An Architectural Journey into RISC Architectures for HPC Workloads

Ying Hao Xu Lin

Director: Filippo Mantovani [1]

Codirector: Eduard Ayguadé [2]

[1] Barcelona Supercomputing Center (BSC)
[2] Universitat Politècnica de Catalunya, Barcelona Tech (UPC)

Facultat d'Informàtica de Barcelona (FIB)

This dissertation is submitted for the degree of

Master in Innovation and Research in Informatics:
High Performance Computing

Barcelona, 28th of January 2019

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Acknowledgments

I would like to thank my two advisors, Filippo Mantovani and Eduard Ayguadé for their support and for offering me the opportunity to grow as a researcher by working in very interesting and motivating projects.

A special mention to all my teammates, Constan, Fabio, Kilian and Guillem. Thank you very much for your support and for making the days in the office more entertaining and more productive.

I would like to thank to all my family and friends, especially to my grandparents. For your maximum support and the trust you put on me during all this project. There is not enough words to express how grateful I am. To Cristina, for being there in the best and in the worst moments, for cheering me up when things were not working, for all the sacrifices that you've made on my behalf. Thank you!

Finally, I would like to thank all my colleagues at the Barcelona Supercomputing Center for providing an excellent working environment. And to the research center, I cannot think about a better place to do research. Thank you for bringing me this opportunity.

“Success is a journey, not a destination. The doing is often more important than the outcome.”

- Arthur Ashe

Abstract

The race to the Exascale (*i.e.*, 10^{18} Floating Point operations per seconds) together with the slow-down of Moore's law are posing unprecedented challenges to the whole High-Performance Computing (HPC) community. Computer architects, system integrators and software engineers studying programming models for handling parallelism are especially called to the rescue in a moment like the one in which we are living.

While studying the current HPC market, a careful observer can notice that *i)* the dominance of a single x86 is fading; *ii)* as a consequence of the previous point, new CPU architectures and accelerators are gaining relevance (*e.g.* RISC CPUs and GP-GPUs); *iii)* also, new workloads coming from industry 4.0 and automotive (*e.g.* machine learning) are requiring more and more computational resources. Thus, driving the development of next-generation computational systems.

This thesis explores the boundary of these three observations evaluating the current state-of-the-art of emerging RISC architectures in HPC (Arm and RISC-V). It studies the performance, the instantaneous power consumption and total energy spent to reach the solution of a scientific problem in heterogeneous System-on-Chips (SoCs). For the evaluation, four platforms have been tested: two heterogeneous Arm platforms (CPU+GPU and CPU+FPGA), one RISC-V platform and one Open Source RISC-V core running in an FPGA.

The added values of the thesis come from the fact that:

A. The evaluation of the aforementioned platforms has been performed using a machine learning test-case based on the k-means clustering algorithm related to predictive maintenance and failure detection provided by an industrial partner. While preparing this master thesis, I was in fact involved in the research activities within the collaboration between the Barcelona Supercomputing Center (BSC) and Aingura IIoT.

B. The tests of the k-means algorithm on the RISC-V core implied the implementation of a System on Chip allowing the interaction with the RISC-V core. Even if the Ariane core itself is freely available online, the work of having peripherals for minimal I/O operations and performance counters required careful work on FPGA using a hardware description language (SystemVerilog).

As expected, the more mature Arm Cortex A57 processor outperformed the rest of the platforms and the best RISC-V platform shown to perform as good as the Arm Cortex A9. For the heterogeneous platforms, the studied CPU+GPU system achieved the best performance but the CPU+FPGA used less energy when considering only the active power of the execution. The document makes special emphasis on the reproducibility of the experiments by explaining step-by-step how to set up an FPGA-based research platform using an Open Source RISC-V core and how to interact with the hardware counters defined in RISC-V in order to measure the performance.

The research work behind this thesis generated two contributions to international conferences: the poster *Is Arm software ecosystem ready for HPC?* presented at the Supercomputing conference 2017 [1] and the paper *Implementation of the K-means Algorithm on Heterogeneous Devices: a Use Case Based on an Industrial Dataset* for which I have been corresponding author and speaker at the ParCo conference in September 2017 [2]. My work on RISC-V cores contributed to the preliminary architectural studies performed for the European Processor Initiative (EPI) project, a European project for developing next-generation HPC technology for data centers and automotive industry.

This thesis is organized as follows:

- Chapter 1 presents the motivation of this thesis and the related work;
- Chapter 2 introduces the two RISC architectures we are focusing on, `Arm` and `RISC-V`;
- Due to the less mature environment of `RISC-V`, the entire Chapter 3 focuses on discussing `RISC-V`, including the methods available today to run HPC applications;
- Chapter 4 focuses on explaining the scientific problem derived from a real industrial use-case. The chapter also discusses the optimizations that I applied to map a clustering algorithm into an algebraic matrix manipulation problem.
- Chapter 5 introduces the methods and the evaluation of four RISC platforms using the scientific application described in Chapter 4.
- Chapter 6 wraps up with the conclusions followed by Chapter 7, where the future work and next steps are presented.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High Performance Computing (HPC) has become an essential part of the scientific method. HPC typically refers to the practice of combining multiple computers such in a way that the whole system can works toward solving a large and complex computational problem, generally from the science, engineering or business field. Scientists are taking advantage of the large computation capacity of HPC systems to simulate and study complex situations or phenomena, filling in several cases the gap between experiments and theory. For this reason, HPC is often called the ‘‘third pillar’’ of science [3].

Even if High Performance Computing systems are considered a fundamental tool in order to conduct a research, these systems are so complex that they are a research topic itself. As an HPC system is made of simple computers (also known as *nodes*), the current research in HPC tackles the same problems found in computer research but at a higher scale.

With the end of Moore's law [4], the trend in performance growth has recently been slowing down so domain specific architectures have become more prominent. These architectures are both more energy efficient and faster than general purpose ones, however, as the name says, they are only designed for accomplishing specific tasks. The System-on-Chips (SoC) powering our smartphones are one of the best examples of a system containing domain specific units (*e.g.* wireless radio unit).

Taking advantage of the less complex RISC (Reduced Instruction Set Computer) architecture, current SoCs can integrate multiple domain specific units combined with a general purpose processor enabling a balance between energy consumption and performance.

In the following sections we explain the RISC architectural approach, the motivation and the related work, and finally, we describe the organization of this document.

1.1 Reduced Instruction Set Computer (RISC)

RISC stands for Reduced Instruction Set Computer. This concept is more related to the type of instructions that are executed rather than the actual implementation of the processor. A processor typically implements an Instruction Set Architecture (ISA), this is, a set of well defined instructions and encodings. A RISC ISA is the one that defines a small set of simple and general instructions.

The other existing approach is Complex Instruction Set Computer (CISC). A CISC ISA defines complex instructions, each of which can execute several low-level operations (*e.g.* load plus an arithmetic operation). The main representative of this category is the `x86` architecture from Intel. Although the RISC approach may sound easier to implement than CISC, modern Intel processors read CISC instructions but the instructions are later converted into RISC-like simpler instructions (Micro-Operations) at the back-end.

The RISC approach requires a higher number of cycles per instructions while in CISC architecture a single instruction can take multiple cycles due to its complexity. However instruction complexity and throughput are not the only differences between CISC and RISC. Typically the RISC ISAs use a uniform instruction length and the memory is strictly accessed using separate load/store instructions.

For these reasons, nowadays, the industry is focusing on RISC-like architectures. There are several RISC architectures that are used today but the one with more presence is `Arm` (Advanced RISC Machine). `Arm` processores are used in almost all the embedded devices (*e.g.* smartphones and IoT) and became a really popular architecture due to its embedded nature and the lower power consumption[1].

The next big thing to happen in computer architecture is `RISC-V` (pronounced "risk-five"). `RISC-V` as the name states, also falls in the RISC-like architecture category but the main difference is that is free and the ISA is open. This means that anyone can build a `RISC-V` processor and commercialize it without having to pay any license. In `Arm`'s case, if a company wants to commercialize an `Arm` processor, it has to pay licenses to use the `Arm` instruction set even if the processor is not designed by `Arm`.

Due to the free nature of `RISC-V`, computer microarchitecture researchers are benefiting of online available open-source `RISC-V` cores to conduct their experiments. On the other hand, companies can take advantage of the `RISC-V` ISA, as they can design their own processors targeting their specific problem (domain specific architectures) and commercialize them without having to pay any license.

1.2 Motivation

The work presented in this thesis started with a collaboration between Barcelona Supercomputing Center (BSC) and Aingura IIoT. Aingura IIoT is a Spanish industrial manufacturing company, part of the Etxe-tar group, specialized on industrial IoT solutions. Their solutions focus on optimizing the operation of industrial machines, *e.g.* implementing predictive maintenance techniques.

Part of the Aingura IIoT solutions involves machine learning workloads. These are well known to be compute intensive workloads often classified as HPC workloads. The collaboration between BSC and Aingura IIoT started looking at a real industrial problem where a machine uses a laser to apply a treatment (temper) to a surface. Monitoring the temper process and combining it with on-line machine learning algorithms, BSC and Aingura IIoT worked on predictive maintenance and failure detection techniques. In the temper process, two independent monitoring systems are providing feedback to an embedded board housing a Xilinx Zynq 7020 with two Arm Cortex-A9 and a reconfigurable logic (FPGA). The requirement of Aingura IIoT was to detect anomalies in the temper process within a time frame using clustering techniques (k-means).

The implementation, early tests and the evaluation I performed within this collaboration were collected and presented in the paper *Implementation of the K-means Algorithm on Heterogeneous Devices: a Use Case Based on an Industrial Dataset* [2], which I presented in the ParaFPGA mini-symposium in conjunction with ParCo2017 conference.

Within the collaboration with Aingura IIoT I continued the work started in the paper and extended it to more platforms, *i.e.*, other state-of-the-art `Armv8` and `RISC-V` cores. As in the paper we were trying to solve a real industrial problem, this research was motivated by the interest of knowing how new architectures, such as `RISC-V`, are performing while solving compute intensive problems coming from a real industrial environment.

1.3 Related work

The work presented in this thesis tackles three research areas: *i)* `Arm` in HPC, *ii)* `RISC-V`, and *iii)* Clustering techniques and industrial data analysis. In this section I present the papers, books and online resources that I used to build my background on each of this topic.

1.3.1 `Arm` in HPC

The Arm architecture is gaining importance in the race to Exascale [5]. Several international projects announced the adoption of Arm technology for high-end production HPC systems, *e.g.* the European

[1] It is important to remember that the fact that a processor is `Arm` does not mean that it will be really energy efficient. The main benefit of `Arm` would be more on the opportunities that the `Arm` ISA offers to the designers in order to optimize the power consumption.

Mont-Blanc[2], the Japanese Post-K[6], and the UK's GW4/EPSRC[3]. Also, in November 2018 the Astra supercomputer, powered by the Arm-based Marvell's ThunderX2 installed at the Sandia National Laboratories (US), has been ranked 204 in the Top500 list [7].

For more than six years, research projects in collaboration with industry evaluated Arm-based systems for parallel and scientific computing advocating the higher efficiency of this technology mutated from the mobile and the embedded market. The Barcelona Supercomputing Center has been pioneer in delivering the first mobile-based HPC research clusters. Several papers have been published with the preliminary analysis of benchmarks and performance projections of Arm-based SoCs coming from the mobile and embedded market [8, 9, 10]. More recently tests on Arm-based server SoCs also appeared in the literature [11, 12].

The AXIOM project[4] also demonstrated that embedded platforms using `Arm` cores plus an accelerator (in this case a Field Programmable Gate Array or simply FPGA) are serious competitive options [13].

1.3.2 RISC-V

The `RISC-V` project originated in 2010 at the University of California, Berkeley [14] for designing small, fast, and low-power CPUs [15]. As of May 2017, version 2.2 of the userspace ISA is fixed and the privileged ISA is available as draft version 1.10[5].

The Parallel Ultra Low Power (PULP) Platform[6] aims to develop an open, scalable hardware and software research and development platform based on `RISC-V` [16, 17, 18]. The same authors of the PULP platforms released the HERO platform [19], which is an heterogeneous embedded system on chip (HESoC) that combines general-purpose with domain-specific architectures. The HERO platform combines PULP-based cores implemented on FPGA with an `Arm` Cortex-A multicore processor (host) running Linux.

The PULP platform also developed efficient 32-bit and 64-bit `RISC-V` cores. These cores are available in Github (open) and one of their 32-bit core, RI5CY, was already included in Google's Pixel Visual Core Image Processing Unit (IPU) [20]. This IPU is currently present on commercial devices like the Pixel 2 smartphone. For this thesis I used their 64-bit core called Ariane[7].

1.3.3 Clustering techniques and industrial data analysis

Clustering methods are used to identify groups of similar objects in multivariate data sets collected from images, sensors or data sources in general. An example of research using the same algorithm that I use in this thesis (k-mean) but applied to text mining is described in [21].

The term *Industry 4.0* refers to the current trend in automation and manufacturing technologies of using data sensing, cyber-physical systems, Internet of things, cloud computing and cognitive computing for improving the efficiency of the manufacturing process. A review of Industry 4.0 is introduced in [22], while the computational challenges introduced by Industry 4.0 can be found in [23].

As mentioned in the introduction, in this thesis we use the data provided by a process of laser temper explained by our colleagues of Aingura IIoT / Etxe-tar in [24]. Machine learning techniques in manufacturing are described in [25, 26].

1.3.4 Energy analysis

The interest of the scientific community and the system integrators towards the energy efficiency of the processors in HPC systems has significantly grown during the last years. With the increase of power consumption, more energy must be dissipated. Ware-house scale computer facilities are having serious problems to effectively cool the systems. The cost of maintaining a ware-house scale computer has increased due to the extra cost of keeping everything cool and the environmental footprint is starting

[2] http://montblanc-project.eu/
[3] https://gw4-isambard.github.io/docs/
[4] http://www.axiom-project.eu
[5] https://riscv.org/specifications/
[6] https://www.pulp-platform.org
[7] https://github.com/pulp-platform/ariane

to be non-negligible too [27]. In fact, energy has become the primary cost driver for data centers. A comprehensive study of the problems related to the deployment of a data-center, including observations about energy and power consumption can be found in [28]

Also, in [29], the importance of analyzing the performance and the energy-efficiency in multi-node clusters is shown followed by a discussion on how important is to use good analysis techniques in order to be able to correlate performance and power. In this thesis, we covered the impact of both performance and energy consumption in heterogeneous platforms when the CPU and the accelerator are used.

Chapter 2

RISC Architectures

As mentioned in Section 1.1, RISC stands for Reduced Instruction Set Computer and represents a class of CPU architectures. In this section we are going to discuss the two most popular RISC architectures available today: `Arm` and `RISC-V`. A short introduction of each architecture will be provided followed by the current software ecosystem and the available platforms.

2.1 Advanced RISC Machine (Arm)

Advanced RISC Machine or simply `Arm`, is a family of RISC architectures for computer processors. It belongs to Arm Holdings, a British multinational semiconductor and software design company owned by the SoftBank group. The main `Arm` product is the `Arm` ISA but the company also designs its own processors based on the `Arm` instruction set. Both products are provided under the form of intellectual property (IP) to partners that needs to pay a fee (license) to build final silicon out of the `Arm` IPs.

The `Arm` architecture originally targeted low power embedded systems that do not require high compute capacity, however over time, `Arm` has improved the compute capacity of its designs, supporting 32-bit address space in versions ARMv3 to Armv7 and introducing a new 64-bit support in the Armv8 version. Since Armv7-A version of the ISA, `Arm` introduced its Single Instruction Multiple Data (SIMD) support, called NEON. The NEON extension allows to operate with registers of 128 bits using IEEE compliant arithmetic. During 2016 `Arm` announced the Scalable Vector Extension (SVE) as part of the Armv8.2-A architecture [30]. SVE allows implementation choices for vector lengths that scale from 128 to 2048 bits.

With the computation capability increasing every new generation, `Arm` processors found its way into the HPC market. The European project Mont-Blanc significantly contributed to adoption of `Arm` within HPC. Since 2011 in fact the Mont-Blanc consortium pushes `Arm` into HPC deploying `Arm`-based HPC clusters and improving the system software required for the adoption by the scientific community and the data center. In the early days of the Mont-Blanc project, the `Arm` software ecosystem for executing HPC applications was not mature enough. In the framework of the Mont-Blanc project a significant effort was put in order to develop the HPC software ecosystem required to run HPC applications in the same way as in traditional `x86` based machines.

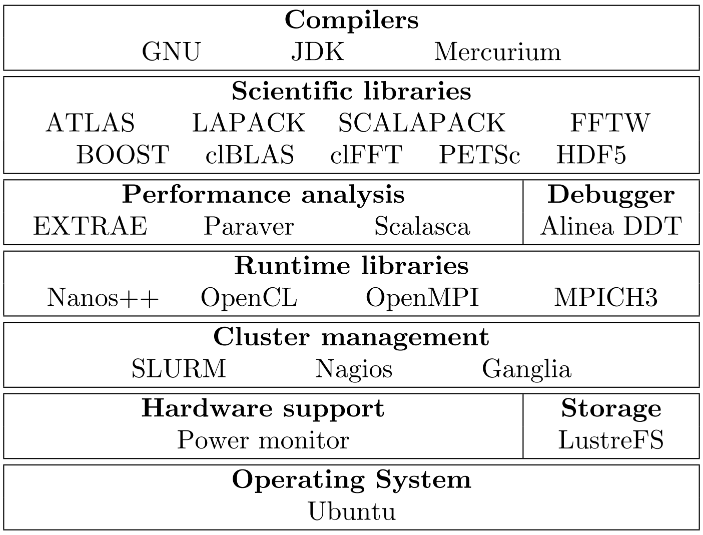
System software in an HPC node refers to the complete set of libraries and utilities required to run scientific applications on a distributed fashion. Figure 2.1 shows the software stack of the Mont-Blanc prototype [10]. In the following sections we will detail the parts of the system software that have been improved or delivered from scratch by the Mont-Blanc project for enabling the `Arm` architecture in HPC.

2.1.1 Compilers

Compilers are a fundamental pillar of any software ecosystem. Without a compiler, none of the software can be ported to the target architecture. Luckily nowadays there are several `Arm` compilers available. The most popular one is the GNU Compiler Collection. However, `Arm` also offers the Arm HPC Compiler[8] which is built in Clang (front-end) and LLVM (back-end). The LLVM core libraries implements two of

[8] https://developer.arm.com/products/software-development-tools/hpc/arm-compiler-for-hpc

the three basic parts of a compiler: optimizer and back-end (machine code generation). The third and missing part is the front-end (typically Clang). This is responsible of the lexical analysis and the parsing. In other words, it takes the source code and generates an abstract syntax tree (LLVM IR) that is passed to the LLVM optimizer. The combination of a front-end plus LLVM let us replace the full GCC stack and in the `Arm` case, they tuned the compiler for the most popular `Arm` platforms.



Compilers
GNU JDK Mercurium

Scientific libraries
ATLAS LAPACK SCALAPACK FFTW
BOOST clBLAS clFFT PETSc HDF5

Performance analysis
EXTRAE Paraver Scalasca

Debugger
Alinea DDT

Runtime libraries
Nanos++ OpenCL OpenMPI MPICH3

Cluster management
SLURM Nagios Ganglia

Hardware support
Power monitor

Storage
LustreFS

Operating System
Ubuntu

Figure 2.1: Mont-Blanc prototype software stack

2.1.2 Scientific libraries

When we talk about libraries, we tend to think about a place where there are a lot of books, however, in computer science, a library consists in a collection of non-volatile resources that any program can use. In HPC, when we talk about scientific libraries, we refer to a collection of routines implementing mathematical operations in a very efficient way. The most common one is Basic Linear Algebra Subprograms (BLAS) which defines vector and matrix operations.

BLAS itself is a specification rather than an implementation. The BLAS specification defines three levels of complexity: level one are vector operations, level two are matrix-vector operations and level three are matrix-matrix operations. There are closed implementations of BLAS, *e.g.* the Intel Math Kernel Library (MKL), which are optimized specifically for Intel processors, while the two most popular open-source implementations are OpenBLAS and ATLAS. `Arm` provides the Arm Performance Libraries, an implementation of BLAS specifically tuned for server-class `Arm` based platforms (closed source). However, both open-source implementations, OpenBLAS and ATLAS are available too. Using these libraries have a non-negligible performance impact and the actual effects are discussed in Chapter 5. I also took part in a wider study of the impact of using Arm Performance Libraries on several HPC codes. This work has been published in a poster presented at the Supercomputing Conference 2017 [1].

2.1.3 Parallel Programming models

In computing, a programming model refers to an abstraction of the parallel computer architecture which usually take the form of a library invoked as an extension of an existing language. In HPC the Parallel Programming Models are very important as they are key to use all the resources (in parallel) of the machines. Modern parallel programming models also support parallel heterogeneous executions. In Section 5.3 we will discuss and show the impact of using the OmpSs parallel programming model to exploit the parallelism in devices featuring an accelerator. The programming models usually take care of the interaction among parallel processes and the problem decomposition.

Process interaction defines how the parallel processes interacts/communicates among them. Since process interaction usually leverages read/write into memory locations, the two main kinds of process interactions are:

- **Shared memory:** Processes share a global address space that can be read and written asynchronously.
- **Distributed memory:** Processes do not share the same address space and requires a message-passing mechanism to exchange data between the other processes.

In `Arm`, there are several shared memory programming models available. The two most popular ones are OpenMP and OmpSs. In the distributed memory category, popular implementations of MPI (Message Passing Interface) like OpenMPI or MPICH are available too and are used to run applications in a multi-node environment.

Problem decomposition explains how the workload is divided among the parallel processes. The decomposition can be classified in:

- **Domain decomposition:** In this type of partitioning, the data of the problem is decomposed such that each parallel process works on a portion of the data.
- **Functional decomposition:** In this type of partitioning, the problem is decomposed from the point of view of the work that must be done. In this case, the processes perform a portion of the total work.

These two approaches of problem decomposition are not orthogonal and they are typically combined in most of the parallel applications.

2.1.4 Platforms

The current number of platforms using `Arm` processors has increased a lot. Almost all the smartphones sold today use an `Arm` based processor and the embedded world is basically dominated by `Arm` too. Platforms like the Jetson TX1 from Nvidia packs 4 `Arm` cores plus a GPU based on the Nvidia Maxwell architecture. This combination gives enough compute capacity to provide 1 TFlop/s using 16bit floating point data. A comercial device that uses the same processor as the Jetson TX1 is the popular handheld Nintendo Switch gaming console. Another popular platform is the Xilinx Zynq 7000 family. This platform packs 2 `Arm` cores plus a user programmable logic (FPGA). This type of devices are very popular in the Industry 4.0. For example, Aingura IIoT uses the Xilinx Zynq 7020 platform in their numerical control machines for gathering information from the sensors. They use the programmable unit to develop a custom solution adapted for each machine. However, the `Arm` presence goes beyond the embedded market. Amazon recently launched the Graviton processor. This processor is custom designed by Amazon and it is based on `Arm`v8. Amazon is currently offering AWS (Amazon Web Services) nodes that uses these chips [31]. Huawei has also recently launched a server-class Arm-based processor called Kunpeng 920 [32]. Huawei claims that it is the industry's highest-performance Arm-based processor, delivering a 25% more perfomance than the competition, but by the time of this thesis no official performance numbers have been revealed.

Another important milestone for the `Arm` architecture is its presence in supercomputers. Table 2.1 shows the current most important supercomputers running `Arm` processors. The Mont-Blanc prototype is the world's first Arm-based HPC cluster and first using only embedded processors. This prototype demonstrated that using Arm technology in HPC was doable. The rest of the supercomputers are using server-class `Arm` processors, being the ThunderX2 processor the clear dominant. Recently, the HPE Astra entered into the TOP500 list. This is the very first time that an Arm-based system appears on the TOP500. The HPE Astra, which uses the ThunderX2 processor, achieved 1529 TFlop/s in the Linpack test and was ranked in the position 204 of the list[9].

[9] https://www.top500.org/system/179565

Name	Location	Processor	State
HPE Astra	Sandia Labs (USA)	ThunderX2	In production
Fujitsu Post-K	RIKEN (Japan)	ARM64FX	Expected in 2021
Dibona	Bull Atos (France)	ThunderX2	In production
Isambard	GW4 (UK)	ThunderX2	In production
Mont-Blanc prototype	BSC (Spain)	Samsung Exynos 5 Dual	In production
<unnamed>	French Atomic Energy Commission (CEA)	ThunderX2	Expected in 2019

Table 2.1: List of the current available Arm supercomputers.

2.2 RISC-V

RISC-V started back in 2010 in the University of California (UC) Berkeley. At first UC Berkeley was looking on the nowadays currently available ISAs to use it in their next set of projects. At that time, the two immediate choices were x86 and Arm [33]. However, these two easily became unrealistic options for their purpose. The first reason is because both ISAs were extremely complex for their initial goal and second, both ISAs are protected under Intelectual Property laws. As UC Berkeley already had past experience in designing RISC ISAs, they decided to start a 3 months project of designing their own simple and open ISA. Four years later, what started as a small inhouse project ended up gaining a lot of interest from the community and UC Berkeley decided to release the first frozen base user-spec of their ISA under the name of RISC-V. UC Berkeley previous RISC projects were: RISC-I[34] (1981), RISC-II[35] (1983), SOAR[36] (1985) and SPUR[37] (1986). RISC-V is the fifth UC Berkeley RISC project, hence the V (number 5 in roman numerals).

In 2015, the RISC-V Foundation was established. The Foundation comprises more than 150 member organizations working together to build the first open, collaborative community of software and hardware around RISC-V. With the promise of being a non-profit corporation controlled by its members, the foundation is in charge of the development of the RISC-V specification and pushes forward the adoption of the RISC-V ISA.

Due to the already present shift toward domain specific architectures, SoCs are becoming more popular. These systems pack together really different units targeting a very specific function in mind. But the problem is that each of them talk a unique and different ISA (typically propietary) due to how the units were designed. In most of the cases, these are not designed from scratch specifically for our SoC, instead, they use other existing units as a starting point. This results in the unnecessary pain of having to deal with many different ISAs for the correct functioning of the system. RISC-V borns with the idea of bringing an open and free ISA that everyone could use for everything. The main characteristics of RISC-V are:

- **Simple:** RISC-V takes advantatge of being new. It does not take any other legacy ISA as base, therefore it can implement only the useful instructions resulting in a much more compact ISA compared to other commercial ISAs.
- **Clean design:** After decades of research of what has gone well (and what has not), the RISC-V specification avoids many bad decisions included in older legacy ISAs and boosts the ones that gone well. For example, in RISC-V there is a clear separation between the user and the privileged ISA and it does not expose microarchitecture or technology dependent features.
- **Modularity:** In RISC-V, you are not forced to implement the whole ISA. Only the Integer extension is mandatory while the rest of the ISA is optional (extensions). This approach helps to mitigate the issue of dealing with an enormous ISA, as only the extensions that matter to you are the ones you end up implementing.
- **Specialization:** Part of the benefit of having modules is the possibility of building a RISC-V processor tailored to your needs. The RISC-V specification not only defines some extensions targeting very specific environments but it also reserves space for user-defined extensions. So for example if you need to add custom instructions to interact with an accelerator, the ISA has support for it.

The RISC-V modularity approach is to focus on stability and compatibility rather than functionality. For this reason, once an extension is frozen, it will remain as is forever. If for some reason, the extension has to be modified, they way to go is to define a new extension.

2.2.1 Software Ecosystem

The current software status in RISC-V can be found on their official website[10], however, in this chapter we are focusing only on the software related to run scientific applications in HPC environments.

Considering how young is the RISC-V architecture, is not difficult to guess that the scientific software stack is not ready yet. However, for the small amount of time it has been around, the software ecosystem maturity is quite impressive. To Arm, it took many more years to reach this point of maturity and part of the reason why nowadays there are server-class computers based on Arm is due to the joint effort in the Mont-Blanc project. Following the same steps as in Mont-Blanc, the European Processor Initiative (EPI) project aims to push forward the HPC ecosystem in RISC-V by developing an accelerator based on RISC-V, containing only European and Open-Source technology.

In HPC, having a Linux Operating System is a hard requirement. Lukily, the Linux kernel has already been ported to RISC-V[11] and there are already efforts in porting popular Linux distributions like openSUSE, Fedora or Debian to RISC-V. Another popular kernel in RISC-V is the Proxy Kernel (PK), a lightweight application execution environment that supports statically-linked RISC-V ELF binaries. PK targets tethered environments[12] therefore it does not fit in the HPC environment, however, sometimes it is quite useful to test applications in a isolated environment.

Compilers

Regarding the compilers, GCC has already been ported to RISC-V. The supported languages are C, C++ and Fortran, and also supports the OpenMP parallel programming model. The GCC port for RISC-V has two versions: GCC Unknown Elf, which is a version targeting the execution of application in a Proxy Kernel environment (bare-metal), and the other one is GCC Linux GNU, which targets the Linux execution environment. Another well-known compiler toolchain available in RISC-V is Clang+LLVM. However it is still under development and the current version is not part of an official release yet. It is important to notice that the current compilers (specially LLVM) are focusing more in the correctness rather than tuning the generated code for specific platforms.

Libraries

The scientific libraries are a fundamental part of the software stack in a HPC system. However, in RISC-V there are many of these libraries missing. Althought OpenMP is supported, there is no MPI implementation ready, meaning that distributed memory parallel computing is not supported yet.

Scientific libraries like BLAS are also missing. These libraries typically make heavy use of the data parallelism paradigm (SIMD), and because the Vector extension is not frozen yet, our feeling is that these libraries will continue to be missing until the Vector extension becomes stable.

2.2.2 Platforms

The official RISC-V website also has a compilation of the current available Open Source RISC-V cores[13]. Most of the cores are very simple 32-bit single core configurations and not all of them are capable of booting Linux. For the time of this thesis, the only ones that boot Linux are:

[10] https://riscv.org/software-status

[11] https://github.com/riscv/riscv-linux

[12] In a tethered environment, the RISC-V core handles the I/O-related systems calls by proxying them to a host computer.

[13] https://riscv.org/risc-v-cores

- **Rocketchip**: 64-bit core developed by SiFive and supporting all the frozen extensions. Currently, has already been tapped out in a quadcore configuration and can be found on the HiFive Unleashed development kit.

- **BOOM**: 64-bit core supporting multicore configurations, out of order execution and all the frozen extensions in the specification. The core has been developed by Esperanto and UCB-BAR.

- **Ariane**: 64-bit core developed in ETH Zurich in collaboration with the Università di Bologna. Currently only supports single core configuration, but multicore support is comming soon.

- **SHAKTI C-CLASS**: 64-bit core developed by IIT Madras. It only supports single core configuration.

Chapter 3

RISC-V for HPC

Due to RISC-V's short-life, the HPC capability is not quite ready, some of the reasons have been discussed in Section 2.2, however, this does not mean that it is impossible to execute HPC applications in the current RISC-V state.

As RISC-V is open, there are several Open-Source implementations available online and one could potentially choose one and build new things on top of it for research purposes. In this section, we are going to explain step-by-step the process of picking an open-source RISC-V core and the initial steps that have to be done in order to run an HPC workload in a RISC-V core that it is running on an FPGA.

3.1 Processor selection

The modularity of the RISC-V ISA is a double-edged sword. For example, the success of the Intel processors came due to the backward compability in newer generations, this is that newer processors have to implement all the instructions that have been defined since the definition of x86, even if they make no sense in the current days. In RISC-V, if some extension makes no sense anymore, you can drop it and only implement the really necessary ones in favour of reducing the chip area. On the other hand, having extensions will give way to fragmentation, as not all the processors are going to be equal (even if the base is RISC-V) and not all the binaries will run in every machine.

The very first requirement before choosing a processor is to delimit the scope. In this case we are analyzing what type of instructions are present in the Linux kernel. The reason why we are choosing Linux is because HPC environments uses Linux as the main Operative System. This analysis is also known as static instruction mix. The static instruction mix let us identify which instructions are present, therefore which RISC-V extensions must be supported by our processor in order to execute that code.

Figure 3.1 shows the static instruction mix of the RISC-V Linux Kernel. Analyzing the pie chart, we can conclude that most of the instructions are Integer (I), Compressed (C) and few ones are Integer multiplication/division (M) and Atomic (A). Even if the presence of these last two is almost negligible, the processor must support it, otherwise, an ilegal instruction exception will be raised when these instructions are executed.

Considering these extensions, we decided to pick Ariane. Ariane is a 6-stage, single issue, in-order CPU implementing the 64-bit RISC-V ISA[14]. It fully implements the I, M, C and A[15] extensions, therefore we can execute all the instructions present in the Linux Kernel. The other considered option was Rocketchip. Rocketchip is also 64-bit, however, in Rocketchip the hardware generation is done using Chisel. Chisel, as a hardware language is too abstract, in the sense that the developer does not have a 100% control of the generated hardware, as you only specify parameters (*e.g.* number of cores) and the compiler does the job. In comparison, Ariane is built entirely using SystemVerilog, the new industry standard Hardware Description Language fully supporting the UVM (Universal Verification Methodology) verification standard.

[14] The instructions in RISC-V are always 32-bit length. The 64-bit ISA forces the data width to be 64 bits, therefore the general purpose registers are 64 bits width.

[15] For the date of this thesis, the atomics support in Ariane is only valid in single core configurations.

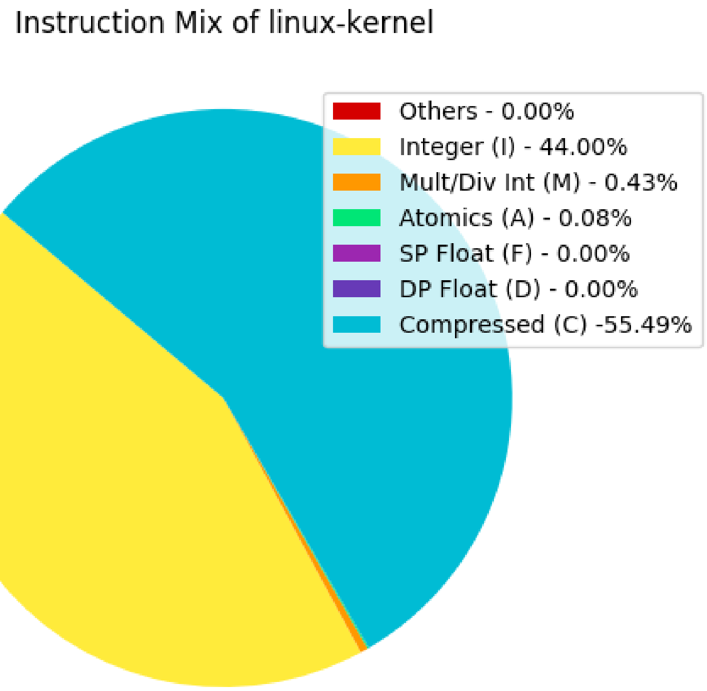


Figure 3.1: Static Instruction mix of a Linux Kernel

Base	Version	Frozen?
RV32I	2.0	Yes
RV32E	1.9	No
RV64I	2.0	Yes
RV128I	1.7	No
Extension	Version	Frozen?
M	2.0	Yes
A	2.0	Yes
F	2.0	Yes
D	2.0	Yes
Q	2.0	Yes
L	0.0	No
C	2.0	Yes
B	0.0	No
J	0.0	No
T	0.0	No
P	0.1	No
V	0.2	No
N	1.1	No

Table 3.1: `RISC-V` extensions status (ISA version 2.2).

3.2 Bare-metal UART

A core by itself is useless and modern processors pack much more than only cores. As first step in order to be able to test some code we designed an SoC with only the essential units to accomplish this goal. These units are the core, the memory and a UART unit to communicate with the outside.

Figure 3.2 shows a block diagram of the minimal SoC we used as first approach. The figure shows in green the AXI4 connections, which goes from Ariane to the other units. We can also see the `uart2debug` unit, which will take care of the communication between the SoC and external devices using the UART (Universal Asynchronous Receiver-Transmitter) protocol. All these units will be explained in more detail in the following subsections.

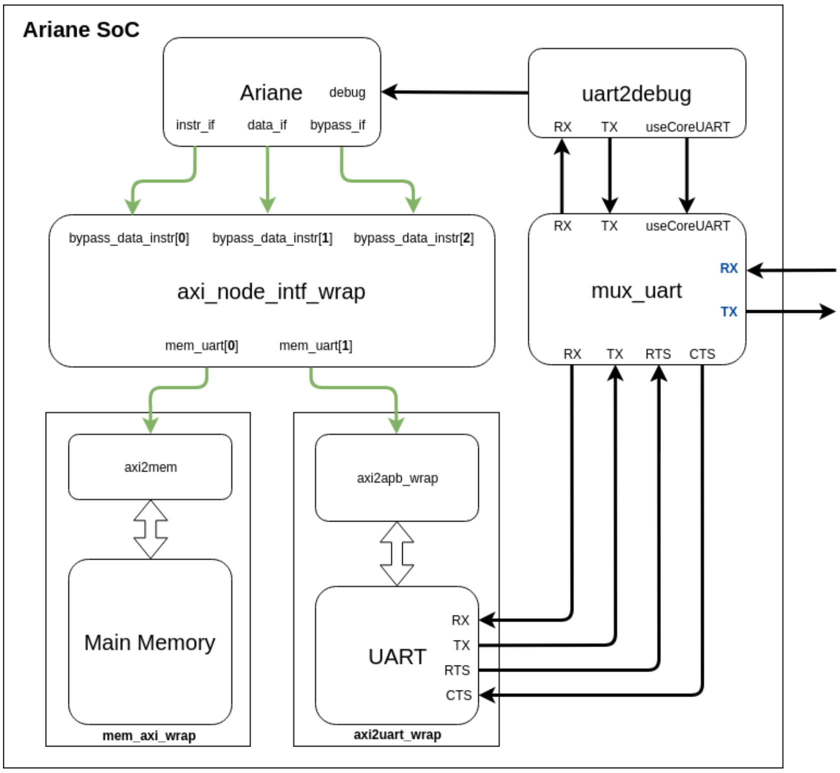


Figure 3.2: Ariane's minimal SoC block diagram. In green the AXI4 connections.

3.2.1 AXI4 on Ariane

The first step is to learn how Ariane expects to interact with other units. In this case, Ariane bases its external communication using the AXI4 protocol. AXI stands for Advanced eXtensible Interface and is part of the `Arm` Advanced Microcontroller Bus Architecture (AMBA) specification.[16] One particular thing that simplifies a lot the communication in AXI4 is the fact that interfaces are memory mapped. So if a unit A wants to communicate with another unit B, the only thing that it needs to know is in which memory address range unit B is mapped on, so that sending the data to an address in that range will arrive to that unit. This approach makes the communication to be more homogeneous, as every unit only works around a defined memory map.

In Arine's case, they use an Open Source AXI4 implementation developed by their team. This includes the `AXI_BUS` interface and the `axi_node_intf_wrap` module, being this last one an AXI Crossbar interconnection unit. Figure 3.3 shows how the AXI4 crossbar is instantiated. Note that we are defining the address range of each destination port with the `start_addr_soc` and `end_addr_soc` variables. These variables are later passed to the `start_addr_i` and `end_addr_i` ports from the crossbar unit.

[16] More information in: Introduction to AXI Protocol: Understanding the AXI interface

```
localparam logic [63:0] MEM_START         = 64'h0000_0000;
localparam logic [63:0] MEM_END           = 64'h8100_0000;

localparam logic [63:0] PERIPHERALS_START = 64'h1000_0000;
localparam logic [63:0] PERIPHERALS_END   = 64'h1000_1000;

localparam logic [1:0][63:0] start_addr_soc = { PERIPHERALS_START, MEM_START };
localparam logic [1:0][63:0] end_addr_soc   = { PERIPHERALS_END, MEM_END };

axi_node_intf_wrap #(
    ...
) i_axi_node (
    ...
    .start_addr_i   ( start_addr_soc    ),
    .end_addr_i     ( end_addr_soc      ),
    ...
);
```

Figure 3.3: Ariane's AXI4 Crossbar module instantiation.

In Figure 3.3 we defined two destination ports, one for the main memory, which is mapped from the address 0x00000000 to 0x81000000 and another one for the UART, which is mapped from address 0x1000000 to 0x10001000. This means that for example, any transaction with a target address in the {0x00000000, 0x81000000} interval will be received by the memory, and same applies for the UART unit.

3.2.2 uart2debug module

As seen in Figure 3.2, there is a unit called uart2debug. This unit has been designed from scratch in order to handle the communication between the SoC and external devices. On one hand, it implements the UART protocol, with which the SoC will talk with an external device (*e.g.* a computer), and on the other hand, it is connected to the core. Note that it is connected to the mux_uart unit because the UART port in the FPGA is shared between the debug unit and the actual output serial device of the core (axi2uart_wrap unit).

For the communication with external devices, we have defined a simple custom protocol. This protocol is based on two type of transactions, the ones that controls the SoC state (*e.g.* halt, resume) and the ones for reading/writing data from/to the SoC. Figure 3.4 shows the Finite State Machine for reading and writing data. The main bottleneck in this type of transactions is the UART bandwidth. As the communication granularity is 1 byte, we have to repeat the WAIT->SEND and the RECV->RECV transitions 8 times.

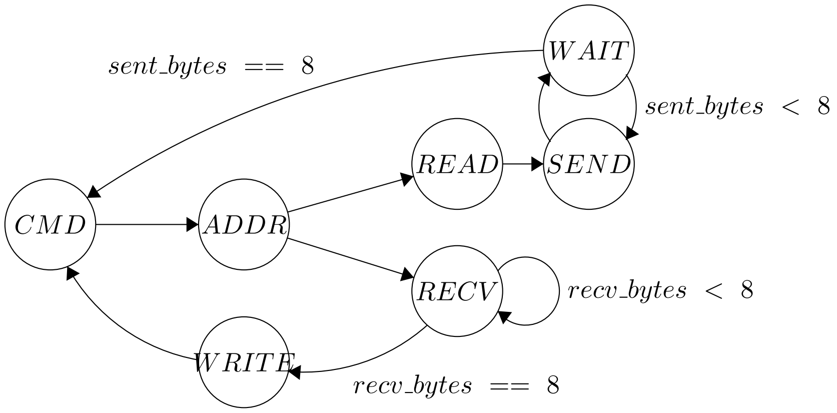


Figure 3.4: Finite State Machine of uart2debug's R/W protocol.

For the SoC control, we defined 5 different commands. The Finite State Machine of each of these commands can be seen in Figure 3.5. These commands changes the SoC state in the following way:

- **do_halt:** This command will automatically halt the processor. In order to avoid inconsistent states, the processor will not be halted until the whole pipeline has been flushed.
- **do_resume:** This command will resume the processor if it was halted, otherwise, nothing will change.
- **do_fetch_en:** This command will enable the fetch of new instructions in the processor. Typically, this is used at the beginning so that we can write data to the memory when the processor starts (the processor boots in a resume state and with the fetch disabled).
- **use_core_uart:** This command switches the multiplexer (`mux_uart`) in order to connect the UART port to the output serial device of the SoC (and disconnects it from the `uart2debug` unit).
- **set_PC:** This command changes the current PC of the core to the address received through the UART.

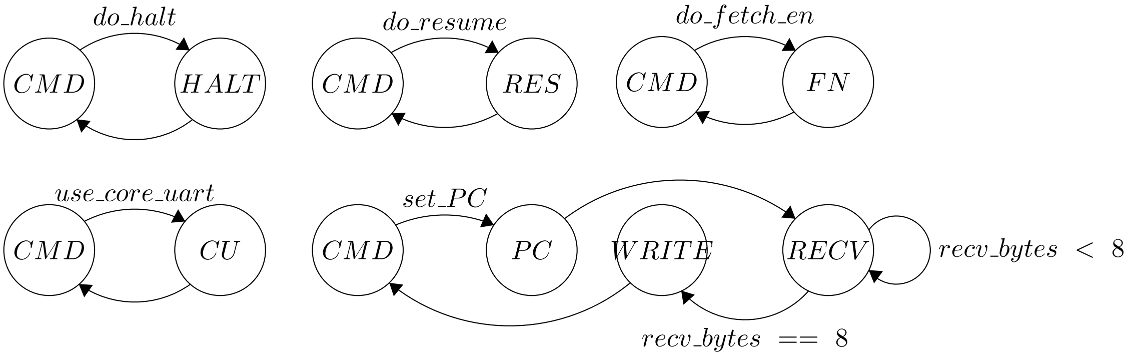


Figure 3.5: Finite State Machine of `uart2debug`'s SoC control protocol.

3.2.3 External interaction

Now that we have an SoC with a dedicated unit to handle the communication with external devices, it is time to communicate from a laptop's shell (`minicom`) to the SoC running on the FPGA. To do so, we have implemented a Python class that adds a level of abstraction and let us interact with the SoC in an easier manner.

The Python class has a method to read a binary (compiled with `gcc` for example), and then load it into processor's memory at a user-defined address. Similarly, there are methods to read or modify the registers and the memory. To test that we can successfully interact with the core, we wrote the following C code:

```c
int add(int a, int b) {
    return a + b;
}

void _start(void) {
  int x = add(1, 2);
  while (1) {
    __asm__ __volatile__("" : : "r"(x));
  }
}
```

Figure 3.6: Mini example written in C for testing Ariane's functionality in an FPGA.

And then using the following Python code, we will write the instructions into Ariane’s memory and make it execute them:

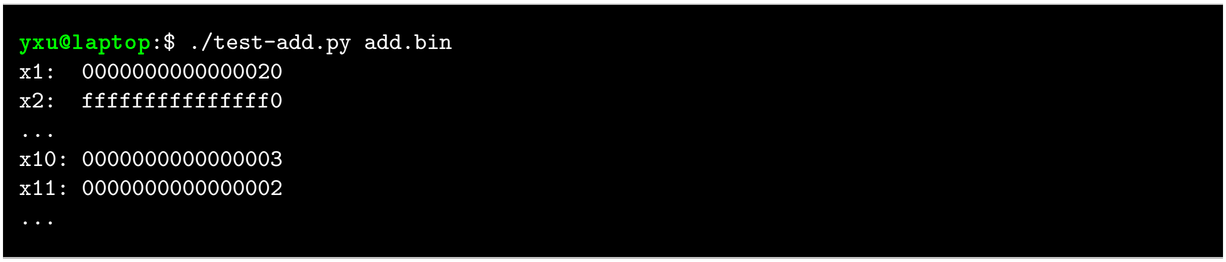
```
#Connect to the serial unit
du = DebugUnit("/dev/ttyUSB0")

#Halt the processor
du.doHalt()

#Set the General Purpose registers to a known state
for i in range(0, 32) :
    du.writeRegister(format(i, ’x’), format(0, ’x’))


#Load binary to Ariane’s memory
with open(tmpfile) as f:
    base_addr = 0
    lines = f.readlines()
    for inst in lines:
        a = du.writeMemory(format(base_addr, ’x’), inst)
        base_addr += 8

#Set PC
init_addr = 8
du.setPC(format(init_addr, ’x’))

du.doResume()
du.doFetchEn()
```

Figure 3.7: Python script to load the binary into the SoC’s memory.

To check if Ariane executed the instructions correctly, we read the registers also using the Python class:

```
...
du.doHalt()
du.readAllRegisters()
```

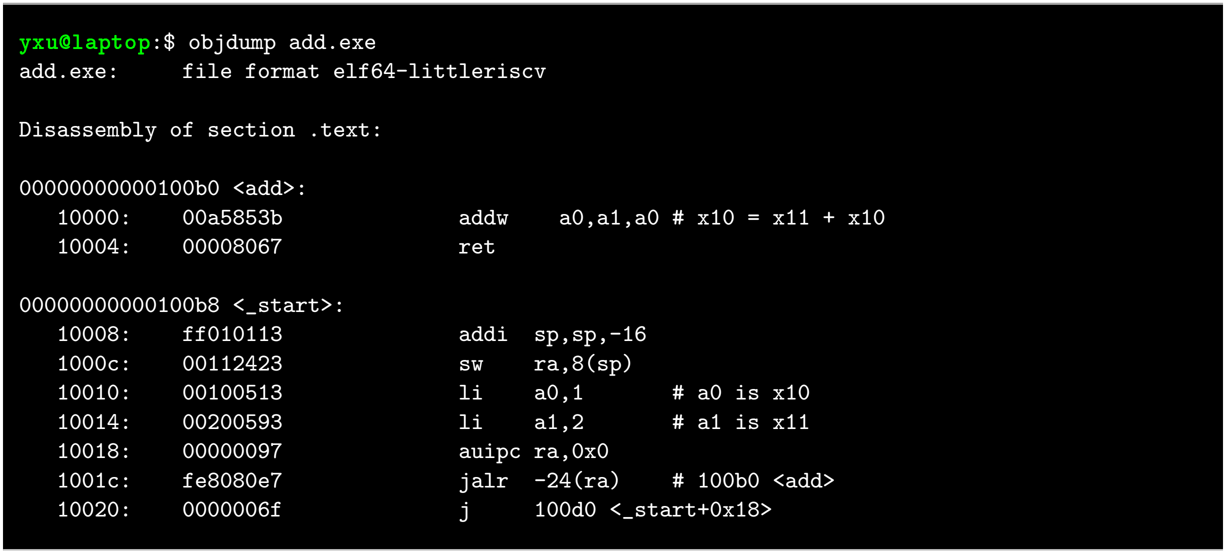
The `readAllRegisters()` method actually reads the 32 general purpose registers in Ariane and prints their value. In this case, the values reported by the script can be seen in Figure 3.8.



```
yxu@laptop:$ ./test-add.py add.bin
x1:  0000000000000020
x2:  fffffffffffffff0
...
x10: 0000000000000003
x11: 0000000000000002
...
```

Figure 3.8: Result of the add example running in Ariane on an FPGA.

In order to understand this output we have to first check the instructions that the `RISC-V` compiler has generated from our C code. Figure 3.9 shows the disassembly of the `add.exe` executable. The output shows the virtual registers (`aX`) however, if we manually check the instruction’s code with the `RISC-V` ISA specification, we can see that `a0` has been mapped to the physical register `x10` and `a1` has been mapped to the physical register `x11`. Considering this mapping, the expected value for register `x10` would be 3, which is in fact the reported value above. Another observation is that in `RISC-V`, the register `x1` stores the *return address* when we execute a jump (`jalr` in this case). In the above output, `x1`’s value was `0x0020`, which is in fact the address of the instruction after the jump. Similarly, `x2` is assigned to the *stack pointer* and as the initial value of that register was `0`, the instruction in address `0x10008` sets the value `-16` (`0xfff0` in two’s complement), which matches again with the reported value. Considering these checks, we can conclude that our minimal SoC running on an FPGA is working correctly.



```
yxu@laptop:$ objdump add.exe
add.exe:     file format elf64-littleriscv

Disassembly of section .text:

00000000000100b0 <add>:
   10000:    00a5853b              addw    a0,a1,a0 # x10 = x11 + x10
   10004:    00008067              ret

00000000000100b8 <_start>:
   10008:    ff010113              addi  sp,sp,-16
   1000c:    00112423              sw    ra,8(sp)
   10010:    00100513              li    a0,1       # a0 is x10
   10014:    00200593              li    a1,2       # a1 is x11
   10018:    00000097              auipc ra,0x0
   1001c:    fe8080e7              jalr  -24(ra)    # 100b0 <add>
   10020:    0000006f              j     100d0 <_start+0x18>
```

Figure 3.9: Disassembly of the add example executable (compiled with GGC).

3.3 Bare-metal OpenOCD

In the previous section we defined a simple and effective way to interact with the Ariane core running on an FPGA. Even if the `uart2debug` unit is simple and capable enough for running programs, it has some limitations. For example, our unit is not designed to actually debug low level software and hardware and does not offer user-friendly ways to achieve that, instead, we can execute all or none of the instructions, but not an intermediate term. Luckily, in `RISC-V` exists the debug specification that although is still a draft for the time of this thesis, the industry is starting to move towards this direction (including Ariane).

3.3.1 RISC-V debug specification

In this subsection we are going to introduce briefly the debug specification in `RISC-V`. Currently, the latest version of the specification is `0.13`. The debug specification started out as an email list but due to the common need of having a standarized way to debug any `RISC-V` core, a working group started in August 2016. The main goal of the specification is to let any debugger connect blindly to any `RISC-V` platform and discover everything it needs to know.

Figure 3.10 shows a block diagram of the main components involved in the debug support. Blocks shown in dotted lines are optional and it is up to the designer to implement them, allowing different implementations for different use-cases.

The debug specification is designed to work in conjunction with a debugger (software). Currently the two most popular options are telnet[17] and GDB (GNU Project Debugger). The debugger talks with a Debug Translator, typically OpenOCD, and this one talks to the Debug Transport Hardware which talks with the Debug Transport Module (DTM) inside the SoC. Note that the Debug Translator may have to implement the drivers of the Debug Transport Hardware in order to communicate with him and that both Debug Transport Hardware and Debug Transport Module must communicate using the same protocol (*e.g.* JTAG).

The DTM communicates with the Debug Module Interface (DMI), who knows how to communicate with the Debug Modules (DM). A `RISC-V` core must implement at least one hart (Hardware Thread) and each hart can only be controlled by one DM. However, one DM can control multiple harts, which is the most common case. Also, the core must implement a ”Debug” execution mode where the core waits for instructions from the debugger, the interrupts are disabled and the exceptions are handled by the debugger.

[17] Telnet is a protocol used on the Internet or local area network to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection. - *Wikipedia*

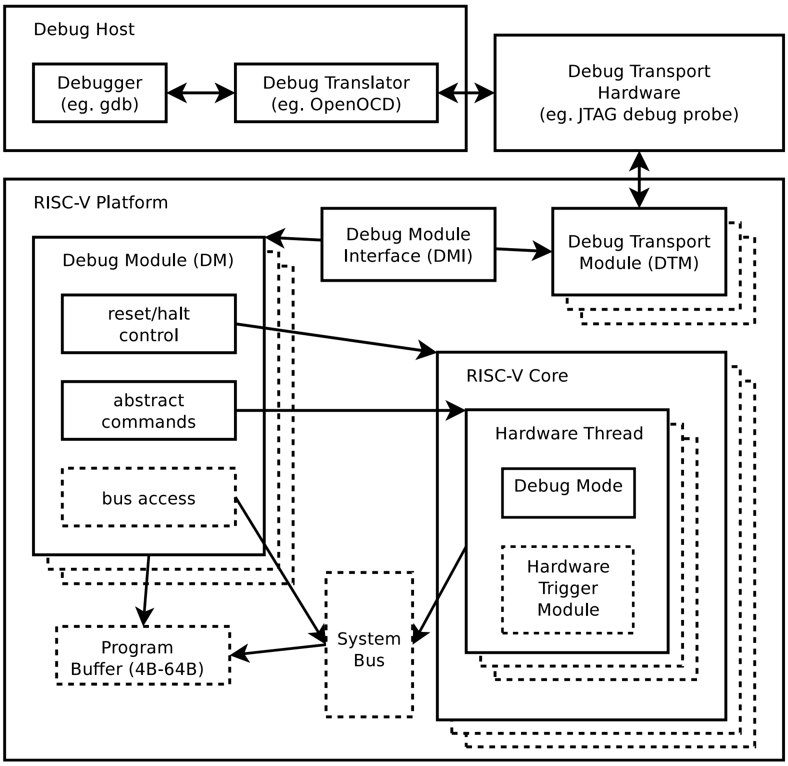


Figure 3.10: `RISC-V` Debug System Overview from Debug Specificacion version 0.13 draft.

The DM interacts with the core using Abstract commands and optionally a Program Buffer can be used too. The Abstract commands provide access to the core internal registers. These are the Control Status Registers (CSR), which includes the Program Counter (PC) for example, the General Purpose Registers (GPR), the Floating point registers, etc. On the other hand, the Program Buffer lets the debugger to write any instruction into the buffer and make the hart execute them. This is typically used to implement memory accesses.

As there are a lot of optional functionality, the debugger must discover which are supported by trial and error. In the case of the Abstract commands, the debugger writes into the internal `command` register, waits until `busy` (bit `12`) is not high and later checks the `cmderr` (bits `10:8`) to check if no errors happened. Both informations come from the `abstractcs` (Abstract Control Status) register.

3.3.2 Hardware setup

Ariane includes in the same repository both Debug Module (DM) and Debug Module Interface (DMI). The later also implements the Debug Transport Module (DTM) and uses the JTAG protocol to communicate with the Debug Transport Hardware.

```
module dmi_jtag (
    input  logic        clk_i,   // DMI Clock
    input  logic        rst_ni,  // Asynchronous reset active low
    ...
    input  logic        tck_i,   // JTAG test clock pad
    input  logic        tms_i,   // JTAG test mode select pad
    input  logic        trst_ni, // JTAG test reset pad
    input  logic        td_i,    // JTAG test data input pad
    output logic        td_o     // JTAG test data output pad
);
```

Figure 3.11: Ariane Debug Module Interface ports definition (`dmi_jtag.sv`).

Figure 3.11 shows some of the SystemVerilog ports definition of the DMI module. In this case, we want to focus on the JTAG ports (`tck_i`, `tms_i`, `trst_ni`, `td_i`, `td_o`). These ports are going to be directly connected to the Debug Transport Hardware. For this task, as we are working with an FPGA, having user-assignable pins is a hard requirement. For the board we have (Digilent Genesys 2), we can make use of the Pmod connectors. The Pmod interface is tipically found in comercial peripheral modules that extend the board features (*e.g.* SD card reader) but there is no restriction regarding the pin assignment of the Pmod ports. For this reason, we decided to assign the JTAG ports to the Pmod connectors.

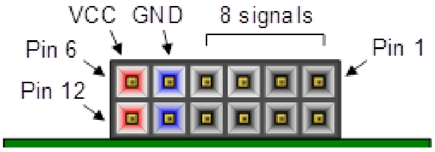


Figure 3.12: Pmod interface pin numbering (Genesys 2 reference manual).

On the Genesys 2 board we have 5 Pmod connectors. Table 3.2 shows the characteristic of each port. For Ariane's JTAG module, we chose the `JC` connector. The reason behind this decision instead of choosing the other ones is mainly due to the 200Ω protection resistor, which will avoid damages to the board in the event of mismatched connections.

Pmod conector	Power	Analog/Digital	Series protection	Use-case
`JXADC`	VADJ	Dual	100Ω	Analog inputs
`JA, JB`	3.3 V	Digital-only	0Ω	>=10MHz
`JC, JD`	3.3 V	Digital-only	200Ω	<10MHz

Table 3.2: Genesys 2 Pmod interfaces.

Finally, in the XDC (Xilinx Design Constraints) file is where we specify the mapping between the Pmod pins and the JTAG ports of the `dmi_jtag` module (Figure 3.13).

```
...
set_property -dict {PACKAGE_PIN AK29 IOSTANDARD LVCMOS33} [get_ports tck]
set_property -dict {PACKAGE_PIN AG30 IOSTANDARD LVCMOS33} [get_ports tdi]
set_property -dict {PACKAGE_PIN AJ27 IOSTANDARD LVCMOS33} [get_ports tdo]
set_property -dict {PACKAGE_PIN AK30 IOSTANDARD LVCMOS33} [get_ports tms]
set_property -dict {PACKAGE_PIN AH30 IOSTANDARD LVCMOS33} [get_ports trst_n]
...
```

Figure 3.13: Ariane Genesys 2 user defined constraints for the JTAG port (`genesys-2.xdc`).

The other part of the communication chain is the Debug Translator and the Debug Transport Hardware. For the first one, we chose OpenOCD which stands for Open On-Chip Debugger. OpenOCD supports many architectures (*e.g.* `Arm`, MIPS) and recently has been ported to `RISC-V`[18]. For the Debug Transport Hardware we chose the Olimex ARM-USB-OCD-H debug adapter for two reasons: first because it is natively supported by OpenOCD, and second because the voltage range is between 1.65 and 5.0 V (Pmod works at 3.3 V). However it is important to note that Olimex ARM-USB-OCD-H and the Pmod header on the Genesys 2 have different pin assignment schemes. Tables 3.3, 3.4 and 3.5 clarifies the pinout assignment we followed in both connectors[19].

Figure 3.14 shows how the real setup looks like with the Olimex debugger connected to the Genesys 2 board.

[18] `https://github.com/riscv/riscv-openocd`

[19] The Pmod pinout assignment we used is different to the pinout used in the official Ariane repository.

Signal Name	ARM-USB-OCD-H Pin Number	Cable Color	Genesys 2 Pmod Pin Number
VREF	1	Red	12
VREF	2	Brown	6
trst_n	3	Purple	3
tdi	5	Black	8
tms	7	Grey	9
tck	9	Green	4
tdo	13	White	2
GND	14	Orange	5
GND	16	Yellow	11

Table 3.3: Olimex to Pmod pins assignment.

	1:VREF(red)	2:VREF(brown)
	3:trst_n (purple)	4
	5:tdi (black)	6
	7:tms (grey)	8
NOTCH	9:tck (green)	10
NOTCH	11	12
	13:tdo (white)	14:GND(orange)
	15	16:GND(yellow)
	17	18
	19	20

Table 3.4: Olimex ARM-USB-OCD-H pin connections.

square pad	1	7
	2:tdo (white)	8:tdi (black)
	3:trst_n(purple)	9:tms (grey)
	4:tck(green)	10
	5:GND(orange)	11:GND (yellow)
	6:VREF(brown)	12:VREF (red)

Table 3.5: Genesys 2 JC Pmod pin connections.

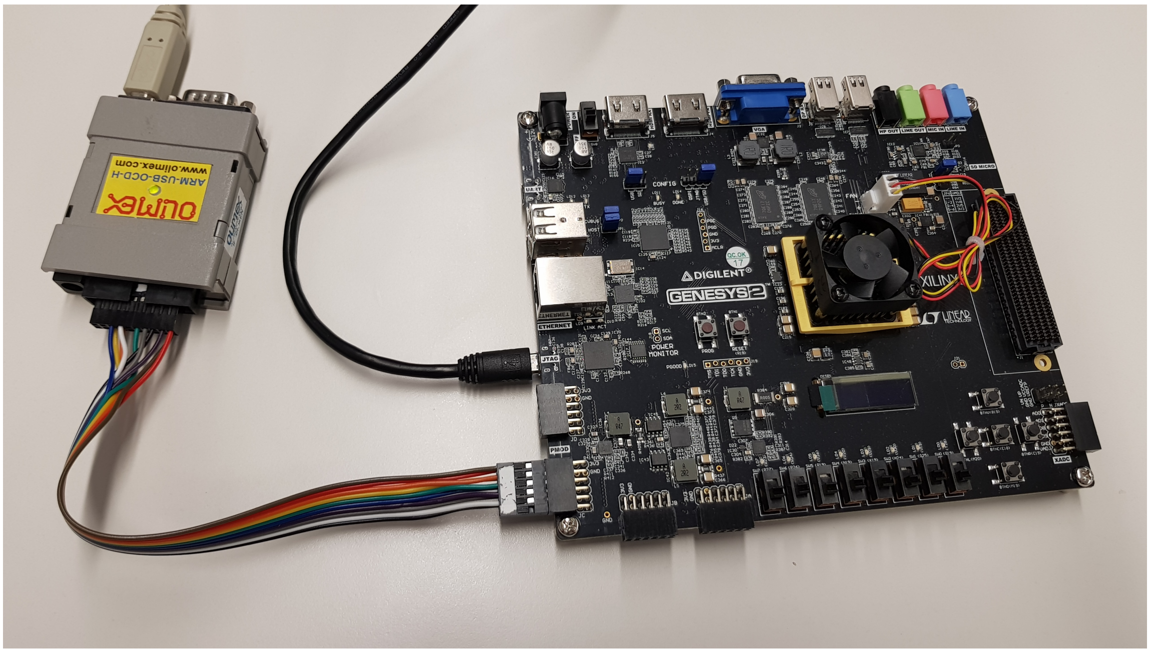
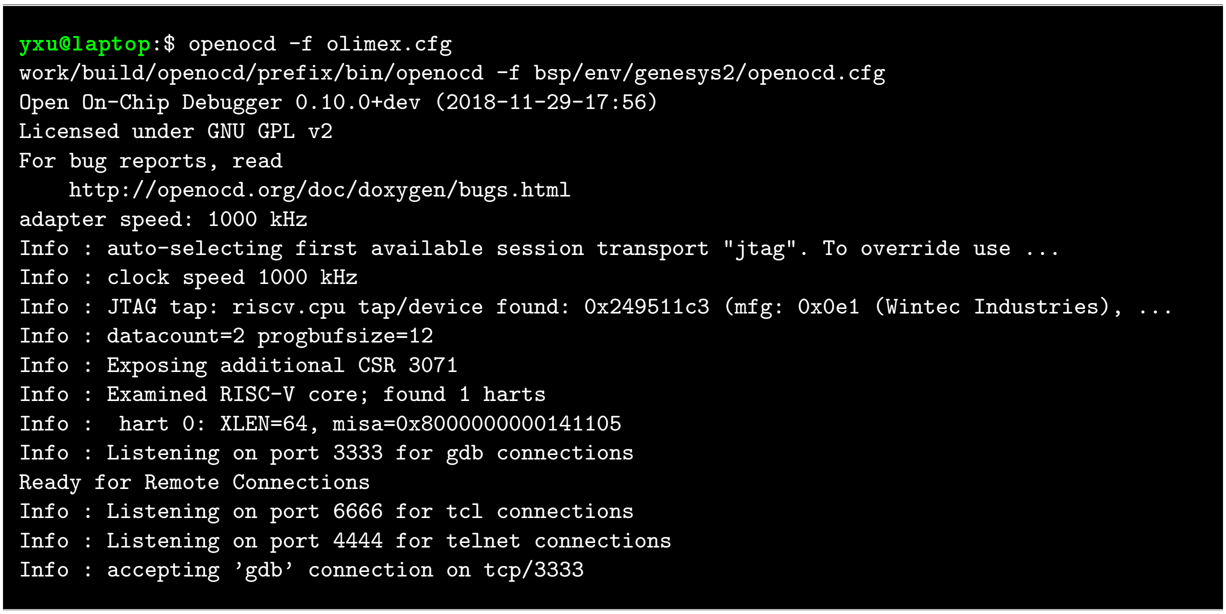


Figure 3.14: Photo of the Genesys 2 board connected to the Olimex ARM-USB-OCD-H debugger.

3.3.3 External Interaction

Similarly as in the previous case, we are interacting with the SoC using a shell in a laptop. Now, instead of using a custom made Python script, we are using GDB combined with OpenOCD. The first step is to connect the Olimex ARM-USB-OCD-H to the computer. After that, Linux will recognize it as a /dev/ttyUSB* device.

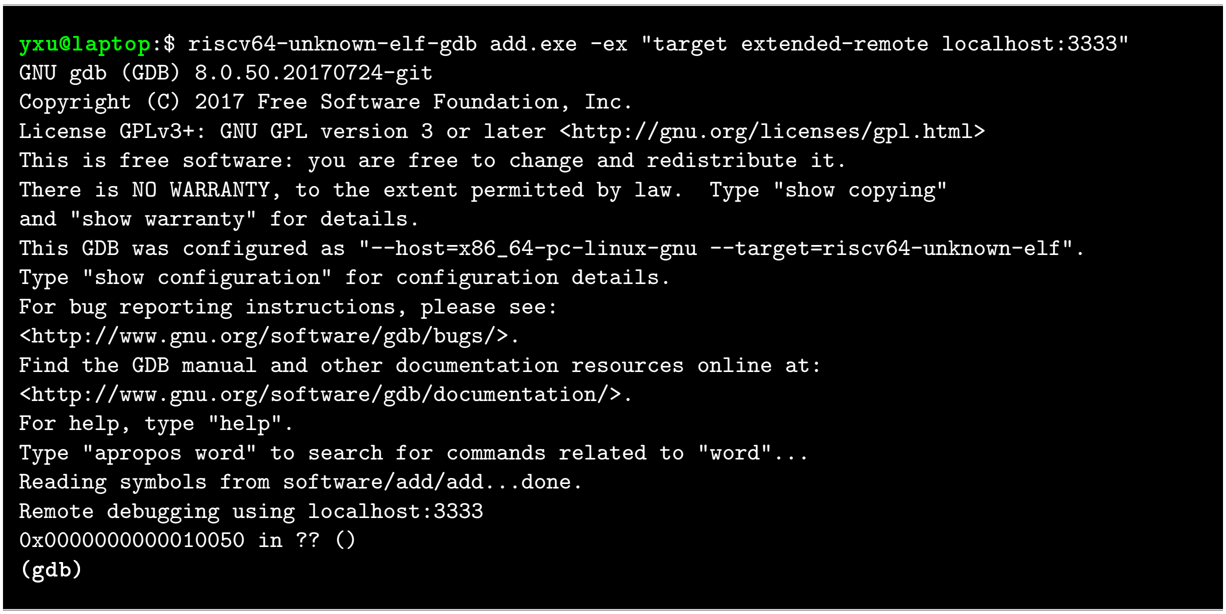
At this point, we can run OpenOCD, which will scan all the devices and connect to Olimex ARM-USB-OCD-H. Figure 3.15 shows the command and the output once we successfully connect to the Olimex device. The `olimex.cfg` file is the same file found in OpenOCD's installation directory but slightly modified for Ariane. This file can be found in the Apendix A.



```
yxu@laptop:$ openocd -f olimex.cfg
work/build/openocd/prefix/bin/openocd -f bsp/env/genesys2/openocd.cfg
Open On-Chip Debugger 0.10.0+dev (2018-11-29-17:56)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
adapter speed: 1000 kHz
Info : auto-selecting first available session transport "jtag". To override use ...
Info : clock speed 1000 kHz
Info : JTAG tap: riscv.cpu tap/device found: 0x249511c3 (mfg: 0x0e1 (Wintec Industries), ...
Info : datacount=2 progbufsize=12
Info : Exposing additional CSR 3071
Info : Examined RISC-V core; found 1 harts
Info :  hart 0: XLEN=64, misa=0x8000000000141105
Info : Listening on port 3333 for gdb connections
Ready for Remote Connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : accepting 'gdb' connection on tcp/3333
```

Figure 3.15: OpenOCD connecting to the Olimex ARM-USB-OCD-H.

After starting OpenOCD, it will be listening for GDB, TCL or telnet communications. In this case, we are using GDB. GDB let us execute a binary in a remote session and have a full control of the execution. Some of the GDB features are to read/write any register or memory position, load a binary into the memory or add breakpoints into the execution. Figure 3.16 shows the commands and arguments we used to load Figure 3.6's code.

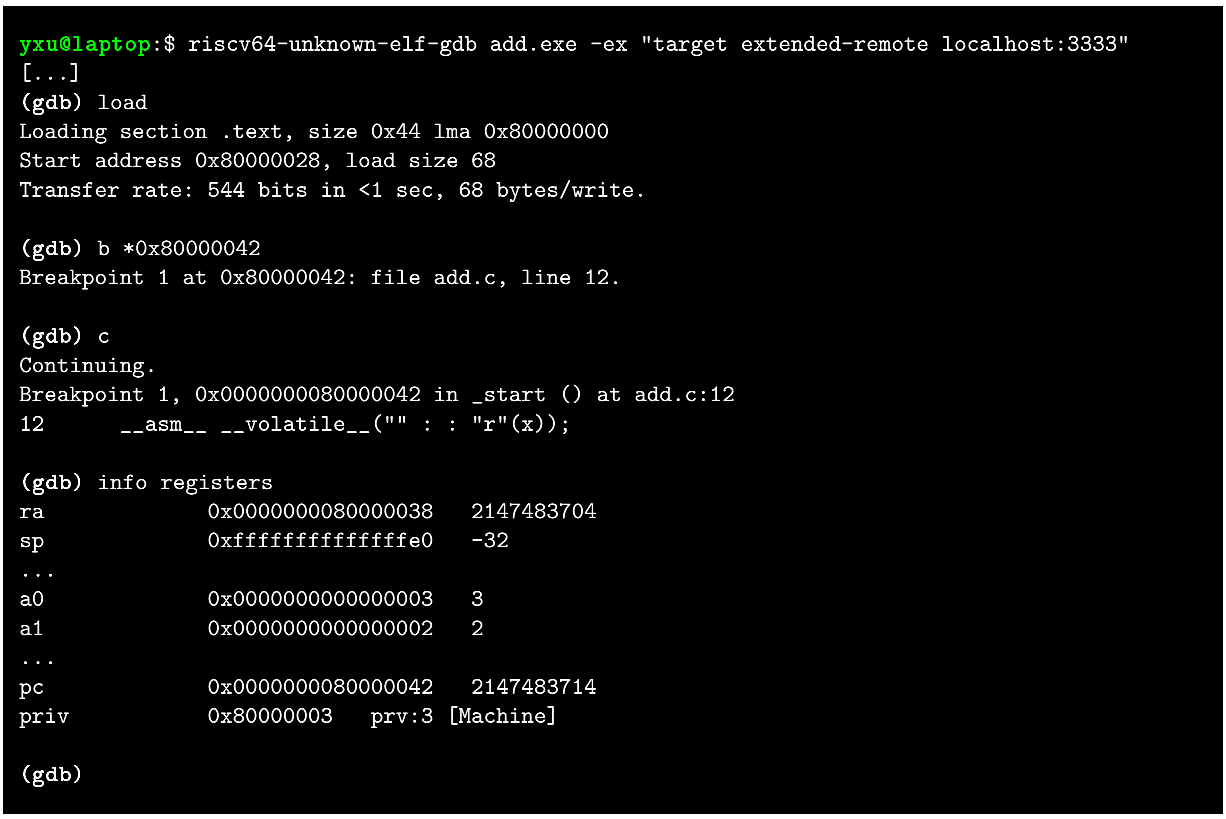


```
yxu@laptop:$ riscv64-unknown-elf-gdb add.exe -ex "target extended-remote localhost:3333"
GNU gdb (GDB) 8.0.50.20170724-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from software/add/add...done.
Remote debugging using localhost:3333
0x0000000000010050 in ?? ()
(gdb)
```

Figure 3.16: GDB session connected remotely to Ariane using OpenOCD.

GDB offers a lot of commands to the user. The most relevant ones for this experiment are the following:

- **load:** This command loads the binary into the memory. The file must be in Executable and Linkable Format (ELF) format as the address where the instructions are going to be loaded are read from the ELF header.
- **x:** This command is used to read an address value. For example, for reading the memory position `0x80000000`, the command would be: `(gdb) x /32x 0x80000000`.
- **info *argument*:** This command shows information of the *argument* you pass. For example if you want to check the integer registers and their content, the command would be: `(gdb) info registers`.
- **stepi:** This command steps into the next machine instruction. If the instruction is a function call, the `stepi` command steps into the function being called.
- **b:** This command sets a breakpoint into the code so that the execution stops at the address specified by the breakpoint. For example if you want to stop before the instruction at address `0x80000b78` is executed, the command would be: `(gdb) b *0x80000b78`.
- **c:** This command is the abreviation of `continue`. Executing this command will continue the execution until the next breakpoint is found.
- **p:** This command prints the value of a symbol. For example if you want to check the current Program Counter value, the command would be: `(gdb) p $pc`.



```
yxu@laptop:$ riscv64-unknown-elf-gdb add.exe -ex "target extended-remote localhost:3333"
[...]
(gdb) load
Loading section .text, size 0x44 lma 0x80000000
Start address 0x80000028, load size 68
Transfer rate: 544 bits in <1 sec, 68 bytes/write.

(gdb) b *0x80000042
Breakpoint 1 at 0x80000042: file add.c, line 12.

(gdb) c
Continuing.
Breakpoint 1, 0x0000000080000042 in _start () at add.c:12
12      __asm__ __volatile__("" : : "r"(x));

(gdb) info registers
ra             0x0000000080000038   2147483704
sp             0xffffffffffffffe0   -32
...
a0             0x0000000000000003   3
a1             0x0000000000000002   2
...
pc             0x0000000080000042   2147483714
priv           0x80000003   prv:3 [Machine]

(gdb)
```

Figure 3.17: Loading and executing `add.exe` in GDB.

Figure 3.17 shows the commands we used in order to load the `add.exe` executable (Figure 3.6) into Ariane's memory using GDB plus OpenOCD, and how using GDB commands we can drive the execution remotely. The final state of the registers matches the expected behaviour. In register `a0` we have a 3 and in the same way, the `ra` (Return Address) and `sp` (Stack Pointer) matches with the behaviour of the compiler's generated code. The `objdump` of the binary we used for this test can be found in the Appendix B.

3.4 Linux

Linux Kernel is an open-source computer operating system kernel first released on 1991, by Linus Torvalds. Operating Systems packing the Linux Kernel are known as Linux Distributions. These are currently wide used on both, traditional personal computers and servers, and on variuous embedded devices such as routers, access-points, smart TVs, Network Attached Storage systems, etc. In RISC-V the most popular option is Buildroot[20] due to its simplicity and the easy-to-use process of generating embedded Linux systems through cross-compilation.

For Ariane, there is an Open Source repository[21] containing the necessary tools (including Buildroot) to build a Linux Image from scratch. In the next subsections we are going to explain the main requirements a RISC-V processor has to support in order to boot a Linux based Operating System.

3.4.1 Device Tree

For the previous tests, we successfully executed a simple code without any issue, however, in order to run an Operating System like Linux in an embedded system, a Device Tree is required. The Device Tree is basically a data structure describing the hardware components available in the system. This information is required by the kernel so it can use and manage those components. Typically the Device Tree describes the number of cores (and its frequency), the memory and its (AXI4) address range and the peripherals such as the Platform Level Interrupt Controller (PLIC), Core Local Interrupt Controller (CLINT) and the UART device (frequency, baudrate, AXI4 address range, etc). Figure 3.18 shows a snippet of code from the Device Tree describing the properties of the core.

```
/dts-v1/;

/ {
  ...
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    timebase-frequency = <32768>; // 32.768 kHz
    CPU0: cpu@0 {
      clock-frequency = <50000000>; // 50 MHz
      device_type = "cpu";
      reg = <0>;
      status = "okay";
      compatible = "eth, ariane", "riscv";
      riscv,isa = "rv64imacsu";
      mmu-type = "riscv,sv39";
      tlb-split;
      // HLIC - hart local interrupt controller
      CPU0_intc: interrupt-controller {
        #interrupt-cells = <1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
      };
    };
  };
  ...
};
```

Figure 3.18: Snippet of code from Ariane's Device Tree . Full description available in Appendix C.

The usage of the Device Tree is free and open source, and the current specification[22] targets small systems. Traditional computers using an x86 architecture generally do not use a device tree, instead they use auto configuration protocols to discover the hardware. Embedded Systems typically do not change, therefore

[20] https://www.buildroot.org

[21] https://github.com/pulp-platform/ariane-sdk

[22] https://www.devicetree.org/specifications/

there is no need of relying on automatic detection of the devices. Instead, this information is directly provided by the device tree. However, there are boot loaders (*e.g.* U-Boot) that supports reading the Device Tree from a file and load it into an specific memory address. This approach is often used in SoCs with an embedded FPGA, as you may program the FPGA with a custom device and this must appear in the Device Tree in order to be usable from the Linux environment. In `RISC-V`, the current available bootloaders do not support this approach and a static Device Tree is written into a ROM (Read Only Memory). In our case, this must be done before generating the FPGA bitstream[23], therefore if we want to change the Device Tree we have to regenerate the bitstream.

3.4.2 Bootrom

The bootrom device as its name state is a ROM that is read during the boot. In fact, it contains the first instructions that a processor executes when it powers on or resets. In `RISC-V`, the most typical use case is to load into the `x10` and `x11` registers the hart id and the Device Tree address respectively and then jump into the bootloader (Figure 3.19). The Device Tree address is not fixed, instead, the way how it is done is to include the Device Tree (`ariane.dtb`) into the assembly code using the keyword `.incbin` and define a label (`_dtb`). With this label, in the assembly code we can refer to the Device Tree address without knowing the actual value. The final address of the Device Tree is decided during the compilation and linking process.

```
#define DRAM_BASE 0x80000000

.section .text.start, "ax", @progbits
.globl _start
_start:
  li s0, DRAM_BASE
  csrr a0, mhartid
  la a1, _dtb
  jr s0

.section .text.hang, "ax", @progbits
.globl _hang
_hang:
  csrr a0, mhartid
  la a1, _dtb
1:
  wfi
  j 1b

.section .rodata.dtb, "a", @progbits
.globl _dtb
.align 5, 0
_dtb:
.incbin "ariane.dtb"
```

Figure 3.19: `RISC-V` Bootrom code

3.4.3 Bootloader

After the Bootrom stage, it comes the Bootloader. The Bootloader is the responsible of loading the final OS (Linux). This includes initializing all the devices to the state in which Linux expects them to be. In `RISC-V`, the most common one is BBL (Berkeley Bootloader), which is a First Stage Bootloader (FSBL). Unlike in the Bootrom, the Bootloader is not embedded in the hardware nor its unmodifiable. The Bootloader it is packed together with the Linux image and must be loaded into the main memory everytime it boots.

[23] An FPGA bitstream is a file that contains the programming information for an FPGA.

BBL expects the processor to be running in `machine` mode and to already have the Device Tree at some address in the memory. The BBL will do the following steps [38]:

- Select one main hart (Hardware Thread) and put the other harts into sleep until it is time to pass the control to Linux.
- The Device Tree passed from the previous stage (Bootrom) is read and filtered. This is done in order to strip out information that is not relevant for Linux (*e.g.* platform specific information).
- All the other harts are woken up so they can setup their Physical Memory Protection (PMP), trap handlers and enter supervisor mode.
- The `mhartid`[24] register is read in order to ensure that a unique per-hart identifier is passed to Linux.
- Set up a PMP to allow Supervidor mode to access all the memory (the hart is in Machine mode at this point).
- Set up Machine mode trap handlers. BBL's machine mode code must handle unimplemented instructions and machine-mode interrupts.
- The processor executes the instruction `mret` which will change the privilege level from Machine mode to Supervisor.
- BBL jumps to the first address of the Linux Kernel.

In Ariane's case, the bootloader is stored in the address `0x80000000`, which is the instruction address where we jump in Figure 3.19's code (`DRAM_BASE`).

3.4.4 Booting Linux

In this subsection we are going to focus on which steps you have to follow in order to boot Linux in the Ariane core running on an FPGA, this includes the steps to follow in order to generate a Bootloader that contains Linux. For accomplishing this task we first need a compiler that can generate `RISC-V` instructions. In our case, we used the `RISC-V GNU Toolchain`. Figure 3.20 shows the commands to build the compiler.

```
yxu@laptop:$ git clone https://github.com/riscv/riscv-gnu-toolchain.git
yxu@laptop:$ cd riscv-gnu-toolchain
yxu@laptop:$ git reset --hard 45f5db5a2dc167ef040c70143b94a806912f5771
yxu@laptop:$ git submodule update --init --recursive
yxu@laptop:$ cd ..
yxu@laptop:$ mkdir -p toolchain_build
yxu@laptop:$ mkdir -p riscv-gnu-toolchain/build
yxu@laptop:$ export RISCV=$(pwd)/toolchain_build
yxu@laptop:$ cd riscv-gnu-toolchain/build
yxu@laptop:$ ../configure --prefix=$RISCV --with-arch=rv64imac --with-abi=lp64 --disable-gdb
yxu@laptop:$ make -j8
yxu@laptop:$ make linux -j8
yxu@laptop:$ cd ../..
```

Figure 3.20: Building the `RISC-V` Cross-Compiler GNU Toolchain

Once we have the cross compiler, we can start building the Linux kernel image. For this task, we are using Buildroot as mentioned in the previous subsections. It is important to note that Buildroot expects 2 configuration files (`busybox.config` and `linux_defconfig`) to be in the `configs` directory. Figure 3.21 shows the commands we followed in order to generate the `vmlinux` file. The configuration we used (`buildroot_defconfig`) can be found in Appendix D.

[24] The `mhartid` is a CSR register containing an integer ID of the hardware thread running the code.

```
yxu@laptop:$ mkdir -p configs rootfs && cd configs
yxu@laptop:$ export BASE_URL="https://raw.githubusercontent.com/pulp-platform/ariane-sdk/master"
yxu@laptop:$ wget $BASE_URL/configs/linux_defconfig
yxu@laptop:$ wget $BASE_URL/configs/busybox.config
yxu@laptop:$ wget $BASE_URL/configs/buildroot_defconfig
yxu@laptop:$ wget $BASE_URL/configs/0001-Add-RISC-V-architecture-to-Xilinx-ethernet-Kconfig.patc
yxu@laptop:$ wget $BASE_URL/configs/0002-emaclite-Align-buffer-and-iomem-on-64-bits.patch
yxu@laptop:$ cd ..
yxu@laptop:$ export PATH=$(pwd)/toolchain_build/bin:$PATH
yxu@laptop:$ git clone git://git.buildroot.net/buildroot buildroot
yxu@laptop:$ make -C buildroot clean
yxu@laptop:$ make -C buildroot defconfig BR2_DEFCONFIG=../configs/buildroot_defconfig
yxu@laptop:$ make -C buildroot && cp buildroot/output/images/vmlinux vmlinux
```

Figure 3.21: Building the Linux Kernel image for Ariane.

The next step is to build the BBL which will have as payload the `vmlinux` file we just generated (Figure 3.22).

```
yxu@laptop:$ git clone https://github.com/pulp-platform/riscv-pk.git
yxu@laptop:$ cd riscv-pk && git reset --hard e2c9af30180eeeb751428d4821ebfcbdc6513ecf
yxu@laptop:$ git submodule update --init --recursive
yxu@laptop:$ cd ..
yxu@laptop:$ mkdir -p build && cp vmlinux build/.
yxu@laptop:$ cd build
yxu@laptop:$ ../riscv-pk/configure --host=riscv64-unknown-elf --with-payload=vmlinux && cd ..
yxu@laptop:$ make -C build && cp build/bbl bbl
```

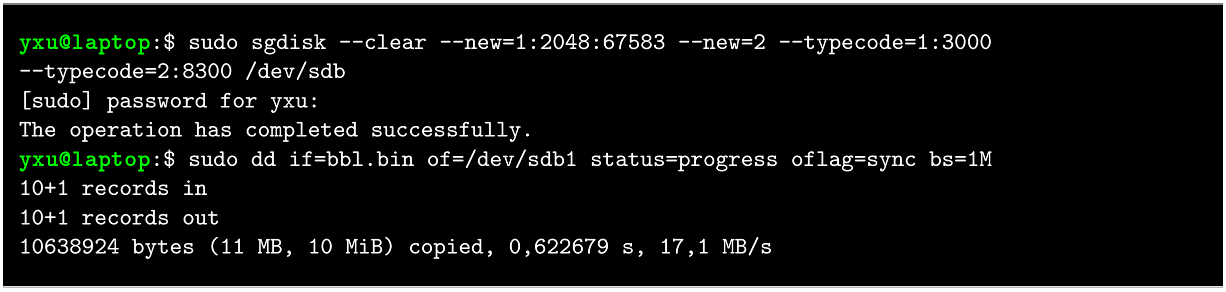
Figure 3.22: Generating the `bbl` file with the `vmlinux` as a payload.

At this point we have generated the `bbl` file, but the file is in ELF format, this means that inside the file, we can find information regarding the file content (*e.g.* Machine Architecture) apart from the actual executable code. In our case, we are interested in having only the explicit instructions to be loaded into the memory. For this reason, we must convert the ELF file into a Bin file (Figure 3.23).

```
yxu@laptop:$ riscv64-unknown-linux-gnu-readelf -h bbl
ELF Header:
  ...
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x80000000
  Flags:                             0x1, RVC, soft-float ABI
  ...
yxu@laptop:$ riscv64-unknown-elf-objcopy -S -O binary --change-addresses -0x80000000 bbl bbl.bi
```

Figure 3.23: Converting the `bbl` ELF into a Bin file.

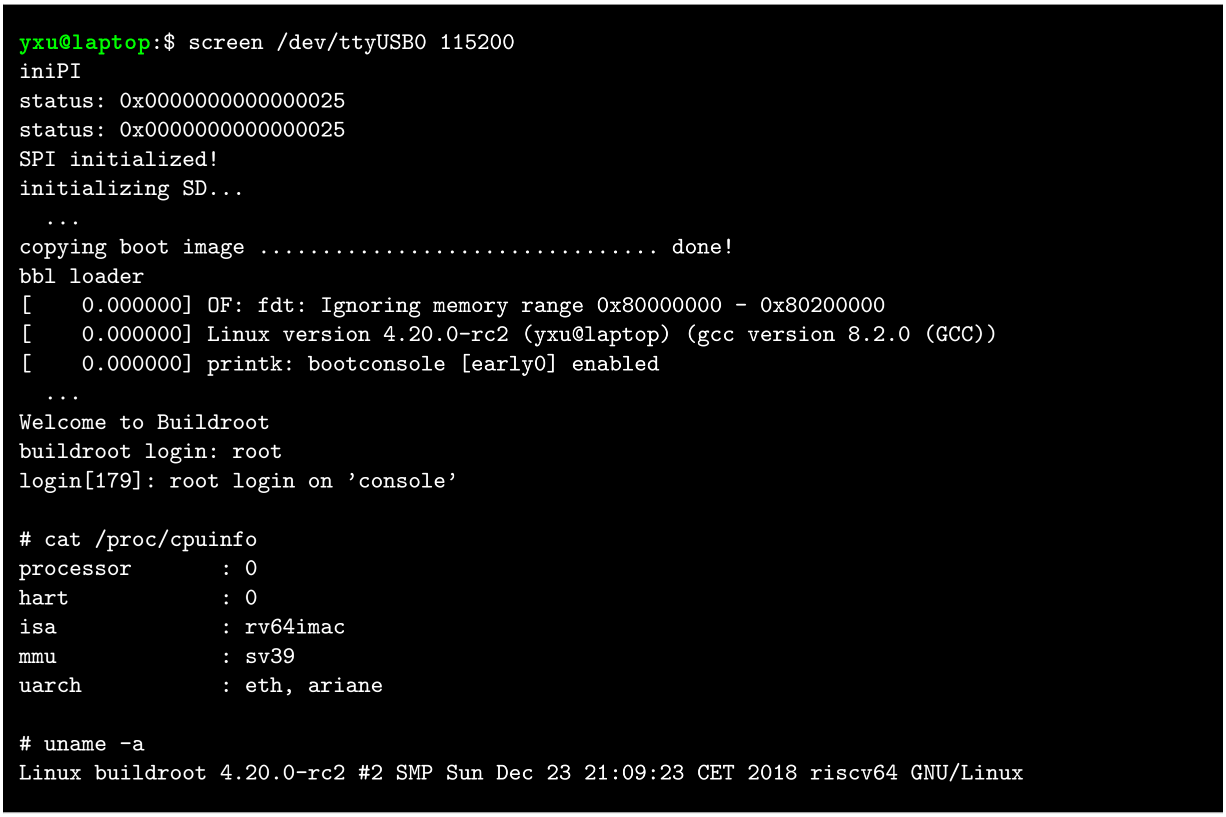
Now we are ready to boot Linux on our FPGA. The way how Ariane copies the BBL into the main memory is through the MicroSD card. The bootloader of Ariane requires a GPT partition table so we first have to create one. To do so, we have to connect the MicroSD into the computer and check the name of the device (`sdb` in this case). Figure 3.24's first command shows how to create the GPT partition table and two partitions, the first partition of 32MB for the `bbl.bin` and the second one for the Linux root. The second command is for copying the `bbl.bin` file into the MicroSD.



```
yxu@laptop:$ sudo sgdisk --clear --new=1:2048:67583 --new=2 --typecode=1:3000
--typecode=2:8300 /dev/sdb
[sudo] password for yxu:
The operation has completed successfully.
yxu@laptop:$ sudo dd if=bbl.bin of=/dev/sdb1 status=progress oflag=sync bs=1M
10+1 records in
10+1 records out
10638924 bytes (11 MB, 10 MiB) copied, 0,622679 s, 17,1 MB/s
```

Figure 3.24: Preparing the SD card to boot Linux.

Now we are ready to boot Linux! We have to program first the FPGA with the Ariane bitstream[25] and then with the MicroSD inserted in the FPGA, Ariane will boot Linux. From this point, the interaction with Ariane will be done through the serial port, so it is important to have the FPGA UART port connected to the computer. Figure 3.25 shows a snippet of the messages you find once it starts booting. Once the device booted, the user is `root` and no password is required.



```
yxu@laptop:$ screen /dev/ttyUSB0 115200
iniPI
status: 0x0000000000000025
status: 0x0000000000000025
SPI initialized!
initializing SD...
  ...
copying boot image ............................... done!
bbl loader
[    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
[    0.000000] Linux version 4.20.0-rc2 (yxu@laptop) (gcc version 8.2.0 (GCC))
[    0.000000] printk: bootconsole [early0] enabled
  ...
Welcome to Buildroot
buildroot login: root
login[179]: root login on 'console'

# cat /proc/cpuinfo
processor       : 0
hart            : 0
isa             : rv64imac
mmu             : sv39
uarch           : eth, ariane

# uname -a
Linux buildroot 4.20.0-rc2 #2 SMP Sun Dec 23 21:09:23 CET 2018 riscv64 GNU/Linux
```

Figure 3.25: Booting Linux in Ariane running on an FPGA.

[25] For this test, we used the Ariane version 4.0 and the bitstream available in the release

Chapter 4

The k-means algorithm

Current comercial and enterprise applications such as e-commerce, health monitoring, industrial production and financial data analysis rely more and more on Machine Learning techniques to get the best result.

In [39], the importance of organizing data has been deeply discussed, analyzing the evolution of clustering algorithms since k-means appeared. One of the most important reasons of clustering algorithm's popularity in the scientific community is that cluster analysis and classification are present in a wide range of disciplines, specially the ones involving multivariate data analysis.

The k-means clustering algorithm has been used in several studies to compare the performance of different platforms. The algorithm itself results to be of high interest for such comparative studies due to the characteristics it presents [40]. Firstly, the iterative nature of the algorithm implies that the current iteration results are needed in the next iteration. Secondly, calculating the centroids is a compute-intensive task. And thirdly, in order to obtain the global solution when the algorithm is parallelized, a reduction of the local results is needed. However, clustering algorithms are not only benchmarking algorithms, they have been used to solve computationally demanding applications.

Aingura IIoT presented in [41] a real industrial application from acquisition to processing and interpretation of industrial data. In this application, a set of different machine learning techniques including k-means clustering are used to develop a knowledge discovery application for a real industrial use case.

In the framework of the collaboration, the study was restricted to a thermal process performed by a laser over mechanical pieces supervised by a high frequency thermal camera tracking 32×32 pixel pictures every 1 ms. The stream of frames needs to be analyzed with clustering techniques within a given time window in order to find anomalies in the thermal process. The industrial machine performing this process is equiped with a compute unit based on `Arm` cores plus programmable logic (FPGA) that can be used as an accelerator. This unit is responsible of acquiring the data from the sensors (in this case from the camera) and perform the clustering algorithm. The computational intensity of the algorithm is not trivial and involves heavy floating point computation that can be tackled using techniques derived from High Performance Computing.

4.1 Algorithm analysis

The k-means algorithm consists of classifying a set of N points of D dimension in K different groups, called *clusters*. The criteria for classifying the points is to minimize the intra-class variance, *e.g.* minimizing the sum of squared distances from each point to the cluster point. It is a well known clusterization technique already applied in similar cases [42].

Typically, the k-means++ algorithm [43] is used to initialize the cluster centers (centroids) before proceeding with the standard k-means. This algorithm specifies a procedure to initialize the cluster centers (centroids) before proceeding with the standard k-means. This algorithm helps avoiding poor clusterings found by the standard k-means algorithm and to converge to the desired solution faster.

As part of the k-means problem consists in computing distances between two points, the binomial theorem can be applied. The computation of the distance between two points is mathematically defined in Equation 4.1.

$$d^2 = \sum_{i=1}^{D} (c_i - p_i)^2 \tag{4.1}$$

where c_i is the coordinate of the centroids and p_i the coordinate of each of the point to clusterize. In our case $D = 1024$, as we are working with 32×32 pixels images. The binomial theorem defines the following equivalence:

$$(c_i - p_i)^2 = c_i^2 + p_i^2 - 2c_i p_i \tag{4.2}$$

Applying it to the original distance formula, we obtain that the distance can be expressed as:

$$d^2 = \sum_{i=1}^{D} (c_i^2 + p_i^2 - 2c_i p_i) \tag{4.3}$$

We can also note that, as the points do not change their position during the clustering process, any operation that only involves the p_i can be precomputed (memoization) and reused each time is needed. In our case, all the p_i^2 are computed at the beginning as a simple dot product before starting the k-means algorithm. Following the same idea, the c_i^2 operations are computed as a dot product, but this time during each iteration (as the centroids change their value during the clustering iterations).

The $c_i p_i$ operation can be computed as a matrix multiplication of the matrices P, storing the D coordinates of the N points to clusterize, and the transposed of the matrix C storing the D coordinates of the K centroids.

$$P = \begin{bmatrix} p_1^1 & p_2^1 & \cdots & p_D^1 \\ p_1^2 & p_2^2 & \cdots & p_D^2 \\ \cdots & & & \\ p_1^N & p_2^N & \cdots & p_D^N \end{bmatrix} \qquad C^T = \begin{bmatrix} c_1^1 & c_1^2 & \cdots & c_1^K \\ c_2^1 & c_2^2 & \cdots & c_2^K \\ \cdots & & & \\ c_D^1 & c_D^2 & \cdots & c_D^K \end{bmatrix}$$

Each cell of the final matrix is then multiplied by the constant -2. The resulting matrix will contain for each cell, the $-2c_i p_i$ operation that is part of the original expression of d^2.

Computational cost of the matrix implementation

As result of applying the optimizations explained in Section 4.1, the number of operations have been reduced as some of the operations are precomputed once or within an iteration. Table 4.1 shows the computational cost of the two implementations for the size evaluated. Note that, following the original k-means algorithm, for each iteration, the distance from the points to the centroids is calculated. The distance formula is computed K times for each point and, as the centroids do not change their position within an iteration, the operation c_i^2 can be precomputed at the beginning of each iteration.

	Precomputed (once)	Per iteration
Original	-	$7DNK$
Optimized	$2DN$	$(2DK) + (2 + 2D)NK$

Table 4.1: Computational cost comparison: dimension (D), num. elements (N), num. centroids (K).

Chapter 5

Test and results

In this section we are going to discuss the performance of 4 RISC platforms executing the algorithm explained in the previous section. This includes the methodology we followed in order to obtain the results and the environment setup where we run the tests. As our two target architectures (Arm and RISC-V) are not at the same level of maturity, comparing them using their best setup is not fair. Instead, we will first do a CPU only test comparing the single core performance of both architectures and then, we will make a comparison of only Arm platforms but using the accelerator and a very optimized linear algebra library. This comparision will show how different the Arm platforms are at their best setup.

5.1 Platforms

During this thesis, we had the chance to play with 4 different platforms: 2 heterogeneous Arm boards and 2 RISC-V platforms. The technical details of each platform will be discussed in the following subsections.

5.1.1 Arm

Table 5.1 shows the Arm platforms we used and their main architectural features. One board has a 32-bit processor (Cortex A9) and the other one has a 64-bit processor (Cortex A57). These processors are from different generations, however, both feature an accelerator embedded inside the SoC.

	Zynq 7020		**Jetson TX1**	
	CPU	FPGA accel.	CPU	GPU accel.
Compute resources	2× Cortex-A9	106.4k FFs, 53.2k LUTs, 220 DSPs	4× Cortex-A57	256 Maxwell CUDA cores
L1 I-cache	32KB 4-way	-	48KB 3-way	-
L1 D-cache	32KB 4-way	-	32KB 2-way	-
Frequency [MHz]	667	200	up to 1730	up to 998
Memory [MB]	1024	4.9 + CPU mem	4096	shared with CPU
Interconnection	1 GbE (native)	-	1 GbE (USB3 bridge)	-

Table 5.1: Technical specifications of the evaluated Arm platforms.

The 32-bit processor (Xilinx Zynq 7020) has an FPGA that is directly connected to the CPU and the 64-bit processor (Nvidia Jetson TX1) has a GPU (Graphics Processing Units) with CUDA cores. It is important to note that in both cases, we are in front of a multi-core setup.

In both platforms we installed a standard software stack for scientific computing, including Linux OS (Ubuntu), Network File System (NFS), GNU Compiler Suite together with linear algebra (ATLAS) and communication libraries (MPI). We operated the compute nodes as nodes of a cluster and we took advantage of the Mont-Blanc system software stack already deployed on Arm-based clusters. A key part of the software stack installed in these machines is the OmpSs programming model [44, 45], composed of the source-to-source Mercurium compiler and the Nanos++ runtime library. OmpSs is a task-based programming model with explicit inter-task dataflow that allows the runtime system to orchestrate out-of-order execution of the tasks, selectively off-loading tasks to the GPU/FPGA when possible. OmpSs is developed at Barcelona Supercomputing Center and in this thesis has been used on the CPU plus accelerator part to maintain a single portable and scalable code that can be executed on parallel heterogeneous devices by only changing few pragmas.

5.1.2 RISC-V

Table 5.2 shows the RISC-V platforms we used. On one hand we have Ariane, which is not a physically available platform, instead it is only HDL (Hardware Description Language) code synthesized and running on the Genesys 2[26] FPGA. The reason why there is not a physical board using Ariane yet is because it is still in development. For this reason, at the current state, Ariane does not support multi-core configurations and lacks of a synthesizable Floating Point Unit (FPU). In the same way, the total memory and the interconnection is not known as any option is possible. The resources utilization of Ariane in the Genesys 2 FPGA is: 10% of BRAMs, 2% of DSPs, 11% of Flip-Flops and 24% of LUTs. On the other hand, the HiFive Unleashed is a board with a real RISC-V chip. The SoC packs 4 cores based on the Rocketchip (open-source) and supports the floating point extensions. Note that unlike in the Arm platforms, our RISC-V platforms lack an accelerator.

	Ariane	**HiFive Unleashed**
# Cores	1	4
ISA Extensions	RV64IMAC	RV64GC (RV64IMAFDC)
L1 I-cache	32KB 4-way	32KB 8-way
L1 D-cache	64KB 8-way	32KB 8-way
Frequency	1.5 GHz (50MHz on FPGA)	1.5 GHz
Memory	-	8 GB
Interconnection	-	1 GbE (native)

Table 5.2: Technical specifications of the evaluated RISC-V platforms.

Both platforms are capable of booting Linux but because they run in a very limited environment, everything is cross-compiled from an x86 machine. On the other hand, it is important to remember that there is no standard software stack ready for scientific computing in RISC-V. This includes the lack of very optimized linear algebra libraries (*e.g.* BLAS), communication libraries (*e.g.* MPI) and job scheduling systems (*e.g.* SLURM). There is OpenMP support for exploiting the parallelism in multicore environments, but in single core, the programs only count with the compiler optimization flags to increase the performance.

5.2 CPU only

In this section we are going to discuss the performance of both architectures (Arm and RISC-V). It is important to notice that the Arm architecture is way more mature than the RISC-V one. The current software ecosystem in Arm gives much more advantage in terms of performance than RISC-V's ecosystem can provide. For example the Vector extension (SIMD) is not ready yet, which is very important in HPC workloads and most of the current available RISC-V 64-bit implementations are single core (*e.g.* Ariane). For these reasons, in this test we are going to limit the scope to a single core comparison without using scientific libraries like BLAS and relying only on the optimization flags of the available compilers.

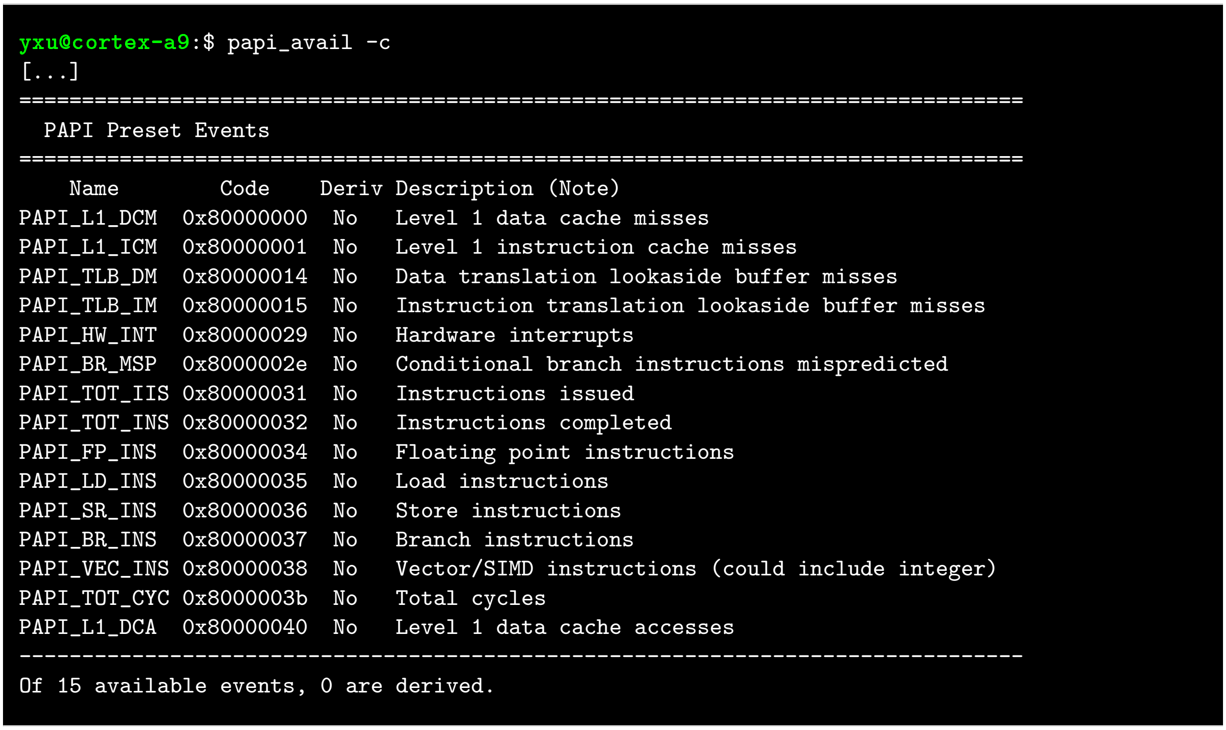
We are going to use the k-means algorithm as our reference benchmark for measuring the performance. Regarding the input set, we are using a real industrial dataset provided by Aingura IIoT. This dataset represents a short video of 21.5 seconds at 1000 FPS ($N = 21500$). Each frame has a resolution of 32×32 pixels, giving a total of $D = 1024$ pixels per frame. The initial centroids are chosen using k-means++ algorithm which helps avoiding to fall into local optimums and get poor clusterings.

[26] https://store.digilentinc.com/genesys-2-kintex-7-fpga-development-board

5.2.1 Methodology

We run the k-means algorithm with the following parameters: dimension $D = 128$, number of elements to clusterize $N = 128$, number of clusters $K = 16$, tolerance (convergence threshold) = 0 and 1 repetition.

For measuring the performance we relied on the information given by the hardware counters available inside the processor. In Arm we have the PAPI (Performance Application Programming Interface) library, which provides a portable way for reading the hardware counters in different platforms and architectures. Appendix E shows an example of the functions we used for reading the hardware counters. Figure 5.1 shows the command we used to check the available counters in our Arm machines.



```
yxu@cortex-a9:$ papi_avail -c
[...]
================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No   Level 1 instruction cache misses
PAPI_TLB_DM  0x80000014  No   Data translation lookaside buffer misses
PAPI_TLB_IM  0x80000015  No   Instruction translation lookaside buffer misses
PAPI_HW_INT  0x80000029  No   Hardware interrupts
PAPI_BR_MSP  0x8000002e  No   Conditional branch instructions mispredicted
PAPI_TOT_IIS 0x80000031  No   Instructions issued
PAPI_TOT_INS 0x80000032  No   Instructions completed
PAPI_FP_INS  0x80000034  No   Floating point instructions
PAPI_LD_INS  0x80000035  No   Load instructions
PAPI_SR_INS  0x80000036  No   Store instructions
PAPI_BR_INS  0x80000037  No   Branch instructions
PAPI_VEC_INS 0x80000038  No   Vector/SIMD instructions (could include integer)
PAPI_TOT_CYC 0x8000003b  No   Total cycles
PAPI_L1_DCA  0x80000040  No   Level 1 data cache accesses
--------------------------------------------------------------------------------
Of 15 available events, 0 are derived.
```

Figure 5.1: Checking the available PAPI counters in the Zynq 7020.

In RISC-V there is no PAPI version available yet, however, the ISA offers mechanisms to the designers to implement hardware counters in their RISC-V processors. There are 3 mandatory basic counters that any RISC-V processor must provide. These are the total number of cycles, the total number of retired instructions and the real time, which counts wall-clock real time that has passed from an arbitrary start time in the past. To read these counters, there are 3 mandatory instructions: `rdcycle`, `rdinstret` and `rdtime` respectively. The way how we read the counters can be seen in Figure 5.2. We took advantatge of the Extended Asm feature present in the compiler in order to add assembler instructions with C expression operands.

```c
unsigned long read_cycles(void) {
    unsigned long cycles;
    asm volatile ("rdcycle %0" : "=r" (cycles));
    return cycles;
}
unsigned long read_loads(void){
   unsigned long result;
   asm volatile ("csrr %0, hpmcounter7" : "=r" (result));
   return result;
}
```

Figure 5.2: Functions to read the hardware counters in RISC-V using Extended Asm from GCC.

However, that is not all. The hardware performance monitor includes 29 additional user-level event counters (`hpmcounter3-hpmcounter31`). The events to count are platform specific and the `RISC-V` specification does not limit it. It is important to notice that the specification forces the counters to be 64-bit width even for the 32-bit architecture. Table 5.3 shows the addresses of these registers.

Address	**Name**	**Description**
		User Counter/Timers
0xC00	cycle	Cycle counter for `rdcycle` instruction.
0xC01	time	Timer for `rdtime` instruction.
0xC02	instret	Instructions-retired counter for `rdinstret` instruction.
0xC03	hpmcounter3	Performance-monitoring counter.
0xC04	hpmcounter4	Performance-monitoring counter.
	⋮	
0xC1F	hpmcounter31	Performance-monitoring counter.
0xC80	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	timeh	Upper 32 bits of time, RV32I only.
0xC82	instreth	Upper 32 bits of instret, RV32I only.
0xC83	hpmcounter3h	Upper 32 bits of hpmcounter3, RV32I only.
0xC84	hpmcounter4h	Upper 32 bits of hpmcounter4, RV32I only.
	⋮	
0xC9F	hpmcounter31h	Upper 32 bits of hpmcounter31, RV32I only.

Table 5.3: `RISC-V` user-level performance-monitoring registers addresses.

When running a code using the debug module with OpenOCD and GDB (Section 3.3), we can read the hardware counters registers too. In order to do so, we have to make sure that the `*.cfg` file we pass to OpenOCD contains the following line:

```
# ariane.cfg file
adapter_khz 1000
...
riscv expose_csrs 3071-3086
init
```

By adding the above line into the configuration file, we will be able to read the hardware counters inside GDB using the `p` and/or the `info` commands:

```
yxu@laptop:$ riscv64-unknown-elf-gdb add.exe -ex "target extended-remote localhost:3333"
[...]
(gdb) p /u $csr3076
$1 = 59
(gdb) info reg csr3076
csr3076        0x3B     59
(gdb)
```

In order to measure the absolute execution time, as our embedded platforms run at a fixed frequency, we computed the execution time as the number of cycles divided by the frequency of the processor in Hertz. For example, Ariane runs at 50MHz on the FPGA and measuring the execution time will not be fair. Instead we considered the expected tapped out frequency of 1.5Ghz at 22nm (GlobalFoundries 22FDX) [46].

All these counters are part of the CSR (Control Status Registers) registers. For more information about CSR registers, you can check the *Chapter 2 - Control and Status Registers (CSRs)* from the `RISC-V` privileged specification (version 1.10). In this chapter more insights regarding the machine-level counters will be shown, however, for this thesis we will limit the scope to the user-level registers shown above.

Regarding the execution, in the `Arm` platforms we executed the application inside a Linux environment, however, in order to reduce the noise as much as possible, the machine where the application run was exclusively allocated for our tests using SLURM. For the two `RISC-V` platforms, our initial intention was to use Linux too. However, our preliminary tests in Ariane shown that the Linux execution was not reliable (Figure 5.3). This is due the fact that in Linux we have more than one process running at a time, and as there is only one core in Ariane, our measurements were heavily affected by the context switches between processes. For this reason, the tests in Ariane have been done using the debug module in combination of OpenOCD and GDB. In the HiFive Unleashed, as it has more than one core, our preliminary tests did not register any affecting noise, so the execution in this platform was done in Linux. Due to lack of `RISC-V` support in SLURM, we run the application in interactive mode, but we double checked the setup in the board such that no other heavy load process could interfere with the execution.

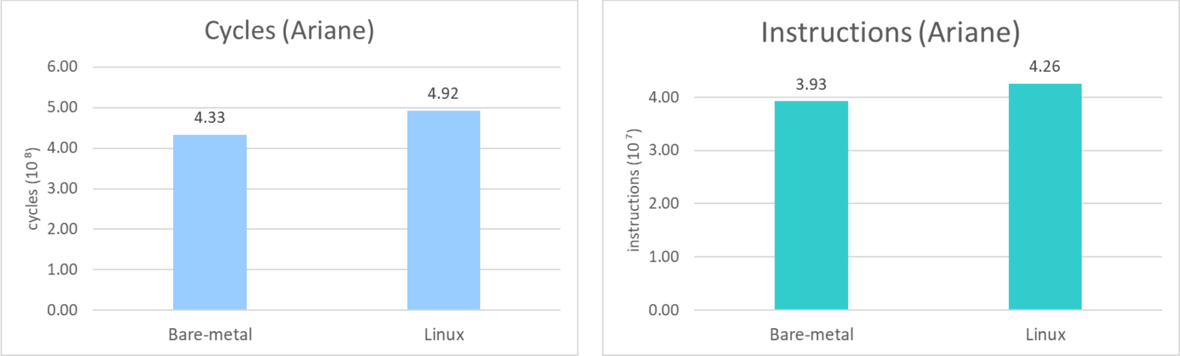


Figure 5.3: Ariane executing k-means in Bare-metal vs Linux. Compiled with GCC Unknown Elf.

A side effect of running an application using the debug module (Ariane) is that there is no Operating System. This means that we cannot make a `printf` neither open a CSV file nor allocate dynamic memory using a `malloc`. For this reason, we had to adapt our application to run in a bare-metal environment. The steps we followed are:

- Embed the CSV into the application binary.
- Use global variables to store the data.
- Implement custom functions for reading the CSV file.
- Compile using the `-static` flag.

First, for embedding the CSV file into the application binary we used the `.incbin` directive (Figure 5.4). This directive takes any file and includes it within the file being compiled. The file is included as it is, without being assembled. We also define the `csv_begin` and `csv_end` symbols that are used to know the address where the CSV is stored. Now, for reading the file, we implemented a custom function called `read_csv()`. Given the number of elements per line and the number of lines, this function reads the CSV data (from an address pointer) and writes it into an array (global variable) which size must be known at compile time. This way is how we can handle the lack of dynamic memory and not being able to open a CSV file. Refer to the Appendix F for the C code.

```
.data

.global csv_begin
csv_begin:
.incbin "input.csv"
.global csv_end
csv_end:
```

Figure 5.4: `input.s` file containing CSV input file.

The final step is to compile everything. It is important to remeber that in order to compile the `input.s` file, we used the GNU assembler utility (*e.g.* `riscv64-unknown-elf-as`) that reads an `*.s` file and compiles it into a `*.o` file. This file is later linked to the final binary using gcc (*e.g.* `riscv64-unknown-elf-gcc`). Appendix G shows the Makefile we used for compiling the application.

About the compilers available in each platform, we tested both GNU and LLVM versions of each architecture (Table 5.4). GNU Compiler Collection (GCC) is free and open and it is produced by the GNU Project. Testing it is very important because it is the standard compiler for most Unix like operating systems. On the other hand, LLVM+Clang is another really popular option. The standard version is free, open-source and its customization potential makes it perfect for some scenarios.

For `Arm` we used the GCC compiler that came with the Linux distribution while in the LLVM case, we tested the Arm HPC Compiler (a version of Clang/LLVM optimized by `Arm`) in the Jetson TX1 and because the Armv7 architecture is not supported in this compiler, we tested the standard Clang/LLVM in the Zynq 7020. In the case of GNU in `RISC-V`, we tested the Unknown Elf and the Linux GNU version. On the LLVM side, we used a preliminary version of Clang/LLVM toolchain developed in the Barcelona Supercomputing Center called EPI Compiler.

	GNU	**LLVM**
Zynq 7020	GGC 6.2.0	Clang 6.0
Jetson TX1	GGC 5.4.0	Arm HPC Compiler 19.0
Ariane	GCC Unknown Elf 8.1.0 GCC Linux GNU 8.2.0	EPI Compiler
HiFive Unleashed	GCC Unknown Elf 8.1.0 GCC Linux GNU 8.2.0	EPI Compiler

Table 5.4: Tested compilers in `Arm` and `RISC-V`.

For consistency reasons, not only Ariane but also the rest of platforms have been tested using the bare-metal ready version described in this section.

5.2.2 Evaluation

As first approach, due to the different compilers available in each architecture, we evaluated the impact they have on the performance in each of our platforms. In the `Arm` machines (Figure 5.5, we see a different trend between the standard Clang/LLVM (used in the Zynq7020) and the Clang/LLVM version optimized by `Arm` (used in the Jetson TX1). In the Zynq, the Cycles Per Instruction (CPI) in LLVM are lower (which is better) however, the actual execution time is higher than GCC, which means that Clang/LLVM generates more but less complex instructions from the same code, but in the end, the extra number of instructions penalized the performance. In the Jetson case, `Arm` did a great job optimizing their compiler and even if the CPI is greater in LLVM, the final performance (time) is 1.12× faster. This means that `Arm` managed to generate a code that is more compact (less instructions) but they do more (require more cycles per instructions) resulting in a better usage of the processor microarchitecture.

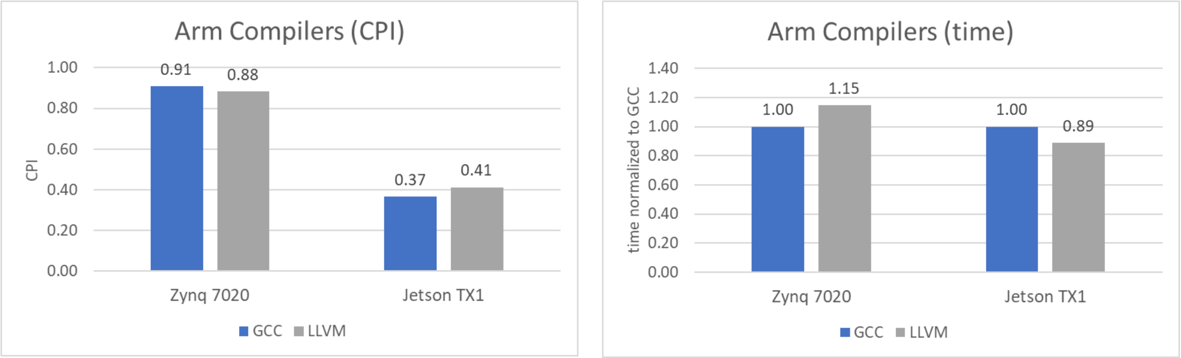


Figure 5.5: Comparison of the `Arm` compilers executing k-means (integers) with -O3 flag. For the figure on the right, baseline time is 36 miliseconds for Zynq and 7 miliseconds in Jetson TX1.

For the `RISC-V` machines (Figure 5.6, we found a similar trend as in the standard Clang/LLVM of the Zynq. The performance is worse when using the custom Clang/LLVM. The GCC Linux and the LLVM compilers we used target Linux environments, therefore, the generated code, expects to be running in Linux for the correct functioning. For this reason, in Ariane we tested the GCC Unknown Elf in a bare metal environment and the GCC Linux and LLVM in Linux. That is the reason why the execution time is lower in GCC Unknown Elf, but the important point of the comparison is that it shows the same trend as in Arm, where the LLVM performance is worse. For the HiFive Unleashed, the performance is the same for both GCC Unknown Elf and GCC Linux, but the LLVM shown a higher CPI and execution time. Analyzing the raw numbers, we see that for the three compilers, LLVM generates 12% more instructions and takes 24% more cycles than both GCCs, hence the increase in CPI and the execution time. This behaviour is somehow expected as the Clang/LLVM version we used in RISC-V still on a preliminary stage.

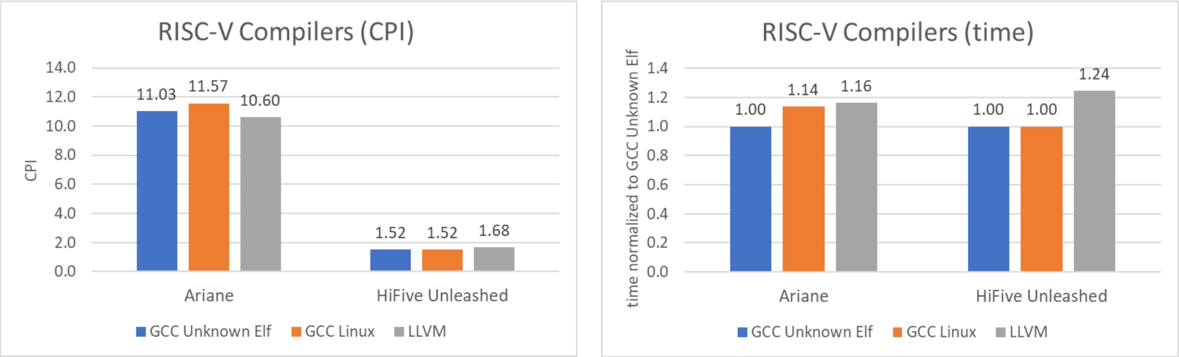
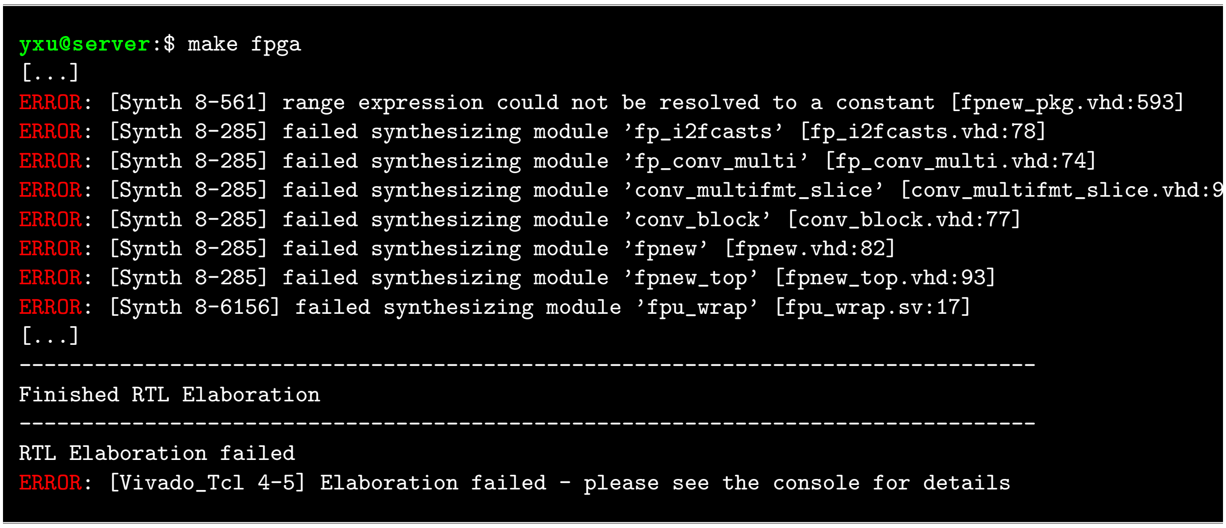


Figure 5.6: Comparison of the RISC-V compilers executing k-means (integers) with -O3 flag. For the figure on the right, baseline time is 289 miliseconds for Ariane and 40 miliseconds in HiFive Unleashed.

Considering the non-availability of the Arm HPC Compiler for Armv7, we decided to stick to GCC in the future tests as GCC is free, open source and available in most of the Linux distributions. On the RISC-V side, as the Clang/LLVM compiler still on a preliminary stage we also decided to use GCC for our next tests. From the two GCC version available in RISC-V, both shown to perform equally good and for convenience, we ended up using the GCC Unknown Elf, which runs in bare-metal and in Linux without any issue. For each case, we tested the 4 different optimization flags offered by the compiler (-O0, -O1, -O2, -O3) in order to evaluate how good is each compiler optimizing the code and how good is each platform in taking advantatge of such optimizations.

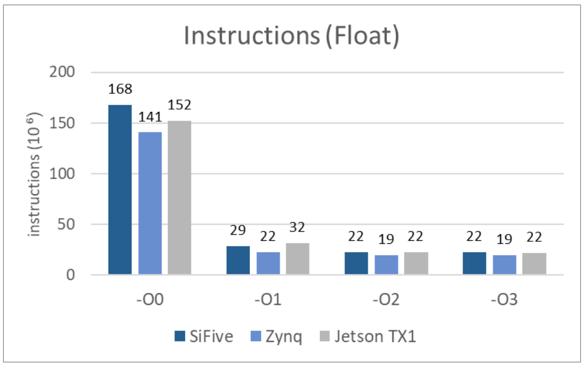
Theoretically, the 4 platforms we chose support Floating Point operations. In our experience, only 3 of the 4 actually supported it without any issue. The only platform we were unable test the floating point performance is Ariane. In the official website, it is mentioned that Ariane supports the RISC-V F extension and enabling the floating point support is as easy as changing a constant variable. The reality is that while the Floating Point unit is there, the code they provide only works in simulations and cannot be synthesized (Figure 5.7) into real hardware (FPGA).



```
yxu@server:$ make fpga
[...]
ERROR: [Synth 8-561] range expression could not be resolved to a constant [fpnew_pkg.vhd:593]
ERROR: [Synth 8-285] failed synthesizing module 'fp_i2fcasts' [fp_i2fcasts.vhd:78]
ERROR: [Synth 8-285] failed synthesizing module 'fp_conv_multi' [fp_conv_multi.vhd:74]
ERROR: [Synth 8-285] failed synthesizing module 'conv_multifmt_slice' [conv_multifmt_slice.vhd:9
ERROR: [Synth 8-285] failed synthesizing module 'conv_block' [conv_block.vhd:77]
ERROR: [Synth 8-285] failed synthesizing module 'fpnew' [fpnew.vhd:82]
ERROR: [Synth 8-285] failed synthesizing module 'fpnew_top' [fpnew_top.vhd:93]
ERROR: [Synth 8-6156] failed synthesizing module 'fpu_wrap' [fpu_wrap.sv:17]
[...]
---------------------------------------------------------------------------------
Finished RTL Elaboration
---------------------------------------------------------------------------------
RTL Elaboration failed
ERROR: [Vivado_Tcl 4-5] Elaboration failed - please see the console for details
```

Figure 5.7: Error when trying to enable Floating Point support in Ariane.

The first test we have done is to measure the performance executing the floating point single precision version of k-means. We intentionally left out Ariane in this comparison due to the errors mentioned above. Figure 5.8 shows the number of retired instructions. This comparison let us check any imbalance in the number of instructions generated by the compiler for each architecture. In this case, the number of instructions is pretty similar between the platforms for each optimization flag. On the other hand, Figure 5.9 shows the execution time for each case. As expected, the best performing platform is the Jetson TX1 which packs one of the best `Arm` cores today (Cortex A57). For the other `Arm` platform (Zynq 7020), it packs the Cortex A9, which it was a really good processor at that time and surprisingly, it has been outperformed by the SiFive HiFive Unleashed with a speedup of 1.65× in average.



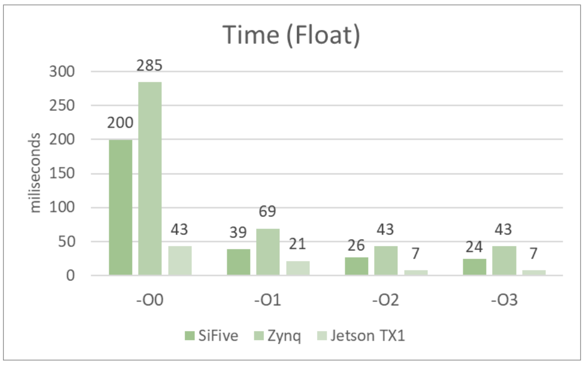
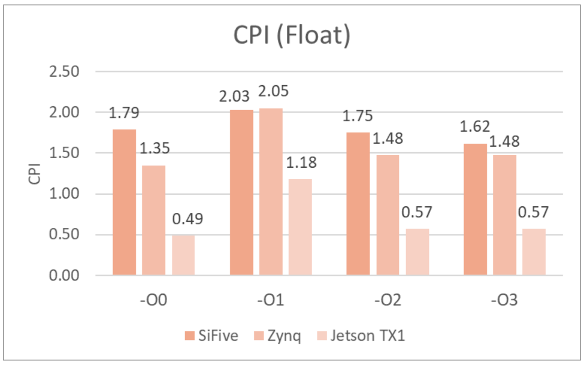


Figure 5.8: K-means (Float) retired instructions in single core cpu only.

Figure 5.9: K-means (Float) execution time in single core cpu only.

Even if the execution time is the typical metric used to measure how good is a platform, it does not give us a clear view of the actual microarchitecture design performance. For example, a platform with a higher frequency clock can potentially outperform another platform just because the frequency. For this reason, we also measured the Cycles Per Instruction (CPI). This metric does not consider the clock frequency but the actual number of cycles a processor spends in each instruction in average. Figure 5.10 shows the CPI for the previous case. Again Jetson TX1 has the lower CPI. For the two other platforms, the CPI in the Zynq is lower than in the SiFive in most of the cases. This means that part of the reason why the execution time is lower in SiFive is due to the microarchitecture's capability of running at a higher frequency (2, 25× higher).



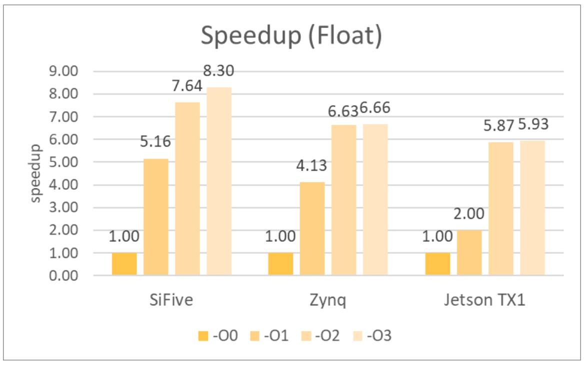


Figure 5.10: K-means (Float) CPI in single core cpu only.

Figure 5.11: K-means (Float) speedup between optimization flags on the same platform.

Figure 5.11 shows the speedup achieved in each platform using the optimizations flags provided by the compiler. It is important to notice that even if the CPI increased from `-O0` to `-O1` in all the platforms, the actual performance improvement is quite high (3.36× in average). Overall the `RISC-V` compiler achieved the higher improvement and among `Arm`, Zynq achieved a higher improvement than the Jetson TX1.

Ariane is one of the state of the art RISC-V 64-bit processors written in HDL available today. Comparing its performance to other available platforms is important. For this reason, we repeated the previous tests but using Integer data types. The main downside of using Integers rather than Floats is that we lose precision during our calculations, but for some use-cases, Integer precision is enough.

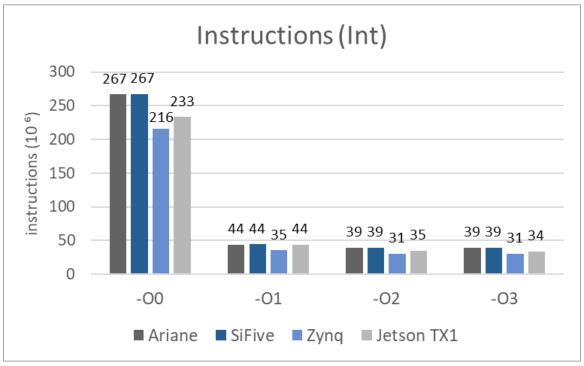


Figure 5.12: K-means (Int) retired instructions in single core cpu only.

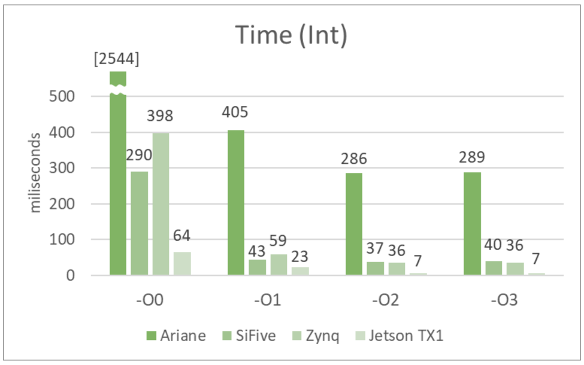


Figure 5.13: K-means (Int) execution time in single core cpu only.

At first glance, the number of retired instructions in Ariane is identical to the SiFive HiFive Unleashed, meaning that there is no load imbalance between both. This is the expected result because we are running the same binary. In the Arm case, the two platforms are different (32-bit vs 64-bit) and that is the reason why the number is not equal, however, they are similar and in the same order of magnitude. For the performance (Figure 5.9), Ariane shown to perform much worse than the SiFive (and the rest of platforms), being this latter 8.26× faster in average than Ariane. For the rest of platforms, the Jetson TX1 still the best performer, however, the SiFive is only better than the Zynq in the -O0 and -O1 cases, in the two other cases, the Zynq is 1.07× faster in average.

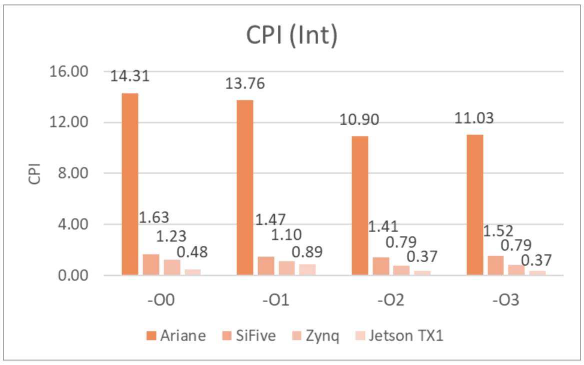


Figure 5.14: K-means (Int) CPI in single core cpu only.

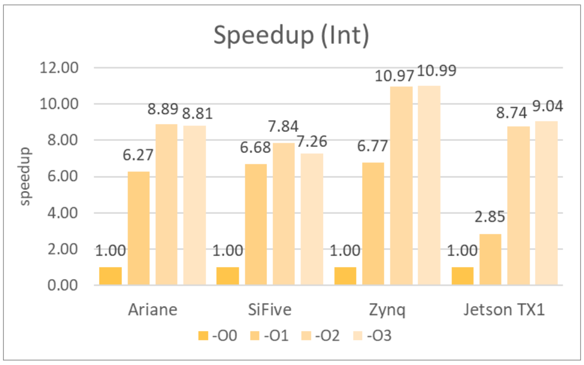


Figure 5.15: K-means (Int) speedup between optimization flags on the same platform.

Analyzing the CPI (Figure 5.14), Ariane is at 1 order of magnitude of difference. This clearly shows how much improvement still to be done in Ariane. As in the Floats case, the Jetson TX1 has the lower CPI followed by the Zynq and the SiFive. In this case, the difference between the Zynq and the SiFive is much higher than in the Floats for the -O2 and -O3 cases. This explains why the execution time of the Zynq is better than the SiFive in this same cases. Regarding the speedup achieved (Figure 5.15), the compilers did a great job with the optimization flags even in Ariane, which shows that the bad performance in Ariane is not directly related with the code generated by the compiler. For the rest, the platform that improved the most was the Zynq, achieving a 10.99× speedup in the best case. It is important to notice that in the RISC-V platforms, the performance decreased when passing from -O2 to -O3. Checking the retired instructions in these cases, we found out that using `-O3` actually reduced the instructions, but the CPI has increased, which means that the compiler reduced the number of instructions in favor of more complex ones, but in the end, did not improve the performance at all. As this happens in both `RISC-V` platforms, it seems like a problem directly related to the `-O3` optimizations applied by the GCC compiler in `RISC-V` .

Considering that the performance in Ariane was really bad, we tried to analyze in more detail the other available performance counters. During this process we found a bug related to the performance counters in Ariane and we already opened an issue with the corresponding fix[27]. With the bug fixed, we tried to measure other metrics like the L1 data cache misses, load instructions and store instructions. However, we found out that the reported L1 cache misses made no sense (*e.g.* more cache misses than load accesses). For the other two metrics, they were similar to the `Arm` executions, therefore we can assume that they are correct. We consider that analyzing the HDL code and fixing the bugs is out of the scope of this thesis. For this reason, we did not go further into the analysis of the HDL code and in the end, no conclusive result were drawn regarding why the performance was so bad in Ariane.

K-means kernels

In Section 4.1 we have seen that the k-means algorithm uses 3 different kernels: GEMM (Matrix Matrix Multiplication), AXPY ($Y = A \times X + Y$) and DOT (dot product). Table 5.5 shows the number of dynamic instructions of these kernels in one iteration of the previous k-means execution. This measurement is derived from the static instructions generated by the compiler. This is that the dynamic instructions are computed as instructions inside a loop multiplied by the number of iterations of the loop plus the static instructions.

	DOT	**AXPY**	**IGEMM**
Arm (armv7)	2618	3130	11617680
RISC-V (RV64)	2904	3553	14515738

Table 5.5: Dynamic code size of the k-means kernels.

From the 3 kernels, the more time consuming one is the GEMM (Matrix Multiplication) and because this kernel has a lot of relevance in scientific applications and machine learning algorithms, we decided to analyze the GEMM operation on our platforms. This will help other developers to have a reference on how their applications will work on the platforms we used, if the application makes use of the GEMM operation.

Following the same methodology as in k-means ($N = 128$, $K = 16$, $D = 128$), we tested both Integer and Floating Point versions of the GEMM kernel. Figure 5.16 shows the execution time in each platform. Overall, the performance trend is similar to the k-means tests. As expected, Ariane does not perform very well and the gap still of 1 order of magnitude. The Jetson TX1 is the best performing platform in both Interger and Floating Point versions. The SiFive performance is almost as good as the Zynq in the Integers version but the Floating Point performance in SiFive is clearly better than Zynq.

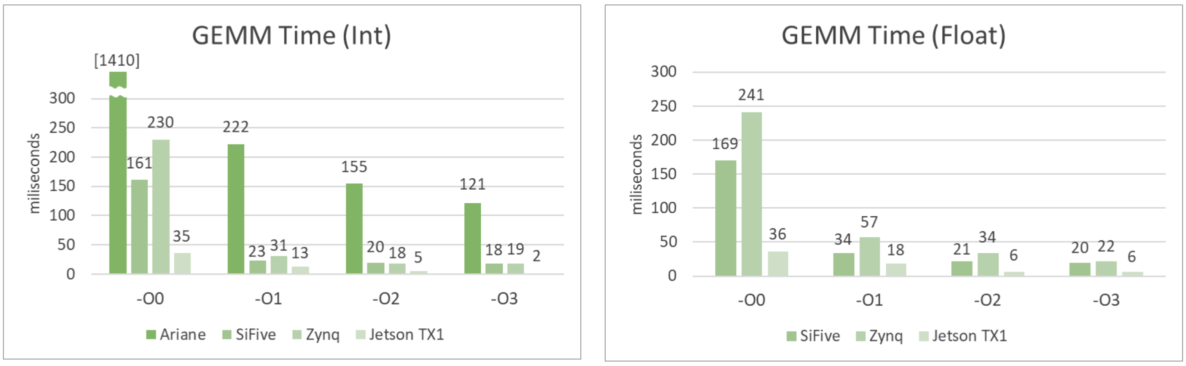


Figure 5.16: GEMM execution time in both Integer and Floating Point version.

For more metrics like the CPI and the Speedup on the GEMM kernel, check the Appendix H.

[27] `https://github.com/pulp-platform/ariane/issues/158`

5.3 CPU plus accelerator

In the previous section we focused in comparing the performance of two Arm and two RISC-V processors in a single core environment. However, the reality is that the Arm platforms have much more potential than the shown previously. In RISC-V the software ecosystem is not ready yet and furthermore, the platforms we tested do not have an accelerator. In this section we will focus only on the Arm platforms but with the idea of optimizing the application as much as the Arm software ecosystem let us and also take advantage of the accelerator embedded in the SoC.

5.3.1 Methodology

We followed a similar methodology as in the CPU only tests. We measured the performance using the PAPI library and executed the application in a Linux environment. The machine where the application run was exclusively allocated for our tests using SLURM. However, we will only use the `float` data type in order to be closer to a real world scenario.

For the dataset, we used a input file provided by Aingura IIoT which represents the same manufacturing process described in Section 5.2. Regarding the input parameters, as we executed the application in a multicore setup plus an accelerator, we used the following ones for all the tests: dimension $D = 1024$, number of elements to clusterize $N = 21500$, number of clusters $K = 8$, 1 repetition and an error tolerance of 10^{-4}. For choosing the initial centroids, we also used the k-means++ algorithm.

For accelerating the k-means kernels (GEMM, DOT, AXPY), we took advantatge of a BLAS implementation present in Arm. In this case, we used the ATLAS (Automatically Tuned Linear Algebra Software) library (version 3.10.3). Using this library let us achieve a huge improvement that will be discussed in the next subsections.

The two platforms we used in this test (Zynq 7020 and Jetson TX1) are considered heterogeneous platforms. This means that inside the same SoC, both platforms have two or more processing units with different characteristics. This difference typically comes from the ISA of each unit, but it also applies to the microarchitecture and the speed (*e.g.* Arm big.LITTLE). In the Jetson TX1 platform, there are 4 identical Arm Cortex A57 cores plus an Nvidia GPU with CUDA cores. The GPU do not follow the Arm ISA hence the platform is heterogeneous. In the same way, the Zynq 7020, packs 2 identical Arm Cortex A9 cores plus user programmable units (FPGA) and is also considered heterogeneous for this reason.

In order to run the code in multiple cores and handle the heterogeneity of the platforms, we used the OmpSs programming model. OmpSs extends OpenMP by adding new directives for supporting asynchronous parallelism and executions in heterogeneous devices (*e.g.* GPUs, FPGAs). This means that by using OmpSs, we can offload some workloads to run in parallel in the heterogeneous accelerators and in the main cores (*e.g.* Arm) at the same time. All the management related to the architectural difference of the devices is done by OmpSs, letting the programmers to focus on optimizing the code without caring about the (usually) cumbersome details of accelerator programming.

OmpSs programming is based on pragmas and this characteristic let us maintain a single portable and scalable code supporting multiple platforms. For running in a heterogeneous configuration, we only had to change few pragmas and recompile the application.

5.3.2 Serial Optimizations

In this section we will discuss about the advantatges of applying the optimizations explained in Section 4.1. At first we wanted to test the effect on both Arm and RISC-V architectures, but we limited it to only the Arm platforms due to the following reasons:

- The SiFive HiFive Unleashed board does not have some of the counters we are interested in (*e.g.* L1 Data Cache Miss[28].)
- Some of the performance counters in Ariane are not 100% reliable.

[28] Table 3 in https://static.dev.sifive.com/FU540-C000-v1.0.pdf

- The lack of a Vector unit in the analyzed RISC-V platforms.
- The non-availability of a BLAS implementation in RISC-V .

The optimizations discussed in Section 4.1 helped us to reduce the number computations, however, the most beneficial effect of the optimization is to reduce the original algorithm to simple computational kernels. Without isolating the kernels, none of the ATLAS library routines could have been used.

Table 5.6 presents the data locality improvement achieved with the ATLAS libary. This comparison shows the total number of L1 data cache misses of a k-means execution for both platforms in a single core configuration. The improvement is noticeable, as the number of data cache misses in the optimized version decreases 1.56× in Jetson TX1 and 2.49× in Zynq 7020.

The new implementation strategy ensures that the coordinates of the points are stored consecutively in memory. For this reason, SIMD instructions can be used. Besides the L1 data cache misses, Table 5.6 shows the floating point and vector operations executed in a k-means execution. In the Jetson TX1 case, the vector operations represents the 99% of the total floating point operations, meaning that a huge level of data parallelism has been exploited. On the other hand, due to the fact that Armv7 NEON hardware does not fully implement the IEEE 754 standard for floating-point arithmetic [47] (*e.g.* the direct comparison of single-precision values, used by our algorithm), no SIMD operations have been used for the Zynq 7020 execution.

	Jetson TX1			**Zynq 7020**		
	L1 DCM	FP	VEC	L1 DCM	FP	VEC
Original	$4.3 \cdot 10^8$	$1.5 \cdot 10^{10}$	~ 0	$8.9 \cdot 10^8$	$3.2 \cdot 10^{10}$	~ 0
Optimized	$2.7 \cdot 10^8$	$3.9 \cdot 10^7$	$5.7 \cdot 10^9$	$3.6 \cdot 10^8$	$1.3 \cdot 10^{10}$	~ 0

Table 5.6: Comparison of figures of merit related to the reference implementation and the one based on matrix operations. Parameters considered are L1 Data Cache Misses (DCM), Floating Point operations (FP) and Vectorial operations (VEC).

5.3.3 Parallel Implementations

In Section 5.2, we implemented a basic k-means algorithm following the optimizations described in Section 4.1, but due to the limited software ecosystem in RISC-V, we limited our tests to run only in a single core. In the current tests, we took Section 5.2's code as baseline and we modified it to take advantage of the multiple cores and the accelerator available in the Arm platforms.

```
A = dot_product(DATA)
CENTROIDS = kpp(DATA)
do {
    B = dot_product(CENTROIDS)
    for each block i {
        #pragma omp target device(fpga,smp,cuda) copy_deps
        #pragma omp task in(DATA,CENTROIDS) out(Ci)
        Ci = MxM(DATA,CENTROIDS)
        #pragma omp task in(A,B,Ci) out(CENTROIDS,LABELS)
            <CENTROIDS,LABELS,error> = compute_centroids( A, B, Ci)
            #pragma omp atomic
            total_error+=error;
    }
    #pragma omp taskwait
} while (total_error > tolerance);
```

Figure 5.17: K-means pseudocode with OmpSs pragmas with support for SMP, FPGA and GPU execution.

As mentioned before, the OmpSs programming model has been used not only for parallelizing the code but also to handle the heterogeneous devices in a transparent way to the user. Figure 5.17 shows in a pseudocode how by just adding few pragmas within the almost same code, OmpSs is capable of managing three different devices: SMP (Symmetric Multiprocessor), FPGA and GPU. Our heterogeneous k-means implementations follow this approach and leverages in OmpSs to exploit the hardware accelerators.

The **OmpSs** version is the parallelized version of the sequential code. In this case, the parallel granularity is expressed at point level. This means that a parallel task computes the dot product and the matrix multiply of a continuous subset of points (domain decomposition).

The **OmpSs+BLAS** version is identical to the OmpSs version but adapting the dot product and matrix multiply operations to be done using the ATLAS (BLAS) library.

The **OmpSs@CUDA+BLAS** version uses as baseline the OmpSs+BLAS version but combines the power of an NVIDIA GPU accelerator. The OmpSs@CUDA ecosystem manages all the data transfers between the GPU and the host in a transparent way. The OmpSs runtime scheduler will optimize the data transfers by analyzing the data dependencies and moving data only when it is necessary.

The **OmpSs@FPGA+BLAS** version uses as baseline the OmpSs+BLAS version but takes advantage of the programmable logic of the board. In this case, The OmpSs@FPGA ecosystem has generated a Matrix Matrix Multiplication (GEMM) accelerator that is programmed into the FPGA. The data transfers between the FPGA and the host are also managed by the runtime. The percentage of hardware resources reported by Vivado is: 32% BRAMs, 80% DSPs, 26% FFs and 62% LUTs.

5.3.4 Evaluation

In this section we will evaluate the platforms from two points of view, the pure performance and the energy to solution.

Performance

Figure 5.18 shows the speedup achieved when we executed the different implementations on both platforms. We took as reference a CPU-only serial implementation. By just annotating the code with few OmpSs pragmas and execute it in parallel, the performance improvement in both cases (Jetson TX1 and Zynq 7020) is quite good (3.13× and 1.58× respectively). The version that combines OmpSs plus ATLAS (BLAS), achieves a performance boost up to 7.39× and 6.20× respectively. Combining the CPU plus the accelerator embedded in the SoC (GPU or FPGA), the performance in the Jetson TX1 (GPU) is worse than only using the CPU. This is directly related to the fact that the dimension of the used input set is too small to benefit from the embedded accelerator. However, the performance in the Zynq 7020 (FPGA) increased up to 7.76× compared to the serial version. This shows that for certain real case scenarios (as this industrial one), using a GPU is not always the best way to achieve performance.

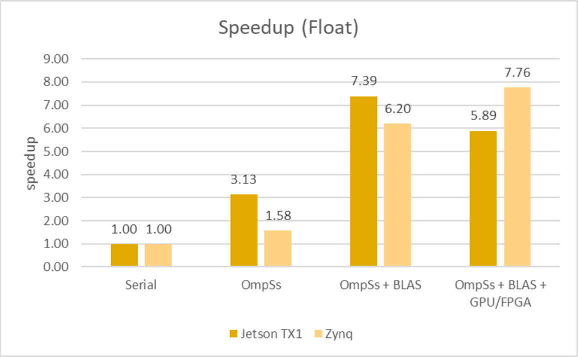


Figure 5.18: Speedup between different implementations on Jetson TX1 and Zynq 7020.

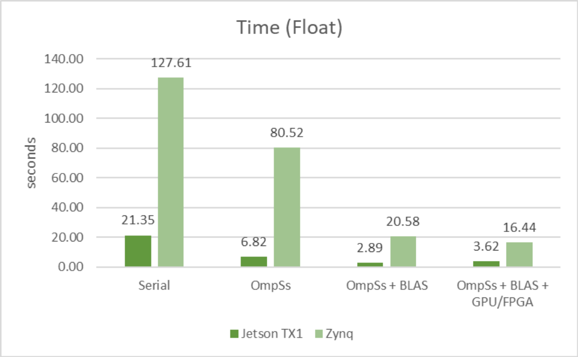


Figure 5.19: Execution time between different implementations on Jetson TX1 and Zynq 7020.

The Figure 5.19 shows the same results as the Figure 5.18 but quantifying the performance in seconds instead of the achieved speedup between implementations. As expected, the Zynq 7020 performs worse than the Jetson TX1. In the serial execution, the Jetson TX1 is 6× faster than Zynq 7020 while in the execution using OmpSs plus BLAS plus the accelerator Jetson TX1 is only 4.5× faster. Even if the speedup is greater on the Zynq 7020, as expected, the raw performance is much better in the Jetson TX1, as the Jetson TX1 features the double of cores clocked to a higher frequency and a much more refined architecture (Armv8 vs Armv7).

Energy to solution

Embedded `Arm` processors are well known for its power consumption. In this section we will discuss the power profile and the total energy to reach the solution of the problem (called *Energy to Solution*) on both platforms. The power data has been collected using a Yokogawa power meter [48]. The energy to solution has been computed as the sum over execution time of the instantaneous power. The evaluated implementations are the ones using an accelerator (GPU / FPGA).

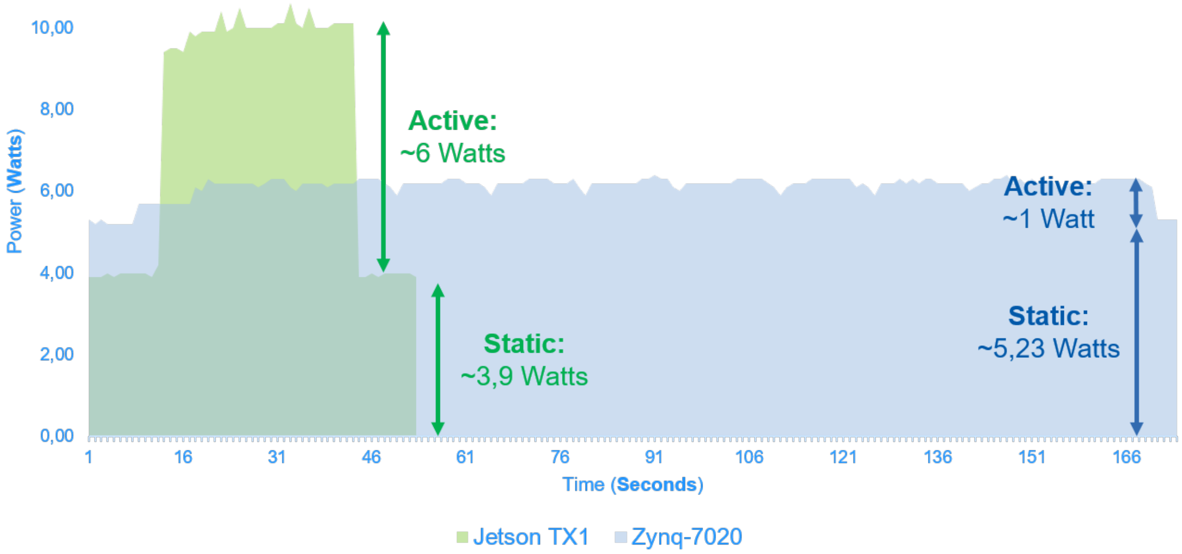


Figure 5.20: Power consumption of 10 k-means repetitions in the `Arm` platforms.

Both platforms have been tested using the input set mentioned in the methodology of this section but computing 10 iterations instead of only one for power sampling reasons (Figure 5.20). The Jetson TX1 board consumed 307.54 J in total. 181.43 J of them are given by the contribution of active power, *i.e.*, ignoring static idle power. The Zynq 7020 consumed 1009.72 J in total and only 157.99 J was the contribution of the active part.

The total energy of Jetson TX1 is 3× lower than the Zynq 7020. This is mostly due to the fact that the first one has an idle power of 3.9 W while in the second platform is 5.23 W. The lower idle power combined with the fact that the execution time is 4.5× faster in the Jetson TX1, makes the total energy consumed by this platform significantly lower. However, considering only the active energy, the Zynq 7020 consumes 15% less energy than Jetson TX1 even if the lithography process in Zynq 7020 is 28 nm and in Jetson TX1 is 20 nm.

We observe that the majority of the active power is currently used by the `Arm` cores: the impact of the power consumption of the FPGA is below 1 W, but its performance benefits are notable. From all these observations, a platform with newer `Arm` cores like the Cortex A57 plus an FPGA would probably outperform the Jetson TX1 in both, pure performance and power consumption in the Aingura IIoT case scenario.

Chapter 6

Conclusions

From a real industrial case scenario presented by Aingura IIoT and due to the limitations of the industrial environment, we considered the two most popular RISC architectures available today (Arm and RISC-V). Starting from the technical specifications, we took the journey of analyzing the HPC capability of each architecture at the current state.

RISC-V presence is growing really fast, although the HPC support is not ready yet, the idea behind an open ISA is really interesting. By far, the ISA is the most important interface in a computer system. It is the middle point where software meets hardware. There are open standards for almost every other interface (for example: TCP/IP for networking, OpenGL for graphics, etc) and they worked really well so we believe in the RISC-V proposal. However, as shown in our analysis, the support in HPC still not mature enough. The lack of a frozen vector extension makes RISC-V to be a step down compared to the competitors, however the community is putting a lot of effort in this direction and a lot of progress is expected to happen in the short term. In our test, we included two RISC-V platforms: the SiFive HiFive Unleashed, representing the best comercial option, and Ariane, representing the best research focused option today.

In our tests, the SiFive HiFive Unleashed performed really well, achieving a speedup of 1.65× in Floating Point performance and the same Integer performance compared to the Arm Cortex A9. This is really impressive considering the short life-time of the SiFive company (2015). We cannot say the same for the Ariane platform. In this case, the execution time was at 1 order of magnitude of difference compared to the other platforms, being the SiFive 8.26× faster in average than Ariane. While this result shocked us, it is important to remember that Ariane still an under development platform (*e.g.* real Atomic support is still pending) and the current focus is more in the correctness direction rather than achieving the best performance. However, it important to remember that we detected an inconsistent behaviour with some of the performance counters and we even found a bug on the counters unit. Although the measured cycles and the retired instructions follow a consistent behaviour, due to the other issues, we are taking the performance numbers of Ariane with a grain of salt. On the other hand, a very important milestone in the Ariane development progress is the booting Linux capability. While in our tests we found that running applications in Linux for Ariane is not reliable, this issue will be solved once the multicore support is added. Working in a Linux environment has a lot of advantatges, for example the pthreads (POSIX Threads) support which opens the posibility of executing parallel programs. For these reasons, we think that Ariane is an up-and-coming platform with a lot of potentials behind.

From our tests, it is very clear that the RISC-V platforms performance is not at the level of the current state-of-the-art Arm chips. The performance of the Cortex A57 is 3.43× faster than SiFive, but that is totally expected. The important point of the comparison was to check how far RISC-V processors are compared to a processor from a big and very experienced company such as Arm. Considering only the Arm platforms we analized, we can easily conclude that the Arm architecture found its way into the HPC market. The software ecosystem in Arm got to a point where executing applications is as easy as in traditional x86 platforms. With the Mont-Blanc software stack, we were able to run scientific workloads and tackle real industrial needs like the Aingura IIoT case scenario. The most interesting part of the analyzed Arm platforms is the fact that they are heterogeneous, packing Arm cores plus an accelerator (FPGA or GPU). From the sequential code used in the RISC-V tests, we adapted it to be compatible with three different computing resources at the same time: SMP, GPU and FPGA. Our tests shown that by optimizing the code and using the correct software, we can achieve a speedup of 5.9× and 7.8× in the Zynq 7020 and the Jetson TX1 respectively.

Thanks to the advanced OmpSs programming model, adapting the applications to the different devices was less painful. Mainly because we did not have to care about the interaction between the `Arm` cores and the accelerators. OmpSs handles that allowing us to focus on optimizing the code without caring about the details of accelerator programming. This shows how important is to have a good software support in a platform. On the other hand, the compilers also play a really important role on the performance. With a very optimized compiler as the `Arm` HPC compiler, in our tests the performance was 1.12× faster than GCC. Using the optimization flags provided by the own compiler, we achieved an effortless speedup of 10× in some cases. It is important to mention that although the compiler performance in `RISC-V` is not as good as in `Arm`, this is expected as the `RISC-V` architecture is relatively new and the efforts of the community are focusing in the correctness rather than tuning the generated code.

To conclude, we want to remark that both `Arm` or `RISC-V` are simply ISAs. If a processor is `Arm` or `RISC-V`, does not guarantee us to be really energy efficient or to perform really well. A clear example of this is Ariane, which performs much worse than the SiFive option and both are `RISC-V`. Instead, the important part when comparing ISAs is the opportunities they offer to the designers to come up with a better implementation. Regarding the `RISC-V` ISA, it is improving really fast. The embedded world already started shifting from `Arm` to `RISC-V` microcontrollers in commercial products (*e.g.* Nvidia Falcon microcontroller) but for the HPC world the support is not ready yet. Heterogeneous SoCs like the ones we used in this thesis (Zynq 7020 and Jetson TX1) demonstrated the huge potential behind these type of configurations. Achieving a significant performance improvement without sacrificing the power consumption. Considering the `RISC-V` modular approach, we see a lot of potential on this ISA to also fit in heterogeneous platforms too.

Chapter 7

Future work

This chapter covers the future work of our research. Firstly, we plan to also measure the power consumption in the SiFive HiFive Unleashed and compare it to the Jetson TX1 and the Zynq 7020. This would allow us to calculate other metrics like the performance per watt and compare how the fastest RISC-V platform today compares against the competition.

Secondly, we will continue the Ariane research line. We plan to double check that the performance counters are working properly and contribute in the project if any bug is found. In case the read cycles were correct, we will analyze why the performance is so bad and see where is the bottleneck.

As Ariane is a project in continue development, we plan to repeat the evaluation once a synthesizable FPU is released. In the same way, once the multicore support is ready, we plan to repeat the tests in the 4 platforms using a k-means version ported to OpenMP.

Finally, taking advantatge of the modularity of RISC-V, we plan to test a k-means version using SIMD operations in a RISC-V platform. To achieve that, we plan to use as baseline the Ariane core and we will design in SystemVerilog a vector unit that will integrate with the core. This unit will follow a simple custom ISA with an encoding defined within the custom extensions space mentioned in the RISC-V specification.

Appendices

A ariane.cfg

```
#
# Olimex ARM-USB-OCD-H
#
# http://www.olimex.com/dev/arm-usb-ocd-h.html
#

adapter_khz     1000

interface ftdi
ftdi_device_desc "Olimex OpenOCD JTAG ARM-USB-OCD-H"
ftdi_vid_pid 0x15ba 0x002b

ftdi_layout_init 0x0908 0x0b1b
ftdi_layout_signal nSRST -oe 0x0200
ftdi_layout_signal nTRST -data 0x0100 -oe 0x0100
ftdi_layout_signal LED -data 0x0800

#
# Ariane TAP
#

set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME

gdb_report_data_abort enable
gdb_report_register_access_error enable

riscv set_reset_timeout_sec 120
riscv set_command_timeout_sec 120

# prefer to use sba for system bus access
riscv set_prefer_sba off
riscv expose_csrs 3071-3086

init
halt
echo "Ready for Remote Connections"
```

B Objectdump of `add.exe`

```
0000000080000000 <add>:
    80000000:   1101                    addi    sp,sp,-32
    80000002:   ec22                    sd      s0,24(sp)
    80000004:   1000                    addi    s0,sp,32
    80000006:   87aa                    mv      a5,a0
    80000008:   872e                    mv      a4,a1
    8000000a:   fef42623                sw      a5,-20(s0)
    8000000e:   87ba                    mv      a5,a4
    80000010:   fef42423                sw      a5,-24(s0)
    80000014:   fec42703                lw      a4,-20(s0)
    80000018:   fe842783                lw      a5,-24(s0)
    8000001c:   9fb9                    addw    a5,a5,a4
    8000001e:   2781                    sext.w  a5,a5
    80000020:   853e                    mv      a0,a5
    80000022:   6462                    ld      s0,24(sp)
    80000024:   6105                    addi    sp,sp,32
    80000026:   8082                    ret

0000000080000028 <_start>:
    80000028:   1101                    addi    sp,sp,-32
    8000002a:   ec06                    sd      ra,24(sp)
    8000002c:   e822                    sd      s0,16(sp)
    8000002e:   1000                    addi    s0,sp,32
    80000030:   4589                    li      a1,2
    80000032:   4505                    li      a0,1
    80000034:   fcdff0ef                jal     ra,80000000 <add>
    80000038:   87aa                    mv      a5,a0
    8000003a:   fef42623                sw      a5,-20(s0)
    8000003e:   fec42783                lw      a5,-20(s0)
    80000042:   bff5                    j       8000003e <_start+0x16>
```

C Ariane Device Tree

```
/dts-v1/;

/ {
  #address-cells = <2>;
  #size-cells = <2>;
  compatible = "eth,ariane-bare-dev";
  model = "eth,ariane-bare";
  // chosen {
  //   stdout-path = "/soc/uart@10000000:115200";
  // };
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    timebase-frequency = <32768>; // 32.768 kHz
    CPU0: cpu@0 {
      clock-frequency = <50000000>; // 50 MHz
      device_type = "cpu";
      reg = <0>;
      status = "okay";
      compatible = "eth, ariane", "riscv";
      riscv,isa = "rv64imacsu";
      mmu-type = "riscv,sv39";
      tlb-split;
      // HLIC - hart local interrupt controller
      CPU0_intc: interrupt-controller {
        #interrupt-cells = <1>;
        interrupt-controller;
        compatible = "riscv,cpu-intc";
      };
    };
  };
  memory@80000000 {
    device_type = "memory";
    reg = <0x0 0x80000000 0x0 0x1800000>;
  };
  soc {
    #address-cells = <2>;
    #size-cells = <2>;
    compatible = "eth,ariane-bare-soc", "simple-bus";
    ranges;
    clint@2000000 {
      compatible = "riscv,clint0";
      interrupts-extended = <&CPU0_intc 3 &CPU0_intc 7>;
      reg = <0x0 0x2000000 0x0 0xc0000>;
      reg-names = "control";
    };
    PLIC0: interrupt-controller@c000000 {
      #address-cells = <0>;
      #interrupt-cells = <1>;
      compatible = "sifive,plic-1.0.0", "riscv,plic0";
      interrupt-controller;
      interrupts-extended = <&CPU0_intc 11 &CPU0_intc 9>;
      reg = <0x0 0xc000000 0x0 0x4000000>;
      riscv,max-priority = <7>;
      riscv,ndev = <2>;
    };
    debug-controller@0 {
      compatible = "riscv,debug-013";
      interrupts-extended = <&CPU0_intc 65535>;
      reg = <0x0 0x0 0x0 0x1000>;
      reg-names = "control";
    };
    uart@10000000 {
      compatible = "ns16750";
      reg = <0x0 0x10000000 0x0 0x1000>;
      clock-frequency = <50000000>;
      current-speed = <115200>;
      interrupt-parent = <&PLIC0>;
      interrupts = <1>;
      reg-shift = <2>; // regs are spaced on 32 bit boundary
      reg-io-width = <4>; // only 32-bit access are supported
    };
  };
};
```

D Kernel configuration file (**buildroot_defconfig**)

```
BR2_riscv=y
BR2_riscv_custom=y
BR2_RISCV_ISA_CUSTOM_RVM=y
BR2_RISCV_ISA_CUSTOM_RVC=y
BR2_CCACHE=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_PATH="$(RISCV)"
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_PREFIX="$(ARCH)-unknown-linux-gnu"
BR2_TOOLCHAIN_EXTERNAL_GCC_8=y
BR2_TOOLCHAIN_EXTERNAL_HEADERS_4_13=y
BR2_TOOLCHAIN_EXTERNAL_CUSTOM_GLIBC=y
BR2_TOOLCHAIN_EXTERNAL_CXX=y
BR2_ROOTFS_OVERLAY="../rootfs"
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_CUSTOM_VERSION=y
BR2_LINUX_KERNEL_CUSTOM_VERSION_VALUE="4.20-rc2"
BR2_LINUX_KERNEL_PATCH="../configs/0001-Add-RISC-V-architecture-to-Xilinx-ethernet-Kconfig.patch ../configs/0002
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="../configs/linux_defconfig"
BR2_PACKAGE_BUSYBOX_CONFIG="../configs/busybox.config"
BR2_PACKAGE_DHRYSTONE=y
BR2_PACKAGE_MEMSTAT=y
BR2_PACKAGE_RAMSPEED=y
BR2_PACKAGE_STRACE=y
BR2_PACKAGE_STRESS=y
BR2_PACKAGE_STRESS_NG=y
BR2_PACKAGE_TRACE_CMD=y
# BR2_PACKAGE_IFUPDOWN_SCRIPTS is not set
BR2_PACKAGE_HTOP=y
BR2_PACKAGE_NANO=y
BR2_TARGET_ROOTFS_CPIO_GZIP=y
BR2_TARGET_ROOTFS_INITRAMFS=y
# BR2_TARGET_ROOTFS_TAR is not set
```

E PAPI library functions

```
int events[2] = {PAPI_TOT_INS, PAPI_TOT_CYC}, papi_ret;
long unsigned papi_c1, papi_c2;
void papi_check() {
    long_long values[2];
    if (PAPI_num_counters() < 2) {
        fprintf(stderr, "No hardware counters here, or PAPI not supported.\n");
        exit(1);
    }
}

void papi_start() {
    if ((papi_ret = PAPI_start_counters(events, 2)) != PAPI_OK) {
        fprintf(stderr, "PAPI failed to start counters: %s\n", PAPI_strerror(papi_ret));
        exit(1);
    }
}

void papi_read() {
    long_long values[2];
    if ((papi_ret = PAPI_stop_counters(values, 2)) != PAPI_OK) {
        fprintf(stderr, "PAPI failed to read counters: %s\n", PAPI_strerror(papi_ret));
        exit(1);
    }
    papi_c1 = (long unsigned) values[0];
    papi_c2 = (long unsigned) values[1];
    printf("PAPI_TOT_INS: %lu\n", papi_c1);
    printf("PAPI_TOT_CYC: %lu\n", papi_c2);
}
```

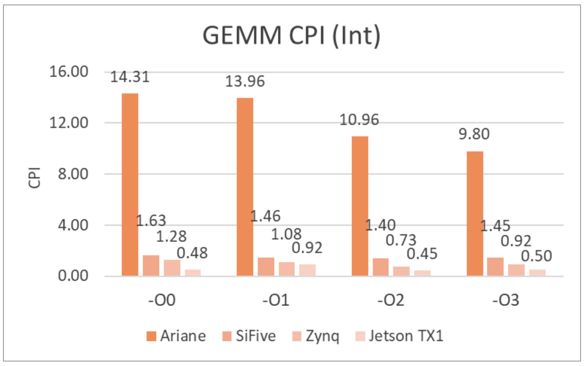
F Read CSV functions

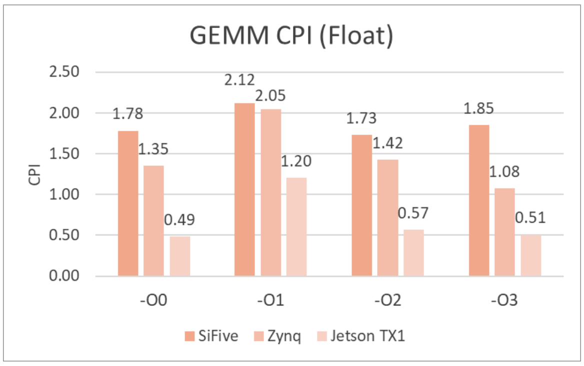
```
#include <stdlib.h>     /* atoi */
void* getline_from_pointer (DATA_TYPE* line, char* file, char separator, int nelem) {
    char buff[256];
    int n = 0;
    while ((*file != ’\n’) && (n < nelem)) {
        char* q = buff;
        while ((*file != separator) && (*file != ’\n’)) {
            *q = *file;
            q++;
            file++;
        };
        n++;
        *q = ’\0’;
        *line = atoi(buff);
        line++;
        file++;
    }
    file++;
    return file;
}
void read_csv(DATA_TYPE* q, char* file_p, int m, int n) {
    for (int j = 0; j < m; j++) {
        file_p = getline_from_pointer (q, file_p, ’;’, n);
        q += n;
    }
}
```

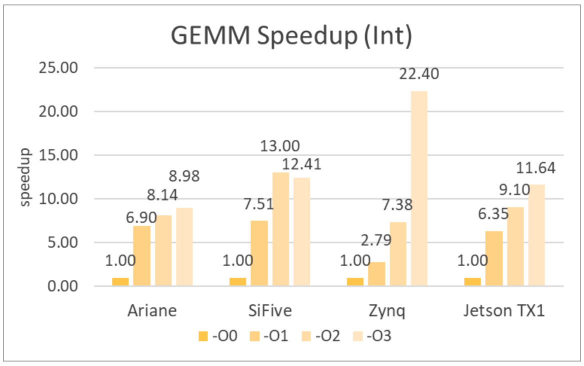
G Bare-Metal k-means Makefile

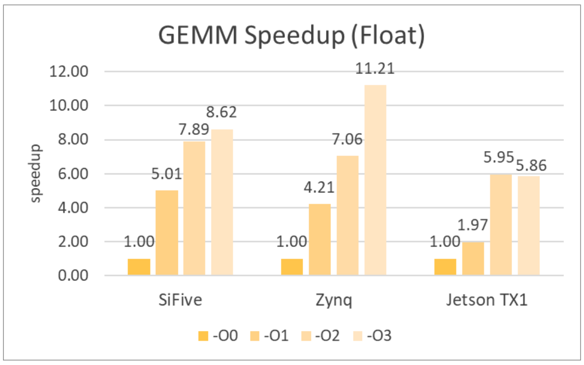
```
CC      = riscv64-unknown-elf-
CFLAGS  = "-mcmodel=medany -march=rv64imac -mabi=lp64"
LDFLAGS = "-mcmodel=medany -march=rv64imac -mabi=lp64 -Wl,--section-start=.text=0x80000000"
ASFLAGS = "-march=rv64imac -mabi=lp64"

kmeans : kmeans.o input.o utils.o
    $(CC)gcc $(LDFLAGS) -o kmeans kmeans.o input.o utils.o -static

input.o: input.s
    $(CC)as -o input.o input.s $(ASFLAGS)

kmeans.o: kmeans.c
    $(CC)gcc $(CFLAGS) -c kmeans.c -g

utils.o: utils.c
    $(CC)gcc $(CFLAGS) -c utils.c -g

.PHONY: clean
clean:
    rm -vf kmeans kmeans_papi *.o *.bin *.exe *.gch
```

H GEMM Kernel figures of merit









Bibliography

[1] Banchelli, F., Ruiz, D., Xu, Y., Mantovani, F.: Is Arm software ecosystem ready for HPC? In: SC17: International Conference for High Performance Computing, Networking, Storage and Analysis. (2017)

[2] Xu, Y., Vidal, M., Arejita, B., Díaz, J., Álvarez, C., Jiménez, D., Martorell, X., Mantovani, F.: Implementation of the K-means Algorithm on Heterogeneous Devices: a Use Case Based on an Industrial Dataset. In: Parallel Computing is Everywhere (serie: Advances in Parallel Computing), IOS Press (2018) 642--651

[3] Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P., Tomov, S.: The impact of multicore on math software. In: International Workshop on Applied Parallel Computing, Springer (2006) 1--10

[4] Wikipedia contributors: Moore's law --- Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=873729534 (2018) [Online; accessed 16-December-2018].

[5] Beard, J., Rusitoru, R.: MOMENTUM IS BUILDING FOR ARM IN HPC. URL: https://www.nextplatform.com/2017/06/30/momentum-building-arm-hpc/ (2017)

[6] Morgan, T.P.: DETAILS EMERGE ON POST-K EXASCALE SYSTEM WITH FIRST PROTOTYPE. URL: https://www.nextplatform.com/2018/06/21/details-emerge-on-post-k-exascale-system-with-first-prototype (2018)

[7] Sandia National Laboratories: Astra supercomputer at Sandia Labs is fastest Arm-based machine on TOP500 list. URL: https://share-ng.sandia.gov/news/resources/news_releases/top_500

[8] Rajovic, N., Rico, A., Puzovic, N., Adeniyi-Jones, C., Ramirez, A.: Tibidabo1: Making the case for an arm-based hpc system. Future Generation Computer Systems **36** (2014) 322--334

[9] Rajovic, N., Carpenter, P.M., Gelado, I., Puzovic, N., Ramirez, A., Valero, M.: Supercomputing with commodity cpus: Are mobile socs ready for hpc? In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM (2013) 40

[10] Rajovic, N., et al.: The Mont-blanc Prototype: An Alternative Approach for HPC Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '16, IEEE Press (2016) 38:1--38:12

[11] Calore, E., Mantovani, F., Ruiz, D.: Advanced performance analysis of hpc workloads on cavium thunderx. In: 2018 International Conference on High Performance Computing & Simulation (HPCS), IEEE (2018) 375--382

[12] Garcia-Gasulla, M., Josep-Fabrego, M., Eguzkitza, B., Mantovani, F.: Computational fluid and particle dynamics simulations for respiratory system: Runtime optimization on an arm cluster. In: ICPP'18 Proceedings of the 47th International Conference on Parallel Processing Companion, Association for Computing Machinery (ACM) (2018)

[13] Mazumdar, S., Ayguade, E., Bettin, N., Bueno, J., Ermini, S., Filgueras, A., Jimenez-Gonzalez, D., Martinez, C.A., Martorell, X., Montefoschi, F., et al.: Axiom: a hardware-software platform for cyber physical systems. In: 2016 Euromicro Conference on Digital System Design (DSD), IEEE (2016) 539--546

[14] Asanović, K., Patterson, D.A.: Instruction sets should be free: The case for risc-v. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146 (2014)

[15] Asanovic, K., Patterson, D.A., Celio, C.: The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical report, University of California at Berkeley Berkeley United States (2015)

[16] Traber, A., Zaruba, F., Stucki, S., Pullini, A., Haugou, G., Flamand, E., Gurkaynak, F.K., Benini, L.: Pulpino: A small single-core risc-v soc. In: 3rd RISCV Workshop. (2016)

[17] Gautschi, M., Schiavone, P.D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F.K., Benini, L.: Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(10) (2017) 2700--2713

[18] Cheikh, A., Cerutti, G., Mastrandrea, A., Menichelli, F., Olivieri, M.: The microarchitecture of a multi-threaded risc-v compliant processing core family for iot end-nodes. arXiv preprint arXiv:1712.04902 (2017)

[19] Kurth, A., Vogel, P., Capotondi, A., Marongiu, A., Benini, L.: Hero: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA. CoRR **abs/1712.06497** (2017)

[20] Matt Cockrell: Evaluation of RISC-V for Pixel Visual Core. URL: https://content.riscv.org/wp-content/uploads/2018/05/13.15-13.30-matt-Cockrell.pdf (2018)

[21] Steinbach, M., Karypis, G., Kumar, V., et al.: A comparison of document clustering techniques. In: KDD workshop on text mining. Volume 400., Boston (2000) 525--526

[22] Brettel, M., Friederichsen, N., Keller, M., Rosenberg, M.: How virtualization, decentralization and network building change the manufacturing landscape: An industry 4.0 perspective. International Journal of Mechanical, Industrial Science and Engineering **8**(1) (2014) 37--44

[23] Lee, J., Kao, H.A., Yang, S.: Service innovation and smart analytics for industry 4.0 and big data environment. Procedia Cirp **16** (2014) 3--8

[24] Montealegre, M.A., Arejita, B., Alvarez, P., Laorden, C., Diaz-Rozo, J.: Control quality on process of laser heat treatment. In: THERMEC 2018. Volume 941 of Materials Science Forum., Trans Tech Publications (1 2019) 1860--1866

[25] Diaz-Rozo, J., Bielza, C., Larrañaga, P.: Machine learning-based cps for clustering high throughput machining cycle conditions. Procedia Manufacturing **10** (2017) 997--1008

[26] Larrañaga, P., Atienza, D., Diaz-Rozo, J., Ogbechie, A., Puerto-Santana, C.E., Bielza, C.: Industrial Applications of Machine Learning. CRC Press (2018)

[27] Guitart, J.: Toward sustainable data centers: a comprehensive energy management strategy. Computing **99**(6) (Jun 2017) 597--615

[28] Barroso, L.A., Clidaras, J., Hölzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition. Synthesis Lectures on Computer Architecture **8**(3) (July 2013) 1--154

[29] Mantovani, F., Calore, E.: Performance and power analysis of hpc workloads on heterogeneous multi-node clusters. Journal of Low Power Electronics and Applications **8**(2) (2018)

[30] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., et al.: The ARM scalable vector extension. IEEE Micro **37**(2) (2017) 26--39

[31] : New – EC2 Instances (A1) Powered by Arm-Based AWS Graviton Processors. URL: https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors (2019)

[32] : Huawei Unveils "Industry's Highest-Performance ARM-based CPU". URL: https://insidehpc.com/2019/01/huawei-unveils-industrys-highest-performance-arm-based-cpu (2019)

[33] Rick O'Connor: RISC-V ISA & Foundation Overview. URL: https://content.riscv.org/wp-content/uploads/2018/05/13.00-13.15-RISC-V-ISA-Foundation-Overview-Barcelona-7May2018-1.pdf (2018)

[34] Patterson, D.A., Sequin, C.H.: RISC I: A Reduced Instruction Set VLSI Computer. In: Proceedings of the 8th Annual Symposium on Computer Architecture. ISCA '81, Los Alamitos, CA, USA, IEEE Computer Society Press (1981) 443--457

[35] Katevenis, M.G., Sherburne, Jr., R.W., Patterson, D.A., Séquin, C.H.: The RISC II Micro-architecture. Adv. VLSI Comput. Syst. **1**(2) (October 1984) 138--152

[36] Samples, A.D., Klein, M., Foley, P.: SOAR Architecture. Technical Report UCB/CSD-85-226, EECS Department, University of California, Berkeley (1985)

[37] Hill, M., J. Eggers, S., Larus, J., S. Taylor, G., Adams, G., K. Bose, B., Gibson, G., Hansen, P., Keller, J., Kong, S., Lee, C., Lee, D., Pendleton, J., Ritchie, S., Wood, D., Zorn, B., Hilfinger, P., Hodges, D., Katz, R., Patterson, D.: Design Decisions in SPUR. Computer **19** (11 1986) 8--22

[38] Palmer Dabbelt: All Aboard, Part 6: Booting a RISC-V Linux Kernel. URL: https://www.sifive.com/blog/all-aboard-part-6-booting-a-risc-v-linux-kernel (2017)

[39] Jain, A.K.: Data clustering: 50 years beyond k-means. Pattern recognition letters **31**(8) (2010) 651--666

[40] Singh, D., Reddy, C.K.: A survey on platforms for big data analytics. Journal of Big Data **2**(1) (2015) 8

[41] Javier Diaz-Rozo, Concha Bielza, P.L.: Machine learning-based cps for clustering high throughput machining cycle conditions. Procedia Technology (2017)

[42] Bekkerman, R., Bilenko, M., Langford, J.: Scaling up machine learning: Parallel and distributed approaches. Cambridge University Press (2011)

[43] Arthur, D., Vassilvitskii, S.: K-means++: The advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '07 (2007) 1027--1035

[44] Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(02) (2011) 173--193

[45] Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguadé, E., Labarta, J.: Productive Programming of GPU Clusters with OmpSs. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium. (2012) 557--568

[46] Florian Zaruba: Ariane: An Open-Source 64-bit RISC-V application class processor and latest improvements. URL: https://content.riscv.org/wp-content/uploads/2018/05/14.15-14.40-FlorianZaruba_riscv_workshop-1.pdf (2018)

[47] : ARM Information Center: Cortex-A9 Reference Manual

[48] : Yokogawa Power Meter Specifications