DIPLOMA THESIS

A Hypervisor Layer for Virtualization of a System-on-Chip

A Virtualization extension for Network-on-Chip

Submitted at the Faculty of Electrical Engineering and Information Technology,
TU Wien
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ. Prof. Dipl.-Ing. Dr. techn. Axel Jantsch
Dipl.-Ing. Dr. techn. Martin Pongratz

Institut of Computertechnik Technology (E384)
TU Wien

by

Elvin Sebastian
Matr.Nr. 0825309
Kliviengasse 78, 1220 Wien

1. Oktober. 2016 _____

Abstract

This work presents a novel concept for a hardware-based virtualization solution, to provide spatial seperation between multiple application on a Cyberphysical System-on-Chip. This Cyberphysical System-on-Chip is able to sense its underlying substrate and adapt to degradation effects in it. The presented virtualization solution will support the Cyberphysical System-on-Chip to adapt its architecture, to meet the performance requirements of its applications, by providing the necessary means to reloacte applications to processors, which can provide the performance. The solution introduces a paging approach within a Network-on-Chip interface, known from Memory Management Units in processors, and a configuration scheme, which transforms the network to a virtualization layer within the chip. This approach will provide the means to virtualize entire operating systems and a hardware partioning scheme to run them.

Kurzfassung

Diese Arbeit präsentiert ein neues Konzept einer hardwarebasierter Virtualisierung, für die Umsetzung eines Cyberpyhsical System-on-Chip, dass in der Lage ist den physikalischen Zustand seiner Funktionseinheiten zu messen. Dieses Konzept bettet mehrere Funktionsblöcke, die teilweise auf dem Konzept einer Memory Management Unit basieren, in allen Network-on-Chip Schnittstellen des Cyberphysical System-on-Chip ein. Dieser Ansatz wandelt das gesamte Network-on-Chip, unter der Ebene der Memory Management Units in den Prozessoren, in eine zweite Virtualisierungsebene um. Diese Ebene ermöglicht Hardwarepartionierungen durchzuführen und gesamte Betriebssysteme zu virtualisieren. Das gibt dem Cyberpyhsical System-on-Chip die Möglichkeitje nach dem Zustand des darunterliegenden Substrats, Applikationen auf verschiedenen Prozessoreinheiten zu verteilen um den Leistungsanforderungen der Applikationen gerecht zu werden. Die Arbeit liefert zusätzlich Mechanismen diese Virtualiserungsebene zu konfigurieren und einen modularen Aufbau der Funktionsblöcke um die Austauschbarkeit/Erweiterbarkeit zu gewährleisten.

Ich danke an dieser Stelle meinen Eltern und widme die Arbeit meiner Schwester.
Ein spezielles Danke gilt auch für Andrej Hanic für seine Hilfe.

Table of Contents

Abbreviations

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Adanved Peripheral Bus
AXI	Advanced eXtensible Bus
CAN	Controller Area Network
CMOS	Complementary Metal-Oxid Semiconductur
COTS	Commerical off-the-shelf
CPSoC	Cyberphysical System-on-Chip
CPU	Central Processing Unit
DCR	Device Control Register (Bus)
DDR	Double Data Rate
DLC	Data Link Control
DMA	Direct Memory Access
DSP	Digital Signal Processor
DSU	Debug Support Unit
FIFO	First In - First Out
FPGA	Field Programmable Gate Array
HCI	Hot Carrier Injection
IMA	Integrated Modular Avionics
I2C	Inter-Intergrated Circuit
IP	Intellectual Property
I/O	Input/Output
MAC	Medium Access Control
MAGIC	Malicious Aging In Circuits/Cores
MSI	Message Signaled Interrupt
NBTI	Negative-Bias Temperature Instability
NI	Network Interface
NoC	Network-on-Chip
OCP	Open Core Protocol
OPB	On-Chip Peripheral Bus
OS	Operating System
OSI	Open System Interconnect
PCB	Printed Circuit Board
PCIe	Peripherial Component Interface Express
PLB	Processor Local Bus
QoS	Quality of Service
RAM	Random Access Memory
ROM	Read-only Memory
RTL	Register-Transfer-Level
SPI	Serial Peripheral Interface

SoC	System-on-Chip
SR-IOV	Single Root I/O Virtualization
UART	Universal Asynchronous Receiver Transmitter
VLSI	Very-Large-Scale-Integration
VM	Virtual Machine
VMM	Virtual Machine Monitor

1 The Cyberphysical System-on-Chip

The Cyberphysical System-on-Chip is a novel concept of a self-aware system-on-chip, which is able to sense its underlying substrate and act according to its condition. The need for such a chip will be explained in the following chapter and how this work will support to achieve this goal.

1.1 Motivation

Many advances in science, economy and society in the last decades are due to the fact of ever increasing computing power at every person's disposal. These continuous increases in performance are possible, because of architectural improvements and the doubling of integration density every few years described by the famous Law of Moore [Moo65]. This increase enabled VLSI-Systems (Very-Large-Scale Integration) with increased clock frequencies for digital circuits, to a point where thermal and power constraints became major limiting factors. This cap in clock frequency was solved by power efficient architectures and high scale parallelization. The rising availability of chip IPs (Intellectual Property), which are blueprints for subsystems, allows companies to design their customized modular systems and provide means for reusability of those subsystems. Today the number of IP subsystems like hardware accelerators and I/O interface controllers on a chip are increasing. Because all of these components were independent components outside the chip before, these systems are often referenced as System-on-Chip or SoC. This capability for companies to design their SoCs and give them into production at foundries, promoted new business models and products like the ThunderX Processor of Cavium [Gwe14] depicted in Figure 1.1, which was designed to handle high server loads with up to 48 processing cores and many hardware accelerators for encryption and compression. The ThunderX uses the most prominent example how IPs are changing the semiconductor industry. It uses processor architecture of ARM Limited, which is currently dominating the mobile market and is gaining more ground in other markets due to the fact, that their processors can be licensed for SoC applications and it can run Linux.
System-on-Chips enable a lot of flexibility, because it is possible to integrate many subsystems to offload tasks from the application processor. Offloading increases performance and reduces power consumption since subsystem can be optimized for its specific application instead of being run on a general purpose processor. [FL11] gives a good overview of SoC-Design and discusses some example of processor offloading like AES-Encryption (Advanced Encryption Standard), 3D Graphics or simple Audio/Video Compression. Subsystems can have their independent firmware

or micro-kernel, like the proprietary REX OS which manages the Baseband subsystem on Qualcomm mobile SoCs, and they provide a high-level interface for the application processor to access them. Because of the number of subsystems and their independent application, modern SoCs can be seen as miniaturized distributed systems.
The increasing amount of subsystems are causing challenges for floorplanners in interconnecting them and layouting a system bus through a chip while maintaining timing constraints and minimizing clock skew. The increased number of subsystems on a chip also lead to rising load on interconnects to a point where the cost of communication increased relative to the cost of computation in the last years. This challenge pushed researchers towards a Network-on-Chip concept more than a decade ago and is currently becoming more and more popular in the industry out of necessity. Network-on-Chip (NoC) communicates in segments with various topologies. Data will be routed through the chip via routers as endpoints of segments like in the all familiar IP (Internet Protocol) networks. This concept reduces the amount of wire routing effort enormously and increases the performance of a SoC.

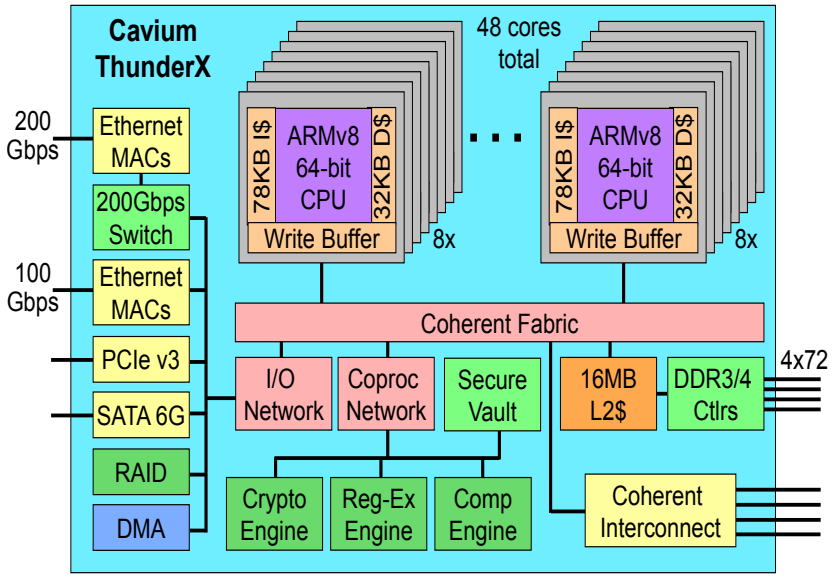

Figure 1.1: Cavium ThunderX SoC-Architecture Source: [Gwe14]

While increasing miniaturization provided space for more subsystems, new problems have arisen in the building blocks of the modern semiconductor industry. Since transistor sizes are currently decreasing by half every three years, effects like Negative-Bias Temperature Instability (NBTI), electromigration and hot carrier injection (HCI) [SWV+09] are dramatically influencing the reliability of CMOS technology. Because the size of the transistor is becoming smaller, these degradation effects affect the transistor faster. These aging effects are being noticed in a performance decrease of the processor or subsystem. NTBI affects the switching speed of pMOS-Transistors while HCI affects nMOS-Transistors. There is currently research into software patterns to accelerate aging of the underlying silicon by exploiting processor and pipeline architectures [KKW+15]. These so called MAGIC attacks (Malicious Aging in Circuits/Cores) can be used for research purposes, but also as malware. There are efforts to delay aging by load balancing in multi-processor systems. One attempt is to develop a self-aware SoC, also called a Cyber-Physical System-on-Chip (CPSoC) [SDG+15]. The CPSoC can sense the state of its substrate and act upon it by adjusting frequency and voltage of its subsystems and network-on-chip

routers. This capability would require the ability to adapt to changing conditions for applications, operating systems, interconnect architecture and hardware architecture during runtime and requires extensive research.

1.2 Problem statement

This thesis will provide a starting point which is needed to explore the implementation options for a Cyberphysical System-on-Chip (CPSoC). A System-on-Chip, in general, can be divided into several clusters. Each of them can be occupied by a single or multiple subsystems, which can be Intellectual Properties (IP) of different vendors. Within a cluster there can be an IP compatible interconnect to connect the local subsystems, while a different interconnect, most likely Network-on-Chip, can be used as a global interconnect like displayed in Figure 1.2. The focus will be on a controlled access of resources over Network-on-Chip, for a consistent address space within the CPSoC by exploring ways to establish and manage logical connections between single subsystems. The applications running on the clusters, which can be bare-metal code or an operating systems, should not require modification to be executeable on the target design.

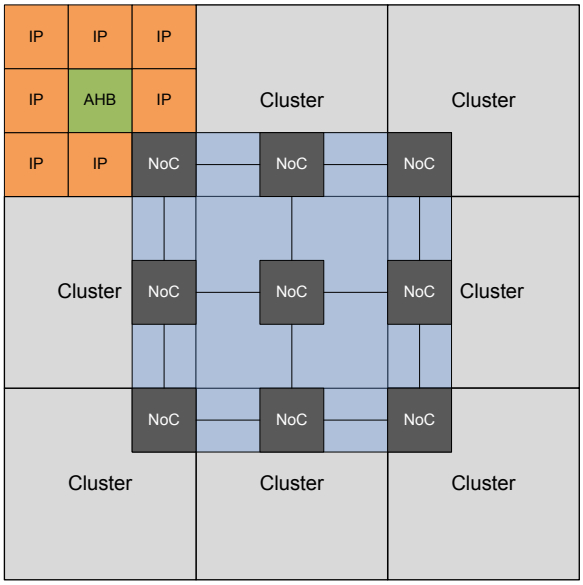

Figure 1.2: Cluster and Interconnect Schema

The GRLIB from Gaisler Inc. is a library with a collection of IPs and many preconfigured FPGA designs (Field Programmable Gate Array). An AMBA AHB (Advanced Microcontroller Bus Architecture - Advanced High-Performance Bus) to interconnect the IPs is included in the library. Another valuable IP is the LEON3 processor, based on the SPARC v8 architecture, which is the prefered architecture in aerospace application besides the PowerPC architecture. With the provided development tools like compiler and debugger, the GRLIB environment provides all prerequisites for exploring first ideas and concepts regarding the CPSoC. The gained insight of this work should provide direction for future works. The next section will provide a overview of considerations, which were the starting point of this work.

1.3 Related Work

The Cyberphysical System-on-Chip is a novel architectural concept for future SoCs. Therefore there are only few or no related works until now. However, since the CPSoC involves many partial aspects, which are already relevant in today's SoC architecture, papers to each of those aspects can be found. Aspects like Network-on-Chip interconnects, hardware-based virtualization, hardware partitioning and much more, are topics which have to be considered in the overall design of the CPSoC. Aguiar et al. [AH10][AMSH12] gives a good overview of the general topic of virtualization of embedded systems. They introduced a hardware modification to a MIPS processor architecture to reduce hypervisor overhead. The MIPS architecture uses both memory virtualization and fixed mapping for different parts of the memory even in Kernel Mode of the Operating System (OS). Some parts can not be reached in User Mode. The challenge for achieving full-virtualization on a MIPS processor is, how to run a GuestOS without direct access to priviledged instructions and to the Translation Lookaside Buffer (TLB) of the Memory Management Unit in Kernel Mode of the OS. To achieve the goal Aguiar et al. removed the restriction on the address space in User Mode and turned off the TLB in Kernel Mode of the processor, which is is occupied by the hypervisor. The OS will be pushed into User Mode. The TLB will then be used by the hypervisor to redirect access from the exception handling routines of the GuestOS to that of the hypervisor. These exception handling routines will then be called whenever the OS uses priviledged instructions. The presented solution allows to run multiple GuestOS on a single processor, but it requires modification of the processor and a hypervisor to schedule the Operating Systems. Because the processor has to be shared in this approach, it is not recommended for safety critical systems.

The works of Kliem and Voigt with [KV12][KV13] introduced a SecureBridge concept based on the GRLIB library, for partitioning the system memory to provide spatial separation between multiple operating systems. The work introduced a system with multiple processing subsystems, each capable of running an independent operating system. The external memory controller can only be reached by a backbone, which is separated by a bridge from the subsystem. This SecureBridge monitors the accessed memory addresses and enforces spatial separation of the subsystems. The bridge has different implementations to test performance. One implementation has combinatory logic with a direct access to the backbone, once the master interface on the backbone side of the bridge is arbitrated. Only the address is manipulated by the SecureBridge during the access. The other implementation has a cache and an optimizer integrated to enhance overall system performance. The optimizer has the responsibility to optimize the access to the backbone by utilizing the whole data width of the backbone. This implementation has latency in the bridge but improves the overall performance. The work shows that the system is not scalable with a single clock domain without badly influencing the clock frequency of the system. Therefore it is suggested to implement multiple clock domains for the subsystems and the backbone and the SecureBridge had to be adapted, because it would interface with two clock domains. This is now managed by asynchronous FIFO queues for input and output. The concept to enforce spatial separation at the boundary of one logical component to another is quite popular, because it is easier to implement at the source of a transfer instead at the destination and it removes avoidable traffic between them. The downside of this concept is, that the processor itself can not be virtualized, because the focus is only on spatial separation and not on temporal separation of applications. Since no temporal separation is required, there is no need for a hypervisor intervention in the subsystem.

The source of transfers is not always a processor, it can also be a DMA capable (Direct Memory Access) I/O. Münch et al. provide a hardware-based I/O virtualization solution for PCIe (Pe-

ripheral Component Interconnect Express) [MIM+13][MPHH15][MPH15]. It utilizes the Single Root - I/O Virtualization (SR-IOV) feature of the PCIe standard to share a physical I/O to multiple virtualizedd OS. This feature provides multiple virtual functions to access the physical hardware. These virtual functions are virtual interfaces, which can be mapped into the address space of the virtual machines. To isolate I/O DMA access to their respective virtual environment an I/O Memory Management Unit (IOMMU) can be used in the PCIe controller, but Münch et. al decided to implement a non-transparent bridge (NTB) close to the I/O. Firstly IOMMUs are not common in embedded system components as they are in IT server technology and secondly the NTB has less complexity and is more predictable. This is especially necessary for safety critical applications like avionics. The NTB acts as a firewall to the traffic coming from the I/O and from the PCIe controller and can be only configured with a limited amount of rules, which the traffic has to comply.

The same firewall concept is applied by Grammatikakis et al. with [KGC12][GPP+14]. They introduced a virtualization and a security concept for architectures based on a Network-on-Chip. In [KGC12] the focus is on a fully virtualized hardware with the help of traditional MMUs and IOMMUs. The system is comprised of multiple processors and accelerators, which are connected by a Network-on-Chip. Between the accelerators and the NoC a Command Processing Unit (CPE) is used to regulate access to the hardware. This CPE has an internal TLB cache for translating virtual addresses to physical addresses, an interrupt unit for signaling an access violation to the hypervisor and a monitoring unit. The main purpose of the CPE is not only sharing various resources among virtualized environments, but also for protection of each environment from malicous hardware drivers or processes on other environments. An impressive design feature of the CPE is, that differentiation between multiple sources of access and individual translation is possible. But all this comes with increased complexity of the design. [GPP+14] contributes a NoC-Firewall, which acts as a gatekeeper to the Network-on-Chip. The firewall has less complexity than the CPE with its IOMMU features. To provide a lightweight solution, the firewall implemented a segmentation-based approach into the NoC interface. Segementing requires more logic than paging for calculating segements/pages, but size of a segmentation table can be smaller than of the size a page table. The work showed that the network delay can be reduced up to 50% in an environment with malicous memory access, due to reduced traffic. The test was performed with various numbers of processors and memory controller, each with their own network access.

SoC virtualization is about mapping applications to processors. Multiprocessor SoCs are systems, which may not have to support an operating system because of limited processing architecture features or memory. An example is a system comprised of several Digital Signal Processors (DSP), which are not OS capable processors. This circumstance means, there is the necessity of finding other ways for a SoC to schedule its tasks. [Bie14] approaches a solution to this problem by implementing a hardware-based virtualization layer between processor and storage. The virtualization layer consists of an interconnect and a VirtBridge for each application. The VirtBridge has the responsibility of saving the application context in case of a task interruption. There are hardware-based means implemented to interrupt an executing application and save its context by injecting code into the instruction stream of the application. These injected code will flush the pipeline and save the programm counter, register data and more. This context will then be stored in a specific memory of the VirtBridge and kept there until the application is assigned a processor again. Then it is restored again by injected code. This processes it totally transparent for the application. This solution provides an unique hardware-based temporal separation of applications and was proved on a Xilinx MicroBlaze processor.

The presented works inspired the design of the proposed solution in this work. The approach of

Kliem and Voigt with the SecureBridge and the NoC-Firewall approach of Grammatikakis et al. were chosen as a reference. The SecureBridge design was interesting because it used the GRLIB and showed the potential of the library. The NoC-Firewall design of Grammatikakis et al. was the prefered design choice, because the target design of this work uses a Network-on-Chip. But unlike Grammatikakis, the target design uses a paging mechanism with a reduced page table size for less complexity. The works of Münch et al. provided insight in DMA transactions, but their focus is mostly on PCIe. Because of the provided awareness for DMA, the target solution in this work also provides isolation of DMA transfers. The other works mentioned provided better understanding of the topic, but were not considered at this point since focus was on spatial separation instead of temporal separation. Once more, the mentioned papers here are just an excerpt of works about virtualization. Even the authors mentioned before have more papers to offer, which can contribute for a much deeper understanding of the topic. Especially Grammatikakis has several papers worth reading.

2 Overview on System Communications

Interconnects are functional units that provide the means for data exchange between many components in a system and therefore are essential for the capabilities of a system. The following chapter gives an overview of interconnect characteristics and established standards. It will provide a good basis to compare the State-of-the-Art with the emerging Network-on-Chip paradigm in system interconnection.

2.1 System Communication Characteristics

System communication can be implemented in many ways to improve performance, power efficiency and area requirement. Depending on the application type and the given design constraints, it is important to choose between different implementations. This section provides an overview on interconnect characteristics when selecting an implementation for a system.

2.1.1 Signal Types

Signal types involved in system communication can be broken down into three types. These types are address, data and control signals. These signals are typically exchanged between a master and a slave. The master initiates a transfer to a slave and provides all the necessary address and control information for the slave to execute the transfer. The flow of data in a transfer is dependent on a control signal, which indicates if it is a read or a write. In the case of a write, the data is provided by the master. Otherwise, the data is provided by the slave. There can be different channels implemented for read and write with their respective address and data signals to improve performance. The required amount of wires is determined by the physical structure and the data transfer modes of the interconnect.

2.1.2 Physical Structure

The physical structure of an interconnect influences its capabilities. The most commonly known structure is the shared bus as depicted in Figure 2.1. All masters and slaves share a common bus for exchanging information. All masters can write on the address and control bus while the data bus can be accessed by masters and slaves to share data independent of a write or a read transfer. Shared buses utilize tristate buffers to switch between read, write and high impedance.

They are commonly used for off-chip communication because they reduce the amount of wires involved. The downside is the power consumption of the bus and the tri-state buffers. For on-chip communication, an AND-OR bus depicted in Figure 2.2 or a multiplexed (MUX) bus shown in Figure 2.3 is preferred, because of the reduced power requirements.

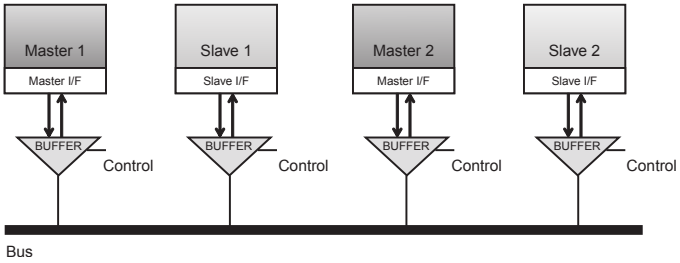

Figure 2.1: Shared Bus Interconnect. Source: [PD08]

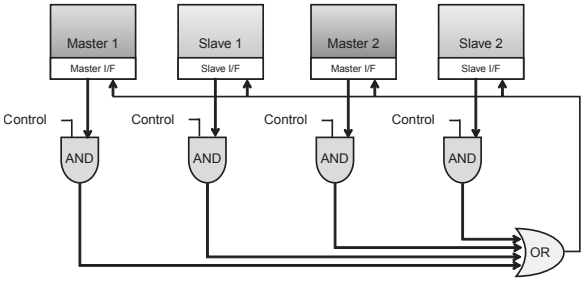

Figure 2.2: AND-OR Interconnect. Source: [PD08]

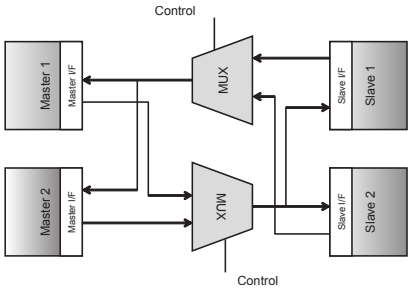

Figure 2.3: MUX Interconnect. Source: [PD08]

Beside masters and slaves, there is an arbiter and a decoder involved in the communication. The arbiter is responsible for scheduling the access of multiple masters to the bus, while the decoder selects the designated slave of a transfer. These selects have dedicated signals for every member of a bus. In MUX and AND-OR buses the decoder and the arbiter also control the multiplexers or the AND-Gates. The decoder and the arbiter can be centralized but also distributed, directly at each master or slave. For example CAN bus (Controller Area Network) has its arbitration protocol, which implements a priority access scheme.

2.1.3 Data Transfer Modes

There are many modes of transfer to increase the performance of system communication. The goal is to eliminate idle states of the interconnect to maximize the throughput. Therefore arbitration of masters is a key factor and many modes of transfers are implemented in today's interconnects.

Pipelined Transfer

One of these modes is the pipelined transfer. In its basic form, a transfer can be divided into phases, an arbitration phase, an address phase and a data phase. In the arbitration phase, a master, which requests access to the bus, will be granted by the bus arbiter. This grant can be seen in Figure 2.4 between T3 and T4. Once the access is granted the master will provide all the necessary information in the address phase between T4 and T5 like address, if it is a read or

write data transfer, and other information depending on the implementation. After that, the data phase between T5 and T6 delivers the actual data and responses, while the address phase for the next transfer has started in parallel. Possibilities here are the address phase of the next transfer of the same master or a new transfer of another master. During the address phase arbitration can be given to the next master in line. In a non-pipelined interconnect a transfer has to be finished before the next master is granted access to the bus and the transfer has to start from the arbitration phase.

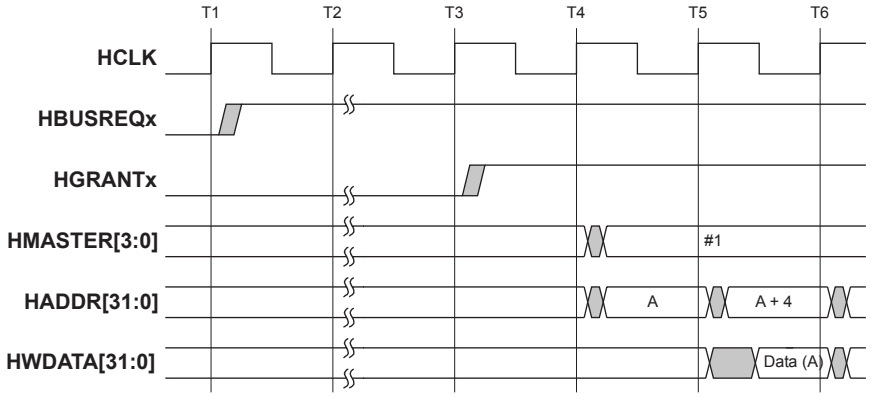

Figure 2.4: Pipelined transfer in AMBA AHB interconnects. Source: [ARM99]

Burst Mode

It is important to reduce the time for data fetches to reduce idling times of masters, especially when a big amount of data is accessed. In this mode the master does not request arbitration for every single transfer, but requests arbitration once and keeps the bus access grant until a burst is over.

Split Mode

It is possible that a slave can't serve a request immediately and the master has to hold until the slave is ready. This request will block any other masters from performing their requests and decreases performance. A split transfer mode can be introduced, which will be issued by the slave to eliminate this bottleneck. The split indicates the arbiter of the bus to release the master and give access to another master until the slave indicates the arbiter to re-engage the held master.

Out-of-Order

The split transfer mode increases the overall performance of the bus but still leaves room for improvement. While a master is split from the bus, it idles and can't perform any further operation until the transfer is resumed. Out-of-Order access can further improve performance in such cases. It would allow a split master to access other slaves until the first transfer can be continued. Unlike the split transfer, this technique would require the master to support such Out-of-Order execution capabilities to be able to guarantee data consistency.

2.2 On-Chip Communication Standards

In times of decreasing Time-to-Market for semiconductor products, it has become essential to buy IPs and customize a product according to individual requirements. The actual degree of freedom for customizability is provided by open interconnect standards. Most of the state-of-the-art interconnect standards were developed by vendors of processors. These interconnects were publicly available so vendors of specialized subsystems would adapt them to make the subsystem compatible with the processor. An increasing amount of supported subsystems strengthens the processor on the market. This section will discuss the most common open interconnect standards.

2.2.1 AMBA

The Advanced Microcontroller Bus Architecture was one of the first open interconnect standards provided by the Advanced RISC Machine (ARM) Limited. Its second revision of the standard became quite popular when it was released in 1999 and is still being used today. It comprises of the Advanced High-Performance Bus (AHB) for increased performance and the Advanced Peripheral Bus (APB) with low-power focus. The standard is a MUX bus and has introduced a pipelined bus with burst and split capabilities. It supports a 32-bit address width and a data width of 8 bit up to 1024 bit[1]. A schema of AHB is depicted in Figure 2.5 and consists of an arbiter and a decoder for controlling the communication between masters and slaves.

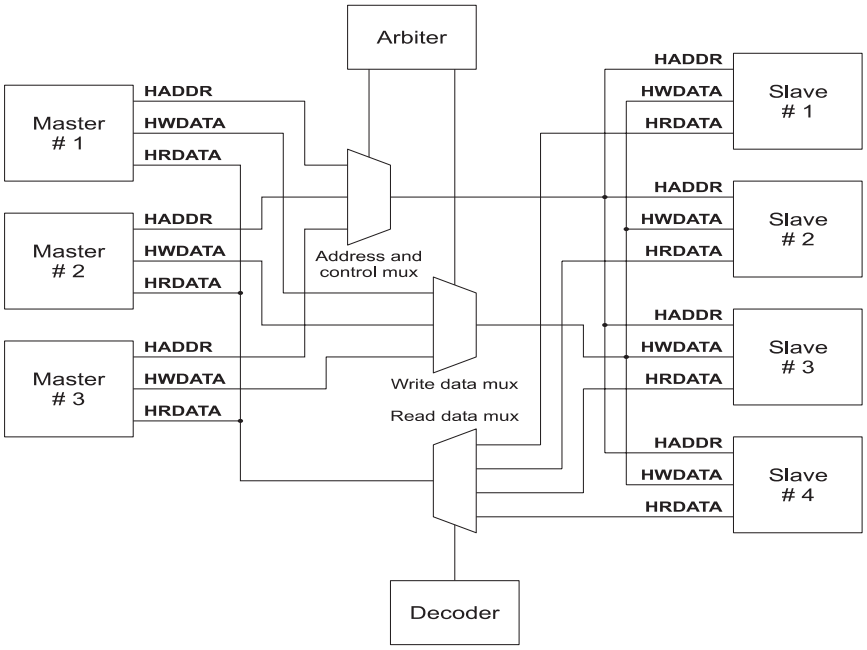

Figure 2.5: AHB interconnect schema. Source: [ARM99]

The arbiter has to guarantee that the multiplexer from the master to the slave is correctly set in the arbitration phase, so the master can provide the necessary information for the transfer during the address phase. The decoder uses the information provided by the master in the address phase to set up the multiplexer for returning data from the slave to the master in the data phase. For

[1] it is recommended to keep it between 32 bit and 256 bit

better visualization take a look at Figure 2.4. Because of the pipelined architecture, a single cycle bus handover to other masters can be provided. APB is an extension of AHB with reduced complexity for easy interfacing with the interconnect. A bridge is necessary to connect the APB slaves to the AHB as shown in Figure 2.6. The bridge will deal with the more complex AHB protocol and provides the transfer data on a simpler interface to the APB slaves. Besides this standard topology with one active master and one active slave on the same AHB bus, there is also a matrix topology, which allows multiple masters to access different slaves at the same time. Figure 2.7 shows a partial matrix topology. A full matrix topology would only have slaves on the right side and all units would be connected directly to the matrix, instead of sharing connections to the matrix. Each matrix port for masters would have an input stage with a decoder for selecting the slave and each matrix port for slaves would have an output stage with an arbiter for selecting the master. This setup increases the performance of the whole interconnect due to concurrently access for multiple masters.

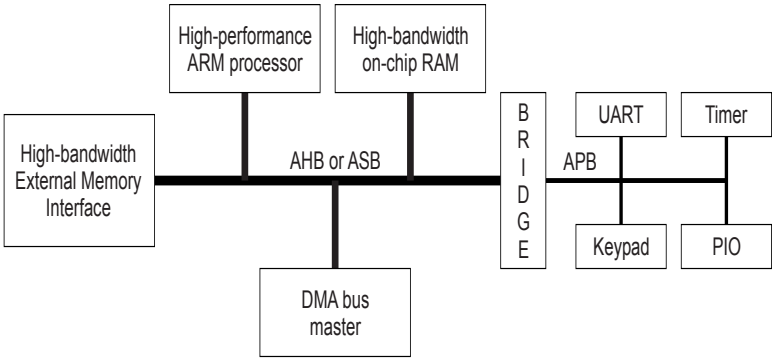

Figure 2.6: AMBA Interconnect Hierarchy. Source: [ARM99]

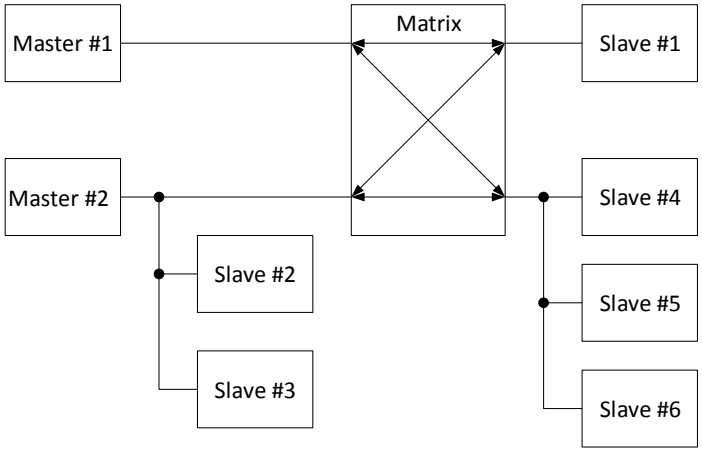

Figure 2.7: Partial Matrix AHB interconnect schema.

Revision 3 of the AMBA standard brought a new interconnect called the Advanced eXtensible Interface (AXI) with improved architecture and functionality. One improvement is the implementation of a channel based communication. Five channels are defined, a read address channel, a read data channel, a write address channel, a write data channel and a write response channel. Each channel possesses individual handshakes. The read and write channels can operate concur-

rent and each of them supports out-of-order execution. Therefore each transfer is assigned an ID. Among transfers with the same ID the order has to be kept, but the order of different IDs can be changed to improve bus utilization. For burst modes, only the start address has to be provided and the slave will calculate the increment. In AHB this was not the case and the address has to be provided for each increment. This improvement means the bandwidth requirements for the address channels is lower than for the data channels for AXI burst transfers. Also, AHB supported only incremental and wrapping bursts while AXI supports fixed bursts for accessing FIFOs (First In - First Out) of memory mapped I/Os. This burst will continuously access the same address. AXI also supports the matrix topology as AHB does for improved performance.

2.2.2 IBM CoreConnect

The IBM CoreConnect was developed at the same time as the AMBA standard, but it was not open until 1999. It was used as the interconnect standard for IBMs PowerPC architecture. The CoreConnect standard comprises of 3 buses. The Processor Local Bus (PLB) for high performance, the On-Chip Peripheral Bus (OPB) for large number and low-bandwidth peripherals and the Device Control Register (DCR) bus for low-bandwidth access to registers. These buses are specified to be AND-OR interconnects, where each unit interfaced to the bus controls its own enable signal for the AND gate or have to guarantee that their output is set low while they are inactive. PLB was the direct contender to ARM AHB and supported a data width up to 256 bit, burst mode, split transactions, separate read and write bus as well as pipelining. Also, it is possible to assign priorities (4 groups) to masters to implement a variety of arbitration schemes. One major downside of PLB is the number of members on the bus, directly influences bus clock because of increased wire load. Therefore it is required to reduce the number of members on PLB and offload slaves to OPB. DCR is being used for configuration purposes of slaves even when the PLB is overloaded. Figure 2.8 shows a configuration example of CoreConnect. DCR is arranged in a Daisy Chain there, but can also be arranged as a point-to-point connection if required.

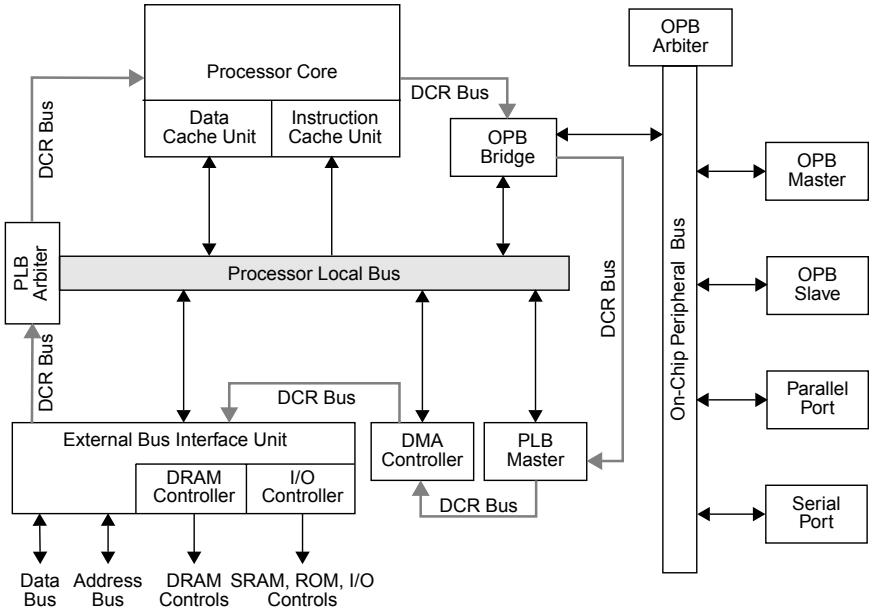

Figure 2.8: IBM CoreConnect Architecture. Source: [IBM]

One major benefit of CoreConnect is, that OPB is capable of handling multiple masters and therefore can connect to various PLB with a PLB-to-OPB bridge. Another option would be to implement a bridge for direct access to the OPB domain and a Direct Memory Access controller between the same PLB and OPB.

2.2.3 Wishbone

Wishbone is a bus specification devised by Silicore Corporation and is now maintained by the OpenCores community. It is freely available and has been placed into the public domain with the intent to make the intellectual property free. The Wishbone interface has a simple design to make it viable for small embedded systems, but it also has the capability to be used for high-performance systems too. The bus has a synchronous design, but also needs combinatory logic for maximum performance. The bus supports 8-bit to 64-bit data width and burst transfer with a pipeline architecture. Wishbone defines a single bus, which can be instanced many times within a design to meet different performance and power requirements for each instance. The specification also leaves room for various topologies like shared bus, crossbar or pipeline depicted in Figure 2.9.

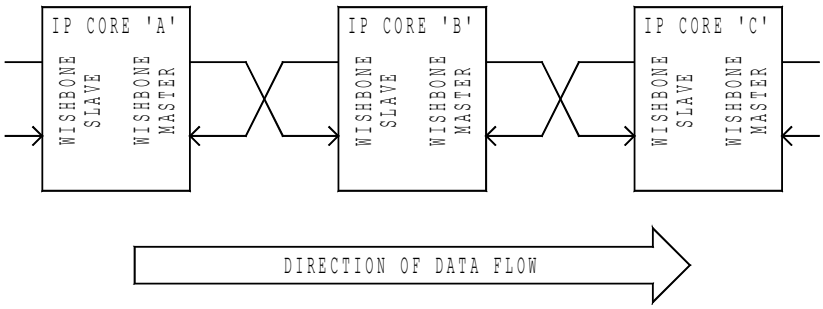

Figure 2.9: Wishbone Pipeline topology. Source: [Ope]

To accomplish a shared bus or crossbar topology, the arbiter has to pretend to the master it is the slave and stall transmissions until the bus is free. This is necessary because there are no arbitration signals defined. The bus is designed as a point-to-point protocol as depicted in Figure 2.10. Address decoding needs to be implemented locally in the slaves.

The Wishbone specification defined a tagging mechanism to add user specific signals to the interconnect. Each attached tag has to be associated with one of eight tag types which correspond to the eight basic Wishbone signals. It is required to document added tags in the Wishbone datasheet of each device manual delivered with the IP.

2.2.4 Open Core Protocol

The Open Core Protocol (OCP) is socket-based interconnect standard. It defines neither an architecture nor any components and only defines the interface between devices and the interconnect. OCP is also designed as a point-to-point protocol and allows many topologies. The OCP interface can be used as a wrapper for other buses as the ones previously mentioned. This capability makes it possible to use IPs and interconnects which are incompatible, by implementing

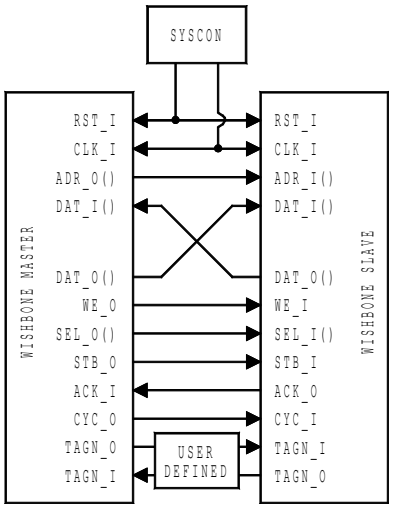

Figure 2.10: Wishbone standard connection and signals. Source: [Ope]

an OCP wrapper with a converter logic around the IPs and interconnects. OCP defines three categories of signals for data flow, sideband and test signals. Only a subset of dataflow signals has to be implemented to support a basic OCP interface while the others are for enhanced features like JTAG support in test signals or metadata in sideband signals. OCP also has profiles for its interfaces to meet different requirements for individual devices attached to the interconnect. The main profiles are Bridging profiles for components designed for other interconnects and Native OCP profiles for components designed for OCP. Each of these has subsets of profiles for single transfers, burst transfers and more.

2.3 Off-Chip Communication Standards

While more and more controllers and accelerators are being integrated into a SoC, there are still components outside the SoC which need to be interfaced. The best example is the RAM (Random Access Memory) with its DDR-Interface (Double Data Rate). While the controller for the interface is already integrated into most SoCs, RAMs will still be placed outside of SoC for a while due to their area costs. Because the primary constraint for off-chip communication is the pin count, which influences the packaging cost of a device, the preferred bus type is a bi-directional tri-state communication bus for parallel communication. To reduce the pin count further, it is possible to implement a uni-directional serial communication standards like PCI Express (PCIe). PCIe uses a tree topology with multiple lanes to communicate with the attached devices. Each lane consists of 2 pairs of differential RX and TX wires. All data is exchanged as packets, even interrupts in so-called Message Signaled Interrupts (MSI). Each packet contains a header with the destination address, so switches in the nodes of the tree can route the packets. The Central Processing Unit (CPU) accesses the PCIe endpoints over the address space of the Root Complex, which is the root of the tree topology. The Root Complex translates the address into the PCIe address and sends the packetized request to the PCIe endpoint assigned to the accessed address. PCIe drivers are required to set up the Root Complex, but are not needed for accessing the PCIe endpoints. Beside PCIe, which is used for high-performance communication, there are several well known low-bandwidth communication standards like I2C (or IIC for Inter-Integrated Circuit)or SPI (Serial Peripheral Interface). It is not uncommon to connect components in smartphones

via these protocols, and occasionally even UART (Universal Asynchronous Receiver Transmitter) is being used to utilize all the available resources efficiently, to reduce complexity and Printed Circuit Board (PCB) footprint.

3 Background Topics

Interconnects are the backbone of every SoC implementation. There are many different interconnect types. In this section the AMBA AHB (Advanced High-performance Bus) and Network-on-Chip (NoC) will be discussed to give a brief introduction to understand how to exploit the specification of those different interconnects to implement a bridging/tunneling concept. There will be an introduction to AMBA APB (Advanced Peripheral Bus) as well because it is used to connect low-bandwidth I/O to the system. At the end of this chapter, some extensions of the AHB specification in the GRLIB library of Cobham Gaisler will be discussed as well as a short introduction into system virtualization.

3.1 Network-on-Chip

There are several Network-on-Chip implementations for different applications. Many quality and performance considerations have to be investigated before designing a new or using an existing NoC design methodology. As previously explained the goal of a NoC is to reduce the cost of design and implementation of a global die interconnect. Therefore the interconnect is segmented into parts which are connected by switches. That means that sending data from one cluster to another can take several clock cycles on the interconnect. Because each of the segment is shorter in length than traditional buses, the capacitive and inductive load is smaller and each of them can be clocked faster. This switching approach also enables a whole new dimension of topologies with multiple routes from point to point for interconnects. These topologies can be categorized according to [PD08] into three groups:

- Regular Networks: network with symmetric topology
  - Direct Networks: each router has network entry/exit port for subsystems (see Figure 3.1)
  - Indirect Networks: not all router have network entry/exit ports (see Figure 3.2)
- Irregular Networks: application specific networks with asymmetric topology (see Figure 3.3)

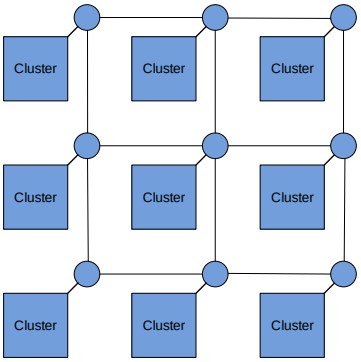

Figure 3.1: 2D-Mesh topology (direct)

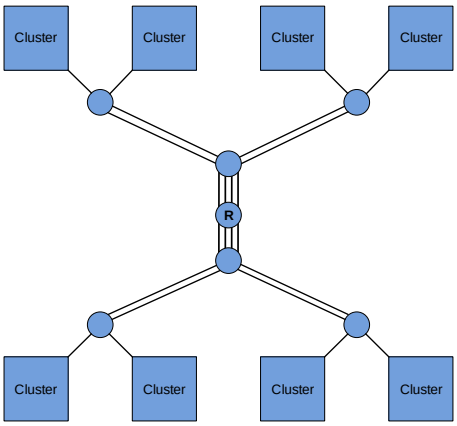

Figure 3.2: Fat-Tree topology (indirect)

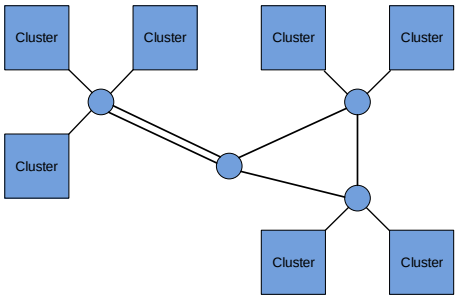

Figure 3.3: Irregular topology

Direct and indirect networks are subcategories of regular networks, which is beside irregular networks, the main category. Figure 3.1 and Figure 3.2 are examples of a 2D-mesh network, which is a regular one, and a fat-tree, which is an irregular one. The 2-D mesh topology is very popular because of easy hardware implementation. But it is prone to traffic accummulation in the center and that needs to be considered in the implemented routing algorithm. Before implementing a routing algorithm, it is important to determine the switching type of the network. There is the choice between circuit switched and packet switched networks. Packets consists of multiple Flits (Flow control digits) which can be divided into header, body and tail flit. There is only one header flit to indicate the beginning of a packet, multiple body flits for the actual payload and a tail flit to indicate the end of a packet. Circuit Switching establishes a logical connection between source and destination before sending any data. A header-flit is being sent to the destination and reserves any link (router to router) it traverses through. In case the header meets a link which is already reserved on the way, then the source will receive a NACK from that corresponding switch. If the header is acknowledged by the destination, the physical link is set up and body-flits of a message can be transmitted. After complete transmission, a tail-flit will be sent to tear down the connection. The benefit of circuit switching is the guaranteed bandwidth for an established link. The downside of this technique is the possibility of underutilized links. For Packet Switching the sent message is split into multiple packets, each with a head-, couple

of body- and a tail-flit like in Figure 3.4. Each of these packets has to go independently through the network and can share the same link with other packets from other sources. The Quality of Service (QoS) is harder to guarantee with this technique, but it enables to utilize 100% of a link if necessary. A NoC network has three schemes for packet switching, Store and Foward (SAF), Virtual Cut Through (VCT) and Wormhole (WH) switching. In SAF a router accepts only new packets if there is enough buffer space for the entire packet. A packet on the way will always be stored in one router. This process causes latency in the network because the packet is only passed to the next router when (a) in the next router, buffer is available for a whole packet and (b) the whole packet has been received by the transmitting router. VCT tries to reduce that latency by immediately transmitting incoming flits to the next router, if the next router has buffer space for an entire packet. Both SAF and VCT have big buffer requirements and therefore are costly for integration into a SoC. WH improves the cost by reducing the buffer requirements to a single flit. Like VCT the flits are immediately transmitted as soon as the next router has space for a flit. Therefore a packet can be stored on multiple routers along a path between source and destination in the network. The downside of this scheme is the increased congestion probability of links in the network when a path is occupied. If a different source requires a connection on the same path for its communication, it has to wait until the link is free again.

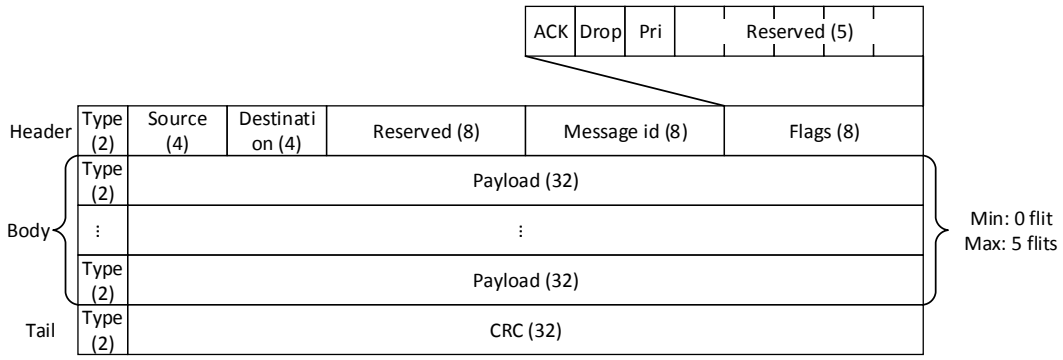

Figure 3.4: Flits aggregated to a NoC packet. Source: Junshi Wang

So far topologies and switching have been covered. But to get a clear structure of the NoC communication scheme, it is vital to implement a layered approach. Each layer provides a service for the upper one and all layers together provide the means of communication between entities. For that purpose, the OSI-Model (Open System Interconnect) is the most popular choice in literature. The model divides the communication into seven layers and each layer provides a service to the ones above them. These layers are in ascending order:

1. Physical Layer: Defines the physical requirements for the communication medium
2. Link Layer: Controls the access to communication medium
3. Network Layer: Ensures that packets are routed through the network to the destination
4. Transport Layer: Provides reliability and flow control for communications
5. Session Layer: Sets up and tears down logical connections

6. Presentation Layer: Provides context of data. Is responsible for data format conversion
7. Application Layer: Service implementation for users

Layer 7 is the closest to the user of a service. All other layers provide services to the ones above them to enable the service of the application layer. In many implementations of a layered communication system, the presentation and session layer is integrated into the application layer, which leaves in total five layers. For Network-on-Chip the focus will be on the first four layers. The physical layer concerns itself with the electrical, thermal and other physical constraints given by the manufacturing process. Also, the possible topology and floorplanning, provided by a design methodology, is part of the physical layer. The link layer is divided into Medium Access Control (MAC) and Data Link Control (DLC). MAC for NoC architectures is fairly simple compared to other standards. Because NoC is on link layer a point-to-point system, a router only has to wait until the receiver indicates it is free to receive. The switching technique in the network, which goes hand in hand with routing of packets in a NoC, determines the way the router arbitrates its ports and accepts new packets/flits. The DLC controls the correct transmission by error detection and can be implemented point-to-point or endpoint-to-endpoint. Point-to-point would mean each switch has to perform an error detection which is costly and would increase latency in the network. The network layer is all about the routing of packets through the network and implementing the application specific algorithm for that purpose. Figure 3.5 shows important considerations when implementing a routing algorithm.

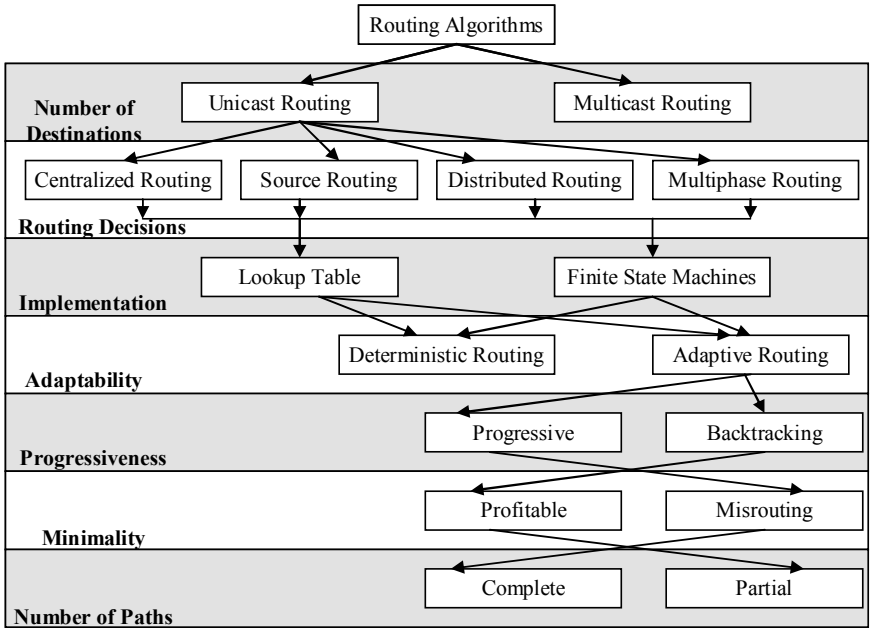

Figure 3.5: Routing algorithm characterizations. Source: [AIS09]

Transport layer has the responsibility to segment messages into processable sizes and regulate the flow between subsystems. The application layer for NoC is the task running in a subsystem to which the NoC is attached to. This layered approach will come useful to determine which service needs to be provided first to enable certain functionality/service for the whole system, which utilizes the Network-on-Chip to communicate.

3.2 AMBA Rev. 2 Interconnect

The AMBA Revision 2 provides specifications for three bus systems, the ASB (Advanced System Bus), the AHB (Advanced High-Performance Bus) and the APB (Advanced Peripheral Bus). This section will give an in-depth look into AHB and a brief look into APB, provided by [ARM99].

3.2.1 AMBA AHB Signals

The GRLIB chose the AHB interconnect standard to interconnect its IPs. As mentioned before, AHB is a bus with pipelined architecture and therefore has an arbitration, an address and a data phase. The involved signals will be discussed in this section.

HADDR[31:0]

Signalwidth: 32 bits
Direction: Master to Slave and Decoder
Phase: Address
Function: Important for decoder to select slave by setting HSELx and for slave to select data.

HBURST[2:0]

Signalwidth: 3 bits
Direction: Master to Slave
Phase: Address
Function: At the beginning of a new AHB Transaction this signal indicates the number of transfers being handled within the transaction. This doesn't indicate the amount of data being transfered. To calculate the data volume, HBURST transfer length has to be multiplied by HSIZE transfer size. The lengths are shown in Table 3.1.

HBURST[2:0]	Type	Description
b'000'	SINGLE	Single transfer
b'001'	INCR	Unspecified transfer-length
b'010'	WRAP4	4 beat transaction with address wrapping
b'011'	INCR4	4 beat transaction
b'100'	WRAP8	8 beat transaction with address wrapping
b'101'	INCR8	8 beat transaction
b'110'	WRAP16	16 beat transaction with address wrapping
b'111'	INCR16	16 beat transaction

Table 3.1: HBURST signals

HBUSREQx

Signalwidth: 1 bit (for each master)
Direction: Master to Arbiter
Phase: -
Function: Master x requests bus by setting signal to high

HCLK

Signalwidth: 1 bit
Direction: -
Phase: -
Function: Bus Clock Signal

HGRANTx

Signalwidth: 1 bit (for each master)
Direction: Arbiter to Master
Phase: Arbitration
Function: Arbiter grants Bus to Master x by setting signal to high

HLOCKx

Signalwidth: 1 bit (for each master)
Direction: Master to Arbiter
Phase: -
Function: Master x requests locked access to bus by setting signal to high. The transaction is locked until the master ends it

HMASTER[3:0]

Signalwidth: 4 bit
Direction: Arbiter to Slave
Phase: -
Function: Indicates Master ID to Slave and is bus internally used for multiplexing masters. For the slave it is important for continuing RETRY and SPLIT transactions explained under HRESP.

HMASTLOCK

Signalwidth: 1 bit
Direction: Arbiter to Slave
Phase: Address
Function: Indicates to slave that the running transaction is a locked access by setting signal to high

HPROT[3:0]

Signalwidth: 4 bits
Direction: Master to Slave
Phase: Address
Function: Master can use these signals to send metadata to the slave for the current transaction. These metadata can be access mode, data type and more as shown in Table 3.2.

	HPROT[3]	HPROT[2]	HPROT[1]	HPROT[0]
Low (0)	Not cacheable	Not bufferable	User access	Opcode fetch
High (1)	cacheable	bufferable	Priviledge access	Data fetch

Table 3.2: HPROT signals

HRDATA[31:0]

Signalwidth: 32 bits
Direction: Slave to Master
Phase: Data
Function: Sends reading data from Slave to Master. HRDATA, like HWDATA is segmented into 4 lanes with 8 bit width. Each of these lanes represent a single increment of an address as shown in Table 3.5. Which means AHB is able to be used by IPs that have different data width than a 32 bit e.g. UART has 8bit.

HREADY

Signalwidth: 1 bit
Direction: Slave to Master
Phase: Data
Function: Slave indicates Master to pause transaction and hold bus by pulling signal to low until signal is high again.

HRESET

Signalwidth: 1 bit
Direction: -
Phase: -
Function: Low-active reset signal

HRESP[1:0]

Signalwidth: 2 bits
Direction: Slave to Master
Phase: Data
Function: Slave responds to master indicating the acceptance of a transfer or other status as summarized in Table 3.3. Except for OKAY the indication of the other responses requires a two cycle response procedure starting in the data phase of a transaction. In the first cycle when setting up the response ERROR, RETRY and SPLIT HREADY has to be pulled to low, to signal the master to pause the transfer. This action prevents the master to finish the ongoing transfer when HREADY reaches the master and to continue to the next transfer. Instead, the master has to resolve the response procedure by idling the bus with IDLE HTRANS-signal. At the same time, the slave pulls HREADY to high while still maintaining HRESP. Mind that even if slave and master are setting signals on the bus for example at T2 in Figure 3.6, the signals will only be registered at the edge of T3 by the opposite side which means that a reaction to an action takes two cycles to return. Therefore 2-phase response procedure is necessary to flush the bus pipeline.

HRESP[1:0]	Type	Description
b'00'	OKAY	Transfer accepted / Slave ready
b'01'	ERROR	Transfer rejected
b'10'	RETRY	Slave interrupts transfer. Only higher priority master can use bus anymore
b'11'	SPLIT	Slave interrupts transfer. Any other Master can use bus

Table 3.3: HRESP signals

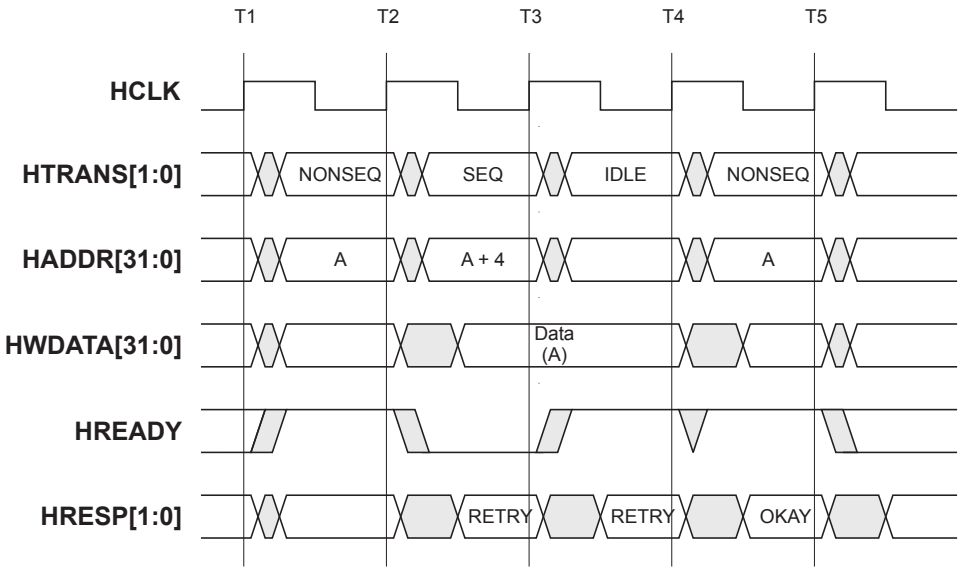

Figure 3.6: AHB 2 cycle response procedure. Source: [ARM99]

HSELx

Signalwidth: 1 bit (for each Slave)
Direction: Decoder to Slave x
Phase: Address
Function: Decoder selects slaves corresponding to HADDR by setting the signal to high. Decoder has to decode address before next rising edge so slave can accept Address Phase signals.

HSIZE[2:0]

Signalwidth: 3 bits
Direction: Master to Slave
Phase: Address
Function: Indicates the size of the transmitted data. In combination with HADDR it is needed to determine on which HWDATA or HRDATA lanes the data is transmitted. If HADDR doesn't align with HSIZE as shown in Table 3.5, the slave has to respond with an error. Also HSIZE can't be higher than the databus. This means not all HSIZE signals are always needed. HSIZE supports eight sizes starting with 8 bits (b'000') to 1024 bits (b'111') with power of 2 steps in between.

HSPLIT[15:0]

Signalwidth: 16 bits
Direction: Slave to Arbiter
Phase: -
Function: Slave indicates Arbiter after RETRY or SPLIT of one or several masters which can be granted access again. HSPLIT[x] corresponding with HMASTER[3:0] is set high. Once Master reengages transaction, HSPLIT[x] should be reset immediately by the slave.

HTRANS[1:0]

Signalwidth: 2 bits
Direction: Master to Slave
Phase: Address
Function: Defines Burst-Sequence state of a transfer as listed in Table 3.4.

HTRANS[1:0]	Type	Description
b'00'	IDLE	No transfer
b'01'	BUSY	Transfer pause
b'10'	NONSEQ	New Sequence
b'11'	SEQ	Continuation of Sequence

Table 3.4: HTRANS signals

HWDATA[31:0]

Signalwidth: 32 bits
Direction: Master to Slave
Phase: Data
Function: Sends write data from Master to Slave. HWDATA, like HRDATA is segmented into 4 lanes with 8 bit width. Each of these lanes represent a single increment of an address as shown in Table 3.5. Which means AHB is able to be used by IPs that have different data width than 32 bit e.g. UART has 8 bit.

HSIZE	HADDR[7:0]	DATA[31:24]	DATA[23:16]	DATA[15:8]	DATA[7:0]
BYTE	0x00				x
BYTE	0x01			x	
BYTE	0x02		x		
BYTE	0x03	x			
HALFWORD	0x00			x	x
HALFWORD	0x02	x	x		
WORD	0x00	x	x	x	x

Table 3.5: HWDATA or HRDATA lane selection depending on HSIZE and HADDR

HWRITE

Signalwidth: 1 bit
Direction: Master to Slave
Phase: Address
Function: Master sends a high signal if transfer is for writing or a low if it is for reading data. Depending on it HWDATA or HRDATA is used.

3.2.2 AMBA APB Signals

The Advanced Peripheral Bus (APB) is part of the AMBA Bus hierarchy. It is used to connect low performance I/O Units to the high-performance system bus. The design emphasis is on power efficiency and reduced interface complexity for low-bandwidth slaves. Slaves are attached to an AHB-2-APB Bridge as depicted in Figure 2.6, which handles AHB handshakes. Signals like HADDR and HWRITE are latched through the bridge to the slaves. As shown in Table 3.6, the amount of signals involved in APB is far less than in AHB. More details about APB can be found in the specification [ARM99].

Name	Source	Description
PCLK	-	APB Clock
PRESET	-	APB low-active Reset
PADDR[31:0]	Bridge	APB address bus
PSELx	Bridge	For selecting Slave x
PENABLE	Bridge	Strobe to signal data phase of APB access
PWRITE	Bridge	Indicates Data direction. High is write, low is read
PRDATA[31:0]	Slave	Reading Data. Active when PWRITE is low
PWDATA[31:0]	Bridge	Writing Data. Active when PWRITE is high

Table 3.6: APB signals

3.3 Gaisler GRLIB

The most important question at beginning of a SoC-Design is, which processing unit to choose. To be able to design a prototype without a foundry, it is necessary that the processing unit is available as a softcore. This makes it possible to test a design in a FPGA. One popular choice for a softcore is the LEON family provided by Cobham Gaisler. The LEON processor is based on a SPARC architecture and is currently in its 4th iteration with the LEON4. Gaisler made its LEON3 processor available in their GRLIB library, which is under the LGPL open source license and gained quite a popularity among hobbyists and in academia. Beside the LEON3 processor, the library provides an AHB Controller IP, which is necessary to interconnect the processor with other IPs provided in the GRLIB. Gailser implemented their AHB bus interface extending beyond the AMBA AHB specifiction to provide also Plug&Play, scan test and interrupt support for attached IPs. Listing 3.1 and Listing 3.2 show the AHB Slave interface definition, which has the extra signals added at the bottom to support the extra functionalities.

```
-- AHB slave inputs
  type ahb_slv_in_type is record
    hsel        : std_logic_vector(0 to NAHBSLV-1);
    haddr       : std_logic_vector(31 downto 0);
    hwrite      : std_ulogic;
    htrans      : std_logic_vector(1 downto 0);
    hsize       : std_logic_vector(2 downto 0);
    hburst      : std_logic_vector(2 downto 0);
    hwdata      : std_logic_vector(AHBDW-1 downto 0);
    hprot       : std_logic_vector(3 downto 0);
    hready      : std_ulogic;
    hmaster     : std_logic_vector(3 downto 0);
    hmastlock   : std_ulogic;
    ----------------------------------------------------
    hmbsel      : std_logic_vector(0 to NAHBAMR-1);
    hirq        : std_logic_vector(NAHBIRQ-1 downto 0);
    testen      : std_ulogic;
    testrst     : std_ulogic;
    scanen      : std_ulogic;
    testoen     : std_ulogic;
    testin      : std_logic_vector(NTESTINBITS-1 downto 0);
  end record;
```

Listing 3.1: Gaisler AHB Slave input interface definition

```
-- AHB slave outputs
  type ahb_slv_out_type is record
    hready      : std_ulogic;
    hresp       : std_logic_vector(1 downto 0);
    hrdata      : std_logic_vector(AHBDW-1 downto 0);
    hsplit      : std_logic_vector(NAHBMST-1 downto 0);
    ----------------------------------------------------
    hirq        : std_logic_vector(NAHBIRQ-1 downto 0);
    hconfig     : ahb_config_type;
    hindex      : integer range 0 to NAHBSLV-1;
  end record;
```

Listing 3.2: Gaisler AHB Slave output interface definition

In the definition of `ahb_slv_in_type` there are 5 signals for scan test. The signal `hirq` is for encapsulating the interrupt request signals within the bus. `hmbsel` is for selecting a memory bank within the slave and consists of 4 signals, one for each memory bank. The Gaisler AHB Plug&Play functionality provides support for four banks per slave. This memory bank information is passed by the slave to the AHBCTRL[Cob16] IP (the bus controller) via the `hconfig` registers defined in the `ahb_slave_out_type` interface. The arrangement of the register is depicted in Figure 3.7 and shows eight 32 bit registers. The first is used for identifying the slave and the last four for its address range. There are three types of memory banks supported

- APB I/O Space
- AHB Memory Space
- AHB I/O Space

Information in the last four registers is interpreted differently for AHB and APB Space. The difference between AHB Memory and I/O Space is the support of cacheability and prefetchability for memory space and is indicated by their respective bits. In Figure 3.7, it is shown that the Memory Bank registers have each a 12-bit field for ADDR and MASK. These fields define the address space of the individual memory banks of a slave. The difference between APB and AHB is, that these 12 bits correspond to the first 12 bits of HADDR[31:20] for AHB, while for APB they correspond with HADDR[19:8] as APB Slaves need an AHB-2-APB Bridge before them to handle the AHB handshakes. So the first 12 bits of the APB Slave address selects the bridge, while the APB Slave itself will be selected by the second 12 bits.

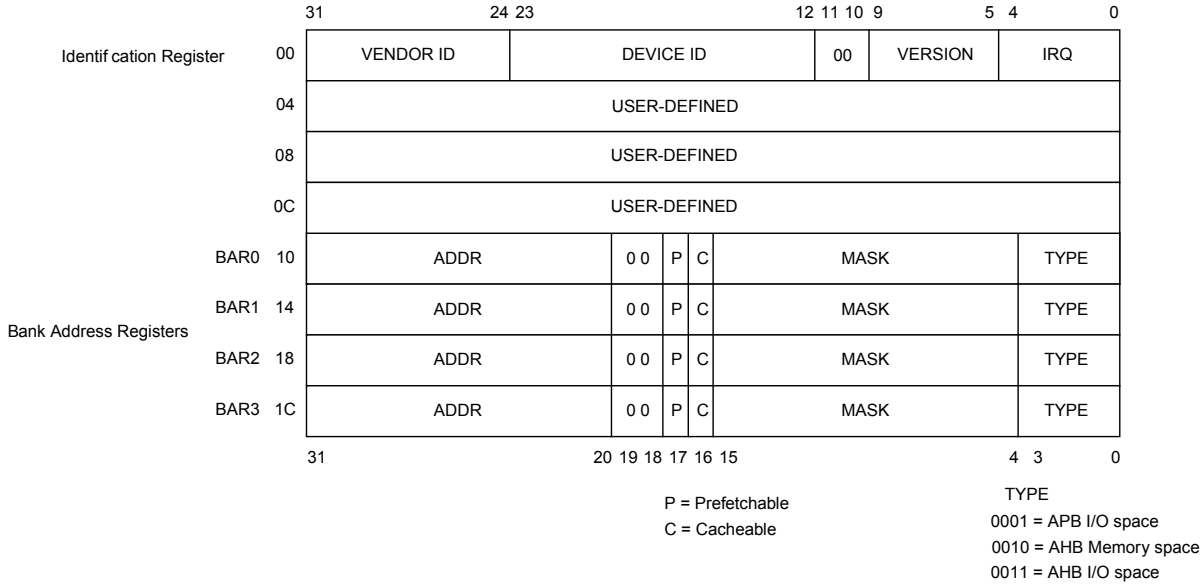

Figure 3.7: Gaisler AHB Plug&Play information register. Source: [Cob16]

Gaisler GRMON debugger & Sparc-elf compiler

Gaisler provides a compiler and a debugger for their LEON Processors. These tools make it possible to write some programs and load them into the softcore and run them. The GRMON debugger comes in handy for monitoring the AHB-Bus.

3.4 Virtualization

Virtualization is a big topic in server applications nowadays, but was not considered for embedded applications for a long time. Recently the interest of virtualized embedded system is growing

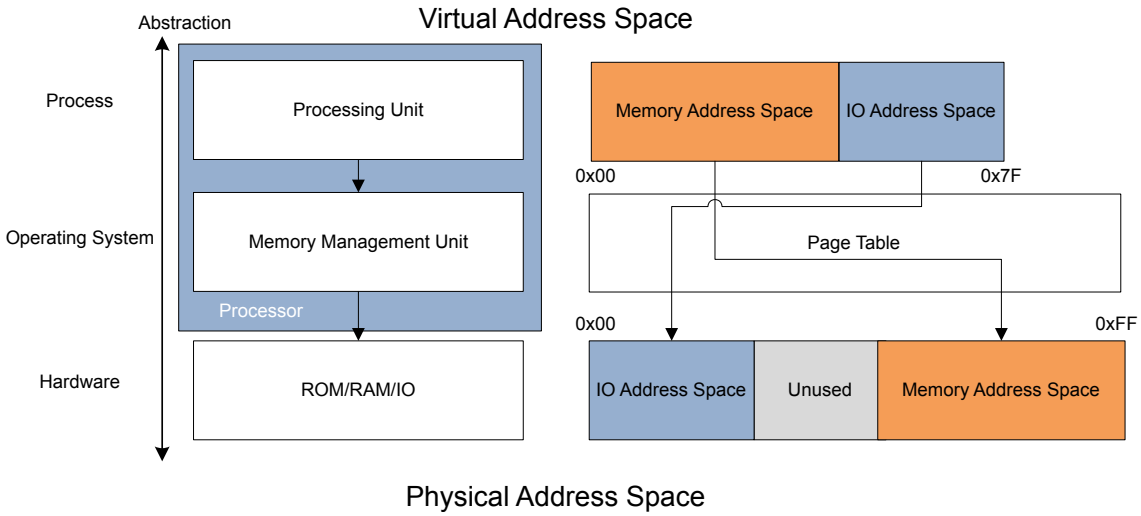

Figure 3.8: Virtual to Physical Memory translation

because it enables advanced security and safety features [AH10]. Virtualization is an important topic regarding a Cyberphysical SoC. As mentioned before, adjusting frequency and voltage of a processor-cluster may require the relocation of applications to another processor-cluster in a homogeneous multi-processor system. Virtualization was developed in the early computer days to increase the efficiency of mainframes by enabling parallel execution of applications to circumvent underutilization. This paradigm made it possible for users to share unused resources efficiently with each other. While processes were waiting for data from e.g. a tape, usually with access times in the range of seconds, the idle processor was given to another process, which could do other computations. This procedure developed to temporal separation which granted each process resources for a portion of the time. After this period exceeded a timer interrupt occurred and the control was yielded to a control-process called kernel. The kernel has the duty of scheduling and allocating resources for processes.

While a process is in control of a processor, it has full access to it, which posed a security risk. It was able to access the entire memory space and manipulate data from other processes. A solution to that problem, is to isolate each process with limited privileges in a so called User Mode with restricted access to memory and other resources, while the kernel is running in a Privileged Mode, which can grant controlled access to required resources or memory to processes. This spatial isolation is enforced by a hardware unit called the Memory Management Unit (MMU). It enables memory protection, but also a dynamic allocation of memory to a process. Any time a process tries to access data in the memory, the MMU checks in a table if the access is permitted and translates the access to the location where the data is stored as depitced in Figure 3.8. The process, therefore, is operating in a so-called virtual memory space, while every access to the physical memory space is translated by the MMU. This technique makes it possible for the kernel to accommodate data and opcodes of a process where there is space in memory.

There are two approaches to enable dynamic memory allocation. One is Segmenting and the other one is Paging. Segmenting allocates memory for a process as a contiguous section. Because processes have a different amount of opcodes and data, this leads to external fragmentation. This fragmentation is caused by unallocated memory, which doesn't have enough space to accommodate an entire process. This lost memory can be recovered by rearranging the memory by the

kernel, but that would pose lost processing time. In Paging, the memory is divided into equal sized pages. If a process requires memory, then it is only granted in amounts of a minimum of a page size, even if it doesn't need as much. This leads to internal fragmentation, which can't be recovered by the operating system. The benefit of paging, is the easy handling of their tables and allocation in memory. For more details [Sta09] is recommended.

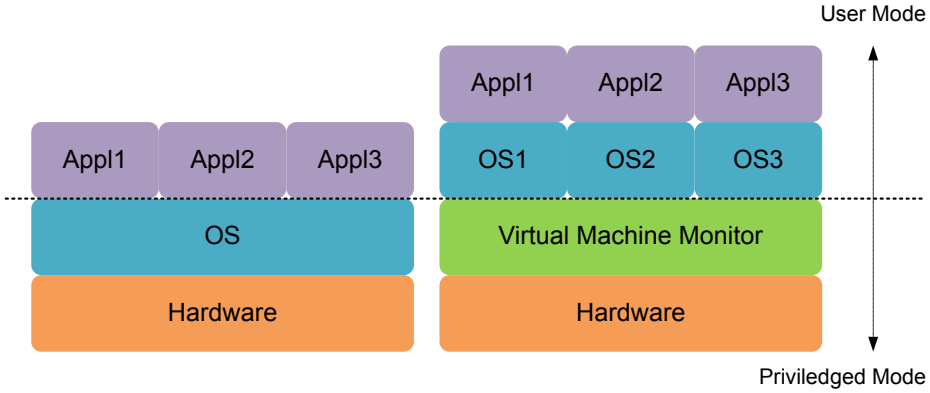

Figure 3.9: Secure computing by virtualized environments.

Modern servers use the same principle to accommodate whole operating systems of different users at the same time. It is also used as a security feature by isolating defective machines, for example due to memory leaks. Because this faulty machine is only granted a limited amount of resources, it won't affect other machines, which host other applications. If applications shared the same machine, the memory leak of one application would affect the other too. [SN05] gives a good overview of this topic and lists applications for system virtualization like

- Multiple Secure Environments: protects multiple systems from each other
- Managed Application Environments: provides customized environment for a specific application.
- Mixed OS environment
- System encapsulation: enable to suspend complete machines and store or move them
- Operating system instrumentation: monitoring entire operating systems

and much more. Especially the possibility for secure environments and mixed os environments are interesting in SoC applications and are actively explored in mixed critical systems for Integrated Modular Avionics (IMA). [MIM+13][MPHH15][MPH15] present an approach to reduce I/O Virtualization overhead utilizing commercial off-the-shelf components (COTS) like PCIe controller using an I/O Memory Management Unit for isolating DMA (Direct Memory Access) transfers from Ethernet or USB to a respective virtual machine. A hardware-based I/O virtualization approach is preferable, because an application can almost utilize 100% of the performance compared to software-based methods due to overheads, which cost in the order of 20% of the performance according to [MIM+13]. Usually, virtualized environments don't have direct access to hardware. Even the processor can only be obtained in User Mode instead of Privileged Mode. An entity

called the Virtual Machine Manager (VMM) or Hypervisor controls the hardware (Memory, I/O) as shown in Figure 3.9 and emulates the hardware via software to the virtualized machines. This emulation causes overhead and prevents 100% resource utilization. Hardware-based approaches grant protected access to memory by using Memory Management Unit in the processor (MMU) and shadow page tables [SN05] handled by the hypervisor instead of normal page tables managed by the operating system. An I/O Memory Management Unit (IOMMU) in Peripheral Controller like PCIe Root Complex would translate all DMA access of I/Os to the dedicated memory space of a virtual machine utilizing the shadow page table. This hardware support for virtualization is becoming more crucial to maintain system performance, while providing features as listed above.

4 Virtualization Solution

In previous chapters, the term Virtualization was discussed. In this chapter, there will be a step by step approach to enabling virtualization of a System-on-Chip. The first section will handle the sharing of information between SoC-Clusters. The second section discusses the necessity of restricting access to resources over Network-on-Chip to prevent memory inconsistency. The third section in this chapter will summarize the previous ones and introduces a feature, which enables the configuration of the proposed design. Subsequently a protocol will be introduced to support all the presented features.

4.1 Network-on-Chip Abstraction

Network-on-Chip has many benefits as previously mentioned for connecting distributed subsystems on a System-on-Chip together. In a SoC, subsystems fulfill various tasks for the whole system. To coordinate the effort, message passing between these systems is necessary. Since Network-on-Chip only covers transport, network, link and physical layer, the network interface of a NoC is not able to put the transported data into context. This situation means a NoC attached to a traditional system bus, which connects modules within a subsystem, doesn't act as an extension of that bus, but as a passive endpoint, also called slave. A master, usually a processing unit, feeds data to and fetches data from the network interface (NI) of the NoC. This master can give the data context and use the data accordingly. If there are multiple tasks handled in a subsystem, then there are also different types of data. It is, therefore, necessary to identify the data which is communicated between subsystems. Therefore it is required to implement a protocol on the application layer. To exemplify this, the Hypertext Transport Protocol (HTTP) is a good comparison. HTTP is an application layer protocol to exchange information between a browser on a computer (Client) and a Server which stores application specific data, in this case, a website. HTTP is not the only application protocol using the layers underneath it to communicate over a network. Other examples are the Simple Mail Transfer Protocol (SMTP) for mailing services or the File Transfer Protocol (FTP) for file exchange and much more. Out of this comparison, the first step before specifying an application protocol will be to identify the application itself.

A traditional embedded system consists of at least one processing unit for computation, some memory to store opcodes and data and some means to communicate to the outside world for receiving data to process and communicate the result back to the outside world again. These parts of an embedded system also require a way to communicate with each other. For that

purpose, a system bus was needed and many different buses were developed by various companies as mentioned before. The focus of this work will be specific on the AMBA AHB bus from ARM because it is used by Gaisler GRLIB to connect all of their IPs (Intellectual Property). Since the spatial distance between those parts, now on refered to as subsystems, are increasing it is not viable to connect them with a contiguous system bus. However, there is still the need to communicate with those subsystems. As mentioned before the Network-on-Chip enables the communication between subsystems, but their interfaces are slaves and needs to be controlled by software-based hardware driver or some form of hardware-based system which can put context to the data communicated. [GWHB11] implements a virtualization methodology utilizing a message passing library to control computation subsystems over a NoC. This message passing library is used by applications during runtime to access the NoC with kernel-modules of the operating system. This way of accessing the NoC requires a processor on both sides of communication to run an operating system or at least some kind of program to handle NoC transactions and interpret it. If one end does not have a processor to run a software-based solution, it would then require a hardware-based implementation which can interpret the incoming data and perform required actions on the respective side. Putting this in perspective of a LEON3 processor, sending data to various IPs like memory and I/O controller over a NoC, a hardware controller, further called transaction controller, has to fetch the incoming data from the NoC network interface and perform the requested transaction to the addressed IP. For efficiency reason, this transaction controller is better situated between the NoC network interface and the IPs instead of sharing a system bus with all of them. The network interface, which is at the time of this work being planned, will be an AHB slave. Therefore 2 AHB master interfaces are required for the transaction controller. One towards the network interface and one towards the various IPs. This setup would enable to interact with an IP, while fetching incoming data from the network interface. The aggregation of transaction controller and NoC network interface can be seen as a Gateway, which translates and manages access to the network, visualized by Figure 4.1.

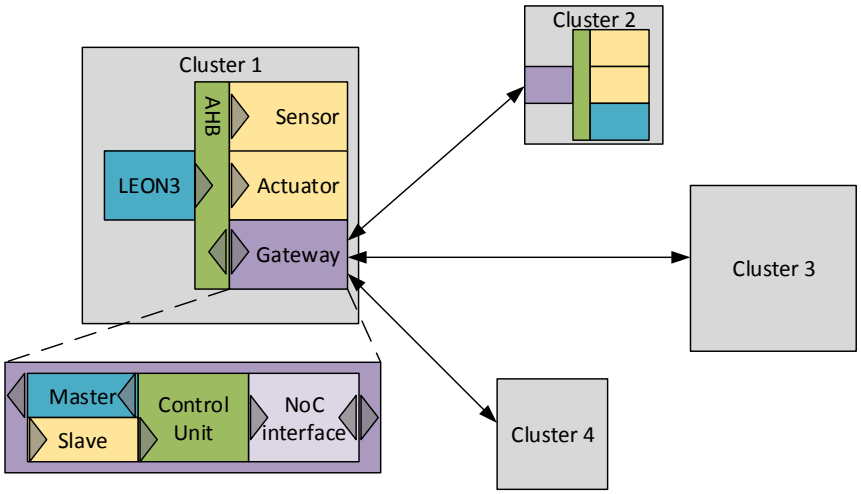

Figure 4.1: Gateway Design - Outgoing arrow for active unit, Incoming arrow for passive unit.

Until now IPs were considered as a passive units, but some IPs can be active themselves. Examples are I/O controllers like Ethernet or USB. These I/O controllers provide Direct Memory Access (DMA) for fetching and storing data, independently from a processor, directly from or to the

memory. In case that the memory controller is not located on the same system bus as those I/O controllers, a DMA access would require to be routed over NoC. The Gateway would have to accept those DMA transactions and forward them to the cluster which locates the memory controller. Therefore an AHB slave interface is necessary. Arteris, an upcoming player in the NoC IP market, has also developed a NoC standard which provides sockets for connecting clusters with bus standards of different vendors [Mar05]. This gives SoC architects a wider range of choices for selecting IPs. This approach, to tunnel bus transaction through a NoC with a Gateway, would facilitate the means to design a contiguous SoC architecture that can be natively accessed from anywhere instead of separated islands, which would require specific access routines. The local address spaces of an individual cluster can be partitioned to access certain clusters in the network. For example, a portion of the local address space of a cluster can be used to access local sensors and actuators, which are necessary for a Cyberphysical System-on-Chip (CPSoC), and other partitions of the local address space can be used to access IPs directly in remote clusters of the network. This capability to natively access any resource from anywhere is a benefit which comes with a cost. The unrestricted access to independent applications to any resource on the SoC, might lead to incoherent data. This matter leads to the next topic and section.

4.2 Application Isolation

As the Gateway enables native access to resources over Network-on-Chip, it also gives an opportunity for uncoordinated access of resources. In a traditional system an operating system manages the resources and the access to it, but in a SoC, this is not necessarily given. As mentioned before SoCs can be minimalistic designed and do not have to have the resources required for hosting an operating system. Therefore some resource management and memory protection process are needed to share resources safely without crashing the system. The discussion about safe computing is also a discussion about secure computing because software misbehavior can be unintentional, a bug, as it can be intentional, for attack purposes on a system. [GPP+14] uses a firewall approach for restricting access over NoC based on segmentation-based deny rules. [KGC12] implements a paging-based memory management approach targeting virtualization for embedded system. Memory-Management-Units as mentioned in chapter 3.4 are mainly used for translation of virtual address to physical but can also be used for memory protection. Both solutions implement a table into the NoC interface to check access permission to a given destination. This same procedure can be applied once the Gateway accepts a transaction from the AHB bus and translates them into a NoC confirm packet. This packet can be inspected by the accessed address and granted access to NoC when it confirms with a page table entry. Entries are based on addresses and can be used to distinguish NoC destination addresses. Also, the AHB address can be replaced to provide memory management capability and therefore virtualization capacity.

Because there are different ways to implement application isolation, it is necessary to analyze the requirements. Since the field of research regarding CPSoC is very young, one main requirement would be a maximum of flexibility to be able to explore as many options as possible in the future. A CPSoC is a Multiprocessor System-on-Chip with self-awareness and self-adaption. How these multiple cores are configured and what kind of processors are being used, is yet to be defined. The GRLIB and its LEON3 is a starting point for exploring implementation options. One option would be the flexibility to operate each processing cluster independently each with its own independent statically assigned application or all processing clusters are grouped in a single logical compound where a pool of applications is dynamically assigned to each processor.

To reduce cost and time in designing a CPSoC, it is recommendable to design a single processing cluster and reuse it. This means, the local address space would be the same on each cluster and applications can be programmed independently of the processing cluster, on which it will run later. This hardware/software decoupling is usually provided by virtualization mechanisms. The programmer can assume every resource can be accessed via the same addresses. Access going out of a cluster needs to be checked by the Gateway and depending on I/O access or memory access there are different requirements. Because I/Os can only be used by one application at a time a temporal isolation is enough. So a rule-based system to permit or deny access is sufficient. Memory access needs spatial isolation because data of different applications are stored at the same time. A rule-based system can also be implemented here, but this would violate hardware software decoupling. The programmer would need to know which part of the memory he can access. Virtualization by translating address for memory access would decouple hardware and software again. An application can always access the same address to store or retrieve data, while the address will be translated in the background. This procedure is provided by an operating system with the help of a MMU in the processor, but if this operating system itself needs to be virtualized, it is necessary to implement the second level of virtualization as depicted in the scheme in Figure 4.2. One benefit of such a virtualization scheme would be to restrict DMA transfers from I/Os. If a Gateway on the I/O side shares the same page table as a Gateway on the application side, then this application can be given direct access to this DMA-capable I/Os. USB host controller, for example, use a table provided by the host controller driver to access data in the memory. So in the case of a shared page table the USB host controller can access the same physical address as the application it is assigned to.

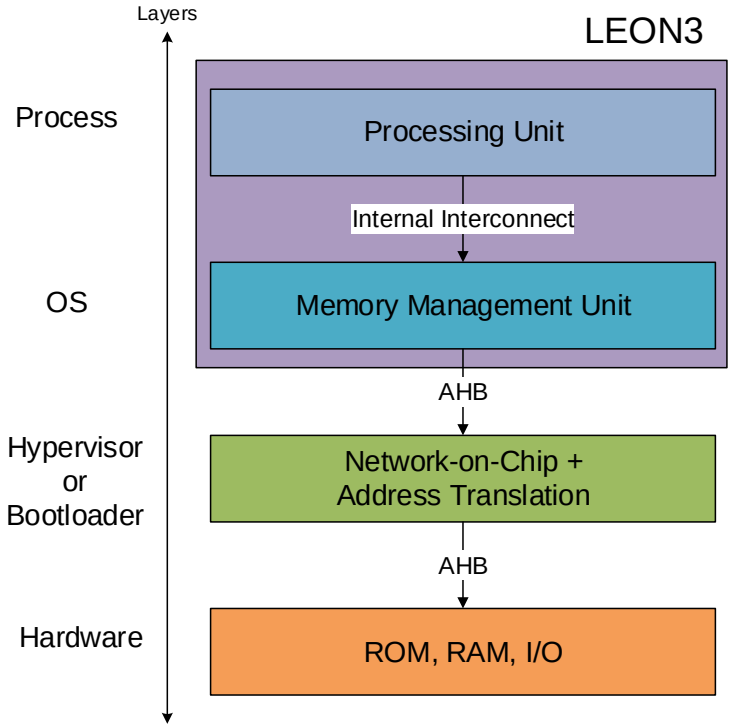

Figure 4.2: Network-on-Chip as 2nd Level Virtualization.

Virtualization can be dynamically managed during runtime by a hypervisor (Virtual Machine Monitor), but can be also statically setup by the bootloader before starting any applications. At startup of the SoC, an elected processor can run the bootloader which initializes all the global hardware and sets up filesystems or applications in the memory. After that, the Gateway of each cluster can be configured to access only to their respective filesystem or application. Extensive virtualization capabilities like dealing with table walks, a miss in the local translation-look-aside-buffer (TLB), are complicated to implement and also require an extra amount of area in every network interface. Because the main purpose would be memory protection without the need for dynamic memory allocation for the beginning, a compromise would be preferable. The first option to choose, is to implement a segment table or a page table. Segment tables allow flexibility and efficiency. Segments can be placed in the address space wherever and only as many as they are required. The range of the segments can be set on demand while page tables can only be set according to their page size and the corresponding border address. For example, pages have a size of 4kB then page locations can only be chosen by the upper 20 bits of a 32-bit address space. Page sizes have to be set once and can't be changed afterwards. If a page table is configured to 4kB page size, then a whole page will be wasted for an I/O which only requires maybe ten addresses for its hardware control. At the same time to support an operating system, requiring 16 MB of RAM for uCLinux[1], a minimum of 4000-page entries of 4kB pages is required. So a bigger page size would make management easier. For implementation, a table with 16 pages is intended. This would require a minimum page size of 2MB to support an embedded Linux OS with RAM and I/Os. This choice would mean, I/Os needs to be distanced at least 2MB from each other in the physical memory space. If the distance is lower for two I/Os, then those can only be assigned to an application as a group. In segment table, I/Os could be placed more flexible because of adjustable segment length. Extra effort has to be made to prevent overlappings and the data-structure to support the table, which would be more complicated. It would require virtual and physical segment starting address, as well as the length of the segment. A page table only needs one-page size mask for its length and a physical address or invalid address for each of its table entries. The virtual address pages would be required to be adjacent to implement a minimal sized table. The physical address pages can be located as wished. For the purpose of exploration, this work will implement a 16-page table because of its reduced design effort and therefore a reduced area cost. The next section will view the matter of how to configure the page table into the Gateway.

4.3 Architecture Configuration

As mentioned before architecture configuration is an important part for virtualization. There are many ways for different application scenarios. The focus in this section will be on static page tables which are set up, in the Gateway of a cluster, after initial startup by the bootloader. The bootloader code would be fetched into the cache of the LEON3 processor and executed. Because the page table is intended to be small and compact with 16 entries, it would be viable to write the table entries directly into the Gateway via an AHB-interface for configuration only. The other option would be to write a base address of the page table into the Gateway and it will automatically fetch the table by itself. The latter is the better option because this will not require a processor when the Gateway is extended with a remote configuration option. For the I/O side, this feature would be substantial due to a lack of a processor. Instead of fetching all the

[1] highly reduced in functionality, certain systems can even be run with 4 MB or less

page entries by the bootloader and then sending them to a remote Gateway, it would only require to fetch and send a single table base address, with which the remote Gateway can fetch the data itself. This procedure would reduce traffic in the network especially during startup where there is a high probabilty of increased traffic. To support this functionality, a new packet type has to be implemented in the application layer. So far the communication in the network has been between an AHB slave sending a request to an AHB master for handling the request with an eventual response back to the slave. With the configuration scheme, a new type of packet needs to be introduced and sums up with

1. *AHB Request packet* - Send from a slave to a master
2. *AHB Response packet* - Send from master handling a request to the source slave
3. *Config Request packet* - Send from control unit to another control unit
4. *Config Response packet* - Send from a control unit handling a request to the source
5. *DMA Injection packet* - Send from a control unit to a master

These would require multiplexing packets in the Gateway to the respective endpoints which are the master, the slave and the control unit. The DMA injection packet has not been discussed so far, but it would enable basic DMA transfer capabilities between clusters for future research with little effort. DMA injection will be discussed in the following chapter because it is a nice-to-have requirement, but is not essential.

4.4 Application Protocol

The need to establish an application protocol to support inter-cluster-communications has been strongly expressed in chapter 4.1. Figure 3.4 shows the intended design of a NoC-packet which is at the moment of this work being planned. Each packet provides space for up to 5 flits for payload. One flit of this payload is going to be used as a header/identifier for the type of the payload. The end of chapter 4.3 summarized the different types of operations needed and their packet types to implement various functionalities of the intended solution. Figure 4.3 depicts the basic layout of the header flit.

The header can be split into two parts. The first 16 bits are essential header information while the last 16 bits are only relevant for AHB Request and Response packet because it contains AHB transaction control information and is irrelevant for Gateway config packets. The packet type will be identified by the first 4 bits and gives an option for 16 payload types which give some flexibility if different applications need to be transported over NoC in the future. These types, as mentioned in chapter 4.3 are

- b'0001' - DMA Injection Packet
- b'0010' - AHB Request Packet
- b'0011' - AHB Response Packet
- b'0100' - Controller Config Request Packet

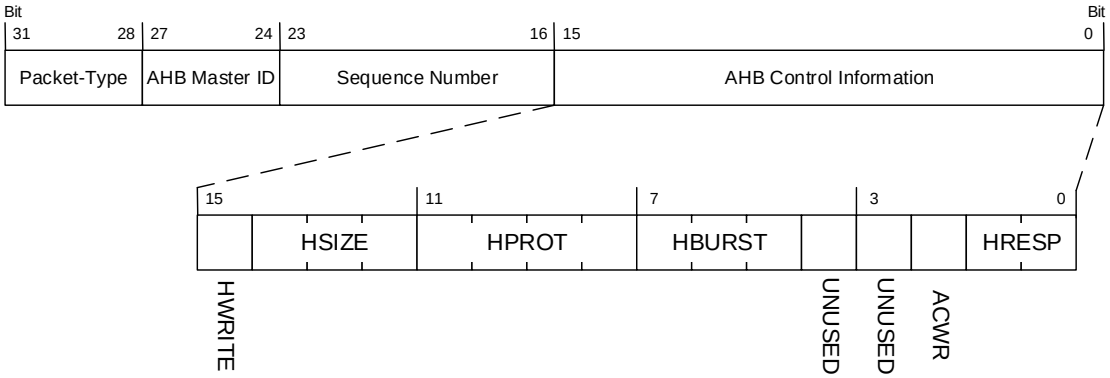

Figure 4.3: Application Protocol for AHB transcation tunneling.

- b'0101' - Controller Config Response Packet

and can be extended in the future to implement for example Message Signaled Interrupts (MSI) like supported in the PCIe standard. Following the packet type, the 4 bits in position [27 - 24] of the header are the AHB Master ID to identify the source of a request. The Master ID is the HMASTER signal of the AHB bus. This Master ID is crucial if multiple masters are located in one cluster. It would also enable the ability to implement different page tables for each master. The combination of master ID and source NoC address gives each AHB master an unique identifier in the system. For this work, the focus will only be on one master per cluster. The following 8 bits after master ID are reserved for sequence numbers, in case 5 flits per packet are not enough for future applications.

The second part of the application header is specific to individual packet types. In Figure 4.3 the second half of the header is occupied with AHB control signals. These are only required for DMA injection packets, AHB request packets and AHB response packets and are assigned as following:

[15]	HWRITE	determines read or write transcation
[14 - 12]	HSIZE	size of the data transmitted
[11 - 9]	HPROT	transaction meta data
[8 - 5]	HBURST	burst length of transaction
[2]	ACWR	active write response, not a AHB signal
[1 - 0]	HRESP	response code for transfers

This arrangement has been chosen to increase efficiency, instead of confirming with conventions. Because the header is smaller than a regular integer data type, manual packet construction by software could be bothersome and has to be done with integer manipulations. If in future new packet types are added, then they only have to adopt the first 16 bit of the header. As mentioned before, the second part is specific to packet types. HRESP are only used for response packets, but its position should not be used for other purposes in request packets or injection packets. The ACWR bit is not an AHB signal and has the purpose of signaling a master handling a request, to

send a response to a write transaction back to the source. The master is held until the response is returned. Handling burst writes in this mode could lead to extensive delays, instead a NoC packet can be utilized more efficiently and send over the network with multiple data without a response to individual transfers accepted by the slave. An error returned couldn't be mapped to the corresponding transfer, because it has already been accepted with an OKAY response. That is why ACWR will give an actual response to every single AHB transfer. Read transactions always have a response because they have to retrieve data. ACWR is activated by a status bit in the slave generating the request so it can be turned on and off on demand. The proper usage of this functionality should be explored in future research.

Packet Structure and Handling

After the header, there are four flits left to be used in a packet. Figure 4.6 til Figure 4.8 show the packet utilization of each packet type. In the first position after the header, it is usually an address for request packets. AHB Request Packets always contain the address for write or read transactions. Configuration Request Packets contain the physical Base Address of the page table in the memory.

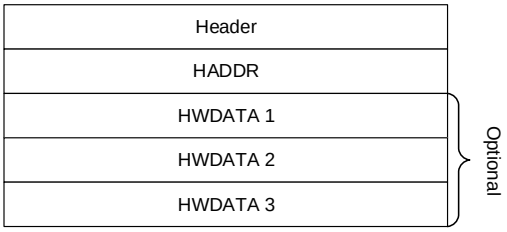

Figure 4.4: AHB Request Packet

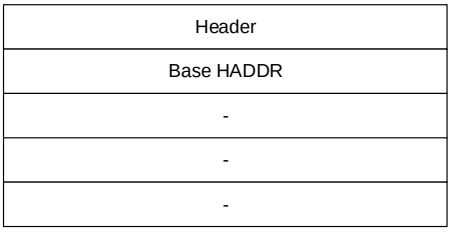

Figure 4.5: Configuration Request Packet

DMA Injection packets have addresses in the first two positions after the header because they need a source and destination address. A control unit issues a DMA Injection packet to a master. This injection packet as seen in Figure 4.6 will have a source HADDR to read data from and send the data with a write AHB Request packet, containing the destination HADDR, to a destination cluster. Because the packet is an AHB Request packet, it will be handled by the master at the destination and the data will be written to the destination HADDR.

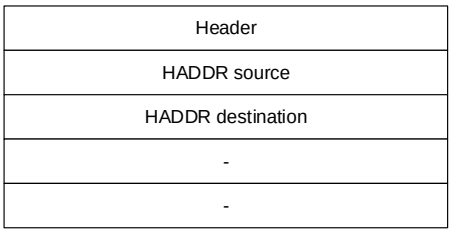

Figure 4.6: DMA Injection Packet

These AHB Request Packets have their ACWR bit turned off to be able to send three flits of

data with each packet. If the ACWR bit would be set, then the last two flits would be dropped at the destination master. The downside would be, that no response would be given to this DMA transfer. Because of that, it would be recommended to implement the control logic at the destination side. However, where and how the control logic of this DMA procedure should be implemented is not the topic of this thesis. The implemented master module will just provide the interpretation of an injection packet and respond accordingly. Beside for DMA purposes, *AHB Request Packets* are issued by Slave endpoints which accept AHB transactions. The master on the other endpoint of a communication handles this request and responds with an *AHB Response Packet*, if the request was a read request or in the case of a write request, when the ACWR bit was set. Responses to reading requests have 1 or 4 flits of data. If the read was a SINGLE burst, then there is only one. WRAP Bursts are not supported. All INCRx burst will return 4 flits of data for prefetching purpose and all data which are not needed will be dropped. If there is an ERROR while AHB request is served by a master, the HRESP field in the response packet will be set accordingly and the flit in the packet will be marked. Responses to ACWR have no additional data. A Configuration Response Packet also doesn't have any data in their packets. Chapter 5 is going into detailed view on how the packets are generated by the gateway and how they are handled internally.

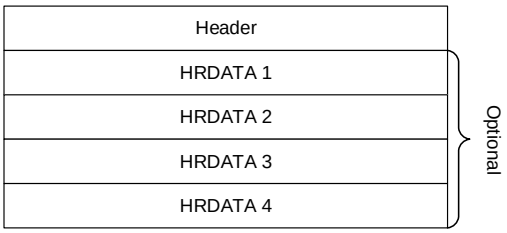

Figure 4.7: AHB Response Packet

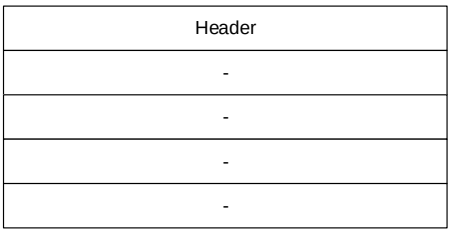

Figure 4.8: Configuration Response Packet

5 Implementation and Testing

In previous chapters the challenges and their solutions were discussed. In this chapter, the focus will be on the implementation details to give a good overview and to ease future research and developments based on the solution provided in this work. This chapter will be divided into two parts, implementation details and testing details.

5.1 Architecture and Interfaces

In chapter 4.1, basic modules were introduced, which are essential for a inter-cluster-communication interface. The introduced Gateway concept depicted in Figure 4.1 has three major components. A slave for accepting AHB transactions, a master for performing AHB transaction of incoming *AHB Request Packets* and a control unit, which multiplexes outgoing packets and demultiplexes incoming packets. Also, this control unit is essential for performing basic memory protection functionality, based on a NoC-Firewall concept discussed in chapter 4.2, as well as a configuration interface for the Gateway and the whole network discussed in 4.3. One task of the control unit, which has not been discussed, is the interfacing with the NoC network interface. The network interface is being designed with an AHB interface so it can be used as an independent slave. Figure 5.1 shows the configuration registers of the interface. Because it is not yet implemented, the control unit has to be split into two. One unit is handling control tasks and the other unit is dealing with the interfacing with the network interface. The interface unit can be developed at a later time, when the network interface is ready. This work will provide a common interface definition for all modules within the Gateway, so each module can be replaced at any time with an improved version. This modularity also provides a mean to change the architecture altogether and reuse the modules if the defined interface is being used in the new design. This common interface was a key design requirement from the start to enable modularity. Because of this modular interface, being defined in this chapters, testing can be done by joining a master and a slave together for unidirectional communication or even join two Gateways togehter after their control unit for a bi-directional bridge. Later this bridge can be split again to integrate the Network-on-Chip.

Before moving to the module interface definition, a short look at the AHB interface of the network interface is necessary, to determine some key characteristics of the NoC implementation, which should be considered in the module interface definition. With a quick look on Figure 5.1, there are 5 parts distinguishable. A control register section and four groups with six registers are distinguishable. One for transmitting and three for receiving data. The last three are receive

Base Address	+0x00	+0x01	+0x02	+0x03	+0x04	+0x05	+0x06	+0x07	+0x08	+0x09	+0x0A	+0x0B	+0x0C	+0x0D	+0x0E	+0x0F
+0x00	NISTAT0		NIADDR	NICTRL	NIINTE	NIINTF	NI0FLIT	NI1FLIT	NI2FLIT	NI3FLIT						
+0x10	TXCTRL	TXDEST	TXLEN	TXFLAG	TXDATAF0				TXDATAF1				TXDATAF2			
+0x20	TXDATAF3				TXDATAF4											
+0x30	RX0CTRL	RX0SRC	RX0LEN	RX0FLAG	RX0DATAF0				RX0DATAF1				RX0DATAF2			
+0x40	RX0DATAF3				RX0DATAF4											
+0x50	RX1CTRL	RX1SRC	RX1LEN	RX1FLAG	RX1DATAF0				RX1DATAF1				RX1DATAF2			
+0x60	RX1DATAF3				RX1DATAF4											
+0x70	RX2CTRL	RX2SRC	RX2LEN	RX2FLAG	RX2DATAF0				RX2DATAF1				RX2DATAF2			
+0x80	RX2DATAF3				RX2DATAF4											

Figure 5.1: Configuration Registers of the NoC network interface. Source: [Wan15]

buffers each with individual source filters. The source address can be set in NInFLIT as shown in Figure 5.2, with n in the range of 0 to 3. This registers can be selected by the four NIFLITm bits of RXnCTRL in Figure 5.3, to select individual source addresses for each receive buffer n from 0 to 2. This is a nice feature, considering all of the communication of the Gateway will be to the I/O cluster and the memory cluster, in the design of this work.

Figure 5.2: Rule register. SOURCE ADDR rule for filtering incoming traffic. Source: [Wan15]

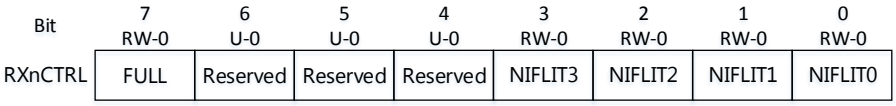

Figure 5.3: Filter select register for each receive buffer. Source: [Wan15]

As noticeable in Figure 5.2, a cluster address is defined by 4 bits. Therefore the NoC in design is capable of handling 16 endpoints. Because a receive buffer can allow multiple sources, the register RXnSRC in Figure 5.4 indicates exactly from which source the packet came. RXnLEN in Figure 5.5 shows the length of the packet, so every RXnDATAm (n=0..2, m=0..4) register has not to be checked. The same goes for the transmit buffers. Instead of the source address, the destination address is given in TXDEST and after writing the data into TXDATAm, the TXLEN is set before sending the packet.

Figure 5.4: Source Address register RXnSRC (n=0..2). Source: [Wan15]

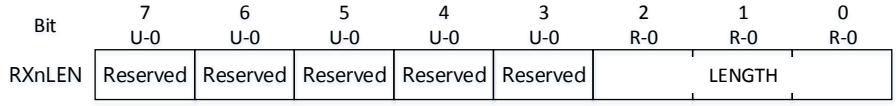

Figure 5.5: Packet length register RXnLEN (n=0..2). Source: [Wan15]

The information about the NoC address, and packet length, is essential when passing data from one module to the next and should be aggregated to a record when implemented in VHDL.

5.1.1 Module Interfaces

As mentioned before, the packets assembled by modules when dealing with AHB tranasctions, which are accessing remote resources, need to be passed between modules, but also between processes within a VHDL entity/component. For this purpose, a record was defined, as listed in Listing 5.1, to aggregate all relevant information, which are needed to be passed along with the packets.

```
type flits is array (0 to 4) of std_logic_vector(31 downto 0);

type noc_transfer_reg is record
   len : std_logic_vector(2 downto 0);
   addr : std_logic_vector(3 downto 0);
   flit : flits;
end record;
```

Listing 5.1: Record definition of NoC-relevant data.

The record contains the length of the packet, meaning the number of flits the record contains, the destination address for outgoing packets or the source address for incoming packets and the flits themselves in an array for easier indexing. Besides the data, which is passed between components, two control signals have been implemented. A "ready" signal for indicating valid data by the sender and an "acknowledged" signal for indicating acceptance by the receiver. The data handover is designed to be concluded in 3 cycles if the receiver accepts immediately. Otherwise, the data and the ready signal will be held. Figure 5.6 shows delayed and immediate acceptance of `slv_tx`. `slv_tx` and `slv_tx_ready` are cleared as soon as the acknowledged `slv_tx_ack` has reached the sender. Mind the two clock-cycle round-trip-time for any reaction to a signal. The signals `len`, `addr` and `flit` are members of `slv_tx`. The signals only indicated with numbers in brackets are members of `flit`.

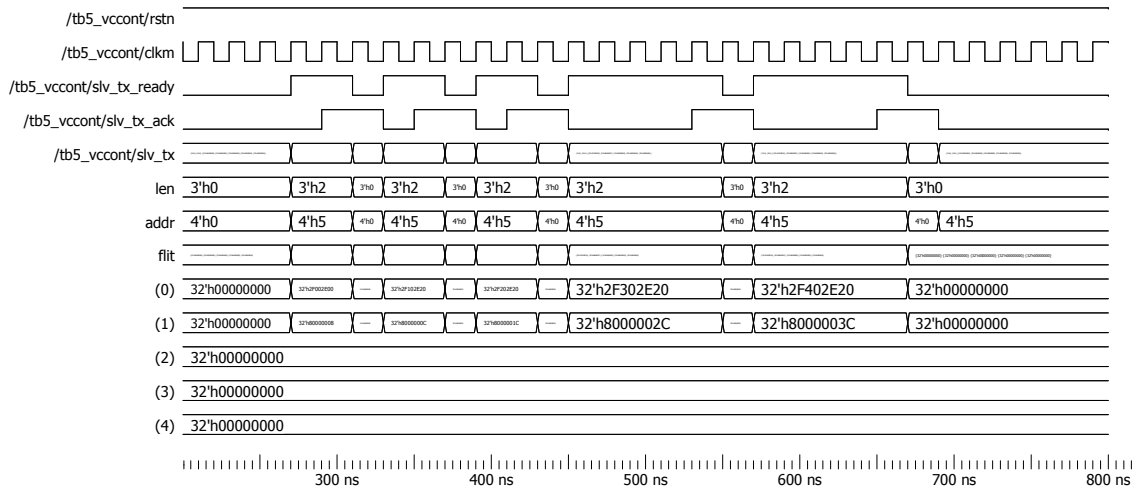

Figure 5.6: Module interface data handover with immediate and delayed acceptance.

To implement the 3 cycle handover, two variables and the "acknowledge" signal are being used. The names vary from process to process, but the procedure is always implemented in the same way. The procedure is shown schematically in Listing 5.2. `r` is a variable and `B` is a condition like `tmp = '1'` or an assignment like `tmp := '0'`. Then `not B` is the opposite assignment or condition. `ready` signal indicates valid data on the `output` and `ack` is the acceptance signal from the receiver. When comparing Listing 5.2 with Figure 5.6, then it is clear, that the condition `B` in the end prevents `r` and `output` from being cleared, just after being set for a new transmission following an old one, when `ack` is still high after the third cycle, because the cleared `ack` signal hasn't reached the sender yet.

```
-- begin of a process on positive clock edge
 if(r = '1') then
   B;
 end if;

-- some code

 if(r = '0') then
   output <= send_data();
   r := '1';
   not B;
 end if;

-- more code

 if(ack = '1' and B) then
   r := '0';
   output <= clear();
 end if;

 ready <= r;
-- end of a process
```

Listing 5.2: 3 cycle handover procedure for module interface

5.1.2 Abstraction Slave

So far the discussion about NoC abstraction was more of a conceptual one. All the modules have been explained and their role in the Gateway have been discussed. There is the control unit responsible for the virtualization tasks and controlling the flow of data between the interfacing unit, which handles the connection to the NoC network interface, and the master for handling requests or the slave generating requests. Because the slave is the generation point for most of the requests, it is the best start for the detailed explanation of the Gateway functionalities.

As mentioned before, AHB is a pipelined bus with burst capabilities. A slave has the duty to accept the incoming transfers, packetize them and to send them to the corresponding cluster. The transfer will come over the AHB interface and the basic principles of an AHB transfer have

already been discussed in chapter 3.2.1. Now it is necessary to analyze a transfer, to design a proper interface on Regitster-Transfer-Level (RTL). The interface has some conditions to pass, before it starts accepting data over AHB. First, the slave has to be selected by the respective HSEL signal. Then HREADY needs to be high, indicating that a transfer is not stalled by the slave itself. Subsequent HRESP has to be OKAY, meaning the slave is not in ERROR, RETRY of SPLIT handling mode. Once these conditions are met, the different modes of HTRANS, have to be evaluated. IDLE does not need to be considered, because when the bus is in IDLE, the slave is not selected, if the master accessing the slave is correctly designed. Meaning HADDR should be zero, and therefore no slave is selected. If HTRANS is in BUSY, then the slave is in idle and waits until the transfer continues. The standard modes are NONSEQ, starting a new transaction and SEQ, continuing a transaction. Once a transaction is started with a NONSEQ, a variable acting as a temporal packet record called `noc_tx_reg` of type `noc_transfer_reg` is initialized by setting `noc_tx_reg.len` to a value other than 0. The best starting value is two, because when a packet is generated with NONSEQ, the header and HADDR is stored in the variable seen in Figure 4.4. Another variable called `flit_index` is used to keep track of the next empty flit meaning if `noc_tx_reg.len` is 2 then `flit_index` is 3. The reason `flit_index` is used instead of `noc_tx_reg.len` is because `flit_index` is also used to select the next flit in a full response packet called `noc_rx_reg`. There `noc_rx_reg.len` indicates the absolute length of the packet compared to the current length of `noc_tx_reg.len`. `noc_tx_reg.addr` is not set by the slave, but by the control unit. The address of the destination is stored in the page table. So far everything was trivial til the first NONSEQ. From here there are three ways, how a transaction is processed. These are dependent on the HWRITE AHB signal of the transaction and the module input bit `acwr` of the slave mentioned in Listing 5.3.

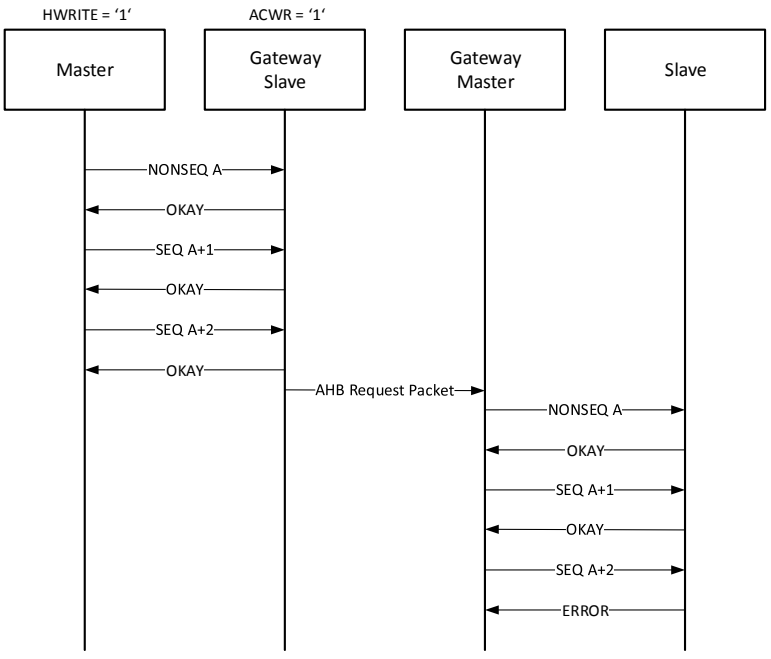

Figure 5.7: Write transaction handling without active write response (ACWR low-active).

`acwr` is only considered, when HWRITE is high, indicating a write transaction. `acwr` high means,

no response is required for a request, and HWDATA will be collected until a packet is full and then send to the destination as shown in Figure 5.7. Packets will be continuously generated until the AHB transaction is over, if there are more transfers than flits in a packet. In the previous and the following figures, an OKAY response means, that data will be accepted for writes and data will be returned with OKAY in the case of reads. HWRITE low indicates a read transaction, for which a packet is generated and immediately sent, while the HRESP is being returned, compared to a write transaction. In a write transaction, HWDATA is being transmitted by the master, while HRESP is returned by the slave. So the packet can only be sent once the data has arrived.

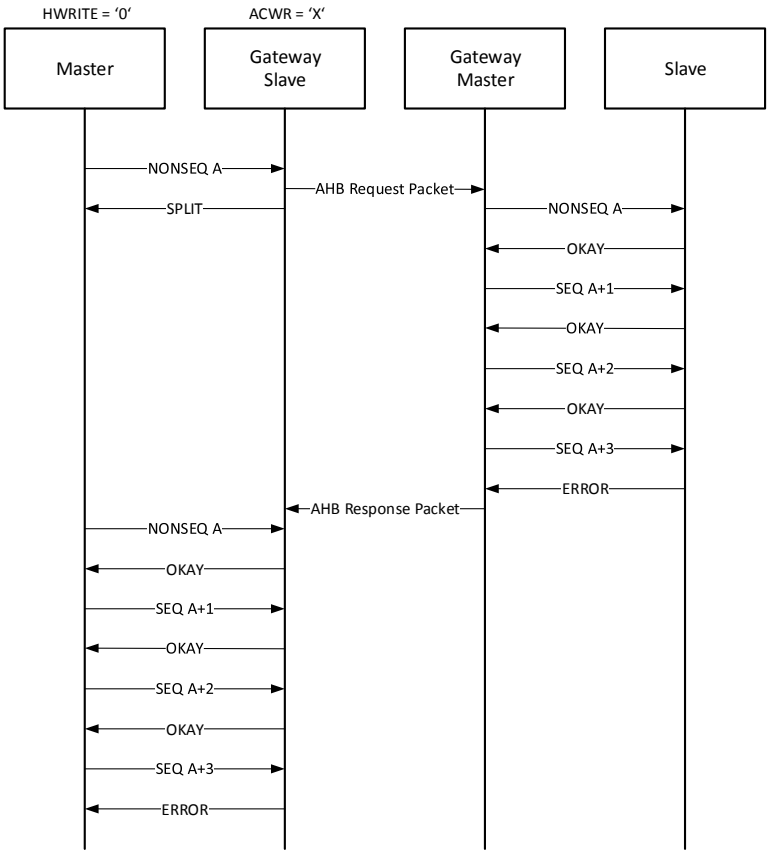

Figure 5.8: Read transaction handling of Gateways.

In Figure 5.8, the slave responds to the master with a SPLIT, which cuts the master off the bus until a response packet is returned to resolve the split. The master will retry the transaction this time with the demanded data at the disposal of the slave. In case `acwr` is low while a write transaction, responses will be expected for each transfer of data. So after every NONSEQ or SEQ, the master will be split until the response of every transfer is returned from the remote cluster. Then the master will retry the same transaction again, and the slave can give it the actual HRESP, ignoring the retransmitted data as shown in Figure 5.9.

Because there are no internal buffers in the slave, it can store only two request packets. One in its temporal record and one in the outgoing module interface. Once the packet in the module interface is accepted, the packet in the record can be moved to the interface and frees the record

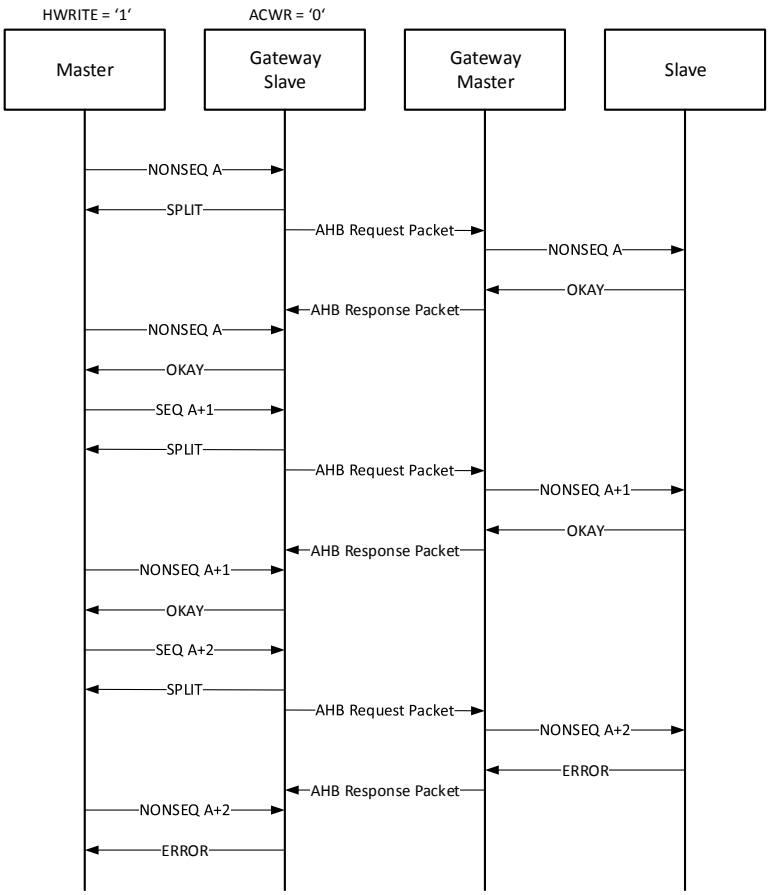

Figure 5.9: Write transaction handling with active write response (ACWR low-active).

for continuation. While the record is full, every incoming transaction will be SPLIT and queued. In the variable `split_reg`, which is a FIFO, the HMASTER signal will be stored and actively worked through, when no AHB Response Packets are pending. If a response packet is pending, the master ID in the packet header will be used to request a split continuation from the AHB arbiter by setting HSPLIT signal of the respective master. If no response packets are waiting, then the HSPLIT will be set by `split_reg`. If the master does not continue immediately and in the meantime, a response packet arrives, the HSPLIT signal of the owner will also be set. Any master contacting the slave will be first checked, if its HMASTER signal corresponds with the first in `split_reg`. If it does then, the master will be cleared out of the FIFO. If the master is not the one in the FIFO, then in the NONSEQ condition of HTRANS the HMASTER signal will be checked against the master ID in the response packet. If this check passes, the response will be served. Otherwise, a new request packet will be generated and the response packet will be kept until its been fetched.

The implemented abstraction modules have several ports. While most of the ports have been discussed in previous sections, the `acwr` port of the slave is an input port for the response mode setting of the slave. This input decides, if the corresponding ACWR bit in the AHB Control Information of a packet header is set or not. This input is low active, meaning responses will be

demanded when low and will be driven by the control unit. By default, it will be active after a reset of the control unit. There are also some generics required for the module. `hindex` is the slave ID to determine which HSEL has to be selected and `mindex` is the ID of the abstraction master, which is necessary to prevent a loopback of requests. If a loopback happens, the slave will respond with an ERROR. `mindex` is by default set to 16 and therefore turned off, unless an ID under 16 is given. The rest of the generics are AHB Plug&Play information for the AHB slave. Currently only address `ioaddr` and its mask `iomask` are being used. `memaddr` and `memmask` are unused, but can be considered to implement two separate page tables in the control unit.

```
component vcslv is
    generic( hindex : integer := 0;
        memaddr : integer := 16#600#;
        memmask : integer := 16#fff#;
        ioaddr : integer := 16#800#;
        iomask : integer := 16#fff#;
        mindex : integer := 16);
    port ( res : in  STD_LOGIC;
           clk : in  STD_LOGIC;
        acwr : in std_logic;
        requ_ready : out std_logic;
        requ_ack : in std_logic;
        requ : out noc_transfer_reg;
        resp_ready : in std_logic;
        resp_ack : out std_logic;
        resp : in noc_transfer_reg;
        ahbsi : in ahb_slv_in_type;
        ahbso : out ahb_slv_out_type);
end component;
```

Listing 5.3: Abstraction Slave module interface

At last, it should be mentioned, that because the Plug&Play support, multiple address spaces can be mapped to a single slave with the HCONFIG registers, shown in Figure 3.7. In combination with the custom signal `hmbsel`, of the AHB slave input interface definition in Listing 3.1, it is also possible to bypass the slave and control unit completely, to access the network interface directly.

5.1.3 Abstraction Master

Once the slave generates a request packet and forwards it over the network to a respective cluster, the request packet needs to be translated back to an AHB request. For this purpose, the Gateway also comprises a module with a master interface. This abstraction master also has a temporal packet record for storing active requests and generated responses. These containers are also called `noc_tx_reg` and `noc_rx_reg`. Unlike in the abstraction slave, both containers can be used at the same time depending on the type of request. The master would check every received packet if it is structured properly, meaning a read request should only have a length of two, and a write packet should have a minimum length of three otherwise the packets will be dropped. The master is set up internally as a state machine consisting of five states. At the end of each phase, signals will be set before transitioning into the next phase.

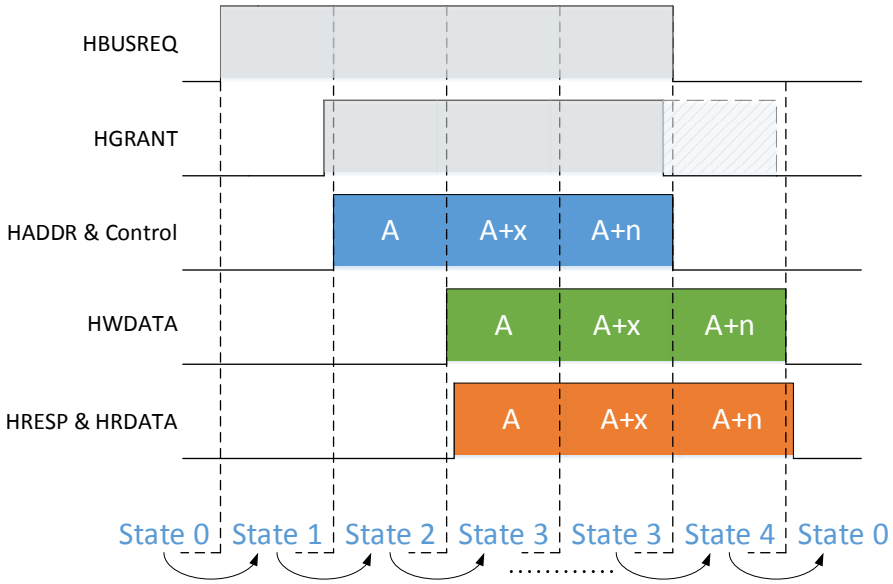

Figure 5.10: Phases of request handling in Abstraction Master.

These states are held until a condition to transit into the next state is met like HGRANT in Figure 5.10. Once the required condition for a state is fulfilled, the necessary AHB signals, like HADDR and control signals, are set before a transition occurs. Figure 5.11 visualizes the internal state machine of the master. State 0 is the Idle state. Once a new request packet has arrived, a transition occurs into state 1. During the transition, the HBUSREQx is set to indicate an AHB access request to the bus arbiter. State 1 is the Wait state and is held until HGRANTx is active. Once HGRANTx is acquired, the first address and control signals will be set, and the master goes into state 2. State 2 is the first fork in the state machine. Depending on the burst type, there is a transition to state 0, state 3 or state 4. In case HBURST is SINGLE, the transition goes into state 4 or state 0, otherwise, if HBURST is INCR, the transition goes into state 3. In latter case, at the end of state 2, also the 2nd address of the next transfer in the burst transaction is applied to the bus. Also if the transaction is a write, HWDATA of the first transfer will be sent. In state 3, there is a loop into itself until the end of the `noc_rx_reg` in the case of a write transaction or `noc_tx_reg` in the case of read transaction is reached. Just before the last index of the packet is reached, the transition into state 4 will occur. During the transition of state 3 into itself, or into state 4, HRDATA will be collected, and HRESP processed. These signals take two cycles to be received, because of the round-trip-time for each requested transfer. This round-trip-time also poses the biggest challenge in designing the abstraction master compared to the abstraction slave. While the slave reacts immediately to a request, the master only gets a response two cycles later to each transfer. State 4 finishes any transaction and if a response is required by a request, it will be sent before automatically transitioning into state 0. Responses will be given to read transactions and write transactions with active ACWR bit in the AHB Control Information fields of the protocol header. `noc_rx_reg` has to be free to accept any incoming request packets. Similar to the abstraction slave, the master can store two response packets. One in the temporal record `noc_tx_reg` and one in the outgoing module interface. One important aspect of the state machine implementation, is the HGRANTx and HREADY signal. HREADY is responsible for enabling

the state machine. When low, the state machine is held. The HGRANT signal grants bus access to a master and also signals the relevance of any AHB responses to the master. The response to the last transfer in a transaction can be without an active HGRANTx. Due to the design of the AHB bus, the responses of every transaction can be read by every single master on the bus, as every single slave on the bus can see every request on the bus. This circumstance means, that state 2 and state 4 must be processed when the HGRANTx is active high and low, while state 1 and 3 only need to be treated when HGRANTx is active high.

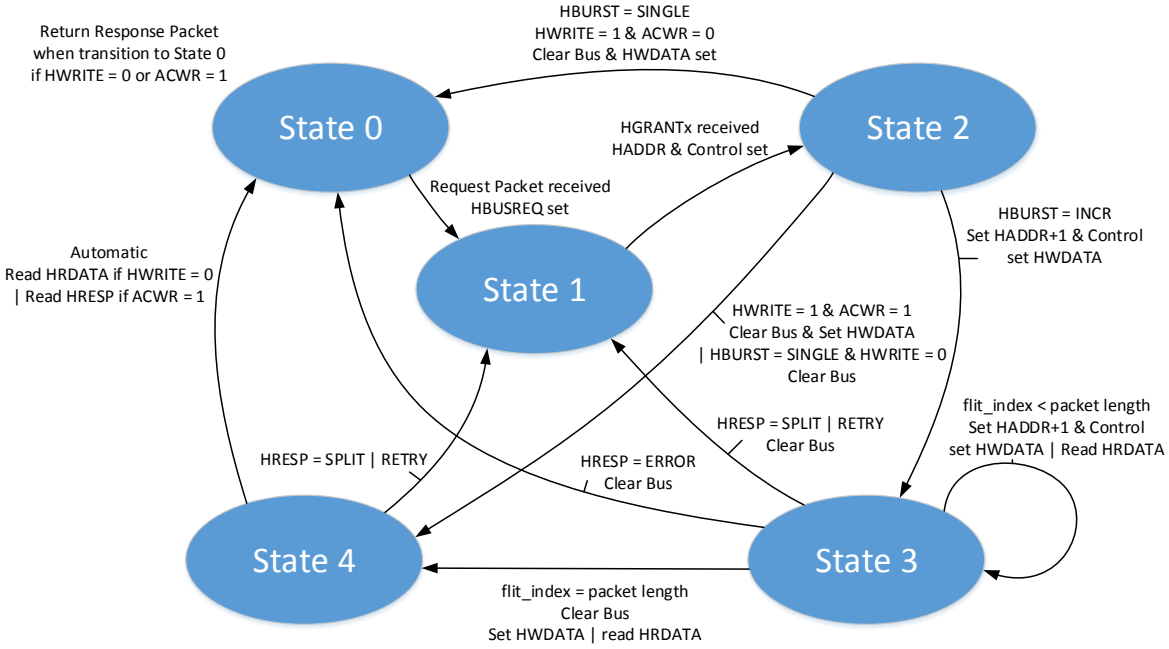

Figure 5.11: State Diagram of Abstraction Master.

5.1.4 Control Unit

So far all the modules discussed are translative ones, which process incoming transmissions, be it from the AHB or Network-on-Chip side, and translate them during the transition through the Gateway. There are no higher functions implemented and are strictly designed without any intelligence. All the intelligence and control is carried out in the control unit. These functions, as previously discussed, are a virtualization functionality for resource access over the Network-on-Chip and also local and remote configuration capabilities for the page table. All the Gateways combined form a layer for enabling the virtualization functionality in the CPSoC. The control unit is in the first place a multiplexer between the network interface and the abstraction slave/master. While multiplexing, the virtualization function is performed for the slave input. Currently, the Gateway provides 16 table entries, with three options:

1. *Rejected Access* - Access is rejected and an error response will be returned to the slave
2. *Direct Access* - Access is granted and will be forwarded with unchanged address
3. *Translated Access* - Access is granted and will be forwarded with updated address

The page table is stored in an array of a 32-bit register called `datastore`, which stores control registers for the local and remote configuration and response mode for write transactions (`acwr`). These are followed by 17 registers for the Gateway page table. The first is the mask for the address translation, followed by the 16 entries. The entries are basically all '1' for *Rejected Access* or all '0' for *Direct Access* or the destination address to which a translation should occur for *Translated Access*. The translation is limited to the first 24 bits of the address. The last 4 bits in the entry contain the destination address in the Network-on-Chip. So if any translation to the 0x00000000 address space is required, it is necessary to set one or more bits between bit 7 and bit 4 to one in the page table entry, like e.g. 0x000000F0. The mask determines the actual length of the translation. All bits, which are required to be translated, needs to be set to active high in the mask. The entry which is used among the 16, is selected by the last 4 bits of the address covered by the mask as depicted in Figure 5.12. Therefore the mask has to have at least 4 active upper bits, otherwise the translation is deactivated, and all accesses are *Direct Access*. This setup enables page sizes from 256 Bytes up to 512 Mbytes.

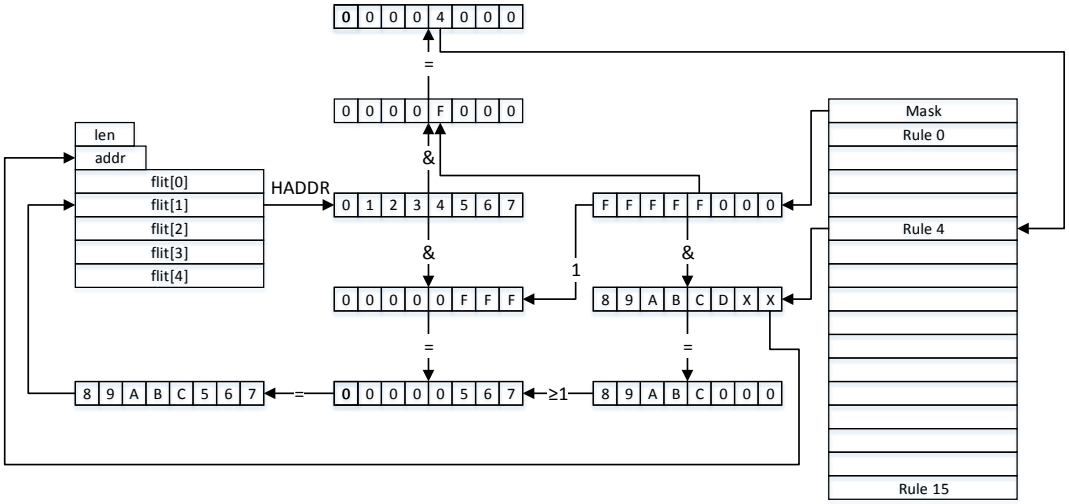

Figure 5.12: Address translation in control unit with hexadecimal numbers.

To configure the Gateway, the control unit has an own AHB interface. For this purpose, Table 5.1 lists the four configuration registers and 17 registers for the table, which are provided to be accessed over the AHB interface. With this, three options can be considered to configure the Gateway to provide maximum flexibility for future research.

- Local configuration over AHB by a local task
- Automatic configuration over NoC, once table location is provided over AHB
- Remote configuration started by providing table location over NoC

The control unit is after startup by default in *Direct Access* mode, to enable immediate access to the Network-on-Chip. Because of this, the AHB address of the slave should be mapped in the same address space as a remote Boot-ROM. If a cluster has a local Boot-ROM with a valid page table, the Gateway can be programmed directly. Otherwise, a valid location for the page table

has to be provided to start an automatic configuration by the Gateway. Once in configuration mode, the abstraction slave is cut off from the NoC until the configuration is complete. The location for the settings can also be transmitted to the Gateway remotely by a *Config Request Packet* from another cluster, which has a complete bootloader available and once the bootloader has set up the page table in an accessible memory.

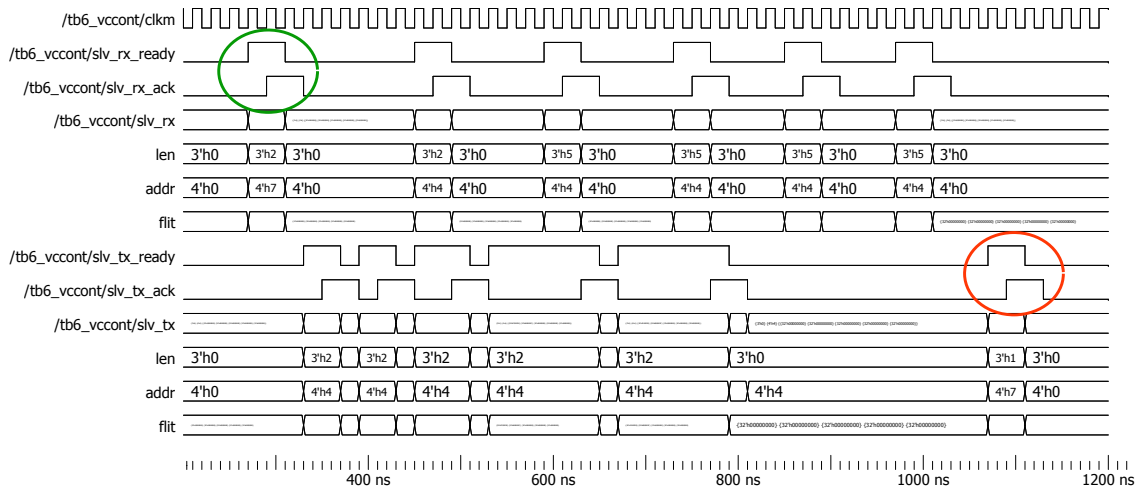

Figure 5.13: Gateway traffic for remote configuration. Green: Config Request. Red: Config Response.

Address	Register
0x00	Automatic Configuration Control
0x04	Configuration Address
0x10	Remote Configuration Control
0x14	Remote Configuration Address
0x20	Page Table Mask
0x24 - 0x60	Page Table

Table 5.1: Control Unit configuration registers

Table 5.2 lists all configuration bits in the control unit. In the Automatic Configuration registers, there are the control bits for configuration. These bits will be set automatically when a *Config Request Packet* is received, and its origin will be set in bitfield [19-16]. In the case of remote configuration, the NoC address of the table location cluster will be stored in bitfield [27-24] and the table address will be retained in register 0x04. This information is included in the *Config Request Packet*, or has to be provided over the AHB interface. Bit [23] activates the automatic configuration, and once it is finished, it will be reset and bit [22] will be set to indicate successful configuration. If the configuration was triggered remotely, then also bit [21] will be set and a *Config Response Packet* will be sent to acknowledge as marked in red in Figure 5.13. The marked packet in green is the config request. Once received by the opposite side, the receiver starts to fetch the settings by sending five *AHB Request Packets*, using the sequence field in the header to identify them. Every Gateway can send *Config Request Packets* via the registers 0x10. This ability might raise security concerns, which have to be tackled in future research about overall

Address	Bit	Activity	Description
0x00	27 - 24	-	NoC Address of table location
0x00	23	High	Start Configuration; Abstraction Slave cut off when set
0x00	22	High	Configuration Complete; Bit 23 must be low
0x00	21	High	Remote Configured
0x00	19 - 16	-	Remote configuration origin
0x00	0	Low	ACWR bit - Active Write Response bit for local Slave
0x10	31	High	Send Remote Configuration
0x10	30	High	Remote Configuration Complete; Bit 31 must be low
0x10	27 - 24	-	Remote Configuration Gateway NoC Address
0x10	23 - 20	-	NoC Address of table location for Remote Configuration

Table 5.2: Control Unit configuration bits.

system security. To send a configuration request, it is necessary to provide destination NoC address [27-24], the NoC address where the table is stored [19-16] and the table location address in register 0x14.

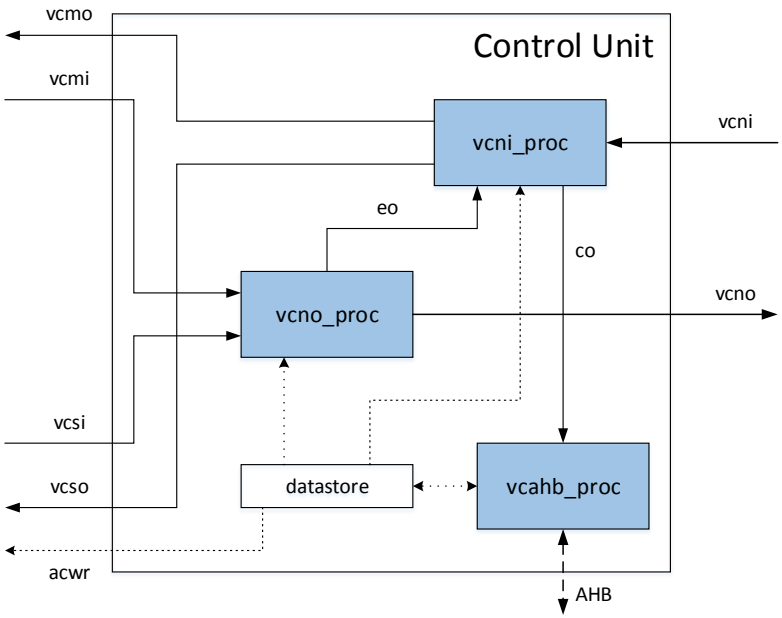

Figure 5.14: Control Unit structure. Solid Lines: Module Interfaces. Dotted Lines: Signals

All the functionalities discussed so far, are implemented into the control unit. To prevent a monolithic implementation, the control unit is structured in three processes, depicted in Figure 5.14. `vcno_proc` is responsible for all outgoing network traffic to `vcno` coming in from the slave port `vcsi` and the master port `vcmi`. It also generates the *AHB Request Packets* for fetching the configuration settings and *Config Response Packet*, once the right bits are set in the `datastore`. It is also responsible to performing the virtualization functionality and handing over an *AHB Response Packet* to `vcni_proc` with an ERROR HRESP over `eo`, to return it to the slave over `vcso`. `vcni_proc` has to demultiplex the incoming traffic over `vcni` to the master port `vcmo` in

case of an *AHB Request Packet*, or to the slave port `vcso` in case of an *AHB Response Packet*. If an *AHB Response Packet* is identified as a configuration fetch, by a header with the master ID of the abstraction master, it is sent to `vcahb_proc` over `co`, to be stored in the `datastore`. This design allows to connect two control units together to form a bridge, by connecting the `vcno` of the first unit to the `vcni` of the second unit and vice versa. All the testing procedures performed in the next section, are made in that configuration.

5.2 FPGA Implementation and Testing

In this section, the focus from the design of the Gateway is shifted towards the integration into one of the FPGA designs of the Gaisler GRLIB. GRLIB provides Makefiles for synthesis, simulation, and programming FPGA designs. Besides the Gateway model, this work also provides a Ubuntu 64-bit virtual machine setup, to support all the functionalities of the GRLIB, SPARC cross compiler, and GRMON debugger. These Makefiles and the GRLIB file-tree enables easy integration of custom packages into the library. All tests are performed under `grlib-gpl-1.5.0-b4164`. For the testing purpose, the pre-existing Digilent NEXYS4DDR design was chosen, because of cost, area and already existing experience with Digilent boards. To integrate all the VHDL files into the GRLIB, a VHDL package file called *custom* was created with component and type declarations of all discussed modules and records and some extra modules, which have not been mentioned and have been developed for testing purposes. This package and all the VHDL files have to be placed in a folder in the `lib` subfolder of the GRLIB, which is structured on the top-level like

```
grlib-gpl-1.5.0-b4164/

bin boards designs doc lib netlists software
```

The `lib` folder contains a file called `libs.txt`. It contains all the libraries in the project. If a new library is placed in `lib`, the new library folder name has to be appended to the text file. Instead of creating a new library, the package was added to the existing `gaisler` library. It has been placed in `grlib-gpl-1.5.0-b4164/lib/gaisler/custom/` and contains all developed files of this work. The `gaisler` folder contains a text file called `dirs.txt`, which lists all the local folders with packages, so the Makefile can find all of them, when creating a design project. It is also necessary to place a `vhdlsyn.txt`, with a list of all the synthesis relevant VHDL files, in the folder containing the package files. The `designs` subfolder in the top-level folder of GRLIB, contains the NEXYS4DDR design in a subfolder called `leon3-digilent-nexys4ddr`. `leon3mp.vhd` is the top module of the design and can be configured by a GUI interface called by the Make command in

```
grlib-gpl-1.5.0-b4164/designs/leon3-digilent-nexys4ddr/

make xconfig
```

The window in Figure 5.15 will pop up. Configuring the entire project from the GUI is possible. After saving, a file called `config.vhd` will be created with the constants necessary to generate the components in `leon3mp.vhd`. Most of the default settings can be taken as they are, except the AMBA configuration. In the AMBA submenu shown in Figure 5.16, AHB split-transaction support must be turned on, otherwise a SPLIT response will cause a lock of the bus. One important

advice when experimenting with the configuration is, not to disable the Ethernet to save area in the FPGA. There is a bug in the design with the Ethernet clock causing unpredictable modes on the bus. After the configuration, `leon3mp.vhd` can be edited to accommodate the Gateways. The Gateway uses a wrapper called `vcont`, containing the abstraction master, abstraction slave, and the control unit. Two Gateways have been integrated back to back into the system as a bridge with both ends attached to the each other. The reason for this, is to monitor the bus activity caused by both ends of the Gateway. AHB monitoring is possible due to a Debug Support Unit (DSU) provided in the GRLIB, which can be accessed with the Gaisler GRMON debugger once the design is built and programmed with the commands

```
grlib-gpl-1.5.0-b4164/designs/leon3-digilent-nexys4ddr/

make vivado

make vivado-prog-fpga
```

It is recommended to build the design with at least Vivado 2015.4. Earlier versions are not advisable.

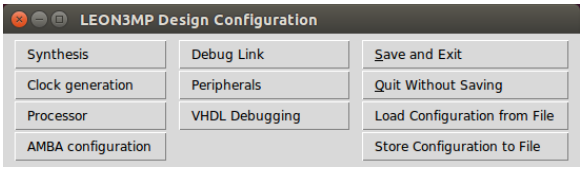

Figure 5.15: Design configuration main menu.

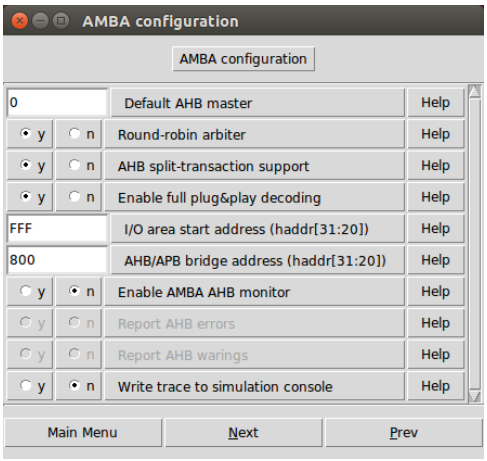

Figure 5.16: AMBA configuration menu.

After the FPGA is programmed, the SoC can be tested by loading and running test programms with GRMON. For that purpose the SoC DDR controller needs to be configured with the following commands

```
grmon -digilent -u

ddr2delay scan
```

The first command connects to the SoC over JTAG link (`-digilent`) with the UART output redirected to the active debug link (`-u`). Debugging is also possible over Ethernet and UART. After that DDR configuration, the debugger has to disconnect with `exit`. Once reconnected again, the command `info sys` should display updated the AHB Plug&Play information of all components attached to the AHB bus. The DDR controller should displays the DDR details of Listing 5.4. Then the system is ready to be run.

```
  ddr2spa0  Cobham Gaisler  Single-port DDR2 controller
            AHB: 40000000 - 48000000
            AHB: FFF00100 - FFF00200
            16-bit DDR2 : 1 * 128 MB @ 0x40000000, 8 internal banks
            140 MHz, col 10, ref 7.8 us, trfc 135 ns
```

Listing 5.4: AHB Plug&Play information of a configured DDR2 controller

Test Setup

For testing purposes, a C source file was created with the registers in use. Listing 5.5 lists all registers. Naming convention is G1 or G2 for the Gateway, PT for page table for the local Gateway configuration, RT for remote page table, BASE for an address and Exx for page table entries. The purpose of these registers can be read again in Table 5.1, where the last 2 hexadecimal digits of the address correspond.

```
volatile unsigned int G1CTRL0  = (unsigned int *) 0x70A00000;
volatile unsigned int G1PTBASE = (unsigned int *) 0x70A00004;
volatile unsigned int G1RTCTRL = (unsigned int *) 0x70A00010;
volatile unsigned int G1RTBASE = (unsigned int *) 0x70A00014;
volatile unsigned int G1PTMASK = (unsigned int *) 0x70A00020;
volatile unsigned int G1PTE01  = (unsigned int *) 0x70A00024;
volatile unsigned int G1PTE02  = (unsigned int *) 0x70A00028;

volatile unsigned int G2CTRL0  = (unsigned int *) 0xC0A00000;
volatile unsigned int G2PTBASE = (unsigned int *) 0xC0A00004;
volatile unsigned int G2RTCTRL = (unsigned int *) 0xC0A00010;
volatile unsigned int G2RTBASE = (unsigned int *) 0xC0A00014;
volatile unsigned int G2PTMASK = (unsigned int *) 0xC0A00020;
volatile unsigned int G2PTE01  = (unsigned int *) 0xC0A00024;
volatile unsigned int G2PTE02  = (unsigned int *) 0xC0A00028;

volatile unsigned int G1TEST   = (unsigned int *) 0x60001000;
volatile unsigned int G1MEMREG = (unsigned int *) 0x46001000;
volatile unsigned int G1ERROR1 = (unsigned int *) 0x60000010;
volatile unsigned int G1ERROR2 = (unsigned int *) 0x60002000;

volatile unsigned int G2TEST   = (unsigned int *) 0xB0001000;
volatile unsigned int G2MEMREG = (unsigned int *) 0x47001000;
volatile unsigned int G2ERROR  = (unsigned int *) 0xB0000010;
```

Listing 5.5: Configuration and test registers for implemented design.

All names are Precompiler code, which will be replaced with their lowercase form and the pointer operator `*`. For an example `#define G1CTRL0 *g1ctrl0`. GxTEST is an address in the address space of the abstraction slave. That address will be translated to GxMEMREG, a location in the memory. G1ERROR1 is an address, that will be rejected by the Gateway. G1ERROR2 is an

address, that will be translated, but the translated address does not belong to any component. This will cause an ERROR response from the AHB bus controller. The configuration of the Gateways will be discussed in the last section. Until then the results of basic functionalities will be explained.

Single Read & Single Write without response

From here on, the different modes and functionalities of the Gateways are going to be tested. The results of the tests will be discussed. The configuration results of the Gateways will be shown later. After loading a test into the SoC with the `load` command and executing it with `run`, the results can be displayed with `ahb 100`. The first test will be the SINGLE write without response. To turn off the response of requests, bit 0 in 0x00 register of the control unit must be turned on. This happens in line 4 of Listing 5.7 and can be seen as an action on the bus in line 2 of Listing 5.6. When writing control registers, it is important not to reset existing settings. Therefore a read of the register must be done the get the current state, which can not be seen in the result, because the buffer of the DSU was not long enough, and add or remove the respective bit. This is done with the function `clearnset()`. The first parameter is the register which needs to be altered, while the second sets all active bits to 0 and the third sets all active bits to 1.

```
1  TIME     ADDRESS   D[31:0]   TYPE   TRANS SIZE BURST MST LOCK RESP
2    35881 70A00000 00400001 read     2    2     0     0   0     0
3    35898 40001980 C4006030 read     2    2     1     0   0     0
4    ------------------------------------------------------------
5    35922 4000403C B0001000 read     3    2     1     0   0     0
6    35928 46001000 00000007 write    2    2     0     0   0     0
7    35948 400019A0 C400A048 read     2    2     1     0   0     0
8    ------------------------------------------------------------
9    35955 400019BC 81E80000 read     3    2     1     0   0     0
10   35956 60001000 0000000F write    2    2     0     0   0     0    A1
11   35967 40004040 47001000 read     2    2     1     0   0     0
12   ------------------------------------------------------------
13   35974 4000405C 40004344 read     3    2     1     0   0     0
14   35975 46001000 0000000F write    2    2     0     4   0     0    A2
15   35953 60001000 00000000 read     2    2     0     0   0     3    B1
16   35967 46000000 0000000F read     2    2     0     4   0     0    B2
17   35974 60001000 0000000F read     2    2     0     0   0     0    B3
18   35979 60002000 00000000 read     2    2     0     0   0     3    C1
19   35984 49003000 00000000 read     2    2     0     4   0     1    C2
20   35991 60002000 00000000 read     2    2     0     0   0     1    C3
```

Listing 5.6: Results with page mask 0xFFFFF000 and active write response turned off. Coresponding transactions grouped with same letter.

The dotted line in the results, are LEON3 opcode fetches, which have been removed to reduce the length of the results. In line 10 of the results, the SINGLE write transaction is performed by the program executed, which corresponds to line 6 of the source code. This request is being transmitted to the other Gateway and executed in line 14 in the results. After that, the LEON3 performs a read on the same register, which is the first SPLIT in line 15, executed right afterwards

by MST 4 and the response returned, seen in line 17. Line 18 shows another read, which is performed. But this time, an error is returned, because the translated address in line 19 does not exist. Then the ERROR is returned in line 20. All tests had to be exited by causing an ERROR for the LEON3 (MST 0), which stops the execution, because there had been no trap vectors defined to catch the exception. Otherwise, the LEON3 continues to perform more actions, and the DSU overwrites the relevant entries in the AHB monitor buffer.

```
1 main()
2 {
3   int v = 7;
4   clearnset(G1CTRL0,0x00000000,0x00000001);
5   G1MEMREG = v;
6   G1TEST = 15;
7   v = G1TEST;
8   ERROR = G1ERROR1;
9 }
```

Listing 5.7: Test programm for single read and write without response

Single Read & Single Write with response

For this test, the same source code as in Listing 5.7 is being used, but instead of setting Bit 0 in line 4 the bit has been deleted, meaning responses for each write request are required. This test is performed on G2 Gateway. The result of this change can be seen in line 7, 8 and 9 of Listing 5.8. The request is executed by MST 3. The read afterwards follows the same pattern seen before, with the only difference of the address, on which the action is performed. The last read carried out in this set returns an ERROR, because the Gateway rejected the access. The read is first SPLIT and immediately returned with an ERROR, because the request will not be forwarded to MST 3.

```
1  TIME     ADDRESS   D[31:0]   TYPE   TRANS SIZE BURST MST LOCK RESP
2    35917 C0A00000 00640000 read     2    2     0     0   0    0
3    35934 40001980 C400603C read     2    2     1     0   0    0
4    ------------------------------------------------------------
5    35964 47001000 00000007 write    2    2     0     0   0    0
6    ------------------------------------------------------------
7    35993 B0001000 0000000F write    2    2     0     0   0    3    A1
8    35998 47001000 0000000F write    2    2     0     3   0    0    A2
9    36005 B0001000 0000000F write    2    2     0     0   0    0    A3
10   36018 40004040 B0000010 read     2    2     1     0   0    0
11   ------------------------------------------------------------
12   36031 B0001000 00000000 read     2    2     0     0   0    3    B1
13   36044 47001000 0000000F read     2    2     0     3   0    0    B2
14   36051 B0001000 0000000F read     2    2     0     0   0    0    B3
15   36056 B0000010 00000000 read     2    2     0     0   0    3    C1
16   36062 B0000010 00000000 read     2    2     0     0   0    1    C2
```

Listing 5.8: Results with page mask 0xFFFFF000 and active write response turned on. Coresponding transactions grouped with same letter.

Burst Write without response

The results until now were showing SINGLE transactions, because burst transactions can not be performed by software. The only occasion, where bursts are used by the LEON3 is, when it is fetching opcodes. Another component was developed, to test the Gateway on burst transactions. It is a combination of simple AHB slave with registers and the abstraction master. This master `MST 5` generates AHB bursts, while the request are forwarded to `MST 4` in Listing 5.9. The test master `ahbtstmst` executes standard read and write bursts with eight transfers. The values to write can be set in the registers of the slave interface, but there are already values defined and only an address and the start bit have to be set. These are done by the actions in line 2 and 5 in the test results as shown in Listing 5.9. The abstraction slave accepts the eight transfers from line 8 onwards and forwards them. From the structure of the *AHB Request Packets*, there should be two packets with three flits of data and one packet with two flits of data transmitted to `MST 4`. Immediately after the burst, the LEON3 makes a read request. This request has also been accepted. This shows, that the Gateway can handle requests from multiple sources.

```
1  TIME     ADDRESS   D[31:0]   TYPE   TRANS SIZE BURST MST LOCK RESP
2    37265 50A00004 60001000 write    2     2      0     0    0     0
3    37278 40004000 D0000000 read     2     2      1     0    0     0
4    ------------------------------------------------------------------
5    37317 50A00000 80008000 write    2     2      0     0    0     0
6    37329 40004040 10800010 read     2     2      1     0    0     0
7    ------------------------------------------------------------------
8    37337 60001000 10000000 write    2     2      1     5    0     0    A1
9    37338 60001004 02000000 write    3     2      1     5    0     0    B1
10   37339 60001008 00300000 write    3     2      1     5    0     0    C1
11   37340 6000100C 00040000 write    3     2      1     5    0     0    D1
12   37341 60001010 00005000 write    3     2      1     5    0     0    E1
13   37342 60001014 00000600 write    3     2      1     5    0     0    F1
14   37343 60001018 00000070 write    3     2      1     5    0     0    G1
15   37344 6000101C 00000008 write    3     2      1     5    0     0    H1
16   37347 60001000 00000000 read     2     2      0     0    0     3
17   37351 60001000 00000000 read     2     2      0     0    0     3    I1
18   37356 46001000 10000000 write    3     2      1     4    0     0    A2
19   37357 46001004 02000000 write    3     2      1     4    0     0    B2
20   37358 46001008 00300000 write    3     2      1     4    0     0    C2
21   37366 4600100C 00040000 write    2     2      1     4    0     0    D2
22   37367 46001010 00005000 write    3     2      1     4    0     0    E2
23   37368 46001014 00000600 write    3     2      1     4    0     0    F2
24   37376 46001018 00000070 write    2     2      1     4    0     0    G2
25   37377 4600101C 00000008 write    3     2      1     4    0     0    H2
26   37405 46001000 10000000 read     2     2      0     4    0     0    I2
27   37412 60001000 10000000 read     2     2      0     0    0     0    I3
28   37417 10800010 10000000 write    2     2      0     0    0     1
```

Listing 5.9: Burst write handling without individual transfer response. Coresponding transactions grouped with same letter.

Burst Read

Burst reads are more complicated to handle than burst writes without responses. The same setup as in the previous test for burst writes is used. In line 5 of Listing 5.10, the abstraction slave accepts the request by responding with a SPLIT to `MST 5` and forwarding the request, which is executed by `MST 4` in the next four lines. Also, the LEON3 (`MST 0`) is performing a single read request, which is SPLIT in line 10. The request is also forwarded, while the first request from the burst access is returned and served. After the 4th transfer, the master continues to finish his eight transfers, but is SPLIT again at the 5th transfer. Meanwhile, the read request from LEON3 has returned in line 17. This response shows, that the abstraction slave can handle multiple SPLIT at the same time.

```
1  TIME     ADDRESS  D[31:0]  TYPE  TRANS SIZE BURST MST LOCK RESP
2    37067 50A00000 80000000 write    2    2     0    0    0    0
3    37079 40004040 10800010 read     2    2     1    0    0    0
4    ------------------------------------------------------------
5    37088 60001000 00000000 read     2    2     1    5    0    3   A1
6    37101 46001000 10000000 read     2    2     1    4    0    0   A2
7    37102 46001004 02000000 read     3    2     1    4    0    0   B1
8    37103 46001008 00300000 read     3    2     1    4    0    0   C1
9    37104 4600100C 00040000 read     3    2     1    4    0    0   D1
10   37107 60001000 00000000 read     2    2     0    0    0    3
11   37120 46001000 10000000 read     2    2     0    4    0    0
12   37121 60001000 10000000 read     2    2     1    5    0    0   A3
13   37122 60001004 02000000 read     3    2     1    5    0    0   B2
14   37123 60001008 00300000 read     3    2     1    5    0    0   C2
15   37124 6000100C 00040000 read     3    2     1    5    0    0   D2
16   37126 60001010 00000000 read     3    2     1    5    0    3   E1
17   37128 60001000 10000000 read     2    2     0    0    0    0
18   37139 46001010 00005000 read     2    2     1    4    0    0   E2
19   37140 46001014 00000600 read     3    2     1    4    0    0   F1
20   37141 46001018 00000070 read     3    2     1    4    0    0   G1
21   37142 4600101C 00000008 read     3    2     1    4    0    0   H1
22   37145 60001000 00000000 read     2    2     0    0    0    3
23   37158 46001000 10000000 read     2    2     0    4    0    0
24   37159 60001010 00005000 read     2    2     1    5    0    0   E3
25   37160 60001014 00000600 read     3    2     1    5    0    0   F2
26   37161 60001018 00000070 read     3    2     1    5    0    0   G2
27   37162 6000101C 00000008 read     3    2     1    5    0    0   H2
28   37165 60001000 10000000 read     2    2     0    0    0    0
29   37170 10800010 10000000 write    2    2     0    0    0    1
```

Listing 5.10: Burst read handling of the Gateway. Coresponding transactions grouped with same letter.

Burst Write with response

Final test setup for the abstraction slave and abstraction master is the burst write with response. After each write, the abstraction slave responds with a SPLIT and waits until `MST 4` returns an

AHB Response Packet with the AHB response in Listing 5.11. The write request is repeated again, but this time the abstraction slave returns HRESP and accepts the next transfer in the burst and SPLIT again. Mixed in between, is a read from LEON3 `MST 0` in line 19, executed by `MST 4` in line 21 and returned to the LEON3 in line 24.

```
1  TIME     ADDRESS   D[31:0]   TYPE   TRANS  SIZE  BURST  MST  LOCK  RESP
2    37371  50A00000  80008000  write    2     2      0     0    0     0
3    37382  400019C0  C2004000  read     2     2      1     0    0     0
4    ------------------------------------------------------------------
5    37389  400019DC  96102000  read     3     2      1     0    0     0
6    37391  60001000  00000000  write    2     2      1     5    0     3    A1
7    37408  40004060  10800010  read     2     2      1     0    0     0
8    ------------------------------------------------------------------
9    37415  4000407C  400044F4  read     3     2      1     0    0     0
10   37416  46001000  10000000  write    2     2      0     4    0     0    A2
11   ------------------------------------------------------------------
12   37451  60001000  10000000  write    2     2      1     5    0     0    A3
13   37453  60001004  00000000  write    3     2      1     5    0     3    B1
14   37456  60001000  00000000  read     2     2      0     0    0     3
15   37459  46001004  02000000  write    2     2      0     4    0     0    B2
16   37479  46001000  10000000  read     2     2      0     4    0     0
17   37481  60001004  02000000  write    2     2      1     5    0     0    B3
18   37483  60001008  00000000  write    3     2      1     5    0     3    C1
19   37486  60001000  10000000  read     2     2      0     0    0     0
20   37488  46001008  00300000  write    2     2      0     4    0     0    C2
21   37492  10800010  10000000  write    2     2      0     0    0     1
```

Listing 5.11: Burst write handling with individual transfer response. Coresponding transactions grouped with same letter.

Remote Configuration

All the features regarding the abstraction master and abstraction slave have been tested and documented. The last module to be proven is the control unit. Here only the remote configuration is documented, because it is a remotely triggered automatic configuration. Unfortunately, the AHB monitor buffer was not long enough in Listing 5.12 to trace the access to `G1RTBASE`, but it shows the access to `G1RTCTRL` in line 3, which sends the *Configuration Request Packet*. After that, the configuration is performed with five accesses. The first one in line 7, is a SINGLE read of the page mask. The other 4 are bursts, fetching the entries of the table. Figure 5.13 shows how the Gateways are communicating with each other. After finishing the configuration, the registers of the configured control unit, where the page table entries are stored, are accessed by the LEON3 to verify the configuration. This is happening in line 29 and 34 to 38 in Listing 5.12. In line 39, the remote configuration register of the other Gateway is checked, to verify the reception of the *Configuration Response Packet*. This can be seen because Bit 31 is cleared and Bit 30 is set.

```
1  TIME     ADDRESS  D[31:0]  TYPE  TRANS SIZE BURST MST LOCK RESP
2    36207 4000403C C0A0002C read    3     2     1     0   0     0
3    36213 70A00010 84000000 write   2     2     0     0   0     0
4    36225 400019E0 07100010 read    2     2     1     0   0     0
5    ------------------------------------------------------------
6    36232 400019FC D8030000 read    3     2     1     0   0     0
7    36242 40800400 FFFFF000 read    2     2     0     3   0     0
8    36252 40004080 46001000 read    2     2     1     0   0     0
9    ------------------------------------------------------------
10   36259 4000409C 00000000 read    3     2     1     0   0     0
11   36272 40800404 FFFFFFFF read    2     2     1     3   0     0
12   36273 40800408 47000000 read    3     2     1     3   0     0
13   36274 4080040C FFFFFFFF read    3     2     1     3   0     0
14   36275 40800410 FFFFFFFF read    3     2     1     3   0     0
15   36286 40004040 C0A00030 read    2     2     1     0   0     0
16   ------------------------------------------------------------
17   36293 4000405C 50000018 read    3     2     1     0   0     0
18   36303 40800414 FFFFFFFF read    2     2     1     3   0     0
19   36304 40800418 FFFFFFFF read    3     2     1     3   0     0
20   36305 4080041C FFFFFFFF read    3     2     1     3   0     0
21   36315 40800420 FFFFFFFF read    3     2     1     3   0     0
22   36326 40001A00 DA034000 read    2     2     1     0   0     0
23   ------------------------------------------------------------
24   36333 40001A1C 81E80000 read    3     2     1     0   0     0
25   36343 40800424 FFFFFFFF read    2     2     1     3   0     0
26   36344 40800428 FFFFFFFF read    3     2     1     3   0     0
27   36345 4080042C FFFFFFFF read    3     2     1     3   0     0
28   36346 40800430 FFFFFFFF read    3     2     1     3   0     0
29   36348 C0A00020 FFFF0000 read    2     2     0     0   0     0   *
30   36359 40800434 FFFFFFFF read    2     2     1     3   0     0
31   36360 40800438 FFFFFFFF read    3     2     1     3   0     0
32   36361 4080043C FFFFFFFF read    3     2     1     3   0     0
33   36371 40800440 FFFFFFFF read    3     2     1     3   0     0
34   36373 C0A00024 FFFFFFFF read    2     2     0     0   0     0   *
35   36377 C0A00028 47000000 read    2     2     0     0   0     0   *
36   36381 C0A0002C FFFFFFFF read    2     2     0     0   0     0   *
37   36385 C0A00030 FFFFFFFF read    2     2     0     0   0     0   *
38   36389 C0A00000 00640000 read    2     2     0     0   0     0
39   36393 70A00010 44000000 read    2     2     0     0   0     0
40   36398 10800010 0000007F write   2     2     0     0   0     1
```

Listing 5.12: Remote Configuration procedure of the Gateways. Transactions with asterisk verify table has been loaded.

6 Conclusion

This work has presented a novel virtualization scheme for Network-on-Chip (NoC) based System-on-Chip, to meet the reconfigurable architecture requirement of a Cyberphysical System-on-Chip (CPSoC). This virtualization scheme introduced a second virtualization layer between the processor and the hardware, by embedding it into the Network-on-Chip interconnect of the design. At the current state, the developed layer is just two endpoints put together to a bridge with memory management capabilities. The defined interfaces should allow to integrate the NoC easily into the bridge and turn the NoC into virtualization layer with multiple entry/exit ports. Then it can also be seen as a Multi-Layer AMBA AHB (Advanced High-Performance Bus) bus, mentioned in chapter 2.2.1, with virtualization capabilities. Therefore, when compared to the matrix in Figure 2.5, each entry point into the virtualization layer, can be configured to a set of permitted or denied connections to individual slaves on the entire chip. These logical connections, partition the underlying architecture into multiple logical architectures, for various applications or operating systems, which run simultaneously and use the partitioned hardware as shown in Figure 6.1.

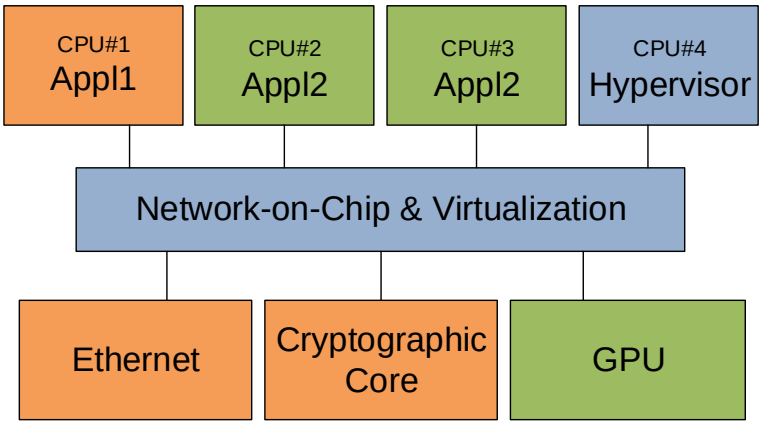

Figure 6.1: Hardware partioning example with a Hypervisor controlling the environment.

Basic configuration methods were implemented to manage the virtualization layer. The performed tests demonstrated, that the implemented controller and the NoC protocol, are capable of configuration over NoC and tunneling AHB requests over the NoC. It was also shown, that

the controller can handle any incoming AHB requests from the LEON3. The modularity and the interface definition between the modules should enable flexibility and reduce complexity for further research into different topics regarding the CPSoC. These topics could be

- interrupt forwarding with Message Signal Interrupts
- dynamic page table handling
- security features to protect the virtualization layer
- monitoring capabilities for NoC and underlying hardware

and more. Interrupt forwarding is necessary, in order to forward an interrupt of a resource to the correct CPU in Figure 6.1. Dynamic page table handling can be useful to reconfigure the architecture during runtime. Security features would be necessary, as currently all endpoints can configure other endpoints at any time. Monitoring capabilities could prove valuable for the hypervisor to have access to network load data from the endpoints and information on the condition of the underlying substrate of a cluster. The mentioned functionalities should display the potential of the CPSoC, and this work should provide a starting point for future research on the topic.

Literature

[AH10] Aguiar, A. ; Hessel, F.: Embedded systems' virtualization: The next challenge? In: *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, 2010. – ISSN 2150–5500, S. 1–7

[AIS09] Argawal, Ankur ; Iskander, Cyril ; Shankar, Ravi: Survey of Network on Chip (NoC) Architectures & Contributions. In: *Journal of Engineering, Computing and Architecture* Bd. 3, 2009

[AMSH12] Aguiar, A. ; Moratelli, C. ; Sartori, M. L. L. ; Hessel, F.: Hardware-assisted virtualization targeting MIPS-based SoCs. In: *2012 23rd IEEE International Symposium on Rapid System Prototyping (RSP)*, 2012. – ISSN 2150–5500, S. 2–8

[ARM99] ARM Limited: *AMBA Specification (Rev. 2.0)*. May 1999

[Bie14] Biedermann, Alexander: *Design Concepts for a Virtualizable Embedded MPSoC Architecture*. Springer Vieweg, 2014

[Cob16] Cobham Gaisler: *GRLIB IP Core User's Manual*. Version 1.5.0. January 2016

[FL11] Flynn, Michael J. ; Luk, Wayne: *Computer System Design: System-on-Chip*. John Wiley & Sons Inc., 2011

[GPP+14] Grammatikakis, M. D. ; Papadimitriou, K. ; Petrakis, P. ; Papagrigoriou, A. ; Kornaros, G. ; Christoforakis, I. ; Coppola, M.: Security Effectiveness and a Hardware Firewall for MPSoCs. In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*, 2014, S. 1032–1039

[Gwe14] Gwennap, Linley. *Microprocessor Report: ThunderX Rattles Server Market*. http://www.cavium.com/pdfFiles/ThunderX_Rattles_Server_Market.pdf. 2014

[GWHB11] Gohringer, D. ; Werner, S. ; Hubner, M. ; Becker, J.: RAMPSoCVM: Runtime Support and Hardware Virtualization for a Runtime Adaptive MPSoC. In: *2011 21st International Conference on Field Programmable Logic and Applications*, 2011. – ISSN 1946–147X, S. 181–184

[IBM] IBM Corperation: *32-bit Processor Local Bus Specification*. 2.9

[KGC12] Kornaros, G. ; Grammatikakis, M. D. ; Coppola, M.: Towards Full Virtualization of Heterogeneous NoC-based Multicore Embedded Architectures. In: *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, 2012, S. 345–352

[KKW+15] Karimi, Naghmeh ; Kanuparthi, Arun K. ; Wang, Xueyang ; Sinanoglu, Ozgur ; Karri, Ramesh: MAGIC: Malicious Aging in Circuits/Cores. In: *ACM Transactions on Architecture and Code Optimization,* Bd. 12, 2015

[KV12] Kliem, D. ; Voigt, S. O.: A multi-core FPGA-based SoC architecture with domain segregation. In: *2012 International Conference on Reconfigurable Computing and FPGAs*, 2012. – ISSN 2325–6532, S. 1–7

[KV13] Kliem, D. ; Voigt, S. O.: An asynchronous bus bridge for partitioned multi-soc architectures on FPGAs. In: *2013 23rd International Conference on Field programmable Logic and Applications*, 2013. – ISSN 1946–147X, S. 1–4

[Mar05] Martin, P.: Design of a virtual component neutral network-on-chip transaction layer. In: *Design, Automation and Test in Europe*, 2005. – ISSN 1530–1591, S. 336–337 Vol. 1

[MIM+13] Muench, D. ; Isfort, O. ; Mueller, K. ; Paulitsch, M. ; Herkersdorf, A.: Hardware-Based I/O Virtualization for Mixed Criticality Real-Time Systems Using PCIe SR-IOV. In: *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, 2013, S. 706–713

[Moo65] Moore, Gordon E. *Cramming More Components onto Integrated Circuits.* http://www.cs.utexas.edu/ fussell/courses/cs352h/papers/moore.pdf. 1965

[MPH15] Münch, D. ; Paulitsch, M. ; Herkersdorf, A.: IOMPU: Spatial Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using Non-transparent Bridges. In: *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, 2015, S. 1037–1044

[MPHH15] Münch, D. ; Paulitsch, M. ; Hanka, O. ; Herkersdorf, A.: MPIOV: Scaling hardware-based I/O virtualization for mixed-criticality embedded real-time systems using non transparent bridges to (Multi-Core) multi-processor systems. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015. – ISSN 1530–1591, S. 579–584

[Ope] OpenCores: *Wishbone.* B4

[PD08] Pasricha, Sudeep ; Dutt, Nikil: *On-Chip Communication Architectures.* Elsevier, 2008

[SDG+15] Sarma, S. ; Dutt, N. ; Gupta, P. ; Venkatasubramanian, N. ; Nicolau, A.: CyberPhysical-System-On-Chip (CPSoC): A Self-Aware MPSoC Paradigm with Cross-Layer Virtual Sensing and Actuation. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015

[SN05] Smith, James E. ; Nair, Ravi: *Virtual Machines. Versatile Plattforms for Systems and Processes.* Elsevier, 2005

[Sta09] Stallings, William: *Operating Systems: Internals and Design Principles.* 6. Pearson, 2009

[SWV+09] Strong, A. W. ; Wu, E. Y. ; Vollertsen, R. P. ; Sune, J. ; Rosa, G. L. ; Sullivan, T. D. ; Rauch, S. E.: *Reliability Wearout Mechanisms in Advanced CMOS Technologies.* Wiley-IEEE Press, 2009

[Wan15] Wang, Junschi: Cyber Physical System on Chip (CPSoC) ASIC Specification v2.6 / Institut of Computertechnik Technology, TU Wien. 2015. – Forschungsbericht

Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, am 1. Oktober. 2016

Elvin Sebastian