HDL Library

Lab 1: Introduction to Vitis Model Composer HDL Library

In this lab, you will learn how to use the Vitis™ 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 Vitis 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 Vitis 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 Vitis 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 2022.1**.
- On Linux systems, type `model_composer` at the command prompt.

2. Navigate to the Lab1 folder: `\HDL_Library\Lab1`.

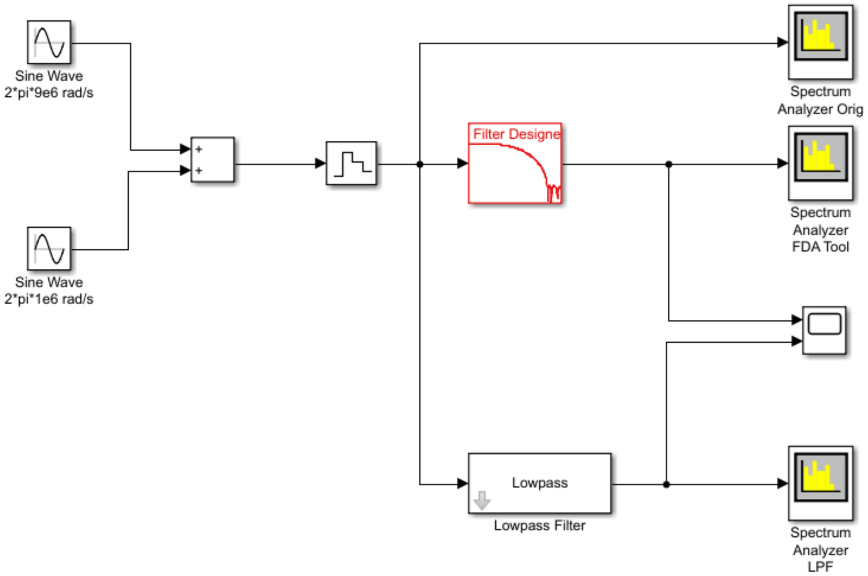
   You can view the directory contents in the MATLAB® Current Folder browser, or type `ls` at the command line prompt.

3. Open the Lab1_1 design as follows:

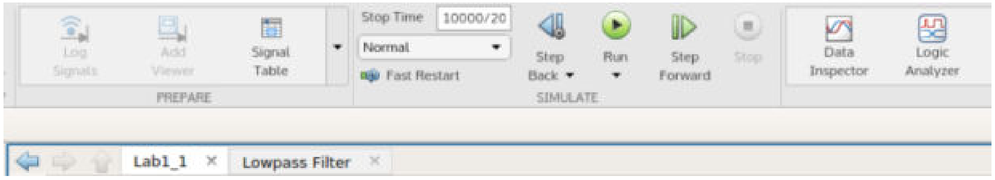
   a. At the MATLAB command prompt, type `open Lab1_1.slx` OR

   b. Double-click `Lab1_1.slx` in the Current Folder browser.

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



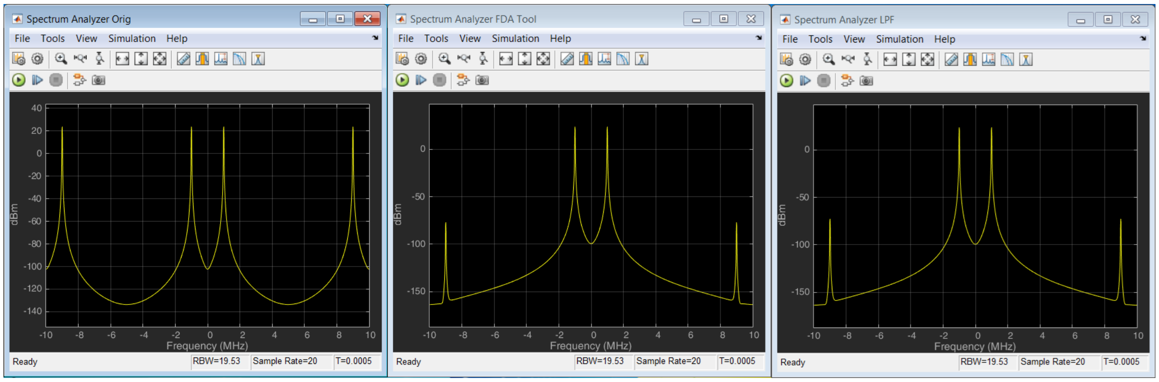
4. From your Simulink project worksheet, select **Simulation → Run** or click the **Run** simulation button.



Stop Time 10000/20

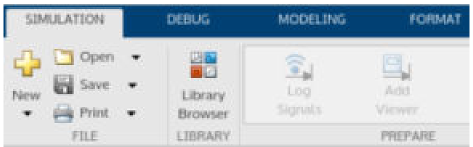
Normal

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.

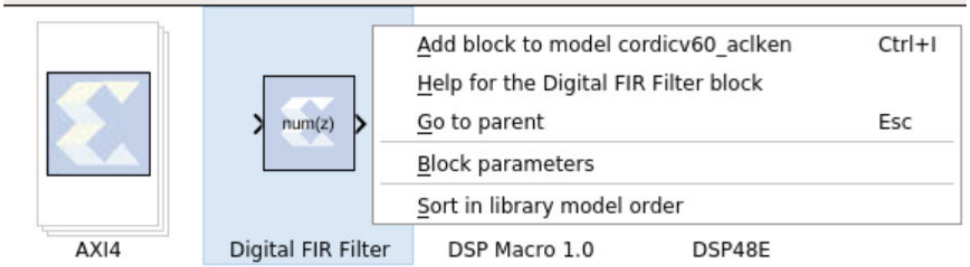
5. Click the **Library Browser** button in the Simulink toolbar to open the Simulink Library Browser.



When using Vitis 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 Vitis Model Composer in the *Vitis Model Composer User Guide* (UG1483).

6. Expand the **Xilinx Toolbox→HDL** menu, select **DSP**, then select **Digital FIR Filter** from **Non AXI-S**.

7. Right-click the **Digital FIR Filter** block and select **Add block to model Lab1_1**.


Add block to model cordicv60_aclken Ctrl+I
Help for the Digital FIR Filter block
Go to parent Esc
Block parameters
Sort in library model order
num(z)
AXI4
Digital FIR Filter
DSP Macro 1.0
DSP48E

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

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

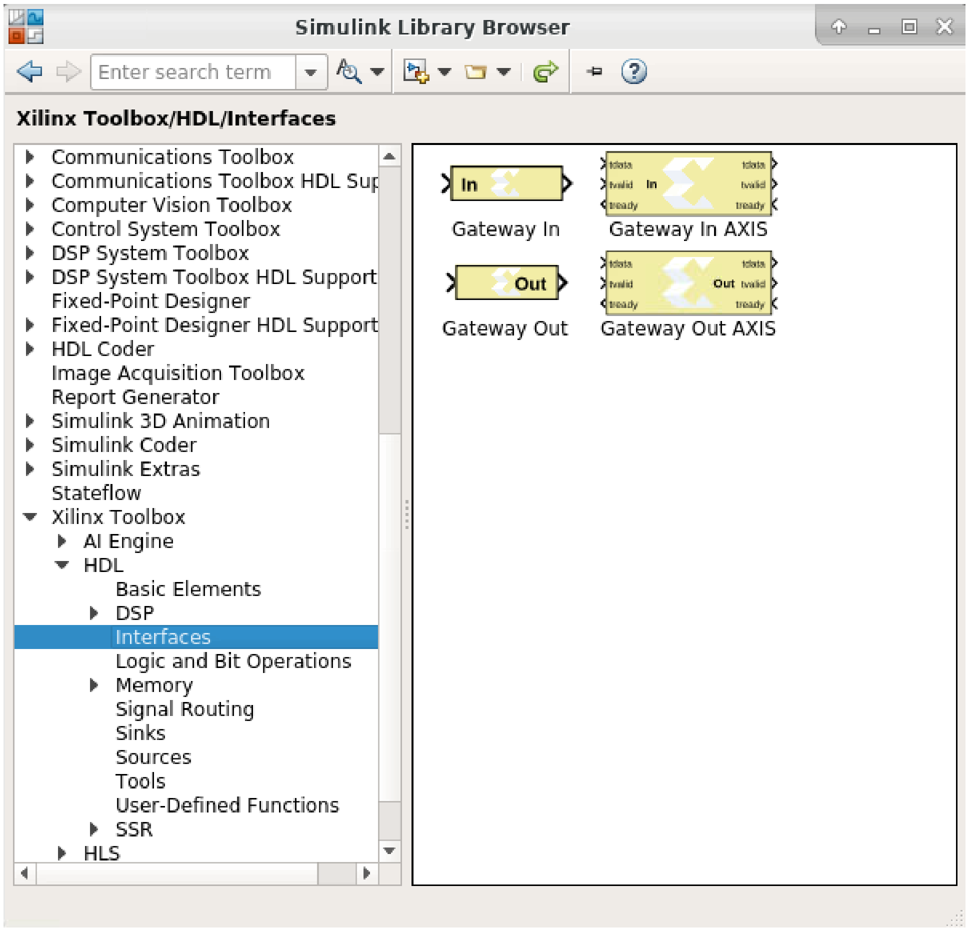
   An FPGA design requires three important aspects to be defined:

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

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

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

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

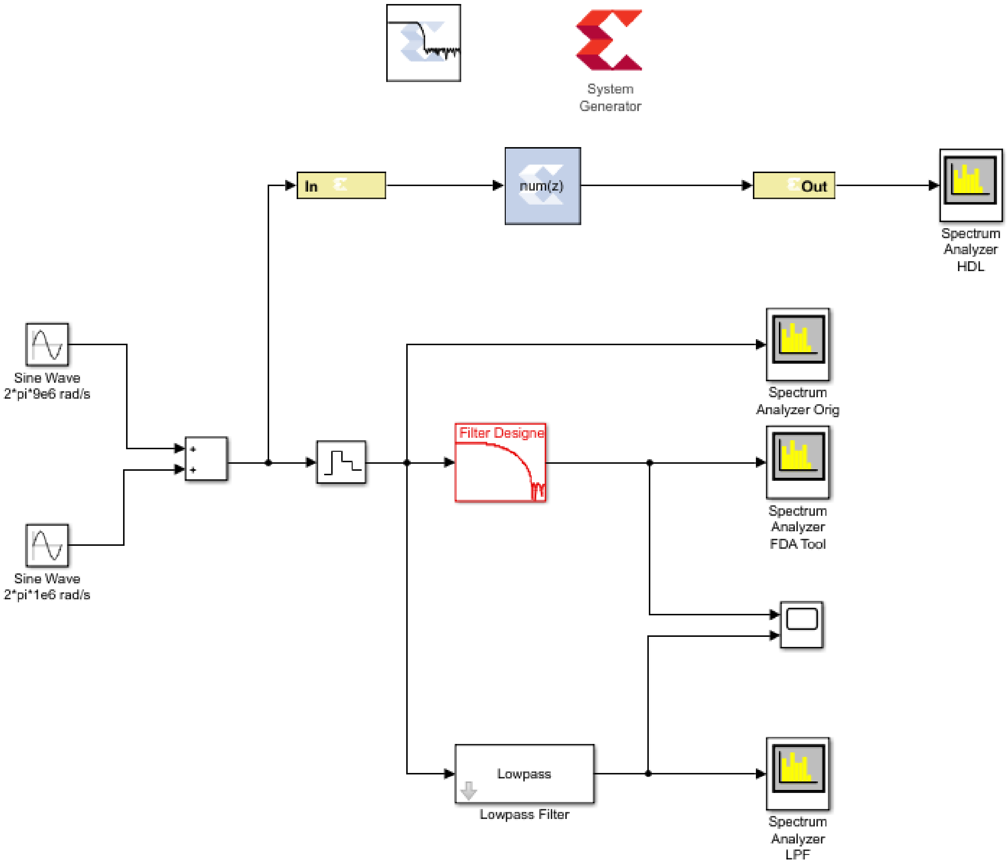


10. Similarly, from the same menu, add a Gateway Out block to the design.

11. From the Tools menu, under the HDL menu, add the System Generator token used to define the FPGA technology.

12. Finally, make a copy of one of the existing Spectrum Analyzer blocks, and rename the instance to Spectrum Analyzer HDL by clicking the instance name label and editing the text.

13. Connect the blocks as shown in the following figure. Use the left-mouse key to make connections between ports and nets.



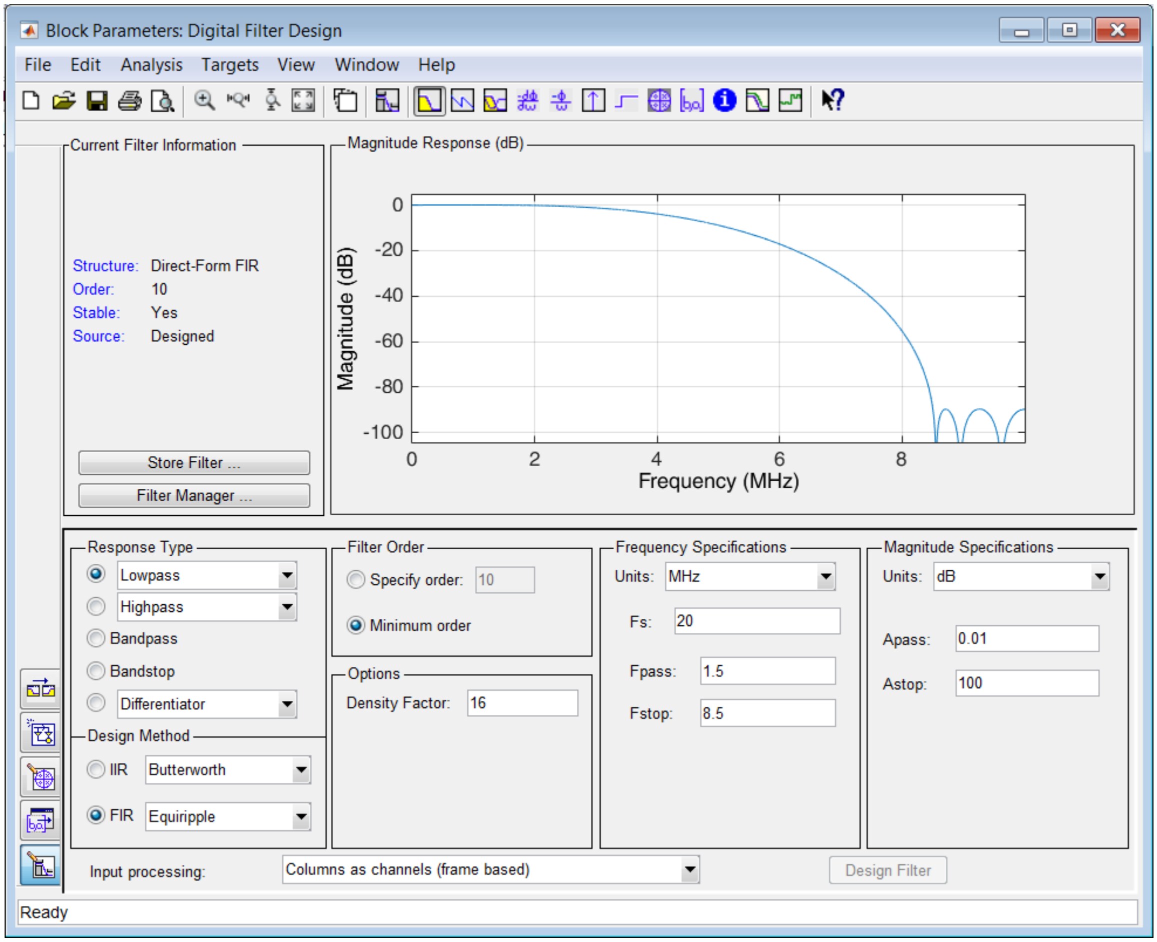
The next part of the design process is to configure the HDL blocks.

Configure the HDL Blocks

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

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

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



Current Filter Information

Structure: Direct-Form FIR
Order: 10
Stable: Yes
Source: Designed

Response Type

● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method

○ IIR Butterworth
● FIR Equiripple

Filter Order

○ Specify order: 10
● Minimum order

Options

Density Factor: 16

Frequency Specifications

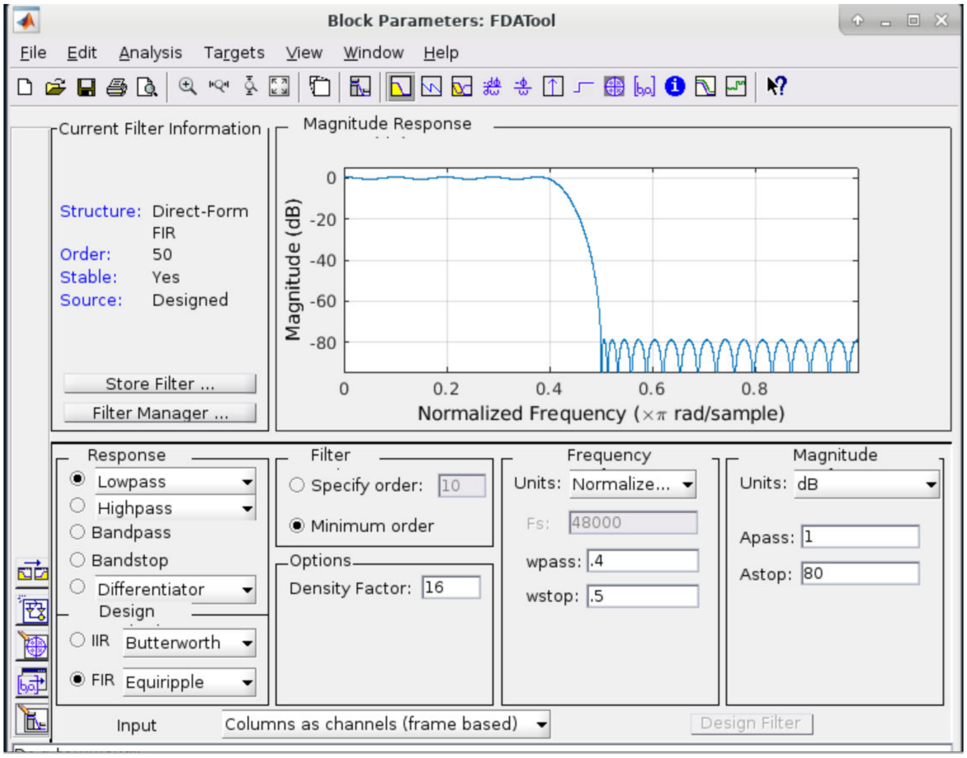
Units: MHz
Fs: 20
Fpass: 1.5
Fstop: 8.5

Magnitude Specifications

Units: dB
Apass: 0.01
Astop: 100

Input processing: Columns as channels (frame based)

2. Close the Properties Editor for the Digital Filter Design instance.
3. Double-click the **FDATool** instance to open the Properties Editor.



Current Filter Information

Structure: Direct-Form FIR
Order: 50
Stable: Yes
Source: Designed

Response
- ● Lowpass
- ○ Highpass
- ○ Bandpass
- ○ Bandstop
- ○ Differentiator

Design
- ○ IIR Butterworth
- ● FIR Equiripple

Filter
- ○ Specify order: 10
- ● Minimum order

Options
Density Factor: 16

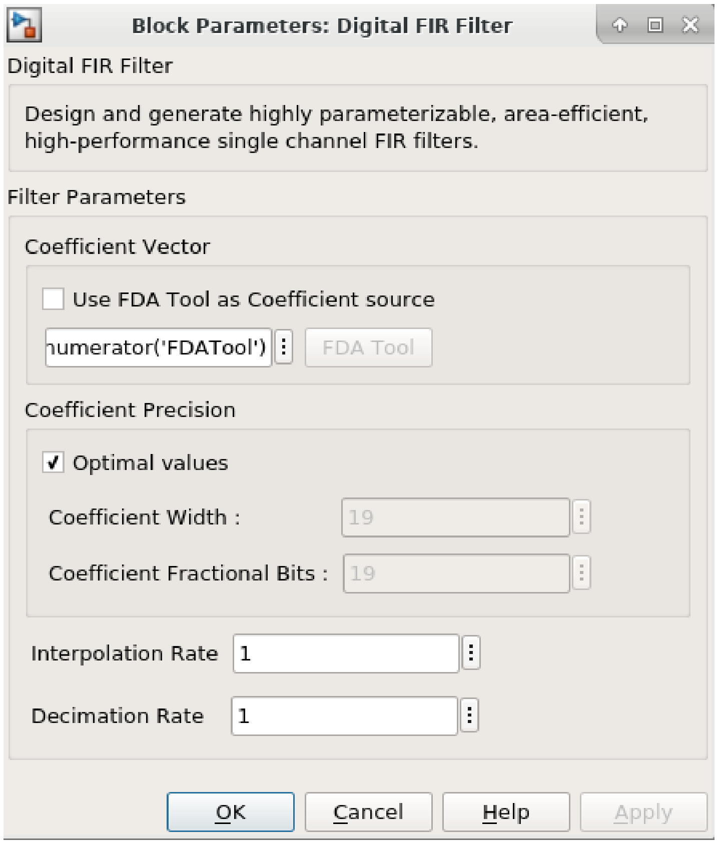
Frequency
Units: Normalize...
Fs: 48000
wpass: .4
wstop: .5

Magnitude
Units: dB
Apass: 1
Astop: 80

Input: Columns as channels (frame based)

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.



Block Parameters: Digital FIR Filter

Digital FIR Filter

Design and generate highly parameterizable, area-efficient, high-performance single channel FIR filters.

Filter Parameters

Coefficient Vector

☐ Use FDA Tool as Coefficient source

numerator('FDATool') FDA Tool

Coefficient Precision

☑ Optimal values

Coefficient Width : 19

Coefficient Fractional Bits : 19

Interpolation Rate 1

Decimation Rate 1

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 Vitis 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).

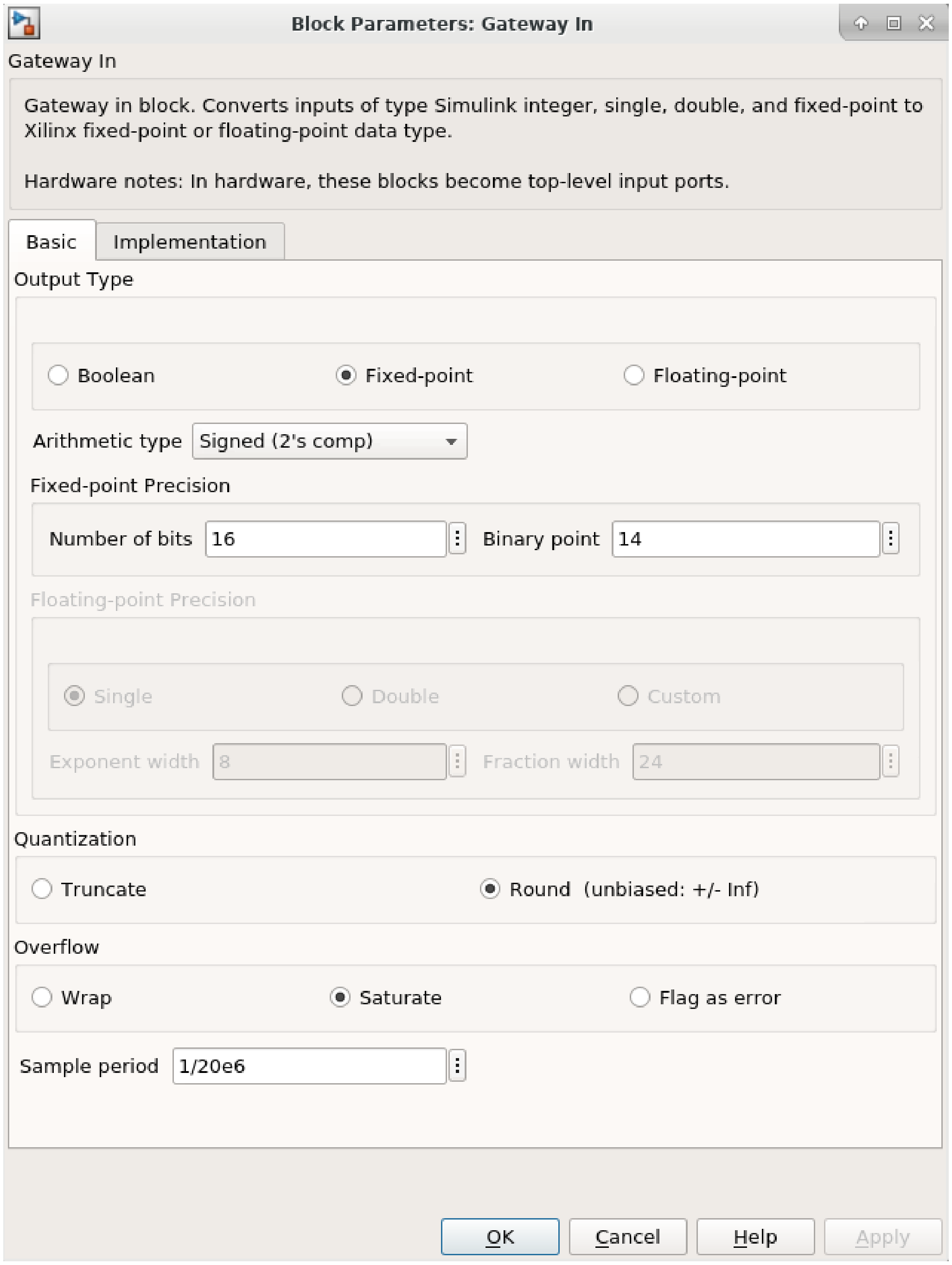
   Both of these attributes (rate and bit width) determine the degree of accuracy with which the continuous time signal is represented. Both of these attributes also have an impact on the size, performance, and hence cost of the final hardware.

   Vitis 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).



Block Parameters: Gateway In

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

Floating-point Precision

● Single ○ Double ○ Custom

Exponent width: 8 Fraction width: 24

Quantization

○ Truncate ● Round (unbiased: +/- Inf)

Overflow

○ Wrap ● Saturate ○ Flag as error

Sample period: 1/20e6

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.

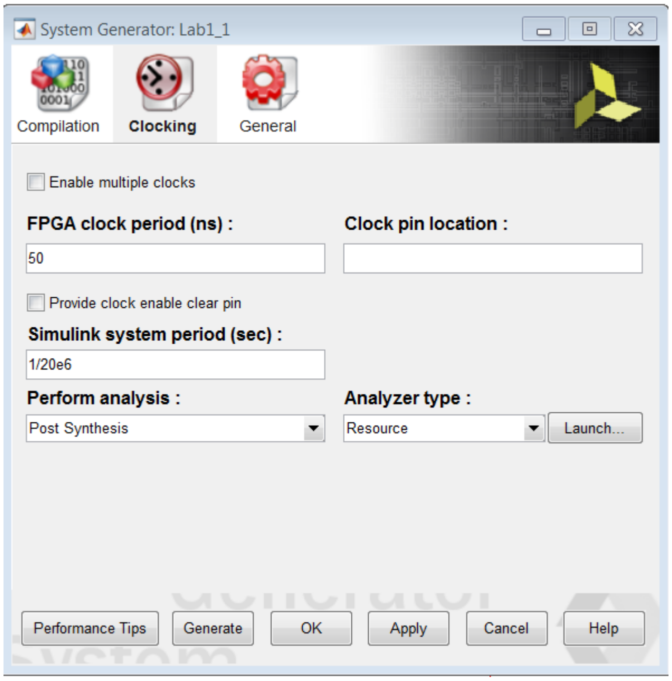
    Because the input port is sampled at 20 MHz to adequately represent the data, you must define the clock rate of the FPGA and the Simulink sample period to be at least 20 MHz.

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.



☐ Enable multiple clocks

FPGA clock period (ns) : 50

Clock pin location :

☐ Provide clock enable clear pin

Simulink system period (sec) : 1/20e6

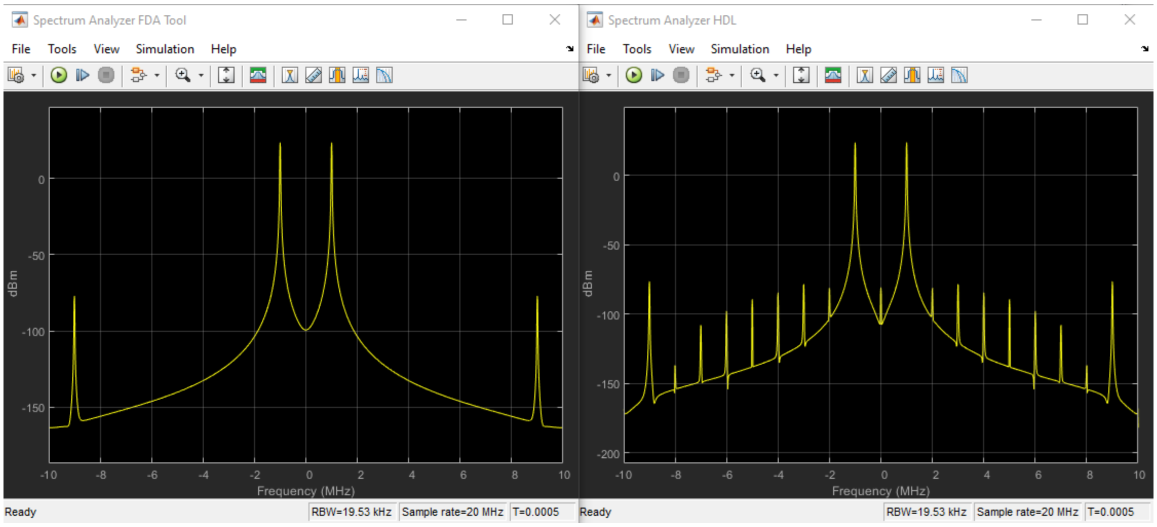
Perform analysis : Post Synthesis

Analyzer type : Resource

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.



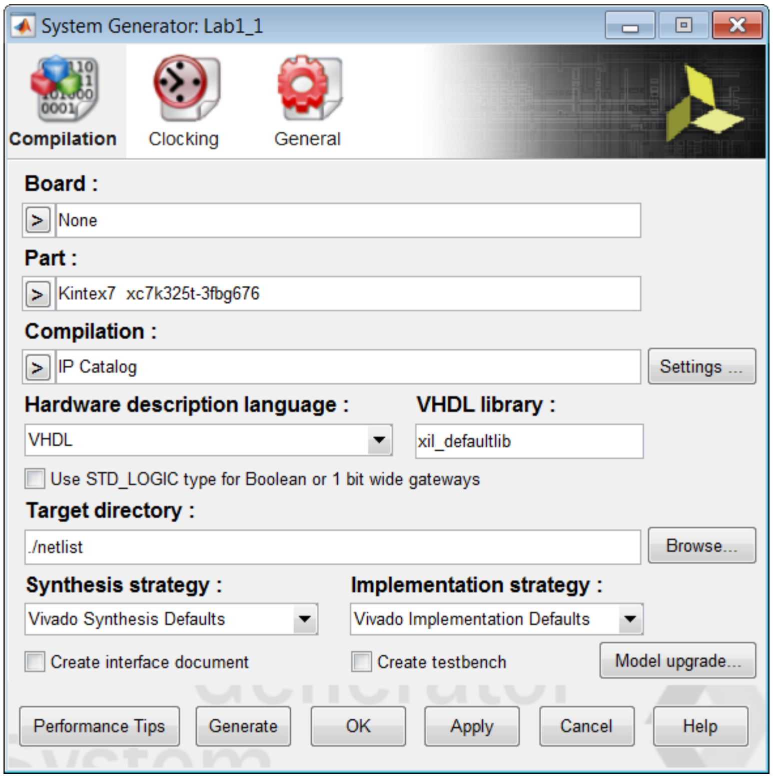
Ready RBW=19.53 kHz Sample rate=20 MHz T=0.0005

Ready RBW=19.53 kHz Sample rate=20 MHz T=0.0005

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.



Board : None

Part : Kintex7 xc7k325t-3fbg676

Compilation : IP Catalog

Hardware description language : VHDL

VHDL library : xil_defaultlib

□ Use STD_LOGIC type for Boolean or 1 bit wide gateways

Target directory : ./netlist

Synthesis strategy : Vivado Synthesis Defaults

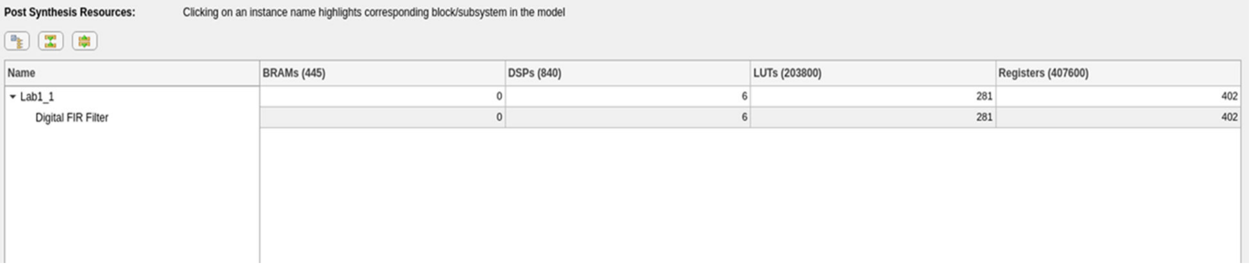
Implementation strategy : Vivado Implementation Defaults

□ Create interface document

□ Create testbench

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

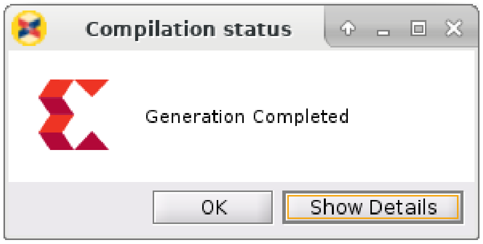
    The compilation process transforms the design captured in Simulink blocks into an industry standard Register Transfer Level (RTL) design description. The RTL design can be synthesized into a hardware design. A Resource Analyzer window appears when the hardware design description has been generated.

    Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▾ Lab1_1	0	6	281	402
Digital FIR Filter	0	6	281	402

    The Compilation status dialog box also appears.

    Generation Completed

21. Click **OK** to dismiss the Compilation status dialog box.

22. Click **OK** to dismiss the Resource Analyzer window.

23. Click **OK** to dismiss the System Generator token.

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

Review the Results

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

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

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

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

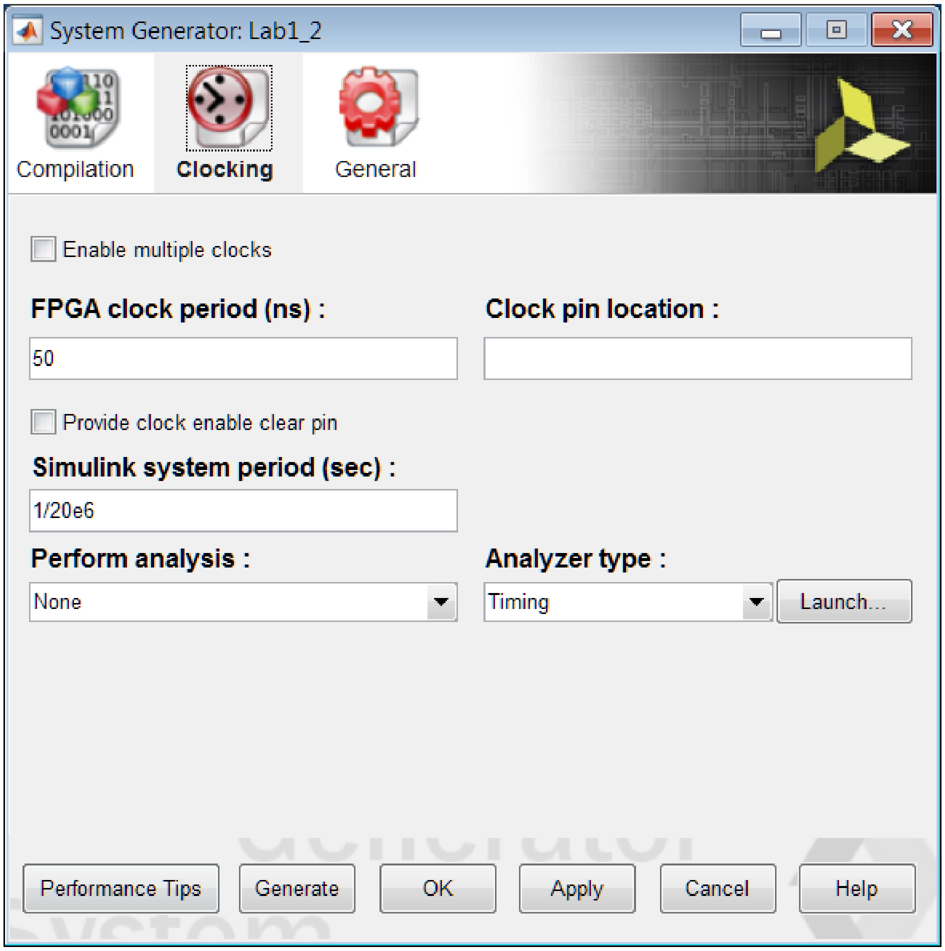
When you have reviewed the results, exit the `Lab1_1.slx` Simulink worksheet.

Step 2: Creating an Optimized Design in an FPGA

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

1. At the command prompt, type `open Lab1_2.slx`.
2. From your Simulink project worksheet, select **Simulation → Run** or click the Run simulation button ▶ to confirm this is the same design used in Step 1: Creating a Design in an FPGA.
3. Double-click the System Generator token to open the Properties Editor.

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



□ Enable multiple clocks

FPGA clock period (ns) : 50

Clock pin location :

□ Provide clock enable clear pin

Simulink system period (sec) : 1/20e6

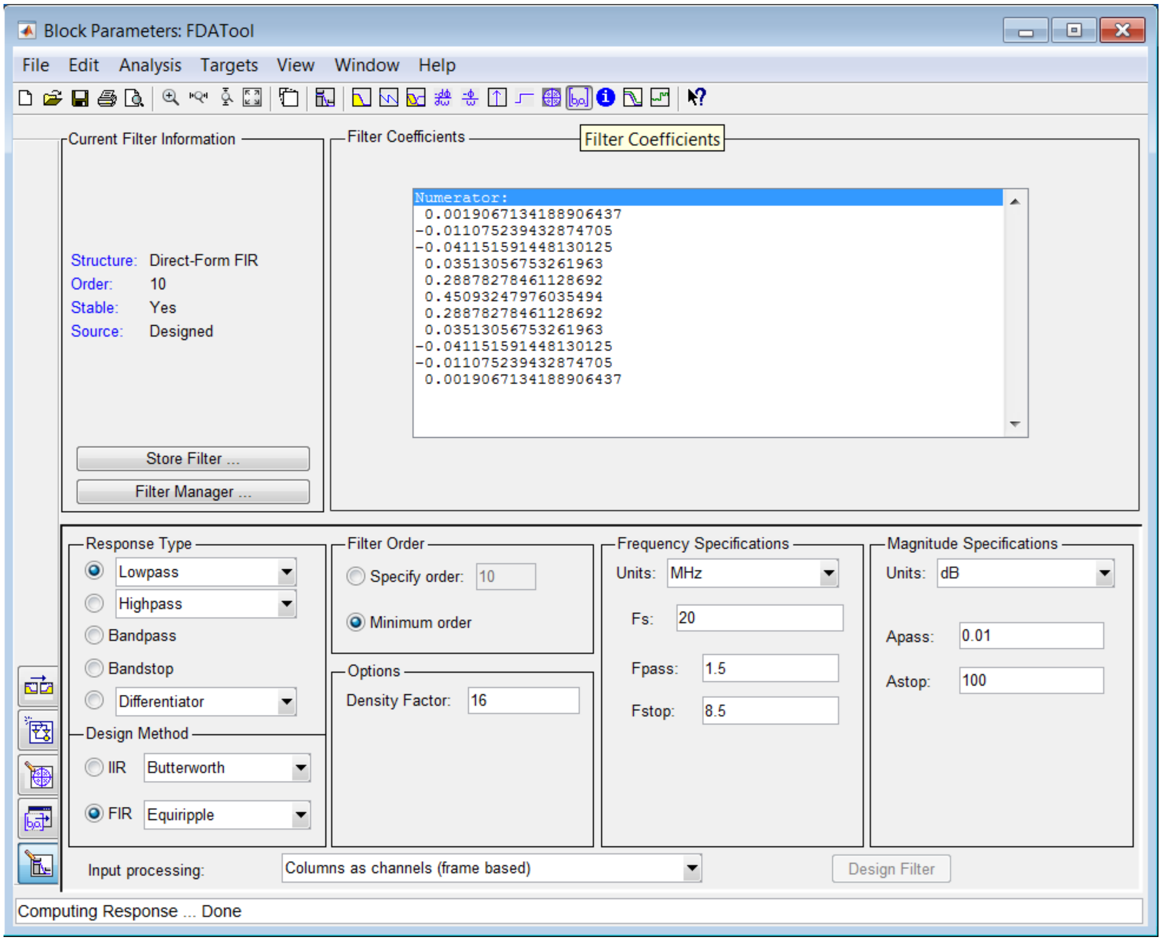
Perform analysis : None

Analyzer type : Timing

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 Vitis 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 [b,a] to view the filter coefficients.


Block Parameters: FDATool
File Edit Analysis Targets View Window Help
Filter Coefficients
Store Filter ...
Filter Manager ...
Design Filter

Current Filter Information

Structure: Direct-Form FIR
Order: 10
Stable: Yes
Source: Designed

Filter Coefficients

```
Numerator:
 0.0019067134188906437
-0.011075239432874705
-0.041151591448130125
 0.03513056753261963
 0.28878278461128692
 0.45093247976035494
 0.28878278461128692
 0.03513056753261963
-0.041151591448130125
-0.011075239432874705
 0.0019067134188906437
```

Response Type
- ● Lowpass
- ○ Highpass
- ○ Bandpass
- ○ Bandstop
- ○ Differentiator

Design Method
- ○ IIR Butterworth
- ● FIR Equiripple

Filter Order
- ○ Specify order: 10
- ● Minimum order

Options
Density Factor: 16

Frequency Specifications
Units: MHz
Fs: 20
Fpass: 1.5
Fstop: 8.5

Magnitude Specifications
Units: dB
Apass: 0.01
Astop: 100

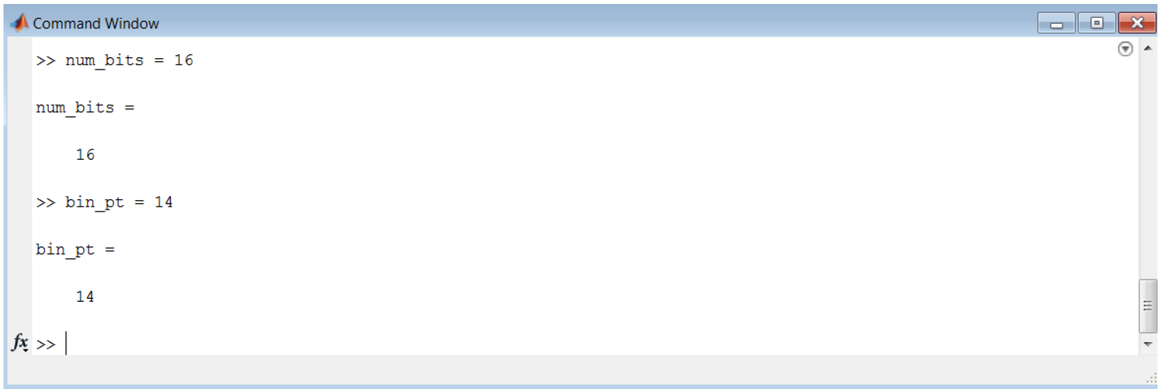
Input processing: Columns as channels (frame based)

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.

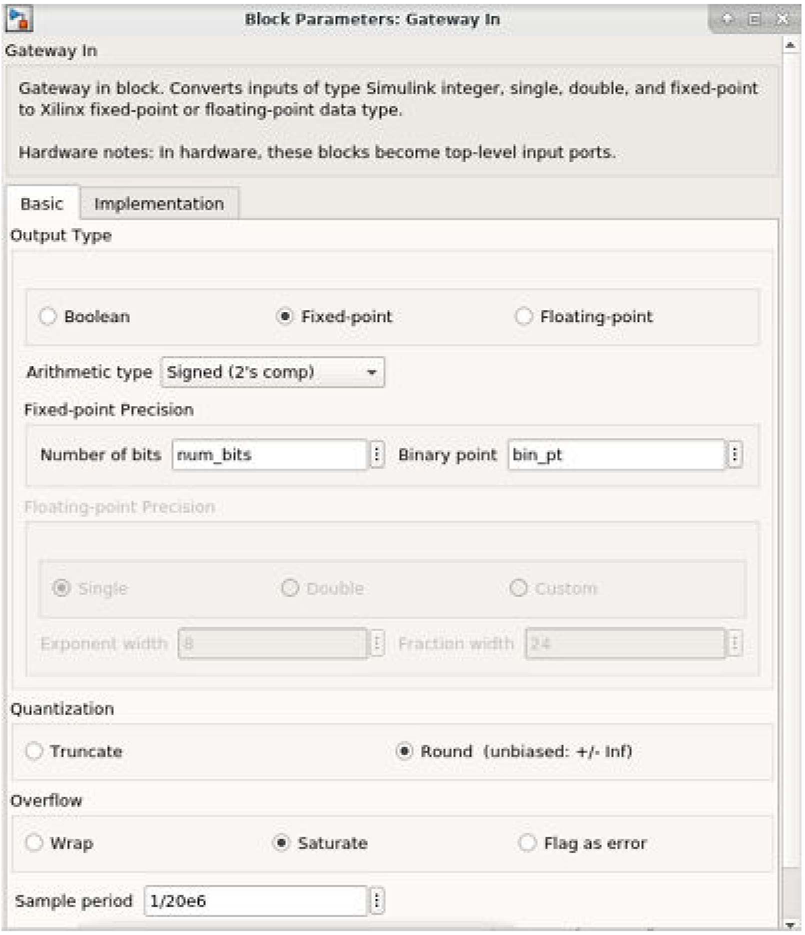
The current design samples the input at a rate of 20 MHz. If the input is sampled at 6 times the current frequency, it is possible to perform all calculations using a single multiplier.

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`



```
>> 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 `num_bits` and replace 14 with `bin_pt`, as shown in the following figure.



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.

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 ● Round (unbiased: +/- Inf)

Overflow

○ Wrap ● Saturate ○ Flag as error

Sample period: 1/20e6

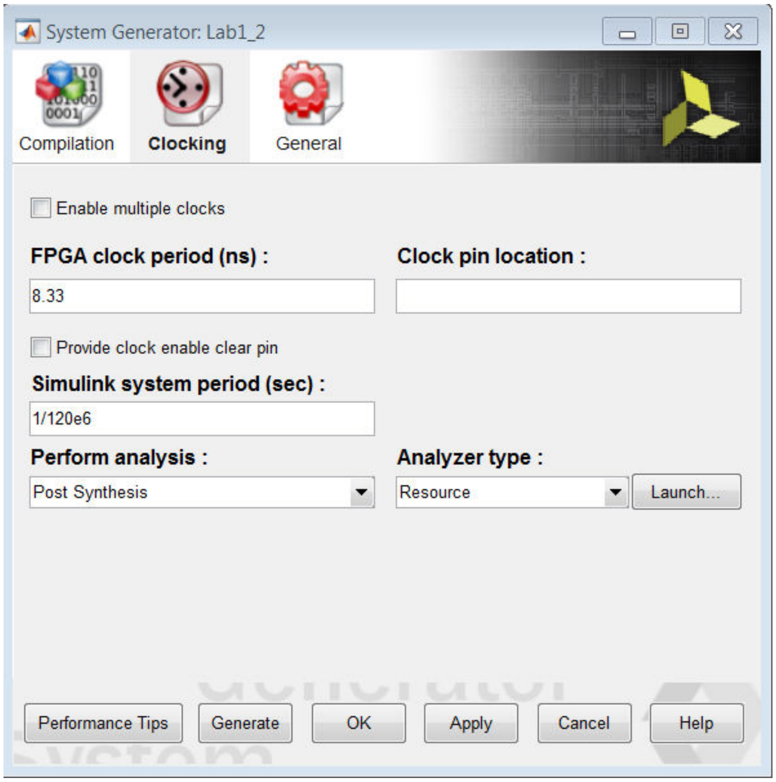
11. Click **OK** to save and exit the Properties Editor.

    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 the Resource Analyzer Window it is recommended to specify a new target directory rather than use the current working directory.



System Generator: Lab1_2

☐ Enable multiple clocks

FPGA clock period (ns) : 8.33

Clock pin location :

☐ Provide clock enable clear pin

Simulink system period (sec) : 1/120e6

Perform analysis : Post Synthesis

Analyzer type : Resource

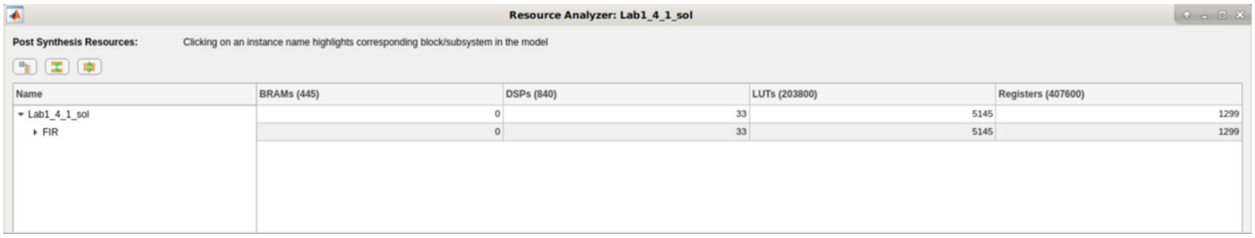
12. Click **Generate** to compile the design into a hardware description.

    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.



Resource Analyzer: Lab1_4_1_sol

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Lab1_4_1_sol	0	33	5145	1299
FIR	0	33	5145	1299

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

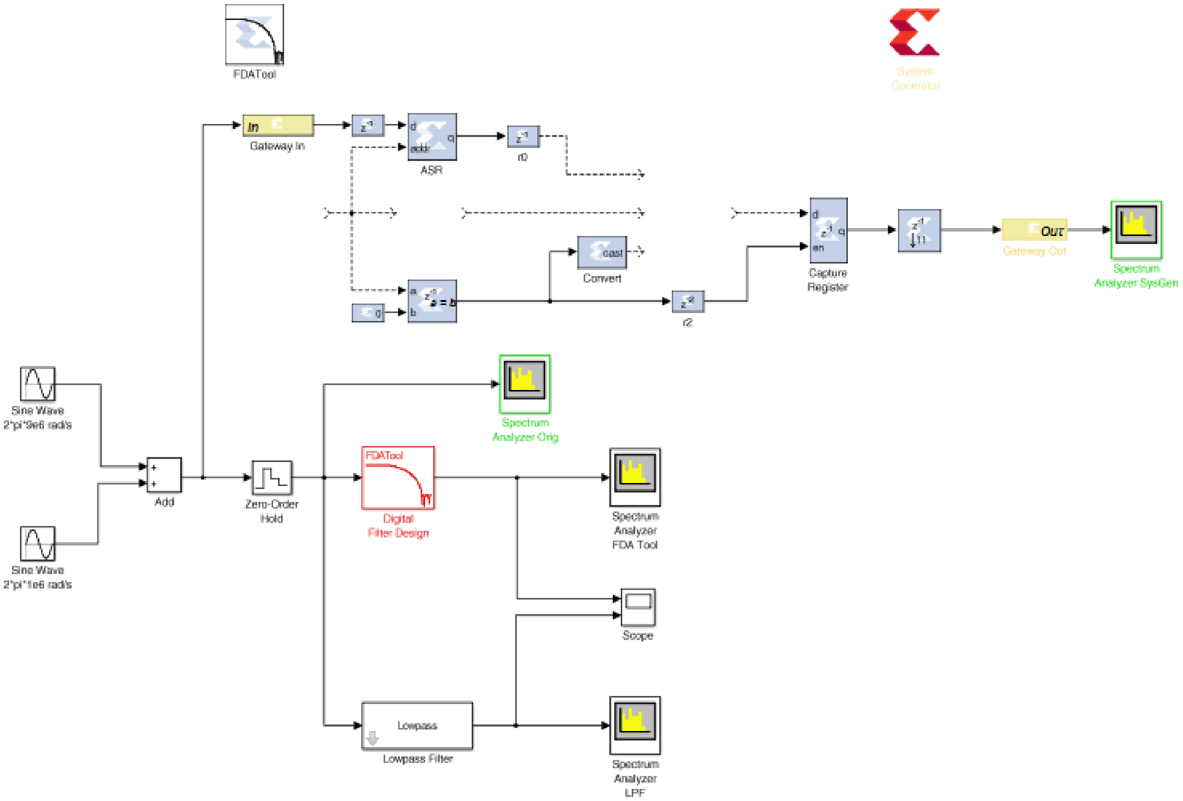
Exit the `Lab1_2.slx` Simulink worksheet.

Step 3: Creating a Design using Discrete Components

In this step you will see how Vitis 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:

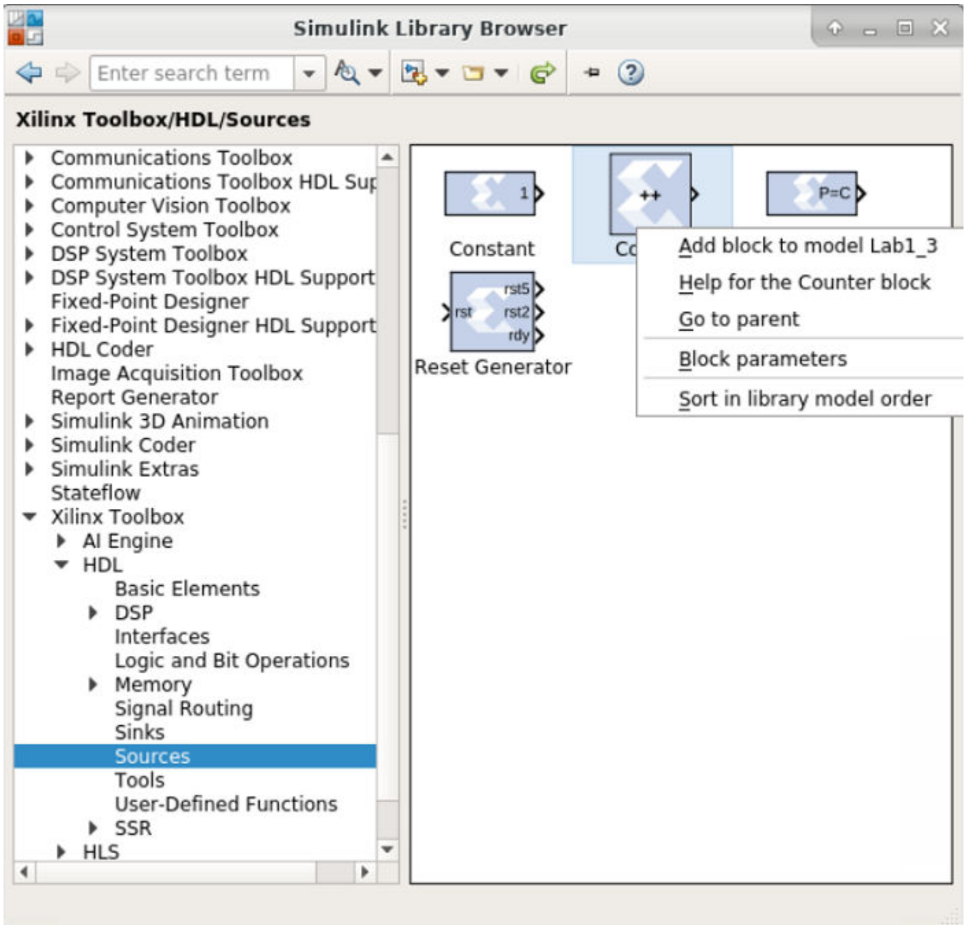
- Samples arrive through port In and after a delay stored in a shift register (instance ASR).
- A ROM is required for the filter coefficients.
- A counter is required to select both the data and coefficient samples for calculation.
- A multiply accumulate unit is required to perform the calculations.
- 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.

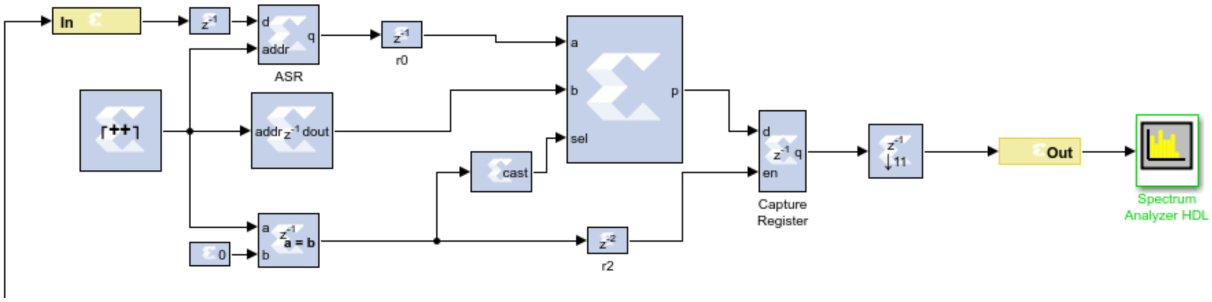
   b. As shown in the following figure, select the **Sources** section in the HDL library, then right-click **Counter** to add this component to the design.



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

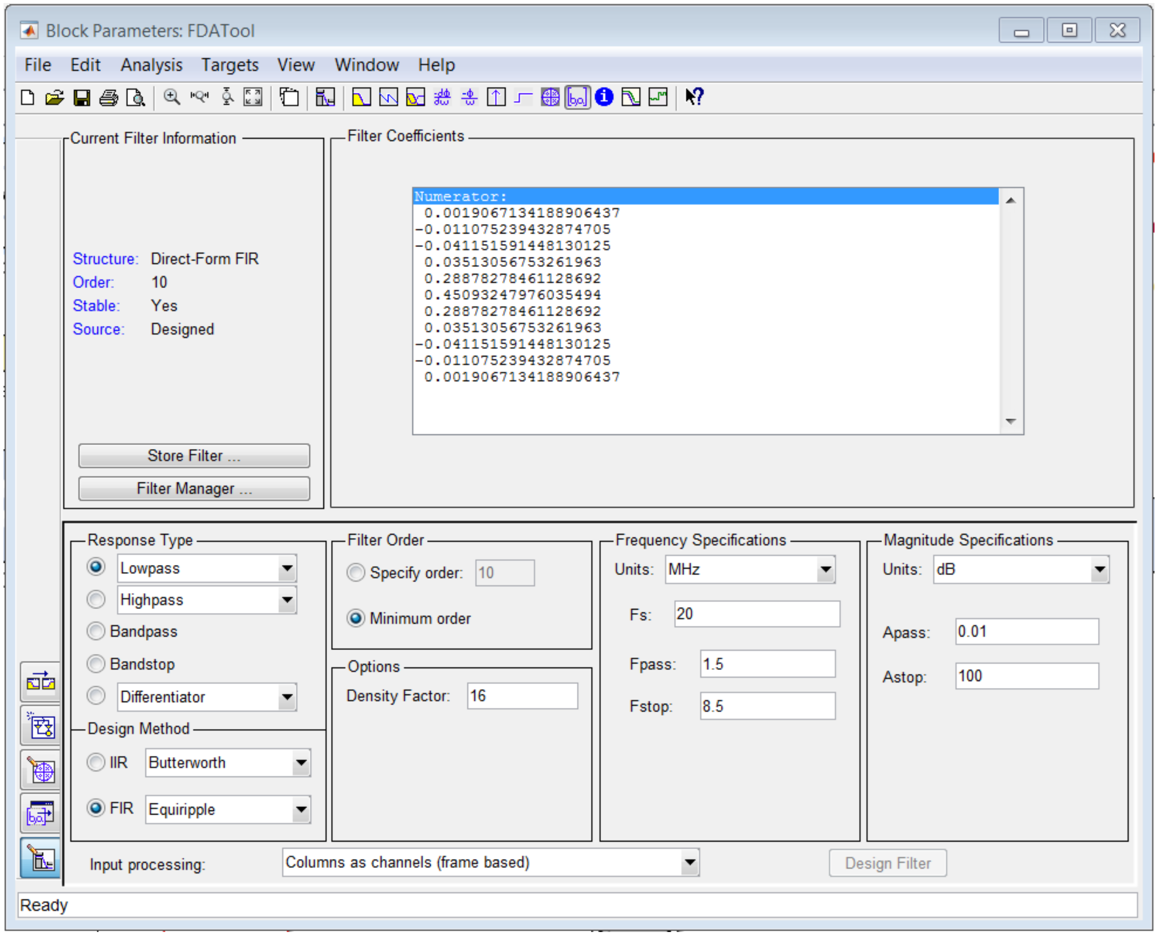
d. Finally, select the **DSP** section and add a DSP Macro 1.0 to the design.

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



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

4. Double-click the **FDATool** instance and select Filter Coefficients [b,a] from the toolbar to review the filter specifications.



Current Filter Information

Structure: Direct-Form FIR
Order: 10
Stable: Yes
Source: Designed

Filter Coefficients

```
Numerator:
 0.0019067134188906437
-0.011075239432874705
-0.041151591448130125
 0.03513056753261963
 0.28878278461128692
 0.45093247976035494
 0.28878278461128692
 0.03513056753261963
-0.041151591448130125
-0.011075239432874705
 0.0019067134188906437
```

Response Type
● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method
○ IIR Butterworth
● FIR Equiripple

Filter Order
○ Specify order: 10
● Minimum order

Options
Density Factor: 16

Frequency Specifications
Units: MHz
Fs: 20
Fpass: 1.5
Fstop: 8.5

Magnitude Specifications
Units: dB
Apass: 0.01
Astop: 100

Input processing: Columns as channels (frame based)

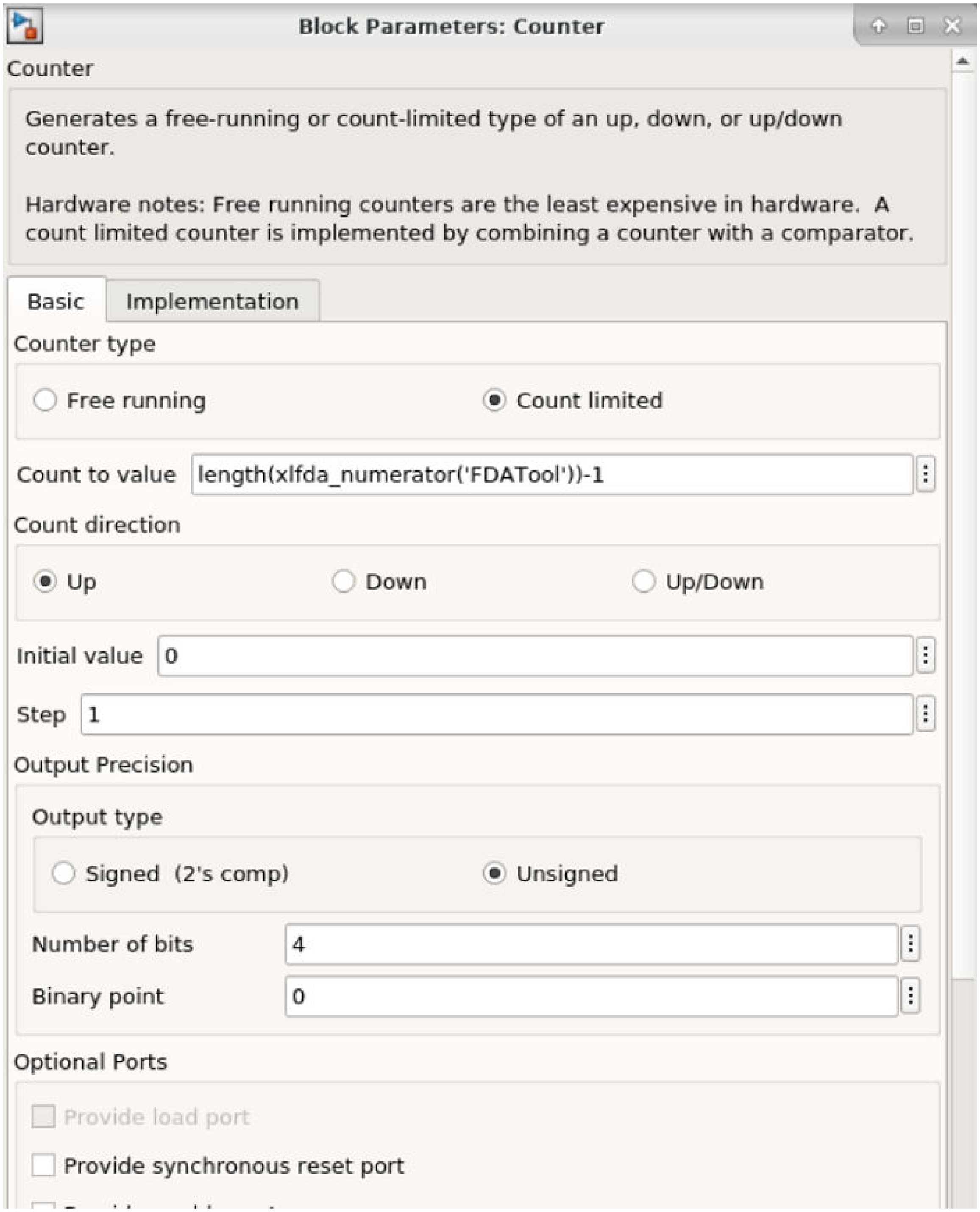
Ready

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.



Block Parameters: Counter

Counter

Generates a free-running or count-limited type of an up, down, or up/down counter.

Hardware notes: Free running counters are the least expensive in hardware. A count limited counter is implemented by combining a counter with a comparator.

Counter type

○ Free running ● Count limited

Count to value: length(xlfda_numerator('FDATool'))-1

Count direction

● Up ○ Down ○ Up/Down

Initial value: 0

Step: 1

Output Precision

Output type

○ Signed (2's comp) ● Unsigned

Number of bits: 4

Binary point: 0

Optional Ports

□ Provide load port

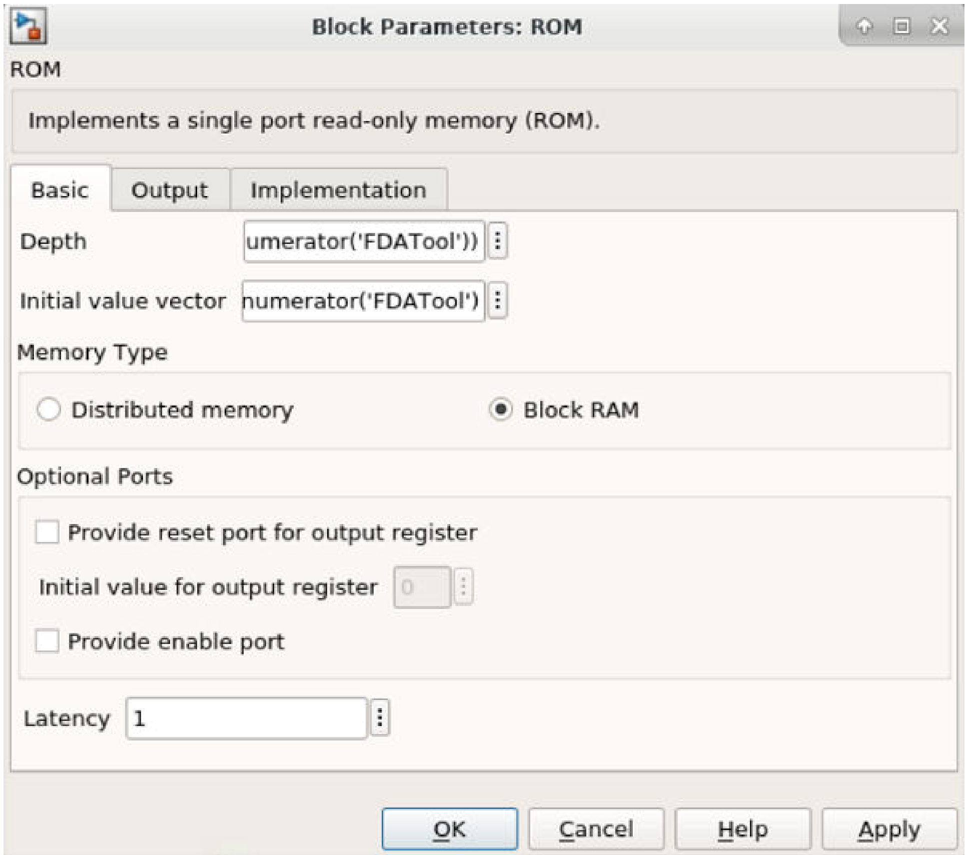
□ Provide synchronous reset port

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.



Block Parameters: ROM

ROM

Implements a single port read-only memory (ROM).

Depth: umerator('FDATool'))

Initial value vector: numerator('FDATool')

Memory Type

○ Distributed memory ● Block RAM

Optional Ports

□ Provide reset port for output register

Initial value for output register: 0

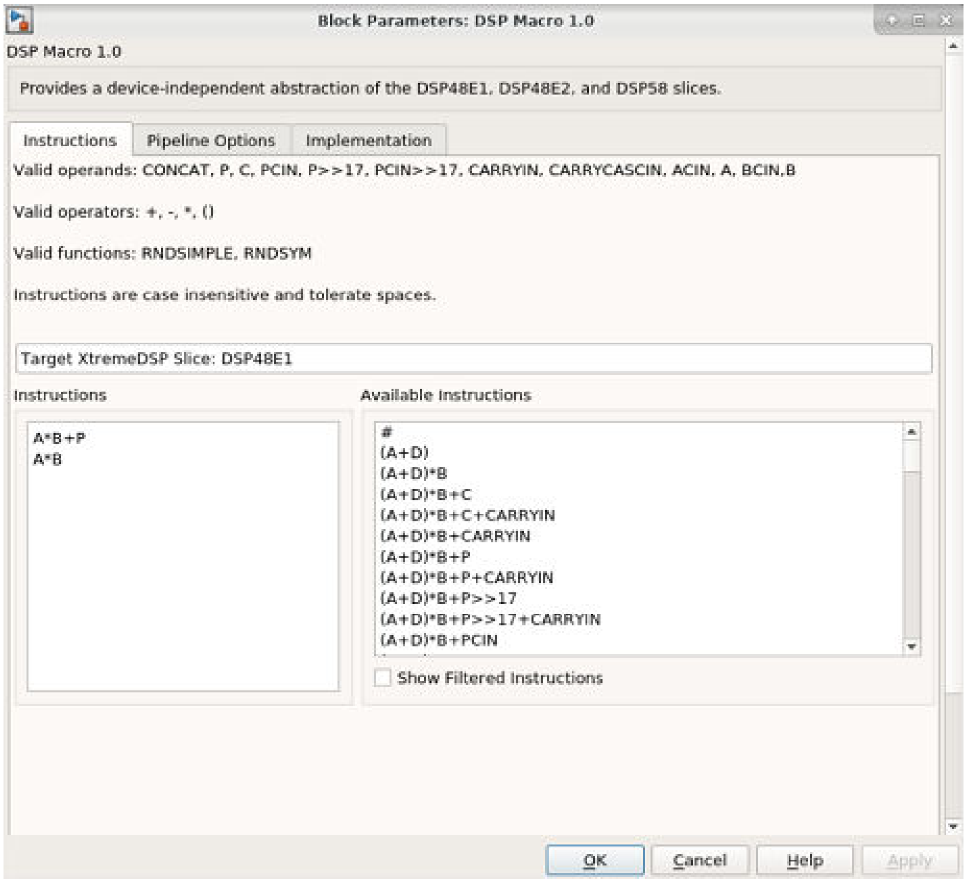
□ Provide enable port

Latency: 1

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

Provides a device-independent abstraction of the DSP48E1, DSP48E2, and DSP58 slices.

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

```
A*B+P
A*B
```

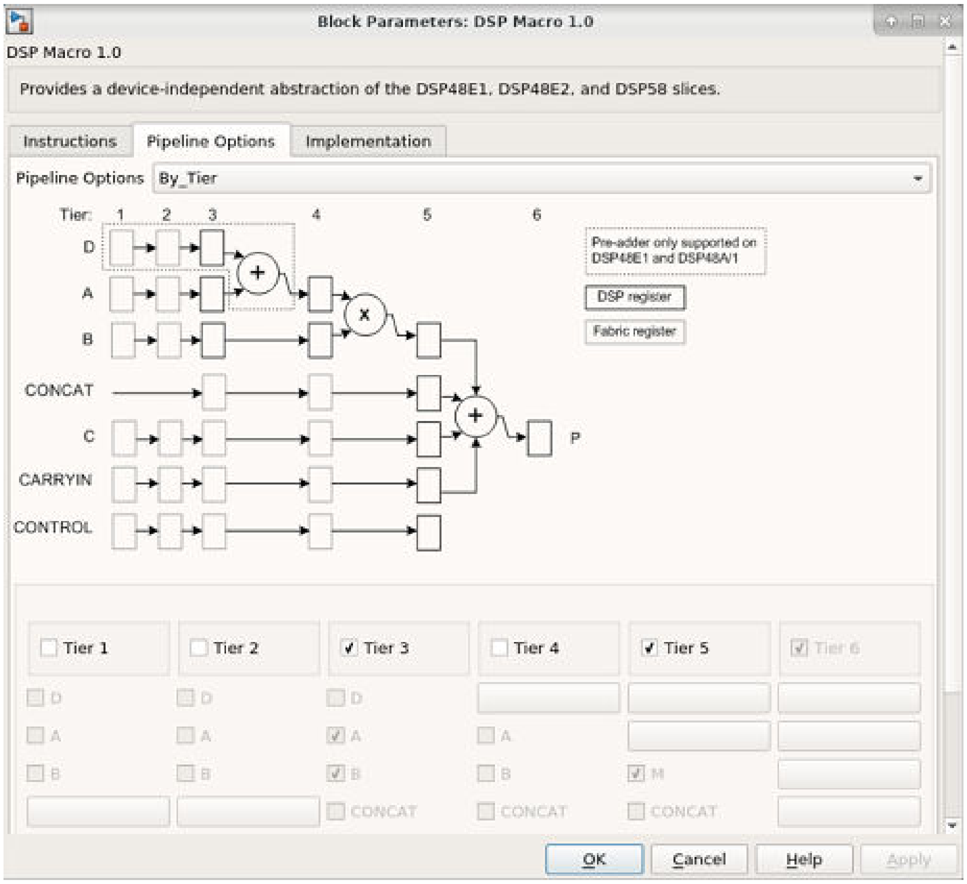
Available Instructions

```
#
(A+D)
(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+P>>17+CARRYIN
(A+D)*B+PCIN
```

□ Show Filtered Instructions

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.



Block Parameters: DSP Macro 1.0

DSP Macro 1.0

Provides a device-independent abstraction of the DSP48E1, DSP48E2, and DSP58 slices.

Instructions | Pipeline Options | Implementation

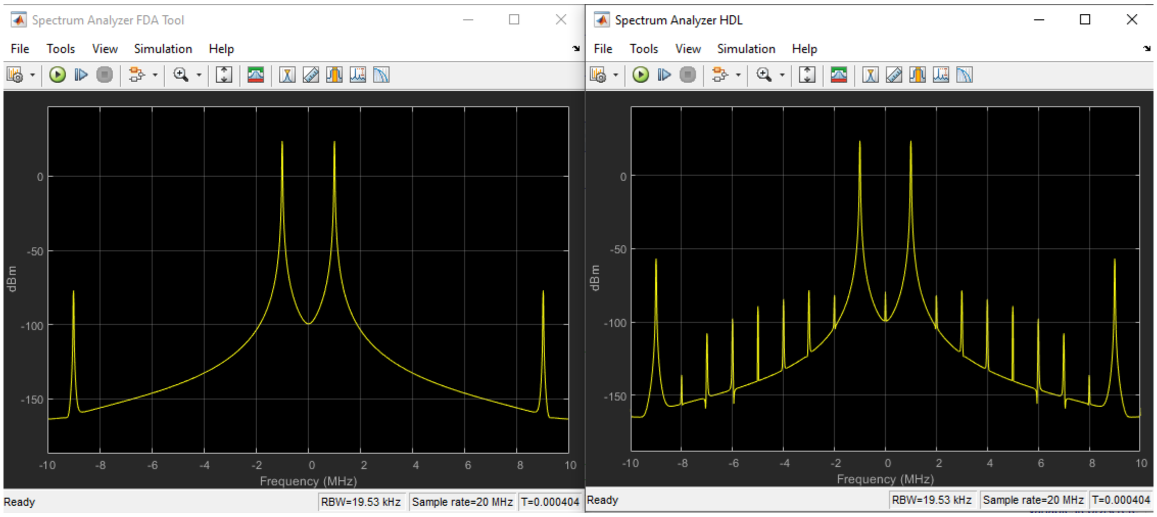
Pipeline Options: By_Tier

Tier 1	Tier 2	Tier 3	Tier 4	Tier 5	Tier 6
□ Tier 1	□ Tier 2	☑ Tier 3	□ Tier 4	☑ Tier 5	☑ Tier 6
□ D	□ D	□ D			
□ A	□ A	☑ A	□ A		
□ B	□ B	☑ B	□ B	☑ M	
		□ CONCAT	□ CONCAT	□ CONCAT	

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.

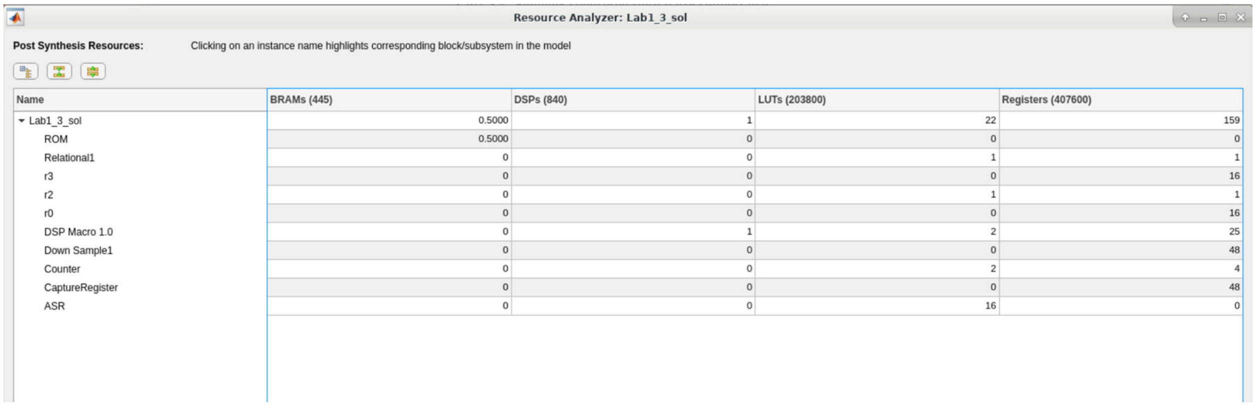


The final step is to compile the design into a hardware description and synthesize it.

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_sol

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▾ Lab1_3_sol	0.5000	1	22	159
ROM	0.5000	0	0	0
Relational1	0	0	1	1
r3	0	0	0	16
r2	0	0	1	1
r0	0	0	0	16
DSP Macro 1.0	0	1	2	25
Down Sample1	0	0	0	48
Counter	0	0	2	4
CaptureRegister	0	0	0	48
ASR	0	0	16	0

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

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

Step 4: Working with Data Types

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

This step has two primary parts:

- In Part 1, you will review and synthesize a design using floating-point data types.
- In Part 2, you will work with the same design, captured as a fixed-point implementation, and refine the data types to create a hardware-efficient design which meets the same requirements.

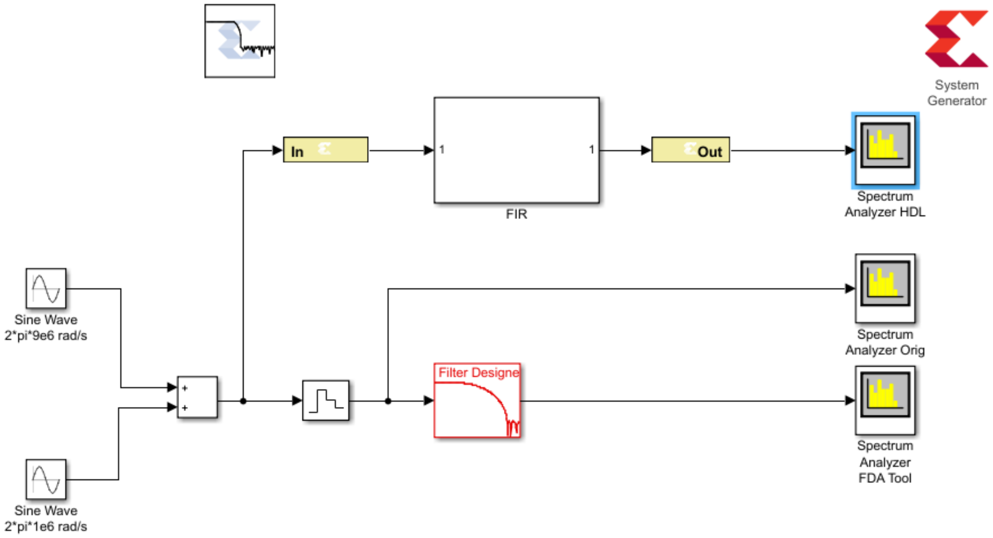
Part 1: Designing with Floating-Point Data Types

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

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

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

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



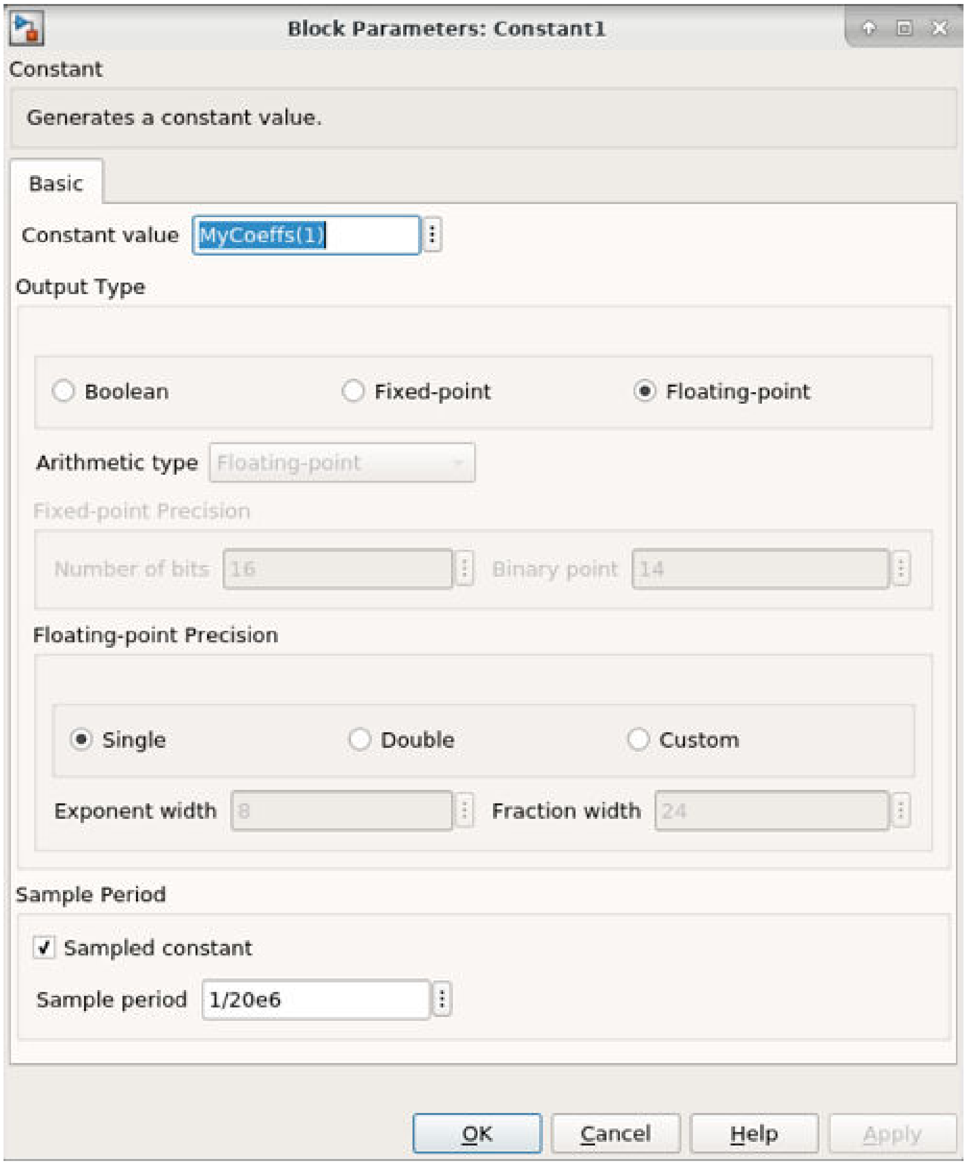
   In the previous figure, both the input and output of instance FIR are of type double.

2. In the MATLAB Command Window enter:

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

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

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



Block Parameters: Constant1

Constant

Generates a constant value.

Basic

Constant value: MyCoeffs(1)

Output Type

○ Boolean ○ Fixed-point ● Floating-point

Arithmetic type: Floating-point

Fixed-point Precision

Number of bits: 16 Binary point: 14

Floating-point Precision

● Single ○ Double ○ Custom

Exponent width: 8 Fraction width: 24

Sample Period

☑ Sampled constant

Sample period: 1/20e6

5. Close the Constant1 Properties editor.

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

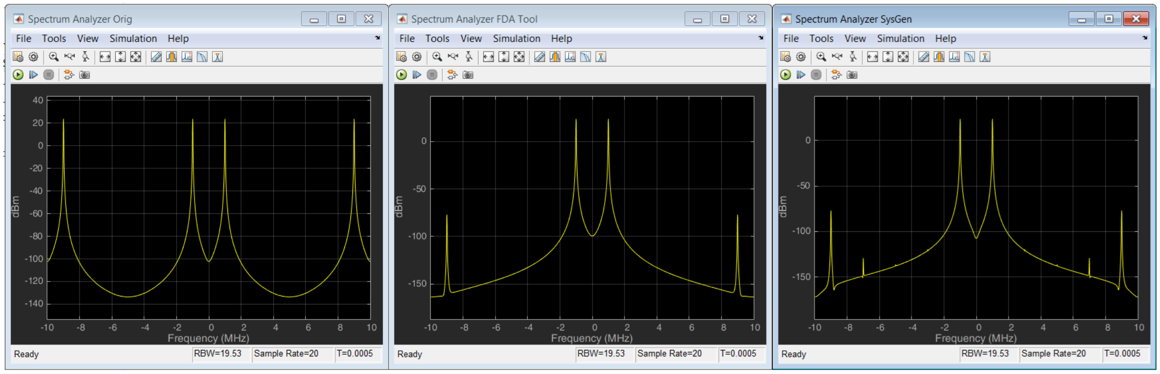
   The design is summing two sine waves, both of which are 9 MHz. The input gateway to the Vitis Model Composer must therefore sample at a rate of at least 18 MHz.

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.

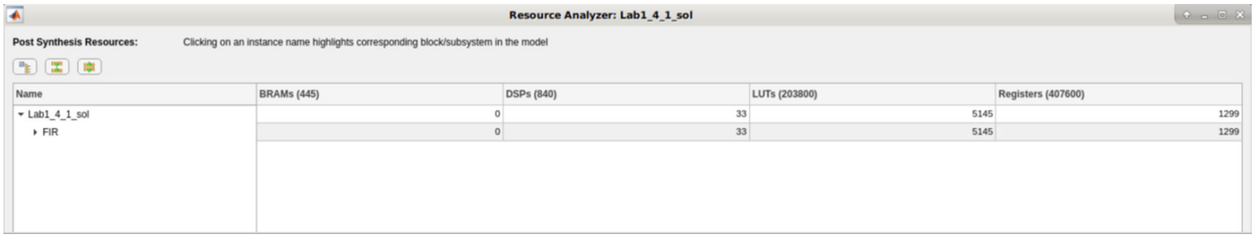
9. Click the Run simulation button to simulate the design.

The results shown in the following figure show the Vitis 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 Vitis 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 the 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.



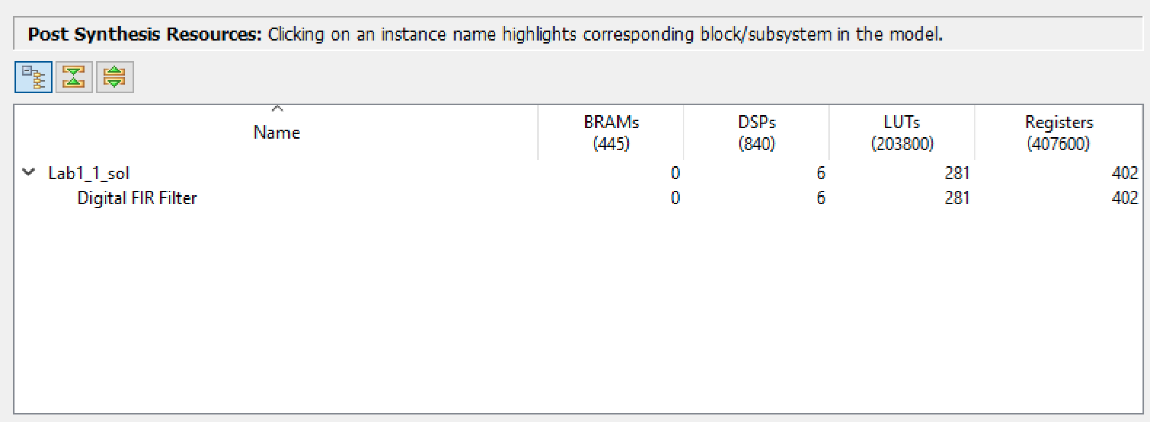
Resource Analyzer: Lab1_4_1_sol

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▾ Lab1_4_1_sol	0	33	5145	1299
▸ FIR	0	33	5145	1299

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 this 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), BRAMs, LUTs, and DSP resources (Xilinx dedicated multiplier/add units).



Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in 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

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

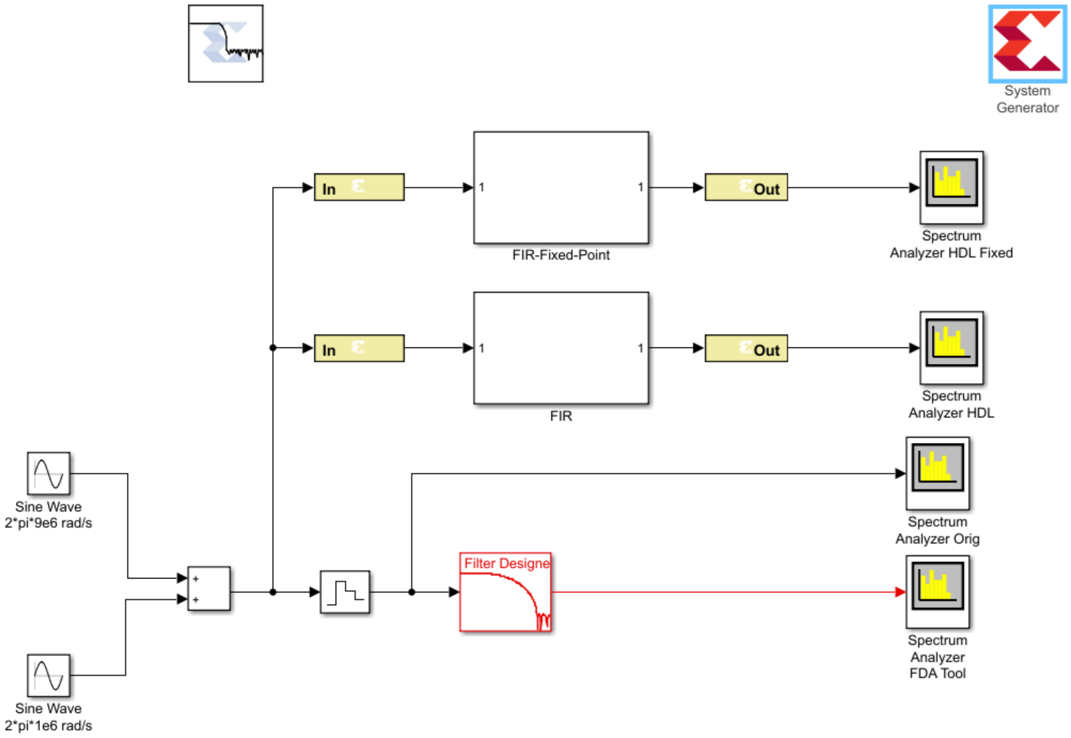
For this particular design, the entire range of the floating-point types is not required. The design is using considerably more resources than what is required. In the next part, you will learn how to compare designs with different data types inside the Simulink environment.

16. Exit the `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 Vitis Model Composer allows you to easily compare the designs, allowing you to make trade-offs between accuracy and resources within the Simulink environment before committing to an FPGA implementation.

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



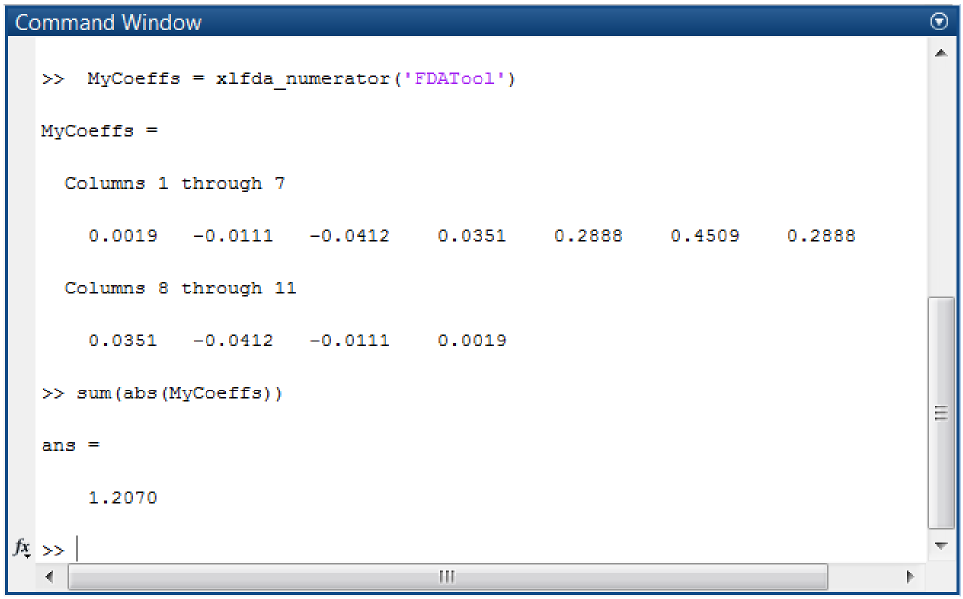
2. In the MATLAB Command Window enter:

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

3. Double-click the instance **Gateway In2** to confirm the data is being sampled as 16-bit fixed-point value.
4. Click **Cancel** to exit the Properties Editor.
5. Click the Run simulation button to simulate the design and confirm instance Spectrum Analyzer HDL Fixed shows the filtered output.

   As you will see if you examine the output of instance FIR-Fixed-Point (shown in the previous figure) Vitis 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.

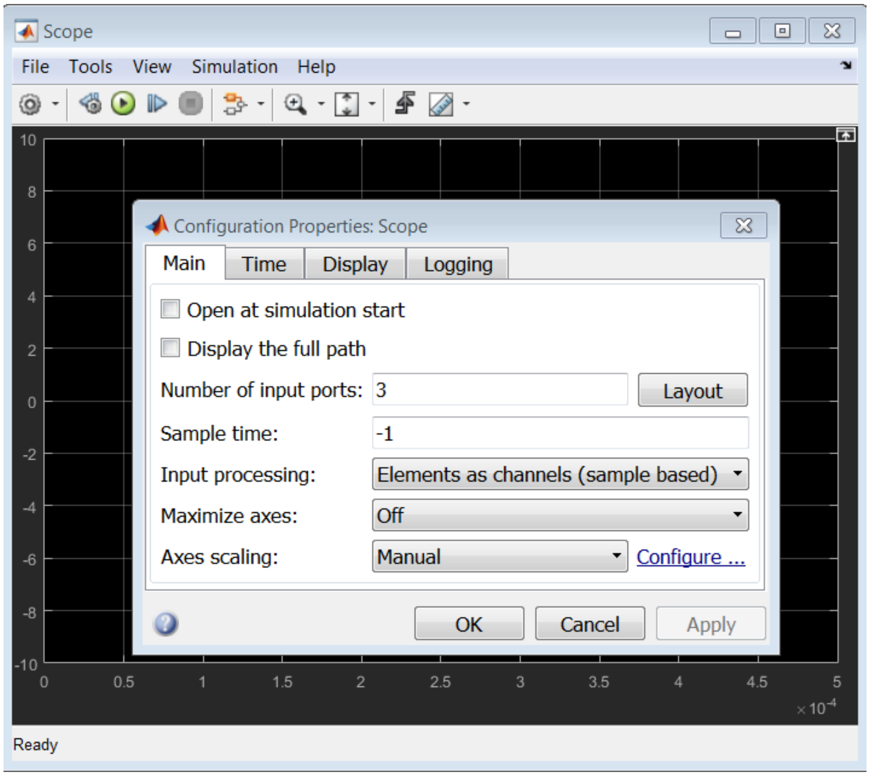


```
>>  MyCoeffs = xlfda_numerator('FDATool')

MyCoeffs =

  Columns 1 through 7

    0.0019   -0.0111   -0.0412    0.0351    0.2888    0.4509    0.2888

  Columns 8 through 11

    0.0351   -0.0412   -0.0111    0.0019

>> sum(abs(MyCoeffs))

ans =

    1.2070

>>
```

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

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

7. Right-click with the mouse anywhere in the canvas and select **Xilinx BlockAdd**.
8. In the Add Block entry box, type `Reinterpret`.
9. Double-click the **Reinterpret** component to add it to the design.
10. Repeat the previous three steps for these components:
    a. Convert
    b. Scope
11. In the design, select the **Gateway Out2** instance.
    a. Right-click and use Copy and Paste to create a new instance of the Gateway Out block.
    b. Paste twice again to create two more instances of the Gateway Out (for a total of three new instances).
12. Double-click the **Scope** component.
    a. In the Scope properties dialog box, select **File → Number of Inputs → 3**.
    b. Select **View → Configuration Properties** and confirm that the Number of input ports is 3.



Configuration Properties: Scope

□ Open at simulation start

□ 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 ...

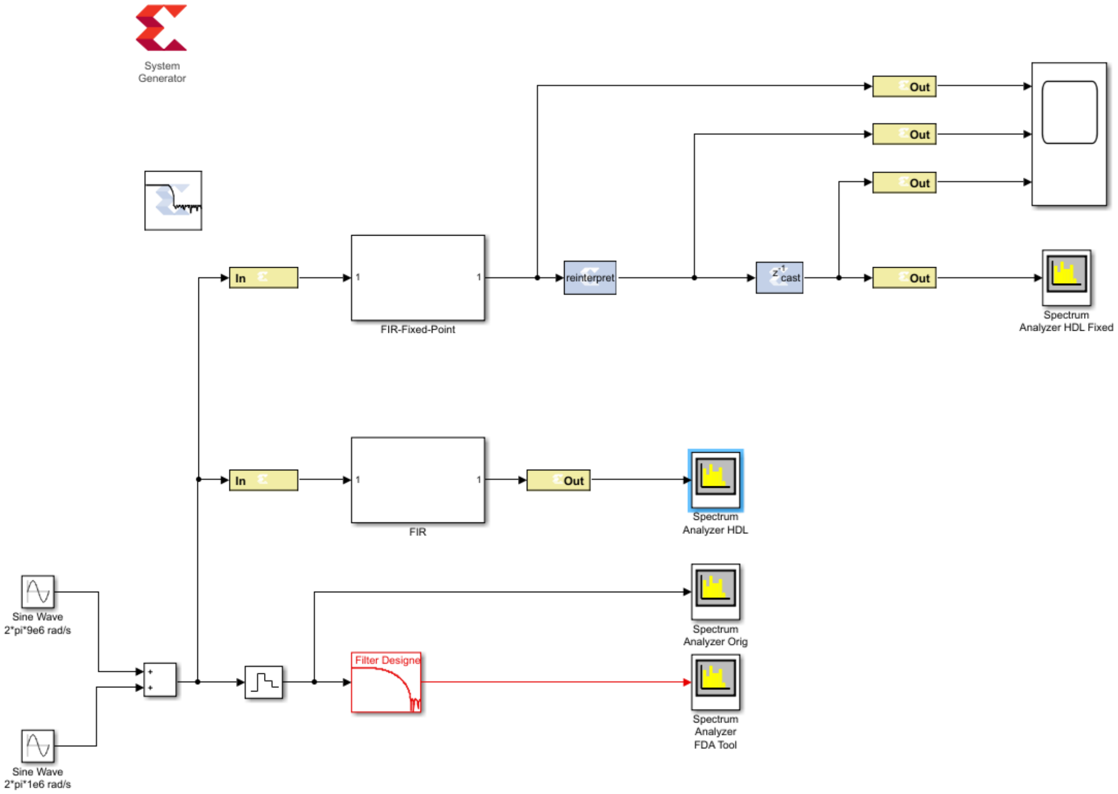
c. Click **OK** to close the Configuration Properties dialog box.

d. Select **File→Close** to close the Scope properties dialog box.

13. Connect the blocks as shown in the next figure.

14. Rename the signal names into the scope as shown in the following figure: Convert, Reinterpret and Growth.

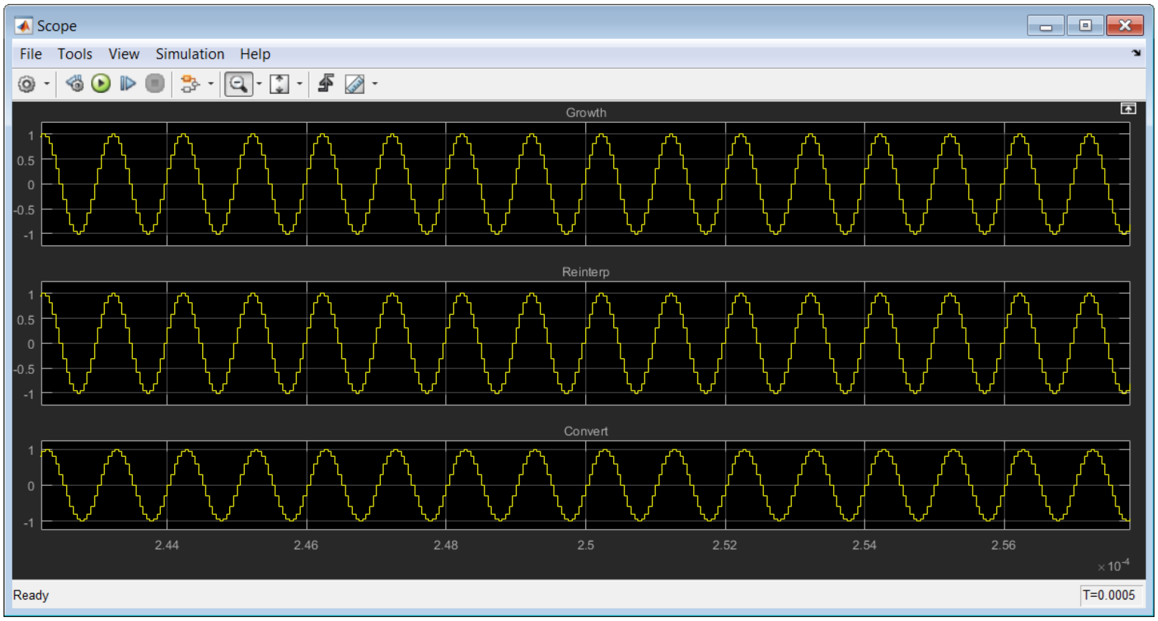
    To rename a signal, click the existing name label and edit the text, or if there is no text double-click the wire and type the name.



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

16. Double-click the **Scope** to examine the signals.

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



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

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

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

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

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

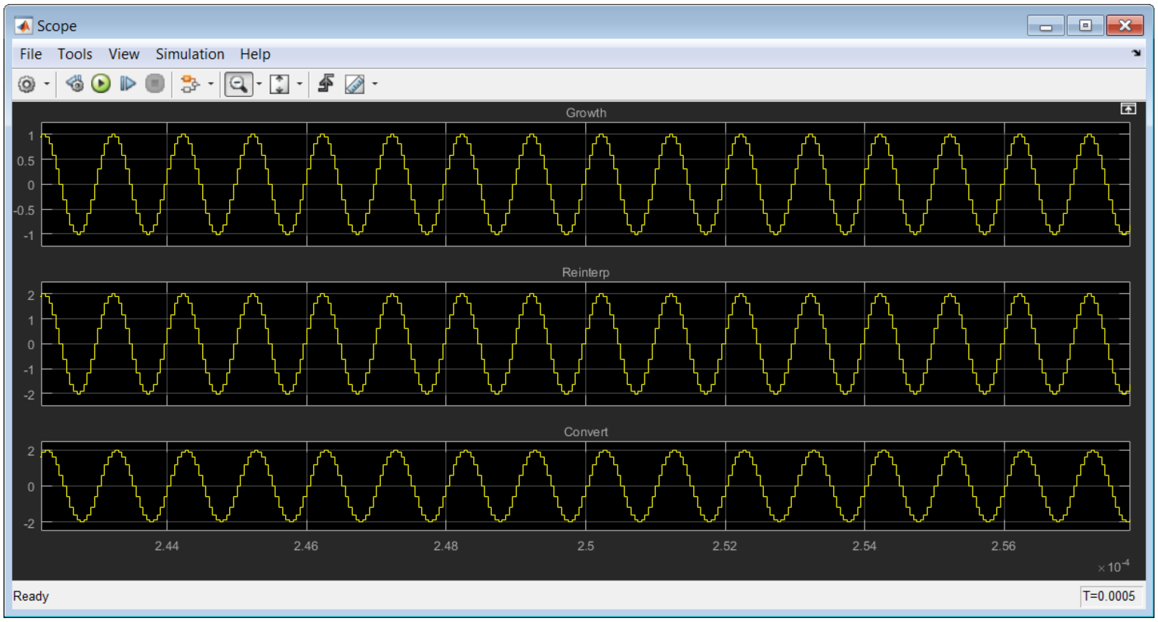
In this exercise, you will use the Convert block to reduce the size of the 43-bit word back to a 16-bit value. In this exercise the Reinterpret block has been used to model a more complex design and scaled the data by a factor of 2. You must therefore ensure the output has enough bits above the binary point to represent this increase.

20. Double-click the **Convert** block to open the Properties Editor.
21. In the Fixed-Point Precision section, enter `13` for the Binary Point and click **OK**.
22. Save the design.
23. Click the Run simulation button to simulate the design.
24. Double-click the **Scope** to examine the signals.

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

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

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

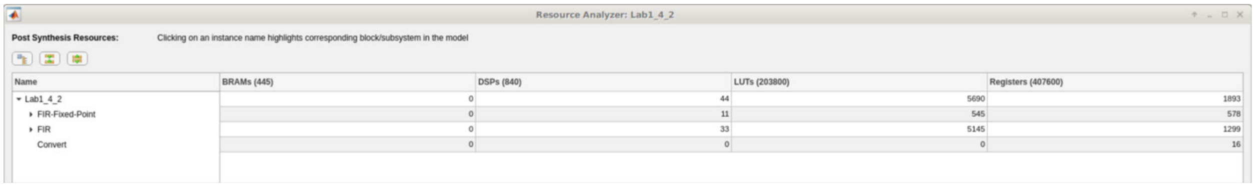


The final step is to synthesize this design into hardware.

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

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

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



Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
Lab1_4_2	0	44	5690	1893
FIR-Fixed-Point	0	11	545	578
FIR	0	33	5145	1299
Convert	0	0	0	16

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 Vitis 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 Vitis 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 Vitis 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 Vitis Model Composer (`*.slx`) files for this lab.

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

Lab 2: Importing Code into a Vitis Model Composer HDL Design

Objectives

After completing this lab, you will be able to:

- Create a Finite State Machine using the MCode block in Vitis Model Composer.

- Import an RTL HDL description into Vitis 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

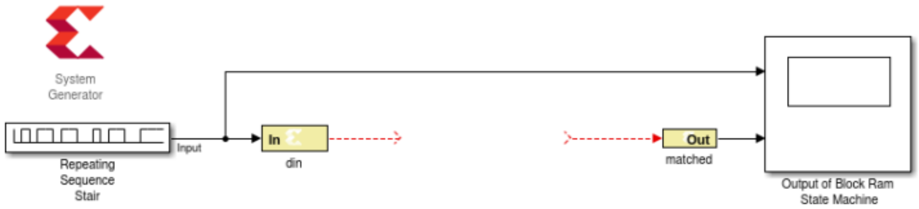
In this step you will be creating a simple Finite State Machine (FSM) using the MCode block to detect a sequence of binary values 1011. The FSM needs to be able to detect multiple transmissions as well, such as 10111011.

Procedure

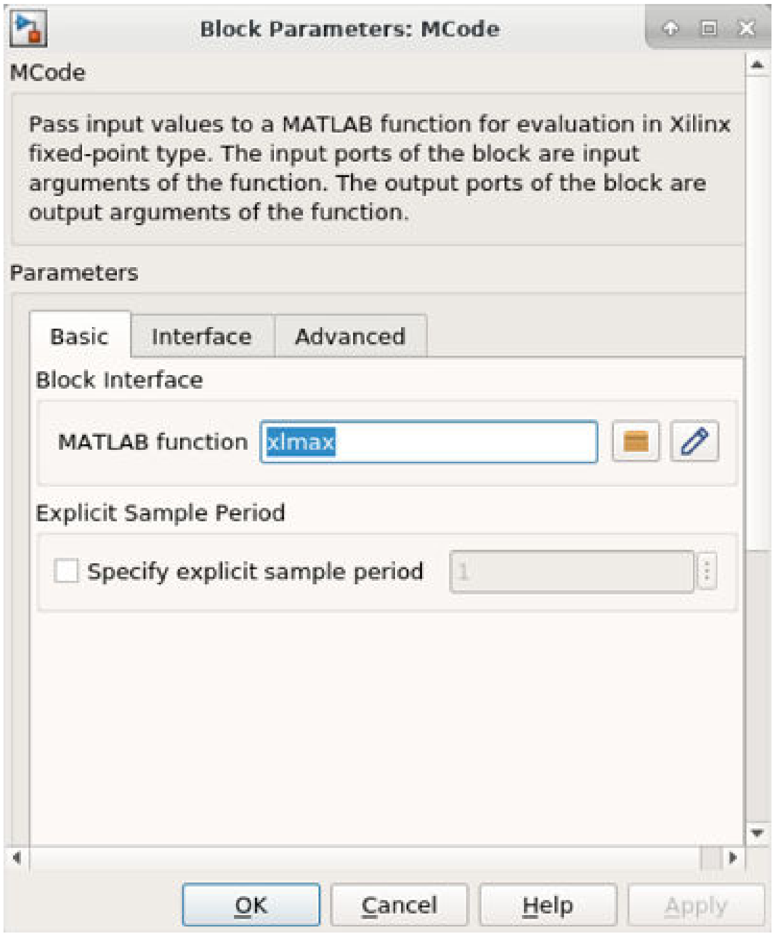
In this step you will create the control logic for a Finite State Machine using M-code. You will then simulate the final design to confirm the correct operation.

1. Launch Vitis 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 the Xilinx 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.



Block Parameters: MCode

MCode

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.

Parameters

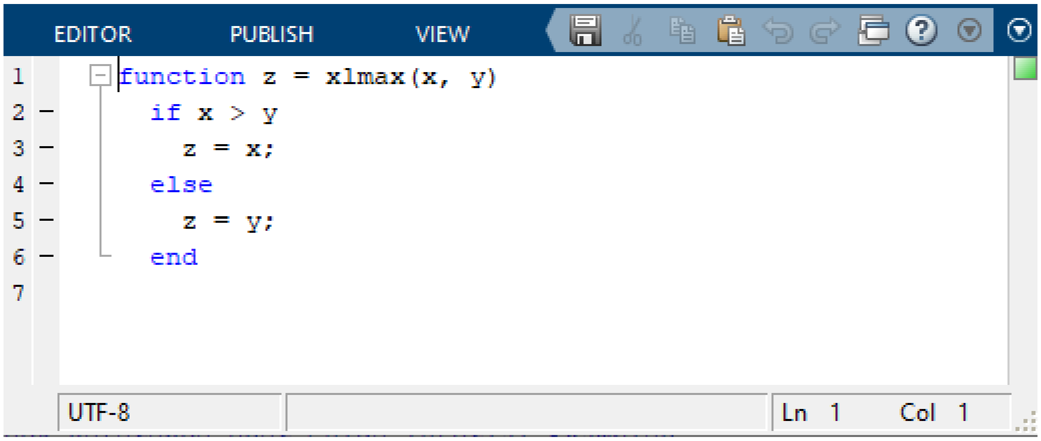
Block Interface

MATLAB function: xlmax

Explicit Sample Period

□ Specify explicit sample period: 1

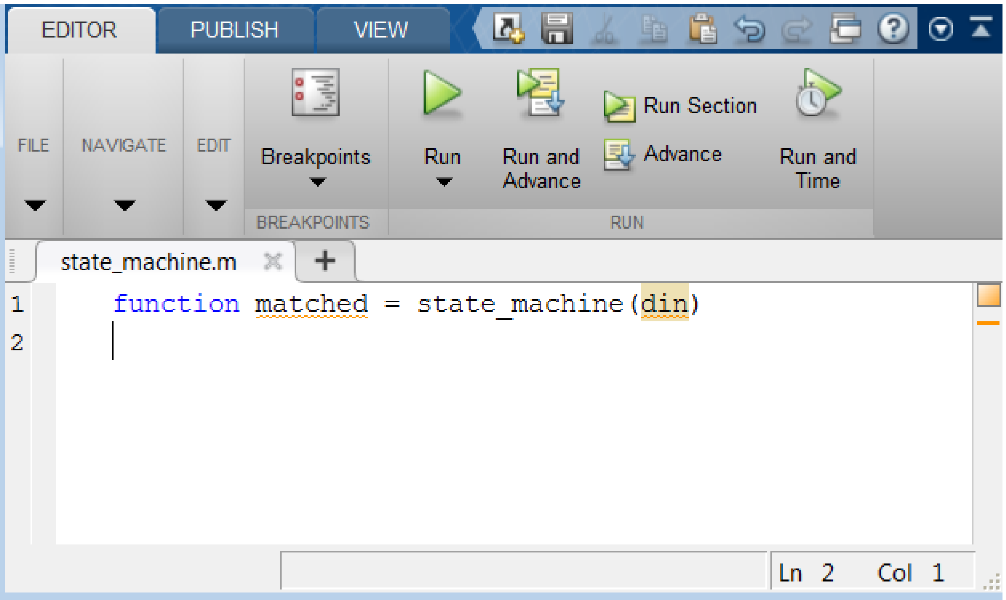
The following figure shows the default M-code in the MATLAB text editor.



```
function z = xlmax(x, y)
  if x > y
    z = x;
  else
    z = y;
  end
```

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.



```
function matched = state_machine(din)

```

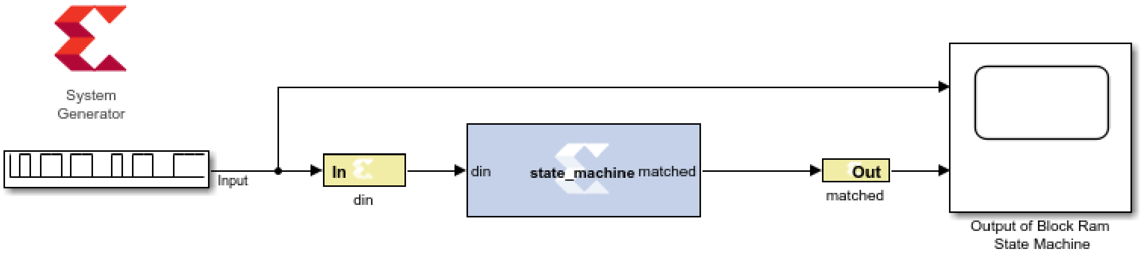
7. After you make the edits, use Save As to save the MATLAB file as `state_machine.m` to the `Lab2/M_code` folder.

    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`).

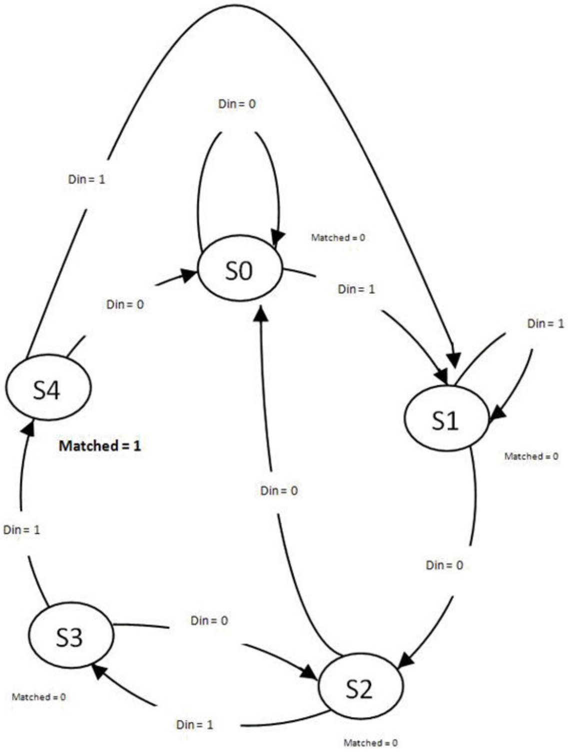
8. In the MCode Properties Editor, click **OK**.

    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});
    ```

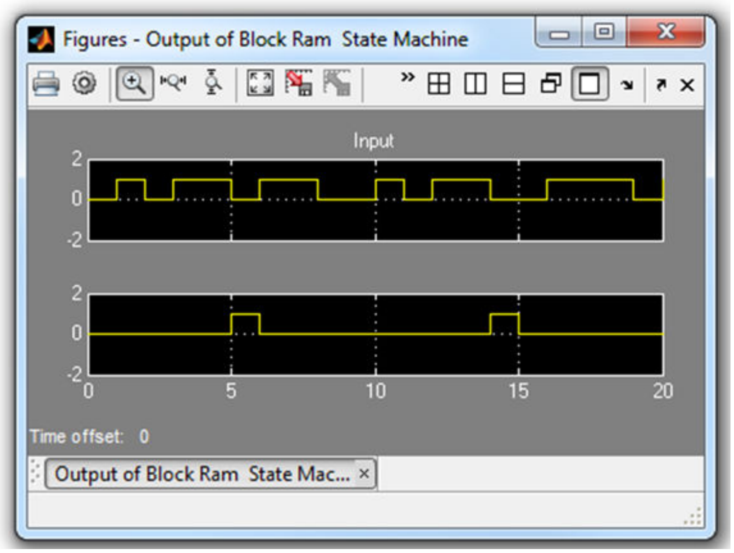
11. Use a switch-case statement to define the FSM states shown. A small sample is provided, shown as follows, to get you started.

    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.


Figures - Output of Block Ram State Machine
Input
2
0
-2
2
0
-2
0
5
10
15
20
Time offset: 0
Output of Block Ram State Mac...

Step 2: Modeling Blocks with HDL

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

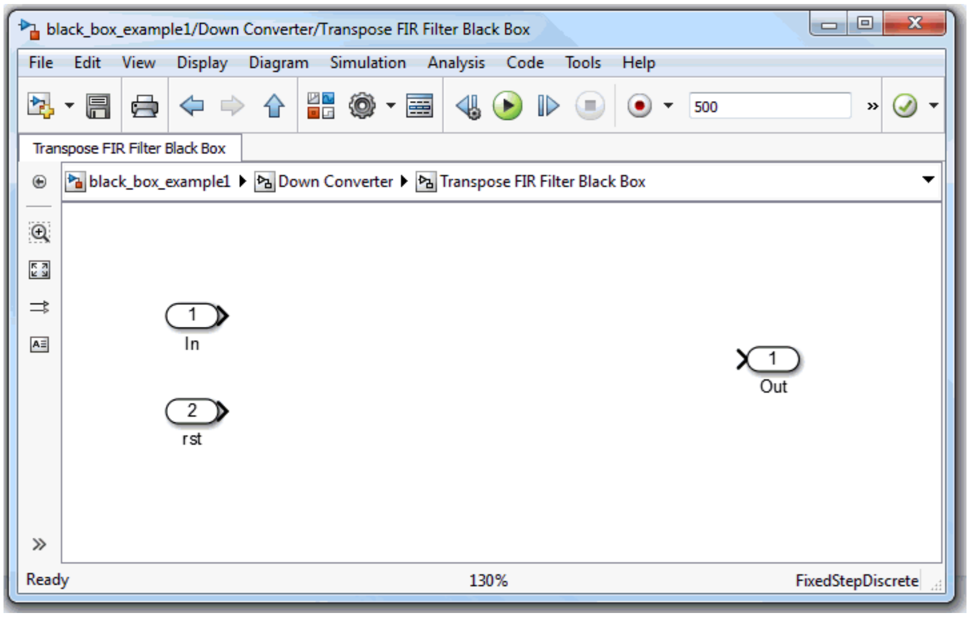
A black box allows the design to be imported into Vitis Model Composer even though the description is in Hardware Description Language (HDL) format.

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` on the MATLAB command line.
3. Open the subsystem named Down Converter.
4. Open the subsystem named Transpose FIR Filter Black Box.

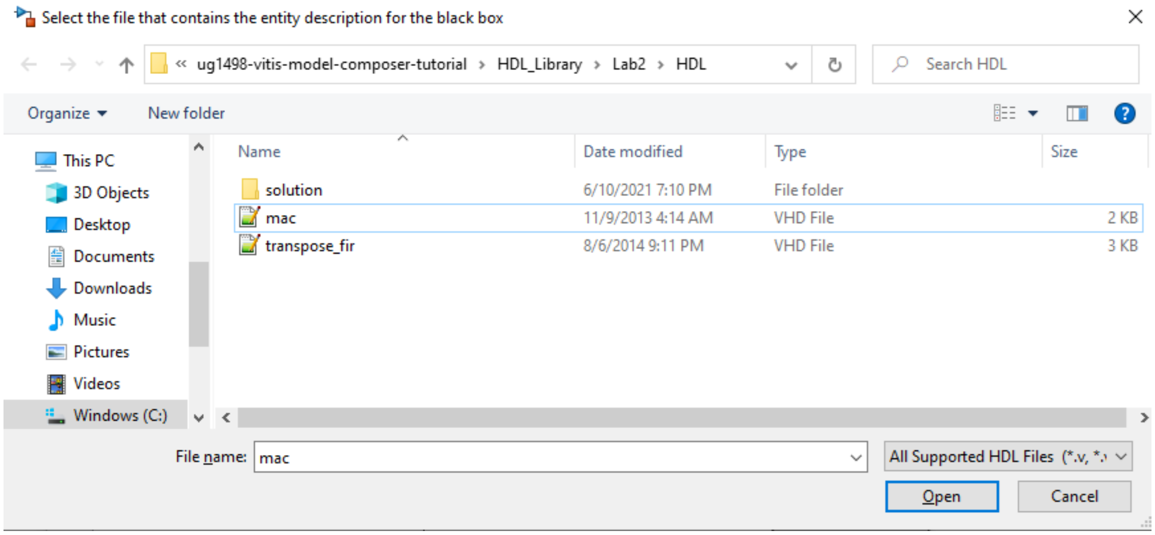
   At this point, the subsystem contains two input ports and one output port. You will add a black box to this subsystem:


black_box_example1/Down Converter/Transpose FIR Filter Black Box
File Edit View Display Diagram Simulation Analysis Code Tools Help
500
Transpose FIR Filter Black Box
black_box_example1 ▸ Down Converter ▸ Transpose FIR Filter Black Box
1
In
2
rst
1
Out
Ready
130%
FixedStepDiscrete

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.


Select the file that contains the entity description for the black box

« ug1498-vitis-model-composer-tutorial › HDL_Library › Lab2 › HDL

Name	Date modified	Type	Size
solution	6/10/2021 7:10 PM	File folder	
mac	11/9/2013 4:14 AM	VHD File	2 KB
transpose_fir	8/6/2014 9:11 PM	VHD File	3 KB

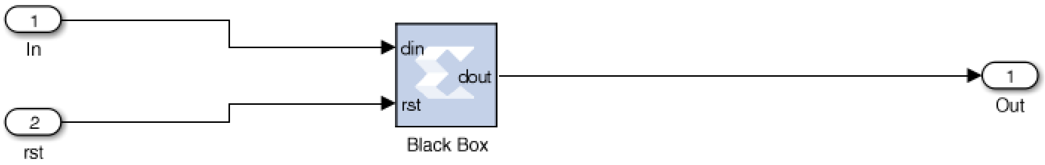
File name: mac

All Supported HDL Files (*.v, *.

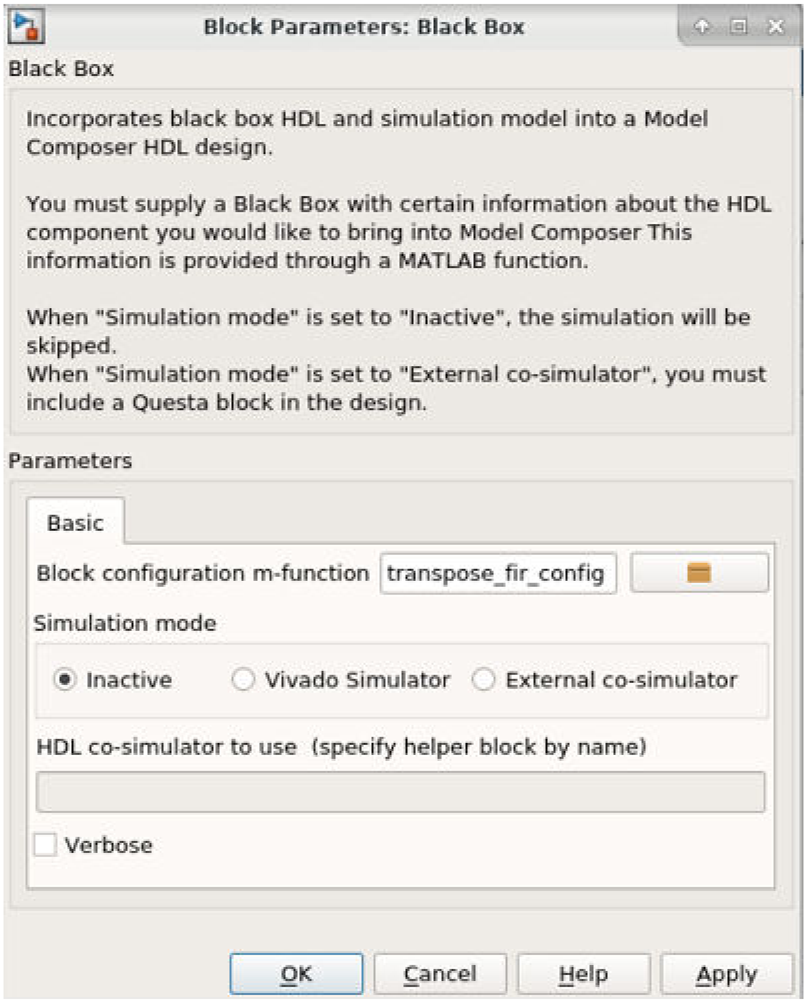
The associated configuration M-code `transpose_fir_config.m` opens in an Editor for modifications.

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:



Block Parameters: Black Box

Black Box

Incorporates black box HDL and simulation model into a Model Composer HDL design.

You must supply a Black Box with certain information about the HDL component you would like to bring into Model Composer This information is provided through a MATLAB function.

When "Simulation mode" is set to "Inactive", the simulation will be skipped.
When "Simulation mode" is set to "External co-simulator", you must include a Questa block in the design.

Parameters

Basic

Block configuration m-function: transpose_fir_config

Simulation mode

● Inactive ○ Vivado Simulator ○ External co-simulator

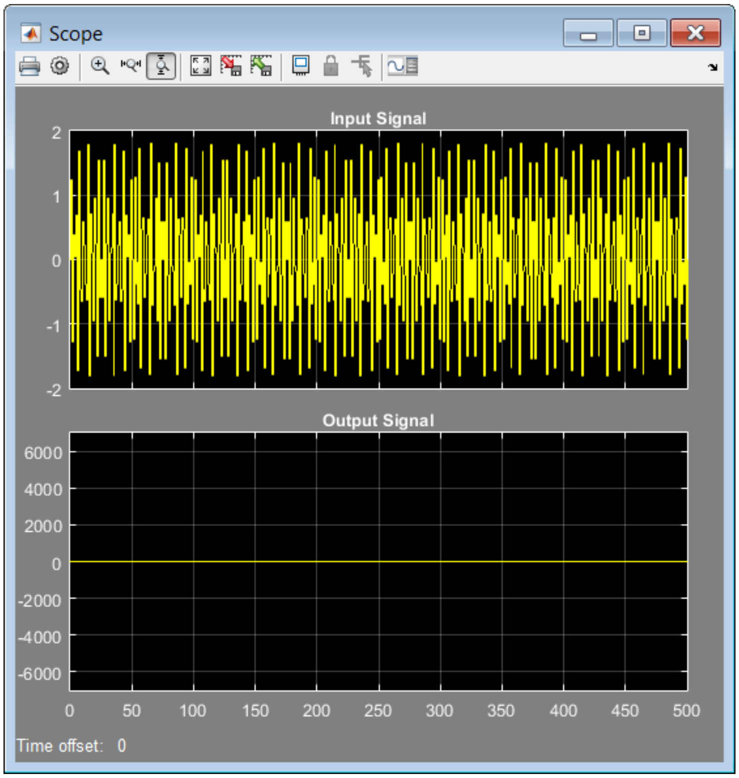
HDL co-simulator to use (specify helper block by name)

□ Verbose

The following are the fields in the dialog box:

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

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



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

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

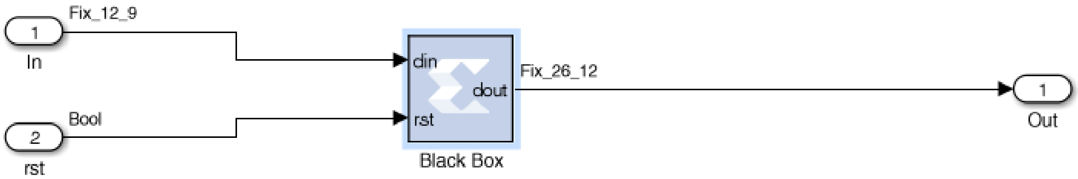
    ```
    dout_port.setType('Fix_26_12');
    ```

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

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

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

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

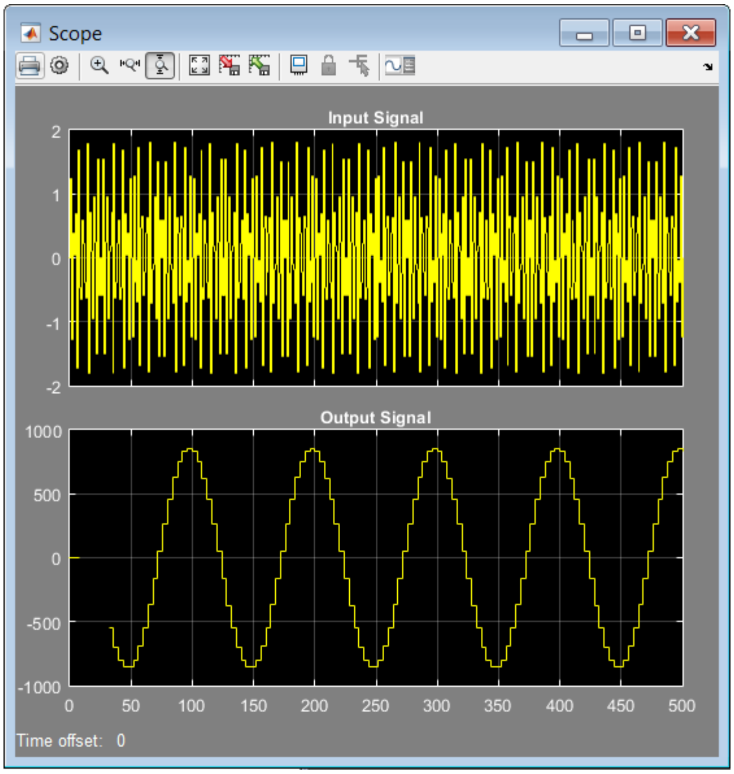


20. From the Black Box block parameter dialog box, change the Simulation mode field from **Inactive** to **Vivado Simulator** and then click **OK**.
21. Move to the top-level of the design and run the simulation.
22. Examine the scope output after the simulation has completed.

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

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

    You should see a display similar to that shown below.



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 Vitis Model Composer using HDL by incorporating an existing VHDL RTL design and the importance of matching the data types of the Vitis Model Composer model with those of the RTL design and how the RTL design is simulated within Vitis Model Composer.
- Not part of ELEC 522 Project 1. 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 Vitis 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 Vitis 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:

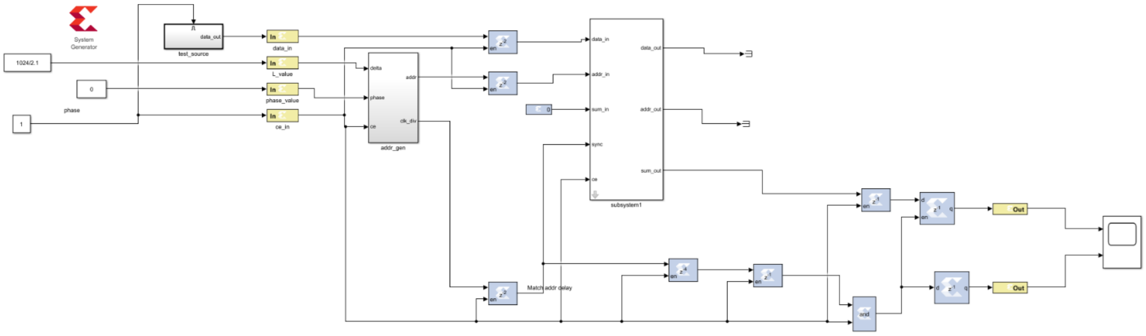
- In Step 1 you will learn how to do timing analysis in Vitis Model Composer.
- In Step 2 you will learn how to perform resource analysis in Vitis Model Composer.

Step 1: Timing Analysis in Vitis Model Composer

1. Invoke Vitis Model Composer.
    - On Windows systems select **Windows → Xilinx Design Tools → Vitis Model Composer 2021.2**.
    - On Linux systems, type `model_cpomposer` at the command prompt.
2. Navigate to the Lab3 folder: `\HDL_Library\Lab3`.

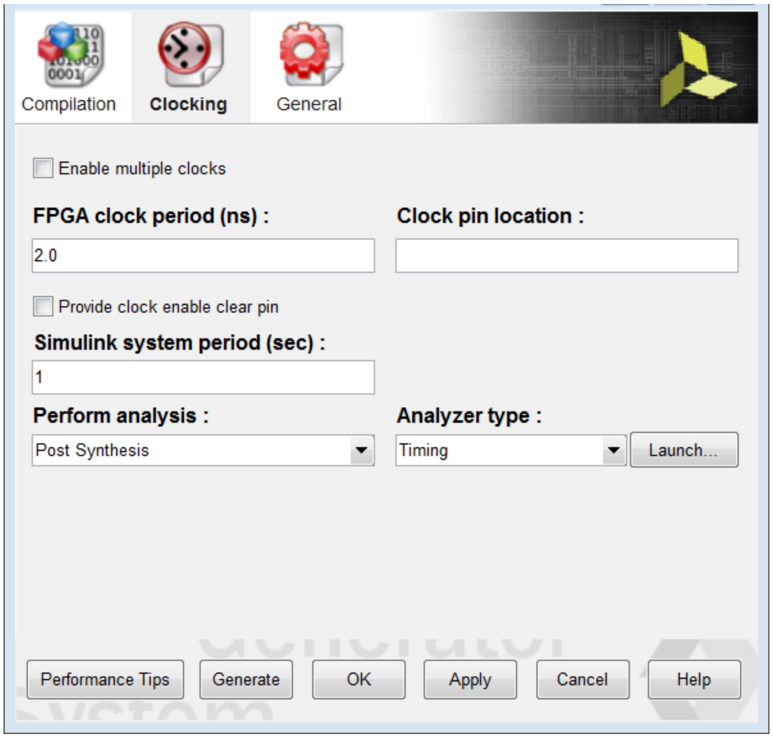
    You can view the directory contents in the MATLAB® Current Folder browser, or type `ls` at the command line prompt.
3. Open the Lab3 design using one of the following:
    - At the MATLAB command prompt, type `open Lab3.slx`
    - Double-click `Lab3.slx` in the Current Folder browser.

    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.



□ Enable multiple clocks

FPGA clock period (ns) : 2.0

Clock pin location :

□ Provide clock enable clear pin

Simulink system period (sec) : 1

Perform analysis : Post Synthesis

Analyzer type : Timing Launch...

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

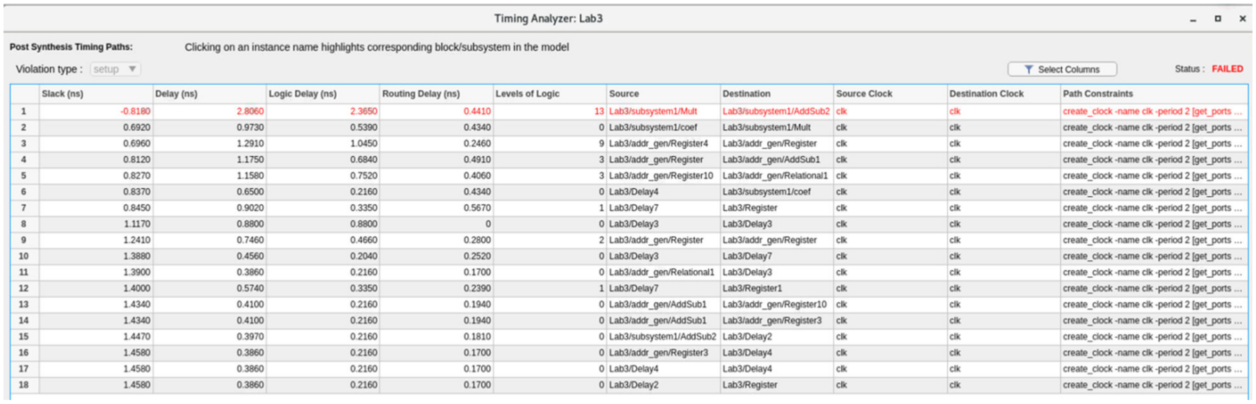
   When you generate, the following occurs:

   a. Vitis Model Composer generates the required files for the selected compilation target. For timing analysis Vitis 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. Vitis Model Composer processes the timing information and displays a Timing Analyzer table with timing paths information as shown in the following figure.



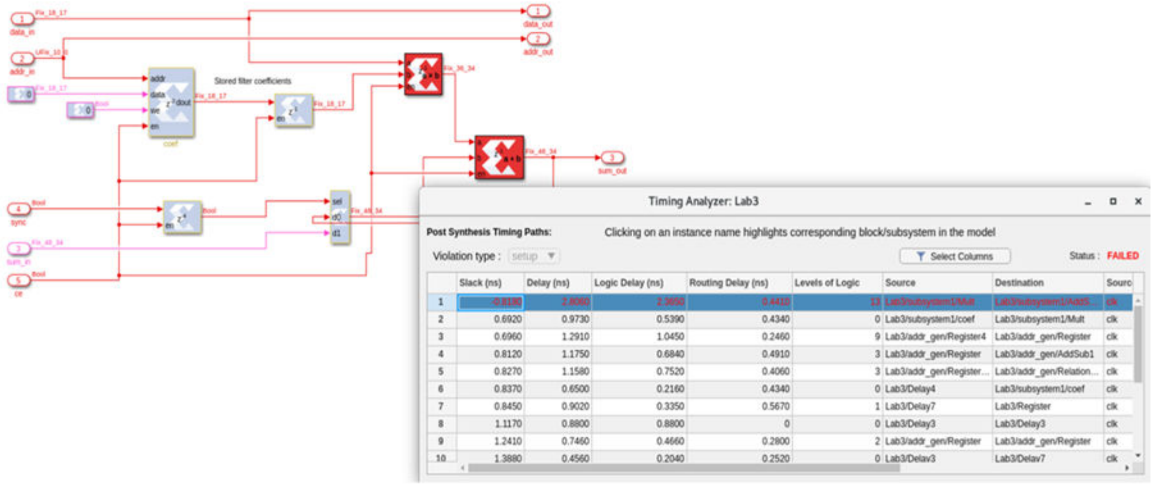
Post Synthesis Timing Paths: Clicking on an instance name highlights corresponding block/subsystem in the model

Violation type : setup

Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path Constraints
1	-0.8180	2.8060	2.3650	0.4410	13	Lab3/subsystem1/Mult	Lab3/subsystem1/AddSub2	clk	clk	create_clock -name clk -period 2 [get_ports ...
2	0.6920	0.9730	0.5390	0.4340	0	Lab3/subsystem1/coef	Lab3/subsystem1/Mult	clk	clk	create_clock -name clk -period 2 [get_ports ...
3	0.6960	1.2910	1.0450	0.2460	9	Lab3/addr_gen/Register4	Lab3/addr_gen/Register	clk	clk	create_clock -name clk -period 2 [get_ports ...
4	0.8120	1.1750	0.6840	0.4910	3	Lab3/addr_gen/Register	Lab3/addr_gen/AddSub1	clk	clk	create_clock -name clk -period 2 [get_ports ...
5	0.8270	1.1580	0.7520	0.4060	3	Lab3/addr_gen/Register10	Lab3/addr_gen/Relational1	clk	clk	create_clock -name clk -period 2 [get_ports ...
6	0.8370	0.6500	0.2160	0.4340	0	Lab3/Delay4	Lab3/subsystem1/coef	clk	clk	create_clock -name clk -period 2 [get_ports ...
7	0.8450	0.9020	0.3350	0.5670	1	Lab3/Delay7	Lab3/Register	clk	clk	create_clock -name clk -period 2 [get_ports ...
8	1.1170	0.8800	0.8800	0	0	Lab3/Delay3	Lab3/Delay3	clk	clk	create_clock -name clk -period 2 [get_ports ...
9	1.2410	0.7460	0.4660	0.2800	2	Lab3/addr_gen/Register	Lab3/addr_gen/Register	clk	clk	create_clock -name clk -period 2 [get_ports ...
10	1.3880	0.4560	0.2040	0.2520	0	Lab3/Delay3	Lab3/Delay7	clk	clk	create_clock -name clk -period 2 [get_ports ...
11	1.3900	0.3860	0.2160	0.1700	0	Lab3/addr_gen/Relational1	Lab3/Delay3	clk	clk	create_clock -name clk -period 2 [get_ports ...
12	1.4000	0.5740	0.3350	0.2390	1	Lab3/Delay7	Lab3/Register1	clk	clk	create_clock -name clk -period 2 [get_ports ...
13	1.4340	0.4100	0.2160	0.1940	0	Lab3/addr_gen/AddSub1	Lab3/addr_gen/Register10	clk	clk	create_clock -name clk -period 2 [get_ports ...
14	1.4340	0.4100	0.2160	0.1940	0	Lab3/addr_gen/AddSub1	Lab3/addr_gen/Register3	clk	clk	create_clock -name clk -period 2 [get_ports ...
15	1.4470	0.3970	0.2160	0.1810	0	Lab3/subsystem1/AddSub2	Lab3/Delay2	clk	clk	create_clock -name clk -period 2 [get_ports ...
16	1.4580	0.3860	0.2160	0.1700	0	Lab3/addr_gen/Register3	Lab3/Delay4	clk	clk	create_clock -name clk -period 2 [get_ports ...
17	1.4580	0.3860	0.2160	0.1700	0	Lab3/Delay4	Lab3/Delay4	clk	clk	create_clock -name clk -period 2 [get_ports ...
18	1.4580	0.3860	0.2160	0.1700	0	Lab3/Delay2	Lab3/Register	clk	clk	create_clock -name clk -period 2 [get_ports ...

9. In the timing analyzer table:
    - Paths with lowest slack values display, with the worst slack showing at the top and increasing toward the bottom.
    - 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 Vitis 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.



Timing Analyzer: Lab3

Post Synthesis Timing Paths: Clicking on an instance name highlights corresponding block/subsystem in the model

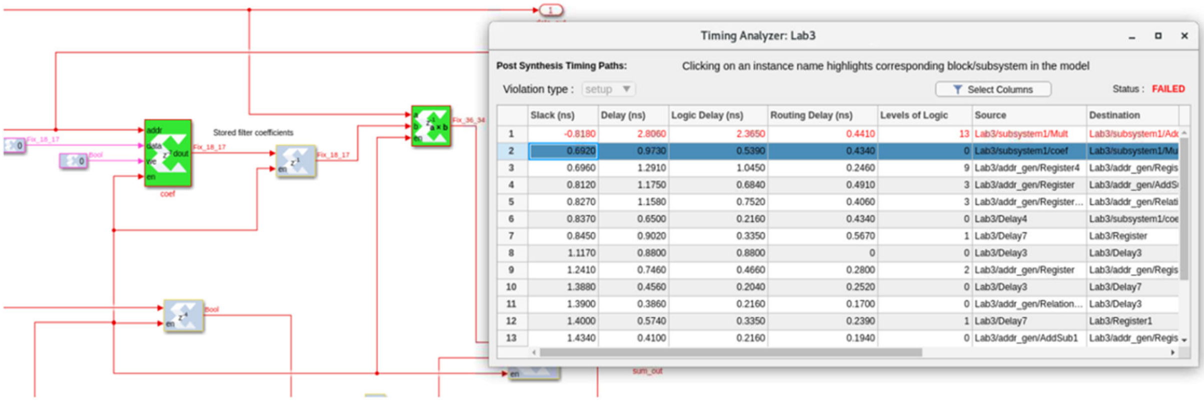
Violation type : setup

Select Columns

Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Sourc
1	-0.8180	2.8060	2.3650	0.4410	13	Lab3/subsystem1/Mult	Lab3/subsystem1/AddS...	clk
2	0.6920	0.9730	0.5390	0.4340	0	Lab3/subsystem1/coef	Lab3/subsystem1/Mult	clk
3	0.6960	1.2910	1.0450	0.2460	9	Lab3/addr_gen/Register4	Lab3/addr_gen/Register	clk
4	0.8120	1.1750	0.6840	0.4910	3	Lab3/addr_gen/Register	Lab3/addr_gen/AddSub1	clk
5	0.8270	1.1580	0.7520	0.4060	3	Lab3/addr_gen/Register...	Lab3/addr_gen/Relation...	clk
6	0.8370	0.6500	0.2160	0.4340	0	Lab3/Delay4	Lab3/subsystem1/coef	clk
7	0.8450	0.9020	0.3350	0.5670	1	Lab3/Delay7	Lab3/Register	clk
8	1.1170	0.8800	0.8800	0	0	Lab3/Delay3	Lab3/Delay3	clk
9	1.2410	0.7460	0.4660	0.2800	2	Lab3/addr_gen/Register	Lab3/addr_gen/Register	clk
10	1.3880	0.4560	0.2040	0.2520	0	Lab3/Delav3	Lab3/Delav7	clk

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.



Timing Analyzer: Lab3

Post Synthesis Timing Paths: Clicking on an instance name highlights corresponding block/subsystem in the model

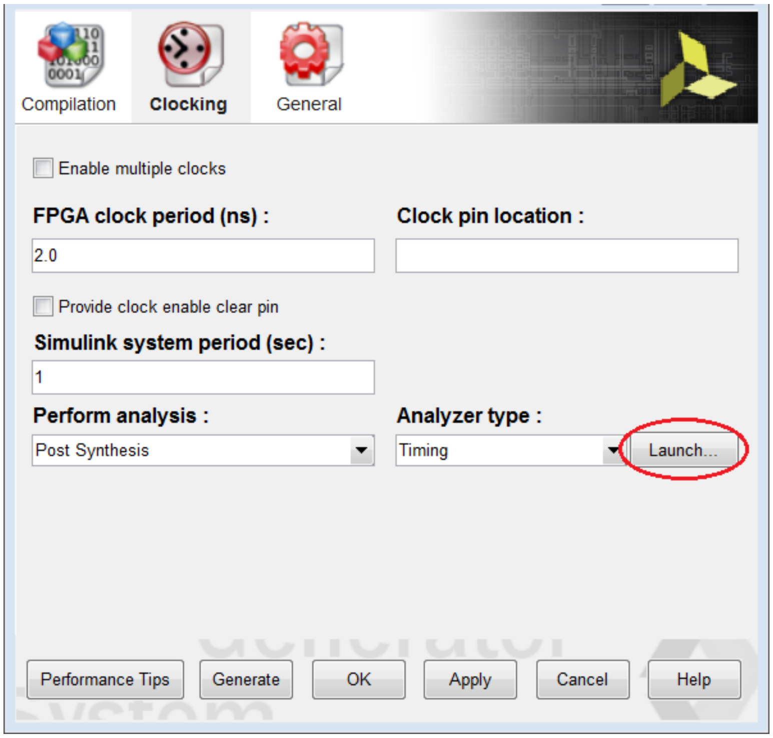
Violation type : setup

Select Columns

Status : FAILED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination
1	-0.8180	2.8060	2.3650	0.4410	13	Lab3/subsystem1/Mult	Lab3/subsystem1/Adc
2	0.6920	0.9730	0.5390	0.4340	0	Lab3/subsystem1/coef	Lab3/subsystem1/Mu
3	0.6960	1.2910	1.0450	0.2460	9	Lab3/addr_gen/Register4	Lab3/addr_gen/Regis
4	0.8120	1.1750	0.6840	0.4910	3	Lab3/addr_gen/Register	Lab3/addr_gen/AddS
5	0.8270	1.1580	0.7520	0.4060	3	Lab3/addr_gen/Register...	Lab3/addr_gen/Relati
6	0.8370	0.6500	0.2160	0.4340	0	Lab3/Delay4	Lab3/subsystem1/coe
7	0.8450	0.9020	0.3350	0.5670	1	Lab3/Delay7	Lab3/Register
8	1.1170	0.8800	0.8800	0	0	Lab3/Delay3	Lab3/Delay3
9	1.2410	0.7460	0.4660	0.2800	2	Lab3/addr_gen/Register	Lab3/addr_gen/Regis
10	1.3880	0.4560	0.2040	0.2520	0	Lab3/Delay3	Lab3/Delay7
11	1.3900	0.3860	0.2160	0.1700	0	Lab3/addr_gen/Relation...	Lab3/Delay3
12	1.4000	0.5740	0.3350	0.2390	1	Lab3/Delay7	Lab3/Register1
13	1.4340	0.4100	0.2160	0.1940	0	Lab3/addr_gen/AddSub1	Lab3/addr_gen/Regis

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.



☐ Enable multiple clocks

FPGA clock period (ns) : 2.0

Clock pin location :

☐ Provide clock enable clear pin

Simulink system period (sec) : 1

Perform analysis : Post Synthesis

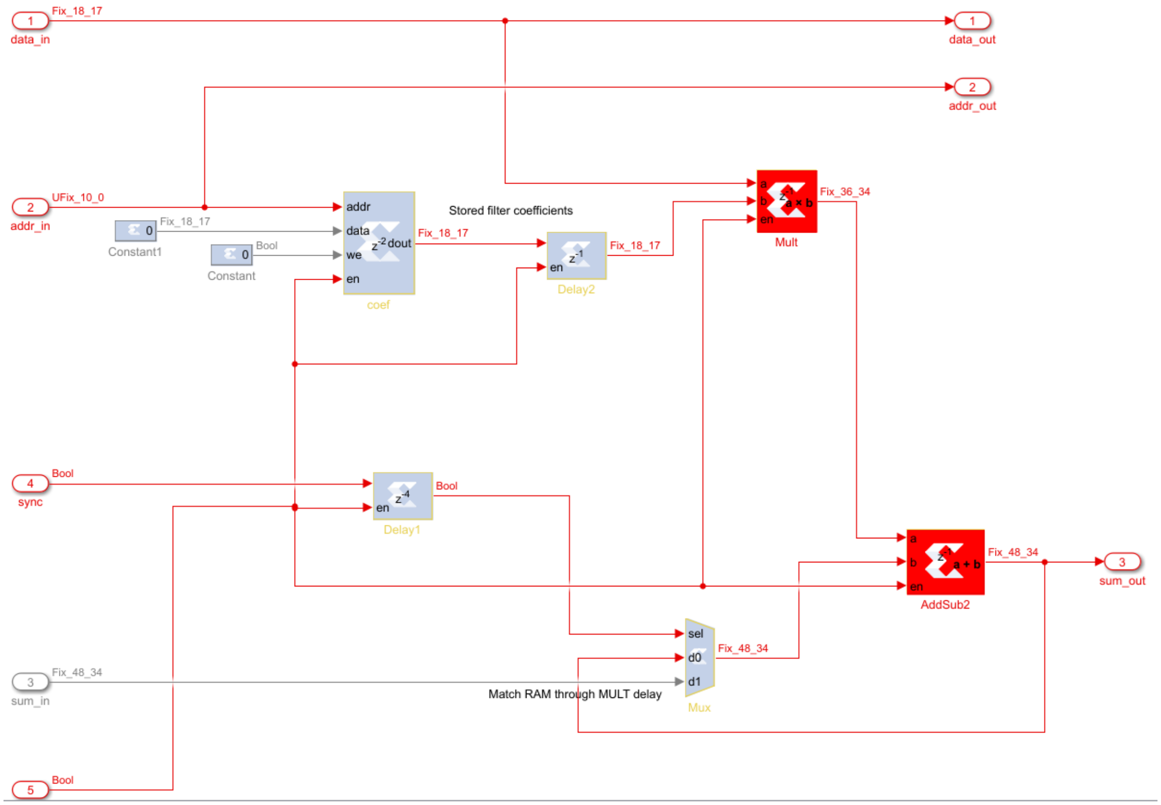
Analyzer type : Timing

***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 in the Target directory specified in the System Generator token dialog box, regardless of the option selected for Perform analysis (Post Synthesis or Post Implementation).

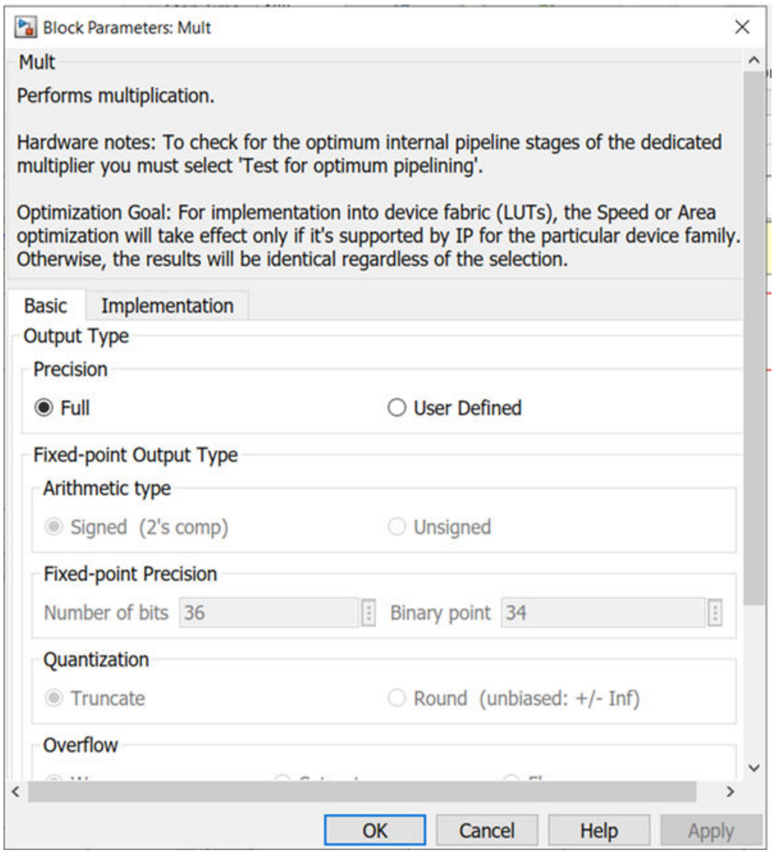
Troubleshooting Timing Violations

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. 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.



Block Parameters: Mult

Mult

Performs multiplication.

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.

Output Type

Precision

● Full ○ User Defined

Fixed-point Output Type

Arithmetic type

● Signed (2's comp) ○ Unsigned

Fixed-point Precision

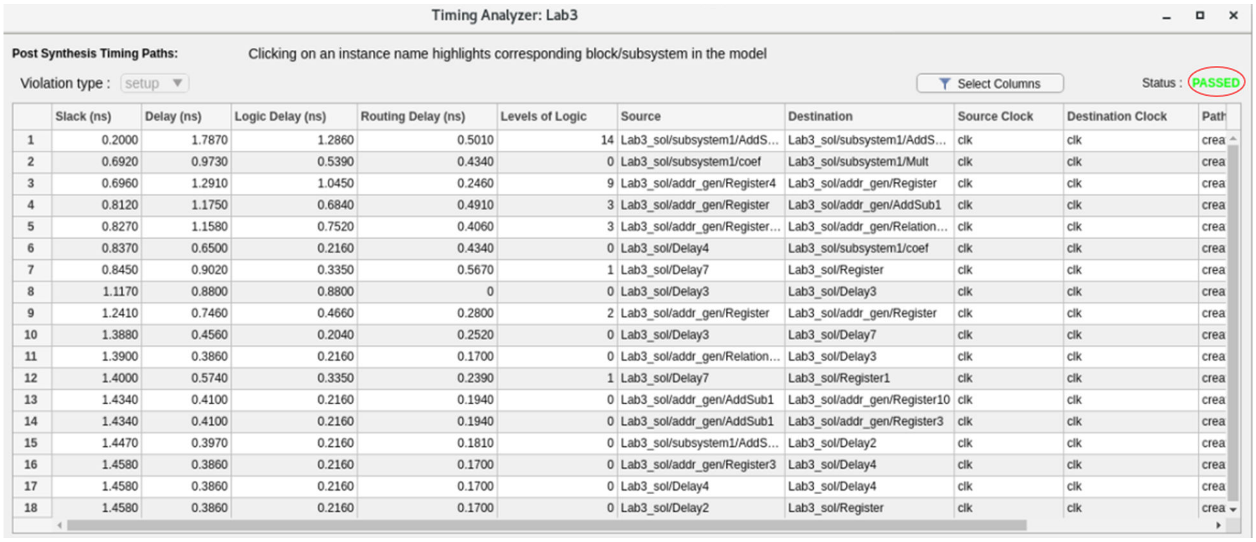
Number of bits 36 Binary point 34

Quantization

● Truncate ○ Round (unbiased: +/- Inf)

Overflow

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.



Timing Analyzer: Lab3

Post Synthesis Timing Paths: Clicking on an instance name highlights corresponding block/subsystem in the model

Violation type : setup

Status : PASSED

	Slack (ns)	Delay (ns)	Logic Delay (ns)	Routing Delay (ns)	Levels of Logic	Source	Destination	Source Clock	Destination Clock	Path
1	0.2000	1.7870	1.2860	0.5010	14	Lab3_sol/subsystem1/AddS...	Lab3_sol/subsystem1/AddS...	clk	clk	crea
2	0.6920	0.9730	0.5390	0.4340	0	Lab3_sol/subsystem1/coef	Lab3_sol/subsystem1/Mult	clk	clk	crea
3	0.6960	1.2910	1.0450	0.2460	9	Lab3_sol/addr_gen/Register4	Lab3_sol/addr_gen/Register	clk	clk	crea
4	0.8120	1.1750	0.6840	0.4910	3	Lab3_sol/addr_gen/Register	Lab3_sol/addr_gen/AddSub1	clk	clk	crea
5	0.8270	1.1580	0.7520	0.4060	3	Lab3_sol/addr_gen/Register...	Lab3_sol/addr_gen/Relation...	clk	clk	crea
6	0.8370	0.6500	0.2160	0.4340	0	Lab3_sol/Delay4	Lab3_sol/subsystem1/coef	clk	clk	crea
7	0.8450	0.9020	0.3350	0.5670	1	Lab3_sol/Delay7	Lab3_sol/Register	clk	clk	crea
8	1.1170	0.8800	0.8800	0	0	Lab3_sol/Delay3	Lab3_sol/Delay3	clk	clk	crea
9	1.2410	0.7460	0.4660	0.2800	2	Lab3_sol/addr_gen/Register	Lab3_sol/addr_gen/Register	clk	clk	crea
10	1.3880	0.4560	0.2040	0.2520	0	Lab3_sol/Delay3	Lab3_sol/Delay7	clk	clk	crea
11	1.3900	0.3860	0.2160	0.1700	0	Lab3_sol/addr_gen/Relation...	Lab3_sol/Delay3	clk	clk	crea
12	1.4000	0.5740	0.3350	0.2390	1	Lab3_sol/Delay7	Lab3_sol/Register1	clk	clk	crea
13	1.4340	0.4100	0.2160	0.1940	0	Lab3_sol/addr_gen/AddSub1	Lab3_sol/addr_gen/Register10	clk	clk	crea
14	1.4340	0.4100	0.2160	0.1940	0	Lab3_sol/addr_gen/AddSub1	Lab3_sol/addr_gen/Register3	clk	clk	crea
15	1.4470	0.3970	0.2160	0.1810	0	Lab3_sol/subsystem1/AddS...	Lab3_sol/Delay2	clk	clk	crea
16	1.4580	0.3860	0.2160	0.1700	0	Lab3_sol/addr_gen/Register3	Lab3_sol/Delay4	clk	clk	crea
17	1.4580	0.3860	0.2160	0.1700	0	Lab3_sol/Delay4	Lab3_sol/Delay4	clk	clk	crea
18	1.4580	0.3860	0.2160	0.1700	0	Lab3_sol/Delay2	Lab3_sol/Register	clk	clk	crea

***Note*:**

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

Step 2: Resource Analysis in Vitis Model Composer

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

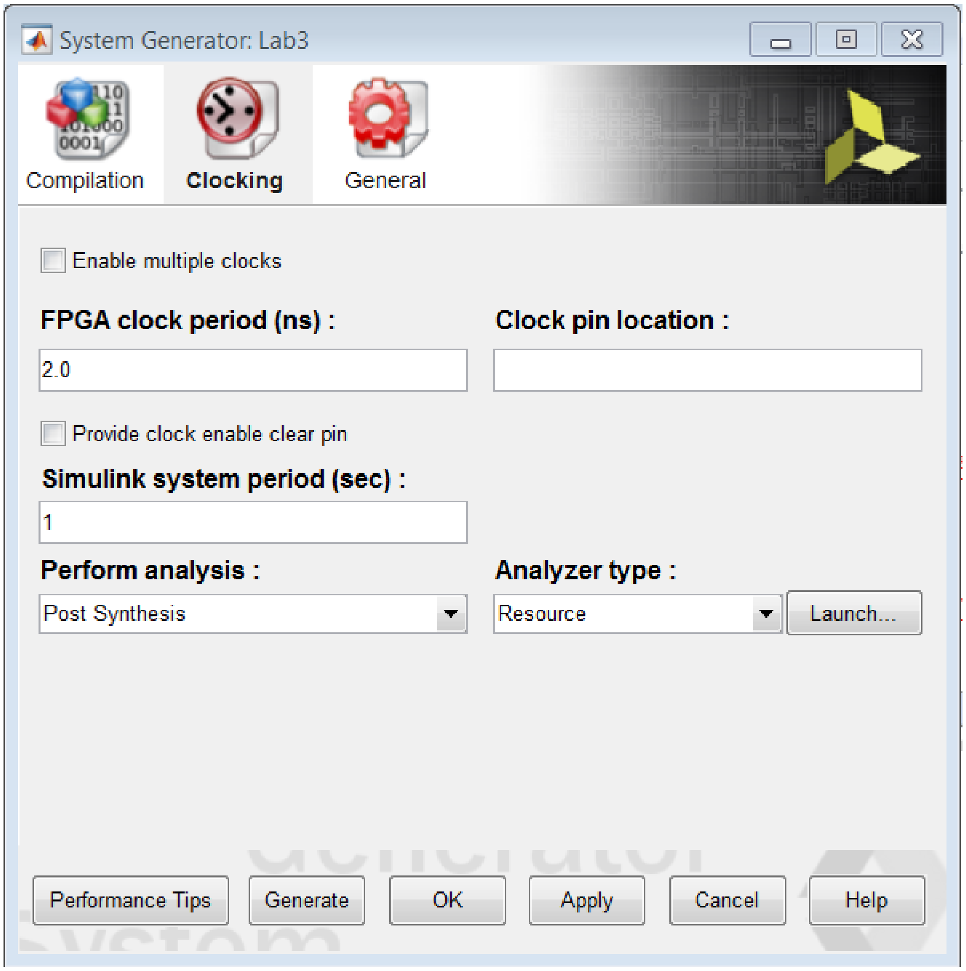
TIP: *Resource Analysis can be performed whenever you generate any of the following compilation targets:*

- 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**.



☐ Enable multiple clocks

FPGA clock period (ns) : 2.0

Clock pin location :

☐ Provide clock enable clear pin

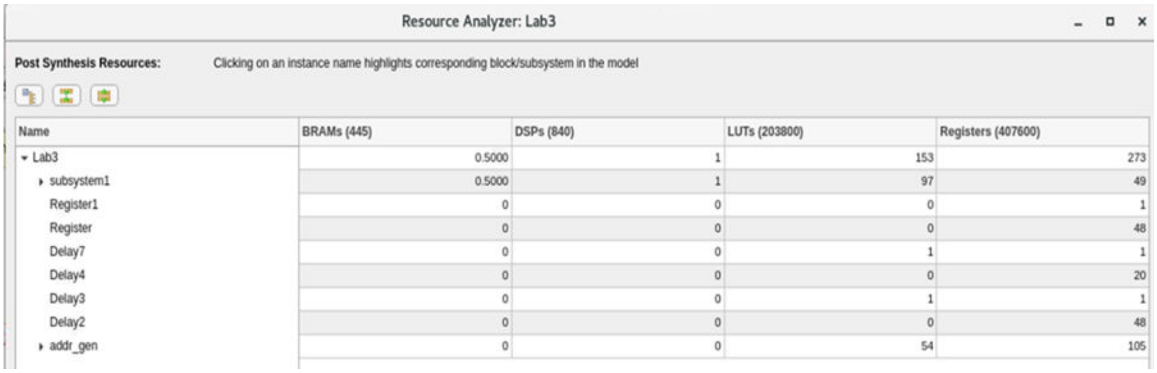
Simulink system period (sec) : 1

Perform analysis : Post Synthesis

Analyzer type : Resource

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

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



Resource Analyzer: Lab3

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

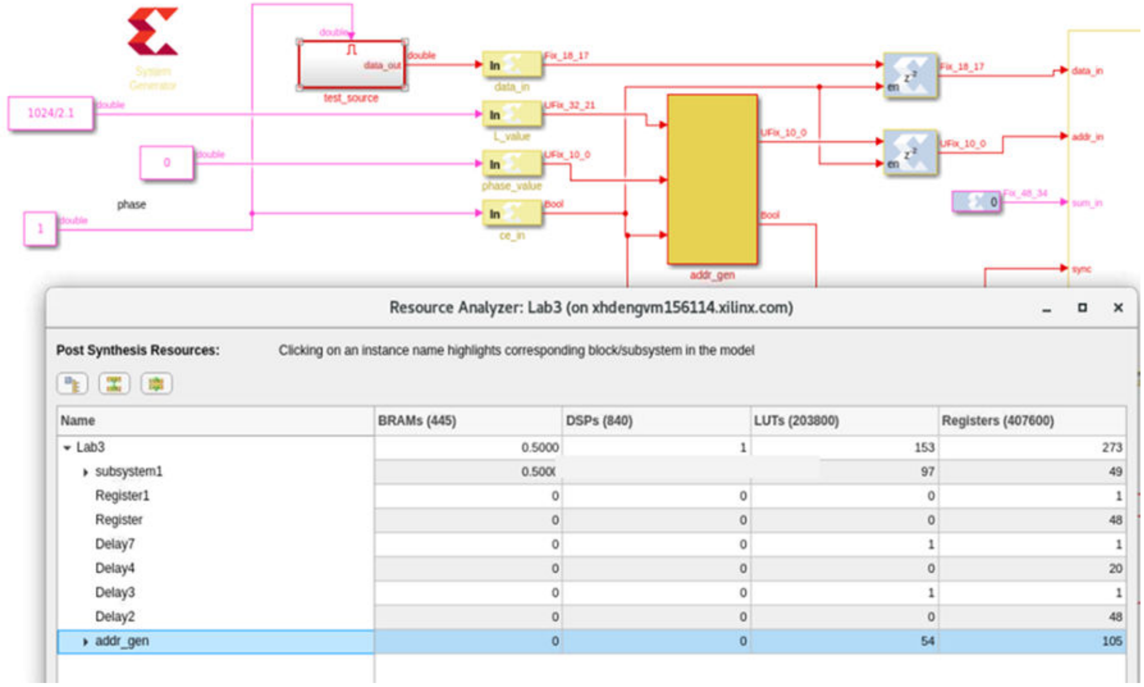
Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▾ Lab3	0.5000	1	153	273
▸ subsystem1	0.5000	1	97	49
Register1	0	0	0	1
Register	0	0	0	48
Delay7	0	0	1	1
Delay4	0	0	0	20
Delay3	0	0	1	1
Delay2	0	0	0	48
▸ addr_gen	0	0	54	105

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 Vitis 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.

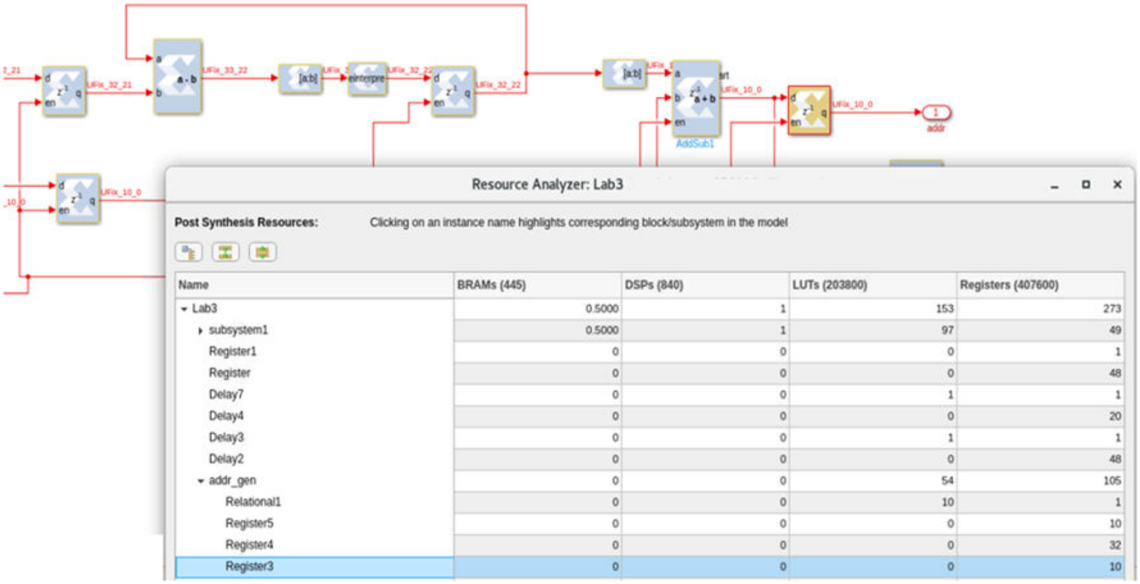


Resource Analyzer: Lab3 (on xhdengvm156114.xilinx.com)

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▾ Lab3	0.5000	1	153	273
▸ subsystem1	0.500(		97	49
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
Delay2	0	0	0	48
▸ addr_gen	0	0	54	105

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.



Resource Analyzer: Lab3

Post Synthesis Resources: Clicking on an instance name highlights corresponding block/subsystem in the model

Name	BRAMs (445)	DSPs (840)	LUTs (203800)	Registers (407600)
▾ Lab3	0.5000	1	153	273
▸ subsystem1	0.5000	1	97	49
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
Delay2	0	0	0	48
▾ addr_gen	0	0	54	105
Relational1	0	0	10	1
Register5	0	0	0	10
Register4	0	0	0	32
Register3	0	0	0	10

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.