Design and Implementation of a Transparent Memory Encryption and Transformation System

Diplomarbeit aus der Informatik

vorgelegt von

Alexander Würstlein

begonnen am

1.4.2012

fertiggestellt am

26.8.2012

angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
Friedrich-Alexander-Universität Erlangen-Nürnberg

betreut von

Wolfgang Schröder-Preikschat
Michael Gernoth

Erklärung

Hiermit erkläre ich gemäss der Diplomprüfungsordnung, «dass die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt wurde und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen wurde. Des weiteren [...], dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. »

Erlangen, den 26.8.2012

Alexander Würstlein

Abstract

Starting from the hardware-based mitigation of cold-boot attacks by transparently encrypting portions of RAM, a novel approach to supplement computer systems through the application of transparent transformations on memory areas is presented. This work provides a working, proof-of-concept implementation of such a system on an FPGA-based PCI Express card. More general possible transformations and applications such as data integrity and consistency, aggregation and processing or transactional memory are discussed.

Kurzzusammenfassung

Inspiriert von der Idee der hardware-basierten Vorbeugung sogenannter ≪Cold-Boot≫-Angriffe durch transparente Verschlüsselung von Teilen des Arbeitsspeichers wird ein neuartiger Ansatz vorgestellt, um die Funktionalität von Rechnersystemen durch die Anwendung transparenter Transformationen auf Speicherbereiche zu erweitern. Diese Arbeit präsentiert einen funktionsfähigen Prototypen eines solchen Systems basierend auf einer über PCI Express angebundenen FPGA-Karte. Mögliche Transformationen und Anwendungen wie beispielsweise Datenintegrität und -konsistenz, Datensammlung und -verarbeitung oder ≪transactional memory≫ werden erläutert.

Contents

Chapter 1

Introduction

1.1 Motivation

To allow for further development of computer systems and their operating systems, a continued exchange of concepts and ideas between software developers creating new operating systems and hardware developers creating new hardware concepts is necessary. However, very often, this exchange is hindered by the necessarily more abstract nature of the discussions: The implementation of new hardware features is a time-consuming and resource-intensive task compared to the development of software. It is not only necessary for communication between the diverse groups of hardware and software developers to take place, but also the hardware feature in question needs to be implemented to test it and judge its fitness and viability for its intended purpose, which is usually very expensive. Therefore promising ideas will often be discarded prematurely because of lacking expertise and resources for hardware development.

This situation could, at least in part and for some types of hardware, be alleviated by using programmable hardware in the form of ≪Field Programmable Gate Arrays (FPGAs)≫. FPGAs are available in various forms and sizes, but generally within the reach of all but the smallest budgets. Necessitated by the faster development cycles today and the sinking costs of FPGAs, they even appear in larger numbers in some consumer devices. One example of such a consumer device is the popular ≪FritzBox≫ router appliance[1, 2]. Therefore the assumption seems warranted, that through FPGAs, the introduction of new hardware features to support operating systems and applications might be easier.

Yet the amount of work happening in this direction seems relatively low. This is likely due to two factors: First, the still relatively complicated process of developing hardware. Hardware development happens on a very low level of abstraction and can be very tedious because of the relatively long cycles of coding, compiling and testing, lacking tools and the opaque nature of hardware. Second, as there is no large body of previous work, the potential of hardware development seems hardly known.

Thus, the first motivation of this work is to improve on this situation by providing a flexible, high-level abstraction that can be used as a model to demonstrate various possible features that software developers have always wanted, but which hardware has never provided.

The second motivation comes from a problem that is hardware-related in nature, but where all current attempts at a solution are essentially software-based. Cold-boot attacks on data and encryption keys in the RAM of a computer system make use of the fact that RAM holds its contents for a few minutes when sufficiently cooled. This makes it possible to physically extract the RAM modules, plug them into another computer and copy the contained data or encryption keys.

Software-based mitigations for these attacks like ≪TRESOR≫[3] or others[4] hide encryption keys or delete them when an unusual condition is detected or during normal operation, incurring a, sometimes severe, performance penalty. Such mechanisms can be made transparent to application software and even to the operating system via a hypervisor[5]. When looking at the quite elaborate tricks like hiding keys in the processor's debug registers or cache, the question rises: why is normal hardware unable to provide transparent memory encryption and facilities to store the key in use.

If the encryption and decryption were to happen transparently in specialised hardware, the problem —that key extraction is simply the physical extraction of a memory module— would be turned into the hopefully much harder problem of extracting the key from a register of an uncooperating FPGA, that does not just allow an attacker to read register contents. The transparency of the encryption and decryption operation would also simplify the implementation on the operating system side and could improve performance of the whole setup.

1.2 Scope of this Work

Considering both of these motivations, I intend to create an abstraction layer or module that is capable of transparently encrypting and decrypting portions of memory. The module or abstraction layer should be hardware-based and therefore transparent even to the operating system, with the reasonable restriction that of course some initalisation and configuration of the hardware might be necessary.

The module should also be capable of more generally supporting arbitrary hardware-defined mappings of memory, where encryption and decryption is just one possible mapping. Others should be definable by exchanging the mapping application. This might enable applications such as consistency checks on the stored data, error correction, transactional semantics or advanced access controls on memory areas. A schematic overview of the transformation process is shown in figure 1.1.

As a hardware platform, an FPGA on a PCI Express card, namely a ≪Xilinx Spartan-6≫ on an ≪Enterpoint RaggedStone2≫ development board will be used. PCI Express seems the logical choice given its perfomance and ubiquity as well as the future prospect of technologies like Thunderbolt being based on PCI Express. The Xilinx Spartan-6 FPGA is a viable choice because the necessary physical interface for PCI Express as well as a freely usable IP core for the PCI Express interface are available.

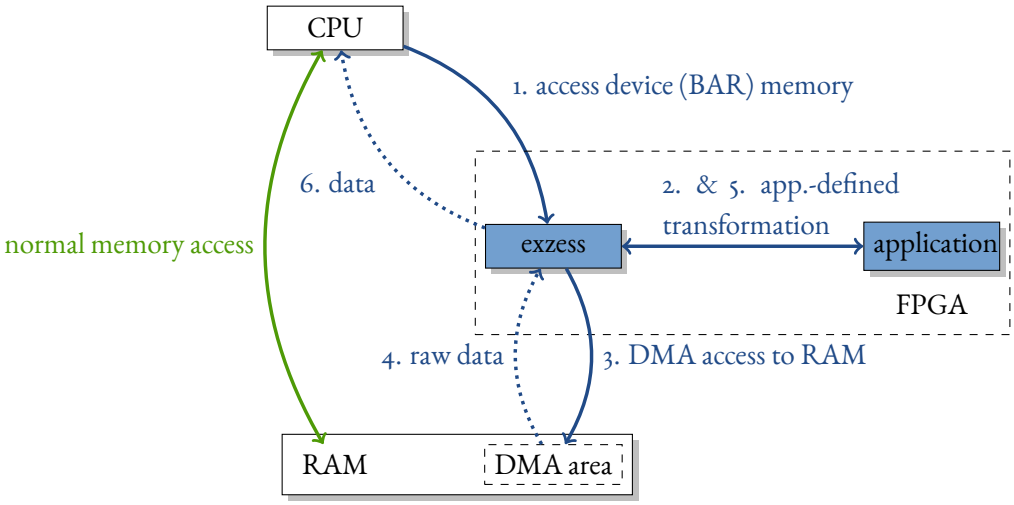


Figure 1.1: Schematic view of the ≪exzess≫ abstraction and application to be implemented. Shown is a reading or writing memory access to a memory area assigned to the FPGA device from the CPU (step 1). The exzess module communicates with the application (step 2), which transforms the original request by transforming the request address and/or request data. The exzess module will execute this transformed request on its assigned DMA area (step 3). Optionally, on a reading access, the raw data read from the DMA area will be returned to the exzess module (step 4), which will again let the application perform its transformation (step 5). The read data will then be returned to the CPU (step 6). Normal memory accesses (green) to addresses not belonging to the device memory area (or BAR area) of the FPGA will not be affected.

The RaggedStone2 card is a platform providing the aforementioned FPGA and PCI Express interface with the necessary development infrastructure like JTAG interface and flash memory.

This diploma thesis will first introduce the reader to the important technologies that are necessary to understand the implementation. First the various layers of the PCI Express protocol and important aspects of memory access in PCI Express systems will be discussed, followed by an introduction to FPGAs and hardware development on and using FPGAs. After an introduction of the relevant IP cores and the Wishbone on-chip bus system, the implementation will be explained in detail. Starting from the hardware the implementation will run on, the FPGA abstraction called ≪exzess≫ will be introduced as well as implemented example applications which make use of the ≪exzess≫ abstraction. The operating system interface and userspace infrastracture to access, test and benchmark the FPGA will also be described. The work will end with a discussion of possible further applications besides those implemented and already described as well as an evaluation of the implemented applications' performance and viability, finally completed by an outlook and conclusion.

1.3 Related Work

Related work exists regarding various aspects of this work, but no similar combination of these various aspects is known to me. Therefore I will discuss the related work separately for each aspect.

As mentioned before in section 1.1, there is a large body of work towards software-based mitigation of cold-boot attacks[3, 5, 4]. Usual lines of work in this area aim to hide the key material in cache[4] or CPU registers[3, 5].

Other mitigation strategies[6] suggest the use of hard-disk drives with built-in encryption functions. In some limited scenarios, where the added expense of replacing hard-disks is warranted and where problems like multi-user-access do not exist, such self-encrypting hardware is an option. Yet this would only solve the cold-boot problem in the context of whole-disk-encryption, the danger to other kinds of key material like signing keys through the same attack can not be mitigated by this.

Similarly, CPU register based approaches like TRESOR[3] are limited to very small amounts of key material. Others that hide keys in cache[4] fare better in this regard, as cache sizes range from kilobytes to megabytes. However, excessive performance penalties prevent the use of cache-based techniques at normal system run-time, therefore the keys are usually only moved into the cache when a suspend or sleep mode is initiated.

The hardware-based encryption method presented in this work provides improvements over the mentioned approaches. First, no performance penalty for the system is incurred except when accessing the key material, and this penalty could be lessened by the use of caching (which this work does not implement). The amount of storable material in the encrypted area is only limited by the available amount of RAM and address space. In extreme cases, some variant of ≪almost-whole-RAM≫ encryption might be viable. Furthermore, because the encryption is transparent to both applications and the operating system, except for a device driver for initialisation and assignment of the encrypted memory resources, virtually any kind of cryptographic key material or important data is protectable.

On the subject of providing memory mapping transformations as described in the introduction, the literature is even less yielding. While the use of DMA is certainly widespread in the industry[7, 8, 9] (e.g. to accelerate computations of various kinds and provide easy and fast data access to the accelerating FPGA) and access to device memory or device register space is even more widespread, no combination of the kind described in this work could be found.

Similarly, as described in the same publications[8, 9], the use of FPGAs as coprocessors for mostly number-crunching applications is well-known. To the best of my knowledge, no work has been published on the use of FPGAs as coprocessors in other contexts like the one described in this work.

Consequently, this work provides a novel approach —transparent memory transformations— that enables various new applications.

Chapter 2

PCI Express

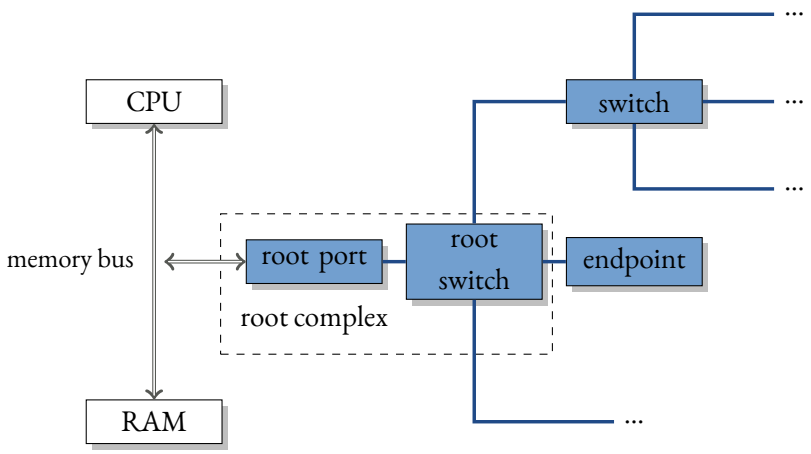


Figure 2.1: Architecture of a PCI Express system

Modern personal computer systems generally use ≪PCI Express≫[10, 11] as bus system to connect onboard and extension-card-type peripherals. The PCI Express bus system was specified in the ≪PCI Express Base Specification≫[10][1]. PCI Express was developed to replace the older ≪Peripheral Connect Interface (PCI)≫ family of busses and protocols. While the earlier technologies ≪PCI≫, ≪PCI-X≫ and ≪PCI-64≫ were parallel busses with a shared, arbitrated medium, PCI *Express* changed this to a tree topology with serial point-to-point links between the nodes of the tree.

Some of the terminology that comes with PCI Express is somewhat unusual, both in the context of communication protocols and bus systems. For example, what is called a ≪Transaction Layer Packet (TLP)≫ in PCI Express terminology would be called ≪frame≫ in Ethernet and other network systems.

[1] At the time of writing the PCI Express base specification was available in revision 3.0.

To be consistent with existing documentation on PCI Express and hardware used for this work, the PCI Express terminology will be used.

This work and the following discussion will be based on the 1.0 and 1.1 revisions of the PCI Express Base Specification[10], since available hardware and IP cores are based on revision 1.1 and available literature are similarly limited. Obtaining the specification is also very costly. The PCI Express Base Specification will usually be refered to as the ≪specification≫ within this chapter.

2.1 Topology and Lower Protocol Layers

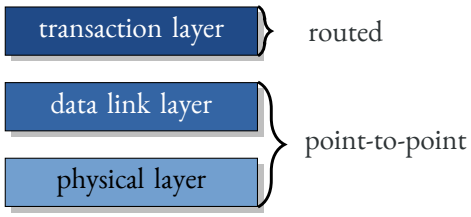


Figure 2.2: PCI Express protocol layers

2.1.1 Structure of a PCI Express system

PCI Express forms a switched, hierarchical, tree-shaped network. Inner leaves of this tree are called ≪switches≫, outer nodes are called ≪endpoint devices≫, corresponding to the respective functions inside the network. Examples of endpoint devices are plug-in cards, such as a network interface card, or soldered-on chips on the mainboard like an integrated graphics processor. Each switch has at least one ≪upstream≫ port and one or more ≪downstream≫ ports. These directions denote whether a port is a link to a higher or lower level within the tree hierarchy. The root node, called ≪root complex≫, is formed by the PCI Express ≪root switch≫ and ≪root port≫. The root switch and port are usually combined in one device as part of the mainboard chipset of a computer system.

The root port has the special role of being the interface between the PCI Express bus, the CPU and the main memory of the computer system. During the boot process when the connected peripherals of the computer system are discovered, the PCI Express root port is used as starting point for the subsequent discovery of deeper levels of the switch and device hierarchy. Configuration requests and other ≪housekeeping tasks≫ are performed through the root port.

Accesses to main memory —called ≪Direct Memory Access (DMA)≫— from PCI Express devices are processed by the root port. In the other direction, accesses from the CPU to device memory are also issued via the root port. In both cases, the root port acts as a gateway between the CPU and RAM on one side and the PCI Express devices on the other side.

2.1.2 Physical Layer

The physical layer of PCI Express was designed with the primary requirements of speed and flexibility in mind. Many older bus systems like the predecessor to PCI Express, PCI, were based on parallel links. In a parallel bus, there is a larger number —e.g. 32 for PCI— of signal lines carrying data. For error-free transmission of data it is essential that on all data lines, all bits that belong to one data word arrive within the same clock cycle of the bus clock.

If the length s of those data lines differs by more than $\Delta s \gtrsim \tau \cdot v_g$ where τ is the length of a clock cycle and v_g is the speed of signal propagation, this condition is violated. While for a clock frequency of 100 MHz allowable values for Δs are of the order 3 mm to 6 mm. For 1 GHz or higher, Δs needs to be 1 mm or lower. Because these constraints are hard to meet when designing circuit boards, a reduction in the number of signal lines is necessary with higher speeds.

Therefore, like many current bus and communication technologies such as SAS/SATA or USB, PCI Express uses serial data paths. A PCI Express ≪port≫ or ≪link≫ may consist of one or more ≪lanes≫ (up to 32) with independent clocks. This allows high clock frequencies in combination with increased capacity by using multiple parallel data pathways if neccessary. The number of lanes in a port or link is given in a notation prefixed with ≪x≫, so ≪x16≫ means that a port or link has 16 lanes and therefore 16 times the theoretical capacity of a ≪x1≫ link or port.

The relation between ports and links is simply that each device has a number of ports with a given maximum number of lanes. When connecting two ports, a link is formed by negotiating connection parameters like available lanes, etc.

Each lane consists of two data transmission lines, using differential signaling. Data is encoded using the ≪8b10b≫ encoding, which allows for the clock signal to be generated from the data lines. Additional symbols in the 8b10b encoding, which are not needed for data transmission are used as start and end symbols for transaction layer and data link layer packets as well as for link training and transmission error signalling.

2.2 Data Link Layer Protocol

The ≪data link layer protocol (DLLP)≫ defines the communication between the two sides of a single PCI Express link, e.g. an endpoint and the switch it is connected to. The DLLP is responsible for acknowledging the receipt of certain TLPs, flow control, link error checking and power management functions.

2.3 Transaction Layer Protocol

An important part of this work consists of implementing the handling of TLPs, therefore the following discussion of the transaction layer will be more extensive than the previous brief overview of the

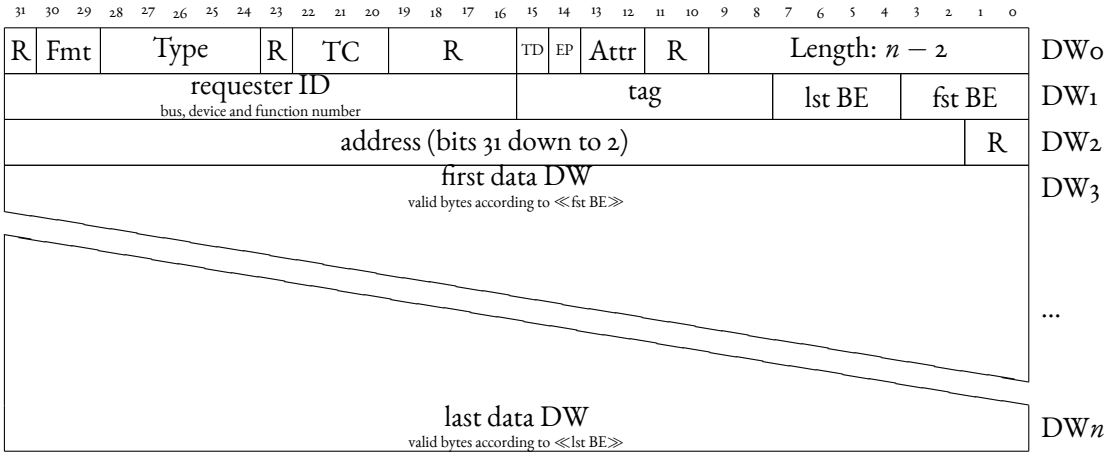


Figure 2.3: PCI Express MWr TLP, created after [10]. Format (labelled ≪Fmt≫) and type define the length of the TLP header, the TLP type and whether the TLP carries data. Requester ID and tag form the transaction ID through which errors can be relayed back to the requester. The address is always specified as 32 bit-aligned, which is why the last two address bits are reserved. All reserved fields (marked ≪R≫) must be set to 0. ≪Length≫ specifies the data length in DWs which is 3 DW less than the total length of the TLP.

physical and data link layer. Other than the previously described physical and data link layer protocols, which describe behaviour on a single link between two ports, the transaction layer protocol describes the routable topmost layer of the PCI Express protocol. This means that a TLP will travel across a number of links from one sending endpoint to one or more receiving endpoints. As the name suggests, a sequence of one or more exchanged TLPs is used to implement logical transactions like e.g. ≪write 0x123456ab to address 0x08154711≫.

2.3.1 Posted and Non-Posted Transactions

As suggested by the preceding paragraph, PCI Express transactions are conducted via the exchange of packets. Transactions can be one of two types, either posted or non-posted. Generally all transactions are initiated by a request. The initiating party sending the request is called the ≪requester≫.

Posted transactions usually comprise only one packet and are finished once this packet is successfully received. A posted request requires no answering packet, except if an error occurs. An example would be a ≪memory write≫ transaction, with the memory write request containing all necessary address and data information to execute the transaction. An answer is unnecessary, because success is assumed if no error is explicitly signalled.

Correspondingly, non-posted transactions are transactions which require an answer in both, successful and unsuccessful, cases. The request issued by the requester has to be answered by a so-called ≪completion≫. Accordingly, in all transactions, the second party is called the ≪completer≫ in PCI Express parlance. An example would be a ≪memory read≫ request which must be ≪completed≫ —i.e. answered— with the requested data that was read.

2.3.2 TLP Types and Organisation

TLPs consist of a header sometimes followed by data, depending on their type. The PCI Express specification commonly uses the term ≪doubleword (DW)≫ to denominate the size unit in which TLP sizes are given. A DW is 32 bit —or 4 byte— long, consequently TLP header, data and total sizes are always integer multiples of 4 byte.

Headers may be 3 or 4 DW in size. In all TLP types, the first DW of the header has the same layout, specifying the type of the TLP, the header size, whether the TLP contains data and the data length. Attributes to indicate the traffic class used for prioritizing TLPs in routing, an ≪error poisoned≫ bit to mark a damaged TLP and an attribute to control cache coherency are also present. The interpretation of following header DWs depends on the type of the TLP.

TLP types can be classified by various criteria. The difference between requests and completions has already been discussed in section 2.3.1. Completions are further differentiated into ≪completion with data (CplD)≫ and ≪completion without data (Cpl)≫. The former is used to return data from successful non-posted requests, the latter is usually used to signal error conditions or to complete non-posted transactions which do not return data.

Requests are classified into reads and writes as well as by the target memory or register type. The latter stems from the different semantics of transactions pertaining to configuration and I/O space. The most commonly used, are of course ≪normal≫ memory requests, ≪memory read (MRd)≫ and ≪memory write (MWr)≫. Message TLPs can be used to signal interrupts as well as a number of other conditions. Because they are not important for the scope of this work they will only be mentioned for the sake of completeness and not discussed in depth. As a legacy mechanism, several types of locked transactions are specified. We will also skip further description of such locked type TLPs as the PCI Express specification forbids non-legacy endpoints from supporting them, rendering their discussion unnecessary for this work.

Memory and I/O Requests

Memory and I/O request headers are identically structured due to the fact that they all address a certain area of memory. The header always ends with the address the request refers to. Because PCI Express supports memory requests with 32 bit and 64 bit address widths, the corresponding request headers may be 3 or 4 DW long, respectively. Endpoints are recommended to support 64 bit addressing. In each case, the last or lowermost two address bits are always reserved and set to zero. This stems from

the fact that all requests have to be aligned to 4 byte boundaries. Unaligned or smaller requests are possible by specifying a ≪byte enable≫ bitmask in the second DW of the request header.

To match a request with its corresponding completion, the second header DW also contains a 2 byte requester ID and a 1 byte ≪tag≫. The concatenation of the requester ID and tag forms the transaction ID. The tag is used to uniquely identify the transaction among all open transactions of a requester. This means that one requester can have at most 256 open transactions[2]. The requester ID is composed of bus, device and function number. These numbers correspond in function and form to the PCI-IDs. They are assigned to each device at boot or initalisation time by a configuration process.

Completions

Completions are required to conclude non-posted transactions. Typically, non-posted transactions are all memory transactions that are expected to return data, i.e. memory reads. Also, I/O read and write transactions require completions, even if no data is returned. Errors for posted requests are also signalled by completions.

All completions have the same header layout, regardless of whether they carry data. To identify the corresponding transaction, the transaction ID from the original request is included, consisting of requester ID and tag. In layout and function identical to the requester ID, the ID of the completer is included as well. Three completion status bits signal if the completion is considered successful or if the request could for some reason not be completed, e.g. because the type of request is unsupported or because the completer lacks resources to fulfil the request.

A single request may be answered by multiple completions. This may happen if a memory read requests a larger amount of memory which the completer is not able to provide in one piece. To support such types of completions the header includes a ≪lower address≫ field, indicating the lower address bits of the returned data, and a ≪byte count≫ field. This field indicates the number of remaining bytes that still need to be transmitted for a successful completion of the transaction. It is required that the partial completions arrive in an ordered fashion, so that addresses of the transmitted data portions are always ascending. Completions without data ignore all these data-related fields.

Other TLP types

There are further TLP types to transmit messages of various kinds and to be used at configuration time to set the configuration registers of endpoints. These types are more varied and have a wide range of semantics. An in-depth discussion of them will not be provided here for reasons of brevity. An understanding of them is not necessary for the purposes of this work.

[2] Usually only 32 transactions may be open, using only the lowermost 5 bit of the tag. The number of open transactions for a requester can be extended by various means if necessary, e.g. by using the whole tag field, leading to the mentioned 256 open transactions. For the scope of this work, the standard amount of 32 to 256 is more than enough.

2.3.3 TLP Routing

Depending on the TLP type, there are several ways for a PCI Express switch to decide how to forward a given TLP. TLPs that include a memory address, e.g. memory and I/O reads and writes, are routed by inspecting that address. Completions are routed via their included requester ID back to the requester. Messages provide both these routing methods as well as some special routing methods like ≪broadcast≫, depending on the message type.

2.3.4 Transaction Timing and Ordering Requirements

PCI Express uses a transaction protocol where the request and the corresponding completion or completions ending the transaction are separate packets. This packet-based approach leads to some important questions. The answers to these questions directly influence whether the goals set for this work are achievable.

As was discussed earlier in section 2.3.2, each endpoint is allowed a certain number of open requests, i.e. transactions that are not yet completed. The rather large number of open transactions allowed directly leads to the question of whether all those transactions must be completed in a certain order and whether a completer might issue its own requests having transactions yet to complete.

The specification defines certain ordering requirements to ensure that deadlocks do not occur. Most importantly, completions to read transactions should be processed before processing any new non-posted requests. Also, posted requests should be processed before non-posted requests. This leads to two important conclusions: The opportunity for a deadlock is only present if the completion of a transaction by the completer depends on the completion of a request issued by that completer. This means that the idea of transactions depending on the completion of another transaction was known to the writers of the specification and the specification has been written to allow such behaviour. It is completely acceptable to process transactions in the order they arrive in, only giving precedence to incoming completions over incoming requests to avoid having to track the state of multiple uncompleted transactions internally.

The actual ordering requirements in the specification are a little more elaborate, especially since they also define requirements for configuration, message and I/O TLPs that will not be handled by the implementation of this work.

Knowing that transactions may depend on the completion of other transactions issued by the completer of the outer transaction begs the following question: How long may the completion of a transaction take and how long may the internal data processing on the FPGA take?

The PCI Express specification defines a completion timeout mechanism. This mechanism defines the behaviour of an endpoint or the root complex if an expected completion does not arrive in time and a maximum and minimum acceptable definition of ≪in time≫. The timeout event must not occur in less than 50 μs. It must not take longer than 50 ms to occur and, except if there are strong requirements to the contrary, at least 10 ms are recommended. The timeout event generates an error report by the

requester and the time is tracked by the requester. This means that the complete forwarding delay through the PCI Express bus also adds to the measured transaction time. Each requester may define its own timeouts within the given range, depending on its requirements.

In this work, the FPGA uses a clock frequency of 100 MHz, meaning that one clock cycle is 10 ns long. This means that the absolute upper limits for processing a request are of the order of $5 \cdot 10^3$ to $5 \cdot 10^6$ clock cycles. In reality this will of course be somewhat shorter to allow for buffering and forwarding delay. Given that for example encryption algorithms usually take only in the order of tens[3] of clock cycles in FPGA implementations, there is the realistic expectation that such transformations are possible within the given time. Especially if the root port as the most common or only requester in our case does not implement the strictest possible timeout requirements, the time a transformation takes should be completely negligible compared to the allowed timeout.

Furthermore, nested requests should also be possible within the timeout range, because if the timeout requirements were too strict for such nesting, bridges to other, slower bus systems and deeper hierarchies of switches would quickly run into problems.

2.4 DMA and Device Memory

Between a PCI Express endpoint and the software running on CPU and main memory there are two possible memory domains and directions for memory accesses.

Coming from and initiated by the software on the CPU, device memory can be accessed. Device memory is a memory area assigned to a specific endpoint for which that endpoint handles accesses. Device memory is configured using the so-called ≪base address registers (BARs)≫. In PCI Express terms, device memory accesses are accesses where the completer for a memory transaction is an endpoint and the requester for that transaction is the root port[4].

In the other direction, coming from and initiated by an endpoint, a computer's main memory can be accessed. These accesses can happen without interrupting normal program flow of the software running on the CPU. Such accesses are commonly known as ≪direct memory access≫, or short ≪DMA≫. In PCI Express terms, DMA accesses are memory transactions where the requester is an endpoint and the completer is the root port.

2.4.1 Base Address Registers

PCI and PCI Express devices are configured at boot time or when they are initialised after having been plugged in. In an enumeration process, endpoint IDs are assigned and configuration registers are initialised. In this phase, base address registers are programmed.

[3] As we will later see, the AES128 encryption core used in this work takes 13 cycles plus 2 more for loading input data and keys.

[4] Device memory accesses from and to other devices are possible, but seem to be uncommon.

Base address registers —commonly known by their acronym ≪BARs≫— describe the memory and I/O areas within a PCI or PCI Express endpoint. The operating system or BIOS manipulates configuration registers through ≪configuration write≫ or ≪configuration read≫ transactions. To configure a BAR, the endpoint requests a certain amount of address space with the desired properties like prefetchability[5], address width[6] and type. The BAR type may be either ≪memory≫, meaning that the address space configured by that BAR is a ≪normal memory area≫ which can be written or read by the usual memory transactions. Alternatively, the BAR type may be ≪I/O≫, meaning that only I/O transactions with special semantics are required to read and write values from that area. The use of I/O space is discouraged by the PCI Express specification.

The size and other properties are encoded into a 32 bit or 64 bit data word. That data word is read[7] from the endpoint register by the configuring software and the assigned address space base address is written into the register. PCI configuration space contains 6 BARs, usually named BAR0 through BAR5. Those BARs have a width of 32 bit. 64 bit BARs are formed by concatenating adjacent BARs, meaning there may be at most 3 of 64 bit wide BARs.

The address space assigned through a BAR may be handled by the endpoint in various ways. One possible way is the use of physical memory modules addressed by that BAR address space. The most obvious example for such use is the memory residing on a graphics card, containing a framebuffer, shader programs or textures that the graphics driver can read and write at the mapped address space location. In other cases, the BAR memory area may be mapped to configuration registers with special semantics controlling the internal workings of the endpoint device. Almost every device has at least one such BAR for interaction with driver and application software. In this work a third way will be used where the endpoint does not provide any physical means of storing the data in the BAR memory area. Instead, the memory area is used as a means to map requests onto a DMA main memory area, providing only a ≪transformed view≫ onto that DMA area.

Sometimes the denotation ≪BAR≫ is used not only for the register in configuration space but synonymously with the area of device memory configured using that configuration space register. So one might for example say ≪The packet to be transmitted is written to BAR3≫, meaning that the packet will be written to the device memory area configured in the fourth base address register. Such use should of course be avoided where it may lead to confusion. For this work, I will prefer the more precise terms ≪device register space≫ or ≪device memory space≫.

[5] A memory area is considered prefetchable, if read operations do not have side effects and if multiple write operations may be merged into one larger operation. This allows for read-ahead and delayed-write caching, hence the word ≪prefetchable≫.

[6] Available address widths are 32 bit and 64 bit.

[7] Actually the process of reading the requested values is a little more elaborate, but the details are of no consequence for this work.

2.4.2 Direct Memory Access

Configuration of DMA memory is specified in far less detail than accesses to device memory via BARs. Generally, every physical address addressable over a bus system is reachable to a requesting endpoint. Yet there are some exceptions from this general rule of accessibility that need to be discussed in detail.

An ≪IOMMU≫ is a device similar to the ≪memory management unit (MMU)≫ found in all modern desktop and many embedded processor architectures. The MMU of a processor translates between configurable address space layouts by mapping physical and logical memory pages, segments or addresses. This means that by mapping or not mapping a certain page or segment into the logical address space of a process, this process can be allowed to or prevented from accessing this specific page respectively. Furthermore, the types of access to a memory area can usually be defined, that is whether reading, writing or execution is allowed. While an MMU fulfils this role for accesses between CPU and main memory, an IOMMU does the same for accesses from and to devices and device memory. As such, an IOMMU may prevent an endpoint from accessing memory outside its assigned DMA area. Therefore, in order to find a suitable DMA area for a device, the IOMMU, if the system is equiped with one, needs to be configured to allow access to the chosen area. In the other direction, the IOMMU may prevent a software program or another device from accessing device memory not assigned to it.

Because IOMMUs are not available in legacy systems and in many systems on sale today, this means that a PCI Express endpoint may read and write almost all available physical memory without limitations. Even without an IOMMU, there may be areas of memory unavailable to a PCI Express endpoint. Usually this happens when an endpoint only supports 32 bit transactions on a 64 bit system with a sufficient[8] amount of main memory. In that case, there will be physical addresses that are not addressable for the endpoint. For the configuration of DMA that means that the DMA area needs to be located in the part of the physical 64 bit address space that is mapped to the 32 bit address space.

Also, depending on the cache hierarchy of a processor architecture and its configurability and locality with respect to the PCI Express root port, there may be necessary configuration settings such as cache behaviour and constraints such as unusable areas for DMA. An operating system therefore needs to find, configure and assign[9] DMA areas to devices requesting such areas.

After the operating system has chosen and configured a suitable memory area, the way to transmit the information necessary to access that area to the device is device-specific. The easiest and most common way is for the endpoint to provide one or more registers in the device register space. The device driver writes the start address and either length or end address of the DMA area to the appropriate device registers. This practice will also be followed in this work.

[8] ≪Sufficient≫ memory means 2^{32} B = 4 GB, the amount addressable by a 32 bit address space.

[9] Some classes of devices support a mode of operation known as ≪scatter-gather-I/O≫ where no fixed DMA area exists. Examples of such devices are certain network interface cards which can assemble network packets for transmission from multiple memory locations containing only parts of a packet like various layers of headers and the data payload. Those special cases will not be discussed.

2.4.3 Cache Behaviour

There are several mechanisms which can be used to control cache behaviour in PCI Express. They differ depending on whether DMA or device memory is involved.

Caching for DMA memory is configured by the operating system with appropriate flags in the page or segment descriptors of the relevant memory pages. The Linux kernel API provides us with mechanisms to do this, which will be described later on in chapter 4. Because the endpoint needs at least some control over the cache presence and coherence of its DMA memory, the PCI Express specification defines special semantics for a memory read with a length field of zero: Such a zero-length-read is supposed to initiate a cache flush. When the completion to this flush request arrives, the memory area should be in a consistent state.

Memory transactions also contain a ≪no snoop≫ bit. Setting this bit indicates that, according to [10] ≪[h]ardware enforced cache coherency [is] not expected≫ for this transaction. While the PCI Express specification does not specify further what the exact semantics should be, I understand this to mean that e.g. writes with the ≪no snoop≫ bit set would not be immediately visible to the CPU if an old version of that portion of memory were in the CPU cache. Setting the ≪no snoop≫ bit would prevent measures for cache synchronisation from being performed[12].

For device memory the ≪prefetchable≫ option, configured through a bit in the BAR, offers some control over the caching behaviour. If a device memory area is configured to be prefetchable, read or write accesses must not have immediate side effects. Such side effects are typical for BARs refering to device register space, where e.g. writing a certain register address might cause the endpoint to initiate an immediate reset. Therefore a device memory area where such side-effects are implemented should not be set prefetchable. If no side-effects are implemented, prefetchability will improve performance e.g. by allowing the CPU to issue read requests for larger amounts of data than immediately needed and caching the returned data.

Chapter 3

FPGAs

3.1 FPGAs as Programmable Hardware

3.1.1 Definition

≪Field Programmable Gate Arrays (FPGAs)≫[13, 14] are configurable arrays of logic elements. By configuring them, within certain limits of course, arbitrary circuits can be created, re-created and changed. Reconfiguration of an FPGA is possible within a few seconds, meaning that development cycles for FPGA hardware are very much shorter than for ≪ordinary≫ hardware that has to be soldered, etched or fabricated in other time-consuming processes. While the speed and ease of hardware development using FPGAs is still not on par with the development of software, certain applications may justify the effort.

The basic building blocks of FPGAs are ≪lookup-tables (LUTs)≫, ≪multiplexers (MUXes)≫ and a configurable interconnect fabric. Lookup-tables are memory-elements with n outputs and an m-bit selection input. For each of the 2^m select input states, an output state is programmed into the memory element. Upon encountering such an input state, the respective output state is assigned to the output lines. A multiplexer is an element with n selection inputs, 2^n data inputs and one data output. For each different state of the selection inputs, the corresponding data input line is connected to the data output line. Together[1] both of these types of elements can be used to recreate any existing boolean logic function of a certain number of inputs and outputs. These functions are then strung together by the configurable interconnect fabric. Usually this fabric is a grid of connections that can be connected by programmable fuses at the startup of the FPGA. Clock distribution networks and additional clock inputs to the logical elements mentioned above provide the means for implementing clock-synchronous logic.

[1] MUXes and LUTs are very similar in their nature and can be transformed into each other. Therefore it is not really important for the user of an FPGA how the vendor implements the hardware.

FPGAs often provide additional capabilities beyond the implementation of arbitrary boolean logic, for example input and output pins with programmable electrical characteristics, analog elements like ADCs, embedded processor cores or bus interfaces.

3.1.2 Availability

Various hardware platforms are commercially available that include FPGAs as coprocessors[15], on add-on cards[16] or are FPGAs with embedded processor cores. The prices[17] for FPGA chips are currently, for the model used for this work, well below $ 100, for certain models and in larger numbers even below $ 10. Larger models are of course more expensive, but also more capable.

FPGA add-on cards are available in various price-ranges and sizes and for different applications, e.g. as coprocessors for high-performance computing applications or as development boards providing the means to easily develop custom hardware prototypes. The Raggedstone2 board from Enterpoint[16] is one such development board that is used in this work.

3.2 Hardware Description Languages

On the lowermost level, electronics are made of transistors and wires, out of which logic gates, flip-flops and other higher-level structures are formed. In a trend whereby the complexity of larger circuits is managed via higher levels of abstraction, the current state of the art in circuit design utilises ≪hardware description languages (HDLs)≫. Three HDLs, ≪VHSIC hardware description language (VHDL)≫, ≪Verilog≫ and ≪SystemC≫ are currently in widespread use.

3.2.1 VHDL and Verilog

Because VHDL[18, 19] and Verilog[20, 14] have been used in certain parts of this work, this chapter will outline the most important characteristics of those languages.

For the understanding of this work, a knowledge of the syntax of VHDL and Verilog is of course beneficial, but it is not my intention to teach[2] the reader any of those languages. Instead I will describe the terminology, programming model and the semantics that both languages share, as far as they are relevant to the understanding of the implementation. The terminology used will mostly reflect that found in VHDL and software development. The reader will also find all these concepts in Verilog as well, just perhaps under a different name.

Modelling of Time, Sequence and Causality

In most popular software programming languages, time is modelled mostly by the linear sequence of lines of code. In line n it is guaranteed that line $n-1$ has been executed and all changes up to line $n-1$

[2] For an introduction into VHDL or Verilog see e.g. [18] or [14], respectively.

have been performed. Loops and jumps simply reorder this strictly linear sequence of lines by deciding which line gets to be $n + 1$. This linear equivalence of time and lines of code is of course derived from the linear sequence of instructions the CPU executes. Time is understood to progress with the linear sequence of instructions, although there is typically no fixed amount of time per line of code. Line n is just understood to be executed after $n - 1$.

In HDLs, linearity is only available where explicitly introduced by certain language constructs, everything not explicitly linearised is by default executed in parallel. Sequencing can be introduced in two ways: Either by data dependence together with the use of immediate assignments, this is called ≪asynchronous≫; or sequencing is achieved by synchronising the execution of statements or blocks of statements to a clock signal, this is called ≪synchronous≫. A combination of both techniques is common. A block (in VHDL a ≪process≫) is triggered by the rising edge of the clock signal. Inputs to that block are expected to be valid at that point in time. All computations in that block are executed in parallel or sequence depending on their data dependence, yielding results. Those results are expected to be valid no later than at the next rising edge of the clock so they can be used in the same or other processes for further computations.

Entities

Entities are independent modules. Each entity consists of its ≪ports≫ or ≪pinout≫ and its ≪architecture≫ or ≪implementation≫. Input and output ports are often referred to simply as ≪inputs≫ and ≪outputs≫. The most fitting image is of course that of a microchip with a number of pins with certain characteristics and functions that the user of a chip has to worry about when designing a board with it. The producer of the chip has the additional task of providing the implementation: the inner workings of the chip.

In quite the same way, an entity with a given set of ports may be used as a black box, the implementation to be filled in later from a netlist file or a hard-IP part of the FPGA. Alternatively the implementation of the entity may be provided by the programmer, using the entity as a means of encapsulation.

Data Types and Variable Types

Like in software programming languages, HDLs have the basic concept of data types. One may, for example, declare something an ≪integer≫ or ≪float≫[3]. The most common type used in digital circuits is of course the ≪bit≫ and one-dimensional arrays thereof. Although for example the most common VHDL data type ≪`std_logic`≫ used as a bit type is actually an enumerated data type with nine possible values, the seven ≪other≫ values are rarely used. In almost all situations it is sufficient to consider ≪`std_logic`≫ to be one bit.

[3] The actual names of the available data types may differ from these examples depending on the language, its version and the library package in use

≪Variable types≫are a concept that is unknown in the field of software programming. While a data type describes the form of data that is stored or transmitted, the variable type describes whether a piece of data is stored or transmitted. To understand the difference, one needs to look at the two things that can happen to data in a piece of hardware. The data can be stored, in a register or a similar storage element, or it can be transmitted over a wire. As one often simply ≪wires up≫ existing modules, the transmission of data between these modules is essential. Introducing storage elements in between two modules would add delays and possibly change timing behaviour of the circuit so that performance would suffer or necessary timing constraints would not be met, making the circuit non-functional. Therefore control over the introduction or non-introduction of storage elements into a circuit is very important.

Verilog calls its two variable types ≪wire≫ and ≪reg≫ in accordance with the explanation provided above. VHDL deviates from this naming as well as from the semantics. In VHDL, a ≪signal≫ is roughly the equivalent of a wire, a ≪variable≫ is almost the same as a ≪reg≫ in that the usual roles of being a storage element or a transmission element are similar. The main differences lie in the assignment semantics of both languages.

Verilog provides the operators ≪=≫ and ≪<=≫ for ≪immediate≫ or ≪blocking≫ and ≪delayed≫ or ≪non-blocking≫ assignments, respectively. An immediate assignment is visible in the next line of code, similar to the usual assignment semantics in software programming languages. A delayed assignment will be executed at some future time, in simulations at the next simulation time increment, in hardware some time before the next clock cycle.

VHDL also has immediate and delayed assignment operators, ≪:=≫ and ≪<=≫, but those are only allowed to be used with one variable type each. Immediate assignment may only be applied to variables, delayed assignment may only be applied to signals. In other words, the lvalue for ≪:=≫ must be a variable, for ≪<=≫ a signal. Therefore the more common terms for such assignments are variable and signal assignment.

While this may seem like a reduction in functionality compared to Verilog, VHDL signals and variables are not exact equivalents of Verilog's wires and regs. While in Verilog, a reg will always generate a storage element, where a wire will never do so, in VHDL both signal and variable may create either a connection, a storage element or both, only depending on the assignment semantics. This means that to create an output port where a driver holding the signal to be output is necessary, one always needs to create a combination of a wire and reg in Verilog, where the reg is the driver holding the data and the wire is the transmission line. In VHDL, using a signal as output is sufficient, the necessary driver is created automatically.

There are several further small differences between Verilog and VHDL in this area, but those are of lesser importance.

3.2.2 Compound Data Types and Other Niceties

The support for more complicated data types than bits and bit vectors is often lacking in many implementations. Two of these types were used in this work where convenient: integers and record data types. Integers are transformed into arrays of bits by the synthesis tool, with implementations for adders, comparators and similar inserted to support the arithmetic operations used.

Records are used to group and structure input and output ports which would otherwise consist of a large number of single bits and bit-arrays. A common example is a parallel bus system, which defines a large number of data and control signals like ≪ack≫, ≪busy≫, etc. Using a record type for that bus allows a bus connection to be created in one line, where otherwise a large amount of redundant declarations would have been necessary. In addition to the readability and writeability improvement, maintenance of the code is simplified. New signals can be added to a bus by changing a common file where the record type is defined instead of changing every file where that bus type is used.

Given that record types only containing `std_logic` and `std_logic_vector` types are no more than a convenient renaming of their members, there is a surprising number of warnings concering possible problems, but also some endorsement[21]. Consequently, there seem to be only few projects and standards, for example on OpenCores[22] using record types. Because no major problems, only some minor inconveniences, have been observed, this work uses record types extensively.

3.2.3 Implementation of State Machines in HDLs

Most of the implementation in HDLs in this work consists of state machines. The implementation of state machines is a very basic technique for implementing hardware functions. Generally, the definition of a state machine in this context involves four necessary parts: First, the enumeration of possible states, second, the possible transitions between these states, third, the constellations and conditions of inputs under which these transitions take place and fourth, the outputs or more generally the actions happening in each state.

From the view of a software programmer, these parts of the definition provide perfect division lines along which the implementation may be structured. For example the second part might be encapsulated in a decision function taking all inputs and the current state as a parameter and returning the next state. The fourth part, the actions and outputs of each state might be implemented as an execution procedure for each state. One VHDL process might be used to encapsulate the decision function or the execution procedures

Yet the characteristics of HDLs discussed above make this separation of the various concerns impossible: The aforementioned variable or assignment types force a common scope for assignments. If one wishes to avoid this common scope, additional registers and wait-states need to be introduced and data needed for outputs as well as decisions for state transitions need to be duplicated unnecessarily. This unfortunately means that the implementation will use one process per state machine, containing all decisions, inputs and outputs within the same code parts.

3.3 Toolchain

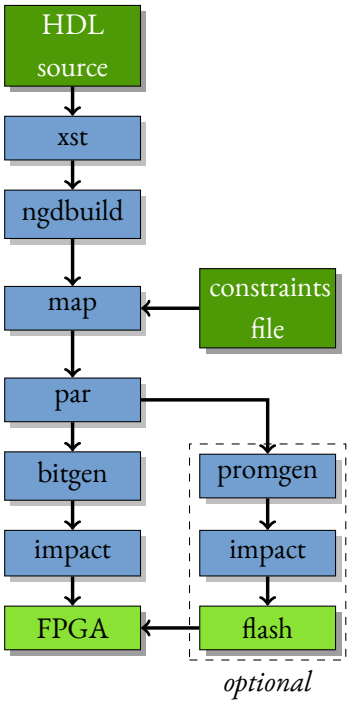


Figure 3.1: Build process used to program the FPGA from HDL source code. The alternative optional path can be used to permanently program a flash memory chip which programs the FPGA on each powerup.

To translate the HDL source code into a bitstream which can be loaded into an FPGA, a multi-stage toolchain is employed. For use with Xilinx FPGAs, Xilinx provides a software suite called ≪ISE≫. While other toolchains are available, ISE has been chosen for being free and naturally most suitable for use with Xilinx hardware.

In a first stage, called ≪synthesis≫ the HDL sourcecode is translated into a so-called ≪netlist≫. The netlist represents the behaviour of the HDL source in terms of interconnected generic components like state machines, memory or logic gates. The command-line version of the synthesis tool is named ≪xst≫.

The second and third stage with the tools ≪ngdbuild≫ and ≪map≫ assign concrete components that are specific to the target FPGA, where the output of xst only contained generic components. High-level generic components like state machines are decomposed into lower-level components. Also, the constraints such as the location of I/O pins are incorporated into the netlist. Constraints are specified

via a special constraints file, but it would also be possible to annotate constraints within the HDL source if necessary.

Fourth, in the place & route stage, the ≪par≫ tool assigns fixed locations on the FPGA to all components which have not already been assigned a location earlier by constraints. Signals are routed between these resources with respect to timing constraints. The place & route stage is usually an iterative process where, starting from an initial non-optimal solution, the placement and routing is incrementally improved until timing constraints are met.

The information is then transformed into a format suitable for loading into the FPGA by the ≪bitgen≫ tool. The resulting bitstream can either be loaded directly into the FPGA by use of the ≪impact≫ programming tool, or the bitstream can be converted into a flash file format. This flash file can then be loaded onto an onboard flash-memory-chip that automatically programs the FPGA on power-up. Programming the flash memory is usually done for the delivery of a finished product. During development the FPGA is loaded directly to conserve time and limit flash write cycles.

3.4 IP Cores

≪IP cores≫, with ≪IP≫ standing for ≪intellectual property≫, is a common denomination for available building blocks a hardware design may incorporate. Often the designation is shortened to just ≪IP≫. For a software developer the usual analogon would be software libraries, which can be linked to a software design and provide the desired functionality. IP cores can be classified into two main categories, ≪soft IP≫ and ≪hard IP≫.

Soft IP consists of HDL source or netlists included in the appropriate place by the toolchain. Its functionality is then provided by use of the usual FPGA primitives, like any other piece of HDL code. Typical examples for soft IP are interconnect busses for slower bus systems that do not require elaborate physical interfaces like Wishbone[23], or implementations of algorithms like matrix operations, video encoders or DSP functions.

Hard IP on the other hand consists of primitives already available on the FPGA, usually because their implementation in terms of the usual FPGA primitives like multiplexers and fuses is impossible or too resource intensive. An example of such hard IP is the high-speed serial interface necessary to support PCI Express on an FPGA: Typical FPGA hardware would not be able to handle the required high clock frequencies in the 2 GHz range. For that purpose, the Xilinx Spartan 6 FPGA used in this work has a builtin serial interface hard IP that can, among other things, be used as a PCI Express physical interface.

3.4.1 ≪Xilinx PCI Express Endpoint Block≫ core

In the implementation, a Xilinx IP core[24] is used to provide a basic PCI Express interface.

The Xilinx PCIe Core provides a 1-lane PCI Express interface compliant to version 1.1 of the PCI Express specification. The core is able to handle the physical interface and the DLLP completely. With

respect to the transaction layer protocol, the core handles all configuration requests by implementing the required PCI-compatible configuration space as well as some extensions and extended capabilities. TLPs other than configuration TLPs are passed to the user-implemented application[4].

Two internal on-chip interfaces are provided, an older variant called ≪TRN≫ as well as a newer one intended to replace TRN, called ≪AXI≫. In this work, the TRN interface was used. While there are differences between these interfaces, they mostly concern details like line polarity and the naming of signals. Line polarity determines whether a high voltage should mean a logical ≪true≫ or ≪false≫. Usually these polarities are called ≪active-high≫ and ≪active-low≫ respectively. Some consider active-low polarity, where an electrical ≪1≫ represents a logical ≪0≫ to be poor style because of lower readability and possible confusion. Standards like ≪Wishbone≫[23] that are used in the implementation make similar arguments, therefore all signals in this work will be implemented as or converted to active-high. With the AXI interface using active-high polarity in almost all signals as well, most of the conversion from TRN to AXI has therefore already been done internally, the remaining differences should be quick to resolve if necessary.

3.4.2 ChipScope core

Additionally, IP cores for the Xilinx ≪ChipScope≫ debugging system have been used during development. ChipScope allows, among other possible functions, to embed a virtual on-chip logic analyzer which can be used to monitor internal states which would have been unavailable or hard and expensive to monitor with external instrumentation. Because ChipScope only consists of an IP core and a software component, its use only requires reprogramming the core instead of obtaining an expensive logic analyzer and attaching external probes after routing all internal signals to output pins of the FPGA.

3.4.3 AES128 core

The AES128 encryption used for the encryption example is an IP core from OpenCores[22] written by Hemanth Satyanarayana. The core has been published as open source under the ≪LGPL v2.1 or later≫ license, is well written and usable with only very minor adaptions.

The documentation included in the core source code states a 13 cycle processing time after loading the key and data registers. This means that within 15 cycles, also accounting for input and output cycles, one block of data can be encrypted. The core itself implements only the trivial ECB block cipher mode, which has been extended as described later on in section 4.6.

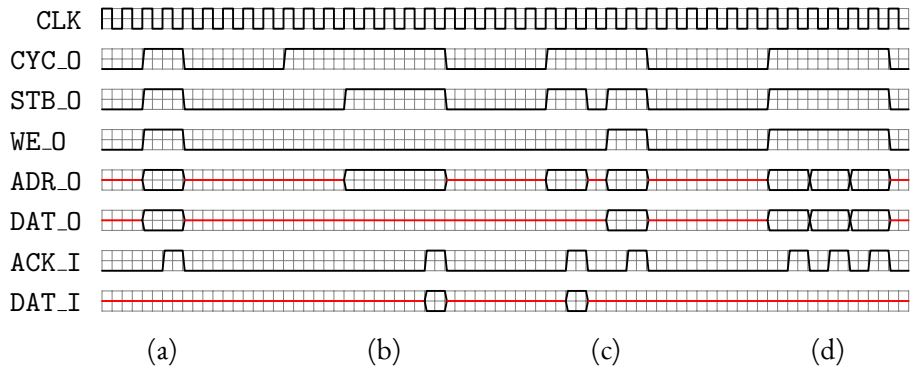


Figure 3.2: Examples of synchronous Wishbone Classic cycles. Signal directions are as seen from the master, meaning the master drives all signals with _O, the slave drives all signals with _I. (a) shows a simple single write cycle, which is completed as fast as possible. (b) shows a read cycle, where the master as well as the slave each take a few cycles of time to output the address the master would like to read and to answer with data from the slave. In (c) a cycle containing two transactions, first one read, the a write is shown. As shown in (d), the fastest possible transmission of data with Wishbone Classic synchronous cycles is one transfer per two clock cycles. Red lines signify undefined states that are ignored by all interfaces.

3.5 The Wishbone Bus

The Wishbone bus[23] is a standard for on-chip connections between IP cores. It is most popular among Open-Source communities like OpenCores[22]. A sigificant part of the free and Open-Source IP cores provided there feature Wishbone-compliant interfaces. So while one reason for its use is the availability of compatible components, the other reason is that the specification is well thought-out and easy to use. A sensible set of signals for the transmission of data with defined timing and semantics is provided.

Wishbone being an on-chip bus system, the specification does not define any physical signal requirements. Any definitions and requirements are based on the assumption that there is a stable clock as well as the means to transmit logical states available. Because these assumptions hold for an FPGA it is of course possible to implement Wishbone-compliant connections on such devices. There is no required topology for Wishbone interconnects, the standard is flexible enough to allow for point-to-point, switched, shared-bus or sequential/pipelined arrangements. In our case simple point-to-point connections will be used, therefore a discussion of the more complicated variants will be skipped.

[4] The terminology ≪application≫ is used throughout the Xilinx documentation and code, usually meaning ≪what the user of a core implements utilizing the core≫. Keeping with this terminology, the parts of the FPGA software that are intended to be replaced or modified by the user are called application or ≪app≫.

Wishbone interfaces can be implemented in one of several modes, depending on the performance and implementation complexity requirements. Only one of these, the synchronous variant of the Wishbone ≪Classic≫ transfer mode will be explained here, as it will be used in the implementation.

All signals in Wishbone are active-high, meaning that an electrical ≪high≫ voltage state is equivalent to a logical ≪true≫ or ≪1≫ state. The direction of each signal is indicated by a suffix, outputs are named with the suffix `_O`, inputs with `_I`.

Interfaces can be of one of two types, ≪master≫ or ≪slave≫. In any system there must be at least one master and one slave interface. A master is capable of initiating a ≪bus cycle≫ by asserting its `CYC_O` output. The arbitration logic of e.g. a switch or shared bus interconnect may use this signal in the arbitration process. The slave has a corresponding `CYC_I` input which, when asserted, causes the slave to evaluate the rest of the inputs of its Wishbone interface.

In a cycle, the master may make requests to the slave by asserting the `STB_O` signal and simultaneously setting the appropriate address, data and write enable bit to `ADR_O`, `DAT_O` and `WE_O`. The data is only necessary if the data is to be written, when reading `DAT_O` from the master is ignored. The master keeps `STB_O` asserted until the slave acknowledges the completion of the request by asserting its `ACK_O`. When the request was to read data, the slave simultaneously outputs the data on its `DAT_O`. The `ACK_O` signal stays asserted for one clock cycle per unit of data transmitted. This means that if the master keeps its `STB_O` asserted after receiving an acknowledgement on its `ACK_I`, this constitutes a new request.

The specification provides other modes and further signals to transmit data faster without wait-states, signalling errors, locking the bus for exclusive access and more. As these are not used in this implementation their discussion will be skipped.

Chapter 4

Implementation

The following sections are arranged in an approximately chronological order. I will describe my progress in the implementation of the overall project as well as the occuring problems and dead ends.

4.1 PCI Express FPGA Card

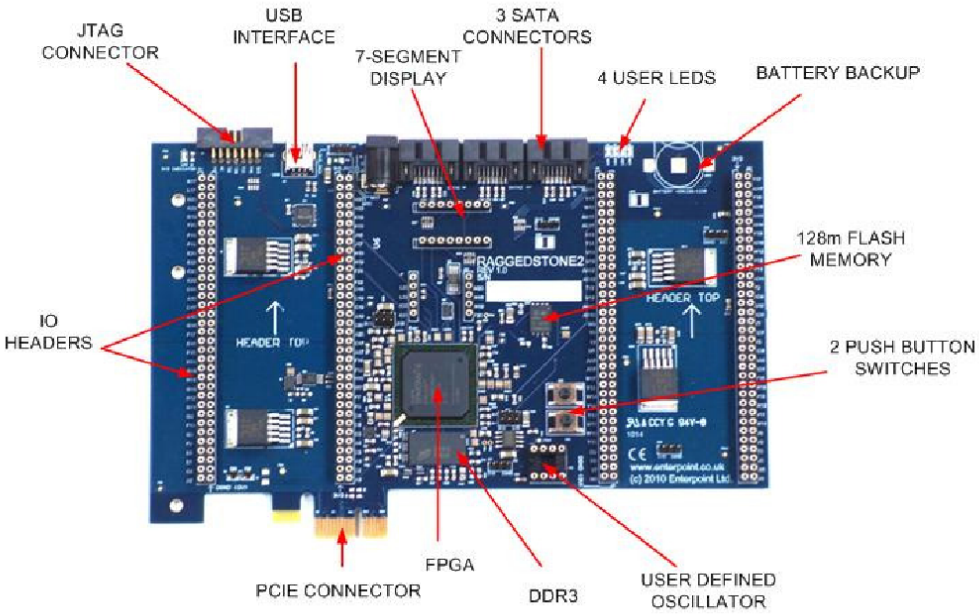


Figure 4.1: Photography of the Enterpoint Raggedstone2 FPGA PCI Express development board. From [16].

The PCI Express FPGA card used for this work is a ≪Raggedstone2≫ board (see figure 4.1) by Enterpoint[16, 25]. The main component of the card is a Xilinx ≪Spartan-6 XC6SLX45T-4FGG484C≫ FPGA. The PCI Express card has, in addition to the standard 1-lane PCI Express connector, a large number of IO pins for extension modules as well as several LEDs and switches. Three SATA and an USB connector, some RAM and a flash memory module are also available. For this work, only the LEDs were used to signal internal states of the FPGA for debugging purposes. Programming of the FPGA and flash memory is done via the JTAG port using a compatible programmer.

4.2 Xilinx PCI Express Endpoint Core

Xilinx provides, through its ≪Core Generator≫ application, two variants of a PCI Express interface core, named ≪Spartan-6 FPGA Integrated Endpoint Block for PCI Express≫[24]. For the sake of brevity, I will abbreviate the name to ≪Xilinx PCIe core≫ or ≪core≫ where appropriate. As mentioned in section 3.4.1, several versions of the core exist, differing in their on-chip bus interface. Also, each new ISE revision seems to ship a new revision of the core.

The core is configurable through the Core Generator application, where details concerning e.g. the PCI configuration space like the PCI-IDs and the configuration of the BARs can be set.

Together with the core, a constraints file specifying relevant timing and pinout constraints as well as an example application implementing device memory are provided. In an application note[26] an example DMA application together with some benchmarks and related tools is also provided.

4.3 First Tests with the Example Application

In a first attempt to familiarise myself with the Xilinx PCIe core and to verify the correct function of the RaggedStone2 card and the computer system in which it was installed I tried to recreate the example mentioned in the RaggedStone2 documentation[25]. First attempts were unsuccessful due to a very strange behaviour of the computer system containing the card: no PCI Express devices where recognised in the Fujitsu-Siemens Esprimo P5915 system used, no matter which kind of PCI Express card was plugged in. The ≪`lspci`≫ tool as well as the linux kernel log showed no sign of any PCI Express devices being present, besides those that are part of the mainboard chipset. I can only speculate about the reasons for this behaviour, the most plausible one being that the single PCI Express slot in the system is intentionally limited to the use with a DVI adaptor card for the on-board graphics chip.

Using a newer and more capable Fujitsu-Siemens Celsius W360 system alleviated this problem, after loading the RaggedStone2 card with the appropriate software, a device with the PCI-ID configured in the Core Generator was found by `lspci`.

As a next step, the example applications described in the RaggedStone2 documentation and the Xilinx DMA example were tried. Once the card's presence was successfully recognised, the next problem was the absence of sensible data in `lspci -vvv`. An unassigned pin or a wrong polarity for the reset

signal of the card as well as a possibly broken version of the PCI Express core in ISE version 12.1 were presumably responsible for these problems. After fixing the aforementioned problems and changing the ISE version to 13.1, configuration space data was output in `lspci`. All further implementation was done based on the ISE 13.1 version of the core and the other tools.

While all my attempts to make the example application from the RaggedStone2 documentation functional have failed, the Xilinx DMA example in [26] worked with a few adaptions. The ≪`xbmd.ko`≫ kernel module contained therein had to be ported to newer kernels and to 32 bit compatibility system calls in 64 bit kernels. With these adaptions the userspace application also included in the Xilinx DMA example code could be successfully used.

Taking the working example application as a basis, several tests in order to verify the function of various parts of the application were done. Another purpose was to familiarise myself with the functioning of the application and PCI Express transactions in general.

To avoid working with the userspace application and kernel module from the Xilinx example application, I first started to implement a bespoke kernel module named ≪`exzess.ko`≫ for the FPGA card. Using that kernel module, writing and reading the device memory of the card as well as allocating DMA memory and having the card write to it was significantly easier. These functions could also be performed by the Xilinx set of application and kernel module, but due to the rather elaborate structure and larger extent of the code, the necessary adaptions would have taken longer.

One specific linux kernel feature, unused by the Xilinx application, simplified matters immensely: device memory can be accessed by files representing the respective device memory areas under the `/sys/bus/pci/devices/` hierarchy. Any userspace application with root priviledges can access the files named `resource0` through `resource5`, e.g. using the `mmap` syscall, thereby mapping the device memory area into the application's address space. Changing the userspace application or just using commonly available tools like `dd` or `hexdump` is far easier than frequently having to reload the kernel module.

The Xilinx FPGA application was modified in several ways: First to check if reading and writing DMA memory and device memory worked as expected, e.g. by setting or changing memory contents. After initial success with these tests, the implementation of similar nested transactions like the ones shown in figure 4.3 was tried. However, given the structure of the Xilinx example application, this was a much more complicated task. Predominantly because the example application was designed to only support benchmarking DMA accesses. To implement these benchmark accesses, some device memory registers were implemented, holding settings like the DMA base address, the write and read mode and size, the byte pattern to be written or read, the read and write count and a start trigger bit. After the start trigger bit was set, the FPGA started to first write, then read the DMA area in a loop with increasing addresses starting from the base address. After finishing the write-phase, the following read-phase compared the pattern read from memory with the previously written pattern. In the userspace application, the quotient of the bytes written or read and the elapsed time was calculated and shown as the performance value. In this setup, possible reactions to incoming TLPs were limited: Incoming com-

pletions to memory reads were compared against the pattern previously written and then discarded, the data was not intended to be passed on within the FPGA application. The generation of outgoing completions and reactions to writes was limited to the registers in the device memory. Therefore, the creation of outgoing TLPs and the handling of incoming TLPs was, adapted to the requirements, quite simple: The receiving and transmitting parts of the TLP handling were completely independent, except for one ≪completion needed≫ bit that reacted to incoming reads to the BAR area by generating a completion. In all other cases, incoming and outgoing TLPs were totally independent of each other, meaning that their handling never necessitated the access to any shared state.

This situation was not easy to change into the required architecture for a more complex and stateful reaction to incoming TLPs like in figure 4.3. To verify that nested transactions were possible, a trivial implementation was successfully done based on the Xilinx code. Yet after this success it had become quite clear than a bespoke implementation of the TLP handling and a generalised application interface would be required.

4.4 Bespoke Implementation of TLP Generation and Application Interface

The principal architecture of the new implementation was planned as follows: Starting from the interface the Xilinx PCI Express core provides, modules should be added that provide higher levels of abstraction to the next layer module, until the application module is provided with a simple interface that mainly consists of specifying an address, data and a read or write bit. The module layout that has been chosen for this can be seen in figure 4.2. The following paragraphs will present the function of the modules I have implemented from bottom to top.

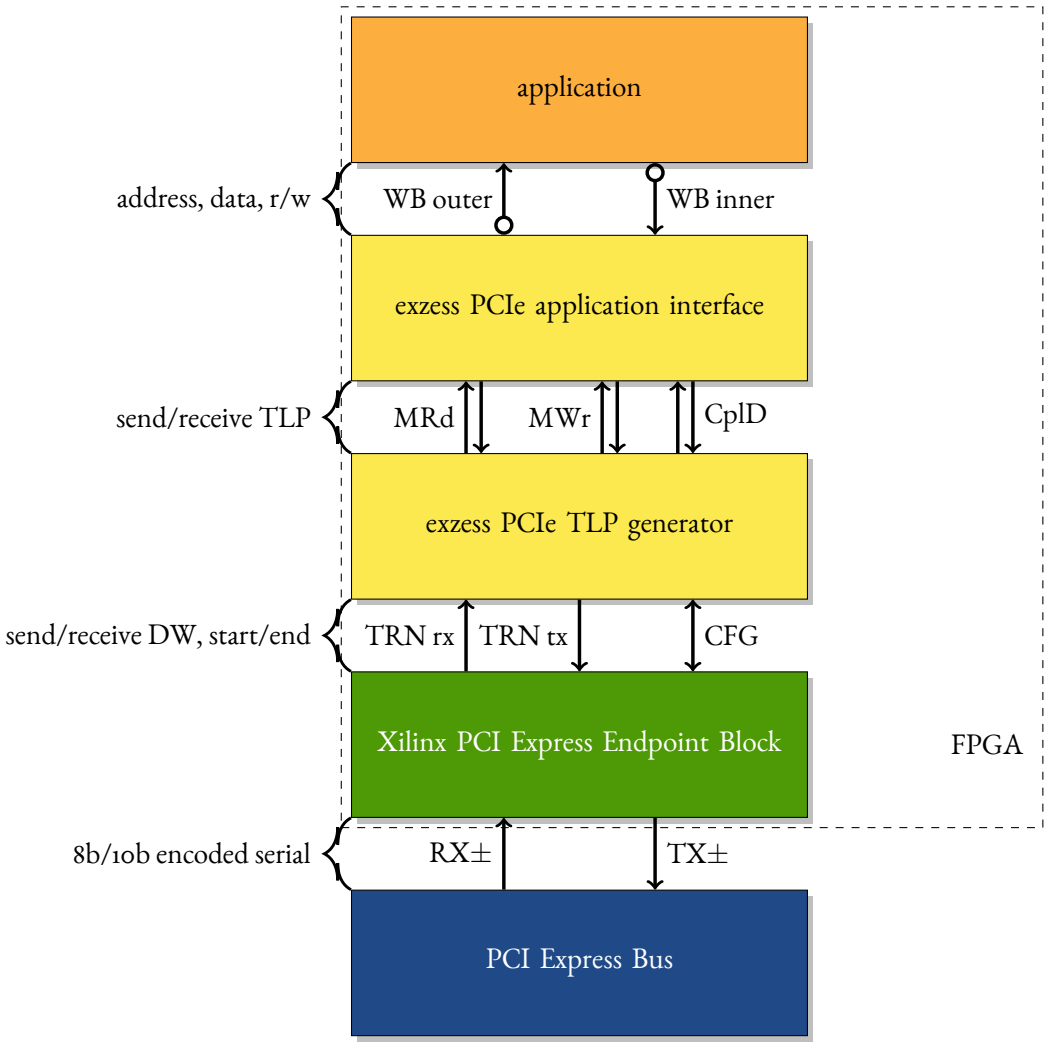


Figure 4.2: Block diagram of the software on the FPGA. Arrow directions specify the flow of data, control signals like e.g. ≪busy≫ or ≪ack≫ usually flowing against the direction shown by the arrows are not drawn for clarity. Wishbone (WB) connections are shown as with their initiating, master sides marked with ≪—○≫ and their slave sides marked with ≪—→≫. Shown on the left are the principal elements of communications between the respective modules.

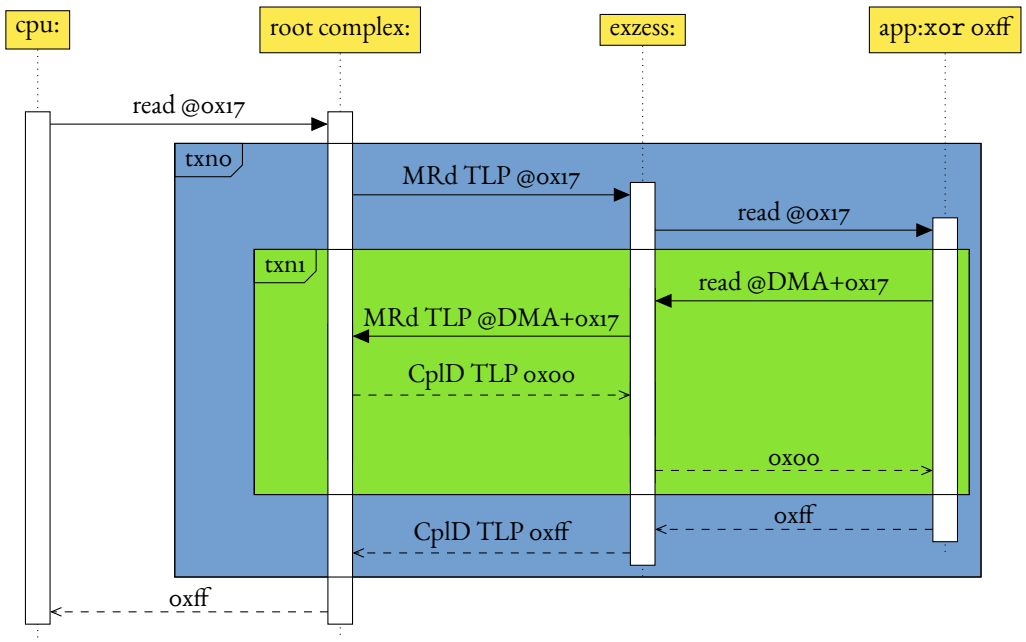


Figure 4.3: Example of a memory read transaction with the XOR application. The CPU requests to read memory at a fictional address ox17, which is forwarded to the application via a PCI Express MRd and the Wishbone application interface. The application calculates the address in the DMA memory area and issues its own request via its Wishbone master interface. The data read from that location (ox00) is forwarded via a CplD TLP and Wishbone back to the application, which applies its transformation, in this example an XOR, to the data. The original, still open request is then answered by the application with the transformed data (oxff), which is then forwarded back to the CPU. Marked in blue is the scope of the original transaction (txn0) requested by the root complex on behalf of the CPU. Marked in green is the scope of the secondary, inner transaction (txn1) issued by the exzess module on behalf of the application.

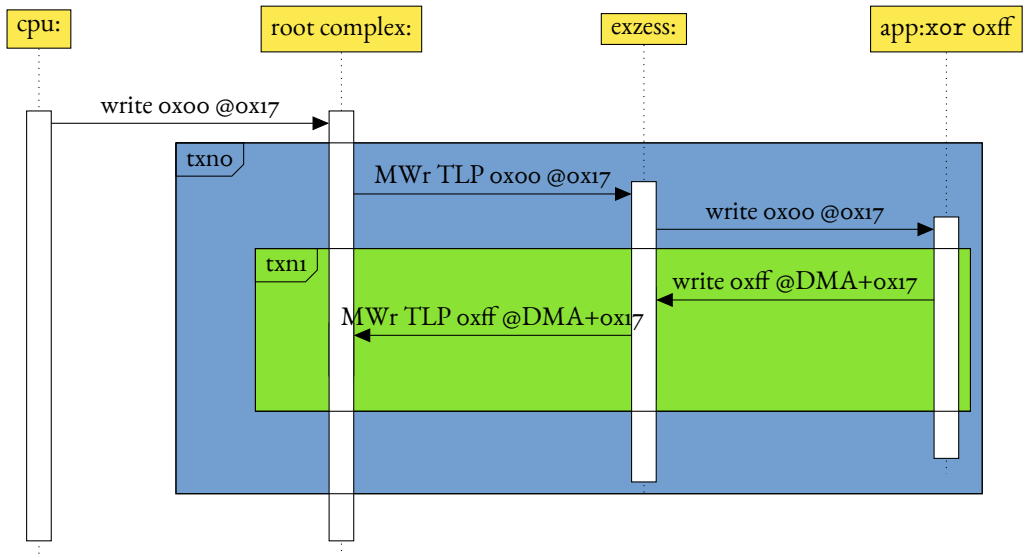


Figure 4.4: Example of a memory write transaction with the XOR application. The CPU requests to write a value of 0x00 to a fictional memory location 0x17. The request is forwarded to the application, which transforms the request data through XOR into 0xff and calculates an address by use of the DMA base address and the request address. With both the transformed data and address, the application issues a write request, which the exzess module forwards via PCI Express to the root complex and finally to the RAM.

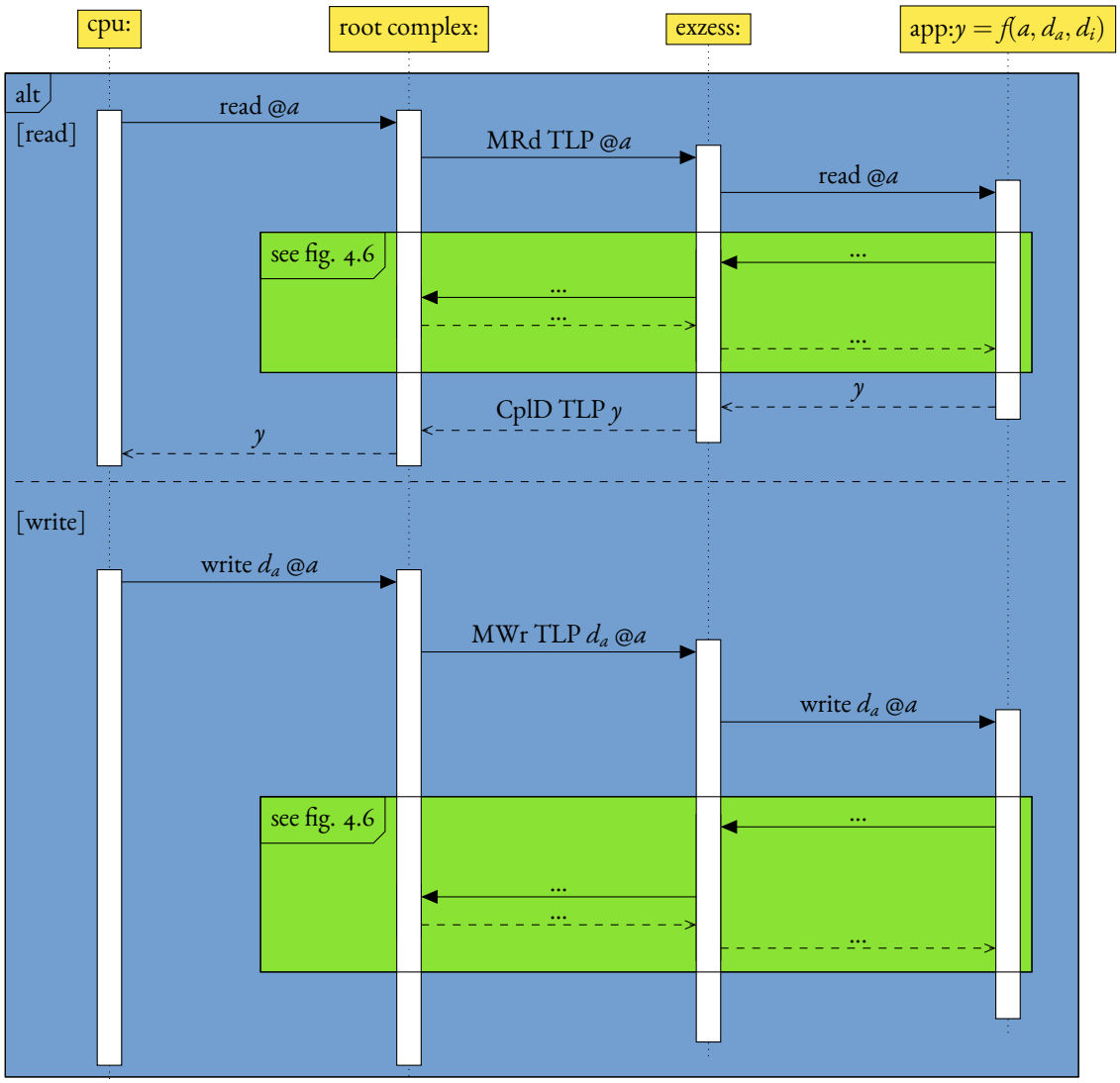


Figure 4.5: General memory read and write transactions which the implementation is supposed to handle. Data d_a at address a is requested to be read or written by the CPU. In both cases, requests are forwarded to the application, which, to handle this ≪outer≫ transaction (blue) may issue ≪inner≫ transactions (green), which are detailed in figure 4.6. In case of an outer read transaction a return value is expected. The application calculates this return value y via its programmed function $f(a, d_a, d_i)$ from the original request and the results of the inner transactions d_i.

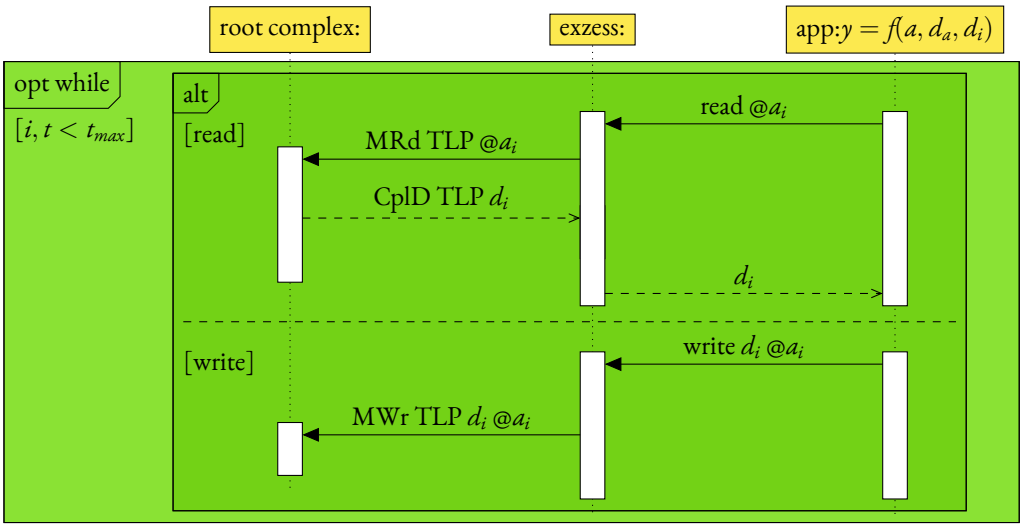


Figure 4.6: General memory read and write transactions which the application may issue while handling an ≪outer≫ transaction as described in figure 4.5. Given the original request address a and data d_a (if applicable), the application calculates addresses a_i and possibly portions of data d_i. The address a_i (and possibly data d_i) are then used to issue write or read requests, which the exzess module forwards via PCI Express TLPs to RAM. In case of read transactions, a returned data value d_i is returned via a CplD. This process may be repeated if necessary, limited by the timeout of the outer transaction. If the outer transaction requires a return value, all values read d_i may be used in its calculation. An outer write transaction will usually be completed by issuing at least one inner write transaction as described above.

4.4.1 TLP Generator

Two state machines should handle the reception and transmission of TLPs for the receiving case, as shown in figure 4.7. The transmitting state machine diagram is almost identical and has been omitted for brevity. The somewhat larger number of states originates from the protocol the Xilinx PCI Express core uses: All TLPs are divided into 32 bit portions, also called ≪doubleword≫ or short ≪DW≫[1]. Start and stop signals as well as a ≪data available≫ signal qualify the incoming or outgoing data bits. Transmitting a TLP therefore consists of outputting the first 32 bit of data together with the data-valid and start-of-frame signal. In the next clock cycle the next DW is transmitted, until the last DW, which terminates the frame with the simultaneous assertion of the end-of-frame signal. Receiving a TLP works in the same way, except the directions are of course reversed.

For the layout of the state machines this means that there has to be a different state for each header DW, as they need to be (dis-)assembled from different inputs. Also, the headers differ in length and content for each TLP type. Lastly, there needs to be a data-state which receives or transmits the expected number of data DWs. Therefore the number of states is the number of TLP types to be handled, times the number of header DWs for each given TLP type plus, if the TLP type expects data after the header, a data-state. Additionally, a wait-state has been introduced for more stable timing and easier synchronisation with other modules. In contrast to the high number of states, the number of transitions is low. The reason for this is, after reading the TLP type from the first DW, there is only one transition from each state to the next one, forming a linear sequence until the last DW of that particular TLP was received.

This large number of states would create redundancy in the code by decoding and encoding similar or identical portions of a header in different TLP types, e.g. the first DW of each TLP header is identical in all TLP types. To avoid having to copy and paste large amounts of code, a library of functions and data types is used which assembles and disassembles TLPs from and into VHDL structures. Those structures are then also used to transmit the TLPs to other modules. The module containing the receive and transmit state machines is called ≪exzess_pcie_tlpgen≫.

[1] There is also the term ≪quadword≫ or ≪QW≫ for portions of 8 byte. The example implementation described in [26] mixed the terms ≪DW≫ and ≪QW≫ up when naming states in its state machine, which lead to the same mixup in my later implementation and description thereof. All of those mixups should be cleared up, but I would of course like to advise the reader not to make the same mistake.

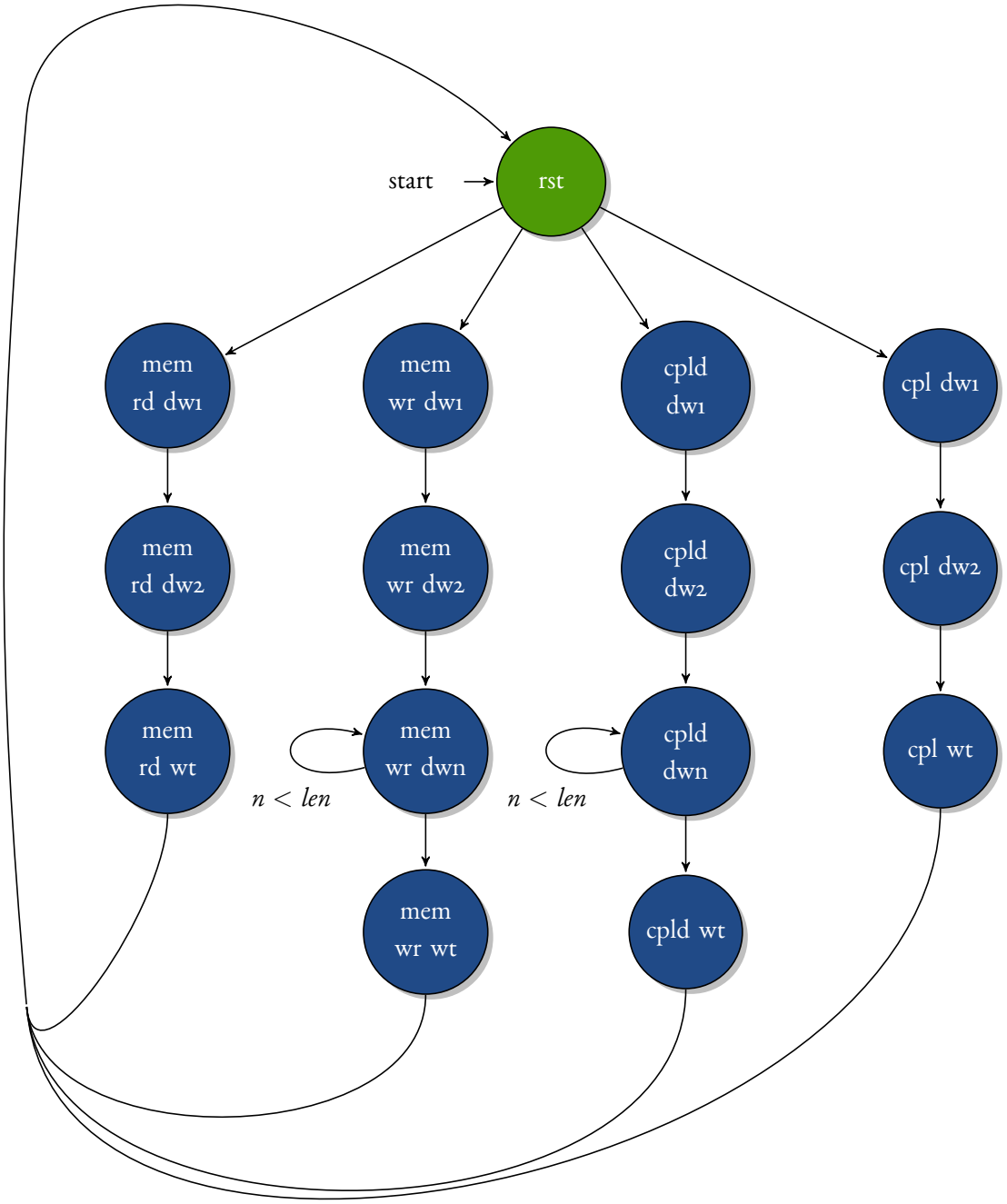


Figure 4.7: TLP receive state machine in the TLP generator module. Reset, initial and idle state is the ≪rst≫ state, marked in green. The TLP types (blue columns from left to right) are memory reads and writes, completions with data and without data. Rows from top to bottom are the sequence of DWs received for each TLP type. The ≪rst≫ state decodes the zeroth DW of each TLP and from that decides on the next state. All other transitions are taken if the next DW is available. The self-loops when receiving memory write or completion with data TLPs receive further data after the header of the corresponding TLP has been received until the total length specified in the header is reached. In the current implementation, these loops are not available.

4.4.2 *Application Interface*

No direct communication takes place between the receiving and transmitting state machine. All logic is handled by the later modules in the architecture, as the relations between those states are to be defined mostly by the application modules, and thus cannot be expressed simply by relations like ≪instantly answer each read with a completion≫. To communicate with those later modules, the interface provided has inputs or outputs for each type of TLP to be received or transmitted. This means that in later modules, the creation and reception of a TLP can be done within one clock cycle instead of several, at the expense of routing resources within the FPGA. Directly attached to `exzess_pcie_tlpgen` is the module containing the application interface as well as all control registers. In this case ≪application≫ —as mentioned earlier— denotes the part of the FPGA code that a user of this project is supposed to modify or replace. This application interface module is called ≪exzess_pcie_app_if≫.

The control registers in exzess_pcie_app_if store the DMA base address which is set by the exzess.ko kernel module. This address is also permanently output to the FPGA application. To check on the function of the FPGA, several counters are maintained counting all types of received and transmitted TLPs. As these registers are all local to exzess_pcie_app_if and registers accesses are not supposed to have side-effects, accesses to them are short-circuited within the exzess_pcie_app_if state machine.

As shown in figure 4.8 the normal sequence of events for accesses to others than the control registers is more complex. The application is supposed to handle accesses reading and writing device memory in the second BAR area BAR1, which is usually mapped —possibly via an application defined mapping function— to the DMA area. As a simple test one could create an application such that each DW of data read from BAR1 is read from the DMA area and then transmitted as a completion for the initial read request.

Different graphical representations of transactions are provided in figures 4.5 and 4.6 for a generalised representation and 4.3 as well as 4.4 for two concrete examples. In all these diagrams, both exzess modules as well as the PCI Express core have been condensed into ≪exzess≫ for the sake of clarity.

Generally, the handling of a read request can be explained by following the states and transitions in figure 4.8: After an incoming memory read TLP, the relevant address from that TLP is transmitted to the application. The state machine leaves the ≪rst≫ state and enters the ≪wait a bit≫ state, where it remains waiting for relevant inputs. The application may then react by reading or writing DMA memory. Those memory read or write requests cause the state machine to transit into the ≪outgoing write≫ or ≪outgoing read≫ states, where the creation of the corresponding TLP is triggered in ≪exzess_pcie_tlpgen≫. If a read TLP is requested by the application, the application interface then waits for the corresponding completion (in the ≪wait for CplD≫ state) and transmits the completion data to the application in the ≪wait for ack≫ state. The application may also trigger a write TLP, after which a completion is not expected. In both cases, after the reception of a completion or the successful transmission of a memory write TLP, the application interface may then expect further requests from the application in the ≪wait a bit≫ state. Alternatively the application signals the application inter-

face, that the current request cycle is done, triggering the transitions marked with ≪app_ack≫. The two possible transitions from each state marked ≪we≫ or ≪!we≫ —write enable or not write enable— are taken depending on whether the original request was a write or read request. In the case of a read request it is necessary to create a completion for the initial request. After successful transmission of the completion, the cycle ends and the state machine returns to the normal idle state.

The cycle for an initial write TLP is quite similar, with one difference. After the application signals the end of the cycle, no completion is transmitted because no completion data is expected.

Communication of the exzess application interface and the application is achieved through two Wishbone interfaces. According to their roles these interfaces are called ≪outer≫ and ≪inner≫ after the part of the nested transactions they are handling or ≪app≫ and ≪mem≫ after the kind of requests that interface handles.

Seen from the application interface, a Wishbone master interface signals a cycle along with the read address or write address and data. When the application signals an `ack` signal to that bus cycle, the request is considered to be handled successfully, in the case of a read request the data is transmitted in a completion. The transaction or bus cycle remains open until the application has completely processed the original request. This bus cycle coincides with the ≪outer≫, blue, primary transaction in figure 4.5. Therefore this Wishbone interface is called the ≪outer≫ or ≪app≫ interface.

A wishbone slave interface receives requests from the application. Those requests are then translated into memory read and write TLPs to the DMA area. Completions to these (read) requests or the successful transmission of write requests are signalled by sending an `ack` signal to the application. One important difference to the usual Wishbone semantics is that the application interface only handles transactions on the slave (mem or inner) interface when a bus cycle on the master (app or inner) interface is currently active. This reflects the intended usage pattern, but probably makes the Wishbone implementation in these interfaces non-compliant with the Wishbone specification. The bus cycles of the secondary, ≪inner≫, ≪mem≫ interface coincides with the inner, secondary, green transactions towards main memory in figures 4.5 and 4.6.

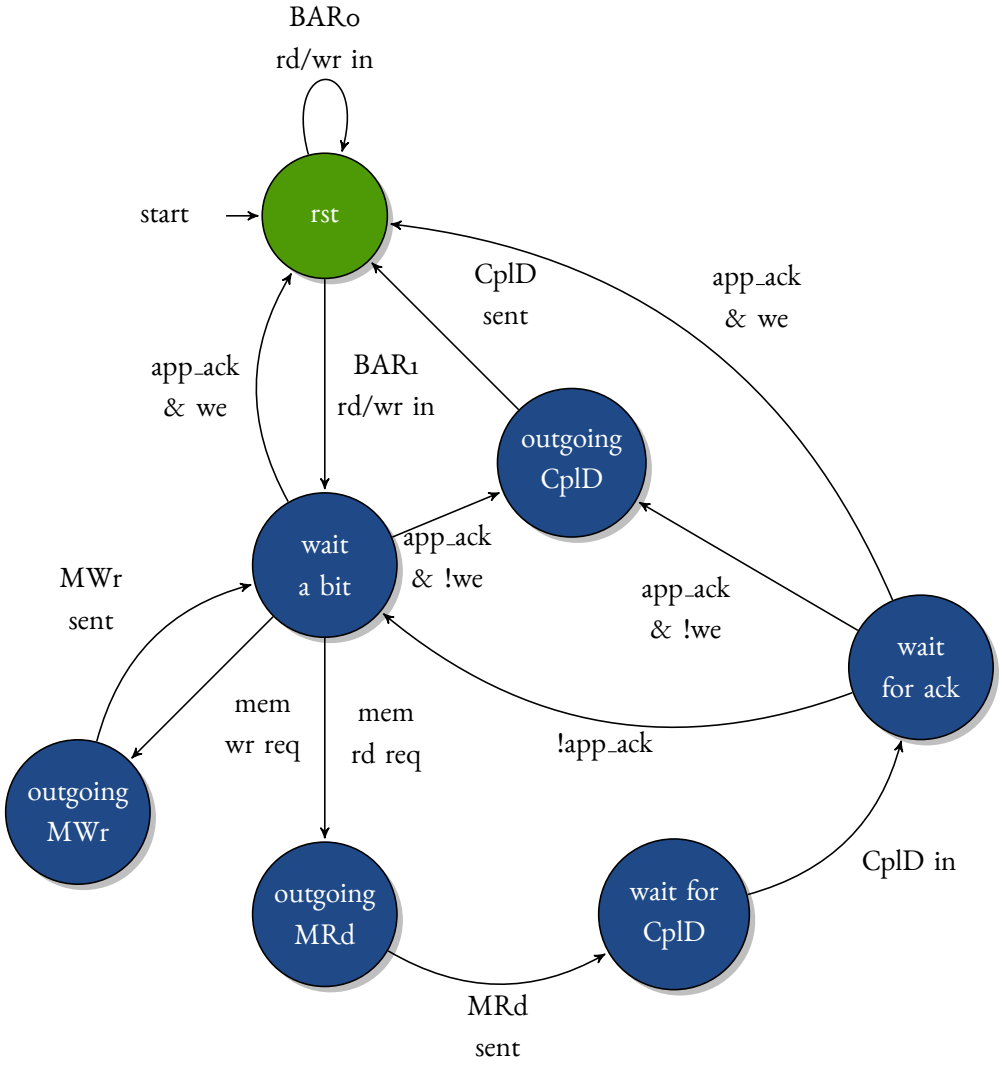


Figure 4.8: ≪exzess_pcie_app_if≫ state machine controlling the two wishbone interfaces to the application and the interface to the TLP generation in ≪exzess_pcie_tlpgen≫. The transitions are marked by their expected inputs, the states by their function. The outputs of each state are explained in the text on page 44. The state machine also loops in a state if no transition is triggered by an expected input, those transitions are not shown to keep the diagram simple. The only exception is the ≪wait for ack≫ state which is always left after one clock cycle and is only necessary for timing reasons. The transitions marked ≪app≫ wait for an event on the Wishbone ≪app≫ or ≪outer≫ interface, those marked ≪mem≫ listen on the ≪mem≫ or ≪inner≫ Wishbone interface. The reception and transmission of TLPs is marked by their type, ≪MWr≫, ≪MRd≫ or ≪CplD≫ and the direction, ≪sent≫ or ≪in≫. The transitions with ≪we≫, write enable, or ≪!we≫, not write enable, are taken if the original incoming TLP to BAR1 was a write or read TLP respectively. Requests to the registers in BAR0 are handled without leaving the ≪rst≫ state, as indicated by the loop.

4.4.3 Application

An application thus needs to implement a Wishbone slave and a master interface. On the slave interface the application will receive the initial requests and signal their eventual completion. During an open request the application may issue requests to the DMA area via its master interface.

This architecture presents the intended semantics for the FPGA software this work intends to create. Therefore the next logical step is testing this architecture by the implementation of applications. Two applications that have been implemented will be discussed.

4.5 XOR Application

The first real application, mostly because the mapping performed by this application is quite trivial, was intended to be a trivial encryption by ≪XOR-ing≫ some fixed value.

During the implementation of this application, many problems with the timing and function of the application interface were uncovered and fixed. Examples of such problems are the missing creation of completions, because the requesting signal was asserted for an insufficient amount of clock cycles, or the creation of multiple identical completions because the requesting signal was asserted after its reception had been acknowledged. Because all intricacies of generating PCI Express TLPs and the transaction handling are mostly abstracted away by the application interface, this module was quite short and easy to implement.

4.6 AES Application

Of course the previous example of using XOR with a fixed value is anything but a useful encryption. As an example of a state-of-the-art encryption algorithm, AES with a key size of 128 bits was chosen. As nothing useful can be gained from implementing yet another AES encryption core, a ready-made core from OpenCores[22] was used.

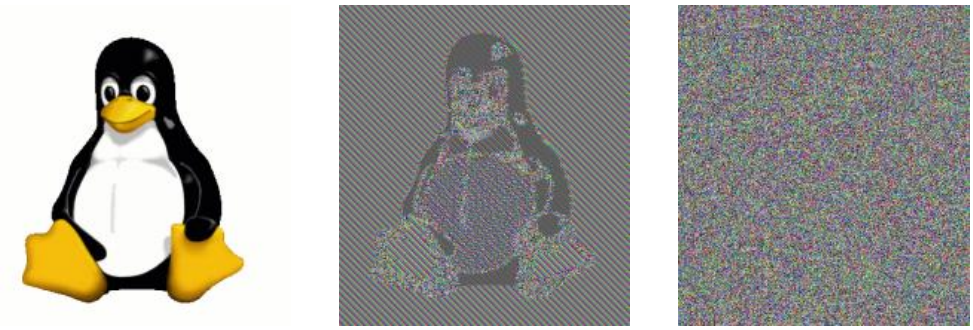
AES is a block cipher with a blocksize of 128 bits. This means that for 128 bit numbers x, y and k AES encrypts the cleartext x into ciphertext y by use of a key k:

$$y = f_{AES}(x, k)$$

While AES could directly be used in this way to encrypt memory contents with a static key k, one problem would emerge: AES has the mathematical property of being a bijection, that is, given two cleartexts x' and x'',

$$x' = x'' \Leftrightarrow f_{AES}(x', k) = f_{AES}(x'', k)$$

applies. This means that even if the cleartext was unknown to an attacker, certain patterns would be visible. Identical cleartext blocks would be indicated by identical ciphertext blocks. This could tell an eavesdropping attacker about the location of longer sequences of constant values, e.g. 0x00, or as in the



(a) unencrypted (b) ECB mode (c) chained or salted mode

Figure 4.9: Comparison of an unencrypted image (4.9a), the same image encrypted with an unspecified block cipher in ECB mode (4.9b) and in a chained (e.g. CBC) or salted (e.g. CTR) mode (4.9c). In ECB mode, an outline of the original image may be visible because evenly-colored image segments are encrypted into the same ciphertext and therefore a repeating pattern. More secure block cipher modi obscure such outlines by encrypting the same cleartext into different ciphertexts depending on their preceding block (CBC) or position (CTR). The Linux logo and encrypted images have been taken from [27].

example image in figure 4.9, areas of a bitmap image with the same color will be encrypted identically, revealing shapes in the bitmap.

Several solutions exist to deal with the weakness of the so-called ≪electronic code-book≫ or ≪ECB≫-mode. One possibility would be to include the previous data block into the key or input data of the current one. Such modi, like ≪cipher-block chaining≫ or ≪CBC≫ would introduce the problem that one data block can never be encrypted or decrypted on its own, adjacent blocks will always have to be read. A graphic representation of all mentioned block cipher modi is shown in figure 4.10.

For this reason, the ≪Counter≫, or ≪CTR≫ mode of operation will be used. In CTR mode, the encryption key is derived from the number of the current block in a sequence, or, in this case, from the memory address of the current block. This means the encryption function for the data block x_a at address a will be:

$$y_a = f_{AES}(x_a, a \oplus k)$$

However, this usage has one problem. The input data needs to be known before starting the AES encryption. This can be dealt with by encrypting the address a with the key k and `XOR`-ing the result with the cleartext:

$$y_a = x_a \oplus f_{AES}(a, k)$$

This way, the request for the ciphertext, when reading, can be sent at the same time the AES encryption is started. This mode of usage is also the usual definition of the CTR mode. With this optimisation, the performance on read transactions should be similar to the performance of the `XOR` application. Write

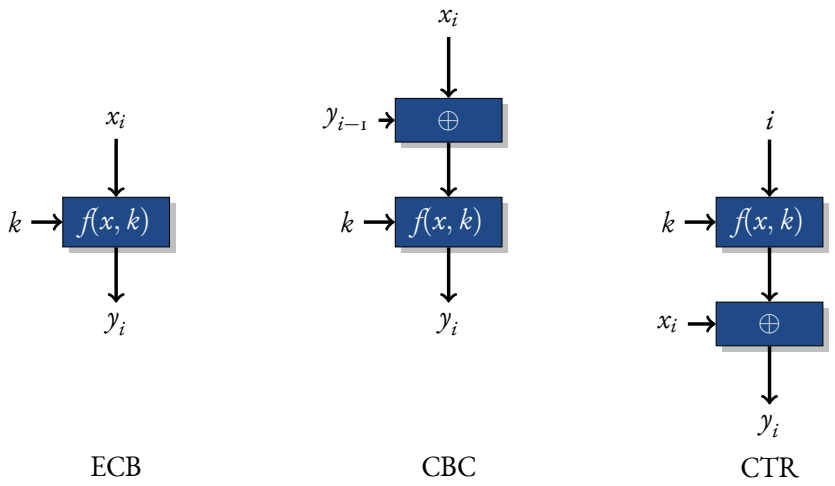


Figure 4.10: ECB, CBC and CTR encryption modi for block ciphers $f(x, k)$ encrypting the ith data block x_i with key k resulting in ciphertext y_i.

transactions should of course be somewhat slower since the data is readily available both in the `XOR` and AES application, but with the AES case the subsequent write of the encrypted data to the DMA area can only be sent after the encryption is finished.

The current application handles only 32 bit-portions of data. Yet the output block size of the AES encryption is 128 bit long. Currently this mismatch is handled by only using the lowermost 32 bit of each AES block. Of course this behaviour could be optimised by using all bits of the AES output to encrypt or decrypt a sequence of four data portions.

As the AES input block size and key size is also much larger than the address size of 32 bit, the address is padded to the required 128 bit by prepending a 96 bit nonce value n. Therefore the encryption used in the implementation is:

$$y_a = x_a \oplus f_{AES}(n|a, k)$$

The nonce and key values are currently implementation constants. This is of course unacceptable for real world use, but sufficient for a proof-of-concept implementation. The incorporation of random elements into the key and nonce values are trivial, for example by passing a random value via a BAR0 register when initializing the device in the operating system. This is currently not implemented as such randomness would make testing and debugging more complicated.

Implemented in this way the AES application is not much more complex than the previous XOR example. Only a small number of changes are necessary to trigger the AES core, provide it with data and when finished, `XOR` with the AES result instead of a constant.

4.7 Linux Kernel Module and Test Programs

Some initialisation is necessary to access the hardware from a Linux[28] system. This is done by the ≪exzess.ko≫ kernel module. The module reserves a DMA area, programs the BAR0 registers of the FPGA card with the DMA base address and provides access to the DMA area for userspace programs with a character device `/dev/exzess`. Through this device, easy manipulation via common tools is possible. The contents of the DMA area can be displayed via `hexdump -C /dev/exzess`, overwritten with zeroes via `dd if=/dev/zero of=/dev/exzess`, etc.

The DMA area is obtained through the `pci_alloc_consistent` kernel API which assigns a suitable DMA area to the device. The ≪consistent≫ or ≪coherent[2]≫ variant ensures that the memory area in question will not be subject to caching. The Linux kernel documentation[29, 30] words this as: ≪The invariant [...] is that any CPU store to memory is immediately visible to the device, and vice versa. Consistent mappings guarantee this.≫

Similar to the `/dev/exzess` device file, the contents of the BAR0 and BAR1 device memory areas can be accessed through files in the `/sys` sysfs-pseudo-filesystem that the kernel automatically provides.

Test programs have been written mapping device memory areas using the `mmap` syscall. The mapped area is then read and written to. From the number of bytes accessed and the elapsed time, a throughput value is calculated and displayed.

[2] `pci_alloc_consistent` is a wrapper for `dma_alloc_coherent`.

Chapter 5

Results and Measurements

The example applications implemented as described in sections 4.5 and 4.6 were used to prove that the application interface and other components of ≪exzess≫ were working and the concept was sound. With both example applications, benchmarks were conducted to verify the performance of both the applications and the ≪exzess≫ abstraction.

5.1 Proof of Concept

```
[...]
02:00.0 Memory controller: Xilinx Corporation Zomojo Z1
        Subsystem: Xilinx Corporation Zomojo Z1
        Flags: bus master, fast devsel, latency 0, IRQ 16
        Memory at c0010000 (32-bit, non-prefetchable) [size=1K]
        Memory at c0000000 (32-bit, non-prefetchable) [size=64K]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] MSI: Enable- Count=1/1 Maskable- 64bit+
        Capabilities: [58] Express Endpoint, MSI 00
        Capabilities: [100] Device Serial Number 78-1c-86-96-44-e8-cd-cb
        Kernel driver in use: exzess
[...]
```

Figure 5.1: Output of `lspci -v` with the AES application loaded.

The correct configuration of the PCI Express interface of the FPGA card was verified by `lspci` and the `dmesg` log output of the kernel module. A typical `lspci` output is shown in figure 5.1. The build

system includes the current GIT[1] revision as the device serial number, ensuring that the FPGA was loaded correctly and the correct revision of the code is being used. The device name in figure 5.1 is the same as for a Xilinx example application[26] as the PCI-ID of that application was used. For real-world use a proper PCI-ID would have to be assigned.

A typical initialisation log of the kernel module is shown in figure 5.2. After verifying the PCI Express device and kernel module initialisation, the applications were tested.

First both applications were subjected to manual testing. The BAR1 area was mapped in a Linux userspace application and known values were written to this area. By monitoring received and generated PCI Express TLPs on the FPGA and viewing the contents of the DMA area, correct function of the FPGA was verified.

5.1.1 XOR application

For the XOR application, such an exchange of TLPs is shown in figures 5.3 and 5.4.

In figure 5.3, a memory write TLP is received from the PCI Express core, starting at sample 11 signalled by the `trn_i.rx.rsof`(≪receive start of frame≫) and `trn_i.rx.rsrc_rdy`(≪ receive source ready≫) signals. The TLP is 4 DW long which are transmitted within 4 clock cycles on `trn_i.rx.rd`. The meaning of each DW can be interpreted according to figure 2.3: The first DW signals a memory write transaction with a data length of 1 DW, the second DW shows that the request comes from the root port (having a PCI-ID of all zeroes) and that in the one transmitted data DW all bytes are valid because all byte enables are set. The third DW is the request address, in this case pointing to the beginning of the BAR1 memory area. The address of this BAR1 area can also be obtained from figure 5.2. The fourth received DW is the data to be written, here 4 byte of 0x00. The end of the received TLP in sample 14 should be accompanied by an end-of-frame signal, but during tests the `trn_i.rx.reof` signal from the PCI Express core seemed to be unreliable and has therefore been removed from this implementation. The lack of an end-of-frame signal is dealt with by knowing the expected length of each TLP from the start-of-frame signal. The PCI Express core also outputs a `bar_hit`[2] signal which indicates whether a memory transaction affects a configured BAR area, and if so, which one.

This `bar_hit` signal is transmitted to the application interface together with the received TLP qualified by the `rx_mem_wr_o.valid` signal in samples 14 to 17. The application interface acknowledges receiving the TLP in cycles 17 and 18 with the corresponding `rx_mem_wr_i.ack` signal.

Around sample 19, the application will receive as well as process the TLP and decide to create a memory write TLP to the DMA area in response. The application interface signals the memory write TLP to be transmitted using the `tx_mem_wr_i.valid` signal which is immediately acknowledged. Starting from cycle 22, the outgoing memory write TLP is handed over to the PCI Express core on `trn_o.tx.td`, accompanied by the appropriate sequence of start/end-of-frame and ready signals, named with prefix `trn_o.tx` and suffixes `.tsof`, `.teof` and `.tsrc_rdy`.

[1] GIT is the revision control system that was used for this work.

[2] The one-hot encoding of the `bar_hit` signal in bits is not shown here, instead only the interpretation is used in figure 5.3.

With the PCI Express core being able to buffer a number of incoming TLPs, a following incoming memory write TLP is already waiting, starting in cycle 17. By deasserting the `rdst_rdy` signal, the reception is delayed until cycle 25, such that only one TLP is processed at any given time.

The TLP sent can also be recognised as a memory write TLP with a data length of 1 DW, starting with the first DW in cycle 22. The requester ID in the second DW is consistent with the PCI-ID `02:00.0` in figures 5.1 and 5.2. The ≪byte enable≫ bitmask in the last byte of the second DW is of course identical to the original request. The third byte contains the generated request tag, which is created from a counter, leading to this byte being changed for each transaction requested by the PCI Express card. The third DW contains the requested address in the DMA space assigned to the card, again consistent with figure 5.2. The lower bytes of the DMA address are copied from the lower bytes of the original request. The fourth DW contains the data to be written to the DMA area, in this case identical to the XOR pattern as the original data consisted only of zeroes.

The interpretation of figures 5.3 and 5.5 allows to conclude that the XOR application works as expected for memory write transactions. Similar manual tests show read transactions also working as expected.

From figure 5.3 a rough estimate of the maximum performance for memory write transactions can also be obtained. From one incoming transaction to the second incoming transaction, 15 clock cycles are needed for the whole process of receiving, processing and sending. Within those 15 clock cycles, 4 bytes are written. Therefore, at a FPGA clock speed of 100 MHz, a maximum throughput of 26.6 MB/s should be possible in theory. Memory reads are non-posted transactions, making it necessary to wait for completions. Therefore the performance for read transactions should always be significantly slower than for writes.

From ChipScope traces, the time between the start of two read transactions is 230 cycles, corresponding to a maximum theoretical performance of 1.7 MB/s. The length of one read transaction from receiving the request to sending the completion is only 130 cycles, but unlike in write transactions, the CPU will wait until the data is received before issuing another read. From this, the time data or requests take from the CPU to the FPGA and back can be estimated as 100 cycles or 1 μs. A comparison of these values with measured throughput values will be done later in section 5.2.

These estimates are not completely realistic as they ignore the effects of flow control and buffering on the PCI Express bus and in the core as well as effects on CPU and RAM which increase the absolute time each transaction takes.

5.1.2 AES application

The AES application creates the same sequences of PCI Express TLPs as the XOR application. As an example, a read TLP transaction is shown in figure 5.6. Only the data part of each TLP from and to the DMA area differs in the encryption used. Therefore verification of the function for the AES

application is already mostly done or works exactly like for the XOR application. The AES encryption is the only part which needs to be evaluated.

For comparison, the AES symmetric encryption mode of the OpenSSL command line tool is used. OpenSSL[31] is a popular and open-source cryptography framework. After writing test data, an increasing counter, into the BAR1 memory area, the DMA area will contain the data shown in figure 5.7.

To decrypt and verify the data shown in figure 5.7, an openssl command[3] can first be used to calculate the key for each DW of memory. This key is then XORed to the ciphertext stored in this DW to obtain the plaintext. The first two key DWs as calculated by the given command are 0x0dff5efb and 0x1136c312. When XOR-ing those values with the ciphertext above, the result is 0x00000000 and 0x01000000 respectively, consistent with the increasing counter written into the BAR1 area.

When writing only a sequence of zeroes or some other constant, the effect of the CTR encryption mode can also be observed: While in ECB mode, the DMA area would repeat one value, now a seemingly random sequence is visible as shown in figure 5.8.

This sequence is of course also the sequence of keys. Therefore an attacker that is able to read the DMA area and to choose a plaintext is also able to obtain the key sequence and decrypt every other ciphertext. The same applies for an attacker knowing a plaintext, the key sequence can be obtained by XORing the ciphertext with the known plaintext. An example can be seen when comparing figures 5.7 and 5.8. Both assumptions do not apply for the cold-boot attack scenario, but for other scenarios, changes would have to be made to the use of encryption. In any case, the nonce and key values should be more randomly chosen, and new nonce and key values should be chosen at each bootup.

The maximum performance of the AES application can also be estimated in the same manner as for the XOR application. A memory write transaction takes 29 cycles, which is 14 cycles longer than for the XOR case. The difference comes from the time the AES core takes for the encryption of one block. A memory read takes roughly 230 cycles, which is the same as for the XOR case, as the implementation ≪hides≫ the cycles needed for the AES encryption within the time spent waiting for the DMA read completion to arrive. Therefore performance estimates are 13.8 MB/s for writing and roughly 1.7 MB/s for reading transactions.

[3] With the currently used values for the nonce, the command is `echo 123456789ABCDEF0123456789ABCDEF0 | xxd -r -p | openssl enc -aes-128-ecb -nopad -K 081547112412AE5128150182bb080000 -nosalt | hexdump -C`, where 0xbb080000 should be replaced by the physical address of the DW in the DMA region. The key is the last DW of the output, the rest of the output is discarded.

```
[...] exzess: initializing kernel module
[...] exzess: got probe for device 10ee:0007 (sub ffffffff:ffffffff)
[...] exzess 0000:02:00.0: enabling device (0000 -> 0002)
[...] exzess 0000:02:00.0: PCI INT A -> GSI 16 (level, low) -> IRQ 16
[...] exzess: setting master
[...] exzess 0000:02:00.0: setting latency timer to 64
[...] exzess: enabling MWI
[...] exzess: enabling MWI failed
[...] exzess: got ioaddr c0010000 len 00000400 type mem at BAR0
[...] exzess: got addr ffffc90013806000 after pci_iomap
[...] exzess: memory region already in use
[...] exzess: got ioaddr c0000000 len 00010000 type mem at BAR1
[...] exzess: got addr ffffc90001ca0000 after pci_iomap
[...] exzess: memory region already in use
[...] exzess: nothing at BAR2
[...] exzess: nothing at BAR3
[...] exzess: nothing at BAR4
[...] exzess: nothing at BAR5
[...] exzess: setting DMA mask
[...] exzess: trying DMA stuff, getting 64k consistent memory
[...] exzess: got DMA at address ffff8800bb080000 virtual, 00000000¶
              bb080000 physical, size 65536
[...] exzess: got character device 262144000
[...] exzess: init done...
```

Figure 5.2: Typical initialisation of the kernel module from the Linux `dmesg` kernel log. Timestamps have been replaced by ellipses and an additional linebreak marked with ≪¶≫ has been inserted to improve readability.

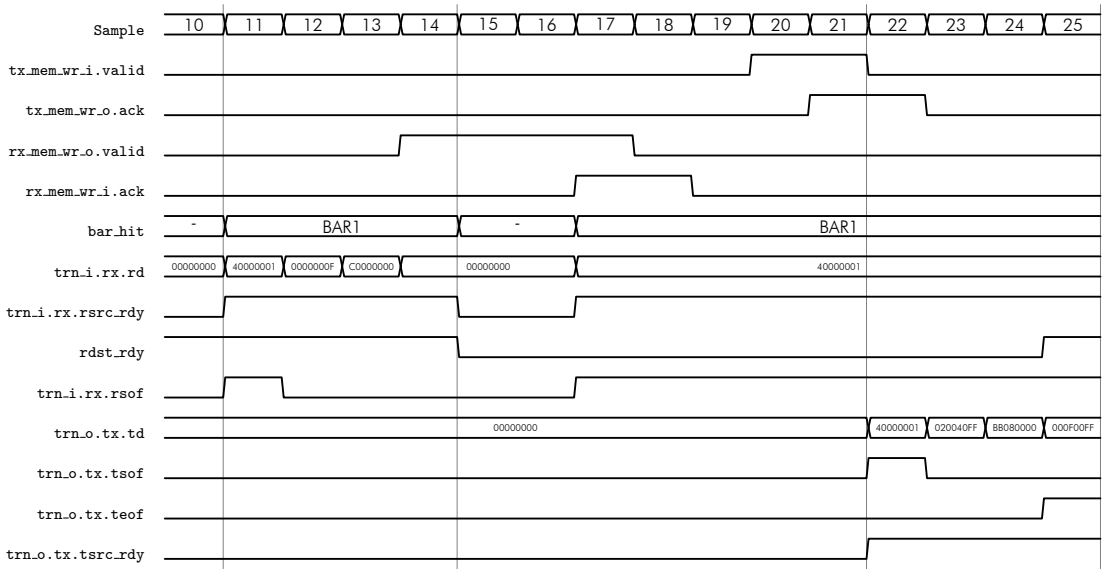


Figure 5.3: Receiving, processing and sending of one memory write (`mem_wr`) TLP. The ≪sample≫ sequence numbers the clock cycles starting from an arbitrary point. Data is exported from debugging data captured via ChipScope and appropriately shortened and selected for display. Grey lines mark the start and end of the reception (sample 11 to 14) and transmission (sample 22 to 25) of the TLPs. The signal names are consistent with the ChipScope files that accompany the implementation, but not always equal to the signal names used in the VHDL implementation.

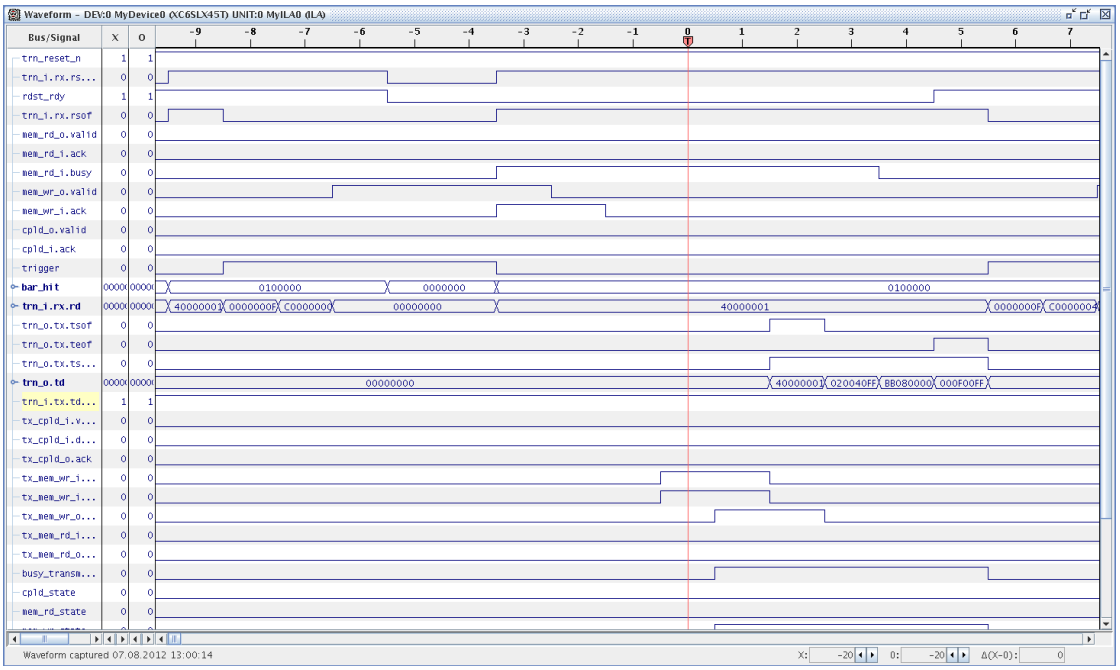


Figure 5.4: Reception, processing and sending of one memory write TLP as displayed in the ChipScope user interface. The data was exported and is explained in the context of figure 5.3.

```
# hexdump -C /dev/exzess | head -4
00000000  00 0f 00 ff 01 0f 00 ff  02 0f 00 ff 03 0f 00 ff  |................|
00000010  04 0f 00 ff 05 0f 00 ff  06 0f 00 ff 07 0f 00 ff  |................|
00000020  08 0f 00 ff 09 0f 00 ff  0a 0f 00 ff 0b 0f 00 ff  |................|
00000030  0c 0f 00 ff 0d 0f 00 ff  0e 0f 00 ff 0f 0f 00 ff  |................|
```

Figure 5.5: Output of `hexdump` on the `/dev/exzess` device file which outputs the contents of the DMA area. The content shown is the result of writing an increasing counter into each DW of the BAR1 area of the PCI Express card, which applys an XOR with oxooofooff to each value.

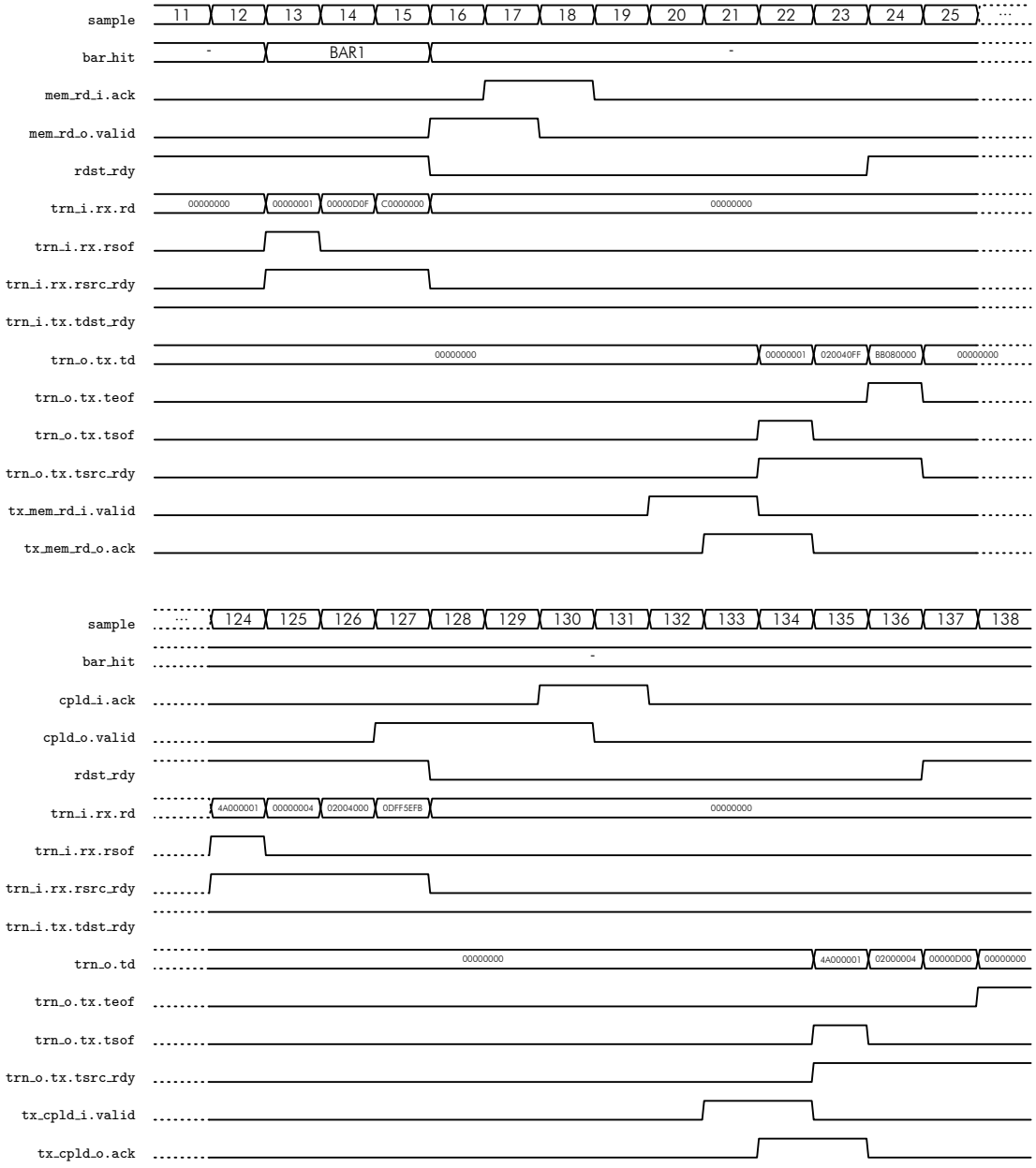


Figure 5.6: Reception of a memory read TLP for the BAR1 memory area (samples 13 to 15), transmission of the corresponding completion (samples 135 to 138) and transmission of the secondary memory request to the DMA area (cycles 22 to 24) and the corresponding completion (cycles 124 to 127). Only a selection of signals is shown and 99 clock cycles have been removed as indicated by the dotted parts in the diagram, where no state changes take place. The correspondence of the transaction IDs consisting of the PCI-ID and tag fields can be checked in first three bytes of the second DW of each memory read TLP (cycle 14 and 23) and in the first three bytes of the third DW of each completion with data TLP (sample 126 and 137). The completion data field in cycle 127 contains the encrypted data read from DMA, the data field in cycle 138 contains the decrypted data in answer to the original BAR1 read request. The application in use is the AES application, therefore the encryption in use is AES as also shown in figure 5.7.

```
# hexdump  -C /dev/exzess | head -4
00000000  0d ff 5e fb 10 36 c3 12  bd 51 5b 86 a9 40 ad e3  |..^..6...Q[..@..|
00000010  b4 7f c9 fb 94 fe 19 22  43 7c 84 a4 00 fe 45 ba  |........"C|....E.|
00000020  67 91 45 13 95 dc a3 8c  93 6a 71 04 6b 83 8a 18  |g.E......jq.k...|
00000030  d4 6c 28 c6 f3 a5 d8 09  43 0d 4e 75 8a 73 23 3d  |.l(.....C.Nu.s#=|
```

Figure 5.7: Output of `hexdump` on the `/dev/exzess` device file which outputs the contents of the DMA area. The content shown is the result of writing an increasing counter into each DW of the BAR1 area of the card, which applys an AES128-CTR encryption to each value.

```
# hexdump  -C /dev/exzess | head -4
00000000  0d ff 5e fb 11 36 c3 12  bf 51 5b 86 aa 40 ad e3  |..^..6...Q[..@..|
00000010  b0 7f c9 fb 91 fe 19 22  45 7c 84 a4 07 fe 45 ba  |........"E|....E.|
00000020  6f 91 45 13 9c dc a3 8c  99 6a 71 04 60 83 8a 18  |o.E......jq.`...|
00000030  d8 6c 28 c6 fe a5 d8 09  4d 0d 4e 75 85 73 23 3d  |.l(.....M.Nu.s#=|
```

Figure 5.8: Output of `hexdump` on the `/dev/exzess` device file which outputs the contents of the DMA area. The content shown is the result of writing zeroes into the BAR1 area of the card, which applys an AES128-CTR encryption to each value. Note the similarity to figure 5.7.

5.2 Benchmarks

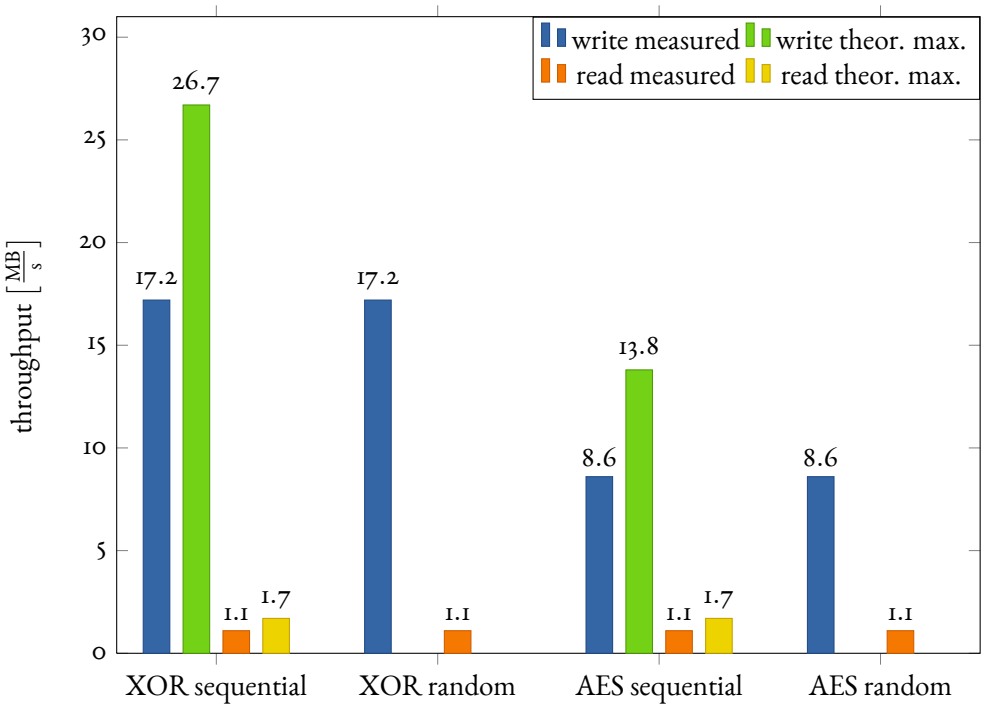


Figure 5.9: Benchmark results for both applications, XOR and AES in read and write mode. The measured values are averaged over 100 iterations. Standard deviations and numerical values are shown in table 5.1. Calculated theoretical maximum values should be the same for sequential and random access patterns, but processing times have only been explicitly measured in the sequential access case.

To measure the actual performance of the AES and XOR applications, a simple benchmarking application has been implemented which reads and writes the BAR1 memory area 100 times and from this creates averaged measurements and error estimates. These measurements are shown in figure 5.9 and table 5.1.

As expected, the measured values are lower than the calculated theoretical maximum values. The performance for read transactions, when comparing the XOR and AES applications, is equal within the precision of measurement. Performance values for write transactions are lower in the case of AES, the XOR application has roughly twice the write throughput of the AES application. This is also consistent with the predicted behaviour.

Random accesses have essentially the same performance as sequential accesses, at least for writing accesses. In the case of reads, the performance is approximately 1 % lower for random reads than for

application	access pattern	access direction	theoretical maximum $\left[\frac{\text{kB}}{\text{s}}\right]$	measured $\left[\frac{\text{kB}}{\text{s}}\right]$
XOR	sequential	write	26667	17223 ± 7
XOR	sequential	read	1739	1085 ± 1
XOR	random	write		17221 ± 6
XOR	random	read		1078 ± 1
AES	sequential	write	13793	8621 ± 3
AES	sequential	read	1739	1086 ± 1
AES	random	write		8620 ± 3
AES	random	read		1077 ± 1

Table 5.1: Benchmark results for both applications in read and write mode. The data for this table is also shown in the plot in figure 5.9, therefore the same remarks as in figure 5.9 apply.

sequential reads, both in the XOR and AES application. The difference is small but much larger than 5σ and therefore significant. While I have no definitive explanation for this difference, I would consider readahead behaviour the most likely cause.

I consider the performance values very good for a proof-of-concept implementation. Of course there are many measures that can be taken to improve these values if necessary, some of which will be discussed in section 6.1.

Chapter 6

Future Work and Conclusion

This chapter will present several ideas for applications based on the ≪exzess≫ abstraction and for improvements for the ≪exzess≫ modules themselves. These ideas are provided to give the reader an idea what might be possible and —if developed further— how similar hardware devices could influence the state of the art. As the following examples have not yet been implemented, their viability is of course unclear.

6.1 Future Improvements

First of all, performance can be increased by various measures. Currently, the configuration of the BARs and the DMA area forbids any caching. This means that on repeated reads, the usual cache speedup will not take place. Also, mechanisms like read-ahead, with memory being read in cache-line sized portions and —depending on the access pattern— the following cache-line also being speculatively read, would work and increase performance for many usage patterns. It would of course be necessary in this case to also implement the provided zero-length-read method for flushing the cache from the PCI Express device. The handling of larger reads than 32 bit might be necessary when the cache mechanisms request whole cache-lines which are typically 64 byte long.

A read-ahead mechanism for reads to the DMA area would also be beneficial, as currently the round-trip-time seems to be the main performance issue when reading from the DMA area. On a 4 byte read, the FPGA could request 8 byte or more of data from the DMA area and use the additional data in the following request if that request is a memory read on the following address. For reads in linear sequences, this might increase performance by a factor of 2 or more.

The BAR memory area and the DMA area are currently only 64 kB in size. While this is sufficient for benchmarking purposes, real world use will of course require more available memory. A simple measure to increase the memory available via the BAR area would be to allocate a larger DMA area and use the DMA base address as a pointer to a moving window. The window size would be the smaller BAR area size, while the larger DMA area can be accessed as a whole by resetting the DMA base address

if necessary. Yet this implementation would be problematic because software support would be needed to ≪move≫ the window. The generally preferable way to increase available memory is to increase the BAR area size. This size is limited by the general allocation for PCI address space a computer system provides, which prevents growth to sizes in the hundreds of megabytes range. Perhaps a move to 64 bit addresses on the PCI Express interface could circumvent this limitation. This would also allow for DMA areas larger than what is addressable by 32 bit addresses.

The current interface to access BAR memory areas from a userspace program is unsuitable for frequent use as root access priviledges are necessary and the whole memory area can be accessed. For access as a ≪normal≫ user and to ensure the usual page-granular memory mapping and protection, an operating system interface to map pages from the BAR memory area is necessary.

Similarly, key creation and management for the AES application would provide greater security than the current static nonce and key, which is only secure as long as an attacker is unable to access the source code. Generally the use of cryptography needs to be checked for possible problems and improved accordingly.

DMA attacks are attacks used to copy RAM contents including cryptographic keys. While the attack goal of obtaining RAM contents is the same as in a cold-boot attack, the means to attain this goal are different. A DMA attack exploits the fact that with physical access to a running computer, several vulnerable[1] bus systems like FireWire and PCI and PCI Express variants such as Thunderbolt, PCCard or ExpressCard may be accessible. If a suitable device is plugged in to one of the aforementioned external ports, the device can then read the RAM contents via DMA requests these busses allow and make the contents available to the attacker. When using the AES application as a cryptographic key storage mechanism, the attacker will be able to read the encrypted portion of main memory via DMA. The attacker will also be able to request the device memory of the FPGA card and thus the decrypted contents. Such requests will carry the PCI-ID of the attacking device or the bridge device for the respective bus system. If this PCI-ID is not equal to the PCI-ID of the root complex, such malicious requests could be rejected, thereby denying the attacker access to the unencrypted contents. However, this measure will not prevent more sophisticated DMA attacks, where a malicious device does not directly generate read requests, but instead injects malicious code which will be executed by the CPU. This injected code would then create malicious requests from the CPU which would be indistinguishable from legitimate requests.

Currently, only one application can be used at any time, any change of applications makes resynthesizing and reprogramming of the FPGA necessary. But there should be no problem in principle to provide multiple applications in the same FPGA image and allow switching between those applications, even on a per-address basis, such that allocations with different applications are possible. Applications could also be ≪stacked≫, applying several transformations to the same memory area at the same time. One could for example imagine checksumming an encrypted area for additional integrity protection.

[1] A bus system is vulnerable in the context of DMA attacks if it allows a device to create DMA requests to unrestricted memory addresses.

Finally, applications or the operating system can be modified to use the various applications the FPGA provides. In the case of the AES encryption application, one possible use would be to store sensitive material in the encrypted memory area. Various ≪keyring≫ applications that store encrypted passwords, which are decrypted by a master password for a certain period of time could store the decrypted passwords there. The passwords would be accessible for use, but even an attacker employing the usual cold-boot techniques would be unable to extract these passwords from memory. In a similar way, decrypted private keys of asymmetric cryptographic systems like the keys in an ≪ssh-agent≫ or X.509 certificates, Kerberos ticket caches and the Linux kernel key store can be stored in encrypted memory. In the case of the Linux kernel key store, this would also protect keys for HDD encryption systems like LUKS. While of course the current performance of exzess might be somewhat insufficient for purposes like HDD encryption, in the other cases I would not expect problems even without the aforementioned performance improvements. For signing certificates, SSH-keys or Kerberos ticket caches, the sensitive material is small enough, the frequency of access relatively low and the added latency neglegible, so that no discernable performance impact in these applications should occur.

6.2 Applications

6.2.1 Data Integrity and Consistency Checking

There are several possible ways to ensure the integrity and consistency of data. The first solution would simply be redundancy: data is stored multiple times in different locations and on reading, the integrity is verified by comparing the different copies of the data read. If, for example by a damaged memory chip or in a high-radiation environment, stored data is damaged, one copy of the data would be sufficient to recognise the damage, two copies could be used to correct errors by a majority vote determining the two undamaged versions.

The amount of memory used in this kind of redundant setup would of course, for n copies of the original amount data s be $(n+1)\cdot s$. While in some cases such a high amount of redundancy is acceptable, in most other situations, a significantly reduced memory usage in trade for a somewhat lower level of redundancy is preferable.

The generally accepted and commonly available way of achieving data integrity and consistency is via use of checksums, in RAM usually by using ≪ECC≫ or ≪Chipkill≫ memory with compatible memory controllers. For most use cases, ≪ECC≫ will of course be preferable, being a well-tested and readily available technology. But use cases can always be found where an improvement on the current use of checksumming technologies may be useful. First improvement would be the implementation of checksums with more redundancy, enabling the recognition and correction of more bit-errors than currently available. Such an improved redundancy could also be applied adaptively, to increase or decrease the protection on different parts of memory, depending on the importance of the data stored therein.

Another application of the improved flexibility would be to utilise the more detailed knowledge about the structure of the data stored in memory. Data that by its nature contains checksums, e.g. certain types of network packets, could be automatically verified on writing or reading, if the FPGA knows enough about the internal structure of the packet to locate the checksum and to identify the portion of data the checksum was calculated over.

Other constraints on the data could be applied and enforced, too. If for example an array of floating point values is known to only contain values within a certain interval $[a; b]$, the FPGA application could then check for compliance with this constraint and either clip the stored values to this interval or replace problematic values with special error markers.

6.2.2 Data Aggregation and Processing

In some areas like embedded applications, it might be preferable to minimise the load on the system's CPU and use specialised hardware components to conserve energy or improve reliability as much as possible. One idea in this direction would be to use the FPGA for sensor readout, sensor fusion and filtering of data.

A necessary precondition is that sensor readout can happen either through the device memory of the sensor devices, or through DMA areas where all sensors deposit their readouts. The interface provided to the CPU would be an area of device memory, where the current, aggregated and filtered sensor readout can be read. On receiving a read request for this area, the FPGA application would query all sensors via their DMA or device memory areas, evaluate and filter these values and provide a completion, with the one aggregated, filtered output value.

By using an FPGA in a DSP-like function in this way, more sophisticated processing of sensor data is possible without the need to modify CPU applications in any major way. Through the ≪exzess≫ abstraction it may be possible to hide all complexity from the CPU application, which would still only read one value from a memory area, as if a single normal unfiltered sensor had been used.

6.2.3 Transactional Memory

Transactional memory[32] (short ≪TM≫) is the concept of creating memory architectures, either in software or hardware, where read and write operations on memory form transactions which observe the traditional ≪ACID≫ guarantees of atomicity, consistency, isolation and durability. Consider a multi-processor system with several threads of execution $T_0 \ldots T_n$ running in parallel, sharing a common memory area. One process T_w issues a sequence of write operations w that form a transaction. The expected semantics dictate that the write operations w are only visible to other threads after successful completion of the whole transaction or not at all. Concurrent transactions w and w' should be synchronised in such a way, that there is a consistent serial ordering of those transactions, i.e., either the effects of w appear to have happened entirely after w', or the effects of w' appears to have taken place entirely after w. Several variants of how these guaratees can be implemented are known.

A TM application for ≪exzess≫ could be implemented by first assigning each thread $T_0 \ldots T_n$ a non-overlapping indentically-sized part of device memory $a_0 \ldots a_n$. Two DMA areas of the same size b_r and b_w are allocated and filled with the same content. When no writing transaction is taking place, all reading accesses to an a_i area are executed by reading from b_r and forward the result, such that all device memory areas $a_0 \ldots a_n$ show the contents of b_r. On a new writing transaction from thread T_w, all write accesses and all read accesses from thread T_w are directed to the DMA area b_w. Reading accesses from other threads are still directed to the b_r DMA area containing the state of the memory before the start of the writing transaction. When T_w issues a commit command for the write transaction, b_w and b_r are switched, such that all new read accesses observe the newly modified contents after the write transaction. The new b_w is prepared for further use by copying the contents of the new b_r.

In this way, ≪exzess≫ can be used to implement a buffer management device for a hardware TM system. Several problems, however, need to be resolved, for example how synchronisation between different writing transactions should be handled (e.g. by aborting one transaction or using multiple write buffers and merging), what the size of the buffers and therefore the granularity size of the transactions should be and how actions like starting, commiting and aborting of a transaction should be signalled.

6.3 Conclusion

This work has presented a functional proof-of-concept implementation of a transparent memory encryption and transformation device based on an FPGA PCI Express development board. Two example applications for memory encryption by XOR and AES have been presented. Benchmarks of both example applications are already very promising.

After initial problems with the hardware, the implementation has progressed quickly, such that the achieved state of the software and hardware at this point is satisfactory. The original motivation of providing a hardware-based mitigation for cold-boot attacks by transparently encrypting parts of the systems memory space has been achieved. Future performance improvements as well as a wide range of further possible applications may expand the field of use far beyond the original motivation.

Bibliography

[1] AVM Wiki – Lexikon – Spartan, available via `http://www.wehavemorefun.de/fritzbox/Spartan`, 2012.

[2] AVM website – Produkte – Fritz!Box, available via `http://avm.de/de/Produkte/FRITZBox/index.php`, 2012.

[3] T. Müller, F. C. Freiling, and A. Dewald, Tresor runs encryption securely outside ram, in *Proceedings of the 20th USENIX Security Symposium*, p. 251ff, USENIX Association, 2011.

[4] J. Pabel, FrozenCache — Mitigating Cold-Boot-Attacks For Software-Based Full-Disk-Encryption, in *27th Chaos Communication Congress*, 2010.

[5] T. Müller, B. Taubmann, and F. C. Freiling, Lecture Notes in Computer Science 7341, 66 (2012).

[6] M. E. Kabay, Cold-boot attacks change the data leakage landscape, Published via Network World, available at `http://www.networkworld.com/newsletters/2009/032309sec1.html`, 2009.

[7] G. Marcus, W. Gao, and A. Kugel, The mprace framework, available via `http://li5.ziti.uni-heidelberg.de/mprace/`.

[8] Accelerated fir with built-in direct memory access example, available via `http://www.altera.com/support/examples/nios2/exm-accelerated-fir.html`.

[9] D. Lau, O. Pritchard, and P. Molson, Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions, in *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pp. 45 – 56, 2006.

[10] PCI Express Base Specification Revision 1.0a, 2002, 2003.

[11] R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture* (Addison-Wesley Professional, 2003).

[12] D. A. Patterson, Lecture 18: Snooping vs. directory based coherency, 1996.

[13] FPGA, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/FPGA`, 2012.

[14] K. Coffman, *Real World FPGA Design with Verilog* (Prentice Hall, 1999).

[15] Intel releases x86-FPGA hybrid - app store to follow?, The Daily Circuit, available via `http://www.dailycircuitry.com/2010/11/intel-releases-x86-fpga-hybrid-app.html`, 2010.

[16] Enterpoint Ltd., *RaggedStone2 User Manual*, , issue 1.02 ed., 2010.

[17] Analysis: Xilinx Spartan-6 and Virtex-6, The EE Times, available via `http://www.eetimes.com/design/signal-processing-dsp/4017756/Analysis-Xilinx-Spartan-6-and-Virtex-6`, 2009.

[18] P. J. Ashenden, *The Designer's Guide to VHDL*, 2 ed. (Morgan Kaufmann Publishers, 2002).

[19] VHDL, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/VHDL`, 2012.

[20] Verilog, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/Verilog`, 2012.

[21] J. Gaisler, A Structured VHDL Design Method, available via `http://www.gaisler.com/doc/structdes.pdf`.

[22] OpenCores — Open-Source hardware community website at `http://www.opencores.org`.

[23] Wishbone B4 specification, available at `http://www.opencores.org`, 2010.

[24] Xilinx, Inc., *Spartan-6 FPGA Integrated Endpoint Block for PCI Express Pre-Production User Guide*, , ug672 (v1.0) ed., 2010.

[25] Enterpoint Ltd., *PCI Express Test Build*.

[26] J. Wiltgen and J. Ayer, Xilinx, Inc. Report No. XAPP1052 (v3.2), 2011.

[27] Electronic Code Book, Wikipedia — The Free Encyclopedia, available via `http://en.wikipedia.org/wiki/Electronic_codebook`, images by L. Ewing (lewing@isc.tamu.edu, made with The GIMP) and ≪Lunkwill≫, 2012.

[28] M. Opdenacker, Linux PCI drivers, 2010.

[29] D. S. Miller, R. Henderson, and J. Jelinek, *Dynamic DMA mapping Guide*, Documentation/DMA-API-HOWTO.txt in the Linux kernel 3.0.31 source tree.

[30] *Linux kernel documentation for kernel version 3.0.31*, Documentation/ subdirectory in the Linux kernel source tree.

[31] OpenSSL, OpenSSL documentation, available via `http://www.openssl.org/docs/`, 2012.

[32] J. R. Larus and R. Rajwar, *Transactional Memory*, 1 ed. (Morgan & Claypool Publishers, 2007).

Glossary

BAR or base address register is a configuration register in the configuration space of a PCI device where the access mode, size and address of device memory is configured. Sometimes BAR or BAR area is used synonymous with the device memory area configured by the BAR. 9, 19–21, 34, 36, 44, 52, 63, 64

ChipScope is a debugging tool providing a virtual logic analyzer which can be used to sample the states of internal signals of a FPGA. 30, 53, 56, 57

cold-boot is an attack type on cryptographic systems where keys stored for use in main memory are obtained by an attacker. The attacker is given physical access to a running computer system employing encryption with keys in use stored in RAM. The attacker cools the RAM modules, preventing the thermal decay of the stored bits, physically extracts them or reboots the system into a forensic software package and reads the preserved keys which are still available in RAM. 10, 67

CplD or completion with data is a PCI Express TLP type that is used to return data e.g. for MRd requests. 15, 38, 41, 44, 46

DMA or Direct Memory Access is a method for devices on DMA-capable bus systems like PCI Express to issue read and write requests for addresses in RAM. 9, 10, 12, 18–21, 34, 35, 38, 39, 44, 45, 47, 49, 50, 52–54, 57–59, 63, 64, 66, 67

DW or doubleword denotes 4 byte or 32 bit of data and is a frequently used unit in the PCI Express specification and literature. 14–16, 42–44, 52–54, 57–59

FPGA or Field Programmable Gate Array is a configurable matrix of boolean logic elements that can be programmed to create arbitrary user-defined chips. Main advantage is the very short reconfiguration time measured in seconds, as opposed to months for the production of chips by traditional means. See also the definition on page 23. 7–10, 23–25, 28–31, 33–37, 44, 47, 50–53, 63–67, 73–75

HDL is a language to create hardware designs. Common examples are VHDL and Verilog. 24, 25, 27, 28, 74, 75

IP core or often just «core» is the common denomination for modules that can be included into hardware designs. «Soft» and «hard» variants refer to the implementation of the IP core as HDL software or «real» hardware. IP in this context stands —somewhat misleadingly— for intellectual property, refering to the usually needed license for the use of an IP core. 8, 9, 12, 29–31

JTAG is a commonly used interface standard for programming and debugging chips such as FPGAs. 9

MRd or memory read, mem rd is a PCI Express TLP type used to request the data contents of a memory location. 15, 38, 46, 73

MWr or memory write, mem wr is a PCI Express TLP type used to write data to a certain memory location. 14, 15, 46

PCI or Peripheral Connect Interface is a parallel computer bus system, employed in many older personal computer systems in the last two decades. Variants include PCI-64 and PCI-X, succeded and now mostly replaced by PCI Express. 11, 13, 18, 19, 30, 34, 64, 73, 74

PCI Express is a computer bus system available in virtually all modern personal computers, based on the transmission of packets over a switched network of serial links. Successor to PCI. See also chapter 2. 8, 9, 11–15, 17–21, 29, 33–36, 38, 39, 41, 42, 44, 51–53, 63, 64, 67, 73, 74

PCI-ID is an identification number assigned to each PCI or PCI Express device at boot time. PCI-IDs are unique on a given system and are derived from the position of a device within the bus topology. See also page 16. 16, 34, 52, 53, 58, 64

root complex is the root node of the PCI Express network tree. It acts as gateway for requests from the CPU to PCI Express devices and from devices to RAM, among other functions. 12, 39

rst is a common shorthand for «reset». 43, 46

TLP is a packet on the transaction layer of the PCI Express bus where they are the primary carriers of all bus transactions like reading and writing data. 11, 13–17, 30, 35, 36, 38, 41–47, 52, 53, 56–58, 73, 74

Verilog is a common HDL. Developed by Phil Moorby and Prabhu Goel, later standardised by the IEEE as IEEE1364. 24, 26, 74

VHDL is a common HDL. Developed for the VHSIC (very high speed integrated circuit) initiative of the US Department of Defense and standardised by the IEEE as IEEE1076. 24–27, 42, 74

we is a common shorthand for the ≪write enable≫ bit that indicates whether a transaction or request should read or write. 46

Wishbone is a on-chip bus system used to connect different components e.g. in FPGA designs. See also section 3.5. 9, 29–32, 38, 45, 47

Postface

Danksagung

Ich bedanke mich sehr herzlich bei meinen Betreuern, Wosch und Michael, und bei allen die mir durch Korrekturlesen, Inspirationen und Tips geholfen haben, Kathrin, Karin, Metty, Rainer, und ganz besonders Morgan und Steffi.

Typeset with LaTeX
This version is built from git revision 890e893d1872967f310f16e90c033031ab39a457