATLAB.

Solutions to this lab can be found corresponding locations:

- `\HDL_Library\Lab2\C_code\solution`
- `\HDL_Library\Lab2\HDL\solution`
- `\HDL_Library\Lab2\M_code\solution`

Lab 3: Timing and Resource Analysis

In this lab, you learn how to verify the functionality of your designs by simulating in Simulink® to ensure that your Model Composer design is correct when you implement the design in your target Xilinx® device.

Objectives

After completing this lab, you will be able to:

- Identify timing issues in the HDL files generated by Model Composer and discover the source of the timing violations in your design.
- Perform resource analysis and access the existing resource analysis results, along with recommendations to optimize.

Procedure

This lab has two primary parts:

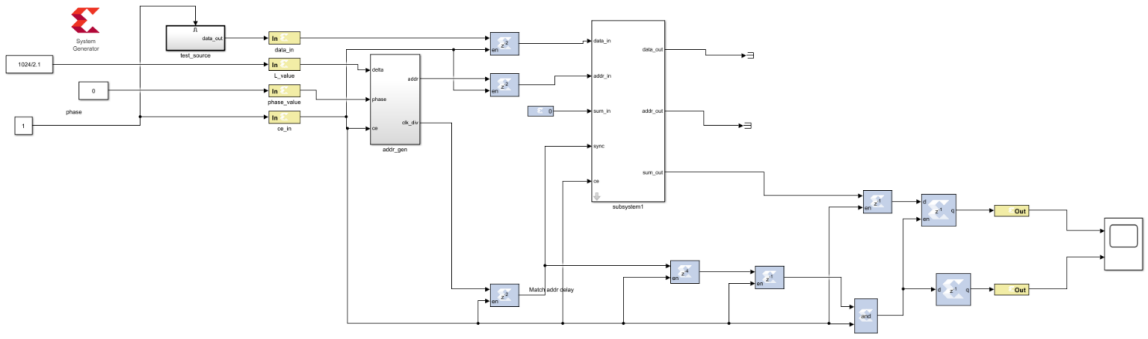
- In Step 1 you will learn how to do timing analysis in Model Composer.
- In Step 2 you will learn how to perform resource analysis in Model Composer.

Step 1: Timing Analysis in Model Composer

1. Invoke Vitis Model Composer.
   - On Windows systems select **Windows → Xilinx Design Tools → Vitis Model Composer 2021.1**.
   - On Linux systems, type `model_cpomposer` at the command prompt.
2. Navigate to the Lab3 folder: `\HDL_Library\Lab3`.

   You can view the directory contents in the MATLAB® Current Folder browser, or type `ls` at the command line prompt.
3. Open the Lab3 design using one of the following:
   - At the MATLAB command prompt, type `open Lab3.slx`
   - Double-click `Lab3.slx` in the Current Folder browser.

   The Lab3 design opens, as shown in the following figure.



4. From your Simulink project worksheet, select **Simulation → Run** or click the **Run simulation** button to simulate the design.

   ***Note*:** In order to see accurate results from Resource Analyzer Window it is recommended to specify a new target directory rather than use the current working directory.
5. Double-click the **System Generator** token to open the Properties Editor.
6. Select the **Clocking** tab.
7. From the Perform analysis menu, select **Post Synthesis** and from Analyzer type menu select **Timing** as shown in the following figure.

8. In the System Generator token dialog box, click **Generate**.

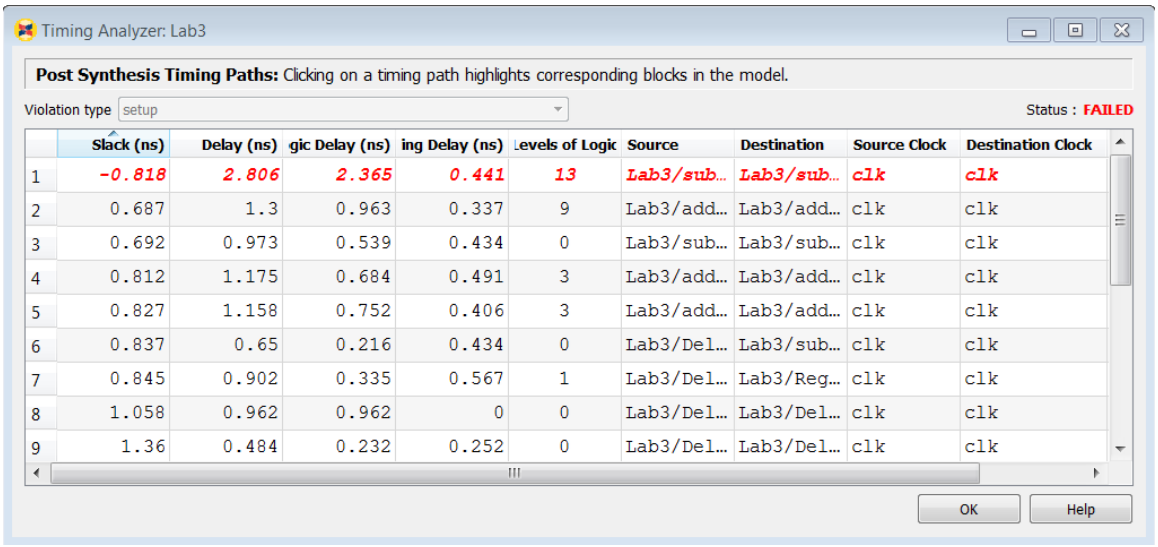
   When you generate, the following occurs:

   a. Model Composer generates the required files for the selected compilation target. For timing analysis Model Composer invokes Vivado in the background for the design project, and passes design timing constraints to Vivado.

   b. Depending on your selection for Perform Analysis (Post Synthesis or Post Implementation), the design runs in Vivado through synthesis or through implementation.

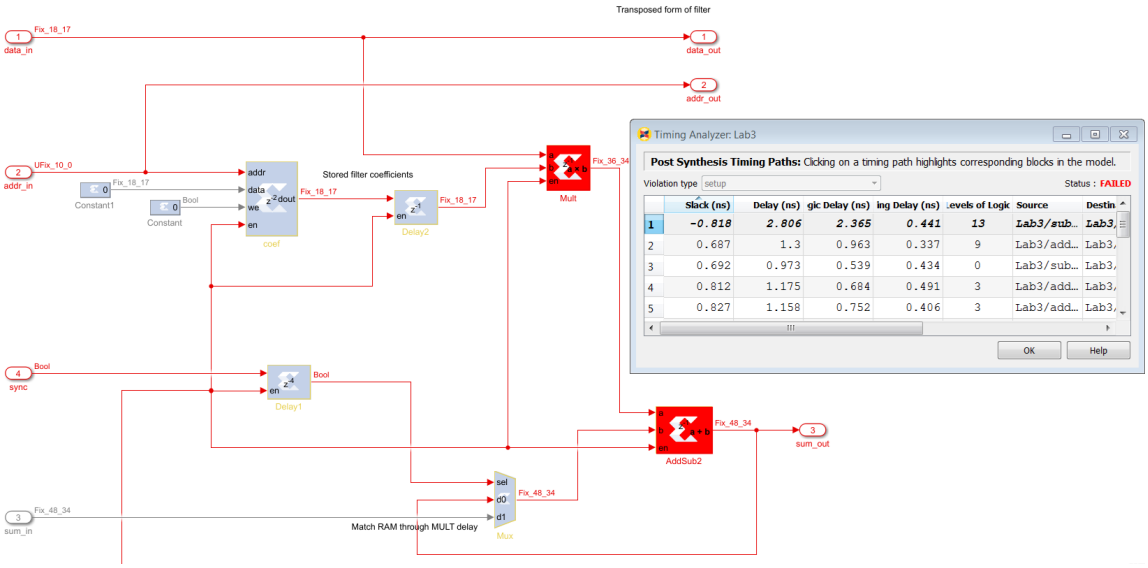
   c. After the Vivado tools run is completed, timing paths information is collected and saved in a specific file format from the Vivado timing database.

   d. Model Composer processes the timing information and displays a Timing Analyzer table with timing paths information as shown in the following figure.

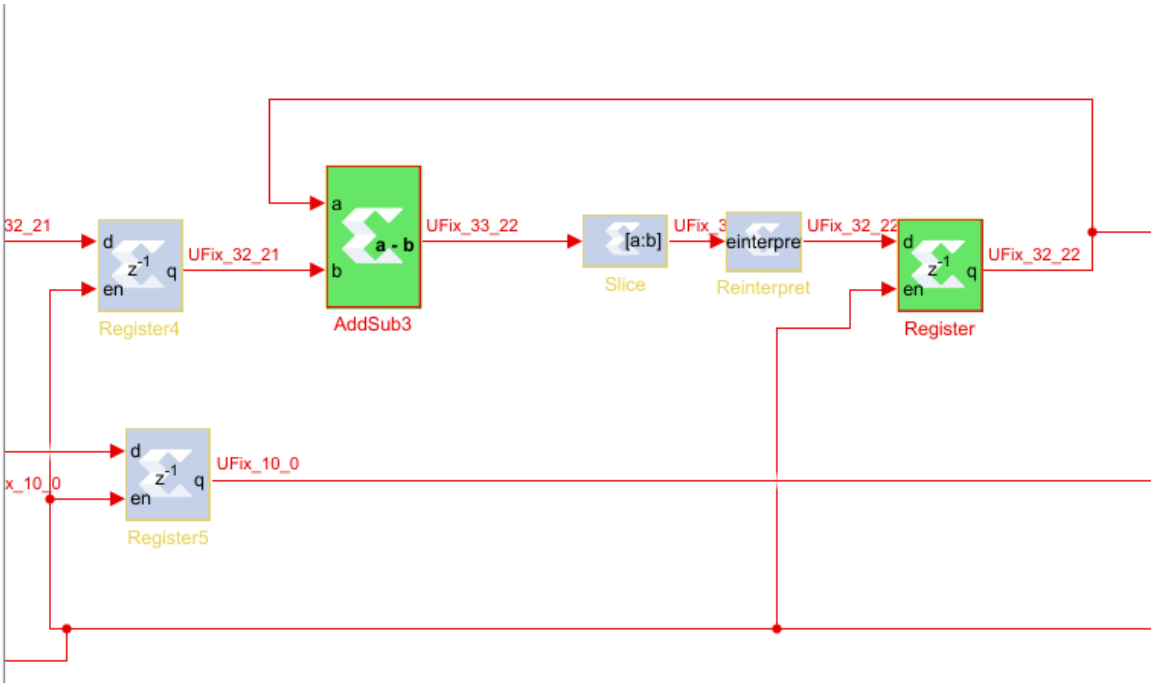


	Slack (ns)	Delay (ns)	gic Delay (ns)	ing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock
1	-0.818	2.806	2.365	0.441	13	Lab3/sub...	Lab3/sub...	clk	clk
2	0.687	1.3	0.963	0.337	9	Lab3/add...	Lab3/add...	clk	clk
3	0.692	0.973	0.539	0.434	0	Lab3/sub...	Lab3/sub...	clk	clk
4	0.812	1.175	0.684	0.491	3	Lab3/add...	Lab3/add...	clk	clk
5	0.827	1.158	0.752	0.406	3	Lab3/add...	Lab3/add...	clk	clk
6	0.837	0.65	0.216	0.434	0	Lab3/Del...	Lab3/sub...	clk	clk
7	0.845	0.902	0.335	0.567	1	Lab3/Del...	Lab3/Reg...	clk	clk
8	1.058	0.962	0.962	0	0	Lab3/Del...	Lab3/Del...	clk	clk
9	1.36	0.484	0.232	0.252	0	Lab3/Del...	Lab3/Del...	clk	clk

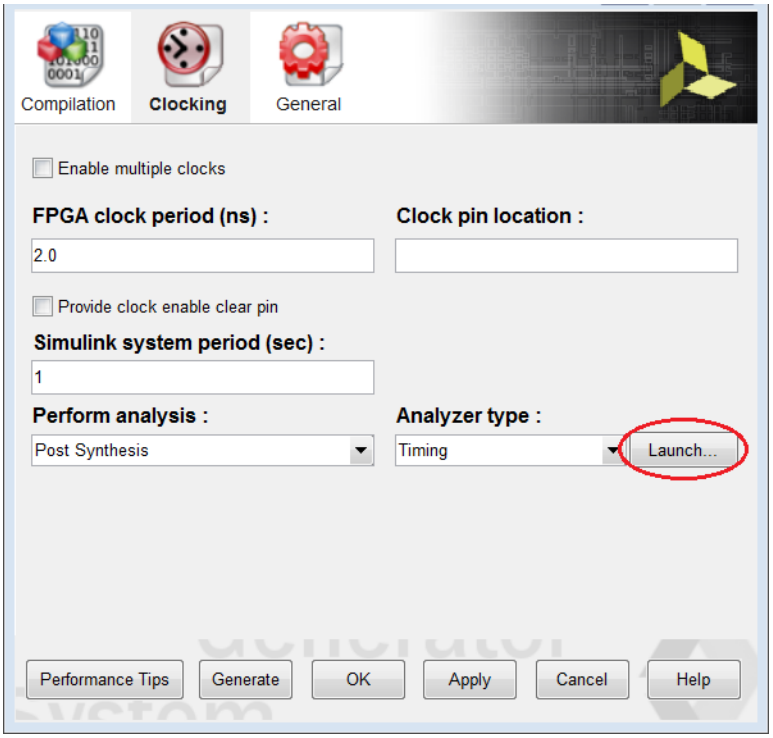
9. In the timing analyzer table:
   - Paths with lowest slack values display, with the worst Slack at the top and increasing slack below
   - Paths with timing violations have a negative slack and display in red.
10. Cross probe from the Timing Analyzer table to the Simulink model by clicking any path in the Timing Analyzer table, which highlights the corresponding Model Composer HDL blocks in the model. This allows you to troubleshoot timing violations by analyzing the path on which they occur.
11. When you cross probe, you see the corresponding path as shown in the following figure.
12. Blocks with timing violations are highlighted in red.



13. Double-click the second path in the Timing Analyzer table and cross-probe, the corresponding highlighted path in green which indicates no timing violation.



If you close the Timing Analyzer sometime later you might want to relaunch the Timing Analyzer table using the existing timing analyzer results for the model. A Launch button is provided under the Clocking tab of the System Generator token dialog box. This will only work if you already ran timing analysis on the Simulink model.

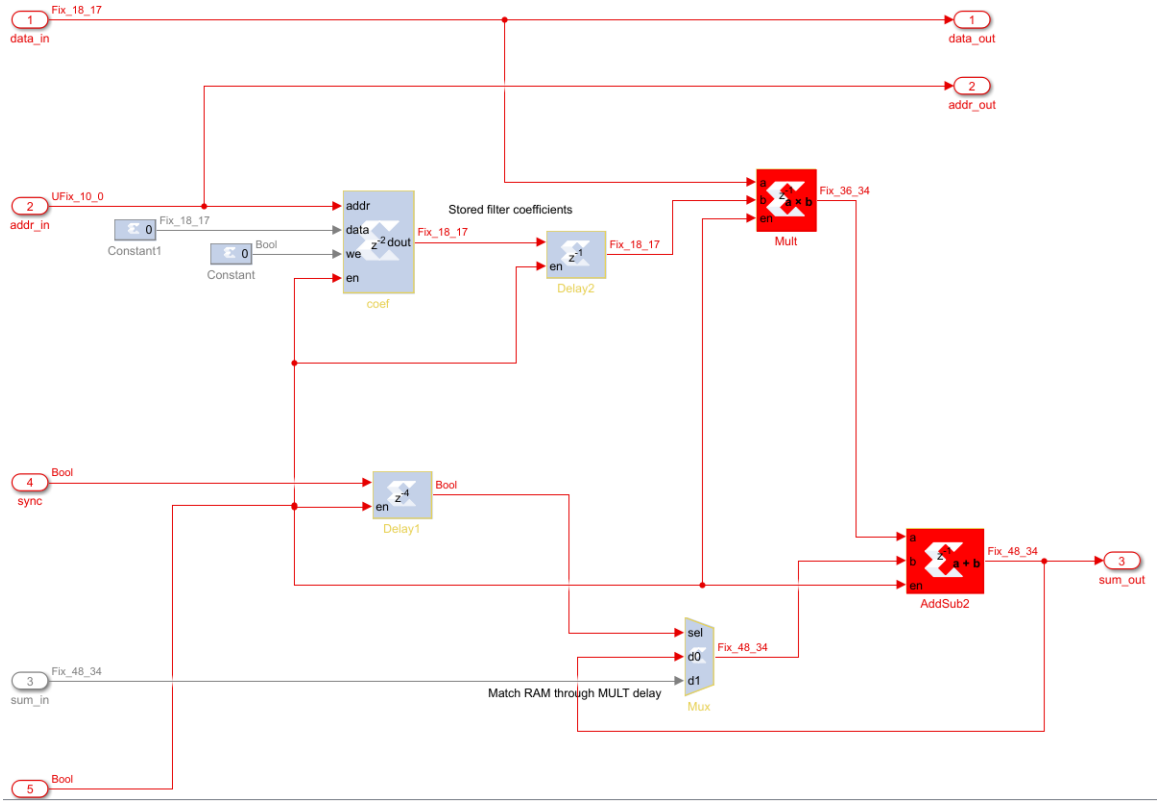


Note: If you relaunch the Timing Analyzer window, make sure that the Analyzer type field is set to Timing. The table that opens will display the results stored Target directory specified in the System Generator token dialog box, regardless of the option selected for Perform analysis (Post Synthesis or Post Implementation).

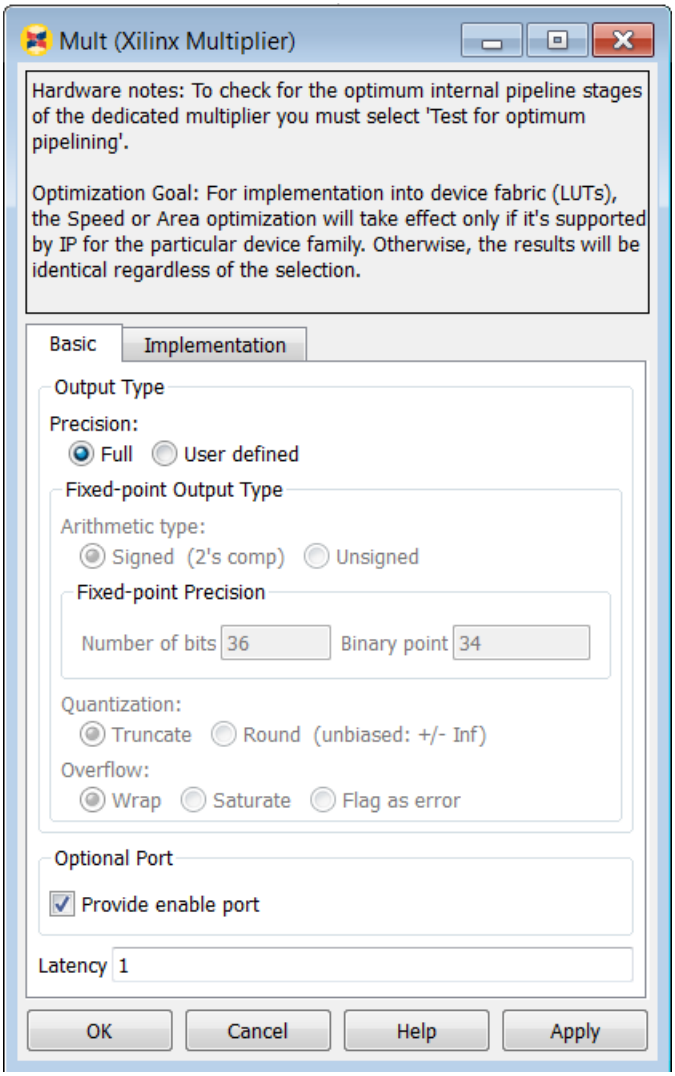
Troubleshooting Timing Violations

Inserting some registers in the combinational path might give better timing results and might help overcome timing violations if any. This can be done by changing latency of the combinational blocks as explained in the following.

1. Double-click the violated path from the Timing Analyzer window which opens the violated path as shown in the following figure.

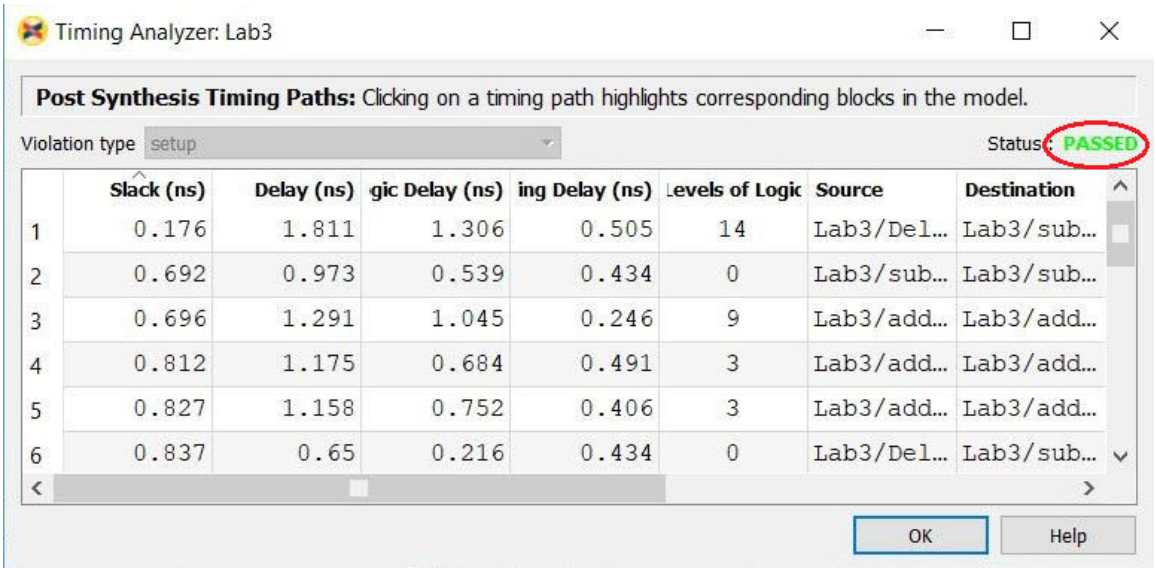


2. Double-click the **Mult** block to open the Multiplier block parameters window as shown in the following figure.



3. Under Basic tab, change the latency from 1 to 2 and click **OK**.
4. Double-click the **System Generator** token, and ensure that the Analyzer Type is Timing and click **Generate**.

5. After the generation completes, it opens the timing Analyzer table as shown in the following figure. Observe the status pass at the top-right corner. It indicates there are no timing violated paths in the design.



***Note*:**

1. For quicker timing analysis iterations, post-synthesis analysis is preferred over post-implementation analysis.
2. Changing the latency of the block might increase the number of resources which can be seen using Step 2: Resource Analysis in Model Composer.

Step 2: Resource Analysis in Model Composer

In this step we use same design, `Lab3.slx`, used for Step 1 but we are going to perform Resource Analysis.

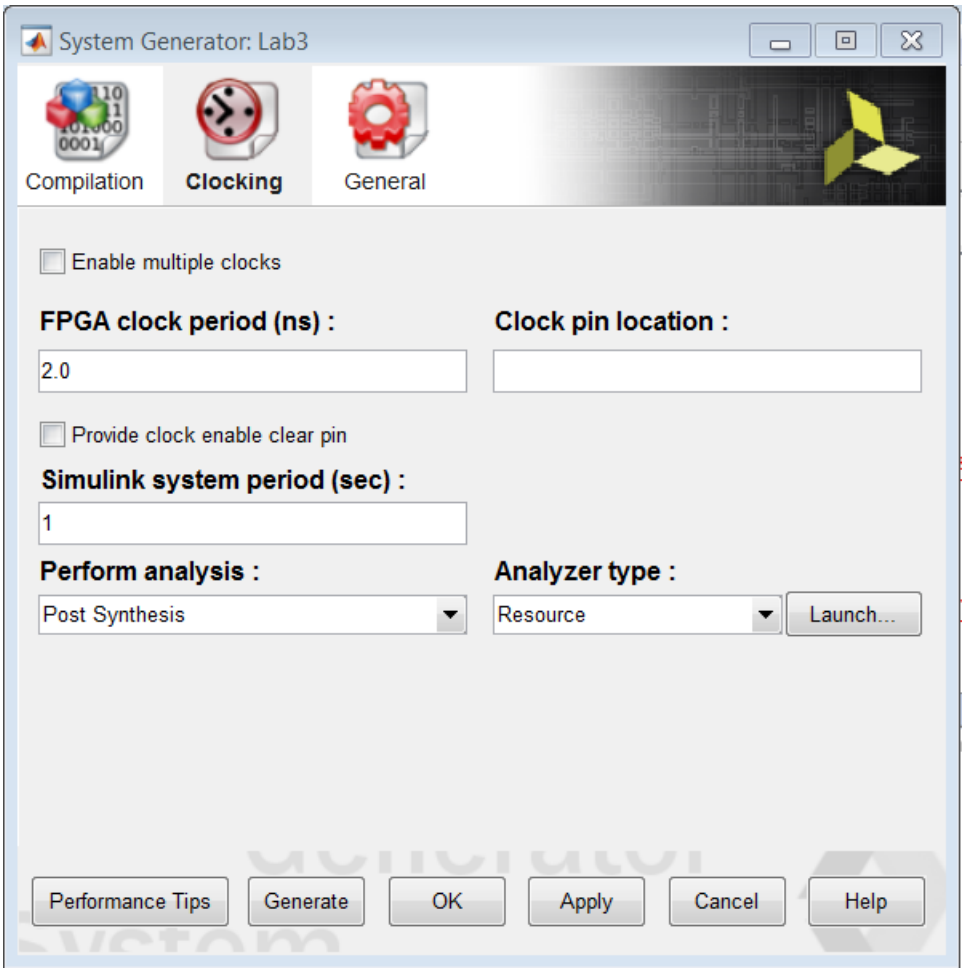
TIP: *Resource Analysis can be performed whenever you generate any of the following compilation targets:*

- IP catalog
- Hardware Co-Simulation
- Synthesized Checkpoint
- HDL Netlist

1. Double-click the **System Generator** token in the Simulink model. Ensure that the part is specified and Compilation is set to any one of the compilation targets listed above.

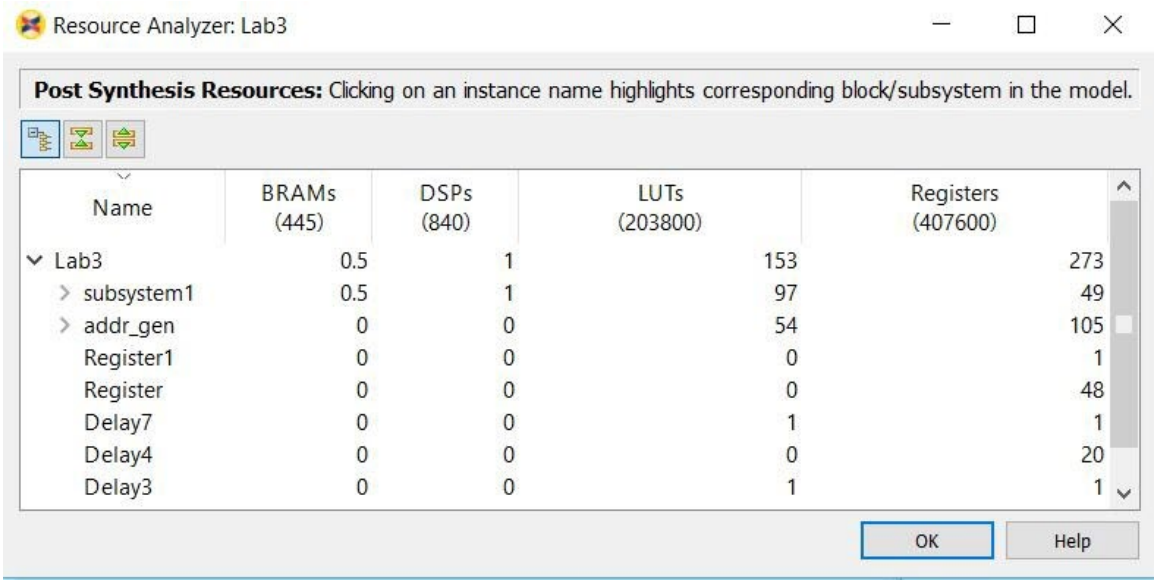
   ***Note*:** In order to see accurate results from Resource Analyzer Window it is recommended to specify a new target directory rather than use the current working directory.

2. In the Clocking tab, set the Perform Analysis field to **Post Synthesis** and Analyzer type field to **Resource**.



3. In the System Generator token dialog box, click **Generate**.

   Model Comoser processes the resource utilization data and displays a Resource Analyzer window with resource utilization information.

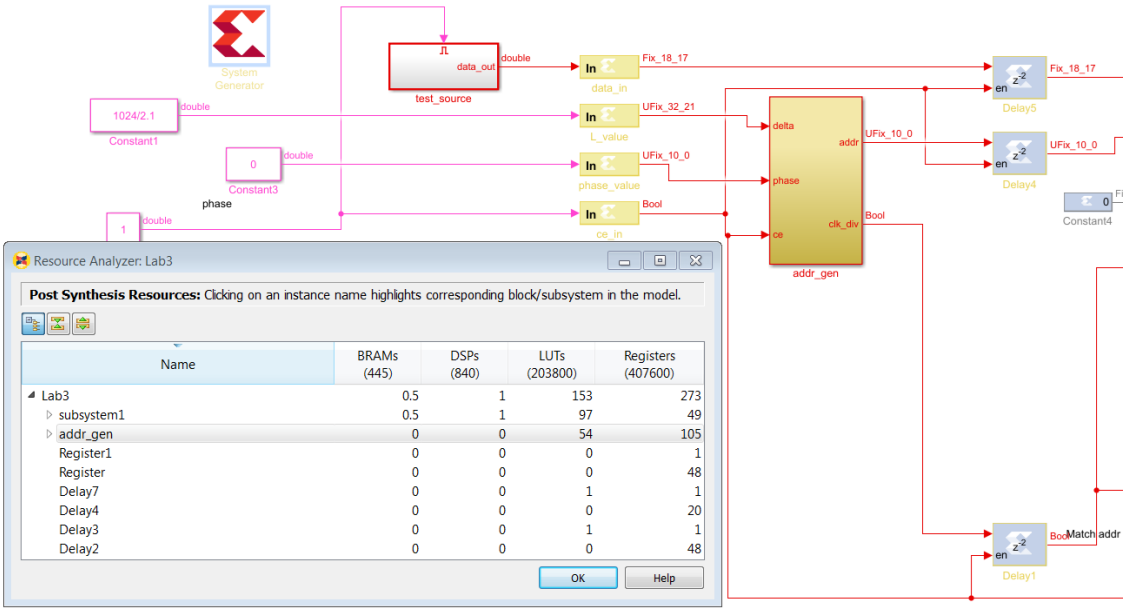


Each column heading (for example, BRAMs, DSPs, or LUTs) in the window shows the total number of each type of resources available in the Xilinx device for which you are targeting your design. The rest of the window displays a hierarchical listing of each subsystem and block in the design, with the count of these resource types.

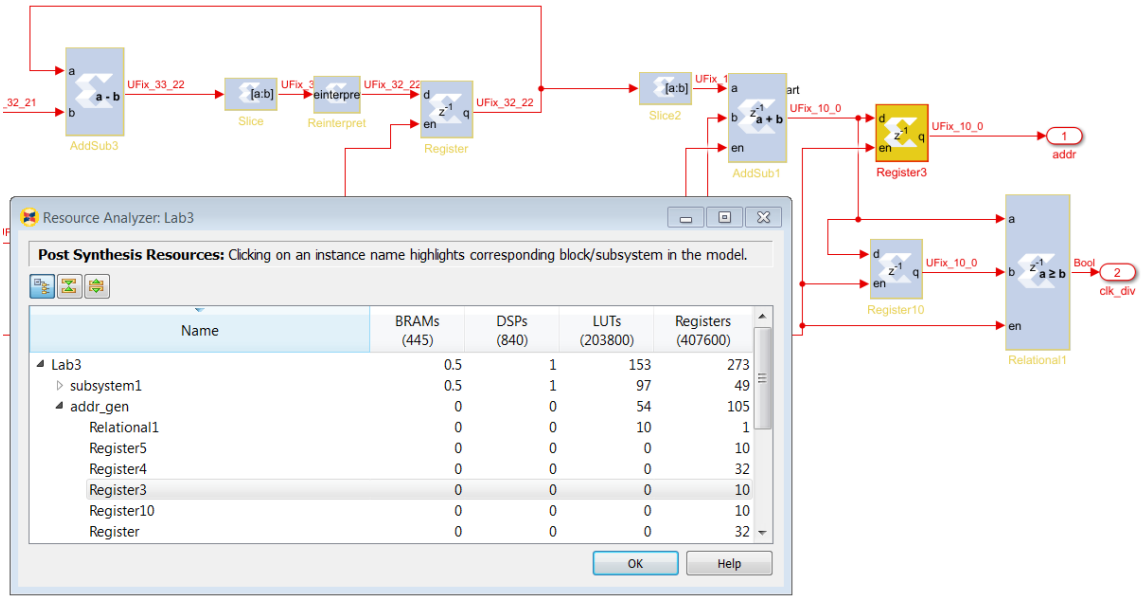
4. You can cross probe from the Resource Analyzer window to the Simulink model by clicking a block or subsystem name in the Resource Analyzer window, which highlights the corresponding Model Composer HDL block or subsystem in the model.

   Cross probing is useful to identify blocks and subsystems that are implemented using a particular type of resource.

5. The block you have selected in the window will be highlighted yellow and outlined in red.



6. If the block or subsystem you have selected in the window is within an upper-level subsystem, then the parent subsystem is highlighted in red in addition to the underlying block as shown in the following figure.



IMPORTANT! *If the Resource Analyzer window or the Timing Analyzer window opens and no information is displayed in the window (table cells are empty), double-click the System Generator token and set the Target directory to a new directory, that is, a directory that has not been used before. Then run the analysis again.*

Summary

In this lab you learned how to use timing and resource analysis inside Model Composer which, in turn, invokes Vivado synthesis to collect the information for the analysis. You also learned how to identify timing violated paths and to troubleshoot them for simple designs.

Lab 4: Working with Multi-Rate Systems

In this lab exercise, you will learn how to efficiently implement designs with multiple data rates using multiple clock domains.

Objectives

After completing this lab, you will be able to:

- Understand the benefits of using multiple clock domains to implement multi-rate designs.

- Understand how to isolate hierarchies using FIFOs to create safe channels for transferring asynchronous data.
- How to implement hierarchies with different clocks.

Procedure

This lab has three primary parts:

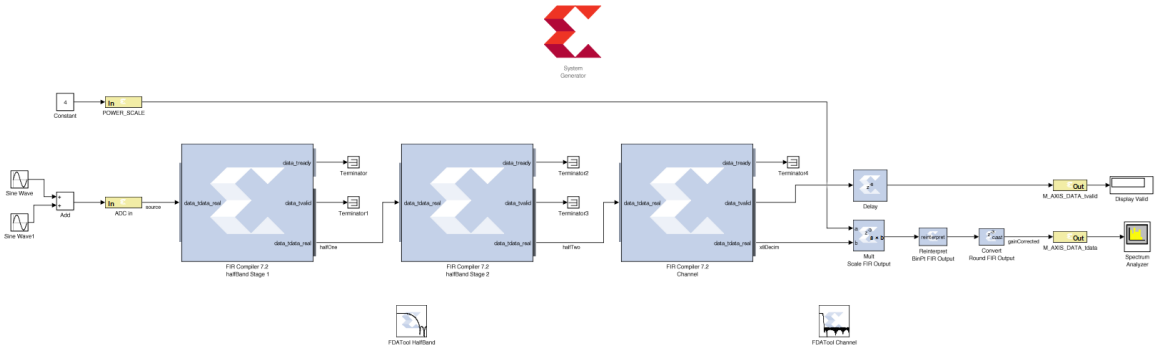
- In Step 1, you will learn how to create hierarchies between the clock domains.
- In Step 2, you will learn how to add FIFOs between the hierarchies.
- In Step 3, you will learn how to add separate clock domains for each hierarchy.

Step 1: Creating Clock Domain Hierarchies

In this step you will review a design in which different parts of the design operate at different data rates and partition the design into subsystems to be implemented in different clock domains.

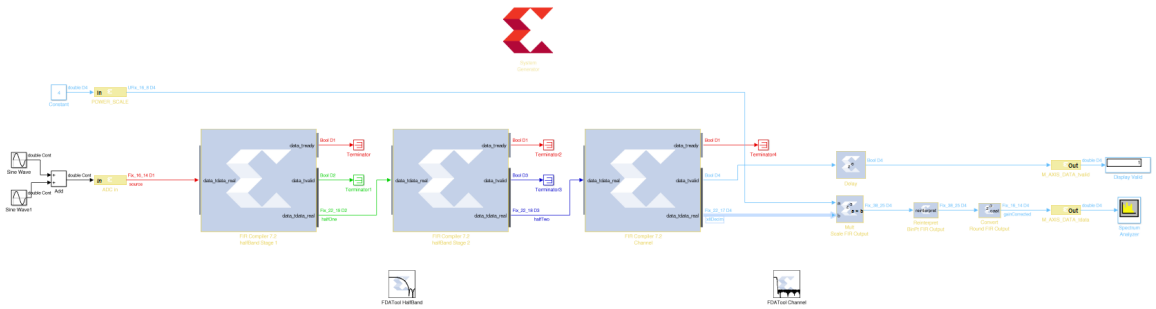
1. Invoke Model Composer:
   - On Windows systems select **Windows → Xilinx Design Tools → Vitis Model Composer 2021.1**.
   - On Linux systems, type `model_composer` at the command prompt.
2. Navigate to the Lab4 folder: `\HDL_Library\Lab4`.
3. At the command prompt, type `open Lab4_1.slx`.

   This opens the Simulink design shown in the following figure. This design is composed of three basic parts:
   - The channel filter digitally converts the incoming signal (491.52 MSPS) to near baseband (61.44 MSPS) using a classic multi-rate filter: the use of two half-band filters followed by a decimation of 2 stage filter, which requires significantly fewer coefficients than a single large filter.
   - The output section gain-controls the output for subsequent blocks which will use the data.
   - The gain is controlled from the POWER_SCALE input.



4. Click the Run simulation button to simulate the design.

   In the following figure Sample Time Display is enabled with colors (right-click in the canvas, **Sample Time Display → Colors**), and shows clearly that the design is running at multiple data rates.



5. The Model Composer environment automatically propagates the different data rates through the design.

   When a multi-rate design such as this is implemented in hardware, the most optimal implementation is to use a clock at the same frequency as the data; however, the clock is abstracted away in this environment. The following methodology demonstrates how to create this ideal implementation in the most efficient manner.

6. To efficiently implement a multi-rate (or multi-clock) design using Model Composer you should capture each part running at the same data rate (or clock frequency) in its own hierarchy with its own System Generator token. The separate hierarchies should then be linked with FIFOs.

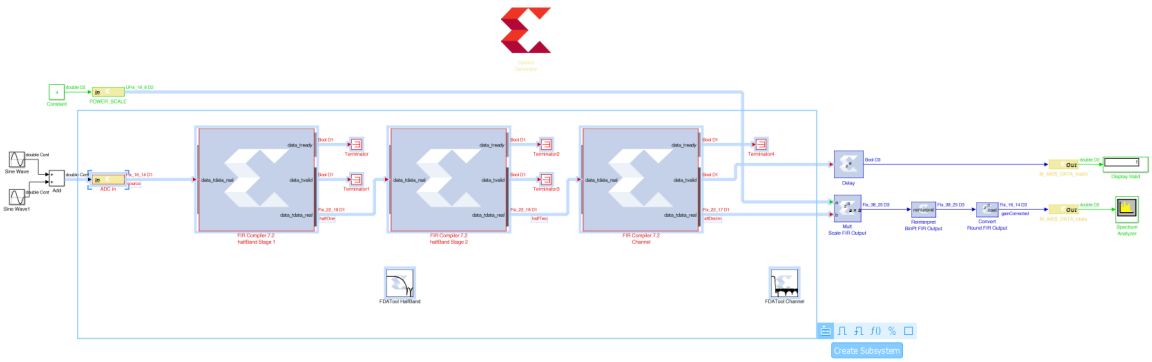
7. The current design has two obvious, and one less obvious, clock domains:

   - The gain control input POWER_SCALE could be configurable from a CPU and therefore can run at the same clock frequency as the CPU.
   - The actual gain-control logic on the output stage should run at the same frequency as the output data from the FIR. This will allow it to more efficiently connect to subsequent blocks in the system.

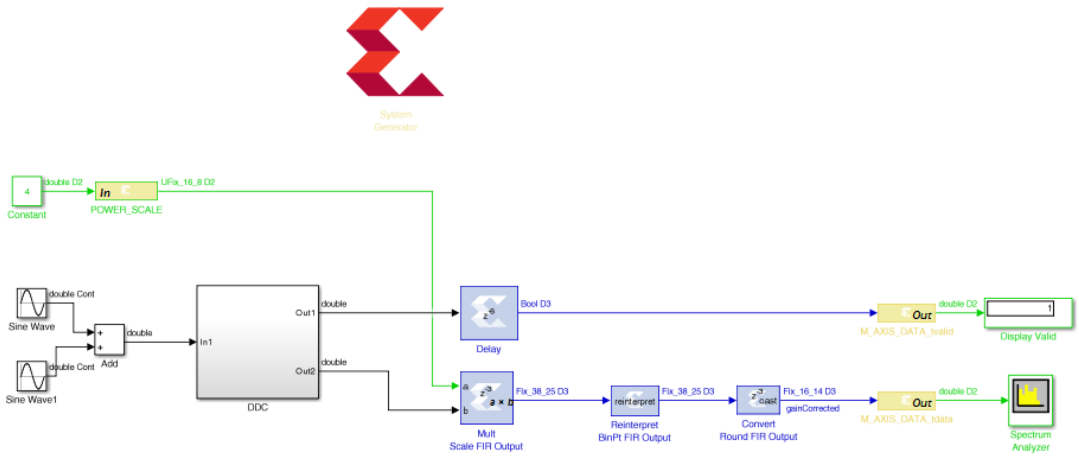
- The less obvious region is the filter-chain. Remember from Lab 1 that complex IP provided with Model Composer, such as the FIR Compiler, automatically takes advantage of over-sampling to provide the most efficient hardware. For example, rather than use 40 multipliers running at 100 MHz, the FIR Compiler will use only eight multipliers if clocked at 500 MHz (= 40*100/500). The entire filter chain can therefore be grouped into a single clock domain. The first FIR Compiler instance will execute at the maximum clock rate and subsequent instances will automatically take advantage of over-sampling.

You will start by grouping these regions into different hierarchies.

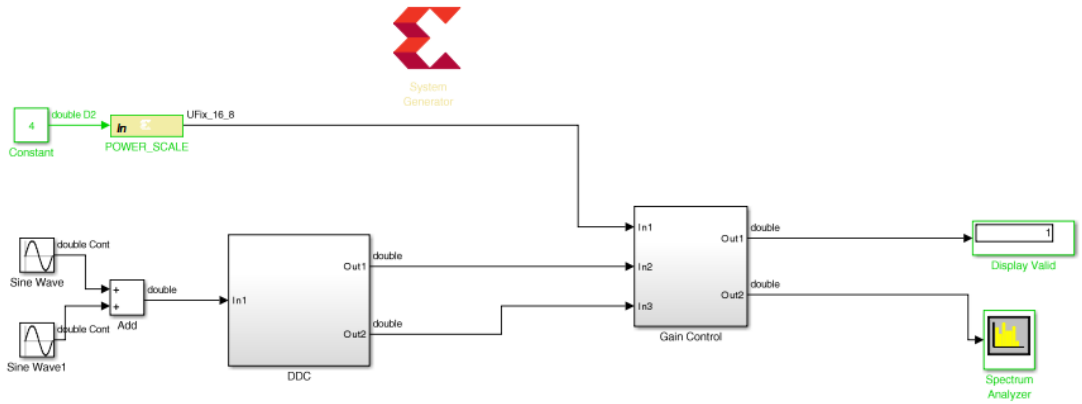
8. Select all the blocks in the filter chain – all those to be in the same clock domain, including the FDATool instances - as shown in the following figure.
9. Select **Create Subsystem**, also as shown in the following figure, to create a new subsystem.



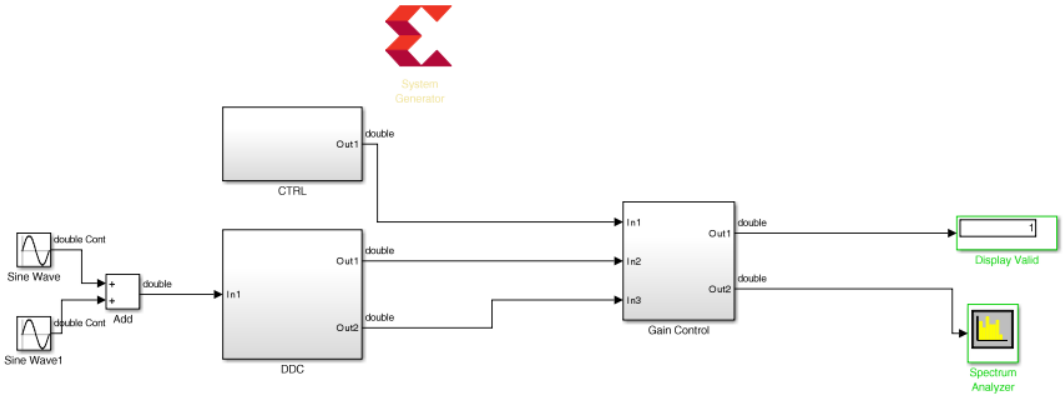
10. Select the instance name subsystem and change this to DDC to obtain the design shown.



11. Select the components in the output path and create a subsystem named Gain Control.



12. Finally, select the Gateway In instance **POWER_SCALE** and **Constant** to create a new subsystem called CTRL. The final grouped design is shown in the following figure.



When this design is complete, the logic within each subsystem will execute at different clock frequencies. The clock domains might not be synchronous with each other. There is presently nothing to prevent incorrect data being sampled between one subsystem and another subsystem.

In the next step you will create asynchronous channels between the different domains to ensure data will asynchronously and safely cross between the different clock domains when the design is implemented in hardware.

Step 2: Creating Asynchronous Channels

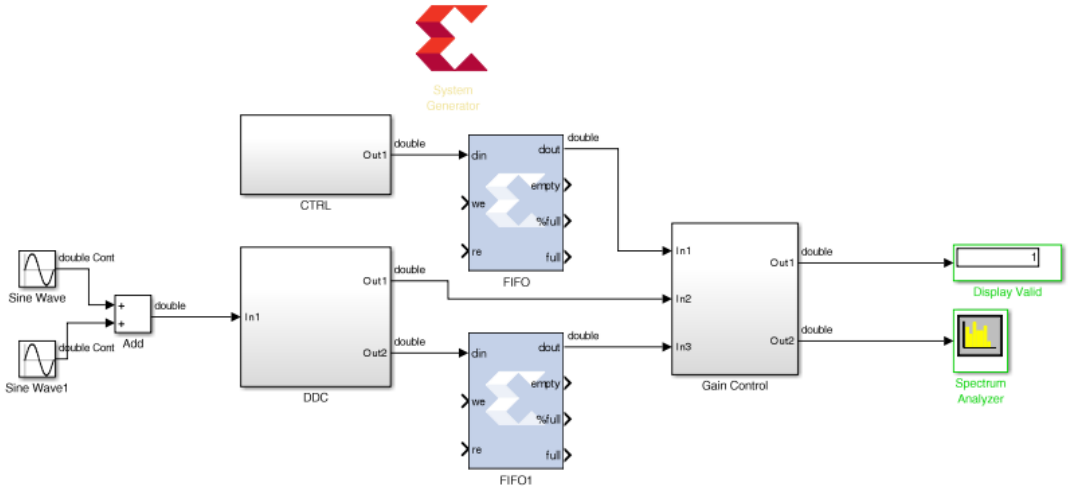
In this step you will implement asynchronous channels between subsystems using FIFOs. The data in FIFOs operates on a First-In-First-Out (FIFO) basis, and control signals ensure data is only read when valid data is present and data is only written when there is space available. If the FIFO is empty or full the control signals will stall the system. In this design the inputs will always be capable of writing and there is no requirement to consider the case for the FIFO being full.

There are two data paths in the design where FIFOs are required:

- Data from CTRL to Gain Control.
- Data from DDC to Gain Control.

1. Right-click anywhere in the canvas and select **Xilinx BlockAdd**.
2. Type `FIFO` in the Add Block dialog box.
3. Select FIFO from the menu to add a FIFO to the design.
4. Connect the data path through instance FIFO. Delete any existing connections to complete this task.
   a. Connect `CTRL/Out1` to `FIFO/din`.
   b. Connect `FIFO/dout` to `Gain Control/In1`.
5. Make a copy of the FIFO instance (using Ctrl-C and Ctrl-V to copy and paste).
6. Connect the data path through instance FIFO1. Delete any existing connections to complete this task.
   a. Connect `DDC/Out2` to `FIFO1/din`.
   b. Connect `FIFO1/dout` to `Gain Control/In3`.

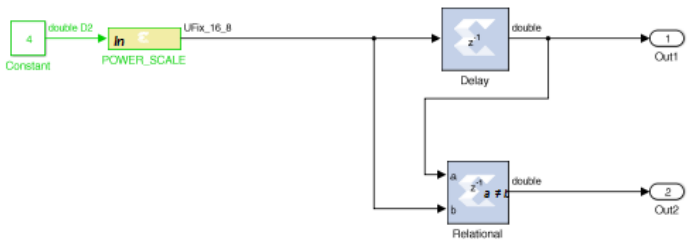
   You have now connected the data between the different domains and have the design shown in the following figure.



You will now connect up the control logic signals to ensure the data is safely passed between domains.

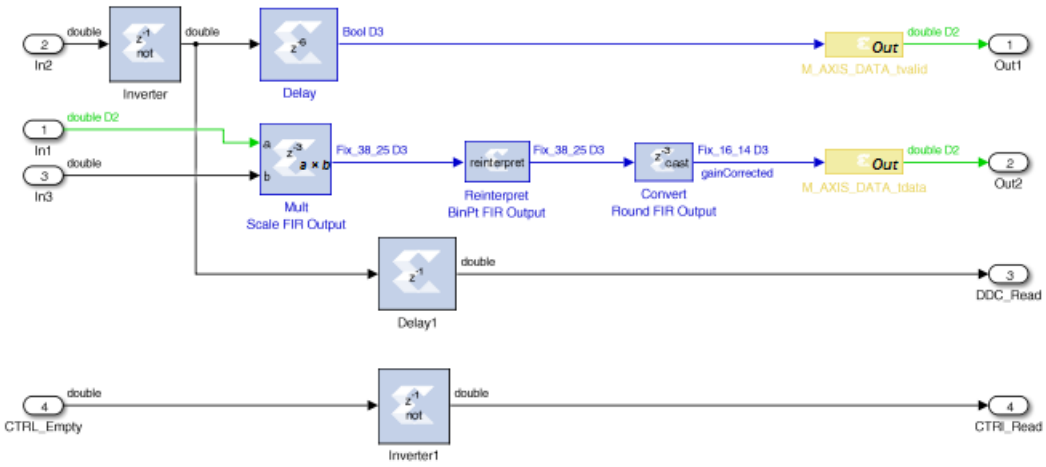
- From the CTRL block a write enable is required. This is not currently present and needs to be created.
- From the DDC block a write enable is required. The `data_tvalid` from the final FIR stage can be used for this.
- The Gain Control must generate a read enable for both FIFOs. You will use the empty signal from the FIFOs and invert it; if there is data available, this block will read it.

7. Double-click the **CTRL** block to open the subsystem.
8. Right-click in the canvas and use **Xilinx BlockAdd** to add these blocks:
   a. Delay (Xilinx)
   b. Relational
9. Select instance Out1 and make a copy (use Ctrl-C and Ctrl-V to cut and paste).
10. Double-click the **Relational** block to open the Properties Editor.
11. Use the Comparison drop-down menu to select **a!=b** and click **OK**.
12. Connect the blocks as shown in the following figure.



This will create an output strobe on Out2 which will be active for one cycle when the input changes, and be used as the write-enable from CTRL to the Gain Control (the FIFO block at the top level).

13. Click the **Up to Parent** toolbar button to return to the top level.
14. Double-click the instance **Gain Control** to open the subsystem.
15. Right-click in the canvas and use **Xilinx BlockAdd** to add these blocks:
    a. Inverter
    b. Inverter (for a total of two inverters)
    c. Delay (Xilinx)
16. Select the instance Out1 and make a copy `Out3` (use Ctrl-C and Ctrl-V to cut and paste).
    - Rename Out3 to `DDC_Read`
17. Select instance Out1 and make a copy `Out3` (use Ctrl-C and Ctrl-V to cut and paste).
    - Rename Out3 to `CTRL_Read`
18. Select instance In1 and make a copy `In4` (use Ctrl-C and Ctrl-V to cut and paste).
    - Rename In4 to `CTRL_Empty`
19. Connect the blocks as shown in the following figure.

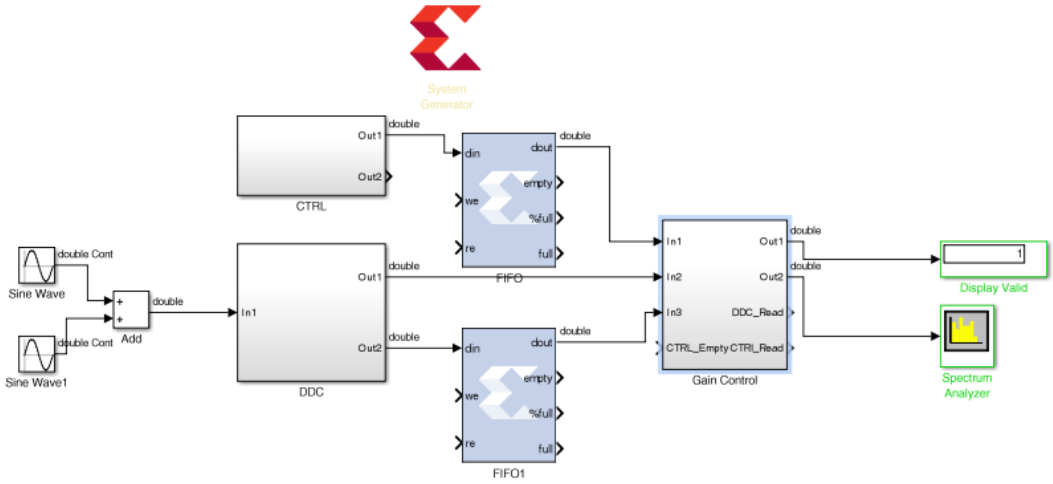


- The FIFO empty signal from the top-level Gain Control FIFO (FIFO) block is simply an inverter block used to create a read-enable for the top-level DDC FIFO (FIFO1). If the FIFO is not empty, the data will be read.
- Similarly, the FIFO empty signal from the top-level DDC FIFO (FIFO1) is inverted to create a FIFO read-enable.

- This same signal will be used as the new `data_tvalid` (which was In2). However, because the FIFO has a latency of 1, this signal must be delayed to ensure this control signal is correctly aligned with the data (which is now delayed by 1 through the FIFO).

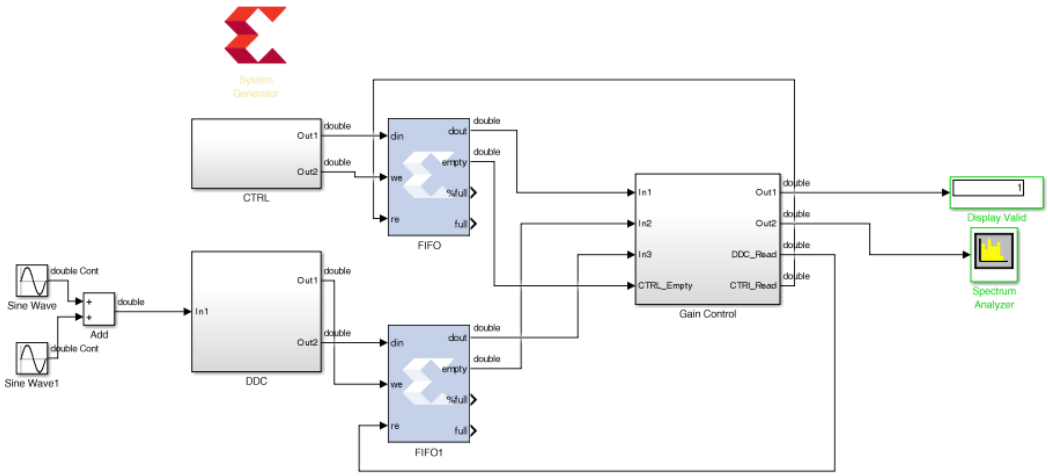
20. Use the **Up to Parent** toolbar button to return to the top level.

    This shows the control signals are now present at the top level.



You will now complete the final connections.

21. Connect the control path through instance FIFO. Delete any existing connections to complete this task.
    a. Connect `CTRL/Out2` to `FIFO/we`.
    b. Connect `FIFO/empty` to `Gain Control/CTRL_Empty`.
    c. Connect `Gain Control/CTRL_Read` to `FIFO/re`.
22. Connect the control path through instance FIFO1. Delete any existing connections to complete this task.
    a. Connect `DDC/Out1` to `FIFO1/we`.
    b. Connect `FIFO1/empty` to `Gain Control/In2`.
    c. Connect `Gain Control/DDC_Read` to `FIFO1/re`.



23. Click the Run simulation button to simulate the design and confirm the correct operation – you will see the same results as Step 1 action 4.

In the next step, you will learn how to specify different clock domains are associated with each hierarchy.

Step 3: Specifying Clock Domains

In this step you will specify a different clock domain for each subsystem.

1. Double-click the System Generator token to open the Properties Editor.
2. Select the **Clocking** tab.
3. Click **Enable multiple clocks**.

   ***Note*:** The FPGA clock period and the Simulink system period are now greyed out. This option informs Model Composer that clock rate will be specified separately for each hierarchy. It is therefore important the top level contains only subsystems and FIFOs; no other logic should be present at the top level in a multi-rate design.

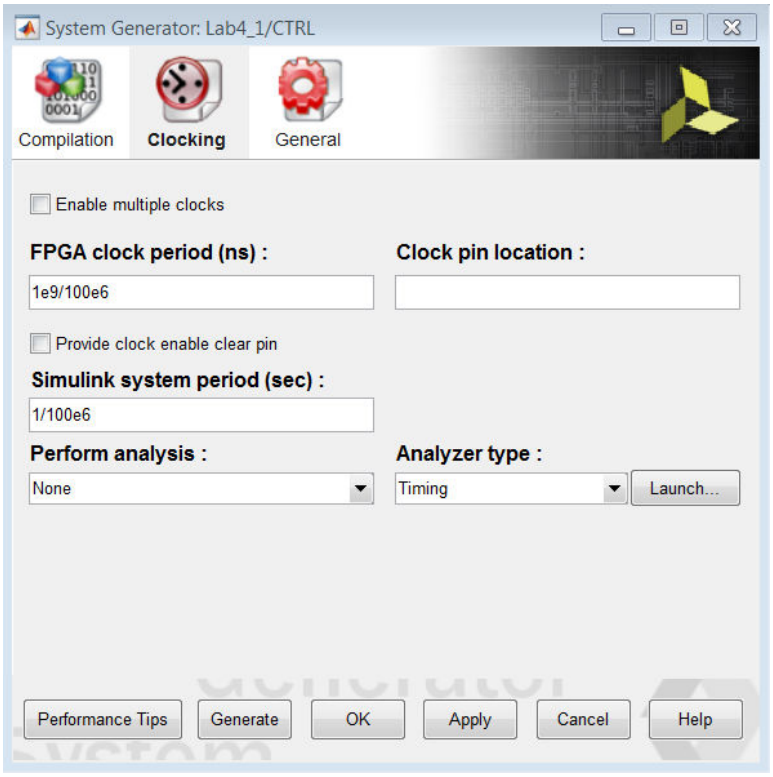
4. Click **OK** to close the Properties Editor.

   You will now specify a new clock rate for the CTRL block. The CTRL block will be driven from a CPU which executes at 100 MHz.

5. Select the **System Generator** token.
6. Press the Ctrl+C key or right-click to copy the token.

   You will specify a new clock rate for the CTRL block. This block will be clocked at 100 MHz and accessed using an AXI4-Lite interface.

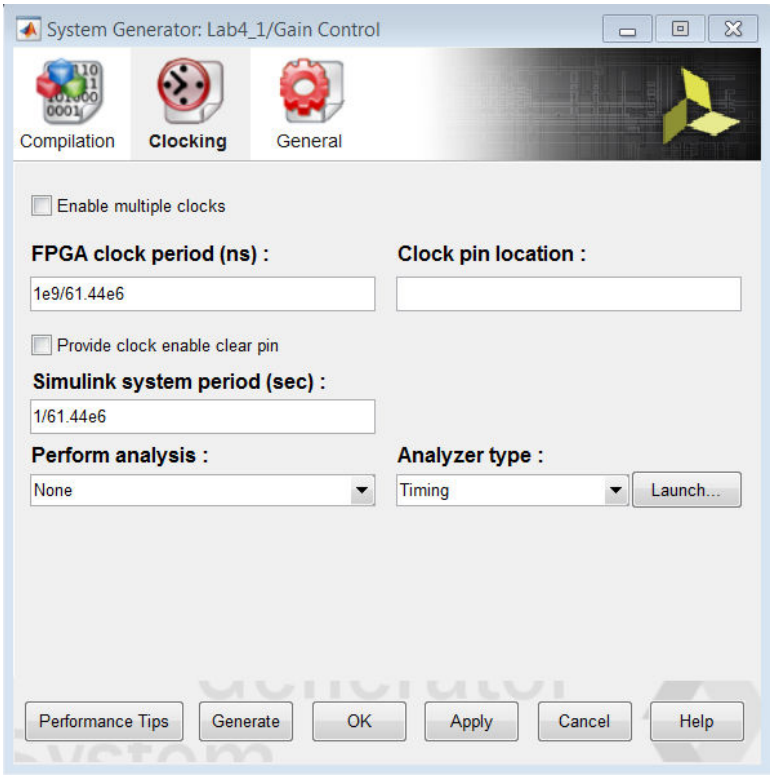
7. Double-click the **CTRL** block to navigate into the subsystem.
8. Press the Ctrl+V key or right-click to paste a System Generator token into CTRL.
9. Double-click the **System Generator** token to open the Properties Editor.
10. Select the **Clocking** tab.
11. Deselect **Enable multiple clocks** (this was inherited when the token was copied).
12. Change the FPGA clock period to 1e9/100e6.
13. Change the Simulink system period to 1/100e6.



14. Click **OK** to close the Properties Editor.
15. Double-click the Gateway In instance **POWER_SCALE** to open the Properties Editor.
16. Change the Sample period to 1/100e6 to match the new frequency of this block.

    In the Implementation tab, note that the Interface is set to AXI4-Lite. This will ensure this port is implemented as a register in an AXI4-Lite interface.
17. Click **OK** to close the Properties Editor.
18. Select and copy the System Generator token.
19. Click the **Up to Parent** toolbar button to return to the top level.

    You will now specify a new clock rate for the Gain Control block. The Gain Control block will be clocked at the same rate as the output from the DDC, 61.44 MHz.
20. Double-click the **Gain Control** block to navigate into the subsystem.
21. Press the Ctrl+V key or right-click to paste a System Generator token into Gain Control.
22. Double-click the **System Generator** token to open the Properties Editor.
23. Select the **Clocking** tab.
24. Change the FPGA clock period to 1e9/61.44e6.
25. Change the Simulink system period to 1/61.44e6.



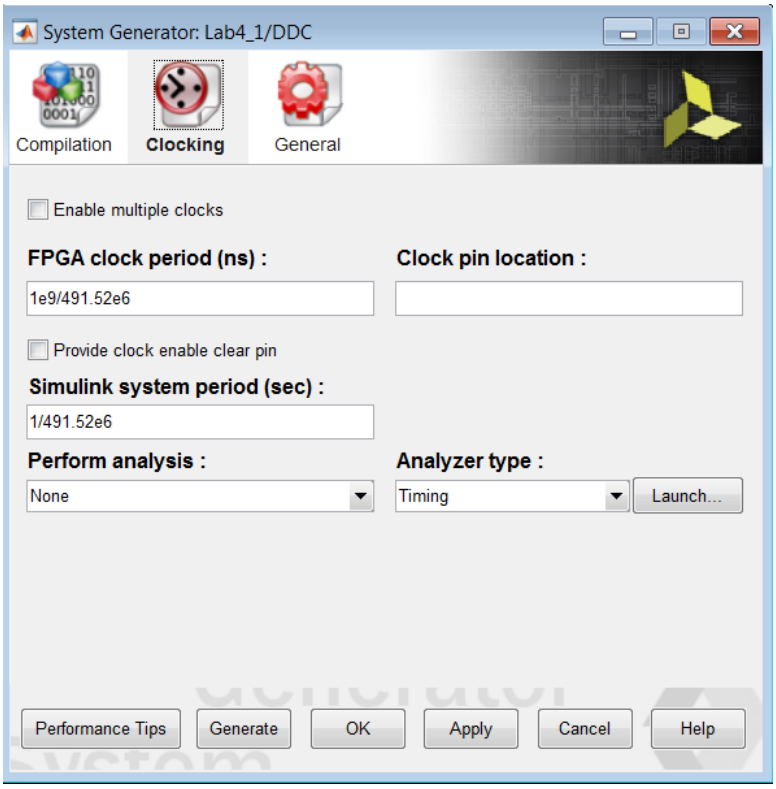
26. Click **OK** to close the Properties Editor.

    Note that the output signals are prefixed with `M_AXI_DATA_`. This will ensure that each port will be implemented as an AXI4 interface, because the suffix for both signals is a valid AXI4 signal name (`tvalid` and `tdata`).

27. Click the **Up to Parent** toolbar button to return to the top level.

    The DDC block uses the same clock frequency as the original design, 491 MHz, because this is the rate of the incoming data.

28. In the top-level design, select and copy the System Generator token.
29. Double-click the **DDC** block to navigate into the subsystem.
30. Press the Ctrl+V key or right-click to paste a System Generator token into the DDC.
31. Double-click the **System Generator** token to open the Properties Editor.
32. Select the **Clocking** tab.
33. Deselect **Enable multiple clocks**. The FPGA clock period and Simulink system period are now set to represent 491 MHz.



34. Click **OK** to close the Properties Editor.
35. Use the **Up to Parent** toolbar button to return to the top level.
36. Save the design.
37. Click the Run simulation button to simulate the design and confirm the same results as earlier.

    The design will now be implemented with three clock domains.
38. Double-click the top-level **System Generator** token to open the Properties Editor.
39. Click **Generate** to compile the design into a hardware description.
40. Click **Yes** to dismiss the simulation warning.
41. When generation completes, click **OK** to dismiss the Compilation status dialog box.
42. Click **OK** to dismiss the System Generator token.
43. Open the file `\HDL_Library\Lab4\IPP_QT_MCD_0001\DDC_HB_hier\ip\hdl\lab4_1.vhd` to confirm the design is using three clocks, as shown in the following.

```
entity lab4_1 is
  port (
     ctrl_clk : in std_logic;
     ddc_clk : in std_logic;
     gain_control_clk : in std_logic;
```

Summary

In this lab, you learned how to create separate hierarchies for portions of the design which are to be implemented with different clock rates. You also learned how to isolate those hierarchies using FIFOs to ensure safe asynchronous transfer of the data and how to specify the clock rates for each hierarchy.

The following `solution` directory contains the final Model Composer (`*.slx`) files for this lab. The `solution` directory does not contain the IP output from Model Composer or the files and directories generated by Vivado.

```
/HDL_Library/Lab4/solution
```

- The results from Step 1 are provided in file `Lab4_1_sol.slx`
- The results from Step 2 are provided in file `Lab4_2_sol.slx`
- The final results from Step 3 are provided in file `Lab4_3_sol.slx`

Lab 5: Using AXI Interfaces and IP Integrator

In this lab, you will learn how AXI interfaces are implemented using Model Composer. You will save the design in IP catalog format and use the resulting IP in the Vivado® IP integrator environment. Then you will see how IP integrator enhances your productively by supplying connection assistance when you use AXI interfaces.

Objectives

After completing this lab, you will be able to:

- Implement AXI interfaces in your designs.
- Add your design as IP in the Vivado IP catalog.
- Connect your design in IP integrator.

Procedure

This lab has four primary parts:

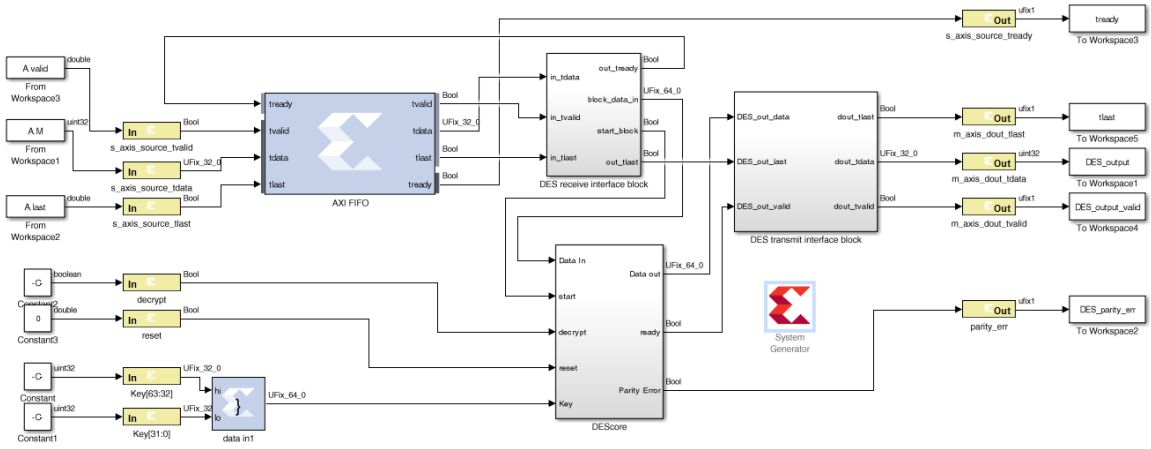
- In Step 1, you will review how AXI interfaces are implemented using Model Composer.
- In Step 2, you will create a Vivado project for your Model Composer IP.
- In Step 3, you will create a design in IP integrator using the Model Composer IP.
- In Step 4, you will implement the design and generate an FPGA bitstream (the file used to program the FPGA).

Step 1: Review the AXI Interfaces

In this step you review how AXI interfaces are defined and created.

1. Invoke Vitis Model Composer and use the Current Folder browser to change the directory to `\HDL_Library\Lab5`.
2. Type `open Lab5_1.slx` in the Command Window.

   This opens the design shown in the following figure.



   This design uses a number of AXI interfaces. You will review these shortly.

   - Using AXI interfaces allows a design exported to the Vivado IP catalog to be efficiently integrated into a larger system using IP integrator.
   - It is not a requirement for designs exported to the IP catalog to use AXI interfaces.

   This design uses the following AXI interfaces:

   - An AXI4-Stream interface is used for ports `s_axis_source_*`. All Gateway In and Out signals are prefixed with the same name (`s_axis_source_`), ensuring they are grouped into the same interface. The suffixes for all ports are valid AXI4-Stream interface signal names (`tready`, `tvalid`, `tlast` and `tdata`).
   - An AXI4-Stream interface is used for ports `m_axis_dout_*`.
   - An AXI4-Lite interface is used for the remaining ports. You can confirm this using the following steps:

3. Double-click Gateway In instance **decrypt** (or any of **reset**, **Keys[63:32]**, **Keys[31:0]**, or **parity_err**).
4. In the Properties Editor select the **Implementation** tab.
5. Confirm the Interface is specified as AXI4-Lite in the Interface options.
6. Click **OK** to exit the Properties Editor.

Details on simulating the design are provided in the canvas notes. For this exercise, you will concentrate on exporting the design to the Vivado IP catalog and use the IP in an existing design.

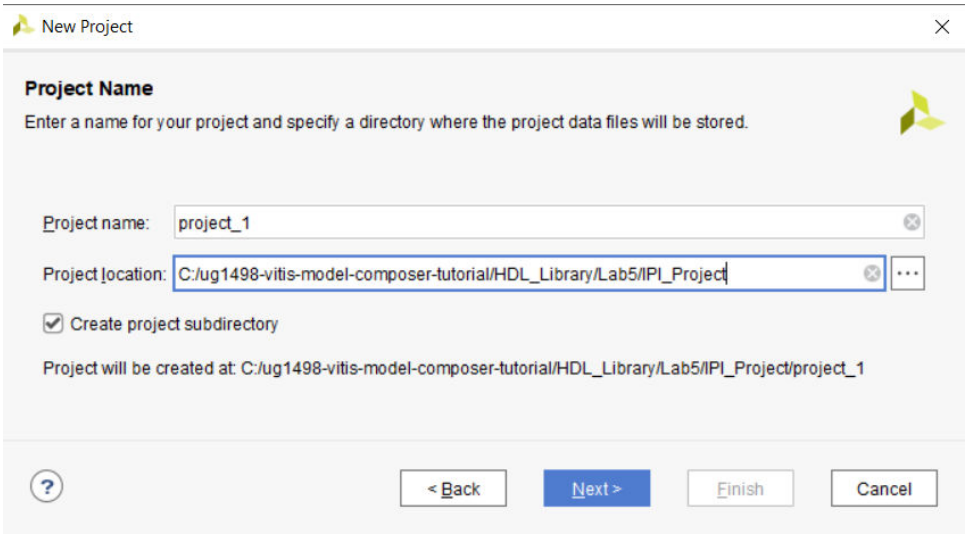
Step 2: Create a Vivado Project using Model Composer HDL IP

In this step you create a Vivado project which you will use to create your hardware design.

1. Double-click the **System Generator** token to open the Properties Editor.
2. In the Properties Editor, make sure IP catalog is selected for the Compilation type.
3. Click **Generate** to generate a design in IP catalog format.
4. Click **OK** to dismiss the Compilation status dialog box.
5. Click **OK** to dismiss the System Generator token.

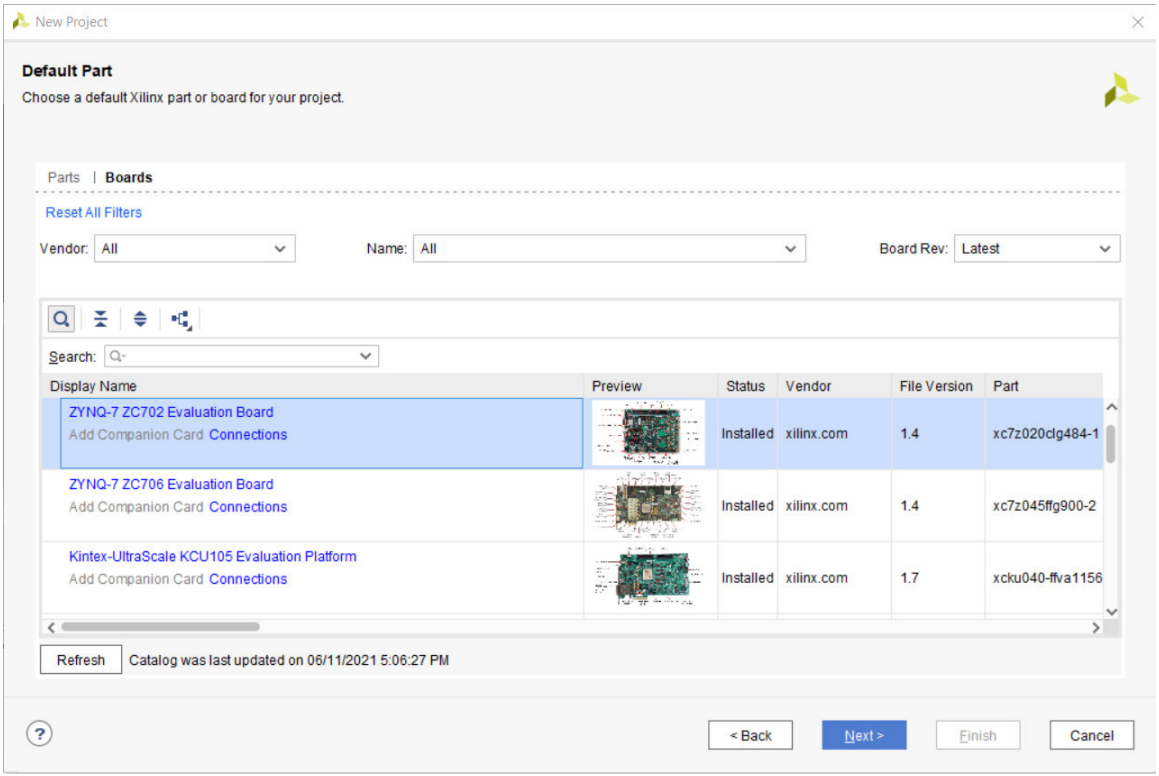
   The design has been written in IP catalog format to the directory `./IPI_Project`. You will now import this IP into the Vivado IP catalog and use the IP in an existing example project.
6. Open the Vivado IDE using **Windows → Xilinx Design Tools → Vivado 2021.1**.
7. Click **Create Project**.
8. Click **Next**.
9. Enter `\HDL_Library\Lab5\IPI_Project` for the Project Location.

TIP: *You will have to manually type `/IPI_Project` in the Project location box to create the IPI_Project directory.*



10. Click **Next**.
11. Select both **RTL Project** and **Do not specify sources** at this time and click **Next**.

12. Select **Boards** and **ZYNQ-7 ZC702 Evaluation Board** as shown in the next figure.



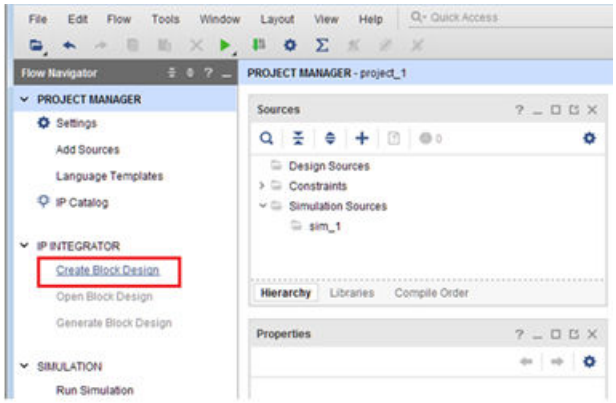
13. Click **Next**.
14. Click **Finish**.

You have now created a Vivado project based on the ZC702 evaluation board.

Step 3: Create a Design in IP Integrator

In this step you will create a design using the Model Composer IP.

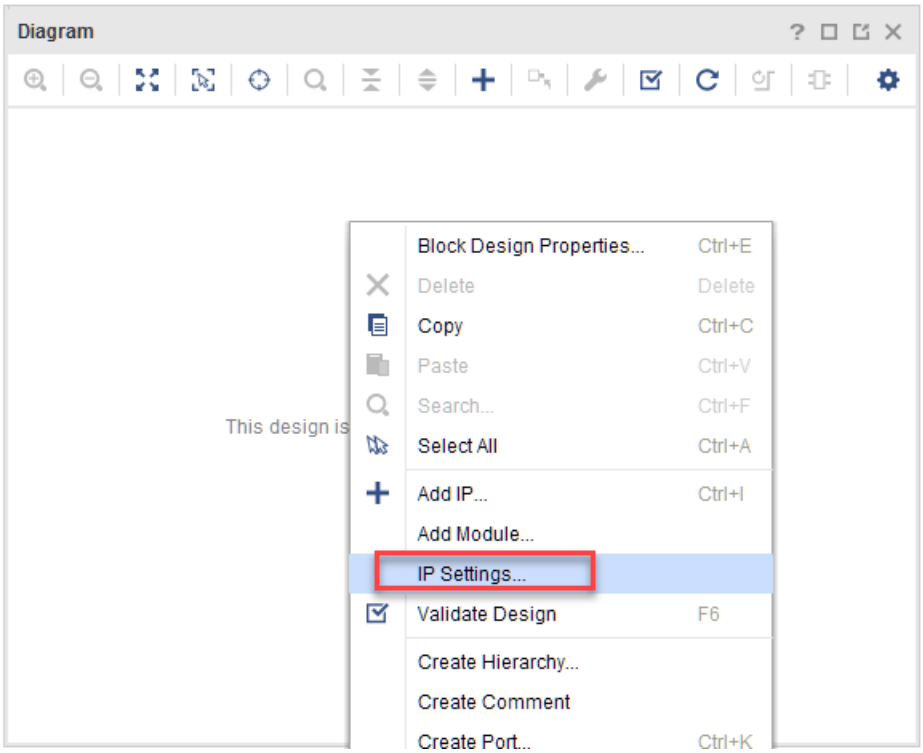
1. Click **Create Block Design** in the Flow Navigator pane.



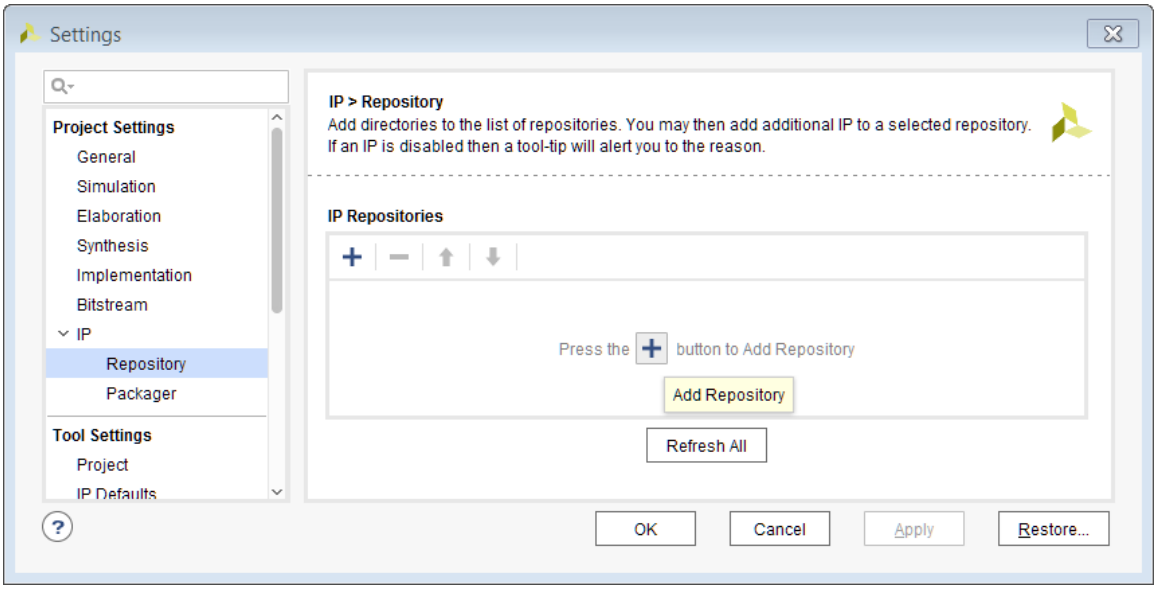
2. In the Create Block Design dialog box, click **OK** to accept the default name.

You will first create an IP repository for the Model Composer IP, and add the IP to the repository.

3. Right-click in the Diagram window, and select **IP Settings**.



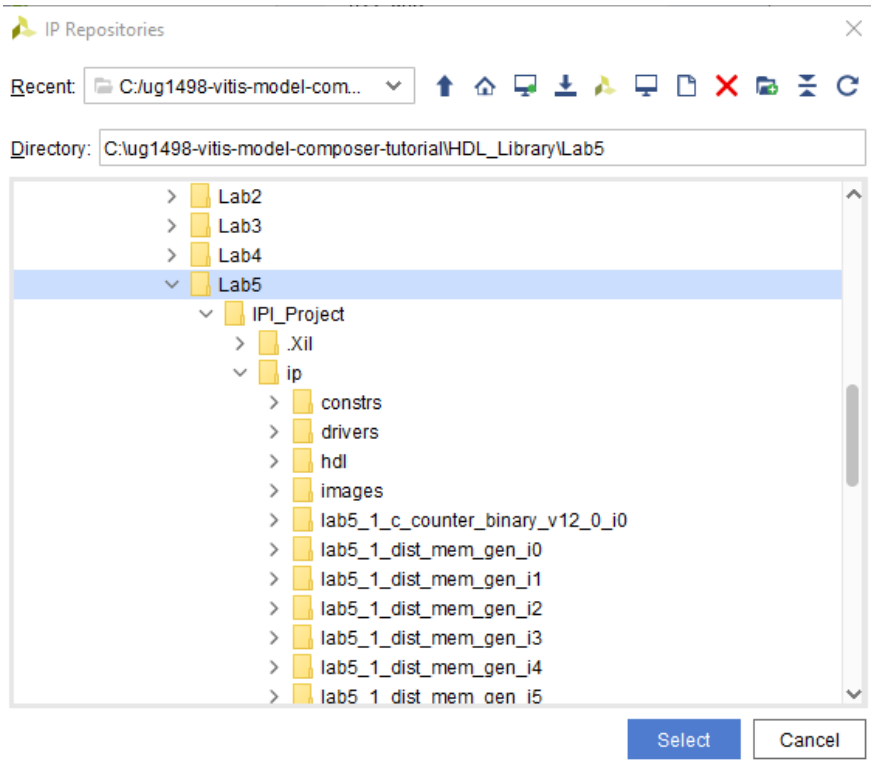
4. In the Settings dialog box, select **IP → Repository** under Project Settings and click the **Add Repository** button (+) to add a repository.



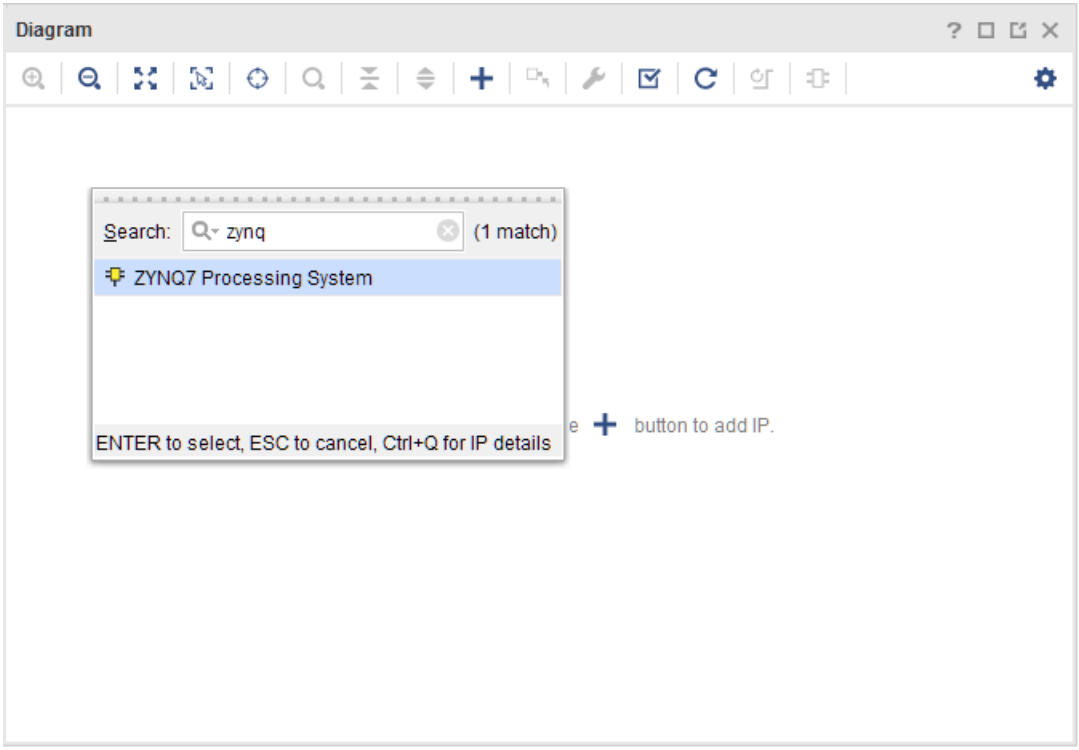
5. In the IP Repositories dialog box, navigate to the following directory:

```
C:\ug1498-model-composer-sys-gen-tutorial\HDL_Library\Lab5\IPI_Project\ip
```

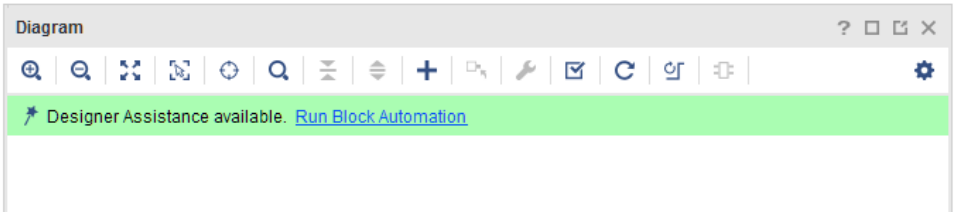
6. With folder `ip` selected, click **Select** to create the new repository as shown in the following figure.



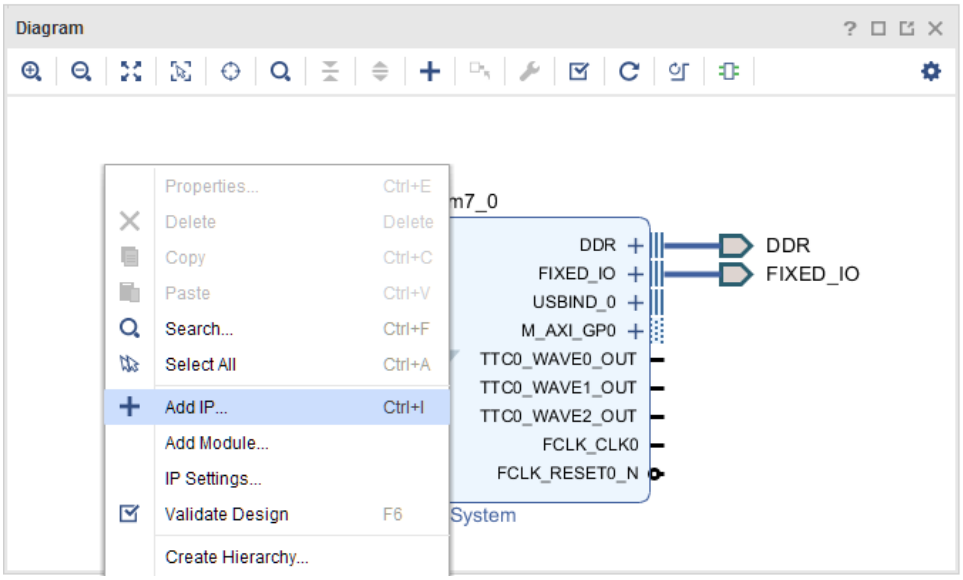
7. Click **OK** to exit the Add Repository dialog box.
8. Click **OK** to exit the Settings dialog box.
9. Click the **Add IP** button in the center of the canvas.
10. Type `zynq` in the Search field.
11. Double-click **ZYNQ7 Processing System** to add the CPU.



12. Click **Run Block Automation** as shown in the following figure.



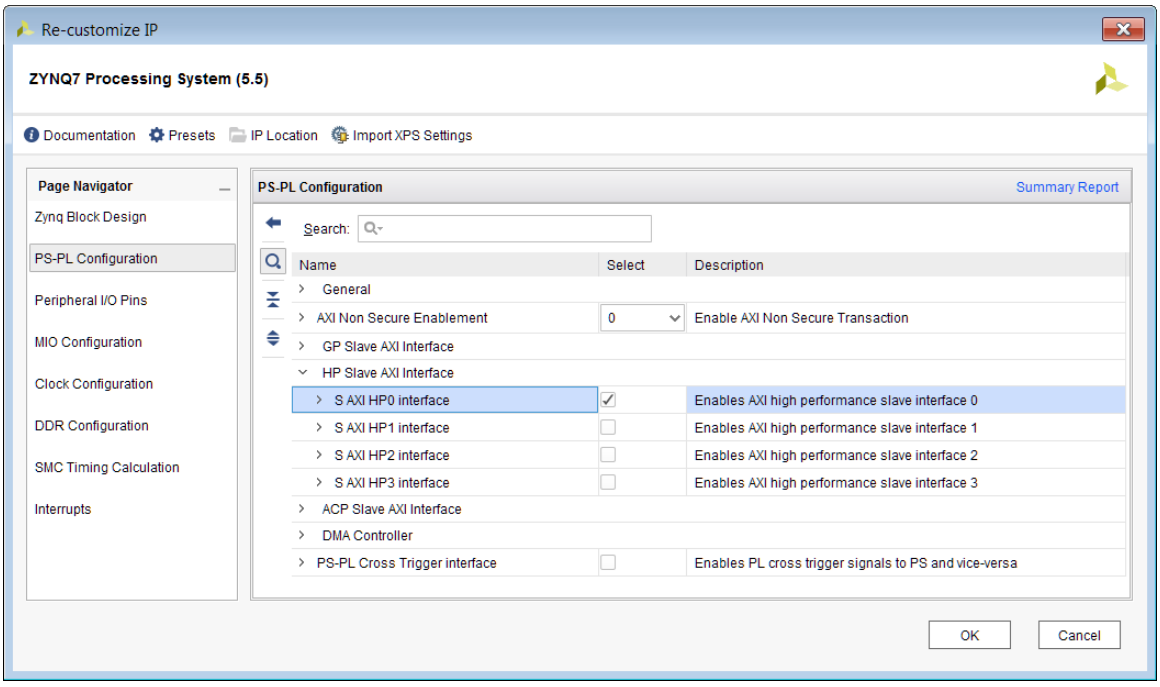
13. Leave Apply Board Preset selected and click **OK**. This will ensure the design is automatically configured to operate on the ZC702 evaluation board.
14. Right-click anywhere in the block diagram and select **Add IP**.



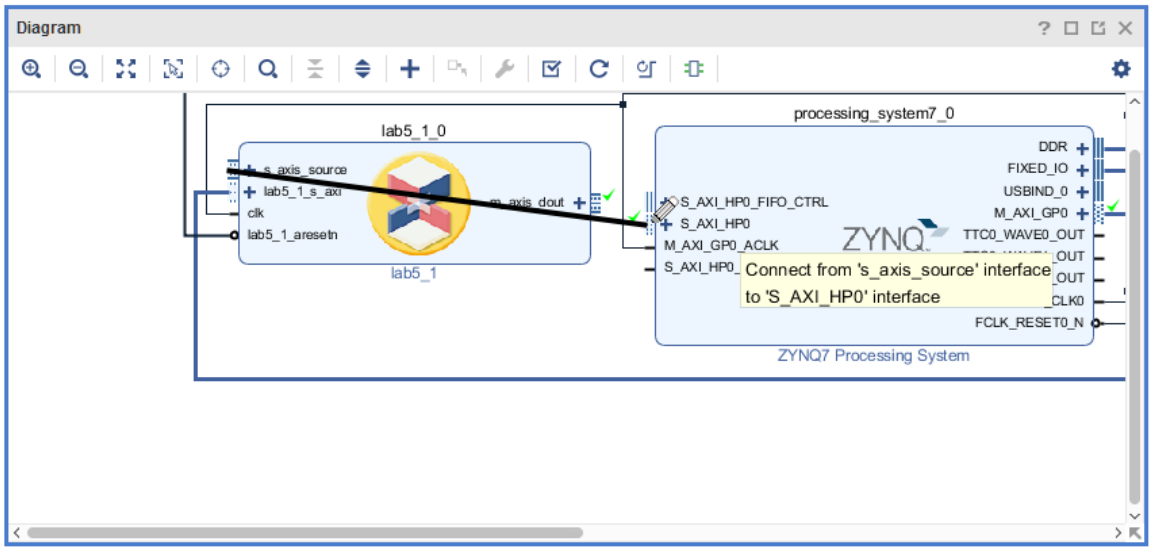
15. Type `lab5` in the Search dialog box.
16. Double-click **lab5_1** to add the IP to the design.

    You will now connect the IP to the rest of the design. Vivado IP integrator provides automated assistance when the design uses AXI interfaces.
17. Click **Run Connection Automation** (at the top of the design canvas).
18. Click **OK** to accept the default options (`lab5_1_0/lab5_1_s_axi` to `processing_system7_0/M_AXI_GP0`) and connect the AXI4-Lite interface to the Zynq®-7000 IP SoC.
19. Double-click the **ZYNQ7 Processing System** to customize the IP.
20. Click the **PS-PL Configuration** as shown in the following figure.
21. Expand the HP Slave AXI Interface and select the **S AXI HP0** interface.

    Make sure to check the box next to S AXI HP0 interface.

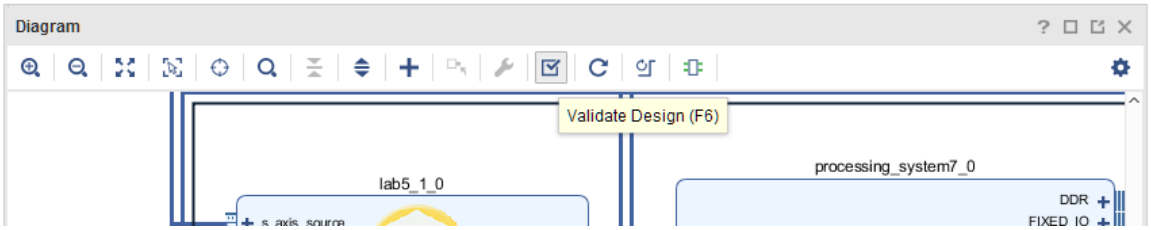


22. Click **OK** to add this port to the Zynq Processing System.
23. On the Model Composer IP lab5_1 block, click the AXI4-Stream input interface port `s_axis_source` and drag the mouse. Possible valid connections are shown with green check marks as the pencil cursor approaches them. Drag the mouse to the `S_AXI_HP0` port on the Zynq Processing System to complete the connection.



24. Click **OK** in the Make Connection dialog box.
25. Click **Run Connection Automation** to connect the AXI4-Lite interface on the AXI DMA to the processor.
26. Click **OK** to accept the default.

27. Use the Validate Design toolbar button to confirm the design has no errors.



28. Click **OK** to close the Validate Design message.

    The design from Model Composer has now been successfully incorporated into an IP integrator design. The IP in the repository can be used within any Vivado project, by simply adding the repository to the project.

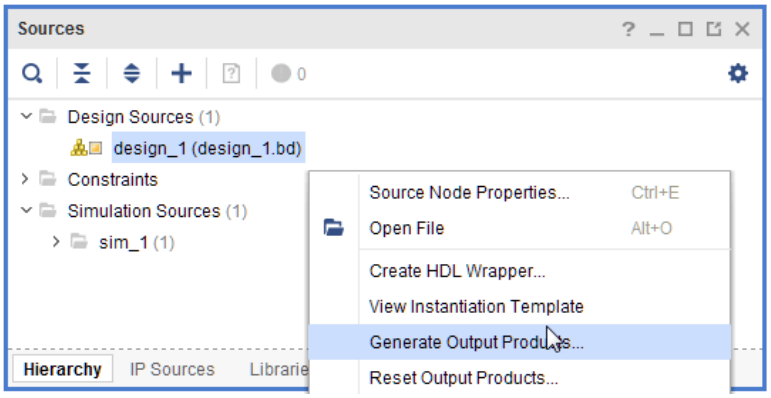
29. You will now process the design through to bitstream.

You will now process the design through to bitstream.

Step 4: Implement the Design

In this step, you will implement the IP integrator design and generate a bitsteam.

1. In the Flow Navigator, click **Project Manager** to return to the Project Manager view.
2. In the Sources browser in the main workspace pane, a Block Diagram object called `design_1` is at the top of the Design Sources tree view.
3. Right-click this object and select **Generate Output Products**.



4. In the Generate Output Products dialog box, click **Generate** to start the process of generating the necessary source files.
5. When the generation completes, right-click the `design_1` object again, select **Create HDL Wrapper**, and click **OK** (and let Vivado manage the wrapper) to exit the resulting dialog box.

   The top level of the Design Sources tree becomes the `design_1_wrapper.v` file. The design is now ready to be synthesized, implemented, and have an FPGA programming bitstream generated.

6. In the Flow Navigator, click **Generate Bitstream** to initiate the remainder of the flow.
7. Click **Yes**, and from the launch runs window click **OK** to generate the synthesis and implementation files.
8. In the dialog that appears after bitstream generation has completed, select **Open Implemented Design** and click **OK**.
9. After you view your implemented design, exit the Vivado IDE.

Summary

In this lab, you learned how AXI interfaces are added to a Model Composer design and how a Model Composer design is saved in the IP catalog format, incorporated into the Vivado IP catalog, and used in a larger design. You also saw how the IP integrator can substantially increase productivity with connection automation and hints when AXI interfaces are used in your design.

The following `solution` directory contains the final Model Composer (*.slx) files for this lab. The `solution` directory does not contain the IP output from Model Composer or the files and directories generated by Vivado.

```
\HDL_Library\Lab5\solution
```

Lab 6: Using a Model Composer HDL Design with a Zynq-7000 SoC

In this lab, you will learn how to export your Vivado® design with Model Composer HDL IP to a software environment and use driver files created by Model Composer to quickly implement your project on a Xilinx® evaluation board, running hardware with software in the same design.

Objectives

After completing this lab, you will have learned:

- How to export your Vivado design with Model Composer HDL IP to a software environment (Vitis™ software platform).
- How Model Composer automatically creates software driver files for AXI4-Lite interfaces.
- How to integrate the Model Composer driver files into your software application.

Procedure

This lab has two primary parts:

- In Step 1, you will review the AXI4-Lite interface and associated C drivers.

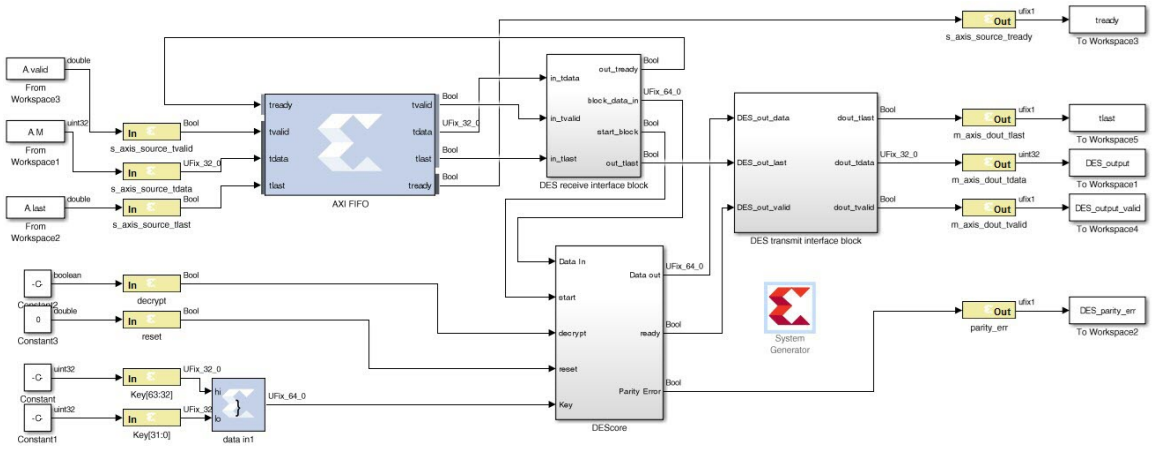
- In Step 2, you will export your Vivado design to a Vitis software environment and run it on a board.

Step 1: Review the AXI4-Lite Interface Drivers

In this step you review how AXI4-Lite interface drivers are provided when a design with an AXI4-Lite interface is saved.

This exercise uses the same design as Lab 5: Using AXI Interfaces and IP Integrator.

1. Invoke Vitis Model Composer and use the **Current Folder** browser to change the directory to: `\HDL_Library\Lab6`.
2. At the command prompt, type `open Lab6_1.slx`. This opens the design as shown in the following figure.



This design uses a number of AXI interfaces. These interfaces were reviewed in Lab 5: Using AXI Interfaces and IP Integrator and the review is repeated here with additional details on the AXI4-Lite register addressing.

- Using AXI interfaces allows a design exported to the Vivado IP Catalog to be efficiently integrated into a larger system using IP integrator.
- It is not a requirement for designs exported to the IP Catalog to use AXI interfaces. The design uses the following AXI interfaces:
  - An AXI4-Stream interface is used for ports `s_axis_source_*`. All Gateway In and Out signals are prefixed with same name (`s_axis_source_`) ensuring they are grouped into the same interface. The suffix for all ports are valid AXI4-Stream interface signal names (`tvalid`, `tlast`, and `tdata`).
  - An AXI4-Lite interface is used for the remaining ports. You can confirm this by performing the following steps:
3. Double-click **Gateway In decrypt** (or any of **reset**, **Keys[63:32]**, **Keys[31:0]**, **parity_err**).
4. In the Properties Editor select the **Implementation** tab.

5. Confirm the Interface is specified as AXI4-Lite in the Interface options.

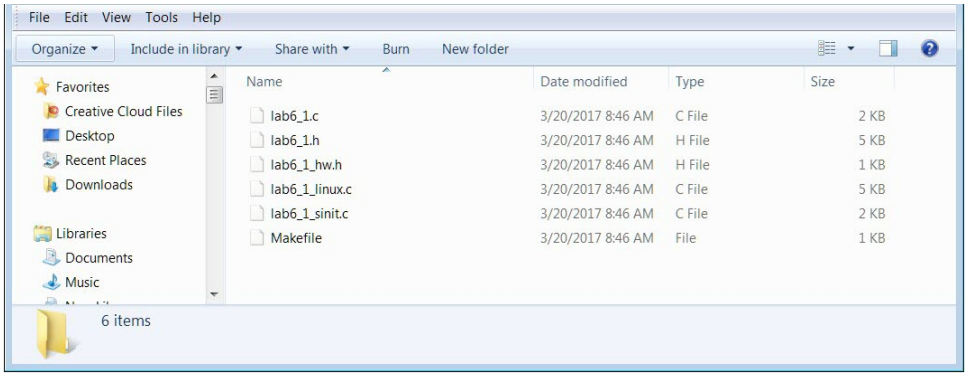
   Also note how the address of this port may be automatically assigned (as the current setting of **Auto assign address offset** indicates), or the address may be manually specified.

6. Click **OK** to exit the Properties Editor.

   Details on simulating the design are provided in the canvas notes. For this exercise, you will concentrate on exporting the design to the Vivado IP catalog and use the IP in an existing design.

7. In the System Generator token, select **Generate** to generate a design in IP Catalog format.
8. Click **OK** to dismiss the Compilation status dialog box.
9. Click **OK** to dismiss the System Generator token.
10. In the file system, navigate to the directory `\HDL_Library\Lab6\sys_gen_ip\ip\drivers\lab6_1_v1_2\src` and view the driver files.

    The driver files for the AXI4-Lite interface are automatically created by Model Composer when it saves a design in IP Catalog format.



11. Open file `lab6_1_hw.h` to review which addresses the ports in the AXI4-Lite interface were automatically assigned.

```
/**
*
* @file lab6_1_hw.h
*
* This header file contains identifiers and driver functions (or
* macros) that can be used to access the device.  The user should refer to the
* hardware device specification for more details of the device operation.
*/
#define LAB6_1_RESET 0x0/**< reset */
#define LAB6_1_DECRYPT 0x4/**< decrypt */
#define LAB6_1_KEY_63_32 0x8/**< key_63_32 */
#define LAB6_1_KEY_31_0 0xc/**< key_31_0 */
#define LAB6_1_PARITY_ERR 0x10/**< parity_err */
```

12. Open file `lab6_1.c` to review the C code for the driver functions. These are used to read and write to the AXI4-Lite registers and can be incorporated into your C program running on the Zynq®-7000 CPU. The function to write to the decrypt register is shown in the following figure.

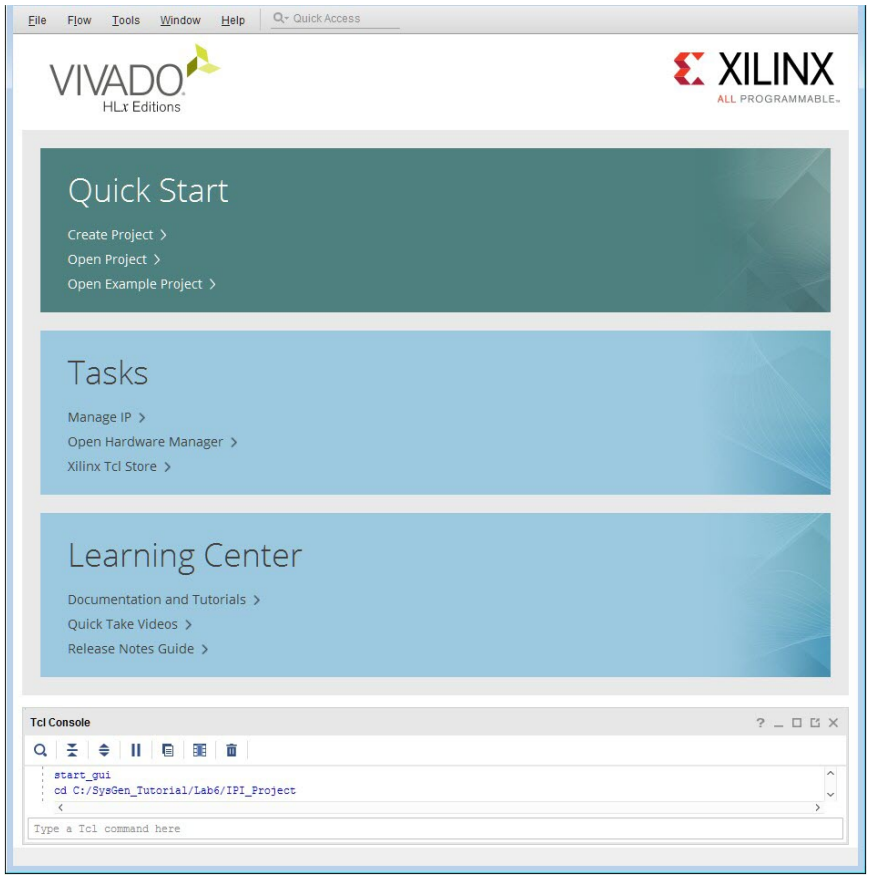
```
#include "lab6_1.h"
#ifndef __linux__
int lab6_1_CfgInitialize(lab6_1 *InstancePtr, lab6_1_Config *ConfigPtr) {
    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(ConfigPtr != NULL);

    InstancePtr->lab6_1_BaseAddress = ConfigPtr->lab6_1_BaseAddress;

    InstancePtr->IsReady = 1;
    return XST_SUCCESS;
}
#endif
void lab6_1_reset_write(lab6_1 *InstancePtr, u32 Data) {

    Xil_AssertVoid(InstancePtr != NULL);

    lab6_1_WriteReg(InstancePtr->lab6_1_BaseAddress, 0, Data);
}
u32 lab6_1_reset_read(lab6_1 *InstancePtr) {

    u32 Data;
    Xil_AssertVoid(InstancePtr != NULL);

    Data = lab6_1_ReadReg(InstancePtr->lab6_1_BaseAddress, 0);
    return Data;
}
void lab6_1_decrypt_write(lab6_1 *InstancePtr, u32 Data) {

    Xil_AssertVoid(InstancePtr != NULL);

    lab6_1_WriteReg(InstancePtr->lab6_1_BaseAddress, 4, Data);
}
```

The driver files are automatically included when the Model Composer design is added to the IP Catalog. The procedure for adding a Model Composer design to the IP Catalog is detailed in Lab 5: Using AXI Interfaces and IP Integrator. In the next step, you will implement the design.

Step 2: Developing Software and Running it on the Zynq-7000 System

1. Open the Vivado IDE:
   - Click **Windows → Xilinx Design Tools → Vitis 2021.1**.

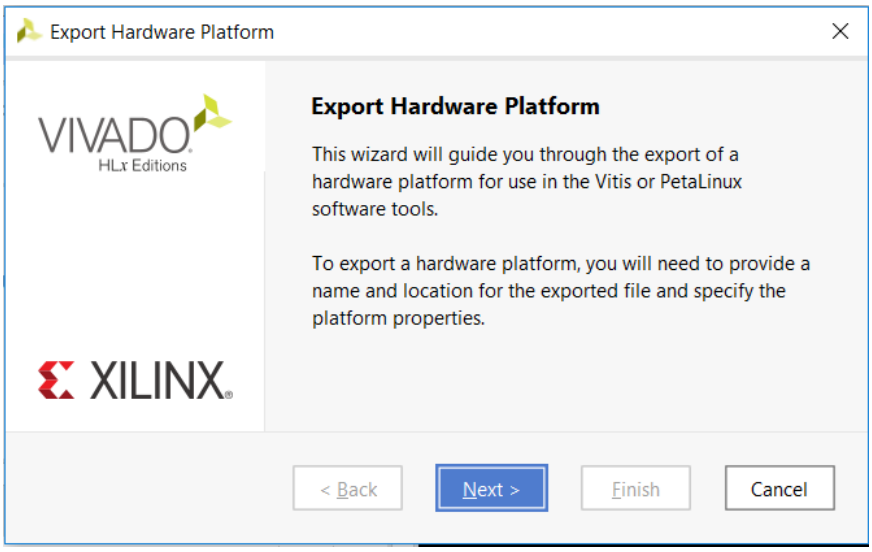
     In this lab you will use the same design as Lab 5: Using AXI Interfaces and IP Integrator, but this time you will create the design using a Tcl file, rather than the interactive process.
2. Using the Tcl console as shown in the following figure:
   a. Type `cd C:\ug1498-model-composer-sys-gen-tutorial\HDL_Library\Lab6\IPI_Project` to change to the project directory.
   b. Type `source lab6_design.tcl` to create the RTL design.



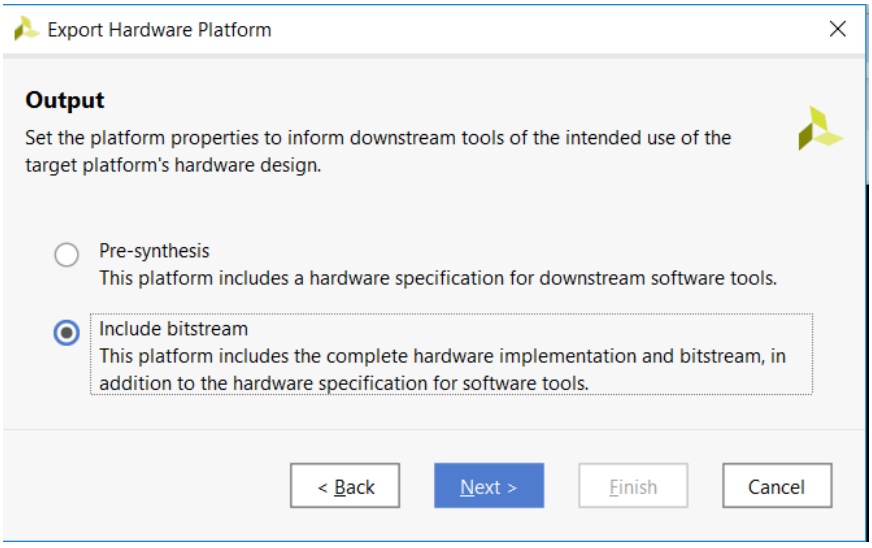
This creates the project, creates the IP integrator design and builds the implementation (RTL synthesis, followed by place and route). This may take some time to complete (same as the final step of Lab 5: Using AXI Interfaces and IP Integrator).

When it completes:

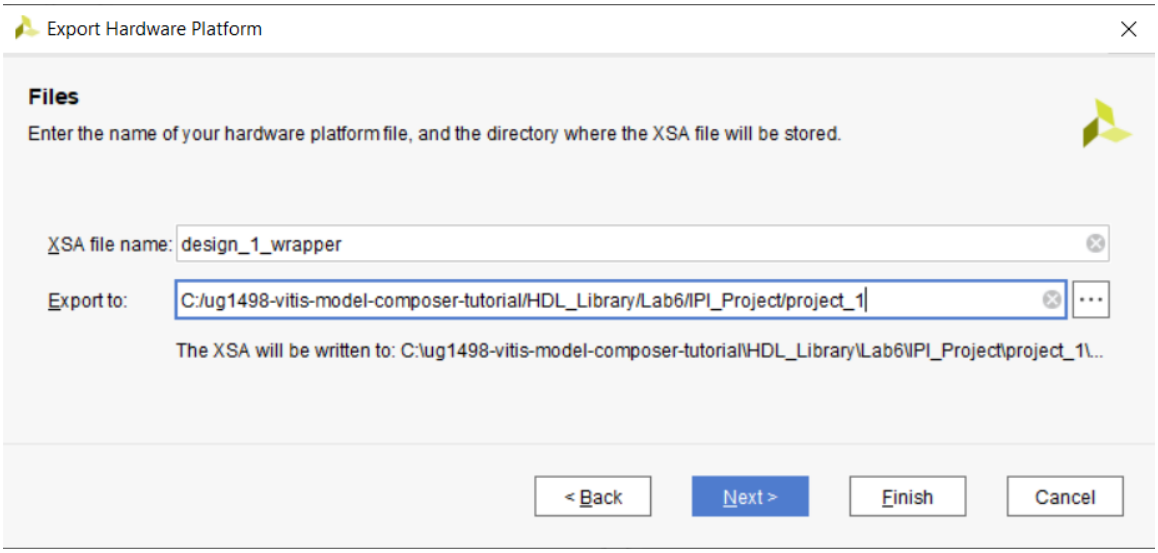
3. Click **Open Implemented Design** in the Flow Navigator pane.
4. From the Vivado IDE main menu select **File → Export → Export Hardware**.
5. Click **Next** in the Export Hardware Platform page.



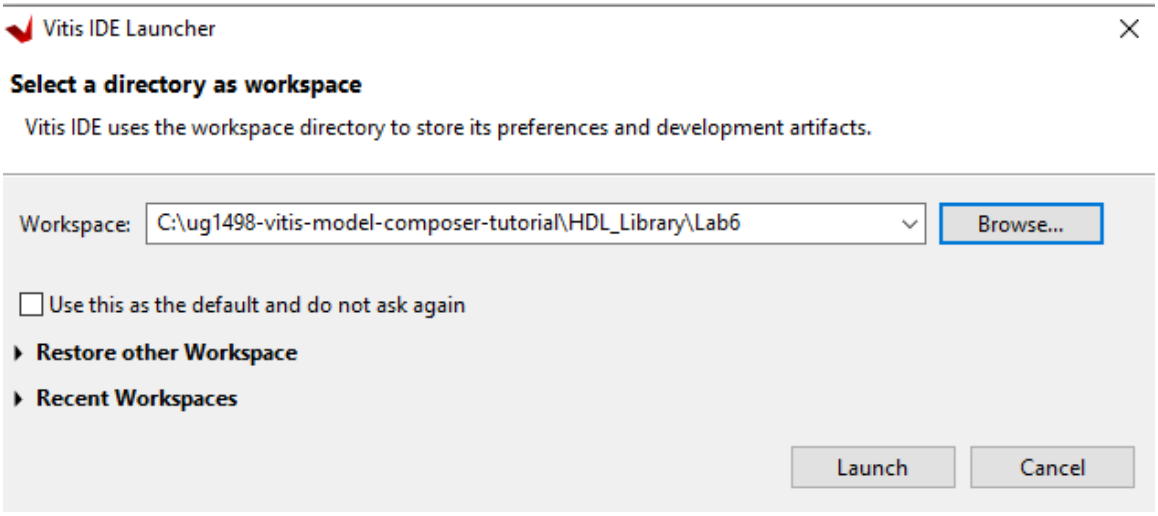
6. Select the **Include Bitstream** option in the Output page and click **Next**.



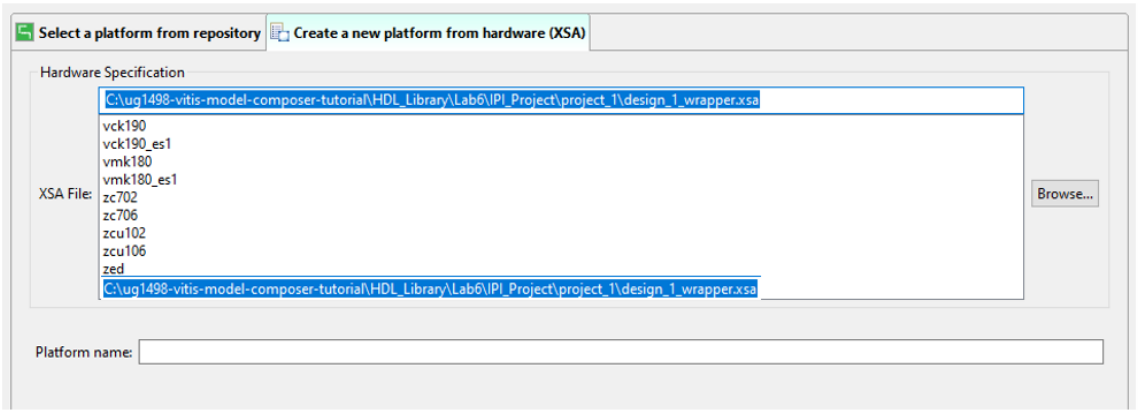
7. Leave the **XSA file name** and the **Export to** fields at the default setting and click **Next**.



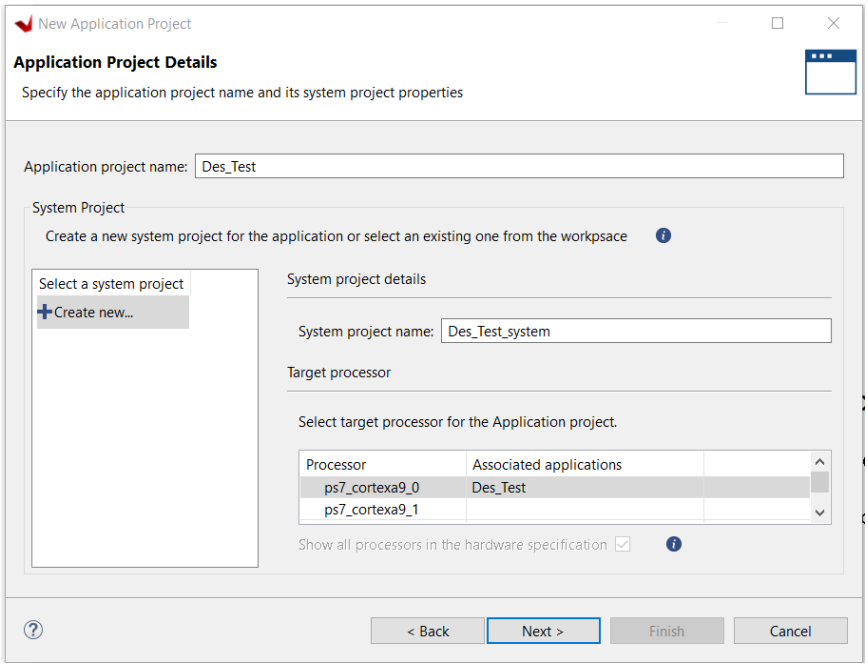
8. Click **Finish** to export the hardware.
9. Open the Vitis IDE:
   - Click **Windows → Xilinx Design Tools → Vitis 2021.1**.
10. Select the workspace space directory to store preferences and click **Launch**.



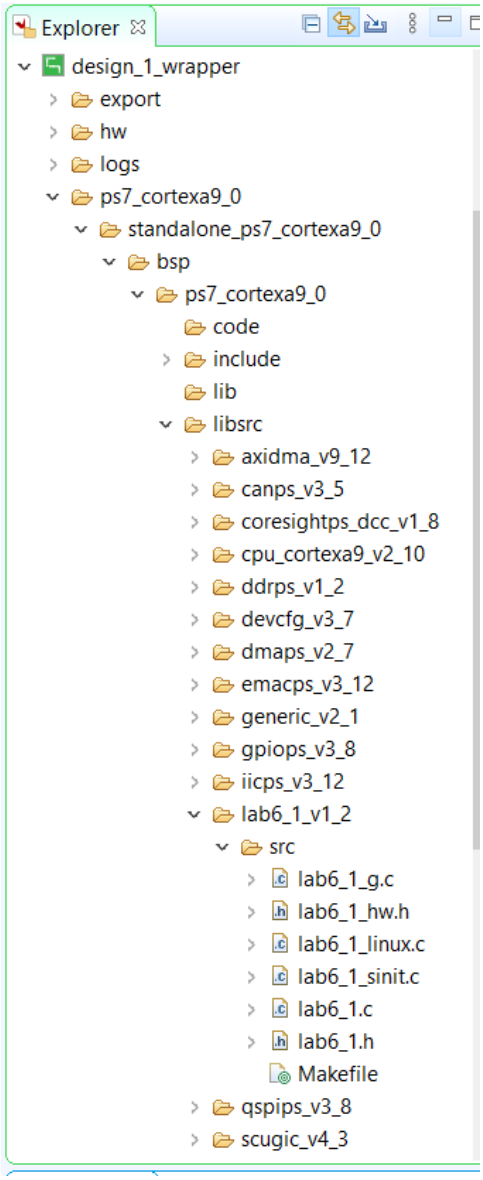
11. From the Vitis IDE, select **Create Application Project**.
12. Click **Next** in the Welcome page.
13. Switch to the **Create a new platform from hardware(XSA)** tab and click **Browse** to create a custom platform from the XSA.
14. Navigate to **Lab6 → IPI_Project → project_1**, select **design_1_wrapper.xsa** and click **Open**.



15. Enter the application project name `Des_Test` in the Application project name field.
16. In the Target processor section, select the processor **ps7_cortexa9_0** and click **Next**.

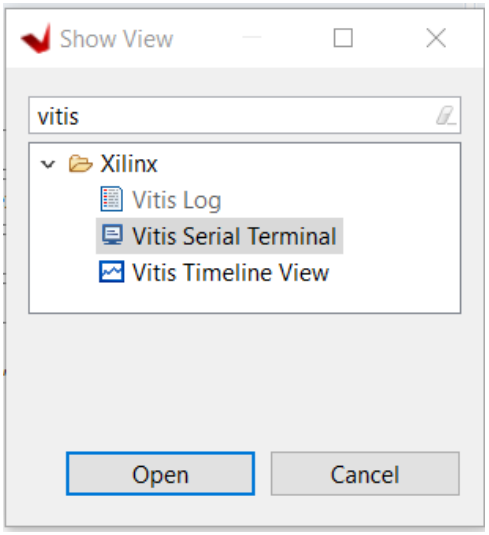


17. Click **Next**.
18. In the Domain page ensure the CPU selected is **ps7_cortexa9_0** and click **Next**.
19. Select the **Hello World** template and click **Finish**.
20. Expand the **design_1_wrapper** container as shown to confirm the AXI4-Lite driver code is included in the project.

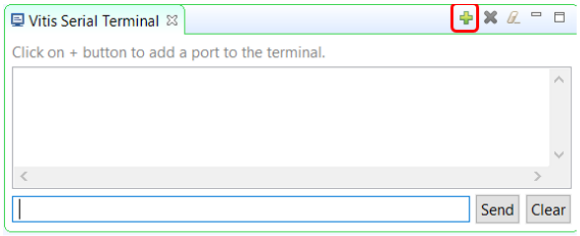


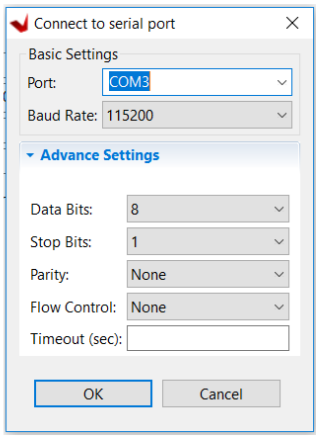
21. Power up the ZC702 board to program the FPGA.
22. Click **Xilinx → Program Device** and from the resulting window, click **Program**.

    The Done LED (DS3) goes ON, on the FPGA board.
23. Click **Window → Show View** and in the Show view window, type `Vitis`, select **Vitis Serial Terminal** and click **Open**.



24. To set up the terminal in the Vitis Serial Terminal view, click the **+** icon and perform the following:





    a. Select the COM port to which the USB UART cable is connected. On Windows, if you are unsure, open the **Device Manager** and identify the port with the "Silicon Labs" driver under Ports (COM & LPT).

    b. Change the Baud Rate to `115200`.

    c. Click **OK** to exit the Terminal Settings dialog box.

    d. Check that the terminal is connected by the message in tab title bar.

25. Right-click the application project **Des_Test** in the Explorer view, select **Build Project**.

    When this completes, you will see the message “Build Finished” in the console.

26. Right-click on application project **Des_Test**, select **Run As → Launch on Hardware**.

27. Switch to the **Vitis Serial Terminal** tab and confirm that `Hello World` was received.

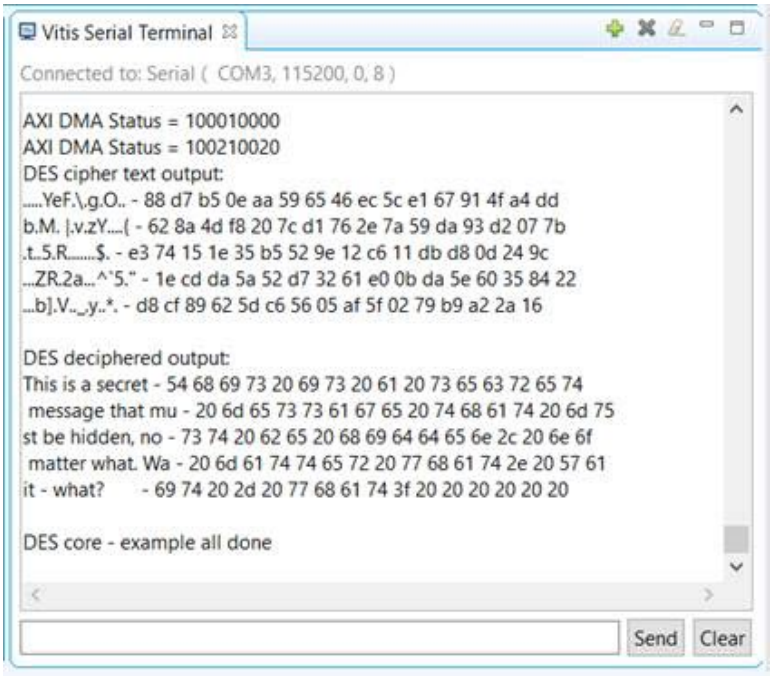
28. Expand the container `Des_Test` and then expand the container `src`.

29. Double-click the **helloworld.c** file.

30. Replace the contents of this file with the contents of the file `hello_world_final.c` from the `lab6` directory.
31. Save the `helloworld.c` source code.
32. Right-click application project `Des_Test` in the Explorer view, and select **Build Project**.

    When this completes, you will see the message "Build Finished" in the console.
33. Right-click again and select **Run As → Launch on Hardware**.

    Note: If a window opens displaying the text "Run Session is already active", click **OK** in that window.
34. Review the results in the Vitis Serial Terminal tab (shown in the following figure).



Summary

In this lab, you learned how to export your Vivado IDE design containing Model Composer HDL IP to the Vitis software environment and to integrate the driver files automatically created by Model Composer to run the application on the ZC702 board. You then viewed the results of the acceleration.

The following solutions directory contains the final Model Composer (`*.slx`) files for this lab. The solutions directory does not contain the IP output from Model Composer, the files and directories generated when the Vivado IDE is executed, or the Vitis workspace.

`\HDL_Library\Lab6\solution`.

HLS Library

Lab 1: Introduction to Model Composer HLS Library

This tutorial shows how you can use Model Composer HLS Library 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 HLS library.
- In Step 2, you build a simple design using HLS 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 HLS Library

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

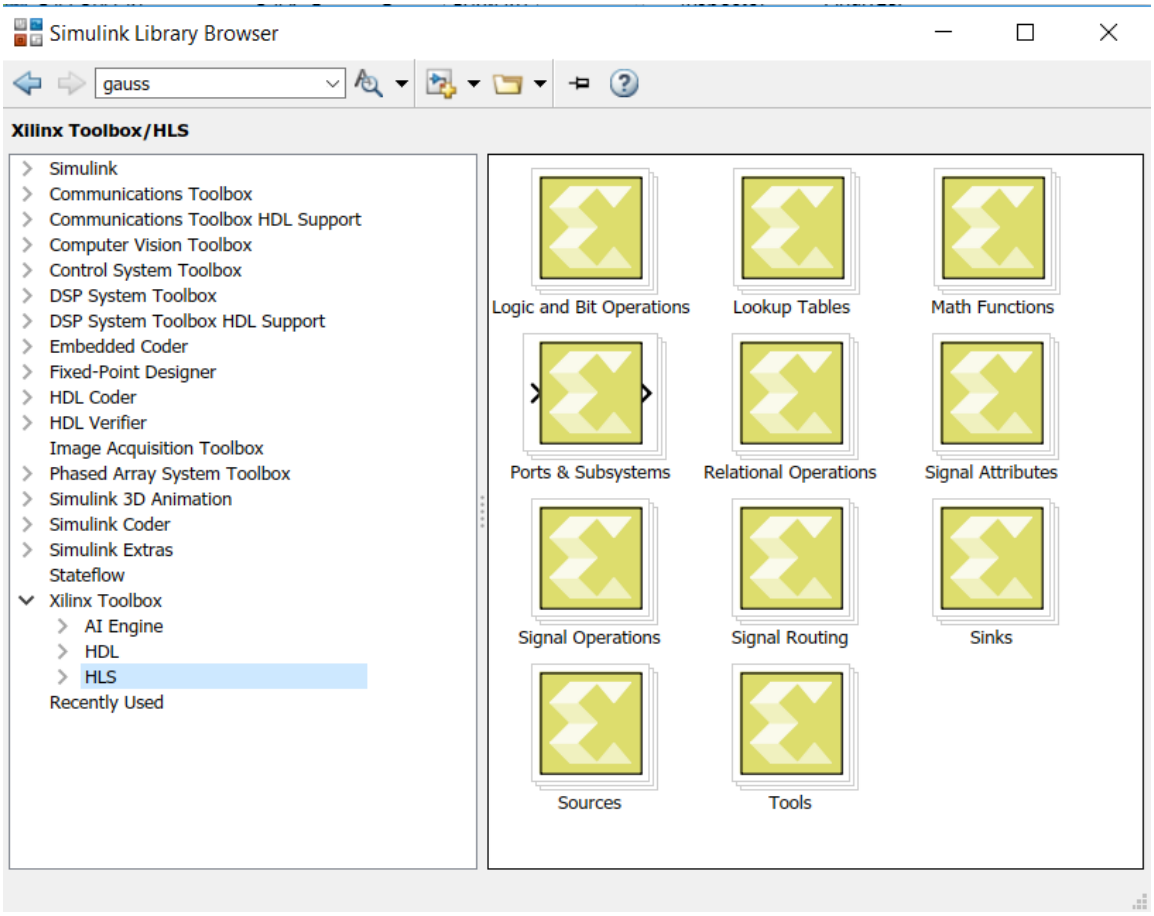
Access the HLS Library

Model Composer provides an HLS Library 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, the click **Blank Model**, 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 HLS library in the Xilinx Toolbox.



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

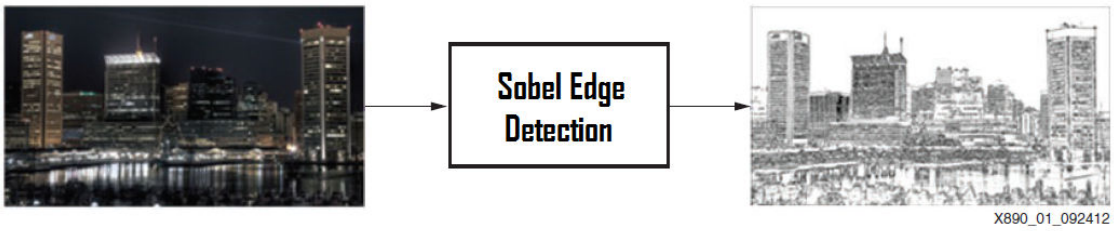
Step 2: Build Designs with HLS Blocks

In this step, you build a simple design using the existing HLS 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**

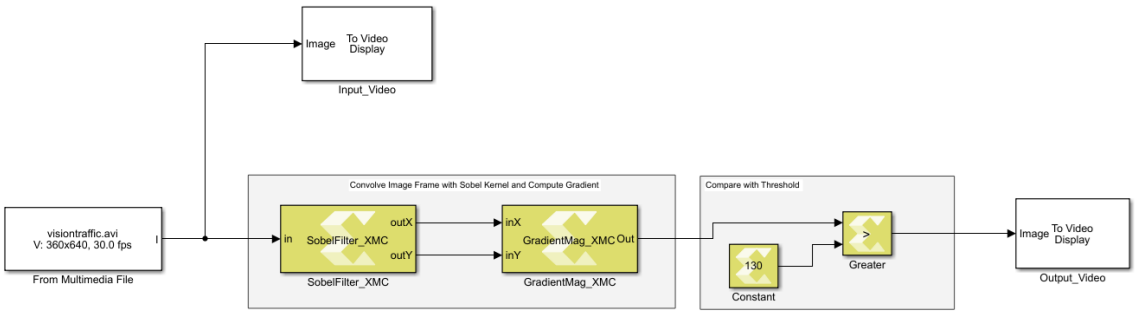


Implementing Algorithm in Model Composer

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

   This model already contains source and sink blocks to stream video files as input directly into your algorithm and view the results. The model also contains some of the needed HLS blocks required for this section. Note the difference in appearance for the HLS blocks in the design versus the Simulink blocks.
3. Double-click **Sobeledge_lib.slx** library model and and drag the SobelFilter_XMC 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.
4. Select the **GradientMag_XMC** block from the `Sobeledge_lib.slx` file and 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.

   Note: The blocks SobelFilter_XMC and GradientMag_XMC have been generated using the `xmcImportFunction` feature.

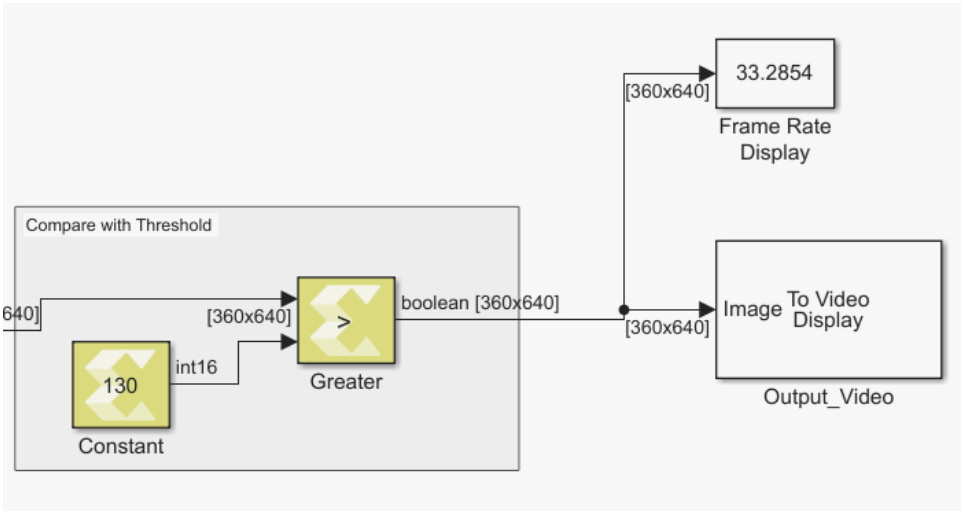


6. Select the **Simulation → Run** command or click the  button to simulate the model and view the results of the Sobel Edge Detection algorithm.

   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 changing 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.

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.

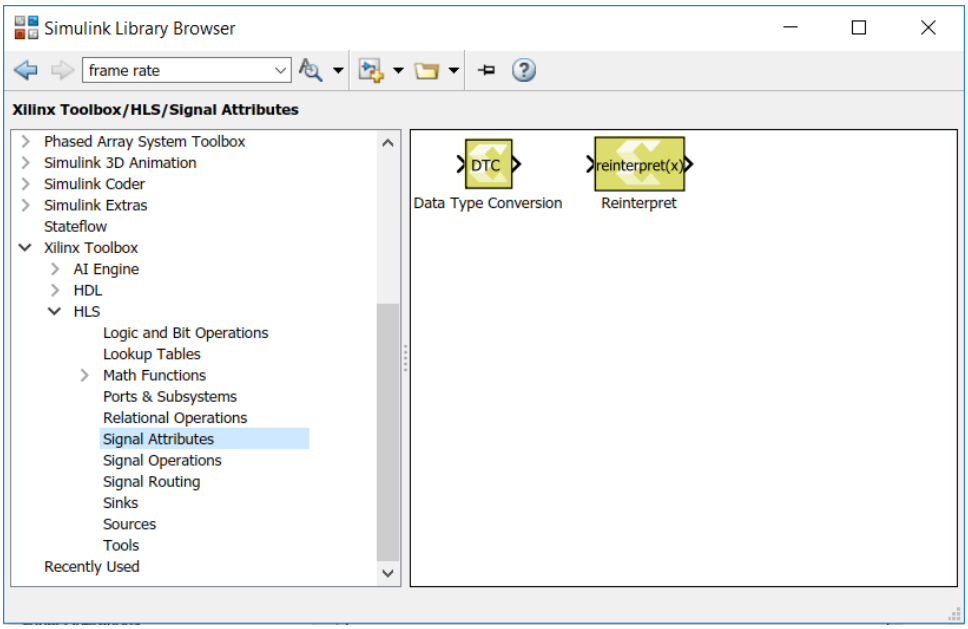
   This is a Color Space conversion design, built with basic Model Composer blocks, that performs a RGB to YCbCr conversion.
3. Update the model (**Ctrl+D**) and observe that the Data Types, Signal Dimensions and Sample Times from the Source blocks in Simulink all propagate through the Model Composer blocks. Note that the design uses single precision floating point data types.
4. Simulate the model and observe the results from simulation.

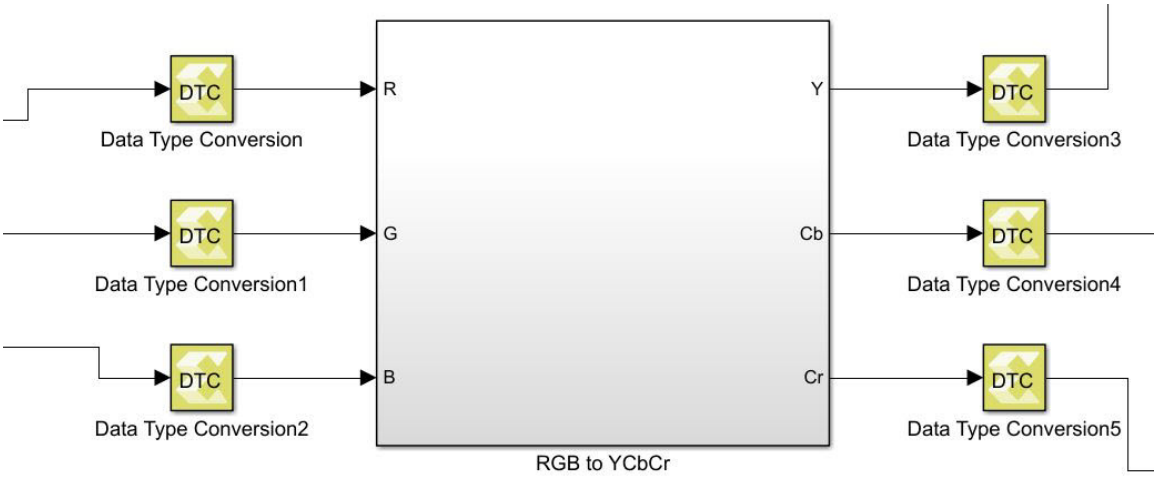
Convert Data Types

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

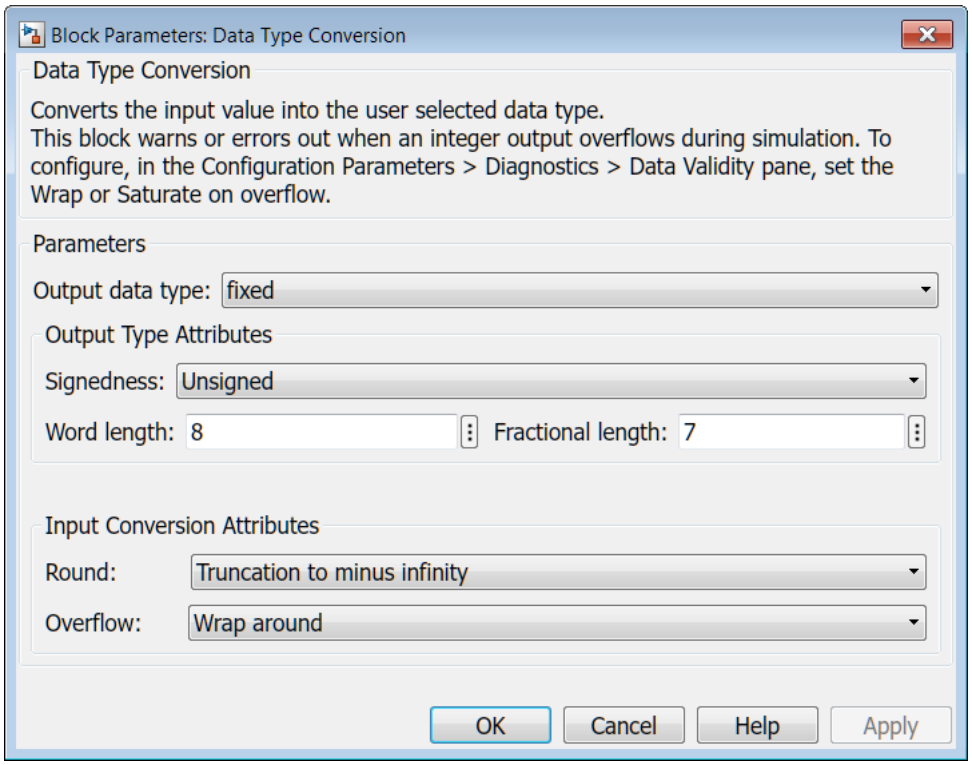
Note: Fixed point representation helps to achieve optimal resource usage and performance for a usually acceptable trade-off in precision, depending on the dataset/algorithm.

1. Double-click **ColorSpace_Conversion_fixed_start.slx** in the Current Folder to open the design.
2. Open the **HLS** 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.

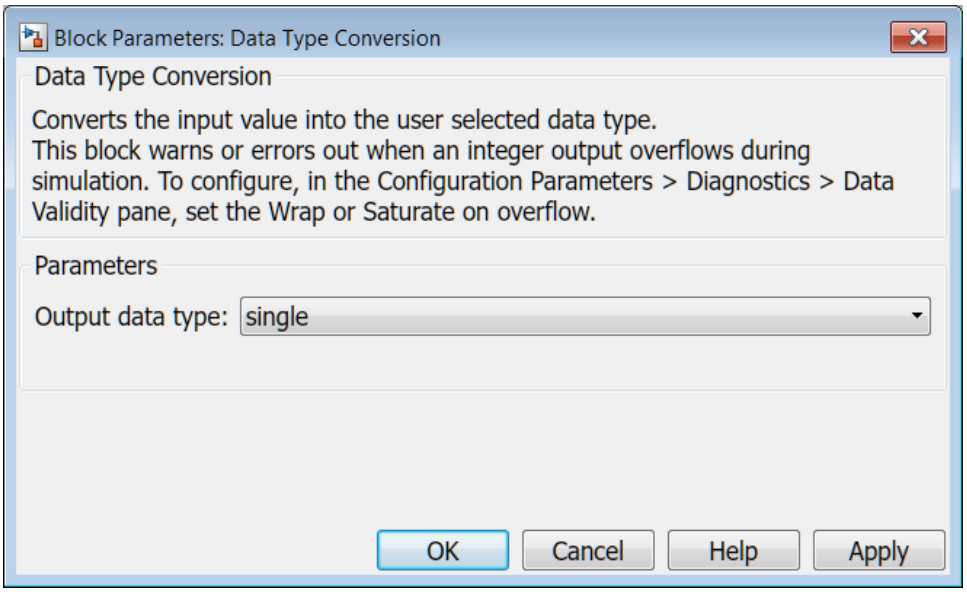




4. Open the Data Type Conversion blocks at the inputs of the RGB to YCbCr Subsystem, and do the following:
   - Change the **Output data type** parameter to **fixed**.
   - Set the **Signedness** to **Unsigned**.
   - Set the **Word length** to **8**.
   - Set **Fractional length** to **7**.
   - Click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

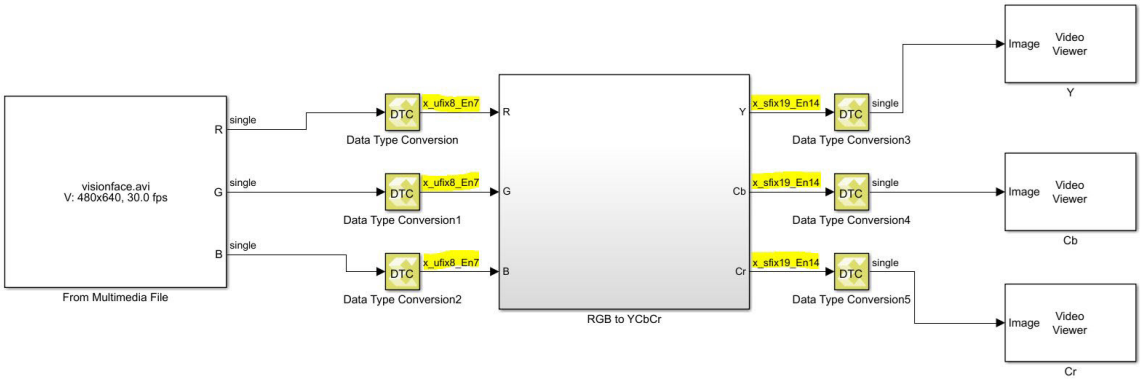
You can control the fixed point types for the gain parameter in the Gain blocks and the value in the Constant blocks. You can do this by opening up the **Calculate_Y**, **Calculate_Cb**, and **Calculate_Cr** blocks and setting the data types as follows.

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**
- **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

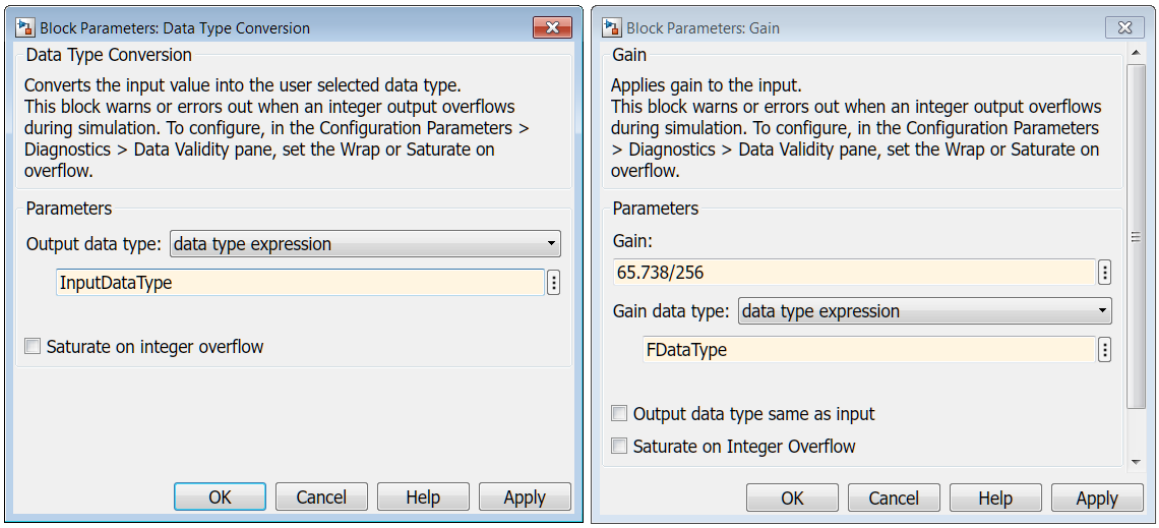
For example, `x_sfix16_En8` represents a signed fixed point number with Word Length=16 and Fractional Length=8.

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 HLS 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 HLS 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

Table 2: **Overflow Index** *(cont'd)*

Index	Meaning
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 HLS blocks directly to native Simulink blocks.
- How the HLS 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:

- `\HLS_Library\Lab1\Section1\solution`
- `\HLS_Library\Lab1\Section2\solution`

Lab 2: Importing Code into Model Composer

Model Composer lets you import Vitis™ 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 *Vitis Model Composer User Guide* (UG1483).

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 *Vitis Model Composer User Guide* (UG1483).

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

   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])
   ```

2. 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.

3. 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.

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

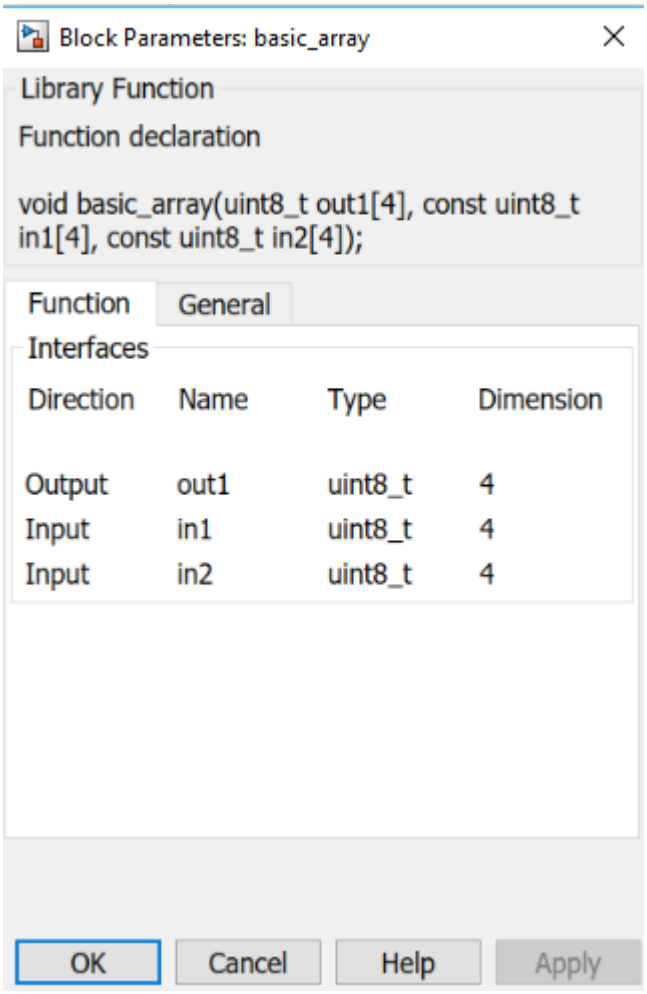
```
>>run('import_function.m')
```

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

   Save this Simulink library model.

5. 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 *Vitis Model Composer User Guide* (UG1483).

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.

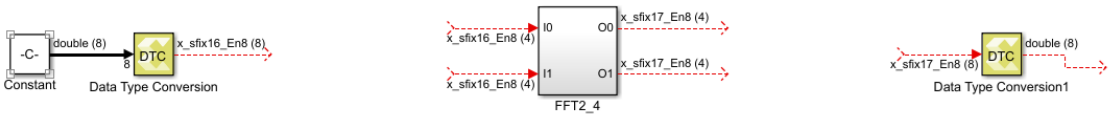
6. Observe the arithmetic operations performed using template variables as shown below, indicating the output signal length is half of the input signal length.

```
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.



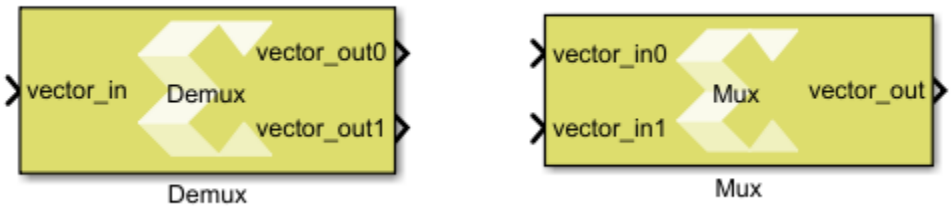
8. Double-click the **import_function.m** script file in the MATLAB command window and observe the following commands that generate library blocks to embed into your actual design.

```
>>xmcImportFunction('design_lib',{'Demux'},'template_design.h',{},
{},'override','unlock')
>>xmcImportFunction('design_lib',{'Mux'},'template_design.h',{},
{},'override','unlock')
```

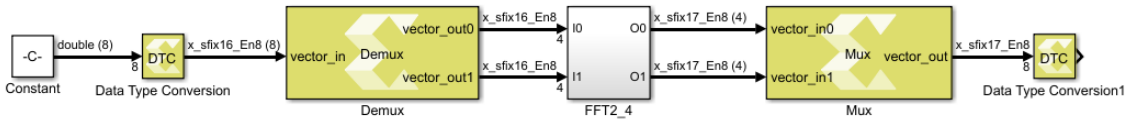
9. Run the `import_function.m` script from the MATLAB command line:

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

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

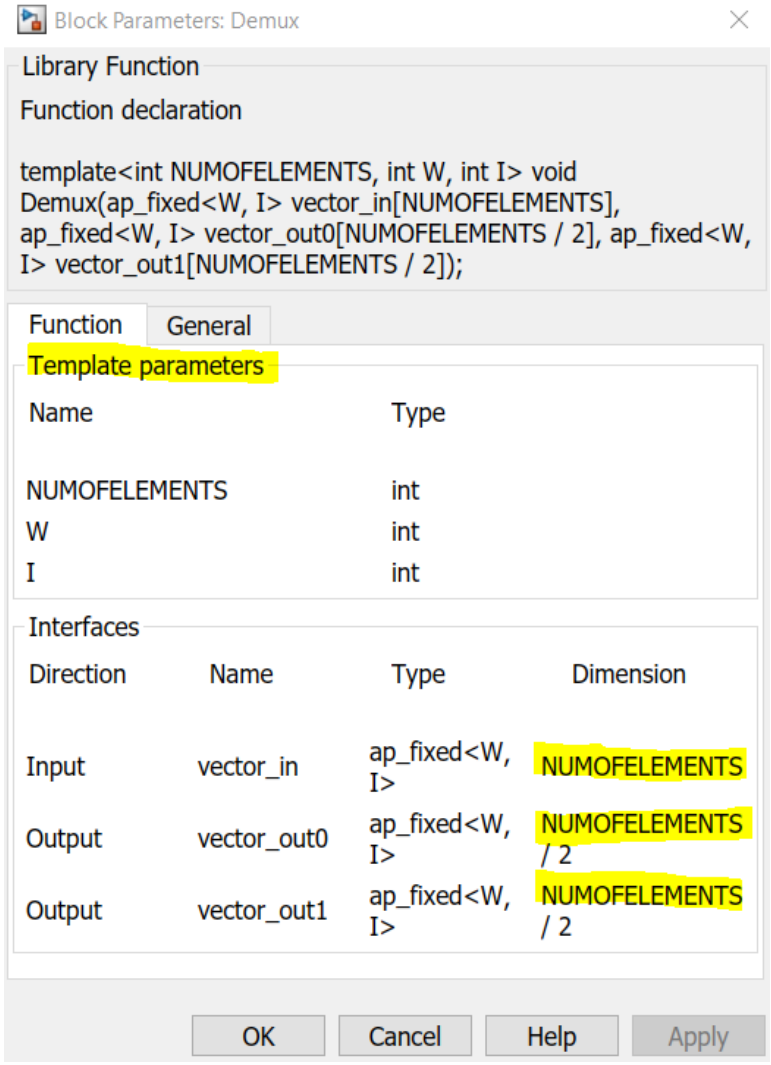


11. Copy the Demux and Mux blocks and paste them in the `design.slx` file and connect them as shown in the following figure.



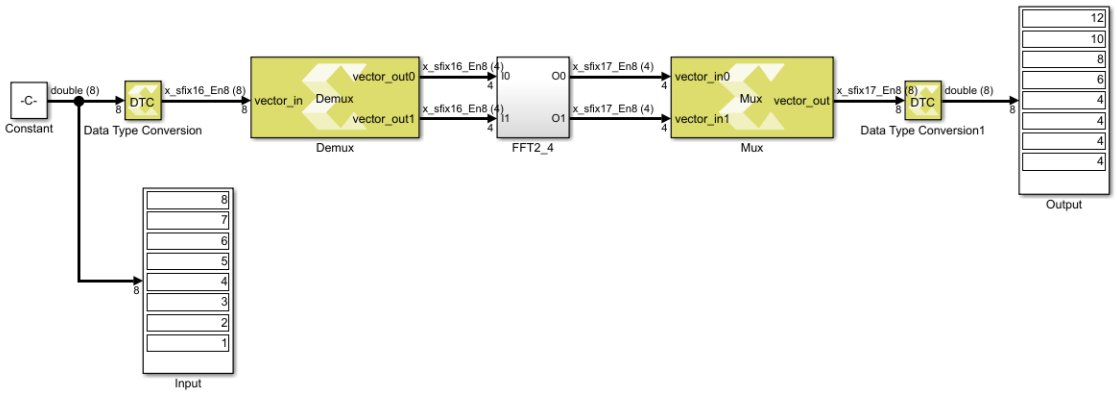
12. Note the following after embedding the custom blocks:
    a. Double-click the Constant block and observe the vector input of type double. SSR is a workspace variable, initially set to 8 from the `initFcn` model callback.
    b. Using the Data Type Conversion (DTC) block, double type is converted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

       Input is configurable to any word length since the design is templatized.
    c. Double-click the Demux block and observe the Template parameters section and Dimension column in the Interface section of the function tab.

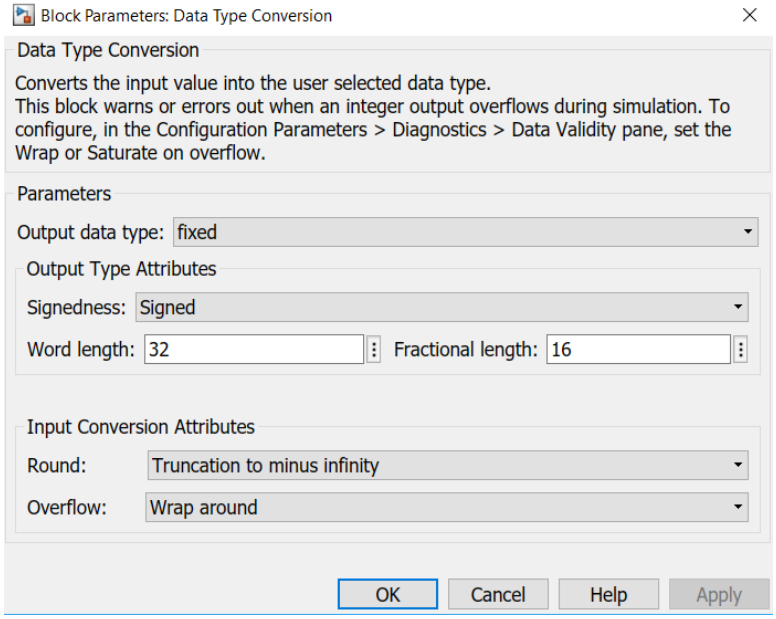


d. Next, double-click the Mux block and observe the Template parameters and Dimension.

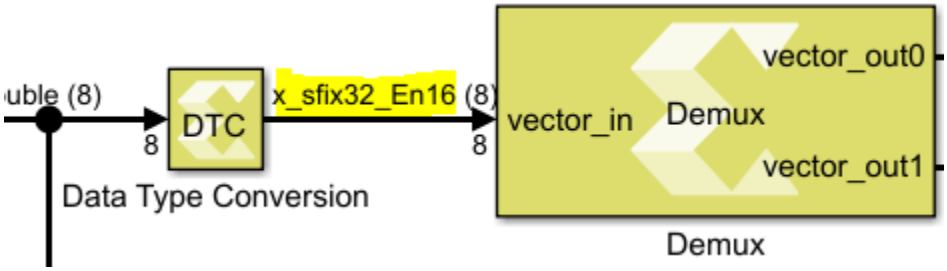
13. Add a Display block at the input and output as shown in the following figure and simulate the model to observe the results.



14. To understand how templatized 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.

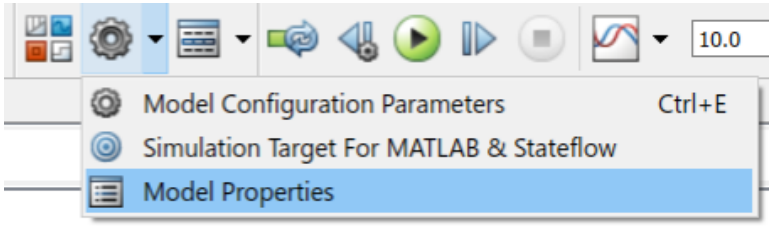


    d. Click **OK** and press **Ctrl+D**. Observe the signal dimensions in the design.

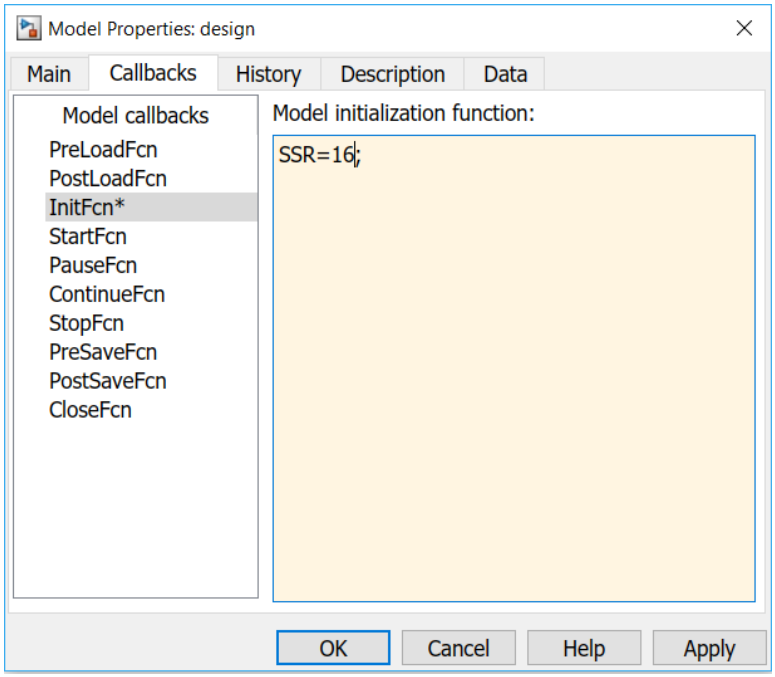


    To make sure the output is correct, run the simulation and observe that the same block can still be used in a generic way for different values of Word length and Fractional length. This is possible only because we have templatized the `W` and `I` values in our `C` design.

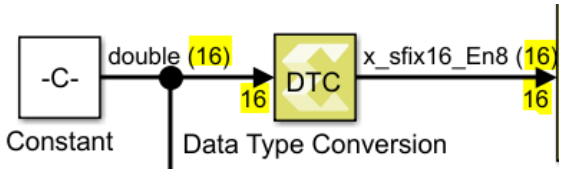
15. For an additional understanding of template parameters, perform the following:
    a. Click the arrow mark beside the Model Configuration Parameters icon and select the **Model Properties** option.



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 *Vitis Model Composer User Guide* (UG1483).

Conclusion

In this lab, you learned:

- How to create a custom block using the xmcImportFunction 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 Vitis HLS fixed-point data types

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

- `\HLS_Library\Lab2\Section1\solution`
- `\HLS_Library\Lab2\Section2\solution`

Lab 3: 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.

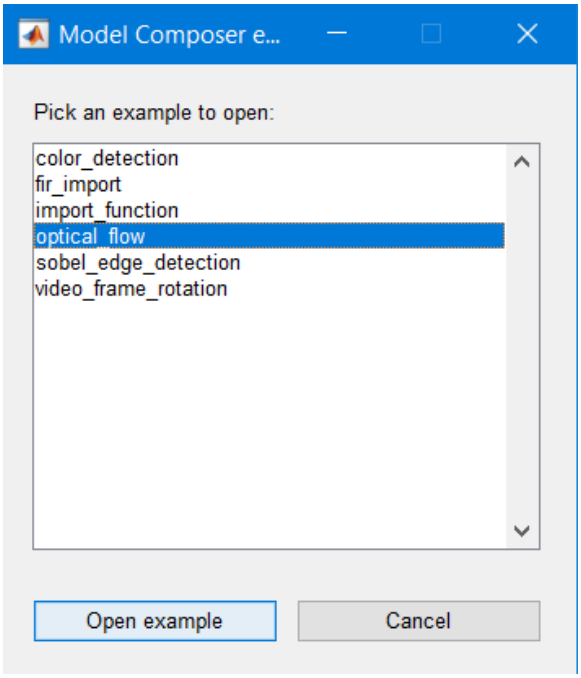
For more details and information about how to create custom blocks, follow this link in *Model Composer User Guide* (UG1262).

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

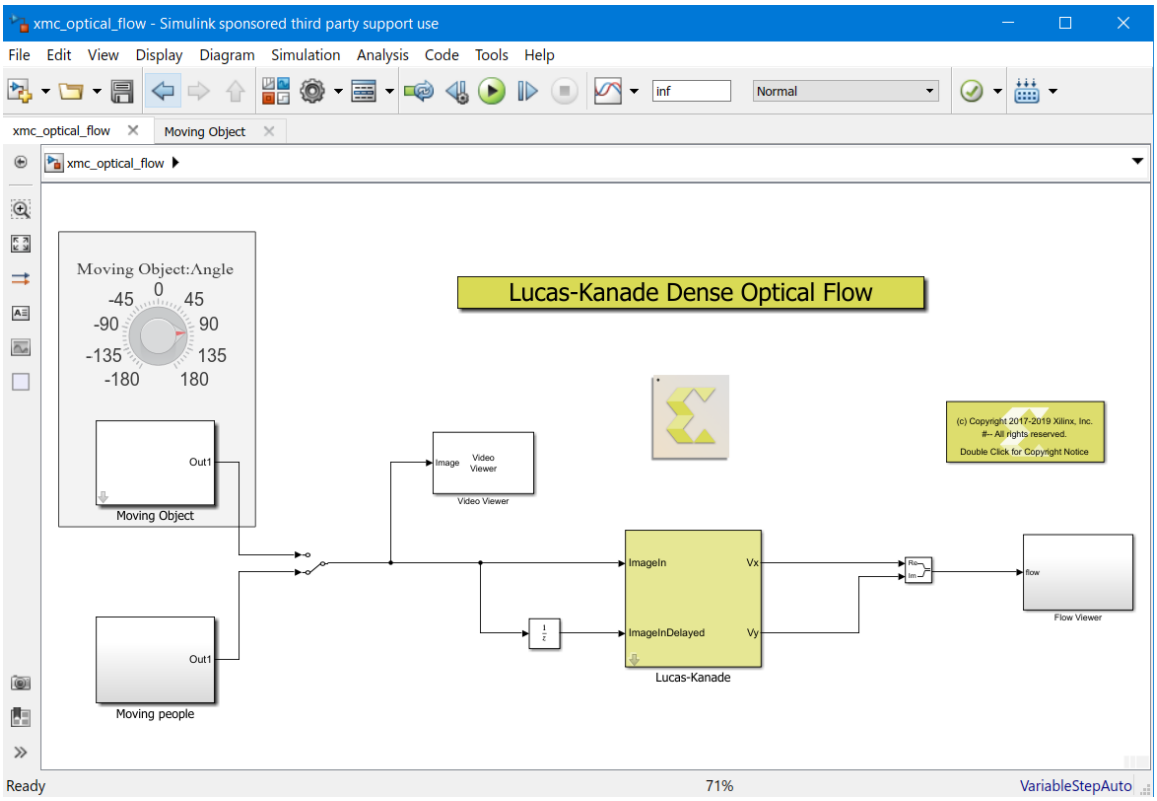
1. Type the following at the MATLAB command prompt:

```
>> xmcOpenExample
```

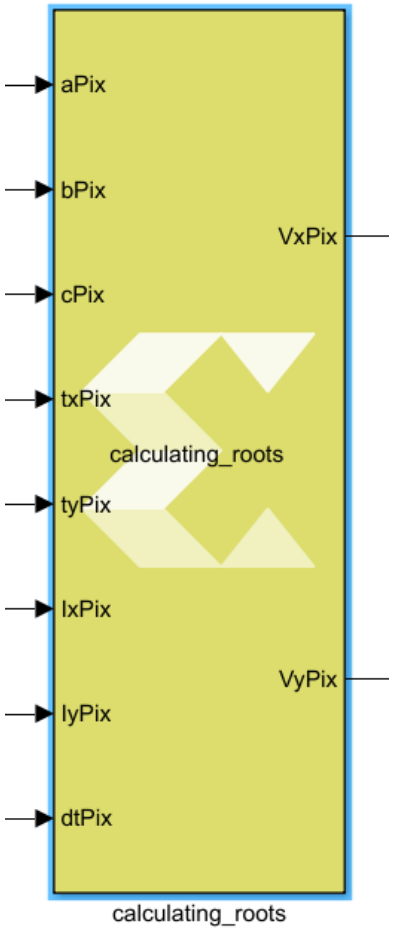
2. Press **Enter** to open the Model Composer examples dialog box.



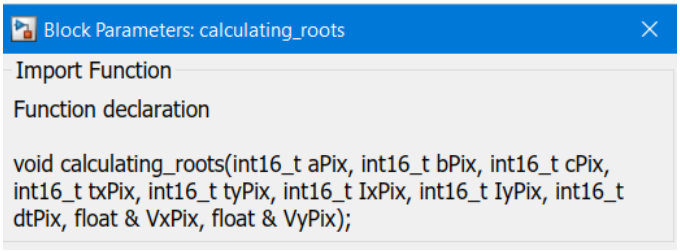
3. In the Model Composer examples dialog box select **optical flow** and click **Open example**. This opens the example design.



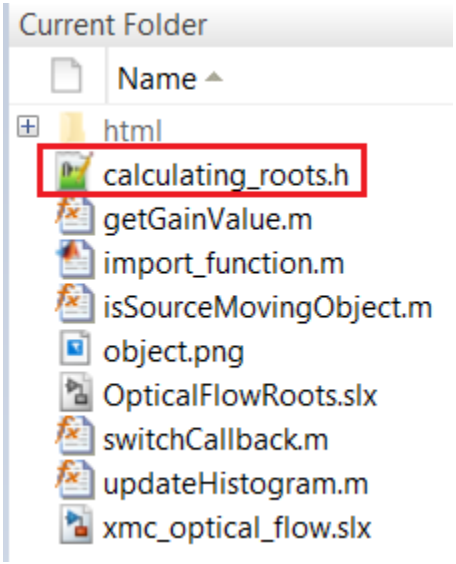
4. Double-click on the block labeled **Lucas-Kanade** and observe the `calculating_roots` block.



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



5. 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:

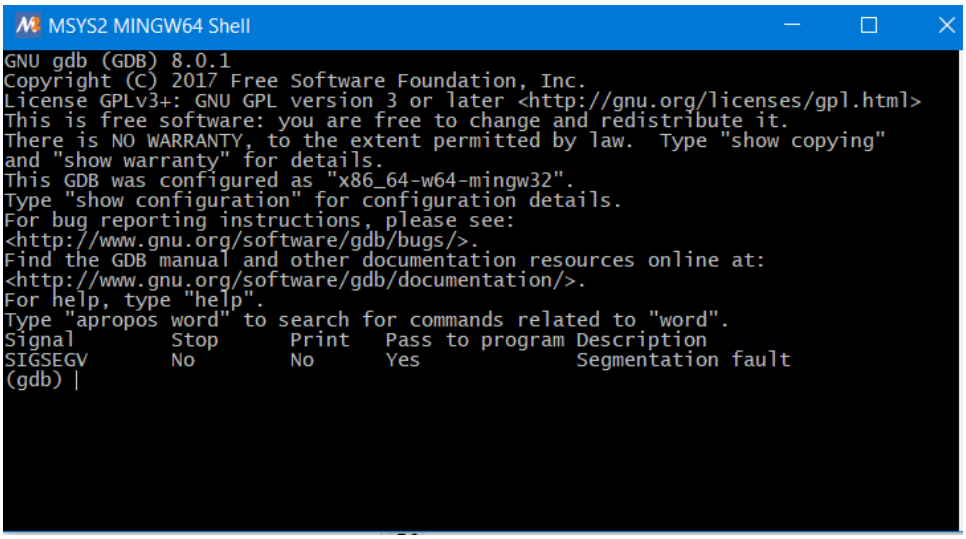
```
>> xmcImportFunctionSettings('build', 'debug');
```

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.



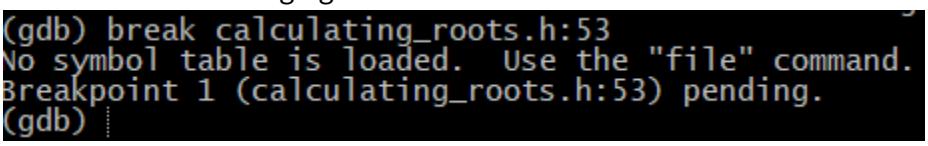
4. At the Windows command prompt, use the following command to specify the breakpoint in the `calculating_roots.h` file where you want the code to stop executing. Press **Enter** to run the command.

```
(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);
```

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



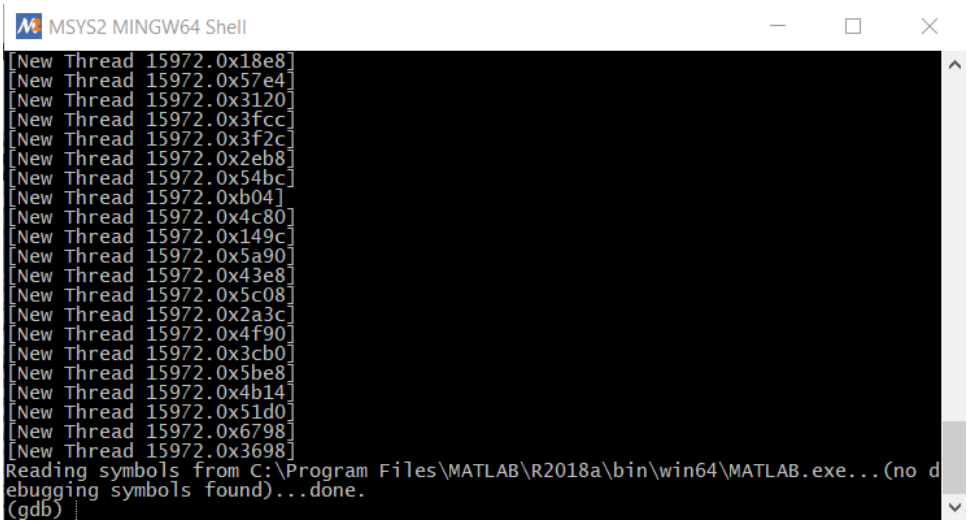
If you see any questions from GDB, answer "yes" and press **Enter**.

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

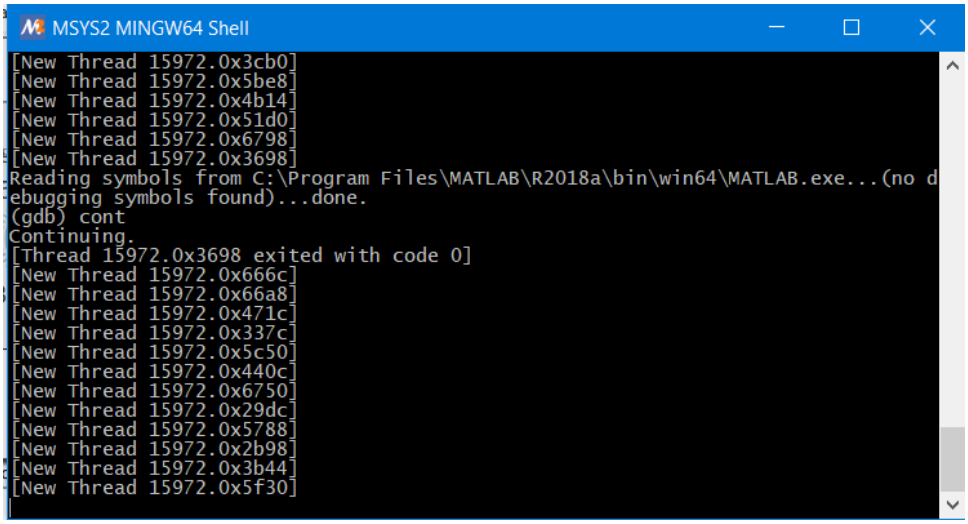
```
(gdb) attach <process_ID>
```

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

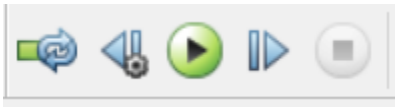
As soon as the MATLAB process is attached, the MATLAB application gets frozen and becomes unresponsive.



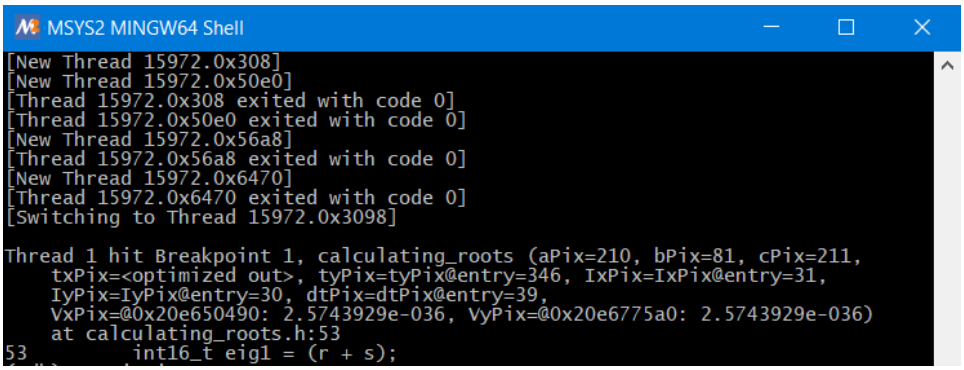
7. Type `cont` at the Windows command prompt.



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.



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

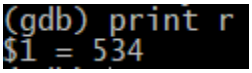
```
(gdb) list
```

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.



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.

Lab 4: 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 Vitis™ 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 HLS design.
- In Step 4, you will look at the flow for generating HLS Synthesizable `C++` code from the Model Composer HLS design.

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 `\HLS_Library\Lab4` 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.

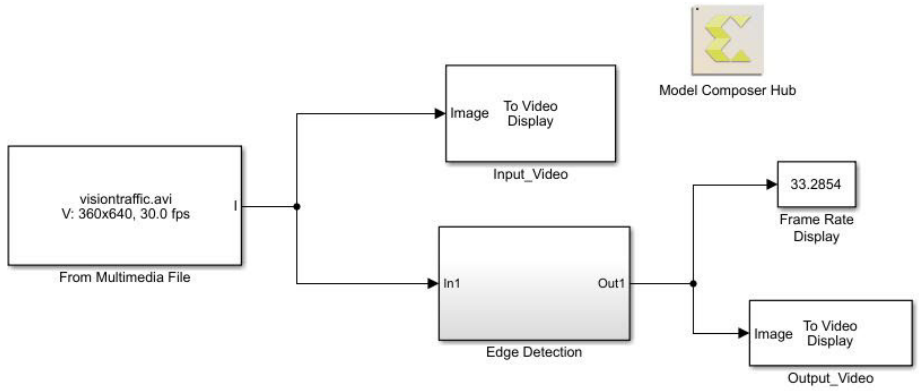
3. Right-click the **Edge Detection** area, and select **Create Subsystem from Area**.

   ***Note*:** For code generation to work, all the blocks within the enclosed subsystem should only be from the Model Composer HLS 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 HLS 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 HLS libraries as well.

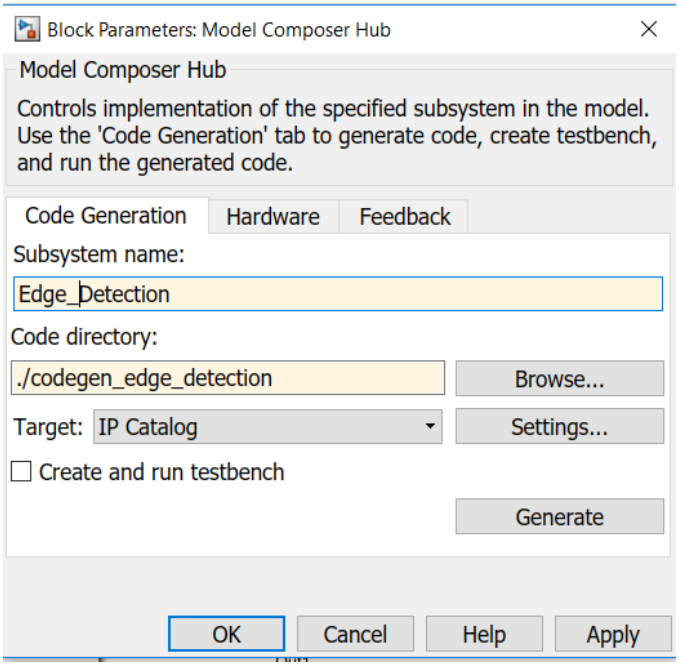
   Next, you add the Model Composer Hub block at the top level of your design.

4. Open the Simulink Library Browser and navigate to **Xilinx Tool Box → HLS → 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 Code Generation tab, you can set the following options as shown in the previous figure:
   - **Code 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.
   - **Target:** This option determines what you want to convert your design into. In this case **IP Catalog**. You can select other compilation targets from the drop-down.
     - Vitis HLS Synthesizable `C++` code.

- System Generator.

Note: The AI Engines (default) target is not applicable for the HLS block library.

8. On the Hardware tab, you can specify the target FPGA clock frequency in MHz. The default value is 200 MHz.

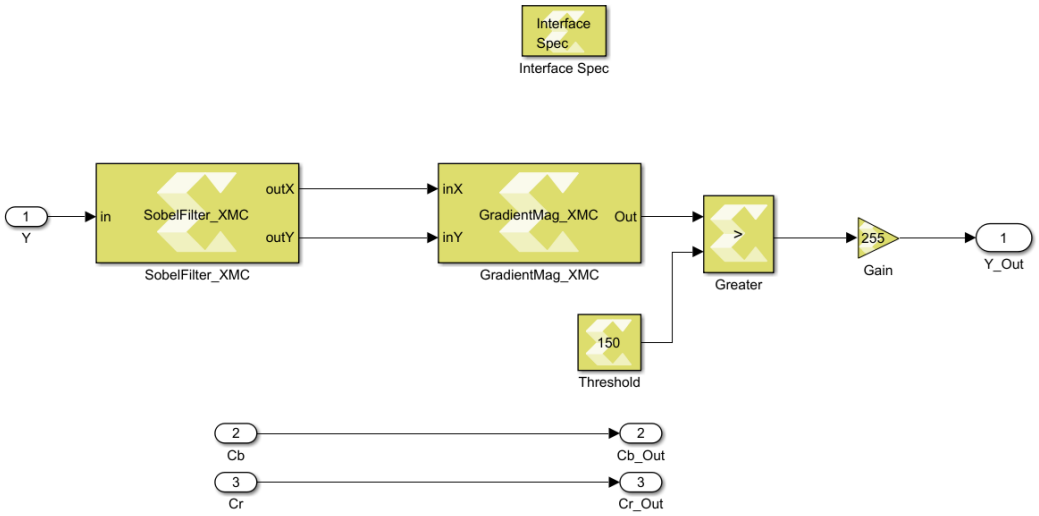
Step 2: Mapping Interfaces

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. Stop simulation before continuing to the next step.

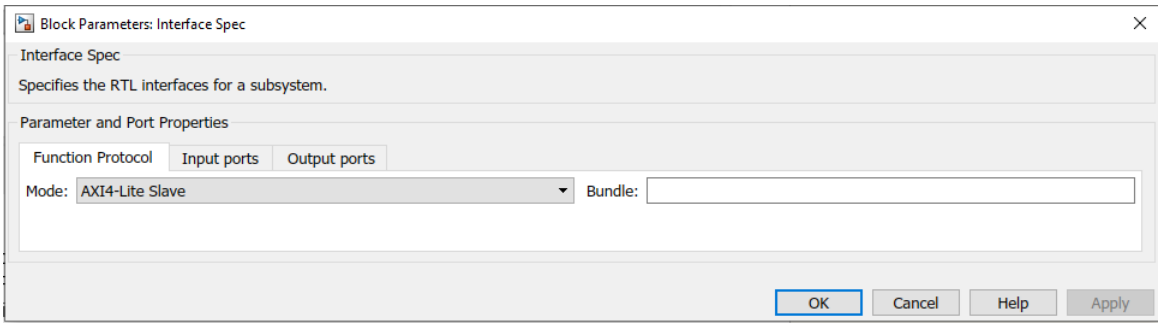
3. Open the Simulink Library browser, navigate to the **Xilinx Toolbox → HLS → Tools** sub-library and add the Interface Spec block inside the Edge Detection subsystem as shown in the following figure.



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

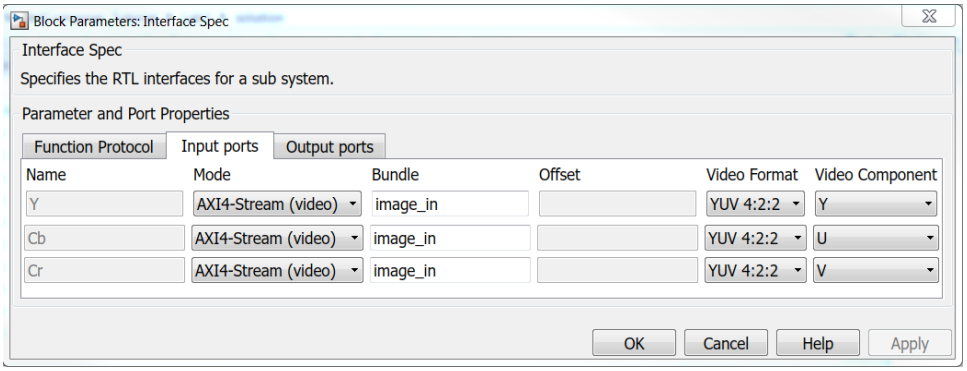
   The Interface Spec block allows you to control what RTL interfaces should be synthesized for the ports of the subsystem in which the block is instantiated. This affects only code generation; it has no effect on Simulink simulation of your design.

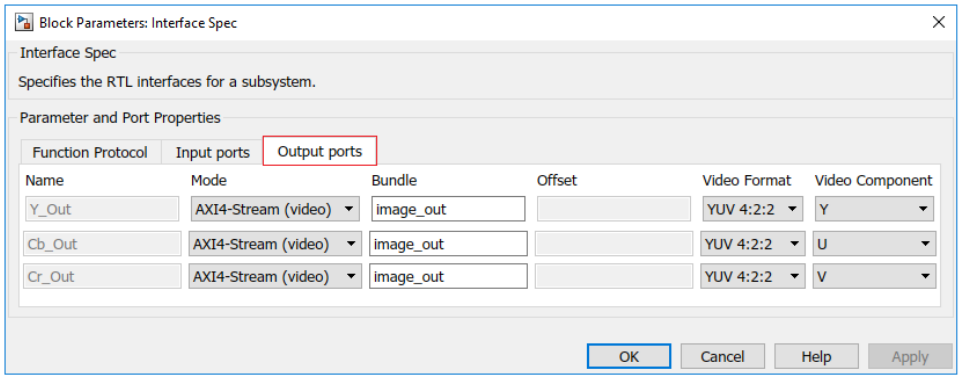
   The information gathered by the Interface Spec block consists of three parts (represented as three Tabs on the block).



- **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.

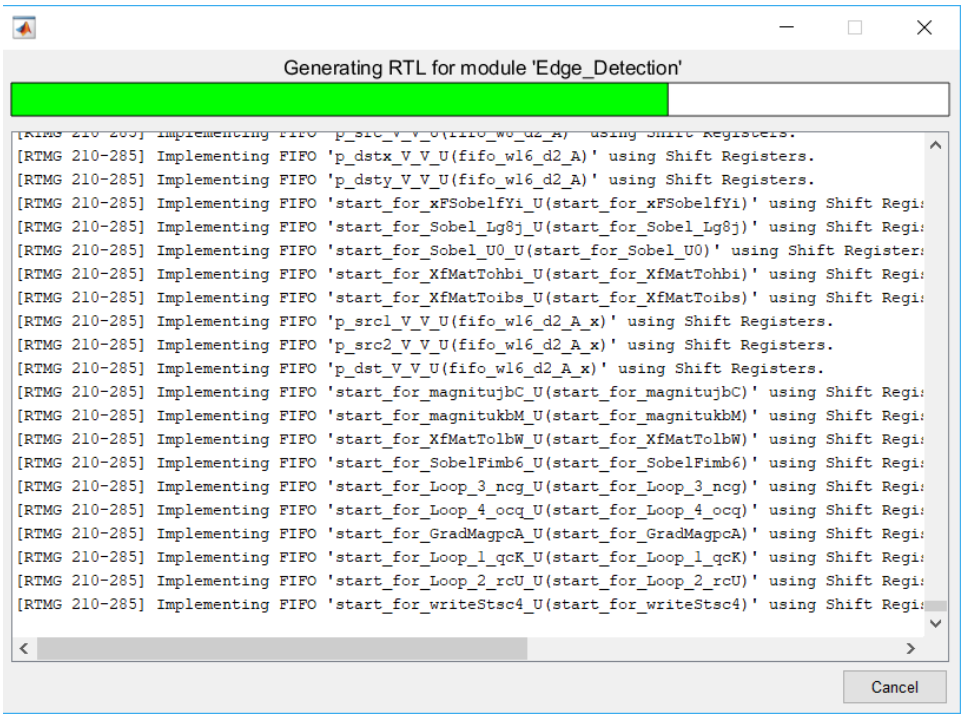
- The Video Format drop-down menu lets you select between the following formats:
  - YUV 4:2:2
  - YUV 4:4:4
  - RGB
  - Mono/Sensor
- The Video Component drop-down menu is used to subsequently select the right component: R, G, B, Y, U, V.

Step 3: Generate IP from Model Composer Design

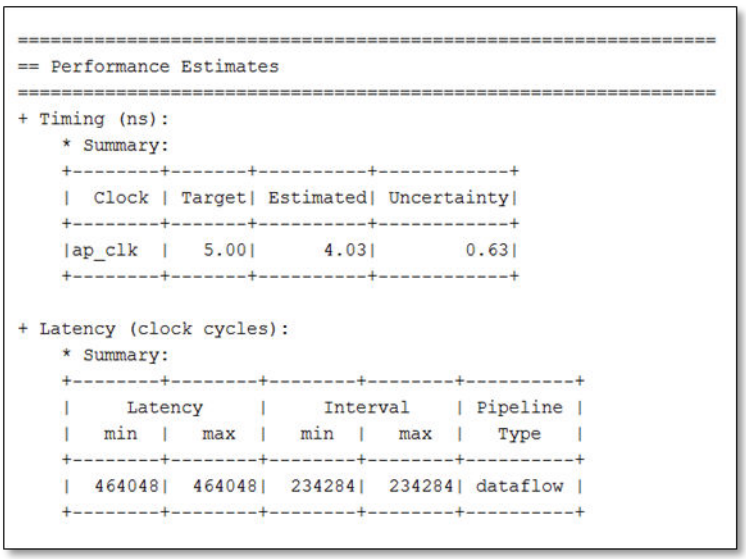
Using the same example, you will generate an IP from the Edge Detection algorithm.

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:
   - **Target:** IP Catalog
   - **Code 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 the Vitis HLS Synthesis report) in the MATLAB Editor, as shown in the following figures.



```
================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_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 Vitis HLS build you choose.

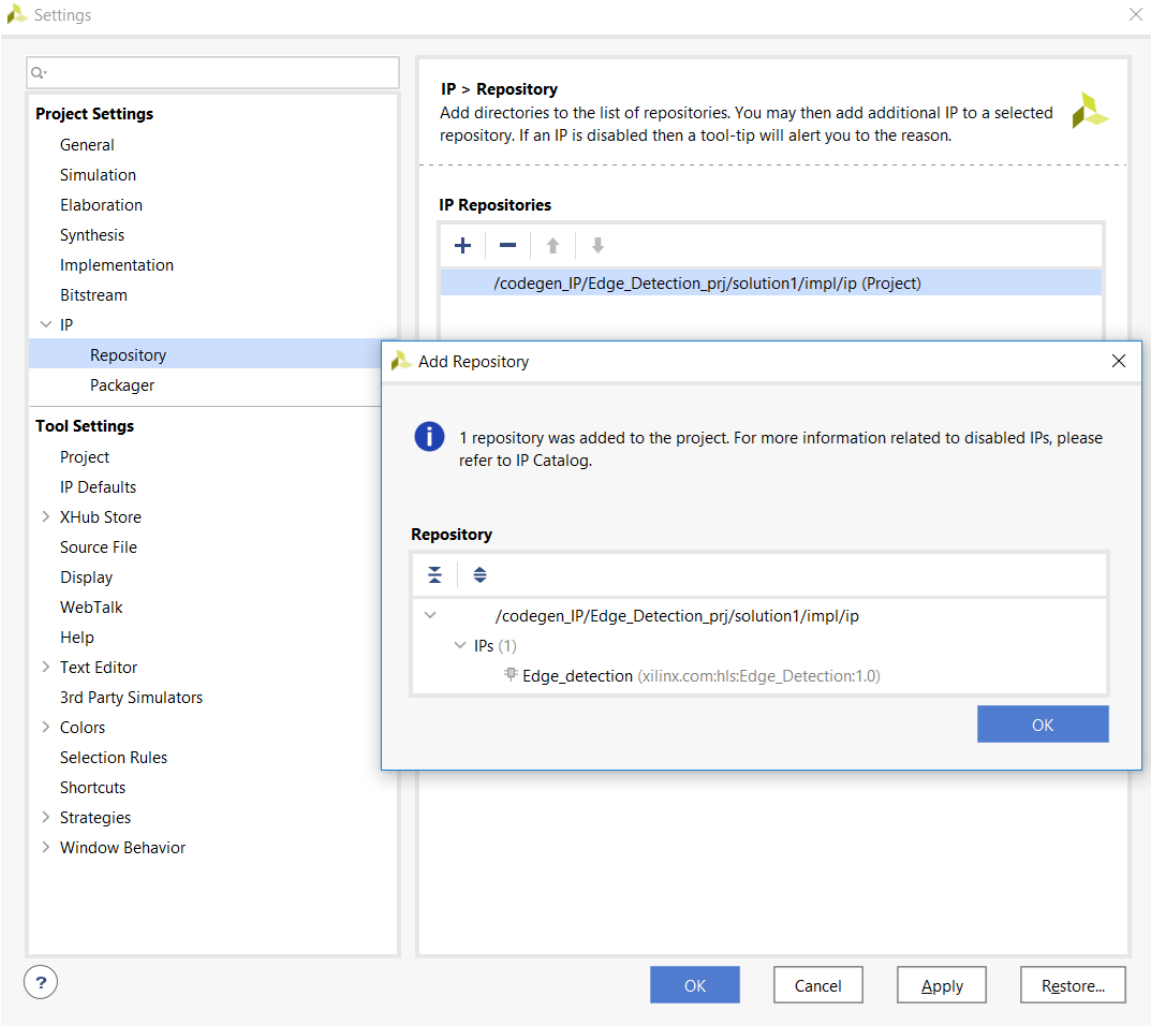
```
================================================================
== Interface
================================================================
* Summary:
+-----------------------+-----+-----+-----------+-------------------+-------------+
|       RTL Ports       | Dir | Bits|  Protocol |    Source 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 |    2|    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 |    2|    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  |
+-----------------------+-----+-----+-----------+--------------------+-------------+
```

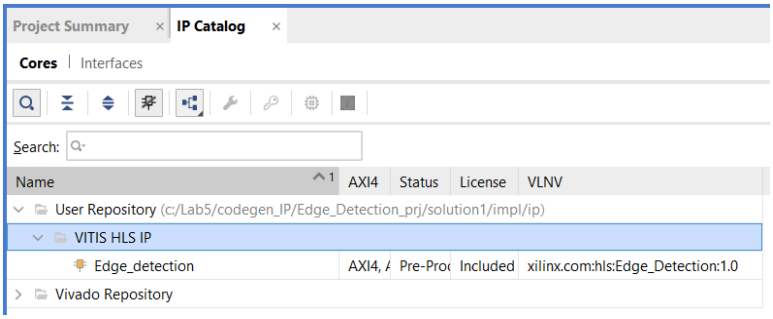
5. Launch Vivado IDE and perform the following steps to add the generated IP to the IP catalog.
6. Create a Vivado RTL project.

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).

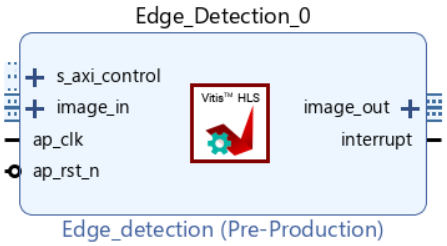
7. In the Project Manager area of the Flow Navigator pane, click **Settings**.
   a. From **Project Settings → IP → Repository**, click the **+** button and browse to `codegen_IP\Edge_Detection_prj\solution1\impl\ip`.
   b. Click **Select** and see the generated IP get added to the repository.
   c.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



8. To view the generated Edge_detection IP in the IP catalog, search for “Edge_Detection”. The generated Edge_detection IP, now appears in the IP catalog under Vitis HLS IP as shown in the following figure.



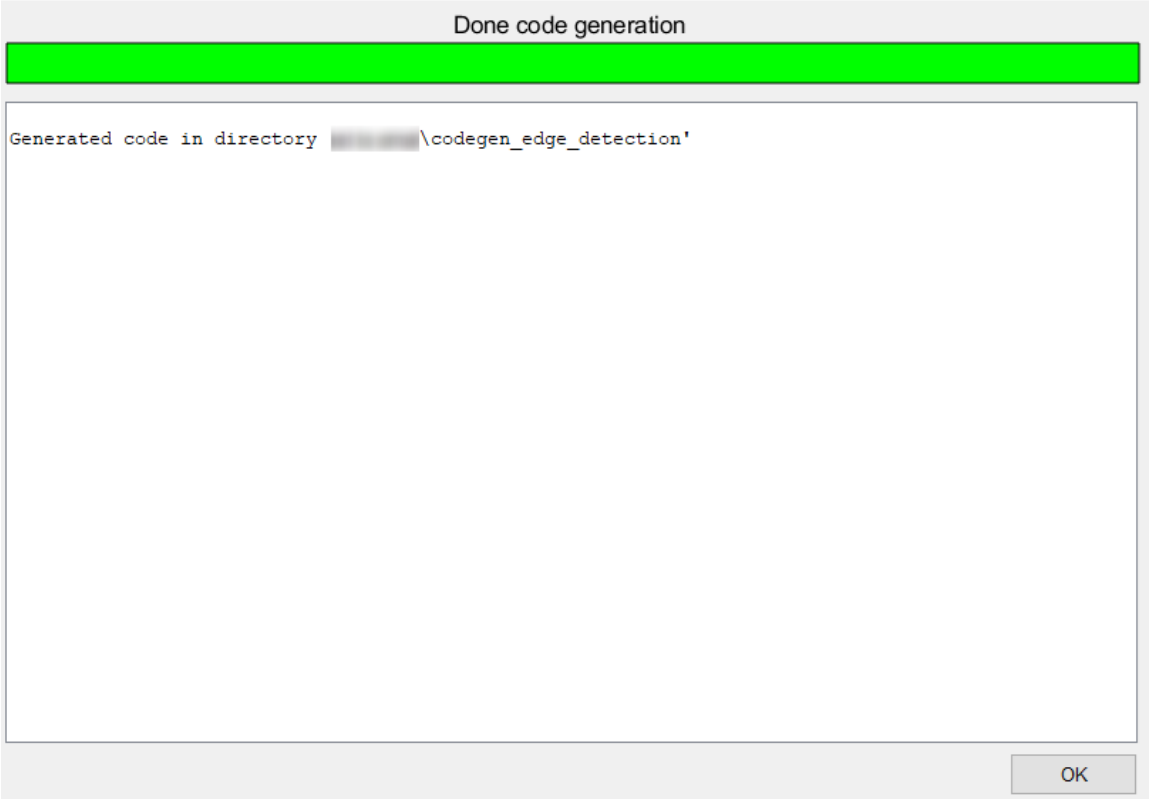
You can now add this IP into an IP integrator block diagram, as shown in the following figure.



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:
   - **Target:** HLS C++ code
   - **Code 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.

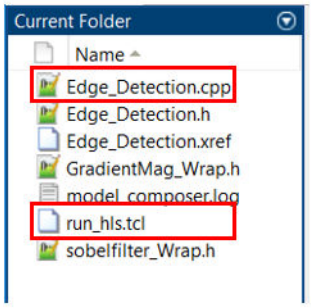


3. At the end of code generation, observe the Current Folder in MATLAB.

   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 folder that was specified in the Code directory setting. 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.

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

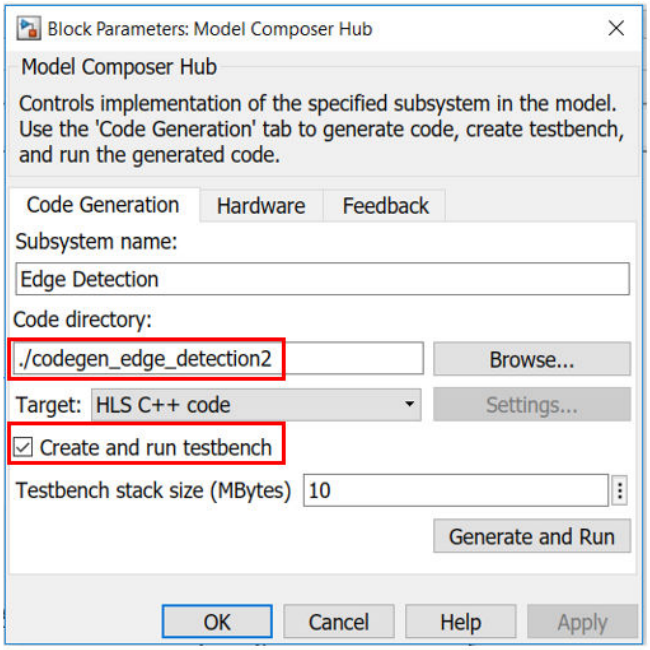


Note:

- `Edge_Detection.cpp` is the main file generated for the subsystem.

- `run_hls.tcl` is the Tcl file needed to create the Vitis HLS project and synthesize the design.

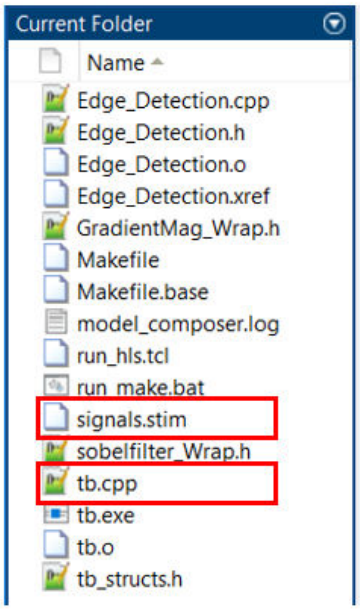
5. In the design, open the Model Composer Hub block dialog box, and modify the block settings, as shown in the following figure.
   - Check the **Create and run testbench** checkbox.
   - Modify the **Code directory** folder.



6. Click **Apply** and regenerate the code by clicking the **Generate and Run** 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 run 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 Vitis 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.

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 Vitis HLS.
- Model Composer automatically generates the test vectors and test benches for C/RTL cosimulation in Vitis HLS.
- Model Composer provides an option to export a design back into HDL model through the Vitis HLS block

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

```
\HLS_Library\Lab4\solution
```

AI Engine Library

Model Composer for AI Engine Lab Overview

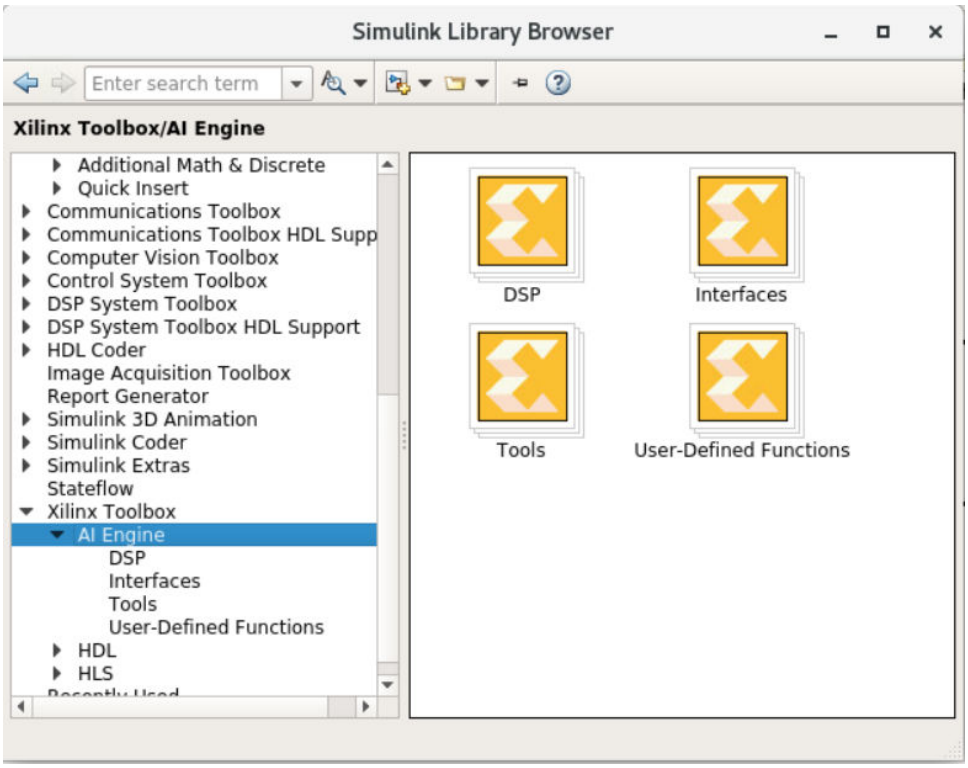
Model Composer enables the rapid simulation, exploration, and code generation of algorithms targeted for AI Engines from within the Simulink® environment. You can achieve this by importing AI Engine kernels and data-flow graphs into Model Composer as blocks and controlling the behavior of the kernels and graphs by configuring the block GUI parameters. Simulation results can be visualized by seamlessly connecting Simulink source and sink blocks with Model Composer AI Engine blocks.

Model Composer provides a set of AI Engine library blocks for use within the Simulink environment. These include:

- Blocks to import kernels and graphs which can be targeted to the AI Engine portion of Versal™ devices.
- Block to import HLS kernels which can be targeted to the PL portion of Versal devices.
- Blocks that support connection between the AI Engine and the Xilinx HDL blockset.
- Configurable AI Engine functions such as FIR, FFT, and IFFT.

IMPORTANT! *The AI Engine Lab can be done only in a Linux environment.*

Figure 2: **Simulink Library Browser: AI Engine**



This tutorial includes the following labs which introduce AI Engine support in Model Composer.

- **Lab 1:** Import an AI Engine Kernel
  - Import an AI Engine kernel using the AIE Kernel block from AI Engine library
  - Generate graph code
  - Simulate the design using the AI Engine SystemC Simulator.
- **Lab 2:** Import an AI Engine Graph
  - Import an AI Engine sub-graph using the AIE Graph block
  - Generate a top-level graph
  - Simulate the design using the AI Engine SystemC Simulator.

Lab 1: Importing AI Engine Kernels

This section of tutorial shows how to import AI Engine kernels into Model Composer, generate the code, and simulate using AIE Simulation.

Procedure

This lab has the following steps:

- In Step 1, you build your design with three AI Engine kernels in the Model Composer..
- In Step 2, you simulate the design.
- In Step 3, you generate a graph code and simulate using AIE simulation.

Step 1: Build the AI Engine Design in Model Composer

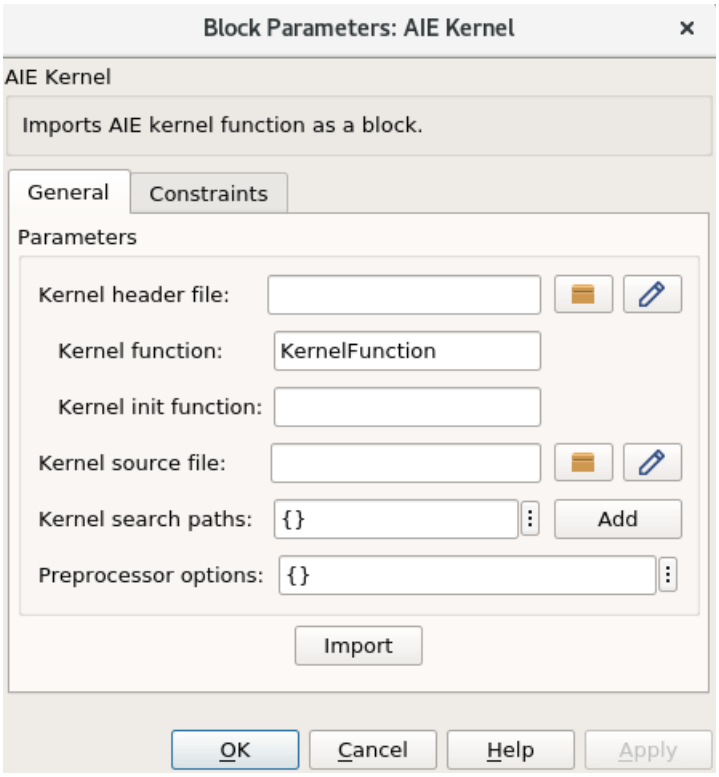
In this step, you will import three kernel functions using the AIE Kernel block available in the Model Composer AI Engine library and build a design.

1. In the MATLAB Current Folder, navigate to `AIEngine_Library/Lab1/`.
2. Use the subsequent steps to import the kernel function `fir_27t_sym_hb_2i` into the design. This is an interpolating-by-two filter fir symmetric filter. Because it is interpolating, the output of the filter is twice the size of the input. Open the source code `hb_27_2i.cpp` from `kernels/src/hb_27_2i.cpp`, and notice this kernel has a Window input and a Window output.

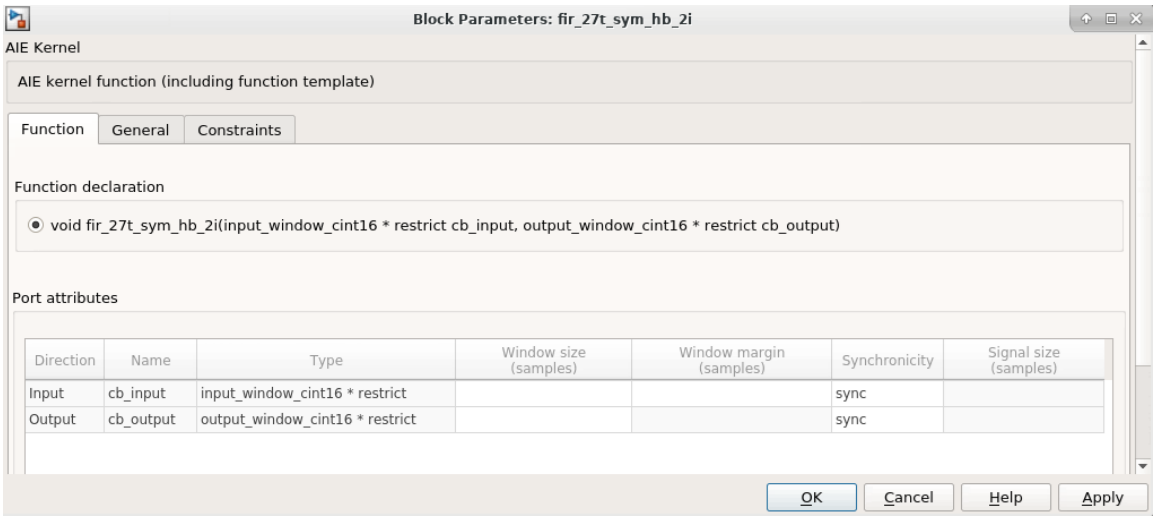
```
void fir_27t_sym_hb_2i
(
    input_window_cint16 * cb_input,
    output_window_cint16 * cb_output)
{
```

3. Double-click `import_kernel.slx` to open the model. This is a model with only sources and sinks. You will fill the design in-between.
4. From the Library Browser, select the **AIE Kernel** block from under the User-Defined functions of the AI Engine library. Drag the block into the `import_kernel.slx` file.

5. Double-click the block. The following Block Parameters dialog box displays.



6. Update the block parameters as follows:
   - **Kernel header file:** Either browse to locate the `hb_27_2i.h` file or enter `kernels/inc/hb_27_2i.h` as the parameter.
   - **Kernel function:** `fir_27t_sym_hb_2i`
   - **Kernel init function:** Leave empty
   - **Kernel source file:** Either browse to locate `hb_27_2i.cpp` file or enter `kernels/src/hb_27_2i.cpp` as the parameter.
   - **Kernel search path:** Leave empty
   - **Preprocessor options:** Leave empty
7. Click **Import**. The tool parses the function signature in the header file and updates the AIE Kernel block GUI interface. The Function tab is displayed as shown in the following figure.



8. Update the parameter values as indicated in the following figure.

IMPORTANT! *All parameters are in samples (not bytes).*

IMPORTANT! *The tool does not parse the kernel function and it does not have any knowledge about the input or output window sizes, nor the input window margin size.*

9. After applying, click **OK** to close the window and connect the block to the input as shown in the following figure.

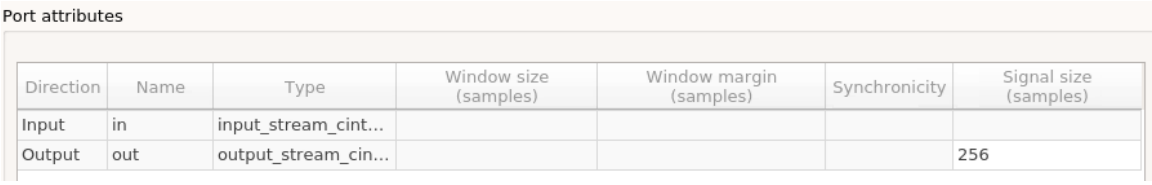
10. Next import the `polar_clip` function. Unlike the previous kernel, the polar clip has a stream in port and a stream out port. The function signature is as follows.

```
void polar_clip(input_stream_cint16 * in, output_stream_cint16 * out)
```

11. Drag the new AIE Kernel block from AI Engine library and update the parameters as follows:
    - **Kernel header file:** `kernels/inc/polar_clip.h`
    - **Kernel function:** `polar_clip`

- **Kernel init function:** Leave empty
- **Kernel source file:** `kernels/src/polar_clip.cpp`
- **Kernel search paths:** Leave empty
- **Preprocessor options:** Leave empty

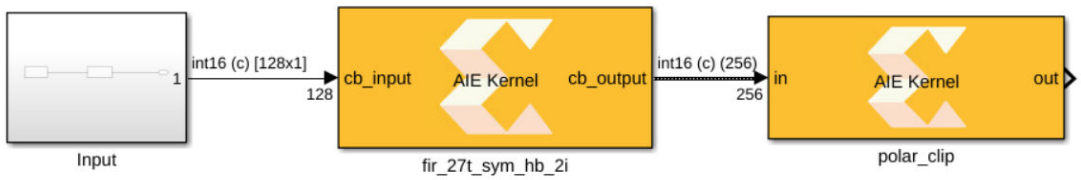
12. Click **Import**. The tool parses the header file and creates the block. Update the parameter value as shown in the followinf figure.



> **IMPORTANT!** *Here, the Window size and Window margin fields are not applicable for signals of type stream.*

> **IMPORTANT!** *The Signal size parameter is the maximum size of the output - variable size signal.*

13. Connect this block to the existing design and it should now look as follows.



You have connected a block with a window output to a block with a stream input.

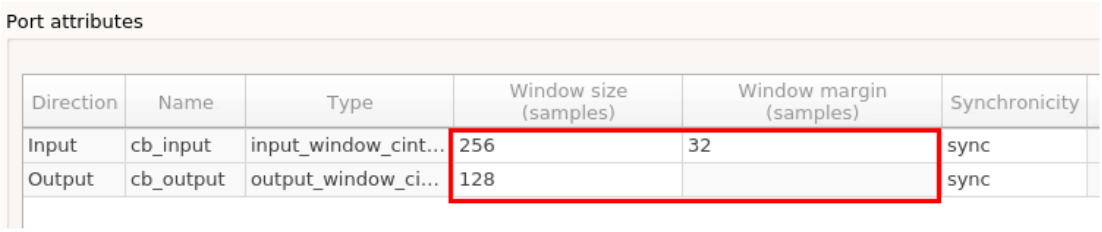
14. Use the subsequent steps to import the final kernel function `fir_27t_symm_hb_dec2` into the design. This is a decimation by two filter and the signature to this function is as follows.

```
void fir_27taps_symm_hb_dec2
(
    input_window_cint16 * inputw,
    output_window_cint16 * outputw
){
```

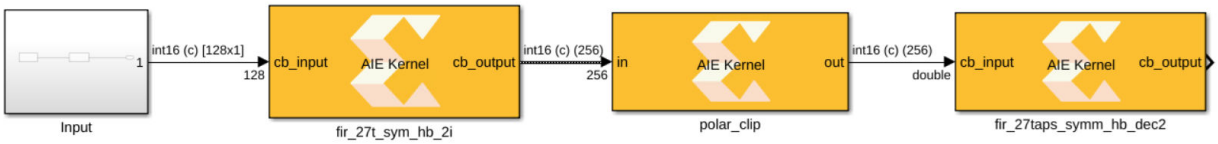
15. Drag the new AIE Kernel block from the AI Engine library and update the parameters as follows:

- **Kernel header file:** `kernels/inc/hb_27_2d.h`
- **Kernel function:** `fir_27taps_symm_hb_dec2`
- **Kernel init function:** Leave empty
- **Kernel source file:** `kernels/src/hb_27_2d.cpp`
- **Kernel search paths:** Leave empty
- **Preprocessor options:** Leave empty

16. After applying, click **OK** to close the window. A new Function tab opens. Set the parameters for this kernel as follows.

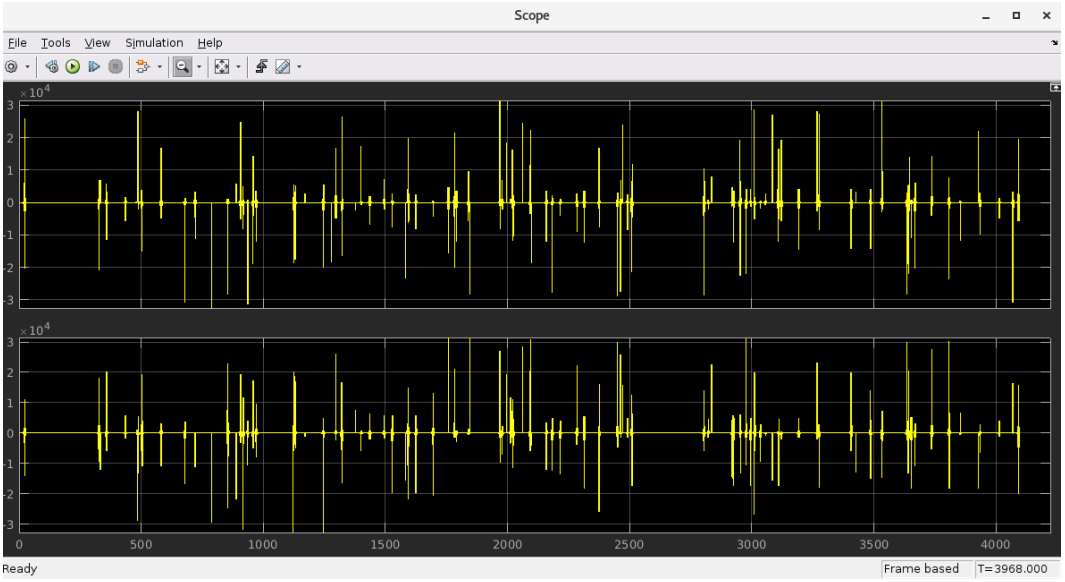


17. Connect the block to the existing design as follows.



Step 2: Simulate the Design

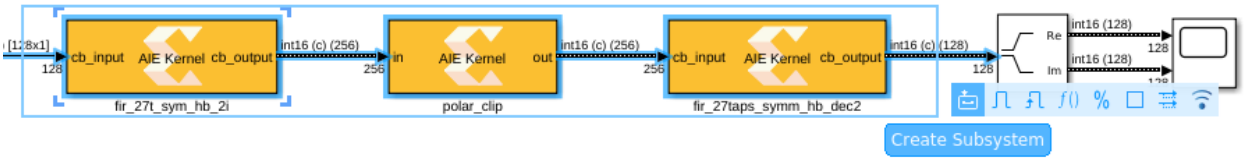
1. As with any other Simulink design, simulate the design. Notice that the first time you simulate, it takes some time (less than a minute) before the simulation starts. During this time, the code for each kernel is getting compiled and executable files are getting created.
2. After compilation, you should get the real and imaginary outputs in scope as shown in the following figure.



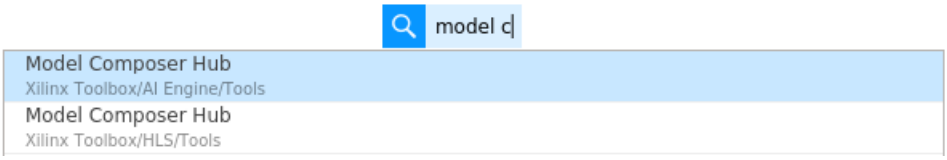
Step 3: Code Generation and Running AI Engine SystemC Simulation

Model Composer can generate graph code from your design. It also generates a make file and collects data from the input and output port of your system. In this step you will see how this is done.

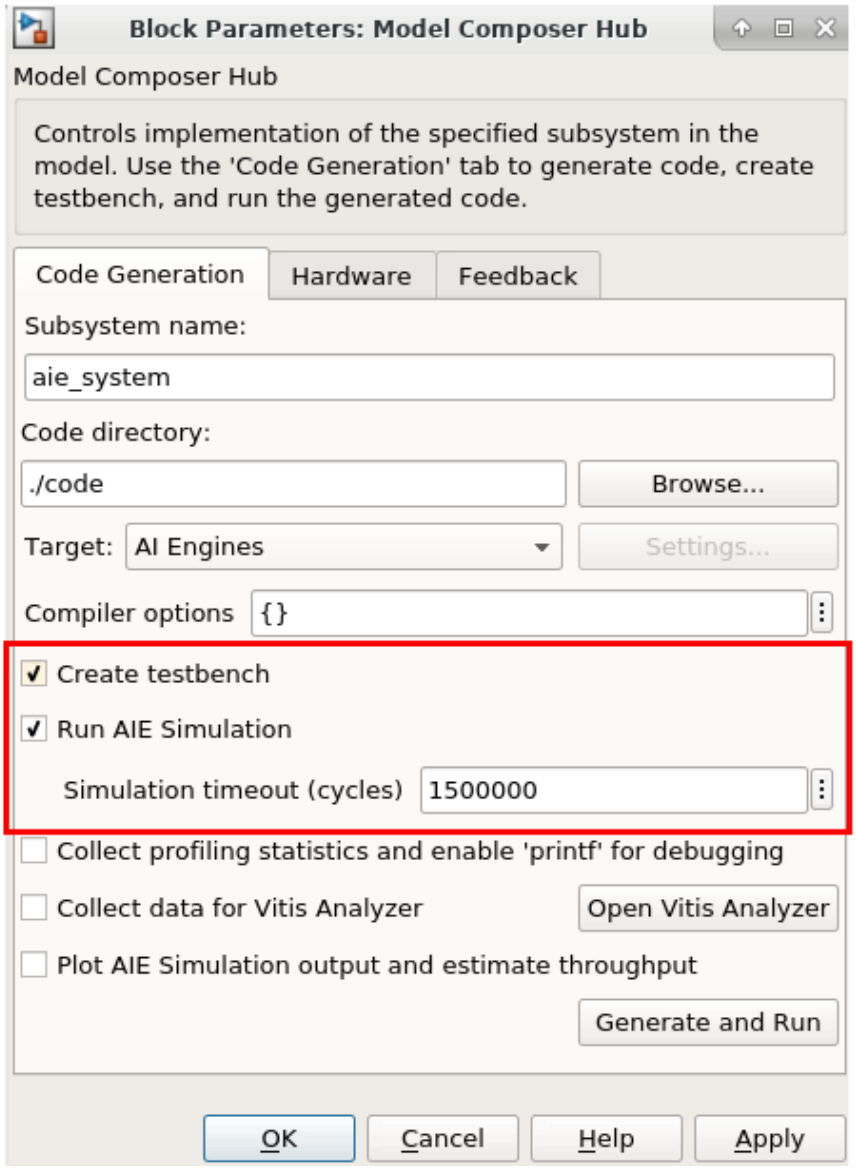
1. Create a subsystem from all three blocks in your design. You can do this by selecting the blocks and clicking the **Create Subsystem** button as shown in the following figure.



2. Assign a name to the subsystem, for example `aie_system`.
3. Drag the Model Composer Hub from the library browser or simply click on the canvas and start typing `Model Composer Hub`.



4. Double-click the Model Composer Hub and make changes as follows.

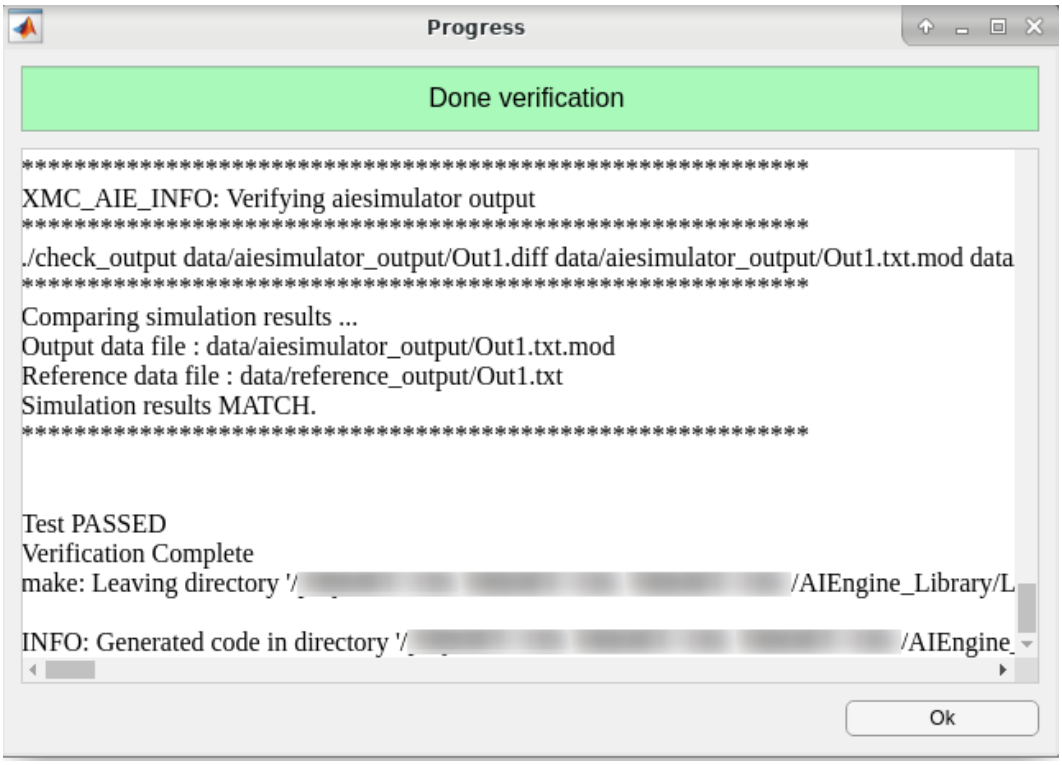


When you check **Create testbench**, the tool generates a testbench, including input and output test vectors from Model Composer. You can use AI Engine SystemC Simulator to verify the correctness of the design by comparing the results with the test vectors from Model Composer.

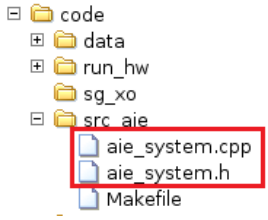
***Note*:** The AIE simulation may take some time to complete.

5. Click **Apply** and then **Generate and Run**. Within a few seconds the code directory gets created. Because you also checked **Create testbench**, the aiecompiler will get invoked under the hood and compile the code using the generated graph code and the kernel source codes. It subsequently runs the AIE Simulation.

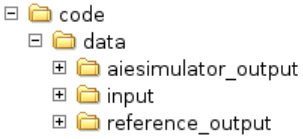
This operation will take a couple of minutes. Observe the simulation completion message along with the comparison of the output to the Simulink output (`data/reference_output/Out1.txt`). It prints any diff, in the wait dialog.



6. Click **OK** to exit the Progress window.
7. Navigate to `code/src_aie` to inspect the generated graph code and Makefile. The tool automates the generation of all these files.



8. Navigate to the data folder and observe the `reference_output/` directory where the data logged from Simulink gets stored. The `aiesimulator_output/` folder logs the data from AI Engine SystemC simulator and Model Composer compares the results at the end of simulation.



Conclusion

In this lab, you learned:

- How to import AI Engine kernels into Model Composer and build a design.
- How to generate the graph code using the Model Composer Hub block.
- How to perform AI Engine SystemC simulation.

The following solution directory contains the final Model Composer files for this lab.

- `AIEngine_Library/Lab1/Solution`

Lab 2: Importing AI Engine Graphs

This section of the tutorial shows how to import AI Engine graphs into Model Composer, generate the code, and simulate using AI Engine SystemC simulation.

Procedure

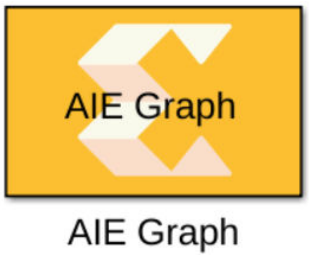
This lab has following steps:

- In Step 1, you build your design by importing AI Engine Graph code in the Model Composer.
- In Step 2, you simulate the design.
- In Step 3, you generate a graph code and simulate using AI Engine SystemC simulation.

Step 1: Build an AI Engine Design using Graph Code

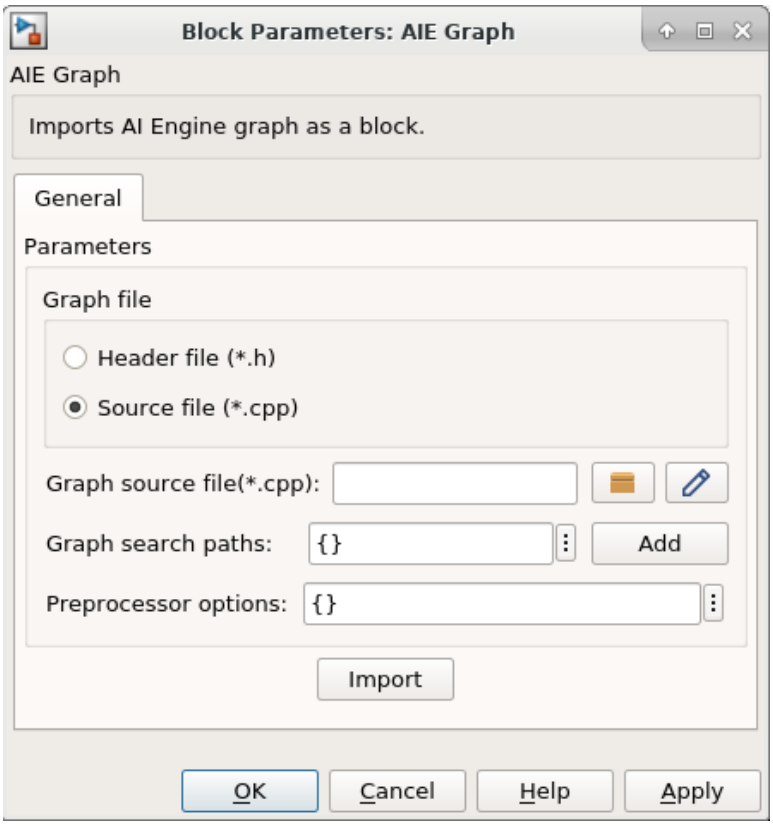
In this step you will import graph code (generated using the design in Lab 1) using the AIE Graph block available in the Model Composer AI Engine library and build a design.

1. In the MATLAB® Current Folder, navigate to `AIEngine_Library/Lab2/`
2. Double-click `import_graph.slx` to open the model. This is a model with a source and a sink and you will fill the design in-between.
3. From the Library Browser, select the AIE Graph block from the AI Engine library. Drag the block into the `import_graph.slx` file.



You can also click on the canvas and start typing `AIE Graph`.

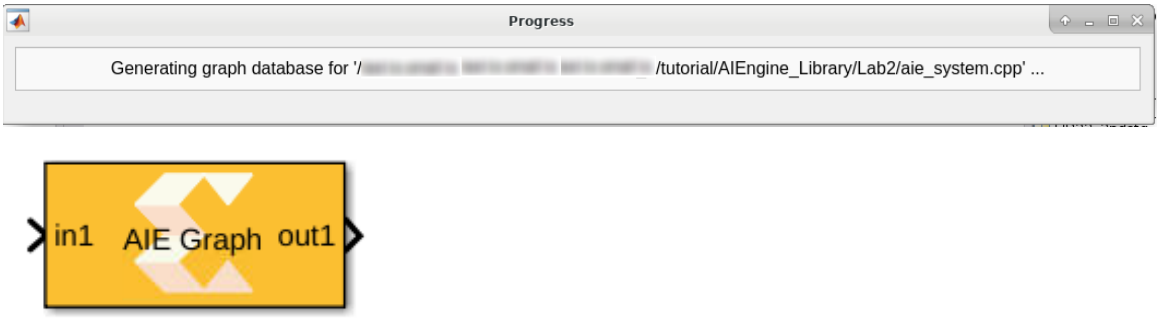
4. Double-click the block and select **Source file (*.cpp)** from the Graph file parameter as shown.



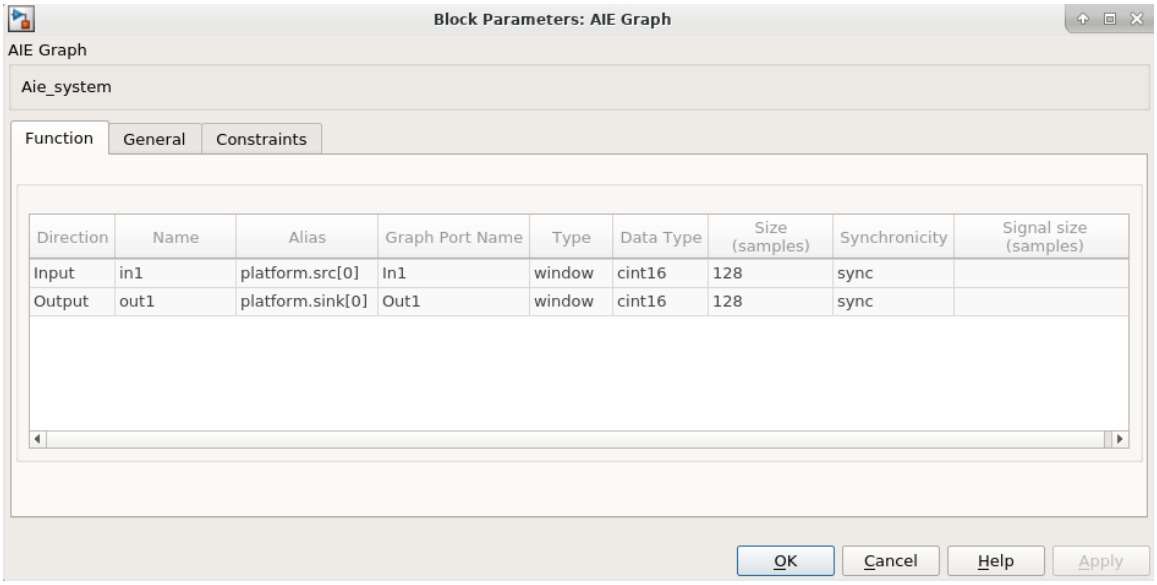
Note: Here, the `*.cpp` flow is used to import the graph. Alternatively, you can use the `*.h` flow (in which case the following steps will differ slightly).

5. Update the block parameters as follows:
   - **Graph source file(*.cpp):** `aie_system.cpp`
   - **Graph search paths:** Either browse to locate the kernels or enter `{'./kernels/src', './kernels/inc', './include'}` as the parameter.
   - **Preprocessor options:** Leave empty

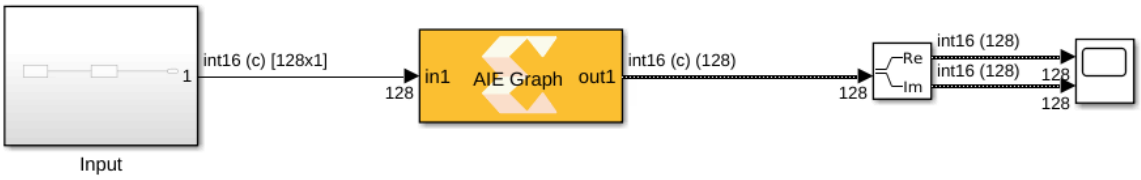
6. Click **Import**. You will see the Progress window as shown in the following figure. Once done, (after about 15 seconds), the AIE Graph block GUI interface will get updated.



7. Observe the Function tab in the AIE graph block parameters as shown.

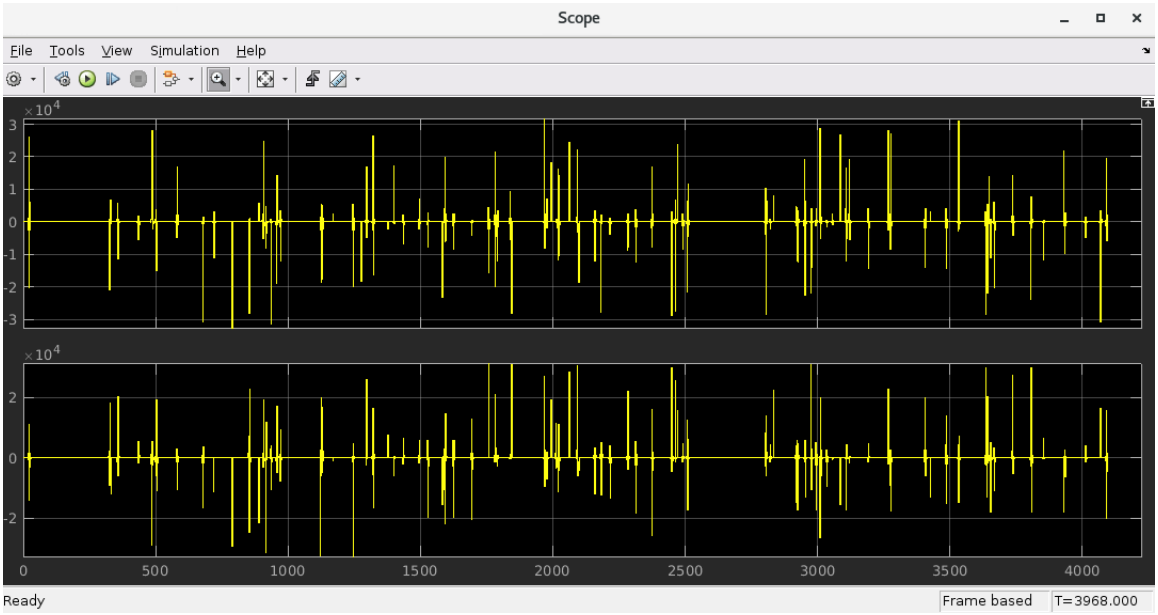


8. Click **Apply** and connect the AIE Graph block as shown in the following figure.



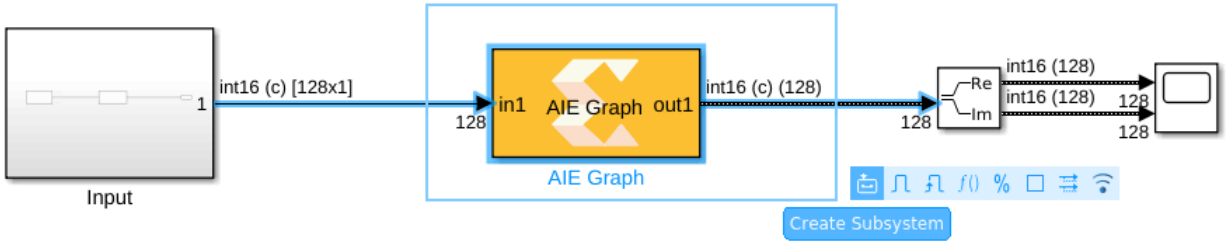
Step 2: Simulate the Design

1. Click **Simulate**. You will get similar results as those in Lab 1 (Import AIE Kernel).



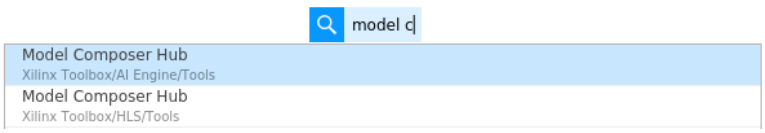
Step 3: Code Generation and AI Engine SystemC Simulation

1. Create a subsystem for the graph block.

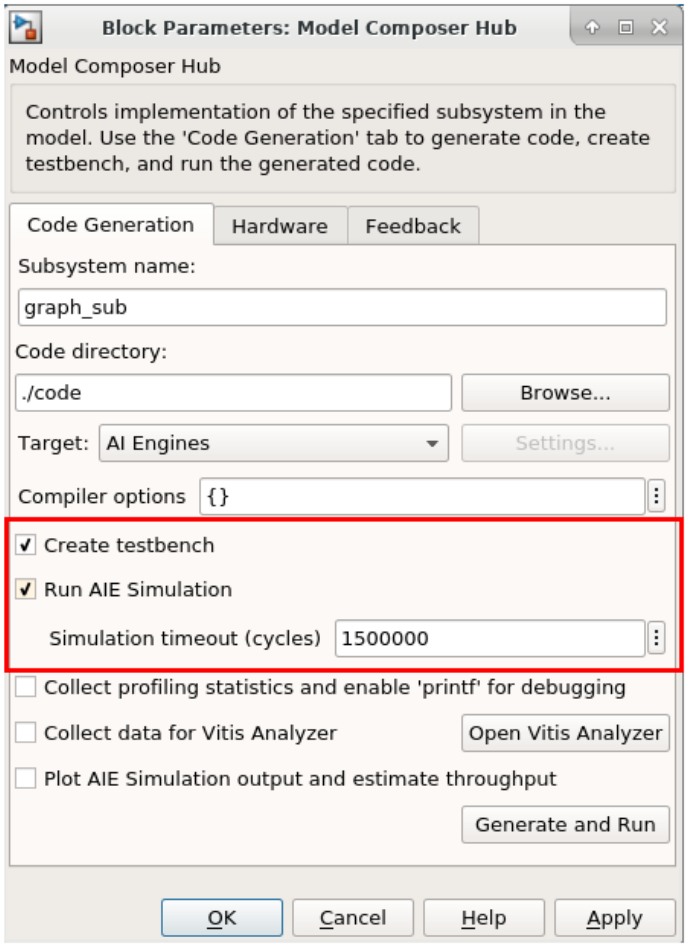


Note: In this particular design scenario only one graph code is imported. But in a case where we have multiple graphs imported and connected, Model Composer automatically generates the top module which includes interconnections of all blocks.

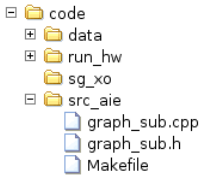
2. Assign the subsystem name as `graph_sub`.
3. Drag the Model Composer Hub block from the library browser or simply click on the canvas and start typing `Model Composer Hub`.



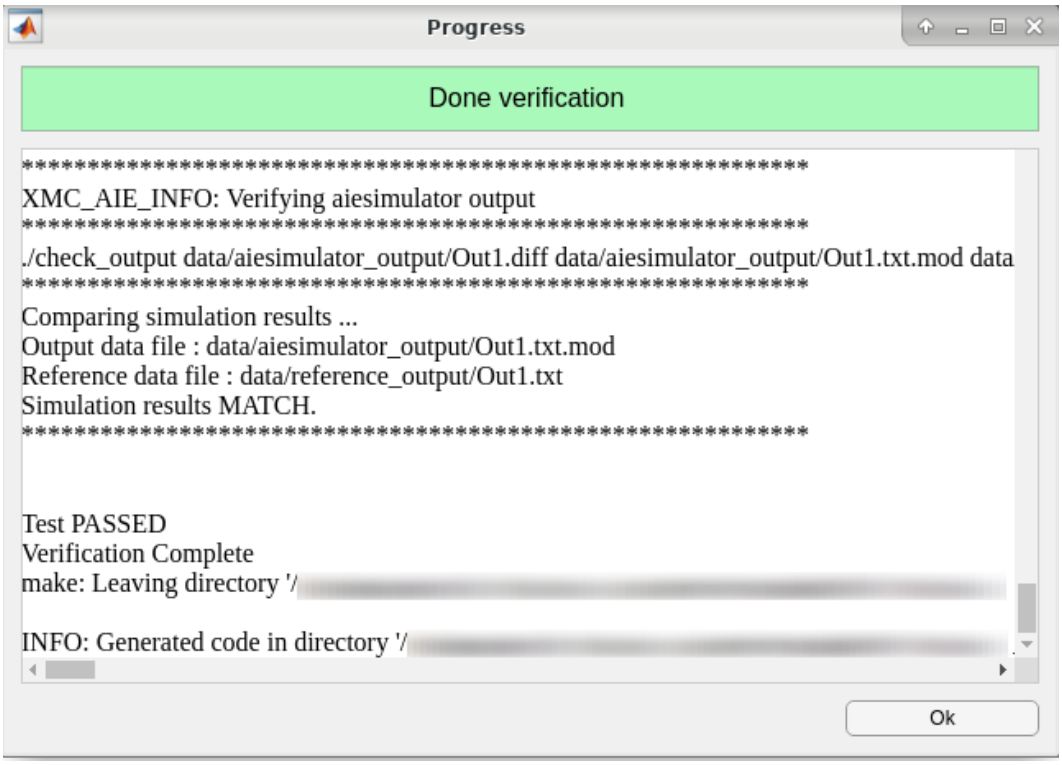
4. Double-click the Model Composer Hub block and make changes as follows (Similar to those in Lab 1 - Import AIE kernel).



5. The Simulation procedure is similar to that of Lab 1 (Import AIE Kernel). It also generates the Target directory (`./code` in this case) under which you can see the top level graph code under `code/src_aie` directory and the `code/data` directory which contains the data logged from the Simulink design along with output from the AIE simulation.



6. Notice the log in the Progress window after completion. Click **OK**.



Conclusion

In this lab, you learned:

- How to import AI Engine graph code into Model Composer.
- How to generate the top level graph code using the Model Composer Hub block.
- How to perform the AI Engine SystemC simulation.

The following solution directory contains the final Model Composer files for his lab.

- `AIEngine_Library/Lab2/Solution`

XILINX

Appendix A

Additional Resources and Legal Notices

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. *Vitis Model Composer User Guide* (UG1483)

Please Read: Important Legal Notices

Copyright