Important Note: This downloadable PDF of an answer record is provided to enhance its usability and readability. It is important to note that answer records are web-based content that are frequently updated as new information becomes available. You are reminded to visit the Xilinx Technical Support Website and review (Xilinx Answer 57342) for the latest version of this solution.

Please note that this document contains references to Xilinx official documentation, including but not limited to the Virtex®-7 FPGA Gen3 Integrated Block for PCI Express® Core Product Guide (PG023). PG023 is the official source of information and should be used as such if there is any conflict with information included below. Also note that there are screenshots included below of the Vivado® tools, such as the PCIe® Gen 3 Customization GUI. The tools are subject to change and subsequent versions may vary slightly from what is shown in this document.

Introduction

Single Root I/O Virtualization (SR-IOV) is a mechanism defined by PCI-SIG that allows a single Physical PCIe device to be multiple PCIe devices. This is realized with Physical Function (PF) and Virtual Function (VF). PF is a full-fledged PCIe entity with its own entire PCIe configuration space whereas VF is a subset of a PF and contains minimal configuration space definition.

SR-IOV is not a hardware only technology. To implement SR-IOV in a system, it should be supported in both BIOS and in the operating system.

This document details how to simulate SRIOV configuration space read and write by modifying the example design testbench generated with the generation of the Virtex-7 FPGA Gen3 Integrated Block for PCI Express core v2.1 in Vivado Design Suite 2013.2.

SRIOV – Conceptual Example

A suitable example of a SRIOV system would be a system using multiple virtual Network Interface Cards (NIC). SRIOV will help move the virtual NICs out of the software emulation envirnoment and into the physical I/O device itself. This prevents the need for a lot of interactions between a guest and its I/O devices.

The main objective of SRIO-V is to allow the guest operating systems to communicate directly with the physical hardware without having to involve the hypervisor at all. In absence of SIRIO-V, the hypervisor can emulate a NIC. The problem with this is for any traffic to go through the physical NIC port, the hypervisor will have to intercept all the traffic, thus slowing down the data transfer and causing serious performance penalty. With the introduction of SRIOV and allowing guests to access the physical device directly, it reduces the complexity of the hypervisor and boosts runtime performance of the system. This is accomplished in the specification by defining hardware to represent virtual devices, called virtual functions. There can be multiple virtual functions, but there will only be one physical connection out of the network. By doing so, the hypervisor manages the connection but no longer needs to intercept the traffic.
Why Virtual Functions?

One of the main reasons behind using virtual functions is the implementation cost. A device with multiple regular functions with the same backend hardware implementation could work too, but each of those functions would require a full set of configuration space and all the hardware resources that go along with a standalone function, however, this is not the case with virtual function. In virtual function implementation, the hypervisor uses one set of BARs to specify the address range for all the virtual functions associated with the device. Each virtual function has the same amount of address space, so the hypervisor can assign a base address to the first virtual function and then all the subsequent virtual functions would automatically get the next set of contiguous address ranges.

The real benefit for virtual functions is seen when using many functions. The resource limitation may not be an issue when implementing four functions in a device, but what if it is required to implement 64 functions or even 164 functions? This would then start to require a lot of resources. This is where the benefit of virtual functions can be utilized. The hardware resources can be drastically reduced.

SRIOV Configuration Space

Figure 2 shows the SR-IOV capability structure. Details of each register field can be found in Single Root I/O Virtualization and Sharing Specification, Rev 1.1. In this section, only a few registers that you would need to be familiar with for the simulation described in this document have been described.
**VF Enable**
This is bit-0 of the SR-IOV Control register at offset 08h. If VF enable is not set, VFs are disabled.

**VF MSE (Memory Space Enable)**
This bit is similar to Memory Space Enable bit in PCI Command Register. Provided VF Enable is set, the VF memory space will respond only if VF MSE bit is set.

**Initial VFs**
This field indicates the initial number of VFs that were associated with the PF.

**TotalVFs**
It is the maximum number of VFs that could be associated with the PF.

**NumVFs**
NumVF is the actual number of VF is active in the PCI express fabric. These VFs will be visible only after the valid number is written into this register field and VF Enable bit is set.

---

**Figure 2 - Single Root I/O Virtualization (SRIOV) Extended Capabilities**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>PCI Express Extended Capability ID</td>
</tr>
<tr>
<td>04h</td>
<td>SR IOV Capabilities</td>
</tr>
<tr>
<td>08h</td>
<td>SR IOV Status</td>
</tr>
<tr>
<td>0Ch</td>
<td>SR IOV Control</td>
</tr>
<tr>
<td>0Ah</td>
<td>TotalVFs (RO)</td>
</tr>
<tr>
<td>0Bh</td>
<td>InitialVFs (RO)</td>
</tr>
<tr>
<td>10h</td>
<td>Function Dependency Link (RO)</td>
</tr>
<tr>
<td>14h</td>
<td>VF Stride (RO)</td>
</tr>
<tr>
<td>18h</td>
<td>First VF Offset (RO)</td>
</tr>
<tr>
<td>1Ch</td>
<td>VF Device ID (RO)</td>
</tr>
<tr>
<td>20h</td>
<td>Supported Page Sizes (RO)</td>
</tr>
<tr>
<td>24h</td>
<td>System Page Size (RW)</td>
</tr>
<tr>
<td>28h</td>
<td>VF BAR0 (RW)</td>
</tr>
<tr>
<td>2Ch</td>
<td>VF BAR1 (RW)</td>
</tr>
<tr>
<td>30h</td>
<td>VF BAR2 (RW)</td>
</tr>
<tr>
<td>34h</td>
<td>VF BAR3 (RW)</td>
</tr>
<tr>
<td>38h</td>
<td>VF BAR4 (RW)</td>
</tr>
<tr>
<td>3Ch</td>
<td>VF BAR5 (RW)</td>
</tr>
<tr>
<td>40h</td>
<td>VF Migration State Array Offset (RO)</td>
</tr>
</tbody>
</table>
**First VF Offset**

This register field defines the routing ID of the first VF associated with the PF. Routing ID of the first VF is calculated by adding the value in this field to the routing ID of the associated PF.

**VF Stride**

VF Stride along with the First VF Offset value defines the routing ID of the second and the subsequent VFs. Routing ID of the third VF associated with the same PF will be the sum of routing ID of the PF, First VF offset and two times VF Stride.

**SRIOV Packet Header Format**

For configuration for PF and VF in Gen3 hard PCIe block, Config Rd/Wr TLP should be sent out from Requester Request Interface of Root Port (RP) model. And Completion data returns at Requester Completion interface.

![Figure 3 - Requester Request Descriptor Format for Configuration Requests (PG023)](image)

If the SRIOV function is enabled, it is necessary to understand how function ID is assigned to different PFs and VFs. The Virtex-7 FPGA Gen3 Integrated Block for PCI Express supports 2 PFs and 6 VFs.

PF0 must be enabled, which is the default option. PF0’s function ID is 0. If PF1 is enabled, PF1’s function ID will be 1, which cannot be modified.

For VFs assigned to PF0, the function ID of first VF of PF0 (VF0) always starts from 64 (0x40) and increments by 1 for the subsequent VFs. If PF1 is enabled and VFs are assigned to PF1, the function ID of VF of PF1 continues after the last VF function ID of PF0.
For example, if 4 VFs are assigned to PF0 and 2 VFs are assigned to PF1. The following table shows how function IDs are assigned.

<table>
<thead>
<tr>
<th>SRIOV Function ID</th>
<th>PF0</th>
<th>PF1</th>
<th>VF0 (PF0)</th>
<th>VF1 (PF0)</th>
<th>VF2 (PF0)</th>
<th>VF3 (PF0)</th>
<th>VF4 (PF1)</th>
<th>VF5 (PF2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td></td>
</tr>
</tbody>
</table>

**SRIOV Example Design Simulation**

When simulating the out of the box example design with the Virtex-7 FPGA Gen3 Integrated Block for PCI Express core configured for SRIOV, the simulation will run into the following error message. It is because the current example design simulation testbench (v2.1) doesn’t support SRIOV.

```
# (22391000) : Writing Cfg Addr [0x00000000]
# [ 22415000] : Writing Cfg Addr [0x00000000]
# [ 32119000] : Transaction Link Is Up...
# [ 32127000] : TSK_PASSE_FRAME on Transmit
# [ 32371000] : TSK_PASSE_FRAME on receive
# TIMEOUT ERROR in usrapp_tx:TSK_WAIT_FOR_READ_DATA. Completion data never received.
```

The simulation times out in the following configuration read task in pcie_exp_usrapp_tx.v file. The completer ID is not correctly configured in the example design simulation testbench. The configuration reads result in completion with unsupported status. The simulation runs into the above error message when it reaches configuration read in line 418 in pci_exp_usrapp_tx.v file.

```
task TSK_SYSTEM_CONFIGURATION_CHECK;
begin
  error_check = 0;
  // Check Link Speed/Width
  TSK_TX_TYPE0_CONFIGURATION_READ(DEFAULT_TAG, 12'hD0, 4'hF);
  TSK_WAIT_FOR_READ_DATA;
  if (P_READ_DATA[19:16] == MAX_LINK_SPEED) begin
    if (P_READ_DATA[8:0] == ...) end
```

In order to get past the simulation for SRIOV configured core example design, the default configurations read in the test bench should be commented out and new configuration read and write tasks will need to be created. This will be discussed in detail later in this document.

The example design simulation discussed in this document has both PFs enabled with four VFs for PF0 and two VFs for PF1. This document describes the modification in the example design testbench required to read configuration space of the corresponding PFs and VFs.

**SRIOV Core Configuration**

The screenshots below show how to configure the core for enabling SRIOV in the core with both PFs; 4VFs for PF0 and 2VFs for PF1.
Figure 5 - Select Link Speed as Gen3

Figure 6 – Enable SRIOV Capability and Physical Function - 1
Figure 7 - PF0 IDs - Leave Default

Figure 8 - PF0 BAR - Leave Default
There is a known issue regarding “PF1_SRIOV_FIRST_VF_OFFSET” and “VF Device ID for PF1” in v2.1 core configuration GUI, please refer to the following answer records for more details:

http://www.xilinx.com/support/answers/56976.html
http://www.xilinx.com/support/answers/56975.html

Example Design Modification for SRIOV Simulation

The default example design simulation will stop with error when it executes TSK_SYSTEM_CONFIGURATION_CHECK task in pci_exp_usrapp_tx.v file. To get past this error, comment out configuration read tasks as shown below:
Create the following tasks for reading and writing to configuration space of PFs and VFs:

- TSK_TX_TYPE0_CONFIGURATION_READ_PF0
- TSK_TX_TYPE0_CONFIGURATION_READ_PF1
- TSK_TX_TYPE0_CONFIGURATION_READ_VF0
- TSK_TX_TYPE0_CONFIGURATION_READ_VF1
- TSK_TX_TYPE0_CONFIGURATION_READ_VF2
- TSK_TX_TYPE0_CONFIGURATION_READ_VF3
- TSK_TX_TYPE0_CONFIGURATION_READ_VF4 // (VF0 of PF1)
- TSK_TX_TYPE0_CONFIGURATION_READ_VF5 // (VF1 of PF1)
- TSK_TX_TYPE0_CONFIGURATION_WRITE_PFVF

Define the above configuration read tasks exactly the same as the default configuration read task in the example design test bench except for the following changes:

**TSK_TX_TYPE0_CONFIGURATION_READ_PF0 (Bus = 0, Function = 0)**

```vhdl
s_axis_rq_data <= #(Tcq) (128'b0, - - - -) // 4DW unused //256
1'b0, // Force CRC //128
3'b000, // Attributes
1'b0, // Traffic Class
1'b0, // RID Enable to use the Client supplied Bus/Device/Func No

// Start: Modification for SRIOV simulation
// Comment out COMPLETE_ID_CFG and add [8'b00,8'b00],
//COMPLETE_ID_CFG,
//(8'b00,8'b00),
// End: Modification for SRIOV simulation
//ATTR_AIXISEN_IF_ENABLE_CLIENT_TAG ? 8'b00 : tag),
```

**TSK_TX_TYPE0_CONFIGURATION_READ_PF1 (Bus = 0, Function = 1)**

```vhdl
s_axis_rq_tdata <= #(Tcq) (128'b0, // 4DW unused //256
1'b0, // Force CRC //128
3'b000, // Attributes
1'b0, // Traffic Class
1'b0, // RID Enable to use the Client supplied Bus/Device/Func No

// Start: Modification for SRIOV simulation
// Comment out COMPLETE_ID_CFG and add [8'b00,8'b01],
//COMPLETE_ID_CFG,
//(8'b00,8'b01),
// End: Modification for SRIOV simulation
//ATTR_AIXISEN_IF_ENABLE_CLIENT_TAG ? 8'b00 : tag),
```
**TSK_TX_TYPE0_CONFIGURATION_READ_VF0 (Bus = 0, Function = 64)**

\[ s\_axis\_rq\_tdata \leftarrow (Q) \{ 128'b0, // DN unused /256 \]

1'b0, // Force ECRC /128
(set
malformed $3'\text{b001}; 3'\text{b000}), // Attributes
8'b000, // Traffic Class
1'b0, // RID Enable to use the Client supplied Bus/Device/Func No

// Start: Modification for SRIOV simulation
// Comment out COMPLETER_ID_CFS and add \{8'h00,8'h40\},
//COMPLETER_ID_CFS,
\{8'b00,8'b0\},
// End: Modification for SRIOV simulation
(ATTR\_AXISTEN\_IF\_ENABLE\_CLIENT\_TAG \? 8'hCC : tag_),

**TSK_TX_TYPE0_CONFIGURATION_READ_VF1 (Bus = 0, Function = 65)**

\[ s\_axis\_rq\_tdata \leftarrow (Q) \{ 128'b0, // DN unused /256 \]

1'b0, // Force ECRC /128
(set
malformed $3'\text{b001}; 3'\text{b000}), // Attributes
8'b000, // Traffic Class
1'b0, // RID Enable to use the Client supplied Bus/Device/Func No

// Start: Modification for SRIOV simulation
// Comment out COMPLETER_ID_CFS and add \{8'h00,8'h41\},
//COMPLETER_ID_CFS,
\{8'b00,8'b0\},
// End: Modification for SRIOV simulation
(ATTR\_AXISTEN\_IF\_ENABLE\_CLIENT\_TAG \? 8'hCC : tag_),

**TSK_TX_TYPE0_CONFIGURATION_READ_VF2 (Bus = 0, Function = 66)**

\[ s\_axis\_rq\_tdata \leftarrow (Q) \{ 128'b0, // DN unused /256 \]

1'b0, // Force ECRC /128
(set
malformed $3'\text{b001}; 3'\text{b000}), // Attributes
3'b000, // Traffic Class
1'b0, // RID Enable to use the Client supplied Bus/Device/Func No

// Start: Modification for SRIOV simulation
// Comment out COMPLETER_ID_CFS and add \{6'h00,6'h42\},
//COMPLETER_ID_CFS,
\{0'b00,0'b0\},
// End: Modification for SRIOV simulation
(ATTR\_AXISTEN\_IF\_ENABLE\_CLIENT\_TAG \? 8'hCC : tag_),

**TSK_TX_TYPE0_CONFIGURATION_READ_VF3 (Bus = 0, Function = 67)**

\[ s\_axis\_rq\_tdata \leftarrow (Q) \{ 128'b0, // DN unused /256 \]

1'b0, // Force ECRC /128
(set
malformed $3'\text{b001}; 3'\text{b000}), // Attributes
3'b000, // Traffic Class
1'b0, // RID Enable to use the Client supplied Bus/Device/Func No

// Start: Modification for SRIOV simulation
// Comment out COMPLETER_ID_CFS and add \{8'h00,8'h43\},
//COMPLETER_ID_CFS,
\{0'b00,0'b0\},
// End: Modification for SRIOV simulation
(ATTR\_AXISTEN\_IF\_ENABLE\_CLIENT\_TAG \? 8'hCC : tag_),

© Copyright 2012 Xilinx

Xilinx Answer 57342 – SRIOV Example Design Simulation 10
This task is for writing to configuration space of PFs and VFs. Define this task with the same definition as the default configuration write task in the example design testbench except for the following modifications.

```verilog
task TSK_TX_TYPE0_CONFIGURATION_WRITE_PFVF;
input [7:0]  tag ;
input [11:0]  reg_addr ;
input [31:0]  reg_data ;
input [3:0]  first_dw be ;
// Start: Modification for SRIOV simulation
// Add input port id
input [15:0]  id ;
// End: Modification for SRIOV simulation
begin
  //--------------------------------------------------------------\n  if (user_lnk_up_n) begin
    s_axis_rq_tdata <= #(To) [56'b0, // 32b unused] [255:
      // (AXISIENT_IF_QQ_ALIGNMENT_MODE="false")? [reg_data[31:24]], reg_data[23:
        1'b0, // Force CRC //128
        (set_malformed ? 3'b001 : 3'b000), // Attributes
        3'b000, // Traffic Class
        1'b0, // RID Enable to use the Client supplied Bus/Device/Func No
        // Start: Modification for SRIOV simulation
        // Comment out COMPLETER_ID_CFG and add (8'O000,8'h44),
        //COMPLETER_ID_CFG,
        // End: Modification for SRIOV simulation
        (ATTR_AIXSTEN_IF_ENABLE_CLIENT_CFG ? 8'hCC : 8'h44),
      ],
    ]
  end
end
```
Comment out TSK_TX_TYPE0_CONFIGURATION_READ and the associated source in TSK_SYSTEM_CONFIGURATION_CHECK and add the following.

```vhdl
// Set NumVF of PF0
TSK_TX_TYPE0_CONFIGURATION_WRITE_PFN0(DEFAULT_TAG, 12'h210, 32'h00000004, 4'hF, 16'h0000):
DEFAULT_TAG = DEFAULT_TAG + 1;
$display("$t : Set NumVF of PF0: Write 0x0004 to 0x210 of PF0", $realtime);

// Read back NumVF of PF0
TSK_TX_TYPE0_CONFIGURATION_READ_PFN0(DEFAULT_TAG, 12'h210, 4'hF):
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("$t : NumVF register 0x210 data is %h", $realtime, P_READ_DATA);
$display("The readback data should be 0x0004");

// Set NumVF of PF1
TSK_TX_TYPE0_CONFIGURATION_WRITE_PFN0(DEFAULT_TAG, 12'h210, 32'h00000002, 4'hF, 16'h0001):
DEFAULT_TAG = DEFAULT_TAG + 1;
$display("$t : Set NumVF of PF1: Write 0x0002 to 0x210 of PF1", $realtime);

// Read back NumVF of PF1
TSK_TX_TYPE0_CONFIGURATION_READ_PFN1(DEFAULT_TAG, 12'h210, 4'hF):
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("$t : NumVF register 0x210 data is %h", $realtime, P_READ_DATA);
$display("The readback data should be 0x0002");
```

According to the Single Root I/O Virtualization and sharing specification, Rev 1.1, NumVFs should only be written while VF Enable bit is clear (i.e., before writing to the PFn control registers). If NumVFs is written when VF Enable is set, the results are undefined.

### 3.3.7. NumVFs (10h)

NumVFs controls the number of VFs that are visible. SR-PCIIM sets NumVFs as part of the process of creating VFs. This number of VFs shall be visible in the PCI Express fabric after both NumVFs is set to a valid value and VF Enable is Set. A visible VF has a Function number reserved for it but might not exist. If the VF exists, it shall respond to PCI Express transactions targeting them, and shall follow all rules defined by this specification and the base specification. A VF exists if either:

- VF Migration Capable is Clear and the VF number is less than or equal to TotalVFs.
- VF Migration Capable is Set and the associated VF is in either the Active,Available or Dormant,MigrateIn state (see Section 2.4).

The results are undefined if NumVFs is set to a value greater than TotalVFs.

NumVFs may only be written while VF Enable is Clear. If NumVFs is written when VF Enable is Set, the results are undefined.

The initial value of NumVFs is undefined.

**Figure 11 - NumVF Definition - Single Root I/O Virtualization and sharing specification, Rev 1.1**

NumVFs resides at address 210h of the PCI Express configuration space.
For successful operation of virtual functions (VF), the VF Enable bit should be set in the SRIOV control register. VF Enable bit is defined in the Single Root I/O Virtualization and Sharing Specification, Rev 1.1 as follows:

### 3.3.3.1. VF Enable

VF Enable manages the assignment of VFs to the associated PF. If VF Enable is set, the VFs associated with the PF are accessible in the PCI Express fabric. When set, VFs respond to and may issue PCI Express transactions following the rules for PCI Express Endpoint Functions.

Figure 13 – VF Enable Definition - Single Root I/O Virtualization and sharing specification, Rev 1.1

SRIOV control register resides at address 208h of the PCI express configuration space.

The following code to be added in pci_exp_usrapp_tx.v writes to the control registers of both PF0 and PF1 and reads back to verify whether the VF Enable bit has been set or not.
The code below shows reads to configuration space registers of PF0 and PF1 and its associated VFs.

```
// Set SRIOV Control Register of PF0
ISK_I0_CONFIGURATION_WRITE_PFVF(DEFAULT_TAG, 12'h208, 32'hffffff00, 4'hF, 16'h0000);
DEFAULT_TAG = DEFAULT_TAG + 1;
$display("%t: Write all 0xff to 0x208 (Control Register) of PF0", $realtime);

// Read back SRIOV control register of PF0
ISK_I0_CONFIGURATION_READ_PF0(DEFAULT_TAG, 12'h208, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
ISK_WAIT_FOR_READ_DATA;
$display("%t : Read back Control Register of PF0 (0x208) data is %h", $realtime, P_READ_DATA);

// Set SRIOV control register of PF1
ISK_I0_CONFIGURATION_WRITE_PFVF(DEFAULT_TAG, 12'h208, 32'hffffff00, 4'hF, 16'h0001);
DEFAULT_TAG = DEFAULT_TAG + 1;
$display("%t: Write all 0xff to 0x208 (Control Register) of PF1", $realtime);

// Read back SRIOV control register of PF1
ISK_I0_CONFIGURATION_READ_PF1(DEFAULT_TAG, 12'h208, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
ISK_WAIT_FOR_READ_DATA;
$display("%t : Read back Control Register of PF1 (0x208) data is %h", $realtime, P_READ_DATA);
```

The code below shows reads to configuration space registers of PF0 and PF1 and its associated VFs.

```
// Read Device ID/Vendor ID of PF0
ISK_I0_CONFIGURATION_READ_PF0(DEFAULT_TAG, 12'h0, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
ISK_WAIT_FOR_READ_DATA;
$display("%t : DeviceID/Vendor ID of PF0 is %h", $realtime, P_READ_DATA);

// Read Device ID/Vendor ID of PF1
ISK_I0_CONFIGURATION_READ_PF1(DEFAULT_TAG, 12'h0, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
ISK_WAIT_FOR_READ_DATA;
$display("%t : DeviceID/Vendor ID of PF1 is %h", $realtime, P_READ_DATA);
```

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device ID</td>
<td>Vendor ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status</td>
<td>Command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class Code</td>
<td>Rev ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIST</td>
<td>Header</td>
<td>Lat Timer</td>
<td>Cache Ln</td>
</tr>
</tbody>
</table>

**Figure 15 - Device ID/Vendor ID Offset (PG023)**
The following code is from the top level file (pcie3_7x_0.v) simulation which is generated as part of the example design. The parameters and attributes values defined in this file are passed into the example design simulation. The code below shows the VF0 configuration space register values for the parameters shown. The default value for VF0_MSIX_CAP_PBAOFFSET and VF0_MSIX_CAP_TABLEOFFSET are both 'H00000000. The values are modified as shown below just for illustration.

```verilog
// Read Total VF and Initial VFs of PF0
TSK_TX_TYPE0_CONFIGURATION_READ_PF0(DEFAULT_TAG, 12'h20C, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("%t: Total VF / Initial VFs of PF0 is %h", $realtime, P_READ_DATA);

// Read Total VF and Initial VFs of PF1
TSK_TX_TYPE0_CONFIGURATION_READ_PF1(DEFAULT_TAG, 12'h20C, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("%t: Total VF / Initial VFs of PF1 is %h", $realtime, P_READ_DATA);

// Read VF Stride and First VF Offset of PF0
TSK_TX_TYPE0_CONFIGURATION_READ_PF0(DEFAULT_TAG, 12'h214, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("%t: VF Stride and First VF Offset of PF0 is %h", $realtime, P_READ_DATA);

// Read VF Stride and First VF Offset of PF1
TSK_TX_TYPE0_CONFIGURATION_READ_PF1(DEFAULT_TAG, 12'h214, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("%t: VF Stride and First VF Offset of PF1 is %h", $realtime, P_READ_DATA);

// Read Supported Page Sizes of PF0
TSK_TX_TYPE0_CONFIGURATION_READ_PF0(DEFAULT_TAG, 12'h214, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("%t: Supported Page Sizes PF0 is %h", $realtime, P_READ_DATA);

// Read System Page Size of PF0
TSK_TX_TYPE0_CONFIGURATION_READ_PF0(DEFAULT_TAG, 12'h214, 4'hF);
DEFAULT_TAG = DEFAULT_TAG + 1;
TSK_WAIT_FOR_READ_DATA;
$display("%t: System Page Size of PF0 is %h", $realtime, P_READ_DATA);
```
To read these values from the configuration space of VF0, use TSK_TX_TYPE0_CONFIGURATION_READ_VF0 task by passing the correct register address. The addresses of these registers are shown in Figure 16.

<table>
<thead>
<tr>
<th>Optional(3)</th>
<th>MSI-X Control</th>
<th>NxtCap</th>
<th>MSI-X Cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table Offset</td>
<td>Table BIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PBA Offset</td>
<td>PBA BIR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>PE Capability</td>
<td>NxtCap</td>
<td>PE Cap</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 16 - VF0 Table and PBA Offset Configuration Space Address (PG023)**

Add the code below in pci_exp_usrapp_tx.v to read MSI-X PBA Offset and MSI-X Table Offset values of VF0 of PF0 and PF1.
SRIOV Simulation Result

Below is the ModelSim console log after running the example design simulation after modification in the pci_exp_ursapp_tx.v file described in the previous section.

```plaintext
# Running default test {sample_smoke_test0}......
# [ 0] : System Reset Is Asserted...
# [ 4995000] : System Reset Is De-asserted...
# [ 22351001] : Transaction Reset Is De-asserted...
# [ 22355000] : Writing Cfg Addr [0x00000001]
# [ 22391000] : Reading Cfg Addr [0x00040082]
# [ 22415000] : Writing Cfg Addr [0x00040082]
# [ 32119000] : Transaction Link Is Up...
# [ 32119000] : Starting SRIOV simulation
# [ 32127000] : TSKPARSEFRAME on Transmit
# [ 32127000] : Set NumVF of PF0: Write 0x0004 to 0x210 of PF0
# [ 32135000] : TSKPARSEFRAME on Transmit
# [ 32375000] : TSKPARSEFRAME on Receive
# [ 32387000] : TSKPARSEFRAME on Receive
# [ 34135000] : NumVF register 0x210 data is 00000004
# The readback data should be 0x0004
# [ 34141000] : TSKPARSEFRAME on Transmit
# [ 34143000] : Set NumVF of PF1: Write 0x0002 to 0x210 of PF1
# [ 34151000] : TSKPARSEFRAME on Transmit
# [ 34391000] : TSKPARSEFRAME on Receive
# [ 34399000] : TSKPARSEFRAME on Receive
# [ 36151000] : NumVF register 0x210 of PF1 data is 00010002
```

© Copyright 2012 Xilinx
# The readback data should be 0x0002
# [36159000] : TSKPARSE_FRAME on Transmit
# [36159000] : Write all 0xff to 0x208 (Control Register) of PF0
# [36167000] : TSKPARSE_FRAME on Transmit
# [36407000] : TSKPARSE_FRAME on Receive
# [36419000] : TSKPARSE_FRAME on Receive
# [38167000] : Read back Control Register of PF0 (0x208) data is 00000019
# [38175000] : TSKPARSE_FRAME on Transmit
# [38175000] : Write all 0xff to 0x208 (Control Register) of PF1
# [38183000] : TSKPARSE_FRAME on Transmit
# [38427000] : TSKPARSE_FRAME on Receive
# [38435000] : TSKPARSE_FRAME on Receive
# [40183000] : Read back Control Register of PF1 (0x208) data is 00000009
# [40191000] : TSKPARSE_FRAME on Transmit
# [40449000] : TSKPARSE_FRAME on Receive
# [42191000] : DeviceID/Vendor ID of PF0 is 703810ee
# [42199000] : TSKPARSE_FRAME on Transmit
# [42443000] : TSKPARSE_FRAME on Receive
# [44190000] : DeviceID/Vendor ID of PF1 is 703810ee
# [44207000] : TSKPARSE_FRAME on Transmit
# [44455000] : TSKPARSE_FRAME on Receive
# [46207000] : Total VF / Initial vrs of PF0 is 00040004
# [46215000] : TSKPARSE_FRAME on Transmit
# [46463000] : TSKPARSE_FRAME on Receive
# [48215000] : Total VF / Initial VFs of PF1 is 00020002
# [48223000] : TSKPARSE_FRAME on Transmit
# [48467000] : TSKPARSE_FRAME on Receive
# [50223000] : VF Stride and First VF Offset of PF0 is 00010044
# [50231000] : TSKPARSE_FRAME on Transmit
# [50475000] : TSKPARSE_FRAME on Receive
# [52231000] : VF Stride and First VF Offset of PF1 is 00010044
# [52239000] : TSKPARSE_FRAME on Transmit
# [52487000] : TSKPARSE_FRAME on Receive
# [54236000] : Supported Page Sizes PF0 is 00010044
# [54247000] : TSKPARSE_FRAME on Transmit
# [54499000] : TSKPARSE_FRAME on Receive
# [56247000] : System Page Size of PF0 is 00010044
# [56255000] : TSKPARSE_FRAME on Transmit
# [56499000] : TSKPARSE_FRAME on Receive
# [58255000] : MSI-X PBA Offset value of VF0 88ffffff8
# [58263000] : TSKPARSE_FRAME on Transmit
# [58511000] : TSKPARSE_FRAME on Receive
# [60263000] : MSI-X Table Offset value of VF0 8ffffffff8
# [60271000] : TSKPARSE_FRAME on Transmit
# [60519000] : TSKPARSE_FRAME on Receive
# [62271000] : MSI-X PBA Offset value of VF5 (VF1 of PF1) 88ffffff8
# [62279000] : TSKPARSE_FRAME on Transmit
# [62527000] : TSKPARSE_FRAME on Receive
# [64279000] : MSI-X Table Offset value of VF5 (VF1 of PF1) 8ffffffff8
# SIOV Simulation Ends here

© Copyright 2012 Xilinx
Gotchas!

- Make sure VF enable bit is set in the SRIOV control register as shown below. This is at Offset 208h and is DWORD aligned. When sending TLP requests, make sure VF memory space enable bit is set as well.

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Register Description</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VF Enable – Enables/Disables VFs. Default value is 0b.</td>
<td>RW</td>
</tr>
<tr>
<td>1</td>
<td>VF Migration Enable – Enables/Disables VF Migration Support. Default value is 0b.</td>
<td>RW or RO (see description)</td>
</tr>
<tr>
<td></td>
<td>See Section 3.3.3.2.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>VF Migration Interrupt Enable – Enables/Disables VF Migration State Change Interrupt. Default value is 0b.</td>
<td>RW</td>
</tr>
<tr>
<td>3</td>
<td>VF MSE – Memory Space Enable for Virtual Functions. Default value is 0b.</td>
<td>RW</td>
</tr>
<tr>
<td>4</td>
<td>ARI Capable Hierarchy</td>
<td>RW or RO (see description)</td>
</tr>
<tr>
<td></td>
<td><em>PCI Express Endpoint:</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The Device is permitted to locate VFs in Function numbers 8 to 255 of the captured Bus Number. Default value is 0b. This field is RW in the lowest numbered PF of the Device and is Read Only Zero in all other PFs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><em>Root Complex Integrated Endpoint:</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Not applicable – must hardwire the bit to 0b.</td>
<td></td>
</tr>
<tr>
<td>15:5</td>
<td>Reserved – These fields are currently reserved.</td>
<td>RsvdP</td>
</tr>
</tbody>
</table>

- Make sure the NumVFs field in the SR-IOV Extended Capabilities space is set correctly. This field must be set by the software to the number of VFs that it wants to enable for each PF. This field cannot be changed if VF Enable bit in the SR-IOV control register is set to ‘1’. This field can be accessed at offset address of 84H, DWORD aligned. The 15:0 of the read value gives the NumVFs.

- In pcie3_7x_v1_3_sim.v, make sure ARI_CAP_ENABLE is set to “TRUE”.

```vhdl
pcie3_7x_v1_3_pcie_3_0_7vx #(  ARI_CAP_ENABLE ("TRUE"),  .AXISTEN_IF_CC_ALIGNMENT_MODE("FALSE"),  .AXISTEN_IF_CQ_ALIGNMENT_MODE("FALSE"),  .AXISTEN_IF_ENABLE_CLIENT_TAG("TRUE"),  .AXISTEN_IF_RC_ALIGNMENT_MODE("FALSE"),  .AXISTEN_IF_RC_STRADDLE("TRUE"),  .AXISTEN_IF_RQ_ALIGNMENT_MODE("FALSE"),
```

© Copyright 2012 Xilinx
SRIOV Packet/Signal Analysis

Figure 17 shows the entire overview of the example design simulation with the modification required to access the configuration space of PFs and VFs.

Configuration Write to NumVF field of PF0

Refer to the Figures 18 and 19 for detailed analysis of the configuration write to NumVF field of PF0.
The ‘Req Type’ value of 1010 indicates, this is the configuration write transaction. The description for each Request Type value is shown in Figure 20.

<table>
<thead>
<tr>
<th>Request Type (binary)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Memory Read Request</td>
</tr>
<tr>
<td>0001</td>
<td>Memory Write Request</td>
</tr>
<tr>
<td>0010</td>
<td>I/O Read Request</td>
</tr>
<tr>
<td>0011</td>
<td>I/O Write Request</td>
</tr>
<tr>
<td>0100</td>
<td>Memory Fetch and Add Request</td>
</tr>
<tr>
<td>0101</td>
<td>Memory Unconditional Swap Request</td>
</tr>
<tr>
<td>0110</td>
<td>Memory Compare and Swap Request</td>
</tr>
<tr>
<td>0111</td>
<td>Locked Read Request (allowed only in Legacy Devices)</td>
</tr>
<tr>
<td>1000</td>
<td>Type 0 Configuration Read Request (on Requester side only)</td>
</tr>
<tr>
<td>1001</td>
<td>Type 1 Configuration Read Request (on Requester side only)</td>
</tr>
<tr>
<td>1010</td>
<td>Type 0 Configuration Write Request (on Requester side only)</td>
</tr>
<tr>
<td>1011</td>
<td>Type 1 Configuration Write Request (on Requester side only)</td>
</tr>
<tr>
<td>1100</td>
<td>Any message, except ATS and Vendor-Defined Messages</td>
</tr>
<tr>
<td>1101</td>
<td>Vendor-Defined Message</td>
</tr>
<tr>
<td>1110</td>
<td>ATS Message</td>
</tr>
<tr>
<td>1111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Figure 20 – Request Type Description (PG023)

Configuration Read to NumVF of PF0

Figure 21 - Configuration Read to NumVF of PF0 (Requester Request Interface)
Figure 22 - Configuration Read to NumVF of PF0 – Requester Completion Interface Header Descriptor Analysis

In Figure 23, the byte count field is ‘4’ which should be the case in the completion header descriptor format for configuration write as defined in Figure 25.

© Copyright 2012 Xilinx
These 13 bits can have values in the range of 0 – 4096 bytes. If a Memory Read Request is completed using a single Completion, the Byte Count value indicates Payload size in bytes. This field must be set to 4 for I/O read Completions and I/O write Completions. The byte count must be set to 1 while sending a Completion for a zero-length memory read, and a dummy payload of 1 Dword must follow the descriptor.

For each Memory Read Completion, the Byte Count field must indicate the remaining number of bytes required to complete the Request, including the number of bytes returned with the Completion.

If a Memory Read Request is completed using multiple Completions, the Byte Count value for each successive Completion is the value indicated by the preceding Completion minus the number of bytes returned with the preceding Completion.

Figure 25 - Byte Count field Definition for Configuration/Memory Request Completion Descriptor Format

Completion for Configuration Read to NumVF of PF0

<table>
<thead>
<tr>
<th>28:16</th>
<th>Byte Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 26 - Completion for Configuration Read to NumVF of PF0 (Requester Completion Interface)

Figure 27 - Completion for Configuration Read to NumVF of PF0 – Requester Completion Interface Header

Descriptor Analysis
Configuration Read to Device ID/Vendor ID of PF0

Figure 28 - Configuration Read to Device ID/Vendor ID of PF0

Figure 29 - Configuration Read to Device ID/Vendor ID of PF0 – Requester Request Interface Header Descriptor

Completion for Configuration Read to Device ID/Vendor ID of PF0

Figure 30 - Completion for Configuration Read to Device ID/Vendor ID of PF0 (Requester Request Interface)
Figure 31 – Completion for Configuration Read to Device ID/Vendor ID of PF0 – Requester Completion Interface Header Descriptor Analysis

Configuration Read to “Total VF and Initial VFs” of PF1

Figure 32 - Configuration Read to “Total VF and Initial VFs” of PF1 (Requester Request Interface)

Figure 33 - Configuration Read to “Total VF and Initial VFs” of PF1 - Requester Request Interface Header Description Analysis
**SRIO-V Application Note: XAPP1177**

This document introduced the SRIO-V concept and how to simulate SRIO-V with the example design generated with the generation of the core. For more comprehensive details and a reference design on SRIO-V, check Xilinx Application Note: XAPP1177. This application note demonstrates the SR-IOV capability of the Xilinx Virtex-7 PCI Express Gen3 Integrated Block core. The application notes illustrates how to create a PCI Express x8 Gen3 Endpoint design configured for two Physical Functions (PF) and six Virtual Functions (VF). The reference design is targeted at a Xilinx Virtex®-7 VC709 development boards and has been hardware-validated on a system with SR-IOV capability by performing Programmed I/O reads and writes to all functions using the supplied drivers.

**References**

1. Virtex-7 FPGA Gen3 Integrated Block for PCI Express v2.1  
   [http://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v2_1/pg023_v7_pcie_gen3.pdf](http://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v2_1/pg023_v7_pcie_gen3.pdf)

**Revision History**

11/13/2013 – Initial release  
© Copyright 2012 Xilinx