TÉCNICO LISBOA

Hypervisor Board Support Package Migration

SPARC and ARM study case

Luis Miguel Carmona Murta Mendes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. João Nuno de Oliveira e Silva

Eng. Daniel Silveira

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques

Supervisor: Prof. João Nuno de Oliveira e Silva

Member of the Committee: Prof. Luís Manuel Antunes Veiga

June 2019

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgements

Agradeço a todos os envolvidos nesta tese, em particular ao Daniel Silveira pela disponibilização de todo o tempo e conhecimento de que precisei e ao Professor João Silva pelas diretrizes e paciência na redação deste documento.

Abstract

A new space market is in its infancy, with both NASA's High Performance Spaceflight Computing (HPSC) and ESA's DAHLIA project developing an ARM based radiation-hardened board for deep space flight, and the lessened radiation requirements for lower earth orbit satellites and launchers which can use industry-grade ARM Systems-on-a-Chip (SoCs). AIR is a Time and Space Partitioning (TSP) hypervisor, implementing the ARINC 653 standard, developed for ESA's last generation satellites running on SPARC-based computers. GMV now seeks to accompany the space industry trend by adding new ARM board support packages (BSPs) to its hypervisor. With the objective of reducing the time to market, the method proposed is to migrate the existing SPARC BSP to ARM. This thesis accomplishes a well documented side-by-side comparison of the two architectures and an indepth review of the changes that ARMv7 brings to a hypervisor. It succeeds in broadening the BSP portfolio of AIR by successfully developing the BSP for an Arty Z7 board from Digilent based on Zynq-7000 SoC by Xilinx, featuring two Cortex-A9 processors.

Keywords: virtualization, hypervisor, SPARCv8, ARMv7, AIR, Zynq-7000.

Resumo

Um novo mercado espacial está na sua infância, com o High Performance Spaceflight Computing (HPSC) da NASA e o projecto DAHLIA da ESA a desenvolverem placas ARM resistentes a radiação para exploração espacial, e os requisitos de radiação menos rigorosos para satélites e lançadores de baixas órbitas terrestres que podem utilizar os sistemas ARM usados na indústria. O AIR é um hipervisor com enfâse na segregação temporal e espacial que implementa a norma ARINC 653, e foi desenvolvido para os satélites de última geração da ESA, baseados na arquitetura SPARC. A GMV procura agora acompanhar a tendência da indústria espacial, adicionando novos pacotes de suporte a placas (BSPs) ARM ao seu hipervisor. Com o objectivo de reduzir o tempo de chegada ao mercado, o método proposto é migrar o SPARC BSP já existente para ARM. Esta tese realiza uma comparação entre as duas arquiteturas e uma revisão das inovações que o ARMv7 traz para um hipervisor. Consegue também ampliar o portfólio de BSPs do AIR desenvolvendo com sucesso o BSP para uma placa Arty Z7 da Digilent baseada no Zynq-7000 SoC da Xilinx, que usa dois processadores Cortex-A9.

Palavras-chave: virtualização, hipervisor, SPARCv8, ARMv7, AIR, Zynq-7000.

Contents

List of Tables

List of Figures

Acronyms

AEEC Airlines Electronic Engineering Committee

APEX application executive

API application programming interface

ARM Advanced RISC Machine

ASI Address Space Identifier

ASR ancillary state register

BSP board support package

CCSDS Consultative Committee for Space Data Systems

CP coprocessor

CPSR current processor state register

CPU central processing unit

CWP current window pointer

ECSS European Cooperation for Space Standardization

ESA European Space Agency

EU European Union

FDIR fault detection, isolation and recovery

FIQ fast interrupt request

FPU floating-point unit

GIC general interrupt controller

GNC Guidance, Navigation and Control

GNSS Global Navigation Satellite System

HM Health Monitor

HPSC High Performance Spaceflight Computer

HVC hypercall

I/O input/output

IMA Integrated Modular Avionics

IMA-SP IMA for Space

IPC interprocess communication

IRQ interrupt request

ISA instruction set architecture

ISS International Space Station

IU integer unit

LEO Lower Earth Orbit

LR link register

LRU line-replaceable unit

MMU memory management unit

MPU memory protection unit

NASA National Aeronautics and Space Administration

OBC onboard computer

OBSW onboard software

OS operating system

PC program counter

PIL Processor Interrupt Level

PMK Partition Management Kernel

POS Partition Operating System

PSO partial store ordering

PSR processor state register

RISC reduced instruction set computer

ROM read-only memory

RTOS Real Time Operating System

SAVOIR Space AVionics Open Interface aRchitecture

SEP System Execution Platform

SMP symmetric multiprocessing

SoC System on a Chip

SOIS Spacecraft Onboard Interface Services

SP stack pointer

SPARC Scalable Processor ARChitecture

SPSR saved processor state register

SVC supervisor call

TBR Trap Base Register

TLB translation lookaside buffer

TSO total store ordering

TSP time and space partitioning

VM virtual machine

VMM virtual machine monitor

WCET worst-case execution time

WIM Window Invalid Mask

1. Introduction

Taking into consideration the need for a $32\,\text{bit}$ processor for newer satellites, the European Space Agency (ESA) performed two architectural studies in 1989 and 1990, to choose a successor to the MA31750A $16\,\text{bit}$ processors [1], based on the MIL-STD-1750A standard [2], used in deep-space missions such as the Rosetta[1] [3]. The accepted approach was to create an European-designed processor based on the Scalable Processor ARChitecture (SPARC) instruction set architecture (ISA). SPARC was selected over its competitors due to an entirely open architecture without any patents or license fees, which gave rise to a well designed and documented standard [4]. First with the ERC32, successfully deployed on the International Space Station (ISS), the Automated Transfer Vehicles[2] and the PROBA-1[3], and later with the LEON series of fault-tolerant processors, first developed by ESA and later by Gaisler Research, still in use [5], [6].

With the recent interest in the autonomy of unmanned space vehicles, as stated by the National Aeronautics and Space Administration (NASA)'s 2015 Technology Roadmaps [7], advanced Guidance, Navigation and Control (GNC) algorithms that require: a) guidance from image processing and trajectory optimization functions; b) navigation that demands more image processing, to extract and track points in the obtained images; c) robust control, based on models to predict the future state and control the spaceship under continuous perturbations; need to be introduced in the satellites' onboard computers (OBCs). Furthermore, payload data processing is also becoming an increased necessity due to higher acquisition rates and low transmission windows [8]. As such, NASA issued the High Performance Spaceflight Computer (HPSC) Processor Chiplet program solicitation in 2016, which is expected to produce a radiation-hardened multi-core Advanced RISC Machine (ARM) processor. The solicitation was won by Boeing and is expected to yield results by 2020.

As a response to NASA's solicitation, the European Union (EU) has launched the COMPET-1-2016: Technologies for European non-dependence and competitiveness call [9], under the Horizon 2020 initiative. As a result of this call, the DAHLIA [10] project, a collaboration between STMicroelectronics, Airbus Defence and Space, Integrated Systems Development, NanoXplore and Thales Alenia Space, has been chosen to also create a radiation-hardened ARM-based System on a Chip (SoC), to be used in future space missions by Europe. Expected to perform 20 to 40 times faster than current SoC for space and more than twice as fast as the LEON4 chip, a quad-core processor based on SPARC and the lastest innovation from Gaisler Research, the ARM-based SoC will be able to run GNC algorithms and handle Global Navigation Satellite System (GNSS) data and telemetry through integrated peripherals, all on the

[1] https://www.esa.int/Our_Activities/Space_Science/Rosetta
[2] https://www.esa.int/Our_Activities/Human_and_Robotic_Exploration/ATV
[3] https://www.esa.int/Our_Activities/Observing_the_Earth/Proba-1

same chip [11]. With the added benefit of being an European company, based in the United Kingdom, led to the ARM Cortex-R52, the only $64\,\mathrm{bit}$ processor in the real-time family up to the decision, being chosen for the DAHLIA project.

1.1. Motivation

With the advent of nano satellites, a low-cost space market is in the making. Previously impossible due to high satellite development and launching costs, nano satellites make it possible to achieve lower building costs and the possibility of aggregating several satellites into one launch (the PSLV-C37 rocket of the Indian Space Agency holds the record with 104 satellites). Also contributing on lowering the costs is the lessened requirement on radiation-hardened processors for Lower Earth Orbits (LEOs), making any mass-market SoC a good contender, with proper shielding and/or redundancy.

Succeeding the emerging market of nano satellites, with the possibility of new processor architectures appearing in the space segment, and the introduction of a radiation-hardened ARM SoC, has captured the attention of GMV into opening their previously mono-architecture operating system (OS) for additional processor families, such has ARM, PowerPC and RISC-V.

Portugal is also eager to enter this new market, recently demonstrated by the creation of the Portuguese Space Agency [12] in March 13, 2019. Led by Chiara Manfletti, the space agency's President, it defined as the agency's priority to implement a shared space strategy with all shareholders and stakeholders in Portugal's space market. It is also responsible for the development of the space port in the isle of *Santa Maria*, in Azores [13]. There is also the Infante Project [14], an 100% Portuguese technology satellite with ARM SoCs onboard, approved in 2017 by the *Agência Nacional de Inovação* (ANI), and expected to launch in 2021, where GMV takes part in the consortium.

With the possibility of multiple ARM processors being deployed to space on multiple fronts, GMV has a keen interest in broadening its hypervisor to this architecture. The hypervisor, codenamed AIR, can be divided into three different segments:

- The Partition Management Kernel (PMK) contains the logic of the hypervisor, making calls to the BSP when it needs to interact with the hardware.
- The BSP is designed for each board AIR needs to be deployed on.
- LIBAIR offers the OSs running on top of AIR an application programming interface (API) to interact with the hypervisor. The implementation of LIBAIR is architecture specific, since it is written in assembly.

The BSP provides a generic API to the PMK, abstracting the PMK from the hardware and removing the necessity for any modifications on its code. AIR is illustrated in Figure 1.1.

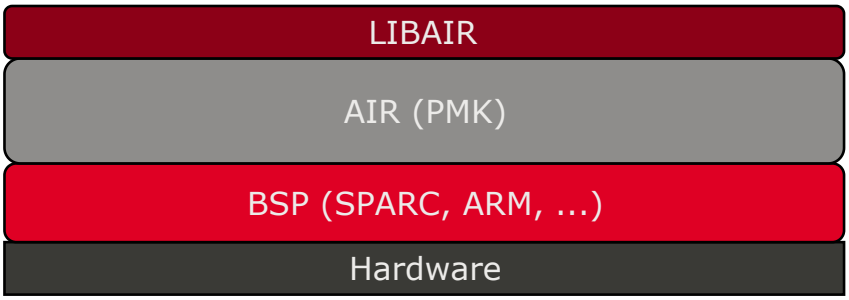


Figure 1.1.: AIR overview

1.2. Objectives and approach

This work aims at broadening the architecture spectrum of AIR. The targeted processor is the ARM Cortex-A9, present in Xilinx's Zynq-7000 series SoCs, in use across the industry. The main objective is then to create a new BSP for AIR. The code developed for this new BSP must take into account what the PMK needs and must conform with the API it is expecting and already in place in the SPARC BSPs. Since the BSP for the Arty Z7 must be contained within the already established API calls, the approach will be to migrate whenever possible the code already developed for SPARC. To realize this step, a comparative study will be performed between the two architectures. Once the architectural differences between the two architectures are found, the code will take two lines of development: the initialization processes required to launch AIR on an ARM BSP, and the necessary procedures for AIR to perform its hypervisor responsibilities. Considering that the purpose of this thesis is to increase the number of boards AIR can run on, the code developed will separate between what belongs to the architecture and what belongs to the board. While these two together are the BSP, it facilitates the development of new BSPs for ARM, since the architectural abstractions will be already in place. Still within the bounds of this objective, some code will also be written for the toolchain responsible in compiling AIR.

In addition to the BSP, this work also aims at creating a barebones OS to test the functionalities of the hypervisor in ARM. Given that an OS (or its BSP) is developed for a specific board, the library that implements the API offered by AIR to help in the OS virtualization effort will also need be ported to ARM, since it is compiled alongside the OS and it is written in assembly.

The development in ARM will take place between a QEMU emulator and an Arty Z7, a Zynq-7010 SoC implementation by Digilent provided by GMV.

1.3. Results

This thesis resulted in AIR correctly running in the Arty Z7 with the same behaviour as found in a SPARC board. Given ARM's potential for higher clock speeds, this results into an inherent gain in performance. The final BSP retains all functionalities offered by the SPARC one.

The comparative study performed between the two architectures is also original work, and can be used as reference material when porting a software (without the need to be a hypervisor) from SPARC version 8 into ARM version 7.

1.4. Thesis outline

This thesis is divided into 7 chapters. The present chapter introduces the environment in which this work takes places, the motivation behind it, the objectives it aspires to accomplish, the approaches taken in doing it and the result that derived from it. Chapter 2 starts by given a concise historical background, it continues by reviewing the area of computing in which this work is inserted, the problem it solves, the possible routes that could be taken and finishes by building onto the solution to the previous problem, with regard to the characteristics of the environment it is expected to perform in. Chapter 3 presents the technologies used in the development of this work, starting with an overview of AIR, followed by a summary of the relevant characteristics of both SPARC and ARM. Chapter 4 builds upon the architectures studied in the previous chapter and describes in-depth the differences found between them. Chapter 5 continues chapter 4 and presents the differences found between architectures, but this time specific to the development of a hypervisor. Chapter 6 exhibits the tests performed on the new BSP. Chapter 7 finishes by drawing conclusions from the tests performed, analyzing the contributions of this thesis and giving insight on future developments that could contribute to the growth of the current achievements.

2. Literature review

Hypervisors are the key enabler of time and space partitioning (TSP) systems. They are relevant in all architectures that require complete isolation between applications and strong fault tolerance. Fundamental in the Integrated Modular Avionics (IMA) concept, their predominance in the aeronautical field has extended to the space domain. But first, the reason for the need of the hypervisor is presented, followed by a review on the branch of computing that enables several applications to run concurrently, with the illusion of complete hardware availability.

Section 2.1 introduces the history on the early interest on hypervisors within the aeronautics industry and its transition to space. In section 2.2 is shown how virtualization came to be, the problem it solves, its categories and where does the hypervisor fit. Afterwards, in section 2.3, the attributes of a space-graded hypervisor are clarified.

2.1. Historical background

As early as microprocessors entered the industrial applications market, embedded systems have entered the aeronautic world [15], [16]. Led by high demands on safety and dependability, allied with economic restraints, onboard data processing has become more computer centred. The increase in aeronautic electronics (avionics) performance shows similarities to Moore's Law [17], which allows for more performant onboard software (OBSW), but the rise in the number of avionic functions has also lead to a vast expansion in the amount of memory necessary to hold all OBSWs and the communications between different systems. This has paved the way to very complex systems.

Until the 1990s, commercial aircraft design conformed to the federated architecture [15], [16]. This systems architecture has at its core the principle of "one function = one computer". This hardware isolation into "black boxes", named line-replaceable units (LRUs), leads to strong fault-containment. However, given the exponential increase in avionic functions, the federated approach in result gave an identical increase in the number of LRUs. Consequently, it increased the weight and volume to values that overstepped the natural limitations of aircrafts [18]. Additionally, problems from high power consumptions and elevated maintenance and storage costs due to the excessive number of different LRUs were also manifested.

As a consequence of these constraints, a new systems architecture was sought after. The evident solution was to integrate multiple avionic functions (modules) into a single hardware unit. This new concept is called IMA. The IMA architecture was first implemented in military aircrafts, such as the F-16 [19]. Shortly after, the first commercial applications appeared. Boeing was the first, by using an

IMA architecture on the cockpit of their new 777 aircraft [20]. Airbus followed, by using an entire IMA approach to the design of the A380, and coining the term Open-IMA, where third party avionic suppliers could manufacture components for the A380 by following the Airbus published API [15], [16].

From 1992 onwards, the Airlines Electronic Engineering Committee (AEEC) published a series of standards defining this new architecture and its interfaces [21]. The most relevant of these standards for the work ahead is the ARINC 653 - Avionics Application Software Standard Interface [22], [23]. This standard outlines the multiple properties of the OS running on the LRUs, now hosting multiple avionic functions, and the application executive (APEX), a standardized interface that enables the application's software to be developed independently from the OS, as illustrated in Figure 2.1. It provides the applications' manufacturers portability, by removing language and hardware dependences; reusability, by reducing the customization effort for each platform; modularity, by separating hardware and software dependencies; and integration of software with multiple criticalities, further clarified in section 2.3.3. It was first issued in 1996 consisting of only one document, but since its initial version it has been updated three times so far, and is now divided into five different documents. These range from the required services of the OS to the recommended capabilities, going through the extended and subset services and conformity tests. The first of these documents, Part 1 - Required Services, specifies the baseline operating environment that the OS must provide to follow an IMA approach and has as its primary objective to define the APEX interface between avionic functions and the OS.

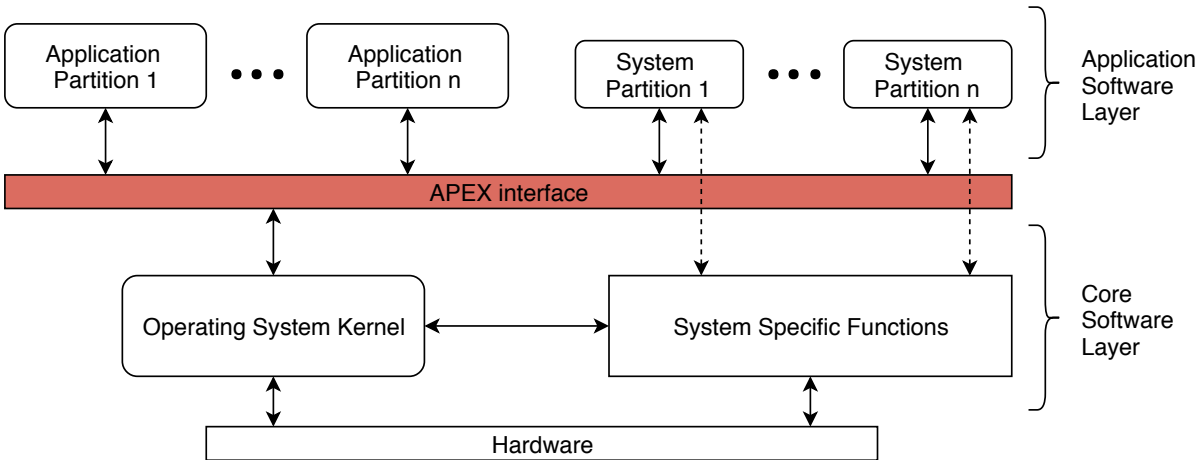


Figure 2.1.: ARINC 653 system architecture overview

Fuelled by the success of the IMA architecture on commercial aircraft, the space community tunnelled their efforts on creating an equivalent for space avionics [24]. The Consultative Committee for Space Data Systems (CCSDS), a multi-national forum founded in 1982 by the major space agencies, addressed the problem by defining the Spacecraft Onboard Interface Services (SOIS) standard, which distinguishes from mission-specific application blocks and the generic execution blocks, such has the OS or the input/output (I/O) handling, and focus on the communications between them [25].

In 2010, the IMA for Space (IMA-SP) project kicks off [26]. Phase 1 of the project was to deliver proofs-of-concept for the IMA concept on space. Three contenders existed at the time. pikeOS from SYSGO [27], with an aeronautical background and already implemented on the Airbus A350; XtratuM

developed by the *Universidad Politécnica de Valencia* [28], [29], based on the ARINC 653 and with space application in view; and finally AIR, from GMV [30], [31], based on the Real-Time Executive for Multiprocessor Systems (RTEMS)[1], a Real Time Operating System (RTOS) designed for embedded systems and with a qualified version for space deployment, the RTEMS improved developed by Edisoft [32].

At the same time, ESA and its space partners have been promoting the Space AVionics Open Interface aRchitecture (SAVOIR) project [33], [34]. SAVOIR is an initiative to improve the way the various space avionics industries integrate their products. By defining reference architectures and standardised interfaces, it streamlines the development, validation and integration phases of the various avionic components. In 2010, the SAVOIR-FAIRE sub-group, tasked with defining the software reference architecture, presented an architecture very similar to the one in Figure 2.1, recognizing the importance of the separation between an execution platform (OS kernel) and the applications running on top [35]. In 2012, the SAVOIR committee launches a sub-group named SAVOIR-IMA. IMA-SP was then reaching phase 2: development of the System Execution Platforms (SEPs), the interface between OS and applications, an equivalent to the APEX, and prototyping of the I/O handling software. The SAVOIR-IMA is now responsible for integrating IMA-SP into the already defined reference architectures and terminologies. All of the work previously done has been continually transferred into the European Cooperation for Space Standardization (ECSS) standards published by ESA. Software development expected to be qualified for space deployment needs to be compliant with the ECSS-Q-ST-80C [36] quality standard and the ECSS-E-ST-40C [37] engineering standard. An additional handbook is available that describes and recommends how to organize and develop software for space, complaint with the ECSS-E-ST-40C, published as the ECSS-E-HB-40A [38].

2.2. Virtualization

Originally as a means to overcome the shortcomings of the typical privileged/non-privileged architecture from a multi-programming perspective, investigation on virtual machines (VMs) gained traction during the late 1960's - early 1970's. In a privileged/non-privileged architecture, all instructions are available to software running in privileged (supervisor) mode, whereas only a subset of the instructions are available to non-privileged (user) programs. The non-privileged programs in turn make system calls, typically implemented as supervisor call (SVC) instructions, to the privileged software, usually an OS kernel, that performs the privileged actions on their behalf, possibly checking if the calling non-privileged program has the rights to access the requested functionality. If a non-privileged program attempts to execute privileged instructions, an exception is raised and the program flow is disrupted. The execution jumps to a pre-registered table of exception handlers, normally controlled by the OS. The set of non-privileged instructions plus the system calls offered by the privileged software kernel present an extended view of the system to the user [39], as shown in Figure 2.2.

While this extended model is successful in many computer applications, it has limitations. Although the extended system is replicated for each user program as demonstrated in Figure 2.2, only one OS can be

[1] `https://www.rtems.org/`

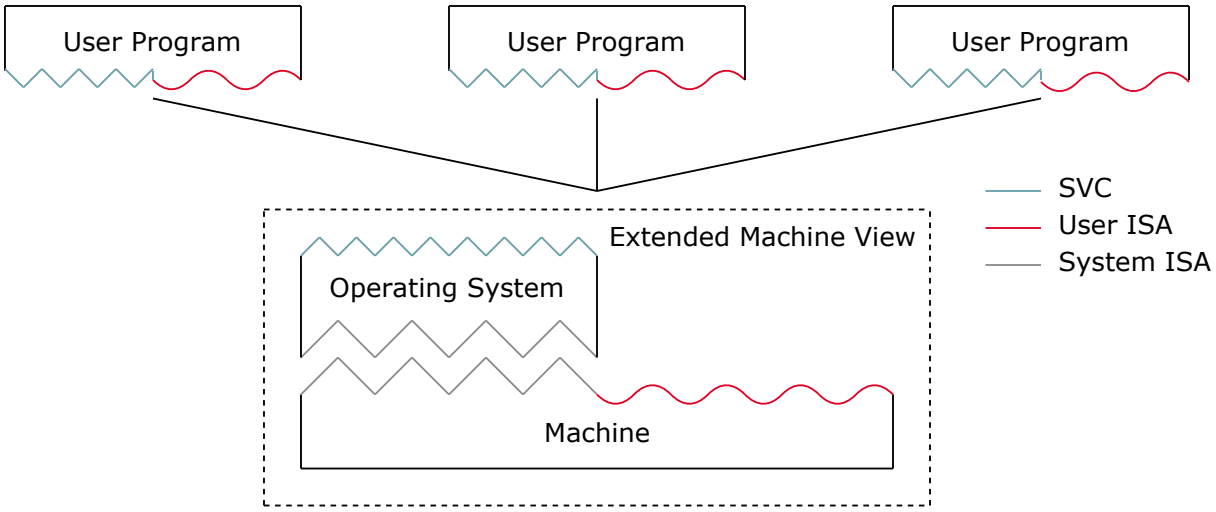


Figure 2.2.: Extended machine view exposed to non-privileged programs

running at a time. This means that it is not only impossible to run more than one OS, but also denies the possibility of running any program that requires direct access to the privileged instructions. Furthermore, it occurs the inability to support older applications designed for older OSs, to modify and test other OSs and to run test and diagnostic programs that require direct access to privileged instructions, without taking down the machine and disconnecting all the users that were logged in [39].

The crucial innovation of virtualization is the introduction of a virtual machine monitor (VMM) that provides the illusion of multiple hardware infrastructures. Each VM behaves as a replica of the original machine, including its ISA and system resources, such as the central processing unit (CPU) state and memory and I/O accesses. This is the first property of interest when analyzing a VMM, also known as equivalence. Equivalence can be stated as: any program executing in a machine without a VMM should behave identically when running in the same machine virtualized by the VMM. The only exceptions to this principle of equivalence are the CPU timings and total resource availability, which can not be maintained for the second case, since the VMM must split resources across all replicas. Each OS now operates on top of a VM instead of a real one, as illustrated in Figure 2.3. The possibility of running more than one OS at once, along with the illusion of the entire hardware availability, offers a very robust multi-programming interface, as long as the VMM manages to run all virtual machines concurrently and keep complete control of the virtualized resources. By making full use of the resources of more powerful computers, a better efficiency is attainable by permitting multiple users to access the machine at once, each in its own VM, thus offering the possibility of a time-sharing system [40].

2.2.1. Critical instructions

For the VMM to work as expected in a privileged/non-privileged architecture, without the OS meddling in its execution, the entire VM must run in user mode, containing both the OS and user applications, letting the supervisor mode entirely as the VMM's playground. The VMM keeps additional information on the privileged level of the instructions carried out by the VMs, if their were executing natively on the machine without the VMM presence, so that it can be used later by the VMM to simulate the behaviour

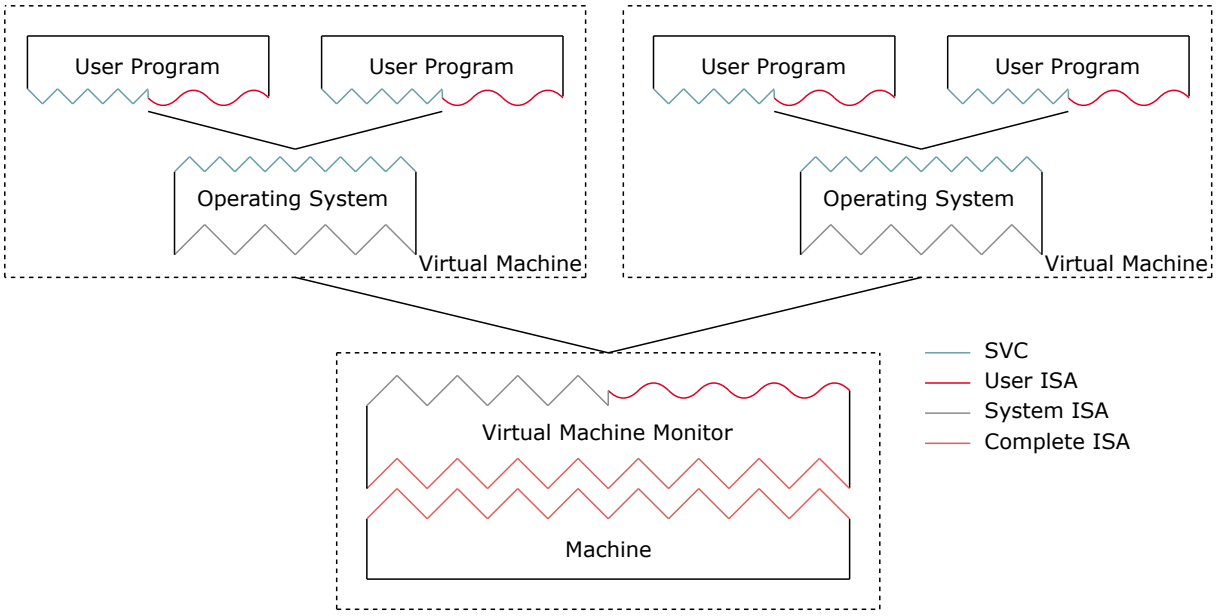


Figure 2.3.: Virtual machine abstraction to the operating systems

of those instruction depending on their original privilege level [41]. Since the VMM is now running at a higher privileged level that the OS, which would normally run at supervisor level, the term hypervisor is also used as an alternative to VMM, and will be the one used henceforth in the rest of the dissertation.

Regardless of being privileged or non-privileged, instructions can be separated into two different sets according to their dependency on the present state of the system. The first is the sensitive instructions set, which groups all instructions that either change or are dependent on the state of the system. A non-exhaustive list on the purpose of these instructions follows:

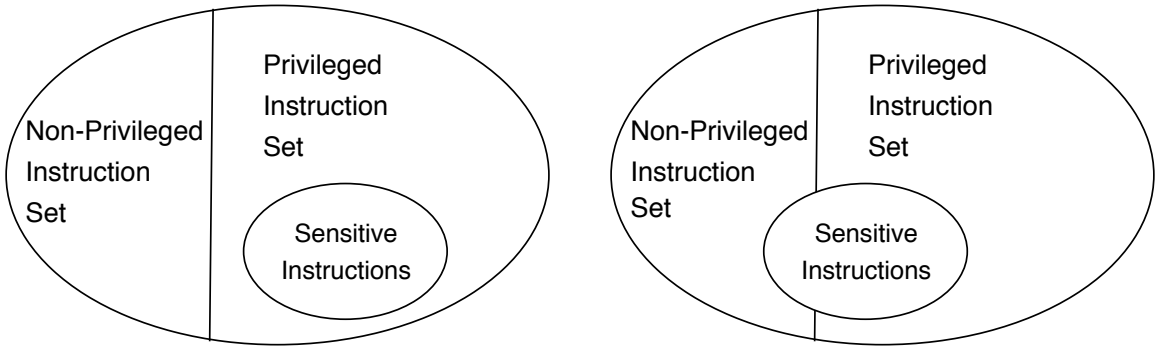
- change the current privilege level;
- read the hardware value of the privilege level in user, instead of the copy virtualized by the hypervisor;
- change a system configuration, such as enabling/disabling the FPU or a system timer;
- read a system configuration as user, such as the core id, also virtualized by the hypervisor;
- change memory configurations, i.e. creating, modifying or destroying page tables;
- access I/O devices directly;
- mask/unmask interrupts;
- halt execution while waiting for an event.

Innocuous instructions comprise every other instruction that does not fall into the previous category and represents the second set. To respect the second property of a hypervisor, complete resource control, all sensitive instructions must be run by the hypervisor and any attempt from user level programs to run them must raise an exception to the hypervisor. This effectively prevents the programs running in the underlying VMs to access or modify resources and time shares not allocated to them.

The third and final property of a hypervisor is efficiency. To allow an efficient construction of a hypervisor, the grand majority of instructions need to run natively on the CPU without hypervisor interference. The instructions that are oblivious to the machine state and do not alter the system configurations, the

available memory or the I/O resources, can run natively in hardware, while the other ones must either be executed by the hypervisor or with its consent, generally being simulated in supervisor mode taking into account their original privileged level previously saved by the hypervisor.

This second group of instructions that have consequences on the forthcoming state of the machine is the same group as the previously denominated sensitive instruction set. To safeguard that the sensitive instructions are executed by the hypervisor, they must be a subset of the privileged instructions [42], as seen in Figure 2.4a. Since the OS inside a VM is running in user mode, whenever it executes a privileged instruction, the execution is trapped into the hypervisor. If the instruction is sensitive, then the hypervisor simulates it in software, updating the respective VM CPU registers and memory boundaries saved in software structures, returning back to the OS after handling the exception. This last property of a hypervisor is the hardest to achieve, as not all architectures are virtualization friendly, having both sensitive and non-privileged instructions, as seen in Figure 2.4b. The instructions in the intersection between the sensitive and non-privileged sets are often called critical instructions. If a sensitive instruction is not trapped into the hypervisor, then unpredictable behaviour will ensue, and the second property, complete resource control, will be void, with a high likelihood of leading the other VMs to experience faults.



(a) without critical instructions (b) including critical instructions

Figure 2.4.: Sensitive instructions

The three properties presented previously are also known as Popek and Goldberg virtualization requirements for an efficient hypervisor [42]. To sum up:

EQUIVALENCE An application running on top of a hypervisor must run as if there was no hypervisor present, with two exceptions: reduced time slice and available memory.

RESOURCE CONTROL An application cannot alter the system resources allocated to it.

EFFICIENCY All innocuous instructions must be executed directly on the hardware without hypervisor intervention.

These properties offer an insightful view on what is a virtualization-friendly architecture, where the last two properties entail an architecture without critical instructions.

There are two possible solutions to the critical instructions problem, full virtualization and paravirtualization, either with its advantages and disadvantages. These do not offer a virtualization as efficient

as in the case of Popek and Goldberg's efficient hypervisor, where there are no critical instructions, but since there are plenty of typical privileged/non-privileged architectures that feature such instructions, alternatives had to be found to host a hypervisor and achieve correct virtualization.

2.2.2. Full Virtualization

The first of the alternatives is full virtualization, where the hypervisor provides the unmodified VMs a complete copy of the bare machine. The guest OS is unware that it is running inside a VM. It is also referred to as hardware virtualization, and as the name suggests, it requires hardware modifications. The modification is normally performed by extensions to the original processor, such as the AMD-V [43], Intel's VT-x [44] and ARM's Virtualization Extensions [45]. All three aforementioned extensions achieve correct virtualization in a similar way.

First, an additional privileged level is added where the hypervisor resides, with the guest OSs running on each VM back to supervisor mode. The instruction set is extended with instructions that jump in and out of the new higher privilege level, saving the VM's state in memory before restoring the following VM's state. In addition, the hypervisor either traps sensitive instruction sequences, including accesses to system information that could be considered sensitive, or duplicates these locations in memory, offering a different copy to the hypervisor and to the multiple OSs running in supervisor, and allowing the latter to be modified by the hypervisor. Secondly, to expedite memory translation and segregation in VMs, an additional hardware level of translation is added to the MMU, as the guest OS will perform an intermediate translation (unaware that the hypervisor is running and it holds the actual physical addresses), with the hypervisor performing the final translation to physical. Caches are also modified to hold information on which VM their entries belong to. Thirdly, all interrupt requests (IRQs) can first be routed by the hypervisor and masked when needed. The last problem is how to handle the I/O, which is always architecture and hypervisor specific.

Albeit hardware extensions are the only form of full virtualization, since the original OS+user programs are running natively without modification, with the only overhead on sensitive instructions, an approximate form of full virtualization can be achieved, at the cost of performance, through emulation. VMware managed to virtualize the x86 architecture before VT-x using binary translation [46], [47], a variety of emulation. The hypervisor employs binary translation by going through every instruction at least once and replacing an instruction if it finds it to be sensitive. While this method provides a big initial overhead, the process can be optimized [48]. Binary translation can also be employed when the original code is not compiled for the target architecture, and can be resorted to in combination with the hardware extensions.

2.2.3. Paravirtualization

The second alternative is paravirtualization. In this virtualization method, the guest OS is aware that it is running inside a VM and is modified (or created from scratch) to interact with the hypervisor [49], [50]. It accomplishes the communication with the hypervisor through HVCs, the equivalent of a SVC when user applications request the OS to perform an action on their behalf, but in this case the OS requests

something from the hypervisor. This method eliminates the critical instructions by replacing them with HVCs and removes the overhead imposed in the hypervisor from discerning the current context of the sensitive instruction attempted in the VM, whenever an exception is raised. The guest OS now has access to an API of HVCs offered by the hypervisor to help in the virtualization effort.

Replacing sensitive code segments with HVCs removes the otherwise occurring exceptions with near function behaviour, with the ability to pass arguments and receive values from the hypervisor. However, as extended kernel modifications are warranted and have to be maintained after updates, greater maintenance costs are associated with it. It also requires the OS availability for customization and since some OSs are proprietary they cannot be freely altered.

Despite the greater maintenance costs, it provides the fastest post-virtualization system. Changing the OSs source-code and recompiling it to interact with the hypervisor natively offers little to no overhead on the OS, notwithstanding that each VM must respect the hypervisor's imposed schedule and may have to share CPU time with other VMs.

Type-2 Hypervisors

Besides the previously mentioned virtualization techniques, which introduce the hypervisor executing on top of the hardware, virtualization research is an extending field which builds upon these early concepts. With the above hypervisors classified as type-1, native or bare-metal hypervisors, that run directly on the hardware, there also exist type-2 or hosted hypervisors, that manage VMs as processes on top of an OS. Complex architectures can be built from these two types of hypervisors, using type-2 hypervisors on top of an OS already on top of a type-1 hypervisor. OSs themselves offer virtualization, also known as containerization [51], [52], recently popularized by Docker[2], which utilizes the OS resources to offer isolation between containers without the need of the additional OS used by VMs. A container is composed of multiple applications, tools and libraries, functioning in complete isolation from other containers. This approach enables different versions of software designed for different versions of the same library to run concurrently, which was one of the earlier requirements for virtualization, all without the overhead of deploying several OSs.

2.3. Hypervisor

As a consequence of the success of the IMA architecture in the aeronautical field, a similar approach has transitioned into the space domain. The hypervisor in an IMA architecture can be boiled down into two fundamental notions, TSP and fault detection, isolation and recovery (FDIR). By taking advantage of the existing regulations such as the ARINC 653 avionics standard, a framework for the OBC OS can be designed using well established and tested principles [53]. Incorporating these two notions into a type-1 hypervisor creates the foundations for a space-graded hypervisor.

[2] `https://www.docker.com/`

2.3.1. Time and Space Partitioning

As described in ARINC 653 (Avionics Application Software Standard Interface) - Part 1 - Required Services [22], the central philosophy of an IMA system is partitioning, where applications are segregated with respect to space (memory partitioning) and time (temporal partitioning). This is useful both for fault containment and for ease of verification, validation and certification. The base unit of partitioning is a partition. Partitions can range from a RTOS plus applications to a single bare-metal OS and a single-threaded program, each having their own data, context, permissions, etc. This notion of partition behaves very much alike that of a VM in the context of virtualization. Partitions are limited to using only the system calls defined in the published APEX. To circumvent the restricted operations permitted by the ARINC APEX, the standard allows an optional partition type, the system partition, that can use additional system calls to the hypervisor, for instance, to manage I/O device drivers. The system partition still needs to conform by the robust temporal and spatial partitioning. The standard also recognizes the concept of multiple partitions belonging to the same component, called modules.

The hypervisor is responsible for enforcing the partitioning, making sure that different partitions are completely contained from each other spatially and temporally. It is also responsible for handling any errors that may arise from the partitions without interfering with other partitions in the system, i.e., errors resulting from a partition are handled during that partition's time slice.

Temporal partitioning is attained by following a fixed and cyclic schedule, determined before deployment, which ensures deterministic behaviour. The schedule uses as the base unit of time a major time frame. These have a fixed duration, and are repeated periodically. The major time frame is composed of smaller frames equal in size, which are allocated for the partitions. At the end of each minor time frame, the hypervisor retakes control of the execution and determines whether the following minor time frame is alloted for the same partition as the previous. If it is, then is resumes the partition, and if not, it saves the previous application context and restores the following one, passing the control to the succeeding application. The period between each minor time frame also determines the precision of the wall clock offered by the hypervisor to the partitions. An example of a schedule of an OBSW in a multicore configuration can be seen in Figure 2.5. Each major time frame lasts $250\,\mathrm{ms}$ and is composed of 10 minor time frames of $25\,\mathrm{ms}$ each. Each partition has a number of tasks or processes running during its execution, but each partition's processes are managed by that partition scheduler and are outside of the scope of the hypervisor. Major time frames are repeated until the board shuts down.

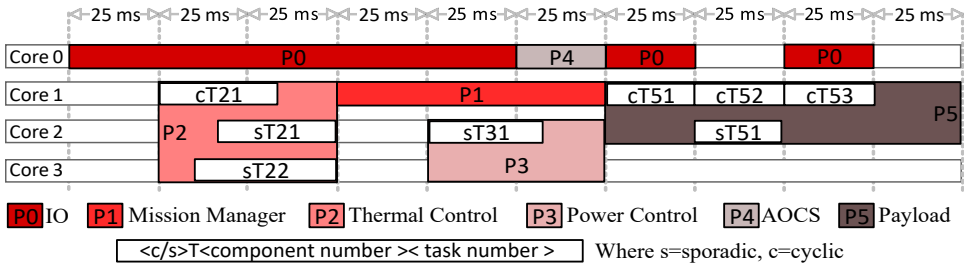


Figure 2.5.: SMP partitions schedule

Spacial partitioning is guaranteed with predetermined areas of memory allocated for each partition

at compilation time. The space partitioning can be enforced by whatever mechanism is present in the available hardware, such as a MMU or a memory protection unit (MPU), or by virtualizing every single store/load instruction, albeit at great performance costs. During each partition's execution time, memory access outside of the assigned memory areas is prohibited. Communication with other partitions or I/O devices must be done by requesting ARINC 653 inter-partition services, such as sampling and queuing ports, provisioned by the hypervisor.

How the hypervisor ensures temporal and spatial partitioning is left to the system designer, as long as it achieves robust partitioning. While these two concepts are applicable to space missions, the operational context in which the software runs is entirely different, both in the components life cycle as well as in the maintenance and upgrade mechanisms, given that aircrafts' maintenance is done on ground, while satellites' must be done during operation in space. For the ARINC 653 Part 1 to be applied in the space context, a few extensions to the base model are required. ARINC 653 Part 2 - Extended Services [23] features a service to handle multiple partition schedules, which is determinant in space missions to manage different modes of operation and phases of a mission [31], and needs to be incorporated. Communication with I/O devices can be achieved by using an accessory system partition which would conduct all messages between partitions and the devices. This partition performs all communications with the external devices and conveys the information back to the partitions through the hypervisor created ports [54].

2.3.2. Fault detection, isolation and recovery

The second major concept of the IMA architecture is FDIR. ARINC 653 defines the existence of a HM for monitoring and reporting errors in the entire system, experienced either in hardware or raised by the partitions. The HM foresees a HM table at the hypervisor level with HM callbacks to respond to pre-defined faults in the system and optional HM tables at the partition level to handle partition level errors, one for each partition. The notion of FDIR entails three stages during the course of a fault.

Detection

Starting out with detection, faults may be detected either:

- in hardware, such as memory violations, privileged execution violations, overflows, timer interrupts and other I/Os;
- in the hypervisor, such as configuration errors and missed deadlines;
- by the partition OS or similar software, such as wrong sensor readings throwed as errors.

The particular list of all errors is implementation specific, as well as where they are detected. Every possible error in the system must be identified prior to the operation of the device and assigned a specific id.

To help categorize all errors in the system and to determine the appropriate HM callbacks, ARINC 653 defines three levels where an error may occur and their inherent impact, shown in Table 2.1.

It is important to state that errors in the hypervisor are not provisioned in the ARINC 653. However,

Level	Impact
process	one or more processes in a partition, entire partition in the worst case
partition	only one partition
module	every partition in the affected module

Table 2.1.: ARINC HM error levels

these errors must still be consistently handled in a complete and recoverable manner, but are outside of the scope of the standard and are the responsibility of the system integrator.

In addition to the id of the occurring fault, the operational state of the system is also taken into consideration (such as module/partition/process initialization, module/partition/process execution, partition switching, etc.) for establishing the level of the fault and the correct HM Table. The operational state is set and kept up-to-date by the hypervisor between each exchange in state. The three parameters in combination will determine the appropriate fault handler to be launched.

Isolation

After a fault is detected, it must be kept isolated from the rest of the system. The first stage in a fault isolation after being detected is the selection of the correct HM callback, which also determines the degree of isolation of the fault. This is performed by extracting the error id and operational stage of where the fault occurred. Using these two parameters, the correct error level can be derived and the appropriate HM Table is used along the error id to select the proper HM callback.

If the HM Table used is the partition level one, then the fault is handled during that partition time slice, ensuring isolation from the rest of the system. The entire HM callback is performed within that partition's context and will relinquish control when another partition is due. If the error affects an entire module and the Module HM table is the one used, then isolation is kept only within that module's partition, still respecting another module's time slices. All partitions within the faulty module are affected.

Process level faults are also handled during the erroneous partition time slice, but are implementation dependent. These are normally handled by a higher priority task within the partition OS. If a fault is experienced during the process error handler, it becomes a partition level fault and is handled appropriately taken into account that it already faulted once before.

Recovery

Following the fault detection and isolation comes recovery. Each entry of the chosen HM Table is accessed through the error id and the operation state and contains a HM Callback to recover from the pre-defined fault. There are three staple actions that can be taken for all levels of errors, stop, restart and ignore. Additional actions can be implemented through additional functions to deal with every possible error in the system.

2.3.3. Mixed Criticality

Associated with the possibility of a fault during each application's execution, is the level of criticality of each of those applications. Different tasks will have different levels of criticality based on the level of assurance needed against system failure, depending on how severe the consequences are. A ranking system of the level of criticality has been published by ESA in the ECSS-Q-ST-30-02C standard [55], shown in Table 2.2.

Name	Level	Consequences
Catastrophic	1	Loss of life, life-threatening or permanently disabling injury of occupation illness Loss of system Loss of an interfacing manned flight system Loss of launch site facilities Severe detrimental environment effects
Critical	2	Loss of mission Temporarily disabling but not life-threatening injury or temporary occupational illness Major damage to an interfacing flight system Major damage to ground facilities Major damage to public or private property Major detrimental environmental effects
Major	3	Major mission degradation
Minor or Negligible	4	Minor mission degradation or any other effect

Table 2.2.: Severity categories

The advent of more speculative computer architectures, with the introduction of instruction and data caches, branch predictors and instruction pipelines, with the purpose of increasing the system performance, has accrued on the unpredictability of those same systems. The timing analysis of a system can now only be calculated with a lower than 100% level of certainty, due to cache hit-miss ratios and multiple bus travels. An important attribute of a timing analysis is the calculation of multiple worst-case execution times (WCETs) associated with their probability degree [56]. An application's WCET will vary depending on the criticality level. As the level goes higher (lower value), the more conservative a WCET for a task is.

While varying this value has drastic influence on the schedulability analysis, as long as the hypervisor supports multiple schedules, as well as a fault-handling system, the criticality value of the different applications has little influence on the hypervisor itself.

3. Technologies

This chapter introduces the technologies used in this thesis. It first presents an overview of the AIR hypervisor by GMV and details the particularities of this specific hypervisor. Afterwards, both SPARC and ARM architectures are studied from the point-of-view of the hypervisor programmer.

3.1. AIR

The AIR hypervisor by GMV appeared has an adaptation of an open-source RTOS to incorporate the ARINC 653 standard required services [30], [31], within the scope of ESA's Technology Harmonization[1] effort. RTEMS was chosen due to its certification for unmanned space missions [57], a fundamental asset for the operational function of the hypervisor being developed. While modification of the RTEMS' kernel voids the certification, it lays the foundations for a well designed hypervisor. RTEMS also features a modular approach to its code base and supports a wide range of processors, such as the SPARC's ERC32 and LEON and ARM's SoCs, which can be adapted on a per case basis, removing some of the initial costs of development.

The result of this work is AIR, which stands for the ARINC 653 Interface for RTEMS. AIR is a TSP hypervisor fully compliant with the ARINC 653 standard, enforcing complete isolation between partitions that follows the paravirtualization methodology of virtualization, currently supporting as partition OS an improved version of RTEMS 4.8, tailored specifically for space projects. Development on the current version of RTEMS is still an ongoing effort. The partition OS set is also supplemented with a bare-C OS and several libraries to facilitate testing on a new architecture for which RTEMS has not yet been ported to AIR. AIR has been continually developed by GMV under several project both in airspace and space, leading to a somewhat extensive code base with multiple increments along the years. It is maintained under the open-source GPLv2 licence[2].

In addition to the hypervisor and the partition OSs, AIR is complemented with a tool suite comprised of LIBAIR, a library designed to virtualize the partition OS with all the relevant system calls, IMASPEX, a collection of the services described in the ARINC 653 and used as AIR's APEX, LIBIOP, a library that configures an extra system partition to deal with peripheral I/Os devices, a python configuration tool to assist in the compilation process and a partition assembler to compress and build the applications into the final executable.

[1] `https://www.esa.int/Our_Activities/Space_Engineering_Technology/Technology_Harmonisation`
[2] `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`

3.1.1. AIR architecture

The AIR architecture resembles that of ARINC's presented in Figure 2.1. The final output of the build process is structured as in Figure 3.1. On top of the hardware is the BSP for each board. While this section is also developed in-house, it is intended to be entirely independent from the hypervisor logic. However, there is still some intertwine between the BSP and the hypervisor, being the complete separation one of the objectives of this work. On top of the BSP is AIR, the hypervisor, where all the logic related to temporal and spatial separation is performed, along with the initialization of the system, scheduling, context-switching and housekeeping. AIR is also referred to as the PMK, and the terms are used interchangeably. On top of AIR are all the partitions that are scheduled to run during the system's runtime. The libraries selected in each partition, including LIBAIR, are built separately for each partition, since there is no shared space between the partitions. The Partition Operating System (POS) is then left to its own execution, with the ability to schedule tasks, setup filesystems, etc.

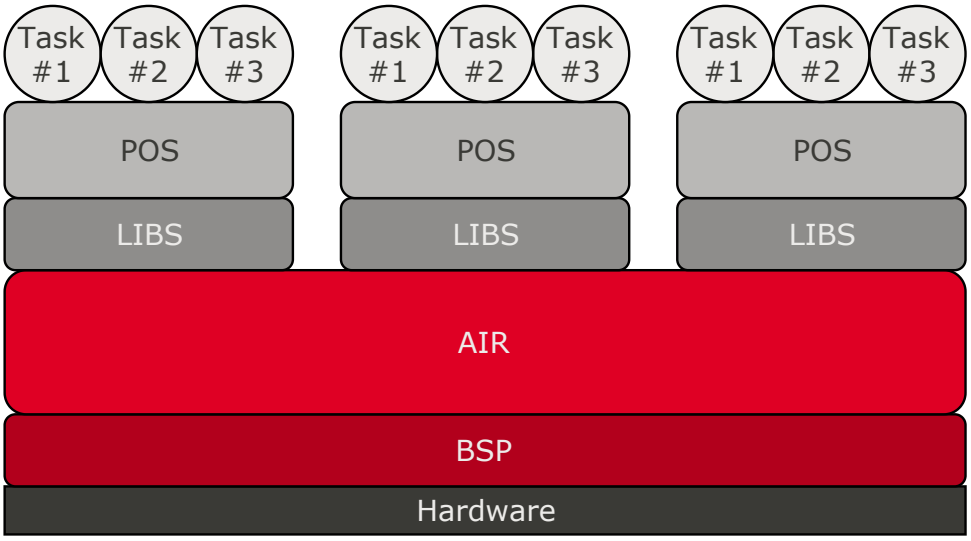


Figure 3.1.: AIR architecture

3.1.2. System initialization and execution

Two files are involved in the initialization of AIR in any board, the `start.S` and the `init.c`. The `start.S` is an assembly file required in all bare-bones software and is architecture-dependent and software-independent. As such it can be seen as part of the BSP box in Figure 3.1. It is responsible for:

- allocating space for the board exception vector, either by filling the space with dummy function handlers or jumps to the actual exception handlers;
- clearing the **.bss** [3];
- setting up the stack pointers;
- invalidate all cache entries, the translation lookaside buffer (TLB) and other speculative mechanisms;

[3] Historically called the Block Started by Symbol, the **.bss** contains all the statically-allocated uninitialized variables.

- configuring the FPU, if enabled.

After the previous procedures the `start.S` calls the `pmk_init()` function, located in the `init.c`, which can be seen in Figure A.1. From here onwards, the remainder of the system initialization is carried from AIR, with the appropriate calls to the BSP where required.

3.1.3. Time and space partitioning

As a TSP hypervisor, AIR is responsible not only for maintaining the VMs, but also for enforcing temporal and spatial isolation. After the initialization is complete, AIR launches a partition responsible for maintaining the core in idle and enables preemption.

The temporal correctness is maintained by reserving a timer for only supervisor access, triggering an interrupt when it reaches the desired counter value, and auto-restarting, thus generating interrupts at regular intervals. The timer interrupt is raised at the beginning of each minor frame and the associated handler is dispatched. This handler is responsible for saving the previous partition's context, checking what partition is allocated for the current minor frame, and restoring the succeeding application's context. The overhead introduced by AIR has been evaluated to $1 \sim 2\%$ in single core while running on the ESA Next Generation Microprocessor (NGMP) [4], and grows inversely proportional to the duration of the minor time frame, deteriorating in performance for periods of under $0.001\,\mathrm{s}$ [58]. There are yet no extensive studies as to the overhead present in a multicore scenario.

The spatial isolation is preserved using the hardware structures of the underlying hardware, therefore it is architecture dependent. So far, AIR has only been developed expecting a MMU, but it keeps the separation between hypervisor and BSP by calling generic functions such as the one in Figure 3.2, that can de adapted for every architecture.

```
void cpu_segregation_map_memory(
      cpu_mmu_context_t *ctrl, void *p_addr, void *v_addr,
      air_sz_t size, air_sz_t unit, air_u32_t permissions)
```

Figure 3.2.: BSP function that fills the necessary MMU tables

3.1.4. Health Monitor

In order to respect the ARINC 653 standard, AIR also realizes the concept of a HM. It follows the same approach as detailed in Section 2.3.2, and the actual implementation can be visualized in Figure 3.3. After a fault is experienced, an exception is raised and the execution jumps to the HM handler. It starts by performing a lookup using the error id and operational state in the system HM table to determine the level of the fault. After it has determined the level of the error, it performs a search in the correspondent HM table using the error id, finally performing the pre-determined action for the detected fault. The actions allowed in a module level error are SHUTDOWN, RESTART and IGNORE, where IGNORE later calls a function handler specified for the experienced error. The partition level table also offers the SHUTDOWN and IGNORE actions, with similar behaviour to the module HM table, but distinguishes between a

[4] `http://microelectronics.esa.int/gr740/index.html`

COLD_START and a WARM_START. Both the COLD_START and WARM_START can be seen as a complete reset of the partition, and the equivalent to the module RESTART, but the information on which of the two occurred is passed on to the partition, that can later act differently based on that information.

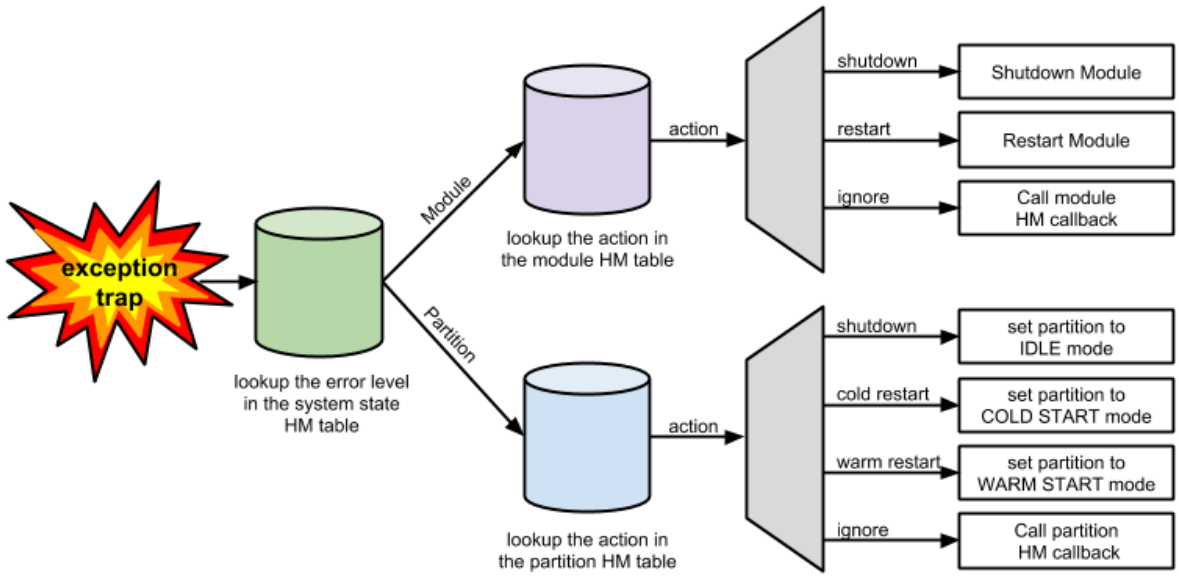


Figure 3.3.: AIR HM procedure

The HM configuration is performed through an extensive table present in a configuration `.xml` that is comprised of the system HM table plus a module HM table and a partition HM table for each partition. A simplified example is presented in Figure A.2. The specific function handlers related to the IGNORE action are placed inside the partition files during each partition's compilation. The error handler is then responsible for retrieving the operational state of where the error occurred and the error id through a system call. An example of a HM handler inside a partition can be seen in Figure A.3. This function can later be modified by the application programmer to handler the errors as he/she sees fit.

3.1.5. AIR toolchain

To help the development effort and streamline the build process, AIR is accompanied by a python toolchain. The toolchain makes use of python libraries to read through AIR's directory structure and generates `Makefiles` in every directory according to the rules specified in the python scripts. It uses the mako template module to generate the `Makefiles` and any additional files by using the library powerful template tools.

The toolchain is organized with a similar structure to the AIR repository to facilitate including additional boards, libraries and POSs. After creating the mako templates, the configuration of the tool is mainly done through the `config.py` files present in each board, library and POS directory. These contain information related to the compilation process, such as the compiler, compiler flags and include directories, and important flags related to the board attributes used in the PMK, POS and libraries code through the use of `#define` directives. The tool is called by using the `configure` script located at the toolchain's root directory. This script is used in two different scopes of the build process.

First, it is used in the outermost scope to generate the PMK, POS and libraries `Makefiles`. In this context, it takes as input the target board, along with the use of hard or soft floating point operations, with the possibility of adding additional options if required down the development line. After the `Makefiles` are generated, unless there is a change in the board's properties in the `config.py` files, the tool is no longer required at this level.

Second, it is used at the application's level after AIR, the required POSs and libraries are compiled. At this level, it takes as input a text file named `config.xml`, that addresses all the configuration options required to compile the application. An example can be seen in Figure 3.4. This file contains information on the partitions' options, the schedules used, the module information and the HM tables, the latter not present in this example. In this example, there is only one module, with one partition, and it takes the entirety of the schedule's available time slice. It also informs that the timer will trigger 10 times per second and it uses a single core. The `configure` script cross-checks the information present here with the attributes defined in the target `config.py` and searches for any syntax error or missing information in the `config.xml`, and only when there are no errors it creates the application's `Makefile` and the additional files generated from the information in the `config.xml` and the mako templates.

```
<?xml version="1.0" encoding="UTF-8"?>
<ARINC_653_Module ModuleName="bare">
   <!-- partition 0 -->
   <Partition PartitionIdentifier="1" PartitionName="p0"
         Criticality="LEVEL_A" SystemPartition="false" EntryPoint="entry_point">
      <PartitionConfiguration Personality="BARE" Cores="1">
         <Libs>LIBAIR;LIBPRINTF</Libs>
         <Cache>CODE; DATA</Cache>
         <Memory Size="0x1000000" />
         <Permissions>
            FPU_CONTROL; CACHE_CONTROL; GLOBAL_TIME; SET_TOD; SET_PARTITION_MODE;
         </Permissions>
      </PartitionConfiguration>
   </Partition>

   <!-- schedule 0 -->
   <Module_Schedule ScheduleIdentifier="1" ScheduleName="test_sched" MajorFrameSeconds="1.0">
      <Partition_Schedule PartitionIdentifier="1" PartitionName="p0"
      PeriodSeconds="1.0" PeriodDurationSeconds="1.0">
         <Window_Schedule WindowIdentifier="1"
               WindowStartSeconds="0.0" WindowDurationSeconds="1.0" PartitionPeriodStart="true"/>
      </Partition_Schedule>
   </Module_Schedule>

   <!-- module configuration -->
   <AIR_Configuration TicksPerSecond="10" RequiredCores="1"/>
</ARINC_653_Module>
```

Figure 3.4.: Application `config.xml` file

3.2. SPARC

First released in 1987, SPARC became one of the most influential early RISC designs with its initial 32 bit architecture, the SPARC V7. The addition of integer multiply and division instructions, among other

improvements, came later in the SPARC V8, incorporated in the SuperSPARC series of processors released in 1992, and became the de facto 32 bit architecture from SPARC. The final installment is the SPARC V9, a 64 bit SPARC architecture released in the following year. Later additions served as extensions for the last two designs, and were mostly introduced based on commercial needs. SPARC saw most of its success on server applications, but with ESA's adoption of the architecture for the ERC32 processor using the SPARC V7, and later with the LEON processors based on the SPARC V8, it will remain in the near future for Europe's deep-space missions. The LEON4 processors is featured in Gaisler Research's (now part of Cobham) GR740 SoC, made available by GMV as AIR's testing bench for its SPARC version. Since the LEON4 is based on the 8th version of SPARC, the rest of the chapter and thesis is only focused on that particular version of the RISC architecture.

3.2.1. Operating Modes

SPARC supports two operating modes: user and supervisor. All instructions and memory are available in supervisor mode. If a privileged instruction or illegal memory access is attempted while in user mode, an exception (section 3.2.4) is generated. The processor status registers can only be written or read directly by supervisor software, however, user software can access certain fields and registers during execution, such as the condition codes on the processor state register (PSR) (section 3.2.2) when executing conditional instructions.

3.2.2. Registers

A SPARC processor includes two types of registers: general-purpose registers and control/status registers. Both the integer unit (IU) and the FPU have their own set of control/status and general-purpose registers.

The IU control/status registers consist of:

- the PSR;
- the Window Invalid Mask (WIM);
- the Trap Base Register (TBR);
- the Multiply/Divide Register;
- the program counter (PC) and nPC;
- implementation-dependent Ancillary State Registers;
- an implementation-dependent IU Deferred-Trap Queue.

The PSR, shown in Figure 3.5, is a representation of the current state of the processor. It includes: the IU four condition codes (*icc* field), negative, zero, overflow and carry; whether the FPU (EF) and the coprocessor (CP) (EC) are enabled; the Processor Interrupt Level (PIL), responsible for determining which interrupts are accepted (section 3.2.4); whether it is in supervisor or user mode (S), along with the previous operating mode (PS), used when returning from exceptions; if traps are enabled (ET); and the current window pointer (CWP), relevant for the general-purpose register windows. The impl and ver fields refer to the specific processor manufacturer.

The WIM keeps information necessary for the general-purpose registers management. The TBR maintains the trap base address. In case of an exception, the execution jumps to the base address plus an offset to calculate the required exception handler. The Multiply/Divide register contains the most significant word of the double-precision result of an integer multiplication or division. This would normally mean either overflow on the multiply or a remainder on the division. The PC and nPC contain the address of the instruction being currently executed and the next instruction to be executed, respectively. The nPC becomes important due to SPARC's delay instruction after all transfer instructions (branches). When executing the delay instruction, the nPC has the value of the transfer instruction target. SPARC can optionally provide up to 31 Ancillary State Registers. It is also optional the implementation of the IU deferred-trap queues.

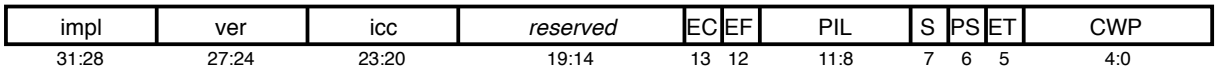


Figure 3.5.: SPARC Program State Register

The IU general-purpose registers are arranged as a circular array of windows. A window is composed of 8 **in**, 8 **local** and 8 **out** registers, plus 8 **global** registers accessible to all windows. One window's **out** registers will overlap into the next window's **in** registers, with the last window overlapping into the first one. The CWP field of the PSR holds the window currently in use and the WIM register the last window that will overlap into the first used one. The CWP is decremented and incremented on SAVE and RESTORE instructions, respectively. These instruction are generally executed on procedure calls and returns. The idea is for the **in** registers to hold the incoming parameters, the **local** registers to perform local calculations and the **out** registers to contain the outgoing values, with the **global** registers having global information that changes little across execution. During a SAVE instruction, the **out** registers of the current window become the **in** registers of the following, thus reducing memory writes and reads to the stack. All this is illustrated in Figure 3.6.

This model has advantages and disadvantages. While it makes normal execution of the program faster during procedures calls and returns, considering that while there is no shortage of windows, there is no need to push all registers into memory, during hypervisor execution, to save the VM state, all windows need to be dumped into memory, nullifying the purpose of the circular array. It also increases the code base size and complexity, as extra functions are required to maintain window coherency.

There are two exceptions associated with the register window's management, the *window_overflow* and *window_underflow*. These are responsible for dumping information on the main memory in the event any of them is raised. A window overflow will happen when the CWP reaches the value in the WIM. This can happen during an application normal life cycle, when function calls reach the number of available windows. The *window_underflow* exception should only happen during context switch, when the hypervisor is going through the windows backwards and dumping them into memory, and it reaches the window after the first window being used by the VM, that equates to the window in the WIM.

The FPU has its own set of 32 general-purpose registers, accessed as Fx, and a Floating-Point State Register.

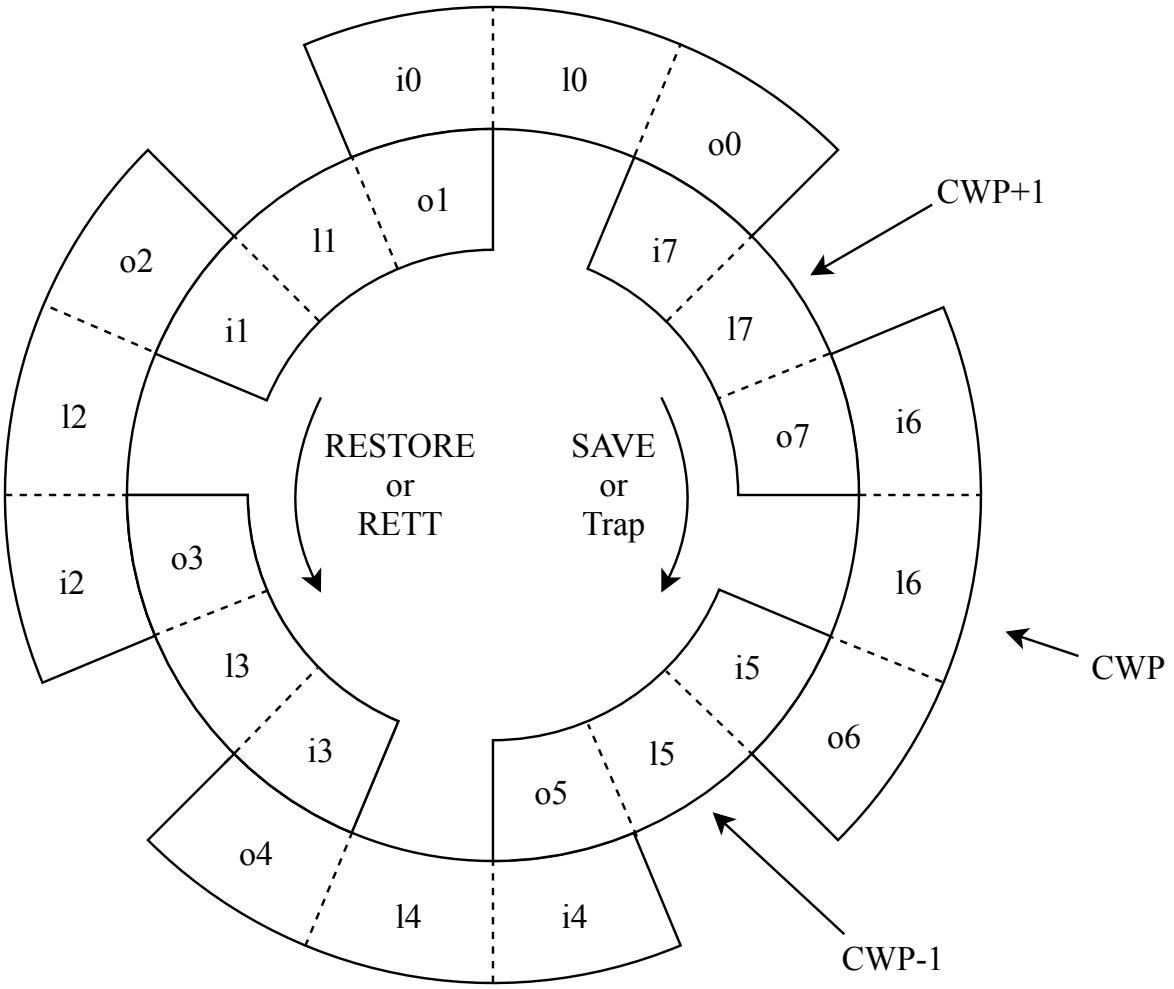


Figure 3.6.: SPARC general-purpose register windows

3.2.3. Memory access

Memory reordering can occur during compile time or during the execution of the instructions by the CPU. During most accesses to data memory, both types of reordering have no diverging effect on the behaviour of system when compared to no reordering done, and are performed to increase the performance of the program. However, some memory accesses require that the order of the issued instructions is followed, for instance, when performing operations that change the behaviour of following instruction, when writing to peripherals or in a multicore scenario. To that extent, most architectures offer the concept of memory barriers and different types of memory to deal with the aforementioned cases, and they play hand in hand in the memory model of the architecture in question.

SPARC supports three memory models to handle load/store instructions: sequential consistency, TSO and partial store ordering (PSO). Sequential consistency is the more stringent of all memory models, and enforces that all memory accesses be executed in the program order. This entails that after a load or store is executed, the processor halts until the instruction finishes executing. This memory model has strong performance drawbacks and its implementation is optional. The second and third memory

models impose the existence of a store buffer, where store instructions are kept until issued to memory, whenever the CPU perceives as the optimal transaction. The difference between the TSO and the PSO is in the order of the store instructions after being placed in the store buffer. The TSO model enforces that the store buffer behaves as a FIFO buffer, where the order of the stores is maintained, while the PSO relaxes the previous model by allowing reordering in the store buffer. In essence, PSO allows for store-store reordering while the TSO model does not. When in a PSO model, correct ordering of store instructions can be enforced by executing a STBAR instruction in between them. Another particularity of these models is that load instructions can retrieve information from the store buffer if it has not yet been committed to memory or entirely bypassing it if the data is not related to a previous store in the buffer. This permits that loads issued after stores can be completed before the stores finish. However, a load instruction effectively halts the processor until it retrieves the information it is accessing, and as such does not allow reordering around it. The PSO model is also optional, and only the TSO is being transitioned to newer processors, along with the sequential consistency. An illustration of the TSO model is shown in Figure 3.7.

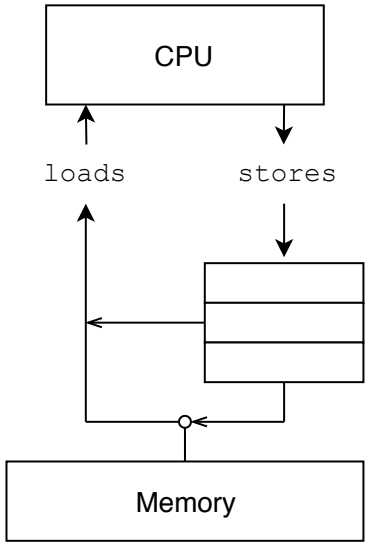


Figure 3.7.: SPARC TSO model

In the LEON4 processor the store buffer is accomplished through the use of caches. Since cache fetches are returned without blocking, loads can complete faster than stores, if the load hits and the store misses. If the caches are disabled, then the system behaves like the sequential consistency model.

Returning to the issue that raised the memory reordering problematic, the solution for operations that affect the system and the result of following operations is solved through the FLUSH instruction. The FLUSH acts as an instruction barrier, ensuring that all memory operations are finished before allowing new instructions to enter the CPU. When accessing memory-mapped peripherals, the GR740 guarantees that these are addresses are uncacheable, thus behaving like sequential memory. The last issue is when dealing with multicore systems. A load from processor 0 issued after a store to the same location in processor 1 can return before the store if the caches from both processors don't offer snooping.

Cache snooping is implemented in the GR740, where writes to one processor's cache are propagated to other processors' caches in the system and its entries are invalidated for that address, requiring an access to the main memory. If an hardware solution if not implement this problematic is usually addressed by avoiding concurrent access to shared structures and using mutexes or semaphores based on atomic instructions. An atomic instruction in SPARC like the SWAP behaves like a store and a load in one instruction, placing the store in the buffer, but at the same time halting the instruction stream until the load returns.

Another important aspect of the SPARC architecture is the use of address space identifiers. These identifiers are appended to all memory access and denote where is the memory correspondent to. For instance, supervisor data or user instruction. Additional identifiers are provided to handle access to I/O in a separate address space, which could only be accessed by a supervisor load/store, but this is implementation-dependent.

3.2.4. Exception handling

SPARC separates its exceptions into three different categories:

- Precise trap.
- Deferred trap.
- Interrupting trap.

A precise trap is induced by a particular instruction and occurs after the previous instructions have completed execution and the following ones remain unexecuted. Precise traps include the software traps generated using the T*icc* instructions. The *icc* stands for the condition codes that are evaluated to check if the exception is executed. A TA instruction, for instance, stands for Trap Always.

A deferred trap is also raised by a particular instruction, but may occur after the state of the CPU has changed. However, it must occur before any instruction that depends on the trap being executed. Deferred traps are implementation-dependent, and as such, may not exist in a particular architecture. Deferred traps in a SPARC SoC can be caused by:

1. floating-point and coprocessor exceptions;
2. implementation-dependent non-resumable errors;
3. the 2nd memory access on a multiple-access load/store (load/store double, atomic load/store or SWAP) if the 1st access return a non-resumable error;
4. events unrelated to the instruction stream, such as IRQs (interrupting traps are also deferred in nature).

The third type of trap is the interrupting trap, or IRQ, and accounts for all traps that are neither precise nor deferred. These will happen in the event of either an I/O assertion, breakpoint logic or in the event of an error related to a preceding incorrect instruction.

All traps except for the T*icc* instructions are hardware traps and are directly connected to a handler in the SPARC trap table. The supervisor, on a system reset, has to initialize the TBR with the address of the

trap table. The trap table must follow the scheme in Table C.1 and each entry contains four instructions, that normally perform a branch to the specified handler. A macro like the one demonstrated in Figure 3.8 is used to jump to the correct handler. The PSR is saved into **local** 0, the trap id is saved into **local** 3 and execution jumps to the handler address. Traps are identified by the trap type (`tt`), that is used has an offset in the trap table. Traps from `0x0` to `0x7f` are reserved for hardware, with values `0x80` to `0xff` left for software.

```
#define TRAP_HANDLER(n, handler)     \
   mov      %psr, %l0;               \
   sethi    %hi(handler), %l4;       \
   jmp      %l4 + %lo(handler);      \
   mov      n, %l3;
```

Figure 3.8.: SPARC trap table entry macro

Traps are controlled by the enable traps (ET) and PIL fields in the PSR. With ET $= 1$, precise and deferred traps will occur normally. For an IRQ, the PIL value will determine if an interrupt is accepted by the CPU. If the request level is higher than the PIL or equal to 15 (unmaskable), then the IRQ is taken. If there is more than one outstanding exception, the one with the highest priority (lower value) is taken first.

While ET $= 0$:

- IRQs will be ignored.
- If a precise trap occurs, the processor halts and enters in an error state.
- If a deferred trap was caused by an instruction while the ET $= 0$ the processor enters in error state.
- If a deferred trap was caused by an instruction while the ET $= 1$ but changed meanwhile, it is ignored.

On trap entry, the hardware will execute the following steps:

1. Traps are disabled, ET $= 0$.
2. The previous operating mode is saved on the PS field of the PSR.
3. The operating mode changes to Supervisor.
4. The register window advances to the next window, without checking for *window_overflow*.
5. The PC and the nPC are saved on the new window's registers **local** 1 and **local** 2, respectively.
6. The `tt` field is filled with the exception id.
7. If the trap is a reset trap, the PC takes the value 0 and the nPC the value 4. If it is any other trap, execution is passed to the TBR + `tt`.

Upon exit from the precise trap, the execution can either: redo the execution that triggered the trap by pushing the pre-trap PC onto the PC; emulate the instructions that triggered the trap and jump execution

to the next instruction by putting the pre-trap nPC onto the PC; terminate the execution of the program. A deferred trap can either emulate the instruction that triggered the deferred trap or terminate the program. Finally, the interrupting trap can either resume executing by reinstating the PC or terminate the program.

The GR740 provides control over the interrupts using a Multiprocessor Interrupt Controller. The maximum number of interrupts is 31. All interrupts are configurable, both for receiving I/O signals and for interrupt broadcast. The relevant registers are illustrated in Table 3.1.

Name	Register Description
ILEVEL	Interrupt level
IPEND	Interrupt pending
ICLEAR	Interrupt clear
MPSTAT	Multiprocessor status register
BRDCST	Broadcast register
IMASK	Processor n interrupt mask
IFORCE	Processor n interrupt force
EXTACK	Processor n interrupt acknowledge

Table 3.1.: SPARC Interrupt Controller Registers

3.3. ARM

Starting out as Acorn Computers Ltd, ARM had its breakthrough in 1993 with Texas Instruments, to whom they licensed their reduced instruction set computer (RISC) architecture, that ended up on the Nokia 6110 and was a massive success [59], [60]. This business model, where ARM licenses their processor architectures as Intellectual Property to other companies, that build, and sell the actual chip, ended up being successful with other silicon vendors and between 2002 and 2005, through the rise of SoC solutions, ARM processors became the de facto standard for mobile devices, with over 90% reported market share by the end of 2017, and 39% of all SoC chips shipped that same year [61]. From 2005 and onwards, they divided their processors into three categories: the Cortex-A, focused on higher performance; the Cortex-R, providing hard real-time functionalities; and the Cortex-M, for extremely low-power, low-cost micro-controllers.

The choice for a processor family to be used in space missions is hard, as all three processor families present strong arguments. The Cortex-A typically offers higher performance than the other two families, and is the only one offering a MMU. The Cortex-M sees the MMU removed, with a MPU as an optional choice, all for the goal of cutting on power consumption. The Cortex-R is branded for real-time and safety-critical applications. Perhaps the most fundamental aspect of real-times systems is their predictability, which sees the R family adding a Tightly-Coupled Memory to serve as a substitute for the cache when using real-time tasks and routines where predictability is fundamental. It also adds Error-Correcting Code on the L1 cache and buses, and it removes the MMU for a MPU.

With the United States investing their time on the Cortex-A and the European Union on the Cortex-R, there is no clear winner. Since the board provided by GMV is an Arty Z7, based on the Zynq-7000 SoC

from Xilinx, featuring a Cortex-A9, most of the following information is based for this specific processor, as well as the Cortex-A family.

The Cortex-A9 includes the Security Extensions introduced by ARMv7, but leaves out the Virtualization Extensions. The information on ARM presented from here on forward includes only the Security Extensions to the base ARMv7 model, unless otherwise stated. ARMv7 fails the Popek and Goldberg requirements for an efficient virtualization, containing critical instructions in its instruction set [62], [63]. The Virtualization Extensions are ARM's way of providing efficient full virtualization, however, since the targeted ARM processor is the A9 and the target guest OS, RTEMS, is open-source, paravirtualization becomes the optimal virtualization technique.

3.3.1. Instruction Sets

All 32 bit Cortex-A cores, with the exception of the Cortex-A32, implement the ARMv7 architecture. The ARMv7 architecture introduces four different instruction sets. The standard 32 bit ARM instruction set, A32, the Jazelle instruction set, which allows for Java Bytecode to run natively on hardware, and the Thumb-2 and ThumbEE instruction sets, that feature a hybrid instruction set of 16 bit and 32 bit instructions to improve code density.

The Thumb-2 came as an improvement for the first release of the Thumb instruction set. Originally only a 16 bit instruction set, with the purpose of improving the code density, it was not able to perform all instructions required in a complete RISC ISA, requiring constant instruction set changes between Thumb and ARM, as well as lengthier code segments of 16 bit instructions that fewer 32 bit instructions could perform. To overcome these shortcomings, ARM released the Thumb-2 instruction set, featuring a mixed length instruction set of 16 bit and 32 bit instructions. This instruction set retains most 16 bit instruction from the previous iteration, adding 32 bit instruction when required without the need to change instruction sets. The ThumbEE has small changes over the Thumb-2, mostly introduced to run code generated at runtime (e.g. JIT compilation). From here on out, the term T32 is used to refer to the Thumb-2 instruction set.

Since most code is written in C, with only the start-up code and exception handlers being written in assembly, only the later cases are in the standard 32 bit ARM instruction set, with the rest of the code being compiled into Thumb-2. Code density is a desirable trait in deep-space software due to the normally low available memory.

3.3.2. Operating Modes

ARMv7 runs away from the traditional model of only two operating modes, with a total of 7 modes, plus other two derived from optional CPU extensions, for a grand total of 9 operating modes. The basic model features two privileged levels, PL0 and PL1, with software running at the lowest level (PL0) raising an exception if it attempts to access privileged resources. The Virtualization Extensions added a third privileged level (PL2), that can manage the access to some system resources from PL1 by trapping some instruction only available in that privileged level. The operating modes are:

USER Software executing in User mode runs at PL0, being the only mode in that level. It can only access unprivileged system resources and assigned memory, and escalation to the higher privileged levels is only available by raising an exception. Most applications are expected to run at in this mode.

SYSTEM The System mode executes at PL1 and shares the same registers available to the User mode. This mode cannot be entered by any exception, and is normally used to access the User's stack pointer (SP) and link register (LR) from PL1, as entering User mode would make it impossible to return to PL1 again.

SUPERVISOR Supervisor mode is the standard entry mode of the processor on a system reset, changed only with the Virtualization Extensions. It is also the default mode to which a SVC exception is taken. It is expected for the OS to run in this mode.

ABORT The Abort mode is a PL1 mode to which a Data or Prefetch Abort exceptions are taken. These can happen on an instruction fetch or data read or write abort respectively, normally caused by an alignment or MMU fault.

UNDEFINED Undefined mode runs at PL1 and is the target of an undefined instruction exception. These will be raised in the event of an inaccessible coprocessor instruction, an exception during a co-processor instruction execution, an attempt to run an undefined instruction, normally due to illegal instruction access, or by a division by zero in the ARMv7-R.

IRQ The IRQ mode is the default mode to which an IRQ is taken.

FIQ The FIQ mode is the default mode to which a fast interrupt request (FIQ) is taken. This mode has its own set of five general-purpose registers, making it possible to handle an interrupt without having to push some global registers to the stack, as these are shared.

HYPERVISOR Only available as part of the Virtualization Extensions, it executes at PL2. It is the mode to which HVC or Hypervisor Traps are taken.

MONITOR Introduced by the Security Extensions, this mode is the medium to change between Secure and Non-Secure states, and introduction in the extension, that provides two nearly identical copies of the whole system, and completely disables applications in non-secure state to interfere with those in secure. Monitor mode is the mode to which a Secure Monitor Call (SMC) is taken. Is has access to both Secure and Non-Secure copies of the system registers.

All operating modes have copies in both the secure and non-secure states, except the Monitor mode that is only in the secure state, and the hypervisor, that is always in non-secure.

Figure 3.9 illustrates the different Operating Modes available on different ARM processors. The one available and used in this thesis is the A9, in light gray. The R family only has access to the non-secure state, with the R52 including the PL2 available through the Virtualization Extensions.

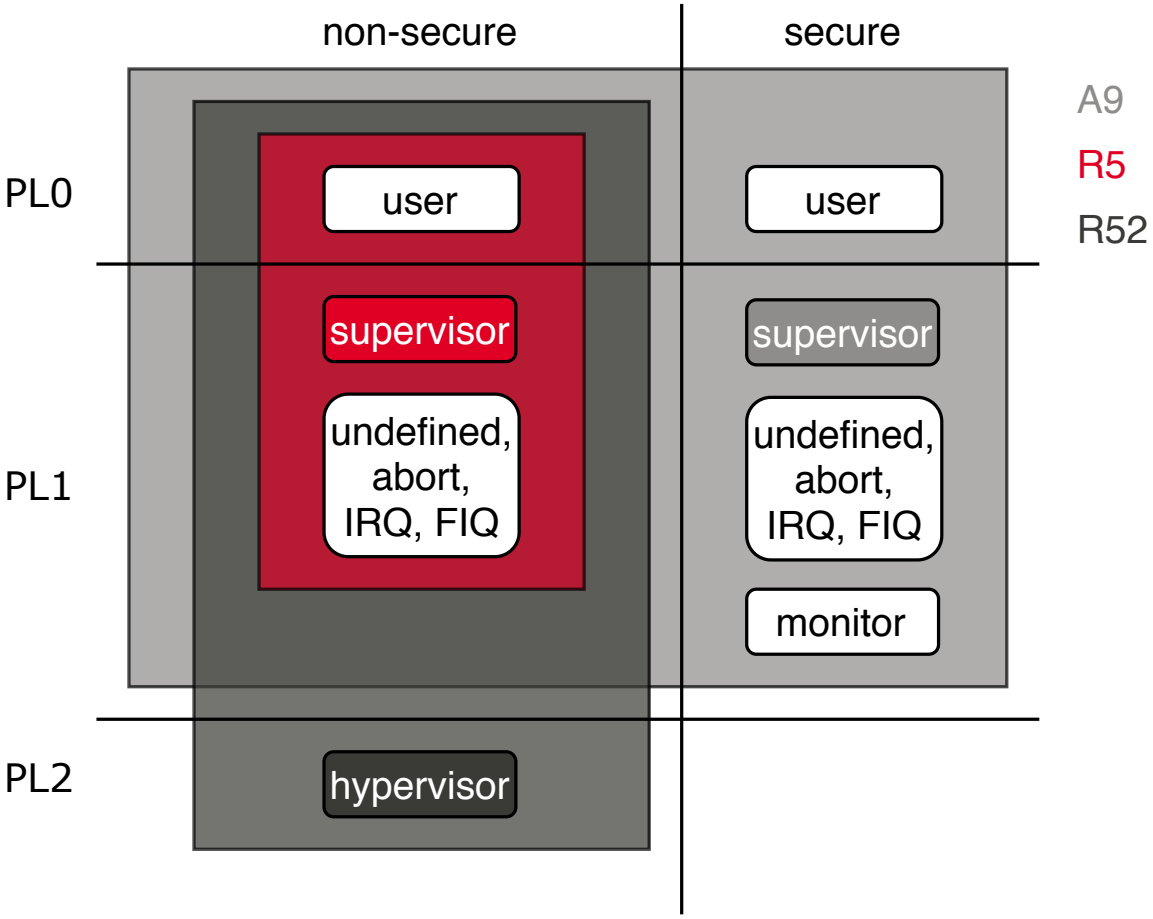


Figure 3.9.: ARM Operating Modes available to the A9, R5 and R52 processors

3.3.3. Registers

As opposed to the SPARC architecture, ARM's control/status access and debug is done through two of the coprocessors. Supporting up to 16, the coprocessor 14 is reserved for debugging capabilities and the coprocessor 15 is reserved to provide control over the system's resources. The only status register available is the PSR, as show in Figure 3.10, that contains information on the state of the processor such as:

- Condition flags (negative, zero, carry or overflow flags).
- Whether the processor is in ARM, Thumb or Jazelle state.
- Endianness of the data.
- Mask bits for Abort, IRQ and FIQ exceptions.
- Operating mode.
- Status bits for digital signal processing and the FPU.

The PSR is available as the current processor state register (CPSR), which has the current state of the processor, and the saved processor state register (SPSR), which contains the previous CPSR upon

entry on the current mode. The SPSR is used to observe the state of the processor before an exception takes place, so that when returning to the mode that raised the exception, the context can be restored.

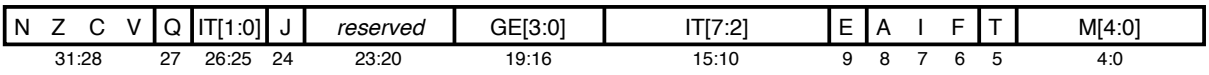


Figure 3.10.: ARM Program State Register

The rest of ARM registers follow the standard flat model with 15 general-purpose register shared among operating modes, except for registers 8 through 12 of the FIQ mode, which are for its sole use, and registers 13 (SP) and 14 (LR), which are exclusive to each operating mode, besides for the ones shared between User and System. Lastly there is the PC, that can also be accessed as register 15.

3.3.4. Memory

ARM is also affected by memory accesses reordering in the same way as SPARC, with the same problems arising: performing operations that change the behaviour of following instructions, writing to peripherals and multicore synchronization.

ARM offers a relaxed memory ordering model and allows for all types of reordering: store-store, store-load, load-store and load-load. As such it is more dependent on memory barriers than the previous architecture.

The Instruction Synchronization Barrier (ISB) and the Data Synchronization Barrier (DSB) solve the first problem, an instruction altering the result of the following instructions. The DSB ensures that the instruction stream halts until all pending memory accesses are completed. The ISB is more thorough than the DSB, flushing the entire pipeline and ensuring all instructions finish executing, including changes to the coprocessor 15 registers, that may have influence the entire system, not included in the DSB.

For memory accesses to memory-mapped peripherals, ARM offers three types of mutually exclusive memory types: Normal, Device and Strongly-ordered. Accesses to both Device and Strongly-ordered memory don't allow neither reads nor writes to access speculative, e.g. in branch delay slots, cannot be repeated if not properly finished, e.g. when returning from an exception, and the number, order and size of the accesses are maintained.

Finally, multicore access to the same memory location suffers from the same problems as in SPARC, and when snooping is not available, can be addressed by avoiding concurrent accesses and using mutexes/semaphores to manage the access to shared structures.

3.3.5. Exception Handling

ARM's exception model goes along with the operating modes, best seen in Table 3.2. This model is valid without the Virtualization Extensions and outside of Monitor Mode. After an exception takes place, each of the modes is then responsible for handling it. In the case of a SVC taken into Supervisor mode, the ID of the SVC can either be passed as a register or into the instruction itself with a range of 2^{24}, or 2^{8} if executed in T32. In case of an IRQ, the ID is in a register mapped in memory.

When an exception is raised, the previous PSR is saved into the SPSR of the operating mode where the exception is taken to, with the preferred return address saved onto the LR. When leaving the mode, the SPSR is written back into the CPSR along with a jump to the LR plus an offset depending of the operation mode it is exiting.

Exception Name	**PL1 mode taken to**	**offset**
Reset	Supervisor	0x00
Undefined Instruction	Undefined	0x04
Supervisor Call	Supervisor	0x08
Prefetch Abort	Abort	0x0c
Data Abort	Abort	0x10
not used	-	0x14
IRQ interrupt	IRQ	0x18
FIQ interrupt	FIQ	0x1c

Table 3.2.: ARM Exception Table

The Zynq-7000 SoC offers a general interrupt controller (GIC) as a centralized resource to manage all interrupts. The GIC divides all interrupts into three different classes:

- The 16 Software Generated Interrupts are interrupts generated by the CPU itself when the Software Generated Interrupt Register is written to.
- The 5 CPU Private Peripheral Interrupts are private for each CPU and contain the among others the CPU private timer and a copy of the global timer.
- The 60 Shared Peripheral Interrupts are interrupts from various peripherals routed to the CPUs.

The GIC is controlled through several registers, the more important of which are illustrated in Table 3.3.

Name	**Register Description**
ICCICR	CPU interface control
ICCPMR	Interrupt priority mask
ICCIAR	Interrupt acknowledge
ICCEOIR	End of interrupt
ICDISER	Interrupt set-enable
ICDICER	Interrupt clear-enable
ICDISPR	Interrupt set-pending
ICDICPR	Interrupt clear-pending
ICDABR	Interrupt active
ICDIPR	Interrupt priority
ICDIPTR	Interrupt processor targets
ICDSGIR	Software-generated interrupts

Table 3.3.: ARM Interrupt Controller Registers

4. Architectural differences

Regardless of the goal of the software meant to execute in a particular processor, and after the code present in a board's read-only memory (ROM) boots-up the device and transfers control to the software, there are unavoidable procedures that every hypervisor, micro-kernel or full-fledged OS must perform. Their implementation is dependent of the targeted architecture but they are, nonetheless, shared across the devices.

The code developed for this initial step must take into account what AIR needs and must conform with the BSP API it is expecting. Since there is already a BSP for SPARC laid out according to the API, the chosen solution was to port where possible the code already developed in SPARC to ARM. This task was performed with the help of both architectures' reference manuals, presented by sections in Table 4.1 and Table 4.2. These show in an organized form the required chapters and manuals for each development phase of the porting.

The following chapter depicts these procedures and explores the differences and the effort required in transitioning them between architectures. Furthermore, it elaborates on the architectural differences that influence the hypervisor design and development and describes in length the positive and negative aspects where appropriate.

Documents procured	**Reference**
ISA	
The SPARC Architecture Manual Version 8 Chapter 5, Annexes B, A and F	[64]
CPU	
The SPARC Architecture Manual Version 8 Chapter 4	[64]
Memory model	
The SPARC Architecture Manual Version 8 Chapter 6, Annexes D, H, I, J and K	[64]
GR740 Data Sheet and User's Manual Chapter 6.3, 6.4 and 6.9	[65]
Exception model	
The SPARC Architecture Manual Version 8 Chapter 7 and Annex D	[64]
GR740 Data Sheet and User's Manual Chapter 21	[65]
Board support	
GR740 Data Sheet and User's Manual	[65]

Table 4.1.: SPARC Reference Manuals

Documents procured	Reference
ISA	
ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition Chapters A4, A5, A6 and A8	[66]
ARM® Compiler armasm User Guide	[67]
ARM Architecture Reference Manual Thumb-2 Supplement	[68]
CPU	
ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition Chapters A1, A2 and B1	[66]
Cortex™-A9 Technical Reference Manual Chapters 2, 3 and 4	[69]
Zynq-7000 SoC Technical Reference Manual Chapter 3.2	[70]
Memory model	
ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition Chapters A3, B2, B3 and B4	[66]
Cortex™-A9 Technical Reference Manual Chapters 6, 7 and 8	[69]
Zynq-7000 SoC Technical Reference Manual Chapters 3.2 and 4	[70]
Exception model	
ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition Chapters B1.8 and B1.9	[66]
ARM® Generic Interrupt Controller Architecture version 2.0 Architecture Specification	[71]
Zynq-7000 SoC Technical Reference Manual Chapter 7	[70]
Board support	
Cortex™-A9 MPCore® Technical Reference Manual	[72]
Zynq-7000 SoC Technical Reference Manual	[70]
Arty Z7 Reference Manual	[73]

Table 4.2.: ARM Reference Manuals

4.1. Instruction set architecture

The two processor architectures are designed as RISC, making their ISA similar in nature. A RISC instruction set is typically defined by the following traits:

- each instruction undertakes a reduced amount of work;
- uniform instruction format and length, leading to simplified processor logic;
- all general-purpose registers can be used either as source or destination in all instructions;
- data-access is performed in a limited number of instructions, with that being their entire objective.

These ISA attributes result in a similar instruction set with an approximately equal number of instructions between SPARC and ARM. Nevertheless, while a big set of both architecture's instructions share much of the same purpose, their architectural dissimilarities translate into additional instructions on both sides. Table 4.3 presents the encountered instructions that either perform a behaviour not found in the other architecture or that realize the same goal differently.

The first row is straightforward. ARM follows an entirely different approach to the general-purpose registers than SPARC and it does not have any concept or operation regarding register windows. Given that window management is an important function of a hypervisor in SPARC, this is a big chunk of code and complexity that simply disappears.

Function	SPARC	ARM
window management	SAVE, RESTORE	-
memory barriers	FLUSH, STBAR, NOP	ISB, DSB, DMB
change instruction set	-	BX, BLX
change operating mode	-	CPS
mutual exclusion	SWAP	LDREX, STREX

Table 4.3.: Architecture specific instructions

4.1.1. Memory barriers

On the subject of memory barriers, while the expected outcome is the same, to guarantee correct memory access, the offered instructions come about for different reasons. FLUSH and DSB perform the same operation, halting the instruction stream until all memory accesses are finished. This becomes important with instructions that may alter the outcome of following instructions. ARM offers an additional barrier, the ISB, that halts the execution stream until the pipeline is flushed and discards all prefetched instructions. This barrier is a necessary due to operations on ARM's coprocessors affecting the behaviour of certain instructions.

ARM also provides the DMB to prevent memory reordering around it. Since SPARC is disregarding the PSO model approach and it is not present on the available hardware, the STBAR is not used in any context of AIR. In any case, it prevents the only type of reordering allowed in SPARC, store-store reordering.

4.1.2. Instruction set and operating mode change

Given ARM's more complex nature, with the possibility of jumping between two different instruction sets and seven native operating modes, comes the necessity for two additional sets of instructions to deal with the aforementioned aspects of the architecture.

To jump between A32 and T32, the instruction set is augmented with two supplementary branch instructions:

- the Branch and Exchange Instruction Set (BX `Rx`);
- the Branch with Link and Exchange Instruction Set (BLX *label*/`Rx`).

The `Rx` is a general-purpose register and the *label* is a `PC`-relative expression, both containing the address to branch to. Taking into account that addresses are word-aligned for A32 and halfword-aligned for T32, the last `bit` of an address is largely irrelevant for addressing purposes. BX makes use of this concept to distinguish between a branch to ARM code or Thumb code by reading the last `bit` of the address present at the register `Rx`. If $\texttt{Rx}[0] = 0$, then the instruction set is set to ARM, otherwise, if $\texttt{Rx}[0] = 1$, then the instruction set is set to T32. BLX follows the same approach when using a register as an argument, but will always change instruction set if the *label* is used as an argument.

Taking advantage of Thumb-2's code density, all C functions are compiled in T32 by using the flag `-mthumb` in the GNU ARM compiler. This operation leaves aside all code already written in assembly set to explicitly compile as A32, by using the `.asm` directive at the start of the `.asm` file or the inline assembly

code segments. Since most operations involving system configuration, exception handlers and context change, that must be written in assembly, are encoded in 32 bit either in T32 or A32, the chosen language was A32, as it simplifies significantly the coding process due to less restrictive instruction arguments, and the transition to T32 is secured though the BLX instruction, as shown in Figure 4.1.

```
    cpsid aif, #(ARM_PSR_IRQ)    ; Change to IRQ mode
    mov   sp, r7
    sub   r7, #irq_stack_size

    cpsid aif, #(ARM_PSR_SVC)    ; Change back to SVC mode

    blx   bsp_start_hook

    mrc   p15, 0, r8, c0, c0, 5 ; Reading the MPIDR into r8
    and   r8, #0xff
    cmp   r8, #0

    mov   r0, #0
    bne   secondary_core_init

    bx    pmk_init               ; core 0 init. never returns

secondary_core_init:
    bx    pmk_core_initialize    ; Secondary cores init. never returns
```

Figure 4.1.: ARM initial setup through C function calls in .asm code

Unlike the standard two privileged levels architectures that follow a simple user/supervisor model, where transitions are secured through exception handling and return, ARM increases the amount of operating modes at the highest privilege level. As such, it presents an extra instruction to hop around the different operating levels, the Change Processor State (CPS(IE/ID) iflags{, #mode}) instruction, also represented in Figure 4.1. This seemingly complicated instruction serves primarily to enable (IE) or disable (ID) the exceptions specified by iflags, which can be one or more of the following options: a for imprecise aborts, i for IRQ and f for FIQ. It can take as an optional argument the desired operating mode to change to, according to table B.1 encoding values. It can also be used without enabling or disabling any exceptions and used specifically for changing operating mode, e.g. `CPS #0x10` (change processor mode to user). However, this instruction cannot be used in the lower privilege level and the only way of exiting user mode remains through raising an exception.

The move general-purpose register to PSR and move PSR to general-purpose register (MSR/MRS) pair can equally be used to change the processing mode and remains the only way of retrieving the current PSR. In light of that, these instructions must be used when the current state is important for the program's logic flow.

4.1.3. Mutual exclusion

Important for resource sharing among multithreaded programs and multicore architectures is the implementation of mutual exclusion, which is a necessary property of multithreaded programs in preventing race conditions to shared resources such as shared memory or peripherals. The objective of mutual

exclusion is to allow a thread to access shared resources only in a specific code segment, called a critical section, while preventing other threads from accessing their own critical sections for the same resource [74]. Additionally, mutual exclusion must avoid deadlocks, where a thread is locked indefinitely waiting to enter its critical section.

In a single-core system, the simplest solution is to disable interrupts during a process's critical section. This will effectively prevent any other process from being scheduled during the critical section. This method introduces two problems:

- the clock will eventually drift during a critical section, since the timer interrupt will not be handled while it is executing, making it impossible to get an accurate time reading during its execution;
- if the process halts during the critical section, control can never be recovered.

Both these problems can be lessened by maintaining critical sections short and time bounded, without halting instructions.

The previous solution does not work for multicore systems, where resources are shared across multiple cores, since interprocess communication (IPC) takes place using interrupts. Supplementary hardware interfaces and software algorithms are required to implement shared resources across different cores in a system. The multicore mutual exclusion protocol used in AIR is the busy-wait method. This method requires an operation that undertakes both a load and a store atomically (i.e. non-interruptible). By defining a lock value in memory, the atomic instruction can write to that location and retrieve its old value, making sure it has the latest information at that location and reserving, if free, the lock. For this purpose, SPARC offers the SWAP instruction that does exactly that. An example of the acquire and release lock functions in SPARC are shown in Figure 4.2a.

On the other hand, ARM deprecated the use of the SWAP instruction in the latest versions of the architecture, and instead uses the global monitors attached to memory to manage mutual exclusion. When an address in memory is accessed by the Load Exclusive (LDREX) instruction, it tags the physical address as exclusive access for the current processor and clears any outstanding tag for any other address by the same processor. The returned value will determine if the lock is taken. If the lock is free, the Store Exclusive (STREX) instruction will attempt to write back to the lock address and succeed if the physical address is tagged as exclusive for the executing processor. After a successful write, it clears the tag of the executing processor for the lock address. Figure 4.2b presents an example of a lock acquire and release in ARM.

It is possible to combine the busy-wait lock and interrupt masking for uninterrupted multicore shared memory access, since the IPC is secured through the access lock variable controlled by the global monitor.

4.2. Stack

The first of the two necessary initializations before jumping into a C function is the stack initialization. Compiled C code will assume that the stack pointer (SP) is correctly initialized and make extensive use of it when jumping between functions, to pass and use arguments. Another very important use of the

```
pmk_lock_acquire:
try:
   mov   0x01, %o1
   swap  [%o0], %o1  ! try acquire lock
   tst   %o1         ! if zero, the lock was free
   bne   try         ! if not, spin it!
   nop

   retl
   nop

pmk_lock_release:
   retl
   st    %g0, [%o0]  ! Lock released
```

(a) SPARC busy-wait algorithm

```
pmk_lock_acquire:
   mov   r1, #0x1
try:
   ldrex r2, [r0]       ; load the lock value
   cmp   r2, #0         ; is the lock free?
   strexeq r2, r1, [r0] ; try and claim the lock
   cmpeq r2, #0         ; did it succeed?
   bne   try            ; if not, try again

   bx    lr

pmk_lock_release:
   mov   r1, #0
   str   r1, [r0]       ; release lock

   bx    lr
```

(b) ARM busy-wait algorithm

Figure 4.2.: Mutual exclusion algorithms

stack is during exceptions. Upon entering an exception, the previously executing application context is dumped into stack memory. This stack zone is called the interrupt stack frame and holds the necessary information to resume the application when returning from the exception. The GNU SPARC compiler (and others presumably) always reserves the necessary space on the stack to be used on the event of an exception, and this occurs for every register window. ARM on the other hand, since exceptions always jump to a operating mode with its own stack, does not need to reserve this additional space on every procedure entry, instead it can save the interrupt stack frame on each of the operating modes' stack.

Were a single OS in charge of PL1&PL0, these multiple stacks would offer a simpler separation of concerns between normal execution and exceptions, as well as providing a lower number of memory operations, since the jump between operating modes no longer has to be followed by a stack push/pop. In spite of these advantages, when dealing with virtualization, it adversely negates the time gained in relation to an individual stack. Due to the increase in the number of stacks used and all of them being unique to each partition context, the time consumed saving all of the different stacks increases. Given the memory constraints in the space domain, the memory inflation also presents a concern.

4.2.1. System stack

The existence of a shared stack between the User and System operating modes creates an opportunity for optimization and code hygiene, all the while saving in space. Considering that System mode is in PL1, it has access to all the instructions present at that privileged level, that is, all of them. The same applies to memory accesses, given that execution at PL1 is reserved for the hypervisor, and as such, the MMU virtual pages accessed are the exact same. With the escalated privilege and the possibility of executing the same functions as in IRQ, Supervisor, Abort or Undefined mode, the correspondent handler functions can be executed in System.

After it has been identified that the escalation in level was raised by the partition and not by an external

event, the interrupt stack containing the partition context can be saved in the System and User shared stack instead of on the requested operating mode's one. This removes the need to allocate an interrupt stack in every single operating mode that could be jumped to while in User mode. While any mode could be used to be the single recipient of exceptions raised by the partitions, the use of the System also brings advantages in keeping everything inside the partition's context. How this proceeds is further explained in Section 5.1.

4.3. .bss

The second of the two necessary initializations is zeroing the **.bss** section. In the C standard statically-allocated variables without an explicit initializer are initialized with the value zero and are expected to hold that value by the programmer, but when starting up a computer no assumptions can be made about the initial state of the memory.

Since this step could possibly take a long time depending on the amount of statically-allocated uninitialized variables, it is important to consider the size of the data bus and take advantage of it. Both the GR740 and the Arty Z7 use an AHB/AXI data bus of 64 bit, so for the best performance when writing to memory, two registers are stored at a time. Both SPARC and ARM offer instructions to take advantage of this characteristic of most modern boards. SPARC offers the STD/LDD pair to store/load doubles (64 bit variables) to/from memory. ARM boosts a more powerful version of the previous instructions, the STM/LDM pair, that can store/load multiple registers to/from memory. To take advantage of the data bus size, ARM's version needs be used with a pair number of registers. This principle holds true not only in the **.bss** zeroing, but everywhere else in the code, with compiled code taking advantage of both pairs of instructions.

4.4. Cache, branch predictors and translation lookaside buffer

After both the stack pointers are initialized and the **.bss** is cleared, any remaining initialization procedures can be done in C.

For the same reason the **.bss** needs to be cleared, so do the L1 instruction and data caches, branch predictors and TLB need to be invalidated. As defined in the ARMv7 reference manual, both the caches, branch predictors and the TLB are disabled at reset. There is not, however, information on the starting content of the caches at reset, so they are considered to not be empty, so they must be cleaned at start-up. All these hardware structures must have their contents invalidated at start-up, and only after can they be enabled.

SPARC differs has there is no constraint for the caches and the TLB to start disabled. The branch predictor is not programmable in the LEON4 processor. While the SPARC architecture manual does not specify the initial state of the cache, the GR740 one does, stating that both caches are disabled at reset. Nevertheless, to ensure compatibility with other BSPs, both the caches and the MMU are disabled during start-up, followed with the cache and the TLB being invalidated, and only after re-enabled.

The L2 caches are dependent of the board used. The GR740 L2 cache is disabled after reset and it is invalidated at the same time it is enabled. The Zynq-7000 SoCs L2 cache is cleared upon reset, but the entries must still be invalidated before enabling it.

4.5. Register window

The immense disparity between SPARC's and ARM's register layouts only comes into play when changing context. Removing SPARC's register windows only eases the hypervisor development when directly dealing with the registers, as the maintenance code on the windows is removed. The interrupt stack frame is also reduced since it only has to keep one set of general-purpose registers, plus the FIQ's registers 8 through 12 (if used) and the LR, SP and SPSR present in every mode, with the exception of the shared LR and SP, and inexistence SPSR, between User and System, up to a total of 29. This is in contrast with SPARC's n windows, 8 in the case of the GR740, with 16 registers each, plus 8 globals, totaling 136. In both cases, the total number of registers is multiplied by the level of nesting permitted, per partition context.

While in theory SPARC's register window improves the performance when changing context during normal operation (no hypervisor), by reducing the number of accesses to the stack when entering/exiting function calls, it introduces a penalty when virtualization comes into the picture. Besides the higher space usage, when changing from one partition to another, all used register windows need to be dumped into memory. Although AIR increases performance by only restoring the last used register window and only restoring additional windows when the partition requires them, the WCET still comes into play, since all windows could be requested. ARM's plain register design increases the predictability of context switching.

4.6. System configuration and identification

The two architectures access information on the system differently, with ARM providing a slightly simpler model. The CWP bits on SPARC's PSR plus the WIM offer information on the register window model of the architecture. Those places on ARM's PSR are filled with bits regarding the more complex execution system provided. The exception vector base address is present on the TBR on the SPARC architecture, while in ARM they are set through the Vector Base Address Register (VBAR), available through the coprocessor 15. For the value on the VBAR to be used, the V field on the System Control Register (SCTLR) must be set to 0. Otherwise the exception base address is set to `0xffff0000`, known as *Hivecs*. The SCTLR is also accessed through the coprocessor 15. The cache and MMU on SPARC are addressed by its Address Space Identifier (ASI) that loads and stores information through the modified store and load instructions by appending an A at the end, eg. LD (Load Word) becomes LDA (Load Word from Alternate space). Additional configuration is implementation-dependent and it is available on the ancillary state registers (ASRs). GR740 provides a configuration register on the ASR17 shown on Table D.1. Apart from the PSR, ARM additional information and configuration is done through the coprocessor 15, where

each register is accessed with the instruction pair MRC/MCR (move to general-purpose register from coprocessor/move to coprocessor from general-purpose register).

The comparison between SPARC and ARM regarding system configuration is illustrated in Table 4.4.

Function	**SPARC**	**ARM**
window management	CWP + WIM	-
exception table base address	MOV %LX, %TBR	MRC/MCR P15, 0, RX, C12, C0, 0
system configuration	RDASR/WRASR %ASR17	MRC/MCR P15, 0, RX, C1, C0, 0
cache control	LDA/STA [0]0X2	MRC/MCR P15, 0, RX, C1, C0, 0
cache operations	ASI	COPROCESSOR 15
MMU control	LDA/STA [0]0X19	MRC/MCR P15, 0, RX, C1, C0, 0
MMU operations	ASI	COPROCESSOR 15

Table 4.4.: Difference between architectures regarding the system configuration

4.7. Exceptions, Traps and Interrupts

The vocabulary used by each architecture manuals differs when exceptions, or traps in the case of SPARC, are referred to. In essence, they specify the same and can be used interchangeably. An exception or trap is an anomalous event that changes the normal flow of a program and jumps the execution to a table of predefined handlers. If the handlers do not exist, unexpected behaviour will ensue. For the remainder of this thesis, the term exception is used. Exceptions can take multiple forms and arrive from multiple sources. On a first level, depending on how an architecture is defined, an exception can take one of the following three forms:

- Synchronous exception. **Generated** as the result of an attempted execution of the current instruction stream. The return address is **guaranteed** to indicate the instruction that raised the exception.
- Precise asynchronous exception. **Not generated** as the result of the current instruction stream. The state of the CPU presented to the exception handler is **guaranteed** to be the same to that when the exception was raised.
- Imprecise or deferred asynchronous exception. **Not generated** as the result of the current instruction stream. The state of the CPU presented to the exception handler is **not guaranteed** to be the same to that when the exception was raised.

SPARC presents their exceptions through similar terms. Precise traps refer to the synchronous exceptions, deferred traps refer to the deferred asynchronous exceptions and the interrupting traps are the precise asynchronous exceptions.

Synchronous exceptions are the result of instructions that directly trigger an event on the execution stream. Within the bounds of this category fall SVCs and undefined instructions, either by attempted execution of an instruction not available at that privileged level or an unimplemented instruction, such as a floating-point operation with the FPU disabled. Precise asynchronous exceptions normally encompass

IRQs. The state previous to the interrupt is known, but won't give any insight as to the interrupt itself. Imprecise asynchronous exceptions are normally the result of memory errors. Due to the optimizations present in most present-day CPUs, such as caches and write buffers, errors when writing/reading to/from memory are seen with some delay and the exact instruction that generated the abort cannot be derived from the state of the computer when the exception is taken. A common example is a read-only memory area marked as cacheable and with a write-back policy. Writes to the cache won't trigger any error and the exception will only appear when the cache line gets evicted and written back to memory.

All the aforementioned scenarios point directly to an operating mode in ARM, making the distinction between each exception type faster but grouping several distinct errors into one function handler. SPARC on the other hand holds a 256 entries table (Table C.1), with 80 entries reserved for a combination of undefined instructions, data aborts, the *window_underflow* and *window_overflow*, and the *reset* exception, 16 entries for IRQs and 128 for user determined SVCs. Not all of these exceptions need to be implemented in a BSP, but they have to respect the order defined in the architectural standard. While failed instruction fetches and data errors can cause imprecise asynchronous exceptions, ARM offers some insight through the Instruction Fault Status Register and the Data Fault Status Register, accessible through coprocessor 15. IRQs in ARM are distinguished later in the interrupt handler through the Interrupt Acknowledge Register (ICCIAR) available in the interrupt controller, found in Table D.2. Finally, SVCs are differentiated by a value passed in the instruction itself, accessed in the exception handler through the preferred return address, as seen in Figure 4.3. In T32, the maximum number of SVCs is restricted to 256, due to the $8\,\text{bit}$ encoding (imm8), but the range goes up to 16777216 in A32 due to the $24\,\text{bit}$ encoding of the SVC id, shown in Figure 4.4.

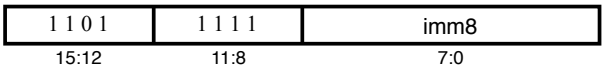


Figure 4.3.: Supervisor Call (SVC) encoded in Thumb-2

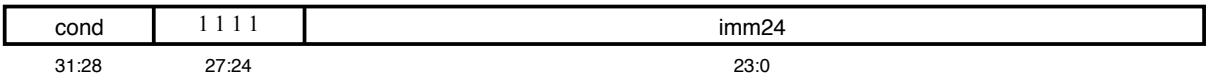


Figure 4.4.: Supervisor Call (SVC) encoded in A32

The major difference going from one architecture to the other is the use of two tables in ARM for handling any exception in contrast with the one in SPARC. While the method employed by ARM introduces an additional level of redirection, it reduces the initial amount of exception handlers, possibly negating duplication of the same code. It also reduces the size of the exception handler table, which is mandatory for the two architectures, going from 256 entries in SPARC to only 8 in ARM.

4.8. Memory Management Unit

Both SPARC's and ARM's MMU's are organized largely in the same manner, with the exception of SPARC's reference MMU proposing three translation levels and ARM's using only two. The respective

Page Table Entries are referred in each of the architecture's manuals, and must be followed for the TLB to work as expected. However, not much differs in their implementations and the code developed for ARM is mostly a repetition of the one for SPARC, adjusting only the Page Table Entries format.

5. BSP hypervisor-related procedures

In the previous chapter, the architectural differences between SPARC and ARM and their impact on any architecturally-dependent software were studied and, where appropriated, were analyzed regarding the use of a hypervisor. This chapter now explores the discovered necessary adaptations specific to a TSP hypervisor in the scope of the BSP.

It is important to state that the following chapter focus on a single virtualization method, paravirtualization, and the developed work was within the scope of the AIR hypervisor, more specifically, it respects the previously defined calls to the BSP defined by AIR. An effort was made to find whenever possible the correspondence between SPARC and ARM, so that the hypervisor logic required no change.

5.1. Virtualization

The primary role of a hypervisor is to maintain several VMs running concurrently. Since the targeted processor is not virtualization efficient according to Popek and Goldberg's requirements [42], either binary translation or paravirtualization must be used. Paravirtualization was chosen since it results in the most optimized hypervisor and the targeted guest OS, RTEMS, is open-source, allowing kernel modifications. The necessary adaptations on a guest OS are further studied in section 6.1. This section focus instead on the required services offered by the hypervisor and their differences when going from SPARC to ARM.

To reiterate, the primary role of the hypervisor is to maintain several VMs running concurrently. To accommodate this, it requires two features:

1. memory earmarked for each VM context;
2. a mechanism to change context.

The first is allocated for each core and for each partition during initialization. The space assigned for each core contains the context of an idle VM, and is the one used when first booting up the core and on every vacant time slice. The one for each partition is loaded with the initial parameters specific to that particular partition. The PMK can use three BSP calls to setup these contexts, which declarations can be seen in Figure 5.1.

```
void core_context_init(core_context_t *context, air_u32_t id);
void core_context_setup_idle(core_context_t *context);
void core_context_setup_partition(core_context_t *context, pmk_partition_t *partition);
```

Figure 5.1.: BSP functions to initialize and setup the partitions' context

Both contexts are composed of:

- CPU id and registers (see Sections 3.2.2, 3.3.3, 4.5 and 4.6)
  - on SPARC: PSR, WIM, TBR, cache control and MMU control;
  - on ARM: PSR, VBAR and MMU control.
- Interrupt stack frame pointer (see Section 4.5).
- Interrupt nesting level.
- Interrupt controller registers (see Sections 3.2.4 and 3.3.5.
- FPU context if compiled for it.
- AIR current state information.
- HM event currently being serviced (if one exits).

The interrupt stack frame contains the architecture register window and the program counter, and along with the CPU, FPU and interrupt controller registers, accounts for the current state of the machine. These must be kept intact between the execution jump to the hypervisor and the return back to the partitions. The interrupt nesting level is used to determine the nesting of the interrupts handed to the partition. Since all interrupts are routed through the hypervisor, AIR uses this value in the virtualization of interrupts, to determine whether the execution returns to the saved PC or jumps to the guest OS exception table, whose value is in the TBR or VBAR fields. The AIR current state is an internal state field of AIR for each partition, to determine the current state of the partition, e.g. partition initialization or HM execution. This is the value used by the HM table (section 3.1.4) to determine the appropriate recovery action. The HM event structure can be later requested by the partitions to discern between errors. The core id is actually the virtual core id assigned for each core to each partition. This value may differ the true core id. This value is important in a symmetric multiprocessing (SMP) partition, where inter-core communication is essential. To perform this communication, the main SMP core, normally core 0, needs the identification of the other cores where the application is running.

In SPARC, not all register windows need to be saved at each jump to the hypervisor, only the current one. Since partitions can spawn multiple minor time frames, with the likelihood increasing for higher resolutions, saving and restoring the entire interrupt stack frame becomes a tedious and expensive process. After the first context save and subsequent restore, where all windows were dumped into memory, AIR only restores the last used window and sets up the WIM register just before that one. AIR takes advantage of the *window_underflow* exception, that is raised if the partition attempts to restore a previous window, to restore the previously dumped windows.

ARM on the other hand has a simple register window, so it saves and restores it on every scheduler entry. Where it differs from SPARC is the multiple PL1 modes. Since the timer interrupt is an IRQ, the AIR timer handler runs entirely in IRQ mode, with no need to jump to another one.

Both architectures save only the essential fields to virtualize the CPU, and delay any memory dump when an actual context switch happens. The cache, MMU and FPU registers only need to be saved on a

context switch. The MMU table for each partition holds not only the page tables relative to the partition, but also the ones used by the hypervisor, with supervisor access. It is also known that AIR uses no FPU registers, so saving these can also be relegated to a context switch.

The second feature of the hypervisor is a mechanism to change context. The behaviour in SPARC was copied to ARM. A single timer is used exclusively for the scheduler interrupt, and is the only one set to the highest priority. Both architectures allow nested interrupts, so the highest priority interrupt, if only one, will always be taken. Code segments with disabled interrupts are only used in first context save upon entering an exception and during the timer handler. Never during partitions. Any request by the partition to disable interrupts is only simulated in the virtualized CPU registers, never actually taking place.

5.1.1. AIR Paravirtualization

AIR follows the paravirtualization approach to achieve correct virtualization. This approach requires guest OS modifications, but more importantly for the hypervisor, it must offer a direct route of communication between the VMs and AIR. This route is maintained by HVCs to AIR, using the SVCs available in both architectures. The HVCs available to the guest OSs can be separated into two main groups:

1. These HVCs are used to implement hardware virtualization. The group is composed of:
    - disable/enable interrupts;
    - disable/enable exceptions;
    - disable/enable the FPU;
    - get/set the CPU registers;
    - get/set interrupt mask;
    - return from an HM event.
2. These HVCs are used to either retrieve information from AIR or to perform more complex procedures using AIR's abstractions. The group is composed of:
    - get physical address;
    - get core id;
    - get µs per tick and get elapsed ticks (these two HVCs can be used to implement timers within the partitions);
    - get partition information;
    - get schedule information;
    - get ports information;
    - get HM event;
    - print;
    - boot secondary core.

Only the first group requires direct migration from one architecture to the other. The second group of calls interact directly with the PMK, although some will later require interaction with the BSP, e.g. when booting secondary core. The “return from an HM event” belongs in the first group because its purpose is to jump to the pre-HM PC, which is accessible directly from the BSP.

In this scenario, ARM achieves the lowest disabled interrupts downtime. Since HVCs can be interpreted as normal function calls, the handler takes advantages of a similar behaviour of the stack usage of normal function entries/returns. After the execution enters the Supervisor mode, the preferred return address and SPSR are saved as per usual. Using ARM’s unique instruction SRS (Store Return State), the Supervisor mode can save the preferred return address and the SPSR into any stack available. By using the System stack, which is shared with the User mode, it attains nearly identical behaviour to that of a function. Furthermore, is can now push onto the System stack the interrupt stack frame before the exception occurred, since the previous instruction does not use any of the general-purpose registers. At this point, preemption can be enabled again, and the exact procedure can happen again, even though it is extremely unlikely, since HVCs do not use HVCs again. The timer interrupt can however happen, without loss of information, as long as the stack is kept coherent. Since the execution will not return to User mode before all exceptions are handled, there is no possible way for the stack to become corrupted. After the HVC is completed, the preferred return address and SPSR are loaded onto the PC and the CPSR through the RFE (Return From Exception) instruction. A shortened version of the HVC handler is illustrated in Figure A.4.

Using the System stack is important in this behaviour, because it keeps thing clean, all the while saving in space and time. If the Supervisor stack were used, it would also need to be saved in memory during a context change. This way, the System stack is the only one used across the entire partition time slice.

In addition to the implementation of the HVCs is also its availability for the guest OSs. The partitions are compiled together with LIBAIR, a collection of the virtualization HVCs. This file is written in assembly in both architectures, and its comparison is shown in Figure 5.2.

```
.align 4
.globl air_syscall_get_physical_addr
air_syscall_get_physical_addr:

    set     AIR_SYSCALL_GET_P_ADDR, %o5
    ta      AIR_SYSCALL_OS_TRAP
    retl
    nop
```

(a) SPARC HVC Implementation

```
.align 4
.globl air_syscall_get_physical_addr
air_syscall_get_physical_addr:

    svc     #(AIR_SYSCALL_GET_P_ADDR)
    bx      lr
```

(b) ARM HVC Implementation

Figure 5.2.: LIBAIR System Call

5.1.2. Interrupt virtualization

Also required to complete the virtualization process, is the virtualization of the interrupts destined to the VMs. For the hypervisor to maintain control over the system, it must catch all exceptions, even if these are bound to the VMs. The code developed is similar in both architectures. First an interrupt is identified about its type by the hypervisor (see Section 4.7). Afterwards, it can be checked if it belongs to any of the scheduled partitions. If it does, the hypervisor recreates the same process of identifying the interrupt by modifying the appropriate virtualized registers and when returning to the partition, it resumes execution from the guest OS' exception table. The paravirtualized guest OS can then redo the same procedures used by AIR, but in its case, by accessing the virtualized information.

The small discrepancy is due to the different exception models (see Sections 3.2.4 and 3.3.5). As identified in the previous chapter, ARM employs an additional redirection level. As such, the virtualized exception checks are present across all exception handlers, where in SPARC all exceptions have the same entry point, and this code only appears once.

5.2. Time and space partitioning

After the hypervisor achieves correct and efficient virtualization, the TSP aspect simply boils down to the schedule and communication protocols used. And the TSP standard is rather concise on it. The schedule is fixed and cyclic, composed of minor and major time frames that repeat themselves. There is no shared memory between VMs. Communication must be done through queuing and sampling ports supplied by the hypervisor and the contents must be written from one VM to the hypervisor and read from the hypervisor to the receiving VM. Nothing in this implies architectural changes, since its application is done entirely in the PMK. The only BSP calls are the writes to and from the hypervisor, that check if the partitions have the correct permissions to access that memory and do a *memcpy*. There are no differences between architectures in this regard.

5.3. Health Monitor

The architectural differences in the HM implementation are derived from the different exception models (see Sections 3.2.4, 3.3.5 and 4.7). The HM is responsible for all the errors that occur in the system, and is designed to be generic enough to accommodate different BSPs. The job of the BSP in this design is to determine the error id of the fault. For this objective, SPARC installs the same HM handler in all relevant exceptions, an excerpt is shown in Figure 5.3, and defines the error via a lookup in a 256 entry vector based on the exception table entry, also in Figure 5.3.

In contrast with SPARC, ARM's exception table is very reduced in size, having only three entries for HM errors (undefined, prefetch and data aborts), and the error id is found through if-else statements. However, for more detailed error ids, the instructions that generated the errors must be decoded according the ARM's instruction set encoding. To implement lazy FPU switching, where the FPU is only

```
air_error_e error_map[] = {
   AIR_POWER_ERROR,            /* Power-on reset                              */
   AIR_SEGMENTATION_ERROR,     /* Error during instruction fetch              */
   AIR_UNIMPLEMENTED_ERROR,    /* UNIMP or other un-implemented instruction   */
   AIR_VIOLATION_ERROR,        /* Privileged instruction in user mode         */
   AIR_FLOAT_ERROR,            /* FP instruction while FPU disabled           */

   (...)

   AIR_VIOLATION_ERROR,        /* Unexpected/Unassigned trap                  */
   AIR_VIOLATION_ERROR,        /* Unexpected/Unassigned trap                  */
   AIR_VIOLATION_ERROR,        /* Unexpected/Unassigned trap                  */
};

void sparc_hm_handler(
        sparc_interrupt_stack_frame_t *isf, pmk_core_ctrl_t *core_ctrl) {

    /* map it to the PMK independent data */
    air_error_e error_id = error_map[isf->tpc];

    /* call the PMK health monitor */
    printk ("HM error detected id - %d\n", error_id);
    pmk_hm_isr_handler(error_id);

    /* jump the current instruction */
    isf->pc   = isf->nkpc;
    isf->nkpc = isf->nkpc + 4;
}

void sparc_healthmonitor_init() {

   sparc_install_vector_trap_handler(0x01, sparc_hm_handler);
   sparc_install_vector_trap_handler(0x02, sparc_hm_handler);
   sparc_install_vector_trap_handler(0x03, sparc_hm_handler);

   (...)
```

Figure 5.3.: SPARC HM handler

re-enabled after a context switch if it is used, the undefined exception faulted instruction is decoded to determine its identity. The code is listed in Figure A.5.

While ARM benefits in size from its simplified HM design due to its exception model, when entering into detail in the determination of the error id, it is clear that SPARC has the advantage. Even with the exception table entries locked in number to take into consideration future versions of the architecture, reverse-engineering every instruction in ARM becomes a very time consuming job.

6. Evaluation

This chapter present the tests performed on AIR compiled with the new BSP. Since AIR is a hypervisor that relies on paravirtualization, a simple barebones OS was implemented in ARM.

6.1. Barebones OS

To reliably test a hypervisor that employs paravirtualization, a paravirtualized OS is required. However, the RTEMS OS is not yet virtualized by AIR in its ARM version. As such, to set-up a test bench for the new BSP, it was created a barebones OS, whose sole function is to test the functionalities of the TSP hypervisor. This OS can run any executable compiled in T32 or A32, as long as it does not have any external dependencies apart from the GCC libraries.

The OS only runs in single-core, extensively uses the LIBAIR API, and besides the exception table with staple handlers, there is not much else. It launches the *main* function of the linked executable. The tests were run both on the QEMU emulator and on the Arty Z7, yielding similar results.

6.2. List of tests

6.2.1. LIBPRINT

The LIBPRINTF tests aim at analysing the behaviour of the *pprintf* HVC offered by LIBPRINTF when used by more than one partition. The tests parameters are presented in Table 6.1.

Test	Number of Partitions	Schedule
LIBPRINTF2	2	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
LIBPRINTF3	3	P1 0.5 s - P2 0.5 s - P2 0.5 s - idle 0.5 s
LIBPRINTF4	4	P1 0.5 s - P2 0.5 s - P3 0.5 s - P4 0.5 s
LIBPRINTF5	5	P1 0.4 s - P2 0.4 s - P3 0.4 s - P4 0.4 s - P5 0.4 s

Table 6.1.: LIBPRINT tests

This test performs as expected, without any observable delay or bottleneck caused within partitions.

6.2.2. Timer

The timer test aims at analysing the behaviour of the timer HVCs, the *air_syscall_get_us_per_tick* and the *air_syscall_get_elapsed_ticks*. The combination of these two calls provides a timer for the partition,

updated at every minor timer frame.

Test	Number of Partitions	Ticks/Second	Schedule
timer10	2	10	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
timer100	2	100	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
timer1000	2	1000	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s
timer10000	2	10000	P1 0.5 s - idle 0.5 s - P2 0.5 s - idle 0.5 s

Table 6.2.: Timer tests

This test performs as expected, with higher resolutions attained for higher ticks per second, but there was a noticeable delay in the timer10000 test in comparison with the previous ones. The actual delay needs to be further studied with timing analysis tests.

6.2.3. Health monitor

The health monitor tests are performed to gauge the response of the HM to different kinds of errors in different states of the program.

Test	Number of Partitions	State	Schedule
hmmi1	1	Module initialization	P1 1 s
hmpe1	1	Partition Execution	P1 1 s
hmpe2	2	Partition Execution	P1 0.5 s - P2 0.5 s
hmhm2	2	HM execution	P1 0.5 s - P2 0.5 s
hmfpu	1	Partition Execution due to FPU error	P1 1 s

Table 6.3.: Health monitor tests

The tests performed as expected, always launching the correct actions from the HM configuration table (see Section 3.1.4). The two partitions tests did not affect each other negatively, and the errors were successfully contained within the partitions. The FPU lazy-switch test also performed as expected, with the partition recovering after an undefined error caused by a FPU instruction with the FPU disabled, by enabling the FPU and repeating the instruction.

7. Conclusion

The aim of this thesis was to increment the BSP portfolio of AIR with a new ARM SoC. It achieved not only that but it also produced documentation relevant for future work in this area.

7.1. Achievements

In conclusion, the new BSP for AIR performed innocuously, yielding the same results as the SPARC one. It maintained the same BSP API calls and performed equivalent procedures when compared side-by-side. The architectural differences were mostly felt in the virtualization aspect of the hypervisor, due to the incompatible register and exception models. However, similarities were found and the new BSP achieved the same result through different methods. It improved on the HVC handling, by providing a lower time frame where the interrupts are disabled, thus increasing the accuracy of the global timer. The objective of creating a BSP for a TSP hypervisor was successful and is currently being used in production and in proposals for future projects.

In addition to the created BSP, a comparative study between SPARC and ARM was also performed and can be used as reference material when migrating from SPARCv8 to ARMv7.

Parallel to the development of the new BSP was also the development of the toolchain used to compile AIR, which was reinforced with new templates and scripts to streamline the development process.

7.2. Future work

The most immediate development to be made is the expansion of the BSP to multicore, accompanied by the a multiprocessing barebones OS. This would not only bring the most immediate performance boost, but also adding another dimension to the scheduling possibilities.

Since this code is expected to run in space, validation and verification facilities should also be added to speed-up the process of qualification.

With the attention given to the modularity of AIR, the addition of new architectures and more interesting BSPs could also bring more depth into AIR. This case is specially true for the addition of the ARMv8, that incorporates the Virtualization Extensions into a more consolidated architecture.

Bibliography

[1] *MA31750 High Performance MIL-STD-1750 Microprocessor*, Datasheet DS3748-8.0, Dynex Semiconductor Ltd, Jan. 2000.

[2] *Military Standard Sixteen-bit Computer Instruction Set Architecture*, MIL-STD-1750A, United States Department of Defense, Jul. 1980.

[3] E. C. Department, *Leading up to LEON: ESA's First Microprocessors*, ESA, Jan. 2013. [Online]. Available: `https://www.esa.int/Our_Activities/Space_Engineering_Technology/Leading_up_to_LEON_ESA_s_first_microprocessors`.

[4] M. Ramström, J. Höglund, B. Enoksson, and R. Svenningsson, "32-bit Microprocessor and Computer Development Programme Final Report", Saab Ericsson Space AB, Göteborg, Sweden, Tech. Rep. MCD/MNT/0015/SE, 1997.

[5] J. Gaisler, "Fault-Tolerant and Radiation-Hardened SPARC Processors", in *Proc. Topical Workshop on Electronics for Particle Physics*, Prague, 2007.

[6] ——, (Oct. 2017). 25 Years of SPARC - a personal retrospective by Jiri Gaisler, ADCSS 2017 - 11th ESA Workshop on Avionics, Data, Control and Software Systems, [Online]. Available: `https://indico.esa.int/event/182/`.

[7] *Nasa Technology Roadmaps TA4: Robotics and Autonomous Systems*, National Aeronautics and Space Administration, 2015. [Online]. Available: `https://www.nasa.gov/sites/default/files/atoms/files/2015_nasa_technology_roadmaps_ta_4_robotics_and_autonomous_systems_final.pdf`.

[8] J.-L. Terraillon, "Multicores in Space", in *Proc. 17th Int. Conf. Realiable Software Technologies Ada-Europe*, Stockholm, 2012.

[9] *Technologies for European non-dependence and competitiveness*, COMPET-1-2016, Funding & tender opportunities, European Commission, Nov. 2015. [Online]. Available: `https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/compet-1-2016`.

[10] *DAHLIA Deep sub-micron microprocessor for spAce rad-Had appLIcation Asic*, DAHLIA Consortium, 2017. [Online]. Available: `https://dahlia-h2020.eu/`.

[11] J. L. Poupat, T. Helfers, P. Basset, A. G. Llovera, M. Mattavelli, C. Papadas, and O. Lepape, *DAHLIA, Very High Performance Microprocessor for Space Applications*, Proc. DASIA 2018 - DAta Systems In Aerospace, to be published, Oxford, May 2018.

[12] Portugal Space, 2019. [Online]. Available: https://www.ptspace.pt/.

[13] *Atlantic International Satellite Launch Programee: Launch services to Space from the Island of Santa Maria, Azores*, International Call for Interest, ESA, Luísa Ferreira and CEiiA, Sep. 2018. [Online]. Available: https://www.portugal.gov.pt/download-ficheiros/ficheiro.aspx?v=28d4b86b-9b43-4005-a968-b4c4730f28a7.

[14] Infante, 2016. [Online]. Available: http://infante.space/.

[15] H. Butz, "The Airbus Approach to Open Integrated Modular Avionics (IMA): Technology, Methods, Processes and Future Road Map", in *Proc. 1st International Workshop on Aircraft System Technologies (AST 2007)*, Hamburg, 2007.

[16] ——, "Open Integrated Modular Avionic (IMA): State of the Art and future Development Road Map at Airbus Deutschland", Department of Avionic Systems at Airbus Deutschland GmbH, Tech. Rep., 2010.

[17] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[18] C. B. Watkins and R. Walter, "Transitioning from Federated Avionics Architectures to Integrated Modular Avionics", in *Proc. IEEE/AIAA 26th Digital Avionics Systems Conf.*, Dallas, TX, 2007.

[19] P. F. Schikora, "A Cost Trade-Off Analysis of F-16 Line Replaceable Unit (LRU) Packaging Options", Master's thesis, Dep. of the Air Force, Air University, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, Dec. 1988.

[20] B. Witwer, "Systems Integration of the 777 Airplane Information Management System (AIMS): A Honeywell Perspective", in *Proc. AIAA/IEEE 14th Digital Avionics Systems Conference*, Cambridge, MA, 1995.

[21] P. J. Prisaznuk, "Integrated Modular Avionics", in *Proc. IEEE 1992 National Aerospace and Electronics Conference*, Dayton, OH, 1992, pp. 39–45.

[22] *Avionics Application Software Standard Interface Part 1 - Required Services*, Specification 653-2, ARINC, Dec. 2005.

[23] *Avionics Application Software Standard Interface Part 2 - Extended Services*, Specification 654P2-1, ARINC, Dec. 2008.

[24] C. Silva, J. Cristóvão, and T. Schoofs, "An I/O Building Block for the IMA Space Reference Architecture", in *Proc. DASIA 2012 - DAta Systems In Aerospace*, vol. 701, Drubrovnik, 2012.

[25] F. Torelli, C. Taylor, A. V. Sanchez, P. Mendham, and S. Fowell, "SOIS and Software Reference Architecture", in *Proc. DASIA 2011 - DAta Systems In Aerospace*, vol. 694, San Anton, Malta, 2011.

[26] J. Windsor. (Nov. 2010). IMA-SP and Security, ADCSS 2010 - 4th ESA Workshop on Avionics, Data, Control and Software Systems, [Online]. Available: https://indico.esa.int/event/63/contributions/.

[27] R. Kaiser and S. Wagner, “Evolution of the PikeOS Microkernel”, in *Proc. International Workshop on Microkernels for Embedded Systems*, Sydney, 2007.

[28] M. Masmano, I. Ripoll, and A. Crespo, “An overview of the XtratuM nanokernel”, in *Proc. 2005 Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Palma de Mallorca, 2005.

[29] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge, “Xtratum: a Hypervisor for Safety Critical Embedded Systems”, in *Proc. 11th Real-Time Linux Workshop*, Dresden, 2009, pp. 263–272.

[30] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor, “ARINC 653 interface in RTEMS”, in *Proc. DASIA 2007 - DAta Systems In Aerospace*, vol. 638, Naples, 2007.

[31] J. Rufino, J. Craveiro, T. Schoofs, C. Tatibana, and J. Windsor, “AIR Technology: A Step Towards ARINC 653 in Space”, in *Proc. DASIA 2009 - DAta Systems In Aerospace*, Istanbul, 2009.

[32] H. Silva, J. Sousa, D. Freitas, S. Faustino, A. Constantino, and M. Coutinho, “RTEMS Improvement – Space Qualification of RTEMS Executive”, in *Proc. INForum Simpósio de Informática 2009*, Lisbon, 2009.

[33] K. Hjortnaes, J. Miro, J. Busseuil, and T. Duhamel. (Nov. 2010). AVIONICS Architecture: SAVOIR and Beyond, ADCSS 2010 - 4th ESA Workshop on Avionics, Data, Control and Software Systems, [Online]. Available: `https://indico.esa.int/event/63/contributions/`.

[34] K. Hjortnaes. (Oct. 2011). Introduction and Status of SAVOIR, ADCSS 2011 - 5th ESA Workshop on Avionics, Data, Control and Software Systems, [Online]. Available: `https://indico.esa.int/event/62/contributions/`.

[35] J.-L. Terraillon. (Nov. 2010). SAVOIR-FAIRE status and perspective, ADCSS 2010 - 4th ESA Workshop on Avionics, Data, Control and Software Systems, [Online]. Available: `https://indico.esa.int/event/63/contributions/`.

[36] *Space product assurance Software product assurance*, Standard ECSS-Q-ST-80C Rev.1, ESA Requirements and Standards Division, Feb. 2017.

[37] *Space engineering Software*, Standard ECSS-E-ST-40C, ESA Requirements and Standards Division, Mar. 2009.

[38] *Space engineering Software engineering handbook*, Handbook ECSS-Q-HB-40A, ESA Requirements and Standards Division, Dec. 2013.

[39] R. P. Goldberg, “Survey of Virtual Machine Research”, *Computer*, vol. 7, no. 6, pp. 34–45, Jun. 1974.

[40] R. A. Meyer and L. H. Seawright, “A virtual machine time-sharing system”, *IBM Systems Journal*, vol. 9, no. 3, pp. 199–218, Sep. 1970.

[41] J. P. Buzen and U. O. Gagliardi, “The evolution of virtual machine architecture”, in *AFIPS Conf. Proc. National Computer Conference and Exposition*, New York, NY, 1973.

[42] G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[43] *AMD64 Virtualization Codenamed “Pacifica”Technology Secure Virtual Machine Architecture Reference Manual*, Revision 3.01, Advanced Micro Devices, Inc., May 2005.

[44] *Intel Vanderpool Technology for IA-32 Processors (VT-x) Preliminary Specification*, Intel Corporation, Jan. 2005.

[45] R. Mijat and A. Nightingale, *Virtualization is Coming to a Platform Near You*, White Paper, ARM Limited, 2011. [Online]. Available: `https://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`.

[46] *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, White Paper, VMware, Inc, Mar. 2008. [Online]. Available: `https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html`.

[47] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, “Binary translation”, *Communications of the ACM*, vol. 36, no. 2, pp. 69–81, Feb. 1993.

[48] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, “Dynamic Binary Translation and Optimization”, *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 529–548, Jun. 2001.

[49] A. Whitaker, M. Shaw, and S. D. Gribble, “Denali: Lightweight virtual machines for distributed and networked applications”, University of Washington, Seattle, WA, Tech. Rep. 02-02-01, 2002.

[50] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the Art of Virtualization”, in *SOSP’03 Proc. 19th ACM Symposium on Operating Systems Principles*, vol. 37, Bolton Landing, NY, 2003, pp. 164–177.

[51] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors”, in *Proc. 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems 2007*, vol. 41, Lisbon, 2007, pp. 275–287.

[52] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment”, *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[53] N. Diniz and J. Rufino, “ARINC 653 in Space”, in *Proc. DASIA 2015 - DAta Systems In Aerospace*, vol. 602, Edinburgh, 2005.

[54] C. Silva, “Integrated Modular Avionics for Space Applications: Input/Output Module”, Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, Oct. 2012.

[55] *Space product assurance Failure modes, effect (and criticality) analysis (FMEA/FMECA)*, Standard ECSS-Q-ST-30-02C, ESA Requirements and Standards Division, Mar. 2009.

[56] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, “The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools”, *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, p. 36, Apr. 2008.

[57] J. Seronie-Vivien and C. Cantenot, “RTEMS Operating System Qualification”, in *Proc. DASIA 2005 - DAta Systems In Aerospace*, vol. 602, Edinburgh, 2005.

[58] C. Silva and C. Tatibana, “MultIMA - Multi-Core in Integrated Modular Avionics”, in *Proc. DASIA 2014 - DAta Systems In Aerospace*, vol. 725, Warsaw, 2014.

[59] B. Walshe, *A Brief History of ARM: Part 1*, ARM Limited, Apr. 2015. [Online]. Available: `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-1`.

[60] ——, *A Brief History of ARM: Part 2*, ARM Limited, May 2015. [Online]. Available: `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-2`.

[61] I. Thornton, *ARM: Investing for future growth*, ARM Limited, Q2 2018. [Online]. Available: `https://arm.com/-/media/global/company/investors/Financial%20Result%20Docs/Arm_SB_Q2_2018_Roadshow_Slides_FINAL.pdf`.

[62] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere, “Formal virtualization requirements for the ARM architecture”, *Journal of Systems Architecture*, vol. 59, no. 3, pp. 144–154, 2013.

[63] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung, “ARMvisor: System Virtualization for ARM”, in *Proc. Linux Symposium*, Ottawa, Ontario, 2012, pp. 93–107.

[64] *The SPARC Architecture Manual Version 8*, SPARC International, Inc., Campbell, CA, 1991.

[65] *GR740 Data Sheet and User’s Manual*, Cobham, Jul. 2018.

[66] *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition*, ARM Limited, Mar. 2018.

[67] *ARM® Compiler armasm User Guide*, ARM Limited, Oct. 2018.

[68] *ARM Architecture Reference Manual Thumb-2 Supplement*, ARM Limited, Dec. 2005.

[69] *Cortex™-A9 Technical Reference Manual*, ARM Limited, Jun. 2012.

[70] *Zynq-7000 SoC Technical Reference Manual*, Xilinx, Jan. 2018.

[71] *ARM® Generic Interrupt Controller Architecture version 2.0 Architecture Specification*, ARM Limited, Jul. 2013.

[72] *Cortex™-A9 MPCore® Technical Reference Manual*, ARM Limited, Jun. 2012.

[73] *Arty Z7 Reference Manual*, Digilent. [Online]. Available: `https://reference.digilentinc.com/reference/programmable-logic/arty-z7/reference-manual`.

[74] E. W. Dijkstra, “Solution of a Problem in Concurrent Programming Control”, *Communications of the ACM*, vol. 8, no. 9, p. 569, Sep. 1965.

A. AIR code

```
void pmk_init(void) {

   /* BSP Initialization */
   if (bsp_core_init() != 0) {
      pmk_fatal_error(PMK_INTERNAL_ERROR_BSP, __func__, __FILE__, __LINE__);
   }

   /* USR configuration initialization */
   pmk_configurations_int();

   /* Workspace initialization */
   pmk_workspace_init();

   /* initializes the PMK memory segregation */
   pmk_segregation_int();

   /* initialize other cores */
   pmk_multicore_init();

   /* Initialize user health monitor tables */
   pmk_hm_init();

   /* Partition initialization */
   pmk_partitions_init();

   /* Schedule Initialization */
   pmk_scheduler_init();

   /* communication channels initialization */
   pmk_channels_init();

   /* Initialize Time of Day */
   pmk_tod_initialization();

   /* Set HM state */
   pmk_hm_set_state(AIR_STATE_MODULE_EXEC);

   /* enable preemption */
   bsp_core_ready();

   printk("    :: BSP core ready\n");

   /* Idle loop */
   bsp_idle_loop();
   return;
}
```

Figure A.1.: PMK initialization

```
<!-- HM configuration -->
<System_HM_Table>
   <System_State_Entry Description="Module Execution" SystemState="0">
      <Error_ID_Level Description="Power Interrupt" ErrorIdentifier="0" ErrorLevel="MODULE"/>
      <Error_ID_Level Description="Illegal Instruction" ErrorIdentifier="1" ErrorLevel="MODULE"/>
      <Error_ID_Level Description="Segmentation Error" ErrorIdentifier="2" ErrorLevel="MODULE"/>
      <Error_ID_Level Description="Unimplemented Error" ErrorIdentifier="3" ErrorLevel="MODULE"/>
      <Error_ID_Level Description="Floating Error" ErrorIdentifier="4" ErrorLevel="MODULE"/>
      <Error_ID_Level Description="Overflow Error" ErrorIdentifier="5" ErrorLevel="MODULE"/>
      <Error_ID_Level Description="Divide by zero" ErrorIdentifier="6" ErrorLevel="MODULE"/>
   </System_State_Entry>
   <System_State_Entry Description="Partition Execution" SystemState="1">
      <Error_ID_Level Description="Power Interrupt" ErrorIdentifier="0" ErrorLevel="PARTITION"/>
      <Error_ID_Level Description="Illegal Instruction" ErrorIdentifier="1" ErrorLevel="PARTITION"/>
      <Error_ID_Level Description="Segmentation Error" ErrorIdentifier="2" ErrorLevel="PARTITION"/>
      <Error_ID_Level Description="Unimplemented Error" ErrorIdentifier="3" ErrorLevel="PARTITION"/>
      <Error_ID_Level Description="Floating Error" ErrorIdentifier="4" ErrorLevel="PARTITION"/>
      <Error_ID_Level Description="Overflow Error" ErrorIdentifier="5" ErrorLevel="PARTITION"/>
      <Error_ID_Level Description="Divide by zero" ErrorIdentifier="6" ErrorLevel="PARTITION"/>
   </System_State_Entry>
</System_HM_Table>

<Module_HM_Table>
   <System_State_Entry Description="Module Execution" SystemState="0">
      <Error_ID_Action Action="SHUTDOWN" Description="Power Interrupt" ErrorIdentifier="0"/>
      <Error_ID_Action Action="SHUTDOWN" Description="Illegal Instruction" ErrorIdentifier="1"/>
      <Error_ID_Action Action="SHUTDOWN" Description="Segmentation Error" ErrorIdentifier="2"/>
      <Error_ID_Action Action="SHUTDOWN" Description="Unimplemented Error" ErrorIdentifier="3"/>
      <Error_ID_Action Action="IGNORE" Description="Floating Error" ErrorIdentifier="4"/>
      <Error_ID_Action Action="SHUTDOWN" Description="Overflow Error" ErrorIdentifier="5"/>
      <Error_ID_Action Action="SHUTDOWN" Description="Divide by zero" ErrorIdentifier="6"/>
   </System_State_Entry>
</Module_HM_Table>

<Partition_HM_Table PartitionIdentifier="1" PartitionName="p0">
   <System_State_Entry Description="Partition Execution" SystemState="1">
      <Error_ID_Action Action="IGNORE" Description="Power Interrupt" ErrorIdentifier="0"/>
      <Error_ID_Action Action="COLD_START" Description="Illegal Instruction" ErrorIdentifier="1"/>
      <Error_ID_Action Action="COLD_START" Description="Segmentation Error" ErrorIdentifier="2"/>
      <Error_ID_Action Action="COLD_START" Description="Unimplemented Error" ErrorIdentifier="3"/>
      <Error_ID_Action Action="IGNORE" Description="Floating Error" ErrorIdentifier="4"/>
      <Error_ID_Action Action="COLD_START" Description="Overflow Error" ErrorIdentifier="5"/>
      <Error_ID_Action Action="COLD_START" Description="Divide by zero" ErrorIdentifier="6"/>
   </System_State_Entry>
</Partition_HM_Table>
```

Figure A.2.: HM configuration `.xml`

```c
#include <air.h>  /* includes the air_hm_event_t structure */
#include <rtems.h>

/**
 * @brief Health-Monitor ISR handler
 */
static void hm_isr_handler(void) {

   /* get HM event */
   air_hm_event_t hm_event;
   air_syscall_get_hm_event(&hm_event);
}

/**
 * @brief Main RTEMS task
 */
rtems_task Init(rtems_task_argument ignored) {

    /* register HM ISR handler */
    rtems_isr_entry isr_ignored;
    rtems_interrupt_catch(
            (rtems_isr_entry)hm_isr_handler,
            AIR_IRQ_HM_EVENT,
            &isr_ignored);

    /* call entry point */
    if (producer != NULL) {
        producer();
    }

    rtems_task_delete(RTEMS_SELF);
}
```

Figure A.3.: Partition HM handler

```
    .arm
    .extern svc_handler

global(exception_svc)
    srsfd sp!, #ARM_PSR_SYS
    cpsie aif, #ARM_PSR_SYS

    push  {r0-r3, r12, lr}

    /* get pmk_core_ctrl_t */
    mrc   p15, 0, r0, c13, c0, 4

    mov   r2, sp
    ldr   r1, [sp, #28] // spsr

    /* retrieving the svc id */
    tst   r1, #ARM_PSR_T
    ldrneh r1, [lr, #-2]
    bicne r1, r1, #0xff00
    ldreq r1, [lr, #-4]
    biceq r1, r1, #0xff000000

    /* if return from trap, just exit peacefully */
    teq   r1, #(AIR_SYSCALL_ARM_RETT)
    beq   arm_rett

    BL2C  arm_svc_handler // branch to C code

    pop   {r0-r3, r12, lr}

    rfefd sp!
```

Figure A.4.: ARM HVC handler

```
air_uptr_t arm_hm_handler(air_u32_t id, air_u32_t far, air_u32_t fsr, air_u32_t *instr) {

   pmk_core_ctrl_t *core = (pmk_core_ctrl_t *)arm_cp15_get_Per_CPU();

   air_error_e error_id;
   air_uptr_t ret = NULL;

   if (id == ARM_EXCEPTION_UNDEF) {

      error_id = AIR_UNIMPLEMENTED_ERROR;

      /* Lazy Switching. fsr is the spsr */
      if ((fsr && ARM_PSR_T)) {

         if ( ((*instr & 0xef00) == 0xef00) ||
               ((*instr & 0x0e10ef00) == 0x0a00ee00) ||
               ((*instr & 0x0e00ee00) == 0x0a00ec00) ||
               ((*instr & 0xff10) == 0xf900) ||
               ((*instr & 0x0e10ef00) == 0x0a10ee00) ||
               ((*instr & 0x0e00efe0) == 0x0a00ec40) )
            error_id = AIR_FLOAT_ERROR;
      } else {
         if ( ((*instr & 0xfe000000) == 0xf2000000) ||
               ((*instr & 0x0f000e10) == 0x0e000a00) ||
               ((*instr & 0x0e000e00) == 0x0c000a00) ||
               ((*instr & 0xff100000) == 0xf4000000) ||
               ((*instr & 0x0f000e10) == 0x0e000a10) ||
               ((*instr & 0x0fe00e00) == 0x0c400a00) )
            error_id = AIR_FLOAT_ERROR;
      }

   } else if (id == ARM_EXCEPTION_PREF_ABORT) {

   (...)

   pmk_hm_isr_handler(error_id);

   if (!core->context->trash)
      ret = arm_partition_hm_handler(id, core);

   return ret;
}
```

Figure A.5.: ARM lazy FPU implementation

B. ARM processor modes

Processor mode		Encoding	Privilege level
User	`usr`	10000	PL0
FIQ	`fiq`	10001	PL1
IRQ	`irq`	10010	PL1
Supervisor	`svc`	10011	PL1
Monitor	`mon`	10110	PL1
Abort	`abt`	10111	PL1
Hypervisor	`hyp`	11010	PL2
Undefined	`und`	11011	PL1
System	`sys`	11111	PL1

Table B.1.: ARM operating modes

C. SPARC Exception Table

Exception Name	Priority	offset
reset	1	0x0
data_store_error	2	0x2b
instruction_access_MMU_miss	2	0x3c
instruction_access_error	3	0x21
r_register_access_error	4	0x20
instruction_access_exception	5	0x01
privileged_instruction	6	0x03
illegal_instruction	7	0x02
fp_disabled	8	0x04
cp_disabled	8	0x24
unimplemented_FLUSH	8	0x25
watchpoint_detected	8	0x0b
window_overflow	9	0x05
window_underflow	9	0x06
mem_address_not_aligned	10	0x07
fp_exception	11	0x08
cp_exception	11	0x28
data_access_error	12	0x29
data_access_MMU_miss	12	0x2c
data_access_exception	13	0x09
tag_overflow	14	0x0a
division_by_zero	15	0x2a
trap_instruction	16	0x80 - 0xff
interrupt_level_15	17	0x1f
interrupt_level_14	18	0x1e
interrupt_level_13	19	0x1d
interrupt_level_12	20	0x1c
interrupt_level_11	21	0x1b
interrupt_level_10	22	0x1a
interrupt_level_9	23	0x19
interrupt_level_8	24	0x18

interrupt_level_7	25	0x17
interrupt_level_6	26	0x16
interrupt_level_5	27	0x15
interrupt_level_4	28	0x14
interrupt_level_3	29	0x13
interrupt_level_2	30	0x12
interrupt_level_1	31	0x11
impl.-dependent exception	impl.-dependent	0x60 - 0x7f

Table C.1.: SPARC Trap Table

D. Configuration Registers

Index	DBP	r1	DBPM	r2	reserved	CS	CF	DW	SV	LD	FPU	M	V8	NWP	NWIN
0 - 3	0	0	1	0	-	0	0 0	0	0	0	0 1	0	1	1 0 0	1 1 1
r	rw	rw	rw	rw	-	r	r	rw	rw	r	r	r	r	r	r

31:28 Processor index.

27 Disable Branch Prediction (DBP): disable branch prediction when set to 1.

26 reserved

25 Disable Branch Prediction on instruction cache misses (DBPM): set to 1 to avoid instruction cache fetches for predicted instructions that may be annulled.

24 reserved

23:18 reserved

17 Clock Switching (CS): is 0 when the implementation does not support clock switching.

16:15 CPU Clock Frequency (CF): is 0 when the CPU runs at the same frequency as the AMBA bus.

14 Disable Write Trap Error (DWT): when set to 1 error write traps (`tt` = `0x2b`) are ignored. Set to 0 on reset.

13 Single-Vector Trapping: if 1 will enable single-vector trapping.

12 Load Delay (LD): is 0 so a 1-cycle delay is used.

11:10 FPU option = `0b01` = GRFPU.

9 Multiply and accumulate (M): is 0 so MAC instructions are unsupported.

8 SPARC V8: is 1, multiply and divide instructions are available.

7:5 Number of implemented Watchpoints (NWP): number of implemented watchpoints is 4.

4:0 Number of register Windows (NWIN): number of implemented windows is 7.

Table D.1.: ASR17: LEON4 configuration register

reserved	CPUID	ACKINTID
-	cpuid	ackintid
-	rw	rw

31:13 reserved.

12:10 CPUID: identifies the processor that requested the interrupt.

9:0 ACKINTID: the interrupt id. A read acts as an acknowledgment for the interrupt.

Table D.2.: ICCIAR: Interrupt Acknowledge Register

Table D.3.: ahahaS