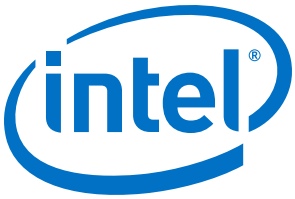


Intel® Stratix® 10 H-Tile/L-Tile Avalon® Memory Mapped (Avalon-MM) Hard IP for PCI Express* User Guide

Updated for Intel® Quartus® Prime Design Suite: **19.3**

Subscribe
Send Feedback

UG-20033 | 2020.01.03
Latest document on the web: **PDF** | **HTML**

Contents

1. Introduction

This User Guide is applicable to the H-Tile and L-Tile variants of the Intel® Stratix® 10 devices.

1.1. Avalon-MM Interface for PCIe

Intel Stratix 10 FPGAs include a configurable, hardened protocol stack for PCI Express* that is compliant with *PCI Express Base Specification 3.0*. This IP core combines the functionality of previous Avalon® Memory-Mapped (Avalon-MM) and Avalon-MM direct memory access (DMA) interfaces. It supports the same functionality for Intel Stratix 10 as the Avalon-MM and Avalon-MM with DMA variants for Arria® 10 devices.

The Hard IP for PCI Express IP core using the Avalon-MM interface removes many of the complexities associated with the PCIe protocol. It handles all of the Transaction Layer Packet (TLP) encoding and decoding, simplifying the design task. This IP core also includes optional Read and Write Data Mover modules facilitating the creation of high-performance DMA designs. Both the Avalon-MM interface and the Read and Write Data Mover modules are implemented in soft logic.

The Avalon-MM Intel Stratix 10 Hard IP for PCI Express IP Core supports Gen1, Gen2 and Gen3 data rates and x1, x2, x4, and x8 configurations. Gen1 and Gen2 data rates are also supported with the x16 configuration.

Note: The Gen3 x16 configuration is supported by another IP core, the Avalon-MM Intel Stratix 10 Hard IP+ core. For details, refer to the *Avalon-MM Intel Stratix 10 Hard IP+ for PCIe* Solutions User Guide*.

Figure 1. Intel Stratix 10 PCIe IP Core Variant with Avalon-MM Interface

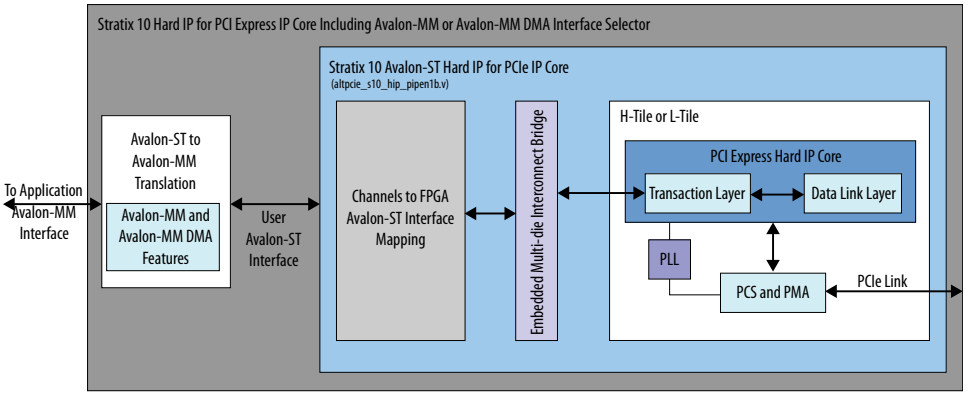


Table 1. PCI Express Data Throughput

The following table shows the theoretical link bandwidth of a PCI Express link for Gen1, Gen2, and Gen3 for 1, 2, 4, 8, and 16 lanes excluding overhead. This table provides bandwidths for a single transmit (TX) or receive (RX) channel. The numbers double for duplex operation. The protocol specifies 2.5 giga-transfers per second (GT/s) for Gen1, 5.0 GT/s for Gen2, and 8.0 GT/s for Gen3. Gen1 and Gen2 use 8B/10B encoding which introduces a 20% overhead. Gen3 uses 128b/130b encoding which an overhead of 1.54%. The following table shows the actual usable data bandwidth in gigabytes per second (GBps). The encoding and decoding overhead has been removed.

	Link Width				
	×1	×2	×4	×8	×16
PCI Express Gen1 (2.5 Gbps)	2	4	8	16	32
PCI Express Gen2 (5.0 Gbps)	4	8	16	32	64
PCI Express Gen3 (8.0 Gbps)	7.87	15.75	31.5	63	Not available in current release

Related Information

- Creating a System with Platform Designer
- PCI Express Base Specification 3.0
- Avalon-MM Intel Stratix 10 Hard IP+ for PCIe* Solutions User Guide

1.2. Features

New features in the Intel Quartus® Prime Pro Edition Software:

- Support for Programmer Object File (`*.pof`) generation for up to Gen3 x8 variants.
- Support for a PCIe* Link Inspector including the following features:
  - Read and write access to the Configuration Space registers.
  - LTSSM monitoring.
  - PLL lock and calibration status monitoring.
  - Read and write access to PCS and PMA registers.
- Software application for Linux demonstrating PCIe accesses in hardware with dynamically generated design examples
- Support for instantiation as a stand-alone IP core from the Intel Quartus Prime Pro Edition IP Catalog, as well as Platform Designer instantiation.

The Avalon-MM Stratix 10 Hard IP for PCI Express IP Core supports the following features:

- A migration path for Avalon-MM or Avalon-MM DMA implemented in earlier device families.
- Standard Avalon-MM master and slave interfaces:
  - High throughput bursting Avalon-MM slave with optional address mapping.
  - Avalon-MM slave with byte granularity enable support for single DWORD ports and DWORD granularity enable support for high throughput ports.
  - Up to 6 Avalon-MM masters associated to 1 or more BARs with byte enable support.
  - High performance, bursting Avalon-MM master ports.
- Optional DMA data mover with high throughput, bursting, Avalon-MM master:
  - Write Data Mover moves data to PCIe system memory using PCIe Memory Write (MemWr) Transaction Layer Packets (TLPs).
  - Read Data Mover moves data to local memory using PCIe Memory Read (MemRd) TLPs.
- Modular implementation to select the required features for a specific application:
  - Simultaneous support for DMA modules and high throughput Avalon-MM slaves and masters.
  - Avalon-MM slave to easily access the entire PCIe address space without requiring any PCI Express specific knowledge.
- Support for 256-bit and 64-bit application interface widths.
- Advanced Error Reporting (AER): In Intel Stratix 10 devices, Advanced Error Reporting is always enabled in the PCIe Hard IP for both the L and H transceiver tiles.
- Available in both Intel Quartus Prime Pro Edition and Platform Designer IP Catalogs.
- Optional internal DMA Descriptor controller.
- Autonomous Hard IP mode, allowing the PCIe IP core to begin operation before the FPGA fabric is programmed. This mode is enabled by default. It cannot be disabled.
- Operates at up to 250 MHz in -2 speed grade device.

Note: For a detailed understanding of the PCIe protocol, please refer to the *PCI Express Base Specification.*

1.3. Release Information

Table 2. Hard IP for PCI Express Release Information

Item	Description
Version	Intel Quartus Prime Pro Edition 18.0 Software
Release Date	May 2018
Ordering Codes	No ordering code is required

Intel verifies that the current version of the Intel Quartus Prime Pro Edition software compiles the previous version of each IP core, if this IP core was included in the previous release. Intel reports any exceptions to this verification in the *Intel IP Release Notes* or clarifies them in the Intel Quartus Prime Pro Edition IP Update tool. Intel does not verify compilation with IP core versions older than the previous release.

Related Information

Timing and Power Models
Reports the default device support levels in the current version of the Quartus Prime Pro Edition software.

1.4. Device Family Support

The following terms define device support levels for Intel FPGA IP cores:

- **Advance support**—the IP core is available for simulation and compilation for this device family. Timing models include initial engineering estimates of delays based on early post-layout information. The timing models are subject to change as silicon testing improves the correlation between the actual silicon and the timing models. You can use this IP core for system architecture and resource utilization studies, simulation, pinout, system latency assessments, basic timing assessments (pipeline budgeting), and I/O transfer strategy (data-path width, burst depth, I/O standards tradeoffs).
- **Preliminary support**—the IP core is verified with preliminary timing models for this device family. The IP core meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.
- **Final support**—the IP core is verified with final timing models for this device family. The IP core meets all functional and timing requirements for the device family and can be used in production designs.

Table 3. Device Family Support

Device Family	Support Level
Intel Stratix 10	Preliminary support.
Other device families	No support. Refer to the *Intel PCI Express Solutions* web page on the Intel website for support information on other device families.

Related Information

PCI Express Solutions Web Page

1.5. Recommended Speed Grades

Table 4. Intel Stratix 10 Recommended Speed Grades for All Avalon-MM with DMA Widths and Frequencies

The recommended speed grades are for production parts.

Lane Rate	Link Width	Interface Width	Application Clock Frequency (MHz)	Recommended Speed Grades
Gen1	x1, x2, x4, x8, x16	256 Bits	125	–1, –2
Gen2	x1, x2, x4, x8,	256 bits	125	–1, –2
	x16	256 bits	250	–1, –2
Gen3	x1, x2, x4	256 bits	125	–1, –2
	x8	256 bits	250	–1, –2

1.6. Performance and Resource Utilization

The Avalon-MM Intel Stratix 10 variants include an Avalon-MM DMA bridge implemented in soft logic. It operates as a front end to the hardened protocol stack. The resource utilization table below shows results for the Gen1 x1 and Gen3 x8 Simple DMA dynamically generated design examples.

The results are for the current version of the Intel Quartus Prime Pro Edition software. With the exception of M20K memory blocks, the numbers are rounded up to the nearest 50.

Table 5. Resource Utilization Avalon-MM Intel Stratix 10 Hard IP for PCI Express IP Core

Variant	Typical ALMs	M20K Memory Blocks[1]	Logic Registers
Gen1 x1	3,018	64	4,690
Gen3 x8	15,976	69	32,393

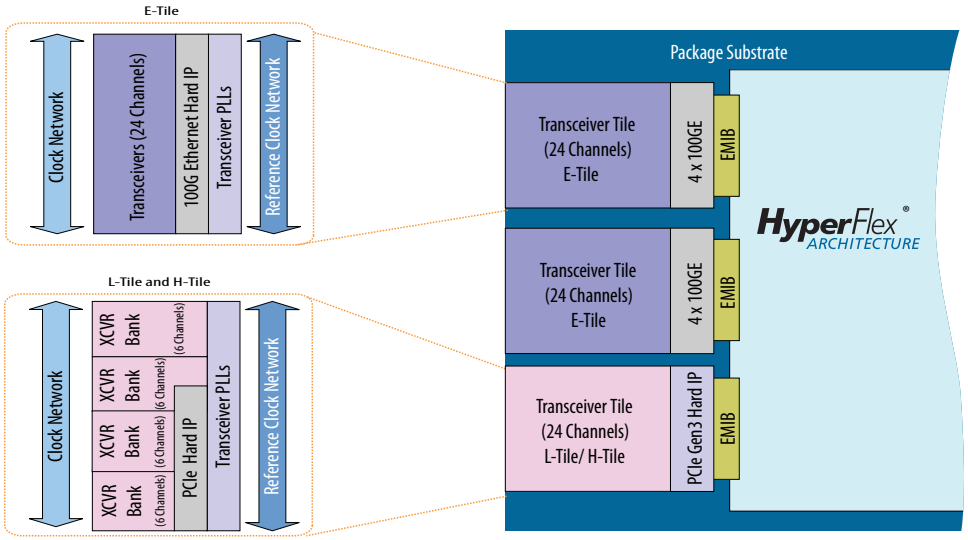
Related Information

Running the Fitter
For information on Fitter constraints.

1.7. Transceiver Tiles

Intel Stratix 10 introduces several transceiver tile variants to support a wide variety of protocols.

Figure 2. Intel Stratix 10 Transceiver Tile Block Diagram



(1) These results include the logic necessary to implement the 2, On-Chip Memories and the PCIe DMA 256-bit Controller which are included in the designs.

Table 6. Transceiver Tiles Channel Types

Tile	Device Type	Channel Capability		Channel Hard IP Access
		Chip-to-Chip	Backplane	
L-Tile	GX	26 Gbps (NRZ)	12.5 Gbps (NRZ)	PCIe Gen3x16
H-Tile	GX	28.3 Gbps (NRZ)	28.3 Gbps (NRZ)	PCIe Gen3x16
E-Tile	GXE	30 Gbps (NRZ), 56 Gbps (PAM-4)	30 Gbps (NRZ), 56 Gbps (PAM-4)	100G Ethernet

L-Tile and H-Tile

Both L and H transceiver tiles contain four transceiver banks-with a total of 24 duplex channels, eight ATX PLLs, eight fPLLs, eight CMU PLLs, a PCIe Hard IP block, and associated input reference and transmitter clock networks. L and H transceiver tiles also include 10GBASE-KR/40GBASE-KR4 FEC block in each channel.

L-Tiles have transceiver channels that support up to 26 Gbps chip-to-chip or 12.5 Gbps backplane applications. H-Tiles have transceiver channels to support 28 Gbps applications. H-Tile channels support fast lock-time for Gigabit-capable passive optical network (GPON).

Intel Stratix 10 GX/SX devices incorporate L-Tiles or H-Tiles. Package migration is available with Intel Stratix 10 GX/SX from L-Tile to H-Tile variants.

E-Tile

E-Tiles are designed to support 56 Gbps with PAM-4 signaling or up to 30 Gbps backplane with NRZ signaling. E-Tiles do not include any PCIe Hard IP blocks.

1.8. PCI Express IP Core Package Layout

Intel Stratix 10 devices have high-speed transceivers implemented on separate transceiver tiles. The transceiver tiles are on the left and right sides of the device.

Each 24-channel transceiver L- or H- tile includes one x16 PCIe IP Core implemented in hardened logic. The following figures show the layout of PCIe IP cores in Intel Stratix 10 devices. Both L- and H-tiles are orange. E-tiles are green.

Figure 3. Intel Stratix 10 GX/SX Devices with 4 PCIe Hard IP Cores and 96 Transceiver Channels

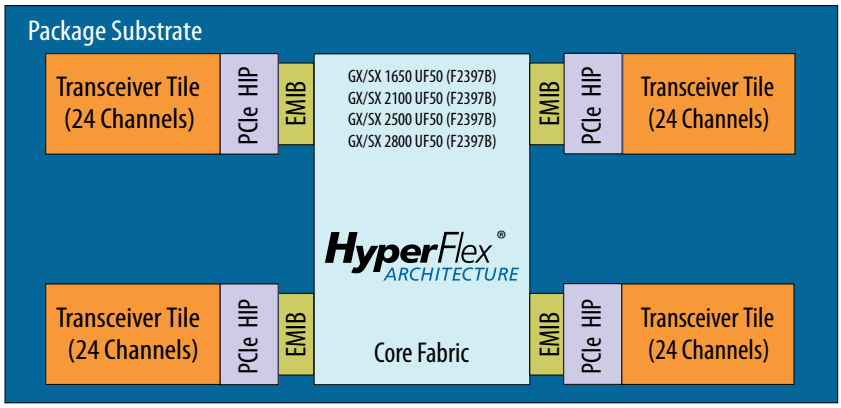


Figure 4. Intel Stratix 10 GX/SX Devices with 2 PCIe Hard IP Cores and 48 Transceiver Channels

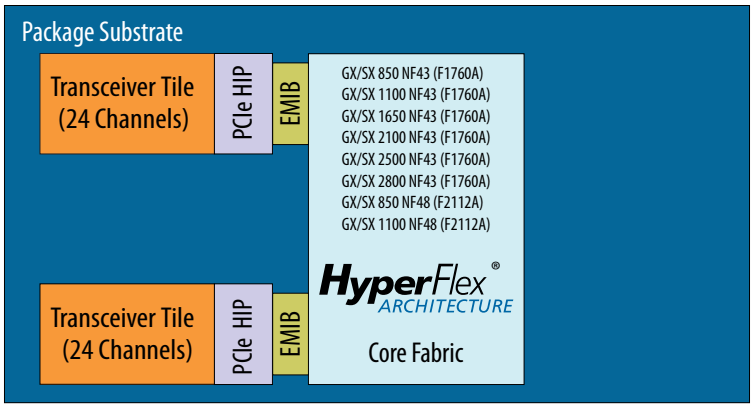


Figure 5. Intel Stratix 10 GX/SX Devices with 2 PCIe Hard IP Cores and 48 Transceiver Channels - Transceivers on Both Sides

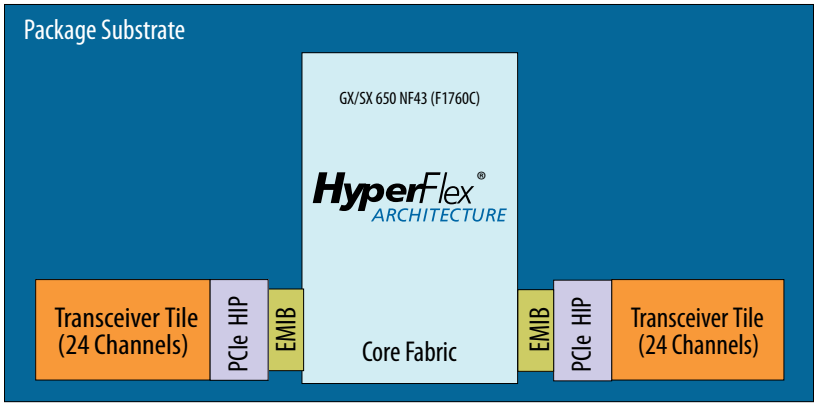
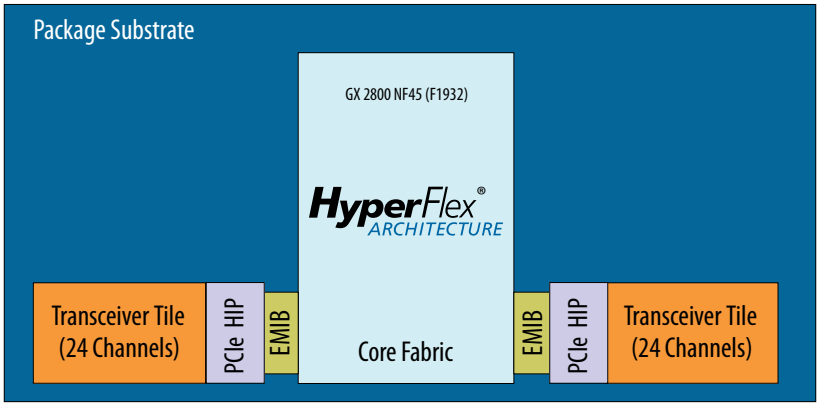


Figure 6. Intel Stratix 10 Migration Device with 2 Transceiver Tiles and 48 Transceiver Channels



Note:

1. Intel Stratix 10 migration device contains 2 L-Tiles which match Intel Arria 10 migration device.

Figure 7. Intel Stratix 10 GX/SX Devices with 1 PCIe Hard IP Core and 24 Transceiver Channels

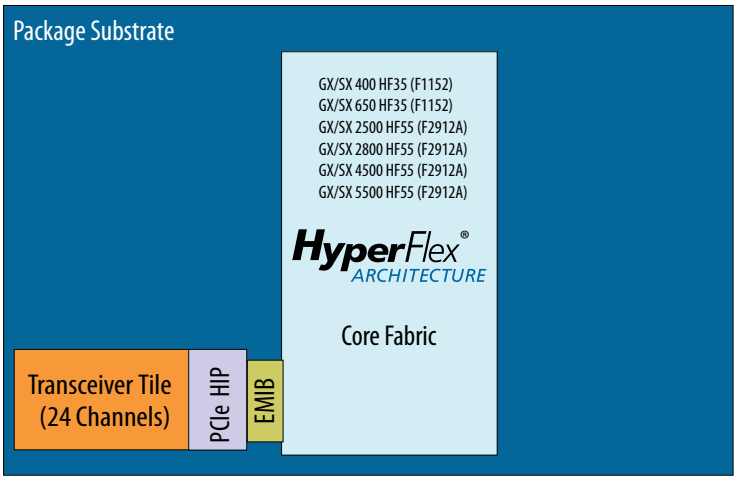
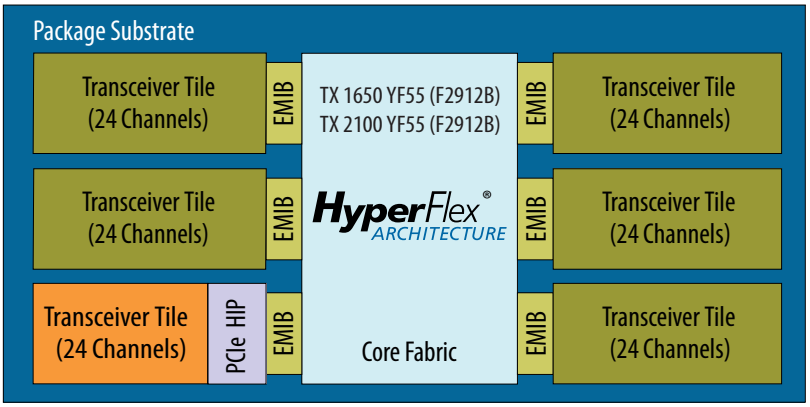


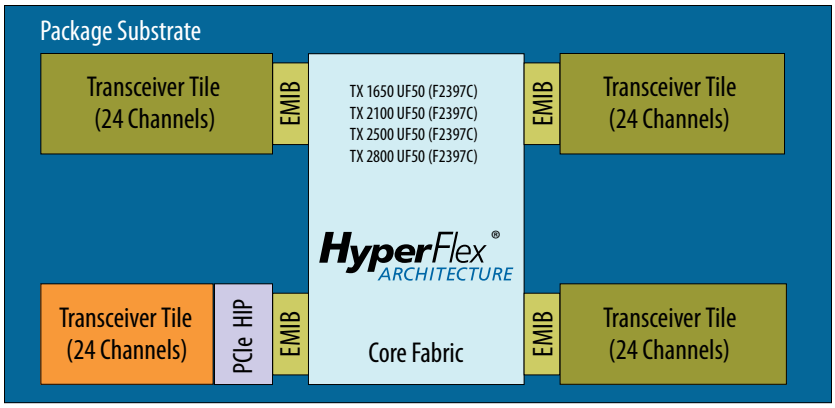
Figure 8. Intel Stratix 10 TX Devices with 1 PCIe Hard IP Core and 144 Transceiver Channels



Note:

1. Intel Stratix 10 TX Devices use a combination of E-Tiles and H-Tiles.
2. Five E-Tiles support 57.8G PAM-4 and 28.9G NRZ backplanes.
3. One H-Tile supports up to 28.3G backplanes and PCIe up to Gen3 x16.

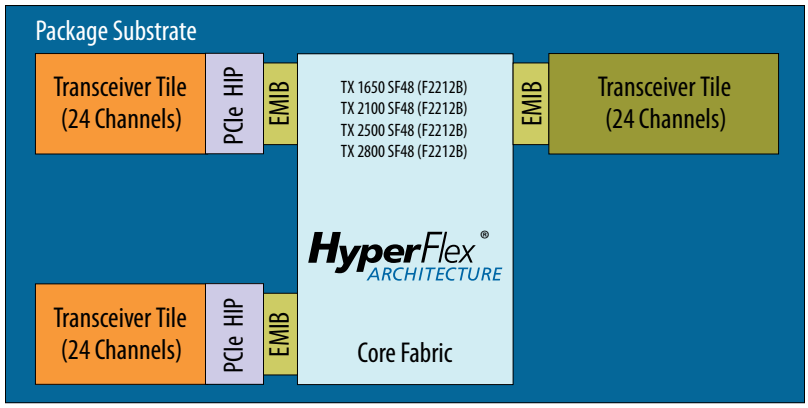
Figure 9. Intel Stratix 10 TX Devices with 1 PCIe Hard IP Core and 96 Transceiver Channels



Note:

1. Intel Stratix 10 TX Devices use a combination of E-Tiles and H-Tiles.
2. Three E-Tiles support 57.8G PAM-4 and 28.9G NRZ backplanes.
3. One H-Tile supports up to 28.3G backplanes PCIe up to Gen3 x16..

Figure 10. Intel Stratix 10 TX Devices with 2 PCIe Hard IP Cores and 72 Transceiver Channels



Note:

1. Intel Stratix 10 TX Devices use a combination of E-Tiles and H-Tiles.
2. One E-Tile support 57.8G PAM-4 and 28.9G NRZ backplanes.
3. Two H-Tiles supports up to 28.3G backplanes PCIe up to Gen3 x16..

Related Information

Stratix 10 GX/SX Device Overview
For more information about Stratix 10 devices.

1.9. Channel Availability

PCIe Hard IP Channel Restrictions

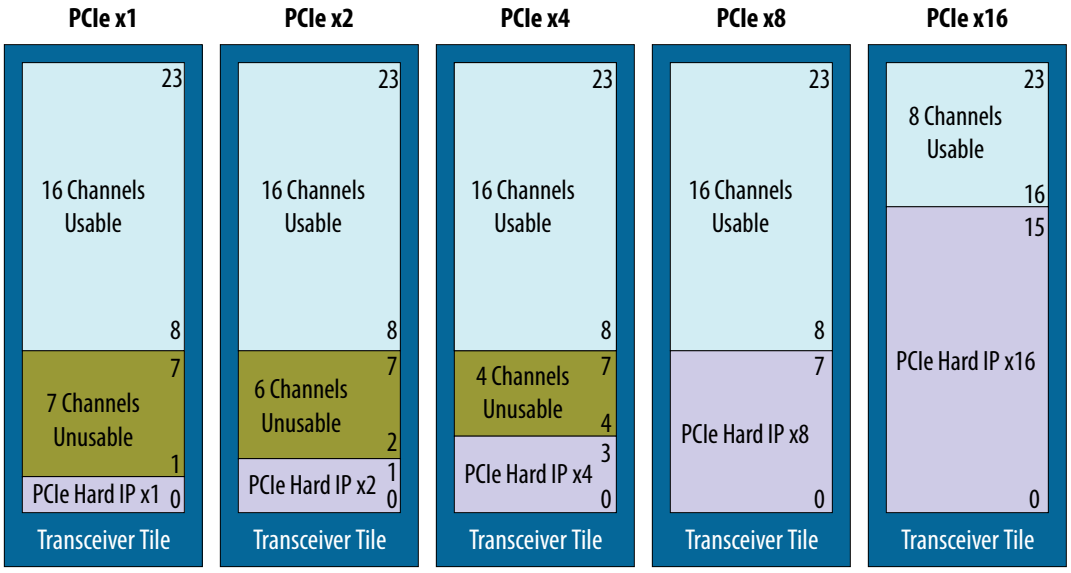
Each L- or H-Tile transceiver tile contains one PCIe Hard IP block. The following table and figure show the possible PCIe Hard IP channel configurations, the number of unusable channels, and the number of channels available for other protocols. For example, a PCIe x4 variant uses 4 channels and 4 additional channels are unusable.

Table 7. Unusable Channels

PCIe Hard IP Configuration	Number of Unusable Channels	Usable Channels
PCIe x1	7	16
PCIe x2	6	16
PCIe x4	4	16
PCIe x8	0	16
PCIe x16	0	8

Note: The PCIe Hard IP uses at least the bottom eight Embedded Multi-Die Interconnect Bridge (EMIB) channels, no matter how many PCIe lanes are enabled. Thus, these EMIB channels become unavailable for other protocols.

Figure 11. PCIe Hard IP Channel Configurations Per Transceiver Tile



The table below maps all transceiver channels to PCIe Hard IP channels in available tiles.

Table 8. PCIe Hard IP channel mapping across all tiles

Tile Channel Sequence	PCIe Hard IP Channel	Index within I/O Bank	Bottom Left Tile Bank Number	Top Left Tile Bank Number	Bottom Right Tile Bank Number	Top Right Tile Bank Number
23	N/A	5	1F	1N	4F	4N
22	N/A	4	1F	1N	4F	4N
21	N/A	3	1F	1N	4F	4N
20	N/A	2	1F	1N	4F	4N
19	N/A	1	1F	1N	4F	4N
18	N/A	0	1F	1N	4F	4N
17	N/A	5	1E	1M	4E	4M
16	N/A	4	1E	1M	4E	4M
15	15	3	1E	1M	4E	4M
14	14	2	1E	1M	4E	4M
13	13	1	1E	1M	4E	4M
12	12	0	1E	1M	4E	4M
11	11	5	1D	1L	4D	4L
10	10	4	1D	1L	4D	4L
9	9	3	1D	1L	4D	4L
8	8	2	1D	1L	4D	4L
7	7	1	1D	1L	4D	4L
6	6	0	1D	1L	4D	4L
5	5	5	1C	1K	4C	4K
4	4	4	1C	1K	4C	4K
3	3	3	1C	1K	4C	4K
2	2	2	1C	1K	4C	4K
1	1	1	1C	1K	4C	4K
0	0	0	1C	1K	4C	4K

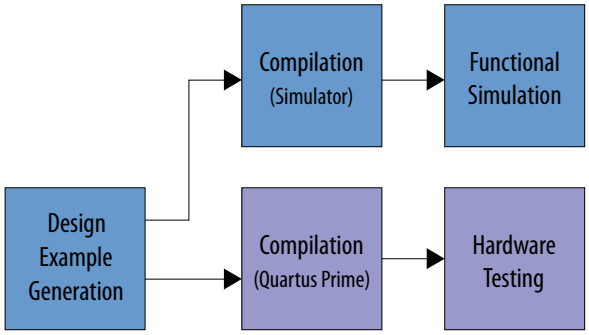
PCIe Soft IP Channel Usage

PCI Express soft IP PIPE-PHY cores available from third-party vendors are not subject to the channel usage restrictions described above. Refer to **Intel FPGA ➤ Products ➤ Intellectual Property** for more information about soft IP cores for PCI Express.

2. Quick Start Guide

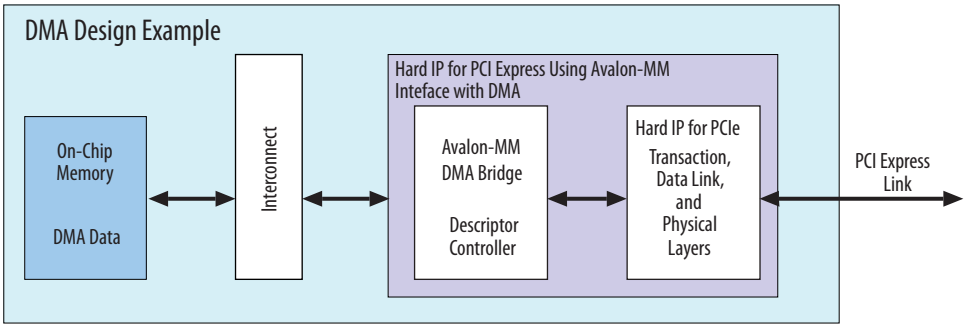
Using Intel Quartus Prime Pro Edition, you can generate a simple DMA design example for the Avalon-MM Intel Stratix 10 Hard IP for PCI Express IP core. The generated design example reflects the parameters that you specify. It automatically creates the files necessary to simulate and compile in the Intel Quartus Prime Pro Edition software. You can download the compiled design to the Intel Stratix 10-GX Development Board. To download to custom hardware, update the Intel Quartus Prime Settings File (`.qsf`) with the correct pin assignments .

Figure 12. Development Steps for the Design Example



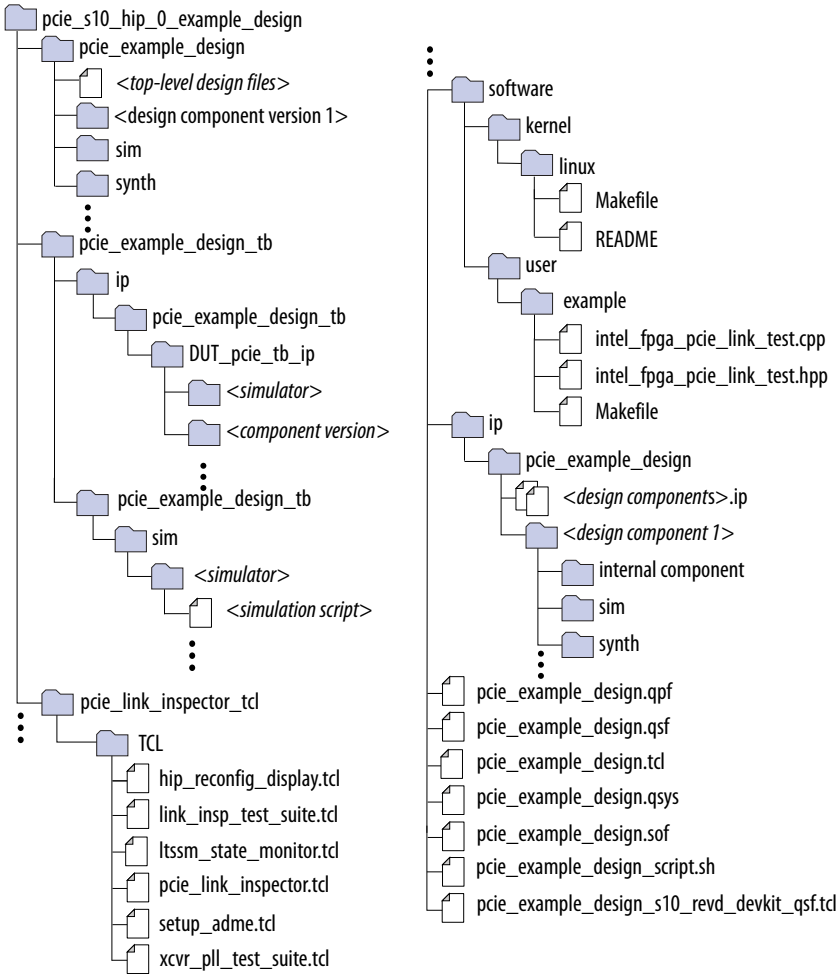
2.1. Design Components

Figure 13. Block Diagram for the Avalon-MM DMA for PCIe Design Example



2.2. Directory Structure

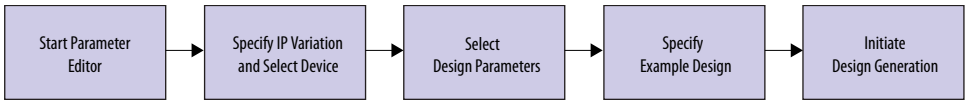
Figure 14. Directory Structure for the Generated Design Example



2.3. Generating the Design Example

Follow these steps to generate your design:

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 Stratix 10 (GX/SX/MX/TX)** and the **Target Device** for your design.
6. Click **Finish**.
7. In the IP Catalog locate and add the **Avalon-MM Intel Stratix 10 Hard IP for PCI Express**.
8. In the **New IP Variant** dialog box, specify a name for your IP. Click **Create**.
9. On the **IP Settings** tabs, specify the parameters for your IP variation.
10. On the **Example Designs** tab, make the following selections:
    a. For **Available Example Designs,** select **DMA**.

       Note: The **DMA** design example is only available when you turn on **Enable Avalon-MM DMA** on the **Avalon-MM Settings** tab.

       Note: If you do not turn on **Enable Avalon-MM DMA**, you can still choose either a **PIO** or a **Simple DMA** design example.

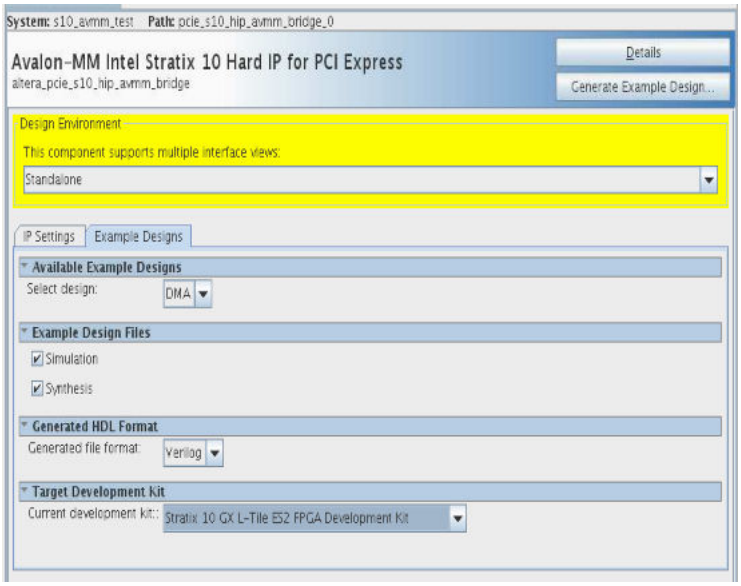
    b. For **Example Design Files**, turn on the **Simulation** and **Synthesis** options. If you do not need these simulation or synthesis files, leaving the corresponding option(s) turned off significantly reduces the example design generation time.

    c. For **Generated HDL Format**, only Verilog is available in the current release.

    d. For **Target Development Kit**, select the appropriate option.

       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.
11. Select **Generate Example Design** to create a design example that you can simulate and download to hardware. If you select one of the Intel Stratix 10 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, `<example_design>/pcie_s10_hip_avmm_bridge_0_example_design`, or choose another directory.

Figure 16. Example Design Tab



When you generate an Intel Stratix 10 example design, a file called `recommended_pinassignments_s10.txt` is created in the directory pcie_s10_hip_avmm_bridge_0_example_design.(2)

12. Click **Finish**. You may save your `.ip` file when prompted, but it is not required to be able to use the example design.
13. The prompt, **Recent changes have not been generated. Generate now?**, allows you to create files for simulation and synthesis of the IP core variation that you specified in Step 9 above. Click **No** if you only want to work with the design example you have generated.
14. Close the dummy project.
15. Open the example design project.
16. Compile the example design project to generate the `.sof` file for the complete example design. This file is what you download to a board to perform hardware verification.
17. Close your example design project.

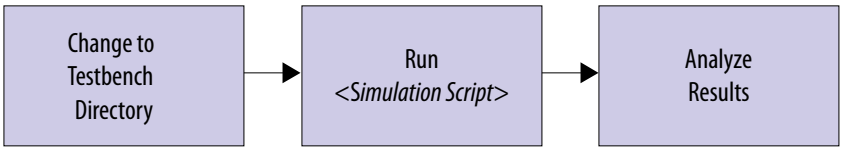
Related Information

AN-811: Using the Avery BFM for PCI Express Gen3x16 Simulation on Intel Stratix 10 Devices

(2) This file contains the recommended pin assignments for all the pins in the example design. If you select a development kit option in the pull-down menu for **Target Development Kit**, the pin assignments in the `recommended_pinassignments_s10.txt` file match those that are in the `.qsf` file in the same directory. If you chose **NONE** in the pull-down menu, the `.qsf` file does not contain any pin assignment. In this case, you can copy the pin assignments in the `recommended_pinassignments_s10.txt` file to the `.qsf` file. You can always change any pin assignment in the `.qsf` file to satisfy your design or board requirements.

2.4. Simulating the Design Example

Figure 17. Procedure



1. Change to the testbench simulation directory, `pcie_example_design_tb`.
2. Run the simulation script for the simulator of your choice. Refer to the table below.
3. Analyze the results.

Table 9. Steps to Run Simulation

Simulator	Working Directory	Instructions
ModelSim*	*<example_design>*/pcie_example_design_tb/pcie_example_design_tb/sim/mentor/	1. Invoke vsim (by typing `vsim`, which brings up a console window where you can run the following commands). 2. `do msim_setup.tcl` *Note:* Alternatively, instead of doing Steps 1 and 2, you can type: `vsim -c -do msim_setup.tcl`. 3. `ld_debug` 4. `run -all` 5. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"
VCS*	<example_design>/pcie_example_design_tb/pcie_example_design_tb/sim/synopsys/vcs	1. `sh vcs_setup.sh USER_DEFINED_COMPILE_OPTIONS="" USER_DEFINED_ELAB_OPTIONS="-xlrm\ uniq_prior_final" USER_DEFINED_SIM_OPTIONS=""` 2. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"
NCSim*	*<example_design>*/pcie_example_design_tb/pcie_example_design_tb/sim/cadence	1. `sh ncsim_setup.sh USER_DEFINED_SIM_OPTIONS="" USER_DEFINED_ELAB_OPTIONS="-timescale\ 1ns/1ps"` 2. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"
Xcelium* Parallel Simulator	*<example_design>*/pcie_example_design_tb/pcie_example_design_tb/sim/xcelium	1. `sh xcelium_setup.sh USER_DEFINED_SIM_OPTIONS="" USER_DEFINED_ELAB_OPTIONS ="-timescale\ 1ns/1ps\ -NOWARN\ CSINFI"` 2. A successful simulation ends with the following message, "Simulation stopped due to successful completion!"

The DMA testbench completes the following tasks:

1. Writes to the Endpoint memory using the DUT Endpoint non-bursting Avalon-MM master interface.
2. Reads from Endpoint memory using the DUT Endpoint non-bursting Avalon-MM master interface.
3. Verifies the data using the `shmem_chk_ok` task.
4. Writes to the Endpoint DMA controller, instructing the DMA controller to perform a MRd request to the PCIe address space in host memory.
5. Writes to the Endpoint DMA controller, instructing the DMA controller to perform a MWr request to PCIe address space in host memory. This MWr uses the data from the previous MRd.
6. Verifies the data using the `shmem_chk_ok` task.

The simulation reports, "Simulation stopped due to successful completion" if no errors occur.

Figure 18. Partial Transcript from Successful Simulation Testbench

```
# INFO:           88656 ns        Maximum Link Width: x8
# INFO:           88656 ns      Supported Link Speed: 8.0GT/s or 5.0GT/s or 2.5GT/s
# INFO:           88656 ns                 L0s Entry: Supported
# INFO:           88656 ns                 L1  Entry: Not Supported
# INFO:           88656 ns          L0s Exit Latency: 2 us to 4 us
# INFO:           88656 ns          L1  Exit Latency: Less Than 1 us
# INFO:           99928 ns BAR Address Assignments:
# INFO:           99928 ns BAR    Size       Assigned Address  Type
# INFO:           99928 ns ---    ----       ----------------
# INFO:           99928 ns BAR1:0   4 KBytes 00000000 80000000 Prefetchable
# INFO:           99928 ns BAR3:2  64 KBytes 00000000 80010000 Prefetchable
# INFO:           99928 ns BAR4   Disabled
# INFO:           99928 ns BAR5   Disabled
# INFO:           99928 ns ExpROM Disabled
# INFO:          100704 ns Completed configuration of Endpoint BARs.
# INFO:          101608 ns Starting Target Write/Read Test.
# INFO:          101608 ns   Target BAR = 0
# INFO:          101608 ns   Length = 000512, Start Offset = 000000
# INFO:          103096 ns   Target Write and Read compared okay!
# INFO:          103096 ns Starting DMA Read/Write Test.
# INFO:          103096 ns   Setup BAR = 2
# INFO:          103096 ns   Length = 000512, Start Offset = 000000
# INFO:          106882 ns Clear Interrupt INTA
# INFO:          111416 ns MSI recieved!
# INFO:          111416 ns   DMA Read and Write compared okay!
# SUCCESS: Simulation stopped due to successful completion!
# Simulation passed
```

2.5. Compiling the Design Example and Programming the Device

1. Navigate to `<project_dir>/pcie_s10_hip_avmm_bridge_0_example_design/` and open `pcie_example_design.qpf`.
2. On the Processing menu, select **Start Compilation**.
3. After successfully compiling your design, program the targeted device with the Programmer.

2.6. Installing the Linux Kernel Driver

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[3] 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 selects device by specifying the bus, device and function (BDF) numbers for the required 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
   ```
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.

   Expected result:
   ```
   Kernel driver in use: intel_fpga_pcie_drv
   ```

2.7. Running the Design Example Application

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
   ```

(3) 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.

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

— In automatic mode, the application automatically selects the device. The test selects the Intel Stratix 10 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 Stratix 10 GX 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:

```
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 0x100
************************************************
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
6: Change device
7: Enable SR-IOV
8: Do a link test for every enabled virtual function
   belonging to the current device
9: Perform DMA
10: Quit program
************************************************
> 0
Doing 100 writes and 100 reads . .
Number of write errors:     0
Number of read errors:      0
Number of DWORD mismatches: 0
```

```
Intel FPGA PCIe Link Test - Manual Mode
Version 1.0
0: Automatically select a device
1: Manually select a device
************************************************
> 1
Enter bus number:
> 1
Enter device number:
> 0
Enter function number:
> 0
BDF is 0x100
Enter BAR number (-1 for none):
> 4
Opened a handle to BAR 4 of a device with BDF 0x100
```

Related Information

PCIe Link Inspector Overview

Use the PCIe Link Inspector to monitor the link at the Physical, Data Link and Transaction Layers.

3. Interface Overview

The Avalon-MM Intel Stratix 10 Hard IP for PCIe IP core includes many interface types to implement different functions.

These include:

- Avalon-MM interfaces to translate the PCIe TLPs into standard memory-mapped reads and writes
- DMA interfaces to transfer large blocks of data

  Note: DMA operations are only available when the application interface width is 256-bit, but not when it is 64-bit. You choose the interface width by selecting **IP Settings**, then **System Settings**, and finally **Application interface width**.
- Standard PCIe serial interfaces to transfer data over the PCIe link or links
- System interfaces for interrupts, clocking, reset, and test
- Optional reconfiguration interface to dynamically change the value of configuration space registers at run-time
- Optional status interface for debug

Unless otherwise noted, all interfaces are synchronous to the rising edge of the main system clock `coreclkout_hip`. You enable the interfaces using the component GUI.

Related Information

Interfaces on page 55

3.1. Avalon-MM DMA Interfaces when Descriptor Controller Is Internally Instantiated

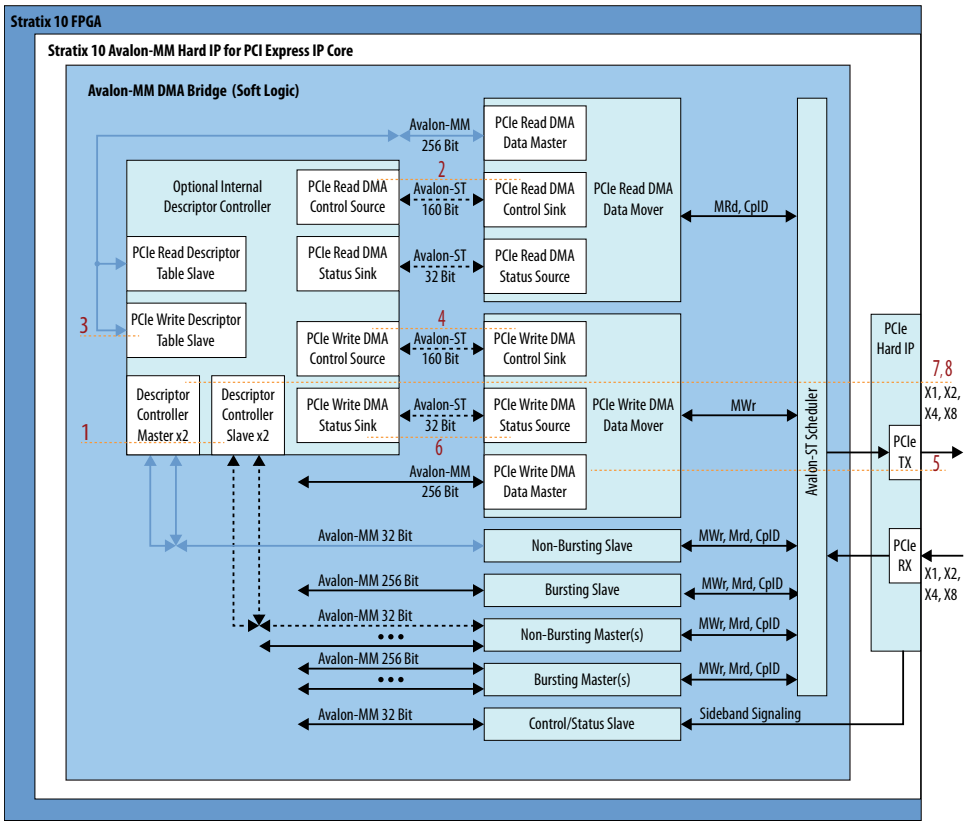
This configuration results from selecting both **Enable Avalon-MM DMA** and **Instantiate internal descriptor controller** in the component GUI.

The following figure shows the Avalon-MM DMA Bridge, implemented in soft logic. It interfaces to the Hard IP for PCIe through Avalon-ST interfaces.

In the following figure, Avalon-ST connections and the connection from the BAR0 non-bursting master to the Descriptor Controller slaves are internal. Dashed black lines show these connections. Connections between the Descriptor Controller Masters and the non-bursting slave and the connections between the Read DMA Data Master and the Descriptor Table Slaves are made in the Platform Designer. Blue lines show these connections.

Note: In the following diagrams and text descriptions, the terms Read and Write are from the system memory perspective. Thus, a Read transaction reads data from the system memory and writes it to the local memory in Avalon-MM address space. A Write transaction writes the data that was read from the local memory in Avalon-MM address space to the system memory.

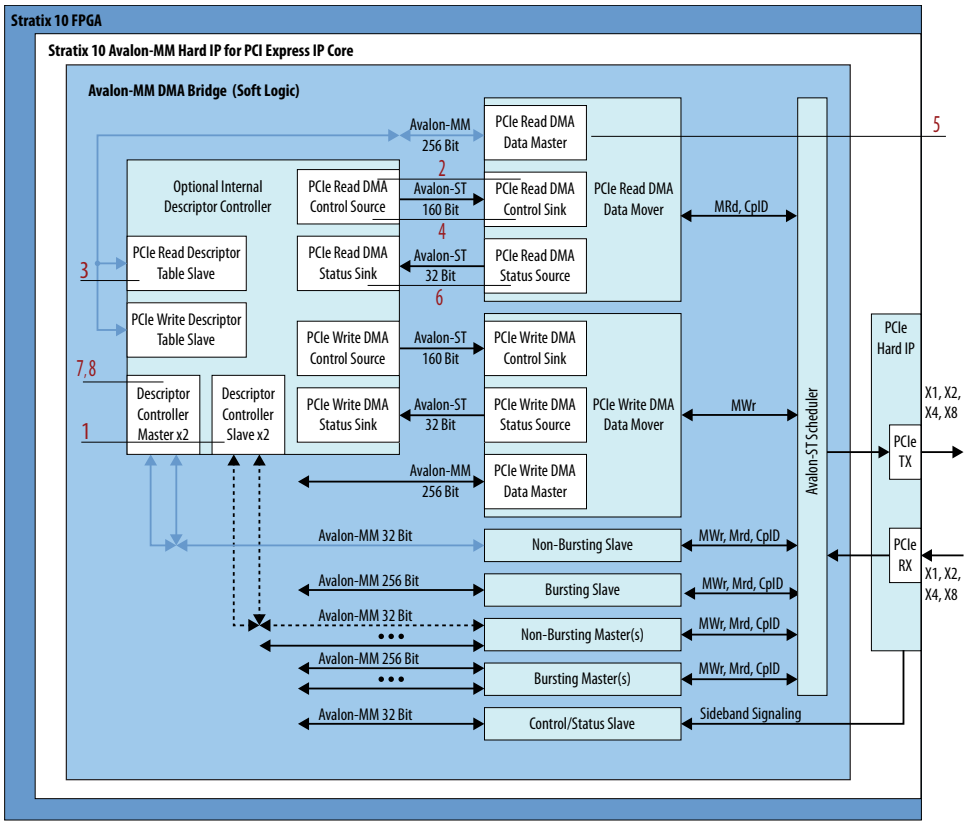
Figure 19. Avalon-MM DMA Bridge Block Diagram with Optional Internal Descriptor Controller (Showing DMA Write Flow)



The numbers in this figure describe the following steps in the DMA write flow:

1. The CPU writes registers in the Descriptor Controller Slave to start the DMA.
2. The Descriptor Controller instructs the Read Data Mover to fetch the descriptor table.
3. The Read Data Mover forwards the descriptor table to the PCIe Write Descriptor Table Slave.
4. The Descriptor Controller instructs the Write Data Mover to transfer data.
5. The Write Data Mover transfers data from FPGA to system memory.
6. The Write Data Mover notifies the Descriptor Controller of the completion of the data transfer using the done bit.
7. The Descriptor Controller Master updates the status of the descriptor table in system memory.
8. The Descriptor Controller Master sends an MSI interrupt to the host.

Figure 20. Avalon-MM DMA Bridge Block Diagram with Optional Internal Descriptor Controller (Showing DMA Read Flow)



The numbers in this figure describe the following steps in the DMA read flow:

1. The CPU writes registers in the Descriptor Controller Slave to start the DMA.
2. The Descriptor Controller instructs the Read Data Mover to fetch the descriptor table.
3. The Read Data Mover forwards the descriptor table to the PCIe Read Descriptor Table Slave.
4. The Descriptor Controller instructs the Read Data Mover to transfer data.
5. The Read Data Mover transfers data from system memory to FPGA.
6. The Read Data Mover notifies the Descriptor Controller of the completion of the data transfer using the done bit.
7. The Descriptor Controller Master updates the status of the descriptor table in system memory.
8. The Descriptor Controller Master sends an MSI interrupt to the host.

When the optional Descriptor Controller is included in the bridge, the Avalon-MM bridge includes the following Avalon interfaces to implement the DMA functionality:

- PCIe Read DMA Data Master (rd_dma): This is a 256-bit wide write only Avalon-MM master interface which supports bursts of up to 16 cycles with the `rd_dma*` prefix. The Read Data Mover uses this interface to write at high throughput the blocks of data that it has read from the PCIe system memory space. This interface writes descriptors to the Read and Write Descriptor table slaves and to any other Avalon-MM connected slaves interfaces.
- PCIe Write DMA Data Master (wr_dma): This read-only interface transfers blocks of data from the Avalon-MM domain to the PCIe system memory space at high throughput. It drives read transactions on its bursting Avalon-MM master interface. It also creates PCIe Memory Write (MWr) TLPs with data payload from Avalon-MM reads. It forwards the MWr TLPs to the Hard IP for transmission on the link. The Write Data Mover module decomposes the transfers into the required number of Avalon-MM burst read transactions and PCIe MWr TLPs. This is a bursting, 256-bit Avalon-MM interface with the `wr_dma` prefix.
- PCIe Read Descriptor Table Slave (rd_dts): This is a 256-bit Avalon-MM slave interface that supports write bursts of up to 16 cycles. The PCIe Read DMA Data Master writes descriptors to this table. This connection is made outside the DMA bridge because the Read Data Mover also typically connects to other Avalon-MM slaves. The prefix for this interface is `rd_dts`.
- PCIe Write Descriptor Table Slave (wr_dts): This is a 256-bit Avalon-MM slave interface that supports write bursts of up to 16 cycles. The PCIe Read DMA Data Master writes descriptors to this table. The PCIe Read DMA Data Master must connect to this interface outside the DMA bridge because the bursting master interface may also need to be connected to the destination of the PCIe Read Data Mover. The prefix for this interface is `wr_dts`.
- Descriptor Controller Master (DCM): This is a 32-bit, non-bursting Avalon-MM master interface with write-only capability. It controls the non-bursting Avalon-MM slave that transmits single DWORD DMA status information to the host. The prefixes for this interface are `wr_dcm and rd_dcm`.
- Descriptor Controller Slave (DCS): This is a 32-bit, non-bursting Avalon-MM slave interface with read and write access. The host accesses this interface through the BAR0 Non-Bursting Avalon-MM Master, to program the Descriptor Controller.

  Note: This is not a top-level interface of the Avalon-MM Bridge. Because it connects to BAR0, you cannot use BAR0 to access any other Avalon-MM slave interface.

Related Information

Programming Model for the DMA Descriptor Controller on page 105

3.2. Avalon-MM DMA Interfaces when Descriptor Controller is Externally Instantiated

This configuration results from selecting **Enable Avalon-MM DMA** and disabling **Instantiate internal descriptor controller** in the component GUI. This configuration requires you to include a custom DMA descriptor controller in your application.

Using the external DMA descriptor controller provides more flexibility. You can either modify the example design's DMA Descriptor Controller or replace it to meet your system requirements.You may need to modify the DMA Descriptor Controller for the following reasons:

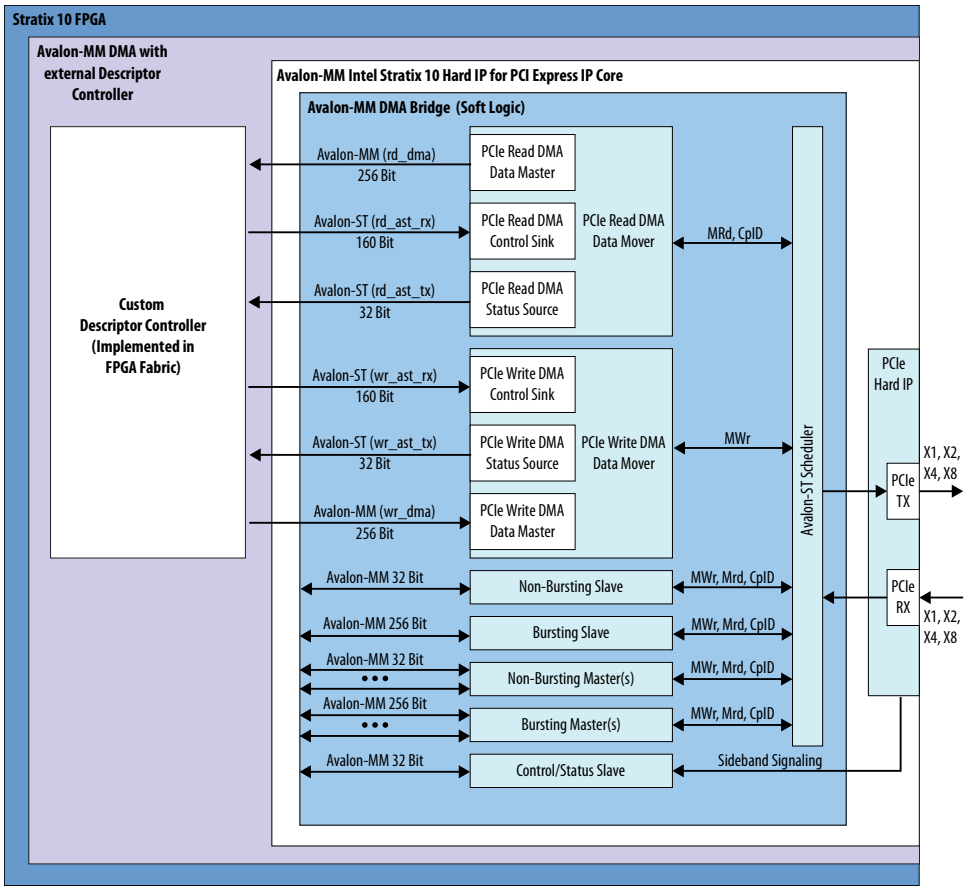
- To implement multi-channel operation
- To implement the descriptors as a linked list or to implement a custom DMA programming model
- To fetch descriptors from local memory, instead of system (host-side) memory

To interface to the DMA logic included in this variant, the custom DMA descriptor controller must implement the following functions:

- It must provide the descriptors to the PCIe Read DMA Data Mover and PCIe Write DMA Data Mover.
- It must process the status that the DMA Avalon-MM write (`wr_dcm`) and read (`rd_dcm`) masters provide.

The following figure shows the Avalon-MM DMA Bridge when the a custom descriptor controller drives the PCIe Read DMA and Write DMA Data Movers.

Figure 21. Avalon-MM DMA Bridge Block Diagram with Externally Instantiated Descriptor Controller



This configuration includes the PCIe Read DMA and Write DMA Data Movers. The custom DMA descriptor controller must connect to the following Data Mover interfaces:

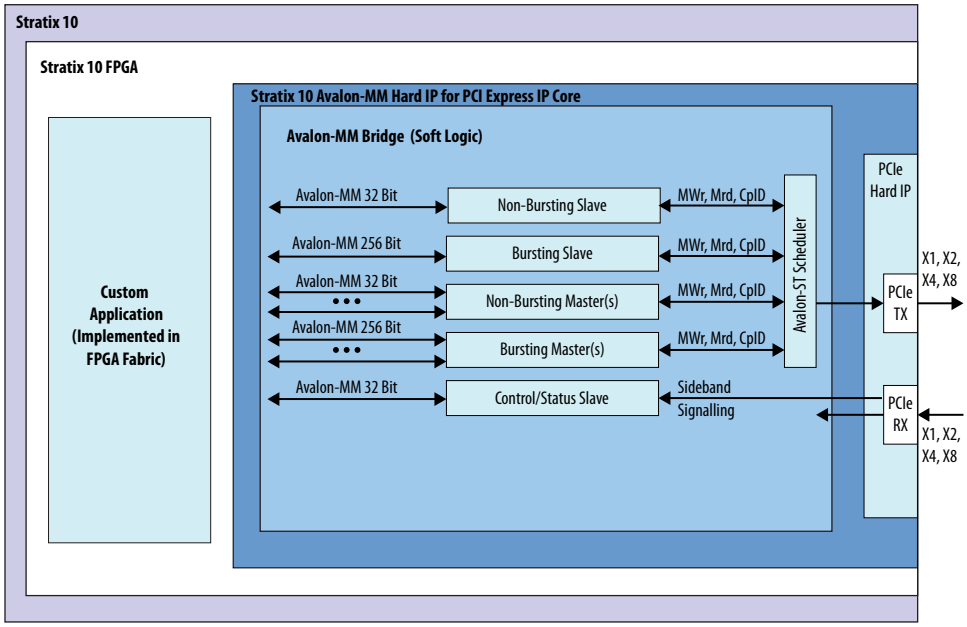
- PCIe Read DMA Control Sink: This is a 160-bit, Avalon-ST sink interface. The custom DMA descriptor controller drives descriptor table entries on this bus. The prefix for the interface is `rd_ast_rx*`.
- PCIe Write DMA Control Sink: This is a 160-bit, Avalon-ST sink interface. The custom DMA descriptor controller drives write table entries on this bus. The prefix for this interface is `wr_ast_rx*`.
- PCIe Read DMA Status Source: The Read Data Mover reports status to the custom DMA descriptor controller on this interface. The prefix for this interface is `rd_ast_tx_*`.
- PCIe Write DMA Status Source: The Write Data Mover reports status to the custom DMA descriptor controller on this interface. The prefix for this interface is `wr_ast_tx_*`.

3.3. Other Avalon-MM Interfaces

This configuration results from deselecting **Enable Avalon-MM DMA** in the component GUI.

This variant hides the complexity of the PCIe Protocol by translating between the TLPs exchanged on the PCIe link into memory-mapped reads and writes in the Avalon-MM domain. The following figure shows the Avalon-MM DMA Bridge interfaces available when the bridge does not enable the PCIe Read DMA and Write DMA Data Movers.

Figure 22. Avalon-MM DMA Bridge Block Diagram without DMA Functionality



3.3.1. Avalon-MM Master Interfaces

Avalon-MM Master modules translate PCI Express MRd and MWr TLP requests received from the PCI Express link to Avalon-MM read and write transactions on their Avalon-MM interface. The Avalon-MM master modules return the read data received on their Avalon-MM interface using PCI Express Completion TLPs (CplD).

Up to six Avalon-MM Master interfaces can be enabled at configuration time, one for each of the six supported BARs. Each of the enabled Avalon-MM Master interfaces can be set to be bursting or non-bursting in the component GUI. Bursting Avalon-MM Masters are designed for high throughput transfers, and the application interface data bus width can be either 256-bit or 64-bit. Non-bursting Avalon-MM Masters are designed for small transfers requiring finer granularity for byte enable control, or for control of 32-bit Avalon-MM Slaves. The prefix for signals comprising this interface is `rxm_bar<bar_num>*`.

Table 10. Avalon-MM Master Module Features

Avalon-MM Master Type	Data Bus Width	Max Burst Size	Byte Enable Granularity	Maximum Outstanding Read Requests
Non-bursting	32-bit	1 cycle	Byte	1
Bursting	256-bit	16 cycles	DWord(4)	32
Bursting	64-bit	64 cycles	Byte	16

3.3.2. Avalon-MM Slave Interfaces

Avalon-MM Slave Interfaces: The Avalon-MM Slave modules translate read and write transactions on their Avalon-MM interface to PCI Express MRd and MWr TLP requests. These modules return the data received in PCI Express Completion TLPs on the read data bus of their Avalon-MM interface.

Two versions of Avalon-MM Slave modules are available: the bursting Avalon-MM Slave is for high throughput transfers, and the application interface data bus width can be either 256-bit or 64-bit. The non-bursting Avalon-MM Slave is for small transfers requiring finer granularity for byte enable control. The prefix for the non-bursting Avalon-MM Slave interface is `txs*`. The prefix for the bursting Avalon-MM Slave interface is `hptxs_*`

Table 11. Avalon-MM Slave Module Features

Avalon-MM Slave Type	Data Bus Width	Max Burst Size	Byte Enable Granularity	Maximum Outstanding Read Requests
Non-bursting	32-bit	1 cycle	Byte	1
Bursting	256-bit	16 cycles	DWord	32
Bursting	64-bit	64 cycles	Byte	16

The bursting Avalon-MM Slave adheres to the maximum payload size and maximum read request size values set by the system software after enumeration. It generates multiple PCIe TLPs for a single Avalon-MM burst transaction when required.

Table 12. Number of TLPs generated for Each Burst Size as Function of Maximum Payload Size and Maximum Read Request Size

Burstcount	Maximum Payload Size or Maximum Read Request Size		
	128 bytes	256 bytes	512 bytes
1 – 4	1 TLP	1 TLP	1 TLP
5 – 8	2 TLPs	1 TLP	1 TLP
9 – 12	3 TLPs	2 TLPs	1 TLP
13 – 16	4 TLPs	2 TLPs	1 TLP

Note: The burst sizes in the table above are for the 256-bit application interface width.

3.3.3. Control Register Access (CRA) Avalon-MM Slave

This optional, 32-bit Avalon-MM Slave provides access to the `Control` and `Status` registers. You must enable this interface when you enable address mapping for any of the Avalon-MM slaves or if interrupts are implemented.

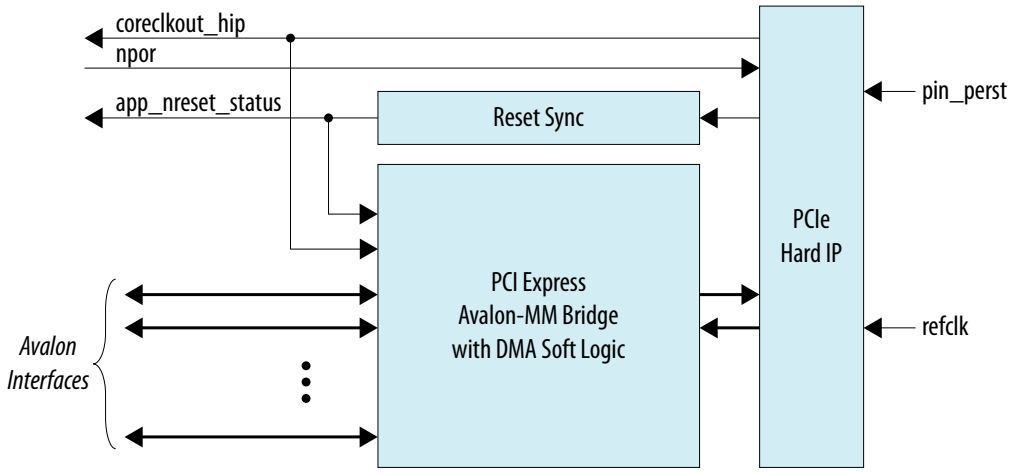
The address bus width of this interface is fixed at 15 bits. The prefix for this interface is `cra*`.

(4) Using less than DWORD granularity has unpredictable results. Buffers must be sized to accommodate DWORD granularity.

3.4. Clocks and Reset

The Stratix 10 Hard IP for PCI Express generates the Application clock, `coreclkout_hip` and reset signal. The Avalon-MM bridge provides a synchronized version of the reset signal, `app_nreset_status`, to the Application. This is an active low reset.

Figure 23. Avalon-MM Intel Stratix 10 Hard IP for PCI Express Clock and Reset Connections



Note: The input reference clock, `refclk`, must be stable and free-running at device power-up for a successful device configuration.

3.5. System Interfaces

TX and RX Serial Data

This differential, serial interface is the physical link between a Root Port and an Endpoint. The PCIe IP Core supports 1, 2, 4, 8, or 16 lanes. Gen1 at 2.5 GT/s, Gen2 at 5 GT/s and Gen3 at 8 GT/s are supported. Each lane includes a TX and RX differential pair. Data is striped across all available lanes.

PIPE

This is a parallel interface between the PCIe IP Core and PHY. The PIPE data bus is 32 bits. Each lane includes four control/data bits and other signals. It carries the TLP data before it is serialized. It is available for simulation only and provides more visibility for debugging.

Interrupts

The Stratix 10 Avalon-MM DMA Bridge can generate legacy interrupts when the `Interrupt Disable` bit, bit[10] of the Configuration Space `Command` register is set to 1'b0.

The Avalon-MM Bridge does not generate MSIs in response to a triggering event. However, the Application can cause MSI TLPs, which are single DWORD memory writes, to be created by one of the Avalon-MM slave interfaces.

To trigger an MSI, the Application performs a write to the address shown in the `msi_intfc[63:0]` bits, using the data shown in the `msi_intfc[79:64]` bits with the lower bits replaced with the particular MSI number.

The Application can also implement MSI-X TLPs, which are single DWORD memory writes. The MSI-X Capability structure points to an MSI-X table structure and MSI-X pending bit array (PBA) structure which are stored in system memory. This scheme is different than the MSI capability structure, which contains all the control and status information for the interrupts.

Hard IP Reconfiguration

This optional Avalon-MM interface allows you to dynamically update the value of read-only Configuration Space registers at run-time. It is available when **Enable dynamic reconfiguration of PCIe read-only registers** is enabled in the component GUI.

Hard IP Status

This optional interface includes the following signals that are useful for debugging

- Link status signals
- Interrupt status signals
- TX and RX parity error signals
- Correctable and uncorrectable error signals

Related Information

PCI Express Base Specification 3.0

4. Parameters

This chapter provides a reference for all the parameters of the Intel Stratix 10 Hard IP for PCI Express IP core.

Table 13. Design Environment Parameter

Starting in Intel Quartus Prime 18.0, there is a new parameter **Design Environment** in the parameters editor window.

Parameter	Value	Description
Design Environment	**Standalone** **System**	Identifies the environment that the IP is in. • The **Standalone** environment refers to the IP being in a standalone state where all its interfaces are exported. • The **System** environment refers to the IP being instantiated in a Platform Designer system.

Table 14. System Settings

Parameter	Value	Description
Application Interface Type	**Avalon-MM**	Selects the interface to the Application Layer.
Application Interface Width	**256-bit** **64-bit**	Selects the width of the interface to the Application Layer. *Note:* DMA operations are only supported when this parameter is set to **256-bit**.
Hard IP Mode	**Gen3x8, 256-bit interface, 250 MHz** **Gen3x4, 256-bit interface, 125 MHz** **Gen3x2, 256-bit interface, 125 MHz** **Gen3x1, 256-bit interface, 125 MHz** **Gen2x16, 256-bit interface, 250 MHz** **Gen2x8, 256-bit interface, 125 MHz** **Gen2x4, 256-bit interface, 125 MHz** **Gen2x2, 256-bit interface, 125 MHz** **Gen2x1, 256-bit interface, 125 MHz** **Gen1x16, 256-bit interface, 125 MHz** **Gen1x8, 256-bit interface, 125 MHz** **Gen1x4, 256-bit interface, 125 MHz** **Gen1x2, 256-bit interface, 125 MHz** **Gen1x1, 256-bit interface, 125 MHz** If the **Application Interface Width** selected is 64-bit, the only available **Hard IP Mode** configurations available are: • **Gen3x2, 64-bit interface, 250 MHz** • **Gen3x1, 64-bit interface, 125 MHz** • **Gen2x4, 64-bit interface, 250 MHz** • **Gen2x2, 64-bit interface, 125 MHz** • **Gen2x1, 64-bit interface, 125 MHz**	Selects the following elements: • The lane data rate. Gen1, Gen2, and Gen3 are supported • The Application Layer interface frequency The width of the data interface between the hard IP Transaction Layer and the Application Layer implemented in the FPGA fabric. *Note:* If the **Mode** selected is not available for the configuration chosen, an error message displays in the **Message** pane.

continued...

Parameter	Value	Description
	• **Gen1x8, 64-bit interface, 250 MHz** • **Gen1x4, 64-bit interface, 125 MHz** • **Gen1x2, 64-bit interface, 125 MHz** • **Gen1x1, 64-bit interface, 125 MHz**	
Port type	**Native Endpoint** **Root Port**	Specifies the port type. The Endpoint stores parameters in the Type 0 Configuration Space. The Root Port stores parameters in the Type 1 Configuration Space. A Root Port testbench is not available in the current release. If you select the Root Port, you have to create your own testbench.

4.1. Avalon-MM Settings

Table 15. Avalon-MM Settings

Parameter	Value	Description
Avalon-MM address width	**32-bit** **64-bit**	Specifies the address width for Avalon-MM RX master ports that access Avalon-MM slaves in the Avalon address domain. When you select **Enable Avalon-MM DMA** or **Enable non-bursting Avalon-MM slave interface with individual byte access (TXS)**, this value must be set to **64**.
Enable Avalon-MM DMA	**On/Off**	When **On**, the IP core includes Read DMA and Write DMA data movers.
Instantiate internal descriptor controller	**Enabled/Disabled**	When **On**, the descriptor controller is included in the Avalon-MM DMA bridge. When **Off**, the descriptor controller should be included as a separate external component, if required. The internal descriptor controller does not support Root Port mode.
Enable control register access (CRA) Avalon-MM slave port	**On/Off**	Allows read and write access to Avalon-MM bridge registers from the interconnect fabric using a specialized slave port. This option is required for **Requester/Completer** variants and optional for **Completer Only** variants. Enabling this option allows read and write access to Avalon-MM bridge registers, except in the Completer-Only single DWORD variations.
Export interrupt conduit interfaces	**On/Off**	When **On**, the IP core exports internal interrupt signals to the top-level RTL module. The exported signals support MSI, MSI-X, and legacy interrupts.
Enable hard IP status bus when using the Avalon-MM interface	**On/Off**	When you turn this option **On**, your top-level variant includes signals that are useful for debugging, including link training and status, and error signals. The following signals are included in the top-level variant: • Link status signals • ECC error signals • LTSSM signals • Configuration parity error signal
Enable non-bursting Avalon-MM Slave interface with individual byte access (TXS)	**On/Off**	When **On**, the non-bursting Avalon-MM slave interface is enabled. This interface is appropriate for low bandwidth applications such as accessing control and status registers.
Address width of accessible PCIe memory space (TXS)	22-64	Specifies the number of bits necessary to access the PCIe address space. (This parameter only displays when the TXS slave is enabled.)

continued...

Parameter	Value	Description
Enable high performance bursting Avalon-MM Slave interface (HPTXS)	**On/Off**	When **On**, the high performance Avalon-MM slave interface is enabled. This interface is appropriate for high bandwidth applications such as transferring blocks of data.
Enable mapping (HPTXS)	**On/Off**	Address mapping for 32-bit Avalon-MM slave devices allows system software to specify non-contiguous address pages in the PCI Express address domain. All high performance 32-bit Avalon-MM slave devices are mapped to the 64-bit PCI Express address space. The Avalon-MM Settings tab of the component GUI allows you to select the number and size of address mapping pages. Up to 10 address mapping pages are supported. The minimum page size is 4 KB. The maximum page size is 4 GB. When you enable address mapping, the slave address bus width is just large enough to fit the required address mapping pages. When address mapping is disabled, the Avalon-MM slave address bus is set to 64 bits. The Avalon-MM addresses are used as is in the resulting PCIe TLPs.
Address width of accessible PCIe memory space (TXS)	22-64	Specifies the number of bits necessary to access the PCIe address space. (This parameter only displays when the HPTXS slave is enabled.)
Number of address pages (HPTXS)	**1-512 pages**	Specifies the number of pages available for address translation tables. Refer to *Address Mapping for High-Performance Avalon-MM 32-Bit Slave Modules* for more information about address mapping.

Related Information

Address Mapping for High-Performance Avalon-MM 32-Bit Slave Modules on page 95

4.2. Base Address Registers

Table 16. BAR Registers

Parameter	Value	Description
Type	**Disabled** **64-bit prefetchable memory** **32-bit non-prefetchable memory**	If you select 64-bit prefetchable memory, 2 contiguous BARs are combined to form a 64-bit prefetchable BAR; you must set the higher numbered BAR to **Disabled**. A non-prefetchable 64-bit BAR is not supported because in a typical system, the maximum non-prefetchable memory window is 32 bits. Defining memory as prefetchable allows contiguous data to be fetched ahead. Prefetching memory is advantageous when the requestor may require more data from the same region than was originally requested. If you specify that a memory is prefetchable, it must have the following 2 attributes: • Reads do not have side effects such as changing the value of the data read • Write merging is allowed *Note:* BAR0 is not available if the internal descriptor controller is enabled.
Size	0-63	The platform design automatically determines the BAR based on the address width of the slave connected to the master port.
Enable burst capability for Avalon-MM Bar0-5 Master Port	**On/Off**	Determines the type of Avalon-MM master to use for this BAR. Two types are available:

continued...

Parameter	Value	Description
		• A high performance, 256-bit master with burst support. This type supports high bandwidth data transfers. • A non-bursting 32-bit master with byte level byte enables. This type supports access to control and status registers.

Note: If the Expansion ROM BAR of PF2 or PF3 is disabled, a memory read access to the BAR is responded to with 32'h0000_0000 indicating that the corresponding ROM BAR does not exist. Software should not take any further action to allocate memory space for the disabled ROM BAR. When the Expansion ROM BAR is enabled, the application is required to respond with 16'hAA55 to a memory read to the first two bytes of the ROM space.

4.3. Device Identification Registers

The following table lists the default values of the read-only registers in the PCI* Configuration Header Space. You can use the parameter editor to set the values of these registers. At run time, you can change the values of these registers using the optional Hard IP Reconfiguration block signals.

To access these registers using the Hard IP Reconfiguration interface, make sure that you follow the format of the `hip_reconfig_address[20:0]` as specified in the table *Hard IP Reconfiguration Signals* of the section *Hard IP Reconfiguration*. Use the address offsets specified in the table below for `hip_reconfig_address[11:0]` and set `hip_reconfig_address[20]` to 1'b1 for a PCIe space access.

Table 17. PCI Header Configuration Space Registers

Register Name	Default Value	Description
Vendor ID	0x00001172	Sets the read-only value of the `Vendor ID` register. This parameter cannot be set to 0xFFFF per the *PCI Express Base Specification*. Address offset: 0x000.
Device ID	0x00000000	Sets the read-only value of the `Device ID` register. Address offset: 0x000.
Revision ID	0x00000001	Sets the read-only value of the `Revision ID` register. Address offset: 0x008.
Class code	0x00000000	Sets the read-only value of the `Class Code` register. Address offset: 0x008.
Subsystem Vendor ID	0x00000000	Sets the read-only value of `Subsystem Vendor ID` register in the PCI Type 0 Configuration Space. This parameter cannot be set to 0xFFFF per the *PCI Express Base Specification*. This value is assigned by PCI-SIG to the device manufacturer. This value is only used in Root Port variants. Address offset: 0x02C.
Subsystem Device ID	0x00000000	Sets the read-only value of the `Subsystem Device ID` register in the PCI Type 0 Configuration Space. This value is only used in Root Port variants. Address offset: 0x02C

4.4. PCI Express and PCI Capabilities Parameters

This group of parameters defines various capability properties of the IP core. Some of these parameters are stored in the PCI Configuration Space - PCI Compatible Configuration Space. The byte offset indicates the parameter address.

4.4.1. Device Capabilities

Table 18. Device Registers

Parameter	Possible Values	Default Value	Address	Description
Maximum payload sizes supported	**128 bytes** **256 bytes** **512 bytes**	512 bytes	0x074	Specifies the maximum payload size supported. This parameter sets the read-only value of the max payload size supported field of the Device Capabilities register. A 128-byte read request size results in the lowest latency for typical systems.

4.4.2. Link Capabilities

Table 19. Link Capabilities

Parameter	Value	Description
Link port number (Root Port only)	**0x01**	Sets the read-only value of the port number field in the `Link Capabilities` register. This parameter is for Root Ports only. It should not be changed.
Slot clock configuration	**On/Off**	When you turn this option **On**, indicates that the Endpoint uses the same physical reference clock that the system provides on the connector. When **Off**, the IP core uses an independent clock regardless of the presence of a reference clock on the connector. This parameter sets the Slot Clock Configuration bit (bit 12) in the `PCI Express Link Status` register.

4.4.3. MSI and MSI-X Capabilities

Table 20. MSI and MSI-X Capabilities

Parameter	Value	Address	Description
MSI messages requested	**1, 2, 4, 8, 16, 32**	0x050[31:16]	Specifies the number of messages the Application Layer can request. Sets the value of the `Multiple Message Capable` field of the `Message Control` register.
MSI-X Capabilities			
Implement MSI-X	**On/Off**		When **On**, adds the MSI-X capability structure, with the parameters shown below.
	Bit Range		
Table size	[10:0]	0x068[26:16]	System software reads this field to determine the MSI-X Table size <*n*>, which is encoded as <*n–1*>. For example, a returned value of 2047 indicates a table size of 2048. This field is read-only in the MSI-X Capability Structure. Legal range is 0–2047 (2^{11}).
Table offset	[31:0]		Points to the base of the MSI-X Table. The lower 3 bits of the table BAR indicator (BIR) are set to zero by software to form a 64-bit qword-aligned offset. This field is read-only.

continued...

Parameter	Value	Address	Description
Table BAR indicator	[2:0]		Specifies which one of a function's BARs, located beginning at 0x10 in Configuration Space, is used to map the MSI-X table into memory space. This field is read-only. Legal range is 0–5.
Pending bit array (PBA) offset	[31:0]		Used as an offset from the address contained in one of the function's Base Address registers to point to the base of the MSI-X PBA. The lower 3 bits of the PBA BIR are set to zero by software to form a 32-bit qword-aligned offset. This field is read-only in the MSI-X Capability Structure. (5)
Pending BAR indicator	[2:0]		Specifies the function Base Address registers, located beginning at 0x10 in Configuration Space, that maps the MSI-X PBA into memory space. This field is read-only in the MSI-X Capability Structure. Legal range is 0–5.

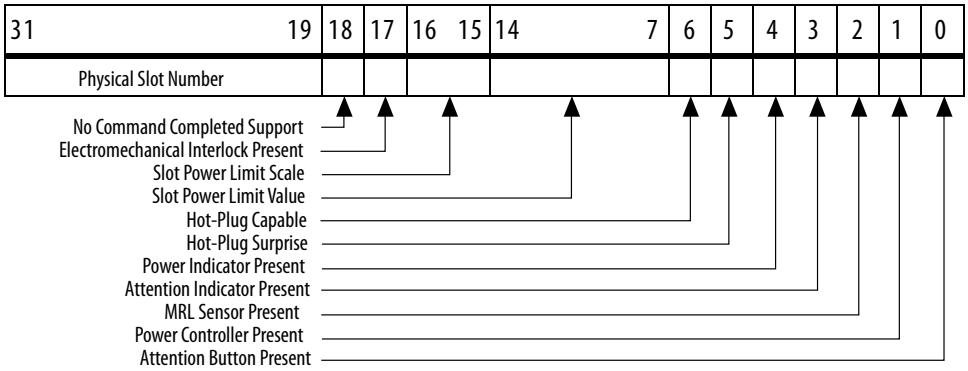
4.4.4. Slot Capabilities

Table 21. Slot Capabilities

Parameter	Value	Description
Use Slot register	**On/Off**	This parameter is only supported in Root Port mode. The slot capability is required for Root Ports if a slot is implemented on the port. Slot status is recorded in the `PCI Express Capabilities` register. Defines the characteristics of the slot. You turn on this option by selecting **Enable slot capability**. Refer to the figure below for bit definitions. Not applicable for Avalon-MM DMA.
Slot power scale	`0–3`	Specifies the scale used for the **Slot power limit**. The following coefficients are defined: • 0 = 1.0x • 1 = 0.1x • 2 = 0.01x • 3 = 0.001x The default value prior to hardware and firmware initialization is b'00. Writes to this register also cause the port to send the `Set_Slot_Power_Limit` Message. Refer to Section 6.9 of the *PCI Express Base Specification Revision* for more information.
Slot power limit	`0–255`	In combination with the **Slot power scale value**, specifies the upper limit in watts on power supplied by the slot. Refer to Section 7.8.9 of the *PCI Express Base Specification* for more information.
Slot number	`0-8191`	Specifies the slot number.

(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.

Figure 24. Slot Capability



4.4.5. Power Management

Table 22. Power Management Parameters

Parameter	Value	Description
Endpoint L0s acceptable latency	**Maximum of 64 ns** **Maximum of 128 ns** **Maximum of 256 ns** **Maximum of 512 ns** **Maximum of 1 us** **Maximum of 2 us** **Maximum of 4 us** **No limit**	This design parameter specifies the maximum acceptable latency that the device can tolerate to exit the L0s state for any links between the device and the root complex. It sets the read-only value of the Endpoint L0s acceptable latency field of the `Device Capabilities Register` (0x084). This Endpoint does not support the L0s or L1 states. However, in a switched system there may be links connected to switches that have L0s and L1 enabled. This parameter is set to allow system configuration software to read the acceptable latencies for all devices in the system and the exit latencies for each link to determine which links can enable Active State Power Management (ASPM). This setting is disabled for Root Ports. The default value of this parameter is 64 ns. This is a safe setting for most designs.
Endpoint L1 acceptable latency	**Maximum of 1 us** **Maximum of 2 us** **Maximum of 4 us** **Maximum of 8 us** **Maximum of 16 us** **Maximum of 32 us** **Maximum of 64 ns** **No limit**	This value indicates the acceptable latency that an Endpoint can withstand in the transition from the L1 to L0 state. It is an indirect measure of the Endpoint's internal buffering. It sets the read-only value of the Endpoint L1 acceptable latency field of the `Device Capabilities Register`. This Endpoint does not support the L0s or L1 states. However, a switched system may include links connected to switches that have L0s and L1 enabled. This parameter is set to allow system configuration software to read the acceptable latencies for all devices in the system and the exit latencies for each link to determine which links can enable Active State Power Management (ASPM). This setting is disabled for Root Ports. The default value of this parameter is 1 µs. This is a safe setting for most designs.

The Intel Stratix 10 Avalon-ST Hard IP for PCI Express and Intel Stratix 10 Avalon-MM Hard IP for PCI Express do not support the L1 or L2 low power states. If the link ever gets into these states, performing a reset (by asserting `pin_perst`, for example) allows the IP core to exit the low power state and the system to recover.

These IP cores also do not support the in-band beacon or sideband WAKE# signal, which are mechanisms to signal a wake-up event to the upstream device.

4.4.6. Vendor Specific Extended Capability (VSEC)

Table 23. VSEC

Parameter	Value	Description
User ID register from the Vendor Specific Extended Capability	Custom value	Sets the read-only value of the 16-bit User ID register from the Vendor Specific Extended Capability. This parameter is only valid for Endpoints.

4.5. Configuration, Debug and Extension Options

Table 24. Configuration, Debug and Extension Options

Parameter	Value	Description
Enable hard IP dynamic reconfiguration of PCIe read-only registers	**On/Off**	When **On**, you can use the Hard IP reconfiguration bus to dynamically reconfigure Hard IP read-only registers. For more information refer to *Hard IP Reconfiguration Interface*. With this parameter set to **On**, the hip_reconfig_clk port is visible on the block symbol of the Avalon-MM Hard IP component. In the System Contents window, connect a clock source to this hip_reconfig_clk port. For example, you can export hip_reconfig_clk and drive it with a free-running clock on the board whose frequency is in the range of 100 to 125 MHz. Alternatively, if your design includes a clock bridge driven by such a free-running clock, the out_clk of the clock bridge can be used to drive hip_reconfig_clk.
Enable transceiver dynamic reconfiguration	**On/Off**	When **On**, provides an Avalon-MM interface that software can drive to change the values of transceiver registers. With this parameter set to **On**, the xcvr_reconfig_clk, reconfig_pll0_clk, and reconfig_pll1_clk ports are visible on the block symbol of the Avalon-MM Hard IP component. In the System Contents window, connect a clock source to these ports. For example, you can export these ports and drive them with a free-running clock on the board whose frequency is in the range of 100 to 125 MHz. Alternatively, if your design includes a clock bridge driven by such a free-running clock, the out_clk of the clock bridge can be used to drive these ports.
Enable Native PHY, ATX PLL, and fPLL ADME for Toolkit	**On/Off**	When **On**, the generated IP includes an embedded Altera Debug Master Endpoint (ADME) that connects internally to an Avalon-MM slave interface for dynamic reconfiguration. The ADME can access the transceiver reconfiguration space. It can perform certain test and debug functions via JTAG using the System Console.
Enable PCIe Link Inspector	**On/Off**	When **On**, the PCIe Link Inspector is enabled. Use this interface to monitor the PCIe link at the Physical, Data Link and Transaction layers. You can also use the Link Inspector to reconfigure some transceiver registers. You must turn on **Enable transceiver dynamic reconfiguration**, **Enable dynamic reconfiguration of PCIe read-only registers** and **Enable Native PHY, ATX PLL, and fPLL ADME for** to use this feature. For more information about using the PCIe Link Inspector refer to *Link Inspector Hardware* in the *Troubleshooting and Observing Link Status* appendix.

Related Information

Hard IP Reconfiguration on page 81

4.6. PHY Characteristics

Table 25. PHY Characteristics

Parameter	Value	Description
Gen2 TX de-emphasis	**3.5dB** **6dB**	Specifies the transmit de-emphasis for Gen2. Intel recommends the following settings: • 3.5dB: Short PCB traces • 6.0dB: Long PCB traces.
VCCR/VCCT supply voltage for the transceiver	**1_1V** **1_0V**	Allows you to report the voltage supplied by the board for the transceivers.

4.7. Example Designs

Table 26. Example Designs

Parameter	Value	Description
Available Example Designs	**DMA** **Simple DMA** **PIO**	When you select the **DMA** option, the generated example design includes a direct memory access application. This application includes upstream and downstream transactions. The **DMA** example design uses the Write Data Mover, Read Data Mover, and a custom Descriptor Controller. The **Simple DMA** example design does not use the Write Data Mover and Read Data Mover, and uses a standard Intel Descriptor Controller. When you select the PIO option, the generated design includes a target application including only downstream transactions.
Simulation	**On/Off**	When **On**, the generated output includes a simulation model.
Synthesis	**On/Off**	When **On**, the generated output includes a synthesis model.
Generated HDL format	**Verilog/VHDL**	Only Verilog HDL is available in the current release.
Target Development Kit	**None** **Intel Stratix 10 H-Tile ES1 Development Kit** **Intel Stratix 10 L-Tile ES2 Development Kit**	Select the appropriate development board. If you select one of the development boards, system generation overwrites the device you selected with the device on that development board. *Note:* If you select **None**, system generation does not make any pin assignments. You must make the assignments in the `.qsf` file.

5. Designing with the IP Core

5.1. Generation

You can use the the Intel Quartus Prime Pro Edition IP Catalog or the Platform Designer to define and generate an Avalon-MM Intel Stratix 10 Hard IP for PCI Express IP Core custom component.

For information about generating your custom IP refer to the topics listed below.

Related Information

- Parameters on page 35
- Creating a System with Platform Designer
- Stratix 10 Product Table

5.2. Simulation

The Intel Quartus Prime Pro Edition software optionally generates a functional simulation model, a testbench or design example, and vendor-specific simulator setup scripts when you generate your parameterized PCI Express IP core. For Endpoints, the generation creates a Root Port BFM.

Note: Root Port example design generation is not supported in this release of Intel Quartus Prime Pro Edition.

The Intel Quartus Prime Pro Edition supports the following simulators.

Table 27. Supported Simulators

Vendor	Simulator	Version	Platform
Aldec	Active-HDL *	10.3	Windows
Aldec	Riviera-PRO *	2016.10	Windows, Linux
Cadence	Incisive Enterprise * (NCSim*)	15.20	Linux
Cadence	Xcelium* Parallel Simulator	17.04.014	Linux
Mentor Graphics	ModelSim PE*	10.5c	Windows
Mentor Graphics	ModelSim SE*	10.5c	Windows, Linux
Mentor Graphics	QuestaSim*	10.5c	Windows, Linux
Synopsys	VCS/VCS MX*	2016,06-SP-1	Linux

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 PCIe IP variation. This BFM allows you to create and run simple task stimuli with configurable parameters to exercise basic functionality of the 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. To ensure the best verification coverage possible, Intel suggests strongly that you obtain commercially available PCI Express verification IP and tools, or do your own extensive hardware testing or both.

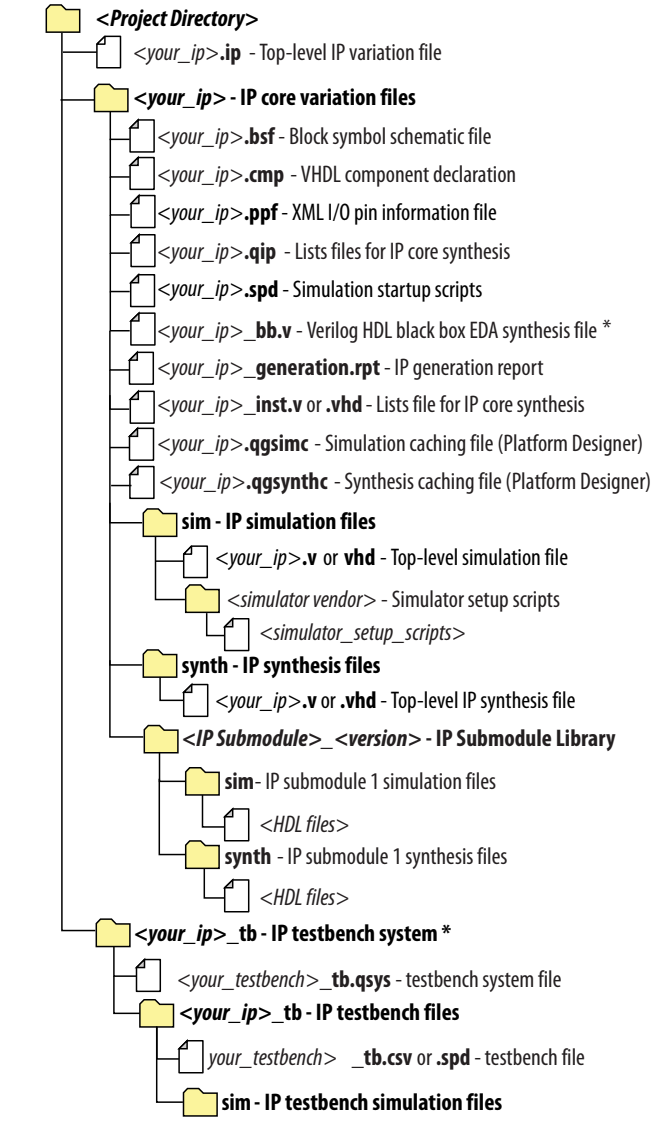
Related Information

- Introduction to Intel FPGA IP Cores, Simulating Intel FPGA IP Cores
- Simulation Quick-Start

5.3. IP Core Generation Output (Intel Quartus Prime Pro Edition)

The Intel Quartus Prime software generates the following output file structure for individual IP cores that are not part of a Platform Designer system.

Figure 25. Individual IP Core Generation Output (Intel Quartus Prime Pro Edition)



* If supported and enabled for your IP core variation.

Table 28. Output Files of Intel FPGA IP Generation

File Name	Description
<your_ip>.ip	Top-level IP variation file that contains the parameterization of an IP core in your project. If the IP variation is part of a Platform Designer system, the parameter editor also generates a .qsys file.
<your_ip>.cmp	The VHDL Component Declaration (.cmp) file is a text file that contains local generic and port definitions that you use in VHDL design files.
<your_ip>_generation.rpt	IP or Platform Designer generation log file. Displays a summary of the messages during IP generation.

continued...

File Name	Description
<*your_ip*>.qgsimc (Platform Designer systems only)	Simulation caching file that compares the .qsys and .ip files with the current parameterization of the Platform Designer system and IP core. This comparison determines if Platform Designer can skip regeneration of the HDL.
<*your_ip*>.qgsynth (Platform Designer systems only)	Synthesis caching file that compares the .qsys and .ip files with the current parameterization of the Platform Designer system and IP core. This comparison determines if Platform Designer can skip regeneration of the HDL.
<*your_ip*>.csv	Contains information about the upgrade status of the IP component.
<*your_ip*>.bsf	A symbol representation of the IP variation for use in Block Diagram Files (.bdf).
<*your_ip*>.spd	Input file that ip-make-simscript requires to generate simulation scripts. The .spd file contains a list of files you generate for simulation, along with information about memories that you initialize.
<*your_ip*>.ppf	The Pin Planner File (.ppf) stores the port and node assignments for IP components you create for use with the Pin Planner.
<*your_ip*>_bb.v	Use the Verilog blackbox (_bb.v) file as an empty module declaration for use as a blackbox.
<*your_ip*>_inst.v or _inst.vhd	HDL example instantiation template. Copy and paste the contents of this file into your HDL file to instantiate the IP variation.
<*your_ip*>.regmap	If the IP contains register information, the Intel Quartus Prime software generates the .regmap file. The .regmap file describes the register map information of master and slave interfaces. This file complements the .sopcinfo file by providing more detailed register information about the system. This file enables register display views and user customizable statistics in System Console.
<*your_ip*>.svd	Allows HPS System Debug tools to view the register maps of peripherals that connect to HPS within a Platform Designer system. During synthesis, the Intel Quartus Prime software stores the .svd files for slave interface visible to the System Console masters in the .sof file in the debug session. System Console reads this section, which Platform Designer queries for register map information. For system slaves, Platform Designer accesses the registers by name.
<*your_ip*>.v <*your_ip*>.vhd	HDL files that instantiate each submodule or child IP core for synthesis or simulation.
mentor/	Contains a msim_setup.tcl script to set up and run a simulation.
aldec/	Contains a script rivierapro_setup.tcl to setup and run a simulation.
/synopsys/vcs /synopsys/vcsmx	Contains a shell script vcs_setup.sh to set up and run a simulation. Contains a shell script vcsmx_setup.sh and synopsys_sim.setup file to set up and run a simulation.
/cadence	Contains a shell script ncsim_setup.sh and other setup files to set up and run an simulation.
/xcelium	Contains an Parallel simulator shell script xcelium_setup.sh and other setup files to set up and run a simulation.
/submodules	Contains HDL files for the IP core submodule.
<*IP submodule*>/	Platform Designer generates /synth and /sim sub-directories for each IP submodule directory that Platform Designer generates.

5.4. Channel Layout and PLL Usage

The following figures show the channel layout and PLL usage for Gen1, Gen2 and Gen3, x1, x2, x4, x8 and x16 variants of the Intel Stratix 10 Avalon-MM Hard IP core. Note that the missing variant Gen3 x16 is supported by another Intel Stratix 10 IP core (the Intel Stratix 10 Avalon-MM Hard IP+ core). For more details on the Avalon-MM Hard IP+ core, refer to https://www.intel.com/content/www/us/en/programmable/documentation/sox1520633403002.html.

The channel layout is the same for the Avalon-ST and Avalon-MM interfaces to the Application Layer.

Note: All of the PCIe hard IP instances in Intel Stratix 10 devices are x16. Channels 8-15 are available for other protocols when fewer than 16 channels are used. Refer to *Channel Availability* for more information.

Figure 26. Gen1 and Gen2 x1

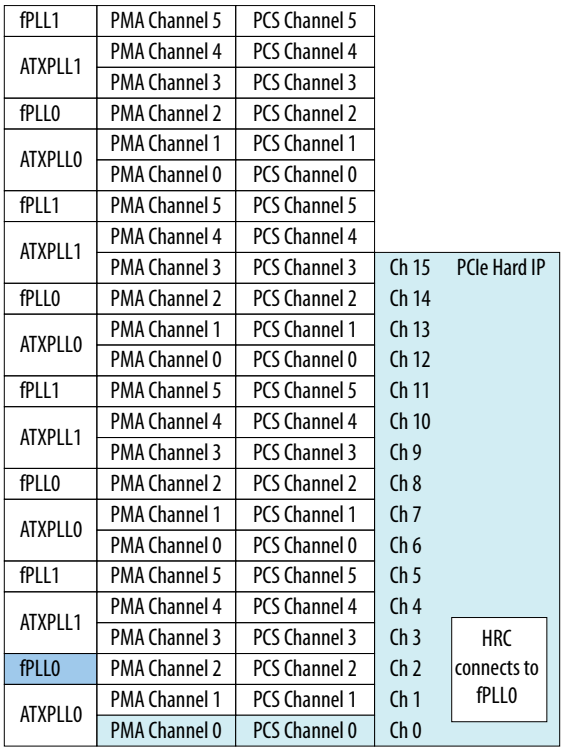


Figure 27. Gen1 and Gen2 x2

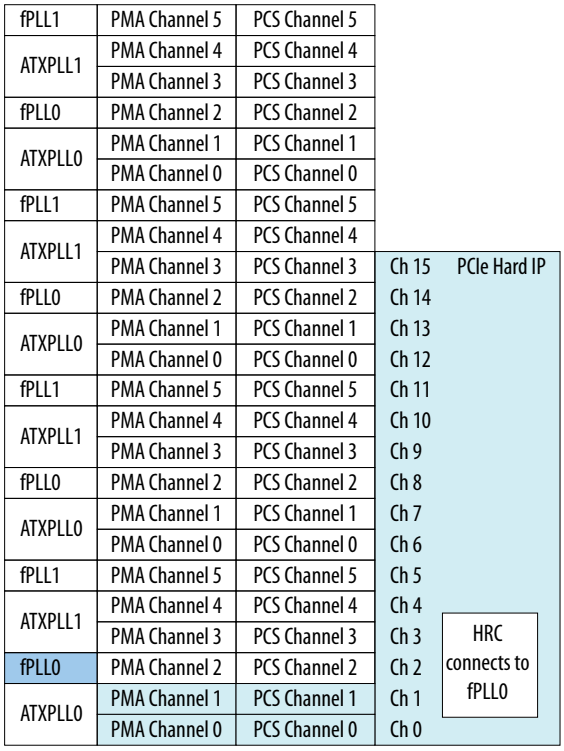


Figure 28. Gen1 and Gen2 x4

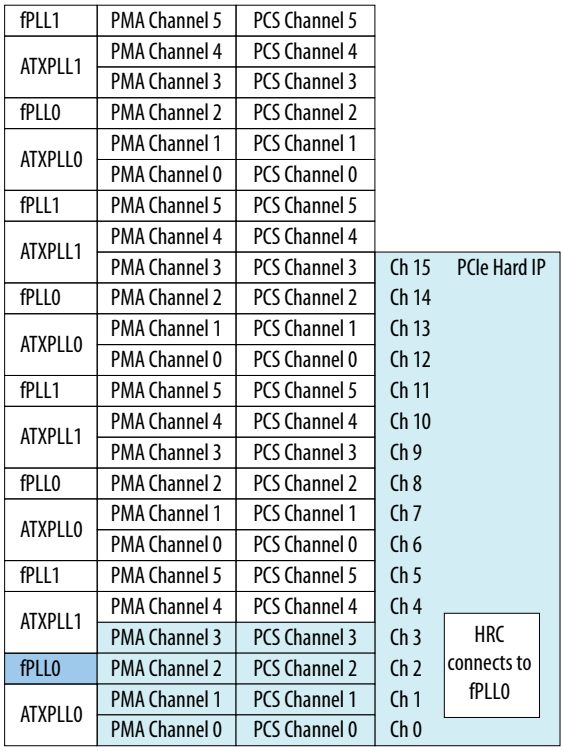


Figure 29. Gen1 and Gen2 x8

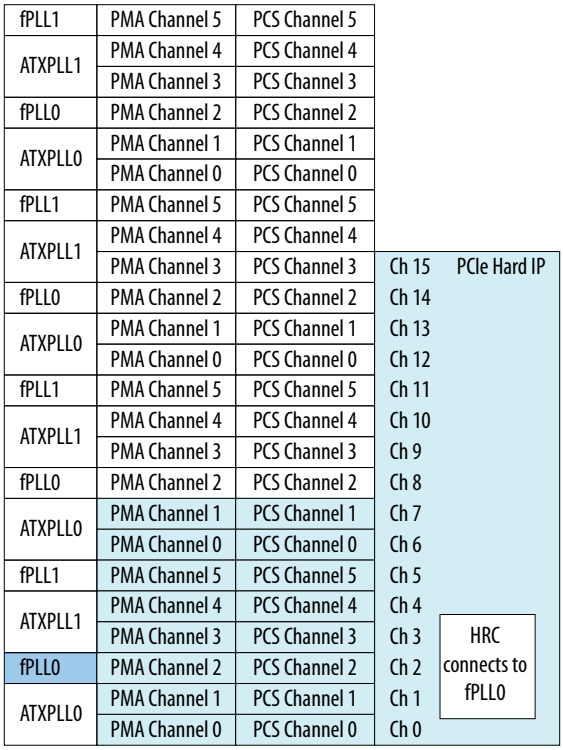


Figure 30. Gen1 and Gen2 x16

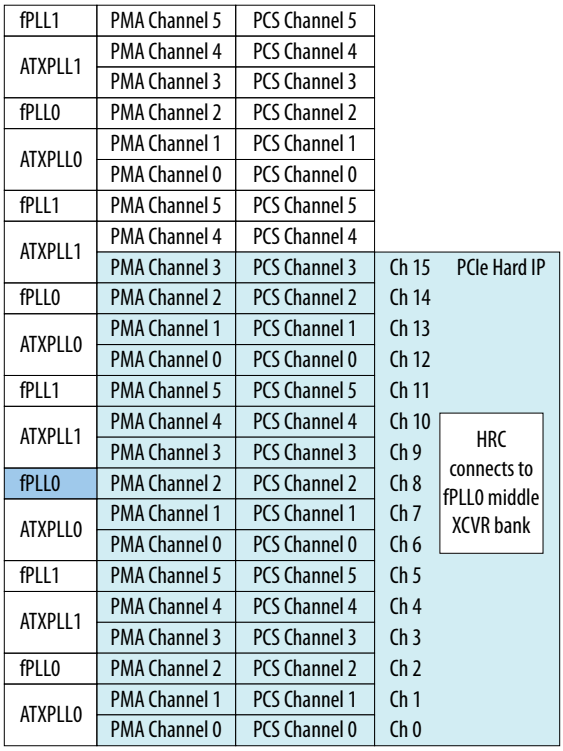


Figure 31. Gen3 x1

fPLL1	PMA Channel 5	PCS Channel 5		
ATXPLL1	PMA Channel 4	PCS Channel 4		
	PMA Channel 3	PCS Channel 3		
fPLL0	PMA Channel 2	PCS Channel 2		
ATXPLL0	PMA Channel 1	PCS Channel 1		
	PMA Channel 0	PCS Channel 0		
fPLL1	PMA Channel 5	PCS Channel 5		
ATXPLL1	PMA Channel 4	PCS Channel 4		
	PMA Channel 3	PCS Channel 3	Ch 15	PCIe Hard IP
fPLL0	PMA Channel 2	PCS Channel 2	Ch 14	
ATXPLL0	PMA Channel 1	PCS Channel 1	Ch 13	
	PMA Channel 0	PCS Channel 0	Ch 12	
fPLL1	PMA Channel 5	PCS Channel 5	Ch 11	
ATXPLL1	PMA Channel 4	PCS Channel 4	Ch 10	
	PMA Channel 3	PCS Channel 3	Ch 9	
fPLL0	PMA Channel 2	PCS Channel 2	Ch 8	
ATXPLL0	PMA Channel 1	PCS Channel 1	Ch 7	
	PMA Channel 0	PCS Channel 0	Ch 6	
fPLL1	PMA Channel 5	PCS Channel 5	Ch 5	
ATXPLL1	PMA Channel 4	PCS Channel 4	Ch 4	HRC connects to fPLL0 & ATXPLL0
	PMA Channel 3	PCS Channel 3	Ch 3	
fPLL0	PMA Channel 2	PCS Channel 2	Ch 2	
ATXPLL0 (Gen3)	PMA Channel 1	PCS Channel 1	Ch 1	
	PMA Channel 0	PCS Channel 0	Ch 0	

Figure 32. Gen3 x2

fPLL1	PMA Channel 5	PCS Channel 5		
ATXPLL1	PMA Channel 4	PCS Channel 4		
	PMA Channel 3	PCS Channel 3		
fPLL0	PMA Channel 2	PCS Channel 2		
ATXPLL0	PMA Channel 1	PCS Channel 1		
	PMA Channel 0	PCS Channel 0		
fPLL1	PMA Channel 5	PCS Channel 5		
ATXPLL1	PMA Channel 4	PCS Channel 4		
	PMA Channel 3	PCS Channel 3	Ch 15	PCIe Hard IP
fPLL0	PMA Channel 2	PCS Channel 2	Ch 14	
ATXPLL0	PMA Channel 1	PCS Channel 1	Ch 13	
	PMA Channel 0	PCS Channel 0	Ch 12	
fPLL1	PMA Channel 5	PCS Channel 5	Ch 11	
ATXPLL1	PMA Channel 4	PCS Channel 4	Ch 10	
	PMA Channel 3	PCS Channel 3	Ch 9	
fPLL0	PMA Channel 2	PCS Channel 2	Ch 8	
ATXPLL0	PMA Channel 1	PCS Channel 1	Ch 7	
	PMA Channel 0	PCS Channel 0	Ch 6	
fPLL1	PMA Channel 5	PCS Channel 5	Ch 5	
ATXPLL1	PMA Channel 4	PCS Channel 4	Ch 4	HRC connects to fPLL0 & ATXPLL0
	PMA Channel 3	PCS Channel 3	Ch 3	
fPLL0	PMA Channel 2	PCS Channel 2	Ch 2	
ATXPLL0 (Gen3)	PMA Channel 1	PCS Channel 1	Ch 1	
	PMA Channel 0	PCS Channel 0	Ch 0	

Figure 33. Gen3 x4

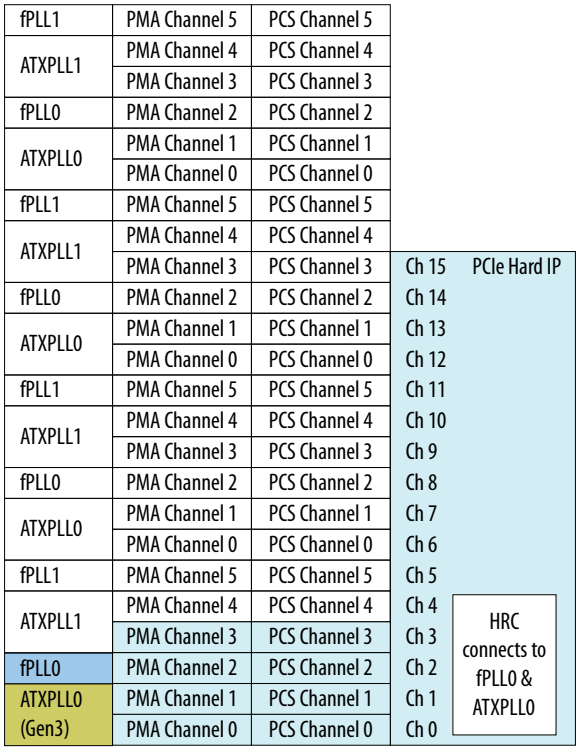
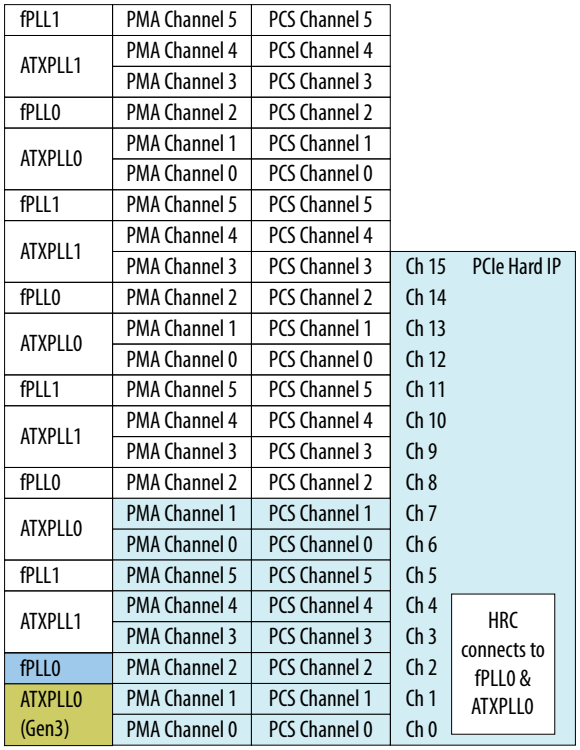


Figure 34. Gen3 x8



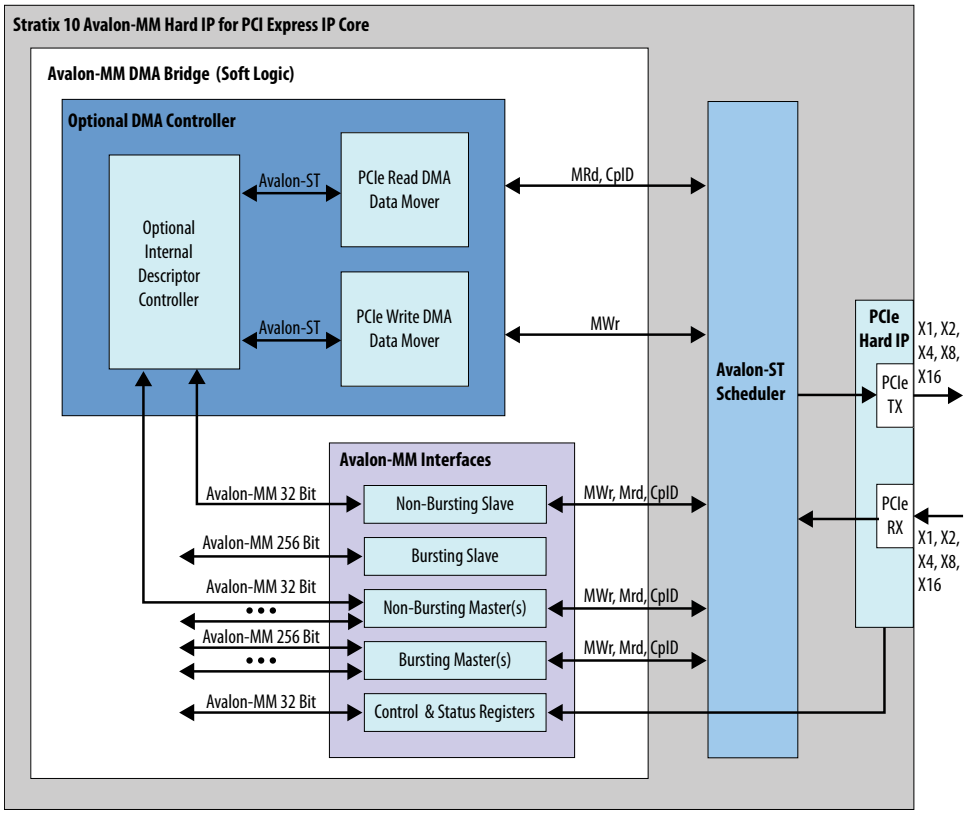
Related Information

Channel Availability on page 15

6. Block Descriptions

The Avalon-MM Stratix 10 Hard IP for PCI Express IP core combines the features of the Avalon-MM and Avalon-MM DMA variants of previous generations.The Avalon-MM DMA Bridge includes these functions in soft logic. The DMA bridge is a front-end to the Hard IP for PCI Express IP core. An Avalon-ST scheduler links the DMA bridge and PCIe IP core. It provides round-robin access to TX and RX data streams.

Figure 35. Avalon-MM Stratix 10 Hard IP for PCI Express Block Diagram



You can enable the individual optional modules of the DMA bridge in the component GUI. The following constraints apply:

- You must enable the PCIe Read DMA module if the PCIe Write DMA module and the Internal DMA Descriptor Controller are enabled. PCIe Read DMA fetches descriptors from the host.
- You must enable the Control Register Access (CRA) Avalon-MM slave port if address mapping is enabled.
-
- When you enable the internal DMA Descriptor Controller, the BAR0 Avalon-MM master is not available. The DMA Descriptor Controller uses this interfaces.

6.1. Interfaces

6.1.1. Intel Stratix 10 DMA Avalon-MM DMA Interface to the Application Layer

This section describes the top-level interfaces in the PCIe variant when it includes the high-performance, burst-capable read data mover and write data mover modules.

Figure 36. Avalon-MM DMA Bridge with Internal Descriptor Controller

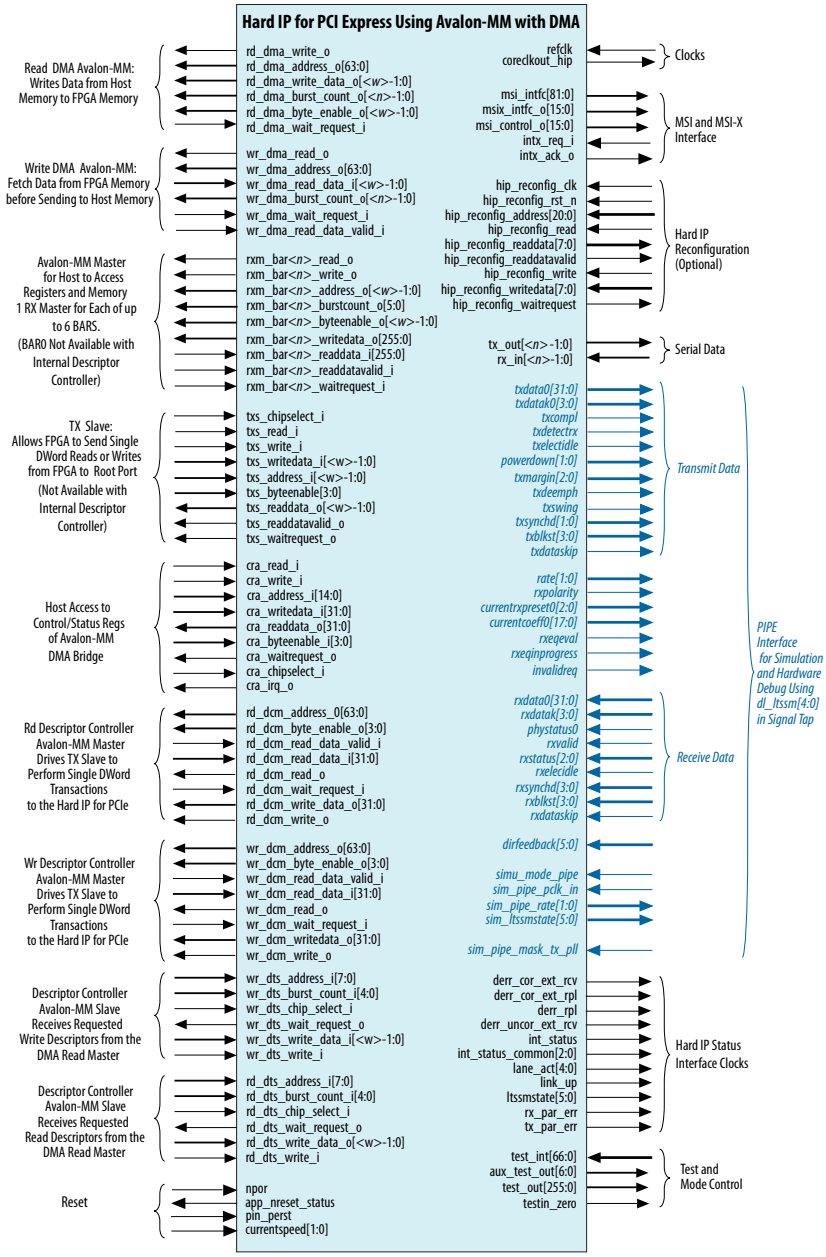
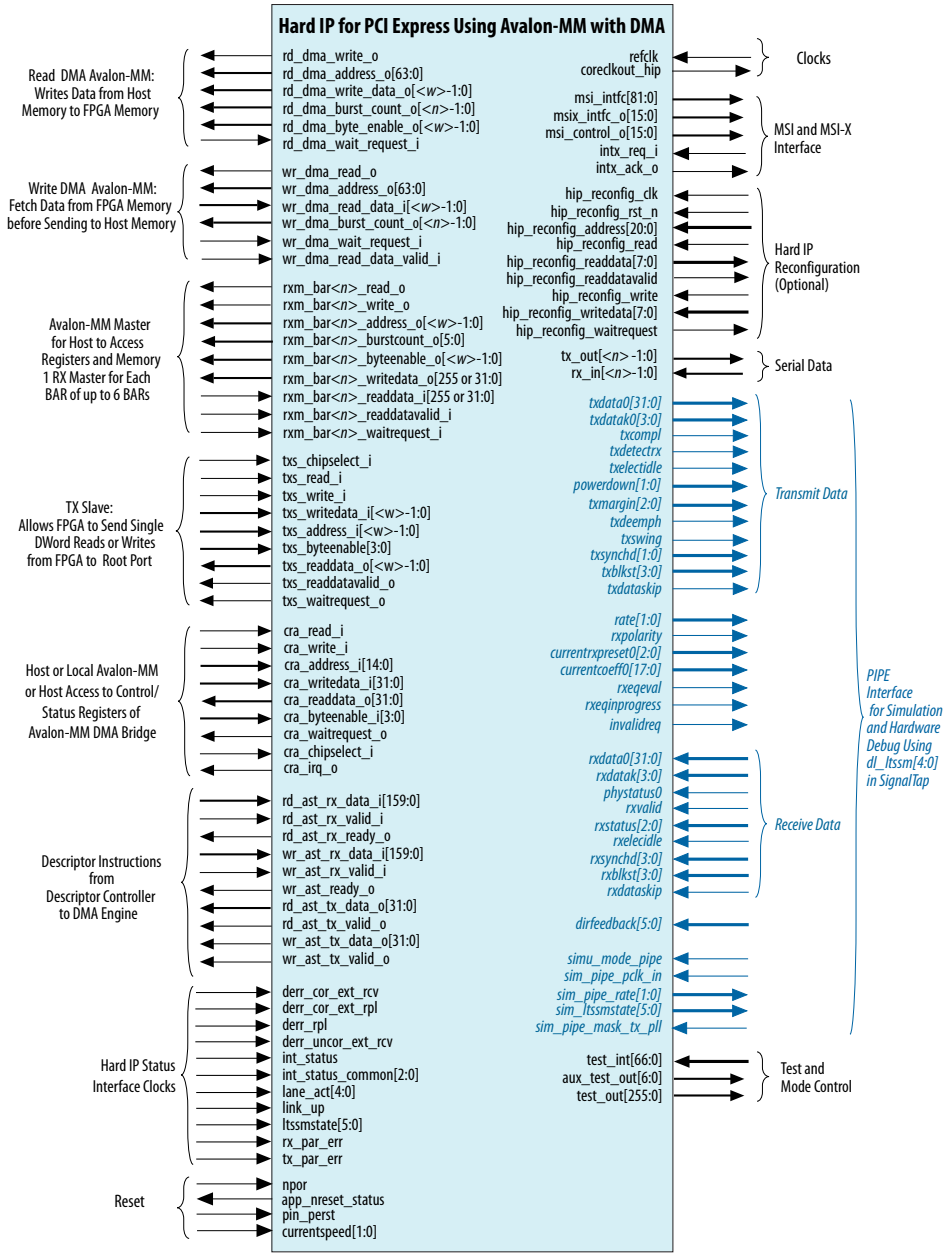


Figure 37. Avalon-MM DMA Bridge with External Descriptor Controller



This section describes the interfaces that are required to implement the DMA. All other interfaces are described in the next section, *Avalon-MM Interface to the Application Layer*.

Related Information

Avalon-MM Interface to the Application Layer on page 67

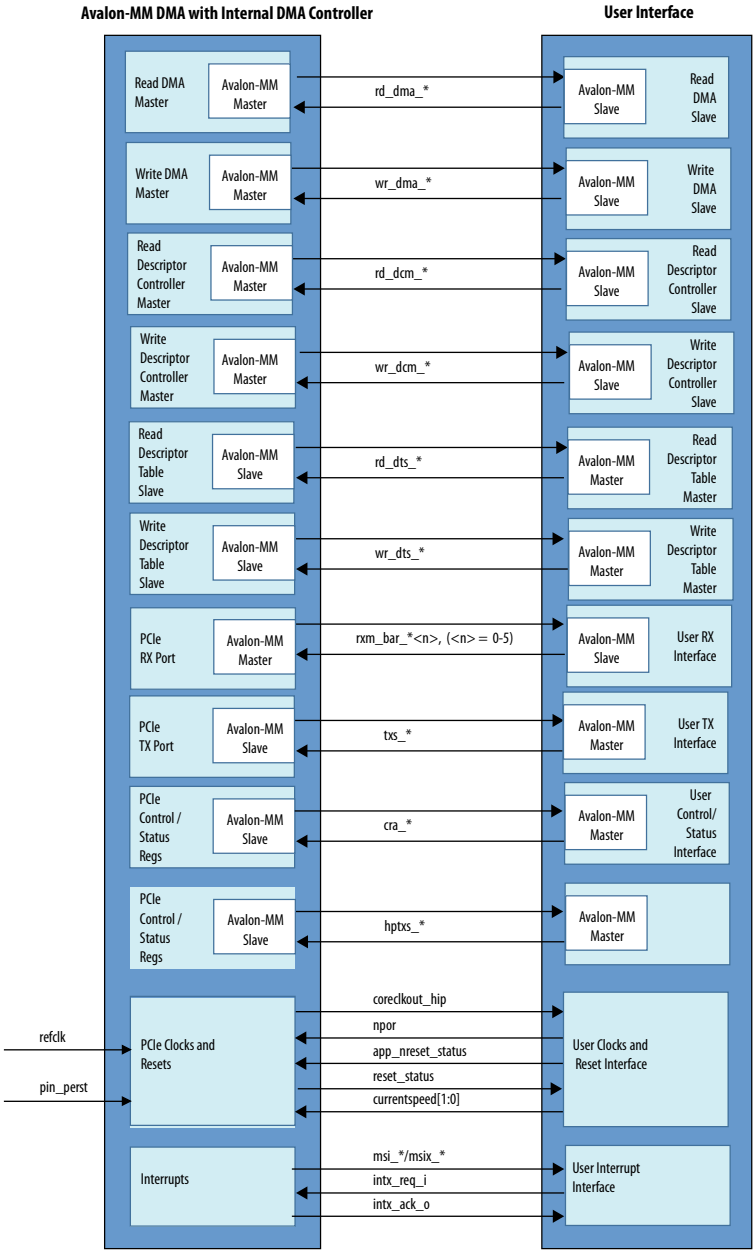
6.1.1.1. Descriptor Controller Interfaces when Instantiated Internally

The Descriptor Controller controls the Read DMA and Write DMA Data Movers. It provides a 32-bit Avalon-MM slave interface to control and manage data flow from PCIe system memory to Avalon-MM memory and in the reverse direction.

The Descriptor Controller includes two, 128-entry FIFOs to store the read and write descriptor tables. The Descriptor Controller forwards the descriptors to the Read DMA and Write DMA Data Movers.

The Data Movers send completion status to the Read Descriptor Controller and Write Descriptor Controller. The Descriptor Controller forwards status and MSI to the host using the TX slave port.

Figure 38. Connections: User Application to Avalon-MM DMA Bridge with Internal Descriptor Controller



6.1.1.1.1. Read Data Mover

The Read Data module sends memory read TLPs. It writes the completion data to an external Avalon-MM interface through the high throughput Read Master port. This data mover operates on descriptors the IP core receives from the DMA Descriptor Controller.

The Read DMA Avalon-MM Master interface performs the following functions:

1. Provides the Descriptor Table to the Descriptor Controller

The Read Data Mover sends PCIe system memory read requests to fetch the descriptor table from PCIe system memory. This module then writes the returned descriptor entries in to the Descriptor Controller FIFO using this Avalon-MM interface.

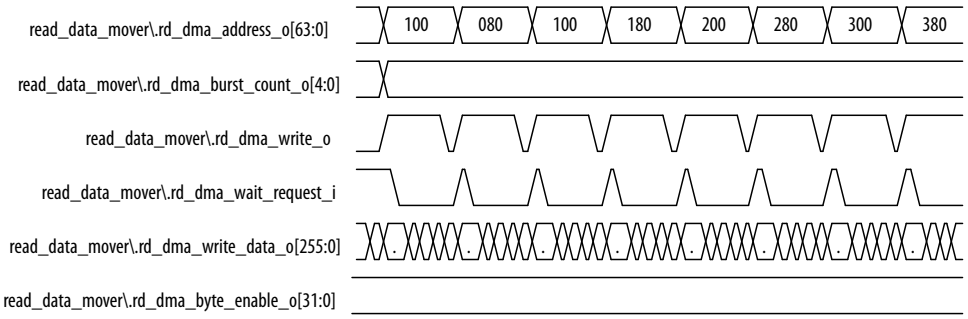
2. Writes Data to Memory Located in Avalon-MM Space

After a DMA Read finishes fetching data from the source address in PCIe system memory, the Read Data Mover module writes the data to the destination address in Avalon-MM address space via this interface.

Table 29. Read DMA 256-Bit Avalon-MM Master Interface

Signal Name	Direction	Description
rd_dma_write_o	Output	When asserted, indicates that the Read DMA module is ready to write read completion data to a memory component in the Avalon-MM address space.
rd_dma_address_o[63:0]	Output	Specifies the write address in the Avalon-MM address space for the read completion data.
rd_dma_write_data_o[255:0]	Output	The read completion data to be written to the Avalon-MM address space.
rd_dma_burst_count_o[4:0]	Output	Specifies the burst count in 128- or 256-bit words. This bus is 5 bits for the 256-bit interface. It is 6 bits for the 128-bit interface.
rd_dma_byte_enable_o[31:0]	Output	Specifies which DWORDs are valid.
rd_dma_wait_request_i	Input	When asserted, indicates that the memory is not ready to receive data.

Figure 39. Read DMA Avalon-MM Master Writes Data to FPGA Memory



6.1.1.1.2. Read Descriptor Controller Avalon-MM Master interface

The Read Descriptor Controller Avalon-MM master interface drives the non-bursting Avalon-MM slave interface. The Read Descriptor Controller uses this interface to write descriptor status to the PCIe domain and possibly to MSI when MSI messages are enabled. This Avalon-MM master interface is only available for variants with the internally instantiated Descriptor Controller.

By default MSI interrupts are enabled. You specify the **Number of MSI messages requested** on the **MSI** tab of the parameter editor. The MSI Capability Structure is defined in *Section 6.8.1 MSI Capability Structure* of the *PCI Local Bus Specification*.

Table 30. Read Descriptor Controller Avalon-MM Master interface

Signal Name	Direction	Description
rd_dcm_address_o[63:0]	Output	Specifies the descriptor status table or MSI address.
rd_dcm_byte_enable_o[3:0]	Output	Specifies which data bytes are valid.
rd_dcm_read_data_valid_i	Input	When asserted, indicates that the read data is valid.
rd_dcm_read_data_i[31:0]	Input	Specifies the read data of the descriptor status table entry addressed.
rd_dcm_read_o	Output	When asserted, indicates a read transaction. Currently, this is a write-only interface so that this signal never asserts.
rd_dcm_wait_request_i	Input	When asserted, indicates that the connected Avalon-MM slave interface is busy and cannot accept a transaction.
rd_dcm_write_data_o[31:0]	Output	Specifies the descriptor status or MSI data..
rd_dcm_write_o	Output	When asserted, indicates a write transaction.

6.1.1.1.3. Write Descriptor Controller Avalon-MM Master Interface

The Avalon-MM Descriptor Controller Master interface is a 32-bit single-DWORD master with wait request support. The Write Descriptor Controller uses this interface to write status back to the PCI-Express domain and possibly MSI when MSI messages are enabled. This Avalon-MM master interface is only available for the internally instantiated Descriptor Controller.

By default MSI interrupts are enabled. You specify the **Number of MSI messages requested** on the **MSI** tab of the parameter editor. The MSI Capability Structure is defined in *Section 6.8.1 MSI Capability Structure* of the *PCI Local Bus Specification*.

Table 31. Write Descriptor Controller Avalon-MM Master interface

Signal Name	Direction	Description
wr_dcm_address_o[63:0]	Output	Specifies the descriptor status table or MSI address.
wr_dcm_byte_enable_o[3:0]	Output	Specifies which data bytes are valid.
wr_dcm_read_data_valid_i	Input	When asserted, indicates that the read data is valid.
wr_dcm_read_data_i[31:0]	Output	Specifies the read data for the descriptor status table entry addressed.
wr_dcm_read_o	Output	When asserted, indicates a read transaction.
wr_dcm_wait_request_i	Input	When asserted, indicates that the Avalon-MM slave device is not ready to respond.
wr_dcm_writedata_o[31:0]	Output	Specifies the descriptor status table or MSI address.
wr_dcm_write_o	Output	When asserted, indicates a write transaction.

6.1.1.1.4. Read Descriptor Table Avalon-MM Slave Interface

This interface is available when you select the internal Descriptor Controller. It receives the Read DMA descriptors which are fetched by the Read Data Mover. Connect the interface to the Read DMA Avalon-MM master interface.

Table 32. Read Descriptor Table Avalon-MM Slave Interface

Signal Name	Direction	Description
rd_dts_address_i[7:0]	Input	Specifies the descriptor table address.
rd_dts_burst_count_i[4:0]	Input	Specifies the burst count of the transaction in words.
rd_dts_chip_select_i	Input	When asserted, indicates that the read targets this slave interface.
rd_dts_write_data_i[255:0]	Input	Specifies the descriptor.
rd_dts_write_i	Input	When asserted, indicates a write transaction.
rd_dts_wait_request_o	Output	When asserted, indicates that the Avalon-MM slave device is not ready to respond.

6.1.1.1.5. Write Descriptor Table Avalon-MM Slave Interface

This interface is available when you select the internal Descriptor Controller. This interface receives the Write DMA descriptors which are fetched by Read Data Mover. Connect the interface to the Read DMA Avalon-MM master interface.

Table 33. Write Descriptor Table Avalon-MM Slave Interface

Signal Name	Direction	Description
wr_dts_address_i[7:0]	Input	Specifies the descriptor table address.
wr_dts_burst_count_i[4:0] or [5:0]	Input	Specifies the burst count of the transaction in words.
wr_dts_chip_select_i	Input	When asserted, indicates that the write is for this slave interface.
wr_dts_wait_request_o	Output	When asserted, indicates that this interface is busy and is not ready to respond.
wr_dts_write_data_i[255:0]	Input	Drives the descriptor table entry data.
wr_dts_write_i	Input	When asserted, indicates a write transaction.

6.1.1.2. Write DMA Avalon-MM Master Port

The Write Data Mover module fetches data from the Avalon-MM address space using this interface before issuing memory write requests to transfer data to PCIe system memory.

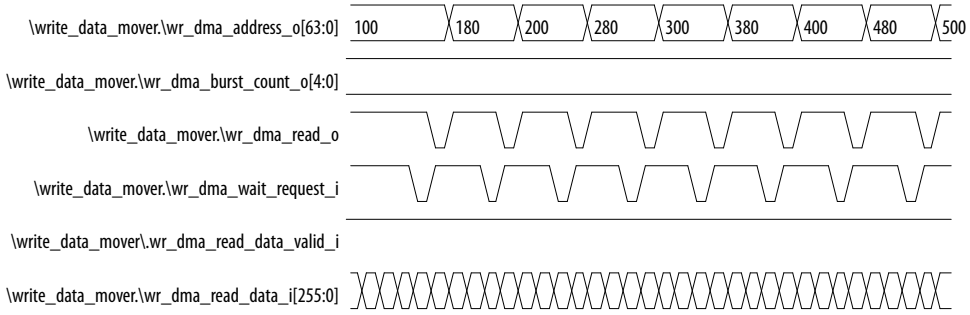
Table 34. DMA Write 256-Bit Avalon-MM Master Interface

Signal Name	Direction	Description
wr_dma_read_o	Output	When asserted, indicates that the Write DMA module is reading data from a memory component in the Avalon-MM address space to write to the PCIe address space.
wr_dma_address_o[63:0]	Output	Specifies the address for the data to be read from a memory component in the Avalon-MM address space .
wr_dma_read_data_i[255:0]	Input	Specifies the completion data that the Write DMA module writes to the PCIe address space.

continued...

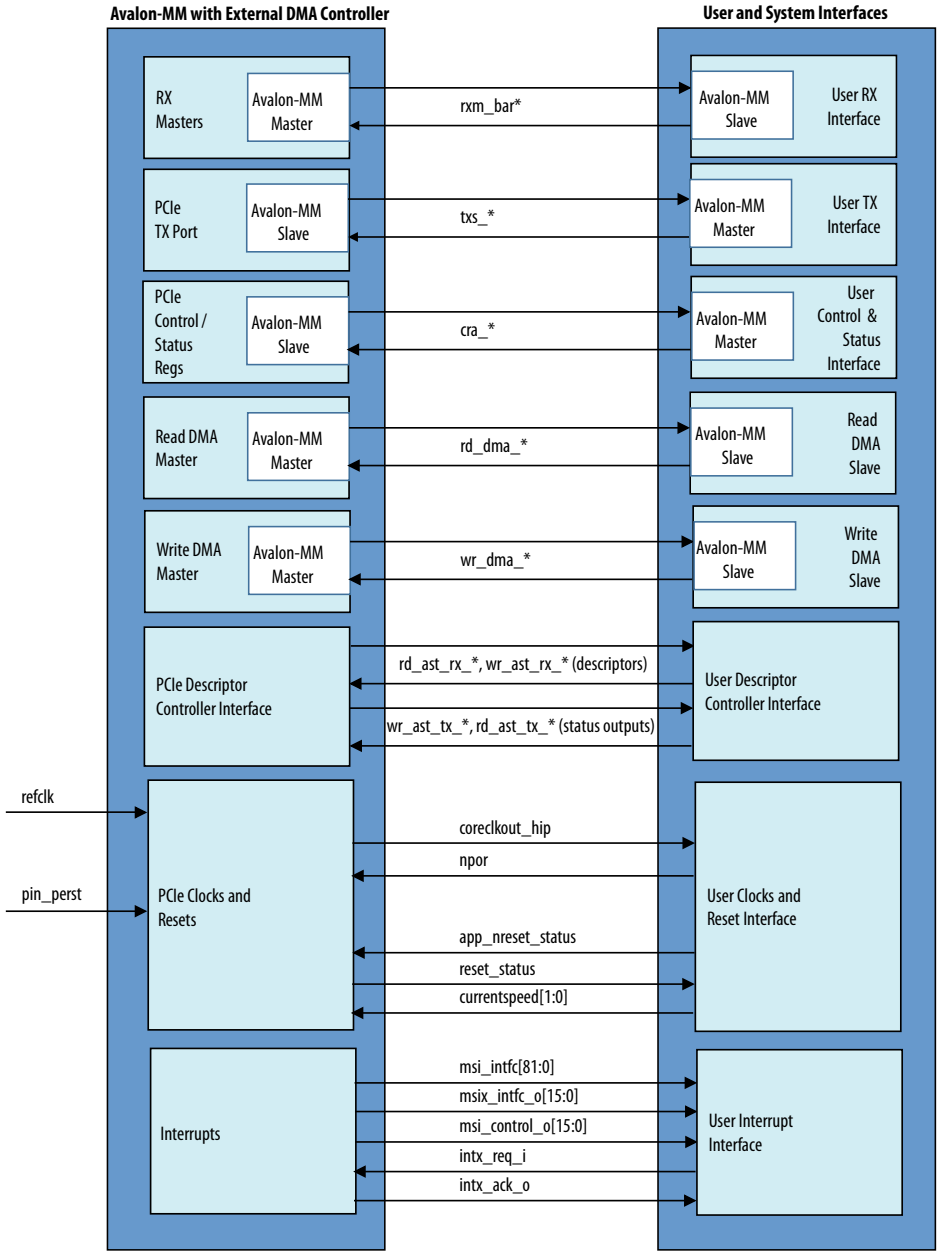
Signal Name	Direction	Description
`wr_dma_burst_count_o[4:0]`	Output	Specifies the burst count in 256-bit words.
`wr_dma_wait_request_i`	Input	When asserted, indicates that the memory is not ready to be read.
`wr_dma_read_data_valid_i`	Input	When asserted, indicates that `wr_dma_read_data_valid_i` is valid.

Figure 40. Write DMA Avalon-MM Master Reads Data from FPGA Memory



6.1.1.3. Descriptor Controller Interfaces when Instantiated Externally

Figure 41. Connections: User Application to Avalon-MM DMA Bridge with External Descriptor Controller



6.1.1.3.1. Avalon-ST Descriptor Source

After fetching multiple descriptor entries from the Descriptor Table in the PCIe system memory, the Descriptor Controller uses its Avalon-ST Descriptor source interface to transfer 160-bit Descriptors to the Read or Write DMA Data Movers.

Table 35. Avalon-ST Descriptor Sink Interface

This interface sends instructions from Descriptor Controller to Read DMA Engine.

Signal Name	Direction	Description
rd_ast_rx_data_i[159:0]	Input	Specifies the descriptors for the Read DMA module. Refer to *DMA Descriptor Format* table below for bit definitions.
rd_ast_rx_valid_i	Input	When asserted, indicates that the data is valid.
rd_ast_rx_ready_o	Output	When asserted, indicates that the Read DMA read module is ready to receive a new descriptor. The ready latency is 3 cycles. Consequently, interface can accept data 3 cycles after ready is asserted.

Table 36. Avalon-ST Descriptor Sink Interface

This interface sends instructions from Descriptor Controller to Write DMA Engine.

Signal Name	Direction	Description
wr_ast_rx_data_i[159:0]	Input	Specifies the descriptors for the Write DMA module. Refer to *DMA Descriptor Format* table below for bit definitions.
wr_ast_rx_valid_i	Input	When asserted, indicates that the data is valid.
wr_ast_rx_ready_o	Output	When asserted, indicates that the Write DMA module engine is ready to receive a new descriptor. The ready latency for this signal is 3 cycles. Consequently, interface can accept data 3 cycles after ready is asserted.

Descriptor Table Format

Descriptor table entries include the source address, the destination address, the size, and the descriptor ID. Each descriptor is padded with zeros to 256-bits (32 bytes) to form an entries in the table.

Table 37. DMA Descriptor Format

Bits	Name	Description
[31:0]	Source Low Address	Low-order 32 bits of the DMA source address. The address boundary must align to the 32 bits so the 2 least significant bits are 2'b00. For the Read Data Mover module, the source address is the PCIe domain address. For the Write Data Mover module, the source address is the Avalon-MM domain address.
[63:32]	Source High Address	High-order 32 bits of the source address.
[95:64]	Destination Low Address	Low-order 32 bits of the DMA destination address. The address boundary must align to the 32 bits so the 2 least significant bits have the value of 2'b00. For the Read Data Mover module, the destination address is the Avalon-MM domain address. For the Write Data Mover module, the destination address is the PCIe domain address.
[127:96]	Destination High Address	High-order 32 bits of the destination address.
[145:128]	DMA Length	Specifies the number of dwords to transfer. The length must be greater than 0. The maximum length is 1 MB - 4 bytes.
[153:146]	DMA Descriptor ID	Unique 7-bit ID for the descriptor. Status information returns with the same ID.
[159:154]	Reserved	—

Avalon-ST Descriptor Status Sources

Read Data Mover and Write Data Mover modules report status to the Descriptor Controller on the `rd_dma_tx_data_o[31:0]` or `wr_dma_tx_data_o[31:0]` bus when a descriptor completes successfully.

The following table shows the mappings of the triggering events to the DMA descriptor status bus:

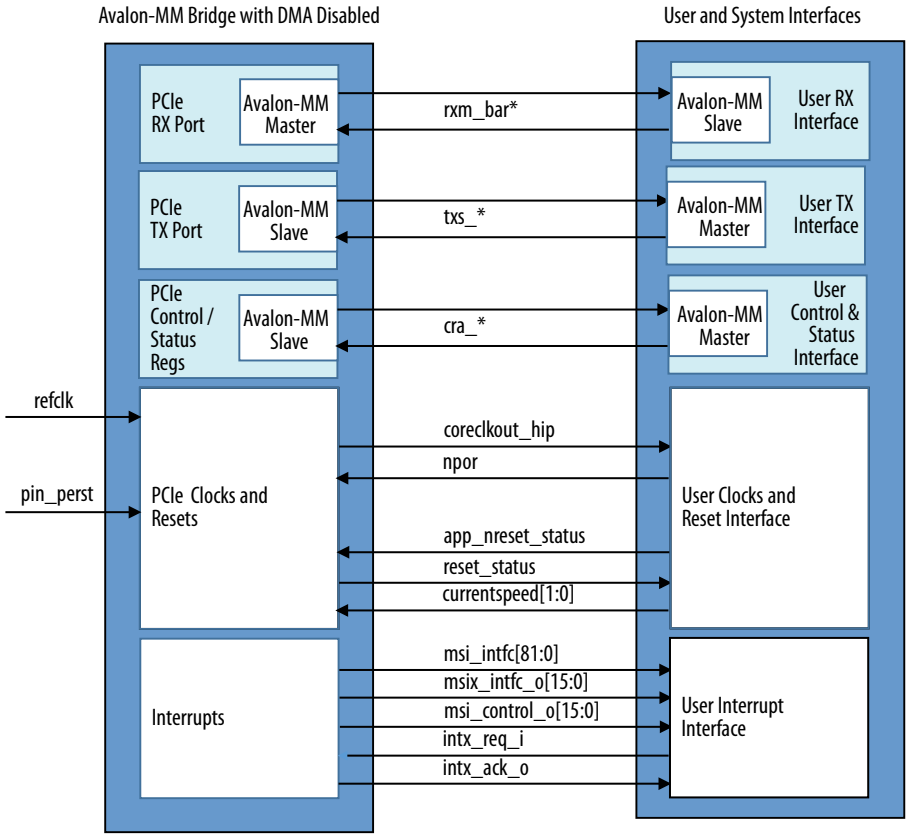
Table 38. DMA Status Bus

Bits	Name	Description
[31:9]	Reserved	—
[8]	`Done`	When asserted, a single DMA descriptor has completed successfully.
[7:0]	`Descriptor ID`	The ID of the descriptor whose status is being reported.

6.1.2. Avalon-MM Interface to the Application Layer

This section describes the top-level interfaces available when the Avalon-MM Intel Stratix 10 Hard IP for PCI Express does not use the DMA functionality.

Figure 42. Connections: User Application to Avalon-MM DMA Bridge with DMA Descriptor Controller Disabled



6.1.2.1. Bursting and Non-Bursting Avalon-MM Module Signals

The Avalon-MM Master module translates read and write TLPs received from the PCIe link to Avalon-MM transactions for connected slaves. You can enable up to six Avalon-MM Master interfaces. One of the six Base Address Registers (BARs) define the base address for each master interface. This module allows other PCIe components, including host software, to access the Avalon-MM slaves connected in the Platform Designer.

The **Enable burst capability for Avalon-MM Bar0-5 Master Port** parameter on the **Base address register** tab determines the type of Avalon-MM master to use for each BAR. Two types are available:

- A high performance, 256-bit master with burst support. This type supports high bandwidth data transfers.
- A non-bursting 32-bit master with byte level byte enables. This type supports for access to control and status registers.

Table 39. Avalon-MM RX Master Interface Signals

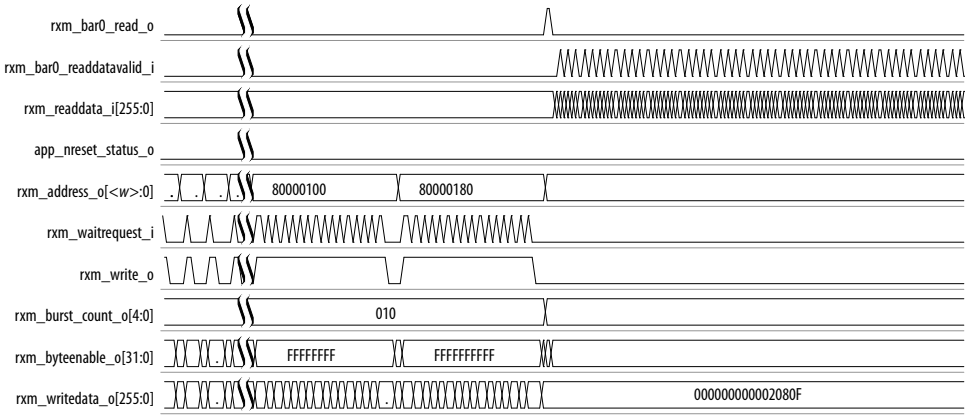
<*n*> = the BAR number, and can be 0, 1, 2, 3, 4, or 5.

Signal Name	Direction	Description
rxm_bar<*n*>_write_o	Output	Asserted by the core to request a write to an Avalon-MM slave.
rxm_bar<*n*>_address_o[<*W*>-1:0]	Output	The address of the Avalon-MM slave being accessed.
rxm_bar<*n*>_writedata_o[255:0]	Output	RX data being written to slave
rxm_bar<*n*>_byteenable_o[31:0]	Output	Dword enables for write data.
rxm_bar<*n*>_burstcount_o[4:0] (available in burst mode only)	Output	The burst count, measured in 256-bit words of the RX write or read request. The maximum data in a burst is 512 bytes. This optional signal is available only when you turn on **Enable burst capability for RXM Avalon-MM BAR<*n*> Master ports.**
rxm_bar<*n*>_waitrequest_i	Input	Asserted by the external Avalon-MM slave to hold data transfer.
rxm_bar<*n*>_read_o	Output	Asserted by the core to request a read.
rxm_bar<*n*>_readdata_i[255:0]	Input	Read data returned from Avalon-MM slave in response to a read request. This data is sent to the IP core through the TX interface.
rxm_bar<*n*>_readdatavalid_i	Input	Asserted by the system interconnect fabric to indicate that the read data is valid.
rxm_irq_i[<*m*>:0], <*m*> < 16	Input	Connect interrupts to the Avalon-MM interface. These signals are only available for the Avalon-MM when the CRA port is enabled. A rising edge triggers an MSI interrupt. The hard IP core converts this event to an MSI interrupt and sends it to the Root Port. The host reads the `Interrupt Status` register to retrieve the interrupt vector. Host software services the interrupt and notifies the target upon completion. As many as 16 individual interrupt signals (<*m*>≤15) are available. If rxm_irq_<n>[<*m*>:0] is asserted on consecutive cycles without the deassertion of all interrupt inputs, no MSI message is sent for subsequent interrupts. To avoid losing interrupts, software must ensure that all interrupt sources are cleared for each MSI message received. *Note:* These signals are not available when the IP core is operating in DMA mode (i.e. when the **Enable Avalon-MM DMA** option in the **Avalon-MM Settings** tab of the GUI is set to **On**).

The following timing diagram illustrates the RX master port propagating requests to the Application Layer and also shows simultaneous read and write activities.

Figure 43. Simultaneous RXM Read and RXM Write

Figure 44. RX Master Interface



6.1.2.2. Non-Bursing Slave Module

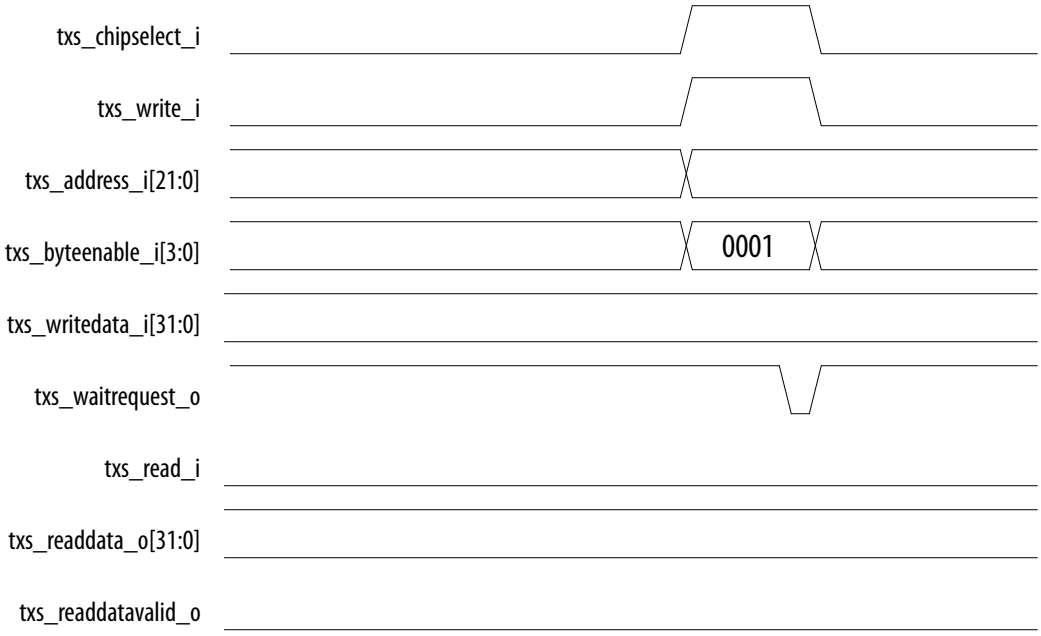
The TX Slave module translates Avalon-MM read and write requests to PCI Express TLPs.

The slave module supports a single outstanding non-bursting request. It typically sends status updates to the host. This is a 32-bit Avalon-MM slave interface.

Table 40. TX Slave Control

Signal Name	Direction	Description
`txs_chipselect_i`	Input	When asserted, indicates that this slave interface is selected. When `txs_chipselect_i` is deasserted, `txs_read_i` and `txs_write_i` signals are ignored.
`txs_read_i`	Input	When asserted, specifies a Avalon-MM Ignored when the chip select is deasserted.
`txs_write_i`	Input	When asserted, specifies a Avalon-MM Ignored when the chip select is deasserted.
`txs_writedata_i[31:0]`	Input	Specifies the Avalon-MM data for a write command.
`txs_address_i[<w>-1:0]`	Input	Specifies the Avalon-MM byte address for the read or write command. The width of this address bus is specified by the parameter **Address width of accessible PCIe memory space**.
`txs_byteenable_i[3:0]`	Input	Specifies the valid bytes for a write command.
`txs_readdata_o[31:0]`	Output	Drives the read completion data.
`txs_readdatavalid_o`	Output	When asserted, indicates that read data is valid.
`txs_waitrequest_o`	Output	When asserted, indicates that the Avalon-MM slave port is not ready to respond to a read or write request. The non-bursting Avalon-MM slave may assert`txs_waitrequest_o` during idle cycles. An Avalon-MM master may initiate a transaction when `txs_waitrequest_o` is asserted and wait for that signal to be deasserted.

Figure 45. Non-Bursting Slave Interface Sends Status to Host



6.1.2.3. 32-Bit Control Register Access (CRA) Slave Signals

The CRA interface provides access to the control and status registers of the Avalon-MM bridge. This interface has the following properties:

- 32-bit data bus
- Supports a single transaction at a time
- Supports single-cycle transactions (no bursting)

Note: When the Avalon-MM Hard IP for PCIe IP Core is in Root Port mode, and the application logic issues a CfgWr or CfgRd via the CRA interface, it needs to fill the Tag field in the TLP Header with the value 0x10 to ensure that the corresponding Completion gets routed to the CRA interface correctly. If the application logic sets the Tag field to some other value, the Avalon-MM Hard IP for PCIe IP Core does not overwrite that value with the correct value.

Table 41. Avalon-MM CRA Slave Interface Signals

Signal Name	Direction	Description
`cra_read_i`	Input	Read enable.
`cra_write_i`	Input	Write request.
`cra_address_i[14:0]`	Input	
`cra_writedata_i[31:0]`	Input	Write data. The current version of the CRA slave interface is read-only. Including this signal as part of the Avalon-MM interface, makes future enhancements possible.
`cra_readdata[31:0]`	Output	Read data lines.
`cra_byteenable_i[3:0]`	Input	Byte enable.

continued...

Signal Name	Direction	Description
cra_waitrequest_o	Output	Wait request to hold off additional requests.
cra_chipselect_i	Input	Chip select signal to this slave.
cra_irq_o	Output	Interrupt request. A port request for an Avalon-MM interrupt.

6.1.2.4. Bursting Slave Module

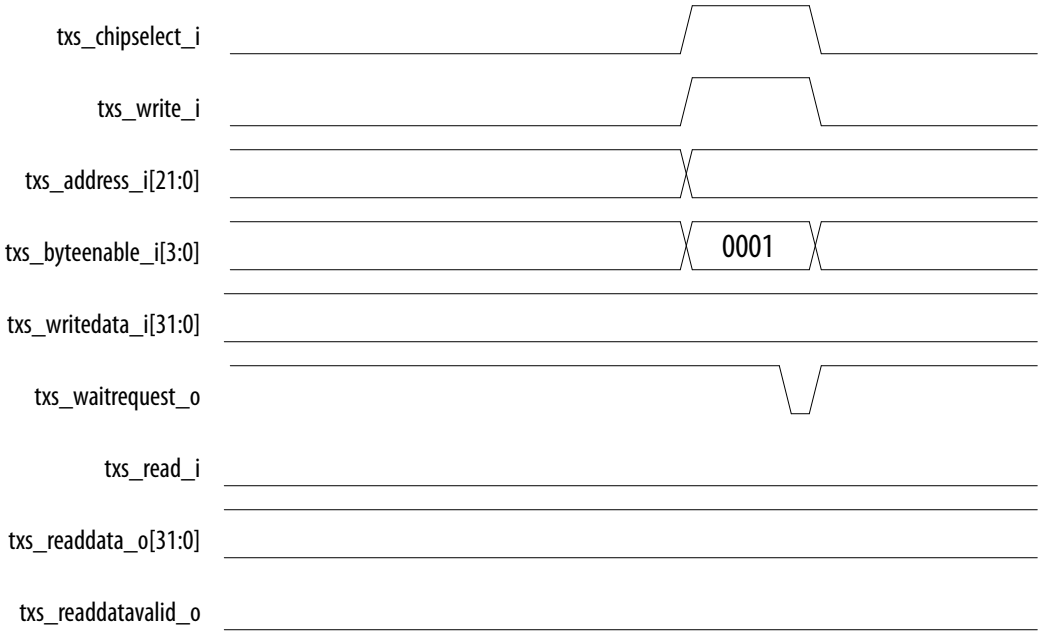
The TX Slave module translates Avalon-MM read and write requests to bursting PCI Express TLPs.

The slave module supports a single outstanding non-bursting request. It typically sends status updates to the host. This is a 32-bit Avalon-MM slave interface.

Table 42. TX Slave Control

Signal Name	Direction	Description
hptxs_read_i	Input	When asserted, specifies an Avalon-MM slave.
hptxs_write_i	Input	When asserted, specifies an Avalon-MM slave.
hptxs_writedata_i[31:0]	Input	Specifies the Avalon-MM data for a write command.
hptxs_address_i[<*w*>-1:0]	Input	Specifies the Avalon-MM byte address for the read or write command. The width of this address bus is specified by the parameter **Address width of accessible PCIe memory space (HPTXS)**. <*w*> <= 63.
hptxs_byteenable_i[31:0]	Input	Specifies the valid dwords for a write command.
hptxs_readdata_o[255:0]	Output	Drives the read completion data.
hptxs_readdatavalid_o	Output	When asserted, indicates that read data is valid.
hptxs_waitrequest_o	Output	When asserted, indicates that the Avalon-MM slave port is not ready to respond to a read or write request. The non-bursting Avalon-MM slave may assert`hptxs_waitrequest_o` during idle cycles. An Avalon-MM master may initiate a transaction when `hptxs_waitrequest_o` is asserted and wait for that signal to be deasserted.

Figure 46. Non-Bursting Slave Interface Sends Status to Host



6.1.3. Clocks and Reset

6.1.3.1. Clocks

Table 43. Clocks

Signal	Direction	Description
refclk	Input	This is the input reference clock for the IP core as defined by the *PCI Express Card Electromechanical Specification Revision 2.0*. The frequency is 100 MHz ±300 ppm. To meet the PCIe 100 ms wake-up time requirement, this clock must be free-running. *Note:* This input reference clock must be stable and free-running at device power-up for a successful device configuration.
coreclkout_hip	Output	This clock drives the Data Link, Transaction, and Application Layers. For the Application Layer, the frequency depends on the data rate and the number of lanes as specified in the table

Data Rate	coreclkout_hip Frequency
Gen1 x1, x2, x4, x8, and x16	125 MHz
Gen2 x1, x2, x4, and x8,	125 MHz
Gen2 x16	250 MHz
Gen3 x1, x2, and x4	125 MHz
Gen3 x8	250 MHz

6.1.3.2. Resets

The PCIe Hard IP generates the reset signal. The Avalon-MM DMA bridge has a single, active low reset input. It is a synchronized version of the reset from the PCIe IP core.

Figure 47. Clock and Reset Connections

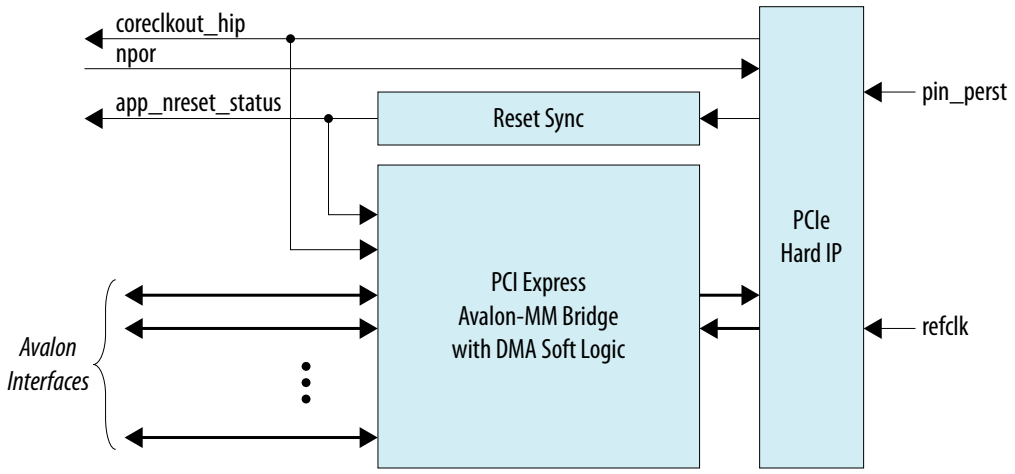


Table 44. Resets

Signal	Direction	Description
app_nreset_status	Output	This is active low reset signal. It is derived from npor or pin_perstn. You can use this signal to reset the Application.
currentspeed[1:0]	Output	Indicates the current speed of the PCIe link. The following encodings are defined: • 2'b00: Undefined • 2'b01: Gen1 • 2'b10: Gen2 • 2'b11: Gen3
npor	Input	The Application Layer drives this active low reset signal. npor resets the entire IP core, PCS, PMA, and PLLs. npor should be held for a minimum of 20 ns. This signal is edge, not level sensitive; consequently, a low value on this signal does not hold custom logic in reset. This signal cannot be disabled.
pin_perst	Input	Active low reset from the PCIe reset pin of the device. Resets the datapath and control registers.

6.1.4. Interrupts

6.1.4.1. MSI Interrupts for Endpoints

The Stratix 10 PCIe Avalon-MM Bridge with DMA does not generate an MSI to signal events. However, the Application can cause an MSI to be sent by the non-bursting Avalon-MM TX slave by performing a memory write to the non-bursting Avalon-MM TX slave.

After the host receives an MSI it can service the interrupt based on the application-defined interrupt service routine. This mechanism allows host software to avoid continuous polling of the status table done bits. This interface provides the required information for users to form the MSI/MSI-X via the TXS interface.

Table 45. MSI Interrupt

Signal	Direction	Description
msi_intfc[81:0]	Output	This bus provides the following MSI address, data, and enabled signals: • msi_intfc_o[81]: Master enable • msi_intfc_o[80]: MSI enable • msi_intfc_o[79:64]: MSI data • msi_intfc_o[63:0]: MSI address
msix_intfc_o[15:0]	Output	Provides for system software control of MSI-X as defined in Section 6.8.2.3 *Message Control for MSI-X* in the *PCI Local Bus Specification, Rev. 3.0*. The following fields are defined: • msix_intfc_o[15]: Enable • msix_intfc_o[14]: Mask • msix_intfc_o[13:11]: Reserved • msix_intfc_o[10:0]: Table size
msi_control_o[15:0]	Output	Provides system software control of the MSI messages as defined in Section 6.8.1.3 *Message Control for MSI* in the *PCI Local Bus Specification, Rev. 3.0*. The following fields are defined: • msi_control_o[15:9]: Reserved • msi_control_o[8]: Per-Vector Masking Capable • msi_control_o[7]: 64-Bit Address Capable • msi_control_o[6:4]: Multiple Message Enable • msi_control_o[3:1]: MSI Message Capable • msi_control_o[0]: MSI Enable
intx_req_i	Input	Legacy interrupt request.

6.1.4.2. Legacy Interrupts

Stratix 10 PCIe Avalon-MM Bridge with DMA can generate PCIe legacy interrupt when `Interrupt Disable` bit 10 of `Command` register in Configuration Header is set to zero and `MSI Enable` bit of `MSI Message Control` register is set to zero.

Table 46. MSI Interrupt

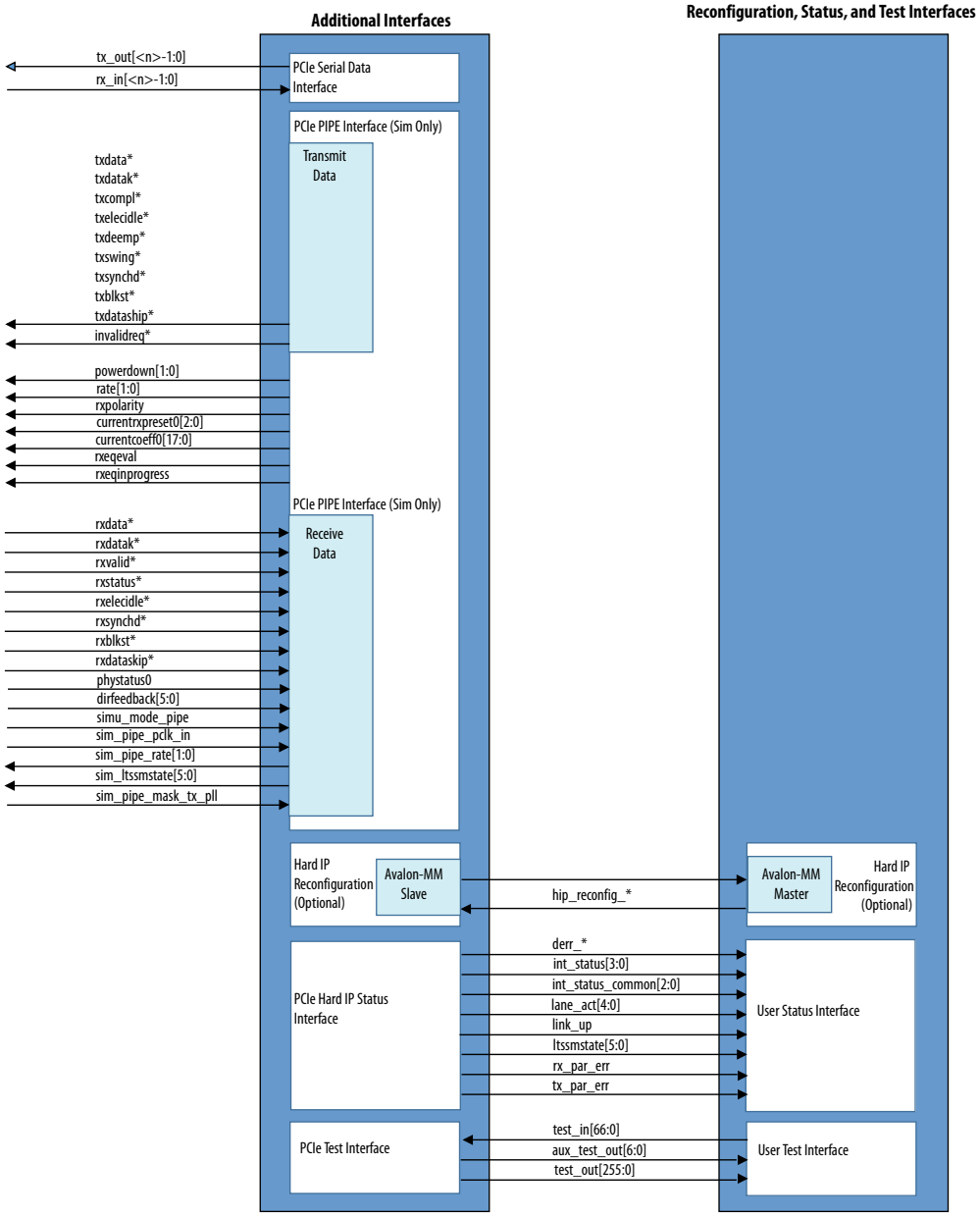
Signal	Direction	Description
intx_req_i	Input	Legacy interrupt request.

6.1.5. Flush Requests

In the PCI Express protocol, a memory read request from the host with a length of 1 dword and byte enables being all 0's translate to a flush request for the Completer, which in this case is the Intel Stratix 10 Avalon-MM Hard IP for PCIe. However, this flush request feature is not supported by the Avalon-MM Hard IP for PCIe.

6.1.6. Serial Data, PIPE, Status, Reconfiguration, and Test Interfaces

Figure 48. Connections: Serial Data, PIPE, Status, Reconfiguration, and Test Interfaces



6.1.6.1. Serial Data Interface

The IP core supports 1, 2, 4, 8, or 16 lanes.

Table 47. Serial Data Interface

Signal	Direction	Description
tx_out[<*n-1*>:0]	Output	Transmit serial data output.
rx_in[<*n-1*>:0]	Input	Receive serial data input.

6.1.6.2. PIPE Interface

The Stratix 10 PIPE interface compiles with the *PHY Interface for the PCI Express Architecture PCI Express 3.0* specification.

Table 48. PIPE Interface

Signal	Direction	Description
txdata[31:0]	Output	Transmit data.
txdatak[3:0]	Output	Transmit data control character indication.
txcompl	Output	Transmit compliance. This signal drives the TX compliance pattern. It forces the running disparity to negative in Compliance Mode (negative COM character).
txelecidle	Output	Transmit electrical idle. This signal forces the tx_out<*n*> outputs to electrical idle.
txdetectrx	Output	Transmit detect receive. This signal tells the PHY layer to start a receive detection operation or to begin loopback.
powerdown[1:0]	Output	Power down. This signal requests the PHY to change the power state to the specified state (P0, P0s, P1, or P2).
txmargin[2:0]	Output	Transmit V_{OD} margin selection. The value for this signal is based on the value from the Link Control 2 Register.
txdeemp	Output	Transmit de-emphasis selection. The Intel Stratix 10 Hard IP for PCI Express sets the value for this signal based on the indication received from the other end of the link during the Training Sequences (TS). You do not need to change this value.
txswing	Output	When asserted, indicates full swing for the transmitter voltage. When deasserted indicates half swing.
txsynchd[1:0]	Output	For Gen3 operation, specifies the receive block type. The following encodings are defined: • 2'b01: Ordered Set Block • 2'b10: Data Block Designs that do not support Gen3 can ground this signal.
txblkst[3:0]	Output	For Gen3 operation, indicates the start of a block in the transmit direction. pipe spec
txdataskip	Output	For Gen3 operation. Allows the MAC to instruct the TX interface to ignore the TX data interface for one clock cycle. The following encodings are defined: • 1'b0: TX data is invalid • 1'b1: TX data is valid
rate[1:0]	Output	The 2-bit encodings have the following meanings: • 2'b00: Gen1 rate (2.5 Gbps) • 2'b01: Gen2 rate (5.0 Gbps) • 2'b1X: Gen3 rate (8.0 Gbps)
rxpolarity	Output	Receive polarity. This signal instructs the PHY layer to invert the polarity of the 8B/10B receiver decoding block.

continued...

Signal	Direction	Description
currentrxpreset[2:0]	Output	For Gen3 designs, specifies the current preset.
currentcoeff[17:0]	Output	For Gen3, specifies the coefficients to be used by the transmitter. The 18 bits specify the following coefficients: • [5:0]: C_{-1} • [11:6]: C_0 • [17:12]: C_{+1}
rxeqeval	Output	For Gen3, the PHY asserts this signal when it begins evaluation of the transmitter equalization settings. The PHY asserts Phystatus when it completes the evaluation. The PHY deasserts rxeqeval to abort evaluation. Refer to the figure below for a timing diagram illustrating this process.
rxeqinprogress	Output	For Gen3, the PHY asserts this signal when it begins link training. The PHY latches the initial coefficients from the link partner. Refer to the figure below for a timing diagram illustrating this process.
invalidreq	Output	For Gen3, indicates that the Link Evaluation feedback requested a TX equalization setting that is out-of-range. The PHY asserts this signal continually until the next time it asserts rxeqeval.
rxdata[31:0]	Input	Receive data control. Bit 0 corresponds to the lowest-order byte of rxdata, and so on. A value of 0 indicates a data byte. A value of 1 indicates a control byte. For Gen1 and Gen2 only.
rxdatak[3:0]	Input	Receive data control. This bus receives data on lane. Bit 0 corresponds to the lowest-order byte of rxdata, and so on. A value of 0 indicates a data byte. A value of 1 indicates a control byte. For Gen1 and Gen2 only.
phystatus	Input	PHY status. This signal communicates completion of several PHY requests. pipe spec
rxvalid	Input	Receive valid. This signal indicates symbol lock and valid data on rxdata and rxdatak.
rxstatus[2:0]	Input	Receive status. This signal encodes receive status, including error codes for the receive data stream and receiver detection.
rxelecidle	Input	Receive electrical idle. When asserted, indicates detection of an electrical idle. pipe spec
rxsynchd[3:0]	Input	For Gen3 operation, specifies the receive block type. The following encodings are defined: • 2'b01: Ordered Set Block • 2'b10: Data Block Designs that do not support Gen3 can ground this signal.
rxblkst[3:0]	Input	For Gen3 operation, indicates the start of a block in the receive direction.
rxdataskip	Input	For Gen3 operation. Allows the PCS to instruct the RX interface to ignore the RX data interface for one clock cycle. The following encodings are defined: • 1'b0: RX data is invalid • 1'b1: RX data is valid
dirfeedback[5:0]	Input	For Gen3, provides a Figure of Merit for link evaluation for H tile transceivers. The feedback applies to the following coefficients: • dirfeedback[5:4]: Feedback applies to C_{+1} • dirfeedback[3:2]: Feedback applies to C_0 • dirfeedback[1:0]: Feedback applies to C_{-1}

continued...

Signal	Direction	Description
		The following feedback encodings are defined: • 2'b00: No change • 2'b01: Increment • 2'b10: Decrement • 2/b11: Reserved Refer to the figure below for a timing diagram illustrating this process.
simu_mode_pipe	Input	When set to 1, the PIPE interface is in simulation mode.
sim_pipe_pclk_in	Input	This clock is used for PIPE simulation only, and is derived from the refclk. It is the PIPE interface clock used for PIPE mode simulation.
sim_pipe_rate[1:0]	Output	The 2-bit encodings have the following meanings: • 2'b00: Gen1 rate (2.5 Gbps) • 2'b01: Gen2 rate (5.0 Gbps) • 2'b10: Gen3 rate (8.0 Gbps)
sim_ltssmstate[5:0]	Output	LTSSM state: The following encodings are defined: • 6'h00 - Detect.Quiet • 6'h01 - Detect.Active • 6'h02 - Polling.Active • 6'h03 - Polling.Compliance • 6'h04 - Polling.Configuration • 6'h05 - PreDetect.Quiet • 6'h06 - Detect.Wait • 6'h07 - Configuration.Linkwidth.Start • 6'h08 - Configuration.Linkwidth.Accept • 6'h09 - Configuration.Lanenum.Wait • 6'h0A - Configuration.Lanenum.Accept • 6'h0B - Configuration.Complete • 6'h0C - Configuration.Idle • 6'h0D - Recovery.RcvrLock • 6'h0E - Recovery.Speed • 6'h0F - Recovery.RcvrCfg • 6'h10 - Recovery.Idle • 6'h20 - Recovery.Equalization Phase 0 • 6'h21 - Recovery.Equalization Phase 1 • 6'h22 - Recovery.Equalization Phase 2 • 6'h23 - Recovery.Equalization Phase 3 • 6'h11 - L0 • 6'h12 - L0s • 6'h13 - L123.SendEIdle • 6'h14 - L1.Idle • 6'h15 - L2.Idle • 6'h16 - L2.TransmitWake • 6'h17 - Disabled.Entry • 6'h18 - Disabled.Idle • 6'h19 - Disabled • 6'h1A - Loopback.Entry • 6'h1B - Loopback.Active • 6'h1C - Loopback.Exit • 6'h1D - Loopback.Exit.Timeout • 6'h1E - HotReset.Entry • 6'h1F - Hot.Reset
sim_pipe_mask_tx_pll_lock	Input	Should be active during rate change. This signal Is used to mask the PLL lock signals. This interface is used only for PIPE simulations.

Signal	Direction	Description
		In serial simulations, The Endpoint PHY drives this signal. For PIPE simulations, in the Intel testbench, The PIPE BFM drives this signal.

Related Information

PHY Interface for the PCI Express Architecture PCI Express 3.0

6.1.6.3. Hard IP Status Interface

Hard IP Status: This optional interface includes the following signals that are useful for debugging, including: link status signals, interrupt status signals, TX and RX parity error signals, correctable and uncorrectable error signals.

Table 49. Hard IP Status Interface

Signal	Direction	Description
derr_cor_ext_rcv	Output	When asserted, indicates that the RX buffer detected a 1-bit (correctable) ECC error. This is a pulse stretched output.
derr_cor_ext_rpl	Output	When asserted, indicates that the retry buffer detected a 1-bit (correctable) ECC error. This is a pulse stretched output.
derr_rpl	Output	When asserted, indicates that the retry buffer detected a 2-bit (uncorrectable) ECC error. This is a pulse stretched output.
derr_uncor_ext_rcv	Output	When asserted, indicates that the RX buffer detected a 2-bit (uncorrectable) ECC error. This is a pulse stretched output.
int_status[10:0](H-Tile) int_status[7:0] (L-Tile) int_status_pf1[7:0] (L-Tile)	Output	The int_status[3:0] signals drive legacy interrupts to the application (for H-Tile). The int_status[10:4] signals provide status for other interrupts (for H-Tile). The int_status[3:0] signals drive legacy interrupts to the application for PF0 (for L-Tile). The int_status[7:4] signals provide status for other interrupts for PF0 (for L-Tile). The int_status_pf1[3:0] signals drive legacy interrupts to the application for PF1 (for L-Tile). The int_status_pf1[7:4] signals provide status for other interrupts for PF1 (for L-Tile). The following signals are defined: • int_status[0]: Interrupt signal A • int_status[1]: Interrupt signal B • int_status[2]: Interrupt signal C • int_status[3]: Interrupt signal D • int_status[4]: Specifies a Root Port AER error interrupt. This bit is set when the cfg_aer_rc_err_msi or cfg_aer_rc_err_int signal asserts. This bit is cleared when software writes 1 to the register bit or when cfg_aer_rc_err_int is deasserts. • int_status[5]: Specifies the Root Port PME interrupt status. It is set when cfg_pme_msi or cfg_pme_int asserts. It is cleared when software writes a 1 to clear or when cfg_pme_int deasserts. • int_status[6]: Asserted when a hot plug event occurs and Power Management Events (PME) are enabled. (PMEs are typically used to revive the system or a function from a low power state.) • int_status[7]: Specifies the hot plug event interrupt status.

continued...

Signal	Direction	Description
		• int_status[8]: Specifies the interrupt status for the Link Autonomous Bandwidth Status register. H-Tile only. • int_status[9]: Specifies the interrupt status for the Link Bandwidth Management Status register. H-Tile only. • int_status[10]: Specifies the interrupt status for the Link Equalization Request bit in the Link Status register. H-Tile only.
int_status_common[2:0]	Output	Specifies the interrupt status for the following registers. When asserted, indicates that an interrupt is pending: • int_status_common[0]: Autonomous bandwidth status register. • int_status_common[1]: Bandwidth management status register. • int_status_common[2]: Link equalization request bit in the link status register.
lane_act[4:0]	Output	Lane Active Mode: This signal indicates the number of lanes that configured during link training. The following encodings are defined: • 5'b0 0001: 1 lane • 5'b0 0010: 2 lanes • 5'b0 0100: 4 lanes • 5'b0 1000: 8 lanes • 5'b1 0000: 16 lanes
link_up	Output	When asserted, the link is up.
ltssmstate[5:0]	Output	Link Training and Status State Machine (LTSSM) state: The LTSSM state machine encoding defines the following states: • 6'h00 - Detect.Quiet • 6'h01 - Detect.Active • 6'h02 - Polling.Active • 6'h03 - Polling.Compliance • 6'h04 - Polling.Configuration • 6'h05 - PreDetect.Quiet • 6'h06 - Detect.Wait • 6'h07 - Configuration.Linkwidth.Start • 6'h08 - Configuration.Linkwidth.Accept • 6'h09 - Configuration.Lanenum.Wait • 6'h0A - Configuration.Lanenum.Accept • 6'h0B - Configuration.Complete • 6'h0C - Configuration.Idle • 6'h0D - Recovery.RcvrLock • 6'h0E - Recovery.Speed • 6'h0F - Recovery.RcvrCfg • 6'h10 - Recovery.Idle • 6'h20 - Recovery.Equalization Phase 0 • 6'h21 - Recovery.Equalization Phase 1 • 6'h22 - Recovery.Equalization Phase 2 • 6'h23 - Recovery.Equalization Phase 3 • 6'h11 - L0 • 6'h12 - L0s • 6'h13 - L123.SendEIdle • 6'h14 - L1.Idle • 6'h15 - L2.Idle • 6'h16 - L2.TransmitWake • 6'h17 - Disabled.Entry • 6'h18 - Disabled.Idle • 6'h19 - Disabled

continued...

Signal	Direction	Description
		• 6'h1A - Loopback.Entry • 6'h1B - Loopback.Active • 6'h1C - Loopback.Exit • 6'h1D - Loopback.Exit.Timeout • 6'h1E - HotReset.Entry • 6'h1F - Hot.Reset
rx_par_err	Output	Asserted for a single cycle to indicate that a parity error was detected in a TLP at the input of the RX buffer. This error is logged as an uncorrectable internal error in the VSEC registers. For more information, refer to *Uncorrectable Internal Error Status Register*. If this error occurs, you must reset the Hard IP because parity errors can leave the Hard IP in an unknown state.
tx_par_err	Output	Asserted for a single cycle to indicate a parity error during TX TLP transmission. The IP core transmits TX TLP packets even when a parity error is detected.

Related Information

Uncorrectable Internal Error Status Register on page 92

6.1.6.4. Hard IP Reconfiguration

The Hard IP reconfiguration interface is an Avalon-MM slave interface with a 21-bit address and an 8-bit data bus. You can use this bus to dynamically modify the value of configuration registers that are read-only at run time.Note that after a warm reset or cold reset, changes made to the configuration registers of the Hard IP via the Hard IP reconfiguration interface are lost as these registers revert back to their default values.

Table 50. Hard IP Reconfiguration Signals

Signal	Direction	Description
hip_reconfig_clk	Input	Reconfiguration clock. The frequency range for this clock is 100–125 MHz.
hip_reconfig_rst_n	Input	Active-low Avalon-MM reset for this interface.
hip_reconfig_address[20:0]	Input	The 21-bit reconfiguration address. When the Hard IP reconfiguration feature is enabled, the hip_reconfig_address[20:0] bits are programmable. Some bits have the same functions in both H-Tile and L-Tile: • hip_reconfig_address[11:0]: Provide full byte access to the 4 Kbytes PCIe configuration space. *Note:* For the address map of the PCIe configuration space, refer to the *Configuration Space Registers* section in the *Registers* chapter. • hip_reconfig_address[20]: Should be set to 1'b1 to indicate PCIe space access. Some bits have different functions in H-Tile versus L-Tile: For H-Tile: • hip_reconfig_address[13:12]: Provide the PF number. Since the H-Tile can support up to four PFs, two bits are needed to encode the PF number. • hip_reconfig_address[19:14]: Reserved. They must be driven to 0.

continued...

Signal	Direction	Description
		For L-Tile: • `hip_reconfig_address[12]`: Provides the PF number. Since the L-Tile only supports up to two PFs, one bit is sufficient to encode the PF number. • `hip_reconfig_address[19:13]`: Reserved. They must be driven to 0.
`hip_reconfig_read`	Input	Read signal. This interface is not pipelined. You must wait for the return of the `hip_reconfig_readdata[7:0]` from the current read before starting another read operation.
`hip_reconfig_readdata[7:0]`	Output	8-bit read data. `hip_reconfig_readdata[7:0]` is valid on the third cycle after the assertion of `hip_reconfig_read`.
`hip_reconfig_readdatavalid`	Output	When asserted, the data on `hip_reconfig_readdata[7:0]` is valid.
`hip_reconfig_write`	Input	Write signal.
`hip_reconfig_writedata[7:0]`	Input	8-bit write model.
`hip_reconfig_waitrequest`	Output	When asserted, indicates that the IP core is not ready to respond to a request.

Related Information

Configuration Space Registers on page 83

6.1.6.5. Test Interface

The 256-bit test output interface is available only for x16 simulations. For x1, x2, x4, and x8 variants a 7-bit auxiliary test bus is available.

Signal	Direction	Description
`test_in[66:0]`	Input	This is a multiplexer to select the `test_out[255:0]` and `aux_test_out[6:0]` buses. Driven from channels 8-15. The following encodings are defined: • [66]: Reserved • [65:58]: The multiplexor selects the EMIB adaptor • [57:50]: The multiplexor selects configuration block • [49:48]: The multiplexor selects clocks • [47:46]: The multiplexor selects equalization • [45:44]: The multiplexor selects miscellaneous functionality • [43:42]: The multiplexor selects the PIPE adaptor • [41:40]: The multiplexor selects for CvP. • [39:31]: The multiplexor selects channels 7-1, `aux_test_out[6:0]` • [30:3]: The multiplexor selects channels 15-8, `test_out[255:0]` • [2]: Results in the inversion of LCRC bit 0. • [1]: Results in the inversion of ECRC bit 0 • [0]: Turns on `diag_fast_link_mode` to speed up simulation.
`test_out[255:0]`	Output	`test_out[255:0]` routes to channels 8-15. Includes diagnostic signals from core, adaptor, clock, configuration block, equalization control, miscellaneous, reset, and pipe_adaptor modules. Available only for x16 variants.

7. Registers

7.1. Configuration Space Registers

Table 51. Correspondence between Configuration Space Capability Structures and the PCIe Base Specification Description

Byte Address	Configuration Space Register	Corresponding Section in PCIe Specification
0x000-0x03C	PCI Header Type 0 Configuration Registers	Type 0 Configuration Space Header
0x040-0x04C	Power Management	PCI Power Management Capability Structure
0x050-0x05C	MSI Capability Structure	MSI Capability Structure, see also and *PCI Local Bus Specification*
0x060-0x06C	Reserved	N/A
0x070-0x0A8	PCI Express Capability Structure	PCI Express Capability Structure
0x0B0-0x0B8	MSI-X Capability Structure	MSI-X Capability Structure, see also and *PCI Local Bus Specification*
0x0BC-0x0FC	Reserved	N/A
0x100-0x134	Advanced Error Reporting (AER) (for PFs only)	Advanced Error Reporting Capability
0x138-0x184	Reserved	N/A
0x188-0x1B0	Secondary PCI Express Extended Capability Header	PCI Express Extended Capability
0x1B4-0xB7C	Reserved	N/A
0x1B8-0x1F4	SR-IOV Capability Structure	SR-IOV Extended Capability Header in *Single Root I/O Virtualization and Sharing Specification, Rev, 1.1*
0x1F8-0x1D0	Transaction Processing Hints (TPH) Requester Capability	TLP Processing Hints (TPH)
0x1D4-0x280	Reserved	N/A
0x284-0x288	Address Translation Services (ATS) Capability Structure	Address Translation Services Extended Capability (ATS) in *Single Root I/O Virtualization and Sharing Specification, Rev. 1.1*
0xB80-0xBFC	Intel-Specific	Vendor-Specific Header (Header only)
0xC00	Optional Custom Extensions	N/A
0xC00	Optional Custom Extensions	N/A

Table 52. Summary of Configuration Space Register Fields

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
0x000	Device ID, Vendor ID	Type 0 Configuration Space Header
0x004	Status, Command	Type 0 Configuration Space Header
0x008	Class Code, Revision ID	Type 0 Configuration Space Header
0x00C	Header Type, Cache Line Size	Type 0 Configuration Space Header
0x010	Base Address 0	Base Address Registers
0x014	Base Address 1	Base Address Registers
0x018	Base Address 2	Base Address Registers
0x01C	Base Address 3	Base Address Registers
0x020	Base Address 4	Base Address Registers
0x024	Base Address 5	Base Address Registers
0x028	Reserved	N/A
0x02C	Subsystem ID, Subsystem Vendor ID	Type 0 Configuration Space Header
0x030	Reserved	N/A
0x034	Capabilities Pointer	Type 0 Configuration Space Header
0x038	Reserved	N/A
0x03C	Interrupt Pin, Interrupt Line	Type 0 Configuration Space Header
0x040	PME_Support, D1, D2, etc.	PCI Power Management Capability Structure
0x044	PME_en, PME_Status, etc.	Power Management Status and Control Register
0x050	MSI-Message Control, Next Cap Ptr, Capability ID	MSI and MSI-X Capability Structures
0x054	Message Address	MSI and MSI-X Capability Structures
0x058	Message Upper Address	MSI and MSI-X Capability Structures
0x05C	Reserved Message Data	MSI and MSI-X Capability Structures
0x0B0	MSI-X Message Control Next Cap Ptr Capability ID	MSI and MSI-X Capability Structures
0x0B4	MSI-X Table Offset BIR	MSI and MSI-X Capability Structures
0x0B8	Pending Bit Array (PBA) Offset BIR	MSI and MSI-X Capability Structures
0x100	PCI Express Enhanced Capability Header	Advanced Error Reporting Enhanced Capability Header
0x104	Uncorrectable Error Status Register	Uncorrectable Error Status Register
0x108	Uncorrectable Error Mask Register	Uncorrectable Error Mask Register
0x10C	Uncorrectable Error Mask Register	Uncorrectable Error Severity Register
0x110	Correctable Error Status Register	Correctable Error Status Register
0x114	Correctable Error Mask Register	Correctable Error Mask Register
0x118	Advanced Error Capabilities and Control Register	Advanced Error Capabilities and Control Register
0x11C	Header Log Register	Header Log Register
0x12C	Root Error Command	Root Error Command Register

continued...

Byte Address	Hard IP Configuration Space Register	Corresponding Section in PCIe Specification
0x130	Root Error Status	Root Error Status Register
0x134	Error Source Identification Register Correctable Error Source ID Register	Error Source Identification Register
0x188	Next Capability Offset, PCI Express Extended Capability ID	Secondary PCI Express Extended Capability
0x18C	Enable SKP OS, Link Equalization Req, Perform Equalization	Link Control 3 Register
0x190	Lane Error Status Register	Lane Error Status Register
0x194:0x1B0	Lane Equalization Control Register	Lane Equalization Control Register
0xB80	VSEC Capability Header	Vendor-Specific Extended Capability Header
0xB84	VSEC Length, Revision, ID	Vendor-Specific Header
0xB88	Intel Marker	Intel-Specific Registers
0xB8C	JTAG Silicon ID DW0	
0xB90	JTAG Silicon ID DW1	
0xB94	JTAG Silicon ID DW2	
0xB98	JTAG Silicon ID DW3	
0xB9C	User Device and Board Type ID	
0xBA0:0xBAC	Reserved	
0xBB0	General Purpose Control and Status Register	
0xBB4	Uncorrectable Internal Error Status Register	
0xBB8	Uncorrectable Internal Error Mask Register	
0xBBC	Correctable Error Status Register	
0xBC0	Correctable Error Mask Register	
0xBC4:BD8	Reserved	N/A
0xC00	Optional Custom Extensions	N/A

Related Information

- PCI Express Base Specification 3.0
- PCI Local Bus Specification

7.1.1. Register Access Definitions

This document uses the following abbreviations when describing register access.

Table 53. Register Access Abbreviations

Sticky bits are not initialized or modified by hot reset or function-level reset.

Abbreviation	Meaning
RW	Read and write access
RO	Read only
WO	Write only

continued...

Abbreviation	Meaning
RW1C	Read write 1 to clear
RW1CS	Read write 1 to clear sticky
RWS	Read write sticky

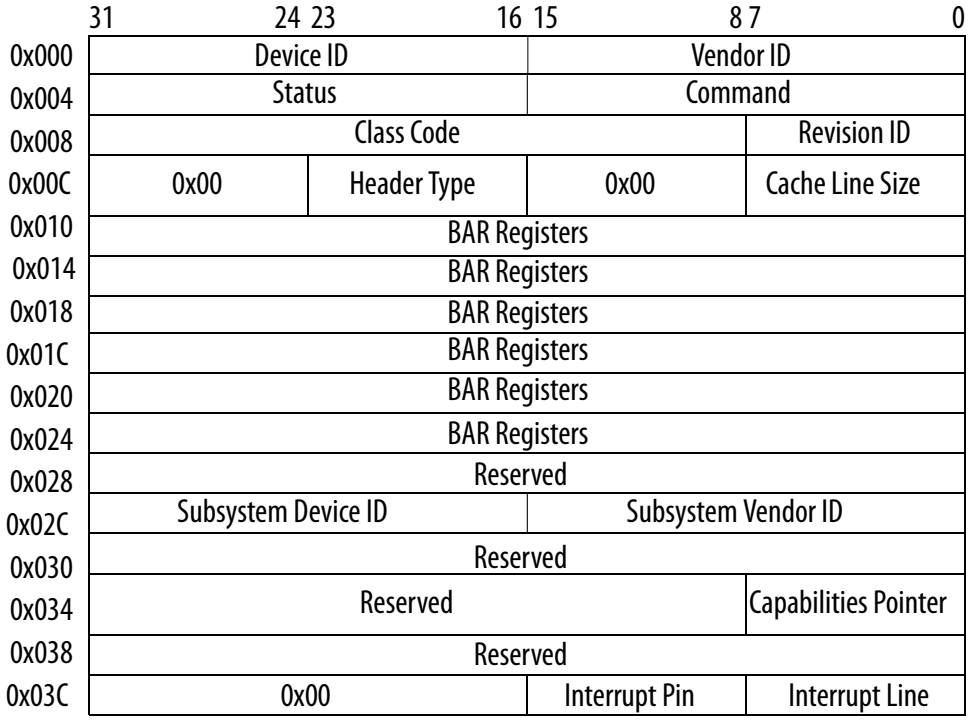
7.1.2. PCI Configuration Header Registers

The *Correspondence between Configuration Space Registers and the PCIe Specification* lists the appropriate section of the *PCI Express Base Specification* that describes these registers.

Figure 49. Configuration Space Registers Address Map

End Point Capability Structure	Required/Optional	Starting Byte Offset
PCI Header Type 0	Required	0x00
PCI Power Management	Required	0x40
MSI	Optional	0x50
PCI Express	Required	0x70
MSI-X	Optional	0xB0
AER	Required	0x100
Secondary PCIe	Required	0x188
VSEC	Required	0xB80
Custom Extensions	Optional	0xC00

Figure 50. PCI Configuration Space Registers - Byte Address Offsets and Layout



Related Information

PCI Express Base Specification 3.0

7.1.3. PCI Express Capability Structures

The layout of the most basic Capability Structures are provided below. Refer to the *PCI Express Base Specification* for more information about these registers.

Figure 51. Power Management Capability Structure - Byte Address Offsets and Layout

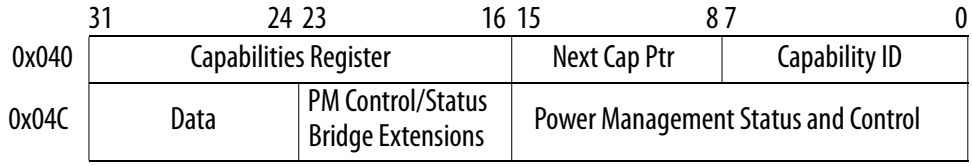


Figure 52. MSI Capability Structure

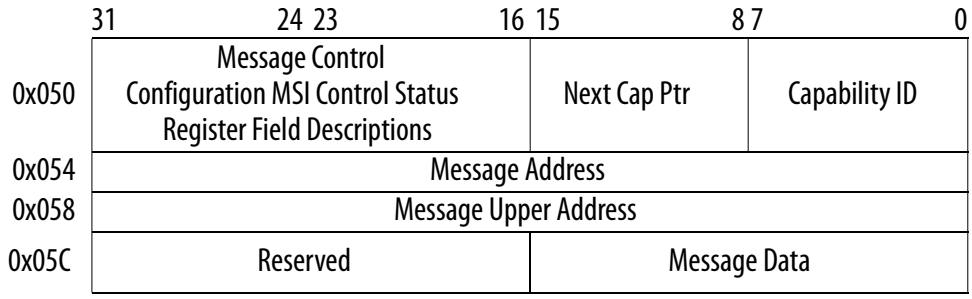


Figure 53. PCI Express Capability Structure - Byte Address Offsets and Layout

In the following table showing the PCI Express Capability Structure, registers that are not applicable to a device are reserved.

	31 24	23 16	15 8	7 0
0x070	PCI Express Capabilities Register		Next Cap Pointer	PCI Express Capabilities ID
0x074	Device Capabilities			
0x078	Device Status		Device Control	
0x07C	Link Capabilities			
0x080	Link Status		Link Control	
0x084	Slot Capabilities			
0x088	Slot Status		Slot Control	
0x08C	Root Capabilities		Root Control	
0x090	Root Status			
0x094	Device Compatibilities 2			
0x098	Device Status 2		Device Control 2	
0x09C	Link Capabilities 2			
0x0A0	Link Status 2		Link Control 2	
0x0A4	Slot Capabilities 2			
0x0A8	Slot Status 2		Slot Control 2	

Figure 54. MSI-X Capability Structure

	31 24	23 16	15 8	7 3	2 0
0x0B0	Message Control		Next Cap Ptr	Capability ID	
0x0B4	MSI-X Table Offset				MSI-X Table BAR Indicator
0x0B8	MSI-X Pending Bit Array (PBA) Offset				MSI-X Pending Bit Array - BAR Indicator

Figure 55. PCI Express AER Extended Capability Structure

	31	16 15	0
0x100	PCI Express Enhanced Capability Register		
0x104	Uncorrectable Error Status Register		
0x108	Uncorrectable Error Mask Register		
0x10C	Uncorrectable Error Severity Register		
0x110	Correctable Error Status Register		
0x114	Correctable Error Mask Register		
0x118	Advanced Error Capabilities and Control Register		
0x11C	Header Log Register		
0x12C	Root Error Command Register		
0x130	Root Error Status Register		
0x134	Error Source Identification Register	Correctable Error Source Identification Register	

Note: Refer to the *Advanced Error Reporting Capability* section for more details about the PCI Express AER Extended Capability Structure.

Related Information

- PCI Express Base Specification 3.0
- PCI Local Bus Specification

7.1.4. Intel Defined VSEC Capability Header

The figure below shows the address map and layout of the Intel defined VSEC Capability.

Figure 56. Vendor-Specific Extended Capability Address Map and Register Layout

	31 24	23 16	15 8	7 0
00h	Next Cap Offset	Version	PCI Express Extended Capability ID	
04h	VSEC Length	VSEC Revision	VSEC ID	
08h	Intel Marker			
0Ch	JTAG Silicon ID DW0			
10h	JTAG Silicon ID DW1			
14h	JTAG Silicon ID DW2			
18h	JTAG Silicon ID DW3			
1Ch	Reserved		User Configurable Device/Board ID	
20h	Reserved			
24h	Reserved			
28h	Reserved			
2Ch	Reserved			
30h	General-Purpose Control and Status			
34h	Uncorrectable Internal Error Status Register			
38h	Uncorrectable Internal Error Mask Register			
3Ch	Correctable Internal Error Status Register			
40h	Correctable Internal Error Mask Register			
44h	Reserved			
48h	Reserved			
4Ch	Reserved			
50h	Reserved			
54h	Reserved			
58h	Reserved			

Table 54. Intel-Defined VSEC Capability Header - 0xB80

Bits	Register Description	Default Value	Access
[31:20]	Next Capability Pointer: Value is the starting address of the next Capability Structure implemented. Otherwise, NULL.	Variable	RO
[19:16]	Version. PCIe specification defined value for VSEC version.	1	RO
[15:0]	PCI Express Extended Capability ID. PCIe specification defined value for VSEC Capability ID.	0x000B	RO

7.1.4.1. Intel Defined Vendor Specific Header

Table 55. Intel defined Vendor Specific Header - 0xB84

Bits	Register Description	Default Value	Access
[31:20]	VSEC Length. Total length of this structure in bytes.	0x5C	RO
[19:16]	VSEC. User configurable VSEC revision.	Not available	RO
[15:0]	VSEC ID. User configurable VSEC ID. You should change this ID to your Vendor ID.	0x1172	RO

7.1.4.2. Intel Marker

Table 56. Intel Marker - 0xB88

Bits	Register Description	Default Value	Access
[31:0]	Intel Marker - An additional marker for standard Intel programming software to be able to verify that this is the right structure.	0x41721172	RO

7.1.4.3. JTAG Silicon ID

This read only register returns the JTAG Silicon ID. The Intel Programming software uses this JTAG ID to make ensure that it is programming the SRAM Object File (`*.sof`) .

Table 57. JTAG Silicon ID - 0xB8C-0xB98

Bits	Register Description	Default Value (6)	Access
[31:0]	JTAG Silicon ID DW3	Unique ID	RO
[31:0]	JTAG Silicon ID DW2	Unique ID	RO
[31:0]	JTAG Silicon ID DW1	Unique ID	RO
[31:0]	JTAG Silicon ID DW0	Unique ID	RO

7.1.4.4. User Configurable Device and Board ID

Table 58. User Configurable Device and Board ID - 0xB9C

Bits	Register Description	Default Value	Access
[15:0]	Allows you to specify ID of the **.sof** file to be loaded.	From configuration bits	RO

(6) Because the Silicon ID is a unique value, it does not have a global default value.

7.1.5. Uncorrectable Internal Error Status Register

This register reports the status of the internally checked errors that are uncorrectable.When these specific errors are enabled by the `Uncorrectable Internal Error Mask` register, they are forwarded as Uncorrectable Internal Errors. This register is for debug only. Only use this register to observe behavior, not to drive logic custom logic.

Table 59. Uncorrectable Internal Error Status Register - 0xBB4

This register is for debug only. It should only be used to observe behavior, not to drive custom logic.

Bits	Register Description	Reset Value	Access
[31:13]	Reserved.	0	RO
[12]	Debug bus interface (DBI) access error status.	0	RW1CS
[11]	ECC error from Config RAM block.	0	RW1CS
[10]	Uncorrectable ECC error status for Retry Buffer.	0	RO
[9]	Uncorrectable ECC error status for Retry Start of the TLP RAM.	0	RW1CS
[8]	RX Transaction Layer parity error reported by the IP core.	0	RW1CS
[7]	TX Transaction Layer parity error reported by the IP core.	0	RW1CS
[6]	Internal error reported by the FPGA.	0	RW1CS
[5:4]	Reserved.	0	RW1CS
[3]	Uncorrectable ECC error status for RX Buffer Header #2 RAM.	0	RW1CS
[2]	Uncorrectable ECC error status for RX Buffer Header #1 RAM.	0	RW1CS
[1]	Uncorrectable ECC error status for RX Buffer Data RAM #2.	0	RW1CS
[0]	Uncorrectable ECC error status for RX Buffer Data RAM #1.	0	RW1CS

7.1.6. Uncorrectable Internal Error Mask Register

The `Uncorrectable Internal Error Mask` register controls which errors are forwarded as internal uncorrectable errors.

Table 60. Uncorrectable Internal Error Mask Register - 0xBB8

The access code RWS stands for Read Write Sticky meaning the value is retained after a soft reset of the IP core.

Bits	Register Description	Reset Value	Access
[31:13]	Reserved.	1b'0	RO
[12]	Mask for Debug Bus Interface.	1b'1	RO
[11]	Mask for ECC error from Config RAM block.	1b'1	RWS
[10]	Mask for Uncorrectable ECC error status for Retry Buffer.	1b'1	RO
[9]	Mask for Uncorrectable ECC error status for Retry Start of TLP RAM.	1b'1	RWS
[8]	Mask for RX Transaction Layer parity error reported by IP core.	1b'1	RWS
[7]	Mask for TX Transaction Layer parity error reported by IP core.	1b'1	RWS
[6]	Mask for Uncorrectable Internal error reported by the FPGA.	1b'1	RO

continued...

Bits	Register Description	Reset Value	Access
[5]	Reserved.	1b'0	RWS
[4]	Reserved.	1b'1	RWS
[3]	Mask for Uncorrectable ECC error status for RX Buffer Header #2 RAM.	1b'1	RWS
[2]	Mask for Uncorrectable ECC error status for RX Buffer Header #1 RAM.	1b'1	RWS
[1]	Mask for Uncorrectable ECC error status for RX Buffer Data RAM #2.	1b'1	RWS
[0]	Mask for Uncorrectable ECC error status for RX Buffer Data RAM #1.	1b'1	RWS

7.1.7. Correctable Internal Error Status Register

Table 61. Correctable Internal Error Status Register - 0xBBC

The `Correctable Internal Error Status` register reports the status of the internally checked errors that are correctable. When these specific errors are enabled by the `Correctable Internal Error Mask` register, they are forwarded as Correctable Internal Errors. This register is for debug only. Only use this register to observe behavior, not to drive logic custom logic.

Bits	Register Description	Reset Value	Access
[31:12]	Reserved.	0	RO
[11]	Correctable ECC error status for Config RAM.	0	RW1CS
[10]	Correctable ECC error status for Retry Buffer.	0	RW1CS
[9]	Correctable ECC error status for Retry Start of TLP RAM.	0	RW1CS
[8]	Reserved.	0	RO
[7]	Reserved.	0	RO
[6]	Internal Error reported by FPGA.	0	RW1CS
[5]	Reserved	0	RO
[4]	PHY Gen3 SKP Error occurred. Gen3 data pattern contains SKP pattern (8'b10101010) is misinterpreted as a SKP OS and causing erroneous block realignment in the PHY.	0	RW1CS
[3]	Correctable ECC error status for RX Buffer Header RAM #2.	0	RW1CS
[2]	Correctable ECC error status for RX Buffer Header RAM #1.	0	RW1CS
[1]	Correctable ECC error status for RX Buffer Data RAM #2.	0	RW1CS
[0]	Correctable ECC error status for RX Buffer Data RAM #1.	0	RW1CS

7.1.8. Correctable Internal Error Mask Register

Table 62. Correctable Internal Error Status Register - 0xBBC

The `Correctable Internal Error Status` register controls which errors are forwarded as Internal Correctable Errors.

Bits	Register Description	Reset Value	Access
[31:12]	Reserved.	0	RO
[11]	Mask for correctable ECC error status for Config RAM.	0	RWS
[10]	Mask for correctable ECC error status for Retry Buffer.	1	RWS

continued...

Bits	Register Description	Reset Value	Access
[9]	Mask for correctable ECC error status for Retry Start of TLP RAM.	1	RWS
[8]	Reserved.	0	RO
[7]	Reserved.	0	RO
[6]	Mask for internal Error reported by FPGA.	0	RWS
[5]	Reserved	0	RO
[4]	Mask for PHY Gen3 SKP Error.	1	RWS
[3]	Mask for correctable ECC error status for RX Buffer Header RAM #2.	1	RWS
[2]	Mask for correctable ECC error status for RX Buffer Header RAM #1.	1	RWS
[1]	Mask for correctable ECC error status for RX Buffer Data RAM #.	1	RWS
[0]	Mask for correctable ECC error status for RX Buffer Data RAM #1.	1	RWS

7.2. Avalon-MM DMA Bridge Registers

7.2.1. PCI Express Avalon-MM Bridge Register Address Map

The registers included in the Avalon-MM bridge map to a 32 KB address space. Reads to undefined addresses have unpredictable results.

Table 63. Stratix 10 PCIe Avalon-MM Bridge Register Map

Address Range	Registers
0x0050	Avalon-MM to PCIe Interrupt Enable Register
0x0060	Avalon-MM to PCIe Interrupt Status Register
0x0800-0x081F	Reserved.
0x0900-0x091F	Reserved.
0x1000-0x1FFF	Address Translation Table for the Bursting Avalon-MM Slave
0x3060	PCIe to Avalon-MM Interrupt Status Register
0x3070	PCIe to Avalon-MM Interrupt Enable Register
0x3A00–0x3A1F	Reserved
0x3B00–0x3B1F	Reserved.
0x3C00-0x3C1F	PCIe Configuration Information Registers

7.2.1.1. Avalon-MM to PCI Express Interrupt Status Registers

These registers contain the status of various signals in the PCI Express Avalon-MM bridge logic. These registers allow MSI or legacy interrupts to be asserted when enabled.

Only Root Complexes should access these registers; however, hardware does not prevent other Avalon-MM masters from accessing them.

Table 64. Avalon-MM to PCI Express Interrupt Status Register, 0x0060

Bit	Name	Access	Description
[31:16]	Reserved	N/A	N/A
[15:0]	AVL_IRQ_ASSERTED[15:0]	RO	Current value of the Avalon-MM interrupt (IRQ) input ports to the Avalon-MM RX master port: • 0—Avalon-MM IRQ is not being signaled. • 1—Avalon-MM IRQ is being signaled. A PCIe variant may have as many as 16 distinct IRQ input ports. Each AVL_IRQ_ASSERTED[] bit reflects the value on the corresponding IRQ input port.

7.2.1.2. Avalon-MM to PCI Express Interrupt Enable Registers

The interrupt enable registers enable either MSI or legacy interrupts.

A PCI Express interrupt can be asserted for any of the conditions registered in the `Avalon-MM to PCI Express Interrupt Status` register by setting the corresponding bits in the Avalon-MM to PCI Express `Interrupt Enable` register.

Table 65. Avalon-MM to PCI Express Interrupt Enable Register, 0x0050

Bits	Name	Access	Description
[31:16]	Reserved	N/A	N/A
[15:0]	AVL_IRQ[15:0]	RW	Enables generation of PCI Express interrupts when a specified Avalon-MM interrupt signal is asserted. Your system may have as many as 16 individual input interrupt signals.

7.2.1.3. Address Mapping for High-Performance Avalon-MM 32-Bit Slave Modules

Address mapping allows Avalon-MM masters with an address bus smaller than 64 bits that are connected to the bursting Avalon-MM slave to access the entire 64-bit PCIe address space. The **PCIe Settings** tab of the component GUI allows you to select the number and size of address mapping pages. This IP supports up to 512 address mapping pages. The minimum page size is 48 KB. The maximum page size is 4 GB.

When you enable address mapping, the slave address bus width is just large enough to fit the required address mapping pages. When address mapping is disabled, the Avalon-MM slave address bus is set to the value you specify in the GUI at configuration time. The Avalon-MM addresses are used as-is in the resulting PCIe TLPs.

When address mapping is enabled, bursts on the Avalon-MM slave interfaces must not cross address mapping page boundaries. This restriction means:

```
(address + 32 * burst count) <= (page base address + page size )
```

Address Mapping Table

The address mapping table is accessible through the `Control` and `Status` registers. Each entry in the address mapping table is 64 bits (8 bytes) wide and is composed of two successive registers. The even address registers holds bits [31:0]. The odd address registers holds bits [63:32].The higher order bits of the Avalon-MM address

select the address mapping window. The Avalon-MM lower-order address bits are passed through to the PCIe TLPs unchanged and are ignored in the address mapping table.

For example, if you define16 address mapping windows of 64 KB each at configuration time and the registers at address 0x1018 and 0x101C are programmed with 0x56780000 and 0x00001234 respectively, a read or write transaction to address 0x39AB0 on the bursting Avalon-MM slaves' interface gets transformed into a memory read or write TLP accessing PCIe address 0x0000123456789AB0.
The number of LSBs that are passed through defines the size of the page and is set at configuration time. If bits [63:32] of the resulting PCIe address are zero, TLPs with 32-bit wide addresses are created as required by the PCI Express standard.

Table 66. Avalon-MM-to-PCI Express Address Translation Table, 0x1000–0x1FFF

Address	Name	Access	Description
0x1000	A2P_ADDR_MAP_LO0	RW	Lower bits of Avalon-MM-to-PCI Express address map entry 0.
0x1004	A2P_ADDR_MAP_HI0	RW	Upper bits of Avalon-MM-to-PCI Express address map entry 0.
0x1008	A2P_ADDR_MAP_LO1	RW	Lower bits of Avalon-MM-to-PCI Express address map entry 1. This entry is only implemented if the number of address translation table entries is greater than 1.
0x100C	A2P_ADDR_MAP_HI1	RW	Upper bits of Avalon-MM-to-PCI Express address map entry 1. This entry is only implemented if the number of address translation table entries is greater than 1.

7.2.1.4. PCI Express to Avalon-MM Interrupt Status and Enable Registers for Endpoints

These registers record the status of various signals in the PCI Express Avalon-MM bridge logic. They allow Avalon-MM interrupts to be asserted when enabled. A processor local to the interconnect fabric that processes the Avalon-MM interrupts can access these registers.

Note: These registers must not be accessed by the PCI Express Avalon-MM bridge master ports. However, nothing in the hardware prevents a PCI Express Avalon-MM bridge master port from accessing these registers.

The following table describes the Interrupt Status register for Endpoints. It records the status of all conditions that can cause an Avalon-MM interrupt to be asserted.

An Avalon-MM interrupt can be asserted for any of the conditions noted in the `Avalon-MM Interrupt Status` register by setting the corresponding bits in the `PCI Express to Avalon-MM Interrupt Enable` register.

PCI Express interrupts can also be enabled for all of the error conditions described. However, it is likely that only one of the Avalon-MM or PCI Express interrupts can be enabled for any given bit. Typically, a single process in either the PCI Express or Avalon-MM domain handles the condition reported by the interrupt.

Table 67. INTX Interrupt Enable Register for Endpoints, 0x3070

Bits	Name	Access	Description
[31:0]	1-for1 enable mapping for the bits in the `Avalon-MM Interrupt Status` register.	RW	When set to 1, indicates the setting of the associated bit in the `Avalon-MM Interrupt Status` register causes the Avalon-MM interrupt signal, `cra_irq_o`, to be asserted.

7.2.1.5. PCI Express Configuration Information Registers

The `PCIe Configuration Information` duplicate some of the information found in Configuration Space Registers. These registers provide processors residing in the Avalon-MM address domain read access to selected configuration information stored in the PCIe address domain.

Table 68. PCIe Configuration Information Registers 0x0x3C00–0x3C1F for L-Tile Devices

Address	Name	Access	Description
0x3C00	`CONFIG_INFO_0`	RO	The following fields are defined: • [31]: `IDO request enable` • [30]: `No snoop enable` • [29]: `Relax order enable` • [28:24]: `Device Number` • [23:16]: `Bus Number` • [15]: `Memory Space Enable` • [14]: Reserved • [13:8]: `Auto Negotiation Link Width` • [7]: `Bus Master Enable` • [6]: `Extended Tag Enable` • [5:3]: `Max Read Request Size` • [2:0]: `Max Payload Size`
0x3C04	`CONFIG_INFO_1`	RO	The following fields are defined: • [31]: `cfg_send_f_err` • [30]: `cfg_send_nf_err` • [29]: `cfg_send_corr_err` • [28:24] : `AER Interrupt Msg No` • [23:18]: `Auto Negotiation Link Width` • [17]: `cfg_pm_no_soft_rs` • [16]: `Read Cpl Boundary (RCB) Control` • [15:13]: Reserved • [12:8]: `PCIe Capability Interrupt Msg No` • [7:5]: Reserved • [4]: `System Power Control` • [3:2]: `System Attention Indicator Control` • [1:0]: `System Power Indicator Control`
0x3C08	`CONFIG_INFO_2`	RO	The following fields are defined: • [31]: Reserved • [30:24]: `Index of Start VF[6:0]` • [23:16]: `Number of VFs` • [15:12]: `Auto Negotiation Link Speed` • [11:8] `ATS STU[4:1]`

continued...

Address	Name	Access	Description
			• [7]: ATS STU[0] • [6]: ATS Cache Enable • [5]: ARI forward enable • [4]: Atomic request enable • [3:2]: TPH ST mode[1:0] • [1]: TPH enable[0] • [0]: VF enable
0x3C0C	CONFIG_INFO_3	RO	MSI Address Lower
0x3C10	CONFIG_INFO_4	RO	MSI Address Upper
0x3C14	CONFIG_INFO_5	RO	MSI Mask
0x3C18	CONFIG_INFO_6	RO	The following fields are defined: • [31:16] : MSI Data • [15:7]: Reserved • [6]: MSI-X Func Mask • [5]: MSI-X Enable • [4:2]: Multiple MSI Enable • [1]: 64-bit MSI • [0]: MSI Enable
0x3C1C	CONFIG_INFO_7	RO	The following fields are defined: • [31:10]: Reserved • [9:6]: Auto Negotiation Link Speed • [5:0]: Auto Negotiation Link Width

Table 69. PCIe Configuration Information Registers 0x0x3C00–0x3C27 for H-Tile Devices

Address	Name	Access	Description
0x3C00	CONFIG_INFO_0	RO	The following fields are defined: • [31]: IDO request enable • [30]: No snoop enable • [29]: Relax order enable • [28:24]: Device Number • [23:16]: Bus Number • [15]: Memory Space Enable • [14]: IDO completion enable • [13]: perr_en • [12]: serr_en • [11]: fatal_err_rpt_en • [10]: nonfatal_err_rpt_en • [9]: corr_err_rpt_en • [8]: unsupported_req_rpt_en • [7]: Bus Master Enable • [6]: Extended Tag Enable • [5:3]: Max Read Request Size • [2:0]: Max Payload Size
0x3C04	CONFIG_INFO_1	RO	The following fields are defined:

continued...

Address	Name	Access	Description
			• [31:16]: Number of VFs [15:0] • [15]: pm_no_soft_rst • [14]: Read Cpl Boundary (RCB) Control • [13]: interrupt disable • [12:8]: PCIe Capability Interrupt Msg No • [7:5]: Reserved • [4]: System Power Control • [3:2]: System Attention Indicator Control • [1:0]: System Power Indicator Control
0x3C08	CONFIG_INFO_2	RO	The following fields are defined: • [31:28]: auto negotiation link speed • [27:17]: Index of Start VF[10:0] • [16:14]: Reserved • [13:9]: ATS STU[4:0] • [8] ATS cache enable • [7]: ARI forward enable • [6]: Atomic request enable • [5:3]: TPH ST mode • [2:1]: TPH enable • [0]: VF enable
0x3C0C	CONFIG_INFO_3	RO	MSI Address Lower
0x3C10	CONFIG_INFO_4	RO	MSI Address Upper
0x3C14	CONFIG_INFO_5	RO	MSI Mask
0x3C18	CONFIG_INFO_6	RO	The following fields are defined: • [31:16] : MSI Data • [15]: cfg_send_f_err • [14]: cfg_send_nf_err • [13]: cfg_send_cor_err • [12:8]: AER interrupt message number • [7]: Reserved • [6]: MSI-X func mask • [5]: MSI-X enable • [4:2]: Multiple MSI enable • [1]: 64-bit MSI • [0]: MSI Enable
0x3C1C	CONFIG_INFO_7	RO	AER uncorrectable error mask
0x3C20	CONFIG_INFO_8	RO	AER correctable error mask
0x3C24	CONFIG_INFO_9	RO	AER uncorrectable error severity

7.2.2. DMA Descriptor Controller Registers

The DMA Descriptor Controller manages Read and Write DMA operations. The DMA Descriptor Controller is available for use with Endpoint variations. The Descriptor Controller supports up to 128 descriptors each for Read and Write Data Movers. Read and Write are from the perspective of the FPGA. A read is from PCIe address space to the FPGA Avalon-MM address space. A write is to PCIe address space from the FPGA Avalon-MM space.

You program the Descriptor Controller internal registers with the location and size of the descriptor table residing in the PCIe address space. The DMA Descriptor Controller instructs the Read Data Mover to copy the table to its own internal FIFO. When the DMA Descriptor Controller is instantiated as a separate component, it drives table entries on the `RdDmaRxData_i[159:0]` and `WrDmaRxData_i[159:0]` buses. When the DMA Descriptor Controller is embedded inside the Avalon-MM DMA bridge, it drives this information on internal buses. .

The Read Data Mover transfers data from the PCIe address space to Avalon-MM address space. It issues memory read TLPs on the PCIe link. It writes the data returned to a location in the Avalon-MM address space. The source address is the address for the data in the PCIe address space. The destination address is in the Avalon-MM address space.

The Write Data Mover reads data from the Avalon-MM address space and writes to the PCIe address space. It issues memory write TLPs on the PCIe link. The source address is in the Avalon-MM address space. The destination address is in the PCIe address space.

The DMA Descriptor Controller records the completion status for read and write descriptors in separate status tables. Each table has 128 consecutive DWORD entries that correspond to the 128 descriptors. The actual descriptors are stored immediately after the status entries at offset 0x200 from the values programmed into the `RC Read Descriptor Base` and `RC Write Descriptor Base` registers. The status and descriptor table must be located on a 32-byte boundary in the PCIe physical address space.

The Descriptor Controller writes a 1 to the `Update` bit of the status DWORD to indicate successful completion. The Descriptor Controller also sends an MSI interrupt for the final descriptor of each transaction, or after each descriptor if `Update` bit in it in the `RD_CONTROL` or `WR_CONTROL` register is set.After receiving this MSI, host software can poll the `Update` bit to determine status. The status table precedes the descriptor table in memory. The Descriptor Controller does not write the `Update` bit nor send an MSI as each descriptor completes. It only writes the `Update` bit or sends an MSI for the descriptor whose ID is stored in the `RD_DMA_LAST PTR` or `WR_DMA_LAST_PTR` registers , unless the `Update` bit in the `RD_CONTROL` or `WR_CONTROL` register is set.

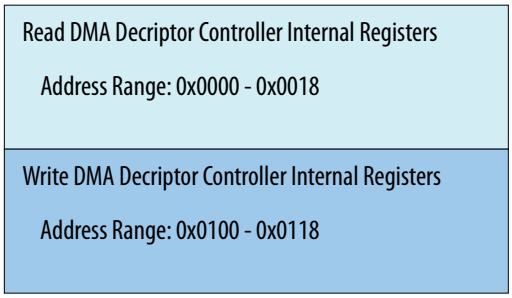
Note: Because the DMA Descriptor Controller uses FIFOs to store descriptor table entries, you cannot reprogram the DMA Descriptor Controller once it begins the transfers specified in the descriptor table.

Related Information

Programming Model for the DMA Descriptor Controller on page 105

7.2.2.1. Read DMA Internal Descriptor Controller Registers

Figure 57. Address Offsets for Read DMA and Write DMA Internal Descriptor Controller Registers



The internal Read DMA Descriptor registers provide the following information:

- Original location of descriptor table in host memory.
- Required location of descriptor table in the internal Endpoint read controller FIFO memory.
- Table size. The maximum size is 128 entries. Each entry is 32 bytes. The memory requirement is 4096 KB.
- Additional fields to track completion of the DMA descriptors.

When you choose an internal these registers are accessed through the Read Descriptor Controller Slave.When you choose an externally instantiated Descriptor Controller these registers are accessed through BAR0. The Endpoint read controller FIFO is at offset 0x0000

The following table describes the registers in the internal Read DMA Descriptor Controller and specifies their offsets. These registers are accessed through the Read Descriptor Controller Slave. When you choose an externally instantiated DMA Descriptor Controller, these registers are accessed through a user-defined BAR. Software must add the address offsets to the base address, RdDC_SLV_ADDR of the Read DMA Descriptor Controller. When you choose an internal Descriptor Controller these registers are accessed through BAR0. The Read DMA Descriptor Controller registers start at offset 0x0000.

Table 70. Read DMA Descriptor Controller Registers

Address Offset	Register	Access	Description	Reset Value
0x0000	`Read Status and Descriptor Base (Low)`	RW	Specifies the lower 32-bits of the base address of the read status and descriptor table in the PCIe system memory. This address must be on a 32-byte boundary.	Unknown
0x0004	`Read Status and Descriptor Base (High)`	RW	Specifies the upper 32-bits of the base address of the read status and descriptor table in the PCIe system memory.	Unknown
0x0008	`Read Descriptor FIFO Base (Low)`	RW	Specifies the lower 32 bits of the base address of the read descriptor FIFO in Endpoint memory.The address must be the Avalon-MM address of the Descriptor Controller's Read Descriptor Table Avalon-MM Slave Port as seen by the Read Data Mover Avalon-MM Master Port.	Unknown

continued...

Address Offset	Register	Access	Description	Reset Value
0x000C	Read Descriptor FIFO Base (High)	RW	Specifies the upper 32 bits of the base address of the read descriptor FIFO in Endpoint Avalon-MM memory. This must be the Avalon-MM address of the descriptor controller's Read Descriptor Table Avalon-MM Slave Port as seen by the Read Data Mover Avalon-MM Master Port.	Unknown
0x0010	RD_DMA_LAST_PTR	RW	[31:8]: Reserved. [7:0]: DescriptorID. When read, returns the ID of the last descriptor requested. If the DMA is in reset, returns a value 0xFF. When written, specifies the ID of the last descriptor requested. The difference between the value read and the value written is the number of descriptors to be processed. For example, if the value reads 4, the last descriptor requested is 4. To specify 5 more descriptors, software should write a 9 into the RD_DMA_LAST_PTR register. The DMA executes 5 more descriptors. To have the read DMA record the Update bit of every descriptor, program this register to transfer one descriptor at a time, or set the Update bit in the RD_CONTROL register. The descriptor ID loops back to 0 after reaching RD_TABLE_SIZE. If you want to process more pointers than RD_TABLE_SIZE - RD_DMA_LAST_PTR, you must proceed in two steps. First, process pointers up to RD_TABLE_SIZE by writing the same value as is in RD_TABLE_SIZE, Wait for that to complete. Then, write the number of remaining descriptors to RD_DMA_LAST_PTR.	[31:8]: Unknown [7:0]:0xFF
0x0014	RD_TABLE_SIZE	RW	[31:7]: Reserved. [6:0]: Size -1. This register provides for a table size less than the default size of 128 entries. The smaller size saves memory. Program this register with the value desired -1. This value specifies the last Descriptor ID.	[31:7]: Unknown [6:0]: 0x7F
0x0018	RD_CONTROL	RW	[31:1]: Reserved. [0]: Update. Controls how the descriptor processing status is reported. When the Update bit is set, returns status for every descriptor processed. If not set, then sends status back for latest entry in the RD_DMA_LAST_PTR register. The default value is 0.	[31:1]: Unknown [0]: 0x0

7.2.2.2. Write DMA Internal Descriptor Controller Registers

Table 71. Write DMA Internal Descriptor Controller Registers

The following table describes the registers in the internal write DMA Descriptor Controller and specifies their offsets. When the DMA Descriptor Controller is externally instantiated, these registers are accessed through a user-defined BAR. The offsets must be added to the base address of the Write DMA Descriptor Controller, `WrDC_SLV_ADDR`. When the Write DMA Descriptor Controller is internally instantiated these registers are accessed through BAR0. The Write DMA Descriptor Controller registers start at offset 0x0100.

Address Offset	Register	Access	Description	Reset Value
0x0000	`Write Status and Descriptor Base (Low`	R/W	Specifies the lower 32-bits of the base address of the write status and descriptor table in the PCIe system memory . This address must be on a 32-byte boundary.	Unknown
0x0004	`Write Status and Descriptor Base (High)`	R/W	Specifies the upper 32-bits of the base address of the write status and descriptor table in the PCIe system memory.	Unknown
0x0008	`Write Status and Descriptor FIFO Base (Low)`	RW	Specifies the lower 32 bits of the base address of the write descriptor FIFO in Endpoint memory. The address is the Avalon-MM address of the Descriptor Controller's Write Descriptor Table Avalon-MM Slave Port as seen by the Write Data Mover Avalon-MM Master Port.	Unknown
0x000C	`Write Status and Descriptor FIFO Base (High)`	RW	Specifies the upper 32 bits of the base address of the write descriptor FIFO in Endpoint memory. The address is the Avalon-MM address of the Descriptor Controller's Write Descriptor Table Avalon-MM Slave Port as seen by the Write Data Mover Avalon-MM Master Port.	Unknown
0x0010	`WR_DMA_LAST_PTR`	RW	[31:8]: Reserved. [7:0]: `DescriptorID`. When read, returns the ID of the last descriptor requested. If no DMA request is outstanding or the DMA is in reset, returns a value 0xFF. When written, specifies the ID of the last descriptor requested. The difference between the value read and the value written is the number of descriptors to be processed. For example, if the value reads 4, the last descriptor requested is 4. To specify 5 more descriptors, software should write a 9 into the `WR_DMA_LAST_PTR` register. The DMA executes 5 more descriptors. To have the read DMA record the `Update` bit of every descriptor, program this register to transfer one descriptor at a time, or set the `Update` bit in the `WR_CONTROL` register. The descriptor ID loops back to 0 after reaching `WR_TABLE_SIZE`. If you want to process more pointers than `WR_TABLE_SIZE - WR_DMA_LAST_PTR`, you must proceed in two steps. First, process pointers up to `WR_TABLE_SIZE` by writing the same value as is in `WR_TABLE_SIZE`, Wait for that to complete. Then, write the number of remaining descriptors to `WR_DMA_LAST_PTR`. To have the write DMA record the `Status Update` bit of every descriptor, program this register to transfer one descriptor at a time.	[31:8]: Unknown [7:0]:0xFF
0x0014	`WR_TABLE_SIZE`	RW	[31:7]: Reserved. [6:0]: `Size -1`.	[31:7]: Unknown [6:0]: 0x7F

continued...

Address Offset	Register	Access	Description	Reset Value
			This register gives you the flexibility to user to specify a table size less than the default size of 128 entries. The smaller size saves memory. Program this register with the value desired - 1. . This value specifies the last `Descriptor ID`.	
0x0018	`WR_CONTROL`	RW	[31:1]: Reserved. [0]: `Update`. Controls how the descriptor processing status is reported. When the `Update` bit is set, returns status for every descriptor processed. If not set, then only sends status back for latest entry in the `WR_DMA_LAST_PTR` register.	[31:1]: Unknown [0]: 0x0

8. Programming Model for the DMA Descriptor Controller

The Avalon-MM DMA Bridge module includes an optional DMA Descriptor Controller. When you enable this Descriptor Controller, you must follow a predefined programming model.This programming model includes the following steps:

1. Prepare the descriptor table in PCIe system memory as shown in the following figure.

Figure 58. Sample Descriptor Table

	Bits		Offset
	Bits [31:1]	Bit 0	
Read Descriptor 0 Status	Reserved	Done	0x000
Read Descriptor 1 Status	Reserved	Done	0x004
⋮	⋮	⋮	⋮
Write Descriptor 0 Status	Reserved	Done	0x200
Write Descriptor 1 Status	Reserved	Done	0x204
⋮	⋮	⋮	⋮
Read Descriptor 0: Start	SRC_ADDR_LOW		0x400
	SRC_ADDR_HIGH		0x404
	DEST_ADDR_LOW		0x408
	DEST_ADDR_HIGH		0x40C
	DMA_LENGTH		0x410
Read Descriptor 0: End	Reserved		0x414-0x41C
⋮	⋮	⋮	⋮
Read Descriptor 127: Start	SRC_ADDR_LOW		0x13F8
Read Descriptor 127: End	Reserved		0x13FC
Write Descriptor 0: Start	SRC_ADDR_LOW		0x1400
	SRC_ADDR_HIGH		0x1404
	DEST_ADDR_LOW		0x1408
	DEST_ADDR_HIGH		0x140C
	DMA_LENGTH		0x1410
Write Descriptor 0: End	Reserved		0x141C
⋮	⋮	⋮	⋮
Write Descriptor 127: Start	SRC_ADDR_LOW		0x23F8
Write Descriptor 127: End	Reserved		0x23FC

2. Program the descriptor table, providing the source and destination addresses and size for all descriptors.
3. For intermediate status updates on the individual descriptors, also program the `RD_CONTROL` or `WR_CONTROL Update` bits. Refer to the sections *Read DMA Internal Descriptor Controller Registers* and *Write DMA Internal Descriptor Controller Registers* for the descriptions of these bits.
4. Tell the DMA Descriptor Controller to instruct the Read Data Mover to copy the table to its own internal FIFO.
5. Wait for the MSI interrupt signaling the completion of the last descriptor before reprogramming the descriptor table with additional descriptors. You cannot update the descriptor table until the completion of the last descriptor that was programmed.

Here is an example for the following configuration:

- An Endpoint including the Avalon-MM Bridge with the DMA IP core
- The internal DMA Descriptor Controller
- The non-bursting Avalon-MM slave

Host software can program the Avalon-MM DMA Bridge's BAR0 non-bursting Avalon-MM master to write to the DMA Descriptor Controller's internal registers. This programming provides the information necessary for the DMA Descriptor Controller to generate DMA instructions to the PCIe Read and Write Data Movers. The DMA Descriptor Controller transmits DMA status to the host via the Avalon-MM DMA

Bridge's non-bursting Avalon-MM slave interface. The DMA Descriptor Controller's non-bursting Avalon-MM master and slave interfaces are internal and cannot be used for other purposes.

Note: When you enable the internal DMA Descriptor Controller, you can only use BAR0 to program the internal DMA Descriptor Controller. When you use BAR0, treat is as non-prefetchable.

Related Information

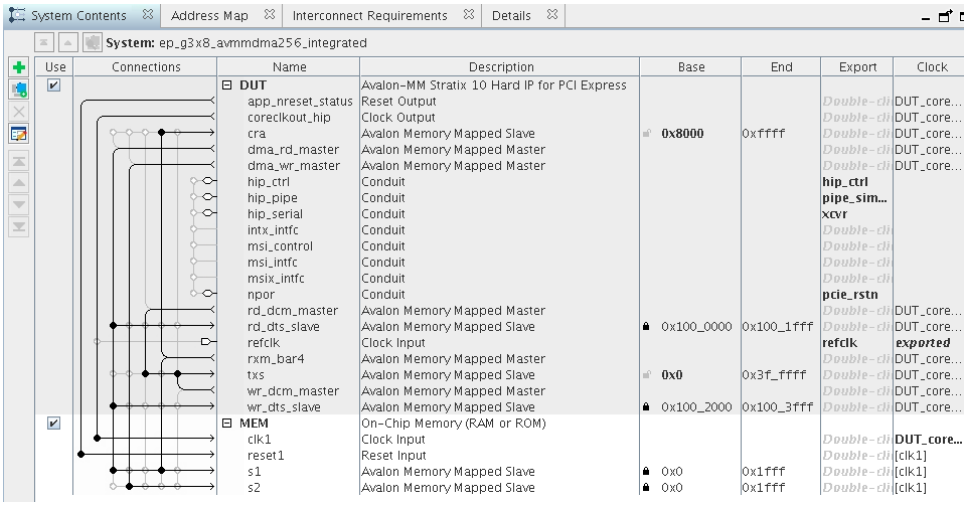
DMA Descriptor Controller Registers on page 99

8.1. Read DMA Example

This example moves three data blocks from the PCIe address space (system memory) to the Avalon-MM address space.

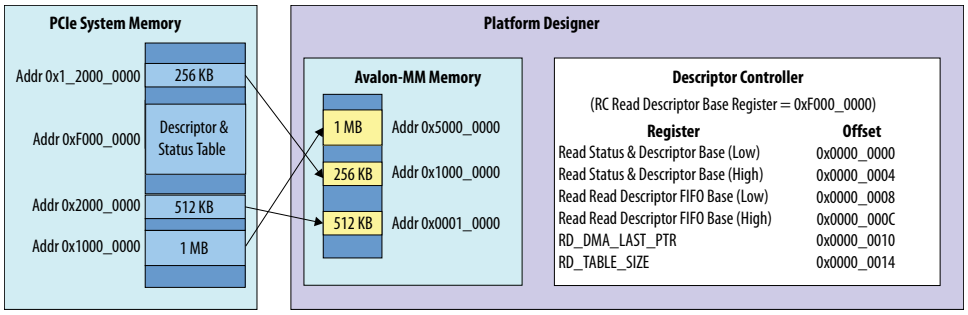
Note: Beginning with the 17.1 release, the Intel Quartus Prime Pro Edition software dynamically generates example designs for the parameters you specify in the using the parameter editor. Consequently, the Intel Quartus Prime Pro Edition installation directory no longer provides static example designs for Intel Stratix 10 devices. Static example designs are available for earlier device families, including Intel Arria 10 and Intel Cyclone® 10 devices.

Figure 59. Intel Stratix 10 Gen3 x8 Avalon-MM DMA Integrated Platform Designer



The following figures illustrate the location and size of the data blocks in the PCIe and Avalon-MM address spaces and the descriptor table format. In this example, the value of RD_TABLE_SIZE is 127.

Figure 60. Data Blocks to Transfer from PCIe to Avalon-MM Address Space Using Read DMA



The descriptor table includes 128 entries. The status table precedes a variable number of descriptors in memory. The Read and Write Status and Descriptor Tables are at the address specified in the `Read Descriptor Base Register` and `Write Descriptor Base Register`, respectively.

Figure 61. Descriptor Table In PCIe System Memory

	Bits		Offset
	Bits [31:1]	Bit 0	
Descriptor 0 Status	Reserved	Done	0x000
Descriptor 1 Status	Reserved	Done	0x004
Descriptor 2 Status	Reserved	Done	0x008
⋮	⋮	⋮	⋮
Descriptor 127 Status	Reserved	Done	0x1FC
Descriptor 0: Start	SRC_ADDR_LOW		0x200
	SRC_ADDR_HIGH		0x204
	DEST_ADDR_LOW		0x208
	DEST_ADDR_HIGH		0x20C
	DMA_LENGTH		0x210
	Reserved		0x214
	Reserved		0x218
Descriptor 0: End	Reserved		0x21C
⋮	⋮	⋮	⋮
Descriptor 127: Start	SRC_ADDR_LOW		0xFE0
Descriptor 127: End	Reserved		0xFFC

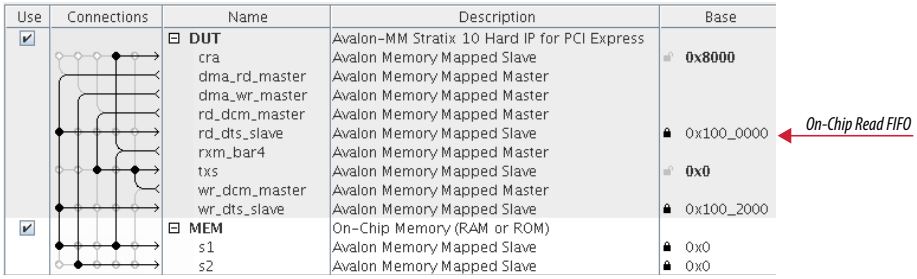
1. Software allocates memory for the Read Descriptor Status table and Descriptor table in PCIe system memory. The memory allocation requires the following calculation:
   a. Each entry in the read status table is 4 bytes. The 128 read entries require 512 bytes of memory.
   b. Each descriptor is 32 bytes. The three descriptors require 96 bytes of memory.

Note: To avoid a possible overflow condition, allocate the memory needed for the number of descriptors supported by RD_TABLE_SIZE, rather than the initial number of descriptors.

The total memory that software must allocate for the status and descriptor tables is 608 bytes. The start address of the allocated memory in this example is 0xF000_0000. Write this address into the Read Descriptor Controller `Read Status and Descriptor Base` registers.

2. Program the Read Descriptor Controller table starting at offset 0x200 from the `Read Status and Descriptor Base`. This offset matches the addresses shown in Figure 60 on page 108. The three blocks of data require three descriptors.
3. Program the Read Descriptor Controller `Read Status and Descriptor Base` register with the starting address of the descriptor status table.
4. Program the Read Descriptor Controller `Read Descriptor FIFO Base` with the starting address of the on-chip descriptor table FIFO. This is the base address for the `rd_dts_slave` port in Platform Designer. In this example, the address is 0x0100_0000.

Figure 62. Address of the On-Chip Read FIFO



5. To get status updates for each descriptor, program the Read Descriptor Controller `RD_CONTROL` register with 0x1. This step is optional.
6. Program the Read Descriptor Controller register RD_DMA_LAST_PTR with the value 3. Programming this register triggers the Read Descriptor Controller descriptor table fetch process. Consequently, writing this register must be the last step in setting up DMA transfers.
7. The host waits for the MSI interrupt. The Read Descriptor Controller sends the MSI to the host after completing the last descriptor. The Read Descriptor Controller also writes the `Update`.
8. If there are additional blocks of data to move, complete the following steps to set up additional transfers.
   a. Program the descriptor table starting from memory address 0xF000_0200 + (*<previous last descriptor pointer>* * 0x20). In this case the descriptor pointer was 3.
   b. Program the Read Descriptor Controller register RD_DMA_LAST_PTR with `previous_value (3 in this case) + number of new descriptors`. Programming this register triggers the Read Descriptor Controller descriptor table fetch process. Consequently, programming this register must be the last step in setting up DMA transfers.

Note: When RD_DMA_LAST_PTR approaches the RD_TABLE_SIZE, be sure to program the RD_DMA_LAST_PTR with a value equal to RD_TABLE_SIZE. Doing so ensures that the rollover to the first descriptor at the lowest offset occurs, (0xF000_0200 in this example). Refer to the description of the RD_DMA_LAST_PTR in the *Read DMA Descriptor Controller Registers* section for further information about programming the RD_DMA_LAST_PTR register.

Related Information

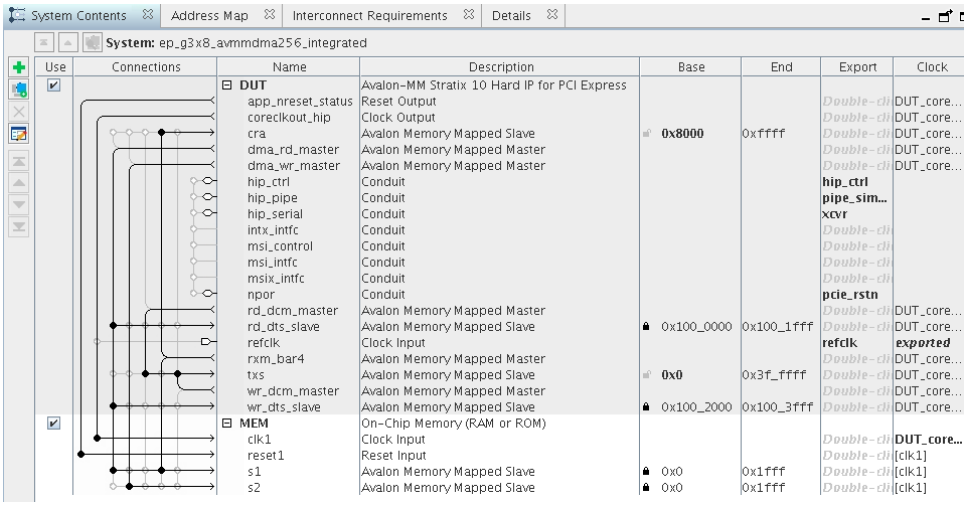
Read DMA Internal Descriptor Controller Registers on page 101

8.2. Write DMA Example

This example moves three data blocks from the Avalon-MM address space to the PCIe address space (system memory).

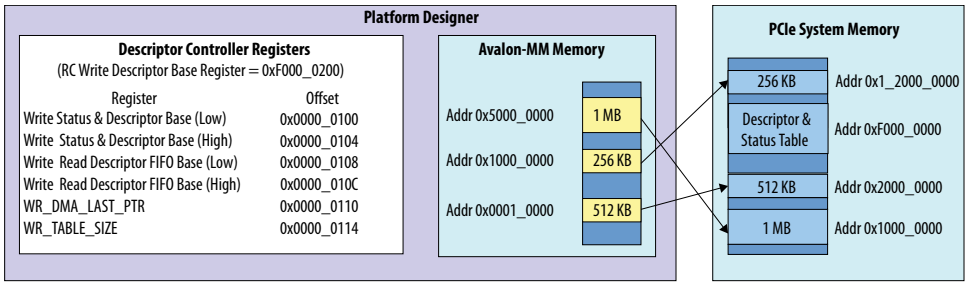
Note: Beginning with the 17.1 release, the Intel Quartus Prime Pro Edition software dynamically generates example designs for precisely the parameters you specify in the using the parameter editor. Consequently, the Intel Quartus Prime Pro Edition installation directory no longer provides static example designs for Intel Stratix 10 devices. Static example designs are available for earlier device families, including Intel Arria 10 and Intel Cyclone 10 devices.

Figure 63. Intel Stratix 10 Gen3 x8 Avalon-MM DMA Integrated Platform Designer



The following figures illustrate the location and size of the data blocks in the PCIe and Avalon-MM address spaces and the descriptor table format. In this example, the value of RD_TABLE_SIZE is 127.

Figure 64. Data Blocks to Transfer from Avalon-MM Address Space to PCIe System Memory Using Write DMA



The descriptor table includes 128 entries. The status table precedes a variable number of descriptors in memory. The Read and Write Status and Descriptor Tables are at the address specified in the `Read Descriptor Base Register` and `Write Descriptor Base Register`, respectively.

Figure 65. Descriptor Table In PCIe System Memory

	Bits		Offset
	Bits [31:1]	Bit 0	
Descriptor 0 Status	Reserved	Done	0x000
Descriptor 1 Status	Reserved	Done	0x004
Descriptor 2 Status	Reserved	Done	0x008
⋮	⋮	⋮	⋮
Descriptor 127 Status	Reserved	Done	0x1FC
Descriptor 0: Start	SRC_ADDR_LOW		0x200
	SRC_ADDR_HIGH		0x204
	DEST_ADDR_LOW		0x208
	DEST_ADDR_HIGH		0x20C
	DMA_LENGTH		0x210
	Reserved		0x214
	Reserved		0x218
Descriptor 0: End	Reserved		0x21C
⋮	⋮	⋮	⋮
Descriptor 127: Start	SRC_ADDR_LOW		0xFE0
Descriptor 127: End	Reserved		0xFFC

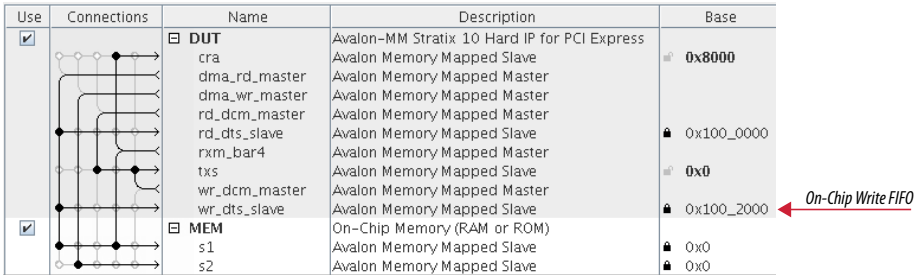
1. Software allocates memory for Write Descriptor Status table and Write Descriptor Controller table in host memory. The memory allocation requires the following calculation:
   a. Each entry in the write status table is 4 bytes. The 128 write entries require 512 bytes of memory.
   b. Each descriptor is 32 bytes. The three descriptors require 96 bytes of memory.

Note: To avoid a possible overflow condition, allocate the memory needed for the number of descriptors supported by RD_TABLE_SIZE, rather than the initial number of descriptors.

The total memory that software must allocate for the status and descriptor tables is 608 bytes. The Write Descriptor Controller Status table follows the Read Descriptor Controller Status table. The Read Status table entries require 512 bytes of memory. Consequently, the Write Descriptor Status table begins at 0xF000_0200.

2. Program the Write Descriptor Controller table starting at offset 0x200 from the address shown in Figure 64 on page 111 . The three blocks of data require three descriptors.
3. Program the Write Descriptor Controller register `Write Status and Descriptor Base` register with the starting address of the descriptor table.
4. Program the Write Descriptor Controller `Write Descriptor FIFO Base` with the starting address of the on-chip write descriptor table FIFO. This is the base address for the `wr_dts_slave` port in Platform Designer. In this example, the address is 0x0100_0200.

Figure 66. Address of the On-Chip Write FIFO



5. To get status updates for each descriptor, program the Write Descriptor Controller register `WR_CONTROL` with 0x1. This step is optional.
6. Program the Write Descriptor Controller register `WR_DMA_LAST_PTR` with the value 3. Writing this register triggers the Write Descriptor Controller descriptor table fetch process. Consequently, writing this register must be the last step in setting up DMA transfers.
7. The host waits for the MSI interrupt. The Write Descriptor Controllers sends MSI to the host after completing the last descriptor. The Write Descriptor Controller also writes the `Update`.
8. If there are additional blocks of data to move, complete the following steps to set up additional transfers.
   a. Program the descriptor table starting from memory address 0xF000_0200 + (*<previous last descriptor pointer>* * 0x20). In this case the descriptor pointer was 3.
   b. Program the Write Descriptor Controller register `WR_DMA_LAST_PTR` with `previous_value (3 in this case) + number of new descriptors`. Writing this register triggers the Write Descriptor Controller descriptor table fetch process. Consequently, writing this register must be the last step in setting up DMA transfers.

Note: When WR_DMA_LAST_PTR approaches the WR_TABLE_SIZE, be sure to program the WR_DMA_LAST_PTR with a value equal to WR_TABLE_SIZE. Doing so, ensures that the rollover to the first descriptor at the lowest offset occurs, (0xF000_0200 in this example). Refer to the description of theWR_DMA_LAST_PTR in the *Write DMA Descriptor Controller Registers* section for further information about programming the WR_DMA_LAST_PTR register.

Related Information

Write DMA Internal Descriptor Controller Registers on page 103

8.3. Software Program for Simultaneous Read and Write DMA

Program the following steps to implement a simultaneous DMA transfer:

1. Allocate PCIe system memory for the Read and Write DMA descriptor tables. If, for example, each table supports up to 128, eight-DWORD descriptors and 128, one-DWORD status entries for a total of 1152 DWORDs. Total memory for the Read and Write DMA descriptor tables is 2304 DWORDs.
2. Allocate PCIe system memory and initialize it with data for the Read Data Mover to read.
3. Allocate PCIe system memory for the Write Data Mover to write.
4. Create all the descriptors for the read DMA descriptor table. Assign the DMA `Descriptor IDs` sequentially, starting with 0 to a maximum of 127. For the read DMA, the source address is the memory space allocated in Step 2. The destination address is the Avalon-MM address that the Read Data Mover module writes. Specify the DMA length in DWORDs. Each descriptor transfers contiguous memory. Assuming a base address of 0, for the Read DMA, the following assignments illustrate construction of a read descriptor:
   a. `RD_LOW_SRC_ADDR = 0x0000` (The base address for the read descriptor table in the PCIe system memory.)
   b. `RD_HIGH_SRC_ADDR = 0x0004`
   c. `RD_CTRL_LOW_DEST_ADDR 0x0008`
   d. `RD_CTRL_HIGH_DEST_ADDR = 0x000C`
   e. `RD_DMA_LAST_PTR = 0x0010`

   Writing the RD_DMA_LAST_PTR register starts operation.
5. For the Write DMA, the source address is the Avalon-MM address that the Write Data Mover module should read. The destination address is the PCIe system memory space allocated in Step 3. Specify the DMA size in DWORDs. Assuming a base address of 0x100, for the Write Data Mover, the following assignments illustrate construction of a write descriptor:
   a. `WD_LOW_SRC_ADDR = 0x0100` (The base address for the write descriptor table in the PCIe system memory.)
   b. `WD_HIGH_SRC_ADDR = 0x0104`
   c. `WD_CTRL_LOW_DEST_ADDR 0x0108`
   d. `WD_CTRL_HIGH_DEST_ADDR = 0x010C`

e. WD_DMA_LAST_PTR = 0x0110

Writing the WD_DMA_LAST_PTR register starts operation.

6. To improve throughput, the Read DMA module copies the descriptor table to the Avalon-MM memory before beginning operation. Specify the memory address by writing to the Descriptor Table Base (Low) and (High) registers.
7. An MSI interrupt is sent for each WD_DMA_LAST_PTR or RD_DMA_LAST_PTR that completes. These completions result in updates to the Update bits. Host software can then read Update bits to determine which DMA operations are complete.

Note: If the transfer size of the read DMA is greater than the maximum read request size, the Read DMA creates multiple read requests. For example, if maximum read request size is 512 bytes, the Read Data Mover breaks a 4 KB read request into 8 requests with 8 different tags. The Read Completions can come back in any order. The Read Data Mover's Avalon-MM master port writes the data received in the Read Completions to the correct locations in Avalon-MM memory, based on the tags in the same order as it receives the Completions. This order is not necessarily in increasing address order;. The data mover does not include an internal reordering buffer. If system allows out of order read completions, then status for the latest entry is latest only in number, but potentially earlier than other completions chronologically

8.4. Read DMA and Write DMA Descriptor Format

Read and write descriptors are stored in separate descriptor tables in PCIe system memory. Each table can store up to 128 descriptors. Each descriptor is 8 DWORDs, or 32 bytes. The Read DMA and Write DMA descriptor tables start at a 0x200 byte offset from the addresses programmed into the Read Status and Descriptor Base and Write Status and Descriptor Base address registers.

Programming RD_DMA_LAST_PTR or WR_DMA_LAST_PTR registers triggers the Read or Write Descriptor Controller descriptor table fetch process. Consequently, writing these registers must be the last step in setting up DMA transfers.

Note: Because the DMA Descriptor Controller uses FIFOs to store descriptor table entries, you cannot reprogram the DMA Descriptor Controller once it begins the transfers specified in the descriptor table.

Table 72. Read Descriptor Format

You must also use this format for the Read and Write Data Movers on their Avalon-ST when you use your own DMA Controller.

Address Offset	Register Name	Description
0x00	RD_LOW_SRC_ADDR	Lower DWORD of the read DMA source address. Specifies the address in PCIe system memory from which the Read Data Mover fetches data.
0x04	RD_HIGH_SRC_ADDR	Upper DWORD of the read DMA source address. Specifies the address in PCIe system memory from which the Read Data Mover fetches data.
0x08	RD_CTRL_LOW_DEST_ADDR	Lower DWORD of the read DMA destination address. Specifies the address in the Avalon-MM domain to which the Read Data Mover writes data.

continued...

Address Offset	Register Name	Description
0x0C	RD_CTRL_HIGH_DEST_ADDR	Upper DWORD of the read DMA destination address. Specifies the address in the Avalon-MM domain to which the Read Data Mover writes data.
0x10	CONTROL	Specifies the following information: • [31:25] Reserved must be 0. • [24:18] ID. Specifies the Descriptor ID. Descriptor ID 0 is at the beginning of the table. For descriptor tables of the maximum size, Descriptor ID 127 is at the end of the table. • [17:0] SIZE. The transfer size in DWORDs. Must be non-zero. The maximum transfer size is (1 MB - 4 bytes). If the specified transfer size is less than the maximum, the transfer size is the actual size entered.
0x14 - 0x1C	Reserved	N/A

Table 73. Write Descriptor Format

Address Offset	Register Name	Description
0x00	WR_LOW_SRC_ADDR	Lower DWORD of the write DMA source address. Specifies the address in the AvalonMM domain from which the Write Data Mover fetches data.
0x04	WR_HIGH_SRC_ADDR	Upper DWORD of the write DMA source address. Specifies the address in the AvalonMM domain from which the Write Data Mover fetches data.
0x08	WR_CTRL_LOW_DEST_ADDR	Lower DWORD of the Write Data Mover destination address. Specifies the address in PCIe system memory to which the Write DMA writes data.
0x0C	WR_CTRL_HIGH_DEST_ADDR	Upper DWORD of the write DMA destination address. Specifies the address in PCIe system memory to which the Write Data Mover writes data.
0x10	CONTROL	Specifies the following information: • [31:25]: Reserved must be 0. • [24:18]:ID: Specifies the Descriptor ID. Descriptor ID 0 is at the beginning of the table. Descriptor ID is at the end of the table. • [17:0] :SIZE: The transfer size in DWORDs. Must be non-zero. The maximum transfer size is (1 MB - 4 bytes). If the specified transfer size is less than the maximum, the transfer size is the actual size entered.
0x14 - 0x1C	Reserved	N/A

9. Programming Model for the Avalon-MM Root Port

The Application Layer writes TLP-formatted data for configuration read and write requests, message requests or single-dword memory read and write requests for endpoints to the Root Port TLP TX Data Registers by using the Control Register Access (CRA) interface.

Software should check the Root Port Link Status Register to ensure the Data Link Layer Link Active bit is set to 1'b1 before issuing a configuration request to downstream ports.

The TX TLP programming model scales with the data width. The Application Layer performs the same writes for both the 64- and 128-bit interfaces. It can only support one outstanding non-posted request at a time, and must use tags 16 - 31 to identify non-posted requests.

Note: The Hard IP reconfiguration interface must be enabled for the Intel Stratix 10 Avalon-MM Root Port to provide the Application Layer direct access to the configuration space of the Root Port.

9.1. Root Port TLP Data Control and Status Registers

The 32-bit CRA Avalon-MM interface must be enabled for the Intel Stratix 10 Avalon-MM Root Port to construct the TLPs. The CRA interface provides the four registers below for this purpose.

Table 74. Root Port TLP Data, Control and Status Registers

Register Address	Register Name	Access Mode	Description
0x2000	RP_TX_REG	W	Contains 1 dword of the TX TLP. The Application Layer keeps writing to this register to construct TX TLPs.
0x2004	RP_TX_CNTRL	W	[31:3] : Reserved [2] Type : Type of request • 1 : Posted request • 0 : Non-posted request [1] EOP : Specifies the end of a packet. [0] SOP : Specifies the start of a packet.
0x2008	RP_RX_REG	R	Contains 1 dword of the Completion TLP or Message TLP.
0x200C	RP_RX_STATUS	RC	[31:2] Reserved

continued...

Register Address	Register Name	Access Mode	Description
			[1] EOP : Indicates the end of data for the TLP. The Application Layer must poll this bit to determine when the final data is available. [0] SOP : indicates that the Completion TLP or Message TLP is present.

9.2. Sending a TLP

The Application Layer performs the following sequence of Avalon-MM accesses to the CRA slave port to send a TLP Request:

1. Write the first 32 bits of the TX TLP to RP_TX_REG at address 0x2000.
2. Set RP_RP_TX_CNTRL[2:0] to 3'b001 to push the first dword of the TLP of the non-posted request into the Root Port TX FIFO.
3. Write the next 32bits of the TX TLP to RP_TX_REG at address 0x2000.
4. Set RP_RP_TX_CNTRL[2:0] to 3'b010 if the TPL is completed. Otherwise, set RP_RP_TX_CNTRL[2:0] to 3'b000 to push the next data to the TX FIFO and continue.
5. Repeat Steps 3 and 4.
6. When the TLP is completed, the Avalon-MM bridge will construct the TLP and send it downstream.

9.3. Receiving a Non-Posted Completion TLP

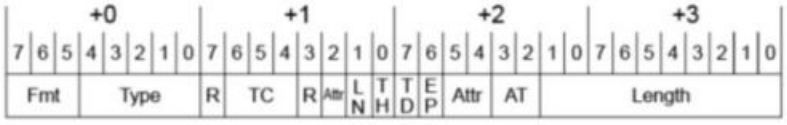
The TLPs associated with the non-posted TX requests are stored in the RP RX FIFO buffer and subsequently loaded into RP_RX_REG/STATUS registers. The Application Layer performs the following sequence to retrieve the TLPs:

1. Polls the RP_RX_STATUS.SOP bit to determine when it is set to 1'b1.
2. If RP_RX_STATUS.SOP = 1'b'1, reads RP_RX_REG to retrieve the first dword of the TLP.
3. Reads the RP_RX_STATUS.EOP bit.
   - If RP_RX_STATUS.EOP = 1'b0, reads RP_RXCPL_REG to retrieve the next dword of the TLP, then repeats this step.
   - If RP_RX_STATUS.EOP = 1'b1, reads RP_RXCPL_REG to retrieve the final dword of the TLP.

9.4. Example of Reading and Writing BAR0 Using the CRA Interface

You can use the CRA interface to send TLP requests. The Fmt and Type fields of the TLP Header provide the information required to determine the size of the remaining part of the TLP Header, and if the packet contains a data payload following the Header.

Figure 67. TLP Header Format



The CRA interface uses register addresses 0x2000 and 0x2004 to send TLPs, and register addresses 0x2008 and 0x200C to check for Completions. For details on these registers, refer to the table Root Port TLP Data Registers.

Below are examples of how to use Type 0 configuration TLPs to read from BAR0 and write to it.

1. Use the CRA interface to read an uninitialized BAR0 using a Type 0 configuration TLP with the format as shown below:

```
| fmt |   typ    |t|  tc |t|a|l|t|t|e|att| at|        length       |
| 000b|  00100b |0|_ 0 _|0|0|0|0|0|0|  0  | 0 |_______ 001 _______|
|________ req_id: 0000 _________|___ tag: 17 ___|lbe: 0 |fbe: f |
|    bdf.bus   |bdf.dev|bdf.func|rsvd20|reg_no.ext|reg_no.low|rsv|
|______ 01 ___|__ 00 _|___ 0 __|_ 0 __|___ 0 ____|___ 04 ___| 0 |
04000001 0000170f 01000010
```

   To send the TLP using the CRA interface, do the following steps:

   a. Write 0x0400_0001 to CRA interface address 0x2000.
   b. Write 0x0000_0001 to CRA interface address 0x2004 (Start of Packet).
   c. Write 0x0000_170F to CRA interface address 0x2000.
   d. Write 0x0000_0000 to CRA interface address 0x2004 (Continue).
   e. Write 0x0100_0010 to CRA interface address 0x2000.
   f. Write 0x0000_0000 to CRA interface address 0x2004 (Continue).
   g. Write 0x0000_0000 to CRA interface address 0x2000 (dummy data to achieve alignment).
   h. Write 0x0000_0002 to CRA interface address 0x2004 (End of Packet).

   Check the corresponding Completion using the CRA interface. The Completion TLP has four dwords, with the first three dwords as shown below, followed by one dword of uninitialized BAR0 value (which is 0xFFEF0010 in the following picture).

```
| fmt |   typ    |t|  tc |t|a|l|t|t|e|att| at|        length       |
| 010b|  01010b |0|_ 0 _|0|0|0|0|0|0|  0  | 0 |_______ 001 _______|
|_____ cpl_id: 0100 ___|cpl_status: 0|bcm: 0|__ byte_cnt: 004 __|
|_____ req_id: 0000 ___|____ tag: 17 ___|rsvd20: 0| low_addr: 00 |
4a000001 01000004 00001700 ffef0010
```

   To read the Completion using the CRA interface, do the following steps:

   a. Keep reading CRA interface address 0x200C until bit [0] = 0x1 (indicating the Completion packet has arrived, and you can receive the SOP in the next step).
   b. Read CRA interface address 0x2008. The read data value in this example is 0x4A00_0001.
   c. Read CRA interface address 0x200C. In this example, bits [1:0] = 0, which indicate the value read in the next step is still in the middle of the packet.
   d. Read CRA interface address 0x2008. The read data value is 0x0100_0004.

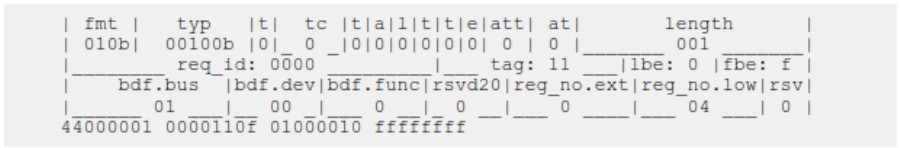
e. Read CRA interface address 0x200C. In this example, bits [1:0] = 0, which indicate the value read in the next step is still in the middle of the packet.

f. Read CRA interface address 0x2008. The read data value is 0x00001700.

g. Read CRA interface address 0x200C. In this example, bits [1:0] = 2, which indicate the value read in the next step is the EOP of the packet.

h. Read CRA interface address 0x2008. The read data value is BAR0's uninitialized value 0xFFEF0010.

2. Use the CRA interface to initialize BAR0 with 0xFFFF_FFFF using a Type 0 configuration TLP with the format as shown below:



To send the TLP using the CRA interface, do the following steps:

a. Write 0x0400_0001 to CRA interface address 0x2000.

b. Write 0x0000_0001 to CRA interface address 0x2004 (Start of Packet).

c. Write 0x0000_110F to CRA interface address 0x2000.

d. Write 0x0000_0000 to CRA interface address 0x2004 (Continue).

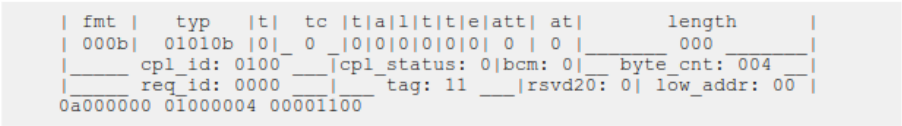
e. Write 0x0100_0010 to CRA interface address 0x2000.

f. Write 0x0000_0000 to CRA interface address 0x2004 (Continue).

g. Write 0xFFFF_FFFF to CRA interface address 0x2000.

h. Write 0x0000_0002 to CRA interface address 0x2004 (End of Packet).

Check the corresponding Completion using the CRA interface. The Completion TLP has three dwords as shown below:



To read the Completion using the CRA interface, do the following steps:

a. Keep reading CRA interface address 0x200C until bit [0] = 0x1 (indicating the Completion packet has arrived, and you can receive the SOP in the next step).

b. Read CRA interface address 0x2008. The read data value is 0x0A00_0000.

c. Read CRA interface address 0x200C. In this example, bits [1:0] = 0, which indicate the value read in the next step is still in the middle of the packet.

d. Read CRA interface address 0x2008. The read data value is 0x0100_0004.

e. Read CRA interface address 0x200C. In this example, bits [1:0] = 0, which indicate the value read in the next step is still in the middle of the packet.

f. Read CRA interface address 0x2008. The read data value is 0x00001100.

g. Read CRA interface address 0x200C. In this example, bits [1:0] = 2, which indicate the value read in the next step is the EOP of the packet.

h. Read CRA interface address 0x2008. The read data value is BAR0's size.

You can repeat Step 1 to read BAR0 after writing 0xFFFF_FFFF to it, and repeat Step 2 to configure the BAR0 address space.

Use the same method to configure BAR1, BAR2, BAR3, BAR4 and BAR5.

10. Avalon-MM Testbench and Design Example

This chapter introduces the Endpoint design example including a testbench, BFM, and a test driver module. You can create this design example using design flows described in *Quick Start Guide*. This testbench uses the parameters that you specify in the *Quick Start Guide.*

This testbench simulates up to x16 variants. However, the provided BFM only supports x1 - x8 links. It supports x16 variants by downtraining to x8. To simulate all lanes of a x16 variant, you can create a simulation model to use in an Avery testbench. This option is currently available for the Avalon-ST variants only. For more information refer to *AN-811: Using the Avery BFM for PCI Express Gen3x16 Simulation on Intel Stratix 10 Devices*.

When configured as an Endpoint variation, the testbench instantiates a design example and a Root Port BFM, which 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 a single Endpoint DUT.

The testbench uses a test driver module, `altpcietb_bfm_rp_gen3_x8.sv`, to exercise the target memory and DMA channel in the Endpoint BFM. The test driver module displays information from the Root Port Configuration Space registers, so that you can correlate to the parameters you specify using the parameter editor. The Endpoint model consists of an Endpoint variation combined with the DMA application.

Starting from the Intel Quartus Prime 18.0 release, you can generate an Intel Arria 10 PCIe example design that configures the IP as a Root Port. In this scenario, the testbench instantiates an Endpoint BFM and a JTAG master bridge.

The simulation uses the JTAG master BFM to initiate CRA read and write transactions to perform bus enumeration and configure the endpoint. The simulation also uses the JTAG master BFM to drive the TXS Avalon-MM interface to execute memory read and write transactions.

Note: The Intel testbench and Root Port BFM or Endpoint 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 with configurable parameters to exercise basic functionality of the Intel example design. The testbench and 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, or do your own extensive hardware testing or both.

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:

- It is unable to generate or receive Vendor Defined Messages. Some systems generate Vendor Defined Messages and the Application Layer must be designed to process them. The Hard IP block passes these messages on to the Application Layer which, in most cases should ignore them.
- It can only handle received read requests that are less than or equal to the currently set equal to **Device ➤ PCI Express ➤ PCI Capabilities ➤ Maximum payload size** using the parameter editor. 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 uses a fixed credit allocation.
- It does not support parity.
- It does not support multi-function designs which are available when using Configuration Space Bypass mode or Single Root I/O Virtualization (SR-IOV).

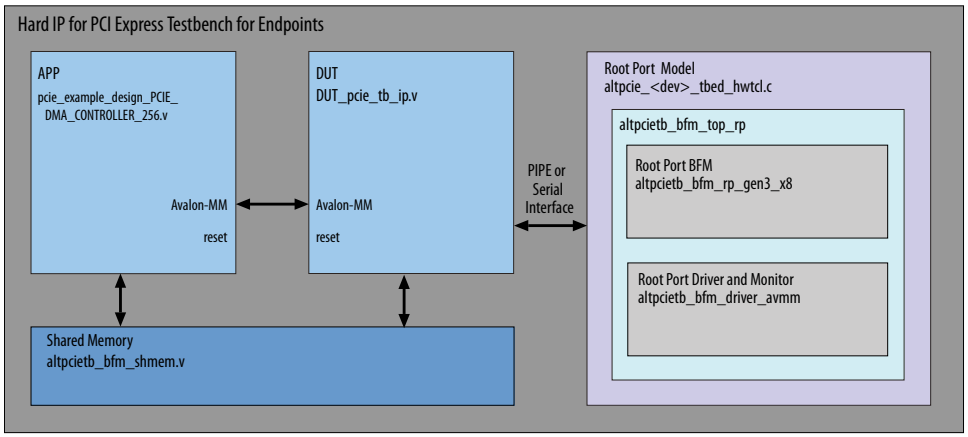
Related Information

- Quick Start Guide on page 17
- AN-811: Using the Avery BFM for PCI Express Gen3x16 Simulation on Intel Stratix 10 Devices

10.1. Avalon-MM Endpoint Testbench

You can generate the testbench from the example design by following the instructions in *Quick Start Guide*.

Figure 68. Design Example for Endpoint Designs



The Root Port BFM includes the following top-level modules in the *<testbench_dir>*/pcie_*<dev>*_hip_avmm_bridge_0_example_design/pcie_example_design_tb/ip/pcie_example_design_tb/DUT_pcie_tb_ip/altera_pcie_s10_tbed_<ver>/sim directory:

- `altpcietb_bfm_top_rp.sv`: This is the Root Port PCI Express BFM. For more information about this module, refer to *Root Port BFM*.
- `altpcietb_bfm_rp_gen3_x8.sv`: This module drives transactions to the Root Port BFM. The main process operates in two stages:
  - First, it configures the Endpoint using the task `ebfm_cfg_rp_eg`.
  - Second, it runs a memory access test with the task `target_mem_test` or `target_mem_test_lite`.

    Finally, it runs a DMA test with the task `dma_mem_test` .

  This is the module that you modify to vary the transactions sent to the example Endpoint design or your own design.
- `altpcietb_bfm_shmem.v`: This memory 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

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.

Note: Before running the testbench, you should set the `serial_sim_hwtcl` parameter in *<testbench_dir>*/ pcie_*<dev>*_hip_avmm_bridge_example_design_tb/ip/pcie_example_design_tb/DUT_pcie_tb_ip/altera_pcie_*<dev>*_tbed_*<ver>*/sim/altpcie_*<dev>*_tbed_hwtcl.v. Set to 1 for serial simulation and 0 for PIPE simulation.

Related Information

Root Port BFM on page 126

10.2. Endpoint Design Example

This design example comprises a native Endpoint, a DMA application and a Root Port BFM. The write DMA module implements write operations from the Endpoint memory to the Root Complex (RC) memory. The read DMA implements read operations from the RC memory to the Endpoint memory.

When operating on a hardware platform, a software application running on the Root Complex processor typically controls the DMA. In simulation, the generated testbench, along with this design example, provide a BFM driver module in Verilog HDL that

controls the DMA operations. Because the example relies on no other hardware interface than the PCI Express link, you can use the design example for the initial hardware validation of your system.

System generation creates the Endpoint variant in Verilog HDL. The testbench files are only available in Verilog HDL in the current release.

Note: To run the DMA tests using MSI, you must set the **Number of MSI messages requested** parameter under the **PCI Express/PCI Capabilities** page to at least 2.

The DMA design example uses an architecture capable of transferring a large amount of fragmented memory without accessing the DMA registers for every memory block. For each memory block, the DMA design example uses a descriptor table containing the following information:

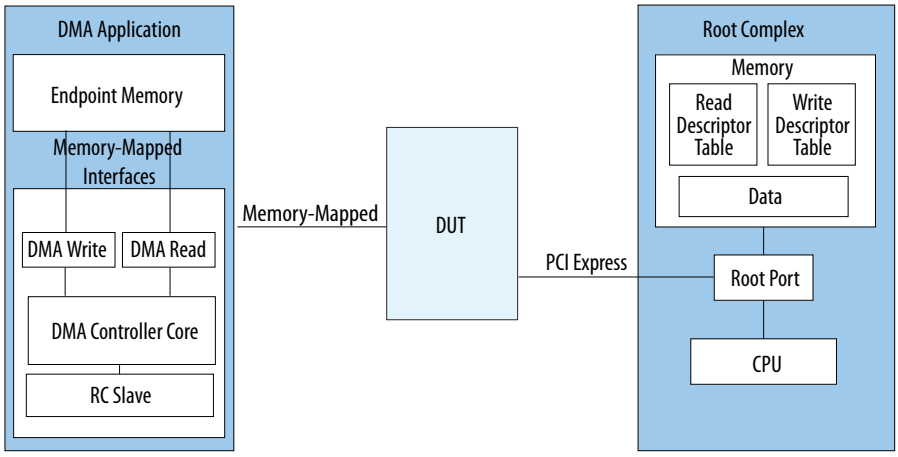
- Size of the transfer
- Address of the source
- Address of the destination
- Control bits to set the handshaking behavior between the software application or BFM driver and the DMA module

Note: The DMA design example only supports DWORD-aligned accesses. The DMA design example does not support ECRC forwarding.

The BFM driver writes the descriptor tables into BFM shared memory, from which the DMA design engine continuously collects the descriptor tables for DMA read, DMA write, or both. At the beginning of the transfer, the BFM programs the Endpoint DMA control register. The DMA control register indicates the total number of descriptor tables and the BFM shared memory address of the first descriptor table. After programming the DMA control register, the DMA engine continuously fetches descriptors from the BFM shared memory for both DMA reads and DMA writes, and then performs the data transfer for each descriptor.

The following figure shows a block diagram of the design example connected to an external RC CPU.

Figure 69. Top-Level DMA Example for Simulation



The block diagram contains the following elements:

- The DMA application connects to the Avalon-MM interface of the Intel Stratix 10 Hard IP for PCI Express. The connections consist of the following interfaces:
  - The Avalon-MM RX master receives TLP header and data information from the Hard IP block.
  - The Avalon-MM TX slave transmits TLP header and data information to the Hard IP block.
  - The Avalon-MM control register access (CRA) IRQ port requests MSI interrupts from the Hard IP block.
  - The sideband signal bus carries static information such as configuration information.
- The BFM shared memory stores the descriptor tables for the DMA read and the DMA write operations.
- A Root Complex CPU and associated PCI Express PHY connect to the Endpoint design example, using a Root Port.

The example Endpoint design and application accomplish the following objectives:

- Show you how to interface to the Intel Stratix 10 Hard IP for PCI Express using the Avalon-MM protocol.
- Provide a DMA channel that initiates memory read and write transactions on the PCI Express link.

The DMA design example hierarchy consists of these components:

- A DMA read and a DMA write module
- An on-chip Endpoint memory (Avalon-MM slave) which uses two Avalon-MM interfaces for each engine

The RC slave module typically drives downstream transactions which target the Endpoint on-chip buffer memory. These target memory transactions bypass the DMA engines. In addition, the RC slave module monitors performance and acknowledges incoming message TLPs.

Related Information

Embedded Peripherals IP User Guide Introduction
For more information about the DMA Controller.

10.2.1. BAR Setup

The `find_mem_bar` task in Root Port BFM `altpcietb_bfm_rp_gen3_x8.sv` sets up BARs to match your design.

10.3. Avalon-MM Test Driver Module

The BFM driver module, `altpcie_bfm_rp_gen3_x8.sv` tests the DMA example Endpoint design. The BFM driver module configures the Endpoint Configuration Space registers and then tests the example Endpoint DMA channel. This file is in the

<*testbench_dir*>pcie_<*dev*>_hip_avmm_bridge_0_example_design/pcie_example_design_tb/ip/pcie_example_design_tb/DUT_pcie_tb_ip/altera_pcie_<*dev*>_tbed_<*ver*>/sim directory.

The BFM test driver module performs the following steps in sequence:

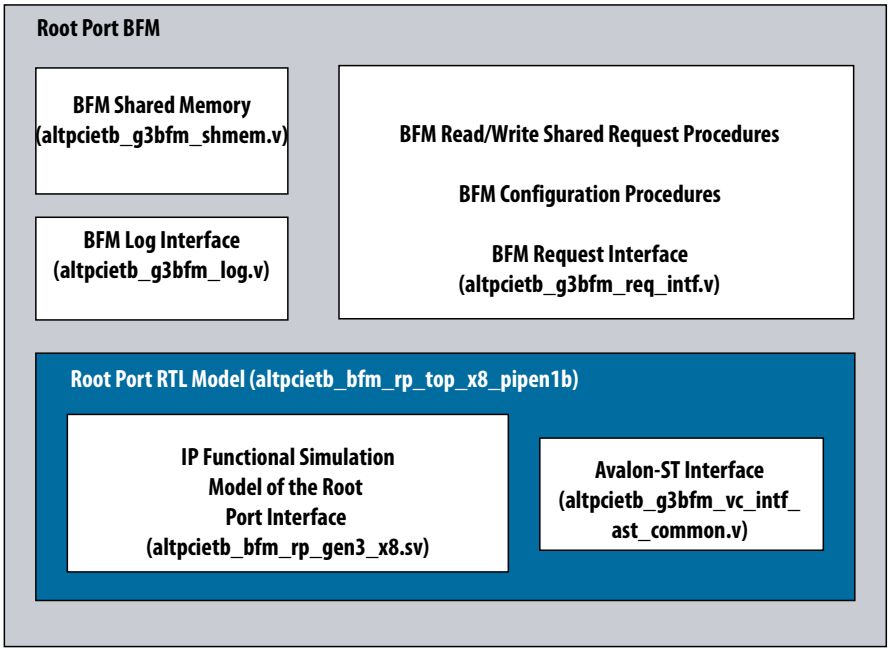
1. Configures the Root Port and Endpoint Configuration Spaces, which the BFM test driver module does by calling the procedure ebfm_cfg_rp_ep, which is part of altpcietb_bfm_rp_gen3_x8.sv.
2. Finds a suitable BAR to access the example Endpoint design Control Register space.
3. If find_mem_baridentifies a suitable BAR in the previous step, the driver performs the following tasks:
   a. DMA read: The driver programs the DMA to read data from the BFM shared memory into the Endpoint memory. The DMA issues an MSI when the last descriptor completes.
   b. DMA writ: The driver programs the DMA to write the data from its Endpoint memory back to the BFM shared memory. The DMA completes the following steps to indicate transfer completion:
      - The DMA issues an MSI when the last descriptor completes.
      - A checker compares the data written back to BFM against the data that read from the BFM.
      - The driver programs the DMA to perform a test that demonstrates downstream access of the DMA Endpoint memory.

10.4. Root Port BFM

10.4.1. Overview

The basic Root Port BFM provides a Verilog HDL task-based interface to test the PCIe link. The Root Port BFM also handles requests received from the PCIe link. The following figure provides an overview of the Root Port BFM.

Figure 70. Root Port BFM



The following descriptions provides an overview of the blocks shown in the *Root Port BFM* figure:

- BFM shared memory (`altpcietb_g3bfm_shmem.v`): The BFM memory performs the following tasks:
  - —
    - Stores data received with all completions from the PCI Express link.
    - Stores data received with all write transactions received from the PCI Express link.
    - Sources data for all completions in response to read transactions received from the PCI Express link.
    - Sources data for most write transactions issued to the link. The only exception is certain BFM PCI Express write procedures that have a four-byte field of write data passed in the call.
    - Stores a data structure that contains the sizes of and the values programmed in the BARs of the Endpoint.

A set of procedures read, write, fill, and check the shared memory from the BFM driver. For details on these procedures, see *BFM Shared Memory Access Procedures*.

- BFM Read/Write Request Functions (`altpcietb_g3bfm_rdwr.v`): These functions provide the basic BFM calls for PCI Express read and write requests. For details on these procedures, refer to *BFM Read and Write Procedures*.
- BFM Configuration Functions (`altpcietb_g3bfm_rp.v`): These functions provide the BFM calls to request configuration of the PCI Express link and the Endpoint Configuration Space registers. For details on these procedures and functions, refer to *BFM Configuration Procedures*.
- BFM Log Interface (`altpcietb_g3bfm_log.v`): The BFM log functions provides routines for writing commonly formatted messages to the simulator standard output and optionally to a log file. It also provides controls that stop simulation on errors. For details on these procedures, refer to *BFM Log and Message Procedures*.
- BFM Request Interface (`altpcietb_g3bfm_req_intf.v`): This interface provides the low-level interface between the `altpcietb_g3bfm_rdwr.v` and `altpcietb_g3bfm_configure.v` procedures or functions and the Root Port RTL Model. This interface stores a write-protected data structure containing the sizes and the values programmed in the BAR registers of the Endpoint. This interface also stores other critical data used for internal BFM management. You do not need to access these files directly to adapt the testbench to test your Endpoint application.
- 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.

10.4.2. Issuing Read and Write Transactions to the Application Layer

The `ebfm_bar` procedures in `altpcietb_bfm_rdwr.v` implement read and write transactions to the Endpoint Application Layer. The procedures and functions listed below are available in the Verilog HDL include file `altpcietb_bfm_rdwr.v`.

- `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 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.

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

10.4.3. 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 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 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 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, any 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 75. BAR Table Structure

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

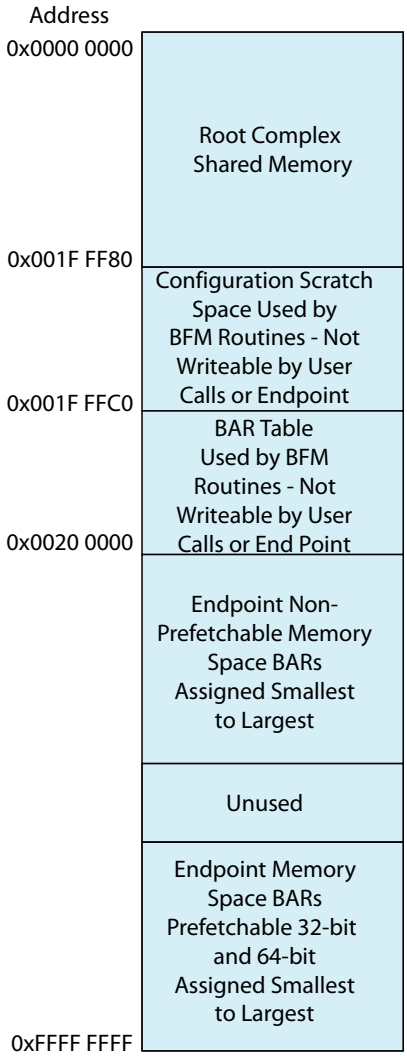
continued...

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

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

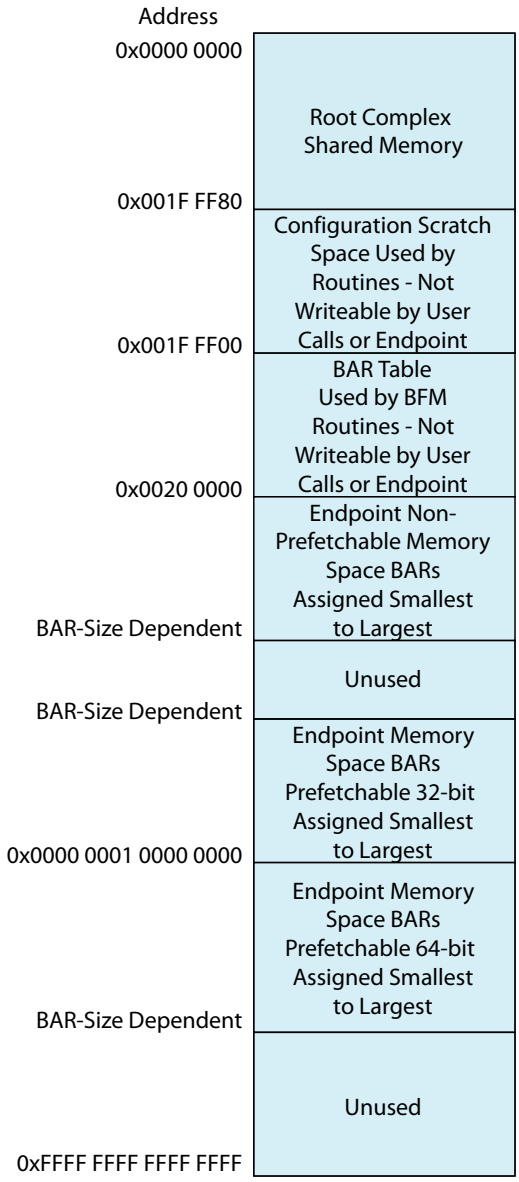
Besides the `ebfm_cfg_rp_ep` procedure in `altpcietb_bfm_rp_gen3_x8.sv`, routines to read and write Endpoint Configuration Space registers directly are available in the Verilog HDL include file. After the `ebfm_cfg_rp_ep` procedure runs the PCI Express I/O and Memory Spaces have the layout shown in the following three figures. The memory space layout depends on the value of the **addr_map_4GB_limit** input parameter. The following figure shows the resulting memory space map when the **addr_map_4GB_limit** is 1.

Figure 71. Memory Space Layout—4 GB Limit



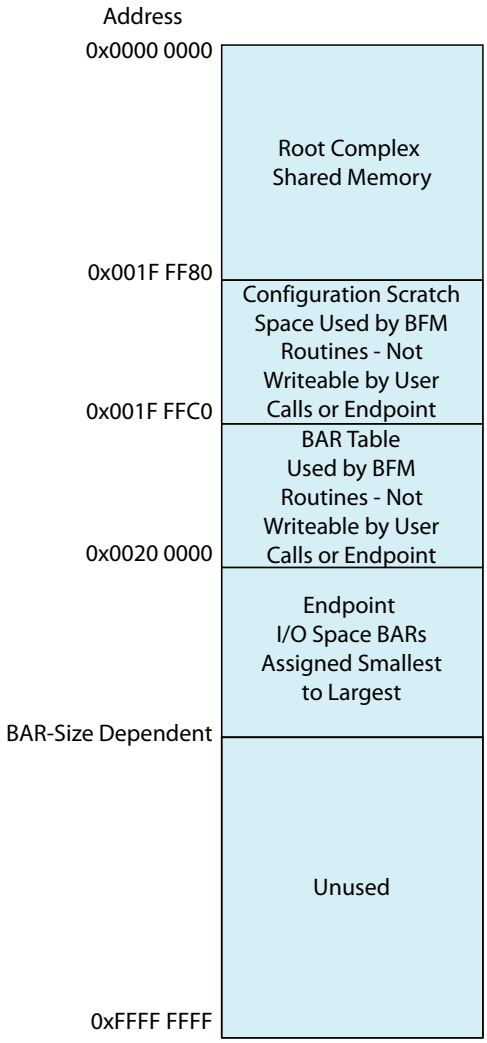
The following figure shows the resulting memory space map when the **addr_map_4GB_limit** is 0.

Figure 72. Memory Space Layout—No Limit



The following figure shows the I/O address space.

Figure 73. I/O Address Space



10.4.4. 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` 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.

10.4.5. 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.

10.5. 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.

10.5.1. 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.

10.5.2. 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.

continued...

Location	altpcietb_g3bfm_rdwr.v	
	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.

10.5.3. 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.

10.5.4. 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.

continued...

Location	altpcietb_g3bfm_rdwr.v	
	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.

10.5.5. 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.
	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

10.5.6. 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.

continued...

Location	altpcietb_g3bfm_rdwr.v	
	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]

10.5.7. 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

10.5.8. 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.

10.5.9. BFM Configuration Procedures

The BFM configuration procedures are available in `altpcietb_bfm_rp_gen3_x8.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.

10.5.9.1. 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.

10.5.9.2. 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.

10.5.10. BFM Shared Memory Access Procedures

These procedures and functions support accessing the BFM shared memory.

10.5.10.1. 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 76. 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

10.5.10.2. 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

10.5.10.3. 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. 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.

10.5.10.4. 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.

10.5.10.5. 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.

Related Information

Shared Memory Constants on page 140

10.5.10.6. 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

10.5.11. 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 77. 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:`
`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:`

10.5.11.1. 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 Table 76 on page 140.
	`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.

10.5.11.2. 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.

10.5.11.3. 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.

10.5.11.4. 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.

Related Information

BFM Log and Message Procedures on page 142

10.5.11.5. 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.

10.5.11.6. 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

10.5.12. 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.

10.5.12.1. 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

10.5.12.2. 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

10.5.12.3. 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.	

10.5.12.4. 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.

10.5.12.5. 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.

10.5.12.6. 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.

10.5.12.7. 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.

10.5.12.8. 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.

10.5.12.9. 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.

10.5.12.10. 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.

10.5.12.11. 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.

10.5.12.12. 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.

11. Troubleshooting and Observing the Link

11.1. Troubleshooting

11.1.1. Simulation Fails To Progress Beyond Polling.Active State

If your PIPE simulation cycles between the Detect.Quiet, Detect.Active, and Polling.Active LTSSM states, the PIPE interface width may be incorrect. The width of the DUT top-level PIPE interface is 32 bits for Intel Stratix 10 devices.

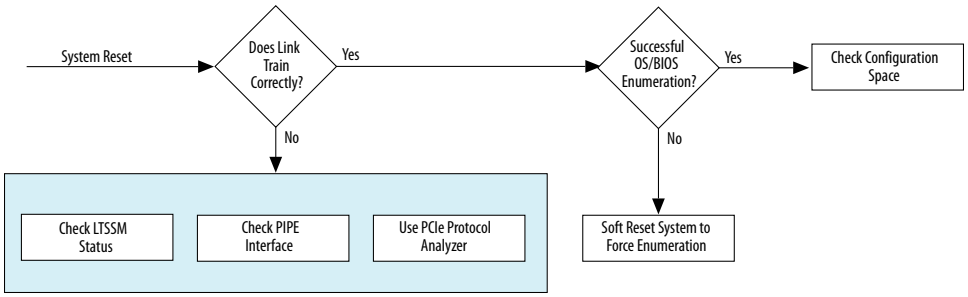
11.1.2. Hardware Bring-Up Issues

Typically, PCI Express hardware bring-up involves the following steps:

1. System reset
2. Link training
3. BIOS enumeration

The following sections describe how to debug the hardware bring-up flow. Intel recommends a systematic approach to diagnosing bring-up issues as illustrated in the following figure.

Figure 74. Debugging Link Training Issues



11.1.3. Link Training

The Physical Layer automatically performs link training and initialization without software intervention. This is a well-defined process to configure and initialize the device's Physical Layer and link so that PCIe packets can be transmitted. If you encounter link training issues, viewing the actual data in hardware should help you determine the root cause. You can use the following tools to provide hardware visibility:

- Signal Tap Embedded Logic Analyzer
- Third-party PCIe protocol analyzer

You can use Signal Tap Embedded Logic Analyzer to diagnose the LTSSM state transitions that are occurring on the PIPE interface. The `ltssmstate` bus encodes the status of LTSSM. The LTSSM state machine reflects the Physical Layer's progress through the link training process. For a complete description of the states these signals encode, refer to *Reset, Status, and Link Training Signals*. When link training completes successfully and the link is up, the LTSSM should remain stable in the L0 state. When link issues occur, you can monitor `ltssmstate` to determine the cause.

11.1.4. Use Third-Party PCIe Analyzer

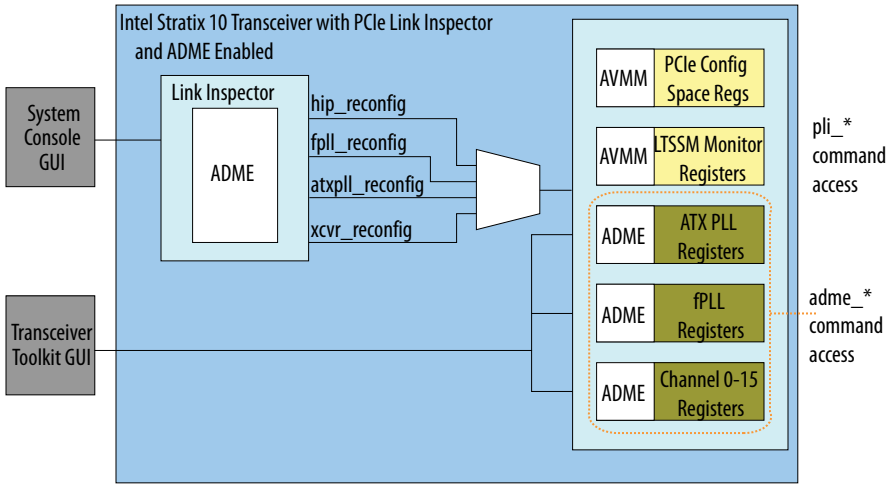
A third-party protocol analyzer for PCI Express records the traffic on the physical link and decodes traffic, saving you the trouble of translating the symbols yourself. A third-party protocol analyzer can show the two-way traffic at different levels for different requirements. For high-level diagnostics, the analyzer shows the LTSSM flows for devices on both side of the link side-by-side. This display can help you see the link training handshake behavior and identify where the traffic gets stuck. A traffic analyzer can display the contents of packets so that you can verify the contents. For complete details, refer to the third-party documentation.

11.2. PCIe Link Inspector Overview

Use the PCIe Link Inspector to monitor the PCIe link at the Physical, Data Link and Transaction Layers.

The following figure provides an overview of the debug capability available when you enable all of the options on the **Configuration, Debug and Extension Option** tab of the Intel Stratix 10 Hard IP for PCI Express IP component GUI.

Figure 75. Overview of PCIe Link Inspector Hardware



As this figure illustrates, the PCIe Link (`pli*`) commands provide access to the following registers:

- The PCI Express Configuration Space registers
- LTSSM monitor registers
- ATX PLL dynamic partial reconfiguration I/O (DPRIO) registers from the dynamic reconfiguration interface
- fPLL DPRIO registers from the dynamic reconfiguration interface
- Native PHY DPRIO registers from the dynamic reconfiguration interface

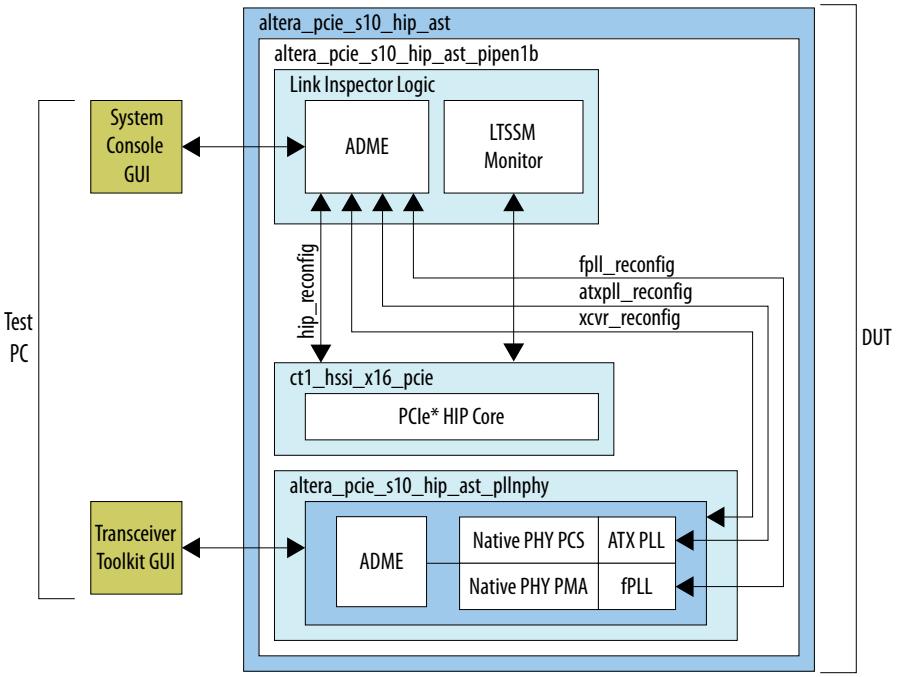
The ADME commands (`adme*_`) provide access to the following registers

- ATX PLL DPRIO registers from the ATX PLL ADME interface
- fPLL DPRIO registers from the fPLL ADME interface
- Native PHY DPRIO registers from the Native PHY ADME interface

11.2.1. PCIe Link Inspector Hardware

When you enable, the PCIe Link Inspector, the `altera_pcie_s10_hip_ast_pipen1b` module of the generated IP includes the PCIe Link Inspector as shown in the figure below.

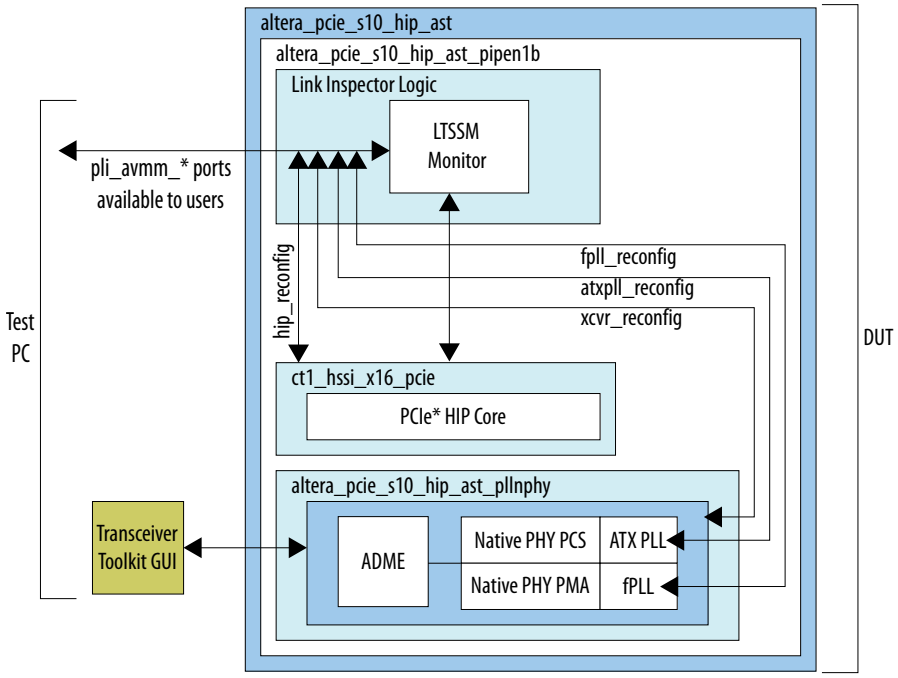
Figure 76. Intel Stratix 10 Avalon-ST and SR-IOV Hard IP for PCIe IP with the PCIe Link Inspector



You drive the PCIe Link Inspector from a System Console running on a separate test PC. The System Console connects to the PCIe Link Inspector via an Altera Debug Master Endpoint (ADME). An Intel FPGA Download Cable makes this connection.

You can also access low-level link status information from the PCIe Hard IP, XCVR or PLL blocks via the Link Inspector Avalon-MM Interface by enabling the **Enable PCIe Link Inspector Avalon-MM Interface** option in the IP GUI. See the section *Enabling the Link Inspector* for more details. When you enable this option, you do not need to use the System Console. The `pli_avmm_*` ports that are exposed connect directly to the LTSSM Monitor without going through an ADME block.

Figure 77. Intel Stratix 10 Avalon-ST and SR-IOV Hard IP for PCIe IP with the PCIe Link Inspector and the Link Inspector Avalon-MM Interface Enabled



Note: When you enable the PCIe Link Inspector, the PCIe IP has a clock, `hip_reconfig_clk`, and a reset, `hip_reconfig_rst_n`, brought out at the top level. These signals provide the clock and reset to the following interfaces:

- The ADME module
- `FPLL reconfiguration interface (fpll_reconfig)`
- `ATXPLL reconfiguration interface (atxpll_reconfig)`
- `Transceiver reconfiguration interface (xcvr_reconfig)`
- `Hard IP reconfiguration interface (hip_reconfig)`

You must provide a clock source of up to 100 MHz to drive the `hip_reconfig_clk` clock. When you run a dynamically-generated design example on the Intel Stratix 10-GX Development Kit, these signals are automatically connected.

If you run the PCIe Link Inspector on your own hardware, be sure to connect the `hip_reconfig_clk` to a 100 MHz clock source and the `hip_reconfig_rst_n` to the appropriate reset signal.

When you generate a PCIe design example (with a PCIe IP instantiated) without enabling the Link Inspector, the following interfaces are not exposed at the top level of the PCIe IP:

- `fpll_reconfig interface`
- `atxpll_reconfig interface`
- `xcvr_reconfig interface`
- `hip_reconfig interface`

If you later want to enable the Link Inspector using the same design, you need to provide a free-running clock and a reset to drive these interfaces at the top level of the PCIe IP. Intel recommends that you generate a new design example with the Link Inspector enabled. When you do so, the design example will include a free-running clock and a reset for all reconfiguration interfaces.

11.2.1.1. Enabling the PCIe Link Inspector

You enable the PCIe Link Inspector on the **Configuration Debug and Extension Options** tab of the parameter editor. You must also turn on the following parameters to use the **PCIe Link Inspector**:

- **Enable transceiver dynamic reconfiguration**
- **Enable dynamic reconfiguration of PCIe read-only registers**
- **Enable Native PHY, ATX PLL, and fPLL ADME for Transceiver Toolkit**

To have access to the low-level link status information such as the information from the LTSSM, XCVR, and PLLs using the PCIe Link Inspector from the top level of the PCIe IP, you can enable the **Enable PCIe Link Inspector AVMM Interface** option. This allows you to extract the information from the `pli_avmm_*` ports for link-level debugging without JTAG access. This optional debug capability requires you to build custom logic to read and write data from the PCIe Link Inspector.

Note: The IP GUI only exposes the **Enable PCIe Link Inspector AVMM Interface** option if you enable the **Enable PCIe Link Inspector** option.

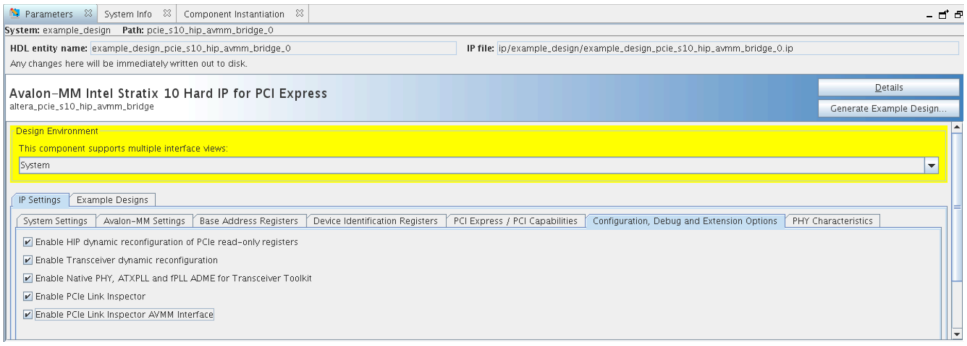
Table 78. PCIe Link Inspector Avalon-MM Interface Ports

Signal Name	Direction	Description
`pli_avmm_master_clk`	Input	Clock for the Avalon-MM defined interface
`pli_avmm_master_reset`	Input	Active-low Avalon-MM reset
`pli_avmm_master_write`	Input	Write signal
`pli_avmm_master_read`	Input	Read signal
`pli_avmm_master_address[19:0]`	Input	20-bit address
`pli_avmm_master_writedata[31:0]`	Input	32-bit write data

continued...

Signal Name	Direction	Description
pli_avmm_master_waitrequest	Output	When asserted, this signal indicates that the IP core is not ready to respond to a request.
pli_avmm_master_readdatavalid	Output	When asserted, this signal indicates that the data on pli_avmm_master_readdata[31:0] is valid.
pli_avmm_master_readdata[31:0]	Output	32-bit read data

Figure 78. Enable the Link Inspector in the Avalon-MM Intel Stratix 10 Hard IP for PCI Express IP



By default, all of these parameters are disabled.

For the design example generation, a JTAG-to-Avalon Bridge instantiation is connected to the exported pli_avmm_* ports, so that you can read all the link information via JTAG. The JTAG-to-Avalon Bridge instantiation is to verify the pli_avmm_* ports through JTAG. Without the design example generation, the JTAG-to-Avalon Bridge instantiation is not present.

11.2.1.2. Launching the PCIe Link Inspector

Use the design example you compiled in the *Quick Start Guide*, to familiarize yourself with the PCIe Link Inspector. Follow the steps in the *Generating the Avalon-ST Design* or *Generating the Avalon-MM Design* and *Compiling the Design* to generate the SRAM Object File, (.sof) for this design example.

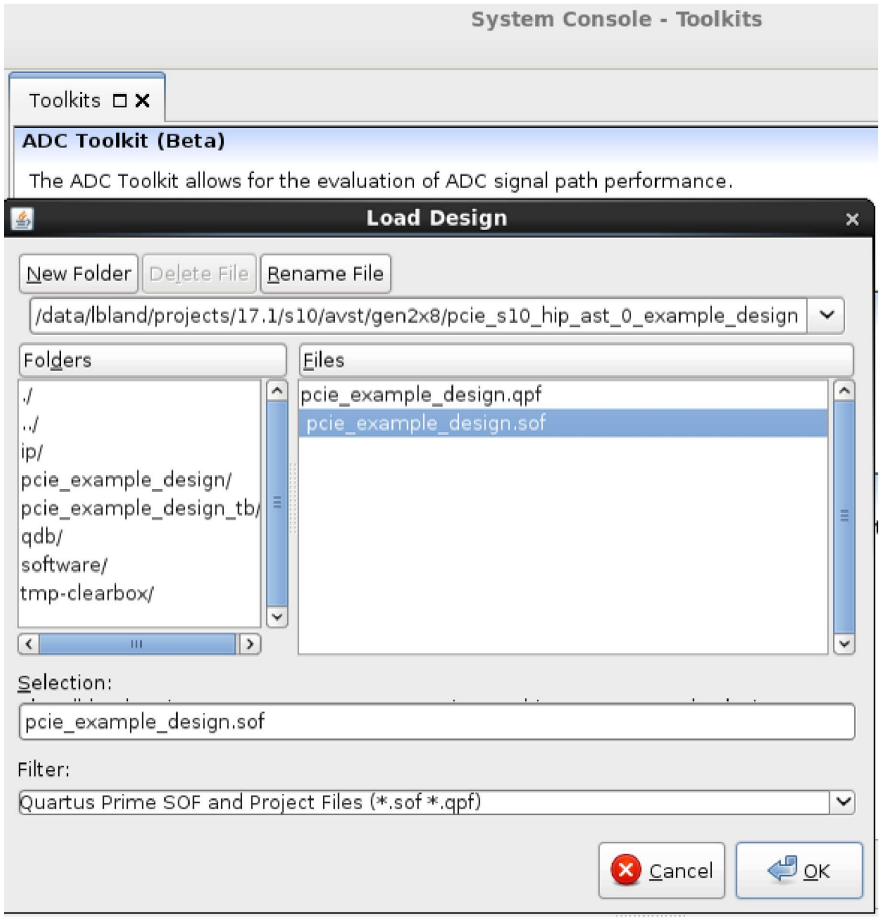
To use the PCIe Link Inspector, download the .sof to the Intel Stratix 10 Development Kit. Then, open the System Console on the test PC and load the design to the System Console as well. Loading the .sof to the System Console allows the System Console to communicate with the design using ADME. ADME is a JTAG-based Avalon-MM master. It drives an Avalon-MM slave interfaces in the PCIe design. When using ADME, the Intel Quartus Prime software inserts the debug interconnect fabric to connect with JTAG.

Here are the steps to complete these tasks:

1. Use the Intel Quartus Prime Programmer to download the .sof to the Intel Stratix 10 FPGA Development Kit.

Note: To ensure that you have the correct operation, you must use the same version of the Intel Quartus Prime Programmer and Intel Quartus Prime Pro Edition software that you used to generate the `.sof`.

2. To load the design to the System Console:
   a. Launch the Intel Quartus Prime Pro Edition software on the test PC.
   b. Start the System Console, **Tools ➤ System Debugging Tools ➤ System Console**.
   c. On the System Console File menu, select **Load design** and browse to the `.sof` file.



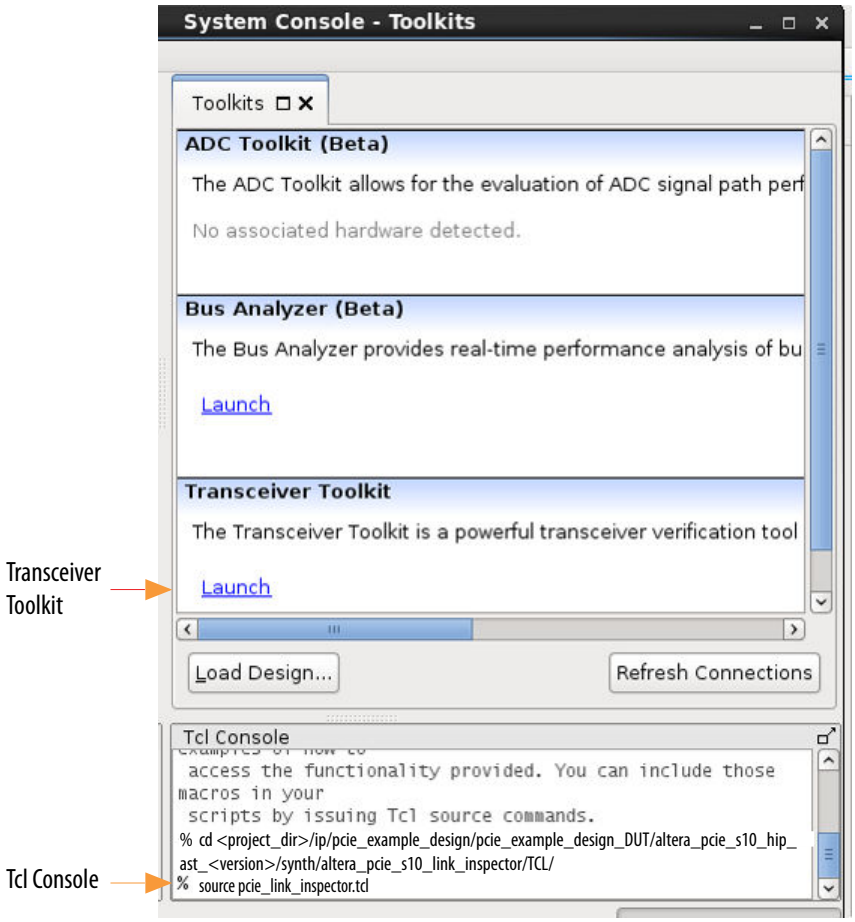
   d. Select the `.sof` and click **OK**.
   The `.sof` loads to the System Console.

3. In the System Console Tcl console, type the following commands:

```
% cd  <project_dir>/ip/pcie_example_design/
pcie_example_design_DUT/altera_pcie_s10_hip_ast_<version>/synth/
altera_pcie_s10_link_inspector
% source TCL/setup_adme.tcl
% source TCL/xcvr_pll_test_suite.tcl
% source TCL/pcie_link_inspector.tcl
```

The `source TCL/pcie_link_inspector.tcl` command automatically outputs the current status of the PCIe link to the Tcl Console. The command also loads all the PCIe Link Inspector functionality.

Figure 79. Using the Tcl Console to Access the PCIe Link Inspector



You can also start the Transceiver Toolkit from the System Console. The Transceiver Toolkit provides real-time monitoring and reconfiguration of the PMA and PCS. For example, you can use the Transceiver Toolkit Eye Viewer to get an estimate of the eye-opening at the receiver serial data sampling point. The PCIe Link Inspector extends the functionality of the Transceiver Toolkit.

11.2.1.3. The PCIe Link Inspector LTSSM Monitor

The LTSSM monitor stores up to 1024 LTSSM states and additional status information in a FIFO. When the FIFO is full, it stops storing. Reading the LTSSM offset at address 0x02 empties the FIFO. The `ltssm_state_monitor.tcl` script implements the LTSSM monitor commands.

11.2.1.3.1. LTSSM Monitor Registers

You can program the LTSSM monitor registers to change the default behavior.

Table 79. LTSSM Registers

Base Address	LTSSM Address	Access	Description
0x20000 (7)	0x00	RW	`LTSSM Monitor Control` register. The `LTSSM Monitor Control` includes the following fields: • [1:0]: `Timer Resolution Control`. Specifies the number of `hip_reconfig_clk` the PCIe link remains in each LTSSM state. The following encodings are defined: — 2'b00: The main timer increments each `hip_reconfig_clk` cycle. This is the default value. — 2'b01: The main timer increments each 16 `hip_reconfig_clk` cycles. — 2'b10: The main timer increments each 256 `hip_reconfig_clk` cycles. — 2'b11: The main timer increments each <*n*> `hip_reconfig_clk` cycles. The `Timer Resolution Step` field defines <*n*>. • [17:2]: `Timer Resolution Step`. Specifies the value of <*n*> when `Timer Resolution Control` = 2'b11. • [18]: `LTSSM FIFO reset`. The following encodings are defined: — 1'b0: The LTSSM FIFO operates normally. — 1'b1: The LTSSM FIFO is in reset. • [19]: Reserved. • [20]: `LTSSM State Match Enable`. The following encodings are defined: — 1'b0: The LTSSM State match function is disabled — 1'b1: The LTSSM State match function is enabled.When the current LTSSM state matches a state stored in the `LTSSM State Match` register, the `State Match Flag` asserts. • [27:22] `LTSSM State Match`. When enabled, the LTSSM monitor compares the value in this register against each LTSSM state. If the values match, the LTSSM state match flag (offset address 0x01, bit 29) is set to 1. • [31:28]: Reserved.
	0x01	RO	`LTSSM Quick Debug Status` register. The `LTSSM Quick Debug Status` register includes the following fields: • [9:0]: `Number LTSSM States`. Specifies the number of states currently stored in the FIFO. • [10]: `LTSSM FIFO Full Flag`. When asserted, the LTSSM FIFO is full. • [11]: `LTSSM FIFO Empty Flag`. When asserted, the LTSSM FIFO is empty. • [12]: `Current PERSTN Status`. Stores the current PERSTN value. • [13]: `Current SERDES PLL Locked`. The following encodings are defined: — 1'b0: The SERDES PLL is not locked. — 1'b1: The SERDES PLL is locked. • [14]: `PCIe Link Status`. The following encodings are defined: — 1'b0: The link is down. — 1'b1: The link is up. • [16:15] `Current PCIe Data Rate`. The following encodings are defined: — 2'b00: Reserved. — 2'b01=Gen1. — 2'b10=Gen2. — 2'b11=Gen3.

continued...

(7) When the **Enable PCIe Link Inspector AVMM Interface** option is **On**, the base address of the LTSSM Registers becomes 0x8000. Use this value to access these registers via the `pli_avmm_master_address[19:0]` ports.

Base Address	LTSSM Address	Access	Description
			• [17]: Native PHY Channel Locked to Data. The following encodings are defined: — 1'b0: At least one CDR channel is not locked to data. — 1'b1: All CDR channels are locked to data. • [21:18]: Current Number of PCIe Active Lanes. • [22]: Reserved. • [28:23]: Current LTSSM State. • [29]: LTSSM State Match Flag. Asserts when the current state matches the state you specify in LTSSM State Match. • [31:30]: Reserved.
	0x02	RO	LTSSM FIFO Output. Reading this register is equivalent to reading one entry from the LTSSM FIFO. Reading this register also updates the LTSSM FIFO, 0x03. The following fields are defined: • [5:0] LTSSM State. • [7:6]: PCIe Current Speed. • [12:8:] PCIe Lane Act. • [13]: SerDes PLL Locked. • [14]: Link Up. • [15]: PERSTN. • [16]:Native PHY Channel 0. When asserted, the CDR is locked to data. • [17]: Native PHY Channel 1. When asserted, the CDR is locked to data. • [18]: Native PHY Channel 2. When asserted, the CDR is locked to data. • [19]: Native PHY Channel 3. When asserted, the CDR is locked to data. • [20]: Native PHY Channel 4. When asserted, the CDR is locked to data. • [21]: Native PHY Channel 5. When asserted, the CDR is locked to data. • [22]: Native PHY Channel 6. When asserted, the CDR is locked to data. • [23]: Native PHY Channel 7. When asserted, the CDR is locked to data. • [24]: Native PHY Channel 8. When asserted, the CDR is locked to data. • [25]: Native PHY Channel 9. When asserted, the CDR is locked to data. • [26]: Native PHY Channel 10. When asserted, the CDR is locked to data. • [27]: Native PHY Channel 11. When asserted, the CDR is locked to data. • [29]: Native PHY Channel 12. When asserted, the CDR is locked to data. • [28]: Native PHY Channel 13. When asserted, the CDR is locked to data. • [30]: Native PHY Channel 14. When asserted, the CDR is locked to data. • [31]: Native PHY Channel 15. When asserted, the CDR is locked to data.
	0x03	RO	LTSSM FIFO Output [63:32] [29:0] Main Timer. The timer resets to 0 on each LTSSM transition. The value in this register indicates how long the PCIe link remains in each LTSSM state.
	0x04	RW	LTSSM Skip State Storage Control register. Use this register to specify a maximum of 4 LTSSM states. When LTSSM State Skip Enable is on, the LTSSM FIFO does not store the specified state or states. Refer to Table 80 on page 159 for the state encodings. [5:0]: LTSSM State 1. [6]: LTSSM State 1 Skip Enable. [12:7]: LTSSM State 2. [13]: LTSSM State 2 Skip Enable. [19:14]: LTSSM State 3. [20]: LTSSM State 3 Skip Enable. [26:21]: LTSSM State 4. [27]: LTSSM State 4 Skip Enable.

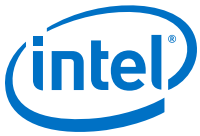


Table 80. LTSSM State Encodings for the LTSSM Skip Field

State	Encoding
Detect.Quiet	6'h00
Detect.Active	6'h01
Polling.Active	6'h02
Polling.Compliance	6'h03
Polling.Configuration	6'h04
PreDetect.Quiet	6'h05
Detect.Wait	6'h06
Configuration.Linkwidth.Start	6'h07
Configuration.Linkwidth.Accept	6'h08
Configuration.Lanenum.Wait	6'h09
Configuration.Lanenum.Accept	6'h0A
Configuration.Complete	6'h0B
Configuration.Idle	6'h0C
Recovery.RcvrLock	6'h0D
Recovery.Speed	6'h0E
Recovery.RcvrCfg	6'h0F
Recovery.Idle	6'h10
Recovery.Equalization Phase 0	6'h20
Recovery.Equalization Phase 1	6'h21
Recovery.Equalization Phase 2	6'h22
Recovery.Equalization Phase 3	6'h23
L0	6'h11
L0s	6'h12
L123.SendEIdle	6'h13
L1.Idle	6'h14
L2.Idle	6'h15
L2.TransmitWake	6'h16
Disabled.Entry	6'h17
Disabled.Idle	6'h18
Disabled	6'h19
Loopback.Entry	6'h1A
Loopback.Active	6'h1B
Loopback.Exit	6'h1C
Loopback.Exit.Timeout	6'h1D
HotReset.Entry	6'h1E
Hot.Reset	6'h1F

11.2.1.3.2. ltssm_save2file <file_name>

The `ltssm_save2file <file_name>` command empties and LTSSM FIFO and saves all states to the file name you specify.

```
% ltssm_save2file <file_name>
```

11.2.1.3.3. ltssm_file2console

You can use the command `ltssm_file2console` to display the contents of a previously saved file on the TCL system console window. For example, if you previously used the command `ltssm_save2file` to save all the LTSSM states and other status information into a file, you can use `ltssm_file2console` to display the saved contents of that file on the TCL system console window.

This is a new command that is added in the 18.1 release of Intel Quartus Prime.

11.2.1.3.4. ltssm_debug_check

The `ltssm_debug_check` command returns the current PCIe link status. It provides general information about the health of the link. Because this command returns real-time data, an unstable link returns different states every time you run it.

```
% ltssm_debug_check
```

Sample output:

```
################################################################
#####          LTSSM Quick Debugs Check Status          #####
################################################################
This PCIe is GEN1X8
Pin_Perstn signal is High
SerDes PLL is Locked
Link is Up
NOT all Channel CDRs are Locked to Data
LTSSM FIFO is Full
LTSSM FIFO is NOT Empty
LTSSM FIFO stored 1024 data
Current LTSSM State is 010001 L0
```

11.2.1.3.5. ltssm_display <num_states>

The `ltssm_display <num_states>` command reads data from the LTSSM FIFO and outputs <num_states> LTSSM states and associated status to the System Console.

```
% ltssm_display <num_states>
```

Sample output:

LTSSM[4:0]	Perstn	Locktodata	Link Up	Active Lanes	Current Speed	Timer	PLL Locked
0x00 Detect.Quiet	0	0x0000	0	0	00	0	0
0x00 Detect.Quiet	1	0x00FF	1	8	00	183237	1
0x11 L0	1	0x00FF	1	8	01	26607988	1

continued...

0x0D Recovery.Rcvlock	1	0x00FF	1	8	01	252	1
0x0F Recovery.Rcvconfig	1	0x00FF	1	8	01	786	1
0x0E Recovery.Speed	1	0x00FF	1	8	02	175242	1
0x0D Recovery.Rcvlock	1	0x00FF	1	8	02	6291458	1
0x0E Recovery.Speed	1	0x00FF	1	8	01	175296	1
0x0D Recovery.Rcvlock	1	0x00FF	1	8	01	2628	1
0x0F Recovery.Rcvconfig	1	0x00FF	1	8	0001	602	1
0x00 Recovery.Idle	1	0x00FF			0001	36	1

You can use this command to empty the FIFO by setting <num_states> to 1024:

```
% ltssm_display 1024
```

11.2.1.3.6. ltssm_save_oldstates

If you use `ltssm_display` or `ltssm_save2file` to read the LTSSM states and other status information, all that information is cleared after the execution of these commands. However, you can use the `ltssm_save_oldstates` command to overcome this hardware FIFO read clear limitation.

The first time you use `ltssm_save_oldstates`, it saves all states to the file provided at the command line. If you use this command again, it appends new readings to the same file and displays on the TCL console all the LTSSM states saved to the file.

This is a new command that is added in the 18.1 release of Intel Quartus Prime.

11.2.1.4. Accessing the Configuration Space and Transceiver Registers

11.2.1.4.1. PCIe Link Inspector Commands

These commands use the PCIe Link Inspector connection to read and write registers in the Configuration Space, LTSSM monitor, PLLs, and Native PHY channels.

Table 81. PCIe Link Inspector (PLI) Commands

These commands are available in the `link_insp_test_suite.tcl` script.

Command	Description
`pli_read32 <slave_if> <pli_base_addr> <pli_reg_addr>`	Performs a 32-bit read from the slave interface at the base address and register address specified.
`pli_read8 <slave_if> <base_addr> <reg_addr>`	Performs an 8-bit read from the slave interface at the base address and register address specified.
`pli_write32 <slave_if> <pli_base_addr> <pli_reg_addr> <value>`	Performs a 32-bit write of the value specified to the slave interface at the base address and the register address specified.
`pli_write8 <slave_if> <base_addr> <reg_addr> <value>`	Performs an 8-bit write of the value specified to the slave interface at the base address and the register address specified.

continued...

Command	Description
`pli_rmw32 <slave_if> <base_addr> <reg_addr> <bit_mask> <value>`	Performs a 32-bit read-modify-write of the value specified to the slave interface to the slave interface at the base address and register address using the bit mask specified.
`pli_rmw8 <slave_if> <base_addr> <reg_addr> <bit_mask> <value>`	Performs an 8-bit read-modify-write to the slave interface at the base address and register address using the bit mask specified.
`pli_dump_to_file <slave_if> <filename> <base_addr> <start_reg_addr> <end_reg_addr>`	Writes the contents of the slave interface to the file specified. The base address and the start and end register addresses specify range of the write. The `<slave_if>` argument can have the following values: • `$atxpll` • `$fpll` • `$channel(<n>)`

PCIe Link Inspector Command Examples

The following commands use the addresses specified below in the *Register Address Map*.

Use the following command to read register 0x480 from the ATX PLL:

```
% pli_read8 $pli_adme $atxpll_base_addr 0x480
```

Use the following command to write 0xFF to the fPLL register at address 0x4E0:

```
% pli_write8 $pli_adme $fpll_base_addr 0x4E0 0xFF
```

Use the following command to perform a read-modify-write to write 0x02 to channel 3 with a bit mask of 0x03 for the write:

```
% pli_rmw8 $pli_adme $xcvr_ch3_base_addr 0x481 0x03 0x02
```

Use the following command to instruct the LTSSM monitor to skip recording of the Recovery.Rcvlock state:

```
$pli_write $pli_adme $ltssm_base_addr 0x04 0x0000000D
```

Related Information

Register Address Map on page 164

11.2.1.4.2. ADME PLL and Channel Commands

These commands use the Native PHY and channel PLL ADME master ports to read and write registers in the ATX PLL, fPLL, and Native PHY transceiver channels.

Table 82. ADME Commands To Access PLLs and Channels

These commands are available in the `xcvr_pll_test_suite.tcl`

Command	Description
`adme_read32 <slave_if> <reg_addr>`	Performs a 32-bit read from the slave interface of the register address specified.
`adme_read8 <slave_if> <reg_addr>`	Performs an 8-bit read from the slave interface of the register address specified.
`adme_write32 <slave_if> <reg_addr> <value>`	Performs a 32-bit write of the value specified to the slave interface and register specified
`adme_write8 <slave_if> <reg_addr> <value>`	Performs an 8-bit write of the value specified to the slave interface and register specified
`adme_rmw32 <slave_if> <reg_addr> <bit_mask> <value>`	Performs a 32-bit read-modify-write to the slave interface at the register address using the bit mask specified.
`adme_rmw8 <slave_if> <reg_addr> <bit_mask> <value>`	Performs an 8-bit read-modify-write to the slave interface at the register address using the bit mask specified.
`adme_dump_to_file <slave_if> <filename> <start_addr> <end_addr>`	Writes the contents of the slave interface to the file specified. The start and end register addresses specify the range of the write. The `<slave_if>` argument can have the following values: • `$atxpll` • `$fpll` • `$channel(<n>)`
`atxpll_check`	Checks the ATX PLL lock and calibration status
`fpll_check`	Checks the fPLL lock and calibration status
`channel_check`	Checks each channel clock data recovery (CDR) lock status and the TX and RX calibration status

ADME Command Examples

The following PLL commands use the addresses specified below in the *Register Address Map*.

Use the following command to read the value register address 0x480 in the ATX PLL:

```
% adme_read8 $atxpll_adme 0x480
```

Use the following command to write 0xFF to register address 0x4E0 in the fPLL:

```
% adme_write8 $fpll_adme 0x4E0 0xFF
```

Use the following command to perform a read-modify-write to write register address 0x02 in channel 3:

```
% adme_rmw8 $channel_adme(3) 0x03 0x02
```

Use the following command to save the register values from 0x100-0x200 from the ATX PLL to a file:

```
% adme_dump_to_file $atxpll <directory_path>atx_regs.txt 0x100 0x200
```

11.2.1.4.3. Register Address Map

Here are the base addresses when you run the as defined in `link_insp_test_suite.tcl` and `ltssm_state_monitor.tcl`.

Table 83. PCIe Link Inspector and LTSSM Monitor Register Addresses

Base Address	Functional Block	Access
0x00000	fPLL	RW
0x10000	ATX PLL	RW
0x20000	LTSSM Monitor	RW
0x40000	Native PHY Channel 0	RW
0x42000	Native PHY Channel 1	RW
0x44000	Native PHY Channel 2	RW
0x46000	Native PHY Channel 3	RW
0x48000	Native PHY Channel 4	RW
0x4A000	Native PHY Channel 5	RW
0x4C000	Native PHY Channel 6	RW
0x4E000	Native PHY Channel7	RW
0x50000	Native PHY Channel 8	RW
0x52000	Native PHY Channel 9	RW
0x54000	Native PHY Channel 10	RW
0x56000	Native PHY Channel 11	RW
0x58000	Native PHY Channel 12	RW
0x5A000	Native PHY Channel 13	RW
0x5C000	Native PHY Channel 14	
0x5E000	Native PHY Channel 15	RW
0x80000	PCIe Configuration Space	RW

Related Information

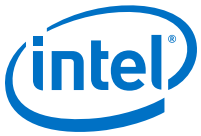
Logical View of the L-Tile/H-Tile Transceiver Registers
For detailed register descriptions for the ATX PLL, fPLL, PCS, and PMA registers.

11.2.1.5. Additional Status Commands

11.2.1.5.1. Displaying PLL Lock and Calibration Status Registers

1. Run the following System Console Tcl commands to display the lock and the calibration status for the PLLs and channels.

   ```
   % source TCL/setup_adme.tcl
   ```



```
% source TCL/xcvr_pll_suite.tcl
```

2. Here are sample transcripts:

```
##########################################
#####         ATXPLL  Status         #####
##########################################
ATXPLL is Locked
ATXPLL Calibration is Done
##########################################
#####          FPLL  Status          #####
##########################################
FPLL is Locked
FPLL Calibration is Done
##########################################
#####       Channel# 0 Status        #####
##########################################
Channel#0 CDR is Locked to Data
Channel#0 CDR is Locked to Reference Clock
Channel#0 TX Calibration is Done
Channel#0 RX Calibration is Done
##########################################
...
##########################################
Channel#7 CDR is Locked to Data
Channel#7 CDR is Locked to Reference Clock
Channel#7 TX Calibration is Done
Channel#7 RX Calibration is Done
```

A. PCI Express Core Architecture

A.1. Transaction Layer

The Transaction Layer is located between the Application Layer and the Data Link Layer. It generates and receives Transaction Layer Packets. The following illustrates the Transaction Layer. The Transaction Layer includes three sub-blocks: the TX datapath, Configuration Space, and RX datapath.

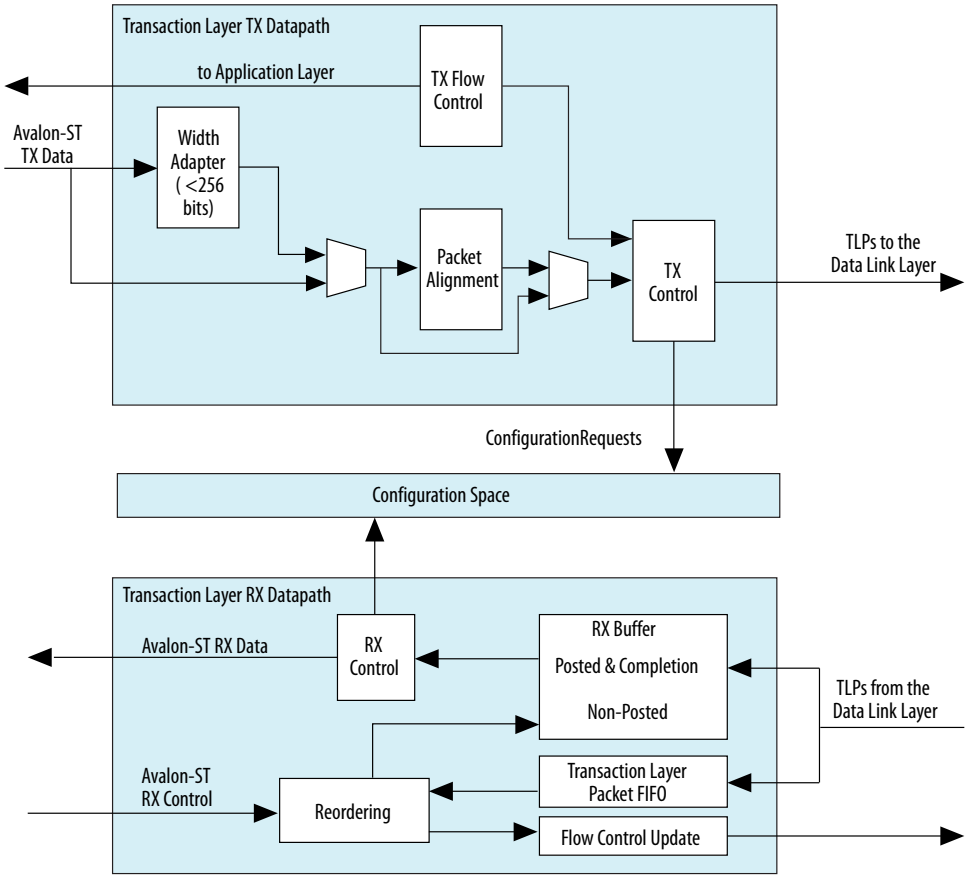
Tracing a transaction through the RX datapath includes the following steps:

1. The Transaction Layer receives a TLP from the Data Link Layer.
2. The Configuration Space determines whether the TLP is well formed and directs the packet based on traffic class (TC).
3. TLPs are stored in a specific part of the RX buffer depending on the type of transaction (posted, non-posted, and completion).
4. The receive reordering block reorders the queue of TLPs as needed, fetches the address of the highest priority TLP from the TLP FIFO block, and initiates the transfer of the TLP to the Application Layer.

Tracing a transaction through the TX datapath involves the following steps:

1. The Transaction Layer informs the Application Layer that sufficient flow control credits exist for a particular type of transaction using the TX credit signals. The Application Layer may choose to ignore this information.
2. The Application Layer requests permission to transmit a TLP. The Application Layer must provide the transaction and must be prepared to provide the entire data payload in consecutive cycles.
3. The Transaction Layer verifies that sufficient flow control credits exist and acknowledges or postpones the request. If there is insufficient space in the retry buffer, the Transaction Layer does not accept the TLP.
4. The Transaction Layer forwards the TLP to the Data Link Layer.

Figure 80. Architecture of the Transaction Layer: Dedicated Receive Buffer



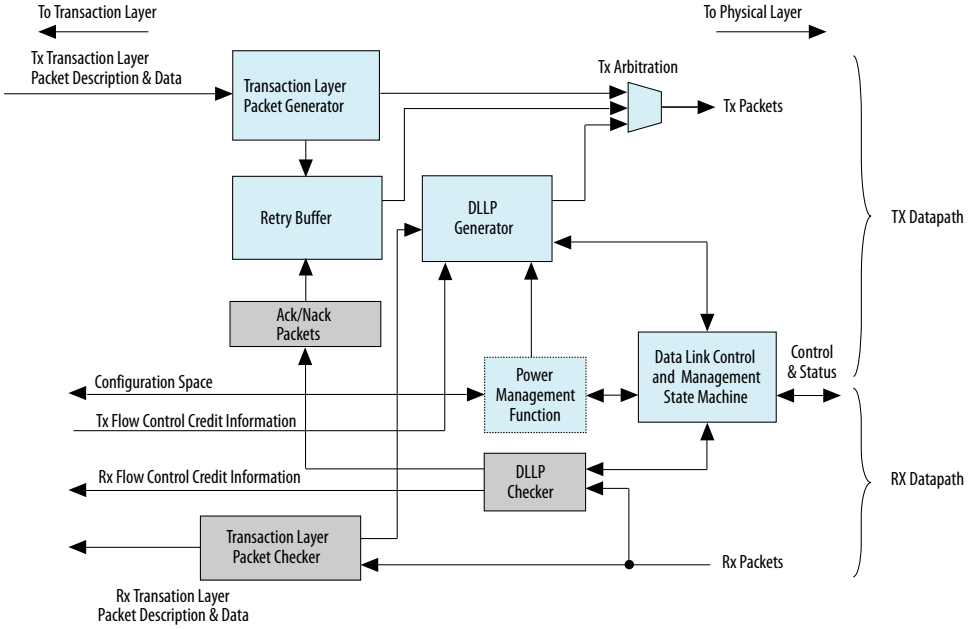
A.2. Data Link Layer

The Data Link Layer is located between the Transaction Layer and the Physical Layer. It maintains packet integrity and communicates (by DLL packet transmission) at the PCI Express link level.

The DLL implements the following functions:

- Link management through the reception and transmission of DLL Packets (DLLP), which are used for the following functions:
  - Power management of DLLP reception and transmission
  - To transmit and receive ACK/NAK packets
  - Data integrity through generation and checking of CRCs for TLPs and DLLPs
  - TLP retransmission in case of NAK DLLP reception or replay timeout, using the retry (replay) buffer
  - Management of the retry buffer
  - Link retraining requests in case of error through the Link Training and Status State Machine (LTSSM) of the Physical Layer

Figure 81. Data Link Layer



The DLL has the following sub-blocks:

- Data Link Control and Management State Machine—This state machine connects to both the Physical Layer's LTSSM state machine and the Transaction Layer. It initializes the link and flow control credits and reports status to the Transaction Layer.
- Power Management—This function handles the handshake to enter low power mode. Such a transition is based on register values in the Configuration Space and received Power Management (PM) DLLPs. All of the Intel Stratix 10 Hard IP for PCIe IP core variants do not support low power modes.
- Data Link Layer Packet Generator and Checker—This block is associated with the DLLP's 16-bit CRC and maintains the integrity of transmitted packets.
- Transaction Layer Packet Generator—This block generates transmit packets, including a sequence number and a 32-bit Link CRC (LCRC). The packets are also sent to the retry buffer for internal storage. In retry mode, the TLP generator receives the packets from the retry buffer and generates the CRC for the transmit packet.
- Retry Buffer—The retry buffer stores TLPs and retransmits all unacknowledged packets in the case of NAK DLLP reception. In case of ACK DLLP reception, the retry buffer discards all acknowledged packets.

- ACK/NAK Packets—The ACK/NAK block handles ACK/NAK DLLPs and generates the sequence number of transmitted packets.
- Transaction Layer Packet Checker—This block checks the integrity of the received TLP and generates a request for transmission of an ACK/NAK DLLP.
- TX Arbitration—This block arbitrates transactions, prioritizing in the following order:
  - Initialize FC Data Link Layer packet
  - ACK/NAK DLLP (high priority)
  - Update FC DLLP (high priority)
  - PM DLLP
  - Retry buffer TLP
  - TLP
  - Update FC DLLP (low priority)
  - ACK/NAK FC DLLP (low priority)

A.3. Physical Layer

The Physical Layer is the lowest level of the PCI Express protocol stack. It is the layer closest to the serial link. It encodes and transmits packets across a link and accepts and decodes received packets. The Physical Layer connects to the link through a high-speed SERDES interface running at 2.5 Gbps for Gen1 implementations, at 2.5 or 5.0 Gbps for Gen2 implementations, and at 2.5, 5.0 or 8.0 Gbps for Gen3 implementations.

The Physical Layer is responsible for the following actions:

- Training the link
- Scrambling/descrambling and 8B/10B encoding/decoding for 2.5 Gbps (Gen1), 5.0 Gbps (Gen2), or 128b/130b encoding/decoding of 8.0 Gbps (Gen3) per lane
- Serializing and deserializing data
- Equalization (Gen3)
- Operating the PIPE 3.0 Interface
- Implementing auto speed negotiation (Gen2 and Gen3)
- Transmitting and decoding the training sequence
- Providing hardware autonomous speed control
- Implementing auto lane reversal

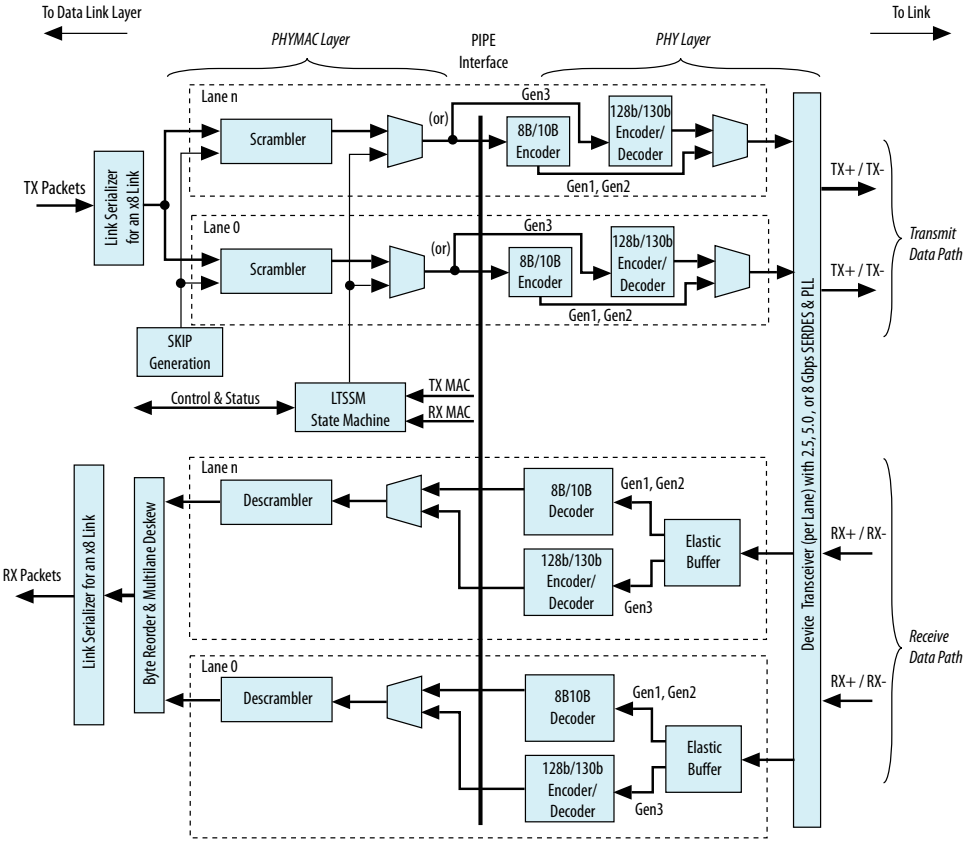
The Physical Layer is subdivided by the PIPE Interface Specification into two layers (bracketed horizontally in above figure):

- PHYMAC—The MAC layer includes the LTSSM and the scrambling/descrambling. byte reordering, and multilane deskew functions.
- PHY Layer—The PHY layer includes the 8B/10B encode and decode functions for Gen1 and Gen2. It includes 128b/130b encode and decode functions for Gen3. The PHY also includes elastic buffering and serialization/deserialization functions.

The Physical Layer integrates both digital and analog elements. Intel designed the PIPE interface to separate the PHYMAC from the PHY. The Intel Stratix 10 Hard IP for PCI Express complies with the PIPE interface specification.

Note: The internal PIPE interface is visible for simulation. It is not available for debugging in hardware using a logic analyzer such as Signal Tap. If you try to connect Signal Tap to this interface the design fails compilation.

Figure 82. Physical Layer Architecture



The PHYMAC block comprises four main sub-blocks:

- MAC Lane—Both the RX and the TX path use this block.
  - On the RX side, the block decodes the Physical Layer packet and reports to the LTSSM the type and number of TS1/TS2 ordered sets received.
  - On the TX side, the block multiplexes data from the DLL and the Ordered Set and SKP sub-block (LTSTX). It also adds lane specific information, including the lane number and the force PAD value when the LTSSM disables the lane during initialization.
- LTSSM—This block implements the LTSSM and logic that tracks TX and RX training sequences on each lane.
- For transmission, it interacts with each MAC lane sub-block and with the LTSTX sub-block by asserting both global and per-lane control bits to generate specific Physical Layer packets.
  - On the receive path, it receives the Physical Layer packets reported by each MAC lane sub-block. It also enables the multilane deskew block. This block reports the Physical Layer status to higher layers.
  - LTSTX (Ordered Set and SKP Generation)—This sub-block generates the Physical Layer packet. It receives control signals from the LTSSM block and generates Physical Layer packet for each lane. It generates the same Physical Layer Packet for all lanes and PAD symbols for the link or lane number in the corresponding TS1/TS2 fields. The block also handles the receiver detection operation to the PCS sub-layer by asserting predefined PIPE signals and waiting for the result. It also generates a SKP Ordered Set at every predefined timeslot and interacts with the TX alignment block to prevent the insertion of a SKP Ordered Set in the middle of packet.
  - Deskew—This sub-block performs the multilane deskew function and the RX alignment between the initialized lanes and the datapath. The multilane deskew implements an eight-word FIFO buffer for each lane to store symbols. Each symbol includes eight data bits, one disparity bit, and one control bit. The FIFO discards the FTS, COM, and SKP symbols and replaces PAD and IDL with D0.0 data. When all eight FIFOs contain data, a read can occur. When the multilane lane deskew block is first enabled, each FIFO begins writing after the first COM is detected. If all lanes have not detected a COM symbol after seven clock cycles, they are reset and the resynchronization process restarts, or else the RX alignment function recreates a 64-bit data word which is sent to the DLL.

B. Document Revision History

B.1. Document Revision History for the Intel Stratix 10 Avalon Memory Mapped (Avalon-MM) Hard IP for PCI Express User Guide

Document Version	Intel Quartus Prime Version	Changes
2020.01.03	19.3	Updated resource utilization numbers for the Gen1 x1 variant. Added notes stating that the Gen3 x16 variant is supported by the Intel Stratix 10 Avalon Memory Mapped (Avalon-MM) Hard IP+ for PCI Express.
2019.09.30	19.3	Added a note to clarify that this User Guide is applicable to H-Tile and L-Tile variants of the Intel Stratix 10 devices only. Added Autonomous Hard IP mode to the *Features* section.
2019.07.18	19.1	Added a note stating that `refclk` must be stable and free-running at device power-up for a successful device configuration.
2019.03.30	19.1	Added a chapter on the programming model for Root Ports. Removed the note stating that Root Port mode is not recommended. Removed the *BIOS Enumeration* section from the *Troubleshooting* chapter.
2019.03.12	18.1.1	Updated the E-Tile PAM-4 frequency to 57.8G and NRZ frequency to 28.9G.
2019.03.04	18.1.1	Updated the commands to run VCS, NCSim and Xcelium simulations in the *Simulating the Design Example* topic.
2018.12.24	18.1.1	Added the description for the Link Inspector Avalon-MM Interface. Added the Avalon-MM-to-PCIe `rxm_irq` for MSI feature.
2018.10.26	18.1	Added the statements that the IP core does not support the L1/L2 low-power states, the in-band beacon and sideband WAKE# signal.
2018.09.24	18.1	Added the `ltssm_file2console` and `ltssm_save_oldstates` commands for the PCIe Link Inspector. Updated the steps to run ModelSim simulations for a design example. Updated the steps to run a design example.
2018.08.29	18.0	Added the step to invoke vsim to the instructions for running a ModelSim simulation.

Date	Version	Changes
May 2018	18.0	Made the following changes to the user guide: • Edited the chapter on *32-Bit Control Register Access (CRA)* to state that in RP mode, application logic must set the Tag field in the TLP Header to 0x10. • Added the note that AER is always enabled to the *Features* chapter. • Added a sub-topic in the chapter *Interfaces* to state that flush requests are not supported.

continued...

Date	Version	Changes
		• Updated the chapter *PCI Express Configuration Information Registers* to state that Extended Tag is not supported. • Updated the GUI screenshot and the list of steps in *Generating the Design Example*. Also added a description for the recommended_pinassignments_s10.txt file. • Updated the chapter *Parameters* to add the **Application Interface Width** parameter and the configurations available when the 64-bit option for that parameter is chosen. • Updated the chapters *Interface Overview*, *Avalon-MM Master Interfaces*, and *Avalon-MM Slave Interfaces* to state that DMA operations are available for 256-bit application interface width but not for 64-bit, and add a row for the 64-bit bursting case to the features table.
November 2017	17.1	Removed **Enable RX-polarity inversion in soft logic** parameter. This parameter is not required for Intel Stratix 10 devices.
November 2017	17.1	Made the following changes to the user guide: • Revised the *Testbench and Design Example for the Avalon-MM Interface* chapter. Although the functions and tasks that implement the testbench have not changed, the organization of these functions and task in files is entirely different than in earlier device families. • Improved descriptions of the DMA registers. • Revised *Generating the Avalon-MM Design* to generate the example design from the `.ip` file. This IP core is now available in the Intel Quartus Prime Pro Edition IP Catalog. • Added definition for `rxm_irq_<n>[15:0]`. This signal is available for the Avalon-MM interface when you enable the CRA port. • Added bit encoding for the Expansion ROM which is supported in this release. • Corrected address range for `Lane Equalization Control Register` in the *Correspondence between Configuration Space Capability Structures and PCIe Base Specification Correspondence between Configuration Space Capability Structures and PCIe Base Specification Description* table. Up to 16 lanes each have a 4-byte register. • Corrected the *Legacy Interrupt Assertion* and *Legacy Interrupt Deassertion* figures. Intel Stratix 10 devices do not support the `app_int_ack` signal. • Updated maximum throughput for L-Tile transceivers from 17.4 Gbps to 26 Gbps. • Removed -3 from the recommended speed grades. • Added note that you must treat BAR0 as non-prefetchable when you enable the internal Descriptor Controller. • Removed description of `testin_zero`. This signal is not a top-level signal of the IP. Made the following changes to the Intel Stratix 10 hard IP for PCI Express IP core: • This IP core is now available in the Intel Quartus Prime Pro Edition IP Catalog.
May 2017	QuartusPrime Pro v17.1 Stratix 10 ES Editions Software	Made the following changes to the IP core: • Added (`*.pof`) support for up to Gen3 x8 variants with an `.ini` file. • Added support for the H-Tile transceiver. • Added support for a Gen3x16 simulation model that you can use in an Avery testbench.

continued...

Date	Version	Changes
		Made the following changes to the user guide: • Added descriptions for DMA Descriptor Controller registers. • Replaced the *Getting Started with the Avalon-MM DMA* static design example with the dynamically generated *Quick Start Guide* design example. • Added *Performance and Resource Utilization* results. • Changed *Read DMA Example* to use larger data block transfers. • Added *Write DMA Example*. • Added *Testbench and Design Example for the Avalon-MM Interface* chapter. • Added reference to *AN-811: Using the Avery BFM for PCI Express Gen3x16 Simulation on Intel Stratix 10 Devices*. • Added figures showing the connections between the Avalon-MM DMA bridge and user application and between the PCIe IP core system interfaces and user application. • Revised *Generation* discussion to match the Quartus Prime Pro – Stratix 10 Edition 17.1 Interim Release design flow. • Added definitions for **Advance**, **Preliminary**, and **Final** timing models. • Fixed minor errors and typos.
October 2016	Quartus Prime Pro – Stratix 10 Edition Beta	Initial release