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

Revision History	2
Locating the Tutorial Design Files	3
Chapter 1: HDL Library	5
Lab 1: Introduction to Model Composer HDL Library	5
Lab 2: Importing Code into Model Composer HDL Design	
Lab 3: Timing and Resource Analysis	57
Lab 4: Working with Multi-Rate Systems	
Lab 5: Using AXI Interfaces and IP Integrator	81
Lab 6: Using a Model Composer HDL Design with a Zynq-7000 SoC	91
Chapter 2: HLS Library	102
Lab 1: Introduction to Model Composer HLS Library	
Lab 2: Importing Code into Model Composer	112
Lab 3: Debugging Imported C/C++-Code Using GDB Debugger	
Lab 4: Automatic Code Generation	129
Chapter 3: AI Engine Library	142
Model Composer for AI Engine Lab Overview	142
Lab 1: Importing AI Engine Kernels	143
Lab 2: Importing AI Engine Graphs	152
Appendix A: Additional Resources and Legal Notices	158
Xilinx Resources	158
Documentation Navigator and Design Hubs	158
References	158
Please Read: Important Legal Notices	

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 <code>open Lab1_1.slx</code> 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.

File Edit	View D	isplay	Diagram	Simula	ation Anal	ysis Code	Tools
•	<p td="" ⇒<=""><td></td><td></td><td></td><td></td><td></td><td></td></p>						
Model Brows	er	*	Introducti	on_Step1	× Lov,	Run ilter >	<

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 \rightarrow 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 \rightarrow HDL \rightarrow 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.



🛛 🖬 Simulink Library Browser			_	×
💠 💠 frame rate 🗸 🔌	- 🛃 - 1	ॼ ▾ ➾ ②		
Xilinx Toolbox/HDL/Interfaces				
 Computer Vision Toolbox Control System Toolbox DSP System Toolbox HDL Support Embedded Coder Fixed-Point Designer HDL Coder HDL Verifier Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Extras Stateflow Xilinx Toolbox HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	~	Gateway In Gateway Out		

- 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.





Block Parameters: Digital Filter Desi	gn 🗖 🗖 💌
File Edit Analysis Targets View	Window Help
🗅 🚅 🖶 🚑 💽 🔍 (연. 1년 🖸	1 🖬 💽 💀 😹 🗄 🖵 🌐 🗑 🔂 🗹 🕺
Current Filter Information	Magnitude Response (dB)
Structure: Direct-Form FIR Order: 10 Stable: Yes Source: Designed	0 (B) -20 e) -40 -40 -40 -40 -40 -40 -40 -40
Filter Manager	
Response Type	Filter Order Frequency Specifications Magnitude Specifications Specify order: 10 Units: MHz V Units: dB
O Highpass	Specify order: 10 Units: MHz VIIIts: dB
Bandpass	Minimum order Fs: 20 Apass: 0.01
◎ Bandstop	- Options
Differentiator	Density Factor: 16 Fstop: 8.5
🔁 — Design Method —	
IIR Butterworth	
FIR Equiripple	
Input processing: Colum	Ins as channels (frame based)
Ready	

- 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.

🔀 Digital FIR Filter (Xilinx FIR Block)
Filter Parameters Coefficient Vector Use FDA Tool as Coefficient source
xlfda_numerator('FDATool') FDA Tool
Coefficient Precision Optimal values Coefficient Width : 19 Coefficient Fractional Bits : 19
Interpolation Rate 1
Decimation Rate 1
OK Cancel Help Apply

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).

🔀 Gateway In (Xilinx Gateway In)							
Gateway in block. Converts inputs of type Simulink integer, single, double and fixed-point to Xilinx fixed-point or floating-point data type.							
Hardware notes: In hardware these blocks become top level input ports.							
Basic Implementation							
Output Type							
🔘 Boolean 💿 Fixed-point 🔘 Floating-point							
Arithmetic type Signed (2's comp) -							
Fixed-point Precision							
Number of bits 16 Binary point 14							
Exponent width 8 Fraction width 24 Quantization: Truncate Round (unbiased: +/- Inf) Overflow:							
Wrap Saturate Flag as error							
Sample period 1/20e6							
OK Cancel Help Apply							

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.

💽 System Ge	enerator: Lab1_	1			
	Clasking	Canard			
Compilation	Clocking	General			
Enable m	ultiple clocks				
FPGA cloc	ck period (ns)	:	Clock pin loca	ation :	
50					
Provide cl	ock enable clear	pin			
_	system period				
1/20e6					
Perform a	nalysis :		Analyzer type	:	
Post Synthes	sis	•	Resource	•	Launch
Performance	e Tips Gener	ate OK	Apply	Cancel	Help

- 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.





承 System Ge	enerator: Lab1_	1				
101300 0001						
Compilation	Clocking	General				
Board :						
> None						
Part :						
> Kintex7 :	xc7k325t-3fbg676	5				
Compilatio	n:					
> IP Catalo	g					Settings
Hardware	description la	anguage		VHDL librar	y:	
VHDL			•	xil_defaultlib]
Use STD_	LOGIC type for B	oolean or 1	bit wide g	gateways		
Target dire	ctory :					
./netlist						Browse
Synthesis	strategy :		Impler	nentation str	ategy :	
Vivado Synth	esis Defaults	•	Vivado I	mplementation [Defaults 🔹 🔻]
Create inte	erface document		Crea	te testbench	Mo	del upgrade
Performance	Tips Gener	ate	ОК	Apply	Cancel	Help

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.

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Lab1_1_sol	0	6	281	40
Digital FIR Filter	0	6	281	40

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.

Send Feedback



- 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).

承 System Ge	enerator: Lab1_	2			- • ×	
Compilation	Clocking	General				
Enable m	ultiple clocks					
FPGA cloc	ck period (ns)	:	Clock pin lo	cation :		
50						
	lock enable clear system perioc nalysis :		Analyzer ty	pe:		
None		•	Timing		▼ Launch	
Performance	e Tips Gener	rate OK	Apply	Cance	el Help	

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 **I** to view the filter coefficients.



Block Parameters: FDATool			_ • •
File Edit Analysis Targets View	Window Help		
다 🚅 🖬 🖨 🖻, 🔍 🔍 🗟 🛅 🛅	, 📐 😡 🔂 🍀 🙁 🍸 🖵 🌐 😡)1 🖸 🖳 🕅	
Current Filter Information	Filter Coefficients	ilter Coefficients	
Structure: Direct-Form FIR Order: 10 Stable: Yes Source: Designed	Numerator: 0.0019067134188906 -0.0110752394328747 -0.0411515914481301 0.0351305675326196 0.2887827846112869 0.4509324797603543 0.2887827846112869 0.0351305675326196 -0.0411515914481301 -0.0110752394328747 0.0019067134188906	705 225 33 22 24 24 25 33 25 705	
Store Filter Filter Manager			•
Response Type	Filter Order	Frequency Specifications	Magnitude Specifications
Lowpass	Specify order: 10	Units: MHz 💌	Units: dB
 ◯ Highpass ➡ ◯ Bandpass 	Minimum order	Fs: 20	Apass: 0.01
Bandstop	Options	Fpass: 1.5	Astop: 100
Differentiator	Density Factor: 16	Fstop: 8.5	
Design Method —			
IIR Butterworth			
FIR Equiripple			
Input processing: Colum	nns as channels (frame based)		esign Filter
Computing Response Done			

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



section with the section of the sect	_ • •
>> num_bits = 16	•
num_bits =	
16	
>> bin_pt = 14	
bin_pt =	
14	=
fx; >>	-

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 <code>num_bits</code> and replace 14 with <code>bin_pt</code>, as shown in the following figure.

😫 Gateway In (Xilinx Gateway In)
Gateway in block. Converts inputs of type Simulink integer, single, double and fixed-point to Xilinx fixed-point or floating-point data type.
Hardware notes: In hardware these blocks become top level input ports.
Basic Implementation
Output Type
🔘 Boolean 💿 Fixed-point 🔘 Floating-point
Arithmetic type Signed (2's comp) 🔻
Fixed-point Precision
Number of bits num_bits Binary point bin_pt
Floating-point Precision
Single Double Custom
Exponent width 8 Fraction width 24
Quantization:
○ Truncate
Overflow: Wrap O Saturate Flag as error
Sample period 1/20e6
<u>OK</u> <u>C</u> ancel <u>H</u> elp <u>A</u> pply

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.

\Lambda System Generator: Lab1_2	
Compilation Clocking General	
Enable multiple clocks	
FPGA clock period (ns) :	Clock pin location :
8.33	
Provide clock enable clear pin Simulink system period (sec):	
1/120e6	
Perform analysis :	Analyzer type :
	Analyzer type : Resource
Perform analysis :	

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.



Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Lab1_2_sol	0	1	105	
Digital FIR Filter	0	1	105	
Digital FIR Filter	0	1	105	

- 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 nth 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 rightclick **Counter** to add this component to the design.





Fixed-Point Designer HDL Coder HDL Verifier Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Coder Simulink Extras Stateflow Xilinx Toolbox Y HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	
 Embedded Coder Fixed-Point Designer HDL Coder HDL Verifier Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Coder Simulink Extras Stateflow Xilinx Toolbox HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	
 Fixed-Point Designer HDL Coder HDL Verifier Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Coder Simulink Extras Stateflow Xilinx Toolbox MBasic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	
Fixed-Point Designer HDL Coder HDL Verifier Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Coder Simulink Extras Stateflow Xilinx Toolbox Y HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	rst5 🔪
 HDL Coder HDL Verifier Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Extras Stateflow Xilinx Toolbox MBDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	st2
Image Acquisition Toolbox Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Extras Stateflow Xilinx Toolbox V HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	rdy
Add block to model Lab1_3 Add block to model Lab1_3 Simulink 3D Animation Simulink Coder Simulink Extras Stateflow Xilinx Toolbox V HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	
 Phased Array System Toolbox Simulink 3D Animation Simulink Coder Simulink Extras Stateflow Xilinx Toolbox MDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	Ctrl+
 Simulink Coder Simulink Extras Stateflow Xilinx Toolbox HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	curr
Simulink Extras Stateflow X Xiinx Toolbox V HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	
Simulink Extras Stateflow Xilinx Toolbox WHDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	Esc
 Xilinx Toolbox HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	
 Xilinx Toolbox HDL Basic Elements DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR 	
Basic Elements Sort in library model order DSP Interfaces Logic and Bit Operations Memory Signal Routing Sources SSR	
 basic Elements > DSP Interfaces Logic and Bit Operations > Memory Signal Routing Sources > SSR 	
Interfaces Logic and Bit Operations > Memory Signal Routing Sources > SSR	
Logic and Bit Operations > Memory Signal Routing Sources > SSR	
Memory Signal Routing Sources SSR	
Signal Routing Sources > SSR	
Sources > SSR	
> SSR	
Tools	
User-Defined Functions	
> HLS Recently Used	

- 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 **b** from the toolbar to review the filter specifications.





承 Block P	Parameters: FDATool			
File Edit	t Analysis Targets View	Window Help		
D 🗳 🔒	a 🖪 🔍 🔍 š 🖾 🛍 🖪	. 📐 😡 😖 🍀 🙁 🗂 🗩 🐻 😡	1 🖸 🗠 🕅	
Curr	rent Filter Information	Filter Coefficients		
Orde		Numerator: 0.0019067134188906 -0.0110752394328747 -0.0411515914481301 0.0351305675326196 0.2887827846112869 0.2887827846112869 0.0351305675326196 -0.0411515914481301 -0.0110752394328747 0.0019067134188906	05 25 3 2 4 2 3 3 25 05	
	Store Filter Filter Manager			v
R	lesponse Type	-Filter Order	-Frequency Specifications	Magnitude Specifications
۹	Lowpass 🔹	Specify order: 10	Units: MHz 💌	Units: dB
C	Highpass Bandpass	Minimum order	Fs: 20	Apass: 0.01
	Bandstop	Options	Fpass: 1.5	Astop: 100
	Differentiator	Density Factor: 16	Fstop: 8.5	
	esign Method —			
	IIR Butterworth			
	FIR Equiripple			
In In	nput processing: Colum	ns as channels (frame based)	▼ De	sign Filter
Ready]

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.



ounter with a	ount limited counter is implemented by combining a comparator.
Basic Im	plementation
Counter type:	ning
Count to value	length(xlfda_numerator('FDATool'))-1
Count direction	n: Down 🛞 Up/Down
Initial value	0
Step	1
Output Preci	sion
Output type: Signed	(2's comp) () Unsigned
Number of bi	ts 4
Binary point	0
Optional Port	ts
Provide lo	sad port
100	inchronous reset port nable port
Explicit Sam	ple Period
Sample perio	id source:
-	d 1/(11*20e6)

- 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.





🔀 ROM ((Xilinx Sin	gle Port Read-Only 🗖 🖻 🔀
Basic	Output	Implementation
Depth	ŀ	ength(xlfda_numerator('FDATool'))
Initial val	ue vector >	dfda_numerator('FDATool')
Memory Dist	ributed me	mory 🔘 Block RAM
Prov	ide reset p	ort for output register
Initial va	lue for out	put register 0
Prov	ide enable	port
Latency :	1	
ОК		Cancel Help Apply

- 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.





😝 DSP Macro 1 0 (Xilinx DSP Macro 1.0)		_	×
Instructions Pipeline Options Implementation			
Valid operands: CONCAT, P, C, PCIN, P>>17, PCIN>>17,	CARRYIN, CARRYCASCIN, ACIN, A	, BCIN,B	
Valid operators: +, -, *, ()			
Valid functions: RNDSIMPLE, RNDSYM			
Instructions are case insensitive and tolerate spaces.			
Target XtremeDSP Slice: DSP48E1			
Instructions	Available Instructions		
A*B A*B	# (A+D)*B (A+D)*B+C (A+D)*B+C+CARRYIN (A+D)*B+CARRYIN (A+D)*B+P (A+D)*B+P+CARRYIN (A+D)*B+P>>17 (A+D)*B+PCIN (A+D)*B+PCIN (A+D)*B+PCIN>17 (A+D)*B+PCIN>17 (A+D)*B+PCIN>17 (A+D)*B+PCIN>17 (A+D)*B-PCIN (A+D)*B-PCIN (A+D)*B-PCIN (A+D)*B-PCIN (A+D)*B-PCIN (A+D)*B-PCIN>17 (A+D)*B-	ı	~
	Show Filtered Instructions		
OK Cancel	Help Apply		

- 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.





鮳 DSP Macro	510 (X	ilinx DSP N	facro 1.0)				_		×
Instructions	Pipelin	e Options	Implementation						
Pipeline Options	By_Tier	• •							
-Custom Pipeli	ne option	s							
Tier: D A B CONCAT C CARRYIN CONTROL					6	Pre-adde DSP48E	ar only su 1 and DS gister	upported on SF48AV1	
Tier 1		Tier 2	✓ Tier 3	Tier 4		Tier 5		Tier 6	
D		D	D						
A		A	- A	A					
В		В	В	В	\checkmark	M			
			CONCAT	CONCAT		CONCAT	\checkmark		
		C				CARRYIN		F	
CONTROL		CONTROL				CONTROL			
		ОК	Cancel	Help)	Apply			

- 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.

Resource Analyzer: Lab1_3 Post Synthesis Resources: Clicking on an ins	stance name highlights (corresponding b	olock/subsystem	in the model.
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
rO	0	0	0	16
Relational1	0	0	0	t
ROM	0.5	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	(
			ОК	Help

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).



Basic	DSP48				
onstant	value My	Coeffs(1)			
Output	Туре				
B	olean 🤇	Fixed-po	int 💿 Float	ing-point	
Arithme	tic type F	loating-pol	int 👻		
	point Pred				
	8. j			-	
Numb	er of bits	16	Binary po	int 14	
Floati	ng-point P	recision			
10200	8 m cz		© Cutur		
	ingle	Double	Custom		
Ехро	nent width	8	Fraction	width 24	
Sample	Period				
✓ Sam	pled cons	tant			
Sample	period 1	20.06			
Sample	period 1	2000			
ОК		Cancel	He		Apply
UN		Cancer	ne	P	Mppiy

- 5. Close the Constant1 Properties editor.
- 6. Return to the top-level design using the toolbar button Up To Parent Υ , 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.

Name	Registers
	(407600)
4 Lab1_4_1 0 33 5578	133
▷ FIR 0 33 5578	133.

- 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).

Post Synthesis Resources: Clicking on an instance	e name highlights corresponding	block/subsystem i	n the model.	
Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
✓ Lab1_1_sol	0	6	281	402
Digital FIR Filter	0	6	281	402
2				

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 <code>open Lab1_4_2.slx</code> 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** \rightarrow **Number of Inputs** \rightarrow **3**.
 - b. Select View \rightarrow Configuration Properties and confirm that the Number of input ports is 3.

▲ Scope	×					
File Tools View Simulation Help	ъ					
© - < l> ■ ⇒ - Q - I = / ■						
	—T					
8						
6 Configuration Properties: Scope						
Main Time Display Logging						
4 Depen at simulation start	Open at simulation start					
2 Display the full path						
Number of input ports: 3						
Sample time: -1						
Input processing: Elements as channels (sample based) •						
Maximize axes: Off						
Axes scaling: Manual Configure						
-8 OK Cancel Apply						
-10 0.5 1 1.5 2 2.5 3 3.5 4 4.5	5					
×10	_					
Ready						

- c. Click **OK** to close the Configuration Properties dialog box.
- d. Select File \rightarrow 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 \rightarrow 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 $View \rightarrow Configuration$ 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.

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Lab1_4_2	0	44	6167	1926
FIR-Fixed-Point	0	11	589	578
▷ FIR	0	33	5578	1332
Convert	0	0	0	10

- 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.





😸 MCode (Xilinx MCode Block)
Pass input values to a MATLAB function for evaluation in Xilinx fixed-point type. The input ports of the block are input arguments of the function. The output ports of the block are output arguments of the function.
Basic Interface Advanced
Block Interface MATLAB function
Browse Edit M-File
Explicit Sample Period
Specify explicit sample period
1

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.





FILE NAVIGATE EDIT Breakpoints Run Run and Advance Advance state_machine.m + state_machine.m H H	7 P 🛱 🎝	20	2						
			0 0 0	\triangleright		Nun Section			
FILE	NAVIGATE	EDIT	Breakpoints T	Run ▼		Advance	Run and Time		
	-		BREAKPOINTS			RUN			
	state_mach	ine.m	× +						
1	LE NAVIGATE EDIT EDIT Breakpoints Run Run and Advance Run and Time BREAKPOINTS RUN								
2	FLE NAVIGATE EDIT Breakpoints THE NAVIGATE EDIT Breakpoints Breakpoints BREAKPOINTS BREAKPOINTS BREAKPOINTS RUN State_machine.m + function matched = state_machine (din)								
							Ln 2	Col 1	

- 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:



bla 🔁	ack_box	_examp	le1/Down	Convert	er/Tra	nspose	e FIR Fil	ter Blac	k Box						• X
File	Edit	View	Display	Diagra	n Si	mulati	on A	nalysis	Code	Tools	Help				
2	- 8		⇐ 🔿		2	<u>و</u>	•	4			•	500			» 🥑 🔻
Tran	spose FI	R Filter B	Black Box												
۲	🍡 blac	k_box_e	example1	▶ 🔁 Do	wn Co	nverte	r ▶ 🖄	Transpo	ose FIR F	ilter Blac	k Box				•
Image: Solution of the second sec															
	Image: Solution of the second sec														
⇒		(
AE			In										X 1)	
													Out	·	
		(_												
			rst												
»															
Read	у							130	%					FixedStep	Discrete

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.

→ × ↑ 📙	« ug14	498-vitis-model-composer-tutorial	> HDL_Library > Lab2 > HDL	ٽ ~	,○ Search HDL
janize 🔻 🛛 Ne	w folder				III 🕶 🛄 🤇
This PC	^	Name	Date modified	Туре	Size
3D Objects		solution	6/10/2021 7:10 PM	File folder	
Desktop		📔 mac	11/9/2013 4:14 AM	VHD File	21
Documents		📔 transpose_fir	8/6/2014 9:11 PM	VHD File	31
🕹 Downloads					
Music					
Pictures					
Videos					
Windows (C:)	~ <				
	File <u>n</u> an	ne: mac		~	All Supported HDL Files (*.v, *.v ~
					Open Cancel

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:

😝 Black Box5 (Xilinx Black Box)
Incorporates black box HDL and simulation model into a System Generator design.
You must supply a Black Box with certain information about the HDL component you would like to bring into System Generator. This information is provided through a Matlab function.
When "Simulation mode" is set to "Inactive", you will typically want to provide a separate simulation model by using a Simulation Multiplexer. When "Simulation mode" is set to "External co-simulator", you must include a ModelSim block in the design.
Basic Implementation
Block configuration m-function
transpose_fir_config
Simulation mode: Inactive Vivado Simulator External co-simulator HDL co-simulator to use (specify helper block by name)
Verbose
OK Cancel Help Apply





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



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 \rightarrow Xilinx Design Tools \rightarrow 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.



🐋 Select Folder							×	
← → • ↑ <mark>.</mark> «	HDL_	Library > Lab2 > C_code	~	ē		de		
Organize 🔻 New fo	older						?	
2021.1	^	Name	Date modifie	ed	Туре	Size	_	
tutorial			hls_project	6/10/2021 7:	10 PM	File folder		
UserGuide Versal_Vitis OneDrive - Xilinx, User This PC 3D Objects Desktop Cocuments		- solution	6/10/2021 7:	10 PM	File folder			
Downloade	× <						>	
Fo	lder:	hls_project						
					Select Folder	Cancel		

- 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**.





×
**
Browse
Browse
Browse
].

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 ${\tt Vitis}~{\tt HLS}$ 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.



😝 Vitis HLS (Xilinx High Level 🛛 —		\times
This block allows including C,C++ and SystemC so System Generator for DSP designs.	ource files i	n
Colution w/Lob2/C. and /blo. project/colution1	Drouwer	
Solution pr/Lab2/C_code/hls_project/solution1'	Browse	
Refresh	Edit	
Use C simulation model if available		
Display signal types		
Output Sample Times Simulink system period	•	
OK Cancel Help	Арр	ly

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.



🔁 Block Parameters: Image From File	\times
Image From File (mask) (link)	
Reads an image from a file.	
Use the File name parameter to specify the image file you want to import into your model. Use the Sample time parameter to set the sample period of the block.	
Main Data Types	
Parameters	
File Name: DL_Library\Lab2\C_code\xilinx_logo.jpg Browse	
Sample time: ImSize*ImSize	
Image signal: Separate color signals	
Output port labels: R G B	
OK Cancel Help Apply	

- 9. Click **OK** to exit the Block Parameters dialog box.
- 10. Use the Up to Parent \hat{T} 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.





Compilation	Clocking	General		
Enable m	ultiple clocks			
FPGA cloc	ck period (ns):	Clock pin locat	ion :
2.0				
Provide cl	lock enable clea	r pin		
Simulink s	system perio	d (sec) :		
1				
'				
Perform a	nalysis :		Analyzer type :	
· · · · · · · · · · · · · · · · · · ·	-	•	Analyzer type : Timing	▼ Launch

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.





iolatio	on type setup				~				Status : FA	JL
	Slack (ns)	Delay (ns)	gic Delay (ns)	ing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	-
L	-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	
F	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	
	0.837	0.65	0.216	0.434	0	Lab3/Del	Lab3/sub	clk	clk	
	0.845	0.902	0.335	0.567	1	Lab3/Del	Lab3/Reg	clk	clk	
	1.058	0.962	0.962	0	0	Lab3/Del	Lab3/Del	clk	clk	
	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.

Compilation	Clocking	General				
Enable m	ultiple clocks					
FPGA clos	ck period (ns)):	Clock pin location :			
2.0						
Provide cl	ock enable clear	pin				
Simulink s	system period	d (sec) :				
1						
Perform analysis :			Analyzer type :			
Post Synthe	sis	•	Timing Launch			
Performance	e Tips Gener	rate OK	Apply Cancel Help			





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.





😸 Mult (Xilinx Multiplier)
Hardware notes: To check for the optimum internal pipeline stages of the dedicated multiplier you must select 'Test for optimum pipelining'.
Optimization Goal: For implementation into device fabric (LUTs), the Speed or Area optimization will take effect only if it's supported by IP for the particular device family. Otherwise, the results will be identical regardless of the selection.
Basic Implementation
Output Type
Precision:
Fixed-point Output Type
Arithmetic type: Signed (2's comp) Unsigned Fixed-point Precision
Number of bits 36 Binary point 34
Quantization: (a) Truncate Round (unbiased: +/- Inf) Overflow: (a) Wrap Saturate Flag as error
Optional Port Provide enable port
Latency 1
OK Cancel Help Apply

- 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.

Post Synthesis Timing Paths: Clicking on a timing path highlights corresponding blocks in the model.							
lolatio	Slack (ns)	Delay (ns)	gic Delay (ns)	ing Delay (ns)	Levels of Logic	Source	Status PAS
1	0.176	1.811	1.306	0.505	14		Lab3/sub
2	0.692	0.973	0.539	0.434	0	Lab3/sub	Lab3/sub
3	0.696	1.291	1.045	0.246	9	Lab3/add	Lab3/add
4	0.812	1.175	0.684	0.491	3	Lab3/add	Lab3/add
5	0.827	1.158	0.752	0.406	3	Lab3/add	Lab3/add
6	0.837	0.65	0.216	0.434	0	Lab3/Del	Lab3/sub
5 6 <	2012 - 2010 - 20						10/2/38:

Note:

- 1. For quicker timing analysis iterations, post-synthesis analysis is preferred over postimplementation 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**.



承 System G	enerator: Lab3				
101300 0001					
Compilation	Clocking	General			
Enable m	ultiple clocks				
FPGA clos	ck period (ns):	Clock pin loca	tion :	
2.0					
_	lock enable clear				
	system perio	d (sec) :			
1					
Perform a	nalysis :		Analyzer type	:	
Post Synthe	sis	•	Resource	▼ La	unch
Performance	e Tips Gene	rate OK	Apply	Cancel	Help

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.





🔀 Resource Analyzer: Lab3

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)	
/ Lab3	0.5	1	153		273
> subsystem1	0.5	1	97		49
> addr_gen	0	0	54		105
Register1	0	0	0		1
Register	0	0	0		48
Delay7	0	0	1		1
Delay4	0	0	0		20
Delay3	0	0	1		1

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** \rightarrow **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 oversampling 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 FIF01/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 Υ 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 Υ 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 FIF01/we.
 - b. Connect FIFO1/empty to Gain Control/In2.
 - c. Connect Gain Control/DDC_Read to FIF01/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.





\Lambda System Generator: Lab4_1			
Compilation Clocking General			
Enable multiple clocks			
FPGA clock period (ns) :	Clock pin location :		
1e9/491.52e6			
Provide clock enable clear pin			
Simulink system period (sec) :			
1/491.52e6			
Perform analysis :	Analyzer type :		
None	Timing	Launch	

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.



📣 System Gener	ator: Lab4_1	/CTRL			
Compilation C	locking	General			<i>.</i>
Enable multip	le clocks				
FPGA clock p	eriod (ns)	:	Clock pin loca	ation :	
1e9/100e6					
Provide clock Simulink syst					
Contraction of the					
Perform analy	/sis :		Analyzer type	•:	
None			Timing	.	Launch

- 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.



Clock pin locatio	on :	
Clock pin locatio	on :	
Clock pin locatio	on :	
Clock pin locatio	on :	
Analyzer type :		
Timing	•	Launch
Apply	Cancel	Help
	Timing	Timing

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.





承 System G	enerator: Lab4	_1/DDC	
Compilation	Clocking	General	
Enable m	ultiple clocks		
FPGA clo	ck period (ns):	Clock pin location :
1e9/491.52e	6		
Simulink	lock enable clear system perio		1
1/491.52e6			
Perform a	nalysis :		Analyzer type :
None		-	Timing Launch
Performance	e Tips Gene	rate OK	Apply Cancel Help

- 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.

🍌 New Project		×
Project Name Enter a name for y	your project and specify a directory where the project data files will be stored.	4
Create proje	project_1 C:/ug1498-vitis-model-composer-tutorial/HDL_Library/Lab5/IPI_Project C:/ug1498-vitis-model-composer-tutorial/HDL_Library/Lab5/IPI_Project/project_1 C:/ug1498-vitis-model-composer-tutorial/HDL_Library/Lab5/IPI_Project/project_1	8
?	< <u>B</u> ack <u>N</u> ext> Einish Ca	ncel

- 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

fault Part					
bose a default Xilinx part or board for your project.					
Parts Boards					
Reset All Filters					
/endor: All Vame: All			~	Board Rev: Lat	est
Search: Q-					
Display Name	Preview	Status	Vendor	File Version	Part
ZYNQ-7 ZC702 Evaluation Board Add Companion Card Connections		Installed	xilinx.com	1.4	xc7z020clg484-1
ZYNQ-7 ZC706 Evaluation Board	्रां स्ट्रेली से स्ट्र				
Add Companion Card Connections		Installed	xilinx.com	1.4	xc7z045ffg900-2
Kintex-UltraScale KCU105 Evaluation Platform					
Add Companion Card Connections		Installed	xilinx.com	1.7	xcku040-ffva1156
<	1				>
Refresh Catalog was last updated on 06/11/2021 5:06:27 PM					
Catalog was last updated on 00/11/2021 5.00.27 PM					

- 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.

Diagram				? 🗆 🖒 X
⊕, Q, X, X, ⊕ Q,	×	$\mathbf{r} = \mathbf{r} + \mathbf{r} \mathbf{r} + \mathbf{r} \mathbf{r}$	୯ ା	0
ſ				
		Block Design Properties	Ctrl+E	
	\times	Delete	Delete	
		Сору	Ctrl+C	
		Paste	Ctrl+V	
This design is	O,	Search	Ctrl+F	
This design is	123	Select All	Ctrl+A	
	+	Add IP	Ctrl+I	
		Add Module		
		IP Settings		
	Y	Validate Design	F6	
		Create Hierarchy		
		Create Comment		
		Create Port	Ctrl+K	

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

i Settings	8
Q- Project Settings	IP > Repository Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.
Simulation Elaboration	IP Repositories
Synthesis Implementation Bitstream	+ - ± ∓
✓ IP Repository	Press the + button to Add Repository
Packager Tool Settings Project	Add Repository Refresh All
IP Defaults ~	OK Cancel Apply Restore

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.

ecent: 🚍 C:/ug1498-vitis-model-com 🔻 👔 🏠 📮 🛓 🙏 🖵 🗋 🗙 📾	¥ c
irectory: C:\ug1498-vitis-model-composer-tutorial\HDL_Library\Lab5	
> Lab2	1
> Lab3	
> Lab4	
V Lab5	
V IPI_Project	
Xii	
V 📴 ip	
> constrs	
> drivers	- 1
> 📙 hdi	
> 📊 images	
lab5_1_c_counter_binary_v12_0_i0	
> ab5_1_dist_mem_gen_i0	
> lab5_1_dist_mem_gen_i1	
> ab5_1_dist_mem_gen_i2	
> ab5_1_dist_mem_gen_i3	
> ab5_1_dist_mem_gen_i4	
> lab5 1 dist mem gen i5	1

- 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.





Diagram				? 🗆 🖒 X
Θ, Θ,	20	🔀 O Q 🖫	≑ +	ା ା 🖌 🖉 🛛 C ା ଫାା ଅତା 🔅
				1
		Properties	Ctrl+E	m7 0
	\times	Delete	Delete	
		Сору	Ctrl+C	
		Paste	Ctrl+V	FIXED_IO + FIXED_IO
	Q,	Search	Ctrl+F	M_AXI_GP0 +
	123	Select All	Ctrl+A	TTC0_WAVE0_OUT
	+	Add IP	Ctrl+I	TTC0_WAVE1_OUT - TTC0_WAVE2_OUT -
		Add Module		FCLK_CLK0
		IP Settings		FCLK_RESET0_N
	r	Validate Design	F6	System
		Create Hierarchy		1

- **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 HPO** interface.

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





Documentation 🔅 Presets	P Loca	ation 🍈 Import XPS Settings					
Page Navigator	PS-PI	L Configuration		Summary Report			
Zynq Block Design	+	Search: Q-					
PS-PL Configuration	Q	Name	Select	Description			
Peripheral I/O Pins	¥	> General					
		> AXI Non Secure Enablement	0 ~	Enable AXI Non Secure Transaction			
MIO Configuration	\$	GP Slave AXI Interface					
Clock Configuration		 HP Slave AXI Interface 					
Clock Conliguration		> S AXI HP0 interface	\checkmark	Enables AXI high performance slave interface 0			
DDR Configuration		> S AXI HP1 interface		Enables AXI high performance slave interface 1			
		> S AXI HP2 interface		Enables AXI high performance slave interface 2			
SMC Timing Calculation		> S AXI HP3 interface		Enables AXI high performance slave interface 3			
Interrupts		> ACP Slave AXI Interface					
		> DMA Controller					
		> PS-PL Cross Trigger interface		Enables PL cross trigger signals to PS and vice-versa			

- 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.

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

Favorites	Name	Date modified	Туре	Size
Creative Cloud Files	lab6_1.c	3/20/2017 8:46 AM	C File	2 KB
E Desktop	lab6_1.h	3/20/2017 8:46 AM	H File	5 KB
laces Recent Places	lab6_1_hw.h	3/20/2017 8:46 AM	H File	1 KB
👍 Downloads	lab6_1_linux.c	3/20/2017 8:46 AM	C File	5 KB
	lab6_1_sinit.c	3/20/2017 8:46 AM	C File	2 KB
🔋 Libraries	Makefile	3/20/2017 8:46 AM	File	1 KB
Documents				
🕹 Music				

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

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;
3
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 \rightarrow Export \rightarrow Export Hardware.
- 5. Click **Next** in the Export Hardware Platform page.







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

🝌 Expo	ort Hardware Platform	×
	ut platform properties to inform downstream tools of the intended use of the platform's hardware design.	2
\bigcirc	Pre-synthesis This platform includes a hardware specification for downstream software tools.	
۲	Include bitstream This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.	
	< <u>B</u> ack <u>N</u> ext > <u>Finish</u> Cancel	

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





À Export Hardwa	re Platform X
Files Enter the name o	f your hardware platform file, and the directory where the XSA file will be stored.
XSA file name:	design_1_wrapper
Export to:	C:/ug1498-vitis-model-composer-tutorial/HDL_Library/Lab6/IPI_Project/project_1
	The XSA will be written to: C:\ug1498-vitis-model-composer-tutorial\HDL_Library\Lab6\\PI_Project\project_1\
	< <u>B</u> ack <u>Next</u> > <u>F</u> inish Cancel

- 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.

🧹 Vitis IDE La	auncher	×
Select a dire	ectory as workspace	
Vitis IDE use	s the workspace directory to store its preferences and development artifacts.	
Workspace:	C:\ug1498-vitis-model-composer-tutorial\HDL_Library\Lab6 V Brows	ie
Use this a	s the default and do not ask again	
Restore of	ther Workspace	
Recent W	orkspaces	
	Launch	ancel

- 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 $Lab6 \rightarrow IPI_Project \rightarrow project_1$, select design_1_wrapper.xsa and click Open.



Hardware	Specification C\ug1498-vitis-model-composer-tutorial\HDL_Library\Lab6\IPI_Project\project_1\design_1_wrapper.xsa]
XSA File:	vck190 vck190_es1 vmk180_es1 zc702 zc706 zcu102 zcu106 zed <u>Cr\ug1498-vitis-model-composer-tutoriaf\HDL_Library\Lab6\IPI_Project_1\design_1_wrapper.xsa</u>	Browse
Platform r	ame:	

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**.

✓ New Application Project Application Project Details					×
Specify the application project nar	ne and its system project proper	ties			
Application project name: Des_T System Project Create a new system project fo Select a system project + Create new	r the application or select an exis System project details System project name: Target processor	sting one from the workpsace ① Des_Test_system for the Application project.			
	Processor ps7_cortexa9_0 ps7_cortexa9_1 Show all processors in	Associated applications Des_Test the hardware specification 🗸 1) 	~	
?	< Bac	k Next > Finish	1	Cancel	

- 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.



🕒 Explorer 🛛 🕞 🔁 🕴 💻 🗉			
✓ G design_1_wrapper			
> 🗁 export			
> 🗁 hw			
> 🔁 logs			
✓ iog3 ✓ i> ps7_cortexa9_0			
 > bs/_contextaj_0 > bs/_contextaj_0 > bs/_contextaj_0 			
✓ ≥ bsp			
✓ ≥ ps7_cortexa9_0			
≥ code			
> 🗁 include			
🗁 lib			
✓ ➢ libsrc			
> 🗁 axidma_v9_12			
> 🗁 canps_v3_5			
> > coresightps_dcc_v1_8			
> 🗁 cpu_cortexa9_v2_10			
> 🗁 ddrps_v1_2			
> 🗁 devcfg_v3_7			
> 🗁 dmaps_v2_7			
> > emacps_v3_12			
> 🗁 generic_v2_1			
> 😕 gpiops_v3_8			
> > iicps_v3_12			
✓ ➢ lab6_1_v1_2			
V 🗁 SFC			
> 🖻 lab6_1_g.c			
> 🖻 lab6_1_hw.h			
> 🖻 lab6_1_linux.c			
> 🖻 lab6_1_sinit.c			
> 🖻 lab6_1.c			
> 🗈 lab6_1.h			
là Makefile			
> 🗁 qspips_v3_8			
> 😕 scugic_v4_3			

- 21. Power up the ZC702 board to program the FPGA.
- 22. Click Xilinx \rightarrow 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.





🖌 Show View 👘 —		
vitis	6	2
 ✓ ➢ Xilinx ☑ Vitis Log ☑ Vitis Serial Te 		
Vitis Timeline	e View	
Open	Cancel	

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

🗏 Vitis Serial Terminal 🛛		🕂 🗶 🖉 🗖	🚽 Connect	to serial port	×
Click on + button to add a	a port to the terminal.		Basic Sett	ings	
		^	Port:	COM3	~
			Baud Rate	e: 115200	~
<		~	- Advanc	e Settings	
		Send Clear	Data Bits	8	~
			Stop Bits	1	~
			Parity:	None	~
			Flow Con	trol: None	~
			Timeout	[sec):	
				Ж	Cancel

- 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** \rightarrow **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 \rightarrow 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).

모 Vitis Serial Terminal 🛛	\$	×	R		۵
Connected to: Serial (COM3,	115200, 0, 8)				
AXI DMA Status = 100010000					^
AXI DMA Status = 100210020					
DES cipher text output:					
YeF.\.g.O 88 d7 b5 0e aa	59 65 46 ec 5c e1 67 91 4f a4 dd				
b.M. .v.zY(- 62 8a 4d f8 20 7	'c d1 76 2e 7a 59 da 93 d2 07 7b				
t5.R\$ e3 74 15 1e 35 b5	52 9e 12 c6 11 db d8 0d 24 9c				
ZR.2a^`5." - 1e cd da 5a 52	d7 32 61 e0 0b da 5e 60 35 84 22				
b].V,y* d8 cf 89 62 5d c6	56 05 af 5f 02 79 b9 a2 2a 16				
DES deciphered output:					
This is a secret - 54 68 69 73 2	0 69 73 20 61 20 73 65 63 72 65 74				
message that mu - 20 6d 65 7	3 73 61 67 65 20 74 68 61 74 20 6d 75				
	65 20 68 69 64 64 65 6e 2c 20 6e 6f				
	4 74 65 72 20 77 68 61 74 2e 20 57 61				
it - what? - 69 74 20 2d 20	77 68 61 74 3f 20 20 20 20 20 20 20				
DES core - example all done					
					*
<				2	
		Sen	d	Cle	ear

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 <code>Sobeledge_lib.slx</code> 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** \rightarrow **Run** command or click the \bowtie 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.





Block Parameters: Data Type Conversion				
Data Type Conversion				
Converts the input value into the user selected data type. This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.				
Parameters				
Output data type: fixed 🔹				
Output Type Attributes				
Signedness: Unsigned				
Word length: 8 Fractional length: 7				
Input Conversion Attributes Round: Truncation to minus infinity Overflow: Wrap around				
OK Cancel Help Apply				

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.

Block Parameters: Data Type Conversion
Data Type Conversion
Converts the input value into the user selected data type. This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.
Parameters
Output data type: single
OK Cancel Help Apply

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 \rightarrow 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, $x_s fix16_En8$ 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.

🔁 Block Parameters: Data Type Conversion	Block Parameters: Gain
Data Type Conversion	Gain
Converts the input value into the user selected data type. This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.	Applies gain to the input. This block warns or errors out when an integer output overflows during simulation. To configure, in the Configuration Parameters > Diagnostics > Data Validity pane, set the Wrap or Saturate on overflow.
Parameters	Parameters
Output data type: data type expression	Gain:
InputDataType	65.738/256
	Gain data type: data type expression -
Saturate on integer overflow	FDataType
	Output data type same as input
	Saturate on Integer Overflow
OK Cancel Help Apply	OK Cancel Help Apply

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'
```

 $'x_sfix_{En6}'$ 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 <code>basic_array</code> function in <code>C++</code>, which calculates the sum of two arrays of size 4. You will import this function as a Model Composer block using the <code>xmcImportFunction</code> 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 <code>basic_array</code> function, the <code>in1</code> and <code>in2</code> 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 <code>basic_array</code>.

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:





🛅 Block Parameters: basic_array				
Library Function				
Function declaration				
	array(uint8_t t uint8_t in2	out1[4], con [4]);	st uint8_t	
Function	General			
Interfaces				
Direction	Name	Туре	Dimension	
O to t				
Output	out1	uint8_t	4	
Input	in1	uint8_t	4	
Input	in2	uint8_t	4	
OK	Cancel	Help	Apply	

- 6. Open the test_array.slx model, which is just a skeleton to test the generated block.
- 7. Add the generated <code>basic_array</code> 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.

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.

7. Similar explanation follows for Mux function.

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 initFon 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.



🚹 Block Param	eters: Demux		×
Library Functi	on		
Function decla	aration		
Demux(ap_fix ap_fixed <w, 1<="" td=""><td>NUMOFELEMENT ed<w, i=""> vecto [> vector_out0[f 1[NUMOFELEME</w,></td><td>r_in[NUMOFELE NUMOFELEMEN⁻</td><td></td></w,>	NUMOFELEMENT ed <w, i=""> vecto [> vector_out0[f 1[NUMOFELEME</w,>	r_in[NUMOFELE NUMOFELEMEN ⁻	
Function (General		
Template par	ameters		
Name		Туре	
NUMOFELEME W I	ENTS	int int int	
Interfaces			
Direction	Name	Туре	Dimension
Input	vector_in	ap_fixed <w, I></w, 	NUMOFELEMENTS
Output	vector_out0	ap_fixed <w, I></w, 	NUMOFELEMENTS / 2
Output	vector_out1	ap_fixed <w, I></w, 	NUMOFELEMENTS / 2
	ОК	Cancel	Help Apply

- d. Next, double-click the Mux block and observe the Template parameters and Dimension.
- 13. Add a Display block at the input and output as shown in the following figure and simulate the model to observe the results.





- 14. To understand how 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.

Paramet Block Paramet	ters: Data Type Conversion	×
Data Type Con	version	
This block warn	put value into the user selected data type. is or errors out when an integer output overflows during simulation. To e Configuration Parameters > Diagnostics > Data Validity pane, set the te on overflow.	
Parameters		
Output data typ	pe: fixed	•
Output Type A	Attributes	
Signedness:	Signed	•
Word length:	32 Fractional length: 16	:
Input Convers	sion Attributes	
Round:	Truncation to minus infinity	•
Overflow:	Wrap around	•
	OK Cancel Help Appl	у

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.

E XILINX.



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.

🚹 Model Properties: design	1	×
Main Callbacks Hi	istory Description Data	
MainCallbacksHiModel callbacksPreLoadFcnPostLoadFcnInitFcn*StartFcnPauseFcnContinueFcnStopFcnPreSaveFcnPostSaveFcnCloseFcnCloseFcn	istory Description Data Model initialization function: SSR=16;	
	OK Cancel Help App	oly

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).

Send Feedback



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.

Model Composer e		×
Pick an example to open:		
color_detection fir_import import_function		^
optical_flow sobel_edge_detection video_frame_rotation		
		~
Open example	Cancel	

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







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







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

Block Parameters: calculating_roots	×
Import Function	
Function declaration	
void calculating_roots(int16_t aPix, int16_t bPix, int16_t cPix, int16_t txPix, int16_t tyPix, int16_t IxPix, int16_t IyPix, int16_t dtPix, float & VxPix, float & VyPix);	

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.





С	irrent Folder
	📄 Name 🔺
+	html
-[<pre>Mail Content Cont</pre>
	🚵 getGainValue.m
	import_function.m
	isSourceMovingObject.m
	object.png
	渣 OpticalFlowRoots.slx
	🖄 switchCallback.m
	🖄 updateHistogram.m
	🔊 xmc_optical_flow.slx

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 <u>gdb</u> to debug your C/C++ code.

MATLAB process ID is <u>15972</u>.

You can also get the process ID by typing "feature getpid" in the MATLAB command window.
```

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

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





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

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.





MSYS2 MINGW64 Shell	—		\times
[New Thread 15972.0x18e8]			~
[New Thread 15972.0x57e4]			
[New Thread 15972.0x3120]			
[New Thread 15972.0x3fcc]			
[New Thread 15972.0x3f2c]			
[New Thread 15972.0x2eb8]			
[New Thread 15972.0x54bc] [New Thread 15972.0xb04]			
[New Thread 15972.0x004] [New Thread 15972.0x4c80]			
[New Thread 15972.0x4680]			
[New Thread 15972.0x5a90]			
[New Thread 15972.0x43e8]			
[New Thread 15972.0x5c08]			
New Thread 15972.0x2a3c]			
[New Thread 15972.0x4f90]			
[New Thread 15972.0x3cb0]			
New Thread 15972.0x5be8			
New Thread 15972.0x4b14			
[New Thread 15972.0x51d0]			
[New Thread 15972.0x6798]			
[New Thread 15972.0x3698]			
Reading symbols from C:\Program Files\MATLAB\R2018a\bin\win64	4\MATLAB.€	exe(r	io d
ebugging symbols found)done.			
(gdb)			\sim

7. Type cont at the Windows command prompt.

MSYS2 MINGW64 Shell	—		×
[New Thread 15972.0x3cb0] [New Thread 15972.0x5be8] [New Thread 15972.0x5b4] [New Thread 15972.0x51d0] [New Thread 15972.0x6798] [New Thread 15972.0x3698] Reading symbols from C:\Program Files\MATLAB\R2018a\bin\win64\M ebugging symbols found)done.	ATLAB.0	exe(1	no d
(gdb) cont Continuing. [Thread 15972.0x3698 exited with code 0] [New Thread 15972.0x666c] [New Thread 15972.0x471c] [New Thread 15972.0x471c] [New Thread 15972.0x37c] [New Thread 15972.0x37c] [New Thread 15972.0x440c]			
[New Thread 15972.0x6750] [New Thread 15972.0x29dc] [New Thread 15972.0x5788] [New Thread 15972.0x2b98] [New Thread 15972.0x3b44] [New Thread 15972.0x5f30]			~

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



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



MSYS2 MINGW64 Shell	—		×
[New Thread 15972.0x308] [New Thread 15972.0x50e0] [Thread 15972.0x50e0] [Thread 15972.0x50e0 exited with code 0] [New Thread 15972.0x56a8] [Thread 15972.0x56a8 exited with code 0] [New Thread 15972.0x6470] [Thread 15972.0x6470 exited with code 0] [Switching to Thread 15972.0x3098]			^
<pre>Thread 1 hit Breakpoint 1, calculating_roots (aPix=210, bPix=81 txPix=<optimized out="">, tyPix=tyPix@entry=346, IxPix=IxPix@e IyPix=IyPix@entry=30, dtPix=dtPix@entry=39, VxPix=@0x20e650490: 2.5743929e-036, VyPix=@0x20e6775a0: 2.5 at calculating_roots.h:53 int16_t eig1 = (r + s);</optimized></pre>	ntry=3	1,	

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.



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 <code>quit</code> 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 sublibrary.
- 5. Find the Model Composer Hub block, and add it into the design as shown in the following figure.

Send Feedback





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.

🚹 Block Parameters: Model Composer Hub	×
Model Composer Hub	
Controls implementation of the specified sul Use the 'Code Generation' tab to generate of and run the generated code.	,
Code Generation Hardware Feedbac	k
Subsystem name:	
Edge_Detection	
Code directory:	
./codegen_edge_detection	Browse
Target: IP Catalog •	Settings
Create and run testbench	
	Generate
OK Cancel	Help Apply

- 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).



Block Parameters: Interface Spec		×
Interface Spec		
Specifies the RTL interfaces for a subsystem.		
Parameter and Port Properties		
Function Protocol Input ports Output ports		
Mode: AXI4-Lite Slave	▼ Bundle:	
		OK Cancel Help Apply

- 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.

Block Parameters: Inter	face Spec			X
Interface Spec				
Specifies the RTL inter	rfaces for a sub system.			
Parameter and Port Pr	roperties			
Function Protocol	Input ports Output por	ts		
Name	Mode	Bundle	Offset	Video Format Video Component
Υ	AXI4-Stream (video) 🔻	image_in		YUV 4:2:2 • Y •
Cb	AXI4-Stream (video) -	image_in		YUV 4:2:2 • U •
Cr	AXI4-Stream (video)	image_in		YUV 4:2:2 • V •
			ОК	Cancel Help Apply
指 Block Parameters: Inte	erface Spec			×
Block Parameters: Interface Spec	erface Spec			×
—				×
Interface Spec	faces for a subsystem.			×
Interface Spec Specifies the RTL interf	faces for a subsystem.	s		×
Interface Spec Specifies the RTL interf Parameter and Port Pro	faces for a subsystem.	s Bundle	Offset	X Video Format Video Component
Interface Spec Specifies the RTL interf Parameter and Port Pro Function Protocol	faces for a subsystem. operties Input ports Output port		Offset	
Interface Spec Specifies the RTL interf Parameter and Port Pro Function Protocol Name	faces for a subsystem. operties Input ports Output port Mode	Bundle	Offset	Video Format Video Component
Interface Spec Specifies the RTL interf Parameter and Port Pro Function Protocol Name Y_Out	faces for a subsystem. operties Input ports Output port Mode AXI4-Stream (video)	Bundle image_out	Offset	Video Format Video Component YUV 4:2:2 V Y
Interface Spec Specifies the RTL interf Parameter and Port Pro Function Protocol Name Y_Out Cb_Out	faces for a subsystem. operties Input ports Output port Mode AXI4-Stream (video)	Bundle image_out image_out	Offset	Video Format Video Component YUV 4:2:2 V Y V YUV 4:2:2 V U V
Interface Spec Specifies the RTL interf Parameter and Port Pro Function Protocol Name Y_Out Cb_Out	faces for a subsystem. operties Input ports Output port Mode AXI4-Stream (video)	Bundle image_out image_out	Offset	Video Format Video Component YUV 4:2:2 V Y V YUV 4:2:2 V U V

• 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 <code>image_in</code> port in the generated IP while the YCbCr outputs are bundled together as an <code>image_out</code> 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 <code>codegen_IP/Edge_Detection_prj</code> folder in the Current Folder, which contains the generated IP <code>solution1</code> folder.



XILINX.

<pre>RTMG 210-285] Implementing FIFO 'p_dstx_V_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dsty_V_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_xFSobelfYi U(start_for_xFSobelfYi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbs_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_start_for_magnitubCU(start_for_magnitubC)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubMU (ustart_for_magnitubM)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcKU(start_for_Loop_1_qcK)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Registers.</pre>	▲ - □ ×
<pre>RTMG 210-285] Implementing FIFO 'p_dstx_V_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dsty_V_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_xFSobelfYi U(start_for_xFSobelfYi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbs_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_start_for_magnitubCU(start_for_magnitubC)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubMU (ustart_for_magnitubM)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcKU(start_for_Loop_1_qcK)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Registers.</pre>	Generating RTL for module 'Edge_Detection'
<pre>RTMG 210-285] Implementing FIFO 'p_dstx_V_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dsty_V_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_xFSobelfYi U(start_for_xFSobelfYi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbs_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_start_for_magnitubCU(start_for_magnitubC)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubMU (ustart_for_magnitubM)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcKU(start_for_Loop_1_qcK)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Registers.</pre>	
RTMG 210-285] Implementing FIFO 'p_dsty_V_U(fifo_w16_d2_A)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_xFSobelfYi_U(start_for_xFSobelfYi)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Register: RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Register: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Register: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohba_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(start_for_magnitujbC)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubC_U(start_for_magnitujbC)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubD_U(start_for_XfMatTolbW)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_Loop_3_ncg)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Gop_4_ocqU(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_witeStsc4)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_witeStsc4)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_witeStsc4_U(start_for_witeStsc4)' using S	[KIMS 210 203] implementing fire p_sit_v_v_0(life_wo_dz_w) using shire Registers.
RTMG 210-285] Implementing FIFO 'start_for_xFSobelfYi_U(start_for_xFSobelfYi)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Regi: RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubC_U(start_for_magnitubD)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_MagnitubM_U(start_for_magnitubD)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_CUU(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_CUU(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_CUU(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_CUU(start_for_Loop_2_ccU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_CUU(start_for_Loop_2_ccU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_CUU(start_for_witeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_witeStsc4_U(star	<pre>[RTMG 210-285] Implementing FIFO 'p_dstx_V_V_U(fifo_w16_d2_A)' using Shift Registers.</pre>
<pre>RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Register: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Register: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubC U(start_for_magnitubK)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubC U(start_for_magnitubK)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_MatTolbW_U(start_for_XfMatTolbW)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_3_ncg)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocq_U(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_Loop_4_ocq)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qck_U(start_for_Loop_1_qck)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qck_U(start_for_Loop_1_qck)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_writeStsc4)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using</pre>	[RTMG 210-285] Implementing FIFO 'p_dsty_V_V_U(fifo_w16_d2_A)' using Shift Registers.
RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Register: RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Regis RTMG 210-285] Implementing FIFO 'p_srcl_V.V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_srcl_V.V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_src2_V.V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_src2_V.V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V.V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitublc_U(start_for_magnitubbC)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_magnitubLM_U(start_for_magnitubbM)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_4_ocq)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_witeStsc4)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_witeStsc4_U(start_for_witeStsc4)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_witeStsc4_U(start_for_witeStsc4	<pre>[RTMG 210-285] Implementing FIFO 'start_for_xFSobelfYi_U(start_for_xFSobelfYi)' using Shift Regi:</pre>
RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_XfMatToibs_U(start_for_XfMatToibs)' using Shift Regi: RTMG 210-285] Implementing FIFO 'p_srcl_V_U_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_src2_V_U_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_U_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_U_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubc()(start_for_magnitubb)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_magnitubM_U(start_for_magnitubM)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocgU(start_for_Loop_4_ocg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcAU(start_for_GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcKU(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_f	<pre>[RTMG 210-285] Implementing FIFO 'start_for_Sobel_Lg8j_U(start_for_Sobel_Lg8j)' using Shift Regi:</pre>
RTMG 210-285] Implementing FIFO 'start_for_XfMatToibs_U(start_for_XfMatToibs)' using Shift Regi: RTMG 210-285] Implementing FIFO 'p_srcl_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_src2_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitublc(using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitubb(using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_magnitubb(using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_magnitubb(using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO	<pre>[RTMG 210-285] Implementing FIFO 'start_for_Sobel_U0_U(start_for_Sobel_U0)' using Shift Register:</pre>
RTMG 210-285] Implementing FIFO 'p_srcl_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_src2_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitujbC_U(start_for_magnitujbC)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_magnitukbM_U(start_for_magnitukbM)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_GradMagpcA)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_writeStsc4)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regis RTMG 210-285] Implementing FIFO 's	<pre>[RTMG 210-285] Implementing FIFO 'start_for_XfMatTohbi_U(start_for_XfMatTohbi)' using Shift Regi:</pre>
RTMG 210-285] Implementing FIFO 'p_src2_V_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitujbC_U(start_for_magnitujbC)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_magnitukbM_U(start_for_magnitukbM)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_XfMatTolbW_U(start_for_XfMatTolbW)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_writeStsc4)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regis	[RTMG 210-285] Implementing FIFO 'start_for_XfMatToibs_U(start_for_XfMatToibs)' using Shift Regi:
RTMG 210-285] Implementing FIFO 'p_dst VU(fifo_w16_d2_A_x)' using Shift Registers. RTMG 210-285] Implementing FIFO 'start_for_magnitujbC_U(start_for_magnitujbC)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_magnitukbM_U(start_for_magnitukbM)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6U(start_for_SobelFimb6)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6U(start_for_SobelFimb6)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncgU(start_for_Loop_3_ncg)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_writeStsc4)' using Shift Regis RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regis	<pre>[RTMG 210-285] Implementing FIFO 'p_srcl_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers.</pre>
RTMG 210-285] Implementing FIFO 'start_for_magnitujbC_U(start_for_magnitujbC)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_magnitukbM_U(start_for_magnitukbM)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_XfMatTolbw_U(start_for_XfMatTolbw)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_3_ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocqU(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcUU(start_for_writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4U(start_for_writeStsc4)' using Shift Regi: RTM	<pre>[RTMG 210-285] Implementing FIFO 'p_src2_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers.</pre>
RTMG 210-285] Implementing FIFO 'start_for_magnitukbM_U(start_for_magnitukbM)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_XfMatTolbW_U(start_for_XfMatTolbW)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_3_ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocq] (start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4' using Shift Regi: RTMG	[RTMG 210-285] Implementing FIFO 'p_dst_V_V_U(fifo_wl6_d2_A_x)' using Shift Registers.
RTMG 210-285] Implementing FIFO 'start_for_XfMatTolbW_U(start_for_XfMatTolbW)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_3_ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocq_U(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi: ************************************	<pre>[RTMG 210-285] Implementing FIFO 'start_for_magnitujbC_U(start_for_magnitujbC)' using Shift Regi:</pre>
RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_3_ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocq_U(start_for_Loop_4_ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_1_qcK_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_1_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi:	[RTMG 210-285] Implementing FIFO 'start_for_magnitukbM_U(start_for_magnitukbM)' using Shift Regi:
RTMG 210-285] Implementing FIFO 'start for Loop 3 ncg U(start for Loop 3 ncg)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for Loop 4 ocq U(start for Loop 4 ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for GradMagpcA U(start for GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for Loop 1 qcK U(start for Loop 1 qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for Loop 2 rcU U(start for Loop 2 rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi:	[RTMG 210-285] Implementing FIFO 'start_for_XfMatTolbW_U(start_for_XfMatTolbW)' using Shift Regi:
RTMG 210-285] Implementing FIFO 'start for Loop 4 ocq U(start for Loop 4 ocq)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for GradMagpcA U(start for GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for Loop 1 qcK U(start for Loop 1 qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for Loop 2 rcU U(start for Loop 2 rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi:	<pre>[RTMG 210-285] Implementing FIFO 'start_for_SobelFimb6_U(start_for_SobelFimb6)' using Shift Regi:</pre>
RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_l_qcK_U(start_for_Loop_l_qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi:	[RTMG 210-285] Implementing FIFO 'start_for_Loop_3_ncg_U(start_for_Loop_3_ncg)' using Shift Regi:
RTMG 210-285] Implementing FIFO 'start for Loop 1 qcK U(start for Loop 1 qcK)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for Loop 2 rcU U(start for Loop 2 rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start for writeStsc4 U(start for writeStsc4)' using Shift Regi:	[RTMG 210-285] Implementing FIFO 'start_for_Loop_4_ocq_U(start_for_Loop_4_ocq)' using Shift Regi:
RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regi: RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi:	[RTMG 210-285] Implementing FIFO 'start_for_GradMagpcA_U(start_for_GradMagpcA)' using Shift Regi:
RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi:	[RTMG 210-285] Implementing FIFO 'start_for_Loop_l_qcK_U(start_for_Loop_l_qcK)' using Shift Regi:
· · · · · · · · · · · · · · · · · · ·	[RTMG 210-285] Implementing FIFO 'start_for_Loop_2_rcU_U(start_for_Loop_2_rcU)' using Shift Regi:
	[RTMG 210-285] Implementing FIFO 'start_for_writeStsc4_U(start_for_writeStsc4)' using Shift Regi
	< >
Cancel	Cancel

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.

	g (ns) ummary						
+			+	+		+	
				Estimated			
		-+-	+	+		+	
1	clk	T.	5.001	4 031		0.631	
lap				4.03		0.63	
ap + Laten * S	cy (cl	Loc 7: -+-	ck cycle	s): ++-	+	+	





== Utilization Es	ti	mates				
	==					
* Summary:						
+	-+	+	+-	+-	+	
Name	1	BRAM_18K	DSP48E	FF	LUT	
+	-+	+	+-	+-	+	
DSP	1	-1	-1	-1	-1	
Expression	1	-1	-1	0	12	
FIFO		01	-1	01	10	
Instance		12	-	2562	2994	
Memory	1	- 1	-1	-1	-1	
Multiplexer	1	- 1	-	-	-1	
Register	1	- 1	-1	-	-1	
+	-+	+	+-	+-	+	
Total	1	12	01	2562	3016	
+	-+	+	+-	+-	+	
Available	1	8901	840	407600	203800	
+	-+	+	+-	+-	+	
Utilization (%)	1	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.

	-						the second second second second second second second second		
Summary:									
	-+		+	+		-+		+-	
RTL Ports				Bits	Protocol	۱	Source Object	1	C Type
s axi AXILiteS AWVALID	-+	in	+	11	s axi	-	AXILiteS	1	return void
s axi AXILiteS AWREADY		out		11	s axi	1	AXILITES	2	return void
s axi AXILiteS AWADDR		in		41	s axi	1	AXILITES	2	return void
s axi AXILiteS WVALID		in		11	s axi	1	AXILiteS	2	return void
s axi AXILiteS WREADY	- 1	out		11	s axi	1	AXILITES	2	return void
s axi AXILiteS WDATA	- 1	in		321	s axi	1	AXILITES	2	return void
s axi AXILiteS WSTRB	- 1	in		41	s axi	1	AXILITES	1	return void
s axi AXILiteS ARVALID		in		11	s axi	1	AXILITES	1	return void
s axi AXILITES ARREADY		out		11	s axi	1	AXILiteS	4	return void
s axi AXILITES ARADDR		in		41		1	AXILITES	1	return void
		out		11	s_axi s axi	1	AXILITES	1	return void
s_axi_AXILiteS_RVALID s_axi_AXILiteS_RREADY		in		11	s_axi	1	AXILITES	1	return void
s axi AXILITES RREADY		out		321	s_axi	1	AXILITES		return void
		out		21		1	AXILITES	4	return void
s_axi_AXILiteS_RRESP	1	out			s_axi	1	AXILITES	1	return void
s_axi_AXILiteS_BVALID			1	11	s_axi	1		5	
s_axi_AXILiteS_BREADY	1	in	1	11	s_axi	1	AXILiteS	1	return void
s_axi_AXILiteS_BRESP	1	out	1	21	s_axi	1	AXILiteS	1	return void
ap_clk	1	in	1	11	ap_ctrl_hs	1	Edge_Detection	1	return value
lap_rst_n	1	in	1	11	ap_ctrl_hs	1	Edge_Detection	1	return value
linterrupt	1	out	1	11	ap_ctrl_hs	1	Edge_Detection	1	return value
Y TDATA	1	in	1	161	axis	1	image in V data V	1	pointer
Y TREEP	i.	in	î.	21	axis	1	image in V keep V	1	pointer
Y TSTRB	i	in	i	21	axis	i	image in V strb V	i.	pointer
TUSER	i	in	i	11	axis	i	image in V user V	i.	pointer
Y TLAST	i	in	î	11	axis	ì	image in V last V	i.	pointer
YTID	1	in	1	11	axis	i	image in V id V	i.	pointer
Y TDEST	1	in	1	11	axis	i	image in V dest V	1	pointer
Y TVALID	1	in	1	11	axis	i	image in V dest V	1	pointer
Y TREADY	1	out	1	11	axis	1	image in V dest V	1	pointer
Y Out TDATA	1	out	1	161	axis	i.	image out V data V	1	pointer
Y OUT THEEP	1	out	1	21	axis	1	image out V keep V	1	pointer
Y_OUT_TSTRB	1	out	1	21	axis	1	image out V strb V	I.	pointer
Y_OUT_TUSER	1	out	1	11	axis	1	image_out_V_user_V	1	pointer
Y_OUT_TLAST	1	out	1	11	axis	1	image_out_V_last_V	I.	pointer
Y_OUT_TID	1	out	1	11	axis	i.	image_out_V_id_V	I.	pointer
Y_OUT_TDEST	1	out	1	11	axis	I.	image_out_V_dest_V	L	pointer
Y_OUT_TVALID	1	out	1	11	axis	I.	image_out_V_dest_V	Ľ.	pointer
Y OUT TREADY	1	in		11	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.

roject Settings General	IP > Repository Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.	2
Simulation Elaboration	IP Repositories	
Synthesis Implementation Bitstream	+ - + - + + + /codege_Detection_prj/solution1/impl/ip (Project)	
P IP		
Repository Packager	Add Repository	
XHub Store		
Source File Display WebTalk Help Text Editor	Repository ▼ /codegen_IP/Edge_Detection_prj/solution1/impl/ip ∨ /codegen_IP/Edge_Detection_prj/solution1/impl/ip ∨ IPs (1)	
Display WebTalk	★ ◆ ✓ /codegen_IP/Edge_Detection_prj/solution1/impl/ip	

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.



Project Summary × IP Catalog ×							
Cores Interfaces							
Search: Q-							
Name	^1 AX	14 Status	License	VLNV			
✓ □ User Repository (c:/Lab5/codegen_IP/E	dge_Dete	ction_prj/so	lution1/imp	l/ip)			
🗸 🕒 VITIS HLS IP							
Edge_detection	AX	14, / Pre-Pro	Included	xilinx.com:hls:Edge_Detection:1.0			
> 📄 Vivado Repository							

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.





Generated code in directory \codegen_edge_detection'

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.

Block Parameters: Model Composer Hub			
Model Composer Hu	ıb		
The second se	ration' tab t		system in the model. ode, create testbench,
Code Generation	Hardware	Feedback	
Subsystem name:			
Edge Detection			
Code directory:			
./codegen_edge_detection2 Browse.			Browse
Target: HLS C++ code Settings.			Settings
✓ Create and run te	stbench		
Testbench stack size (MBytes) 10			
Generate and Run			
	OK	Cancel	Help Apply

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.





Simulink Library Browser × - 12 - 2 - 2 2 🗢 🗇 Enter search term Xilinx Toolbox/AI Engine Additional Math & Discrete Quick Insert Communications Toolbox Communications Toolbox HDL Supp Computer Vision Toolbox Control System Toolbox DSP Interfaces DSP System Toolbox DSP System Toolbox HDL Support HDL Coder Image Acquisition Toolbox **Report Generator** Simulink 3D Animation Simulink Coder User-Defined Functions Tools Simulink Extras Stateflow Xilinx Toolbox AL Er DSP Interfaces Tools **User-Defined Functions** HDL HLS conthu llood n.

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 <code>import_kernel.slx</code> file.



AIE Kernel

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




Block Parameters: AIE Kernel ×
AIE Kernel
Imports AIE kernel function as a block.
General Constraints
Parameters
Kernel header file:
Kernel function: KernelFunction
Kernel init function:
Kernel source file:
Kernel search paths: {} Add
Preprocessor options: {}
Import
<u>O</u> K <u>C</u> ancel <u>H</u> elp <u>Apply</u>

- 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.





	unction (inc	luding function template)				
unction	General	Constraints				
nction do	claration					
netion de						
	_27t_sym_h	b_2i(input_window_cint16 * restrict c	b_input, output_window	_cint16 * restrict cb_out	put)	
	_27t_sym_h	b_2i(input_window_cint16 * restrict c	b_input, output_window:	_cint16 * restrict cb_out	put)	
		b_2i(input_window_cint16 * restrict c	b_input, output_window:	/_cint16 * restrict cb_out	put)	
void fir		b_2i(input_window_cint16 * restrict c	b_input, output_window	/_cint16 * restrict cb_out	put)	
void fir		b_2i(input_window_cint16 * restrict c	b_input, output_window Window size (samples)	v_cint16 * restrict cb_out Window margin (samples)	put) Synchronicity	Signal size (samples)
void fir rt attribut	es		Window size	Window margin		Signal size (samples)

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

Direction	Name	Туре	Window size (samples)	Window margin (samples)	Synchronicity	Signal size (samples)
Input	cb_input	input_window_cint	128	16	sync	
Output	cb output	output_window_cin	256		sync	

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.

Port attributes

Direction	Name	Туре	Window size (samples)	Window margin (samples)	Synchronicity	Signal size (samples)
Input	in	input_stream_cint				
Output	out	output_stream_cin				256

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.

- 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

Send Feedback



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

rt attribut	es				
Direction	Name	Туре	Window size (samples)	Window margin (samples)	Synchronicity
Input	cb_input	input_window_cint	256	32	sync
Output	cb_output	output_window_ci	128		sync

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 <code>aie_system</code>.
- 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.





Block Parameters: Model Compose	er Hub 🔶 🙃 兴
Model Composer Hub	
Controls implementation of the specified su model. Use the 'Code Generation' tab to ge testbench, and run the generated code.	2
Code Generation Hardware Feedbac	k
Subsystem name:	
aie_system	
Code directory:	
./code	Browse
Target: Al Engines 🔹	Settings
Compiler options {}	:
✔ Create testbench	
✓ Run AIE Simulation	
Simulation timeout (cycles) 1500000	:
Collect profiling statistics and enable 'prin	ntf' for debugging
Collect data for Vitis Analyzer	Open Vitis Analyzer
Collect data for Vitis Analyzer	
Collect data for Vitis Analyzer	hroughput

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.

Progress	Ŷ	•		×
Done verification				

XMC_AIE_INFO: Verifying aiesimulator output **********************************				
/check_output data/aiesimulator_output/Out1.diff data/aiesimulator_output/Out1.t	xt.mo	od d	lata	
Comparing simulation results Output data file : data/aiesimulator_output/Out1.txt.mod Reference data file : data/reference_output/Out1.txt Simulation results MATCH. ************************************				
Test PASSED Verification Complete make: Leaving directory '//AIEngine	e_Lił	orar	y/L	
INFO: Generated code in directory '/	/AII	Eng	ine_ ▶	-
	C	k		

- 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.

AIE Graph Xilinx Toolbox/AI Engine/User-Defined Functions	-
AIE Kernel Villey Teolboy/AL Engine/Illeg Defined Exections	

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

Block Parameters: AIE Graph 🔹 🗉 🗙
AIE Graph
Imports AI Engine graph as a block.
General
Parameters
Graph file
O Header file (*.h)
Source file (*.cpp)
Graph source file(*.cpp):
Graph search paths: {} Add
Preprocessor options: {}
Import
<u>O</u> K <u>C</u> ancel <u>H</u> elp <u>A</u> pply

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.

e_system								
unction	General C	Constraints						
Direction	Name	Alias	Graph Port Name	Туре	Data Type	Size (samples)	Synchronicity	Signal size (samples)
Input	in1	platform.src[0]	In1	window	cint16	128	sync	
Output	out1	platform.sink[0]	Out1	window	cint16	128	sync	
(

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.

	Q model c	
Model Composer Hub Xilinx Toolbox/Al Engine/Tools		
Model Composer Hub Xilinx Toolbox/HLS/Tools		

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



Block Para	meters: Mode	el Composer	Hub	
Model Composer Hul	D			
Controls implement model. Use the 'Co testbench, and run	de Generatior	n' tab to gen	2	
Code Generation	Hardware	Feedback		
Subsystem name:				
graph_sub				
Code directory:				
./code			Browse	
Target: Al Engines		•	Settings.	
Compiler options {	[}			:
✔ Create testbench				
🖌 Run AIE Simulatio	on			
Simulation timed	out (cycles)	500000		:
Simulation timed	,		f' for debuggi	
	tatistics and e	enable 'print	f' for debuggin Open Vitis An	ng
Collect profiling s	itatistics and e	enable 'print	Open Vitis An	ng
Collect profiling s	itatistics and e	enable 'print	Open Vitis An	ng alyzer
Collect profiling s	itatistics and e	enable 'print	Open Vitis Ana roughput	ng alyzer

- 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.
 - G code G cod
- 6. Notice the log in the Progress window after completion. Click **OK**.





•	Progress			
	Done ver	ification		
****	*******	****		
XMC_AIE_INFO:	Verifying aiesimulator output	*******		
./check_output data	/aiesimulator_output/Out1.dif *******************************	ff data/aiesimulator_output/(*********	Out1.txt.mod	data
	ta/aiesimulator_output/Out1. : data/reference_output/Out1.			
Test PASSED Verification Compl make: Leaving dire				ų
INFO: Generated c	ode in directory '/			+
			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





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:

Send Feedback



1. Vitis Model Composer User Guide (UG1483)

Please Read: Important Legal Notices

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

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.





Copyright

© Copyright 2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. All other trademarks are the property of their respective owners.

