R-Tile Avalon® Streaming Intel® FPGA IP for PCI Express* Design Example User Guide

Updated for Intel® Quartus® Prime Design Suite: **23.2**

IP Version: **10.0.0**

Online Version
Send Feedback

UG-20330

683544
2023.06.26

Contents

1. About the R-Tile Avalon® Streaming Intel® FPGA IP for PCI Express

The following table presents an overview of the design examples supported by the R-tile Avalon® Streaming Intel® FPGA IP for PCI Express.

Table 1. Design Examples Supported by the R-tile Avalon Streaming Intel FPGA IP for PCI Express

Design Example	Hard IP Mode	Simulators Supported	Development Kits Supported
PIO	Gen5 1x16 1024-bit Endpoint	VCS*, VCS MX, Siemens* EDA QuestaSim*, Xcelium* (1) *Note:* Simulation support is not available for Gen3 and Gen4 in this release of Intel Quartus® Prime.	Intel Agilex 7 I-Series FPGA Development Kit ES (2)
	Gen4 1x16 1024-bit Endpoint		
	Gen3 1x16 1024-bit Endpoint		
	Gen5 2x8 512-bit Endpoint		
	Gen4 2x8 512-bit Endpoint		
	Gen3 2x8 512-bit Endpoint		
SR-IOV	Gen5 1x16 1024-bit Endpoint	VCS, VCS MX, Siemens EDA QuestaSim, Xcelium (1)	Intel Agilex 7 I-Series FPGA Development Kit ES (2)
Performance	Gen5 1x16 1024-bit Endpoint	Not supported	Intel Agilex 7 I-Series FPGA Development Kit ES (2)

1.1. Functional Description for the Programmed Input/Output (PIO) Design Example

The Programmed Input/Output (PIO) design example performs memory transfers from a host processor to a target device. In this example, the host processor requests single-dword Memory Read (MemRd) and Memory Write (MemWr) Transaction Layer Packets (TLPs).

The PIO design example automatically creates the files necessary to simulate and compile in the Intel Quartus Prime software. The design example covers a wide range of parameters. However, it does not cover all possible parameterizations of the R-tile Hard IP for PCIe.

(1) Xcelium simulator support is only available in devices with the following OPN numbers: AGIx027R29AxxxxR3, AGIx027R29AxxxxR2, AGIx023R18AxxxxR0, AGIx041R29DxxxxR0, AGIx041R29DxxxxR1. For more details on OPN decoding, refer to the Intel Agilex® 7 FPGAs and SoCs Device Overview

(2) For more information, refer to the Intel Agilex 7 I-Series FPGA Development Kit

This design example supports the following configurations:

Table 2. Configurations Supported by the PIO Design Example

Support level keys: S = simulation, C = compilation, T = timing, H = hardware, N/A = configuration not supported.

Port Mode	Link Width	Lin Speed	Data Width (Bits)	Design Example Support	Simulators Supported
Endpoint	x16	Gen5	1024 (4 x 256)	SCTH	Siemens EDA QuestaSim, VCS, VCS MX, Xcelium (3)
		Gen4	1024 (4 x 256)	CTH	N/A
			512 (2 x 256)	N/A	N/A
		Gen3	1024 (4 x 256)	CTH	N/A
			512 (2 x 256)	N/A	N/A
	x8	Gen5	512 (2 x 256)	SCTH	Siemens EDA QuestaSim, VCS, VCS MX, Xcelium (3)
		Gen4	512 (2 x 256)	CTH	N/A
			256 (1 x 256)	N/A	N/A
		Gen3	512 (2 x 256)	CTH	N/A
			256 (1 x 256)	N/A	N/A
	x4	Gen5	256 (2 x 128)	N/A	N/A
		Gen4	256 (2 x 128)	N/A	N/A
			128 (1 x 128)	N/A	N/A
		Gen3	256 (2 x 128)	N/A	N/A
			128 (1 x 128)	N/A	N/A
Root Port	N/A	N/A	N/A	N/A	N/A
TLP Bypass	N/A	N/A	N/A	N/A	N/A
PIPE-D	N/A	N/A	N/A	N/A	N/A

The clock comes from the coreclkout_hip output of the IP and runs at 500 MHz.

Note: In the 23.2 release of Intel Quartus Prime, this design example only supports the default settings in the Parameter Editor of the R-tile Avalon Streaming Intel FPGA IP for PCIe.

This design example includes the following components:

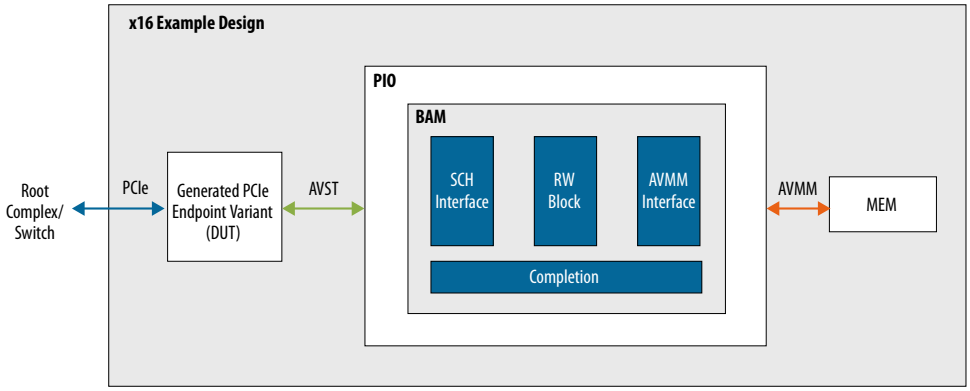
(3) Xcelium simulator support is only available in devices with the following OPN numbers: AGIx027R29AxxxxR3, AGIx027R29AxxxxR2, AGIx023R18AxxxxR0, AGIx041R29DxxxxR0, AGIx041R29DxxxxR1. For more details on OPN decoding, refer to the Intel Agilex 7 FPGAs and SoCs Device Overview.

- The generated R-tile Avalon Streaming (Avalon-ST) Hard IP Endpoint variant (DUT) with the parameters you specified. This component drives TLP data received to the PIO application. It translates the PCIe serial data received from the link to the Avalon-ST data format.
- The PIO Application (APPS) component, which performs the necessary translation between the PCI Express TLPs and simple Avalon Memory-mapped (Avalon-MM) writes and reads to the on-chip memory.

  Note: The current APPS component supports only single-cycle data transfers. Data transfers longer than one clock cycle are not supported.
- An on-chip memory (MEM) component (one 32 KB memory for the x16 design example, and two 32 KB memories for the 2x8 design example).
- Reset Release IP: This IP holds the control circuit in reset until the device has fully entered user mode. The FPGA asserts the nINIT_DONE output to signal that the device is in user mode. The nINIT_DONE signal is high until the entire device enters user mode. After nINIT_DONE deasserts (low), all logic is in user mode and operates normally.

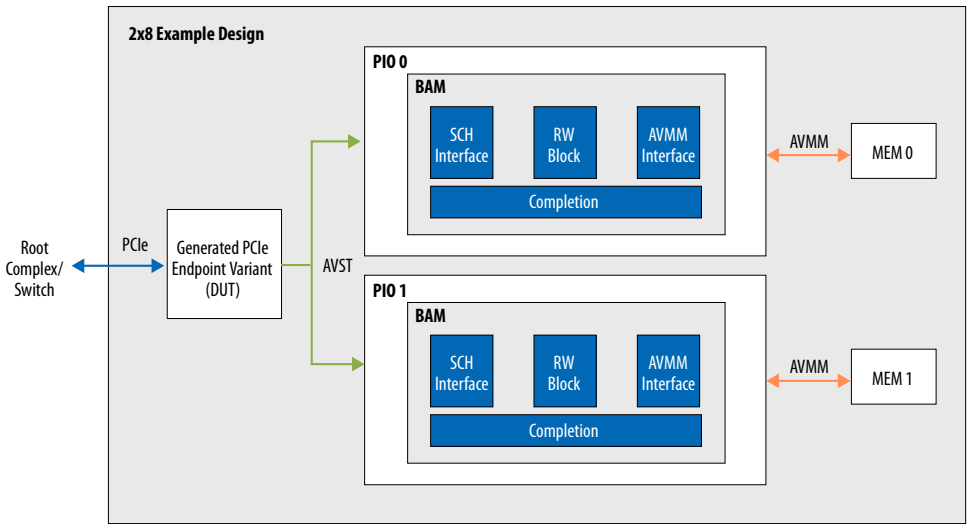
The Gen5 x16 design example instantiates a PIO component with a 1024-bit data path to interface with the 1024-bit DUT. Also, the design example instantiates only one MEM device as shown in the figure below.

Figure 1. Gen5 x16 Design Example Block Diagram



The Gen5 2x8 design example instantiates two PIO components with 512-bit data paths to interface with the 2x512-bit DUT. Also, the design example instantiates two MEM devices as shown in the figure below.

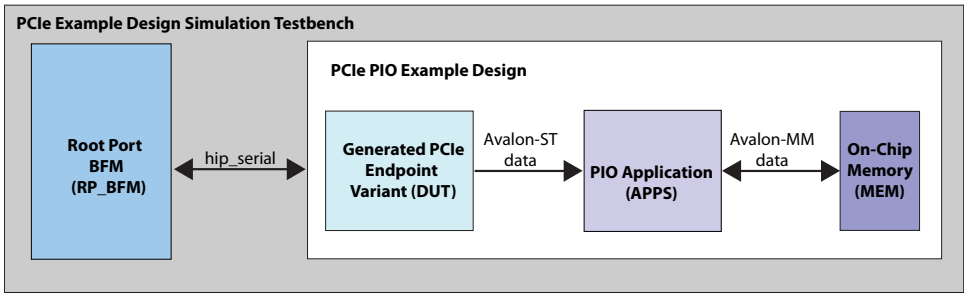
Figure 2. Gen5 2x8 Design Example Block Diagram



For simulation purposes, this design example also generates a testbench that instantiates the PIO design example and a Root Port BFM to interface with the target Endpoint.

Note: The simulation testbench for the PCIe 2x8 PIO design example has a single PCIe x8 link although the actual design implements two PCIe x8 links.

Figure 3. Block Diagram for the Platform Designer PIO Design Example Simulation Testbench



The test program writes to and reads back data from the same location in the on-chip memory. It compares the data read to the expected result. The test reports, "Simulation stopped due to successful completion" if no errors occur.

Figure 4. Platform Designer System Contents for the R-tile Avalon-ST PCI Express Gen5 x16 PIO Design Example

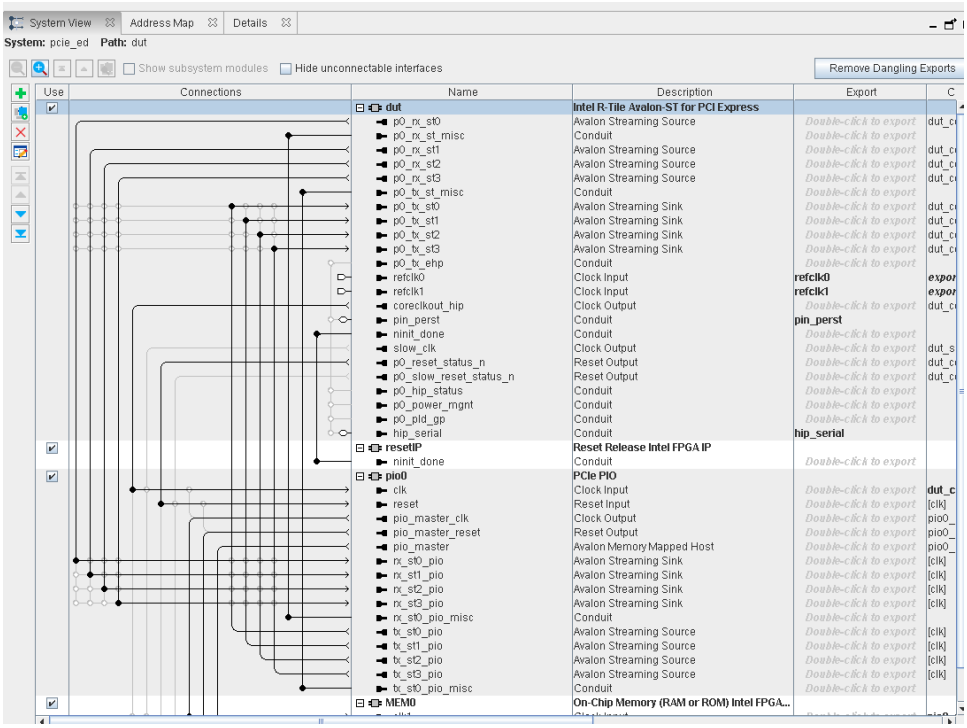
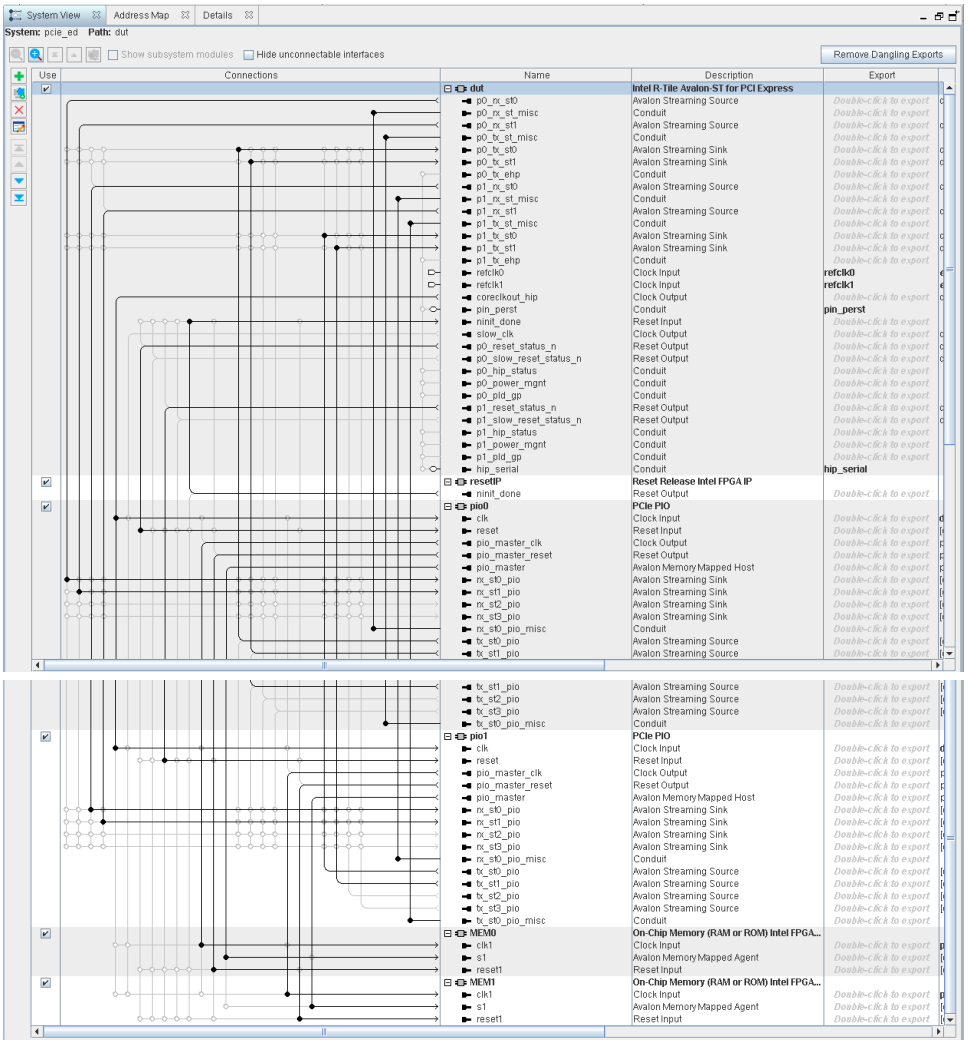


Figure 5. Platform Designer System Contents for the R-tile Avalon-ST PCI Express Gen5 2x8 PIO Design Example

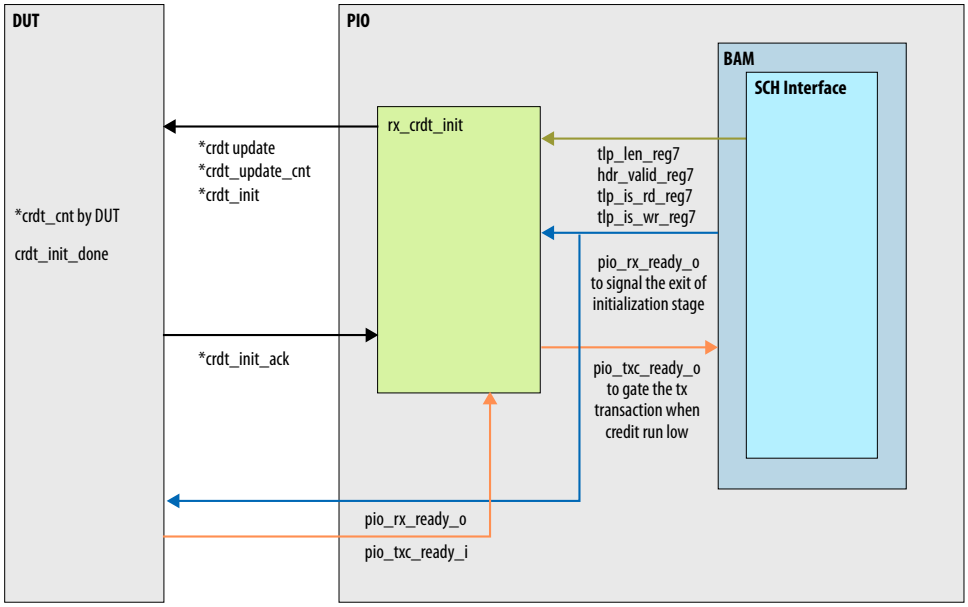


1.1.1. Credit Initialization Sequence

In R-tile, the back-pressure mechanism by the PIO component is done through the credit system. Therefore, the credit value must be declared during the credit initialization stage. An initial TX credit value is captured by the DUT and deducted for any TLP sent to the PIO. When the credit value reaches zero, the DUT stops sending any more TLP until the PIO returns the credit.

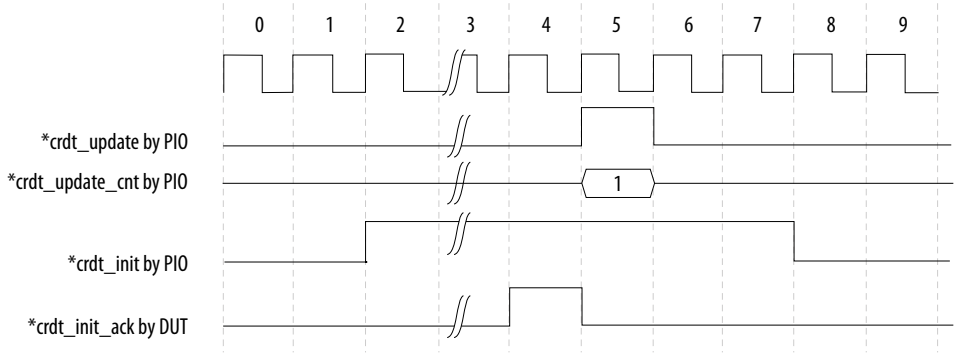
For the R-tile design example, the RX_CRDT_INIT block interfaces with the credit signals from the DUT as shown in the figure below. The block focuses on initializing and returning the RX credit. In the TX direction, the block only asserts `crdt_init_ack` to complete the initialization stage. The PIO component captures the TX credit during the initialization stage initiated by the DUT.

Figure 6. Credit System of the Design Example



The waveforms below show the initialization sequence for RX credit from PIO to DUT. During the initialization phase, the PIO asserts the `*crdt_init` signal. In response, the DUT asserts the `*crdt_init_ack` signal. After receiving the ack signal, the PIO asserts `*crdt_update` and the internal crdt_cnt of the DUT captures the `*crdt_update_cnt` value.

Figure 7. Waveforms of Credit Transactions During the Initialization Stage



1.1.1.1. Credit Value Initialization and Return

The following table shows an example of a posted 1024-bit write sequence consuming one posted header credit and eight posted data credits.

Table 3. Example of a 1024-bit Posted Write Sequence

Header								H0
Data	D7	D6	D5	D4	D3	D2	D1	D0

A non-posted read sequence consumes one non-posted header credit only as shown in the table below.

Table 4. Example of a Non-Posted Read Sequence

Header	H0
Data	

1.1.1.2. Credit Distribution

Based on the available FIFO depth, the available credit is distributed as shown below.

Table 5. Credit Distribution for RX Credit Initialization

Credit Type	Credit Initialization		
Header Type	**Posted**	**Non-Posted**	**Completion**
Header Credit	11	11	10
Data Credit	85	85	85

Note: Each 16 bytes of data consume one credit.

1.2. Functional Description for the Single Root I/O Virtualization (SR-IOV) Design Example

The SR-IOV design example performs memory transfers from a host processor to a target device. It supports up to two PFs and 32 VFs per PF.

This design example automatically creates the files necessary to simulate and compile in the Intel Quartus Prime software. You can download the compiled design to an Intel Agilex 7 I-Series FPGA Development Kit.

Table 6. Configurations Supported by the SR-IOV Design Example

Support level keys: S = simulation, C = compilation, T = timing, H = hardware, N/A = configuration not supported.

Port Mode	Link Width	Link Speed	Data Width (Bits)	Design Example Support	Simulators Support
Endpoint	x16	Gen 5	1024 (4 x 256)	SCTH	VCS, VCS MX, Siemens EDA QuestaSim, Xcelium [4]
		N/A	N/A	N/A	N/A
			N/A	N/A	N/A
		N/A	N/A	N/A	N/A
			N/A	N/A	N/A

continued...

[4] Xcelium simulator support is only available in devices with the following OPN numbers: AGIx027R29AxxxxR3, AGIx027R29AxxxxR2, AGIx023R18AxxxxR0, AGIx041R29DxxxxR0, AGIx041R29DxxxxR1. For more details on OPN decoding, refer to the Intel Agilex 7 FPGAs and SoCs Device Overview.

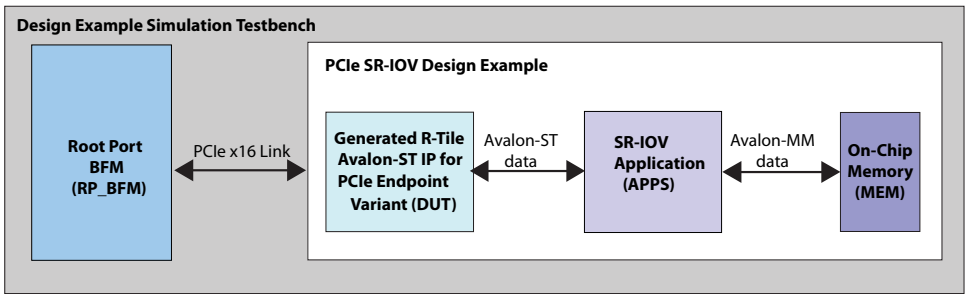
Port Mode	Link Width	Link Speed	Data Width (Bits)	Design Example Support	Simulators Support
	x8	N/A	N/A	N/A	N/A
		N/A	N/A	N/A	N/A
			N/A	N/A	N/A
		N/A	N/A	N/A	N/A
			N/A	N/A	N/A
	x4	N/A	N/A	N/A	N/A
		N/A	N/A	N/A	N/A
			N/A	N/A	N/A
		N/A	N/A	N/A	N/A
			N/A	N/A	N/A
Root Port	N/A	N/A	N/A	N/A	N/A
TLP Bypass	N/A	N/A	N/A	N/A	N/A
PIPE-D	N/A	N/A	N/A	N/A	N/A

This design example includes the following components:

- The generated R-Tile Avalon Streaming (Avalon-ST) IP Endpoint variant (DUT) with the parameters you specified. This component drives the received TLP data to the SR-IOV application.
- The SR-IOV Application (APPS) component, which performs the necessary translation between the PCI Express TLPs and simple Avalon-ST writes and reads to the on-chip memory. For the SR-IOV APPS component, a memory read TLP generates a Completion with data.
- A Reset Release IP.

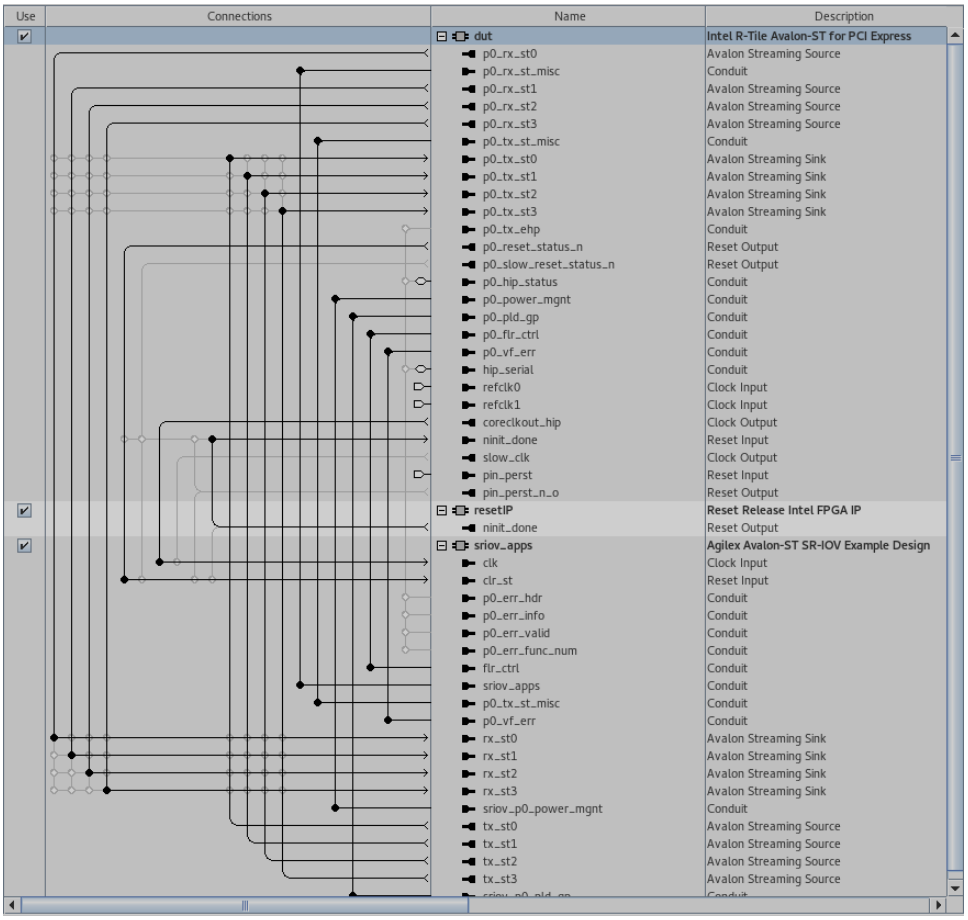
The simulation testbench instantiates the SR-IOV design example and a Root Port BFM to interface with the target Endpoint.

Figure 8. Block Diagram for the Platform Designer SR-IOV Design Example Simulation Testbench



The test program writes to and reads back data from the same location in the on-chip memory across 2 PFs and 32 VFs per PF. It compares the data read to the expected result. The test reports, "Simulation stopped due to successful completion" if no errors occur.

Figure 9. Platform Designer System Contents for the R-Tile Avalon-ST IP with SR-IOV for PCI Express 1x16 Design Example



1.3. Functional Description for the Performance Design Example

Note: This design example is only supported in devices with the following OPN numbers: AGIx027R29AxxxxR3, AGIx027R29AxxxxR2, AGIx023R18AxxxxR0, AGIx041R29DxxxxR0, AGIx041R29DxxxxR1. For more details on OPN decoding, refer to the Intel Agilex 7 FPGAs and SoCs Device Overview.

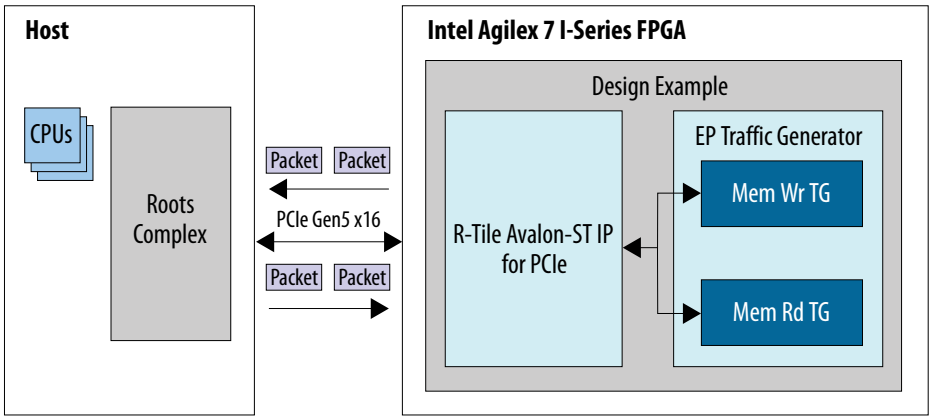
The Performance design example performs memory transfers from the R-Tile Avalon Streaming Intel FPGA IP for PCIe to the host system memory. You can configure the End Point Traffic Generator design example to send:

- Memory Write-only TLPs
- Memory Read-only TLPs
- Both Memory Write and Memory Read TLPs

There is a traffic counter implemented in the FPGA Application logic to measure the amount of traffic that is being generated. To make a traffic measurement, the software application running at the host side issues a memory read TLP, acquires the counter

value, and prints the traffic generated on the system terminal. The software application performs a memory write to the control register within the Application logic to start and stop the traffic.

Figure 10. High-Level View of the Endpoint Traffic Generator Design Example



The Performance design example includes all the necessary files to be compiled in the Intel Quartus Prime software. It supports the Gen5 x16, 1024-bit interface Hard IP Mode, with a 500MHz clock frequency.

The design example also includes the following components:

- The generated R-Tile Avalon Streaming Hard IP Endpoint variant (DUT). This component interacts with the root complex or switch at the other end of the PCIe link and translates the data on the PCIe link into the Avalon Streaming (Avalon-ST) data format.
- The `pioperf_multitlp_adapter` (Avalon-ST Interface Adapter) module converts the 4 data segments of the Avalon-ST interface into two single-segment streams of Avalon-ST data.
- The `pioperf_rx_diverter` module diverts Memory Write, Memory Read and Completion TLPs from the Host to their respective destinations for further processing.
- The `pioperf_rx_intf` (RX Interface) module decodes the TLP headers and data from the `pioperf_rx_diverter` module. It also extracts the information needed to construct the TLP header of the Completion data such as the requester ID, tag, attribute, Traffic Class (TC) and byte count.
- The `pioperf_wr_traffic_gen` (Write Traffic Controller) module generates memory writes based on the information in the control register.

- The `pioperf_rd_traffic_gen` (Read Traffic Controller) module generates Memory Read TLPs based on the information in the control register. Every Memory Read request is monitored until the arrival of its corresponding Completion.
- The `crdt_intf` module updates the necessary credits between the DUT and the `pioperf_multitlp_adapter` to ensure proper flow control for the received and transmitted TLPs.
- The Reset Release IP holds the control circuit in reset until the FPGA has fully entered into user mode. The FPGA asserts the INIT_DONE output to signal that the device is in user mode. The Reset Release IP generates an inverted version of the internal INIT_DONE signal to create the nINIT_DONE output.

Figure 11. Gen5 x16 Performance Design Example

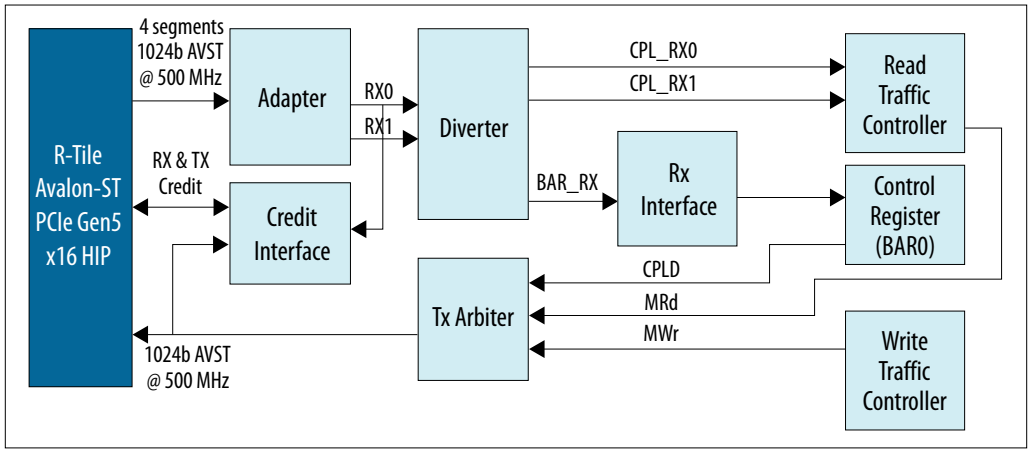
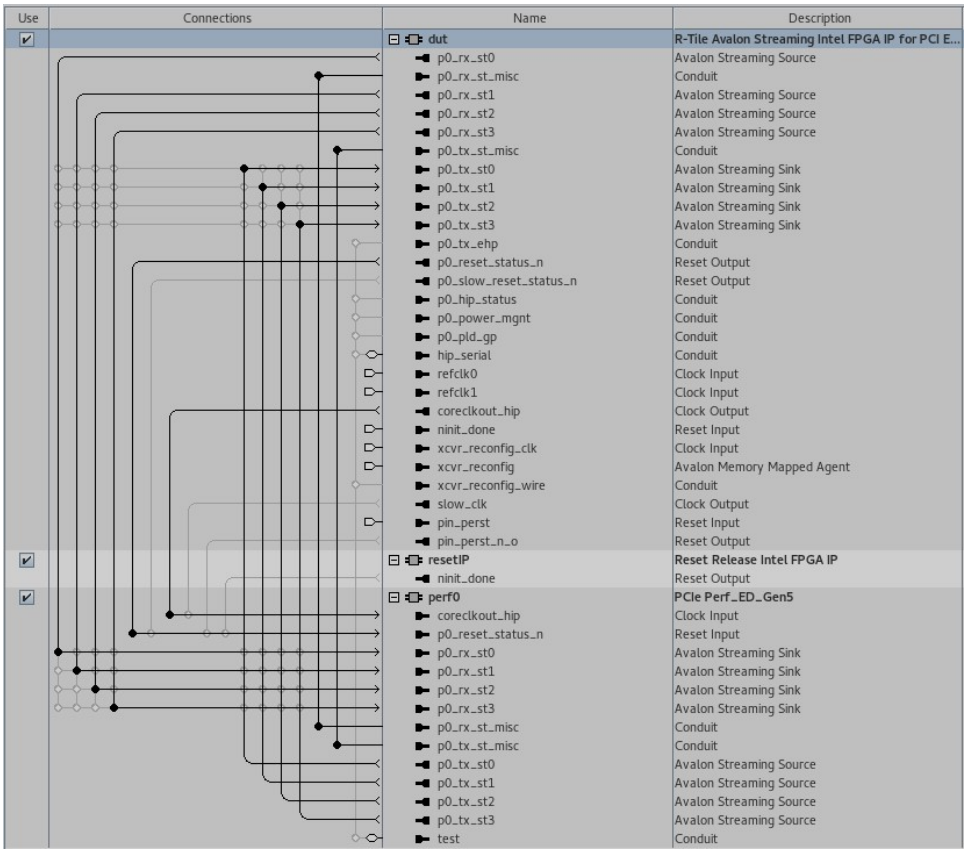


Figure 12. Platform Designer System Contents for the R-Tile 1x16 Performance Design Example



1.4. Hardware and Software Requirements

- Intel Quartus Prime Pro Edition Software version 23.2
- Operating System: CentOS 7.0, 64-bit with 3.10.514 kernel compiled for x86_64 architecture
- Intel Agilex 7 I-Series FPGA Development Kit

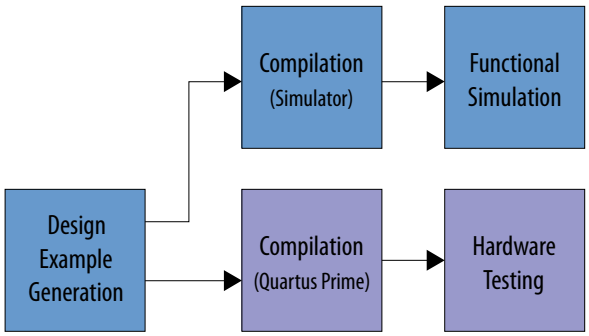
For details on the design example simulation steps and how to run Hardware tests, refer to Quick Start Guide on page 16.

For more information on development kits, refer to FPGA Development Kits on the Intel web site.

2. Quick Start Guide

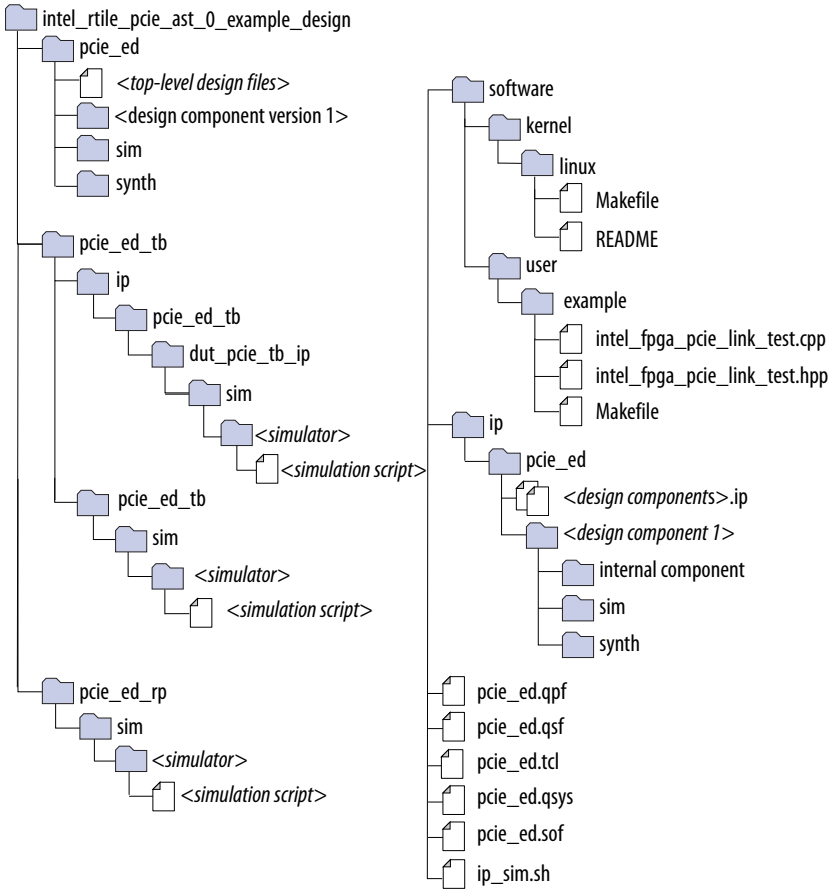
Using the Intel Quartus Prime Pro Edition software, you can generate a programmed I/O (PIO) design example for the Intel FPGA R-tile Avalon-ST Hard IP for PCI Express* IP core. The generated design example reflects the parameters that you specify. The PIO example transfers data from a host processor to a target device. It is appropriate for low-bandwidth applications. This design example automatically creates the files necessary to simulate and compile in the Intel Quartus Prime Pro Edition software. You can download a compiled version of this PIO design example to the Intel Agilex 7 I-Series ES0 FPGA Development Board for evaluation. To download to custom hardware, update the Intel Quartus Prime Settings File (`.qsf`) with the correct pin assignments .

Figure 13. Development Steps for the Design Example



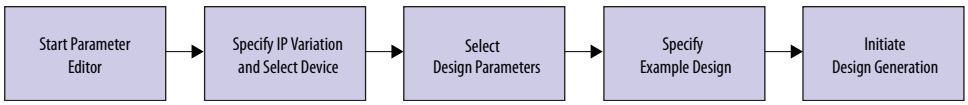
2.1. Directory Structure

Figure 14. Directory Structure for the Generated Design Example



2.2. Generating the Design Example

Figure 15. Procedure



1. In the Intel Quartus Prime Pro Edition software, create a new project (**File ➤ New Project Wizard**).
2. Specify the **Directory, Name**, and **Top-Level Entity.**
3. For **Project Type**, accept the default value, **Empty project**. Click **Next**.
4. For **Add Files** click **Next**.
5. For **Family, Device & Board Settings** under **Family**, select **Intel Agilex 7 I-Series**.

6. Select the **Target Device** for your design.
7. Click **Finish**.
8. In the IP Catalog locate and add the **Intel R-Tile Avalon-ST Hard IP for PCI Express**.
9. In the **New IP Variant** dialog box, specify a name for your IP. Click **Create**.
10. On the **Top-Level Settings** and **PCIe* Settings** tabs, specify the parameters for your IP variation.

    If you want to generate the SR-IOV design example, perform the following steps to enable SR-IOV:

    a. On the **Top-Level Settings** tab, set the **PCIe Hard IP Mode** parameter to **Gen5 1x16, interface - 1024 bit**

    b. On the **PCIe0 Settings** tab, navigate to **PCIe0 PCI Express/PCI Capabilities** > **PCIe0 Device** > **PCIe0 Multifunction and SR-IOV System Settings** and:

        i. Check the **Enable Multiple Physical Functions** parameter.

        ii. Set the **Total Physical Functions (PFs)** parameter to 2.

        iii. Check the **Enable SR-IOV Support** parameter.

        iv. Set the **Total Virtual Functions of Physical Function 0 (PF0 VFs)** parameter to 16.

        v. Set the **Total Virtual Functions of Physical Function 1 (PF1 VFs)** parameter to 16.

    c. On the **PCIe0 Settings** tab, navigate to **PCIe0 Base Address Registers** > **PCIe0 PF0 BAR Configuration** > **PCIe0 PF0 BAR** and:

        i. Set the **BAR0 Type** parameter to **64-bit prefetchable memory** or **64-bit non-prefetchable memory** or **32-bit non-prefetchable memory**.

        ii. Set the **BAR0 Size** parameter to any value other than **N/A**.

    d. Repeat steps i and ii for **PCIe0 PF0 BAR** above in the **PCIe0 PF0 VF BAR** section.

    e. On the **PCIe0 Settings** tab, navigate to **PCIe0 Base Address Registers** > **PCIe0 PF1 BAR Configuration** > **PCIe0 PF1 BAR** and:

        i. Set the **BAR0 Type** parameter to **64-bit prefetchable memory** or **64-bit non-prefetchable memory** or **32-bit non-prefetchable memory**.

        ii. Set the **BAR0 Size** parameter to any value other than **N/A**.

    f. Repeat steps i and ii for **PCIe0 PF1 BAR** above in the **PCIe0 PF1 VF BAR** section.

    g. On the **PCIe0 Settings** tab, navigate to **PCIe0 PCI Express/PCI Capabilities** > **PCIe0 MSI-X** > **PCIe0 PF MSI-X** > **PCIe0 PF0 MSI-X** and check the **Enable MSI-X** parameter.

    h. On the **PCIe0 Settings** tab, navigate to **PCIe0 PCI Express/PCI Capabilities** > **PCIe0 MSI-X** > **PCIe0 PF MSI-X** > **PCIe0 PF1 MSI-X** and check the **Enable MSI-X** parameter.

    i. On the **PCIe0 Settings** tab, navigate to **PCIe0 Device Identification Registers** and set the following parameters for the PCIe0 PF0 IDs and the PCIe0 PF1 IDs:

- Vendor ID: 0x00001172
- Device ID: 0x00000000
- Revision ID: 0x00000001
- Class Code: 0x00ff0000
- Subsystem Vendor ID: 0x00001172
- Subsystem Device ID: 0x00000000

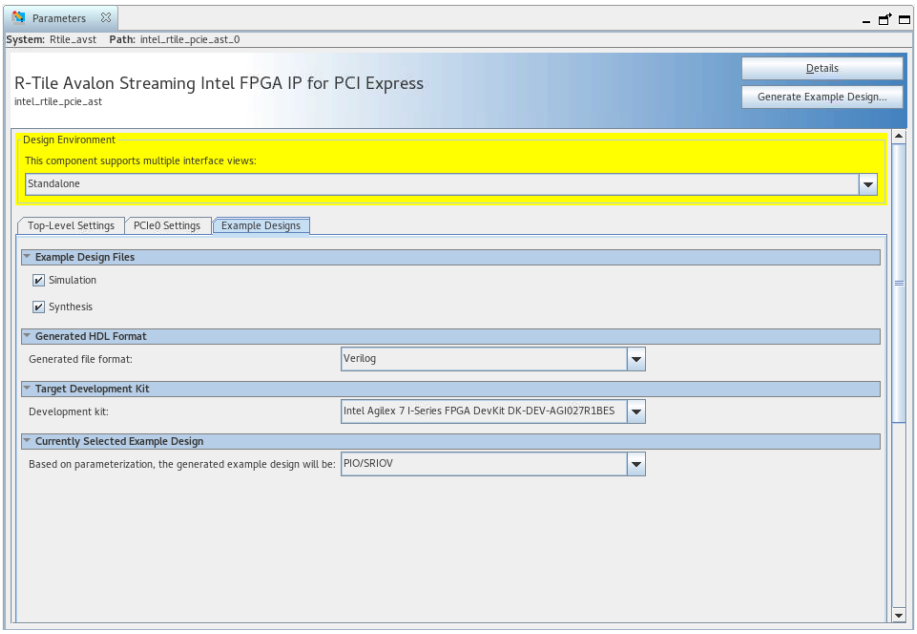
11. On the **Example Designs** tab, make the following selections:
    a. For **Available Example Designs,** select the **PIO**, **SR-IOV** or **Performance** design example.
    b. For **Example Design Files**, turn on the **Simulation** and **Synthesis** options. If you do not need these simulation or synthesis files, leaving the corresponding option(s) turned off significantly reduces the example design generation time.
    c. For **Generated HDL Format**, only Verilog is available in the current release.
    d. For **Target Development Kit**, select the appropriate development kit.

    Note: If you select **None**, the generated design example targets the device you specified in Step 5 above. If you intend to test the design in hardware, make the appropriate pin assignments in the `.qsf` file. You can also use the pin planner tool to make pin assignments.

    Note: If you select a development kit, the device on that board overwrites the device selected in the Intel Quartus Prime project if the devices are different.

    Note: For **Currently Selected Example Design**, select **PIO/SRIOV** or **PERFORMANCE_DESIGN**.
12. Select **Generate Example Design** to create a design example that you can simulate and download to hardware. If you select one of the R-tile development boards, the device on that board overwrites the device previously selected in the Intel Quartus Prime project if the devices are different. When the prompt asks you to specify the directory for your example design, you can accept the default directory, `./intel_rtile_pcie_ast_0_example_design`, or choose another directory.

Figure 16. IP Parameter Editor Screen for Generating Example Design



13. Click **Finish**. You may save your `.ip` file when prompted, but it is not required to be able to use the example design.
14. Close the current open project.
15. Open the example design project. This is the new project that has been generated in the location specified in step 12.
16. Compile the example design project to generate the `.sof` file for the complete example design.
17. Close your example design project.

    Note that you cannot change the PCIe pin allocations in the Intel Quartus Prime project. However, to ease PCB routing, you can take advantage of the lane reversal and polarity inversion features supported by this IP.

2.3. Simulating the Design Example

The simulation setup involves the use of a Root Port Bus Functional Model (BFM) to exercise the R-tile Avalon Streaming Intel FPGA IP for PCIe (DUT) as shown in the following figure.

Figure 17. PIO Design Example Simulation Testbench

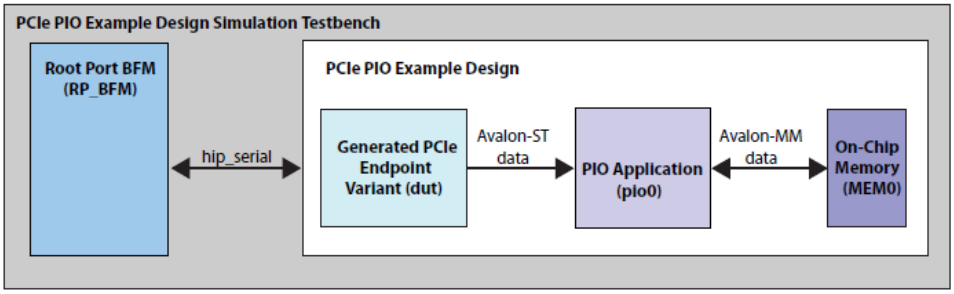
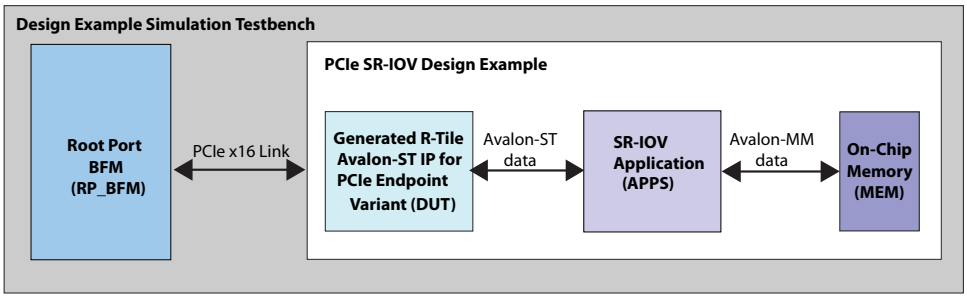


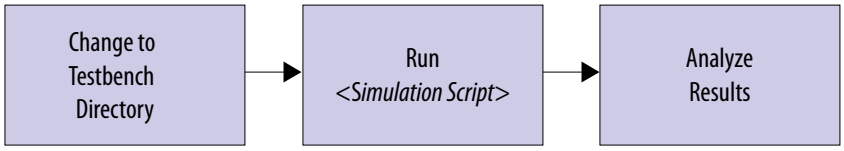
Figure 18. SR-IOV Design Example Simulation Testbench



For a more detailed description of the testbench and the modules inside it, refer to Testbench on page 24. Also, for more details on the Root Port BFM specifically, refer to the Root Port BFM on page 29 section.

The following flow diagram shows the steps to simulate the design example:

Figure 19. Procedure



Note: **R-tile does not support parallel PIPE simulations.**

The following figure shows the link status information for a Gen5 x16 Endpoint simulation:

Figure 20. Link Status for a Gen5 x16 Endpoint Simulation

```
INFO:          188405 ns   EP PCI Express Link Status Register (0105):
INFO:          188405 ns     Negotiated Link Width: x16
INFO:          188405 ns         Slot Clock Config: Local Clock Used
INFO:          188405 ns             New Link Speed: 32.0GT/s
INFO:          188405 ns
INFO:          188829 ns   EP PCI Express Link Control Register (0000):
INFO:          188829 ns       Common Clock Config: Local Clock Used
INFO:          188829 ns
INFO:          189181 ns
INFO:          189181 ns   EP PCI Express Capabilities Register (0002):
INFO:          189181 ns       Capability Version: 2
INFO:          189181 ns                Port Type: Native Endpoint
```

After a successful simulation, the `simulation.log` file contains a "successful completion" message.

This testbench simulates up to a Gen5 x16 variant.

2.3.1. Steps to Run Simulations

The following sections describe the necessary steps to simulate the design example in each of the simulators supported by the R-Tile Avalon Streaming IP.

2.3.1.1. Siemens EDA QuestaSim Simulator

Perform the following steps:

1. Change to the simulation working directory: cd `<my_design>/pcie_ed_tb/pcie_ed_tb/sim/mentor`.
2. Invoke vsim, which brings up a console window where you can run the next commands: Type `vsim`
   a. set TOP_LEVEL_NAME "pcie_ed_tb.pcie_ed_tb"
   b. `do msim_setup.tcl`
   c. `ld_debug`
   d. `run -all`

A successful simulation includes the following message: "Simulation stopped due to successful completion!"

Note: When running simulations under Windows* OS, if your project path is too long, you may encounter access errors to the design files. To avoid this, shorten your project path as much as possible. The longest path for any design file should be less than 189 characters.

2.3.1.2. VCS Simulator

Perform the following steps to execute the simulation via a command line:

1. Change to the simulation working directory: cd `<my_design>/pcie_ed_tb/pcie_ed_tb/sim/synopsys/vcs`.
2. Execute the following command:
```
sh vcs_setup.sh USER_DEFINED_COMPILE_OPTIONS="" USER_DEFINED_ELAB_OPTIONS="-xlrm\ uniq_prior_final\ -debug_access+all" USER_DEFINED_SIM_OPTIONS="" TOP_LEVEL_NAME="pcie_ed_tb" | tee simulation.log
```
   Note: The command above is a single-line command.

A successful simulation includes the following message: "Simulation stopped due to successful completion!"

Perform the following steps to execute the simulation in interactive mode. Note that in case you have already generated a simv executable in non-interactive mode, you need to delete the simv file and simv.diadir directory.

1. Open the vcs_setup.sh file and add a debug option to the VCS command: `vcs -kdb -debug_access+all`
2. Execute the following command: `sh vcs_setup.sh USER_DEFINED_ELAB_OPTIONS="-xlrm\ uniq_prior_final\ -debug_access+all" SKIP_SIM=1 TOP_LEVEL_NAME="pcie_ed_tb"`
3. Start the simulation in interactive mode: `simv -gui &`

2.3.1.3. VCS MX Simulator

Perform the following steps to execute the simulation via a command line:

1. Change to the simulation working directory: cd `<my_design>/pcie_ed_tb/pcie_ed_tb/sim/synopsys/vcsmx`.
2. Execute the following command:
```
sh vcsmx_setup.sh
USER_DEFINED_COMPILE_OPTIONS="" USER_DEFINED_ELAB_OPTIONS="-xlrm\ uniq_prior_final\ -debug_access+all"
USER_DEFINED_SIM_OPTIONS=""
TOP_LEVEL_NAME="pcie_ed_tb.pcie_ed_tb" | tee simulation.log
```
   Note: The command above is a single-line command.

A successful simulation includes the following message: "Simulation stopped due to successful completion!"

Perform the following steps to execute the simulation in interactive mode. Note that in case you have already generated a simv executable in non-interactive mode, you need to delete the simv file and simv.diadir directory.

1. Execute the following command:
```
sh vcsmx_setup.sh
USER_DEFINED_COMPILE_OPTIONS="-kdb"
USER_DEFINED_ELAB_OPTIONS="-debug_access+all"
USER_DEFINED_ELAB_OPTIONS="-xlrm\ uniq_prior_final"
USER_DEFINED_SIM_OPTIONS="" SKIP_SIM=1
TOP_LEVEL_NAME="pcie_ed_tb.pcie_ed_tb"
```
2. Start the simulation in interactive mode: `simv -gui &`

2.3.1.4. Xcelium Simulator

Note: Xcelium simulator support is only available in devices with the following OPN numbers:

- AGIx027R29AxxxxR3
- AGIx027R29AxxxxR2
- AGIx023R18AxxxxR0
- AGIx041R29DxxxxR0
- AGIx041R29DxxxxR1

For more details on OPN decoding, refer to the Intel Agilex 7 FPGAs and SoCs Device Overview.

Perform the following steps to execute the simulation via a command line:

1. Export the following environment variables:

a. `export CADENCE_ENABLE_AVSREQ_6614_PHASE_1=1`

b. `export CADENCE_ENABLE_AVSREQ_12055_PHASE_1=1`

2. Change to the simulation working directory: `cd <my_design>/pcie_ed_tb/pcie_ed_tb/sim/xcelium`

3. Execute the following command:
```
sh xcelium_setup.sh
USER_DEFINED_VERILOG_COMPILE_OPTIONS="-sv\ "
USER_DEFINED_ELAB_OPTIONS="-timescale\ 1ns/1ps"
USER_DEFINED_SIM_OPTIONS="-input\ @run"
TOP_LEVEL_NAME="pcie_ed_tb.pcie_ed_tb" | tee simulation.log
```
Note: The command above is a single-line command.

A successful simulation includes the following message: "Simulation stopped due to successful completion!"

Figure 21. Successful Simulation Message

```
INFO:          195229 ns BAR Address Assignments:
INFO:          195229 ns BAR    Size       Assigned Address  Type
INFO:          195229 ns ---    ----       ----------------
INFO:          195229 ns BAR0    64 KBytes          00200000 Non-Prefetchable
INFO:          195229 ns BAR1   Disabled
INFO:          195229 ns BAR2   Disabled
INFO:          195229 ns BAR3   Disabled
INFO:          195229 ns BAR4   Disabled
INFO:          195229 ns BAR5   Disabled
INFO:          195229 ns ExpROM Disabled
INFO:          195837 ns
INFO:          195837 ns Completed configuration of Endpoint BARs.
INFO:          196309 ns ---------
INFO:          196813 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          197309 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          197813 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          198309 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          198805 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          199301 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          199805 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          200309 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          200805 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
INFO:          201301 ns Passed: 0004 same bytes in BFM mem addr 0x00000040 and 0x00000840
SUCCESS: Simulation stopped due to successful completion!
Simulation passed
```

2.3.2. Testbench

The testbench uses a test driver module, `altpcietb_bfm_rp_gen5_x16.sv`, to initiate the configuration and memory transactions. At startup, the test driver module displays information from the Root Port and Endpoint Configuration Space registers, so that you can correlate to the parameters you specified using the Parameter Editor.

The example design and testbench are dynamically generated based on the configuration that you choose for the R-tile IP for PCIe. The testbench uses the parameters that you specify in the Parameter Editor in Intel Quartus Prime.

This testbench simulates a Gen5 ×16 PCI Express link using the serial PCI Express interface.

Figure 22. PIO Design Example Simulation Testbench

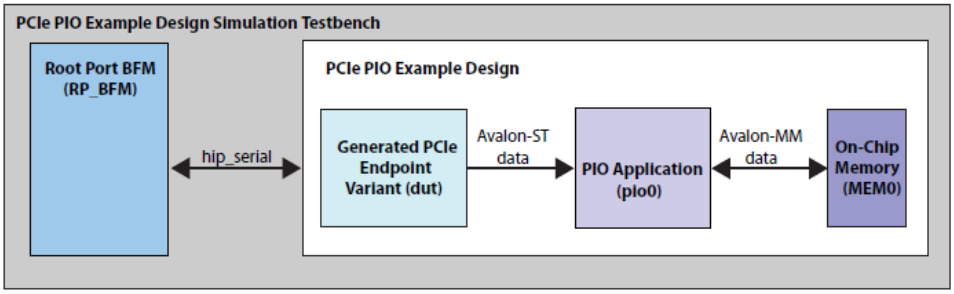
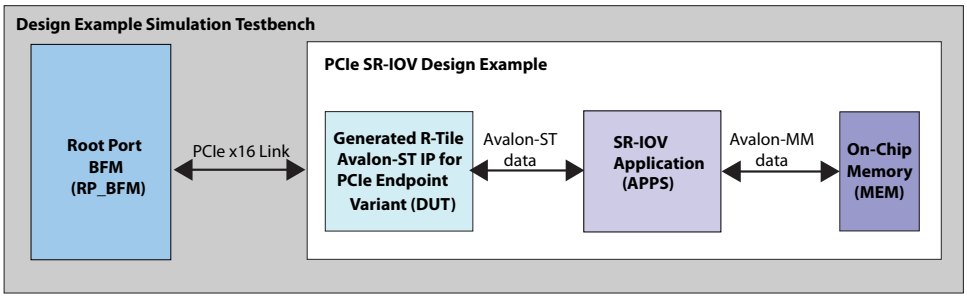


Figure 23. SR-IOV Design Example Simulation Testbench



When configured as an Endpoint variation, the testbench instantiates a design example with a R-tile Endpoint and a Root Port BFM containing a second R-tile (configured as a Root Port) to interface with the Endpoint. For more details on this Root Port BFM, refer to *Section 2.5: Root Port BFM*. The Root Port BFM provides the following functions:

- A configuration routine that sets up all the basic configuration registers in the Endpoint. This configuration allows the Endpoint application to be the target and initiator of PCI Express transactions.
- A Verilog HDL procedure interface to initiate PCI Express transactions to the Endpoint.

This testbench simulates the scenario of a single Root Port talking to a single Endpoint.

The testbench uses a test driver module, altpcietb_bfm_rp_gen5_x16.sv, to initiate the configuration and memory transactions. At startup, the test driver module displays information from the Root Port and Endpoint Configuration Space registers, so that you can correlate to the parameters you specified using the Parameter Editor.

Note: The Intel testbench and Root Port BFM provide a simple method to do basic testing of the Application Layer logic that interfaces to the variation. This BFM allows you to create and run simple task stimuli to exercise basic functionality of the Intel example design. The testbench and Root Port BFM are not intended to be a substitute for a full verification environment. Corner cases and certain traffic profile stimuli are not covered. Refer to the items listed below for further details. To ensure the best verification coverage possible, Intel suggests strongly that you obtain commercially available PCI Express verification IP and tools, in combination with performing extensive hardware testing.

Your Application Layer design may need to handle at least the following scenarios that are not possible to create with the Intel testbench and the Root Port BFM, or are due to the limitations of the example design:

- It is unable to generate or receive Vendor Defined Messages. Some systems generate Vendor Defined Messages. The Hard IP block simply passes these messages on to the Application Layer. Consequently, you should make the decision, based on your application, whether to design the Application Layer to process them.
- It can only handle received read requests that are less than or equal to 256 bits or 8 DW. Many systems are capable of handling larger read requests that are then returned in multiple completions.
- It always returns a single completion for every read request. Some systems split completions on every 64-byte address boundary.
- It always returns completions in the same order the read requests were issued. Some systems generate the completions out-of-order.
- It is unable to generate zero-length read requests that some systems generate as flush requests following some write transactions. The Application Layer must be capable of generating the completions to the zero-length read requests.
- It does not support parity.
- It does not support multi-function designs.
- It does not support Single Root I/O Virtualization (SR-IOV).

2.3.2.1. Testbench Modules

The top-level of the testbench instantiates the following main modules:

- `altpcietb_bfm_rp_gen5x16.sv` —This is the Root Port PCIe BFM.

  ```
  //Directory path
  <project_dir>/intel_rtile_pcie_ast_0_example_design/pcie_ed_tb/ip/
  pcie_ed_tb/dut_pcie_tb_ip/intel_rtile_pcie_tbed_<ver>/sim
  ```

- `pcie_ed_dut.ip`: This is the Endpoint design with the parameters that you specify.

  ```
  //Directory path
  <project_dir>/intel_rtile_pcie_ast_0_example_design/ip/pcie_ed
  ```

- `pcie_ed_pio0.ip`: This module is a target and initiator of transactions for the PIO design example.

  ```
  //Directory path
  <project_dir>/intel_rtile_pcie_ast_0_example_design/ip/pcie_ed
  ```

- `pcie_ed_sriov0.ip`: This module is a target and initiator of transactions for the SR-IOV design example.

  ```
  //Directory path
  <project_dir>/intel_rtile_pcie_ast_0_example_design/ip/pcie_ed
  ```

In addition, the testbench has routines that perform the following tasks:

- Generates the reference clock for the Endpoint at the required frequency.
- Provides a PCI Express reset at start up.

2.3.2.2. Test Driver Module

The test driver module, `intel_rtile_pcie_tbed_hwtcl.v`, instantiates the top-level BFM,`altpcietb_bfm_top_rp.v`.

The top-level BFM completes the following tasks:

1. Instantiates the driver and monitor.
2. Instantiates the Root Port BFM.
3. Instantiates the serial interface.

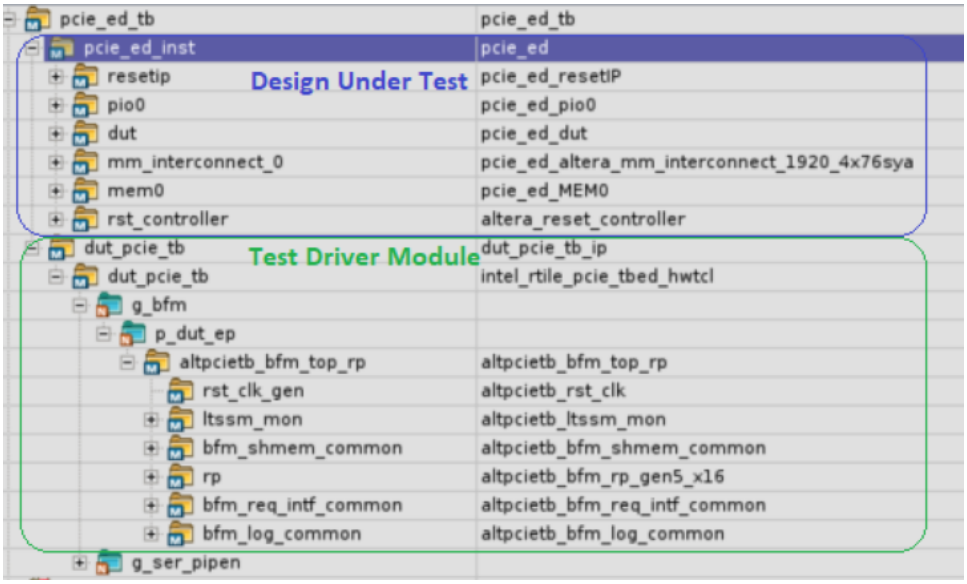
The configuration module, `altpcietb_g3bfm_configure.v`, performs the following tasks:

1. Configures and assigns the BARs.
2. Configures the Root Port and Endpoint.
3. Displays comprehensive Configuration Space, BAR, MSI, MSI-X, and AER settings.

2.3.2.3. PIO Design Example Testbench

The figure below shows the PIO design example simulation design hierarchy. The tests for the PIO design example are defined with the `apps_type_hwtcl` parameter set to 3. The tests run under this parameter value are defined in `ebfm_cfg_rp_ep_rootport`, `find_mem_bar` and `downstream_loop`.

Figure 24. PIO Design Example Simulation Design Hierarchy



The testbench starts with link training and then accesses the configuration space of the IP for enumeration. A task called `downstream_loop` (defined in the Root Port PCIe BFM `altpcietb_bfm_rp_gen5_x16.sv`) then performs the PCIe link test. This test consists of the following steps:

1. Issue a memory write command to write a single dword of data into the on-chip memory behind the Endpoint.
2. Issue a memory read command to read back data from the on-chip memory.
3. Compare the read data with the write data. If they match, the test counts this as a Pass.
4. Repeat Steps 1, 2 and 3 for 10 iterations.

The first memory write takes place around 219 us. It is followed by a memory read at the Avalon-ST RX interface of the R-tile Hard IP for PCIe. The Completion TLP appears shortly after the memory read request at the Avalon-ST TX interface.

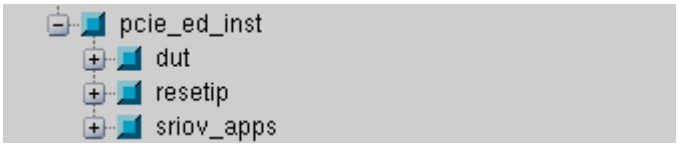
Note: In the 2x8 design example, memory read and memory write transactions are simulated on Port 0 only.

2.3.2.4. SR-IOV Design Example Testbench

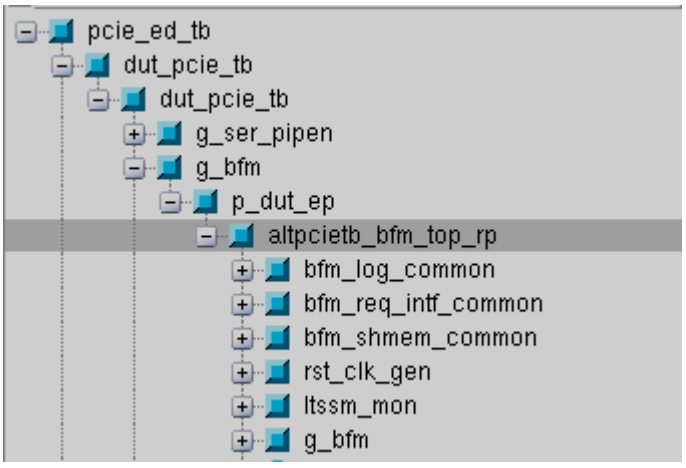
The figure below shows the SR-IOV design example simulation design hierarchy. The tests for the SR-IOV design example are defined with the `apps_type_hwtcl` parameter. The tests run under this parameter value are defined in `ebfm_cfg_rp_ep_rootport`, `find_mem_bar` and `downstream_loop`.

Figure 25. SR-IOV Design Example Simulation Design Hierarchy

Device Under Test



Test Driver Module



The SR-IOV testbench supports up to two Physical Functions (PFs) and 16 Virtual Functions (VFs) per PF.

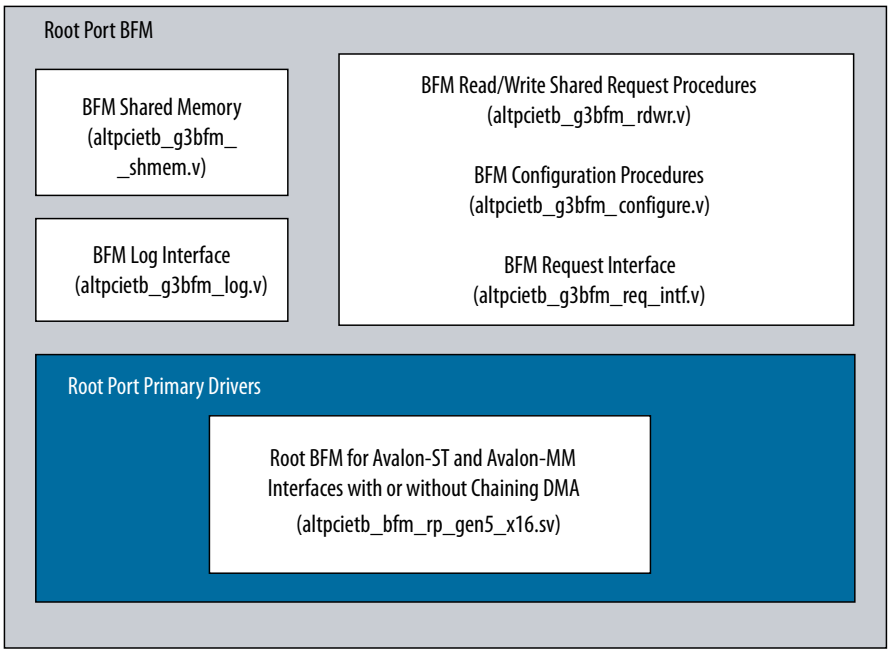
The testbench starts with link training and then accesses the configuration space of the R-Tile PCIe IP for enumeration. After that, it performs the following steps:

1. Send a memory write request to a PF followed by a memory read request to read back the same data for comparison. The test passes if the read data matches the write data.
2. Send a memory write request to a VF followed by a memory read request to read back the data for comparison. The test passes if the read data matched the write data. This test is repeated for each VF.

2.4. Root Port BFM

The basic Root Port BFM provides a Verilog HDL task-based interface to request transactions to issue on the PCI Express link. The Root Port BFM also handles requests received from the PCI Express link. The following figure shows the major modules in the Root Port BFM.

Figure 26. Root Port BFM



These modules implement the following functionality:

- BFM Log Interface, `altpcietb_g3bfm_log.v` and `altpcietb_bfm_rp_gen5_x16.sv`: The BFM Log Interface provides routines for writing commonly formatted messages to the simulator standard output and optionally to a log file. It also provides controls that stop simulations on errors.
- BFM Read/Write Request Functions, `altpcietb_bfm_rp_gen5_x16.sv`: These functions provide the basic BFM calls for PCI Express read and write requests.
- BFM Configuration Functions, `altpcietb_g3bfm_configure.v` : These functions provide the BFM calls to request a configuration of the PCI Express link and the Endpoint Configuration Space registers.

- BFM shared memory, `altpcietb_g3bfm_shmem.v`: This module provides the Root Port BFM shared memory. It implements the following functionality:
  - Provides data for TX write operations
  - Provides data for RX read operations
  - Receives data for RX write operations
  - Receives data for received completions
- BFM Request Interface, `altpcietb_g3bfm_req_intf.v`: This interface provides the low-level interface between the `altpcietb_g3bfm_rdwr` and `altpcietb_g3bfm_configure` procedures or functions and the Root Port RTL Model. This interface stores a write-protected data structure containing the sizes and values programmed in the BAR registers of the Endpoint. It also stores other critical data used for internal BFM management.
- `altpcietb_g3bfm_rdwr.v`: This module contains the low-level read and write tasks.
- Avalon-ST Interfaces, `altpcietb_g3bfm_vc_intf_ast_common.v`: These interface modules handle the Root Port interface model. They take requests from the BFM request interface and generate the required PCI Express transactions. They handle completions received from the PCI Express link and notify the BFM request interface when requests are complete. Additionally, they handle any requests received from the PCI Express link, and store or fetch data from the shared memory before generating the required completions.

In the PIO design example, the `apps_type_hwtcl` parameter is set to 3. The tests run under this parameter value are defined in `ebfm_cfg_rp_ep_rootport`, `find_mem_bar` and `downstream_loop`.

The function `ebfm_cfg_rp_ep_rootport` is described in `altpcietb_g3bfm_configure.v`. This function performs the steps necessary to configure the root port and the endpoint on the link. It includes:

- Root port memory allocation
- Root port configuration space (base limit, bus number, etc.)
- Endpoint configuration (BAR, Bus Master enable, maxpayload size, etc.)

The functions `find_mem_bar` and `downstream_loop` in `altpcietb_bfm_rp_gen5_x16.sv` return the BAR implemented and perform the memory Write and Read accesses to the BAR, respectively.

2.4.1. BFM Memory Map

The BFM shared memory is 2 MBs. The BFM shared memory maps to the first 2 MBs of I/O space and also the first 2 MBs of memory space. When the Endpoint application generates an I/O or memory transaction in this range, the BFM reads or writes the shared memory.

2.4.2. Configuration Space Bus and Device Numbering

Enumeration assigns the Root Port interface device number 0 on internal bus number 0. Use the `ebfm_cfg_rp_ep` procedure to assign the Endpoint to any device number on any bus number (greater than 0). The specified bus number is the secondary bus in the Root Port Configuration Space.

2.4.3. Configuration of Root Port and Endpoint

Before you issue transactions to the Endpoint, you must configure the Root Port and Endpoint Configuration Space registers.

The `ebfm_cfg_rp_ep` procedure in `altpcietb_g3bfm_configure.v` executes the following steps to initialize the Configuration Space:

1. Sets the Root Port Configuration Space to enable the Root Port to send transactions on the PCI Express link.
2. Sets the Root Port and Endpoint PCI Express Capability Device Control registers as follows:
   a. Disables `Error Reporting` in both the Root Port and Endpoint. The BFM does not have error handling capability.
   b. Enables `Relaxed Ordering` in both Root Port and Endpoint.
   c. Enables `Extended Tags` for the Endpoint if the Endpoint has that capability.
   d. Disables `Phantom Functions`, `Aux Power PM`, and `No Snoop` in both the Root Port and Endpoint.
   e. Sets the `Max Payload Size` to the value that the Endpoint supports because the Root Port supports the maximum payload size.
   f. Sets the Root Port `Max Read Request Size` to 4 KB because the example Endpoint design supports breaking the read into as many completions as necessary.
   g. Sets the Endpoint `Max Read Request Size` equal to the `Max Payload Size` because the Root Port does not support breaking the read request into multiple completions.
3. Assigns values to all the Endpoint BAR registers. The BAR addresses are assigned by the algorithm outlined below.
   a. I/O BARs are assigned smallest to largest starting just above the ending address of the BFM shared memory in I/O space and continuing as needed throughout a full 32-bit I/O space.
   b. The 32-bit non-prefetchable memory BARs are assigned smallest to largest, starting just above the ending address of the BFM shared memory in memory space and continuing as needed throughout a full 32-bit memory space.
   c. The value of the `addr_map_4GB_limit` input to the `ebfm_cfg_rp_ep` procedure controls the assignment of the 32-bit prefetchable and 64-bit prefetchable memory BARS. The default value of the `addr_map_4GB_limit` is `0`.

If the `addr_map_4GB_limit` input to the `ebfm_cfg_rp_ep` procedure is set to 0, then the `ebfm_cfg_rp_ep` procedure assigns the 32-bit prefetchable memory BARs largest to smallest, starting at the top of 32-bit memory space and continuing as needed down to the ending address of the last 32-bit non-prefetchable BAR.

However, if the `addr_map_4GB_limit` input is set to 1, the address map is limited to 4 GB. The `ebfm_cfg_rp_ep` procedure assigns 32-bit and 64-bit prefetchable memory BARs largest to smallest, starting at the top of the 32-bit memory space and continuing as needed down to the ending address of the last 32-bit non-prefetchable BAR.

d. If the `addr_map_4GB_limit` input to the `ebfm_cfg_rp_ep` procedure is set to 0, then the `ebfm_cfg_rp_ep` procedure assigns the 64-bit prefetchable memory BARs smallest to largest starting at the 4 GB address assigning memory ascending above the 4 GB limit throughout the full 64-bit memory space.

   If the `addr_map_4 GB_limit` input to the `ebfm_cfg_rp_ep` procedure is set to 1, the `ebfm_cfg_rp_ep` procedure assigns the 32-bit and the 64-bit prefetchable memory BARs largest to smallest starting at the 4 GB address and assigning memory by descending below the 4 GB address to memory addresses as needed down to the ending address of the last 32-bit non-prefetchable BAR.

   The above algorithm cannot always assign values to all BARs when there are a few very large (1 GB or greater) 32-bit BARs. Although assigning addresses to all BARs may be possible, a more complex algorithm would be required to effectively assign these addresses. However, such a configuration is unlikely to be useful in real systems. If the procedure is unable to assign the BARs, it displays an error message and stops the simulation.

4. Based on the above BAR assignments, the `ebfm_cfg_rp_ep` procedure assigns the Root Port Configuration Space address windows to encompass the valid BAR address ranges.
5. The `ebfm_cfg_rp_ep` procedure enables master transactions, memory address decoding, and I/O address decoding in the Endpoint PCIe control register.

The `ebfm_cfg_rp_ep` procedure also sets up a `bar_table` data structure in BFM shared memory that lists the sizes and assigned addresses of all Endpoint BARs. This area of BFM shared memory is write-protected. Consequently, application logic write accesses to this area cause a fatal simulation error.

BFM procedure calls to generate full PCIe addresses for read and write requests to particular offsets from a BAR use this data structure. This procedure allows the testbench code that accesses the Endpoint application logic to use offsets from a BAR and avoid tracking specific addresses assigned to the BAR. The following table shows how to use those offsets.

Table 7. BAR Table Structure

Offset (Bytes)	Description
+0	PCI Express address in BAR0
+4	PCI Express address in BAR1
+8	PCI Express address in BAR2

continued...

Offset (Bytes)	Description
+12	PCI Express address in BAR3
+16	PCI Express address in BAR4
+20	PCI Express address in BAR5
+24	PCI Express address in Expansion ROM BAR
+28	Reserved
+32	BAR0 read back value after being written with all 1's (used to compute size)
+36	BAR1 read back value after being written with all 1's
+40	BAR2 read back value after being written with all 1's
+44	BAR3 read back value after being written with all 1's
+48	BAR4 read back value after being written with all 1's
+52	BAR5 read back value after being written with all 1's
+56	Expansion ROM BAR read back value after being written with all 1's
+60	Reserved

Note: The configuration routine does not configure any advanced PCI Express capabilities such as the AER capability.

Besides the `ebfm_cfg_rp_ep` procedure in `altpcietb_bfm_rp_gen5_x16.sv`, routines to read and write Endpoint Configuration Space registers directly are available in the Verilog HDL include file. After the `ebfm_cfg_rp_ep` procedure runs, the PCI Express I/O and Memory Spaces have the layout shown in the following three figures. The memory space layout depends on the value of the **addr_map_4GB_limit** input parameter.

Figure 27. Memory Space Layout—4 GB Limit

The following figure shows the resulting memory space map when the **addr_map_4GB_limit** is 1.

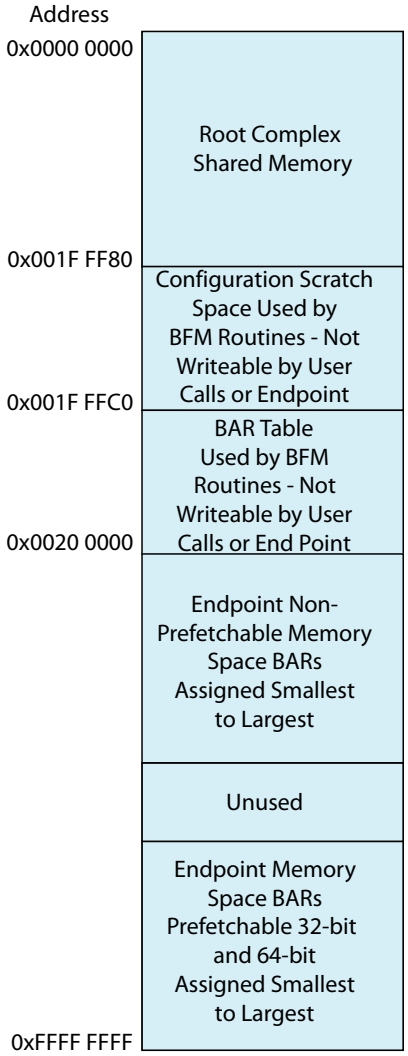


Figure 28. Memory Space Layout—No Limit

This figure shows the resulting memory space map when the **addr_map_4GB_limit** is 0.

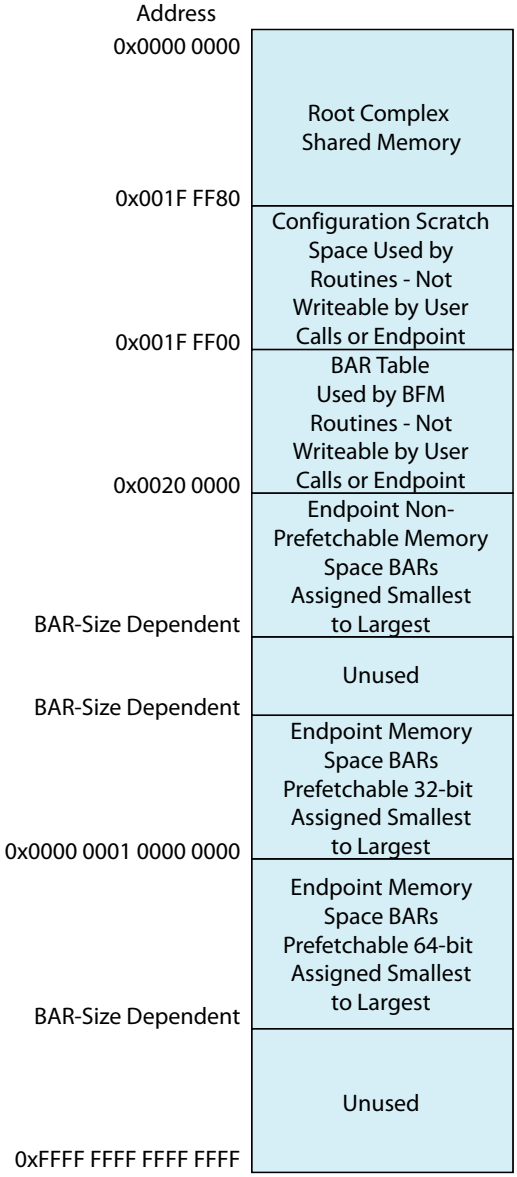
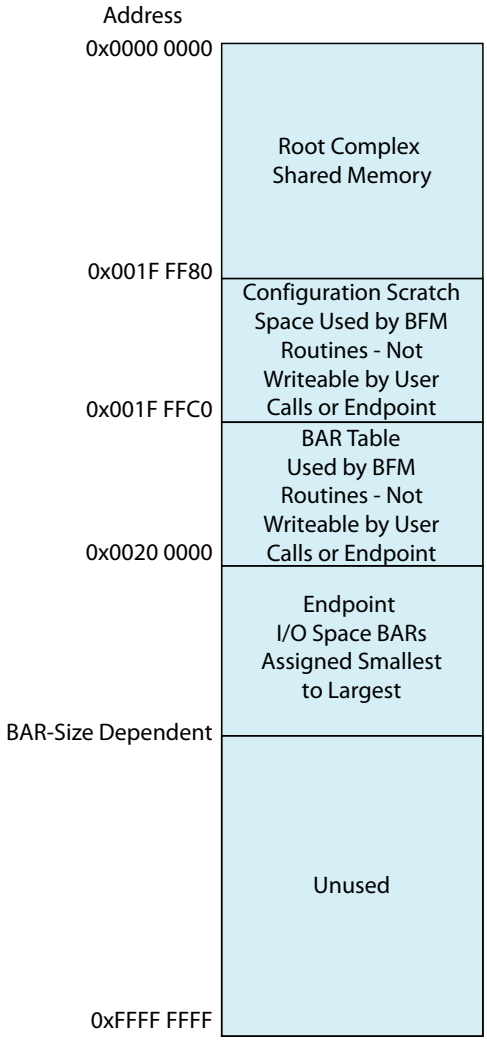


Figure 29. I/O Address Space

This figure shows the I/O address space.



2.4.4. Issuing Read and Write Transactions to the Application Layer

The Root Port Application Layer issues read and write transactions by calling one of the `ebfm_bar` procedures in `altpcietb_g3bfm_rdwr.v`. The procedures and functions listed below are available in the Verilog HDL include file `altpcietb_g3bfm_rdwr.v`. The complete list of available procedures and functions is as follows:

- `ebfm_barwr`: writes data from BFM shared memory to an offset from a specific Endpoint BAR. This procedure returns as soon as the request has been passed to the VC interface module for transmission.
- `ebfm_barwr_imm`: writes a maximum of four bytes of immediate data (passed in a procedure call) to an offset from a specific Endpoint BAR. This procedure returns as soon as the request has been passed to the VC interface module for transmission.
- `ebfm_barrd_wait`: reads data from an offset of a specific Endpoint BAR and stores it in BFM shared memory. This procedure blocks waiting for the completion data to be returned before returning control to the caller.
- `ebfm_barrd_nowt`: reads data from an offset of a specific Endpoint BAR and stores it in the BFM shared memory. This procedure returns as soon as the request has been passed to the VC interface module for transmission, allowing subsequent reads to be issued in the interim.

These routines take as parameters a BAR number to access the memory space and the BFM shared memory address of the `bar_table` data structure that was set up by the `ebfm_cfg_rp_ep` procedure. (Refer to *Configuration of Root Port and Endpoint*.) Using these parameters simplifies the BFM test driver routines that access an offset from a specific BAR and eliminates calculating the addresses assigned to the specified BAR.

Note: The Root Port BFM does not support accesses to Endpoint I/O space BARs.

2.4.5. BFM Procedures and Functions

The BFM includes procedures, functions, and tasks to drive Endpoint application testing. It also includes procedures to run the chaining DMA design example.

The BFM read and write procedures read and write data to BFM shared memory, Endpoint BARs, and specified configuration registers. The procedures and functions are available in the Verilog HDL. These procedures and functions support issuing memory and configuration transactions on the PCI Express link.

2.4.5.1. ebfm_barwr Procedure

The `ebfm_barwr` procedure writes a block of data from BFM shared memory to an offset from the specified Endpoint BAR. The length can be longer than the configured `MAXIMUM_PAYLOAD_SIZE`. The procedure breaks the request up into multiple transactions as needed. This routine returns as soon as the last transaction has been accepted by the VC interface module.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_barwr(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory. The `bar_table` structure stores the address assigned to each BAR so that the driver code does not need to be aware of the actual assigned addresses only the application specific offsets from the BAR.
	`bar_num`	Number of the BAR used with `pcie_offset` to determine PCI Express address.
	`pcie_offset`	Address offset from the BAR base.
	`lcladdr`	BFM shared memory address of the data to be written.
	`byte_len`	Length, in bytes, of the data written. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.
	`tclass`	Traffic class used for the PCI Express transaction.

2.4.5.2. ebfm_barwr_imm Procedure

The `ebfm_barwr_imm` procedure writes up to four bytes of data to an offset from the specified Endpoint BAR.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_barwr_imm(bar_table, bar_num, pcie_offset, imm_data, byte_len, tclass)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory. The `bar_table` structure stores the address assigned to each BAR so that the driver code does not need to be aware of the actual assigned addresses only the application specific offsets from the BAR.
	`bar_num`	Number of the BAR used with `pcie_offset` to determine PCI Express address.
	`pcie_offset`	Address offset from the BAR base.
	`imm_data`	Data to be written. In Verilog HDL, this argument is `reg [31:0]`.In both languages, the bits written depend on the length as follows: Length Bits Written • 4: 31 down to 0 • 3: 23 down to 0 • 2: 15 down to 0 • 1: 7 down to 0
	`byte_len`	Length of the data to be written in bytes. Maximum length is 4 bytes.
	`tclass`	Traffic class to be used for the PCI Express transaction.

2.4.5.3. ebfm_barrd_wait Procedure

The `ebfm_barrd_wait` procedure reads a block of data from the offset of the specified Endpoint BAR and stores it in BFM shared memory. The length can be longer than the configured maximum read request size; the procedure breaks the request up into multiple transactions as needed. This procedure waits until all of the completion data is returned and places it in shared memory.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_barrd_wait(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory. The bar_table structure stores the address assigned to each BAR so that the driver code does not need to be aware of the actual assigned addresses only the application specific offsets from the BAR.
	`bar_num`	Number of the BAR used with `pcie_offset` to determine PCI Express address.
	`pcie_offset`	Address offset from the BAR base.
	`lcladdr`	BFM shared memory address where the read data is stored.
	`byte_len`	Length, in bytes, of the data to be read. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.
	`tclass`	Traffic class used for the PCI Express transaction.

2.4.5.4. ebfm_barrd_nowt Procedure

The `ebfm_barrd_nowt` procedure reads a block of data from the offset of the specified Endpoint BAR and stores the data in BFM shared memory. The length can be longer than the configured maximum read request size; the procedure breaks the request up into multiple transactions as needed. This routine returns as soon as the last read transaction has been accepted by the VC interface module, allowing subsequent reads to be issued immediately.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_barrd_nowt(bar_table, bar_num, pcie_offset, lcladdr, byte_len, tclass)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory.
	`bar_num`	Number of the BAR used with `pcie_offset` to determine PCI Express address.
	`pcie_offset`	Address offset from the BAR base.
	`lcladdr`	BFM shared memory address where the read data is stored.
	`byte_len`	Length, in bytes, of the data to be read. Can be 1 to the minimum of the bytes remaining in the BAR space or BFM shared memory.
	`tclass`	Traffic Class to be used for the PCI Express transaction.

2.4.5.5. ebfm_cfgwr_imm_wait Procedure

The `ebfm_cfgwr_imm_wait` procedure writes up to four bytes of data to the specified configuration register. This procedure waits until the write completion has been returned.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_cfgwr_imm_wait(bus_num, dev_num, fnc_num, imm_regb_ad, regb_ln, imm_data, compl_status`	
Arguments	`bus_num`	PCI Express bus number of the target device.
	`dev_num`	PCI Express device number of the target device.
	`fnc_num`	Function number in the target device to be accessed.

continued...

Location	altpcietb_g3bfm_rdwr.v	
	regb_ad	Byte-specific address of the register to be written.
	regb_ln	Length, in bytes, of the data written. Maximum length is four bytes. The regb_ln and the regb_ad arguments cannot cross a DWORD boundary.
	imm_data	Data to be written. This argument is reg [31:0]. The bits written depend on the length: • 4: 31 down to 0 • 3: 23 down to 0 • 2: 15 down to 0 • 1: 7 down to 0
	compl_status	This argument is reg [2:0]. This argument is the completion status as specified in the PCI Express specification. The following encodings are defined: • 3'b000: SC— Successful completion • 3'b001: UR— Unsupported Request • 3'b010: CRS — Configuration Request Retry Status • 3'b100: CA — Completer Abort

2.4.5.6. ebfm_cfgwr_imm_nowt Procedure

The ebfm_cfgwr_imm_nowt procedure writes up to four bytes of data to the specified configuration register. This procedure returns as soon as the VC interface module accepts the transaction, allowing other writes to be issued in the interim. Use this procedure only when successful completion status is expected.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	ebfm_cfgwr_imm_nowt(bus_num, dev_num, fnc_num, imm_regb_adr, regb_len, imm_data)	
Arguments	bus_num	PCI Express bus number of the target device.
	dev_num	PCI Express device number of the target device.
	fnc_num	Function number in the target device to be accessed.
	regb_ad	Byte-specific address of the register to be written.
	regb_ln	Length, in bytes, of the data written. Maximum length is four bytes, The regb_ln the regb_ad arguments cannot cross a DWORD boundary.
	imm_data	Data to be written This argument is reg [31:0]. In both languages, the bits written depend on the length. The following encodes are defined. • 4: [31:0] • 3: [23:0] • 2: [15:0] • 1: [7:0]

2.4.5.7. ebfm_cfgrd_wait Procedure

The ebfm_cfgrd_wait procedure reads up to four bytes of data from the specified configuration register and stores the data in BFM shared memory. This procedure waits until the read completion has been returned.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_cfgrd_wait(bus_num, dev_num, fnc_num, regb_ad, regb_ln, lcladdr, compl_status)`	
Arguments	`bus_num`	PCI Express bus number of the target device.
	`dev_num`	PCI Express device number of the target device.
	`fnc_num`	Function number in the target device to be accessed.
	`regb_ad`	Byte-specific address of the register to be written.
	`regb_ln`	Length, in bytes, of the data read. Maximum length is four bytes. The `regb_ln` and the `regb_ad` arguments cannot cross a DWORD boundary.
	`lcladdr`	BFM shared memory address of where the read data should be placed.
	`compl_status`	Completion status for the configuration transaction. This argument is reg [2:0]. In both languages, this is the completion status as specified in the PCI Express specification. The following encodings are defined. • 3'b000: SC— Successful completion • 3'b001: UR— Unsupported Request • 3'b010: CRS — Configuration Request Retry Status • 3'b100: CA — Completer Abort

2.4.5.8. ebfm_cfgrd_nowt Procedure

The `ebfm_cfgrd_nowt` procedure reads up to four bytes of data from the specified configuration register and stores the data in the BFM shared memory. This procedure returns as soon as the VC interface module has accepted the transaction, allowing other reads to be issued in the interim. Use this procedure only when successful completion status is expected and a subsequent read or write with a wait can be used to guarantee the completion of this operation.

Location	altpcietb_g3bfm_rdwr.v	
Syntax	`ebfm_cfgrd_nowt(bus_num, dev_num, fnc_num, regb_ad, regb_ln, lcladdr)`	
Arguments	`bus_num`	PCI Express bus number of the target device.
	`dev_num`	PCI Express device number of the target device.
	`fnc_num`	Function number in the target device to be accessed.
	`regb_ad`	Byte-specific address of the register to be written.
	`regb_ln`	Length, in bytes, of the data written. Maximum length is four bytes. The `regb_ln` and `regb_ad` arguments cannot cross a DWORD boundary.
	`lcladdr`	BFM shared memory address where the read data should be placed.

2.4.5.9. BFM Configuration Procedures

The BFM configuration procedures are available in `altpcietb_bfm_rp_gen5_x16.sv` . These procedures support configuration of the Root Port and Endpoint Configuration Space registers.

All Verilog HDL arguments are type `integer` and are input-only unless specified otherwise.

2.4.5.9.1. ebfm_cfg_rp_ep Procedure

The `ebfm_cfg_rp_ep` procedure configures the Root Port and Endpoint Configuration Space registers for operation.

Location	altpcietb_g3bfm_configure.v	
Syntax	`ebfm_cfg_rp_ep(bar_table, ep_bus_num, ep_dev_num, rp_max_rd_req_size, display_ep_config, addr_map_4GB_limit)`	
Arguments	`bar_table`	Address of the Endpoint `bar_table` structure in BFM shared memory. This routine populates the `bar_table` structure. The `bar_table` structure stores the size of each BAR and the address values assigned to each BAR. The address of the `bar_table` structure is passed to all subsequent read and write procedure calls that access an offset from a particular BAR.
	`ep_bus_num`	PCI Express bus number of the target device. This number can be any value greater than 0. The Root Port uses this as the secondary bus number.
	`ep_dev_num`	PCI Express device number of the target device. This number can be any value. The Endpoint is automatically assigned this value when it receives the first configuration transaction.
	`rp_max_rd_req_size`	Maximum read request size in bytes for reads issued by the Root Port. This parameter must be set to the maximum value supported by the Endpoint Application Layer. If the Application Layer only supports reads of the `MAXIMUM_PAYLOAD_SIZE`, then this can be set to 0 and the read request size is set to the maximum payload size. Valid values for this argument are 0, 128, 256, 512, 1,024, 2,048 and 4,096.
	`display_ep_config`	When set to 1 many of the Endpoint Configuration Space registers are displayed after they have been initialized, causing some additional reads of registers that are not normally accessed during the configuration process such as the Device ID and Vendor ID.
	`addr_map_4GB_limit`	When set to 1 the address map of the simulation system is limited to 4 GB. Any 64-bit BARs are assigned below the 4 GB limit.

2.4.5.9.2. ebfm_cfg_decode_bar Procedure

The `ebfm_cfg_decode_bar` procedure analyzes the information in the BAR table for the specified BAR and returns details about the BAR attributes.

Location	altpcietb_bfm_configure.v	
Syntax	`ebfm_cfg_decode_bar(bar_table, bar_num, log2_size, is_mem, is_pref, is_64b)`	
Arguments	`bar_table`	Address of the Endpoint bar_table structure in BFM shared memory.
	`bar_num`	BAR number to analyze.
	`log2_size`	This argument is set by the procedure to the log base 2 of the size of the BAR. If the BAR is not enabled, this argument is set to 0.
	`is_mem`	The procedure sets this argument to indicate if the BAR is a memory space BAR (1) or I/O Space BAR (0).
	`is_pref`	The procedure sets this argument to indicate if the BAR is a prefetchable BAR (1) or non-prefetchable BAR (0).
	`is_64b`	The procedure sets this argument to indicate if the BAR is a 64-bit BAR (1) or 32-bit BAR (0). This is set to 1 only for the lower numbered BAR of the pair.

2.4.5.10. BFM Shared Memory Access Procedures

These procedures and functions support accessing the BFM shared memory.

2.4.5.10.1. Shared Memory Constants

The following constants are defined in `altrpcietb_g3bfm_shmem.v`. They select a data pattern for the `shmem_fill` and `shmem_chk_ok` routines. These shared memory constants are all Verilog HDL type `integer`.

Table 8. Constants: Verilog HDL Type INTEGER

Constant	Description
SHMEM_FILL_ZEROS	Specifies a data pattern of all zeros
SHMEM_FILL_BYTE_INC	Specifies a data pattern of incrementing 8-bit bytes (0x00, 0x01, 0x02, etc.)
SHMEM_FILL_WORD_INC	Specifies a data pattern of incrementing 16-bit words (0x0000, 0x0001, 0x0002, etc.)
SHMEM_FILL_DWORD_INC	Specifies a data pattern of incrementing 32-bit DWORDs (0x00000000, 0x00000001, 0x00000002, etc.)
SHMEM_FILL_QWORD_INC	Specifies a data pattern of incrementing 64-bit qwords (0x0000000000000000, 0x0000000000000001, 0x0000000000000002, etc.)
SHMEM_FILL_ONE	Specifies a data pattern of all ones

2.4.5.10.2. shmem_write Task

The `shmem_write` procedure writes data to the BFM shared memory.

Location	altpcietb_g3bfm_shmem.v	
Syntax	`shmem_write(addr, data, leng)`	
Arguments	`addr`	BFM shared memory starting address for writing data
	`data`	Data to write to BFM shared memory. This parameter is implemented as a 64-bit vector. `leng` is 1–8 bytes. Bits 7 down to 0 are written to the location specified by `addr`; bits 15 down to 8 are written to the `addr+1` location, etc.
	`length`	Length, in bytes, of data written

2.4.5.10.3. shmem_read Function

The `shmem_read` function reads data to the BFM shared memory.

Location	altpcietb_g3bfm_shmem.v	
Syntax	`data:= shmem_read(addr, leng)`	
Arguments	`addr`	BFM shared memory starting address for reading data
	`leng`	Length, in bytes, of data read
Return	`data`	Data read from BFM shared memory.

continued...

Location	altpcietb_g3bfm_shmem.v	
		This parameter is implemented as a 64-bit vector. `leng` is 1- 8 bytes. If `leng` is less than 8 bytes, only the corresponding least significant bits of the returned data are valid. Bits 7 down to 0 are read from the location specified by `addr`; bits 15 down to 8 are read from the addr+1 location, etc.

2.4.5.10.4. shmem_display Verilog HDL Function

The `shmem_display` Verilog HDL function displays a block of data from the BFM shared memory.

Location	altrpcietb_g3bfm_shmem.v	
Syntax	Verilog HDL: `dummy_return:=shmem_display(addr, leng, word_size, flag_addr, msg_type);`	
Arguments	`addr`	BFM shared memory starting address for displaying data.
	`leng`	Length, in bytes, of data to display.
	`word_size`	Size of the words to display. Groups individual bytes into words. Valid values are 1, 2, 4, and 8.
	`flag_addr`	Adds a <== flag to the end of the display line containing this address. Useful for marking specific data. Set to a value greater than 2**21 (size of BFM shared memory) to suppress the flag.
	`msg_type`	Specifies the message type to be displayed at the beginning of each line. See "BFM Log and Message Procedures" on page 18–37 for more information about message types. Set to one of the constants defined in Table 18–36 on page 18–41.

2.4.5.10.5. shmem_fill Procedure

The `shmem_fill` procedure fills a block of BFM shared memory with a specified data pattern.

Location	altrpcietb_g3bfm_shmem.v	
Syntax	`shmem_fill(addr, mode, leng, init)`	
Arguments	`addr`	BFM shared memory starting address for filling data.
	`mode`	Data pattern used for filling the data. Should be one of the constants defined in section *Shared Memory Constants*.
	`leng`	Length, in bytes, of data to fill. If the length is not a multiple of the incrementing data pattern width, then the last data pattern is truncated to fit.
	`init`	Initial data value used for incrementing data pattern modes. This argument is `reg [63:0]`. The necessary least significant bits are used for the data patterns that are smaller than 64 bits.

2.4.5.10.6. shmem_chk_ok Function

The `shmem_chk_ok` function checks a block of BFM shared memory against a specified data pattern.

Location	altrpcietb_g3bfm_shmem.v	
Syntax	`result:= shmem_chk_ok(addr, mode, leng, init, display_error)`	
Arguments	`addr`	BFM shared memory starting address for checking data.
	`mode`	Data pattern used for checking the data. Should be one of the constants defined in section "Shared Memory Constants" on page 18–35.
	`leng`	Length, in bytes, of data to check.
	`init`	This argument is `reg [63:0]`.The necessary least significant bits are used for the data patterns that are smaller than 64-bits.
	`display_error`	When set to 1, this argument displays the data failing comparison on the simulator standard output.
Return	`Result`	Result is 1-bit. • 1'b1 — Data patterns compared successfully • 1'b0 — Data patterns did not compare successfully

2.4.5.11. BFM Log and Message Procedures

The following procedures and functions are available in the Verilog HDL include file `altpcietb_bfm_log.v`

These procedures provide support for displaying messages in a common format, suppressing informational messages, and stopping simulation on specific message types.

The following constants define the type of message and their values determine whether a message is displayed or simulation is stopped after a specific message. Each displayed message has a specific prefix, based on the message type in the following table.

You can suppress the display of certain message types. The default values determining whether a message type is displayed are defined in the following table. To change the default message display, modify the display default value with a procedure call to `ebfm_log_set_suppressed_msg_mask`.

Certain message types also stop simulation after the message is displayed. The following table shows the default value determining whether a message type stops simulation. You can specify whether simulation stops for particular messages with the procedure `ebfm_log_set_stop_on_msg_mask`.

All of these log message constants type `integer`.

Table 9. Log Messages

Constant (Message Type)	Description	Mask Bit No	Display by Default	Simulation Stops by Default	Message Prefix
`EBFM_MSG_DEBUG`	Specifies debug messages.	0	No	No	`DEBUG:`
`EBFM_MSG_INFO`	Specifies informational messages, such as configuration register values, starting and ending of tests.	1	Yes	No	`INFO:`

continued...

Constant (Message Type)	Description	Mask Bit No	Display by Default	Simulation Stops by Default	Message Prefix
EBFM_MSG_WARNING	Specifies warning messages, such as tests being skipped due to the specific configuration.	2	Yes	No	WARNING:
EBFM_MSG_ERROR_INFO	Specifies additional information for an error. Use this message to display preliminary information before an error message that stops simulation.	3	Yes	No	ERROR:
EBFM_MSG_ERROR_CONTINUE	Specifies a recoverable error that allows simulation to continue. Use this error for data comparison failures.	4	Yes	No	ERROR:
EBFM_MSG_ERROR_FATAL	Specifies an error that stops simulation because the error leaves the testbench in a state where further simulation is not possible.	N/A	Yes Cannot suppress	Yes Cannot suppress	FATAL:
EBFM_MSG_ERROR_FATAL_TB_ERR	Used for BFM test driver or Root Port BFM fatal errors. Specifies an error that stops simulation because the error leaves the testbench in a state where further simulation is not possible. Use this error message for errors that occur due to a problem in the BFM test driver module or the Root Port BFM, that are not caused by the Endpoint Application Layer being tested.	N/A	Y Cannot suppress	Y Cannot suppress	FATAL:

2.4.5.11.1. ebfm_display Verilog HDL Function

The `ebfm_display` procedure or function displays a message of the specified type to the simulation standard output and also the log file if `ebfm_log_open` is called.

A message can be suppressed, simulation can be stopped or both based on the default settings of the message type and the value of the bit mask when each of the procedures listed below is called. You can call one or both of these procedures based on what messages you want displayed and whether or not you want simulation to stop for specific messages.

- When `ebfm_log_set_suppressed_msg_mask` is called, the display of the message might be suppressed based on the value of the bit mask.
- When `ebfm_log_set_stop_on_msg_mask` is called, the simulation can be stopped after the message is displayed, based on the value of the bit mask.

Location	altrpcietb_g3bfm_log.v	
Syntax	`Verilog HDL: dummy_return:=ebfm_display(msg_type, message);`	
Argument	`msg_type`	Message type for the message. Should be one of the constants defined in Constants: Verilog HDL Type INTEGER.
	`message`	The message string is limited to a maximum of 100 characters. Also, because Verilog HDL does not allow variable length strings, this routine strips off leading characters of 8'h00 before displaying the message.
Return	`always 0`	Applies only to the Verilog HDL routine.

2.4.5.11.2. ebfm_log_stop_sim Verilog HDL Function

The `ebfm_log_stop_sim` procedure stops the simulation.

Location	altrpcietb_bfm_log.v	
Syntax	Verilog HDL: `return:=ebfm_log_stop_sim(success);`	
Argument	`success`	When set to a 1, this process stops the simulation with a message indicating successful completion. The message is prefixed with SUCCESS. Otherwise, this process stops the simulation with a message indicating unsuccessful completion. The message is prefixed with FAILURE.
Return	Always 0	This value applies only to the Verilog HDL function.

2.4.5.11.3. ebfm_log_set_suppressed_msg_mask Task

The `ebfm_log_set_suppressed_msg_mask` procedure controls which message types are suppressed.

Location	altrpcietb_bfm_log.v	
Syntax	`ebfm_log_set_suppressed_msg_mask (msg_mask)`	
Argument	`msg_mask`	This argument is `reg [EBFM_MSG_ERROR_CONTINUE: EBFM_MSG_DEBUG]`. A 1 in a specific bit position of the `msg_mask` causes messages of the type corresponding to the bit position to be suppressed.

2.4.5.11.4. ebfm_log_set_stop_on_msg_mask Verilog HDL Task

The `ebfm_log_set_stop_on_msg_mask` procedure controls which message types stop simulation. This procedure alters the default behavior of the simulation when errors occur as described in the *BFM Log and Message Procedures*.

Location	altrpcietb_bfm_log.v	
Syntax	`ebfm_log_set_stop_on_msg_mask (msg_mask)`	
Argument	`msg_mask`	This argument is `reg [EBFM_MSG_ERROR_CONTINUE:EBFM_MSG_DEBUG]`. A 1 in a specific bit position of the `msg_mask` causes messages of the type corresponding to the bit position to stop the simulation after the message is displayed.

2.4.5.11.5. ebfm_log_open Verilog HDL Function

The `ebfm_log_open` procedure opens a log file of the specified name. All displayed messages are called by `ebfm_display` and are written to this log file as simulator standard output.

Location	altrpcietb_bfm_log.v	
Syntax	`ebfm_log_open (fn)`	
Argument	`fn`	This argument is type `string` and provides the file name of log file to be opened.

2.4.5.11.6. ebfm_log_close Verilog HDL Function

The `ebfm_log_close` procedure closes the log file opened by a previous call to `ebfm_log_open`.

Location	altrpcietb_bfm_log.v
Syntax	`ebfm_log_close`
Argument	NONE

2.4.5.12. Verilog HDL Formatting Functions

The Verilog HDL Formatting procedures and functions are available in the`altpcietb_bfm_log.v` . The formatting functions are only used by Verilog HDL. All these functions take one argument of a specified length and return a vector of a specified length.

2.4.5.12.1. himage1

This function creates a one-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= himage(vec)`	
Argument	`vec`	Input data type `reg` with a `range` of 3:0.
Return range	`string`	Returns a 1-digit hexadecimal representation of the input argument. Return data is type `reg` with a `range` of 8:1

2.4.5.12.2. himage2

This function creates a two-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= himage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 7:0.
Return range	`string`	Returns a 2-digit hexadecimal presentation of the input argument, padded with leading 0s, if they are needed. Return data is type `reg` with a `range` of 16:1

2.4.5.12.3. himage4

This function creates a four-digit hexadecimal string representation of the input argument can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= himage(vec)`	
Argument range	vec	Input data type `reg` with a `range` of 15:0.
Return range	Returns a four-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type `reg` with a `range` of 32:1.	

2.4.5.12.4. himage8

This function creates an 8-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= himage(vec)`	
Argument range	vec	Input data type reg with a range of 31:0.
Return range	`string`	Returns an 8-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type `reg` with a `range` of 64:1.

2.4.5.12.5. himage16

This function creates a 16-digit hexadecimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= himage(vec)`	
Argument range	vec	Input data type reg with a range of 63:0.
Return range	`string`	Returns a 16-digit hexadecimal representation of the input argument, padded with leading 0s, if they are needed. Return data is type `reg` with a `range` of 128:1.

2.4.5.12.6. dimage1

This function creates a one-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	vec	Input data type `reg` with a `range` of 31:0.
Return range	`string`	Returns a 1-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 8:1. Returns the letter *U* if the value cannot be represented.

2.4.5.12.7. dimage2

This function creates a two-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 31:0.
Return range	`string`	Returns a 2-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 16:1. Returns the letter *U* if the value cannot be represented.

2.4.5.12.8. dimage3

This function creates a three-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 31:0.
Return range	string	Returns a 3-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 24:1. Returns the letter *U* if the value cannot be represented.

2.4.5.12.9. dimage4

This function creates a four-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 31:0.
Return range	`string`	Returns a 4-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 32:1. Returns the letter *U* if the value cannot be represented.

2.4.5.12.10. dimage5

This function creates a five-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of 31:0.
Return range	`string`	Returns a 5-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of 40:1. Returns the letter *U* if the value cannot be represented.

2.4.5.12.11. dimage6

This function creates a six-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of `31:0`.
Return range	`string`	Returns a 6-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of `48:1`. Returns the letter *U* if the value cannot be represented.

2.4.5.12.12. dimage7

This function creates a seven-digit decimal string representation of the input argument that can be concatenated into a larger message string and passed to `ebfm_display`.

Location	altpcietb_bfm_log.v	
Syntax	`string:= dimage(vec)`	
Argument range	`vec`	Input data type `reg` with a `range` of `31:0`.
Return range	`string`	Returns a 7-digit decimal representation of the input argument that is padded with leading 0s if necessary. Return data is type `reg` with a `range` of `56:1`. Returns the letter <*U*> if the value cannot be represented.

2.5. Compiling the Design Example

Note: The R-Tile Avalon Streaming Intel FGPA IP for PCIe design example has limited hardware testing support on the 23.2 release of Intel Quartus Prime. The instructions below can be used for early testing and for the flow required to run the design example on the Intel Agilex 7 I-Series FPGA Development Kit.

1. Navigate to `<project_dir>/intel_rtile_pcie_ast_0_example_design/` and open `pcie_ed.qpf`.
2. If you select a specific development kit when generating the design example, the VID-related settings are included in the .qsf file and you are not required to add them manually. Note that these settings are board-specific.
3. On the Processing menu, select **Start Compilation**.

2.6. Installing the Linux Kernel Driver

Note: The R-Tile Avalon Streaming Intel FGPA IP for PCIe design example has limited hardware testing support on the 23.2 release of Intel Quartus Prime. The instructions below can be used for early testing and for the flow required to run the design example on the Intel Agilex 7 I-Series FPGA Development Kit.

Before you can test the design example in hardware, you must install the Linux kernel driver. You can use this driver to perform the following tests:

- A PCIe link test that performs 100 writes and reads
- Memory space DWORD(5) reads and writes
- Configuration Space DWORD reads and writes

In addition, you can use the driver to change the value of the following parameters:

- The BAR being used
- The selected device (by specifying the bus, device and function (BDF) numbers for the device)

Complete the following steps to install the kernel driver:

1. Navigate to `./software/kernel/linux` under the example design generation directory.
2. Change the permissions on the `install`, `load`, and `unload` files:

   ```
   $ chmod 777 install load unload
   ```
3. Install the driver:

   ```
   $ sudo ./install
   ```
4. Verify the driver installation:

   ```
   $ lsmod | grep intel_fpga_pcie_drv
   ```

   Expected result:

   ```
   intel_fpga_pcie_drv 17792 0
   ```

   Note: The 17792 number from the outcome above may be different on each platform.
5. Verify that Linux recognizes the PCIe design example:

   ```
   $ lspci -d 1172:000 -v | grep intel_fpga_pcie_drv
   ```

   Note: If you have changed the Vendor ID, substitute the new Vendor ID for Intel's Vendor ID in this command (1172).

   Expected result:

   ```
   Kernel driver in use: intel_fpga_pcie_drv
   ```

2.7. Running the Design Example

Here are the test operations you can perform on the R-tile Avalon-ST PCIe design examples:

Table 10. Test Operations Supported by the R-tile Avalon-ST IP for PCIe Design Examples

Operations	Required BAR	Supported by R-tile Avalon-ST IP for PCIe Design Examples		
		PIO	SR-IOV	Performance
0: Link test - 100 writes and reads	0	Yes	Yes	No
1: Write memory space	0	Yes	Yes	No
2: Read memory space	0	Yes	Yes	No

continued...

(5) Throughout this user guide, the terms word, DWORD and QWORD have the same meaning that they have in the PCI Express Base Specification. A word is 16 bits, a DWORD is 32 bits, and a QWORD is 64 bits.

Operations	Required BAR	Supported by R-tile Avalon-ST IP for PCIe Design Examples		
		PIO	SR-IOV	Performance
3: Write configuration space	N/A	Yes	No	No
4: Read configuration space	N/A	Yes	No	No
5: Change BAR	N/A	Yes	Yes	No
6: Change device	N/A	Yes	Yes	No
7: Enable SR-IOV	N/A	No	Yes	No
8: Do a link test for every enabled virtual function belonging to the current device	N/A	No	Yes	No
9: Perform DMA for Performance test	N/A	No	No	Yes

Note: When using the Intel Agilex 7 I-Series FPGA Development Kit, set the PCIe REFCLK Select switch to select the clock from the PCIe Connector. For more details, refer to the Intel Agilex 7 I-Series FPGA Development Kit User Guide.

2.7.1. Running the PIO Design Example

1. Navigate to `./software/user/example` under the design example directory.
2. Compile the design example application:

   ```
   $ make
   ```

3. Run the test:

   ```
   $ sudo ./intel_fpga_pcie_link_test
   ```

   You can run the Intel FPGA IP PCIe link test in manual or automatic mode. Choose from:

   - In automatic mode, the application automatically selects the device. The test selects the Intel PCIe device with the lowest BDF by matching the Vendor ID. The test also selects the lowest available BAR.
   - In manual mode, the test queries you for the bus, device, and function number and BAR.

   For the Intel Agilex 7 Development Kit, you can determine the BDF by typing the following command:

   ```
   $ lspci -d 1172:
   ```

4. Here are sample transcripts for automatic and manual modes:

   Automatic mode:

```
Intel FPGA PCIe Link Test - Automatic Mode
Version 2.0
0: Automatically select a device
1: Manually select a device
**************************************************
>0
Opened a handle to BAR 0 of a device with BDF 0x3800
**************************************************
0: Link test - 100 writes and reads
1: Write memory space
2: Read memory space
```

```
3: Write configuration space
4: Read configuration space
5: Change BAR for PIO
6: Change device
7: Quit program
************************************************
> 0
Doing 100 writes and 100 reads . .
Number of write errors:     0
Number of read errors:      0
Number of DWORD mismatches: 0
```

Manual mode:

```
Intel FPGA PCIe Link Test
Version 2.0
0: Automatically select a device
1: Manually select a device
************************************************
> 1
Enter bus number, in hex:
> 4b
Enter device number, in hex:
> 0
BDF is 0x4b00
B:D.F, in hex, is 4b:0.0
Enter BAR number (-1 for none):
> 0
Opened a handle to BAR 0 of a device with BDF 0x4b00
```

2.7.2. Running the SR-IOV Design Example

Follow the steps below to test the SR-IOV design example on hardware:

1. Navigate to `./software/user/example` under the design example directory.
2. Compile the design example application by running the command:

   `make`

   Then perform the following steps:

   a. Run the link test application by running the command:

      `sudo ./intel_fpga_pcie_link_test`

   b. Select Option 1: **Manually Select the Device**

   c. Enter the BDF of the Physical Function for which the virtual functions are allocated.

   d. Enter **BAR 0** to proceed to the test menu.

```
************************************************************
Intel FPGA PCIe Link Test
Version 2.0
0: Automatically select a device
1: Manually select a device
************************************************************
> 1
Enter bus number, in hex:
> 38
Enter device number, in hex:
> 0
Enter function number, in hex:
> 0
BDF is 0x3800
B:D.F, in hex, is 38:0.0
Enter BAR number (-1 for none):
```

```
> 0
Opened a handle to BAR 0 of a device with BDF 0x3800

*******************************************************
0: Link test - 100 writes and reads
 1: Write memory space
 2: Read memory space
 3: Write configuration space
 4: Read configuration space
 5: Change BAR for PIO
 6: Change device
 7: Enable SRIOV
 8: Do a link test for every enabled virtual function
    belonging to the current device
 9: Quit program
*******************************************************
>
```

e. Enter **Option 7** to enable SR-IOV for the current device.

f. Enter **16** as the number of virtual functions to be enabled for the current device.

```
*******************************************************
 0: Link test - 100 writes and reads
 1: Write memory space
 2: Read memory space
 3: Write configuration space
 4: Read configuration space
 5: Change BAR for PIO
 6: Change device
 7: Enable SRIOV
 8: Do a link test for every enabled virtual function
    belonging to the current device
 9: Quit program
*******************************************************
> 7
Enter the number of VFs to enable for the current device (0-31):
> 16
Enabled 16 VFs.
Type 'lspci -d 1172:' in a new terminal to determine newly enabled
devices' BDFs.
[root@localhost ~]# lspci -d 1172:
38:00.0 Unassigned class [ff00]: Altera Corporation Device 0000 (rev 01)
38:00.2 Unassigned class [ff00]: Altera Corporation Device 0000
38:00.3 Unassigned class [ff00]: Altera Corporation Device 0000
38:00.4 Unassigned class [ff00]: Altera Corporation Device 0000
38:00.5 Unassigned class [ff00]: Altera Corporation Device 0000
38:00.6 Unassigned class [ff00]: Altera Corporation Device 0000
38:00.7 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.0 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.1 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.2 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.3 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.4 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.5 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.6 Unassigned class [ff00]: Altera Corporation Device 0000
38:01.7 Unassigned class [ff00]: Altera Corporation Device 0000
38:02.0 Unassigned class [ff00]: Altera Corporation Device 0000
38:02.1 Unassigned class [ff00]: Altera Corporation Device 0000
```

g. Enter **Option 8** to perform a link test for every enabled virtual function allocated within the physical function. The link test application performs 100 memory writes with a single dword of data each and then reads back the data. At the end of the testing, the application prints the number of virtual functions that failed the test.

```
Testing VF with BDF 0x380d...
Doing 100 writes and 100 reads..
Number of write errors:       0
Number of read errors:        0
Number of dword mismatches:   0
Testing VF with BDF 0x380e...
Doing 100 writes and 100 reads..
Number of write errors:       0
Number of read errors:        0
Number of dword mismatches:   0
Testing VF with BDF 0x380f...
Doing 100 writes and 100 reads..
Number of write errors:       0
Number of read errors:        0
Number of dword mismatches:   0
Testing VF with BDF 0x3810...
Doing 100 writes and 100 reads..
Number of write errors:       0
Number of read errors:        0
Number of dword mismatches:   0
Testing VF with BDF 0x3811...
Doing 100 writes and 100 reads..
Number of write errors:       0
Number of read errors:        0
Number of dword mismatches:   0
Test failed for 0 VFs out of 16 VFs
```

h. Exit the application and run the command

```
lspci -d 1172: | grep -c Altera
```

to verify the enumeration of PFs and VFs. The expected result is the sum of PFs and VFs.

In case you enable the 16 VFs for only one of the physical functions, you would have 18 devices reported, 2 PFs + 16 VFs of PF0. You can also repeat the same process for the second function included in the design example and you would have a total of 34 devices reported, 2 PFs + 32 VFs.

```
[root@localhost ~]# lspci -d 1172: | grep -c Altera
```

2.7.3. Running the Performance Design Example

1. Navigate to `./software/user/example` under the design example directory.
2. Compile the design example application:

   ```
   $ make
   ```

3. Run the test:

   ```
   $ sudo ./intel_fpga_pcie_link_test
   ```

   You can run the Intel FPGA IP PCIe link test in manual or automatic mode. Choose from:

- In automatic mode, the application automatically selects the device. The test selects the Intel PCIe device with the lowest BDF by matching the Vendor ID. The test also selects the lowest available BAR.
- In manual mode, the test queries you for the bus, device, and function numbers and BAR.

For the Intel Agilex 7 Development Kit, you can determine the BDF by typing the following command:

```
$ lspci -d 1172:
```

4. Here is a sample transcript for the selection between automatic and manual modes:

```
# ./intel_fpga_pcie_link_test
*******************************************************
Intel FPGA PCIe Link Test
Version 2.0
0: Automatically select a device
1: Manually select a device
*******************************************************
```

5. The Performance design example for the Intel FPGA IP R-Tile Avalon Streaming Hard IP for PCI Express IP core only supports menu option 9. Enter **9** and press **Enter** to proceed.

```
Opened a handle to BAR 0 of a device with BDF 0x3800
*******************************************************
 0: Link test - 100 writes and reads
 1: Write memory space
 2: Read memory space
 3: Write configuration space
 4: Read configuration space
 5: Change BAR for PIO
 6: Change device
 7: Enable SRIOV
 8: Do a link test for every enabled virtual function
    belonging to the current device
 9: Perform DMA for Throughput
10: Quit program
*******************************************************
> 9
```

6. Select option **0: Execute Max Traffic Test**

```
*******************************************************
   END POINT (EP) ORIGINATED TRAFFIC
 0: Execute Max Traffic Test
 1: Quit
*******************************************************
> 0
*******************************************************
  MAX END POINT (EP) TRAFFIC
*******************************************************

   EXECUTING WRITE TRAFFIC...
100% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
   PERF WRITE GB/s: 59.94

   EXECUTING READ TRAFFIC...
100% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
   PERF READ GB/s: 59.1

   EXECUTING WRITE & READ TRAFFIC...
100% [||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
```

```
   PERF WRITE GB/s: 55.51
   PERF READ GB/s: 56.87
*******************************************************
```

3. R-Tile Avalon Streaming Intel FPGA IP for PCI Express Design Example User Guide Archives

For the latest and previous versions of these user guides, refer to the R-Tile Avalon Streaming Intel FPGA IP for PCI Express Design Example User Guide. If an IP or software version is not listed, the user guide for the previous IP or software version applies.

4. Document Revision History for the R-Tile Avalon Streaming Intel FPGA IP for PCI Express Design Example User Guide

Document Version	Intel Quartus Prime Version	IP Version	Changes
2023.06.26	23.2	10.0.0	Changed the Intel Quartus Prime version number and IP version number.
2023.04.03	23.1	9.0.0	• Updated product family name to "Intel Agilex 7" • Added a new section *Functional Description for the Performance Design Example* • Added a new section *Running the Performance Design Example*
2022.12.19	22.4	8.0.0	Updated the ACDS version number and the IP version number.
2022.09.26	22.3	7.0.0	• Added the section *Functional Description for the Single Root I/O Virtualization (SR-IOV) Design Example*. • Added steps to generate the SR-IOV design example to the section *Generating the Design Example*. • Added the section *SR-IOV Design Example Testbench*. • Added the section *Running the SR-IOV Design Example*.
2022.06.20	22.2	6.0.0	Added a section on how to simulate the design example using the Xcelium simulator.
2022.03.28	22.1	5.0.0	• Added a summary table of all the configurations supported by this design example to the *Design Example Description* section. • Added the *Hardware and Software Requirements* section. • Updated the *Simulating the Design Example* section to include clearer instructions on how to simulate the design example using various simulators.
2021.12.13	21.4	4.0.0	Added support for the Gen4 x16, Gen3 x16, Gen4 2x8 and Gen3 2x8 design examples.
2021.10.04	21.3	3.0.0	Added instructions on how to simulate the design example using the VCS MX simulator to the section *Simulating the Design Example*.
2021.07.12	21.2	2.0.0	Initial release.