

Processor and FPGA Synchronization

In the HDL Workflow Advisor, you can choose a **Processor/FPGA synchronization mode** for your processor and FPGA when you:

- Generate a custom IP core to use in an embedded system integration project.
- Use the **Simulink Real-Time FPGA I/O** workflow.

The following synchronization modes are available:

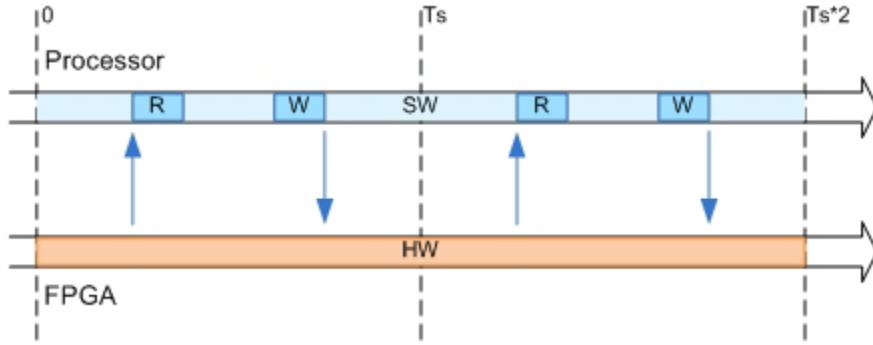
- Free running (default)
- Coprocessing – blocking
- Coprocessing – nonblocking with delay (available only for the **Simulink Real-Time FPGA I/O** workflow)

Free Running Mode

In free running mode, the processor and FPGA each run nonsynchronized, continuously, and in parallel.

Select **Free running** as the **Processor/FPGA synchronization mode** when you do not want your processor and FPGA to be automatically synchronized.

The following diagram shows how the processor and FPGA can communicate in free running mode. The shaded areas indicate that the processor and FPGA are running continuously.

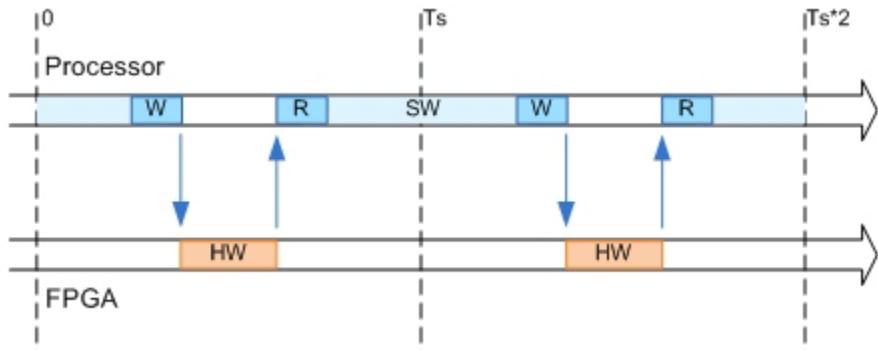


Coprocessing - Blocking Mode

In blocking coprocessor mode, HDL Coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem.

Select **Coprocessing - blocking** as the **Processor/FPGA synchronization mode** when FPGA execution time is short relative to the processor sample time, and you want the FPGA to complete before the processor continues.

The following diagram shows how the processor and FPGA run in blocking coprocessing mode.



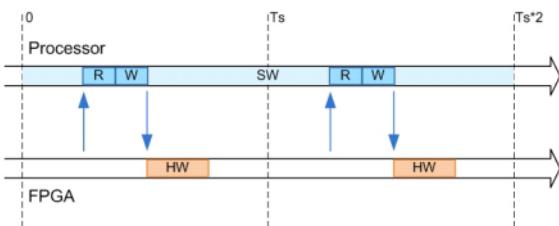
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor writes to the FPGA, then stops and waits for an indication that the FPGA has finished processing before continuing to run. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Coprocessing - Nonblocking With Delay Mode

In delayed nonblocking coprocessor mode, HDL Coder automatically generates synchronization logic for the FPGA so that the processor and FPGA run in tandem. This mode is only available to Speedgoat IO modules that use Xilinx ISE with the **Simulink Real-Time FPGA I/O** workflow.

Select **Coprocessing - nonblocking with delay** as the **Processor/FPGA synchronization mode** when the FPGA processing time is long relative to the processor sample time, or you do not want the processor to wait for the FPGA to finish before the processor continues to run.

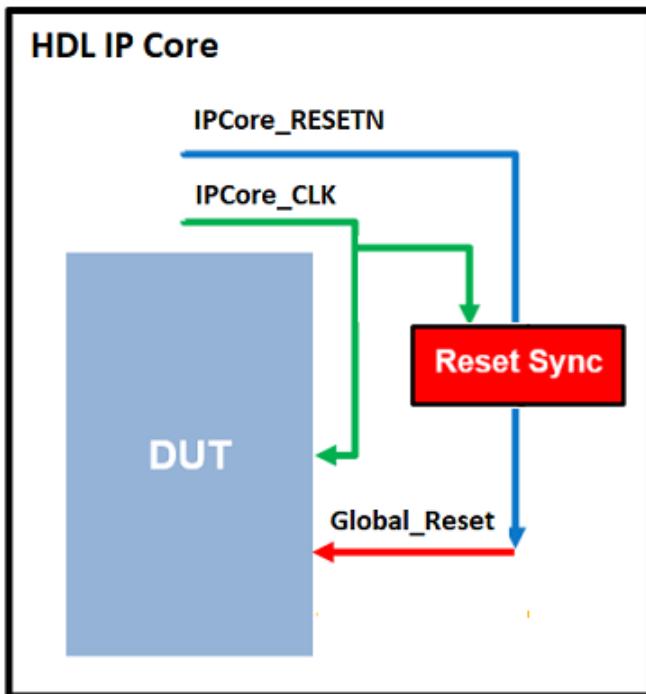
The following diagram shows how the processor and FPGA run in delayed nonblocking coprocessor mode.



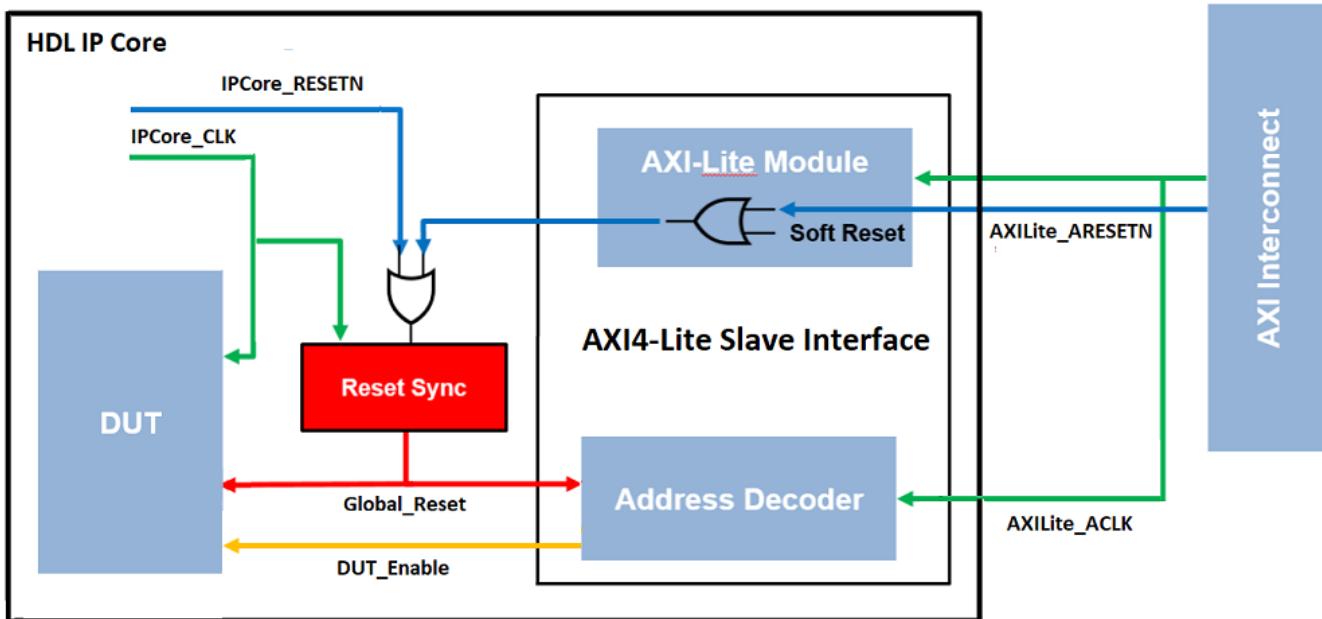
The shaded areas indicate when the processor and FPGA are running. During each sample time, the processor reads FPGA data from the previous sample time, then writes to the FPGA and continues to run without waiting for the FPGA to finish. Each time the FPGA runs, it executes the logic generated for one DUT subsystem sample time.

Synchronization of Global Reset Signal to IP Core Clock Domain

The HDL DUT IP core and the Address Decoder logic in the AXI4 Slave interface wrapper of the HDL IP core are driven by a global reset signal. If you generate an HDL IP core without any AXI4 slave interfaces, HDL Coder does not generate the AXI4 slave interface wrapper. The global reset signal becomes the same as the IP core reset signal and drives the HDL IP core for the DUT. To learn how you can generate an IP core without AXI4 slave interfaces, see “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19.



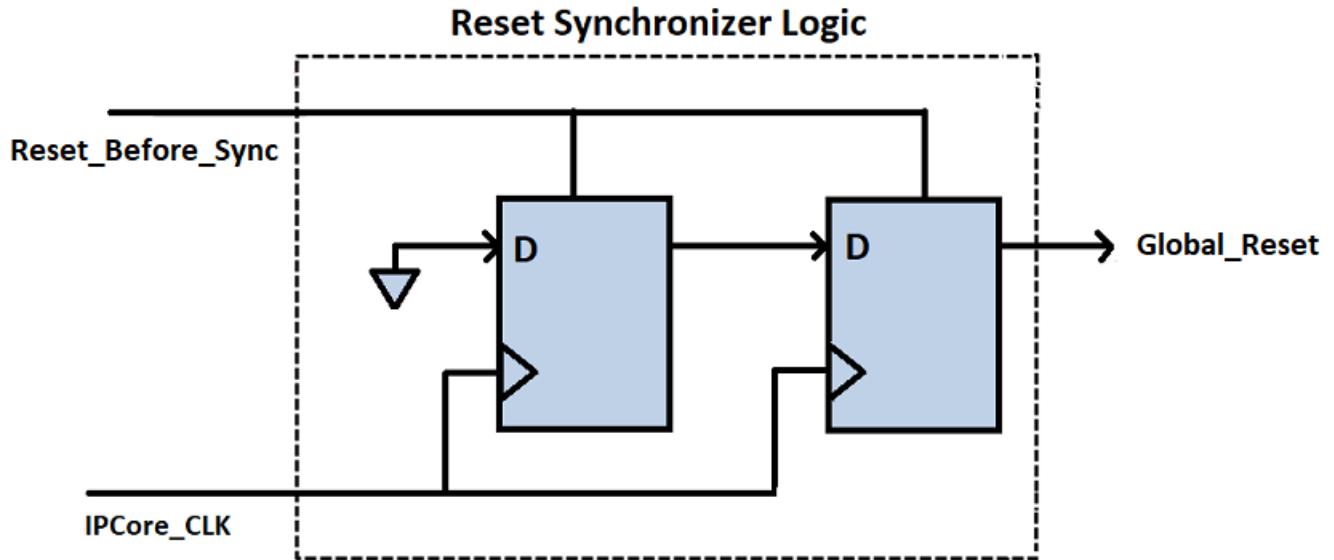
When you generate the AXI4 slave interfaces in the HDL IP core, the global reset signal is driven by three reset signals: the IP core external reset, the reset signal of the AXI interconnect, and the soft reset for the ARM processor core. The global reset signal in this case drives the HDL IP core for the DUT and the Address Decoder logic in the AXI4 slave wrapper.



The **IPCore_CLK** and **AXILite_ACLK** must be connected to the same clock source. The **IPCore_RESETN** and **AXILite_ARESETN** must be connected to the same reset source.

These reset signals can be either synchronous or asynchronous. Using asynchronous reset signals can be problematic and result in potential metastability issues in flipflops when the reset de-asserts within the latching window of the clock. To avoid generation of possible metastable values when combining the reset signals, HDL Coder automatically inserts a reset synchronization logic, as indicated by the **Reset Sync** block. The reset synchronization logic synchronizes the global reset signal to the IP core clock domain. This logic is inserted when you open the HDL Workflow Advisor and run the **Generate RTL Code and IP Core** task of the IP Core Generation workflow.

The reset synchronization logic contains two back-to-back flipflops that are synchronous to the **IPCore_CLK** signal. The flipflops make sure that de-assertion of the reset signal occurs after two clock cycles of when the **IPCore_CLK** signal becomes high. This synchronous de-assertion avoids generation of a global reset signal that has possible metastable values.



The logic works differently depending on whether you specify the **Reset type** as **Synchronous** or **Asynchronous** on the model. If your **Reset type** is **Asynchronous**, the synchronization logic asserts the reset signal asynchronously and de-asserts the reset signal synchronously. For example, this code illustrates the generated Verilog code for the reset synchronization logic when you generate the IP core with asynchronous reset.

```

...
...
reg_reset_pipe_process : PROCESS (clk, reset_in)
BEGIN
  IF reset_in = '1' THEN
    reset_pipe <= '1';
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      reset_pipe <= const_0;
    END IF;
  END IF;
END PROCESS reg_reset_pipe_process;

reg_reset_delay_process : PROCESS (clk, reset_in)
BEGIN
  IF reset_in = '1' THEN
    reset_out <= '1';
  ELSIF clk'EVENT AND clk = '1' THEN
    IF enb = '1' THEN
      reset_out <= reset_pipe;
    END IF;
  END IF;
END PROCESS reg_reset_delay_process;
  
```

```
END rtl;
```

If your **Reset type** is **Synchronous**, the synchronization logic asserts and de-asserts the reset signal synchronously. For example, this code illustrates the generated Verilog code for the reset synchronization logic when you generate the IP core with synchronous reset.

```
...
...
reg_reset_pipe_process : PROCESS (clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF reset_in = '1' THEN
      reset_pipe <= '1';
    ELSIF enb = '1' THEN
      reset_pipe <= const_0;
    END IF;
  END IF;
END PROCESS reg_reset_pipe_process;

reg_reset_delay_process : PROCESS (clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF reset_in = '1' THEN
      reset_out <= '1';
    ELSIF enb = '1' THEN
      reset_out <= reset_pipe;
    END IF;
  END IF;
END PROCESS reg_reset_delay_process;

END rtl;
```

See Also

More About

- “Custom IP Core Generation” on page 40-10
- “Custom IP Core Report” on page 40-13
- “Processor and FPGA Synchronization” on page 40-23

IP Caching for Faster Reference Design Synthesis

In this section...

- “Requirements for Using IP Caching” on page 40-29
- “What Is an IP Cache?” on page 40-29
- “How IP Caching Works” on page 40-30
- “Enable IP Caching” on page 40-30
- “IP Caching in HDL Coder Reference Designs” on page 40-31
- “IP Caching in Custom Reference Designs” on page 40-32

For target platforms that support the IP Core Generation workflow with Xilinx Vivado, you can use IP caching. IP caching reduces the synthesis time of reference designs that have many IP modules or that have IP modules with a significant synthesis run time. When you enable IP caching, the Vivado project uses an out-of-context (OOC) workflow. This workflow synthesizes the IP in the reference design out of context from the top-level design. The OOC workflow accelerates project runs because the synthesis tool reuses the IP cache, and does not have to resynthesize the IP when you run the workflow.

If you do not enable IP caching, by default, the Vivado project uses the global synthesis flow. This flow synthesizes the IP modules in the reference design along with the top-level design. In subsequent project runs, this workflow resynthesizes the IP modules in the reference design.

Requirements for Using IP Caching

- **Target workflow:**
 - IP Core Generation
 - Simulink Real-Time FPGA I/O for Speedgoat boards that use Xilinx Vivado
- **Synthesis tool:** Xilinx Vivado

What Is an IP Cache?

An IP cache is a folder that consists of subfolders corresponding to IP modules in the reference design. Each subfolder is organized by a hash index that corresponds to the file name. For each IP module, the subfolder consists of Xilinx Core Instance (XCI) files, Design Checkpoint (DCP) files, and synthesis log files. The DCP is a container file that contains synthesized netlists, black box HDL stub files, and the output clock constraints.

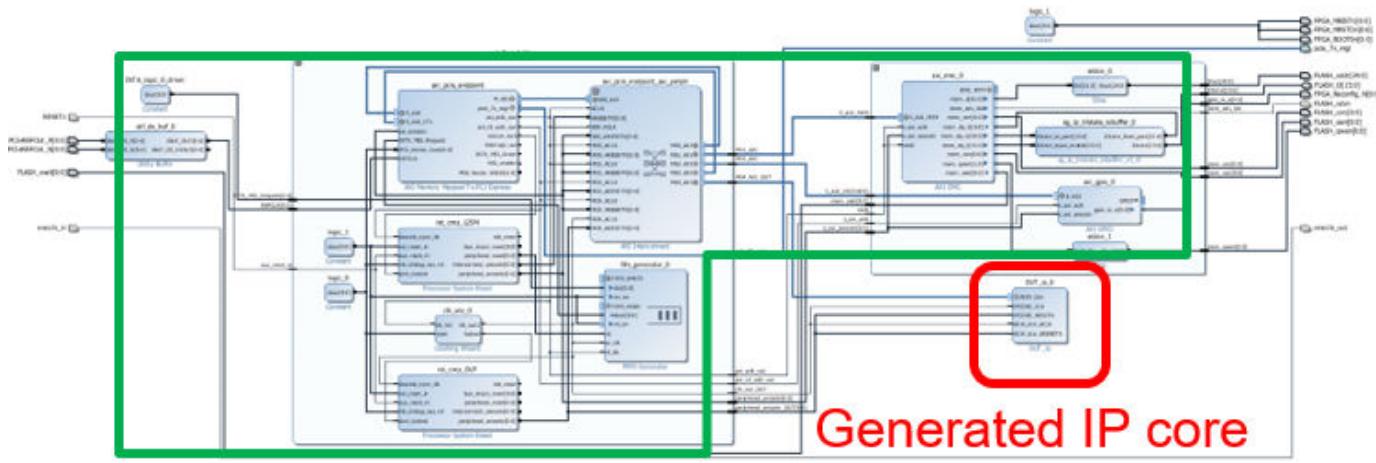
To reuse the IP cache when you run the workflow, the IP synthesis has to match the hash index in the IP cache. The hash index match corresponds to a hit in the IP cache. To hit the IP cache in subsequent runs, use the same:

- Part, language, and target platform settings
- Reference design version
- Target frequency
- `hdl_prj` folder when you created the IP cache

How IP Caching Works

When you enable IP caching, the Xilinx Vivado project uses an out-of-context (OOC) workflow. The OOC design flow is a bottom-up workflow that:

- 1 Synthesizes the IP modules in the reference design separately from the top-level design. The synthesis output is the Design Checkpoint (DCP) file.
- 2 Synthesizes your top-level design while treating the IP in the reference design as a black box by using the HDL stub files provided with the DCP.
- 3 Implements your design on the target device by linking the netlists from the IP design checkpoint files with your top-level netlist.



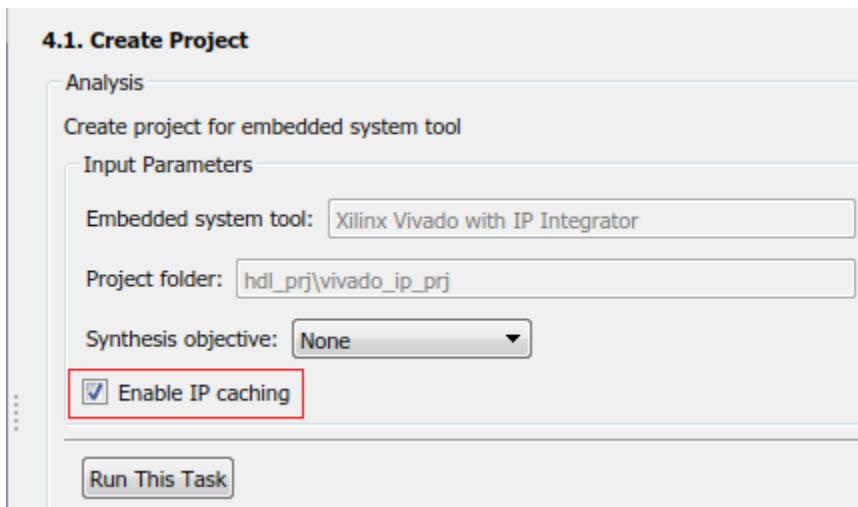
No need to re-synthesize

For large reference designs, the OOC flow improves synthesis run time, because you do not have to resynthesize the IP when you modify your design and run the workflow. To learn more about the OOC workflow and IP synthesis options, refer to the Xilinx documentation.

Enable IP Caching

Before you enable IP caching, specify IP Core Generation as the target workflow, and then specify the target platform settings. To enable IP caching:

- From the HDL Workflow Advisor, in the **Create Project** task, select the **Enable IP caching** check box.



- From the command line, use the `EnableIPCaching` property of the `hdlcoder.WorkflowConfig` class. To use this property, create an object of the `hdlcoder.WorkflowConfig` class, or export the HDL Workflow Advisor settings to a script.

```
hWC = hdlcoder.WorkflowConfig('SynthesisTool','Xilinx Vivado','TargetWorkflow','IP Core Generation');
%
%
hWC.EnableIPCaching = true;
```

IP Caching in HDL Coder Reference Designs

Use IP caching for large reference designs that have a significant synthesis time. For example, the HDL Coder reference design `Default video system (requires HDMI FMC module)` is a potential candidate for IP caching.

Note The Speedgoat I0333-325K board that you use with the `Simulink Real-Time FPGA I/O` workflow comes with an IP cache. The first time that you run the workflow, the code generator reuses this IP cache, which improves reference design synthesis time.

To enable IP caching, in the HDL Workflow Advisor, specify `IP Core Generation` as the target workflow, and then specify the target platform settings. Before you run the workflow for the first time:

- In the **Create Project** task, select the **Enable IP caching** check box.

When you run this task, the workflow creates an empty IP cache folder. You can see the `ipcache` folder in the `hdl_prj/vivado_ip_prj` path.

- Run the **Build FPGA Bitstream** task.

This task populates the IP cache folder with synthesis logs and design checkpoint files generated for the HDL IP core and other IP blocks in the reference design. When this task has run successfully, you can see the generated files in the `ipcache` folder.

When you run the `IP Core Generation` workflow a second time, in the **Build FPGA Bitstream** task, you can see an improvement in the task run time. Make sure that you use the same IP settings

and `hdl_prj` folder as the first time that you ran the workflow. When this task has run successfully, to see if your workflow reused the IP cache, open the `workflow_task_BuildFPGABitstream.log` file.

The screenshot shows the 'Build FPGA Bitstream' task in the HDL Workflow Advisor. The task is listed under 'Analysis' with the sub-task 'Synthesis and generate bitstream for embedded system on FPGA'. Under 'Input Parameters', there is a checkbox 'Run build process externally' which is unchecked. Below it is a dropdown 'Tcl file for synthesis build' set to 'Default' with a 'Browse...' button. A large 'Run This Task' button is at the bottom of the task panel. The result is shown as 'Passed' with a green checkmark. Below the result, a message says 'Passed Build Embedded System.' and 'Synthesis Tool Log:' followed by the log output.

Result: Passed

Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.
Generated logfile: hdl_prj\hdlsrc\xaxi_video_basic\workflow task BuildFPGABitstream.log
WARNING: Default location for XILINX_VIVADO_HLS not found:

***** Vivado v2016.2 (64-bit)
***** SW Build 1577090 on Thu Jun 2 16:32:40 MDT 2016
***** IP Build 1577682 on Fri Jun 3 12:00:54 MDT 2016
** Copyright 1986-2016 Xilinx, Inc. All Rights Reserved.
```

This code snippet shows that the Vivado project launches a maximum number of jobs to synthesize the design and reuse the IP modules in the IP cache folder. You can see that the `cacheID` of the IP modules match the file names of the subfolders in the `ipcache` folder.

```
...
# reset_run impl_1
# reset_run synth_1
# launch_runs -jobs 4 synth_1
...
...
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_RGBtoYCbCr_0_0, cacheID = 3575924730488800
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_YCbCrtoRGB_0_0, cacheID = e71459f41e26e141
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_xbar_0, cacheID = d0f0971cb77bcaed
INFO: [IP_Flow 19-4760] Using cached IP synthesis design for IP system_top_axis2hdmi_0_0, cacheID = 7601a322f9fd0ec4
...
```

IP Caching in Custom Reference Designs

If you are using your own custom reference design, IP caching can accelerate reference design synthesis when you run the workflow for the first time. To reuse the IP cache, create an IP cache zip file, and then make sure that the reference design definition file points to this zip file.

To create an IP cache zip file:

- 1 Open the HDL Workflow Advisor for any Simulink model that has a DUT subsystem, and then run the **IP Core Generation** workflow to the **Generate RTL Code and IP Core** task.

- 2 In the **Create Project** task, select the **Enable IP caching** check box, and then click **Run This Task**. This task creates an empty IP cache folder.
- 3 Run the workflow to the **Build FPGA Bitstream** task. This task populates the IP cache with the HDL IP core and the reference design IP modules.
- 4 In the IP cache folder, delete the IP core files generated for the DUT. Extract the remaining files from this folder into a zip file, name it `ipcache.zip`, and then save the file in the reference design folder.

To reuse the IP cache, in the reference design definition file `plugin_rd.m`, use the `IPCacheZipFile` property of the `hdlcoder.ReferenceDesign` class. By using that property, you add the `ipcache.zip` file to the Xilinx Vivado project.

```
function hRD = plugin_rd()
% Reference design definition

hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
%
%
hRD.IPCacheZipFile = 'ipcache.zip';
```

When you use the workflow to target your custom reference design, the code generator selects the **Enable IP caching** check box. To see the improvement in synthesis time, run the **Build FPGA Bitstream** task.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

More About

- “Custom IP Core Generation” on page 40-10
- “Custom IP Core Report” on page 40-13
- “Board and Reference Design Registration System” on page 41-39
- “Run HDL Workflow with a Script” on page 31-53

Resolve Timing Failures in IP Core Generation and Simulink Real-Time FPGA I/O Workflows

In this section...

["Step 1: Identify the Timing Failure" on page 40-34](#)

["Step 2: Find the Critical Path" on page 40-37](#)

["Step 3: Resolve Timing Failures" on page 40-41](#)

Synchronous circuits require that data propagates from a source register to a destination register within one clock cycle. For the synthesis tools, the Delay blocks that you add to your Simulink model run at the clock rate. The tools require data to travel between the blocks within one clock cycle. If the tool is unable to propagate the data between the registers for one or more signal paths in your model within one clock cycle, a timing failure occurs.

The tools report a slack information for each signal path, which corresponds to the difference between the required time and the arrival time. Required time is the expected time at which a signal must arrive at the destination register. Arrival time is the time elapsed for a signal to arrive at that point. A positive slack indicates that the signal arrived much faster than the required time, and the path passes the timing requirement. A negative slack indicates that the signal path is slower than the required time, and the path fails the timing requirement. To make sure that your design meets the timing requirements, speed up all signal paths that have a negative slack.

To identify if your design meets the timing requirements and how you can resolve timing failures, perform these steps.

Step 1: Identify the Timing Failure

When you run the IP Core Generation workflow or the Simulink Real-Time FPGA I/O workflow for Vivado-based boards, if your Simulink model does not meet the timing requirements, HDL Coder generates an error in the **Build FPGA Bitstream** step of the workflow. See:

- “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 for information about how to run the IP Core Generation workflow.
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63 for information about how to run the Simulink Real-Time FPGA I/O workflow. When you run this workflow, use a Speedgoat board that supports Xilinx Vivado as the synthesis tool.

When you run the **Build FPGA Bitstream** task, if you left the **Run build process externally** check box selected by default, whether or not there is a timing failure, HDL Coder displays the results as **Passed** and provides warning messages. View the build log in the external console to identify if there is a potential timing failure.

4.3. Build FPGA Bitstream

Analysis
Synthesis and generate bitstream for embedded system on FPGA

Input Parameters

Run build process externally

Tcl file for synthesis build: Default

Run This Task

Result:  Passed

Warning Run build process externally: The system build has been launched in an external shell, please check the external console to make sure the bitstream generation is completed. The system build may fail if the timing constraints are not met by the synthesis tool. If a timing failure occurs, the bitstream will be renamed to "system_timingfailure.sof". For more details on how to resolve the timing failure, please follow the "[Article on timing failure](#)" and also read the "[Timing report](#)" for details. The system build may also fail because of other reasons, please read the log in the external console to identify reason.

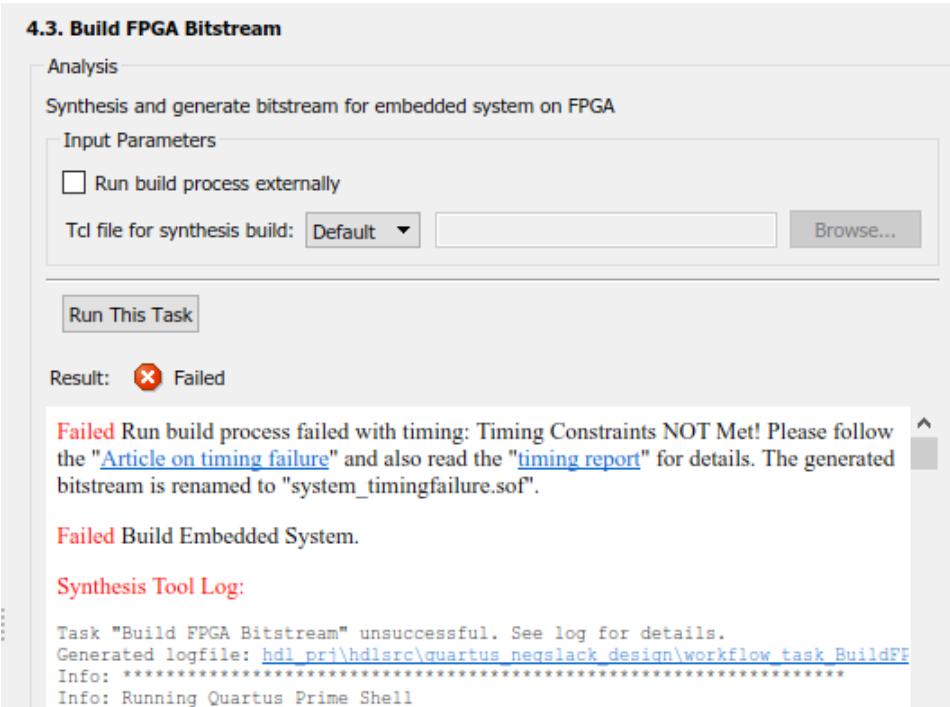
Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.
Generated logfile: hdl\_prj\hdlsrc\quartus\_negslack\_design\workflow\_task\_BuildFPGA
Running embedded system build outside MATLAB.
Please check external shell for system build progress.
```

In the external console, if there is a timing failure, you see the worst slack and this message: **Timing constraints NOT met!**

When you clear the **Run build process externally** check box, and then run the **Build FPGA Bitstream** task, if a timing failure occurs, the task fails, and you see these messages in the **Result** subpane.



In both cases, when there is a timing failure, the code generator replaces the previous bitstream with a bitstream that has the same name and the postfix `_timingfailure.bit` or `_timingfailure.sof` depending on whether you created a project by using Vivado or Quartus. For example, if the previous generated bitstream was called `system_top_wrapper.bit`, and if there is a timing failure, HDL Coder renames this bitstream to `system_top_wrapper_timingfailure.bit`.

If you run the **Program Target Device** task, the task fails.

Report Timing Failures as Warnings

If you have already implemented the custom logic to resolve the timing failures, you can specify the timing failures to be reported as warnings instead of errors. You can then continue the workflow and generate the FPGA bitstream. Before programming the target SoC device, it is recommended that you have resolved the timing failures.

After you have resolved the timing failures, to verify that the failures have been resolved, you can use the HDL Coder software. Change the timing failures to be reported as errors and then rerun the IP Core Generation workflow to ensure that the **Build FPGA Bitstream** task passes. If the **Build FPGA Bitstream** task still fails, perform the steps in the preceding sections to identify and resolve the timing failures.

To specify timing failures to be reported as warnings:

- After you run the **Build FPGA Bitstream** task, export the HDL Workflow Advisor to a script. In the script, to report timing failures as warnings, use the `ReportTimingFailure` property of the `hdlcoder.WorkflowConfig` class. You can then run the script or import the script to the HDL Workflow Advisor and then run the workflow.

```
hWC.ReportTimingFailure = hdlcoder.ReportTiming.Warning;
```

- If you are targeting a custom reference design that you have already defined for the board, to report timing failures as warnings, use the `ReportTimingFailure` property of the `hdlcoder.ReferenceDesign` class.

```
hRD.ReportTimingFailure = hdlcoder.ReportTiming.Warning;
```

To learn how you can identify the critical path and resolve the timing failures, perform the steps in the preceding sections.

Step 2: Find the Critical Path

Critical path is a combinational path between the input and the output that has the maximum delay. This path corresponds to the signal path that has the worst negative slack. By identifying and optimizing the critical path, you can resolve timing failures and improve the timing of your design. You can identify the critical path in your design by using either of these strategies.

Strategy 1: Check the Timing Report

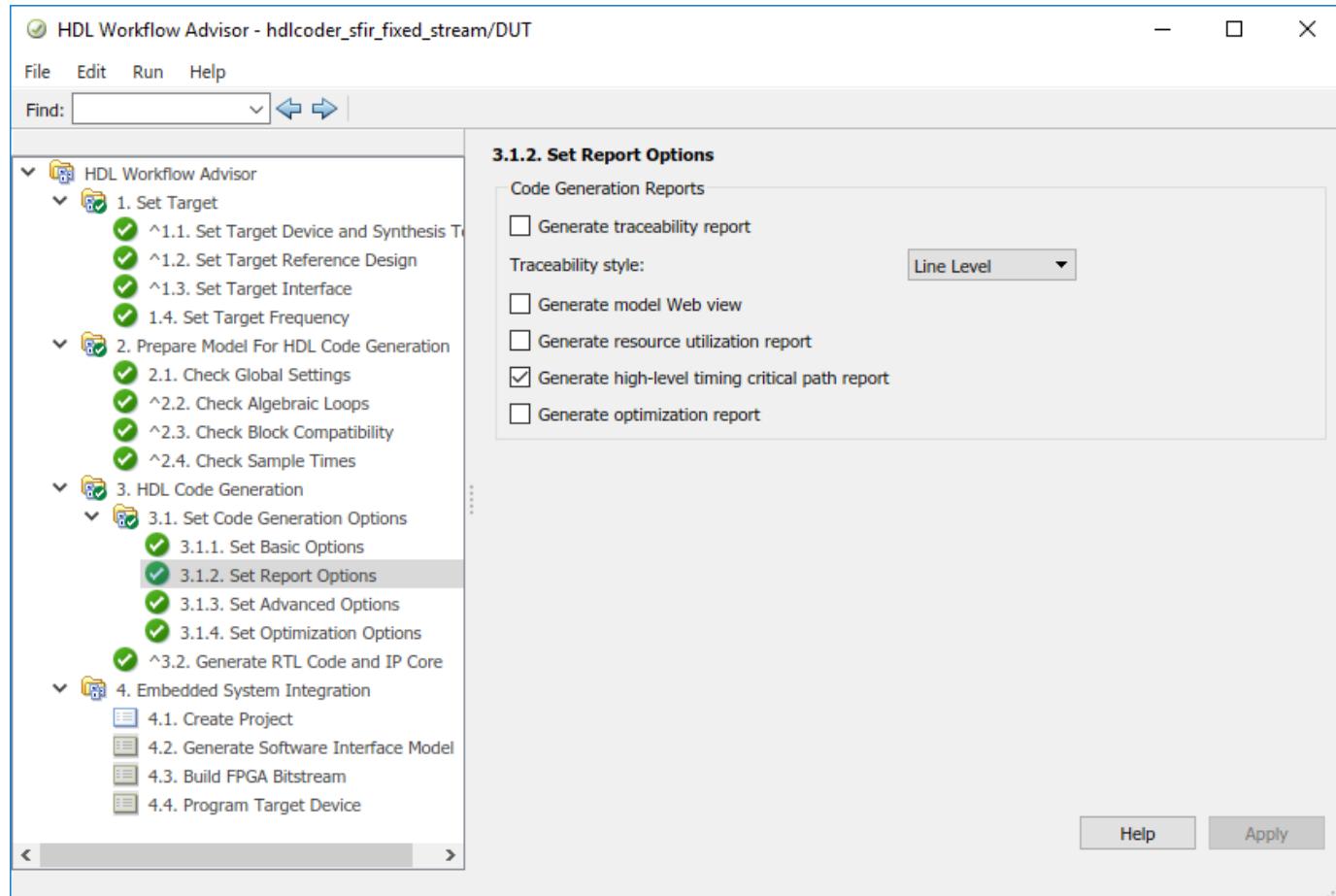
In the **Result** subpane, to open the timing report that is generated by the synthesis tool, select the **timing_report** link. You can use the report to identify the critical path in your design. In the report, if you search for **Worst Slack**, you can identify the worst setup slack. Then, use the **Source** and **Destination** points to identify the critical path. For example, this report for the LED blinking model `hdlcoder_led_blinking` shows that the critical path is inside the HDL Counter block.

```
-----  
From Clock: clk_out1_system_top_clk_wiz_0_0  
To Clock: clk_out1_system_top_clk_wiz_0_0  
  
Setup : 1193 Failing Endpoints, Worst Slack -2.478ns, Total Violation -1226.784ns  
Hold : 0 Failing Endpoints, Worst Slack 0.034ns, Total Violation 0.000ns  
PW : 2 Failing Endpoints, Worst Slack -0.576ns, Total Violation -0.731ns  
-----  
  
Max Delay Paths  
-----  
Slack (VIOLATED) : -2.478ns (required time - arrival time)  
Source: system_top_i/led_count_ip_0/U0/u_led_count_ip_dut_inst/  
u_led_count_ip_src_led_counter/HDL_Counter1_out1_reg[0]/C  
(rising edge-triggered cell FDRE clocked by clk_out1_system_top_clk_wiz_0_0  
{rise@0.000ns fall@1.000ns period=2.000ns})  
Destination: system_top_i/led_count_ip_0/U0/u_led_count_ip_dut_inst/  
u_led_count_ip_src_led_counter/HDL_Counter1_out1_reg[20]/R  
(rising edge-triggered cell FDRE clocked by clk_out1_system_top_clk_wiz_0_0  
{rise@0.000ns fall@1.000ns period=2.000ns})  
Path Group: clk_out1_system_top_clk_wiz_0_0  
Path Type: Setup (Max at Slow Process Corner)  
Requirement: 2.000ns (clk_out1_system_top_clk_wiz_0_0 rise@2.000ns -  
clk_out1_system_top_clk_wiz_0_0 rise@0.000ns)  
Data Path Delay: 3.899ns (logic 1.412ns (36.211%) route 2.487ns (63.789%))
```

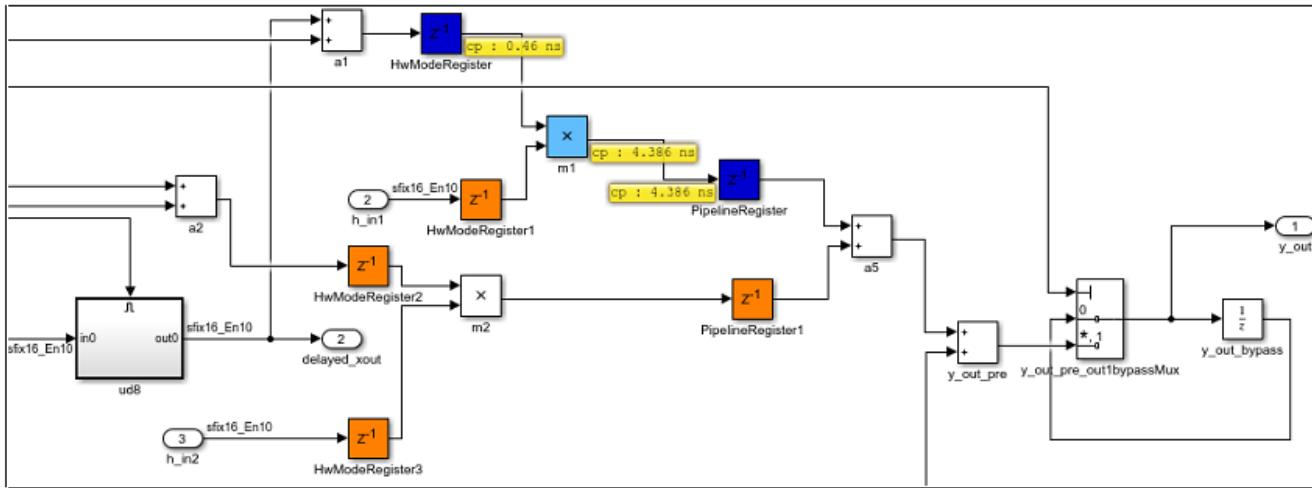
Strategy 2: Estimate Critical Path Without Running Synthesis

Use HDL Coder to estimate and highlight the critical path in your model without synthesizing your design. Critical path estimation identifies the critical path by performing static timing analysis with timing data from target-specific databases. Estimating the critical path without using synthesis tools can lead to inaccurate timing results. Critical path estimation speeds up the process of identifying and optimizing the critical path in your design. It is an alternative to performing **FPGA Synthesis and Analysis** with the HDL Workflow Advisor. To learn more, see “Critical Path Estimation Without Running Synthesis” on page 24-137.

To estimate the critical path, in the **Set Report Options** task, select the **Generate high-level timing critical path report** check box. Run the workflow to the **Generate RTL Code and IP Core** task.



HDL Coder generates a critical path estimation section in the Code Generation Report. On this section, when you select the link to the `criticalpathestimated` script, the code generator highlights the critical path in the generated model. This figure shows a section of the `hdlcoder_sfir_fixed_stream` model with the critical path highlighted.



Strategy 3: Annotate Critical Path By Using Backannotation

For more accurate critical path information and highlighting of critical path in your design, use backannotation. To use backannotation, you have to leave the current Workflow Advisor session, and then run the **Generic ASIC/FPGA** workflow to annotate the model with the synthesis results.

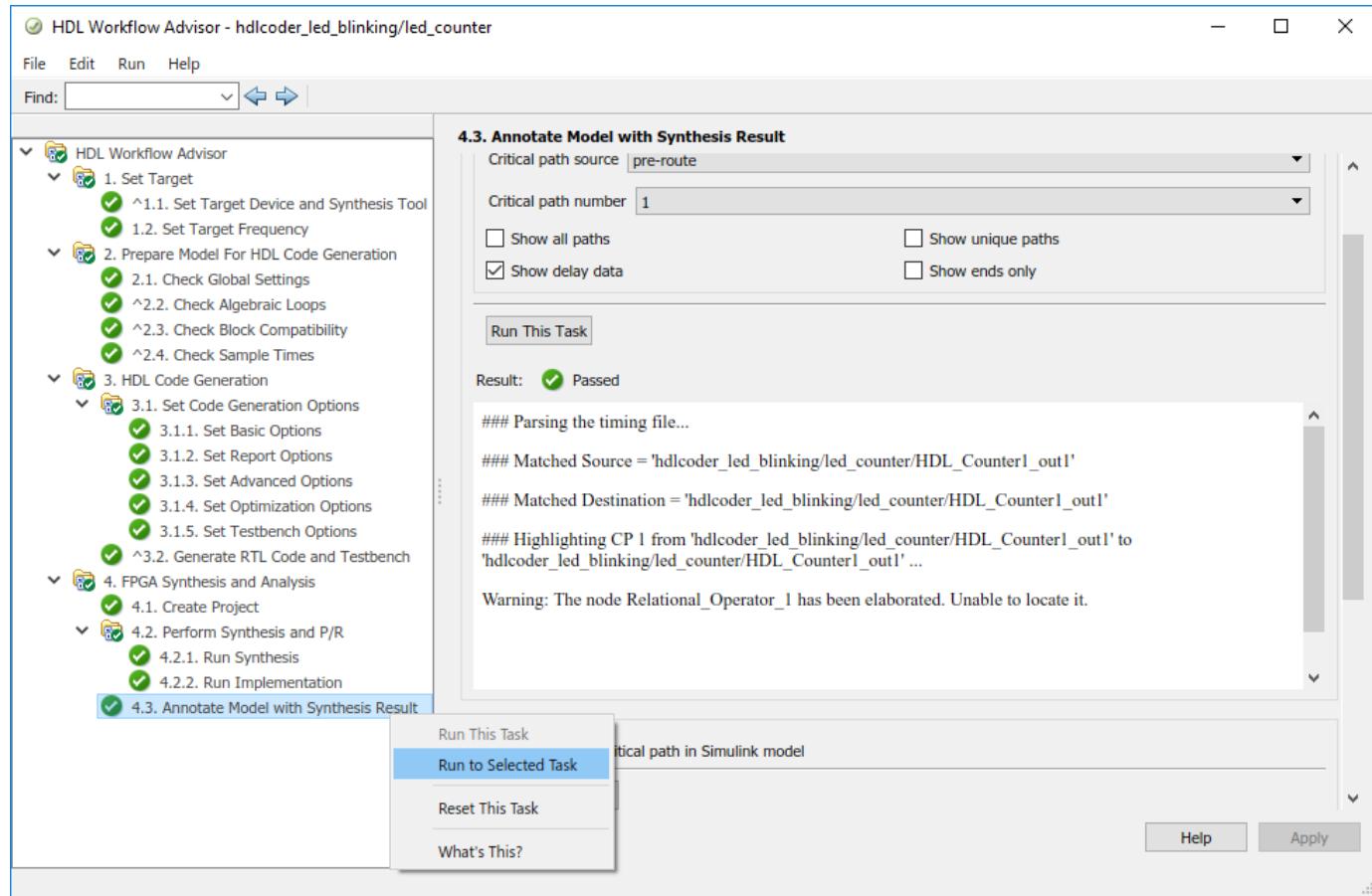
Before you use backannotation, it is recommended that you export the current HDL Workflow Advisor settings to a script. By exporting the settings to a script, you can iterate on the critical path and customize various settings to optimize your design until you meet the timing requirements. You can import the Workflow Advisor script to the HDL Workflow Advisor and then run the workflow. See also “Run HDL Workflow with a Script” on page 31-53.

To use backannotation:

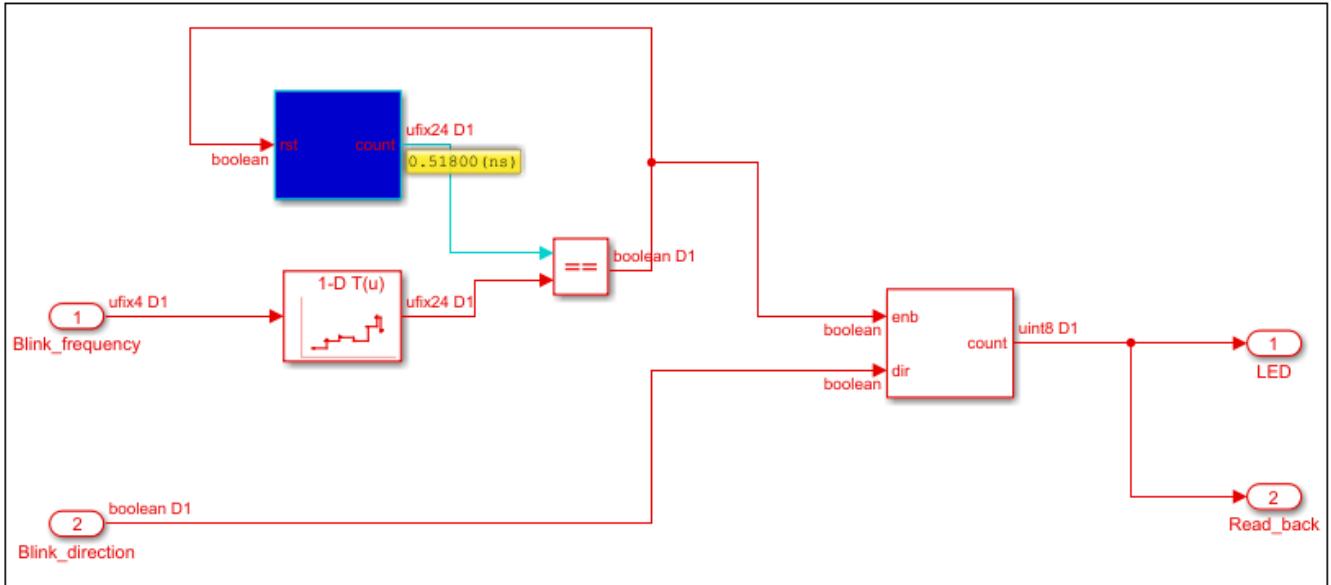
- 1 In the **Set Target Device and Synthesis Tool** task, select **Generic ASIC/FPGA** as the **Target workflow**. For **Synthesis tool**, specify the same tool that you used to run the **IP Core Generation** workflow.

When you specify these settings, HDL Coder resets this task and all tasks that follow it.

- 2 Select **Run This Task**.
- 3 In the **Set Target Frequency** task, specify the same target frequency that you used to run the **IP Core Generation** workflow. Select **Run This Task**.
- 4 Right-click the **Annotate Model with Synthesis Result** task and select **Run to Selected Task**.



When you run the link to the **Annotate Model with Synthesis Result** task, the code generator highlights the critical path in the generated model. This figure shows that the critical path in the `hdlcoder_led_blinking` model is inside the HDL Counter block.



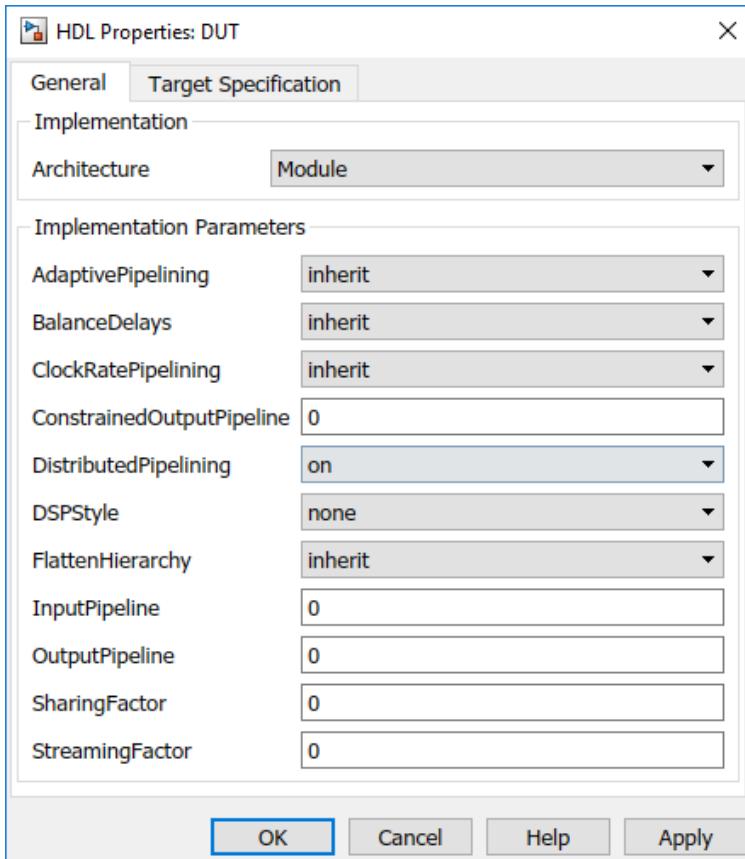
Step 3: Resolve Timing Failures

To resolve timing failures, you can use any of these recommendations or a combination of the recommendations until your design meets the target frequency.

Recommendation 1: Use Speed Optimizations

You can use speed optimizations such as distributed pipelining and clock-rate pipelining to break the critical path by adding pipeline registers while preserving the functional behavior. By reducing the critical path, you can achieve higher clock frequencies and increase the arrival time of the signal until it equals the required time and the slack becomes zero.

Distributed pipelining and hierarchical distributed pipelining optimizations move the existing delays you have in your design across the subsystem hierarchy. When you use hierarchical distributed pipelining enabled, make sure that all Subsystem blocks have DistributedPipelining enabled. If your design does not meet the timing requirements, you can add more pipelines by using **InputPipeline** or **OutputPipeline** block properties. You can specify these properties in the HDL Block Properties dialog box of the Subsystem.

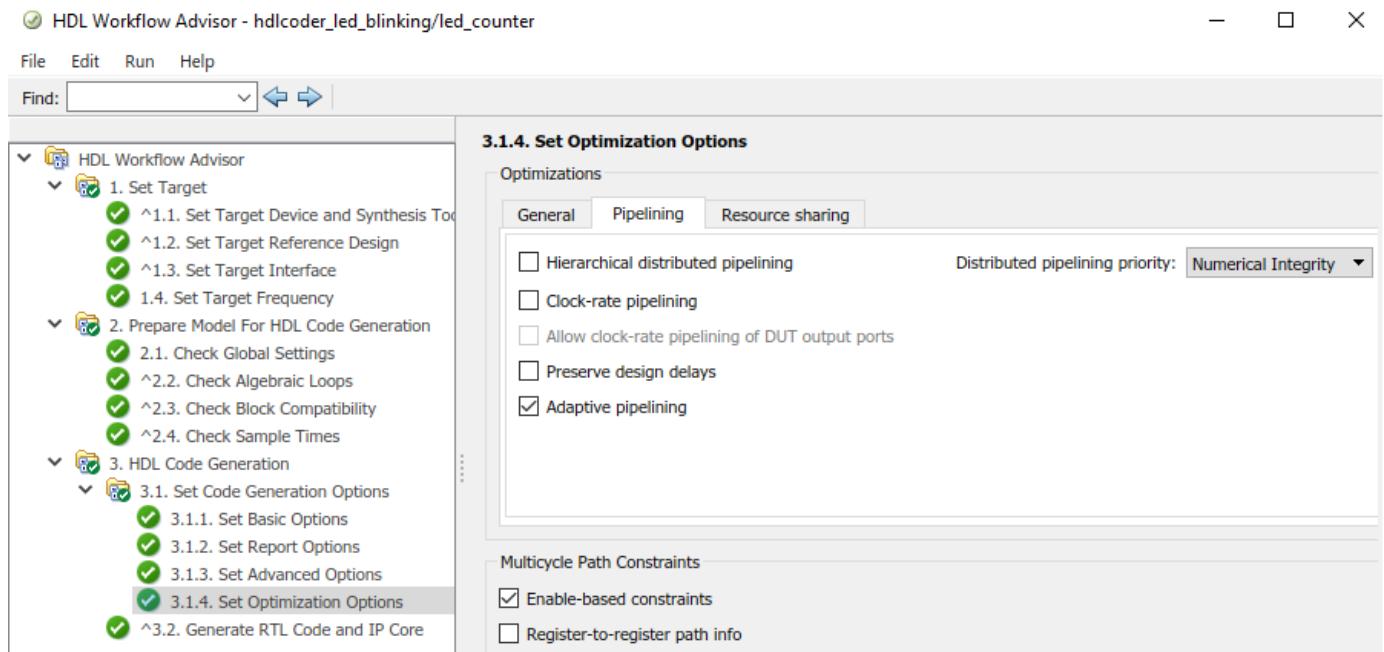


If distributed pipelining is unable to move the registers, you can add Delay blocks to your model, and then enable the **Preserve design delays** setting. Reset the **Check Global Settings** task and run the workflow to the **Build FPGA Bitstream** task. You can iterate on this approach and use other optimizations such as clock-rate pipelining with a large value for the **Oversampling factor** if the design does not meet the timing requirements. To specify these settings, use the **Pipelining** tab of the **Set Optimization Options** task in the HDL Workflow Advisor. For more information, see “Speed Optimization”.

Recommendation 2: Specify Enable-Based Multicycle Path Constraints

If your design contains multiple sample rates or uses certain HDL block implementations or speed optimizations that insert pipeline registers at a faster rate after code generation, your design can have multicycle paths. By default, HDL Coder uses a single clock mode that generates a master clock at the fastest sample rate and creates a timing controller entity. The timing controller generates a set of clock enables with the required rate and phase information to sample the clock signal for blocks that operate at a slower sample time.

If your critical path is on a slower signal rate, synthesis tools can fail to meet the timing requirements. A timing failure occurs because the tools cannot infer the sample rates from the generated HDL code and assume that these paths have to run at the fastest rate. You can use enable-based multicycle path constraints to generate a constraints file that enables the synthesis tool to ease the clock constraint on the multicycle paths. To specify generation of multicycle path constraints, in the **Set Optimization Options** task, select the **Enable-based constraints** check box. Run the workflow to the **Build FPGA Bitstream** task. For an example, see “Use Multicycle Path Constraints to Meet Timing for Slow Paths” on page 23-32.



Recommendation 3: Reduce the Target Frequency

Use the **Target Frequency (MHz)** setting to specify the target frequency for HDL Coder to modify the clock module setting in the reference design to produce the clock signal with that frequency. See also “[Target Frequency Parameter](#)” on page 14-8.

To resolve timing failures, reduce the **Target Frequency (MHz)** setting so that the synthesis tool can meet the timing constraint. To see what target frequency you can specify, use the slack and the critical path information from the synthesis tool timing report. Because slack is equal to the difference between the required time and arrival time, you can add the slack to the required time, and then use this sum as the **New required time**. Use the reciprocal of this **New required time** as the target frequency value to meet the timing requirements because the **New required time** equals the arrival time. To compute the target frequency, in the MATLAB Command Window, run this script.

```
% Specify the required time (ns) and slack (ns) using timing report
required_time = 2;
slack = 2.2;

% Slack is difference between required_time and arrival_time
% By adding slack to required_time you can resolve
New_required_time = required_time + slack;
Target_frequency = 1 / (New_required_time);

% Since we computed the new time in nanoseconds
Target_frequency_MHz = Target_frequency * 10^3;
```

In the **Set Target Frequency** task, specify this value for **Target Frequency (MHz)**, and then run the workflow to the **Build FPGA Bitstream** task. If you see a timing failure, you can use this approach to iterate on the target frequency value until your design meets the timing requirements and the slack becomes zero.

You can also export the HDL Workflow Advisor settings to a script and keep iterating on the target frequency value by specifying `Target_frequency_MHz` as the value for the `TargetFrequency` property. Then, run the script.

```
% Set this frequency as the new target frequency
hdlset_param('hdlcoder_led_blinking', 'TargetFrequency', Target_frequency_MHz);
```

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Custom IP Core Generation” on page 40-10
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 41-93
- “Program Target FPGA Boards or SoC Devices” on page 40-49

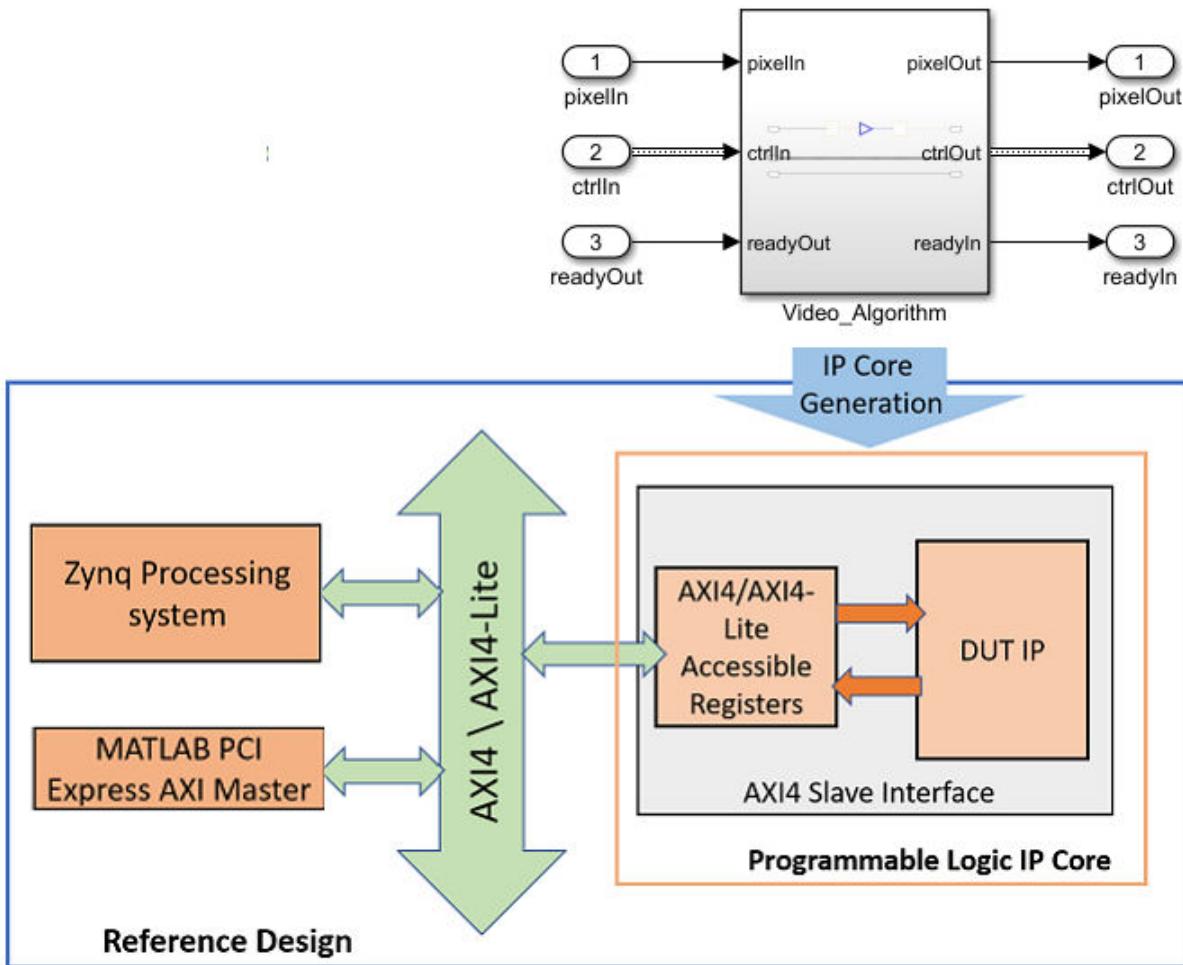
Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface

In this section...

["Vivado-Based Reference Designs" on page 40-45](#)

["Qsys-Based Reference Designs" on page 40-47](#)

You can define multiple AXI Master interfaces in your custom reference design and access the AXI4 slave interfaces in the generated HDL DUT IP core for the DUT. This capability enables you to simultaneously connect the HDL DUT IP core to two or more AXI Master IP in the reference design, such as the HDL Verifier JTAG AXI Master IP and the ARM processor in the Zynq processing system.



Vivado-Based Reference Designs

To define multiple AXI Master interfaces, you specify the `BaseAddressSpace` and `MasterAddressSpace` for each AXI Master instance, and also the `IDWidth` property.

`IDWidth` is the width of all ID signals, such as `AWID`, `WID`, `ARID`, and `RID`, specified as a positive integer. By default, the `IDWidth` is 12, which enables you to specify one AXI Master interface connection to the DUT IP core. To connect the DUT IP core to multiple AXI Master interfaces, you may have to increase the `IDWidth`. The `IDWidth` value is tool-specific. To see the value that you must use when specifying more than one AXI Master interface, refer to the documentation for that tool. If you use an incorrect ID width, the synthesis tool generates an error, and reports the correct `IDWidth` that you must use.

This code is the syntax for the `MasterAddressSpace` field when specifying multiple AXI Master interfaces in Vivado-based reference designs:

```
'MasterAddressSpace', ...
    {'AXI Master Instance Name1/Address Space of Instance Name1', ...
     'AXI Master Instance Name2/Address_Space of Instance Name2', ...};
```

For example, this code illustrates how you can modify the `plugin_rd` file to define two AXI Master interfaces.

```
% ...
%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');

% ...
% ...

% The DUT IP core in this reference design is connected
% to both Zynq Processing System and the MATLAB as AXI
% Master IP. Because of 2 AXI Master, ID width
% has to be increased from 12 to 13.
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress',        {'0x40010000', '0x40010000'}, ...
    'MasterAddressSpace', {'processing_system7_0/Data', 'hdlverifier_axi_master_0/axi4m'}, ...
    'IDWidth',            13);

% ...
```

In this example, the two AXI Master IP are the HDL Verifier MATLAB as AXI Master IP and the ARM processor. Based on the syntax of the `MasterAddressSpace`, for the HDL Verifier MATLAB as AXI Master IP, the `AXI Master Instance Name` is `hdlverifier_axi_master_0` and the `Address_Space of Instance Name` is `axi4m`.

The AXI4 slave interfaces in the HDL DUT IP core connect to the Xilinx AXI Interconnect IP that is defined by the `InterfaceConnection` property of the `addAXI4SlaveInterface` method. The AXI4 slave interfaces have a `BaseAddress`. This `BaseAddress` must map to the `MasterAddressSpace` for the two AXI Master IP, which is specified as a cell array of character vectors.

You must make sure that the AXI Master IPs have already been included in the Vivado reference design project. `system_top.tcl` is the TCL file that is defined by the `CustomBlockDesignTcl` property of the `addCustomVivadoDesign` method. In this TCL file, you must make sure that the two AXI Master IP are connected to the same Xilinx AXI Interconnect IP. The interconnects then connect the AXI Master IPs to the AXI4 slave interfaces in the HDL IP core.

After you run the `IP Core Generation` workflow and create the Vivado project, open the project. In the Vivado project, if you open the block design, you see the two AXI Master IP connected to the

HDL DUT IP core. If you select the **Address Editor** tab, you see the AXI Master instance names and the corresponding address spaces.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
hdverifier_axi_master_0					
axi4m (32 address bits : 4G)					
led_count_ip_0	AXI4_Lite	reg0	0x4001_0000	64K	0x4001_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
led_count_ip_0	AXI4_Lite	reg0	0x4001_0000	64K	0x4001_FFFF

Qsys-Based Reference Designs

To define multiple AXI Master interfaces, you specify the `InterfaceConnection` and `BaseAddressSpace` for each AXI Master instance, and also the `IDWidth` property. This code is the syntax for the `InterfaceConnection` field when specifying multiple AXI Master interfaces in Qsys-based reference designs:

```
'InterfaceConnection', ...
  {'AXI Master Instance Name1/Port name of Instance Name1', ...,
   'AXI Master Instance Name2/Port name of Instance Name1', ...};
```

For example, this code illustrates how you can modify the `plugin_rd` file to define three AXI Master interfaces.

```
% ...
%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign('CustomQsysPrjFile', 'system_soc.qsys');
hRD.CustomConstraints = {'system_soc.sdc', 'system_setup.tcl'};

% ...
% add AXI4 slave interfaces
hRD.addAXI4SlaveInterface( ...
  'InterfaceConnection', {'hps_0.h2f_axi_master', 'master_0.master', 'MATLAB_as_AXI_Master_0.axm_m0'}, ...
  'BaseAddress', {'0x0000_0000', '0x0000_0000', '0x0000_0000'}, ...
  'InterfaceType', 'AXI4',...
  'IDWidth', 14);

%
```

Based on the syntax of the `InterfaceConnection` option, for the HDL Verifier MATLAB as AXI Master IP, the AXI Master Instance Name is `MATLAB_as_AXI_Master_0` and the Port name is `axm_m0`. For each AXI Master IP, the `BaseAddress` of the HDL IP core and `InterfaceConnection` must be specified as a cell array of character vectors.

You must make sure that the AXI Master IPs have already been included in the Qsys reference design project. `system_soc.qsys` is the file that is defined by the `CustomQsysPrjFile` property of the

`addCustomQsysDesign` method. In this file, you must make sure that the two AXI Master IP are connected to the same Qsys AXI Interconnect IP.

The interconnects then connect the AXI Master IPs to the AXI4 slave interfaces in the HDL IP core.

After you run the `IP Core Generation` workflow and create the Quartus project, open the project. In the Quartus project, you see the three AXI Master IP and the AXI Master interfaces connected to the HDL IP core for the DUT. If you select the **Address Map** tab, you see the AXI Master instance names, the port names, and the corresponding address spaces.

System: system_soc Path: hps_0				
	MATLAB_as_AXI_Master_0.axm_m0	hps_0.h2f_axi_master	hps_0.h2f_lw_axi_master	master_0.master
hps_0.f2h_axi_slave				
led_count_ip_0.s_axi	0x0000_0000 - 0x0000_ffff	0x0000_0000 - 0x0000_ffff		0x0000_0000 - 0x0000_ffff

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Board and Reference Design Registration System” on page 41-39
- “Register a Custom Board” on page 41-42
- “Register a Custom Reference Design” on page 41-45

Program Target FPGA Boards or SoC Devices

In this section...

- “How to Program Target Device” on page 40-49
- “Programming Methods” on page 40-50

To configure or program the connected target SoC device or FPGA board, use the IP Core Generation workflow in the HDL Workflow Advisor.

How to Program Target Device

Using the Workflow Advisor UI

- 1 Open the HDL Workflow Advisor. Right-click the DUT Subsystem that contains the algorithm to be deployed on the target FPGA, and select **HDL Code > HDL Workflow Advisor**.
- 2 In the **Set Target Device and Synthesis Tool** task, specify IP Core Generation as the **Target workflow**, and specify a **Target platform** other than the Generic Xilinx Platform or Generic Altera Platform.
- 3 Right-click the **Program Target Device** task and select **Run to Selected Task**.
- 4 In the **Program Target Device** task, specify the **Programming method**, and run this task.

To learn more about the HDL Workflow Advisor, see “Getting Started with the HDL Workflow Advisor” on page 31-6.

Using the Workflow Advisor Script

To program the target device at the command line, after you specify the target workflow and target platform in the **Set Target Device and Synthesis Tool** task, export the HDL Workflow Advisor settings to a script. In the HDL Workflow Advisor window, select **File > Export to Script**. The script creates and configures an `hdlcoder.WorkflowConfig` object that is denoted by `hWC`.

Before you run the script, you can specify how to program the target hardware by using the `ProgrammingMethod` property of the `WorkflowConfig` object. This code snippet shows an example script that is exported from the HDL Workflow Advisor when you use the default Download **Programming method**. To run the **Program Target Device** task, customize this script by setting the `RunTaskProgramTargetDevice` attribute of the `WorkflowConfig` object to `true`.

```
% This script was generated using the following parameter values:
%
% Set properties related to 'RunTaskProgramTargetDevice' Task
hWC.RunTaskProgramTargetDevice = true;
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download;

% Validate the Workflow Configuration Object
hWC.validate;

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

After you run the **Build FPGA Bitstream** task, the Workflow Advisor provides you a link to generate a Workflow script that programs the target device without rerunning the previous tasks in the Workflow Advisor.

Result: Passed

Passed Build Embedded System.

Synthesis Tool Log:

```
Task "Build FPGA Bitstream" successful.  
Generated logfile: hdl\_prj\hdlsrc\hdlcoder\_led\_blinking\workflow\_task\_BuildFPGABitstream.log  
  
Running embedded system build outside MATLAB.  
Please check external shell for system build progress.  
  
The generated bitstream file is located at: hdl_prj\vivado_ip_prj\vivado_prj.runs\impl_1\system_top_wrapper.bit  
Generate an HDL Workflow Command-Line Interface script to program the target device: hdlworkflow\_ProgramTargetDevice.m.
```

Click the link to open the script in the MATLAB Editor. This code snippet shows an example script that is generated by the HDL Workflow Advisor. If close the HDL Workflow Advisor, you can run the script to program the target hardware without running other workflow tasks.

```
% Load the Model  
  
% ...  
  
% Set Workflow tasks to run  
hWC.RunTaskGenerateRTLCodeAndIPCore = false;  
hWC.RunTaskCreateProject = false;  
hWC.RunTaskGenerateSoftwareInterface = false;  
hWC.RunTaskBuildFPGABitstream = false;  
hWC.RunTaskProgramTargetDevice = true;  
  
% Set properties related to 'RunTaskProgramTargetDevice' Task  
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download;  
  
% Validate the Workflow Configuration Object  
hWC.validate;
```

To learn more about the script-based workflow, see “Run HDL Workflow with a Script” on page 31-53.

Programming Methods

Download

If you use the reference designs for Xilinx Zynq and Intel SoC hardware platforms, the code generator uses **Download** as the default **Programming method**.

When you use **Download** as the **Programming method** and run the **Program Target Device** task, HDL Coder copies the generated FPGA bitstream, Linux devicetree, and system initialization scripts to the SD card on the target board, and then keeps the bitstream on the SD card persistently. To use this programming method, you do not require an Embedded Coder license. You can create an SSH object by specifying the **IP Address**, **SSH Username**, and **SSH Password**. HDL Coder uses the SSH object to copy the bitstream to the SD card and reprogram the board.

It is recommended that you use the **Download** method, because this method programs the FPGA bitstream and loads the corresponding Linux devicetree before booting the Linux system. Therefore, the **Download** method is more robust and does not result in a kernel panic or kernel hang on boot up. During the Linux reboot, the FPGA bitstream is reprogrammed from the SD card automatically. To specify **Download** as the **Programming method** when you run the workflow using the Workflow Advisor script, before you run the script, make sure that the **ProgrammingMethod** property of the **WorkflowConfig** object, **hWC**, is set to **Download**.

```
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.Download;
```

JTAG

When you specify **JTAG** as the **Programming method** and run the **Program Target Device** task, HDL Coder uses a JTAG cable to program the target SoC device. Use this method to program Intel and Xilinx SoC devices and standalone FPGA boards.

For programming SoC devices, it is recommended that you use the **Download** method. The **JTAG** mode does not involve the ARM processor and programs the onboard FPGA directly. This mode does not update the Linux devicetree, and can crash the Linux system or result in a kernel panic when the bitstream does not match the devicetree. To avoid this situation, use the **Download** method to program the SoC device.

The standalone Intel and Xilinx FPGA boards do not have an embedded ARM processor. **JTAG** is the default method that you use to program the FPGA boards.

To specify **JTAG** as the **Programming method** when you run the workflow using the Workflow Advisor script, before you run the script, change the **ProgrammingMethod** property of the **WorkflowConfig** object, **hWC**, to **JTAG**.

```
hWC.ProgrammingMethod = hdlcoder.ProgrammingMethod.JTAG;
```

Custom

To program the target device, you can specify a custom programming method when you create your own custom reference design. Use the **CallbackCustomProgrammingMethod** of the **hdlcoder.ReferenceDesign** class to register a function handle for the callback function that gets executed when running the **Program Target Device** task. To define your callback function, create a file that defines a MATLAB function and add the file to your MATLAB path.

This example code snippet shows a reference design definition file that uses a custom programming method. To learn more, see **CallbackCustomProgrammingMethod**.

```
unction hRD = plugin_rd()
% Reference design definition

% ...

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Parameter Callback Custom';
hRD.BoardName = 'ZedBoard';

% Tool information
hRD.SupportedToolVersion = {'2017.2'};
```

```
% ...  
hRD.CallbackCustomProgrammingMethod =  
@my_reference_design.callback_CustomProgrammingMethod;
```

See Also

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Program Target Device” on page 37-23
- “IP Core Generation Workflow for Standalone FPGA Devices” on page 41-89
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2

Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core

When you run the hardware-software co-design workflow for SoC platforms, you generate an HDL IP core for the DUT algorithm, and then integrate the IP core into the reference design. See “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2.

To test the HDL IP core on the target hardware, you can generate a software interface script and a software interface model. The software interface model uses AXI driver blocks to test the HDL IP core functionality in external mode simulation. For rapid prototyping, use the generated software script instead. The script contains the DUT ports and interface mapping information which HDL Coder uses to create the AXI drivers and access the HDL IP core.

Prerequisites

- Use a target platform that you want to deploy the software interface model to, such as ZedBoard™
- Install the latest version of the third-party synthesis tool, such as Xilinx Vivado as mentioned in “HDL Language Support and Supported Third-Party Tools and Hardware”. In your MATLAB session, set the path to that installed synthesis tool by using the `hdlsetuptoolpath` function.
- If you are generating the software interface model, you must install Embedded Coder and Simulink Coder.
- Install the HDL Coder and Embedded Coder support packages for the target platform. On the MATLAB Toolstrip, click **Home** > **Add-Ons** > **Get Add-Ons** button. See “Get and Manage Add-Ons”.

Generate Software Interface

When running the IP Core Generation workflow, you can generate a software interface script and model from the HDL Workflow Advisor UI and at the command line.

From the UI, in the **Embedded System Integration > Generate Software Interface** task, select the **Generate Software interface model** and **Generate Software interface script** check boxes.

If you are targeting standalone FPGA boards, you cannot generate a software interface model. Instead, you can generate a software interface script and test the IP core by using the MATLAB AXI Master driver.

- 1 In the **Set Target Reference Design** task, set **Insert JTAG MATLAB as AXI Master** to on. Run the workflow to the **Generate Software Interface** task.
- 2 In the **Generate Software Interface** task, select the **Generate Software interface script** check box and run this task.

At the command line, export the HDL Workflow Advisor settings to a script, and then use these properties with the Workflow Configuration object. This script specifies running the software interface task by generating both model and script. If you skip the task by setting `RunTaskGenerateSoftwareInterface` to `false`, then the model and script are not generated. See “Configure and Run IP Core Generation Workflow with a Script”..

```
% Export Workflow Configuration Script
```

```
% ...

%% Load the Model
load_system('hdlcoder_led_blinking');

%% Model HDL Parameters
% Set Model HDL parameters

% ...

hdlset_param('hdlcoder_led_blinking', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_led_blinking', 'Workflow', 'IP Core Generation');

% ...

% Set Workflow tasks to run
hWC.RunTaskGenerateSoftwareInterface = true;
hWC.GenerateSoftwareInterfaceModel = true;
hWC.GenerateSoftwareInterfaceScript = true;

% ...

%% Run the workflow
hdlcoder.runWorkflow('hdlcoder_led_blinking/led_counter', hWC);
```

Software Interface Model

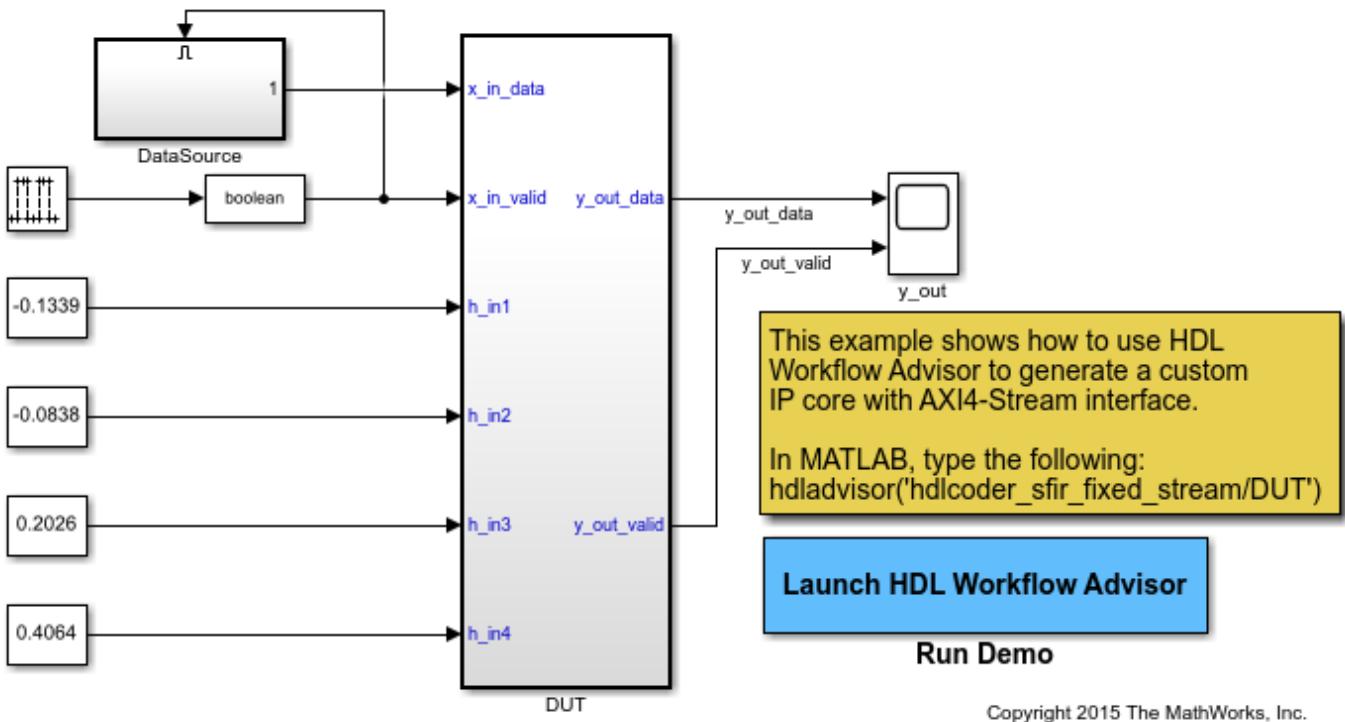
When you run the workflow for SoC platforms, generate a software interface model to test the HDL IP core functionality. If you have Embedded Coder and Simulink Coder installed, you can generate embedded code from the model, and build and run the executable on the Linux kernel on the ARM processor. When you target standalone FPGA boards, you cannot generate a software interface model because the boards do not have an embedded ARM processor. Instead, generate a software interface script to test the IP core by using the MATLAB AXI Master.

The generated software interface model replaces the DUT algorithm in your original model with AXI driver blocks based on the Target platform interface table and reference design settings. To test the HDL IP core functionality, simulate the model in external mode to run on the target hardware. The model has concurrent execution enabled by default, which means that multiple tasks are executed concurrently by the processor on board the SoC platform.

The software interface model has the same name as your original model with the prefix `gm_` and the postfix `_interface`. The generated model from HDL code generation has the prefix `gm_`. To indicate that you are using this model as the software interface model, you can change the prefix to `sm_`.

For example, open the model `hdlcoder_sfir_fixed_stream`. This model maps the `sfir_fixed` model to the simplified AXI4-Stream protocol by inserting the `Valid` signal as an input control port.

```
open_system('hdlcoder_sfir_fixed_stream')
```



Copyright 2015 The MathWorks, Inc.

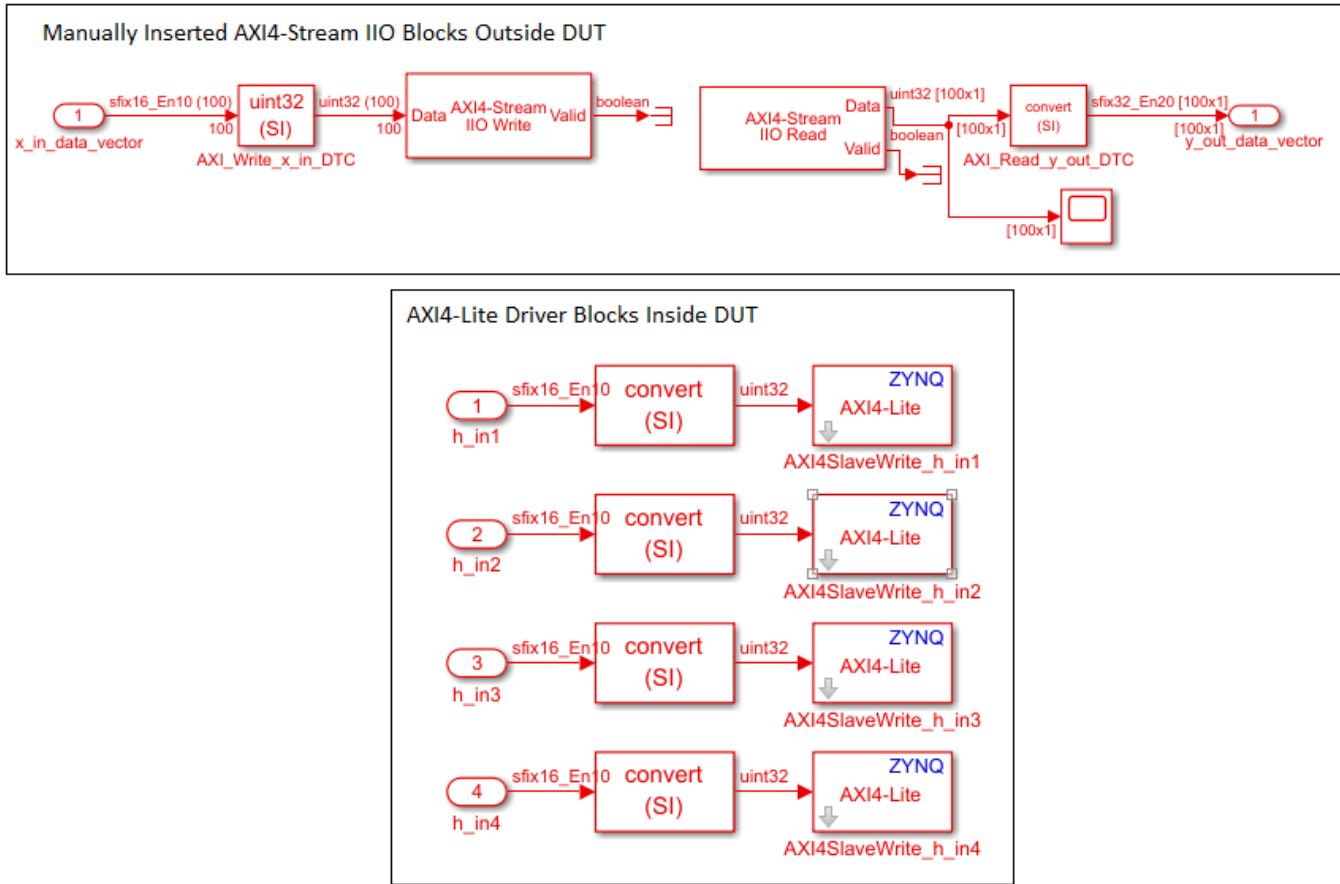
- 1 Open the HDL Workflow Advisor for the DUT subsystem. In the **Set Target Device and Synthesis Tool** task, specify IP Core Generation as **Target workflow** and ZedBoard as **Target platform**. Click **Run this task**.
- 2 In the **Set Target Reference Design** task, specify Default system with AXI4-Stream interface as the **Target Reference Design**. Click **Run this task**.
- 3 In the **Set Target Interface** task, map the DUT ports to target interfaces in the Target platform interface table. Click **Run this task**.

Target platform interface table						
Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options	
x_in_data	Import	sfix16_En...	AXI4-Stream Slave	Data		
x_in_valid	Import	boolean	AXI4-Stream Slave	Valid		
h_in1	Import	sfix16_En...	AXI4-Lite	x"100"	Options...	
h_in2	Import	sfix16_En...	AXI4-Lite	x"104"	Options...	
h_in3	Import	sfix16_En...	AXI4-Lite	x"108"	Options...	
h_in4	Import	sfix16_En...	AXI4-Lite	x"10C"	Options...	
y_out_data	Outport	sfix32_En...	AXI4-Stream Master	Data		
y_out_valid	Outport	boolean	AXI4-Stream Master	Valid		

- 4 Right-click the **Generate Software Interface** task and select **Run to selected task**. Run the workflow to generate the software interface model.

The generated software interface model has the name `gm_hdlcoder_sfir_fixed_stream_interface`. As your original model uses scalar ports for the Data ports `x_in_data` and `y_out_data`, the **Generate software interface** task displays a warning that AXI4-Stream IIO driver blocks are not automatically generated in the software interface model. You can either insert the driver blocks from the Embedded Coder Support Package Library for the target platform in the Simulink Library Browser or rerun the workflow by mapping the Data ports to vector signals. To see a software interface model that has the AXI4-Stream IIO driver blocks inserted in it, open `hdlcoder_sfir_fixed_stream_sw`. The model uses AXI4-Stream IIO Write and AXI4-Stream IIO Read blocks for the Data ports. The filter ports are mapped to AXI4-Lite driver blocks.

```
open_system('hdlcoder_sfir_fixed_stream_sw')
```



Configure the model with a stop time of `inf`. On the **Hardware** tab, you see the hardware settings specified on the model. You can then connect, and build and run the application on the target platform to verify the HDL IP core functionality. For an example, see “Getting Started with AXI4-Stream Interface in Zynq Workflow” on page 41-137.

Software Interface Script

For rapid prototyping and testing the HDL IP core functionality, use the software interface script. The script does not require an Embedded Coder license. The script is a MATLAB file that is generated based on the reference design and Target platform interface table settings. It contains commands that enable you to connect to the target hardware, and to write to or read from the generated IP core from MATLAB. For standalone FPGA boards, use the generated software interface script to verify the HDL IP core functionality by using the MATLAB AXI Master.

The software interface script has the same name as your original model with the prefix `gs_` and the postfix `_interface`. The script instantiates a setup function that is generated when you enable generation of the software interface script. For example, this code shows the setup function generated for the model `hdlcoder_sfir_fixed_stream` with the reference design and Target platform interface table settings specified above. The setup function contains commands for the AXI4 slave and AXI4-Stream interfaces that HDL Coder uses to control the DUT ports in the generated HDL IP core that are mapped to the corresponding interfaces.

```
function gs_hdlcoder_sfir_fixed_stream_setup(hFPGA)
%
%-----%
% Software Interface Script Setup
%
% Generated with MATLAB 9.10 (R2021a) at 09:13:05 on 10/07/2020.
% This function was created for the IP Core generated from design 'hdlcoder_sfir_fixed_stream'.
%
% Run this function on an "fpga" object to configure it with
% the same interfaces as the generated IP core.
%-----%

%% AXI4-Lite
addAXI4SlaveInterface(hFPGA, ...
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xA0000000, ...
    "AddressRange", 0x10000);

hPort_h_in1 = hdlcoder.DUTPort("h_in1", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100");

hPort_h_in2 = hdlcoder.DUTPort("h_in2", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x104");

hPort_h_in3 = hdlcoder.DUTPort("h_in3", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x108");

hPort_h_in4 = hdlcoder.DUTPort("h_in4", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x10C");

mapPort(hFPGA, [hPort_h_in1, hPort_h_in2, hPort_h_in3, hPort_h_in4]);

%% AXI4-Stream
addAXI4StreamInterface(hFPGA, ...
    "InterfaceID", "AXI4-Stream", ...
    "WriteEnable", true, ...
    "WriteFrameLength", 1024, ...
    "ReadEnable", true, ...
    "ReadFrameLength", 1024);

hPort_x_in_data = hdlcoder.DUTPort("x_in_data", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

hPort_y_out_data = hdlcoder.DUTPort("y_out_data", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,32,20), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

mapPort(hFPGA, [hPort_x_in_data, hPort_y_out_data]);

end
```

The software interface script instantiates this setup function to connect to the target and send read or write commands. You can uncomment and send meaningful data by using the inputs to the DUT in your original model. After interfacing with the hardware, the script disconnects from the hardware resource associated with the `fpga` object.

```
%-----
% Software Interface Script
%
% Generated with MATLAB 9.10 (R2020b) at 09:13:10 on 10/07/2020.
% This script was created for the IP Core generated from design 'hdlcoder_sfir_fixed_stream'.
%
% Use this script to access DUT ports in the design mapped to compatible IP core interfaces.
% You can write to input ports in the design and read from output ports directly from MATLAB.
%
% To write to input ports, use the "writePort" command and specify port name and input data.
% The input data will be cast to the DUT port's data type before writing.
%
% To read from output ports, use the "readPort" command and specify the port name.
% The output data will be returned with the same data type as the DUT port.
%
% Use the "release" command to release MATLAB's control of the hardware resources.
%-----

%% Create fpga object
hFPGA = fpga("Xilinx");

%% Setup fpga object
% This function configures "fpga" object with same interfaces as the generated IP core
gs_hdlcoder_sfir_fixed_stream_setup(hFPGA);

%% Write/read DUT ports
% Uncomment the following lines to write/read DUT ports in the generated IP Core.
% Update the example data in the write commands with meaningful data to write to the DUT.
%% AXI4-Lite
writePort(hFPGA, "h_in1", zeros([1 1]));
writePort(hFPGA, "h_in2", zeros([1 1]));
writePort(hFPGA, "h_in3", zeros([1 1]));
writePort(hFPGA, "h_in4", zeros([1 1]));

%% AXI4-Stream
writePort(hFPGA, "x_in_data", zeros([1 1024]));
data_y_out_data = readPort(hFPGA, "y_out_data");

%% Release hardware resources
release(hFPGA)
```

See Also

Objects

`fpga | hdlcoder.WorkflowConfig`

Functions

`hdlcoder.runWorkflow`

More About

- “Model Design for AXI4-Stream Interface Generation” on page 41-10
- “Model Design for AXI4 Slave Interface Generation” on page 41-3
- “Create Software Interface Script to Control and Rapidly Prototype HDL IP Core” on page 40-60

Create Software Interface Script to Control and Rapidly Prototype HDL IP Core

When you run the `IP Core Generation` workflow for your Simulink model, you can generate a software interface script to rapidly prototype the HDL IP core. The script contains commands that enable you to connect to the target hardware from MATLAB, and to write to or read from the generated IP core.

Software Interface Script

When you run the `IP Core Generation` workflow to the **Generate Software Interface** task and select the **Generate MATLAB software interface script** check box, two MATLAB files are generated:

- `gs_modelName_setup.m` is a setup script that adds the AXI4 slave and AXI4-Stream interfaces. The script also contains DUT port objects that have the port name, direction, data type, and interface mapping information. It then maps the DUT ports to the corresponding interfaces.
- `gs_modelName_interface.m` creates a target object, instantiates the setup script `gs_modelName_setup.m`, and then connects to the target hardware. It then sends read and write commands to the generated HDL IP core.

See “Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core” on page 40-53.

Customizing Software Interface Script

For rapid prototyping, customize the software interface script or create your own script based on how you modify your original design. Customize the script to specify:

- A target object for a different FPGA vendor.
- Additional interfaces or configure existing interfaces based on modifications to your original design. HDL Coder uses this information to automatically create the IIO drivers to access the HDL IP core.
- Additional DUT port objects or remove existing objects based on how you modify your design, and then change the mapping information accordingly.
- Input data to write to the DUT ports and output data to read from the ports.

Develop Software Interface Script

You can customize the generated software script or create your own software interface script from scratch. This documentation page describes how to develop your own script.

Step 1: Create fpga Object for Target FPGA Vendor

Create an `fpga` object for the target device.

```
hFPGA = fpga("Xilinx")
```

```
hFPGA =
```

```
fpga with properties:
```

```

Vendor: "Xilinx"
Interfaces: [0x0 fpgaio.interface.InterfaceBase]

```

To use an Intel target:

```

hFPGA = fpga("Intel")
hFPGA =
fpga with properties:
Vendor: "Intel"
Interfaces: [0x0 fpgaio.interface.InterfaceBase]

```

Step 2: Configure AXI Interfaces

Configure the AXI interfaces for mapping to the DUT ports in the generated HDL IP core. You can add AXI4 slave and AXI4-Stream interfaces. To add AXI4 slave interfaces, use the `addAXI4SlaveInterface` function.

```

addAXI4SlaveInterface(hFPGA, ...
... % Interface properties
"InterfaceID", "AXI4-Lite", ...
"BaseAddress", 0xA0000000, ...
"AddressRange", 0x10000, ...
... % Driver properties
"WriteDeviceName", "mwipcore0:mmwr0", ...
"ReadDeviceName", "mwipcore0:mmrd0");

```

To add AXI4-Stream interfaces, use the `addAXI4StreamInterface` function.

```

addAXI4StreamInterface(hFPGA, ...
... % Interface properties
"InterfaceID", "AXI4-Stream", ...
"WriteEnable", true, ...
"ReadEnable", true, ...
"WriteFrameLength", 1024, ...
"ReadFrameLength", 1024, ...
... % Driver properties
"WriteDeviceName", "mwipcore0:mm2s0", ...
"ReadDeviceName", "mwipcore0:s2mm0");

```

The interface mapping information that you specified is saved as a property on the `fpga` object, `hFPGA`.

```

hFPGA
hFPGA =
fpga with properties:
Vendor: "Xilinx"
Interfaces: [1x2 fpgaio.interface.InterfaceBase]

```

For standalone FPGA boards that do not have an embedded ARM processor, you can create an object and then use the `aximaster` object and then use this object as the driver for the `addAXI4SlaveInterface` function. The `aximaster` object requires the HDL Verifier support package for the Intel or Xilinx FPGA board.

```
% Create an "aximaster" object
hAXIMDriver = aximaster("Xilinx");

% Pass it into the addInterface command
addAXI4SlaveInterface(hFPGA, ...
    ... % Interface properties
    "InterfaceID", "AXI4-Lite", ...
    "BaseAddress", 0xB0000000, ...
    "AddressRange", 0x10000, ...
    ... % Driver properties
    "WriteDriver", hAXIMDriver, ...
    "ReadDriver", hAXIMDriver, ...
    "DriverAddressMode", "Full");
```

Step 3: Configure Port Mapping Information

You can specify information about the DUT ports in the generated HDL IP core as a port object array by using the `hdlcoder.DUTPort` object. The object represents the ports of your DUT on the target hardware.

```
hPort_h_in1 = hdlcoder.DUTPort("h_in1", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Lite", ...
    "IOInterfaceMapping", "0x100")
```

```
hPort_h_in1 =
DUTPort with properties:
    Name: "h_in1"
    Direction: IN
    DataType: [1×1 embedded.numerictype]
    Dimension: [1 1]
    IOInterface: "AXI4-Lite"
    IOInterfaceMapping: "0x100"
```

To write to or read from the DUT ports in the generated HDL IP core, map the ports to the AXI interface by using the `mapPort` function. After you map the ports to the interfaces, this information is saved on the `fpga` object as the `Interfaces` property.

```
mapPort(hFPGA, hPort_h_in1);
hFPGA.Interfaces
```

```
ans =
AXI4Slave with properties:
```

```

InterfaceID: "AXI4-Lite"
BaseAddress: "0xA0000000"
AddressRange: "0x10000"
WriteDriver: [1x1 fpgaio.driver.AXIMemoryMappedIOWrite]
ReadDriver: [1x1 fpgaio.driver.AXIMemoryMappedIOWRead]
InputPorts: "h_in1"
OutputPorts: [0x0 string]

```

You can also specify this information for ports mapped to AXI4-Stream interfaces.

```

hPort_x_in_data = hdlcoder.DUTPort("x_in_data", ...
    "Direction", "IN", ...
    "DataType", numerictype(1,16,10), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

hPort_y_out_data = hdlcoder.DUTPort("y_out_data", ...
    "Direction", "OUT", ...
    "DataType", numerictype(1,32,20), ...
    "Dimension", [1 1], ...
    "IOInterface", "AXI4-Stream");

```

To write to or read from the DUT ports in the generated HDL IP core, map the ports to the AXI interface by using the `mapPort` function.

```
mapPort(hFPGA, [hPort_x_in_data, hPort_y_out_data]);
```

After you map the ports to the interfaces, this information is saved on the `fpga` object as the `Interfaces` property.

```

hFPGA
hFPGA =
    fpga with properties:
        Vendor: "Xilinx"
        Interfaces: [1x2 fpgaio.interface.InterfaceBase]
hFPGA.Interfaces

ans =
    AXI4Slave with properties:
        InterfaceID: "AXI4-Lite"
        BaseAddress: "0xA0000000"
        AddressRange: "0x10000"
        WriteDriver: [1x1 fpgaio.driver.AXIMemoryMappedIOWrite]
        ReadDriver: [1x1 fpgaio.driver.AXIMemoryMappedIOWRead]
        InputPorts: "h_in1"
        OutputPorts: [0x0 string]

    AXI4Stream with properties:

```

```
InterfaceID: "AXI4-Stream"
WriteEnable: 1
ReadEnable: 1
WriteFrameLength: 1024
ReadFrameLength: 1024
WriteDriver: [1x1 fpgaio.driver.AXISstreamIIOWrite]
ReadDriver: [1x1 fpgaio.driver.AXISstreamIIORead]
InputPorts: "x_in_data"
OutputPorts: "y_out_data"
```

Step 4: Write Data and Read Output

To test the HDL IP core functionality, use the `readPort` and `writePort` functions to write data to or read data from these ports.

```
writePort(hFPGA, "h_in1", 5);
writePort(hFPGA, "x_in", sin(linspace(0, 2*pi, 1024)));
data = readPort(hFPGA, "y_out");
```

Step 5: Release Hardware Resources

After you have tested the HDL IP core, you can release the hardware resource associated with the `fpga` object by using the `release` function.

```
release(hFPGA)
```

See Also

Objects

`fpga` | `hdlcoder.DUTPort`

More About

- “Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core” on page 40-53
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2

Getting Started with Targeting Xilinx Zynq Platform

This example shows how to use the hardware-software co-design workflow to blink LEDs at various frequencies on the Xilinx® Zynq® ZC702 evaluation kit.

Introduction

This example is a step-by-step guide that helps you use HDL Coder™ to generate a custom HDL IP core which blinks LEDs on the Xilinx Zynq ZC702 evaluation kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Zynq-7000 AP SoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM Cortex-A9.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic using HDL Coder, generate C code for the ARM using Embedded Coder, and implement the design on the Xilinx Zynq Platform.

In this workflow, you perform the following steps:

- 1 Set up your Zynq hardware and tools.
- 2 Partition your design for hardware and software implementation.
- 3 Generate an HDL IP core using HDL Workflow Advisor.
- 4 Integrate the IP core into a Xilinx Vivado project and program the Zynq hardware.
- 5 Generate a software interface model.
- 6 Generate C code from the software interface model and run it on the ARM Cortex-A9 processor.
- 7 Tune parameters and capture results from the Zynq hardware using External Mode.

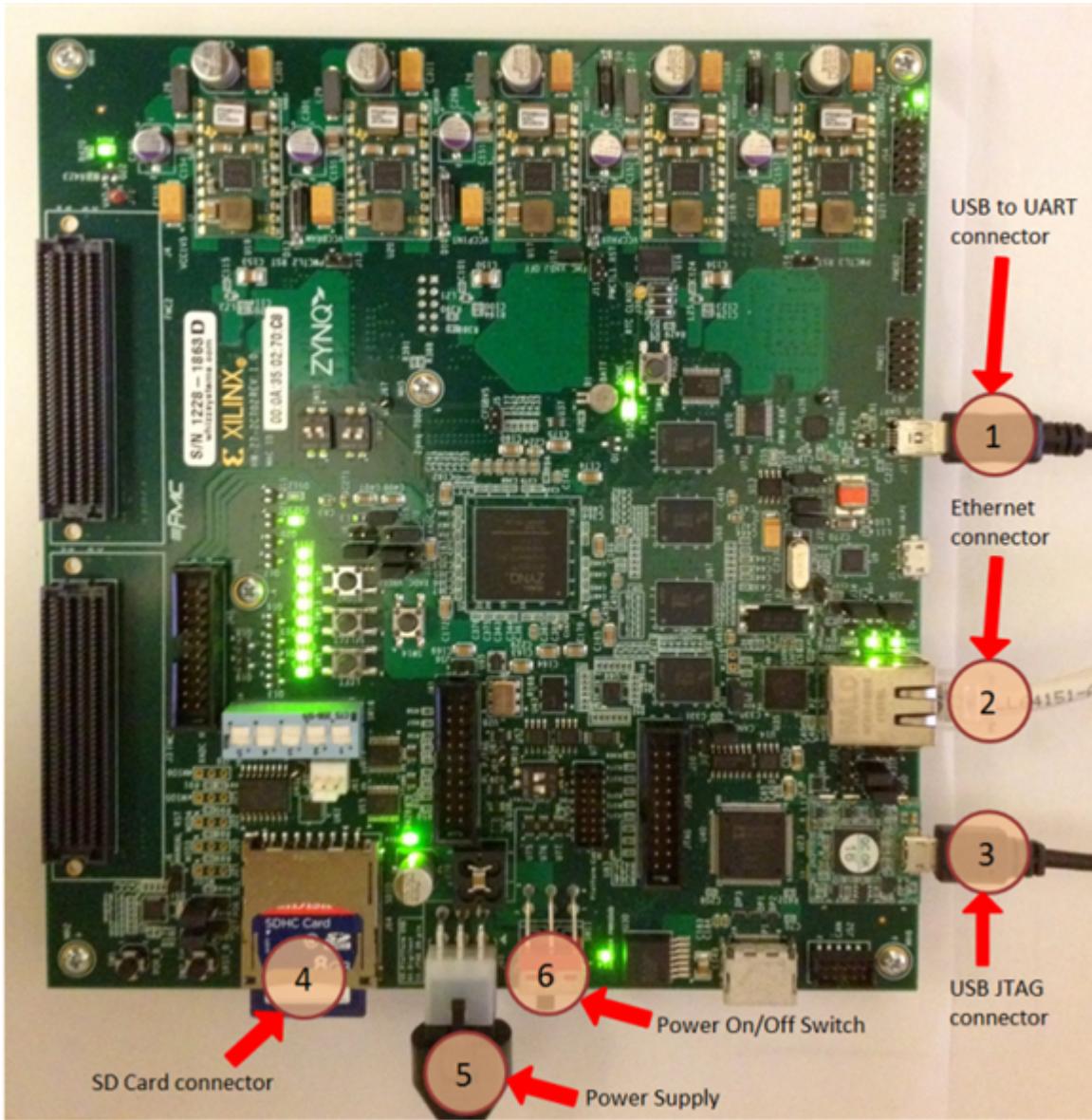
For more information, refer to other more advanced examples, and the HDL Coder and Embedded Coder documentation.

Requirements

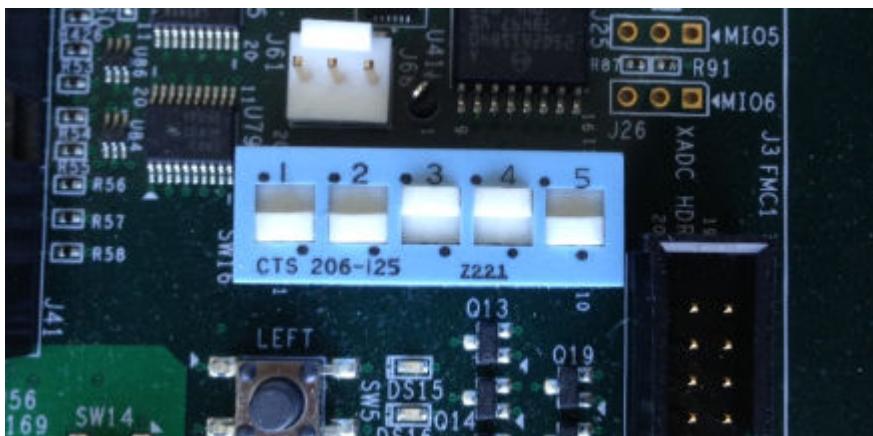
- 1 Xilinx Vivado Design Suite, with supported version listed in the HDL Coder documentation
- 2 Xilinx Zynq-7000 SoC ZC702 Evaluation Kit
- 3 HDL Coder Support Package for Xilinx Zynq Platform
- 4 Embedded Coder Support Package for Xilinx Zynq Platform

Set up Zynq hardware and tools

1. Set up the Xilinx Zynq ZC702 evaluation kit as shown in the figure below. To learn more about the ZC702 hardware setup, please refer to Xilinx documentation.



1.1. Make sure the SW16 switch is set as shown in the figure below, so you can boot Linux from the SD card.



1.2. Make sure the SW10 switch (JTAG chain input select two-position DIP switch) is set as shown in the figure below, so you can use the Digilent USB-to-JTAG interface (U23). Position 1: Off; Position 2: On.



1.3 Connect your computer to the USB UART connector using a Micro-USB cable. Make sure your USB device drivers, such as for the Silicon Labs CP210x USB to UART Bridge, are installed correctly. If not, search for the drivers online and install them.

1.4 Connect your computer and the Zynq board using an Ethernet cable.

2. Install the HDL Coder and Embedded Coder Support Packages for Xilinx Zynq Platform if you haven't already. To start the installer, go to the MATLAB toolbar and click **Add-Ons > Get Hardware Support Packages**. For more information, please refer to the Support Package Installation documentation.

3. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Xilinx Zynq Platform. If you need to update your SD card image, refer to the Hardware Setup section of this document.

4. Set up the Zynq hardware connection by entering the following command in the MATLAB command window:

```
h = zynq
```

The `zynq` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

5. You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: 115200; Data bits: 8; Stop bits: 1; Parity: None; Flow control: None. You should be able to observe Linux booting log on the serial console when you power cycle the Zynq board. You must close this serial connection before using the `zynq` function again.

6. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.ba
```

Partition your design for hardware and software implementation

The first step of the Zynq hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

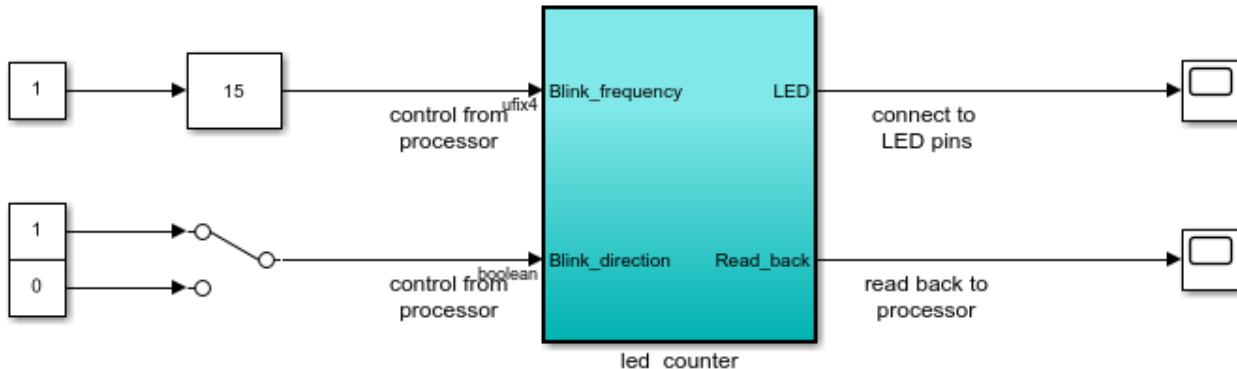
Group all the blocks you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem will be implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

Launch HDL Workflow Advisor

Run Demo

Copyright 2012 The MathWorks, Inc.

Generate an HDL IP core using the HDL Workflow Advisor

Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Xilinx Vivado environment.

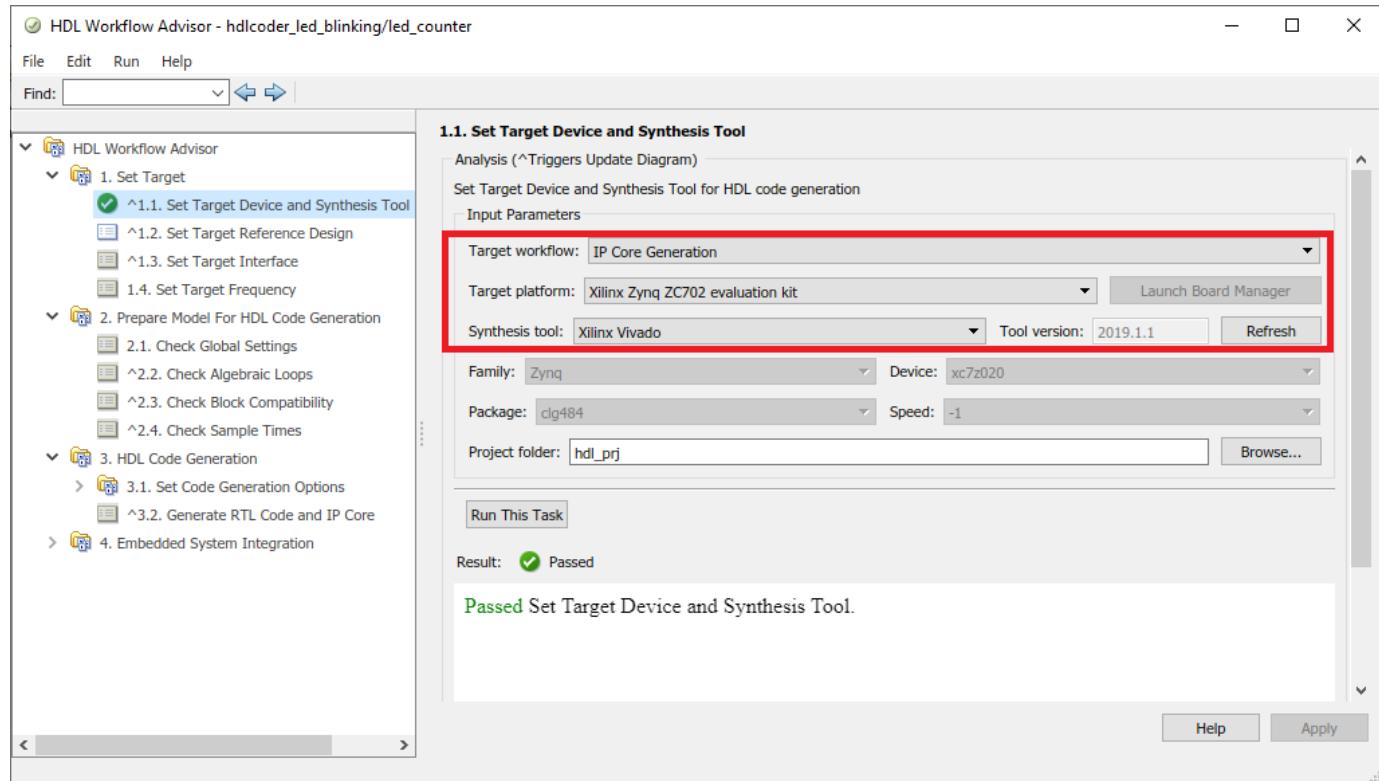
1. Start the IP core generation workflow.

1.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code > HDL Workflow Advisor**.

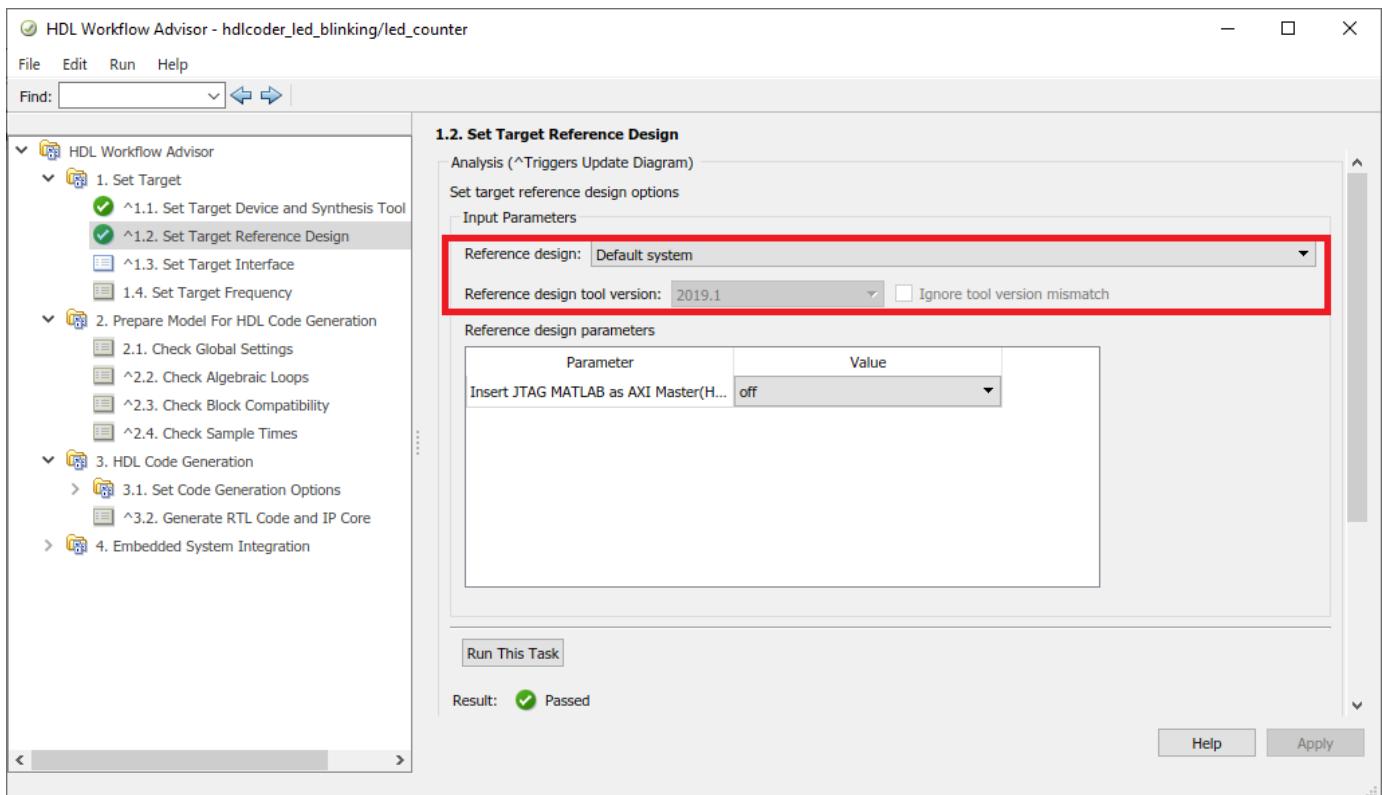
1.2. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

1.3. For **Target platform**, select **Xilinx Zynq ZC702 evaluation kit**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx Zynq Platform and follow the instructions provided by the Support Package Installer to complete the installation.

1.4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



1.5 In the **Set Target > Set Target Reference Design** task, choose **Default system**.



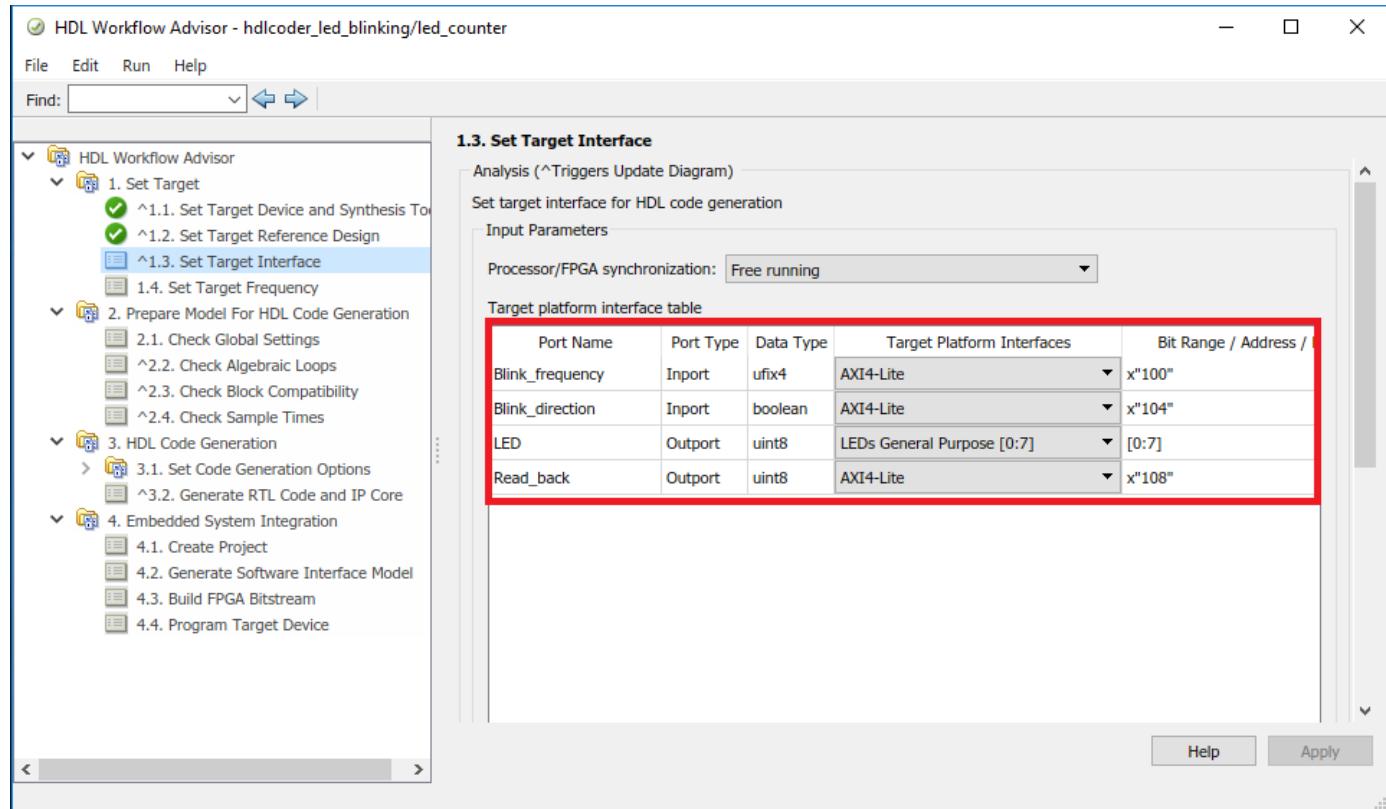
1.6. Click Run This Task to run the Set Target Reference Design task.

2. Configure the target interface.

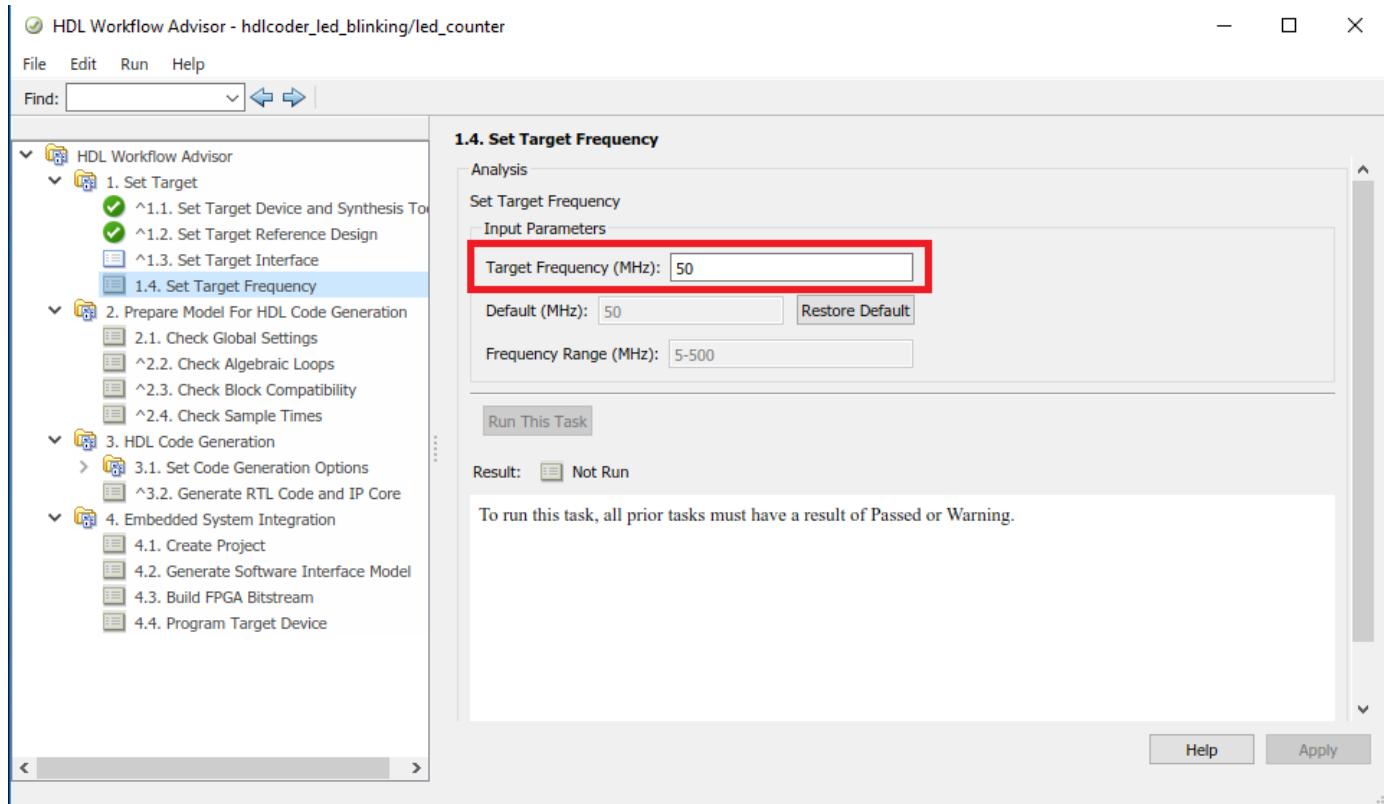
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:7]**, which connects to the LED hardware on the Zynq board.

2.1 In the **Set Target > Set Target Interface** task, choose **AXI4-Lite** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2.2 Choose **LEDs General Purpose [0:7]** for **LED**.

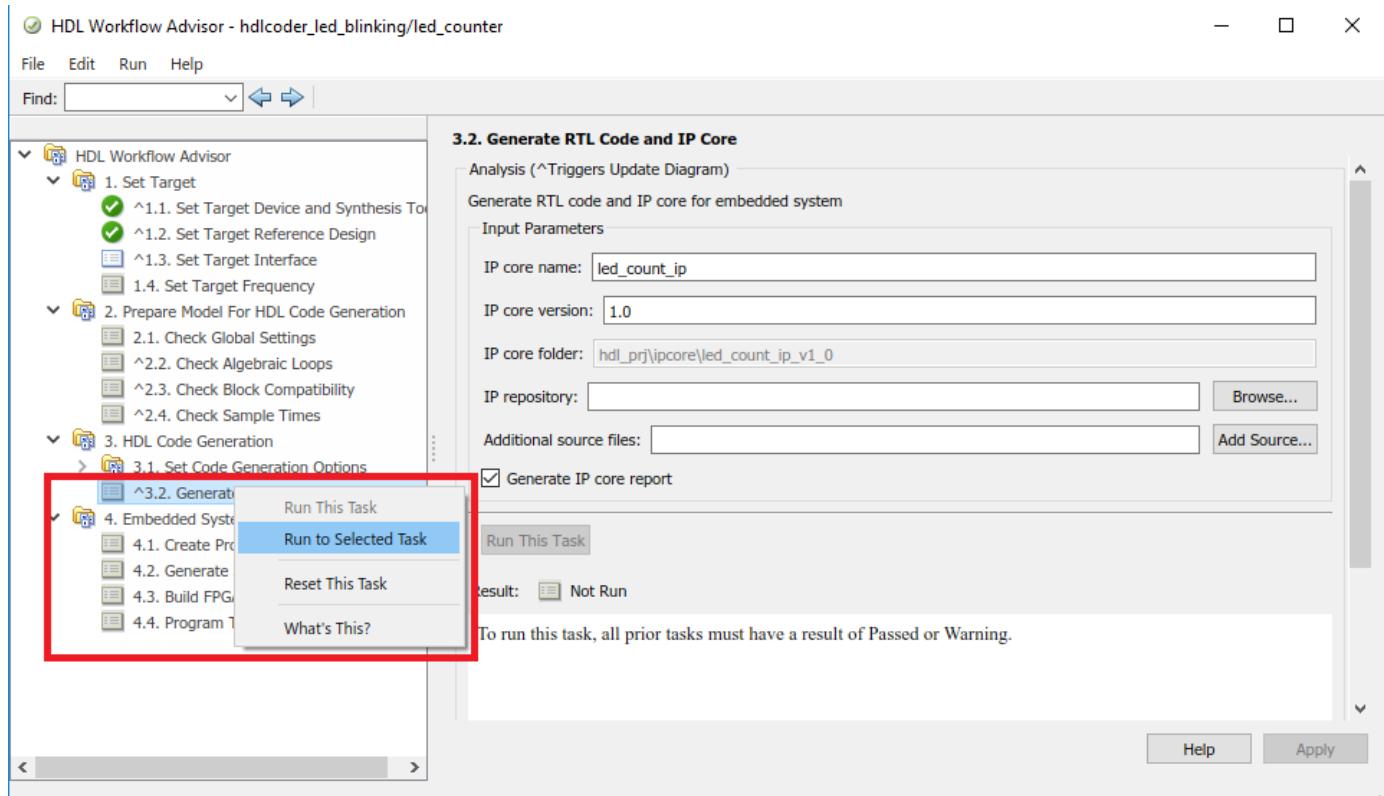


2.3 In the **Set Target > Set Target Frequency** task, choose **Target Frequency as 50 MHz**.



3. Generate the IP Core.

To generate the IP core, right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



4. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated together with the custom IP core. The report describes the behavior and contents of the generated custom IP core.

Code Generation Report

Find: Match Case

Contents

- [Summary](#)
- [Clock Summary](#)
- [Code Interface Report](#)
- [Timing And Area Report](#)
- [High-level Resource Report](#)
- [Optimization Report](#)
- [Distributed Pipelining](#)
- [Streaming and Sharing](#)
- [Delay Balancing](#)
- [Adaptive Pipelining](#)
- [IP Core Generation Report](#)
- [Traceability Report](#)

Generated Source Files

- [led_count_ip_src_led_counter_pkg.vhd](#)
- [led_count_ip_src_led_counter.vhd](#)

IP Core User Guide

Theory of Operation

This IP core is designed to be connected to an embedded processor with an **AXI4-Lite interface**. The processor acts as master, and the IP core acts as slave. By accessing the generated registers via the AXI4-Lite interface, the processor can control the IP core, and read and write data from and to the IP core.

For example, to reset the IP core, write 0x1 to the bit 0 of `IPCore_Reset` register. To enable or disable the IP core, write 0x1 or 0x0 to the `IPCore_Enable` register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.

```

graph LR
    PS[Processing System] <-->|AXI4-Lite| ALR[AXI4-Lite Accessible Registers]
    ALR <-->|Algorithm from MATLAB/Simulink| EP[External Ports]
    
```

This IP core also support the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx Vivado environment.

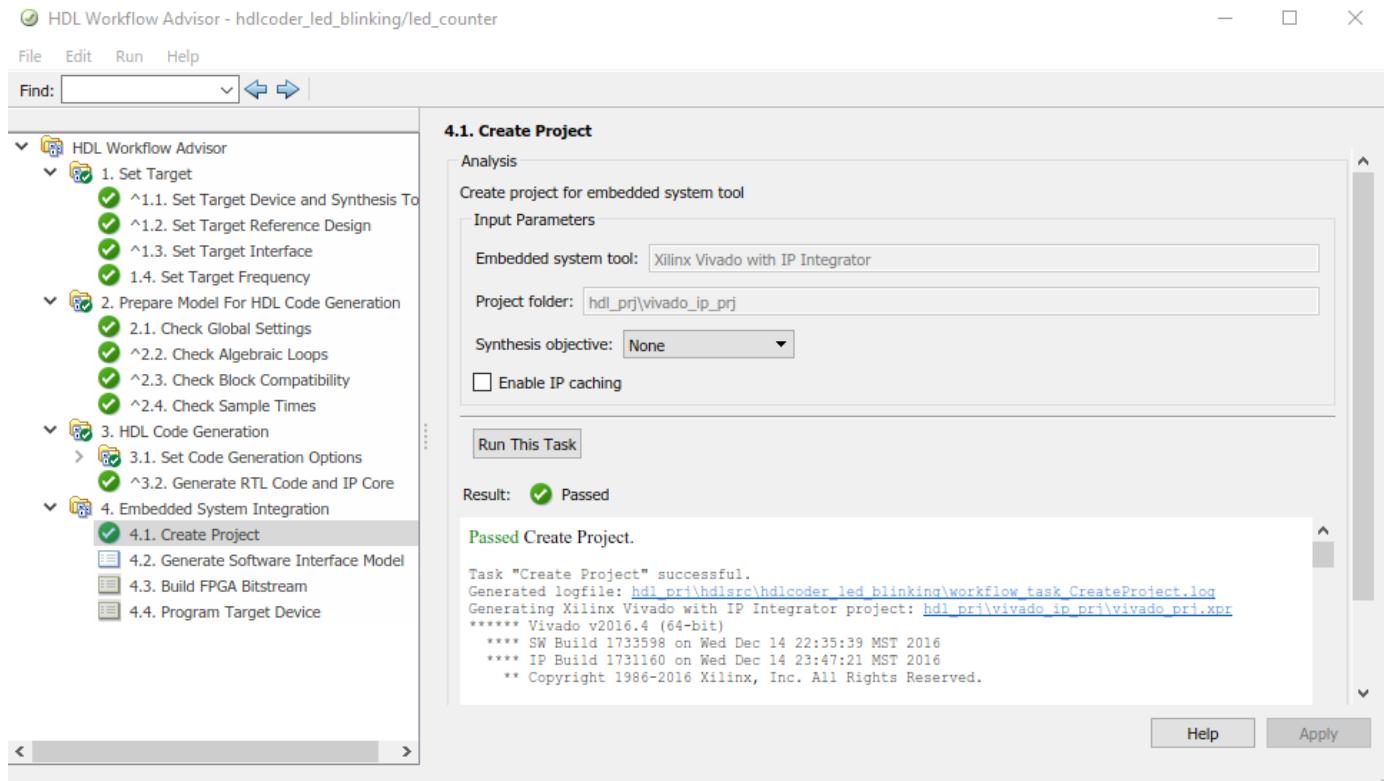
OK Help

Integrate the IP core with the Xilinx Vivado environment

In this part of the workflow, you insert your generated IP core into a embedded system reference design, generate an FPGA bitstream, and download the bitstream to the Zynq hardware.

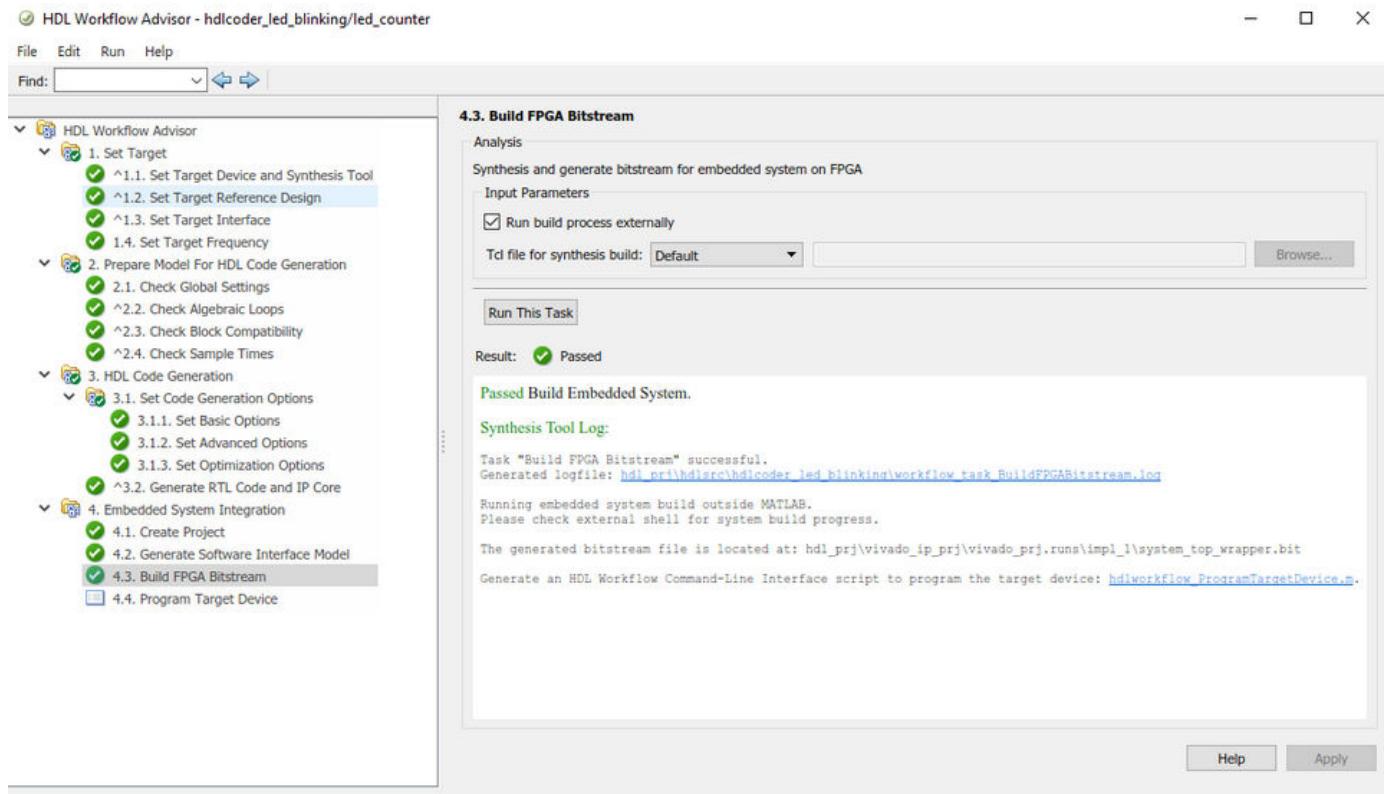
The reference design is a predefined Xilinx Vivado project. It contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software that you generate.

1. To integrate with the Xilinx Vivado environment, select the **Create Project** task under **Embedded System Integration**, and click **Run This Task**. A Xilinx Vivado project with IP Integrator embedded design is generated, and a link to the project is provided in the dialog window. You can optionally open up the project to take a look.

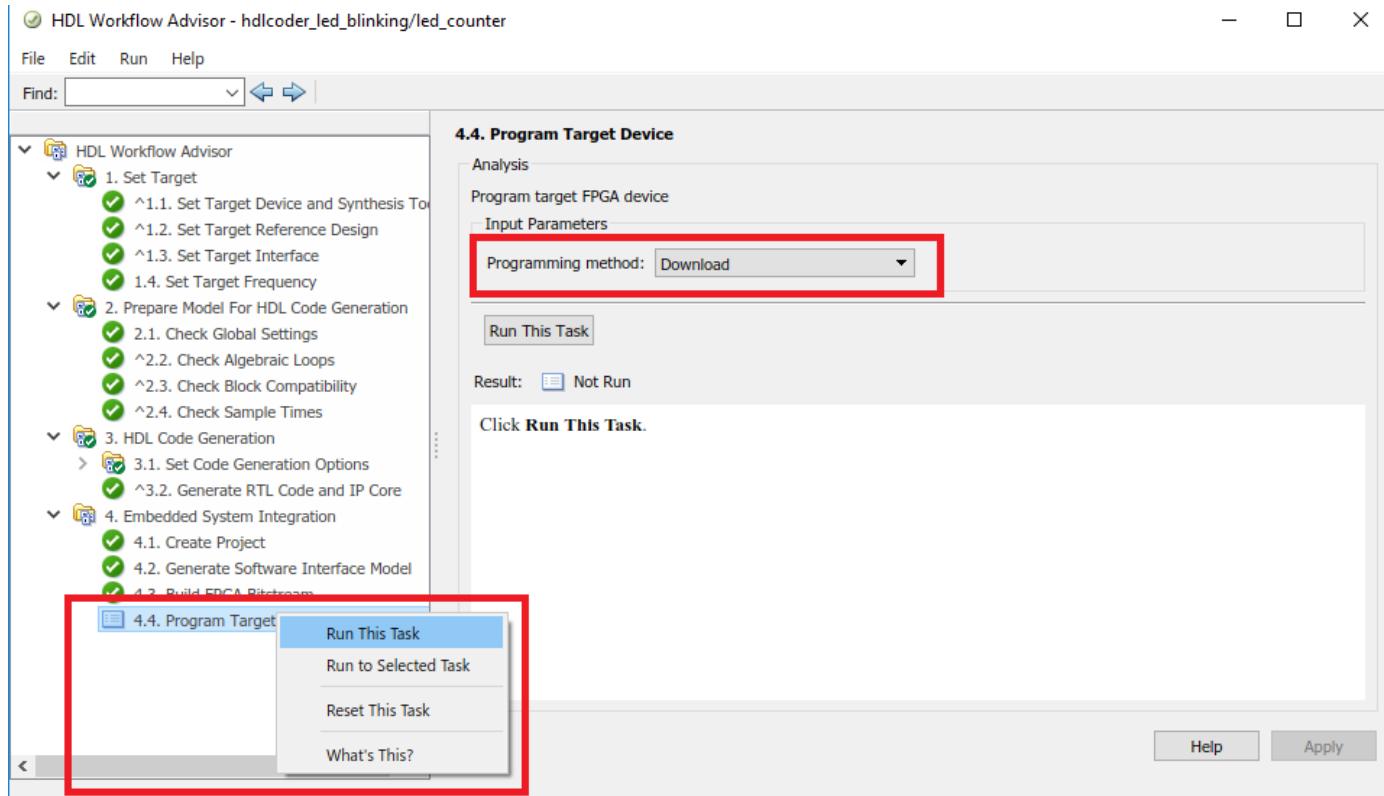


2. If you have an Embedded Coder license, you can generate a software interface model in the next task, **Generate Software Interface Model**. The details of the software interface model are explained in the next section of this example, "Generate a software interface model".

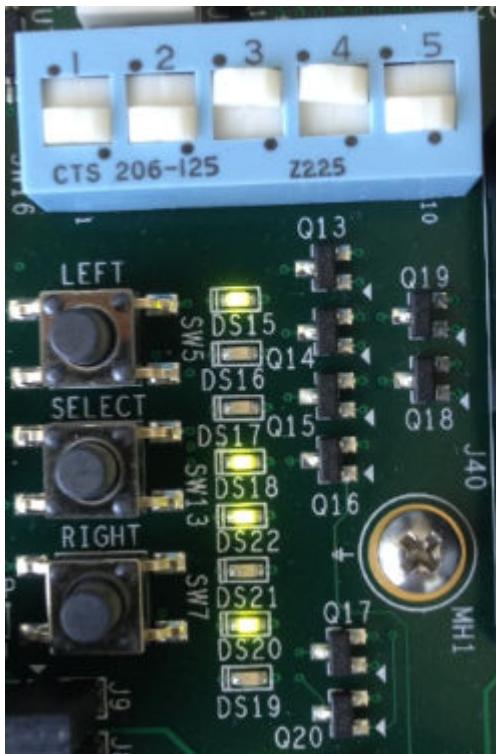
3. Build the FPGA bitstream in the **Build FPGA Bitstream** task. Make sure the **Run build process externally** option is checked, so the Xilinx synthesis tool will run in a separate process from MATLAB. Wait for the synthesis tool process to finish running in the external command window.



4. After the bitstream is generated, select the **Program Target Device** task. Choose **Download** for **Programming method** to download the FPGA bitstream onto the SD card on the Zynq board, so your design will be automatically reloaded when you power cycle the Zynq board. click **Run This Task** to program the Zynq hardware.



After you program the FPGA hardware, the LED starts blinking on your Zynq board.



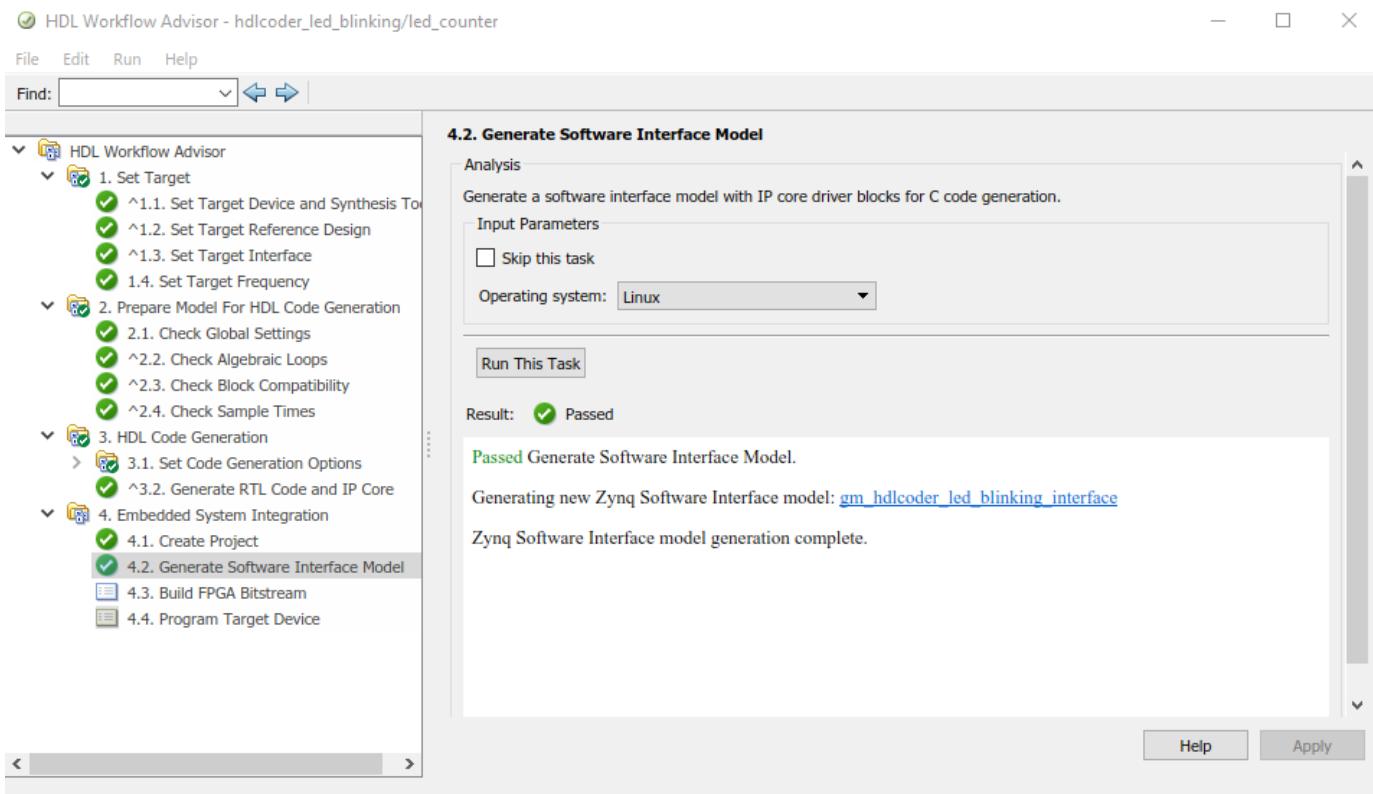
Next, you will generate C code to run on the ARM processor to control the LED blink frequency and direction.

Generate a software interface model

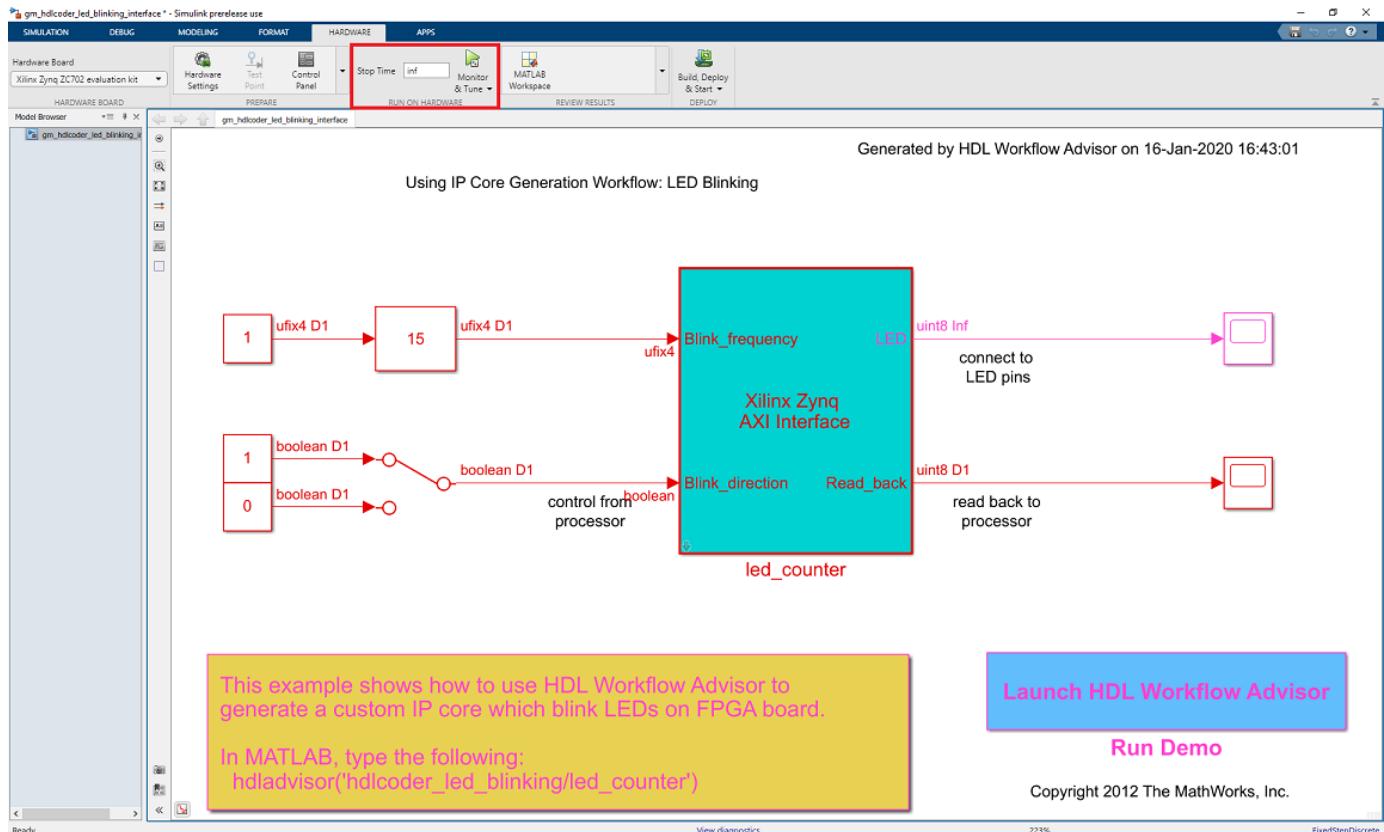
In the HDL Workflow Advisor, after you generate the IP core and insert it into the Vivado reference design, you can optionally generate a software interface model in the **Embedded System Integration > Generate Software Interface Model** task.

The software interface model contains the part of your design that runs in software. It includes all the blocks outside of the HDL subsystem, and replaces the HDL subsystem with AXI driver blocks. If you have an Embedded Coder license, you can automatically generate embedded code from the software interface model, build it, and run the executable on Linux on the ARM processor. The generated embedded software includes AXI driver code, generated from the AXI driver blocks, that controls the HDL IP core.

Run the **Generate Software Interface Model** task and see that a new model is generated. The task dialog shows a link to the model.



In the generated software interface model, the "led_counter" subsystem is replaced with the AXI driver blocks which generate the interface logic between the ARM processor and FPGA.

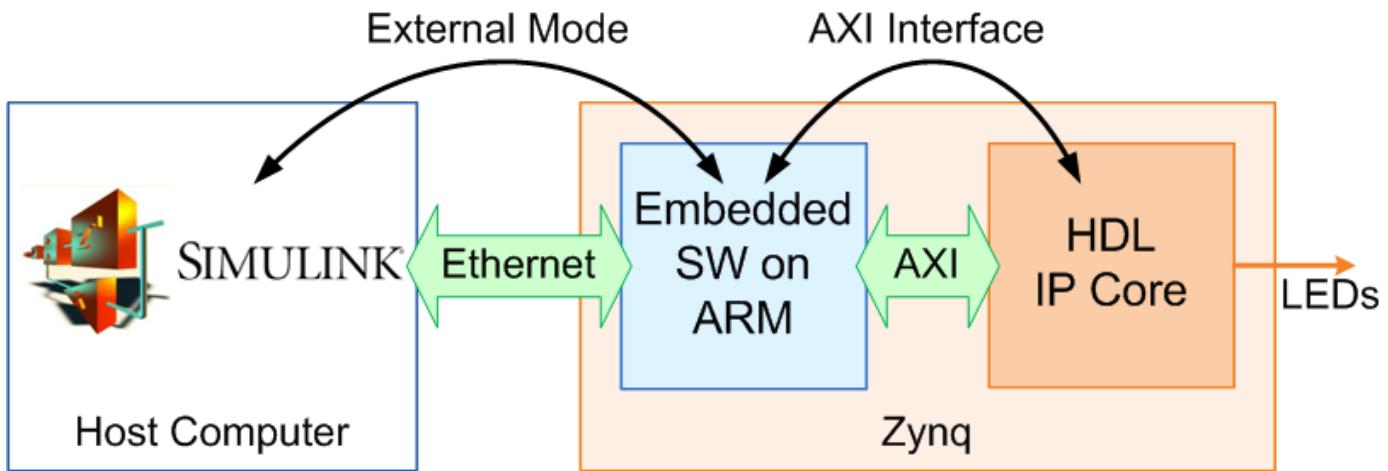


Run the software interface model on Zynq ZC702 hardware

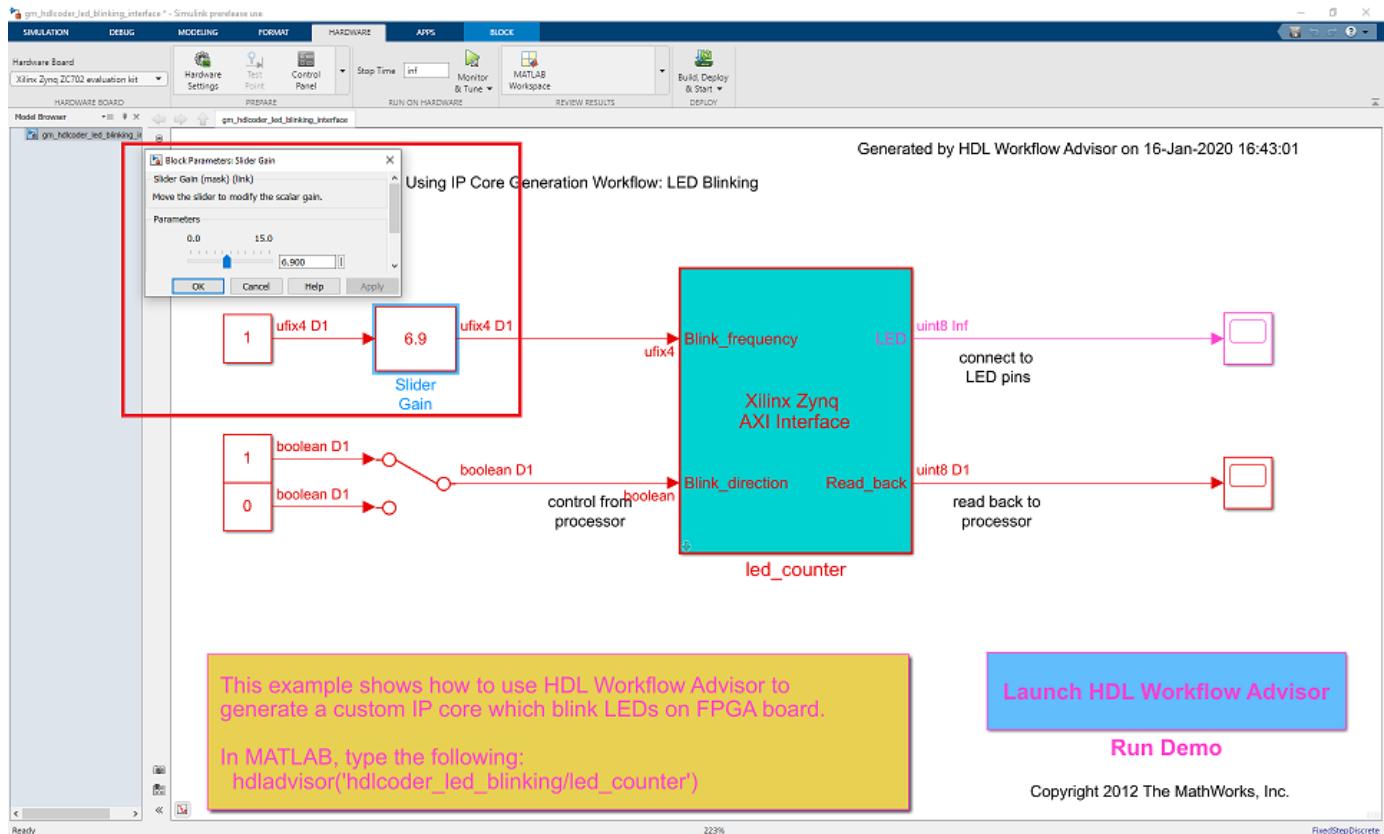
In this part of the workflow, you configure the generated software interface model, automatically generate embedded C code, and run your model on the ARM processor in the Zynq hardware in External mode.

When you are prototyping and developing an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. The External mode feature in Simulink enables this capability. In this mode, your algorithm is first deployed to the ARM processor in the Zynq hardware, and then linked with the Simulink model on the host computer through an Ethernet connection.

The main role of the Simulink model is to tune and monitor the algorithm running on the hardware. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use External mode to tune parameters, and capture data from the FPGA.



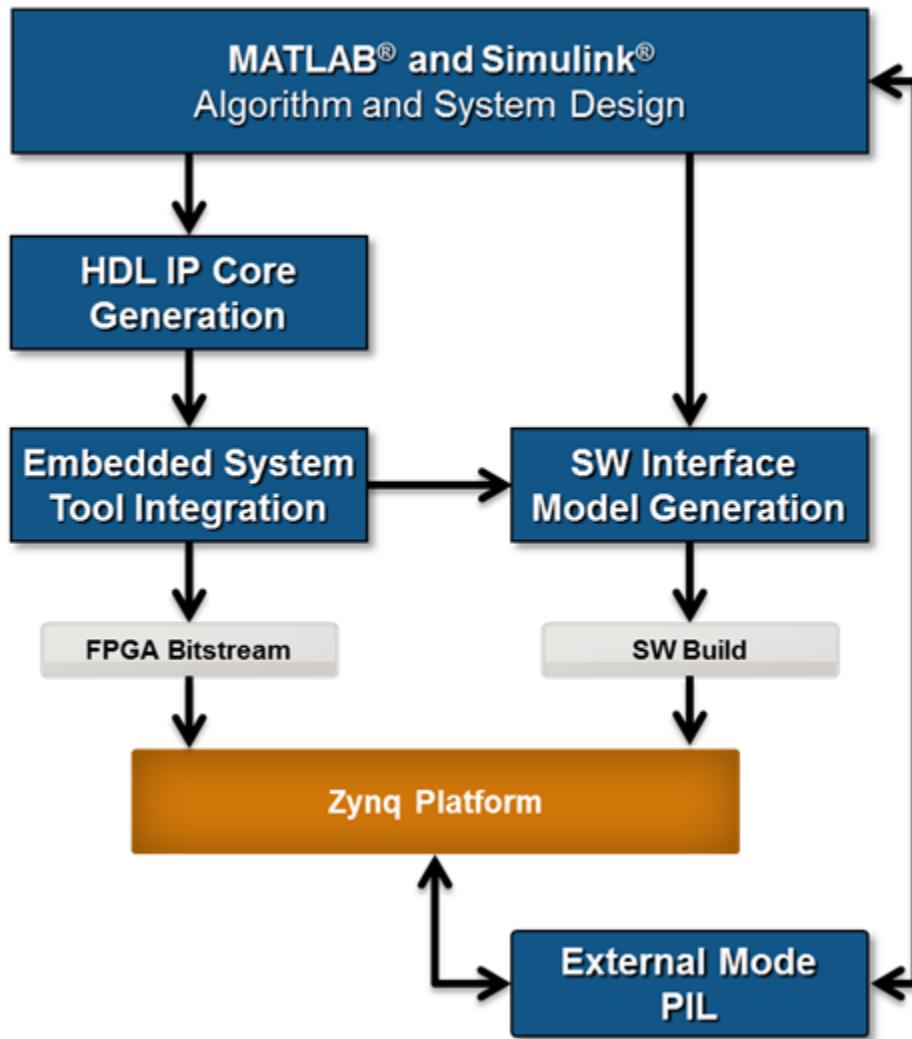
- 1 In the generated model, click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
- 2 Select **Solver** and set "Stop Time" to "inf".
- 3 From the **HARDWARE** pane, click the **Monitor & Tune** button on the model toolbar to run your model on the ARM processor in the Zynq ZC702 hardware in External mode. Embedded Coder builds the model, downloads the ARM executable to the Zynq ZC702 hardware, executes it, and connects the model to the executable running on the Zynq ZC702 hardware.
- 4 Double-click the **Slider Gain** block. Change the Slider Gain value and observe the change in frequency of the LED array blinking on the Zynq ZC702 hardware. Double-click the **Manual Switch** block to switch the direction of the blinking LEDs.
- 5 Double-click the scope connected to the **Read_back** output port and observe that the output data of the FPGA IP core is captured and sent back to the Simulink scope.
- 6 When you are done changing model parameters, click the **Stop** button on the model. Observe that the system command window opened in the previous step indicates that the model has been stopped. At this point, you can close the system command window.



Summary

This example shows how the hardware and software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Zynq-7000 All Programmable SoC. You can explore the best ways to partition and deploy your design by iterating through the workflow.

The following diagram shows the high-level picture of the workflow you went through in this example. To learn more about the hardware and software co-design workflow, please refer to the HDL Coder documentation.



Getting Started with Targeting Zynq UltraScale+ MPSoC Platform

This example shows how to use the hardware-software co-design workflow to blink LEDs at various frequencies on the Xilinx® Zynq® UltraScale+ MPSoC.

Introduction

This example is a step-by-step guide that helps you use the HDL Coder™ software to generate a custom HDL IP core which blinks LEDs on the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Xilinx Zynq UltraScale+ MPSoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM Cortex-A53.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic using HDL Coder, generate C code for the ARM processor using Embedded Coder, and implement the design on the Xilinx Zynq UltraScale+ MPSoC Platform.

In this workflow, you perform the following steps:

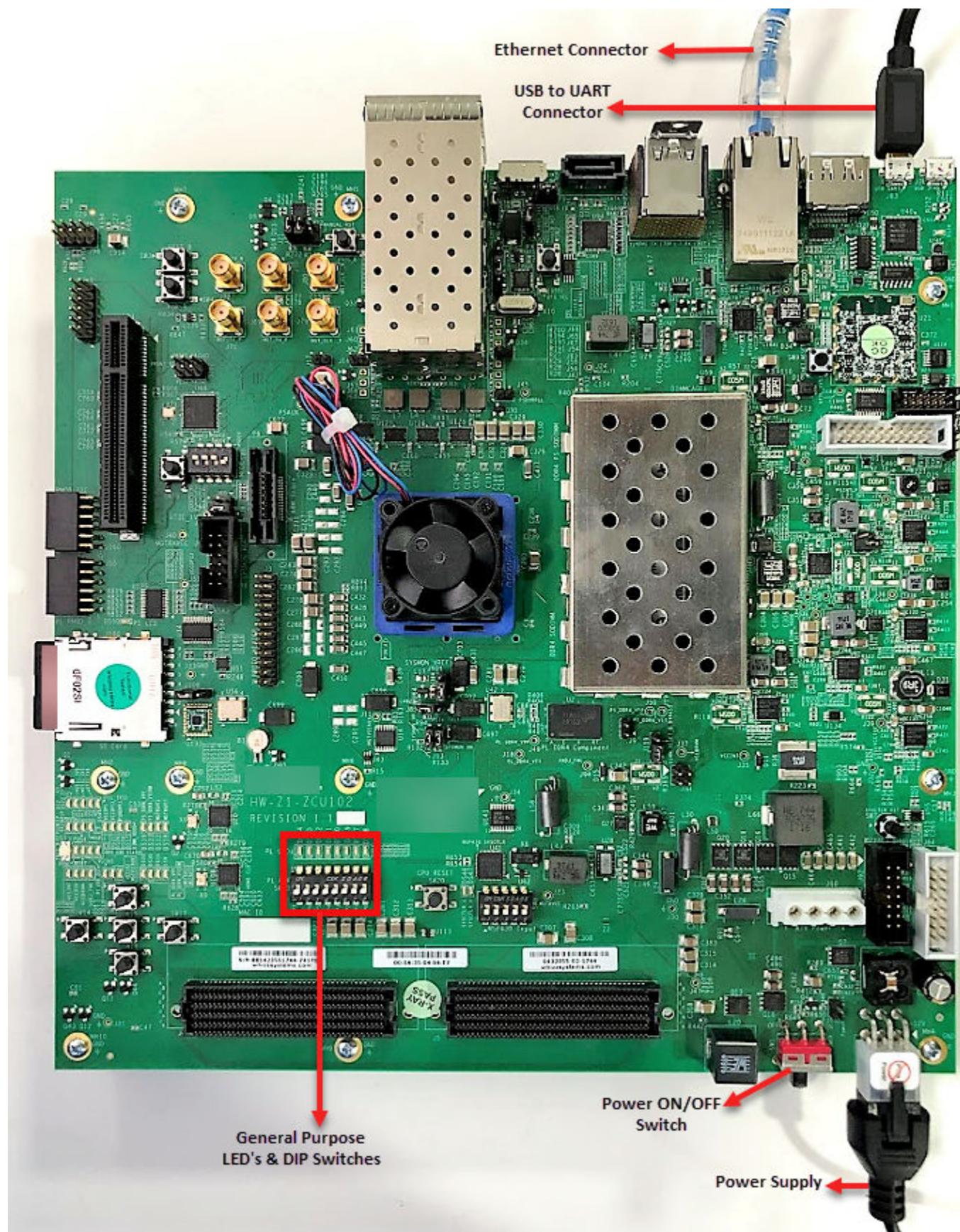
- 1 Set up your Xilinx Zynq UltraScale+ MPSoC ZCU102 hardware and tools.
- 2 Partition your design for hardware and software implementation.
- 3 Generate an HDL IP core using HDL Workflow Advisor.
- 4 Integrate the IP core into a Xilinx Vivado project and program the Xilinx Zynq UltraScale+ MPSoC hardware.
- 5 Generate a software interface model.
- 6 Generate C code from the software interface model and run it on the ARM Cortex-A53 processor.
- 7 Tune parameters and capture results from the Zynq hardware using External Mode.

Requirements

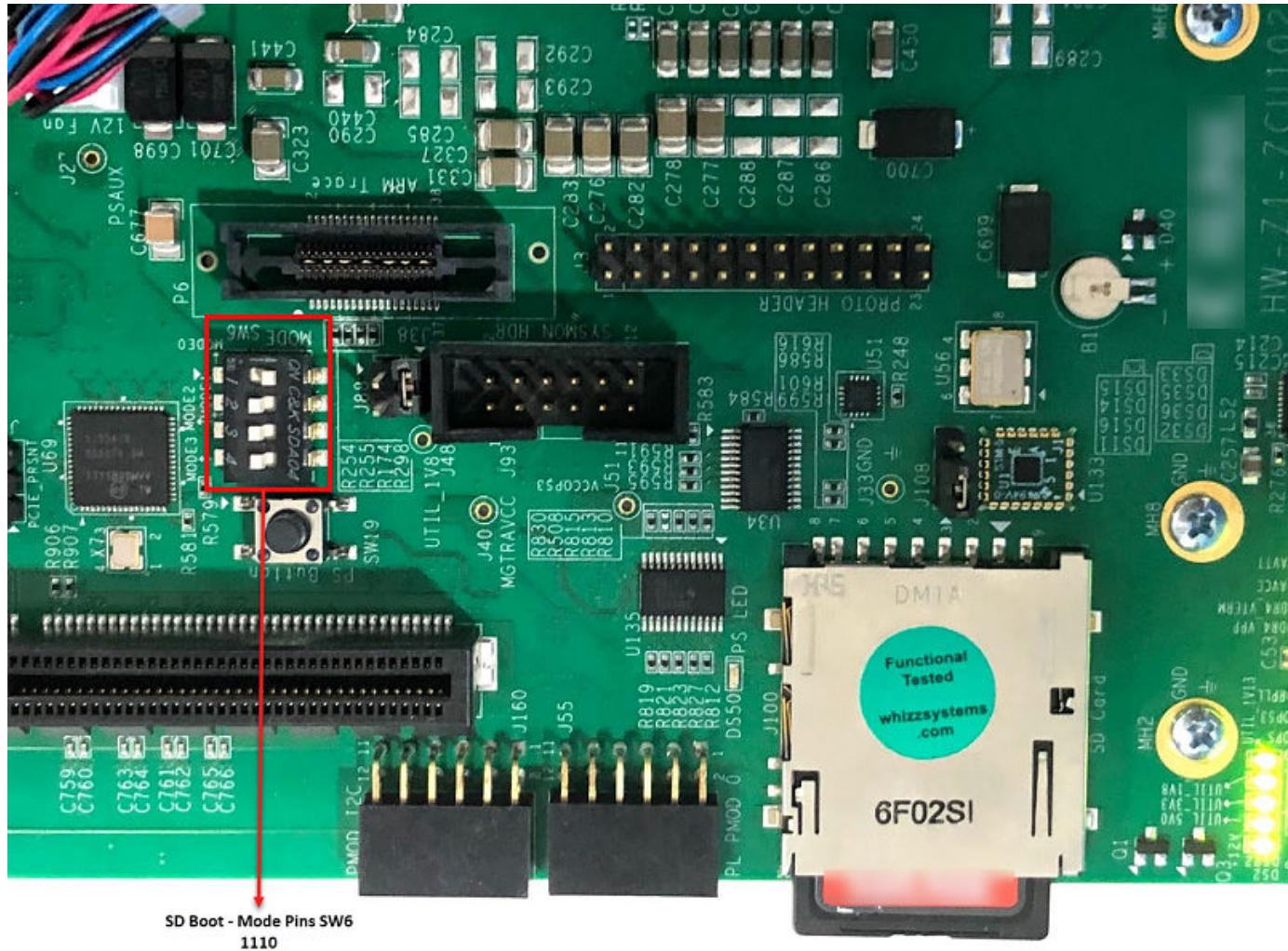
- 1 Xilinx Vivado Design Suite, with supported version listed in the HDL Coder documentation
- 2 Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
- 3 HDL Coder Support Package for Xilinx Zynq Platform
- 4 Embedded Coder Support Package for Xilinx Zynq Platform

Set up your Xilinx Zynq UltraScale+ MPSoC hardware and tools

1. Set up the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation kit as shown in the figure below. To learn more about the ZCU102 hardware setup, please refer to Xilinx documentation.



1.1. Make sure the SW6 switch is set as shown in the figure below, so you can boot up Linux from the SD card.



1.2 Connect your computer to the USB UART connector of ZCU102 using a Micro-USB cable. Make sure your USB device drivers, such as for the Silicon Labs CP210x USB to UART Bridge, are installed correctly. If not, search for the drivers online and install them.

1.3 Connect Xilinx Zynq UltraScale+ MPSoC board to your computer using an Ethernet cable.

2. Install the HDL Coder and Embedded Coder Support Packages for Xilinx Zynq Platform if you haven't already.

2.1 On the MATLAB **Home** tab in the **Environment** section, Click Add-Ons > Manage Add-Ons.

2.2 In the Add-On Manager, start the hardware setup process by clicking the setup button for Embedded Coder Support Package for Xilinx Zynq Platform.

3. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Xilinx Zynq Platform.

- 4.** Set up the Zynq hardware connection by entering the following command in the MATLAB command window:

```
h = zynq
```

The `zynq` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

- 5.** You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: 115200; Data bits: 8; Stop bits: 1; Parity: None; Flow control: None. You should be able to observe Linux booting log on the serial console when you power cycle the MPSoC board. You must close this serial connection before using the `zynq` function again.

- 6.** Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat')
```

Partition your design for hardware and software implementation

The first step of the Zynq hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

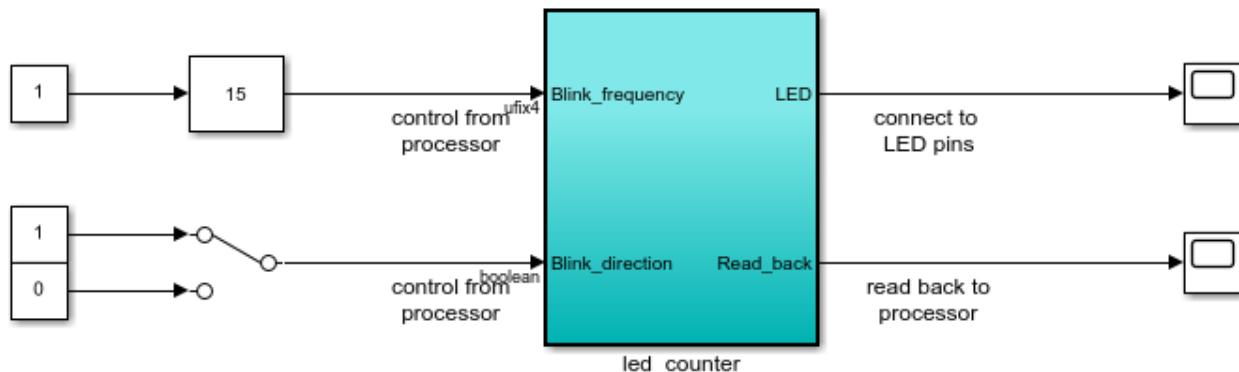
Group all the blocks you want to implement on programmable logic into an Atomic Subsystem. This Atomic Subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem will be implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are used for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

Launch HDL Workflow Advisor

Run Demo

Copyright 2012 The MathWorks, Inc.

Generate an HDL IP core using the HDL Workflow Advisor

Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Xilinx Vivado environment.

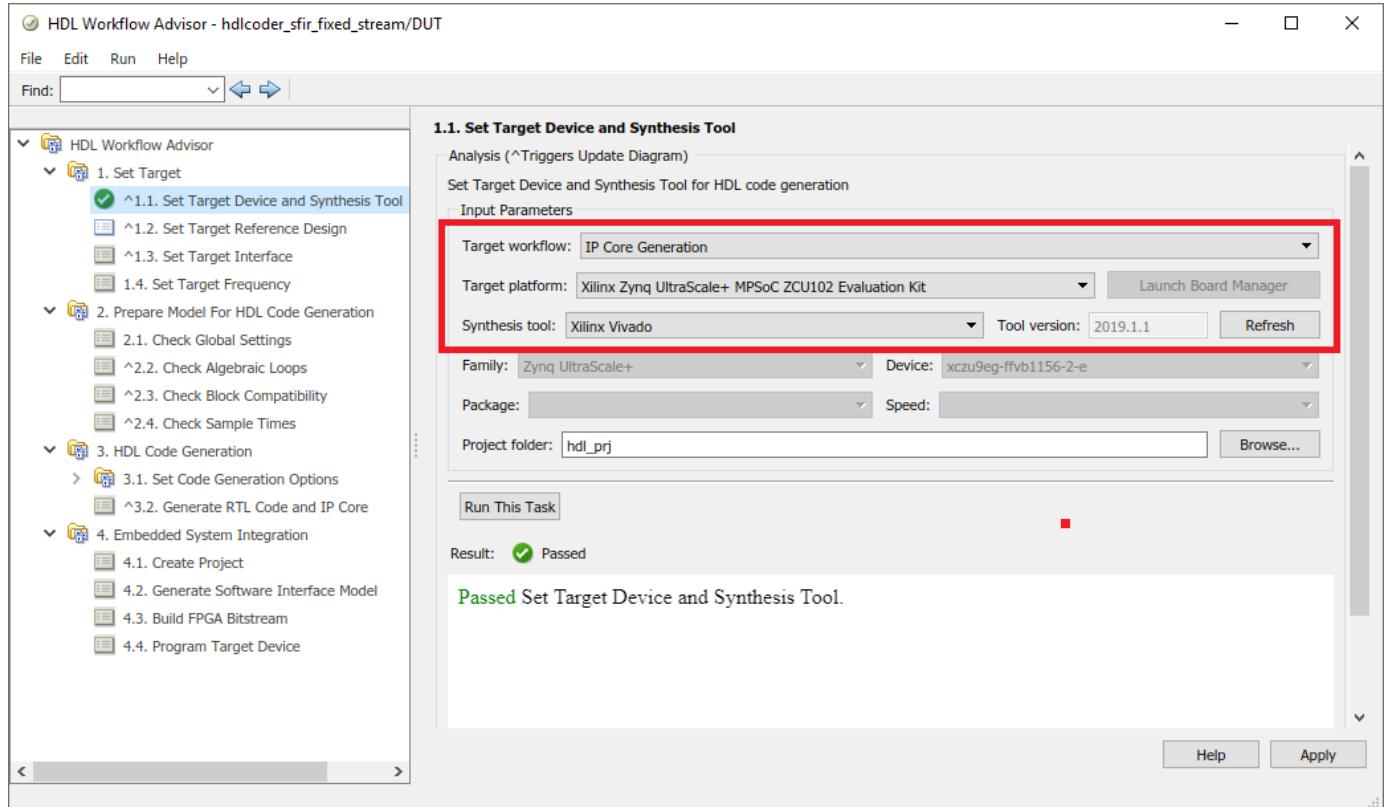
1. Start the IP core generation workflow.

1.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code > HDL Workflow Advisor**.

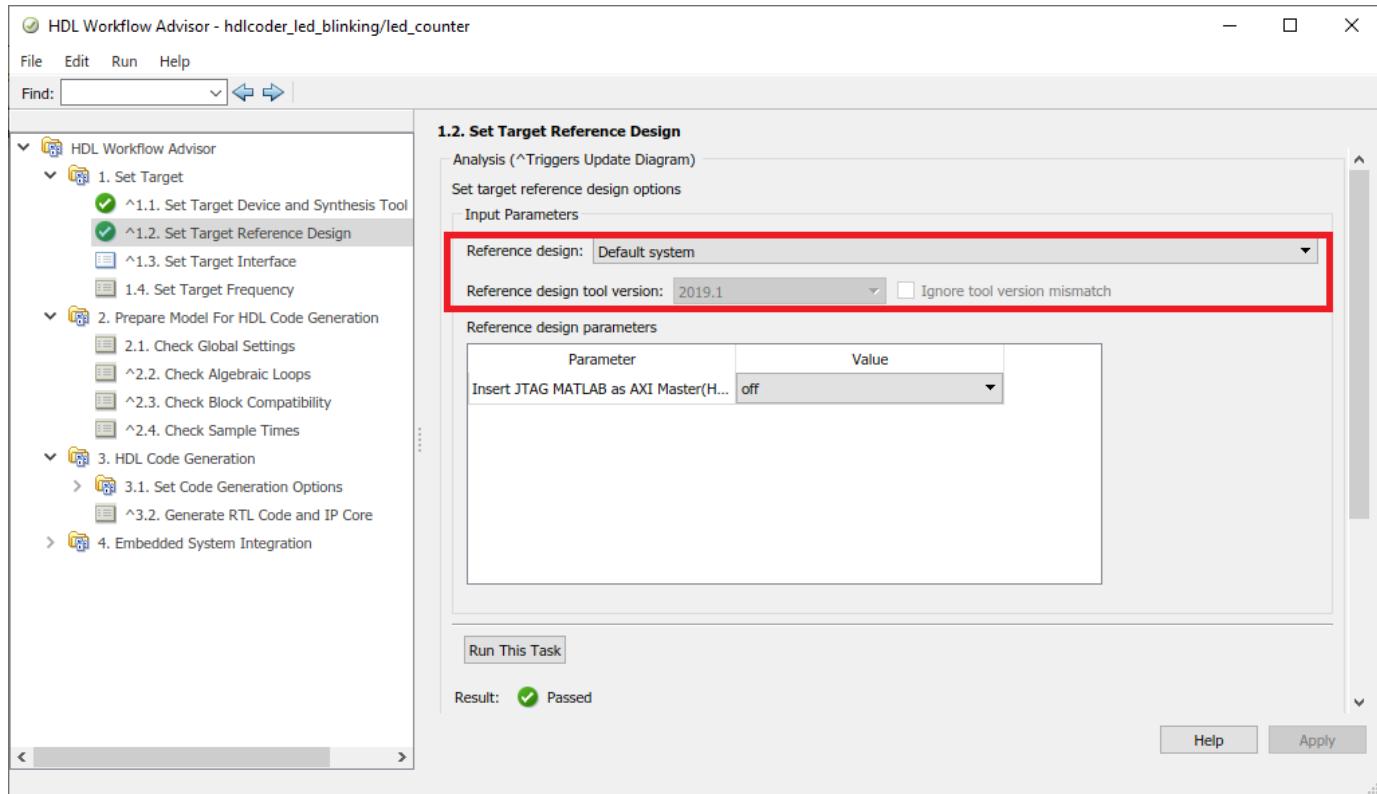
1.2. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

1.3. For **Target platform**, select **Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx Zynq Platform and follow the instructions provided by the Support Package Installer to complete the installation.

1.4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



1.5 In the **Set Target > Set Target Reference Design** task, choose **Default system**.



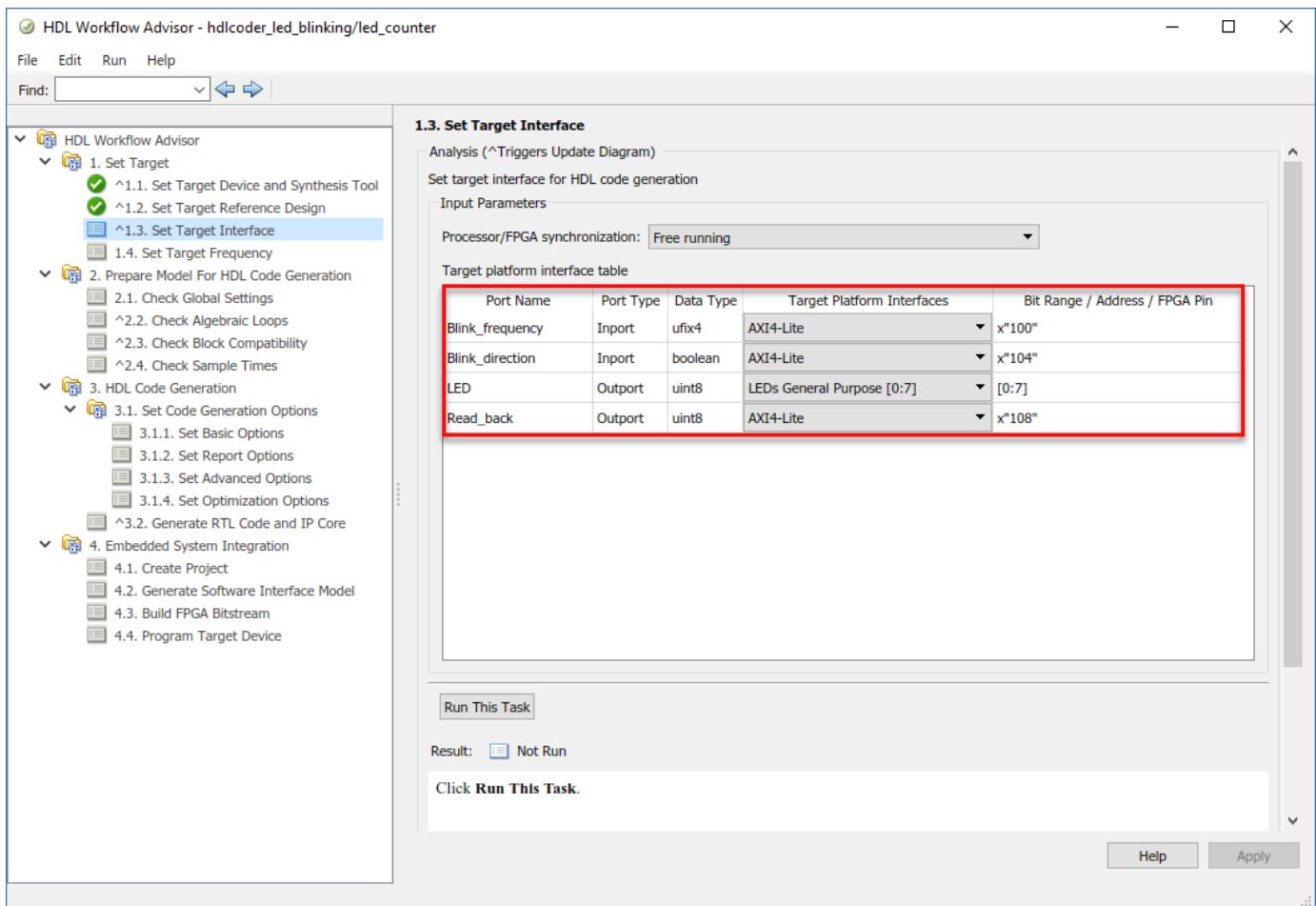
1.6. Click Run This Task to run the Set Target Reference Design task.

2. Configure the target interface.

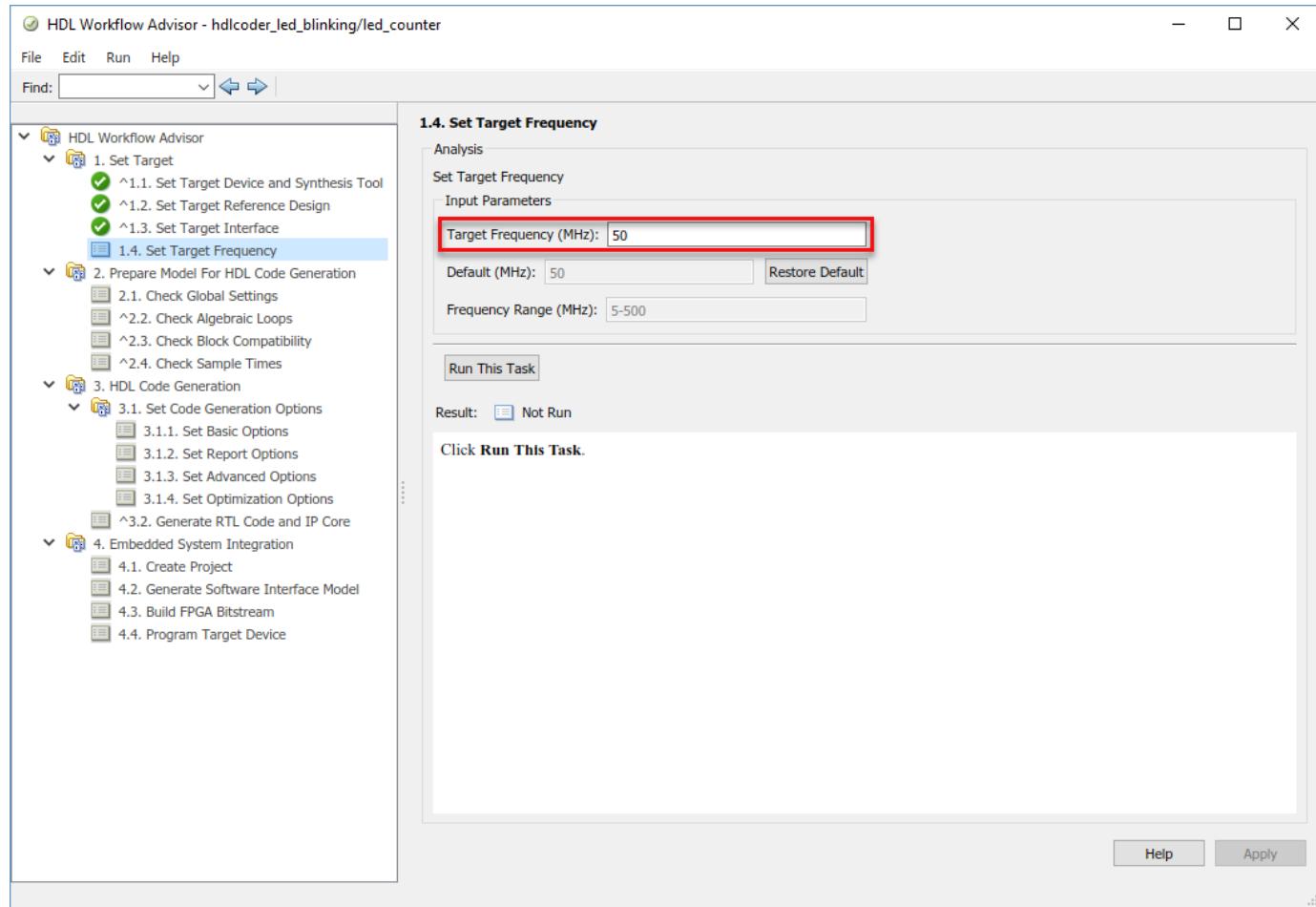
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:7]**, which connects to the LED hardware on the Zynq board.

2.1 In the **Set Target > Set Target Interface** task, choose **AXI4-Lite** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2.2 Choose **LEDs General Purpose [0:7]** for **LED**.

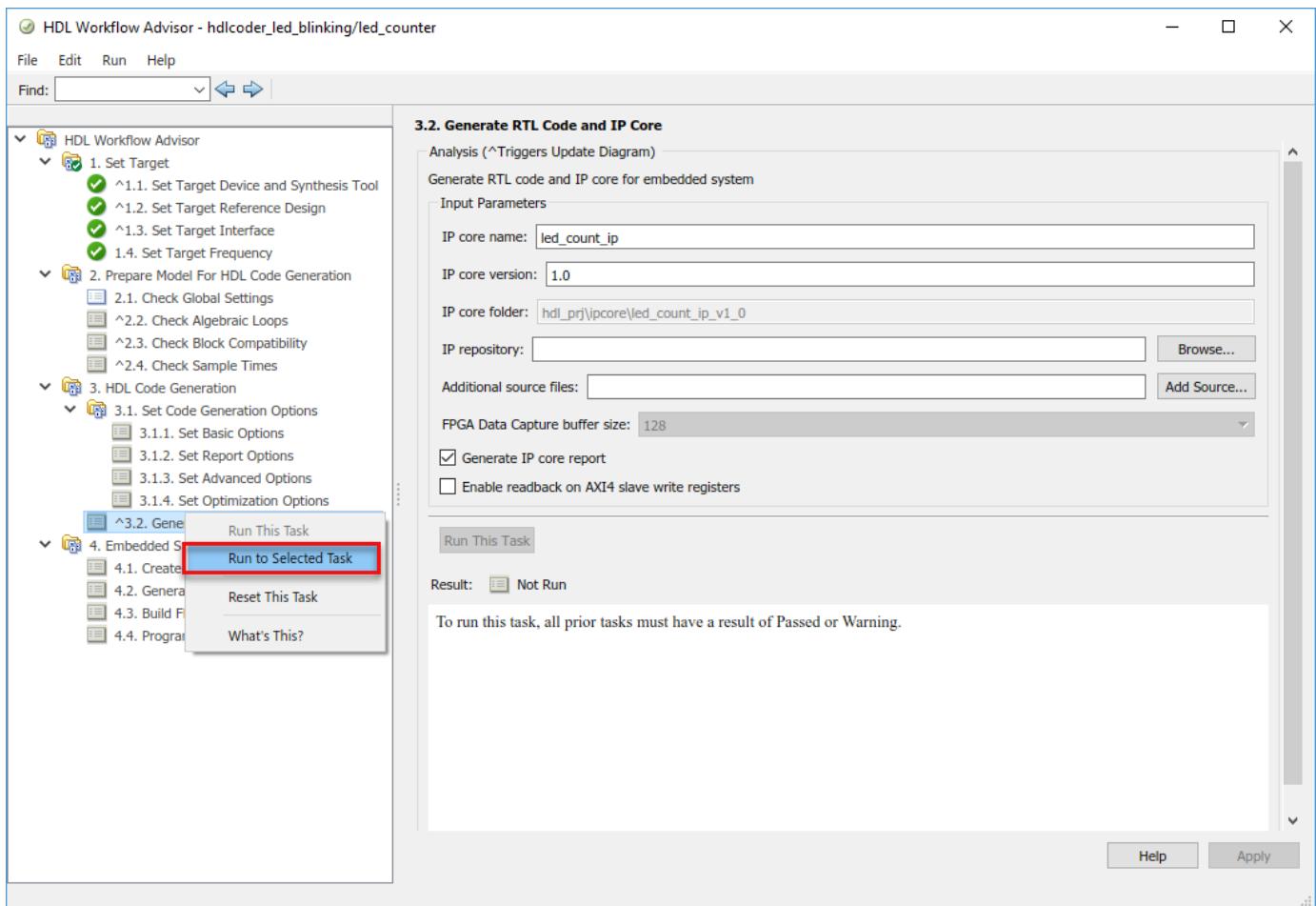


2.3 In the **Set Target > Set Target Frequency** task, choose **Target Frequency as 50 MHz**.



3. Generate the IP Core.

To generate the IP core, right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



4. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated together with the custom IP core. The report describes the behavior and contents of the generated custom IP core.

Code Generation Report

Find: Match Case

Contents

- [Summary](#)
- [Clock Summary](#)
- [Code Interface Report](#)
- [Timing And Area Report](#)
- [High-level Resource Report](#)
- [Optimization Report](#)
- [Distributed Pipelining](#)
- [Streaming and Sharing](#)
- [Delay Balancing](#)
- [Adaptive Pipelining](#)
- [IP Core Generation Report](#)
- [Traceability Report](#)

Generated Source Files

- [led_count_ip_src_led_counter_pk](#)
- [led_count_ip_src_led_counter.vh](#)

Referenced Models

IP Core User Guide

Theory of Operation

This IP core is designed to be connected to an embedded processor with an **AXI4-Lite interface**. The processor acts as master, and the IP core acts as slave. By accessing the generated registers via the AXI4-Lite interface, the processor can control the IP core, and read and write data from and to the IP core.

For example, to reset the IP core, write 0x1 to the bit 0 of IPCore_Reset register. To enable or disable the IP core, write 0x1 or 0x0 to the IPCore_Enable register. To access the data ports of the MATLAB/Simulink algorithm, read or write to the associated data registers.

```

graph LR
    PS[Processing System] <--> AL[AXI4-Lite Accessible Registers]
    AL <--> AS[Algorithm from MATLAB/Simulink]
    AL --> EP[External Ports]
  
```

This IP core also support the **External Port** interface. To connect the external ports to the FPGA external IO pins, add FPGA pin assignment constraints in the Xilinx Vivado environment.

Processor/FPGA Synchronization

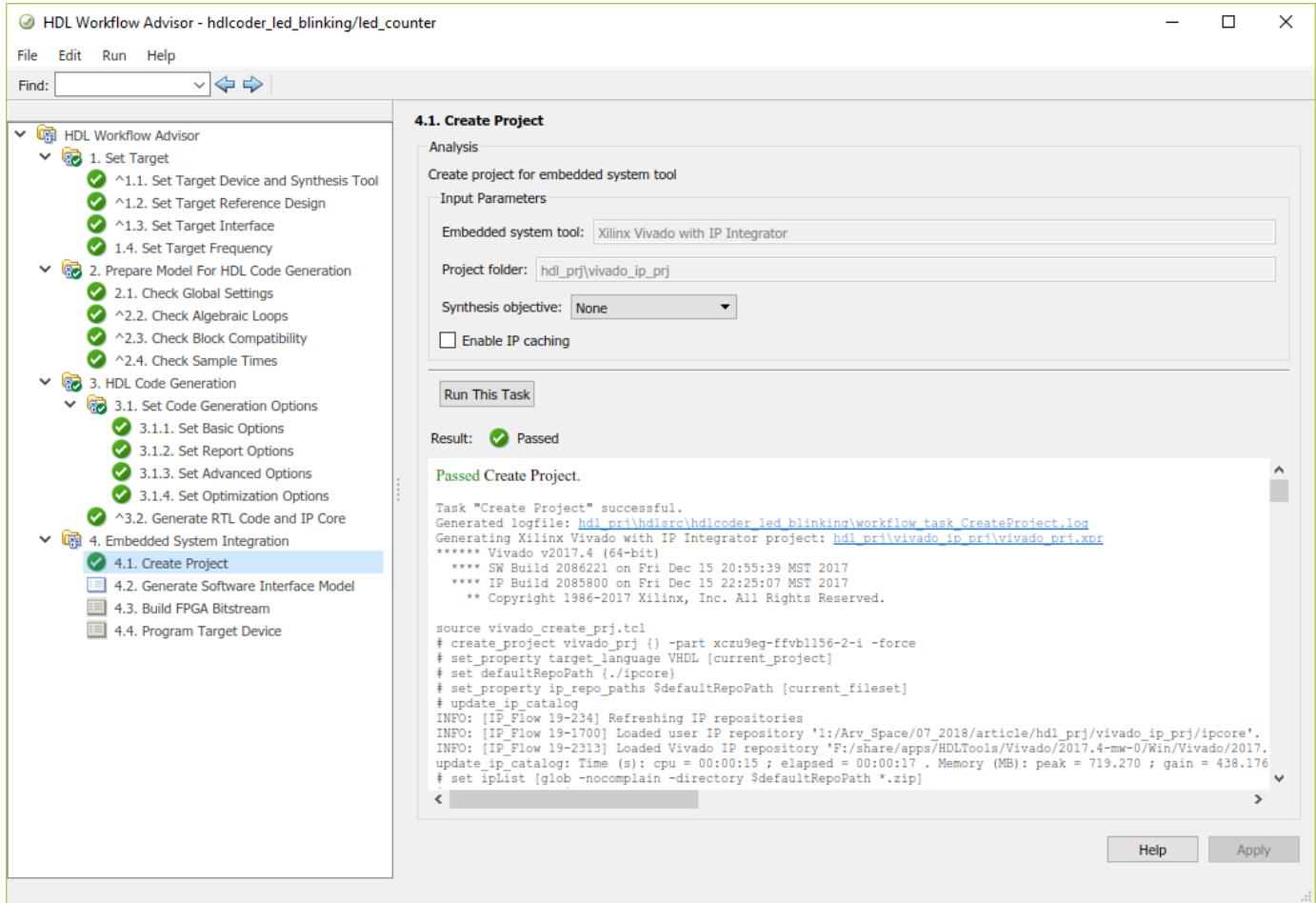
The **Free running** mode means there is no explicit synchronization between embedded processor software execution (SW) and the IP core (HW). SW and HW runs independently. The data written from the processor to IP core takes effect immediately, and the data read from the IP core is the latest

Integrate the IP core with the Xilinx Vivado environment

In this part of the workflow, you insert your generated IP core into a embedded system reference design, generate an FPGA bitstream, and download the bitstream to the Zynq hardware.

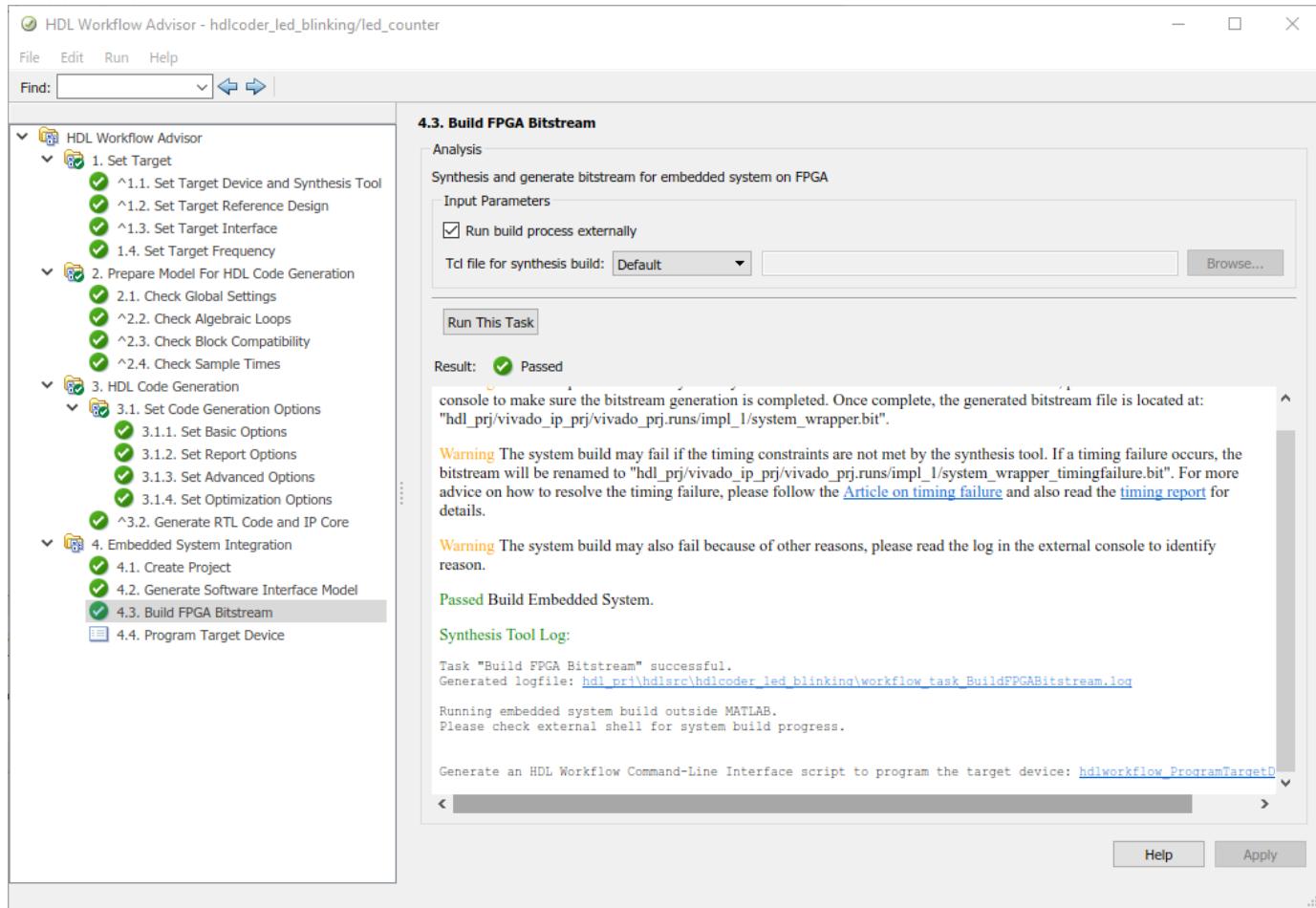
The reference design is a predefined Xilinx Vivado project. It contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software that you generate.

1. To integrate with the Xilinx Vivado environment, select the **Create Project** task under **Embedded System Integration**, and click **Run This Task**. A Xilinx Vivado project with IP Integrator embedded design is generated, and a link to the project is provided in the dialog window. You can optionally open up the project to take a look.

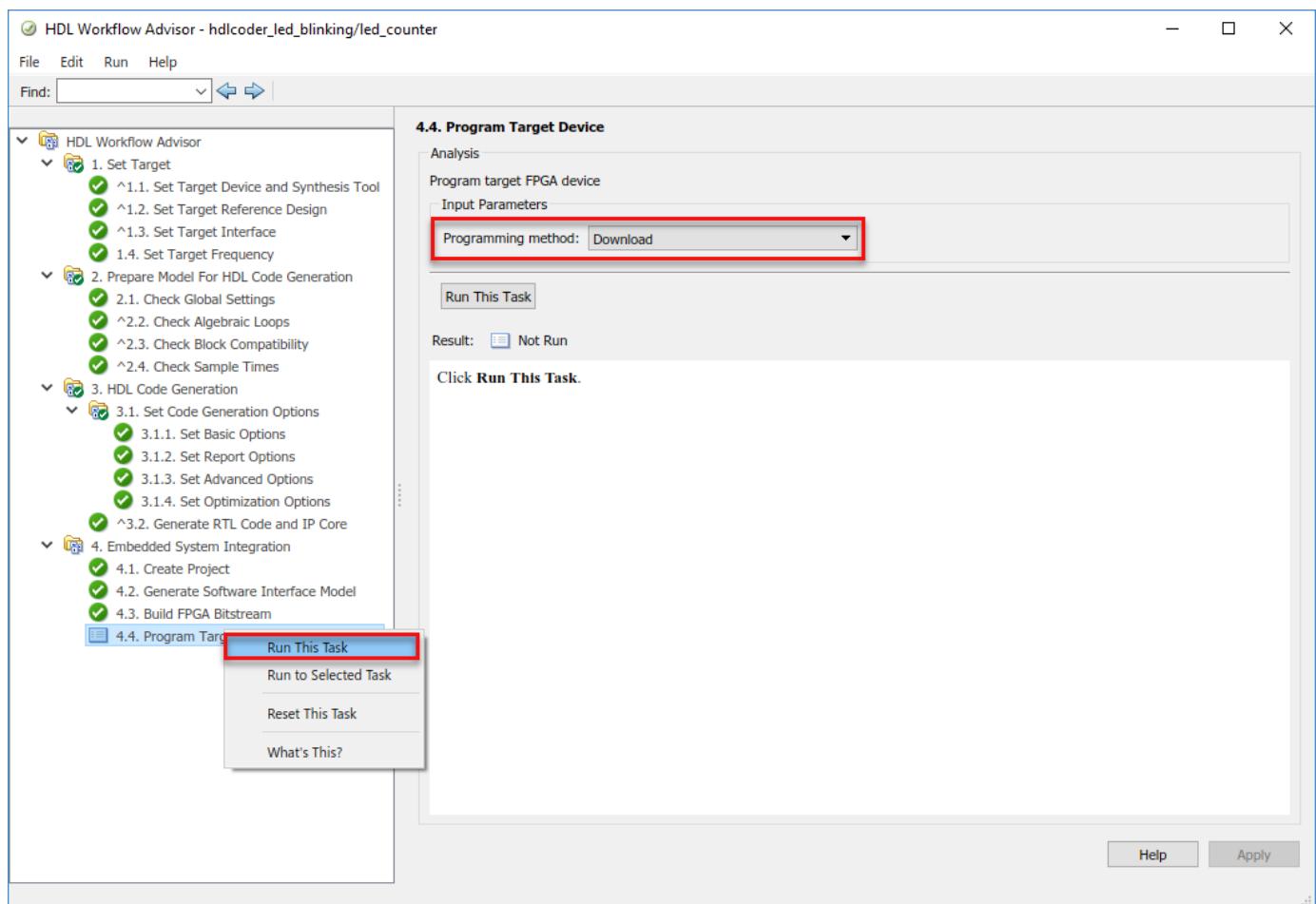


2. If you have an Embedded Coder license, you can generate a software interface model in the next task, **Generate Software Interface Model**. The details of the software interface model are explained in the next section of this example, "Generate a software interface model".

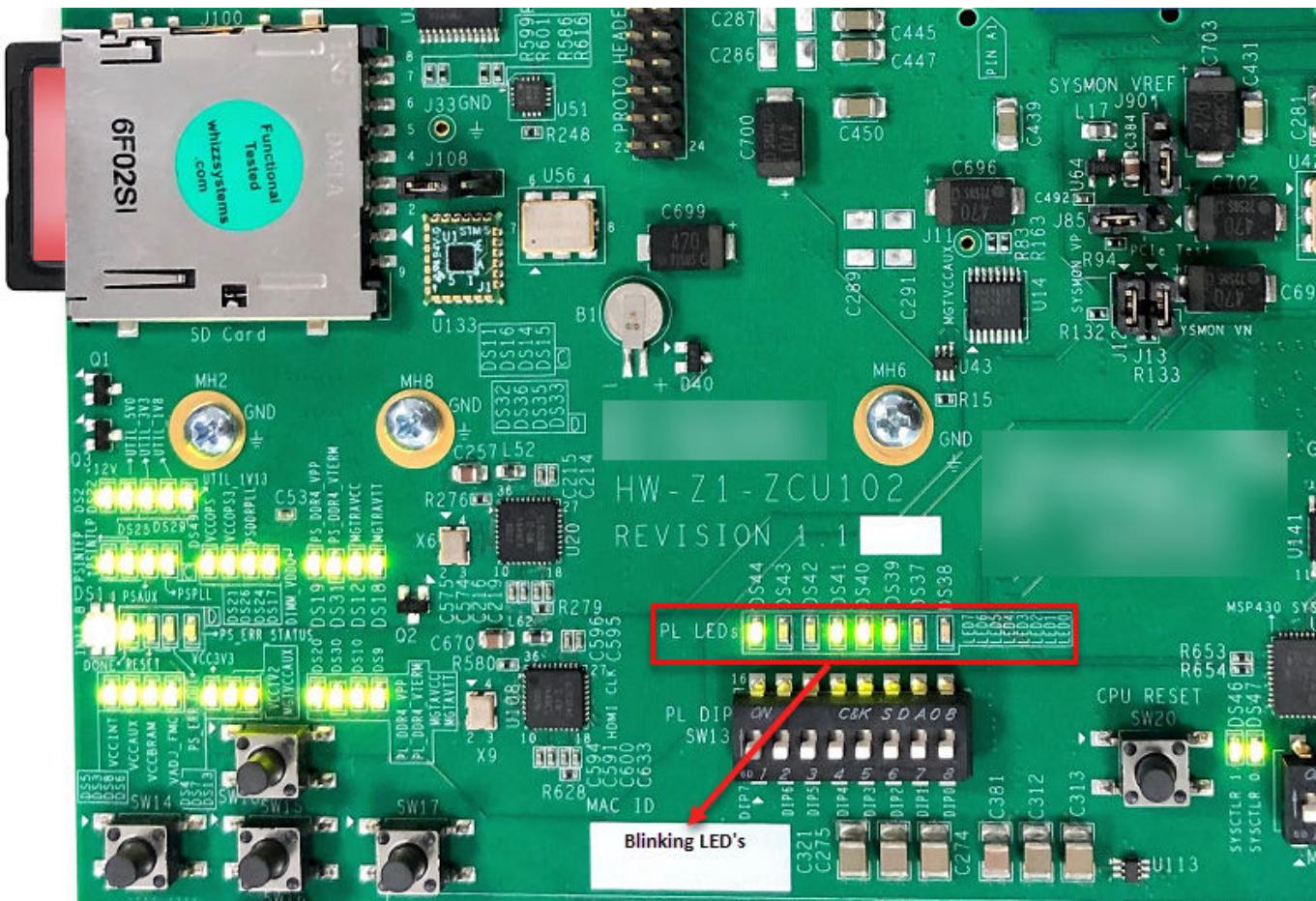
3. Build the FPGA bitstream in the **Build FPGA Bitstream** task. Make sure the **Run build process externally** option is checked, so the Xilinx synthesis tool will run in a separate process from MATLAB. Wait for the synthesis tool process to finish running in the external command window.

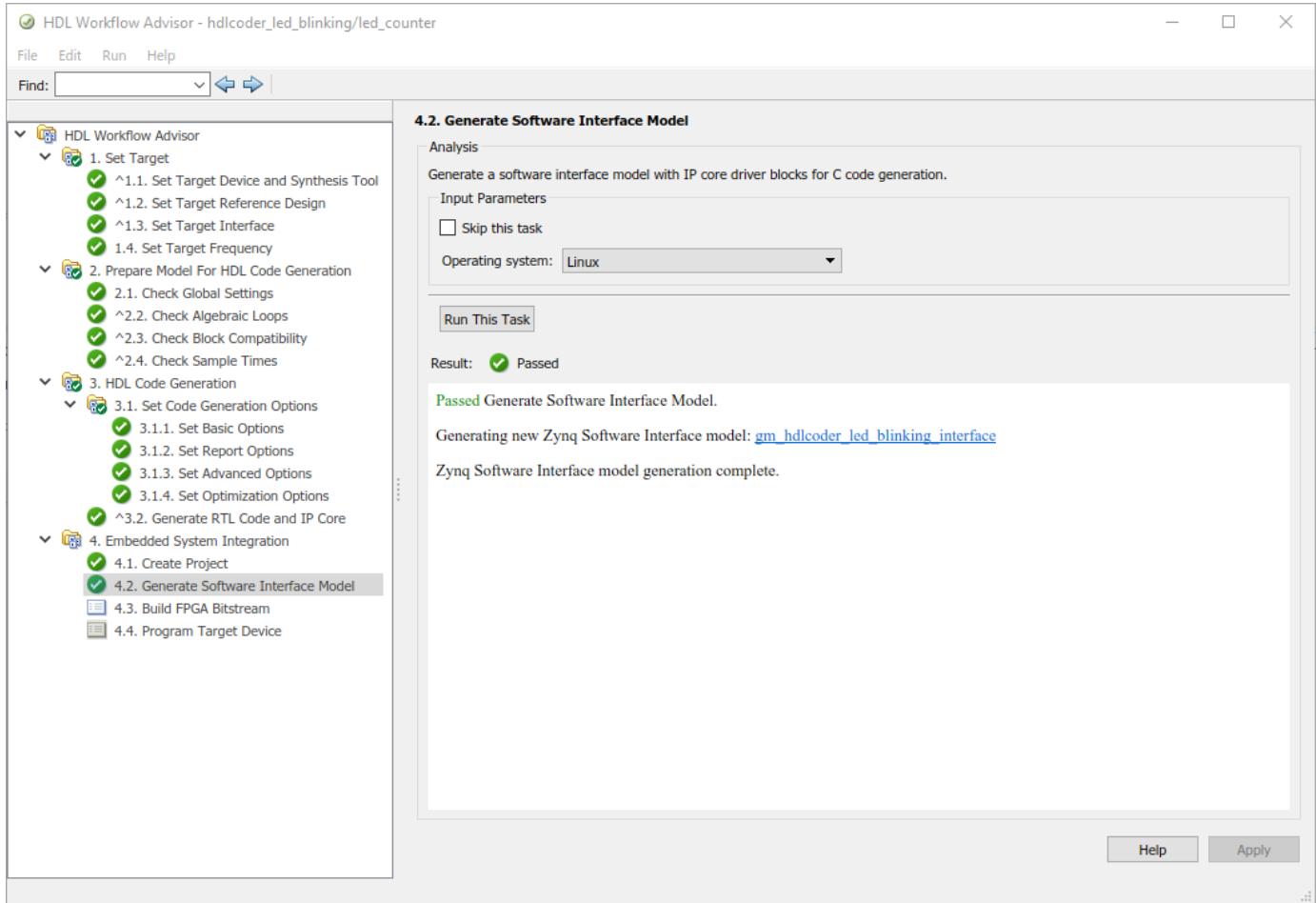


4. After the bitstream is generated, select the **Program Target Device** task. Choose **Download for Programming method** to download the FPGA bitstream onto the SD card on the Xilinx Zynq UltraScale+ MPSoC board, so your design will be automatically reloaded when you power cycle the Zynq board. click **Run This Task** to program the Zynq hardware.

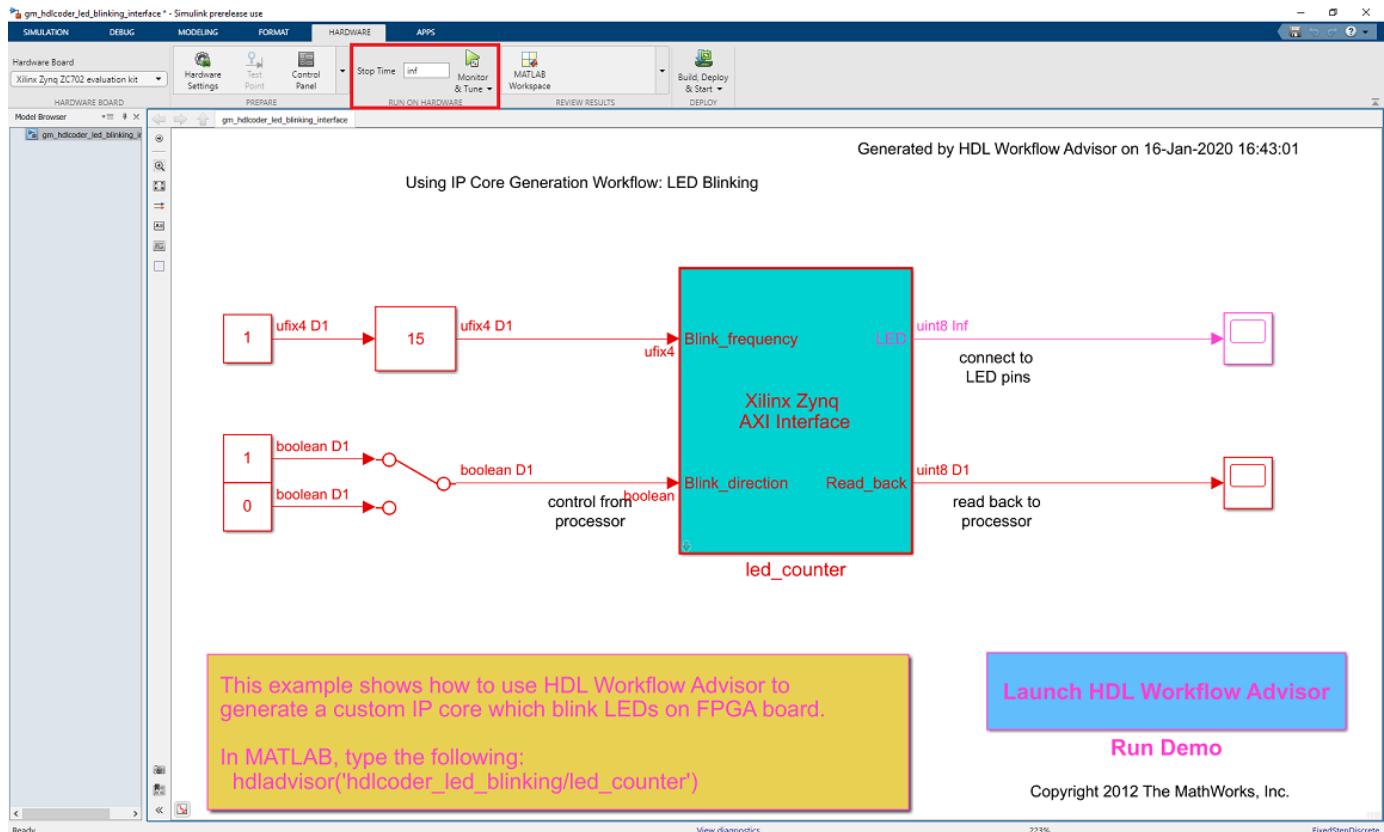


After you program the FPGA hardware, the LED starts blinking on your Xilinx Zynq UltraScale+ MPSoC ZCU102 board.





In the generated software interface model, the "led_counter" subsystem is replaced with the AXI driver blocks which generates the interface logic between the ARM processor and FPGA.

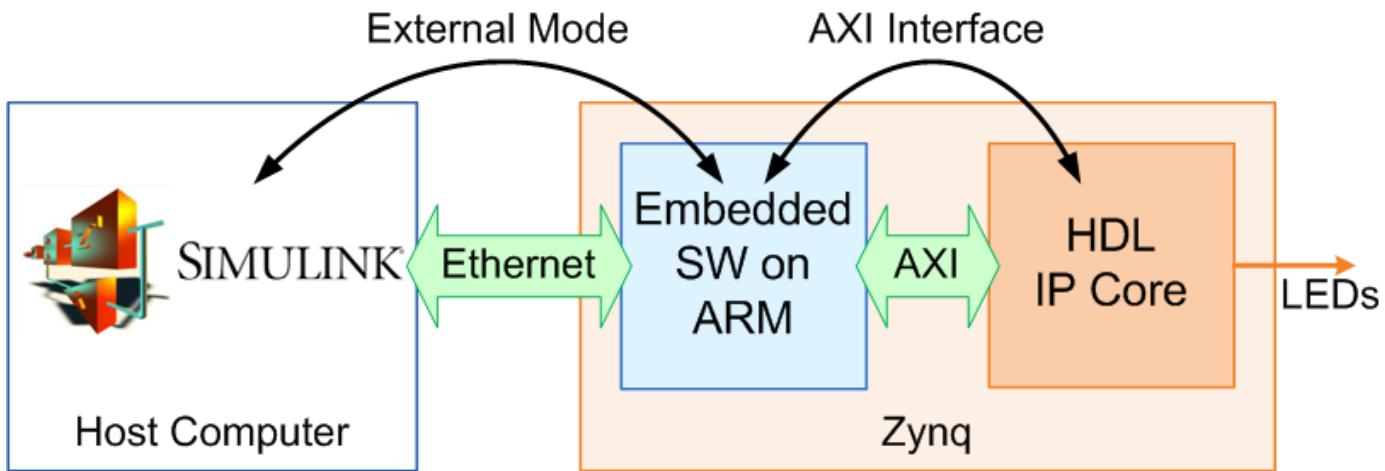


Run the software interface model on Zynq ZCU102 hardware

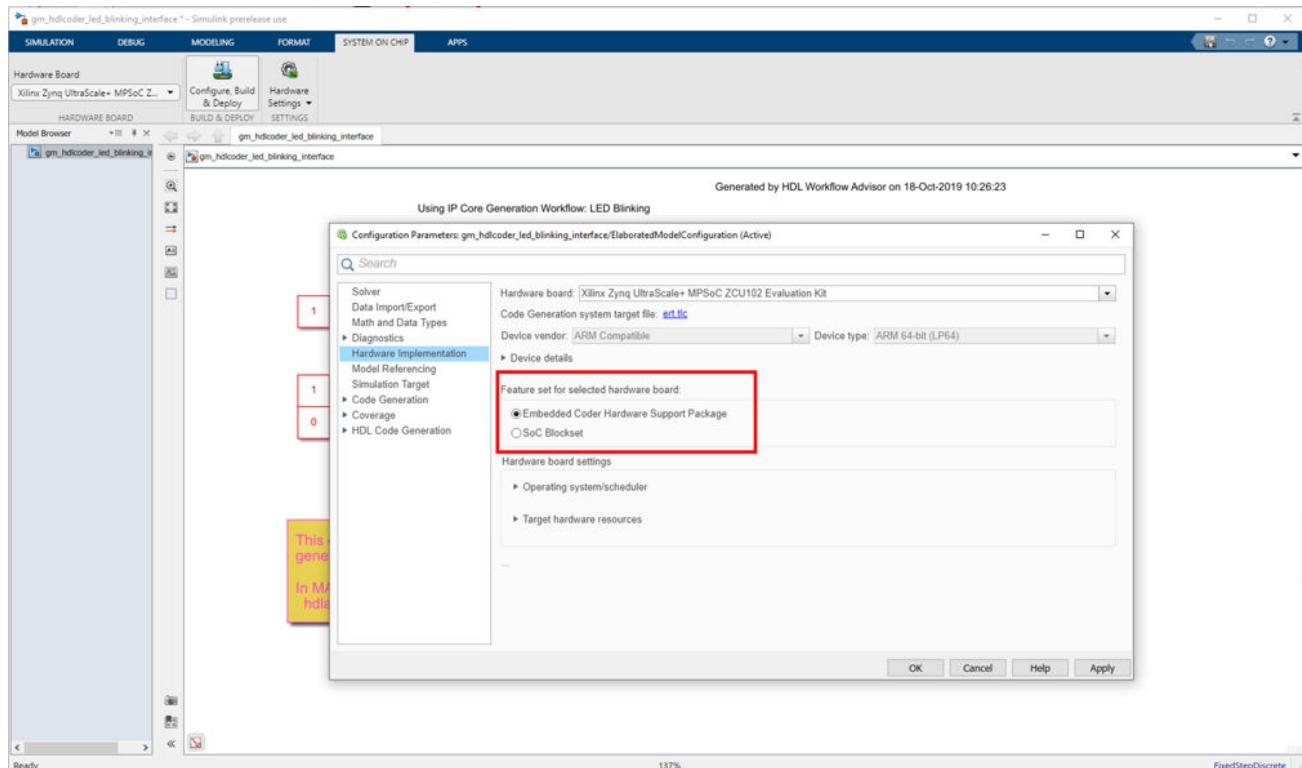
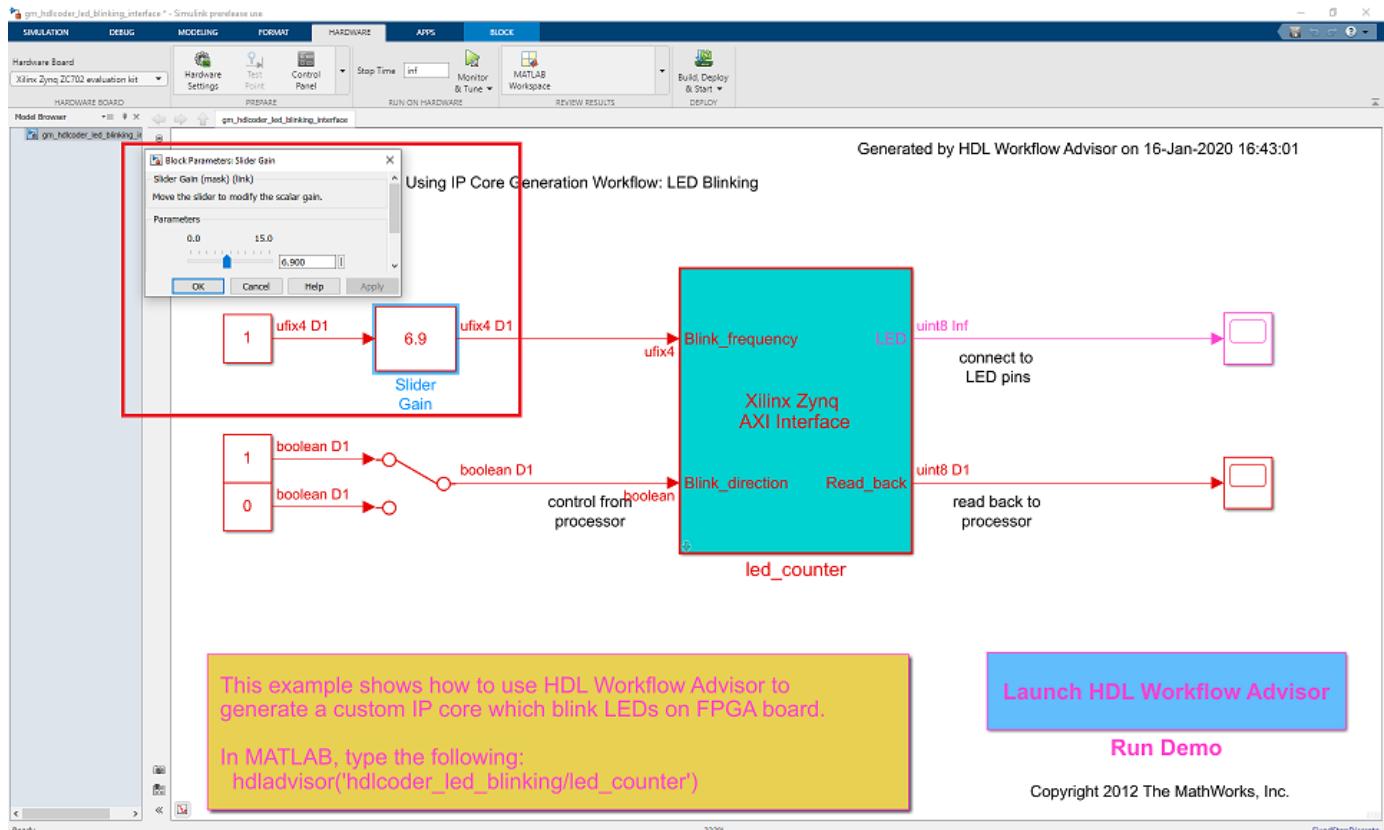
In this part of the workflow, you configure the generated software interface model, automatically generate embedded C code, and run your model on the ARM processor in the Zynq hardware in External mode.

When you are prototyping and developing an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. The External mode feature in Simulink enables this capability. In this mode, your algorithm is first deployed to the ARM processor in the Zynq hardware, and then linked with the Simulink model on the host computer through an Ethernet connection.

The main role of the Simulink model is to tune and monitor the algorithm running on the hardware. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use External mode to tune parameters, and capture data from the FPGA.



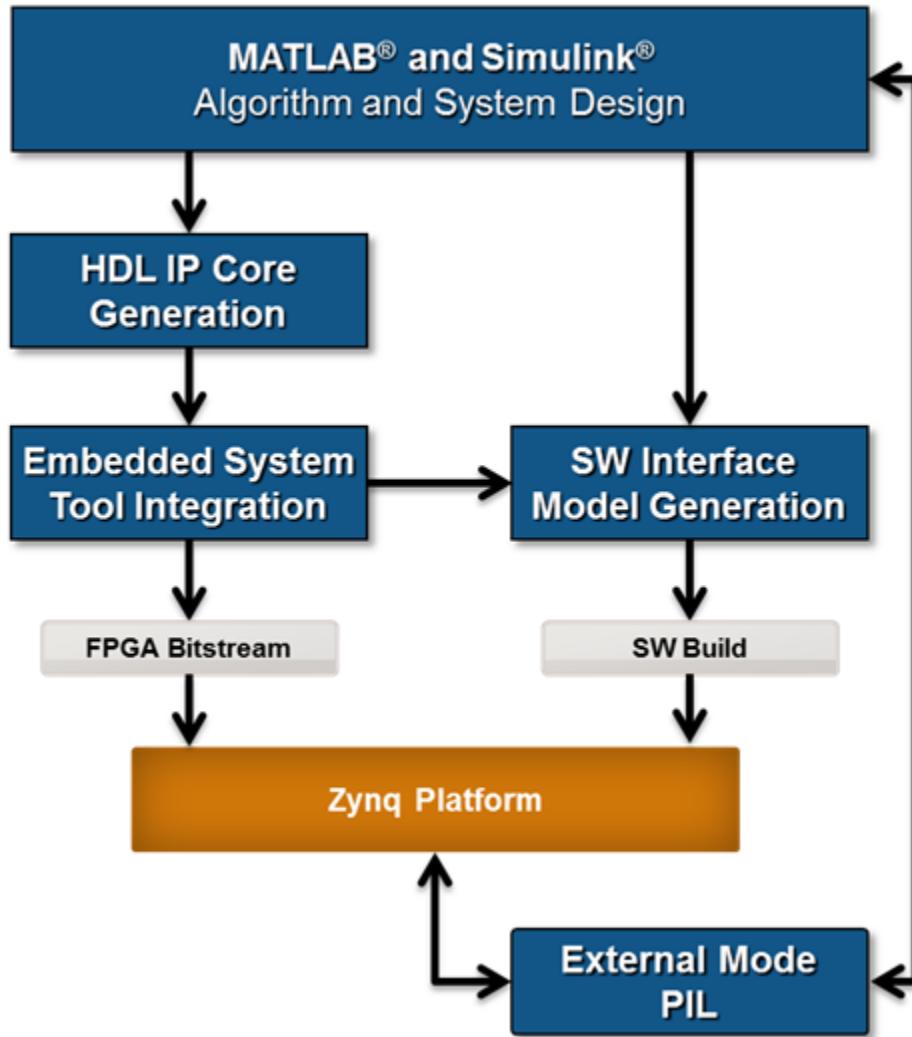
- 1 In the generated model, click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
- 2 Select **Solver** and set "Stop Time" to "inf".
- 3 Select **Hardware Implementation** and set "Feature set for selected hardware board" to "Embedded Coder Hardware Support Package".
- 4 From the **HARDWARE** menu, click the **Monitor & Tune** button on the model toolbar to run your model on the ARM processor in the Zynq UltraScale+ MPSoC ZCU102 hardware in External mode. Embedded Coder builds the model, downloads the ARM executable to the Xilinx Zynq UltraScale+ MPSoC ZCU102 hardware, executes it, and connects the model to the executable running on the Zynq hardware.
- 5 Double-click the **Slider Gain** block. Change the Slider Gain value and observe the change in frequency of the LED array blinking on the Zynq hardware. Double-click the **Manual Switch** block to switch the direction of the blinking LEDs.
- 6 Double-click the scope connected to the **Read_back** output port and observe that the output data of the FPGA IP core is captured and sent back to the Simulink scope.
- 7 When you are done changing model parameters, click the **Stop** button on the model.



Summary

This example shows how the hardware-software co-design workflow helps automate the deployment of your MATLAB and Simulink design to a Xilinx Zynq Ultrascale+ MPSoC. You can explore the best ways to partition and deploy your design by iterating through the workflow.

The following diagram shows the high-level picture of the workflow you went through in this example. To learn more about the hardware and software co-design workflow, please refer to the HDL Coder documentation.



Getting Started with Targeting Intel SoC Devices

This example shows how to use the hardware-software co-design workflow to blink LEDs at various frequencies on the Arrow® SoCKit® evaluation kit.

Introduction

This example is a step-by-step guide that helps you use the HDL Coder™ software to generate a custom HDL IP core which blinks LEDs on the Arrow SoCKit evaluation kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Altera Cyclone V SoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM Cortex-A9.

Using the guided workflow shown in this example, you automatically generate HDL code for the programmable logic using HDL Coder, generate C code for the ARM using Embedded Coder, and implement the design on the Intel SoC devices.

In this workflow, you perform the following steps:

- 1 Set up your Intel SoC hardware and tools.
- 2 Partition your design for hardware and software implementation.
- 3 Generate an HDL IP core using HDL Workflow Advisor.
- 4 Integrate the IP core into a Intel Qsys project and program the Intel SoC hardware.
- 5 Generate a software interface model.
- 6 Generate C code from the software interface model and run it on the ARM Cortex-A9 processor.
- 7 Tune parameters and capture results from the Intel SoC hardware using External Mode.

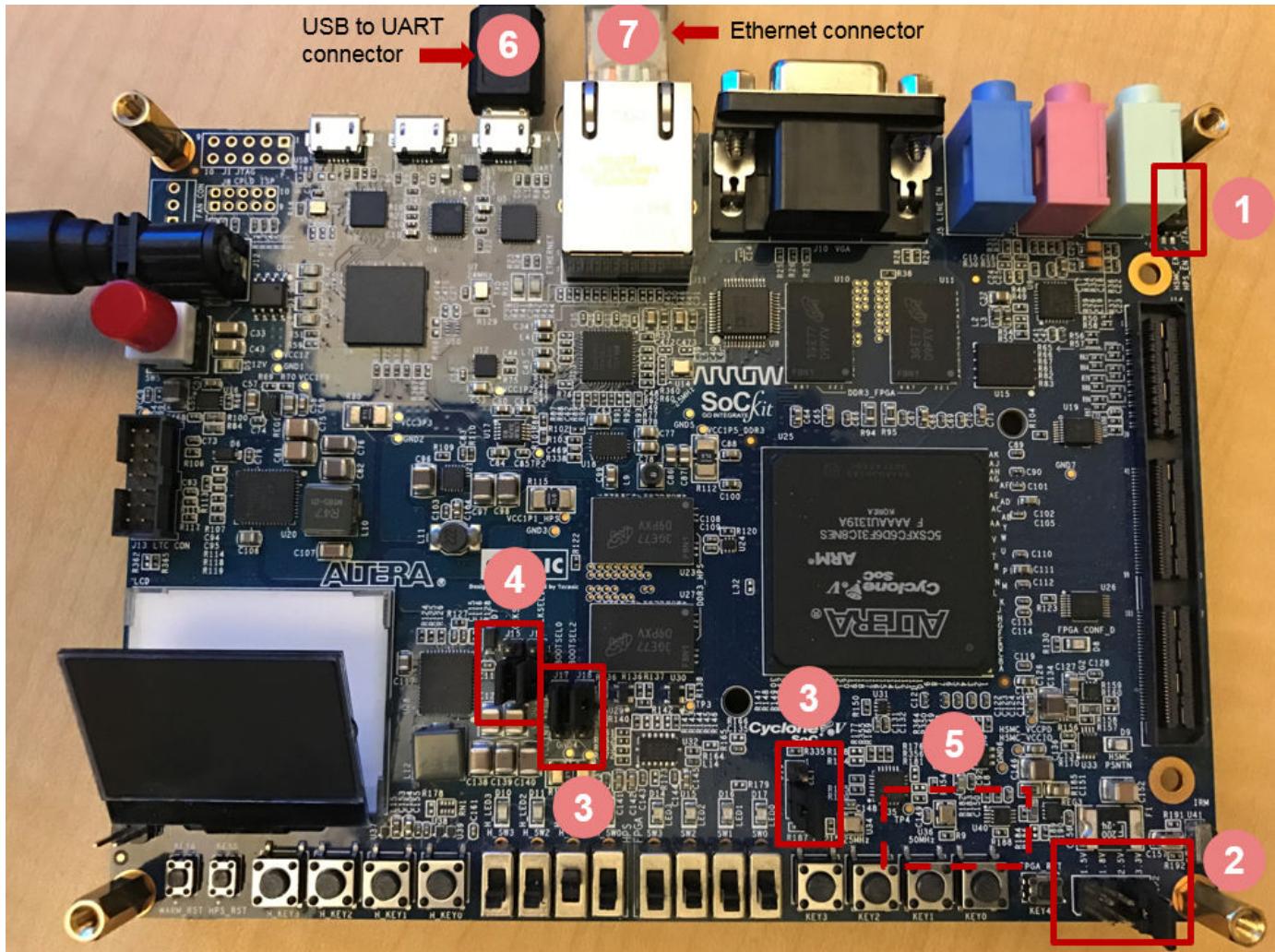
For more information, refer to other more advanced examples, and the HDL Coder and Embedded Coder documentation.

Requirements

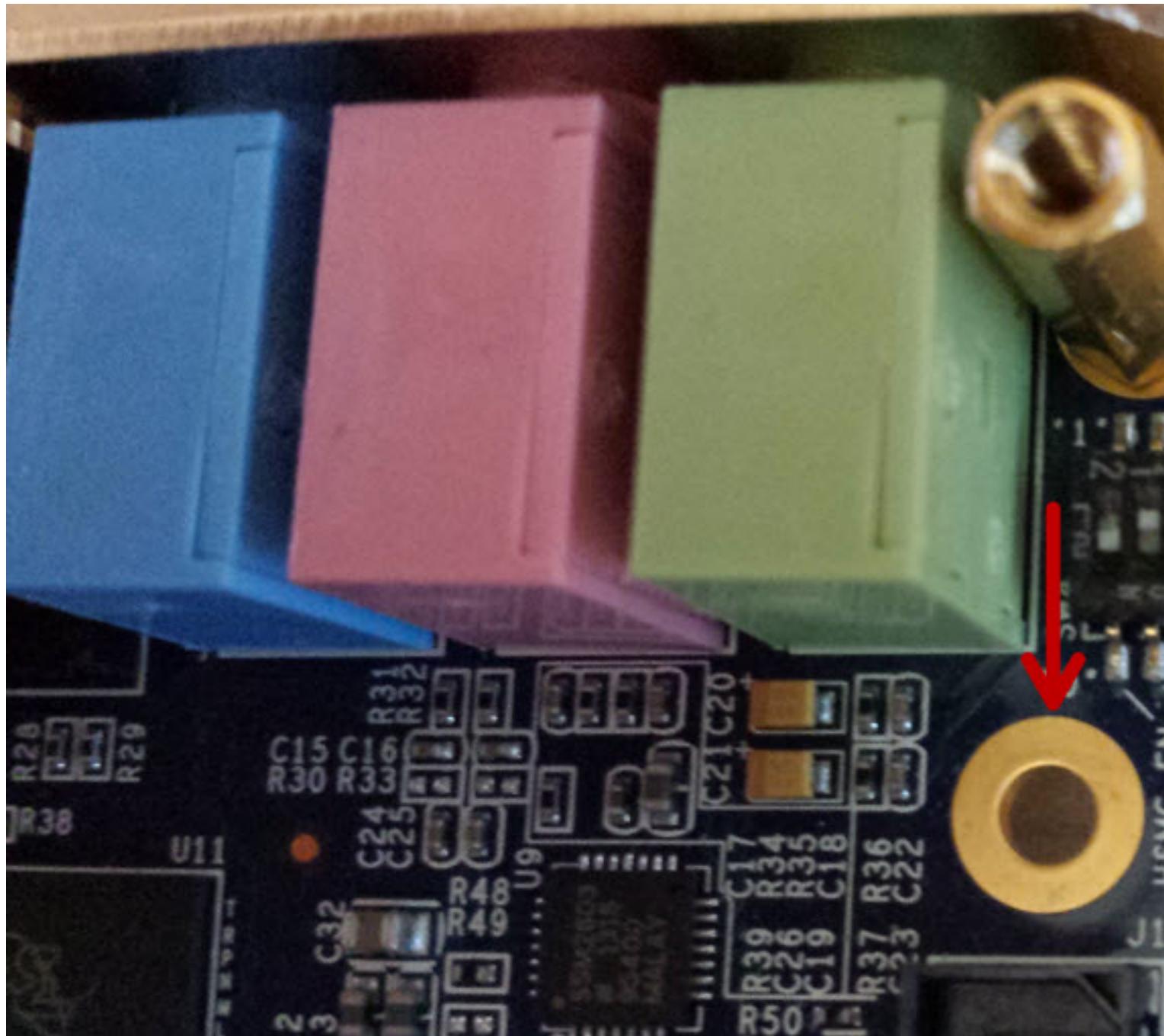
- 1 Intel Quartus Prime, with supported version listed in the HDL Coder documentation
- 2 Intel SoC Embedded Design Suite
- 3 Arrow SoCKit Cyclone V SoC evaluation kit
- 4 HDL Coder Support Package for Intel SoC Devices
- 5 Embedded Coder Support Package for Intel SoC Devices

Set up Intel SoC hardware and tools

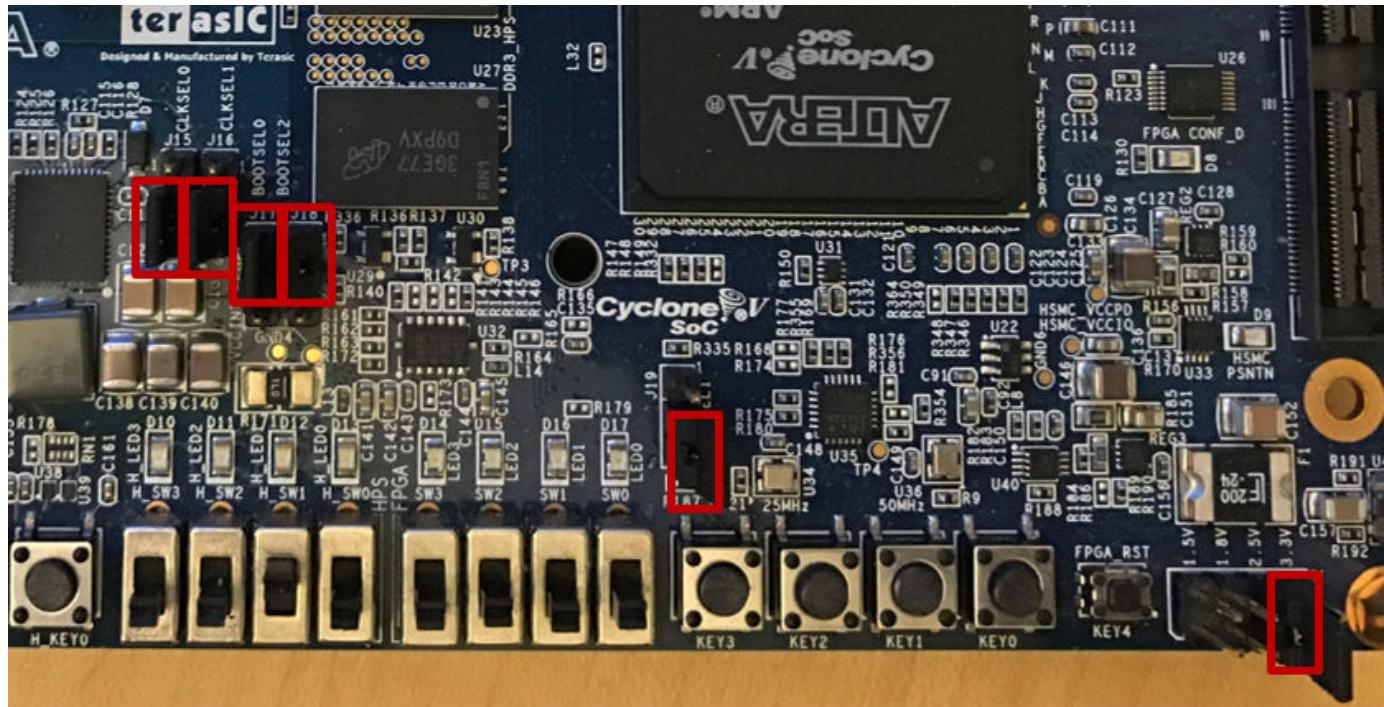
1. Set up the Arrow SoCKit evaluation kit as shown in the figure below. To learn more about the Arrow SoCKit hardware setup, please refer to the board documentation.



1.1 Set up SW4 switch (JTAG chain select) as shown in the figure below. Position 1 : OFF; Position 2 : ON. This configuration includes HPS in JTAG chain, and bypasses HSMC.



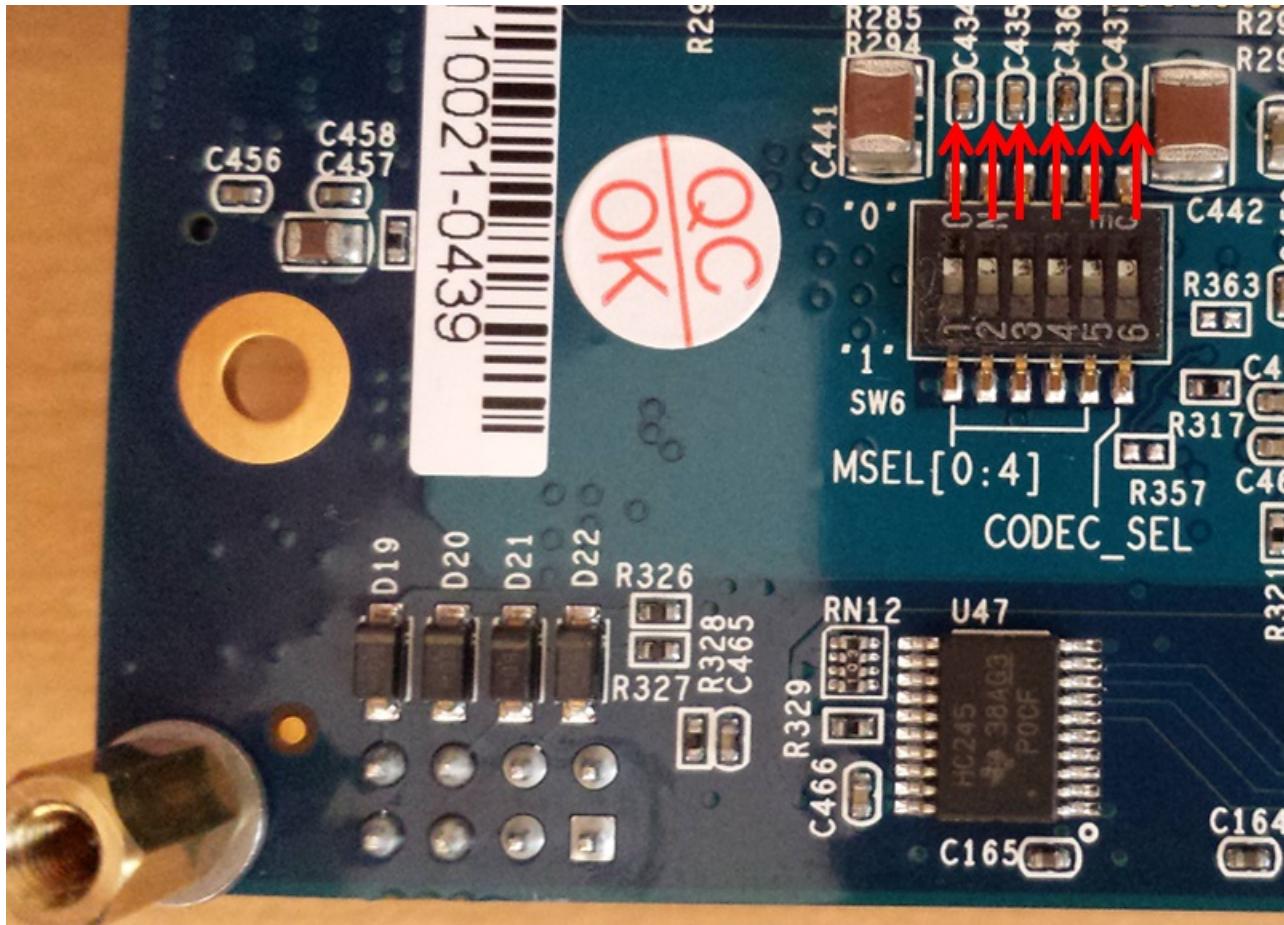
1.2 Set up JP2 as shown in the figure below to adjust the I/O Standard of the FPGA/HSMC pins. Short Pin 5 and 6 to set the I/O voltage to 2.5V.



1.3 Set up J17 - J19 as shown in the figure above to boot HPS from SD card. J17: Short Pin 1 and 2; J18: Short Pin 1 and 2; J19: Short Pin 2 and 3.

1.4 Set up J15 - J16 as shown in the figure above for HPS clock setting. J15: Short Pin 2 and 3; J16: Short Pin 2 and 3.

1.5 Set up SW6 on the back side of the board as shown in the figure below. This switch set the FPGA configuration mode. Set all 6 positions to ON.



1.6 Connect your computer to the USB UART connector using a Micro-USB cable. Make sure your USB device drivers, such as for the FTDI USB to UART, are installed correctly. If not, search for the drivers online and install them.

1.7 Connect your computer and the Arrow SoCKit board using an Ethernet cable.

2. Install the HDL Coder and Embedded Coder Support Packages for Intel SoC Devices if you haven't already. To start the installer, go to the MATLAB toolbar and click **Add-Ons > Get Hardware Support Packages**. For more information, please refer to the Support Package Installation documentation.

3. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Intel SoC Devices. If you need to update your SD card image, refer to the Hardware Setup section of this document.

4. Set up the Arrow SoCKit hardware connection by entering the following command in the MATLAB command window:

h = alterasoc

The `alterasoc` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

5. You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: 115200; Data bits: 8; Stop bits: 1; Parity: None; Flow control: None. You should be able to observe Linux booting log on the serial console when you power cycle the Arrow SoCKit board. You must close this serial connection before using the `alterasoc` function again.

6. Set up the Intel Quartus synthesis tool path using the following command in the MATLAB command window. Use your own Quartus installation path when you run the command.

```
hdlsetup('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\18.0\quartus\bin64\q')


```

Partition your design for hardware and software implementation

The first step of the Intel SoC hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

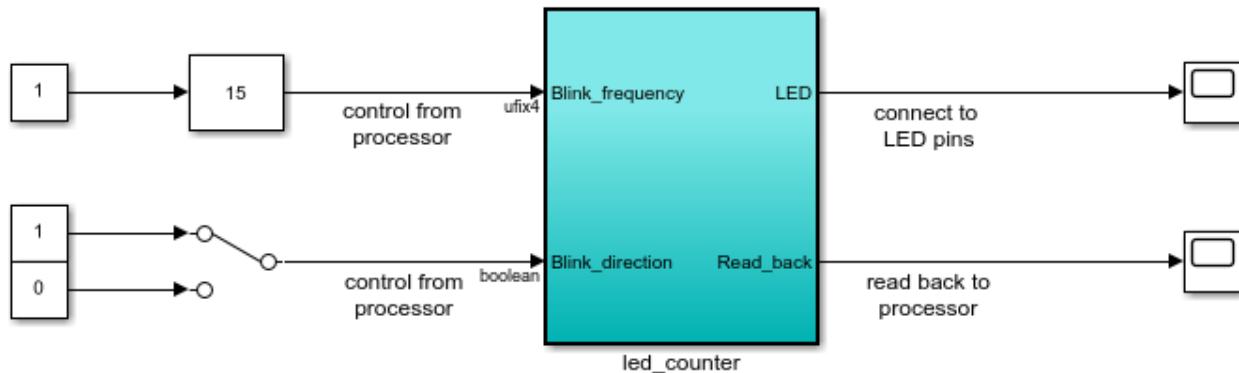
Group all the blocks you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem will be implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking_4bit');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking_4bit/led_counter')`

Launch HDL Workflow Advisor

Run Demo

Copyright 2014-2017 The MathWorks, Inc.

Generate an HDL IP core using the HDL Workflow Advisor

Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Intel Qsys environment.

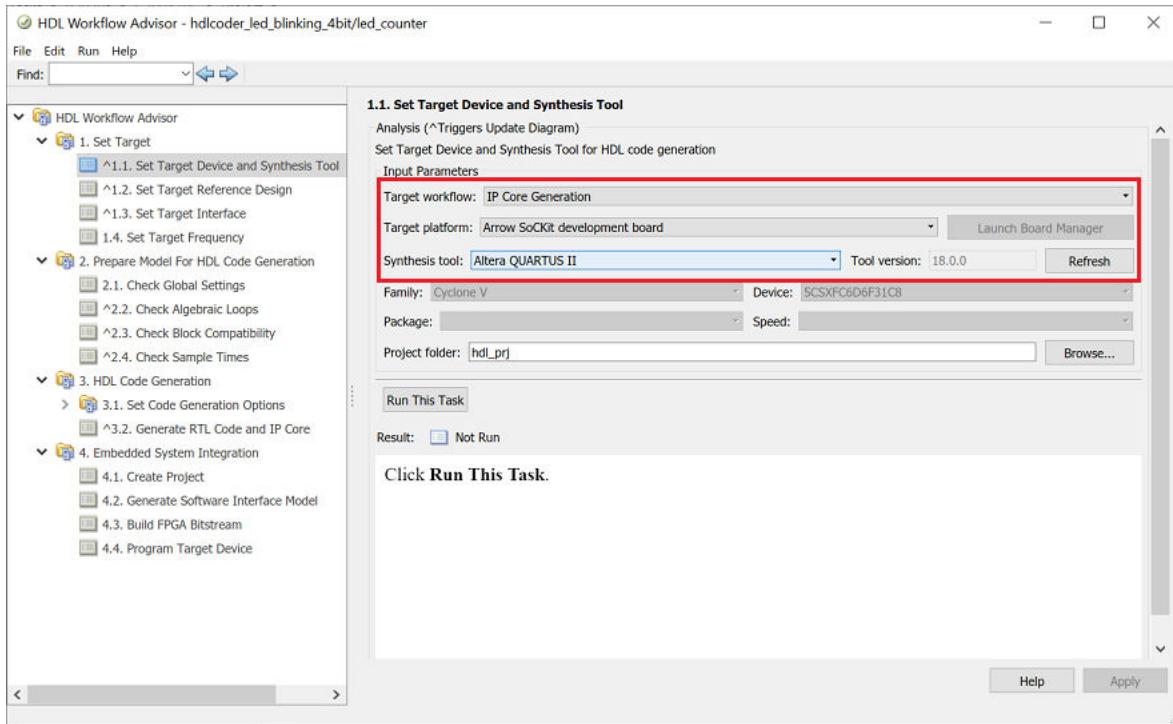
1. Start the IP core generation workflow.

1.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking_4bit/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code > HDL Workflow Advisor**.

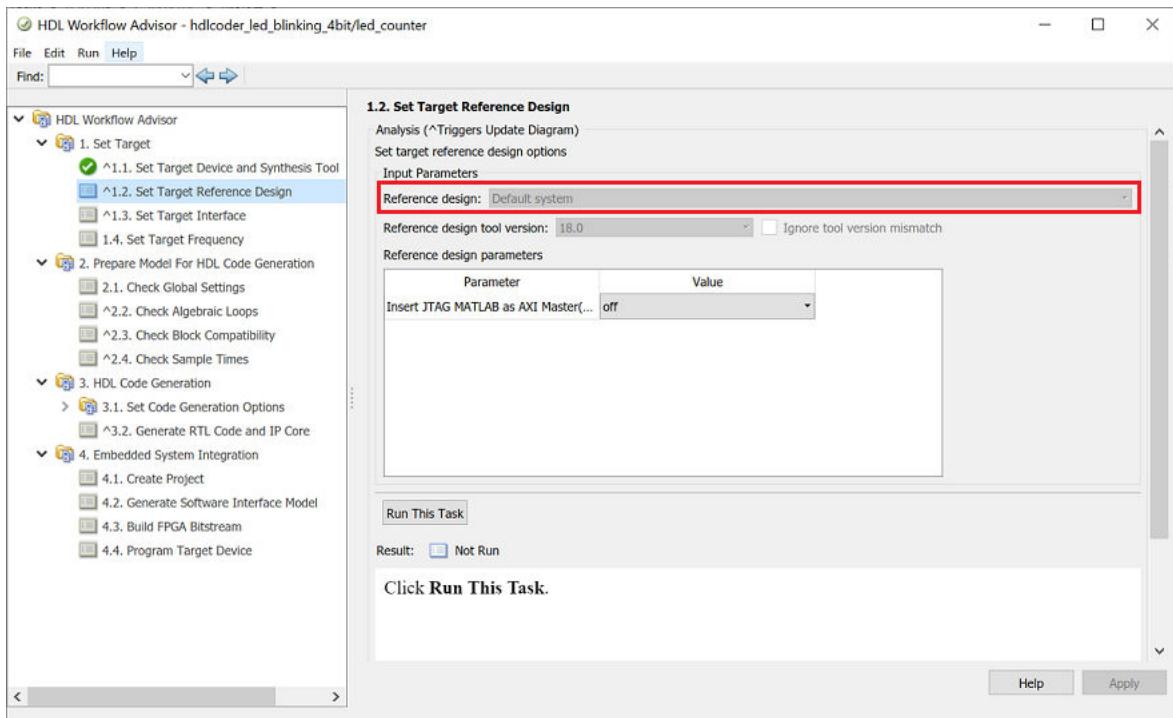
1.2. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

1.3. For **Target platform**, select **Arrow SoCKit development board**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Intel SoC Devices and follow the instructions provided by the Support Package Installer to complete the installation.

1.4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



1.5 In the **Set Target > Set Target Reference Design** task, choose **Default system**. For this example, it is selected by default



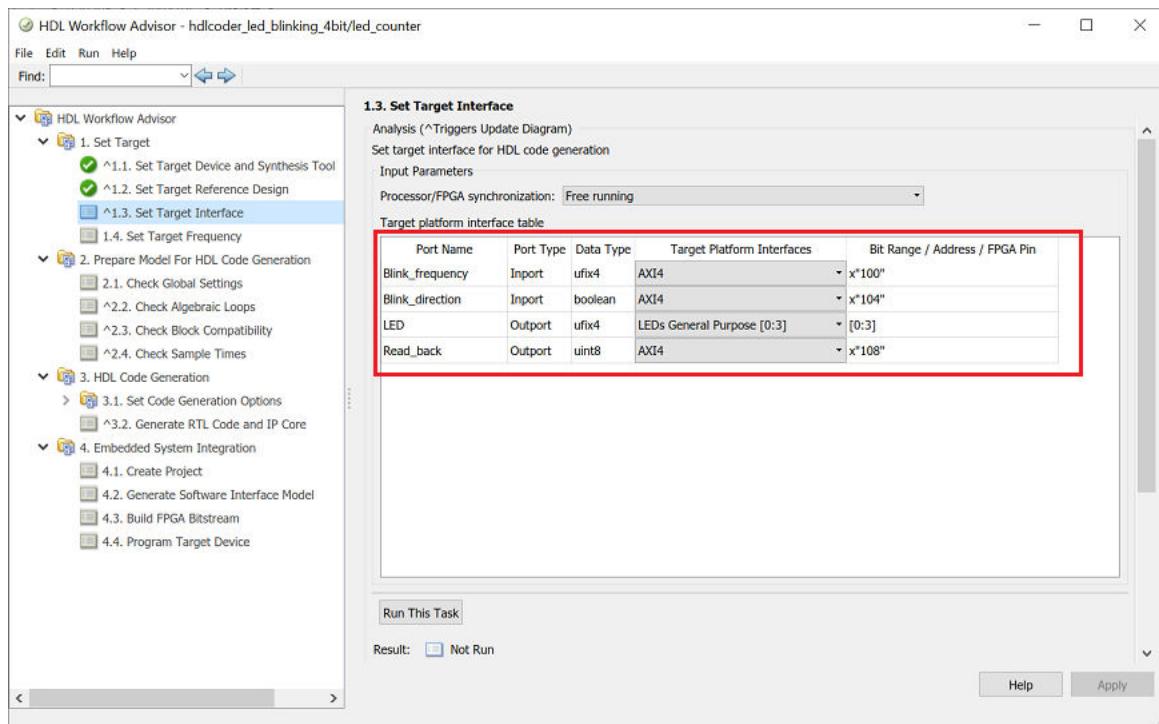
1.6. Click **Run This Task** to run the **Set Target Reference Design** task.

2. Configure the target interface.

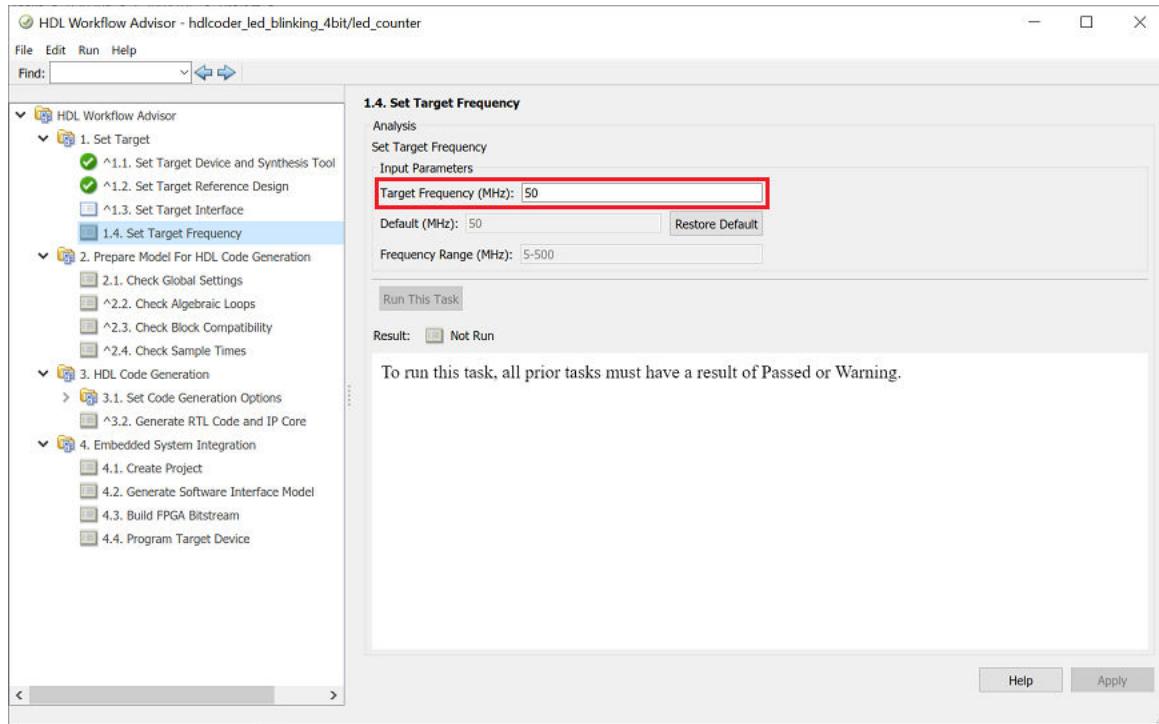
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4 interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:3]**, which connects to the LED hardware on the Intel SoC board.

2.1 In the **Set Target > Set Target Interface** task, choose **AXI4** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2.2 Choose **LEDs General Purpose [0:3]** for **LED**.

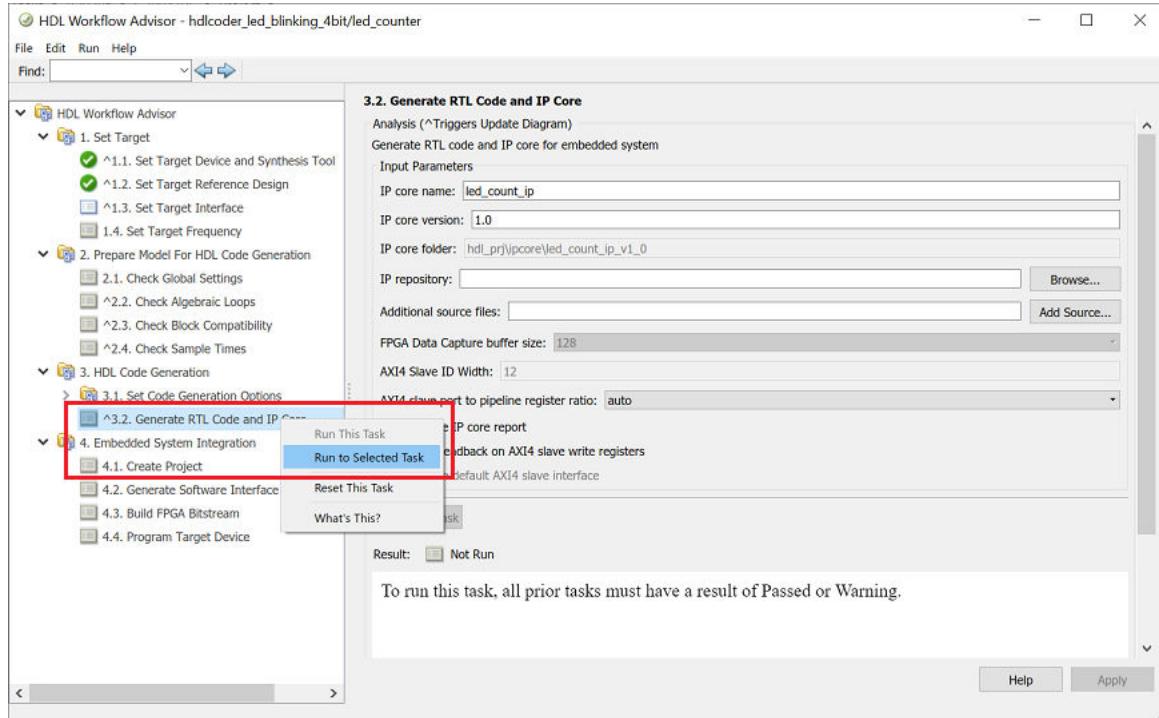


2.3 In the **Set Target > Set Target Frequency** task, choose **Target Frequency as 50 MHz**.



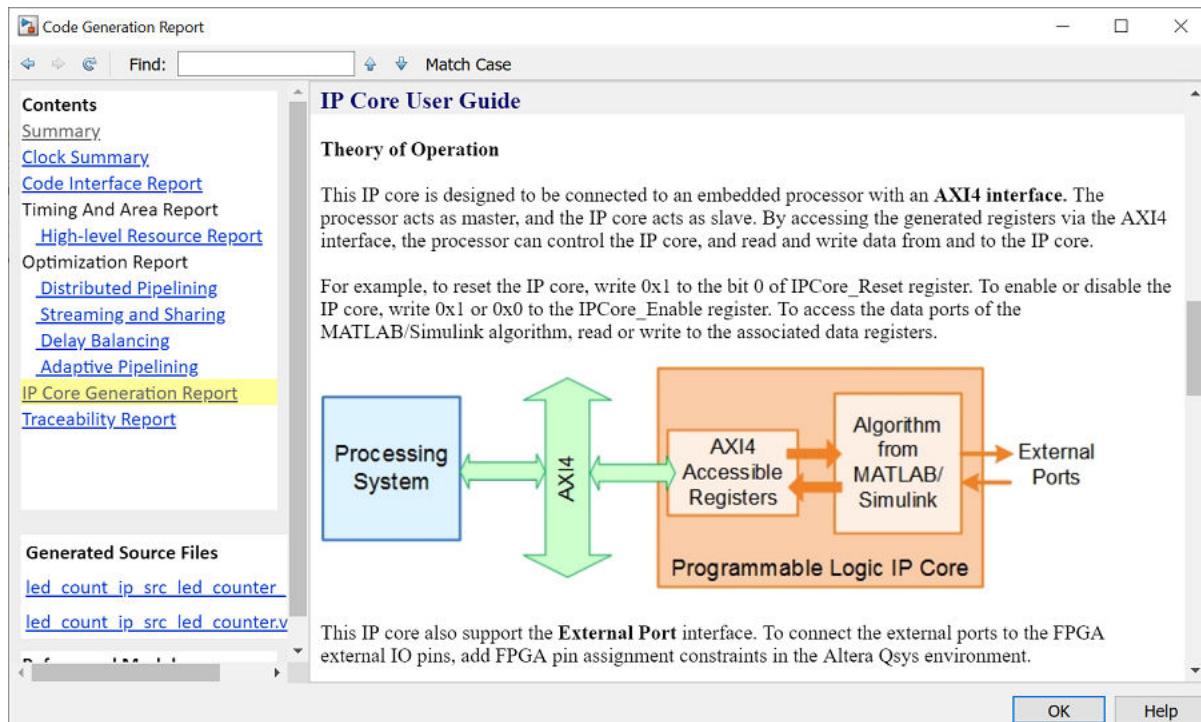
3. Generate the IP Core.

To generate the IP core, right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



4. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated together with the custom IP core. The report describes the behavior and contents of the generated custom IP core.

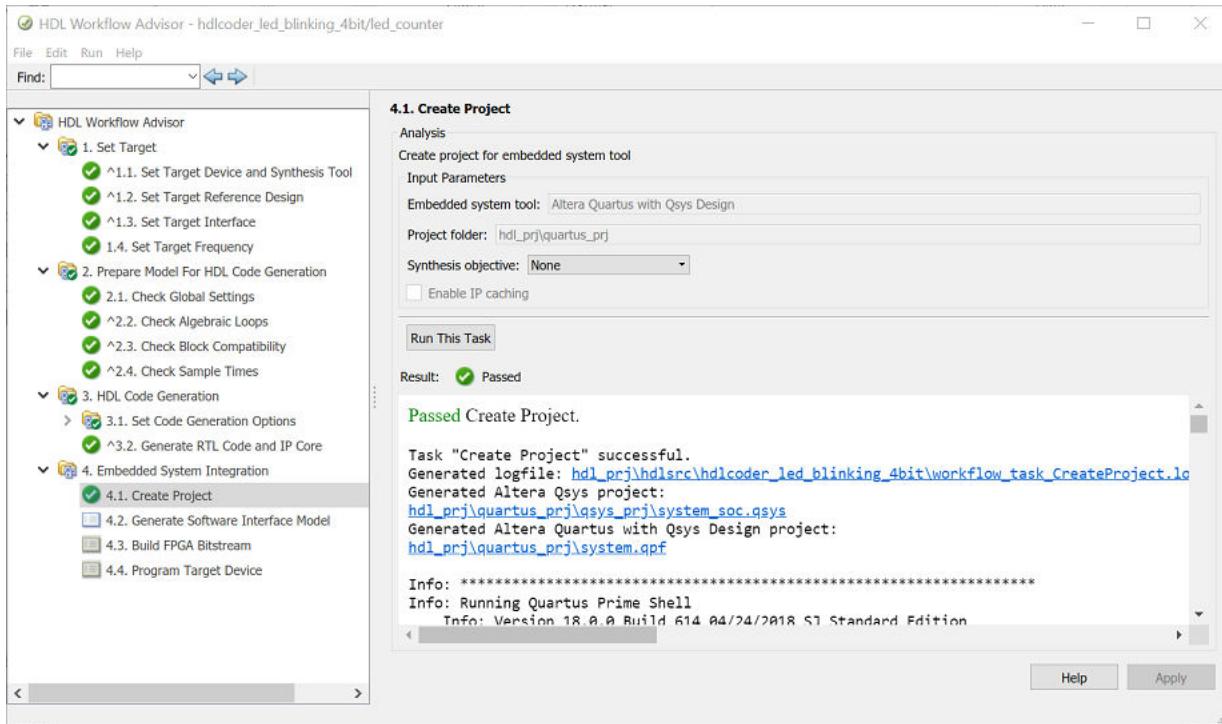


Integrate the IP core with the Intel Qsys environment

In this part of the workflow, you insert your generated IP core into a embedded system reference design, generate an FPGA bitstream, and download the bitstream to the Intel SoC hardware.

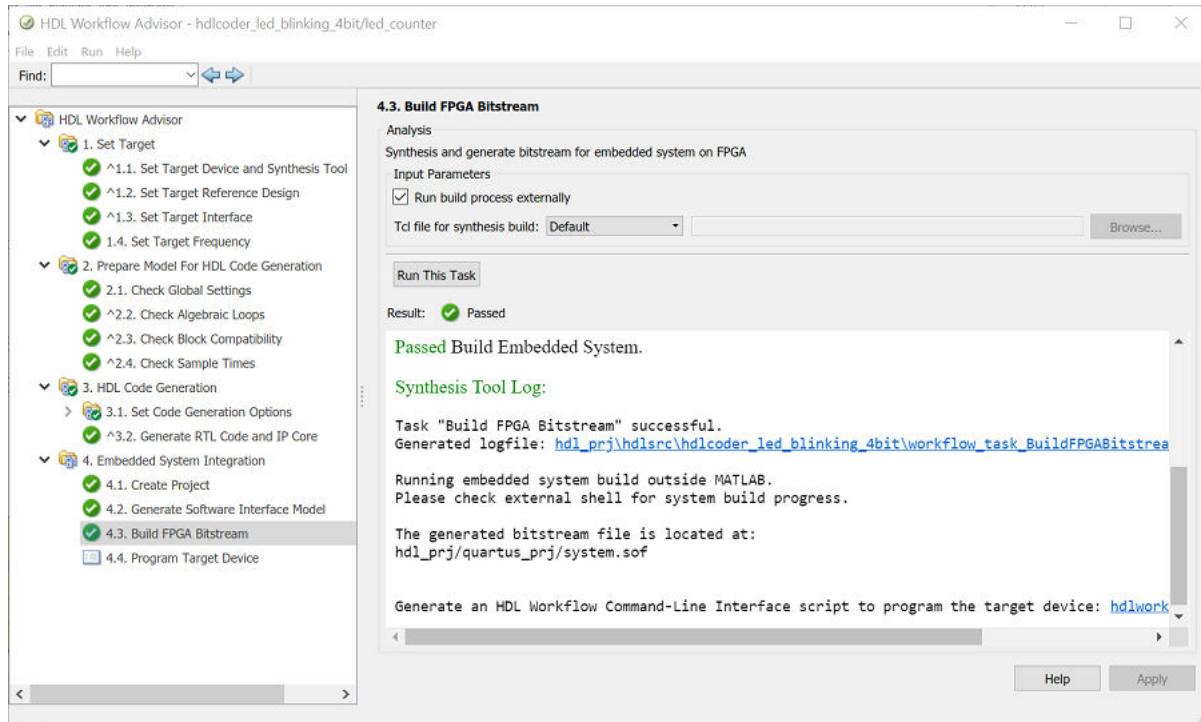
The reference design is a predefined Intel Qsys project. It contains all the elements the Intel software needs to deploy your design to the Intel SoC devices, except for the custom IP core and embedded software that you generate.

1. To integrate with the Intel Qsys environment, select the **Create Project** task under **Embedded System Integration**, and click **Run This Task**. Both an Intel Qsys project and an Intel Quartus project are generated, with links to the projects provided in the dialog window. You can optionally open up the projects to take a look.

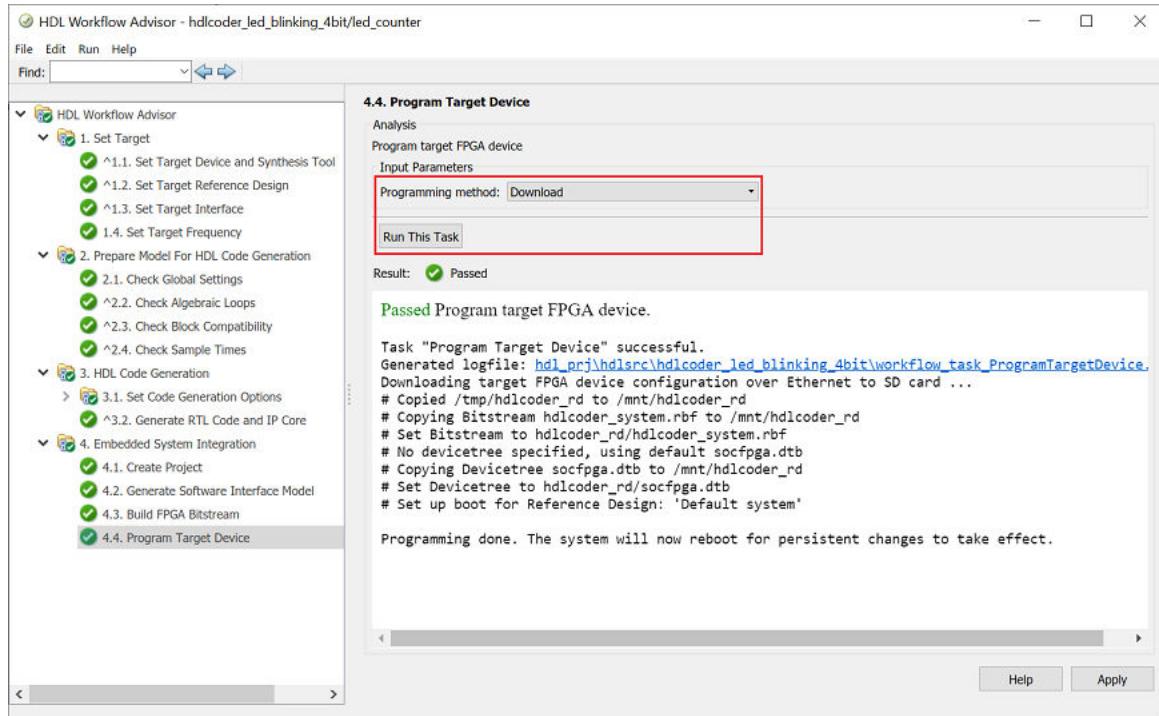


2. If you have an Embedded Coder license, you can generate a software interface model in the next task, **Generate Software Interface Model**. The details of the software interface model are explained in the next section of this example, "Generate a software interface model".

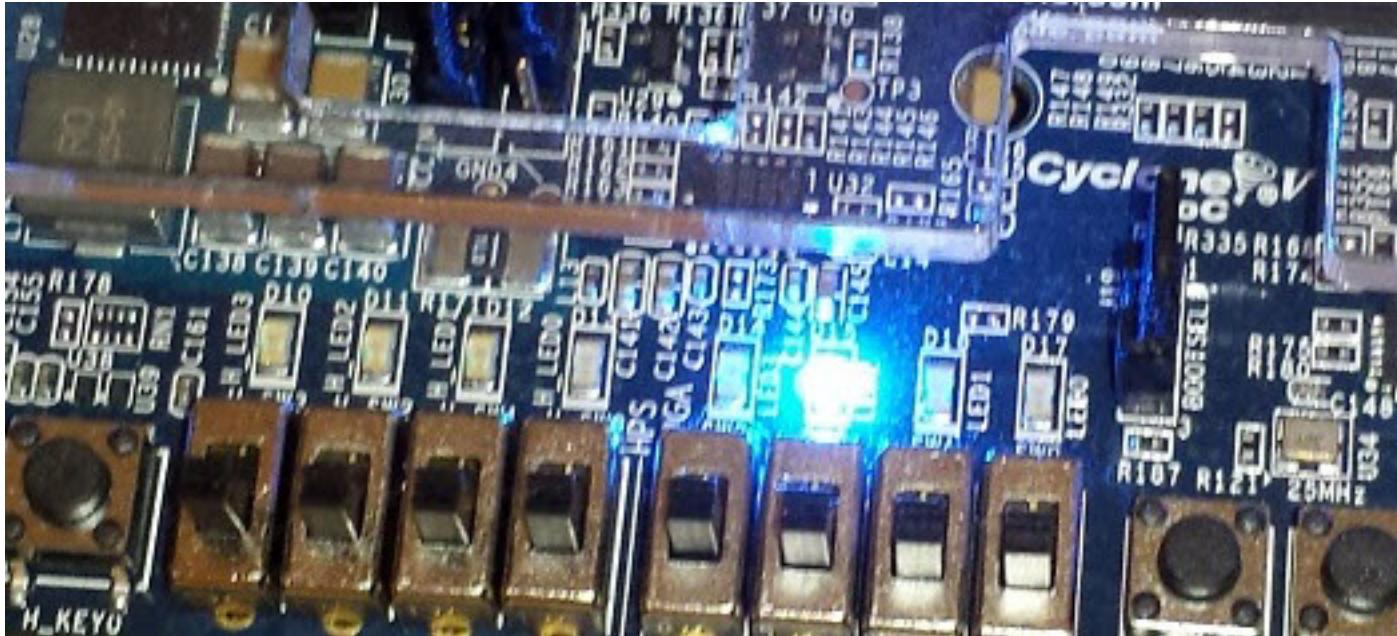
3. Build the FPGA bitstream in the **Build FPGA Bitstream** task. Make sure the **Run build process externally** option is checked, so the Intel synthesis tool will run in a separate process from MATLAB. Wait for the synthesis tool process to finish running in the external command window.



4. After the bitstream is generated, select the **Program Target Device** task. Choose **Download** for **Programming method** to download the FPGA bitstream onto the SD card on the Intel SoC board, so your design will be automatically reloaded when you power cycle the Intel SoC board. click **Run This Task** to program the Intel SoC hardware.



After you program the FPGA hardware, the LED starts blinking on your Intel SoC board.



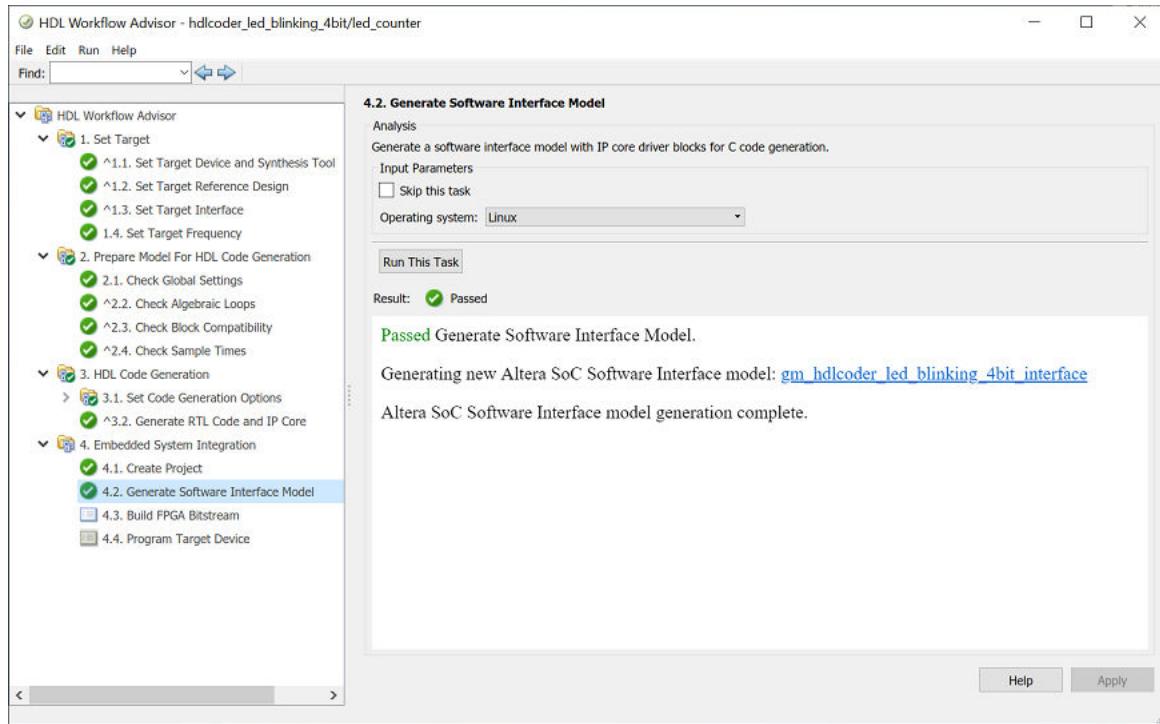
Next, you will generate C code to run on the ARM processor to control the LED blink frequency and direction.

Generate a software interface model

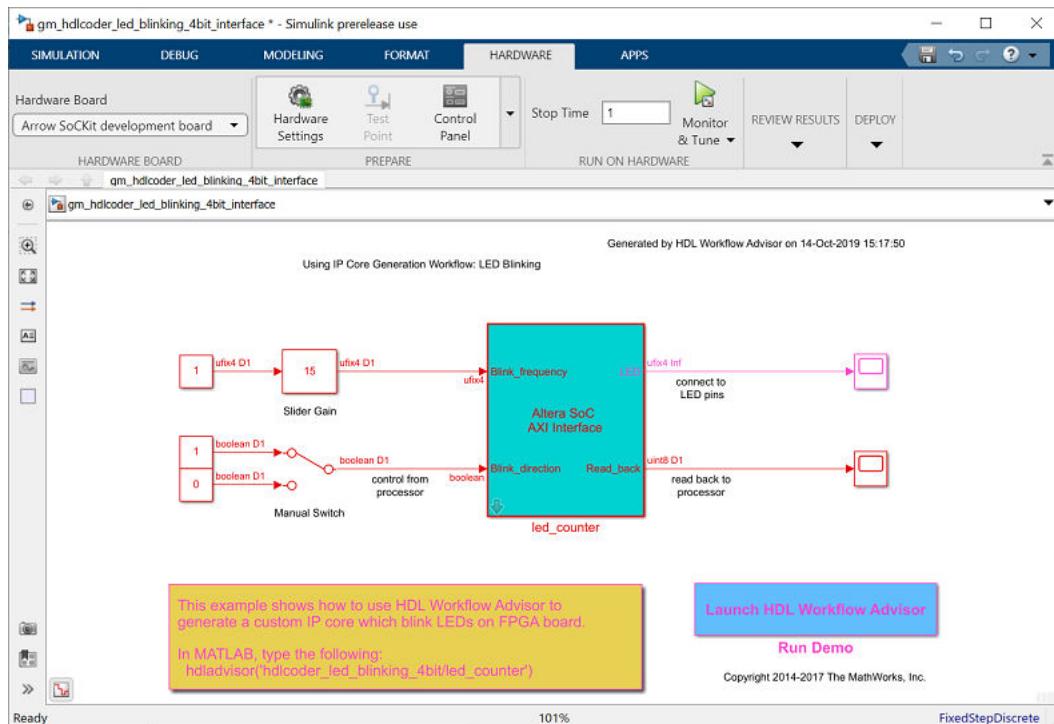
In the HDL Workflow Advisor, after you generate the IP core and insert it into the Qsys reference design, you can optionally generate a software interface model in the **Embedded System Integration > Generate Software Interface Model** task.

The software interface model contains the part of your design that runs in software. It includes all the blocks outside of the HDL subsystem, and replaces the HDL subsystem with AXI driver blocks. If you have an Embedded Coder license, you can automatically generate embedded code from the software interface model, build it, and run the executable on Linux on the ARM processor. The generated embedded software includes AXI driver code, generated from the AXI driver blocks, that controls the HDL IP core.

Run the **Generate Software Interface Model** task and see that a new model is generated. The task dialog shows a link to the model.



In the generated software interface model, the "led_counter" subsystem is replaced with the AXI driver blocks which generate the interface logic between the ARM processor and FPGA.

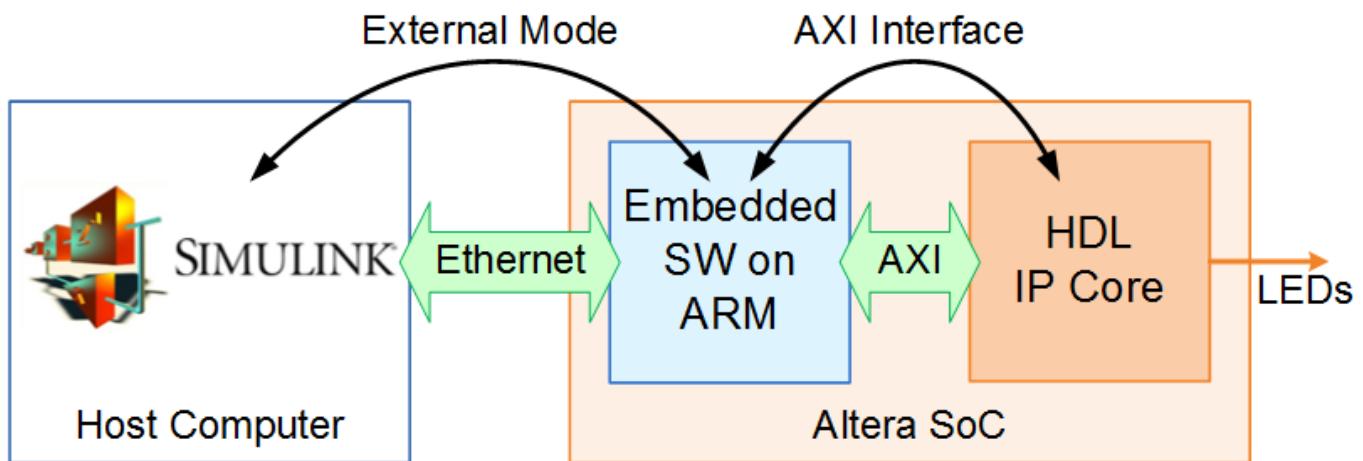


Run the software interface model on Intel SoC hardware

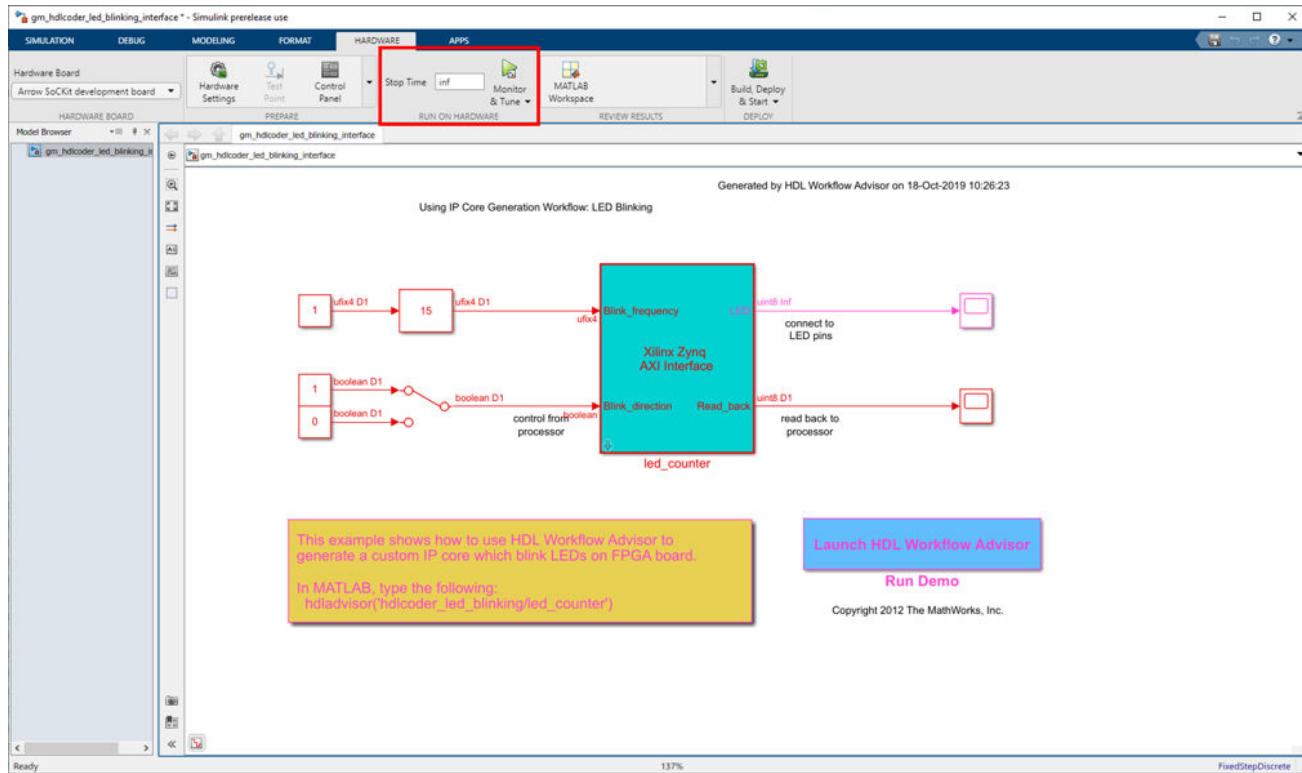
In this part of the workflow, you configure the generated software interface model, automatically generate embedded C code, and run your model on the ARM processor in the Intel SoC hardware in External mode.

When you are prototyping and developing an algorithm, it is useful to monitor and tune the algorithm while it runs on hardware. The External mode feature in Simulink enables this capability. In this mode, your algorithm is first deployed to the ARM processor in the Intel SoC hardware, and then linked with the Simulink model on the host computer through an Ethernet connection.

The main role of the Simulink model is to tune and monitor the algorithm running on the hardware. Because the ARM processor is connected to the HDL IP core through the AXI interface, you can use External mode to tune parameters, and capture data from the FPGA.



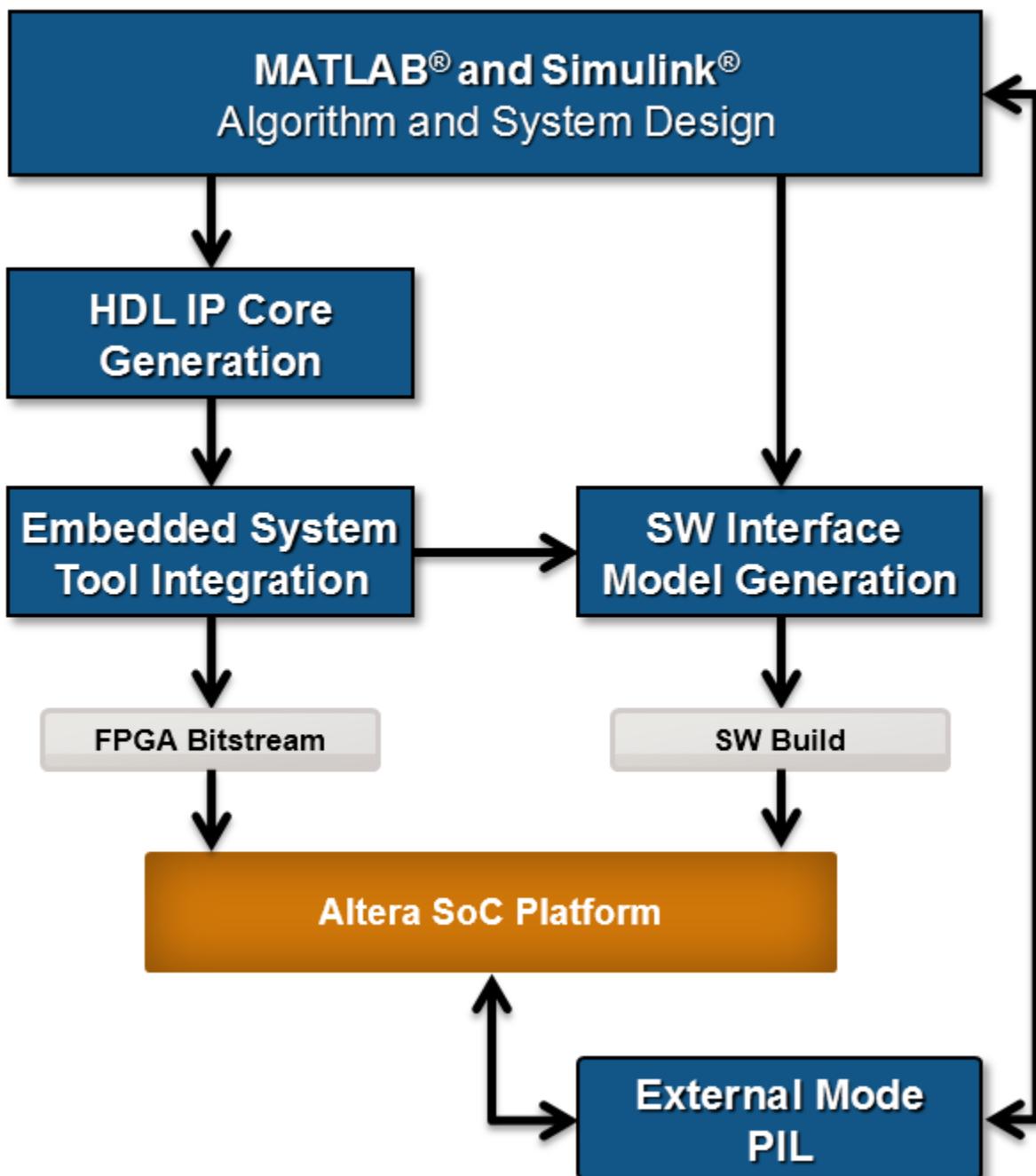
- 1 In the generated model, open the **Configuration Parameters** dialog box.
- 2 Select **Solver** and set "Stop Time" to "inf".
- 3 From the **HARDWARE** menu, click the **Monitor & Tune** button on the model toolbar to run your model on the ARM processor in the Intel SoC hardware in External mode. Embedded Coder builds the model, downloads the ARM executable to the Intel SoC hardware, executes it, and connects the model to the executable running on the Intel SoC hardware.
- 4 Double-click the **Slider Gain** block. Change the Slider Gain value and observe the change in frequency of the LED array blinking on the Intel SoC hardware. Double-click the **Manual Switch** block to switch the direction of the blinking LEDs.
- 5 Double-click the scope connected to the **Read_back** output port and observe that the output data of the FPGA IP core is captured and sent back to the Simulink scope.
- 6 When you are done changing model parameters, click the **Stop** button on the model. Observe that the system command window opened in the previous step indicates that the model has been stopped. At this point, you can close the system command window.



Summary

This example shows how the hardware and software co-design workflow helps automate the deployment of your MATLAB and Simulink design to the Intel SoC devices. You can explore the best ways to partition and deploy your design by iterating through the workflow.

The following diagram shows the high-level picture of the workflow you went through in this example. To learn more about the hardware and software co-design workflow, please refer to the HDL Coder documentation.



Getting Started with Targeting Intel Quartus Pro based Devices

This example shows how to define and register the board and reference design for the Intel Arria10 SoC development kit and use the hardware-software co-design workflow to blink LEDs at various frequencies on the Intel Arria 10 SoC development kit.

Introduction

Using this example, you can register the Arria 10 SoC development kit and the reference design in the HDL Workflow Advisor. The reference design also shows the **Early I/O (Split bitstream)** feature supported by Intel Arria 10 SoC in HDL Workflow Advisor. This example is a step-by-step guide that helps you use the HDL Coder™ software to generate a custom HDL IP core which blinks LEDs on the Intel Arria 10 SoC development kit, and shows how to use Embedded Coder® to generate C code that runs on the ARM® processor to control the LED blink frequency.

You can use MATLAB® and Simulink® to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Intel Arria 10 SoC by deciding which system elements are performed by the programmable logic, and which system elements will run on the ARM Cortex-A9.

In this workflow, you perform the following steps:

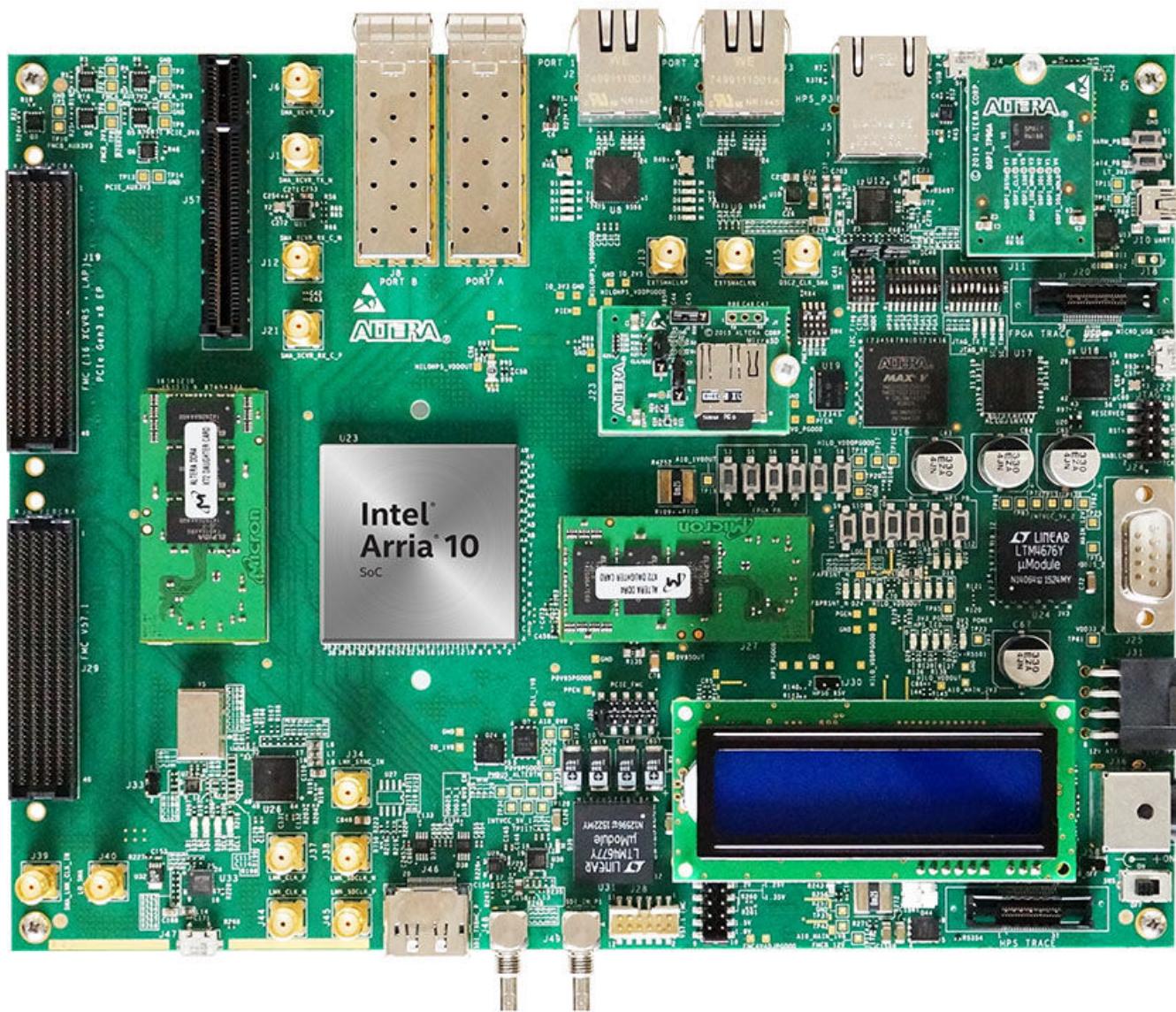
- 1 Set up your Intel SoC hardware and tools.
- 2 Create reference design for Intel Arria 10 SoC which uses the Early I/O feature.
- 3 Partition your design for hardware and software implementation.
- 4 Generate an HDL IP core using HDL Workflow Advisor.
- 5 Integrate the IP core into Intel Platform Designer Qsys project and program the Intel SoC hardware.
- 6 Generate a software interface model.
- 7 Generate C code from the software interface model and run it on the ARM Cortex-A9 processor.
- 8 Tune parameters and capture results from the Intel SoC hardware using External Mode.

Requirements

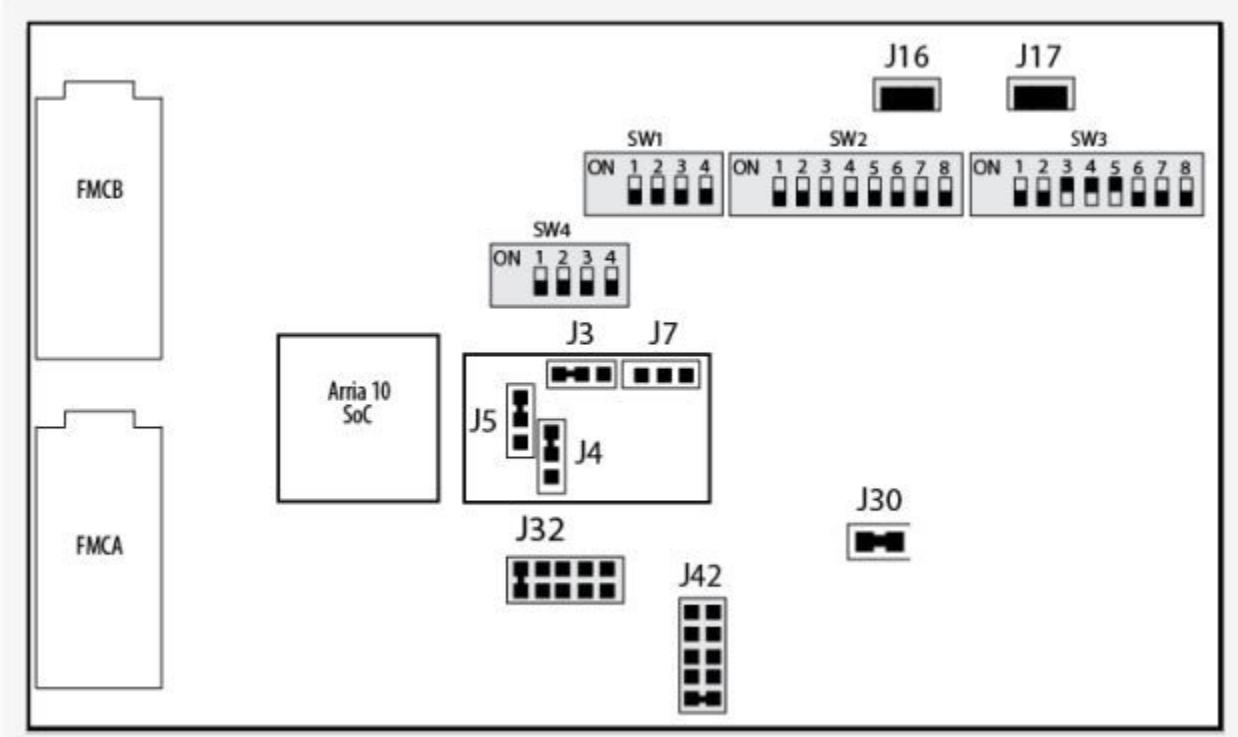
- 1 Intel Quartus Pro (or Intel QUARTUS II), with supported version listed in the HDL Coder documentation
- 2 Intel SoC Embedded Design Suite
- 3 Intel Arria 10 SoC development kit
- 4 HDL Coder Support Package for Intel SoC Devices
- 5 Embedded Coder Support Package for Intel SoC Devices

Set up Intel SoC hardware and tools

1. Set up the Arria 10 SoC as shown in the figure below. To learn more about the Arria 10 SoC hardware setup, please refer to the board documentation.



1.1 Set up DIP switches and Jumper settings as shown in the figure below.



1.2 Connect the Arria10 SoC Kit's USB UART using a Micro-USB cable to your computer. Make sure your USB device drivers, such as for the FTDI USB to UART, are installed correctly. If not, search for the drivers online and install them.

1.3 Connect the Arria10 SoC Kit to your computer using an Ethernet cable.

2. Install the HDL Coder and Embedded Coder Support Packages for Intel SoC Devices if you haven't already. To start the installer, go to the MATLAB toolbar and click **Add-Ons > Get Hardware Support Packages**. For more information, please refer to the Support Package Installation documentation.

3. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Intel SoC Devices. If you need to update your SD card image, refer to the Hardware Setup section of this document.

4. Set up the Arria10 SoC hardware connection by entering the following command in the MATLAB command window:

```
h = alterasoc
```

The `alterasoc` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

5. You can optionally test the serial connection using the following configuration using a program such as PuTTY™. Baud rate: 115200; Data bits: 8; Stop bits: 1; Parity: None; Flow control: None. You should be able to observe Linux booting log on the serial console when you power cycle the Arria10 SoC Kit board. You must close this serial connection before using the `alterasoc` function again.

- 6.** Set up the Intel Quartus Pro synthesis tool path using the following command in the MATLAB command window. Use your own Quartus installation path when you run the command.

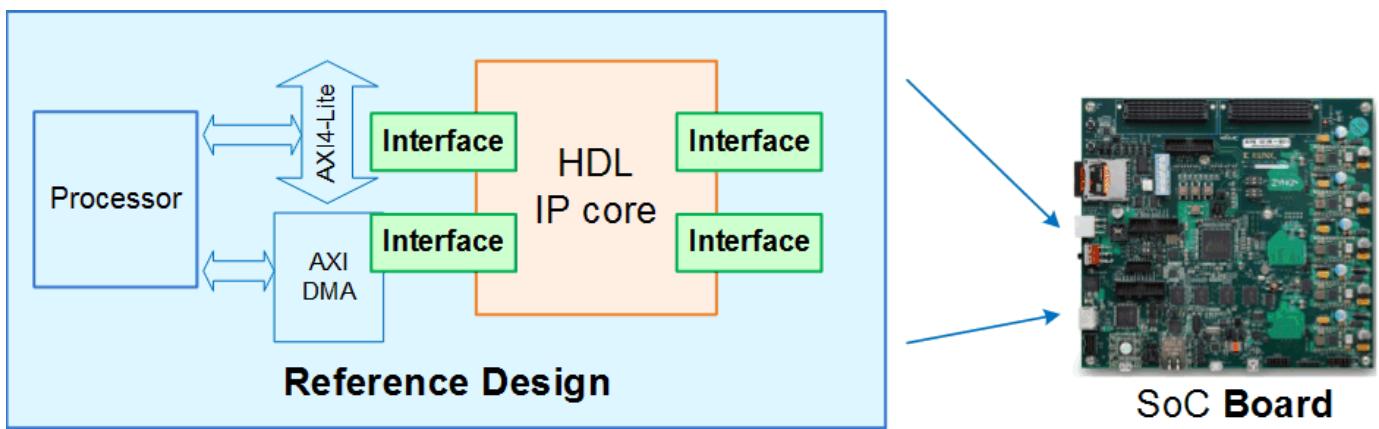
```
hdlsetupoolpath('ToolName', 'Intel Quartus Pro', 'ToolPath', 'C:\intelFPGA\19.4\quartus\bin64\quartus')
```

If you are using Intel QUARTUS II, Use the follow command:

```
hdlsetupoolpath('ToolName', 'Intel QUARTUS II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\quartus')
```

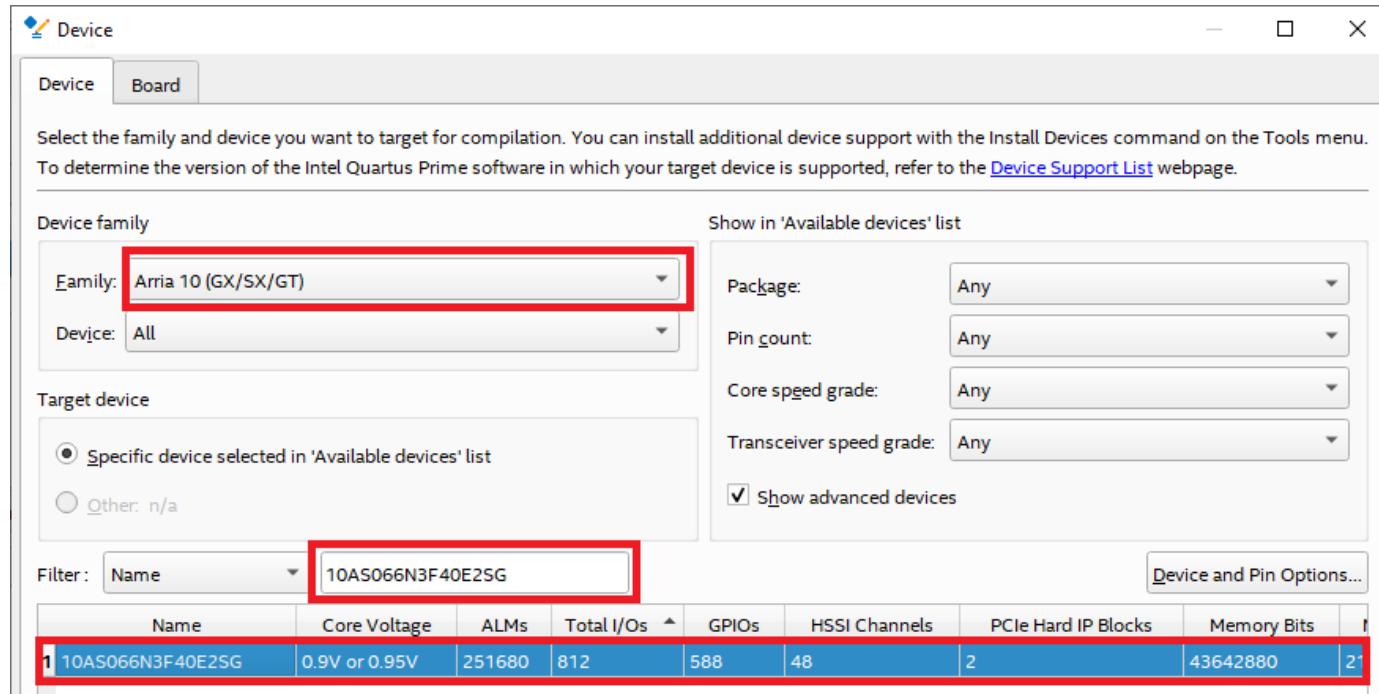
Reference design creation using Intel Quartus Pro

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. The HDL Coder SoC workflow generates an IP core that integrates with the reference design, and is then used to program an SoC board. The following figure describes the relationship between a reference design, an HDL IP core and an SoC board

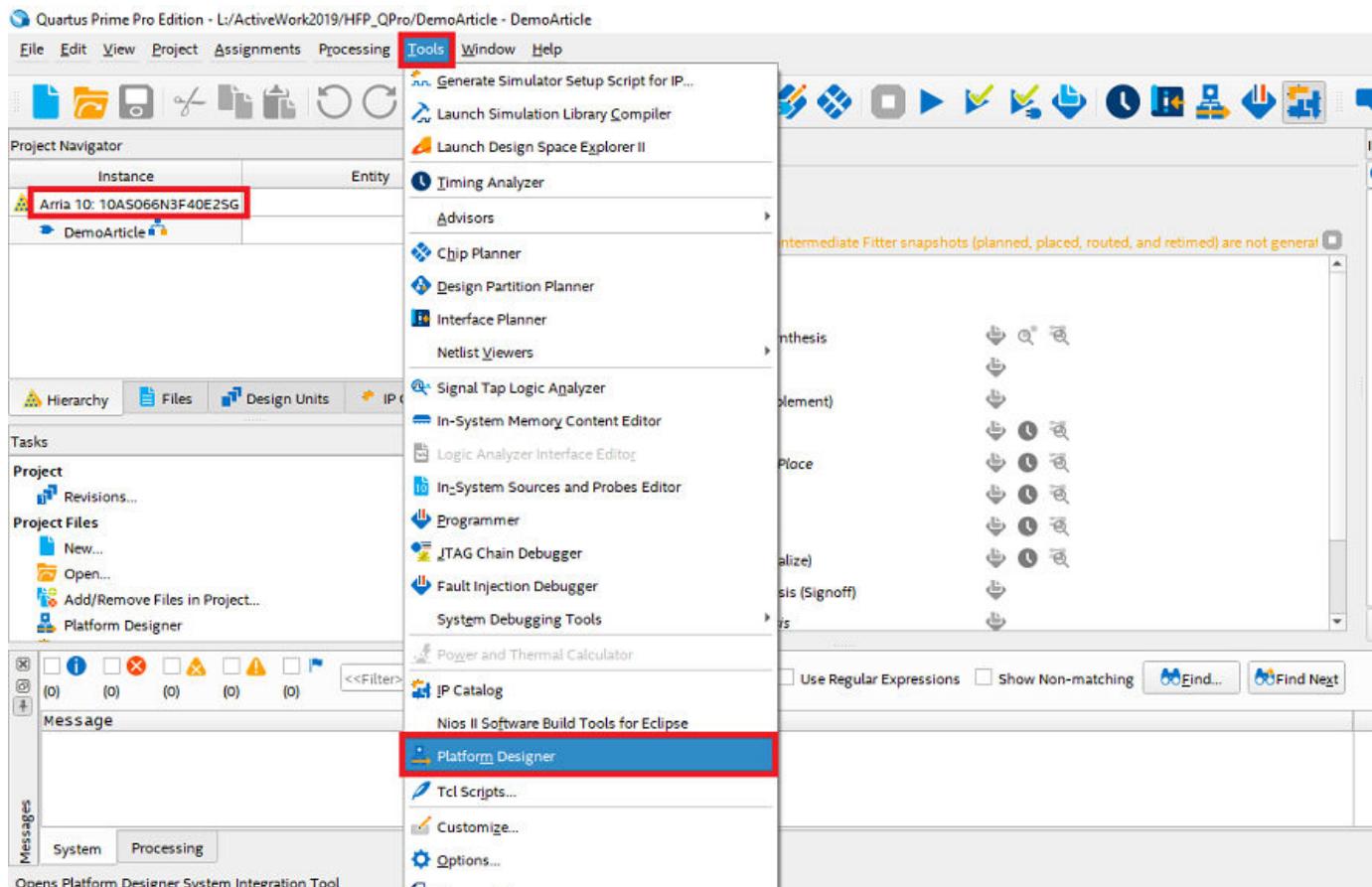


In this section, we outline the basic steps necessary to create and export a simple reference design using the Intel Quartus Pro and Platform Designer (QSys) environment. For more information about the QSys system integration tool, refer to Intel documentation.

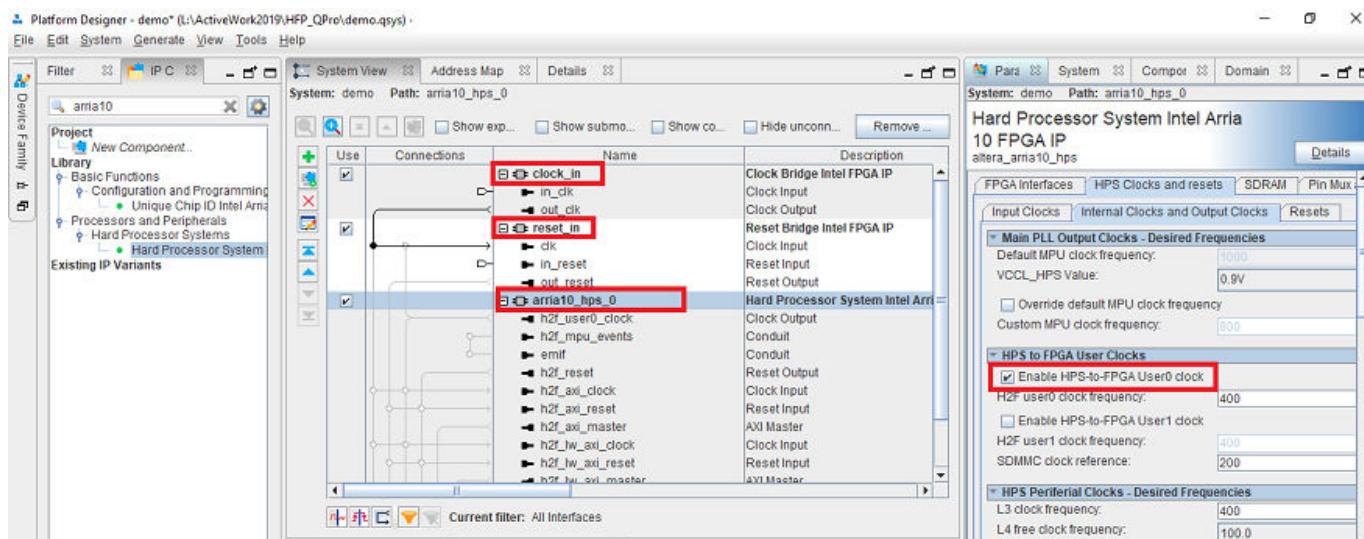
1. Create an empty Quartus project using the New project wizard with device part number as shown in the following figure.



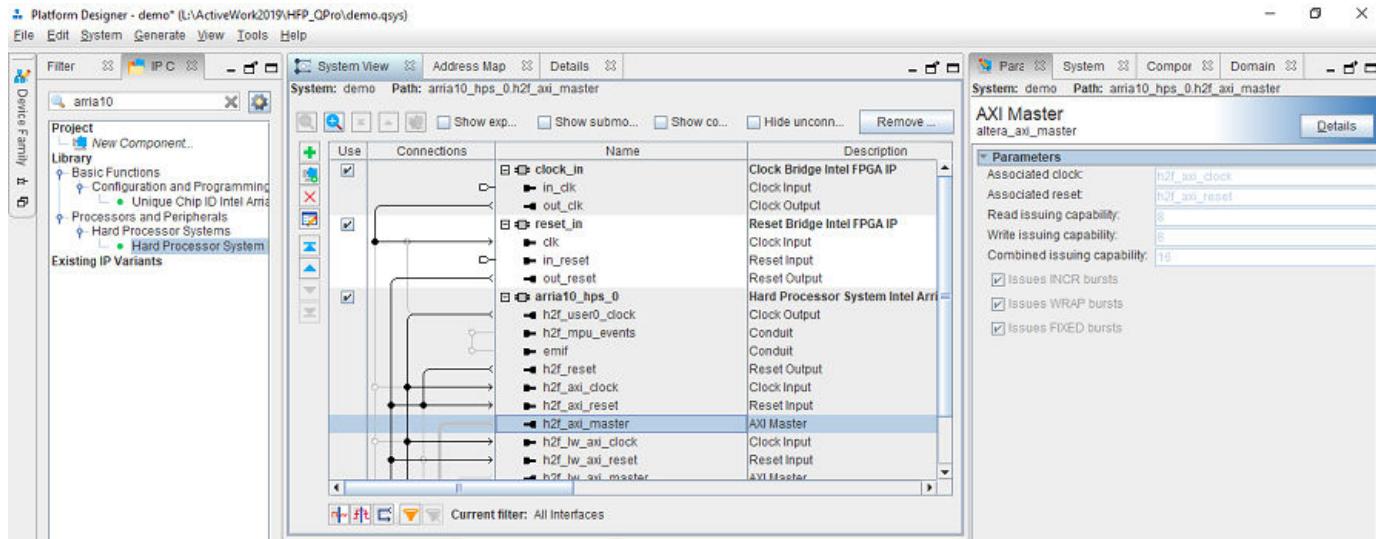
2. Initialize the Platform Designer(Qsys) in Quartus Pro by navigating to **Tools --> Platform Designer** as shown in the following figure.



3. Select Hard Processor System Intel Arria 10 FPGA IP(HPS), clock and reset IPs from IP catalog to the created Platform Designer project. Connect the required clocks and resets to the Arria 10 HPS IP as shown in the following figure. complete the other settings required for Arria 10 Hard Processor System such as Peripheral pin set and mode settings.



4. Keep h2f_axi_master port connection open in order to connect to DUT IP during the process of workflow IP integration. Complete the rest of the connections between Altera PLL IP and HPS IP as shown in the following figure.



5. Save the Qsys file. This file is used when you create reference design plugin.

Register the Intel Arria 10 SoC development kit.

In this section, You register the Intel Arria 10 SoC development kit in HDL Workflow Advisor.

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB path.

For more details on creating board registration file, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215.

2. Create the board definition file.

A board definition file contains information about the SoC board.

For more details on creating board defination file, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215.

Register the custom reference design in HDL Workflow Advisor

In this section, You register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

For more details on creating custom reference design, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215.

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that is generated by the HDL Coder SoC workflow. For more details on creating the

reference design definition file, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215.

Early I/O for Arria 10: The Intel Arria 10 SoC FPGA device supports Early I/O Release.

Early IO release allows you to enable DDR functioning prior programming the core raw binary file (RBF) for speeding up the boot time. In this flow, the shared I/O and hard memory controller I/O are configured and released allowing HPS immediate access to them.

This feature splits the FPGA configuration sequence into two parts. The first part configures the FPGA I/O, the Shared I/O and also enables the HPS External Memory Interface (EMIF) if present. The second part of the sequence configures the core FPGA fabric. By splitting the configuration sequence, the Arria10 Hard Processing System now has access to Shared I/O and EMIF before the FPGA fabric is configured. This allows more flexibility for designs that need faster boot times or alternate boot sources. In this Early I/O, two Raw Binary Format (.rbf) files are generated: (1) **peripheral.rbf** file. (2) **core.rbf** file. Together, these configuration files contain the same data as a combined configuration .rbf file that is generated when the Early I/O Release feature is not used. The **peripheral.rbf** file is loaded first and configures the FPGA I/O, Shared I/O and HPS EMIF. The **core.rbf** is loaded next and completes the FPGA configuration sequence by configuring the FPGA fabric. After the **peripheral.rbf** is successfully loaded, the Intel Arria 10 SoC FPGA HPS EMIF pins are released and the interface begins calibration.

To use this feature, you need to enable a reference design parameter `hRD.GenerateSplitBitstream = true;` as shown in the below `plugin_rd` file. Accordingly if this reference design parameter is made true, it generates two .rbf files for configuring the FPGA as mentioned above.

The contents of this reference design definition file `plugin_rd.m` is similar to the Intel Quartus standard version, the differences for the Intel Quartus Pro are listed below.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Intel Quartus Pro');

%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign( ...
    'CustomQsysPrjFile', 'system_soc.qsys');

% split the full rbf file into core and peripheral rbf files for Early I/O feature
hRD.GenerateSplitBitstream = true;
```

Partition your design for hardware and software implementation

The first step of the Intel SoC hardware-software co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

Group all the blocks you want to implement on programmable logic into an atomic subsystem. This atomic subsystem is the boundary of your hardware-software partition. All the blocks inside this subsystem are implemented on programmable logic, and all the blocks outside this subsystem will run on the ARM processor.

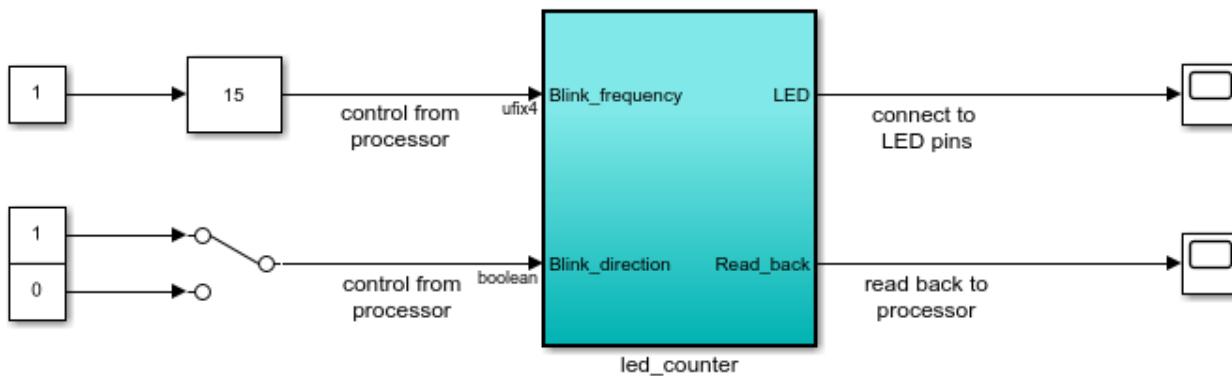
In this example, the subsystem **led_counter** is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are

control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem **led_counter** are for software implementation.

In Simulink, you can use the **Slider Gain** or **Manual Switch** block to adjust the input values of the hardware subsystem. In the embedded software, this means the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
open_system('hdlcoder_led_blinking_4bit');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking_4bit/led_counter')`

[Launch HDL Workflow Advisor](#)

[Run Demo](#)

Copyright 2014-2017 The MathWorks, Inc.

Generate an HDL IP core using the HDL Workflow Advisor

Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Intel Qsys environment.

1. Start the IP core generation workflow.

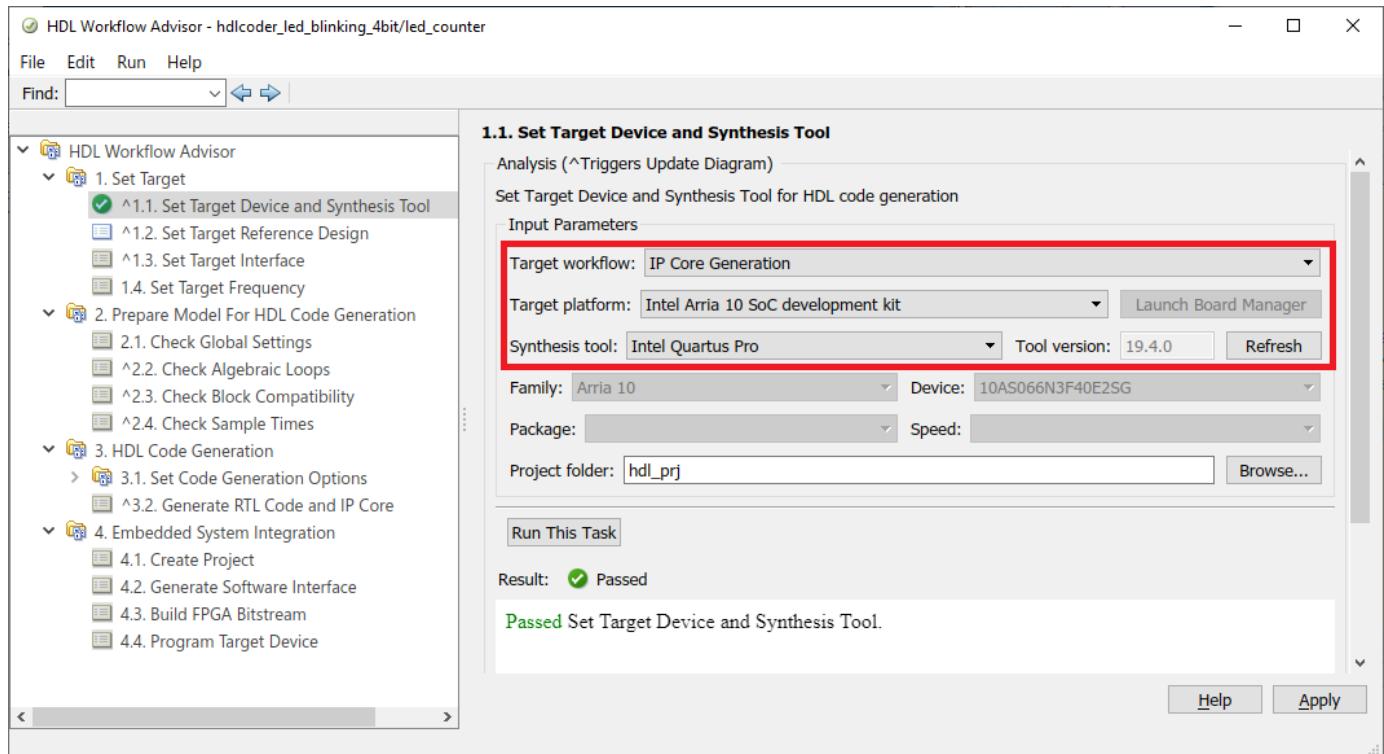
1.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking_4bit/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code > HDL Workflow Advisor**.

1.2. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

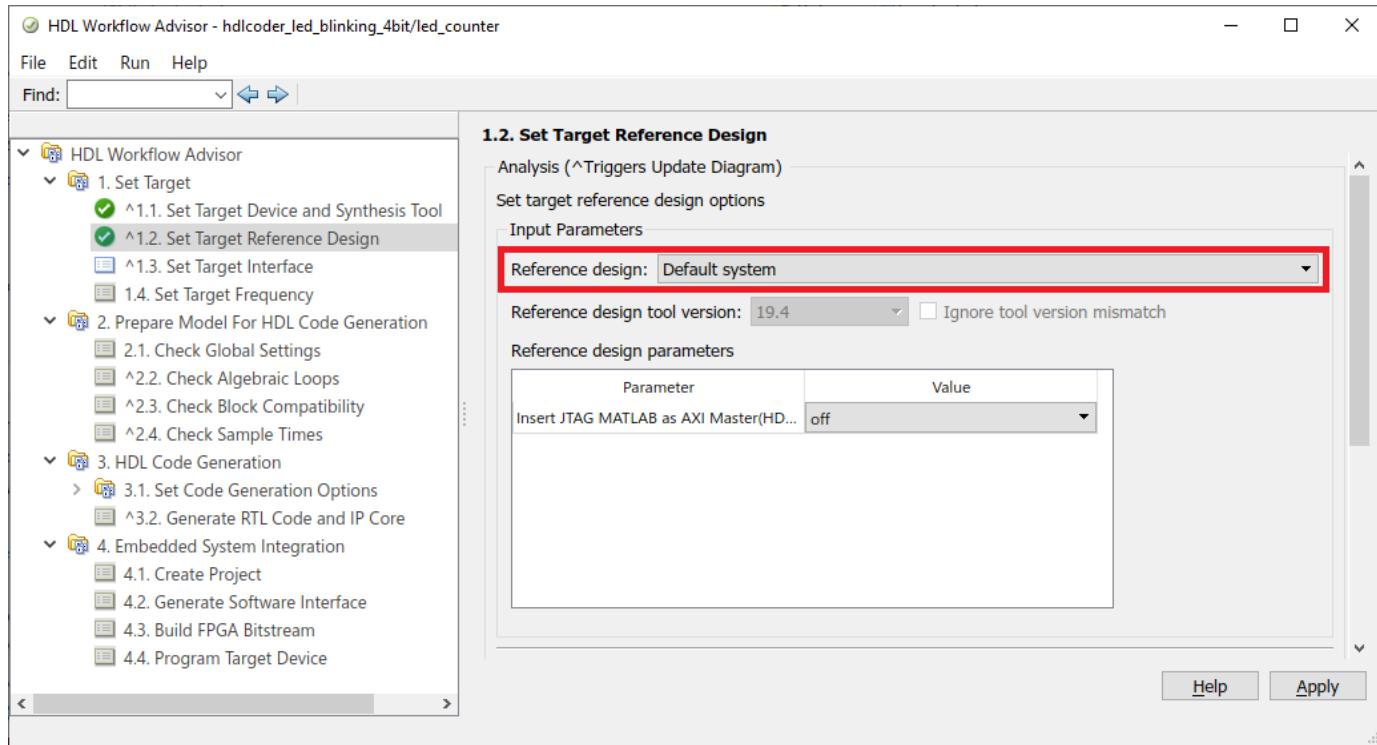
1.3. For **Target platform**, select **Intel Arria 10 SoC development kit**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Intel SoC Devices and follow the instructions provided by the Support Package Installer to complete the installation.

1.4. Select the **Synthesis tool** as **Intel Quartus Pro** (or **Altera QUARTUS II**)

1.5. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



1.6. In the **Set Target > Set Target Reference Design** task, choose **Default system**. For this example, it is selected by default.



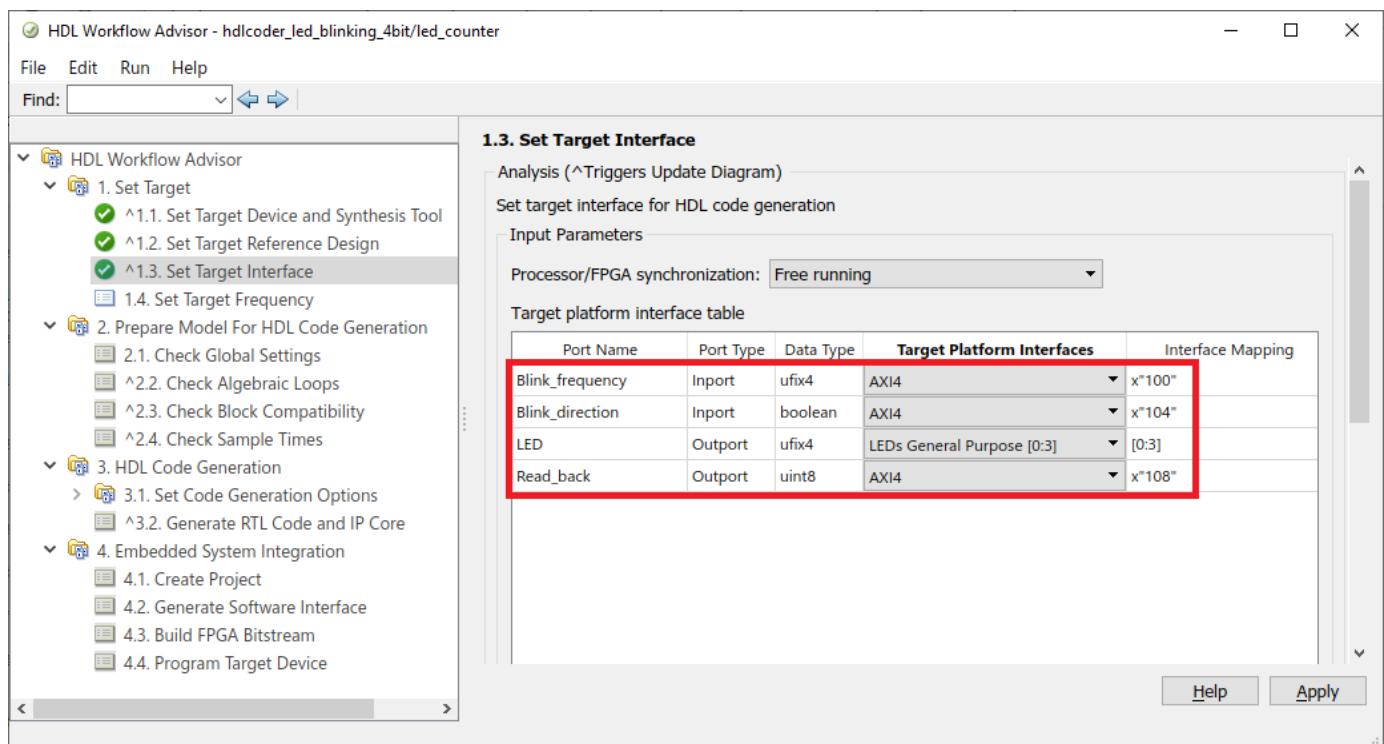
1.7. Click **Run This Task** to run the **Set Target Reference Design** task.

2. Configure the target interface.

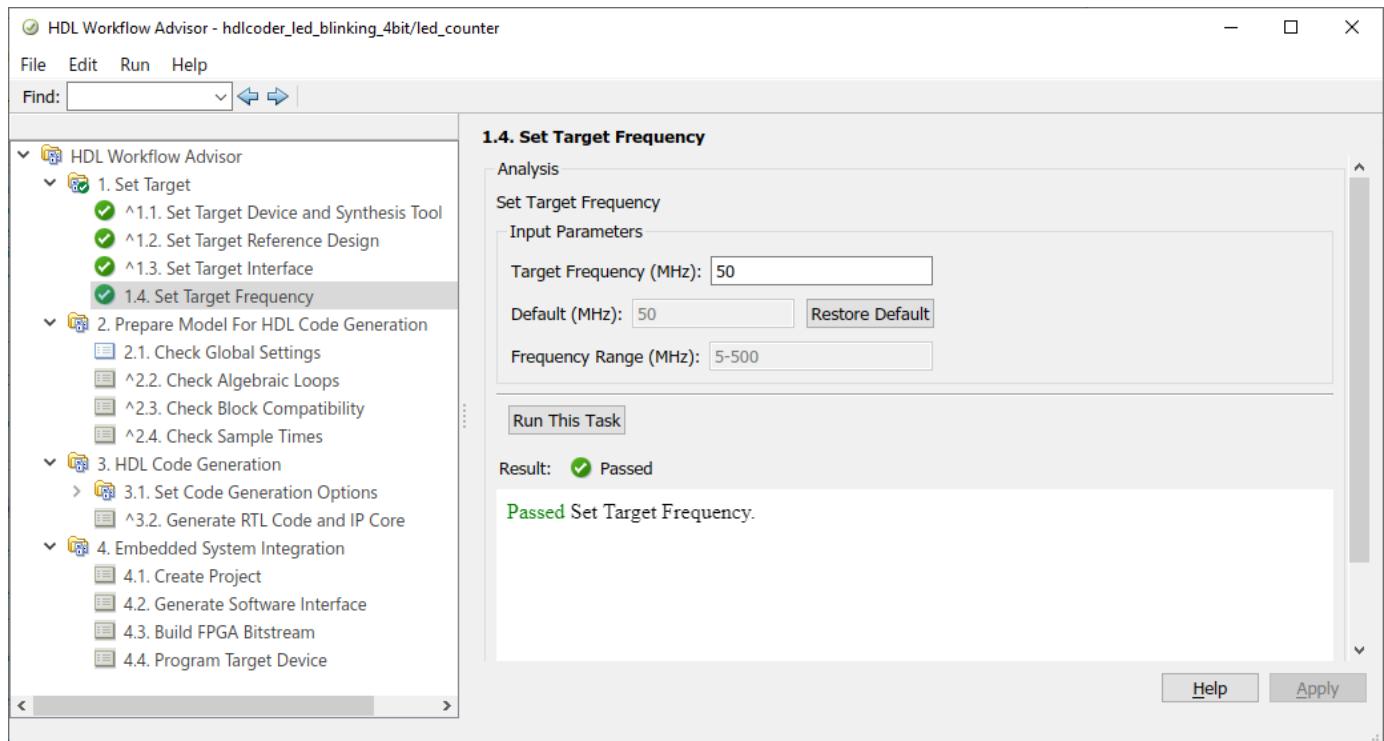
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4 interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:3]**, which connects to the LED hardware on the Intel SoC board.

2.1 In the **Set Target > Set Target Interface** task, choose **AXI4** for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2.2 Choose **LEDs General Purpose [0:3]** for **LED**.

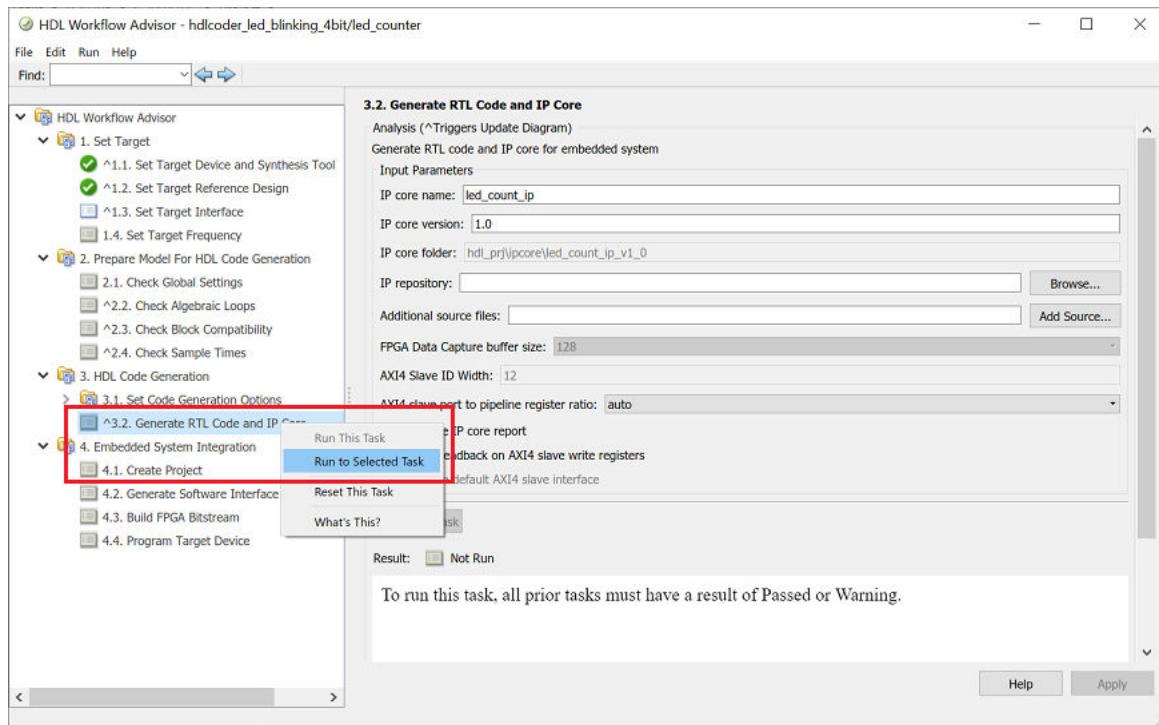


2.3 In the Set Target > Set Target Frequency task, choose Target Frequency as 50 MHz.



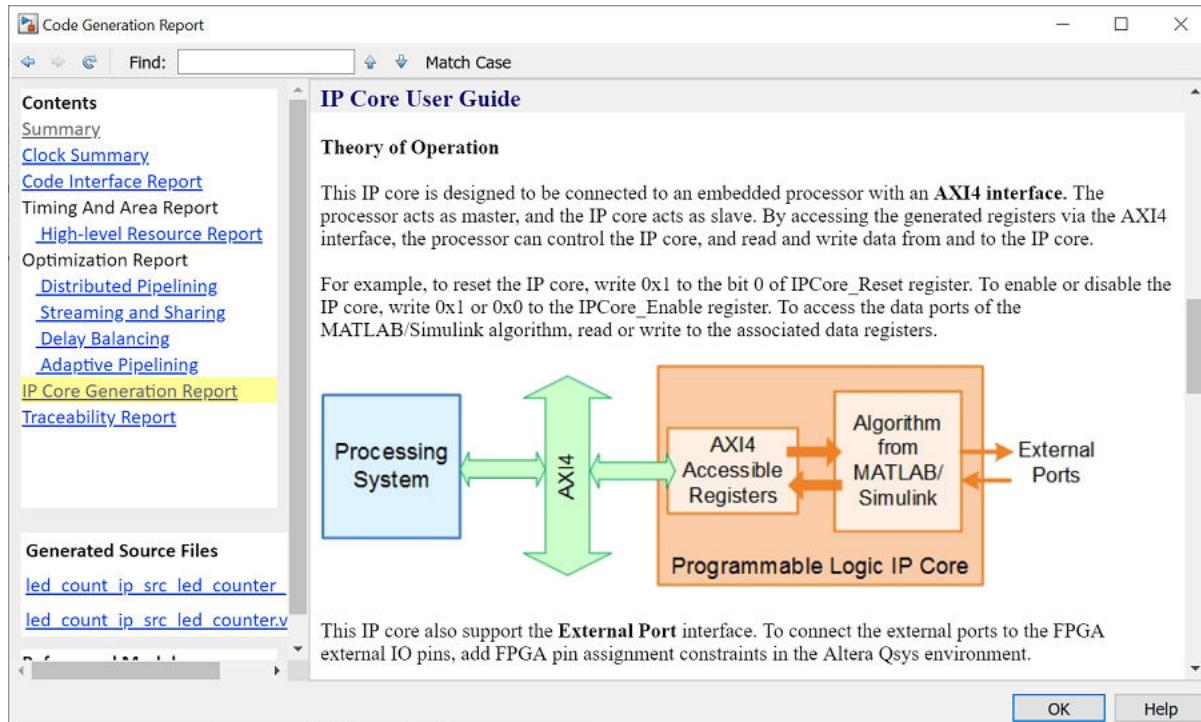
3. Generate the IP Core.

To generate the IP core, right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.



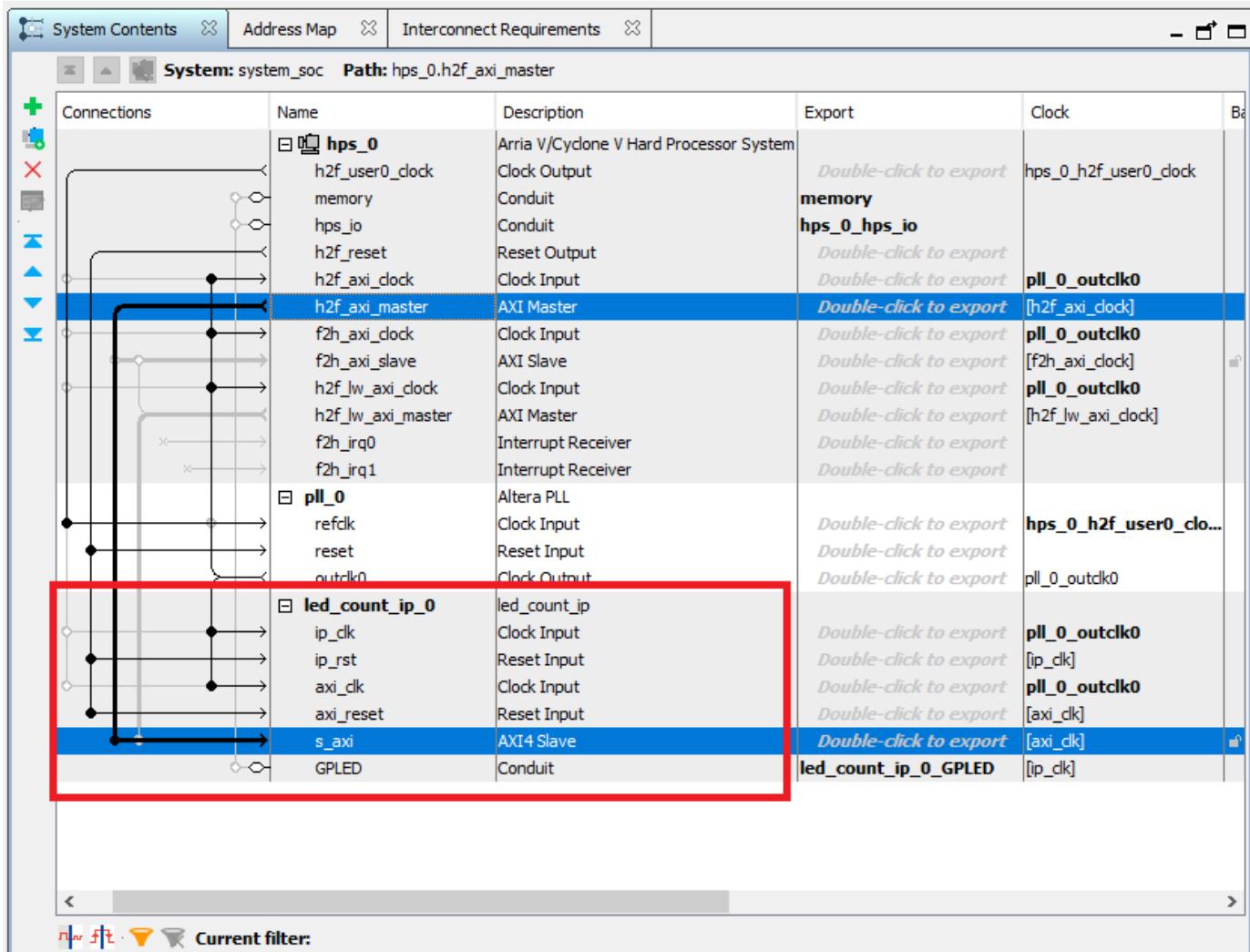
4. Generate and view the IP core report.

After you generate the custom IP core, the IP core files are in the **ipcore** folder within your project folder. An HTML custom IP core report is generated together with the custom IP core. The report describes the behavior and contents of the generated custom IP core.



5. Follow step 1 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to integrate the IP core in the reference design and create the Qsys project.

6. Now let us examine the Intel Qsys project created by the SoC workflow after completing the Create Project task under Embedded System Integration. The following figure shows the SoC project where we have highlighted the HDL IP Core. It is instructive to compare this project with the previous project used in the custom reference design plugin for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



7. Follow the steps 2, 3 and 4 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to generate software interface model, generate FPGA bitstream and program target device respectively.

8. The LEDs on the Arria 10 SoC will start blinking after loading the bitstream. In addition, you can control the LED blink frequency and direction by executing the software interface model. Refer to the example “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to control the LED blink frequency and direction from the generated software interface model.

Save Target Hardware Settings in Model

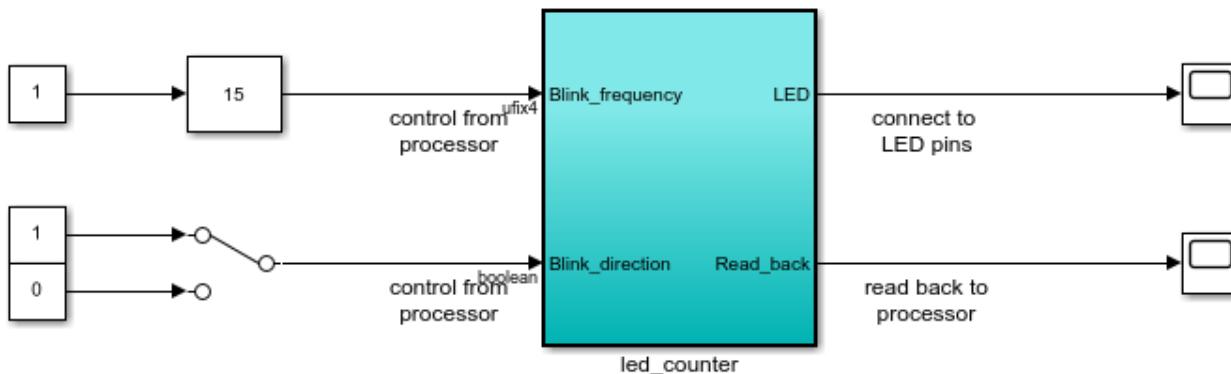
This example shows how to save your target hardware settings in a Simulink® model.

This example also shows different ways that you can export, modify, and import target hardware settings. This example uses the Xilinx Zynq platform, but in the same way, you can save target hardware settings in models that target the Intel SoC devices, FPGA Turnkey, and Simulink Real-Time FPGA I/O boards.

Open the Model

```
open_system('hdlcoder_led_blinking');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

Launch HDL Workflow Advisor

Run Demo

Copyright 2012 The MathWorks, Inc.

Configure the Target Hardware Settings

When you configure the target hardware settings, you modify the model. If you save the model, the target hardware settings are saved as part of the model.

You can configure the target hardware settings in three ways:

- HDL Workflow Advisor
- HDL Block Properties dialog box for Import or Outport
- `hdlset_param`

Since the HDL Workflow Advisor provides a dropdown menu for each target hardware option, it is best to use the HDL Workflow Advisor when you configure the target hardware settings for the first time. After you save the model with a valid configuration, you can view, modify, and apply settings from the command line.

Use HDL Workflow Advisor to configure model or port hardware settings

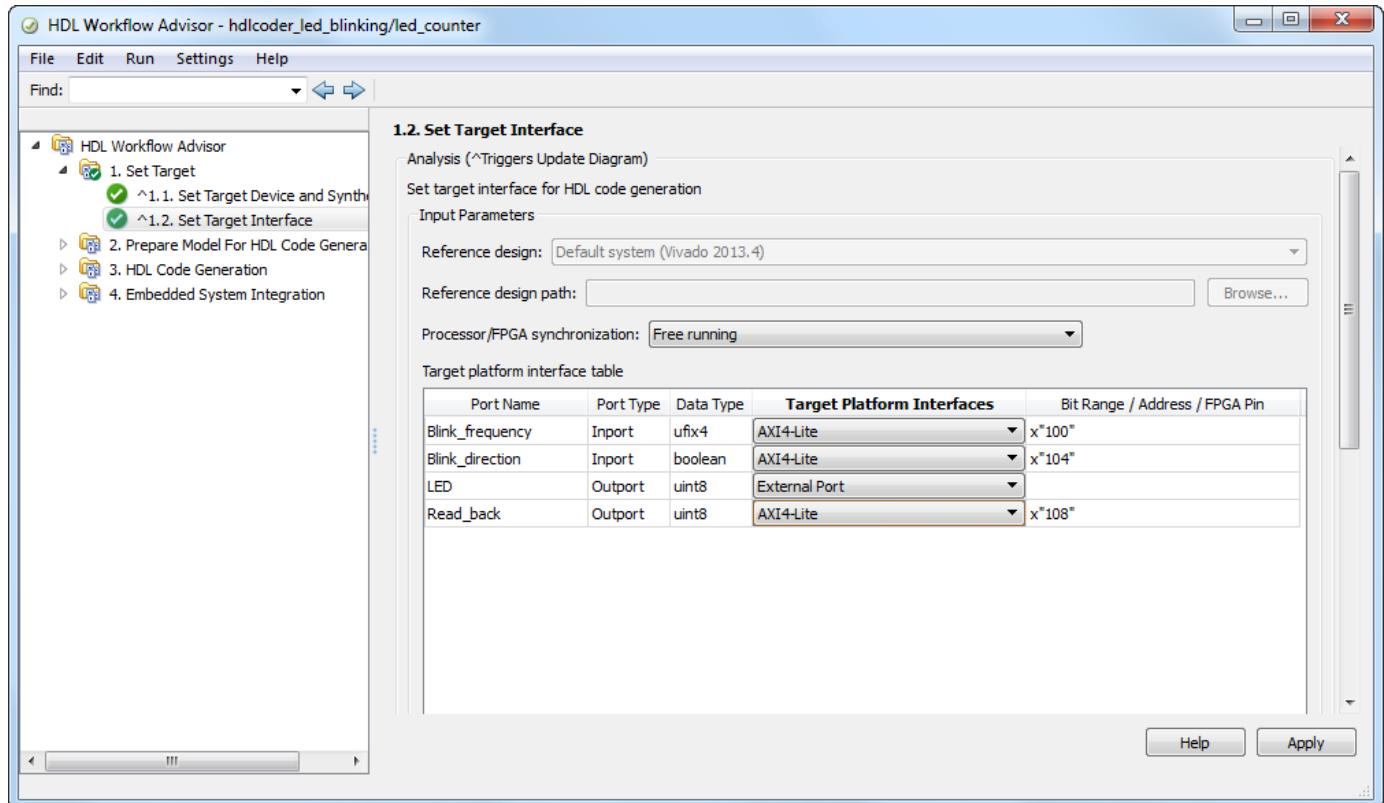
Open the HDL Workflow Advisor from the subsystem `hdlcoder_led_blinking/led_counter` and specify your target hardware settings in tasks 1.1 and 1.2.

In the **Set Target > Set Target Device and Synthesis Tool** task:

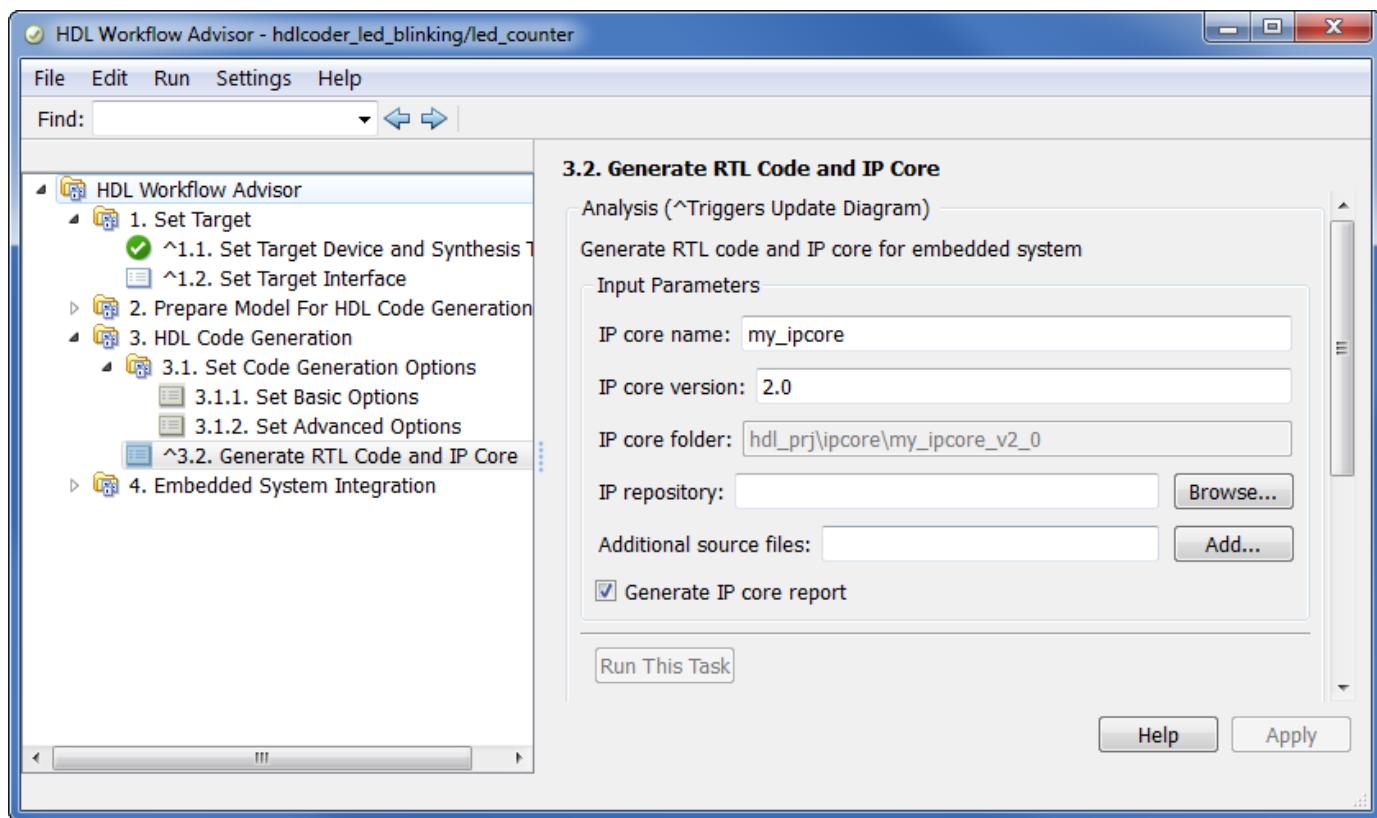
- For **Target workflow**, select IP Core Generation.
- For **Target platform**, select Xilinx Zynq ZC702 evaluation kit.

In the **Set Target > Set Target Interface** task, map the ports to interfaces as follows:

- For **Blink_frequency** and **Blink_direction** input ports, select the AXI4-Lite interface.
- For the **LED** output port, select External Port.
- For the **Read_back** output port, select the AXI4-Lite interface.



Specify HDL IP core name and version in task 3.2 **Generate RTL Code and IP Core**.



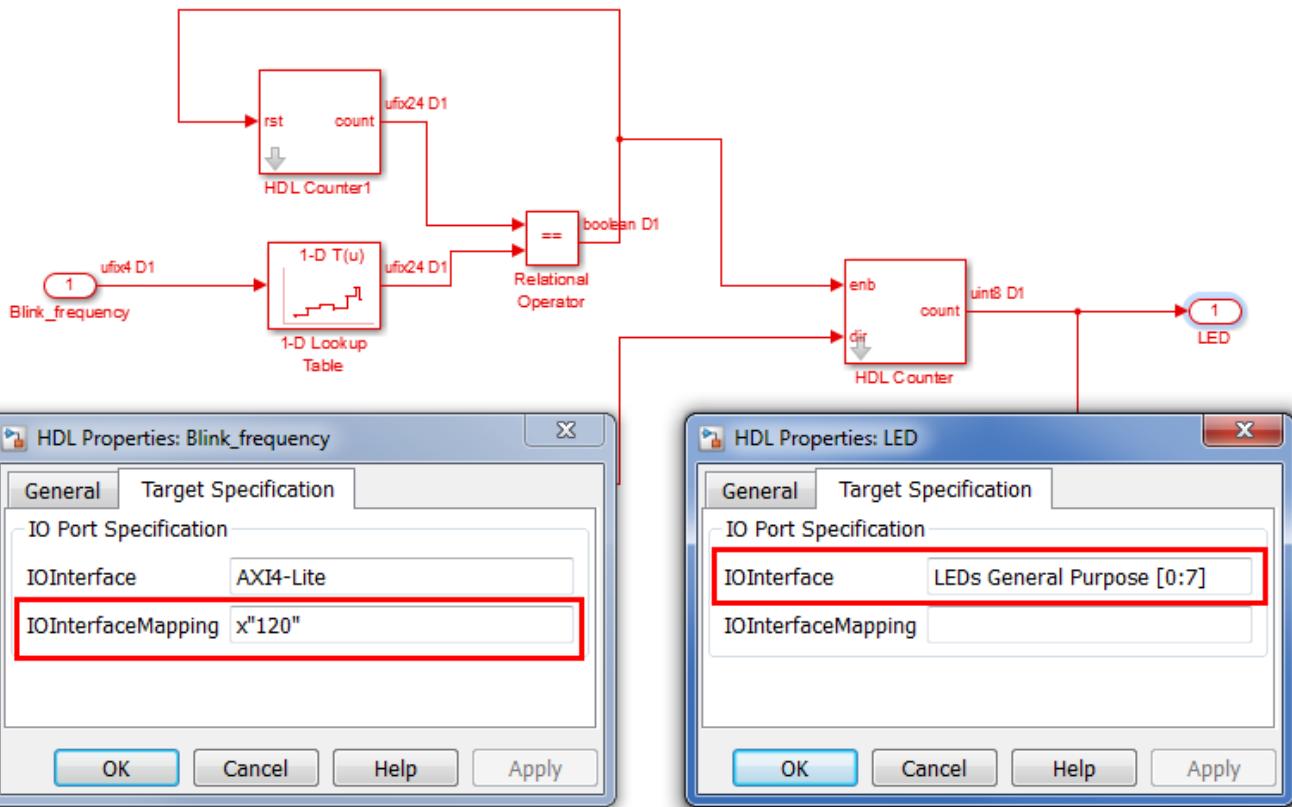
For details, see “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65.

Use HDL Block Properties dialog box to map DUT ports to target interface

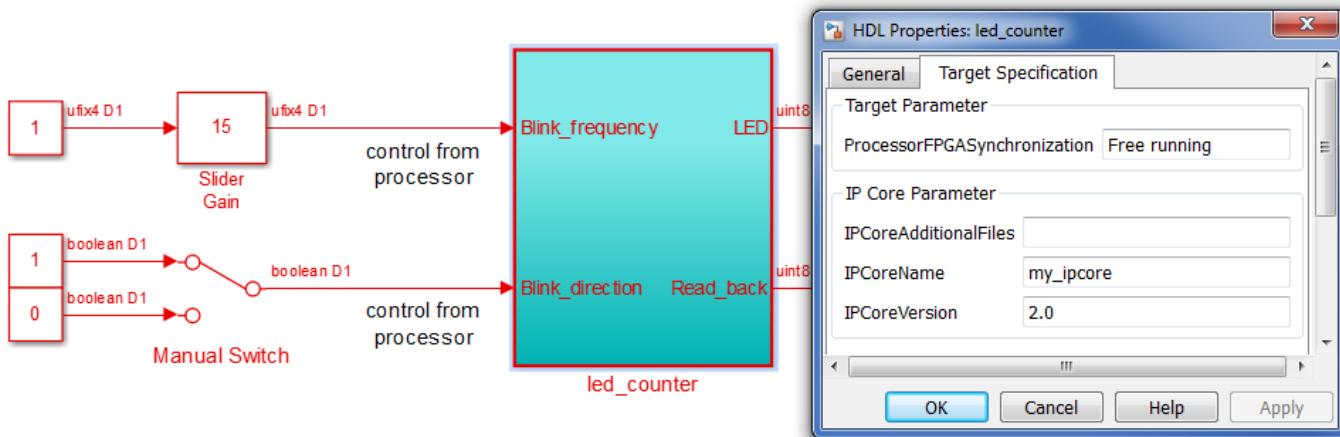
You can specify target interface settings for the DUT interface by using the HDL Block Properties dialog box for any Import or Outport. You can also specify the HDL IP core settings by using the HDL Block Properties dialog box for the DUT subsystem. However, you can use the HDL Block Properties dialog box to configure only the DUT target interface and HDL IP core settings. Set other target hardware settings from the HDL Workflow Advisor, or by using `hdlset_param` at the command line.

For example, you can change the bit range of the **Blink_frequency** Import to x"120" and remap the **LED** Outport to LEDs General Purpose [0:7]:

- 1 From the subsystem `hdlcoder_led_blinking/led_counter`, right-click the **Blink_frequency** Import, and select **HDL Code > HDL Block Properties**. Click the **Target Specification** Tab. For **IOInterfaceMapping**, enter x"120".
- 2 Similarly, for the **LED** Outport, for **IOInterface**, enter **LEDs General Purpose [0:7]**.



Right-click the subsystem `hdlcoder_led_blinking/led_counter`, and select **HDL Code > HDL Block Properties**. Note you can change **IPCoreName** and **IPCoreVersion** under **Target Specification** Tab.



The target interface and HDL IP core settings you specify using the HDL Block Properties dialog box are validated when you open the HDL Workflow Advisor.

Use `hdlset_param` to configure model or DUT port hardware settings

To configure target hardware settings for your model or DUT ports, you can use `hdlset_param`.

For example, to change the **TargetPlatform** to Xilinx Zynq ZC706 evaluation kit, enter:

```
hdlset_param('hdlcoder_led_blinking', 'TargetPlatform', 'Xilinx Zynq ZC706 evaluation kit');
```

To set the Bit Range of **Blink_frequency** Input to `x"120"`; and set the **LED** Outport to LEDs General Purpose [0:7], enter:

```
hdlset_param('hdlcoder_led_blinking/led_counter/Blink_frequency', 'IOInterfaceMapping', 'x"120"';
hdlset_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface', 'LEDs General Purpose [0:7]
```

To set the IP core name and version, enter:

```
hdlset_param('hdlcoder_led_blinking/led_counter', 'IPCoreName', 'my_ipcore');
hdlset_param('hdlcoder_led_blinking/led_counter', 'IPCoreVersion', '2.0');
```

Export and Import Target Hardware Settings

To export all non-default HDL code generation options in your model, including the target hardware settings, you can use **hdlsaveparams** and **hdlrestoreparams**. You can modify the model settings in the saved MATLAB file, and apply the settings to the same model or to a different model.

For example, to export the settings from the `hdlcoder_led_blinking` model to a MATLAB file, `targetSetting.m`, enter:

```
hdlsaveparams('hdlcoder_led_blinking/led_counter', 'targetSetting.m')
```

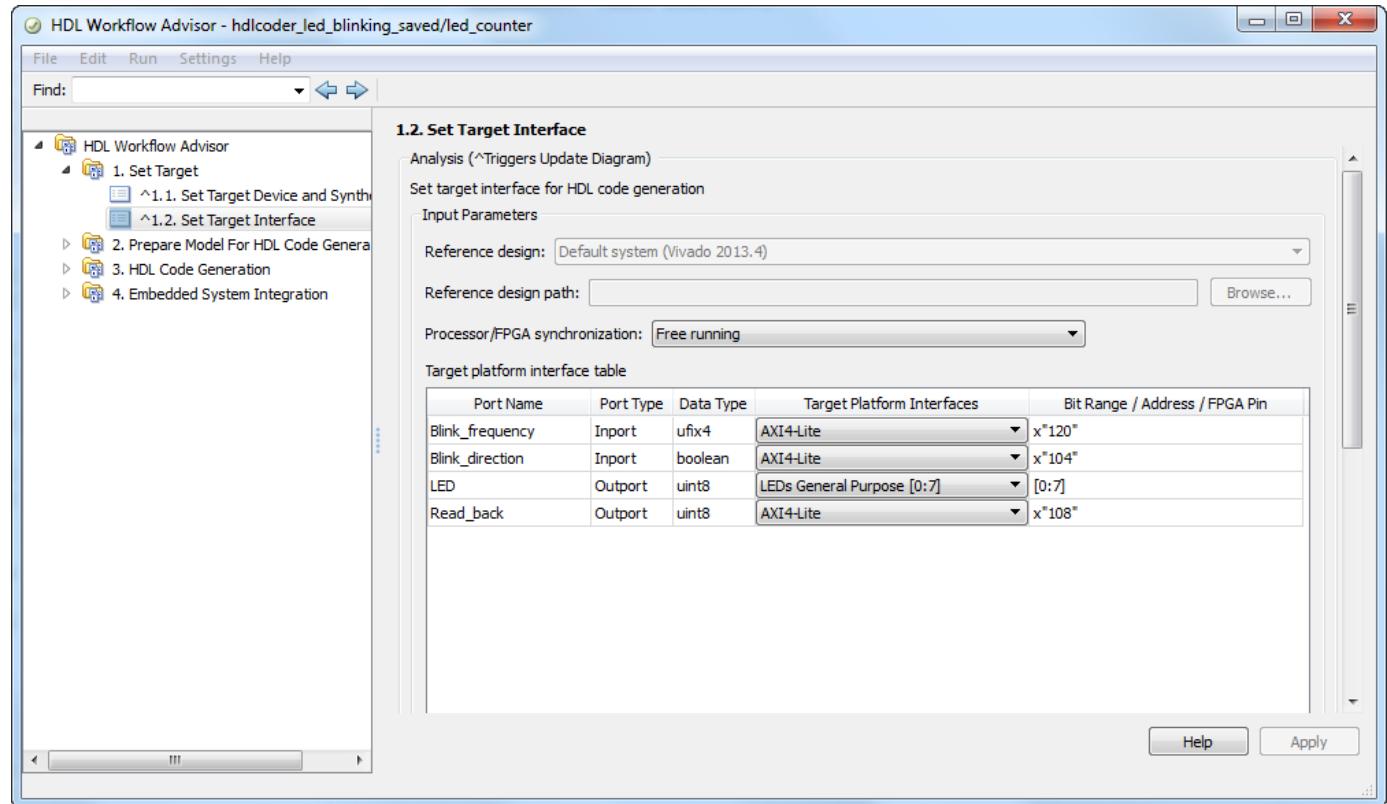
You can modify the settings in `targetSetting.m` as desired, then enter the following command to apply the settings to the model:

```
hdlrestoreparams('hdlcoder_led_blinking/led_counter', 'targetSetting.m')
```

Save and Reopen the Model

- 1** Save the model `hdlcoder_led_blinking` as `hdlcoder_led_blinking_saved`.
- 2** Open the saved model, `hdlcoder_led_blinking_saved`.
- 3** Open the HDL Workflow Advisor from the subsystem `hdlcoder_led_blinking_saved/led_counter`.

Notice that the modified settings are automatically loaded to tasks 1.1 and 1.2 in the HDL Workflow Advisor.



Using IP Core Generation Workflow from MATLAB: LED Blinking

This example shows how to use MATLAB® HDL Workflow Advisor to generate a custom HDL IP core which blinks LEDs on FPGA board. The generated IP core can be used on Xilinx® Zynq® platform, or on any Xilinx FPGA with MicroBlaze processor.

Introduction

You can use MATLAB to design, simulate, and verify your application, perform what-if scenarios with algorithms, and optimize parameters. You can then prepare your design for hardware and software implementation on the Zynq-7000 AP SoC by deciding which system elements will be performed by the programmable logic, and which system elements will run on the ARM® Cortex-A9.

Using the guided workflow shown in this example, you can automatically generate VHDL® code for the programmable logic using HDL Coder™, export hardware information from the automatically generated EDK project to an SDK project for integration of handwritten C code for the ARM processor, and implement the design on the Xilinx Zynq Platform.

This example is a step-by-step guide that helps introduce you to the HW/SW co-design workflow. In this workflow, you perform the following steps:

- 1** Set up your Zynq hardware and tools.
- 2** Partition your design for hardware and software implementation.
- 3** Generate an HDL IP core using MATLAB HDL Workflow Advisor.
- 4** Integrate the IP core into a Xilinx EDK project and program the Zynq hardware.

For more information, refer to other more advanced examples, and the HDL Coder documentation.

Requirements

- 1** Xilinx ISE 14.4
- 2** Xilinx Zynq-7000 SoC ZC702 Evaluation Kit running the Linux® image in the Base Targeted Reference Design 14.4
- 3** HDL Coder Support Package for Xilinx Zynq Platform

Set up Zynq hardware and tools

- 1.** Set up the Xilinx Zynq ZC702 evaluation kit. Please follow the hardware setup steps in HDL Coder example "Getting Started with HW/SW Co-Design Workflow for Xilinx Zynq Platform".
- 2.** Set up the Xilinx ISE synthesis tool path using the following command in the MATLAB command window. Use your own ISE installation path when you run the command.

```
hdlsetupoolpath('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\14.4\ISE_DS\ISE\bin\nt64\ise.exe')
```

Partition your design for hardware and software implementation

The first step of the Zynq HW/SW co-design workflow is to decide which parts of your design to implement on the programmable logic, and which parts to run on the ARM processor.

Group the parts of your algorithm that you want to implement on programmable logic into a MATLAB function. This function is the boundary of your hardware/software partition. All the MATLAB code within this function will be implemented on programmable logic. You must provide C code that implements the MATLAB code outside this function to run on the ARM processor.

In this example, the function **mlhdlc_ip_core_led_blinking** is implemented on hardware. It models a counter that blinks the LEDs on an FPGA board. Two input ports, **Blink_frequency** and **Blink_direction**, are control ports that determine the LED blink frequency and direction. You can adjust the input values of the hardware subsystem via prompt options in the included embedded software, 'mlhdlc_ip_core_led_blinking_driver.c' and 'mlhdlc_ip_core_led_blinking_driver.h'. The embedded software, which runs on the ARM processor, controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem, **LED**, connects to the LED hardware. The output port, **Read_Back**, can be used to read data back to the processor.

```
design_name = 'mlhdlc_ip_core_led_blinking';
testbench_name = 'mlhdlc_ip_core_led_blinking_tb';
sw_driver_name = 'mlhdlc_ip_core_led_blinking_driver.c';
sw_driver_header_name = 'mlhdlc_ip_core_led_blinking_driver.h';
```

Let us take a look at the MATLAB design.

```
type(design_name);

function [LED, Read_back] = mlhdlc_ip_core_led_blinking(Blink_frequency, Blink_direction)
%
% Copyright 2013-2015 The MathWorks, Inc.

persistent freqCounter LEDCounter

if isempty(freqCounter)
    freqCounter = 0;
    LEDCounter = 255;
end

if Blink_frequency <= 0
    Blink_frequency = 0;
elseif Blink_frequency >= 15
    Blink_frequency = 15;
end

blinkFrequencyOut = LookupTable(Blink_frequency);
if blinkFrequencyOut == freqCounter
    freqMatch = 1;
else
    freqMatch = 0;
end

freqCounter = freqCounter + 1;

if freqMatch
    freqCounter = 0;
end

if Blink_direction
    LED = 255 - LEDCounter;
else
    LED = LEDCounter;
end

if LEDCounter == 255
    LEDCounter = 0;
```

```

elseif freqMatch
    LEDCounter = LEDCounter + 1;
end

Read_back = LED;
end

function y = LookupTable(idx)
s = 2.^(23:-1:8)';
y = s(idx+1);
end

type(testbench_name);

%
% Copyright 2013-2015 The MathWorks, Inc.

for i=1:16
    [yout, ~] = mlhdlc_ip_core_led_blinking(i-1, 0);
    [yout2, ~] = mlhdlc_ip_core_led_blinking(i-1, 1);
end

```

Setup for the Example

The following commands copy the necessary example files into a temporary folder.

```

mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_ip_core_led_blinking'];

% Create a temporary folder and copy the MATLAB files.
cd(tempdir);
[~, ~, ~] = rmdir(mlhdlc_temp_dir, 's');
mkdir(mlhdlc_temp_dir);
cd(mlhdlc_temp_dir);

% Copy the design files to the temporary directory
copyfile(fullfile(mlhdlc_demo_dir, [design_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, [testbench_name, '.m*']), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, sw_driver_name), mlhdlc_temp_dir);
copyfile(fullfile(mlhdlc_demo_dir, sw_driver_header_name), mlhdlc_temp_dir);

```

Create a New HDL Coder Project

```
coder -hdlcoder -new mlhdlc_ip_core_led_blinking_prj
```

Next, add the file 'mlhdlc_ip_core_led_blinking.m' to the project as the MATLAB Function and 'mlhdlc_ip_core_led_blinking_tb.m' as the MATLAB Test Bench.

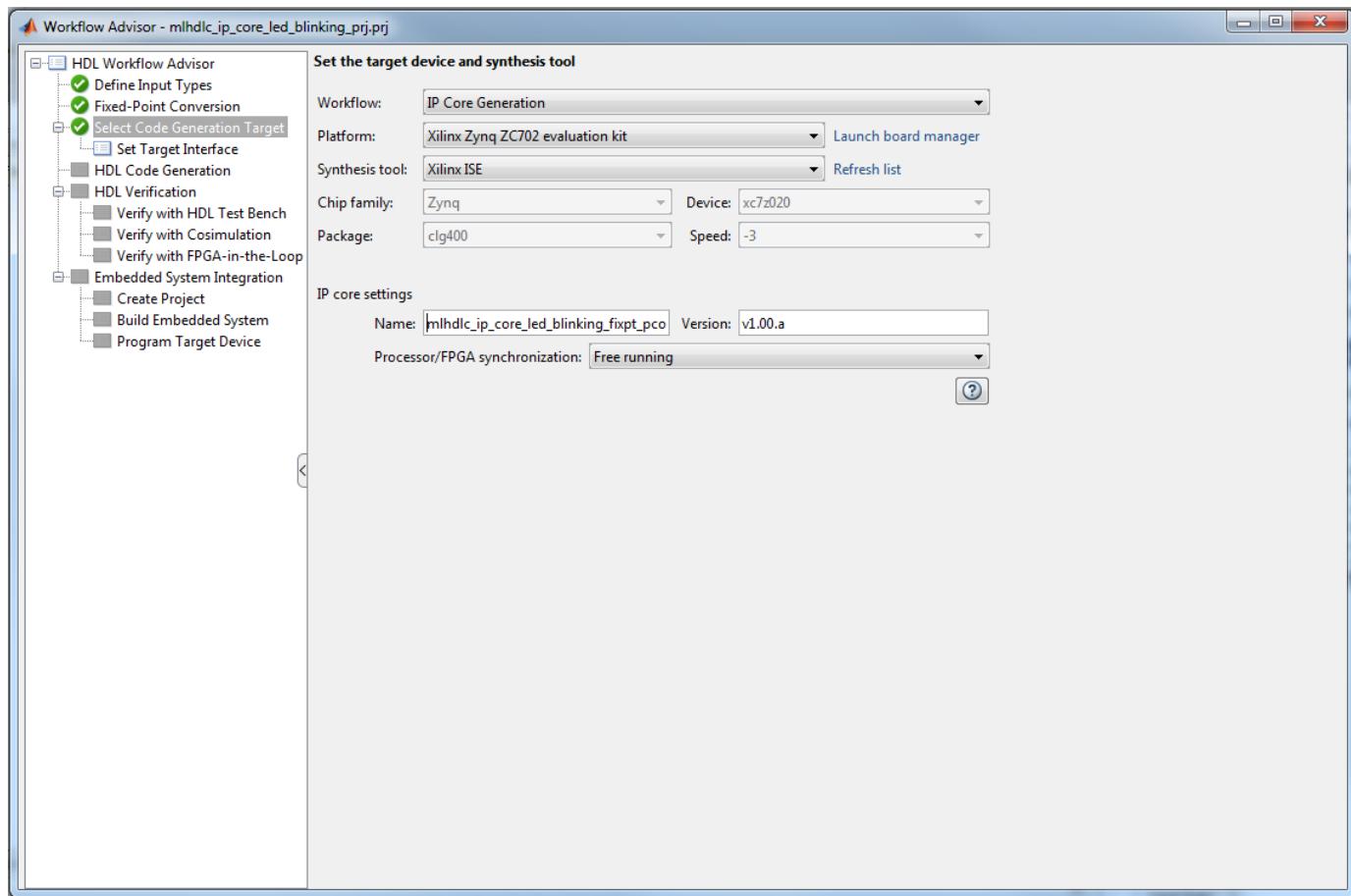
See "Getting Started with MATLAB to HDL Workflow" for a more complete tutorial on creating and populating MATLAB HDL Coder projects.

Select Code Generation Target

Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a MATLAB function. The generated IP core is designed to be connected to an embedded processor on an FPGA device. HDL Coder generates HDL code from the MATLAB design function, and also generates HDL code for the AXI interface logic connecting the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Xilinx EDK environment.

To choose the IP core Generation workflow:

1. Open the HDL Workflow Advisor and right-click **Select Code Generation Target**.
2. For **Workflow**, select **IP Core Generation**.



Platform Selection

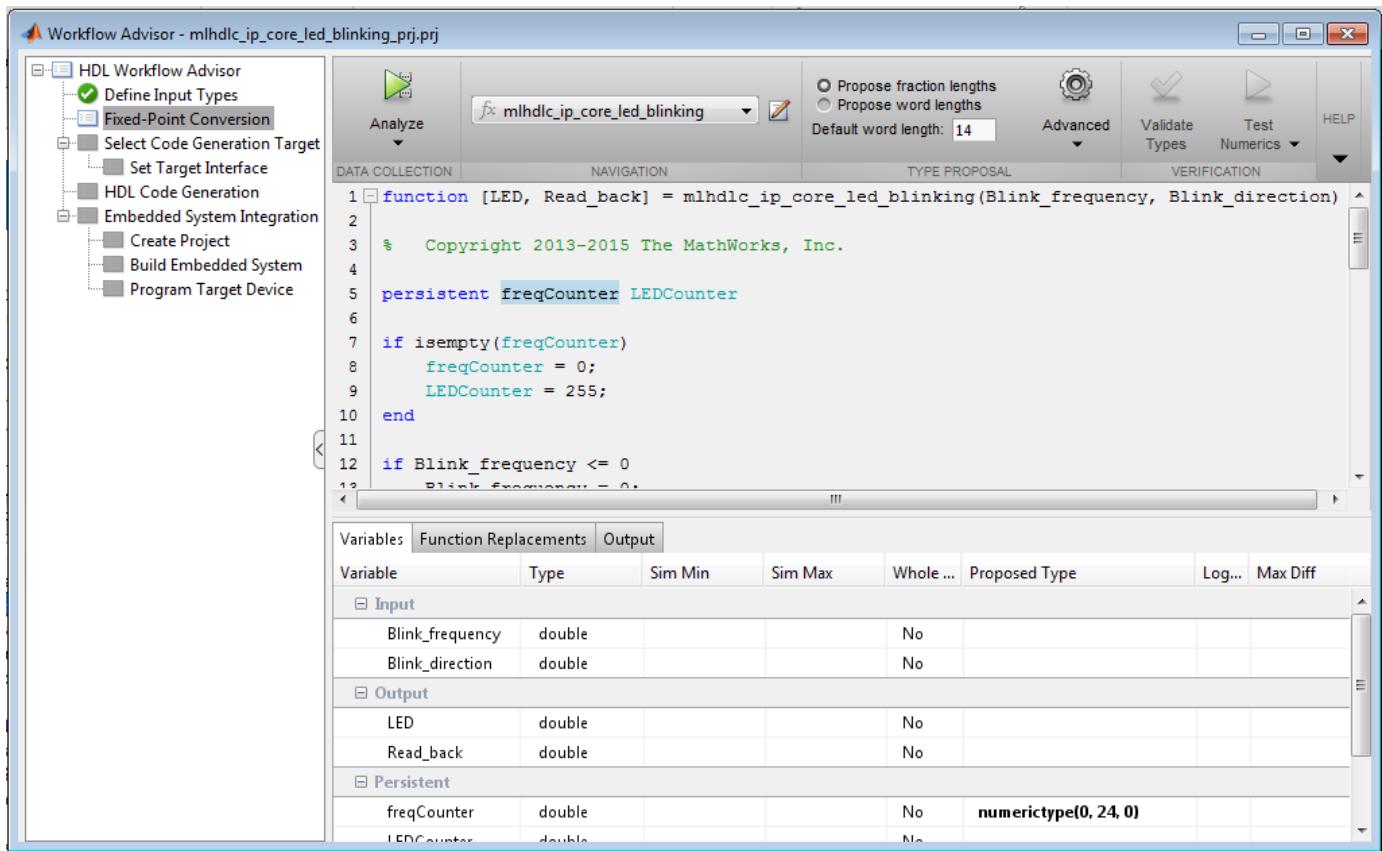
There is a generic option called **Generic Xilinx Platform** in the platform selection. This option is board-independent and generates a generic Xilinx IP core, which has to be manually integrated into your EDK environment.

The remaining options are board-specific and provide the additional capability of integrating the generated IP core into a Xilinx PlanAhead project, synthesize the project and download the bitstream to FPGA within the HDL Workflow Advisor.

For **Platform**, select **Xilinx Zynq ZC702 evaluation kit**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx Zynq Platform and follow the instructions provided by the Support Package Installer to complete the installation.

Convert Design To Fixed-Point

1. Right-click the **Define Input Types** task and select **Run This Task**.
2. In the **Fixed-Point Conversion** task, click **Advanced** and set the **Safety margin for sim min/max (%)** to 0.
3. Set the proposed type of the **freqCounter** variable to unsigned 24-bit integer by entering **numerictype(0, 24, 0)** in its 'Proposed Type' column.

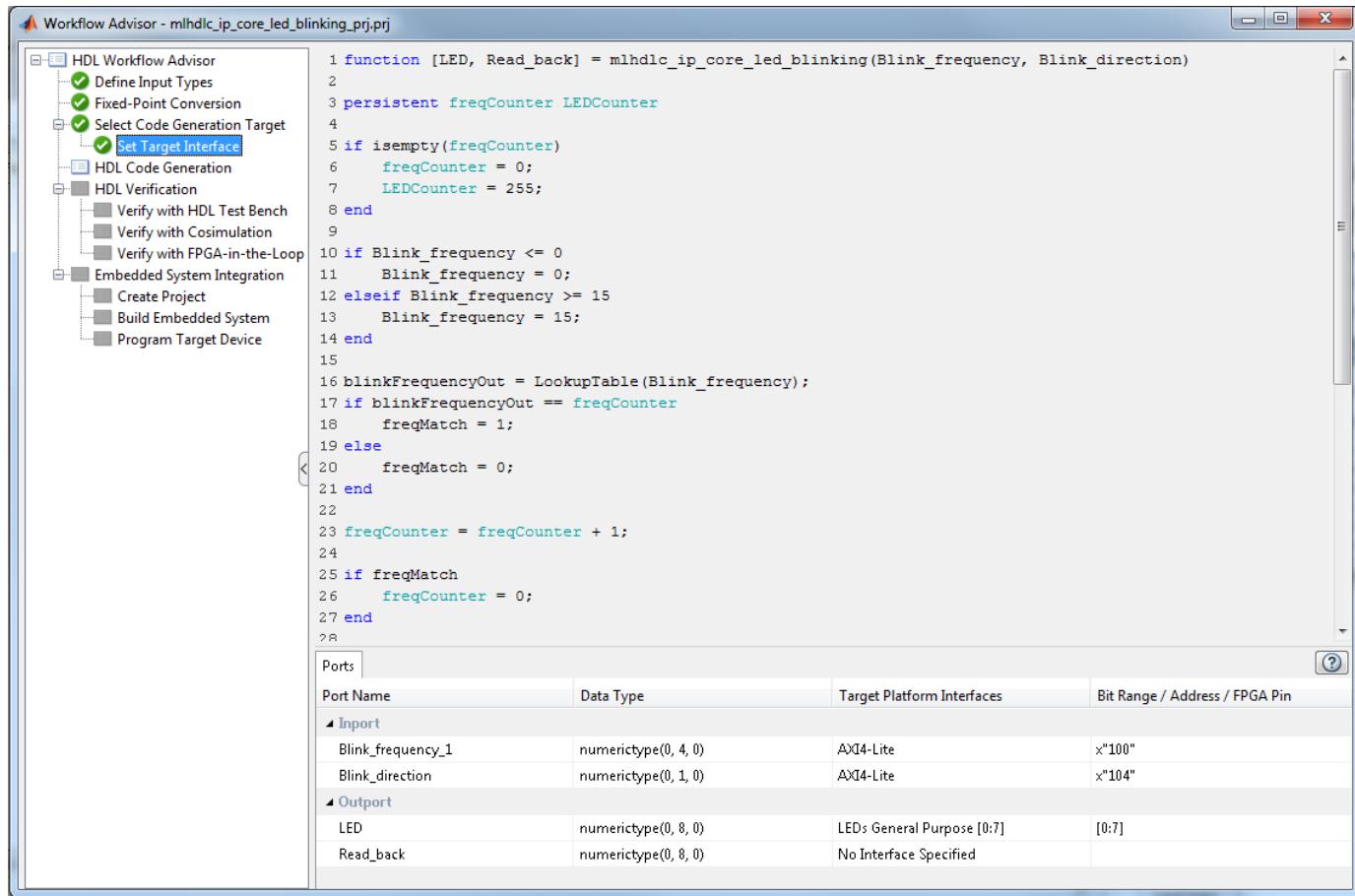


4. On the left, right-click the **Fixed-Point Conversion** task and select **Run This Task**.

Configure the Target Interface

Map each port in your MATLAB design function to one of the IP core target interfaces in the **Set Target Interface** subtask.

In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder™ generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:7]**, which connects to the LED hardware on the Zynq board.



The screenshot shows the 'Workflow Advisor - mlhdlc_ip_core_led_blinking_prj.prj' window. The left pane displays a tree view of the workflow steps:

- HDL Workflow Advisor
 - Define Input Types (green checkmark)
 - Fixed-Point Conversion (green checkmark)
 - Select Code Generation Target
 - Set Target Interface** (highlighted with a red box)
- HDL Code Generation
- HDL Verification
- Embedded System Integration
 - Verify with HDL Test Bench
 - Verify with Cosimulation
 - Verify with FPGA-in-the-Loop
- Program Target Device

The right pane shows the generated HDL code for the **mlhdlc_ip_core_led_blinking** function:

```

1 function [LED, Read_back] = mlhdlc_ip_core_led_blinking(Blink_frequency, Blink_direction)
2
3 persistent freqCounter LEDCounter
4
5 if isempty(freqCounter)
6     freqCounter = 0;
7     LEDCounter = 255;
8 end
9
10 if Blink_frequency <= 0
11     Blink_frequency = 0;
12 elseif Blink_frequency >= 15
13     Blink_frequency = 15;
14 end
15
16 blinkFrequencyOut = LookupTable(Blink_frequency);
17 if blinkFrequencyOut == freqCounter
18     freqMatch = 1;
19 else
20     freqMatch = 0;
21 end
22
23 freqCounter = freqCounter + 1;
24
25 if freqMatch
26     freqCounter = 0;
27 end
28

```

Below the code is a table titled 'Ports' showing the port mapping:

Port Name	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Import			
Blink_frequency_1	numerictype(0, 4, 0)	AXI4-Lite	x"100"
Blink_direction	numerictype(0, 1, 0)	AXI4-Lite	x"104"
Output			
LED	numerictype(0, 8, 0)	LEDs General Purpose [0:7]	[0:7]
Read_back	numerictype(0, 8, 0)	No Interface Specified	

Generate IP Core

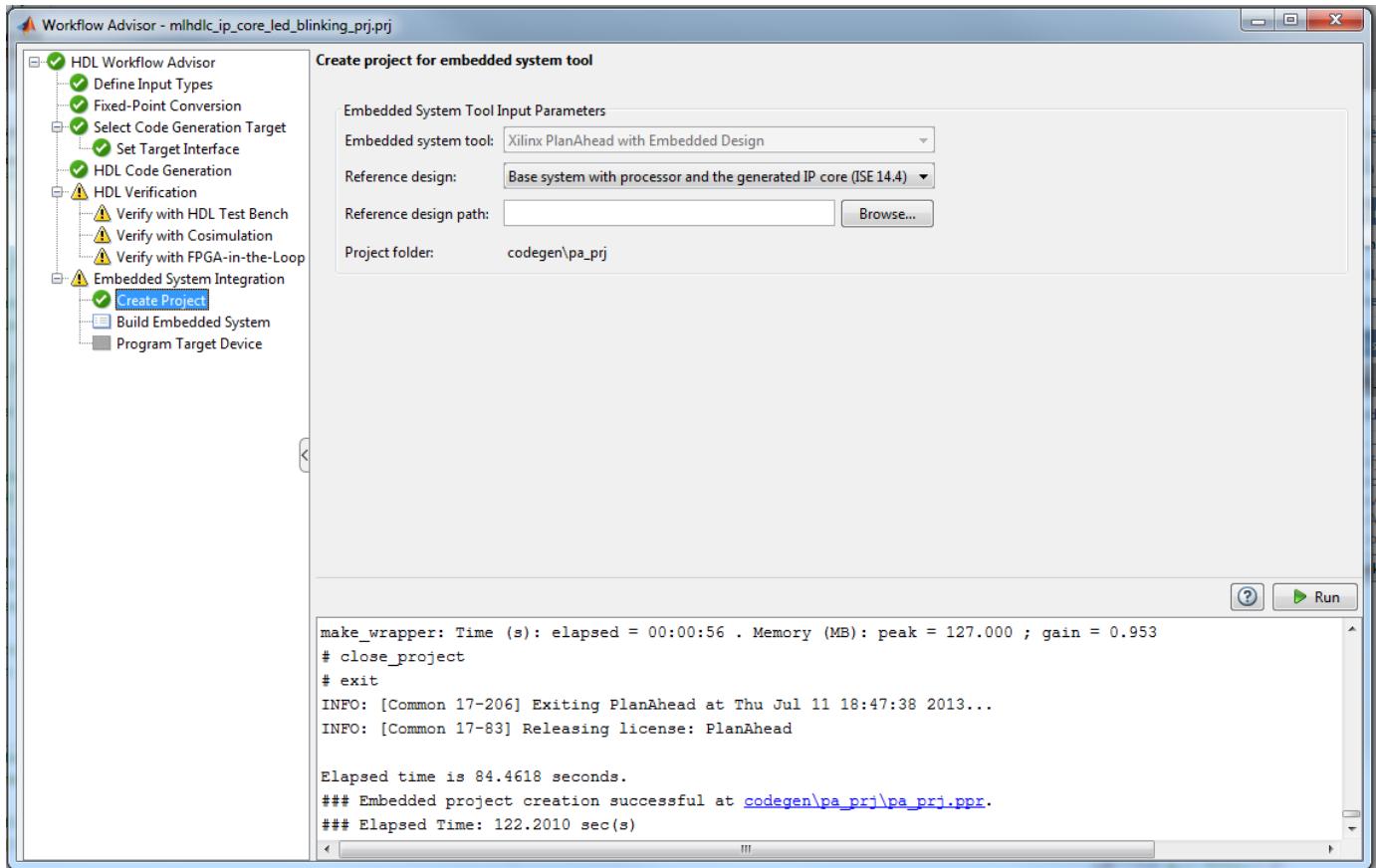
Right-click the **HDL Code Generation** step and select **Run this task** to generate the IP Core along with the IP Core Report.

Integrate the IP core with the Xilinx EDK environment

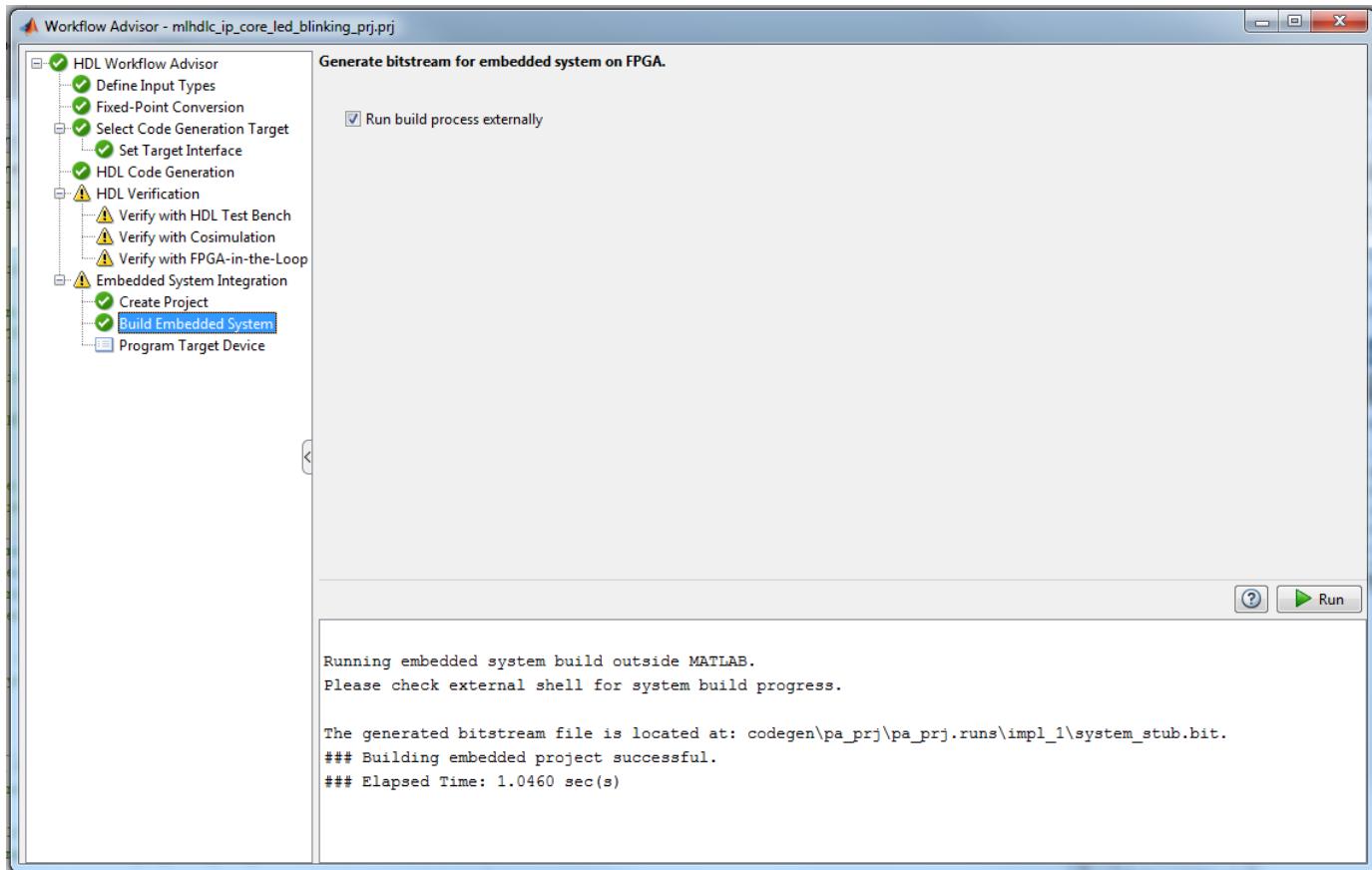
In this part of the workflow, you insert your generated IP core into a embedded system reference design, generate an FPGA bitstream, and download the bitstream to the Zynq hardware.

The reference design is a predefined Xilinx EDK project. It contains all the elements the Xilinx software needs to deploy your design to the Zynq platform, except for the custom IP core and embedded software.

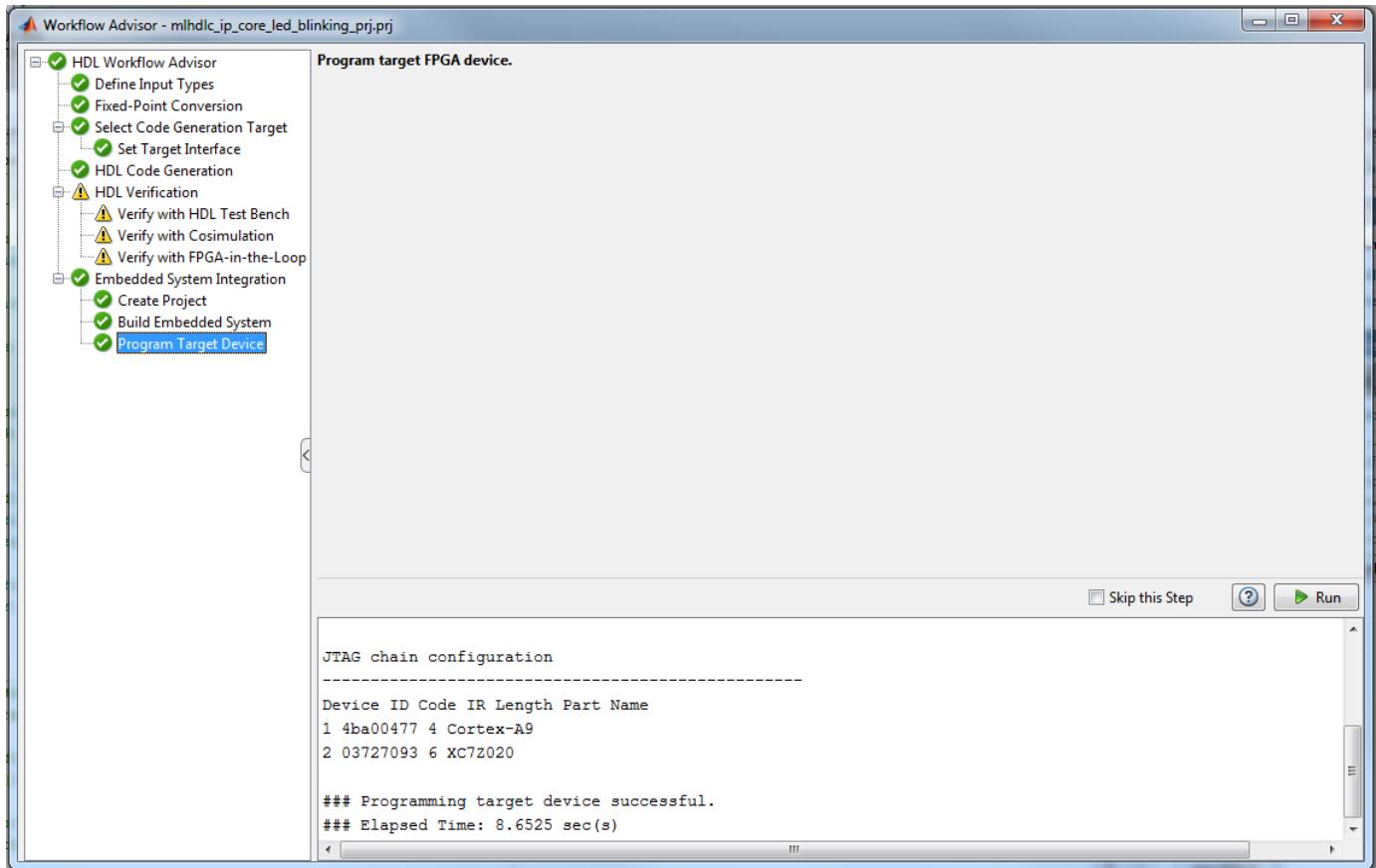
1. To integrate with the Xilinx EDK environment, right-click the **Create Project** step under 'Embedded System Integration', and choose the option 'Run This Task'. A Xilinx PlanAhead project with EDK embedded design is generated, and a link to the project is provided in the dialog window. You can optionally open up the project to take a look.



2. Build the FPGA bitstream in the **Build Embedded System** step. Make sure the 'Run build process externally' option is checked, so the Xilinx synthesis tool will run in a separate process from MATLAB. Wait for the synthesis tool process to finish running in the external command window.



3. After the bitstream is generated, right-click the 'Program Target Device' step and choose the option **Run This Task** to program the Zynq hardware.



After you program the FPGA hardware, the LED starts blinking on your Zynq board.

Next, you will integrate the included handwritten C code to run on the ARM processor to control the LED blink frequency and direction.

Run the software on Zynq ZC702 hardware

The included C code files, '**mlhdlc_ip_core_led_blinking_driver.c**' and '**mlhdlc_ip_core_led_blinking_driver.h**', implement a simple menu that enables you to set the LED blink frequency and direction. You can use them for your Linux-based SDK project.

```
----- MLHDL LED Blinking IP: -----
1 -> Change blinking frequency
2 -> Change blinking direction
0 -> Exit

Enter your choice :1
Please enter the frequency index [0:15]
2

Possible choices: 0, 1, 2
----- MLHDL LED Blinking IP: -----

1 -> Change blinking frequency
2 -> Change blinking direction
0 -> Exit

Enter your choice :2
Please specify the blinking direction? (1 - up, 0 - down)
0
```

For instructions on how to integrate the included C code into an SDK project and run it on Zynq ZC702 hardware, please refer to the Xilinx documentation.

Clean up the Generated Files

Run the following commands to clean up the temporary project folder.

```
mlhdlc_demo_dir = fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'matlabhdlcoderde...
mlhdlc_temp_dir = [tempdir 'mlhdlc_ip_core_led_blinking'];
clear mex;
cd (mlhdlc_demo_dir);
rmdir(mlhdlc_temp_dir, 's');
```

IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705

This example shows how to use the HDL Coder™ IP Core Generation Workflow to develop reference designs for Xilinx® parts without an embedded ARM® processor present, but which still utilize the HDL Coder™ generated AXI interface to control the DUT. This example uses MATLAB as AXI Master IP from HDL Verifier™ to access the HDL Coder™ generated DUT registers by enabling the reference design parameter option **Insert JTAG MATAB as AXI Master**. You can then access DUT registers from MATLAB directly. Alternatively, you can use Xilinx JTAG AXI Master to access the DUT registers using Vivado Tcl Console by writing Tcl commands. For Xilinx JTAG AXI Master, you need to create a custom reference design. The FPGA design is implemented on the Xilinx Kintex-7 KC705 board.

Requirements

- Xilinx Vivado Design Suite, with supported version listed in the HDL Coder documentation
- Xilinx Kintex-7 KC705 development board
- HDL Coder™ support package for Xilinx FPGA Boards
- (Optional) HDL Verifier™ support package for Xilinx FPGA Boards

Xilinx Kintex-7 KC705 development board



Example Reference Designs

There are many designs which will benefit from using the HDL Coder™ IP Core Generation Workflow without using either an embedded ARM® processor or an Embedded Coder™ Support Package, but which still leverages the HDL Coder generated AXI4-Lite registers. These designs include:

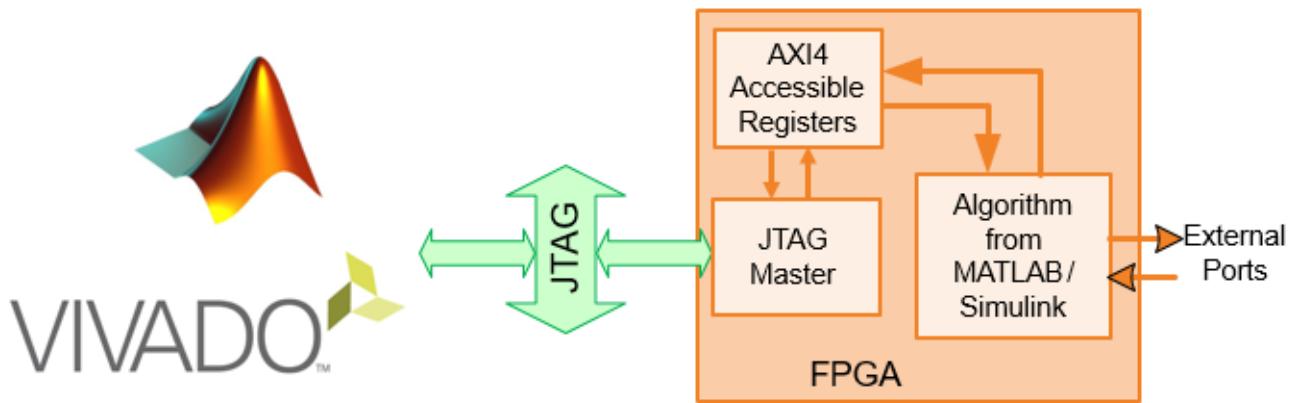
- 1 HDL Verifier™ MATLAB as AXI Master + HDL Coder™ IP Core
- 2 Xilinx JTAG Master + HDL Coder™ IP Core
- 3 MicroBlaze™ + HDL Coder™ IP Core

4 PCIe Endpoint + HDL Coder™ IP Core

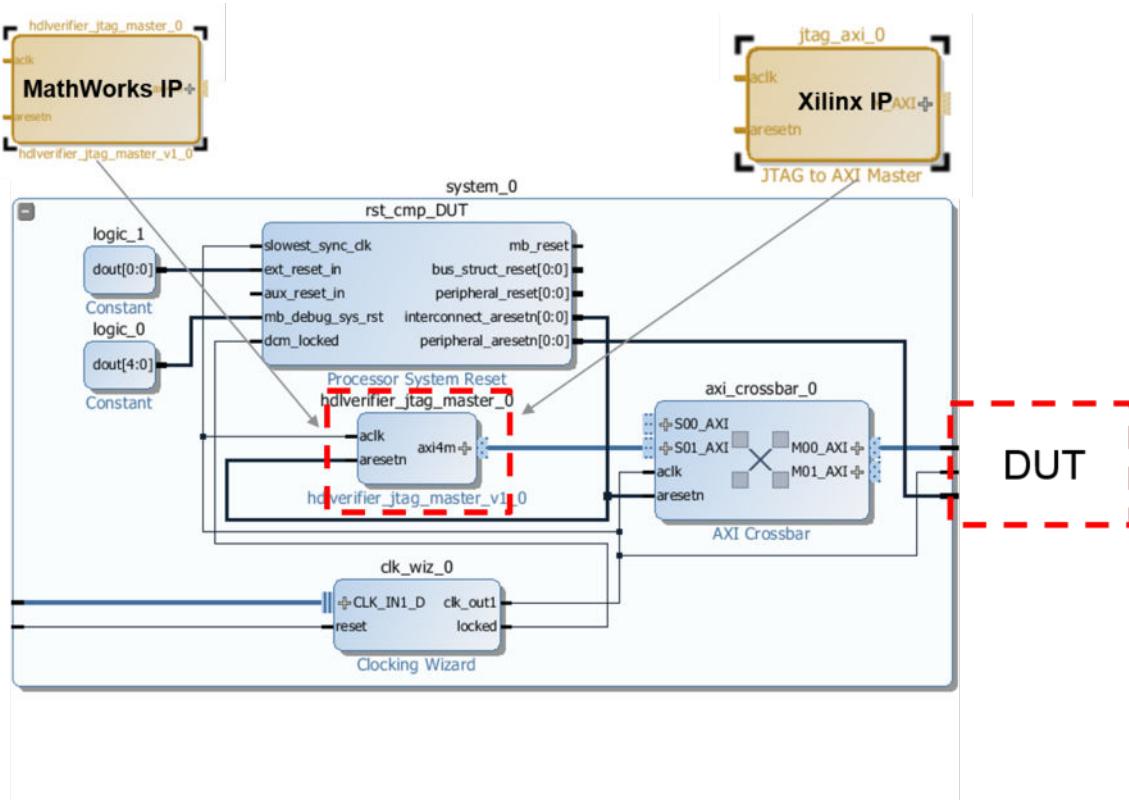
There are two reference designs included in this example:

- The **Default system** reference design uses MathWorks IP and a MATLAB command line interface for issuing read and write commands by enabling the reference design parameter option "Insert JTAG MATAB as AXI Master". Note that to use this parameter, you must have HDL Verifier™ installed.
- The **Xilinx JTAG to AXI Master** reference design uses Vivado IP for the JTAG to AXI Master and therefore requires using the Vivado Tcl console to issue reads and writes.

The two reference designs are nearly identical, except for the JTAG Master IP used in the block diagram shown below:



The reference design, "Xilinx JTAG to AXI Master", uses Vivado™ IP for the JTAG to AXI Master and therefore requires using the Vivado™ Tcl console to issue reads and writes:



1. HDL Verifier™ MATLAB as AXI Master reference design

Specify **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** to **on** in the **Set Target Reference Design** task of IP Core Generation workflow. This adds MATLAB AXI Master IP automatically into the reference design and connects to the DUT IP using AXI4 slave interface. The detailed steps to auto insert the MATLAB JTAG AXI Master in the reference design are discussed in the following section.

Execute the IP Core Workflow

The following instructions in this section applies to **Default System** reference design which uses **MATLAB JTAG as AXI Master**. Using this reference design, you can generate an HDL IP Core that blinks LEDs on the KC705 board.

1. Set up the Xilinx Vivado™ tool path by using the following command:

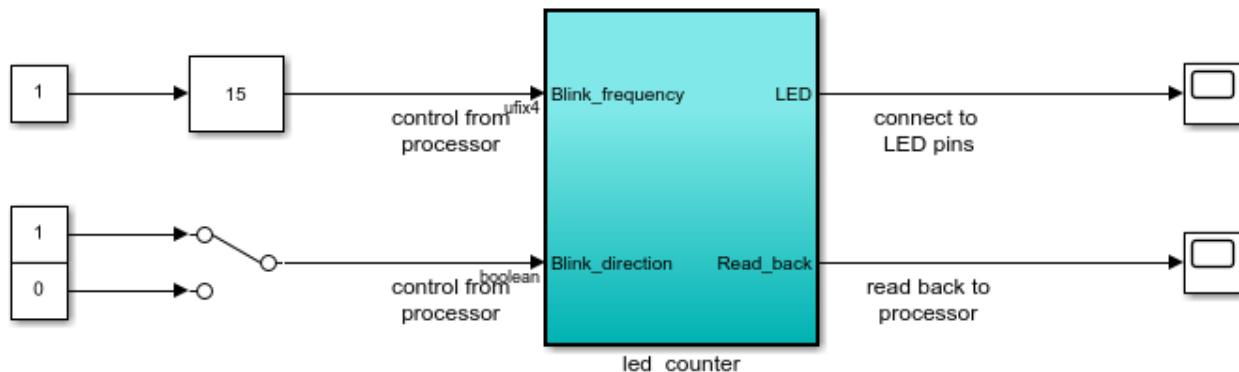
```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat')
```

Use your own Xilinx Vivado™ installation path when executing the command.

2. Open the Simulink model that implements LED blinking using the command:

```
open_system('hdlcoder_led_blinking')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

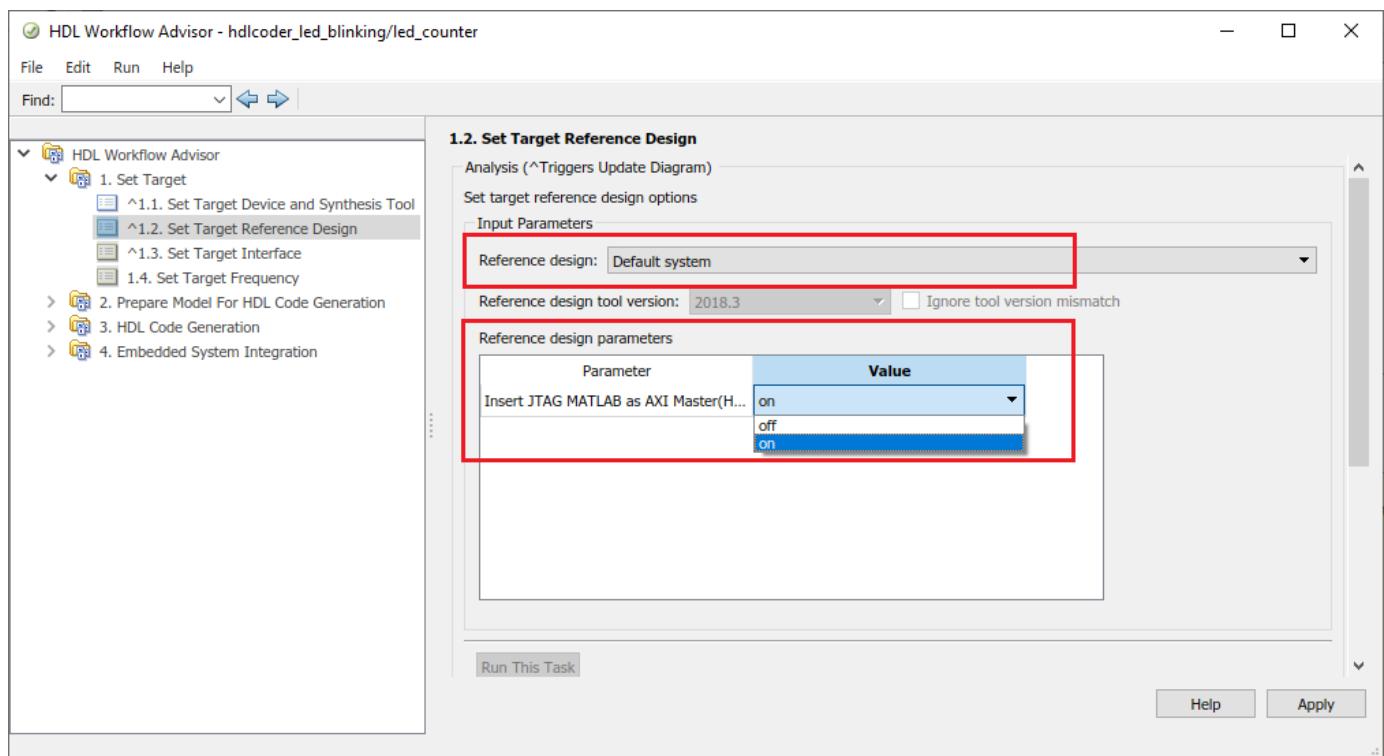
In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

Launch HDL Workflow Advisor

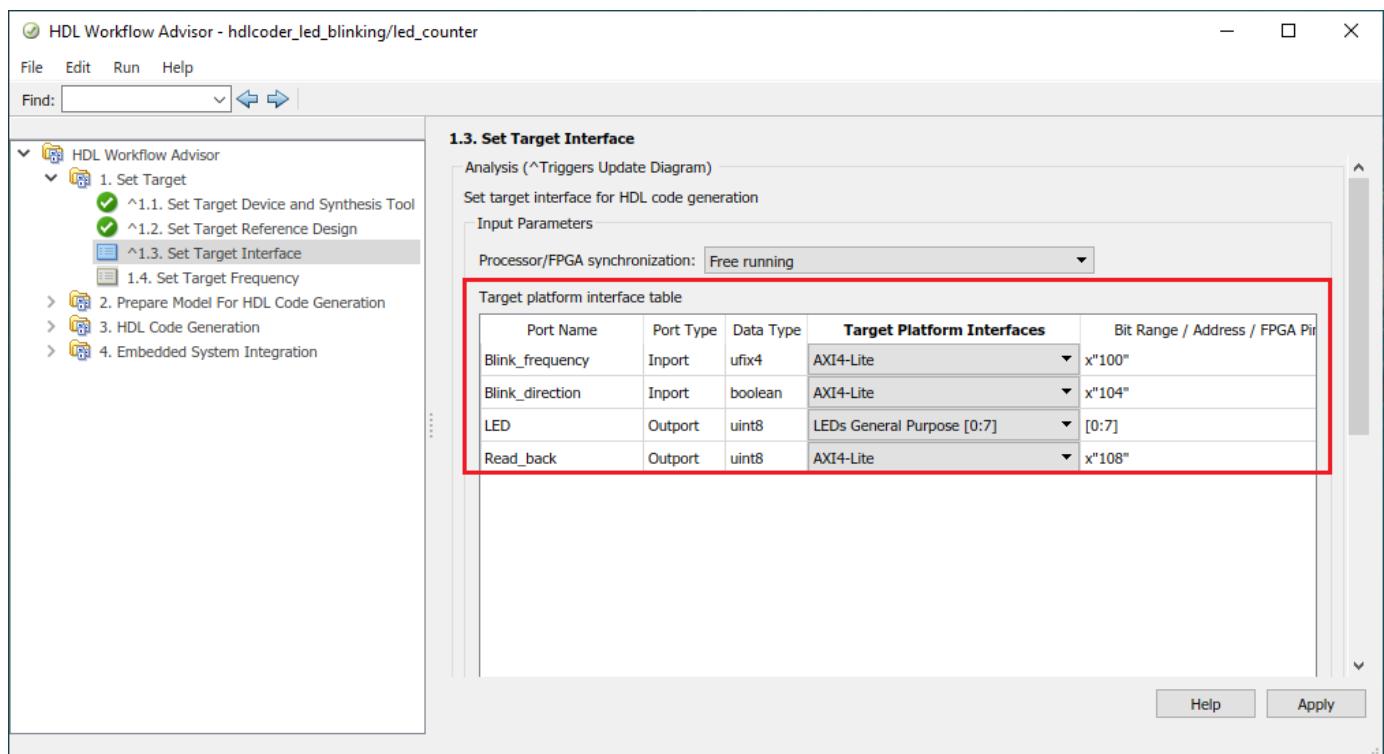
Run Demo

Copyright 2012 The MathWorks, Inc.

3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and selecting **HDL Code > HDL Workflow Advisor**.
4. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**, For **Target platform**, select **Xilinx Kintex-7 KC705** development board and Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task
5. In the **Set Target > Set Target Reference Design** task, Choose **Default System** as reference design and set **Insert JTAG MATLAB as AXI Master** dropdown choice to **on** which is present in the reference design parameter options.

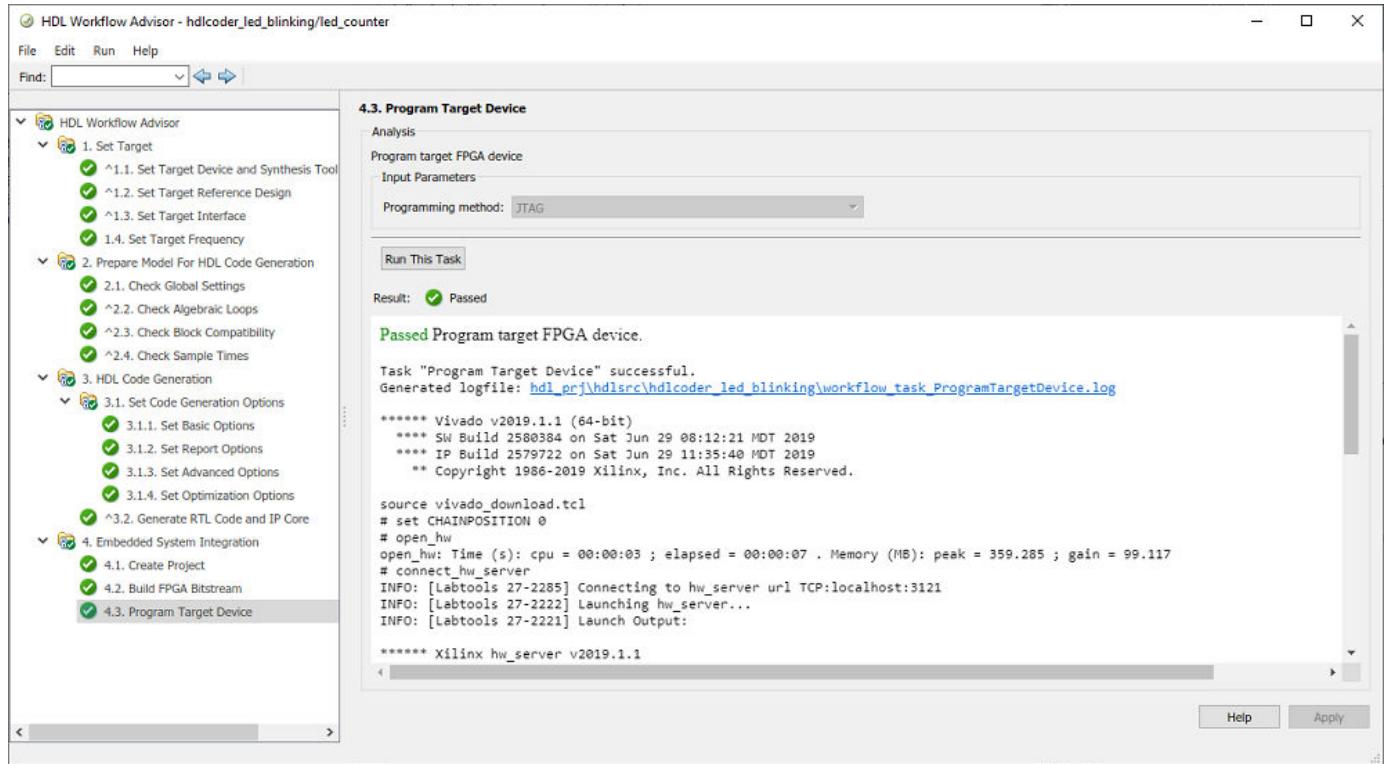


6. In the **Set Target > Set Target Interface** task, choose **AXI4-Lite** for **Blink_frequency**, **Blink_direction**, and **Read_back**. Choose **LEDs General Purpose [0:7]** for **LED**.



7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

Notice that unlike the Zynq-based reference design, there is no **Generate Software Interface Model** task. This is shown in the following figure.



Determining Addresses from the IP Core Report

The base address for an HDL Coder™ IP Core is defined as **0x40000000** for the Default System reference design which uses MATLAB AXI Master IP. You can see this in the generated IP Core report as shown in the following figure.

Code Generation Report

Find: Match Case

HW

FPGA

Use JTAG AXI Master to control the IP core from MATLAB

In 1.2 Step "Set Target Reference design", "Insert JTAG MATLAB as AXI Master" is turned "on". This adds Matlab as an "AXI Master" to control the DUT IP core using AXI4 interface as shown.

The diagram illustrates the connection between MATLAB, MATLAB JTAG AXI Master IP, and DUT IP Core. MATLAB connects to the MATLAB JTAG AXI Master IP via JTAG. The MATLAB JTAG AXI Master IP connects to the DUT IP Core via AXI. The DUT IP Core has LEDs connected to it.

Requires a HDL Verifier license to use this feature. After that use MATLAB® Command line interface to access the DUT IP core registers. The Base Address of AXI4 Slave is **0x40000000**.

The offsets can be found in the IP Core Report Register Address Mapping table:

Register Address Mapping

The following AXI4-Lite bus accessible registers were generated for this IP core:

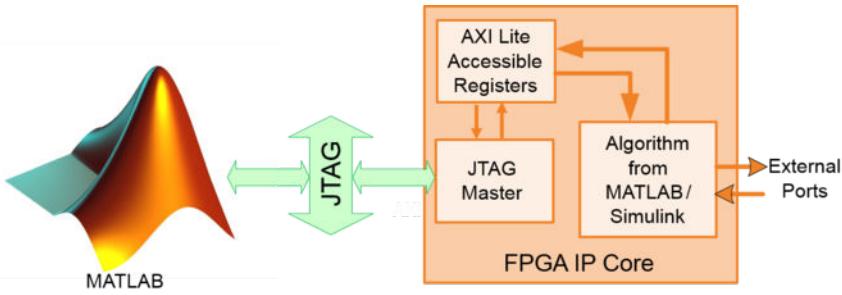
Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
Blink_frequency_Data	0x100	data register for port Blink_frequency
Blink_direction_Data	0x104	data register for port Blink_direction
Read_back_Data	0x108	data register for port Read_back

The register address mapping is also in the following C header file for you to use when programming the processor:
`include/led_count_ip_addr.h`

The IP core name is appended to the register names to avoid name conflicts.

HDL Verifier Command Line Interface

If HDL Verifier support package for Xilinx FPGA boards is installed and the reference design "MATLAB as AXI Master" reference design is selected, then a simple MATLAB command line interface can be used to access the IP core generated by HDL Coder.



1. create the AXI master object

```
h = aximaster('Xilinx')
```

2. Issue a simple write command. For example, to disable the DUT

```
h.writememory('40000004', 0)
```

3. To re-enable the DUT, use the following write command

```
h.writememory('40000004', 1)
```

4. Issue a read command. For example, to read the current counter value

```
h.readmemory('40000108', 1)
```

5. Delete the object when done to free up the JTAG resource. If the object is not deleted, other JTAG operations such as programming the FPGA will fail.

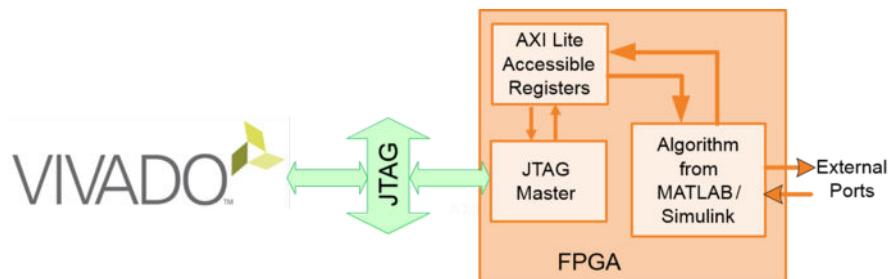
```
delete(h)
```

2. Xilinx JTAG Master reference design

You need to create a custom reference design to use **Xilinx JTAG AXI Master** in reference design and then add the reference design files to the MATLAB path using `addpath` command.

To access the HDL Coder™ IP Core registers using Xilinx JTAG AXI Master, the base address is defined in reference design plugin file.

Vivado Tcl Commands for AXI Read and Write



This example will use the stand alone Vivado Tcl console for the basic commands to issue reads and writes. The following commands can be used to open the JTAG device and setup an 'enable' and 'disable' write to the DUT. These can be entered directly into the Vivado Tcl console or saved in a Tcl file and sourced. For simplicity, copy the following Tcl commands into a file "open_jtag.tcl":

```

# Open connection to the JTAG Master
open_hw
connect_hw_server
open_hw_target
refresh_hw_device [lindex [get_hw_devices] 0]

# Create some reads/writes
create_hw_axi_txn wr_enable [get_hw_axis hw_axi_1] -address 44a0_0004 -data 0000_0001 -type wr
create_hw_axi_txn wr_disable [get_hw_axis hw_axi_1] -address 44a0_0004 -data 0000_0000 -type wr

```

Now launch the Vivado™ Tcl console, sourcing the file you just created:

```
>> system('vivado -mode tcl -source open_jtag.tcl&')
```

```

C:\Windows\system32\cmd.exe - vivado -mode tcl
WARNING: Default location for XILINX_VIVADO_HLS not found:
***** Vivado v2015.4 (64-bit)
***** SW Build 1412921 on Wed Nov 18 09:43:45 MST 2015
***** IP Build 1412160 on Tue Nov 17 13:47:24 MST 2015
** Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.

Vivado>

```

When you are done using the JTAG Master, close the connection using the following Tcl commands:

```

# Close and disconnect from the JTAG Master
close_hw_target;
disconnect_hw_server;

```

Summary

Using a JTAG to AXI Master is a simple way to interface with HDL Coder™ IP core registers in systems which do not have an embedded ARM® processor, such as the Kintex-7. This can be used as first step to debug stand alone HDL Coder™ IP cores, used prior to hand coding software for soft processors, such as MicroBlaze™, or as an easy way to tune parameters on a running system.

IP Core Generation Workflow Without an Embedded ARM Processor: Arrow DECA MAX 10 FPGA Evaluation Kit

This example shows how to use the HDL Coder™ IP Core Generation Workflow to develop reference designs for Intel® parts without an embedded ARM® processor present, but which still utilize the HDL Coder™ generated AXI interface to control the DUT. This example uses MATLAB as AXI Master IP from HDL Verifier™ to access the HDL Coder™ generated DUT registers by enabling the reference design parameter option **Insert JTAG MATLAB as AXI Master**. You can then access DUT registers from MATLAB directly. Alternatively, you can use Intel Qsys (TM) JTAG to Avalon Master Bridge IP to access the FPGA registers using Tcl commands in the Qsys System Console. For Intel JTAG AXI Master, you need to create a custom reference design. The FPGA design is implemented on the Arrow DECA MAX 10 FPGA evaluation kit.

Requirements

- Intel Quartus Prime, with supported version listed in the HDL Coder documentation
- Arrow DECA MAX 10 FPGA evaluation kit
- HDL Coder™ Support Package for Intel FPGA Boards
- HDL Verifier™ Support Package for Intel FPGA Boards (Optional)
- HDL Coder™ Support Package for Intel SoC Devices (Optional: To integrate the IP core into your own custom reference design.)

Arrow DECA MAX 10 FPGA evaluation kit



Example Reference Designs

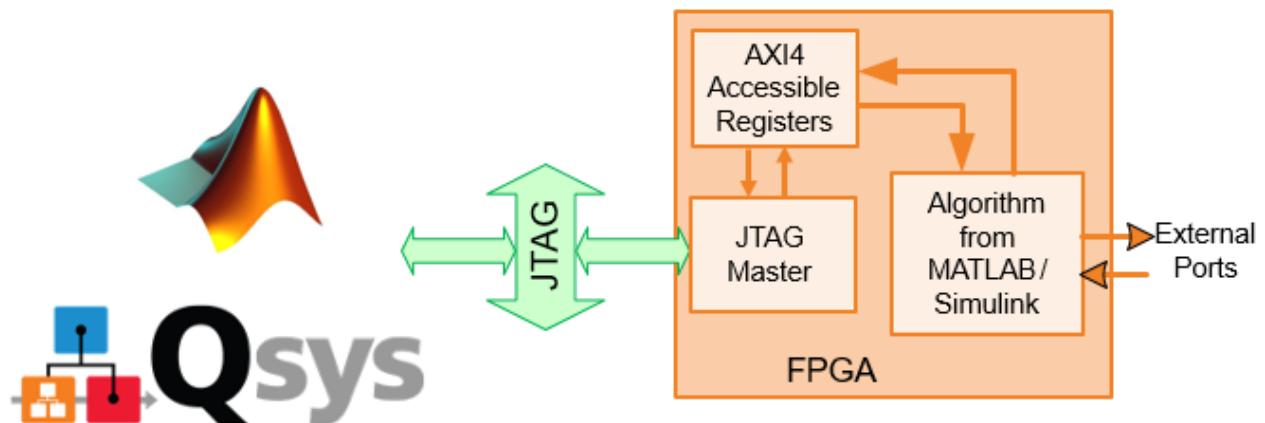
There are many designs which will benefit from using the HDL Coder™ IP Core Generation Workflow without using either an embedded ARM® processor or an Embedded Coder™ Support Package, but which still leverages the HDL Coder™ generated AXI4 registers. These designs include:

- 1 HDL Verifier™ MATLAB as AXI Master + HDL Coder™ IP Core
- 2 JTAG Master + HDL Coder™ IP Core
- 3 Nios® II + HDL Coder™ IP Core
- 4 PCIe® Endpoint + HDL Coder™ IP Core

There are two reference designs included in this example:

- The **Default system** reference design uses MathWorks IP and a MATLAB command line interface for issuing read and write commands by enabling the reference design parameter option "Insert JTAG MATLAB as AXI Master". Note that to use this parameter, you must have HDL Verifier™ installed.
- The **Intel JTAG to AXI Master** reference design uses Quartus IP for the JTAG to AXI Master and therefore requires using the Quartus Tcl console to issue reads and writes.

The two reference designs are nearly identical, except for the JTAG Master IP shown in the block diagram.



The reference design, "Altera JTAG to AXI Master", uses Qsys™ IP for the JTAG to AXI Master and therefore requires using the Intel® System Console to issue reads and writes:

Use	Connections	Name	Description	Export	Clock	Base	End
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	clk_0	Clock Source				
		dk_in	Clock Input				
		dk_in_reset	Reset Input				
		dk	Clock Output	clk	<i>Double-click to export</i>	[dk_in]	
		dk_reset	Reset Output		<i>Double-click to export</i>	dk_0	
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	altpll_0	Avalon ALPLL				
		indk_interface	Clock Input		<i>Double-click to export</i>	clk_0	
		indk_interface_reset	Reset Input		<i>Double-click to export</i>	[indk_interf...	
		pll_slave	Avalon Memory Mapped Slave		<i>Double-click to export</i>	[indk_interf...	
		c0	Clock Output		<i>Double-click to export</i>	altpll_0_c0	
		areset_conduit	Conduit		<i>Double-click to export</i>		
		locked_conduit	Conduit		<i>Double-click to export</i>		
		phasedone_conduit	Conduit		<i>Double-click to export</i>		
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	master_0	JTAG to Avalon Master Bridge				
		dk	Clock Input		<i>Double-click to export</i>	altpll_0_c0	
		dk_reset	Reset Input		<i>Double-click to export</i>		
		master	Avalon Memory Mapped Master		<i>Double-click to export</i>	[dk]	
		master_reset	Reset Output		<i>Double-click to export</i>		
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	led_count_ip_0					
		led_count_ip	Clock Input		<i>Double-click to export</i>	altpll_0_c0	
		ip_dk	Reset Input		<i>Double-click to export</i>	[ip_dk]	
		ip_st	Clock Input		<i>Double-click to export</i>	altpll_0_c0	
		axi_dk	Reset Input		<i>Double-click to export</i>	[axi_dk]	
		axi_reset	AXI4 Slave		<i>Double-click to export</i>	[axi_dk]	
		s_axi	Conduit		<i>Double-click to export</i>	led_count_ip_0_GPLED	
		GPLED				0x0000_0000	0x0000_ffff

1. HDL Verifier™ MATLAB as AXI Master reference design

Specify **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** to **on** in the **Set Target Reference Design** task of IP Core Generation workflow. This adds MATLAB AXI Master IP automatically into the reference design and connects to the DUT IP using AXI4 slave interface. The detailed steps to auto insert the MATLAB JTAG AXI Master in the reference design are discussed in the following section.

Execute the IP Core Workflow

The following instructions in this section applies to Default System reference design which uses MATLAB JTAG as AXI Master. Using this reference design, you can generate an HDL IP Core that blinks LEDs on the DECA board.

1. Set up the Intel Quartus™ tool path. Replace the Quartus™ installation path with your local installation

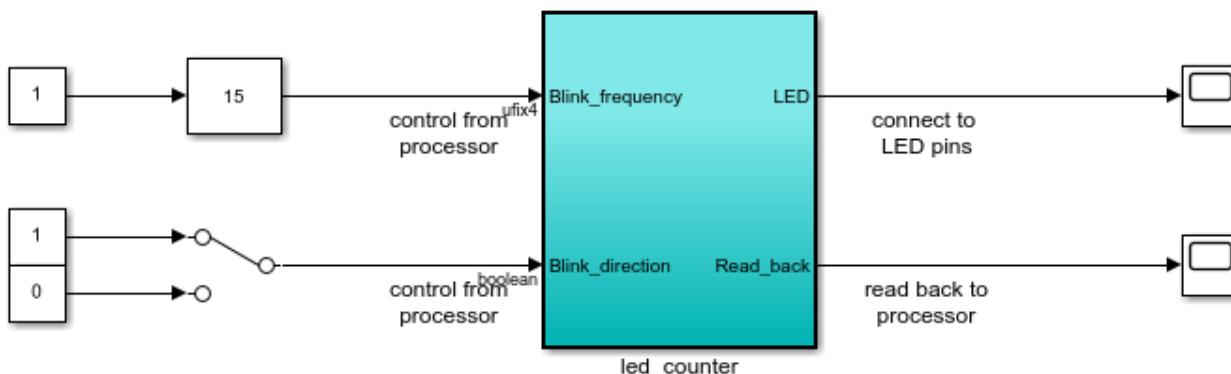
```
hdlsetupoolpath('ToolName', 'Altera QUARTUS II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\q')


```

2. Open the Simulink model that implements LED blinking using the command:

```
open_system('hdlcoder_led_blinking')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

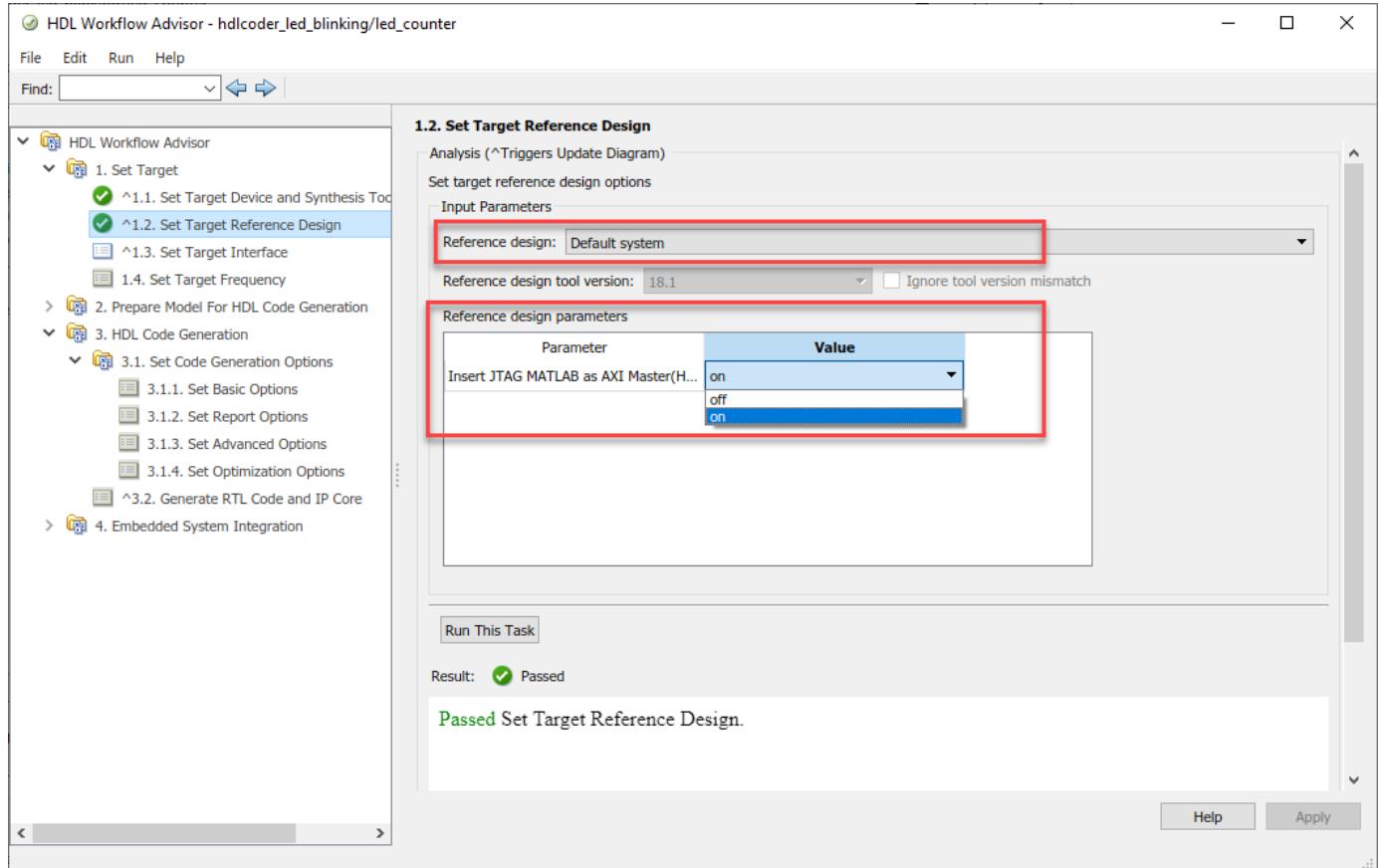
[Launch HDL Workflow Advisor](#)

[Run Demo](#)

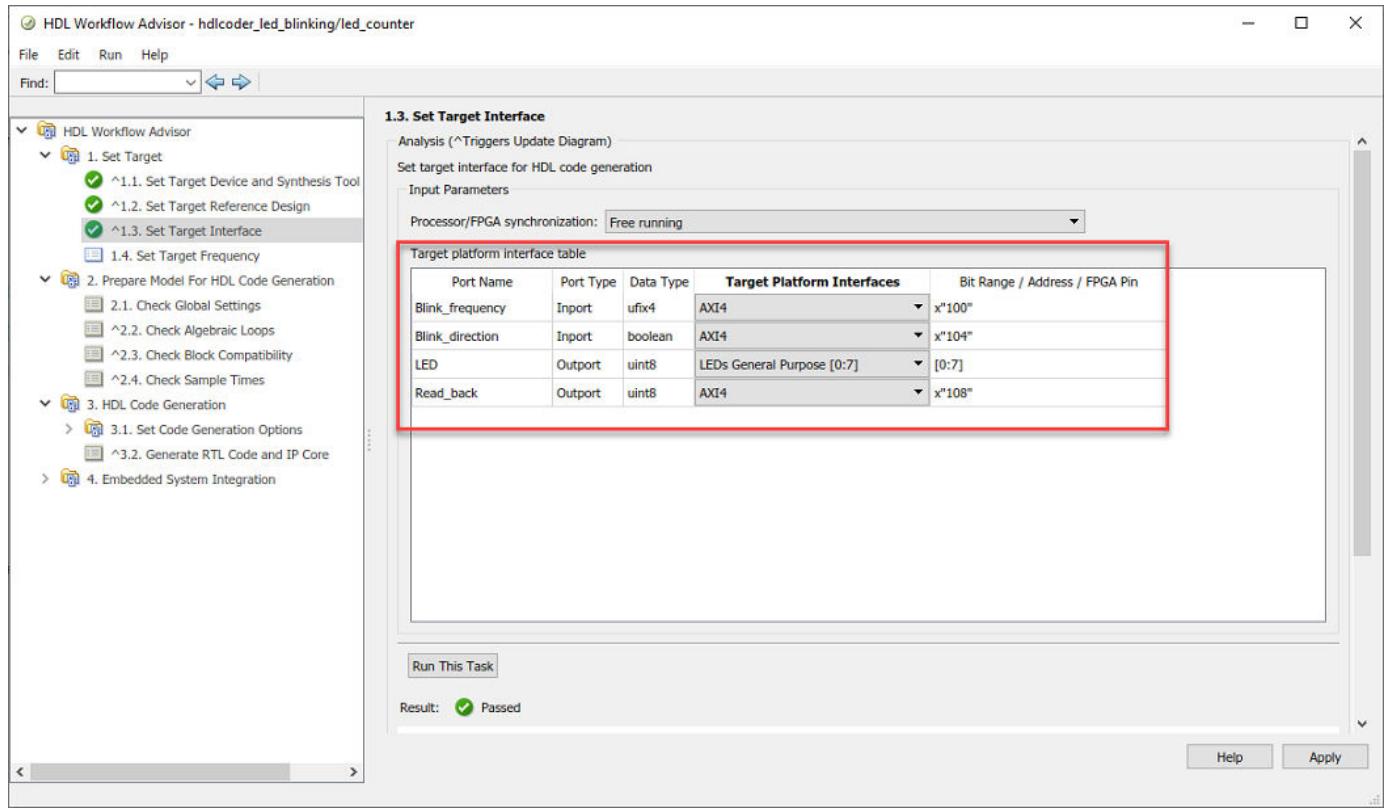
Copyright 2012 The MathWorks, Inc.

3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and selecting **HDL Code > HDL Workflow Advisor**.
4. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select IP Core Generation, For **Target platform**, select Arrow DECA MAX 10 FPGA evaluation kit and Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task

5. In the **Set Target > Set Target Reference Design** task, Choose **Default System** as reference design and set **Insert JTAG MATLAB as AXI Master** dropdown choice to **on** which is present in the reference design parameter options.

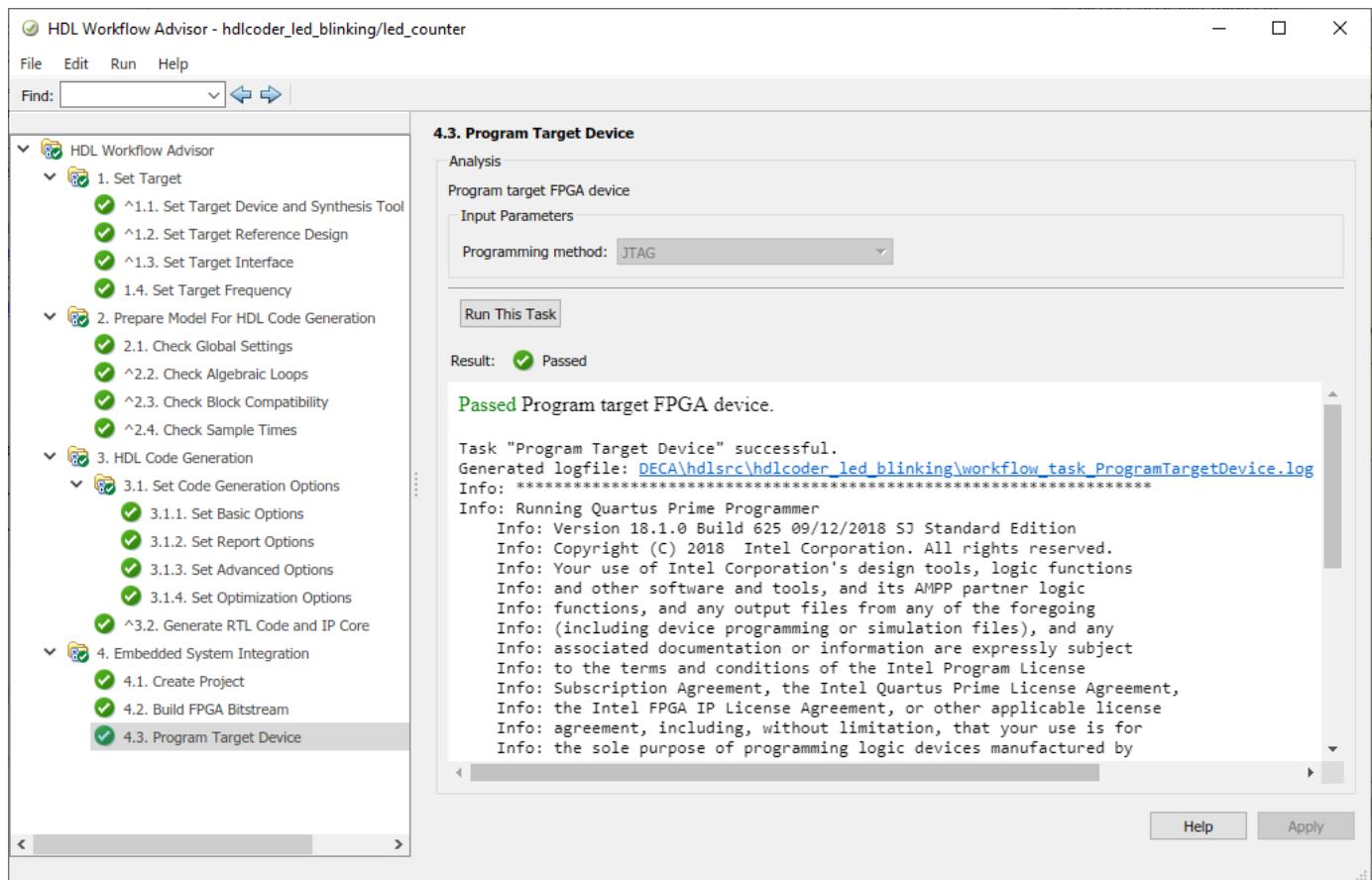


6. In the **Set Target > Set Target Interface** task, choose **AXI4** for **Blink_frequency**, **Blink_direction**, and **Read_back**. Choose **LEDs General Purpose [0:7]** for **LED**.



7. Run the remaining steps in the workflow to generate a bitstream and program the target device.

Notice that unlike the Intel SoC-based reference design, there is no 'Generate Software Interface Model' task. This is shown in the following figure.



Determining Addresses from the IP Core Report

The Base Address for an HDL Coder™ IP Core is defined as **0x00000000** for the Default System reference design which uses MATLAB AXI Master IP. You can see this in the generated IP Core report as shown in the following figure.

Code Generation Report

Find: Match Case

Contents

- Summary
- Clock Summary
- Code Interface Report
- Timing And Area Report
 - [High-level Resource Report](#)
- Optimization Report
 - [Distributed Pipelining](#)
 - [Streaming and Sharing](#)
 - [Delay Balancing](#)
 - [Adaptive Pipelining](#)
- IP Core Generation Report**
- Traceability Report

Generated Source Files

- [led_count_ip_src_led_counter_pk](#)
- [led_count_ip_src_led_counter.vhd](#)

Referenced Models

FPGA

HW

Use JTAG AXI Master to control the IP core from MATLAB

In 1.2 Step "Set Target Reference design", "Insert JTAG MATLAB as AXI Master" is turned "on". This adds Matlab as an "AXI Master" to control the DUT IP core using AXI4 interface as shown.

The diagram illustrates the connection between MATLAB, JTAG AXI Master IP, and DUT IP Core. MATLAB is connected to JTAG AXI Master IP via a green double-headed arrow labeled "JTAG". JTAG AXI Master IP is connected to DUT IP Core via a red double-headed arrow labeled "AXI". Both connections pass through a central "Reference Design" area.

Requires a HDL Verifier license to use this feature. After that use MATLAB® Command line interface to access the DUT IP core registers. The Base Address of AXI4 Slave is 0x0000 0000 .

OK Help

The offsets can be found in the IP Core Report Register Address Mapping table:

Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

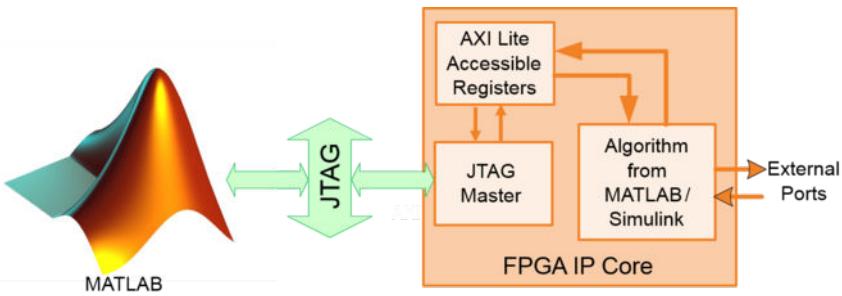
Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
Blink_frequency_Data	0x100	data register for port Blink_frequency
Blink_direction_Data	0x104	data register for port Blink_direction
Read_back_Data	0x108	data register for port Read_back

The register address mapping is also in the following C header file for you to use when programming the processor:
[include\led_count_ip_addr.h](#)

The IP core name is appended to the register names to avoid name conflicts.

HDL Verifier Command Line Interface

If HDL Verifier support package for Intel FPGA boards is installed and the reference design "MATLAB as AXI Master" reference design is selected, then a simple MATLAB command line interface can be used to access the IP core generated by HDL Coder.



1. Create the AXI master object

```
h = aximaster('Altera')
```

2. Issue a simple write commands. For example, to disable the DUT

```
h.writememory('4', 0)
```

3. To re-enable the DUT, use the following write command

```
h.writememory('4', 1)
```

4. To read the current counter value

```
h.readmemory('108', 1)
```

5. Delete the object to free up the JTAG resource. If the object is not deleted, other JTAG operations such as programming the FPGA will fail.

```
delete(h)
```

Intel JTAG AXI Master reference design

You need to create a custom reference design to use **Intel JTAG AXI Master** in reference design and then add reference design files to the MATLAB path using `addpath` command.

To access the HDL Coder™ IP Core registers using Intel JTAG AXI Master, the base address is defined in reference design plugin file.

Qsys System Console Tcl Commands for AXI Read and Write

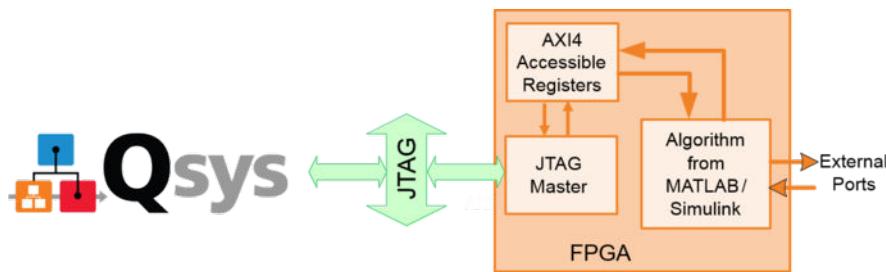
Before we open a System Console, lets look at the basic commands to issue reads and writes. There are a number of flavors of Qsys read and write methods, but we will use the following since all HDL Coder™ generated IP Core registers are currently 32-bits:

```
% master_write_32 <service-path> <start-address> <list-of-32-bit-values>
% master_read_32 <service-path> <start-address> <size-in-multiples-of-32-bits>
```

For example, assume we would like to write the 32 bit hex value '0x12345678' to the IP Core register defined by offset '0x100' using a previously defined service path stored in the variable `$jtag`:

```
% master_write_32 $jtag 0x100 0x12345678
```

Before you can generate reads and writes, you must first launch a System Console and open a connection to the JTAG Master that will issue the register reads and writes. Refer again to the system diagram below:



To open a connection to JTAG Master, first set a variable that stores the service path (in this case, there is only one master):

```
% set jtag [lindex [get_service_paths master] 0]
```

Then use the variable to open the JTAG Master in master mode.

```
% open_service master $jtag
```

Now launch the Altera® System Console and enter the commands to open the jtag master:

```
>> system('C:\intelFPGA\17.1\quartus\sopc_builder\bin\system-console&')
```

The screenshot shows the "Tcl Console" window with the following command history:

```
Tcl Console
%
% set jtag [lindex [get_service_paths master] 0]
/devices/10M50DA(.|ES)|10M50DC@1#USB-1#Arrow MAX 10 DECA/(link)/JTAG/(110:132 v1 #0)/phy_0/master
% open_service master $jtag

% master_write_32 $jtag 0x04 0x00
% master_write_32 $jtag 0x04 0x01
% master_read_32 $jtag 0x108 1
0x000000f0
% close_service master $jtag

%
```

When you are done using the JTAG Master, make sure to close the connection using the following Tcl command:

```
close_service master $jtag
```

Summary

Using a JTAG to AXI Master is a simple way to interface with HDL Coder™ IP core registers in systems which do not have an embedded ARM® processor, such as the MAX 10. This can be used as

first step to debug stand-alone HDL Coder™ IP cores, used prior to hand-coding software for soft processors, such as Nios® II, or as an easy way to tune parameters on a running system.

IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705

This example shows how to use the HDL Coder™ IP Core Generation Workflow to develop reference designs for Xilinx® parts without an embedded ARM® processor present, but which still utilize the HDL Coder™ generated AXI interface to control the DUT. Specifically, this example will use the Xilinx Kintex-7 KC705 board and a MicroBlaze™ soft processor running a LightWeightIP (lwIP) based TCP/IP firmware server in the reference design to access the HDL Coder™ generated DUT registers from anywhere on the connected network. Further, this example will also highlight the difference between accessing data from a collection of registers implemented as multiple scalar ports and a collection of registers implemented as a single vector port.

Requirements

- Xilinx Vivado Design Suite, with supported version listed in the HDL Coder documentation
- Xilinx Kintex-7 KC705 development board
- HDL Coder™ support package for Xilinx FPGA Boards
- Ethernet connection

Xilinx Kintex-7 KC705 development board



Example Reference Designs

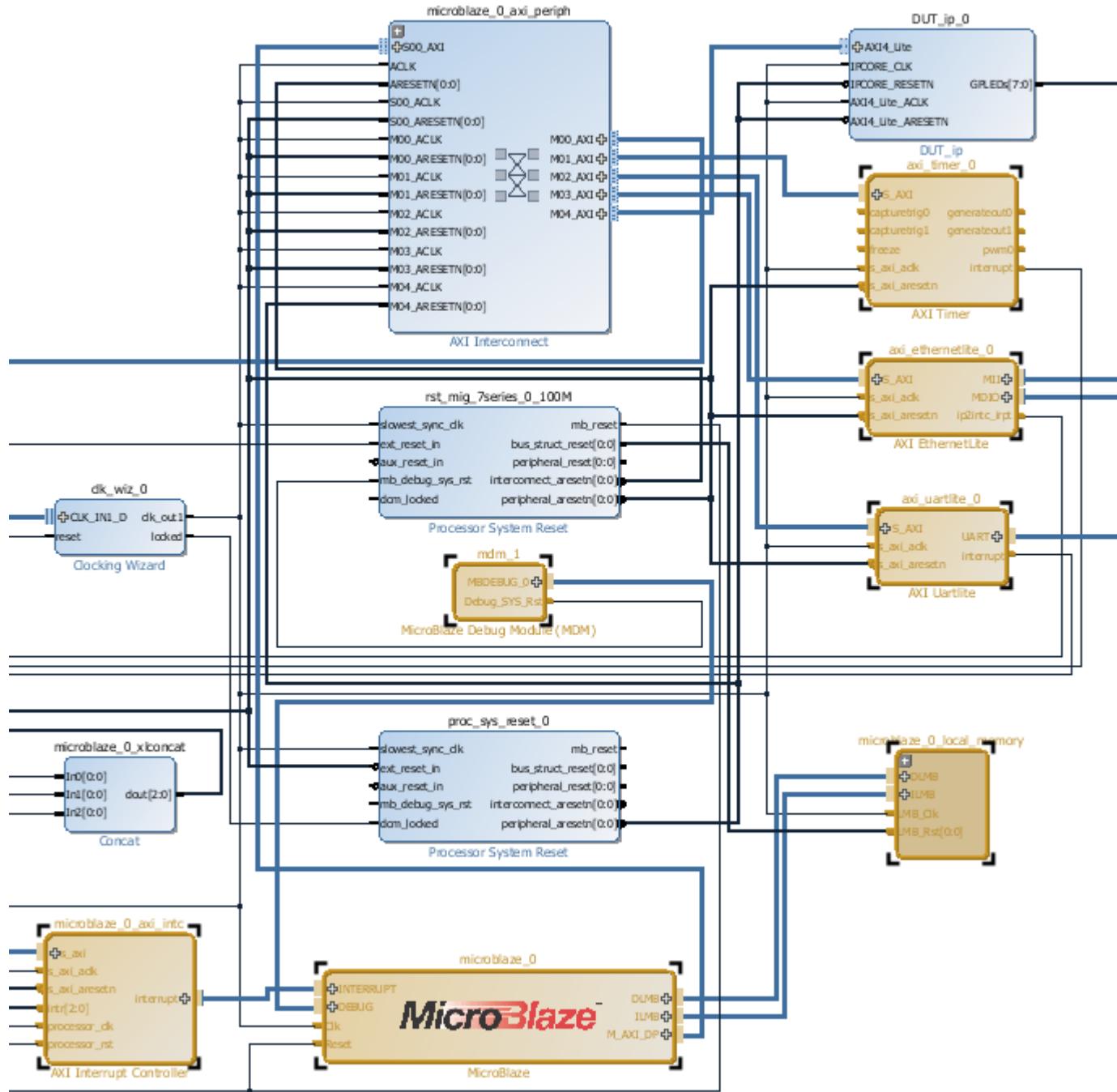
MicroBlaze is a simple, versatile soft-core processor that can be used in Xilinx FPGA only platforms, such as the Kintex-7, to perform the functionality a full-fledged processor, or as a flexible, programmable IP. When programs are small, the ELF can sit in BRAM and the design becomes completely self contained in the FPGA. There are many applications well suited to being implemented on a MicroBlaze. Here we list just a few:

- 1 Remote networked control of a deployed IP Core algorithm
- 2 Embedded web server for control and data display
- 3 Integration of existing software algorithms to hardware-only platforms

The following is a system level diagram for this MicroBlaze system:



The reference design, "Xilinx MicroBlaze TCP/IP to AXI4-Lite Master", uses Vivado™ MicroBlaze IP to translate TCP/IP packets into AXI4-Lite reads and writes. Below is a block diagram of the complete system, including all the peripherals required to operate the TCP/IP server and debug via the UART serial console.



MicroBlaze Setup

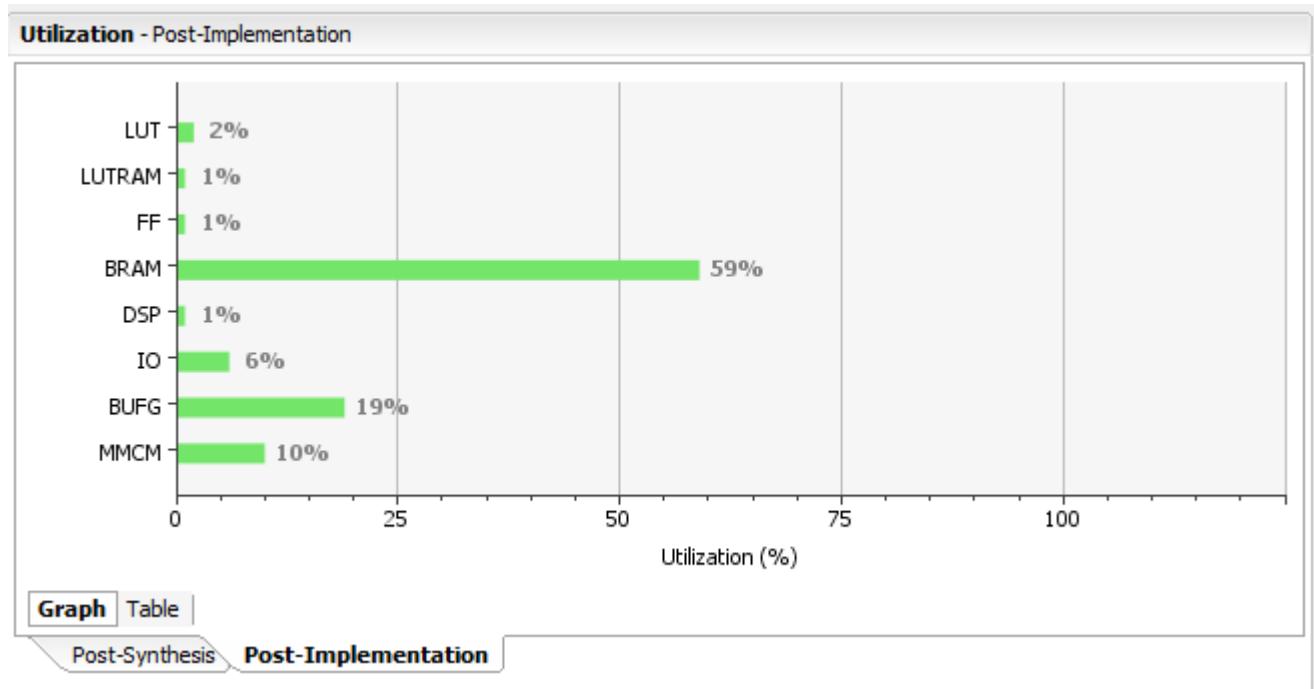
In order to operate the TCP/IP server, the MicroBlaze IP needs a few basic peripherals:

- local memory (BRAM) for data/instructions
- Ethernet core for transmitting and receiving frames
- UART core for sending debug messages
- Timer core for generating timeout interrupts
- Interrupt controller for handling interrupts from all these peripherals.

As can be seen in the address editor below, all these peripherals are connected to the MicroBlaze via AXI4 interfaces.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
axi_ethernetlite_0	S_AXI	Reg	0x40E0_0000	64K	0x40E0_FFFF
axi_timer_0	S_AXI	Reg	0x41C0_0000	64K	0x41C0_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	1M	0x000F_FFFF
microblaze_0_axi_intc	s_axi	Reg	0x4120_0000	64K	0x4120_FFFF
led_count_ip_0	AXI4_Lite	reg0	0x44A0_0000	64K	0x44A0_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	1M	0x000F_FFFF

The amount of local memory allocated using BRAM is 1MB. This amount is needed to run the lwIP stack. The benefit of specifying BRAM for local memory is that the executable ELF can be included in the bitstream, which simplifies programming and enables targeting existing FPGA boards which may not have external DRAM memory. However, this convenience comes at a cost of increased utilization:



If BRAM utilization is a concern and DRAM resources are available, you may opt to replace local BRAM memory with external DRAM memory. See Xilinx app note "xapp1026" for more details on alternate configurations and application information.

Example Reference Design plugin_rd.m

The plugin_rd.m for this reference design is shown below:

```
function hRD = plugin_rd()
% Reference design definition

% Copyright 2014-2018 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Xilinx MicroBlaze TCP/IP to AXI4-Lite Master';
hRD.BoardName = 'Xilinx Kintex-7 KC705 development board';

% Tool information
hRD.SupportedToolVersion = {'2017.2','2017.4'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl',...
    'VivadoBoardPart',      'xilinx.com:kc705:part0:1.1');

% add custom files, use relative path
hRD.CustomFiles        = {'mw_lwip_tcpip_axi4.elf'};

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',      'clk_wiz_0/clk_out1', ...
    'ResetConnection',      'proc_sys_reset_0/peripheral_aresetn', ...
    'DefaultFrequencyMHz', 100, ...
    'MinFrequencyMHz',     100, ...
    'MaxFrequencyMHz',     100, ...
    'ClockNumber',          1, ...
    'ClockModuleInstance',  'clk_wiz_0');

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'microblaze_0_axi_periph/M04_AXI', ...
    'BaseAddress',         '0x44A00000', ...
    'MasterAddressSpace',  'microblaze_0/Data', ...
    'InterfaceType',       'AXI4-Lite', ...
    'InterfaceID',         'MicroBlaze AXI4-Lite Interface');

hRD.HasProcessingSystem = false; % No hard processing system
```

Note that the reference design includes the MicroBlaze executable `mw_lwip_tcpip_axi4.elf` in the reference design property `CustomFiles`. This will copy the executable from the reference design to the Vivado project so that it can be associated with the MicroBlaze IP.

Additional code in 'system_top.tcl' to attach ELF to uBlaze

The 'system_top.tcl' file included in most reference designs is used to create the top level Vivado IP Integrator block diagram containing most of the reference design IP. Here we are adding some

additional Tcl code to this file after the block diagram has been created to associate the standalone MicroBlaze ELF executable with the MicroBlaze IP.

```
import_files -norecurse mw_lwip_tcpip_axi4.elf
generate_target all [get_files system_top.bd]
set_property SCOPED_TO_REF system_top [get_files -all -of_objects [get_fileset sources_1] {mw_lwip_tcpip_axi4.elf}]
set_property SCOPED_TO_CELLS { microblaze_0 } [get_files -all -of_objects [get_fileset sources_1] { system_top.bd }]
```

Doing this allows the ELF to be packaged with the bitstream and programmed into the MicroBlaze BRAM memory at the same time as the FPGA.

Execute the IP Core Workflow

Using the above reference design you will generate an HDL IP Core that blinks LEDs on the KC705 board. You will then use `tcpclient` to send/receive formatted packets to the MicroBlaze to issue reads/writes over the AXI4-Lite interface to the generated HDL IP Core. The files used in the following demonstration are located at:

- `matlab/toolbox/hdlcoder/hdlcoderdemos/customboards/KC705`

1. Add the MicroBlaze reference design files to the MATLAB path using the command:

```
>> addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','KC705'));
```

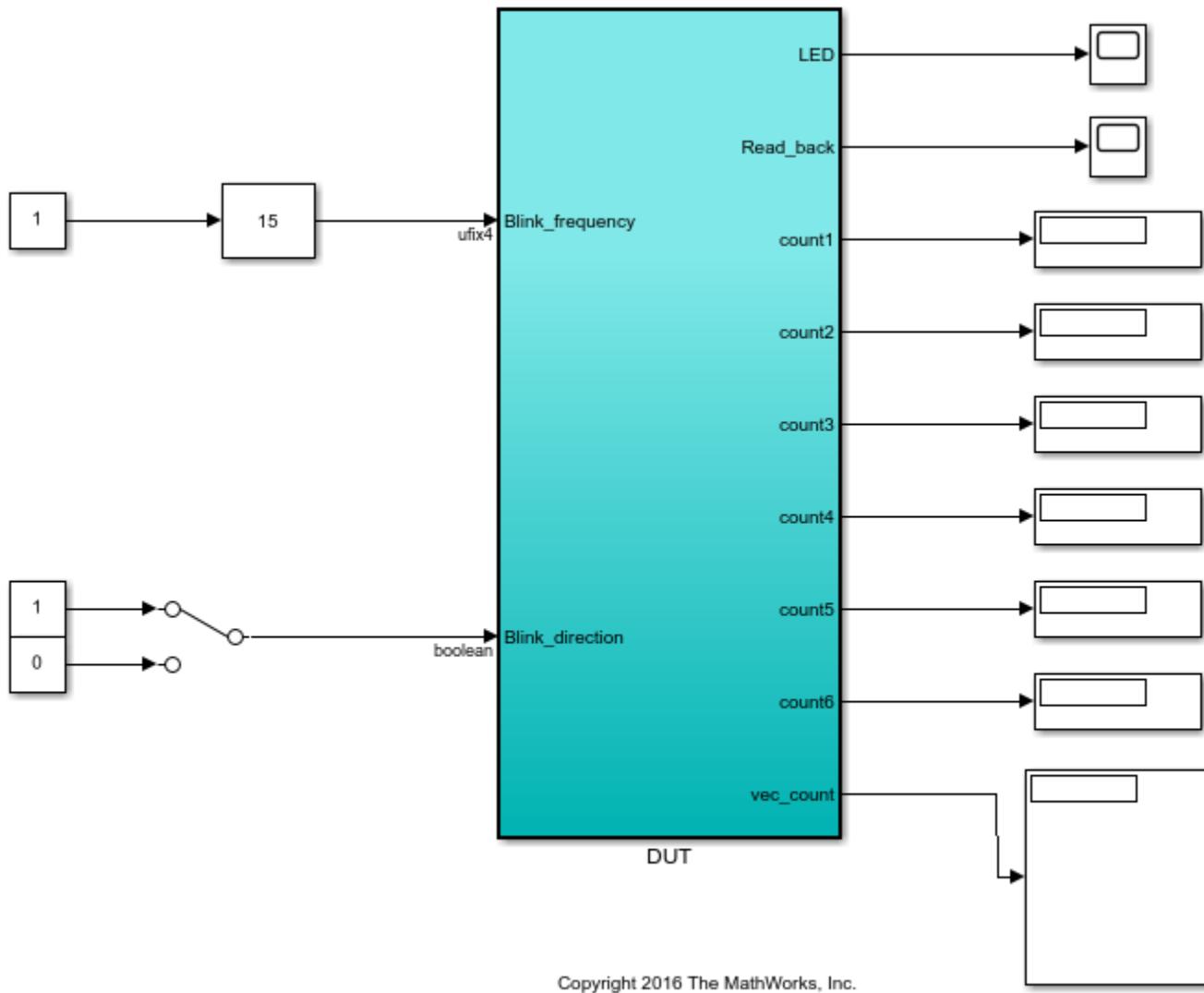
2. Set up the Xilinx Vivado™ tool path by using the following command:

```
>> hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2017.4\bin\vivado')
```

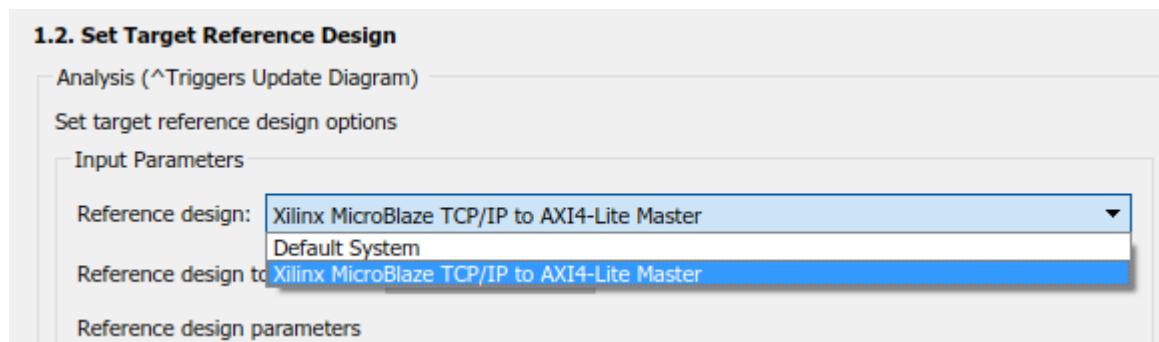
Use your own Xilinx Vivado™ installation path when executing the command.

3. Open the Simulink model that implements LED blinking, as well as vector output ports for comparison to scalar ports, using the command:

```
open_system('hdlcoder_led_vector')
```



4. Launch HDL Workflow Advisor from the `hdlcoder_led_vector/DUT` subsystem by right-clicking the DUT subsystem, and selecting **HDL Code > HDL Workflow Advisor**.
5. Select reference design from the drop down in step 1.2



- 6.** Assign register ports to the "MicroBlaze AXI4-Lite Interface". These will then be accessible at the hex offset shown in the table.

1.3. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
Blink_frequency	Import	ufix4	MicroBlaze AXI4-Lite Interface	x"100"
Blink_direction	Import	boolean	MicroBlaze AXI4-Lite Interface	x"104"
LED	Outport	uint8	LEDs General Purpose [0:7]	[0:7]
Read_back	Outport	uint8	MicroBlaze AXI4-Lite Interface	x"108"
count1	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"10C"
count2	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"110"
count3	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"114"
count4	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"118"
count5	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"11C"
count6	Outport	uint32	MicroBlaze AXI4-Lite Interface	x"120"
vec_count	Outport	uint32 (6)	MicroBlaze AXI4-Lite Interface	x"140"

- 7.** Run the remaining steps in the workflow to generate a bitstream and program the target device.

Determining Addresses from the IP Core Report

The Base Address for an HDL Coder™ IP Core is defined in the reference design plugin_rd.m with the following command:

```
% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'microblaze_0_axi_periph/M04_AXI', ...
    'BaseAddress', '0x44A0_0000',...
    'MasterAddressSpace', 'microblaze_0/Data',...
    'InterfaceType', 'AXI4-Lite',...
    'InterfaceID', 'MicroBlaze AXI4-Lite Interface');
```

For this design, the base address is `0x44A0_0000`. The offsets can be found in the IP Core Report Register Address Mapping table:

Register Address Mapping

The following AXI4-Lite bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
Blink_frequency_Data	0x100	data register for Import Blink_frequency
Blink_direction_Data	0x104	data register for Import Blink_direction
Read_back_Data	0x108	data register for Outport Read_back
count1_Data	0x10C	data register for Outport count1
count2_Data	0x110	data register for Outport count2
count3_Data	0x114	data register for Outport count3
count4_Data	0x118	data register for Outport count4
count5_Data	0x11C	data register for Outport count5
count6_Data	0x120	data register for Outport count6
vec_count_Data	0x140	data register for Outport vec_count, vector with 6 elements, address ends at 0x154
vec_count_Strobe	0x160	strobe register for port vec_count

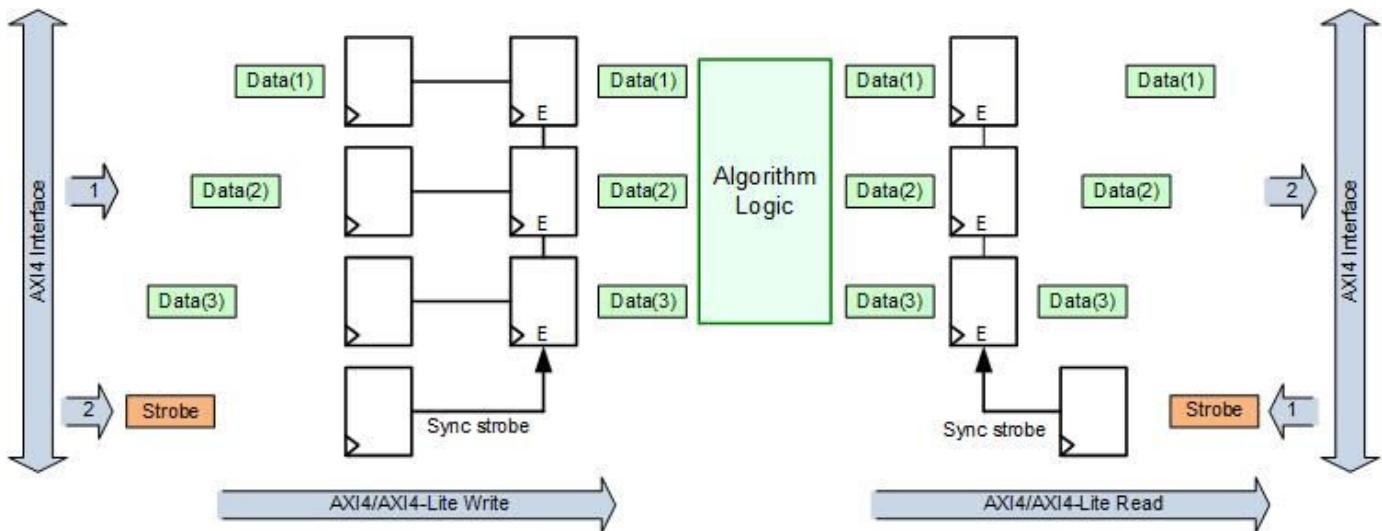
The register address mapping is also in the following C header file for you to use when programming the processor:
[include\DUT_ip_addr.h](#)

The IP core name is appended to the register names to avoid name conflicts.

Vector Data Read/Write with Strobe Synchronization

Vector data is supported on the AXI4/AXI4-Lite interfaces as of R2017a. Unlike a collection of scalar ports, all the elements of vector data are treated as synchronous to the IP Core algorithm logic. Additional strobe registers added for each vector input and output port maintain this synchronization across multiple sequential AXI4 reads/writes.

For input ports, the strobe register controls the enables on a set of shadow registers, allowing the IP core logic to see all the updated vector elements simultaneously. For output ports, the strobe register controls the synchronous capturing of vector data to be read. Below is a diagram of the synchronization logic generated with vector data:

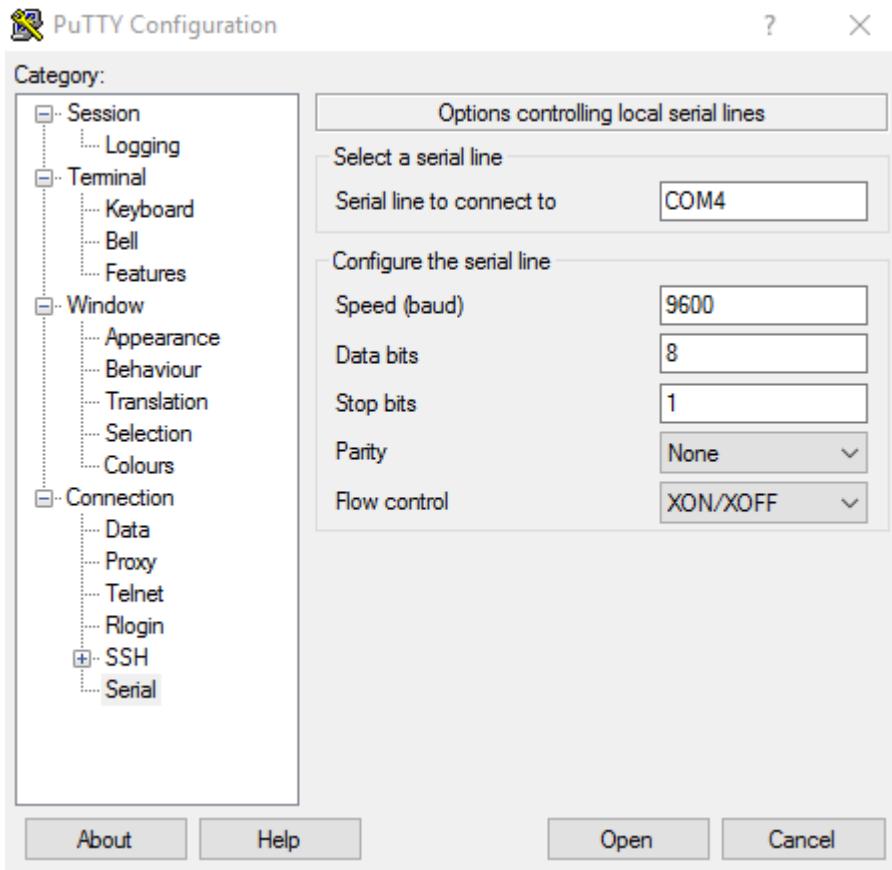


Connect to the TCP/IP server

To start interacting with the TCP/IP server running on the MicroBlaze, first connect the UART serial console to view debug messages, which will help ensure things are working as expected. First find the serial port that is connected to the UART on the board:



Then use this port to connect using a program such as PuTTY™:



Once connected to the UART serial console, run the `hdlworkflow_ProgramTargetDevice.m` script to reprogram the board.

```
>>hdlworkflow_ProgramTargetDevice
### Workflow begin.
### Loading settings from model.
### ++++++ Task Program Target Device ++++++
### Generated logfile: hdl_prj\hdlsrc\hdlcoder_led_vector\workflow_task_ProgramTargetDevice.log
### Task "Program Target Device" successful.
### Workflow complete.
```

In the console window, you should see the following header, displaying the IP address and port number the server is connected to.

```

-----MathWorks HDL Coder AXI4-Lite IP Core Read/Write Server -----
TCP packets sent to port 7 will be issued as AXI4-Lite Read/Writes

[ 32-bit address ] (Base Address = 0x44a0_0000)
[ 32-bit cmd ] (read = 0x00, write = 0x01, debug = 0x03)
[ 32-bit len ] ( N<255)
[ 32-bit data ] (N 32-bit data values for write cmd)

auto-negotiated link speed: 100
DHCP Timeout
Configuring default IP of 192.168.1.10
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
TCP AXI4-Lite server started @ port 7

```

NOTE: If the board is connected to a network with a DHCP server enabled, the IP information will be different than shown above. In this case, you will need to modify line 43 of the `read_write_test.m` script to connect to the correct IP address of the board:

```
t = tcpclient('192.168.1.10',7);
```

Sending AXI4-Lite transactions to the MicroBlaze from MATLAB using `tcpipclient`

In order to issue reads and writes to the IP Core via TCP/IP and AXI4-Lite, the address,data and command to be performed must be encoded in the packet sent to the TCP/IP server. For this example, we use the following packet format:

```

[--Address--] 32-bits
[----Cmd----] lower byte of 32-bit word (READ = 0, WRITE = 1, DEBUG = 2)
[---Length---] lower byte of 32-bit work (N<255)
[----Data---] 32-bits, used only for WRITE cmd

```

For example, a packet issuing a read of 3 consecutive values starting at address 0x44a0010c:

```
[44 a0 01 0c]
[00 00 00 00]
[00 00 00 03]
```

This will return data at offsets 0x10c,0x110,0x114.

For example, a packet issuing a write of 0x0 to offset 0x04, would be:

```
[44 a0 00 04]
[00 00 00 01]
[00 00 00 01]
[00 00 00 00]
```

To change the debug level used to print to the console, send a debug cmd packet:

```
[xx xx xx xx]
[00 00 00 03]
[00 00 00 01] %0 = no msg, 1 = READ|WRITE, 2 = full pkt
```

Run the `read_write_test.m` script

This example includes a script which will setup a connection to the TCP/IP server running on the MicroBlaze, create commands to enable/disable the DUT, read 6 scalar ports and the same data as a vector port and compare the results.

1. To run this script, first copy it to your local directory

```
>> copyfile(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','ublaze_lwip_read_write_vec'))
```

and open the script in the editor:

```
>> edit('ublaze_test.m');
```

The script has three sections. The first section, connects to the board and sets up the commands that will be used. If required, update the IP address of the board on line 41.

2. Execute section 1. You will have generated the following commands as arrays of uint32 types:

```
read6_cmd      = uint32([hex2dec('44a0010c') 0 6]);    %read 6 32-bit regs
read_vec_cmd   = uint32([hex2dec('44a00140') 0 6]);    %read 6 elements of vec
strobe_vec_cmd = uint32([hex2dec('44a00160') 1 1 1]); %write strobe for vec
enable_cmd     = uint32([hex2dec('44a00004') 1 1 1]); %enable ip core
disable_cmd    = uint32([hex2dec('44a00004') 1 1 0]); %disable ip core
debug0_cmd     = uint32([hex2dec('00000000') 3 0]);    %disable all debug printf
debug1_cmd     = uint32([hex2dec('00000000') 3 1]);    %enable READ|WRITE printf
debug2_cmd     = uint32([hex2dec('00000000') 3 2]);    %enable pkt printf
```

NOTE: these arrays store data in the endian format used by the local machine, which for many x86 systems is the little endian format. However, the TCP/IP server expects values in the big endian format (network byte order). As a result, if the system you are on is little endian, the bytes in each element must be swapped using `swapbytes`.

3. Execute section 2 to disable the DUT logic and read a single counter value connected to the 6 scalar ports as well as all 6 elements in the vector port. Notice that all the counter values match. This is because the same data is driven to all the ports and the DUT is disabled, so the asynchronous access across the AXI4 interface is not apparent.

```
Scalar port (top) vs vector port (bottom) access with DUT disabled:
7e8aec14    7e8aec14    7e8aec14    7e8aec14    7e8aec14    7e8aec14
7e8aec14    7e8aec14    7e8aec14    7e8aec14    7e8aec14    7e8aec14
```

- 4 . Execute section 3 to re-enable the DUT logic and read the same counter values back. Notice that the 6 scalar ports all show different values, while the 6 elements of the vector port are all the same. This is due to the sequential access that must occur across the AXI4 interface and the lack of synchronization register in the scalar port case and the presence of an explicit synchronization register in the vector port case.

```
Scalar port (top) vs vector port (bottom) access with DUT enabled:
7f7796dc    7fc70e4b    8016860a    8065fdce    80b5758d    8104ed4d
815964dd    815964dd    815964dd    815964dd    815964dd    815964dd
```

The corresponding debug output on the serial console will be:

```
DEBUG | packet payload:
44 A0 00 04
00 00 00 01
00 00 00 01
00 00 00 00
WRITE | address: 0x44A00004, data[0]: 0x00000000
DEBUG | packet payload:
44 A0 01 0C
00 00 00 00
00 00 00 06
READ | address: 0x44A0010C, data[0]: 0x7E8AEC14
READ | address: 0x44A00110, data[1]: 0x7E8AEC14
READ | address: 0x44A00114, data[2]: 0x7E8AEC14
READ | address: 0x44A00118, data[3]: 0x7E8AEC14
READ | address: 0x44A0011C, data[4]: 0x7E8AEC14
READ | address: 0x44A00120, data[5]: 0x7E8AEC14
DEBUG | packet payload:
00 00 00 00
00 00 00 03
00 00 00 00
Debug level set to : 0x00
Debug level set to : 0x01
READ | address: 0x44A0010C, data[0]: 0x7F7796DC
READ | address: 0x44A00110, data[1]: 0x7FC70E4B
READ | address: 0x44A00114, data[2]: 0x8016860A
READ | address: 0x44A00118, data[3]: 0x8065FDCE
READ | address: 0x44A0011C, data[4]: 0x80B5758D
READ | address: 0x44A00120, data[5]: 0x8104ED4D
WRITE | address: 0x44A00160, data[0]: 0x00000001
READ | address: 0x44A00140, data[0]: 0x815964DD
READ | address: 0x44A00144, data[1]: 0x815964DD
READ | address: 0x44A00148, data[2]: 0x815964DD
READ | address: 0x44A0014C, data[3]: 0x815964DD
READ | address: 0x44A00150, data[4]: 0x815964DD
READ | address: 0x44A00154, data[5]: 0x815964DD
```

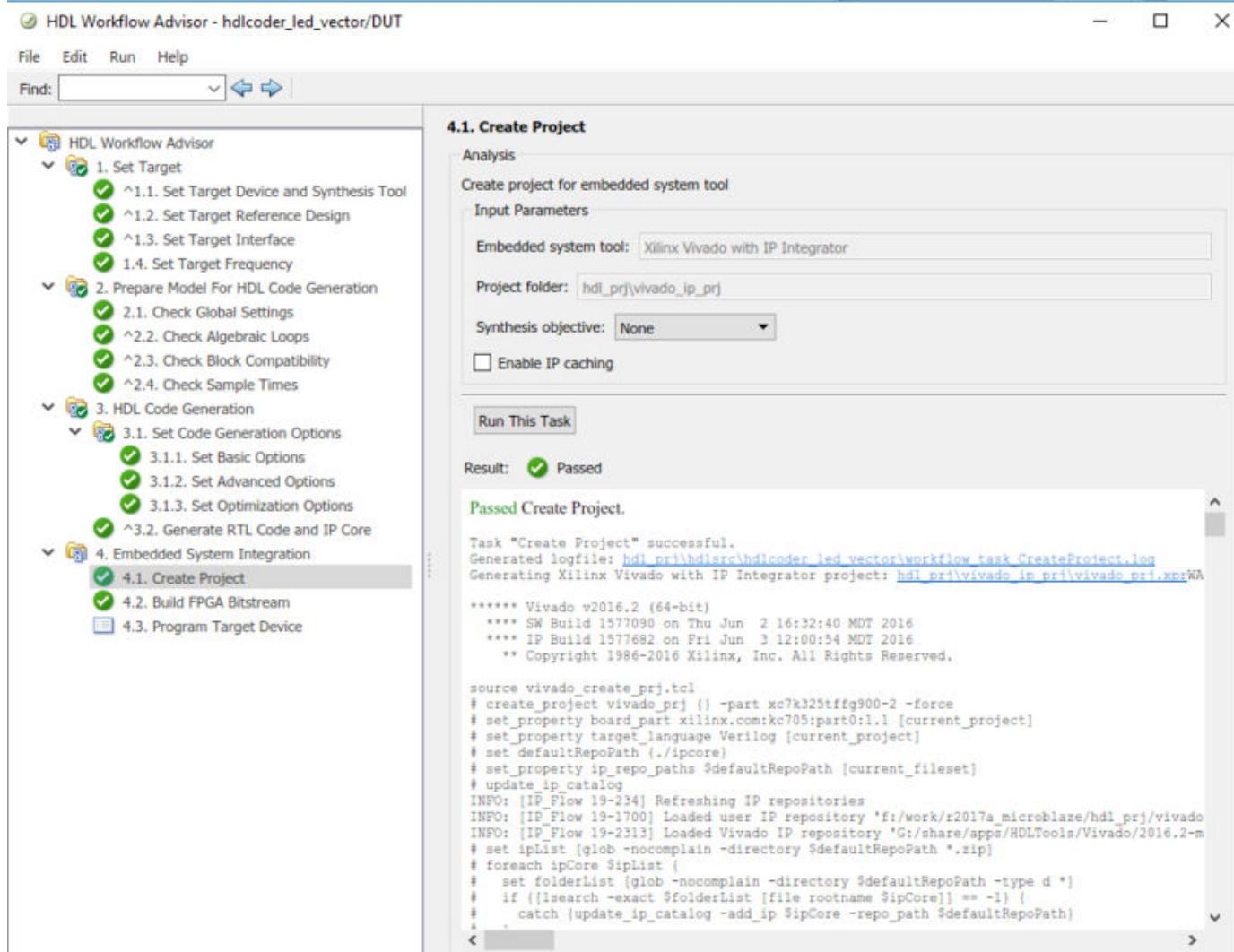
Summary

This demo highlighted the use of a MicroBlaze soft-core processor in FPGA only designs. The MicroBlaze is well suited to function as a full-fledged processor or as a flexible IP running legacy C code as a firmware application. This demo also showed the difference between a collection of scalar ports and a vector port in regards to data synchronization across the AXI4 interface.

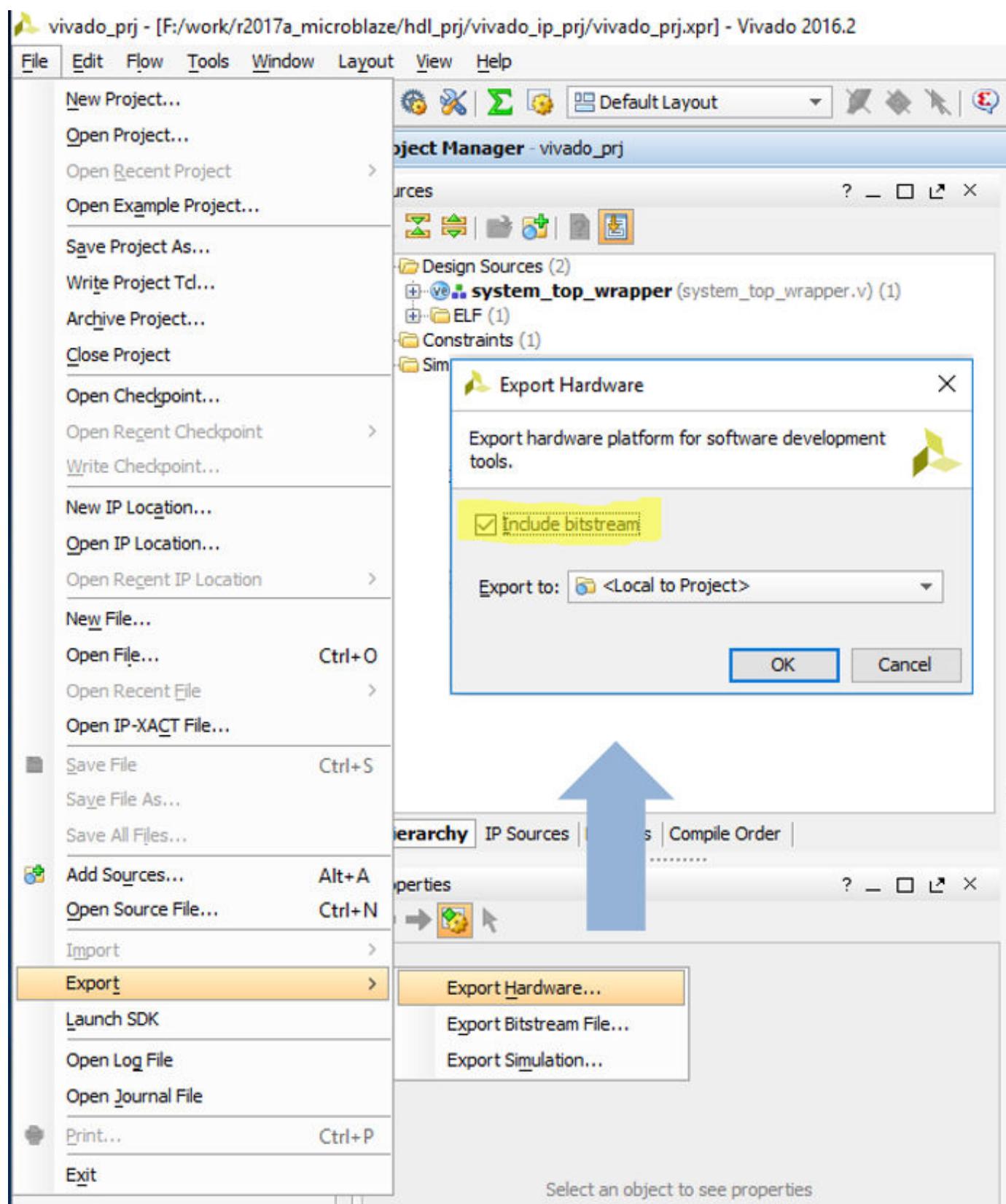
Appendix A: Creating and editing a Xilinx SDK application

This section will show how to create a new Xilinx SDK project and incorporate the code from this example to then modify or extend.

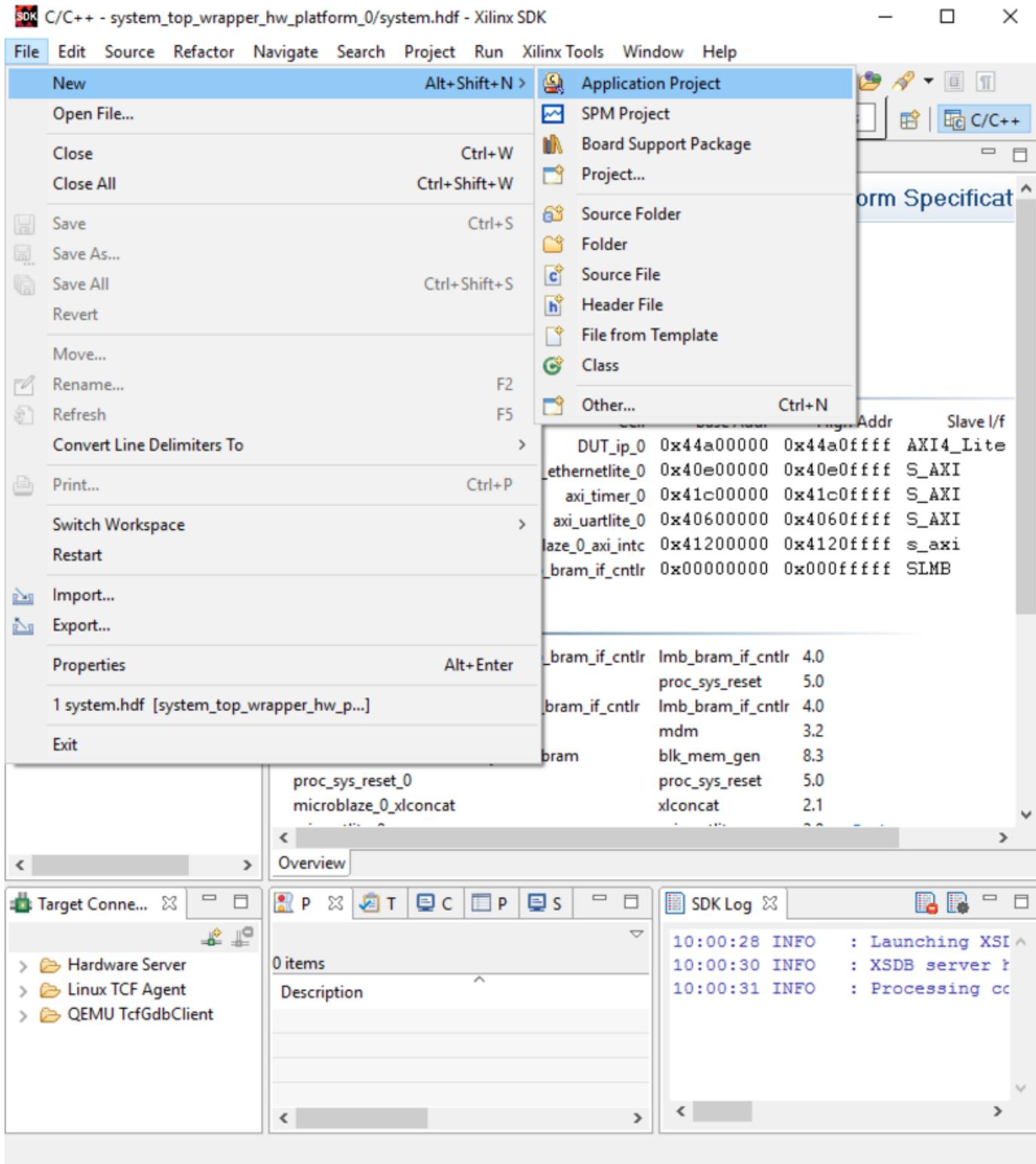
1. Open the Xilinx Vivado project by clicking the link in HDL Workflow Advisor step 4.1 "Create Project":



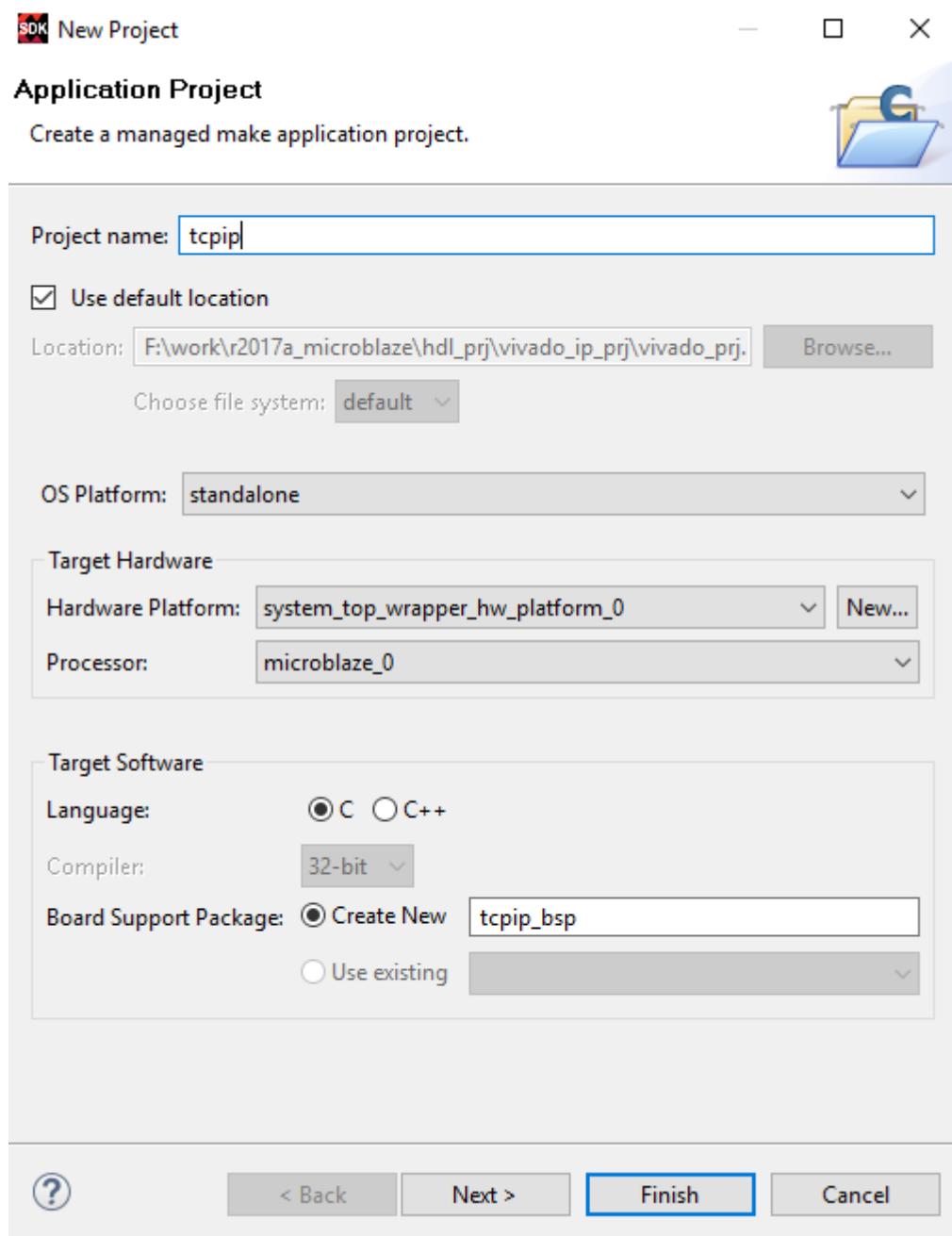
2. Export the existing design, including the generated bitstream, to a local folder. Once this is done, go ahead and "Launch SDK" as well.



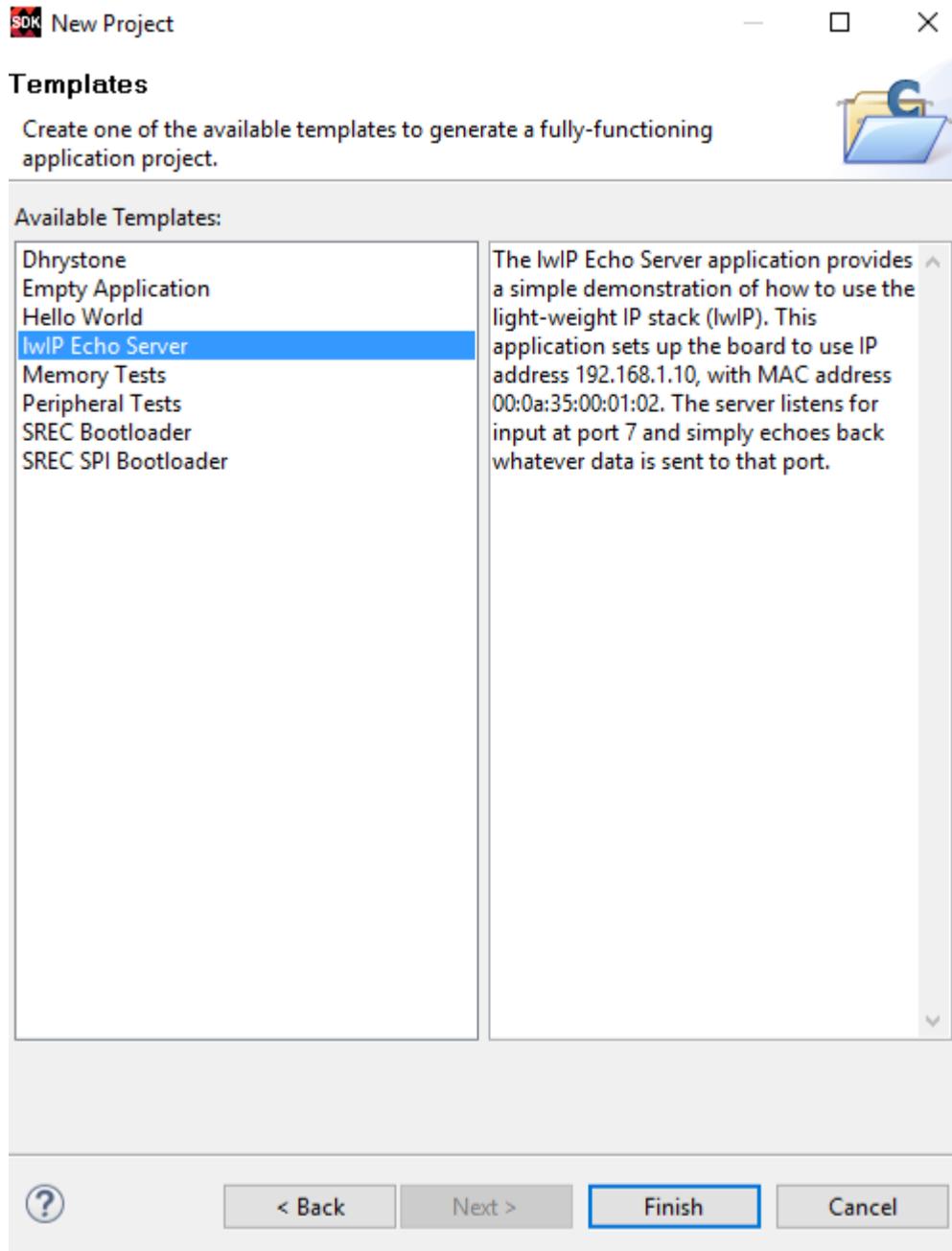
3. From within the SDK, create a new application project



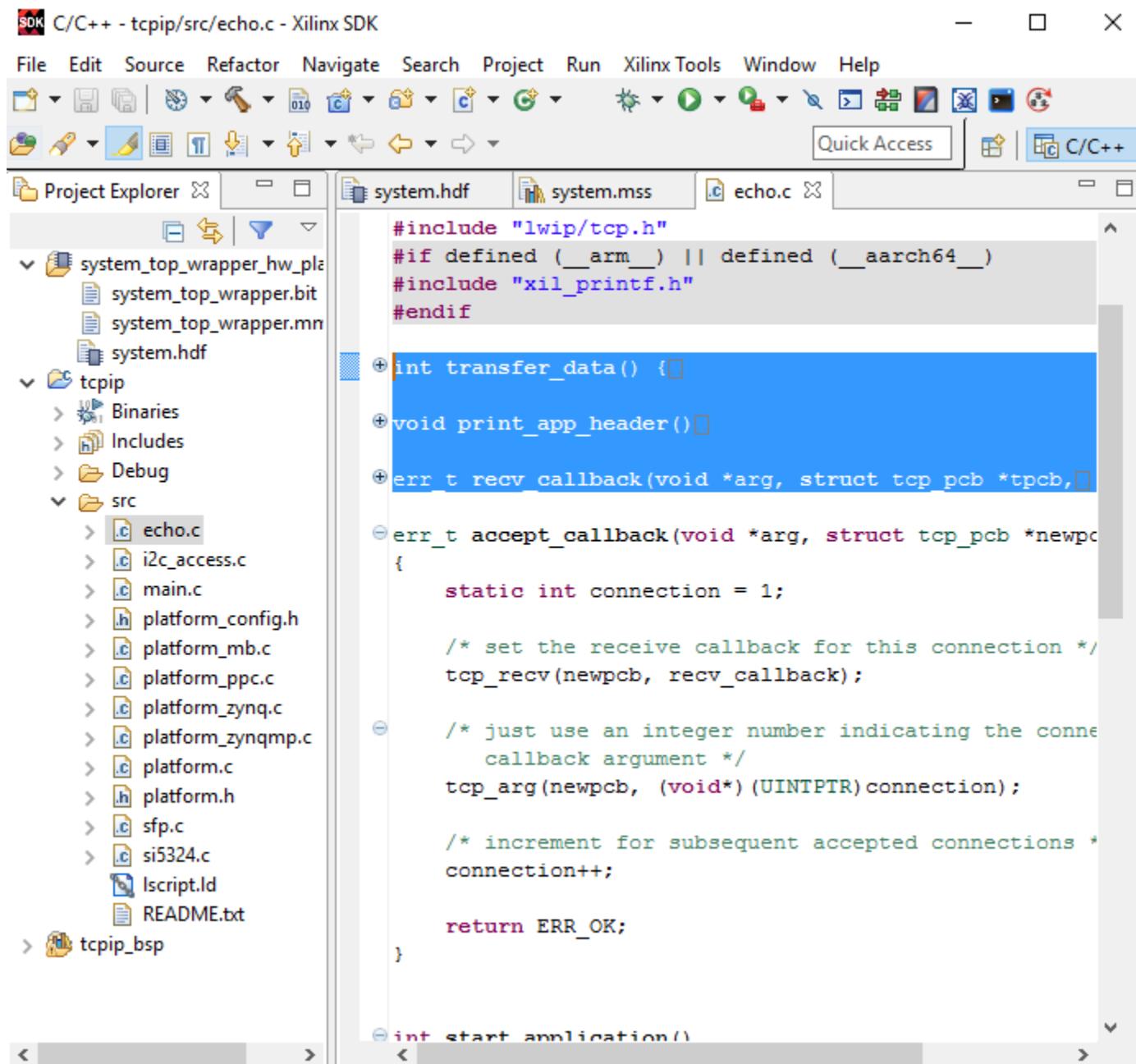
4. You will then have the option of naming the project and creating a new bsp



Next you can choose from a few pre-configured example/template projects to get started. This example is built off of the "lwIP Echo Server" project, so select that now.



5. Using the echo server as a template, you can replace the following 3 methods with the code snippet below to modify the behavior of the server



Appendix B: Copy C file contents to project

```
#define IPCOREBASE 0x44a00000
#define WRITE 0x01
#define READ 0x00
#define DEBUG 0x03

int transfer_data() {
    return 0;
}
```

```
void print_app_header()
{
    xil_printf("\n\r\n\r----MathWorks HDL Coder AXI4-Lite IP Core Read/Write Server ----\n\r");
    xil_printf("  TCP packets sent to port 7 will be issued as AXI4-Lite Read/Writes\n\r");
    xil_printf("\n\r");
    xil_printf("  [ 32-bit address ] (Base Address = 0x44a0_0000)\n\r");
    xil_printf("  [ 32-bit cmd ] (read = 0x00, write =0x01, debug = 0x03)\n\r");
    xil_printf("  [ 32-bit len ] ( N<255)\n\r");
    xil_printf("  [ 32-bit data ] (N 32-bit data values for write cmd)\n\r");
    xil_printf("-----\n\r");
}

void print_packet(struct pbuf *p) {
    u16 ii;
    u8 *pktPtr;

    pktPtr = p->payload;
    xil_printf("DEBUG | packet payload:\r\n");
    for (ii=0;ii<p->len;ii+=4) {
        xil_printf("%02x %02x %02x %02x\r\n",*(pktPtr+ii),*(pktPtr+ii+1),*(pktPtr+ii+2),*(pktPtr+ii+3));
    }

}
err_t recv_callback(void *arg, struct tcp_pcb *tpcb,
                    struct pbuf *p, err_t err)
{
    u8 *pktPtr,*pktEnd;
    volatile u32 *addr;
    u32 data[255],cmd;
    u16 len;
    int ii;
    static u8 debug = 3;

    /* do not read the packet if we are not in ESTABLISHED state */
    if (!p) {
        tcp_close(tpcb);
        tcp_recv(tpcb, NULL);
        return ERR_OK;
    }

    /* indicate that the packet has been received */
    tcp_recved(tpcb, p->len);
    if (debug > 1) print_packet(p);

    //#[ 32 bits address ]
    //#[ 32 bits read = 0x00, write =0x01]
    //#[ 32 bits length ]
    //#[ 32 bits write data]
    pktPtr = p->payload;
    pktEnd = pktPtr+p->len;

    /* could be multiple commands per packet */
    while ( pktPtr < pktEnd) {
        addr = (u32*) (pktPtr[0]<<24 | pktPtr[1]<<16 | pktPtr[2]<<8 | pktPtr[3]);
    }
}
```

```

cmd = (u32) pktPtr[7];           // cmd is 32 bits, but only 1st byte used, ignore rest
pktPtr += 8;

switch(cmd) {
case WRITE :
    len = (u32) pktPtr[3]; // len is 32 bits, but only 1st byte used, ignore rest
    pktPtr += 4;
    for (ii=0;ii<len; ii++) {
        data[0] = (u32) (pktPtr[0]<<24 | pktPtr[1]<<16 | pktPtr[2]<<8 | pktPtr[3]);
        *addr = data[0];
        if (debug > 0) xil_printf("WRITE | address: 0x%08x, data[0]: 0x%08x\r\n",addr,data[0]);
        addr++;
        pktPtr += 4;
    }
    break;
case READ :
    len = (u32) pktPtr[3]; // len is 32 bits, but only 1st byte used, ignore rest
    pktPtr += 4;
    for (ii=0;ii<len; ii++) {
        data[ii] = *addr;
        if (debug > 0) xil_printf("READ | address: 0x%08x, data[%d]: 0x%08x\r\n",addr,data[ii]);
        addr++;
    }
    /* send the packet back */
    if (tcp_sndbuf(tpcb) > p->len)
        err = tcp_write(tpcb, data, 4*len, 1);
    else
        xil_printf("no space in tcp_sndbuf\r\n");
    break;
case DEBUG:
    debug = pktPtr[3]; // only need the low byte
    pktPtr += 4;
    xil_printf("Debug level set to : 0x%02x\r\n",debug);
    break;
default :
    xil_printf("INVALID | cmd: 0x%08x\r\n",cmd);

}
/* free the received pbuf */
pbuf_free(p);

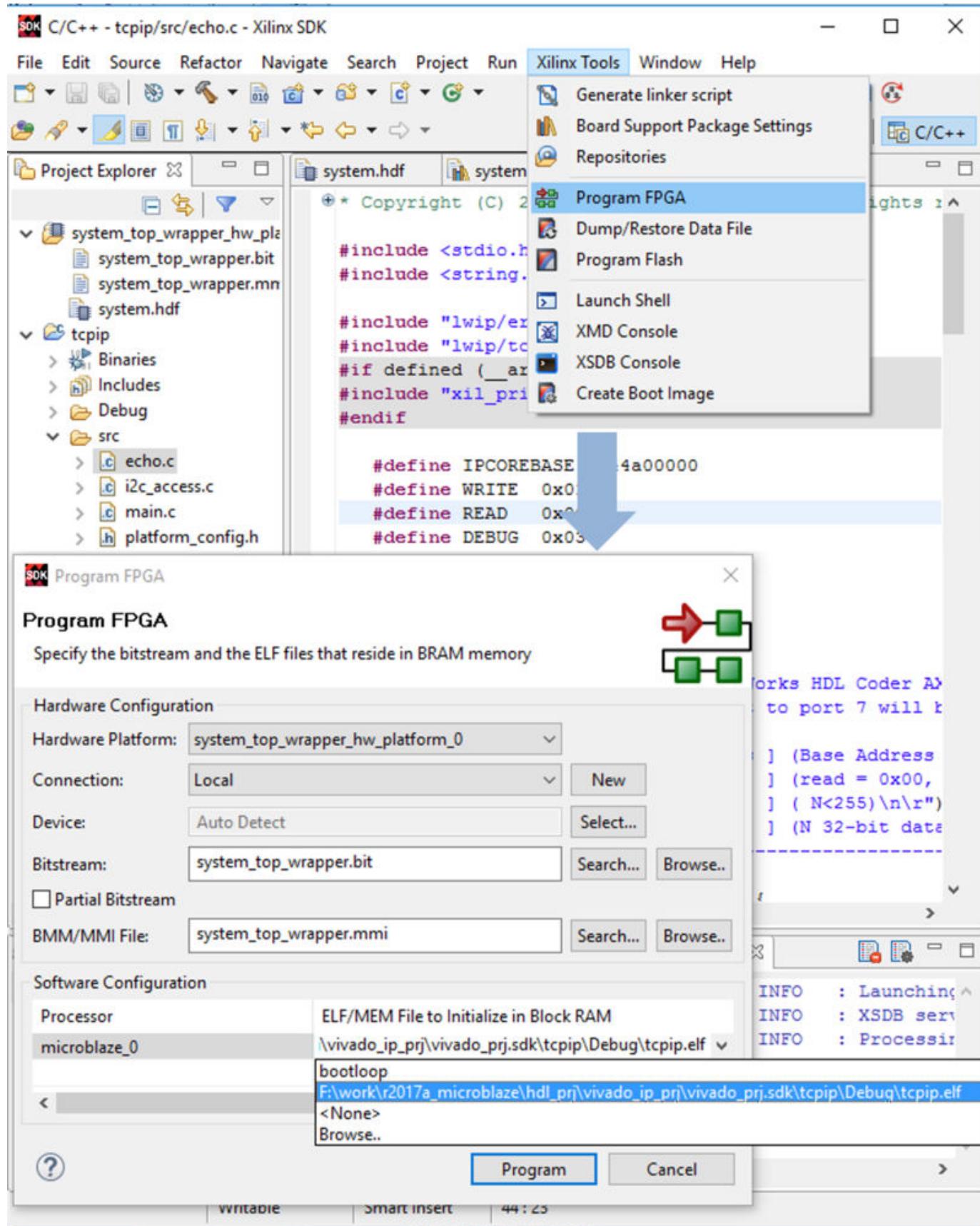
return ERR_OK;
}

```

6. Save the modified echo.c file and the application will be rebuilt.

Appendix C: Program the FPGA with ELF and bitstream

Now, you can program the FPGA using the exported bitstream and the newly created ELF file.



Target SoC Platforms and Speedgoat Boards

- “Model Design for AXI4 Slave Interface Generation” on page 41-3
- “Model Design for AXI4-Stream Interface Generation” on page 41-10
- “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 41-17
- “Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard” on page 41-22
- “Multirate IP Core Generation” on page 41-35
- “Board and Reference Design Registration System” on page 41-39
- “Register a Custom Board” on page 41-42
- “Register a Custom Reference Design” on page 41-45
- “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 41-48
- “Customize Reference Design Dynamically Based on Reference Design Parameters” on page 41-54
- “Define and Add IP Repository to Custom Reference Design” on page 41-59
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63
- “Model Design for AXI4-Stream Video Interface Generation” on page 41-69
- “Model Design for AXI4 Master Interface Generation” on page 41-78
- “IP Core Generation Workflow for Standalone FPGA Devices” on page 41-89
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 41-93
- “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 41-96
- “Running an Audio Filter on Live Audio Input using Intel Board” on page 41-116
- “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 41-126
- “Getting Started with AXI4-Stream Interface in Zynq Workflow” on page 41-137
- “Getting Started with AXI4-Stream Video Interface in Zynq Workflow” on page 41-152
- “Performing Large Matrix Operation on FPGA using External Memory” on page 41-162
- “Authoring a Reference Design for Audio System on a Zynq Board” on page 41-170
- “Authoring a Reference Design for Audio System on a ZYBO Board” on page 41-180
- “Authoring a Reference Design for Audio System on Intel board” on page 41-186
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215
- “Dynamically Create Master Only or Slave Only or Both Slave and Master Reference Designs” on page 41-229
- “Using JTAG MATLAB as AXI Master to control HDL Coder generated IP Core” on page 41-242

- “Debug a Zynq Design Using HDL Coder and Embedded Coder” on page 41-248
- “Debug IP Core Using FPGA Data Capture” on page 41-253

Model Design for AXI4 Slave Interface Generation

In this section...

- ["Considerations" on page 41-3](#)
- ["Map Scalar Ports to AXI4 Slave Interface" on page 41-3](#)
- ["Map Vector Ports to AXI4 Slave Interface" on page 41-4](#)
- ["Initial Value of AXI4 Slave Registers" on page 41-5](#)
- ["Read Back Value of AXI4 Slave Interfaces" on page 41-6](#)
- ["Optimize AXI4 Slave Read Back Logic" on page 41-9](#)

To perform lightweight data transfer or to access control registers, use AXI4 slave interfaces. The AXI4 slave interfaces include the AXI4 and AXI4-Lite interfaces. With the HDL Coder software, you don't have to implement AXI4 or AXI4-Lite protocol in your model. The software generates AXI4 or AXI4-Lite interfaces in the HDL IP core.

When you model your design, specify the data ports that you want to map to the AXI4 slave interfaces. HDL Coder then maps the data ports to memory-mapped AXI4 slave interfaces and allocates address offsets for the ports.

Considerations

When you map your DUT ports to AXI4 or AXI4-Lite interfaces:

- You can map all scalar or vector ports in your design to either AXI4 or AXI4-Lite interfaces. You cannot map some DUT ports to AXI4 interfaces and other DUT ports to AXI4-Lite interfaces for the same design.
- You can use a single-rate design or a design with multiple sample rates without any restrictions when mapping the DUT ports to AXI4 slave interfaces by using the **Free Running** synchronization mode for **Processor/FPGA Synchronization**.
- You can use the **Coprocessing-Blocking** mode for **Processor/FPGA Synchronization** when mapping to AXI4 or AXI4-Lite interfaces. Other interface types such as AXI4-Stream and AXI4 Master do not support this mode. See also "Processor and FPGA Synchronization" on 40-23.

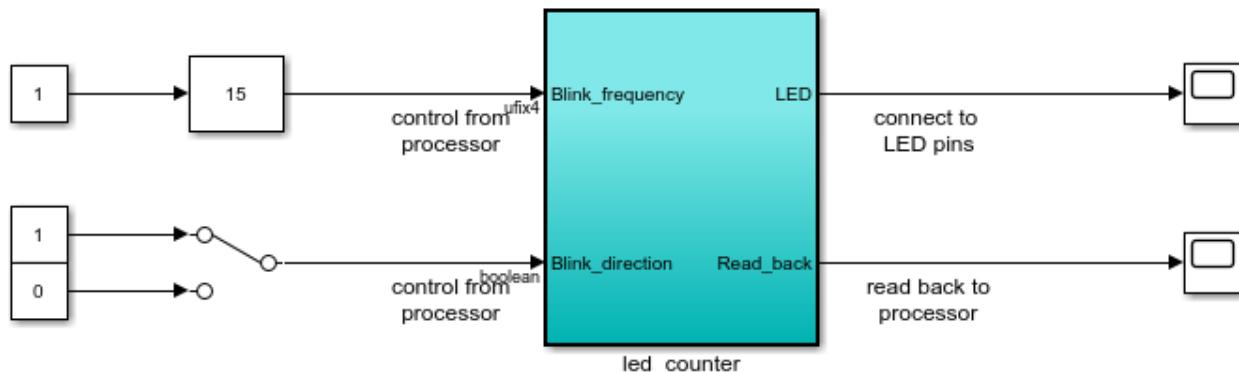
Map Scalar Ports to AXI4 Slave Interface

When you use scalar data types at the DUT interface ports, you can map the interface ports directly to AXI4 or AXI4-Lite interfaces. The code generator assigns a unique address to each data port that you want to map to the AXI4 interface.

For an example that shows how to map scalar ports to AXI4-Lite interfaces, open the model `hdlcoder_led_blinking`.

```
open_system('hdlcoder_led_blinking')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking/led_counter')`

[Launch HDL Workflow Advisor](#)

[Run Demo](#)

Copyright 2012 The MathWorks, Inc.

In this model, the subsystem `led_counter` is the hardware subsystem. It models a counter that blinks the LEDs on an FPGA board. Two input ports, `Blink_frequency` and `Blink_direction`, are control ports that determine the LED blink frequency and direction. All the blocks outside of the subsystem `led_counter` are for software implementation.

In Simulink, you can use the Slider Gain block or the Manual Switch block to adjust the input values of the hardware subsystem. In the embedded software, this means that the ARM processor controls the generated IP core by writing to the AXI interface accessible registers. The output port of the hardware subsystem connects to the LED hardware. You can use the output port `Read_back` to read data back to the processor.

When you run the IP Core Generation workflow, in the **Set Target Interface** task, you see that the ports `Blink_frequency`, `Blink_direction`, and `Read_back` map to AXI4-Lite interfaces.

To learn more about this example, see:

- “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65
- “Getting Started with Targeting Intel SoC Devices” on page 40-104

Map Vector Ports to AXI4 Slave Interface

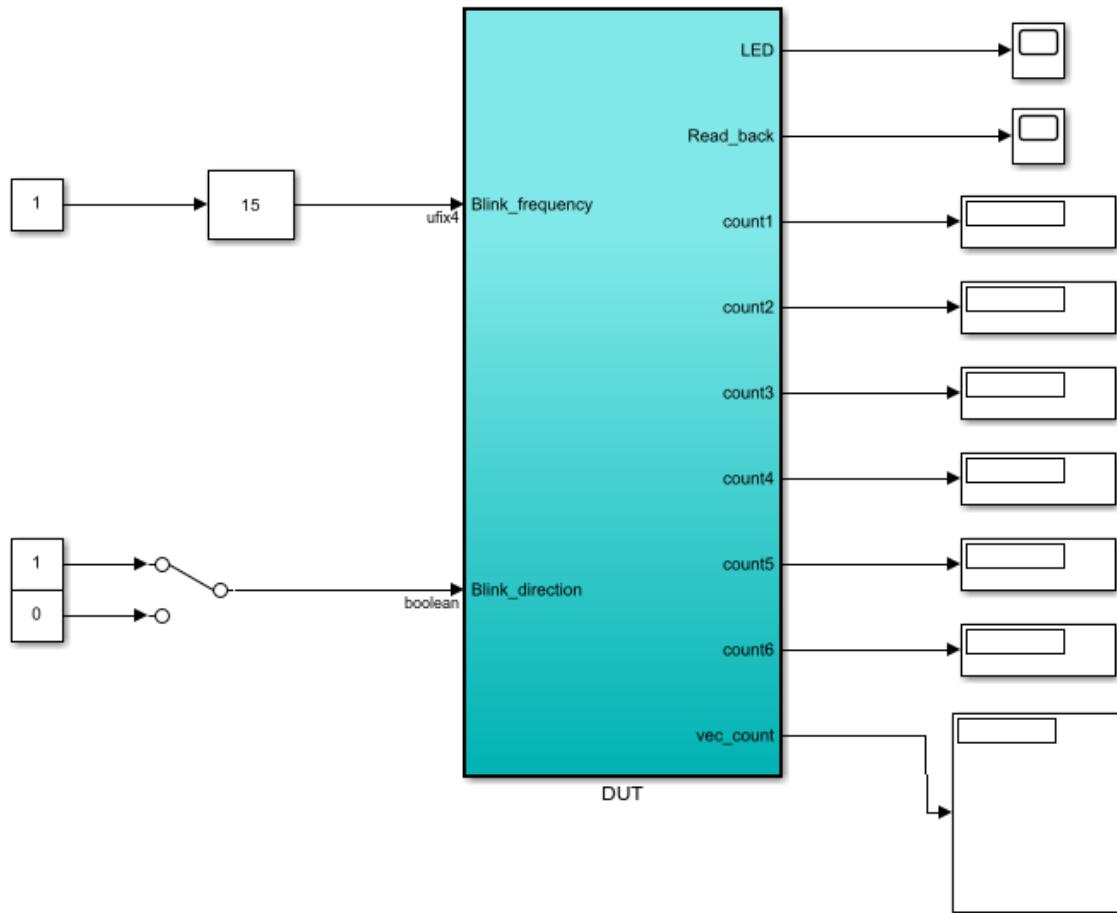
When you use vector data types at the DUT interface ports, you can map the interface ports directly to AXI4 or AXI4-Lite interfaces. The code generator assigns a unique address for each data port that you want to map to the AXI4 interface.

When you map vector ports, HDL Coder uses additional strobe registers for each port to maintain the synchronization with the IP core algorithm logic. For input ports, the strobe registers control the

enable signals for a set of shadow registers, which makes the IP core algorithm logic see the updated vector elements simultaneously. For output ports, the strobe registers make sure that the vector data to be read is captured synchronously.

For an example that shows how to map vector ports to AXI4-Lite interfaces, open the model `hdlcoder_led_vector`.

```
open_system('hdlcoder_led_vector')
```



In this model, the subsystem DUT implements the LED blinking algorithm and has vector output ports. When you run the IP Core Generation workflow, in the **Set Target Interface** task, you see that the input ports and output ports map to AXI4-Lite interfaces.

To learn more, see “IP Core Generation Workflow with a MicroBlaze processor: Xilinx Kintex-7 KC705” on page 40-172.

Initial Value of AXI4 Slave Registers

When you run the IP Core Generation workflow or the Simulink Real-Time FPGA I/O workflow, you can specify an initial value for the AXI4 slave registers. You can specify an initial value when mapping scalar and vector input and output ports to these target interfaces.

- AXI4
- AXI4-Lite
- PCIe

By default, the initial value is zero. To specify a nonzero value:

- 1 In the target platform interface table, when you map DUT port to an AXI4 slave interface, an **Options** button appears in the **Interface Options** column.
- 2 Click the **Options** button and then specify the **RegisterInitialValue**.

The specified value is saved on the DUT Import and Outport blocks as the HDL block property **IOInterfaceOptions** in the **Target Specification** tab. For example, if you map a DUT input port to AXI4-Lite interface, set **RegisterInitialValue** to 5, and then run this task, the **IOInterfaceOptions** property of that input port is saved with the value `{'RegisterInitialValue', '5'}`.

To view the **IOInterfaceOptions** value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```
hdlget_param('hdlcoder_led_blinking/led_counter/LED', ...
    'IOInterfaceOptions')
```

This example shows how you specify the initial value for vector ports.

```
hdlset_param('sfir_fixed_stream/DUT/In2', ...
    'IOInterfaceOptions', {'RegisterInitialValue', '[3.5 3.5 3.5']});
```

Read Back Value of AXI4 Slave Interfaces

When you run the **IP Core Generation** workflow, you can read back the value that is written to the AXI4 slave registers by using the AXI4 slave interface. For example, you can read back the values that are written to the AXI4 slave registers by using the `devmem` command in the Linux console of the ARM processor. If you have HDL Verifier installed, you can use the MATLAB as AXI Master IP to read back the values.

To use this capability, in the **Generate RTL Code and IP Core** task of the **IP Core Generation** workflow, select the **Enable read back on AXI4 slave write registers** check box, and then run this task.

3.2. Generate RTL Code and IP Core

Analysis (^Triggers Update Diagram)

Generate RTL code and IP core for embedded system

Input Parameters

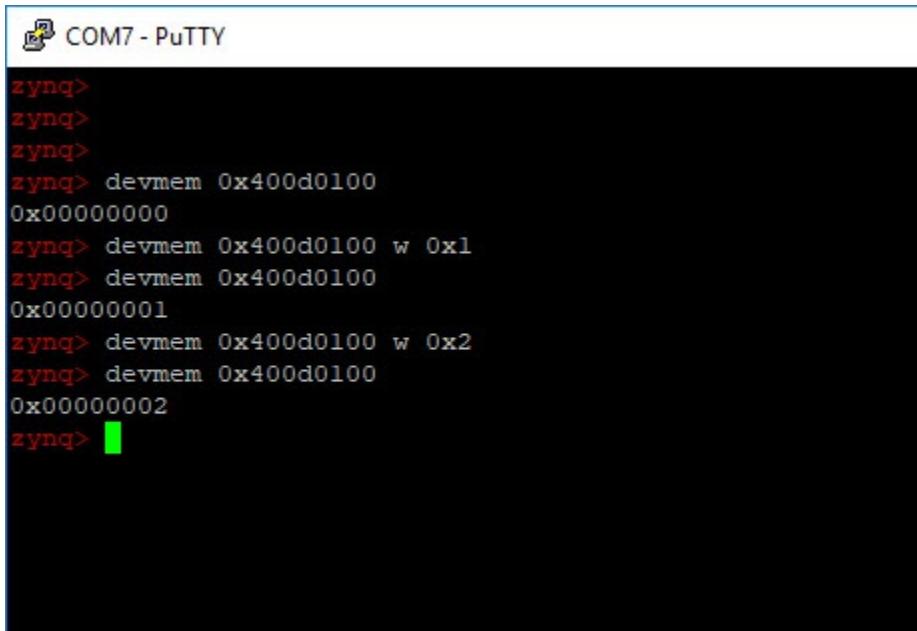
IP core name:	DUT_ip
IP core version:	1.0
IP core folder:	hdl_prj\ipcore\DUT_ip_v1_0
IP repository:	<input type="text"/> <input type="button" value="Browse..."/>
Additional source files:	<input type="text"/> <input type="button" value="Add Source..."/>
FPGA Data Capture buffer size:	128
<input checked="" type="checkbox"/> Generate IP core report <input checked="" type="checkbox"/> Enable readback on AXI4 slave write registers	

When you run this task, HDL Coder saves the read back setting that you enabled on the model. In the HDL Block Properties of the DUT Subsystem, on the **IP Core Parameter** section of the **Target Specification** tab, you see a parameter **AXI4RegisterReadback** set to **on**. If you export the HDL Workflow Advisor run to a script, you see this setting saved on the model by using `hdlset_param`.

```
hdlset_param('hdlcoder_led_vector/DUT', 'AXI4RegisterReadback', 'on');
```

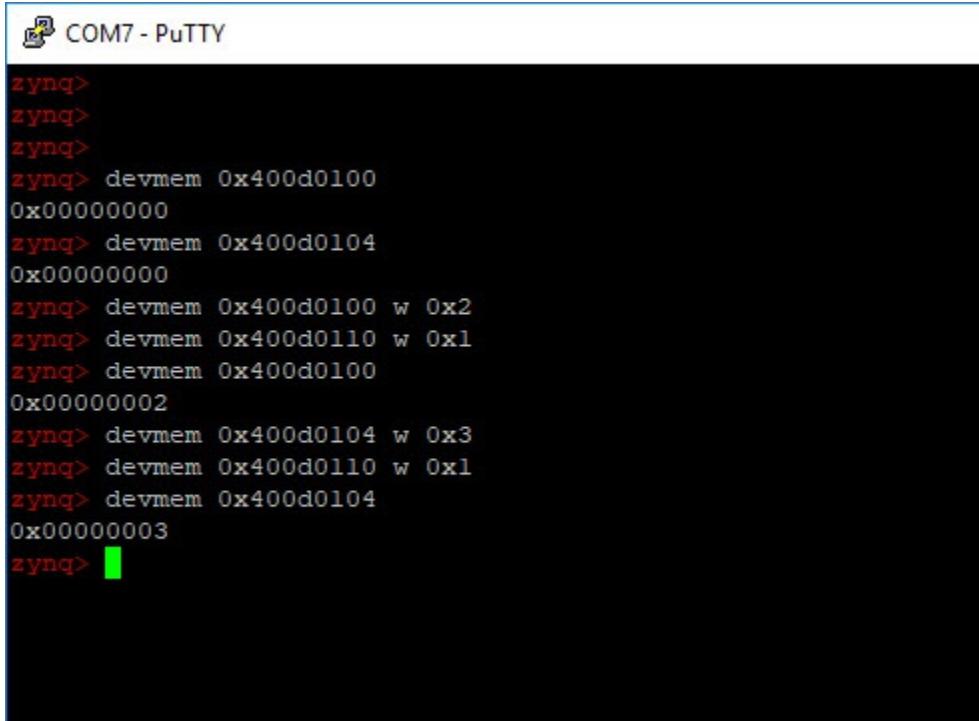
These examples show how you can read back values by using the `devmem` command in the Linux console with a program such as PuTTy.

To read back values when mapping scalar ports to AXI4 interfaces, you first write values to the AXI4 registers, and then read back the values. You can see the memory address of the AXI4 registers in the IP Core Generation report.



zynq>
zynq>
zynq>
zynq> devmem 0x400d0100
0x00000000
zynq> devmem 0x400d0100 w 0x1
zynq> devmem 0x400d0100
0x00000001
zynq> devmem 0x400d0100 w 0x2
zynq> devmem 0x400d0100
0x00000002
zynq> █

To read back values when mapping vector ports to AXI4 interfaces, you first write to the AXI4 registers, then write the strobe register address with 0x1, and then read back the values. You can see the memory address of the AXI4 registers and the strobe register in the IP Core Generation report.



zynq>
zynq>
zynq>
zynq> devmem 0x400d0100
0x00000000
zynq> devmem 0x400d0104
0x00000000
zynq> devmem 0x400d0100 w 0x2
zynq> devmem 0x400d0110 w 0x1
zynq> devmem 0x400d0100
0x00000002
zynq> devmem 0x400d0104 w 0x3
zynq> devmem 0x400d0110 w 0x1
zynq> devmem 0x400d0104
0x00000003
zynq> █

Optimize AXI4 Slave Read Back Logic

When your model contains several output registers and you want to read back data from multiple AXI4 slave registers, the read back logic becomes a long mux chain that can reduce the synthesis frequency. If you select the **Enable readback on AXI4 slave write registers** setting in the **Generate RTL Code and IP Core** task, HDL Coder adds a mux for each AXI4 register in the Address Decoder logic. As the number of AXI4 slave registers increases, the mux chain becomes longer, which further reduces the synthesis frequency.

You can optimize the readback logic and achieve the target frequency that you want. When you run the IP Core Generation workflow, in the **Generate RTL Code and IP Core** task, you see a setting **AX4 slave port to pipeline register ratio**. The default value of this setting is **auto**. This setting indicates how many AXI4 slave registers a pipeline register is inserted for. For example, an **AX4 slave port to pipeline register ratio** of 20 means that one pipeline register is inserted for every 20 AXI4 slave registers. The **auto** setting means that the code generator inserts a certain number of pipelines for the AXI4 slave ports depending on the number of ports and the synthesis tool that you specify. You can disable this setting or select a number between 5 and 50 for this ratio.

When you run this task, HDL Coder saves the value that you specified for the setting on the model. In the HDL Block Properties of the DUT Subsystem, on the **IP Core Parameter** section of the **Target Specification** tab, you see a parameter **AX4SlavePortToPipelineRegisterRatio** set to the value that you specified. If you export the HDL Workflow Advisor run to a script, you see this setting saved on the model by using `hdlset_param`.

```
hdlset_param('hdlcoder_led_vector/DUT', ...
    'AXI4SlavePortToPipelineRegisterRatio', '20');
```

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Custom IP Core Generation” on page 40-10

Model Design for AXI4-Stream Interface Generation

In this section...

- “Simplified Streaming Protocol” on page 41-10
- “Map Scalar Ports To AXI4-Stream Interface” on page 41-11
- “Map Vector Ports To AXI4-Stream Interface” on page 41-14
- “Model Designs with Multiple Streaming Channels” on page 41-15
- “Model Designs with Multiple Sample Rates” on page 41-15
- “Restrictions” on page 41-15

With the HDL Coder software, you can implement a simplified, streaming protocol in your model. The software generates AXI4-Stream interfaces in the IP core.

Simplified Streaming Protocol

To map the DUT ports to AXI4-Stream interfaces, use the simplified AXI4-Stream protocol. You do not have to model the actual AXI4-Stream protocol and instead you can use the simplified protocol. When you run the **IP Core Generation** workflow, the generated HDL code contains a wrapper logic that translates between the simplified protocol and the actual AXI4-Stream protocol. The simplified protocol requires you to use less protocol signals, eases the handshaking mechanism between valid and ready signals, and supports bursts of arbitrary lengths.

Use the simplified AXI4-Stream protocol for both write and read transactions. When you want to generate an AXI4-Stream interface in your IP core, in your DUT interface, implement the following signals:

- Data
- Valid

Optionally, when you map scalar DUT ports to an AXI4-Stream interface, you can model the following signals:

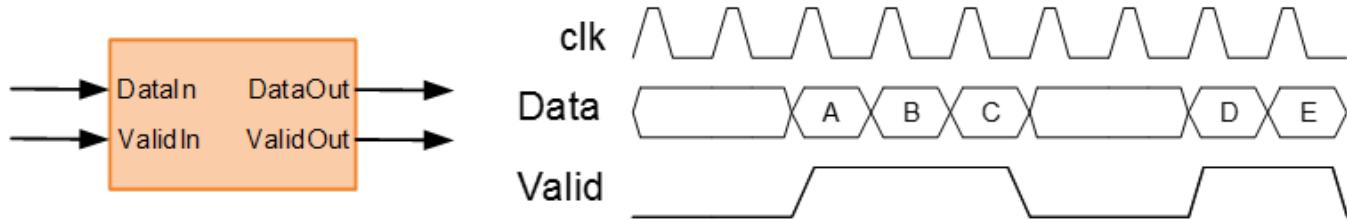
- Ready
- Other protocol signals, such as:
 - TSTRB
 - TKEEP
 - TLAST
 - TID
 - TDEST
 - TUSER

Data and Valid Signals

When the Data signal is valid, the Valid signal is asserted.

Note This diagram illustrates the Data and Valid signal relationship according to the simplified streaming protocol. When you run the **IP Core Generation** workflow, HDL Coder adds a streaming

interface module in the HDL IP core that translates the simplified protocol to the full AXI4-stream protocol.



Map Scalar Ports To AXI4-Stream Interface

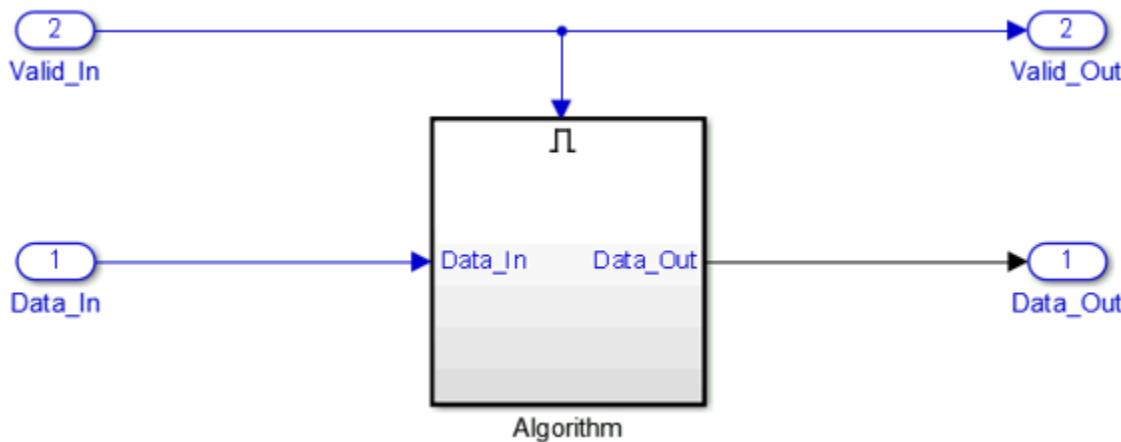
If you want to generate a hardware IP core, but do not need to model and simulate the interaction between the software and hardware, use scalar data ports at your DUT interface. Map the data ports to AXI4-Stream interfaces.

Data and Valid Signal Modeling Pattern

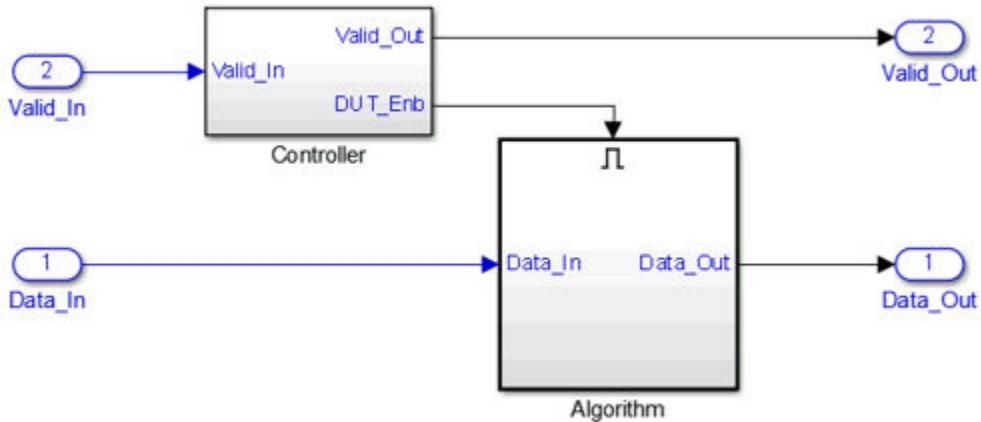
To model the Data and Valid signals in Simulink:

- 1 Enclose the algorithm that processes the Data signal by using an enabled subsystem.
- 2 Control the enable port of the enabled subsystem by using the Valid signal.

For example, you can directly connect the Valid signal to the enable port.



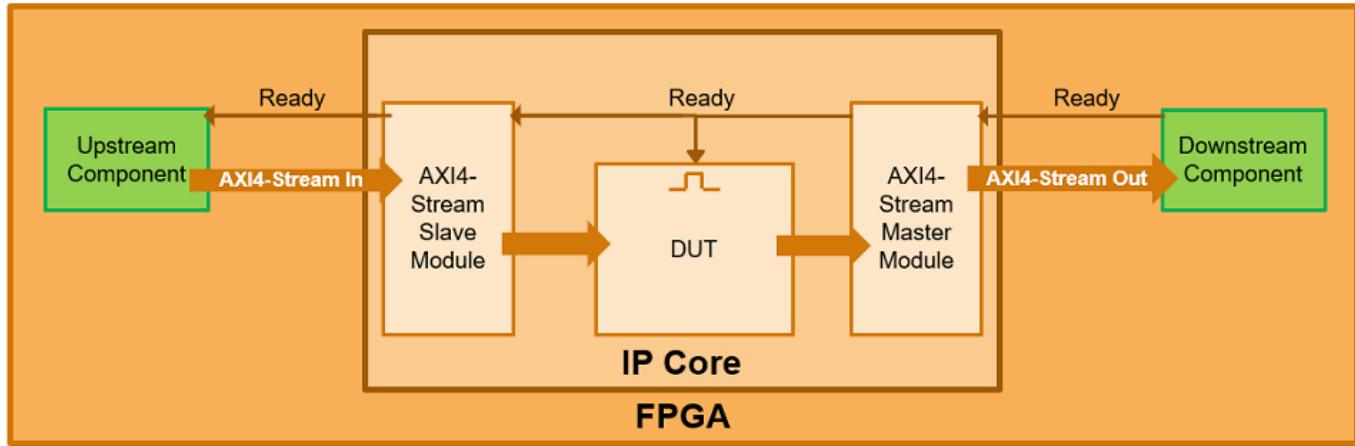
You can also use a controller in your DUT that generates an enable signal for the enabled subsystem.



Ready Signal (Optional)

The AXI4-Stream interfaces in your DUT can optionally include a Ready signal. In a Slave interface, you use the Ready signal to apply back pressure. In a Master interface, you use the Ready signal to respond to back pressure.

By default, HDL Coder generates the Ready signal automatically. When you use a single streaming channel, HDL Coder also generates the logic that handles back pressure. The back pressure logic ties the Ready signal to the DUT Enable signal. When the input Master Ready signal goes low, the DUT is disabled, and the output slave Ready signal is driven low. Therefore, when you use a single streaming channel, the Ready signal is optional and you do not have to model this signal at the DUT port.

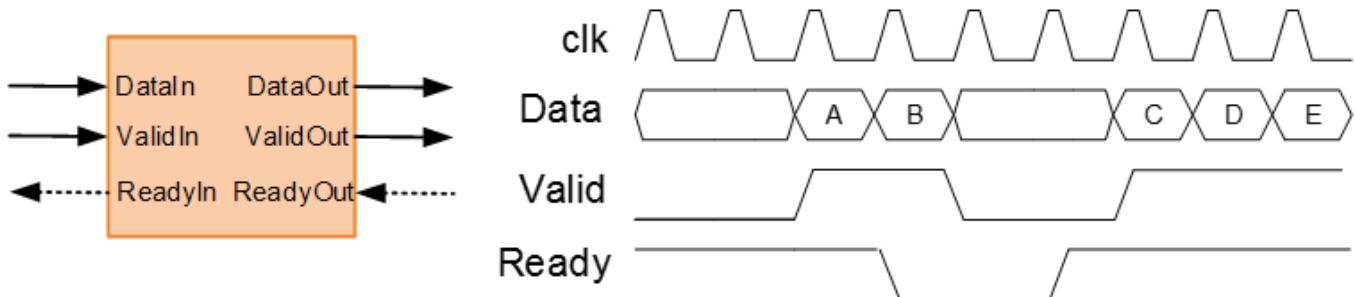


If you use multiple streaming channels, HDL Coder does not automatically generate the back pressure logic. In this case, the Ready signal is generated but the master Ready signal at the input is ignored and the slave Ready signal at the output is tied to high value. The absence of a back pressure logic can result in samples being dropped. If you want your design to apply back pressure on the Slave interface or respond to back pressure from the Master interface, you must model the Ready signal for each additional interface and then map the port to the Ready signal for that interface. When you do not model, the **Set Target Interface** task displays a warning that provides names of interfaces that require a Ready port. If your design does not need to apply or respond to back pressure, you can ignore this warning and you do not have to model the Ready signal.

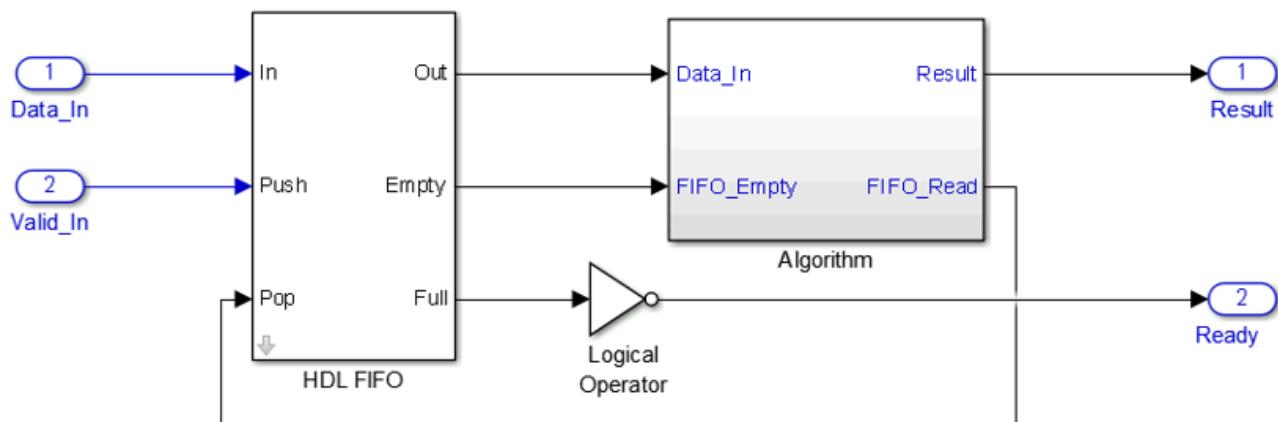
If you model the Ready signal in your AXI4-Stream interfaces, your Master interface ignores the Data and Valid signals one clock cycle after the Ready signal is de-asserted. You can start sending Data and Valid signals once the Ready signal is asserted. You can send one more Data and Valid signal after the Ready signal is de-asserted.

If you do not model the Ready signal, HDL Coder generates the signal and the associated back pressure logic.

Note This diagram illustrates the relationship between the Data, Valid, and Ready signals according to the simplified streaming protocol. When you run the IP Core Generation workflow, HDL Coder adds a streaming interface module in the HDL IP core that translates the simplified protocol to the full AXI4-stream protocol.



For example, if you have a FIFO in your DUT to store a frame of data, to apply back pressure to the upstream component, you can model the Ready signal based on the FIFO Full signal.



Note If you enable delay balancing, the coder can insert one or more delays on the Ready signal. Disable delay balancing for the Ready signal path.

Other Protocol Signals (optional)

You can optionally model other AXI4-Stream protocol signals. If you model only the required Data and Valid signals, the coder generates the TREADY and TLAST AXI4-Stream protocol signals.

If you do not model the TLAST signal, the coder generates a programmable register in the IP core so that you can specify your packet size. The details of the programmable packet size register are in your IP core generation report.

Map Vector Ports To AXI4-Stream Interface

If you want to model and simulate the system interaction between the software and hardware, and generate code for the software driver, use vector data ports at your DUT interface. Map the data ports to AXI4-Stream interfaces.

Data and Valid Signal Modeling Requirements

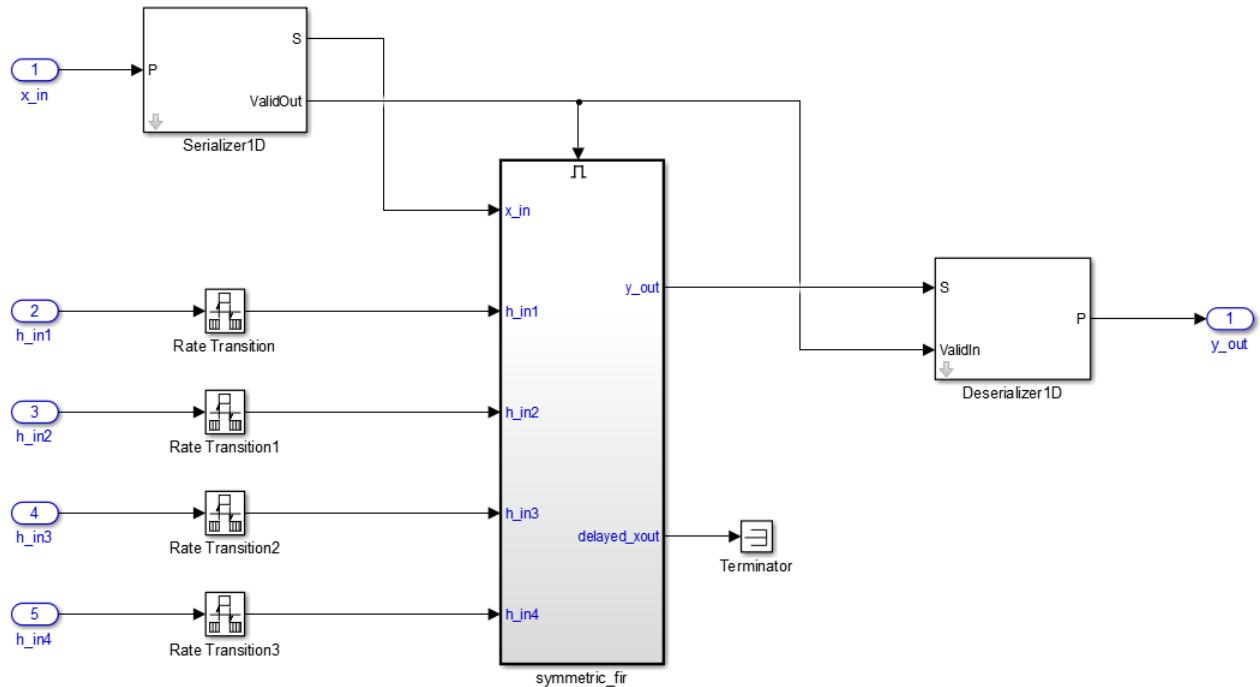
When you map vector ports to AXI4-Stream interfaces, your model has these requirements:

- Connect each DUT input vector data port to a Serializer1D block.
The Serializer1D block must have a ValidOut port and the Ratio set to the vector bit width.
- Connect each DUT output vector data port to a Deserializer1D block.
The Deserializer1D block must have a ValidIn port and the Ratio set to the vector bit width.
- Connect each scalar port that maps to an AXI4-Lite interface to a Rate Transition block.
The Ratio in the Rate Transition block must match the Ratio in the Serializer1D and Deserializer1D blocks.
- Each scalar port that maps to an external port must have the same sample time as the streaming algorithm subsystem.

The streaming algorithm subsystem follows the same Data and Valid signal modeling pattern as for mapping scalar ports to an AXI4-Stream interfaces. See “Data and Valid Signal Modeling Pattern” on page 41-11.

Example

For an example that shows how to map vector ports to AXI4-Stream interfaces, open the `hdlcoder_sfir_fixed_vector` model. In the `hdlcoder_sfir_fixed_vector` model, `symmetric_fir` is the streaming algorithm subsystem.



Model Designs with Multiple Streaming Channels

When you run the IP Core Generation workflow, you can map multiple scalar DUT ports to AXI4-Stream Master and AXI4-Stream Slave channels. When you use vector ports, you can map the ports to at most one AXI4-Stream Master channel and one AXI4-Stream Slave channel.

Note If you use multiple streaming channels, HDL Coder generates the Ready signal but does not generate the back pressure logic. If you want your design to handle back pressure, model the Ready signal in your design.

To learn more, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 41-17.

Model Designs with Multiple Sample Rates

The HDL Coder software supports designs with multiple sample rates when you run the IP Core Generation workflow. When you map the interface ports to AXI4-Stream Master or AXI4-Stream Slave interfaces, to use multiple sample rates, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation.

To learn more, see “Multirate IP Core Generation” on page 41-35.

Restrictions

When you map scalar or vector DUT ports to AXI4-Stream interfaces:

- Xilinx Zynq-7000 or Intel Quartus Prime must be your target platform.
- Xilinx Vivado or Intel Quartus Prime must be your synthesis tool.
- **Processor/FPGA synchronization** must be Free running.

When you map vector DUT ports to AXI4-Stream interfaces, you cannot use protocol signals other than Data and Valid. For example, Ready and TLAST are not supported.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19

See Also

Related Examples

- “Getting Started with AXI4-Stream Interface in Zynq Workflow” on page 41-137
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces

When you run the generic IP Core Generation workflow for your Simulink model or target your own custom reference design that you authored, you can generate an HDL IP core with multiple AXI4-Stream interfaces, AXI4-Stream Video interfaces, or AXI4 Master interfaces. To learn about these interfaces, see “Target Platform Interfaces” on page 40-10.

Why Use Multiple AXI4 Interfaces

You can use multiple streaming interfaces to facilitate high-speed data transfer in various applications such as:

- Transferring data between A/D and D/A converters
- Software-defined radio algorithms that process multiple transceiver channels
- Vision algorithms that perform image annotation or object detection

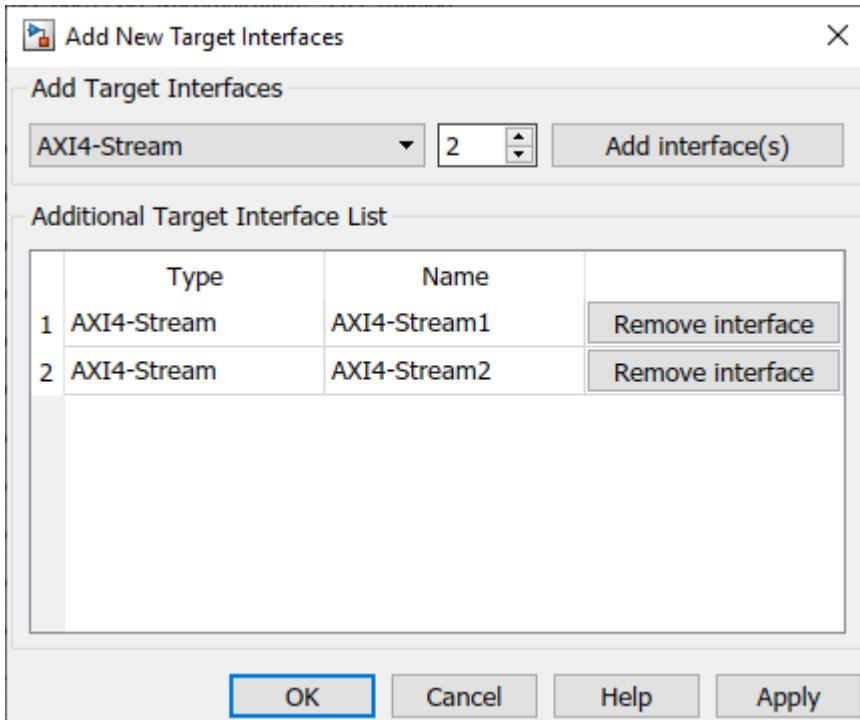
Specify Multiple AXI4 Interfaces in Generic IP Core Generation Workflow

To specify more than one AXI4-Stream, AXI4-Stream Video, or AXI4 Master channel:

- 1 In the **Set Target Device and Synthesis Tool** task, select IP Core Generation as the **Target workflow** and Generic Xilinx Platform or Generic Altera Platform as the **Target platform**. Run this task.
- 2 To add multiple target interfaces, in the **Set Target Interface** task, on the **Target Platform Interfaces** section of the Target platform interface table, select **Add more ...**.

Target platform interface table					
Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin	
x_in_data	Input	sfix16_E...	AXI4-Stream Slave	Data	
x_in_valid	Input	boolean	No Interface Specified	Valid	
x_in_ready	Input	boolean	AXI4		
x_in_data1	Input	sfix16_E...	AXI4-Stream Master		
x_in_valid1	Input	boolean	AXI4-Stream Slave		
x_in_ready1	Input	boolean	External Port		
y_out_data	Output	sfix32_E...	AXI4 Master Read	Data	
y_out_valid	Output	boolean	AXI4 Master Write		
y_out_ready	Output	boolean	AXI4-Stream1 Master		
y_out_data1	Output	sfix32_E...	AXI4-Stream1 Slave		
y_out_valid1	Output	boolean	Add more...		
y_out_ready1	Output	boolean	AXI4-Stream Master	Data	
				Valid	
				Ready (optional)	
				Data	
				Valid	
			No Interface Specified		

- 3 You can then add more interfaces in the Add New Target Interfaces dialog box. Specify the type of interface you want to add, the number of interfaces, and a custom name for each additional interface.



After you apply the settings, the interfaces you created appear in the Target platform interface table. After you run this task, the additional interfaces specified are saved on the DUT subsystem as the HDL block property **AdditionalTargetInterfaces**.

If you modify the additional interfaces that were already mapped to DUT ports such as deleting or renaming an interface that was already mapped, the previous interface mapping information might be lost. The ports then become unmapped to interfaces and the **Target platform interfaces** section displays **No interface specified**. Therefore, if you make changes to the additional target interfaces, verify that the DUT ports are mapped to the correct target interfaces.

Specify Multiple AXI4 Interfaces in Custom Reference Designs

When you create your own custom reference design, you can add multiple AXI4-Stream, AXI4-Stream Video, and AXI4 Master interfaces. Depending on the interface type you want to add, specify additional interfaces by using the `addAXI4StreamInterface`, `addAXI4StreamVideoInterface`, or `addAXI4MasterInterface` methods of the `hdlcoder.ReferenceDesign` class.

To add more interfaces, in the `plugin_rd` file, call the interface method each time you want to add more interfaces. This example shows how to add two AXI4-Stream interfaces.

```
function hRD = plugin_rd()
% Reference design definition

% Copyright 2017-2019 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Multiple Interface Reference Design';
hRD.BoardName = 'ZedBoard';
```

```
% Tool information
hRD.SupportedToolVersion = {'2019.1'};

% ...
% ...

% Add AXI4-Stream interface 1
hRD.addAXI4StreamInterface (...
    'MasterChannelEnable', true, ...
    'SlaveChannelEnable', true, ...
    'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
    'SlaveChannelConnection', 'axi_dma_mm2s/M_AXIS_MM2S', ...
    'MasterChannelDataWidth', 32, ...
    'SlaveChannelDataWidth', 32, ...
    'InterfaceID', 'AXI4-Stream1');

% Add AXI4-Stream interface 2
hRD.addAXI4StreamInterface (...
    'MasterChannelEnable', true, ...
    'SlaveChannelEnable', true, ...
    'MasterChannelConnection', 'ADC/S_AXIS_S2MM', ...
    'SlaveChannelConnection', 'DAC/M_AXIS_MM2S', ...
    'MasterChannelDataWidth', 32, ...
    'SlaveChannelDataWidth', 32, ...
    'InterfaceID', 'AXI4-Stream2');

% ...
% ...
```

When you run the **IP Core Generation** workflow and target the custom reference design **Multiple Interface Reference Design**, in the **Set Target Interface** task, you can map the DUT ports to AXI4-Stream1 Master and Slave channels and AXI4-Stream2 Master and Slave channels.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
x_in_data	Import	sfix16_E...	AXI4-Stream1 Slave	Data
x_in_valid	Import	boolean	AXI4-Stream1 Slave	Valid
x_in_ready	Import	boolean	AXI4-Stream1 Master	Ready (optional)
h_in1	Import	sfix16_E...	AXI4-Lite	x"100"
h_in2	Import	sfix16_E...	AXI4-Lite	x"104"
h_in3	Import	sfix16_E...	AXI4-Lite	x"108"
h_in4	Import	sfix16_E...	AXI4-Lite	x"10C"
x_in_data1	Import	sfix16_E...	AXI4-Stream2 Slave	Data
x_in_valid1	Import	boolean	AXI4-Stream2 Slave	Valid
x_in_ready1	Import	boolean	AXI4-Stream2 Master	Ready (optional)
h_in5	Import	sfix16_E...	AXI4-Lite	x"110"
h_in6	Import	sfix16_E...	AXI4-Lite	x"114"

Note When you target your own custom reference design and map the additional interfaces to DUT ports in the **Set Target Interfaces** task, the additional interfaces are not saved on the model as the **AdditionalTargetInterfaces** HDL block property. Instead, the additional interfaces are saved on the custom reference design in the `plugin_rd.m` file.

You can also dynamically customize the reference design to specify the number of interfaces you want to add and the interface properties.

- 1 In the `plugin_rd` file, create a reference design parameter for the number of additional interfaces you want to add.
- 2 Create a callback function that has different choices for the number of interfaces you want to add and then reference the function in the `plugin_rd` file by using the `CustomizeReferenceDesignFcn` method of the `hdlcoder.ReferenceDesign` class.

To learn more, see “Customize Reference Design Dynamically Based on Reference Design Parameters” on page 41-54.

Ready Signal Mapping for Multiple Streaming Interfaces

When you use a single streaming channel, HDL Coder automatically generates the Ready signal and the associated back pressure logic.

If you use multiple streaming channels, HDL Coder does not automatically generate the back pressure logic. In this case, the Ready signal is generated but the master Ready signal at the input is ignored and the slave Ready signal at the output is tied to high value. The absence of a back pressure logic can result in samples being dropped. If you want your design to apply back pressure on the Slave interface or respond to back pressure from the Master interface, you must model the Ready signal for each additional interface and then map the port to the Ready signal for that interface.

When you do not model, the **Set Target Interface** task displays a warning that provides names of interfaces that require a Ready port. If your design does not need to apply or respond to back pressure, you can ignore this warning and you do not have to model the Ready signal.

When using multiple AXI4-Stream interfaces, if you want your design to apply back pressure on the Slave interface or respond to back pressure from the Master interface, you must model the Ready signal for each additional interface and then map the port to the Ready signal for that interface. To learn how the back pressure logic is generated for a single streaming channel and how to model the Ready signal, see “Ready Signal (Optional)” on page 41-12.

Restrictions

- When you run the generic IP Core Generation workflow, you can specify the interface type and a custom interface ID for each additional interface. Other interface properties such as the data width cannot be customized and use default values. When you create your own custom reference design, you can customize the interface name and interface properties.
- When mapping your DUT ports to multiple AXI4-Stream interface channels, you can only use scalar ports. Vector ports can have at most one AXI4-Stream Master channel and one AXI4-Stream Slave channel.
- Xilinx Zynq-7000 or Intel Quartus Prime must be your target platform.
- **Processor/FPGA synchronization** must be Free running.

- Xilinx Vivado or Intel Quartus Prime must be your synthesis tool.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Generate Board-Independent HDL IP Core from Simulink Model” on page 40-19

See Also

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

Running Audio Filter with Multiple AXI4-Stream Channels on ZedBoard

This example shows how to model an audio system with multiple AXI4-Stream channels and deploy it on a ZedBoard™ by using an audio reference design.

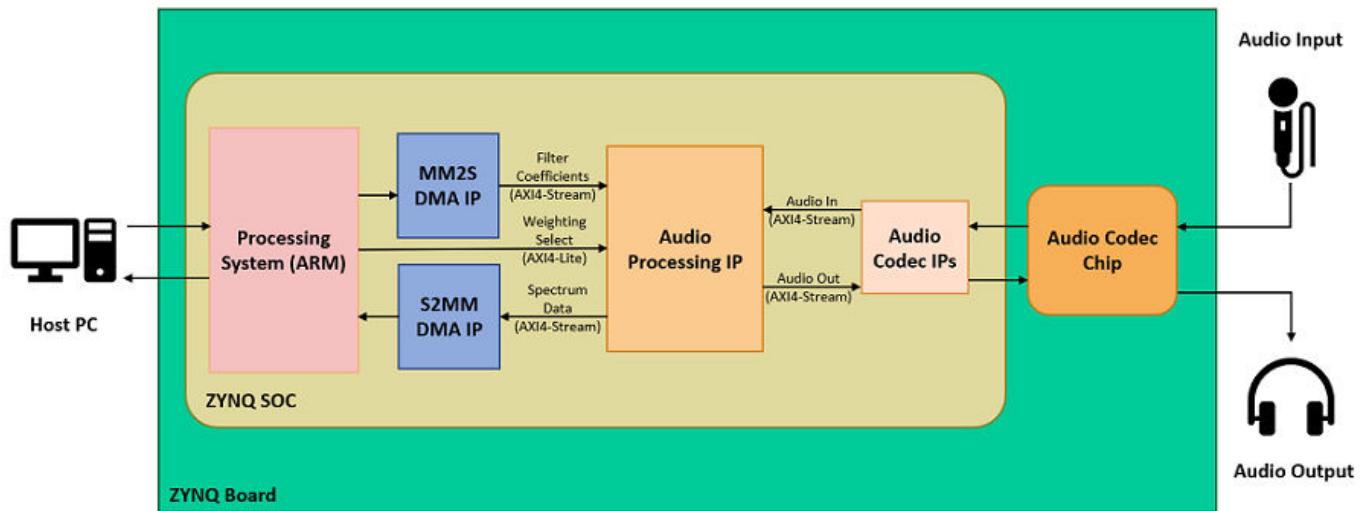
Introduction

In this example, you model a programmable audio filter with spectrogram using multiple AXI4-Stream channels and advanced AXI4-Stream signals Ready and TLAST. One AXI4-Stream channel transfers data between the filter and the audio codec. The other AXI4-Stream channel interfaces with the Processing System to program filter coefficients and transmit spectrogram data to the host computer for analysis.

You can then run the IP Core Generation workflow to generate an HDL IP core and deploy the algorithm on a ZedBoard by using an audio reference design.

System Architecture

This figure shows the high-level architecture of the system.



The **Audio Codec IPs** configure the audio codec and transfer audio data between the ZedBoard and audio codec. The Audio Processing IP generated by HDL Coder™ performs filtering and spectrum analysis. The DMA IPs transfer AXI4-Stream data between the Processing System and the FPGA. The stream data transmitted from the Processing System through the **MM2S DMA IP** programs the filter coefficients on the FPGA. The stream data received by the Processing System through the **S2MM DMA IP** contains the spectrogram data computed on the FPGA. The Processing System also configures the weighting curve for spectrum analysis using an AXI4-Lite interface.

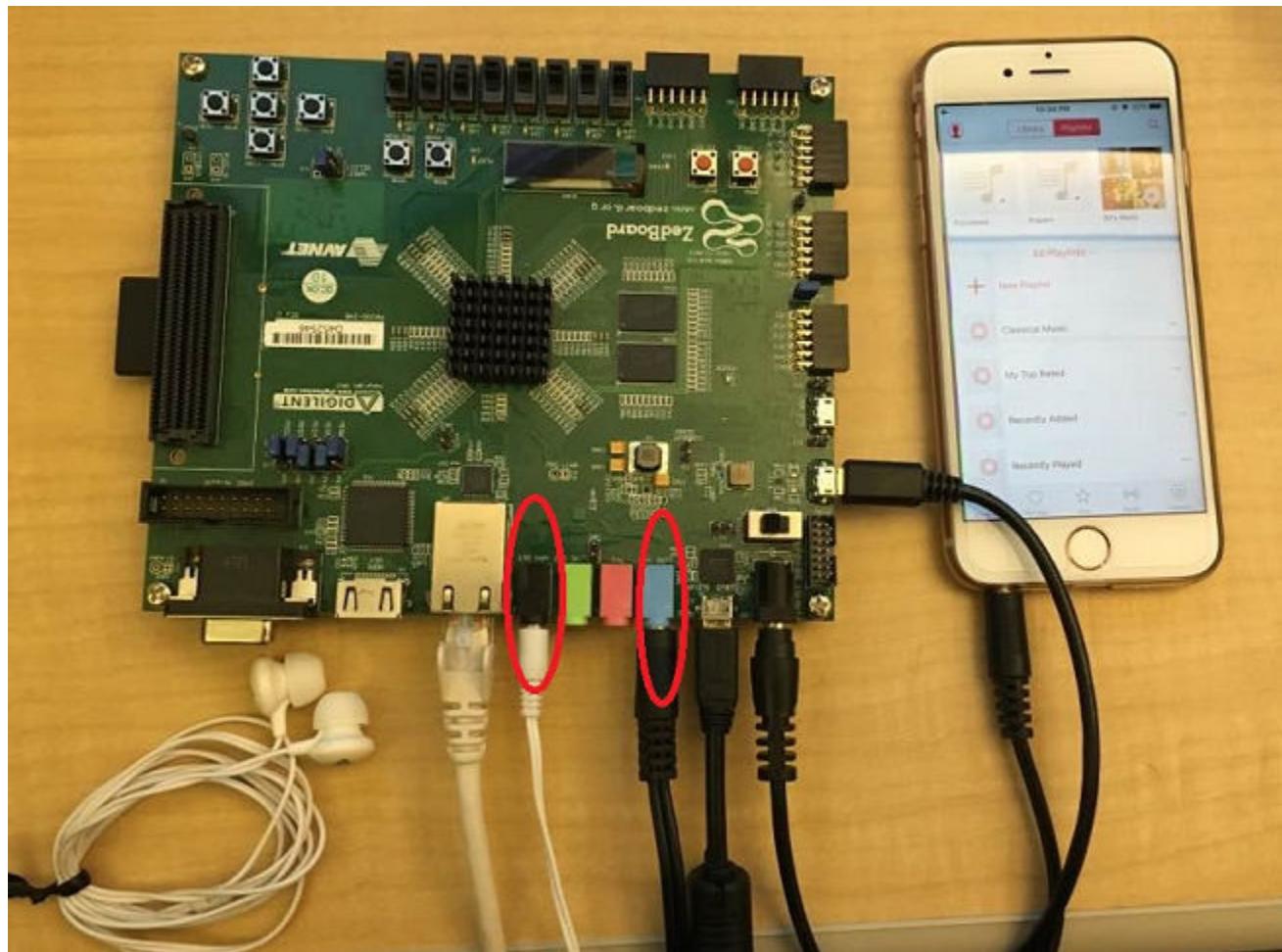
Prerequisites

This example extends the audio filter on live input example to use multiple streaming channels. To learn about the example that uses a single streaming channel, see “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 41-126.

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder Support Package for Xilinx® Zynq® Platform
- Embedded Coder® Support Package for Xilinx Zynq Platform
- Xilinx Vivado® Design Suite latest version, as mentioned in “HDL Language Support and Supported Third-Party Tools and Hardware”
- ZedBoard

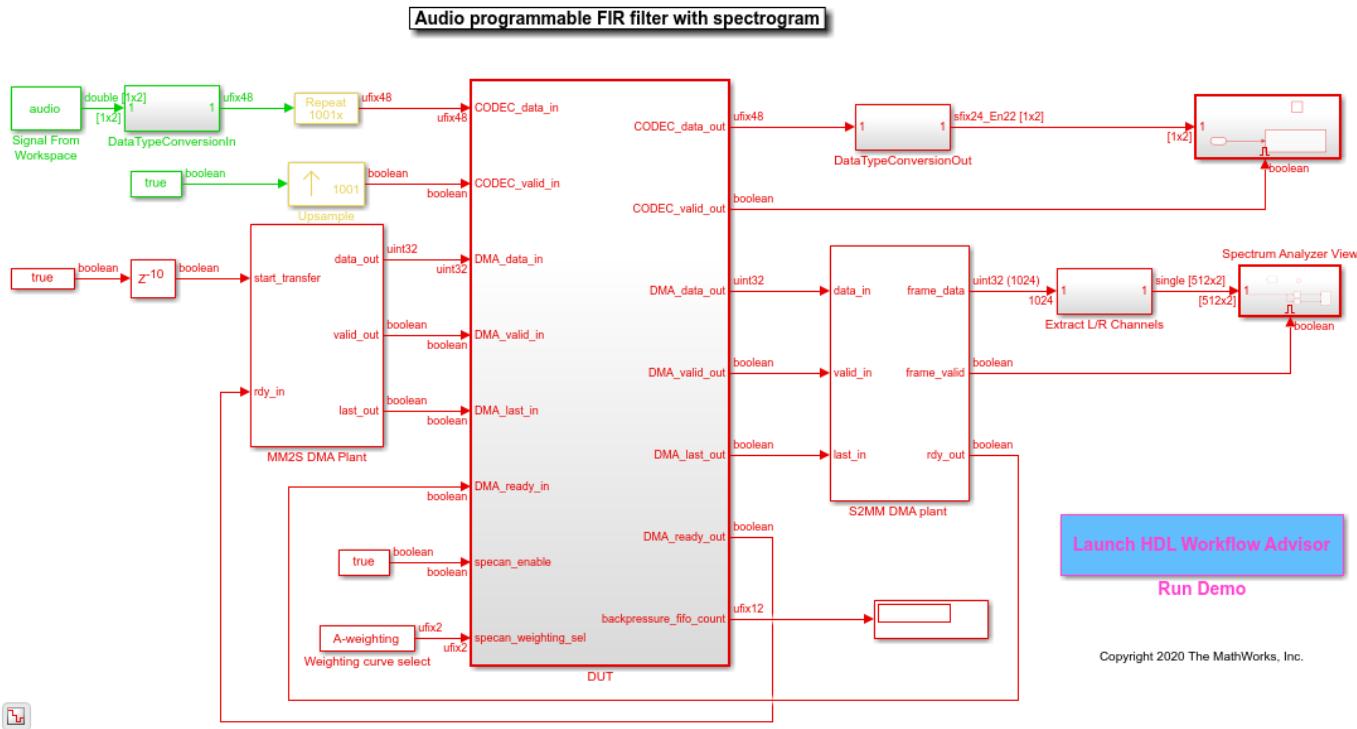
To setup the ZedBoard, refer to the *Set up Zynq hardware and tools* section in the “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example. Connect an audio input from a mobile or an MP3 player to the **LINE IN** jack and either earphones or speakers to the **HPH OUT** jack on the ZedBoard as shown below.



Model Audio Processing Algorithm

Open the model `hdlcoder_audio_filter_multistream`.

```
open_system('hdlcoder_audio_filter_multistream')
set_param('hdlcoder_audio_filter_multistream', 'SimulationCommand', 'Update')
```



The model contains the DUT subsystem for audio processing, source and sink blocks for simulating the audio, and plant models for DMAs that transfer stream data between the Processing System and FPGA.

Rate Considerations

For audio applications running on the FPGA, the FPGA clock rate is several times faster than the audio sample rate. The ratio of the FPGA clock rate to the audio data sample rate is the **Oversampling factor**. In this example, the **Oversampling factor** is modeled by using Repeat and Upsample blocks.

Modeling your design at the FPGA clock rate allows you to optimize resource usage on the target hardware platform by leveraging idle clock cycles and reusing various components. The audio application illustrated in this example uses an audio sample rate of 48kHz and an FPGA clock rate of 96MHz. The **Oversampling factor** in this case is 2000. Such a large value of **Oversampling factor** slows down the Simulink simulation significantly.

To reduce the simulation time, instead of using the **Oversampling factor** setting, you can model your design at the minimum **Oversampling factor** that is required by the design. The minimum required **Oversampling factor** for the design can be determined by the length of the audio filter, which is 1001. This value reduces the simulation time by half and provides sufficient idle cycles between the data samples for the serial filter logic.

Audio Filter

Inside the DUT subsystem, the FIR filter processes data from the audio codec AXI4-Stream channel. The filter coefficients are generated in MATLAB® and programmed by using the second AXI4-Stream interface that interfaces with the Processing System. The filtered audio output is streamed back to the audio codec.

The audio filter is a fully serial implementation of an FIR filter. This filter structure is best suited for audio applications that require large **Oversampling factor** because the filter uses a multiply accumulate (MAC) operation for each channel. The filter also uses RAM blocks to implement the data delay line and the coefficient source. This implementation saves area by avoiding the high slice logic usage of high-order filters.

Spectrum Analyzer

The audio signal is fed into a spectrum analyzer after passing through the FIR filter. The spectrum analyzer computes the FFT of the filtered signal, applies a weighting function, and converts the result to dBm. You can program the type of weighting to be performed by using the AXI4-Lite interface as either No-weighting, A-weighting, C-weighting, or K-weighting. The actual weighting functions are implemented using lookup tables that have been generated by using Audio Toolbox™ function `weightingFilter`.

Model AXI4-Stream Interfaces

The model contains two AXI4-Stream interfaces. One AXI4-Stream interface communicates with the audio codec. The other AXI4-Stream interface communicates with the Processing System through the DMAs. The audio codec interface only requires the `Data` and `Valid` signals. The DMA interface, on the other hand, additionally uses the `Ready` and `TLAST` signals of the AXI4-Stream protocol.

To learn more about the signals used in AXI4-Stream modeling, see “Model Design for AXI4-Stream Interface Generation” on page 41-10.

Ready Signal

In an AXI4-Stream interface, you use the `Ready` signal to apply or respond to back pressure. The model uses the `Ready` signal on the AXI4-Stream Master channel from the FPGA to the Processing System to respond to back pressure from the DMA. When the downstream DMA cannot receive more spectrogram samples, it de-asserts the input `Ready` signal on the AXI4-Stream Master channel. To ensure that the spectrogram samples are not dropped, the model buffers the data in a FIFO until the `Ready` signal is asserted, indicating that the DMA is ready to receive samples again.

The AXI4-Stream Slave channel from the Processing System to the FPGA does not have to apply back pressure, and hence its `Ready` signal is always asserted. The audio codec does not process back pressure and does not use its `Ready` signal on either channel. To learn more about the `Ready` signal in AXI4-Stream modeling, see “Ready Signal (Optional)” on page 41-12.

TLAST Signal

The `TLAST` signal is used to indicate the last sample of a frame. The model uses the `TLAST` signal on the AXI4-Stream Slave channel as an indicator that it has received a full set of filter coefficients. On the AXI4-Stream Master channel, the `TLAST` signal is used to indicate the end of a spectrum analyzer frame. To learn more about the `TLAST` signal in AXI4-Stream modeling, see “Other Protocol Signals (optional)” on page 41-14.

Customize the Model for ZedBoard

To implement this model on the ZedBoard, you must first have a reference design in Vivado that receives audio input on the ZedBoard and transmits the processed audio data out of the ZedBoard. For details on how to create a reference design which interfaces with the audio codec on the ZedBoard, see “Authoring a Reference Design for Audio System on a Zynq Board” on page 41-170. This example extends the reference design in that example by adding DMA IPs for communication with the Processing System.

In the reference design, left and right channel audio data are combined to form a single channel such that the lower 24 bits form the left channel and upper 24 bits form the right channel. In the Simulink® model shown above, CODEC_data_in is split into left and right channels. Filtering is done on each channel individually. The channels are then concatenated to form a single channel for CODEC_data_out.

Generate HDL IP Core with AXI4-Stream Interfaces

Next, you can start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the Getting Started with HW/SW Co-design Workflow for Xilinx Zynq Platform example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', ...
    'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat');
```

2. Add both the IP repository folder and the ZedBoard registration file to the MATLAB path using following commands:

```
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ipcore'));
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ZedBoard'));
```

3. Open the HDL Workflow Advisor from the DUT subsystem, hdlcoder_audio_filter_multistream/DUT or double-click the **Launch HDL Workflow Advisor** box in the model.

The target interface settings are already saved for ZedBoard in this example model, so the settings in tasks **1.1** to **1.3** are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the Save Target Hardware Settings in Model example.

4. Run the **Set Target Device and Synthesis Tool** task.

In this task, **IP Core Generation** is selected for **Target workflow**, and **ZedBoard** is selected for **Target platform**.

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: IP Core Generation

Target platform: ZedBoard

Synthesis tool: Xilinx Vivado

Tool version: 2019.1.1

Launch Board Manager

Family: Zynq

Device: xc7z020

Package: clg484

Speed: -1

Project folder: hdl_prj

Run This Task

Result: Passed

Passed Set Target Device and Synthesis Tool.

5. Run the **Set Target Reference Design** task. Audio system with DMA Interface is selected as the **Reference Design**.

1.2. Set Target Reference Design

Analysis (^Triggers Update Diagram)

Set target reference design options

Input Parameters

Reference design: Audio System with AXI DMA interface

Reference design tool version: 2019.1

Ignore tool version mismatch

Reference design parameters

Parameter	Value
Insert JTAG MATLAB as AXI Master(H...)	off

Run This Task

Result: Passed

Passed Set Target Reference Design.

6. Run the **Set Target Interface** task.

In this task, the ports of the DUT subsystem are mapped to the IP Core interfaces. The audio codec ports are mapped to the Audio Interface and the DMA ports are mapped to the AXI DMA interface. These are both AXI4-Stream interfaces. The AXI4-Stream interface communicates in master/slave

mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, it is assigned to an AXI4-Stream Slave interface, and if a data port is output port, it is assigned to an AXI4-Stream Master interface. The exception to this is the **Ready** signal. The AXI4-Stream Master Ready signal is an input to the model, and the AXI4-Stream Slave Ready signal is an output of the model. The spectrum analyzer control ports are mapped to AXI4-Lite.

1.3. Set Target Interface

Analysis (^Triggers Update Diagram)

Set target interface for HDL code generation

Input Parameters

Processor/FPGA synchronization: Free running

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
CODEC_data_in	Input	ufix48	Audio Interface Slave	Data
CODEC_valid_in	Input	boolean	Audio Interface Slave	Valid
DMA_data_in	Input	uint32	AXI DMA Slave	Data
DMA_valid_in	Input	boolean	AXI DMA Slave	Valid
DMA_last_in	Input	boolean	AXI DMA Slave	TLAST (optional)
DMA_ready_in	Input	boolean	AXI DMA Master	Ready (optional)
specan_enable	Input	boolean	AXI4-Lite	x"100"
specan_weighting_sel	Input	ufix2	AXI4-Lite	x"104"
CODEC_data_out	Output	ufix48	Audio Interface Master	Data
CODEC_valid_out	Output	boolean	Audio Interface Master	Valid
DMA_data_out	Output	uint32	AXI DMA Master	Data
DMA_valid_out	Output	boolean	AXI DMA Master	Valid

Run This Task

Result: ✓ Passed

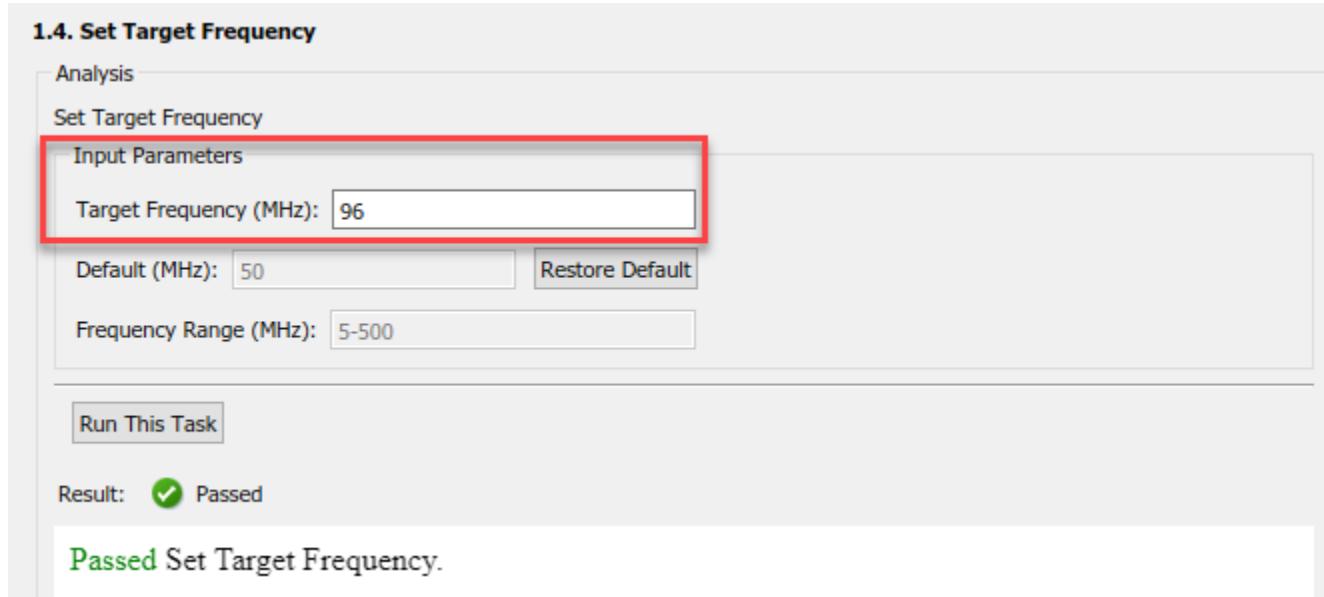
Warning Auto-generation of the Ready signal on AXI4-Stream interfaces has been disabled because multiple AXI4-Stream interfaces are in use. When multiple AXI4-Stream interfaces are in use and not all interfaces assign a port to the Ready signal, HDL Coder generates the signal, but does not generate back pressure logic, which can result in samples being dropped. Interfaces without their Ready port assigned are: Audio Interface Master, Audio Interface Slave. It is recommended that you model the Ready port on these interfaces if your design needs to apply back pressure on the Slave interface or respond to back pressure on the Master interface.

Passed Set Target Interface Table.

Running this task issues a warning that auto-generation of the **Ready** signal is disabled, and that the **Audio Interface** does not assign a **Ready** port. You can ignore the warning for this design, because back pressure has already been accounted for. Namely, the design addressed back pressure on the DMA interface by using a FIFO. On the audio codec interface, back pressure cannot be applied, so no **Ready** signal logic is needed.

7. In the **Set Target Frequency** task, set the **Target Frequency (MHz)** to 96. Run this task.

This target frequency value makes the **Oversampling factor** an even integer relative to the audio sample rate of 48kHz.



8. Right-click the **Generate RTL Code and IP Core** task and select **Run to Selected Task**.

You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

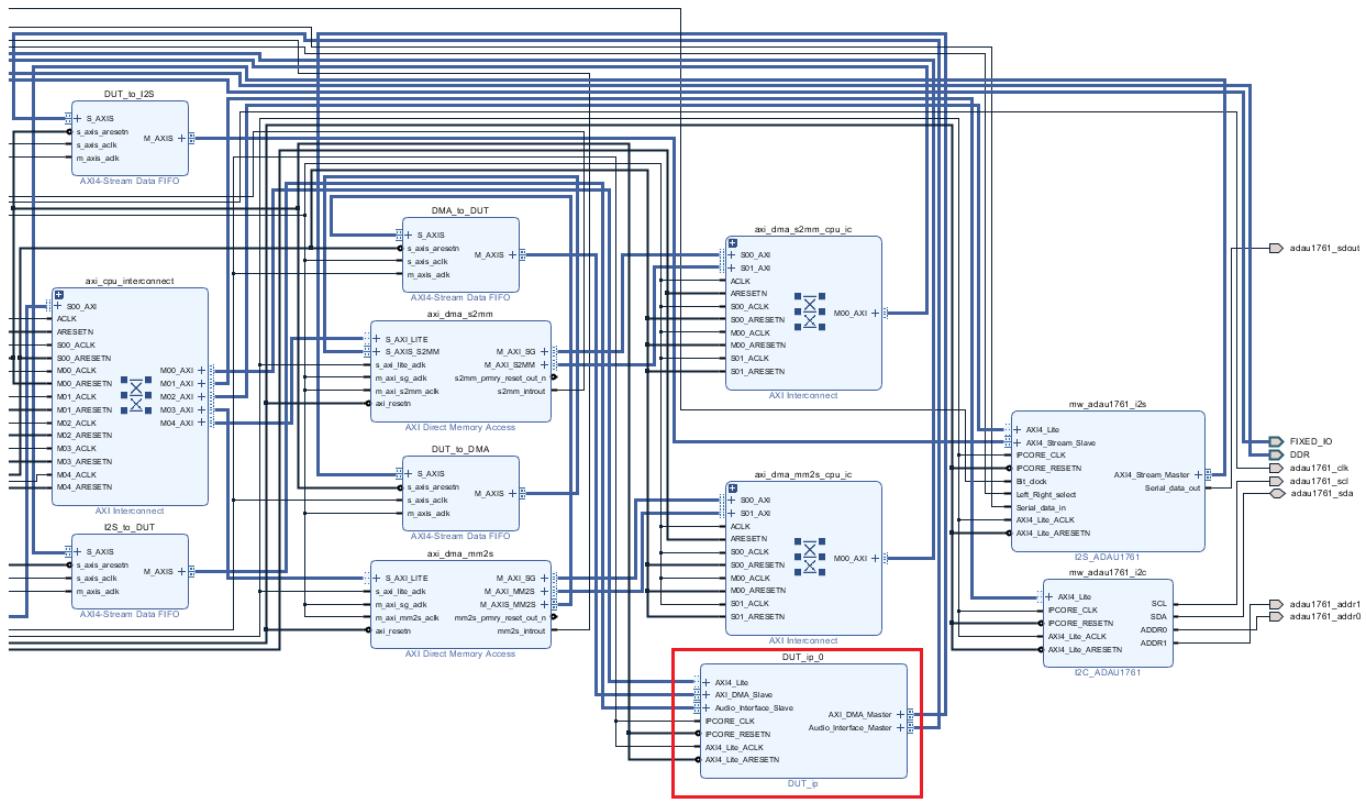
Integrate IP into AXI4-Stream Audio-Compatible Reference Design

Next, in the HDL Workflow Advisor, you run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq® hardware.

1. Run the **Create Project** task.

This task inserts the generated IP core into the **Audio System with AXI DMA Interface** reference design. As shown in the first diagram, this reference design contains the IPs to handle streaming audio data in and out of ZedBoard, and for streaming data in and out of the Processing System. The generated project is a complete ZedBoard design. It includes the algorithm part, which is the generated DUT algorithm IP, and the platform part, which is the reference design.

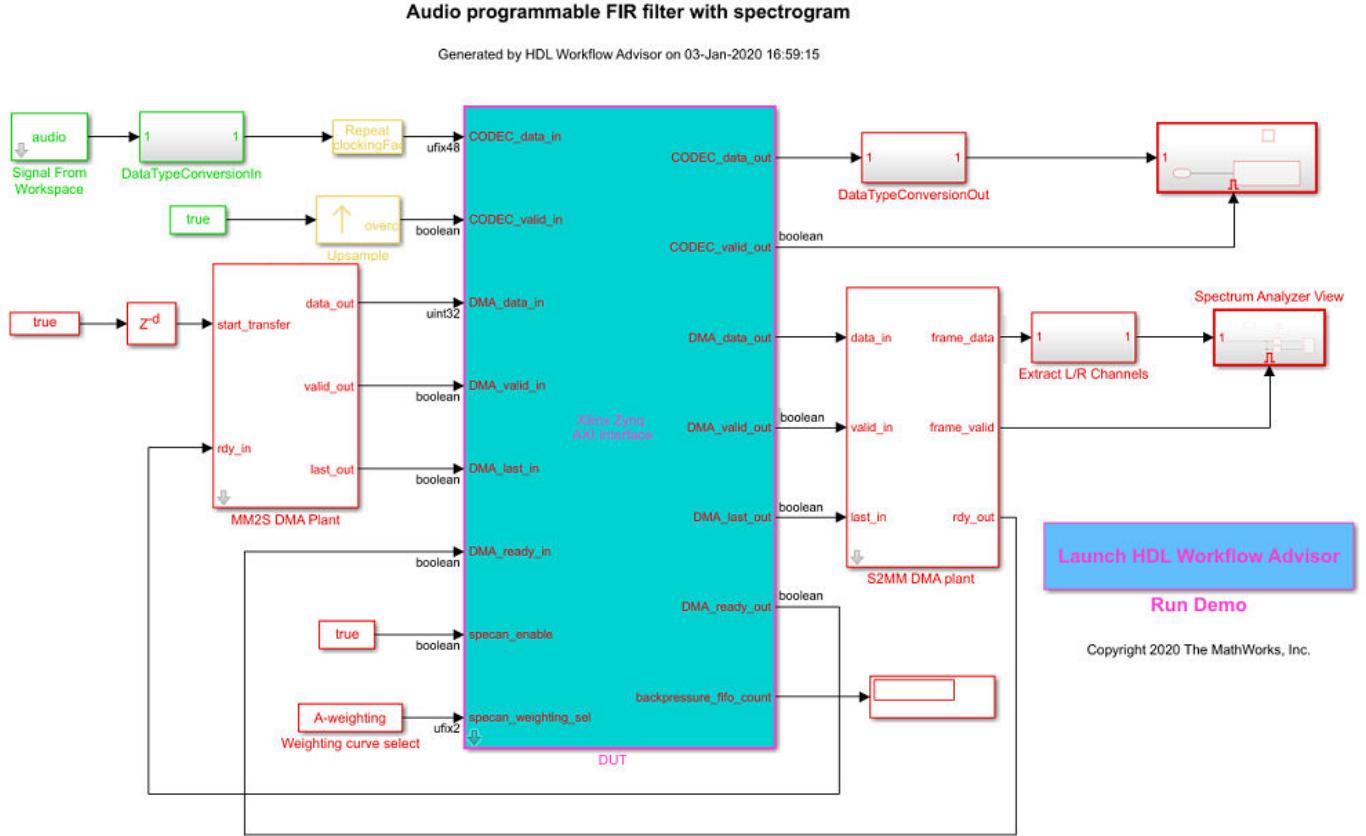
2. Click the link in the **Result** pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, other audio processing IPs and the Zynq processor.



3. In the HDL Workflow Advisor, run the remaining tasks to generate the software interface model, and build and download the FPGA bitstream. Choose **Download** programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the Zynq board. Your design is then automatically reloaded when you power cycle the Zynq board.

Generate ARM Executable to Tune Parameters on FPGA Fabric

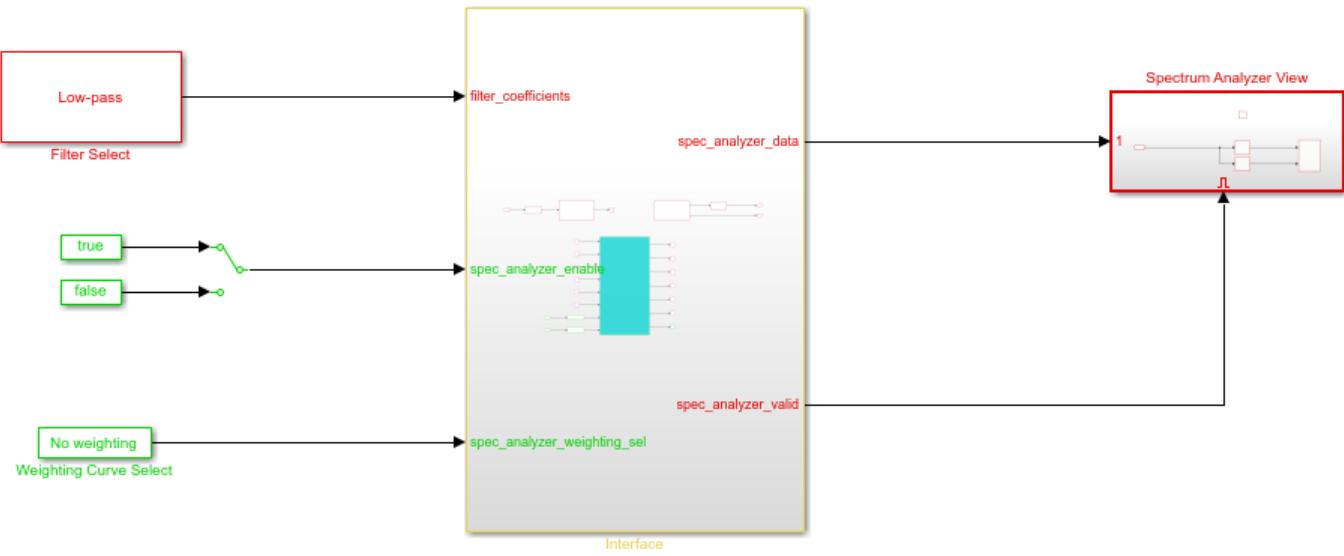
In task **Generate Software Interface Model**, a software interface model is generated.



In the generated model, AXI4-Lite driver blocks have been automatically added. However, AXI4-Stream driver blocks cannot be automatically generated, because the driver blocks expect vector inputs on the software side, but the DMA DUT ports are scalar ports. For details on how to update the software interface model with the correct driver blocks, refer to “Getting Started with AXI4-Stream Interface in Zynq Workflow” on page 41-137.

For this example, you use an updated software interface model. To open this model, run:

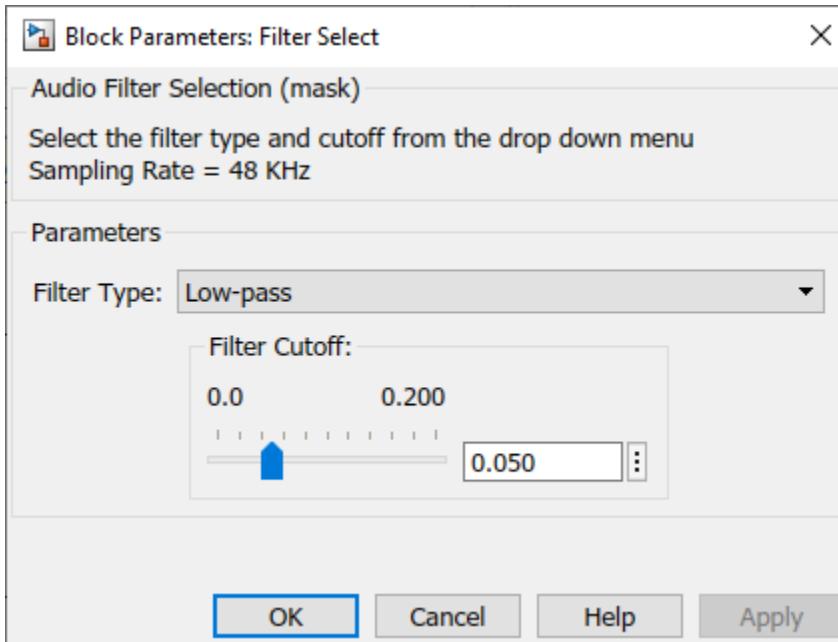
```
open_system('hdlcoder_audio_filter_multistream_sw');
```



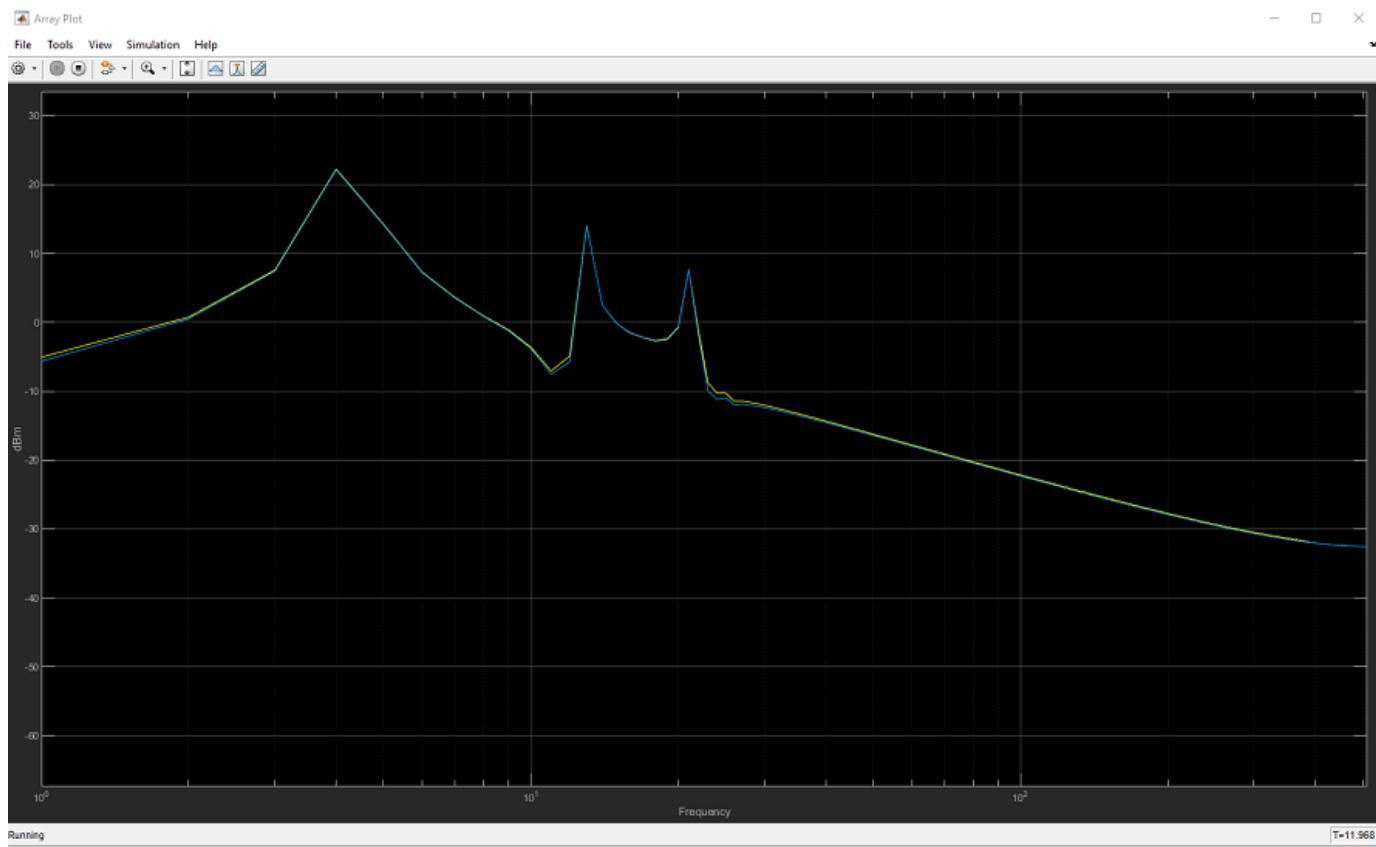
Copyright 2020 The MathWorks, Inc.

To tune the parameters:

1. Click the **Monitor & Tune** button on the **Hardware** tab of model toolbar. Embedded Coder builds the model, downloads the ARM® executable to the ZedBoard hardware, executes it, and connects the model to the running executable. While the model is running, different parameters can be tuned.
2. You can select the type of filter by using the **Filter Type** block parameter of the Filter Select block. The filter coefficients are calculated in this block using the **fir1** function from Signal Processing Toolbox™. The coefficients are sent from the Processing System to the FPGA by using the AXI4-Stream IIO Write block, which communicates through the **MM2S DMA IP**.



3. The weighting curve used by the spectrum analyzer can be selected using the **Curve** block parameter of the Weighting Curve Select block. The selection is sent to the from the Processing System to the FPGA using the AXI4-Lite interface.
4. The spectrum analyzer output can be viewed in the Array Plot. Select a different filter type or modify the weighting curve and observe how the spectrum data changes.



The filtered audio output can be heard by plugging earphones or speakers to **HPH OUT** jack on the ZedBoard.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Custom IP Core Generation” on page 40-10
- “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 41-17

See Also

Multirate IP Core Generation

This example shows HDL Coder™ supports designs with multiple sample rates when you run the IP Core Generation workflow.

If you are only using AXI4 slave interfaces such as AXI4 or AXI4-Lite, and when you use Free running for **Processor/FPGA Synchronization**, you can use multiple sample rates in your design without restrictions.

When you map the interface ports to AXI4-Stream, AXI4-Stream Video, or AXI4 Master interfaces, to use multiple sample rates, make sure that the DUT ports that map to the AXI4 interfaces run at the fastest rate of the design after HDL code generation.

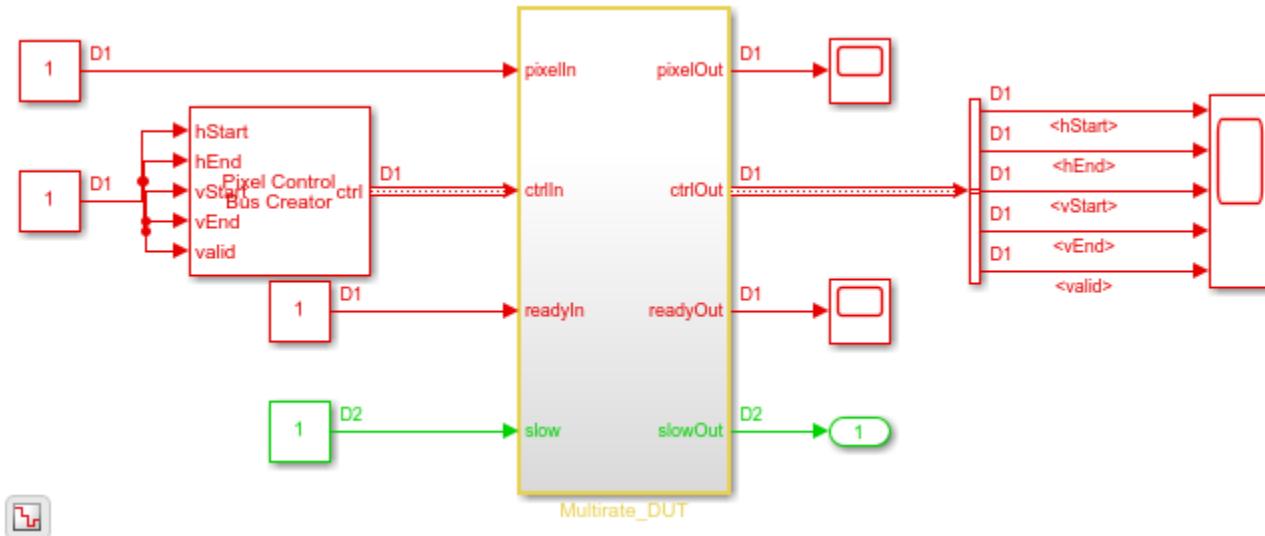
These examples illustrate how you can model your design with multiple sample rates when using AXI4-Stream, AXI4-Stream Video, or AXI4-Master Master interfaces.

Run Part of Design at Slower Rate

You can run part of the design at a slower rate while making sure that the DUT ports that map to the interface run at the fastest rate. This example illustrates mapping to AXI4-Stream Video interfaces but you can map to AXI4-Stream or AXI4 Master interfaces by using this approach.

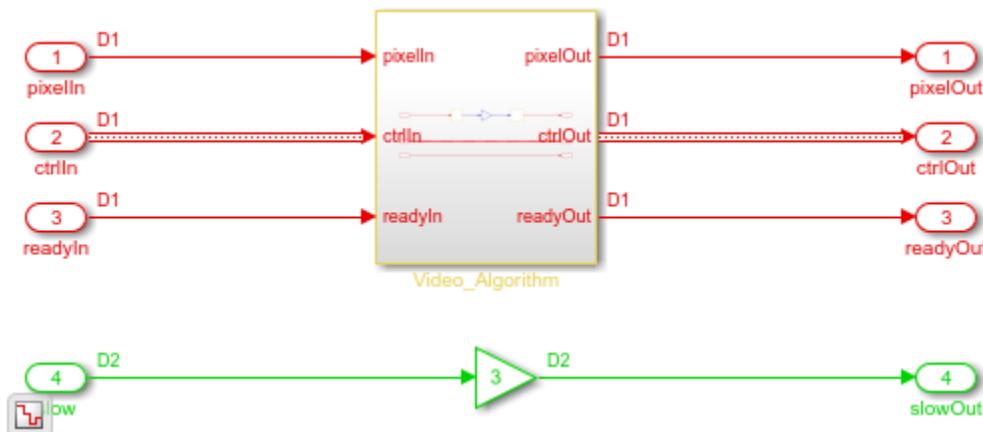
For an example, open the model `hdlcoder_axi_video_multirate`.

```
load_system('hdlcoder_axi_video_multirate')
set_param('hdlcoder_axi_video_multirate','SimulationCommand','update')
open_system('hdlcoder_axi_video_multirate')
```



In this model, the DUT ports corresponding to inputs and outputs of the `Video_Algorithm` run at the fastest rate.

```
open_system('hdlcoder_axi_video_multirate/Multirate_DUT')
```



These ports can therefore map to AXI4-Stream Video interfaces. Part of the design running outside this algorithm corresponding to input `slow` and output `slowOut` running at a slower rate can map to AXI4 or AXI4-Lite interfaces. This figure shows an example of the target platform interface mapping for this model.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
pixelIn	Import	uint16	AXI4-Stream Video Slave	Pixel Data
ctrlIn	Import	bus	AXI4-Stream Video Slave	Pixel Control Bus
readyIn	Import	boolean	AXI4-Stream Video Master	Ready (optional)
slow	Import	uint16	AXI4-Lite	x"100"
pixelOut	Output	uint16	AXI4-Stream Video Master	Pixel Data
ctrlOut	Output	bus	AXI4-Stream Video Master	Pixel Control Bus
readyOut	Output	boolean	AXI4-Stream Video Slave	Ready (optional)
slowOut	Output	uint16	AXI4-Lite	x"104"

Note: To use the Pixel Control Bus Creator and Pixel Control Bus Selector blocks, you must have Vision HDL Toolbox™ installed. If you do not have Vision HDL Toolbox, use Bus Creator and Bus Selector blocks instead.

See also “Model Design for AXI4-Stream Video Interface Generation” on page 41-69.

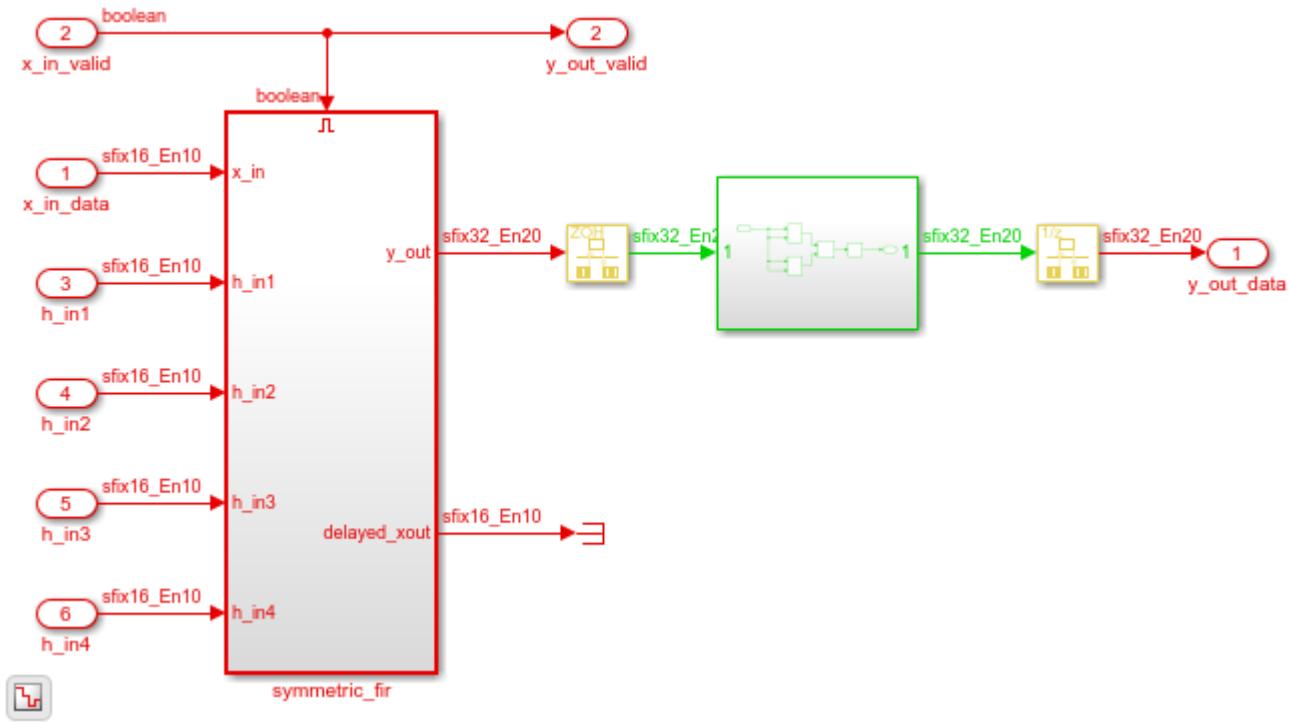
Apply Optimizations to Part of Design Running at Slow Rate

With multirate support, you can apply optimizations such as resource sharing to a part of the design running at a slower rate. Make sure that the optimizations do not introduce a faster rate in your

Simulink™ model. This example illustrates mapping to AXI4-Stream interfaces but you can map to AXI4-Stream Video or AXI4 Master interfaces by using this approach.

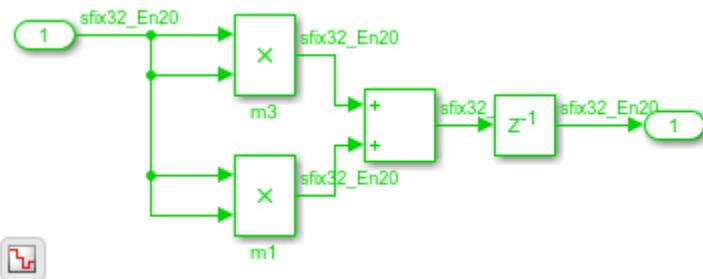
For an example, open the model `hdlcoder_axi_multirate_sharing`

```
load_system('hdlcoder_axi_multirate_sharing')
set_param('hdlcoder_axi_multirate_sharing','SimulationCommand','update')
open_system('hdlcoder_axi_multirate_sharing/DUT')
```



In this model, the Subsystem contains a simple multiply-add algorithm running at a slower rate.

```
open_system('hdlcoder_axi_multirate_sharing/DUT/Subsystem')
```



Resource sharing can be applied to this part of the design. To see the parameters saved on this Subsystem, run `hdlsaveparams`.

```
hdlsaveparams('hdlcoder_axi_multirate_sharing/DUT/Subsystem')
```

```
%% Set Model 'hdlcoder_axi_multirate_sharing' HDL parameters
hdlset_param('hdlcoder_axi_multirate_sharing', 'HDLSubsystem', 'hdlcoder_axi_multirate_sharing/DU...')
```

```

hdlset_param('hdlcoder_axi_multirate_sharing', 'ReferenceDesign', 'Default system with AXI4-Stream');
hdlset_param('hdlcoder_axi_multirate_sharing', 'ResetType', 'Synchronous');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisTool', 'Xilinx Vivado');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolChipFamily', 'Zynq');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolDeviceName', 'xc7z020');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolPackageName', 'clg484');
hdlset_param('hdlcoder_axi_multirate_sharing', 'SynthesisToolSpeedValue', '-1');
hdlset_param('hdlcoder_axi_multirate_sharing', 'TargetDirectory', 'hdl_prj\hdlsrc');
hdlset_param('hdlcoder_axi_multirate_sharing', 'TargetFrequency', 50);
hdlset_param('hdlcoder_axi_multirate_sharing', 'TargetPlatform', 'ZedBoard');
hdlset_param('hdlcoder_axi_multirate_sharing', 'Workflow', 'IP Core Generation');

% Set SubSystem HDL parameters
hdlset_param('hdlcoder_axi_multirate_sharing/DUT/Subsystem', 'SharingFactor', 3);

```

You can map the DUT interface ports to AXI4-Stream Master or AXI4-Stream Slave interfaces. This figure shows an example of the target platform interface mapping for this model.

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces		Bit Range / Address / FPGA Pin
			AXI4-Stream Slave	AXI4-Lite	
x_in_data	Import	sfix16_E...	AXI4-Stream Slave	AXI4-Lite	Data
x_in_valid	Import	boolean	AXI4-Stream Slave	AXI4-Lite	Valid
h_in1	Import	sfix16_E...	AXI4-Lite	AXI4-Lite	x"100"
h_in2	Import	sfix16_E...	AXI4-Lite	AXI4-Lite	x"104"
h_in3	Import	sfix16_E...	AXI4-Lite	AXI4-Lite	x"108"
h_in4	Import	sfix16_E...	AXI4-Lite	AXI4-Lite	x"10C"
y_out_data	Outport	sfix32_E...	AXI4-Stream Master	AXI4-Stream Master	Data
y_out_valid	Outport	boolean	AXI4-Stream Master	AXI4-Stream Master	Valid

See also “Model Design for AXI4-Stream Interface Generation” on page 41-10.

See Also

More About

- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- “Custom IP Core Generation” on page 40-10

Board and Reference Design Registration System

In this section...

["Board, IP Core, and Reference Design Definitions" on page 41-39](#)

["Board Registration Files" on page 41-39](#)

["Reference Design Registration Files" on page 41-40](#)

["Predefined Board and Reference Design Examples" on page 41-41](#)

You can define custom boards and custom reference designs so that they are available as target hardware options in the SoC workflow. Custom boards and custom reference designs use the same system that HDL Coder uses for predefined board and reference design targets.

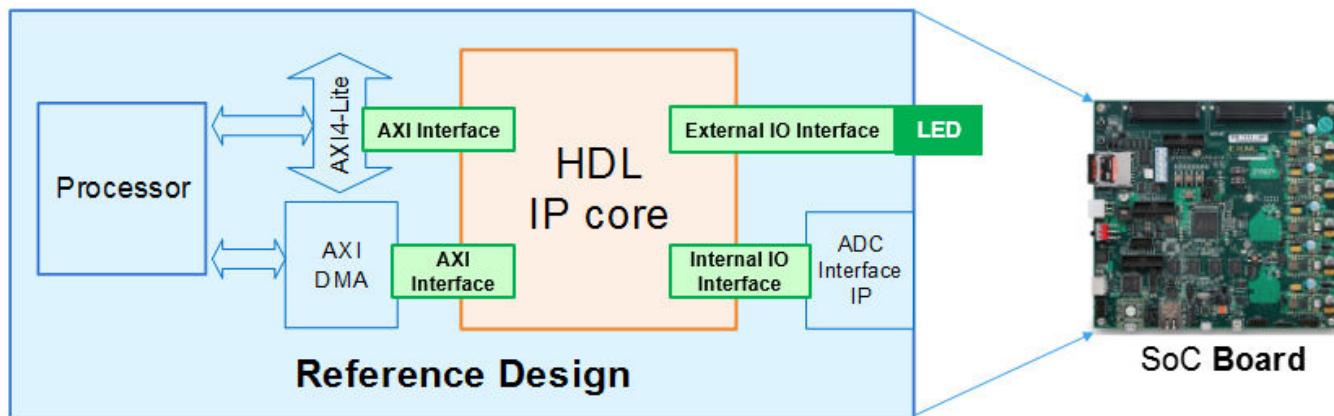
Board, IP Core, and Reference Design Definitions

A reference design is the embedded system design that your generated IP core integrates with. The board is the SoC platform.

For a custom board or custom reference design, you can define different kinds of interfaces:

- *AXI interface*: an interface between your generated IP core and an AXI4 or AXI4-Lite interface.
- *External IO interface*: an interface between your generated IP core and an external interface.
- *Internal IO interface*: an interface between your generated IP core and another IP core in the reference design.

After you integrate your reference design and IP core in an embedded system design project, you can program the board with the embedded system design.



Board Registration Files

To define and register a board, you must have a board definition, a board plugin, and a board registration file.

Board Definition

A board definition is a file that defines the characteristics of a board. You can define more than one custom board.

Board Plugin

A board plugin is a package folder that contains:

- The board definition.
- All reference design plugins that are associated with the board.

A board plugin has one board definition, but can have multiple reference designs.

Board Registration File

A board registration file is always named `hdlcoder_board_customization.m`, and contains a list of board plugins. There can be multiple board registration files on your MATLAB path, but a board plugin cannot be listed in more than one board registration file.

When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_board_customization.m`, and uses the information to populate the target board options. Interfaces you add and define for the board appear as options in the **Target Platform Interface** dropdown list.

Reference Design Registration Files

To define and register a reference design, you must have a reference design definition, a reference design plugin, and a reference design registration file.

Reference Design Definition

A reference design definition is a file that defines the characteristics of a reference design, including its associated board and interfaces. You can define multiple custom reference designs per board.

Reference Design Plugin

A reference design plugin is a package folder that contains:

- The reference design definition.
- Files that are part of the embedded system design project, and are specific to your third-party synthesis tool, including Tcl, project, and design files.

A reference design plugin has one reference design definition and is associated with one board.

Reference Design Registration File

A reference design registration file is always named `hdlcoder_ref_design_customization.m`, and contains a list of reference design plugins for a specific board. There can be multiple reference design registration files for a specific board on your MATLAB path, but a reference design plugin cannot be listed in more than one reference design plugin registration file.

When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_ref_design_customization.m`, and uses the information to populate the reference

design options for each board. Interfaces you add and define for the reference design appear as options in the **Target Platform Interface** dropdown list.

Predefined Board and Reference Design Examples

For examples of working board and reference design definitions, refer to the predefined Altera SoC and Xilinx Zynq board plugins that include predefined reference design plugins:

- *support_package_installation_folder/toolbox/hdlcoder/supportpackages/zynq7000/+ZedBoard/*
- *support_package_installation_folder/toolbox/hdlcoder/supportpackages/zynq7000/+ZynqZC702/*
- *support_package_installation_folder/toolbox/hdlcoder/supportpackages/alterasoc/+AlteraCycloneV/*
- *support_package_installation_folder/toolbox/hdlcoder/supportpackages/alterasoc/+ArrowSoCKit/*

See Also

[hdlcoder.Board](#) | [hdlcoder.ReferenceDesign](#)

Related Examples

- “Register a Custom Board” on page 41-42
- “Register a Custom Reference Design” on page 41-45
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

Register a Custom Board

In this section...

- “Define a Board” on page 41-42
- “Create a Board Plugin” on page 41-43
- “Define a Board Registration Function” on page 41-43

To register a custom board, you must:

- 1 Define a board.
- 2 Create a board plugin.
- 3 Define a board registration function, or add the new board plugin to an existing board registration function.

Define a Board

Before you begin, have the board documentation at hand so you can refer to the details of the board.

Requirements

A board definition must be:

- A MATLAB function that returns an `hdlcoder.Board` object.
The board definition function can have any name.
- In its board plugin folder.

How To Define A Board

- 1 Create a new file that defines a MATLAB function with any name.
- 2 In the MATLAB function, create an `hdlcoder.Board` object and specify its properties and interfaces according the characteristics of your custom board.
- 3 Optionally, to check that the definition is complete, run the `validateBoard` method.

For example, the following code defines a board:

```
function hB = plugin_board()
% Board definition

% Construct board object
hB = hdlcoder.Board;

hB.BoardName      = 'Digilent Zynq ZyBo';

% FPGA device information
hB.FPGAVendor    = 'Xilinx';
hB.FPGAFamily    = 'Zynq';
hB.FPGADevice    = 'xc7z010';
hB.FPGAPackage   = 'clg400';
hB.FPGASpeed     = '-2';

% Tool information
```

```

hB.SupportedTool = {'Xilinx Vivado'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

```

Create a Board Plugin

Requirements

A board plugin:

- Must be a package folder that contains the board definition file.

A package folder has a + prefix before the folder name. For example, the board plugin can be a folder named +ZedBoard.

- Must be on the MATLAB path.
- Can contain one or more reference design plugins.

How To Create a Board Plugin

- 1 Create a folder that has a name with a + prefix.
- 2 Save your board definition file to the folder.
- 3 Add the folder to your MATLAB path.

Define a Board Registration Function

Requirements

A board registration function:

- Must be named `hdlcoder_board_customization.m`.
- Returns a list of board plugins, specified as a cell array of character vectors.
- Must be on the MATLAB path.

How To Define a Board Registration Function

- 1 Create a file named `hdlcoder_board_customization.m` and save it anywhere on the MATLAB path.
- 2 In `hdlcoder_board_customization.m`, define a function that returns a list of board plugins as a cell array of character vectors.

For example, the following code defines a board registration function.

```

function r = hdlcoder_board_customization
% Board plugin registration files
% Format: % board_folder.board_definition_function

```

```
r = {'ZyboRegistration.plugin_board'};  
end
```

See Also

[hdlcoder.Board](#) | [hdlcoder.ReferenceDesign](#)

Related Examples

- “Register a Custom Reference Design” on page 41-45
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Board and Reference Design Registration System” on page 41-39

Register a Custom Reference Design

In this section...

- “Define a Reference Design” on page 41-45
- “Create a Reference Design Plugin” on page 41-46
- “Define a Reference Design Registration Function” on page 41-46

To register a custom reference design:

- 1** Define a reference design.
- 2** Create a reference design plugin.
- 3** Define a reference design registration function, or add the new reference design plugin to an existing reference design registration function.

Define a Reference Design

A reference design definition must be a MATLAB function that returns an `hdlcoder.ReferenceDesign` object. Create the reference design definition function in the reference design plugin folder. You can use any name for the reference design definition function.

To create a reference design definition:

- 1** Create a new file that defines a MATLAB function with any name.
- 2** In the MATLAB function, create an `hdlcoder.ReferenceDesign` object and specify its properties and interfaces according to the characteristics of your embedded system design.
- 3** If you want to check that the definition is complete, run the `validateReferenceDesign` method.

This MATLAB function defines a custom reference design:

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Demo system (Vivado 2014.2)';
hRD.BoardName = 'Digilent Zynq ZyBo';

% Tool information
% It is recommended to use a tool version that is compatible with the supported tool
% version. If you choose a different tool version, it is possible that HDL Coder is
% unable to create the reference design project for IP core integration.
hRD.SupportedToolVersion = {'2015.4'};

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl');

hRD.CustomFiles = {'ZYBO_zynq_def.xml'};
%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'clk_wiz_0/clk_out1', ...
    'ResetConnection', 'proc_sys_reset_0/peripheral_aresetn');

% add AXI4 and AXI4-Lite slave interfaces
```

```
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress', '0x40010000', ...
    'MasterAddressSpace', 'processing_system7_0/Data');
```

By default, HDL Coder generates an IP core with the default settings and integrates it into the reference design project. To customize these default settings, use the properties in the `hdlcoder.ReferenceDesign` object to define custom parameters and to register the function handle of the custom callback functions. For more information, see “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 41-48.

Create a Reference Design Plugin

A reference design plugin is a package folder that you define on the MATLAB path. The folder contains the board definition file and any custom callback functions.

To create a reference design plugin:

- 1 In the board plugin folder for the associated board, create a new folder that has a name with a + prefix.
For example, the reference design plugin can be a folder named `+vivado_base_ref_design`.
- 2 In the new folder, save your reference design definition file and any custom callback functions that you create.
- 3 In the new folder, save any files that are required by the embedded system design project, and are specific to your third-party synthesis tool, including Tcl, project, and design files.
- 4 Add the folder to your MATLAB path.

Define a Reference Design Registration Function

A reference design registration function contains a list of reference design functions and the associated board name. You must name the function `hdlcoder_ref_design_customization.m`. When the HDL Workflow Advisor opens, it searches the MATLAB path for files named `hdlcoder_ref_design_customization.m`, and uses the information to populate the reference design options for each board.

To define a reference design registration function:

- 1 Create a file named `hdlcoder_ref_design_customization.m` and save it anywhere on the MATLAB path.
- 2 In `hdlcoder_board_customization.m`, define a function that returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors.

For example, the following code defines a reference design registration function.

```
function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file

rd = {'ZyBoRegistration.Vivado2015_4.plugin_rd', ...};
      };

boardName = 'Digilent Zynq ZyBo';
```

```
end
```

The reference design registration function returns the associated board name, specified as a character vector, and a list of reference design plugins, specified as a cell array of character vectors.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 41-42
- “Define Custom Parameters and Callback Functions for Custom Reference Design” on page 41-48
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Board and Reference Design Registration System” on page 41-39

Define Custom Parameters and Callback Functions for Custom Reference Design

In this section...

["Define Custom Parameters and Register Callback Function Handle" on page 41-48](#)

["Define Custom Callback Functions" on page 41-52](#)

When you define your custom reference design, you can optionally use the properties in the `hdlcoder.ReferenceDesign` object to define custom parameters and callback functions.

Define Custom Parameters and Register Callback Function Handle

This MATLAB code shows how to define custom parameters and register the function handle of the custom callback functions in the reference design definition function.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'My Reference Design';
hRD.BoardName = 'ZedBoard';

% Tool information
hRD.SupportedToolVersion = {'2018.3'};

%% Add custom design files
% ...
% ...

%% Add optional custom parameters by using addParameter property.
% Specify custom 'DUT path' and 'Channel Mapping' parameters.
% The parameters get populated in the 'Set Target Reference Design'
% task of the HDL Workflow Advisor.
hRD.addParameter( ...
    'ParameterID', 'DutPath', ...
    'DisplayName', 'Dut Path', ...
    'DefaultValue', 'Rx', ...
    'ParameterType', hdlcoder.ParameterType Dropdown, ...
    'Choice', {'Rx', 'Tx'});
hRD.addParameter( ...
    'ParameterID', 'ChannelMapping', ...
    'DisplayName', 'Channel Mapping', ...
    'DefaultValue', '1');

%% Enable JTAG MATLAB as AXI Master IP Insertion. The IP
% insertion setting is visible in the 'Set Target Reference Design'
% task of the HDL Workflow Advisor. By default, the
% AddJTAGMATLABasAXIMasterParameter property is set to 'true'.
hRD.AddJTAGMATLABasAXIMasterParameter = 'true';
hRD.JTAGMATLABasAXIMasterDefaultValue = 'on';

%% Add custom callback functions. These are optional.
% With the callback functions, you can enable custom
% validations, customize the project creation, software
% interface model generation, and the bistream build.
% Register the function handle of these callback functions.

% Specify an optional callback for 'Set Target Reference Design'
% task in Workflow Advisor. Use property name
% 'PostTargetReferenceDesignFcn'.
hRD.PostTargetReferenceDesignFcn = ...
    @my_reference_design.callback_PostTargetReferenceDesign;
```

```
% Specify an optional callback for 'Set Target Interface' task in Workflow Advisor.
% Use the property name 'PostTargetInterfaceFcn'.
hRD.PostTargetInterfaceFcn = ...
    @my_reference_design.callback_PostTargetInterface;

% Specify an optional callback for 'Create Project' task
% Use the property name 'PostCreateProjectFcn' for the ref design object.
hRD.PostCreateProjectFcn = ...
    @my_reference_design.callback_PostCreateProject;

% Specify an optional callback for 'Generate Software Interface Model' task
% Use the property name 'PostSWInterfaceFcn' for the ref design object.
hRD.PostSWInterfaceFcn = ...
    @my_reference_design.callback_PostSWInterface;

% Specify an optional callback for 'Build FPGA Bitstream' task
% Use the property name 'PostBuildBitstreamFcn' for the ref design object.
hRD.PostBuildBitstreamFcn = ...
    @my_reference_design.callback_PostBuildBitstream;

% Specify an optional callback for 'Program Target Device'
% task to use a custom programming method.
hRD.CallbackCustomProgrammingMethod = ...
    @my_reference_design.callback_CustomProgrammingMethod;

%% Add interfaces
% ...
%
```

Define Custom Parameters

With the `addParameter` method of the `hdlcoder.ReferenceDesign` class, you can define custom parameters. In the preceding example code, the reference design defines two custom parameters, `DUT Path` and `Channel Mapping`. To learn more about the `addParameter` method, see [addParameter](#).

Specify Insertion of JTAG MATLAB as AXI Master IP

By default, HDL Coder adds a parameter **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** to all reference designs. When you set this parameter to on, the code generator automatically inserts the JTAG MATLAB AXI Master IP into your reference design. By using the JTAG MATLAB AXI Master IP, you can easily access the AXI registers in the generated DUT IP core on an FPGA board from MATLAB through the JTAG connection. See also “Set Up for MATLAB AXI Master” (HDL Verifier).

To use this capability, you must have the HDL Verifier hardware support packages installed and downloaded. See “Download FPGA Board Support Package” (HDL Verifier).

The code generator adjusts the **AXI4 Slave ID Width** to accommodate the MATLAB as AXI Master IP connection. After you generate the HDL IP core and create the reference design project, you can open the Vivado block design to see the JTAG MATLAB AXI Master IP inserted in the reference design.

In the previous example code, the reference design defines the `AddJTAGMATLABasAXIMasterParameter` and `JTAGMATLABasAXIMasterDefaultValue` properties of the `hdlcoder.ReferenceDesign` class. These properties control the default behavior of the **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** setting in the **Set Target Reference Design** task of the HDL Workflow Advisor. If you do not specify any of these properties in the `hdlcoder.ReferenceDesign` class, the **Insert JTAG MATLAB as AXI Master (HDL Verifier**

Required) parameter is displayed in the **Set Target Reference Design** task and the value is set to off. This example code illustrates the default behavior.

```
% Default behavior of Insert JTAG as AXI Master option

% This parameter controls visibility of the option in
% 'Set Target Reference Design Task' of HDL Workflow Advisor
% By default, the parameter value is set to 'true',
% which means that the option is displayed in the UI. If
% you do not want the parameter to be displayed, set this
% value to 'false'.
hRD.AddJTAGMATLABasAXIMasterParameter = 'true';

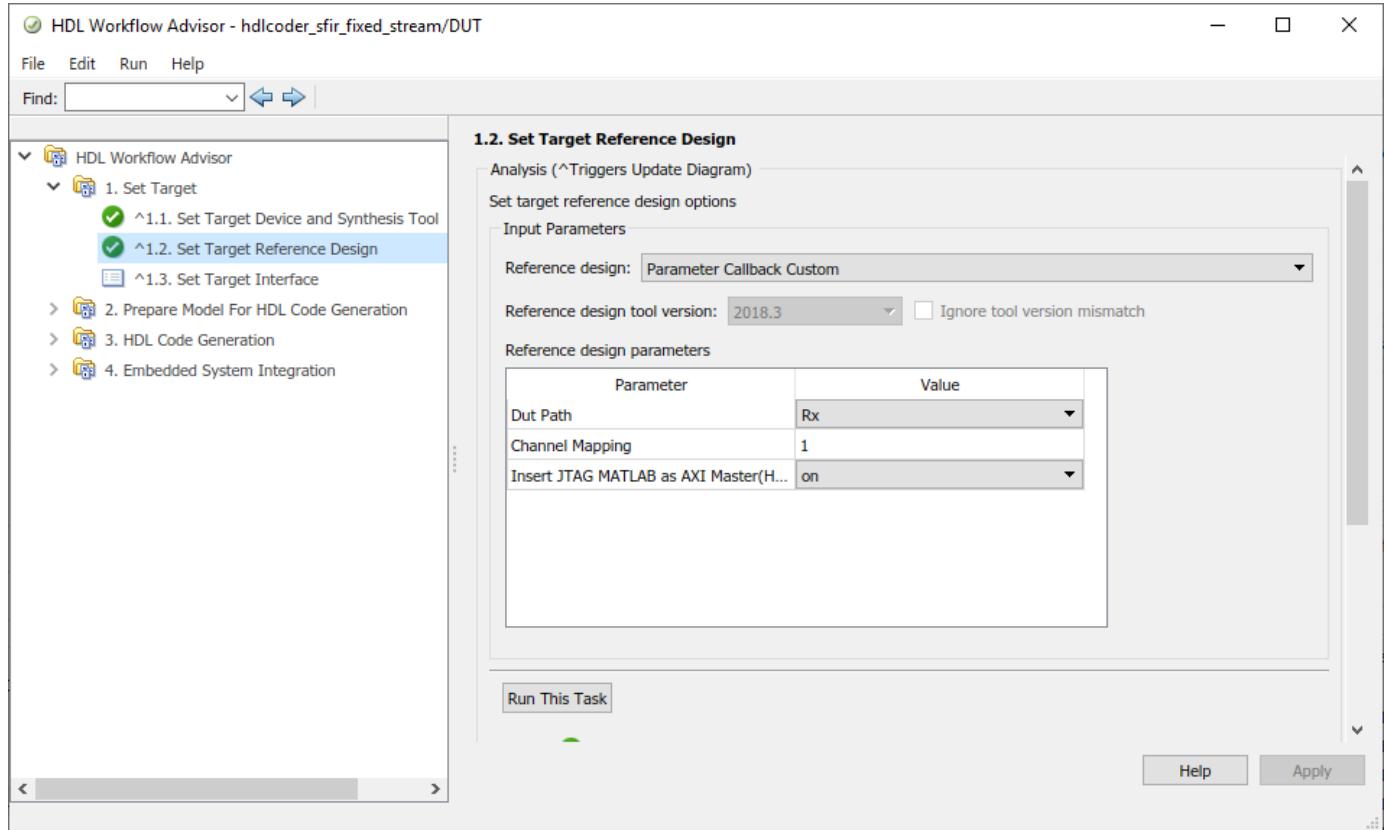
% This parameter controls the value of the option in the
% the 'Set Target Reference Design Task' task. By default,
% the value is 'off', which means that the parameter is
% displayed in the task and the value is off. To enable
% automatic insertion of JTAG AXI Master IP in the reference
% design, set this value to 'on'. In that case, the
% AddJTAGMATLABasAXIMasterParameter must be set to 'true'.
hRD.JTAGMATLABasAXIMasterDefaultValue = 'off';
```

For examples, see:

- Using JTAG MATLAB as AXI Master to control the HDL Coder IP Core on page 41-242
- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705” on page 40-153

Run IP Core Generation Workflow

When you open the HDL Workflow Advisor, HDL Coder populates the **Set Target Reference Design** task with the reference design name, tool version, custom parameters that you specified, and the **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** option set to on.



HDL Coder then passes these parameter values to the callback functions in the input structure.

If your synthesis tool is Xilinx Vivado, HDL Coder sets the reference design parameter values to variables. The variables are then input to the block design Tcl file. This code snippet is an example from the reference design project creation Tcl file.

```
update_ip_catalog
set DutPath {Rx}
set ChannelMapping {1}
source vivado_custom_block_design.tcl
```

The code shows how HDL Coder sets the reference design parameters before sourcing the custom block design Tcl file.

Register Callback Function Handles

In the reference design definition, you can register the function handle to reference the custom callback functions. You then can:

- Enable custom validations.
- Customize the reference design dynamically.
- Customize the reference design project creation settings.
- Change the generated software interface model.
- Customize the FPGA bitstream build process.

- Specify custom FPGA programming method.

With the `hdlcoder.ReferenceDesign` class, you can define callback property names. The callback properties have a naming convention. The callback functions can have any name. In the HDL Workflow Advisor, you can define callback functions to customize these tasks.

Workflow Advisor Task	Callback Property Name	Functionality
Set Target Reference Design	<ul style="list-style-type: none"> <code>CustomizeReferenceDesignFcn</code> <code>PostTargetReferenceDesignFcn</code> 	<ul style="list-style-type: none"> <code>CustomizeReferenceDesignFcn</code> enables customization of the reference design dynamically. By using this callback function, you can customize the block design Tcl file, reference design interfaces, reference design interface properties, and IP repositories in your reference design. See “Customize Reference Design Dynamically Based on Reference Design Parameters” on page 41-54. <code>PostTargetReferenceDesignFcn</code> enables custom validations. For an example that shows how you can validate that the Reset type is Synchronous, see <code>PostTargetReferenceDesignFcn</code>.
Set Target Interface	<code>PostTargetInterfaceFcn</code>	Enable custom validations. For an example that shows how you can validate not choosing a certain interface for a certain custom parameter setting, see <code>PostTargetInterfaceFcn</code> .
Create Project	<code>PostCreateProjectFcn</code>	Specify custom settings when HDL Coder creates the project. For an example, see <code>PostCreateProjectFcn</code> .
Generate Software Interface	<code>PostSWInterfaceFcn</code>	Change the generated software interface model. For an example, see <code>PostSWInterfaceFcn</code> .
Build FPGA Bitstream	<code>PostBuildBitstreamFcn</code>	Specify custom settings when you build the FPGA bitstream. When you use this function, the build process cannot be run externally. You must run the build process within the HDL Workflow Advisor by clearing the Run build process externally check box in the Build FPGA Bitstream task. For an example, see <code>PostBuildBitstreamFcn</code> .
Program Target Device	<code>CallbackCustomProgrammingMethod</code>	Specify a custom FPGA programming method. For an example, see <code>CallbackCustomProgrammingMethod</code> .

Define Custom Callback Functions

- For each of the callback function that you want HDL Coder to execute after running a task, create a file that defines a MATLAB function with any name.
- Make sure that the callback function has the documented input and output arguments.
- Verify that the functions are accessible from the MATLAB path.
- Register the function handle of the callback functions in the reference design definition function.
- Follow the naming conventions for the callback property names.

To learn more about these callback functions, see `hdlcoder.ReferenceDesign`.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 41-42
- “Register a Custom Reference Design” on page 41-45
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Board and Reference Design Registration System” on page 41-39

Customize Reference Design Dynamically Based on Reference Design Parameters

In this section...

- “Why Customize the Reference Design” on page 41-54
- “How Reference Design Customization Works” on page 41-54
- “Customizable Reference Design Parameters” on page 41-55
- “Example: Create Master Only or Slave Only or Both Slave and Master Reference Designs” on page 41-56

When you define your own custom reference design, you can dynamically customize the reference design by using the `CustomizeReferenceDesignFcn` method of the `hdlcoder.ReferenceDesign` class.

Why Customize the Reference Design

By customizing the reference design parameters, instead of maintaining separate reference designs that have different interface choices, data widths, or I/O plugins, create one reference design that has different interface choices or data widths as parameters. You can then use the `CustomizeReferenceDesignFcn` method to reference the callback function that has different choices for interfaces or data widths.

For example, instead of creating separate reference designs that have different data widths for the interfaces, you can parameterize the data width and then create a reference design parameter. When you run the **IP Core Generation** workflow, you can use the parameter to select the data width that you want to use. Similarly, instead of using multiple reference designs, you can create one reference design that has only AXI4-Stream Master or only AXI4-Stream Slave or both AXI4-Stream Master and AXI4-Stream Slave interfaces.

How Reference Design Customization Works

To define the callback function:

- 1 In the `plugin_rd` file, define the reference design parameters you want to customize by using the `addParameter` method.
- 2 Create a MATLAB file that defines the callback function. You can use any arbitrary name for the callback function.
- 3 Save the callback function in the same folder as the `plugin_rd.m` file.
- 4 Register the function handle of the callback function in the reference design definition `plugin_rd` file by using the `CustomizeReferenceDesignFcn` method.

To use the different reference design customizations:

- 1 Open the HDL Workflow Advisor. In the **Set Target Device and Synthesis Tool** task, select **IP Core Generation** as the **Target workflow** and then select the target board for which you created your own custom reference design as the **Target platform**.
- 2 In the **Set Target Reference Design** task, when you select the custom reference design that you want to customize for the target board, HDL Coder populates the reference design

parameters. Depending on the parameter choices such as the interface types you specify, the callback function is evaluated. Run this task.

- 3 Select the **Set Target Interface** task. Depending on the parameter you selected in the previous step, the target interface selection is populated in the Target platform interface table.

Customizable Reference Design Parameters

In the callback function, you can customize these reference design parameters. Do not specify these parameters in the `plugin_rd` file.

- Block design Tcl file

```
% ...
% if ~isempty(ParamValue)
    hRD.addCustomVivadoDesign( ...
        'CustomBlockDesignTcl', 'system_top.tcl', ...
        'VivadoBoardPart',      'xilinx.com:zc706:part0:1.0');

%
```

- Reference design interfaces and reference design interface properties

For example, you can parameterize the data width of the AXI4-Stream Master Channel. In this case, use the `addAXI4StreamInterface` method in the callback function instead of the `plugin_rd` file.

```
% ...
% Add AXI4-Stream interface by parameterizing data width
DataWidth = hRD.getParamValue(paramValue)

if ~isempty(DataWidth)
    hRD.addAXI4StreamInterface(
        'MasterChannelEnable', 'true', ...
        'SlaveChannelEnable', 'true', ...
        'MasterChannelConnection', 'ByPass_0_AXI4_Stream_Slave', ...
        'SlaveChannelConnection', 'ByPass_0_AXI4_Stream_Master', ...
        'MasterChannelDataWidth', DataWidth, ...
        'SlaveChannelDataWidth', DataWidth);
end

%
```

- IP repositories

In the callback function, you must specify the block design Tcl file when you add IP repositories.

```
% ...
%% Add IP Repository
hRD.addIPRepository(...,
    'IPListFunction', 'mathworks.hdlcoder.vivado.hdlcoder_video_iplist',
    'NotExistMessage', 'IP repository not found');

%% Add custom design files
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');

%
```

Note You cannot modify the reference design name, board name, and supported tool versions in the callback function. These parameters are used in the **Set Target Device and Synthesis Tool** task

before the callback function is evaluated when the target reference design is selected in the **Set Target Reference Design** task.

Example: Create Master Only or Slave Only or Both Slave and Master Reference Designs

Instead of using multiple reference designs, you can create one reference design that has only AXI4-Stream Master or only AXI4-Stream Slave or both AXI4-Stream Master and AXI4-Stream Slave interfaces. This example shows how you can customize the AXI4-Stream interface channels you want to use when targeting your own reference design for the **Xilinx Zynq ZC706 evaluation kit**.

This code shows the reference design parameter and interface choices for the reference design `my_reference_design` specified by using the `addParameter` method in the `plugin_rd` file. The `CustomizeReferenceDesignFcn` method references a callback function that has the name `customcallback_axistreamchannel`.

```
function hRD = plugin_rd()
% Reference design definition

% Copyright 2017-2019 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Vivado Custom Reference Design';
hRD.BoardName = 'Xilinx Zynq ZC706 evaluation kit';

% Tool information
hRD.SupportedToolVersion = {'2019.1'};

% ...
% ...

% Parameter For calling AXI4 Master
interface from Callback function
hRD.addParameter ...
    ('ParameterID' , 'stream_channel', ...
    'DisplayName' , 'Stream Channel', ...
    'DefaultValue' , 'Both Master and Slave',...
    'ParameterType' , hdlcoder.ParameterType Dropdown, ...
    'Choice' , {'Both Master and Slave','Master Only','Slave Only'}));

% Reference the callback function.
hRD.CustomizeReferenceDesignFcn = @my_reference_design.customcallback_axistreamchannel;

%
```

When you create the callback function, pass the `infoStruct` argument to the function. The argument contains the reference design and board information in a `structure` format. This code shows the callback function `customcallback_axistreamchannel` that has the AXI4-Stream Master or Slave Channels or both channels specified by using the `addAXI4StreamInterface` method.

```
% Control AXI Master or Slave channel selection by using callback function

function customcallback_axistreamchannel(infoStruct)
% Reference design callback run at the end of the task Set Target Reference Design
%
% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
```

```
% infoStruct.ReferenceDesignToolVersion: Reference design Tool Version set in 1.2 Task
paramStruct = infoStruct.ParameterStruct;

if ~isempty(paramStruct)
    paramIDCell = fieldnames(paramStruct);
    paramValue = '';
    for ii = 1:length(paramIDCell)
        paramID = paramIDCell(ii);
        if strcmp(paramID,'Both Master and Slave')
            paramValue = paramStruct.paramID;
            break;
        elseif strcmp(paramID,'Master Only')
            paramValue = paramStruct.paramID;
            break;
        elseif strcmp(paramID,'Slave Only')
            paramValue = paramStruct.paramID;
            break;
        end
    end
end
interface_type = str2double(paramValue);

if ~isempty(interface_type)

    if strcmp(interface_type, 'Both Master and Slave')

        % add custom vivado design
        hRD.addCustomVivadoDesign( ...
            'CustomBlockDesignTcl', 'system_top.tcl', ...
            'VivadoBoardPart', 'xilinx.com:zc706:part0:1.0');

        hRD.addAXI4StreamInterface( ...
            'MasterChannelEnable', 'true', ...
            'SlaveChannelEnable', 'true', ...
            'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
            'SlaveChannelConnection', 'axi_dma_mm2s/M_AXIS_MM2S', ...
            'MasterChannelDataWidth', 32, ...
            'SlaveChannelDataWidth', 32);

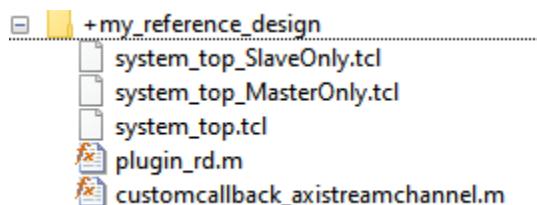
    elseif strcmp(interface_type, 'Master Only')

        % add custom vivado design
        hRD.addCustomVivadoDesign( ...
            'CustomBlockDesignTcl', 'system_top.tcl', ...
            'VivadoBoardPart', 'xilinx.com:zc706:part0:1.0');

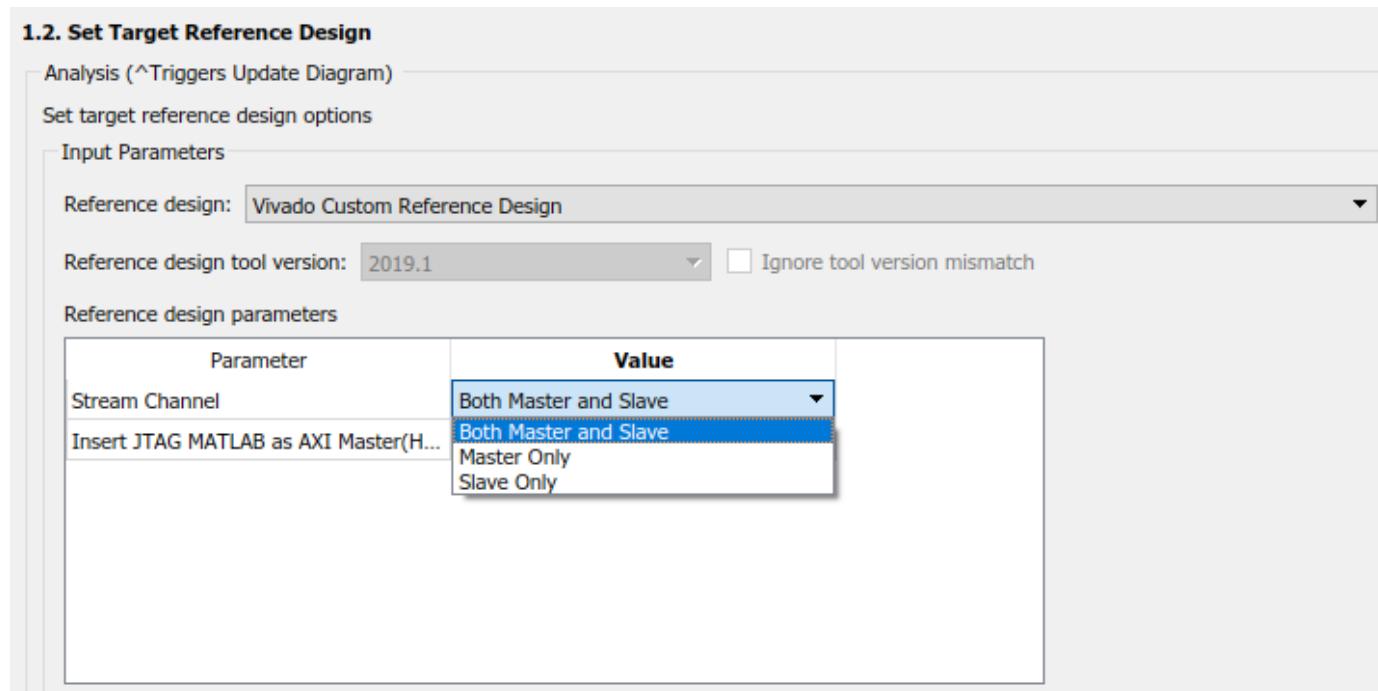
        hRD.addAXI4StreamInterface( ...
            'MasterChannelEnable', true, ...
            'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
            'MasterChannelDataWidth', 32);

    end
end
%
```

Save the callback function in the same folder as the `plugin_rd` file.



When you run the IP Core Generation workflow with Xilinx Zynq ZC706 evaluation kit as the **Target platform**, you see the parameter **Stream Channel** displayed in the **Set Target Reference Design** task.



You can specify the reference design type you want to use and then run the workflow to specify the target platform interfaces and then generate the HDL IP core and then integrate the IP core into the reference design.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Register a Custom Board” on page 41-42
- “Register a Custom Reference Design” on page 41-45
- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Board and Reference Design Registration System” on page 41-39

Define and Add IP Repository to Custom Reference Design

In this section...

["Create an IP Repository Folder Structure" on page 41-59](#)

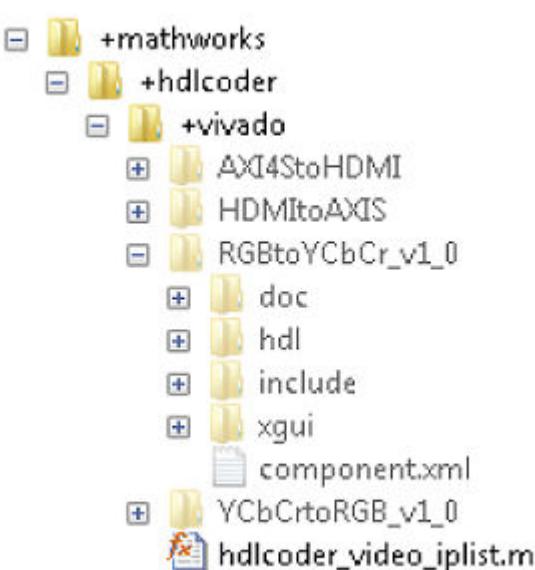
["Define IP List Function" on page 41-60](#)

["Add IP List Function to Reference Design Project" on page 41-61](#)

When you create your custom reference design, you might require custom IP modules that do not come with Altera Qsys or Xilinx Vivado. To use custom IP modules, create your own IP repository folder that contains IP module subfolders. The **IP Core Generation** workflow then uses these custom IP modules when creating the reference design project. You can create multiple IP repositories and add all or some of the IP modules in each repository to your custom reference design project. You can also reuse and share IP repositories across multiple reference designs.

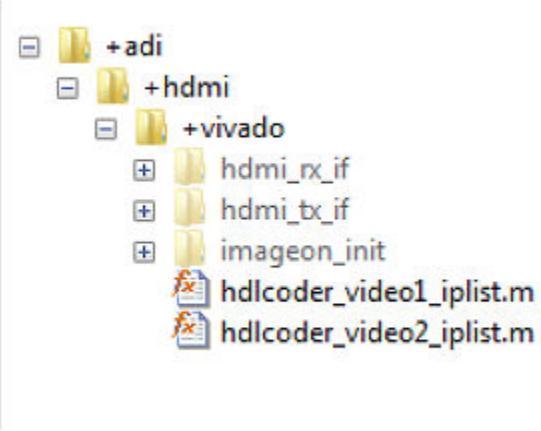
Create an IP Repository Folder Structure

Create an IP repository folder anywhere on the MATLAB path. This figure shows a typical IP repository folder structure.



When you create the folder structure, use the naming convention +(company)/+(product)/+(tool). In this example, the folder structure is +(mathworks)/+(hdlcoder)/+(vivado). The folder +vivado acts as the IP repository. This folder contains subfolders corresponding to IP modules such as AXI4StoHDMI and HDMItoAXIS. The folder also contains a `hdlcoder_video_iplist` MATLAB function. Using this function, specify the IP modules to add to the reference design.

The same repository folder can have multiple MATLAB functions. This figure shows two MATLAB functions, `hdlcoder_video1_iplist` and `hdlcoder_video2_list`, in the +vivado folder. The functions can share the same IP modules or point to different IP modules in the repository.



Note If your synthesis tool is Xilinx Vivado, the IP modules in the repository folder can be in zip file format.

Define IP List Function

Create a MATLAB function that specifies the IP modules to add to the reference design. Save this function in the IP repository folder. For the function name, use the naming convention `hdlcoder_<specific_use>_ipList`. This example uses `hdlcoder_video_ipList` as the function name because it targets video applications. Using this function, specify whether you want to add all or some of the IP modules in the repository to the reference design project. To add all IP modules, use an empty cell array for `ipList`. This MATLAB code shows how to add all IP modules in the repository to the reference design.

```
function [ipList] = hdlcoder_video_ipList( )
% All IP modules in the repository folder.

ipList = {};
```

You can specify the root directory as an optional second output argument to the IP list function. In this case, the IP modules do not have to be located in a path relative to the IP list function. The IP repository can also be located outside the MATLAB path.

If you do not specify the root directory, the function searches for IP modules relative to its location.

```
function [ipList, rootDir] = hdlcoder_video_ipList( )
% All IP modules with a root directory.

ipList = {};
```

To add some of the IP modules that are in the folder, specify the IP modules as a cell array of character vectors. This MATLAB code specifies the AXI4StoHDMI IP and the HDMItoAXIS IP as the IP modules to add to your custom reference design.

```

function [ipList] = hdlcoder_video_iplist( )
% AXI4StoHDMI and HDMItoAXIS IP in the repository folder.

ipList = {'AXI4StoHDMI','HDMItoAXIS'};

```

Add IP List Function to Reference Design Project

Using the `addIPRepository` method of the `hdlcoder.ReferenceDesign` class, add the IP list function to your custom reference design. This example reference design adds `hdlcoder_video_iplist` to the custom reference design `My Reference Design`.

```

function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'My Reference Design';
hRD.BoardName = 'ZedBoard'

% Tool information
hRD.SupportedToolVersion = {'2016.2'};

%% Add custom design files
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');

% Add IP Repository
hRD.addIPRepository(...,
    'IPListFunction', 'mathworks.hdlcoder.vivado.hdlcoder_video_iplist',
    'NotExistMessage', 'IP repository not found');

% ...
% ...

```

To use the IP modules when the code generator creates the project, open the HDL Workflow Advisor, and run the IP Core Generation workflow to the **Create Project** task. After running this task, you can see the IP module subfolders in the repository copied over to the `ipcore` folder of the project. The `CustomBlockDesignTcl` can then use these IP modules.

See Also

`hdlcoder.Board` | `hdlcoder.ReferenceDesign`

Related Examples

- “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196
- “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215

More About

- “Board and Reference Design Registration System” on page 41-39

- “Register a Custom Reference Design” on page 41-45
- “Customize Reference Design Dynamically Based on Reference Design Parameters” on page 41-54

FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules

This example shows how to implement a Simulink® algorithm on a Speedgoat Simulink-programmable I/O module by using the HDL Workflow Advisor. You run the **Simulink Real-Time** FPGA I/O workflow to:

- 1** Specify an FPGA I/O module and its interfaces.
- 2** Synthesize the Simulink algorithm for FPGA programming.
- 3** Generate a Simulink® Real-Time™ interface subsystem model.

The interface subsystem model contains blocks to program the FPGA and communicate with the FPGA module through the PCIe bus during real-time application execution. You add the generated subsystem to your Simulink Real-Time domain model.

This example uses the Speedgoat IO397-50k module. See “Speedgoat FPGA Support with HDL Workflow Advisor” on page 40-8.

Setup and Configuration

Before deploying your algorithm on the Speedgoat IO module:

1. Install the latest version of Xilinx® Vivado® as listed in “HDL Language Support and Supported Third-Party Tools and Hardware”.

Then, set the tool path to the installed Xilinx Vivado executable by using the `hdlsetuptoolpath` function.

```
hdlsetuptoolpath('ToolName','Xilinx Vivado','ToolPath','C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

2. For real-time simulation, set up the development environment and target computer settings. See “Get Started with Simulink Real-Time” (Simulink Real-Time).
3. Install the Speedgoat Library and the Speedgoat HDL Coder Integration packages. See [Install Speedgoat HDL Coder Integration Packages](#).

HDL Workflow Advisor

The HDL Workflow Advisor guides you through HDL code generation and the FPGA design process. Use the Advisor to:

- Check the model for HDL code generation compatibility and fix incompatible settings.
- Generate HDL code, test bench, and scripts to build and run the code and test bench.
- Perform synthesis and timing analysis.
- Deploy the generated code on SoCs, FPGAs, and Speedgoat I/O modules.

To open the HDL Workflow Advisor for a subsystem inside the model, use the `hdladvisor` function.

```
load_system('sschdlexTwoLevelConverterIgbtExample')
hdladvisor('sschdlexTwoLevelConverterIgbtExample/Simscape_system')
```

The left pane of the Advisor contains folders that represent a group of related tasks. Expanding the folders and selecting a task displays information about that task in the right pane. The right pane

contains simple controls for running the task to advanced parameters and option settings that control HDL code and test bench generation. To learn more about each task, right-click that task, and select **What's This?**. See “Getting Started with the HDL Workflow Advisor” on page 31-6.

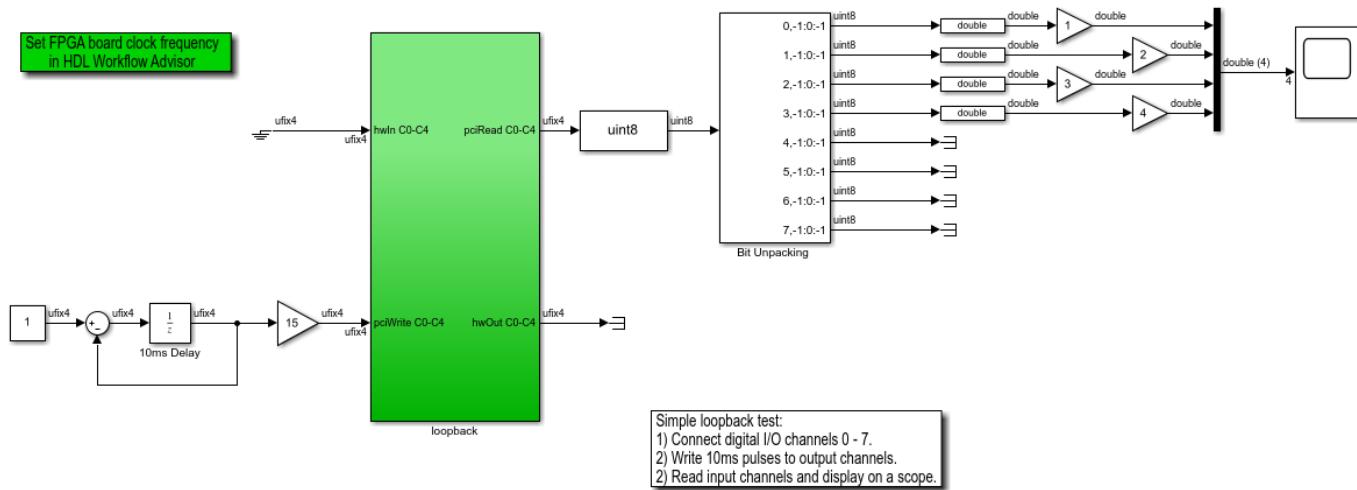
Simulink Loopback Domain Model

The Simulink domain model has a subsystem that contains the algorithm to program onto the FPGA chip. Use this model to test your FPGA algorithm in a simulation environment before you download the algorithm to an FPGA board. In this case, the model is a loopback test.

```
open_system('dslrtSGFPGAloopback_fpga')
```

This model is your FPGA domain model. It represents the simulation sample rate of the clock on your FPGA board. The **loopback** subsystem contains the algorithm to load on the FPGA. The data type and the number of input and output lines of the model are configured to fit the Speedgoat I0397-50k platform.

```
open_system('hdlcoder_slrt_loopback')
set_param('hdlcoder_slrt_loopback', 'SimulationCommand', 'Update')
```



Copyright 2010-2020 The MathWorks, Inc.

Generate Simulink Real-Time Interface Model for Speedgoat I0397 Platform

1. Open the HDL Workflow Advisor for the **loopback** subsystem. This subsystem is loaded on the FPGA.

```
hdladvisor('hdlcoder_slrt_loopback/loopback')
```

2. Expand the **Set Target** folder. In the **Set Target Device and Synthesis Tool** task, specify **Target workflow** as **Simulink Real-Time FPGA I/O** and **Target platform** as **Speedgoat I0397-50k**. Right-click the **Set Target Reference Design** task and select **Run to Selected Task**.

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: Simulink Real-Time FPGA I/O

Target platform: Speedgoat IO397-50k

Synthesis tool: Xilinx Vivado Tool version: 2019.2.1

Family: Artix7 Device: xc7a50t

Package: csg325 Speed: -2

Project folder: hdl_prj

Result:  Passed

Passed Set Target Device and Synthesis Tool.

3. In the **Set Target Interface** task, map ports hwIn and hwOut to IO397_TTL [0:13] and pciRead C0-C4 and pciWrite C0-C4 to PCIe interface. Click **Run This Task**.

1.3. Set Target Interface

Target platform interface table

Port Name	Port Type	Data Type	Target Platform Interfaces	Interface Mapping	Interface Options
hwIn C0-C4	Import	ufix4	IO397_TTL [0:13]	[0:3]	
pciWrite C0-C4	Import	ufix4	PCIe Interface	x"100"	Options...
pciRead C0-C4	Outport	ufix4	PCIe Interface	x"104"	
hwOut C0-C4	Outport	ufix4	IO397_TTL [0:13]	[8:11]	

Run This TaskResult:  Passed**Passed** Set Target Interface Table.

4. Run the **Set Target Frequency** task with the default value set for **Target Frequency (MHz)**. The target frequency must be in the range **Frequency Range (MHz)**.

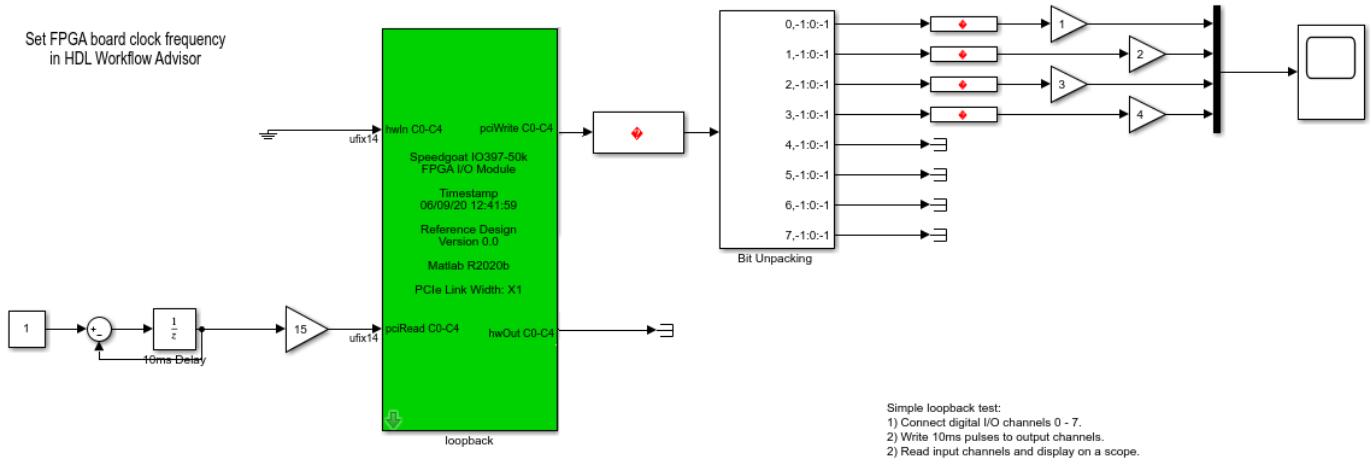
5. Expand the **Download to Target** task. Right-click the **Generate Simulink Real-Time interface** task and select **Run to Selected Task**.

This task generates RTL code and IP core, FPGA bitstream, and the Simulink Real-Time Interface model. In the **Create Project** task, open the Vivado project to see the implemented block design.

Real-Time Subsystem Integration and Execution

After the **Generate Simulink Real-Time interface** task passes, click the link to open the Simulink Real-Time interface model.

Generated by HDL Workflow Advisor on 09-Jun-2020 12:43:35



Copyright 2010-2020 The MathWorks, Inc.

The Simulink-Real Time Interface model contains a masked subsystem that has the same name as the subsystem in the Simulink FPGA domain model. This subsystem is the Simulink Real-Time Interface subsystem that contains the algorithm which is loaded onto the FPGA. Use the generated Simulink Real-Time Interface model or create a Simulink Real-Time Domain model and copy the Simulink Real-Time Interface subsystem into that model to simulate your FPGA algorithm on the Speedgoat target machine.

In the Simulink Real-Time interface subsystem mask, set three parameters:

- Device index
- PCI slot
- Sample time

When the target has a single FPGA I/O board, leave the device index to the default value. For multiple FPGA I/O boards, specify a unique device index. If two or more boards are of the same type, specify the PCI slot for each board.

For real-time testing, you can log the signals and view the simulation results on the Simulation Data Inspector.

- 1 On the **REAL-TIME** tab, open the Simulink Real-Time Explorer and specify the target interface connection settings. For an example, see “Hardware-in-the-Loop Implementation of Simscape Model on Speedgoat FPGA I/O Modules” on page 32-90.
- 2 On the **REAL-TIME** tab, click **Run on Target** to build and download the Simulink Real-Time application. The real-time application loads onto the Speedgoat target machine and the FPGA algorithm bitstream loads onto the FPGA.

You can then view the simulation results on the Simulation Data Inspector.

See Also

Related Examples

- “Processor and FPGA Synchronization” on page 40-23
- “IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules” on page 41-93
- “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2
- Speedgoat I/O Examples

Model Design for AXI4-Stream Video Interface Generation

In this section...

- “Streaming Pixel Protocol” on page 41-69
- “Protocol Signals and Timing Diagrams” on page 41-69
- “Model Data and Control Bus Signals” on page 41-71
- “Map DUT Ports to Multiple Channels” on page 41-75
- “Model Designs with Multiple Sample Rates” on page 41-75
- “Video Porch Insertion Logic” on page 41-75
- “Default Video System Reference Design” on page 41-76
- “Restrictions” on page 41-77

With the HDL Coder software, you can implement a simplified, streaming pixel protocol in your model. The software generates an HDL IP core with AXI4-Stream Video interfaces.

Streaming Pixel Protocol

You can use the streaming pixel protocol for AXI4-Stream Video interface mapping. Video algorithms process data serially and generate video data as a serial stream of pixel data and control signals. To learn about the streaming pixel protocol, see “Streaming Pixel Interface” (Vision HDL Toolbox).

To generate an IP core with AXI4-Stream Video interfaces, in your DUT interface, implement these signals:

- Pixel Data
- Pixel Control Bus

The **Pixel Control Bus** is a bus that has these signals:

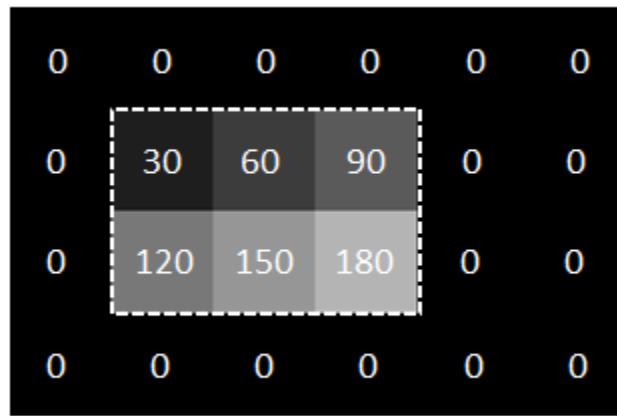
- **hStart**
- **hEnd**
- **vStart**
- **vEnd**
- **valid**

The signals **hStart** and **hEnd** represent the start of an active line and the end of an active line respectively. The signals **vStart** and **vEnd** represent the start of a frame and the end of a frame.

You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

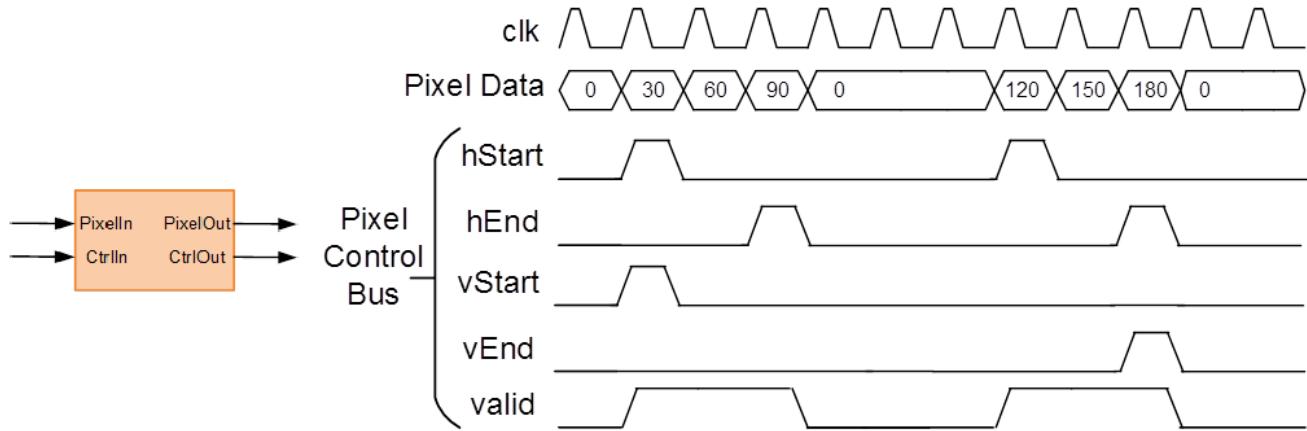
Protocol Signals and Timing Diagrams

This figure is a 2-by-3 pixel image. The active image area is the rectangle with a dashed line around it and the inactive pixels that surround it. The pixels are labeled with their grayscale values.



Pixel Data and Pixel Control Bus

This figure shows the timing diagram for the **Pixel Data** and **Pixel Control Bus** signals that you model at the DUT interface.



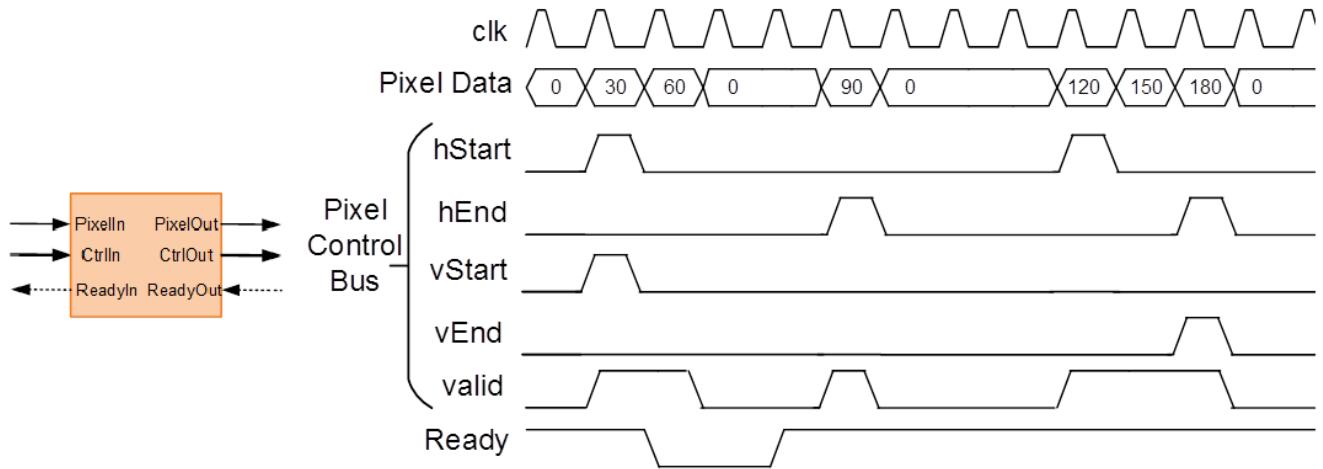
The **Pixel Data** signal is the primary video signal that is transferred across the AXI4-Stream Video interface. When the **Pixel Data** signal is valid, the **valid** signal is asserted.

The **hStart** signal becomes high at the start of the active lines. The **hEnd** signal becomes high at the end of the active lines.

The **vStart** signal becomes high at the start of the active frame in the second line. The **vEnd** signal becomes high at the end of the active frame in the third line.

Optional Ready Signal

This figure shows the timing diagram for the **Pixel Data**, the **Pixel Control Bus**, and the **Ready** signal that you model at the DUT interface.



When you map the DUT ports to an AXI4-Stream Video interface, you can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.

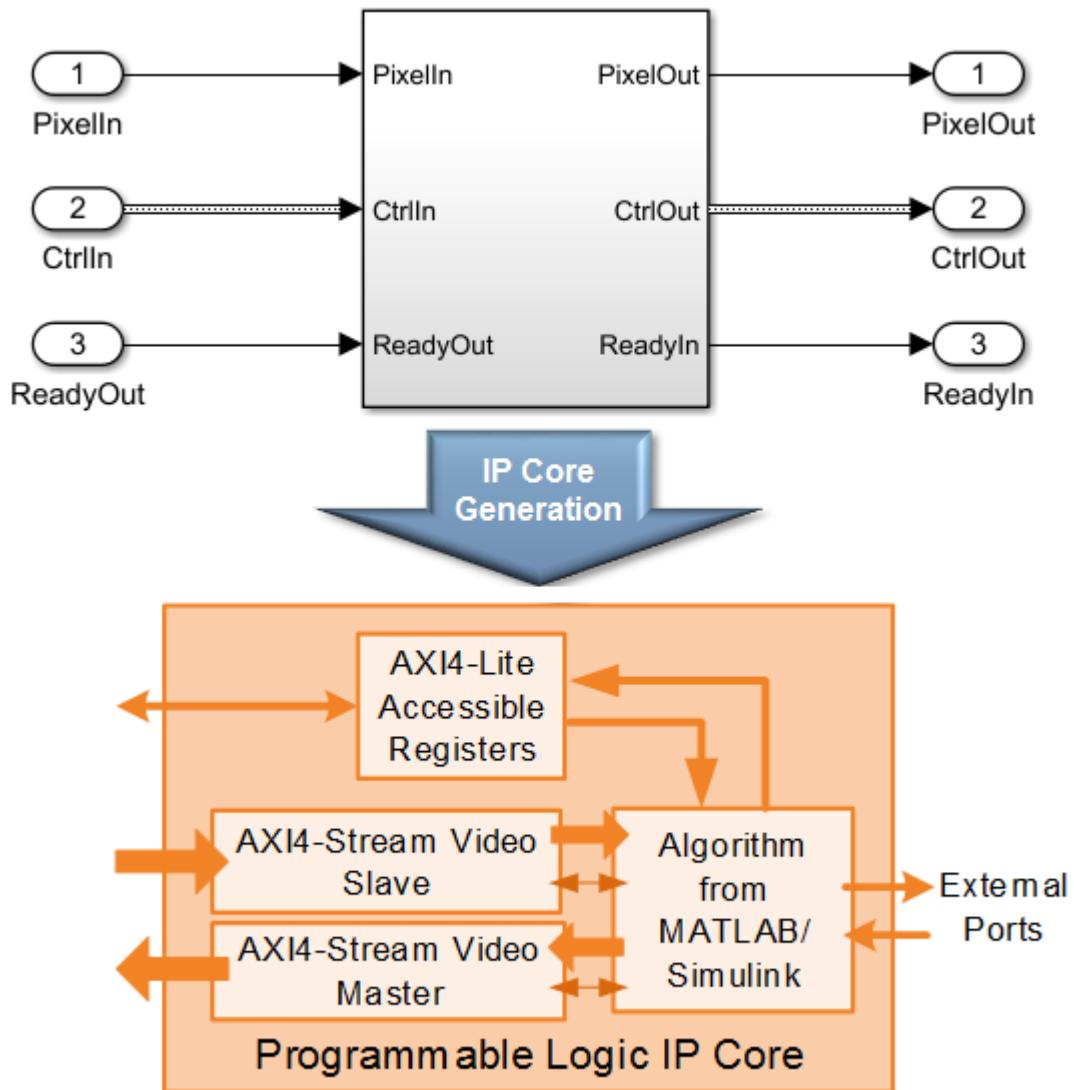
In a Slave interface, with the **Ready** signal, you can apply back pressure. In a Master interface, with the **Ready** signal, you can respond to back pressure.

If you model the **Ready** signal in your AXI4-Stream Video interfaces, your Master interface must deassert its **valid** signal one cycle after the **Ready** signal is deasserted.

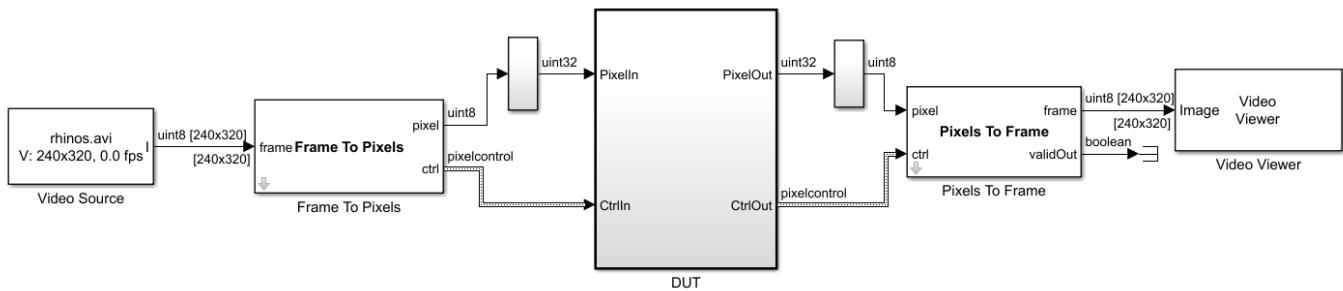
If you do not model the **Ready** signal, HDL Coder generates the associated backpressure logic.

Model Data and Control Bus Signals

You can model your video algorithm with **Pixel Data** and **Pixel Control Bus** signals at the DUT ports and map the signals to AXI4-Stream Video interfaces. You can optionally model the backpressure signal, **Ready**, and map it to the AXI4-Stream Video interface.



This figure shows an example of a top-level Simulink model with a Video Source input.

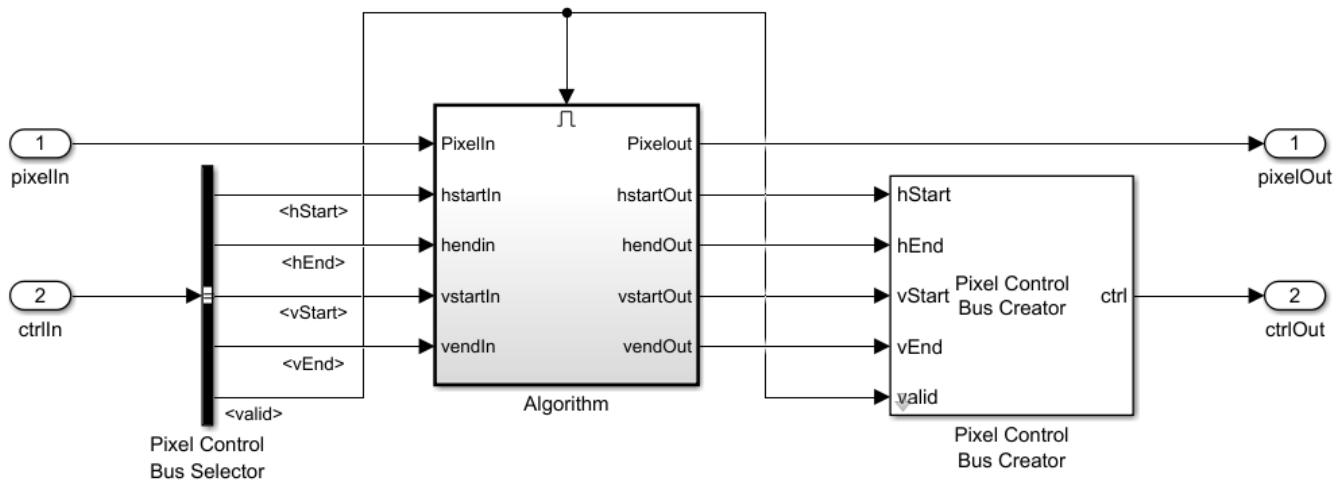


The Frame To Pixels and Pixels To Frame blocks perform the conversion between the video frames and the **Pixel Data** and **Pixel Control Bus** at the DUT interface. To use these blocks, you must have the Vision HDL Toolbox installed.

See also Frame To Pixels and Pixels To Frame.

Pixel Data and Pixel Control Bus Modeling

This figure shows how to model the **Pixel Data** and **Pixel Control Bus** signals inside the **DUT** subsystem.



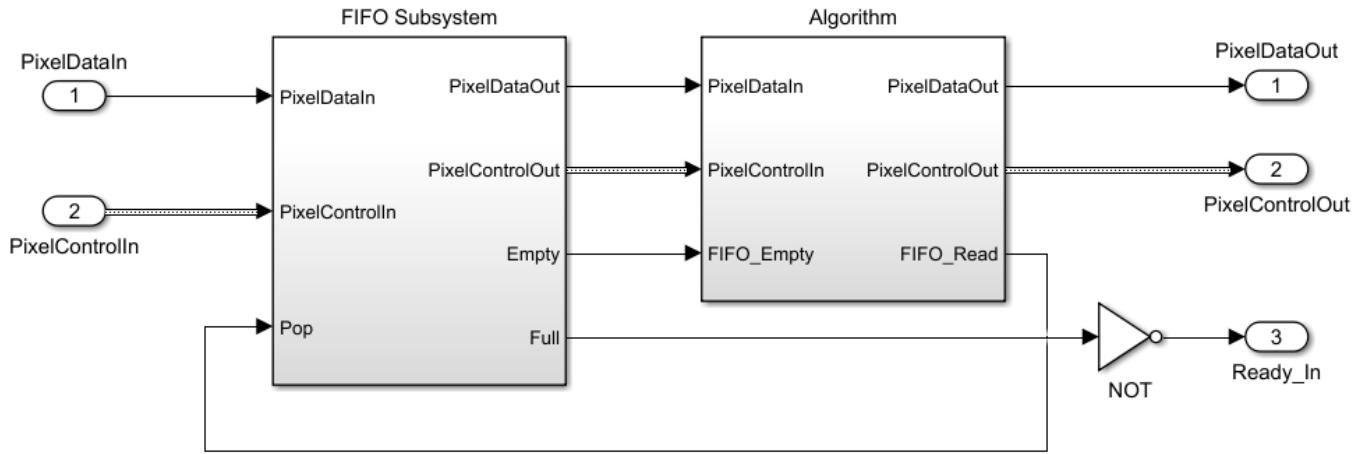
You can directly connect the **valid** signal from the **Pixel Control Bus** to the Enable port. If you do not have the Vision HDL Toolbox software, replace the Pixel Control Bus Selector and Pixel Control Bus Creator blocks with the Bus Selector and Bus Creator blocks respectively.

Ready Signal Modeling

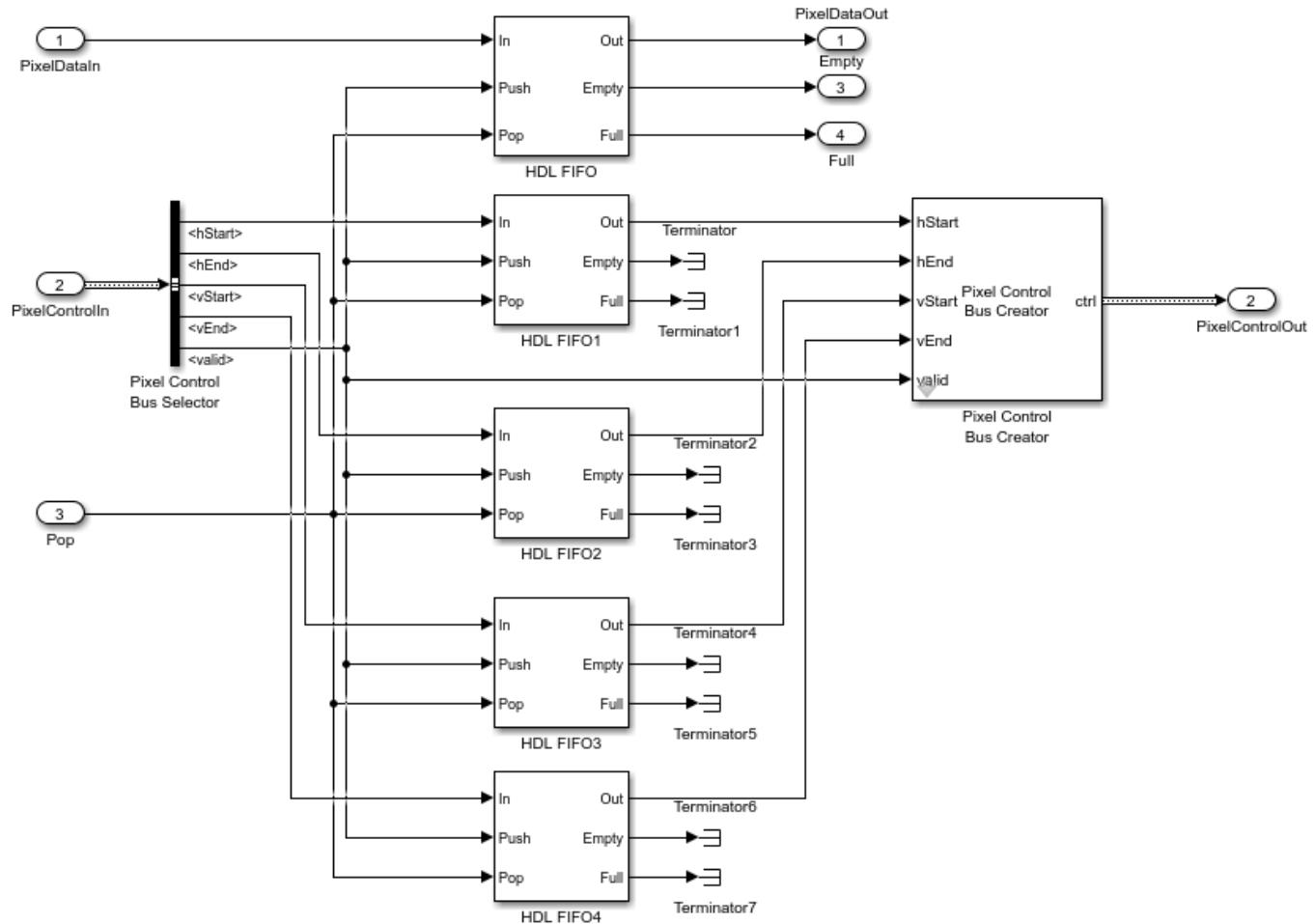
The AXI4-Stream Video interfaces in your DUT can optionally include a **Ready** signal.

For example, you can have a FIFO in your DUT to store some video data before processing the signals. Use a **FIFO Subsystem** that contains HDL FIFO blocks to store the **Pixel Data** and the **Pixel Control Bus** signals. To apply the backpressure to the upstream component, model the **Ready** signal based on the FIFO Full signal.

This figure shows how to model the **Ready** signal inside the **DUT** subsystem.



The **FIFO Subsystem** block uses HDL FIFO blocks for the **Pixel Data** and for the **Pixel Control Bus** signals.



Disable delay balancing for the **Ready** signal path. If you enable delay balancing, the coder can insert one or more delays on the **Ready** signal.

Map DUT Ports to Multiple Channels

When you run the IP Core Generation workflow, you can map multiple DUT ports to AXI4-Stream Video Master and AXI4-Stream Video Slave channels. The DUT ports mapped to multiple interface channels must use scalar data type. When you use vector ports, you can map the ports to at most one AXI4-Stream Video Master channel and one AXI4-Stream Video Slave channel.

To learn more, see “Generate HDL IP Core with Multiple AXI4-Stream and AXI4 Master Interfaces” on page 41-17.

Model Designs with Multiple Sample Rates

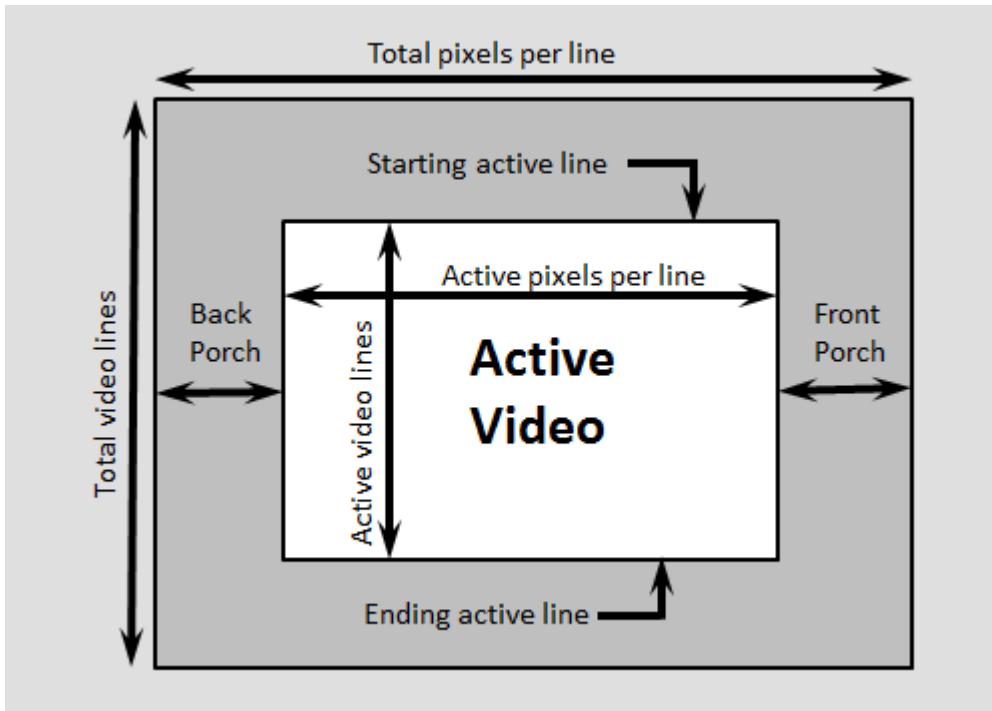
The HDL Coder software supports designs with multiple sample rates when you run the IP Core Generation workflow. When you map the interface ports to AXI4-Stream Video Master or AXI4-Stream Video Slave interfaces, to use multiple sample rates, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation.

To learn more, see “Multirate IP Core Generation” on page 41-35.

Video Porch Insertion Logic

Video capture systems scan video signals from left to right and from top to bottom. As these systems scan, they generate inactive intervals between lines and frames of active video. This inactive interval is called a video porch. The horizontal porch consists of inactive cycles between the end of one line and the beginning of next line. The vertical porch consists of inactive cycles between the ending active line of one frame and the starting active line of next frame.

This figure shows a video frame with the horizontal porch split into a front and a back porch.



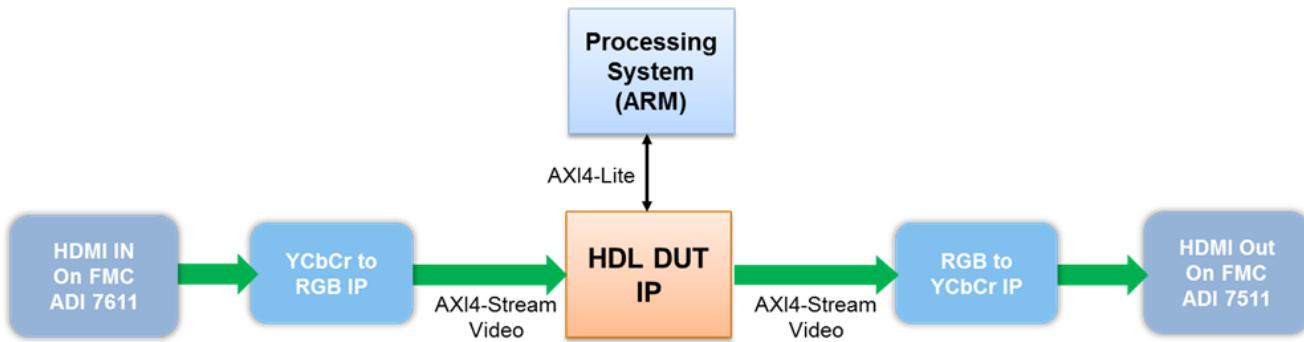
The AXI4-Stream Video interface does not require a video porch, but Vision HDL Toolbox algorithms require a porch for processing video streams. If the incoming pixel stream does not have a sufficient porch, HDL Coder inserts the required amount of porch to the pixel stream. By using the AXI4-Lite registers in the generated IP core, you can customize these porch parameters for each video frame:

- Active pixels per line (Default: 1920)
- Active video lines: (Default: 1080)
- Horizontal porch length (Default: 280)
- Vertical porch length (Default: 45)

Default Video System Reference Design

You can integrate the generated HDL IP core with AXI4-Stream Video interfaces into the Default video system reference design.

This figure is a block diagram of the Default video system reference design architecture.



You can use this **Default video system** reference design architecture with these target platforms:

- Xilinx Zynq ZC702 evaluation kit
- Xilinx Zynq ZC706 evaluation kit
- ZedBoard

To use the **Default video system** reference design, you must install the Computer Vision Toolbox™ Support Package for Xilinx Zynq-Based Hardware.

Restrictions

When you map the DUT ports to AXI4-Stream Video interfaces:

- The DUT port mapped to the **Pixel Data** signal must use a scalar data type.
- Xilinx Zynq-7000 must be your target platform.
- You must use Xilinx Vivado as your synthesis tool.
- **Processor/FPGA synchronization** must be Free running.

See Also

More About

- “Model Design for AXI4-Stream Interface Generation” on page 41-10
- “Streaming Pixel Interface” (Vision HDL Toolbox)

See Also

Related Examples

- “Getting Started with AXI4-Stream Video Interface in Zynq Workflow” on page 41-152

Model Design for AXI4 Master Interface Generation

In this section...

- “Simplified AXI4 Master Protocol - Write Channel” on page 41-78
- “Simplified AXI4 Master Protocol - Read Channel” on page 41-80
- “Base Address Register Calculation” on page 41-81
- “Modeling for AXI4 Master Interfaces” on page 41-81
- “Map Vector Ports to AXI4 Master Interfaces” on page 41-84
- “Model Designs with Multiple Sample Rates” on page 41-86
- “Reference Designs for IP Core Integration” on page 41-87
- “Restrictions” on page 41-88

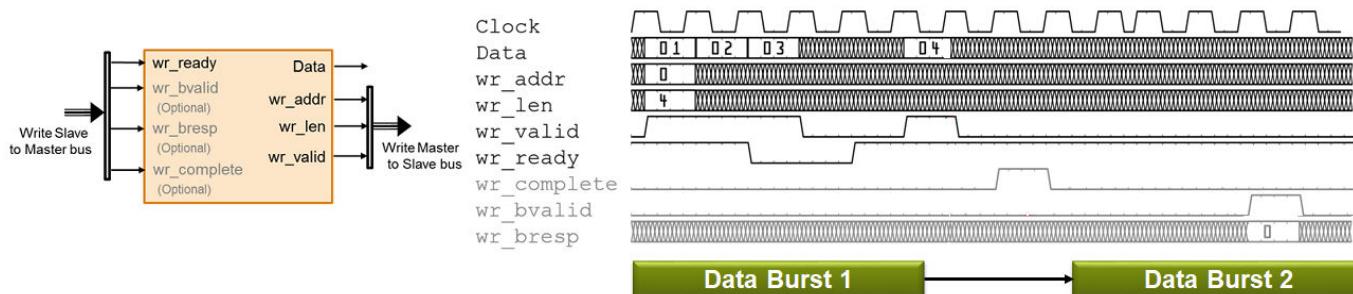
For designs that require accessing large data sets from an external memory, model your algorithm with a simplified AXI4 Master protocol. When you run the **IP Core Generation** workflow, HDL Coder generates an IP core with AXI4 Master interfaces. The AXI4 Master interface can communicate between your design and the external memory controller IP by using the AXI4 Master protocol. Use the AXI4 Master interface when your:

- Design targets multi-frame video processing applications. You can store the image data in external memory, such as a DDR3 memory on board, and then read or write the images to your design in a burst fashion for high-speed processing.
- Algorithm must access memory data in a non-streaming arbitrary pattern.
- DUT IP core must control other IPs with the AXI4 slave interface in the system. This capability is especially useful in standalone FPGA devices.

Simplified AXI4 Master Protocol - Write Channel

To map the DUT ports to AXI4 Master interfaces, use the simplified AXI4 Master protocol. You do not have to model the actual AXI4 Master protocol and instead you can use the simplified protocol. When you run the **IP Core Generation** workflow, the generated HDL code contains a wrapper logic that translates between the simplified protocol and the actual AXI4 Master protocol. The simplified protocol requires you to use less protocol signals, eases the handshaking mechanism between valid and ready signals, and supports bursts of arbitrary lengths.

Use the simplified AXI4 Master write protocol for a write transaction and the simplified AXI4 Master read protocol for a read transaction. This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master write transaction.



The DUT waits for `wr_ready` to become high to initiate a write request. When `wr_ready` becomes high, the DUT can send out the write request. The write request consists of the `Data` and `Write Master to Slave` bus signals. This bus consists of `wr_len`, `wr_addr`, and `wr_valid`. `wr_addr` specifies the starting address that DUT wants to write to. The `wr_len` signal corresponds to the number of data elements in this write transaction. `Data` can be sent as long as `wr_valid` is high. When `wr_ready` becomes low, the DUT must stop sending data within one clock cycle, and the `Data` signal becomes invalid. If the DUT continues to send data after one clock cycle, the data is ignored.

Output Signals

Model the `Data` and `Write Master to Slave` bus signals at the DUT output interface.

- `Data`: The data that you want to transfer, valid each cycle of the transaction.
- `Write Master to Slave` bus that consists of:
 - `wr_addr`: Starting address of the write transaction that is sampled at the first cycle of the transaction. The address is specified in bytes.
 - `wr_len`: The number of data values that you want to transfer, sampled at the first cycle of the transaction. The `wr_len` signal is specified in words. This means each unit of `wr_len` is a complete data element. For example, when `wr_len` is 2, and the bit width of data is 128 bit, two 128-bit data elements are written.
 - `wr_valid`: When this control signal becomes high, it indicates that the `Data` signal sampled at the output is valid.

Input Signals

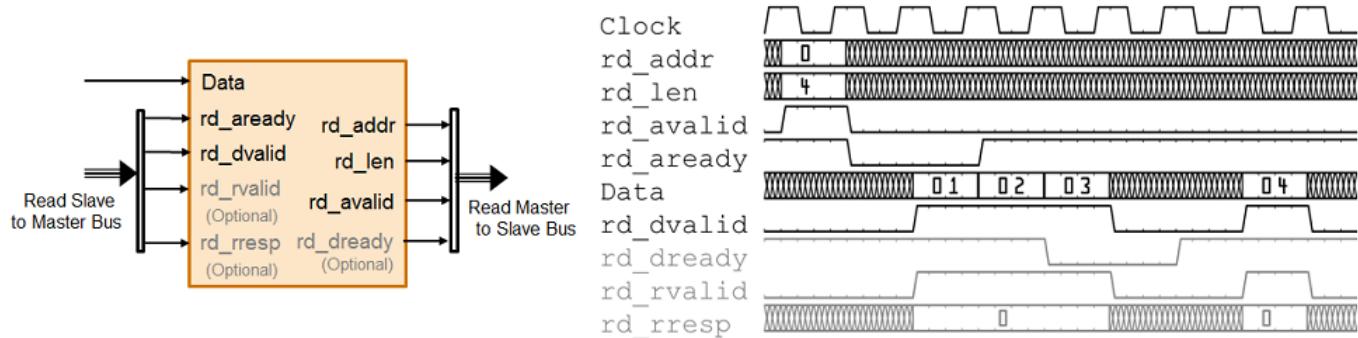
Model the `Write Slave to Master` bus that consists of:

- `wr_complete` (optional signal): Control signal that remains high for one clock cycle indicates that the write transaction has completed. The next burst of data can be sent after `wr_complete` asserts. The early assertion of `wr_complete` makes the average latency nearly 3 clock cycles between two bursts, which makes the write operation pipelined and improves the write throughput.
- `wr_ready`: This signal corresponds to the back pressure from the slave IP core or external memory. When this control signal goes high, it indicates that data can be sent. When `wr_ready` is low, the DUT must stop sending data within one clock cycle. You can also use the `wr_ready` signal to determine whether the DUT can send a second burst signal immediately after the first burst signal has been sent. Multiple burst signals are supported, which means that the `wr_ready` signal remains high to accept the second burst immediately after the last element of the first burst has been accepted. Using `wr_ready` to determine when to start the next burst can reduce the average latency between two bursts to less than 3 clock cycles.
- `wr_bvalid` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. The `wr_bvalid` signal becomes high after the AXI4 interconnect accepts each burst transaction. If `wr_len` is greater than 256, the AXI4 Master write module splits the large burst signal into 256-sized bursts. `wr_bvalid` becomes high for each 256-sized burst.
- `wr_bresp` (optional signal): Response signal from the slave IP core that you can use for diagnosis purposes. Use this signal with the `wr_bvalid` signal.

The AXI4 Master protocol supports a maximum burst size of 256. When you have a large burst of size greater than 256, the AXI Master interface in the generated HDL IP core divides the large burst into multiple smaller bursts with size 256. Therefore, even for large bursts of data, you see an improved write throughput.

Simplified AXI4 Master Protocol - Read Channel

This figure shows the timing diagram for the signals that you model at the DUT input and output interfaces for an AXI4 Master read transaction. These signals include the Data, Read Master to Slave Bus, and Read Slave to Master Bus.



The DUT waits for `rd_ready` to become high to initiate a read request. When `rd_ready` is high, the DUT can send out the read request. The read request consists of the `rd_addr`, `rd_len`, and `rd_valid` signals of the Read Master to Slave bus. The slave IP or the external memory responds to the read request by sending the `Data` at each clock cycle. The `rd_len` signal corresponds to the number of data values to read. The DUT can receive `Data` as long as `rd_dvalid` is high.

Read Request

To model a read request, at the DUT output interface, model the Read Master to Slave bus that consists of:

- `rd_addr`: Starting address for the read transaction that is sampled at the first cycle of the transaction. The address is specified in bytes.
- `rd_len`: The number of data values that you want to read, sampled at the first cycle of the transaction. The `rd_len` signal is specified in words. This means each unit of `rd_len` is a complete data element. For example, when `rd_len` is 2, and the bit width of data is 128 bit, two 128-bit data elements are read.
- `rd_valid`: Control signal that specifies whether the read request is valid.

At the DUT input interface, implement the `rd_ready` signal. This signal is part of the Read Slave to Master bus and indicates when to accept read requests. You can monitor the `rd_ready` signal to determine whether the DUT can send consecutive burst requests. When `rd_ready` becomes high, it indicates that the DUT can send a read request in the next clock cycle.

Read Response

At the DUT input interface, model the `Data` and Read Slave to Master bus signals.

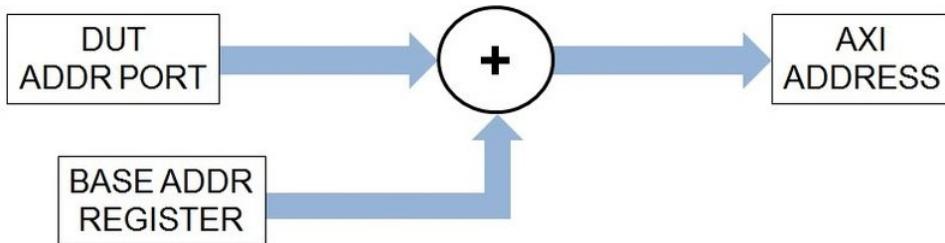
- `Data`: The data that is returned from the read request.
- Read Slave to Master bus that consists of:
 - `rd_dvalid`: Control signal which indicates that the `Data` returned from the read request is valid.

- `rd_rvalid` (optional signal): response signal from the slave IP core that you can use for diagnosis purposes.
- `rd_rresp` (optional signal): Response signal from the slave IP core that indicates the status of the read transaction.

At the DUT output interface, you can optionally implement the `rd_dready` signal. This signal is part of the Read Master to Slave bus and indicates when the DUT can start accepting data. By default, if you do not map this signal to the AXI4 Master read interface, the generated HDL IP core ties `rd_dready` to logic high.

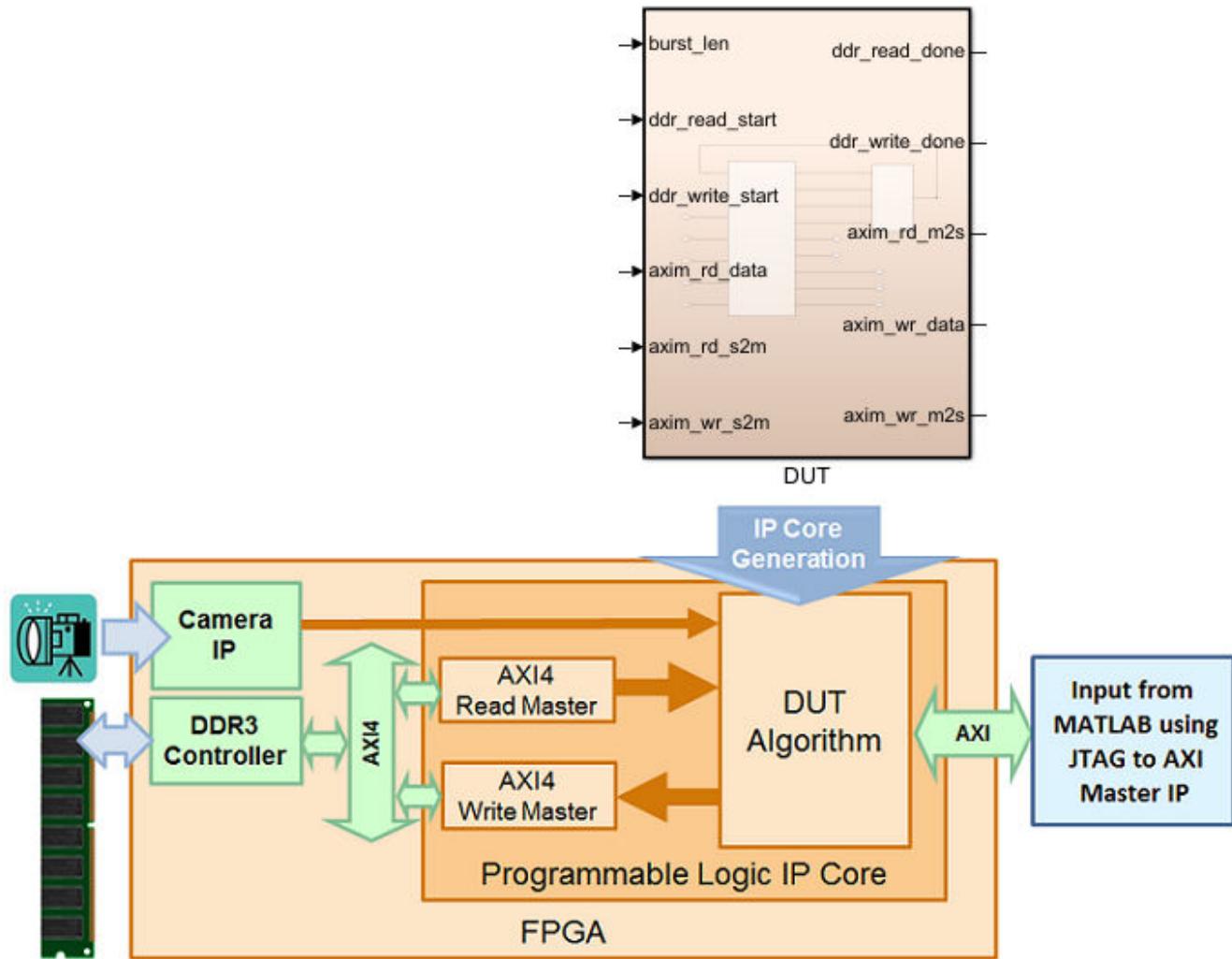
Base Address Register Calculation

For IP cores that you generate, HDL Coder includes a base address register to support driver authoring for both the AXI4 Master read and write channels. The base address register is added to the address that is specified by the DUT `ADDR` port to form the AXI4 Master address. This capability enables the driver to use an addressing mode that programs a fixed register address with the base address of a buffer. The programmed address together with the DUT `ADDR` port is used to index the buffer. By default, the registers take a value of zero, if you do not use them.

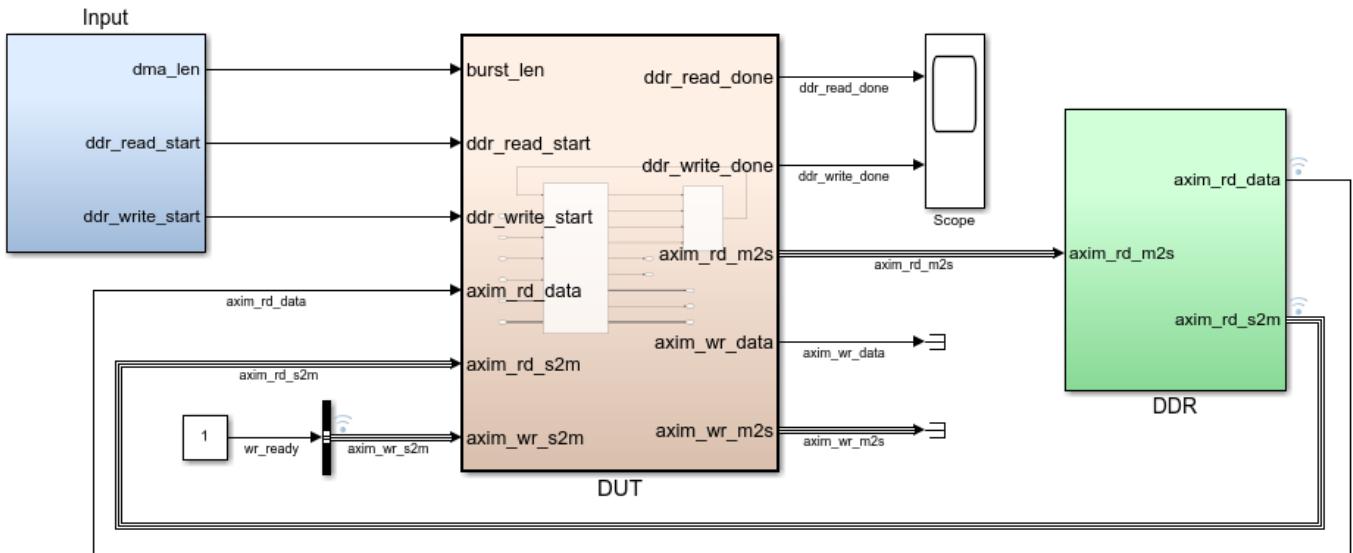


Modeling for AXI4 Master Interfaces

You can model your algorithm with Data and AXI4 Master protocol signals at the DUT ports and then map the signals to AXI4 Master interfaces.



To learn how to model your DUT algorithm for AXI4 Master interface mapping, open this Simulink® model. The DUT Subsystem contains a simple algorithm that reads data from the DDR and writes the data back to a different address in the DDR memory.



Double-click the DUT Subsystem. The DDR_Access_Controller Subsystem models the AXI Master read and write channels and has a Simple Dual Port RAM that calculates the `wr_data` signal. If you double-click the DDR_Access_Controller Subsystem, you see two Edge Detection Subsystem blocks that generate the two start pulses as input to each MATLAB Function block. One Edge Detection Subsystem and DDR Read Controller MATLAB Function models the read transaction. The other Edge Detection Subsystem and DDR Write Controller MATLAB Function models the write transaction. You can modify this design to model only the write transaction or the read transaction by using one Edge Detection Subsystem and the corresponding MATLAB Function block.

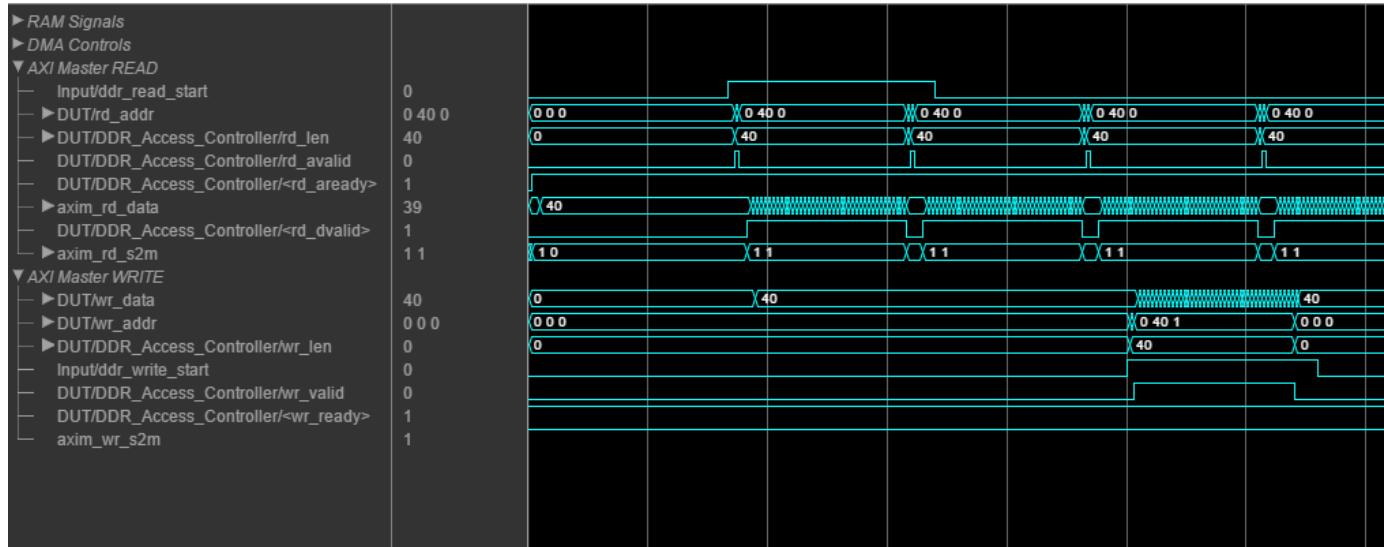
Read Channel

The DDR Read Controller is modeled as a state machine with four states: INIT, IDLE, READ_BURST_START, and DATA_COUNT. The INIT state initializes the read signals and the RAM input signals. When the start signal goes high, the state machine switches to the IDLE state, and then waits for the `rd_ready` signal to become high. When `rd_ready` becomes high, the state machine transitions to the READ_BURST_START state and the DUT starts reading data. The state machine then unconditionally switches to the DATA_COUNT state and continues to read data till `rd_valid` goes low.

Write Channel

The DDR Write Controller is modeled similar to the Read channel as a state machine with four states : IDLE, WRITE_BURST_START, DATA_COUNT, and ACK_WAIT. The DUT is in the IDLE state and then switches to the WRITE_BURST_START state where it waits for the `wr_ready` signal. When `wr_ready` becomes high, the state machine switches to the DATA_COUNT state and starts writing data. The data is valid when `wr_valid` is high. The DUT continues to write data when `wr_ready` is high. As `wr_ready` becomes low, the state machine switches to the ACK_WAIT state and then waits for the ready signal to initiate the next write transaction.

To see the simplified AXI4 Master protocol in effect, simulate the model. If you have DSP System Toolbox™ installed, you can view and analyze the results in the Logic Analyzer.



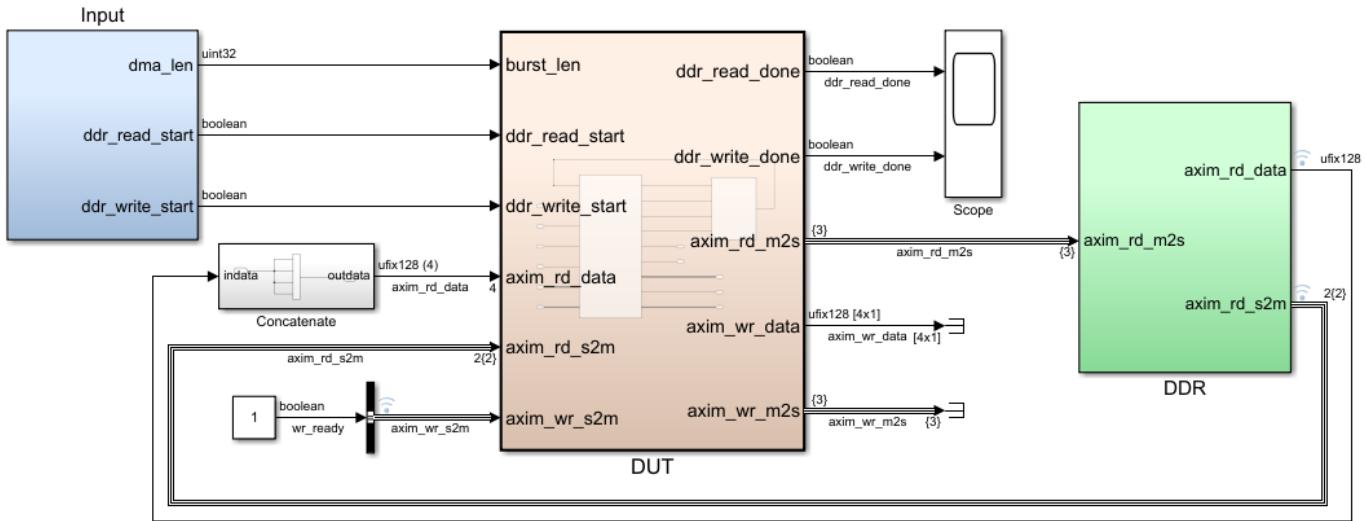
You can use the IP Core Generation workflow to generate an HDL IP core with the AXI4 Master interface. If you have HDL Verifier™ installed, and you use the Xilinx Zynq ZC706 board, then you can integrate the IP core into the Default System with External DDR3 memory access reference design.

Map Vector Ports to AXI4 Master Interfaces

To integrate your HDL IP core into larger reference designs, and to achieve higher throughput when you use the AXI4 Master port to access external DDR memory, you may want to use larger bit widths on the Data port. The AXI4 Master interface bus supports a maximum bit width of 1024 bits.

Simulink supports fixed-point data types that have word length of up to 128 bits. To model your DUT ports with word lengths greater than 128 bits, use vector data types. If you use a vector port such that the combined bit width of all the elements in the vector is greater than 1024 bits, the **Set Target Interface** task displays an error.

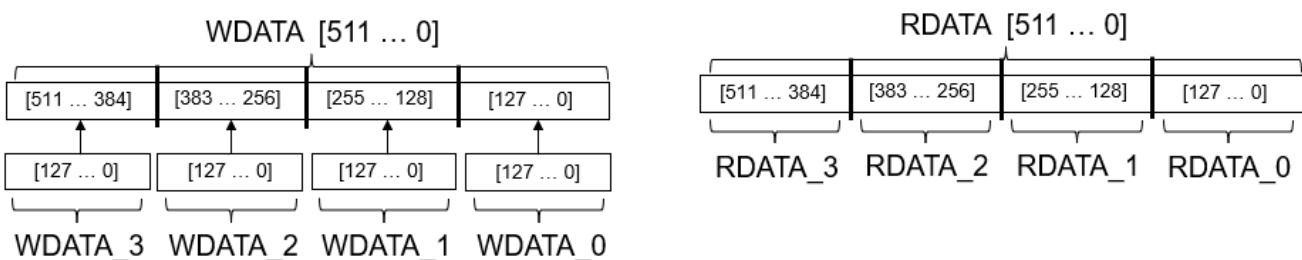
For example, in the `hdlcoder_axi_master` model, to expand the bit width of the `axim_rd_data` port to 512 bits, change the `ddr_data` parameter inside the DDR to `fi(([40:-1:1]),0,128,0)` and then concatenate the 128-bit input four times to generate an output of 512 bits. You can use a Vector Concatenate block to output a combined bit width of 512 bits. To simulate the model, replace the Simple Dual Port RAM block inside the DUT subsystem with a Simple Dual port RAM System.



You can then map these DUT Data ports to AXI4 Master Read or AXI4 Master Write ports in the Target platform interface table, generate the HDL IP core, and integrate the IP core into your Vivado or Qsys reference designs. In the generated HDL code for the DUT IP core, the Data ports are mapped to 512-bit interfaces. Multiple FIFO blocks are generated corresponding to each element of the vector input.

```
ENTITY DUT_ip IS
  PORT( IPCORE_CLK : IN std_logic; -- ufix1
        IPCORE_RESETN : IN std_logic; -- ufix1
        AXI4_Master_Rd_RDATA : IN std_logic_vector(511 DOWNTO 0); -- ufix256
        ...
        ...
        AXI4_Master_Wr_WDATA : OUT std_logic_vector(511 DOWNTO 0); -- ufix256
        ...
      );
END DUT_ip;
```

This figure illustrates the order in which the vector data is written to and read from.



In the HDL code for the DUT IP core, you can see how the AXI4_Master_Rd_RDATA and AXI4_Master_Wr_WDATA interfaces are mapped to the DUT ports and the order in which data is written to the AXI4 Master interface and then read back.

```

...
...
-----
AXI4 Master Read Sequence
-----
AXI4_Master_Rd_RDATA_0 <= AXI4_Master_Rd_RDATA_unsigned(127 DOWNTO 0);
AXI4_Master_Rd_RDATA_1 <= AXI4_Master_Rd_RDATA_unsigned_1(255 DOWNTO 128);
AXI4_Master_Rd_RDATA_2 <= AXI4_Master_Rd_RDATA_unsigned_7(383 DOWNTO 256);
AXI4_Master_Rd_RDATA_3 <= AXI4_Master_Rd_RDATA_unsigned_7(511 DOWNTO 384);

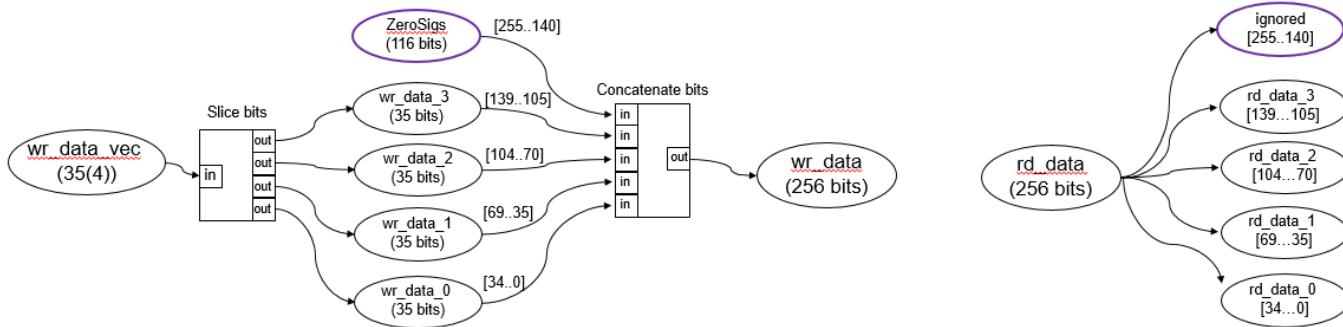
-----
AXI4 Master Write Sequence
-----
AXI4_Master_Wr_WDATA_tmp <= unsigned(AXI4_Master_Wr_WDATA_Vec_3) &
unsigned(AXI4_Master_Wr_WDATA_Vec_2) &
unsigned(AXI4_Master_Wr_WDATA_Vec_1) &
unsigned(AXI4_Master_Wr_WDATA_Vec_0);

AXI4_Master_Wr_WDATA <= std_logic_vector(AXI4_Master_Wr_WDATA_tmp);

...
...

```

If you use a nonstandard bit width for the AXI4 Master Data port, the Data port is upgraded to a standard bit width container that has a bigger size. Standard bit widths include 32, 64, 128, 256, 512, and 1024 bits. For example, if you use a vector that has four 35-bit elements, the resulting bit width of 140 bits (35×4) is mapped to a 256-bit AXI4 Master interface. At the Write channel Data port, bits 255 to 141 are padded with zeroes. At the Read channel Data port, bits 255 to 141 are ignored.



Using nonstandard bit widths can have a performance impact because the entire bandwidth of the AXI4 Master interface is not used. To avoid performance hits, use standard AXI bit widths.

Model Designs with Multiple Sample Rates

The HDL Coder software supports designs with multiple sample rates when you run the IP Core Generation workflow. When you map the interface ports to AXI4 Master interfaces, to use multiple sample rates, ensure that the DUT ports that map to these AXI4 interfaces run at the fastest rate of the design after HDL code generation.

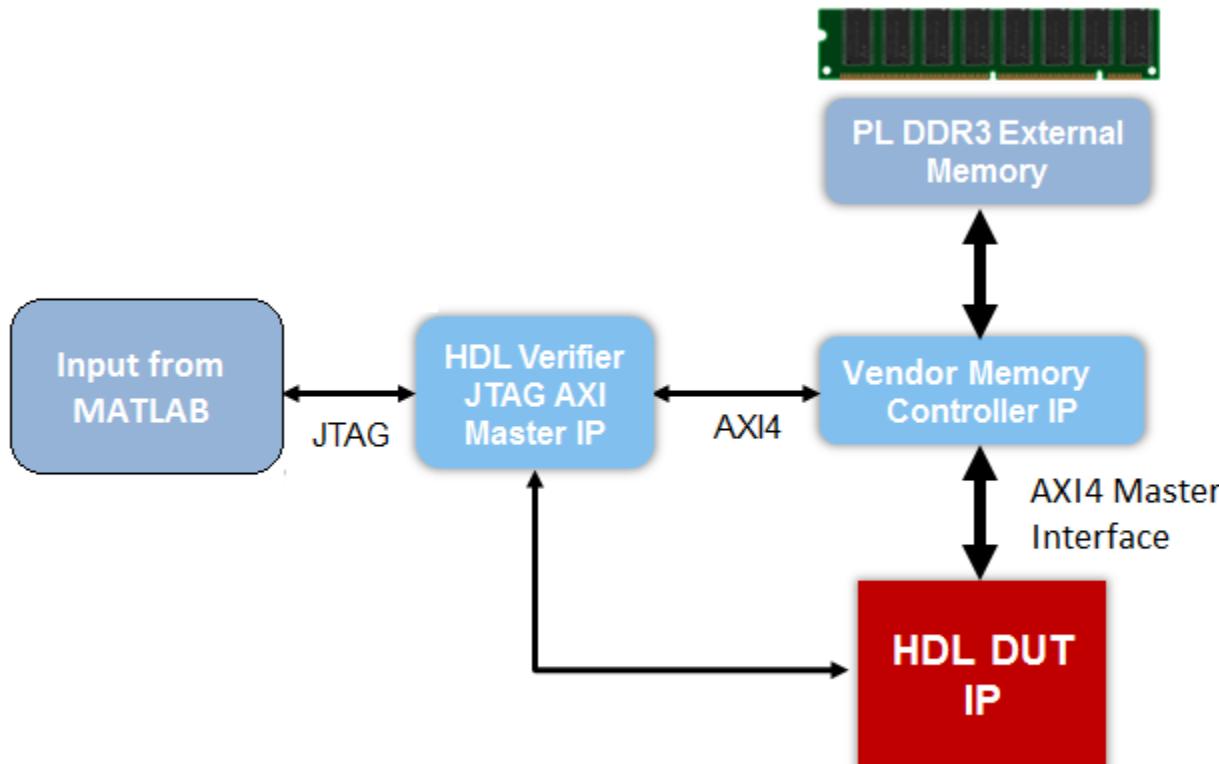
To learn more, see “Multirate IP Core Generation” on page 41-35.

Reference Designs for IP Core Integration

You can integrate the generated HDL IP core with AXI4 Master interfaces into these HDL Coder reference designs:

- Default System with External DDR3 Memory Access: When your target platform is Xilinx Zynq ZC706 evaluation kit.
- Default System with External DDR4 Memory Access: When your target platform is Altera Arria10 SoC development kit.

To use these reference designs, you must have HDL Verifier installed. This figure shows a high level block diagram of the reference design architecture.



In this architecture, the **HDL DUT IP** block corresponds to the IP core that is generated from the **IP Core Generation** workflow. Other blocks in the architecture represent the predefined reference design, that consists of a MATLAB based **JTAG AXI Master IP** that is provided by **HDL Verifier**. After you run the FPGA design on the board, using the **JTAG AXI Master IP**, you can use the input data in MATLAB to initialize the onboard DDR3 external memory. The **HDL DUT IP** core reads the input data from the external memory via the AXI4 Master interface. The IP core then performs the algorithm computation and writes the result to DDR3 memory via the AXI4 Master interface. The **JTAG AXI Master IP** can read the result from DDR3 memory and then verify the result in MATLAB.

Using the `addAXI4MasterInterface` method of the `hdlcoder.ReferenceDesign` class, you can integrate the IP core with AXI4 Master Interface into your own custom reference design.

Restrictions

- **Synthesis tool:** Must be Xilinx Vivado or Altera QUARTUS II. Xilinx ISE is not supported.
- **Target workflow:** Use the IP Core Generation workflow. To run the workflow, open the HDL Workflow Advisor from your DUT algorithm in Simulink. MATLAB to HDL workflow is not supported.
- **Processor/FPGA synchronization:** Must be Free running mode.

See Also

Related Examples

- “Performing Large Matrix Operation on FPGA using External Memory” on page 41-162

More About

- “Model Design for AXI4-Stream Interface Generation” on page 41-10
- “Streaming Pixel Interface” (Vision HDL Toolbox)

IP Core Generation Workflow for Standalone FPGA Devices

In this section...

- “Targeting FPGA Reference Designs with AXI4 Interface” on page 41-90
- “Targeting FPGA Reference Designs Without AXI4 Interface” on page 41-92
- “Board Support” on page 41-92
- “Restrictions” on page 41-92

You can generate a reusable HDL IP core for any supported Xilinx or Altera FPGA device. The workflow produces an IP core report that displays the target interface configuration and the coder settings that you specify. See “Custom IP Core Generation” on page 40-10.

You can optionally build your own custom reference designs and integrate the generated IP core into the reference design. The workflow does not require the Embedded Coder software, because you need not generate the embedded code that is run on the processor. This means that the workflow has a **Generate Software Interface** task, but you cannot generate a software interface model. If you have HDL Verifier installed, on the **Set Target Reference Design** task, set **Insert JTAG MATLAB as AXI Master (HDL Verifier Required)** to on. You can then generate a software interface script in the **Generate Software Interface** task to rapidly prototype and test the HDL IP core functionality by using the MATLAB AXI Master. See “Generate Software Interface to Probe and Rapidly Prototype the HDL IP Core” on page 40-53.

4.2. Generate Software Interface

Analysis

Generate a software interface for the IP core

Input Parameters

<input type="checkbox"/> Generate Simulink software interface mode	Operating system:	<input type="button" value="▼"/>
<input checked="" type="checkbox"/> Generate MATLAB software interface script		

Run This Task

Result:  Passed

Note No driver was generated for port(s) "LED" mapped to interface "LEDs General Purpose" in the software interface script.

Passed Generate Software Interface.

Generating new Zynq Software Interface script: [gs_hdlcoder_led_vector_interface.m](#)

Zynq Software Interface script generation complete.

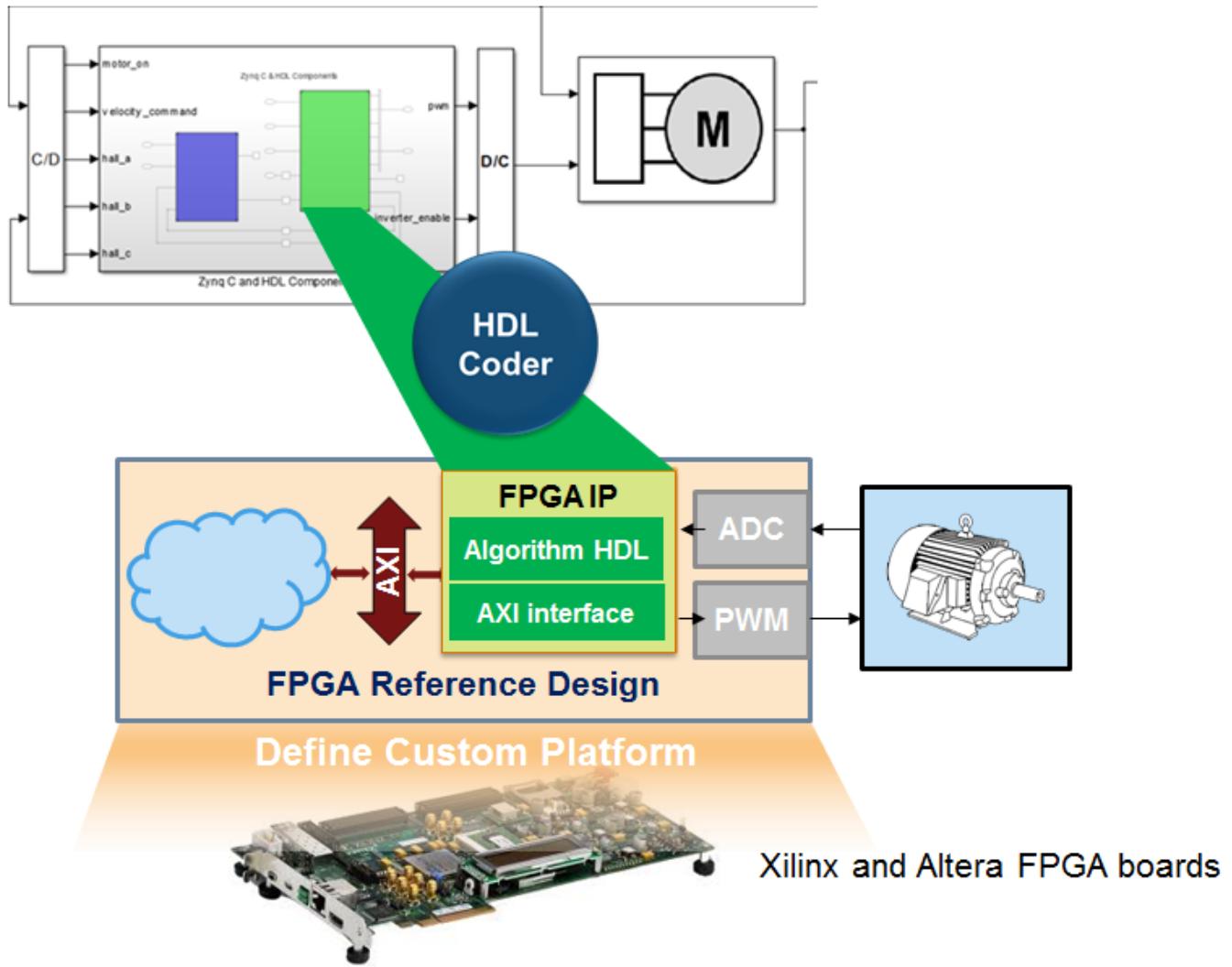
The workflow for the FPGA boards has these features:

- **Set Target Reference Design** task. Populates the reference design, its tool version, and the parameters that you specify.
- **Set Target Interface** task. Map your DUT ports to the interfaces on the target platform.
- **Set Target Frequency** task. Specifies the **Target Frequency (MHz)** to modify the clock module in the reference design to produce a clock signal with that frequency.
- **Generate RTL Code and IP Core** task. Generates a reusable and sharable IP core. The IP core packages the RTL code, a C header file, and the IP core definition files.
- **Create Project** task. Creates a project for integrating the IP core into the predefined reference designs.

You can generate an IP core with an optional AXI4 or an AXI4-Lite interface.

Targeting FPGA Reference Designs with AXI4 Interface

This figure shows how HDL Coder generates an IP core with an AXI4 interface and integrates the IP core into the FPGA reference design. See “Board and Reference Design Registration System” on page 41-39.



Use the HDL Coder generated AXI4-Lite interface to connect the IP core with an AXI4 or AXI4-Lite Master device such as:

- MicroBlaze processor.
- Nios II processor.
- PCIe Endpoint that connects to an external processor.
- JTAG Master.

When you connect the HDL IP core to a processor such as the MicroBlaze, you must integrate the handwritten C code to run on the processor. The generated IP core report displays the register address mapping information. To find the register offsets in the IP core register space, use this mapping information. To get the memory address of each register, add the register offset to the base address that you specify in your reference design. You can also find the register offsets in the C header file in the generated IP core folder.

Targeting FPGA Reference Designs Without AXI4 Interface

In the reference design definition function, you can create your own custom reference designs without the AXI4 slave interface. See also `addAXI4SlaveInterface`.

When creating a custom reference design, to target a standalone FPGA board, use the `EmbeddedCoderSupportPackage` method of the `hdlcoder.ReferenceDesign` class:

```
hRD.EmbeddedCoderSupportPackage = ...
    hdlcoder.EmbeddedCoderSupportPackage.None;
```

See `EmbeddedCoderSupportPackage`.

Board Support

HDL Coder supports these FPGA boards with the IP Core Generation workflow:

- Xilinx Kintex-7 KC705 development board
- Arrow DECA MAX 10 FPGA evaluation kit

Using these boards, you can integrate the generated IP core into the `default_system` reference design. By default, this reference design does not have an AXI4 slave interface. Optionally, you can add the interface in the reference design definition function.

Restrictions

IP Core Generation workflow does not support :

- **RAM Architecture** set to Generic RAM without clock enable.
- Using different clocks for the IP core and the AXI interface. The `IPCore_Clk` and `AXILite_ACLK` must be synchronous and connected to the same clock source. The `IPCore_RESETN` and `AXILite_RESETN` must be connected to the same reset source. See “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 40-25.

See Also

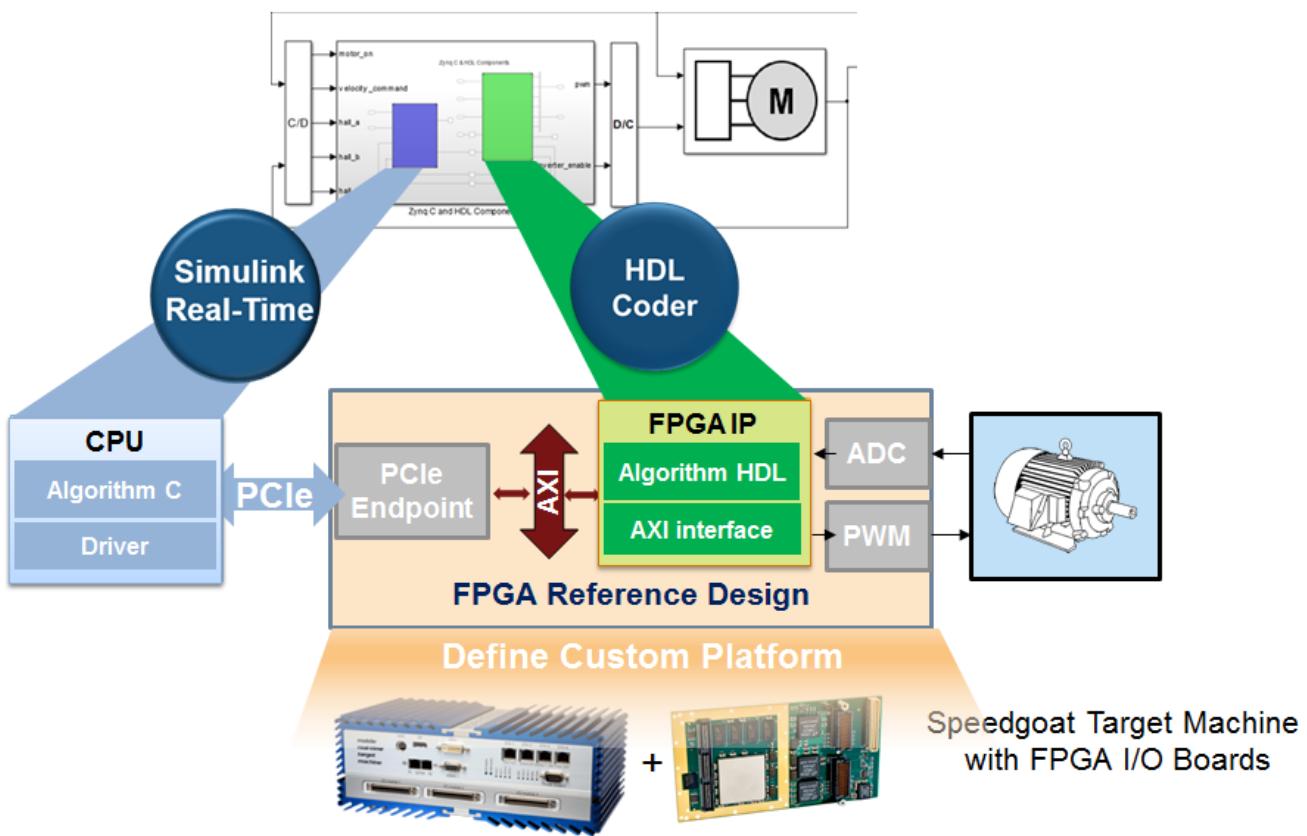
Related Examples

- “IP Core Generation Workflow without an Embedded ARM Processor: Xilinx Kintex-7 KC705” on page 40-153

IP Core Generation Workflow for Speedgoat Simulink-Programmable I/O Modules

HDL Coder uses the IP Core Generation workflow infrastructure to generate a reusable HDL IP core for the Speedgoat Simulink-Programmable I/O modules that support Xilinx Vivado. The workflow produces an IP core report that displays the target interface configuration and the code generator settings that you specify. You can integrate the IP core into a larger design by adding it in an embedded system integration environment. See “Custom IP Core Generation” on page 40-10.

This figure shows how the software generates an IP core with an AXI interface and integrates the IP core into the FPGA reference design.



Supported I/O Modules

To learn about I/O modules that HDL Coder supports with the Simulink Real-Time FPGA I/O workflow, see “Speedgoat FPGA Support with HDL Workflow Advisor” on page 40-8.

IP Core Generation Workflow

This workflow has these key features:

- Uses Xilinx Vivado as the synthesis tool.
- Generates a reusable and sharable IP core. The IP core packages the RTL code, a C header file, and the IP core definition files.
- Creates a project for integrating the IP core into the Speedgoat reference design.
- Generates an FPGA bitstream and downloads the bitstream to the target hardware.

1.1. Set Target Device and Synthesis Tool

Analysis (^Triggers Update Diagram)

Set Target Device and Synthesis Tool for HDL code generation

Input Parameters

Target workflow: Simulink Real-Time FPGA I/O

Target platform: Speedgoat IO397-50k

Synthesis tool: Xilinx Vivado

Family: Artix7

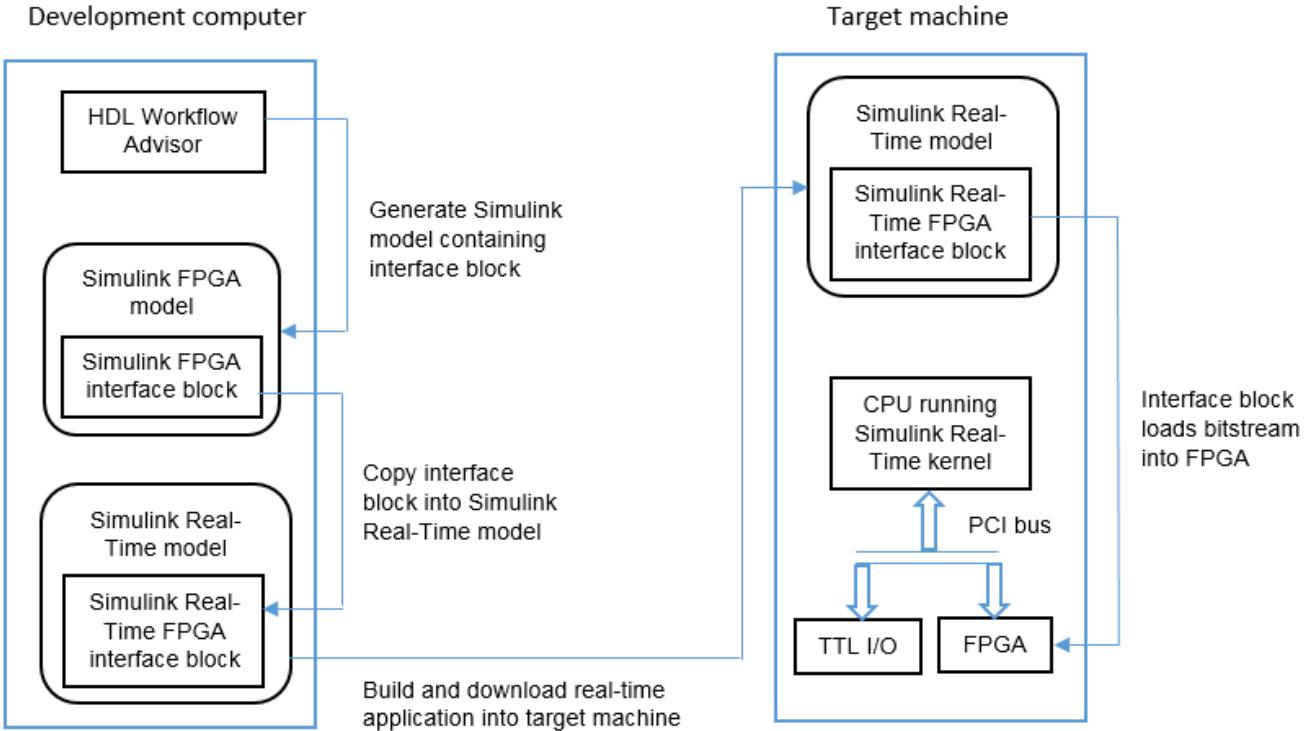
Package: csg325

Project folder: hdl_prj

Result:  Passed

Passed Set Target Device and Synthesis Tool.

After building the FPGA bitstream, the workflow generates a Simulink Real-Time model. The model is an interface subsystem model that contains the blocks to program the FPGA and communicate with the I/O module through the PCIe bus during real-time execution.



Restrictions

IP Core Generation workflow does not support :

- **RAM Architecture** set to Generic RAM without clock enable.
- Using different clocks for the IP core and the AXI interface. The **IPCore_Clk** and **AXILite_ACLK** must be synchronous and connected to the same clock source. The **IPCore_RESETN** and **AXILite_RESETN** must be connected to the same reset source. See “Synchronization of Global Reset Signal to IP Core Clock Domain” on page 40-25.

See Also

More About

- “Speedgoat FPGA Support with HDL Workflow Advisor” on page 40-8
- “FPGA Programming and Configuration on Speedgoat Simulink-Programmable I/O Modules” on page 41-63
- “Custom IP Core Generation” on page 40-10

External Websites

- www.speedgoat.com/support

IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip

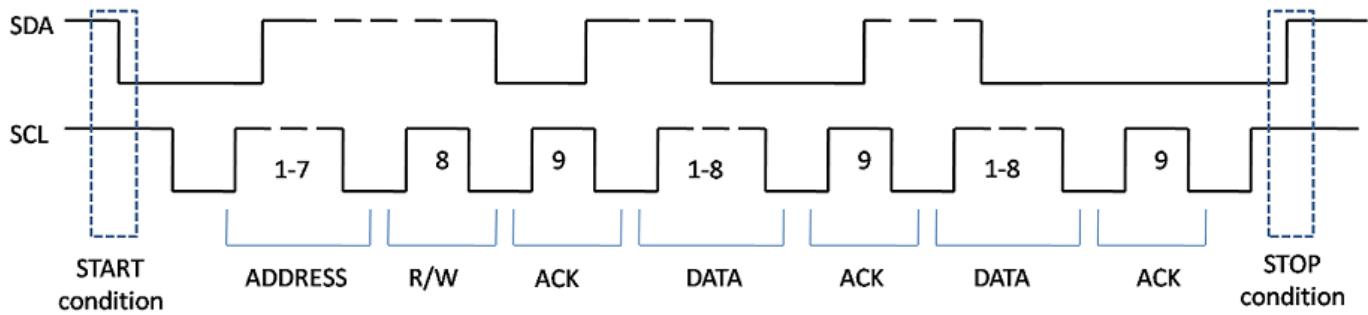
This example illustrates how to model an I2C controller using an I2C Master controller modeled using Stateflow™ blocks for configuring the audio codec chip.

In this example, you:

- 1 Model the I2C Master Controller using Stateflow® blocks in Simulink®
- 2 Model the I2C Controller using the I2C Master Controller block for configuring the Audio Codec Chip
- 3 Use the blackbox subsystem and bidirectional port features to handle tri-state logic in I2C IP core
- 4 Use the IP Core Generation workflow to generate an IP core for the I2C Controller

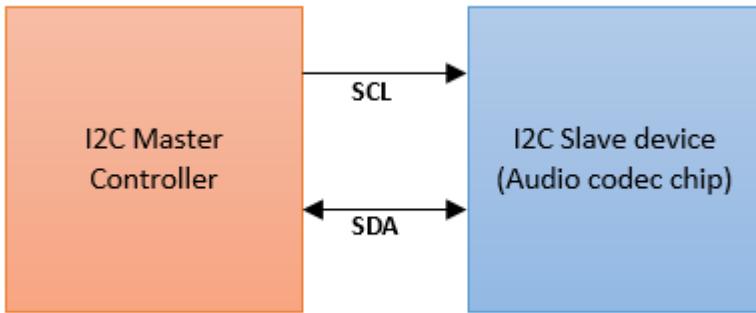
1. Overview of I2C protocol

I2C bus, also called Inter-IC bus, is a simple, multi-master, multi-slave, bidirectional two-wire bus, that consists of serial data (SDA) and serial clock (SCL) lines. Each device connected to the bus is software addressable by a unique 7-bit or 10-bit address, and maintains a simple master-slave relationship. Serial, 8-bit oriented, bi-directional data transfers can be made at up to 100 kbit/s in the Standard mode, up to 400 kbit/s in the Fast-mode, or up to 3.4 Mbit/s in the High-speed mode. I2C bus has two nodes: master node and slave node. The master node generates clock and initiates communication with the slave. The slave node which is addressed by the master receives clock and responds to the master during acknowledgment. There are four modes of operation which are master transmit, master receive, slave transmit and slave receive. The master starts the communication by sending start bit followed by 7 or 10 bit address of the slave followed by read(1) or write(0) bit. If the slave corresponding to that address is present, then it responds with ACK bit. The master continues communication in transmit or receive mode based on the read or write bit. Similarly, the slave continues its operation based on the read or write instruction from the master. The figure below shows the timing diagram of I2C protocol.

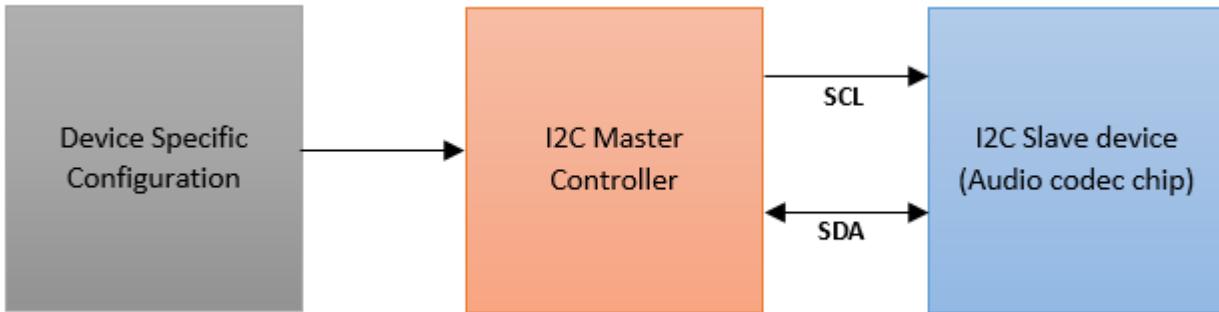


2. Modeling Generalized I2C Master Controller in Simulink Using Stateflow Blocks

Configuring multiple peripherals in the design can be a cumbersome and tedious process. Instead, create a generic I2C Master Controller that you can directly use to configure the audio codec chips. The figure below shows the architecture of the Generalized I2C Master Controller which is implemented using Stateflow blocks in Simulink.



The above part shows the I2C Master Controller block. To configure the peripheral, you must provide device-specific configuration to the I2C Master Controller block. The block diagram to configure the audio codec chip using I2C Master Controller is as shown below.

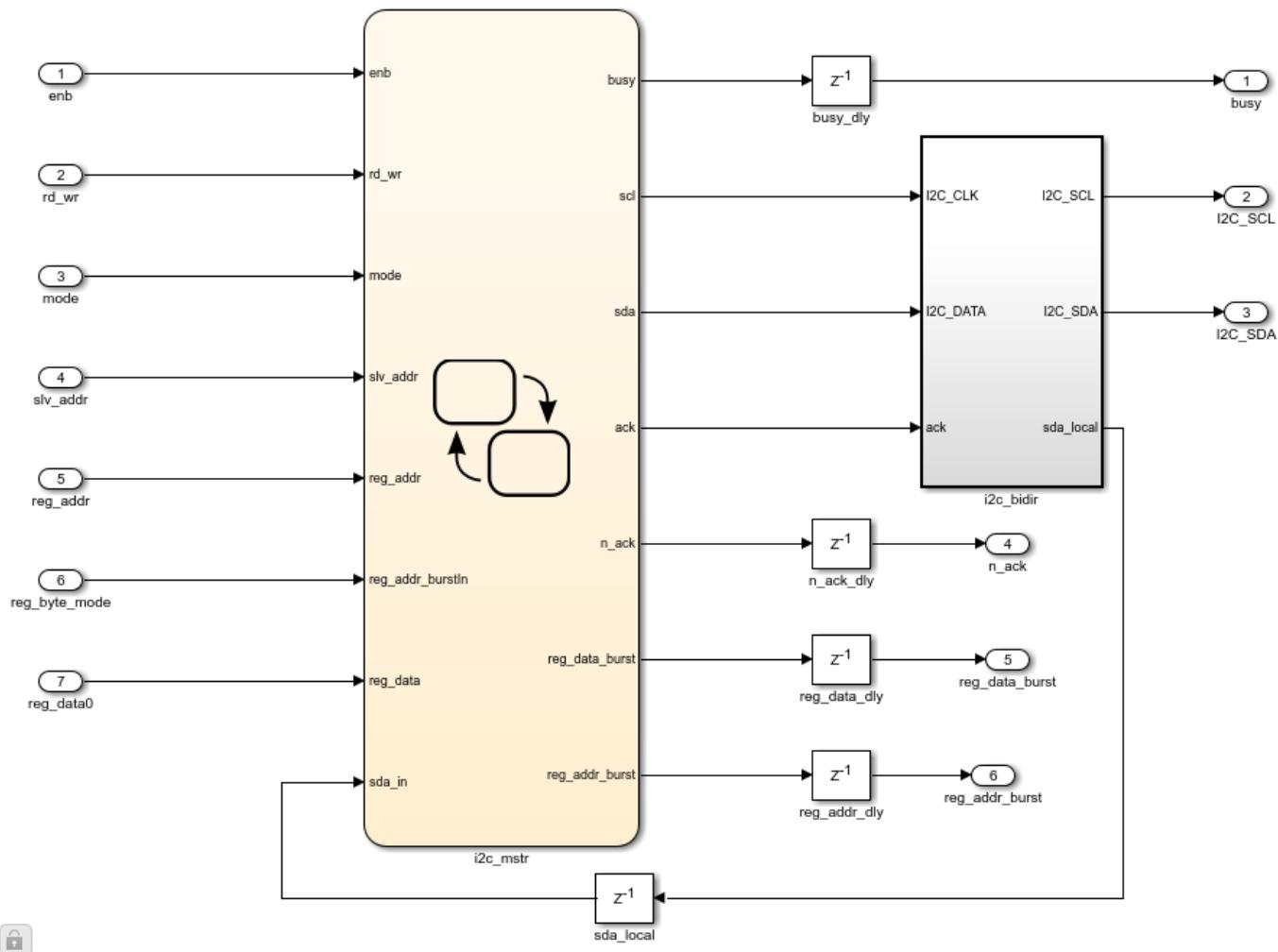


The below model shows the I2C Master Controller which is modeled in Simulink using Stateflow blocks.

```

modelname = 'hdlcoder_I2C_master_controller';
open_system(modelname);
open_system('hdlcoder_I2C_master_controller/I2C_MasterController');

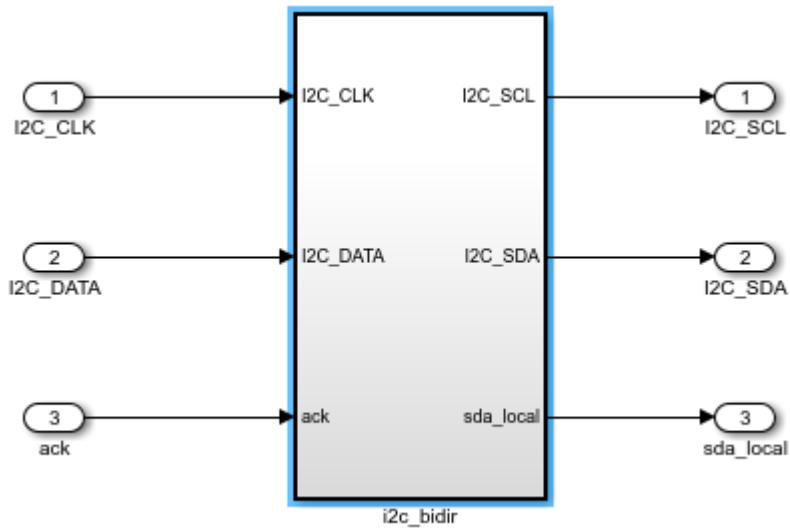
```



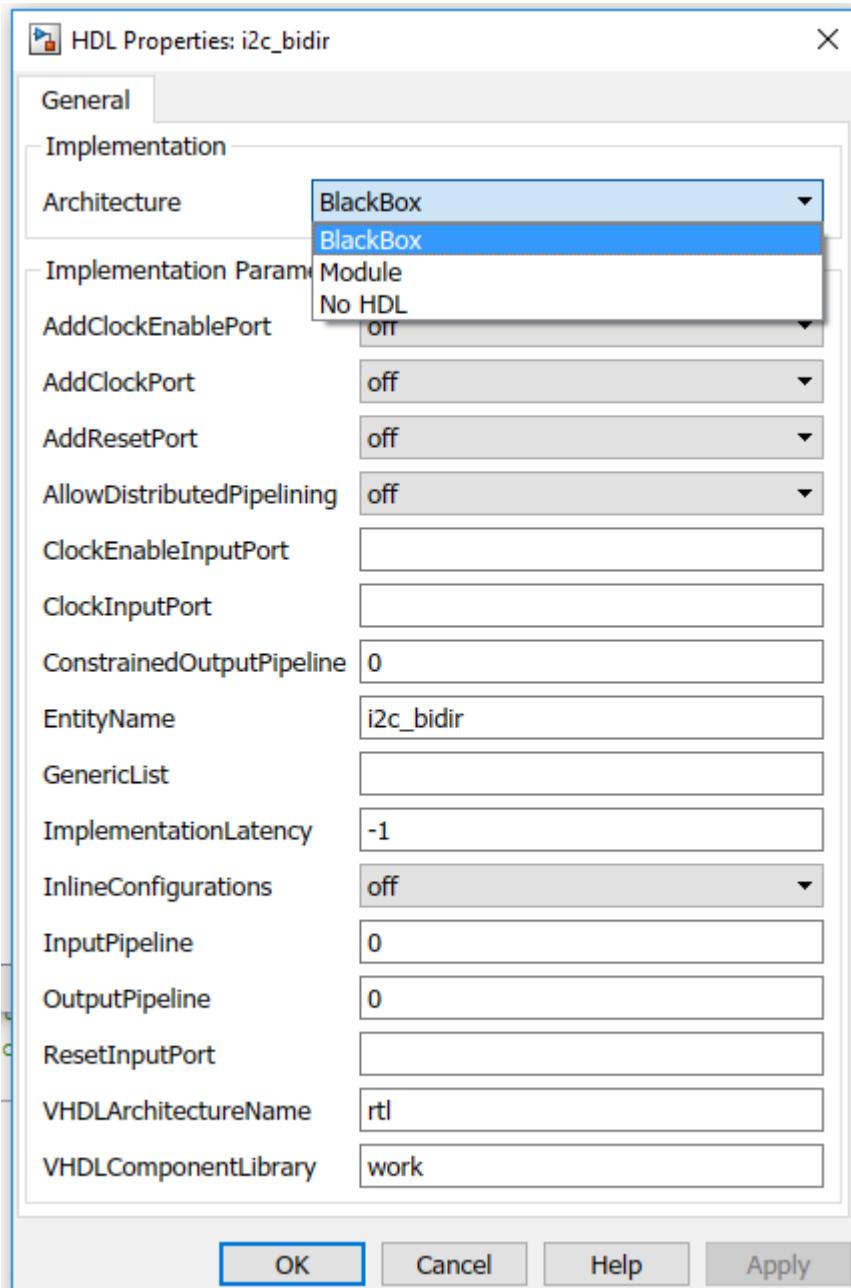
The I2C Master Controller only supports I2C write and doesn't support I2C readback currently. I2C Master Controller consists of two parts, I2C Master Controller chart and tristate buffer blackbox. I2C Master Controller chart provides serial data, SDA and serial clock, SCL to slave device through the tristate buffer blackbox. Tristate buffer blackbox uses the handwritten VHDL code and used for the bidirectional functionality of the SDA port. Tristate buffer blackbox is added in the model as Simulink doesn't support bidirectional port modeling.

To create a blackbox use the following steps.

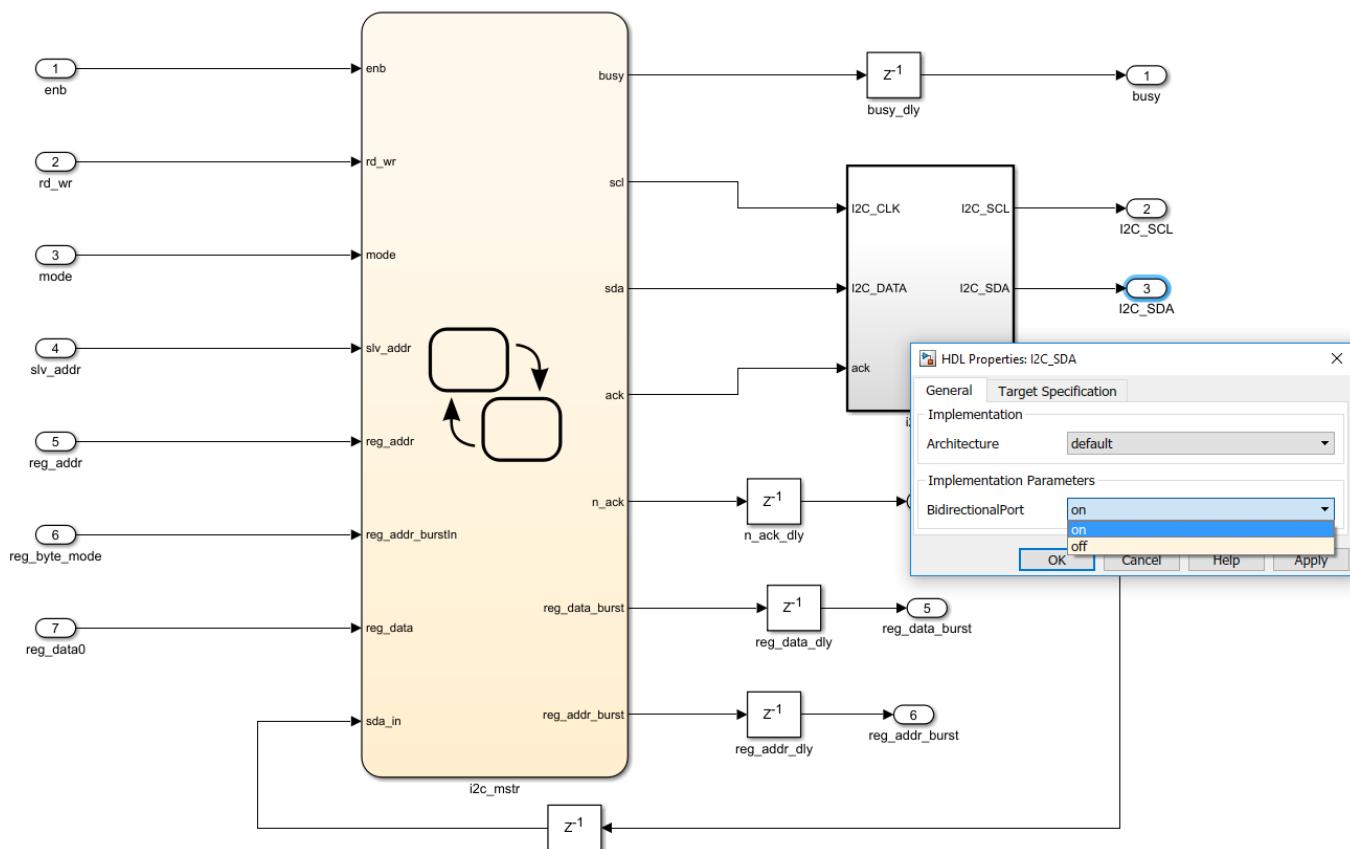
1. Make a subsystem which contains input and output ports of the HDL source code which you want to import for the blackbox creation. The I2C tristate buffer blackbox is as shown below.



2. To specify your subsystem as a black box interface, right click on the subsystem and select **HDL Code > HDL Block Properties** and set the Architecture to Blackbox as shown in the following figure.



3. The data port **I2C_SDA** of I2C Master Controller is bidirectional. To set the port as bidirectional, right click on the **I2C_SDA** port and click on HDL block properties and set the BidirectionalPort on as shown below.



4. During simulation, the actual content inside of the blackbox subsystem will be used for simulation.

During code generation, HDL Coder will not generate the code under the blackbox subsystem. Instead, the code generator integrates your hand-written HDL code into the IP core. Inputs to the I2C Master Controller block can be provided by adding a device specific configuration chart at the input. This chart contains details about the registers that need to be configured for your slave device. More about device configuration is covered in the section of Zedboard, Zybo board and Arrow SoC Development Kit Audio Codec configuration using I2C Master Controller.

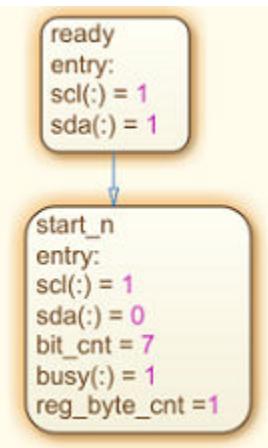
2.1 I/O Description of I2C Master Controller block

The following figure gives the details about input and output ports of the I2C Master Controller block.

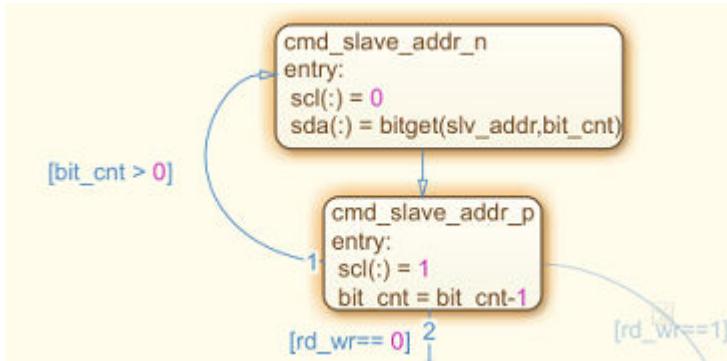
Name	Type	Width (in bits)	Description
enb	Input	1	I2C master controller enable
rd_wr	Input	1	Read/Write cycle
mode	Input	1	Mode of operation Normal = 0 Burst = 1
slv_addr	Input	8	Device address
reg_addr	Input	8	Device register address
reg_byte_mode	Input	1	Register address mode Byte = 0 Burst = 1
reg_data0	Input	8	Device register data
busy	Output	1	Status signal of the master controller chart
I2C_SCL	Output	1	Serial clock to slave device
I2C_SDA	Output	1	Serial data to slave device
n_ack	Output	1	No acknowledgment
reg_data_burst	Output	1	Burst data enable signal
Reg_addr_burst	Output	1	Burst address enable signal

2.2 Description of I2C Master Controller Stateflow chart

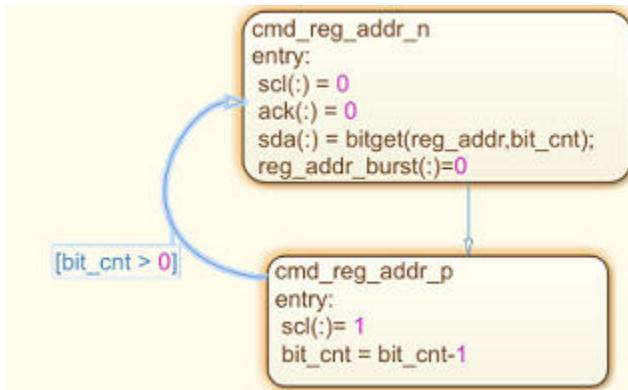
I2C Master Controller chart is made in such a way that in all the states required clock (SCL) is generated and data is provided as per the I2C protocol through serial data (SDA) port. Following states shows the generation of the clock and start bit.



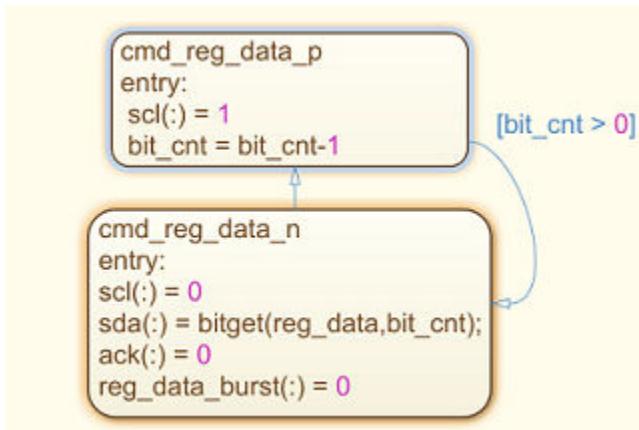
The following states are used to send 7-bit address of the slave device.



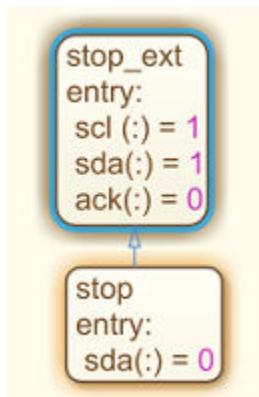
The function **bitget** is used to send bits serially to SDA port. It allows user to get bit value at specified position of the integer mentioned in its argument list. The transition from one state to other state depends on execution order specified for the transition conditions. As shown in the above figure the transition from **cmd_slave_addr_p** to the **cmd_slave_addr_n** state occurs based on the transition condition($\text{bit_cnt} > 0$). The value of bit_cnt keeps decrementing until the transition condition satisfies. The value of bit_cnt is initialized to '7' and its value decrements till it becomes '1' which is used to send 7-bit address of slave device on SDA port. For HDL code generation, supported data types must be used. Colon(:) operator as shown in the states which is a typecasting operator used in **cmd_slave_addr_n** state($\text{scl}(:) = 0$) converts a value of type 'double' to a type 'logical' (SCL is the logical datatype in the states shown). The states shown below are used to send register address to the slave device.



The states shown below are used to send register data to the slave device.



Following states shows the stop bit generation.



3. Configuring audio codec ADAU1761 on Zedboard using I2C Master Controller library block

This section shows how to:

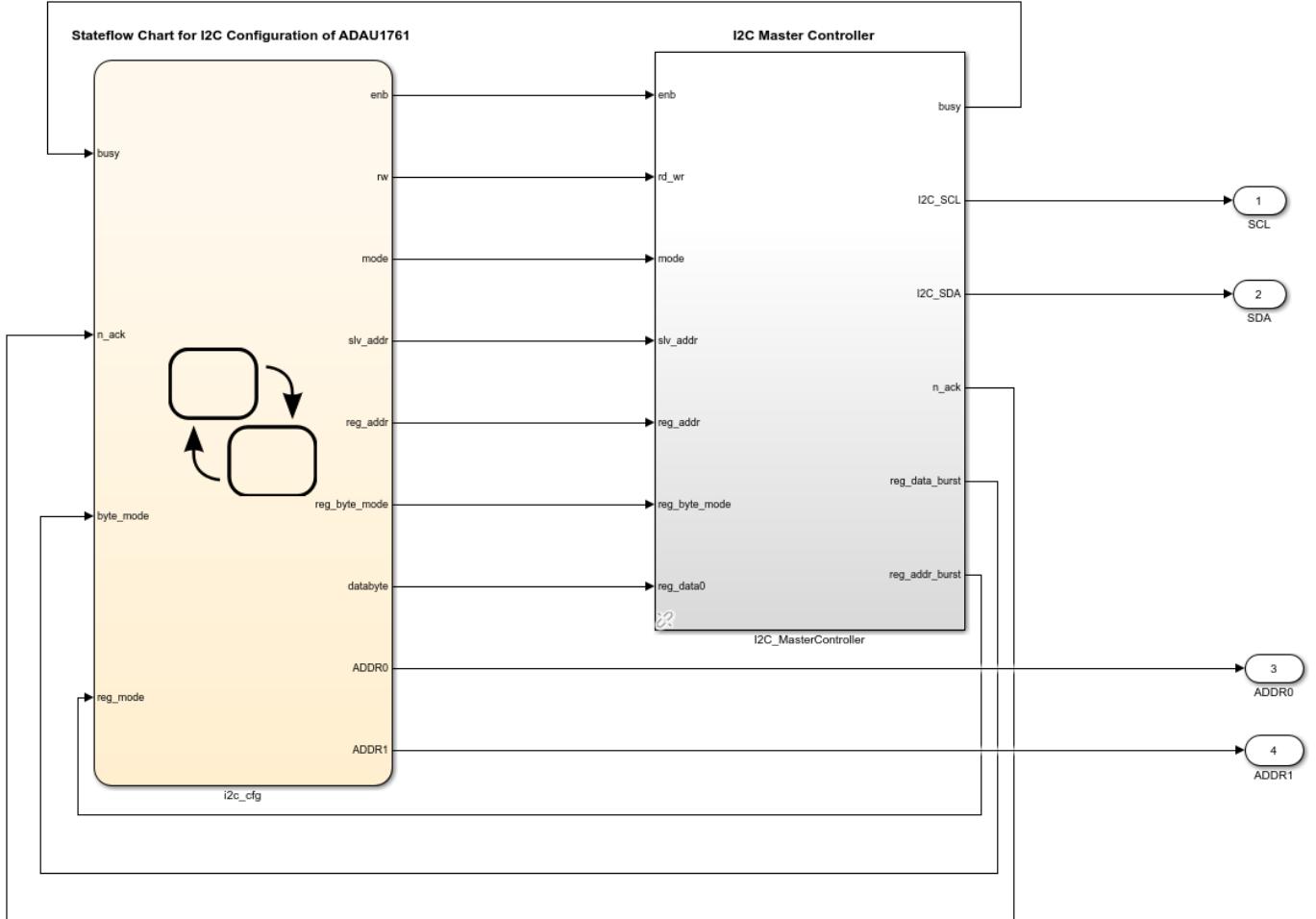
- 1 Model audio codec ADAU1761 device configuration chart using Stateflow blocks in Simulink.
- 2 Use I2C Master controller library block to configure audio codec ADAU1761.
- 3 Perform simulation of created model.

As mentioned earlier to configure the audio codec ADAU1761 on Zedboard, device configuration chart for ADAU1761 need to be created. This chart should be connected to the I2C Master controller library block created earlier.

Note: You have to create device configuration chart for your own device. This example is to show how the I2C Master Controller library block can be used to configure audio codec devices. The device configuration chart used for ADAU1761 is specific to this device and can't be used to configure other devices.

The configuration model created for ADAU1761 is as shown below.

```
modelname = 'hdlcoder_I2C_adau1761';
open_system(modelname);
```



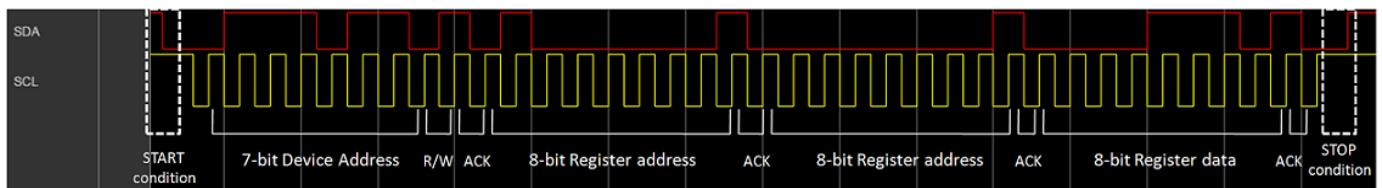
3.1 Simulating audio codec ADAU1761 configuration model

For Audio codec chip ADAU1761, 20 registers need to be configured. Few of them have to be written by I2C Master Controller in burst mode and few in byte mode. The first register is written in byte mode, second in burst mode of length 6-bytes. Remaining 18 registers are written in byte mode.

Simulation waveform for configuration of audio codec ADAU1761 is as shown below.

Byte mode transfer between I2C Master Controller and audio codec chip ADAU1761 is shown in the following figure.

Below simulation shows sending of start bit, followed by 7-bit address of the slave device(0x3B), followed by write(0) bit, followed by 16-bit register address(0x4000), followed by 8-bit register data(0xE) and acknowledgements from the slave device.



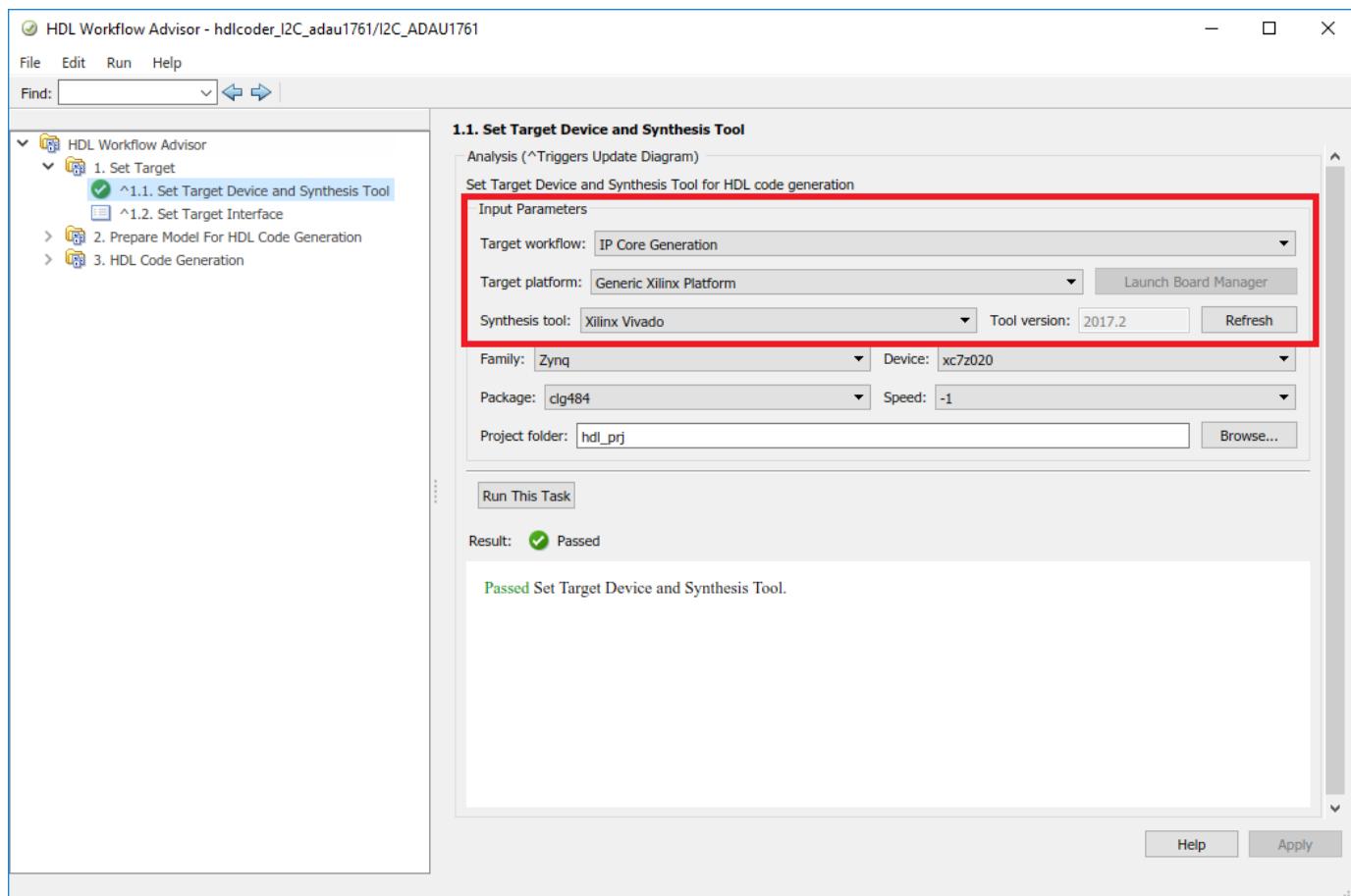
3.2 IP Core generation workflow

To generate the audio codec ADAU1761 configuration HDL IP core, follow the steps given below.

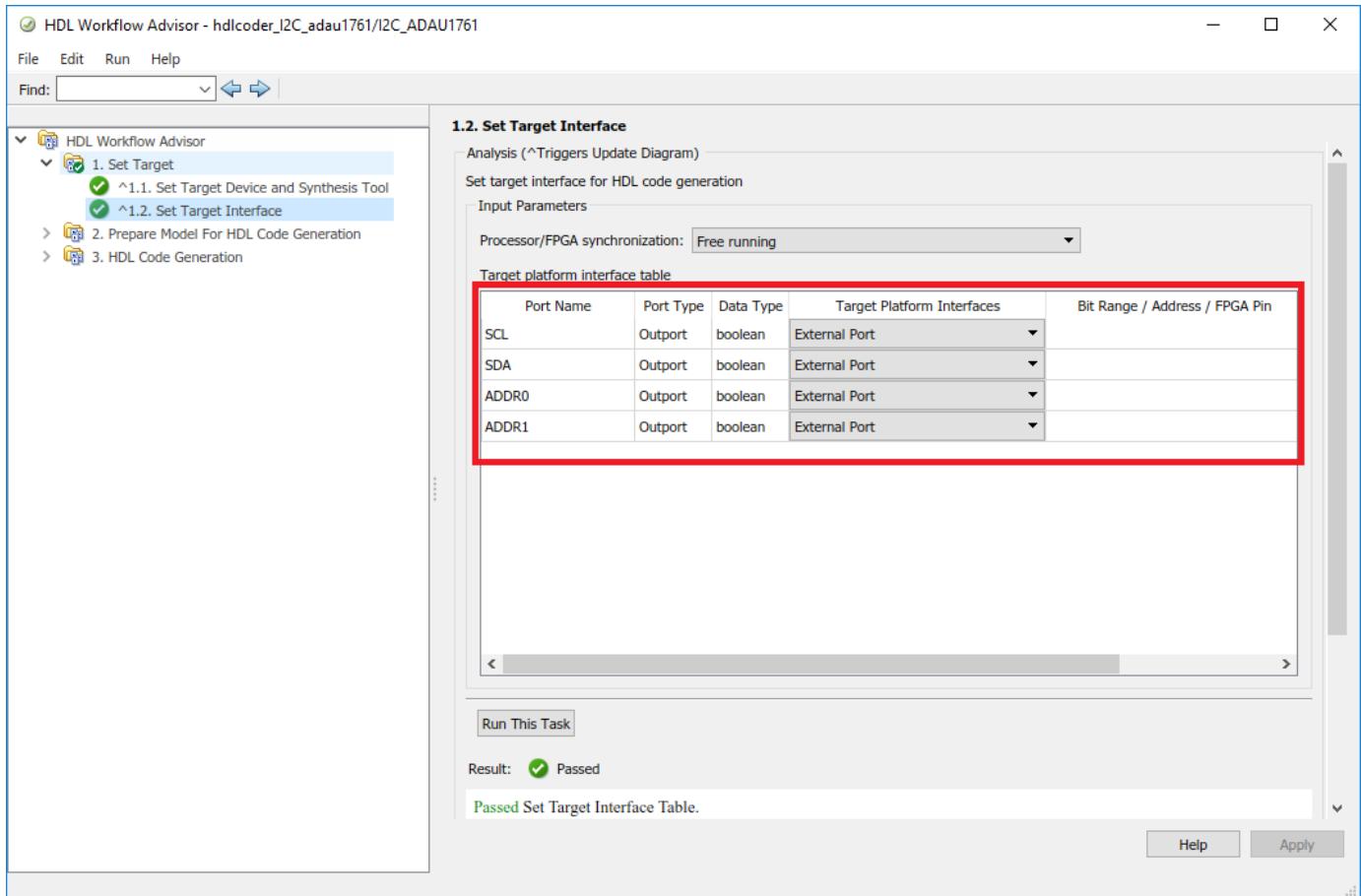
- Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command

```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2017.4\bin\vivado.bat')
```

- In the ADAU1761 configuration model, select I2C_IP subsystem and by right clicking open HDL workflow advisor. In Task 1.1, Select **IP Core Generation** for Target workflow, **Generic Xilinx Platform** for Target platform and **Xilinx Vivado** for Synthesis Tool. Also select family, device, package and speed as shown in the figure below.



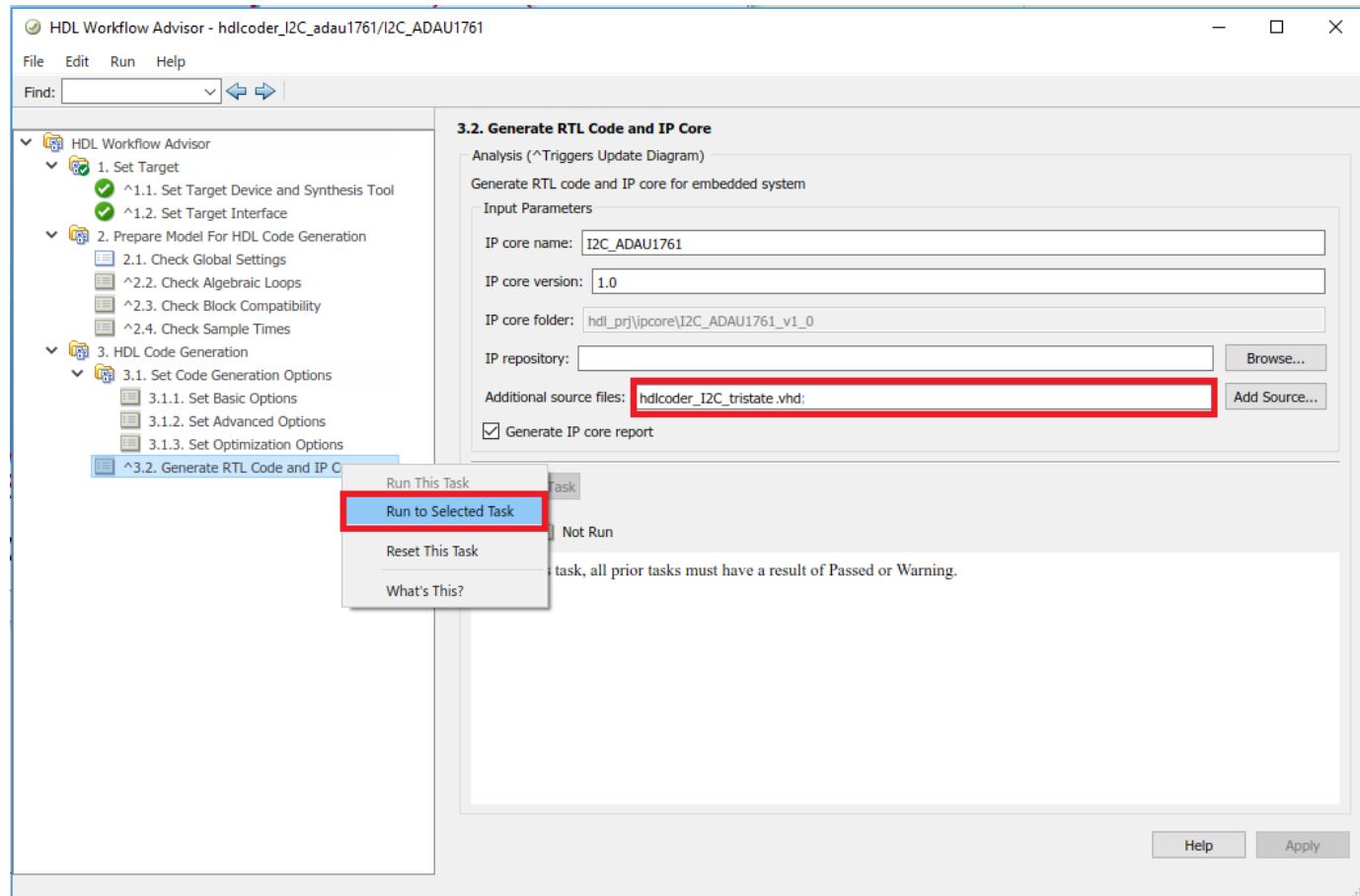
- In Task 1.2, set the Target Platform Interfaces to "External Port" for all the ports.



4. The Tristate HDL logic is present in the VHDL file `hdlcoder_I2C_tristate.vhd`. Copy it into your current working directory.

```
copyfile(fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos', 'hdlcoder_I2C_tristate.vhd'),  
'hdlcoder_I2C_tristate.vhd');
```

5. In Task 3.2, add the tristate buffer VHDL file in the additional source files. Then right click on Generate RTL code and IP Core and click on Run to Selected Task.



I2C IP Core for configuration of ADAU1761 will be generated. Below figure shows the IP Core generation report.

Code Generation Report

Find: Match Case

Contents

- Summary
- Clock Summary
- Code Interface Report
- Timing And Area Report
 - High-level Resource Report
 - Critical Path Estimation
- Optimization Report
 - Distributed Pipelining
 - Streaming and Sharing
 - Delay Balancing
 - Adaptive Pipelining
- IP Core Generation Report**
- Traceability Report

IP Core Generation Report for hdlcoder_I2C_adau1761

Summary

IP core name	I2C_ADAU1761
IP core version	1.0
IP core folder	hdl_prj\ipcore\I2C_ADAU1761_v1_0
IP core zip file name	I2C_ADAU1761_v1_0.zip
Target platform	Generic Xilinx Platform
Target tool	Xilinx Vivado
Target language	VHDL
Model	hdlcoder_I2C_adau1761
Model version	1.455
HDL Coder version	3.11
IP core generated on	13-Oct-2017 12:07:53
IP core generated for	I2C_ADAU1761

Target Interface Configuration

You chose the following target interface configuration for [hdlcoder_I2C_adau1761](#):

Processor/FPGA synchronization mode: Free running

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
SCL	Outport	boolean	External Port	
SDA	Outport	boolean	External Port	
ADDR0	Outport	boolean	External Port	
ADDR1	Outport	boolean	External Port	

Register Address Mapping

The following AXI4-Lite bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8	contains unique IP timestamp (yymmddHHMM): 1710131207

[OK](#) [Help](#)

Generated IP core can be used in user reference designs. For creating the reference design, refer to “Authoring a Reference Design for Audio System on a Zynq Board” on page 41-170.

4. Configuring audio codec SSM2603 on Zybo board using I2C Master Controller library block

This section shows how to:

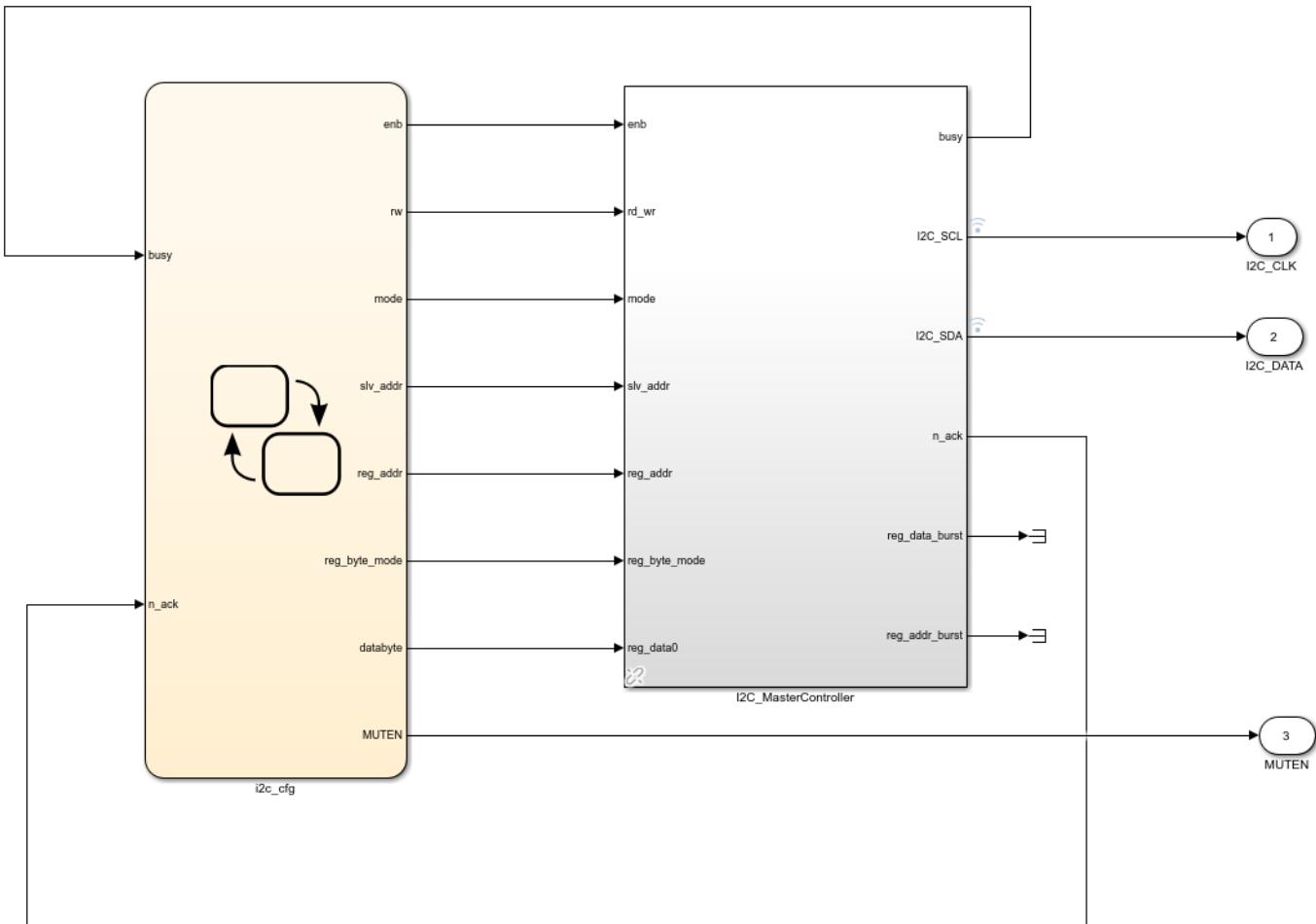
- 1 Model audio codec SSM2603 device configuration chart using Stateflow blocks in Simulink.
- 2 Use I2C Master controller library block to configure audio codec SSM2603.
- 3 Perform simulation of created model.

To configure the audio codec SSM2603 on Zybo board, device configuration chart for SSM2603 need to be created. This chart should be connected to the I2C Master controller library block created earlier.

Note: The device configuration chart used for SSM2603 is specific to this device and can't be used to configure other devices.

The configuration model created for SSM2603 is as shown below.

```
modelname = 'hdlcoder_I2C_ssm2603';
open_system(modelname);
```

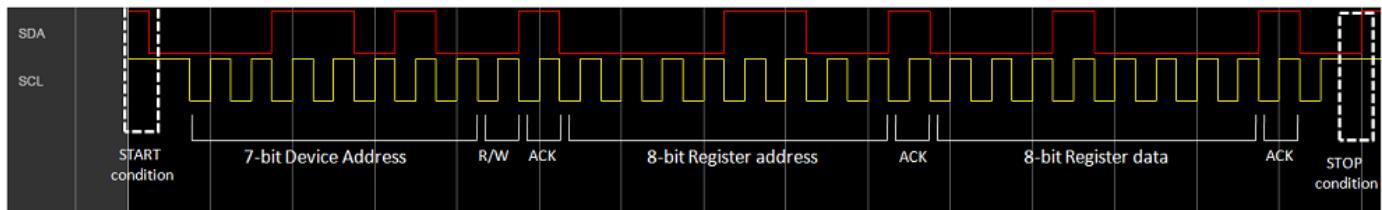


4.1 Simulating audio codec SSM2603 configuration model

For Audio codec chip SSM2603, 11 registers need to be configured. All the to be written in byte mode.

Simulation waveform for configuration of audio codec SSM2603 is as shown below.

Below simulation shows sending of start bit, followed by 7-bit address of the slave device(0x1A), followed by write(0) bit, followed by 8-bit register address(0x0C), followed by 8-bit register data(0x10) and acknowledgments from the slave device.



4.2 IP Core generation workflow

IP Core generation steps for SSM2603 configuration model are same as the steps mentioned above in section 3.2, IP Core generation workflow. Generated IP core can be used in user reference designs. For creating the reference design, refer to “Authoring a Reference Design for Audio System on a ZYBO Board” on page 41-180.

5 Configuring audio codec SSM2603 on Arrow SoC Development Kit using I2C Master Controller library block

This section shows how to:

- 1** Model audio codec SSM2603 device configuration chart using Stateflow blocks in simulink.
- 2** Use I2C Master controller library block to configure audio codec SSM2603.
- 3** Perform simulation of created model.

To configure the audio codec SSM2603 on Arrow SoC Development Kit, device configuration chart for SSM2603 need to be created. This chart should be connected to the I2C Master controller library block created earlier.

Note: The device configuration chart used for SSM2603 is specific to this device and can't be used to configure other devices.

The configuration model for SSM2603 on Arrow SoC Development Kit is same as configuration model for SSM2603 on Zybo board. Please refer to section 4 of this article for SSM2603 configuration model.

5.1 Simulating audio codec SSM2603 configuration model

The audio codec chip SSM2603 configuration on Arrow SoC Development Kit is same as audio codec chip SSM2603 configuration on Zybo board. Please refer to section 4.1 of this article for simulation.

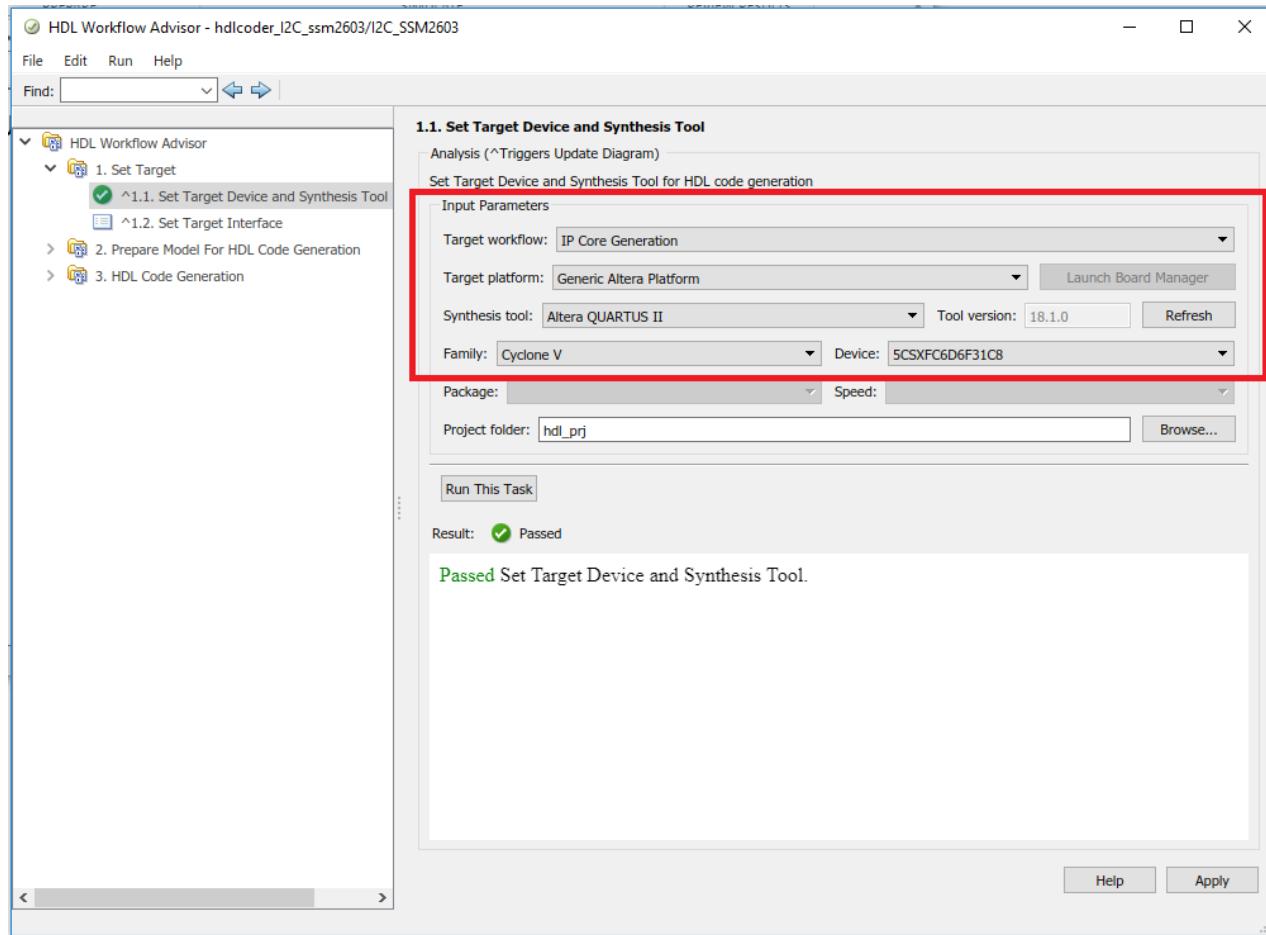
5.2 IP Core generation workflow

To generate the audio codec SSM2603 configuration HDL IP core, follow the steps given below.

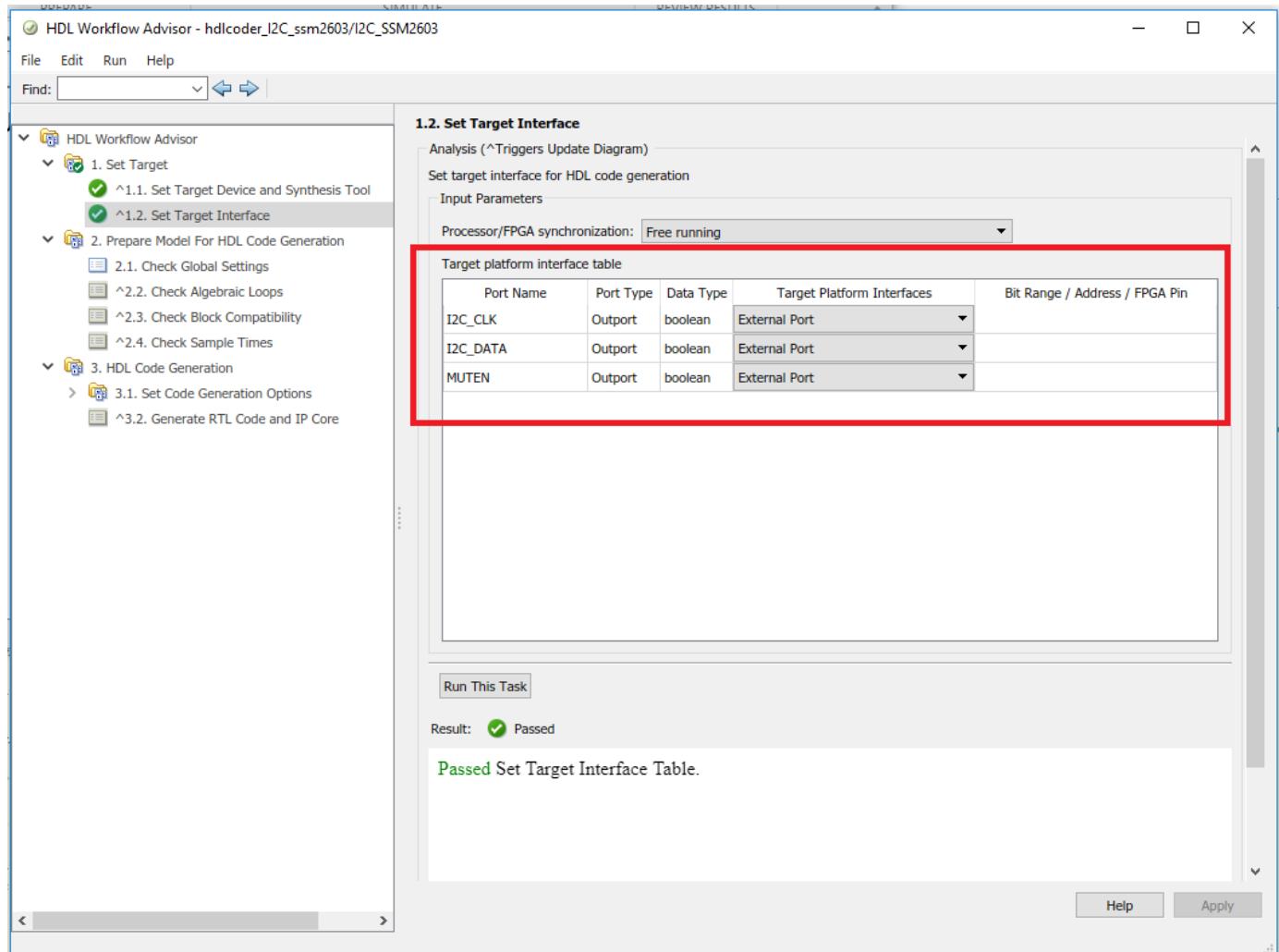
1. Set up the Intel Quartus tool path using the following command in the MATLAB command window. Use your own Quartus installation path when you run the command

```
hdlsetupoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\q
```

2. In the SSM2603 configuration model, select I2C_SSM2603 subsystem and by right clicking open HDL workflow advisor. In Task 1.1, Select **IP Core Generation** for Target workflow, **Generic Altera Platform** for Target platform and **Altera QUARTUS II** for Synthesis Tool. Also select family, device, package and speed as shown in the figure below.



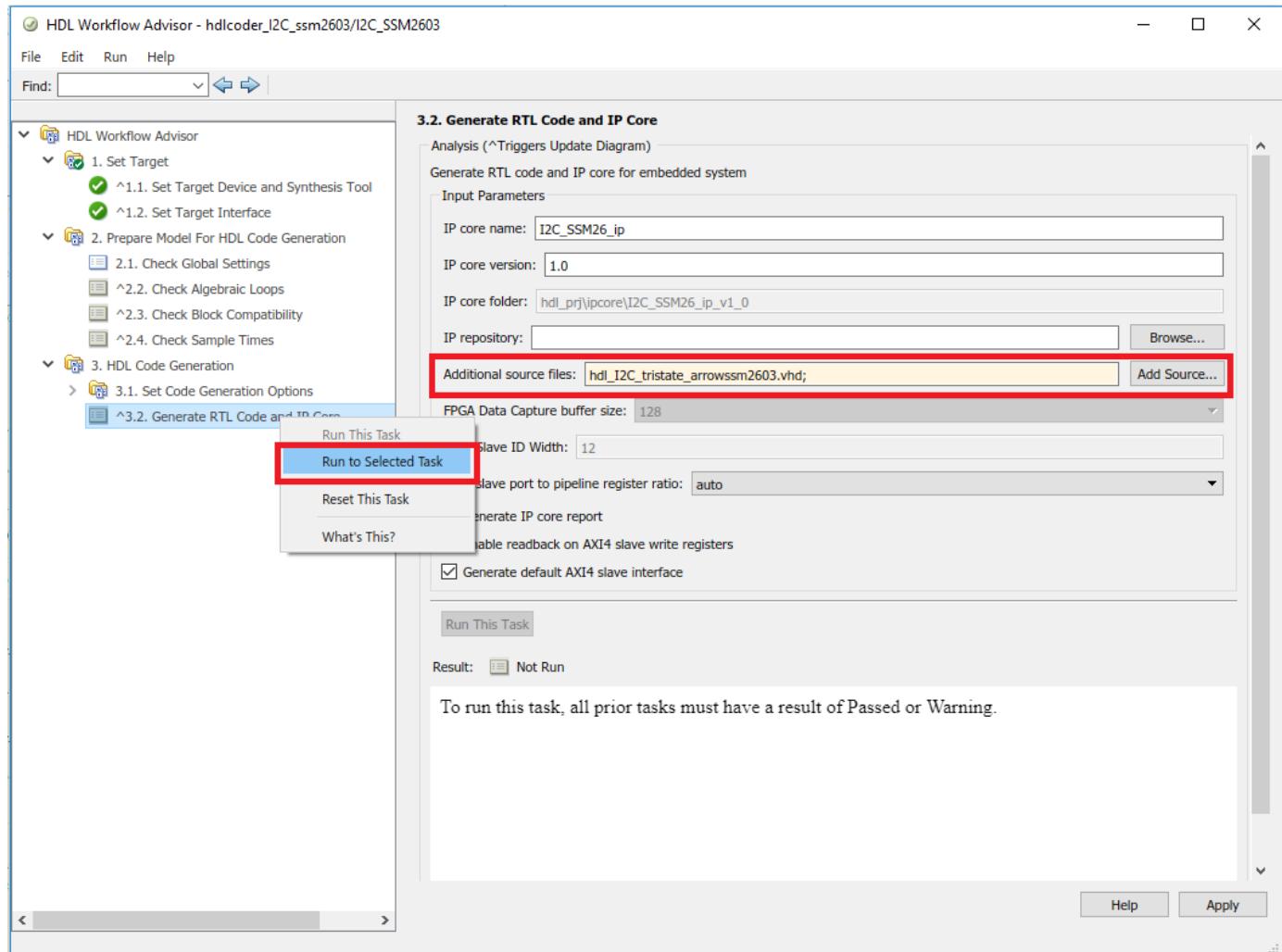
3. In Task 1.2, set the Target Platform Interfaces to "External Port" for all the ports.



4. The Tristate HDL logic is present in the Verilog file hdlcoder_I2C_tristate_arrowssm2603.vhd. Copy it into your current working directory.

```
copyfile(fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdlcoderdemos',
'hdl_I2C_tristate_arrowssm2603.vhd'), 'hdl_I2C_tristate_arrowssm2603.vhd');
```

5. In Task 3.2, add the tristate buffer Verilog file in the additional source files. Then right click on Generate RTL code and IP Core and click on Run to Selected Task.



I2C IP Core for configuration of SSM2603 will be generated. Below figure shows the IP Core generation report.

Code Generation Report

Find: Match Case

Contents

- Summary
- Clock Summary
- Code Interface Report
- Timing And Area Report
 - High-level Resource Report
 - Critical Path Estimation
- Optimization Report
 - Distributed Pipelining
 - Streaming and Sharing
 - Delay Balancing
 - Adaptive Pipelining
- IP Core Generation Report**
- Traceability Report

IP Core Generation Report for hdlcoder_I2C_ssm2603

Summary

IP core name	I2C_SSM26_ip
IP core version	1.0
IP core folder	hdl_prj\ipcore\I2C_SSM26_ip_v1_0
Target platform	Generic Altera Platform
Target tool	Altera QUARTUS II
Target language	VHDL
Model	hdlcoder_I2C_ssm2603
Model version	1.477
HDL Coder version	3.16
IP core generated on	14-Jan-2020 11:40:49
IP core generated for	I2C_SSM2603

Generated Source Files

- [I2C_SSM26_ip_src_I2C_SSM2603_pkg.vhd](#)
- [I2C_SSM26_ip_src_I2c_mstr.vhd](#)
- [I2C_SSM26_ip_src_I2C_MasterController.vhd](#)
- [I2C_SSM26_ip_src_I2c_cfg.vhd](#)
- [I2C_SSM26_ip_src_I2C_SSM2603_tc.vhd](#)
- [I2C_SSM26_ip_src_I2C_SSM2603.vhd](#)

Referenced Models

Target Interface Configuration

You chose the following target interface configuration for [hdlcoder_I2C_ssm2603](#) :

Processor/FPGA synchronization mode: **Free running**

Target platform interface table:

Port Name	Port Type	Data Type	Target Platform Interfaces	Bit Range / Address / FPGA Pin
I2C_CLK	Outport	boolean	External Port	
I2C_DATA	Outport	boolean	External Port	
MUTEN	Outport	boolean	External Port	

Register Address Mapping

The following AXI4 bus accessible registers were generated for this IP core:

Register Name	Address Offset	Description
IPCore_Reset	0x0	write 0x1 to bit 0 to reset IP core
IPCore_Enable	0x4	enabled (by default) when bit 0 is 0x1
IPCore_Timestamp	0x8	contains unique IP timestamp (yymmddHHMM): 2001141140

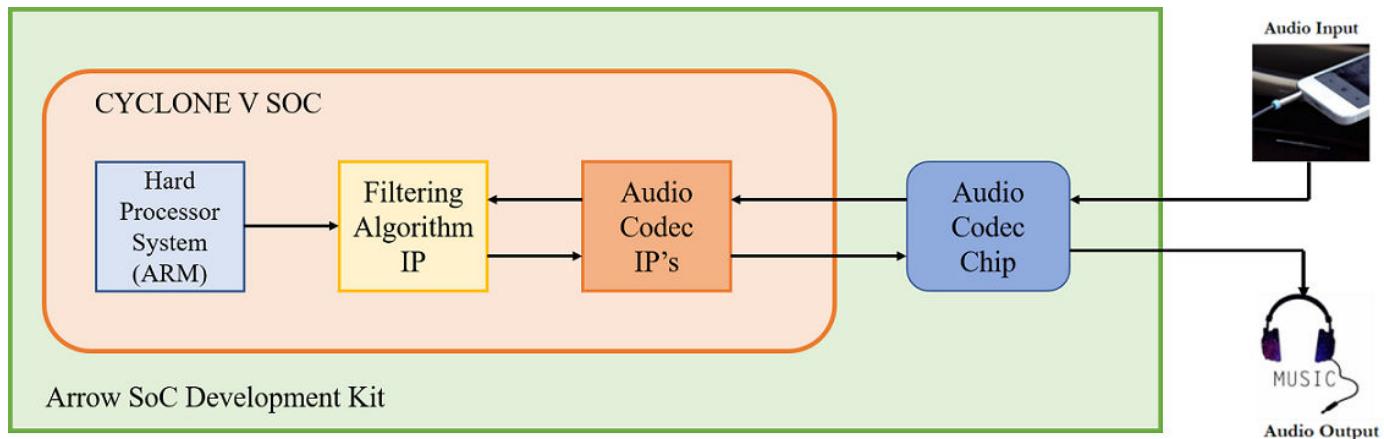
OK **Help**

Generated IP core can be used in user reference designs. For creating the reference design, refer to “Authoring a Reference Design for Audio System on Intel board” on page 41-186.

Running an Audio Filter on Live Audio Input using Intel Board

In this example, we illustrate how to:

- 1 Model an audio system with Low pass and Band pass filters
- 2 Implement it on a Intel board using an audio reference design



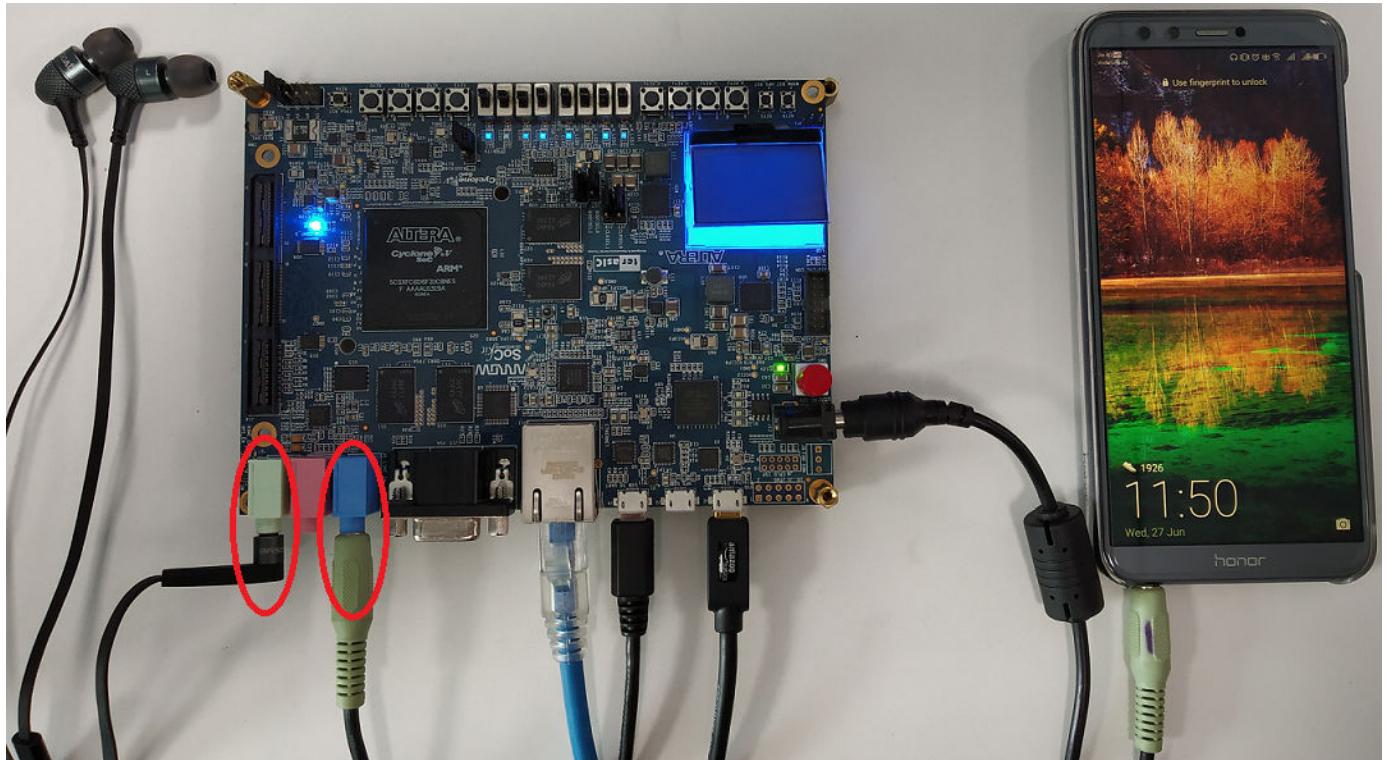
The objective of this example is to receive audio input through Arrow SoC Development Kit's line input, process it on the FPGA and transmit the processed audio to a speaker. The above figure shows the high-level architecture of such a system. It uses an audio codec to interface to the peripherals and to convert analog to digital signals and vice-versa. The Audio Codec IPs are used to configure the audio codec and for transferring audio data between Intel SoC and audio codec. The Filter IP is used for audio processing. ARM processor is used to control the type of filter to be used i.e. low pass or band pass.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder Support Package for Intel SoC Devices
- Embedded Coder Support Package for Intel SoC Devices
- Intel Quartus Prime Standard, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Intel SoC Embedded Design Suite
- Arrow SoC Development Kit

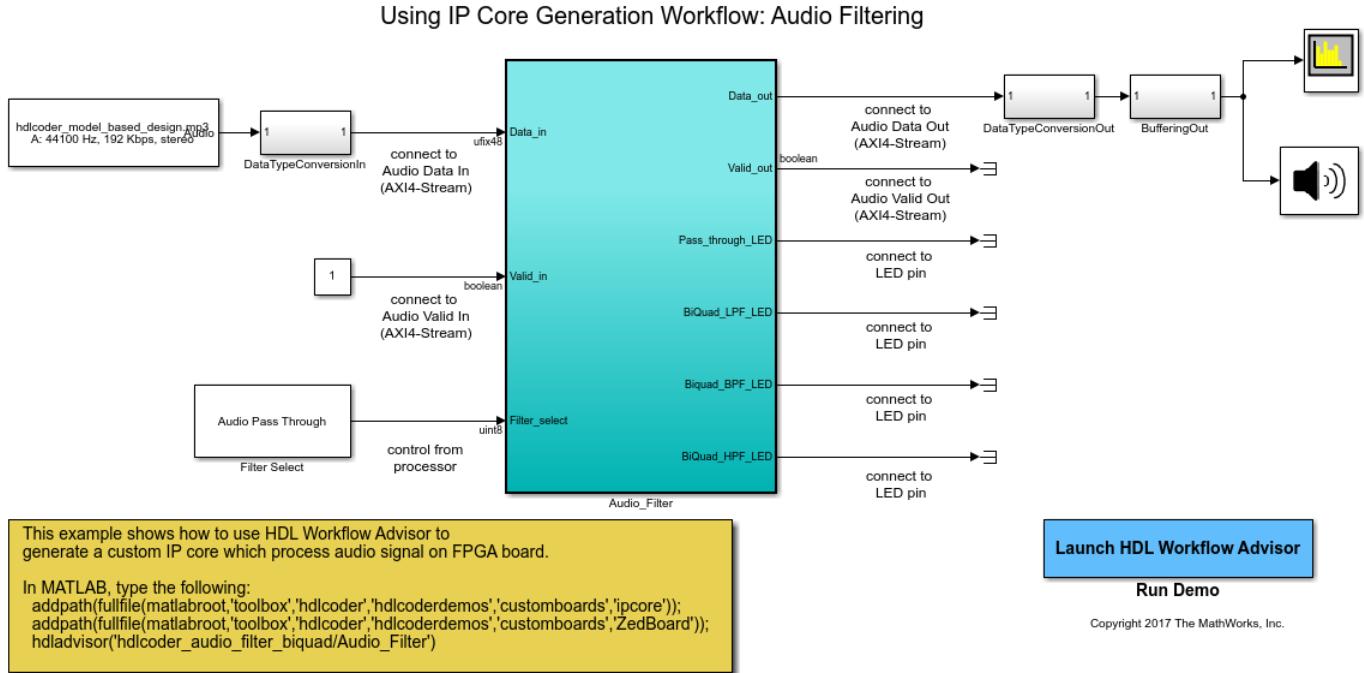
To setup the Arrow SoC Development Kit, refer to the “Set up Intel SoC hardware and tools” section in the “Hardware-Software Co-Design Workflow for SoC Platforms” on page 40-2 example. Connect an audio input from a mobile or an MP3 player to **LINE IN** jack and either earphones or speakers to **LINE OUT** jack on the Arrow SoC as shown below.



Introduction

In the following model, an audio file is used as input to the DUT subsystem, **Audio filter**. On simulating this model in Simulink, the processed audio effect can be heard through the **Audio Device Writer** block and **Spectrum Analyzer** block displays the spectrogram of the filtered audio output.

```
modelname = 'hdlcoder_audio_filter_biquad';
open_system(modelname);
```



Model a System with Low pass and Band pass Filters

Filter coefficients may be generated using a matlab function or in Simulink. In this model, **filterDesigner** (DSP System Toolbox) tool is used to generate the filter coefficients for each type of filter. Then these filter coefficients are exported and stored as a matlab file. These coefficients will be used to design the filters in Simulink. In this model, discrete IIR filter blocks from Simulink® are used as Biquad low pass or band pass filters depending on the corresponding filter coefficients.

You can test this model by simulating the model in Simulink. The range of frequencies seen on the Spectrum Analyzer and the audio effect heard through the Audio Device Writer block should vary depending on the type of filter selected. **Filter Select** block is used to select the type of filtering to be done on the audio input.

Customize the Model for Arrow SoC Development Kit

In order to implement this model on Arrow SoC, you must first create a reference design in Qsys which receives audio input on Arrow SoC and transmits the processed audio data out of Arrow SoC. For details on how to create a reference design which integrates the audio filter model, refer to “Authoring a Reference Design for Audio System on Intel board” on page 41-186 example.

In the reference design, left and right channel audio data are combined to form a single channel. They are concatenated such that lower 24 bits is the left channel and upper 24 bits is the right channel. In the Simulink model shown above, Data_in is split into 2 channels i.e. left and right accordingly. Their magnitude is divided by 2 and the 2 channels are added together to form a single channel. Filtering is done on this channel.

Data_in and **Valid_in** are the AXI4-Stream signals. **Data_in** contains the audio data to be processed and **Valid_in** acts as the enable signal. Each filter is mapped to an LED on Arrow SoC to visually indicate whether the filter is on or off.

FilterSelect input is controlled via AXI4 interface.

Generate HDL IP core with AXI4-Stream Interface

Next, you can start the HDL Workflow Advisor and use the Intel® hardware-software co-design workflow to deploy this design on the Intel hardware. For a more detailed step-by-step guide, you can refer to the “Getting Started with Targeting Intel SoC Devices” on page 40-104 example.

1. Set up the Intel Quartus synthesis tool path using the following command in the MATLAB command window. Use your own Intel Quartus installation path when you run the command. For example:

```
hdlsetupoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\18.1\quartus\bin64\q')


```

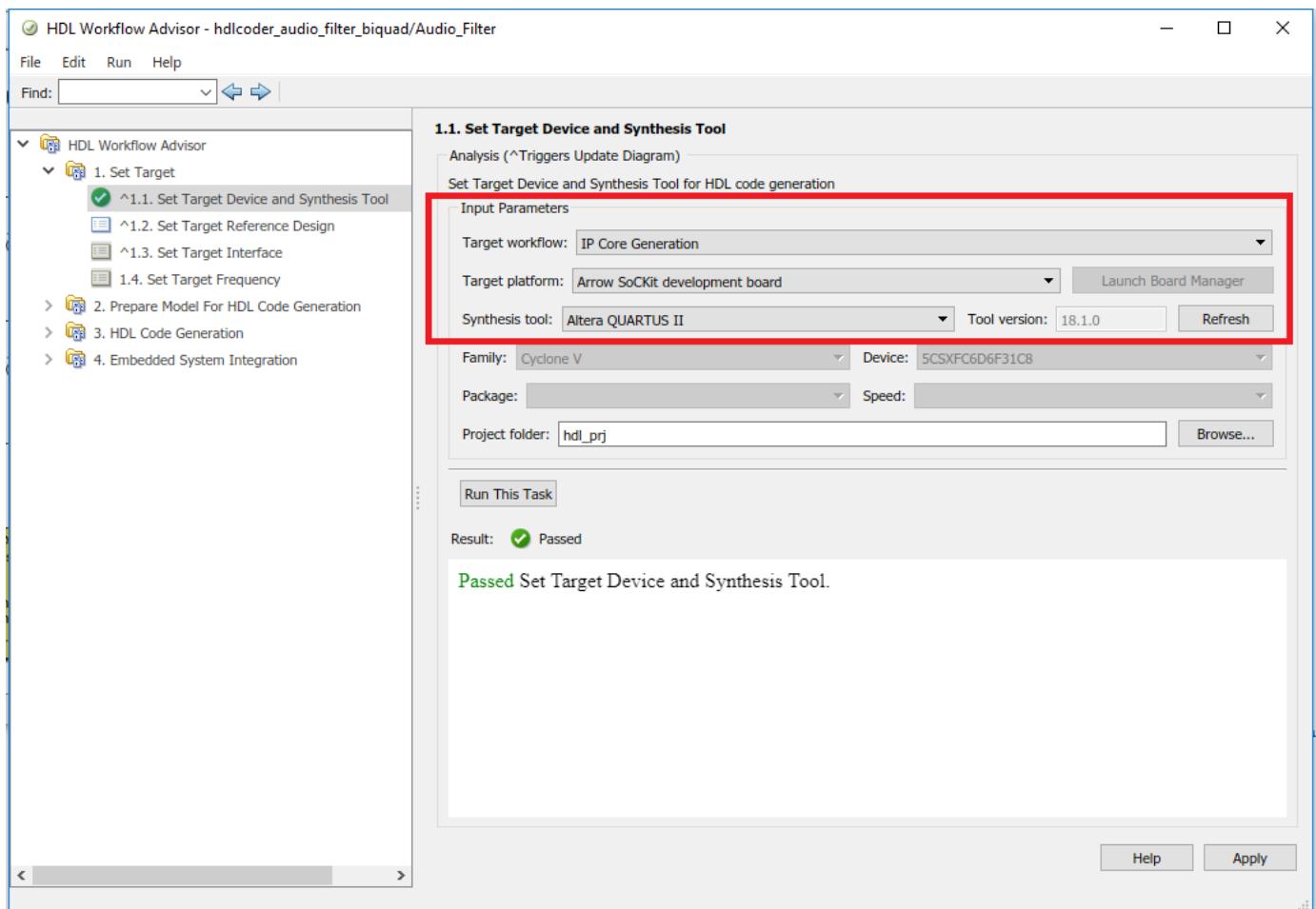
2. Add both the IP repository folder and the Arrow SoC Development Kit registration file to the MATLAB path using following commands:

```
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ipcore'));
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ArrowSoC'));

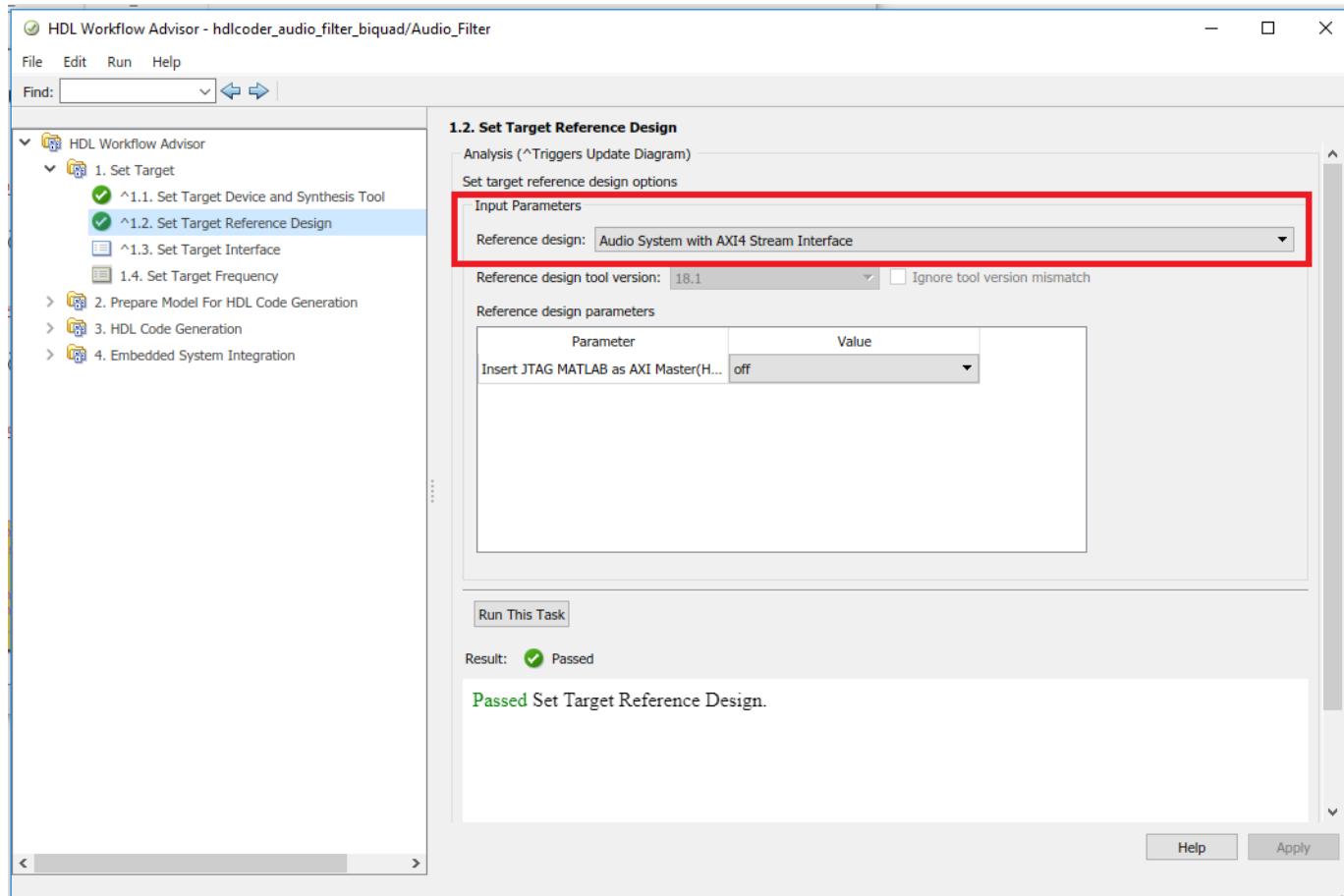

```

3. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_audio_filter_biquad`/`Audio_filter` or by double-clicking the Launch HDL Workflow Advisor box in the model.

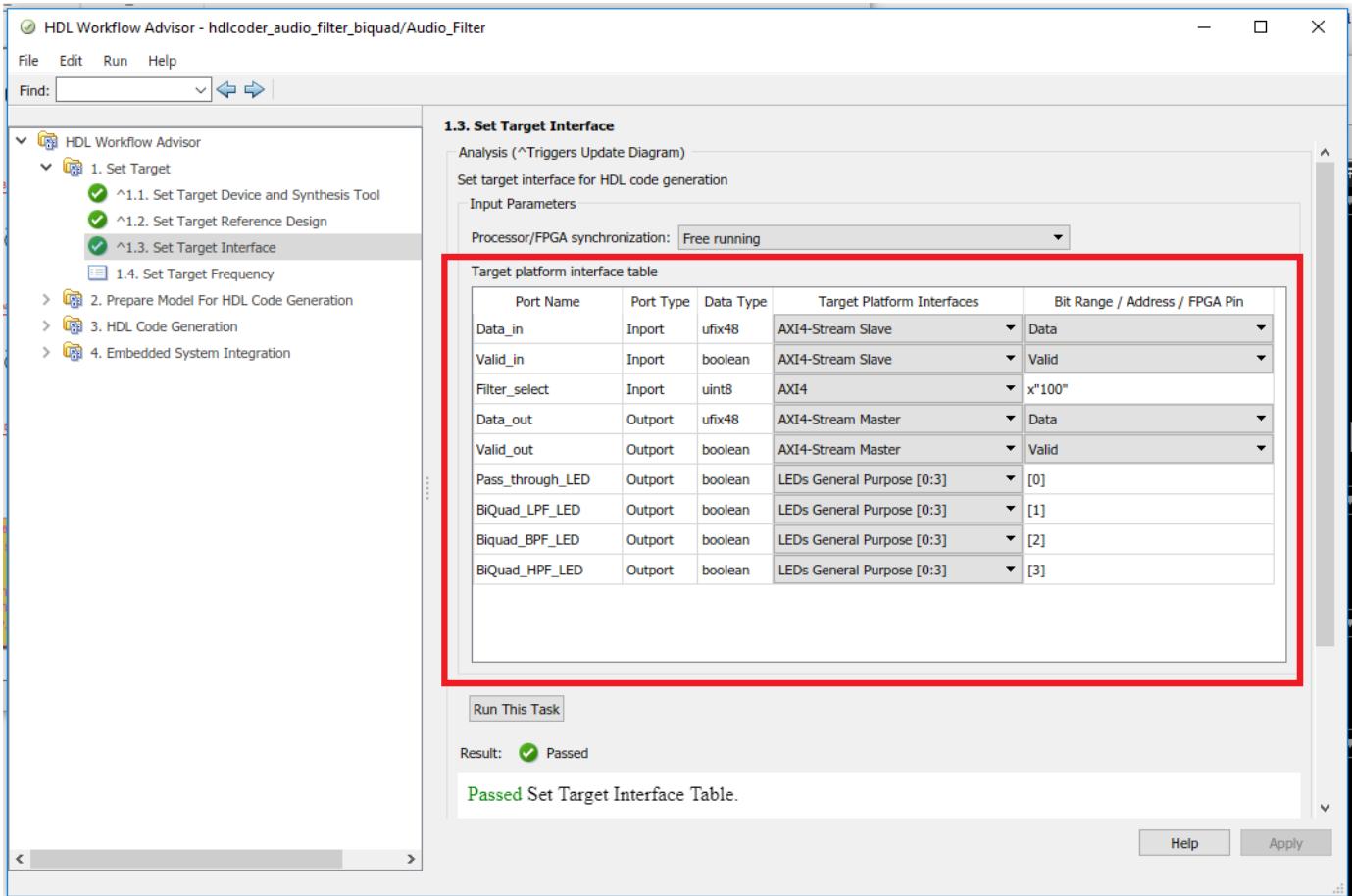
In Task 1.1, select **IP Core Generation** for **Target workflow**, and select **Arrow SoC Development Kit** for **Target platform**.



In Task 1.2, Audio System with AXI4 Stream Interface is selected for **Reference Design**.



The AXI4-Stream interface is used for transferring audio data between the reference design and the filtering algorithm IP. The AXI4-Stream interface contains data (**Data**) and control signals such as data valid (**Valid**), back pressure (**Ready**), and data boundary (**TLAST**). At least **Data** and **Valid** signals are required for AXI4-Stream IP core generation. In Task 1.3, the **Target platform interface table** is loaded as shown in the following picture. The audio data stream ports, **Valid_in**, **Data_in**, **Valid_out** and **Data_out**, are mapped to the AXI4-Stream interfaces, **Pass_through_LED**, **BiQuad_LPF_LED**, **BiQuad_BPF_LED** are mapped to the LEDs on Arrow SoC and the control parameter port **Filter_select** is mapped to the AXI4 interface.



The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Master** interface.

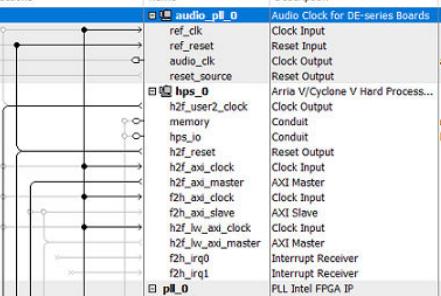
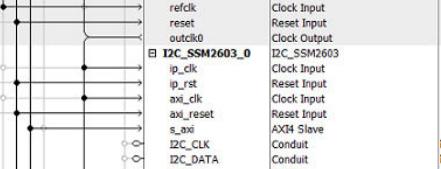
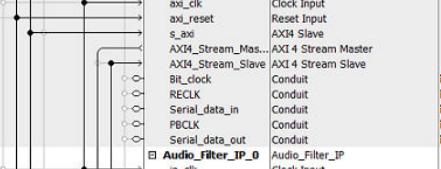
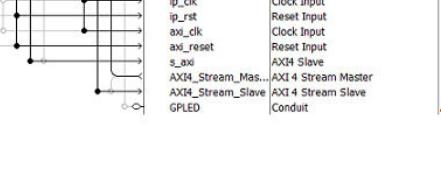
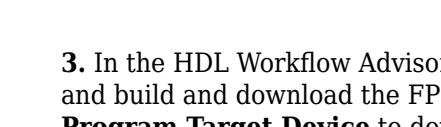
3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP Into AXI4-Stream Audio Compatible Reference Design

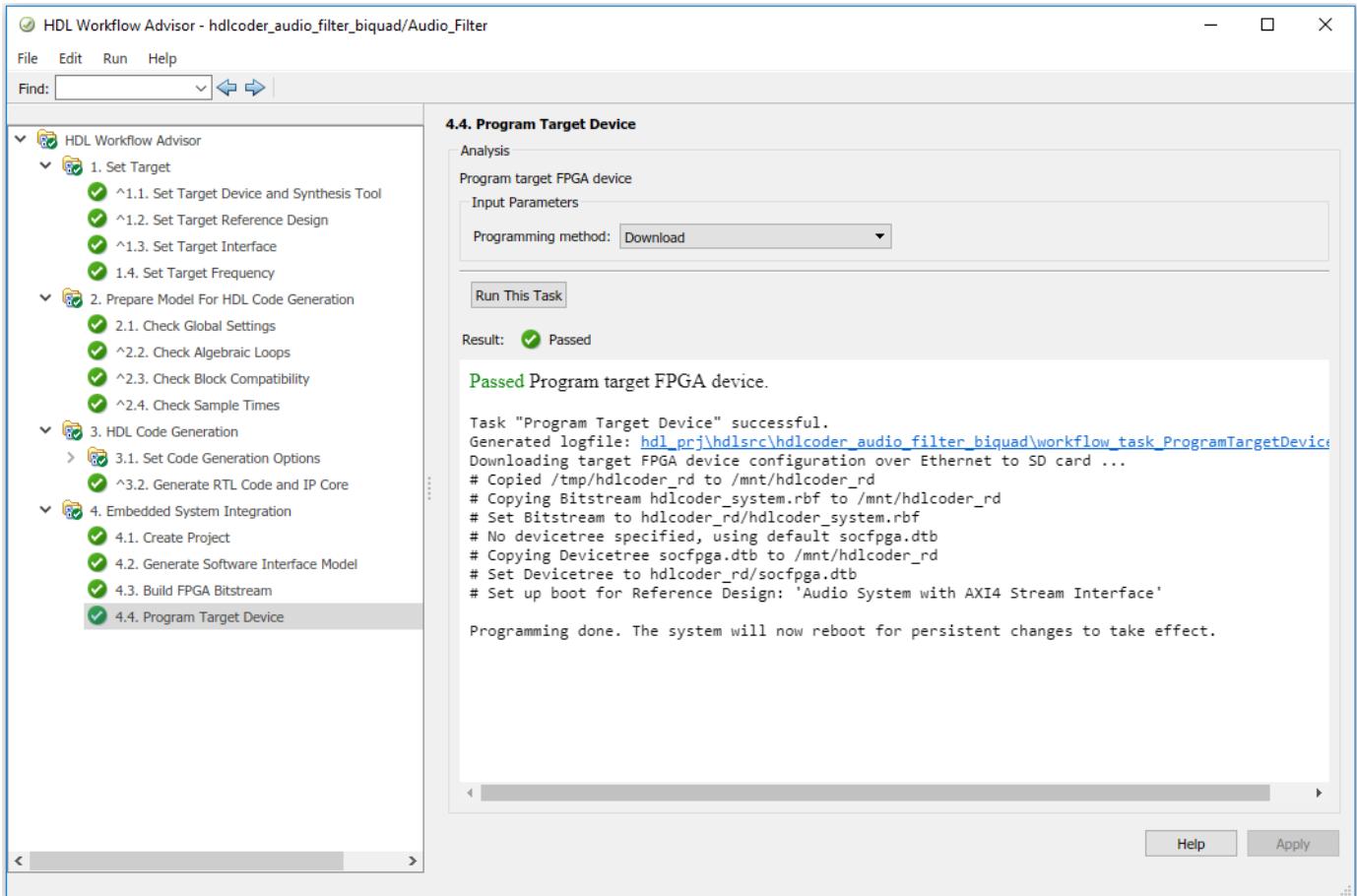
Next, in the HDL Workflow Advisor, you run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Intel SoC.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Audio System with AXI4 Stream Interface** reference design. As shown in the first diagram, this reference design contains the IPs to handle audio data in and out of Arrow SOC. The generated project is a complete design, including the algorithm part (the generated DUT algorithm IP), and the platform part (the reference design). For details on how to create a reference design which integrates the audio filter model, refer to "Authoring a Reference Design for Audio System on Intel board" on page 41-186 example.

2. Click on the "Generated Altera Qsys project" link in the Result pane to open the generated platform designer Qsys project.

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
<input checked="" type="checkbox"/>	 A detailed block diagram showing the connections for the audio_pll_0 component. It includes various clock inputs (ref_clk, ref_reset, audio_clk, reset_source) and outputs (audio_pll_0_audio_clk, hps_0_h2f_user2_clock, hps_0_hps_ip, pll_0_outclk0, hps_0_n2f_user2_clock, pll_0_outclk0, i2c_ssm2603_0_i2c_clk, i2c_ssm2603_0_i2c_data, i2s_ssm2603_0_i2c_clk, i2s_ssm2603_0_i2c_data, and Audio_Filter_IP_0_ip_clk). There are also connections for memory, AXI Master/Slave, interrupt receivers, and other peripherals.	audio_pll_0	Audio Clock for DE-series Boards	<i>Double-click to export</i> ref_clk ref_reset audio_clk reset_source	pll_0_outclk0 audio_pll_0_audio_clk <i>Double-click to export</i>					
<input checked="" type="checkbox"/>	 A detailed block diagram showing the connections for the hps_0 component. It includes various clock inputs (h2f_user2_clock, hps_reset, h2f_axi_clock, h2f_axi_master, f2h_axi_clock, f2h_axi_slave, h2f_lv_axi_clock, h2f_lv_axi_master) and outputs (hps_0_h2f_user2_clock, hps_0_hps_ip, pll_0_outclk0, hps_0_n2f_user2_clock, and i2c_ssm2603_0_i2c_clk). There are also connections for memory, AXI Master/Slave, interrupt receivers, and other peripherals.	hps_0	Arria V/Cyclone V Hard Processor...	<i>Double-click to export</i> memory hps_0_hps_ip	hps_0_h2f_user2_clock					
<input checked="" type="checkbox"/>	 A detailed block diagram showing the connections for the pll_0 component. It includes various clock inputs (refclk, reset, outclk0) and outputs (pll_0_outclk0, hps_0_n2f_user2_clock, and i2c_ssm2603_0_i2c_clk). There are also connections for memory, AXI Master/Slave, interrupt receivers, and other peripherals.	pll_0	PLL Intel FPGA IP	<i>Double-click to export</i> refclk reset outclk0	pll_0_outclk0			IRQ 0	IRQ 31	
<input checked="" type="checkbox"/>	 A detailed block diagram showing the connections for the i2c_ssm2603_0 component. It includes various clock inputs (ip_clk, ip_rst, axi_clk, axi_reset, s_axi) and outputs (i2c_ssm2603_0_i2c_clk, i2c_ssm2603_0_i2c_data, and i2c_ssm2603_0_muten). There are also connections for AXI Slave, Conduit, and MUTEN.	i2c_ssm2603_0	I2C_SSM2603	<i>Double-click to export</i> ip_clk ip_rst axi_clk axi_reset s_axi	pll_0_outclk0 pll_0_outclk0			0x0001_0000	0x0001_ffff	
<input checked="" type="checkbox"/>	 A detailed block diagram showing the connections for the i2s_ssm2603_0 component. It includes various clock inputs (ip_clk, ip_rst, axi_clk, axi_reset, s_axi) and outputs (i2s_ssm2603_0_bit_clock, i2s_ssm2603_0_reck, i2s_ssm2603_0_serial_data_in, i2s_ssm2603_0_pclk, and i2s_ssm2603_0_serial_data_out). There are also connections for AXI Slave, Conduit, and RECLK.	i2s_ssm2603_0	I2S_SSM2603	<i>Double-click to export</i> ip_clk ip_rst axi_clk axi_reset s_axi	pll_0_outclk0 pll_0_outclk0			0x0002_0000	0x0002_ffff	
<input checked="" type="checkbox"/>	 A detailed block diagram showing the connections for the Audio_Filter_IP_0 component. It includes various clock inputs (ip_clk, ip_rst, axi_clk, axi_reset, s_axi) and outputs (Audio_Filter_IP_0_ip_clk, and GPLED). There are also connections for AXI Stream Master/Slave, Conduit, and RECLK.	Audio_Filter_IP_0	Audio_Filter_IP	<i>Double-click to export</i> ip_clk ip_rst axi_clk axi_reset s_axi	pll_0_outclk0 pll_0_outclk0			0x0000_0000	0x0000_ffff	

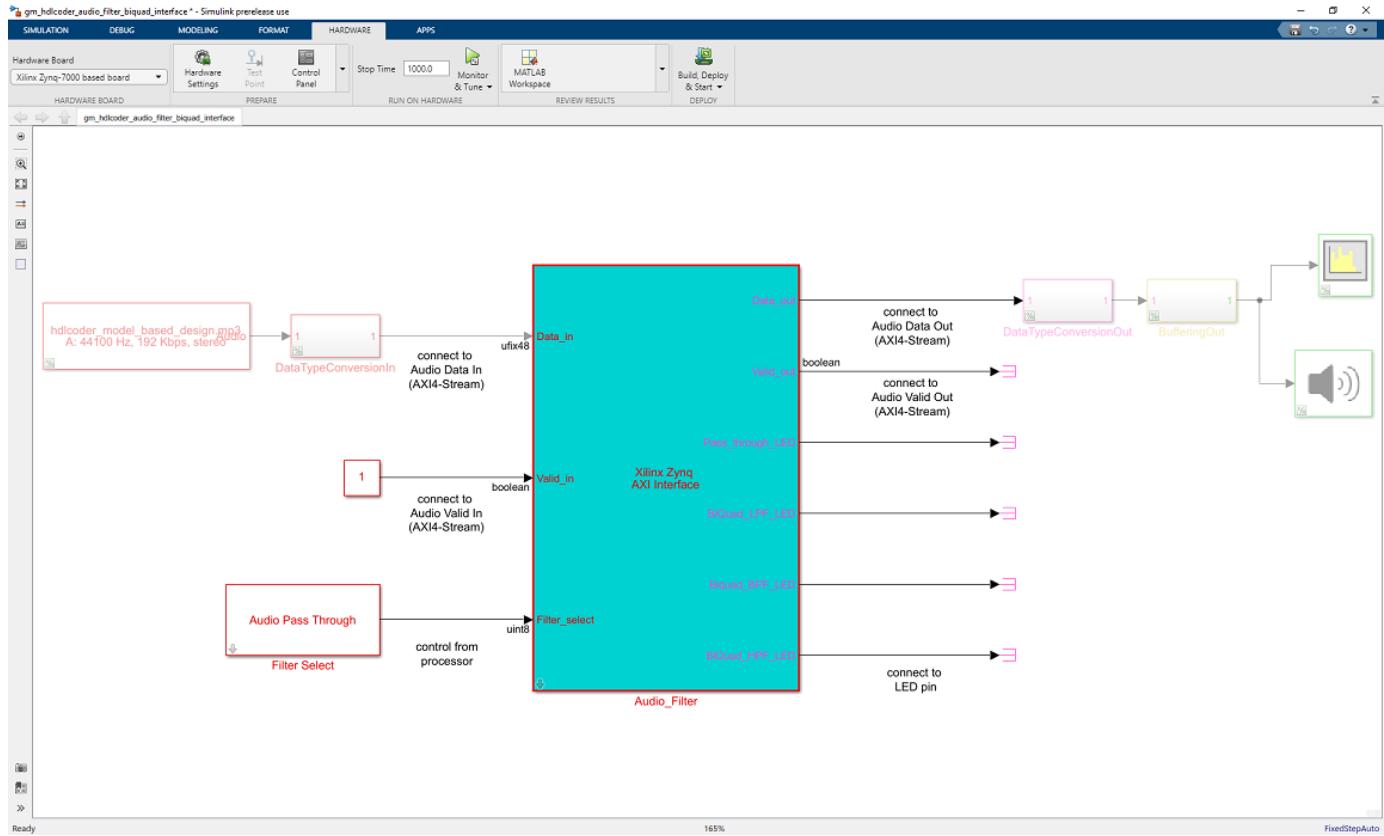
3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream. Choose **Download** programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the Intel board, so your design will be automatically reloaded when you power cycle the Intel board.



Generate ARM executable to Tune Parameters on the FPGA Fabric

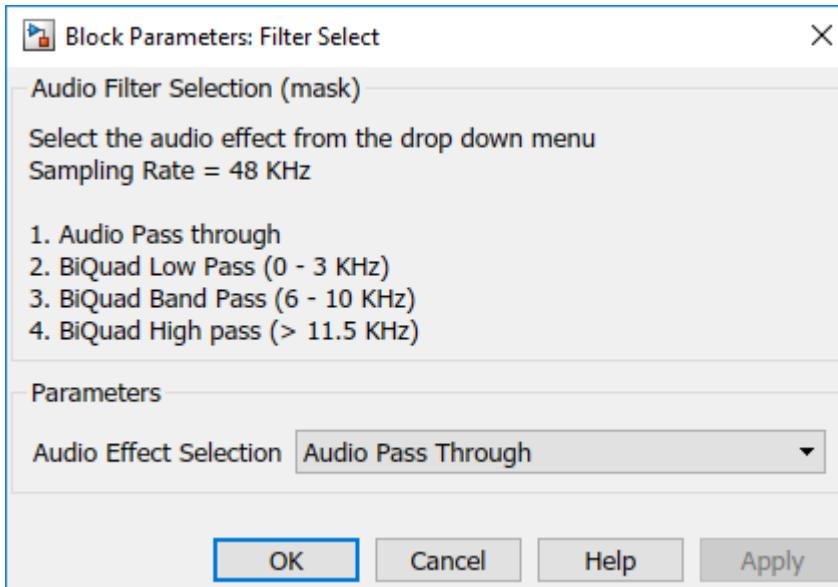
A software interface model is generated in Task 4.2, **Generate Software Interface Model**.

1. Before you generate code from the software interface model, comment out the audio input source and audio output sink i.e From Multimedia File, Data Type Conversion, Buffer, Audio Device Writer and Spectrum Analyzer Blocks. These blocks do not need to be run on the ARM processor. The Audio_filter IP is running as *Filtering_Algorithm" on FPGA fabric. The ARM processor is using AXI4 interface for selecting the filter type i.e. Biquad Low pass, Band pass or Pass Through.



- 1 In the generated model, go to **Hardware** pane and then open the Configuration Parameters dialog box by clicking on **Hardware Settings**.
- 1 Select **Solver** and set "Stop Time" to "inf".
- 2 Then in the **Hardware Pane**, Click on **Monitor & Tune** button. Embedded Coder builds the model, downloads the ARM executable to the Intel board hardware, executes it, and connects the model to the executable running on the Intel board hardware. Then you can tune model parameters.

The type of filter to be used can be selected using the drop down options in **Filter Select** block



The filtered audio output can be heard by plugging earphones or speakers to **LINE OUT** jack on the Arrow SoC. Depending on the filter selected, the corresponding LED on the Arrow SoC turns on. In this example, LD0 turns on when Pass through (No filter used) option is selected, LD1 turns on when Biquad Low pass filter is selected, LD2 turns on when Biquad Band pass filter is selected.

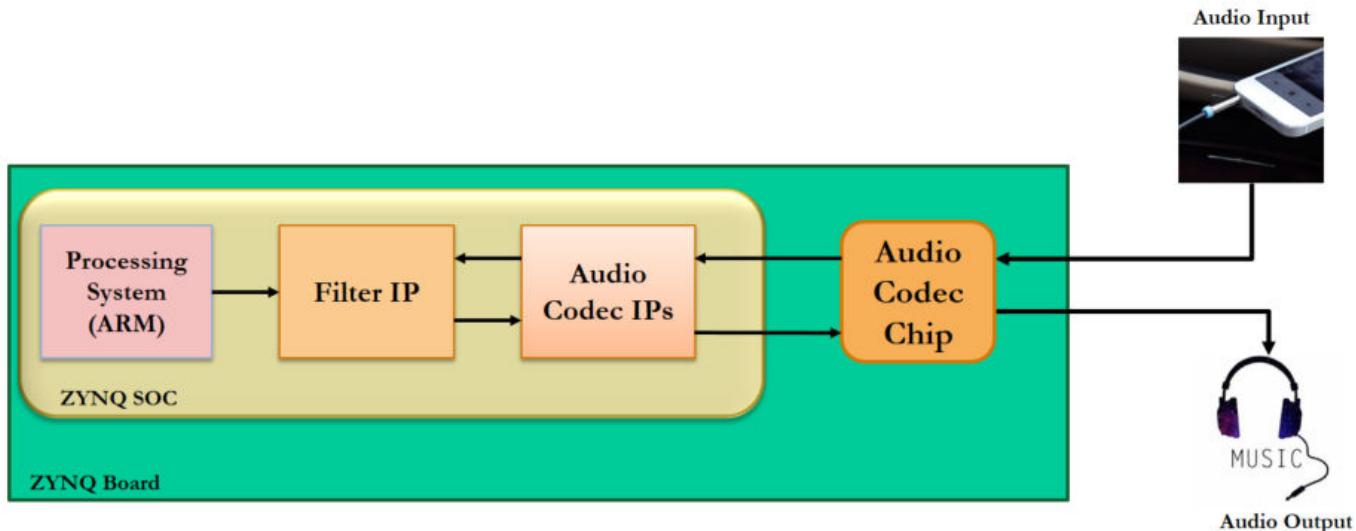
Running an Audio Filter on Live Audio Input Using a Zynq Board

This example shows how to model an audio system and implement it on a Zynq® board using an audio reference design.

Introduction

In this example, you:

- 1 Model an audio system with Low pass, Band pass and High pass filters
- 2 Implement it on a Zynq board using an audio reference design



The objective of this example is to receive audio input through Zedboard or Zybo board's line input, process it on the FPGA and transmit the processed audio to a speaker. The above figure shows the high-level architecture of such a system. It uses an audio codec to interface to the peripherals and to convert analog to digital signals and vice-versa. The Audio Codec IPs are used to configure the audio codec and for transferring audio data between Zynq Soc and audio codec. The Filter IP is used for audio processing. ARM processor is used to control the type of filter to be used i.e. low pass, band pass or high pass.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform
- Xilinx Vivado version 2019.1
- ZedBoard or Zybo Board

To setup the Zedboard board, refer to the *Set up Zynq hardware and tools* section in the “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example. Connect an audio input from a mobile or an MP3 player to **LINE IN** jack and either earphones or speakers to **HPH OUT** jack on the

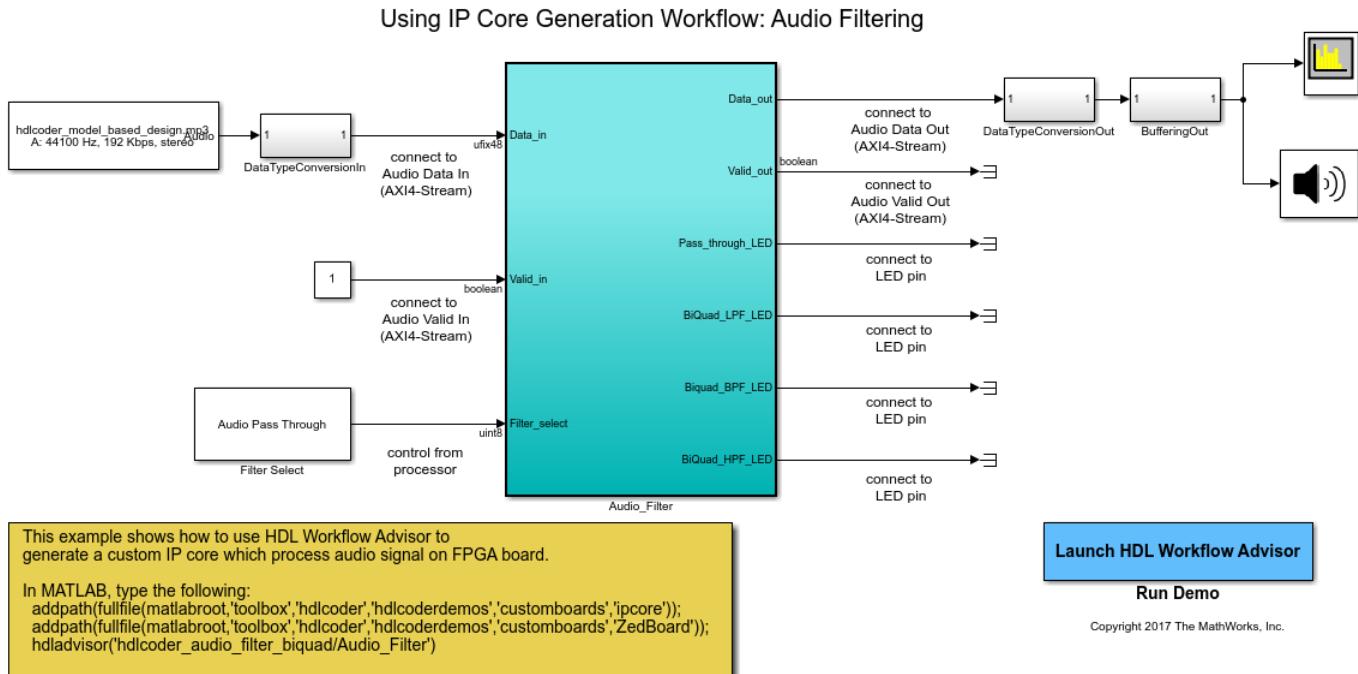
Zedboard as shown below. Similar setup can be done on Zybo board. For Zybo board setup, refer to *Set up the Zybo board* section in the “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196 example.



Introduction

In the following model, an audio file is used as input to the DUT subsystem, **Audio_filter**. On simulating this model in Simulink, the processed audio effect can be heard through the **Audio Device Writer** block and **Spectrum Analyzer** block displays the spectrogram of the filtered audio output.

```
modelname = 'hdlcoder_audio_filter_biquad';
open_system(modelname);
```



Model a system with Low pass, Band pass and High pass filters

Filter coefficients may be generated using a MATLAB® function or in Simulink®. In this model, filterDesigner tool is used to generate the filter coefficients for each type of filter. Then these filter coefficients are exported and stored as a MATLAB file. These coefficients will be used to design the filters in Simulink. In this model, discrete IIR filter blocks from Simulink are used as Biquad low pass, band pass or high pass filters depending on the corresponding filter coefficients.

You can test this model by simulating the model in Simulink. The range of frequencies seen on the Spectrum Analyzer and the audio effect heard through the Audio Device Writer block should vary depending on the type of filter selected. **Filter Select** block is used to select the type of filtering to be done on the audio input.

Customize the model for Zynq board

In order to implement this model on Zedboard, you must first create a reference design in Vivado which receives audio input on Zedboard and transmits the processed audio data out of Zedboard. For details on how to create a reference design which integrates the audio filter model, refer to “Authoring a Reference Design for Audio System on a Zynq Board” on page 41-170 example.

For Zybo board, refer to “Authoring a Reference Design for Audio System on a ZYBO Board” on page 41-180.

In the reference design, left and right channel audio data are combined together to form a single channel. They are concatenated such that lower 24 bits is the left channel and upper 24 bits is the right channel. In the Simulink model shown above, Data_in is split into 2 channels i.e. left and right accordingly. Their magnitude is divided by 2 and the 2 channels are added together to form a single channel. Filtering is done on this channel.

Data_in and **Valid_in** are the AXI4-Stream signals. To understand how AXI4-stream interface is used, refer to *Model Streaming Algorithm with Simplified Streaming Protocol* section in “Getting Started

with AXI4-Stream Interface in Zynq Workflow” on page 41-137 example. **Data_in** contains the audio data to be processed and **Valid_in** acts as the enable signal. Each filter is mapped to an LED on Zedboard or Zybo board to visually indicate whether the filter is on or off.

FilterSelect input is controlled via AXI4 LITE interface.

Generate HDL IP core with AXI4-Stream Interface

Next, you can start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado.bat')
```

2. Add both the IP repository folder and the Zedboard registration file to the MATLAB path using following commands:

```
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ipcore'));
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ZedBoard'));
```

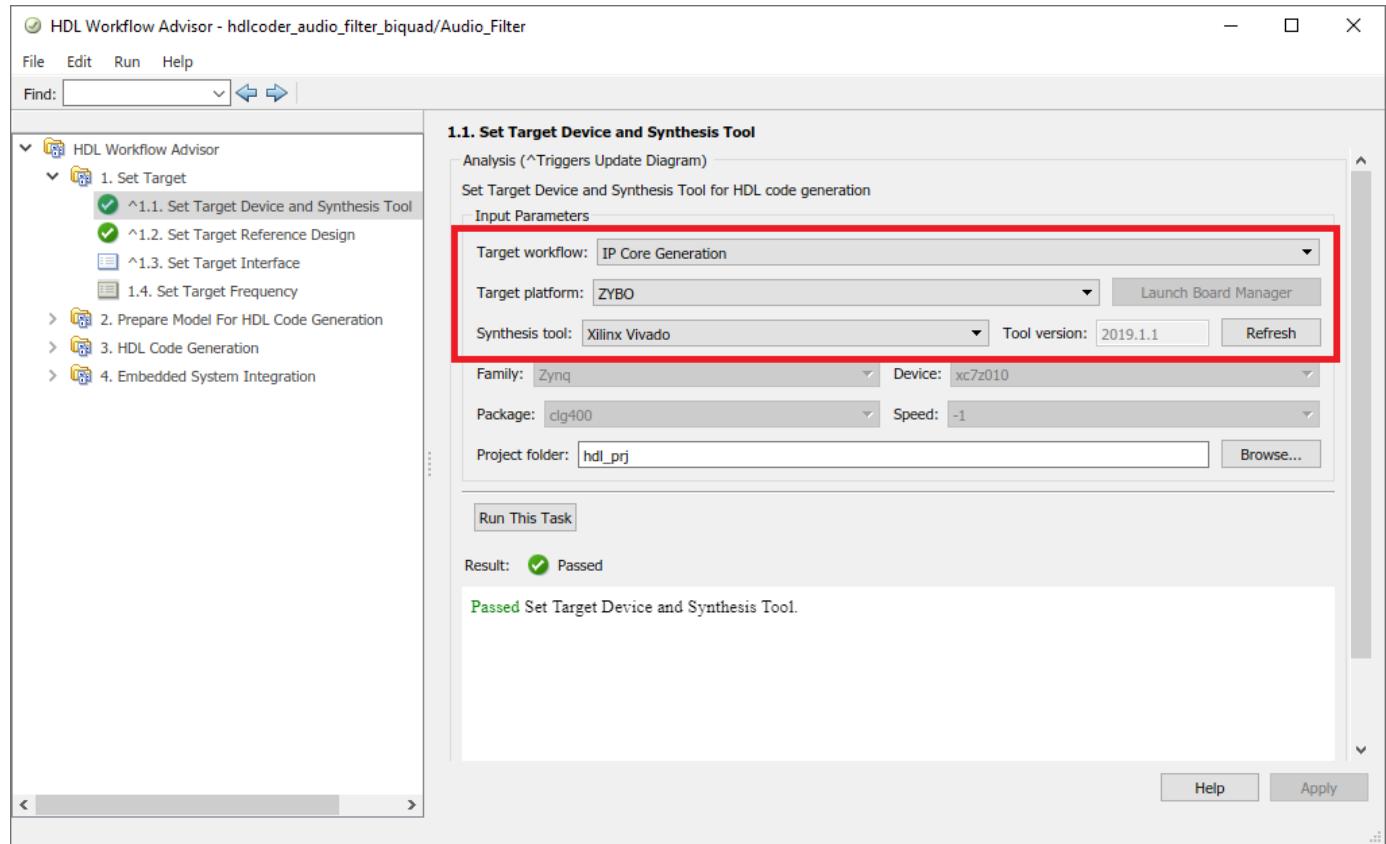
For Zybo board use the following commands.

```
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ipcore'));
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ZYBO'));
```

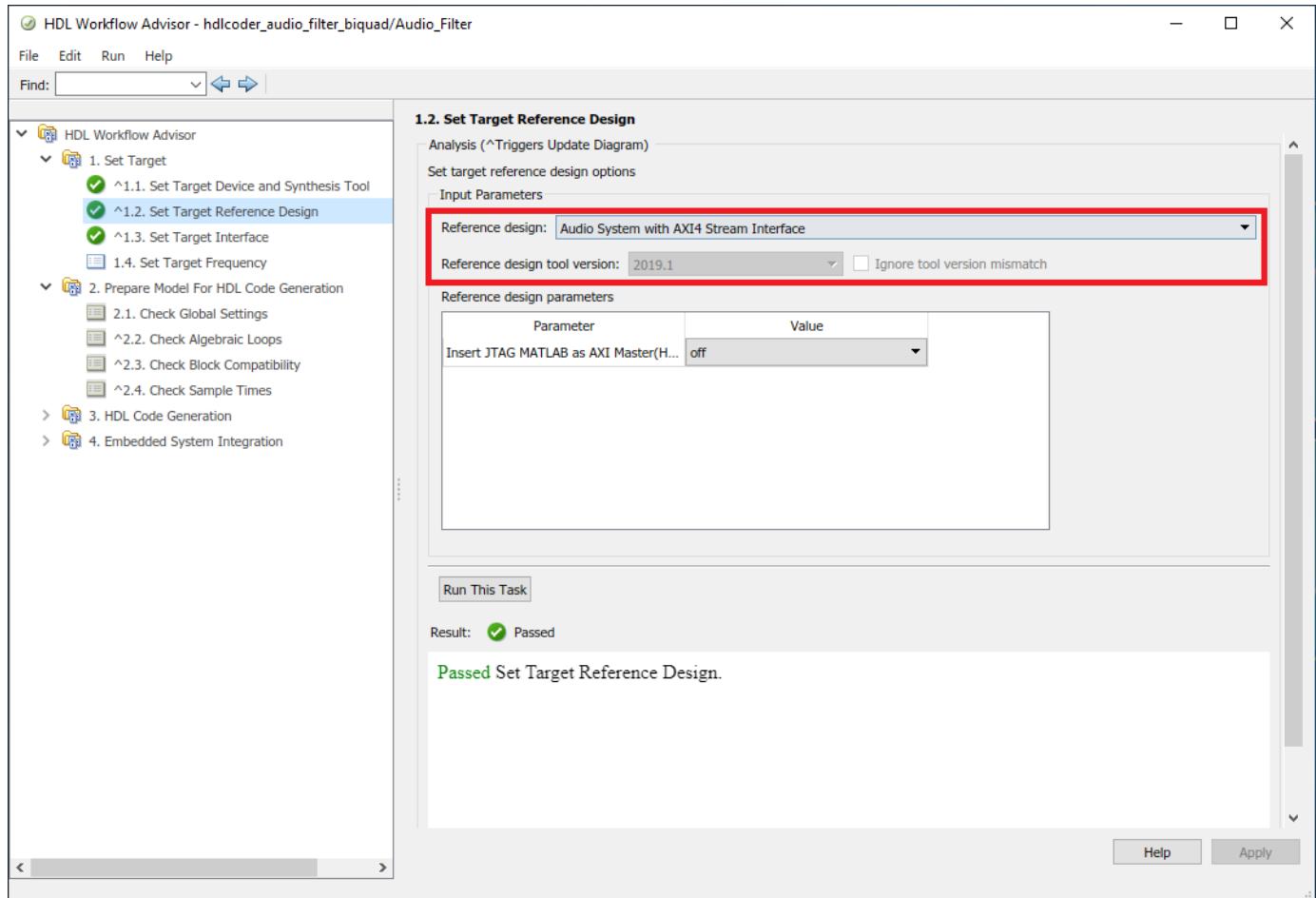
3. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_audio_filter_biquad`/`Audio_filter` or by double clicking on the Launch HDL Workflow Advisor box in the model.

The target interface settings are already saved for Zedboard in this example model, so the settings in Task 1.1 to 1.3 are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the “Save Target Hardware Settings in Model” on page 40-137 example.

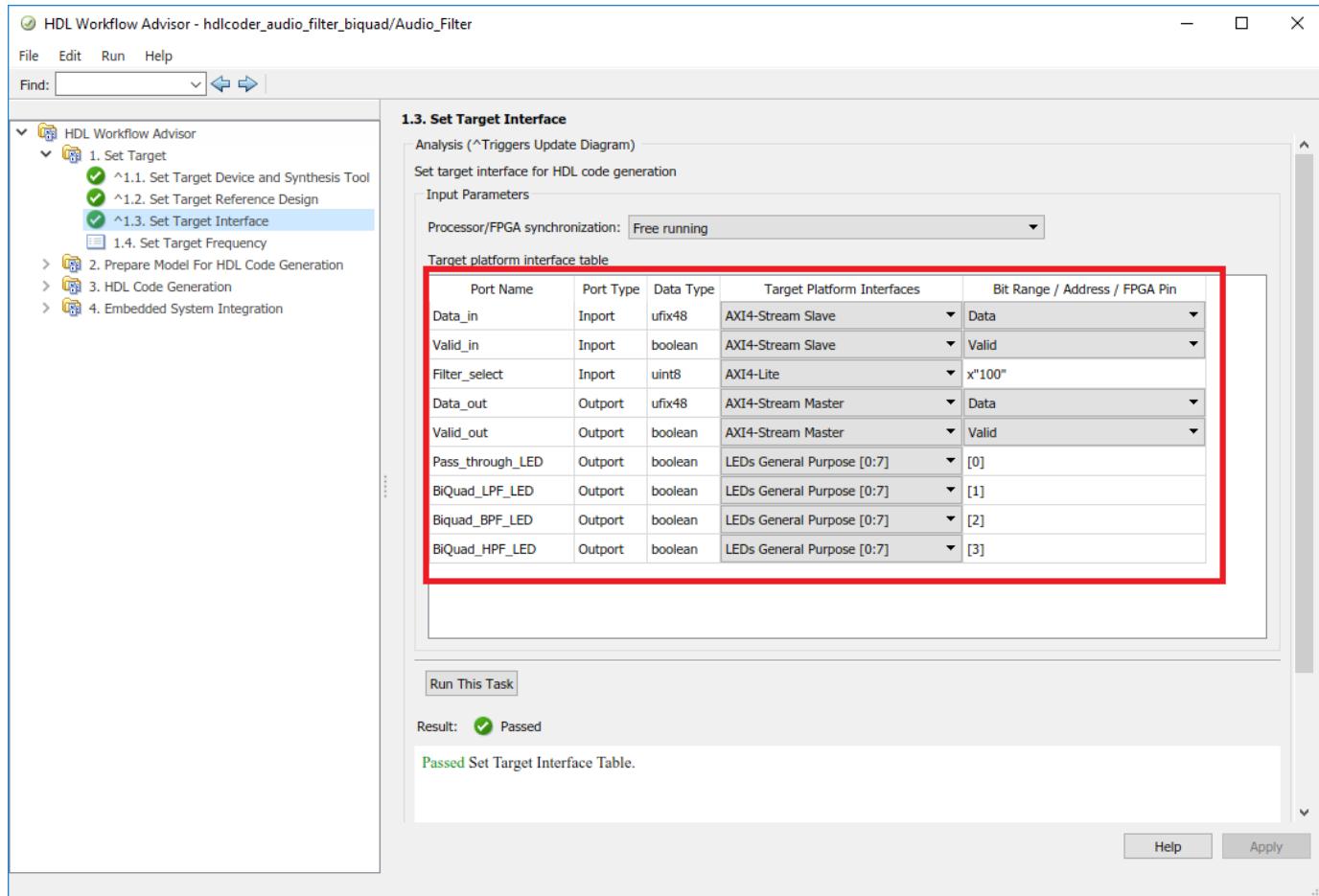
In Task 1.1, **IP Core Generation** is selected for **Target workflow**, and **ZedBoard** is selected for **Target platform**. If you are using Zybo board then select **ZYBO** as **Target Platform** instead of Zedboard.



In Task 1.2, **Audio System with AXI4 Stream Interface** is selected for **Reference Design**.



The AXI4-Stream interface is used for transferring audio data between the reference design and the filtering algorithm IP. The AXI4-Stream interface contains data (**Data**) and control signals such as data valid (**Valid**), back pressure (**Ready**), and data boundary (**TLAST**). At least **Data** and **Valid** signals are required for AXI4-Stream IP core generation. In Task 1.3, the **Target platform interface table** is loaded as shown in the following picture. The audio data stream ports, **Valid_in**, **Data_in**, **Valid_out** and **Data_out**, are mapped to the AXI4-Stream interfaces, **Pass_through_LED**, **BiQuad_LPF_LED**, **BiQuad_BPF_LED**, **BiQuad_HPF_LED** are mapped to the LEDs on Zedboard and the control parameter port **Filter_select** is mapped to the AXI4-Lite interface. If you are using Zybo board then map the LEDs to the filter manually by selecting **LEDs General Purpose[0:4]**.



The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Master** interface.

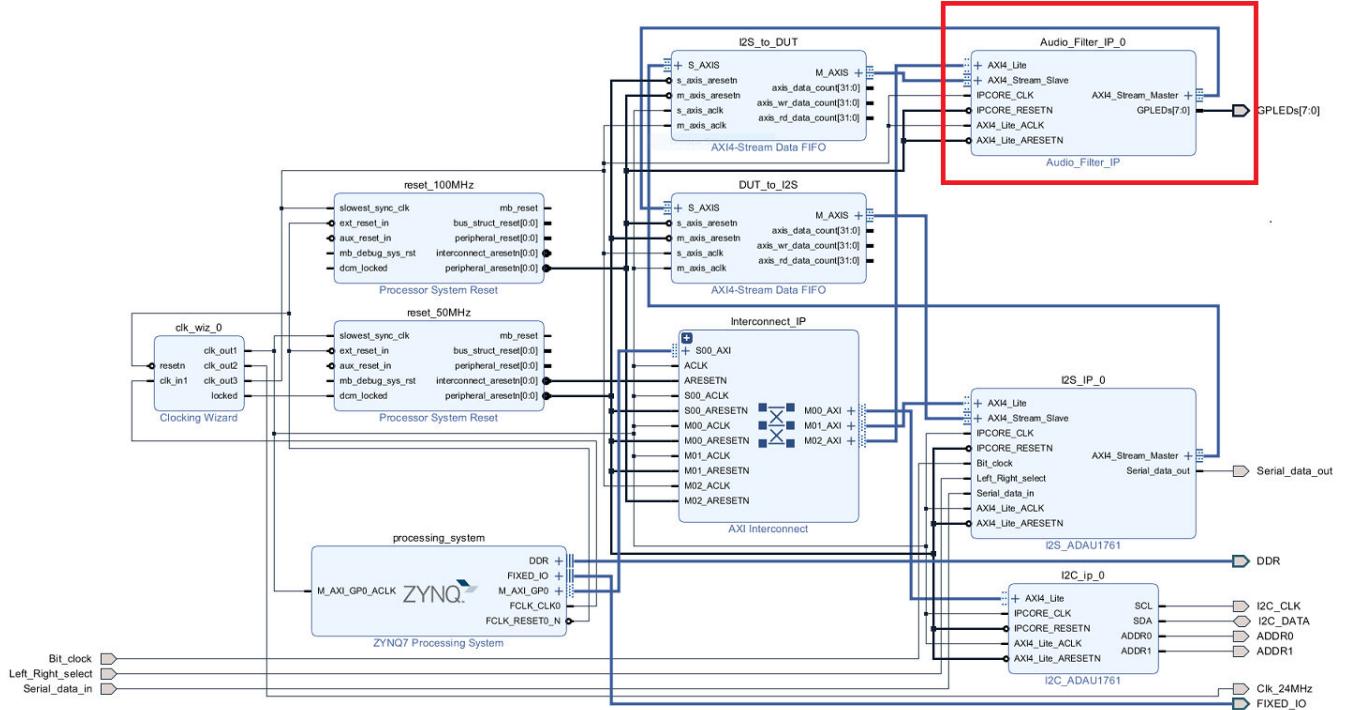
3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP Into AXI4-Stream Audio Compatible Reference Design

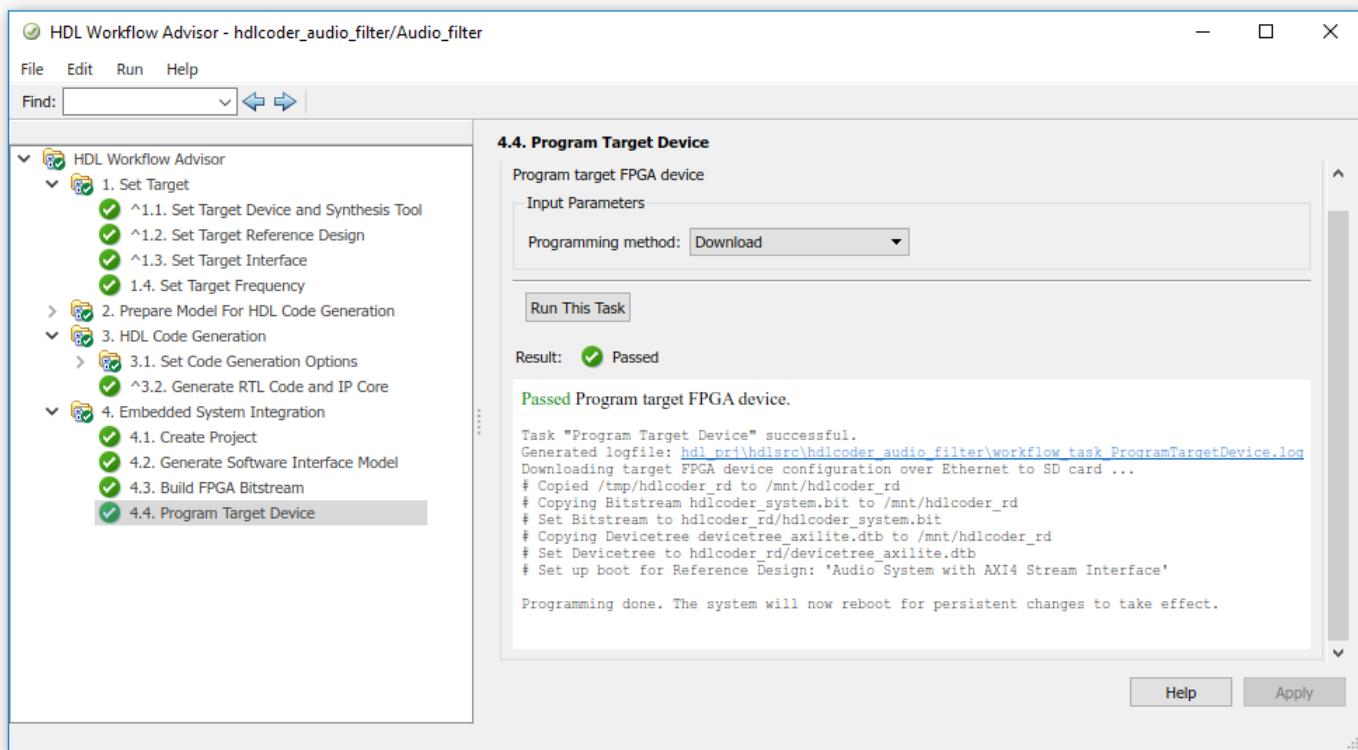
Next, in the HDL Workflow Advisor, you run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Audio System with AXI4 Stream Interface** reference design. As shown in the first diagram, this reference design contains the IPs to handle audio data in and out of Zedboard. The generated project is a complete Zynq design, including the algorithm part (the generated DUT algorithm IP), and the platform part (the reference design). For details on how to create a reference design which integrates the audio filter model, refer to "Authoring a Reference Design for Audio System on a Zynq Board" on page 41-170 or "Authoring a Reference Design for Audio System on a ZYBO Board" on page 41-180 example.

2. Click the link in the Result pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, other audio processing IPs and the Zynq processor.



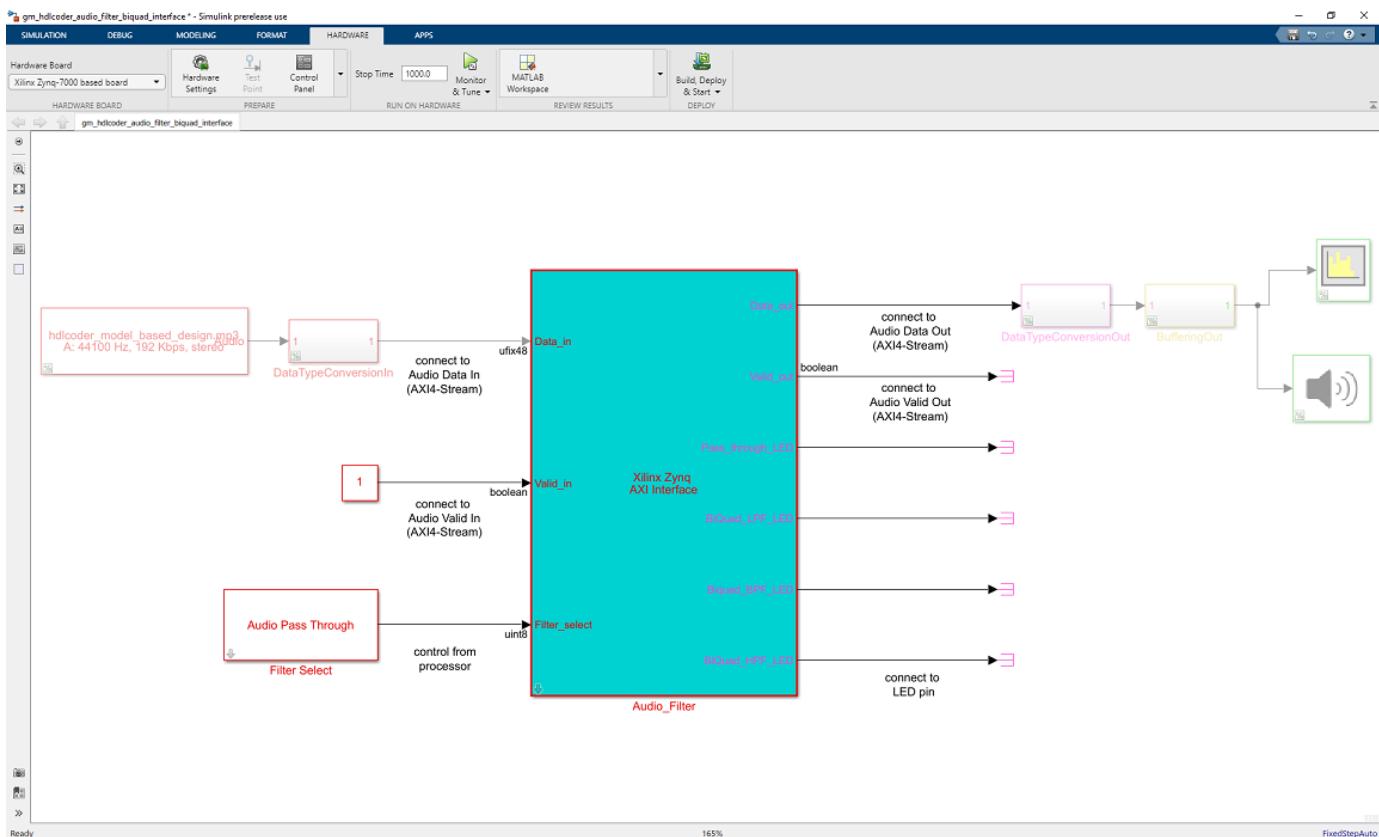
3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream. Choose **Download** programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the Zynq board, so your design will be automatically reloaded when you power cycle the Zynq board.



Generate ARM executable to Tune Parameters on the FPGA Fabric

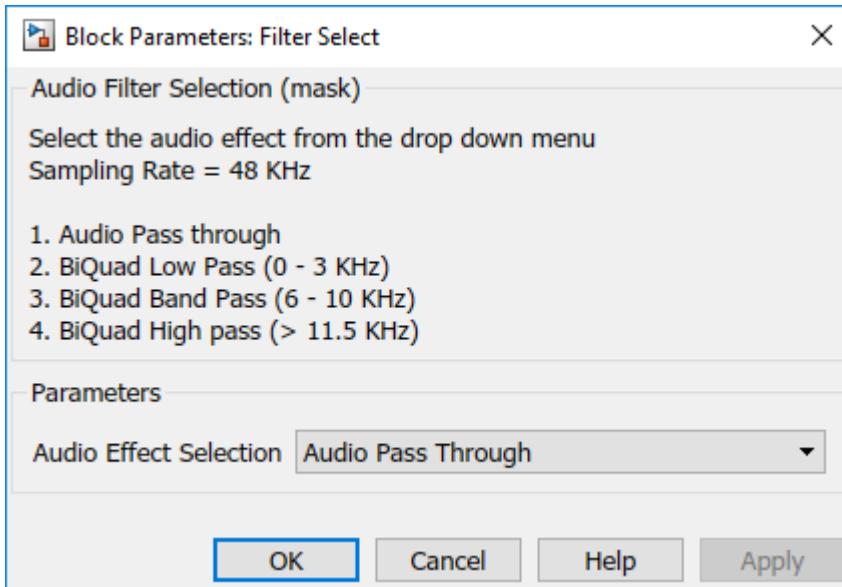
A software interface model is generated in Task 4.2, **Generate Software Interface Model**.

1. Before you generate code from the software interface model, comment out the audio input source and audio output sink i.e From Multimedia File, Data Type Conversion, Buffer, Audio Device Writer and Spectrum Analyzer Blocks. These blocks do not need to be run on the ARM processor. The Audio_filter IP is running as *Filtering_Algorithm" on FPGA fabric. The ARM processor is using AXI4-Lite interface for selecting the filter type i.e. Biquad Low pass, band pass, High pass or Pass Through.



- 1 In the generated model, click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
- 2 Select **Solver** and set "Stop Time" to "inf" and click ok.
- 3 From the Hardware pane, click the **Monitor and Tune** button.
- 4 Click the **Run** button on the model toolbar. Embedded Coder builds the model, downloads the ARM executable to the Zynq board hardware, executes it, and connects the model to the executable running on the Zynq board hardware.

The type of filter to be used can be selected using the drop down options in **Filter Select** block



The filtered audio output can be heard by plugging earphones or speakers to **HPH OUT** jack on the Zynq board. Depending on the filter selected, the corresponding LED on the Zynq board turns on. In this example, LD0 turns on when Pass through (No filter used) option is selected, LD1 turns on when Biquad Low pass filter is selected, LD2 turns on when Biquad Band pass filter is selected and LD3 turns on when Biquad High pass filter is selected.

Getting Started with AXI4-Stream Interface in Zynq Workflow

This example shows how to use the AXI4-Stream interface to enable high speed data transfer between the processor and FPGA on Zynq hardware.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

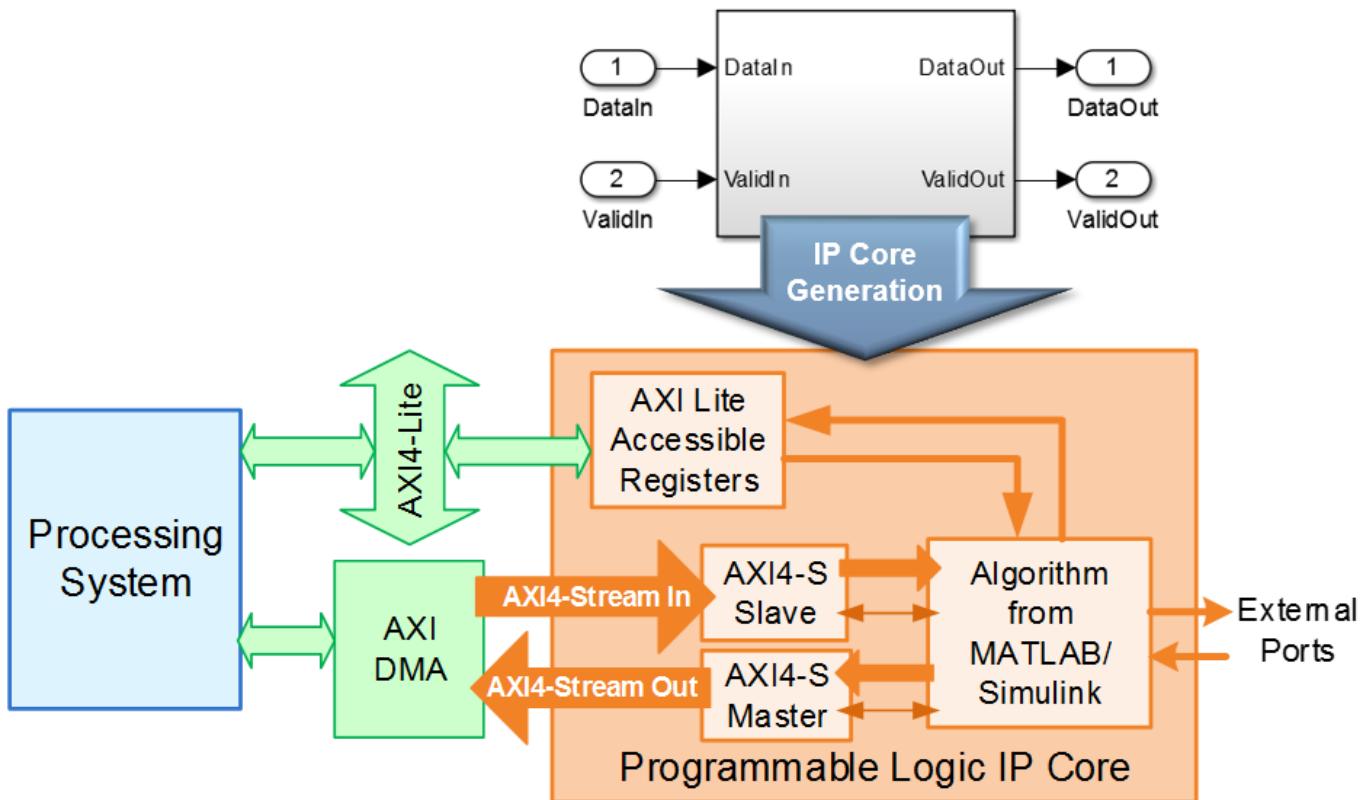
- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform
- Xilinx Vivado Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Zedboard

To setup the Zedboard, refer to the *Set up Zynq hardware and tools* section in the example “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65.

Introduction

This example shows how to:

- 1 Model a streaming algorithm using a simplified streaming protocol.
- 2 Generate an HDL IP core with AXI4-Stream interface.
- 3 Integrate the generated IP core into a Zedboard reference design with DMA controller.
- 4 Use the AXI4-Stream driver block to generate C code that runs on an ARM processor.



The picture above is a high level architecture diagram that shows a streaming data transfer between the processor and FPGA fabric on Zynq platform. Typically, the AXI4-Stream interface is used together with a DMA controller to transfer a large chunk of data from the processor to FPGA. The data is usually represented as vector data on the software side. The DMA controller reads the vector data from memory, and "streams" it to the FPGA IP through the AXI4-Stream interface. The "streaming" process sends one data element per sample, which means the data path of the streaming algorithm in the FPGA IP is using a scalar data type.

The FPGA IP can also include an AXI4-Lite interface for control signals or parameter tuning. Compared to the AXI4-Lite interface, the AXI4-Stream interface transfers data much faster, making it more suitable for the data path of an algorithm.

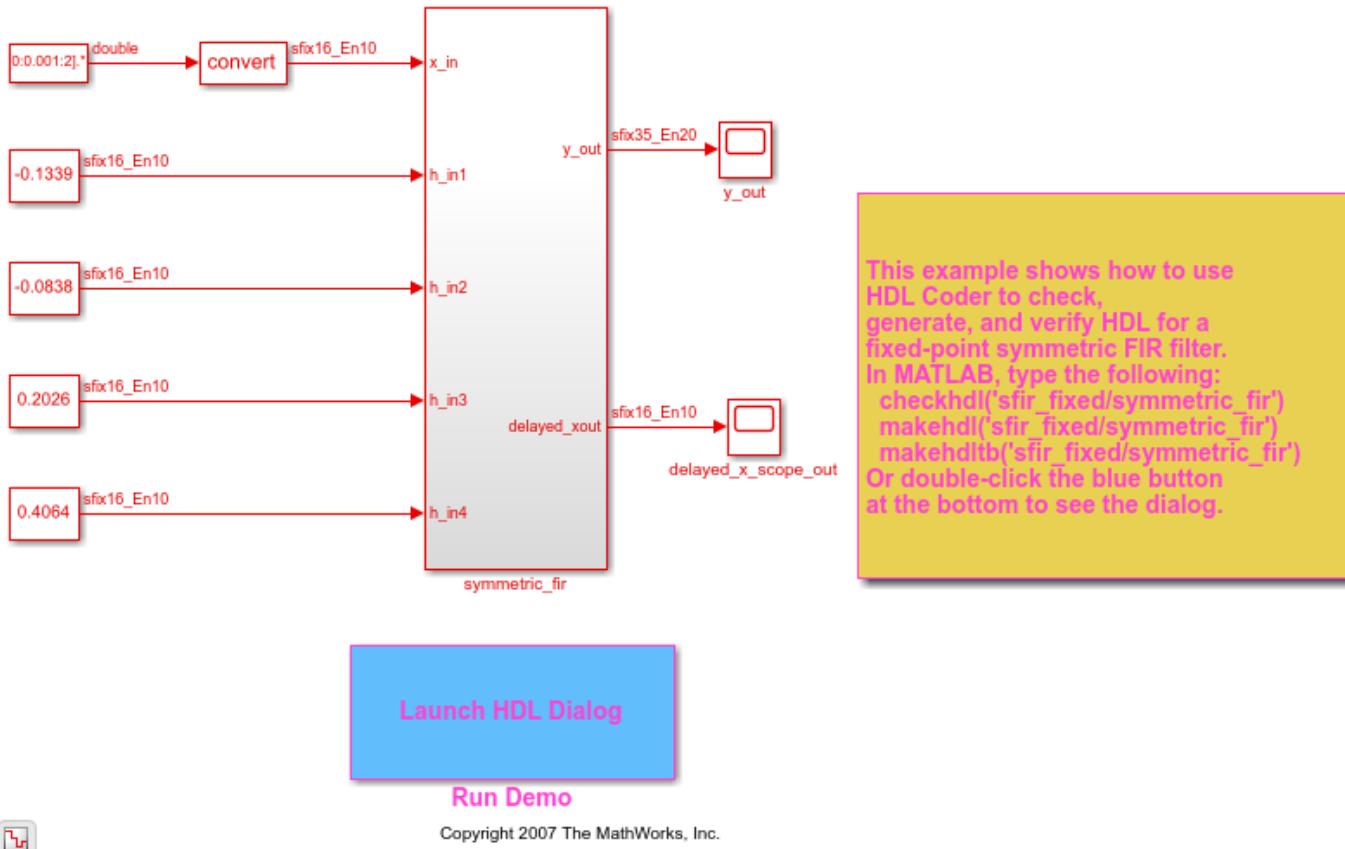
Other than connecting to processor, the FPGA IP with AXI4-Stream interface can also be connected with other IPs with AXI4-Stream interface to transfer data inside of FPGA.

Model Streaming Algorithm with Simplified Streaming Protocol

Suppose we want to deploy a simple symmetric FIR filter on Zynq. We want to implement the filter on FPGA. And the ARM processor generates the source data to stream it to FPGA through the AXI4-Stream interface.

Let's start with the `sfir_fixed` model.

```
open_system('sfir_fixed')
set_param('sfir_fixed', 'SimulationCommand', 'Update')
```

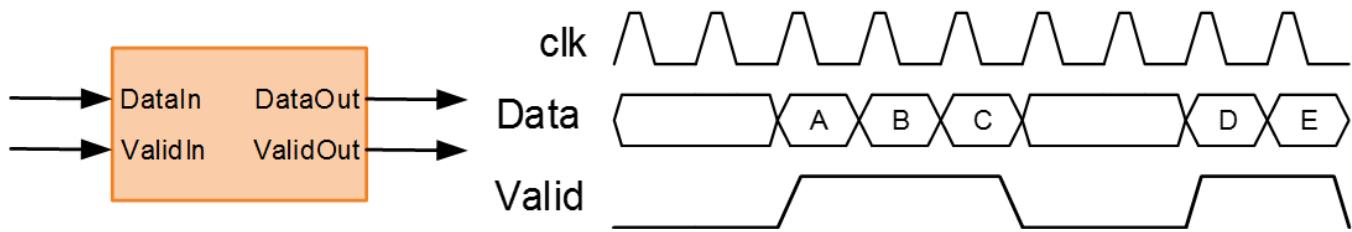


Note that the data path of this model (from **x_in** to **y_out**) is processing scalar input data, which is suitable for a streaming interface.

In order to enable data transfer from the software to the filter algorithm, we need to map the data path ports to the AXI4-Stream interface. The AXI4-Stream interface contains data (**Data**) and control signals such as data valid (**Valid**), back pressure (**Ready**), and data boundary (**TLAST**).

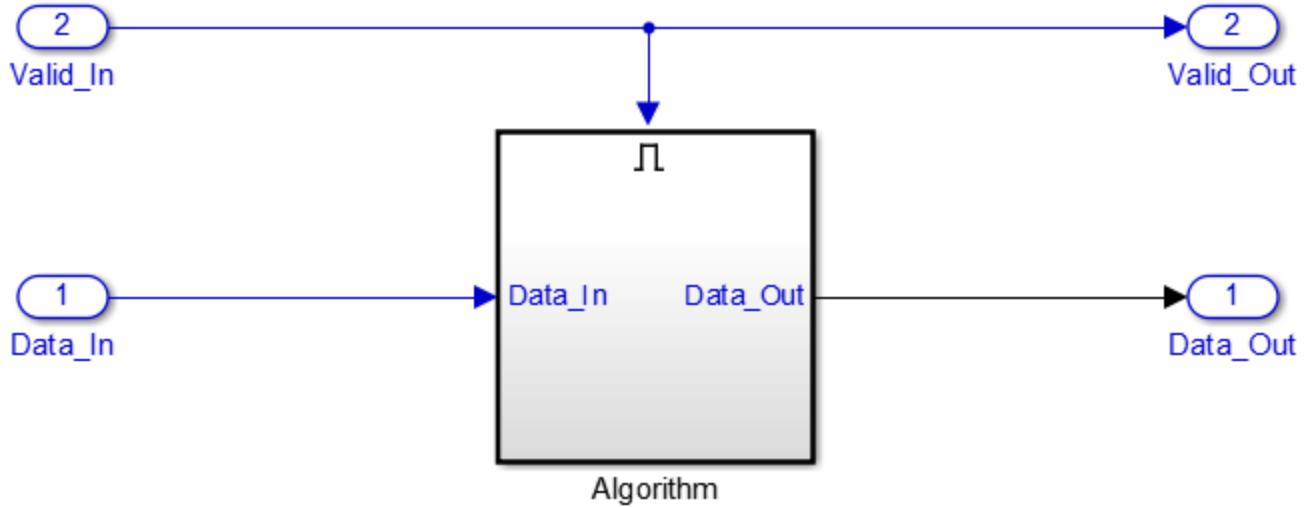
The AXI4-Stream IP core generation feature requires at least the **Data** and **Valid** signals to be modeled in the DUT. The **Data** signal is the primary payload to send across the interface. The **Valid** signal indicates when the **Data** signal is valid. Other control signals are optional.

Note: For IP core generation, **Data** and **Valid** follow a simplified streaming protocol. You don't need to model the full AXI4-Stream protocol, which is more complicated. HDL Coder automatically generates a streaming interface module in the HDL IP core to translate the simplified streaming protocol into the full AXI4-Stream protocol. As shown in the picture below, the protocol is simple: whenever the **Data** signal is valid, the **Valid** signal must also be asserted.



So, in order to map `sfir_fixed` algorithm to the simplified streaming protocol, a **Valid** signal needs to be added. To add the **Valid** signal to your model, we recommend following modeling pattern:

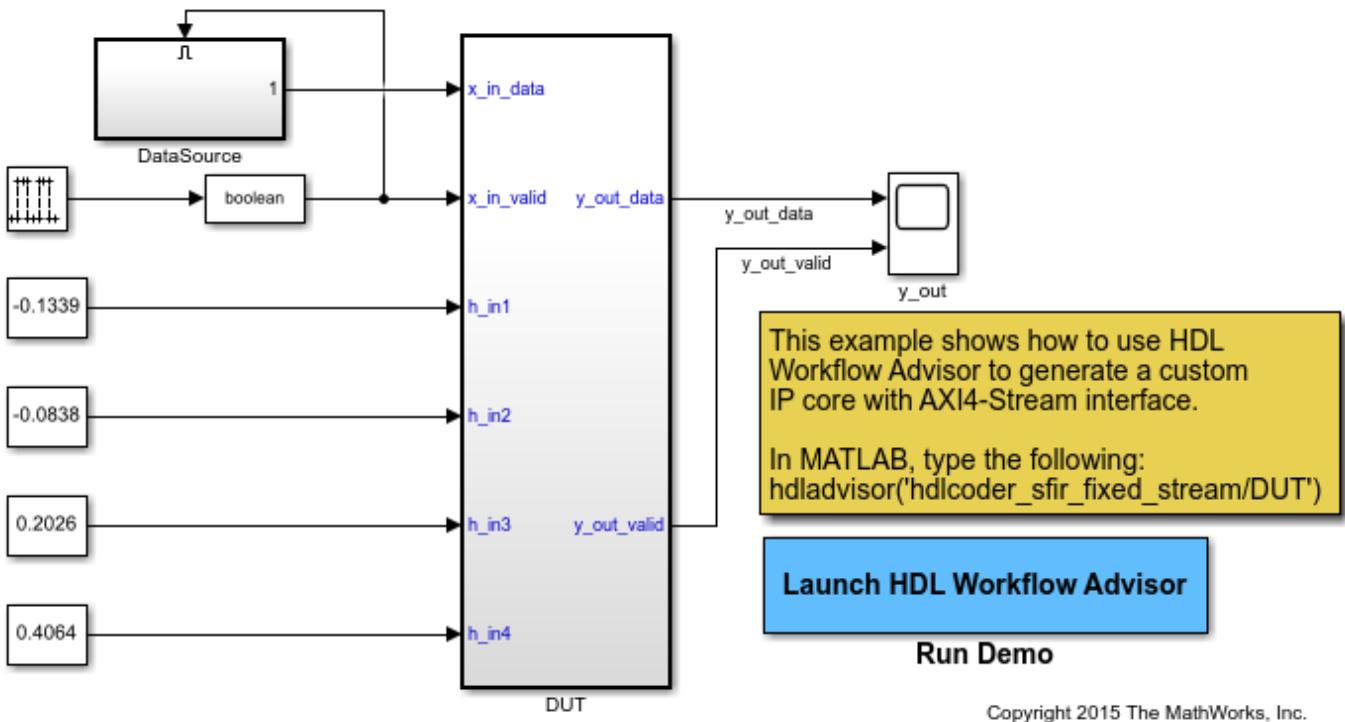
- 1 Convert the algorithm subsystem into an enabled subsystem.
- 2 Add an input control port, **Valid_In**, and output control port, **Valid_Out**.
- 3 Use **Valid_In** to drive both the algorithm subsystem's enable port and **Valid_Out**.



In this pattern, both the input streaming channel and output streaming channel follow the simplified streaming protocol.

Now, let's look at the example model.

```
open_system('hdlcoder_sfir_fixed_stream');
```

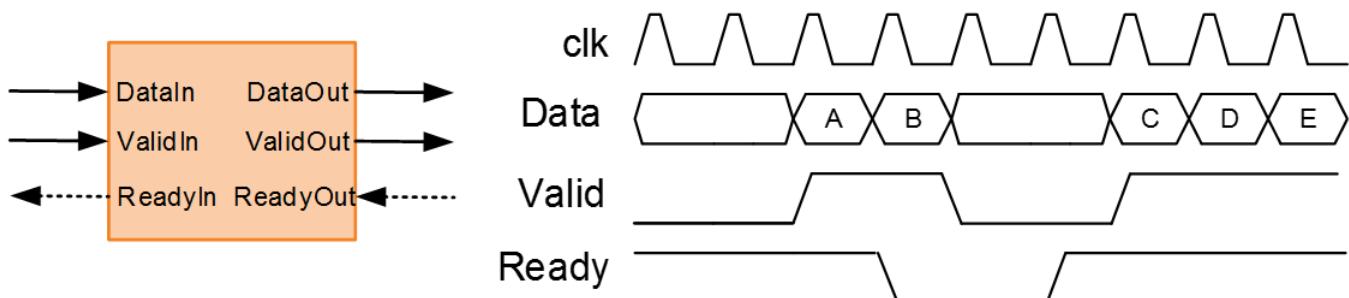


The subsystem **DUT** is the hardware subsystem targeting the FPGA fabric. Inside this subsystem, the **symmetric_fir** subsystem represents the filter algorithm. The input ports, **x_in_data** and **x_in_valid**, and output ports, **y_out_data** and **y_out_valid**, are the data path ports of the filter. The other input ports, such as **h_in1**, are control ports that tune the filter parameters.

The model follows the modeling pattern for simplified streaming protocol. The **symmetric_fir** subsystem is an enabled subsystem. The input control signal, **x_in_valid**, controls the **symmetric_fir** subsystem's enable port and also drives the output control signal, **y_out_valid**.

With AXI4-Stream IP core generation, you can optionally model other streaming control signals. For example, you can model the back pressure signal, **Ready**. The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. The **Ready** signal is a back pressure signal from the slave device to master device that indicates whether the slave device can accept new data. As shown in following diagram, the **Ready** signal is asserted when the slave device can accept new data. When the slave device can no longer accept new data, it needs to de-assert the **Ready** signal. When the master device sees that the **Ready** signal is deasserted, it stops the data transfer at most one sample later. This one sample allowance is built into the protocol.

Note: This diagram illustrates the relationship between the Data, Valid, and Ready signals according to the simplified streaming protocol. When you run the IP Core Generation workflow, the code generator adds a streaming interface module in the HDL IP core that translates the simplified protocol to the full streaming protocol.



For example, you can use the **Ready** signal when you use a FIFO block to collect a frame of incoming streaming data, which is then processed with your algorithm. During data processing, you deassert the **Ready** signal to prevent further incoming data.

Generate HDL IP core with AXI4-Stream Interface

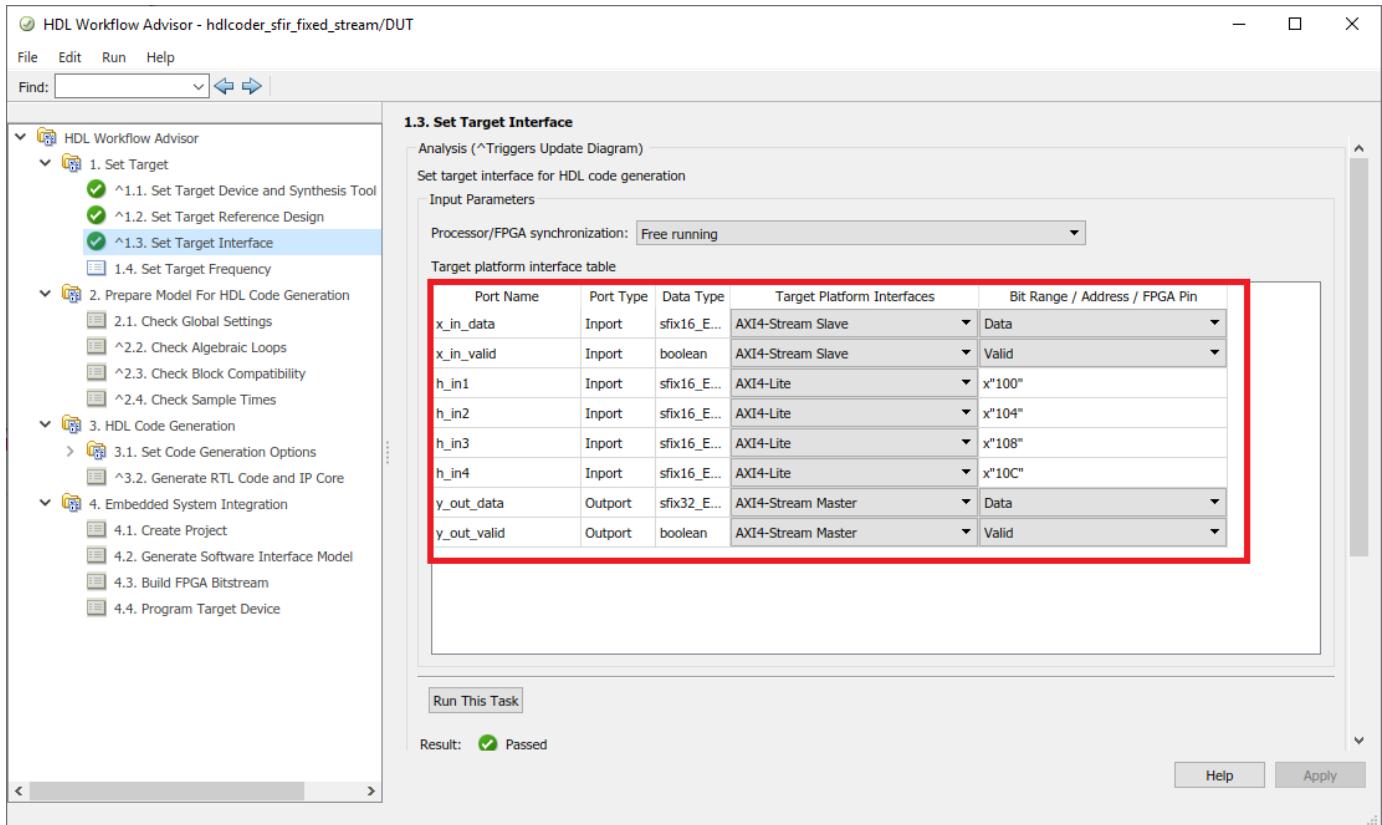
Next, we start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.1\bin\vivado')
```

2. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_sfir_fixed_stream/DUT`. The target interface settings are already saved in this example model, so the settings in Task 1.1 and 1.2 are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the “Save Target Hardware Settings in Model” on page 40-137 example.

In Task 1.1, **IP Core Generation** is selected for **Target workflow**, and **Zedboard** is selected for **Target platform**. In Task 1.2, **Default system with AXI4-Stream interface** is selected for **Reference Design**, and the **Target platform interface table** is loaded as shown in the following picture. The data path ports, `x_in_data`, `x_in_valid`, `y_out_data`, and `y_out_valid`, are mapped to the AXI4-Stream interfaces, and the control parameter ports, such as `h_in1`, are mapped to the AXI4-Lite interface.



The AXI4-Stream interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Master** interface.

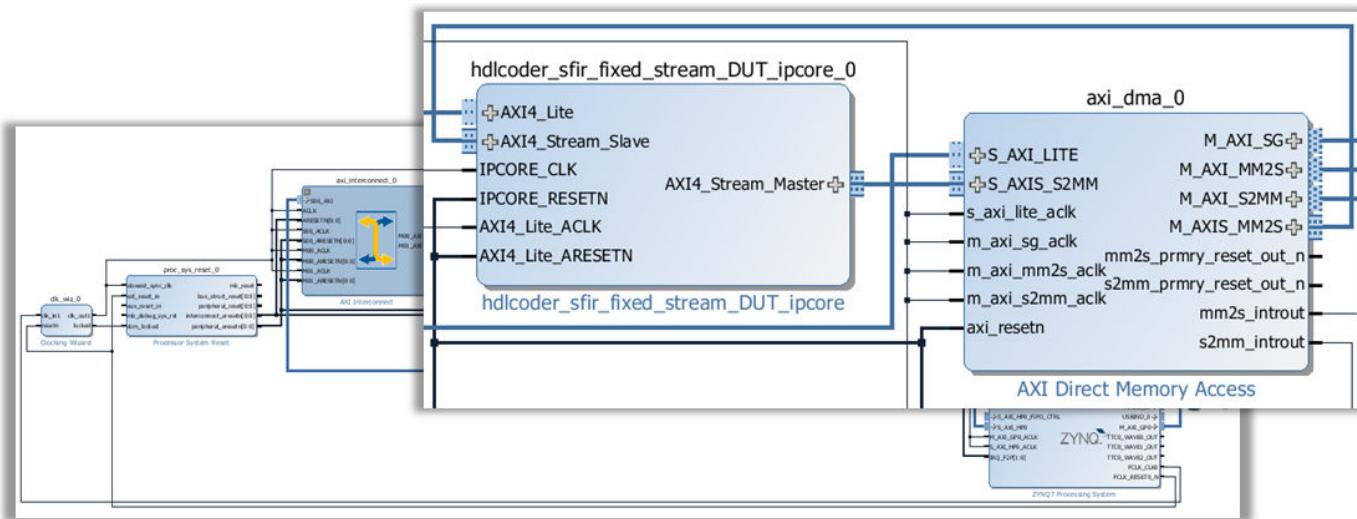
3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP Into AXI4-Stream Compatible Reference Design

Next, in the HDL Workflow Advisor, we run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Default system with AXI4-Stream interface** reference design. This reference design contains Xilinx AXI DMA IP to handle the processor to FPGA fabric data streaming. As shown in the first diagram, or in the IP core report, the data is sent from the ARM processing system, through the DMA controller and AXI4-Stream interface, to the generated HDL FIR filter IP core. The output of the filter IP core is then sent back to the processing system.

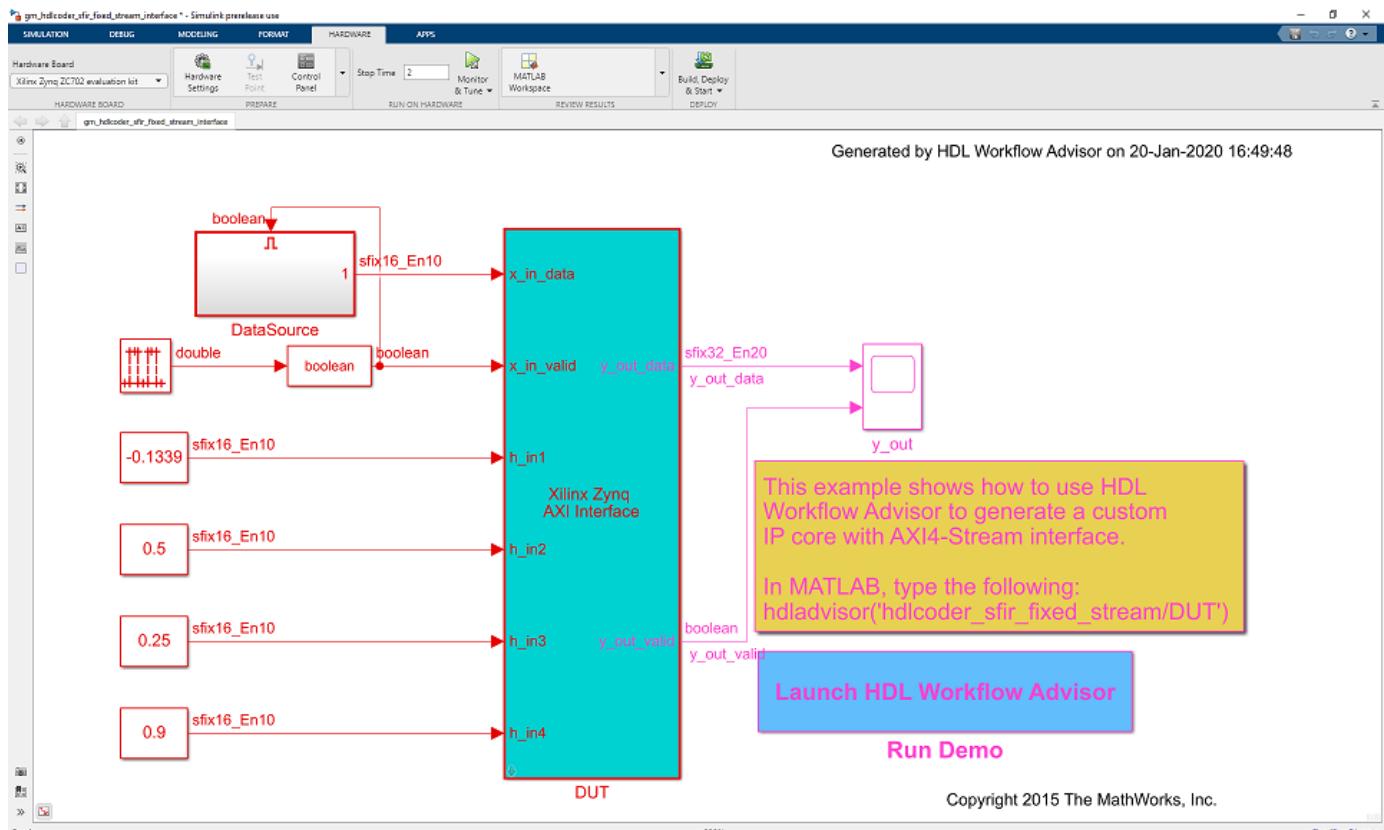
2. Optionally click the link in the Result pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, AXI DMA controller and the processor.



3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream.

Generate ARM executable Using AXI4-Stream Driver Block

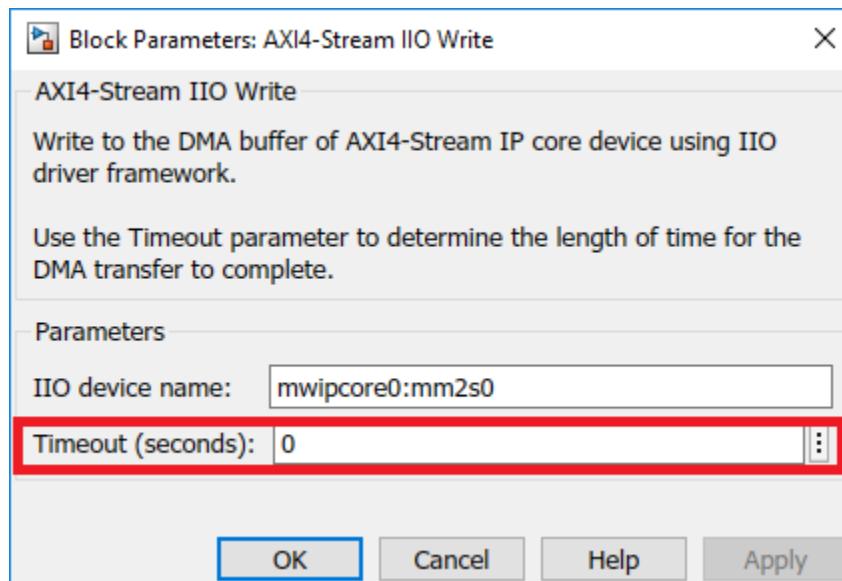
A software interface model is generated in Task 4.2, **Generate Software Interface Model**, as shown in the following picture.



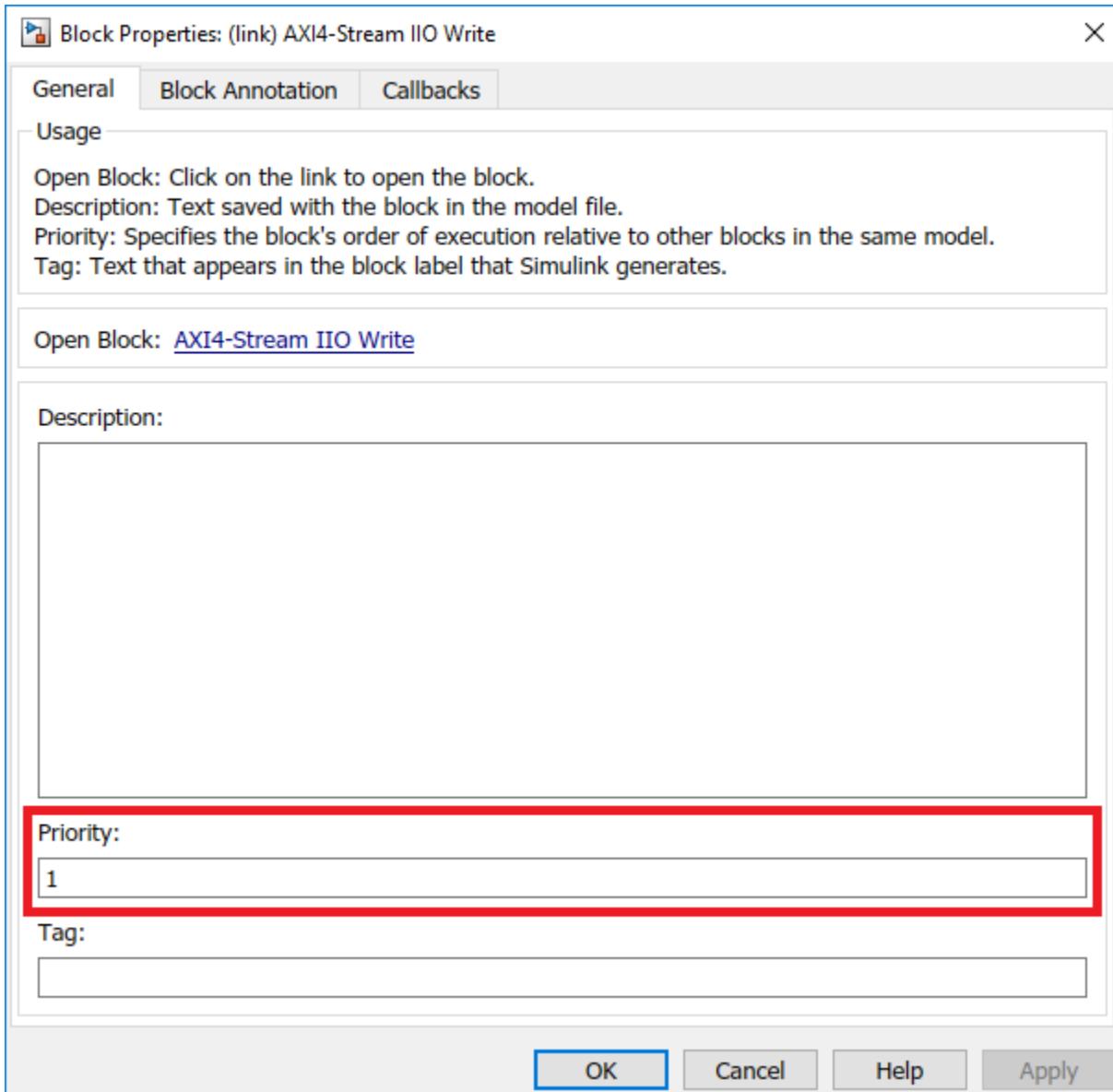
Although the AXI4-Lite driver is automatically generated in the software interface model, the AXI4-Stream driver block cannot be automatically generated. The reason is that the AXI4-Stream driver block expects to be connected to a vector port on the software side, but the **x_in_data** DUT port is a scalar port.

1. Before you generate code from the software interface model:

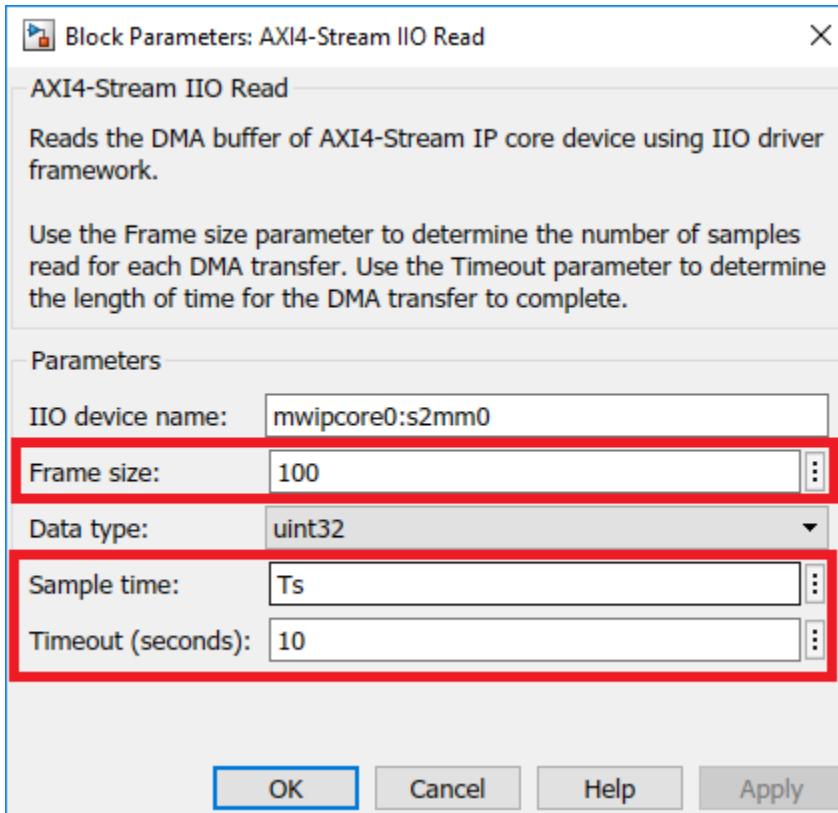
- 1 Add the **AXI4-Stream IIO Read** and **AXI4-Stream IIO Write** driver blocks from **Simulink Library Browser -> Embedded Coder Support Package for Xilinx Zynq Platform** library.
- 2 Use a vector data source to drive the **x_in_data** port.
- 3 Connect the **x_in_data** port to the driver block.
- 4 Double click on the **AXI4-Stream IIO Write** block and set the Timeout to 0 instead of inf. This is as shown below.



5. Set the priority of the **AXI4-Stream IIO Write** block to 1 to make sure that write happens before read. To set the priority, right click on the block and open properties, set the priority to 1. This is as shown below.

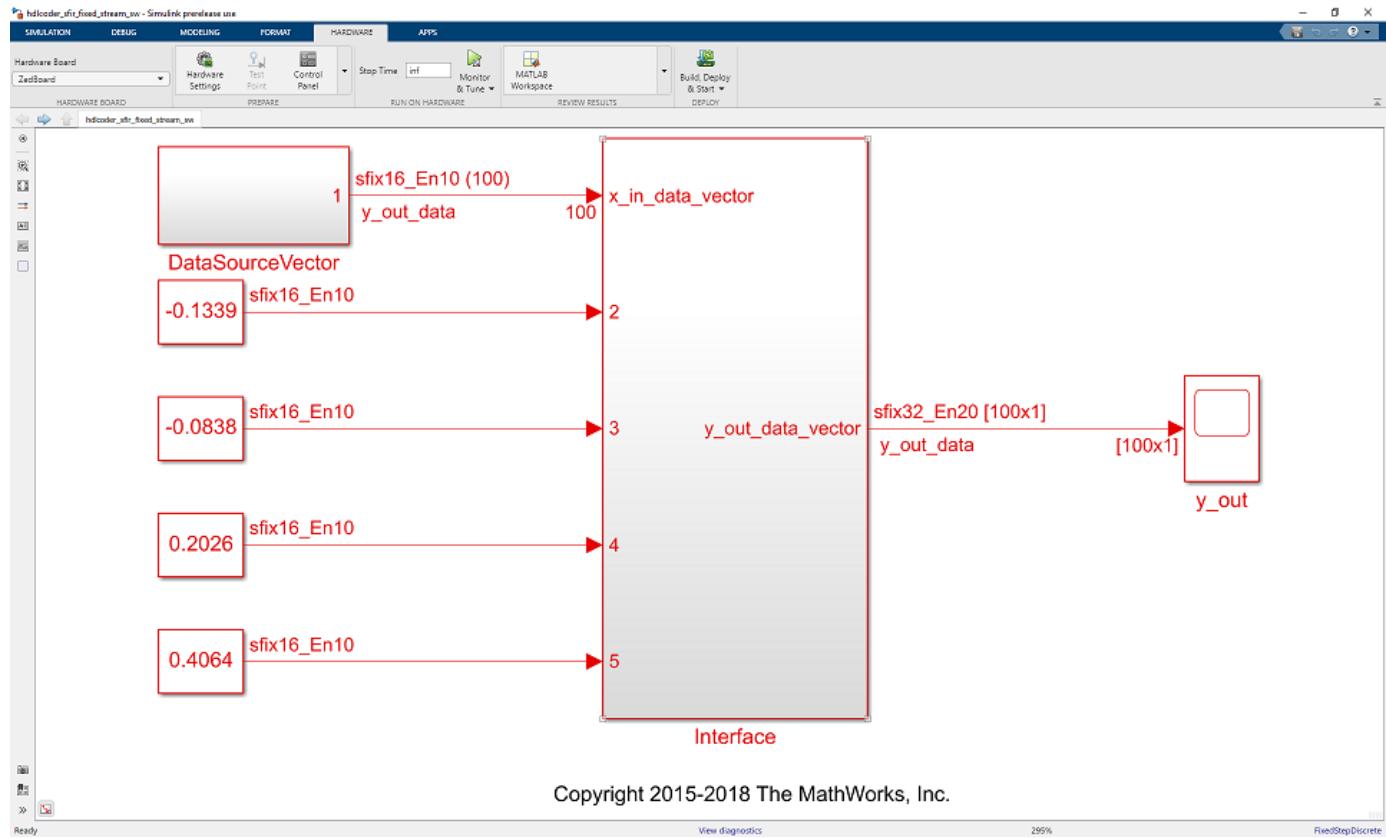


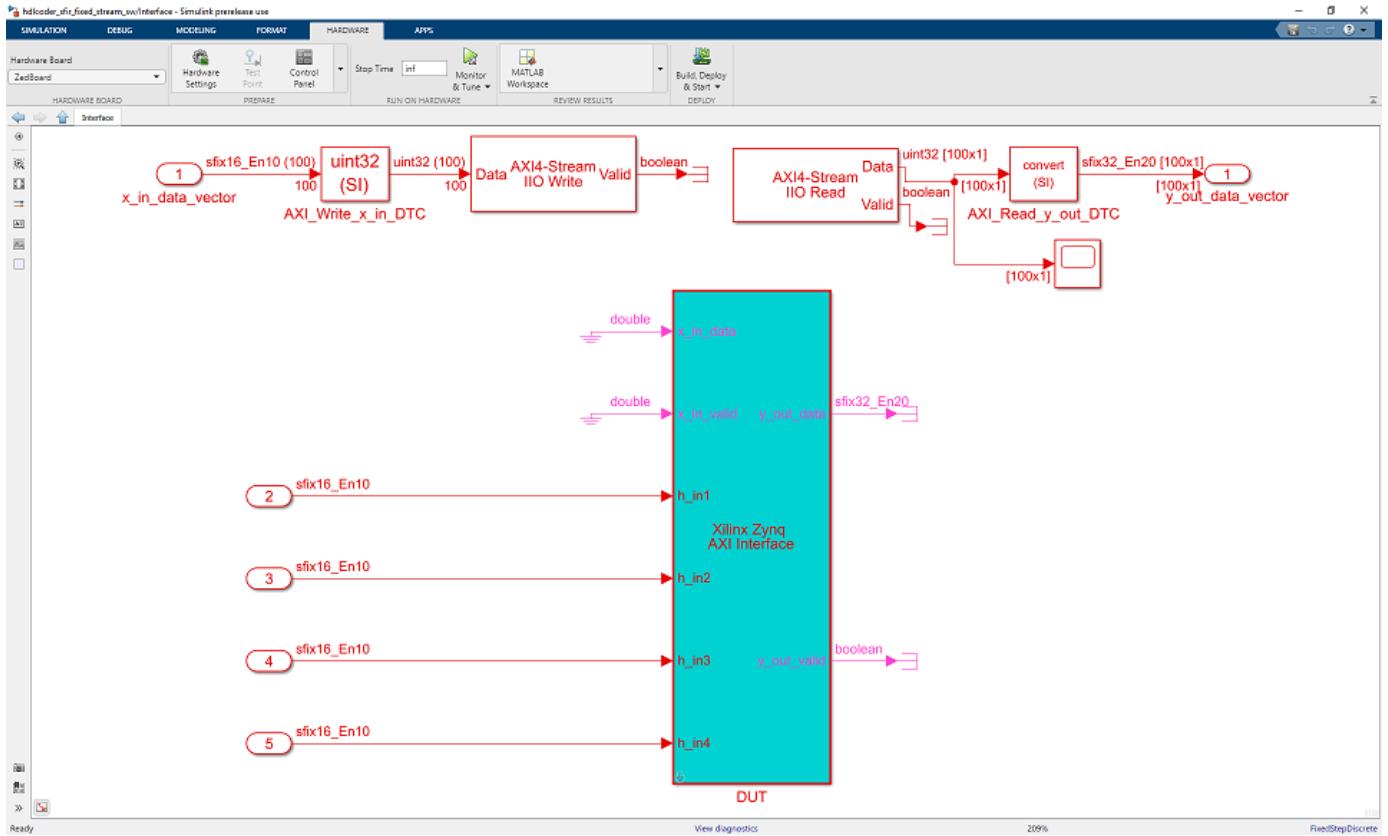
6. Now double click on the **AXI4-Stream IIO Read** block and set the frame size to 100, Sample time to Ts and Timeout to 10. This is as shown below.



7. The priority of the **AXI4-Stream IIO Read** block need not to be set. Setting the priority for write block to 1 alone already ensure that write happens before read.

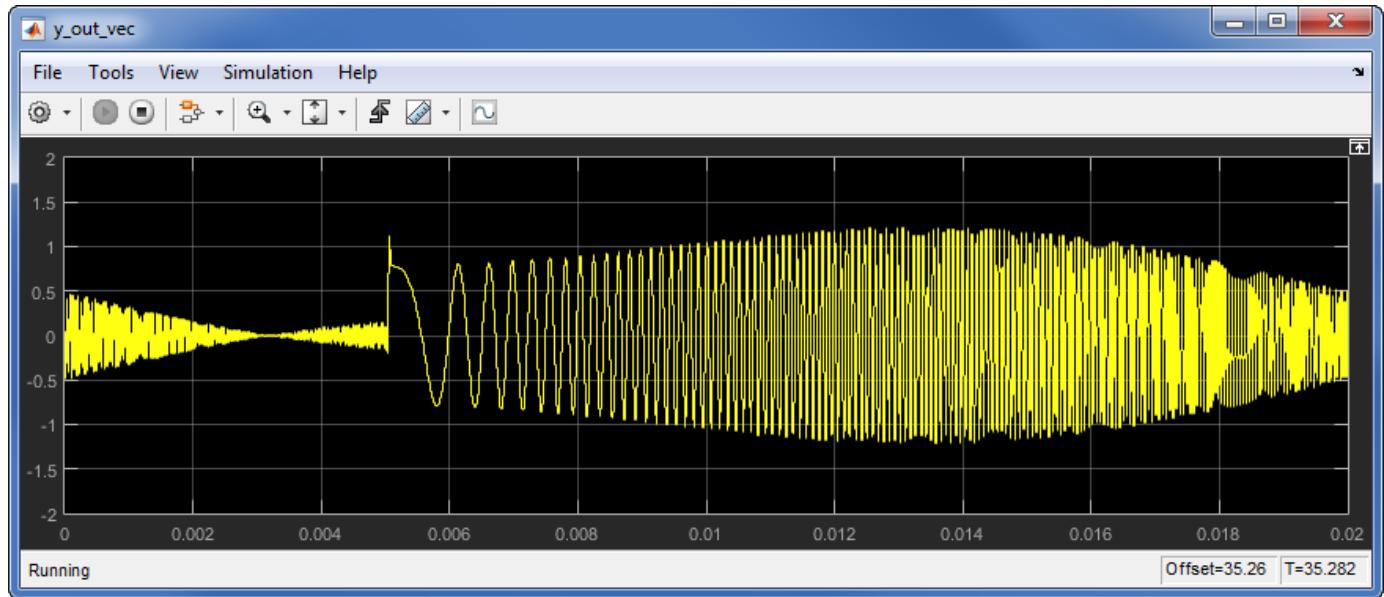
For this example, the updated software interface model is provided: `hdlcoder_sfir_fixed_stream_sw.slx`. A vector data source with 100 data elements is used in this model, and is connected to the AXI4-Stream DMA driver block. This means that for each processor sample time, the DMA controller will stream 100 32-bit data samples to the HDL IP core via the AXI4-Stream interface, and receive 100 32-bit streaming data samples.



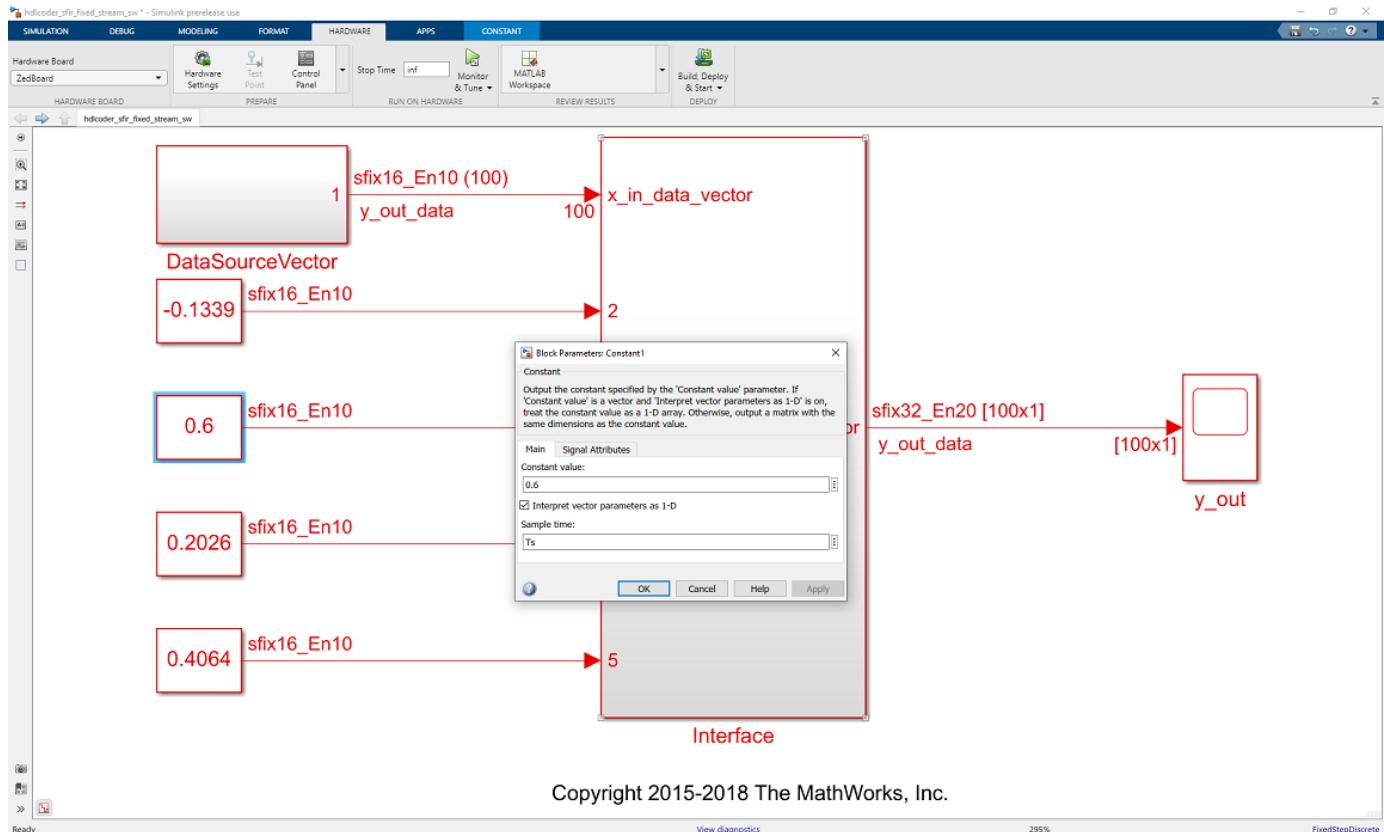


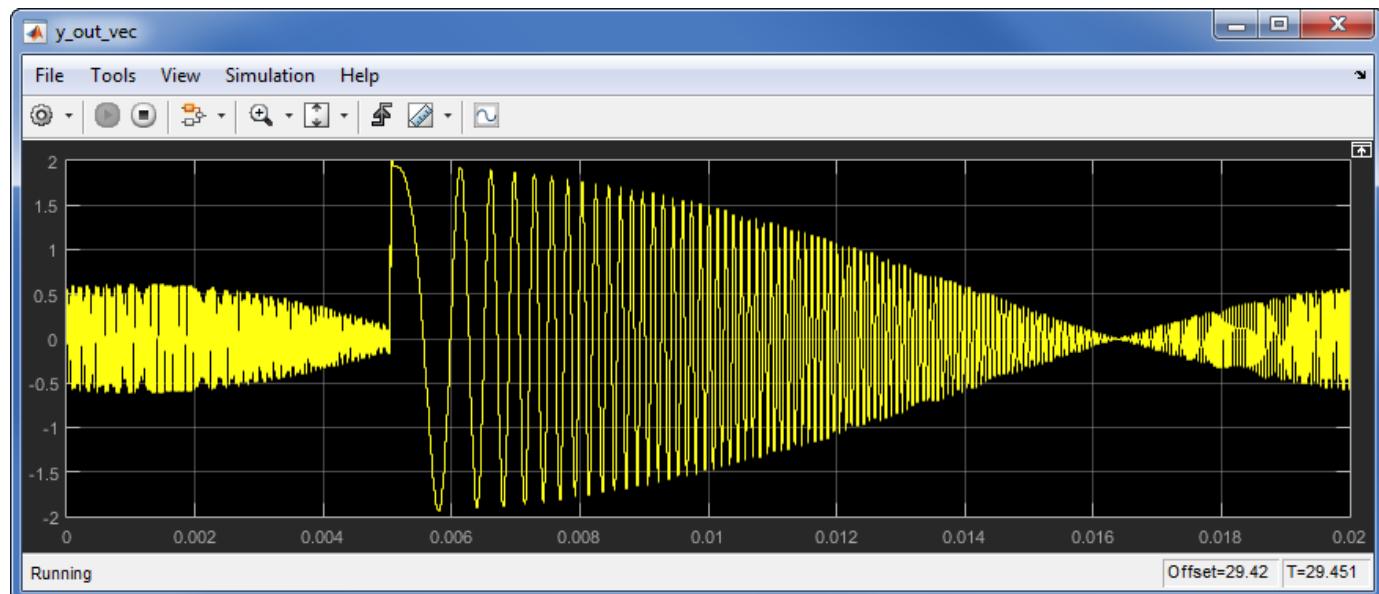
2. Configure and build the software interface model for external mode:

- 1 In the generated model, click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
 - 2 Select **Solver** and set "Stop Time" to "inf".
 - 3 From the Hardware pane, click the **Monitor and Tune** button.
 - 4 Click the **Run** button on the model toolbar. Embedded Coder builds the model, downloads the ARM executable to the Zedboard hardware, executes it, and connects the model to the executable running on the Zedboard hardware.
- 3.** Now, both the hardware and software parts of the design are running on Zynq hardware. The ARM processor sends the source data to the FPGA IP, through the DMA controller and the AXI4-Stream interface. The ARM processor receives the filter result data from the FPGA IP, and sends the result data to Simulink via external mode. Observe the output of the FIR filter IP core from the Zynq hardware on the Time Scope **y_out**.



4. Tune the FIR filter parameters in the software interface model and observe how the output of the FIR filter changes as you tune the parameters. The parameter values are sent to the Zynq hardware via external mode and the AXI4-Lite interface.





Getting Started with AXI4-Stream Video Interface in Zynq Workflow

This example shows how to use the AXI4-Stream Video interface to enable high speed video streaming on the generated HDL IP core.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

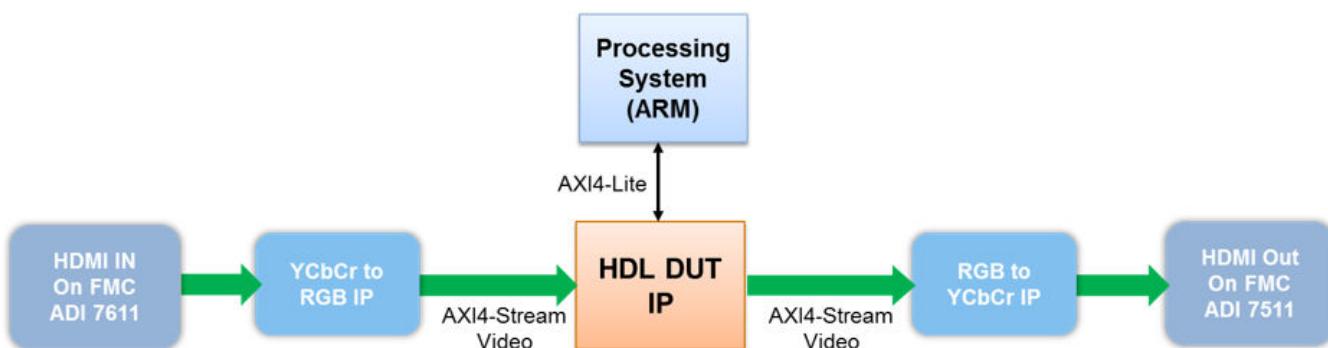
- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform
- Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware
- Vision HDL Toolbox
- Xilinx Vivado Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- ZedBoard
- FMC HDMI I/O card (FMC-HDMI-CAM or FMC-IMAGEON)

To setup the ZedBoard, refer to the *Set up Zynq hardware and tools* section in this example.

Introduction

This example shows how to:

- 1 Model a video streaming algorithm using the streaming pixel protocol.
- 2 Generate an HDL IP core with AXI4-Stream Video interface.
- 3 Integrate the generated IP core into a ZedBoard video reference design with access to HDMI interfaces.
- 4 Use the ARM® processor to tune the parameters on the FPGA fabric to change the live video output.
- 5 Create your own custom video reference design.



The picture above is a high level architecture diagram that shows how the generated HDL DUT IP core works in a pre-defined video reference design. In this diagram, the **HDLC DUT IP** block is the IP core that is generated from the IP core generation workflow. The rest of the diagram represents the

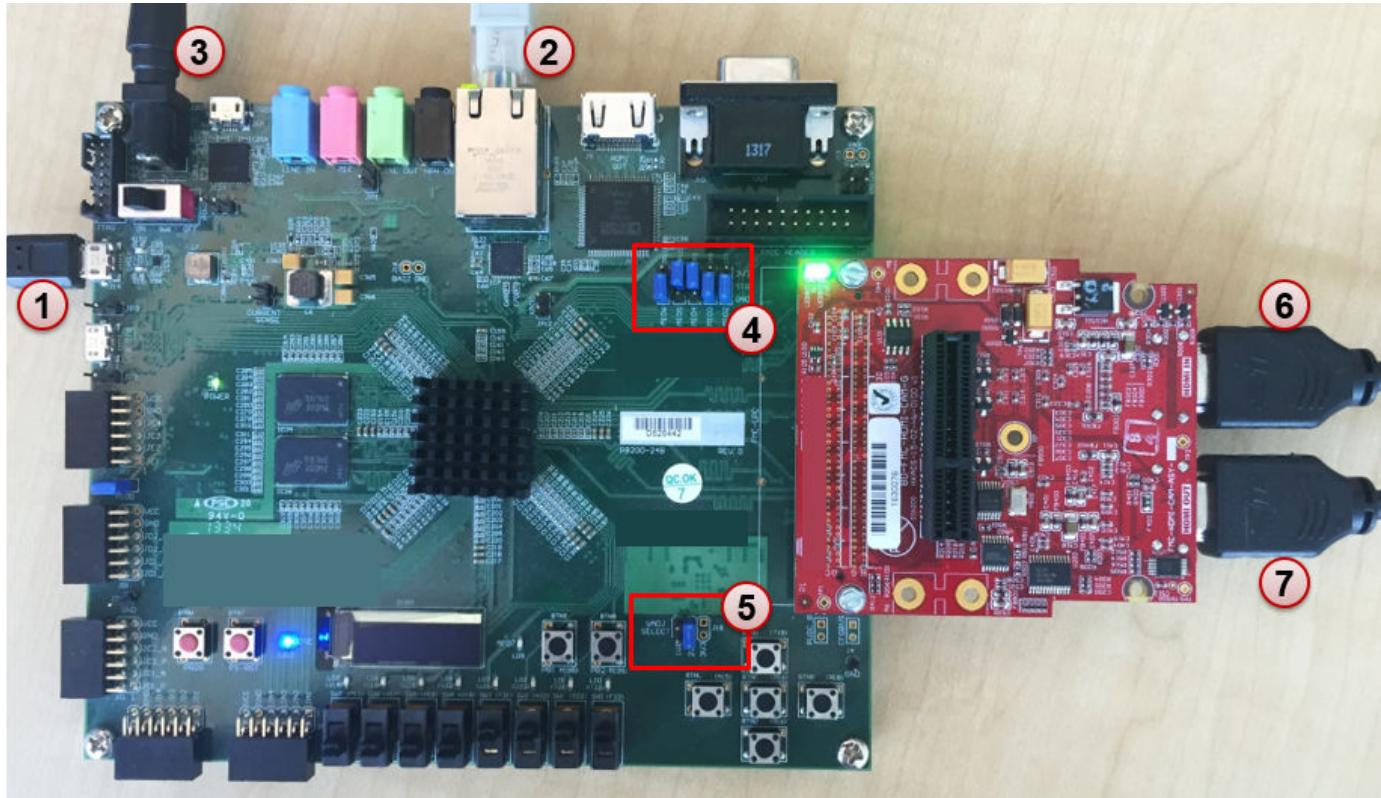
pre-defined video reference design, which contains other IPs to handle the HDMI input and output interfaces.

The **HDL DUT IP** processes a video stream coming from the HDMI input IP, generates an output video stream, and sends it to the HDMI output IP. All of these video streams are transferred in AXI4-Stream Video interface.

The **HDL DUT IP** can also include an AXI4-Lite interface for parameter tuning. Compared to the AXI4-Lite interface, the AXI4-Stream Video interface transfers data much faster, making it more suitable for the data path of the video algorithm.

Set up Zynq hardware and tools

1. Set up the ZedBoard and the FMC HDMI I/O card as shown in the figure below. To learn more about the ZedBoard hardware setup, please refer to the board documentation.



1.1. Connect the USB UART cable, the Ethernet cable and the power cable as shown in the figure above (marker 1 to 3).

1.2. Make sure the JP7 to JP11 jumpers are set as shown in the figure above (marker 4), so you can boot Linux from the SD card. JP7: down; JP8: down; JP9: up; JP10: up; JP11: down.

1.3. Make sure the J18 jumper are set on 2V5 as shown in the figure above (marker 5).

1.4. Connect HDMI video source to the FMC HDMI I/O card as shown in the figure above (marker 6). The video source must be able to provide 1080p video output, for example, it could be a video camera, smart phone, tablet, or your computer's HDMI output.

1.5. Connect a monitor to the FMC HDMI I/O card as shown in the figure above (marker 7). The monitor must be able to support 1080p display.

2. If you haven't already, install the HDL Coder and Embedded Coder Support Packages for Xilinx Zynq Platform, and Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware. To install the support package, go to the MATLAB® toolbar and click **Add-Ons > Get Hardware Support Packages**.

3. Make sure you are using the SD card image provided by the Embedded Coder Support Package for Xilinx Zynq Platform. If you need to update your SD card image, run the following command at the MATLAB prompt:

```
targetupdater
```

4. Set up the Zynq hardware connection by entering the following command in the MATLAB command window:

```
h = zynq
```

The `zynq` function logs in to the hardware via COM port and runs the `ifconfig` command to obtain the IP address of the board. This function also tests the Ethernet connection.

5. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2017.4\bin\vivado')
```

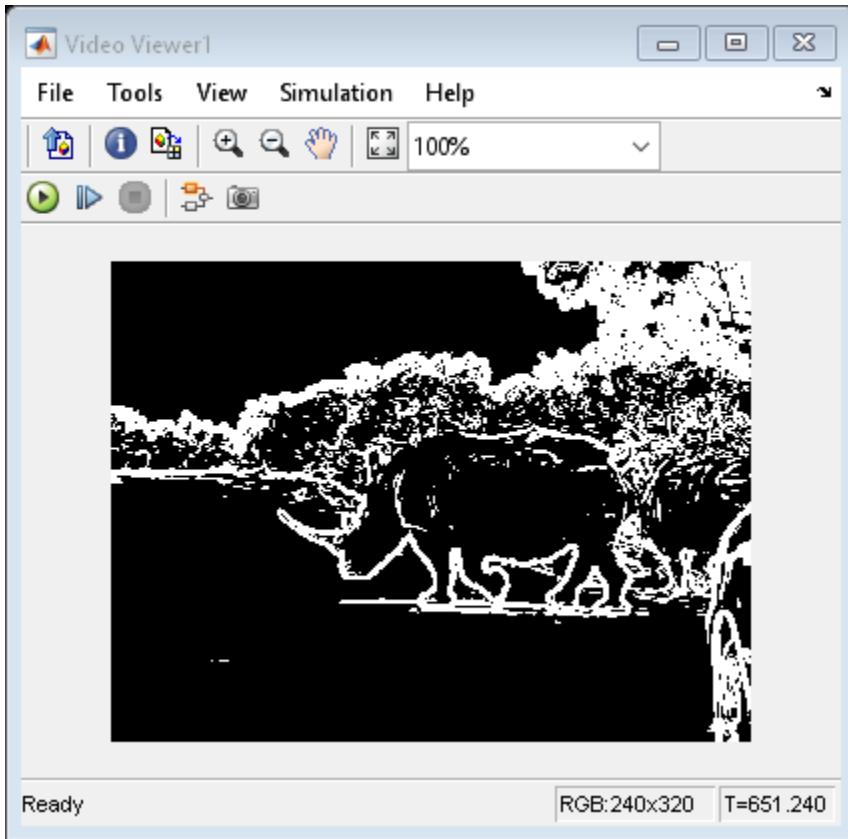
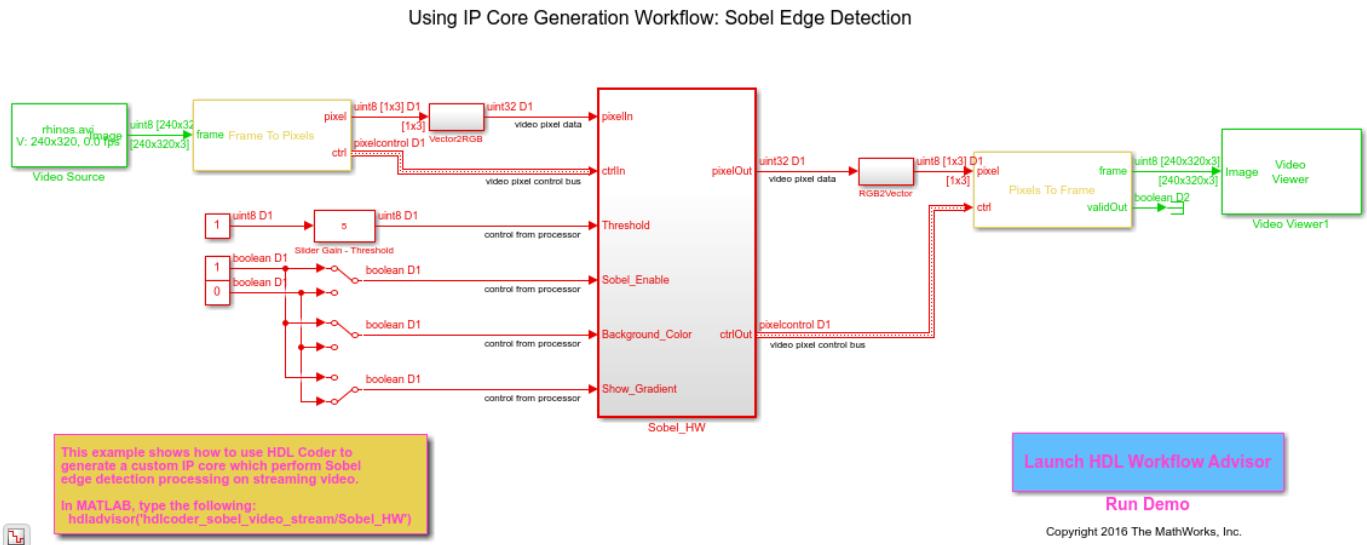
Model Video Streaming Algorithm using the Streaming Pixel Protocol

To deploy a simple Sobel edge detection algorithm on Zynq, the first step is to determine which part of the design to be run on FPGA, and which part of the design to be run on the ARM processor. In this example, we want to implement the edge detector on FPGA to process the incoming video stream in AXI4-Stream Video protocol. And we want to use the ARM processor to tune the parameters on FPGA to change the live video output.

In the example model, the DUT subsystem, **Sobel_HW**, uses a edge detector block to implement the Sobel edge detection algorithm. The video data and control signals are modeled in the video streaming pixel protocol, which is used by all the blocks in Vision HDL Toolbox. **pixelIn** and **pixelOut** are data ports for video streams. **ctrlIn** and **ctrlOut** are control ports for video streams. They are modeled using a bus data type (**Pixel Control Bus**) which contains following signals: **hStart**, **hEnd**, **vStart**, **vEnd**, **valid**.

Four input ports, **Threshold**, **Sobel_Enable**, **Background_Color** and **Show_Gradient**, are control ports to adjust the parameters the Sobel edge detection algorithms. You can use the **Slider Gain** or **Manual Switch** block to adjust the input values of these ports. After mapping these ports to AXI4-Lite interface, the ARM processor can control the generated IP core by writing to the generated AXI interface accessible registers.

```
modelname = 'hdlcoder_sobel_video_stream';
open_system(modelname);
sim(modelname);
```

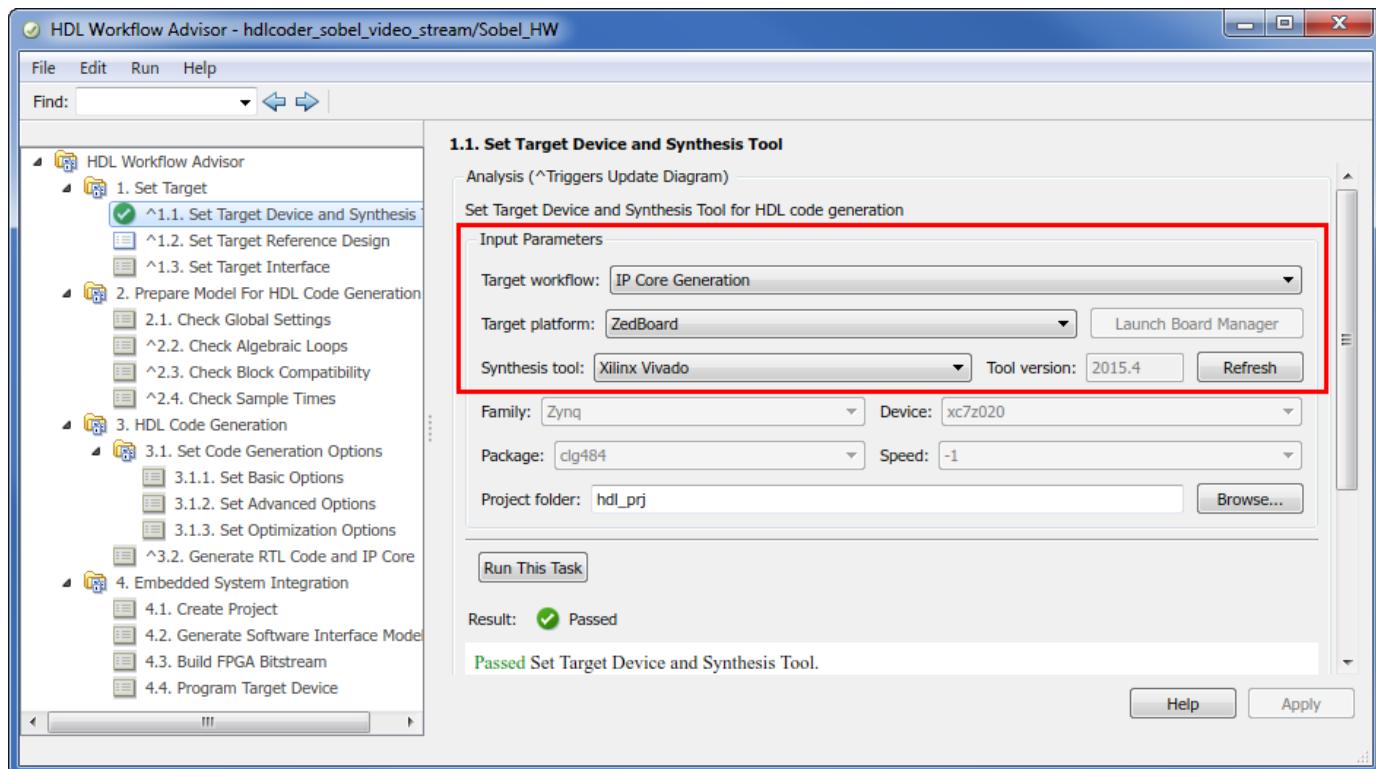


Generate HDL IP core with AXI4-Stream Video Interface

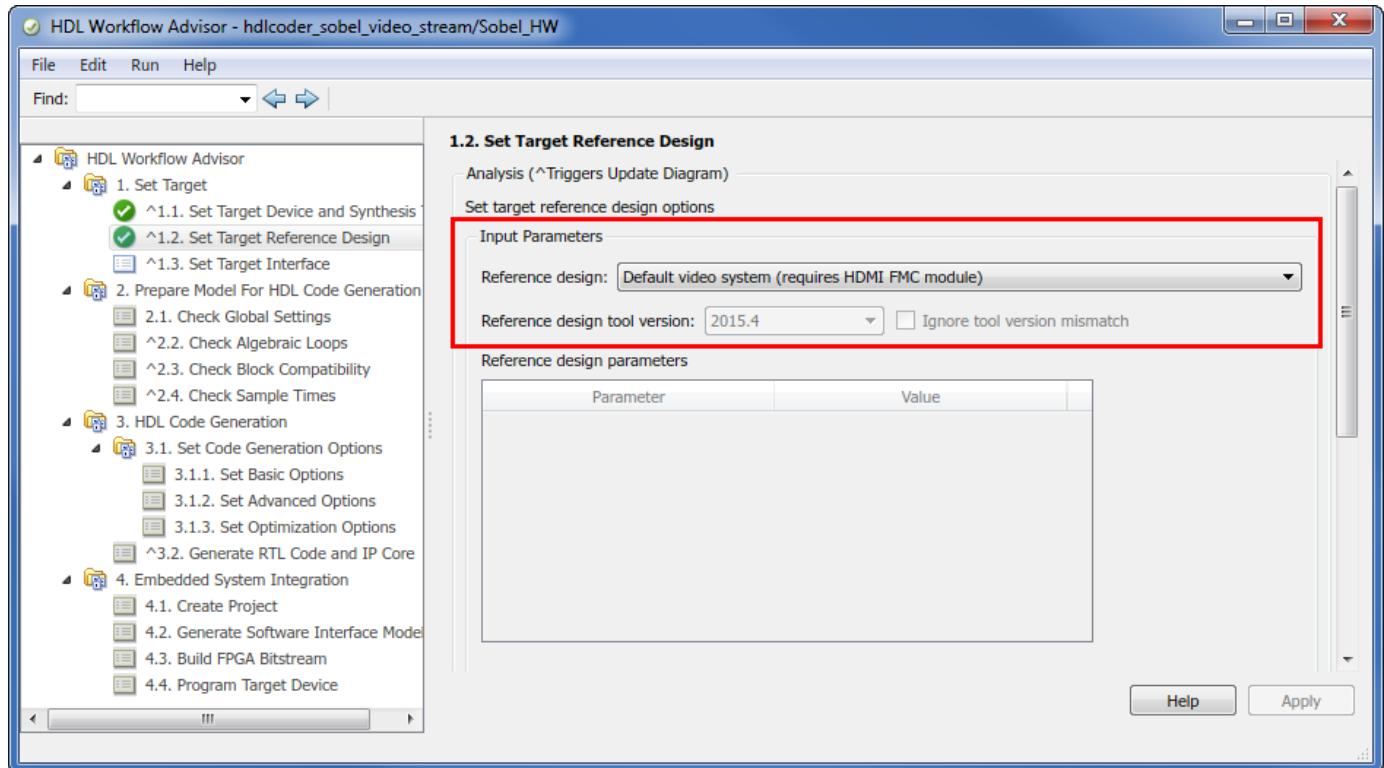
Next, we start the HDL Workflow Advisor and use the Zynq hardware-software co-design workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the "Getting Started with Targeting Xilinx Zynq Platform" on page 40-65 example.

1. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_sobel_video_stream/Sobel_HW`. The target interface settings are already saved in this example model, so the settings in Task 1.1 to 1.3 are automatically loaded. To learn more about saving target interface settings in the model, you can refer to the “Save Target Hardware Settings in Model” on page 40-137 example.

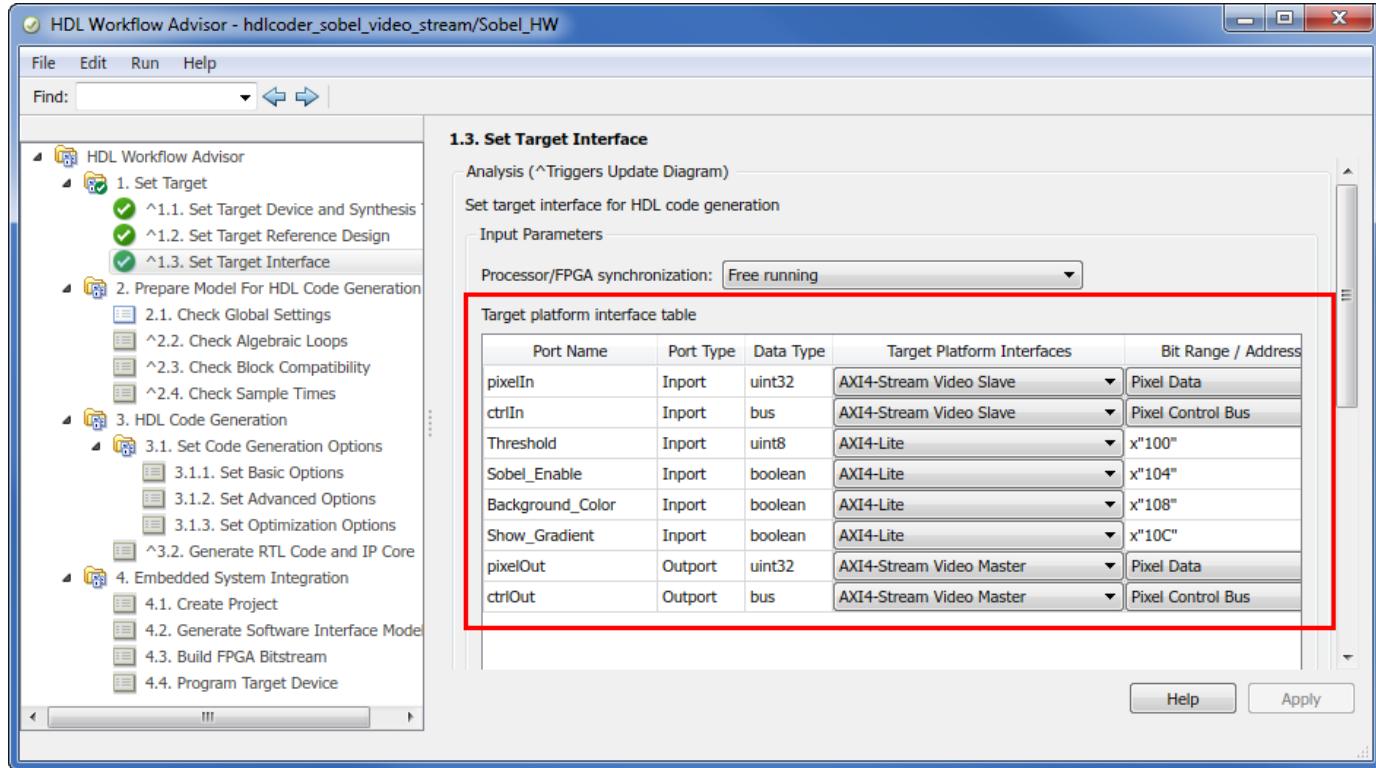
In Task 1.1, **IP Core Generation** is selected for **Target workflow**, and **ZedBoard** is selected for **Target platform**.



In Task 1.2, **Default video system (requires HDMI FMC module)** is selected for **Reference Design**.



In Task 1.3, the **Target platform interface table** is loaded as shown in the following picture. The video data stream ports, **pixelIn**, **ctrlIn**, **pixelOut**, and **ctrlOut**, are mapped to the AXI4-Stream Video interfaces, and the control parameter ports, such as **Threshold**, are mapped to the AXI4-Lite interface.



The AXI4-Stream Video interface communicates in master/slave mode, where the master device sends data to the slave device. Therefore, if a data port is an input port, assign it to an **AXI4-Stream Video Slave** interface, and if a data port is output port, assign it to an **AXI4-Stream Video Master** interface.

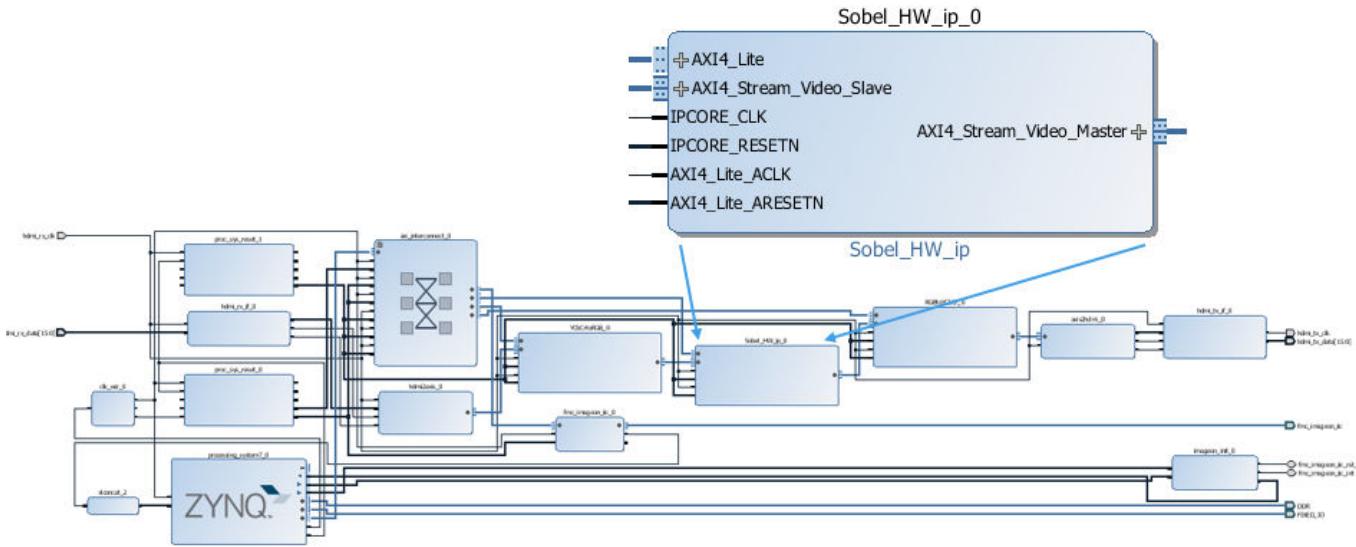
2. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

Integrate IP Into AXI4-Stream Video Compatible Reference Design

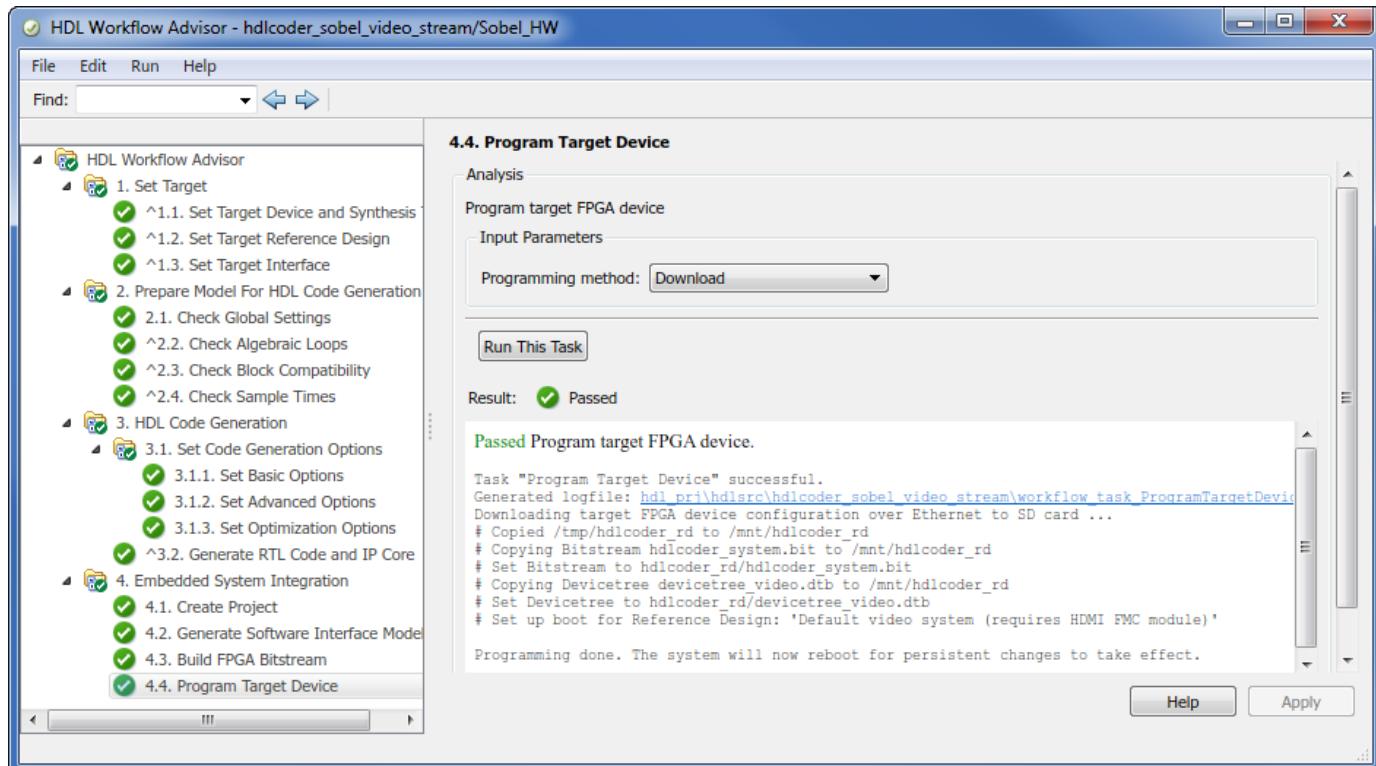
Next, in the HDL Workflow Advisor, we run the **Embedded System Integration** tasks to deploy the generated HDL IP core on Zynq hardware.

1. Run Task 4.1, **Create Project**. This task inserts the generated IP core into the **Default video system** reference design. As shown in the first diagram, this reference design contains the IPs to handle HDMI input and output interfaces. It also contains the IPs to do color space conversion from YCbCr to RGB. The generated project is a complete Zynq design, including the algorithm part (the generated DUT algorithm IP), and the platform part (the reference design).

2. Click the link in the Result pane to open the generated Vivado project. In the Vivado tool, click **Open Block Design** to view the Zynq design diagram, which includes the generated HDL IP core, other video pipelining IPs and the Zynq processor.



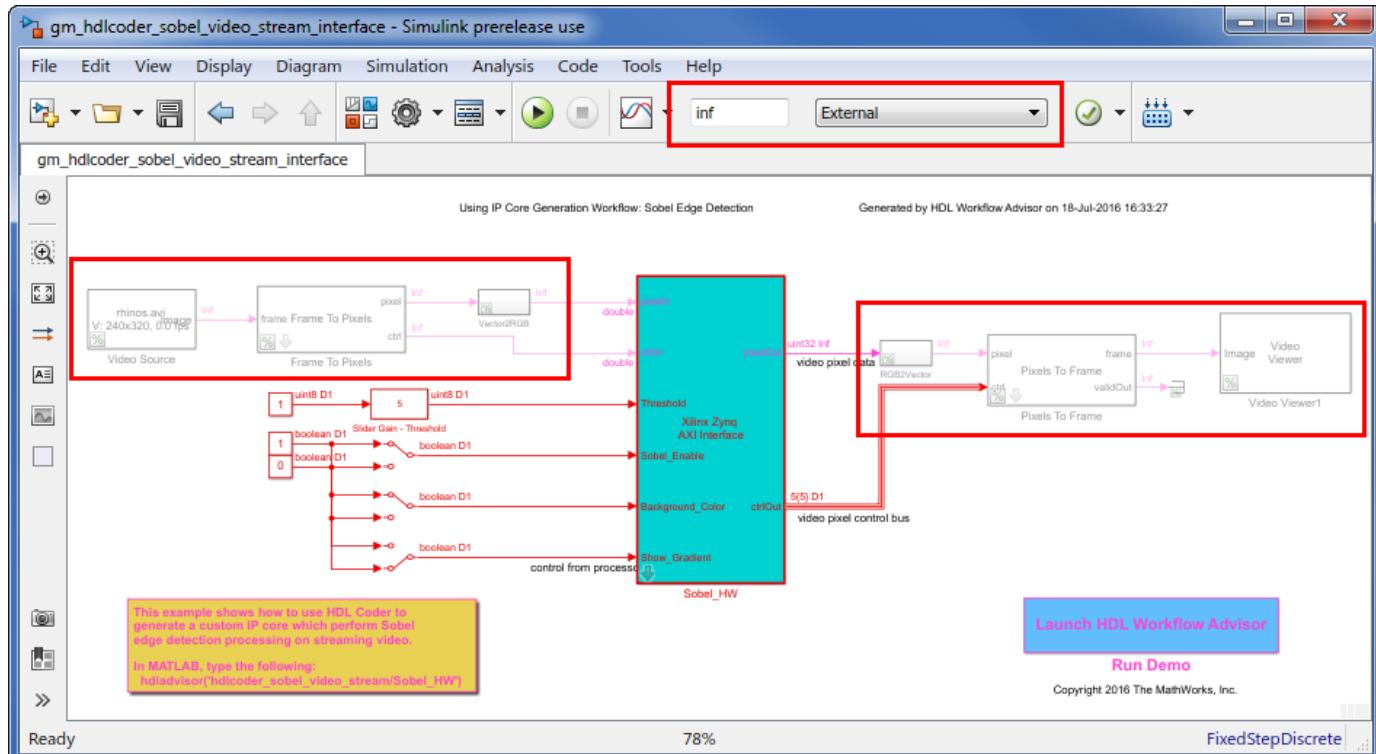
3. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream. Choose **Download** programming method in the task **Program Target Device** to download the FPGA bitstream onto the SD card on the ZedBoard, so your design will be automatically reloaded when you power cycle the ZedBoard.



Generate ARM executable to Tune Parameters on the FPGA Fabric

A software interface model is generated in Task 4.2, **Generate Software Interface Model**.

1. Before you generate code from the software interface model, comment out the **Video Source** and **Video Viewer** in the generated model, as shown in the following picture. These blocks do not need to be run on the ARM processor. The ARM processor is using AXI4-Lite interface to control the FPGA fabric. The actual video source and display interface are all running on the FPGA fabric. The video source comes from the HDMI input, and the video output will be sent to the monitor connected to the HDMI output.



2. Configure and build the software interface model for external mode:

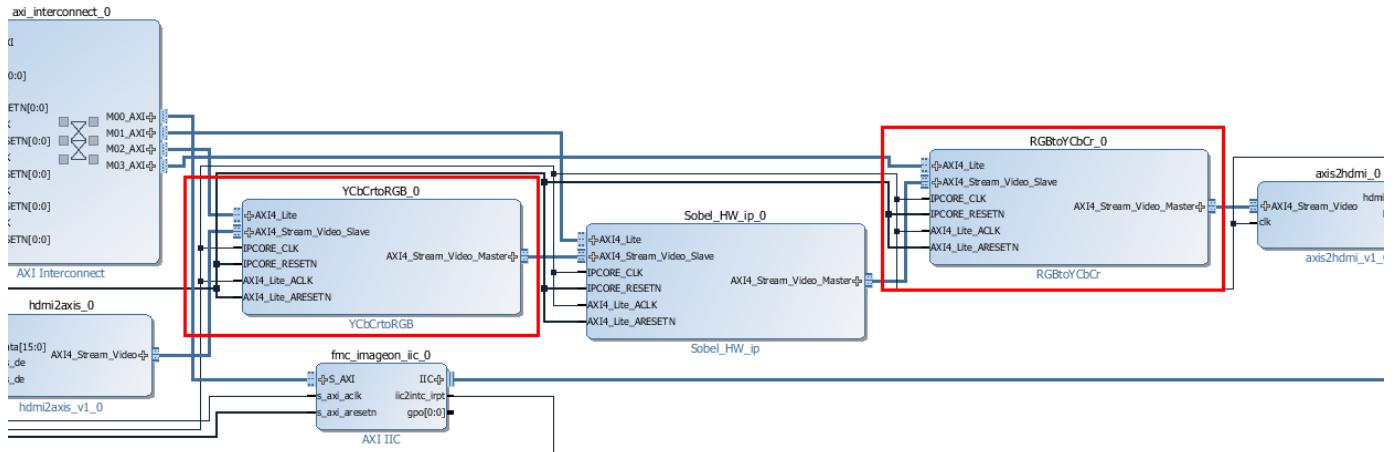
- 1 In the generated model, open the **Configuration Parameters** dialog box.
- 2 Select **Solver** and set "Stop Time" to "inf".
- 3 From the model menu, select **Simulation > Mode > External**.
- 4 Click the **Run** button on the model toolbar. Embedded Coder builds the model, downloads the ARM executable to the ZedBoard hardware, executes it, and connects the model to the executable running on the ZedBoard hardware.

3. Now, both the hardware and software parts of the design are running on Zynq hardware. Use the **Sobel_Enable** switch to observe that the live video output switches between the edge detector output and the original video. Use the **Threshold** or **Background_Color** switch to see the different edge detection effects on the live video output. These parameter values are sent to the Zynq hardware via external mode and the AXI4-Lite interface.

Customize your video reference design

You may want to extend the existing **Default video system** reference design to add additional pre-processing or post-processing camera pipelining IPs, or you may want to use a different SoC hardware or video camera interface. The **Default video system** reference design is an example or a starting point to create your own custom reference design.

For example, the **Default video system** reference design contains two IP cores to do color space conversion from YCbCr to RGB, as shown in following picture. These two IP cores are generated by HDL Coder as well using the IP Core Generation workflow. You can optionally generate other pre-processing or post-processing camera pipelining IP cores, and add them into a custom reference design to extend your video platform.



For more details on creating your own custom reference design, you can refer to the “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196 example.

Performing Large Matrix Operation on FPGA using External Memory

This example shows how to generate an HDL IP core with AXI4 Master interface, perform matrix multiplication in HDL IP core, and write output result to DDR memory.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- Xilinx Vivado Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Xilinx Zynq ZC706 Evaluation Kit
- HDL Coder Support Package for Xilinx Zynq Platform
- HDL Verifier Support Package for Xilinx FPGA Boards
- This example can also be run on a Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit

Introduction

In this example, you:

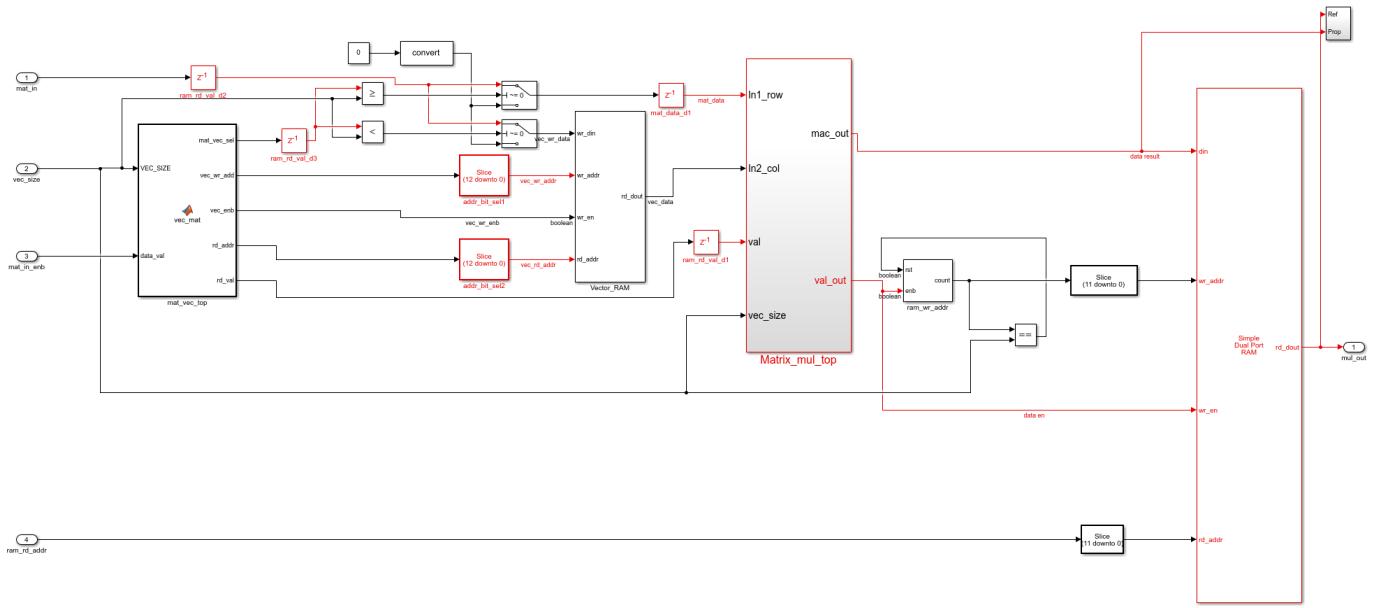
- 1 Generate an HDL IP core with AXI4 Master interface.
- 2 Access large matrices from the external DDR3 memory on the Xilinx Zynq ZC706 board using the AXI4 Master interface.
- 3 Perform matrix vector multiplication in the HDL IP core and write the output result back to the DDR memory using the AXI4 Master interface.

This example can also be run on a Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit, to access the external DDR4 memory.

This example models a matrix vector multiplication algorithm and implements the algorithm on the Xilinx Zynq FPGA board. Large matrices may not map efficiently to Block RAMs on the FPGA fabric. Instead, we can store the matrices in the external DDR memory on the FPGA board. The AXI4 Master interface can access the data by communicating with vendor-provided memory interface IP cores that interface with the DDR memory. This capability enables you to model algorithms that involve large data processing and requires high-throughput DDR access, such as matrix operations, computer vision algorithms, and so on.

The matrix vector multiplication module supports fixed point matrix vector multiplication, with a configurable matrix size ranging from 2 to 4000. The size of the matrix is run-time configurable through AXI4 accessible register.

```
modelname = 'hdlcoder_external_memory';
open_system(modelname);
```



Model Algorithm Using AXI4 Master Protocol

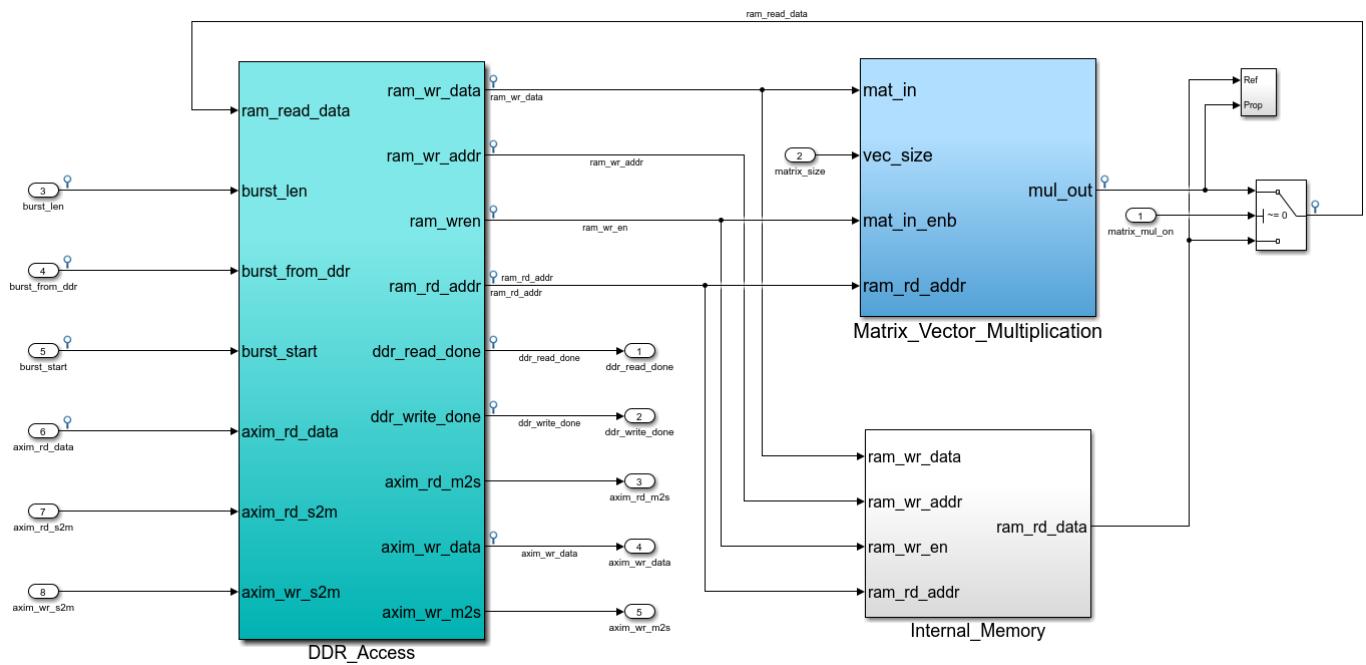
This example model includes the FPGA implementable DUT (Design under test) block, the DDR functional behavior block and the test environment to drive inputs and verify the expected outputs.

The DUT subsystem contains the AXI4 Master read/write controller along with the matrix vector multiplication module. Using the AXI4 Master interface, the DUT subsystem reads data from the external DDR memory, feed the data into the `Matrix_Vector_Multiplication` module, and then write the output data to the external DDR memory using AXI4 Master interface. The DUT module also has several parameter ports. These ports will be mapped to AXI4-Lite accessible registers, so you can adjust these parameters from MATLAB, even after you implement the design onto the FPGA.

The DDR module represents the external DDR memory in simulation environment. The interface between the DUT and DDR modules are the simplified AXI4 Master protocol.

One of the parameter port `matrix_mul_on` controls whether to run the `Matrix_Vector_Multiplication` module. When the input to `matrix_mul_on` is true, the DUT subsystem performs matrix vector multiplication as describe above. When the input to `matrix_mul_on` is false, the DUT subsystem perform a data loop back mode. In this mode, the DUT subsystem read data from the external DDR memory, write it into the `Internal_Memory` module, and then write the same data back to the external DDR memory. The data loop back mode is a simple way to verify the functionality of the AXI4 Master external DDR memory access.

```
open_system('hdlcoder_external_memory/DUT');
```



Inside the DUT subsystem, the **DDR_Access** module models the simplified AXI4 Master protocol, and use it to read and writes data on DDR. During the IP Core Generation workflow, HDL Coder will then generate the translator between the simplified AXI4 Master protocol and the actual AXI4 Master protocol in the generated HDL IP core. For more information on the simplified AXI4 Master protocol, refer to the “Model Design for AXI4 Master Interface Generation” on page 41-78 documentation.

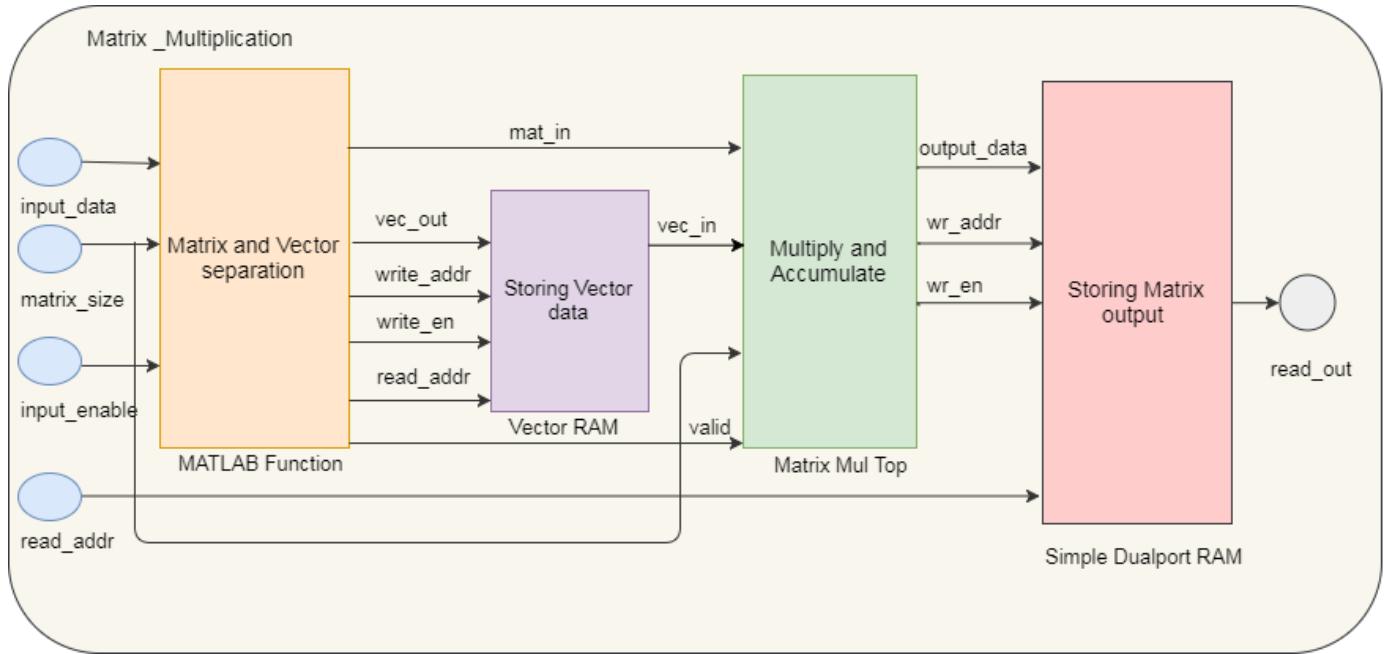
Also inside the DUT subsystem, the **Matrix_Vector_Multiplication** module uses a multiply-add block to implement a streaming dot-product computation for the inner-product of the matrix vector multiplication.

Lets say, A be a matrix of size NxN and B is a vector of size Nx1

Then, matrix vector multiplication output will be: $Z = A * B$, of size Nx1

The first N values from the DDR are treated as the Nx1 size vector, followed by NxN size matrix data. First N values (vector data) are stored into a RAM. From N+1 values onwards, data is directly streamed as matrix data. Vector data will be read from the **Vector_RAM** in parallel. Both matrix and vector inputs are fed into the **Matrix_mul_top** subsystem. The first matrix output is available after N clock cycles and will be stored into output RAM. Again, vector RAM read address is reinitialized to 0 and starts reading same vector data corresponding to new matrix stream. This operation is repeated for all the rows of the matrix.

The follow diagram shows the architecture of the **Matrix_Vector_Multiplication** module.



Functional Simulation in Simulink

You can simulate this example model, and verify the simulation result by running following script in MATLAB:

```
hdlcoder_external_memory_simulation;
PASSED: DDR initialization data matches.
PASSED: Matrix vector multiplication output matches with the expected data
```

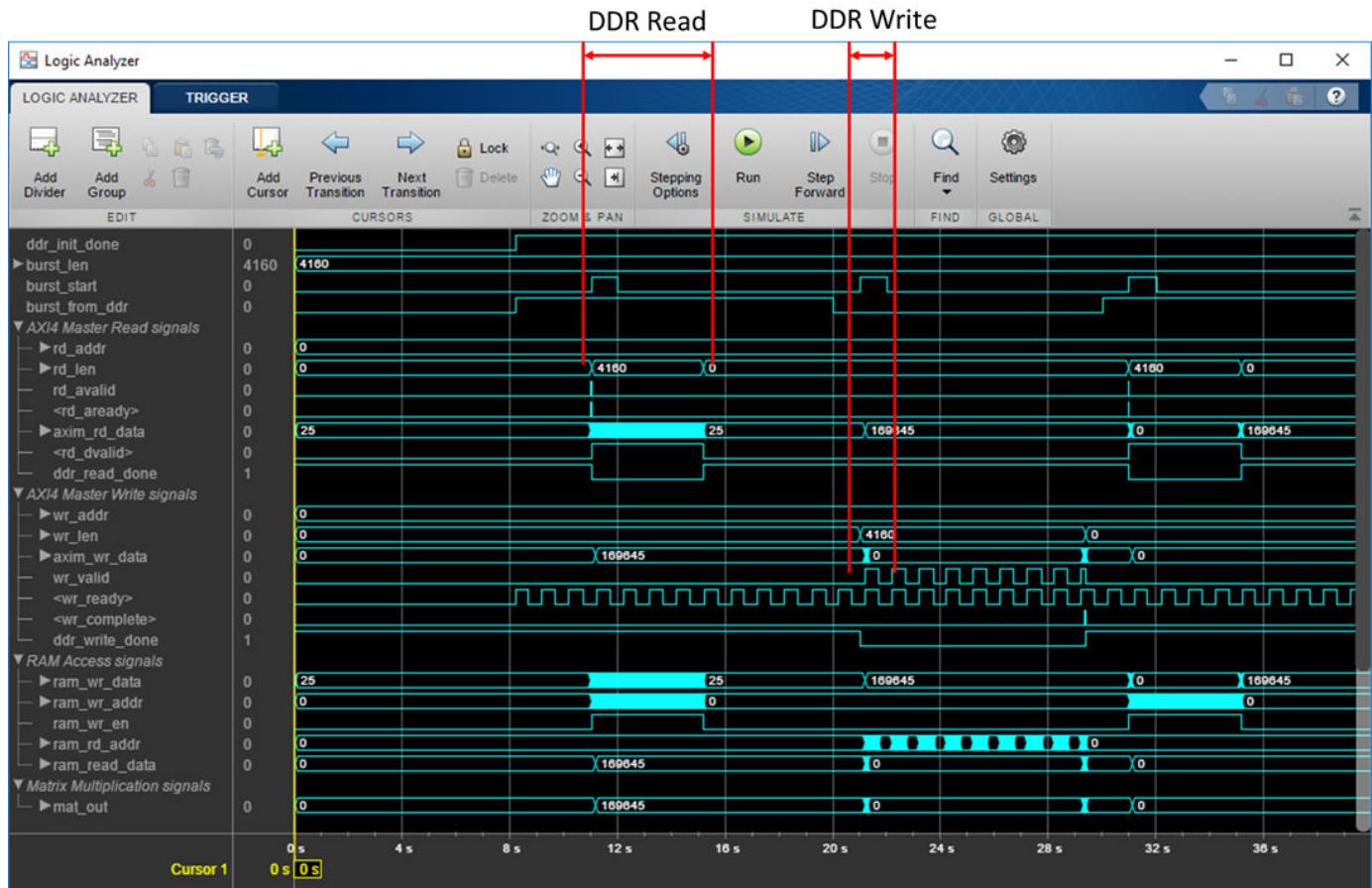
This script first initializes the parameters like `Matrix_Size`. By default the `Matrix_Size` is 64, which means a 64x64 matrix. The default `Matrix_Size` is kept small so the simulation is faster. After the DUT is implemented onto the FPGA board, larger `Matrix_Size` then can be used as the FPGA calculation is much faster. You can also adjust these parameters in the script.

The script then simulates the model, and verifies the result by comparing the logged simulation result with the expected value.

By default, the `Matrix_Multiplication_On` is true, the script verifies the matrix vector multiplication result.

When the `Matrix_Multiplication_On` is false, the script verifies the loop back mode, which means the DUT read `Burst_Length` amount of data from DDR, and write the data back to DDR.

If you have a DSP System Toolbox license, you can view the model signals over time using the Logic Analyzer.



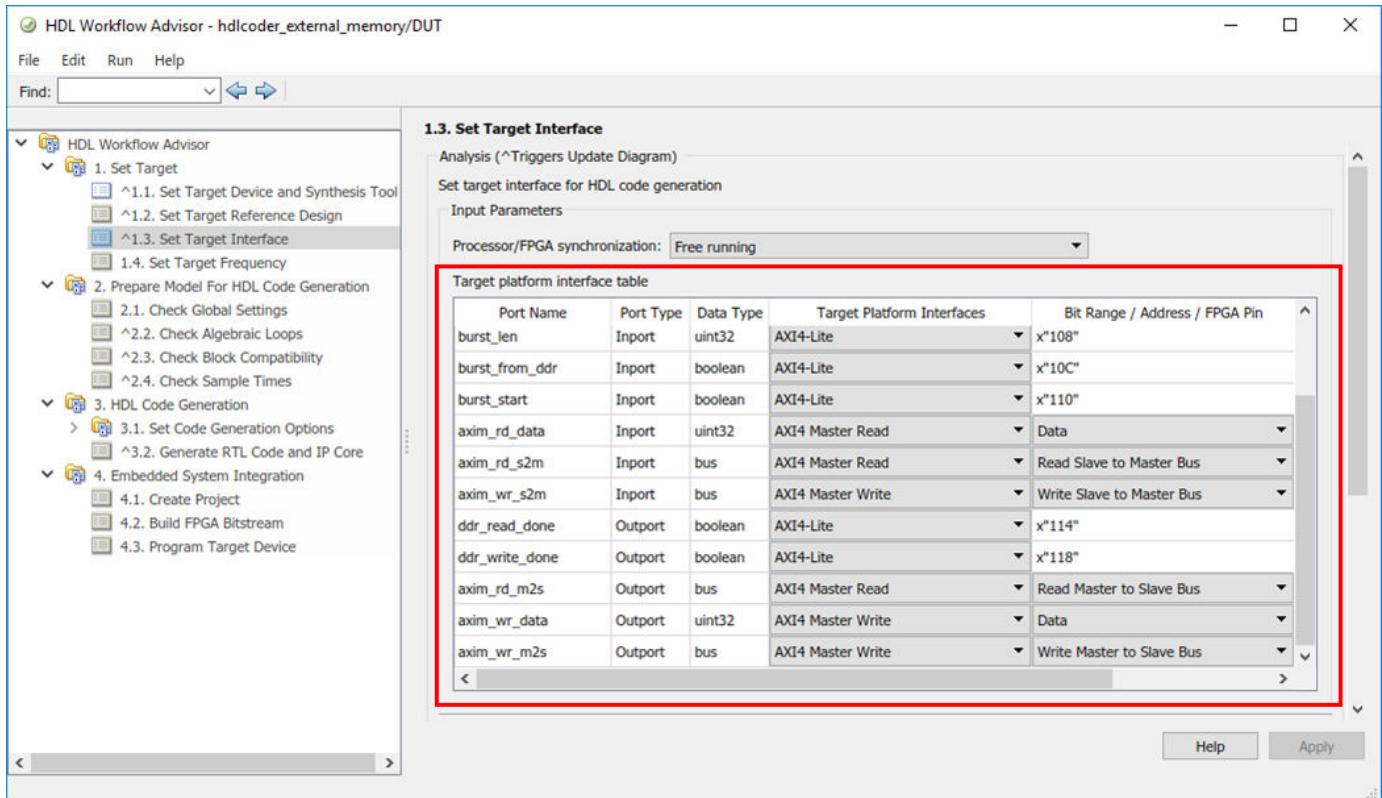
Generate HDL IP core with AXI4 Master Interface

Next, we start the HDL Workflow Advisor and use the IP Core Generation workflow to deploy this design on the Zynq hardware. For a more detailed step-by-step guide, you can refer to the “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example.

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2018.2\bin\vivado.hlp')
```

2. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_external_memory/DUT`. The target interface settings are saved on the model. Notice that **Target workflow** is IP Core Generation, **Target platform** is Xilinx Zynq ZC706 evaluation kit, **Reference Design** is Default System with External DDR3 memory access, and **Target platform interface table** settings are as shown below.



In this example, the input parameter ports like `matrix_mul_on`, `matrix_size`, `burst_len`, `burst_from_ddr` and `burst_start` are mapped to the AXI4-Lite interface. HDL Coder will generate AXI4 interface accessible registers for these ports. Later, you can use MATLAB to tune these parameters at run-time when the design is running on FPGA board.

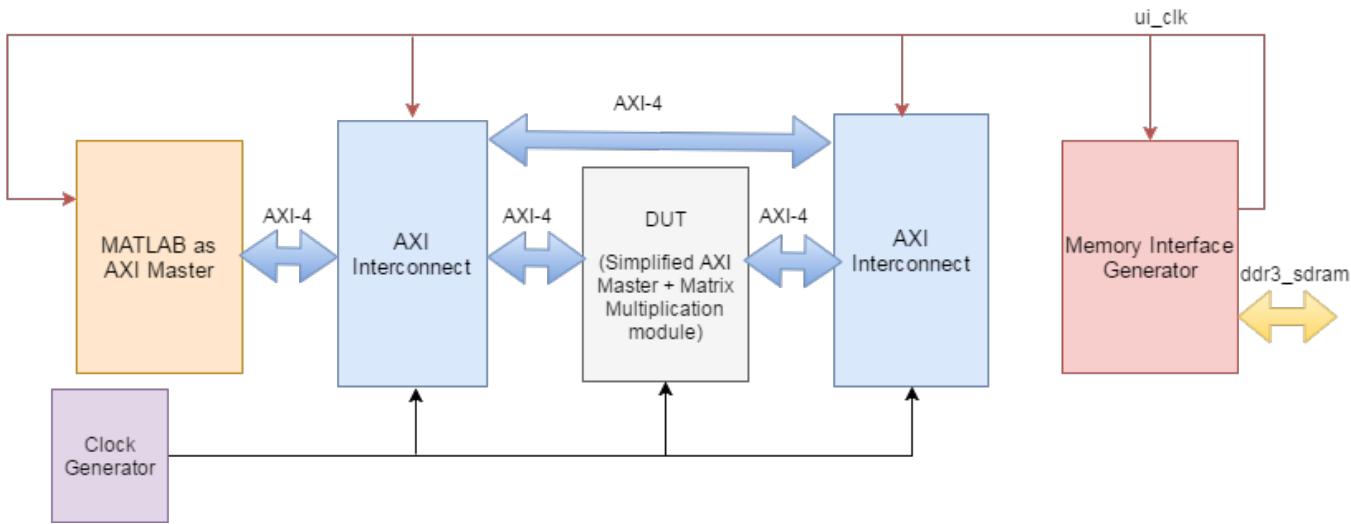
The AXI4 Master interface has separate Read and Write channels. The read channel ports like `axim_rd_data`, `axim_rd_s2m`, `axim_rd_m2s` are mapped to AXI4 Master Read interface. The write channel ports like `axim_wr_data`, `axim_wr_s2m`, `axim_wr_m2s` are mapped to AXI4 Master Write interface.

3. Right-click Task 3.2, **Generate RTL Code and IP Core**, and select **Run to Selected Task** to generate the IP core. You can find the register address mapping and other documentation for the IP core in the generated IP Core Report.

4. Now Right-click Task 4.2 **Build FPGA Bitstream**, and select **Run to Selected Task** to generate the Vivado project, and then build the FPGA bitstream.

During the project creation, the generated DUT IP core is integrated into the **Default System with External DDR3 Memory Access** reference design. This reference design comprises of a Xilinx Memory Interface Generator IP to communicate with the on-board external DDR3 memory on ZC706 platform. The MATLAB as AXI Master IP is also added to enable MATLAB to control the DUT IP, and to initialize and verify the DDR memory content.

You can click the link in the result window in Task 4.1 "Create Project" to view the generated Vivado project. If you open the Vivado block design, the generated reference design project looks similar to this architecture diagram.



Run FPGA Implementation on Xilinx Zynq ZC706 Evaluation Kit

After the FPGA bitstream is generated, you can run Task 4.3 **Program Target Device** to program the FPGA board through JTAG cable.

You can then run the FPGA implementation, and verify the hardware result by running following script in MATLAB

```
hdlcoder_external_memory_hw_run
```

This script first initializes the `Matrix_Size` to 500, which means a 500x500 matrix. You can adjust the `Matrix_Size` up to 4000.

The AXI4 Master Read and Write channel base addresses are then configured. These addresses defines the base address that DUT reads from, and writes to external DDR memory. In this script, the DUT is reading from base address '40000000', and write to base address '50000000'.

Then the MATLAB as AXI Master feature is used to initialize the external DDR3 memory with input vector and matrix data, and also clear the output DDR memory location.

Then the DUT calculation is started by controlling the AXI4-Lite accessible registers. The DUT IP core first read input data from the DDR memory, perform the matrix vector multiplication, and then write the result back to the DDR memory.

Finally, the output result is read back to MATLAB, and compared with the expected value. In this way, the hardware results are verified in MATLAB.

```
>> hdlcoder_external_memory_hw_run
Initializing external DDR3 memory (data size 250500) ...
Starting DUT IP core processing ...
Verifying result ...
PASSED: Matrix vector multiplication output matches with the expected data.
```

Accessing External DDR4 memory on Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit

1. Use the same model `hdlcoder_external_memory` to access external DDR4 memory on ZCU102 using HDL Coder IP core generation workflow.
2. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_external_memory/DUT`. In Task 1.1 Set **Target platform** as **Xilinx Zynq Ultrascale+ MPSoC ZCU102 Evaluation Kit** and in Task 1.2 set **Reference Design** as **Default System with External DDR4 Memory Access**
3. Now Right-click Task 4.2 **Build FPGA Bitstream**, and select **Run to Selected Task** to generate the Vivado project, and then build the FPGA bitstream.
4. You can run Task 4.3 **Program Target Device** to program the device and verify the hardware result by running following script in MATLAB:

```
hdlcoder_external_memory_hw_run_ZCU102
```

This script first initializes the `Matrix_Size` to 2000, which means a 2000x2000 matrix. In this script, the DUT is reading from base address '`80000000`', and write to base address '`90000000`'.

Finally, the output result is read back to MATLAB, and compared with the expected value. In this way, the hardware results are verified in MATLAB.

```
>> hdlcoder_external_memory_hw_run_ZCU102
Initializing external DDR4 memory (data size 4002000) ...
Starting DUT IP core processing ...
Verifying result ...
PASSED: Matrix vector multiplication output matches with the expected data.
```

Authoring a Reference Design for Audio System on a Zynq Board

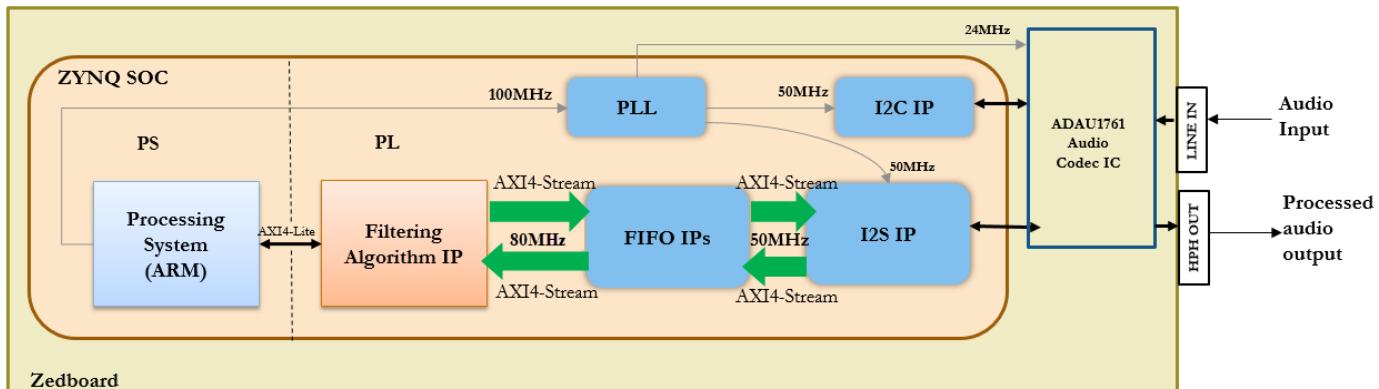
This example shows how to build a reference design to run an audio algorithm and access audio input and output on a Zynq® board.

Introduction

In this example you will create a reference design which receives audio input from Zedboard, performs some processing on it and transmits the processed audio data out of Zedboard. You also generate IP cores for peripheral interfaces using **HDL Workflow Advisor**.

To perform audio processing on Zedboard, you need the following 2 protocols:

- 1 I2C to configure the ADAU1761 audio codec chip on Zedboard.
- 2 I2S to stream the digitized audio data between the codec chip and Zynq fabric.



The above figure is a high level architecture diagram that shows how the reference design is used by the Filtering Algorithm IP. I2S IP operates at 50MHz frequency whereas one may want to run the Filtering Algorithm IP at a higher frequency. This frequency is controlled in Step **1.4** in **HDL Workflow Advisor**. In this example, assume that the filter operates at 80MHz. Since I2S and Filtering Algorithm IPs operate at different frequencies, we need FIFOs to handle clock domain crossing. Depending on the type of filter selected, Filtering Algorithm IP filters a range of frequencies from the incoming audio data and passes the filtered audio data out. In the above figure, the Filtering Algorithm IP is our main algorithm that we are modeling in Simulink. While FIFO IP is provided in Vivado, the I2C and I2S IPs need to be created. Here, the user has 3 choices: (a) use pre-packaged IP, e.g., if one exists, (b) model it in Simulink and generate an IP core using IP core generation workflow or (c) use legacy HDL code. To create an IP out of legacy HDL code, use a Simulink model to black box the HDL code and generate IP core from it. FIFO IPs handle the clock domain crossing between incoming audio data at 50MHz and the filter IP running at 80MHz. I2C, I2S, PLL, FIFO IPs and Processing System form a part of the reference design.

The following steps are used to create the reference design described above:

- 1 Generate IP Cores for peripheral interfaces
- 2 Create a custom audio codec reference design in Vivado
- 3 Create the reference design definition file

4 Verify the reference design

1. Generate IP Cores for peripheral interfaces using HDL Workflow Advisor

In this example,

- 1 I2C IP is developed by modeling it using Stateflow blocks, and also using legacy VHDL code for tristate buffer.
- 2 I2S IP is developed by modeling it in Simulink.

1.1 Creation of I2C IP

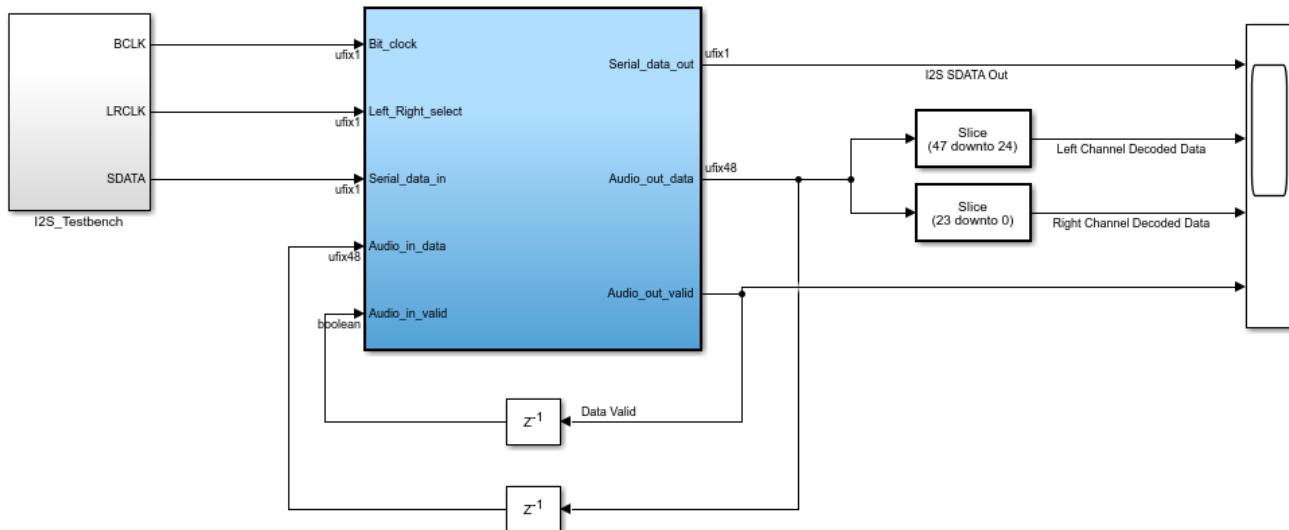
For creation of I2C IP to configure Audio Codec ADAU1761, refer to “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 41-96 article.

1.2 Creation of I2S IP

Design a model in Simulink with a matlab function which implements the I2S protocol.

```
modelname = 'hdlcoder_I2S_adau1761';
open_system(modelname);
```

Using IP Core Generation Workflow: I2S IP generation



This example shows how to use HDL Workflow Advisor to generate a custom IP core for implementing I2S protocol

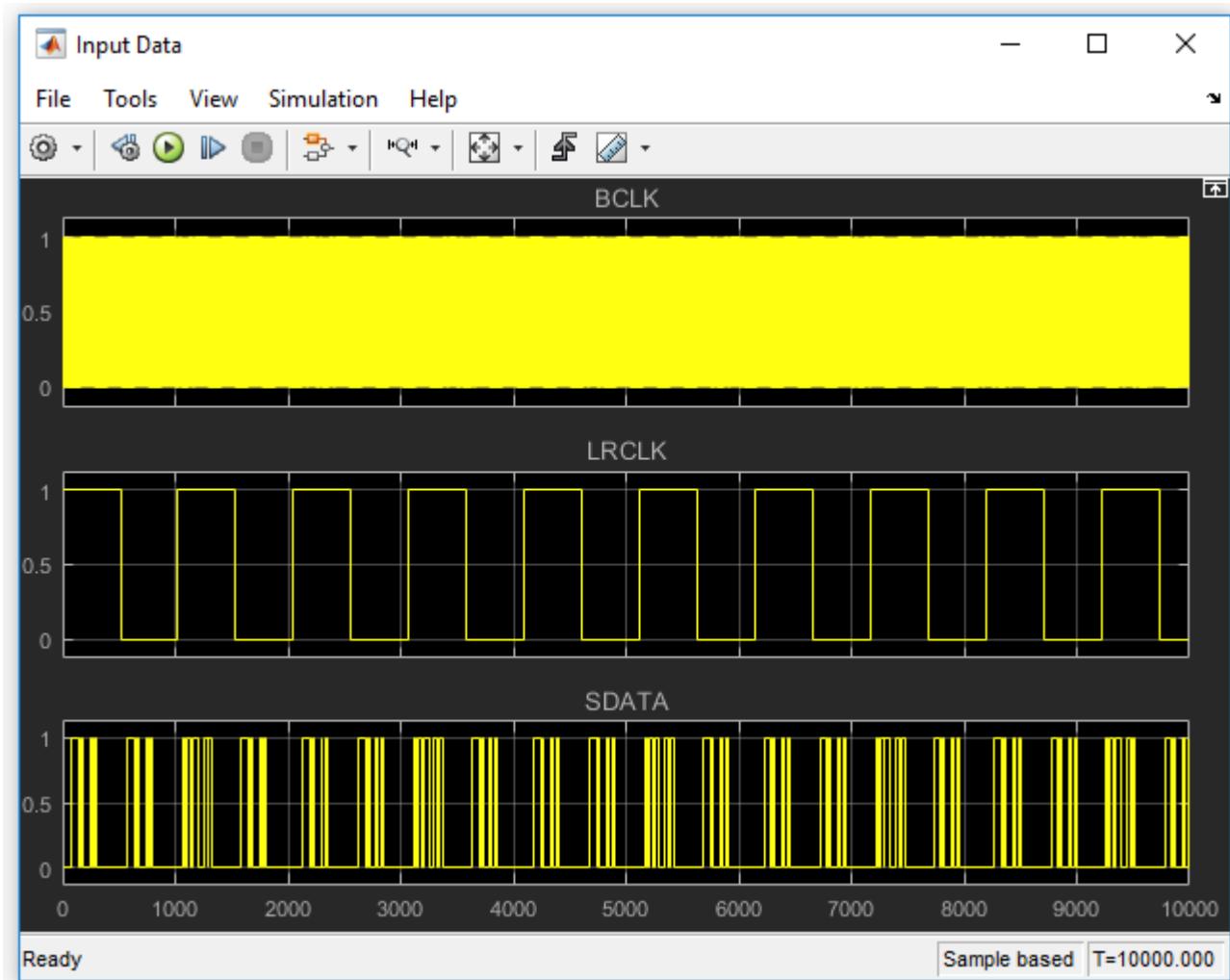
In MATLAB, type the following:
`hdladvisor('hdlcoder_I2S_adau1761/Subsystem')`

[Launch HDL Workflow Advisor](#)

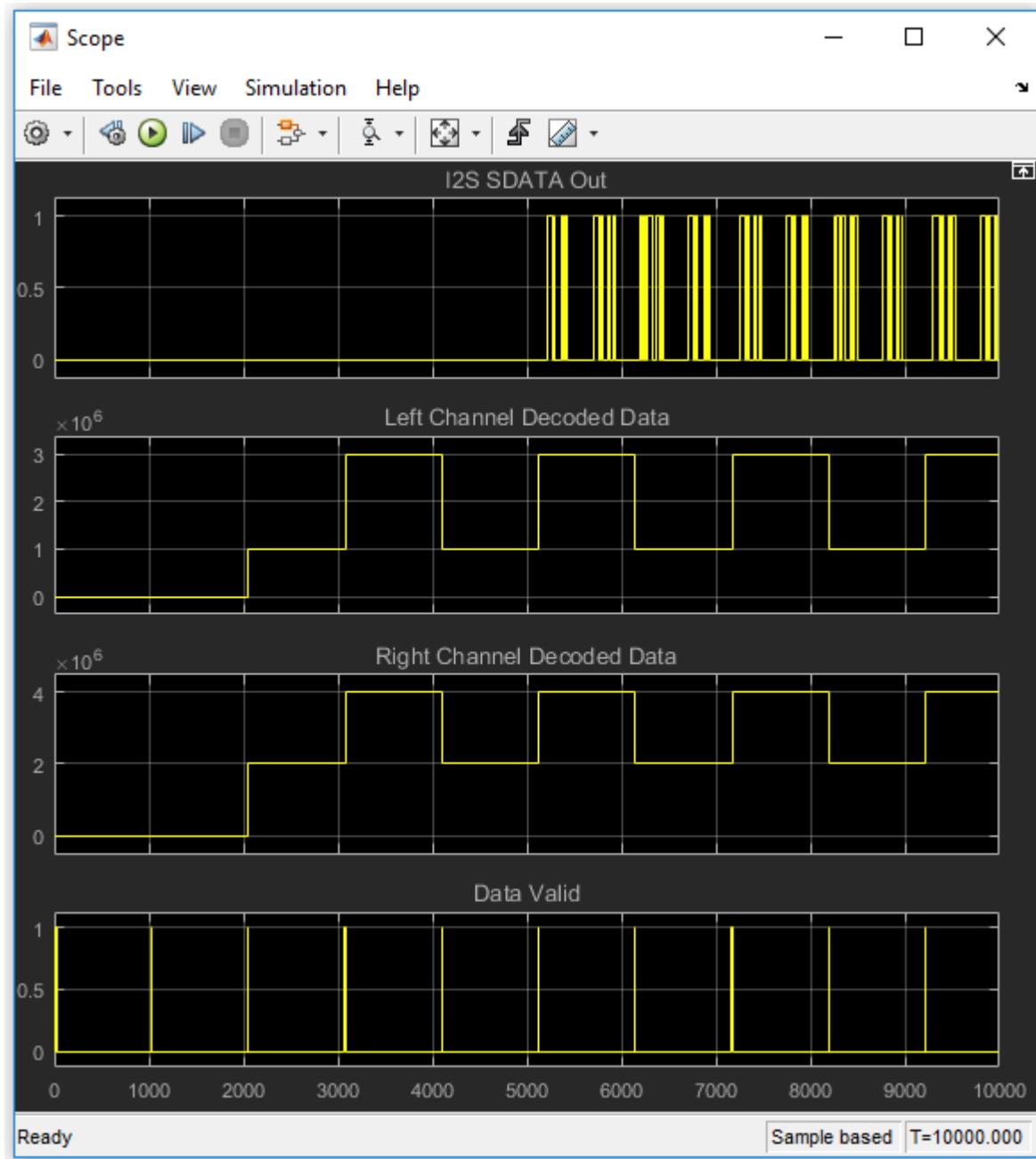
[Run Demo](#)

Copyright 2016 The MathWorks, Inc.

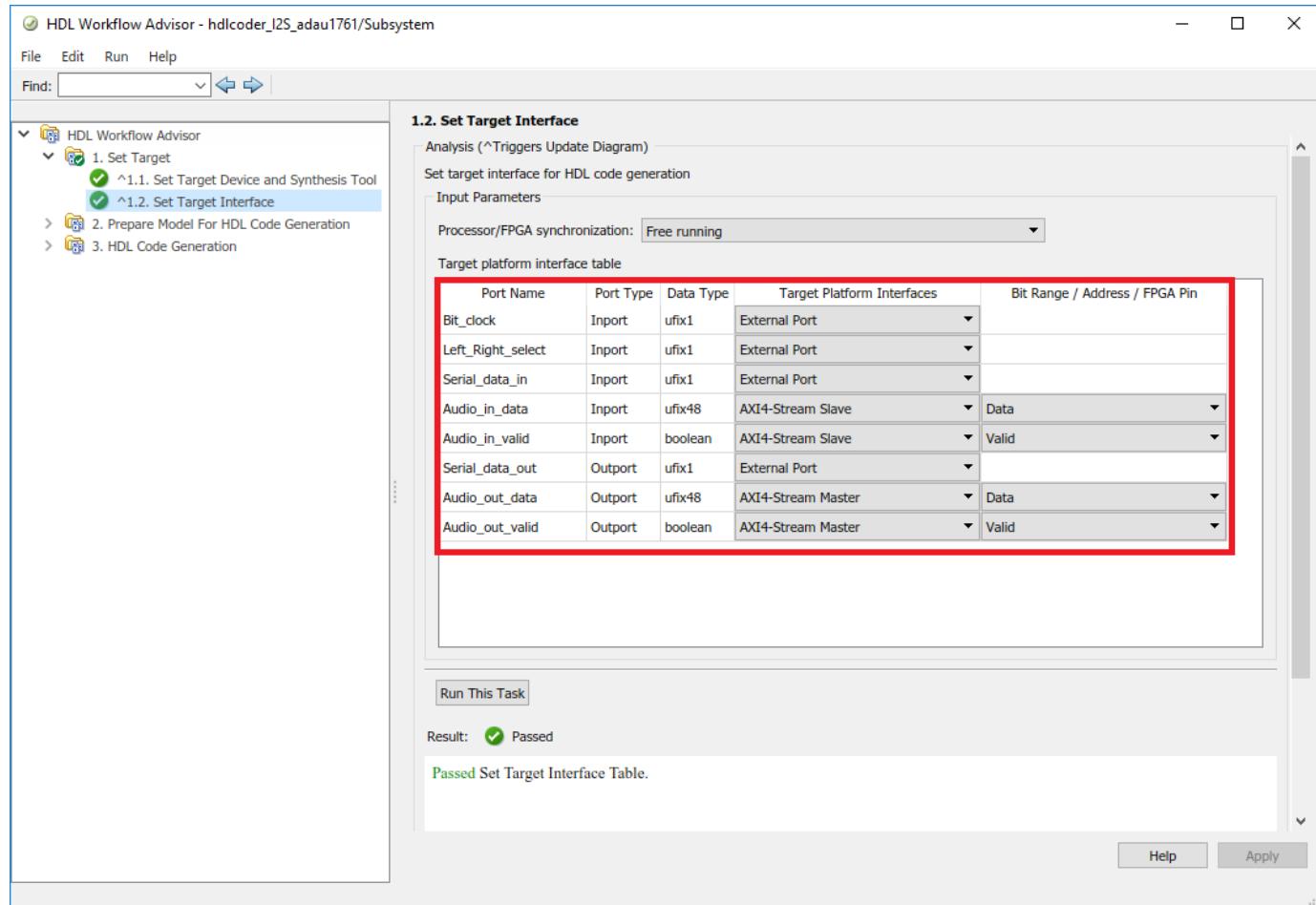
Create a test bench in the model to mimic the incoming audio data from the codec.



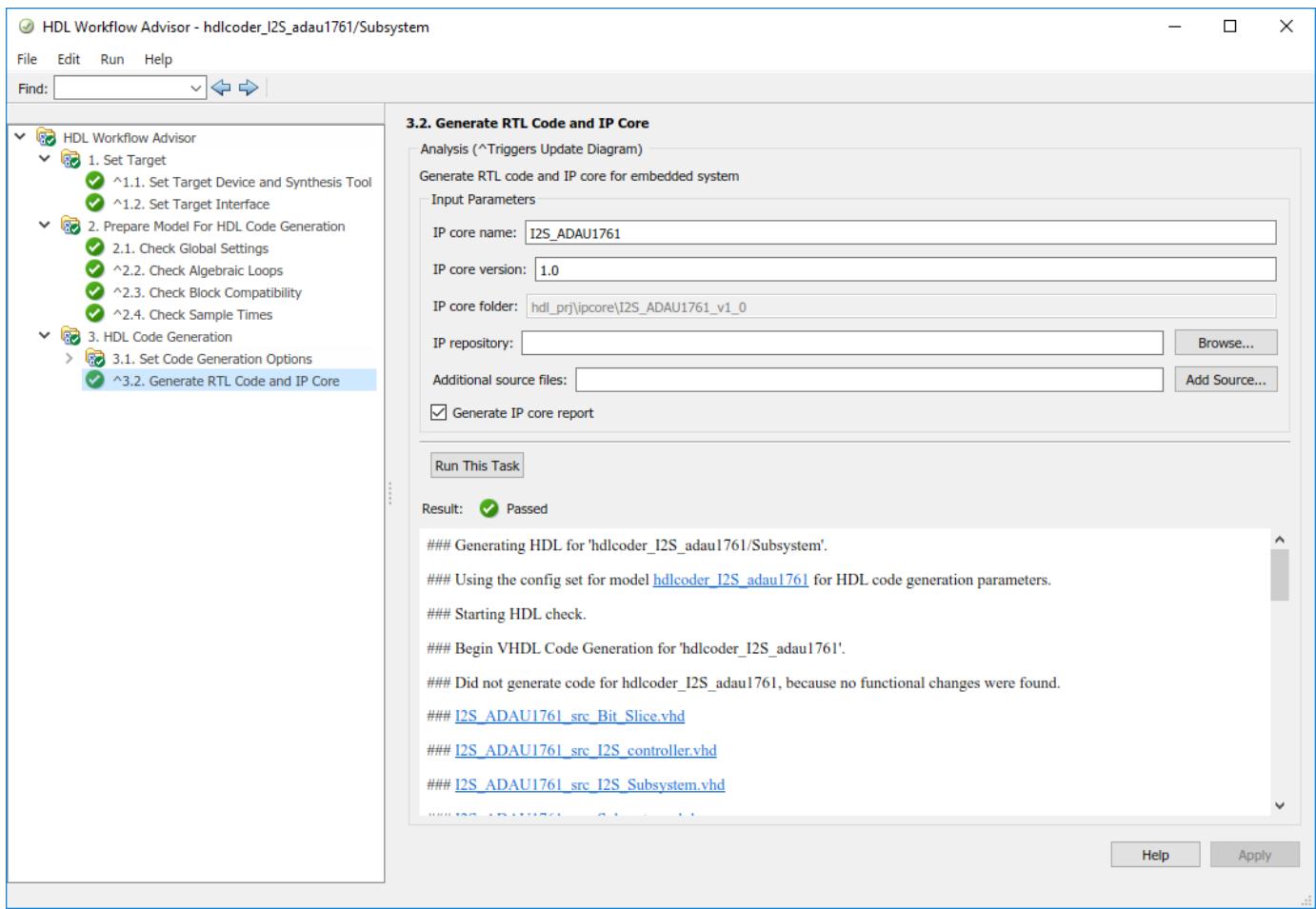
Feed this data to the Subsystem block which does the I2S operation. Verify the output of the Subsystem on a Scope.



Start the HDL Workflow Advisor from the DUT subsystem. In Task 1.1, keep the same settings as those of I2C IP generated earlier. in Task 1.2, set the Target Platform Interfaces as shown below:



Run Task 3.2 and generate the ip core.



2. Create a custom audio codec reference design in Vivado

I2C, I2S and FIFO IPs are incorporated in the custom reference design. To create a custom reference design, refer to the "Create and export a custom reference design using Xilinx Vivado" section in "Define Custom Board and Reference Design for Zynq Workflow" on page 41-196.

Key points to be noted while creating this custom reference design:

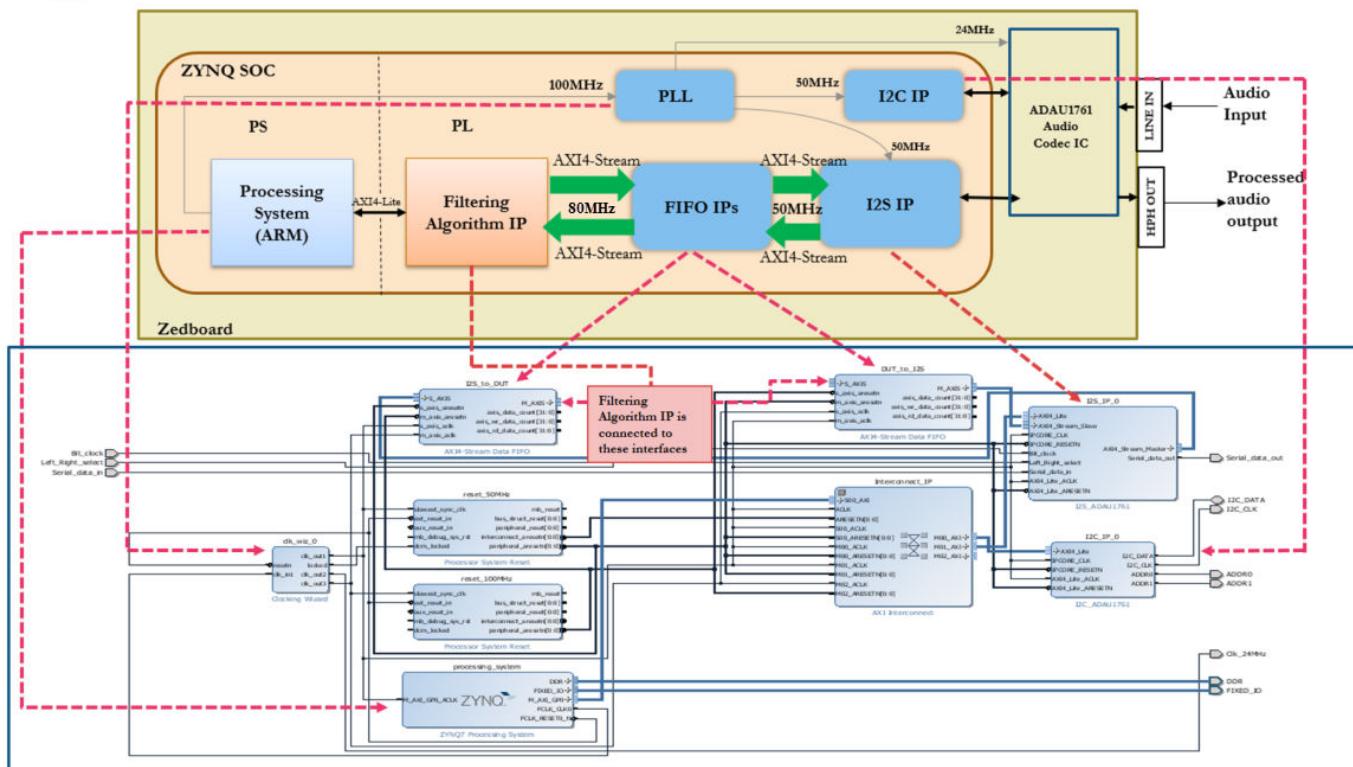
- 1 We must understand the theory of operation of the audio codec chip on the Zedboard.
- 2 The FIFOs are set to default values for their configuration.
- 3 For the IP cores generated using HDL Workflow Advisor, **IPCORE_CLK** and **AXI4_Lite_ACLK** should be connected to the same clock source.
- 4 On validating the block design in Vivado, there should be no critical warnings except for the unconnected ports.
- 5 In this reference design, the audio codec is configured to operate in the Master mode.

The following signals run between the reference design on Zynq Soc and the audio codec on Zedboard:

- 1 **Bit_clock** is the product of the sampling frequency, the number of bits per channel and the number of channels. It is driven by the audio codec in master mode. In this example, Sampling frequency is 48KHz, No of channels is 2, Number of bits per channel is 24.

- 2 **Left_right_select** is to distinguish between left audio channel data and right audio channel data. It is in sync with the Bit clock.
- 3 **Serial_data_in** is the analog to digital converted audio data from the codec.
- 4 **Serial_data_out** is the digital audio data going to codec to be converted into analog form.
- 5 **I2C_CLK** and **I2C_DATA** are standard I2C signals
- 6 **ADDR0** and **ADDR1** are the I2C Address Bits.
- 7 **Clk_24MHz** is the 24MHz clock signal required by the codec.

The custom audio codec reference design created for this example is shown below:



3. Create the reference design definition file

The following code describes the contents of the Zedboard reference design definition file **plugin_rd.m** for the above reference design. For more details on how to define and register custom board, refer to “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196 example.

```

function hRD = plugin_rd()
% Reference design definition

% Copyright 2016-2018 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Audio System with AXI4 Stream Interface';
hRD.BoardName = 'ZedBoard';

%% Tool information
hRD.SupportedToolVersion = {'2017.2','2017.4'};
```

%% Add custom design files

```
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'em.avnet.com:zed:part0:1.0');
```

```
hRD.addIPRepository(... ...
    'IPLFunction', 'mathworks.hdlcoderdemo.vivado.hdlcoderdemo_adau1761_iplist');
```

% Add constraint files

```
hRD.CustomConstraints = {'Audio_Filter_Demo.xdc'};
```

%% Add interfaces

% add clock interface

```
hRD.addClockInterface( ...
    'ClockConnection',      'clk_wiz_0/clk_out3', ...
    'ResetConnection',      'reset_100MHz/peripheral_aresetn',...
    'DefaultFrequencyMHz', 80, ...
    'MinFrequencyMHz',     5, ...
    'MaxFrequencyMHz',     500, ...
    'ClockModuleInstance', 'clk_wiz_0',...
    'ClockNumber',         3);
```

% add AXI4 and AXI4-Lite slave interfaces

```
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'Interconnect_IP/M02_AXI', ...
    'BaseAddress',          '0x400D0000', ...
    'MasterAddressSpace',   'processing_system/Data');
```

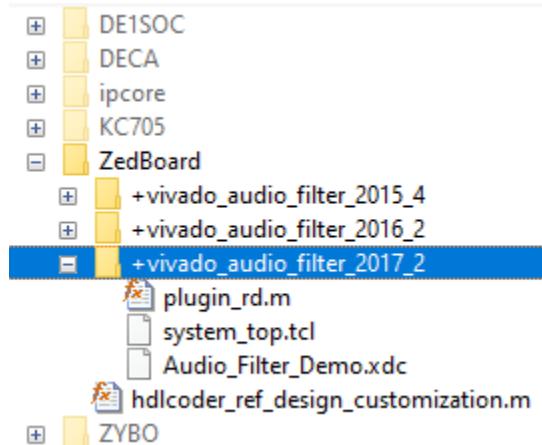
% add AXI4-Stream interface

```
hRD.addAXI4StreamInterface( ...
    'MasterChannelNumber', 1, ...
    'SlaveChannelNumber', 1, ...
    'MasterChannelConnection', 'DUT_to_I2S/S_AXIS', ...
    'SlaveChannelConnection', 'I2S_to_DUT/M_AXIS', ...
    'MasterChannelDataWidth', 48, ...
    'SlaveChannelDataWidth', 48 ...)
```

Go to **Zedboard** folder using the following command:

```
cd ([matlabroot '/toolbox/hdlcoder/hdlcoderdemos/customboards/Zedboard']);
```

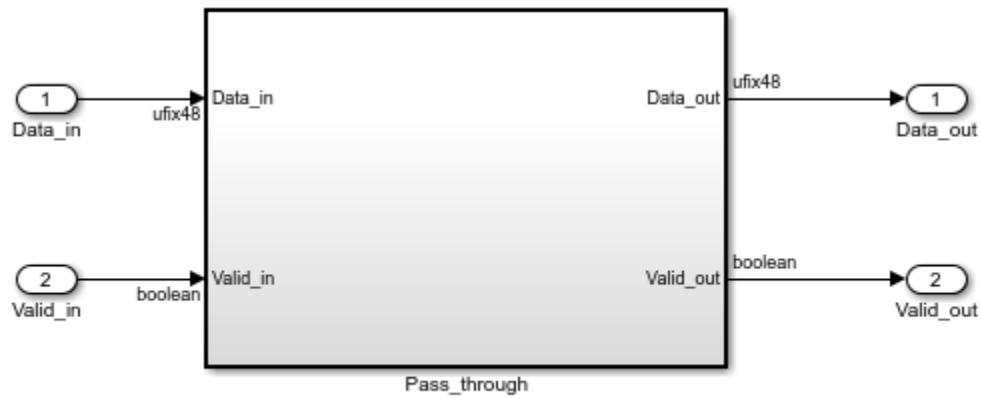
All files that are required for the reference design such as IP core files, XDC files, plugin_rd file etc should be added to the matlab path, inside **Zedboard** folder using the hierarchy shown below. The user generated IP core files should be in **+vivado** folder. plugin_rd.m, tcl files and xdc files should be in the reference design plugin folder, for example, **+vivado_audio_filter_2017_2** folder.



4. Verify the reference design

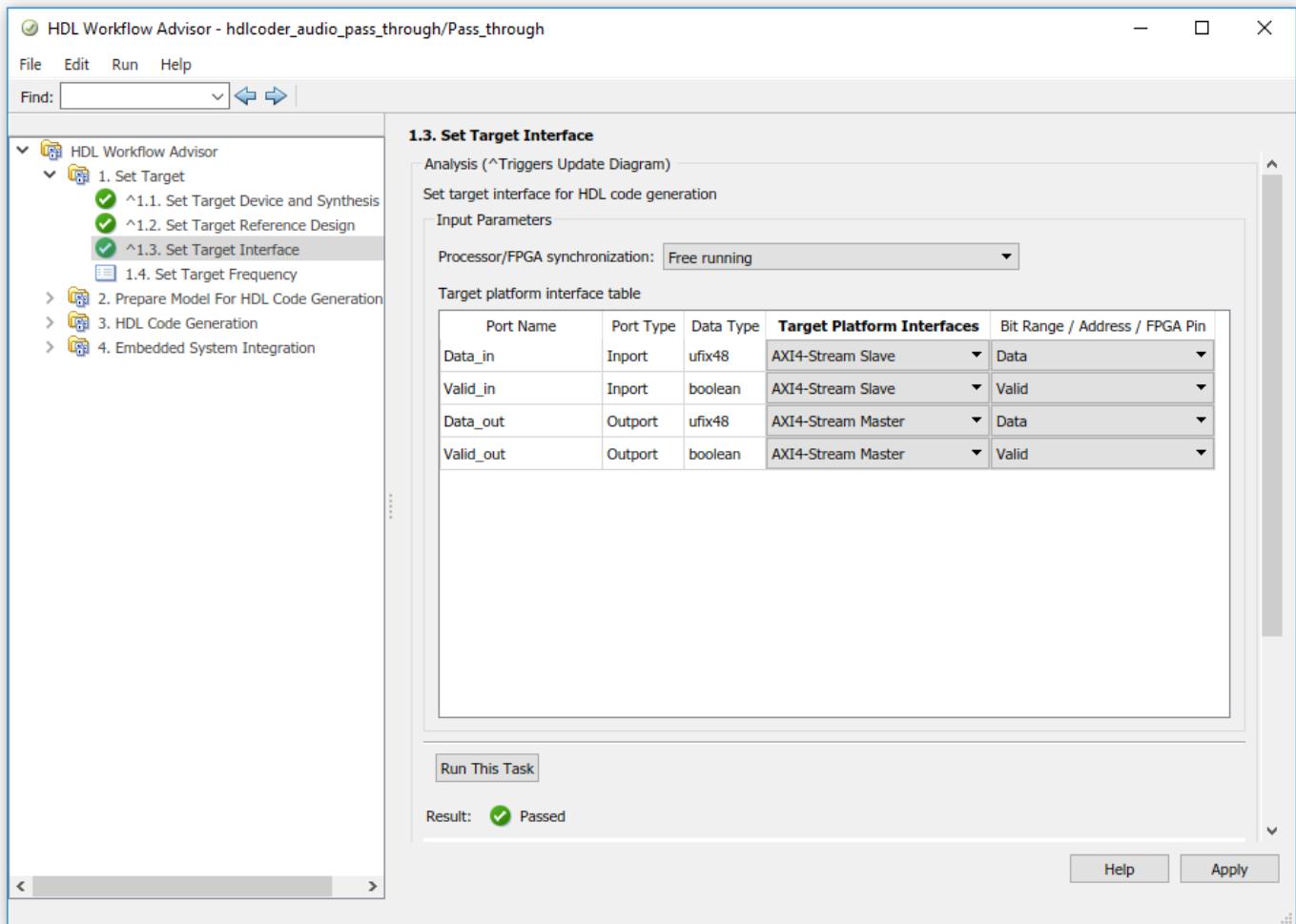
In order to ensure that the reference design and the interfaces in the reference design work as expected, design a Simulink model which just sends the audio through the Algorithm IP, integrate it with the reference design and test it on Zedboard. The user should be able to hear the audio loop back.

```
modelname = 'hdlcoder_audio_pass_through';
open_system(modelname);
```



Copyright 2016 The MathWorks, Inc.

The interfaces in the model should be selected as shown below:



To generate IP core from a model and integrate it with the audio codec reference design, refer to “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 41-126 example.

Authoring a Reference Design for Audio System on a ZYBO Board

This example shows how to build a reference design to run an audio algorithm and access audio input and output on ZYBO board.

Before You Begin

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform
- Xilinx Vivado, with latest version mentioned in the documentation
- Digilent® Zybo Zynq™ development board with the accessory kit

Note: This example uses Digilent® Zybo Zynq-7000 ARM/FPGA SoC trainer board. This example does not work on Digilent® Zybo Z7: Zynq-7000 ARM/FPGA SoC development board which have two variants Zybo Z7-10 and Zybo Z7-20.

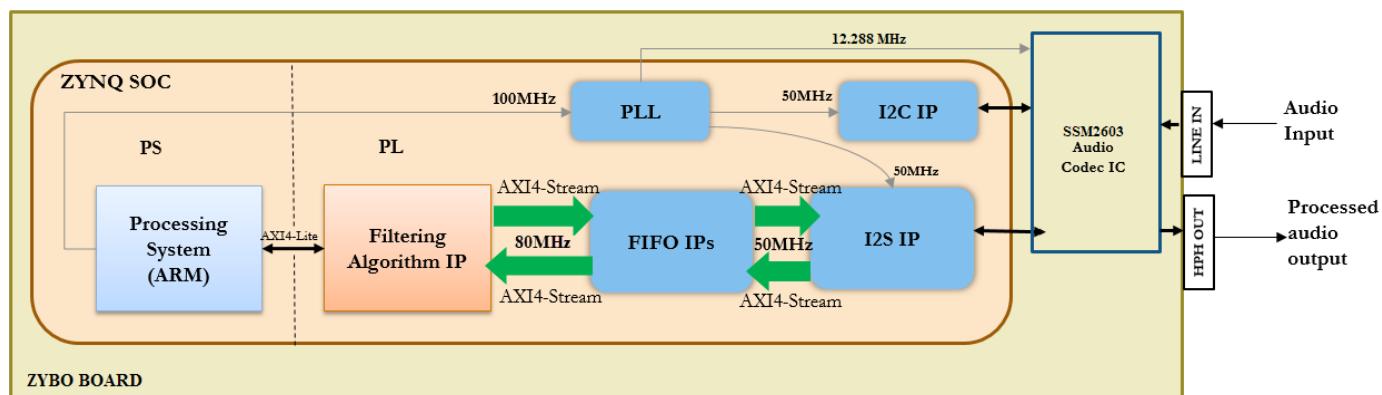
To setup the ZYBO board, refer to the *Set up the Zybo board* section in the “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196 article.

Introduction

In this example you will create a reference design which receives audio input from ZYBO board, performs some processing on it and transmits the processed audio data out of ZYBO board. You also generate IP cores for peripheral interfaces using **HDL Workflow Advisor**.

To perform audio processing on ZYBO board, following 2 protocols are needed:

- 1 I2C to configure the SSM2603 audio codec chip on ZYBO board.
- 2 I2S to stream the digitized audio data between the codec chip and zynq fabric.



The above figure is a high level architecture diagram that shows how the reference design is used by the Filtering Algorithm IP on ZYBO board. This example is similar to the audio system reference design for Zedboard except that the ZYBO board uses a SSM2603 audio codec chip where as the Zedboard uses ADAU1761 audio codec chip. Rest of the operating parameters are same as the Audio System reference design for Zedboard. For more details, please refer to “Authoring a Reference Design for Audio System on a Zynq Board” on page 41-170 example.

The following steps are used to create the reference design described above:

- 1** Generate IP Cores for peripheral interfaces
- 2** Create a custom audio codec reference design in Vivado
- 3** Create the reference design definition file
- 4** Verify the reference design

1. Generate IP Cores for peripheral interfaces using HDL Workflow Advisor

In this example,

- 1** I2C IP is developed using stateflow blocks & legacy VHDL code for tristate buffer.
- 2** I2S IP is developed by modelling it in Simulink.

1.1 Creation of I2C IP

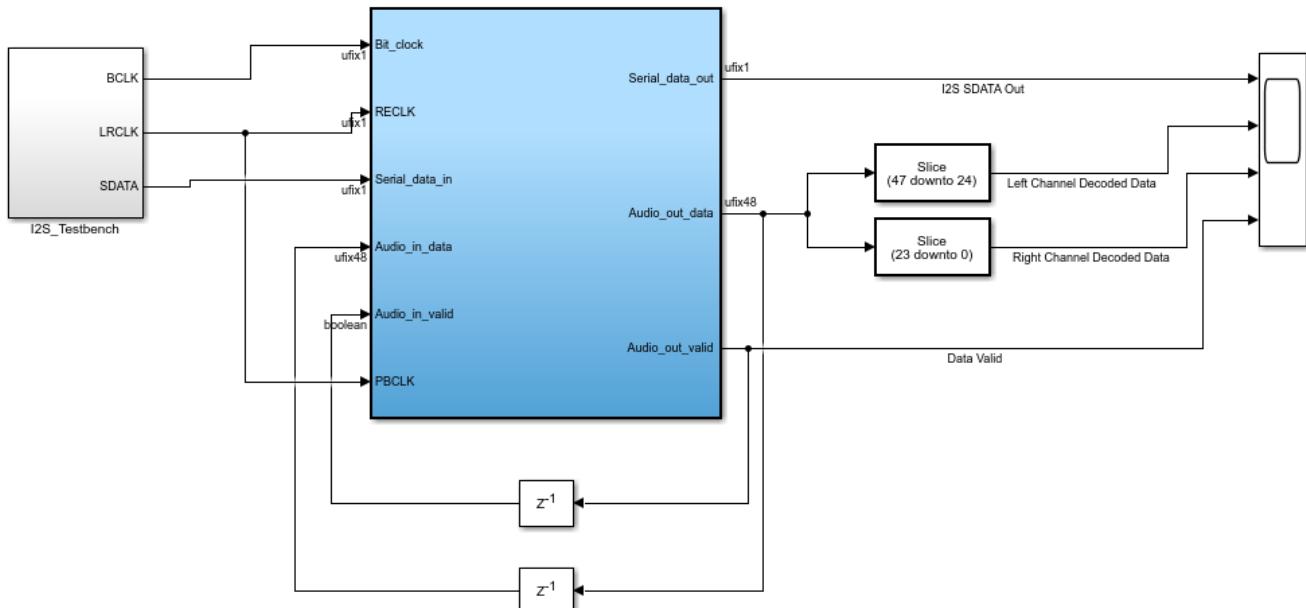
For creation of I2C IP to configure Audio Codec SSM2603, refer to “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 41-96 article.

1.2 Creation of I2S IP

Design a model in Simulink with a matlab function which implements the I2S protocol.

```
modelname = 'hdlcoder_I2S_ssm2603';
open_system(modelname);
```

Using IP Core Generation Workflow: I2S IP generation for Zybo / ArrowSocKit



This example shows how to use HDL Workflow Advisor to generate a custom IP core for implementing I2S protocol

In MATLAB, type the following:
`hdladvisor('hdlcoder_I2S_ssm2603/Subsystem')`

[Launch HDL Workflow Advisor](#)

[Run Demo](#)

Copyright 2016-2017 The MathWorks, Inc.

Testing and IP core generation steps are same as Zedboard I2S model. For generation of I2S IP, see "Authoring a Reference Design for Audio System on a Zynq Board" on page 41-170 example.

2. Create a custom audio codec reference design in Vivado

I2C, I2S and FIFO IPs are incorporated in the custom reference design. To create a custom reference design, refer to the "Create and export a custom reference design using Xilinx Vivado" section in "Define Custom Board and Reference Design for Zynq Workflow" on page 41-196.

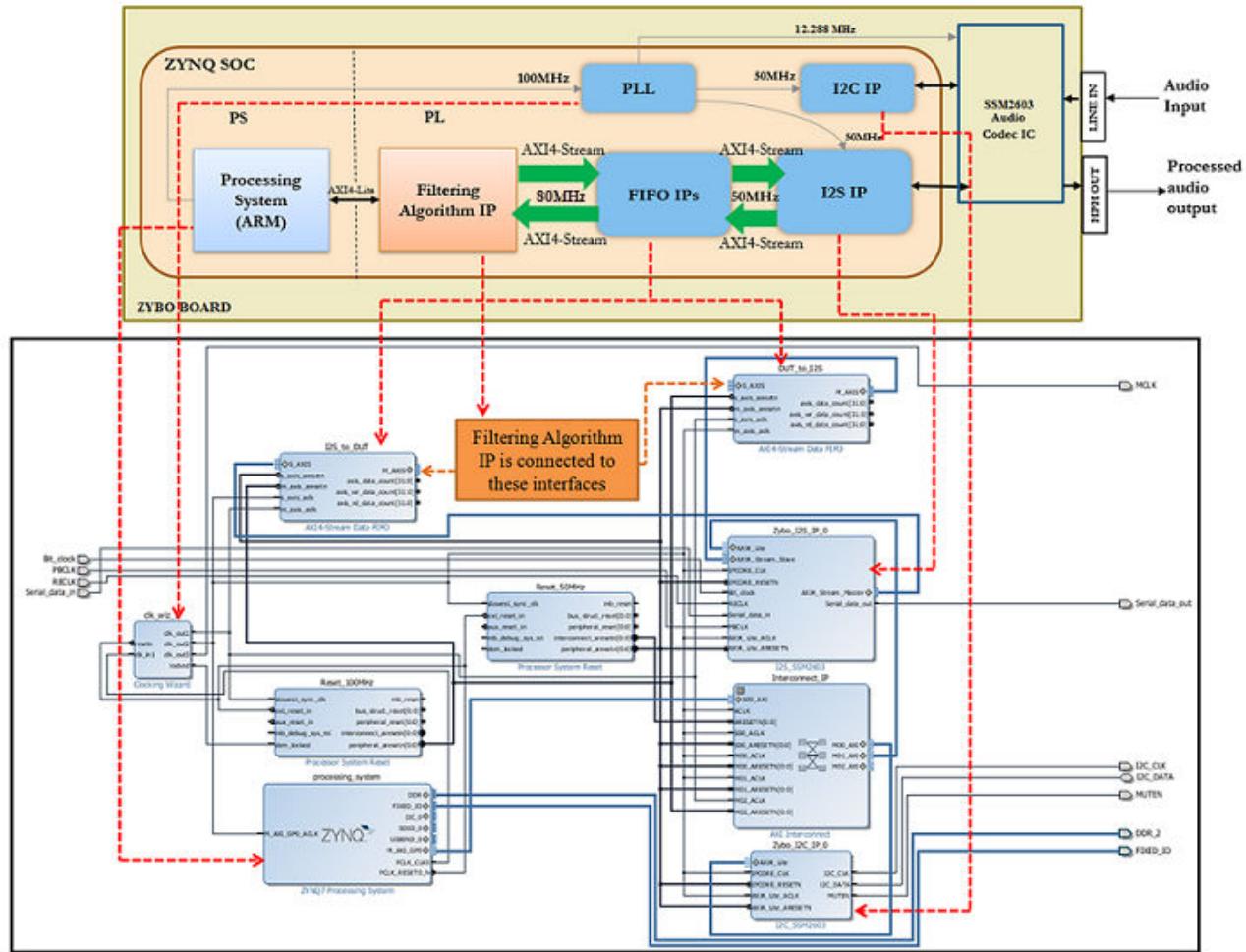
Key points to be noted while creating this custom reference design:

- 1 We must understand the theory of operation of the audio codec chip on the ZYBO board.
- 2 The FIFOs are set to default values for their configuration.
- 3 For the IP cores generated using HDL Workflow Advisor, **IPCORE_CLK** and **AXI4_Lite_ACLK** should be connected to the same clock source.
- 4 On validating the block design in Vivado, there should be no critical warnings except for the unconnected ports.
- 5 In this reference design, the audio codec is configured to operate in the Master mode.

The following signals run between the reference design on Zynq Soc and the audio codec on ZYBO board:

- 1 **Bit_clock** is the product of the sampling frequency, the number of bits per channel and the number of channels. It is driven by the audio codec in master mode. In this example, Sampling frequency is 48KHz, No of channels is 2, Number of bits per channel is 24.
- 2 **Serial_data_in** is the analog to digital converted audio data from the codec.
- 3 **Serial_data_out** is the digital audio data going to codec to be converted into analog form.
- 4 **I2C_CLK** and **I2C_DATA** are standard I2C signals
- 5 **MUTEN** is the Hardware mute pin connected to audio codec SSM2603.
- 6 **MCLK** is the 12.288MHz clock signal required by the codec.

The custom audio codec reference design created for this example is shown below:



3. Create the reference design definition file

The following code describes the contents of the ZYBO board reference design definition file **plugin_rd.m** for the above reference design. For more details on how to define and register custom board, refer to “Define Custom Board and Reference Design for Zynq Workflow” on page 41-196 example.

```

function hRD = plugin_rd()
% Reference design definition

% Copyright 2017-2018 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Audio System with AXI4 Stream Interface';
hRD.BoardName = 'ZYBO';

%% Tool information
hRD.SupportedToolVersion = {'2017.2','2017.4'};
```



```

%% Add custom design files
% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'system_top.tcl', ...
    'VivadoBoardPart',      'digilentinc.com:zybo:part0:1.0');

hRD.addIPRepository(... ...
    'IPLFunction', 'mathworks.hdlcoderdemo.vivado.hdlcoderdemo_ssm2603_iplist');

% Add constraint files
hRD.CustomConstraints = {'ZYBO_audio_filter_demo.xdc'};
```



```

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',      'clk_wiz_0/clk_outl', ...
    'ResetConnection',      'reset_100MHz/peripheral_aresetn',...
    'DefaultFrequencyMHz', 80, ...
    'MinFrequencyMHz',     5, ...
    'MaxFrequencyMHz',     500, ...
    'ClockModuleInstance', 'clk_wiz_0',...
    'ClockNumber',         1);

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'Interconnect_IP/M02_AXI', ...
    'BaseAddress',          '0x40010000', ...
    'MasterAddressSpace',   'processing_system/Data');
```



```

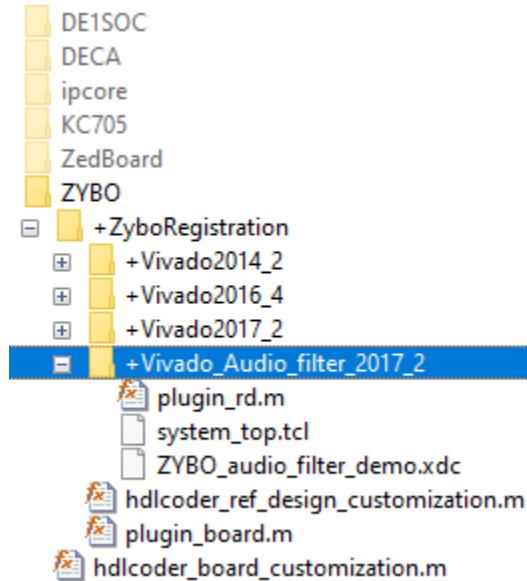
% add AXI4-Stream interface
hRD.addAXI4StreamInterface( ...
    'MasterChannelNumber', 1, ...
    'SlaveChannelNumber', 1, ...
    'MasterChannelConnection', 'DUT_to_I2S/S_AXIS', ...
    'SlaveChannelConnection', 'I2S_to_DUT/M_AXIS', ...
    'MasterChannelDataWidth', 48, ...
    'SlaveChannelDataWidth', 48 ... );
```

41-184

Go to **ZYBO** folder using the following command:

```
cd ([matlabroot '/toolbox/hdlcoder/hdlcoderdemos/customboards/ZYBO']);
```

All files that are required for the reference design such as IP core files, XDC files, plugin_rd file etc should be added to the matlab path, inside **ZYBO** folder using the hierarchy shown below. The user generated IP core files should be in **+vivado** folder. plugin_rd.m, tcl files and xdc files should be in **+vivado_audio_filter_2017_2** folder.



4. Verify the reference design

To verify the reference design, to generate Audio Filter IP core from a model and integrate it with the audio codec reference design, refer to “Running an Audio Filter on Live Audio Input Using a Zynq Board” on page 41-126 example.

Authoring a Reference Design for Audio System on Intel board

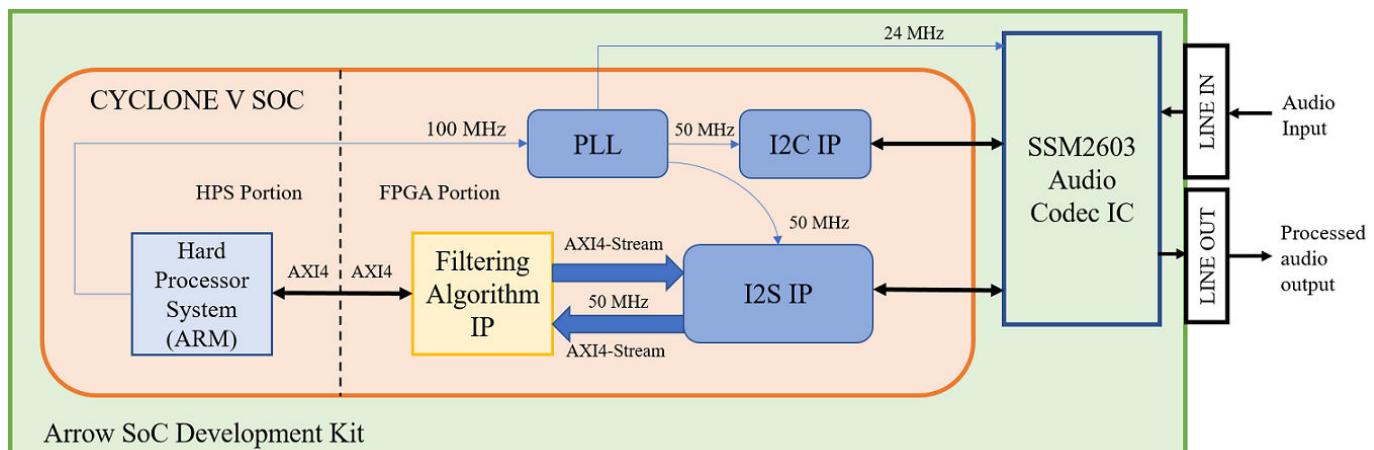
This example shows how to:

- 1 Generate IP cores for peripheral interfaces using **HDL Workflow Advisor**
- 2 Build a reference design to run an audio algorithm and access audio input and output

Introduction

In this example, you create a reference design which receives audio input from Intel Arrow SoC Development Kit, performs some processing on it and transmits the processed audio data out of Arrow SoC Development Kit. To perform audio processing on Arrow SoC, you need the following 2 protocols:

- 1 I2C to configure the SSM2603 audio codec chip on Arrow SoC.
- 2 I2S to stream the digitized audio data between the codec chip and Cyclone V FPGA.



The above figure is a high level architecture diagram that shows how the reference design is used by the Filtering Algorithm IP. I2S IP operates at 50MHz frequency whereas one may want to run the Filtering Algorithm IP at a higher frequency. This frequency is controlled in Step **1.4** in **HDL Workflow Advisor**. In this example, assume that the filter operates at 50MHz. Depending on the type of filter selected, Filtering Algorithm IP filters a range of frequencies from the incoming audio data and passes the filtered audio data out. In the above figure, the Filtering Algorithm IP is our main algorithm that we are modeling in Simulink. While the I2C and I2S IPs need to be created. Here, you has 3 choices:

- Use pre-packaged IP, e.g., if one exists
- Model it in Simulink and generate an IP core using IP core generation workflow or
- Use legacy HDL code. To create an IP out of legacy HDL code, use a Simulink model to black box the HDL code and generate IP core from it. I2C, I2S, PLL IPs and Cyclone V Hard Processor System(HPS) form a part of the reference design.

The following steps are used to create the reference design described above:

- 1 Generate IP Cores for peripheral interfaces
- 2 Create a custom audio codec reference design in Qsys

- 3 Create the reference design definition file
- 4 Verify the reference design

1. Generate IP Cores for peripheral interfaces using HDL Workflow Advisor

In this example,

- 1 I2C IP is developed by modeling it using Stateflow blocks, and also using legacy VHDL code for tristate buffer.
- 2 I2S IP is developed by modeling it in Simulink.

1.1 Creation of I2C IP

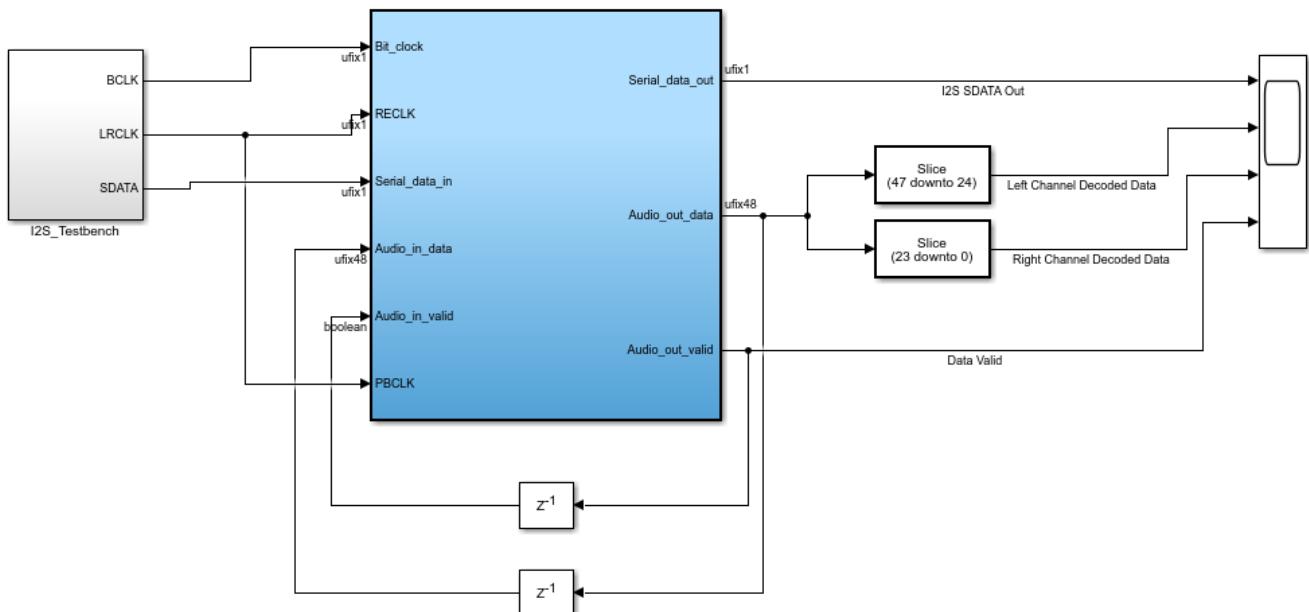
For creation of I2C IP to configure Audio Codec SSM2603, refer to “IP Core Generation of an I2C Controller IP to Configure the Audio Codec Chip” on page 41-96.

1.2 Creation of I2S IP

Design a model in Simulink with a matlab function which implements the I2S protocol.

```
modelname = 'hdlcoder_I2S_ssm2603';
open_system(modelname);
```

Using IP Core Generation Workflow: I2S IP generation for Zybo / ArrowSocKit



This example shows how to use HDL Workflow Advisor to generate a custom IP core for implementing I2S protocol

In MATLAB, type the following:
`hdladvisor('hdlcoder_I2S_ssm2603/Subsystem')`

[Launch HDL Workflow Advisor](#)

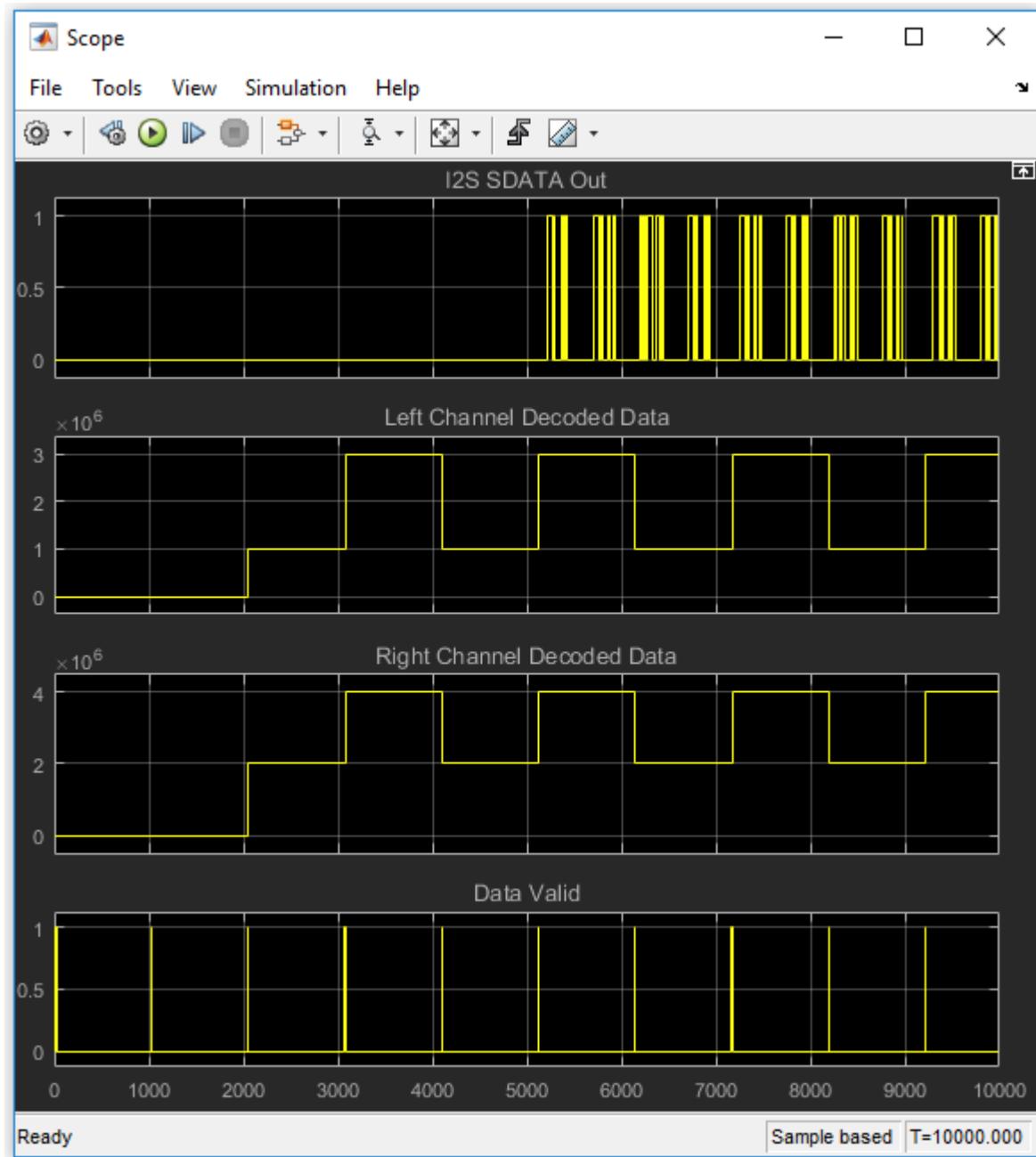
[Run Demo](#)

Copyright 2016-2017 The MathWorks, Inc.

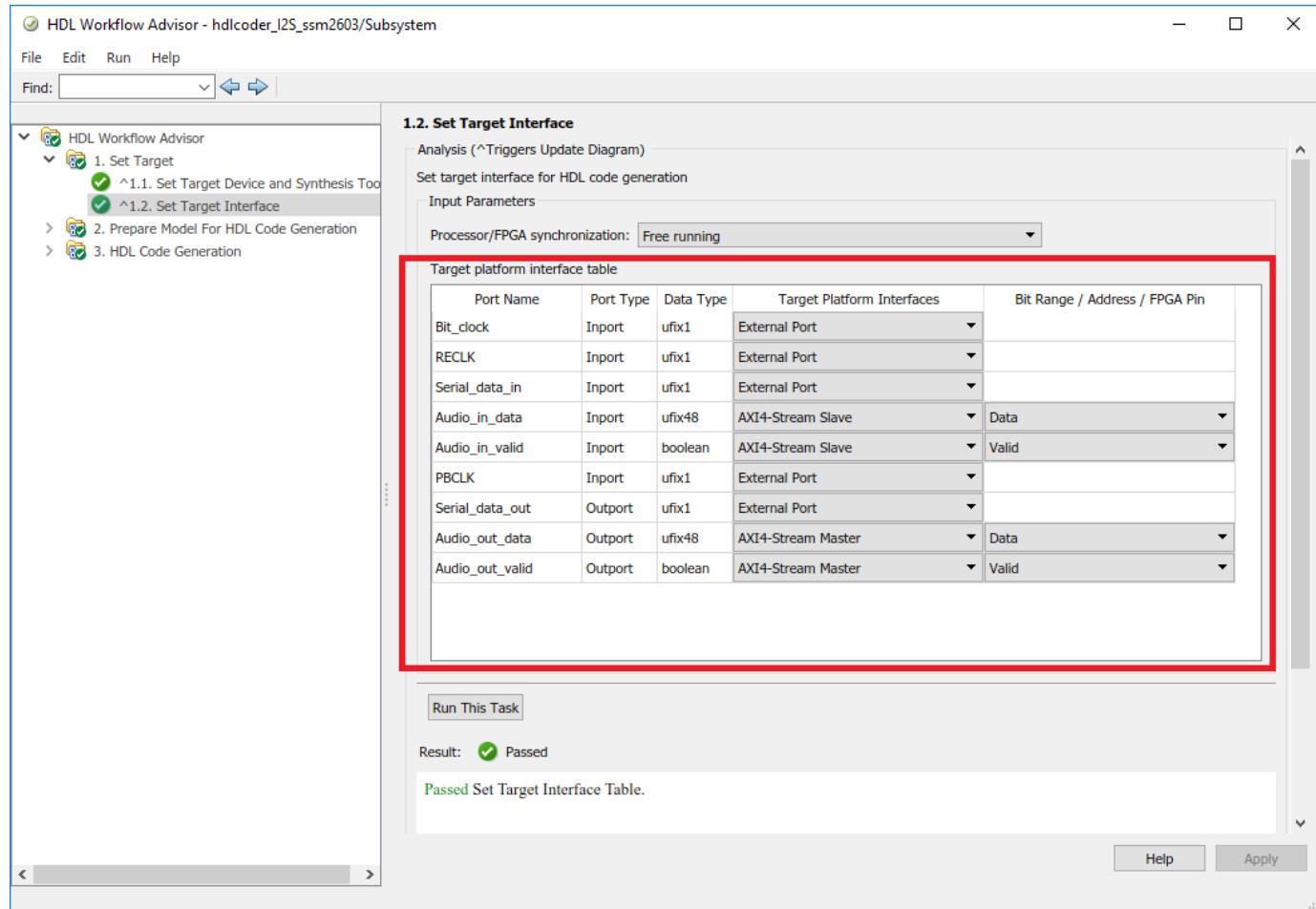
Create a test bench in the model to mimic the incoming audio data from the codec.



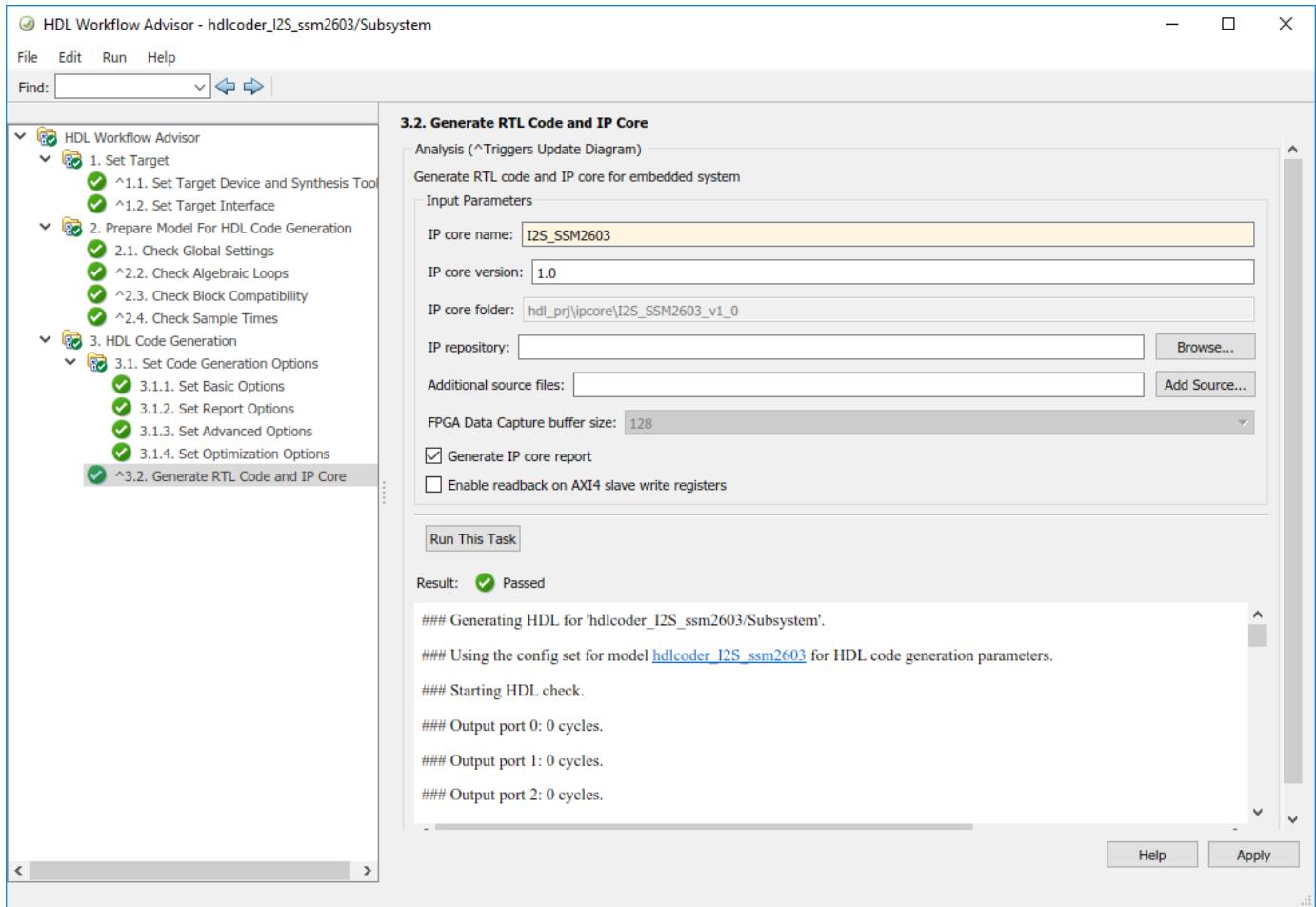
Feed this data to the Subsystem block which does the I2S operation. Verify the output of the Subsystem on a Scope.



Start the HDL Workflow Advisor from the DUT subsystem. In Task 1.1, keep the same settings as those of I2C IP generated earlier. in Task 1.2, set the Target Platform Interfaces as shown below:



Run Task 3.2 and generate the ip core.



2. Create a custom audio codec reference design in Qsys

I2C and I2S IPs are incorporated in the custom reference design. To create a custom reference design, refer to the **Reference Design creation using Intel Quartus Prime** section in “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215.

Key points to be noted while creating this custom reference design:

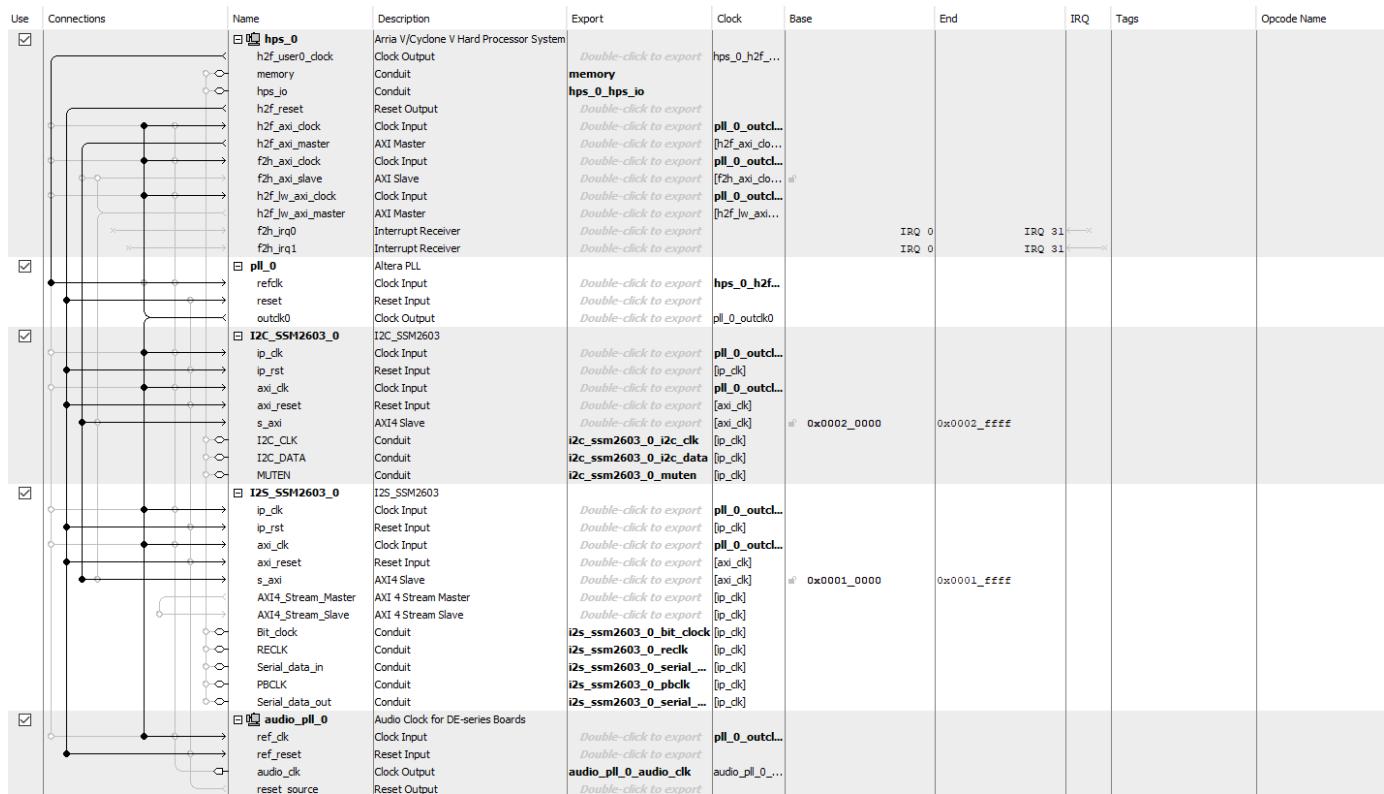
- 1 We must understand the theory of operation of the audio codec chip on the Arrow SoC.
- 2 For the IP cores generated using HDL Workflow Advisor, **IPCORE_CLK** and **AXI4_ACLK** should be connected to the same clock source.
- 3 In this reference design, the audio codec is configured to operate in the Master mode.

The following signals run between the reference design on Intel SoC and the audio codec on Arrow SoC:

- 1 **Bit_clock** is the product of the sampling frequency, the number of bits per channel and the number of channels. It is driven by the audio codec in master mode. In this example, Sampling frequency is 48KHz, No of channels is 2, Number of bits per channel is 24.
- 2 **Left_right_select** is to distinguish between left audio channel data and right audio channel data. It is in sync with the Bit clock.

- 3 **Serial_data_in** is the analog to digital converted audio data from the codec.
- 4 **Serial_data_out** is the digital audio data going to codec to be converted into analog form.
- 5 **I2C_CLK** and **I2C_DATA** are standard I2C signals
- 6 **ADDR0** and **ADDR1** are the I2C Address Bits.
- 7 **Clk_24MHz** is the 24MHz clock signal required by the codec.

The custom audio codec reference design created for this example is shown below:



3. Create the reference design definition file

The following code describes the contents of the Arrow SoC Development Kit reference design definition file **plugin_rd.m** for the above reference design. For more details on how to define and register custom board, refer to “Define Custom Board and Reference Design for Intel SoC Workflow” on page 41-215.

```

function hRD = plugin_rd()
% Reference design definition

% Copyright 2012-2019 The MathWorks, Inc.

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Altera QUARTUS II');

hRD.ReferenceDesignName = 'Audio System with AXI4 Stream Interface';

hRD.BoardName = 'Arrow SoCKit development board';

% Tool information
hRD.SupportedToolVersion = {'18.1'};

%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign( ...
    'CustomQsysPrjFile', 'system_soc.qsys');

hRD.addIPRepository(...
    'IPLListFunction', 'mathworks.hdlcoderdemo.quartus.hdlcoderdemo_ssm2603_quartus_iplist');

% Add constraint files
hRD.CustomConstraints = {'system_soc.tcl'};

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'pll_0.outclk0', ...
    'ResetConnection', 'hps_0.h2f_reset',...
    'DefaultFrequencyMHz', 50, ...
    'MinFrequencyMHz', 5, ...
    'MaxFrequencyMHz', 500, ...
    'ClockModuleInstance', 'pll_0',...
    'ClockNumber', 0);

% add AXI4 slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'hps_0.h2f_axi_master', ...
    'BaseAddress', '0x0000');

% add AXI4-Stream interface
hRD.addAXI4StreamInterface( ...
    'MasterChannelEnable', true, ...
    'SlaveChannelEnable', true, ...
    'MasterChannelConnection', 'I2S_SSM2603_0.AXI4_Stream_Slave', ...
    'SlaveChannelConnection', 'I2S_SSM2603_0.AXI4_Stream_Master', ...
    'MasterChannelDataWidth', 48, ...
    'SlaveChannelDataWidth', 48 ...
);

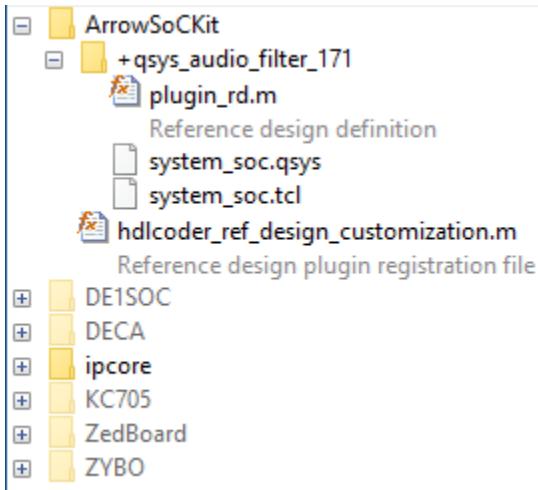
```

Go to **ArrowSoC** folder using the following command:

```
cd ([matlabroot '/toolbox/hdlcoder/hdldcoderdemos/customboards/ArrowSoC']);
```

All files that are required for the reference design such as IP core files, qsys file, tcl file, plugin_rd file etc should be added to the matlab path, inside **ArrowSoC** folder using the hierarchy shown below.

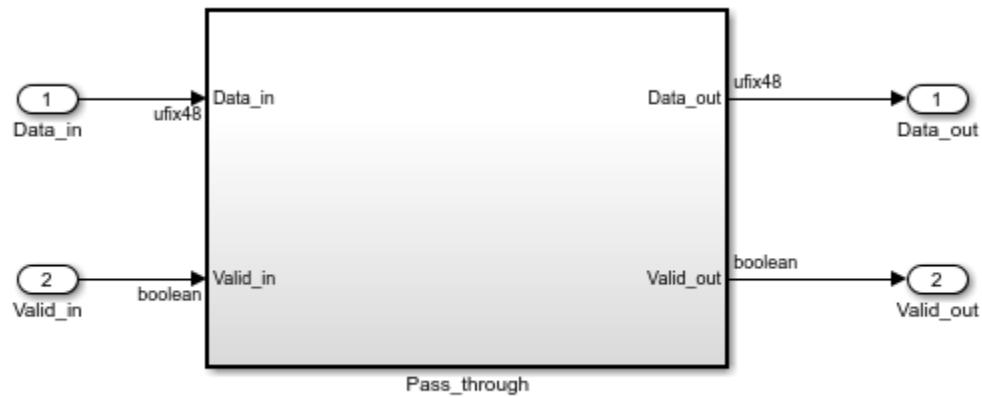
The user generated IP core files should be in **+quartus** folder. plugin_rd.m, tcl files and qsys files should be in the reference design plugin folder, for example, **+qsys_audio_filter_18_1** folder.



4. Verify the reference design

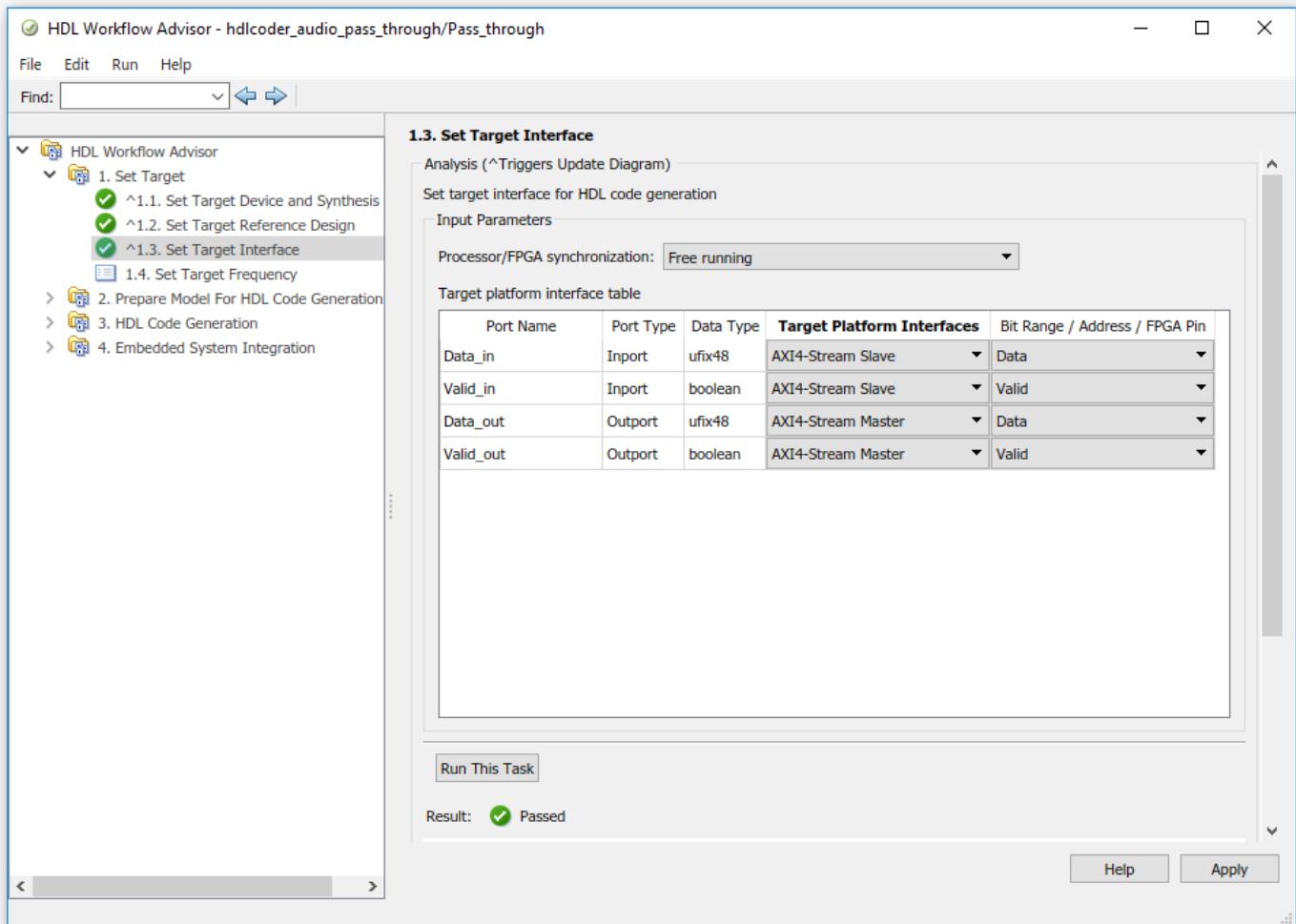
In order to ensure that the reference design and the interfaces in the reference design work as expected, design a Simulink model which just sends the audio through the Algorithm IP, integrate it with the reference design and test it on Arrow SoC. You should be able to hear the audio loop back.

```
modelname = 'hdlcoder_audio_pass_through';
open_system(modelname);
```



Copyright 2016 The MathWorks, Inc.

The interfaces in the model should be selected as shown below:



To generate IP core from a model and integrate it with the audio codec reference design, refer to “Running an Audio Filter on Live Audio Input using Intel Board” on page 41-116.

Define Custom Board and Reference Design for Zynq Workflow

This example shows how to define and register a custom board and reference design in the Zynq® workflow.

Introduction

Using this example, you will be able to register the Digilent® Zybo Zynq development board and a custom reference design in the HDL Workflow Advisor for the Zynq workflow.

This example uses a Zybo Zynq board, but in the same way, you can define and register a custom board or a custom reference design for other Zynq platforms.

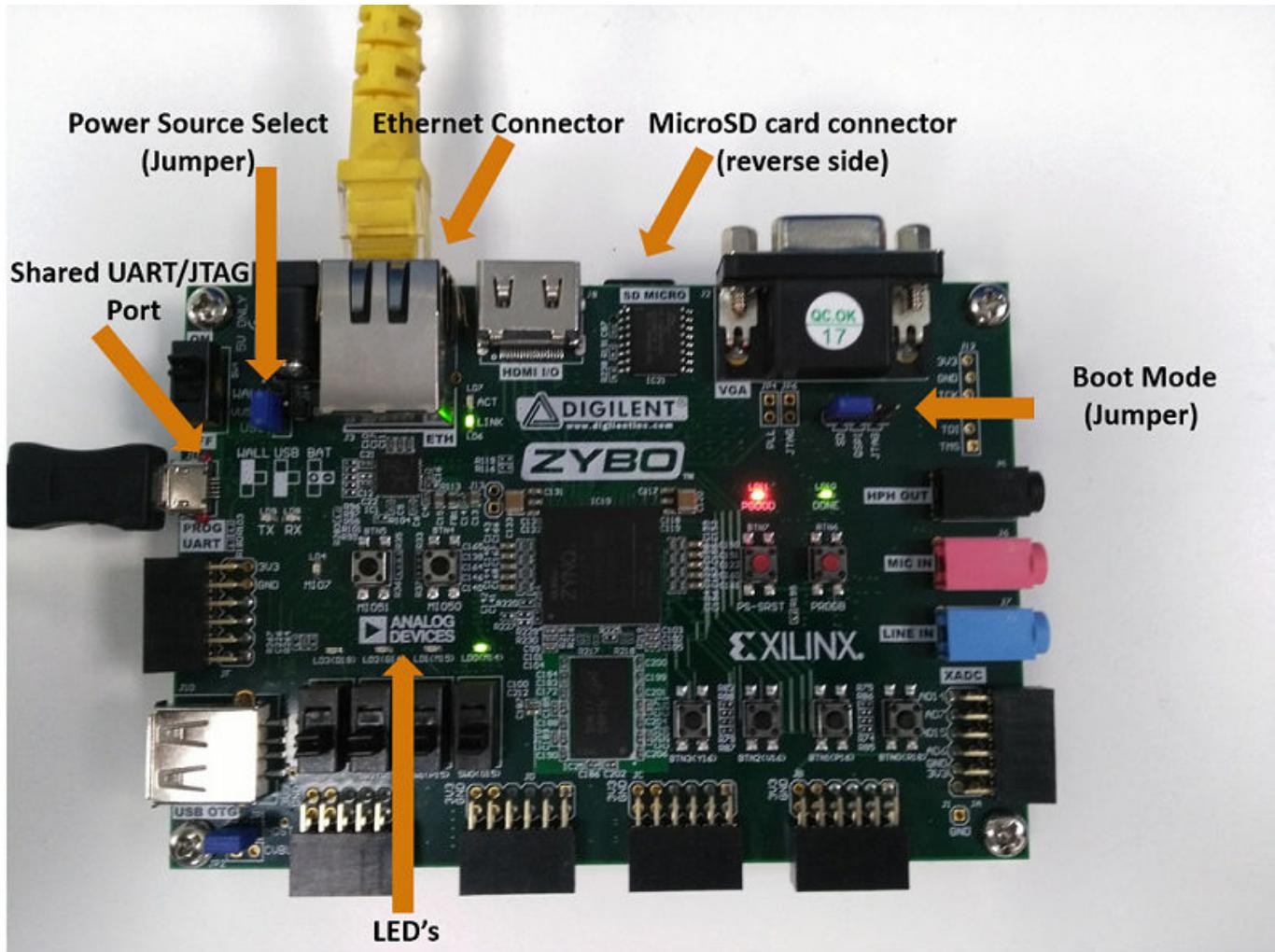
Requirements

- Xilinx Vivado Design Suite, with supported version listed in the HDL Coder documentation
- Digilent® Zybo Zynq™ development board with the accessory kit
- HDL Coder support package for Xilinx Zynq Platform
- Embedded Coder support package for Xilinx Zynq Platform

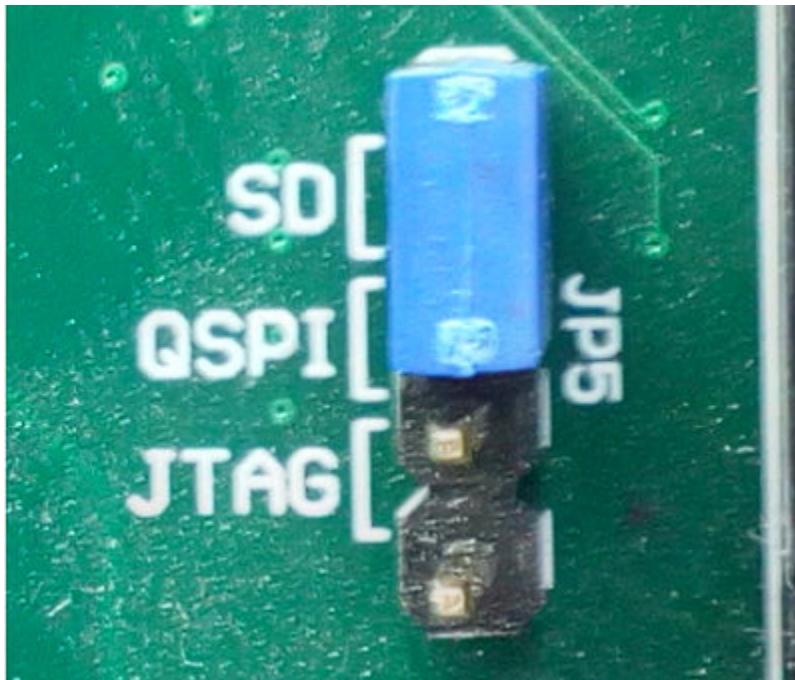
Note: This example uses Digilent® Zybo Zynq-7000 ARM/FPGA SoC trainer board. This example does not work on Digilent® Zybo Z7: Zynq-7000 ARM/FPGA SoC development board which have two variants Zybo Z7-10 and Zybo Z7-20.

Set up the Zybo board

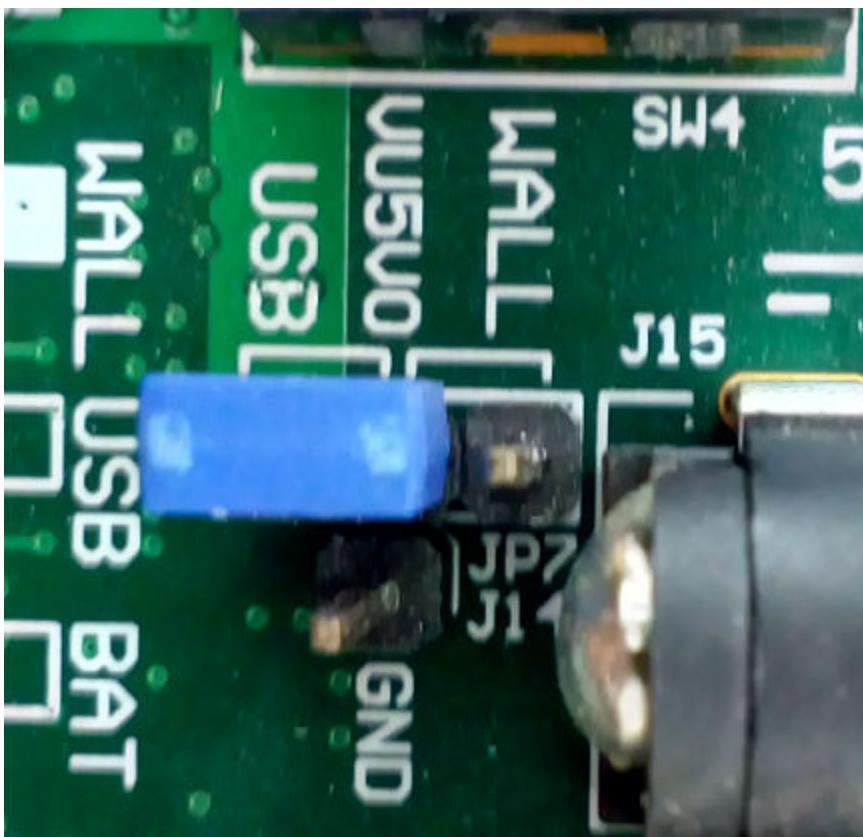
1. Understand the features available on the Zybo board by reading the Zybo board reference manual.
2. Set up the Zybo board as shown in the following figure:



3. Ensure that you have properly installed the USB COM port device drivers on your computer.
4. Configure the JP5 boot mode jumper to enable the loading of a Zynq Linux image from a microSD card connected to connector J4 as shown in the following figure.



5. Configure the JP7 power source select jumper to use USB as the power source as shown in the following figure.



6. Connect the shared UART/JTAG USB port on the Zybo board to your computer.

7. Connect the Zybo board to your computer using an Ethernet cable. The default Zybo IP address is 192.168.1.110.

8. Download the Zybo Zynq Linux image, extract the Zip archive and copy the contents to the microSD card. Insert the microSD card in connector J4.

9. Set up the Xilinx Vivado tool path by using the following command:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2017.4\bin\vivado.bat')
```

Use your own Xilinx Vivado installation path when executing the command.

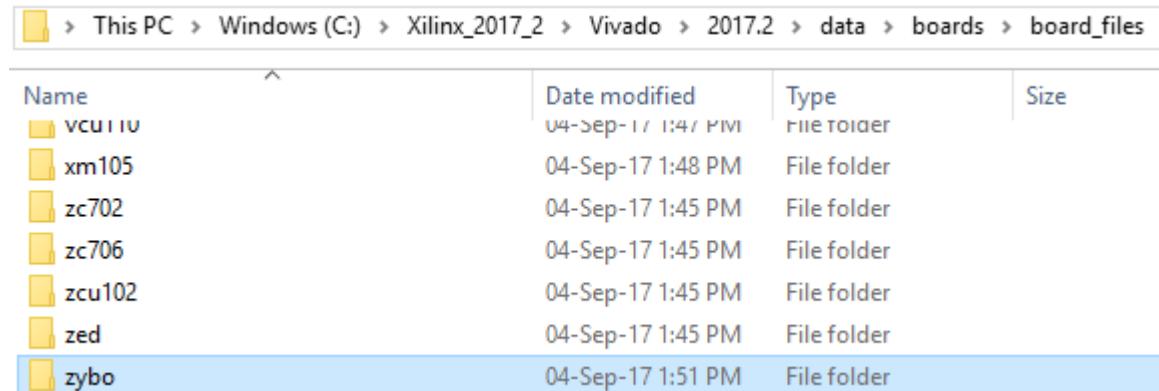
10. Set up the Zynq hardware connection by using the following command:

```
h = zynq();
```

Register the Zybo board part in Xilinx Vivado tool

By Default Installation, Vivado 2017.4 tool will not have the Zybo board part pre-installed. These files must be downloaded from the Digilent website. So unzip the content and navigate to the installation directory of Vivado given below and copy the updated Zybo board files to xilinx vivado tools manually.

C:\Xilinx\Vivado\2017.4\data\boards\board_files



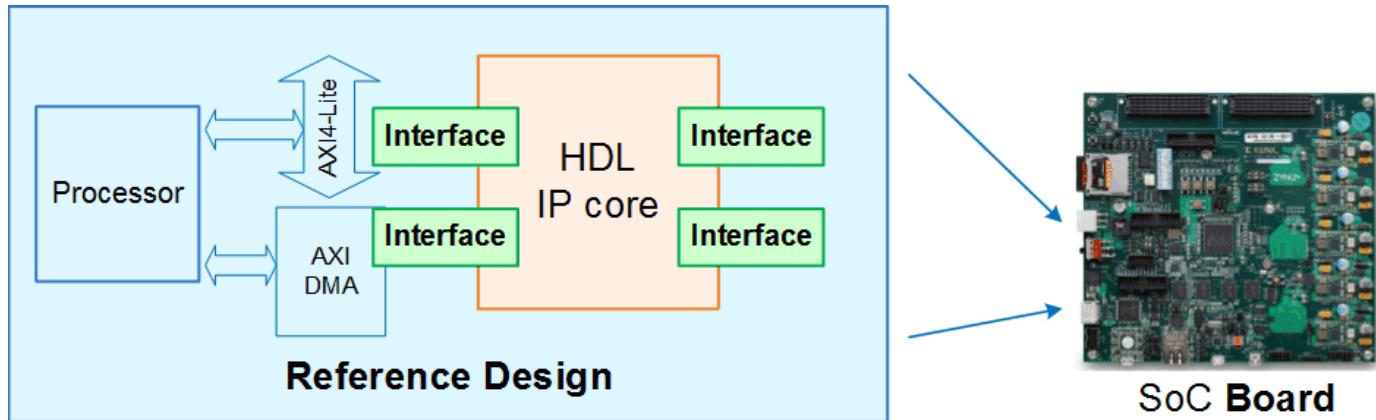
Name	Date modified	Type	Size
vcu110	04-Sep-17 1:47 PM	File folder	
xm105	04-Sep-17 1:48 PM	File folder	
zc702	04-Sep-17 1:45 PM	File folder	
zc706	04-Sep-17 1:45 PM	File folder	
zcu102	04-Sep-17 1:45 PM	File folder	
zed	04-Sep-17 1:45 PM	File folder	
zybo	04-Sep-17 1:51 PM	File folder	

as the result of this step, Zybo board part is added to list of development boards while creating vivado project specific to board.

Note: In case above link is unavailable, get Zybo board files from Digilent website.

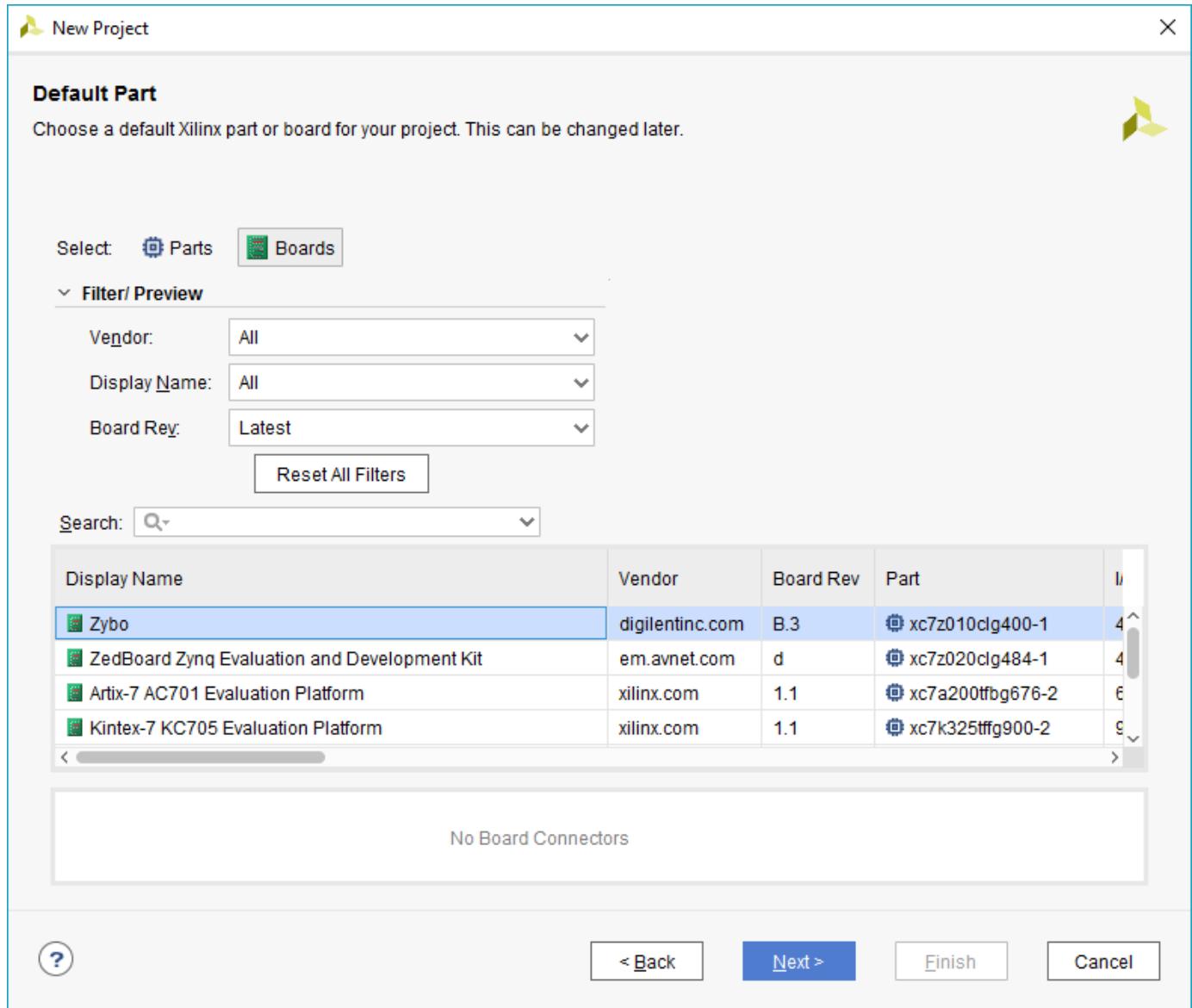
Create and export a custom reference design using Xilinx Vivado

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. The HDL Coder SoC workflow generates an IP core that integrates with the reference design, and is then used to program an SoC board. The following figure describes the relationship between a reference design, an HDL IP core and an SoC board.

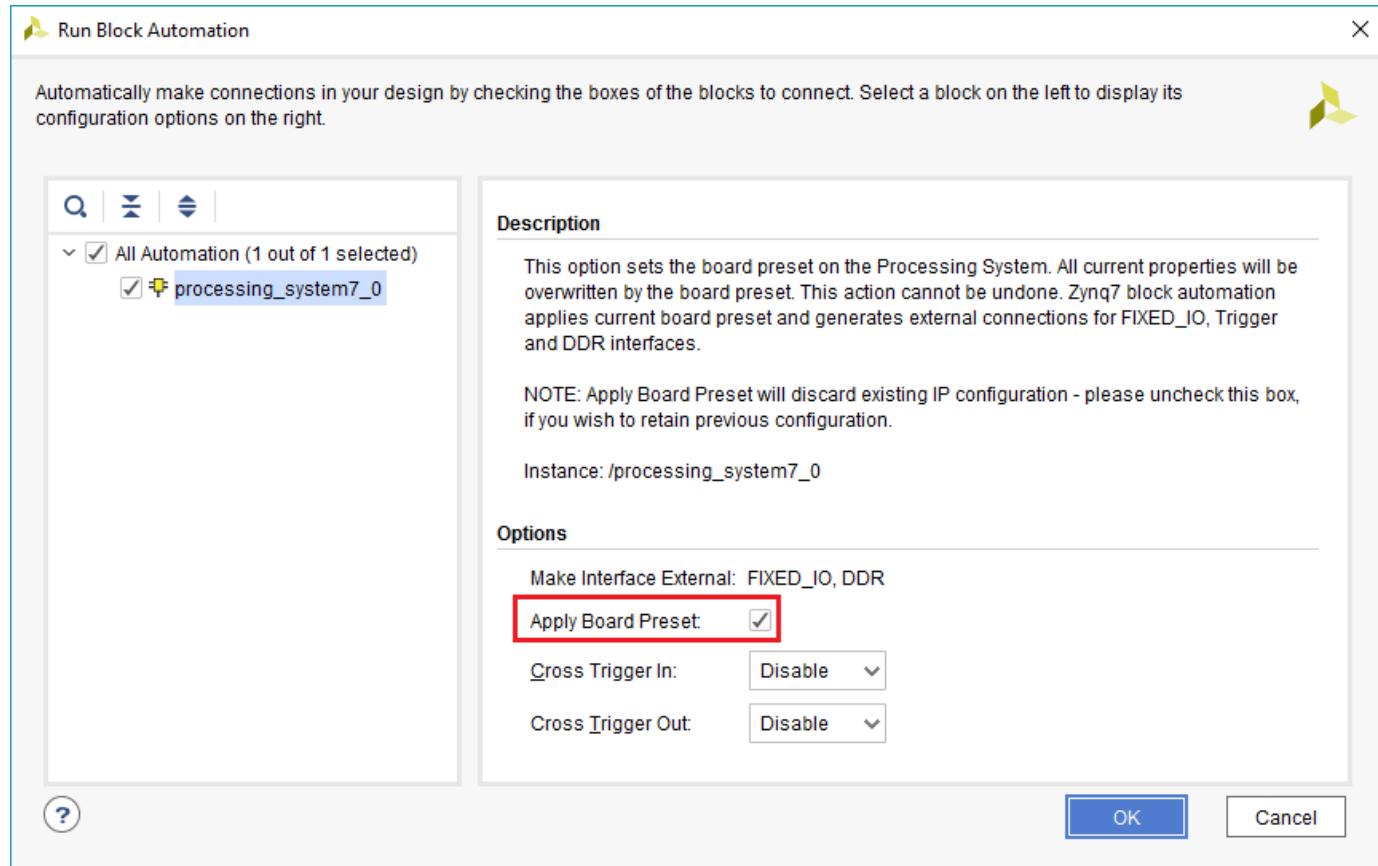


In this section, we outline the basic steps necessary to create and export a simple reference design using the Xilinx Vivado IP Integrator environment. For more information about the IP Integrator tool, refer to Xilinx documentation.

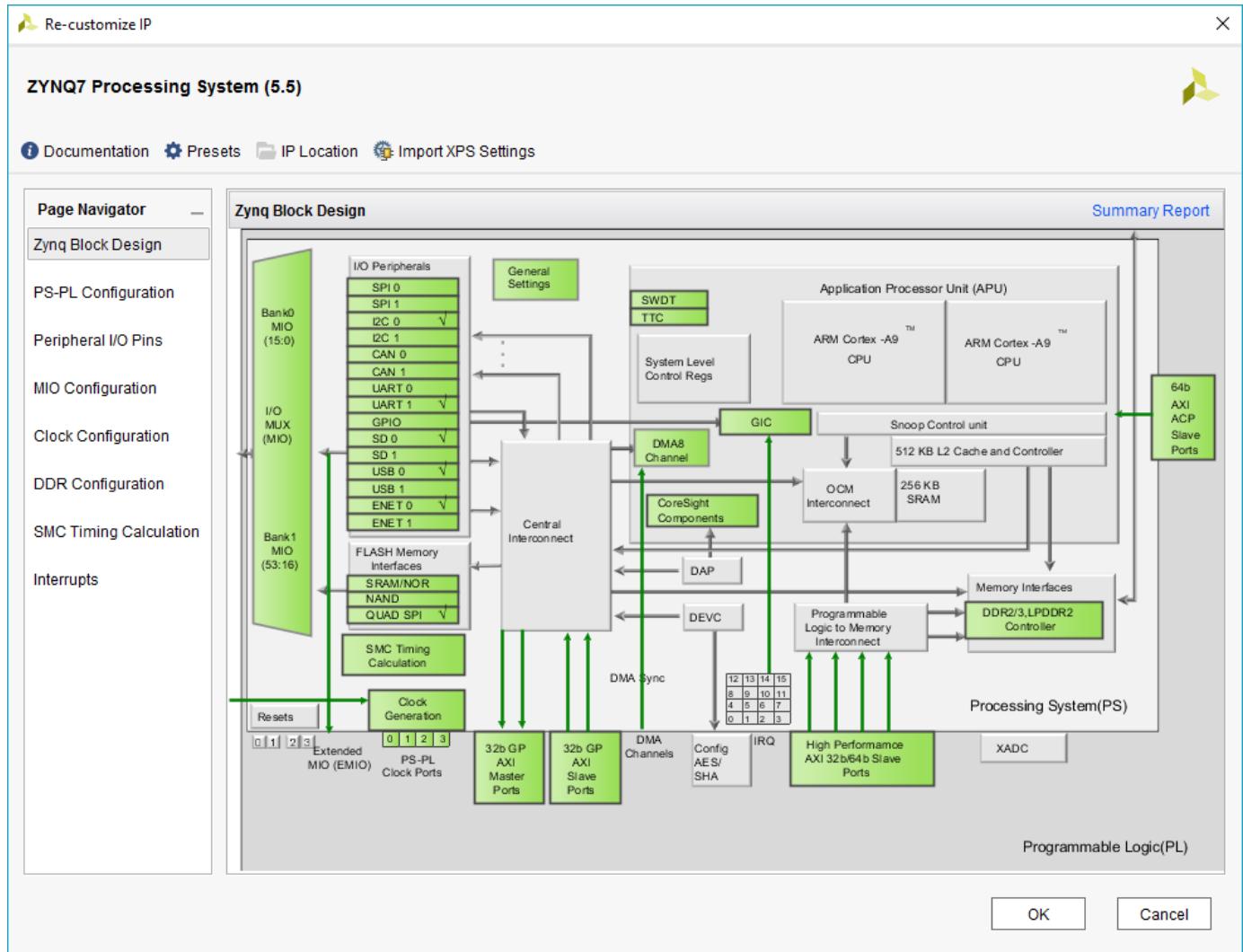
1. Create an empty Xilinx Vivado RTL project using board part **Zybo** as the default board part as shown in the following figure:



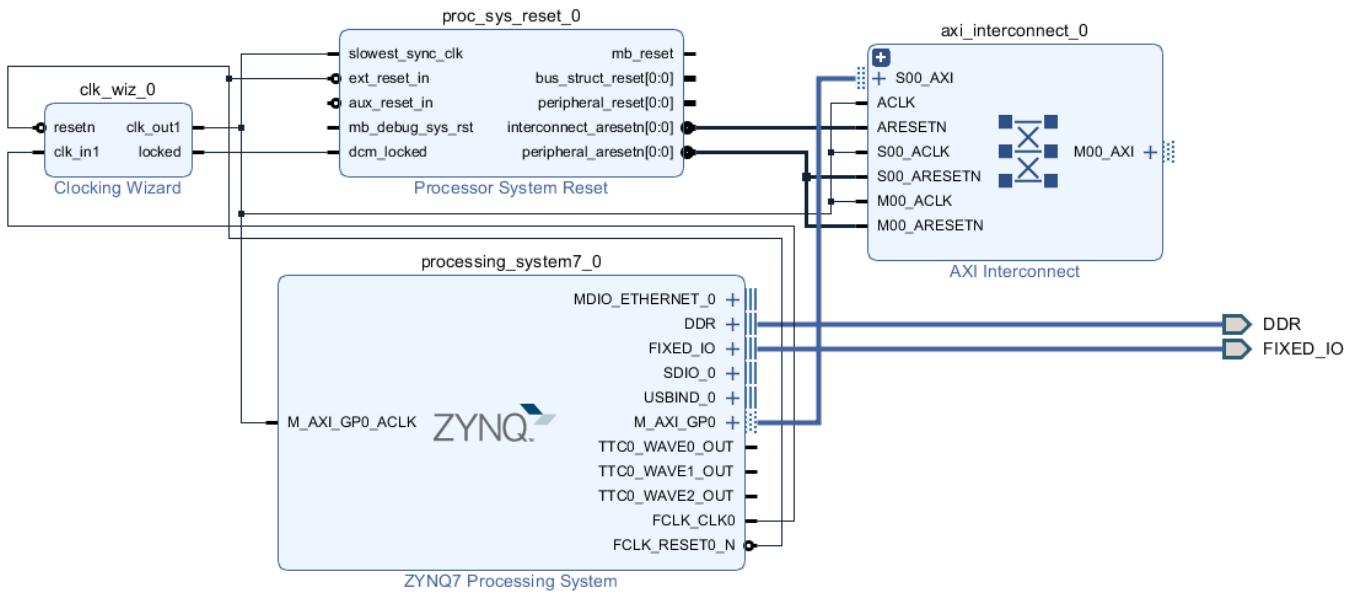
2. Create an empty block design and add the **ZYNQ7 Processing System** IP block. Run block automation as shown in the figure to set board preset for Zybo which contains the parameters for ZYNQ7 Processing system IP related to MIO Configuration, Clock configuration and DDR Configuration.



In the following figure you can see the MIO peripherals are been marked accordance to the Zybo board definition as the result of apply board preset.

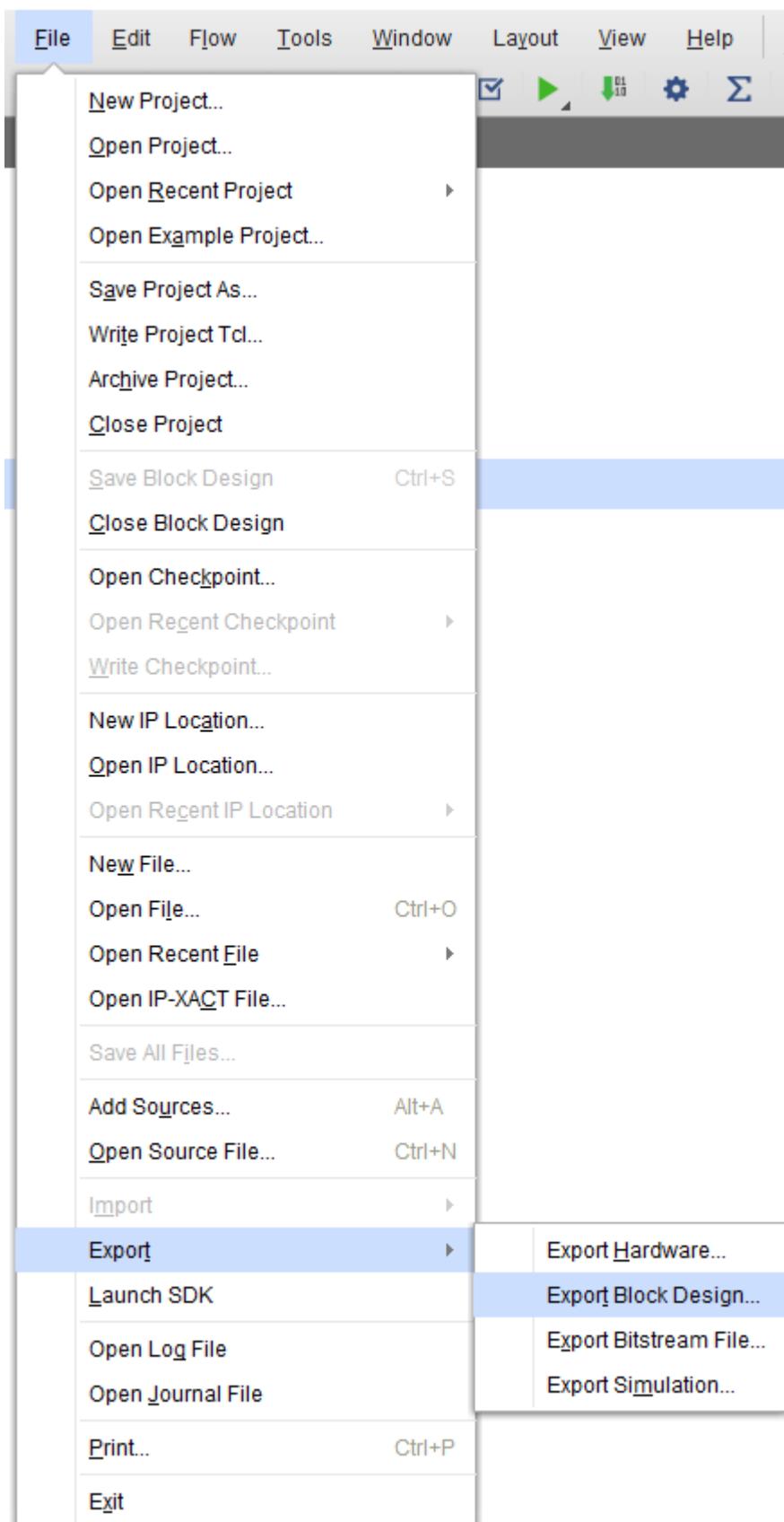


3. Complete the block design as shown in the following figure:



Notice that the block design does not contain any information about the HDL IP core.

4. Export the completed block design as a Tcl script `design_led.tcl` as shown in the following figure:



The exported Tcl script (`design_led.tcl`) constitute the custom reference design. The Tcl script will be used in the HDL Coder SoC workflow to re-create the block design and integrate the generated HDL IP core with the block design in a Xilinx Vivado project.

Register the Zybo board in HDL Workflow Advisor

In this section, we outline the steps necessary to register the Zybo board in HDL Workflow Advisor.

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB path.

A board registration file contains a list of board plugins. A board plugin is a MATLAB package folder containing a board definition file and all reference design plugins associated with the board.

The following code describes the contents of a board registration file that contains the board plugin `ZyboRegistration` to register the Zybo board in HDL Workflow Advisor.

```
function r = hdlcoder_board_customization
% Board plugin registration file
% 1. Any registration file with this name on MATLAB path will be picked up
% 2. Registration file returns a cell array pointing to the location of
%    the board plugins
% 3. Board plugin must be a package folder accessible from MATLAB path,
%    and contains a board definition file

r = { ...
    'ZyboRegistration.plugin_board', ...
};
end
```

2. Create the board definition file.

A board definition file contains information about the SoC board.

The following code describes the contents of the Zybo board definition file `plugin_board.m` that resides inside the board plugin `ZyboRegistration`.

Information about the FPGA I/O pin locations ('`FPGAPin`') and standards ('`IOSTANDARD`') is obtained from the Zybo master constraints file from Digilent.

The property `BoardName` defines the name of the Zybo board as `ZYBO` in HDL Workflow Advisor.

```
function hB = plugin_board()
% Board definition

% Construct board object
hB = hdlcoder.Board;

hB.BoardName      = 'ZYBO';

% FPGA device information
hB.FPGAVendor    = 'Xilinx';
hB.FPGAFamily    = 'Zynq';
hB.FPGADevice    = 'xc7z010';
hB.FPGAPackage   = 'clg400';
hB.FPGASpeed     = '-1';

% Tool information
hB.SupportedTool = {'Xilinx Vivado'};
```

```
% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

% Custom board external I/O interface
hB.addExternalIOInterface( ...
    'InterfaceID', 'LEDs General Purpose', ...
    'InterfaceType', 'OUT', ...
    'PortName', 'LEDs', ...
    'PortWidth', 4, ...
    'FPGAPin', {'M14', 'M15', 'G14', 'D18'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});

hB.addExternalIOInterface( ...
    'InterfaceID', 'Push Buttons', ...
    'InterfaceType', 'IN', ...
    'PortName', 'PushButtons', ...
    'PortWidth', 4, ...
    'FPGAPin', {'R18', 'P16', 'V16', 'Y16'}, ...
    'IOPadConstraint', {'IOSTANDARD = LVCMOS33'});
```

Register the custom reference design in HDL Workflow Advisor

In this section, we outline the steps necessary to register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

A reference design plugin is a MATLAB package folder containing the reference design definition file and all files associated with the SoC design project. A reference design registration file must also contain the name of the associated board.

The following code describes the contents of a Zybo reference design registration file containing the reference design plugin `ZyboRegistration.Vivado2017_2` associated with the board `ZYBO`.

```
function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file
% 1. The registration file with this name inside of a board plugin folder
% will be picked up
% 2. Any registration file with this name on MATLAB path will also be picked up
% 3. The registration file returns a cell array pointing to the location of
% the reference design plugins
% 4. The registration file also returns its associated board name
% 5. Reference design plugin must be a package folder accessible from
% MATLAB path, and contains a reference design definition file

rd = {'ZyboRegistration.Vivado2017_2.plugin_rd', ...};
boardName = 'ZYBO';
end
```

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that will be generated by the HDL Coder SoC workflow.

The following code describes the contents of the Zybo reference design definition file `plugin_rd.m` associated with the board ZYBO that resides inside the reference design plugin `ZyboRegistration.Vivado2017_2`. The property `ReferenceDesignName` defines the name of the reference design as `Demo system` in HDL Workflow Advisor.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');

hRD.ReferenceDesignName = 'Demo system';
hRD.BoardName = 'ZYBO';

% Tool information
hRD.SupportedToolVersion = {'2017.2', '2017.4'};

% add custom Vivado design
hRD.addCustomVivadoDesign( ...
    'CustomBlockDesignTcl', 'design_led.tcl', ...
    'VivadoBoardPart', 'digilentinc.com:zybo:part0:1.0');

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'clk_wiz_0/clk_out1', ...
    'ResetConnection', 'proc_sys_reset_0/peripheral_aresetn');

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress', '0x40010000', ...
    'MasterAddressSpace', 'processing_system7_0/Data');
```

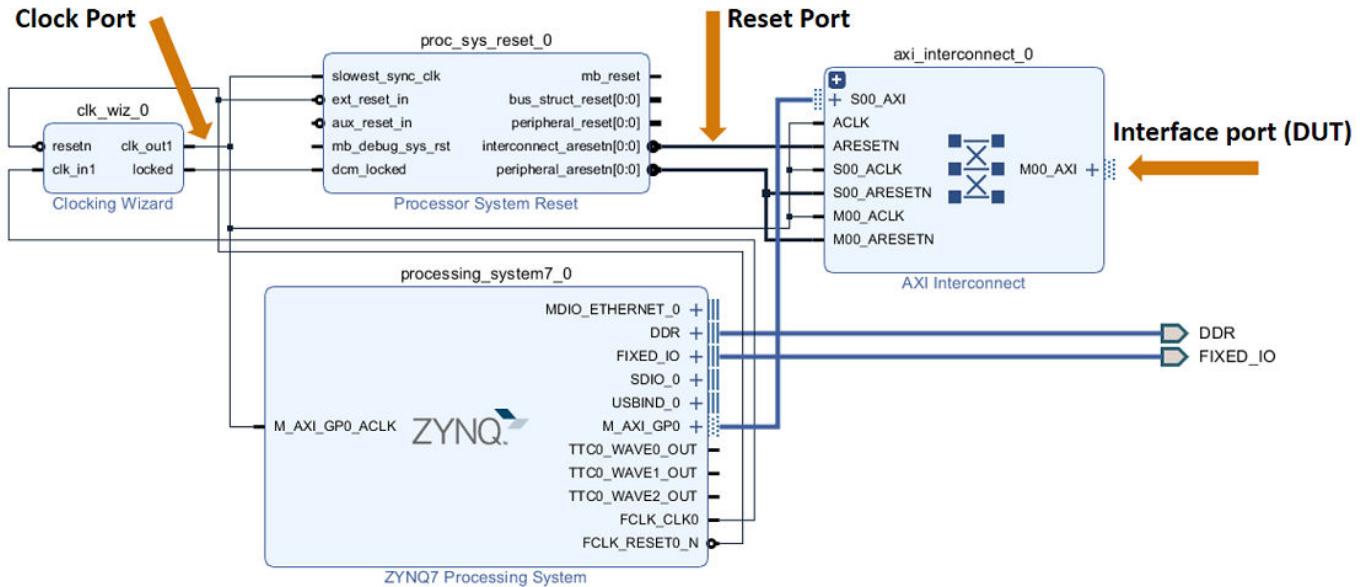
In addition to the reference design definition file, a reference design plugin must also contain the SoC design project files.

The Zybo reference design plugin folder `ZyboRegistration.Vivado2017_2` must contain the Tcl script `design_led.tcl` exported previously from the Xilinx Vivado project. The Zybo reference design definition file `plugin_rd.m` identifies the SoC design project file via the following statement:

```
hRD.addCustomVivadoDesign('CustomBlockDesignTcl', 'design_led.tcl');
```

In addition to the SoC design project files, `plugin_rd.m` also defines the interface connections between the custom reference design and the HDL IP core indicated in the following figure via the statements:

```
hRD.addClockInterface( ...
    'ClockConnection', 'clk_wiz_0/clk_out1', ...
    'ResetConnection', 'proc_sys_reset_0/peripheral_aresetn');
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_interconnect_0/M00_AXI', ...
    'BaseAddress', '0x40010000', ...
    'MasterAddressSpace', 'processing_system7_0/Data');
```



Caution: The 'BaseAddress' of the AXI4 interface must be a valid address in the 'MasterAddressSpace' and should not create any address conflict with other address based peripherals in the custom reference design.

Execute the SoC workflow for the Zybo board

The preceding sections discussed the steps to define and register the Zybo board and a custom reference design in the HDL Workflow Advisor for the SoC workflow. In this section, we use the custom board and reference design registration system to generate an HDL IP core that blinks LEDs on the Zybo board. The files used in the following demonstration are located at,

- matlab/toolbox/hdlcoder/hdltoderdemos/customboards/ZYBO

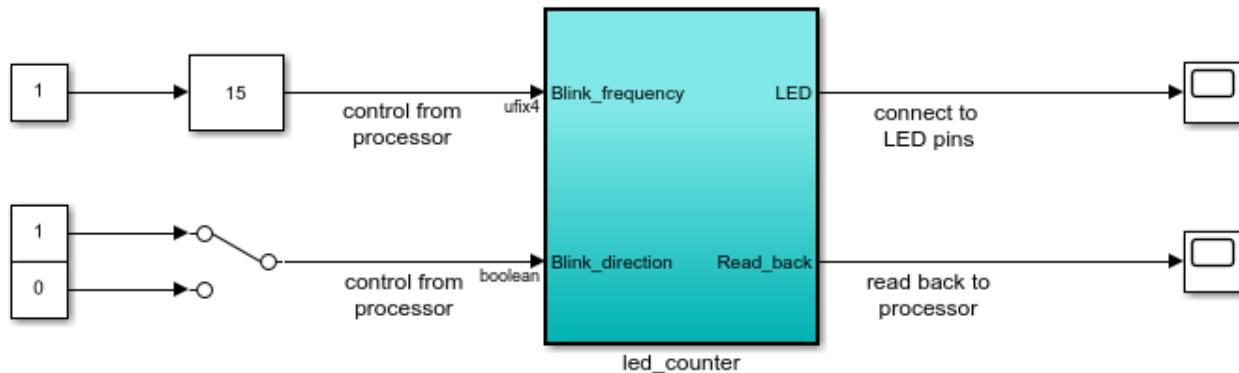
1. Add the Zybo board registration file to the MATLAB path using the command,

```
addpath(fullfile(matlabroot, 'toolbox', 'hdlcoder', 'hdltoderdemos', 'customboards', 'ZYBO'));
```

2. Open the Simulink model that implements LED blinking using the command,

```
open_system('hdlcoder_led_blinking_4bit');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking_4bit/led_counter')`

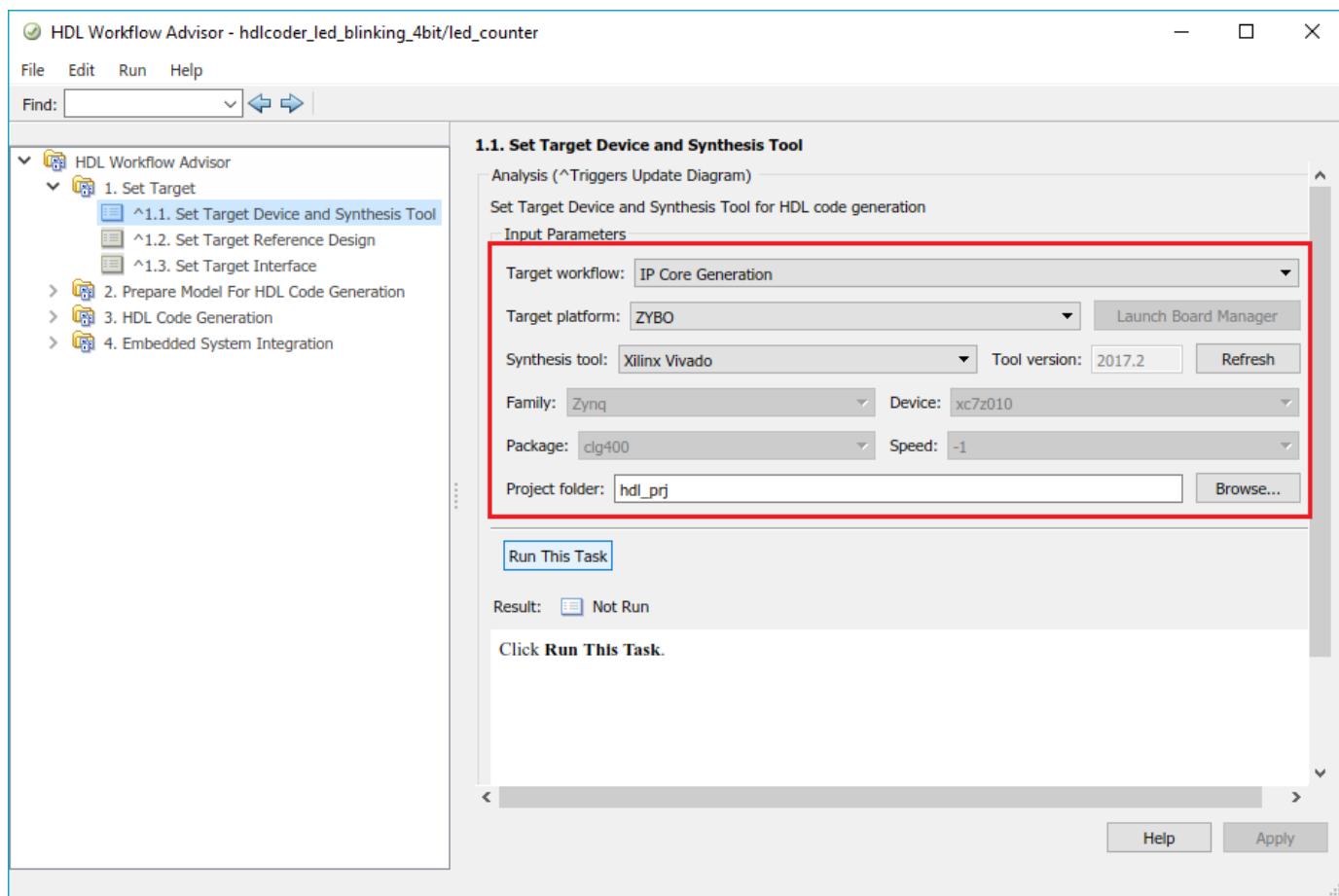
Launch HDL Workflow Advisor

Run Demo

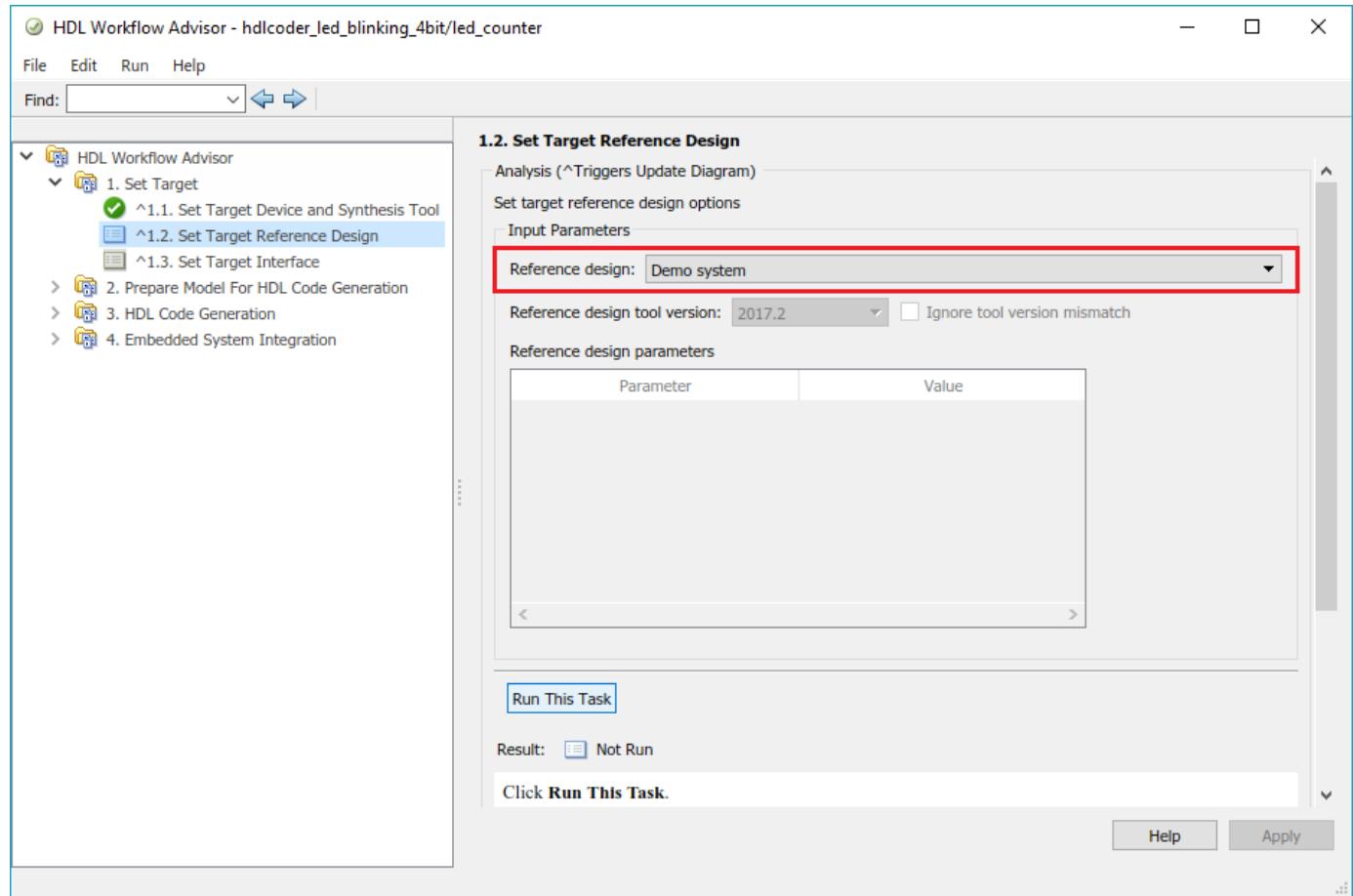
Copyright 2014-2017 The MathWorks, Inc.

3. Launch HDL Workflow Advisor from the `hdlcoder_led_blinking_4bit/led_counter` subsystem by right-clicking the `led_counter` subsystem, and selecting **HDL Code > HDL Workflow Advisor** or just click **Launch HDL Workflow Advisor** box in the model.

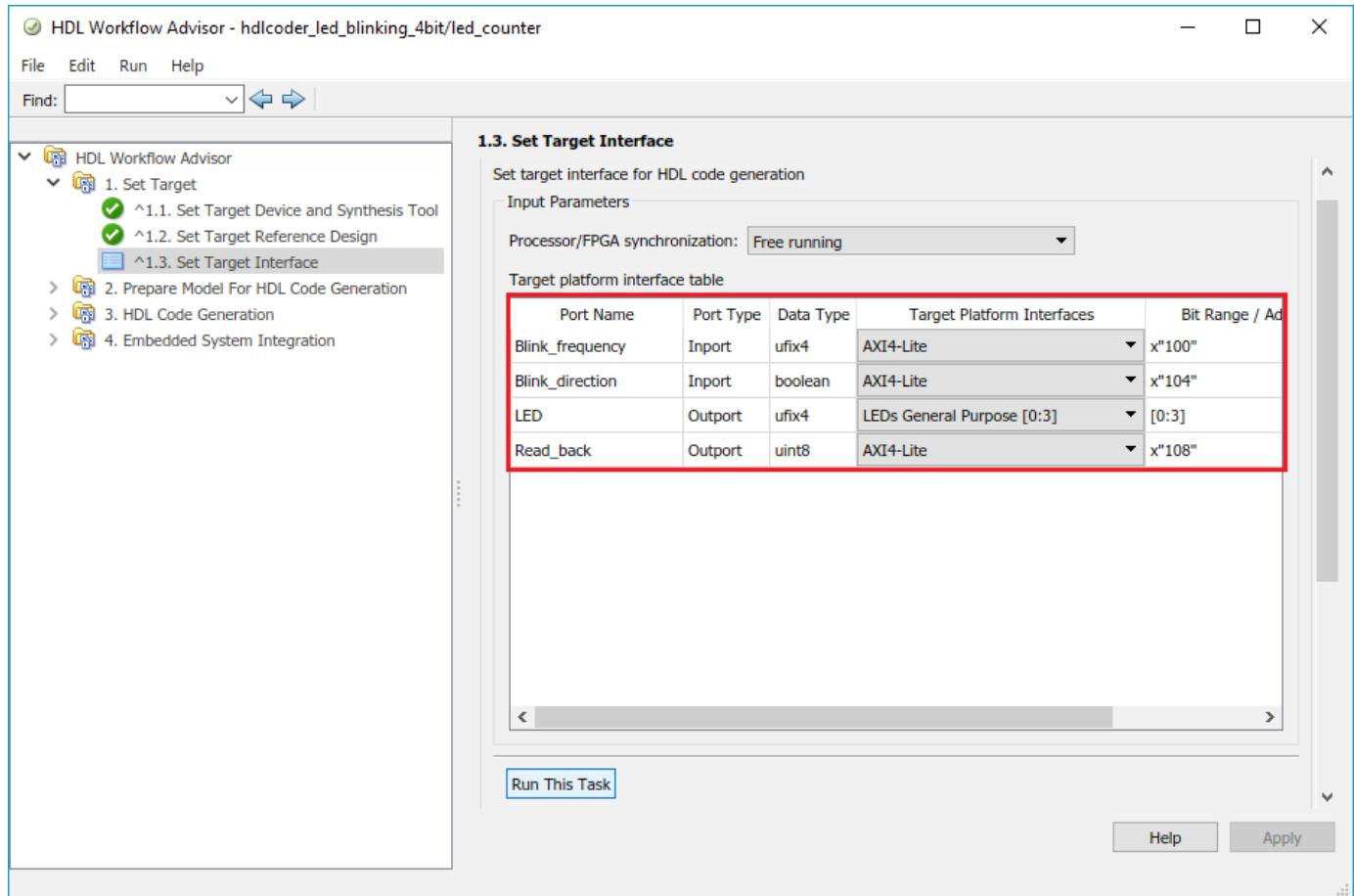
In the **Set Target > Set Target Device and Synthesis Tool** task, select **IP Core Generation** for the **Target workflow**. ZYBO now appears in the drop-down list **Target Platform**. Select ZYBO as a **Target Platform**.



4. Click **Run This Task** to complete the **Set Target Device and Synthesis Tool** task.
5. In the **Set Target > Set Target Reference Design** task, the custom reference design **Demo system** now appears against the **Reference design** field. Select **Demo system** and click on **Run This Task**.



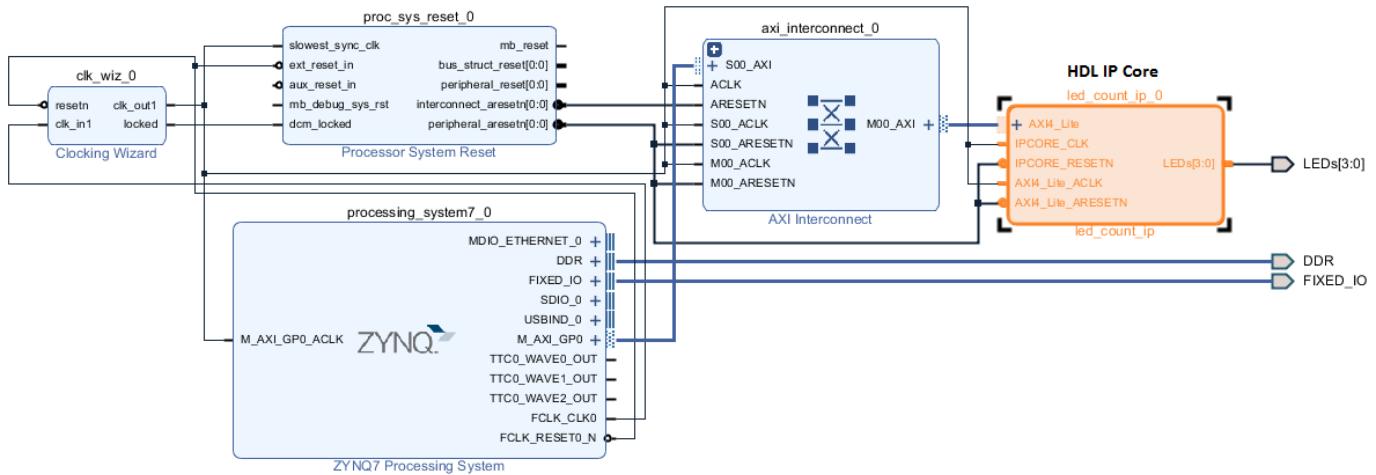
6. In Task 1.3 **Set Target Interface**, select the connections as shown in the figure below and then click on **Run This Task**.



7. Follow step 3 and step 4 of **Generate an HDL IP core using the HDL Workflow Advisor** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example to generate IP core and view the IP core generation report.

8. Follow step 1 of **Integrate the IP core with the Xilinx Vivado environment** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example to integrate the IP core in the reference design and create the vivado project.

9. Now let us examine the Xilinx Vivado project created by the SoC workflow after completing the **Create Project** task under **Embedded System Integration**. The following figure shows the block design of the SoC project where we have highlighted the HDL IP Core. It is instructive to compare this block design with the previous block design used to export the custom reference design for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



10. Follow the steps 2, 3 and 4 of **Integrate the IP core with the Xilinx Vivado environment** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example to generate software interface model, generate FPGA bitstream and program target device respectively.

11. The LEDs on the Zybo board will now start blinking after loading the bitstream. In addition, you will be able to control the LED blink frequency and direction by executing the software interface model on the Zynq ARM processor. Refer to **Generate a software interface model** section of “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65 example to control the LED blink frequency and direction from the generated software interface model.

Define Custom Board and Reference Design for Intel SoC Workflow

This example shows how to define and register a custom board and reference design in the HDL Coder™ Intel SoC workflow.

Introduction

Using this example, you will be able to register the Terasic DE1-SoC development kit and a custom reference design in the HDL Workflow Advisor for the Intel SoC workflow.

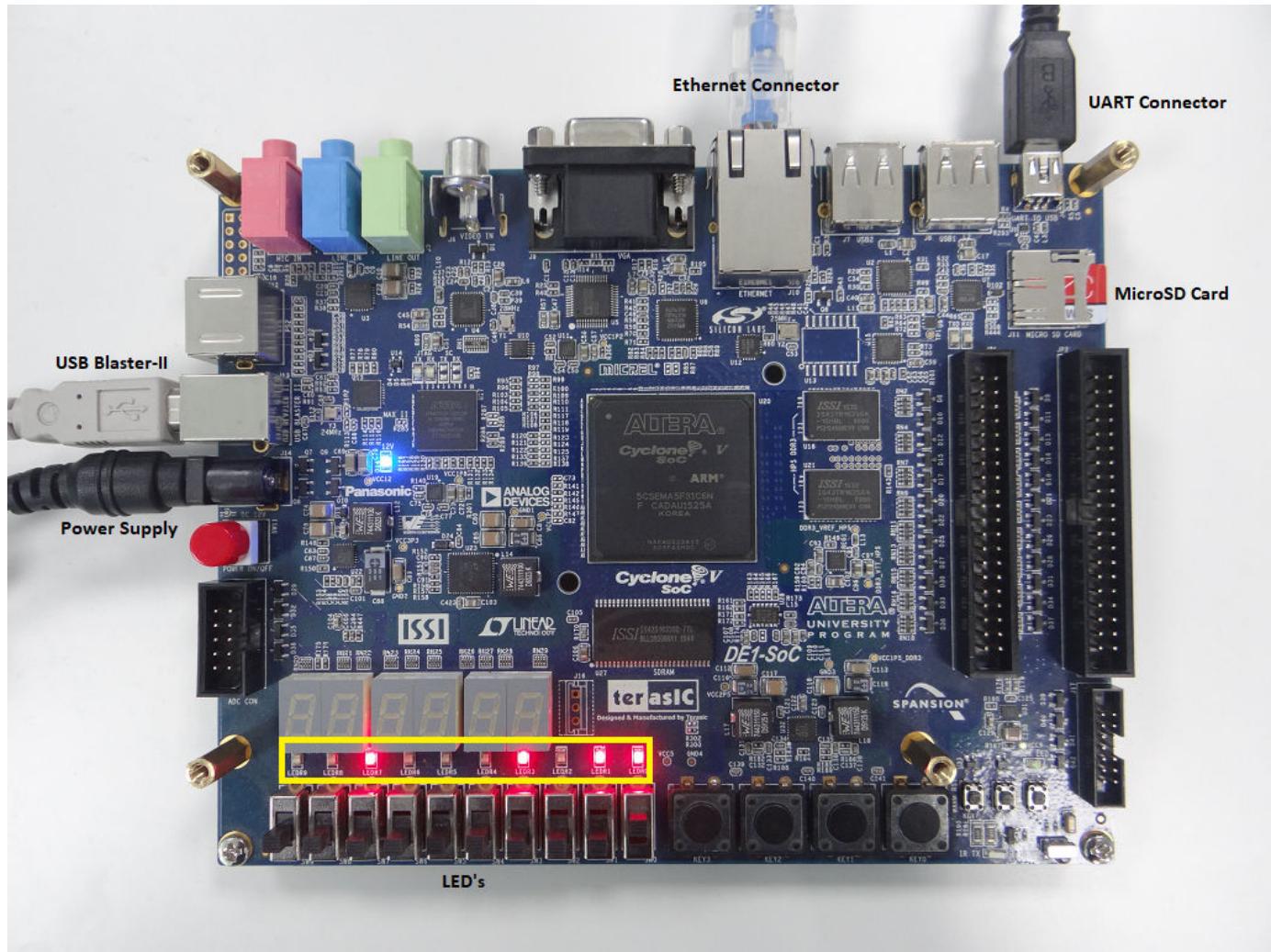
This example uses Terasic DE-1 SoC, but in the same way, you can define and register a custom board or a custom reference design for other Intel SoC devices.

Requirements

- 1 Intel Quartus Prime, with supported version listed in the HDL Coder documentation
- 2 Intel SoC Embedded Design Suite
- 3 Terasic DE1-SoC development Kit
- 4 HDL Coder Support Package for Intel SoC Devices
- 5 Embedded Coder Support Package for Intel SoC Devices

Set up Intel SoC hardware and tools

1. Understand the features available on the Terasic DE1-SoC by reading the board reference manual.
2. Set up the Terasic DE1-SoC as shown in the following figure:



3. Ensure that you have properly installed the USB COM port device drivers on your computer.
4. Connect the UART and USB blaster port on the Terasic DE1-SoC to your computer.
5. Connect the Terasic DE1-SoC to your computer using an Ethernet cable. The default Terasic DE1-SoC IP address is 192.168.1.101.
6. Download the Terasic DE1-SoC Linux image file, extract the GZ archive, and then write the raw disc image file to the microSD card. Insert the microSD card in connector J11.
7. Set up the Intel Quartus tool path by using the following command:

```
hdlsetupoolpath('ToolName', 'Altera Quartus II', 'ToolPath', 'C:\intelFPGA\17.1\quartus\bin')
```

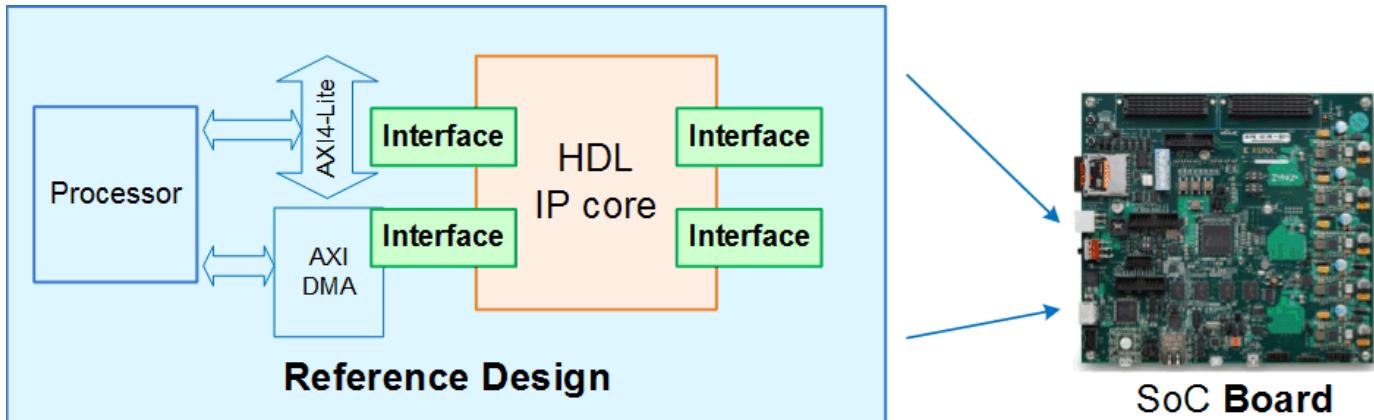
Use your own Intel Quartus installation path when executing the command.

8. Set up the Terasic DE1-SoC hardware connection by using the following command:

```
h = alterasoc('192.168.1.101','root','cyclonevsoc');
```

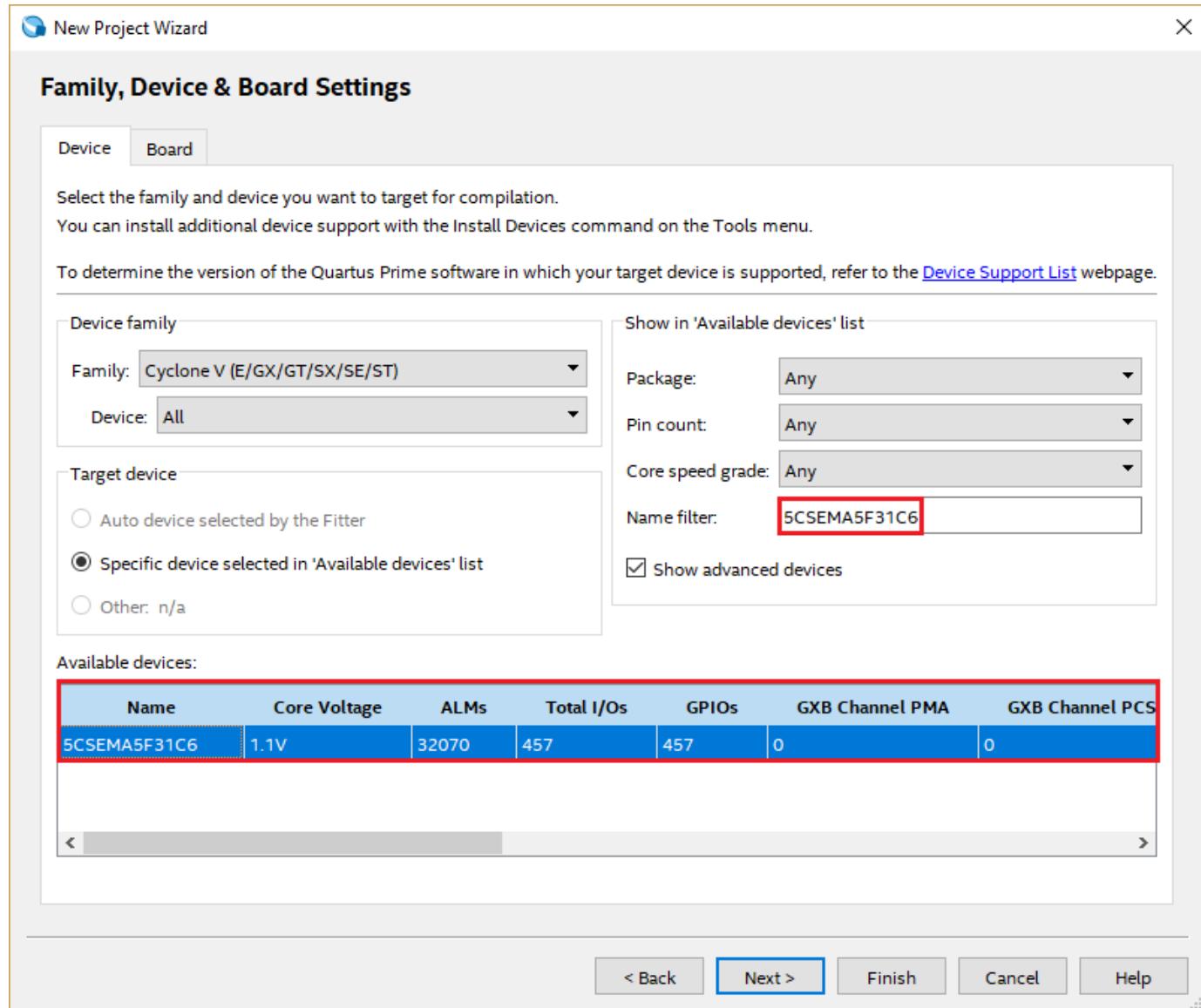
Reference Design creation using Intel Quartus Prime

A reference design captures the complete structure of an SoC design, defining the different components and their interconnections. The HDL Coder SoC workflow generates an IP core that integrates with the reference design, and is then used to program an SoC board. The following figure describes the relationship between a reference design, an HDL IP core and an SoC board.

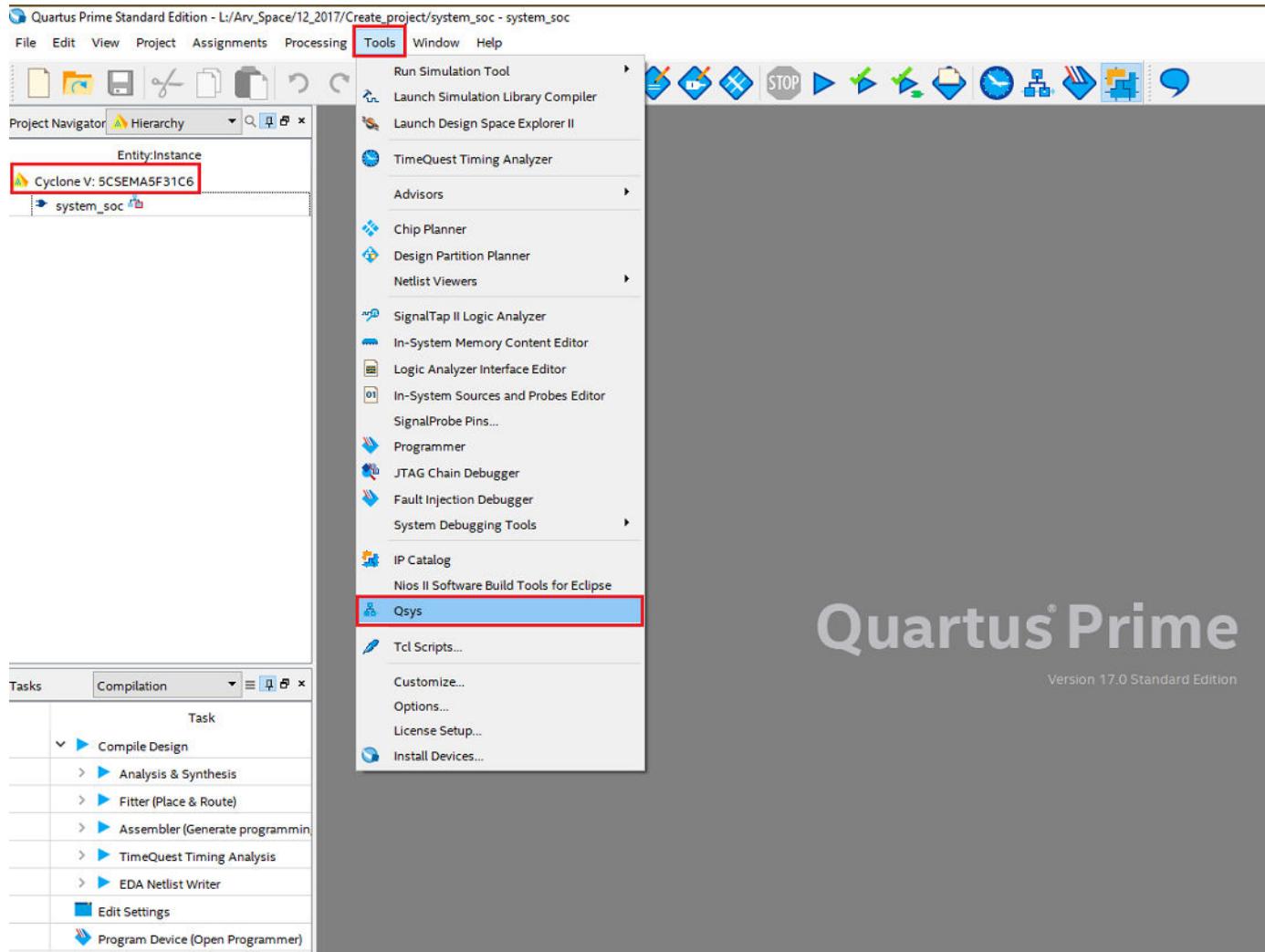


In this section, we outline the basic steps necessary to create and export a simple reference design using the Intel Quartus and QSys environment. For more information about the QSys system integration tool, refer to Altera/Intel documentation.

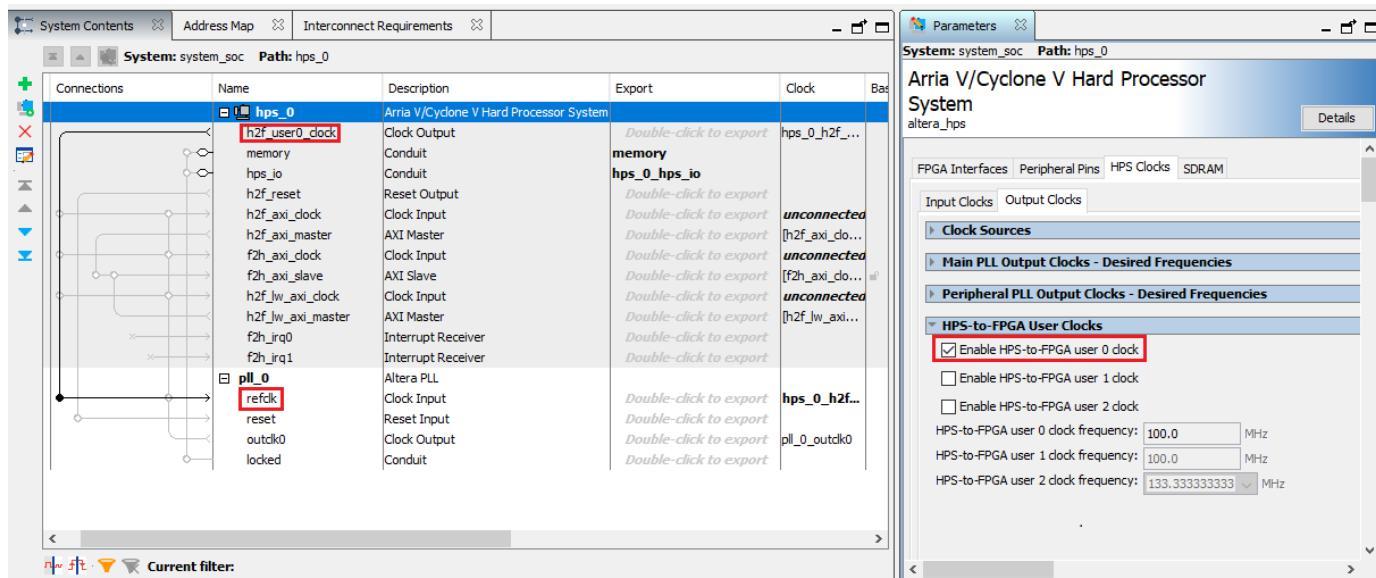
1. Create an empty Quartus project using the New project wizard with device part number as shown in the following figure



2. Initialize the Qsys in Quartus by navigating **Tools --> Qsys** as shown in the following figure

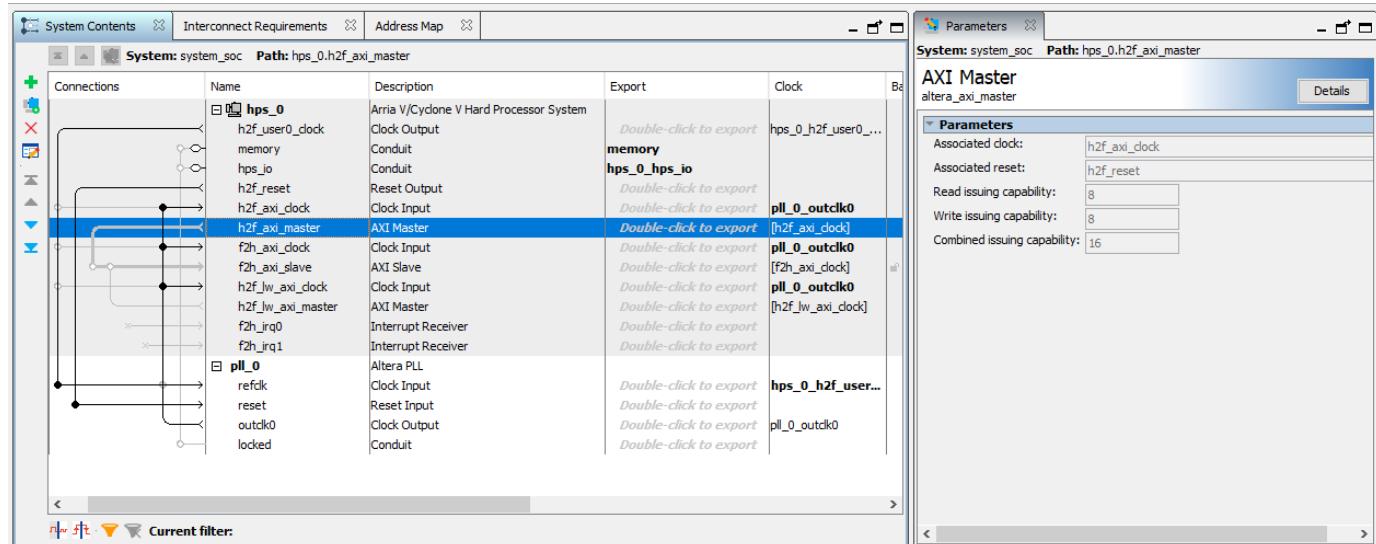


3. Select Cyclone-V Hard Processor System(HPS) & Altera PLL IP's from IP catalog to the created Qsys project. Enable HPS-to-FPGA user 0 clock (h2f_user0_clock) and connect that to refclk of Altera PLL as shown in the following figure



complete the other settings required for Hard Processor System such as Peripheral pin set and mode settings.

4. keep `h2f_axi_master` port connection open in order to connect to DUT IP during the process of workflow IP integration. Complete the rest of the connections between Altera PLL IP and HPS IP as shown in the following figure



5. Save the Qsys file. This file will be used while you create reference design plugin.

Register the DE1-SoC board in HDL Workflow Advisor

In this section, we outline the steps necessary to register the Terasic DE1-SoC development kit in HDL Workflow Advisor.

1. Create a board registration file with the name `hdlcoder_board_customization.m` and add it to the MATLAB path.

A board registration file contains a list of board plug-ins. A board plugin is a MATLAB package folder containing a board definition file and all reference design plug-ins associated with the board.

The following code describes the contents of a board registration file that contains the board plugin DE1SoCRegistration to register the Terasic DE1-SoC development kit in HDL Workflow Advisor.

```
function r = hdlcoder_board_customization
% Board plugin registration file
% 1. Any registration file with this name on MATLAB path will be picked up
% 2. Registration file returns a cell array pointing to the location of
%    the board plugin
% 3. Board plugin must be a package folder accessible from MATLAB path,
%    and contains a board definition file

r = { ...
    'DE1SoCRegistration.plugin_board', ...
};
end
```

2. Create the board definition file.

A board definition file contains information about the SoC board.

The following code describes the contents of the DE1-SoC board definition file `plugin_board.m` that resides inside the board plugin `DE1SoCRegistration`.

Information about the FPGA I/O pin locations ('`FPGAPin`') and standards ('`IOSTANDARD`') is obtained from the Pin Planner of Intel Quartus-II.

The property `BoardName` defines the name of the DE-1 SoC board as `Terasic DE1-SoC development Kit` in HDL Workflow Advisor.

```
function hB = plugin_board()
% Board definition

% Construct board object
hB = hdlcoder.Board;

hB.BoardName      = 'Terasic DE1-SoC development Kit';

% FPGA device information
hB.FPGAVendor    = 'Altera';
hB.FPGAFamily     = 'Cyclone V';
hB.FPGADevice     = '5CSEMA5F31C6';
hB.FPGAPackage    = '';
hB.FPGASpeed      = '';

% Tool information
hB.SupportedTool = {'Altera QUARTUS II'};

% FPGA JTAG chain position
hB.JTAGChainPosition = 2;

%% Add interfaces
% Standard "External Port" interface
hB.addExternalPortInterface( ...
    'IOPadConstraint', {'IO_STANDARD "2.5V"'});

% Custom board external I/O interface
hB.addExternalIOInterface( ...
```

```
'InterfaceID',    'LEDs General Purpose', ...
'InterfaceType',  'OUT', ...
'PortName',       'GPLED', ...
'PortWidth',      10, ...
'FPGAPin',        {'V16', 'W16', 'V17', 'V18', 'W17', 'W19', 'Y19', 'W20', 'W21', 'Y21'}, ...
'IOPadConstraint', {'IO_STANDARD "3.3-V LVTTL"'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'Switches', ...
    'InterfaceType',  'IN', ...
    'PortName',       'SW', ...
    'PortWidth',      10, ...
    'FPGAPin',        {'AB12', 'AC12', 'AF9', 'AF10', 'AD11', 'AD12', 'AE11', 'AC9', 'AD10', 'AE12'}, ...
    'IOPadConstraint', {'IO_STANDARD "3.3-V LVTTL"'});

hB.addExternalIOInterface( ...
    'InterfaceID',    'Push Buttons', ...
    'InterfaceType',  'IN', ...
    'PortName',       'KEY', ...
    'PortWidth',      4, ...
    'FPGAPin',        {'AA14', 'AA15', 'W15', 'Y16'}, ...
    'IOPadConstraint', {'IO_STANDARD "3.3-V LVTTL'"});
```

Register the custom reference design in HDL Workflow Advisor

In this section, we outline the steps necessary to register the custom reference design in HDL Workflow Advisor.

1. Create a reference design registration file named `hdlcoder_ref_design_customization.m` containing a list of reference design plugins associated with an SoC board.

A reference design plugin is a MATLAB package folder containing the reference design definition file and all files associated with the SoC design project. A reference design registration file must also contain the name of the associated board.

The following code describes the contents of a DE1-SoC reference design registration file containing the reference design plugin `DE1SoCRegistration.qsys_base_170` associated with the board Terasic DE1-SoC development Kit.

```
function [rd, boardName] = hdlcoder_ref_design_customization
% Reference design plugin registration file
% 1. The registration file with this name inside of a board plugin folder
% will be picked up
% 2. Any registration file with this name on MATLAB path will also be picked up
% 3. The registration file returns a cell array pointing to the location of
% the reference design plugins
% 4. The registration file also returns its associated board name
% 5. Reference design plugin must be a package folder accessible from
% MATLAB path, and contains a reference design definition file

rd = {'DE1SoCRegistration.qsys_base_170.plugin_rd', ...};
boardName = 'Terasic DE1-SoC development Kit';
end
```

2. Create the reference design definition file.

A reference design definition file defines the interfaces between the custom reference design and the HDL IP core that will be generated by the HDL Coder SoC workflow.

The following code describes the contents of the DE1-SoC reference design definition file `plugin_rd.m` associated with the board Terasic DE1-SoC development Kit that resides inside the reference design plugin `DE1SoCRegistration.qsys_base_170`. The property `ReferenceDesignName` defines the name of the reference design as `Demo system` in HDL Workflow Advisor.

```
function hRD = plugin_rd()
% Reference design definition

% Construct reference design object
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Altera QUARTUS II');

hRD.ReferenceDesignName = 'Demo system';
hRD.BoardName = 'Terasic DE1-SoC development Kit';

% Tool information
hRD.SupportedToolVersion = {'17.0', '17.1'};

%% Add custom design files
% add custom Qsys design
hRD.addCustomQsysDesign( ...
    'CustomQsysPrjFile', 'system_soc.qsys');

%% Add interfaces
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection', 'pll_0.outclk0', ...
    'ResetConnection', 'hps_0.h2f_reset',...
    'DefaultFrequencyMHz', 50, ...
    'MinFrequencyMHz', 5, ...
    'MaxFrequencyMHz', 500, ...
    'ClockModuleInstance', 'pll_0',...
    'ClockNumber', 0);

% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'hps_0.h2f_axi_master', ...
    'BaseAddress', '0x0000');
```

The DE1-SoC reference design plugin folder `DE1SoCRegistration.qsys_base_170` must contain the Qsys file `system_soc.qsys` saved previously from the Intel Quartus Prime project . The DE1-SoC reference design definition file `plugin_rd.m` identifies the SoC design project file via the following statement:

```
hRD.addCustomQsysDesign('CustomQsysPrjFile', 'system_soc.qsys');
```

In addition to the SoC design project files, `plugin_rd.m` also defines the interface connections between the custom reference design and the HDL IP core indicated in the following figure via the statements:

```
hRD.addClockInterface( ...
    'ClockConnection', 'pll_0.outclk0', ...
    'ResetConnection', 'hps_0.h2f_reset',...
    'DefaultFrequencyMHz', 50, ...
    'MinFrequencyMHz', 5, ...
```

```
'MaxFrequencyMHz',      500,...  
'ClockModuleInstance', 'pll_0',...  
'ClockNumber',         0);  
hRD.addAXI4SlaveInterface( ...  
    'InterfaceConnection', 'hps_0.h2f_axi_master', ...  
    'BaseAddress',        '0x0000');
```

Execute the SoC workflow for the Terasic DE1-SoC

The preceding sections discussed the steps to define and register the Terasic DE1-SoC and a custom reference design in the HDL Workflow Advisor for the SoC workflow. In this section, we use the custom board and reference design registration system to generate an HDL IP core that blinks LEDs on the Terasic DE1-SoC. The files used in the following demonstration are located at,

- matlab/toolbox/hdlcoder/hdclcodedemos/customboards/DE1SOC

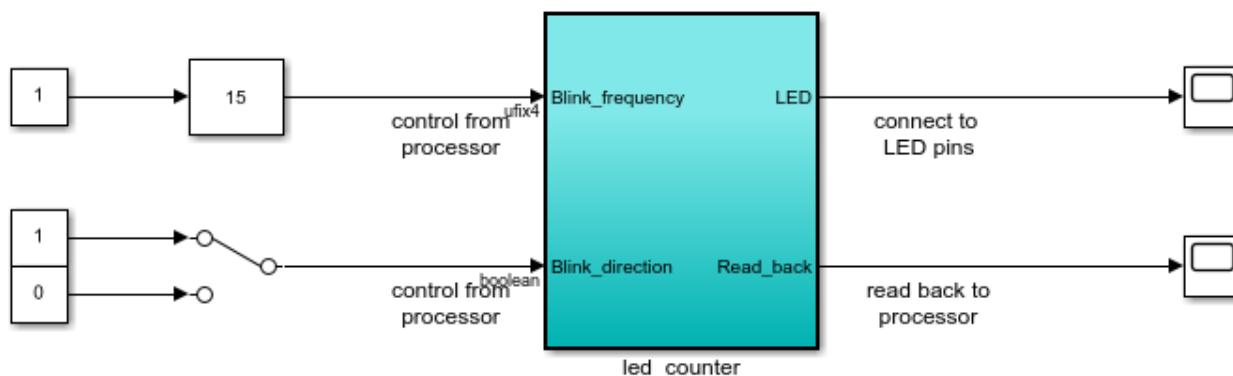
1. Add the Terasic DE1-SoC registration file to the MATLAB path using the command,

```
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdclcodedemos','customboards','DE1SOC'));
```

2. Open the Simulink model that implements LED blinking using the command,

```
open_system('hdclcoder_led_blinking');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:

```
hdladvisor('hdclcoder_led_blinking/led_counter')
```

Launch HDL Workflow Advisor

Run Demo

Copyright 2012 The MathWorks, Inc.

Generate an HDL IP core using the HDL Workflow Advisor

1. Using the IP Core Generation workflow in the HDL Workflow Advisor enables you to automatically generate a sharable and reusable IP core module from a Simulink model. HDL Coder generates HDL code from the Simulink blocks, and also generates HDL code for the AXI interface logic connecting

the IP core to the embedded processor. HDL Coder packages all the generated files into an IP core folder. You can then integrate the generated IP core with a larger FPGA embedded design in the Intel Qsys environment.

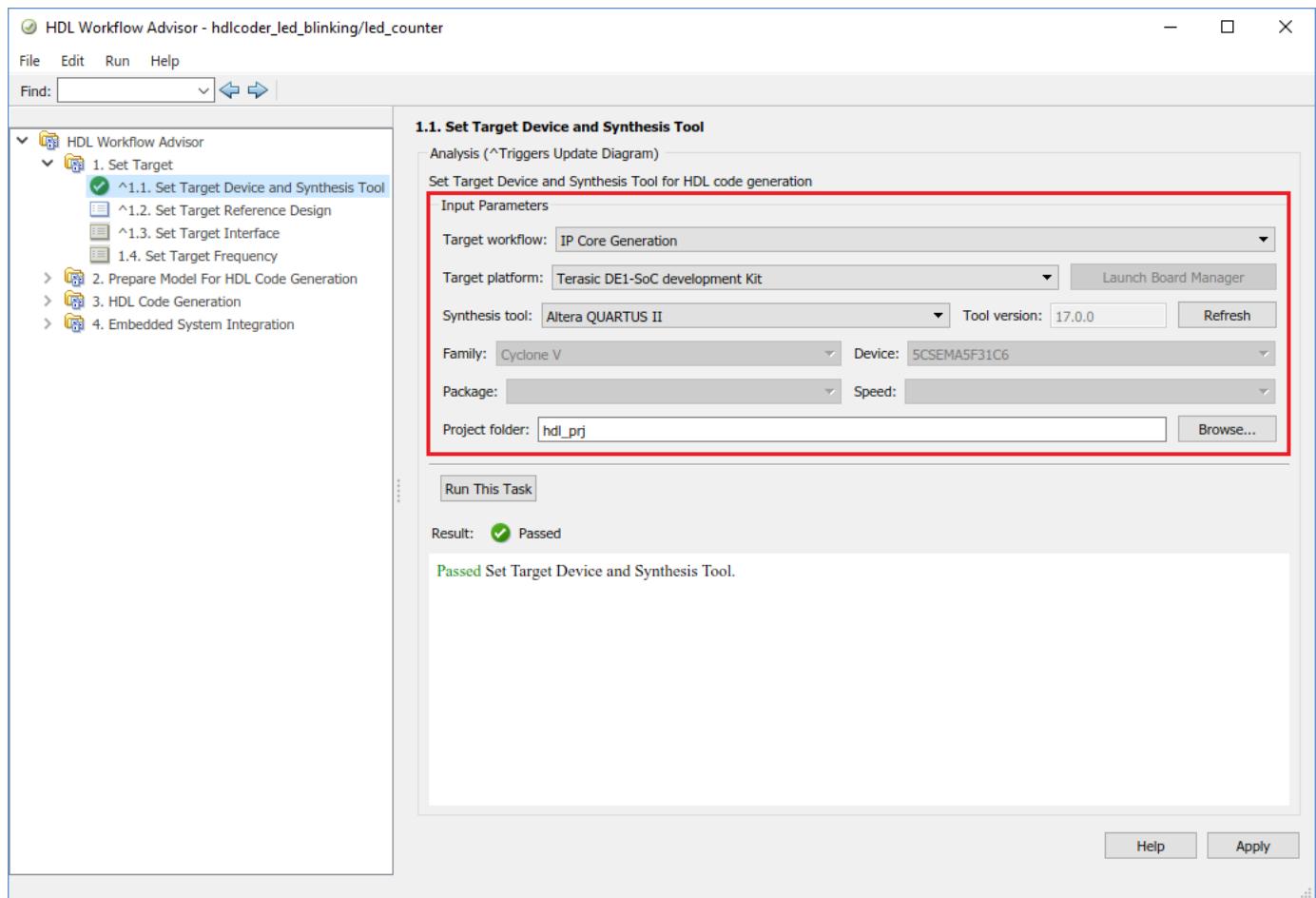
2. Start the IP core generation workflow.

2.1. Open the HDL Workflow Advisor from the `hdlcoder_led_blinking/led_counter` subsystem by right-clicking the `led_counter` subsystem, and choosing **HDL Code > HDL Workflow Advisor**.

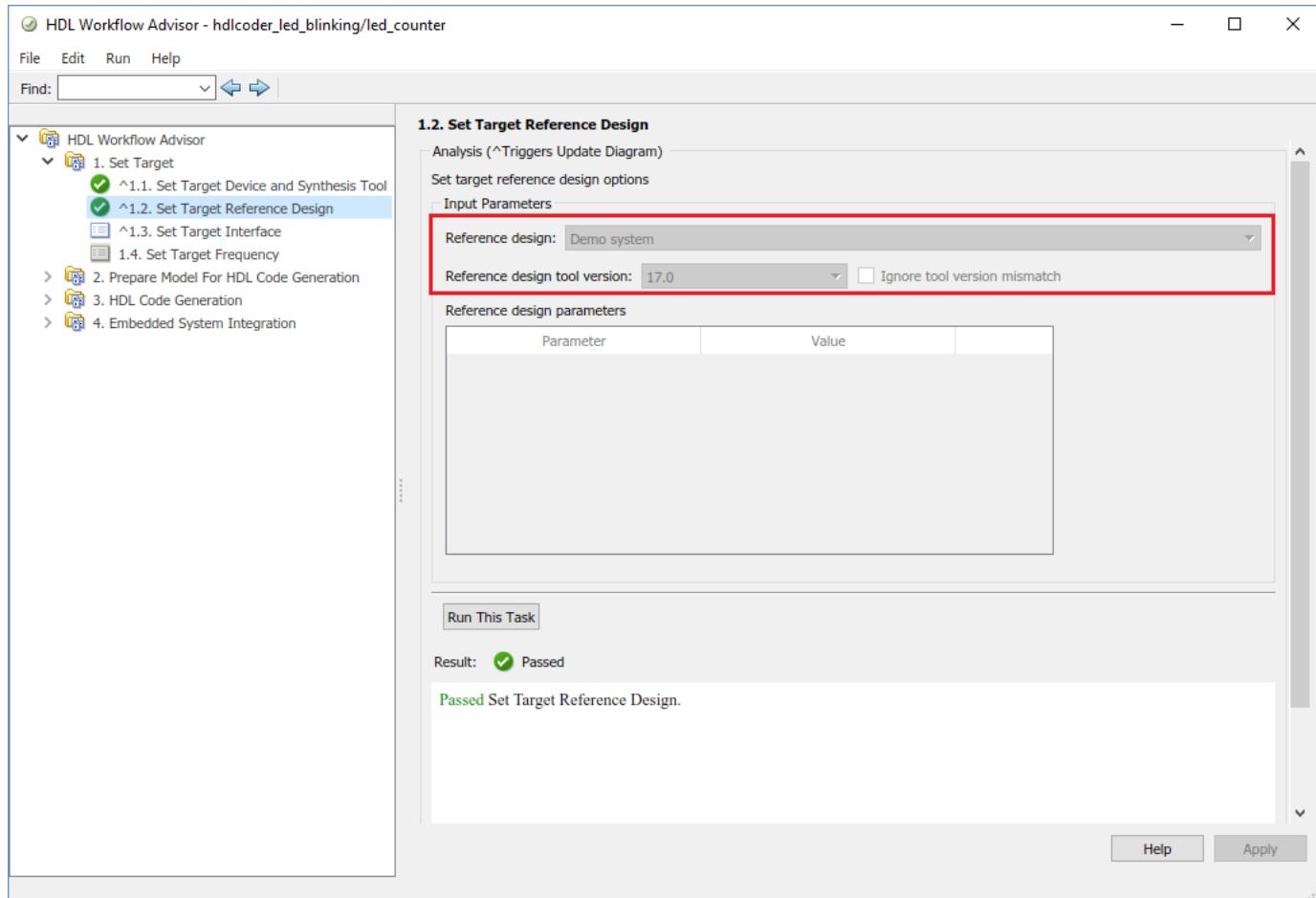
2.2. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.

2.3. For **Target platform**, select **Terasic DE1-SoC development Kit**.

2.4. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.

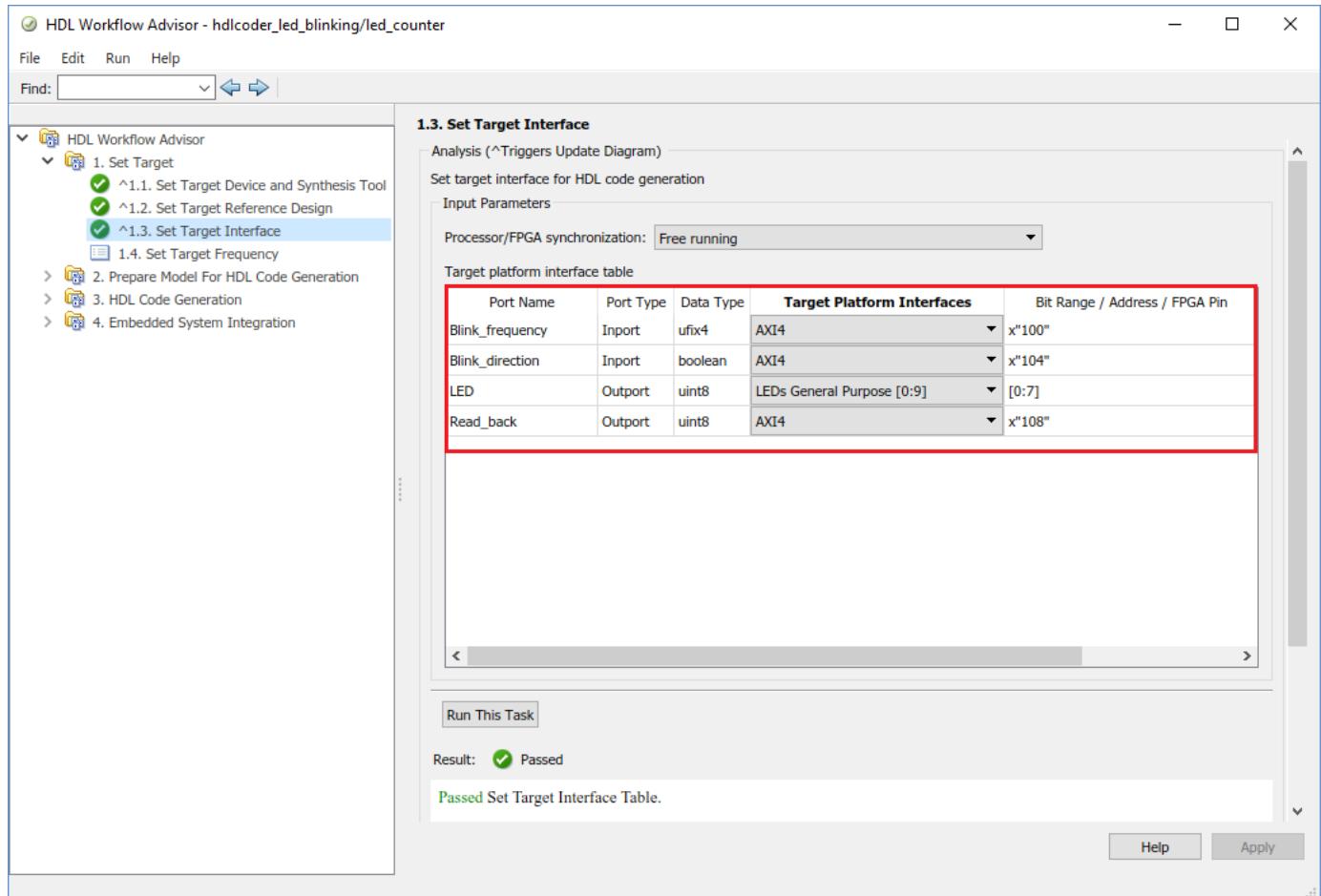


3. In the task 1.2, set target reference design default system is selected. click on **Run This Task**.



4. Configure the Target Interface.

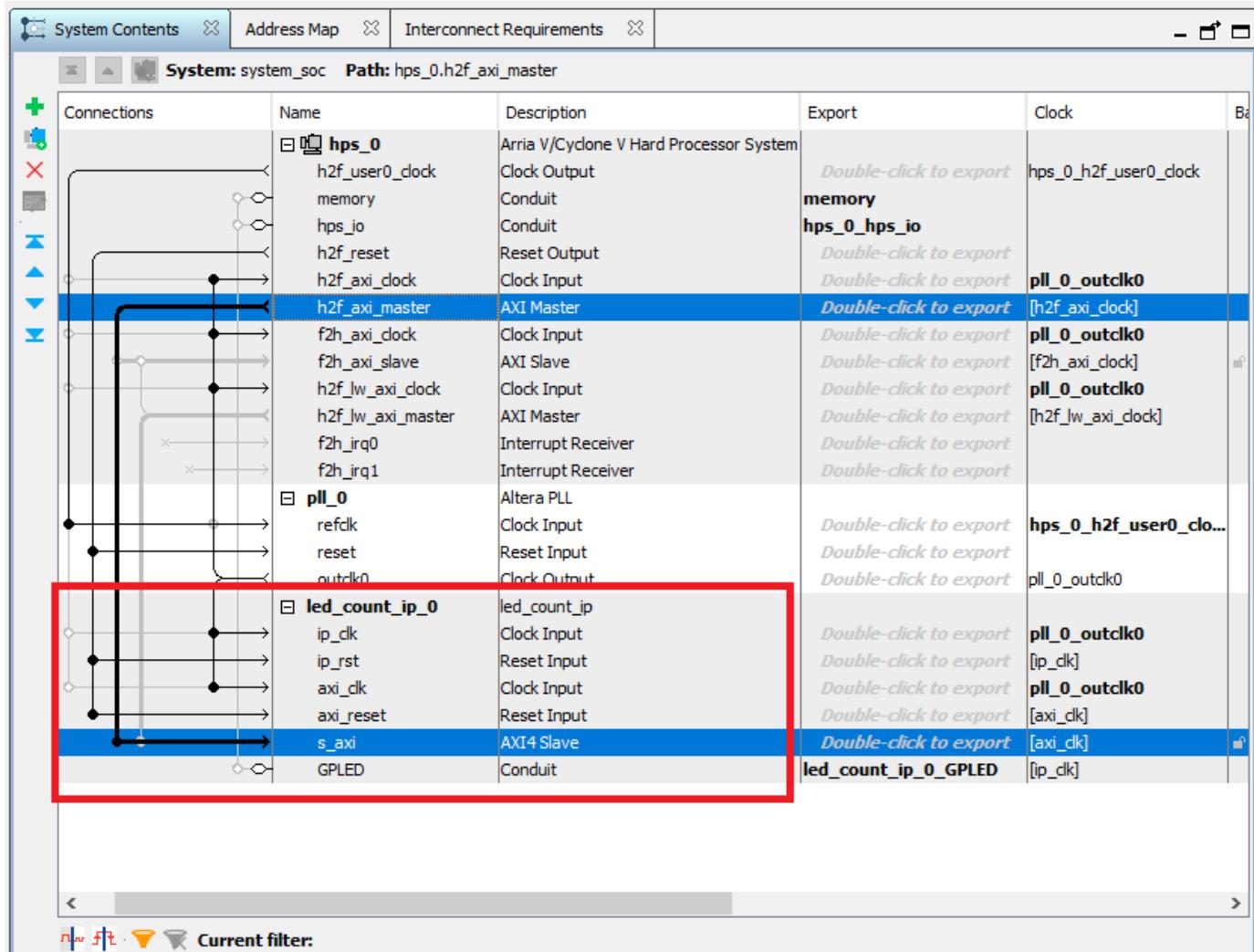
Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to **AXI4**. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:9]**, which connects to the LED hardware on the Terasic DE1-SoC development kit.



5. Follow step 3 and step 4 of **Generate an HDL IP core using the HDL Workflow Advisor** section of “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to generate IP core and view the IP core generation report.

6. Follow step 1 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to integrate the IP core in the reference design and create the Qsys project.

7. Now let us examine the Intel Qsys project created by the SoC workflow after completing the **Create Project** task under **Embedded System Integration**. The following figure shows the SoC project where we have highlighted the HDL IP Core. It is instructive to compare this project with the previous project used in the custom reference design plugin for a deeper understanding of the relationship between a custom reference design and an HDL IP Core.



8. Follow the steps 2, 3 and 4 of **Integrate the IP core with the Intel Qsys environment** section of “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to generate software interface model, generate FPGA bitstream and program target device respectively.

9. The LEDs on the Terasic DE1-SoC will start blinking after loading the bitstream. In addition, you will be able to control the LED blink frequency and direction by executing the software interface model. Refer to **Generate a software interface model** section of “Getting Started with Targeting Intel SoC Devices” on page 40-104 example to control the LED blink frequency and direction from the generated software interface model.

Dynamically Create Master Only or Slave Only or Both Slave and Master Reference Designs

This example illustrates how to customize the reference design dynamically by using a callback function based on the reference design parameter options. Also, this example shows how you can customize the number of AXI4-Stream interface channels in your reference design to be Only AXI4-Stream Master interface, Only AXI4-Stream Slave interface or both the interfaces.

Prerequisites

To run this example, you must have the following software and hardware installed and set up:

- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform
- Xilinx Vivado Design Suite, with supported version listed in “HDL Language Support and Supported Third-Party Tools and Hardware”
- Xilinx Zynq-7000 SoC ZC706 Evaluation Kit
- Latest SD image from ECoder support package setup wizard

Introduction

Instead of creating multiple reference designs that have different interface choices, data widths, or I/O plugins now you have an option of creating one reference design that has different interface choices or data widths as parameters. You can use the `CustomizeReferenceDesignFcn` method to reference the callback function that has different choices for interfaces or data widths in your reference design. For example, you can create one reference design and select the reference design parameter choice that has only AXI4-Stream Master or only AXI4-Stream Slave or both AXI4-Stream Master and AXI4-Stream Slave interfaces instead of creating three separate reference designs.

Create Reference Design Definition File and Callback Function

For this demo, you can consider Xilinx Zynq ZC706 AXI4-Stream reference design which consists of two Xilinx AXI DMAs to handle the data transfer from Processor to FPGA and vice versa using AXI4-Stream Master and Slave interfaces. In this example you customize the number of AXI4-Stream interface channels with a single reference design by creating a callback function. The different interface channels in the callback code shown below uses the different device tree. Similarly, you can modify your created reference design definition file to select different reference design parameter choices using the callback function.

The picture below shows the **Stream_Channel** reference design parameter and interface choices for the **AXI4-Stream interface with Stream channel Selection** reference design specified by using the `addParameter` method. The `CustomizeReferenceDesignFcn` method references a callback function that has the name `callback_CustomizeReferenceDesign`.

```

function hRD = plugin_rd()
hRD = hdlcoder.ReferenceDesign('SynthesisTool', 'Xilinx Vivado');
hRD.ReferenceDesignName = 'AXI4-Stream interface with Stream channel Selection';
hRD.BoardName = 'Xilinx Zynq ZC706 evaluation kit';
% Tool information
hRD.SupportedToolVersion = {'2018.2','2018.3','2019.1','2019.2'};
% add clock interface
hRD.addClockInterface( ...
    'ClockConnection',      'core_clkwiz/clk_outl', ...
    'ResetConnection',      'sys_core_rstgen/peripheral_aresetn',...
    'DefaultFrequencyMHz', 50, ...
    'MinFrequencyMHz',     5, ...
    'MaxFrequencyMHz',     500, ...
    'ClockModuleInstance', 'core_clkwiz',...
    'ClockNumber',         1);
% add AXI4 and AXI4-Lite slave interfaces
hRD.addAXI4SlaveInterface( ...
    'InterfaceConnection', 'axi_cpu_interconnect/M00_AXI', ...
    'BaseAddress',          '0x40010000', ...
    'MasterAddressSpace',   'sys_cpu/Data');

hRD.addParameter( ...
    'ParameterID',      'StreamChanel', ...
    'DisplayName',       'Stream Channel', ...
    'DefaultValue',      'Both Master & Slave', ...
    'ParameterType',     hdlcoder.ParameterType Dropdown, ...
    'Choice',           {'Master Only','Slave Only','Both Master & Slave'});
hRD.CustomizeReferenceDesignFcn = ...
    @Stream_ChannelSelection.callback_CustomizeReferenceDesign;

```

The code below shows the callback function `callback_CustomizeReferenceDesign` that has the AXI4-Stream Master or Slave Channels or both channels specified by using the `addAXI4StreamInterface` method. The `DeviceTreeName` method shown below in the callback is to specify the device tree file, which is different for different stream channels.

```

function callback_CustomizeReferenceDesign(infoStruct)
% Reference design callback run at the end of the task Set Target Reference Design

% infoStruct: information in structure format
% infoStruct.ReferenceDesignObject: current reference design registration object
% infoStruct.BoardObject: current board registration object
% infoStruct.ParameterStruct: custom parameters of the current reference design, in struct format
% infoStruct.HDLModelDutPath: the block path to the HDL DUT subsystem
% infoStruct.ReferenceDesignToolVersion: Reference design Tool Version set in 1.2 Task

hRD = infoStruct.ReferenceDesignObject;
paramStruct = infoStruct.ParameterStruct;

% get the reference design parameter value
ParamValue = paramStruct.StreamChanel;

```

```
% Add the reference design interface into interface list table baed on the
% reference design Parameter value

if strcmp(ParamValue, 'Both Master & Slave')
    % add custom Vivado design
    hRD.addCustomVivadoDesign( ...
        'CustomBlockDesignTcl', 'system_top.tcl', ...
        'VivadoBoardPart', 'xilinx.com:zc706:part0:1.0');
    hRD.addAXI4StreamInterface( ...
        'MasterChannelEnable', true, ...
        'SlaveChannelEnable', true, ...
        'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
        'SlaveChannelConnection', 'axi_dma_mm2s/M_AXIS_MM2S', ...
        'MasterChannelDataWidth', 32, ...
        'SlaveChannelDataWidth', 32, ...
        'HasDMAConnection', true);
    hRD.DeviceTreeName = 'devicetree_axistream_iio.dtb';

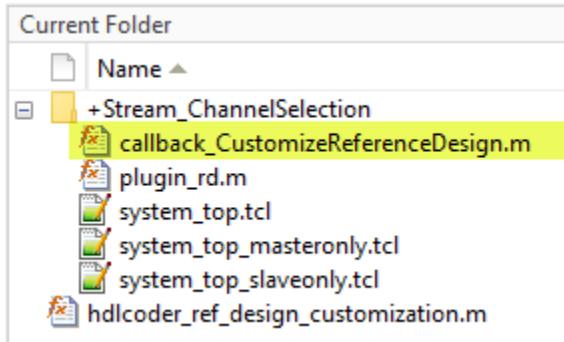
elseif strcmp(ParamValue, 'Master Only')
    % Block design TCL for Master Only reference design
    hRD.addCustomVivadoDesign( ...
        'CustomBlockDesignTcl', 'system_top_masteronly.tcl', ...
        'VivadoBoardPart', 'xilinx.com:zc706:part0:1.0');

    hRD.addAXI4StreamInterface( ...
        'MasterChannelEnable', true, ...
        'SlaveChannelEnable', false, ...
        'MasterChannelConnection', 'axi_dma_s2mm/S_AXIS_S2MM', ...
        'MasterChannelDataWidth', 32, ...
        'HasDMAConnection', true);
    hRD.DeviceTreeName = 'devicetree_axistream_MasterOnly_iio.dtb';

elseif strcmp(ParamValue, 'Slave Only')
    % Block design TCL for Slave Only reference design
    hRD.addCustomVivadoDesign( ...
        'CustomBlockDesignTcl', 'system_top_slaveonly.tcl', ...
        'VivadoBoardPart', 'xilinx.com:zc706:part0:1.0');

    % add AXI4-Stream Slave only interface
    hRD.addAXI4StreamInterface( ...
        'MasterChannelEnable', false, ...
        'SlaveChannelEnable', true, ...
        'SlaveChannelConnection', 'axi_dma_mm2s/M_AXIS_MM2S', ...
        'SlaveChannelDataWidth', 32, ...
        'HasDMAConnection', true);
    hRD.DeviceTreeName = 'devicetree_axistream_SlaveOnly_iio.dtb';
end
end
```

So, create the callback function like as shown above and pass the infoStruct argument to the callback function. The argument contains reference design customization information in a structure format. Save the created callback function in the reference design folder like as shown below or you can save anywhere and add the file to MATLAB path.



Generate HDL IP core with Only AXI4-Stream Master/Only AXI4-Stream Slave Interface

1. Set up the Xilinx Vivado synthesis tool path using the following command in the MATLAB command window. Use your own Vivado installation path when you run the command.

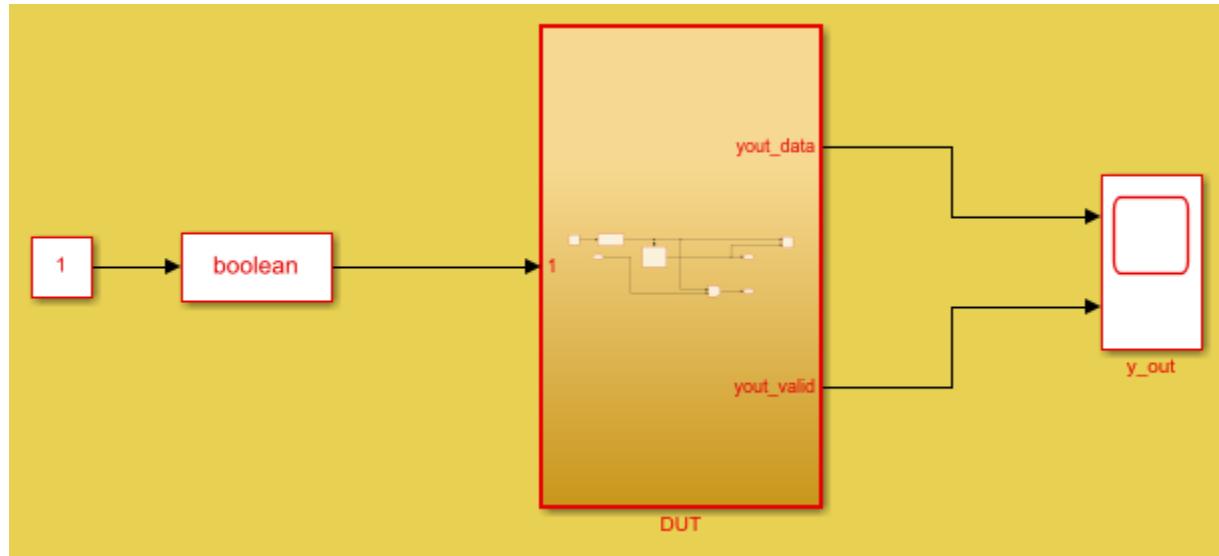
```
hdlsetupoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2019.2\bin\vivado.bat')
```

2. Add the demo reference design folder to the MATLAB path using following command:

```
addpath(fullfile(matlabroot,'toolbox','hdlcoder','hdlcoderdemos','customboards','ZC706'));
```

3. Open AXI4-Stream Master only model using following command:

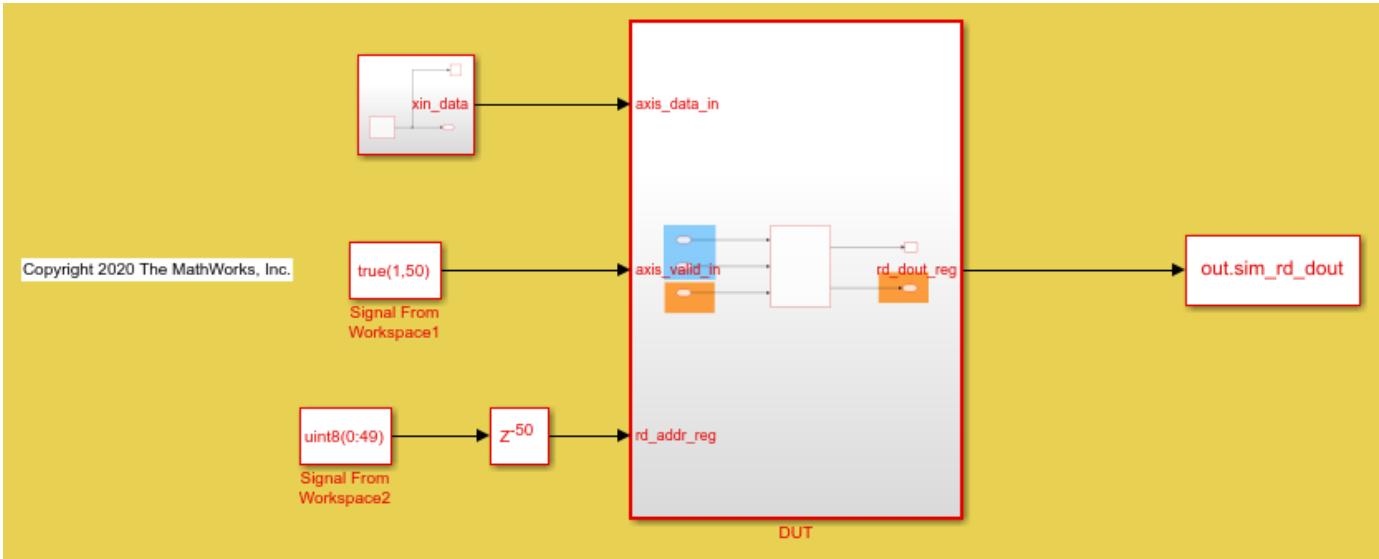
```
open_system('hdlcoder_AXI4StreamMaster');
```



The **DUT** is the hardware subsystem targeting the FPGA fabric. Inside this DUT, the **HDL Counter** subsystem acts as master. This counter counts from 1 to 50 and is connected to **yout_data** output signal which is mapped to AXI4-Stream master interface.

You can also use AXI4 Slave only model. Use the following command to open the model:

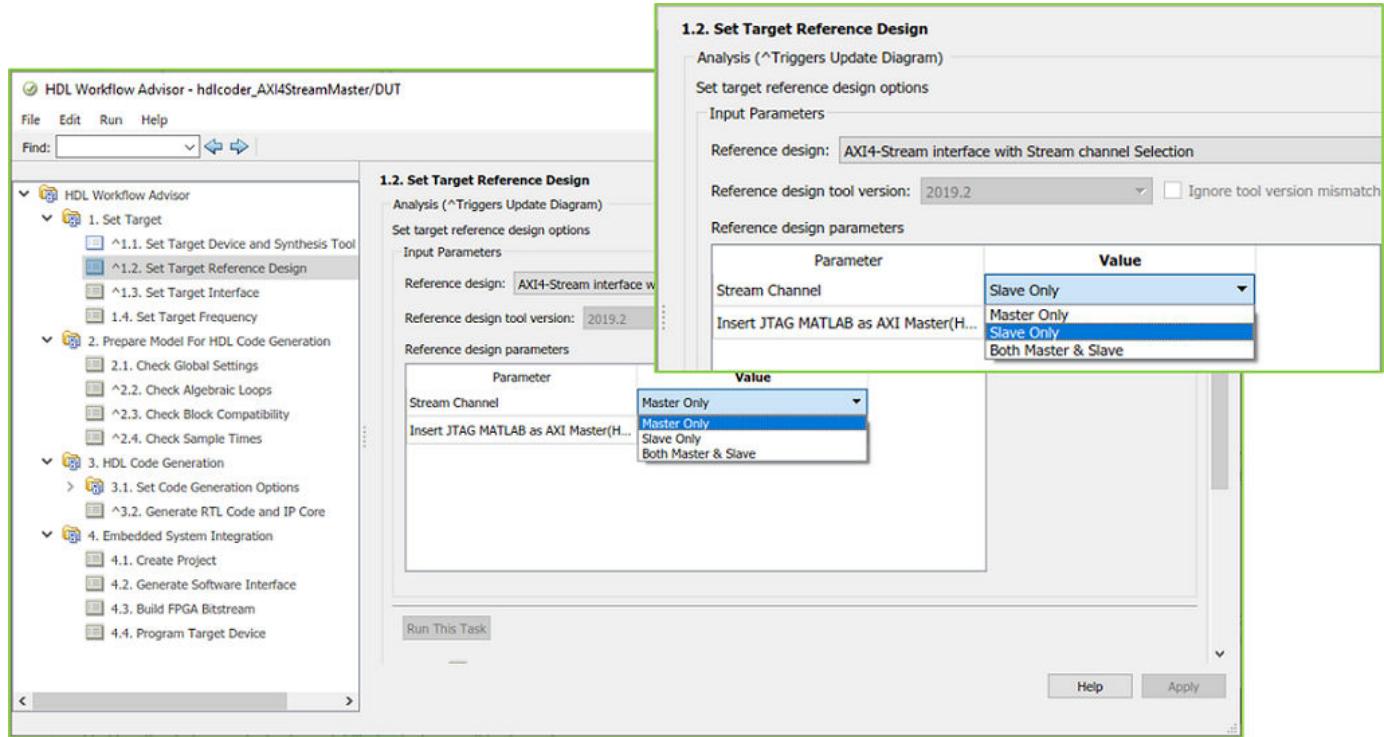
```
open_system('hdlcoder_AXI4StreamSlave');
```



As shown above **DUT** has a **Dual Port RAM** which acts as slave and receives data using AXI4-Stream Slave interface through **axis_data_in** input signal.

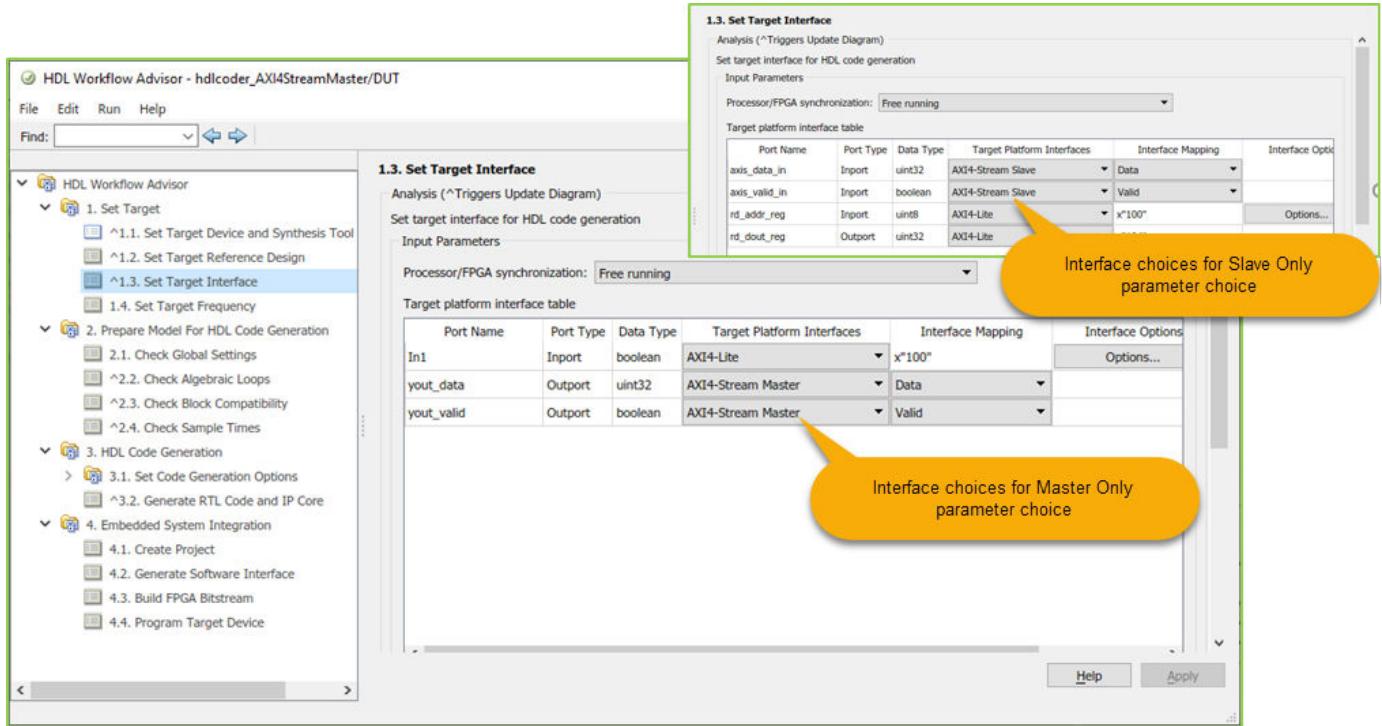
4. Start the HDL Workflow Advisor from the DUT subsystem, `hdlcoder_AXI4StreamMaster/DUT` for AXI4-Stream Master only demo. Similarly Open HDL Workflow Advisor from the DUT subsystem, `hdlcoder_AXI4StreamSlave/DUT` for AXI4-Stream Slave only demo.

The target interface settings are already saved for **ZC706** in these models, so the settings in Task 1.1 to 1.3 are automatically loaded. In Task 1.1, **IP Core Generation** is selected for Target workflow, and **Xilinx Zynq ZC706 evaluation kit** is selected for Target platform. In task 1.2, **AXI4-Stream interface with Stream channel Selection** is selected for reference design. Select **Stream Channel** reference design parameter choice as **Master Only** for AXI4-Stream Master only demo and choose **Slave Only** as parameter choice for AXI4-Stream Slave only demo.



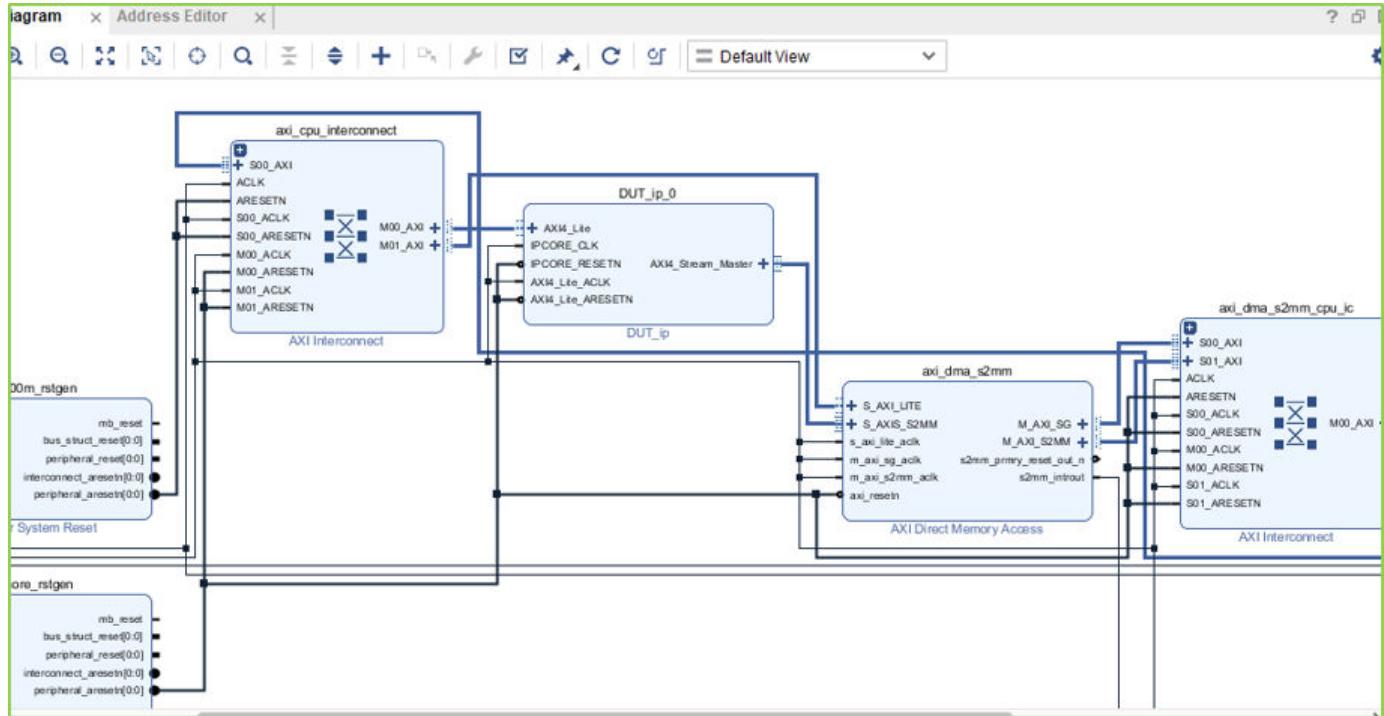
As shown in the figure above you can customize the reference design to Only AXI4-Stream Master or Only Slave or both AXI4-Stream Master and Slave by selecting the **Stream Channel** reference design parameter choice. Callback function with corresponding Tcl gets evaluated at the end of the **Set Target Reference Design** task.

5. If the parameter choice selected as **Master Only**, then the interface choice in task 1.3 shows as **AXI4 Stream Master**. Here, the AXI4-Stream interface communicates in master mode and sends data to AXI4_Stream Slave IP through **yout_data** signal. Similarly, If **Slave Only** parameter choice is selected, then the interface choice shows as **AXI4 Stream Slave**. Where, the AXI4-Stream interface communicates in slave mode and receives data through **axis_data_in** signal as shown below.

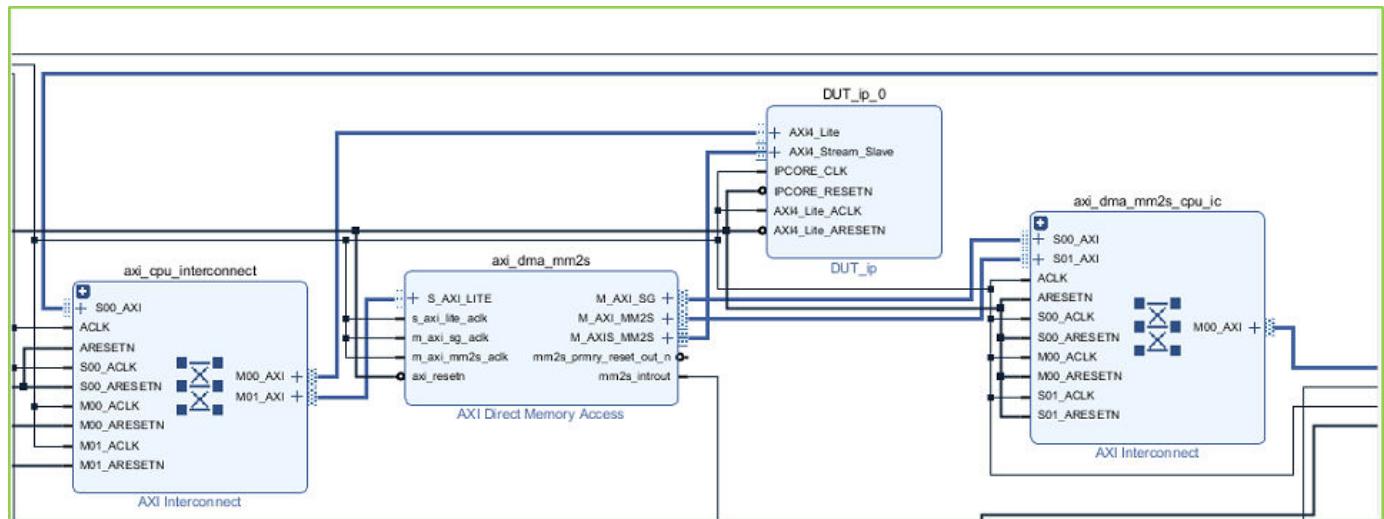


6. Right-click task 4.1, **Create Project**, and select **Run to Selected Task** to insert the generated IP core into the **AXI4-Stream interface with Stream channel Selection** reference design. The reference design contains Xilinx AXI DMA IP to handle the data streaming between FPGA fabric and processor or vice versa based on the reference design interface either Only AXI4-Stream master or only AXI4-Stream Slave.

Following diagram shows the generated vivado project with **AXI4-Stream Master only** interface choice, and you can see the connection between HDL coder generated DUT IP and slave to memory mapped Xilinx AXI DMA IP. In this reference design, DMA Controller reads the data from FPGA IP.



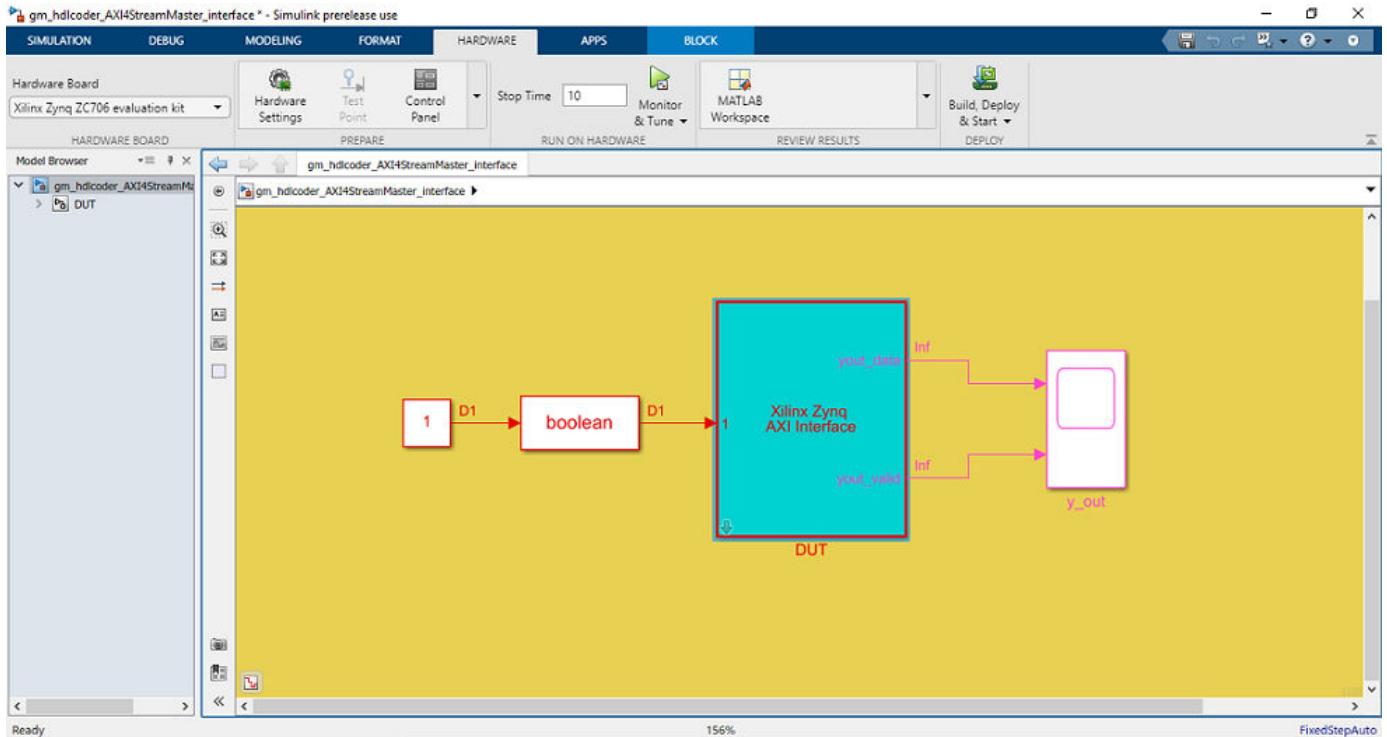
Similarly, following diagram shows the generated vivado project with **AXI4-Stream Slave only** interface choice, where the FPGA IP receives streaming data from DMA Controller.



7. In the HDL Workflow Advisor, run the rest of the tasks to generate the software interface model, and build and download the FPGA bitstream.

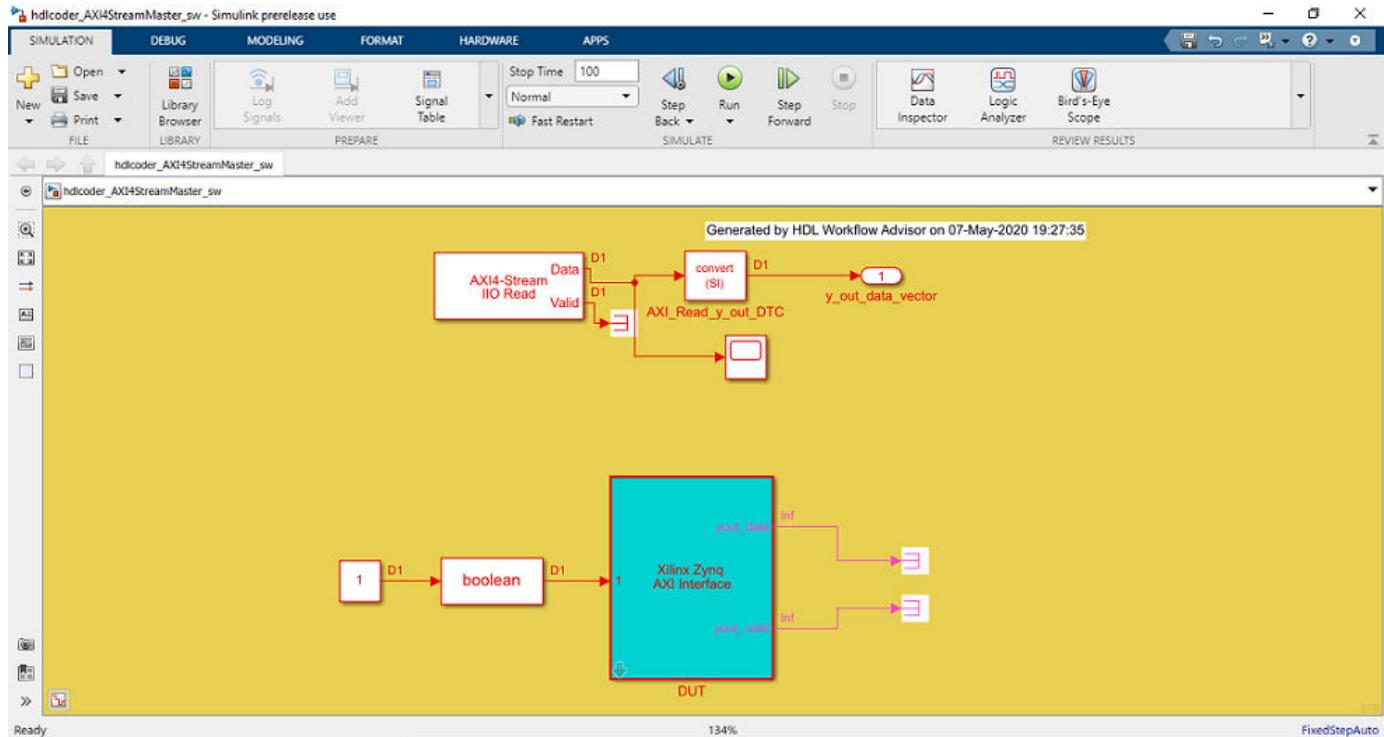
Generate ARM executable Using AXI4-Stream Driver Block for AXI4-Stream Master only reference design

A software interface model is generated in Task 4.2, **Generate Software Interface Model**, as shown in the following picture.

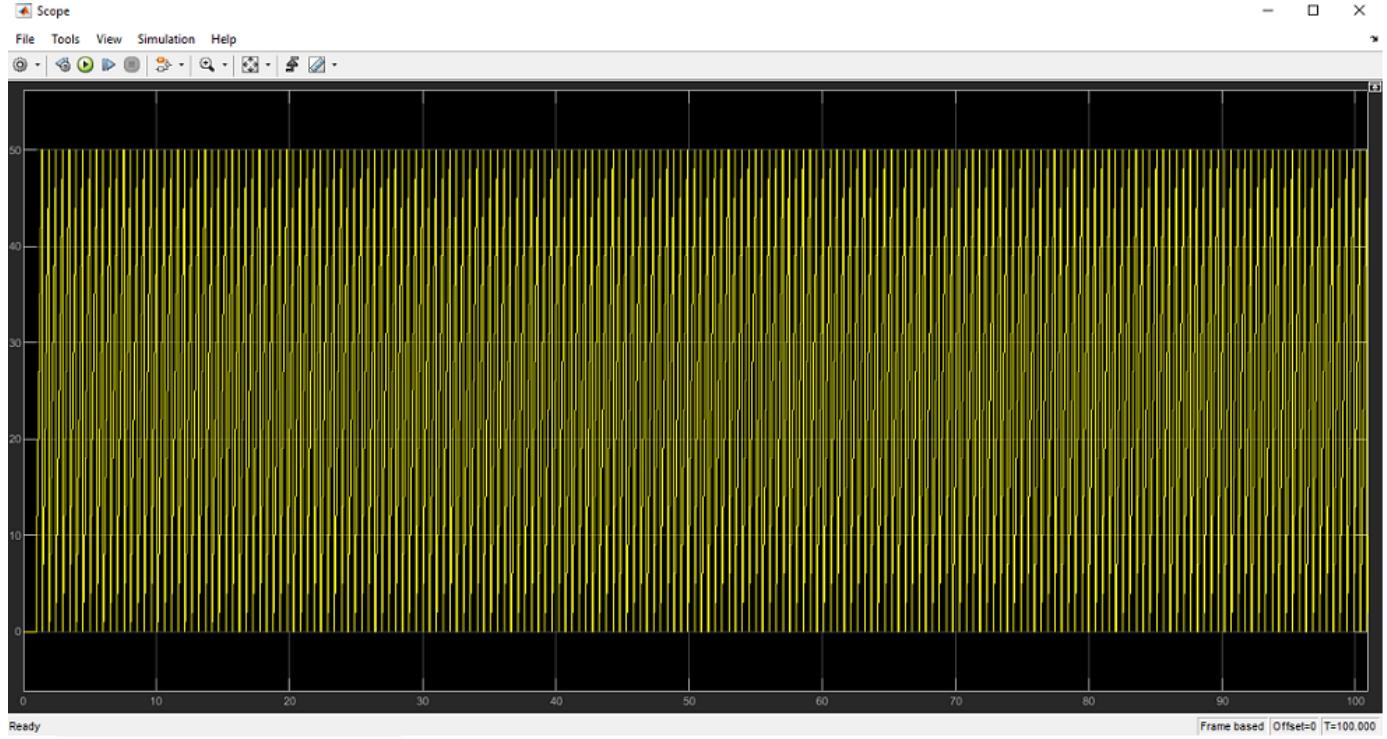


The AXI4-Stream IIO driver block cannot be automatically generated in the software interface model when a scalar port **yout_data** is mapped to AXI4-Stream interface "AXI4-Stream Master". Before you generate code from the software interface model, add the AXI4-Stream IIO Read driver block from the **Embedded Coder Support Package for Xilinx Zynq Platform** Library in the Simulink Library Browser.

The updated software interface model for AXI4-Stream Master only reference designs are provided: **hdlcoder_AXI4StreamMaster_sw.slx** Continuous data of count 0 to 50 is used in this model, and is connected to AXI4 Stream DMA driver block.

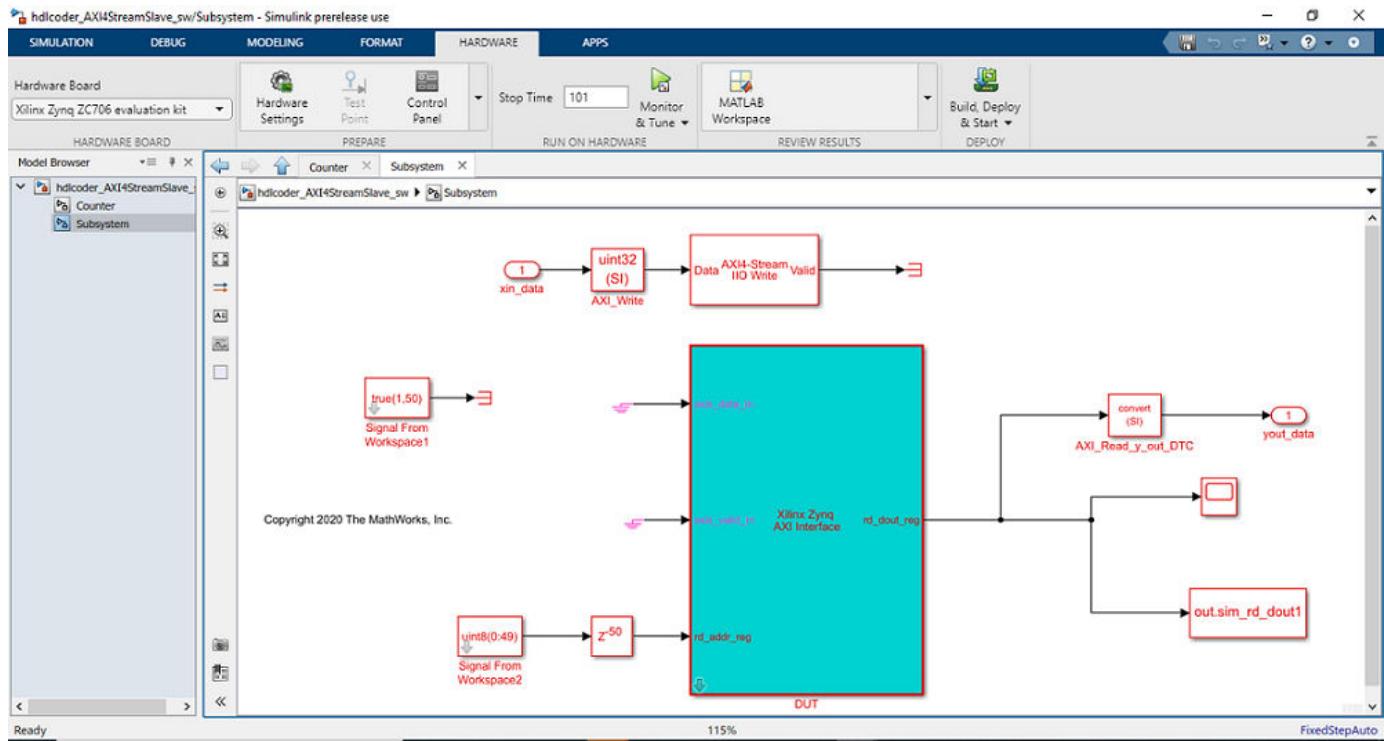


Click the **Monitor & Tune** button on the **Hardware** tab of Simulink Toolstrip. Embedded Coder builds the model, downloads the ARM executable to the Zc706 hardware. Now, both the hardware and software parts of the design are running on Zynq hardware. The FPGA IP sends the source data through the DMA controller and the AXI4-Stream interface. The ARM processor receives the data from the FPGA IP, and sends the result data to Simulink via external mode. Observe the output from the Zynq hardware on the time Scope *y_out*.

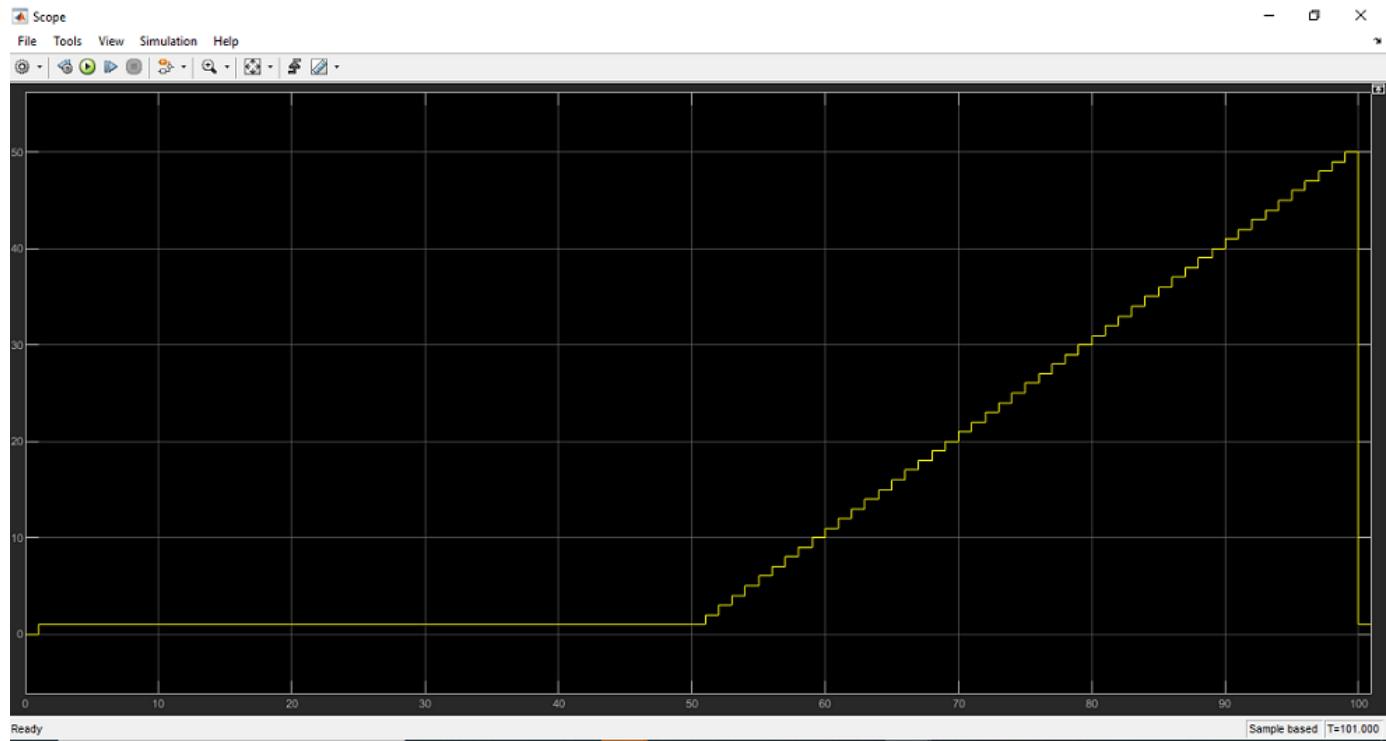


Generate ARM executable Using AXI4-Stream Driver Block for AXI4-Stream Slave only reference design

The updated software interface model for only AXI4-Stream Slave reference design is provided: `hdlcoder_AXI4StreamSlave_sw.slx`. In this model, a counter block which generates 1 to 50 incremental data is connected to AXI4-Stream Write DMA driver block. This means the DMA controller will stream count data samples to the HDL IP core via the AXI4-Stream Slave interface.



Click the **Monitor & Tune** button on the Hardware tab of model toolbar. Embedded Coder builds the model, downloads the ARM executable to the Zc706 hardware. Now, both the hardware and software parts of the design are running on Zynq hardware. The ARM processor sends the source data to the FPGA IP, through the DMA controller and the AXI4-Stream interface. Observe the output of the IP core from the Zynq hardware on the Time Scope *y_out*.



Using JTAG MATLAB as AXI Master to control HDL Coder generated IP Core

This example illustrates how to automatically insert the JTAG MATLAB as AXI Master IP into your reference design, and use MATLAB to prototype your HDL Coder generated FPGA IP Core.

Introduction and Prerequisites

To access onboard memory locations and quickly probe or control the FPGA logic from MATLAB, use the JTAG MATLAB as AXI master IP. The object connects to the IP over a physical JTAG cable, and allows read and write commands to slave memory locations from the MATLAB command line.

You can insert the JTAG MATLAB as AXI Master IP when running the **Set Target Reference Design** task of the **IP Core Generation** workflow.

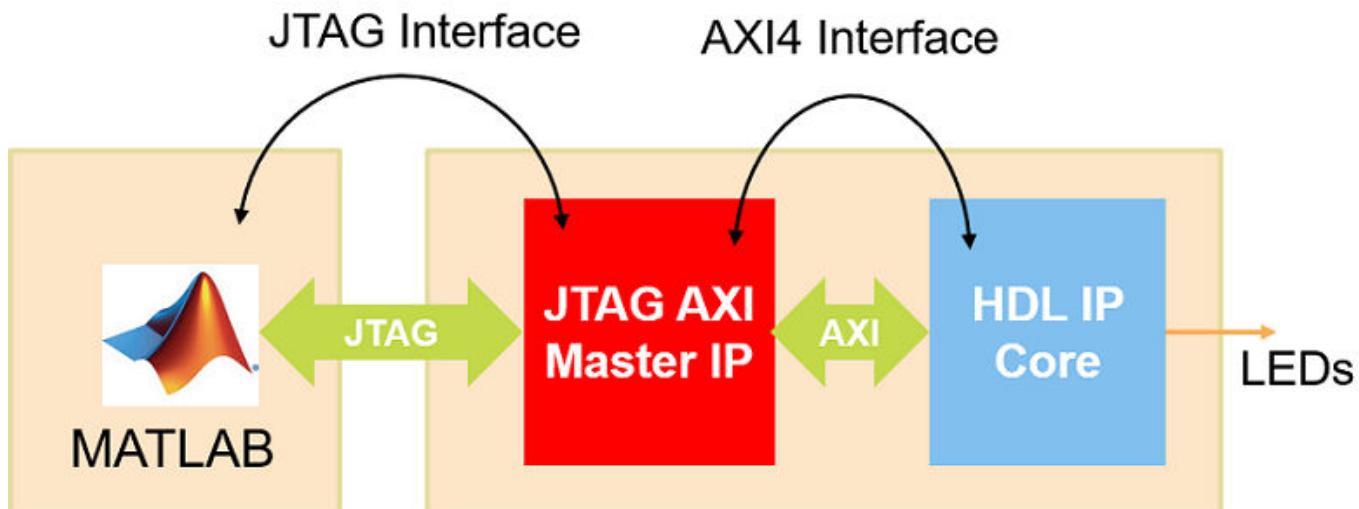
To use this capability:

- You must have the HDL Verifier™ hardware support packages installed and downloaded.
- You must not target standalone boards that do not have the hRD.addAXI4SlaveInterface or boards that are based on Xilinx ISE.

This example uses the ZedBoard™. Before you run the workflow, you must:

- 1 Install Xilinx Vivado™ Design Suite, with supported version listed in the HDL Coder documentation
- 2 Setup the Zynq board for the JTAG MATLAB as AXI Master IP insertion. To learn how to set up the ZedBoard, refer to the Set up Zynq hardware and tools section in the Getting Started with HW/SW Co-design Workflow for Xilinx Zynq Platform example.
- 3 Download and install the HDL Verifier hardware support package for Xilinx FPGA Boards. See setup and configuration section in HDL Verifier Support Package for Xilinx FPGA Boards.
- 4 Set up the path to the synthesis tool by using `hdlsetuptoolpath`. as shown below:

```
hdlsetuptoolpath('ToolName', 'Xilinx Vivado', 'ToolPath', 'C:\Xilinx\Vivado\2018.3\bin\vivado.bat')
```

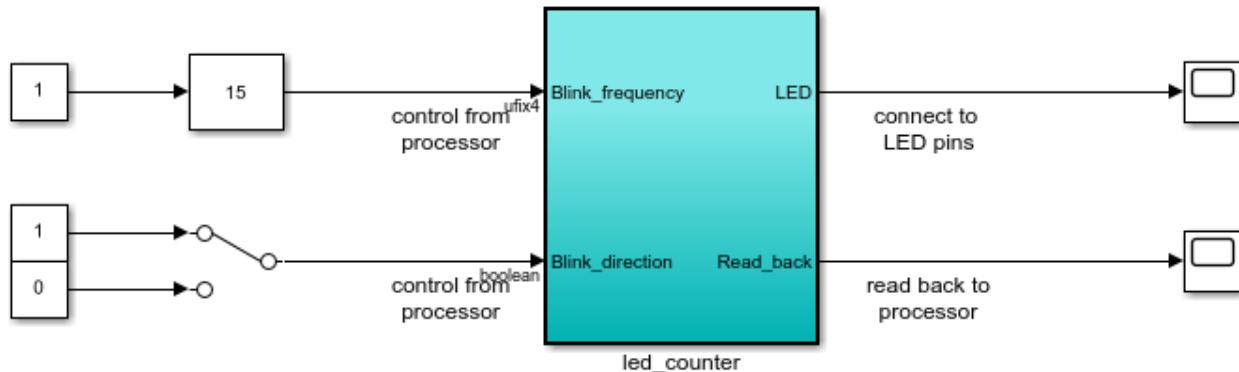


Enable Insertion of JTAG MATLAB as AXI Master

1. Open hdlcoder_led_blinking demo using following command:

```
open_system('hdlcoder_led_blinking')
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:

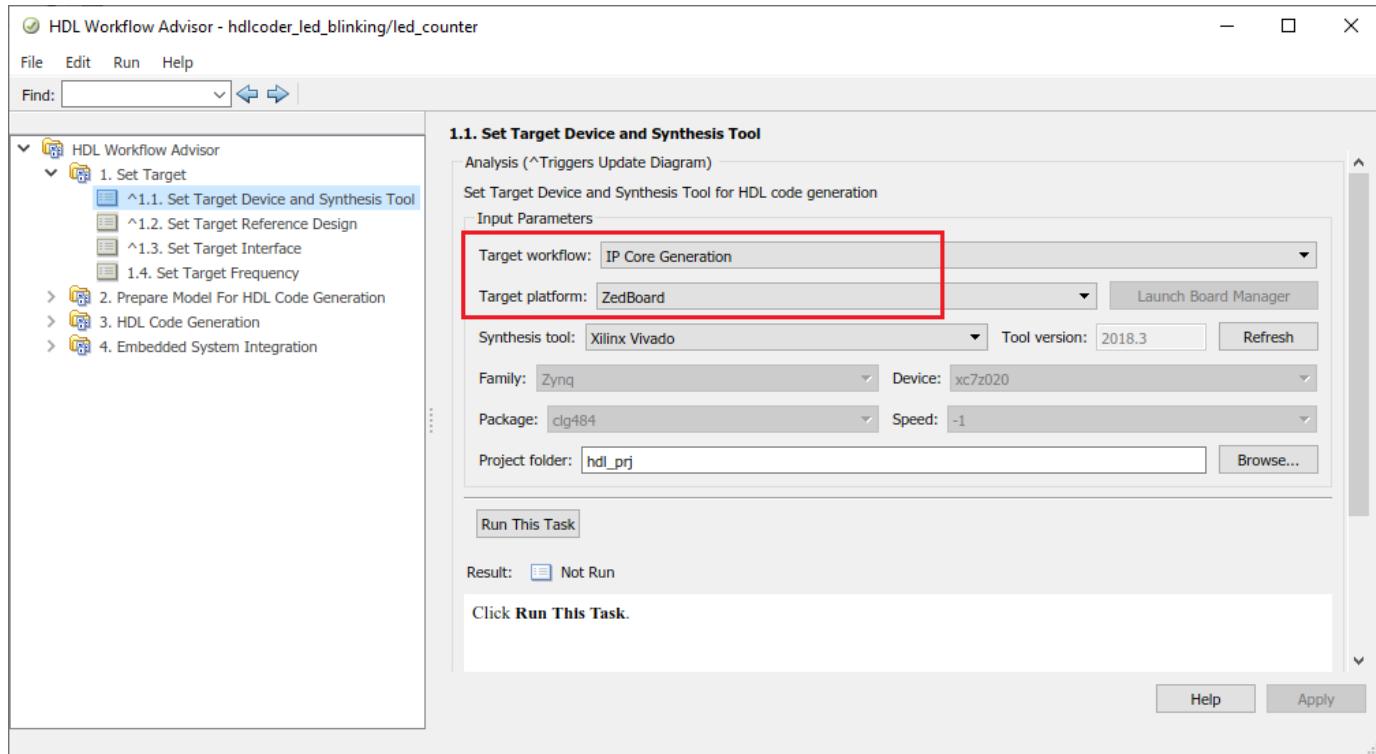
```
hdladvisor('hdlcoder_led_blinking/led_counter')
```

Launch HDL Workflow Advisor

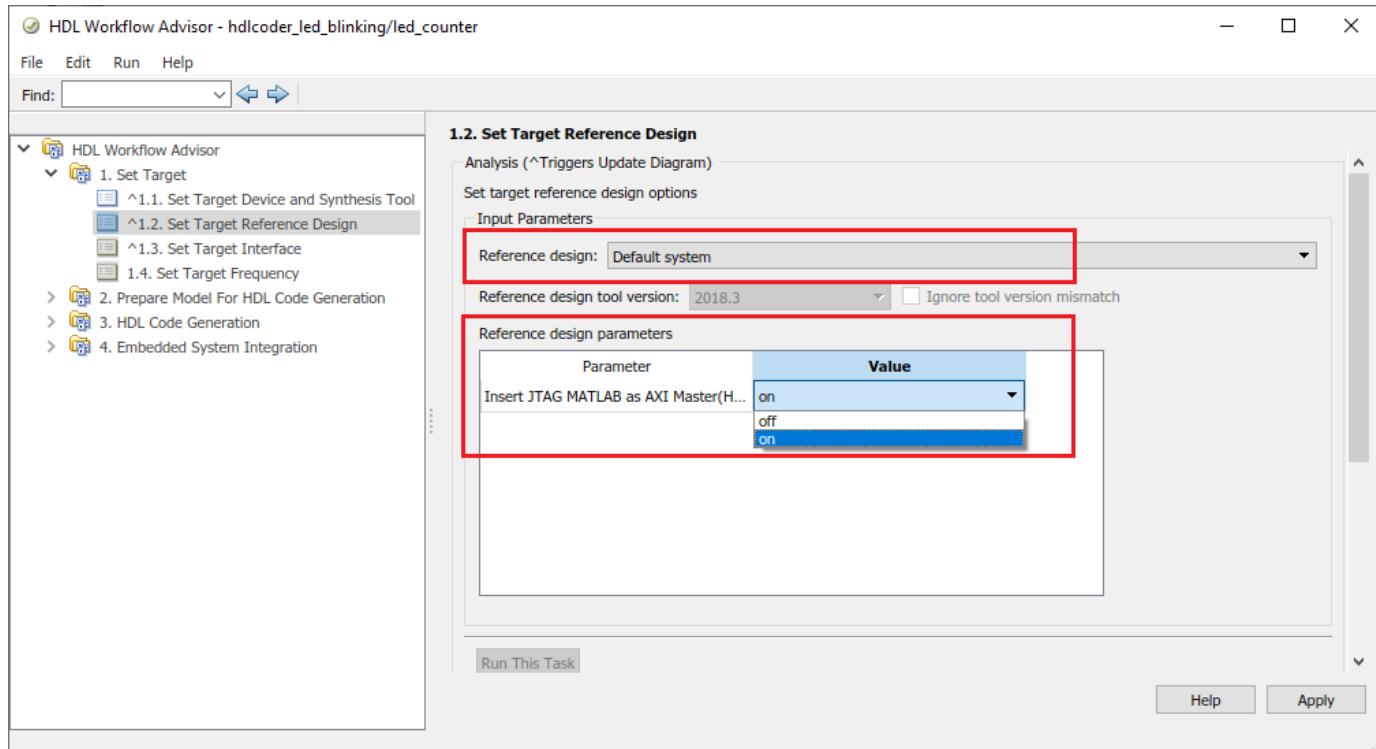
Run Demo

Copyright 2012 The MathWorks, Inc.

2. Open the HDL Workflow Advisor from the hdlcoder_led_blinking/led_counter subsystem by right-clicking the led_counter subsystem, and choosing **HDL Code > HDL Workflow Advisor**.
3. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**.
4. For **Target platform**, select **ZedBoard**. If you don't have this option, select **Get more** to open the Support Package Installer. In the Support Package Installer, select Xilinx Zynq™ Platform and follow the instructions provided by the Support Package Installer to complete the installation.
5. Click **Run This Task** to run the **Set Target Device and Synthesis Tool** task.



6. In the **Set Target > Set Target Reference Design** task, choose **Default system** and set **Insert JTAG MATLAB as AXI Master** dropdown choice to **on** which is present in the reference design parameter options.



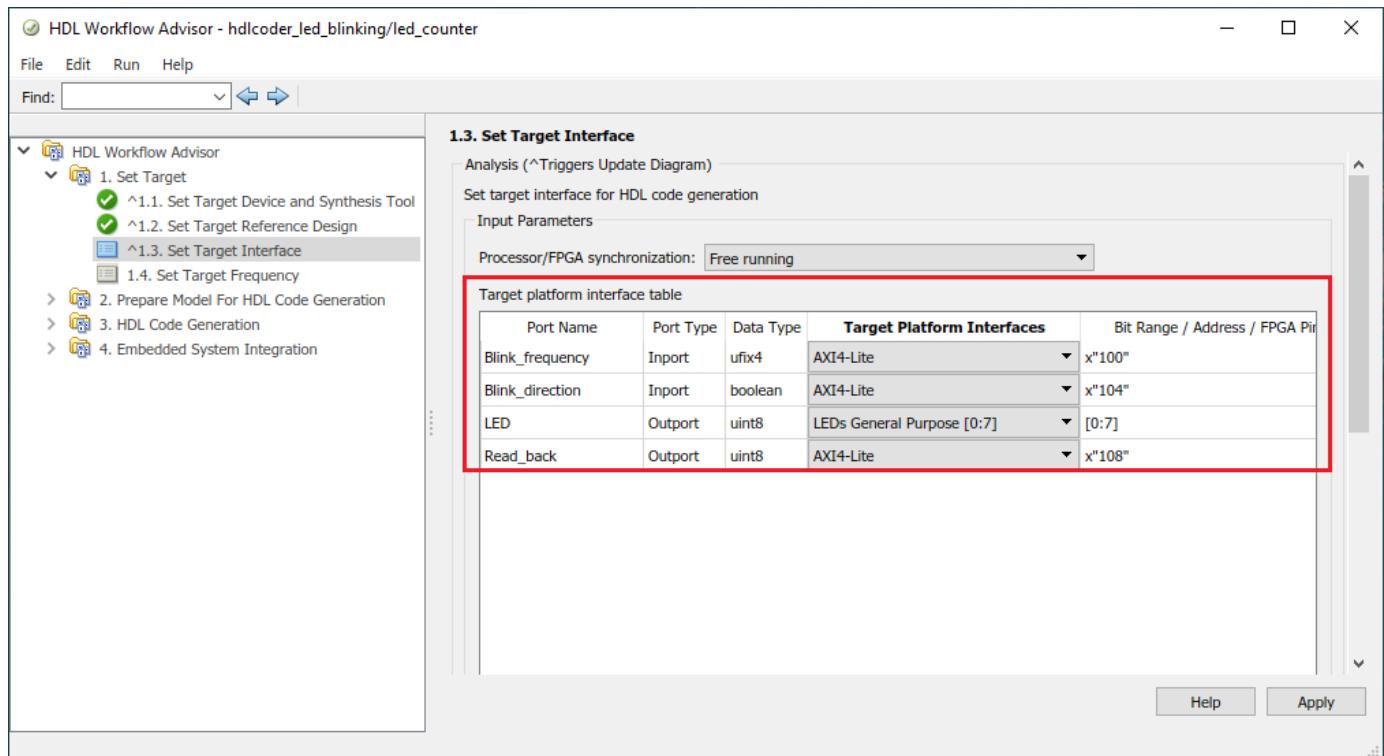
7. Click **Run This Task** to run the **Set Target Reference Design** task.

Generate HDL IP Core and Create Project with AXI Master IP

Map each port in your DUT to one of the IP core target interfaces. In this example, input ports **Blink_frequency** and **Blink_direction** are mapped to the AXI4-Lite interface, so HDL Coder generates AXI interface accessible registers for them. The **LED** output port is mapped to an external interface, **LEDs General Purpose [0:7]**, which connects to the LED hardware on the Zynq board.

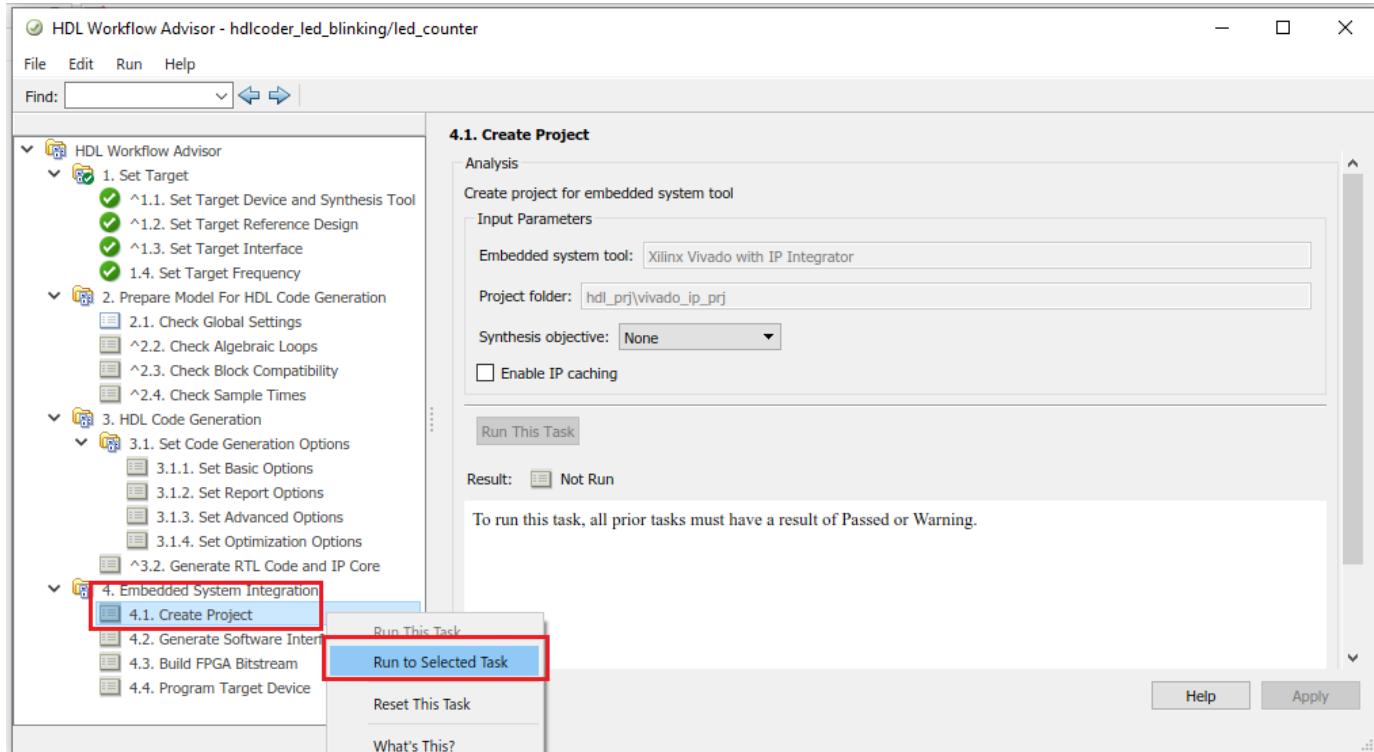
1. In the **Set Target > Set Target Interface** task, choose AXI4-Lite for **Blink_frequency**, **Blink_direction**, and **Read_back**.

2. Choose **LEDs General Purpose [0:7]** for **LED**.

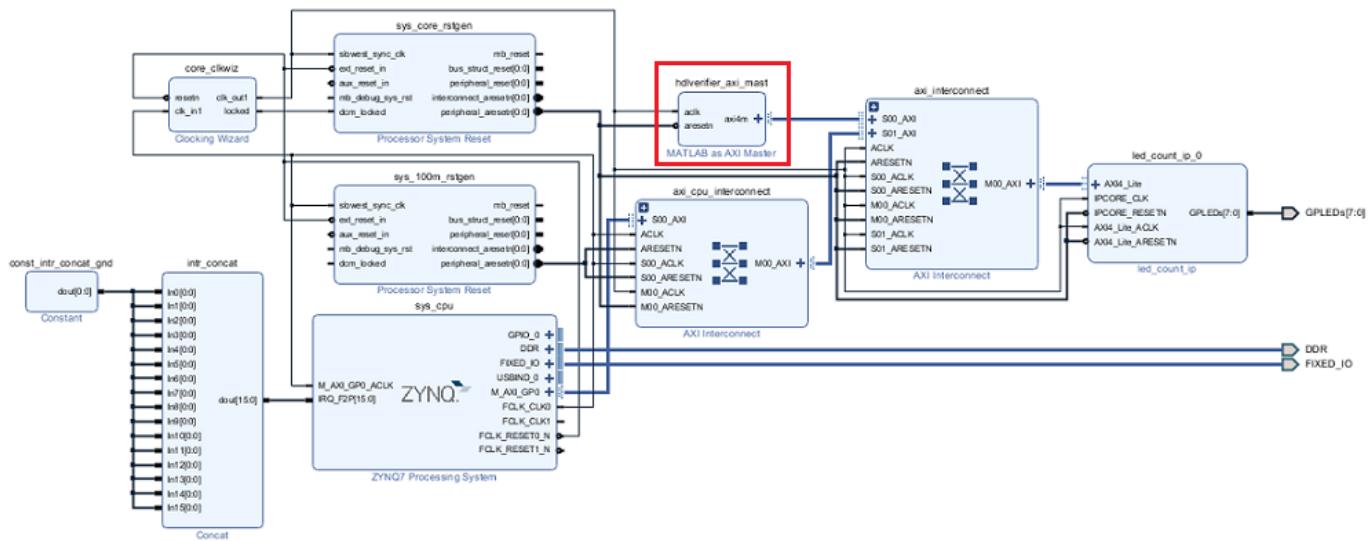


3. Create the reference design project which includes JTAG MATLAB as AXI Master.

To create the project, right-click the **Create Project** task and select **Run to Selected Task**.



"In the Vivado project, you see the JTAG MATLAB as AXI Master IP inserted in the reference design".



Using JTAG MATLAB as AXI Master to Control the HDL Coder IP core

In order to use this feature, you require a HDL Verifier license. After that a simple MATLAB® command line interface can be used to access the IP core generated by HDL Coder.

In the MATLAB command window:

1. Create the AXI master object

```
h = aximaster('Xilinx')
```

2. Input a write command to change the LED Blinking frequency

```
h.writememory('400D0100', 0)
```

Observe the LED blinking frequency is low. Try change the value in AXI Master write command from 0 to 15 to increase the LED blinking frequency.

```
h.writememory('400D0100', 15)
```

3. Input a read command to read the current counter value

```
h.readmemory('400D0108', 1)
```

4. Delete the object when done to free up the JTAG resource. If the object is not deleted, other JTAG operations such as programming the FPGA will fail.

```
delete(h)
```

This demonstration shows how you can easily prototype your FPGA IP core from MATLAB.

JTAG MATLAB as AXI Master in Custom Reference Designs

The "Insert JTAG MATLAB as AXI Master(HDL Verifier required)" reference design parameter is by default added to your custom reference design. The default value for the parameter is "off".

If you want to control these default behaviour for your reference design, you can use following two optional reference design properties:

AddJTAGMATLABasAXIMasterParameter and **JTAGMATLABasAXIMasterDefaultValue** where the reference design author can set those properties to turn off or even disable the parameter option to not to appear in HDL Workflow Advisor.

```
% Insert JTAG MATLAB as AXI Master in reference designs
hRD.AddJTAGMATLABasAXIMasterParameter = true;
hRD.JTAGMATLABasAXIMasterDefaultValue = 'on';
```

1. Parameter visibility option in HDLWA: If you don't want the JTAG MATLAB as AXI Master IP to be inserted in the reference design that you authored, disable this property:

hRD.AddJTAGMATLABasAXIMasterParameter to **false**.

2. Default value of parameter: In the reference design that you authored, you can control the property **hRD.JTAGMATLABasAXIMasterDefaultValue** to '**on**' or '**off**'.

Debug a Zynq Design Using HDL Coder and Embedded Coder

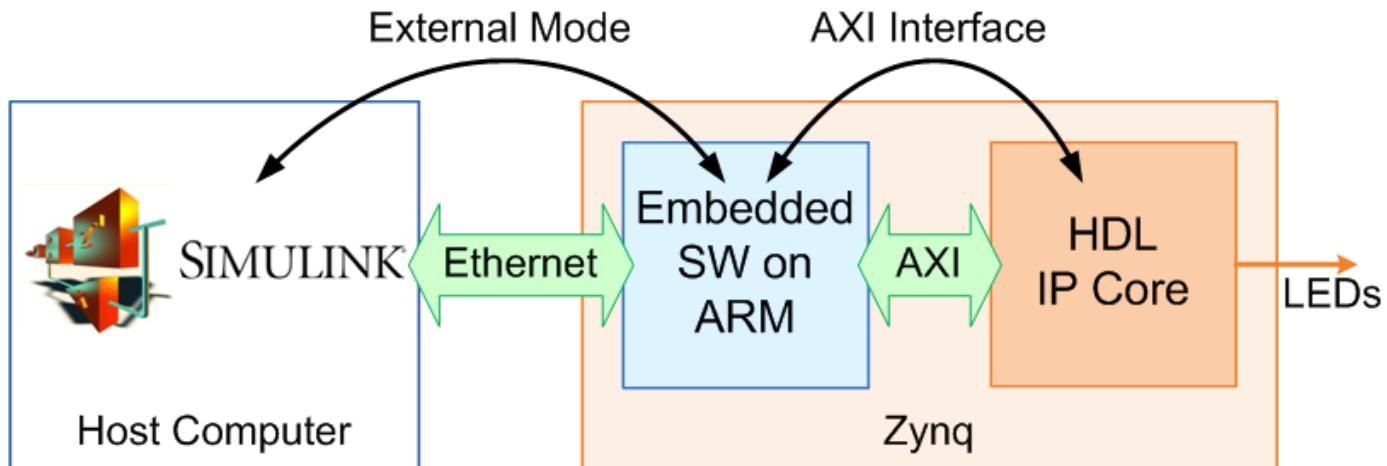
This example shows how to debug a Zynq design using HDL Coder™ and Embedded Coder® features.

Requirements

- Xilinx Zynq-7000 SoC ZC702 Evaluation Kit
- HDL Coder Support Package for Xilinx Zynq Platform
- Embedded Coder Support Package for Xilinx Zynq Platform
- Follow the "Set up Zynq hardware and tools" section in "Getting Started with Targeting Xilinx Zynq Platform" on page 40-65 to setup ZC702 hardware.

Introduction

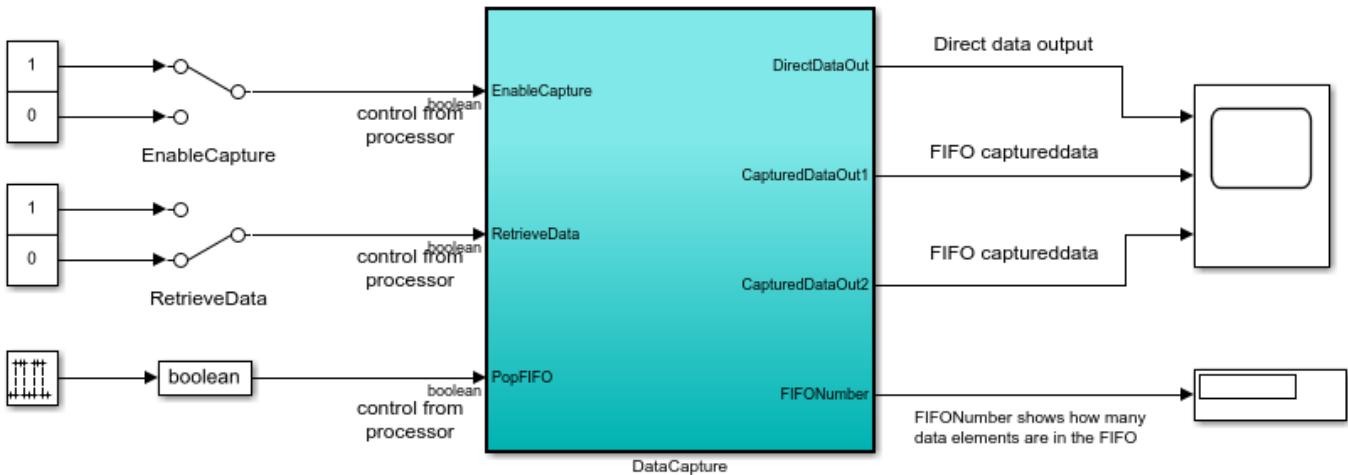
When you are prototyping and developing an algorithm for Zynq platform, it is useful to monitor, tune, and debug the algorithm while it runs on hardware. This example shows how to use features like external mode, AXI interface and HDL FIFO blocks to probe into the Zynq design. Using the external mode feature, you can probe the internal data in the software running on the ARM processor. And because the ARM processor is connected to the FPGA through AXI interface, you can monitor and tune the parameters on FPGA as well. Together with HDL FIFO blocks, you can capture fast FPGA data and retrieve it back to Simulink for analysis.



Let's get started by looking at the example model.

```
open_system('hdlcoder_data_capture');
```

Data Capture In FPGA



This example shows how to use FIFO block to capture FPGA internal data and then retrieve the captured data later.

In MATLAB, type the following:
`hdladvisor('hdlcoder_data_capture/DataCapture')`

[Launch HDL Workflow Advisor](#)

[Run Demo](#)

Copyright 2012-2014 The MathWorks, Inc.

The subsystem **DataCapture** is the hardware subsystem targeting the FPGA fabric. Inside this subsystem, the **OriginalDUT** subsystem contains a **Trigonometric Function** block, which generates fast sine and cosine data streams. The **OriginalDUT** subsystem represents our algorithm design. If we want to debug this design, how do we capture and monitor this fast data stream?

The FPGA runs at a much faster clock frequency than the software code on the ARM processor. External mode can be used with the software running on the ARM processor to monitor slow-changing status parameters, such as FIFO status, but the sample rate of the software code, for example, 1KHz, is not fast enough to capture the fast-changing data in the FPGA, for example, 50MHz.

This example shows how to use a FIFO block to capture the fast FPGA data, and then use the software on the ARM processor to retrieve the captured data through the AXI interface and external mode.

For debugging purposes, we add the subsystem **Debug_FIFOs** to the DUT. This subsystem uses two HDL FIFO blocks to capture the fast data streams for future retrieval. The control signal inputs to the **Debug_FIFOs** subsystem are connected to the DUT interface, and are connected to the ARM processor via the AXI interface.

At the top level of the example model, when the **EnableCapture** switch is turned on, and **RetrieveData** switch is turned off, the **Debug_FIFOs** module will capture 1000 data samples into the **HDL FIFO** blocks. This is the data capture phase. Then, when the **EnableCapture** switch is kept on, and the **RetrieveData** switch is turned back on, the **Debug_FIFOs** module will transfer the captured data back to the ARM processor. This is the data retrieval phase. You can use the manual switches to switch between these two phases to capture and monitor the internal FPGA data.



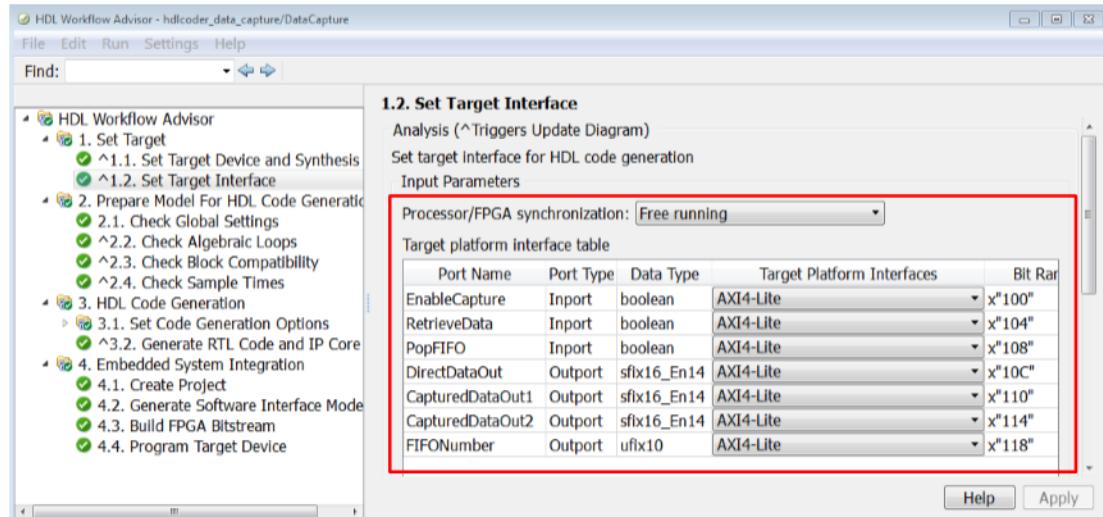
Thus, for every signal you want to monitor, you can insert more **Debug_FIFO** modules to your design to capture and retrieve the data back to Simulink. You can also use your own control signals, or extend this example with your own triggers, or qualifiers.

The output port of the hardware subsystem, **DirectDataOut**, outputs data directly to the AXI interface. In contrast, the output ports **CapturedDataOut1** and **CapturedDataOut2** outputs captured data from the FIFOs. We will compare the results of these two outputs in the last section.

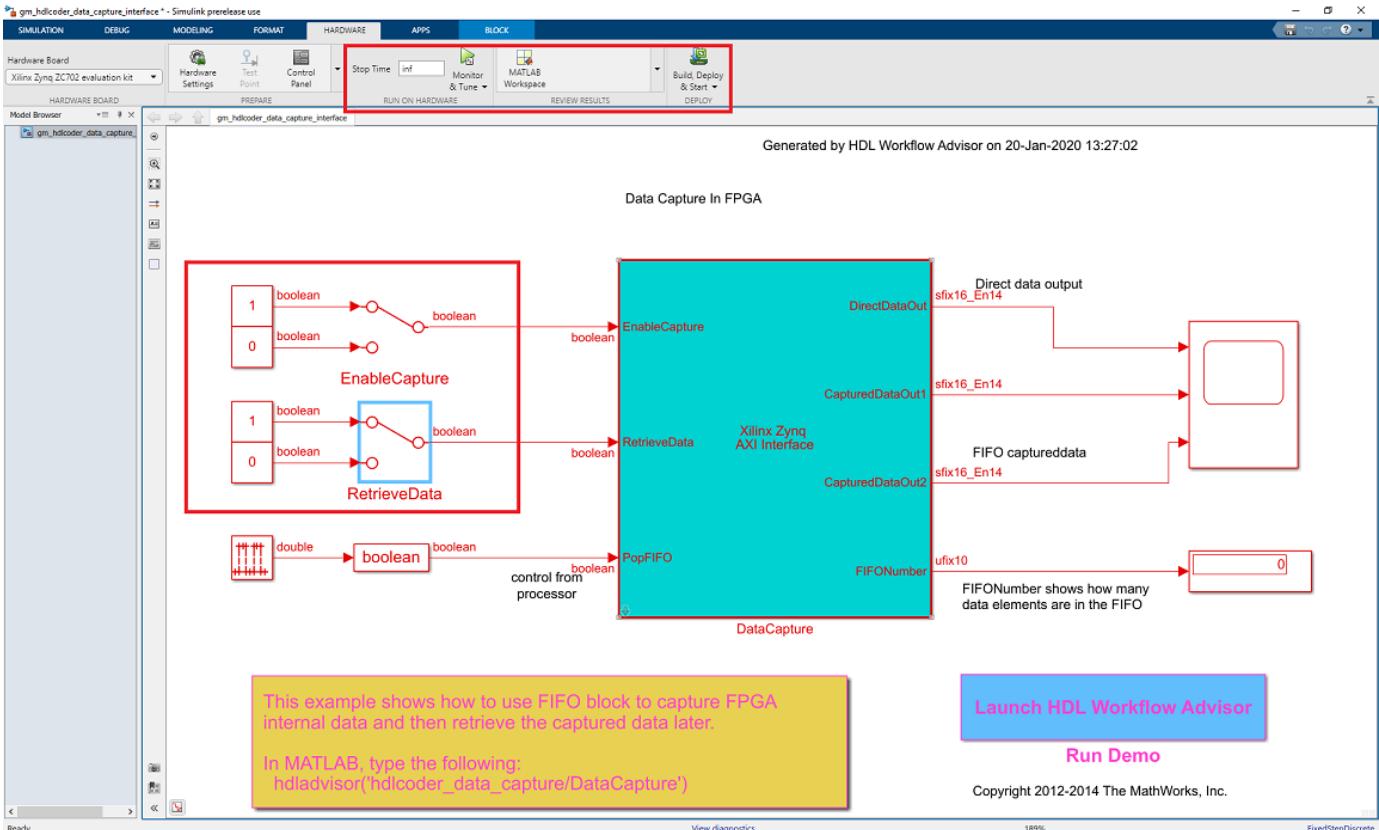
Deploy the design on Zynq hardware

Next, we will start HDL Workflow Advisor from the model and run through the Zynq HW/SW co-design workflow to deploy this design on Zynq hardware. For a detailed step by step guide, please refer to the example “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65.

1. In the **Set Target > Set Target Device and Synthesis Tool** task, for **Target workflow**, select **IP Core Generation**. For **Target platform**, select **Xilinx Zynq ZC702 evaluation kit**. Run this task.
2. In the **Set Target > Set Target Interface** task, choose **AXI4-Lite** for all the input and output ports.



3. Then run through all the workflow steps to generate HDL IP, create an EDK project, generate the software interface model, and build and download the FPGA bitstream. The generated software interface model is shown in following picture:



4. Configure and build the software interface model for external mode:

- 1 In the generated model, click on Hardware pane and go to **Hardware settings** to open **Configuration Parameter** dialog box.
- 2 Select **Solver** and set "Stop Time" to "inf".
- 3 From the Hardware pane, click the **Monitor and Tune** button.
- 4 Click the **Run** button on the model toolbar. Embedded Coder builds the model, downloads the ARM executable to the Zynq ZC702 hardware, executes it, and connects the model to the executable running on the Zynq ZC702 hardware.

Capture and display data from Zynq hardware

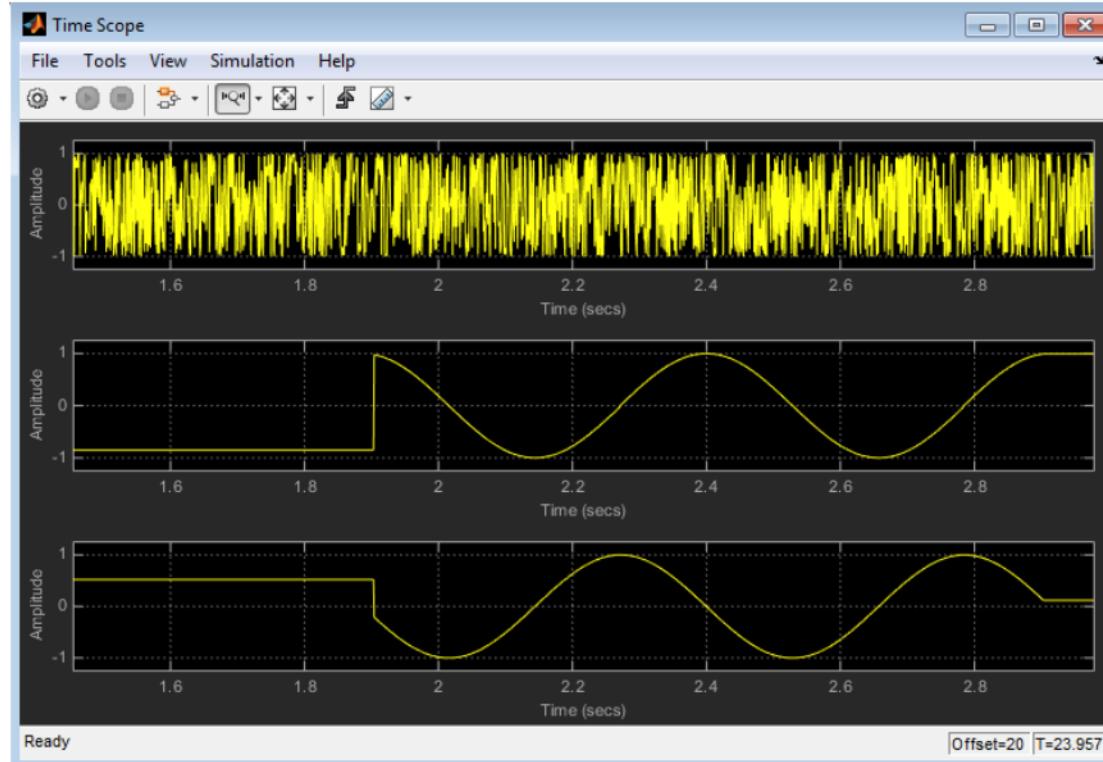
Now both the hardware and software parts of the design are running on Zynq hardware, the next step is to capture and retrieve data from the Zynq board.

Once the external mode is connected, make sure the **EnableCapture** switch is in the **1** position, and the **RetrieveData** switch is in the **0** position. Notice the **FIFONumber** display box increases to 1000 almost immediately. This means the FIFO inside the FPGA fabric started capturing data and was quickly filled with 1000 data samples.

Open the **Time Scope** block and observe the **DirectDataOut** output in the first row. Notice the received data is a seemingly random waveform between -1 and 1. This is because the FPGA is running at a much faster frequency than software is running on the ARM processor. Directly using external mode to monitor fast FPGA data means sampling a fast sine waveform at a very slow rate, which generates a random waveform between -1 and 1.

Now double-click the **RetrieveData** switch to enable data readout. The **EnableCapture** switch needs to be kept on. When the **RetrieveData** switch is turned on, the internal logic modeled in this example sends out the captured data samples one by one from FIFO to the ARM processor, through the AXI interface. These data samples are then sent from ARM processor to Simulink via external mode. Notice the **FIFONumber** display box decreases to 0.

Open the **Time Scope** block, the second and third row of the scope now shows the sine and cosine wave we captured in the FIFO. Following picture shows the scope waveforms.



Summary

This example shows how to use features like external mode, AXI interface and HDL FIFO blocks to probe into the Zynq design, capture fast FPGA data, and retrieve it back to Simulink for analysis. You can also do similar monitoring with FPGA vendor tools like ChipScope™ or SignalTap™. But the strength of this approach is that you can get all the visualization benefits of Simulink, like various scope blocks, and you can also build your own custom controls, triggers, or qualifiers in Simulink.

Debug IP Core Using FPGA Data Capture

This example shows how to debug HDL Coder generated IP Core using HDL Verifier's FPGA Data Capture feature.

Requirements

- Xilinx Zynq ZC702 evaluation kit
- HDL Coder Support Package for Xilinx Zynq Platform
- HDL Verifier Support Package for Xilinx FPGA Boards
- (Optional) Embedded Coder Support Package for Xilinx Zynq Platform
- (Optional) DSP System Toolbox
- Follow the "Set up Zynq hardware and tools" section in HDL Coder example "Getting Started with Targeting Xilinx Zynq Platform" on page 40-65 to setup ZC702 hardware.

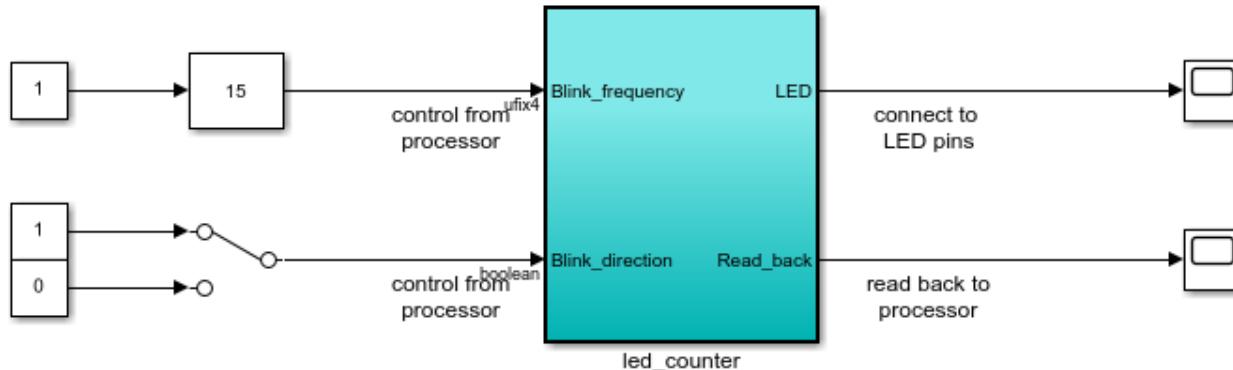
Introduction

When you debug the generated IP Core from HDL Coder, it is useful to monitor the IP Core internal signals when it is running on the real hardware. This example shows how to use the HDL Verifier's FPGA Data Capture to capture such signals into MATLAB for debugging analysis.

Start by looking at the example model:

```
open_system('hdlcoder_led_blinking_data_capture');
```

Using IP Core Generation Workflow: LED Blinking



This example shows how to use HDL Workflow Advisor to generate a custom IP core which blink LEDs on FPGA board.

In MATLAB, type the following:
`hdladvisor('hdlcoder_led_blinking_data_capture/led_counter')`

[Launch HDL Workflow Advisor](#)

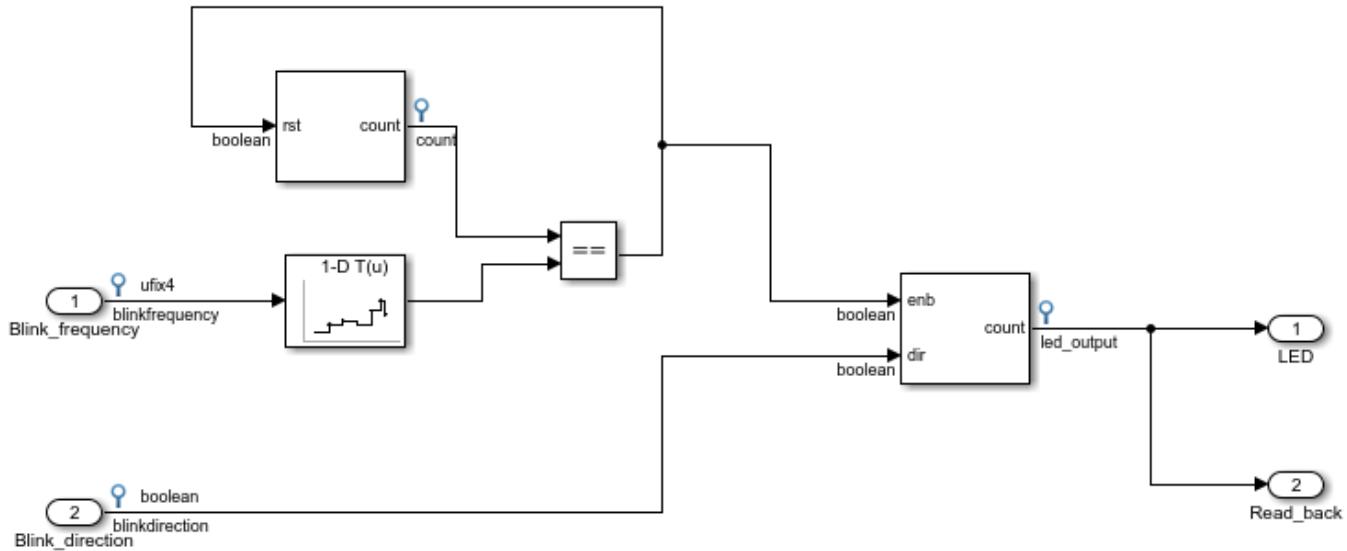
[Run Demo](#)

Copyright 2018 The MathWorks, Inc.

The subsystem `led_counter` is the hardware subsystem targeting the FPGA fabric. Inside this subsystem, we marked several internal signals as test points. HDL Coder will route those internal

signals out of the DUT and into the IP Core wrapper so that the signals can be connected to the FPGA Data Capture HDL IP.

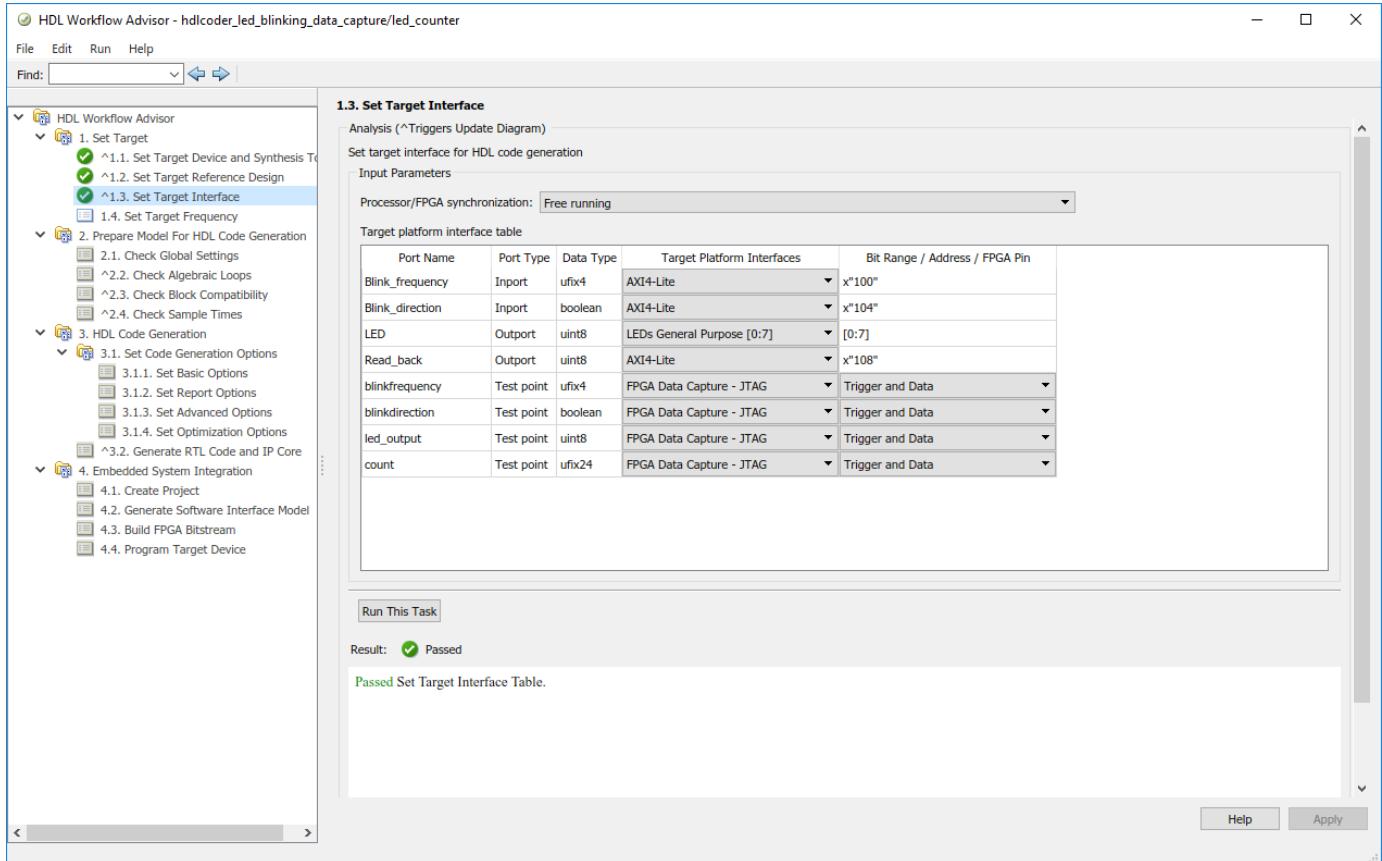
```
open_system('hdlcoder_led_blinking_data_capture/led_counter');
```



Generate HDL IP Core

Start HDL Workflow Advisor from the model and run through the IP Core Generation workflow. For a detailed step by step guide, please refer to the example “Getting Started with Targeting Xilinx Zynq Platform” on page 40-65

1. In step 1.1., select IP Core Generation in the Target workflow. For "Target Platform", select "Xilinx Zynq ZC702 evaluation kit"
2. In step 3.1.3, under "Ports" tab check the "Enable HDL DUT port generation for test points"
3. In Step 1.3, select "FPGA Data Capture - JTAG" interface for `blinkfrequency`, `blinkdirection`, `led_output`, and `count` ports.



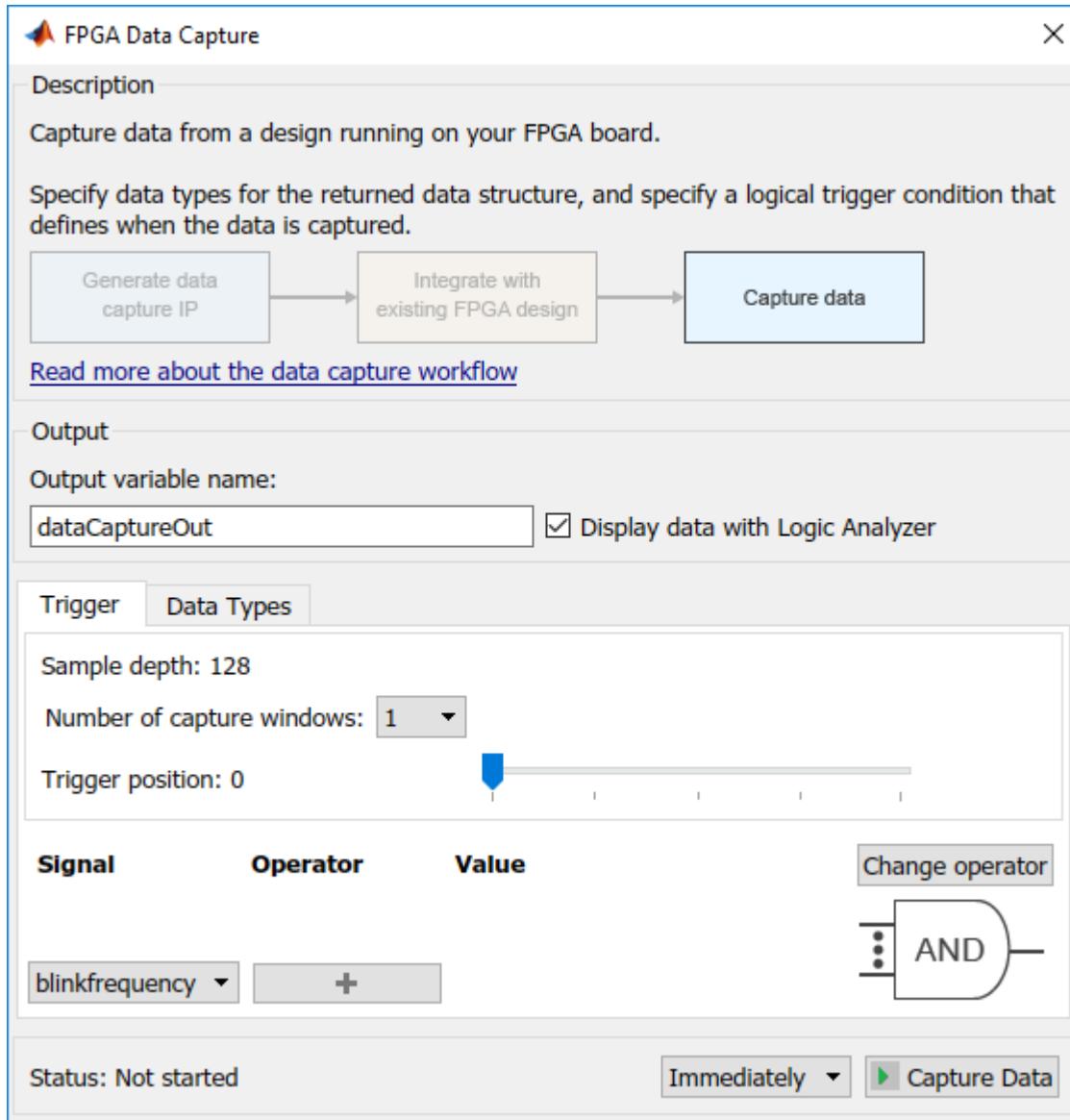
4. Run through the remaining workflow steps to generate HDL IP, and program the target device.

Capture and display data from IP Core

Now FPGA fabric has been programmed and running, the next step is to capture the data from the Zynq board.

First, locate the FPGA Data Capture launch script. In this example, the script is in your HDL code generation directory: `hdl_prj/ip_core/led_count_ip_v1_0/fpga_data_capture/launchDataCaptureApp.m`. You can also locate this script in the code generation report.

Next, run this script in MATLAB. You will need to add the directory where this script is located to the MATLAB path or change your current folder.



After executing this script, the FPGA Data Capture App is launched. You can click the "Capture Data" button to capture data from FPGA without setting up any triggers.

Alternatively, you can setup a trigger condition where `led_counter==0`, and trigger position of 32. Then click "Capture Data" button again.

