XAPP1177 (v1.0) November 15, 2013

Designing with SR-IOV Capability of Xilinx Virtex-7 PCI Express Gen3 Integrated Block

Author: Vivek Surabhi

Summary

This application note demonstrates the Single Root I/O Virtualization (SR-IOV) capability of the Xilinx Virtex®-7 FPGA PCI Express® Gen3 Integrated Block. It explains the key concepts of SR-IOV and provides details about configuring the SR-IOV capability. This document illustrates how to create a PCI Express x8 Gen3 Endpoint design configured for two Physical Functions (PF) and six Virtual Functions (VF) as shown in Figure 1. The reference design is targeted at a Xilinx Virtex-7 FPGA VC709 Connectivity Kit 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.

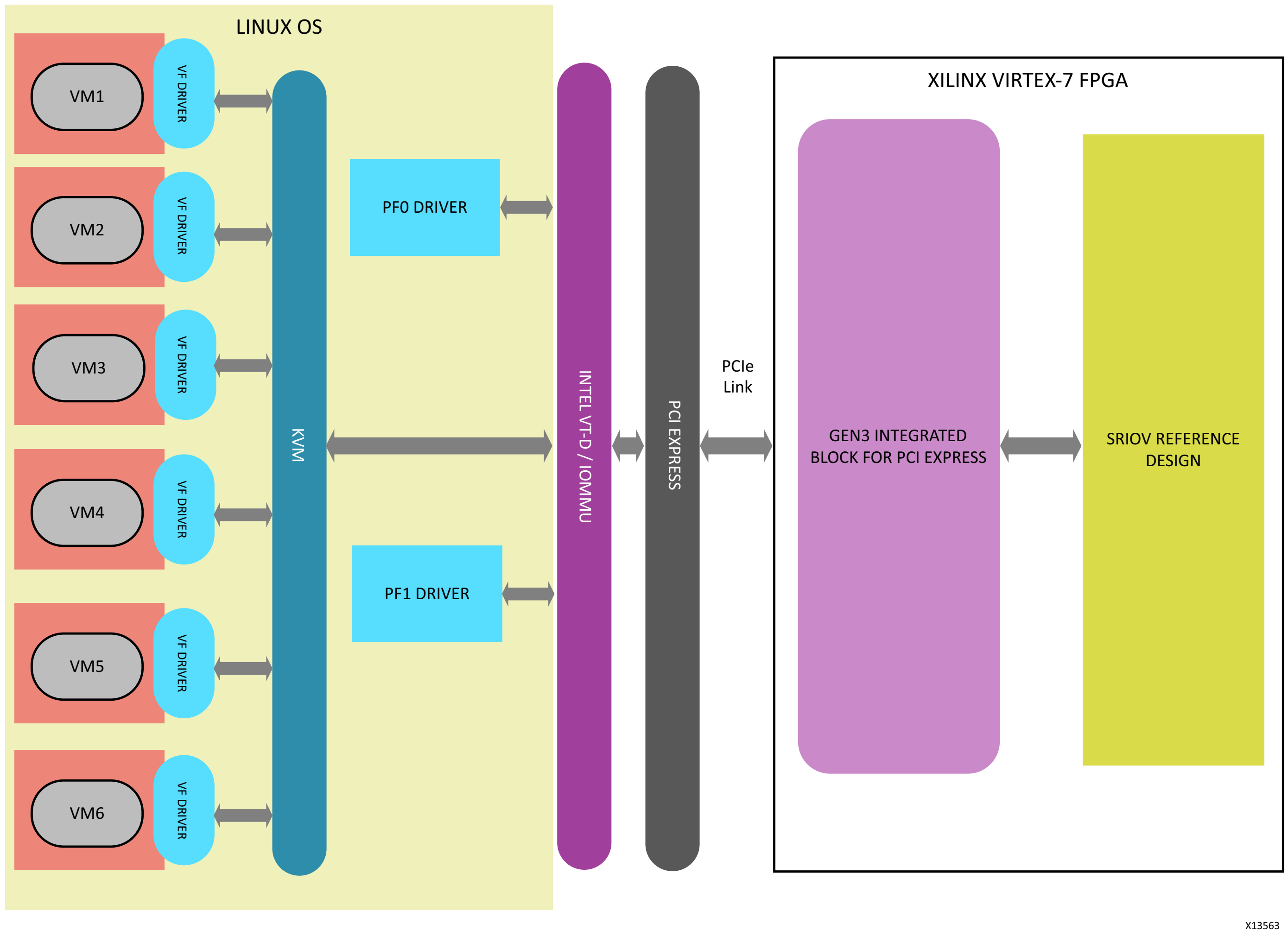


Figure 1: **SR-IOV System Overview**

The SR-IOV reference design and the Linux kernel-mode driver source code are included with this application note.

The evaluation of SR-IOV capability can be a complex process with many variations seen between different operating systems and system platforms. This document establishes a baseline system configuration and provides the necessary software to quickly bring up and evaluate the SR-IOV features of the Virtex-7 FPGA PCI Express Gen3 Integrated Block.

Note: The SR-IOV drivers have been tested only on the Linux operating system (OS) versions described in this document. While there are no known issues using these drivers on other Linux versions, success cannot be guaranteed.

Introduction

Virtualization can be defined as a means for multiple operating systems (virtual systems) running on a host system to share a single I/O device. Virtualization adoption is increasing in industries such as data centers, where physical hardware is shared to create a large number of logical resources used across various virtual machines. With the increasing need to guarantee service and the increase in platform costs, virtualization provides resource optimization and maximum uptime in environments with dynamic workloads. Though widely used, there is no standard implementation for virtualization with the industry using software emulation, direct assignment and other hybrid approaches.

The Virtual Machine Monitor (VMM), also known as a hypervisor, creates and manages virtual machines. The VMM also enables the sharing of the physical I/O devices across the virtual platforms. In a software-based virtualization system, the VMM is involved in all datapath transactions (software-based switching), which consumes significant CPU bandwidth and thereby reduces the system throughput.

The PCI-SIG® Single Root I/O Virtualization and Sharing (SR-IOV) specification [Ref 4] standardizes the method for bypassing the VMM involvement in datapath transactions and allows a single PCI Express endpoint to appear as multiple separate PCI Express endpoints. SR-IOV classifies the functions as:

- **Physical Functions (PF)**: Full featured PCIe® functions which include SR-IOV capabilities among others.
- **Virtual Functions (VF)**: PCIe functions featuring configuration space with Base Address Registers (BARs) but lacking the full configuration resources and controlled by the PF configuration. The main role of the VF is data transfer.

The involvement of VMMs in datapath transactions such as packet classification and address translation is bypassed by providing independent DMA, memory space and interrupts for each of the VFs. The VF configuration space can be mapped to the virtual systems memory which enables direct access to the VF physical address including allowing DMA to the virtual system using Intel® Virtualization Technology for Directed I/O (VT-d). VT-d is responsible for I/O device assignments, DMA and Interrupt remapping.

SR-IOV Support

The Virtex-7 FPGA PCI Express Gen3 Integrated Block supports:

- Two Physical Functions (PFs)
- Six Virtual Functions (VFs)

The two PFs support Alternate Routing ID (ARI) capability and implement SR-IOV capability in Extended PCIe Configuration Space.

The SR-IOV specification addresses the scalability issue of virtual system environments by providing the capability for each PCI Express endpoint device to support up to 256 functions. This is achieved by the ARI specification which enables a single PCI Express device to support more than eight functions by merging the device number and function number fields of a Routing ID.

Routing IDs are 16-bit identifiers that are composed of an 8-bit Bus Number, 5-bit Device Number and 3-bit Function Number as shown in Figure 3. In systems that support ARI, the Routing ID is interpreted as an 8-bit bus number and 8-bit function number as shown in Figure 4. This interpretation supports up to 256 functions instead of the traditional 8.

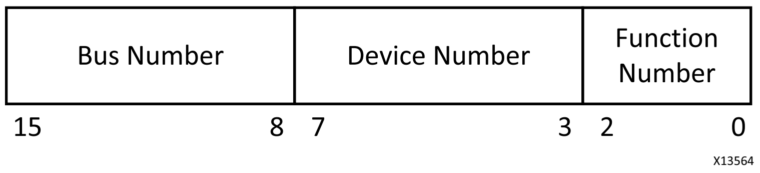


Figure 3: **Non-ARI Routing ID**

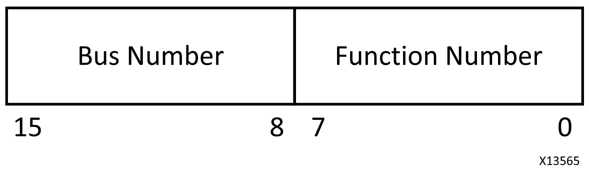
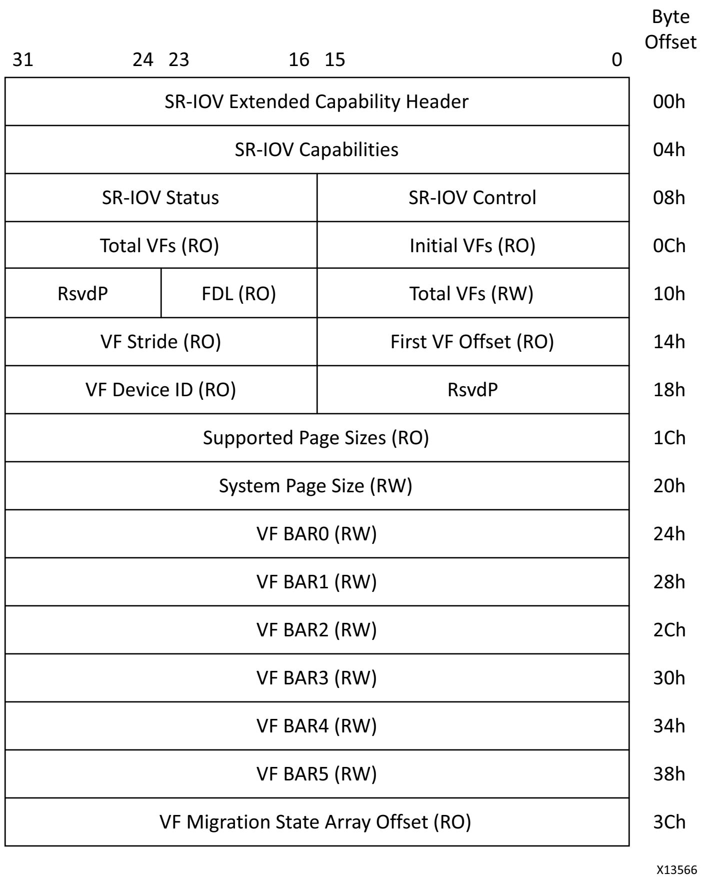


Figure 4: **ARI Routing ID**

The SR-IOV capability structure for a PF is shown in Figure 5.



31 – 24	23 – 16	15 – 0	Byte Offset
SR-IOV Extended Capability Header			00h
SR-IOV Capabilities			04h
SR-IOV Status		SR-IOV Control	08h
Total VFs (RO)		Initial VFs (RO)	0Ch
RsvdP	FDL (RO)	Total VFs (RW)	10h
VF Stride (RO)		First VF Offset (RO)	14h
VF Device ID (RO)		RsvdP	18h
Supported Page Sizes (RO)			1Ch
System Page Size (RW)			20h
VF BAR0 (RW)			24h
VF BAR1 (RW)			28h
VF BAR2 (RW)			2Ch
VF BAR3 (RW)			30h
VF BAR4 (RW)			34h
VF BAR5 (RW)			38h
VF Migration State Array Offset (RO)			3Ch

X13566

Figure 5: **SR-IOV Capability Structure**

Virtual functions do not exist on power-on and it is the function of the system software to discover and enable VFs based on the system capability. The VF support is discovered by scanning the SR-IOV capability structure for each PF.

The register fields Total VFs (0Eh) and Initial VFs (0Ch) are read-only fields (set by core attributes) and indicate the maximum number of VFs associated with the PF. Each PF can have zero or more VFs associated with them. Because the Virtex-7 FPGA PCI Express Gen3 Integrated Block supports a maximum of six VFs between two PFs, the largest value in the Total VFs field (0Eh) can be 6. Based on the values in these fields along with knowledge of the system capabilities, the system software writes the appropriate value into the Num VFs field (10h) which determines the number of VFs visible in the PCIe logic after the VFs are enabled.

The register fields VF Stride (16h) and VF Offset (14h) are fixed for the Virtex-7 FPGA PCI Express Gen3 Integrated Block. VF Offset is fixed to 64 (decimal) and VF Stride is fixed to 1, which means that all Virtual Functions are in the function number range of 64 to 69.

For example,

- If PF0 has four VFs attached and PF1 has two VFs attached, the associated VF function numbers are as shown in Table 1.

Table 1: **VF Function Number Assignment**

Physical Function	Virtual Function	Function Number
PF0	VF1	64
PF0	VF2	65
PF0	VF3	66
PF0	VF4	67
PF1	VF5 (VF1 to PF1)	68
PF1	VF6 (VF2 for PF1)	69

- If PF0 has two VFs attached and PF1 has two VFs attached, the associated VF function numbers are as shown in Table 2.

Table 2: **VF Function Number Assignment**

Physical Function	Virtual Function	Function Number
PF0	VF1	64
PF0	VF2	65
PF1	VF3 (VF1 to PF1)	66
PF1	VF4 (VF2 for PF1)	67

All Virtual Functions associated with a Physical Function support a common Device ID and Device Type. The Device Type of the VF is dependent on the associated PF whereas the Device ID field (1Ah) is configurable (set by core attributes).

The Supported Page Sizes field (1Ch) indicates all the page sizes supported by the PF and, as required by the SR-IOV specification, the Virtex-7 FPGA PCI Express Gen3 Integrated Block supports 4 KB, 8 KB, 64 KB, 256 KB, 1 MB and 4 MB page sizes. Based on the Supported Page Size field, the system software sets the System Page Size field (20h) which is used to map the VF BAR memory addresses. Each VF BAR address is aligned to the system page boundary.

For example, if the VF BAR0 size is set to 4 KB and system page size is set by the system software to 8 KB in a configuration that supports four VFs, the address mapping is as shown in Figure 6. Even though each function needs only 4 KB address space, the start addresses are aligned to 8 KB boundaries.

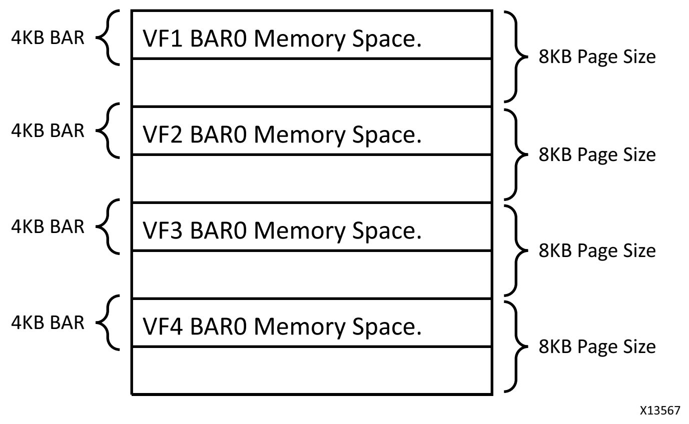


Figure 6: **BAR Address Mapping With Respect to System Page Size**

Each Virtual Function supports up to six 32-bit BARs or three 64-bit BARs. Virtual Function BARs can be configured without any dependency on the settings of the associated Physical Functions BARs, but all virtual functions associated with the physical function share the same settings. Virtual Function BARs do not support I/O space and hence have to be configured to map to memory space. VF BARs can be either 64-bit or 32-bit and can be prefetchable:

- When configuring the core as an Endpoint for PCIe, 64-bit addressing is supported for all SR-IOV BARs (except BAR5) that have the prefetchable bit set.
- 32-bit addressing is permitted for all SR-IOV BARs that do not have the prefetchable bit set.
- When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.

The Function Dependency Link describes the dependencies between PFs. Because the functions supported by the Virtex-7 FPGA PCI Express Gen3 Integrated Block are independent of each other, this field should be set by the core attributes to contain its own function number.

The Virtex-7 FPGA PCI Express Gen3 Integrated Block does not support VF Migration for SR-IOV. Hence, all fields in the SR-IOV capability structure associated with VF Migration are read-only fields with value of zero.

Supported VF Capabilities

All virtual functions have their own configuration space including the PCIe Capability Structure. Xilinx VFs support MSI and MSI-X interrupts. The virtual functions interrupt support can be set independently of the associated PF. Each VF supports MSI, MSI-X or both based on their configuration.

Virtual functions support:

- Pending Bit Array structure for MSI-X. For MSI-X, the table offset and PBA offset is relative to the VF memory address space. However, VFs do not support the per-vector masking capability structure MSI.
- Power Management capability with D1 and D3 hot support. If a VF does not implement the Power Management Capability, then the VF behaves as if it had been programmed into the equivalent power state of its associated PF. If a VF implements the Power Management Capability, the functionality is defined in the *PCI Express Base Specification Rev 3.0* [Ref 3].
- Baseline error reporting capability. Advanced error reporting (AER) is supported only if the associated PF supports AER. Also, the VFs use the Virtual channels of the associated PFs. As such, VFs do not have any virtual channel capabilities.

The capabilities that do not apply to VFs are Resizable BAR, Device Serial Number and Latency Tolerance Reporting.

For more details on configuring PF SR-IOV capabilities and VF capabilities, see Generating and Customizing the Core.

SR-IOV Reference Design Requirements

The hardware, software and system required to generate the reference systems are:

Hardware Requirements

- Xilinx Virtex-7 FPGA VC709 Connectivity Kit (xc7vx690tffg1761-2)
- USB cable (Type A connector to Type B connector)
- JTAG platform USB cable
- AC Power Adapter for the board

Software Requirements

- Vivado Design Suite 2013.3

System Requirements for SR-IOV

- Intel Processor with Gen3 PCIe and VT-d (Intel Core i7-3820 Sandy Bridge-E 3.6GHz LGA 2011 Quad-Core Processor or similar)
- Motherboard with VT-d BIOS support (ASUS RAMPAGE IV GENE LGA 2011 Intel X79 Micro ATX Motherboard or similar)
- Red Had Enterprise Linux version 6.2 with KVM Hypervisor for Host Operating System
- Fedora 16 or later for Guest Operating System

Resource Utilization

Table 3 provides the post-implementation utilization details for the PCIe x8 Gen3 SR-IOV Reference Design for a two PF and six VF configuration.

Table 3: **Reference Design Post-Implementation Utilization**

Resource	Utilization	Available	Utilization Percentage
FF	3865	864736	1
LUT	3795	432368	1
I/O	5	850	1
BRAM	15	2940	1
BUFG	5	32	16
MMCM	1	20	5
GT	10	45	22

Reference Design Specifics

The hardware design includes the Virtex-7 FPGA PCI Express Gen3 Integrated Block.

The reference design has been implemented and verified in hardware on the Virtex-7 FPGA VC709 Connectivity Kit. The reference design matrix is shown in Table 4.

Table 4: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Vivek Surabhi
Target devices (stepping level, ES, production, speed grades)	xc7vx690tffg1761-2 GES & Production
Source code provided	Yes
Source code format	Verilog
Implementation	
Synthesis software tools/version used	Vivado Design Suite 2013.3
Implementation software tools/versions used	Vivado Design Suite 2013.3
Software development tools/versions used	NA
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	VC709

The reference design for this application note can be downloaded from:

https://secure.xilinx.com/webreg/clickthrough.do?cid=345714

Table 5 describes the files provided in this package.

Table 5: **Design Files**

File	Description
readme.txt	Readme file describing the contained files.
sriov_reference_design/	SR-IOV reference design Verilog source files.
pcie_app_7vx.v	Top-level wrapper that integrates with the Vivado Design Suite example design.
sriov_top.v	SR-IOV reference design top wrapper.
sriov_pio.v	SR-IOV programmed i/o module wrapper.
sriov_rx_engine.v	Reference design receive unit.
sriov_tx_engine.v	Reference design transmit unit.
sriov_mem_access.v	Reference design memory unit top wrapper.
sriov_mem.v	Reference design memory unit instantiation.
sriov_pm_ctrl.v	Reference design power management control.
Linux_Driver_PF0/	Linux driver folder for PF0.
README	Procedure to compile and load the driver.
xpcie.c	XPCIE device driver.
met.cpp	Example memory endpoint test. Modify this file to change the test or iteration counts.

Table 5: **Design Files *(Cont'd)***

File	Description
make_device	Script that creates the xpcie device on the system. This must be run before loading the device driver.
makefile	Makefile compiles `xpcie.c` into a loadable `xpcie.ko` and compiles `met.cpp` into MET executable.
Linux_Driver_PF1/	Linux driver folder for PF1.
README	Procedure to compile and load the driver.
xpcie_pf1.c	XPCIE device driver.
met.cpp	Example memory endpoint test. Modify this file to change the test or iteration counts.
make_device	Script that creates the xpcie device on the system. This must be run before loading the device driver.
makefile	Makefile compiles `xpcie.c` into a loadable `xpcie.ko` and compiles `met.cpp` into MET executable.
Linux_Driver_VF/	Linux driver folder for VF.
README	Procedure to compile and load the driver.
xpcie.c	XPCIE device driver.
met.cpp	Example memory endpoint test. Modify this file to change the test or iteration counts.
make_device	Script that creates the xpcie device on the system. This must be run before loading the device driver.
makefile	Makefile compiles `xpcie.c` into a loadable `xpcie.ko` and compiles `met.cpp` into MET executable.

Reference Design Overview

The SR-IOV Reference Design builds on top of the Programmed Input/Output (PIO) example design, which is a completer application for PCI Express Core that is delivered along with the core as an example design. Figure 7 illustrates the reference design.

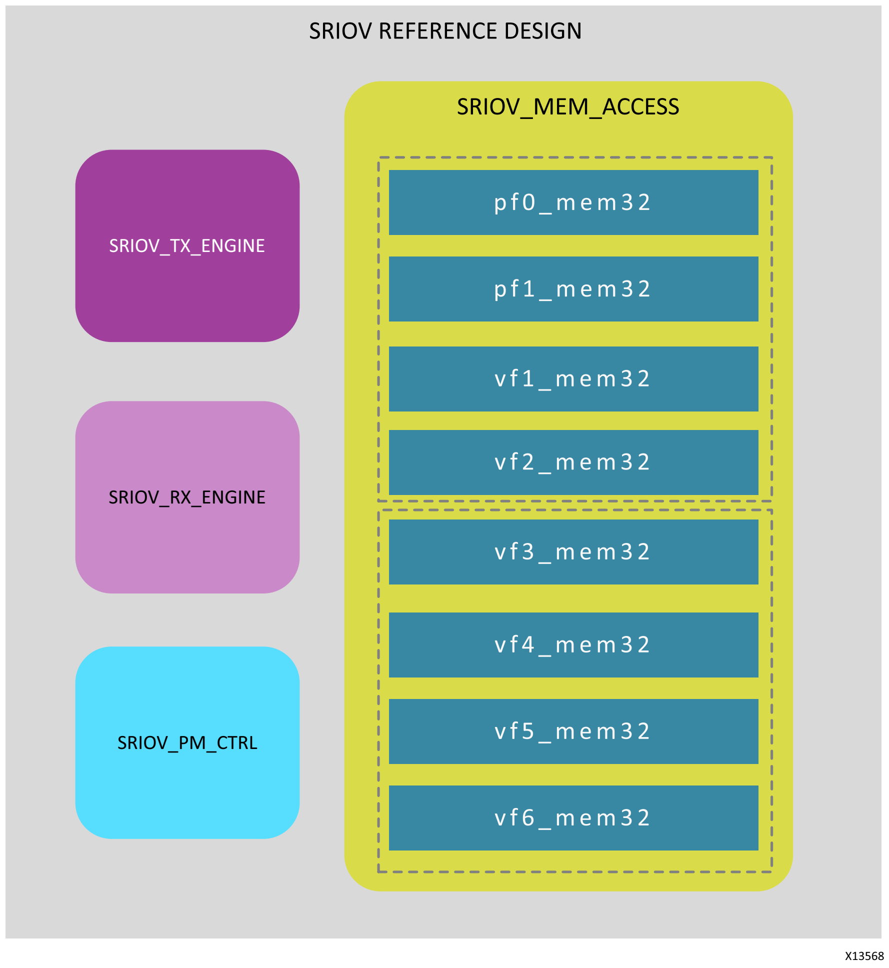


Figure 7: **SR-IOV Reference Design**

The SR-IOV reference design supports:

- Two PF and six VF PCI Express x8 Gen3 core configurations
- 256-bit AXI4-Stream interface
- 4 KB, 32-bit memory space BAR for each individual function (BAR0)
- Single Dword Read and Write transactions to 32-bit memory mapped address

Programmed I/O (PIO) transactions are used by a PCI Express system host to access Memory Mapped I/O (MMIO) and Configuration Mapped I/O (CMIO) locations in the PCI Express logic. Endpoints for PCI Express accept Memory and I/O Write transactions and respond to Memory and I/O Read transactions with Completion Data transactions.

The SR-IOV reference design implements only 2,048 bytes of target space in the FPGA Block RAM for each individual function of the Endpoint. If the corresponding BAR is configured to a wider aperture, accesses beyond the 2 KB limit wrap around the 2 KB memory space. This target space is accessible through single Dword Memory Read 32 and Memory Write 32 TLPs. The SR-IOV reference design generates a completion with one Dword of payload in response to a valid Memory Read 32 TLP request presented to it by the core.

Memory Read or Memory Write TLPs of lengths larger than one Dword are not processed correctly by the PIO design. However, the core does accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than one Dword, the TLP is received from the core and discarded. No corresponding completion is generated.

When the Endpoint for PCIe receives a Memory Write TLP, the TLP destination address and transaction type are compared with the values in the core BARs. If the TLP passes this comparison check, the core passes the TLP over the Completer Requester AXI4-Stream interface of the PIO design. The Completer Request Descriptor is shown in Figure 8 and has the appropriate BAR ID (bits [114:112]) and Target Function (bits [111:104]) to indicate the specific function (PF/VF) and its BAR that matched the incoming TLP. On reception, the reference design RX_ENGINE processes the incoming Write TLP and extracts the data and relevant address fields so that it can pass this along to the MEM_ACCESS block to be stored in the internal Block RAM.

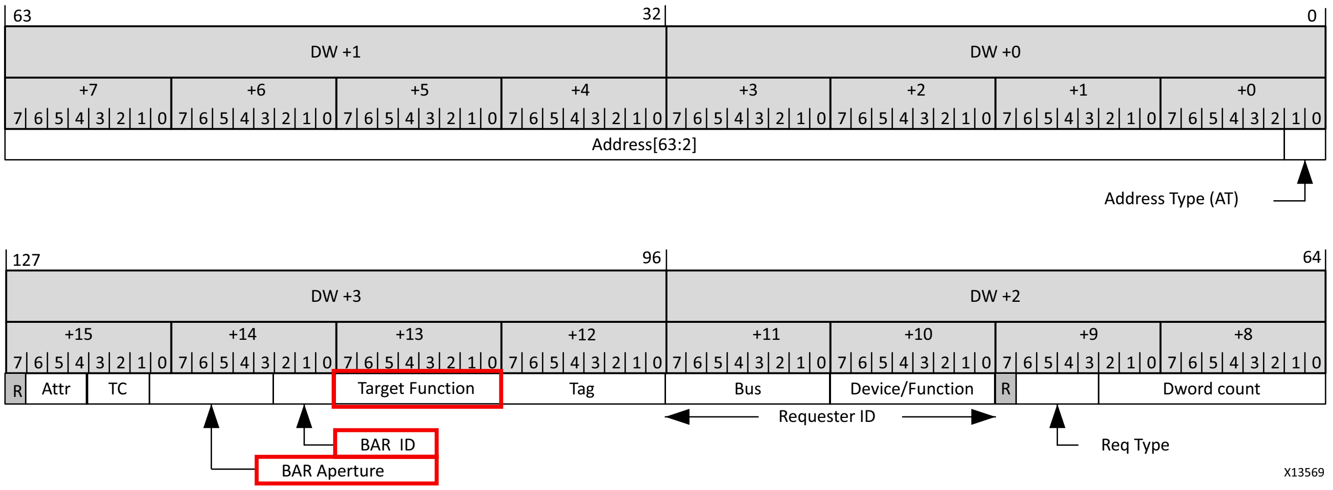


Figure 8: **Completer Request Descriptor**

In the case of a Memory Read TLP, the RX_ENGINE processes the incoming Read TLP and extracts the data and relevant address fields so that it can pass this along to the TX_ENGINE block. The TX_ENGINE block forwards the read request to the MEM_ACCESS block. On receiving the read data from MEM_ACCESS, the TX_ENGINE block sends the Completion with the Data over the Completer Completion AXI4-Stream interface.

The Completer Completion Descriptor is shown in Figure 9 and has the Device/Function field to indicate the specific function (PF/VF) generating the Completion.

Note: The Completer ID Enable bit must always be set to zero.

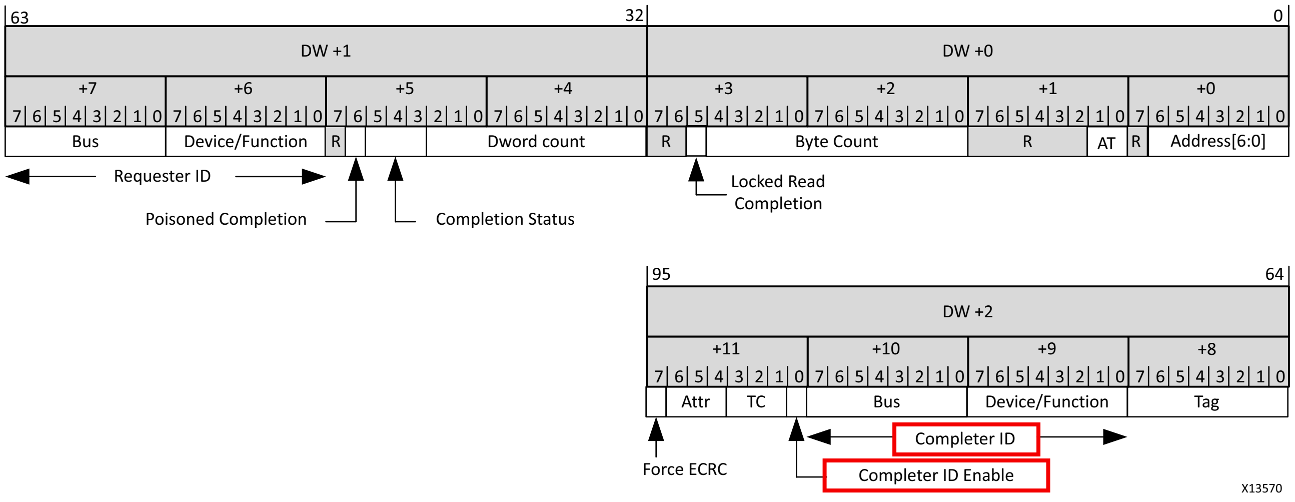


Figure 9: **Completer Completion Descriptor Format**

The location in the Block RAM is indicated using the rd_addr / wr_addr inputs to the MEM_ACCESS controller. Completer Request Descriptor bits [10:2] define the Dword aligned address within the 2 KB space. The target function is defined by the Completer Request Descriptor bits [111:104]. The target function settings are:

- **PF0:** 0h
- **PF1:** 1h
- **VFs:** from 40h to 45h.

For this reason, {`m_axis_cq_tdata[110]`, `m_axis_cq_tdata[106:104]`} are used as the upper 4 bits of the read/write address. The MEM_ACCESS controller uses these upper 4 bits to decode the reads/writes targeted at different Physical and Virtual Functions.

For example, if a Mem32 Write Request is received by the core targeting address 0x04 of BAR0 of VF1. The write is received on the Completer Requester interface with BAR ID set to `3'b0` and Target Function to 40h {`6'b010000`}. The RX_ENGINE extracts the lower address bits and the data field from the TLP and instructs the MEM_ACCESS controller to perform a write at the address {`1'b1`, `3'b000`, `8'b00000001`}. The lower 8 bits indicate the address is 0x1, Dword aligned. The upper 4 bits {`1'b1`, `3'b000`} indicate the transaction is targeted towards VF1.

While the write/read is being performed, the RX_ENGINE state machine deasserts `m_axis_cq_tready` causing the Completer Requester AXI4-Stream interface to delay receiving any further TLPs. De-asserting `m_axis_cq_tready` in this way is not required for all designs using the core. The SR-IOV reference design uses this method to simplify the control logic of the RX_ENGINE.

Reference Design Drivers and Application

The SR-IOV drivers are delivered as a part of the `xapp1177.zip` file and consist of:

- Linux_Driver_PF0
- Linux_Driver_PF1
- Linux_Driver_VF

The PF0 and PF1 drivers were built and tested with the RHEL version 6.2, and the VF driver was tested with Fedora16. To compile and install the drivers make sure the device and vendor ID in the drivers match the device under test. If needed, edit the `xpcie.c` file and change these two lines to match the device and vendor ID in use:

```
#define PCI_VENDOR_ID_XILINX 0x10ee
#define PCI_DEVICE_ID_XILINX_PCIE 0x7038
```

By default the driver Device IDs are set as:

- **PF0:** `0x7038`
- **PF1:** `0x7138`
- **VF:** `0x7238`

Also, the PF0 and PF1 drivers have the field to enable the VFs and the number of VFs to be enabled can be modified by changing this field in the `xpcie.c` file:

```
#define NUMBER_OF_VFS 0x4
```

By default the PF0 has this field set to 4 to enable all four associated VFs. PF1 has this field set to 2.

Installation of the driver requires root privileges. To install the driver and run the test, go to the directory containing the driver files, and type these commands:

1. `make`

   The expected output is as follows:

   ```
   [root@localhost Linux_Driver_PF0]# make
   make -C /lib/modules/2.6.32-220.el6.x86_64/build M=/root/Linux_Driver_PF0
   modules
   make[1]: Entering directory '/usr/src/kernels/2.6.32-220.el6.x86_64'
     Building modules, stage 2.
     MODPOST 1 modules
   make[1]: Leaving directory '/usr/src/kernels/2.6.32-220.el6.x86_64'
   g++ met.cpp -o met
   ```

2. `./make_device`

   The expected output is as follows:

   ```
   [root@localhost Linux_Driver_PF0]# ./make_device
   crw-r--r--. 1 root root 240, 1 Sep 25 18:16 /dev/xpcie
   ```

3. `insmod xpcie.ko`

   There is no noticeable output after running `insmod`.

   Note: Without root privileges, `insmod` fails to load the driver.

4. `./met`

   This runs the application. The expected output is as follows:

   ```
   Pass #[0]
   Pass #[1000]
   Pass #[2000]
   Pass #[3000]
   Pass #[4000]
   Pass #[5000]
   Pass #[6000]
   Pass #[7000]
   Pass #[8000]
   Pass #[9000]
   Pass #[10000]
   ```

The README file provided with the drivers contains descriptions for the preceding commands.

Generating and Customizing the Core

This section illustrates how to generate the Virtex-7 FPGA PCI Express Gen3 Integrated Block using SR-IOV capability.

To generate the core in Vivado Design Suite 2013.3:

1. Launch the Vivado Interactive Design Environment
1. Create a new RTL Project that targets the **xc7vx690tffg1761-2** part.
2. Open the Vivado IP catalog.
3. In the IP catalog, expand the **Standard Bus Interfaces > PCI Express** hierarchy.
4. Right-click **Virtex-7 FPGA PCI Express Gen3 Integrated Block** and select **Customize IP** (see Figure 10).

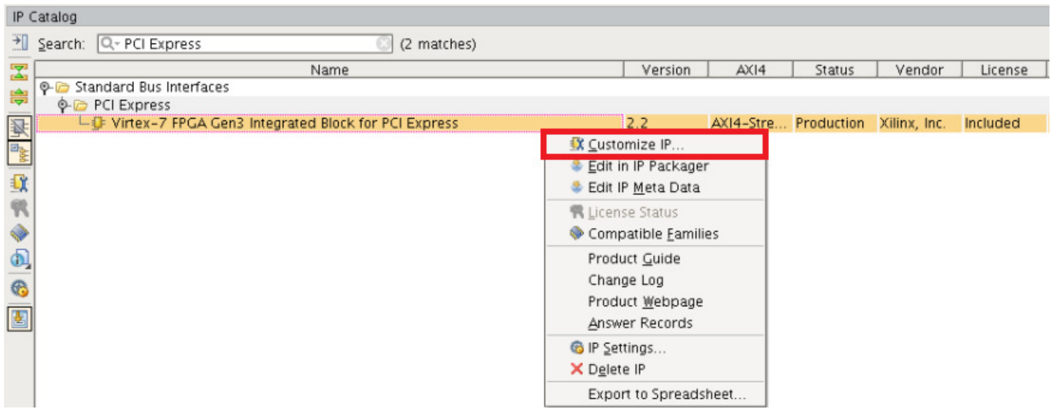


Figure 10: **Customizing the IP**

In the Customize IP dialog box, follow these steps:

5. In the **Basic** tab (Figure 11):
   a. Select the **Advanced** setting from the **Mode** list box.
   b. Set **Lane Width** to **X8**, **Maximum Link Speed** to **8.0 GT/s**, and **Xilinx Development Board** to **VC709**.

   Note: An AXI-ST interface width of 256 bits is required because the example reference design for SR-IOV only supports a 256-bit interface. The only supported configurations are x8 at 8.0 GT/s or 5.0 GT/s.

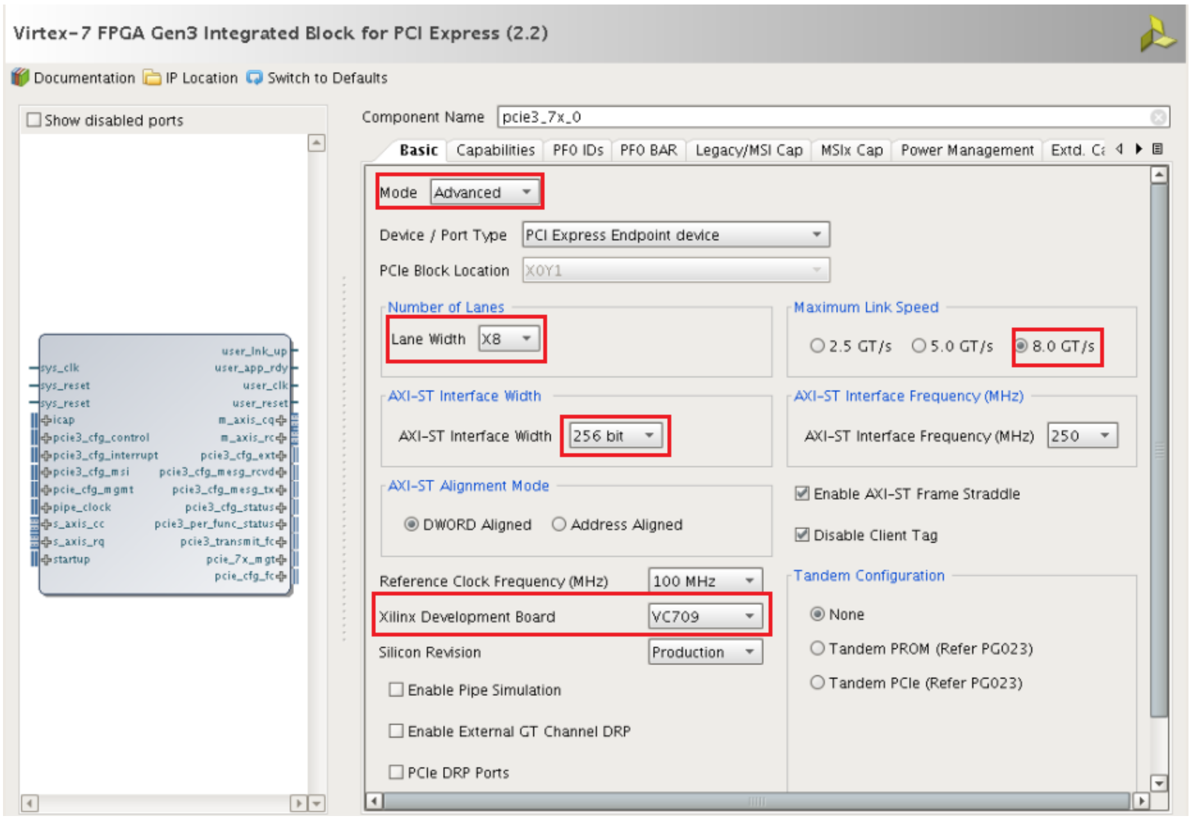
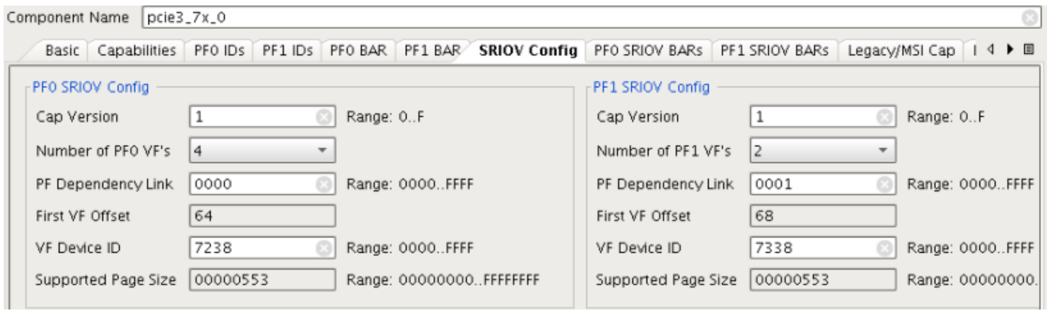


Figure 11: **Basic Tab**

6. In the **Capabilities** tab, click **Enable Physical Function 1** and **SR-IOV Capability** to enable these features.

7. In the **PF0 IDs** tab, use the default values.
8. In the **PF1 IDs** tab, enter a Device ID of **7138**, and leave the other fields as defaults.
9. In the **PF0 BAR** tab, select **Bar0**, the Bar type **Memory**, and size **4 Kilobytes**.
10. In the **PF1 BAR** tab, select **Bar0**, the Bar type **Memory**, and size **4 Kilobytes**.
11. In the **SR-IOV Config** tab, Fig., enter values as specified in Table 6.

    Note: Grayed-out fields are inaccessible and cannot be modified.



Component Name: pcie3_7x_0

PF0 SRIOV Config

Cap Version	1	Range: 0..F
Number of PF0 VF's	4	
PF Dependency Link	0000	Range: 0000..FFFF
First VF Offset	64	
VF Device ID	7238	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

PF1 SRIOV Config

Cap Version	1	Range: 0..F
Number of PF1 VF's	2	
PF Dependency Link	0001	Range: 0000..FFFF
First VF Offset	68	
VF Device ID	7338	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000.

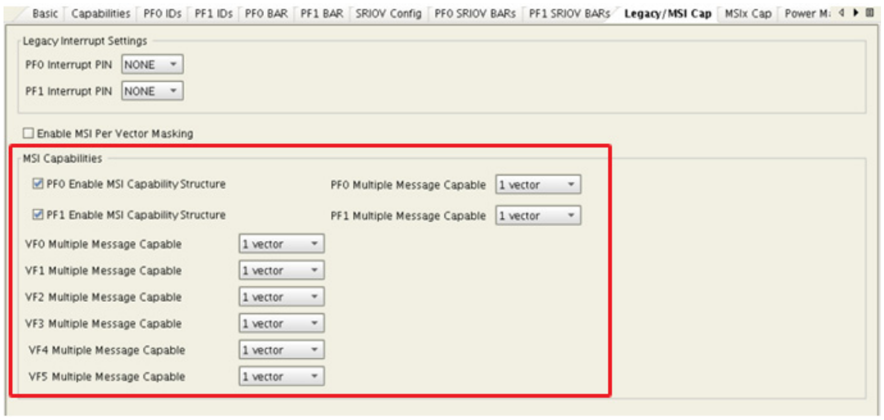
Figure 12: **SR-IOV Config Tab**

Table 6: **SR-IOV Config Settings**

Field	PF0 SRIOV Config	PF1 SRIOV Config	Description
Cap Version	1	1	Corresponds to the Capability Version of SR-IOV Extended Capability Header.
Number of PF0 VFs	4	2	Corresponds to the Initial VFs (0Ch) and the Total VFs field (0Eh) of the SR-IOV capability structure. This number must be set to assign VFs to the PF.
PF Dependency Link	0	1	Corresponds to the Function Dependency Link (12h) of the SR-IOV capability structure. There is no dependency between PF0 and PF1.
First VF Offset	64	68	Corresponds to First VF Offset field (14h). The field is fixed and cannot be modified. The First VF offset for PF1 can be calculated using the examples from Table 1 and Table 2, page 4
VF Device ID	7238	7338	Corresponds to the VF Device ID field (1Ah) of the SR-IOV capability structure.
Supported Page Size	553	553	Corresponds to Supported Page Size field (1Ch). The value set cannot be modified and it indicates that 4-KB, 8-KB, 64-KB, 256-KB, 1-MB and 4-MB page sizes are supported as required by the SR-IOV specification.

12. In the **PF0 SR-IOV BAR** tab, select **Bar0**, the Bar type **Memory**, and size **4 Kilobytes**.
13. In the **PF1 SR-IOV BAR** tab, select **Bar0**, the Bar type **Memory**, and size **4 Kilobytes**.

    Note: Select a minimum BAR size of at least 4 KB which corresponds to the minimum supported page size. Values in these two tabs are used to configure the six 32-bit BARs or three 64-bit BARs supported by VFs and corresponds to sections 24h-38h of the SR-IOV capability structure.
14. In the **Legacy/MSI Cap** tab (Figure 13):
    a. Under **Legacy Interrupt Settings**, select **NONE** for PF0 Interrupt Pin, and **NONE** for PF1 Interrupt Pin.
    b. Check **PF0 Enable MSI Capability Structure** and **PF1 Enable MSI Capability Structure** and set values as shown in Figure 13.



Basic | Capabilities | PF0 IDs | PF1 IDs | PF0 BAR | PF1 BAR | SRIOV Config | PF0 SRIOV BARs | PF1 SRIOV BARs | Legacy/MSI Cap | MSIx Cap | Power M

Legacy Interrupt Settings

PF0 Interrupt PIN: NONE

PF1 Interrupt PIN: NONE

☐ Enable MSI Per Vector Masking

MSI Capabilities

☑ PF0 Enable MSI Capability Structure — PF0 Multiple Message Capable: 1 vector

☑ PF1 Enable MSI Capability Structure — PF1 Multiple Message Capable: 1 vector

VF0 Multiple Message Capable: 1 vector

VF1 Multiple Message Capable: 1 vector

VF2 Multiple Message Capable: 1 vector

VF3 Multiple Message Capable: 1 vector

VF4 Multiple Message Capable: 1 vector

VF5 Multiple Message Capable: 1 vector

Figure 13: **Legacy/MCI Cap Tab**

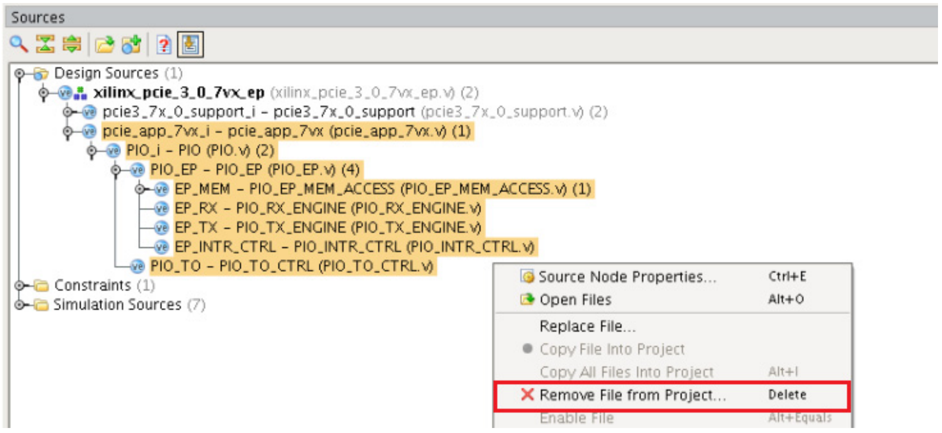
15. In the **MSIx Cap** tab, no MSIx capabilities are selected for PF0 or PF1 for this example design.
16. In the **Power Management** tab, use the defaults settings.
17. In the **Extended Capabilities 1** tab, check **Enable ARI Capability (PF0)** and **Enable ARI Capability (PF1)**.
18. In the **Extended Capabilities 2** tab, check **SR-IOV Cap Enable**.
19. Click **OK** to finish the core configuration.
20. After finishing the configuration, click **Generate** to generate the output products.
21. Right-click the generated IP file and select **Open IP Example Design**.
22. Select the destination folder, and click **OK**.

    This opens a new screen with the example design project.

Integrating the Reference Design with the Example Design Project

This section illustrates how to integrate the SR-IOV reference design files provided with the application note. The files are provided with the generated example design project in the `sriov_reference_design` folder.

1. Unzip the `xapp1177.zip` file and copy the `sriov_reference_design` directory into the local area where the example project file exists.
2. In the **Sources** tab of the example project in the Vivado IDE, select all files under the `pcie_app_7vx_i` hierarchy. Right-click the highlighted file names and select **Remove File from Project** (Figure 14).



```
Sources
Design Sources (1)
  xilinx_pcie_3_0_7vx_ep (xilinx_pcie_3_0_7vx_ep.v) (2)
    pcie3_7x_0_support_i - pcie3_7x_0_support (pcie3_7x_0_support.v) (2)
    pcie_app_7vx_i - pcie_app_7vx (pcie_app_7vx.v) (1)
      PIO_i - PIO (PIO.v) (2)
        PIO_EP - PIO_EP (PIO_EP.v) (4)
          EP_MEM - PIO_EP_MEM_ACCESS (PIO_EP_MEM_ACCESS.v) (1)
          EP_RX - PIO_RX_ENGINE (PIO_RX_ENGINE.v)
          EP_TX - PIO_TX_ENGINE (PIO_TX_ENGINE.v)
          EP_INTR_CTRL - PIO_INTR_CTRL (PIO_INTR_CTRL.v)
        PIO_TO - PIO_TO_CTRL (PIO_TO_CTRL.v)
Constraints (1)
Simulation Sources (7)
```

Source Node Properties... Ctrl+E
Open Files Alt+O
Replace File...
Copy File Into Project
Copy All Files Into Project Alt+I
Remove File from Project... Delete
Enable File Alt+Equals

Figure 14: **Remove pcie_app_7vx Instance Files**

3. Right-click `pcie_app_7vx_i`, and select **Add Sources**.
4. To add files from the `sriov_reference_design` directory, browse to the `sriov_reference_design` folder and add all provided design files.
5. After the files are added, in the Flow Navigator pane, click **Generate Bitstream** to start implementation.

Host System Setup

This section describes how to enable virtualization capability on the host server, and how to set up virtual machines using the default hypervisor of the host OS. Figure 15 shows the host system for this example.

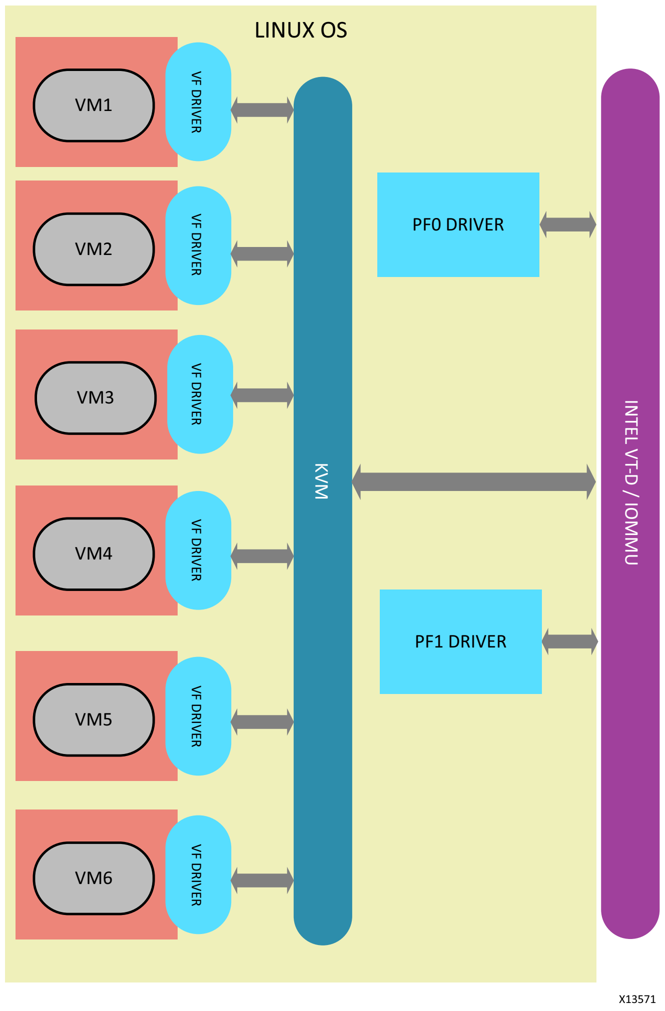
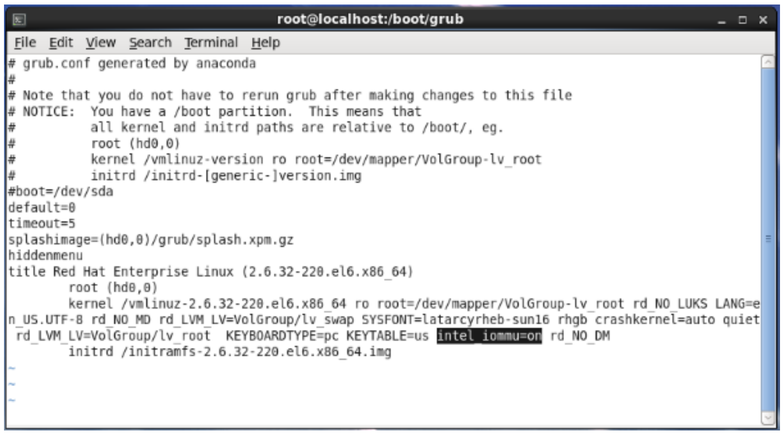


Figure 15: **Host System Setup**

1. In the BIOS options of the motherboard make sure that VT-d is enabled. The VT-d option is usually found in **BIOS > Advance Settings > System Agent > Enable VT-d**.

   As discussed in System Requirements for SR-IOV, page 6, Red Had Enterprise Linux version 6.2 is used as the Host Operating System. To use virtualization on Red Hat Enterprise Linux, the `qemu-kvm` and `qemu-img` packages are needed. These packages provide the KVM emulator and disk image manager functions.
2. Select the **Software Development Workstation** option while installing RHEL. This ensures that all required virtualization packages (`qemu-kvm` and `qemu-img`) are pre-installed.

3. After the OS is installed and ready, add `intel_iommu = on` to the `grub.conf` file located in the `/boot/grub/` directory as shown in Figure 16.


root@localhost:/boot/grub
File Edit View Search Terminal Help

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE:  You have a /boot partition.  This means that
#          all kernel and initrd paths are relative to /boot/, eg.
#          root (hd0,0)
#          kernel /vmlinuz-version ro root=/dev/mapper/VolGroup-lv_root
#          initrd /initrd-[generic-]version.img
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
title Red Hat Enterprise Linux (2.6.32-220.el6.x86_64)
        root (hd0,0)
        kernel /vmlinuz-2.6.32-220.el6.x86_64 ro root=/dev/mapper/VolGroup-lv_root rd_NO_LUKS LANG=e
n_US.UTF-8 rd_NO_MD rd_LVM_LV=VolGroup/lv_swap SYSFONT=latarcyrheb-sun16 rhgb crashkernel=auto quiet
 rd_LVM_LV=VolGroup/lv_root  KEYBOARDTYPE=pc KEYTABLE=us intel_iommu=on rd_NO_DM
        initrd /initramfs-2.6.32-220.el6.x86_64.img
~
~
~
```

Figure 16: **Edit grub.conf**

4. Reboot the system for the changes to take effect.
5. Open the Virtual Machine Manager located under **Applications > System tools**.
6. Click the **Create a new virtual machine** button as shown in Figure 17.

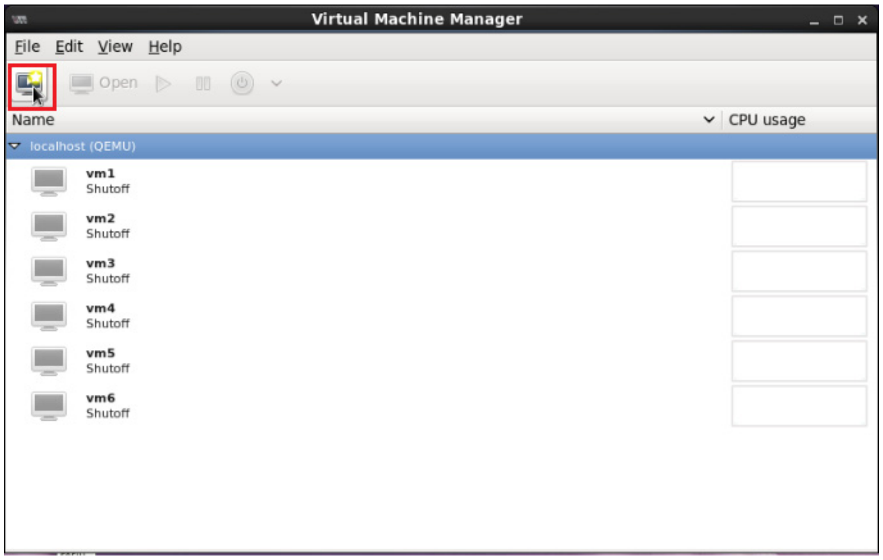


Figure 17: **Create New Virtual Machine**

7. Follow the instructions in the subsequent tabs to create a Virtual Machine with Fedora16 as the Guest OS. Select the **Software Development Workstation** option while installing Fedora 16 so that all libraries required to compile the drivers are pre-installed.
8. After the first Virtual Machine is created, right-click **Virtual Machine**, and select the **clone** option to create 5 other virtual machines. Only one clone can be created at a time.

After all Virtual machines are created, the Virtual Machine Manager should resemble Figure 18.

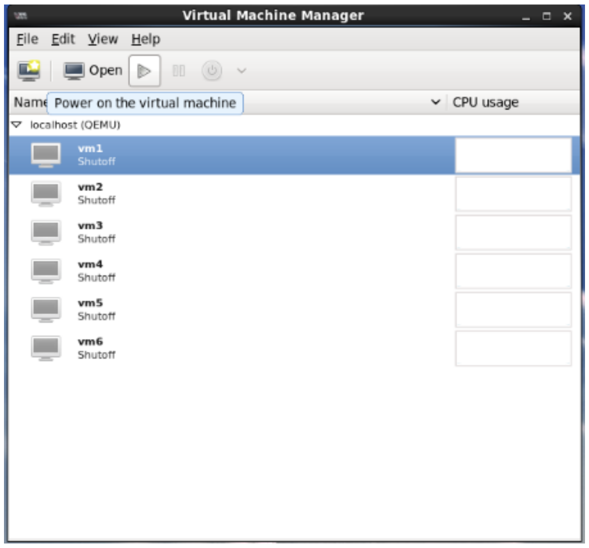


Figure 18: **All Virtual Machines**

For additional details and troubleshooting information, see the *Red Hat Enterprise Linux Virtualization Host Configuration and Guest Installation Guide* [Ref 6].

Reference Design Validation in Hardware

This section describes the hardware validation of the SR-IOV reference design and provides details on VF enumeration, attaching VFs to Virtual Machines and running PIO traffic on the attached VFs.

1. With the host system powered off, plug the VC709 board into a Gen3 PCIe x8 slot on the host system.
2. Use the external AC power adapter supplied and power-on the board.

   Caution! Do not use the PCIe power connector from the ATX power supply of the host machine. See the *VC709 Evaluation Board for the Virtex-7 FPGA User Guide* [Ref 7] for proper procedures.
3. Connect a USB Type-A to Micro-B cable from your local machine to the USB JTAG (Digilent) connector on the VC709 board, as shown in Figure 19.

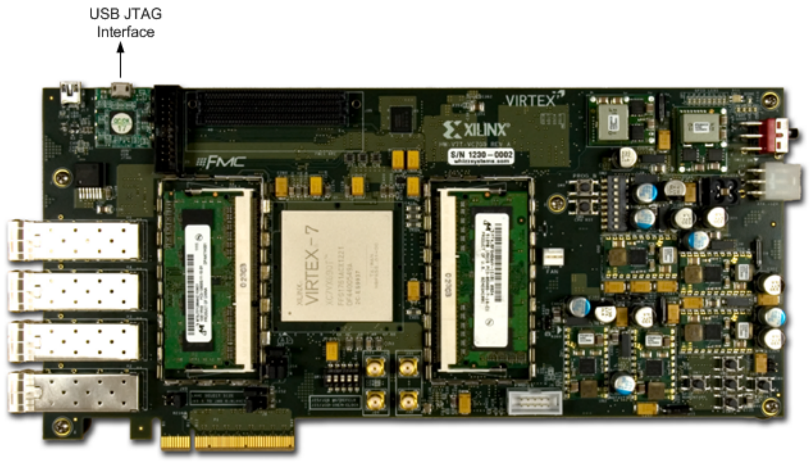
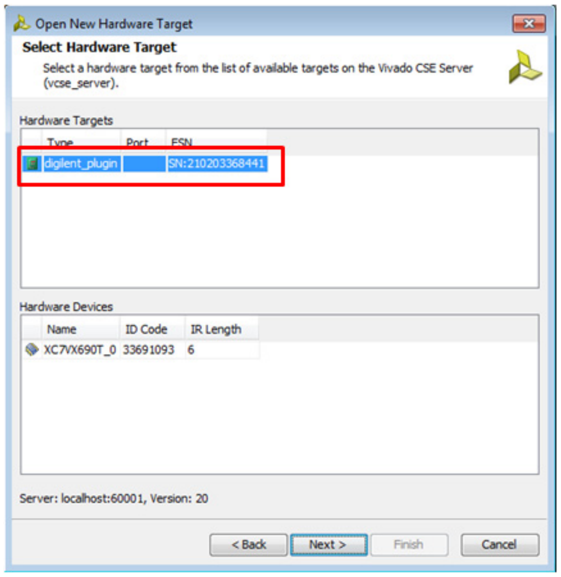


Figure 19: **VC709 Board USB JTAG Connector**

4. In the Vivado IDE Flow Navigator pane, select **Open Hardware Session** under **Program and Debug** to load the bitstream.
5. Select **Open New Hardware Target** and use the default Vivado CSE Server.

6. Select the appropriate target as shown in Figure 20.



Select Hardware Target

Select a hardware target from the list of available targets on the Vivado CSE Server (vcse_server).

Hardware Targets

Type	Port	ESN
digilent_plugin		SN:210203368441

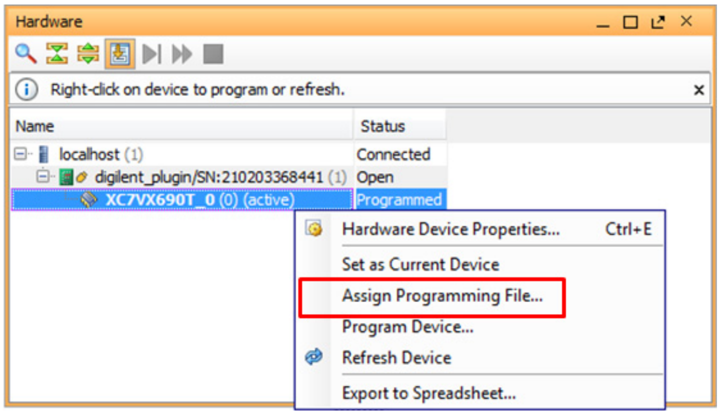
Hardware Devices

Name	ID Code	IR Length
XC7VX690T_0	33691093	6

Server: localhost:60001, Version: 20

Figure 20: **Select Hardware Target**

7. Select the default clock frequency, and click **Finish** to view the Hardware Device.
8. In the Hardware window, right-click the XC7VX690_T device, and select **Assign Programming File** as shown in Figure 21.



Right-click on device to program or refresh.

Name	Status
localhost (1)	Connected
digilent_plugin/SN:210203368441 (1)	Open
XC7VX690T_0 (0) (active)	Programmed

Figure 21: **Assign Programming File**

The programming file is associated with the hardware device.

9. Right-click the device and select **Program Device** to program the hardware device.
10. Power on the host machine and login as the root user.
11. Open a terminal, and type these commands:

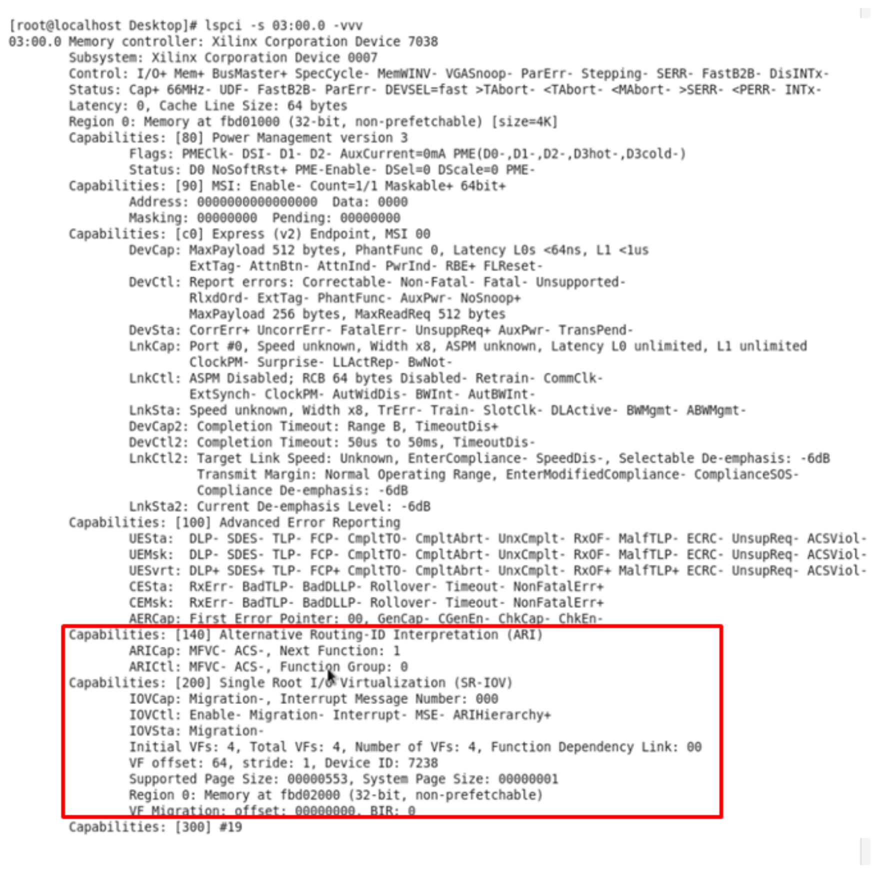
- `lspci | grep -i xilinx`

  The expected output is:

```
03:00.0 Memory controller: Xilinx Corporation Device 7038
03:00.1 Memory controller: Xilinx Corporation Device 7138
```

- `lspci -s 03:00.0 -vvv`

  This command lists the configuration space of PF0 as shown in Figure 22. PF0 ARI capabilities are enabled and are located at offset 0x140. SR-IOV capabilities are located at address 0x200. PF0 has four associated VFs, which have not yet been enabled. Each VF has a 4K BAR0 enabled.

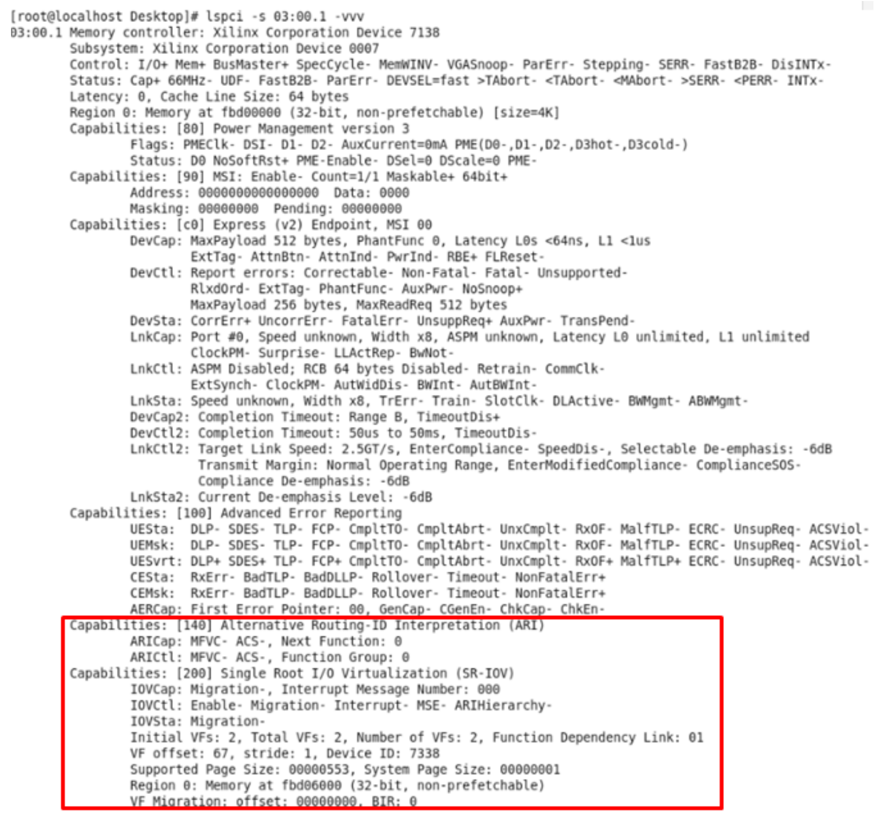


```
[root@localhost Desktop]# lspci -s 03:00.0 -vvv
03:00.0 Memory controller: Xilinx Corporation Device 7038
        Subsystem: Xilinx Corporation Device 0007
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx-
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Region 0: Memory at fbd01000 (32-bit, non-prefetchable) [size=4K]
        Capabilities: [80] Power Management version 3
                Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot-,D3cold-)
                Status: D0 NoSoftRst+ PME-Enable- DSel=0 DScale=0 PME-
        Capabilities: [90] MSI: Enable- Count=1/1 Maskable+ 64bit+
                Address: 0000000000000000  Data: 0000
                Masking: 00000000  Pending: 00000000
        Capabilities: [c0] Express (v2) Endpoint, MSI 00
                DevCap: MaxPayload 512 bytes, PhantFunc 0, Latency L0s <64ns, L1 <1us
                        ExtTag- AttnBtn- AttnInd- PwrInd- RBE+ FLReset-
                DevCtl: Report errors: Correctable- Non-Fatal- Fatal- Unsupported-
                        RlxdOrd- ExtTag- PhantFunc- AuxPwr- NoSnoop+
                        MaxPayload 256 bytes, MaxReadReq 512 bytes
                DevSta: CorrErr+ UncorrErr- FatalErr- UnsuppReq+ AuxPwr- TransPend-
                LnkCap: Port #0, Speed unknown, Width x8, ASPM unknown, Latency L0 unlimited, L1 unlimited
                        ClockPM- Surprise- LLActRep- BwNot-
                LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- Retrain- CommClk-
                        ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
                LnkSta: Speed unknown, Width x8, TrErr- Train- SlotClk- DLActive- BWMgmt- ABWMgmt-
                DevCap2: Completion Timeout: Range B, TimeoutDis+
                DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis-
                LnkCtl2: Target Link Speed: Unknown, EnterCompliance- SpeedDis-, Selectable De-emphasis: -6dB
                         Transmit Margin: Normal Operating Range, EnterModifiedCompliance- ComplianceSOS-
                         Compliance De-emphasis: -6dB
                LnkSta2: Current De-emphasis Level: -6dB
        Capabilities: [100] Advanced Error Reporting
                UESta:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq- ACSViol-
                UEMsk:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq- ACSViol-
                UESvrt: DLP+ SDES+ TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC- UnsupReq- ACSViol-
                CESta:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
                CEMsk:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
                AERCap: First Error Pointer: 00, GenCap- CGenEn- ChkCap- ChkEn-
        Capabilities: [140] Alternative Routing-ID Interpretation (ARI)
                ARICap: MFVC- ACS-, Next Function: 1
                ARICtl: MFVC- ACS-, Function Group: 0
        Capabilities: [200] Single Root I/O Virtualization (SR-IOV)
                IOVCap: Migration-, Interrupt Message Number: 000
                IOVCtl: Enable- Migration- Interrupt- MSE- ARIHierarchy+
                IOVSta: Migration-
                Initial VFs: 4, Total VFs: 4, Number of VFs: 4, Function Dependency Link: 00
                VF offset: 64, stride: 1, Device ID: 7238
                Supported Page Size: 00000553, System Page Size: 00000001
                Region 0: Memory at fbd02000 (32-bit, non-prefetchable)
                VF Migration: offset: 00000000, BIR: 0
        Capabilities: [300] #19
```

Figure 22: **PF0 Capabilities Enabled**

- `lspci -s 03:01.0 -vvv`

  Similar to the previous command, this command lists the capabilities of PF1 as shown in Figure 23.



```
[root@localhost Desktop]# lspci -s 03:00.1 -vvv
03:00.1 Memory controller: Xilinx Corporation Device 7138
        Subsystem: Xilinx Corporation Device 0007
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx-
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Region 0: Memory at fbd00000 (32-bit, non-prefetchable) [size=4K]
        Capabilities: [80] Power Management version 3
                Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot-,D3cold-)
                Status: D0 NoSoftRst+ PME-Enable- DSel=0 DScale=0 PME-
        Capabilities: [90] MSI: Enable- Count=1/1 Maskable+ 64bit+
                Address: 0000000000000000  Data: 0000
                Masking: 00000000  Pending: 00000000
        Capabilities: [c0] Express (v2) Endpoint, MSI 00
                DevCap: MaxPayload 512 bytes, PhantFunc 0, Latency L0s <64ns, L1 <1us
                        ExtTag- AttnBtn- AttnInd- PwrInd- RBE+ FLReset-
                DevCtl: Report errors: Correctable- Non-Fatal- Fatal- Unsupported-
                        RlxdOrd- ExtTag- PhantFunc- AuxPwr- NoSnoop+
                        MaxPayload 256 bytes, MaxReadReq 512 bytes
                DevSta: CorrErr+ UncorrErr- FatalErr- UnsuppReq+ AuxPwr- TransPend-
                LnkCap: Port #0, Speed unknown, Width x8, ASPM unknown, Latency L0 unlimited, L1 unlimited
                        ClockPM- Surprise- LLActRep- BwNot-
                LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- Retrain- CommClk-
                        ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
                LnkSta: Speed unknown, Width x8, TrErr- Train- SlotClk- DLActive- BWMgmt- ABWMgmt-
                DevCap2: Completion Timeout: Range B, TimeoutDis+
                DevCtl2: Completion Timeout: 50us to 50ms, TimeoutDis-
                LnkCtl2: Target Link Speed: 2.5GT/s, EnterCompliance- SpeedDis-, Selectable De-emphasis: -6dB
                         Transmit Margin: Normal Operating Range, EnterModifiedCompliance- ComplianceSOS-
                         Compliance De-emphasis: -6dB
                LnkSta2: Current De-emphasis Level: -6dB
        Capabilities: [100] Advanced Error Reporting
                UESta:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq- ACSViol-
                UEMsk:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq- ACSViol-
                UESvrt: DLP+ SDES+ TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC- UnsupReq- ACSViol-
                CESta:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
                CEMsk:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr+
                AERCap: First Error Pointer: 00, GenCap- CGenEn- ChkCap- ChkEn-
        Capabilities: [140] Alternative Routing-ID Interpretation (ARI)
                ARICap: MFVC- ACS-, Next Function: 0
                ARICtl: MFVC- ACS-, Function Group: 0
        Capabilities: [200] Single Root I/O Virtualization (SR-IOV)
                IOVCap: Migration-, Interrupt Message Number: 000
                IOVCtl: Enable- Migration- Interrupt- MSE- ARIHierarchy-
                IOVSta: Migration-
                Initial VFs: 2, Total VFs: 2, Number of VFs: 2, Function Dependency Link: 01
                VF offset: 67, stride: 1, Device ID: 7338
                Supported Page Size: 00000553, System Page Size: 00000001
                Region 0: Memory at fbd06000 (32-bit, non-prefetchable)
                VF Migration: offset: 00000000, BIR: 0
```

Figure 23: **PF1 Capabilities Enabled**

12. Copy the `Linux_Driver_PF0` directory to the host machine.
13. Modify the `xpcie.c` file so that the values in the fields shown reflect the `lspci` output of PF0:

    ```
    #define PCI_VENDOR_ID_XILINX 0x10ee
    #define PCI_DEVICE_ID_XILINX_PCIE 0x7038
    #define NUMBER_OF_VFS 0x4
    ```
14. Run these commands from Reference Design Drivers and Application, page 11, to install the drivers for PF0:
    a. `make`
    b. `./make_device`
    c. `insmod xpcie.ko`
15. Perform step 12, step 13, and step 14 for the PF1 driver.
16. In the `xpcie_pf1.c` file, verify that the fields shown have the correct value:

    ```
    #define PCI_VENDOR_ID_XILINX 0x10ee
    #define PCI_DEVICE_ID_XILINX_PCIE 0x7138
    #define NUMBER_OF_VFS 0x2
    ```

17. After both drivers are installed, type the command:

    ```
    lspci | grep -i xilinx
    ```

    All VFs should be enumerated as shown:

    ```
    03:00.0 Memory controller: Xilinx Corporation Device 7038
    03:00.1 Memory controller: Xilinx Corporation Device 7138
    03:08.0 Memory controller: Xilinx Corporation Device 7238
    03:08.1 Memory controller: Xilinx Corporation Device 7238
    03:08.2 Memory controller: Xilinx Corporation Device 7238
    03:08.3 Memory controller: Xilinx Corporation Device 7238
    03:08.4 Memory controller: Xilinx Corporation Device 7338
    03:08.5 Memory controller: Xilinx Corporation Device 7338
    ```

18. Use the `virsh` commands which are a part of the RHEL virtualization package to probe the virtual functions. To verify that devices exist with `virsh` and to ensure that virtualization is present and functional on the Host OS, type the command:

    ```
    virsh nodedev-list | grep 03
    ```

    `03` represents the bus number on which the Xilinx PCI Express Endpoint is present.

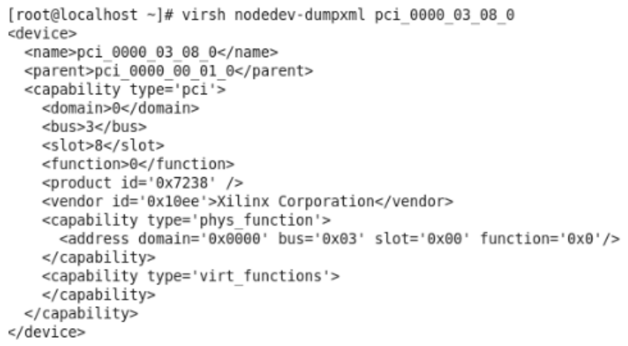
    The expected output is:

    ```
    pci_0000_00_03_0
    pci_0000_03_00_0
    pci_0000_03_00_1
    pci_0000_03_08_0
    pci_0000_03_08_1
    pci_0000_03_08_2
    pci_0000_03_08_3
    pci_0000_03_08_4
    pci_0000_03_08_5
    ```

19. To view the Virtual function details, type the command:

    ```
    virsh nodedev-dumpxml pci_0000_03_08_0
    ```

    In Figure 24, observe that this virtual function is on bus0x03, slot0x08, and function 0x0. The physical function associated with this virtual function is on bus 0x03, slot 0x00 and function 0x0 (PF0).



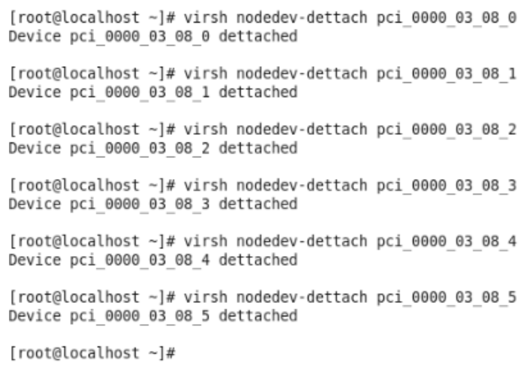
```
[root@localhost ~]# virsh nodedev-dumpxml pci_0000_03_08_0
<device>
  <name>pci_0000_03_08_0</name>
  <parent>pci_0000_00_01_0</parent>
  <capability type='pci'>
    <domain>0</domain>
    <bus>3</bus>
    <slot>8</slot>
    <function>0</function>
    <product id='0x7238' />
    <vendor id='0x10ee'>Xilinx Corporation</vendor>
    <capability type='phys_function'>
      <address domain='0x0000' bus='0x03' slot='0x00' function='0x0'/>
    </capability>
    <capability type='virt_functions'>
    </capability>
  </capability>
</device>
```

Figure 24: **Virtual Function VF0 Details**

20. To detach the devices, type the command:

    ```
    virsh nodedev-dettach pci_0000_03_08_0
    ```

21. Perform step 20 for all VFs as shown in Figure 25.



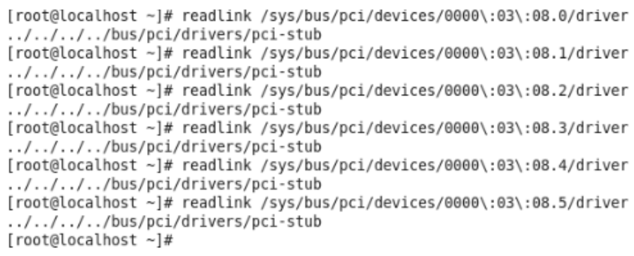
```
[root@localhost ~]# virsh nodedev-dettach pci_0000_03_08_0
Device pci_0000_03_08_0 dettached

[root@localhost ~]# virsh nodedev-dettach pci_0000_03_08_1
Device pci_0000_03_08_1 dettached

[root@localhost ~]# virsh nodedev-dettach pci_0000_03_08_2
Device pci_0000_03_08_2 dettached

[root@localhost ~]# virsh nodedev-dettach pci_0000_03_08_3
Device pci_0000_03_08_3 dettached

[root@localhost ~]# virsh nodedev-dettach pci_0000_03_08_4
Device pci_0000_03_08_4 dettached

[root@localhost ~]# virsh nodedev-dettach pci_0000_03_08_5
Device pci_0000_03_08_5 dettached

[root@localhost ~]#
```

Figure 25: **Detach VFs from Host OS**

22. To ensure that the PF0 device has been detached from the host system, type the command:

    `readlink /sys/bus/pci/devices/0000\:03\:08.0/driver`

    The VFs are detached and are consumed by pci-stub as shown in Figure 26.



```
[root@localhost ~]# readlink /sys/bus/pci/devices/0000\:03\:08.0/driver
../../../../bus/pci/drivers/pci-stub
[root@localhost ~]# readlink /sys/bus/pci/devices/0000\:03\:08.1/driver
../../../../bus/pci/drivers/pci-stub
[root@localhost ~]# readlink /sys/bus/pci/devices/0000\:03\:08.2/driver
../../../../bus/pci/drivers/pci-stub
[root@localhost ~]# readlink /sys/bus/pci/devices/0000\:03\:08.3/driver
../../../../bus/pci/drivers/pci-stub
[root@localhost ~]# readlink /sys/bus/pci/devices/0000\:03\:08.4/driver
../../../../bus/pci/drivers/pci-stub
[root@localhost ~]# readlink /sys/bus/pci/devices/0000\:03\:08.5/driver
../../../../bus/pci/drivers/pci-stub
[root@localhost ~]#
```

Figure 26: **VFs Detached**

23. To enable management of the PCI Express Device (Virtual Function) from the Virtual machine, set the SELinux Boolean. This enables the devices to be managed by the virtual machines. Type this command:

    `setsebool -P virt_use_sysfs 1`

24. Now the VFs are ready to be attached to the Guest OS. Open the Virtual Machine and click the **Add Hardware** button to select the physical device to add to the virtual machine.
25. Select **PCI Host Device** and add the appropriate Host Device (Virtual Function) as shown in Figure 27.

    Note: When a VF is attached to a virtual machine it cannot be attached to another virtual machine unless it is detached first.

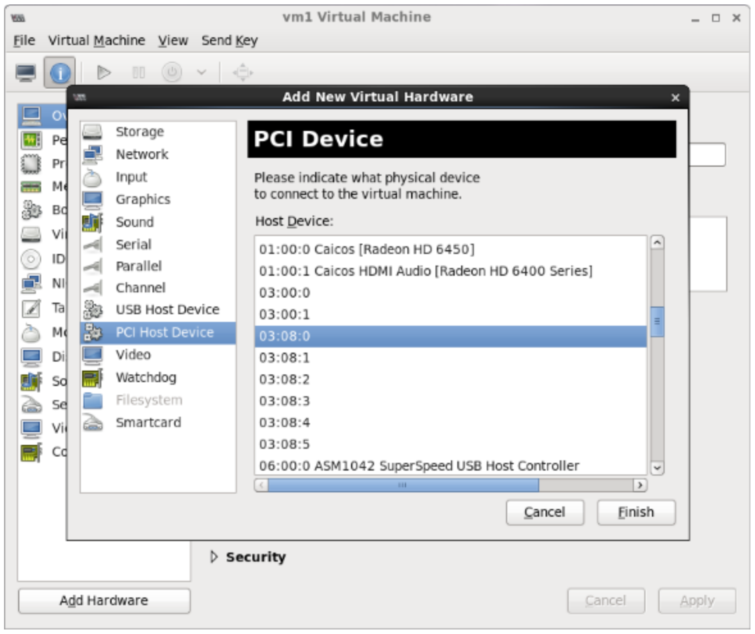
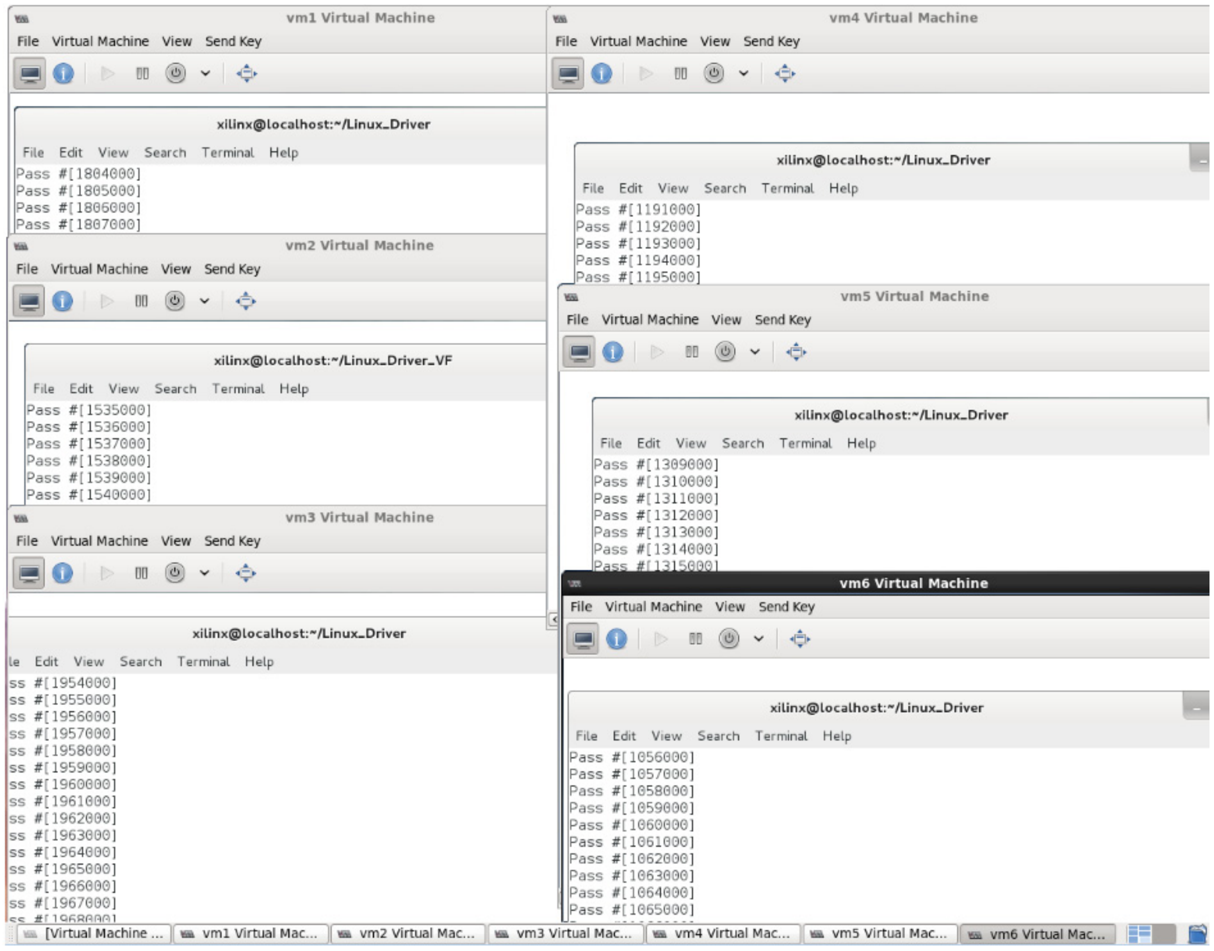


Figure 27: **Select Virtual Function to Attach to Virtual Machine**

26. After the setup is complete, power on the guest virtual machine.
27. Copy the `Linux_Driver_VF` directory onto the guest virtual machine and perform the steps shown from Reference Design Drivers and Application, page 11 to install the drivers and run PIO traffic on the Virtual Function:
    a. `make`
    b. `./make_device`
    c. `insmod xpcie.ko`
28. Perform step 24 through step 27 for all Virtual Machines to attach all six VFs to the virtual machines.

Figure 28 shows PIO traffic running simultaneously on all 6 virtual machines.



```
Pass #[1804000]
Pass #[1805000]
Pass #[1806000]
Pass #[1807000]
```

```
Pass #[1535000]
Pass #[1536000]
Pass #[1537000]
Pass #[1538000]
Pass #[1539000]
Pass #[1540000]
```

```
ss #[1954000]
ss #[1955000]
ss #[1956000]
ss #[1957000]
ss #[1958000]
ss #[1959000]
ss #[1960000]
ss #[1961000]
ss #[1962000]
ss #[1963000]
ss #[1964000]
ss #[1965000]
ss #[1966000]
ss #[1967000]
ss #[1968000]
```

```
Pass #[1191000]
Pass #[1192000]
Pass #[1193000]
Pass #[1194000]
Pass #[1195000]
```

```
Pass #[1309000]
Pass #[1310000]
Pass #[1311000]
Pass #[1312000]
Pass #[1313000]
Pass #[1314000]
Pass #[1315000]
```

```
Pass #[1056000]
Pass #[1057000]
Pass #[1058000]
Pass #[1059000]
Pass #[1060000]
Pass #[1061000]
Pass #[1062000]
Pass #[1063000]
Pass #[1064000]
Pass #[1065000]
```

Figure 28: **Simultaneous PIO Traffic on All Virtual Machines**

References

These documents provide supplemental material useful with this application note:

1. *Xilinx Virtex-7 FPGA Gen3 Integrated Block for PCI Express v2.2* (PG023)
2. *Xilinx Virtex-7 FPGA Integrated Block for PCI Express – Release Notes* (AR 54645)
3. *PCI Express Base Specification Rev 3.0* on PCI-SIG website
4. *Single Root IO Virtualization and Sharing Specification* v1.1 on PCI-SIG website
5. *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology*; Paper by Intel
6. *Red Hat Enterprise Linux Virtualization Host Configuration and Guest Installation Guide*
7. *VC709 Evaluation Board for the Virtex-7 FPGA User Guide* (UG887)
8. *Intel Virtualization Technology for Directed I/O Architecture Specification*

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
11/15/2013	1.0	Initial Xilinx release.

Notice of Disclaimer