Reversing SR-IOV For Fun and Profit

Adir Abraham

5-8 September, 2018

r2con

Legal Notices and disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel and the Intel logo, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

$ whoami

- Name: Adir Abraham
- Title: SW/FW Security Researcher & Reverse Engineer @ Intel
- Graduated: BSc Computer Science Edu.; BA Economics – Technion IIT
- Certified: CISSP, CySA+, CCSK

- Twitter: @adirab
- LinkedIn: https://linkedin.com/in/adirab

Agenda

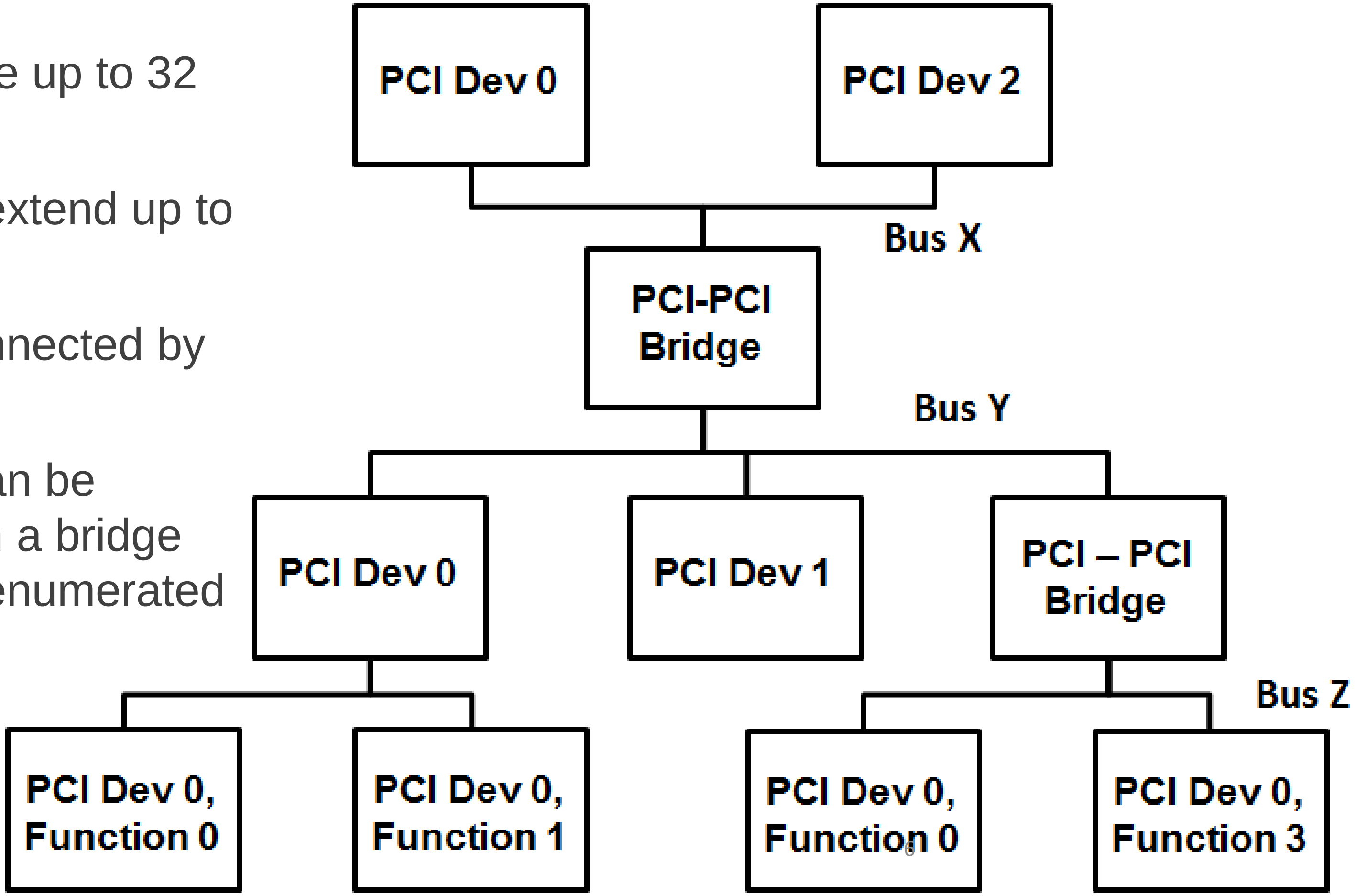
- Overview
  - Introduction
    - PCIe Devices
    - I/O Virtualization
- PCIe Virtualization
  - Motivation
  - Directed I/O
  - PCIe Architecture
- SR-IOV
  - Architecture Supporting SR-IOV Capability
  - ARI – Alternative Routing ID Interpretation
  - ATS - Address Translation Service
- Demo
  - Finding PCIe features using radare2
  - Finding SR-IOV capabilities using radare2

PCI - Peripheral Component Interconnect

- It’s a bus protocol developed by Intel around 1993
- Purpose is to attach/interconnect local hardware devices to a computer
- PCI is integrated into the chipset, forming a “backbone”
  - Holds true for both Intel and AMD-based systems
  - Logically speaking, the Chipset is a PCI System
- 32-bit bus with multiplexed address and data lines
  - Supports 64-bit by performing two 32-bit reads
- PCI component interface is processor independent
  - The CPU/BIOS reads and writes to the configuration space to configure much of the system
- Intel’s MCH/ICH chipsets implement PCI Local Bus Protocol 2.3
- PCH chipsets implement PCI Express protocol (v. 2.0)
  - Still supports PCI 2.3
- PCI standards are currently maintained and defined by the PCI /SIG:
  - http://www.pcisig.com/specifications/

Generic PCI Topology:
Buses, Devices, and Functions

- Up to 256 Buses
- Each Bus can have up to 32 devices attached
- Each Device can extend up to 8 Functions
- Buses are interconnected by Bridges
- Multiple bridges can be connected through a bridge interface and are enumerated



PCI Dev 0

PCI Dev 2

Bus X

PCI-PCI Bridge

Bus Y

PCI Dev 0

PCI Dev 1

PCI – PCI Bridge

Bus Z

PCI Dev 0, Function 0

PCI Dev 0, Function 1

PCI Dev 0, Function 0

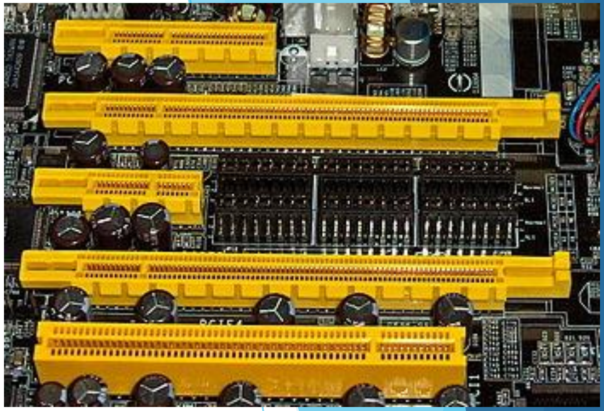
PCI Dev 0, Function 3

PCI Address Spaces

- PCI implements three address spaces:

- 1) PCI Configuration Space (up to 256 Bytes)
  - Required/standard. Defined in the specifications. Every PCI device has a configuration space.
- 2) PCI Memory-mapped space
  - Optional. Dependent on whether the device manufacturer needs to map system memory to the PCI device
- 3) PCI I/O-mapped space
  - Optional. Same as PCI Memory Space

PCI Express (PCIe)

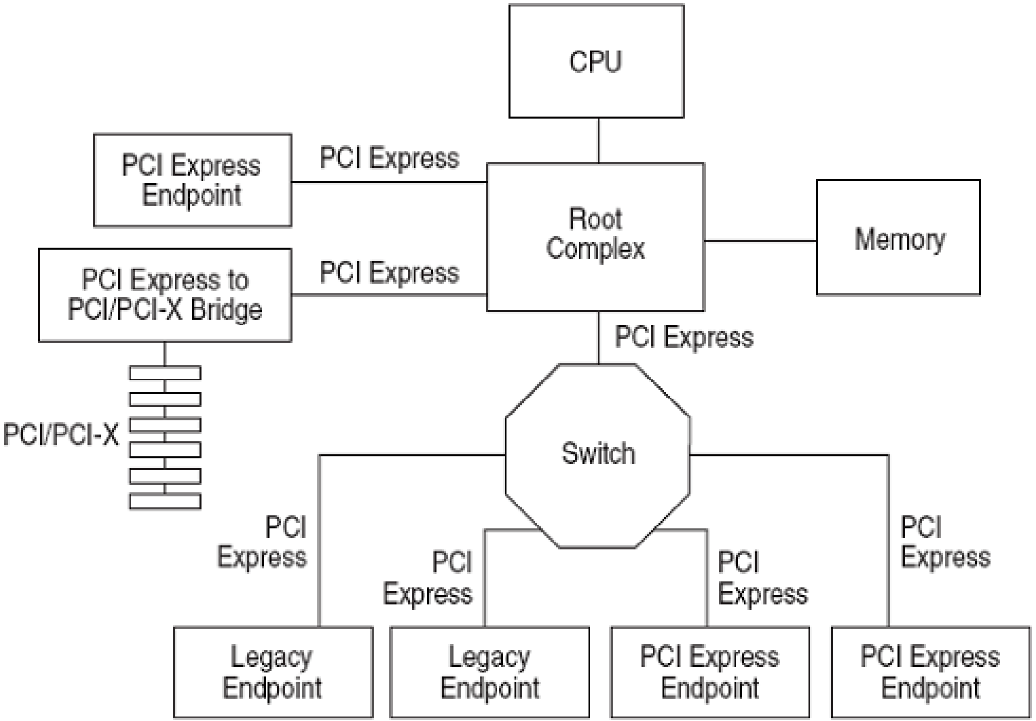


- Peripheral Component Interconnect Express
- Developed around 2004
- Packet-based transaction protocol
- Very different from PCI at the hardware level
- For software configuration purposes, it is mostly the same
  - Adds an extended configuration space of 4KB
- Provides backwards compatibility for Compatible PCI
- Adds 4KB of PCI Express Extended Capabilities registers
  - Located in either the Configuration space starting at offset 256 (immediately following the Compatible PCI configuration space)
  - Or located at a MMIO location specified in the Root Complex Register Block (RCRB)

PCI Express (PCIe) Address Spaces

- PCIe implements four address spaces:
- 1) PCIe Configuration Space (up to 4KBytes)
  - Required/standard. Defined in the specifications. Every PCIe device has its configuration space mapped to memory.
  - Also provides the first 256 bytes of compatible PCI (memory-mapped and via port IO for backwards compatibility)
- 2) PCIe Memory-mapped space
  - Optional. Dependent on whether the device manufacturer needs to map system memory to the PCI device
- 3) PCIe I/O-mapped space
  - Optional. Same as PCI Memory Space
- 4) PCIe Message Space
  - For low-level protocol messaging/interrupts. We don’t get into this in this class

Generic PCIe Topology:



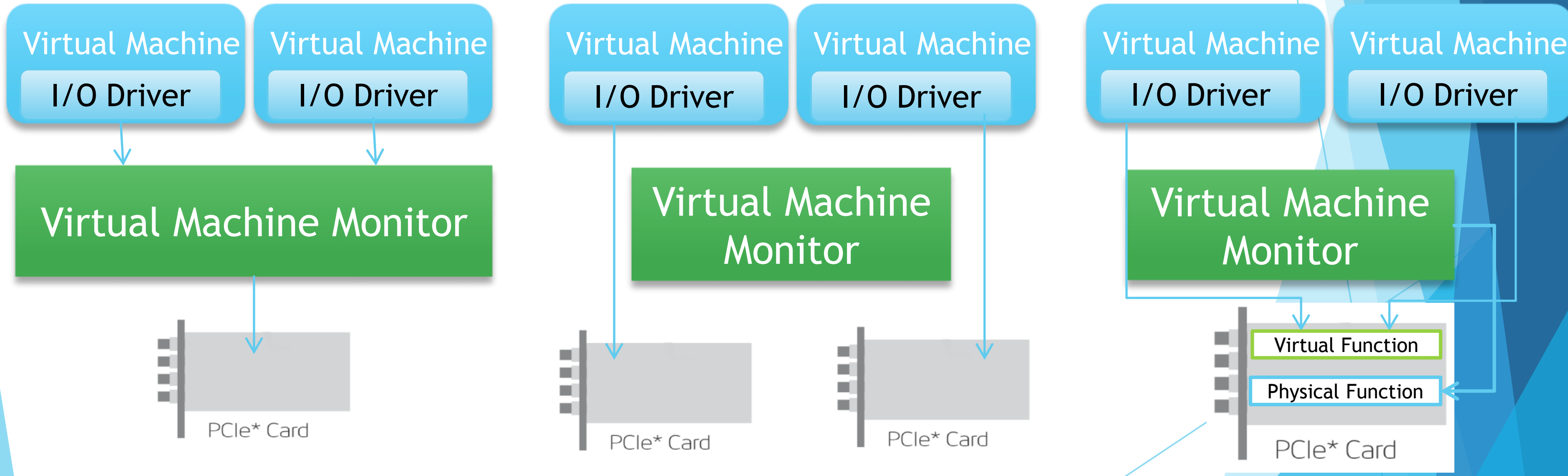
- The Root Complex connects the processor to the system memory and components
- Same number of devices supported as PCI
- Up to 256 PCIe buses
- Up to 32 PCIe devices
- Up to 8 Functions
- Each Function can implement up to 4 KB of configuration space

I/O Virtualization Concept

- Virtualizing the I/O path between a host and a PCIe device
- **Prevent** direct access to physical (memory) resources
- **Managing** and **separating** each device’s I/O space using a VMM
- Can apply to anything that uses an adapter in a computer, such as:
  - Ethernet Network Interface Cards (NICs)
  - Disk Controllers (including RAID controllers)
  - Fibre Channel Host Bus Adapters (HBAs)
  - Graphics/Video cards or co-processors
  - SSDs mounted on internal cards

I/O Virtualization Implementations

- A - Software only (**emulation**)
- B - Directed I/O (**enhance performance**)
- C – Directed I/O and Device Sharing (**resource saving**)



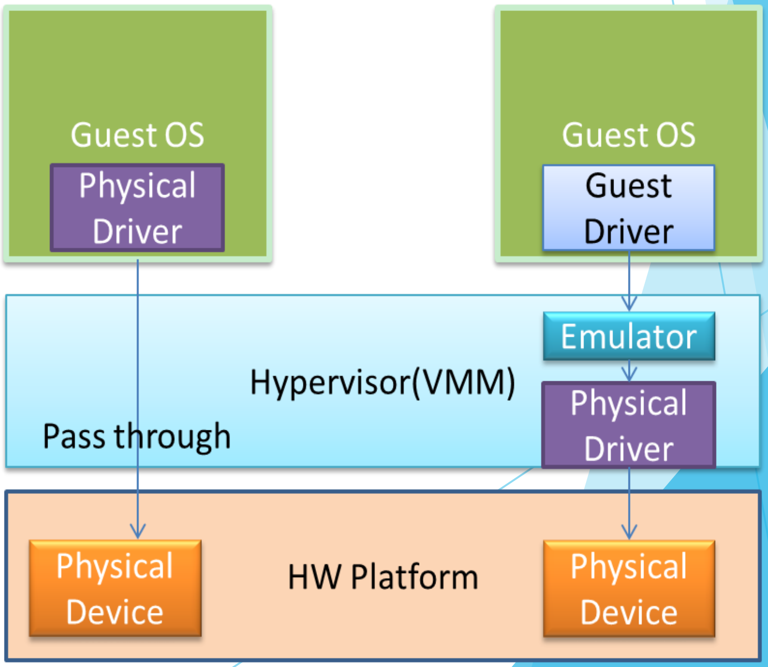
A - Software only

B - Directed I/O

C - Directed I/O & Device Sharing

Directed I/O - Passthrough

- Software-based sharing adds overhead to each I/O due to emulation layer
  - This indirection has the additional affect of eliminating the use of hardware acceleration that may be available in the physical device.
- Directed I/O has added enhancements to facilitate memory translation and ensure protection of memory that enables a device to directly DMA to/form host memory.
  - Bypass the VMM’s I/O emulation layer
  - Throughput improvement for the VMs



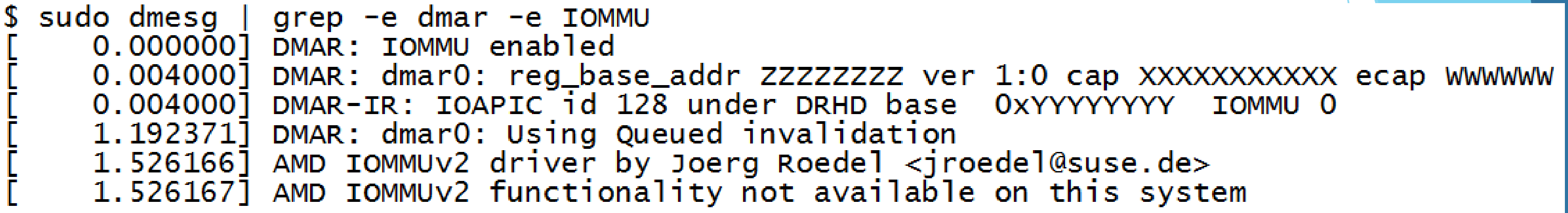
Drawbacks to Directed I/O

- One concern with direct assignment is that it has limited scalability
- Assignment is per physically-isolated functionality
  - A **physical** device can only be assigned to **one** VM.
  - For example, a dual port NIC allows for direct assignment to two VMs. (one port per VM)
  - Consider for a moment a fairly substantial server of the very near future
    - 4 physical CPU's
    - 12 cores per CPU
    - If we use the rule that one VM per core, it would need 48 physical ports.

Terminology relating to Directed I/O

Acronym	Expansion	Defined By	What is it?
I/O MMU	I/O Memory Management Unit	Common parlance	Translation mechanism in the system memory controller (North Bridge) that allows a device or set of devices to use translated addresses when accessing main memory. In many cases, it also translates interrupts coming from the devices as messages.
ATPT	Address Translation and Protection Table	PCI SIG	I/O MMU
VT-d, VT-d2	Virtualization Technology for Directed I/O	Intel	I/O MMU
DMAr	DMA Remapping	Intel, Microsoft	I/O MMU
IOMMU	I/O Memory Management Unit	AMD	I/O MMU

Is IOMMU (vt-d) enabled on your system?



```
$ sudo dmesg | grep -e dmar -e IOMMU
[    0.000000] DMAR: IOMMU enabled
[    0.004000] DMAR: dmar0: reg_base_addr ZZZZZZZZ ver 1:0 cap XXXXXXXXXXX ecap WWWWWW
[    0.004000] DMAR-IR: IOAPIC id 128 under DRHD base  0xYYYYYYYY  IOMMU 0
[    1.192371] DMAR: dmar0: Using Queued invalidation
[    1.526166] AMD IOMMUv2 driver by Joerg Roedel <jroedel@suse.de>
[    1.526167] AMD IOMMUv2 functionality not available on this system
```

IOMMUv2 includes:

- (a) I/O device assignment
- (b) DMA remapping
- (c) interrupt remapping
- (d) reliability features

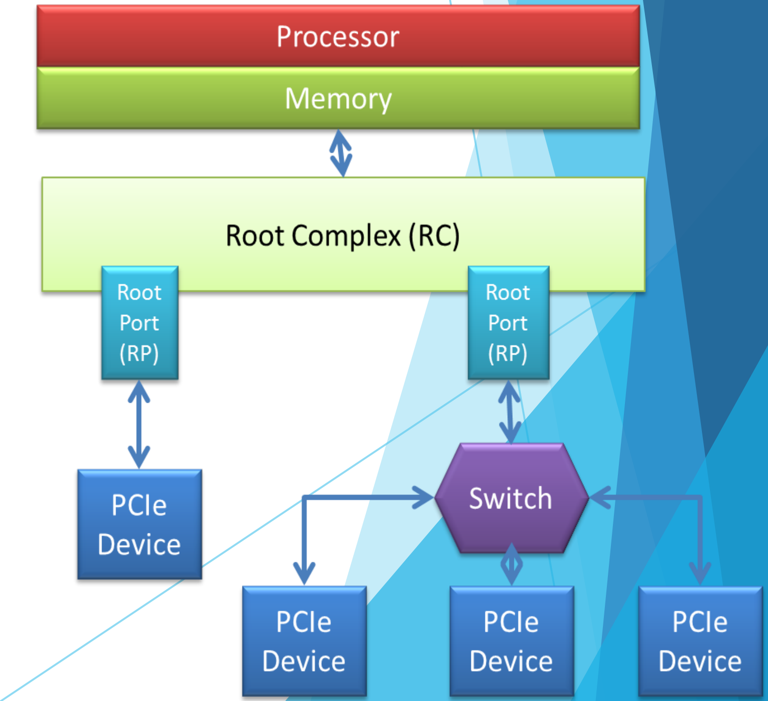
PCIe components

- Root Complex
  - A root complex connects the processor and memory subsystem to the PCIe switch fabric composed of one or more switch devices
  - Similar to a host bridge in a PCI system
    - Generate transaction requests on

      behalf of the processor, which is

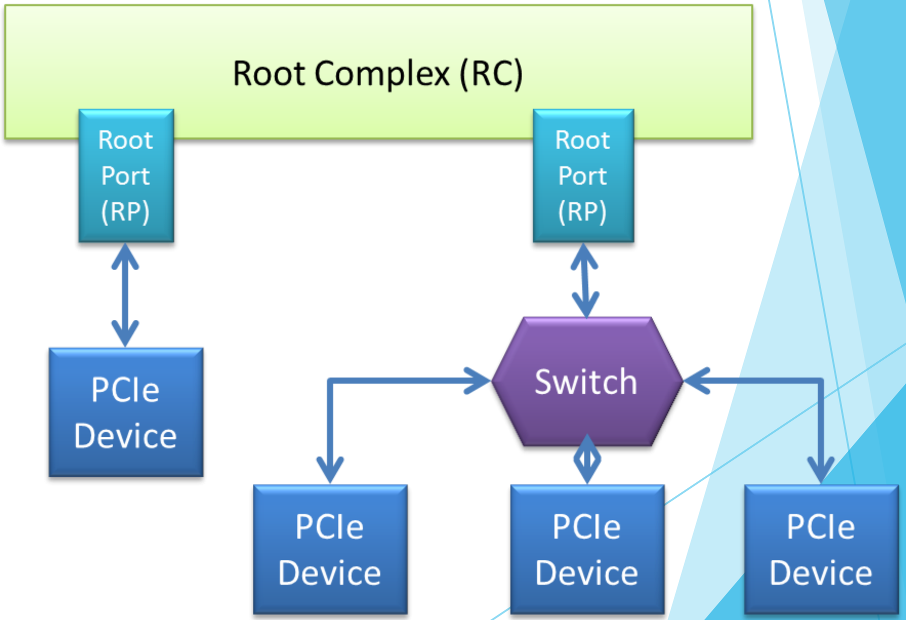
      interconnected through a local bus.
    - May contain more than one PCIe port

      and multiple switch devices.



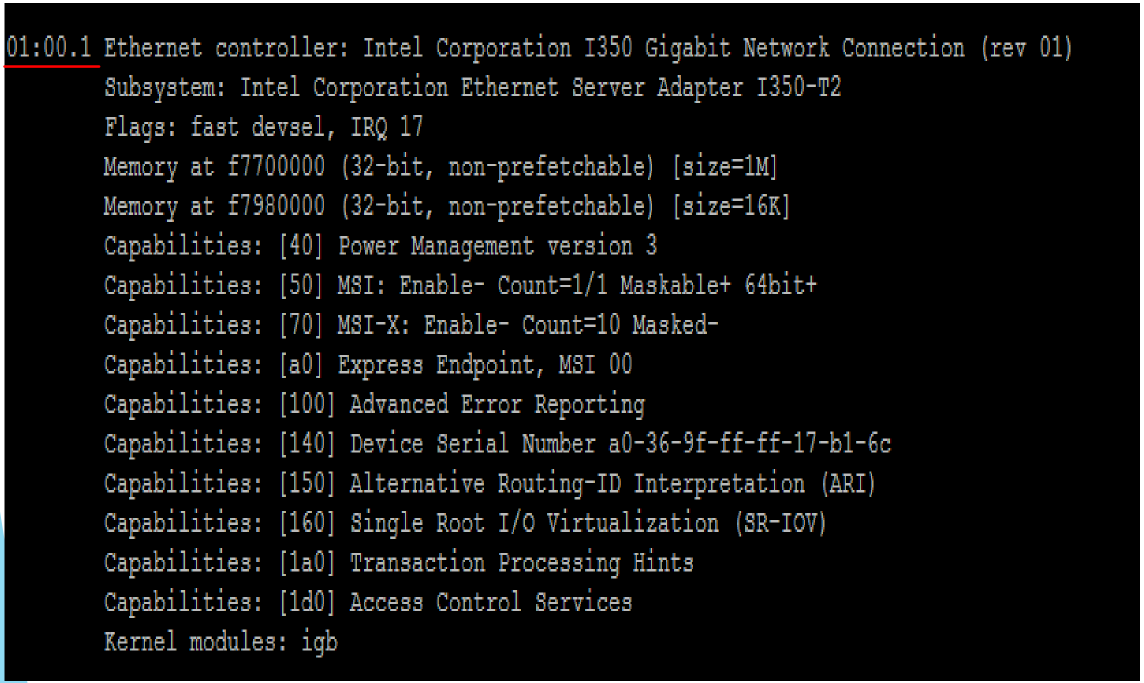
PCIe components

- Root Port (RP)
  - The portion of the motherboard that contains the host bridge. The host bridge allows the PCIe ports to talk to the rest of the computer

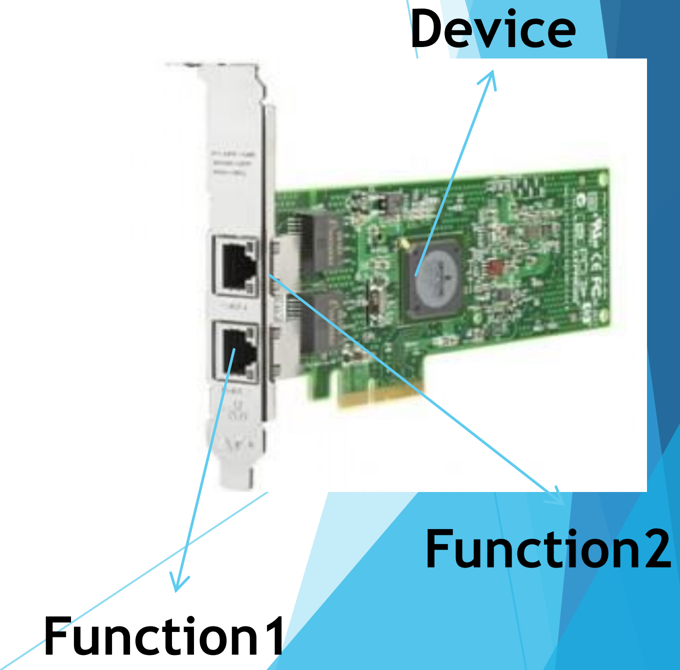


PCIe Device

- **Unique** PCI Function Address
  - Bus / Dev / **Function**
  - Command, lspci -v, can get PCI device information on linux

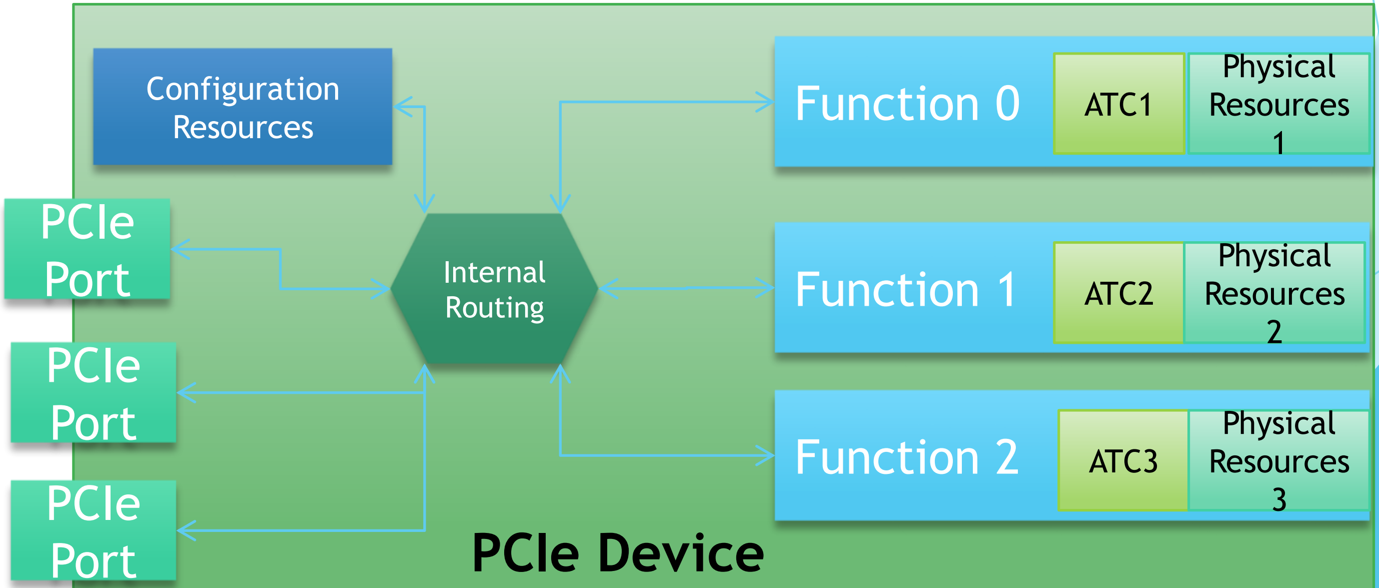


```
01:00.1 Ethernet controller: Intel Corporation I350 Gigabit Network Connection (rev 01)
        Subsystem: Intel Corporation Ethernet Server Adapter I350-T2
        Flags: fast devsel, IRQ 17
        Memory at f7700000 (32-bit, non-prefetchable) [size=1M]
        Memory at f7980000 (32-bit, non-prefetchable) [size=16K]
        Capabilities: [40] Power Management version 3
        Capabilities: [50] MSI: Enable- Count=1/1 Maskable+ 64bit+
        Capabilities: [70] MSI-X: Enable- Count=10 Masked-
        Capabilities: [a0] Express Endpoint, MSI 00
        Capabilities: [100] Advanced Error Reporting
        Capabilities: [140] Device Serial Number a0-36-9f-ff-ff-17-b1-6c
        Capabilities: [150] Alternative Routing-ID Interpretation (ARI)
        Capabilities: [160] Single Root I/O Virtualization (SR-IOV)
        Capabilities: [1a0] Transaction Processing Hints
        Capabilities: [1d0] Access Control Services
        Kernel modules: igb
```

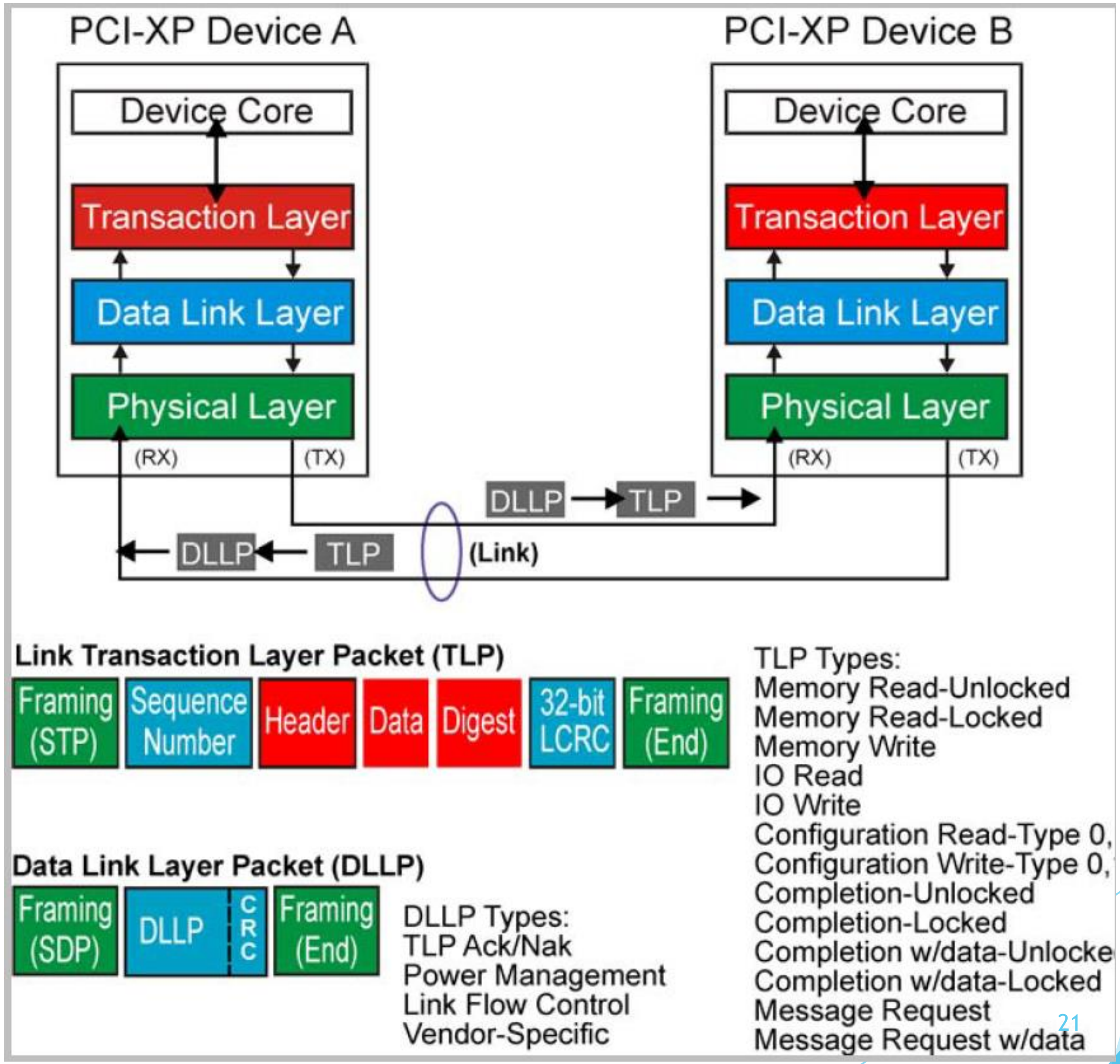


Example: Multi-Function Device

- The link and PCIe functionality shared by all functions is **managed through Function 0**
- All functions use a single Bus Number captured through the PCI enumeration process
- Each function can be assigned to an SI



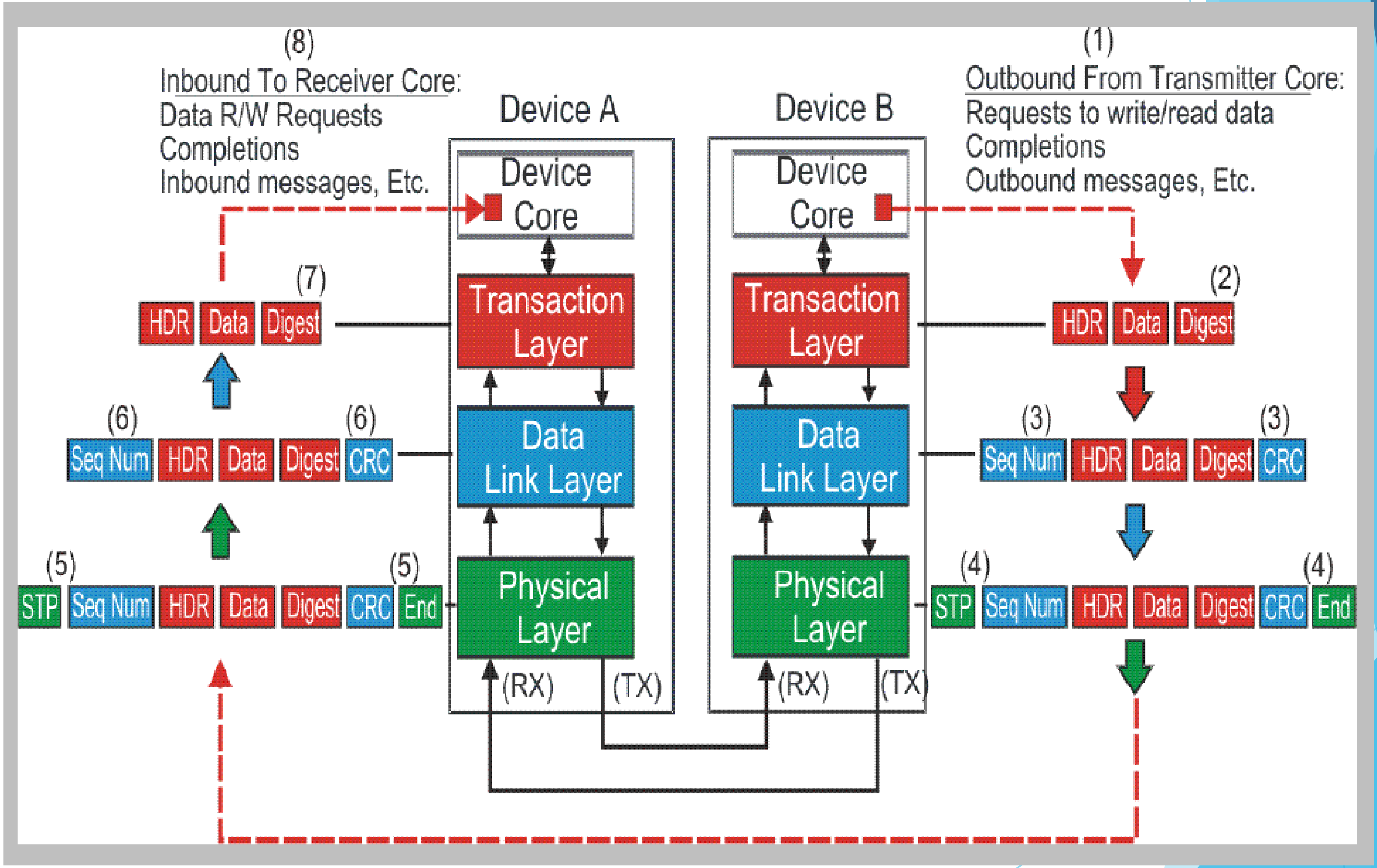
PCIe devices use TLP



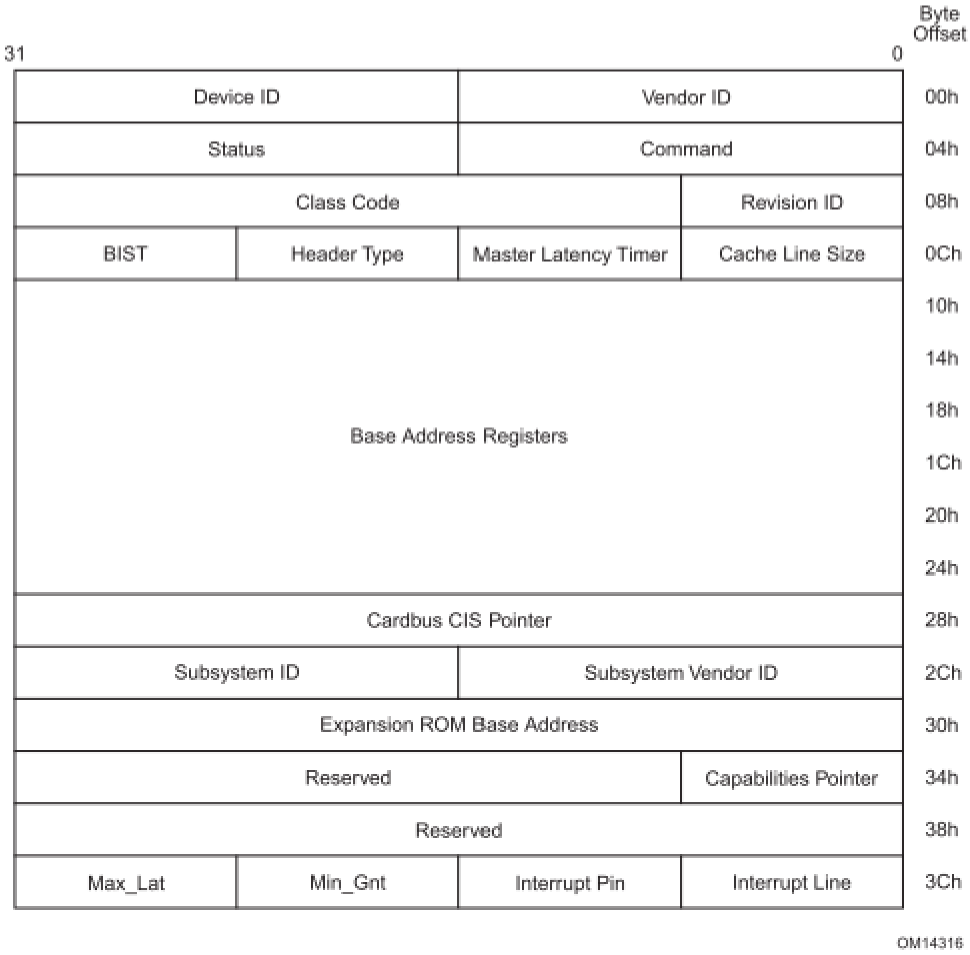
DLLP Types:
TLP Ack/Nak
Power Management
Link Flow Control
Vendor-Specific

TLP Types:
Memory Read-Unlocked
Memory Read-Locked
Memory Write
IO Read
IO Write
Configuration Read-Type 0,
Configuration Write-Type 0,
Completion-Unlocked
Completion-Locked
Completion w/data-Unlocke
Completion w/data-Locked
Message Request
Message Request w/data

TLP Assembly/Disassembly



Components in PCIe Device



31			0	Byte Offset
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Master Latency Timer	Cache Line Size	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

OM14316

Figure 3-2: Type 0 Configuration Space Header

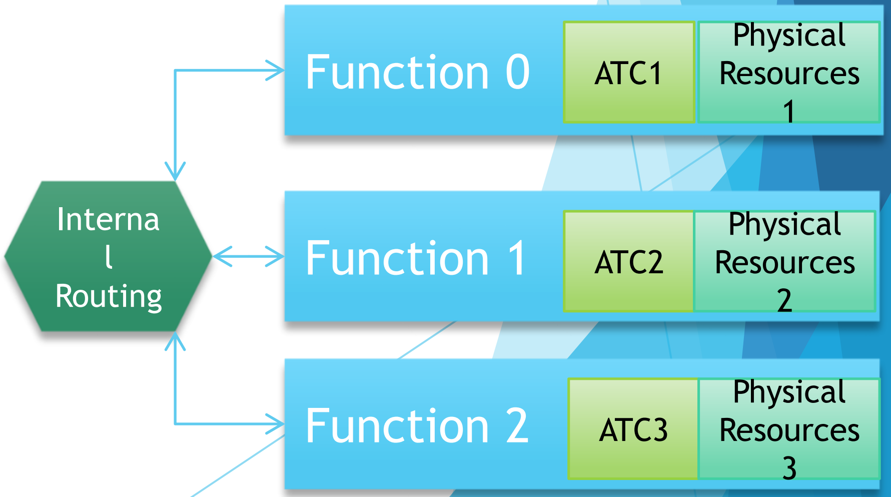
Configuration Resources

- Configuration Space
  - Devices will **allocate resource such as memory** and record the address into this configuration space
  - Reference:
    - PCI Local Bus Specification ver.2.3 Chap 6

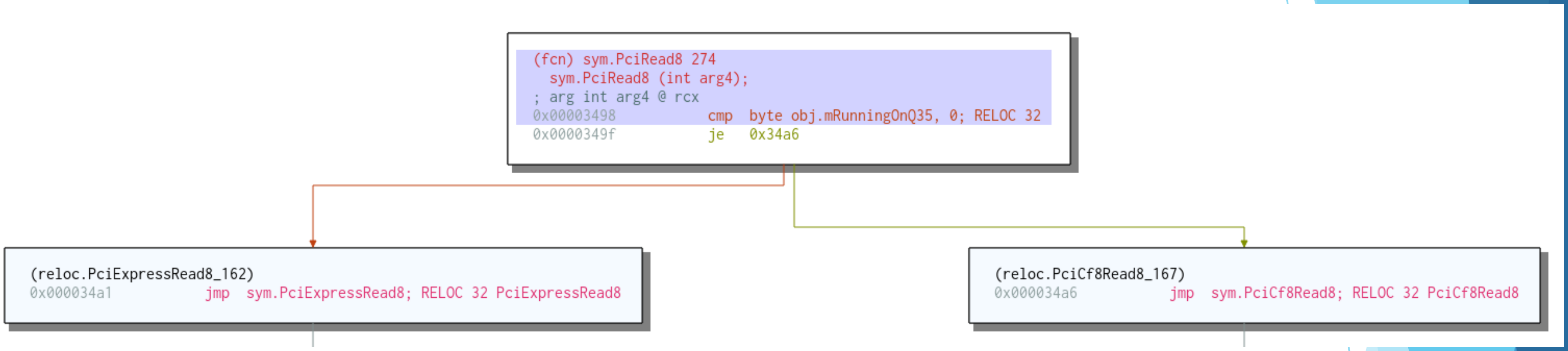
Components in PCIe Device

- Physical Resources
  - Memory which allocated from physical memory

- ATC - Address Translation Cache
  - A **hardware** stores recently used address translations.
  - This term is used instead of TLB buffer
  - To differentiate the **TLB used for I/O** from the **TLB used by the CPU**



Reversing PCI(e) device drivers – READ request

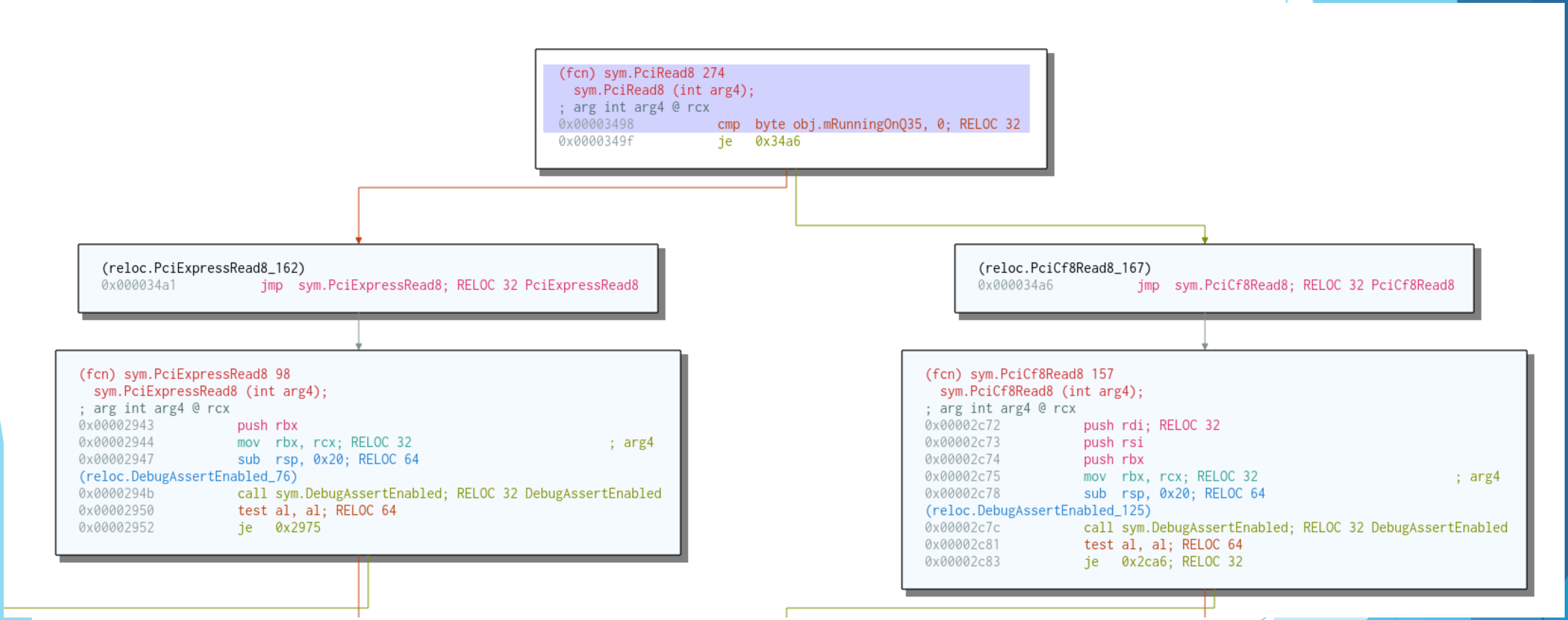


```
(fcn) sym.PciRead8 274
  sym.PciRead8 (int arg4);
; arg int arg4 @ rcx
0x00003498          cmp  byte obj.mRunningOnQ35, 0; RELOC 32
0x0000349f          je   0x34a6
```

```
(reloc.PciExpressRead8_162)
0x000034a1          jmp  sym.PciExpressRead8; RELOC 32 PciExpressRead8
```

```
(reloc.PciCf8Read8_167)
0x000034a6          jmp  sym.PciCf8Read8; RELOC 32 PciCf8Read8
```

Reading PCIe devices vs Reading PCI devices



```
(fcn) sym.PciRead8 274
  sym.PciRead8 (int arg4);
; arg int arg4 @ rcx
0x00003498          cmp  byte obj.mRunningOnQ35, 0; RELOC 32
0x0000349f          je   0x34a6
```

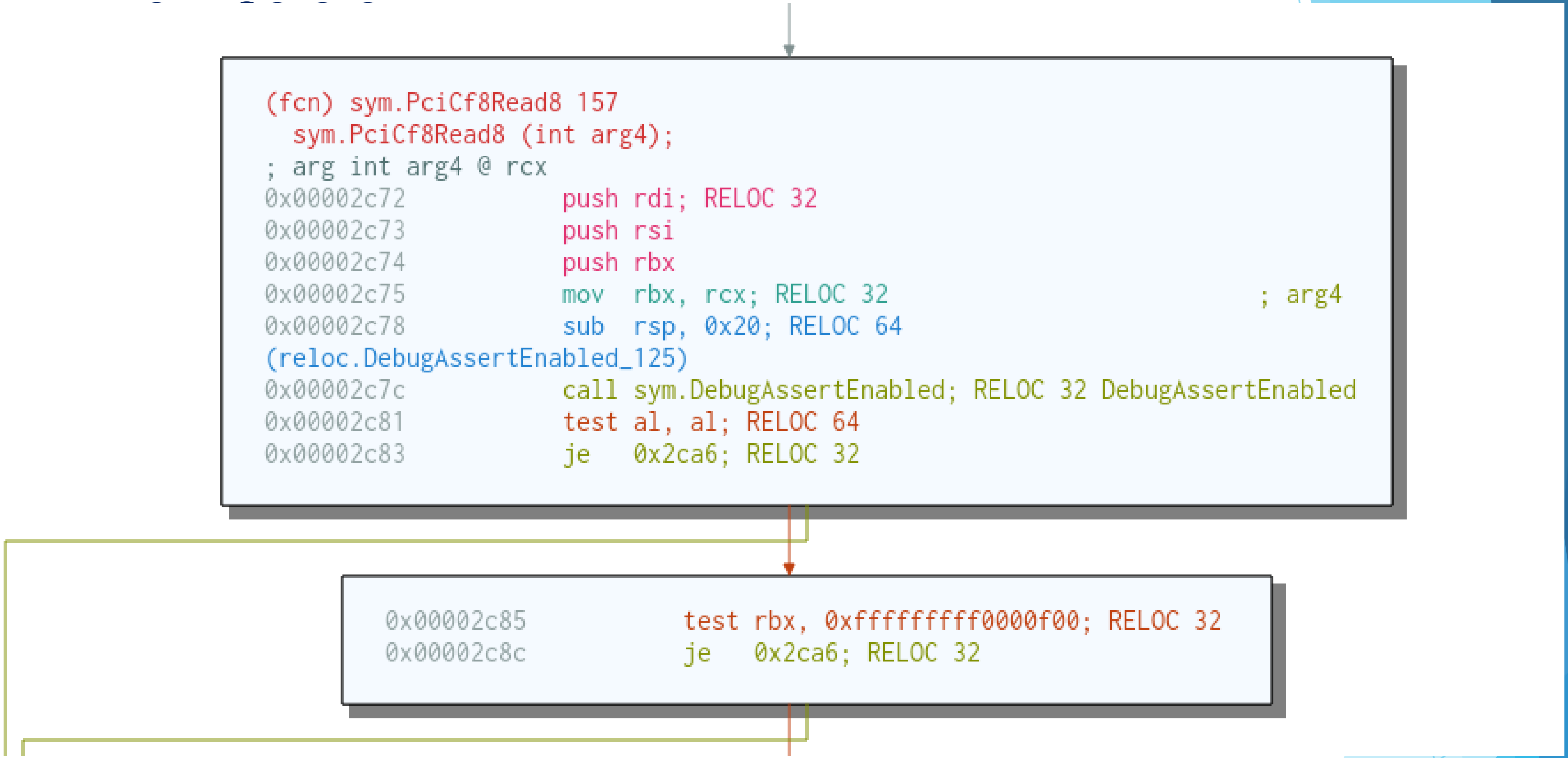
```
(reloc.PciExpressRead8_162)
0x000034a1          jmp  sym.PciExpressRead8; RELOC 32 PciExpressRead8
```

```
(fcn) sym.PciExpressRead8 98
  sym.PciExpressRead8 (int arg4);
; arg int arg4 @ rcx
0x00002943          push rbx
0x00002944          mov  rbx, rcx; RELOC 32                      ; arg4
0x00002947          sub  rsp, 0x20; RELOC 64
(reloc.DebugAssertEnabled_76)
0x0000294b          call sym.DebugAssertEnabled; RELOC 32 DebugAssertEnabled
0x00002950          test al, al; RELOC 64
0x00002952          je   0x2975
```

```
(reloc.PciCf8Read8_167)
0x000034a6          jmp  sym.PciCf8Read8; RELOC 32 PciCf8Read8
```

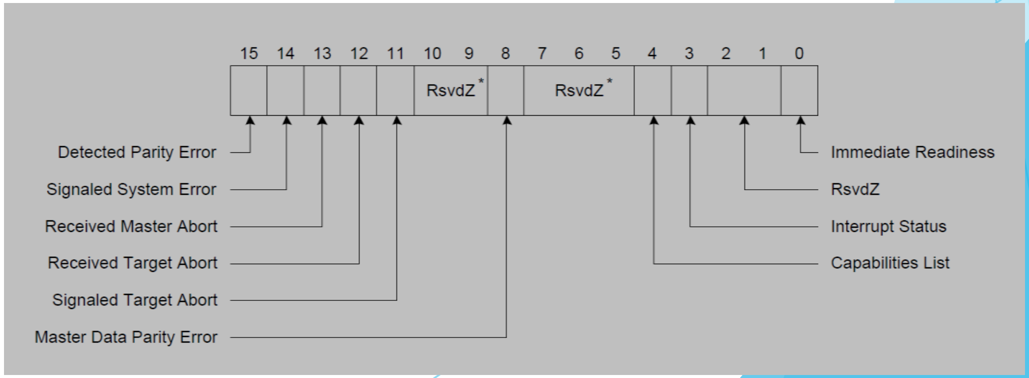
```
(fcn) sym.PciCf8Read8 157
  sym.PciCf8Read8 (int arg4);
; arg int arg4 @ rcx
0x00002c72          push rdi; RELOC 32
0x00002c73          push rsi
0x00002c74          push rbx
0x00002c75          mov  rbx, rcx; RELOC 32                      ; arg4
0x00002c78          sub  rsp, 0x20; RELOC 64
(reloc.DebugAssertEnabled_125)
0x00002c7c          call sym.DebugAssertEnabled; RELOC 32 DebugAssertEnabled
0x00002c81          test al, al; RELOC 64
0x00002c83          je   0x2ca6; RELOC 32
```

PCI Error? Set status register



```
(fcn) sym.PciCf8Read8 157
  sym.PciCf8Read8 (int arg4);
; arg int arg4 @ rcx
0x00002c72          push rdi; RELOC 32
0x00002c73          push rsi
0x00002c74          push rbx
0x00002c75          mov  rbx, rcx; RELOC 32                              ; arg4
0x00002c78          sub  rsp, 0x20; RELOC 64
(reloc.DebugAssertEnabled_125)
0x00002c7c          call sym.DebugAssertEnabled; RELOC 32 DebugAssertEnabled
0x00002c81          test al, al; RELOC 64
0x00002c83          je   0x2ca6; RELOC 32
```

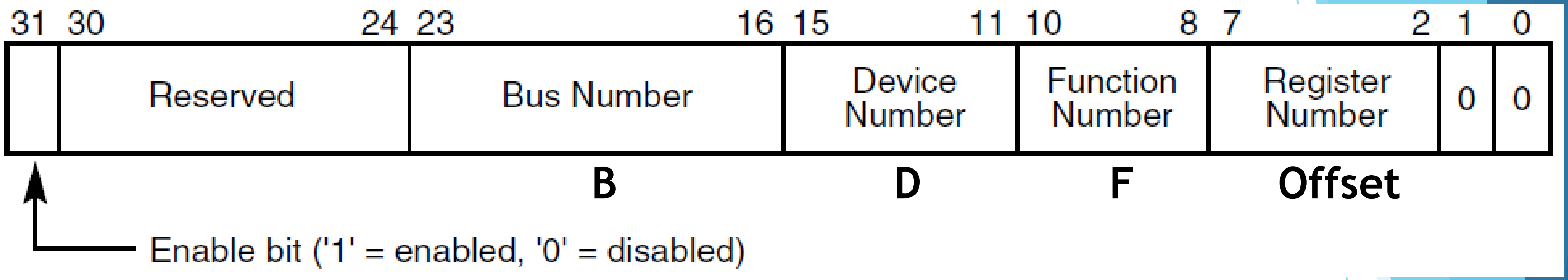
```
0x00002c85          test rbx, 0xffffffff0000f00; RELOC 32
0x00002c8c          je   0x2ca6; RELOC 32
```



Compatible PCI Configuration Space

- This refers to the software generation of PCI configuration transactions
  - those generated by the CPU/BIOS
- Compatible PCI provides 256 bytes of Configuration address space to the CPU/BIOS
- CPU/BIOS programs the registers contained therein to configure the device and system parameters
- Compatible PCI is configured using the port I/O address/data pair (CONFIG_ADDRESS, CONFIG_DATA)
- Two 32-bit I/O locations are used to generate configuration transactions
  - CF8h (CONFIG_ADDRESS)
  - CFCh (CONFIG_DATA)

I/O Port CONFIG_ADDRESS (CF8h)

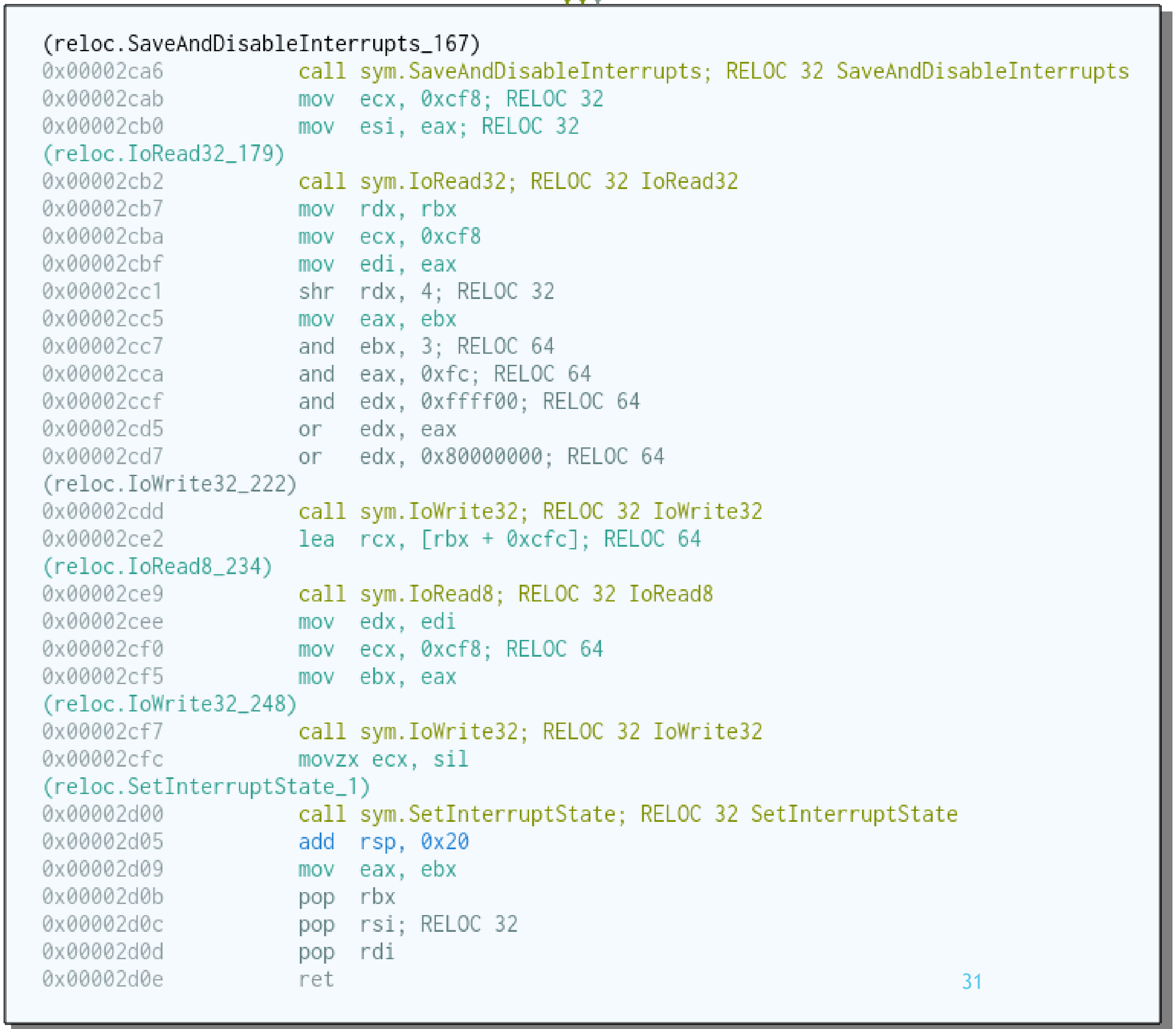


- 32 bits (**GIMME BUS 0, DEVICE 31 (0x1F), Function 0, offset 0x88**)
- Port CF8h
- Bit 31 when set, all reads and writes to CONFIG_DATA are PCI Configuration transactions
- Bits 30:24 are read-only and must return 0 when read
- Bits 23:16 select a specific Bus in the system (up to 256 buses)
- Bits 15:11 specify a Device on the given Bus (up to 32 devices)
- Bits 10:8 Specify the function of a device (up to 8 devices)
- Bits 7:0 Select an offset within the Configuration Space (256 bytes max, DWORD-aligned as bits 1:0 are hard-coded 0)
- Addresses are often given in B/D/F, Offset notation (also written as B:D:F, Offset)

I/O Port CONFIG_DATA (CFCh)

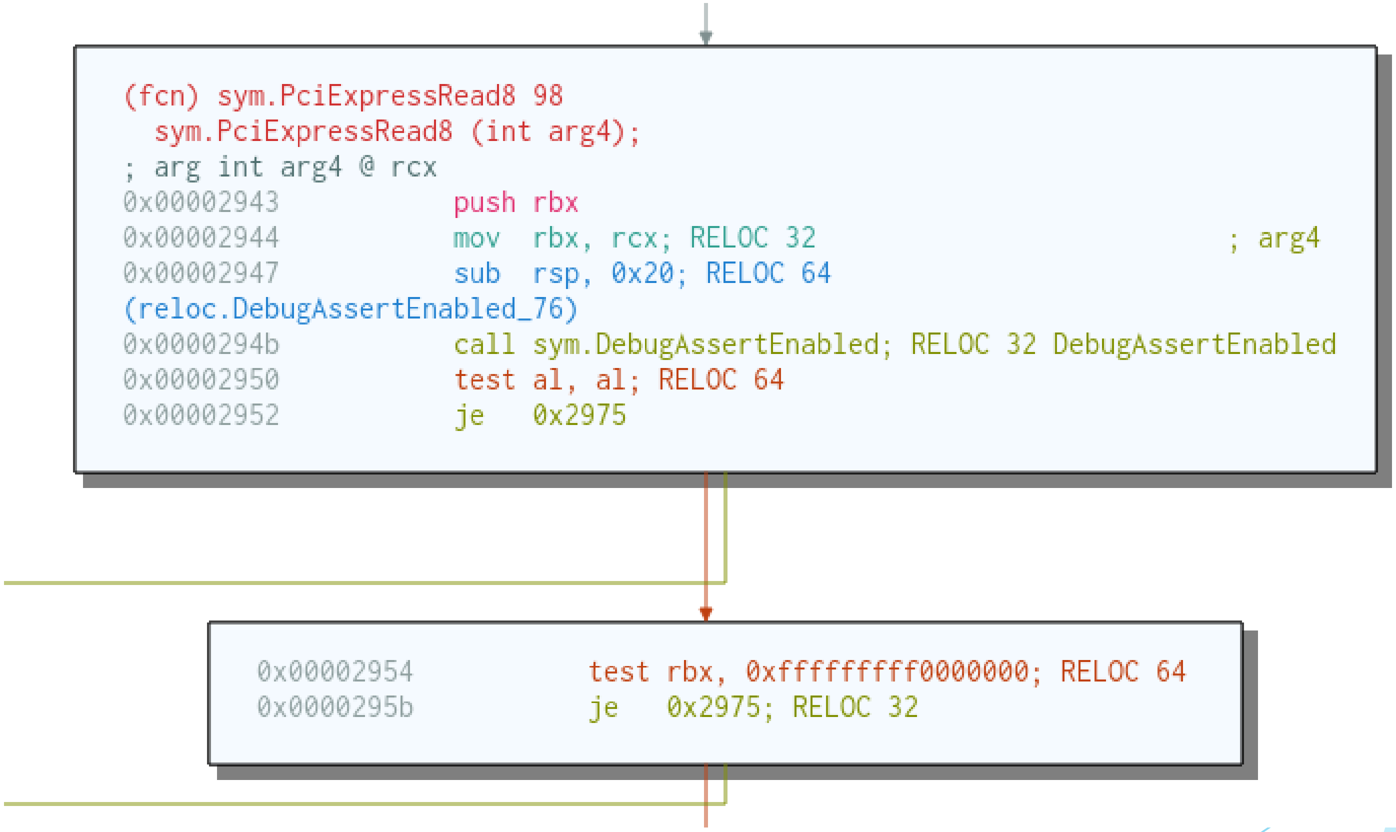
- CONFIG_DATA can be accessed in DWORD, WORD, or BYTE configurations
- Reads and Writes to CONFIG_DATA with Bit 31 in CONFIG_ADDRESS set/enabled results in a PCI Configuration transaction to the device specified in CONFIG_ADDRESS
- PCI spec says that if Bit 31 is not enabled, then the transaction is forwarded out as Port I/O

Performing PCI device READ request



```
(reloc.SaveAndDisableInterrupts_167)
0x00002ca6          call sym.SaveAndDisableInterrupts; RELOC 32 SaveAndDisableInterrupts
0x00002cab          mov  ecx, 0xcf8; RELOC 32
0x00002cb0          mov  esi, eax; RELOC 32
(reloc.IoRead32_179)
0x00002cb2          call sym.IoRead32; RELOC 32 IoRead32
0x00002cb7          mov  rdx, rbx
0x00002cba          mov  ecx, 0xcf8
0x00002cbf          mov  edi, eax
0x00002cc1          shr  rdx, 4; RELOC 32
0x00002cc5          mov  eax, ebx
0x00002cc7          and  ebx, 3; RELOC 64
0x00002cca          and  eax, 0xfc; RELOC 64
0x00002ccf          and  edx, 0xffff00; RELOC 64
0x00002cd5          or   edx, eax
0x00002cd7          or   edx, 0x80000000; RELOC 64
(reloc.IoWrite32_222)
0x00002cdd          call sym.IoWrite32; RELOC 32 IoWrite32
0x00002ce2          lea  rcx, [rbx + 0xcfc]; RELOC 64
(reloc.IoRead8_234)
0x00002ce9          call sym.IoRead8; RELOC 32 IoRead8
0x00002cee          mov  edx, edi
0x00002cf0          mov  ecx, 0xcf8; RELOC 64
0x00002cf5          mov  ebx, eax
(reloc.IoWrite32_248)
0x00002cf7          call sym.IoWrite32; RELOC 32 IoWrite32
0x00002cfc          movzx ecx, sil
(reloc.SetInterruptState_1)
0x00002d00          call sym.SetInterruptState; RELOC 32 SetInterruptState
0x00002d05          add  rsp, 0x20
0x00002d09          mov  eax, ebx
0x00002d0b          pop  rbx
0x00002d0c          pop  rsi; RELOC 32
0x00002d0d          pop  rdi
0x00002d0e          ret
```

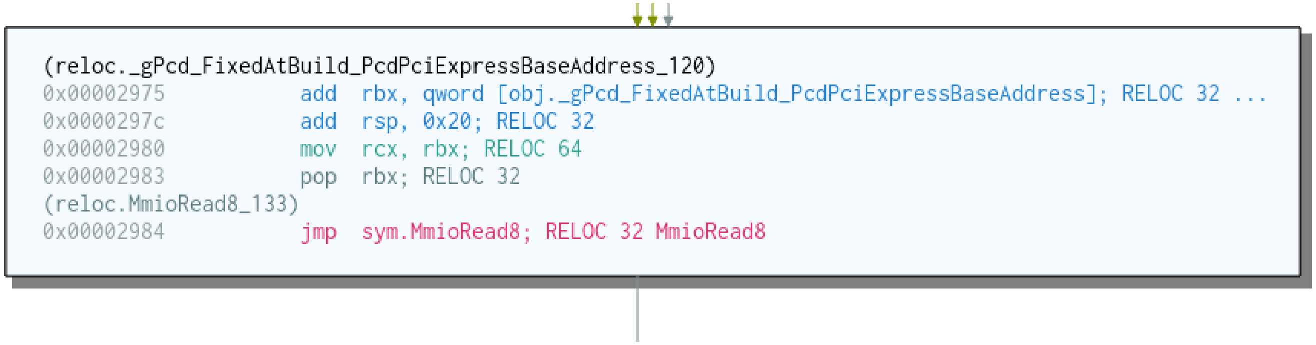
PCI Express – READ example



```
(fcn) sym.PciExpressRead8 98
  sym.PciExpressRead8 (int arg4);
; arg int arg4 @ rcx
0x00002943          push rbx
0x00002944          mov  rbx, rcx; RELOC 32                         ; arg4
0x00002947          sub  rsp, 0x20; RELOC 64
(reloc.DebugAssertEnabled_76)
0x0000294b          call sym.DebugAssertEnabled; RELOC 32 DebugAssertEnabled
0x00002950          test al, al; RELOC 64
0x00002952          je   0x2975
```

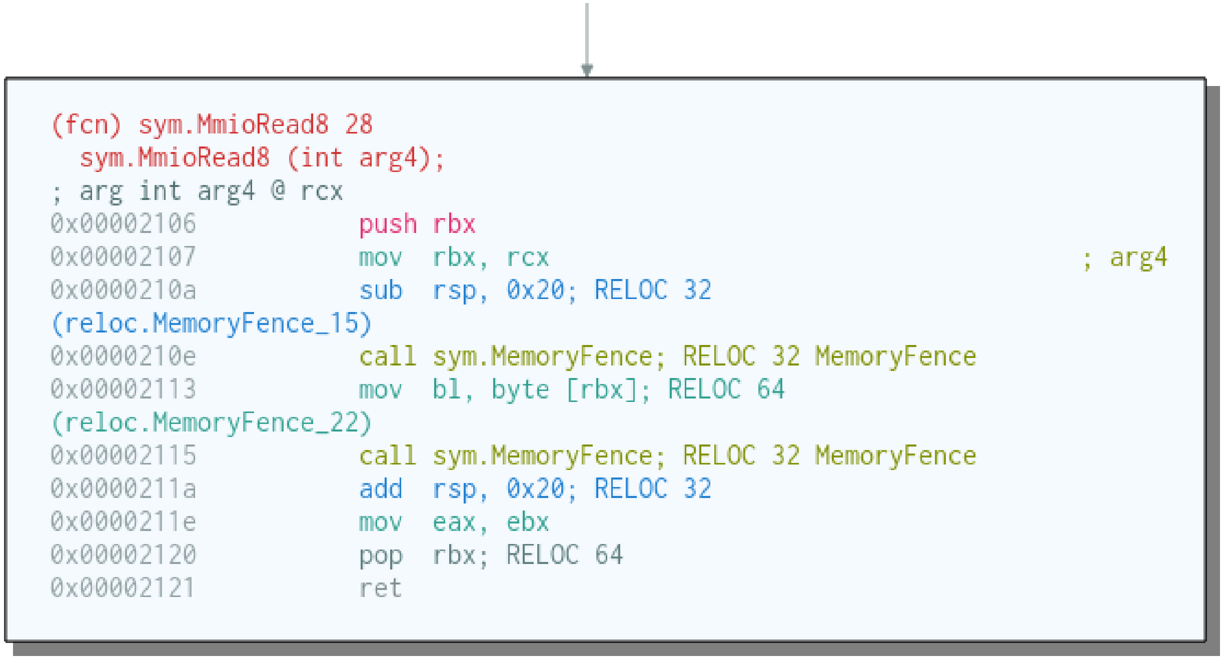
```
0x00002954          test rbx, 0xfffffffff0000000; RELOC 64
0x0000295b          je   0x2975; RELOC 32
```

If everything is OK, perform MmioRead8()



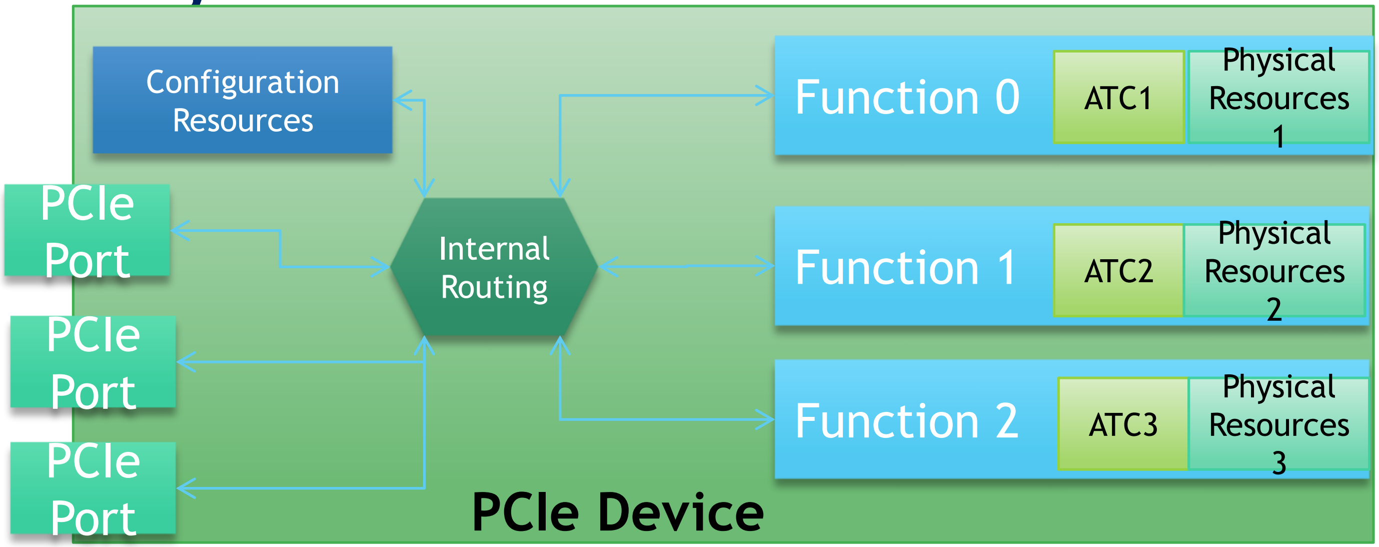
```
(reloc._gPcd_FixedAtBuild_PcdPciExpressBaseAddress_120)
0x00002975          add  rbx, qword [obj._gPcd_FixedAtBuild_PcdPciExpressBaseAddress]; RELOC 32 ...
0x0000297c          add  rsp, 0x20; RELOC 32
0x00002980          mov  rcx, rbx; RELOC 64
0x00002983          pop  rbx; RELOC 32
(reloc.MmioRead8_133)
0x00002984          jmp  sym.MmioRead8; RELOC 32 MmioRead8
```

RCX is transformed as a Base+Offset



```
(fcn) sym.MmioRead8 28
  sym.MmioRead8 (int arg4);
; arg int arg4 @ rcx
0x00002106          push rbx
0x00002107          mov  rbx, rcx                                   ; arg4
0x0000210a          sub  rsp, 0x20; RELOC 32
(reloc.MemoryFence_15)
0x0000210e          call sym.MemoryFence; RELOC 32 MemoryFence
0x00002113          mov  bl, byte [rbx]; RELOC 64
(reloc.MemoryFence_22)
0x00002115          call sym.MemoryFence; RELOC 32 MemoryFence
0x0000211a          add  rsp, 0x20; RELOC 32
0x0000211e          mov  eax, ebx
0x00002120          pop  rbx; RELOC 64
0x00002121          ret
```

Physical V.S. Virtual



Configuration Resources

PCIe Port

PCIe Port

PCIe Port

Internal Routing

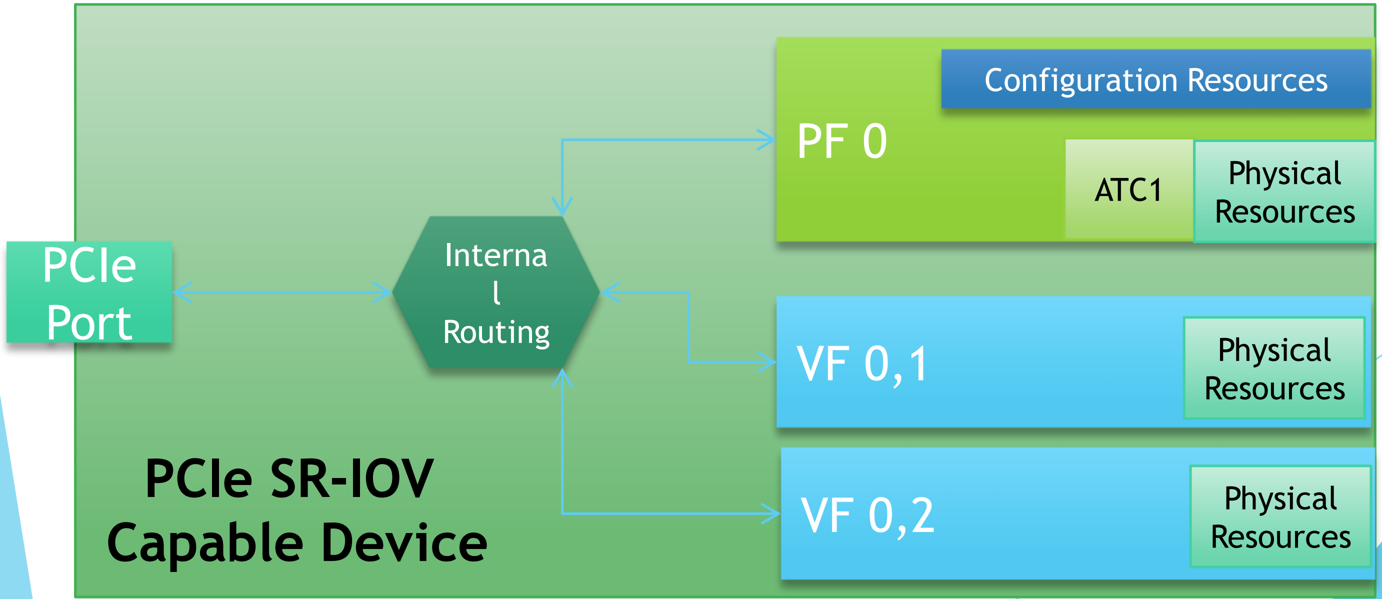
Function 0 ATC1 Physical Resources 1

Function 1 ATC2 Physical Resources 2

Function 2 ATC3 Physical Resources 3

PCIe Device

Physical



PCIe Port

Interna l Routing

PF 0 Configuration Resources ATC1 Physical Resources

VF 0,1 Physical Resources

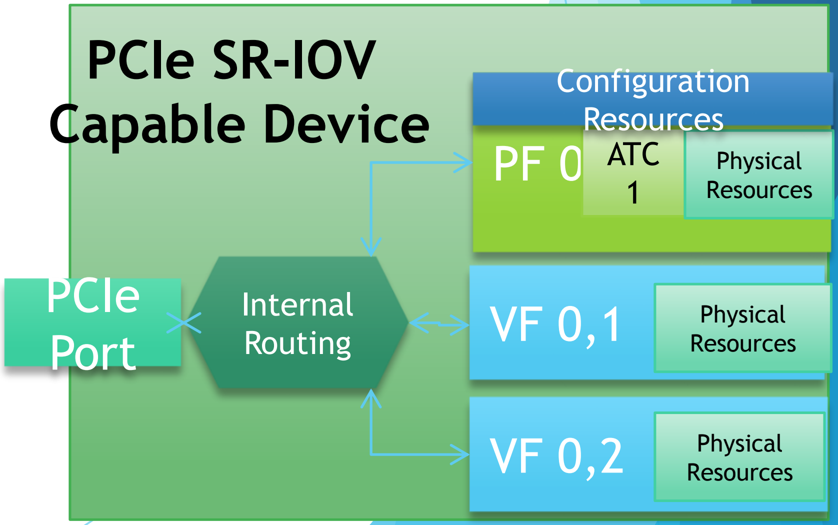
VF 0,2 Physical Resources

PCIe SR-IOV Capable Device

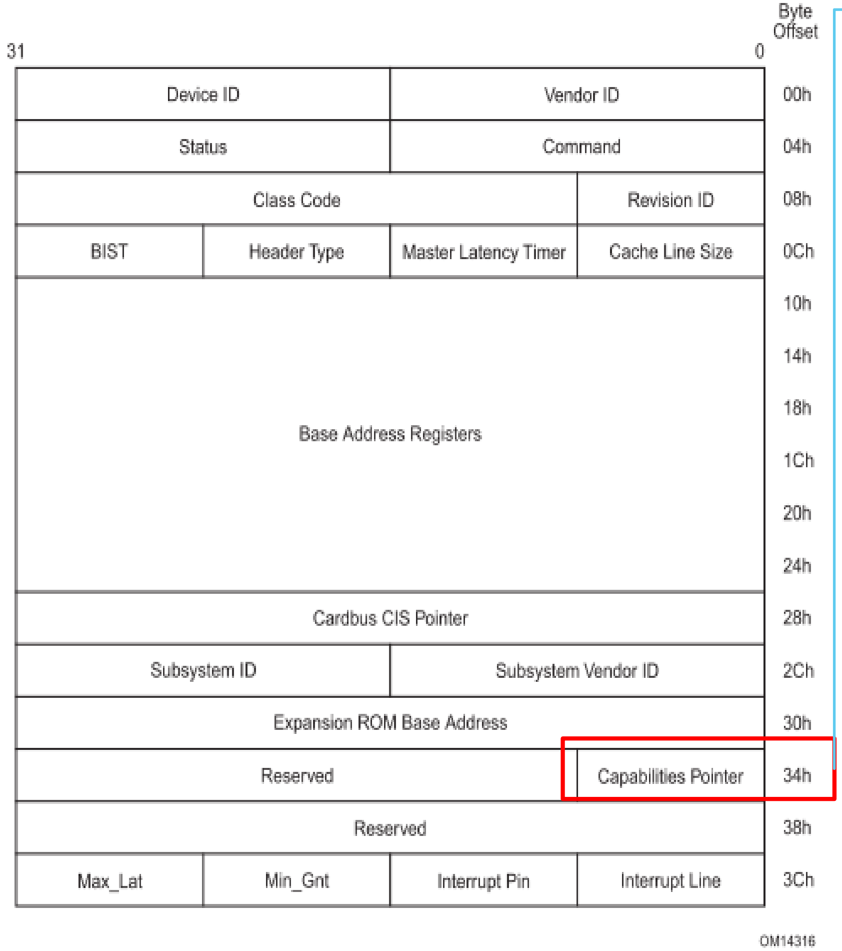
Virtual

PCIe SR-IOV Capable Device

- SR-IOV
  - A technique performs and manages PCIe Virtualization.
- PF – physical Function
  - Provide full PCIe functionality, including the SR-IOV capabilities
  - Discover the **page sizes** supported by a PF and its associated VF
- VF – virtual Function
  - A **“light-weight”** PCIe function that is **directly accessible by an SI,** including an isolated memory space, a work queue, interrupts and command processing.
  - **For data movement**
  - Can be optionally migrated form one PF to another PF
  - Can be serially shared by different SI



Extended Capabilities



31			0	Byte Offset
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Master Latency Timer	Cache Line Size	0Ch
Base Address Registers				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

OM14316

Figure 3-2: Type 0 Configuration Space Header

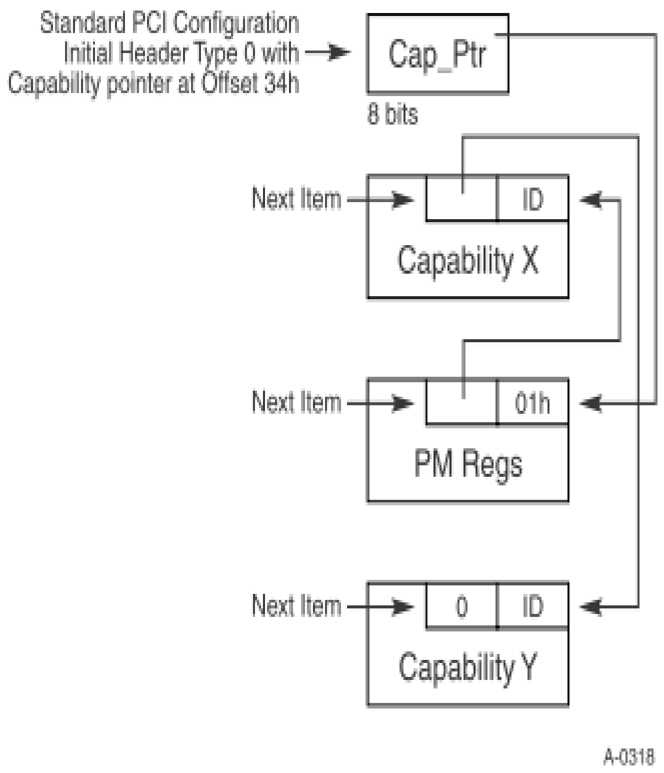
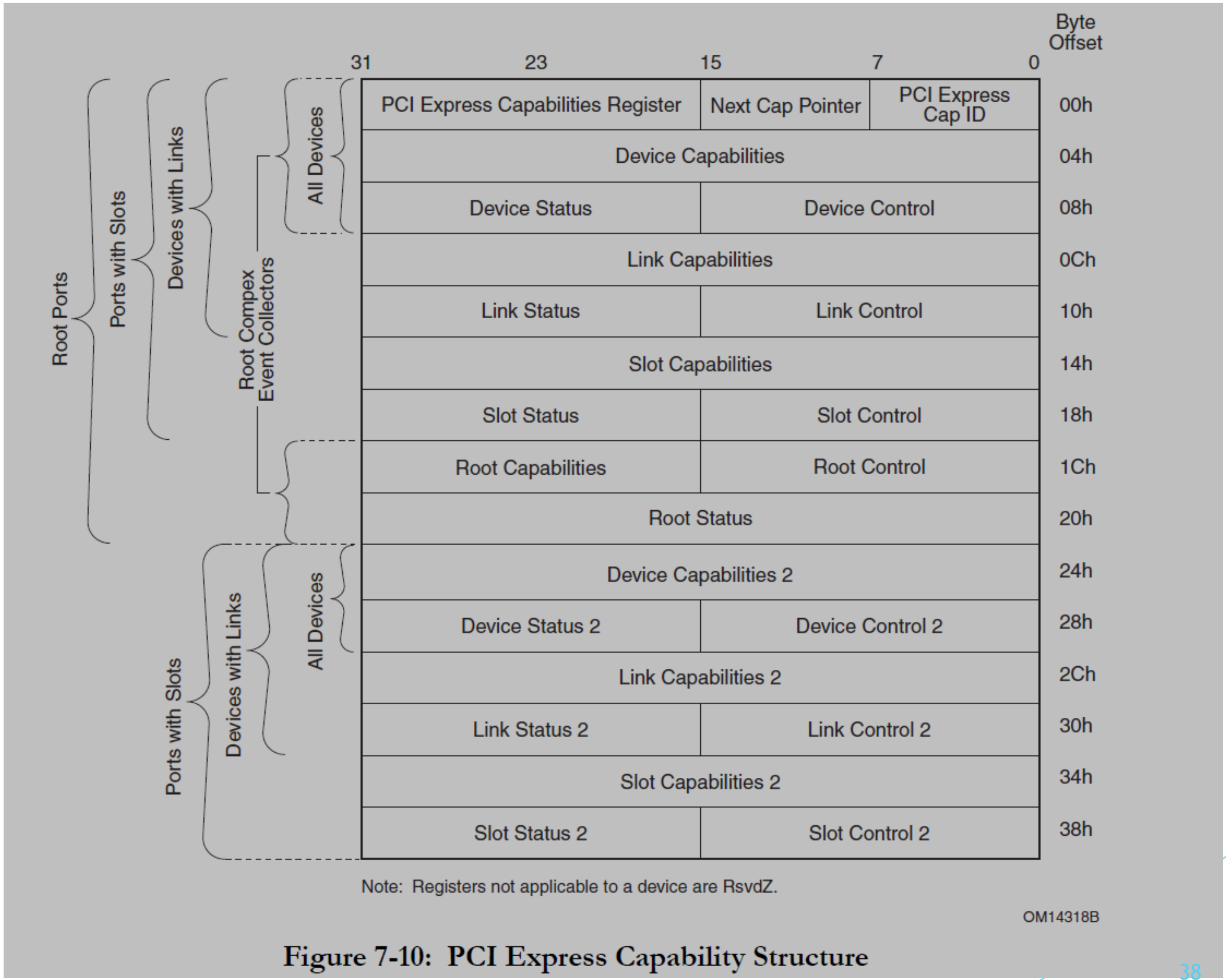


Figure 3-2: Capabilities Linked List

PCI Express Capability Structure



31 23	15	7 0	Byte Offset
PCI Express Capabilities Register	Next Cap Pointer	PCI Express Cap ID	00h
Device Capabilities			04h
Device Status	Device Control		08h
Link Capabilities			0Ch
Link Status	Link Control		10h
Slot Capabilities			14h
Slot Status	Slot Control		18h
Root Capabilities	Root Control		1Ch
Root Status			20h
Device Capabilities 2			24h
Device Status 2	Device Control 2		28h
Link Capabilities 2			2Ch
Link Status 2	Link Control 2		30h
Slot Capabilities 2			34h
Slot Status 2	Slot Control 2		38h

Note: Registers not applicable to a device are RsvdZ.

OM14318B

Figure 7-10: PCI Express Capability Structure

PCI Express Capabilities Register

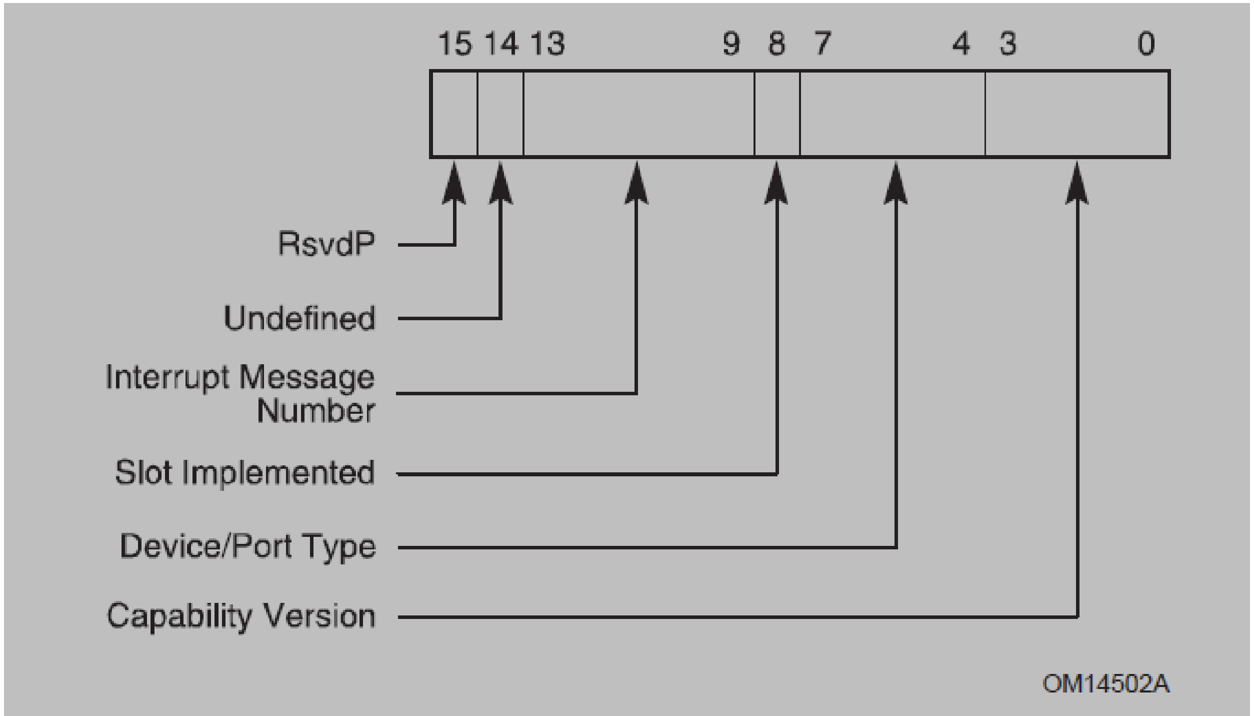
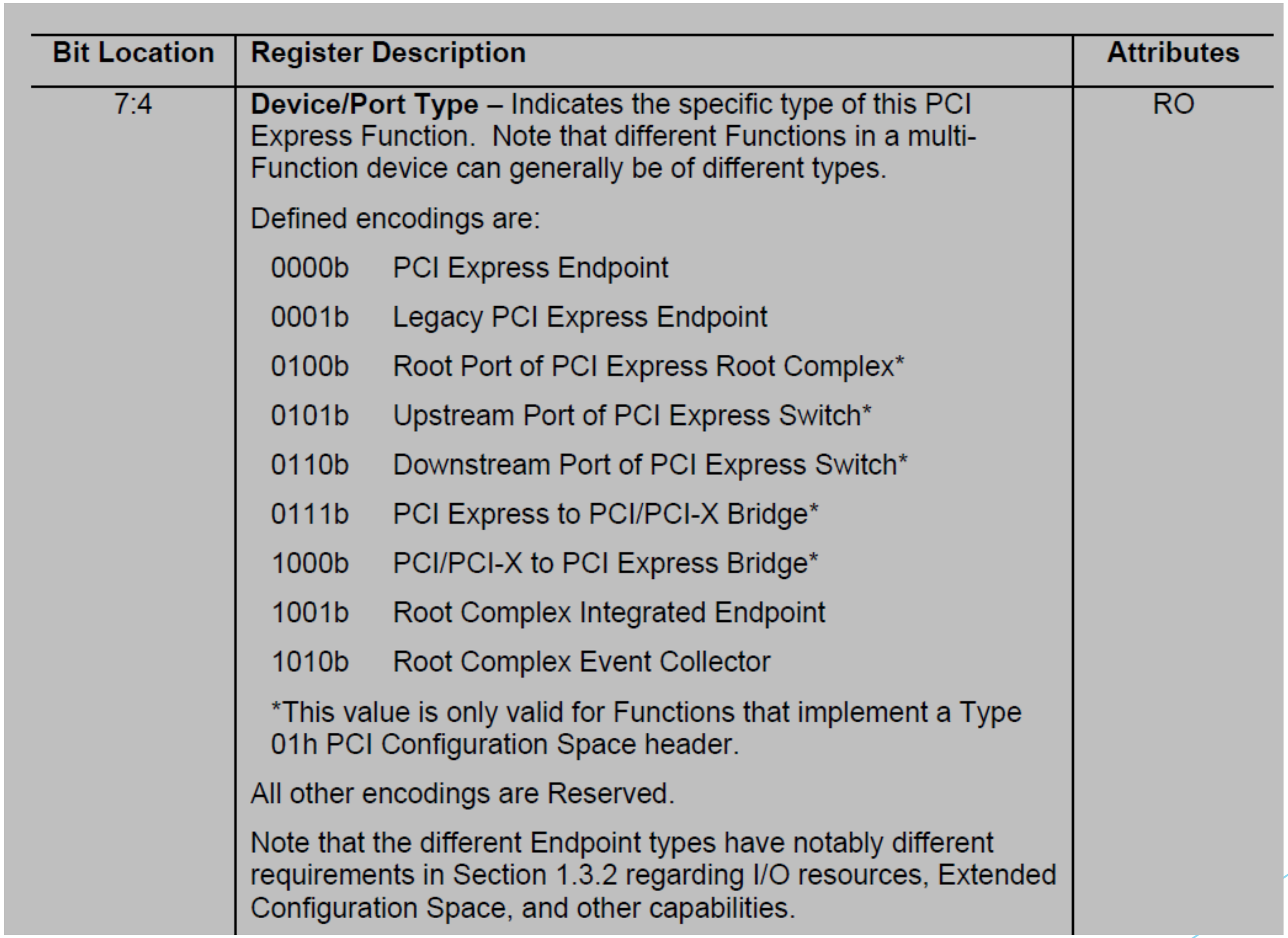


Figure 7-12: PCI Express Capabilities Register

PCI Express Device/Port TYPE



Bit Location	Register Description	Attributes
7:4	**Device/Port Type** – Indicates the specific type of this PCI Express Function. Note that different Functions in a multi-Function device can generally be of different types. Defined encodings are: 0000b PCI Express Endpoint 0001b Legacy PCI Express Endpoint 0100b Root Port of PCI Express Root Complex* 0101b Upstream Port of PCI Express Switch* 0110b Downstream Port of PCI Express Switch* 0111b PCI Express to PCI/PCI-X Bridge* 1000b PCI/PCI-X to PCI Express Bridge* 1001b Root Complex Integrated Endpoint 1010b Root Complex Event Collector *This value is only valid for Functions that implement a Type 01h PCI Configuration Space header. All other encodings are Reserved. Note that the different Endpoint types have notably different requirements in Section 1.3.2 regarding I/O resources, Extended Configuration Space, and other capabilities.	RO

SR-IOV Extended Capabilities

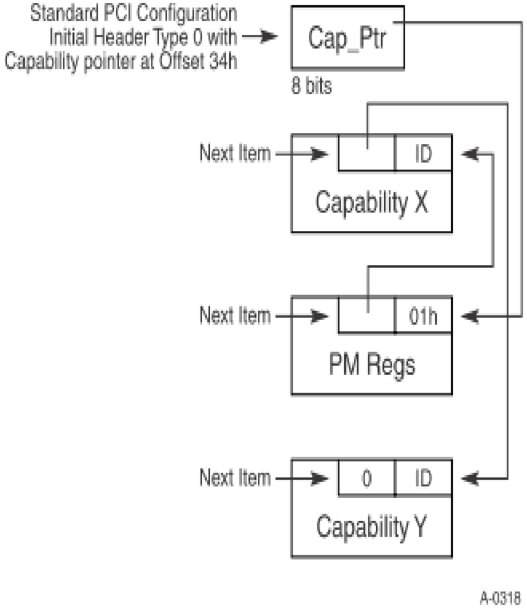
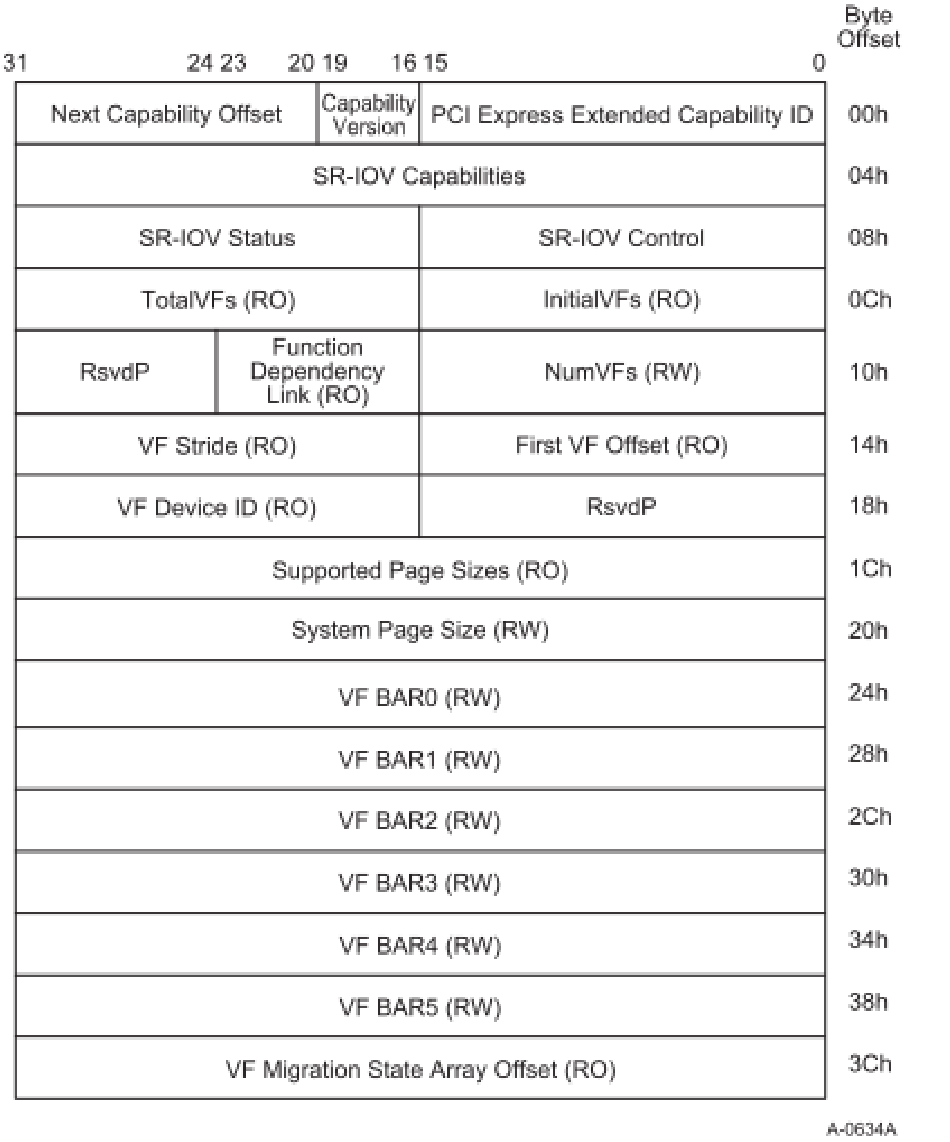


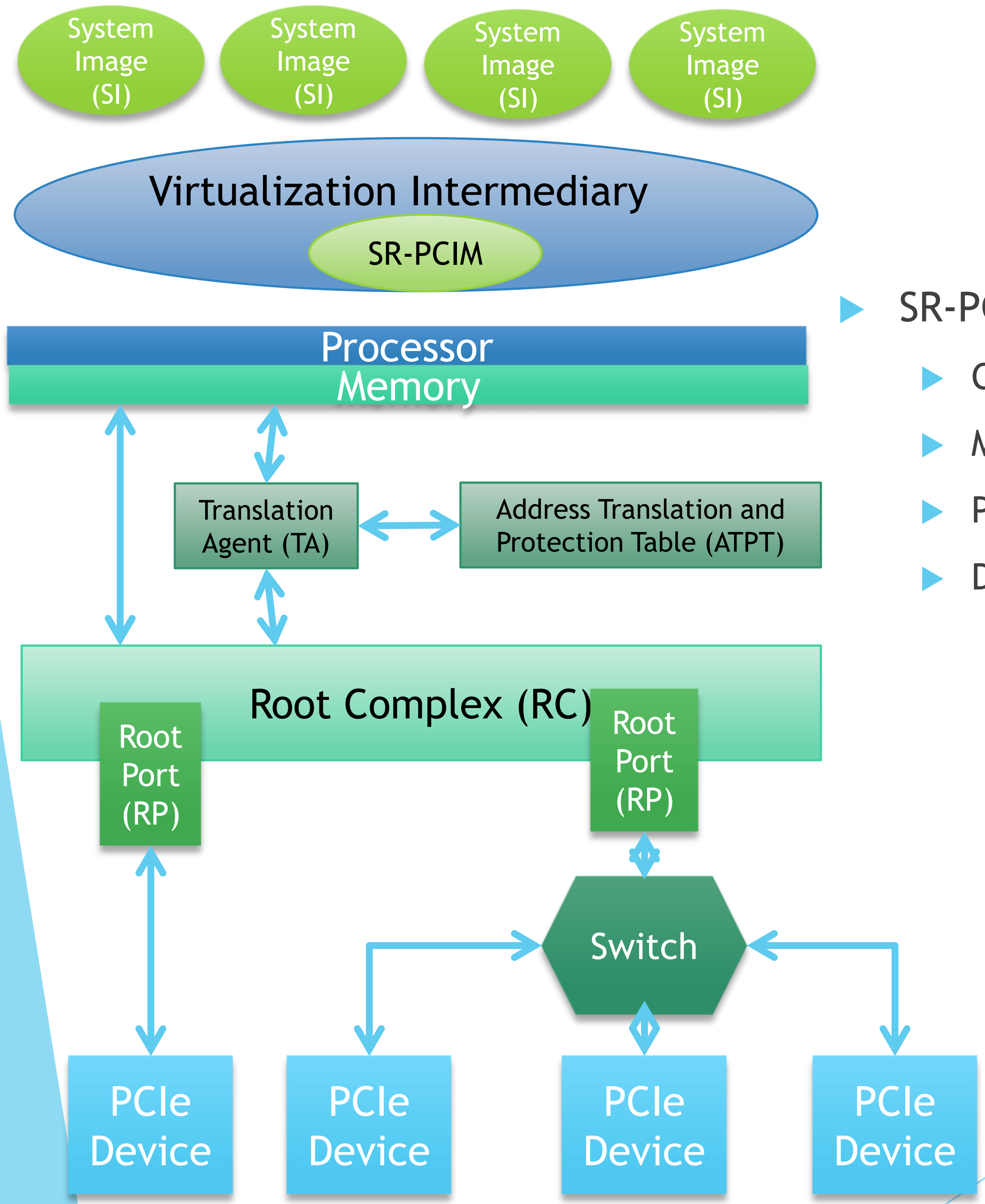
Figure 3-2: Capabilities Linked List



31 24	23 20	19 16	15 0	Byte Offset
Next Capability Offset		Capability Version	PCI Express Extended Capability ID	00h
SR-IOV Capabilities				04h
SR-IOV Status			SR-IOV Control	08h
TotalVFs (RO)			InitialVFs (RO)	0Ch
RsvdP	Function Dependency Link (RO)		NumVFs (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

A-0634A

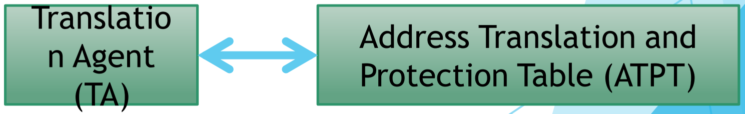
Figure 3-1: Single Root I/O Virtualization Extended Capabilities



- SR-PCIM
  - Configure SR-IOV Capability
  - Management of PFs and VFs
  - Processing of error events
  - Device controls
    - Power management
    - Hot-plug

Components of SR-IOV

- TA – Translation Agent
  - **Translate address** within a **PCIe transaction** into the associated platform **physical address.**
  - Hardware or combination of hardware and software
  - A TA may also support to enable a PCIe function to obtain address translations **a priori to DMA access** to the associated memory.

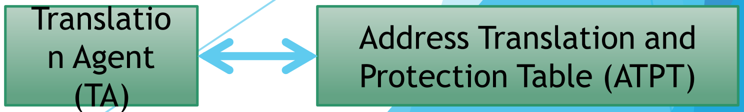


Translation Agent (TA)

Address Translation and Protection Table (ATPT)

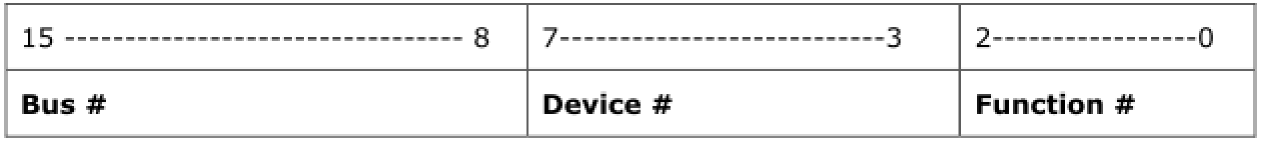
Components of SR-IOV

- ATPT – **A**ddress **T**ranslation and **P**rotection **T**able
  - Contain the set of address translations accessed by a TA to Process PCIe requests
    - DMA Read/Write
    - Interrupt requests
  - DMA Read/Write requests are translated through a combination of the **Routing ID** and **the address contained within a PCIe transaction**
  - In PCIe, interrupts are treated as memory write operations.
    - Though the combination of the Routing ID and the address contained within a PCIe transaction as well



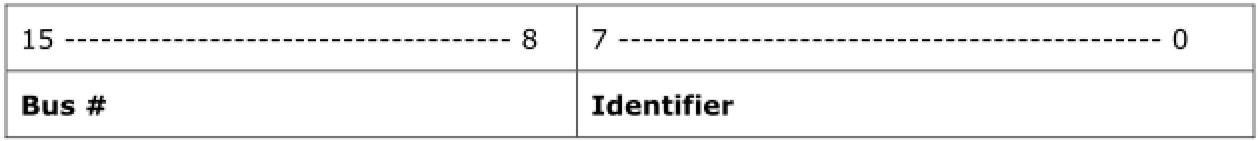
ARI – Alternative Routing ID Interpretation

- Routing ID is used to forward requests to the corresponding PFs and VFs
- All VFs and PFs must have distinct Routing IDs
- ARI provides a mechanism to allow single PCIe component to support **up to 256 functions.**
  - Originally there are **8 functions** at most in a PCIe.



15 -------------------------------- 8	7-------------------------3	2---------------0
Bus #	**Device #**	**Function #**

Table 1. Traditional Routing



15 ------------------------------------ 8	7 -- 0
Bus #	**Identifier**

Table 2. Alternate Routing

Figure from Intel PCI-SIG SR_IOV pr

ARI – Alternative Routing ID Interpretation

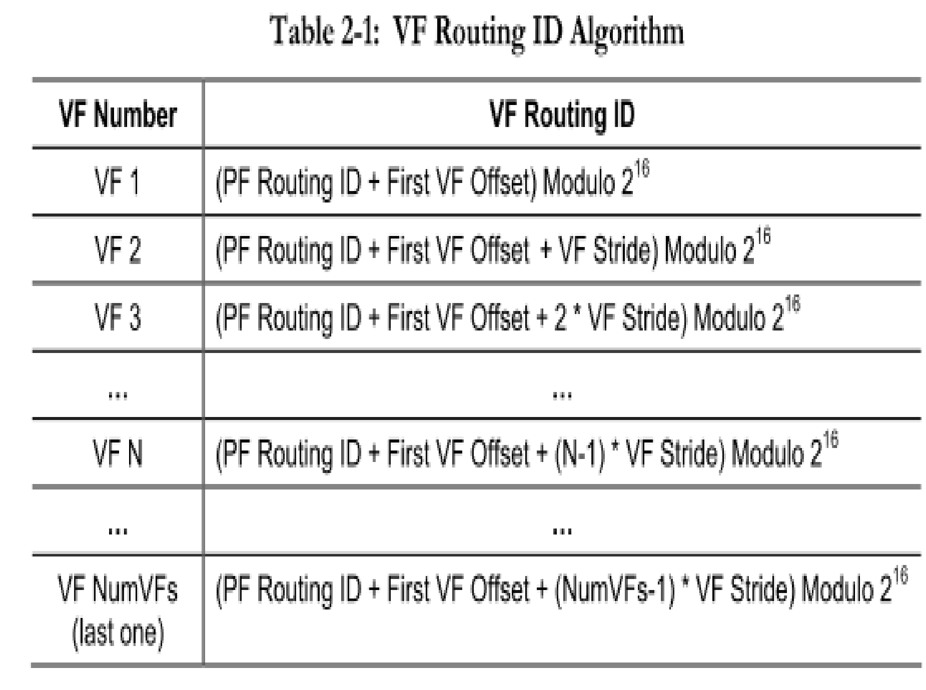


Table 2-1: VF Routing ID Algorithm

VF Number	VF Routing ID
VF 1	(PF Routing ID + First VF Offset) Modulo 2^{16}
VF 2	(PF Routing ID + First VF Offset + VF Stride) Modulo 2^{16}
VF 3	(PF Routing ID + First VF Offset + 2 * VF Stride) Modulo 2^{16}
...	...
VF N	(PF Routing ID + First VF Offset + (N-1) * VF Stride) Modulo 2^{16}
...	...
VF NumVFs (last one)	(PF Routing ID + First VF Offset + (NumVFs-1) * VF Stride) Modulo 2^{16}

Figure from SR-IOV Specification revision . . .

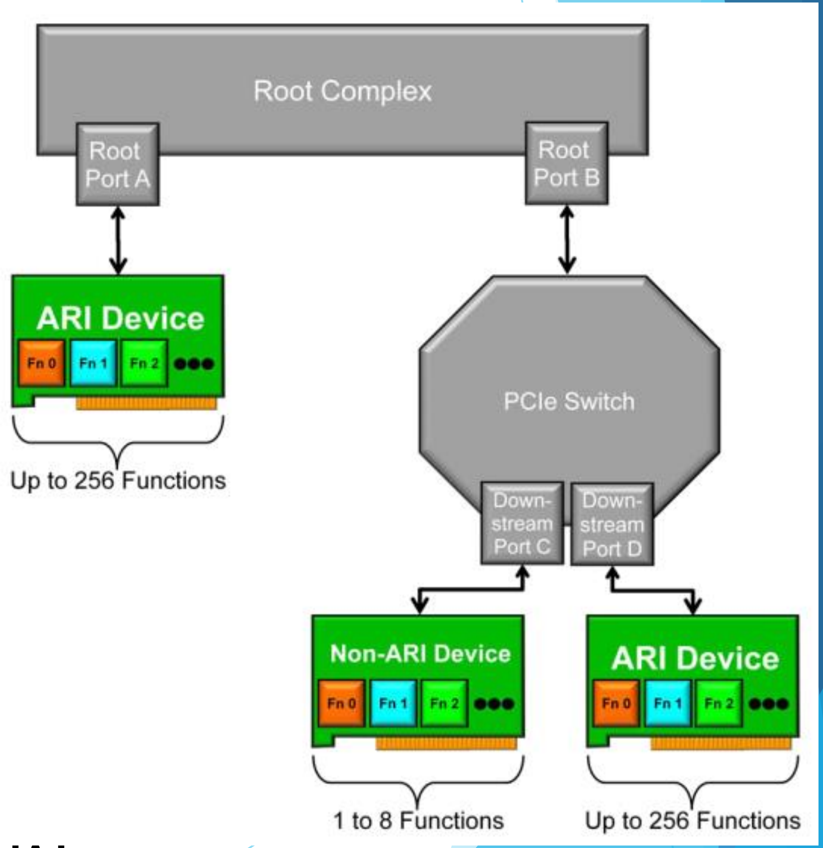


Figure from Intel PCI-SIG SR_IOV prim

ATS – Address Translation Services

- ATS provides a mechanism allowing a virtual machine to perform **DMA transaction directly to and from a PCIe endpoint.**

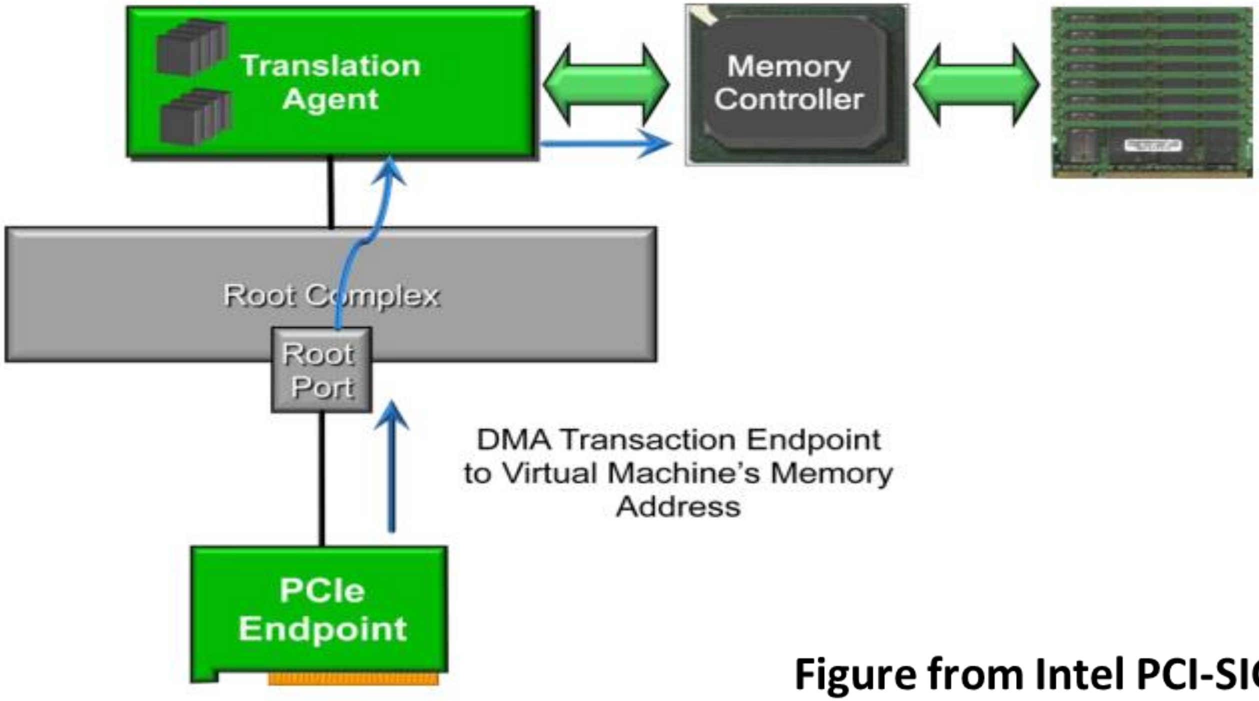
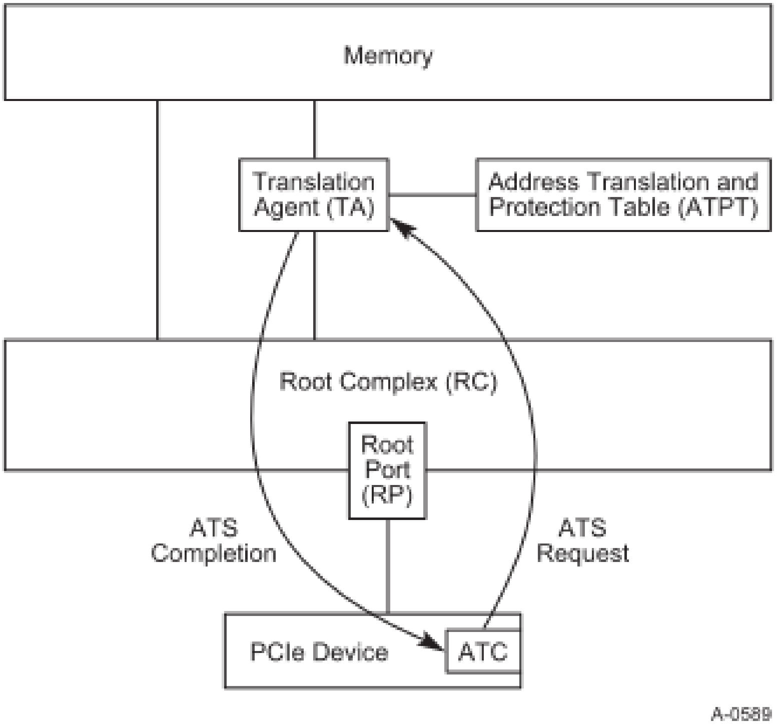


Figure from Intel PCI-SIG SR_IOV prim

ATS – Address Translation Services

- ATS uses a **request-completion** protocol between a Device and a Root Complex (RC)



ATS – Address Translation Services

- When the Function receives the ATS Translation Completion
  - Either updates its ATC to reflect the translation
  - Or notes that a translation does not exist.

- The Function generates subsequent requests using
  - Either a translated address
  - Or an un-translated address based on the results of the Completion.

“Vanilla” PCIe Device

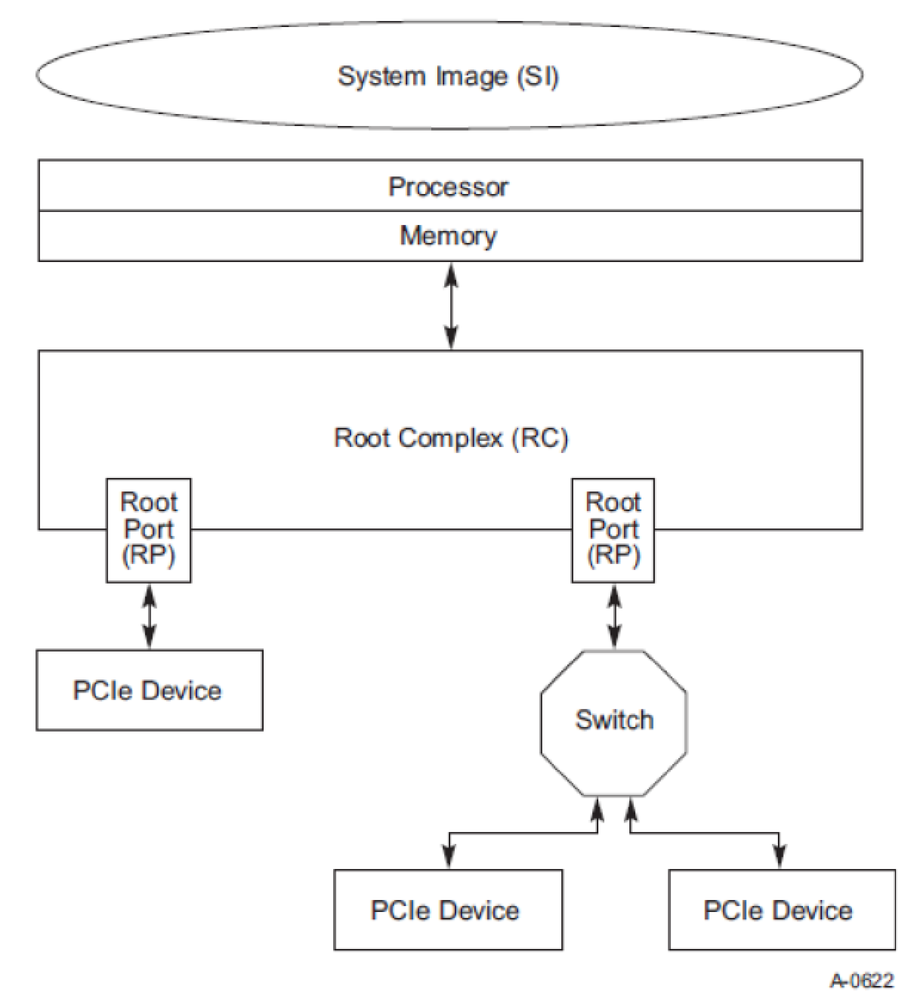


Figure 9-1: Generic Platform Configuration

PCI Express Fabric with PCIe Express Physical Functions

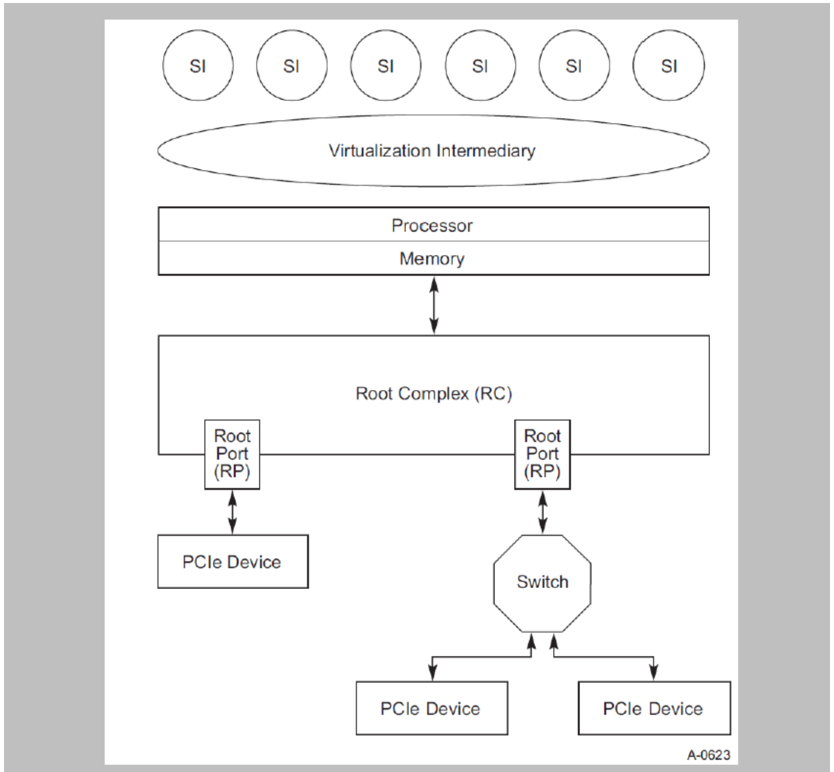


Figure 9-2: Generic Platform Configuration with a VI and Multiple SI

PCI Express with SR-IOV enabled

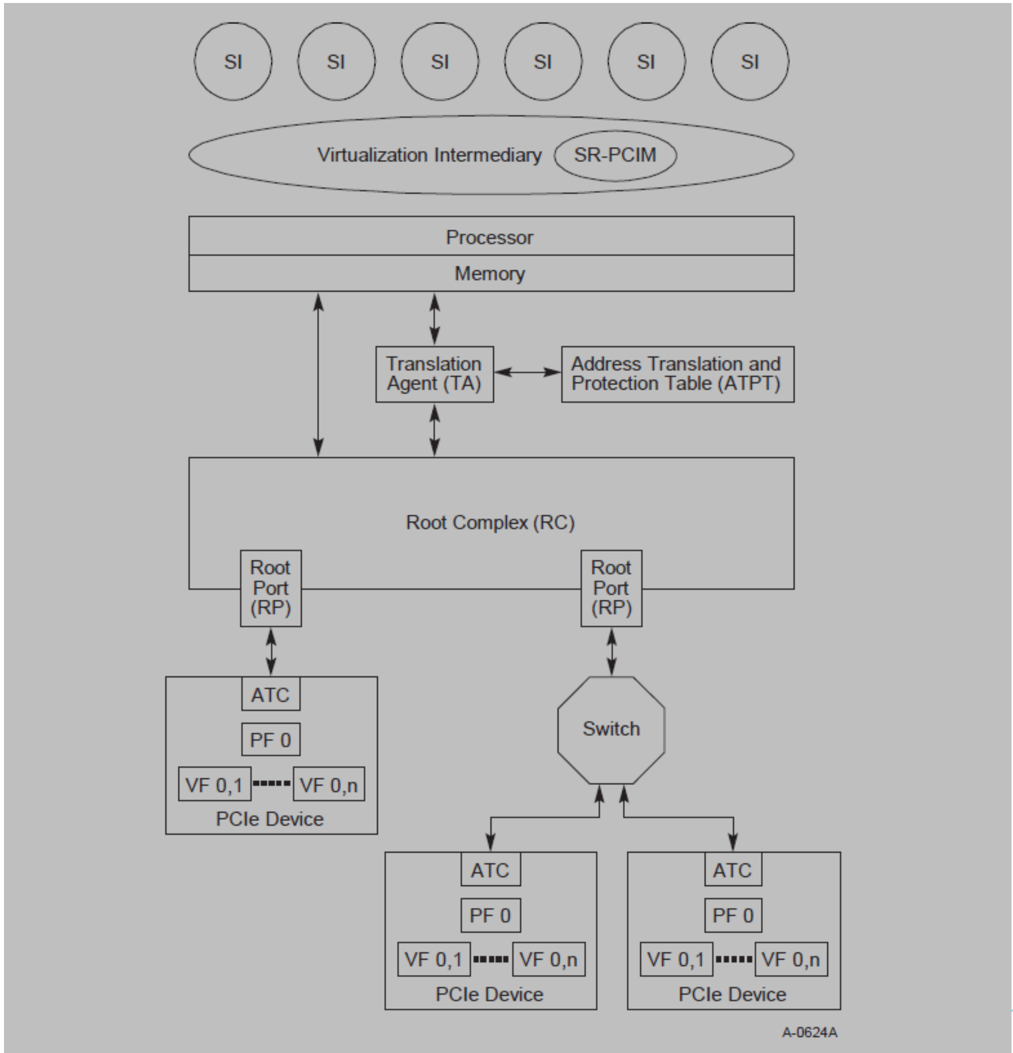


Figure 9-3: Generic Platform Configuration with SR-IOV and IOV Enablers

PCI Express device with Multiple physical functions

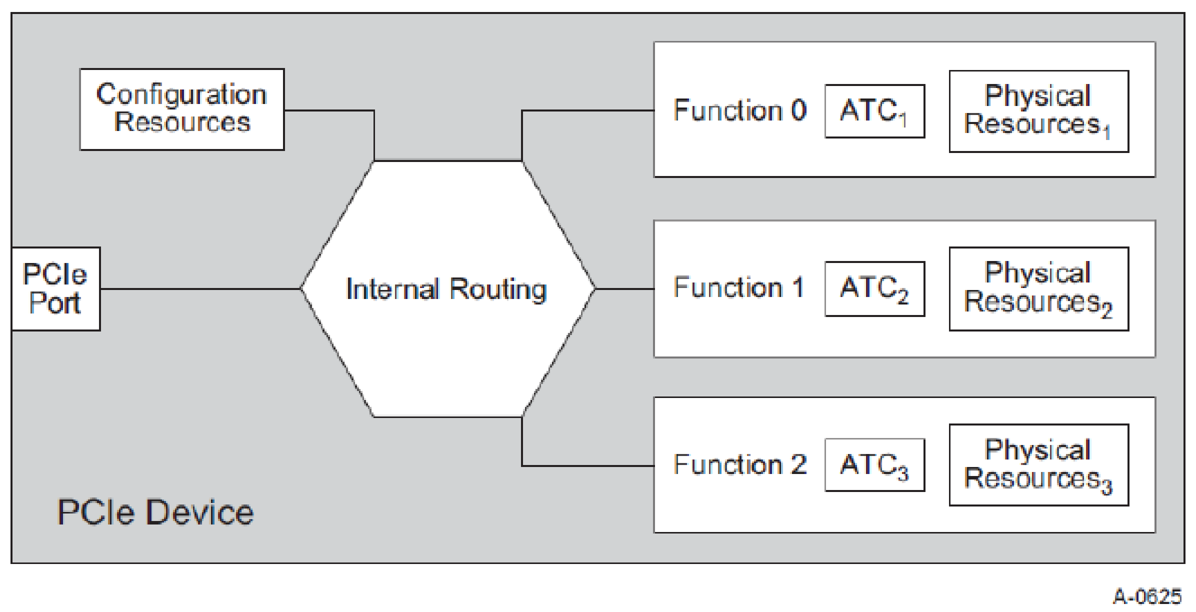


Figure 9-4: Example Multi-Function Device

PCI Express device, SR-IOV capable with single PF (our focus)

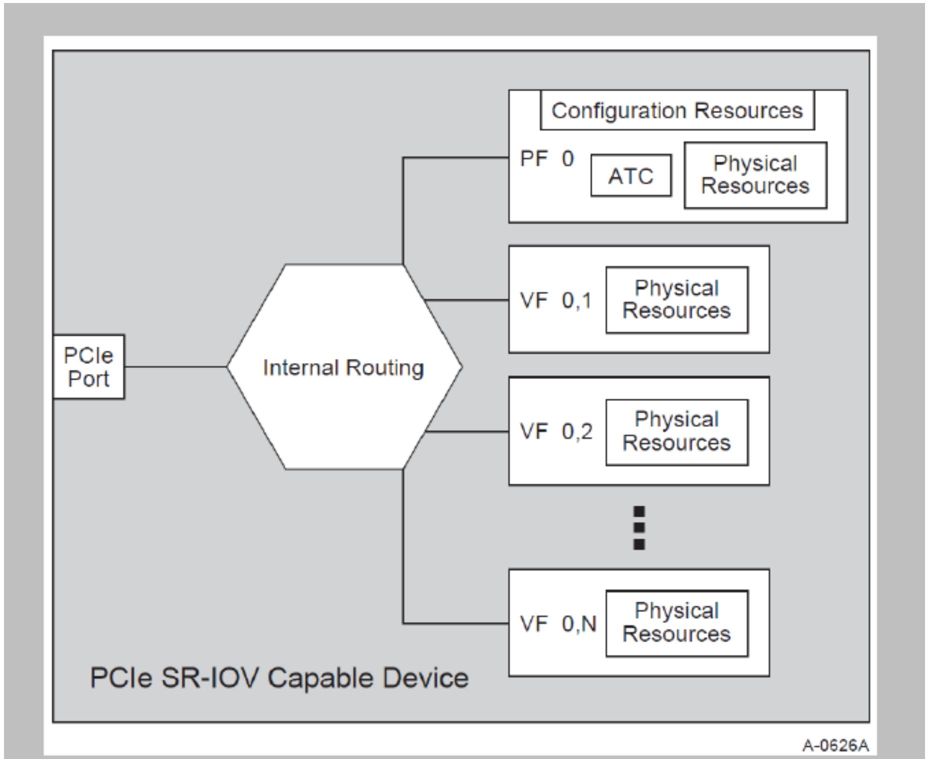


Figure 9-5: Example SR-IOV Single PF Capable Device

PCI Express device, SR-IOV capable with Multiple PFs

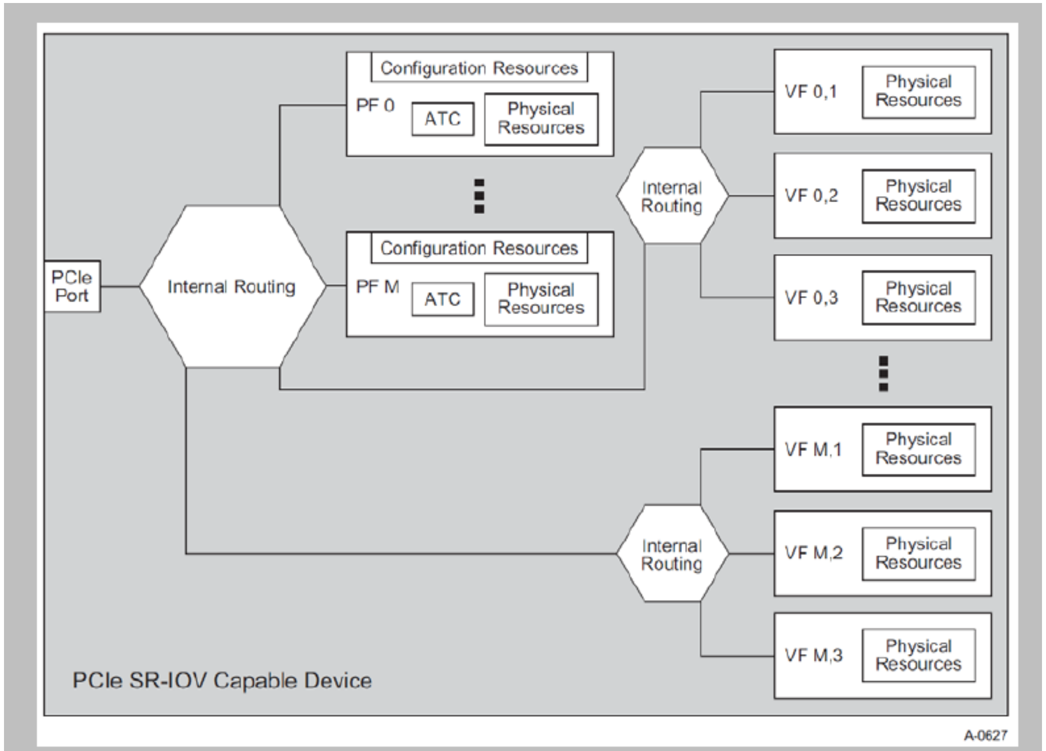
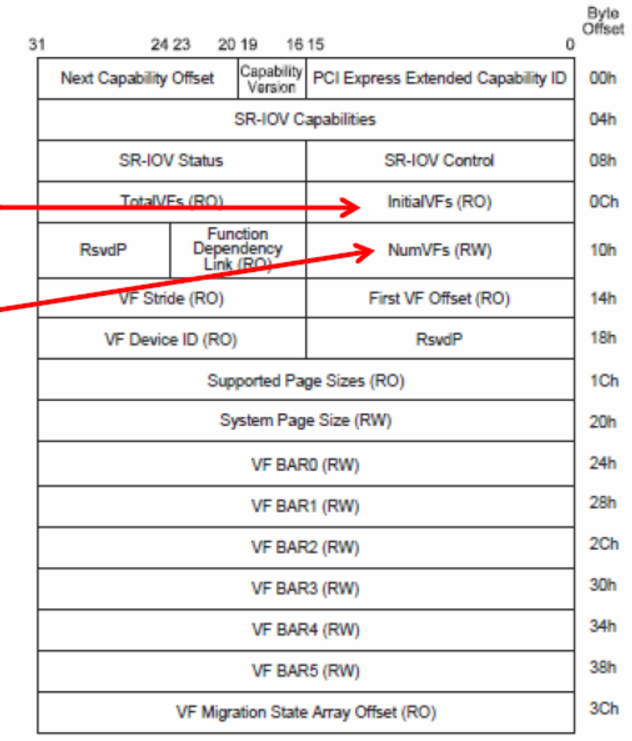


Figure 9-6: Example SR-IOV Multi-PF Capable Device

SR-IOV Configuration Space Mapping

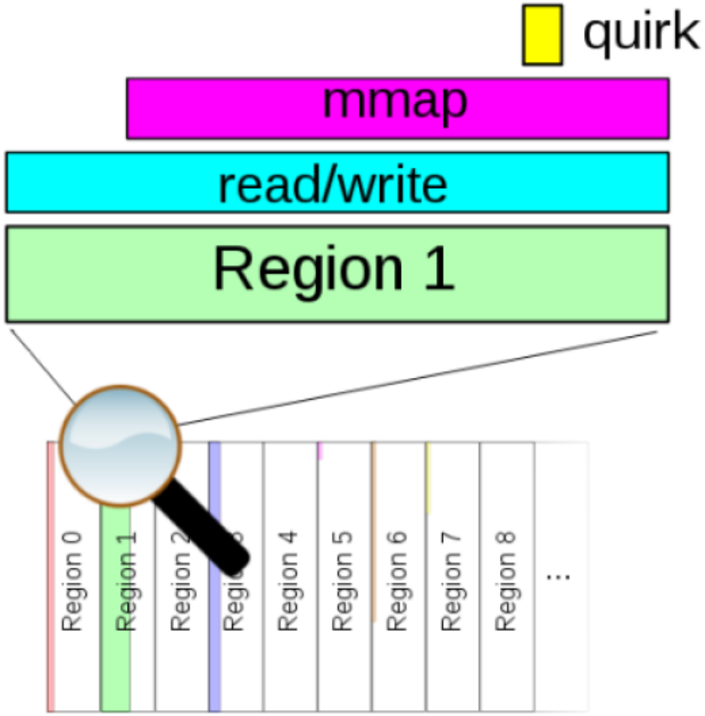
- ❐ Determine desired number of Virtual Functions from *InitialVFs* field
- ❐ Program *NumVFs* field to match

- ❐ Multi-Root adds a further layer where configuration software ***first*** allocates VFs to Virtual Hierarchies – thus *InitialVFs* may be less than *TotalVFs*



31 – 24	23 – 20	19 – 16	15 – 0	Byte Offset
Next Capability Offset		Capability Version	PCI Express Extended Capability ID	00h
SR-IOV Capabilities				04h
SR-IOV Status			SR-IOV Control	08h
TotalVFs (RO)			InitialVFs (RO)	0Ch
RsvdP	Function Dependency Link (RO)		NumVFs (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

The device implements SR-IOV capabilities using **vfio**

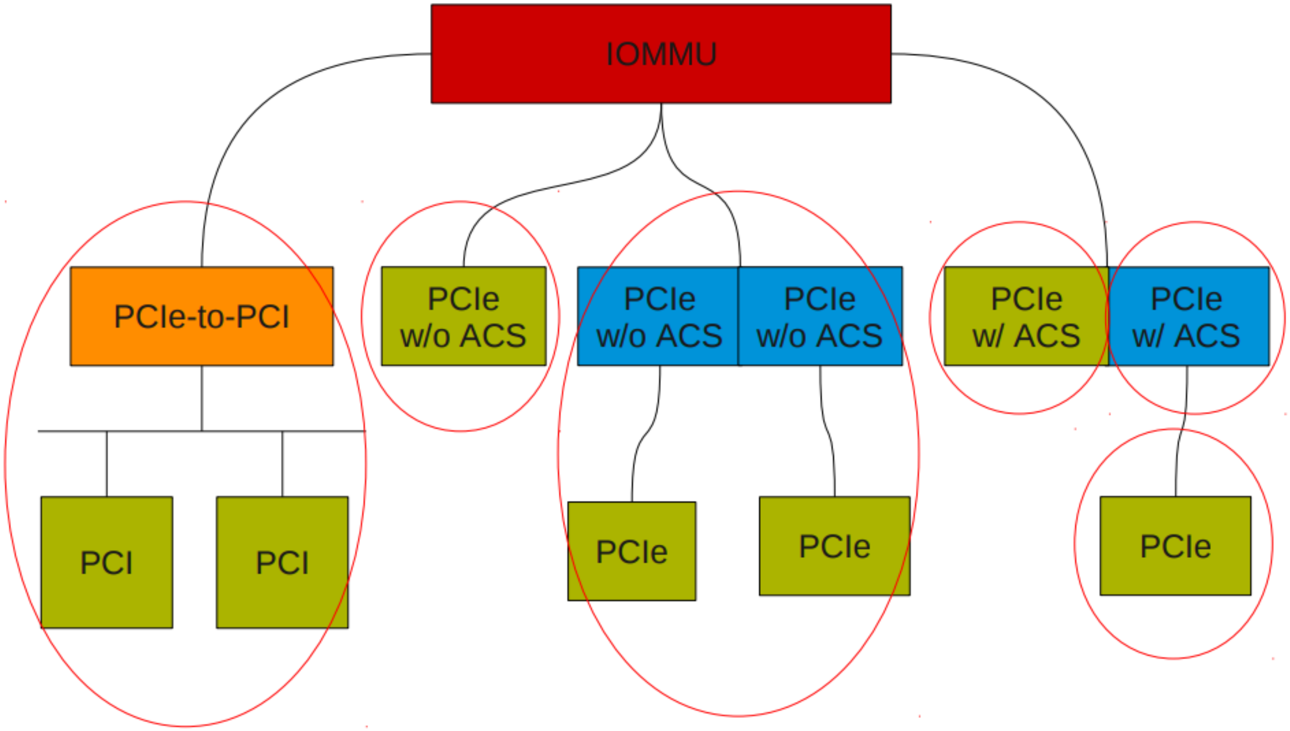


VFIO – User-space Virtual Function I/O

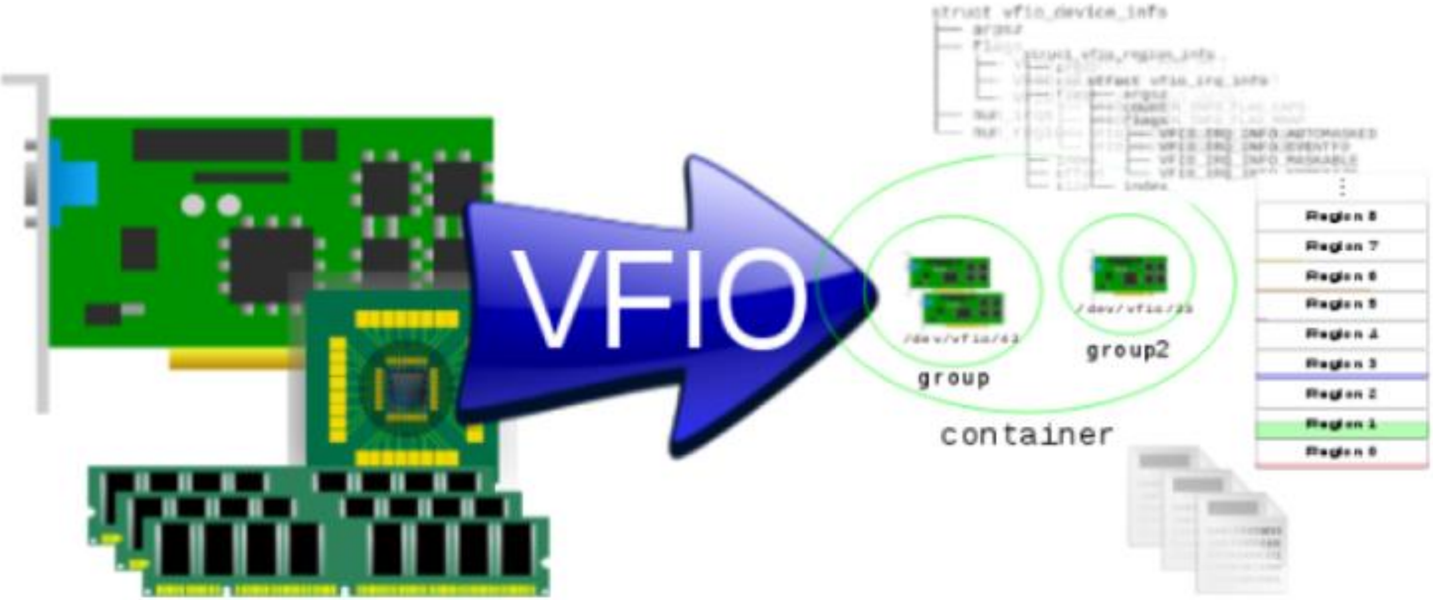
- A new user level driver framework for Linux

- ● Originally developed by Tom Lyon (Cisco)
- ● IOMMU-based DMA and interrupt isolation
- ● Full devices access (MMIO, I/O port, PCI config)
- ● Efficient interrupt mechanisms
- ● Modular IOMMU and device backends

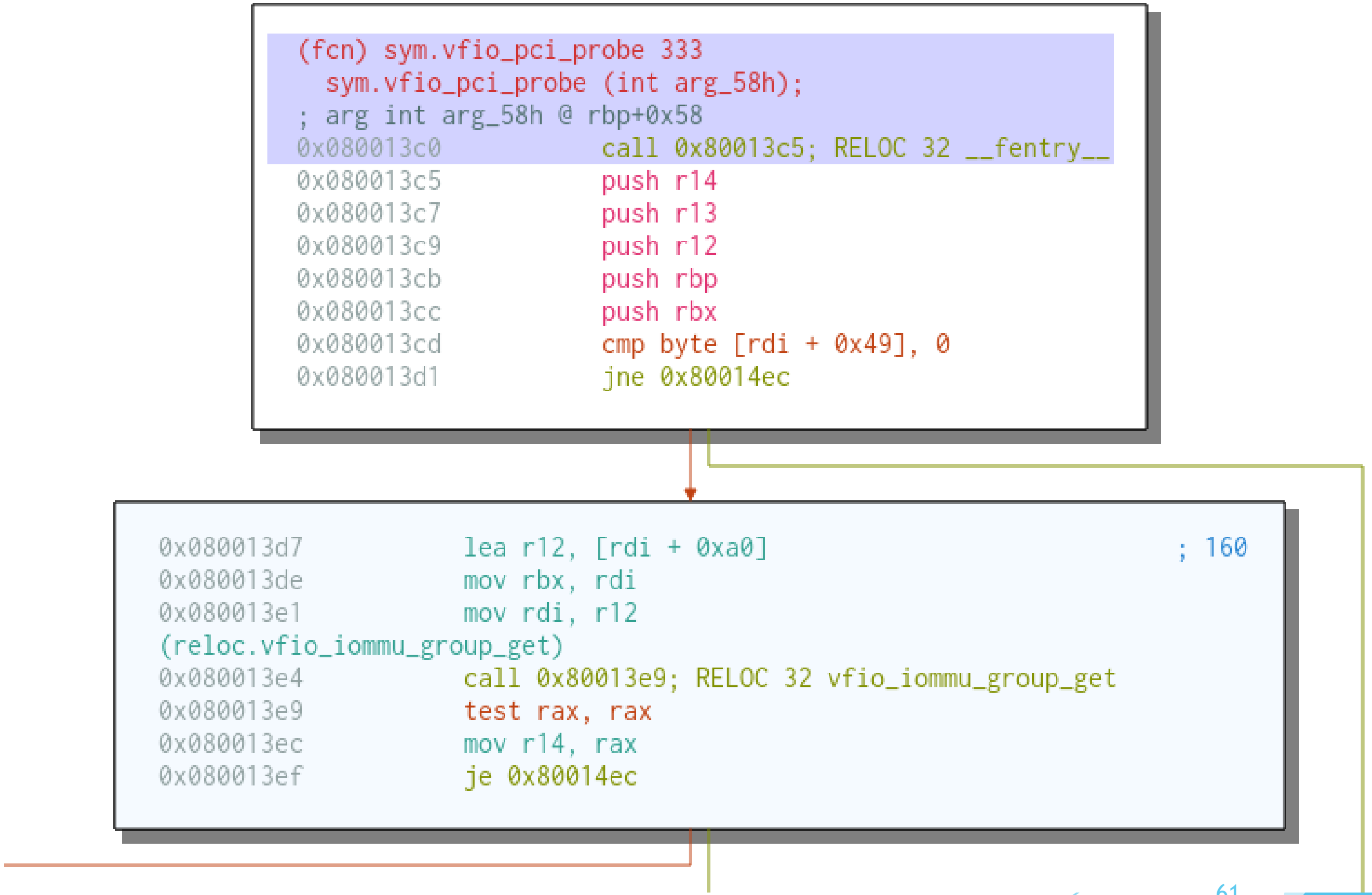
IOMMU is in charge of “isolation GROUPS”



VFIO in a nutshell



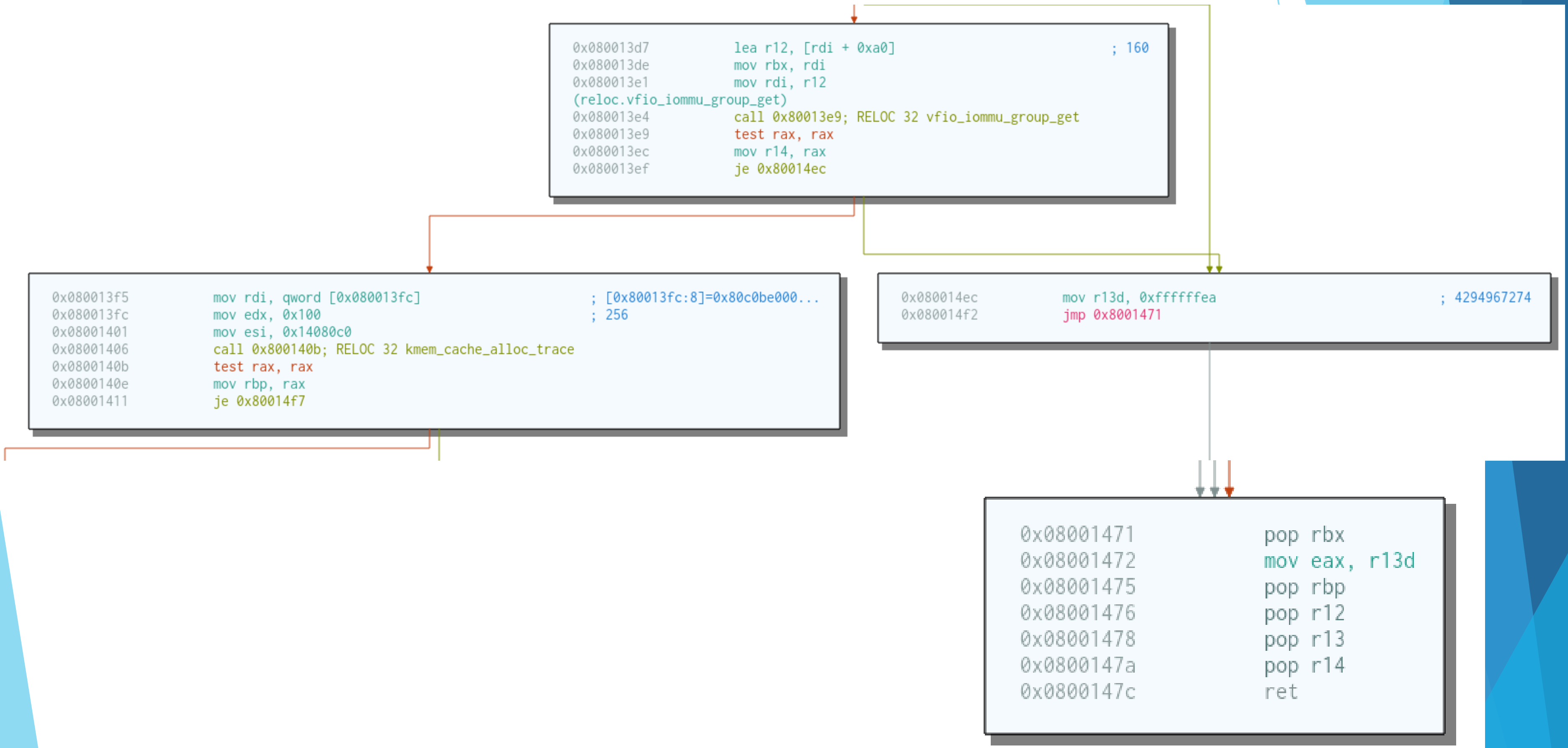
Reverse engineering SR-IOV process – Probe VFs of a PF



```
  (fcn) sym.vfio_pci_probe 333
    sym.vfio_pci_probe (int arg_58h);
  ; arg int arg_58h @ rbp+0x58
  0x080013c0          call 0x80013c5; RELOC 32 __fentry__
  0x080013c5          push r14
  0x080013c7          push r13
  0x080013c9          push r12
  0x080013cb          push rbp
  0x080013cc          push rbx
  0x080013cd          cmp byte [rdi + 0x49], 0
  0x080013d1          jne 0x80014ec
```

```
  0x080013d7          lea r12, [rdi + 0xa0]                          ; 160
  0x080013de          mov rbx, rdi
  0x080013e1          mov rdi, r12
  (reloc.vfio_iommu_group_get)
  0x080013e4          call 0x80013e9; RELOC 32 vfio_iommu_group_get
  0x080013e9          test rax, rax
  0x080013ec          mov r14, rax
  0x080013ef          je 0x80014ec
```

If the device includes SR-IOV capability, trace functions



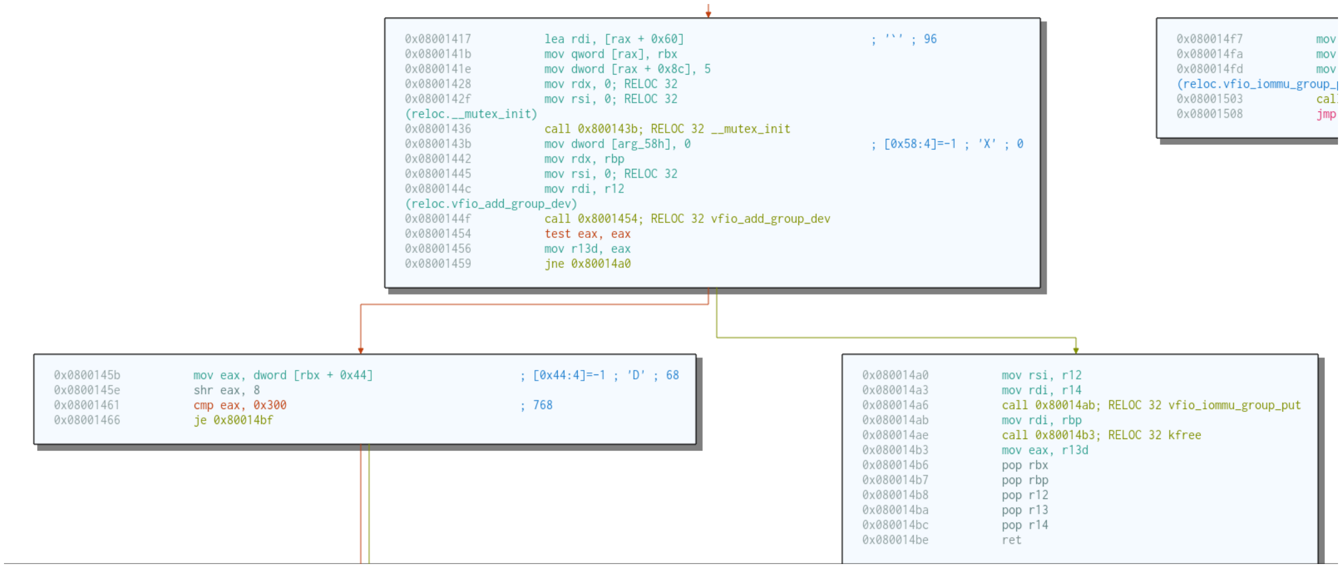
```
0x080013d7          lea r12, [rdi + 0xa0]                                    ; 160
0x080013de          mov rbx, rdi
0x080013e1          mov rdi, r12
(reloc.vfio_iommu_group_get)
0x080013e4          call 0x80013e9; RELOC 32 vfio_iommu_group_get
0x080013e9          test rax, rax
0x080013ec          mov r14, rax
0x080013ef          je 0x80014ec
```

```
0x080013f5          mov rdi, qword [0x080013fc]                   ; [0x80013fc:8]=0x80c0be000...
0x080013fc          mov edx, 0x100                                ; 256
0x08001401          mov esi, 0x14080c0
0x08001406          call 0x800140b; RELOC 32 kmem_cache_alloc_trace
0x0800140b          test rax, rax
0x0800140e          mov rbp, rax
0x08001411          je 0x80014f7
```

```
0x080014ec          mov r13d, 0xffffffea                          ; 4294967274
0x080014f2          jmp 0x8001471
```

```
0x08001471          pop rbx
0x08001472          mov eax, r13d
0x08001475          pop rbp
0x08001476          pop r12
0x08001478          pop r13
0x0800147a          pop r14
0x0800147c          ret
```

If we find a VF, add it to the group (according to IOMMU)



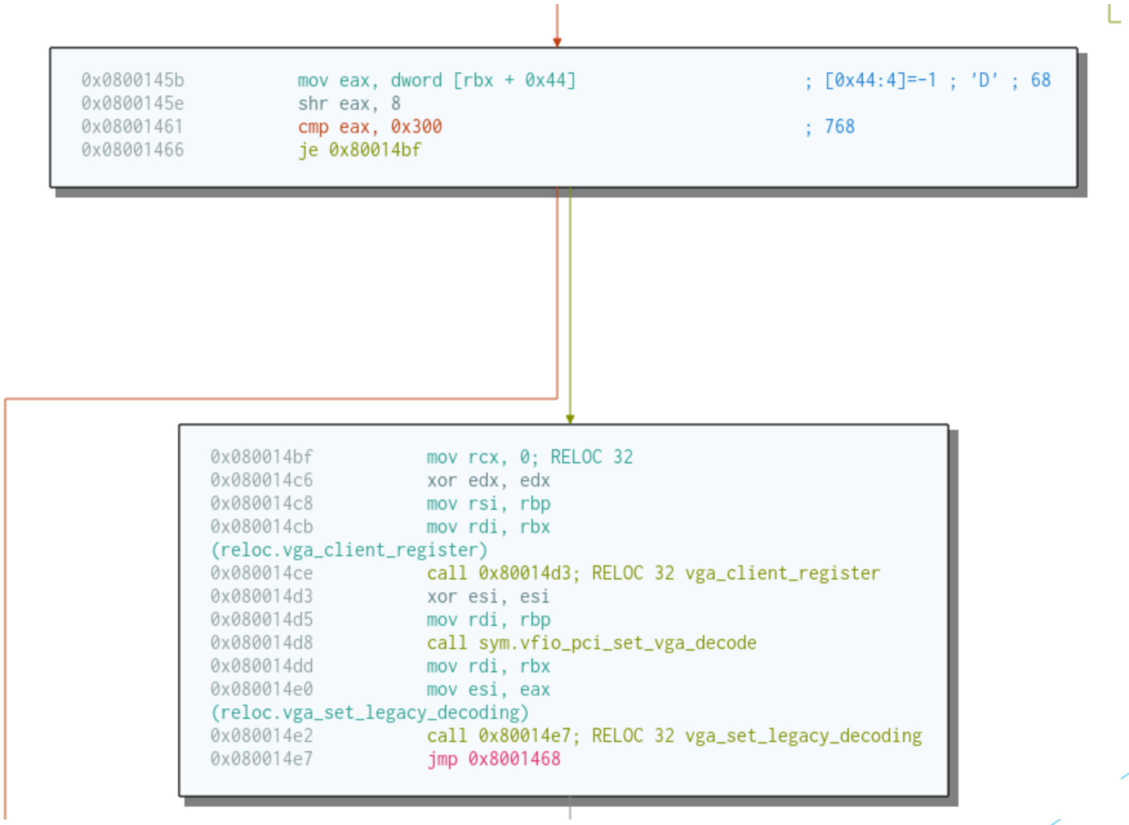
```
0x08001417          lea rdi, [rax + 0x60]                      ; '`' ; 96
0x0800141b          mov qword [rax], rbx
0x0800141e          mov dword [rax + 0x8c], 5
0x08001428          mov rdx, 0; RELOC 32
0x0800142f          mov rsi, 0; RELOC 32
(reloc.__mutex_init)
0x08001436          call 0x800143b; RELOC 32 __mutex_init
0x0800143b          mov dword [arg_58h], 0                     ; [0x58:4]=-1 ; 'X' ; 0
0x08001442          mov rdx, rbp
0x08001445          mov rsi, 0; RELOC 32
0x0800144c          mov rdi, r12
(reloc.vfio_add_group_dev)
0x0800144f          call 0x8001454; RELOC 32 vfio_add_group_dev
0x08001454          test eax, eax
0x08001456          mov r13d, eax
0x08001459          jne 0x80014a0
```

```
0x080014f7          mov
0x080014fa          mov
0x080014fd          mov
(reloc.vfio_iommu_group_
0x08001503          cal
0x08001508          jmp
```

```
0x0800145b          mov eax, dword [rbx + 0x44]                ; [0x44:4]=-1 ; 'D' ; 68
0x0800145e          shr eax, 8
0x08001461          cmp eax, 0x300                             ; 768
0x08001466          je 0x80014bf
```

```
0x080014a0          mov rsi, r12
0x080014a3          mov rdi, r14
0x080014a6          call 0x80014ab; RELOC 32 vfio_iommu_group_put
0x080014ab          mov rdi, rbp
0x080014ae          call 0x80014b3; RELOC 32 kfree
0x080014b3          mov eax, r13d
0x080014b6          pop rbx
0x080014b7          pop rbp
0x080014b8          pop r12
0x080014ba          pop r13
0x080014bc          pop r14
0x080014be          ret
```

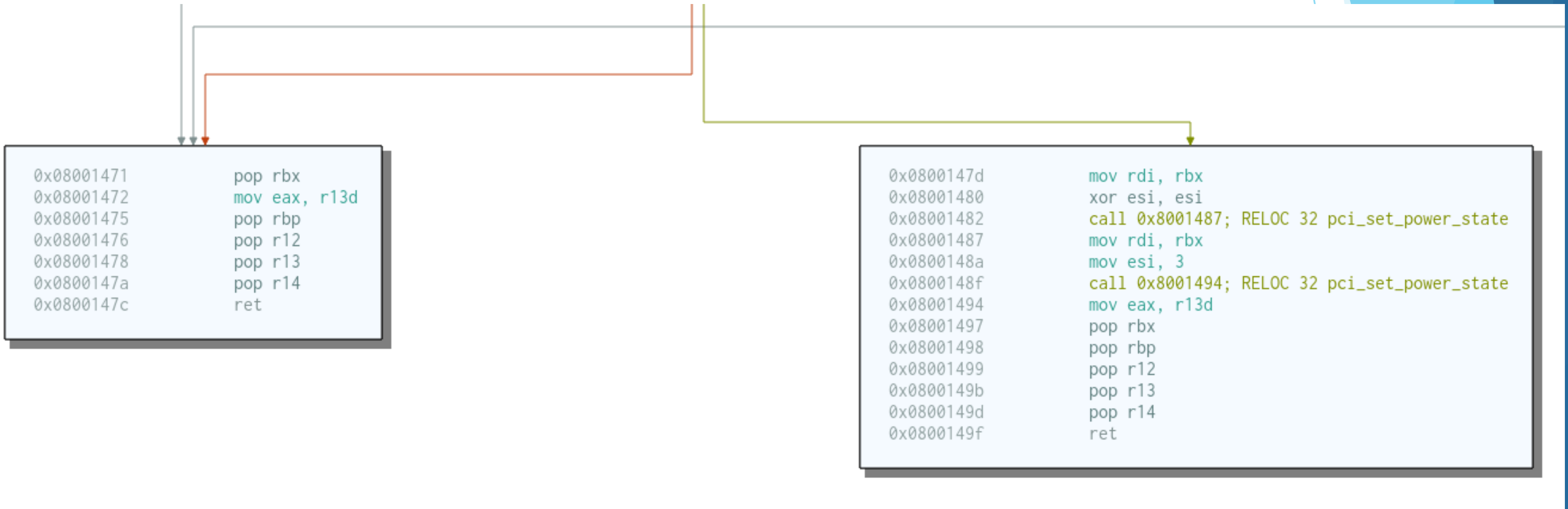
Check for special capabilities (VGA Passthrough, etc.)



```
0x0800145b          mov eax, dword [rbx + 0x44]                   ; [0x44:4]=-1 ; 'D' ; 68
0x0800145e          shr eax, 8
0x08001461          cmp eax, 0x300                                 ; 768
0x08001466          je 0x80014bf
```

```
0x080014bf          mov rcx, 0; RELOC 32
0x080014c6          xor edx, edx
0x080014c8          mov rsi, rbp
0x080014cb          mov rdi, rbx
(reloc.vga_client_register)
0x080014ce          call 0x80014d3; RELOC 32 vga_client_register
0x080014d3          xor esi, esi
0x080014d5          mov rdi, rbp
0x080014d8          call sym.vfio_pci_set_vga_decode
0x080014dd          mov rdi, rbx
0x080014e0          mov esi, eax
(reloc.vga_set_legacy_decoding)
0x080014e2          call 0x80014e7; RELOC 32 vga_set_legacy_decoding
0x080014e7          jmp 0x8001468
```

Set power state (another capability) and end

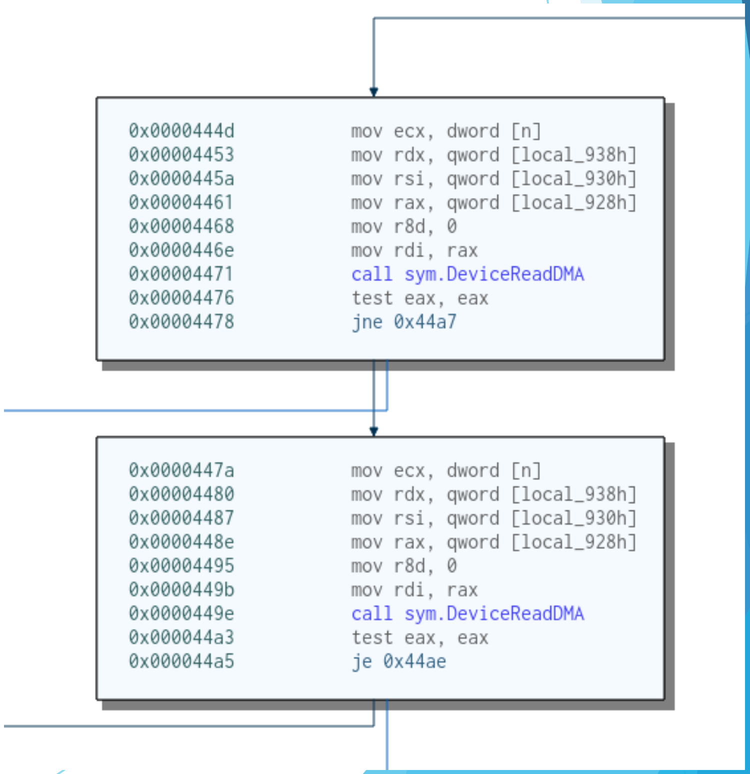


```
0x08001471          pop rbx
0x08001472          mov eax, r13d
0x08001475          pop rbp
0x08001476          pop r12
0x08001478          pop r13
0x0800147a          pop r14
0x0800147c          ret
```

```
0x0800147d          mov rdi, rbx
0x08001480          xor esi, esi
0x08001482          call 0x8001487; RELOC 32 pci_set_power_state
0x08001487          mov rdi, rbx
0x0800148a          mov esi, 3
0x0800148f          call 0x8001494; RELOC 32 pci_set_power_state
0x08001494          mov eax, r13d
0x08001497          pop rbx
0x08001498          pop rbp
0x08001499          pop r12
0x0800149b          pop r13
0x0800149d          pop r14
0x0800149f          ret
```

Profit

- Analyzing PCIe is easy using radare2 – **if** you know what you're looking for
- When you design your PCIe driver – always check that its capabilities are working properly (look for DMA and unknown mappings)



```
0x0000444d        mov ecx, dword [n]
0x00004453        mov rdx, qword [local_938h]
0x0000445a        mov rsi, qword [local_930h]
0x00004461        mov rax, qword [local_928h]
0x00004468        mov r8d, 0
0x0000446e        mov rdi, rax
0x00004471        call sym.DeviceReadDMA
0x00004476        test eax, eax
0x00004478        jne 0x44a7
```

```
0x0000447a        mov ecx, dword [n]
0x00004480        mov rdx, qword [local_938h]
0x00004487        mov rsi, qword [local_930h]
0x0000448e        mov rax, qword [local_928h]
0x00004495        mov r8d, 0
0x0000449b        mov rdi, rax
0x0000449e        call sym.DeviceReadDMA
0x000044a3        test eax, eax
0x000044a5        je 0x44ae
```

Sum-up - Example

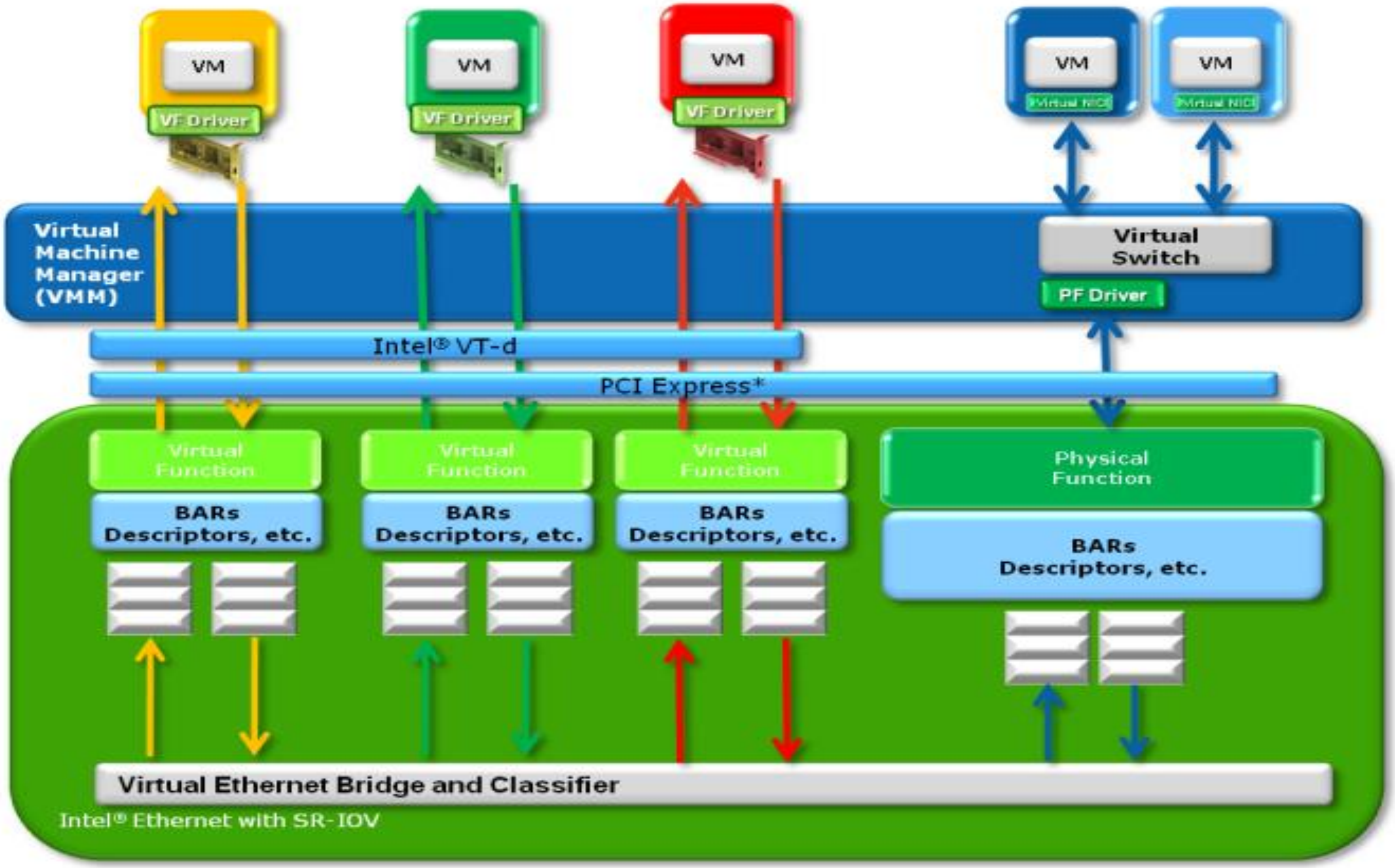
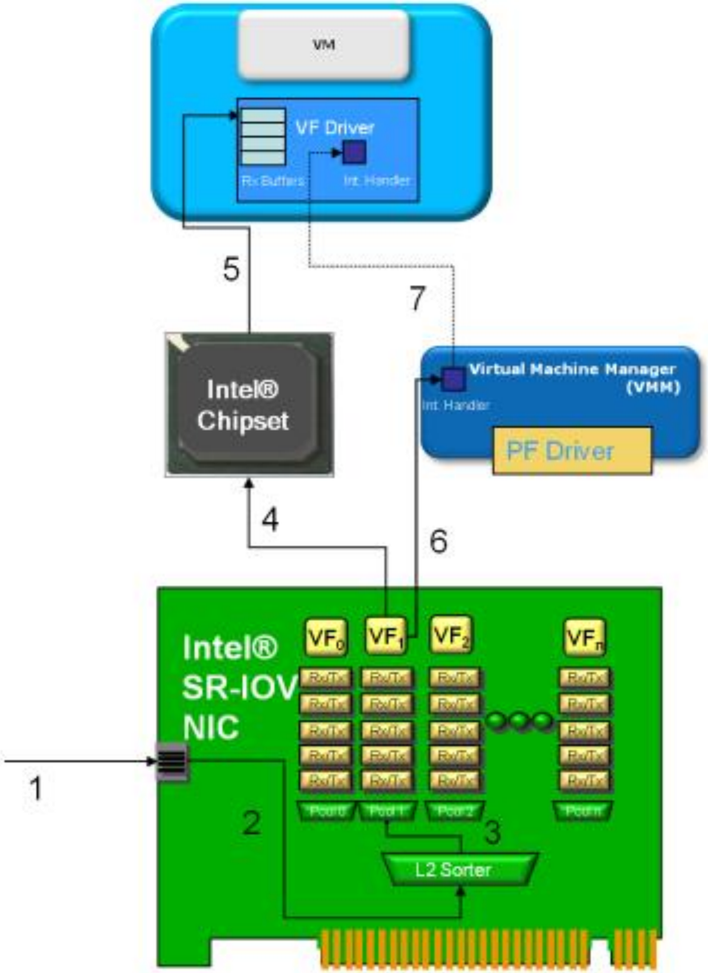


Figure from Inter PCI-SIG SR-IOV Primer

PCIe Data Path for incoming packets

1. The Ethernet packet arrives at the Ethernet NIC

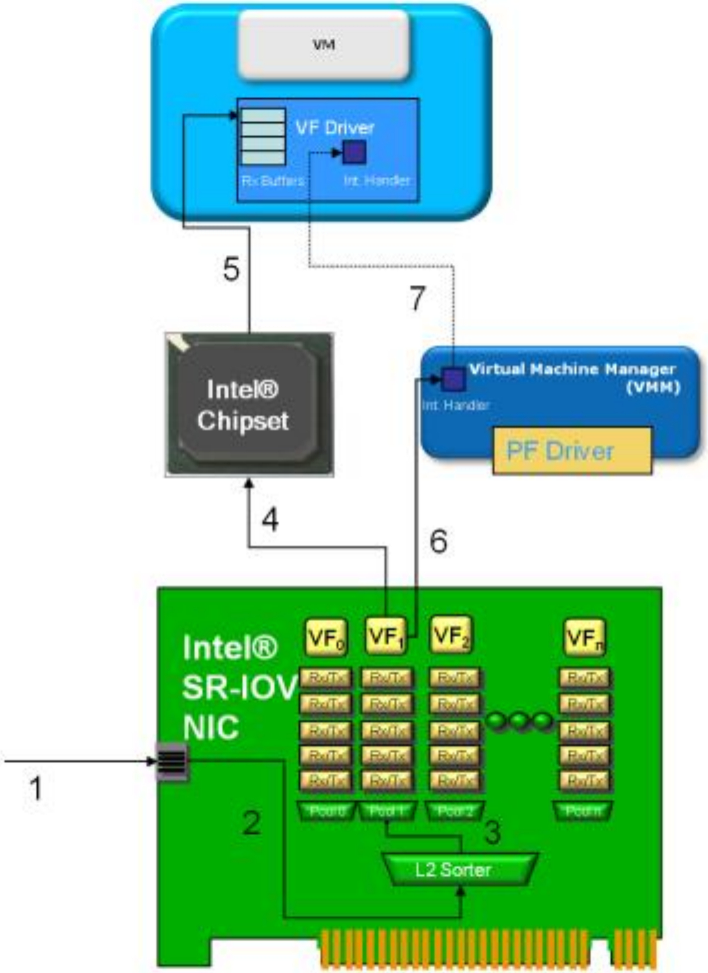
2. The packet is sent to the Layer 2 sorter/switch/classifier
   - This Layer 2 sorter is configured by the Master Driver. When either the MD or the VF Driver configure a **MAC address** or **VLAN**, this Layer 2 sorter is configured.



Data Path for incoming packets

3. After being sorted by the Layer 2 Switch, the packet is placed into a receive queue dedicated to the target VF.

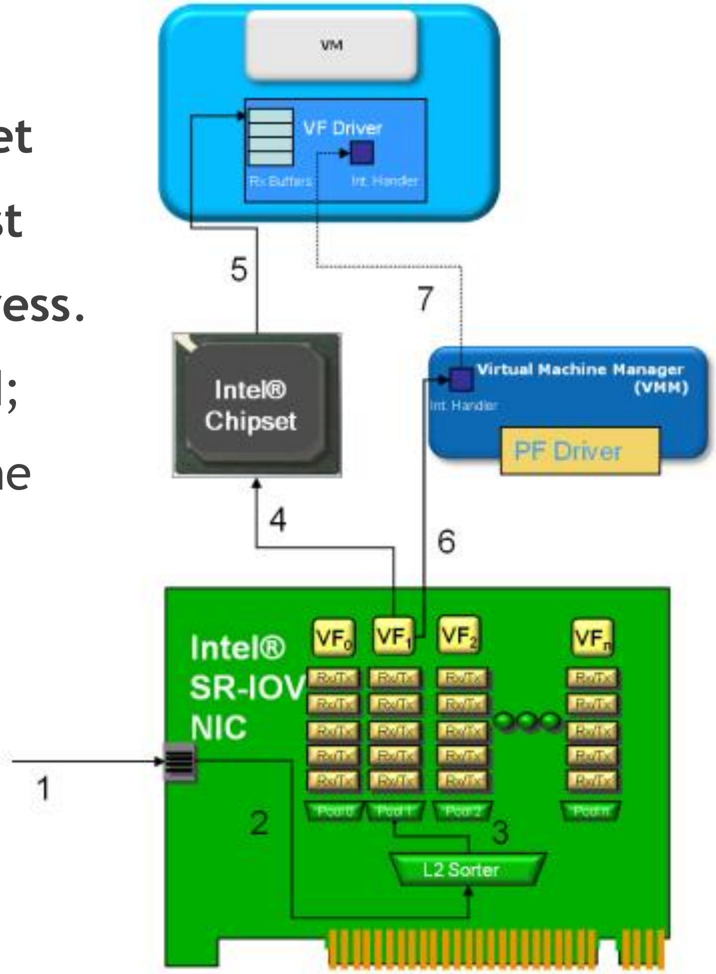
4. The DMA operation is initiated. The target memory address for the DMA operation is defined within the descriptors in the VF, which have been configured by the VF driver within the VM.



Data Path for incoming packets

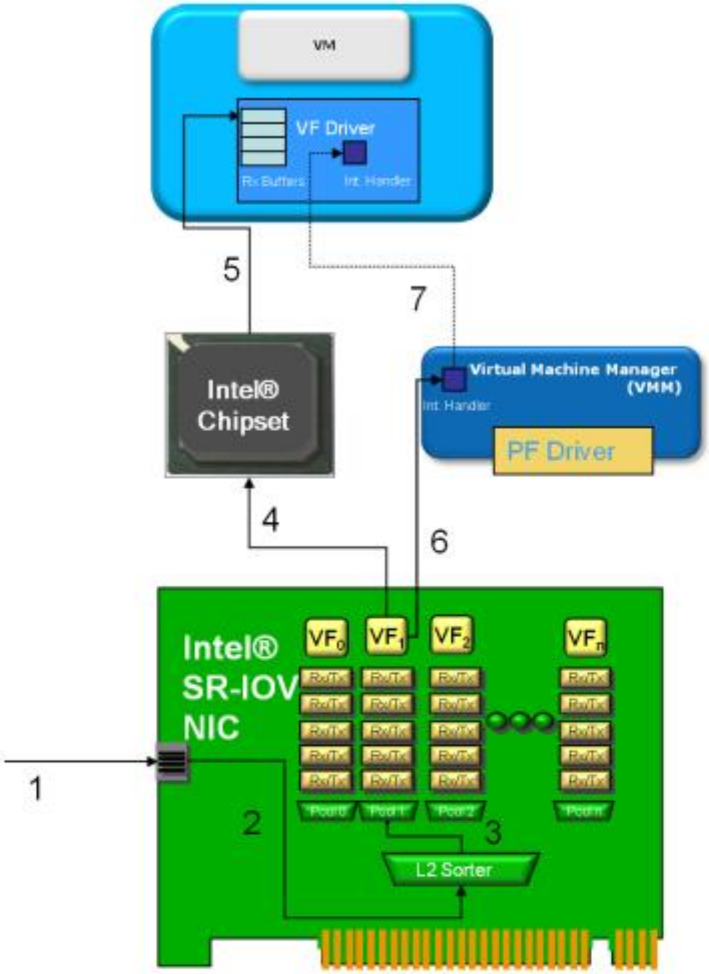
5. The DMA Operation has reached the chipset. Intel VT-d, which has been configured by the **VMM then remaps the target DMA address from a virtual host address to a physical host address.** The DMA operation is completed; the Ethernet packet is now in the memory space of the VM

6. The NIC **fires interrupt**, indicating a packet has arrived. This interrupt is handled by the VMM



Data Path for incoming packets

7. The VMM fires a virtual interrupt to the VM, so that it is informed that the packet has arrived



Now you can too!

- Find your favorite PCIe device
- Look for its device driver
- Check how the device is enumerated
- Look for its capabilities
- Follow the data flow and…
- Hack away!

Thanks!

- Adir Abraham
- Twitter: @adirab
- LinkedIn: https://linkedin.com/in/adirab

Reference

- PCI-SIG:
  - PCI Express Base Specification v4.0 Revision 1.0
  - Single Root I/O Virtualization and Sharing Specification Revision 1.1
  - Address Translation Services Revision 1.1

- Intel PCI-SIG SR-IOV Primer
- http://opensecuritytraining.info
- VFIO: A User’s perspective