Student Research Project

autumn semester 2017

Student Research Study

Muen on ARM - an Evaluation

version: 1.0, date: December 21, 2017

supervisors:

Prof. Dr. Andreas Steffen

MSc Adrian-Ken Rüegsegger

MSc Reto Bürki

HSR, Rapperswil

無縁

David Loosli, student

BSc in Computer Science

HSR Rapperswil

Change History

date	version	change	author
Oct 15, 2017	0.1	prepared template, setup basic version	David Loosli
Nov 5, 2017	0.1	bibliography; changed structure according to previous findings	David Loosli
Nov 11, 2017	0.2	introduction incl. bibliography and glossary	David Loosli
Nov 12, 2017	0.2	first part of chapter 2 (overview, SPARK requirements)	David Loosli
Nov 14, 2017	0.2	second part of chapter 2 (SPARK requirements, virtualization basics)	David Loosli
Nov 14, 2017	0.2	third part of chapter 2 (memory)	David Loosli
Nov 17, 2017	0.2	third part of chapter 2 (memory)	David Loosli
Nov 18, 2017	0.2	fourth part of chapter 2 (interruptions)	David Loosli
Nov 23, 2017	0.2	fourth part of chapter 2 (interruptions)	David Loosli
Nov 24, 2017	0.2	fourth part of chapter 2 (interruptions)	David Loosli
Nov 25, 2017	0.2	fourth and fifth part of chapter 2 (interruptions, device handling)	David Loosli
Nov 26, 2017	0.3	fifth part (interruptions) and summary / last check of chapter 2	David Loosli
Nov 28, 2017	0.3	structure chapter 3 (incl. introduction)	David Loosli
Dec 2, 2017	0.3	corrections of chapter 2 according to meeting	David Loosli
Dec 3, 2017	0.4	first part of chapter 3 (overview, coding)	David Loosli
Dec 4, 2017	0.4	first part of chapter 3 (coding, startup)	David Loosli
Dec 5, 2017	0.4	second part of chapter 3 (fundamentals)	David Loosli
Dec 7, 2017	0.4	second part of chapter 3 (fundamentals, virtualization basics)	David Loosli
Dec 8, 2017	0.4	third part of chapter 3 (virtualization basics, caching)	David Loosli
Dec 9, 2017	0.4	third part of chapter 3 (caching, memory)	David Loosli
Dec 10, 2017	0.4	third part of chapter 3 (memory)	David Loosli
Dec 12, 2017	0.4	fourth part of chapter 3 (exception handling, timer)	David Loosli

date	version	change	author
Dec 13, 2017	0.4	fifth part of chapter 3 (spark and requirement comparison)	David Loosli
Dec 15, 2017	0.5	corrections up to chapter 2 (incl. rewriting)	David Loosli
Dec 16, 2017	0.5	corrections of chapter 3 (incl. rewriting)	David Loosli
Dec 17, 2017	0.5	first part of chapter 4 (overview, and boot process)	David Loosli
Dec 19, 2017	0.5	second part of chapter 4 (exception and device handling)	David Loosli
Dec 19, 2017	0.6	chapter 5 and abstract	David Loosli
Dec 21, 2017	1.0	corrections of chapter 4 and 5 as well as final review	David Loosli

Abstract

The Muen Separation Kernel (SK) is a specialised microkernel developed as a platform for high-security systems at the University of Applied Sciences Rapperswil (HSR). Muen ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system. The programming language SPARK 2014 is used to achieve a particularly high degree of trustworthiness. The Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components.

This feasibility study investigates the ARMv8-A architecture and in particular the AArch64 Virtualization Extensions introduced with the latest ARM architecture and evaluates how this technology could be used for porting the Muen SK to ARM. In order to be able to achieve this, the mechanisms used by Muen SK are first examined in detail. Based on this investigation, the requirements for a target processor architecture are derived and compared with the features provided by the ARMv8-A architecture. Since the target hardware platform for this study is the Raspberry Pi 3, the requirements declared as „implementation defined“ by the ARM documentation are finally assessed with respect to this System on Chip designed by the Raspberry Pi Foundation.

Contents

1 Introduction

The evolution within the last years in the world of information technology not only led to a tremendous increase of mobile devices and networking, but also let the world economy dream of a new technological era, the Industry 4.0[1]. This fourth industrial revolution is characterized in particular by the interconnection of objects and people within a so called information network. In this context, the most frequently mentioned keywords are Internet-of-Things, Cloud Computing and Bioengineering.

One of the consequences of the integration of autonomously communicating devices into our daily life is that a lot of sensitive data is collected and stored that needs best possible access control. A mathematically provable secure approach to control the access to sensitive data is the theory of the Separation Kernel published by John Rushby in a paper presented at the 8th ACM Symposium on Operating System Principles in December 1981[2]. Based on this theoretical foundations and the Intel hardware virtualization extension, Reto Buerki and Adrian-Ken Rueegsegger designed the Muen Separation Kernel (SK) as their Master Thesis at the University of Applied Sciences Rapperswil (HSR)[3]. The Muen SK ensures a strict and reliable isolation of components and protects critical security functions against unreliable software running on the same physical system.

A second consequence of the fourth industrial revolution is the need for small devices with low energy consumption and low production costs that still meet the state of the art with respect to processor architecture and peripheral device integration. Since many of these small devices, especially mobile devices, use an ARM central processing unit (CPU) or an ARM based system on chip (SoC) one could also determine enormous improvements up to the latest ARM architecture, the so called ARMv8 architecture [4].

This Student Research Study, which is part of the Bachelor of Science in Computer Science program at the University of Applied Sciences Rapperswil (HSR), investigates the possibility of porting the Muen SK to the ARMv8 architecture. As the Muen SK was developed specifically for the Intel x86/64 architecture and uses the Intel VT-x and VT-d technology to separate the components, the aim of this feasibility study is to take a closer look at the ARMv8 architecture and in particular the AArch64 Virtualization Extension (VE) introduced with the latest ARM processors. The target hardware for this study is the Raspberry Pi 3 [5].

[1] [3] Devezas, Leitão, and Sarygulov. *Industry 4.0 - Entrepreneurship and Structural Change in the New Digital Landscape*. 2017, Chapter 1, page 2 f.
[2] [17] Rushby. "Design and Verification of Secure Systems". 1981.
[3] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013.
[4] https://en.wikipedia.org/wiki/ARM_architecture#ARMv8-A, December 21, 2017
[5] cf. https://www.raspberrypi.org/, December 21, 2017

1.1 Structure of the Study

The study is divided into three main parts followed by a summarizing conclusion including a risk assessment for the planned Bachelor Thesis to port the Muen SK to the ARMv8 architecture. In the first part (chapter 2), an overview of the Muen SK is given and the most important hardware dependent features are described, from which the general hardware requirements are derived. In the next chapter 3, an introduction to the ARMv8 architecture is presented with focus on the AArch64 architecture and the Virtualization Extension (VE) followed by a qualification of these features with respect to the derived hardware requirements from the first part. As the target hardware for this study is the Raspberry Pi 3, the third part of this document (chapter 4) is dedicated to a detailed description of this single board computer considering hardware related features used by the Muen SK.

1.2 Related Documents

As the focus of this study lies on the feasibility of porting the Muen SK to the ARMv8 architecture, many related documents apart from this document were elaborated. As examples, there can be mentioned the Raspberry Pi 3 Beginner's Guide and all the Evaluation Cases illustrated with small coding examples. All this documents are an integral part of the Student Research Project. A list can be found in the appendix of this document.

1.3 Literature

Due to the task description of the Student Research Project [6], the Muen Report[7] with the related documents and the official ARM documentation, i.e. the ARMv8 Architecture Reference Manual[8] and the ARM Cortex-A Series Programmer's Guide[9], were used as the principal literature. A detailed list of referenced literature can be found in the bibliography at the end of this document (cf. Bibliography).

Because a detailed and with respect to the AArch64 architecture complete Raspberry Pi 3 hardware reference manual did not exist at the time of writing, chapter 4 of this study had to be based on the VideoCore Reference Manual[10] and the BCM2835 ARM Peripherals documentation[11] for the Raspberry Pi 1 as well as different online sources mentioned in the corresponding section 4.1.1 of this document.

[6] cf. assignment from the AVT platform, Appendix B
[7] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013.
[8] [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017.
[9] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015.
[10] [13] n.a. *VideoCore IV 3D Architecture Reference Guide*. 2013.
[11] [11] n.a. *BCM2835 ARM Peripherals*. 2012.

2 Muen Separation Kernel

The design and implementation of the Muen SK is premised on three basic concepts: first of all the Separation Kernel principle, formal verification and hardware supported virtualization.

The concept of a Separation Kernel was introduced by John Rushby in a paper presented at the 8th ACM Symposium on Operating System Principles in December 1981 as a solution to the problem with the development and verification of large, complex security kernels[1]. His proposition was to basically adapt the principles of a distributed system to a single processor to avoid the aforementioned problem. As a consequence, such a system has to physically isolate all the subjects that are part of the security policy. The communication and the access to shared resources of all these subjects must be handled only through likewise isolated, so called trusted components that can be verified[2]. Finally, Rushby verified the outlined proposition with a Proof of Separability[3].

As the verification is a compulsory consequence of the Separation Kernel principle, an implementation of a Separation Kernel has to use a programming language that is amenable to formal verification. Therefore, the SPARK programming language was chosen to write the Muen SK. SPARK is a formally analysable subset of the programming language Ada and used for implementing high integrity systems [4]. A introduction to the programming language SPARK and the related derived requirements can be found in section 2.7.

Another deducible consequence of the Separation Kernel principle is the requirement of a sufficiently small code base for the implementation of such a kernel[5]. To achieve this, the Muen SK relies on the hardware virtualization support of the Intel x86 architecture[6]. To get the full virtualization support for a desktop environment, the Intel IA-32e/64-bit architecture was chosen as the target platform of the Muen SK[7]. Therefore, a first basic requirement for a processor architecture, to be able to run the Muen SK on, can be derived as:

> ***REQ-0:*** *The processor architecture has to support 64 bit datapath widths, integer size and memory address widths as well as to be able to execute 32 bit applications.*

[1] [17] Rushby. "Design and Verification of Secure Systems". 1981, Section 1, page 3 f.
[2] [17] Rushby. "Design and Verification of Secure Systems". 1981, Section 2 f., page 5 ff.
[3] [17] Rushby. "Design and Verification of Secure Systems". 1981, Section 4, page 11 ff.
[4] cf. https://www.adacore.com/sparkpro, December 21, 2017
[5] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 2.4, page 14.
[6] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 2.3, page 11 ff.
[7] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, Section 3.2, page 20.

2.1 Virtualization Basics

A hypervisor or virtual machine monitor (VMM) [8] is special software that emulates computer hardware. In general, two different types of hypervisors are classified[9]: *Type I* native or bare-metal hypervisors and *Type II* hosted hypervisors. A Type I hypervisor directly runs on the target hardware to control and manage the guest operating system, whereas a Type II hypervisor makes use of a conventional operating system. As a Type I hypervisor has comprehensive control over the processor(s) and other platform hardware as well as over the guest software (e.g. memory access, communication etc.), it can also be used as a mechanism for separation purpose[10]. Therefore, the Muen SK can be classified as a Type I hypervisor.

A hypervisor multiplexes the hardware by the usage of different virtualization techniques to provide a virtual environment to the guest software in a way that lets the guest software gain the impression of running directly on the hardware. One approach to achieve this, is to add another privilege level or protection ring to a processor architecture. A protection ring is one of two or more hierarchical layers of privilege within the architecture of a computer system. Normally, the processor architecture enforces this layering by providing different execution modes on hardware level. As an example - in standard protected mode on an Intel x86 architecture there exist four privilege levels or protection rings with ring 0 as the most privileged one whereas ring 3 having the least privileges [11].

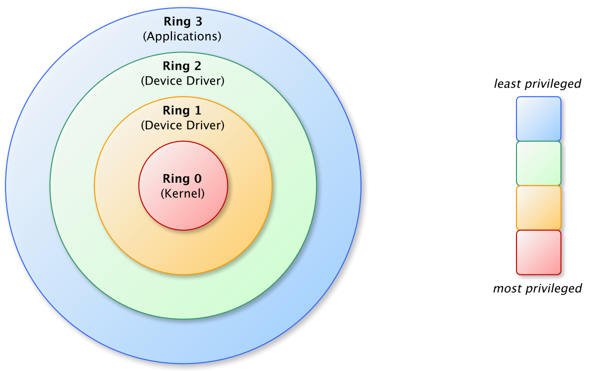


Figure 2.1: Intel x86 protection mode, protection rings hierarchy

As already mentioned, the Muen SK makes use of the Intel Virtualization Technology (VT) to fulfil the requirement of a small code base. One of the basic features of the Intel VT is the so called Intel VT-x.

[8]Because the ARMv8 architecture uses the terms secure monitor and monitor mode for a separate exception level, the expression hypervisor is used instead of VMM throughout this document.

[9][16] Popek and Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. 1974, the first classification approach.

[10][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3, page 11.

[11]It is absolutely important to note that ring 0 has the **most** privileges - because the ARM Exception Levels define the privileges exactly the other way round by giving the Exception Level 0 the **least** privileges; cf. https://en.wikipedia.org/wiki/Protection_ring, December 21, 2017

This feature introduces a new hypervisor execution level with an additional protection ring „-1" as well as some new VMX instructions that simplify the switching between a hypervisor running in VMX root operation and guest software executing in VMX non-root operation [12]. Hence, to be able to execute the Muen SK in hypervisor mode, a target processor architecture has to meet the following requirement:

> ***REQ-1:*** *The target processor architecture must provide a virtualization extension that is capable of running a Type I hypervisor. This requirement includes the hardware assisted support for an additional privilege level and instructions for a simplified switch between this additional and other privilege level.*

Another important feature of the VT-x virtualization technology is that VM exits and entries are handled automatically while the exact behaviour still stays configurable[13]. To do so, a logical processor uses virtual machine control data structures (VMCS) to manage transitions into and out of the VMX non-root operation as well as the processor behaviour in VMX non-root operation[14]. An illustrating example is a VM exit that automatically stores the guest processor state into the guest state area of the VMCS. But one has to be aware that registers, which can be saved and loaded by the hypervisor itself (e.g. general purpose registers), are not stored automatically[15]. Therefore:

> ***REQ-2:*** *The target processor architecture must provide a virtualization extension that supports an automatic handling of guest exits (i.e. traps) and entries. At least, the target processor architecture must provide a support mechanism to completely save and load all the relevant guest state structures.*

2.2 Memory

In modern computer systems, usually different memory and storage technologies are used as an attempt to find the best possible compromise between access time, cost and persistence properties. The first two criteria are interrelated by the fact that the shorter the access times of a specific type of memory is, the more expensive they are. The third criterion not only considers the persistence in the proper sense, i.e. volatile or persistent, but also other properties like the degree of hardware supported manageability. Therefore, memory and storage are normally organized in a so called memory hierarchy to use the advantages of the various components while, at the same time, circumventing their disadvantages [16]. A standard modern memory hierarchy is composed of[17] [18]:

[12] more details can be found in [2], section 2.3.1, and [12], volume 3C, chapter 23 f., page 1083 ff.
[13] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, section 2.3, page 12.
[14] [12] n.a. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3.* 2017, volume 3C, section 24.4, page 1090 ff.
[15] [14] Neiger et al. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". 2006, page 170.
[16] cf. https://de.wikipedia.org/wiki/Speicherverwaltung, December 21, 2017
[17] [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 7.1.3, page 393 f.
[18] https://en.wikipedia.org/wiki/Memory_hierarchy, December 21, 2017

- *CPU Registers:* The fastest (typically one clock cycle) and most expensive type of memory that locates in the processor itself.

- *Caches:* A state of the art processor has numerous internal and shared caches (Static Random Access Memory SRAM), organized in up to four levels with increasing access times from a few tens of clock cycles down to a few hundreds, and additional hardware caching structures, e.g. Translation Lookaside Buffers (TBL) and Branch Prediction Caches (BPC).

- *Primary Storage (main memory):* This type of memory is also referred to as Dynamic Random Access Memory (DRAM). Its speed is moderate with up to 10 GB per second but still relatively affordable. With respect to primary storage, two different applications are distinguished - physical RAM and Virtual Memory (cf. Memory Management Unit 2.2.2).

- *Secondary Storage (disk storage):* On Secondary Storage, data can be permanently stored. It is much cheaper than primary storage but about 10'000 times slower. The most known representatives are Hard Disk Drives (HDD) or Solid State Disks (SSD).

- *Tertiary Storage (input storage):* This category includes various types of removable media devices such as USB devices or SD cards as well as remote storage and peripherals. It is the slowest and cheapest kind of storage.

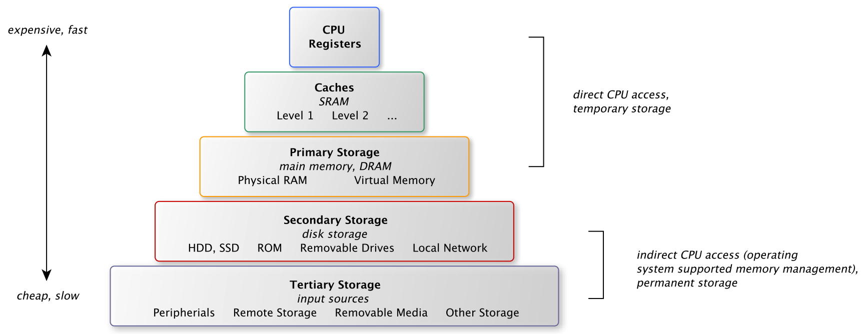

expensive, fast
cheap, slow
CPU Registers
Caches
SRAM
Level 1 Level 2 ...
Primary Storage
main memory, DRAM
Physical RAM Virtual Memory
Secondary Storage
disk storage
HDD, SSD ROM Removable Drives Local Network
Tertiary Storage
input sources
Peripherials Remote Storage Removable Media Other Storage
direct CPU access, temporary storage
indirect CPU access (operating system supported memory management), permanent storage

Figure 2.2: example of a memory hierarchy

At this point, one has to remember the strict distinction between memory and storage. While the CPU has direct access to the memory - whether through the processor's hardware structures or over the memory bus - storage is only available as an I/O device. Pointing out this difference is important because all memory resources of a system running the Muen SK are static and explicitly specified in the so called system policy[19]. This, for example, implies that there is no such mechanism implemented for loading missing page contents from a storage device after a page fault or page miss, as most of the common operating system kernels would do, and that no considerations about side and covert

[19][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4.2.1, page 24.

channels with respect to disk caches or other storage structures have to be made. Since a subject [20] cannot change its own address space, also the page tables are static and therefore can be generated in advance according to the relevant information in the system policy. As for the storage, it is treated by the Muen SK as a pure I/O device (cf. Device Handling 2.5) .

2.2.1 Caches

As already mentioned, only cache and caching structures, that are directly accessible to the CPU, have to be considered with respect to the fundamental requirement of the Muen SK to completely separate the subjects and thus to eliminate side and covert channels. The main problem with caches is that they are shared and can normally only be controlled to a limited degree[21]. Due to performance aspects, the Muen SK has to enable the caches and caching structures. But as the Muen SK uses the Intel Virtualization Extension at least the Translation Lookaside Buffer (cf. section 2.2.2) is cleared automatically. Therefore, a processor architecture has to fulfil the following requirement:

> ***REQ-3:*** *The target processor architecture shall provide a minimal set of cache management features and an automatic cache clearing feature in the context of virtualization. At least, the target processor architecture must provide a support mechanism to clear caches manually.*

Even though out of scope for this study, the cache colouring mechanism has to be mentioned here. This technique first divides the cache into disjoint units and assigns a „color " to each of these partitions. Every process then is assigned a certain color to. A cache area of certain color can only be accessed by processes with the corresponding color. This technique is not only used for performance optimizations but can also serve as a mechanism to prevent processor caches from being used as high-bandwidth side channels[22]. The developers of the Muen SK mentioned this mechanism as one of possible future enhancements[23].

2.2.2 Memory Management

In modern computer systems, the management of the main memory is taken over by a hardware component called Memory Management Unit (MMU) that is usually integrated into the processor. The MMU handles all the access of the CPU to the main memory. In general, it has two main functions: on the one hand it allows the implementation of virtual memory and on the other hand it can manage memory protection and cache control [24].

[20] In the context of the Muen SK, a subject is defined as one of multiple, isolated and through well-defined interfaces interacting components. More informations can be found in section 3.3 and 4.3 in [2]

[21] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.2.1.2, page 7.

[22] [1] Braun, Jana, and Boneh. “Robust and Efficient Elimination of Cache and Timing Side Channels”. 2015, section 4, page 3, with references to other literature.

[23] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 6.2.1.1, page 76.

[24] https://en.wikipedia.org/wiki/Memory_management_unit, December 21, 2017

Virtual Memory is a technique that abstracts the available memory and storage resources on a computer system in such a way that a process is given the illusion of running alone on that system and having unrestricted access to the systems main memory [25]. To be able to provide a linear [26] but virtual logical address space to a process, a modern MMU uses a mechanism called paging. With paging the physical as well as the virtual address space are divided into units with a fixed size. In the context of physical memory, these units are called page frames whereas in the context of virtual memory they are denoted as pages. The mapping between a physical page frame and a virtual page is done by a so called page table that uses the two-part virtual address to calculate the physical address of the page frame.

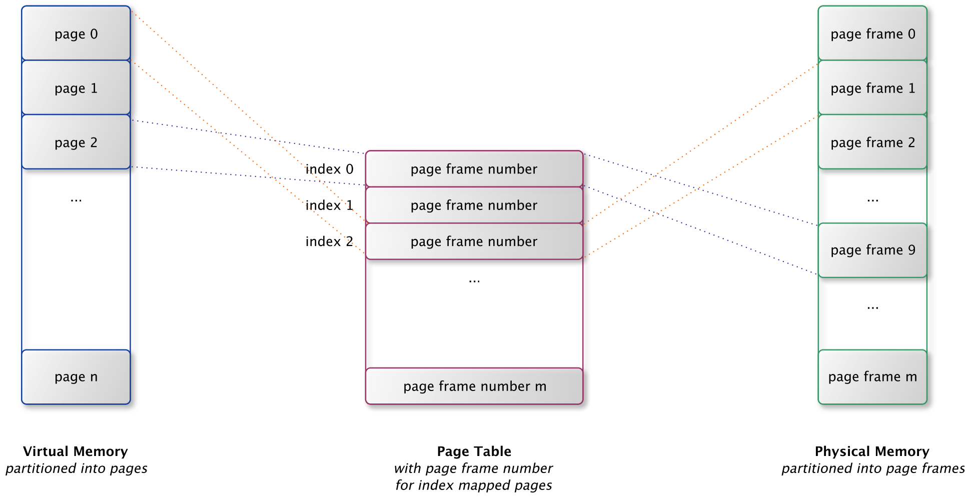


Figure 2.3: example of a one level paging with partitioning

As shown in figure 2.4, a virtual address is divided into two parts - a page number and an address offset. The page number serves as an index into the page table to read out the content at this specified address, namely the page frame number. Then the page frame number is multiplied by the predefined page size to get the base address of the corresponding page frame in the physical memory. The physical address for the requested virtual address can then be obtained by adding the offset of the virtual address to this base address[27].

To prevent illegal accesses, the page table must be initialized completely and undefined page table entries have to be invalidated by at least setting an invalidation bit. In combination with a multiprocessor environment, this requirement can lead to large page tables. One way to address this problem, is to implement so called multi-level page tables. As an example, a two level page table hierarchy is presented: In such a case, the virtual address is divided into three parts. The first part contains the

[25] https://en.wikipedia.org/wiki/Virtual_memory, December 21, 2017
[26] https://en.wikipedia.org/wiki/Flat_memory_model, December 21, 2017
[27] [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 7.5.1, paragraph Seitenbasierte Adressumsetzung, page 450 ff.

directory index to the first page table. The entry in this first page table does not directly return the page frame number, but contains the address of a second page table. The second part of the virtual address is now used as an index into this second page table. The entry at the corresponding index then contains the page frame number, which serves as the basis for the actual address resolution according to the principle described above[28].

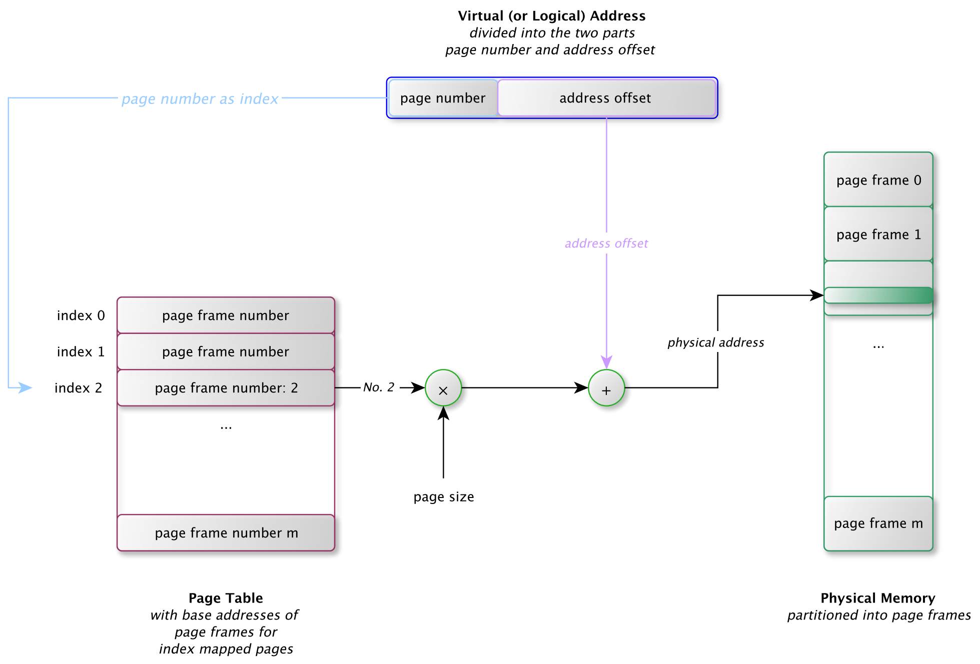


Figure 2.4: example of a one level address translation

To improve the performance of the MMU's address translation, modern processor architectures rely on the implementation of an associative cache structure, the Translation Lookaside Buffer (TLB) . The address translation principle described above remains the same, but instead of a direct lookup of the first part of the virtual address (i.e. the page number) in a page table the MMU first takes a look at the TLB. If the corresponding page entry can be found in the TLB it loads the physical base address directly from there - else the corresponding page descriptor gets first loaded into the TLB from the according page table before returning it to the address translation process[29]. In the context of the Muen SK, the separation concerns described in the section Caches 2.2.1 have to be considered accordingly (i.e. side and convert channels).

[28][4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 7.5.1, paragraph Seitenbasierte Adressumsetzung, page 455 ff.

[29][4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 7.5.1, page 446 f.

The Muen Separation Kernel uses the virtualization functions of the Intel IA-32e mode on the one hand for the implementation of the Type I hypervisor and on the other hand for the partitioning and the separation of the different subjects. Since the 80286 processor, the Intel x86 processor architecture provides an integrated MMU, that is capable of handling the paging mechanism for address virtualization. The corresponding page tables can be defined and used on a per process basis and also serve to define properties and permissions (i.e. memory protection). In addition, the MMU provided by Intel validates and enforces compliance with these additional memory protection features. A hierarchical arrangement of the page tables also enables multi-level paging - the Intel IA-32e mode supports up to 4 such levels and allows page sizes of 4 KB, 2 MB and 1 GB[30]. Therefore, the target architecture has to support the following features:

> ***REQ-4:*** *The target processor architecture has to provide a Memory Management Unit that supports:*
>
> *(i) memory virtualization on a per subject basis (one page table per subject),*
>
> *(ii) definition of properties and permissions per page table (read/write access, execute disable, caching behaviour),*
>
> *(iii) checking and enforcement of defined properties and permissions,*
>
> *(iv) different page sizes (i.e. large page support [31]), but at least a 4 KB page size [32].*

2.2.3 Advanced Memory Virtualization

When using a hypervisor with different guest operating systems (i.e. virtual machines), the address virtualization technologie described above has to be extended with a second layer. The hypervisor assigns a first layer virtual memory area to the guest system, which is interpreted by the guest system as its own physical memory. If the guest system is running a modern operating system, it will use the address translation mechanism again for its applications, creating a complete second address translation layer. In order to be able to cope with the associated performance issues as well as the complexity of the hypervisor implementation, Intel's x86 virtualization technology „Extended Page Tables (EPT)“ provides a hardware assisted Second Level Address Translation (SLAT, also known as nested paging) mechanism.

[30] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.2.2, page 8 f.

[31] Most current CPU architectures support bigger pages, but name it differently: huge pages, super pages or large pages are only the most often used terms.

[32] At the time of writing, the Muen SK only relies on 4KB pages. But as discussed in the meeting of November 20, 2017, the question about large page support by the ARM architecture should be answered in this study too.

The Muen architecture supports native subjects as well as complex Virtual Machines (VM) running their own operating system[33]. To be able to run such complex VM's without having an enormous adaptation effort, the Muen SK makes use of the Intel's x86 SLAT virtualization technology EPT[34]. Therefore, the following must apply:

> ***REQ-5:*** *The target processor architecture must support hardware assisted second level address translation (SLAT).*

2.2.4 Multicore Environment

Even though out of scope for this study, the multicore environment topic has to be mentioned here. A processor architecture that implements more than one core is called a multicore processor [35]. Another feature often implemented by modern processor architectures is the hardware assisted multithreading ability. A processor architecture, that is capable of multithreading, subdivides a central processing unit (CPU) or a single core in a multicore processor into logical cores to execute multiple processes or threads concurrently [36]. While in a multicore environment the CPU itself as well as core specific resources (e.g. MMU, TLB and Caches) are multiplied, logical cores have to share these resources.

First of all, the Muen SK does not concern itself with memory management. All the page table structures needed in a computing system are created by the Muen policy tools and statically initialized at the system startup by the initialization code. In an initialized multicore or multithreading environment, all logical cores execute exactly the same (i.e. binary identical) Muen kernel code. Although each kernel has its own stack page and a page to store per core data, this is fully transparent to the kernels due to the usage of different page table structures per kernel[37]. In the current version of the Muen SK, the multithreading features of the Intel x86 architecture are switched off [38]. Therefore, to be able to port the Muen SK to another multicore or multithreading processor architecture, a target architecture has to provide the following feature:

> ***REQ-6:*** *A multicore target processor architecture has to provide a mechanism to switch off the multithreading mechanism on a per core basis, if multithreading is supported.*

The Muen SK uses a barrier as synchronization mechanism to avoid any interprocessor drift in the context of scheduling plans and hence to eliminate timing side channels. This barrier guarantees that all logical cores have arrived at a specific execution point, i.c. on major frame transition, and are synchronized by waiting for the release. A sense-reversing barrier implemented in SPARK is used as

[33] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4, page 22 f.
[34] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.1.2, page 13, and section 3.3.3, page 22.
[35] https://en.wikipedia.org/wiki/Multi-core_processor, December 21, 2017
[36] https://en.wikipedia.org/wiki/Multithreading_(computer_architecture), December 21, 2017
[37] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 3.4.6, page 28, and section 4.4.2, page 46.
[38] cf. Besprechungsnotiz November 27, 2017 - section 2, page 2

the barrier mechanism [39]. On assembly level, the barrier is realized with a spinlock using the atomic XCHG processor swapping instruction[40].

> ***REQ-7:*** *A multicore target processor architecture shall provide a barrier synchronization mechanism. At least it must offer an atomic swapping instruction to support the according spinlock implementation.*

2.3 Interruption Handling

The various processor architectures and the corresponding literature use different terms (e.g. exception, interrupt, signal, event) for the temporary interruption of a running process by an interruption cause. For this study, the term interruption is used as a generic term for all types of temporary interruptions. The terms for the different types of interruptions are then described in detail according to the usage and the definitions in the respective topic. For example, in this chapter the interruption types are defined as used in the Muen report.

In the literature one can find various criteria to distinguish between interruptions and hence quite a few different categorisations of interruptions[41][42]. For this study only the following criteria are relevant:

- *internal vs. external:* An interruption caused by a device outside the processor is referred to as external while interruptions caused by the processor itself are considered as internal. For example, a keyboard device signalling an input has to be qualified as external - in contrast, interruptions, that occur in response to a processing error, such as referencing an invalid address in memory, division by zero or similar error condition, have to be looked upon as internal.

- *hardware vs. software:* While a hardware interruption is routed to the processor via a channel that is effectively implemented in hardware, the software interruption originates from a program command. In the case of software interruptions, a distinction can also be made between *intentional* and *defective* interruptions. Applying these criteria, a keyboard interruption reflects a hardware interruption, a divison by zero would be a defective software interruption and the execution of a trapping instruction could be qualified as intentional software interruption.

Nearly every processor architecture uses a different naming and separation of the components that are involved in an interruption processing. Therefore, the following explanation of a typical device interruption process is simplified with respect to the components (esp. the CPU) as well as to the architecture.

[39]This type of barrier is described in the book The Art of Multiprocessor Programming by Maurice Herlihy and Nir Shavit

[40][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, section 3.4.6, page 28, and section 4.4.2.2, page 47.

[41][4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 6.2.2, page 300 ff.

[42][18] Tanenbaum and Bos. *Moderne Betriebssysteme.* 2016, section 5.1.5, page 427 ff.

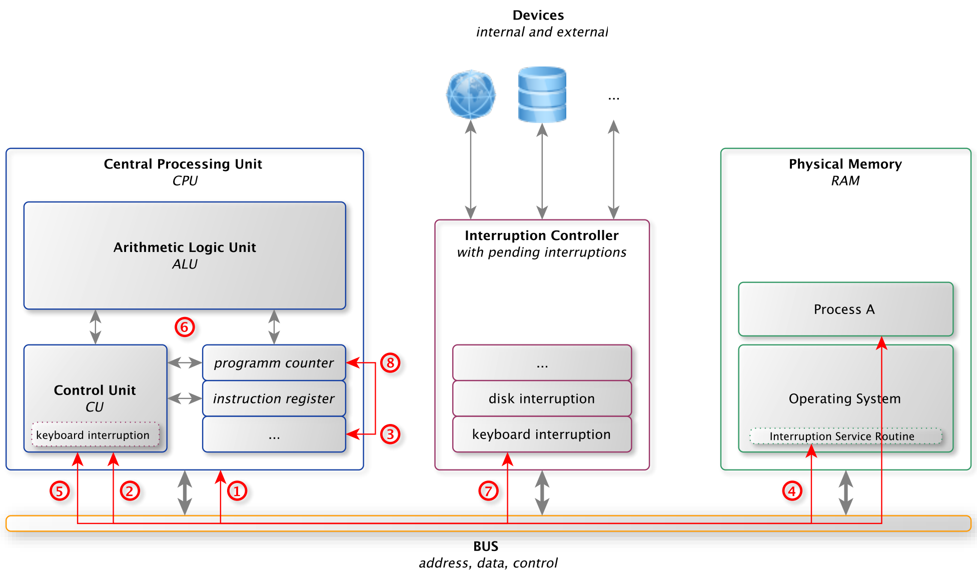


Figure 2.5: simplified interruption process

(1) The starting situation is illustrated in the figure 2.5 - a process *A* running on top of an operating system, both loaded into RAM, is executed by the CPU.

(2) As soon as an interruption (i.c. caused by a keyboard) occurs, the Interruption Controller informs the Control Unit (CU) about it. The CU then stops the execution of the process *A*.

(3) To be able to restore the state of process *A*, the CU saves (on some processor architectures automatically) the programm counter and other registers used by process *A*.

(4) Then, the CU checks the cause number of the interruption and retrieves the base address for the according Interruption Service Routine (ISR) defined in the operation systems code.

(5) After that, the CU loads the instructions of the ISR

(6) and executes its code until the end of the ISR.

(7) When the execution of the ISR is finished, the CU informs the Interrupt Controller with an acknowledgement about the processed interruption.

(8) Last, the CU restores the registers of the process A and continues executing the corresponding instructions. Process *A* does not even realise the interruption.

2.3.1 Programmable Interrupt Controller

Even modern processor architectures often implement only a few input lines for interruption signals and only support a simple interruption logic. In such cases, an external device, the Programmable Interrupt Controller (PIC), can be attached to the associated processor line(s) to first of all combine different interrupt sources onto one CPU interruption line, but also to allow the assignment of priorities to different kind or groups of interruption causes or to mask different types of interruptions[43]. A Programmable Interrupt Controller normally features the following registers: *(a)* an Interruption Request Register (IRR) that specifies the pending interruptions, *b* an In Service Register (ISR) that records the acknowledged but still waiting for an End of Interrupt (EOI) interruptions and *(c)* an Interrupt Mask Register (IMR) that defines which interrupts are to be ignored and not acknowledged.

The Muen SK makes use of Intel's Advanced Programmable Interrupt Controller (APIC) that is composed of two components - the Local APIC as a part of every physical CPU and the I/O-APIC as a part of the chipset[44]. The most important features of this interruption architecture are [45]:

- local interruption management on a per CPU basis and therefore better performance
- support for inter-processor interrupts (IPI) between Local APICs
- Local APICs provide a high-resolution timer for interval and one-off mode usage
- flexible interruption configuration on a per interruption type basis
- support for Message Signaled Interrupts (MSI) [46]
- priority definition on a per interruption type basis
- interrupt and NMI window exiting feature associated with virtualization[47]
- I/O APIC support multiple interruption input lines
- I/O APIC redirection table to route interruptions to one or more Local APIC(s)

The exact determination of the APIC features required by the Muen SK and the therefore resulting requisites for a target architecture are elaborated in the following sections. But in this context, it can already be stated that:

> ***REQ-8:*** *A target processor architecture has to provide a mechanism to programmatically handle interruptions.*

[43][4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 6.2.2, page 306 ff.
[44][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, section 2.2.4, page 9 f.
[45]https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller, December 21, 2017
[46]https://en.wikipedia.org/wiki/Message_Signaled_Interrupts, December 21, 2017
[47][14] Neiger et al. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". 2006, page 171 ff.

2.3.2 Interrupts

In the context of the Muen SK, interrupts are defined as external hardware interruptions. As an example, the Muen report mentions a network card that generates an interrupt whenever a data packet is received[48].

The Muen SK uses the Intel VT-x technology to inform a subject about an external interrupt. An external interrupt request (IRQ) is routed to the in the system policy statically defined subject through the Muen SK that provides a per subject array with up to 32 pending interrupts for delivery. To achieve this routing mechanism, the Muen SK has to enable the I/O APIC and rely on the LAPIC feature to be able to specify not only the physical CPU, that the subject is allocated to, but also the subject itself[49]. To improve the interrupt delivery with respect to performance, the Muen SK also makes use of Intel's virtualization mechanism called interrupt window exiting[50]. Therefore, a target processor architecture has to meet the following requirements:

> ***REQ-9:*** *A target processor architecture has to provide an interruption handling that guarantees the **exclusive** treatment of interrupts by the separation kernel.*

Another important aspect of Intel's x86 architecture is that it allows to enable or disable interrupts for the VMX root mode. This is done by not setting the IF interrupt flag in the host's FLAGS register. The Muen SK uses this mechanism to simplify the the kernel code and to assure that the Muen SK is not disrupted by external interrupts[51]. Therefore, a target processor architecture has to manifest a similar feature:

> ***REQ-10:*** *A target processor architecture has to provide an enabling and disabling mechanism for (external) interrupts, at least for the execution of the hypervisor code.*

2.3.3 Exceptions and Software Generated Interrupts

In the context of the Muen SK, exceptions are defined as defected software interruptions. This means, that an exception is an interruption generated by the processor itself detecting an error condition during the execution of an instruction. As an example, the division by zero is given. While exceptions denote defected software interruptions, software generated interrupts have to be qualified as intentional software interruptions[52]. As both interruption types are treated similarly, they are subsumed in this section.

[48][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.3, page 9.
[49][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.6, page 50 f.
[50][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.4, page 49.
[51][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 4.4.6, page 51.
[52][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.3, page 9, section 3.4.4, page 27 f., and section 4.4.7, page 51 f.

First of all, the Muen report distinguishes between exceptions and software generated interrupts that occur in VMX non-root mode (i.e. while executing a subject) and in VMX root mode (i.e. while the Muen SK is executed). As a basic requirement for the Muen SK, the ability to prove the absence of runtime errors is stated. Hence, if an exception (or even less likely a software generated interruption as well as a non maskable interrupt) occurs during the regular execution of the Muen SK in VMX root mode, it would indicate a serious problem in the kernel code and therefore the whole system would be halted.

In VMX non-root mode, there has to be differentiated between native and VM subjects. While VM subjects must implement their own exception handling and hence exceptions and software generated interrupts must not result in a subject exit, native subjects do not react on exceptions but handover the execution to the kernel[53] (cf. trap in section 2.3.4).

> ***REQ-11:*** *A target processor architecture must support a mechanism to enable and disable exceptions and software generated interrupts resulting in an exit of the guest subject.*

In the context of exceptions and software generated interrupts, also system management exceptions (e.g. non maskable interrupts) have to be mentioned. The Muen SK makes sure that this type of interrupts are not handled by the subject itself but result in a subject exit by all means. Therefore:

> ***REQ-12:*** *A target processor architecture shall provide a mechanism to force system management exceptions to lead to an exit of a guest subject.*

2.3.4 Traps

The term trap, as used by the Muen report, subsumes different kind of interruptions and virtualization techniques that lead to a VM exit. As examples for VM exits, the documentation mentions the execution of a privileged operation or a constrained instruction. The Muen SK uses the VT-x technology to provide the possibility of specifying a per subject trap table in the system policy, whereby all of the VMX basic exit reasons defined by Intel can be configured according to the subjects needs except the following, by the Muen SK internally reserved traps[54]:

- external interrupt (cf. section 2.3.2)
- interrupt window (cf. section 2.3.2)
- VMCALL (cf. section 2.3.5)
- VMX preemption timer expired (cf. section 2.4)

[53] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, section 4.4.7, page 51 f.
[54] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, section 4.4.5, page 49 f.

Therefore, a virtualization extension or interrupt handling mechanism for a target processor architecture has to meet the following requirement:

> ***REQ-13:*** *A target processor architecture must be able to differentiate between exit reasons of a guest system and to handle them as specified per subject.*

2.3.5 Events

The Muen SK implements an event mechanism that is used for inter-subject signalization. This means, that a subject is allowed to send an event to another subject as long as this has been granted by an entry in the subject's policy event table.

The implementation of this event mechanism is based on the VMCALL VMX instruction. Hence, when a subject sends an event to a destination subject, it results in a trap into the Muen SK that handles the event according to the system policy. Additionally, an optional inter-processor interrupt (IPI) can be emitted to speed up the inter-core interrupt delivery. If this option is enabled for an interrupt event, an inter-processor interrupt is delivered to the CPU of the destination subject. Finally, this results in the preemption [55] of the subject, that is executed at the moment on the destination CPU, and therefore the immediate delivery of the event. A target architecture should therefore have the ability to provide a similar mechanism:

> ***REQ-14:*** *A target processor architecture should provide a technique to fast process interruptions between cores.*

2.4 Timers

In the context of timers, the clock generator has to be mentioned first. In a system, the clock generator is responsible for producing a constant timing signal. This so called clock signal normally corresponds to a frequency generated by a quarz piezo-electric oscillator [56]. This signal is then used by all components of the system to synchronize a circuit's operation, including the timer components. In this documentation, the term „clock" refers only to this initial output signal. All other periodic signals, that depend on this initial signal and that are mentioned in the context of synchronization, are termed „timer" (even though in most literature this terms are used interchangeably).

A timer is an integrated circuit that normally signals an interruption after a configurable amount of „time" (Programmable Interval Timer PIT [57]) or after an overflow of a counter register. There exist many different types of and definitions for timers realized in hardware according to their usage, e.g. pause function timers, one-shot timers, periodic timers, time-slicing timers and watchdog timers. For this study, only

[55] https://en.wikipedia.org/wiki/Preemption_(computing), December 21, 2017
[56] https://en.wikipedia.org/wiki/Clock_generator, December 21, 2017
[57] https://en.wikipedia.org/wiki/Programmable_interval_timer, December 21, 2017

the system timers are important: A system timer is a timer integrated into a hardware component that is responsible for producing a periodic signal used by the whole component. Regardless of the designation and application of a timer, its functionality can be described as a device that uses a high-speed clock input to provide a series of time or count-related interruption signals. As a single counter can only generate short time intervals due to the high-speed frequency of the clock, a technique called cascading can be used with some additional programmable scaling registers to multiply this short time intervals and thereby generating longer time intervals[58]. An alternative to programmable scaling registers is the cascading of multiple timer components. A simple unscaled programmable timer can be described as follows:

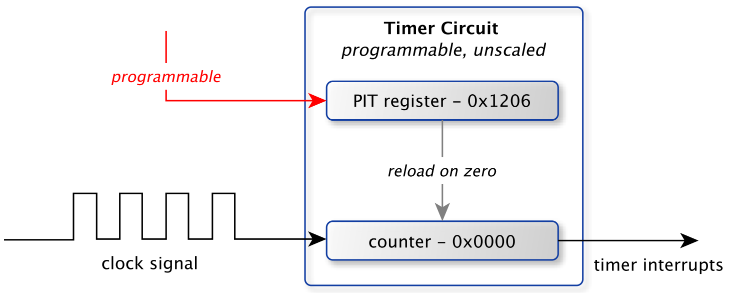


Figure 2.6: timer component

The most important timer used by the Muen SK is the VMX preemption timer in the context of the statically defined scheduling mechanism for subjects running on the same core[59]. This timer provided by Intel's virtualization extension can be set to a specific value according to the time slice definition for the corresponding subject. The subject is then automatically preempted by the processor when the time slice defined in the scheduling plan is over. After that, the Muen SK hands over the execution to the next subject according to the scheduling plan.

> ***REQ-15:*** *A target processor architecture shall provide a preemptive mechanism on a per subject basis. At least it must provide a timer per core.*

2.5 Device Handling

Basically, there are three possibilities to handle devices[60]. The first possibility (and least reasonable one) is the code or software based device handling. It makes use of a polling mechanism by continuously checking the status register of the desired device. The second possibility is an interrupt based approach explained in the previous sections. The third one is called Direct Memory Access (DMA).

[58][4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 6.2.2, page 309.
[59][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, section 4.4.3, page 47 f.
[60][4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung.* 2010, section 6.2, page 300.

This last technique allows attached peripheral devices to directly interact with the main memory over a usually external hardware controller (i.e. DMA controller). The CPU only has to configure the DMA controller at initialization time - after that the controller acts without the usage of the CPU[61].

At the time of its writing, the Muen report declared the device virtualization out of scope[62]. But in the past years the implementation of the Muen SK has been extended and now uses Intel's VT-d Virtualization Technology for Directed I/O to virtualize I/O devices through an IOMMU. The virtualization extension VT-d simplifies the direct assignment of devices to virtual machines in two ways - first, by providing secure direct memory access (DMA) and second, by extending device interrupt remapping functionality. Even though a further evaluation of this topic is out of scope for this study, at least the following requirement can be stated:

> ***REQ-16:*** *A target processor architecture must provide a mechanism to virtualize I/O devices by completely isolating the access to devices and providing support for associated interruption and memory features.*

2.6 Floating Point

Modern processor architectures usually implement a so called Floating Point Unit (FPU), a specialized integrated circuit used for floating point calculations. As these floating point calculations often make use of the single instruction multiple data [63] technique, the SIMD engine has to be mentioned in this context too. Since the Muen SK does not use either component [64], there can't be derived any further requirements in this topic area.

2.7 SPARK

While Ada is a general-purpose language supporting the usual features of modern programming languages including built-in support for the design-by-contract paradigm, SPARK is a specialized well-defined subset of Ada designed for the development of high integrity software. Due to these restrictions of the Ada programming language, SPARK has the ability to simplify the application of formal mathematical methods, so that the correctness of the software or other program properties can be guaranteed with mathematics-based assurance.

At the beginning of the Muen project, the development of SPARK 2014 was still ongoing, so that the Muen SK was initially written in SPARK 2005[65]. Within the last years, the Muen developers have

[61] [4] Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2010, section 6.2.3, page 309 ff.
[62] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, section 2.3.1.3, page 14.
[63] https://en.wikipedia.org/wiki/SIMD, December 21, 2017
[64] cf. Besprechungsnotiz October 23, 2017, and [2] page 41
[65] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. 2013, chapter 2, section 2.1.3, page 5.

changed the underlying programming language and are now using SPARK 2014 [66]. Since SPARK is a true subset of Ada and compilers ignore the SPARK inherent annotations, every correct SPARK program is a valid Ada program and can therefore be compiled with an existing Ada compiler such as GNAT (part of the GNU compiler collection GCC). Hence, to be able to build the Muen SK, the following requirement has to be fulfilled:

> ***REQ-17:*** *There must exist a native or cross compiler for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a native or cross compiler with freely available software.*

To fulfil the requirement of a small code base, the Muen SK uses the Ada Zero Footprint Runtime[67]. A Zero Footprint Runtime (ZFP) is a downscaled runtime system (RTS) where only a minimum of supporting code is required. As no unnecessary libraries are introduced into the system, this setup is ideal for critical low level programming. Therefore, to be able to run the Muen SK on an processor architecture other than Intel x86, a ZFP for the targeted architecture has to be available.

> ***REQ-18:*** *There must exist a Zero Footprint Runtime for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a Zero Footprint Runtime with freely available software.*

2.8 Derived Requirements

The following table summarizes the above derived requirements for a target architecture to be able to run the Muen SK:

number	requirement	topic
REQ-0	The processor architecture has to support 64 bit data-path widths, integer size and memory address widths as well as to be able to execute 32 bit applications.	basics
REQ-1	The target processor architecture must provide a virtualization extension that is capable of running a Type I hypervisor. This requirement includes the hardware assisted support for an additional privilege level and instructions for a simplified switch between this additional and other privilege level.	basics

Table 2.1: requirement summary part one

[66] cf. section kernel, first statement in https://muen.codelabs.ch/#kernel, December 21, 2017
[67] [2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, Section 4.2, page 41.

number	**requirement**	**topic**
REQ-2	The target processor architecture must provide a virtualization extension that supports an automatic handling of guest exits (i.e. traps) and entries. At least, the target processor architecture must provide a support mechanism to completely save and load all the relevant guest state structures.	basics
REQ-3	The target processor architecture shall provide a minimal set of cache management features and an automatic cache clearing feature in the context of virtualization. At least, the target processor architecture must provide a support mechanism to clear caches manually.	memory
REQ-4	The target processor architecture has to provide a Memory Management Unit that supports: (i) memory virtualization on a per subject basis (one page table per subject), (ii) definition of properties and permissions per page table (read/write access, execute disable, caching behaviour), (iii) checking and enforcement of defined properties and permissions, (iv) different page sizes (i.e. large page support), but at least a 4 KB page size.	memory
REQ-5	The target processor architecture must support hardware assisted second level address translation (SLAT).	memory
REQ-6	A multicore target processor architecture has to provide a mechanism to switch off the multithreading mechanism on a per core basis, if multithreading is supported.	memory
REQ-7	A multicore target processor architecture shall provide a barrier synchronization mechanism. At least it must offer an atomic swapping instruction to support the according spinlock implementation.	memory
REQ-8	A target processor architecture has to provide a mechanism to programmatically handle interruptions.	interruption handling
REQ-9	A target processor architecture has to provide an interruption handling that guarantees the **exclusive** treatment of interrupts by the separation kernel.	interruption handling

Table 2.2: requirement summary part two

number	requirement	topic
REQ-10	A target processor architecture has to provide an enabling and disabling mechanism for (external) interrupts, at least for the execution of the hypervisor code.	interruption handling
REQ-11	A target processor architecture must support a mechanism to enable and disable exceptions and software generated interrupts resulting in an exit of the guest subject.	interruption handling
REQ-12	A target processor architecture shall provide a mechanism to force system management exceptions to lead to an exit of a guest subject.	interruption handling
REQ-13	A target processor architecture must be able to differentiate between exit reasons of a guest system and to handle them as specified per subject.	interruption handling
REQ-14	A target processor architecture should provide a technique to fast process interruptions between cores.	interruption handling
REQ-15	A target processor architecture shall provide a preemptive mechanism on a per subject basis. At least it must provide a timer per core.	timer
REQ-16	A target processor architecture must provide a mechanism to virtualize I/O devices by completely isolating the access to devices and providing support for associated interruption and memory features.	device handling
REQ-17	There must exist a native or cross compiler for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a native or cross compiler with freely available software.	SPARK
REQ-18	There must exist a Zero Footprint Runtime for the SPARK 2014 programming language and the targeted processor architecture. At least, it must be possible to build such a Zero Footprint Runtime with freely available software.	SPARK

Table 2.3: requirement summary part three

3 ARMv8 Architecture

The Advanced RISC Machines ARM architecture denotes a Reduced Instruction Set Computing RISC [1] microprocessor design from ARM Limited. Unlike the popular Intel processors, ARM Limited does not manufacture the processors itself, but grants design licenses to semiconductor manufacturing companies. Compared to Complex Instruction Set Computing CISC [2] architectures, the ARM architecture is characterized by a lower number of transistors and as a result lower costs, improved power consumption and less heat generation. Due to the large number of manufacturers and the advantages of this architecture, ARM processors are the most widely used processors in the embedded area. Almost all smartphones, tablets and industrial controllers today use licensed ARM processors [3].

The success of ARM-based processors has led to a steady development of the architecture. With the ARMv8-A architecture introduced in 2011, ARM Limited has presented the first 64-bit architecture with a virtualization extension applicable for embedded systems. In the following years, the ARMv8-A architecture was continuously improved with the versions ARMv8.1-A, ARMv8.2-A and ARMv8.3-A [4]. These enhancements to the ARM architecture now allow software developers to port the latest applications implemented for Intel and AMD processors to the ARM architecture as well as to meet the requirements in the progress of the Industry 4.0 context by developing more secure software.

Due to the application field of ARM processors and the licensing strategy of ARM Limited, a large number of so called ARM-based System on Chip (SoC) was developed. An ARM-based SoC corresponds to the combination of an ARM processor as CPU together with the GPU and other peripheral devices on a single chip [5]. The distinction between the processor and the other devices on such a chip is essential for software development - while the architecture of the processor is defined and very well documented by the ARM company, the accessibility of the processor to the peripherals and its control is not predetermined by ARM. Hence, there are a variety of different SoC architectures with different accessibility strategies: from processor controlled (Odroid C2 with amlogic S905 SoC [6]) to VideoCore controlled (Raspberry Pi 3 with Broadcom 2837 [7]).

This feasibility study follows a general approach to evaluate the portability of the Muen SK to the ARM architecture. Therefore, this chapter only covers the ARM processor architecture and its capabilities. However, some of the derived requirements from the last chapter are SoC specific and can therefore in this context only be qualified as *IMPLEMENTATION DEFINED*. In the next chapter 4, the Raspberry Pi 3 as the target hardware platform of this study is examined in more detail.

[1] cf. https://en.wikipedia.org/wiki/Reduced_instruction_set_computer, December 21, 2017
[2] cf. https://en.wikipedia.org/wiki/Complex_instruction_set_computer, December 21, 2017
[3] cf. https://www.arm.com and https://en.wikipedia.org/wiki/ARM_architecture, December 21, 2017
[4] cf. https://developer.arm.com/products/architecture/a-profile, December 21, 2017
[5] cf. https://en.wikipedia.org/wiki/System_on_a_chip, December 21, 2017
[6] http://www.hardkernel.com/main/products, December 21, 2017
[7] https://www.raspberrypi.org/products/raspberry-pi-3-model-b, December 21, 2017

3.1 Code Examples

After a thorough review of all the available options to run and verify ARMv8 assembly, it was decided to use the following configuration to test the code snippets mentioned in this chapter:

identifier	**description**	**link**
Method	installation as Virtual Machine	VM Ware
Host Operating System	Debian 64-bit 9.2	Debian Download
Toolchain (Cross)	Linaro aarch64-elf cross compiler	Linaro Release
IDE	DS-5 Community Edition Linux64 28rel0	DS-5 IDE
Debugger	DS-5 Community Edition Debugger (integrated into the DS-5 IDE)	DS-5 Debugger
Simulation	ARMv8-A Foundation Model (integrated into DS-5 IDE)	Fast Models

Details on the installation and configuration of the corresponding tools can be found in the respective evaluation case documentation. However, it should be noted that the version of the DS-5 IDE has changed during this project - therefore, the installation process slightly changed compared to this documents. Due to the limitations of the Community Edition of the DS-5 IDE, the code snippets were tested in a minimal environment derived from the official startup example code and only on one processor.

3.1.1 Code Compilation

Principally, three useful compilers are available for compiling assembly code, i.e. the FASMARM Assembler [8], the ARM Compiler 6 [9] and the assembler of the GCC GNU Compiler Collection [10].

The FASMARM v1.42 assembler is a free and Open Source cross assembler add-on for the FASM flat assembler. At the beginning of this project, this assembler was used exclusively because it is easy to install, to configure and to use. However, the main disadvantage of the assembler is that it does not support the 64-bit ELF DWARF debugging format [11] and therefore the assembled code cannot be executed on the Fast Model Simulation Debugger provided by the ARM DS-5 Community Edition.

The ARM Compiler 6 is the latest C/C++ Compiler toolchain provided by ARM Limited. It can be used as a standalone tool but it also supports the integration of the Compiler toolchain into the DS-5 Development Studio Professional and Ultimate Edition. As this compiler is not freely available, it was not tested during this project.

[8] cf. https://arm.flatassembler.net, December 21, 2017
[9] cf. https://developer.arm.com/products/software-development-tools/compilers/arm-compiler, December 21, 2017
[10] cf. https://gcc.gnu.org, December 21, 2017
[11] ReadMe section 5, first paragraph - „... *For 64-bit code only the binary format is currently supported. ELF64 and PE64 formats have not yet been updated.*“; cf. https://arm.flatassembler.net/ReadMe.txt, December 21, 2017

The third compiler tested for compiling assembly code is the compiler of the GCC Gnu Compiler Collection. The Gnu Compiler Collection is a freely available compiler suit for the programming languages C, C++, Objective-C, Fortran, Ada and Go published under the Gnu Public License GPL. As part of the C compiler suite, an assembler for the ARMv8 AArch64 architecture is delivered too. The advantages and therefore the decisive reason to work with this compiler are the supported languages (including the Ada GNAT toolchain), the excellent documentation, the ability to generate ARM ADS AXD 64-bit compatible formats for code simulation on the ARM Fast Model and the large number of existing cross compiler binaries. The preferred cross compiler for this project is the Linaro AArch64 ELF cross compiler [12].

3.1.2 Code Execution and Debugging

There are basically two possibilities available for an informative debugging: on the one hand, one can debug the code over the JTAG interface directly on the target hardware and, on the other hand, the debugger integrated in the DS-5 IDE on a simulated ARMv8 hardware model, the so called Foundation Model, can be used.

To be able to debug the code under consideration directly on the target platform, a JTAG hardware adapter is needed. The JTAG setup was tested with a Segger J-Link Edu Version 10.1 adapter [13] and the Raspberry Pi 3. Detailed instructions for such a setup can be found in the corresponding Development Environment Setup evaluation case for the programming language C/C++. The disadvantages of a JTAG debugging in the context of this chapter are the complicated and time consuming wiring as well as the exclusive view of the processor as one always has to test the peculiarities of the hardware too.

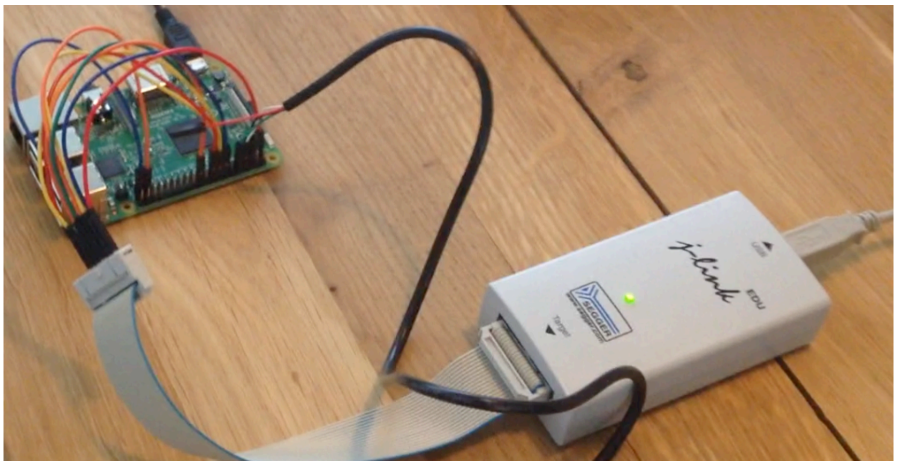
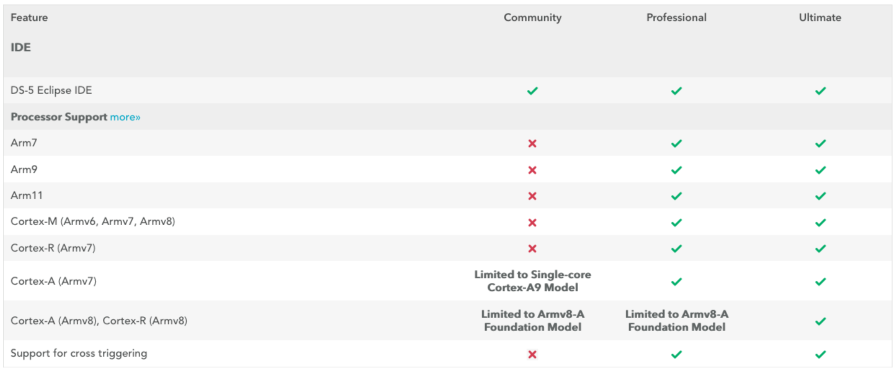


Figure 3.1: JTAG adapter with Raspberry Pi 3

[12] https://www.linaro.org, December 21, 2017
[13] https://www.segger.com/products/debug-probes/j-link/models/j-link-edu, December 21, 2017

The second alternative using the debugger integrated into the DS-5 IDE was really persuasive. Not only the good documentation provided by ARM but also the clear, informative presentation in the IDE as well as the easy handling of the tools convinced to choose this setup. The only disadvantages are the limitations for the freely available community edition - the code can only be debugged on one core, the implementation defined aspects of a SoC cannot be emulated and some restrictions for peripheral devices have to be accepted [14].



Feature	Community	Professional	Ultimate
IDE			
DS-5 Eclipse IDE	✓	✓	✓
Processor Support more»			
Arm7	✗	✓	✓
Arm9	✗	✓	✓
Arm11	✗	✓	✓
Cortex-M (Armv6, Armv7, Armv8)	✗	✓	✓
Cortex-R (Armv7)	✗	✓	✓
Cortex-A (Armv7)	**Limited to Single-core Cortex-A9 Model**	✓	✓
Cortex-A (Armv8), Cortex-R (Armv8)	**Limited to Armv8-A Foundation Model**	**Limited to Armv8-A Foundation Model**	✓
Support for cross triggering	✗	✓	✓

Figure 3.2: DS-5 Community Edition restrictions

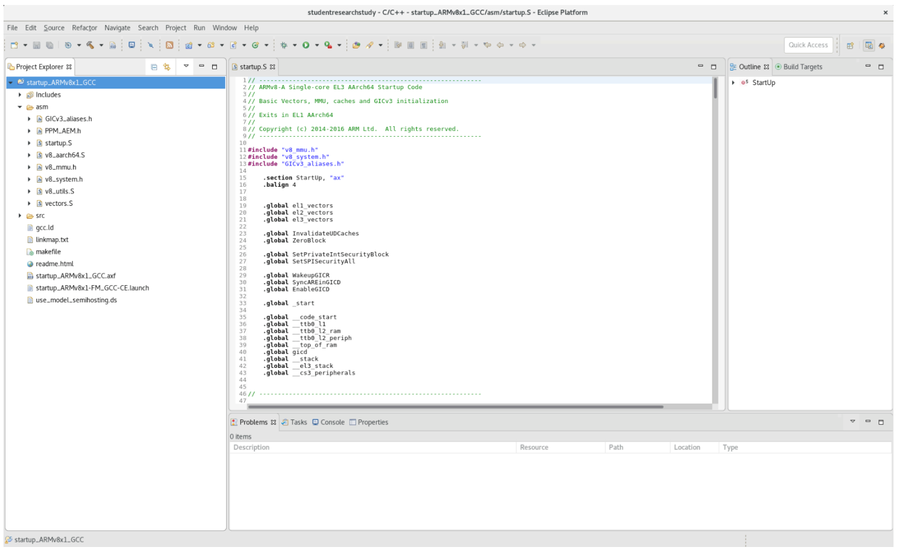
A first good insight into the ARM developer tools can be gained in the videos published on Youtube [15]. In addition to the standard project view (cf. figure 3.3), the debugger view (cf. figure 3.4) is automatically presented during debugging. In the upper left-hand window one can find the debug controls, that show the limitation to only one core. The command window in the middle of the upper half shows the current exception levels and executed commands including line numbers. The most interesting tab „Registers“ in the upper right window shows the general purpose and other registers whereby the currently used registers are shaded in yellow. In addition to the executed code in the lower left corner, the window on the right-hand side contains information about the memory and the stack(s), if defined.

An important note in this context: In order to be able to execute self-written code in the DS-5 debugger, the compiler command line option

```
aarch64-elf-gcc --specs=aem-ve.specs ...
```

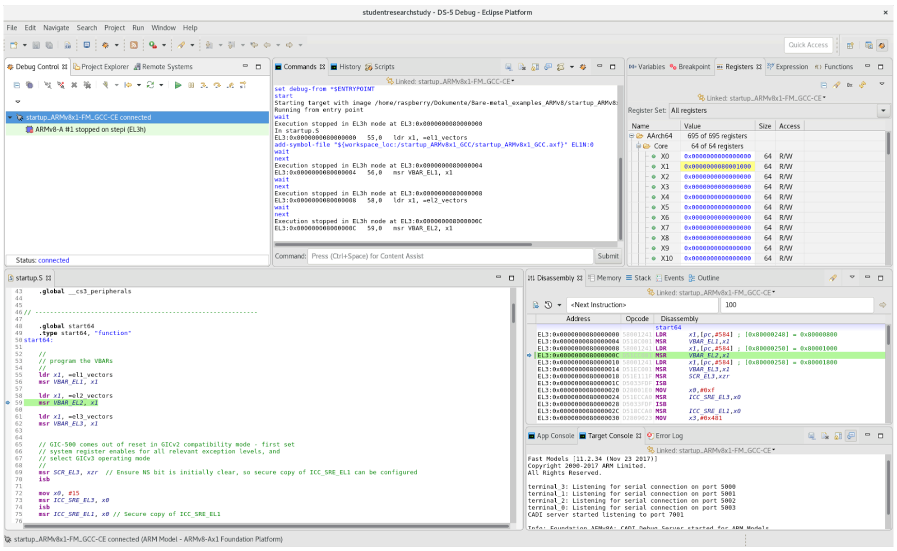
has to be added to load the specification file for the AArch64 baremetal newlib and libgloss appropriate for the foundation model. A good tutorial can be found on the ARM developer page for the DS-5 Community Edition on the tab page „Resources“ [16].

[14] https://developer.arm.com/products/software-development-tools/ds-5-development-studio, December 21, 2017
[15] https://www.youtube.com/watch?v=_tXWrHD8shs, December 21, 2017
[16] https://developer.arm.com/products/software-development-tools/ds-5-development-studio/resources/tutorials/getting-started-with-ds-5-ce-and-armv8-foundation-platform, December 21, 2017



```
// -----------------------------------------------------------
// ARMv8-A Single-core EL3 AArch64 Startup Code
//
// Basic Vectors, MMU, caches and GICv3 initialization
//
// Exits in EL1 AArch64
//
// Copyright (c) 2014-2016 ARM Ltd.  All rights reserved.
// -----------------------------------------------------------

#include "v8_mmu.h"
#include "v8_system.h"
#include "GICv3_aliases.h"

    .section StartUp, "ax"
    .balign 4


    .global el1_vectors
    .global el2_vectors
    .global el3_vectors

    .global InvalidateUDCaches
    .global ZeroBlock

    .global SetPrivateIntSecurityBlock
    .global SetSPISecurityAll

    .global WakeupGICR
    .global SyncAREinGICD
    .global EnableGICD

    .global _start

    .global __code_start
    .global __ttb0_l1
    .global __ttb0_l2_ram
    .global __ttb0_l2_periph
    .global __top_of_ram
    .global gicd
    .global __stack
    .global __el3_stack
    .global __cs3_peripherals


// -----------------------------------------------------------

```

Figure 3.3: DS-5 Community Edition project view



```
set debug-from *$ENTRYPOINT
start
Starting target with image /home/raspberry/Dokumente/Bare-metal_examples_ARMv8/startup_ARMv8
Running from entry point
wait
Execution stopped in EL3h mode at EL3:0x0000000080000000
In startup.S
EL3:0x0000000080000000   55,0   ldr x1, =el1_vectors
add-symbol-file "${workspace_loc:/startup_ARMv8x1_GCC/startup_ARMv8x1_GCC.axf}" EL1N:0
wait
next
Execution stopped in EL3h mode at EL3:0x0000000080000004
EL3:0x0000000080000004   56,0   msr VBAR_EL1, x1
wait
next
Execution stopped in EL3h mode at EL3:0x0000000080000008
EL3:0x0000000080000008   58,0   ldr x1, =el2_vectors
wait
next
Execution stopped in EL3h mode at EL3:0x000000008000000C
EL3:0x000000008000000C   59,0   msr VBAR_EL2, x1
```

Name	Value	Size	Access
AArch64	695 of 695 registers		
Core	64 of 64 registers		
X0	0x0000000000000000	64	R/W
X1	0x0000000080001000	64	R/W
X2	0x0000000000000000	64	R/W
X3	0x0000000000000000	64	R/W
X4	0x0000000000000000	64	R/W
X5	0x0000000000000000	64	R/W
X6	0x0000000000000000	64	R/W
X7	0x0000000000000000	64	R/W
X8	0x0000000000000000	64	R/W
X9	0x0000000000000000	64	R/W
X10	0x0000000000000000	64	R/W

```
    .global __cs3_peripherals


// -----------------------------------------------------------

    .global start64
    .type start64, "function"
start64:

    //
    // program the VBARs
    //
    ldr x1, =el1_vectors
    msr VBAR_EL1, x1

    ldr x1, =el2_vectors
    msr VBAR_EL2, x1

    ldr x1, =el3_vectors
    msr VBAR_EL3, x1


    // GIC-500 comes out of reset in GICv2 compatibility mode - first set
    // system register enables for all relevant exception levels, and
    // select GICv3 operating mode
    //
    msr SCR_EL3, xzr  // Ensure NS bit is initially clear, so secure copy of ICC_SRE_EL1 can be configured
    isb

    mov x0, #15
    msr ICC_SRE_EL3, x0
    isb
    msr ICC_SRE_EL1, x0 // Secure copy of ICC_SRE_EL1

```

```
                                    start64
EL3:0x0000000080000000 58001241 LDR     x1,[pc,#584] ; [0x80000248] = 0x80000800
EL3:0x0000000080000004 D518C001 MSR     VBAR_EL1,x1
EL3:0x0000000080000008 58001241 LDR     x1,[pc,#584] ; [0x80000250] = 0x80001000
EL3:0x000000008000000C D51CC001 MSR     VBAR_EL2,x1
EL3:0x0000000080000010 58001241 LDR     x1,[pc,#584] ; [0x80000258] = 0x80001800
EL3:0x0000000080000014 D51EC001 MSR     VBAR_EL3,x1
EL3:0x0000000080000018 D51E111F MSR     SCR_EL3,xzr
EL3:0x000000008000001C D5033FDF ISB
EL3:0x0000000080000020 D28001E0 MOV     x0,#0xf
EL3:0x0000000080000024 D51ECCA0 MSR     ICC_SRE_EL3,x0
EL3:0x0000000080000028 D5033FDF ISB
EL3:0x000000008000002C D518CCA0 MSR     ICC_SRE_EL1,x0
EL3:0x0000000080000030 D2809023 MOV     x3,#0x481
```

```
Fast Models [11.2.34 (Nov 23 2017)]
Copyright 2000-2017 ARM Limited.
All Rights Reserved.

terminal_3: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_0: Listening for serial connection on port 5003
CADI server started listening to port 7001

Info: Foundation_AEMv8A: CADI Debug Server started for ARM Models
```

Figure 3.4: DS-5 Community Edition debug view

Last but not least, ARM Limited provides a fully functional code example for a single-core AArch64 Startup sequence with basic vectors, MMU, caches and GICv3 (cf. section 3.5.1) initialization based on the GCC C/C++ Compiler suite including all the necessary page tables and memory layout definitions. This code example is provided with the installation of the DS-5 Community Edition [17].

[17] https://developer.arm.com/products/.../community-edition, December 21, 2017

3.2 Fundamentals

3.2.1 Exception Levels

Instead of rings used by the Intel architecture (cf. section 2.1), the ARMv8-A architecture refers to privilege levels as Exception Levels. It is important to note, that, unlike on Intel x86 architecture, code execution at a higher Exception Level (i. e. an Exception Level ELn with a larger value for n) has *more* privileges than code execution at a lower one[18]. With these Exception Levels, the ARMv8 architecture provides a logical separation for software execution privileges. Typically, the Exception Levels can be assigned to the following software examples:

- **EL**0 - normal user applications
- **EL**1 - operating system kernel (usually described as privileged level execution)
- **EL**2 - hypervisor software
- **EL**3 - low-level firmware and secure monitor [19]

In addition to the horizontal subdivision into Exception Levels, the ARMv8-A architecture also physically partitions the upper three Exception Levels into the *Normal World* and the *Secure World*. With this separation, an ARMv8-A processor supports a secure and a non-secure state and allows an operating system to run in parallel with a so called trusted operating system [20]. A trusted OS denotes the operating system running in the Secure World and is responsible to provide secure services to the Normal World. Further details on the TrustedZone technology can be found on the official ARM homepage [21]. The following diagram shows the subdivisions as well as the partitions for the AArch64 execution state:

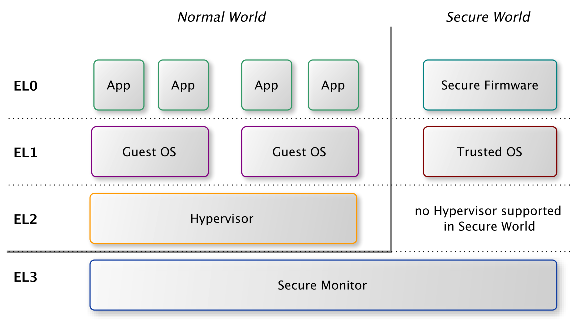


Figure 3.5: ARMv8-A Exception Levels in AArch64

[18][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-1.

[19]ARM Trusted Firmware that takes care of the switching between the non-secure and the secure worlds. The code is available as open source on Github, cf. https://github.com/ARM-software/arm-trusted-firmware, December 21, 2017

[20]The interaction (e.g. access rights) between the secure and non-secure world can be defined using the system monitor and corresponding registers (e.g. for physical address spaces in chapter 12, section 12.9 in [8])

[21]https://www.deepl.com/translator, December 21, 2017

The only differences between the Exception Levels in AArch64 and the AArch32 execution state are, that in the AArch32 execution state there does not exist an Exception Level 2 in the Secure World and that the privilege levels defined for the ARMv7 architecture are mapped to the Exception Levels accordingly. But as the Muen SK needs to be executed in a 64-bit environment, the details of the Exception Level organisation in AArch32 execution state can be omitted.

Even though, the details for changing the Exception Level depend on the execution state of the processor, it can be generally stated that such a change can only take place during the occurrence of an exception (cf. section 3.5), the returning from an exception (i.e. the ERET instruction), a supervisor call or hypervisor call. While changing the Exception Level in AArch32 execution state remains the same as with the ARMv7 architecture[22], for the AArch64 execution state the following rules apply[23][24]:

(i) *Rule 1:* An exception causes a change of program flow by executing an exception handler function from a predefined vector. Exceptions flow from lower Exception Level to higher ones. That means, that an exception cannot be taken to a lower Exception Level (e.g. EL2 to EL1).

(ii) *Rule 2:* Exception Handling at EL0 is not possible, i.e. exceptions must be handled at a higher Exception Level than EL0.

(iii) *Rule 3:* To end an exception handling and return to the previous Exception Level is performed by executing the ERET.

(iv) *Rule 4:* Returning from an exception handler cannot move to higher Exception Levels. Therefore, returning from an exception can stay at the same Exception Level or enter a lower one.

(v) *Rule 5:* The security state changes according to the rules in the section D1.4 of the ARM Architecture Reference Manual [7].

As a practical example, the procedure for changing Exception Level from EL3 to EL1 by using the ERET instruction (returning from an exception) is described in this paragraph [25]. According to the rules mentioned above, the only possibility to switch to a lower Exception Level is to execute the ERET instruction. When performing such an exception return (for this example at EL3), the processor restores the state using the system registers ELR_EL3 (i.e. the address to return to) and SPSR_EL3 (i.e. the state to be restored including the targeted Exception Level). These two registers are writeable, thus allowing the desired entry point and state (the Exception Level EL1 for this practical example) to be programmed manually.

[22][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 3, page 3-5 ff.
[23][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 3, page 3-7 f.
[24][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.1, page D1-1776.
[25]Another example for changing the Exception Level from EL3 to EL2 hypervisor mode can be found on the ARM developer pages in this discussion, December 21, 2017

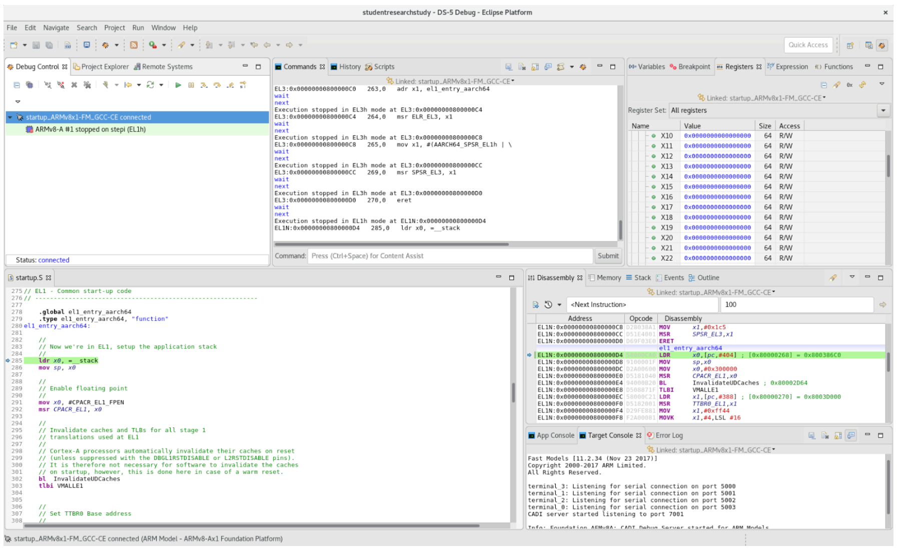
(i) *Step 1:* In this first step, the entry point, i.e. the start address of the code to be executed at EL1, has to be loaded into a general purpose register.

(ii) *Step 2:* The address from step (i) is then stored in the Exception Link Register ELR_ELn of the current Exception Level (i.c. EL3).

(iii) *Step 3:* After that, the Program Status Register at EL3 SPSR_EL3 has to be set accordingly[26]. In the context of the ARMv8 startup code example it has to be noticed, that only a dummy return state for EL1 is loaded. Note, that usually the SPSR_ELn register holds the value of the Programm State PSTATE before taking the exception[27].

(iv) *Step 4:* Finally, the Exception Return instruction ERET has to be executed, using the two registers set in the previous steps for the current Exception Level. When executed, the processor core restores the PSTATE from the SPSR_EL3 register (in this case the Execution Level is set to EL1) and branches to the address held in the ELR_EL3 register[28].

The example can be reproduced with the official startup code of ARM Limited (line 261 to 270) with the debugger included in the DS-5 Community Edition. The result in the debugger view can be found in figure 3.6.


studentresearchstudy - DS-5 Debug - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Quick Access
Debug Control
Project Explorer
Remote Systems
startup_ARMv8x1-FM_GCC-CE connected
ARMv8-A #1 stopped on stepi (EL1h)
Status: connected
Commands
History
Scripts
Linked: startup_ARMv8x1-FM_GCC-CE
Command: Press (Ctrl+Space) for Content Assist
Submit
Variables
Breakpoint
Registers
Expression
Functions
Register Set: All registers
startup.S
Disassembly
Memory
Stack
Events
Outline
<Next Instruction>
100
App Console
Target Console
Error Log
startup_ARMv8x1-FM_GCC-CE connected (ARM Model - ARMv8-Ax1 Foundation Platform)

```
EL3:0x00000000800000C0   263,0   adr x1, el1_entry_aarch64
wait
next
Execution stopped in EL3h mode at EL3:0x00000000800000C4
EL3:0x00000000800000C4   264,0   msr ELR_EL3, x1
wait
next
Execution stopped in EL3h mode at EL3:0x00000000800000C8
EL3:0x00000000800000C8   265,0   mov x1, #(AARCH64_SPSR_EL1h | \
wait
next
Execution stopped in EL3h mode at EL3:0x00000000800000CC
EL3:0x00000000800000CC   269,0   msr SPSR_EL3, x1
wait
next
Execution stopped in EL3h mode at EL3:0x00000000800000D0
EL3:0x00000000800000D0   270,0   eret
wait
next
Execution stopped in EL1h mode at EL1N:0x00000000800000D4
EL1N:0x00000000800000D4   285,0   ldr x0, =__stack
```

Name	Value	Size	Access
X10	0x0000000000000000	64	R/W
X11	0x0000000000000000	64	R/W
X12	0x0000000000000000	64	R/W
X13	0x0000000000000000	64	R/W
X14	0x0000000000000000	64	R/W
X15	0x0000000000000000	64	R/W
X16	0x0000000000000000	64	R/W
X17	0x0000000000000000	64	R/W
X18	0x0000000000000000	64	R/W
X19	0x0000000000000000	64	R/W
X20	0x0000000000000000	64	R/W
X21	0x0000000000000000	64	R/W
X22	0x0000000000000000	64	R/W

```
// EL1 - Common start-up code
// ------------------------------------------------------------

    .global el1_entry_aarch64
    .type el1_entry_aarch64, "function"
el1_entry_aarch64:

    //
    // Now we're in EL1, setup the application stack
    //
    ldr x0, =__stack
    mov sp, x0

    //
    // Enable floating point
    //
    mov x0, #CPACR_EL1_FPEN
    msr CPACR_EL1, x0

    //
    // Invalidate caches and TLBs for all stage 1
    // translations used at EL1
    //
    // Cortex-A processors automatically invalidate their caches on reset
    // (unless suppressed with the DBGL1RSTDISABLE or L2RSTDISABLE pins).
    // It is therefore not necessary for software to invalidate the caches
    // on startup, however, this is done here in case of a warm reset.
    bl  InvalidateUDCaches
    tlbi VMALLE1


    //
    // Set TTBR0 Base address
    //
```

```
Address                     Opcode    Disassembly
EL1N:0x00000000800000C8     D28038A1  MOV      x1,#0x1c5
EL1N:0x00000000800000CC     D51E4001  MSR      SPSR_EL3,x1
EL1N:0x00000000800000D0     D69F03E0  ERET
                                      el1_entry_aarch64
EL1N:0x00000000800000D4     58000CA0  LDR      x0,[pc,#404] ; [0x80000268] = 0x800386C0
EL1N:0x00000000800000D8     9100001F  MOV      sp,x0
EL1N:0x00000000800000DC     D2A00600  MOV      x0,#0x300000
EL1N:0x00000000800000E0     D5181040  MSR      CPACR_EL1,x0
EL1N:0x00000000800000E4     94000B20  BL       InvalidateUDCaches ; 0x80002D64
EL1N:0x00000000800000E8     D508871F  TLBI     VMALLE1
EL1N:0x00000000800000EC     58000C21  LDR      x1,[pc,#388] ; [0x80000270] = 0x8003D000
EL1N:0x00000000800000F0     D5182001  MSR      TTBR0_EL1,x1
EL1N:0x00000000800000F4     D29FE881  MOV      x1,#0xff44
EL1N:0x00000000800000F8     F2A00081  MOVK     x1,#4,LSL #16
```

```
Fast Models [11.2.34 (Nov 23 2017)]
Copyright 2000-2017 ARM Limited.
All Rights Reserved.

terminal_3: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_0: Listening for serial connection on port 5003
CADI server started listening to port 7001

Info: Foundation AEMv8A: CADI Debug Server started for ARM Models
```

Figure 3.6: ARMv8-A Exception Level Switch debugger view

[26][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter C, section C5.2.20, page C5-385.

[27][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.7, page D1-1791.

[28][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter C, section C6.2.71, page C6-622.

3.2.2 Execution States

The ARMv8 architecture defines two Execution States, i.e. the AArch64 Execution State using 64-bit wide and the AArch32 Execution State using 32-bit wide general purpose registers. While the instruction set and the privilege level mapping in the AArch32 Execution State stays the same as in the ARMv7 architecture, the AArch64 Execution State is organised as shown in figure 3.5 and has a different instruction set A64.

Changing between Execution States on the same level is not possible. That means, the system has to first switch to the higher exception level as shown in the previous section, then perform the requested change of the upper exception level and switch back to the original exception level. Of course, such a change between the Execution States has to meet some rules - the most important one is, that changing to AArch64 Execution State requires switching from a lower exception level to a higher one. The following figure 3.7 summarises this rules stated in the ARM Programmer's Guide[29]:

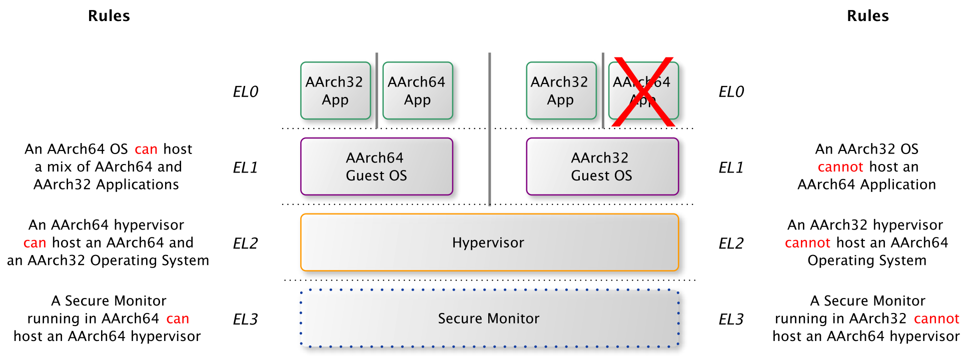

Rules
ELO
AArch32 App
AArch64 App
AArch32 App
AArch64 App
ELO
Rules
An AArch64 OS can host a mix of AArch64 and AArch32 Applications
EL1
AArch64 Guest OS
AArch32 Guest OS
EL1
An AArch32 OS cannot host an AArch64 Application
An AArch64 hypervisor can host an AArch64 and an AArch32 Operating System
EL2
Hypervisor
EL2
An AArch32 hypervisor cannot host an AArch64 Operating System
A Secure Monitor running in AArch64 can host an AArch64 hypervisor
EL3
Secure Monitor
EL3
A Secure Monitor running in AArch32 cannot host an AArch64 hypervisor

Figure 3.7: ARMv8-A Execution States rules

An example of a correct Execution State change would be an application running in a 32-bit Execution State at EL0 on a 64-bit Operating System executing at EL1 and a second application, that needs to be executed in a 64-bit execution state at EL0, on the same Operating System. In such a case, the 32-bit application can change to the OS exception level in AArch64 execution state by calling the Supervisor Call instruction or by receiving an interrupt. Then the OS can change the execution state of the exception level EL0 to AArch64 and switch back to EL0.

The two most important limitations in the context of Execution States are that it is not possible to check the execution state of the actual code running on a specific exception level but only for higher exception levels [30] and that code running at EL3 cannot take an exception to a higher exception level. Therefore, code executing at EL3 cannot change its execution state, except by going through a reset[31].

[29][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-8 f.
[30]cf. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka16146.html, December 21, 2017
[31][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 3, page 3-9.

3.2.3 Startup and Reset

The ARM documentation refers to the startup, i.e. powering on the CPU, as cold reset[32]. A processor based on the ARMv8 architecture always starts execution at the highest exception level, provided that the SoC manufacturer does not apply any additional firmware code to the boot process. In contrast, the Execution State, in which a processor is running immediately after powering it up, is *IMPLEMENTATION DEFINED* [33]. This means that the SoC manufacturer defines this explicitly with a hardware based signal being either logic zero or logic one as an input to the corresponding AA64nAA32 pin of the processor .

As already mentioned, code executing at EL3 can only change its execution state by going through a so called warm reset[34]. Every core has its own reset input and executes the according exception immediately after their reset. In addition, this exception cannot be masked[35]. While the execution state after a warm reset is software defined by setting the AA64 bit in the RMR_EL3 register, the reset vector for the highest Exception Level (i.e. the location of the instruction that the ARM processor jumps to when an exception is raised) is again *IMPLEMENTATION DEFINED*[36].

Further details on resetting an ARMv8 processor can be found in the ARM Architecture Reference Manual[37] as well as in the processors Technical Reference Manuals[38].

The first requirement (cf. REQ-0 in section 2) for porting the Muen SK is that the target processor architecture supports a 64-bit execution state. According to the previously explained mechanism, the following qualification can be stated:

> ***REQ-0 - IMPLEMENTATION DEFINED:*** *The ARMv8 architecture principally supports a 64-bit execution mode. But as the initial execution state as well as the reset vector are defined by the manufacturer of the specific SoC, the fulfilment of this requirement can only be qualified on the basis of the target hardware.*

[32][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.9, page D1-1795.

[33]https://community.arm.com/processors/f/discussions/2874/aarch32-in-armv8 and http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka16239.html, December 21, 2017

[34][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.9, page D1-1795.

[35][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 10, page 10-2.

[36][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D7.2.85, page D1-2448.

[37][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.9.2, page D1-1797 f., provides for example a code sequence to request a warm reset, followed by a pseudocode description in section D1.9.3.

[38][9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual.* 2016, chapter 2, section 2.3.3, page 2-14, and chapter 4, section 4.3.76, page 4-114 as well as appendix A.3.

3.3 Virtualization Basics

Unlike Intel's VT technology, the ARMv8-A Virtualization Extension consists of a number of additional extensions to existing ARM architecture technologies. Accordingly, only one of the ARM documents mentioned above contains a short section dedicated to virtualization[39]. However, the ARM Developer Community provides a summarising document on virtualization[40].

Two of the main features of the ARMv8-A Virtualization Extension are a dedicated Exception Level EL2 for the hypervisor code (cf. section 3.2.1), support for trapping exceptions that change the core context or state and an additional exception type generated by the Hypervisor Call instruction HVC with a 16-bit payload targeting the exception level EL2[41]. These features are explicitly intended for the implementation of a type I hypervisor[42].

> ***REQ-1 - FULFILLED:*** *The ARMv8-A architecture explicitly provides the demanded mechanisms to run a type I hypervisor.*

With respect to the second requirement of the Muen SK in the area of virtualization (cf. section 2.1), it can in advance be stated that the ARM virtualization technology does not support any automatic storing or loading of the guest's state. On the contrary, the hypervisor code has to load both its and the guest's context completely into memory or from memory respectively when performing a context switch. At least, the ARMv8 architecture supports a performance optimized possibility for handling the corresponding registers with the Store and Load Pair instructions[43]. Depending on the guest system, the hypervisor and the specific processor type (e.g. ARMv8 Cortex-A53) as well as the current execution state and the exception level, the following registers could belong to the context and have to be treated accordingly:

- *System Registers:* This category of registers includes different counter, physical timer, MMU, second level address translation and cache registers as well as the Saved Program Status Register SPSR_ELn. An overview can be found in the ARM Programmer's Guide[44].
- *Special Purpose Registers:* The two most important registers of this category are the Stack Pointer Register SP_ELn and the special exception return registers. A list of any registers to be stored can be found in the ARM Architecture Reference Manuel[45].

[39][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter D, section D1.5, page D1-1782.
[40][6] n.a. *AArch64 Virtualization*. 2017.
[41][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 10, page 10-2.
[42][6] n.a. *AArch64 Virtualization*. 2017, chapter 1, page 4.
[43][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter C, section C3.2.2 f., page C3-161 ff.
[44][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 4, section 4.3, page 4-7 ff.
[45][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, chapter C, section C5.2, page C5-336 ff.

- *General Purpose Registers:* Even though there exist some guidelines for the usage of the general purpose registers, the hypervisor has to store and load all these register to ensure a complete handling of the guest's context. As the Muen SK also supports Virtual Machines executing in a 32-bit environment, it is important to store and load the banked registers too. Banked registers are special purpose registers for exceptions in the AArch32 execution state that are stored in the upper general purpose registers to reduce latency for exception handling[46].

- *Floating Point and NEON Registers:* If enabled, the SIMD and floating point registers have to be stored and loaded as well[47].

- *General Interrupt Registers:* If supported by the SoC and enabled by the hypervisor, the according GICD registers have to be considered. Therefore, all pending and active states of private interrupts on the core have to be handled too.

- *Generic and Virtual Timer Registers:* In the case of guests using virtual timers, the timer registers must be saved and restored so that they generate interrupts at the expected intervals.

The physical memory, that is assigned to a guest, does not have to be handled. By using more than one stage of memory translation, the physical memory that the guest uses stays private and distinct from any others.

To get an impression of how the storing and loading of general purpose registers and system registers could look like, the following code snippet presents two examples:

```
; storing and loading the two general purpose registers x0 and x1 in AArch64 execution state
        stp     x0, x1, [memory_address]
        ...
        ldp     x0, x1, [memory_address]

; storing and loading system registers for exception Level 1
        mrs     x2, ESR_EL1
        mrs     x3, ELR_EL1
        stp     x2, x3, [memory_address]
        ...
        ldp     x02 x3, [memory_address]
        msr     ESR_EL1, x2
        msr     ELR_EL1, x3
```

Taking the above explanations into account, the following can be stated with respect to the requirement demanded by Muen SK:

> ***REQ-2 - FULFILLED:*** *Even though a context switch has to be implemented manually in the hypervisor code, it is possible to save and restore all the required registers of a guests context. Therefore, this requirement is qualified as fulfilled by the ARMv8-A architecture.*

[46][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 4, section 4.5.1, page 4-13 ff.
[47][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 4, section 4.6, page 4-17.

3.4 Memory

Normally, processors implementing the ARMv8-A architecture have two or more levels of cache. These are usually organized in such a way that one Level 1 cache per core with different areas for instructions and data is available, one unified level 2 cache is shared by two or more cores and an external level 3 cache is used by the entire cluster. The Main Memory can be accessed over the internal bus[48].

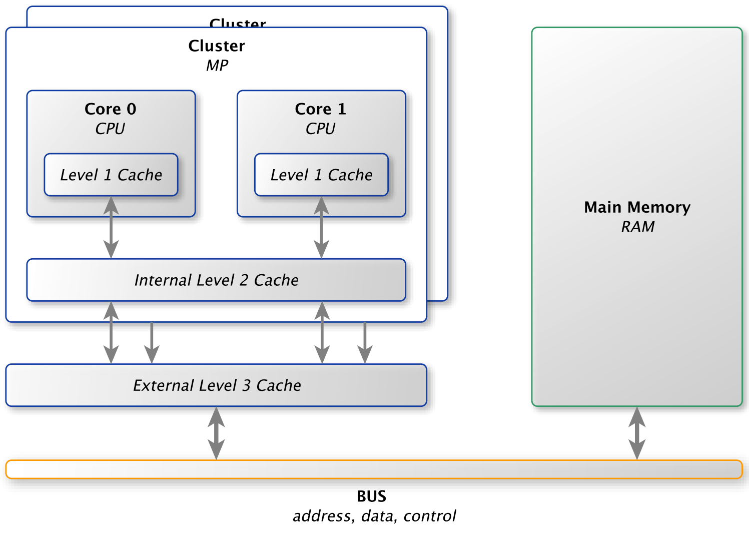


Figure 3.8: ARMv8-A standard memory organisation

3.4.1 Caches

The concrete implementation of the caching structures is not defined in more detail by the ARMv8-A architecture. The only requirement in this context is that the level 1 cache must always be designed as a set of associative caches. This type of cache divides the corresponding memory area into a certain number of equally-sized pieces, called ways. The number of such ways depends on the specific processor architecture - e.g. the ARMv8 Cortex-A53 uses a 2-way set associative instruction cache. Also not defined for the Level 1 cache is the cache addressing mode, i. e. whether a virtual address is first converted into a physical address and then a cash lookup is performed (Physically Indexed Physically Tagged PIPT) or whether the virtual address and the cache lookup are performed in parallel and finally the correctness of the found cache entry is checked against the physical address (Virtually Indexed Physically Tagged VIPT) [49]. To continue the example of the last paragraph, the Cortex-A53 MPCore instruction cache (level 1) uses Virtually Indexed Physically Tagged (VIPT) addressing mode[50].

[48][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 11, page 11-1 ff.
[49]https://www.youtube.com/watch?v=3sX5obQCHNA, December 21, 2017
[50][9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual.* 2016, chapter 2, section 2.1.1, page 2-2.

The organisation and structure of the remaining cache levels is left to the manufacturers of the respective SoC. However, the ARMv8-A architecture establishes some rules in the form of policies and specifies a minimum set of cache maintenance functions for the level 1 cache:

- *Cache Policies:* There exist two categories of policies for caching structures, the allocation and the update policies. The allocation policies are Write Allocation (WA), i.e. a cache line is allocated on a write miss, and Read Allocation (RA), i.e. a cache line is allocated on a read miss. The update policies consist of the Write Back (WB), i.e. a write updates the cache only and marks the cache line as dirty, and the Write Trough (WT), i.e. a write updates both the cache and the external memory system. Additionally, the ARMv8-A architecture provides some preload hint instruction. If a cache structure supports one of this features is *IMPLEMENTATION DEFINED* by the manufacturer. But in contrast to other implementation defined aspects of the caches, the support of this features must be set in the Cache Size ID Register of the processor[51].

- *Cache Maintenance:* The ARMv8-A architecture demands three different ways to clean or invalidate the level 1 cache - *(a)* invalidation of a cache or cache line, i.e. to clear it of data by clearing the valid bit; *(b)* cleaning a cache or cache line, i.e. writing the contents of cache lines, that are marked as dirty, out to the next level of cache or to main memory and clearing the dirty bits in the cache line; *(c)* zeroing, i.e. zero a block of memory within the cache (only for data cache). All three operations must either be applicable to the entire cache (mandatory for instruction cache only) or can be applied based on a virtual address, a set index or a way number. In addition to a list of all operations, the ARM Programmer's Guide also contains some code examples for cache handling[52]. It should be noted that after the corresponding cache operations, a data or instruction synchronisation barrier always has to be called to apply the cache operations that are otherwise executed in any relative order.

According to the explanations above and compared to Intel's x86 cache management, the following qualification can be stated:

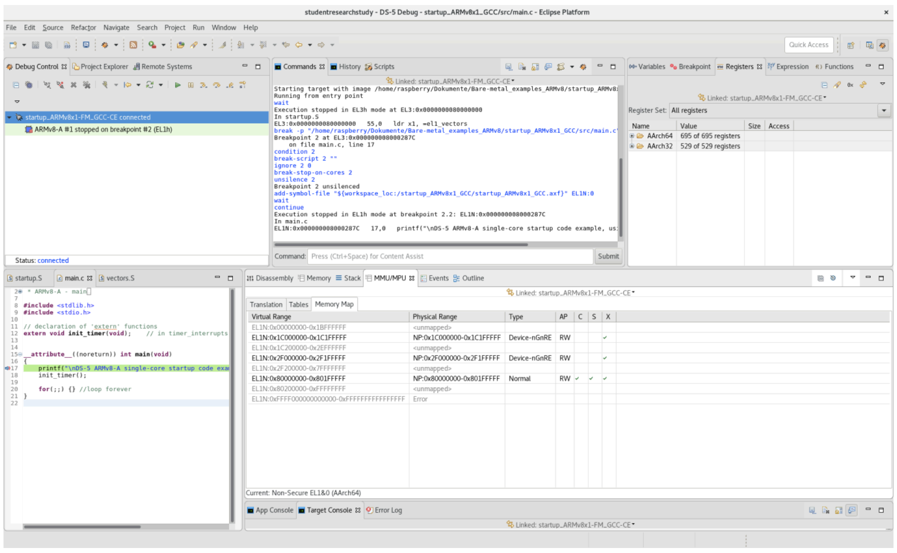
> ***REQ-3 - FULFILLED:*** *Since the cache maintenance of the Intel architecture seems to be quite similar to the one of the ARMv8-A architecture and in particular a cache invalidation can explicitly be performed, this requirement has to be seen as fulfilled by the ARMv8-A architecture.*

[51] [9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual.* 2016, chapter 4, section 4.3.22, page 4-42 f.
[52] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 11, section 11.5, page 11-13 ff.

3.4.2 Memory Management

The ARMv8-A architecture provides one Memory Management Unit (MMU) per core. In addition to the transparent translation of virtual addresses, the MMU also controls and enforces memory access permissions, memory ordering and cache policies for each memory region. Every Exception Level EL3 to EL1 has its own virtual address space[53].

The official startup code of ARM Limited also provides some code for setting up the MMU and the translation tables. In the DS-5 debugger, the code can either be traced step by step or the MMU setup result can be viewed directly in the debugger's MMU view. For the second alternative, the debugger settings must first be adjusted in order to be able to start debugging directly in the main method. To do so, the debugger must be switched to debug from symbol main in the Debug Configurations. As soon as the debugger stops at the corresponding breakpoint, the MMU view can be opened with Windows $\rightarrow$ Show View $\rightarrow$ MMU. This view contains a top-level view of the virtual memory layout (cf. figure 3.9) as well as the associated translation tables (cf. figure 3.10).


studentresearchstudy - DS-5 Debug - startup_ARMv8x1_GCC/src/main.c - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug Control
Project Explorer
Remote Systems
startup_ARMv8x1-FM_GCC-CE connected
ARMv8-A #1 stopped on breakpoint #2 (EL1h)
Status: connected
Commands
History
Scripts
Linked: startup_ARMv8x1-FM_GCC-CE
Variables
Breakpoint
Registers
Expression
Functions
Register Set: All registers
Name
Value
Size
Access
AArch64 695 of 695 registers
AArch32 529 of 529 registers
Command: Press (Ctrl+Space) for Content Assist
Submit
startup.S
main.c
vectors.S
Disassembly
Memory
Stack
MMU/MPU
Events
Outline
Translation
Tables
Memory Map
App Console
Target Console
Error Log

```
Starting target with image /home/raspberry/Dokumente/Bare-metal_examples_ARMv8/startup_ARMv8
Running from entry point
wait
Execution stopped in EL3h mode at EL3:0x0000000080000000
In startup.S
EL3:0x0000000080000000   55,0   ldr x1, =el1_vectors
break -p "/home/raspberry/Dokumente/Bare-metal_examples_ARMv8/startup_ARMv8x1_GCC/src/main.c"
Breakpoint 2 at EL3:0x000000008000287C
    on file main.c, line 17
condition 2
break-script 2 ""
ignore 2 0
break-stop-on-cores 2
unsilence 2
Breakpoint 2 unsilenced
add-symbol-file "${workspace_loc:/startup_ARMv8x1_GCC/startup_ARMv8x1_GCC.axf}" EL1N:0
wait
continue
Execution stopped in EL1h mode at breakpoint 2.2: EL1N:0x000000008000287C
In main.c
EL1N:0x000000008000287C   17,0   printf("\nDS-5 ARMv8-A single-core startup code example, us
```

```c
 * ARMv8-A - main

#include <stdlib.h>
#include <stdio.h>

// declaration of 'extern' functions
extern void init_timer(void);    // in timer_interrupts


__attribute__((noreturn)) int main(void)
{
    printf("\nDS-5 ARMv8-A single-core startup code exa
    init_timer();

    for(;;) {} //loop forever
}

```

Virtual Range	Physical Range	Type	AP	C	S	X
EL1N:0x00000000-0x1BFFFFFF	<unmapped>					
EL1N:0x1C000000-0x1C1FFFFF	NP:0x1C000000-0x1C1FFFFF	Device-nGnRE	RW			✓
EL1N:0x1C200000-0x2EFFFFFF	<unmapped>					
EL1N:0x2F000000-0x2F1FFFFF	NP:0x2F000000-0x2F1FFFFF	Device-nGnRE	RW			✓
EL1N:0x2F200000-0x7FFFFFFF	<unmapped>					
EL1N:0x80000000-0x801FFFFF	NP:0x80000000-0x801FFFFF	Normal	RW	✓	✓	✓
EL1N:0x80200000-0xFFFFFFFF	<unmapped>					
EL1N:0xFFFF000000000000-0xFFFFFFFFFFFFFFFF	Error					

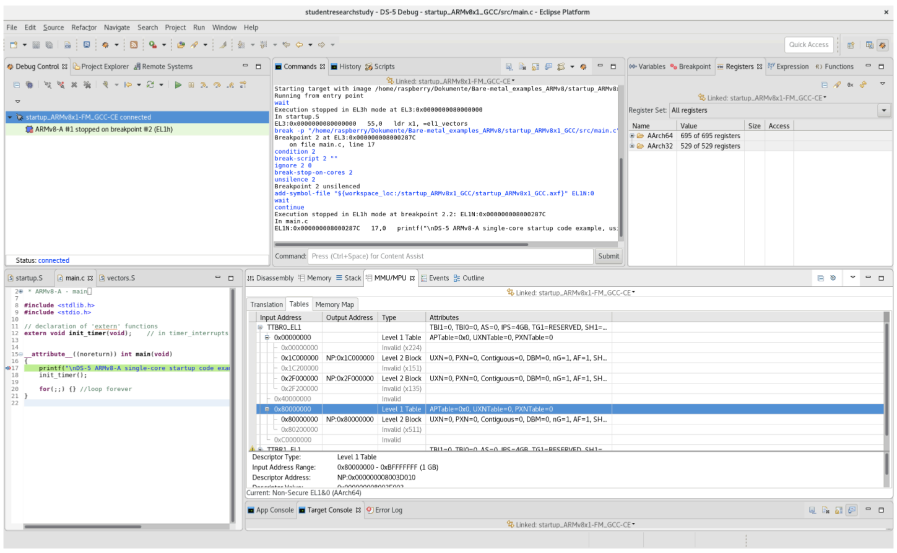
Current: Non-Secure EL1&0 (AArch64)

Figure 3.9: DS-5 Debugger MMU memory map

As already mentioned, the support for cache policies is implementation defined. If a SoC provides this feature the according attributes can be set in the translation table entries as defined in the Memory Attribute Indirection Register MAIR. In contrast to caching, the access permissions controlled through the translation table entries are enforced by the MMU. The access permissions can therefore be set separately on a per exception level basis. The ARMv8-A architecture defines three different types of access permissions - readable, writeable and executable. All possible combinations for a specific

[53][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, page 12-1.

exception level are listed in the ARMv8 Programmer's Guide[54] and the details on the registers, that have to be set accordingly, can be found in the ARMv8 Architecture Reference Manual[55].



startup_ARMv8x1-FM_GCC-CE connected
ARMv8-A #1 stopped on breakpoint #2 (EL1h)

Status: connected

Linked: startup_ARMv8x1-FM_GCC-CE

```
Starting target with image /home/raspberry/Dokumente/Bare-metal_examples_ARMv8/startup_ARMv8
Running from entry point
wait
Execution stopped in EL3h mode at EL3:0x0000000080000000
In startup.S
EL3:0x0000000080000000   55,0   ldr x1, =el1_vectors
break -p "/home/raspberry/Dokumente/Bare-metal_examples_ARMv8/startup_ARMv8x1_GCC/src/main.c"
Breakpoint 2 at EL3:0x000000008000287C
    on file main.c, line 17
condition 2
break-script 2 ""
ignore 2 0
break-stop-on-cores 2
unsilence 2
Breakpoint 2 unsilenced
add-symbol-file "${workspace_loc:/startup_ARMv8x1_GCC/startup_ARMv8x1_GCC.axf}" EL1N:0
wait
continue
Execution stopped in EL1h mode at breakpoint 2.2: EL1N:0x000000008000287C
In main.c
EL1N:0x000000008000287C   17,0   printf("\nDS-5 ARMv8-A single-core startup code example, us
```

Command: Press (Ctrl+Space) for Content Assist

Register Set: All registers

Name	Value	Size	Access
AArch64	695 of 695 registers		
AArch32	529 of 529 registers		

```c
 * ARMv8-A - main

#include <stdlib.h>
#include <stdio.h>

// declaration of 'extern' functions
extern void init_timer(void);    // in timer_interrupts


__attribute__((noreturn)) int main(void)
{
    printf("\nDS-5 ARMv8-A single-core startup code exa
    init_timer();

    for(;;) {} //loop forever
}
```

Input Address	Output Address	Type	Attributes
TTBR0_EL1			TBI1=0, TBI0=0, AS=0, IPS=4GB, TG1=RESERVED, SH1=...
0x00000000		Level 1 Table	APTable=0x0, UXNTable=0, PXNTable=0
0x00000000		Invalid (x224)	
0x1C000000	NP:0x1C000000	Level 2 Block	UXN=0, PXN=0, Contiguous=0, DBM=0, nG=1, AF=1, SH...
0x1C200000		Invalid (x151)	
0x2F000000	NP:0x2F000000	Level 2 Block	UXN=0, PXN=0, Contiguous=0, DBM=0, nG=1, AF=1, SH...
0x2F200000		Invalid (x135)	
0x40000000		Invalid	
0x80000000		Level 1 Table	APTable=0x0, UXNTable=0, PXNTable=0
0x80000000	NP:0x80000000	Level 2 Block	UXN=0, PXN=0, Contiguous=0, DBM=0, nG=1, AF=1, SH...
0x80200000		Invalid (x511)	
0xC0000000		Invalid	
TTBR1_EL1			TBI1=0, TBI0=0, AS=0, IPS=4GB, TG1=RESERVED, SH1=...

Descriptor Type: Level 1 Table
Input Address Range: 0x80000000 - 0xBFFFFFFF (1 GB)
Descriptor Address: NP:0x000000008003D010

Current: Non-Secure EL1&0 (AArch64)

Figure 3.10: DS-5 Debugger MMU translation tables

The ARMv8-A architecture supports two different translation table formats for the AArch32 execution state, i.e. a long descriptor format with Large Physical Address Extension (LPAE) and a short descriptor format. In the AArch64 execution state, however, only the long descriptor format is available, that allows addressing with up to 48-bits. The remaining bits 63:48 of the 64-bit virtual address are used for selecting one of two registers containing the base address of the translation table and optionally the upper 8-bits can be used for tagging the virtual address [56]. The ARMv8-A architecture supports up to three levels of translation tables with granule sizes of 4KB, 16KB and 64KB. It is implementation defined, which of the three sizes actually are supported by a processor. However, processors of the Cortex-A53 series must support all three formats. The addressable memory areas and sizes resulting from the different combinations of page size and translation level can be found in the list provided by the ARMv8 Programmer's Guide[57].

The Translation Lookaside Buffer (TLB) is used as a cache of recently accessed page translations (cf. section 2.2.2). However, a ARMv8-A TLB can not only store and look up physical and virtual addresses, but is also able to handle attributes such as memory types, cache policies and access

[54] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.7, page 12-23 f.

[55] [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. 2017, e.g. executable regions at EL0 and EL1 in chapter D, section D7.2.88, page D7-2456 ff.

[56] The ARMv8-A architecture does not specify or mandate a specific use case for tagged addressing. A use case example can be found in chapter 12, section 12.5.1, page 12-18, of the ARMv8-A Programmer's Guide [8]

[57] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.4, page 12-14 ff.

rights. In addition and in the context of virtualization, the TLB also stores the Address Space ID (ASID) and the Virtual Machine ID (VMID). Enabling and disabling as well as a minimal TLB maintenance are also supported. This means that TLB entries can be invalidated using the VMID, the virtual address or a specific exception level. The ARMv8 Programmer's Guide provides some code examples for the TLB maintenance[58] and details for the ARMv8-A processor in the Technical Reference Manual[59].

According to the above explanations, the requirements in the context of memory management stated by the Muen SK can be judged as follows:

> ***REQ-4 - FULFILLED:*** *The ARMv8 architecture provides a Memory Management Unit per core with the following features:*
>
> *(i) With the possibilities of setting up translation tables on a per exception level basis and defining the according base addresses in different registers, the ARMv8 architecture meets this requirement.*
>
> *(ii) As the ARMv8 architecture provides access permissions controlled through the translation table entries, this requirement can also be rated as fulfilled.*
>
> *(iii) The MMU provided with the ARMv8 architecture has to enforce the access permissions and hence must also be able to check them. This requirement can therefore be qualified as fulfilled.*
>
> *(iv) Even though the supported page sizes are implementation defined by the processor specification, all of the ARMv8-A processor series support at least the 4KB sizes. Therefore, this requirement is met too.*

3.4.3 Advanced Memory Virtualization

To be able to run complex virtual machines, the Muen SK relies on the Second Level address translation provided by Intel's EPT technology. The ARMv8-A Virtualization Extension explicitly provides a similar mechanism for nested page tables to isolate the guest operating systems[60].

Using the ARMv8 Virtualization Extension, the hypervisor is responsible for both its own memory management and that of the guest OS. In a first step, the MMU of the exception level EL2 with the corresponding hypervisor vector tables has to be configured to translate the virtual addresses of the hypervisor correctly. In a second step, the hypervisor must set up and manage the second level address translation mechanism for each virtual machine by enabling the ARMv8-A SLAT mechanism and setting up the corresponding translation tables[61]. A correctly applied SLAT then translates the interme-

[58] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.1, page 12-5.
[59] 9, e.g. chapter 4, section 4.2.6, page 4-7 f.
[60] [6] n.a. *AArch64 Virtualization*. 2017, chapter 1, page 5.
[61] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 12, section 12.6, page 12-20.

diate physical memory addresses of the VM to physical memory addresses. The exception handling of aborts during SLAT address translations has to be done by the hypervisor on exception level EL2.

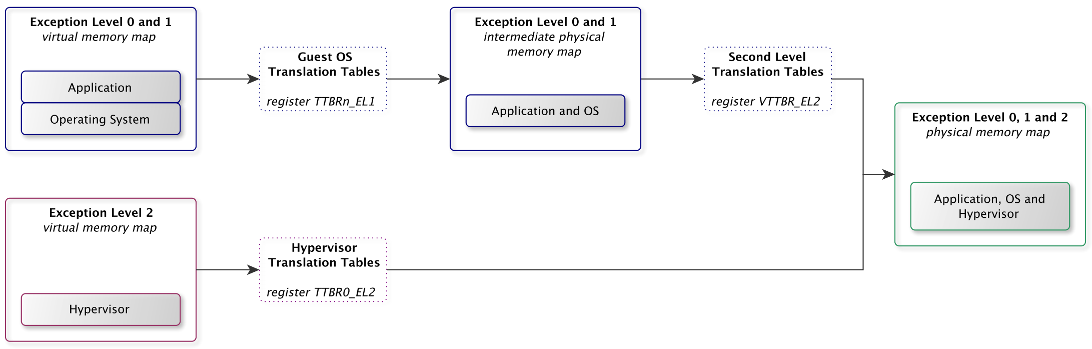


Figure 3.11: ARMv8-A Second Level Address Translation

Code examples for enabling the ARMv8 Second Level Address Translation can be found in the AArch64 Virtualization documentation provided by ARM Limited[62]. Additionally, two practical examples of the usage of nested page tables in the context of a separation kernel can be found in the Phidias hypervisor code [63] written by Jan Nordholz[64] and the HASPOC source code [65] by Vinnova [66]. Therefore, the following can be stated:

> ***REQ-5 - FULFILLED:*** *The ARMv8-A architecture provides a Second Level Address Translation mechanism and hence meets this requirement.*

3.4.4 Multicore Environment

Neither the ARMv8 Cortex-A57 nor Cortex-A53 are Simultaneous Multithreading (SMT) microarchitectures, so at any time there is only one thread executing on one core. As there is not any multithreading support for all currently used processors of the ARMv8-A architecture, the following requirement is always fulfilled.

> ***REQ-6 - FULFILLED:*** *fulfilled per definition*

To synchronise the execution in multicore environment, the Muen SK implements a barrier realized with a spinlock using the atomic XCHG processor swapping instruction. The ARMv8-A architecture provides

[62] [6] n.a. *AArch64 Virtualization*. 2017, chapter 2, section 2.1, page 7 f.
[63] http://phidias-hypervisor.de/repos/core.git, December 21, 2017
[64] [15] Nordholz. *Design and Provability of a Statically Configurable Hypervisor*. 2017, chapter 4, section 4.5, page 30.
[65] https://haspoc.sics.se/source.html, December 21, 2017
[66] https://www.vinnova.se/en, December 21, 2017

some synchronisation primitives that can be used to implement such a barrier[67]. As an example, the Phidias hypervisor implements a spinlock in its assembler file lock.S [68].

> ***REQ-7 - FULFILLED:*** *The ARMv8 architecture provides different synchronisation primitives to fulfil this requirement.*

3.5 Exception Handling

As already mentioned in the last chapter 2, the various processor architectures and the corresponding literature use different terms (e. g. exception, interrupt, signal, event) for the temporary interruption of a running process by an interruption cause. In the ARM terminology, such an interruption is referred to as an exception. The ARM documentation defines an exception as a condition or system event that requires some action by privileged software (i.e. an exception handler) to ensure the continuous functioning of the system and differentiates between the following four types of exceptions - interrupts, aborts, resets and exception generating instructions.

The exception handling is about the same for all types of exceptions. As soon as an event occurs that causes an exception, the processor hardware automatically performs the following actions:

(i) *Update Processor State:* The processor automatically stores the processor state PSTATE into the System Processor State Register SPSR_ELn of the exception level where the exception is taken. That means - if an exception occurs at EL0 it is taken to EL1 (as long as there is not any hypervisor at EL2 and the exception handling is set to be done by the next higher exception level) and therefore the processor state would be stored to SPSR_EL1.

(ii) *Store Return Address:* In a second step, the processor stores the return address to be used at the end of the exception into the register ELR_ELn of the exception level (again) where the exception is taken.

(iii) *Exception Syndrome:* After storing the return address, the processor writes all the information needed to allow the exception handler to determine the reason for the exception to the so called Exception Syndrome Register ESR_ELn. Note, that this register is updated only for synchronous and SError exceptions - status informations on (external) interrupts (i.e. IRQ or FIQ, cf. section 3.5.1) have to be generated and handled by an external interrupt controller (preferable a GIC, cf. section 3.5.6).

(iv) *Exception Handler:* The next action, that the processor performs, is branching to a vector table that contains entries for each exception type. Each exception level has its own exception vector table containing up to 16 instructions in AArch64 execution level to handle and eventually branch

[67] [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter B, section B2.9, page B2-121.

[68] http://phidias-hypervisor.de/repos/core.git, December 21, 2017

to a more sophisticated exception handler. A detailed description of such an exception table used in AArch64 execution state can be found in the ARMv8 Programmer's Guide[69]. Warning: even though the described registers are automatically updated, they are not automatically stored to memory when the exception level is changed within the exception handler. A change of the exception level has to be implemented manually, as described in section 3.3.

(v) *Returning and Restoring:* As soon as the exception handler is done and calls the ERET instruction, the processor restores the processor state of the application, in which the exception occurred, according to the state values stored in the SPSR_ELn register. After completion, the application continues its normal program flow at the location stored in ELR_ELn.

The following section describes the registers used to handle exceptions. In addition, the ARMv8-A Virtualization Extension provides a separate register HCR_EL2 that allows a hypervisor to handle all exceptions by routing or trapping them all to the exception level EL2. A detailed view of this register with explanations to the settable bit positions can be found in the ARM AArch64 Virtualization documentation[70] as well as in the ARM Technical Reference Manual[71].

Another feature in the context of virtualization provided by the ARMv8-A architecture are virtual exceptions. If the hypervisor is given full responsibility for handling exceptions, it can forward virtual exceptions to its guest systems. The ARMv8-A architecture supports the three exception types: Virtual SError, Virtual IRQ and Virtual FIQ. Further information can be found in the AArch64 Virtualzation documentation[72].

Taking into account the explanation on exception handling and, in particular, the features of the ARM Virtualization Extension, it can be stated:

> ***REQ-9 - FULFILLED:*** *The ARMv8-A Virtualization Extension explicitly provides interruption handling that guarantees the* ***exclusive*** *treatment of interrupts by the hypervisor. Therefore, this requirement is fulfilled.*

In this context, it is also worth mentioning that the ARMv8-A architecture automatically masks all external interrupts after an exception is taken to an upper exception level. However, the exception handler can explicitly allow nested exceptions. The ARM Programmer's Guide contains some more details including a code example[73].

[69][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 10, section 10.4, page 10-12.
[70][6] n.a. *AArch64 Virtualization.* 2017, chapter 2, section 2.4, page 9 f.
[71][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D7.2.34, page D7-2302.
[72][6] n.a. *AArch64 Virtualization.* 2017, chapter 2, section 2.6, page 10 f.
[73][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 10, section 10.5, page 10-14.

3.5.1 Interrupts

The ARMv8 architecture refers to external, asynchronous interruptions as interrupts and defines two different types - the interrupt request IRQ and the fast interrupt request FIQ. An FIQ is just a higher priority interrupt request that is handled „faster" by disabling IRQ and other FIQ handlers during its exception handling [74]. Both interrupt types are physical signals to the core that are usually connected to an external interrupt controller. Since all asynchronous exceptions can principally be masked, also IRQ and FIQ can be handled accordingly by setting the DAIF exception mask bits in the SPSR_ELn register[75]. However, a General Interrupt Controller GIC is required for further control of interrupts (cf. section 3.5.6).

As the ARMv8-A architecture provides an exception handling register on a per exception level basis (including EL2 for running the hypervisor), the according requirement can be qualified as follows:

> ***REQ-10 - FULFILLED:*** *The ARMv8-A architecture provides an enabling and disabling mechanism for asynchronous, external interrupts for every exception level and therefore fulfils this requirement.*

3.5.2 SErrors

Another asynchronous exception type is the System Error (SError). This type of exception can have a number of possible causes depending on the SoC and the processor implementation, because in all of the Cortex-A5x processor series there is a separate physical signal to the core specified for the SError. The most common cause for an SError are asynchronous data aborts[76]. An example would be a mistake in a translation table that marks a ROM as read/write. If the corresponding memory is also marked as write-back cacheable, an attempt to write to the address region would initially go into the cache. At some point later the cache line(s) will get evicted, trigger a write-back of the dirty data and the memory system returns a fault (write to read-only slave), which is classed as an asynchronous SError. As already mentioned, all asynchronous exceptions can be masked and the same applies for the SError (cf. section 3.5.1).

3.5.3 Aborts

In the ARMv8 terminology, an abort is a synchronous exception generated either on a failed instruction fetch (instruction aborts) or a failed data access (data aborts). As synchronous exceptions cannot be masked, they have to be handled as described above. Further information on synchronous exception handling can be found in the ARM Technical Reference Manual[77].

[74] In AArch32 execution state, the FIQ has its own set of banked registers, cf. also 3.3)

[75] [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.14.2, page D1-1836 ff.

[76] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 10, section 10.2, page 10-7.

[77] [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.13, page D1-1826.

3.5.4 Exception Generating Instructions

The execution of certain instructions can generate exceptions. On the one hand, this includes all requests for software running at a higher exception level, i.e. the Supervisor Call SVC, Hypervisor Call HVC and Secure Monitor Call SMC. On the other hand, the exception handling can also be configured in such a way that various other instructions are disabled or cause a trap exception. As an example, cache maintenance instructions can be trapped to EL1 from EL0 by setting the according bit SCTLR_EL1.UCI in the System Control Register at EL1. The ARM Technical Reference Manual provides a complete section on all possible modifications and adjustments of the exception handling with respect to exception generating instructions[78].

As already mentioned, all exceptions can be trapped or routed to a hypervisor running at exception level EL2 by selecting the according bits in the Hypervisor Control Register. In addition, the ARM Virtualization Extension also provides a mechanism for trapping certain instructions that are often used in the context of virtualization, i.e. access to virtual memory control registers, certain system instructions (mostly maintenance instructions for caches), access to the Auxiliary Control register etc.[79]. When an instruction has trapped, the hypervisor code can read the Exception Syndrome Register ESR_EL2 to obtain the necessary information about the trapped instruction.

Due to the combination of the ARM Virtualization Extension and the handling of exception generating instructions, the following requirements of the Muen SK can be considered fulfilled:

> ***REQ-11 - FULFILLED:*** *The ARMv8-A architecture supports the configuration of exception generating instructions resulting in an exit of the guest subject and therefore fulfils this requirement.*
>
> ***REQ-13 - FULFILLED:*** *This requirement only demands that a target architecture can distinguish between the four exit reasons used by the Muen SK[80]. This means in particular that the Muen SK does not require detailed status information regarding external interrupts in the context of a guest exit. Therefore, even though the exact state of an external interrupt can only be determined using a General Interrupt Controller GIC, the ARMv8-A architecture fulfils this requirement as a hypervisor can read the demanded four different reasons of a guest exit from the Exception Syndrome Register ESR_EL2.*

[78][7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D.1.15, page D1-1842.
[79][6] n.a. *AArch64 Virtualization.* 2017, chapter 2, section 2.5, page 10.
[80][2] Buerki and Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance.* 2013, chapter 4, section 4.4.5, page 49 f.

3.5.5 Resets

The ARMv8-A architecture does not support non-maskable interrupts[81]. As already described in section 3.2.3, reset exceptions cannot be masked and hence are the only non maskable exceptions. Since every reset exception is guaranteed to be executed by the core receiving it, it can be stated that:

> ***REQ-12 - FULFILLED:*** *The only non maskable interrupt (NMI) not only leads to an exit of a guest subject but also to a restart of the core from EL3. Therefore, this requirement can be qualified as fulfilled.*

3.5.6 Generic Interrupt Controller

The Muen SK relies on the I/O APIC and LAPIC mechanism provided by the Intel x86 architecture (cf. chapter Muen, section 2.3.2). The ARMv8-A architecture implements a similar technology, called Generic Interrupt Controller (GIC), based on an internal GIC CPU Interface (corresponds conceptually to the LAPIC) and an external GIC Distributor (corresponds conceptually to the I/O APIC). This mechanism not only supports routing of software generated, private and shared peripheral interrupts between cores in a multicore environment but also the routing of external interrupts to (an) individual core(s). Furthermore, it enables software to mask, enable and disable interrupts, to prioritise individual sources and to generate software interrupts[82]. Additionally, the GIC technology simplifies the virtualization of exceptions for hypervisor implementations in a multicore environment[83].

The first major function block of the Generic Interrupt Controller technology is the GIC CPU Interface, through which the core receives an interrupt. Every core in a multicore environment has its own CPU Interface that hosts registers to identify, mask and control the states of interrupts forwarded to that core.

The second main function block of the Generic Interrupt Controller technology is the Distributor. This external component has to be implemented by the SoC manufacturer. It controls all the properties of a specific interrupt by according registers, especially the routing information and the enable status for the attached CPU Interfaces.

The details of the configuration, the initialisation and the exception handling as well as the available features are determined by the version of the implemented GIC architecture on the one hand by the respective processor according to the internal GIC CPU Interface and on the other hand by the SoC manufacturer with regard to the external GIC Distributor. For example, Locality Specific Peripheral Interrupts (LPI), i.e. message-based interrupts, are not supported with GICv1 and GICv2, whereas this mechanism can be used in all higher versions[84]. The ARMv8 Cortex-A53 processor supports all GIC

[81] [7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile.* 2017, chapter D, section D1.14.2, page D1-1836.
[82] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, cf. chapter 10, section 10.6, page 10-17.
[83] [6] n.a. *AArch64 Virtualization.* 2017, chapter 2, section 2.4 f., page 9 ff.
[84] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 10, section 10.6, page 10-17.

architectures up to version 4[85]. A good example for the initialisation and configuration of a Generic Interrupt Controller GICv3 running on the ARMv8 Foundation Model can be found in the ARM Limited startup code example delivered with the DS-5 Community Edition.

Accordingly, the two requirements 8 and 14 of the Muen SK can be evaluated as follows:

> ***REQ-8 - IMPLEMENTATION DEFINED:*** *The ARMv8 architecture principally supports the programmatical handling of interruptions. However, since the possibilities and the extent of this handling depend on the implementation of a GIC distributor by the SoC manufacturer, this requirement is qualified as implementation defined.*

> ***REQ-14 - IMPLEMENTATION DEFINED:*** *The ARMv8 architecture only provides hardware assisted routing of interruptions to individual cores through the implementation of a GIC by the SoC manufacturer. Therefore, this requirement has to be judged as implementation defined.*

3.6 Timers

The ARMv8 architecture prescribes the implementation of a system timer for processors of the Cortex-A series (cf. section 2.4). This system timer provides up to four timer channels per core - a secure and a non-secure physical timer as well as two timers for virtualization purposes. Each of these timer channels has at least one comparator, to configure the timers to generate an interrupt when the count is greater or equal to the programmed comparator value[86]. The concrete implementation of the timer is determined by the respective processor type. An example would be the Generic Timer of the ARMv8 Cortex-A53 processor series described in the ARM Cortex-A53 Technical Reference Manual[87]. The following steps are usually necessary to configure the timer:

(i) *Comparator Value:* In a first step, the comparator value for the timer has to be written to the CNTP_CVAL_ELn according to the exception level, the timer should be used for.

(ii) *Enabling Counter:* Then, the counter and the interrupt generation have to be enabled in the register CNTP_CTL_ELn.

(iii) *Reporting:* In the last step, the code can poll the CTP_CTL_ELn register to report the status of the according exception level timer interrupt.

[85][9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual.* 2016, chapter 9, section 9.1, page 9-2.
[86][8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A.* 2015, chapter 14, section 14.1.3, page 14-5 f.
[87][9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual.* 2016, chapter 10, page 10-1 ff.

The virtual timers and counters provided by the ARMv8-A architecture are explicitly designed for the scheduling of guest systems. Even though written in the programming language C, one can find a valuable example of the usage of the timer mechanism supported by the ARMv8-A architecture in the source code of the xvisor hypervisor [88]. Therefore, the following can be stated:

> ***REQ-15 - FULFILLED:*** *The ARMv8-A architecture explicitly supports at least a timer and a counter per core that can be configured to generate an interrupt. Even though the context switch has to be implemented manually (cf. section 3.3), this mechanism can be qualified as preemptive in the sense that it triggers an appropriate exception handling.*

3.7 Device Handling

Even though out of scope for this study, it has to be mentioned that the ARMv8-A supports device emulation as well as device assignment through the already described features of the ARM Virtualization Extension, i.e. the second level address translation and the (virtual) exception handling[89].

However, in order to get full device handling support, a SoC manufacturer also has to implement and provide an SMMU (corresponding to Intel's IOMMU) that meets the ARMv8 SMMU architecture specifications for the SMMU interface[90]. Therefore, the corresponding requirement can be qualified as follows:

> ***REQ-16 - IMPLEMENTATION DEFINED:*** *The ARMv8 architecture only provides a fully featured device handling through the implementation of a SMMU by the SoC.*

3.8 SPARK

As already mentioned in section 2.7, the Muen SK is written in SPARK. Since SPARK is a true subset of the Ada programming language and compilers ignore the SPARK inherent annotations, every correct SPARK program is also a correct Ada program and can therefore be compiled with an existing Ada compiler such as GNAT (part of the GNU compiler collection GCC).

To be able to qualify the requirement that there has to exist an Ada Cross Compiler for the ARMv8-A AArch64 execution state, a separate evaluation case has been written (cf. Development Environment Setup Ada Toolchain, appendix A). This document shows that it is possible to compile a custom Ada Cross Compiler for the ARMv8-A architecture based on the GNAT Ada Compiler toolchain of the GNU Compiler Collection GCC.

[88] cf. A general overview over the xvisor hypervisor can be found here. The source code is published under the GPL-2.0 license on github (https://github.com/xvisor/xvisor) and the mentioned generic timer code for ARMv8 AArch64 can be found in the file generic_timer.c in the directory arm64 commen, basic, timer. December 21, 2017

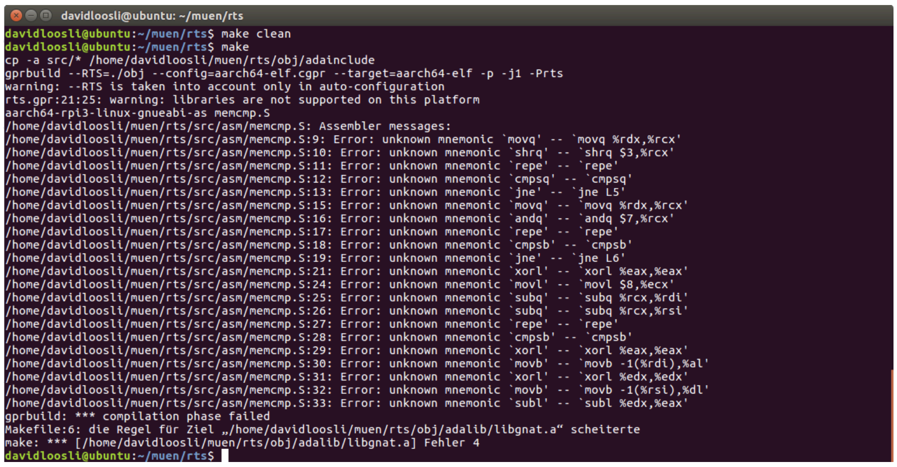
[89] [6] n.a. *AArch64 Virtualization.* 2017, chapter 2, section 2.2 f., page 8 f.

[90] [10] n.a. *ARM System Memory Management Unit, Architecture Specification.* 2017.

Thus applies:

> ***REQ-17 - FULFILLED:*** *An Ada Cross Compiler for the ARMv8-A AArch64 architecture can be compiled based on the GNAT Ada Compiler toolchain.*

The Muen SK relies on a Zero Footprint Runtime for the SPARK 2014 programming language that is provided with the source code of the Muen SK [91]. According to the last meeting with the developers of the Muen SK, the runtime should be independent of the target platform but was written for the Intel x86/64 architecture. As expected, a first test with the custom Ada Cross Compiler for the ARMv8-A AArch64 architecture showed that the Muen SK Zero Footprint Runtime has to be rewritten as it uses Intel IA-32e specific assembly instructions.



```
davidloosli@ubuntu:~/muen/rts$ make clean
davidloosli@ubuntu:~/muen/rts$ make
cp -a src/* /home/davidloosli/muen/rts/obj/adainclude
gprbuild --RTS=./obj --config=aarch64-elf.cgpr --target=aarch64-elf -p -j1 -Prts
warning: --RTS is taken into account only in auto-configuration
rts.gpr:21:25: warning: libraries are not supported on this platform
aarch64-rpi3-linux-gnueabi-as memcmp.S
/home/davidloosli/muen/rts/src/asm/memcmp.S: Assembler messages:
/home/davidloosli/muen/rts/src/asm/memcmp.S:9: Error: unknown mnemonic `movq' -- `movq %rdx,%rcx'
/home/davidloosli/muen/rts/src/asm/memcmp.S:10: Error: unknown mnemonic `shrq' -- `shrq $3,%rcx'
/home/davidloosli/muen/rts/src/asm/memcmp.S:11: Error: unknown mnemonic `repe' -- `repe'
/home/davidloosli/muen/rts/src/asm/memcmp.S:12: Error: unknown mnemonic `cmpsq' -- `cmpsq'
/home/davidloosli/muen/rts/src/asm/memcmp.S:13: Error: unknown mnemonic `jne' -- `jne L5'
/home/davidloosli/muen/rts/src/asm/memcmp.S:15: Error: unknown mnemonic `movq' -- `movq %rdx,%rcx'
/home/davidloosli/muen/rts/src/asm/memcmp.S:16: Error: unknown mnemonic `andq' -- `andq $7,%rcx'
/home/davidloosli/muen/rts/src/asm/memcmp.S:17: Error: unknown mnemonic `repe' -- `repe'
/home/davidloosli/muen/rts/src/asm/memcmp.S:18: Error: unknown mnemonic `cmpsb' -- `cmpsb'
/home/davidloosli/muen/rts/src/asm/memcmp.S:19: Error: unknown mnemonic `jne' -- `jne L6'
/home/davidloosli/muen/rts/src/asm/memcmp.S:21: Error: unknown mnemonic `xorl' -- `xorl %eax,%eax'
/home/davidloosli/muen/rts/src/asm/memcmp.S:24: Error: unknown mnemonic `movl' -- `movl $8,%ecx'
/home/davidloosli/muen/rts/src/asm/memcmp.S:25: Error: unknown mnemonic `subq' -- `subq %rcx,%rdi'
/home/davidloosli/muen/rts/src/asm/memcmp.S:26: Error: unknown mnemonic `subq' -- `subq %rcx,%rsi'
/home/davidloosli/muen/rts/src/asm/memcmp.S:27: Error: unknown mnemonic `repe' -- `repe'
/home/davidloosli/muen/rts/src/asm/memcmp.S:28: Error: unknown mnemonic `cmpsb' -- `cmpsb'
/home/davidloosli/muen/rts/src/asm/memcmp.S:29: Error: unknown mnemonic `xorl' -- `xorl %eax,%eax'
/home/davidloosli/muen/rts/src/asm/memcmp.S:30: Error: unknown mnemonic `movb' -- `movb -1(%rdi),%al'
/home/davidloosli/muen/rts/src/asm/memcmp.S:31: Error: unknown mnemonic `xorl' -- `xorl %edx,%edx'
/home/davidloosli/muen/rts/src/asm/memcmp.S:32: Error: unknown mnemonic `movb' -- `movb -1(%rsi),%dl'
/home/davidloosli/muen/rts/src/asm/memcmp.S:33: Error: unknown mnemonic `subl' -- `subl %edx,%eax'
gprbuild: *** compilation phase failed
Makefile:6: die Regel für Ziel „/home/davidloosli/muen/rts/obj/adalib/libgnat.a“ scheiterte
make: *** [/home/davidloosli/muen/rts/obj/adalib/libgnat.a] Fehler 4
davidloosli@ubuntu:~/muen/rts$
```

Figure 3.12: gprbuild Muen SK ZFP output

Although ARM provides official guidelines for porting code from ARM A32 to ARM A64 assembly[92] as well as from IA-32 to ARM A32 [93] and many freely available tutorials can be found online, the runtime could not be translated during this study due to time constraints. Therefore, it is not possible to make a final judgement regarding the corresponding requirement:

> ***REQ-18 - TESTING REQUIRED:*** *Even though it should be possible to build a Muen Zero Footprint Runtime for the SPARK 2014 programming language and the ARMv8 AArch64 execution state with freely available software, the fulfilment of this requirement has to be tested in a further study.*

[91] https://git.codelabs.ch/?p=muen.git, December 21, 2017

[92] [8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. 2015, chapter 8, page 8-1 ff.

[93] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0274b/index.html, December 21, 2017

3.9 Requirement Comparison

It has be shown that more than two thirds of the requirements of the Muen SK are directly supported by the ARMv8-A architecture. None of the prerequisites had to be qualified as *unsupported*. The fulfillment of the remaining requirements only depends on the target hardware and therefore on the implementation of the ARMv8 architecture by the respective SoC manufacturer. The following requirements had to be judged as *IMPLEMENTATION DEFINED* and thus have to be qualified based on the target hardware platform, i.e. the Raspberry Pi 3:

number	**requirement**	**topic**
REQ-0	The processor architecture has to support 64 bit datapath widths, integer size and memory address widths as well as to be able to execute 32 bit applications.	basics
REQ-8	A target processor architecture has to provide a mechanism to programmatically handle interruptions.	interruption handling
REQ-14	A target processor architecture should optionally provide a technique to fast process interruptions between cores.	interruption handling
REQ-16	A target processor architecture must provide a mechanism to virtualize I/O devices by completely isolating the access to devices and providing support for according interruption and memory features.	device handling

Table 3.1: *IMPLEMENTATION DEFINED* requirement summary

4 Raspberry Pi 3

The Raspberry Pi 3 is the third generation of the Raspberry Pi series and the target platform for this study. The first part of this chapter provides a general overview of the Raspberry Pi 3. In the following sections, the hardware platform is discussed with respect to the requirements qualified as *IMPLEMENTATION DEFINED* in the previous chapter 3.

4.1 Overview

The Raspberry Pi 3 Model B is the latest single board computer developed and released in February 2016 by the Raspberry Pi Foundation. The main component of this small computer is the BCM2837 System on Chip (SoC), which implements an ARMv8 Cortex-A53 processor with four cores. Also worth mentioning in the context of this study are the 1GB RAM, the Micro SD port and the 40-pin GPIO provided by the platform. Further details on the specifications can be found on the homepage of the Raspberry Pi Foundation [1].

The Raspberry Pi 3 Model B was chosen as target platform for this study because it is the first Raspberry Pi generation that is capable of running software written for the 64-bit execution state. In addition, the Raspberry Pi single board computers are explicitly intended for experimentation and are therefore almost not „brickable“ as well as inexpensive.

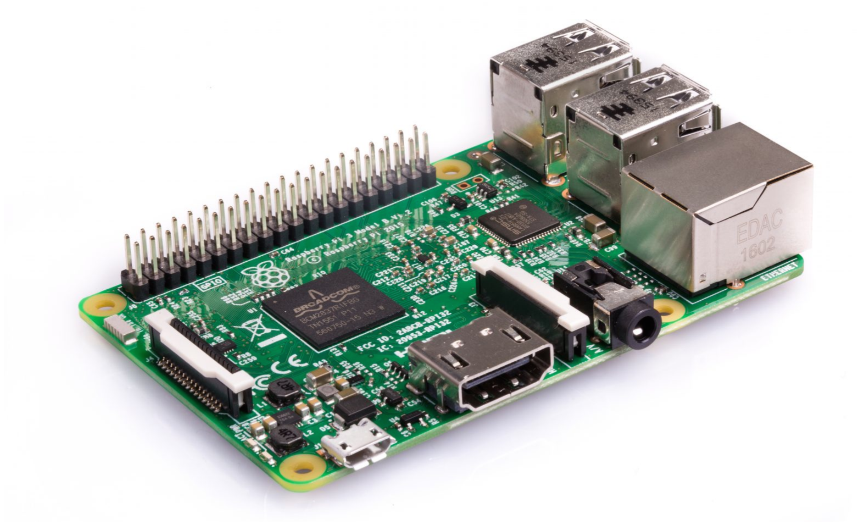


Figure 4.1: Raspberry Pi 3 Model B, ⓒ by the Raspberry Pi Foundation

[1]cf. https://www.raspberrypi.org/products/raspberry-pi-3-model-b, December 21, 2017

The architecture of the Raspberry Pi 3 does not quite come up to one's expectations. In contrast to most other ARM based SoC, not the ARMv8 Cortex-A53 processor but the Broadcom VideoCore is the organising part and has full control over the initialisation of each component. In addition, the VideoCore also contains and controls essential system architecture components such as the memory controller or the level 2 cache. The latter is used almost exclusively by the VideoCore and is usually bypassed when accessing the CPU[2]. The ARM processor is only attached to the organising VideoCore and can be addressed via a corresponding CPU interface. Figure 4.2 shows a schematic overview for the architecture of the Raspberry Pi 3 [3].

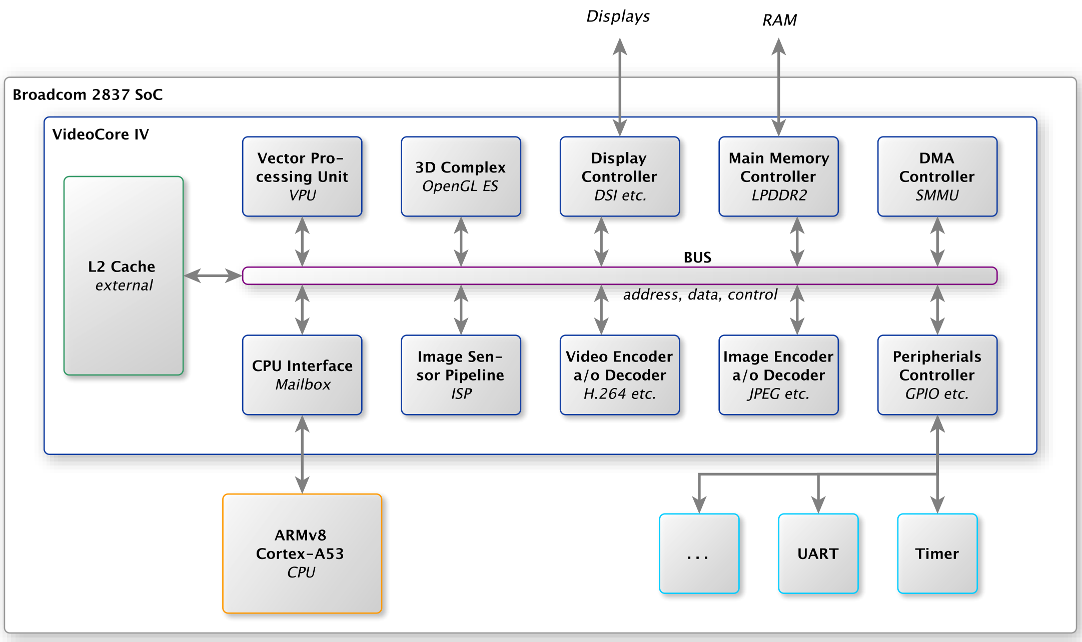

Displays
RAM
Broadcom 2837 SoC
VideoCore IV
L2 Cache external
Vector Processing Unit VPU
3D Complex OpenGL ES
Display Controller DSI etc.
Main Memory Controller LPDDR2
DMA Controller SMMU
BUS
address, data, control
CPU Interface Mailbox
Image Sensor Pipeline ISP
Video Encoder a/o Decoder H.264 etc.
Image Encoder a/o Decoder JPEG etc.
Peripherials Controller GPIO etc.
ARMv8 Cortex-A53 CPU
...
UART
Timer

Figure 4.2: Raspberry Pi 3 schematic

4.1.1 Documentation

First of all, it has to be stated that there exists neither a complete official documentation on the Raspberry Pi 3 nor any official documentation on the changes with respect to the AArch64 mode of the Raspberry Pi 3. On the website of the Raspberry Pi Foundation, it is only mentioned that nothing has changed compared to the Raspberry Pi 2 SoC except for the ARMv8-A processor [4]. The documentation for the Raspberry Pi 2 consists of two datasheets for the Raspberry Pi 1 [5] and a supplementary

[2] [11] n.a. *BCM2835 ARM Peripherals*. 2012, chapter 1, section 1.2.3, page 6.
[3] https://www.heise.de/ct/ausgabe/2016-8-Wie-es-mit-dem-Raspberry-Pi-weitergeht-3150082.html, December 21, 2017
[4] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md, December 21, 2017
[5] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md, December 21, 2017

document for the changes compared to the Raspberry Pi 1 [6]. Even though there obviously exist differences between the 64-bit and the 32-bit mode of the Raspberry Pi [7], most of the following statements should apply to both execution states [8]. Therefore this chapter is primarily based on the following literature:

- *VideoCore:* The official VideoCore IV 3D Architecture Reference Guide[9] for the Raspberry Pi 1 serves as the main source for boot related questions.
- *Broadcom SoC:* As the primary sources for ARM Peripheral related topics, the two official BCM2836 ARM Peripherals[10] and BCM2835 ARM Peripherals[11] documents are used.
- *Raspberry Pi Bare Metal Forum:* A lot of explanations and findings in the context of the AArch64 development can be found on the official Raspberry Pi Bare Metal Forum [12].
- *Raspberry Pi Repositories:* The Raspberry Pi Foundation maintains several Github repositories. In particular, the documentation repository was used for this chapter [13].
- *Bare Metal Repositories:* The most important Raspberry Pi Bare Metal repositories for this study are the two Github repositories maintained by David Welch [14] and by Peter Lemon [15].

Because a detailed and with respect to the AArch64 architecture complete Raspberry Pi 3 hardware reference manual as well as a comprehensive guide for Bare Metal Programming on the Raspberry Pi 3 did not exist at the time of writing, a separate Raspberry Pi 3 Beginner's Guide has been started as a collection of all the existing, but widespread sources on this topic. This guide is going to be continued and developed by the author even after this Student Research Project and is going to be published under an open source license.

[6] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md, December 21, 2017
[7] David Welch and Peter Lemon could show with their code that not only the base address for the kernel image but also some alternative modes for the peripherals change. A **personal assumption** in this regard is that these changes are caused by the firmware of the VideoCore initialising the ARM processor in the AArch64 execution state.
[8] Of course, this would have to be proven in a continuing study (cf. section 5.3)
[9] [13] n.a. *VideoCore IV 3D Architecture Reference Guide.* 2013.
[10] [5] Loo. *BCM2836 ARM Peripherals (documentary supplement).* 2014.
[11] [11] n.a. *BCM2835 ARM Peripherals.* 2012.
[12] https://www.raspberrypi.org/forums/viewforum.php?f=72, December 21, 2017
[13] https://github.com/raspberrypi, December 21, 2017
[14] https://github.com/dwelch67/raspberrypi, December 21, 2017
[15] https://github.com/PeterLemon/RaspberryPi, December 21, 2017

4.1.2 Bare Metal Development

The development of bare metal programs differs greatly from software development on higher abstraction levels. The variety of development tools (compiler, IDE etc.) is relatively wide, but not all available tools are suitable for a specific task. The setups for the Raspberry Pi 3 used in this study are therefore briefly explained in this section.

The first inconvenience in bare metal development is loading newly built or rebuilt kernel images from the IDE to the Raspberry Pi. Actually, there are four possibilities for this task:

(i) *SD Card:* For the AArch64 development, this option consists in formatting an SD Card to FAT32, copying the corresponding kernel image kernel8.img together with the boot files bootcode.bin, start.elf as well as config.txt to the card, inserting the card into the card slot of the Raspberry Pi and restarting it. Further information and two code examples are recorded in the two evaluation cases *Hello Muen! on HDMI* written in assembly and *Hello Muen! on UART* written in C (cf. appendix A).

(ii) *Bootloader:* David Welch provides a bootloader that is capable of loading a kernel image to the Raspberry Pi 3 over a serial connection. Both the bootloader and instructions for its usage can be found on David Welch's Github repository [16].

(iii) *JTAG:* The Joint Test Action Group JTAG interface can not only be used to load a kernel image to the Raspberry Pi 3 but also allows to run a debugger like the freely available Open On-Chip Debugger (OpenOCD). Therefore, this option has been chosen for this study. A complete guide for setting up the hardware as well as the OpenOCD debugger in combination with the Eclipse IDE is contained in the evaluation cases *Development Environment Setup* (cf. appendix A).

(iv) *Netboot:* Since the JTAG option seemed to be the most suitable one for this study, the Netboot was not tested during this project. However, a guide for this option can be found on the official Raspberry Pi Foundation homepage [17].

Of course, the compiler toolchain as well as the IDE depend on the programming language used in a specific project. Nevertheless, the GNU MCU Eclipse IDE from Liviu Ionescu [18] has to be mentioned here, because it has been used as a development environment in almost all experiments of this study and can be adapted to different languages. The GNAT Programming Studio (GPS) [19] of the Community Edition [20] provided by AdaCore was used for the development of code examples written in Ada.

[16] https://github.com/dwelch67/raspberrypi, December 21, 2017
[17] https://www.raspberrypi.org/blog/pi-3-booting-part-ii-ethernet-all-the-awesome, December 21, 2017
[18] https://github.com/gnu-mcu-eclipse/org.eclipse.epp.packages/releases, December 21, 2017
[19] https://www.adacore.com/gnatpro/toolsuite/gps, December 21, 2017
[20] https://www.adacore.com/community, December 21, 2017

4.2 Boot Process

Due to the special architecture of the Raspberry Pi 3 (cf. section 4.1), the boot process also does not correspond to the one of most other ARM development boards. As soon as the Raspberry Pi is turned on, the VideoCore assumes control over the boot process while the ARMv8 Cortex-A53 processor is still off and uninitialised [21]. The VideoCore then takes the following actions [22]:

(i) *First Stage Bootloader:* The VideoCore starts the boot process by executing the first stage bootloader stored in ROM on the Raspberry Pi SoC. This bootloader initialises and reads the SD card and loads the second stage bootloader from the SD card into the level 2 cache.

(ii) *Second Stage Bootloader (bootcode.bin):* This bootloader enables and initialises the SDRAM. While for earlier versions of the Raspberry Pi it loads the third stage bootloader loader.bin from the SD card into RAM, the second stage bootloader for the Raspberry Pi 3 supports loading ELF files and therefore directly loads the GPU firmware from the SD card into RAM .

(iii) *GPU firmware (start.elf):* The start.elf first initialises the GPU, second loads, reads and executes the CPU configuration file config.txt and finally loads the kernel image into RAM.

The above described boot process already suggests that there are basically two possibilities for configuring the ARMv8 processor. The first option is to modify the configuration file accordingly [23]. For all non hardware dependent configurations, the processor can also be initialised manually.

As already explained in section 3.2.3, the initialisation of the ARMv8 processor into the AArch64 execution state depends on a hardware signal to a pin of the processor. Since the VideoCore starts the ARMv8 processor by default in the AArch32 execution state and since a warm reset depends on a hardware defined reset register with an unknown address in AArch64 execution state, the only way to initialise the ARMv8 processor in 64-bit mode is to add the following lines to the config.txt file:

```
arm_control=0x200
kernel_old=1
```

Even though it seems that there exists only this option to start the ARMv8 Cortex-A53 processor on the Raspberry Pi 3 in the AArch64 execution state, the corresponding requirement can be qualified as fulfilled:

> ***REQ-0 - FULFILLED:*** *The Raspberry Pi 3 supports the initialisation of the ARMv8 processor in a 64-bit execution state and hence fulfils this requirement.*

[21] cf. boot process explained by David Welch on https://github.com/dwelch67/raspberrypi, December 21, 2017
[22] https://www.raspberrypi.org/documentation/.../bootflow.md, December 21, 2017
[23] Details to the configuration possibilities can be found on https://www.raspberrypi.org/documentation/configuration/config-txt as well as on the Raspberry Pi Foundation Github repositories, December 21, 2017

4.3 Exception Handling

First of all, it has to be stated that the interrupt controller provided by the Raspberry Pi 3 SoC is neither programmable nor does it implement the Generic Interrupt Controller (GIC) interface specified by the ARMv8 architecture.

The exception handling on the Raspberry Pi 3 is also special. The documentation distinguishes between two different types of interrupts, i.e. core related and core un-related interrupts. The category of the core related interrupts includes the four timer interrupts, a performance monitor interrupt and the four Mailbox interrupts for each core. The only thing that can be determined programmatically with respect to core related interrupts is whether to send an interrupt to either the IRQ pin or the FIQ pin as well as to disable the interrupt handling at all[24]. An example of a Mailbox interrupt handling can be found in the evaluation case *Hello Muen! on HDMI*. All other interrupts and exceptions (GPU interrupts, local timer interrupts, AXI error and Peripheral interrupts) are assigned to the core un-related interrupts category[25]. These interrupts have to be enabled, configured and handled completely in code by setting the according bits of an interrupt register of the corresponding interrupt type as well as by setting up the processor correctly [26]. An example for an UART interrupt handling can be found in the evaluation case *Hello Muen! on UART*.

The question now arises as to whether the described exception handling meets the requirements of the Muen SK. In order to be able to assess this question, one has to take a closer look at two practical examples. Both the Xen hypervisor [27] and the Kernel Virtual Machine KVM [28] explicitly state that they rely on an implementation of the GIC interface specified by ARM Limited. While the Xen hypervisor therefore does not support the Raspberry Pi 3, KVM circumvents this problem by implementing the GIC specification in a virtual GICv2 interface. The second option would also allow the Muen SK to run on the Raspberry Pi 3. However, since the Muen SK requires a smallest possible code base and the Raspberry Pi 3 does not implement the GIC interface, the corresponding requirements derived in chapter 2 have to be qualified as not fulfilled.

> ***REQ-8 - NOT FULFILLED:*** *The Raspberry Pi 3 does neither support a fully programmable interrupt controller nor the GIC interface specified by the ARMv8-A architecture. Therefore, this requirement has to be judged as not fulfilled.*
>
> ***REQ-14 - NOT FULFILLED:*** *Even though the Raspberry Pi 3 provides a mechanism to enable fast interrupt requests FIQ, it does support an inter-core communication due to the missing implementation of the GIC interface. Hence, this requirement has to be qualified as not met by the target platform.*

[24] [5] Loo. *BCM2836 ARM Peripherals (documentary supplement)*. 2014, chapter 3, section 3.2.1, page 5.
[25] [5] Loo. *BCM2836 ARM Peripherals (documentary supplement)*. 2014, chapter 3, section 3.2.2, page 5 f.
[26] cf. https://www.raspberrypi.org/forums/viewtopic.php?f=72&t=38076, December 21, 2017
[27] https://wiki.xenproject.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper, December 21, 2017
[28] https://lwn.net/Articles/557132/, December 21, 2017

4.4 Device Handling

Since this topic is out of scope for this study, the device handling on the Raspberry Pi 3 is not discussed in detail. However, it can be stated that even though the Raspberry Pi 3 has a separate MMU for device handling it does not implement the SMMU interface specified by ARM Limited[29] [30]. Therefore, the corresponding requirement derived in the second chapter is not met:

> ***REQ-16 - NOT FULFILLED:*** *The Raspberry Pi 3 does not support the SMMU interface specified by the ARMv8-A architecture.*

4.5 SPARK

In the context of this study, it was also tried to build the official AdaCore Zero Footprint Runtime for the ARMv8-A AArch64 execution state with hardware specific adaptations for the Raspberry Pi 3 from the AdaCore Github Repository [31]. However, this attempt also failed. A detailed description can be found in the evaluation case *Problem Description Toolchain*.

[29][11] n.a. *BCM2835 ARM Peripherals.* 2012, chapter 1, section 1.2, page 4 ff., and chapter 10, section 10.6.3, page 158 f.
[30]https://www.reddit.com/r/raspberry_pi/comments/4aonbh/why_are_there_two_mmus_on_the_bcm2835 and https://www.raspberrypi.org/forums/viewtopic.php?f=72&t=138108&p=920301, December 21, 2017
[31]https://github.com/AdaCore/bb-runtimes/tree/gpl-2017/aarch64/rpi3, December 21, 2017

5 Conclusion

The aim of this feasibility study was to evaluate the ARMv8 Virtualization Extension for the porting of the Muen SK to the ARMv8 architecture as well as to carry out a risk assessment on its portability to the target platform Raspberry Pi 3 with regard to a possible bachelor thesis. This chapter is dedicated to this two aspects of the study.

5.1 ARMv8 Architecture

Principally, the ARMv8 architecture and the ARMv8 Virtualization Extension can be considered suitable for porting the Muen SK. Nevertheless, there are some risks involved that have to be addressed.

The first point and at the same time the one with the highest risk for the bachelor thesis is the context handling. In contrast to Intel's VT-x technology, the ARM Virtualization Extension does not provide any automatic handling of a context switch (cf. section 3.3). In addition, the registers, that have to be stored, depend to a certain degree on the respective guest system and the current execution state of the subject. Therefore, the context switch has to be implemented completely by the hypervisor developer.

As on the Intel x86/64 architecture, the caching structures of the ARMv8 architecture too have to be considered as potential sources of side channels. Since the ARMv8 architecture does not specify the implementation of the level 2 and an optional level 3 cache, it is also important to investigate the actual implementation of the caching structures by the manufacturer of a target SoC.

The two specifications of the Generic Interrupt Controller and the System Memory Management Unit by ARM Limited also pose a certain risk. Due to the large number of different versions and sometimes only partial implementations of the interfaces by the manufacturers of a SoC, these two components have to be examined particularly thoroughly when choosing a target platform.

5.2 Raspberry Pi 3

First of all, the missing documentation for the AArch64 mode of the Raspberry Pi 3 has to be considered as problematic, since a precisely described and defined operation mode is essential, especially for high-security applications.

The first problem could be mitigated by an open source firmware. Although Broadcom has published the documentation for the Raspberry Pi 1, a large part of the firmware is still only available in a binary format. Since the VideoCore also has complete control over the initialisation of the hardware, many details can only be estimated (e.g. memory allocation VideoCore vs. CPU). This also has to be qualified as a major risk for porting the Muen SK to the Raspberry Pi 3.

The last two risks are related to the implementation of the GIC and the SMMU interfaces specified by the ARMv8 architecture. Even though the two interfaces can be implemented in software, this involves on the one hand a high risk with regard to the bachelor thesis and on the other hand it is fundamentally contradictory to the requirement of a smallest possible code basis stated by the Muen SK.

Therefore the risk for choosing the Raspberry Pi 3 as the target platform is too high, especially without any further investigations. As a conclusion, it cannot be qualified as suitable for porting the Muen SK to the ARMv8-A architecture with respect to a possible bachelor thesis.

5.3 Further Investigations

In this final section of the study, an approach for further investigations is presented. In a first step, one of the following reference kernels could be used as a starting point for the porting of the Muen SK to the ARMv8-A architecture:

- *HASPOC hypervisor:* The HASPOC hypervisor is a high assurance security kernel for the ARMv8 architecture that is available as open source software under the terms and conditions of the Apache License 2.0. The documentation and the source code can be found on the HASPOC homepage [1].
- *seL4 microkernel:* According to the official seL4 homepage [2], the seL4 microkernel is the most advanced member of the L4 microkernel family. The source code is published under the GPLv2 and the BSD2 license on Github [3].
- *Xvisor hypervisor:* The Xvisor hypervisor is an open source type I hypervisor that supports full virtualization also for the ARMv8-A architecture [4]. The source code can be found on the Xvisor Github repository [5].
- *Phidias:* As already mentioned, the Phidias hypervisor developed by Jan Nordholz follows the same principle as the Muen SK but seems to support the ARMv8-A AArch64 architecture[6]. The source code is published on the Phidias Repository [7].

[1] https://bitbucket.org/account/user/sicssec/projects/HASPOC, December 21, 2017
[2] https://sel4.systems, December 21, 2017
[3] https://github.com/seL4/seL4, December 21, 2017
[4] http://xhypervisor.org/, December 21, 2017
[5] https://github.com/xvisor/xvisor/tree/v0.2.10, December 21, 2017
[6] [15] Nordholz. *Design and Provability of a Statically Configurable Hypervisor.* 2017.
[7] http://phidias-hypervisor.de/repos/core.git, December 21, 2017

In a second step, further classifications of the actual features used by the Muen SK in the context of the Programmable Interrupt Controller and the System Memory Management Unit would have to be carried out. In addition, the implementation of the interfaces in software has to be balanced against the requirement of a smallest possible code base stated by the Muen SK.

Depending on the findings of the first two steps, an alternative ARMv8 platform may have to be considered. It is recommended to investigate the Hardkernel Odroid C2 based on an AMLOGIC S905 SoC as the first alternative platform. This target platform seems to be documented in detail and to have hardware support for the GICv2 as well as the SMMU interface.

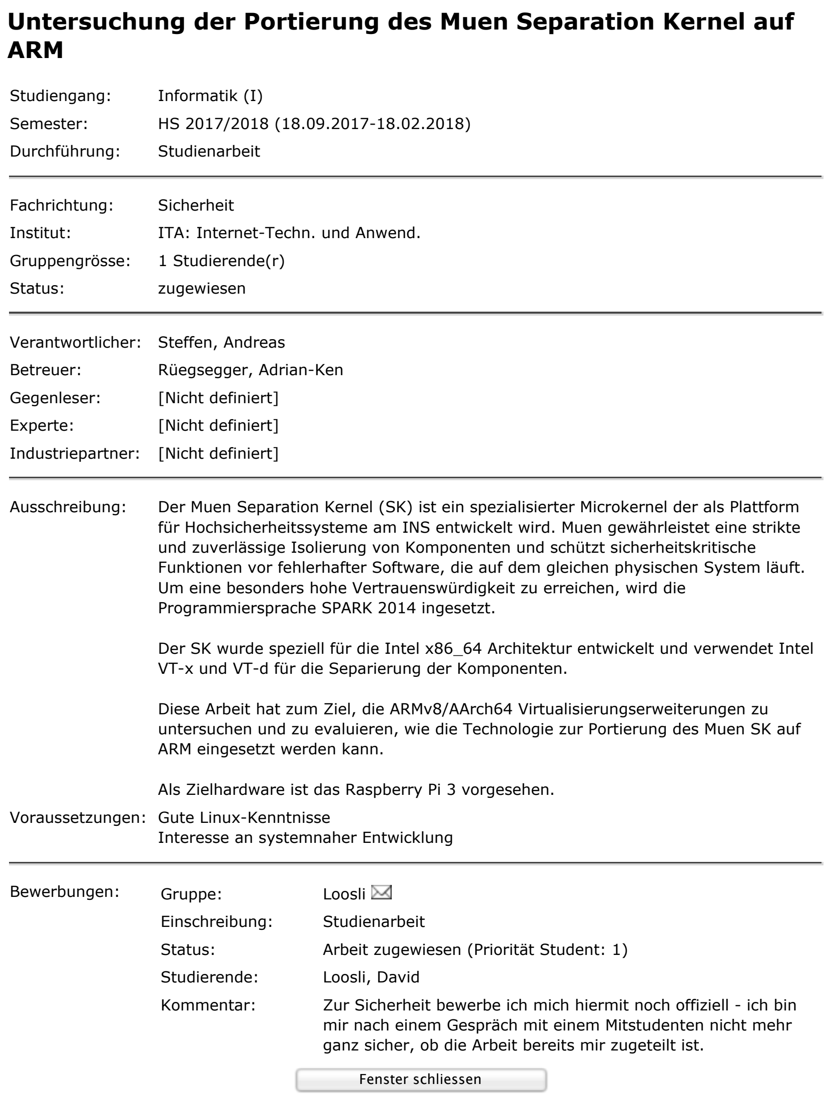
As the last part of the investigation before porting the Muen SK to the ARMv8-A architecture, additional clarifications of the registers, that have to be saved during a context switch, could be helpful.

Appendix

A List of Related Documents

- Glossary
- Hello Muen! on HDMI - *bare metal assembly code for Raspberry Pi 3*
- Hello Muen! on UART - *bare metal C code on Raspberry Pi*
- Development Environment Setup - *Assembly and C/C++ toolchain, JTAG debugger and IDE for ARMv8 AArch64*
- Development Environment Setup - *Ada toolchain, JTAG debugger and IDE for ARMv8 AArch64*
- Problem Description Toolchain - *Ada toolchain ARMv8 AArch64*
- Raspberry Pi 3 AArch64 - *An Unofficial Bare Metal Beginner's Guide (to be continued)*

B Project Assignment AVT (german)



Untersuchung der Portierung des Muen Separation Kernel auf ARM

Studiengang:	Informatik (I)
Semester:	HS 2017/2018 (18.09.2017-18.02.2018)
Durchführung:	Studienarbeit

Fachrichtung:	Sicherheit
Institut:	ITA: Internet-Techn. und Anwend.
Gruppengrösse:	1 Studierende(r)
Status:	zugewiesen

Verantwortlicher:	Steffen, Andreas
Betreuer:	Rüegsegger, Adrian-Ken
Gegenleser:	[Nicht definiert]
Experte:	[Nicht definiert]
Industriepartner:	[Nicht definiert]

Ausschreibung:

Der Muen Separation Kernel (SK) ist ein spezialisierter Microkernel der als Plattform für Hochsicherheitssysteme am INS entwickelt wird. Muen gewährleistet eine strikte und zuverlässige Isolierung von Komponenten und schützt sicherheitskritische Funktionen vor fehlerhafter Software, die auf dem gleichen physischen System läuft. Um eine besonders hohe Vertrauenswürdigkeit zu erreichen, wird die Programmiersprache SPARK 2014 ingesetzt.

Der SK wurde speziell für die Intel x86_64 Architektur entwickelt und verwendet Intel VT-x und VT-d für die Separierung der Komponenten.

Diese Arbeit hat zum Ziel, die ARMv8/AArch64 Virtualisierungserweiterungen zu untersuchen und zu evaluieren, wie die Technologie zur Portierung des Muen SK auf ARM eingesetzt werden kann.

Als Zielhardware ist das Raspberry Pi 3 vorgesehen.

Voraussetzungen: Gute Linux-Kenntnisse
Interesse an systemnaher Entwicklung

Bewerbungen:

Gruppe:	Loosli ✉
Einschreibung:	Studienarbeit
Status:	Arbeit zugewiesen (Priorität Student: 1)
Studierende:	Loosli, David
Kommentar:	Zur Sicherheit bewerbe ich mich hiermit noch offiziell - ich bin mir nach einem Gespräch mit einem Mitstudenten nicht mehr ganz sicher, ob die Arbeit bereits mir zugeteilt ist.

Bibliography

[1] Benjamin A. Braun, Suman Jana, and Dan Boneh. "Robust and Efficient Elimination of Cache and Timing Side Channels". In: *CoRR* abs/1506.00189 (2015), p. 15. URL: http://arxiv.org/abs/1506.00189.

[2] Reto Buerki and Adrian-Ken Rueegsegger. *Muen - An x86/64 Separation Kernel for High Assurance*. Rapperswil (Switzerland): University of Applied Sciences Rapperswil (HSR), 2013. URL: https://muen.codelabs.ch.

[3] Tessaleno Devezas, João Leitão, and Askar Sarygulov. *Industry 4.0 - Entrepreneurship and Structural Change in the New Digital Landscape*. Covilhã (Portugal) and Saint Petersburg (Russia): Springer International Publishing AG, 2017. ISBN: 978-3-319-49603-0.

[4] Eduard Glatz. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 2nd ed. Urdorf (Switzerland): dpunkt.verlag GmbH, Heidelberg, 2010.

[5] Gert van Loo. *BCM2836 ARM Peripherals (documentary supplement)*. revision 3.4. Cambridge (England), 2014.

[6] n.a. *AArch64 Virtualization*. version 1.0. Cambridge (England): ARM Limited, 2017.

[7] n.a. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*. version B.a. Cambridge (England): ARM Limited, 2017. URL: http://www.arm.com.

[8] n.a. *ARM Cortex-A Series, Programmer's Guide for ARMv8-A*. version 1.0. Cambridge (England): ARM Limited, 2015. URL: http://www.arm.com.

[9] n.a. *ARM Cortex-A53 MPCore Processor, Technical Reference Manual*. revision r0p4. Cambridge (England): ARM Limited, 2016. URL: http://www.arm.com.

[10] n.a. *ARM System Memory Management Unit, Architecture Specification*. version 3.0 and version 3.1. Cambridge (England): ARM Limited, 2017. URL: http://www.arm.com.

[11] n.a. *BCM2835 ARM Peripherals*. version 1.0. Cambridge (England): Broadcom Europe Ltd., 2012.

[12] n.a. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*. Santa Clara (USA): Intel Corporation, 2017. URL: https://software.intel.com/en-us/articles/intel-sdm.

[13] n.a. *VideoCore IV 3D Architecture Reference Guide*. version 1.0. Irvine CA (USA): Broadcom Ltd., 2013.

[14] Gil Neiger et al. "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization". In: *Intel Technology Journal* 10.3 (2006), pp. 167–177.

[15] Jan Nordholz. *Design and Provability of a Statically Configurable Hypervisor*. Berlin (Germany): Technische Universität Berlin, 2017.

[16] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *ACM Operating Systems Review* 17.7 (1974), pp. 412–421.

[17] John Rushby. “Design and Verification of Secure Systems”. In: *ACM Operating Systems Review* 15.5 (1981), pp. 12–21.

[18] Andrew S. Tanenbaum and Herbert Bos. *Moderne Betriebssysteme*. 4th ed. München (Germany): Pearson Studium, Hallbergmoos, 2016.

List of Figures

List of Tables