Computer Science Department

University of Crete

B.Sc. Thesis

IOMMU Support for Virtual-Address Remote DMA in an ARMv8 environment

Author:
Antonios Psistakis

Supervisor:
Prof. Manolis G.H. Katevenis

March 20, 2017

The thesis is submitted in fulfillment of the requirements for the degree of B.Sc. in the Computer Science Department (CSD), University of Crete (UoC). It took place in the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS), Foundation of Research and Technology - Hellas (FORTH), from June 2016 to December 2016 and was funded by the distinguished Undergraduate Scholarship "Stelios Orphanoudakis".

IOMMU Support for Virtual-Address Remote DMA in an ARMv8 environment

by Antonios Psistakis

Supervisor: Prof. Manolis G.H. Katevenis

Abstract

In complex systems with many compute nodes consisting of many CPUs, that are coherent inside each node, one issue is to achieve coherence between them in an efficient and valid way. Unimem system [1] addresses this issue and in order to support it, it suggests a virtualized system (virtual global address space) that is able to handle this. This approach requires the use of the I/O Memory Management Unit (IOMMU) in each node.

The goal of this thesis is to support this approach, by testing and using successfully the IOMMU of one node. In order to do that, we used ARM's IOMMU, which is called System Memory Management Unit (SMMU), that, in general, enables the translation of the virtual addresses to their corresponding physical addresses. Using the SMMU is complex - the documentation when it comes to Linux support is not very clear, so we had to implement our own modules, in order to test and use the SMMU.

First, we tested the SMMU in the Processing System (PS) of the Xilinx Zynq UltraScale+ MPSoC [2], by writing a module that adds a virtual address and its corresponding translation to a physical address as an entry to the SMMU. Then, we triggered a DMA transaction sending data to the virtual address that we added to the SMMU and we noticed that our DMA transaction passed through the SMMU in order to translate the virtual address to the corresponding physical address. Later, we tested the SMMU, by doing the same but this time, by triggering the transaction from the Programmable Logic (PL). We noticed, again, that the transaction passed through the SMMU in order to translate the virtual address of the destination. Finally, we created a module that makes possible to trigger transactions from the PL, without the need of mapping the entries of virtual addresses and physical addresses before – we managed to have the SMMU being able to handle and translate all virtual addresses related to a user process, by setting the SMMU with the pointer of the page table of the user process.

In general, the result for all scenarios and modules mentioned above is that we managed to have the SMMU working. Due to the limited time of this thesis, future work for understanding and using more features of the SMMU is expected.

Dedicated to the memory of a
great, kind, smart and
charming person and loyal
friend to me, my grandmother
Maria A. Psistaki (1925 - 2016).

Acknowledgements

The work for this thesis took place in the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS), Foundation of Research and Technology - Hellas (FORTH), from June 2016 to December 2016 and was funded by the distinguished Undergraduate Scholarship "Stelios Orphanoudakis".

I would like to thank and give credit to a lot of people who helped me in many ways throughout the thesis.

First of all, I would like to start by thanking my supervisor Professor Manolis G.H. Katevenis, who trusted me with such new, exciting and perhaps challenging task and advised me many times during the time I was working on my thesis.

Then, I would like to thank Dr. Fabien Chaix. He was very supportive from the beginning of my thesis until the last day. We worked together in many parts of this thesis, like: Petalinux and booting Linux from our Trenz Boards, the theory of the IOMMU and the implementation of it (the kernel modules - Chapter 4). It was his suggestion, that parts (bits) of the StreamID are the AXI ID and the Master Device (in our tests, the HPC0 - Chapter 5). He also helped me with the understanding of how to use Vivado and SDK, in order to create our own Linux images. Dr. Chaix was actually involved in everything I was involved. I thank him for being patient with me and also for his advices and support.

Also, I would like to thank Dr. Vassilis Papaefstathiou. We met almost at the end of my thesis, but we worked together in the understanding of the IOMMU (devices, groups, domains, SMRn, S2CRn etc), how it works and how we could "pass" the page table of a process to the SMMU. The focus of the work we did together was on the last kernel module (Chapter 4) – he explained me how the ioctl works and we investigated together the translation/context faults we had at first and we wanted to fix, before we manage to have a translation table of a user process in a context bank of the SMMU working. I thank him for his support and persistence in guiding me to become better.

For the support in the first three kernel modules (Chapter 4), I would like to thank my colleague John Vardas. He helped me with debugging and also he did some changes/additions to some of the modules, that made them work more efficiently. In both SMMU modules (with transactions from the PS and the PL), he run many tests and helped us understand many things about the SMMU.

I would like to thank Dr. Nikolaos Chrysos, for helping John Vardas and me, with the discussions we had in order to make the third kernel "module" (the module that allowed us to have a single-page transaction from the PL, using the SMMU) work.

For the system software support, I would like to thank Dimitris Poulios (M.Sc.) and Charalampos Aronis. Dimitris helped us boot Linux from the QSPI and the SD Card of the Trenz board and he was always helping us, when we needed Linux

support. Charalampos helped us understand the kernel environment and he was the person who gave me guidance, in order to use the IOMMU API and build our first SMMU module for testing purposes.

For the hardware support, I would like to thank Konstantinos Harteros (M.Sc.), for the discussions we had because of his experience working with an IOMMU in the past, Antonis Psathakis (M.Sc.), for his support with Xilinx Vivado and SDK and Aggelos Ioannou (M.Sc.), for providing us the hardware files that we needed, in order to create our own Linux images and use them to boot from a Trenz board.

I greatly appreciate the feedback that Dr. Fabien Chaix and Panos Peristerakis gave me for my thesis before I submit it.

Last but not least and while I am hoping I did not forget to thank personally anyone that I should have, I would like to thank everyone in the CARV Laboratory of the Institute of Computer Science, FORTH, for all the support throughout my thesis. I could not be more thankful.

Contents

Chapter 1

Introduction

In this chapter, we will describe the motivation behind this thesis, the contributions and the context/background that was needed for the purposes of this thesis, in terms of the Hardware and the Software.

1.1 Motivation

As the years go by, when it comes to the Computer Science and the industry, it seems there is a direction in creating more complex sets of computers. Looking the image below, we can see in a very simplistic way that many nodes, or "Coherent Islands", as they are described in Unimem [1], are trying to communicate. It is obvious that the coherence between all CPUs in one node is somehow granted. The problem begins when we have many nodes and we want to achieve coherence between them – the more nodes the bigger the problem. This is the issue that the Unimem is trying to address, in order for big and complex systems, as we described, to be able to communicate in a valid and efficient way.

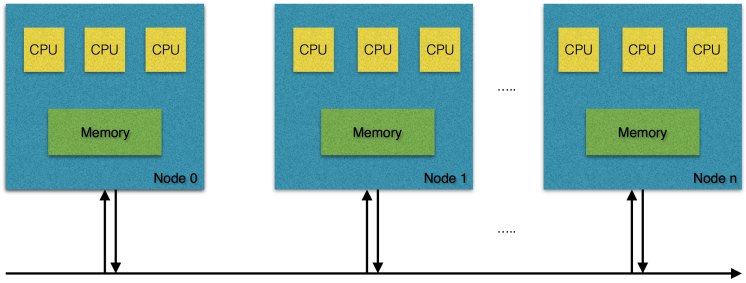

Figure 1.1 – A system with many nodes that are connected

The Unimem suggests that if we want such system to be coherent, each node that needs to access information that belongs to the memory of another node, it should request the information from the other node, without keeping a copy to its memory. In order to achieve this goal, Unimem proposed a virtualized system that distinguishes all nodes and more specifically, a virtual global address space. This has as a result the necessity of an "arbiter" in each node, that will handle all incoming transactions in order to access their memory. This arbiter will be no other than the I/O Memory Management Unit (IOMMU).

Part of this thesis would, in a way, support the implementation of the Unimem.

1.2 Contributions

This thesis is trying to support the Unimem, as described in Section 1.1, by understanding how the SMMU (ARM's IOMMU) works and how we could use it in order to achieve virtual-address remote Direct Memory Access (DMA) transactions in one or between multiple nodes – for the purposes of this thesis only one node was considered in the experiments, but the information and results generated by this thesis could also support future implementations for multiple nodes. Having the understanding and information needed, we could proceed to the implementation of the kernel modules that would test and use the SMMU. In order to achieve these goals, the work for this thesis consisted of three different parts.

The first part is the background/theory of ARM's SMMU, which includes all the information and knowledge that was collected and used in order to proceed with the implementation of the kernel modules. Chapter 3 has a detailed description of all this information.

The second part is the kernel modules that were implemented in order to test and use the SMMU of the Zynq UltraScale+ MPSoC. Chapter 4 has a detailed description of all the different modules that were implemented during this thesis.

The third part is the experiments that took place for all the kernel modules that are described in Chapter 4 and the notes we used, in order to reach our big goal that was having the SMMU ready (initialized) to support the virtual-address remote DMA transactions. Detailed information about the experiments can be found in Chapter 5.

1.3 Context

1.3.1 Hardware

1.3.1.1 Trenz board

The hardware that was used in order to achieve our goal was basically a TET0808 Trenz board with the FPGA: XCZU9EG-FFVC900-1 [3]. It is a MPSoC module integrating a Xilinx Zynq UltraScale+, with 2 Giga Byte DDR4 SDRAM with 64-Bit width, 64 MByte Flash memory for configuration and operation, 20 Gigabit tranceivers, and switch-mode power supplies for all on-board voltages.

1.3.1.2 Zynq UltraScale+

Zynq UltraScale+ MPSoC is the Xilinx second-generation Zynq platform, combining a processing system (PS) and user-programmable logic (PL) into the same device. The Zynq UltraScale+ MPSoC has four different power domains.

- Low-power domain (LPD).
- Full-power domain (FPD).

It also has the PL power domain (PLPD) and the Battery power domain (BPD), that we will not use for the purposes of this thesis.
The Zynq UltraScale+ MPSoC PS block has three major processing units:

- Cortex-A53 application processing unit (APU)—ARM v8 architecture-based 64-bit quad-core or dual-core multiprocessing CPU
- Cortex-R5 real-time processing unit (RPU)—ARM v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM)

The third processing unit is the Mali-400 graphics processing unit (GPU) with pixel and geometry processor and 64KB L2 cache. For the purposes of this thesis we will not use it.

As we can see below (Figure 1.2), in the top-level block diagram of the Zynq UltraScale+ MPSoC, part of the Processing System (PS) is the APU that consists of four Cortex-A53 processors that will run the system and more specifically they will run the Linux that we will use in order to achieve our goals, using the IOMMU. Also, part of the PS is the RPU that consists of two Cortex-R5 real-time processors, that we will not use for the purposes of this thesis. As we can see, the SMMU and the CCI (Cache Coherent Interconnect) are in the same block in the top-level block diagram – that of course does not mean that they indeed are part of the same block, but that they collaborate in order to have coherent accesses to the memory. For the purposes of the thesis and because of the limited time, we will not focus on coherence issues. Also, as we will see later, we will use both the LPD-DMA (Low-Power Domain DMA) and the FPD-DMA (FPD-DMA) that are parts of the PS, in order to trigger transactions that we want to go through the SMMU, when we have virtual addresses. Last but not least, part of the top-level diagram is the Programmable Logic (PL), where a user can program the FPGA,

add blocks that use the PS etc. In our case, we will use the PL to trigger transactions, that we want to go through the SMMU, from there.

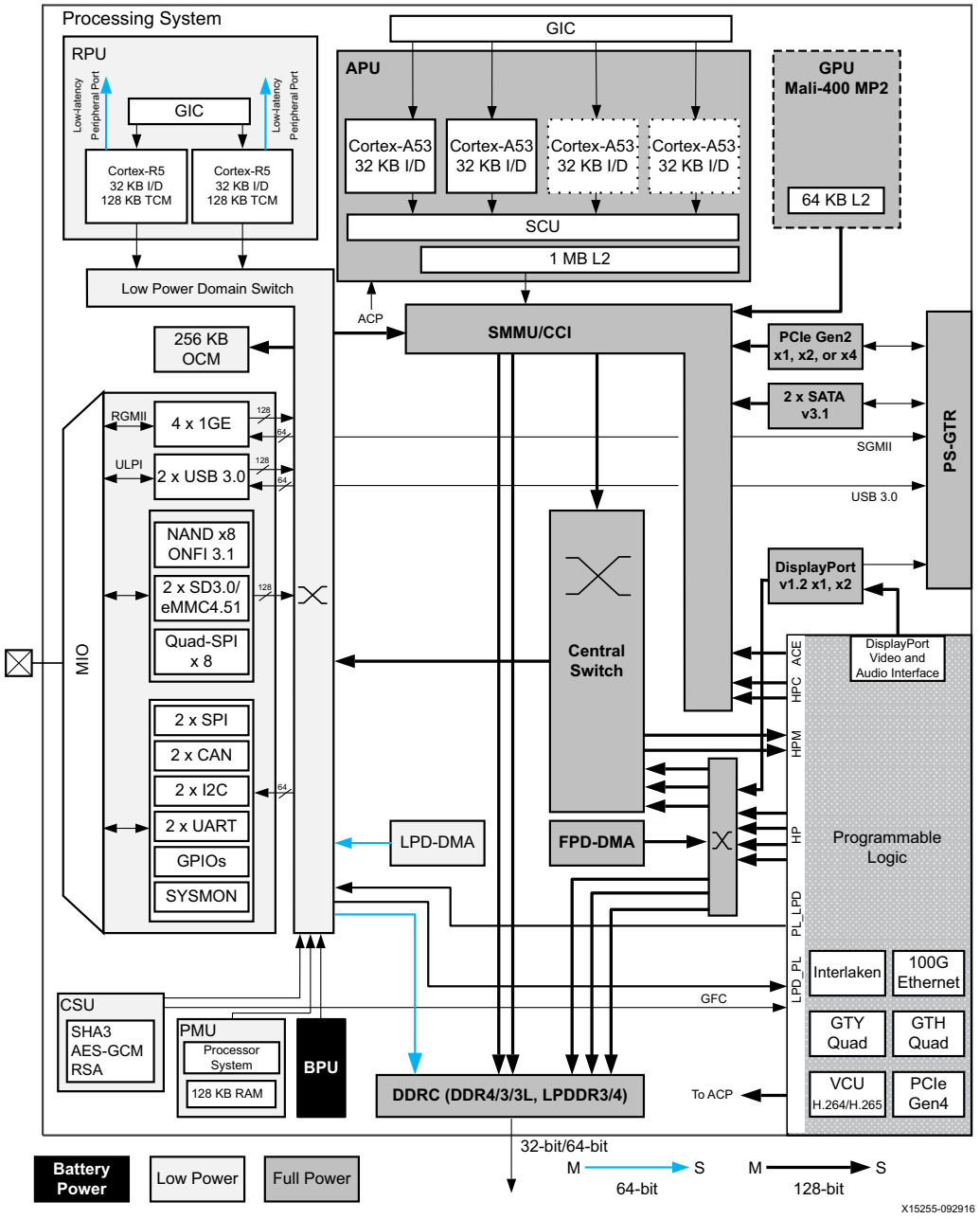

Figure 1.2 – Zynq UltraScale+ MPSoC Top-Level Block Diagram
Source: Xilinx [2]

1.3.1.3 Memory Management

For the purposes of this thesis, the most interesting part of the Zynq UltraScale+ MPSoC is the System Memory Management Unit (SMMU) block. Before we continue with the remaining of the thesis, we should explain the basic idea of the Memory Management Unit (MMU).

The Memory Management Unit (MMU)

The Memory Management Unit (MMU), in general, is a computer hardware unit having all memory references passed through itself, primarily performing the translation of virtual memory addresses to physical addresses. It is usually implemented as part of the CPU, but it also can be in the form of a separate integrated circuit.

A MMU can effectively perform virtual memory management, handling at the same time memory protection, cache control, bus arbitration and, in simpler computer architectures (especially 8-bit systems), bank switching.

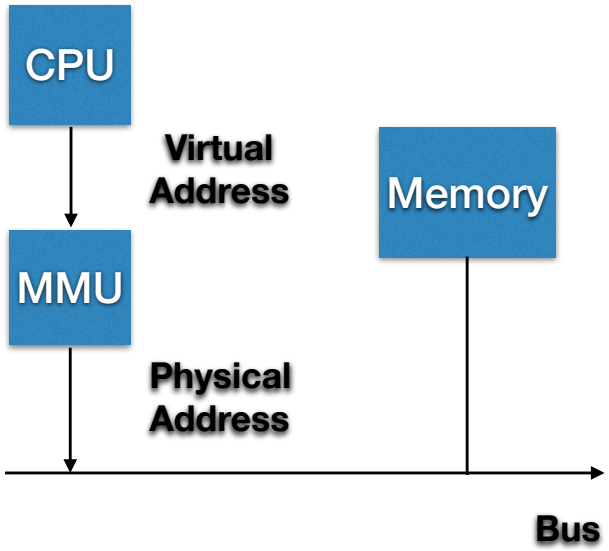

Figure 1.2 – A simple use of the MMU

The I/O Memory Management Unit (IOMMU)

I/O Memory Management Unit (IOMMU) is a MMU that connects a Direct Memory Access-capable (DMA-capable) I/O bus to the main memory. IOMMU is basically responsible to map device-visible virtual addresses (also called device addresses or I/O addresses in this context) to physical addresses.

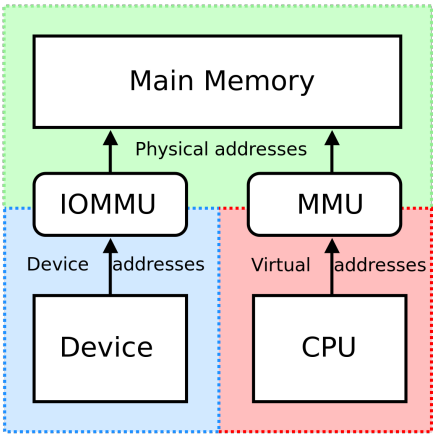

Figure 1.3 – Comparison of the IOMMU to the MMU

Source: Wikipedia

The System Memory Management Unit (SMMU)

In order to address some issues (e.g.: memory fragmentation, multiple DMA-capable masters, system integrity guarantee) and limitations, the ARM Architecture Virtualization Extensions introduced the System Memory Management Unit (SMMU) concept to the ARM Architecture. In other words, as we mentioned before, ARM's IOMMU is called SMMU.

The SMMU performs address translation of an incoming AXI (Advanced eXtensible Interface) [4] address and AXI ID (mapped to context) to an outgoing address (physical address - PA), based on address mapping and memory attribute information held in translation tables.

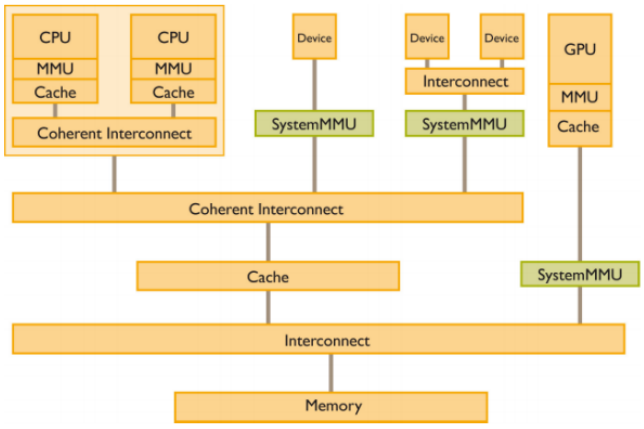

Figure 1.3 – Examples of where a SMMU could be located in a system. Coherent interconnects ensure cache coherence between masters.
Source: ARM

1.3.2 Software

Most of the effort for this thesis was actually spent on Software aspects. That is because, in general, the configuration of the SMMU is difficult to understand and handle, without proper documentation and experience.

1.3.2.1 Linux

We chose to use the Linux OS as the Operating System for the purposes of this thesis - the Linux version was 4.4.0.

At first, we used the pre-built images provided by Trenz in order to flash the QSPI (Queued Serial Peripheral Interface – a type of SPI controller that uses a data queue to transfer data across the SPI bus) memory with Linux. Although the pre-built images worked, it was not enough for our purposes because those pre-built images had a very minimal version of Linux. When we tried to find another way to create our own Linux images, the Petalinux method was the suggested one by Trenz and Xilinx. Although, again, there was a document with the process of creating images via Petalinux environment, the steps and the details were not clear, and after spending much time trying to make them work, we decided to use part of the working files (or images) like the device tree (.dtb) and the first-stage-boot-loader (FSBL) and the rest of the images, that we needed, from other projects that were working already.

As mentioned before, we used to boot from the QSPI (that we were flashing using the JTAG), but currently, since we now have more extensive base boards that support the SD card, we are booting from there. Appendix A describes the process of booting from the SD Card.

1.3.2.2 Xilinx

We used Vivado, a software suite produced by Xilinx for synthesis and analysis of HDL designs and SDK (Software Development Kit), that is the Integrated Design Environment for creating embedded applications on any of Xilinx's microprocessors like the Zynq UltraScale+ MPSoC.

We are using Xilinx Vivado to export the bitstream (.bit) and the hardware design/description file (.hdf), and the Xilinx SDK to create the device tree (.dtb) and the first-stage-boot-loader (.fsbl) from the exported hardware design/description file (.hdf). In Appendix A (part of Section A.1.1), we can see how to generate the FSBL and in Appendix C, we can see how to generate the device tree. We used the 2016.2 versions of the Xilinx Vivado and SDK. Xilinx tools also helped us create the BOOT.bin image, that we are using to boot the Linux. In Appendix A (Section A.1.1) we see how we create the BOOT.bin, with commands like "bootgen" being part of the Xilinx tools.

1.3.2.3 Modules/Drivers

At this point, since we mentioned the "modules" and "drivers" before and they are a big part of this thesis, it is good to define them before we continue.

A device driver (commonly referred to simply as a driver) is a computer program that operates or controls a particular type of device that is attached to a

computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A module is a piece of code that can be loaded and unloaded into the kernel upon demand. Modules extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

In order to test and use the SMMU, we had to write or modify some drivers and modules, which will be described in Chapter 4 and Appendix D.

Below, in Figure 1.3, we can see how a system can use the drivers and the modules.

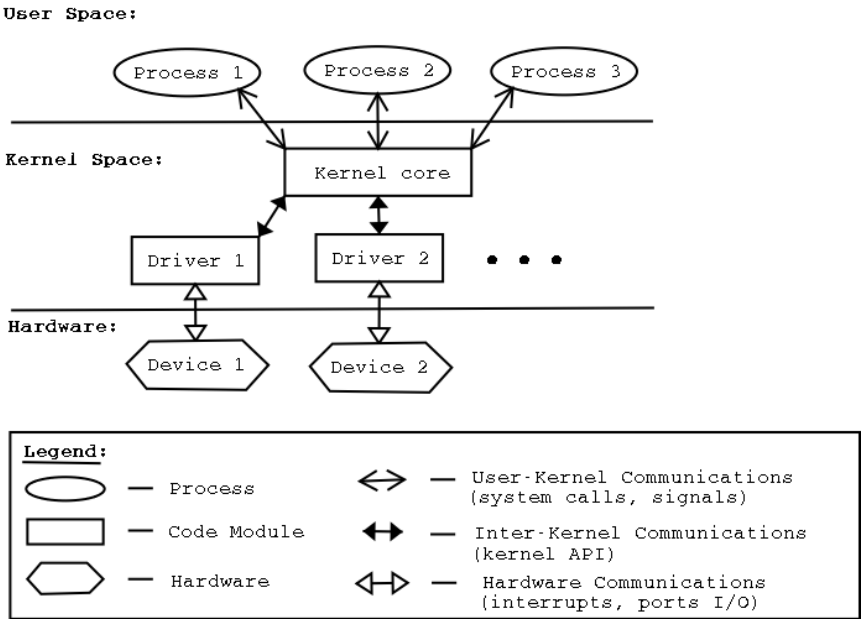

Figure 1.3 – The Kernel, The Processes And The Hardware.
Source: haifux.org

Chapter 2

Related work

The truth is that when we were looking for information or previous work about anything related to the implementations and uses of the IOMMU (or the SMMU) with Linux support, we could not find many things.

There is a book, called **"Linux Device Drivers"**, which introduces the idea of the Input/Output Memory Management Unit (IOMMU) and the virtualization. There are, also, some examples of information coming from companies, who implement and make use of the IOMMU.

In this chapter, we will give a brief description of all the references/information coming from the following sources:

1. Linux Device Drivers
2. Intel IOMMU
3. AMD IOMMU
4. ARM IOMMU

2.1 Linux Device Drivers, Third Edition

In the Linux Device Drivers [5], and more particularly in Chapter 15 "Memory Mapping and DMA" it is mentioned that some architectures can provide an IOMMU that remaps addresses between a bus and main memory. An IOMMU can be helpful in many ways (i.e.: making a buffer scattered in memory appear contiguous to the device), but programming the IOMMU is an extra step that must be performed when setting up DMA operations.

Later, it suggests that a DMA mapping is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device. One approach is to get that address with a simple call to $virt\_to\_bus$, but there are many good reasons for avoiding this approach. The first of those is that reasonable hardware comes with an IOMMU that provides a set of mapping registers for the bus. The IOMMU can arrange for any physical memory to appear within the address range accessible by the device, and it can cause physically scattered buffers to look contiguous to the device. Making use of the IOMMU requires using the generic DMA layer and the $virt\_to\_bus$ is not able to handle this.

According to the book not all architectures have an IOMMU; in particular, by that time, the popular x86 platform had no IOMMU support – a properly written driver did not need to be aware of the I/O support hardware it is running over.

2.2 Intel IOMMU

In Intel systems, the use of the IOMMU is part of the "Intel VT" (Virtualization Technology). More specifically, Intel's "Intel VT-d" (Intel Virtualization Technology for Directed I/O), makes direct access to a PCI device possible for guest systems with the help of the Input/Output Memory Management Unit (IOMMU) provided. This allows a LAN card to be dedicated to a guest system, which makes possible to allow better attainment of increased network performance beyond that of an emulated LAN card. Of course, once such a direct access system has been implemented, live migration of the guest operating system is no longer possible. For instance, VMware can be configured for use with an activated Intel VT-d system using VMware VMDirectPath for direct access to PCI cards.

Another reference on the Intel IOMMU, when it comes to Linux, is that the Intel IOMMU Support is related to the IOVA (IO Virtual Address) generation. "Well behaved" drivers call pci_map_* function before sending command to the device that needs to do a DMA transaction. Once the DMA transaction is completed and the mapping is no longer required, device performs a pci_unmap_* call to unmap the region that was mapped before. The Intel IOMMU driver allocates a virtual address per domain. Each PCI (PCI Express) device has its own domain, which is translated to protection. Devices under p2p (peer to peer) bridges share the virtual address with all devices under the p2p bridge due to transaction id aliasing for the p2p bridges. The IOVA generation is pretty generic - Intel used the same technique as used in $vmalloc$ function, but these are not global address spaces, but are separate for each domain. Different DMA engines may support different number of domains.

Intel, also, allocates guard pages with each mapping, so it can be attempted to catch any overflow that might happen.

More information about the Intel IOMMU can be found in the Intel Virtualization Technology for Directed I/O specifications [6].

2.3 AMD IOMMU

In the case of the AMD processors, the I/O Memory Management Unit (IOMMU) extends the AMD64 system architecture by adding support for address translation and system memory access protection on DMA transfers from peripheral devices. The AMD IOMMU also helps filter and remap interrupts from peripheral devices.

The AMD IOMMU enables several significant system-level enhancements:

- Legacy 32-bit I/O device support on 64-bit systems (generally without requiring bounce buffers and expensive memory copies)
- More secure user-level application access to selected I/O devices
- More secure virtual machine guest operating system access to selected I/O devices

and can be used to replace the existing Graphics Address Remapping Table (GART) mechanism. It can also be used to remap addresses above 4 GB for I/O

devices that do not support 64-bit addressing and allow a guest OS running on a virtual machine to have direct control of a device. It can provide page granularity control of device access to system memory, a device direct access to user space I/O and direct delivery of interrupts to a guest Operating System. Also, it can filter and remap interrupts and share process virtual address space with selected peripheral devices.

The IOMMU can be thought of as a generalization of two facilities included in the AMD64 architecture: the GART and the Device Exclusion Vector (DEV). The GART provides address translation of I/O device accesses to a small range of the system physical address space, and the DEV provides a limited degree of I/O device classification and memory protection. With appropriate software support, the IOMMU can emulate the capabilities of the GART or DEV.

The IOMMU extends the concept of protection domains (or domains) first introduced with the AMD64 DEV. The IOMMU allows each I/O device in the system to be assigned to a specific domain and a distinct set of I/O page tables. When an I/O device attempts to read or write system memory, the IOMMU intercepts the access, determines the domain to which the device has been assigned, and uses the TLB entries associated with that domain or the I/O page tables associated with that I/O device to determine whether the access is to be permitted as well as the actual location in system memory that is to be accessed.

The IOMMU may include optional support for remote IOTLBs (input/output translation look-aside buffers, that speed-up the address resolution). A trusted I/O device with IOTLB support can cooperate with the IOMMU to maintain its own cache of address translations. This creates a framework for creating scalable systems with an IOMMU in which I/O devices may have different usage models and working set sizes. IOTLB-capable I/O devices contain private TLBs tailored for their own needs, creating a scalable distributed system of TLBs. The performance of IOTLB-capable I/O devices is not limited by the number of TLB entries implemented in the IOMMU. A peripheral with an IOTLB may issue untranslated addresses or pre-translated addresses that are determined from IOTLB entries. Pre-translated addresses are not checked by the IOMMU except to validate that the peripheral has the IOTLB enable bit set in the corresponding Device Table Entry.

Optionally, the IOMMU may include support for Peripheral Page Requests (PPR) for peripherals that use Address Translation Services (ATS). This creates a mechanism for peripherals and software to reduce the need for pinned pages during I/O. The IOMMU may include optional support for interrupt virtualization. This uses a virtualized guest APIC (Advanced Programmable Interrupt Controller), which is a family of interrupt controllers, (one implementation of a guest APIC is the Advanced Virtual Interrupt Controller) with memory tables to deliver interrupts to guest VMs.

More information about the AMD IOMMU can be found in the AMD I/O Virtualization Technology (IOMMU) specification [7].

2.4 ARM IOMMU

ARM's I/O Memory Management Unit (IOMMU) is called SMMU (System Memory Management Unit) and is the IOMMU that was described in Section 1.3. A more detailed information about the SMMU Architecture and Operation can be found in Chapter 3.

Most of the information related to the SMMU is coming from the corresponding manual of the ARM IOMMU [8] and the Xilinx UltraScale+ TRM (Technical Reference Manual) [2].

Chapter 3

Overview of the ARM SMMU

In this chapter, we will describe the first part of the contributions of this thesis, which is the background/theory related to the SMMU. More specifically, we will present an overview of the information related to the Architecture and the Operation of the SMMU, that we found reading the manuals of the ARM SMMU version 2 [8] and the Xilinx Zynq UltraScale+ MPSoC [2].

3.1 Architecture

The ARM SMMU architecture supports:

- Aarch32 short descriptor (up to a 32-bit virtual-address and up to 32-bit physical-address)
- Aarch32 long descriptor (up to a 32-bit virtual-address and up to 40-bit physical-address)
- Aarch64 descriptor (up to a 49-bit virtual-address and up to 48-bit physical-address)

An implementation of the ARM SMMU architecture can provide multiple transaction contexts, that apply to specific streams of transactions and single or two stage translation. For any stage of translation, it can provide multiple levels of address lookup for fine-grained memory control. Also, it can provide fault handling, logging, signaling and debug and optional performance monitoring functionality.

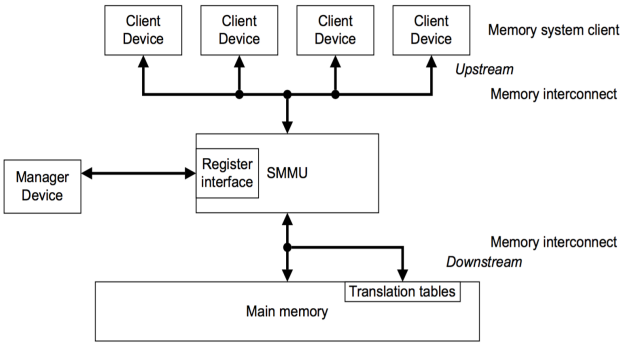

Figure 3.1 – An ARM SMMU in the memory system

Source: ARM [8]

As we can see in Figure 3.1, which shows an implementation of an ARM SMMU in the memory system, one or more Client devices are connected to the SMMU through the memory. Client devices are described as being the "upstream" of the SMMU. The connection between the SMMU and the client devices is the upstream bus. The SMMU is connected to the rest of the memory system, through the main memory. The rest of the memory system is described as being the "downstream" of the SMMU. The connection between the SMMU and the rest of the memory system is the downstream bus. A client device issues a transaction request to the SMMU. The SMMU processes that transaction and returns a response to the client.

3.1.1 ARM Exception levels and Execution states

The ARMv8 exception model defines the Exception Level ELn, where n can have the values 0-3. Software execution privilege increases with the increase in the value of n, with EL0 software having the lowest level of privilege. Execution at EL0 is described as unprivileged execution. It is implementation defined whether a processor implementation includes EL2 or EL3.

The typical use of the different Exception levels is:

- EL0 Application software $\rightarrow$ Secure state (i.e.: The processor can access both the Secure and the Non-secure memory address space. When executing at EL3, the processor can access all the system control resources.) or Non-secure state (i.e.: The processor can access only the Non-secure memory address space and cannot access the Secure system control resources.)
- EL1 Operating system $\rightarrow$ Secure or Non-secure state
- EL2 Hypervisor $\rightarrow$ Non-secure state only
- EL3 Secure monitor

3.1.2 ARM translation regimes

The ARM architecture supports different translation regimes and stages for the AArch32 (32-bits) and the AArch64 (64-bits). Figure 3.1 (next page) shows the ARM architecture translation regimes for memory accesses made from AArch64 state, where:

- VA is the Virtual Address
- PA is the Physical Address
- IPA is the Intermediate Physical Address (used in virtualization context)

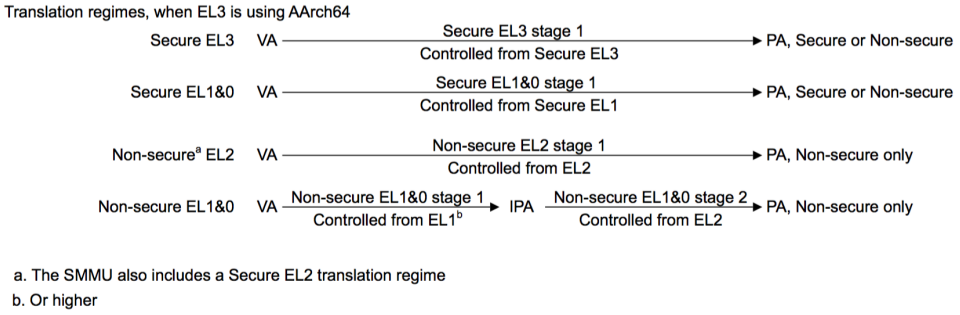

Figure 3.1 – ARM processor architecture AArch64 translation regimes and stages

Source: ARM [8]

In the ARM processor architecture, the behavior of a PE (Processing Element) that is executing in AArch32 state is defined in relation to different PE modes. Most of these modes have the level of execution privilege that is appropriate to an operating system. In a processor implementation that includes EL3, the Exception level of these modes can depend on whether EL3 is using AArch32 or AArch64, something we can also confirm from the Figure 3.1 below.

PE mode	Non-secure state	Secure state		Privilege level
		EL3 using AArch64	EL3 using AArch32	
User	EL0	EL0	EL0	PL0
System, Supervisor, Abort, Undefined, IRQ, FIQ	EL1	EL1	EL3	PL1
Hyp[a]	EL2	-	-	PL2
Monitor[b]	-	-	EL3	PL1

a. Implemented only in Non-secure state, and only present when EL2 is using AArch32.
b. Implemented only in Secure state, and only present when EL3 is using AArch32.

Figure 3.1 – Mapping of AArch32 PE modes to Exception levels

Source: ARM [8]

Figure 3.1 shows the ARM architecture translation regimes for memory accesses made from AArch32 state.

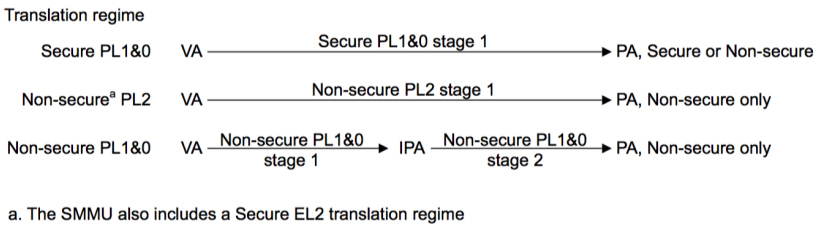

Figure 3.1 – ARM processor architecture AArch32 translation regimes and stages

Source: ARM [8]

3.1.3 SMMU Translation Schemes

The SMMUv2 translation schemes support address translations required by both the AArch32 and AArch64 Execution states, for each translation regime, as follows:

The Aarch32 short descriptor uses 32-bit entries, or descriptors, in its translation tables. Also, this descriptor uses up to a 32-bit VA and 32-bit (or 40 bits with loss of granularity) PA and is compatible with the ARMv7 architecture.

The Aarch32 long descriptor uses 64-bit entries in its translation tables. Also, this descriptor uses up to a 32-bit VA and 40-bit IPA or PA and is compatible with the ARMv7 architecture.

The Aarch64 descriptor uses 64-bit entries in its translation tables. Also the Aarch64 descriptor uses up to a 49-bit VA and 48-bit, as two independent address ranges and is defined by the ARMv8 architecture.

Translation granule is the smallest region of input address space that an SMMU implementation can be configured to support. It defines both the maximum size of a single translation table and the memory page size, that is, the granularity at which attributes can be assigned to memory regions.

Aarch32 translation scheme: supports a translation granule of 4 KB.

Translation regime	Translation context	Note
Secure EL1	Stage 1 context with stage 2 bypass	Either of: • AArch32 Short-descriptor. • AArch32 Long-descriptor.
Non-secure EL2		AArch32 Long-descriptor, hypervisor context (HYPC) only.
Non-secure EL1&0, no EL2		Either of: • AArch32 Short-descriptor. • AArch32 Long-descriptor.
Non-secure EL1&0, with EL2	Stage 1 followed by stage 2	Stage 1 can use either AArch32 translation scheme. Stage 2 can use either of: • AArch32 Long-descriptor translation scheme. • AArch64 translation scheme.

Figure 3.1 – AArch32 translation regimes

Source: ARM [8]

Aarch64 translation scheme: supports a translation granule of 4 KB, 16 KB, or 64 KB.

Translation regime	Translation context	Note
Secure EL3	Stage 1 context with stage 2 bypass	Monitor context only.
Secure EL1		-
Non-secure EL2		*Hypervisor context* (HYPC) only.
Non-secure EL1&0, no EL2		-
Non-secure EL1&0, with EL2	Stage 1 followed by stage 2	For stage 2, VMSAv8 Long-descriptor

Figure 3.1 – AArch64 translation regimes

Source: ARM [8]

3.1.4 SMMU address support

The SMMUv2 architecture supports virtual addresses up to 32 bits when held in 32-bit registers, up to 49 bits when held in 64-bit registers (when bit[49] is valid, it determines the Translation Table Base Register (TTBR)). It also supports Translation granule sizes of 4 KB, 16 KB, and 64 KB - it is implementation defined which of these granule sizes are supported.

3.1.5 SMMU address handling sequence

A transaction bypasses SMMU translation or it does not. In certain cases, the input address is sign-extended. For each stage of translation, the SMMU performs checks on the input address and the output address (i.e.: whether a stage 1 or stage 2 translation context bank is used, the value of SMMU_CBn_SCTLR.M for the context bank, whether a 32-bit or 64-bit descriptor format is used).

3.2 SMMU Operation

In this section we will describe the steps that the SMMU performs on receiving a memory access request. At the memory system level, in performing address translation, an SMMU controls:

- Security state determination, which identifies whether a transaction is from a Secure or Non-secure device
- Context determination, which identifies the stage 1 or stage 2 context resources the SMMU uses to process a transaction. In some cases, the configuration settings for a transaction mean that transaction bypasses the translation process, or is faulted
- Memory access permissions and determination of memory attributes
- Memory attribute checks
- TLB operation

Figure 3.1 shows the SMMU partitioning and the different SMMU transactions.

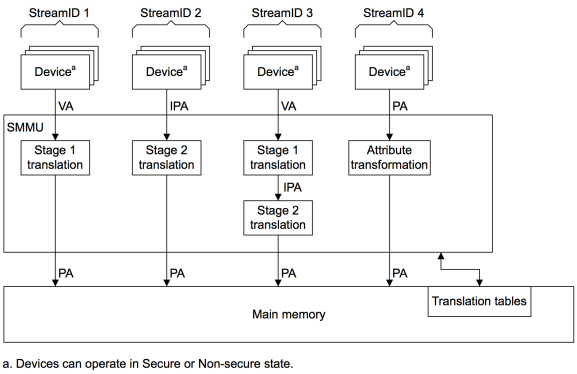

Figure 3.1 – SMMU partitioning

Source: ARM [8]

The **Stream Identifier (StreamID)** uniquely identifies a stream of transactions that can originate from different devices, but are associated with the same context, and are therefore subject to the same type of SMMU processing. The mechanism by which the StreamID is obtained from the incoming bus transaction is implementation defined.

The **Memory attributes** can be modified by SMMU. In some cases, SMMU performs attribute transformation only, with no address translation. In this case, the output physical address is the same with the input virtual address.

An access to the SMMU is referred to as a **transaction**. A client transaction is an access by a client device, that the SMMU is to process. A configuration transaction is a device access to a register in the SMMU configuration address space.

Figure 3.1 below shows the generic SMMU process flow.

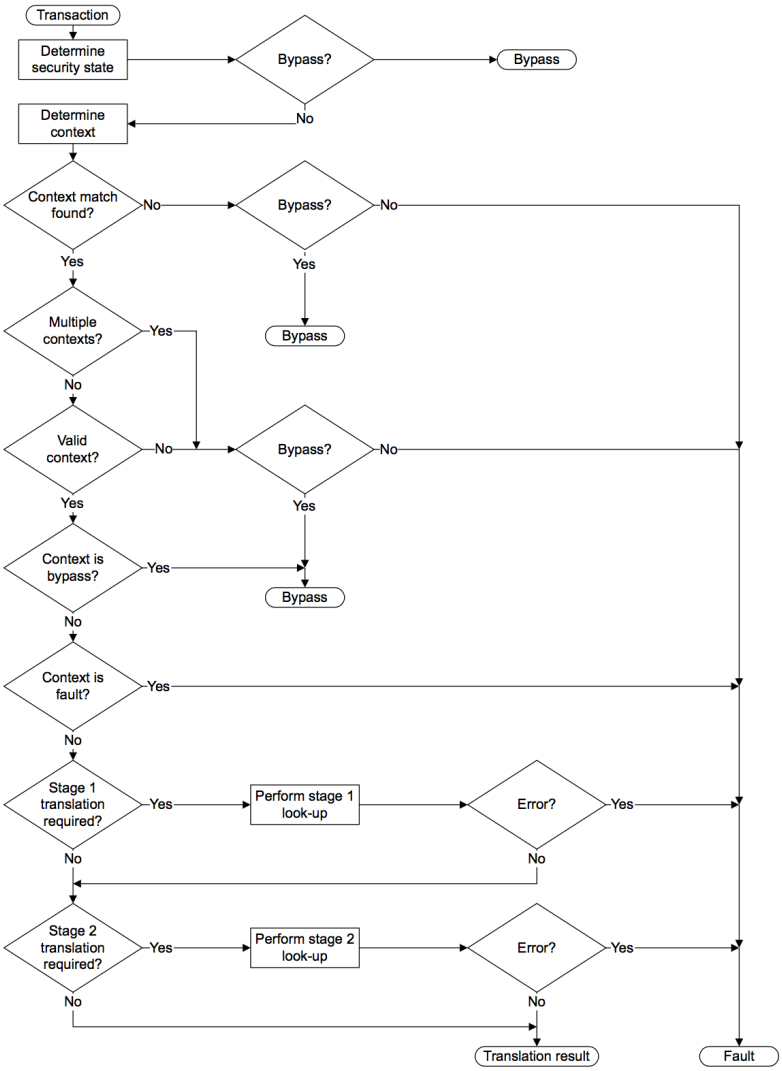

Figure 3.1 – SMMU process flow

Source: ARM [8]

3.2.1 Context determination

The SMMU processes a transaction in one of the following ways:

- Bypass address translation but transform attributes
- Fault the transaction
- Require the resources of either one or two translation context banks, each having its own set of translations, attributes and permissions, that apply to a transaction being processed by that context bank

The context determination actually determines the resources the SMMU uses to process a transaction. The SMMU can process transactions from multiple sources, potentially using a different context for each transaction.

The **Transaction stream** is a sequence of transactions associated with a particular thread of activity in the system. The **StreamID** is derived from transaction identification information, such as:

- The transaction ID
- The read and write status
- The Security state of the upstream bus transaction

The number of implemented StreamID bits in SMMUv2 lies in the range: 0-16 and in SMMUv1 lies in the range: 0-15. A zero-bit StreamID might exist if the source of a single StreamID has a dedicated SMMU.

Stream mapping is the process of mapping a StreamID to the associated Stream-to-Context register (SMMU_S2CRn), that defines the processing required for the stream. There are three (3) Stream mapping mechanisms defined by SMMUv2:

- **Stream matching**
  The StreamID is looked up in the set of the Stream Match registers, as known as the SMMU_SMRn. When a unique match is found, the corresponding SMMU_S2CRn holds the context for the stream. SMMUv1 provides up to 128 Stream Match registers. An SMMUv2 implementation can provide up to 128 Stream Match Registers or can implement the optional Extended Stream Matching Extension, that provides up to 1024 Stream Match Registers and the associated Stream-to-Context registers.

  Note: The implementation/architecture of our Zynq UltraScale+ MPSoC provides 48 Stream Match registers and Stream-to-Context registers.

- **Stream indexing**
  When Stream indexing is used, the StreamID is a direct index to the required SMMU_S2CRn (register). That means, if the StreamID is m, the required Stream-to-Context register is the SMMU_S2CRm. The maximum StreamID is determined by the number of implemented SMMU_S2CRn registers.

- **Compressed StreamID indexing** (available only in an SMMUv2 implementation that includes the optional StreamID Compressed Indexing extension.)
  When Compressed StreamID indexing is used, the StreamID is an indirect index to the required SMMU_S2CRn, as follows:

  1. The StreamID, m, indexes a single-byte S2CRIndex$i$ field in the array of SMMU_COMPINDEXn registers provided by the StreamID Compressed Indexing extension.
  2. The S2CRIndex$i$ field holds the value of the SMMU_S2CRn for the stream. So, if the value of the S2CRIndex$i$ field is x, the required Stream-to-Context register is SMMU_S2CRx.

  Only a single Security state can use Compressed StreamID indexing. In an SMMUv2 implementation that supports two Security states, if one Security state is using Compressed StreamID indexing then the other Security state must use Stream matching.

Note: In our case, we used the Stream Matching mechanism – it is considered the default mechanism for the remaining of this thesis.

The **Stream mapping table** (Figure 3.1) maps a transaction stream to a context and consists of a number of entries, where each entry is a Stream mapping register group containing the following registers, with n defining the Stream mapping register group number. Each entry has:

1. a SMMU_SMRn, which determines whether a transaction matches the group. During configuration, the Stream Match Register table can have multiple entries that match the same Stream Identifier value, possibly resulting in "unpredictable" behavior. To prevent multiple matches, software must ensure that no transactions that might match the StreamID are received, by:

   - Disabling any source client devices that might match
   - Ensuring that no outstanding transactions from these client devices are in progress

   As an extra precaution, software can first disable all affected SMMU_SMRn table entries by setting the SMMU_SMRn.VALID bit to 0, then reprogramming the entries as appropriate.

   Format of the SMMU_SMRn register:
   When the SMMU_sCR0.EXIDENABLE (Extended Stream Matching Extension) is:

   - 0 (zero):

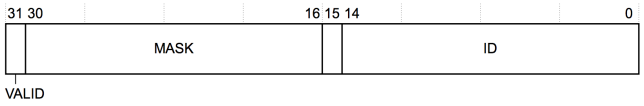

Figure 3.1 – SMRn
Source: ARM [8]

- 1 (one):

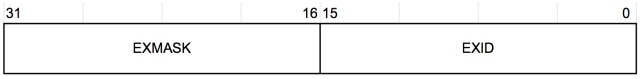

Figure 3.1 – SMRn "extended"
Source: ARM [8]

Some explanations: When MASK[i] == 1, then the ID[i] is ignored. Same with EXMASK and EXID. ID/EXID is actually the StreamID to be matched. When VALID==1, the entry is included in the Stream mapping table search.

2. and a SMMU_S2CRn, which specifies the initial context for the translation process.
Format of the SMMU_S2CRn registers:
When the bits[17:16], the "Type" field of these registers is:

- 00, the type means Translation Context:

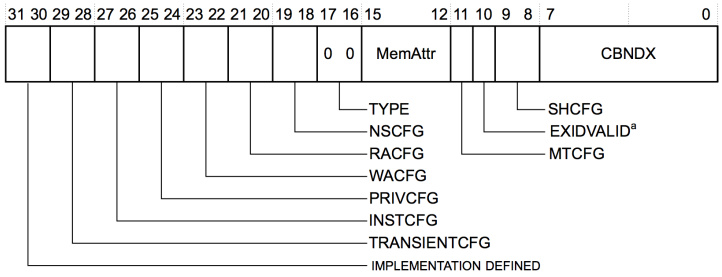

Figure 3.1 – S2CRn Type: 00
Source: ARM [8]

- 01, the type means Bypass Mode:

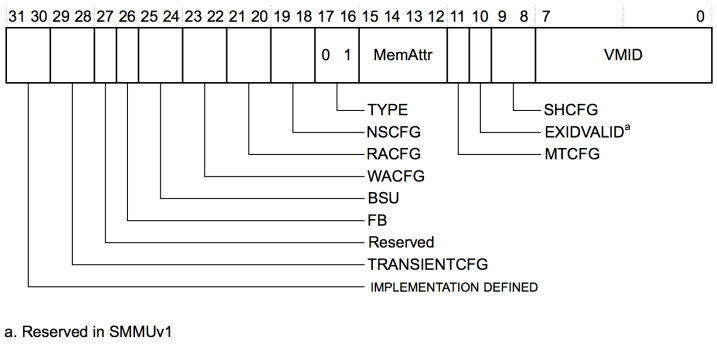

Figure 3.1 – S2CRn Type: 01
Source: ARM [8]

- 10, the type means Fault Context:

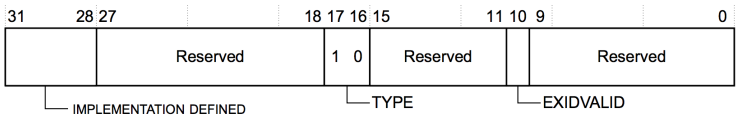

Figure 3.1 – S2CRn Type: 10

Source: ARM [8]

- 11 is Reserved and treated as a fault context

At this point, it is not important to describe the meaning of all fields of the SMMU_S2CRn register, since someone can easily find it on the related manual. For the purposes of this thesis, it is important to mention that when the "Type" field is 00, meaning Translation Context, after a match from the S2CRn register, the CBNDX field of the corresponding SMMU_S2CRn is the index of the Context Bank that is "responsible" for the matched transaction.

When the SMMU_S2CRn register specifies that the initial context for a transaction is a translation context bank, or as we said the "Type" field is 00, then the **SMMU_CBAR**n register specifies the configuration attributes for the translation context bank n.

Format of the SMMU_CBARn registers:
When the bits[17:16], the "Type" field of these registers is:

- 00, the type means Stage 2 context. The context bank provides stage 1 translation, and stage 2 memory attribute-only transformations:

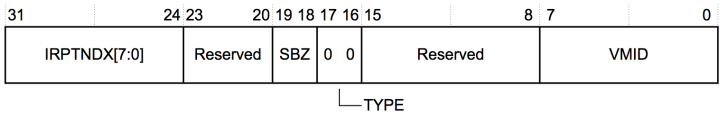

Figure 3.1 – CBARn Type: 00

Source: ARM [8]

- 01, the type means Stage 1 context with stage 2 bypass: In SMMUv2, if the value of the SMMU_IDR2.E2HS bit is 1, then the implementation supports an additional context type, E2HC, that is intended to be used by an OS, that is also acting as a Hypervisor.
  When:
  - no support for E2HC , or SMMU_CR0.HYPMODE == 0:

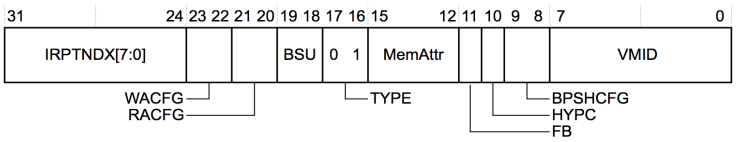

Figure 3.1 – CBARn Type: 01 with no E2HC

Source: ARM [8]

– SMMU_CR0.HYPMODE == 1:

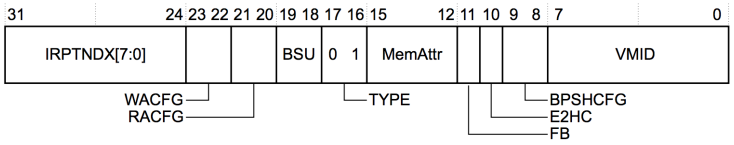

Figure 3.1 – CBARn Type: 01 with E2HC

Source: ARM [8]

- 10, the type means Stage 1 context with stage 2 fault. The context bank generates an invalid context fault:

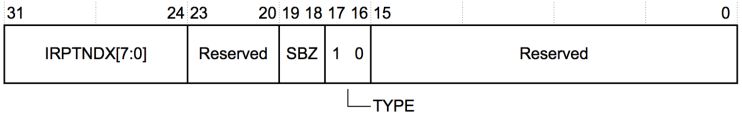

Figure 3.1 – CBARn Type: 10

Source: ARM [8]

- 11, the type means Reserved and Stage 1 followed by stage 2 translation context. The context bank provides stage 1 translation followed by stage 2 translation, and specifies an additional translation context bank for the stage 2 translation.:

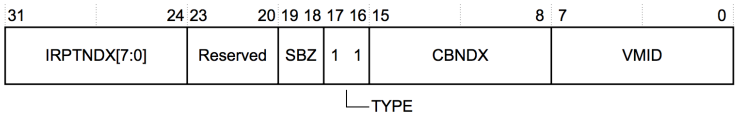

Figure 3.1 – CBARn Type: 11

Source: ARM [8]

From the information of the SMMU_CBARn registers, it is important to mention that each stage of translation (1 and 2) has its own context bank. For instance, if we have a stage 1 translation followed by a stage 2 translation, the "Type" for the initial context bank related to the SMMU_CBARn will be "11" and this register will also give us the index of the context bank for the stage 2 translation (the CBNDX field of the SMMU_CBARn register). The context bank (index) for the stage 1, as we mentioned before, can be found on the S2CRn register shown in Figure 3.1.

SMR0	S2CR0
SMR1	S2CR1
SMR2	S2CR2
...	...
...	...
...	...
SMRn	S2CRn

Figure 3.1 – Stream Mapping Table

Note: In our Zynq UltraScale+ MPSoC we have only 48 groups, which means n lies in the range: 0-47.

3.2.2 Translation Context

The translation context (as known as the context bank) provides information and resources required by the SMMU to process a transaction. SMMU can process multiple transaction streams from different threads of execution and supports multiple live translation contexts.

A translation context bank includes:

- state for configuring the translation process
- capturing fault status, and operations for maintaining cached translations

In implementations that support stage 1 followed by stage 2 translations:

- one translation context bank is specified for single-stage translation.
- two translation context banks are specified for two-stage translation.

A translation context bank is arranged as a table in the SMMU configuration address map. Each entry in the table occupies a 4 KB or 64 KB address space.

In theory, SMMU architecture provides space for up to 128 translation context banks.

Note: In our Zynq UltraScale+ MPSoC, we have only 16 translation context banks.

3.2.3 Page Tables

A page table is the data structure used by a virtual memory system in a computer Operating System to store the mapping between virtual addresses and physical addresses.

Linux

Architectures that manage their Memory Management Unit (MMU) differently are expected to emulate the three-level page tables or four-level page tables. Logically, Linux has four levels of page tables:

- Page Global Directory (PGD)
- Page Upper Directory (PUD)
- Page Middle Directory (PMD)
- Page Table Entry (PTE)

Note: On architectures that do not require all four levels, inner levels may be collapsed.

Each process has a pointer (mm_struct→pgd) to its own Page Global Directory (PGD), which is a physical page frame. This frame contains an array of type pgd_t which is an architecture specific type defined in <asm/page.h>. Each active entry in the PGD table points to a page frame containing an array of Page Upper Directory (PUD) entries of type pud_t. Each active entry in the PUD table points to a page frame containing an array of Page Middle Directory (PMD) entries of type pmd_t, which in turn points to page frames containing Page Table Entries (PTE) of type pte_t, which finally points to page frames containing the actual user data.

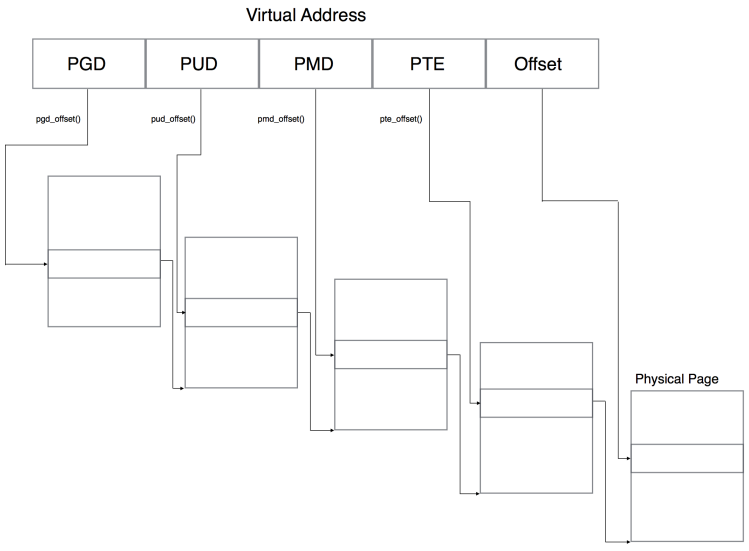

Figure 3.2 – Linux 4-Levels Page Tables

SMMU

Each context bank of the SMMU can be considered as one page table – to be more accurate, each context bank has a field that points to a unique page table for this context.

Each context bank has the **SMMU_CB**n**_TTBR**m, where m can be 0 or 1. TTBR0, as known as the Translation Table Base Register 0 holds the base address of translation table 0. Respectively, the TTBR1 holds the base address of transla-

tion table 0.

It is recommended by ARM that TTBR0 should be used to store the offset to the page tables used by user processes and TTBR1 should be used to store the offset to the page tables used by the kernel. It seems that most Linux implementations for ARM have decided to just basically eliminate the use of TTBR1 and stick to using TTBR0 for everything – this is also happening with the ARM SMMU driver (arm-smmu.c), we worked with.

For each context bank there is also the Translation Control Register, called **SMMU_CBn_TCR**, that determines translation properties, including which one of the Translation Table Base Registers, SMMU_CBn_TTBRm, defines the base address for the translation table walk required when an input address is not found in the TLB. An extension of the SMMU_CBn_TCR exists with the name **SMMU_CBn_TCR2**, that basically extends the SMMU_CBn_TCR by adding control information about the translation granule size and the size of the intermediate physical address.

Chapter 4

Kernel modules for SMMU support

In this chapter, we will describe the second part of the contributions of this thesis, which was the kernel modules we implemented in order to test and use the SMMU of the Zynq UltraScale+ MPSoC. In total we wrote four kernel modules. Below we can see the description of each module:

1. The first module is triggering DMA transactions inside the Processing System using the LPD-DMA or the FPD-DMA.
2. The second module is testing the ARM SMMU from the Processing System (PS) by triggering a transaction from the PS. In Figure 4.1, we can see a simplified path for this.

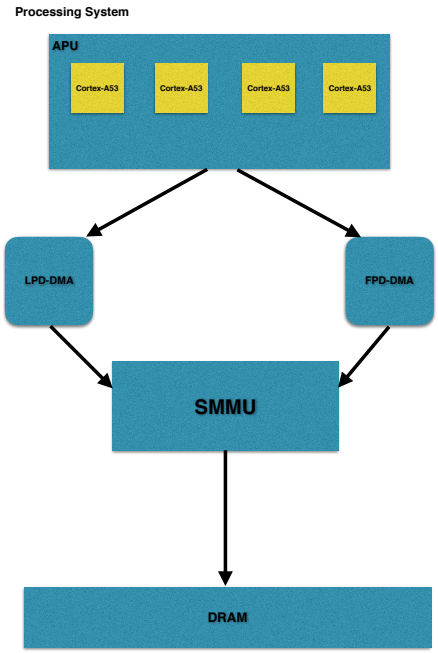

Figure 4.1 – DMAs in the Processing System using the SMMU to access the DRAM

3. The third module is testing the ARM SMMU from the Programmable Logic (PL) by triggering a transaction from the PL, while the SMMU is having a singe-page entry "pre-mapped". In Figure 4.2, we can see a simplified path for this.

4. The fourth module is using the ARM SMMU from the Programmable Logic, by triggering a transaction from the PL, while the SMMU is having all pages of a user process not "pre-mapped" (from the translation table of the process).

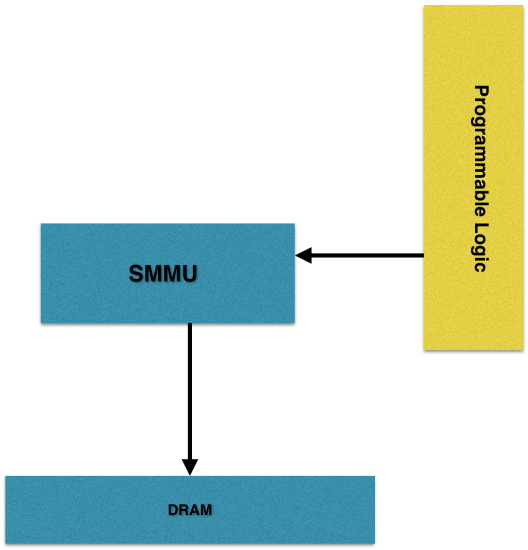

Figure 4.2 – DMAs in the Programmable Logic using the SMMU to access the DRAM

4.1 DMA test in PS

For this module, we actually used a module/driver, that already exists in Linux 4.4.0 with the name: **dmatest.c**. This driver tests all the DMA channels in the Processing System (more information about the DMA in the PS is provided in the Appendix B) by triggering a DMA transaction of, in a way, random bytes from a random source address to a random destination address. It also has an option to run many threads per channel.

Although we liked its usage for testing our DMA from the user-space, we wanted something more accurate, meaning that we could give our own source and destination physical addresses, the length of the data and of course, a specific DMA channel of the Processing System (PS).

At this point, it is good to mention that Linux sees each DMA channel as a unique device, so that's why we could easily modify the module in order to "test" the DMA, or better, do a DMA transaction, using a specific DMA channel.

We faced a problem at first, because by default the LPD-DMA channels were disabled in the device-tree generated by Xilinx tools (Appendix C), but we easily

solved it by following the steps mentioned in the same Appendix. Eventually, we could test 16 different DMA channels (8 for the FPD-DMA and 8 for the LPD-DMA).

The most important function of this module is the **dmatransfer()**, that actually triggers a DMA transaction and has as arguments, the source address, the destination address, the size/length of the data and the name of the DMA channel.

4.2 Transaction from PS with single-page entry

This module was created in order to test if a transaction with a virtual source and/or destination address can be translated to their physical address, successfully, using the SMMU.

In order to achieve this, we had to deeply understand part of the Linux kernel and the IOMMU API. In general, an application program interface (API), is a set of procedures, protocols, and tools for building software applications. IOMMU API was needed in our case because we wanted to access and use the SMMU. We can see the IOMMU API as the more generic driver that actually uses the more specific IOMMU driver and in our case ARM's SMMU driver (the arm-smmu.c).

First of all, before we continue with this module, it would be good to explain the relation between the devices, the groups and the domains, when it comes to the IOMMU API and drivers.

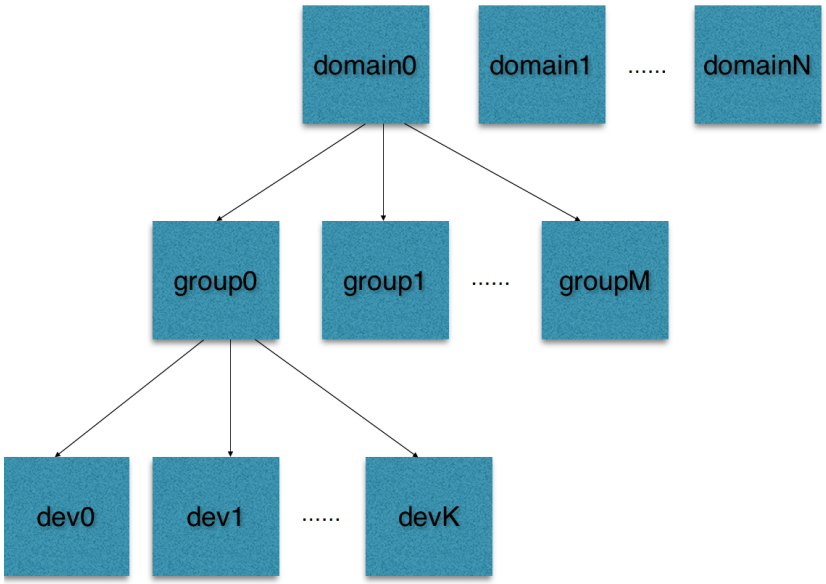

Figure 4.3 – The tree of "hierarchy" of the devices, groups and domains in IOMMU API and drivers.

IOMMU groups are the natural working unit of the IOMMU, but the IOMMU API works on **domains** and **devices**. That gap is bridged by iterating over the devices in a group. Ideally, we would have a single device which represents the requestor ID of the group, but it is also allowed to the IOMMU drivers to create policy-defined minimum sets, where the physical hardware may be able to disti-

nguish members, but wishing to group them at a higher level (ex. untrusted multi-function PCI devices).
In practice (i.e.: when the system was booting), we noticed that each device is added by default to a different group. So, in simple words, if someone wants to use those devices as IOMMU (or SMMU) masters, they should first attach them to one or more domains.

That is exactly what we did with this module. But before starting the implementation of the SMMU module and because we needed to trigger a DMA transaction, we modified the previous module of the Processing System DMA test, in order for other modules to be able to call only the dmatransfer() function and trigger the DMA transaction. So, before inserting the SMMU test kernel module to the system, we make sure we insert first the modified version of the dmatest module – otherwise, our SMMU test module, which is calling the dmatransfer() from the other module, will not work.

Back to the implementation of the SMMU test module, in order to initialize the SMMU and create a context bank, we called (from the IOMMU API):

1. the **iommu_domain_alloc()**, that allocates a new IOMMU domain, using the IOMMU-master device
2. **iommu_group_get()**, that returns the iommu group for our device (which is a DMA channel – one of the 16 DMA channels in the Processing System)
3. **iommu_attach_group()**, that attaches the IOMMU group to the iommu domain

Those functions were enough to create one context bank that we would use for the translation.

The only thing that we had to do then was to add the entry/pair of the virtual and physical address (the physical would be the translation of the virtual) to the translation table of the context bank. So, we called, again from the IOMMU API, the **iommu_map()**, that takes as arguments the domain, that has been allocated (that also has the context bank), the virtual address, the physical address, the size and some flags.

4.3 Transaction from PL with single-page entry

We followed the same procedure when it comes to the IOMMU groups and domains, with the SMMU test module that helped us trigger a DMA transaction from the PS, using the SMMU as we described before, but with one difference: For this module, we do not use the **dmatranfer()** function - this call was for the DMA transactions inside the PS. This time we have a new Intellectual Property (IP) block in the Programmable Logic (PL), that works like a DMA. This IP has these fields: the destination address, the data that they have to be sent to the destination (i.e.: no source address), some bits for the cacheability and the AXI ID of the transaction.

First we initialize the SMMU in the exact way we did before in the Section 4.2 and then we do one mapping (virtual address, physical address) for the destina-

tion (address).

Someone might think that before, we used a device from the PS as an SMMU master in order to make it work - what about now that the IP block is in the PL and also is not an IOMMU master by default? About that, we figured out that the **StreamID** that is given for each device by default has indeed some meaning. In theory, the StreamID of each device in the device-tree has information of the AXI ID and the Master Device, meaning the port that the transaction will go to or is coming from.

So what we noticed was that in the Zynq UltraScale+ MPSoC, if the StreamID is up to 15 or 16 bits (in our case it is up to 15 bits), then the six (6) least significant bits would be about the AXI ID and the next four (4) bits, would be about the Master Device (the port). The rest of the most significant bits were tested and used with the value of zero (0).

Below we see an example of the field of a device, that is a SMMU master, in the source of the device-tree. In this example we see the first channel of the FPD-DMA.

```
1  dma@fd500000 {
2    status = "okay";
3    compatible = "xlnx,zynqmp-dma-1.0";
4    reg = <0x0 0xfd500000 0x1000>;
5    interrupt-parent = <0x1>;
6    interrupts = <0x0 0x7c 0x4>;
7    clock-names = "clk_main", "clk_apb";
8    xlnx,id = <0x0>;
9    xlnx,bus-width = <0x80>;
10   #stream-id-cells = <0x1>;
11   iommus = <0x5 0x14e8>;
12   power-domains = <0x6>;
13   clocks = <0x7 0x3>;
14   linux,phandle = <0x29>;
15   phandle = <0x29>;
16 };
```

In the field "iommus = <0x5 0x14e8>", we see the label (phandle) of the SMMU device, which is 0x5 and the StreamID of our device (first channel of the FPD-DMA) for this SMMU, which is 0x14e8 and is the default StreamID.

Below (in the next page) we see the SMMU field in the source of the device-tree. In the **mmu-masters** field, we see all the devices that are "allowed" to use the SMMU. For example, the entry "0x29 0x14e8" means that the device with the label (or phandle) 0x29, which is the first channel of the FPD-DMA, is an SMMU master and the corresponding and default StreamID for this device is the 0x14e8, as we already saw before.

```
smmu@fd800000 {
   compatible = "arm,mmu-500";
   reg = <0x0 0xfd800000 0x20000>;
   #iommu-cells = <0x1>;
   #global-interrupts = <0x1>;
   interrupt-parent = <0x1>;
   interrupts = <0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4
   0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4
   0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4
   0x0 0x9b 0x4 0x0 0x9b 0x4 0x0 0x9b 0x4>;
   mmu-masters = < 0x1a 0x874
                   0x1b 0x875
                   0x1c 0x876
                   0x1d 0x877
                   0x1e 0x860
                   0x1f 0x861
                   0x20 0x873

                   /*LPD-DMA */
                   0x21 0x868
                   0x22 0x869
                   0x23 0x86a
                   0x24 0x86b
                   0x25 0x86c
                   0x26 0x86d
                   0x27 0x86e
                   0x28 0x86f

                   /* FPD-DMA */
                   0x29 0x14e8
                   0x2a 0x14e9
                   0x2b 0x14ea
                   0x2c 0x14eb
                   0x2d 0x14ec
                   0x2e 0x14ed
                   0x2f 0x14ee
                   0x30 0x14ef

                   0x31 0x870
                   0x32 0x871
                   0x33 0x872>;
   linux,phandle = <0x5>;
   phandle = <0x5>;
};
```

4.4 Transaction from PL using process page table

One of the biggest goals during the thesis was to achieve the SMMU having the translation table of a process, that belongs to a user, in a context bank.

Once again, in the environment we have, we are using the same IP block in the PL (that was similar to a DMA). It is important here to mention that, again, we edit the device-tree accordingly, so the IP block can be treated as an IOMMU-master too (because as we mentioned before, by default it is not). We will try to trigger a transaction from the PL, but this time the destination address will not be a "random" virtual address that corresponds to a physical address, but the virtual address of the buffer that a user has allocated. So, we need one context bank or in other words a translation table of the SMMU to be able to translate the virtual address of the buffer.

In order to achieve this, we used a more clear way than the utility programs, that were somehow opening the /dev/mem and we used in previous modules - we used the ioctl (Input Output ConTroL). In Unix, every device can have its own ioctl commands, which can be: read ioctls (to send information from a process to the kernel), write ioctls (to return information to a process), both or neither. Basically we used ioctl to write to and read from the drivers/modules of the system.

The basic function of this module is the **exanest_virt_ioctl()**, that at first takes the device information we need in order to initialize the SMMU from a "misc device", which in general is a "small" device driver that supports custom "hacks", calling the to_exanest_virt_local() function and then uses a switch() with two (2) cases:

1. **DMA_OPEN_IOCTL**, that basically does the same things we did in the previous SMMU modules in order to initialize the SMMU and create a context bank
   (a) **iommu_domain_alloc()**, that allocates a new IOMMU domain
   (b) calls the **virt_to_phys()**, that takes as argument the pointer of the current process page table, the pgd (3.2.3) and by doing a traverse in all 4 levels of the page-table, it finds its physical translation and returns it
   (c) **iommu_group_get()**, that returns the IOMMU group for our device (which is the IP, similar to a DMA, in the PL)
   (d) **iommu_attach_group()**, that attaches the IOMMU group to the IOMMU domain
2. **DMA_START_IOCTL**, that basically:
   (a) takes a structure, the **dmaCMD**, from the user that contains all the information that the IP in the PL needs, in order to trigger the transaction:
      - the virtual address of the destination (the buffer that the user allocated)
      - the data that we will "send"/write to the destination
      - the cacheability bits (which are fixed but still the user send them)
      - the AXI ID of the transaction

(b) flushes the 4-level translation table for the buffer

(c) writes all the information that came from the user to the addresses of the PL and triggers the transaction

The program that the user runs in order to use the module we described before does the following:

1. allocates a buffer, that we will use as a (virtual) destination address for the transaction
2. initializes this buffer, so we know its value before the transaction
3. opens the exanest_virt module that we described before and that when we insert it in the kernel, it appears in this path: /dev/exanest_virt
4. with ioctl() function, it starts the **DMA_OPEN_IOCTL**, that we described before
5. then, initializes the **dmaCMD** structure, with the values the user wants for the IP in the PL, that will trigger the transaction – as it was mentioned before
6. finally, it reads the buffer to see if the IP in the PL did the transaction successfully - the expected result would be the data he wrote in the **dmaCMD** structure and not the values that the buffer had after the initialization, at the beginning

At first, we noticed the following:

If the **StreamID** in the device-tree was not written like we described in the Section 4.3, then we had no fault and the transaction was not happening (at the end of the user program the buffer had the same values when it was initialized). If the **StreamID** in the device-tree was written like we described in the Section 4.3, then by default we had two faults: a global fault and a context fault.

In effect, the value of the **SMMU_S2CR**n**.INSTCFG** register was 0b11, meaning it was "expecting" an instruction and not data. So, in order to fix that, we overwrote this register and said to the arm-smmu driver, that when we pass our pgd (page table directory), it should also overwrite this value, so the SMMU can treat it as data.

Also, the the **SMMU_CB**n**_TCR** (Translation Control Register) of the SMMU and especially the six (6) least significant bits, the field T0SZ, where the $2^{64-T0SZ}$ gives the address space, had the value 0b01_0000, meaning the input address space (ias) would expect 64-16 = 48 bits (for the incoming virtual address), while the corresponding register in the **MMU** of the processor had the value 0b01_1001, meaning the corresponding address would be 64-24 = 40 bits (incoming virtual address). So, we changed the **T0SZ** field of the SMMU to expect 40 bits too.

We noticed, also, a similar effect of the SMMU and MMU, respectively, TCR register in the **SMMU_CB**n**_TCR2** register in which, the three (3) least significant bits (the field PASize - Physical Address Size), had the value 0xb101, meaning the expected physical address would have to be 48 bits (the out-coming physical address), while in the MMU of the processor, the corresponding field had

the value 0b010, meaning the expected physical address would have to be 40 bits (the out-coming physical address). So, again, we changed the field of the SMMU to expect 40 bits too.

The major changes that we did in order to make this task work, were the changes that we mentioned above and mostly in the **arm-smmu.c** driver. A more detailed description of those changes can be found in the Appendix D.

Chapter 5

Kernel Experiments

In this chapter, we will describe the third part of the contributions of this thesis, which is the experiments we did for all the kernel modules and the notes we used, in order to achieve our goals.

5.1 DMA test in PS

In order to test the DMA module, we were following these steps:

1. Insert kernel module with the **insmod** command

```
# mydmatest.ko is the kernel object of the module
# mydmatest.c
insmod mydmatest.ko
```

2. Use the utility programs (utils) of read/write from/to physical addresses, that existed in the Laboratory, in order to write to a physical address which would be the source of the DMA transaction and then we were reading from an address, that we would use for the destination address, just to make sure it has different data than the source address. It is important to mention that opening part of the /dev/mem in order to read/write from/to physical addresses is far from an acceptable way to access the memory, but at this point we were just testing all the DMA channels in the Processing System (PS). The read/write utils work like this:

```
# ./read_physical32 <address in hex>
./read_physical32 0x60000000
# ./write_physical32 <address in hex> <data in hex>
./write_physical32 0x60000000 0xcafebeef
```

3. Set the proper parameters of the modules, like the source and the destination (physical) addresses and the name of the DMA channel we wanted to test. In order to set the value of the parameters we were calling these commands from a terminal:

```
echo 0x60000000 > /sys/module/mydmatest/parameters/src
echo 0x60002000 > /sys/module/mydmatest/parameters/dst
echo dma1chan0 > /sys/module/mydmatest/parameters/channel
```

4. Set "1" to the parameter with the name "run", in order to trigger the transaction

```
1    echo 1 > /sys/module/mydmatest/parameters/run
```

5. Read the data of the destination address to make sure they are the same with the source address

```
1    ./read_physical32 0x60002000
```

In our tests we used to test the physical address 0x60000000 as the source and the physical address 0x60002000 as the destination, but we also tested many other addresses and the DMA transactions were always successful.

Below we can see the names of the DMA channels in the Processing System as they were detected in Linux.

FPD-DMA

Channel 0 : dma1chan0
Channel 1 : dma2chan0
Channel 2 : dma3chan0
Channel 3 : dma4chan0
Channel 4 : dma5chan0
Channel 5 : dma6chan0
Channel 6 : dma7chan0
Channel 7 : dma8chan0

LPD-DMA

Channel 0 : dma9chan0
Channel 1 : dma10chan0
Channel 2 : dma11chan0
Channel 3 : dma12chan0
Channel 4 : dma13chan0
Channel 5 : dma14chan0
Channel 6 : dma15chan0
Channel 7 : dma16chan0

5.2 Transaction from PS with single-page entry

In order to test the SMMU from the Processing System, after the initialization of the SMMU and the creation/allocation of the context bank, we added two (2) entries in the context bank (meaning two mappings) because in the DMA transaction we wanted to use for both source and destination, a virtual address. So for the source the physical address was: 0x60000000 and the virtual address was: 0x70000000. For the destination the physical address was: 0x60002000 and the virtual address was: 0x70002000.

For testing purposes we used the same read/write utility programs, that we mentioned before. Using those programs, we could write something at the physical address of the source (the 0x60000000 address) and we could expect after running the module (that at the end of it was also triggering a DMA transaction), that we would see the same data at the physical address of the destination (0x60002000).

The following lines are the lines we wrote to a terminal in order to run and test this SMMU module.

```
 insmod mydmatest.ko
./read_physical32 0x60000000    # to see out of curiosity
./write_physical32 0x60000000 0xcafebeef
./read_physical32 0x60000000    # to be sure our data were
                                # written
./read_physical32 0x60002000    # to make sure the destination
                                # does not have the same data
                                # with the source
insmod test_iommu.ko            # the kernel object of our
                                # module
./read_physical32 0x60002000    # checking if it has same data
                                # with the source (0x60000000)
```

At the last run of the read_physical32 program, we saw as the result the same data with the source, as we expected, meaning the translation (using the SMMU) and the DMA transaction worked.

With the tests we did, we also noticed that if we give other addresses that were not mapped as a physical and virtual address entry in the translation table (context bank), the DMA transaction would normally happen with no translation at all (as if all addresses that were given were physical).

5.3 Transaction from PL with single-page entry

The experiments we did for this module, as expected from the description above (Section 4.3), at first included the initialization of the SMMU and a context bank. For this, we followed the same steps that were described in Section 4.2. Then, we used again the utility programs of the Laboratory in order to open part of the /dev/mem and write to physical addresses of the fields of the IP, the destination address, the data to be sent to the destination, the cacheability bits and the AXI ID of the transaction. When we were ready to trigger the transaction, we had

to write the value of one (1) to another field of the IP block, that was triggering the transaction from the PL.

Since we knew that the IP block in the PL was connected to the PS via the HPC0 port, the corresponding StreamID would be the **0x200** or the 000 00**00 10**00 0000 - in bold we see the four Master Device (port) bits for the HPC0 port - and not the default StreamID.

The change we did to a device that was already (by default) an SMMU master (in our example the first channel of the FPD-DMA) was in its StreamID field, in order to give our own StreamID for HPC0 port and AXI ID 0x0, which is translated to the StreamID "0x200". As we already mentioned in Section 4.3, in the "smmu" field of the source of the device-tree, the first channel of the FPD-DMA has by default this entry: "0x29 0x14e8" – so, what we did, in order to pass our StreamID, is that we changed the "0x14e8" to "0x200". We also changed the StreamID field that the device (the first FPD-DMA channel) had in its description in the source of the device-tree, as we mentioned in the Section 4.3.

The assumption that was proven correct by changing the StreamID of a device, that was already a SMMU master, in the device tree, was that when it comes to the SMMU, all the transactions that will eventually go through it, will be "checked" if they are a "match" with any of the StreamIDs that exist in the Stream Mapping Table, no matter the device they are coming from.

By doing these changes, in a "hacking" way, we managed to add our StreamID to the Stream Mapping Table (Figure 3.1), so each transaction that would come through the HPC0 port with AXI ID 0 (since the six least significant bits were zero), would be a match for the SMMU and in our case, it would also "request a translation". And so it did.

We also tested the case where the IP block in the PL, that works like a DMA, is not connected with the PS via HPC0 port, but via HPC1 port and we had the same results – that is not a surprise, since the HPC1 port is also coherent and almost identical to HPC0 port.

Below there is an example that we tested, where the IP block in the PL that was connected to PS via HPC0 port has as the base physical address: 0xb0000000. These were the addresses of its fields:

- Destination Address (least significant bits) : 0xb000_0030
- Destination address (least significant bits): 0xb000_0034
- Destination data : 0xb000_0038
- Cacheability bits : 0xb000_003c
- AXI ID : 0xb000_0040
- Trigger : 0xb000_0044

In order to initialize the SMMU, we have to insert (and initialize) the SMMU kernel module, that except from the initialization, it also adds one (physical address, virtual address) entry of the destination of our transaction from the PL. In the test we did, we had the 0x60002000 as the physical address and the 0x70002000 as the virtual address.

In our example the StreamID is the one we mentioned above: 0x200, meaning the expected AXI ID is 0x0. The cacheability bits, it is suggested to have as a value the 0xf, so that is what we used in our test.

The following lines are the lines we wrote using a terminal that triggered the transaction from the PL that indeed used the SMMU to translate the virtual address before the transfer.

```
# Insert the SMMU module, that initializes the SMMU
# and adds the 0x60002000 0x70002000 mapping
insmod test_iommu.ko

# Check the original values of the destination address
./read_physical32 0x60002000

# Set the IP block in the PL
./write_physical32 b0000030 0x70002000
./write_physical32 b0000034 0x00000000
./write_physical32 b0000038 0xcafebeef
./write_physical32 b000003c 0xf
./write_physical32 b0000040 0x0
./write_physical32 b0000044 0x1

# To make sure the destination has the same data
# with the data the IP block had
./read_physical32 0x60002000
```

5.4 Transaction from PL using process page table

For the testing of the exanest_virt module, we did not change anything in terms of the parameters – the only changes that we had are the changes mentioned in the description of the Section 4.4 and the Appendix D.

Let's suppose that the source code of the user program described in Section 4.4 is called **run_exanest_virt.c** and we already have an executable of it, with the name **run_exanest_virt**. The following lines are the lines we wrote to a terminal in order to test this module.

```
 insmod exanest_virt.ko
 ./run_exanest_virt
```

As mentioned in the description of the module, it works since it's overwriting the user buffer, but it seems that the data are not coherent.

Chapter 6

Conclusion

All the SMMU modules worked, meaning in all the environments (transactions from the Processing System and from the Programmable Logic), that were described in detail previously in Chapter 4, we managed to have the translation, from a virtual address to a physical address.

There are still many areas of the IOMMU (SMMU) to be discovered, since it is still difficult to extract information from the existing documentations, that is related to the implementation or the use of the IOMMU in Linux. For instance, when it comes to the StreamID, it is still not clear how it works - we had an assumption for the 10 least significant bits, that, as mentioned, we used and tested in our implementation, but the whole picture with the StreamID remains unclear. Also, in our case, we only used one context bank (translation table) and although it might seem easy after the analysis in the previous chapters, an implementation of using many context banks, allocating many domains etc is important and should happen in the future. In the last module with the page table of a process, although we managed to make it work, as we mentioned, we noticed lack of coherence, so in order to make it work properly, we flushed the cache lines (only four cache lines, for each one of the levels of the translation table) and changed the snooping option in the FSBL - this change, although it looked optimistic at the beginning, did not fix the coherence problem completely, meaning that sometimes we still had non-coherent data. Another approach was to allocate another much bigger buffer, that we used before triggering the transaction from the PL and by doing that, we ensured that our first buffer had the last/updated data. In the future it should be investigated if there are any options that solve the coherence problem in a more clear way (for instance: by making the process page table write-through, using the proper cacheability flags in the IOMMU settings etc).

To sum up, the greater goal triggering transactions that use the global virtual address space in a remote level did not happen yet, since our implementations were not tested for an environment like the environment the Unimem describes, where many remote coherent islands (nodes) are trying to communicate. As we mentioned at the beginning, this thesis was aiming to help and support some of the big goals of the Unimem (Section 1.1), by providing information and tested ideas, which we hope that we achieved. By expanding the ideas, approaches and perhaps the modules that were described in this thesis, it should be easier for someone to reach Unimem's greater goal.

Bibliography

[1] M. Katevenis, N. Chrysos, M. Marazakis, I. Mavroidis, F. Chaix, N. Kallimanis, J. Navaridas, J. Goodacre, et al. The exanest project: Interconnects, storage, and packaging for exascale systems. In *Proceedings of the Euromicro Conference on Digital System Design (DSD 2016)*, 2016.

[2] Xilinx. *Zynq UltraScale+ MPSoC Technical Reference Manual UG1085*. Xilinx, 1.3 edition, October 2016.

[3] Trenz Electronic. *TE0808 TRM*. Trenz Electronic, October 2016.

[4] Xilinx. *AXI Reference Guide*. Xilinx, 13.11 edition, March 2011.

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. 3 edition, February 2005.

[6] Intel. *Intel Virtualization Technology for Directed I/O*. Intel, June 2016.

[7] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*. AMD, 3.00 edition, December 2016.

[8] ARM. *System Memory Management Unit (SMMU) Architecture version 2.0 Specification*. ARM, June 2016.

[9] Xilinx. *UltraScale Architecture and Product Overview DS890*. Xilinx, 2.10 edition, November 2016.

[10] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. 2.6 edition, May 2007.

[11] Xilinx. Zynq ultrascale+ mpsoc register reference, 2016. UG1087, version 1.2.

[12] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. July 2007.

Appendix A

Appendix

A.1 Booting from the SD Card

In this Appendix, we will write all steps we followed in order to boot Linux using an SD card. Before that, we should mention that in the Laboratory we used a specific format of the SD Card - the "u-boot" file "knew" from what partitions it could find all files that were needed. That means that files should have specific names (e.g.: the "Image" file should have the name "Image" and not "Image2"). The first partition should have all the needed files except the "rootfs", that belongs to the second partition.
In the first partition, in order for our system to boot, we needed those files:

1. BOOT.bin
2. Image (kernel)
3. system.dtb (device-tree)

A.1.1 BOOT.bin

There are specific steps someone should follow to create the image of the BOOT.bin file. Below we write all the files that are needed for BOOT.bin file.

1. bitstream (.bit file)
2. FSBL (fsbl.elf)
3. u-boot (u-boot.elf)
4. bl31.elf

The bitstream is generated from a Xilinx Vivado project. The "u-boot" file, as we mentioned before, is already created so it is like something fixed in this Appendix. Also the bl31.elf is fixed for the purposes of this Appendix - in general, it can be generated easily with the help of Petalinux.

The **FSBL** file is generated by the .hdf file, that can also be exported from a Xilinx Vivado project. Below (next page) we explain how.

Step 1: We open Xilinx SDK
Step 2: We give a path and a name for our workspace (for instance here it is: workspace_Dec19)

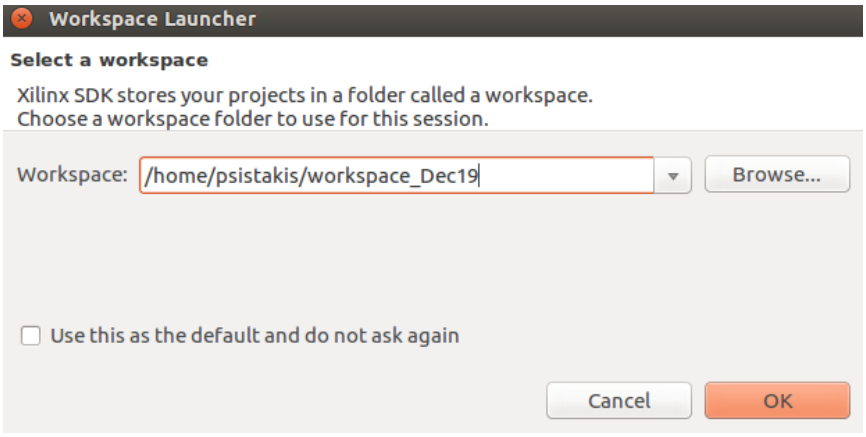

Figure A.1 – Step 2

Step 3: File → New → Application Project

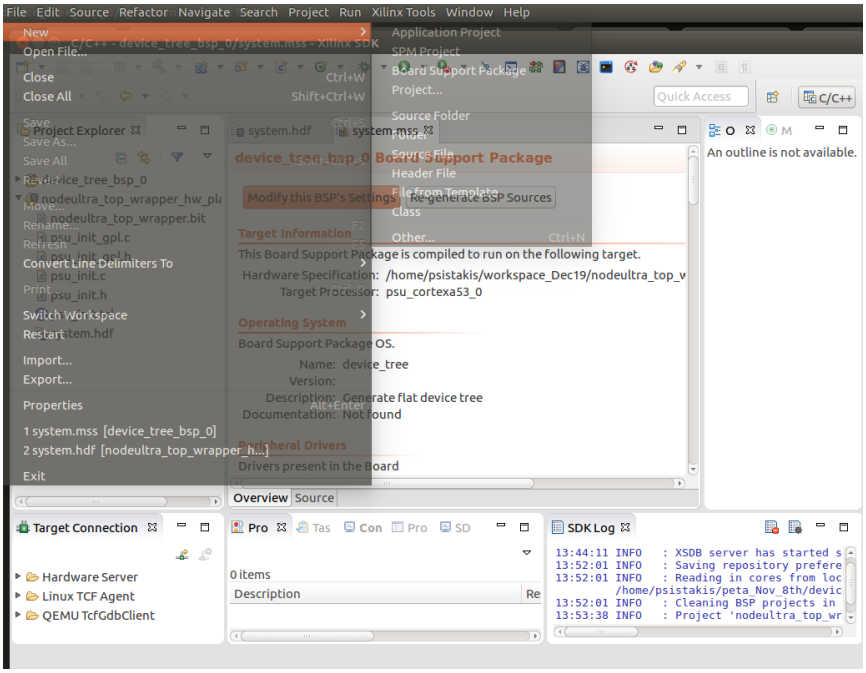

Figure A.2 – Step 3

Step 4: Hardware Platform → New

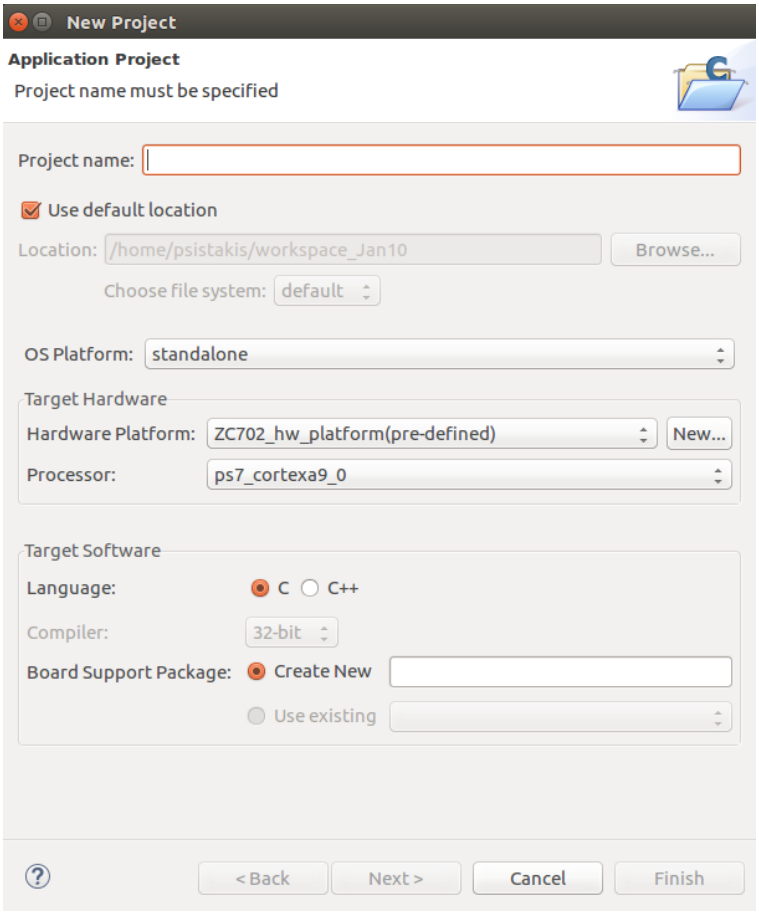

Figure A.3 – Step 4

Step 5: Specify and browse the path of the .hdf file → The project name will appear.

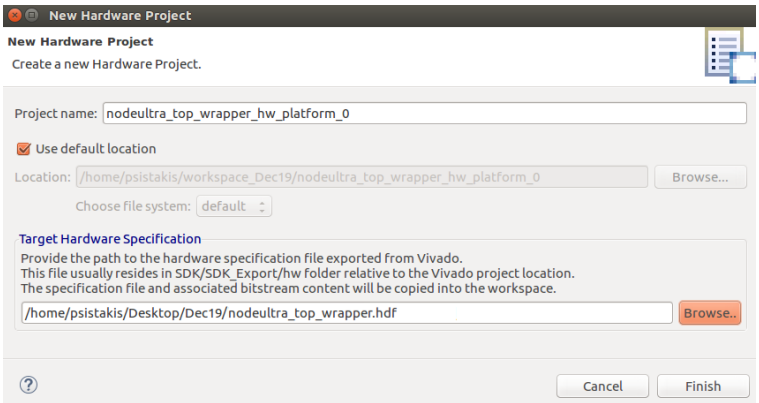

Figure A.4 – Step 5

Step 6: In "Project name" field write: fsbl

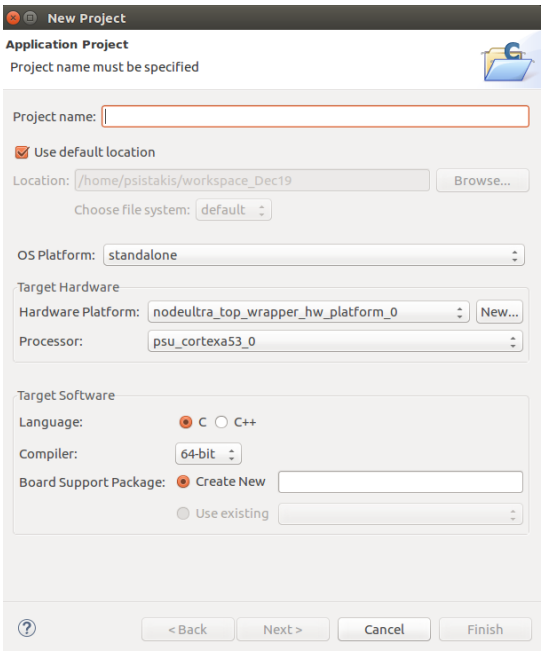

Figure A.5 – Step 6

Step 7: Click "Next"

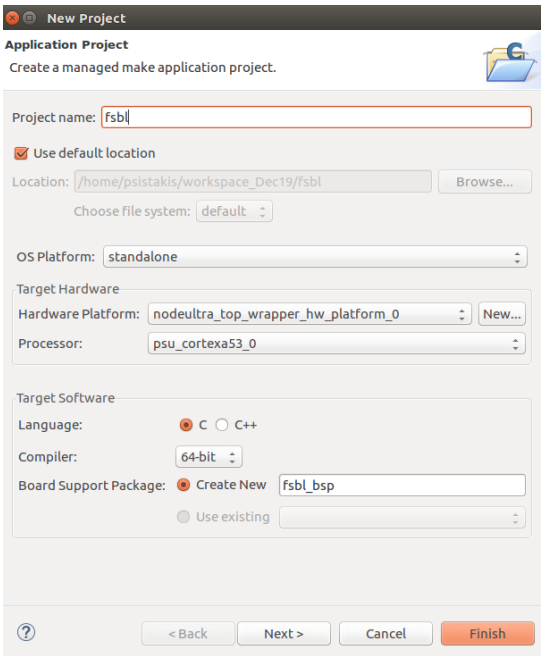

Figure A.6 – Step 7

Step 8: Choose "Zynq MP FSBL" -> Click "Finish".

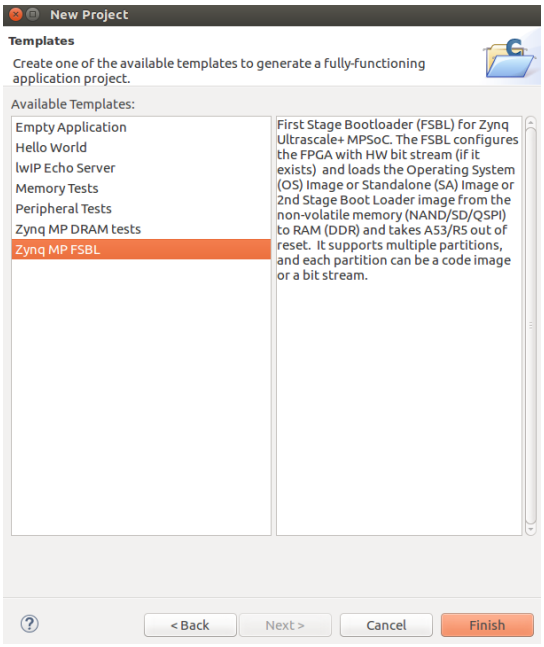

Figure A.7 – Step 8

Step 9: We go at the workspace folder and see some folders/files like in the image below. The "fsbl" is the folder we care about.

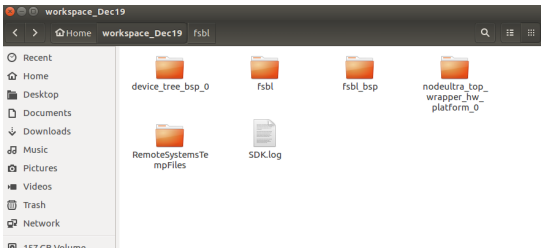

Figure A.8 – Step 9

Step 10: Inside the "Debug" folder there is the fsbl.elf that was generated.

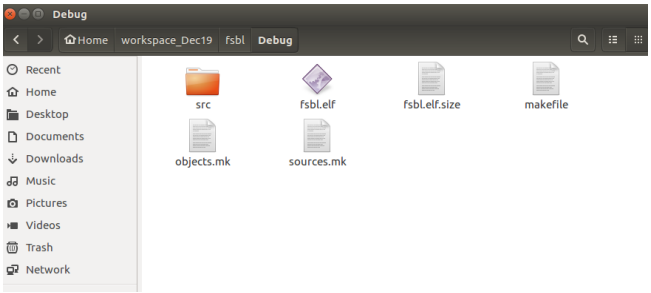

Figure A.9 – Step 10

Except from all the needed files that we mentioned for the BOOT.bin file, we will also need a .bif that mentions them.
An example of the sd_boot.bif we used:

```
//arch = zynqmp; split = false; format = BIN
the_ROM_image:
{
  [fsbl_config]a53_x64
  [bootloader]fsbl.elf
  [destination_cpu = a53-0]bl31.elf
  [destination_cpu = a53-0]u-boot.elf
  [offset = 0x100000, destination_device = pl]nodeultra_top_wrapper.bit
}
```

Now we have the .bif, here is the command we run to a terminal, in order to create the BOOT.bin file:

```
bootgen -arch zynqmp -image sd_boot.bif -o BOOT.bin -log info
```

A.1.2 Image

Having our own configuration file (.config) in the root file of our Linux sources, we use a cross compiler in order to build the kernel in the same environment (Linux version) we want our boards to have. To do that, we run the following commands:

```
export ARCH=arm64
export CROSS_COMPILE=<path of the cross compiler sources>/bin/aarch64-linux-gnu-
```

By running the Makefile (with the command "make") in the root file of our Linux sources (kernel), we build/create the "Image" file that will be located in this path: <Linux 4.4.0 sources>/arch/arm64/boot/Image.

A.1.3 system.dtb

Appendix (C) is dedicated to the generation of the Device-Tree.

Appendix B

Appendix

B.1 Xilinx Zynq UltraScale+ DMA in the PS

Although in [11], the DMA in the Programming System (PS) is mentioned as one unit, the general purpose DMA (ZDMA), we actually see it as two DMAs: one is located in the LPD, the low-power domain DMA (LPD-DMA) and another is located in the FPD, the full-power domain DMA (FPD-DMA).
The basic differences between those two are:

1. The LPD-DMA can be optionally configured as I/O coherent, like all of the PS masters, including the RPU but excluding the full-power DMA controller (FPD-DMA).

   The FPD-DMA does not have a physical path through the CCI and does not support I/O coherency. The LPD-DMA does have an alternate path to the DDR controller through the CCI, which allows it to be marked as I/O coherent.

2. - FPD-DMA is an 8-channel DMA with a 128-bit AXI bus width.
   - LPD-DMA is an 8-channel DMA with a 64-bit AXI bus width.

The addresses of the channels of the **FPD-DMA** :

0xFD500000 (channel 0)
0xFD510000 (channel 1)
0xFD520000 (channel 2)
0xFD530000 (channel 3)
0xFD540000 (channel 4)
0xFD550000 (channel 5)
0xFD560000 (channel 6)
0xFD570000 (channel 7)

The addresses of the channels of the **LPD-DMA**:

0xFFA80000 (channel 0)
0xFFA90000 (channel 1)
0xFFAA0000 (channel 2)
0xFFAB0000 (channel 3)

0xFFAC0000 (channel 4)
0xFFAD0000 (channel 5)
0xFFAE0000 (channel 6)
0xFFAF0000 (channel 7)

Appendix C

Appendix

C.1 The Device-Tree

C.1.1 Generating the Device-Tree

In general, the **device tree** is a data structure for describing hardware, which originated from Open Firmware. The data structure can hold any kind of data as internally it is a tree of named nodes and properties. Nodes contain properties and child nodes, while properties are name–value pairs.

In order to generate the device-tree we want to use for an Operating System (Linux), we only need the .hdf (hardware file) that someone can easily export from Xilinx Vivado, with a specific design for a specific board.

Step 1: We open Xilinx SDK

Step 2: We give a path and a name for our workspace (for instance here it is: workspace_Dec19)

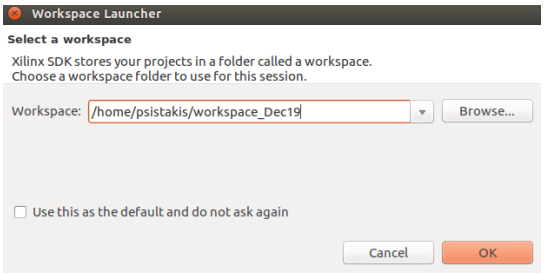

Figure C.1 – Step 2

Step 3: Xilinx Tools → Repositories

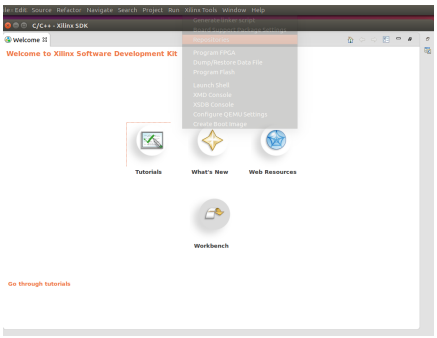

Figure C.2 – Step 3

Step 4: Click "New" (in Local Repositories)

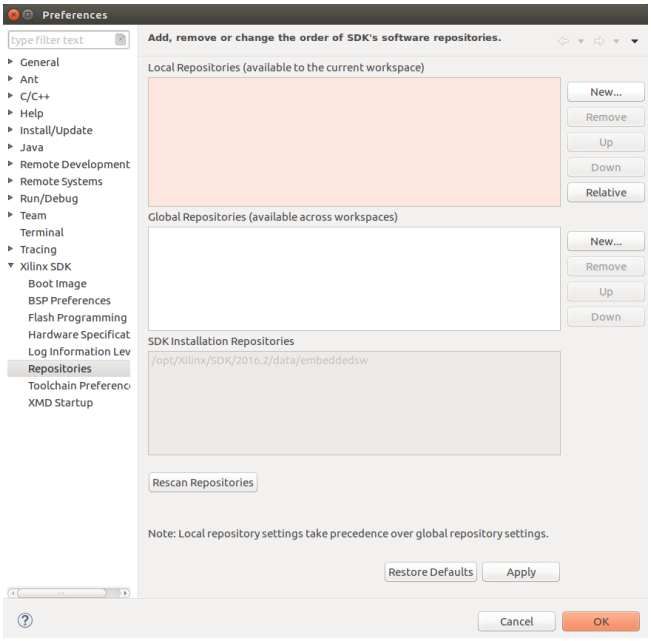

Figure C.3 – Step 4

Step 5: Add specific repository (the device tree source files), provided, in our case, by the Xilinx Git. For instance, we were working with Xilinx Vivado and SDK 2016.2 version, so we also downloaded the 2016.2 version of the device tree sources.

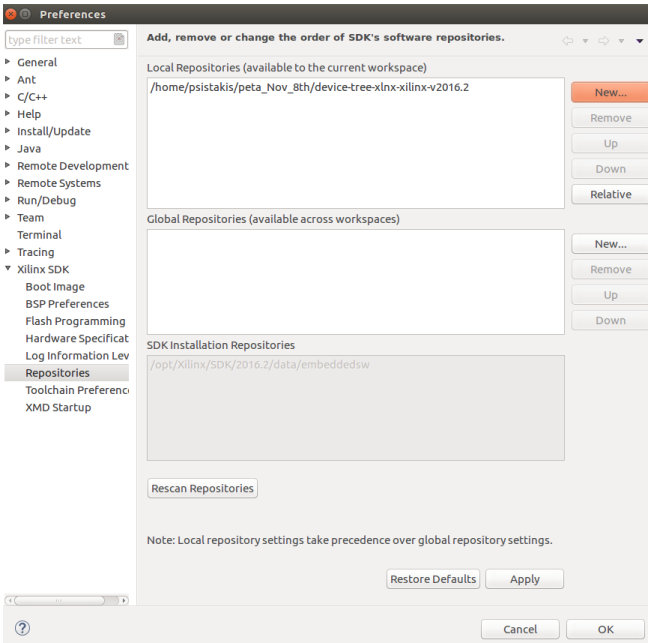

Figure C.4 – Step 5

Step 6: File → New → Board Support Package

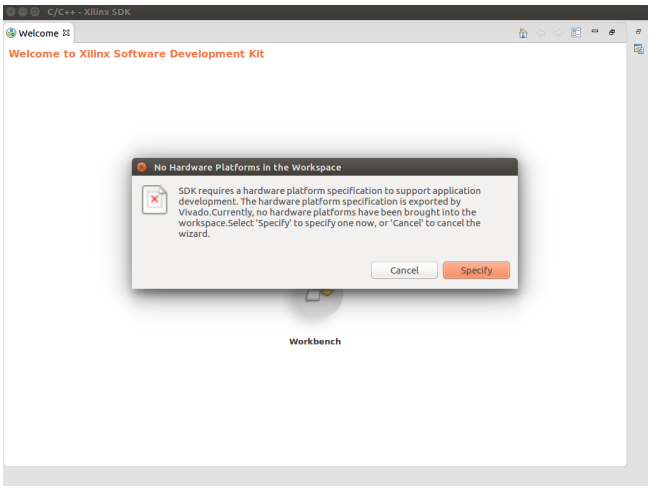

Figure C.5 – Step 6

Step 7: Specify and browse the path of the .hdf file → The project name will appear.

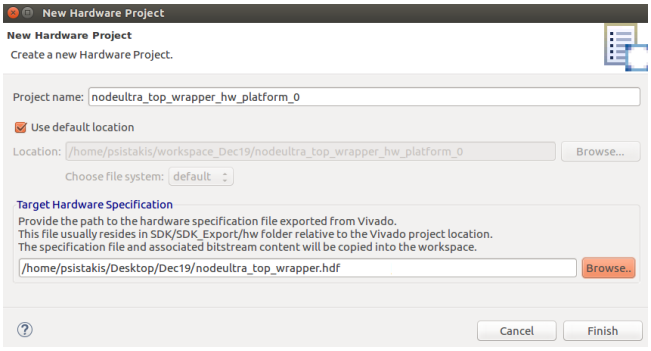

Figure C.6 – Step 7

Step 8: Choose the "device_tree" → Click "Finish".

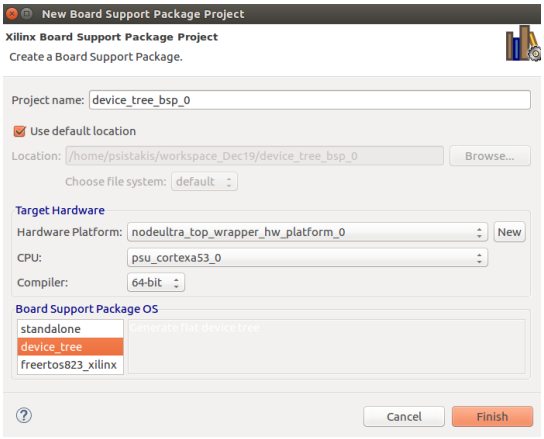

Figure C.7 – Step 8

Step 9: At this step we do not need to change anything.

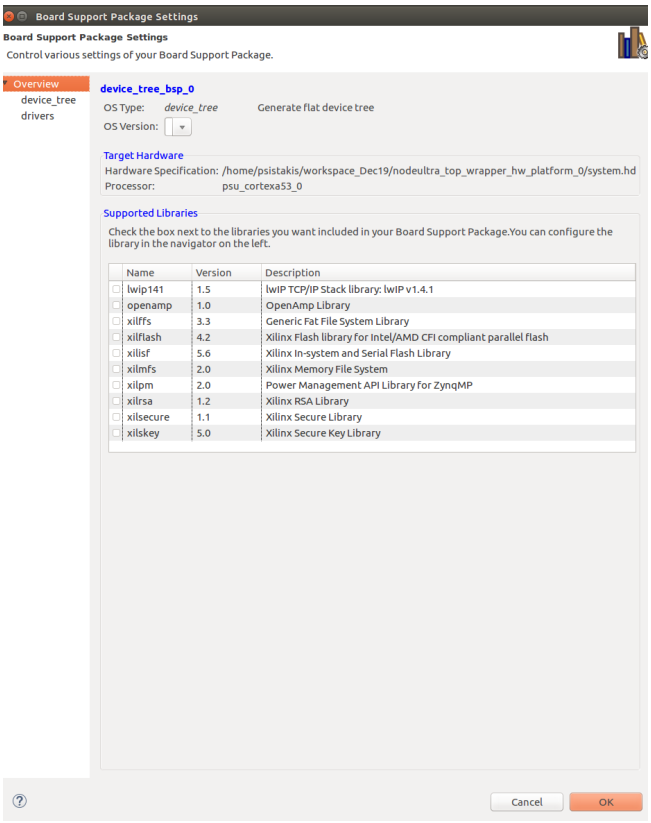

Figure C.8 – Step 9

Step 10: We go at the workspace folder and see some folders/files like in the image below. The "device_tree_bsp_0" is the folder we care about.

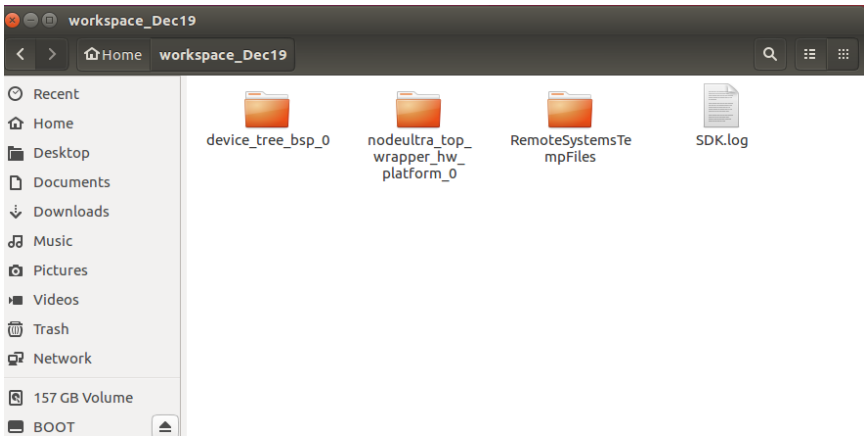

Figure C.9 – Step 10

Step 11: Inside the "device_tree_bsp_0" we see the files like in the image below.

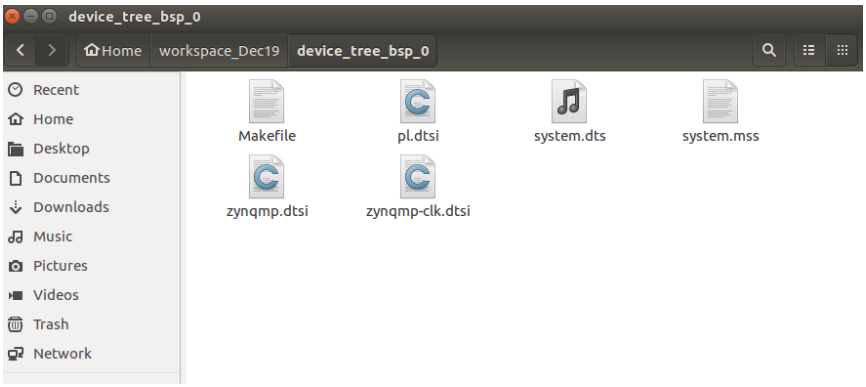

Figure C.10 – Step 11

Step 12: We open a terminal and write this command: **dtc -I dts -O dtb -o system.dtb ./system.dts**. This command will create the Device-Tree Blob (.dtb) the kernel needs. The "system.dts" file "includes" all the other files, so now with the .dtb we created, we have them all in one file.

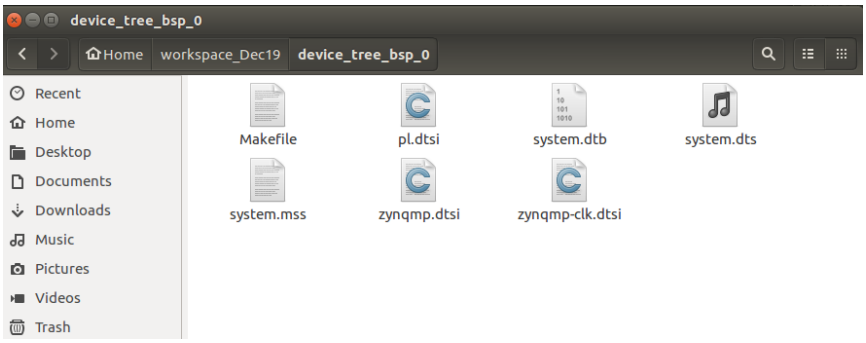

Figure C.11 – Step 12

Step 13: If we want to change the source of the device-tree, we open a terminal and run this command: **dtc -I dtb -O dts -o system_NEW.dts ./system.dtb**. Now in the "system_NEW.dts" file we have all the sources of the device-tree (the files we saw in Figure C.11) in one file.

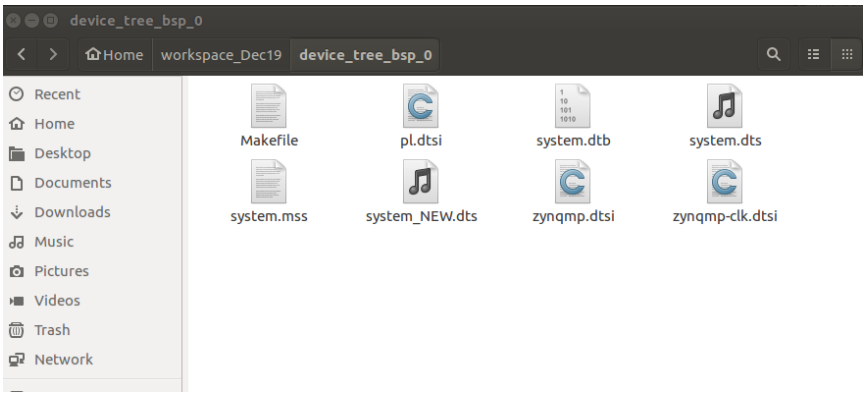

Figure C.12 – Step 13

C.1.2 Enabling the LPD-DMA

By default, all the LPD-DMA channels are disabled in the generated device-tree. In order to fix that, in the source of the device-tree (in our case: the "system_NEW.dts" file) we add the "**clock-names = "clk_main", "clk_apb";**" and the "**clocks = <0x7 0x3>;**" fields, in all LPD-DMA channels, like the example below.

```
dma@ffa80000 {
  status = "okay";
  compatible = "xlnx,zynqmp-dma-1.0";
  reg = <0x0 0xffa80000 0x1000>;
  interrupt-parent = <0x1>;
  interrupts = <0x0 0x4d 0x4>;
  clock-names = "clk_main", "clk_apb";
  xlnx,id = <0x0>;
  xlnx,bus-width = <0x40>;
  #stream-id-cells = <0x1>;
  iommus = <0x5 0x868>;
  power-domains = <0x8>;
  clocks = <0x7 0x3>;
  linux,phandle = <0x21>;
  phandle = <0x21>;
};
```

C.1.3 SD card

If we want to boot from the SD card, we also need to do the following changes in the source of the device tree.

1. Change the bootargs field, in order to be able to detect the root and some other things that apparently are needed. Instead of this: **bootargs = "console=ttyPS0,115200";**, write this: **bootargs = "console=ttyPS0,115200 root=/dev/mmcblk0p2 rw rootwait earlyprintk bootmem_debug=1 memblock_debug=1";**.

   Below we see the part of the source code of the device-tree, that we changed:

```
chosen {
  bootargs = "console=ttyPS0,115200";
};
aliases {
  ethernet0 = "/amba/ethernet@ff0e0000";
  serial0 = "/amba/serial@ff000000";
  spi0 = "/amba/spi@ff0f0000";
};
memory {
  device_type = "memory";
  reg = <0x0 0x0 0x0 0x80000000>;
};
```

2. If you have many boards, you have to change the MAC address according to the board/box you are currently working on. This is a "mini-hack" in order to avoid many boards having the same MAC address - we achieve that by simply changing the second field of zeros (from the end) to the "cx", where x is the number of the board (for instance for board 7: c7). In the example below, instead of this: **"local-mac-address = [00 0a 35 00 00 00];"**, we write this: **"local-mac-address = [00 0a 35 00 c7 00];"**.

```
ethernet@ff0e0000 {
  compatible = "cdns,zynqmp-gem";
  status = "okay";
  interrupt-parent = <0x1>;
  interrupts = <0x0 0x3f 0x4 0x0 0x3f 0x4>;
  reg = <0x0 0xff0e0000 0x1000>;
  clock-names = "pclk", "hclk", "tx_clk";
  #address-cells = <0x1>;
  #size-cells = <0x0>;
  #stream-id-cells = <0x1>;
  iommus = <0x5 0x877>;
  power-domains = <0xe>;
  clocks = <0xb 0xb 0xb>;
  local-mac-address = [00 0a 35 00 c7 00];
  phy-mode = "rgmii-id";
  xlnx,ptp-enet-clock = <0x0>;
  linux,phandle = <0x1d>;
  phandle = <0x1d>;
};
```

3. Add this: **"no-1-8-v"** at the end of the sd-node that is activated. In our case, it was the "sdhci@ff170000" sd-node.

   Below we see the part of the source code of the device-tree, that we changed:

```
sdhci@ff170000 {
  compatible = "arasan,sdhci-8.9a";
  status = "okay";
  interrupt-parent = <0x1>;
  interrupts = <0x0 0x31 0x4>;
  reg = <0x0 0xff170000 0x1000>;
  clock-names = "clk_xin", "clk_ahb";
  broken-tuning;
  #stream-id-cells = <0x1>;
  iommus = <0x5 0x871>;
  power-domains = <0x19>;
  clocks = <0x18 0x18>;
  clock-frequency = <0xa37c42e>;
  linux,phandle = <0x32>;
  phandle = <0x32>;
  no-1-8-v; /* New field */
};
```

Note: After all the changes in the source code, build again the device tree, by creating the Device-Tree blob (Step 12).

Appendix D

Appendix

D.1 Changes to the Linux kernel

In order to make the fourth module of this thesis work (Section 4.4), we had to make some changes to some Linux drivers, like the arm-smmu.c driver, which is the driver of the ARM SMMU, the iommu.c driver, which is the IOMMU driver that implements the IOMMU API and the iommu.h library. Below we see all dominant changes for each file:

D.1.1 iommu.h

In the iommu.h, we just added an extra field in the iommu_domain structure.

```
struct iommu_domain {
  unsigned type;
  const struct iommu_ops *ops;
  iommu_fault_handler_t handler;
  void *handler_token;
  struct iommu_domain_geometry geometry;
  void *iova_cookie;
  u64 ttbr; /* new field */
};
```

D.1.2 iommu.c

In the iommu.c driver, we only edited the **__iommu_domain_alloc()** function, which was called each time we had an allocation of an IOMMU domain. More specifically, we initialized the field "ttbr", that we added in the iommu.h.

```
static struct iommu_domain *__iommu_domain_alloc(struct bus_type *bus,
unsigned type){
  pr_info("%s\n", __func__);
  struct iommu_domain *domain;
  if (bus == NULL || bus->iommu_ops == NULL)
    return NULL;
  domain = bus->iommu_ops->domain_alloc(type);
  if (!domain)
    return NULL;
  domain->ops  = bus->iommu_ops;
  domain->type = type;
  domain->ttbr = 0; /* new field */
  return domain;
}
```

D.1.3 arm-smmu.c

The idea is that when we allocate an IOMMU domain in our exanest_virt module, we would want to "pass" to the arm-smmu driver the pointer of the page table of a process of the user. So, when we allocate the IOMMU domain, we also "pass" this pointer (as we mentioned before, the pgd, which is not equal to zero) to the new field of the structure (the ttbr). Then, the arm-smmu driver, after the changes we did (we can see them below), will understand that this value of the ttbr is different from the expected (default) value, which is zero, and will take the ttbr and "pass" it to its own context bank. After that, the context bank we initialized will have as a translation table, not the translation table that was created at the initialization of the context bank, but the translation table of the user process.
In the function **arm_smmu_init_context_bank()**:

```
if(smmu_domain->domain.ttbr!=0){ /* new field */
  printk(KERN_ALERT "BYPASSED TTBR0\n");
  reg64 = smmu_domain->domain.ttbr;
  printk("smmu_domain->domain.ttbr %lu \n", smmu_domain->domain.ttbr);
}
else{
  printk(KERN_ALERT "DEFAULT TTBR0\n");
  reg64 = pgtbl_cfg->arm_lpae_s1_cfg.ttbr[0];
}
```

In order to fix the issues about the size of the addresses, that were mentioned in Section 4.4, we did some changes to the same function:

```
/* TTBCR */
if (stage1) {
  reg = pgtbl_cfg->arm_lpae_s1_cfg.tcr;
  if ( smmu_domain->domain.ttbr != 0 ) {
    // Use processor given size from TCR_EL1
    // VIRTUAL ADDRESS SIZE
    // Overides bit calculation based on cfg->ias from
    // arm_64_lpae_alloc_pgtable_s1
    reg = (reg & 0xffffffc0) | 0x19;
  }
  writel_relaxed(reg, cb_base + ARM_SMMU_CB_TTBCR);
  pr_info("TTBCR write 0x%x\n",reg);
  if (smmu->version > ARM_SMMU_V1) {
    reg = pgtbl_cfg->arm_lpae_s1_cfg.tcr >> 32;
    if ( smmu_domain->domain.ttbr != 0 ) {
      // Use processor given size from TCR_EL1
      // PHYSICAL ADDRESS SIZE
      // Overides bit calculation based on cfg->oas from
      // arm_64_lpae_alloc_pgtable_s1
      reg = (reg & 0xffffff8) | 0x2;
    }
    reg |= TTBCR2_SEP_UPSTREAM;
    writel_relaxed(reg, cb_base + ARM_SMMU_CB_TTBCR2);
    pr_info("TTBCR2 write 0x%x\n",reg);
  }
} else {
  reg = pgtbl_cfg->arm_lpae_s2_cfg.vtcr;
  writel_relaxed(reg, cb_base + ARM_SMMU_CB_TTBCR);
}
```