

Chapter 6

High-Bandwidth Low-Latency Interfacing with FPGA Accelerators Using PCI Express

Mohammadsadegh Sadri, Christian De Schryver, and Norbert Wehn

Abstract The need for high performance computing dictates constraints on the acceptable bandwidth of data transfer between processing units and the memory. Consequently it is crucial to build high performance, scalable, and energy efficient architectures capable of completing data transfer requests at satisfactory rates. Thanks to increased transfer rates obtained by exploiting high-speed serial data transfer links instead of traditional parallel ones, PCI Express provides a promising solution to the problem of connectivity for today's complex heterogeneous architectures. In this chapter, we first cover the principals of interfacing using PCI Express. To illustrate a practical situation, we select the Xilinx Zynq device and develop an example architecture which allows the x86 CPU cores of the host system, the ARM cores of the Zynq device, and the hardware accelerators directly realized on the FPGA fabric of the Zynq to share the available DRAM memory for efficient data sharing. We provide estimates on possible data transfer bandwidths in our architecture.

6.1 Introduction

As the energy efficiency requirements (e.g. GOPS/W) of silicon chips are growing exponentially, computer architects are seeking solutions to continue application performance scaling. One emerging solution is to use specialized functional units (accelerators) at different levels of a heterogeneous architecture. These specialized units cannot be used as general-purpose compute engines. However, they provide enhanced execution speed and power efficiency for their specific computational workloads [3]. There exist numerous applications for accelerators in both of the embedded and high performance computing markets. However, to make them accessible, a fast and flexible interconnect mechanism to the host system is crucial.

Efficient sharing of data in a heterogeneous architecture which contains different types of integrated elements is a challenging task. A versatile method should

M. Sadri (✉) • C. De Schryver • N. Wehn
Microelectronic Systems Design Research Group, University of Kaiserslautern,
Kaiserslautern, Germany
e-mail: sadri@eit.uni-kl.de; schryver@eit.uni-kl.de; wehn@eit.uni-kl.de

be devised to act as the interconnect between different components in such a system. The method should provide enough data transfer bandwidth meeting the requirements for high performance Central Processing Unit (CPU) cores and hardware accelerators. It should be scalable, allowing addition of new components to the system easily and without degrading overall performance. Furthermore, it is required to guarantee certain levels of robustness and reliability in data transfers. When private caches of CPU cores and dedicated memory of accelerators are used to store local copies of data, it is crucial to ensure that every processing element has a consistent view of the shared memory space. Consequently, the communication method should provide suitable means of ensuring data consistency among different components of the system.

When the components residing in a single chip should communicate with each other, considering the short distances between them and vast on-die routing resources, wide parallel data buses clocked at high frequencies can be used. Examples of such communication schema are Intellectual Property (IP) cores interfacing based on the Advanced eXtensible Interface (AXI) specification developed by ARM.

For inter-chip communications in which the signals should travel long distances, wide parallel data buses are not a suitable choice since the clock frequency of data transfer will be confined to few hundred MHz to ensure correct capturing of all data bits at the receiver side. When transferring data bits over a single link in a continuous serial stream however, the aforementioned problem does no more exist and the clock frequency of data transmission can be increased to multiple GHz. In this case it is crucial to ensure that the receiver can recover each single bit of transmitted data at a suitable time thus it is required to embed the information related to clock signal into the data stream itself. This is usually the source of an additional overhead in transmission of data e.g. in front of each 8 data bits which should be transmitted, practically 10 bits will be sent to ensure correct recovery of data at the receiver. In spite of this, the increase in bandwidth by serial methods is still far beyond parallel methods. Peripheral Component Interconnect Express (PCIe) or shortly PCI Express is currently the leading choice for connecting different hardware units in computational platforms.

6.1.1 History: From ISA to PCI Express

From the early days of appearance of IBM PC computers, different standards were introduced to allow extension cards and peripherals to be added to a computing platform and to communicate to its main CPU through an I/O bus. Examples of such buses include Industry Standard Architecture (ISA), Extended Industry Standard Architecture (EISA), and Micro-Channel. These interfaces had at most a data bus width of 32 bits and were running at frequencies below 10 MHz.

With the appearance of Peripheral Component Interconnect (PCI) the clock frequency of data transfers increased to 33 MHz and above. For applications with

higher demand on bandwidth, 64 bits wide versions of PCI were also introduced. The PCI based graphics card soon replaced the old ISA cards. However, in a short duration the bandwidth provided by these cards was also not enough for high definition 3D graphics, as a result a superset of conventional PCI called Accelerated Graphics Port (AGP) replaced PCI graphics cards. A performance improved version of PCI called Peripheral Component Interconnect Extended (PCI-X) dedicated to servers has a data bus width of 64 bits and operates at frequencies up to 533 MHz. This results in a total bandwidth of 4.26 GB/s over the bus. This is the highest bandwidth achievable by the PCI standard.

PCIe first introduced in 2004 is practically a replacement for all of the previous standards. It uses point-to-point high speed serial data links instead of a shared parallel bus. PCIe uses lower number of pins and smaller physical footprint in comparison to PCI while providing higher level of bandwidth. Thanks to the point-to-point connection topology, it provides better performance scaling as well as improved error detection and reporting mechanisms.

The glsPCIe connection between two nodes in the system can contain from 1 to 32 *lanes*. Each lane contains a separate high-speed serial link for receiving and another link for transmitting the data. Each high-speed serial link consists of two wires that transfer the data using differential signaling. As of the first version of PCIe (which was called PCIe Gen1) each link was capable of transferring 250 MB/s of data in each direction. The latest publicly available version (PCIe Gen3) increased the per-lane per-direction rate to 985 MB/s. For example a Gen3 PCIe card with 16 lanes (a 16X PCIe card) is cable of transferring up to 15.75 GB/s in each of the read and write directions at the same time.

6.2 Essential Basics of PCI Express

PCIe uses Transaction Layer Packets (TLPs) to transfer data between two nodes in the system. Each read or write transaction involves a series of one or more packet transmissions. These packets are responsible for transferring data, configuration parameters, messages, and event information between a PCIe device and the host. This also includes the interrupts generated by the PCIe device which should be delivered to the main CPU. Each TLP contains a header of around 16 bytes and payload of up to 4,096 bytes. The header contains information related to the type of the packet, its length, the ID of requester, its destination address, and so on. Two types of TLPs exists:

1. *Request TLPs*, which contain a request for an operation to a PCIe node in the system and
2. *Completion TLPs* that are generated by the completer and contain the response to the request.

For example, when a PCIe device decides to write to a specific I/O address, it generates a write request TLP which contains the destination address in its header

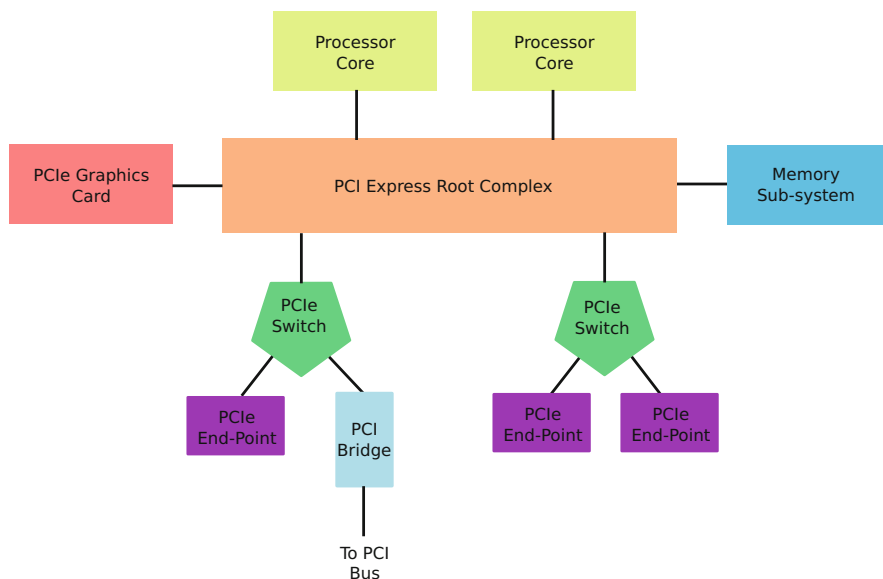


Fig. 6.1 Example architecture of a PCI express based platform

and the write data in its payload. The PCIe sub-system routes the TLP to its destination by looking up the address value in the header. On conclusion of write operation at the I/O device, it returns a completion TLP as the response to the requester to confirm the successful data transfer.

Figure 6.1 shows an example structure of a PCIe sub-system. The basic building elements of a PCIe system are in detail:

1. The *PCIe Root Complex* is usually responsible for connecting the processor and memory sub-system to the PCIe switches. The root complex generates PCIe transactions on behalf of the processor. It usually contain more than one PCIe Express ports.
2. *Switches* are responsible for routing incoming PCIe TLPs towards their suitable destination. The destination will be defined either by the address in the header of the TLP, by the ID of the destination peripheral, or based on the type of the packet (e.g. broadcasts from the Root Complex).
3. *End-Points* are practically the peripherals, boards, and devices installed on the hardware platform.
4. *Bridges* are used to allow hardware components not implementing PCIe directly to be added to the system. The bridge is responsible for performing translation between the other protocol and the PCIe.

PCIe devices are not available only as hardware boards which should be installed on a main board with PCIe backplane. A PCIe peripheral can also be a separate hardware unit in its own box and get connected to another platform through PCIe

Fig. 6.2 One MaxExpansion Gen3 X16 PCIe expansion kit containing one PCIe external cable and its adapter boards



external cables. Figure 6.2 shows a Gen3 X16 PCIe expansion kit which contains 2 adapter boards and a 3 m long PCIe external cable. This setup is capable of transferring data at rates near to 15.75 GB/s. It should be noted that both ends of the cable do not necessarily need to end up an adapter card. For example, it is possible to have one end of the cable connected to the adapter installed in the server machine and the other end directly enter a chip containing an integrated PCIe interface.

With the aid of PCIe fiber optic cables it is possible to extend the physical range of PCIe peripherals for one single platform up to 100 m easily. As an example, the hardware accelerator blocks for a high-end server can be located in another building while they are present to the rest of the system as Gen3 X16 capable PCIe peripherals.

6.2.1 Address Spaces and Base Address Registers

Every hardware component in a computing platform occupies a range of the available physical addresses in the system. Access to that hardware component is done through its base address and according to its address range. For example, each of the Dynamic Random-Access Memory (DRAM) memory, storage devices, and PCIe peripherals have their own specific base address and address range. The Basic Input/Output System (BIOS) is responsible for assigning addresses to the present hardware components at boot time or – for hot-plugging – when the hardware component is plugged into the system. For PCIe, recognition of available PCIe devices, identifying the capabilities and properties of each one, and assigning one or a set of addresses to the device is done through a process called *enumeration*.

Today's computing platforms running operating systems such as Windows or Linux use virtual addresses to manage system memory. Indeed, every process running on the system is given a range of virtual addresses by the Operating System (OS) that it uses for its execution tasks. For every process, accesses to the

memory or any of the hardware components in the system will be done by accessing specific locations in the process virtual address range. The OS is then responsible for converting the virtual address to the real physical one and initiating the transaction to the target. To perform the address translation fast and efficiently the OS uses a hardware unit called *Memory Management Unit (MMU)*.

The advantages of using virtual addresses are numerous, for example:

- Memory protection mechanisms can be implemented by the OS to disallow accesses to memory regions of other processes.
- Libraries that contain widely used routines by all processes can be loaded only once and easily be shared among all processes.
- Access to hardware components being used by several processes at the same time can be better governed by the OS.

However, this at the same time makes the task of software development for communicating to the PCIe hardware more challenging. At the first step, the driver which is responsible for talking to the PCIe component obtains the physical address of the device and its address range. These values are calculated at boot time by the OS. It then requests the OS for a region in the virtual address space to use for communicating with the device. Then the driver remaps the physical address of the device to the obtained virtual address. This way, by performing read and write transactions to virtual address locations, the driver can practically access the physical address locations of the PCIe peripheral. We further describe the basic architecture of a PCIe peripheral Linux Kernel driver in Sect. 6.7.

Now consider the fact that a PCIe peripheral has usually integrated CPU cores that are running an operating system themselves. They also have their own MMU. Moreover, each hardware component within the PCIe peripheral has its own internal physical address. Similar to the main system, the MMU is responsible for converting the virtual addresses generated by processes running on CPUs to equivalent physical ones. However, the difference is that this time every thing is happening within the PCIe peripheral.

Consider a simplified architecture like the one shown in Fig. 6.3. Suppose that the host CPU of the system wants to share an array of data with CPU cores within a PCIe peripheral. In order to do that, the host CPU can copy the data to the memory located inside the PCIe peripheral. Several address translation steps are required to accomplish this task:

1. The virtual address of the memory location which holds the array on the host system should be converted into its equivalent physical address.
2. The virtual address through which the driver running on the host system talks to the PCIe peripheral should be converted into its physical equivalent as well. At this stage a transaction can be initiated to transfer the data from the memory to the PCIe peripheral. This transaction can be initiated by a Direct Memory Access (DMA) engine which we describe later in more detail.

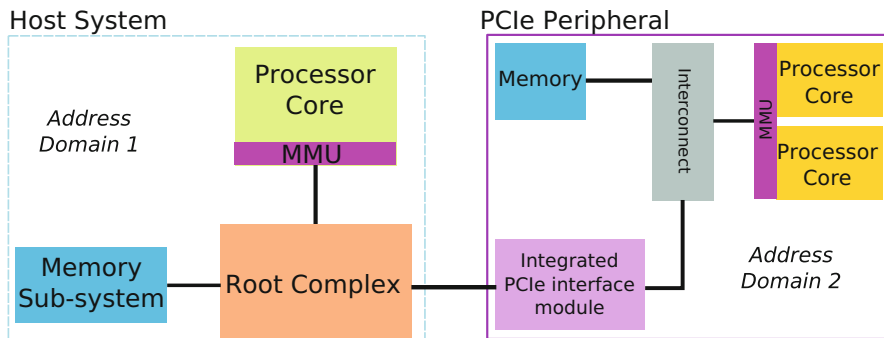


Fig. 6.3 A simplified block diagram of a host system and a PCIe peripheral which contains a set of computing elements inside

3. When the transaction passes the integrated PCIe interface module, its address should be substituted with the correct physical address within the PCIe peripheral hardware subsystem. This physical address usually resides some where in the range of memory address.
4. Finally, for the CPU cores within the PCIe peripheral to access the shared data, a conversion between the virtual address of the shared data array and its corresponding physical address should be done. This will happen using the MMU within the PCIe peripheral.

As we see the address translation can be a tedious task. As a result it is crucial to make sure that it is happening only when it is required and then it is performed in an efficient manner.

The PCIe Base Address Registers (BARs) have a special meaning: they are the base address values assigned to the PCIe peripheral by the host at boot time. However, our example integrated PCIe interface module from Fig. 6.3 has two sets of base address registers: One set representing the physical address of the peripheral for the host system and another set representing its base address as it appears to the local CPU cores within the card. When performing data transactions initiated by the host and targeting the PCIe peripheral or vice versa, it is crucial to have fast translation between these address domains. To improve the performance, the address translation task is usually directly implemented in the hardware of integrated PCIe module. There exist configuration registers within the module where the required translations between two address domains can be defined.

6.3 Interfacing to PCI Express Using FPGAs

With parallel I/O data transfer schemes reaching their bandwidth limits for chip-to-chip communications, high-speed serial interfaces are replacing them wherever possible. A high-speed serial interface transmits the data over differential signal

lines in self-synchronous mode. This means the information related to the clock signal of the transmitter is integrated into the data stream itself. At the receiver side the clock signal will be extracted from the data stream using a Clock Data Recovery (CDR) circuit, which mainly contains a Phase Lock Loop (PLL). The principal component used in high-speed serial links is a Serializer/Deserializer (SerDes). A SerDes is a hardware unit responsible for converting the received parallel data into serial, adding clock information and putting it on the transmission line, and on the other hand, receiving the serial data from outside, extracting the clock information and converting the data into parallel.

Field Programmable Gate Array (FPGA) devices that are widely used by engineers in numerous products should also provide the possibility of talking to outside world through high-speed serial links. The shift from parallel data transfer to high-speed serial interfaces, however, does not come without challenges. In fact, designing a high-speed serial interface from scratch can be so complicated and time consuming that the engineers may prefer to continue using traditional parallel transfer solutions. This made FPGA manufacturers to integrate the rapid serial I/O interfaces as ready-to-use hard-core IP blocks into their products. Thus, all the designers need to do is to configure the IP core according to their serial data transfer specifications and implement suitable hardware modules on the FPGA for interacting with the rapid serial I/O block.

For example, Xilinx has integrated high-speed serial interfaces into its FPGAs from the early Virtex-II devices [2]. Primarily they were called *Rocket-I/Os*. This name was later mostly replaced by Multi-Gigabit Transceiver (MGT). Table 6.1 shows available MGTs in Series-7 and UltraScale Xilinx products [8]. The MGTs are the basis for realization of all of the famous high-speed communication protocols using Xilinx FPGAs. Examples include PCIe, Serial AT Attachment (ATA), 10 Gb Ethernet, Infiniband, and so on.

Table 6.1 Multi-gigabit transceivers available in UltraScale and Series-7 devices. For each family only the highest speed MGT is shown

Device	Transceiver type	Max performance (Gbits/s)	Max transceivers	Total bandwidth (TBits/s)
Kintex UltraScale	GTH	16.3	64	2
Virtex UltraScale	GTY	32.75	120	5.8
ZYNQ 7000	GTX	12.5	16	0.4
Artix-7	GTP	6.6	16	0.21
Kintex-7	GTX	12.5	32	0.8
Virtex-7	GTZ	28.05	96	2.8

6.3.1 Integrated Block for PCI Express

To have a fully operational PCIe end-point or root complex, in addition to MGTs, several other blocks are required as well. These additional modules are responsible for realization of transaction and data-link layers of the PCIe protocol. This includes tasks like configuration management, generation and processing of TLPs, flow-control, power management, data protection, error checking, and status tracking. One possibility is that the developer implement all of these additional functionality in Register-Transfer Level (RTL) and therefore using FPGA resources. But also ready-to-use soft-IP cores [10] can be used. However, this results in valuable FPGA resources being consumed by the core, limiting the freedom of designer to implement his own custom logic. Therefore, in some FPGA devices the PCIe block is available as a hard-IP core, meaning that the required logic is already integrated into the silicon and is ready to be used.

Regardless of being a hard- or soft-IP core, the interfaces to the PCIe block are always the same. This allows a developed design to be adapted to different FPGA devices in a short time. Basically, the main interfaces of a PCIe block for transferring data to other modules within the FPGA fabric are based on AXI stream protocol. Figure 6.4 shows the symbolic representation of a Gen3 8X PCIe block for a Virtex-7 device and its main connections to other modules on the FPGA and also to the

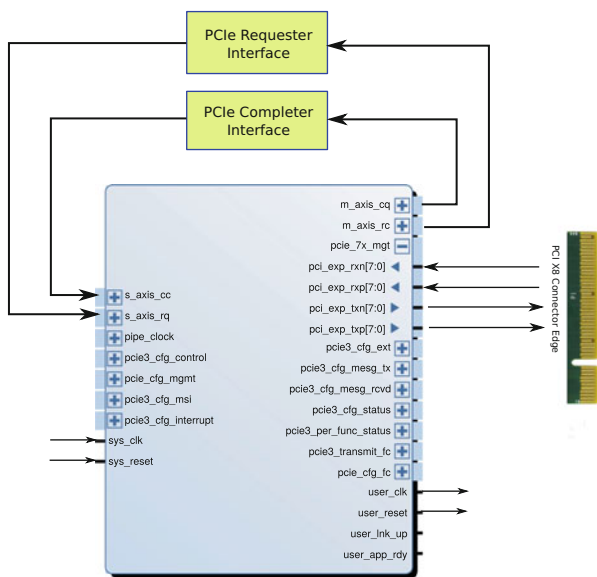


Fig. 6.4 A simplified block diagram of a PCIe block and its main connections to other modules on the FPGA and the outside world. The PCIe block symbol is taken from a Xilinx Vivado project consisting a Gen3 X8 PCIe interface

outside world. In this figure, the PCIe block is shown in blue and then rectangles in green are the logic implemented by the user for talking to the PCIe block.

As described in previous sections, in every PCIe transaction one node in the system is the requester which produces a read or write request and another node is the completer which responds to the requester by performing the required action. The PCIe block shown in Fig. 6.4 is capable of playing both roles. When the user decides to act as the bus master and to initiate a transaction to another PCIe node in the system, it does so through its *PCIe Requester Interface* unit by sending a suitable request packet to the *Requester reQuest (RQ)* interface of PCIe block. The PCIe block forwards the request to the PCIe backplane. When the PCIe block received the response from the completer it sends the response over the *Requester Completion (RC)* interface to the user's Requester Interface module.

The PCIe block can also act as a completer. When another node in the system decides to initiate a transaction to our PCIe peripheral it sends the request with the physical address of our PCIe peripheral. Upon receiving the request TLP, the PCIe block forwards the request to the user's *Completer Interface* through its *Completer reQuest (CQ)* port. The user developed block performs required actions indicated by the request and sends back the response to the *Completer Completion (CC)* interface of the PCIe block. For a Gen3 X8 PCIe block, each of the mentioned ports have a data bus width of 256 bits and are running at 250 MHz. The `sys_reset` and `sys_clk` signals are fundamental to the operation of the PCIe block. These signals can be obtained from the PCIe backplane. There is also the possibility of generating these signals locally on the PCIe peripheral if needed. The `user_reset` and `user_clk` signals are generated by the core and can be used for the logic that user develops on the FPGA fabric for communicating to the PCIe block.

6.4 Introduction to AXI

Looking at a typical System on Chip (SoC) design we see a large number of different modules instantiated and connected together to deliver the required functionality of the product. In order to realize those connections easily and fast with the goal of building a new architecture it is crucial that all of the modules interface to outside world based on a same language. This means that all of the modules should obey a same set of rules and should use the same set of signals as their interface to the rest of the logic. Having a library of different modules meeting this requirement in hand, a designer can build new systems very easily by connecting the required components together. This is the basic motivation behind creation of SoC buses like WishBone [6], IBM CoreConnect [5], and ARM AXI [1].

A SoC bus is not necessarily a data transfer medium which is shared by several modules. It can also be an architecture which provides point-to-point connection between different units of the system, allowing them to initiate transactions concurrently. This is the case for the AXI interconnect, for example.

The basic elements of an AXI based architecture are as follows:

1. *AXI Masters* are units that initiate read or write transactions.
2. *AXI Slaves* are units that receive read or write transactions and produce the suitable response to them.
3. *AXI Interconnects* play the role of switching elements between AXI Masters and AXI Slaves. They are responsible for routing the transactions and data from an AXI Master to the specified destination.

There exist two types of AXI interfaces:

1. *AXI stream* and
2. *AXI memory mapped*.

An *AXI stream* interface is suitable for modules which receive a stream of data, perform some processing on it, and generate another stream of data as the output. In AXI stream interfaces, the source from which the data is coming and the destination to which the data is going are defined from the beginning. Thus the transactions do not need to carry the address for the destination. This simplifies the logic required for interfacing to AXI stream extensively. Figure 6.5 shows the main signals between an AXI stream master and an AXI stream slave plug. In AXI stream, the direction of data is always from the master to the slave.

The TVALID signal indicates to the slave unit that a new data is available. TREADY is generated by the slave and indicates that the AXI stream slave is ready to receive the data. The transmission of data happens when both of the signals are active. TDATA is the actual data being transferred and TLAST indicates if the current data which is being transferred is the last data of the packet. In such a configuration, both modules work at the same clock domain. To transfer data between one AXI stream master and one AXI stream slave where both are in different clock domains, an AXI stream asynchronous First in, First Out (FIFO) can be used.

The simple AXI stream interface does not answer all of the connectivity needs in a complex SoC. This is because in many situations the computational task is more complicated than just performing a set of calculations on an incoming stream of data. Indeed, the task usually involves performing read and write accesses to different variables and array elements that are stored in a memory. Consequently, the read and write transactions initiated by the module need to contain an address to indicate to the target location in the memory for the read or write operation.

Furthermore, the simple case in which one module is always receiving its input data from another fixed module is not always true. It happens that a module needs to talk to a different module in the architecture each time. Again, this highlights then need for an address in the initiated transaction for indicating the target module.

Fig. 6.5 Main signals between an AXI stream master and an AXI stream slave module

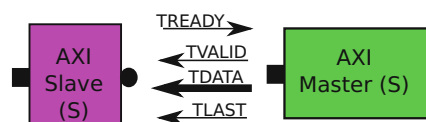
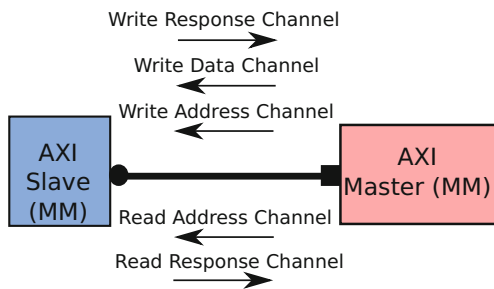


Fig. 6.6 Main channels of an AXI memory mapped interface



The *AXI memory mapped* interfaces solve these issues by adding the address channels to the AXI interface. In contrast to an AXI stream interface that contains only one channel for data transmission and in a fixed direction, an AXI memory mapped interface consist of five channels. The AXI memory mapped interface can perform either reads or writes and for each of these operations since it has a separate read address and write address channels. To improve the performance of data transfers, AXI interfaces support burst transfers of up to 256: For a read or a write transaction, only one address is indicated over the corresponding address channel and the rest of the addresses are automatically calculated at the slave side.

Figure 6.6 shows the five channels of an AXI memory mapped interface. Each channel contains a similar set of signals as one AXI stream interface. In each channel, there are additional signals to carry the information related to transactions bursts, quality of service, memory protection, caching of data, and so on.

As we see in Fig. 6.6, the *address channels* are always outputs from the AXI memory mapped master and inputs to the slave. The *read data channel* that carries the data read of the specified address by the master is an input to the master. Also the *write response channel*, which informs the master if the write transaction has been successful or not, is an input to the master module.

A typical read or write transaction begins first by putting the address on the read or write address channels. After the address is accepted by the slave, the actual data transfer will be performed. For each of the read or write data transfers there are signals which indicate if the transfer has been successful or not. To improve the performance of data transfers it is possible to use all of the five channels in parallel. For example, while an AXI master is receiving data of the previously initiated read transaction, it initiates a write transaction over the write address channel.

Furthermore, there exist the possibility that an AXI slave responds to incoming requests in a out-of-order fashion, i.e. that it receives a set of requests from different masters and it generates the suitable responses to each request as soon as it had the data ready for that specific request. To enable this functionality, a mechanism is required to identifying each transaction. In fact when the slave produces a response to the request of an AXI master, it should be able to indicate that the current response belongs to which of the received requests. The AXI protocol provides an ID signal over each channel. Basically each incoming request over either read or write address channels to an AXI slave has its own specific ID. When the AXI slave produces the

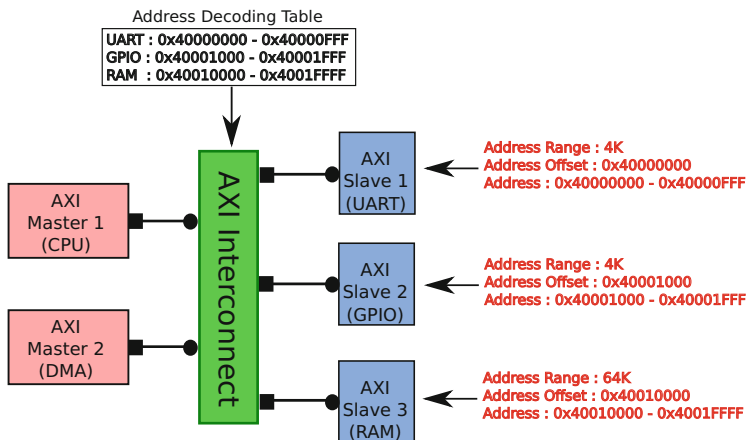


Fig. 6.7 An AXI interconnect operates as the switching element in an architecture consisting modules with AXI memory mapped interfaces

response to the request over the read data channel or the write response channel, it also reflects related ID value of that transaction over the response channel.

In an architecture consisting of modules with AXI memory mapped interfaces, switching mechanisms are required to route transactions to suitable destination based on the transaction address. *AXI interconnects* are responsible for this task. Figure 6.7 shows an example design featuring 2 AXI masters and 3 AXI slaves. Each slave in the architecture has a base address and an address range. The AXI interconnect contains a decoding table which is basis for deciding to which of the slaves an incoming transaction should be routed. The AXI masters can practically initiate transaction to every location they want in the address space. Obviously, only those transactions which fall in the address range of one of the AXI slaves will be responded.

6.4.1 AXI PCIe Bridge

Consider an FPGA based PCIe card which acts as a hardware accelerator and its main operation is to receive an incoming stream of data, to perform a defined set of calculations on the data stream, and then to return back the results to the host. For such an accelerator, the integrated block for PCIe described in Sect. 6.3.1 is a suitable choice. In fact, many of the applications deal only with streams of data. For example, they need to process a stream of data, or to transfer the incoming stream of data to another location e.g. to a fiber optics channel. In these cases, being able to randomly access data, or to keep a history of data in a memory and to retrieve it later if needed, is not required.

However, there exist also many applications in which memories and random accesses to memory are heavily required. For example, imagine a PCIe card which is responsible for accelerating data base operations. The card stores the tables in its internal DRAM to which the FPGA has a very low-latency and high-bandwidth access. Then it receives different query requests over the PCIe interface, performs the required operations using the tables stored on its DRAM and sends back the results. In such a scenario, the stream based architecture is no more efficient. Indeed the logic for the hardware accelerator implemented on the FPGA will mostly be based on memory mapped interfaces. As a result, the output of the integrated block for PCIe should be connected to a unit which is responsible for receiving PCIe packets over AXI stream, looking up their properties and destination address, and converting them into equivalent AXI memory mapped transactions. For this case e.g. Xilinx provides an integrated PCIe IP core with AXI memory mapped interfaces, eliminating the need for using the AXI stream based blocks.

Figure 6.8 shows the symbolic representation of a PCIe AXI unit and also a simplified example architecture representing how this block gets connected to the rest of the system. This figure also contains a processing core and some modules with AXI slave and master plugs which are implemented on the FPGA fabric. In addition, there exist modules that may have more than one AXI plug. AXI interconnects are responsible for connecting the AXI interfaces to each other.

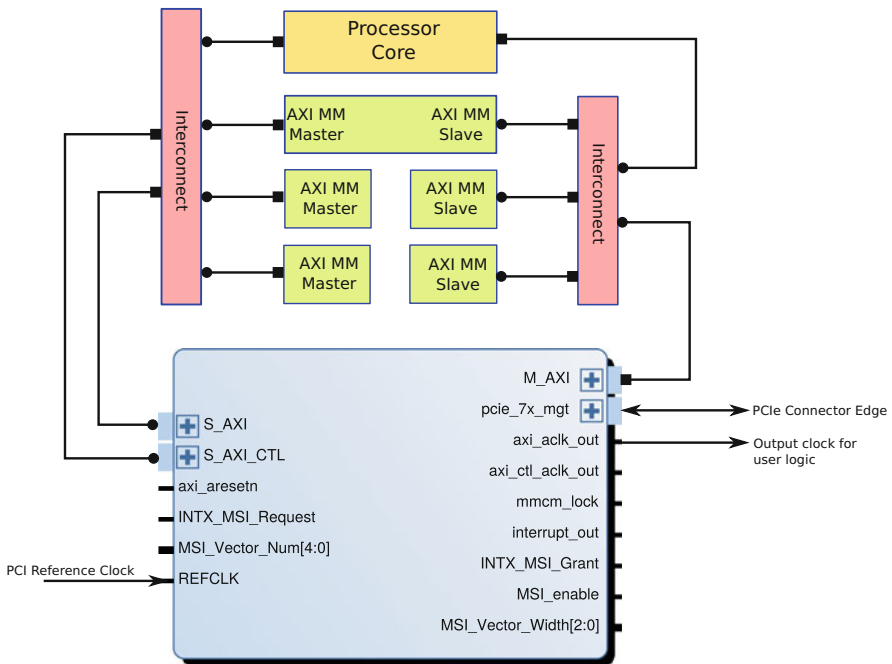


Fig. 6.8 Simplified block diagram showing how a PCIe AXI bridge gets connected to the other modules on the FPGA fabric and also to the outside world

In Fig. 6.8, the `pcie_7x_mgt` port is in fact the interface port of MGTs and should go to the PCIe edge connector. `M_AXI` is an AXI memory mapped master port which will initiate read and write transactions based on the received TLPs at PCIe side. The AXI master port will go to an AXI interconnect from which it gets connected to AXI slaves in the architecture. The `S_AXI` port of PCIe AXI bridge is used when the modules on the FPGA decide to initiate transactions to different address locations over the PCIe bus. For example, the processor core or one of the AXI masters may perform a read or write operation to a specific location in the host's DRAM memory through the `S_AXI` port. In this case the PCIe card acts as a bus master on the PCIe backplane.

Finally, the `S_AXI_CTL` port is an AXI slave port of PCIe AXI bridge which is used for configuring the bridge. Usually this is a task performed by a processing core. For example, as described in Sect. 6.2.1 the PCIe AXI bridge converts the address of incoming transaction from the address space of the host computer to the address space of the embedded architecture on the PCIe card. This address translation can be configured through the `S_AXI_CTL` port using the processor core. Furthermore, the AXI slave port can be used for reading the status and configuration information of the bridge. If the local processing unit is running an operating system like Linux, then a kernel level driver should also be developed to allow the software to access the resources of PCIe AXI bridge.

6.5 Xilinx Zynq Architecture

This section introduces the basic features of the Xilinx Zynq All Programmable SoC required to understand the PCIe example presented in Sect. 6.6.

The Xilinx Zynq All Programmable SoC is a hybrid device with a fast interconnect [12]. It consists of two parts:

1. The *Programmable Logic (PL)* that is roughly a full-featured FPGA and
2. The *Programmable Systems (PS)*, a complete sub-system with ARM CPU cores and different peripherals.

The PS contains the following items:

- An ARM Cortex MPCore-A9 dual core processing engine which also contains NEON Single Instruction Multiple Data (SIMD) units. Each ARM core has its own L1 data and instruction caches. Each cache block has a size of 32 KB.
- One L2 cache with the size of 512 KB which is shared between two CPU cores. The ARM PL310 cache controller is used for implementation of this unit.
- A *Snoop Control Unit (SCU)*, which ensures coherency between the contents of the caches.
- An On-Chip Memory (OCM), a multi-port memory block of 256 KB that can be accessed by the CPU or other ports and units in the system.

- A DMA controller that can be used for transferring data between peripheral and DRAM memories.
- A multi-port memory controller, which is responsible for connecting to DRAM memories and receiving read/write requests from different sections of the hardware and passing them to DRAM.
- A large ensemble of different peripheral such as Universal Asynchronous Receiver/Transmitter (UART), Gigabit ethernet, Universal Serial Bus (USB) peripheral and host, Controller Area Network (CAN) bus, Inter-Integrated Circuit (I²C) Bus, Secure Digital (SD) Card interface, General-Purpose Input/Output (GPIO), and so on. . . , which can be configured and used by the ARM CPU cores very easily.
- An interconnect based on ARM NIC-301 design that connects different blocks of hardware inside PS together.
- A set of AXI interfaces (as shown in Fig. 6.9) are implemented to make the communication between PS and the PL logic possible.

Basically, these AXI interfaces divide into two groups:

- AXI memory mapped master interfaces (GP), connect to AXI slaves residing on the PL. The CPU is able to initiate read/write transactions over these AXI masters to transfer data to PL modules. There are two 32 bits AXI master ports available in the Zynq device: GP0 and GP1.
- AXI memory mapped slave interfaces (High Performance (HP), Accelerator Coherency Port (ACP) and SGP), connect to the implemented AXI masters on the PL. There exist four HP ports and one ACP. Each of these interfaces implements

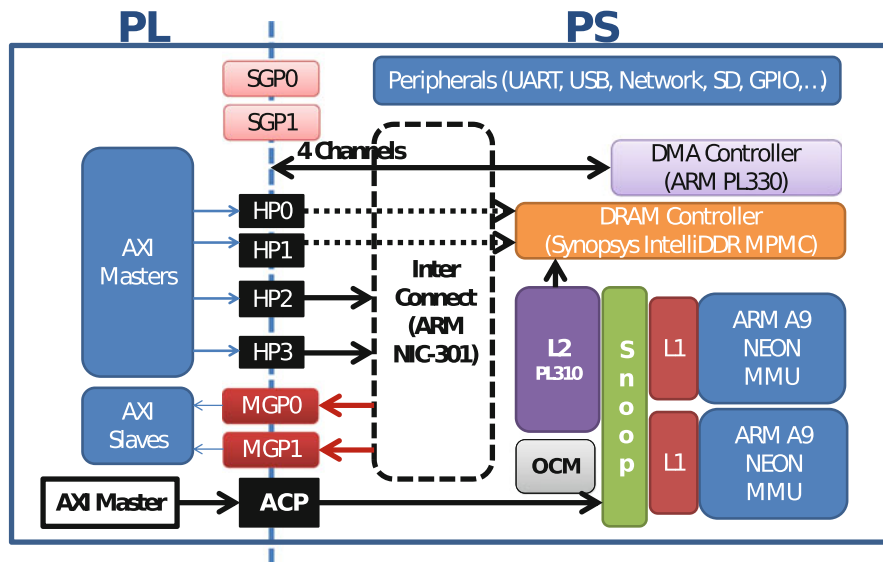


Fig. 6.9 A block diagram representing important elements of the Xilinx Zynq device

a full-duplex 64 bits connection, meaning that at every clock cycle, total 16 bytes of data can be transferred on AXI read and AXI write channels concurrently. The two SGP0 and SGP1 interfaces implement 32 bits connections.

There exists a defined memory map for the Zynq device [12] that indicates the address range of each logic block. Every AXI slave unit implemented on the PL will also occupy a part of this address range. It should be noted that except the CPU cores and their L1 instruction caches the rest of the system is using physical address values.

The ACP is connected to the ARM SCU. Thus, it provides the possibility of initiating cache coherent accesses to the ARM sub-system. Careful use of the ACP can improve overall system performance and energy efficiency. However, inappropriate usage of this port can adversely affect execution speed of other running applications because the accelerator can pollute precious cache area [7].

6.6 Example PCI Express Design Based on Zynq

We develop an example architecture on the Xilinx Zynq to allow the host system and the Zynq ARM subsystem to share their DRAM memory spaces over PCIe. When the data sharing between the host system and the processing elements on the PCIe card is complete, additional hardware accelerator modules can be added to the Zynq PL allowing for accelerating computational tasks. Our target hardware for this project is the Xilinx ZC706 board that contains one Zynq 7045 device. The card provides a PCIe Gen2 X4 interface over a PCIe connector. The developed architecture tries to keep every thing as simple as possible while providing the possibility of reaching acceptable bandwidth on the PCIe interface.

Figure 6.10 represents the developed architecture. This architecture consists of three main elements:

1. The Zynq PS,
2. The PCIe AXI bridge, and
3. A DMA engine.

For configuring the PCIe AXI bridge and reading its status information through the S_AXI_CTL port, we use the GP1 AXI master port of the Zynq PS. Through this port, the ARM CPU cores can have access to configuration and control registers of the PCIe AXI bridge. For this part of the circuit, `axi_ctl_aclk_out` generated by the core is used as the clock signal. For the rest of the logic `axi_aclk_out` is used as the clock.

Transactions initiated by other PCIe nodes in the system and received by our card will arrive at M_AXI port of PCIe AXI bridge. From there these transactions will be routed to the DRAM memory space through the HP0 AXI memory mapped slave port. Obviously, suitable address translation should be done before the transaction enters the AXI architecture. Practically for the incoming transaction address, the

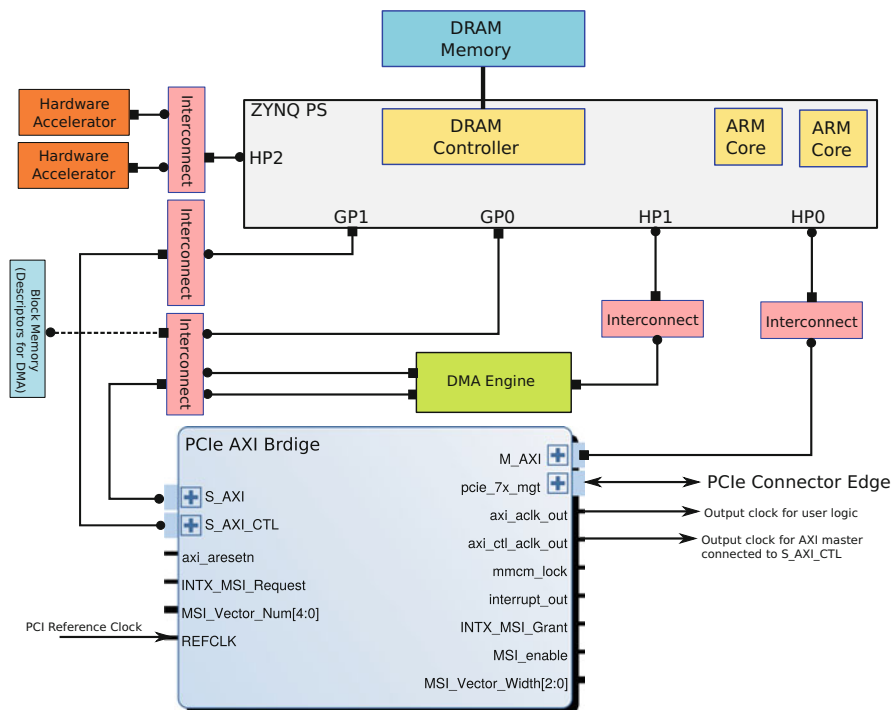


Fig. 6.10 A block diagram representing the architecture of our PCIe based hardware to share the DRAM memory of the host system with the ARM CPU cores on the Zynq and to share the DRAM memory connected to the Zynq PS with the host system. *Squares* are AXI master plugs and *circles* are AXI slaves

base address of the PCIe device should get dropped and instead a suitable base address according to address arrangements in the AXI architecture should be added.

The `S_AXI` port of the PCIe AXI bridge has basically two masters: The ARM CPU cores and the DMA engine. The ARM cores can directly initiate read and write transactions to the address space of the host through the `GP0` port. Again, the destination address for the transaction here will be the address assigned to `S_AXI` port of PCIe AXI bridge, and the translation to the address space of the host will be done within the bridge. Here, the kernel level driver of our PCIe peripheral which is running on the host system, will allocate an amount of memory on the host's DRAM. It then obtains and passes the physical address of the allocated memory to the ARM host on our PCIe peripheral. The ARM host configures the address translation mechanism within the PCIe AXI bridge so that the generated transactions by this block over the PCIe backplane end up the memory range allocated by our driver on the host's DRAM memory.

A similar procedure as what we described holds true when the host CPU wants to access the DRAM memory of the ARM host. This time the kernel level driver running at the ARM side is responsible for memory allocation. Again based on the obtained address values the address translation mechanism within the PCIe AXI bridge will be configured.

When it is required to transfer a large amount of data with a high bandwidth, using the CPU cores for handling data transfers is not efficient. This is true for both CPU cores of the host as well as the ARM CPUs of the PCIe peripheral device. Generating read or write transactions using the CPU core, even when it is done without any interruption, does not eventually result in a satisfactory performance. Consequently, there should exist a dedicated hardware responsible for performing data transfers to and from the PCIe AXI bridge. The DMA engine instantiated in our example architecture is responsible for initiating data transfers to the host without disturbing the CPUs.

In our architecture, the ARM CPU cores are responsible to define the set of data transfer tasks that the DMA engine should do. Each transfer task for the DMA is defined in a structure called descriptor. Through GPO port, the ARM CPU core copies the descriptors to a dual-port block memory. The other port of the block memory is practically connected to the DMA engine, which has a dedicated master port to read the descriptors. Each descriptor contains a source address for the transfer which should be done, a destination address, and the length of the transfer. The block memory can keep a large number of descriptors written to it by the CPU. The DMA engine can operate in a cyclic manner, meaning that by reading and executing the transfer for the last descriptor in the block memory, it can jump back to the first descriptor and continue its operation.

Our DMA engine has two AXI memory mapped master ports: One port is practically connected to the `S_AXI` port of the PCIe AXI bridge initiating transactions over PCIe. Another port is connected to the HP1. For a read operation, the DMA engine initiates the read transaction over PCIe and when it received the read data it writes the incoming data to the DRAM memory through the HP1 port. For a write operation the DMA engine reads the write data from the DRAM memory through the HP1 port, then it initiates a write transaction over the PCIe bus by writing the data to a suitable address location of the `S_AXI` port.

6.7 Linux Kernel Level Driver

As described in Sect. 6.4.1, our architecture consists of two sets of CPU cores: The host CPU cores that are usually x86 and the PCIe device CPU cores, which in our architecture are ARM v7 cores. We assume that the Linux OS is running on the host CPU cores. For the PCIe device, in most common situations, Linux is also running on the ARM CPUs. In order for user-level applications running on the Linux

to be able to interact with the PCIe device, a Linux kernel level driver is needed. As a result, for our architecture two Linux kernel level drivers should be developed separately.

The driver running on the host system should be able to find the PCIe device, allocate I/O and memory regions for it, register required interrupt lines and their corresponding interrupt handlers, and to enable user level applications to communicate with the device. For this purpose, the driver uses structures and functions integrated into the Linux kernel that facilitate interacting with PCI based peripherals. In fact, from the view point of the host driver, our hardware is a PCI peripheral. However, from the view point of the driver running on the ARM host, the PCIe AXI bridge is just an AXI peripheral similar to the other components in the architecture.

For the driver running at the host side, it first announces to the Linux kernel the vendor and device ID of the PCIe device it is seeking for. In addition to the ID values, the driver also gives the Linux kernel pointers to two implemented subroutines inside the driver. The first subroutine is executed when a PCIe device is found in the system with the same ID values as the ones indicated by the driver and the second subroutine is executed when this PCIe device gets removed from the system. The ID values and pointers to subroutines is given to the Kernel through a structure called `pci_driver`. Code Listing 6.1 shows an example definition of this structure.

```
1 static struct pci_driver pci_drv_template =  
2 {  
3     .name = "pci_drv_template",  
4     .id_table = pci_drv_ids,  
5     .probe = device_probe,  
6     .remove = device_deprobe,  
7 };
```

Listing 6.1 Example code showing how `pci_driver` struct is defined.

During the `device_probe` subroutine, the driver obtains the physical address at which the PCIe device is located. This is done through `pci_resource_start` Linux kernel call. The driver then asks the Kernel for permission to access the physical address range of the PCIe device. If granted, a remapping of the physical address of the PCIe device to virtual address space will be done by the driver. This is done with the aid of `ioremap` Linux kernel call. Having the virtual addresses, the driver is able to perform read and write transactions to the address space of the PCIe device using `writel` and `readl` system calls. Code Listing 6.2 shows an example implementation of the `device_probe` routine.

```

1 static int device_probe
  ( struct pci_dev *dev, const struct pci_device_id *id )
3 {
  // get the first i/o region
5  ioport = pci_resource_start( dev, 0 );
  iolen = pci_resource_len( dev, 0 );
7
  // request memory region
9  request_mem_region( ioport, iolen, "ZC706" );
11
  // map hardware memory space to virtual space
  pci_bar_vir_addr = ioremap(ioport, iolen);
13
  // the rest of the code
15 // ...

```

Listing 6.2 Example implementation of the `device_probe` subroutine.

Initial memory allocation can also be done within the `device_probe` routine. For this purpose, the `pci_alloc_consistent` Kernel call can be used. The physical address for the allocated memory can then be transferred to the ARM cores in the PCIe device. The address will be used for configuring the address translation logic within the PCIe AXI bridge.

Both of the drivers represent the device to their Linux system as a character device which supports basic file operations. These file operations include open, close, read, write, i/o control and mmap. In the simplest data transfer scenario, the user level application opens the character device and performs reads and writes to the device using usual file I/O operations. From the host side, the driver can redirect the incoming read and write transactions toward pre-known address offsets in the PCIe device. The same holds true for the driver running at the ARM side. The driver receives the data from the user level application running on the ARM host and transfers it to pre-known offset on the memory allocated by the host driver.

When it is required to transfer and process data at a high rate, the simple file I/O operations can no more be efficient. In fact, during file I/O data transfers the data always gets copied from the user space memory to the kernel memory and from there it gets transferred to the hardware. However, it is possible to share the memory which is obtained by the driver with the user level application directly, eliminating the need for copying data from user space to kernel space and vice versa. This is done through mmap file operation. Usage of mmap for sharing the memory space with user level application is again similar for the drivers at both sides. Code Listing 6.3 shows an example implementation of mmap routine for the host driver. The `io_remap_pfn_range` function receives the user virtual address range and also the physical address range of the device. It then maps the provided range of physical addresses to the provided virtual address range by creating a new set of page tables which essentially perform the address translation between these two ranges. For further information regarding character devices and mmap please refer to [4].

```

1 static int chr_device_map
  (struct file * filep, struct vm_area_struct * vma)
3 {
  io_remap_pfn_range (vma,
5   vma->vm_start,
  ioport >> PAGE_SHIFT,
7   vma->vm_end - vma->vm_start,
  vma->vm_page_prot);
9   vma->vm_flags |= (VM_DONTEXPAND | VM_DONTDUMP);
  vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
11  return 0;
  }

```

Listing 6.3 Example implementation of the mmap file operation routine (Linux kernel 3.7).

6.8 Experimental Results and Discussion

The performance of a PCIe hardware accelerator card is directly affected by the bandwidth of data transfers between the card and the host system. Considering the PCIe card, it is also crucial to know how fast this architecture can transfer data between its own main memory and the integrated PCIe block. These parameters directly affect the processing performance of the hardware accelerator.

Through a set of stress tests it is possible to obtain estimates on the performance of PCIe interface. For the ZC706 board, which features a Gen2 X4 PCIe interface, Table 6.2 shows obtained estimates on each of the read and write data paths [11]. In this test the performance is measured while sweeping the size of the DMA packets being transferred between the PCIe card and the host system from 512 bytes to 32 KB. Obtained values are representing the net data transfer rates after omitting all of the overheads related to packet headers, acknowledge packets, flow control, and other similar items.

As we see, the performance of the PCIe link changes significantly with the packet size. In fact, the efficiency of data transfers to a hardware accelerator using PCIe is heavily dependent on the size of data chunks being transferred. If the acceleration task involves processing of big continuous blocks of data, the PCIe link can be used with acceptable bandwidth efficiency. Instead, if the acceleration task requires a large number of small data chunks being transferred between the host system and the PCIe card, the efficiency of the link can be degraded seriously. In these situations, increasing the PCIe link width (e.g. from X4

Table 6.2 Approximate read and write bandwidth to the host for ZC706 Gen2 X4 PCIe interface

Packet size (KB)	0.5	1	2	4	32
PCIe to host (MB/s)	250	612	762	937	975
Host to PCIe (MB/s)	250	525	750	930	970

Table 6.3 Approximate read plus write bandwidth to the DRAM memory residing on the ZYNQ PS from ZYNQ PL through each of HP1 and ACP ports

Packet size (KB)	4	16	64	128	256
HP0	600	1,250	1,600	1,700	1,730
ACP	600	1,250	1,600	1,650	1,600

to X8), improving caching mechanisms, and exploiting several concurrent threads for initiating required transactions can enhance overall performance.

In the second test, we stress the local DRAM memory available on the accelerator card. We obtain estimates on how fast the data can be transferred between the PCIe block and the local DRAM memory. This is important since in our architecture the host system shares the data with the hardware accelerator on this DRAM memory.

Table 6.3 shows the obtained read plus write bandwidth values between the DMA engine residing on the Zynq PL and the DRAM memory connected to the Zynq PS [7]. Similar to the previous test, we sweep over the size of data chunks being transferred between two nodes to report the bandwidth. We perform accesses to the DRAM through each of the HP1 and ACP ports. As we see, with the increase in the size of packets being transferred the bandwidth improves. However, the increase in bandwidth gets saturated at around 1,700 MB/s. This total bandwidth is almost evenly divided between read and writes.

For the ACP, accesses practically end up the caches of the CPU for smaller packet sizes. However, as the packet size grows and more memory is needed, the extra accesses to DRAM reduce the ACP performance. The real advantage of ACP over HP1 can be seen in scenarios where the accelerators on the Zynq PL want to share data with the ARM CPU cores of the Zynq PS. Considering the obtained estimations we conclude that a careful design is needed to be done so that the Zynq PS and DRAM memory subsystem sustain the traffic generated by the Gen2 X4 PCIe block completely. This involves definition of descriptors for DMA transfers as well as utilization of suitable ports on the Zynq PS to access the DRAM.

6.9 Conclusion

In this chapter, we summarize the basics of PCIe as one of the most popular interfaces for communicating with accelerators in heterogeneous systems. One reason is that – in contrast to other interfaces like Ethernet – relatively low latencies can be achieved with PCIe. We introduce the basic architectural setup and protocol features of a PCIe communication infrastructure. We describe that PCIe peripherals do not necessarily need to be located in one single box together with the host CPU, but can also be physically apart and be connected through PCIe cables. This is in particular beneficial for external accelerator units that need to be attached to a host device.

We also introduce the fundamentals of using FPGAs for interfacing to PCIe. We briefly look at the PCIe integrated block and the PCIe AXI bridge IP cores provided by Xilinx that are available through the Xilinx Vivado design suite. We briefly discuss how these components can be utilized to create a PCIe accelerator card based on FPGAs. One important aspect for achieving good performance is the use of DMA engines besides the CPU.

From the software point of view – both on the host and the accelerator device part – we briefly describe the mechanisms for accessing the PCIe infrastructure and highlight different sections of an appropriate Linux kernel level driver created for talking to a PCIe peripheral. As a practical example, we have used a Xilinx ZC706 for demonstrating how a PCIe design can be realized for the Xilinx Zynq All Programmable SoC device on this board. We show estimated bandwidth values for key parts of this architecture.

Significant improvements in the IP cores and software drivers created to ease designing with PCIe have made building PCIe peripherals easier than ever before. As an example, the Xilinx SDAccel [9] environment provides the designers with a complete CPU/Graphics Processor Unit (GPU) like development experience on FPGAs. If a financial hardware accelerator card comes with relaxed transfer bandwidth and latency requirements, the entire design can be realized by hardware or software engineers with a common level of design skills.

Nevertheless, for some applications it is required to push the transfer bandwidths to their possible maximum. In these cases designing for PCIe is a challenging task. It includes the electronics design of the board, design and implementation of the hardware architecture, and also software aspects of the design. In these situations it is usually a wise decision to leave the project in the hand of professionals who have constant experience of designing PCIe peripherals.

References

1. ARM.: AMBA AXI and ACE Protocol Specification, d edition (2011) http://ee427plblabs.groups.et.byu.net/wiki/lib/exe/fetch.php?media=ih0022d_amba_axi_protocol_spec.pdf, last access: 2015-05-07
2. Athavale, A., Christensen, C.: High-Speed Serial I/O Made Simple, A Designer's Guide, with FPGA Applications. Xilinx, San Jose (2005)
3. Cascaval, C., Chatterjee, S., Franke, H., Gildea, K.J., Pattnaik, P.: A taxonomy of accelerator architectures and their programming models. *IBM J. Res. Dev.* **54**(5), 5:1–5:10 (2010)
4. Corbet, J., Rubini, A., Kroah-Hartman, G.: *Linux Device Drivers*, 3rd edn. O'Reilly Media, Beijing/Sebastopol (2005)
5. IBM. The CoreConnect™ Bus Architecture, (1999) [https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB/\\$file/crcon_wp.pdf](https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569910050C0FB/$file/crcon_wp.pdf), last access: 2015-05-07
6. OpenCores. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, b.3 edn. (2002) http://cdn.opencores.org/downloads/wbspec_b3.pdf, last access: 2015-05-07
7. Sadri, M., Weis, C., Wehn, N., Benini, L.: Energy and performance exploration of accelerator coherency port using xilinx zynq. In: *Proceedings of the 10th FPGAworld Conference (FPGAworld '13)*, New York, pp. 5:1–5:8. ACM (2013)

8. Xilinx. High speed serial. <http://www.xilinx.com/products/technology/high-speed-serial.html>
9. Xilinx. Sdaccel development environment. <http://www.xilinx.com/products/design-tools/sd/sdaccel.html>
10. Xilinx. LogiCORE IP Endpoint for PCI Express, User Guide (2010) http://www.xilinx.com/support/documentation/ip_documentation/pci_exp_ep_ug185.pdf, last access: 2015-05-07
11. Xilinx. ZC706 PCIe Targeted Reference Design (UG963) (2013) http://www.xilinx.com/support/documentation/boards_and_kits/zc706/14_7/ug963-zc706-pcie-trd-ug.pdf, last access: 2015-05-07
12. Xilinx. Zynq-7000 All Programmable SoC Technical Reference Manual (UG585) (2013) http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, last access: 2015-05-07