# Enabling VirtIO Driver Support on FPGAs

Sahan Bandara[*] Ahmed Sanaullah[**] Zaid Tahir[*] Ulrich Drepper[**] Martin Herbordt[*]
[*]*CAAD Lab, ECE Department, Boston University* [**]*Red Hat Inc.*

***Abstract*—Host-FPGA connectivity is critical for enabling a vast number of FPGA use cases in data centers, edge, and IoT. This interface must be reliable, robust, and uniform, whilst supporting necessary protocols and functionality. However, existing support for host-FPGA connectivity has several drawbacks on both the host and the device. This includes a lack of portability and poor upstream support, both of which can make it difficult for CPUs to easily and effectively leverage FPGAs. Native VirtIO drivers in the host operating system can help address some of these limitations, especially on the host side, but implementing device-side support for the VirtIO specification is a challenge due to the substantial hardware complexity involved.**

**In this work, we present a framework for enabling FPGAs to interface native operating system VirtIO drivers on the host. To reduce the implementation overhead and improve portability, this framework uses both generic RTL blocks and modified, chip/device specific PCIe IP blocks. Moreover, this approach implements all the necessary data structures and functionality needed to meet the VirtIO specification requirements. We test the framework using the Xilinx DMA/Bridge Subsystem for PCI Express (XDMA) IP, implemented on an Alinx AX7A200 FPGA board (with a Xilinx XC7A200TFBG484-2 FPGA chip), and a host machine running the Fedora operating system. Our results show that the FPGA can be successfully enumerated as a VirtIO device, and interfaced using only native Linux VirtIO drivers.**

***Index Terms*—FPGA, PCIe, VirtIO**

## I. Introduction

The flexibility of FPGAs enables them to be important complexity offload devices for the CPU. They can be used to accelerate user and system applications, implement networking functions to process data at line rates, perform system administration, provide secure enclaves with hardware isolation, and a number of other tasks [1]–[8]. In order to support these vast number of use cases, the host-FPGA connectivity must implement a required set of features and protocols, as well as provide a reliable, robust and uniform interface.

Despite the critical nature of host-FPGA connectivity, there are a number of limitations of existing PCIe [9] interfaces that support this communication. On the device side, the major limitation is portability. The IP blocks used to implement host interfaces are typically vendor specific, both in terms of chip and board, and can have inconsistencies in the features they support and the APIs they expose to the user logic. This is due to the implementation complexity of the communication stack, differences in resource types/amounts across chips, and the virtually impossible task of building a completely generic hardware stack due to use of essential ASIC blocks organized in chip-specific IO Bank structures, e.g., SERDES units.

On the host side, the major limitations are portability and poor maintenance. From a portability perspective, the differences in capabilities/functionality of the device can lead to compatibility issues if the driver attempts to use nonexistent functionality. This is especially important in FPGAs given their flexibility and that there is no standard set of supported features. Moreover, similar to the hardware side, APIs exposed to user applications by vendor drivers can also be inconsistent. From a maintenance perspective, deprecated software dependencies, especially as the host kernel is updated, and a changing set of features supported by IP blocks, means that the device drivers need to be patched frequently. Unfortunately there is often poor upstream support: most of the work on drivers is done downstream, i.e., by developers modifying the drivers themselves.

VirtIO [10] is an industry standard for I/O virtualization and is one possible solution to the challenges posed by the use of vendor-provided device drivers. There is native support for VirtIO in the host operating systems, such as the Linux kernel, which means that no additional driver needs to be written/maintained, and APIs are relatively consistent. VirtIO also supports feature negotiation, i.e., device and driver can use feature bits to determine the subset of supported features to ensure compatibility. Moreover, there are additional benefits to virtualized environments, such as faster guest-device communication. Exposing the FPGA to the host as a VirtIO device can reduce data copies and latency through direct communication between the guest user space and host device driver.

While VirtIO can help address limitations of the host-device interfaces on the host side, there are two major challenges involved with implementing native VirtIO support on the device. First, the FPGA hardware stack must meet the VirtIO specification, which means that appropriate data structures and state machines must be implemented. This is in contrast to existing approaches to VirtIO support, which build a hardware stack that does not fully meet the VirtIO specification and thus require custom VirtIO drivers to interface with it; these have the disadvantages similar to the typical FPGA drivers discussed above. The second challenge is that the hardware side is still chip/device specific and existing IP blocks may not support all required functionality. Building up the entire hardware stack from scratch is also not feasible due to the complexity of implementing the required IP blocks.

In this work, we address the above challenges by developing a framework that allows VirtIO support to be added to the FPGA hardware stack by: i) building a subset of the required hardware blocks from scratch using generic RTL, and ii) leveraging existing PCIe IP blocks for chip/device specific parts of the implementation. To achieve this, we first identify the interface that an FPGA should expose to the host to meet the VirtIO specification. Next, we instantiate the vendor IP by specifying appropriate parameter values based on the interface requirements. Then, we modify the IP RTL to add/modify critical functionality that was either not available in the IP, or was not exposed to the developer when instantiating the IP. Finally, we add a VirtIO controller block between the IP and user logic; this controller is responsible for implementing additional requirements of the VirtIO specification, such as data structures, arbitration logic, queue support and other state machines. We test our methodology by implementing a VirtIO console device using the Xilinx DMA/Bridge Subsystem for PCI Express IP on the Alinx AX7A200 FPGA board (with a Xilinx XC7A200TFBG484-2 FPGA chip), and a host machine running the Fedora 35 operating system. Xilinx Vivado tool flow is used for this implementation.

The specific contributions of this work are:

- Enabling FPGAs to leverage native VirtIO drivers in the host operating system for host-device communication;
- Identifying and implementing the device requirements for VirtIO, such as data structures, arbitration logic, queue support and other state machines;
- Improving portability and reducing implementation complexity of the hardware stack by building required hardware blocks using both generic RTL, and modifying existing PCIe IP blocks to implement chip/device specific blocks; and
- Demonstrating the effectiveness of our approach by modifying the Xilinx XDMA IP to enable enumeration and communication as a VirtIO console device in Fedora 35.

The rest of the paper is organized as follows. Section II gives the background of our work, as well as relevant existing efforts. Section III presents our framework for adding VirtIO support to PCIe based FPGAs. Section IV gives the results of our implementation of a VirtIO interface between a Xilinx FPGA and a Fedora operating system.

## II. Background

### *A. Typical Use Model for PCIe FPGAs*

Most high-end FPGAs, and even some low-cost FPGAs such as [11], currently support PCIe connectivity. This makes PCIe a popular host-device communication mechanism for FPGAs because of high data rates, etc. The host-FPGA communication is typically carried out with the assistance of a device driver either provided by the FPGA vendor or written by the user. In either case, the driver is specific to the given device. This is unavoidable because of different capabilities and PCIe IP core availability across different FPGA families. This results in the user having to maintain different drivers for different device families, and even for the same device, depending on the PCIe IPs used. There are opensource PCIe IPs and drivers provided by third parties such as [12], and [13]. These have similar limitations such as being specific to a device or possibly a particular board, and the user having to rely on a third party to maintain the drivers.

More recently, FPGA vendors have been providing runtime libraries, such as Xilinx runtime library (XRT) [14] and Open Programmable Acceleration Engine (OPAE) [15], which provide the user with simple APIs for programming, data movement, and controlling the FPGA. These typically provide a user space library, kernel space device drivers, and an FPGA shell that takes up a fixed portion of the FPGA fabric to implement the communication infrastructure such as PCIe and Ethernet controllers, DMA engines, memory interfaces, and any other peripherals. The kernel drivers are designed to match the IP blocks used to build this shell. The user application kernel is typically instantiated inside the shell by using partial reconfiguration. Some of these runtime libraries are even made open source. However, the drivers themselves are still device-specific and these frameworks do not support all the FPGA devices, even from the same vendor.

There exists the requirement for a more general use-model for communication with FPGAs over PCIe. Ideally, the drivers should be agnostic of the device being used, and the user should not have to maintain the device drivers by updating them whenever the kernel is updated. A possible solution that satisfies these requirements is use of the VirtIO drivers that are part of the Linux standard release.

### *B. VirtIO*

VirtIO devices are virtual devices found in virtual environments, yet, by design, they look like physical devices to a guest within a virtual machine [10]. VirtIO devices can use normal bus mechanisms for device discovery, interrupts, and DMA. VirtIO drivers follow suit. This gives the opportunity to use VirtIO drivers to communicate with physical devices as well. The device in question only needs to present a VirtIO compliant interface to the driver and the driver is agnostic to the fact that it is communicating with a physical device.

*1) VirtIO use cases:* The VirtIO architecture consists of three key components; front-end drivers, back-end devices, and the queues used for all communication between the front- and back-end components called ‘virtqueues’. VirtIO drivers are the front-end component used by guest applications running in a virtual machine. These communicate with the back-end VirtIO devices that are emulated by the host machine. There is a multitude of different combinations for the placement of these components in guest and host user versus kernel space. A detailed description is available in [16].

An interesting use-case is the para-virtualization in which there is a physical device attached to the host machine and access to it is virtualized by the VirtIO layer. Figure 1 depicts an abstract view of this use model. The requests from the VirtIO back-end device are sent to the physical device via the legacy device driver running in the host kernel space, and possibly a bridge device which converts the transactions to a format understood by the legacy driver.

If the physical device exposes a VirtIO compliant interface however, the intermediate layers can be bypassed to improve performance. In this scenario, a VirtIO driver running on the guest kernel space and one running in host kernel space see the same interface to the physical device. Figure 2 illustrates two use models that such a VirtIO compliant physical device will enable. In Figure 2(A), the user space application is interacting with the VirtIO driver running in the guest kernel space. The physical PCIe device is exposed to the guest VM with passthrough, which allows the VirtIO driver to directly communicate with the physical device. In Figure 2(B), the VirtIO driver is running in the host kernel space and communicating with the physical device. The guest application directly accesses the host VirtIO driver when it needs to use the physical device.

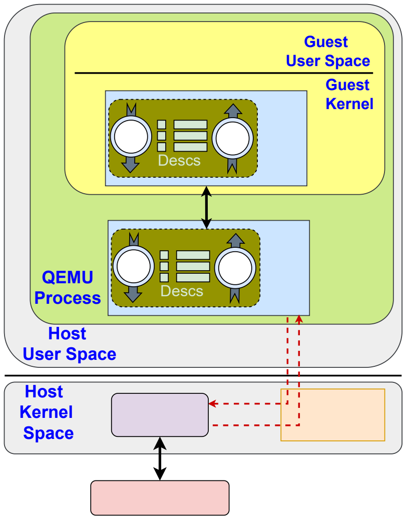


Fig. 1. VirtIO para-virtualization.

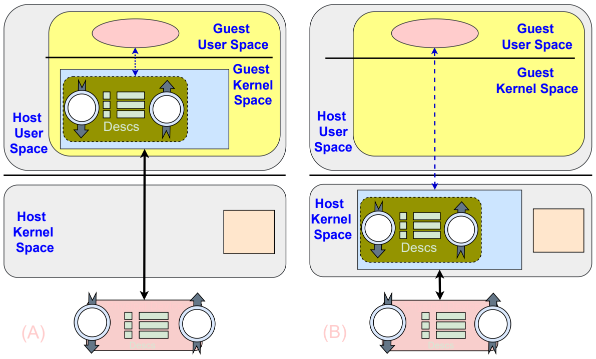


Fig. 2. Physical VirtIO device use models.

### C. Related Work

To the best of our knowledge, there is no published work on enabling VirtIO PCIe support on FPGAs. However, there is an FPGA IPU NIC based on an Intel Stratix 10 FPGA which supports virtio-net and virtio-blk over PCIe [17]. According to the product specifications, this can work as a VirtIO network or storage accelerator with stock VirtIO drivers. No information is available on whether the implementation is based on custom IP cores or vendor provided ones. Hardware support for VirtIO is implemented on a FPGA PCIe endpoint in [18]. However, this is achieved by designing a custom IP core.

VirtIO is used as the front-end driver in [19], where the authors focus on deploying multi-tenant FPGAs in Linux-based cloud infrastructure. While this work relies on VirtIO to decrease communication latency between software and hardware domains, the VirtIO drivers are not directly communicating with the FPGAs. At the host level, a legacy driver is still used to communicate with the FPGA. Other work focused on FPGA virtualization, such as [20] use a similar configuration where the VirtIO drivers only act as the front-end drivers. What we can infer from these studies is that VirtIO provides a good front-end for FPGA virtualization and multi-tenancy on cloud environments. By enabling the VirtIO drivers to directly communicate with the FPGAs, we can potentially improve the efficiency of the above schemes by eliminating multiple data copy operations among the intermediate layers in the communication stack.

## III. Methodology

To enable VirtIO drivers to communicate with an FPGA over PCIe, the FPGA should expose an interface that meets the requirements of the the VirtIO specification [10]. These requirements can be divided into three different operating phases: device identification, device initialization, and data movement. The methods used to implement a VirtIO compliant interface are described below in relation to the Xilinx FPGA and IP cores used in our implementation. The same methods are applicable to most Xilinx IP cores. We believe similar capabilities are available in FPGA devices and IPs from other vendors as well, but have not yet verified this fact.

### A. Device Identification

A VirtIO enabled FPGA should announce the correct vendor and device IDs at the time of PCI bus enumeration, at which the devices on the PCI bus get identified and initialized by the host. The PCI vendor ID should be `0x1AF4`. The PCI device ID is determined depending on the type of VirtIO device implemented on the FPGA. For instance, an FPGA-based smartNIC can use the device ID `0x1041` which is the device ID for a VirtIO network device. This is perhaps the simplest requirement in terms of the work necessary to achieve in an FPGA implementation. The IP generation tool flow generally allows the user to configure the PCI vendor and device IDs within the GUI when generating the PCIe IP core. Some Xilinx IPs provide the ability to change the PCI IDs at runtime as well. However, this is unnecessary as we only need the device to be recognized as a VirtIO device.

### B. Device Initialization

The VirtIO specification specifies five PCI capabilities that must be added to the PCI capability list for the device to be initialized and operate as a VirtIO device. These are: Common configuration, Notification, ISR status, Device specific configuration, and PCI configuration access. The first four of the above capabilities inform the VirtIO driver where to find a corresponding data structure that is used in initialization and regular operation of the device. The PCI configuration access capability provides the driver with an alternative access method to the data structures pointed by the other capabilities. For legacy VirtIO devices, the driver expects the data structures to be in the memory or IO region corresponding to base address register [9] (BAR) 0. However, for modern VirtIO devices, the data structure could be mapped to any of the BARs and the VirtIO capabilities in the capability list are essential to locating those.

Adding new capabilities to the PCI capability list turned out to be more challenging compared to setting the correct PCI vendor and device IDs. We used a Xilinx Artix-7 FPGA for our proof of concept implementation. This is one of the cheapest FPGAs with PCI capability. The reasoning behind the selection is that, if the capabilities provided by a cheaper FPGA family is sufficient to successfully implement an interface compliant with the VirtIO specification, the same should be possible with more expensive device families. Our assumption is that the integrated blocks and IP cores for the more expensive device families will provide similar or better capabilities than what is available for the device family we used.

At this point it is worthwhile to provide a brief description of the PCIe integrated block and the IP core used in this implementation. A Xilinx XC7A200TFBG484-2 device was used in this work. This device offers the 7-series integrated block for PCI express found in Xilinx 7-series devices. The Artix FPGA uses the second generation (Gen2) integrated block which has lesser capabilities than the more recent iteration of the integrated block (Gen3) used in higher end device families such as Virtex. The PCIe IP cores internally instantiate the integrated block and use the transaction (TRN) interface of the block to send and receive PCIe transaction layer packets [9] (TLP). We use the Xilinx DMA/Bridge Subsystem for PCIe Express (XDMA) IP core [21] for our experiments. In order to modify the PCI configuration space with new capabilities, we first need to understand how the PCI configuration space accesses by the host are handled by the IP core/integrated block, and the interfaces exposed to change the contents of the PCI configuration space. For the device used in this work, the PCI configuration space is implemented as part of the integrated block itself. Under default configuration, the integrated block responds to configuration space accesses from the host and the configuration space read and write access TLPs do not leave the integrated block via the TRN interface. We have discovered that the integrated block/IP core combination used in our implementation provides two different mechanisms to change the contents of the PCI configuration space. Since we used a lower end device, we can expect the higher end devices and corresponding IPs to provide similar or more flexible interfaces. For instance the 7-series Gen3 integrated block for PCIe provides a more feature rich configuration management interface compared to the interface available in the Gen2 block.

Adding new capabilities to the PCI capability list involves two tasks. First is to add the capability entries to the configuration space. The second task is forming the capability list by correctly setting the next pointer fields of the capabilities.

*1) Updating configuration space contents:*

*a) Configuration Management Interface:* The first mechanism to change the contents of the PCI configuration space is the Configuration Management Interface. Specifically, the signal group referred to as the Management Interface Ports allow user logic to read and write to the PCI configuration space. This interface has two shortcomings that make it unsuitable for our requirement. First, as we discovered through our testing, the integrated block for PCIe does not implement the full PCI configuration space as write-enabled registers. It seems that only specific portions of the configuration space are implemented as writable registers and the rest of the configuration space is read-only. This makes it unsuitable for adding capabilities to the configuration space. Even if the whole configuration space was writable by the user logic, this method is still not ideal because the configuration space modifications have to be done after the user logic has started operations. This means that the new capabilities may not be visible to the host when enumerating the device. This is especially true when using Tandem configuration [22]. PCI devices have a strict time limit around 100ms from reset, under which they should be ready for enumeration and if not the device will not be enumerated. When it takes longer than the above time limit to read a bitstream from flash memory and fully program an FPGA, Xilinx devices use a technique called *Tandem Configuration*, where the bitstream is built in two parts of which the first part contains only the bare necessities for PCI enumeration and the second part contains user logic. We wish the VirtIO enabled FPGAs to behave as any other PCIe device and to be enumerated at system boot up without requiring an additional PCI bus rescan after boot up.

*b) Configuration Space Access Forwarding:* The second method is forwarding the configuration space accesses to the user space over the TRN interface. While the 7-series integrated block for PCIe supports this feature, this is disabled in the XDMA IP core because the IP is not designed to handle configuration space read and write TLPs. Therefore, we have to make changes to the XDMA IP itself. There are two distinct ways the same outcome can be achieved.

**RTL modification:** The first method is to modify the source files for the IP core manually. We have to modify the source file which instantiates the PCIe integrated block. The configuration space access forwarding can be enabled by setting the attribute named EXT_CFG_CAP_PTR to an appropriate double-word address. All configuration space accesses for addresses above the specified address are forwarded to the user logic. Before modifying the RTL, the IS_LOCKED property for the IP should be set. Otherwise, Vivado compilation flow resets any modifications made to the RTL when recompiling the IP core. After setting the locked property, and modifying the source code, the IP is recompiled as an out-of-context module synthesis run. Out-of-context synthesis is where the IP is compiled independently of the rest of the design.

The XDMA IP used in this work exposes an interface named Dynamic Reconfiguration Port (DRP) that allows the user logic to dynamically control attributes of the PCIe hard block. The EXT_CFG_CAP_PTR attribute can be set using the DRP interface. However, it has the same limitation as the configuration management interface described above as the user logic has to be operational and finish a write operation over the DRP interface before the device enumeration for this method to work correctly.

**Modifying the XML file describing the IP components:** The XDMA IP internally instantiates another IP which corresponds to the actual PCIe integrated block and surrounding logic. We can separately generate this IP with configuration space access forwarding enabled, and add it to the XDMA IP. Xilinx IPs use an XML file to describe their sub-components. The second method is to update this XML file of the XDMA IP core, and to force the tool chain to use the sub-IP we previously generated instead of the one generated by Vivado tool flow as part of the XDMA IP. Similar to the first method, we still have to lock the IP and recompile as an out-of-context synthesis run after updating the XML file.

It should be noted that the PCIe IP for the FPGA we used in this work is free and the source files are not encrypted. If the IP is provided by the vendor in an encrypted format, a user may not be able to use the methods described here to implement a VirtIO compliant interface. However, we believe that this work still acts as a confirmation that it is possible to implement an interface fully compliant with the VirtIO specification and unmodified VirtIO drivers can be used to communicate with FPGAs. There is no limitation in hardware preventing such an implementation but artificial restrictions imposed by the FPGA vendor through the tool chain and encrypted source files.

After configuring the integrated block to forward the configuration space accesses to the TRN interface, we need to implement the configuration space registers with the new VirtIO capabilities, and the control logic to respond to configuration space read and write TLPs. The TLP header field is used to differentiate the configuration space access TLPs from the other TLP types on the TRN interface. Because the transaction interface is converted to an AXI interface by the XDMA IP, the TLP header information is not accessible from user logic. Therefore, the configuration space registers and the logic to build correct response TLPs are implemented within the IP itself by modifying the IP source files.

*2) Forming the capability list:* The next challenge is correctly forming the PCI capability list with the new VirtIO capabilities. The capability list is traversed by following the "next" pointer of each capability entry. Since the PCIe integrated block implements power management, PCIe, Message Signaled Interrupts (MSI), and MSIx capabilities, the next pointer of the last of those capabilities should be set to point to the first VirtIO capability. Surprisingly, the Vivado IP flow does not expose this option to the user even though configuration space access forwarding is exposed. We noticed that there are potentially multiple ways to set this pointer, but later discovered that not all methods satisfy our requirements.

*a) Using the DRP Interface:* Depending on the capabilities chosen when generating the IP core, one of the attributes, MSIX_CAP_NEXTPTR, or PCIE_CAP_NEXTPTR should be set to point to the first VirtIO capability using the DRP interface. However, this method is limited by the fact that user logic may not be operational in time to make this change visible at the time of device enumeration.

*b) RTL modification:* It is also possible to modify the IP source code directly to set the above attributes as parameters when instantiating the PCIe integrated block. Since we expect the FPGA to be correctly enumerated at system boot up, this is our preferred option to set the capability list next pointers.

### *C. VirtIO structures*

The VirtIO structures pointed to by the newly added PCI capabilities are used in both device initialization and regular operation. We briefly describe our implementation here. As we have successfully added VirtIO capabilities to the PCI capability list, we are free to place the VirtIO data structures on any of the BARs. since the XDMA IP core provides an AXI lite master interface with the user logic that is mapped to BAR0, we place all the VirtIO structures on BAR0 at different offsets. Each of the VirtIO capabilities inform the driver of both the BAR and the offset the corresponding data structure is located at. The VirtIO driver will perform read and write accesses to BAR0 in order to access the common configuration, notification, ISR status, and device specific configuration structures. The PCI configuration access capability does not point to a different structure on a memory or I/O region. Instead the driver can access other VirtIO structures by reading/writing configuration space addresses corresponding to this capability.

*1) Common Configuration Structure:* Common configuration structure perhaps is the most important out of the VirtIO structures. First, it informs the driver of device attributes such as the features offered, number of queues supported, size of each queue, etc. The driver writes information such as addresses for different virtqueue regions, MSIx vectors for each queue, etc. to the common configuration structure. The device, and driver feature fields are used in feature negotiation between the device and the driver. The device_status field has bits indicating different stages of device initialization.

Some fields in the common configuration structure represent a group of replicated fields. For instance, all fields referring to individual queues are replicated to match the number of queues supported. The 'queue_select' field determines which register is currently getting read/written to. The common configuration structure implementation should include the necessary logic to support this behavior.

*2) Notification Structure:* The driver writes to the notification structure in order to notify the device when there are new buffers added to the queues. Because the driver never reads the notification structure, it is not implemented as registers. Instead only the logic necessary to respond to write requests is implemented. The controller FSMs for virtqueues monitor the writes to addresses corresponding to the notification structure and initiate data movement between host and the device as necessary. Depending on the features negotiated, the driver may either write to the same address or write to a different address within the notification structure that corresponds to each queue. The data written include which queue the notification is for. Therefore, the virtqueue control logic can differentiate notifications for each queue.

### D. ISA status field

ISR status field is used by the driver to differentiate between queue and device configuration interrupts. This field is only useful when using legacy INTx interrupts. This field is implemented as a register which gets cleared on read.

### E. PCI Configuration Access Capability

The driver should be provided with an alternative access mechanism to the common configuration notification, ISR status, and device specific configuration structures through this capability. We achieved this is by further modifying the logic described in III-B1 which implements the modified PCI capability list. A new state machine is added to capture configuration space read/write TLPs intended for PCI Configuration Access Capability, and issue a read or write request as necessary to the module which implements the VirtIO structures. The portion of the PCI configuration space that corresponds to the PCI Configuration Access Capability is generally readable and writable by the host. Reads/writes to the 'pci_cfg_data' field of the capability are what trigger this special behavior. Rest of the capability fields at a given time determines which other structure is accessed via this alternative mechanism.

The connectivity between different modules is shown in Figure 3. Here, the UDMA module is a sub-IP which implements the DMA functionality. The source code for this module is not visible to the user. All the AXI interfaces going to user logic originates from this module. Please note that the block named $TRN \leftrightarrow AXI{-}S$ is not an actual sub-module of the XDMA IP. Rather it represents the functionality of multiple modules that sit between the PCIe integrated block and the UDMA module and perform conversions between the transaction interface signals of the integrated block and the AXI stream interface connected to the UDMA module. Similarly, the block named 'ext_cfg' represents modifications made to multiple modules of the IP core in order to implement part of the PCI configuration space external to the PCIe integrated block. We may refer to 'ext_cfg' as a single module from now on for simplicity.

When a configuration space access is forwarded from the PCIe integrated block, the 'ext_cfg' logic intercepts the TLP and responds to the request. The modified PCI configuration space registers with VirtIO capabilities are part of 'ext_cfg.' Memory read and write TLPs are not intercepted and instead sent directly to the UDMA module. When a configuration space read/write TLP intended for the PCI Configuration Access Capability is received, the 'ext_cfg' logic still intercepts the TLP. But, instead of immediately responding, it converts the TLP to a memory read/write TLP and sends it to the UDMA module as any other memory access TLP. The fields of PCI Configuration Access Capability are used to modify the fields of the TLP. It also indicates to the UDMA module that BAR0 was hit. This results in the UDMA module sending a read/write request to the 'virtio_controller' module which is connected to the AXI-lite interface and includes the VirtIO data structures. The response from the 'virtio_controller' is sent to the host.

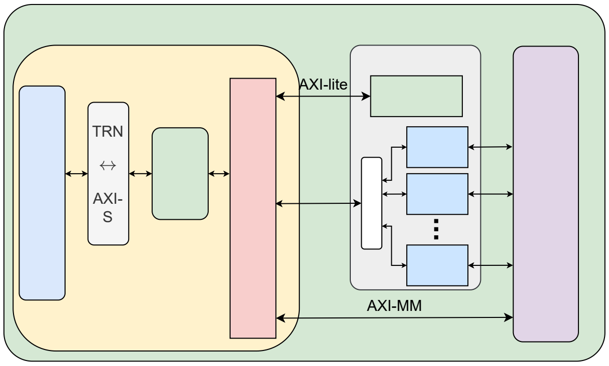


Fig. 3. VirtIO implementation.

### F. Data Movement

Data movement between the host and a VirtIO device is done using virtqueues. While the VirtIO specification describes two different virtqueue formats, we have opted to implement the simpler *split virtqueue* definition. It should be noted that there are no limitations from the FPGA for a *packed virtqueue* implementation. We have made our selection purely based on simplicity of the split virtqueue functionality.

A split virtqueue consists of three regions. These are referred to as available ring, used ring and descriptor table. The starting addresses for each of these regions, for each of the queues are written to the common configuration structure at device initialization. Therefore, when a notification is received for a given queue, the virtqueue controller modules can start data movement without further interventions from the host. We have opted to implement individual controllers for every queue supported by the device. The interface to control the DMA engine is shared between all the virtqueue controllers as shown in Figure 3. Our proof of concept implementation includes one RX and one TX virtqueue.

When data is available to be moved to from host to the device, the VirtIO driver places data in a buffer, updates entries in the available ring and descriptor table of the TX virtqueue, and notifies the device by writing to the notification structure. The virtqueue controller takes over the data movement responsibility at this point. It first accesses the header of the available ring to figure out how many ring entries are created by the driver. The valid entries are traversed one by one while doing the following; (i) Read available ring entry to determine the corresponding descriptor table entry. (ii) Read the descriptor from descriptor table. (iii) Use the descriptor to access the buffer and move data to the device. (iv) Create an entry in the used ring to indicate that an entry from the available ring was completed. (v) Update the header fields of used ring to resemble the new number of entries in the used ring. (vi) Notify the driver via an interrupt.

Each of the first five steps corresponds to the virtqueue controller programming the DMA engine to perform a data movement. The interrupt could be either a legacy interrupt or an MSIx interrupt. We have opted to use MSIx interrupts in our implementation. If sufficient MSIx vectors are enabled, the driver chooses to assign different vectors for each queue and one vector for configuration changes. The virtqueue controllers are responsible for selecting the correct MSIx vector. The vector number corresponding to each queue is written to 'queue_msix_vector' field of the common configuration structure at device initialization.

The VirtIO driver allocates buffers for the RX queue as well and notifies the device immediately after the device initialization. However, these are not used until the user logic asserts an input signal to the virtqueue controller corresponding to a RX queue. The virtqueue controller exposes an interface to the user logic to indicate the source address to read data from and the length of the buffer. When the user logic indicates that data is ready to be moved to the host, the virtqueue controller follows the same steps as when it received a notification for the TX queue. The only difference is that instead of moving data from host to device, it moves the data from device to host into a buffer previously allocated by the VirtIO driver.

*1) Controlling the DMA engine:* The use model for most DMA engines involve a driver running on the host providing the address where the descriptors are stored and the DMA engine first fetching the descriptors and then performing the data movement. The descriptor layout is specific to the DMA engine. Since the VirtIO drivers do not target a specific device, the descriptors in the descriptor table do not match the layout required by the DMA engine of the XDMA IP core. Furthermore, the VirtIO driver do not attempt to program the DMA engine at all. Instead, it provides the device the starting addresses of different regions of the virtqueues. The device has to use these and the information on the size of ring and descriptor table entries available in the VirtIO specification to control the DMA engine and perform data movement.

The XDMA IP core provides an interface named Descriptor Bypass Interface to feed descriptors to the DMA engine from user logic. The virtqueue controllers use this interface to program the DMA engine and move data to and from the FPGA. The DMA status ports are used to monitor the completion of data movement and advance the states of the virtqueue controller FSM. While we expect most PCI/DMA IP cores for different FPGAs to have similar interfaces to control the DMA engine from user logic, the user might have to implement their own DMA engine if such an interface is not available.

## IV. Evaluation

### A. Results

Our proof of concept implementation of the FPGA-based VirtIO console device is implemented using an Alinx AX7A200 development board. It has been tested on a host machine running Fedora 35 operating system. We have shown that with the added hardware support for VirtIO, the device gets enumerated as a VirtIO console device at system boot up without requiring additional PCI bus rescans, etc. Also the VirtIO driver loads completely without any errors and is agnostic to the fact that it is communicating with a physical device and not a virtual device. Figures 4, and 5 present the output of `lspci -v` on the host machine with the FPGA programmed with bitstreams for a Xilinx example PCIe DMA design and the VirtIO console device respectively. The PCI capability list in Figure 5 shows the VirtIO capabilities added by modifying the PCIe IP core.

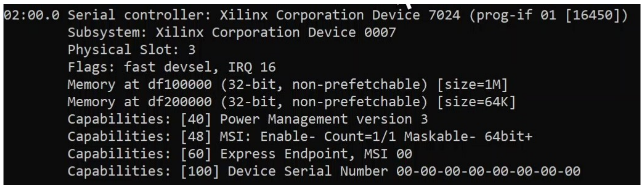
We have observed that the VirtIO PCI configuration access capability is never used by the virtio-pci driver when the FPGA is configured as a VirtIO console device. We have not validate this for other VirtIO device types. However, depending on the device type, it maybe possible to achieve the same behavior when interacting with the VirtIO drivers even if the PCI configuration access capability is not implemented.

While we have not performed detailed testing on performance, our limited testing has shown that the data movement performance is similar to the vendor provided reference driver. We expect further testing to reveal that the VirtIO configuration to have marginally lower latencies than the legacy driver because of the steps involved in data movement and the way the DMA engine is programmed.

*1) Host to Card (H2C):* To perform a H2C transfer, the legacy driver programs the DMA engine through multiple memory read and write accesses over PCIe. When the transfer is complete, the DMA engine interrupts the driver. In contrast, the VirtIO driver only performs one memory write to the VirtIO notification structure of the device. While the 'virtqueue_controller' has to perform multiple reads and writes to host memory, these are DMA transfers and the DMA engine is programmed directly from within the FPGA. The DMA engine can be programmed in a single clock cycle via the descriptor bypass interface, unless the engine is already busy moving data. We expect this to have a lower latency compared to the multiple memory reads and writes necessary for the legacy driver.

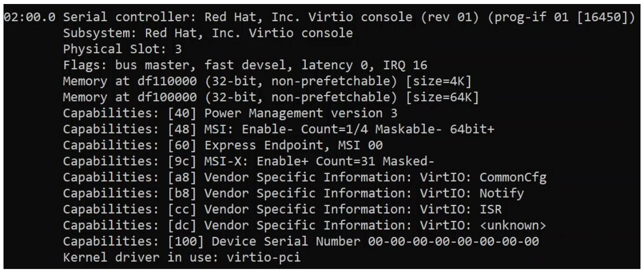
*2) Card to Host (C2H):* When data is ready to be sent back to the host and the legacy driver is being used, the FPGA has to first raise an interrupt to indicate to the driver that data is ready to be moved. Then the driver will program the DMA engine similar to the H2C scenario. If the driver has to figure out the source address on the FPGA by reading a CSR or a similar mechanism, that adds more memory reads to the total. When the data movement is complete, the DMA engine interrupts the driver.

As opposed to this, the operation of the virtqueue is much more streamlined. When user logic indicates the virtqueue controller that data is ready to be sent to the host, the controller determines the buffer location in host memory by reading available ring and descriptor table of the virtqueue. These are DMA operations and have lower latency compared to the memory accesses by the legacy driver. Then the data is moved to the buffer in host memory, and finally the VirtIO driver is notified via an interrupt. We expect this to have a lower latency than the legacy driver operation because of faster DMA operations and one less interrupt. If legacy INTx interrupts are used, the VirtIO driver has to perform one more read operation to the ISR status field. However, it is not necessary if MSIx interrupts are used because each virtqueue can be assigned a unique interrupt vector.



```
02:00.0 Serial controller: Xilinx Corporation Device 7024 (prog-if 01 [16450])
        Subsystem: Xilinx Corporation Device 0007
        Physical Slot: 3
        Flags: fast devsel, IRQ 16
        Memory at df100000 (32-bit, non-prefetchable) [size=1M]
        Memory at df200000 (32-bit, non-prefetchable) [size=64K]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] MSI: Enable- Count=1/1 Maskable- 64bit+
        Capabilities: [60] Express Endpoint, MSI 00
        Capabilities: [100] Device Serial Number 00-00-00-00-00-00-00-00
```

Fig. 4. Device enumeration for Xilinx example design.



```
02:00.0 Serial controller: Red Hat, Inc. Virtio console (rev 01) (prog-if 01 [16450])
        Subsystem: Red Hat, Inc. Virtio console
        Physical Slot: 3
        Flags: bus master, fast devsel, latency 0, IRQ 16
        Memory at df110000 (32-bit, non-prefetchable) [size=4K]
        Memory at df100000 (32-bit, non-prefetchable) [size=64K]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] MSI: Enable- Count=1/4 Maskable- 64bit+
        Capabilities: [60] Express Endpoint, MSI 00
        Capabilities: [9c] MSI-X: Enable+ Count=31 Masked-
        Capabilities: [a8] Vendor Specific Information: VirtIO: CommonCfg
        Capabilities: [b8] Vendor Specific Information: VirtIO: Notify
        Capabilities: [cc] Vendor Specific Information: VirtIO: ISR
        Capabilities: [dc] Vendor Specific Information: VirtIO: <unknown>
        Capabilities: [100] Device Serial Number 00-00-00-00-00-00-00-00
        Kernel driver in use: virtio-pci
```

Fig. 5. Device enumeration for VirtIO console device.

### B. Implementation challenges

Here we provide our assessment of the difficulty of implementing VirtIO support on an FPGA by modifying vendor IP blocks. According to our estimates, over 80% of the implementation effort was put into figuring out the necessary features of the vendor IP cores and using those features to add VirtIO capabilities to the PCI capability list. Implementing the VirtIO data structures and virtqueue controller logic according to the VirtIO specification was straightforward and quick. If the PCIe IP core in question is one used with more than one device family, the effort to figure out the IP details is not necessary when implementing VirtIO support on the other device families. For instance the Xilinx XDMA IP core used in this work can work with UltraScale+, UltraScale, Virtex-7 XT Gen3 (Endpoint), and 7-series Gen 2 (Endpoint) Integrated Blocks for PCIe [21]. Therefore, implementing VirtIO support on those devices can be done with considerably less effort. Also, the FPGA vendors can enable the necessary capabilities to make VirtIO support easily achievable.

### C. Future directions

This work can be extended by implementing other VirtIO device types such as virtio-net. It is also possible to define new VirtIO device types to correctly represent the different FPGA use cases. As future work, we will also perform a detailed performance analysis and comparison against legacy drivers for both host and guest OS VirtIO drivers communicating with the VirtIO enabled FPGA.

## V. Conclusion

In this work we investigate the requirements for enabling host-FPGA communication using native VirtIO drivers. We have identified and implemented the data structures, arbitration logic, queue, and DMA control logic required to create a VirtIO compliant host interface on an FPGA. We have done so by using both generic RTL modules and modifying the vendor provided PCIe IP blocks. Finally, we have demonstrated the effectiveness of our approach by implementing a VirtIO console device on a Xilinx 7-series device and showing correct device enumeration and communication. We believe that this work will act as a proof-of-concept for using unmodified VirtIO drivers to communicate with FPGAs. It also demonstrates that even lower-end FPGAs have the necessary capabilities to implement a host interface fully compliant with the VirtIO specification.

## Acknowledgments

This work was supported, in part, by grants from Red Hat and the NSF through awards CCF-1919130 and CNS-192550.

## References

[1] A.M. Caulfield, et al., "A cloud-scale acceleration architecture," in *MICRO*, 2016.

[2] Q. Xiong, C. Yang, R. Patel, T. Geng, A. Skjellum, and M. Herbordt, "GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs," in *FCCM*, 2019, pp. 258–266.

[3] J. Lant, J. Navaridas, M. Lujan, and J. Goodacre, "Toward FPGA-Based HPC: Advancing Interconnect Technology," *IEEE Micro*, vol. 40, no. 1, pp. 25–34, 2020.

[4] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers," in *FCCM*, 2020.

[5] V. Krishnan, O. Serres, and M. Blocksome, "Configurable Network Protocol Accelerator (COPA)," *IEEE Micro*, vol. 41, no. 1, 2021.

[6] C. Bobda, et al., "The Future of FPGA Acceleration in Datacenters and the Cloud," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 3, pp. 1–42, 2022, doi: 10.1145/3506713.

[7] P. Haghi, et al., "Reconfigurable switches for high performance and flexible MPI collectives," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 2, 2022.

[8] A. Guo, T. Geng, Y. Zhang, P. Haghi, C. Wu, C. Tan, Y. Lin, A. Li, and M. Herbordt, "A Framework for Neural Network Inference on FPGA-Centric SmartNICs," in *FPL*, 2022.

[9] PCI SIG Org., "PCI Express Base Specification Revision 3.0," Nov 2010. [Online]. Available: https://pcisig.com/specifications

[10] M. S. Tsirkin and C. Huck, "Virtual I/O device (VIRTIO) version 1.1," Dec 2018. [Online]. Available: https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html

[11] Xilinx, "Artix-7 FPGA family," 2022. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

[12] "IP Core Factory." [Online]. Available: http://xillybus.com/ipfactory/

[13] D. Richmond, A. Prost-Boucle, and J. O'Brien, "RIFFA," 2022. [Online]. Available: https://github.com/KastnerRG/riffa

[14] Xilinx, "Xilinx Run Time for FPGA," 2022. [Online]. Available: https://github.com/Xilinx/XRT

[15] E. Luebbers, S. Liu, and M. Chu, "Simplify Software Integration for FPGA Accelerators with OPAE." [Online]. Available: https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf

[16] E. Pérez Martín, "Virtio devices and drivers overview: Who is who," Jun 2020. [Online]. Available: https://www.redhat.com/en/blog/virtio-devices-and-drivers-overview-headjack-and-phone

[17] Silicom, "Silicom C5010X data center NIC." [Online]. Available: https://www.silicom-usa.com/wp-content/uploads/2021/12/PB_C5010X-NIC_v1.1_virtio.pdf

[18] RSPwFPGAs, "Virtio-FPGA-Bridge: Virtio front-end and back-end bridge, implemented with FPGA." [Online]. Available: https://github.com/RSPwFPGAs/virtio-fpga-bridge

[19] J. M. Mbongue, D. T. Kwadjo, A. Shuping, and C. Bobda, "Deploying multi-tenant FPGAs within Linux-based cloud infrastructure," *TRETS*, vol. 15, no. 2, pp. 1–31, 2021.

[20] J. M. Mbongue, F. Hategekimana, D. T. Kwadjo, and C. Bobda, "FPGA Virtualization in Cloud-based Infrastructures over Virtio," in *ICCD*, 2018, pp. 242–245.

[21] Xilinx, "DMA/Bridge Subsystem for PCI Express v4.1," Jun 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/pg195-pcie-dma

[22] ——, "7 Series FPGAs Integrated Block for PCI Express v3.3," Jul 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg054-7series-pcie