UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hardware-Software Co-design for Low-cost AI processing in Space Processors

Master in Innovation and Research in Informatics

High Performance Computing

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech
Facultat d'informàtica de Barcelona (FIB)

Author: Marc Solé I Bonet
Email: marc.sole.bonet@estudiantat.upc.edu
Director: Dr. Leonidas Kosmidis
Department: Computer Architecture
Email: leonidas.kosmidis@bsc.es
Date: October 21, 2021

Abstract

In the recent years there has been an increasing interest in artificial intelligence (AI) and machine learning (ML). The advantages of such applications are widespread across many areas and have drawn the attention of different sectors, such as aerospace. However, these applications require much more performance than the one provided by space processors. In space the environment is not ideal for high-performance cutting-edge processors, due to radiation. For this reason, radiation hardened or radiation tolerant processors are required, which use older technologies and redundant logic, reducing the available die resources that can be exploited. In order to accelerate demanding AI applications in space processors, this thesis presents SPARROW, a low-cost SIMD accelerator for AI operations.

SPARROW has been designed following a hardware-software co-design approach by analyzing the requirements of common AI applications in order to improve the efficiency of the module. The design of such module does not use any existing vector extension and instead has in its portability one of the key advantages over other implementations. Furthermore, SPARROW reuses the integer register file of the processor avoiding complex managing of the data while reducing significantly the hardware cost of the module, which is specially interesting in the space domain due to the constraints in the processor area.

SPARROW operates with 8-bit integer vector components in two different stages, performing parallel computations in the first and reduction operations in the second. This design is integrated within the baseline processor not requiring any additional pipeline stage nor a modification of the processor frequency. SPARROW also includes swizzling and masking capabilities for the input vectors as well as saturation to work with 8 bits without overflow.

SPARROW has been integrated with the LEON3 and NOEL-V space-grade processors, both distributed by Cobham Gaisler. Since each of the baseline processors has a different architecture set, software support for SPARROW has been provided for both SPARC v8 and RISC-V ISAs, showing the portability of the design. Software support been developed using two well established compilers, LLVM and GCC allowing for a comparison of the cost of developing support for each of them. The modifications have included the SPARROW instructions in the assembly language of each architecture and with the use of inline assembly and macros allow a programming model similar to SIMD intrinsics.

LEON3 and NOEL-V extended with SPARROW have been simulated and later implemented on a FPGA to evaluate the performance increase provided by our proposal. In order to compare the performance with the scalar version of the processor, different AI related applications have been tested such as matrix multiplication and image filters, which are essential building blocks for convolutional neural networks. With the use of SPARROW a speed-up of 6× has been achieved for matrix multiplication rising over 15× speed-up if saturation is enforced. Finally, SPARROW has been tested with a complex inference application from the open source GPU4S Benchmarking suite, achieving a 5.3× speed-up.

Acronyms

AHB Advanced High-performance Bus

AI Artificial Intelligence

ALU Arithmetic Logic Unit

AMBA Advanced Memory Bus Access

APB Advanced Peripheral Bus

ASIC Application Specific Integrated Circuit

BCC Bare-metal Cross-Compilation

BRAM Block Random Access Memory

BUFG Global BUFfer

COTS Commercial Off-The-Shelf

DMA Direct Memory Access

DPU Data Processing Unit

DRAM Dynamic Random Access Memory

DSP Digital Signal Processor

ESA European Space Agency

ESTEC European Space Research and Technology Centre

FF Flip-Flop

FPGA Field-programmable Gate Array

FPU Floating Point Unit

GCC GNU Compiler Collection

GPL General Public License

HLS High Level Synthesis

IO Input/Output

IPC Instructions Per Cycle

IP Intellectual Propriety

ISA Instruction Set Architecture

IU Integer Unit

LEO Low-Earth Orbit

LUTRAM Look-Up Table Random Access Memory

LUT Look-Up Table

MEC MEmory Controller

ML Machine Learning

MMU Memory Management Unit

NASA National Aeronautics and Space Administration

PLL Phase-Locked Loop

PMP Physical Memory Protection

RTOS Real-Time Operating System

SCR SPARROW Control Register

SIMD Single Instruction Multiple Data

VFP Vector Floating Point

VHDL Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

XPP eXtreme Processing Platform

Acknowledgements

I would like to thank first and foremost my advisor, Dr. Leonidas Kosmidis, who helped me to refine this project and provided me with support and directions during its development. Also I would like to thank him for giving me the opportunity to participate in the GPU4S project and work at the Barcelona Supercomputing Center in the CAOS group where I have met great colleagues who have also helped me in this work.

Specially, I would like to thank Alejandro Serrano and Guillem Cabo, for helping me setting up the FPGA and to work with it. At the same time I would like to express my gratitude to Alvaro Jover and Ivan Rodriguez for providing me access to the GPU4S Benchmark and for helping me to work with it.

I also have to thank Mr. Matthew Johns author of [1] for kindly providing me with the benchmarks used for his project. Similarly, I express my gratitude, to Mr. David Steenari, advisor of [2] and European Space Agency Technical Officer in the GPU4S project, for providing me with an example of LEON3 top-level entity for the Zynq Ultrascale+.

Finally, I thank my family for the support they have given me all this time and the education that they provided me with which allowed me to be where I am today.

This work was partially supported by ESA under the GPU4S (GPU for Space) project (ITT AO/1-9010/17/NL/AF), by the Spanish Ministry of Economy and Competitiveness (MINECO) under grants PID2019-107255GB and FJCI-2017-34095. It received also support by the European Commission's Horizon 2020 programme under the UP2DATE project (grant agreement 871465), the HiPEAC Network of Excellence and the Xilinx University Program (XUP).

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Artificial intelligence (AI) and machine learning (ML) have become a trend during the last decade. Needless to say that such popularity is well-justified, as they have revolutionised many fields. However, AI and ML algorithms require significant computational power, which is usually provided by accelerators such as GPUs or specialised ASICs.

Space processing has not been oblivious to such reality [3]; space, is an unsafe environment for humans and is not ideal either for the devices that operate beyond the boundaries of earth. Thus, such devices must behave as intelligently and reliably as possible. In many cases, due to the long distances, remote control from earth is inefficient and autonomy becomes a required feature for the missions to succeed. Recent missions by the European Space Agency (ESA) and National Aeronautics and Space Administration (NASA), as well as other agencies, have incorporated AI processing utilities such as for the navigation on NASA's latest Mars Rover, Perseverance [4].

Space processors, however, are constrained by the requirements of working in a harsh environment. They require low power consumption and older fabrication technologies in order to work reliably in space. Moreover, they are designed to comply with certain, strict guidelines for hardware design and their software, so that they obtain *space qualification*, which permits their use in space missions. Frequently, space processors such as Cobham Gaisler's LEON family of processors are used either implemented in ASIC or as soft-cores in radiation-hardened FPGAs. However, these processors have very low performance compared to processors used in consumer devices such as cell phones, and therefore cannot provide the performance required for the execution of AI algorithms.

For this reason, commercial off-the-shelf (COTS) accelerators have been considered as an alternative. For example, in ESA's Φ-Sat-1 technology demonstration mission, an Intel Movidius was used in an AI application for detecting clouds in satellite-earth images [5]. However, the drawback of COTS is their lack of radiation hardening which prevents their use beyond low-earth orbit (LEO). Moreover, the software stacks used by COTS accelerators are usually not developed for space nor for real-time operating systems (RTOS). For example they depend on Linux or machine learning frameworks like TensorFlow, which prevents their software qualification and makes their adoption in critical space missions even more challenging. This is the case also for widely FPGA-based AI accelerators used in numerous ground applications such as Xilinx's FINN [6] framework or Xilinx's DPU core which however are not designed subject to space qualification.

With all these considerations in mind, it is necessary to increase the performance for AI applications in a established space processor with as few changes as possible. This can be done by including an AI-centered SIMD unit. By targeting a processor already prepared for the adversities of space it's not necessary to take any other considerations regarding the safety of the device which has been tested in critical systems. Additionally, by including a small module we do not modify any of the characteristics that affect it's safety, while adding the computational power to execute more complex applications, especially AI related ones.

Furthermore, the addition of a module that does not modify the execution on the base processor for non-AI applications, which is mandatory in order to retain backwards compatibility with existing space software which is verified and reused from mission to mission. It would also be positive to not be limited to a single processor and instead be portable among many designs, this would help reduce the cost of new implementations. Moreover, if the programming part of the module is common regardless of the base processor architecture, the high-level code can be reused not having to take into account the differences in the microarchitecture details.

1.2 Objectives

With the considerations mentioned in the previous section, and in order to provide a solution to the mentioned problem, the main goals of this Master's thesis project are listed below:

1. The design of a **low-cost module for accelerating AI applications** by analyzing the requirements of said applications.

2. The design of the module for embedded and critical systems, specifically for the **space domain**, taking into account the constrains in such systems.

3. The implementation of such module in at least one **space processor** and evaluate its performance improvement. In this thesis we achieved this for two space processors, the **LEON3** and the **NOEL-V** enforcing the **portability of the proposed design**.

4. The creation of **software support** to take advantage of the hardware modifications for at least one of the widely used compiler frameworks, GCC or LLVM. Again in this thesis we achieved this goal for both compiler toolchains.

1.3 Thesis organization

This Master's Thesis is organized as follows: In Section 2, the state of the art is presented for vector processing units and space processor focusing on the space processors we have targeted in this work. In Section 3, we describe SPARROW, the AI acceleration module, with its design overview and characteristics. In Section 4, the software part of the project is described with the modifications in GCC and LLVM and the generation of the code. In Section 5, SPARROW is evaluated in terms of hardware overhead and performance speed-up with respect to the baseline processor. In Section 6, we explain some of the lessons we have learned when working on this project. Finally, the conclusions and future work are presented in Section 7 and publications, workshops, competitions and talks were contents of this Thesis were presented are listed in Section 8.

2 State of the Art

2.1 Vector Processing

Vector architectures have been introduced decades ago in order to increase the performance capabilities of computing systems, originally in the supercomputing domain, such as ILLIAC IV [7]. These early vector architectures relied on long vectors which were used to apply the same operation on each of the vector elements, effectively accelerating the processor's front-end, since each instruction could be fetched and decoded only once. However, at the time this was a very expensive technology since it required multiple chips.

With the improvements offered by Moore's law, vector instructions started appearing in commodity, single-chip microprocessors for the acceleration of multimedia applications. The first single-chip vector architecture implementation was T0 (Torrent-0) [8] achieving an IPC of 24 with an in-order single instruction issue per cycle. At the same time, the combination of vector processing with superscalar, out-of-order and multi-threading execution had been explored [9].

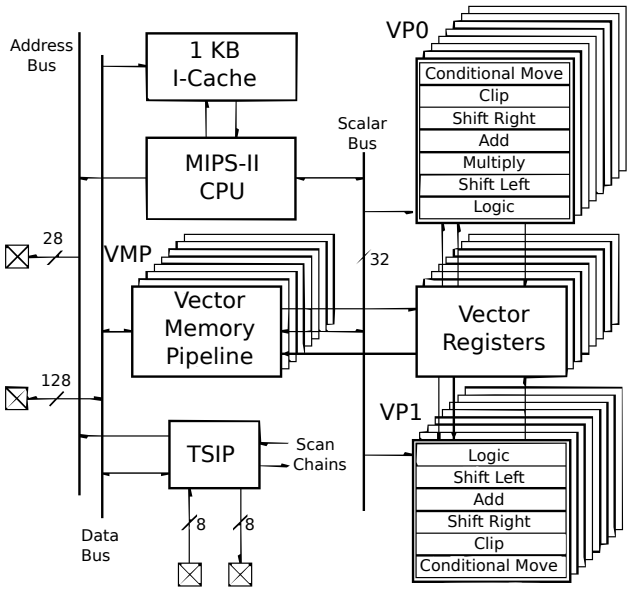


Figure 1: *T0 block diagram [8]*

Soon after, Intel introduced their first short vector ISA extension, MMX [10] for the Pentium and Pentium II processors. MMX instructions used the 64-bit lower bits of the floating point registers, in order to avoid changes in the operating system. However, this meant that only one of the MMX or floating point registers could be used at the same time. Apart from floating point operations, MMX instructions provided the possibility to use integer values of various component sizes, including 8-bits and support for saturation arithmetic. MMX does not support swizzling (re-ordering the fields of a vector within an instruction) in a native manner, but it offers instructions to pack and unpack values from vector registers and shift instructions to obtain the desired values.

In the following years, all major microprocessor vendors introduced SIMD vector extensions for multimedia processing. For example, SPARC introduced VIS [11], which similar to MMX aliased vector registers with the floating point register file, while other vendors introduced additional vector register files such as IBM for their AltiVec vector extension [12], and full support for swizzling. Intel kept releasing updated versions of their SIMD instructions, such as SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2 and AXV-512 and included hardware support for swizzling.

In the embedded domain, the systems were more area constrained, however a couple of years later the first embedded vector processor, VIRAM, appeared [13][14]. Several embedded processors started employing SIMD extensions. ARM started with the VFP extension (Vector Floating Point), sub-architecture of the ARM floating point architecture which is not a mandatory requirement of the architecture. It was named as such because the early ARM implementations featured some instructions with short vector functionality, which for area reasons were using a single ALU. Similar to other early SIMD vector architectures mentioned earlier, VFP was using the floating point register file. Currently this extension is known as ARM Common VFP (ARMv7 floating point) and ARM strongly recommended all its floating point functionality except for these short vector support to be implemented in hardware.

The ARMv7 specification [15] deprecated VFP and included it only for backwards compatibility while starting with the ARMv8 specification [16], it deprecated it completely and suggest implementers to use Advanced SIMD instructions, also known as NEON, for this functionality. Even after that, ARM retained the VFP naming for compatibility reasons. In order to retain backwards compatibility without the cost of an additional register file, Advanced SIMD registers and VFP registers are aliased, so they cannot be used simultaneously. NEON registers can support up to 8 single precision operations. ARM introduced the Advanced SIMD instructions, also known as NEON. NEON registers are aliased with the floating point registers in ARM CPUs, so they cannot be used simultaneously.

Recently, ARM introduced another vector architecture targeting IoT micro-controllers, called Helium [16], which is also known as the M-Profile Vector Extension (MVE). Again, in order to save area, Helium instructions alias with the floating point registers. Moreover, they operate on 32-bit portions of the vector registers each cycle, due to the limited datapath widths in these micro-controllers. Some Helium instructions such as multiplications access both the vector register file and the scalar register file, in order to accumulate 64-bit values.

In addition to silicon implementations of embedded vector architectures, there are several proposals for vector soft-cores on FPGAs. VESPA[17][18][19] uses a softcore accelerator generator for for Intel's (ex-Altera) NIOS II softcore (and the open source NIOS II-compatible processor UTIIe) or MIPS together with VIRAM vector instructions. In order to achieve an efficient design for FPGAs, it partitions the register file per vector lane.

Vegas is an evolution of VESPA [20] which relies on scratchpads instead of vector registers and supports vector chaining i.e. start executing one vector instruction before finishing the other, as well as heterogeneous lanes. While VESPA relies on scratchpads for holding the data on which vector processing is applied, it has additional needs for address registers, used for the DMA memory accesses between the scratchpad and the main memory.

VENICE [21] is a descendant of VESPA which removes the need of the extra address registers, using regular registers instead. This is somewhat similar to ARM's Helium and our design. However, the big difference is with the programming model. VENICE follows an accelerator programming model based on DMA transfers from a host processor.

VIPERS is another FPGA-oriented [22] vector soft-processor, which has been evolved and commercialized from VectorBlox. Its principle relies on the generation of custom vector processors for FPGAs supporting specialized instructions. This concept is similar to High-Level Synthesis (HLS) solutions provided by almost any FPGA vendor nowadays. While this is similar to our co-design, again it follows a different programming model compared to the existing space software. Moreover, both VESPA and VIPERS families of FPGAs vector processors and HLS solutions are only oriented to FPGAs, while our proposal is both applicable to space based systems implemented not only in FPGAs but also in ASICs.

2.2 Space Processing

In 1990, the European Space Research and Technology Centre (ESTEC) begun the development of the ERC32 [23], a 32-bit radiation-tolerant space processor. This CPU core was based on the Cypress CY601 SPARC V7 processor and it was finally completed in 1997, when it was used in various space missions, including the control computers of the International Space Agency [24].

The ERC32 consisted of three devices, an integer unit (IU), a floating point unit (FPU) and a memory controller (MEC) with all the required functions to host a real-time operating system. All three devices were provided with fault-tolerance logic by using a master and checker approach, where a mismatch is indication of an error. Additionally, built-in features for detecting transient and permanent errors, providing an almost 100% error-detection coverage when used in conjunction with the master checker duplex.

As a successor of the ERC32 the LEON processor [24] was developed to satisfy the requirements of the space missions after the year 2000. The new LEON was based on the also recently developed SPARC v8 architecture which allowed to maintain software compatibility with the ERC32 and also allowed the core to be open source. The core included on-chip fault-tolerance to detect and remove errors avoiding the overhead of spare units and voting. Even so, two LEON processors could work in duplex master/checker configuration.

In later years new versions of the LEON processor were released with different improvements but all being based on the SPARC v8 architecture, in this work I decided to use the LEON3 processor for which more details are provided in Section 2.3.1. However, later releases of the LEON family reach up to the LEON5. Additionally, Cobham Gaisler, the company in charge of developing the LEON processors, has also released a RISC-V compliant fault-tolerant processor, the NOEL-V, which is also used for this project and presented in Section 2.3.2.

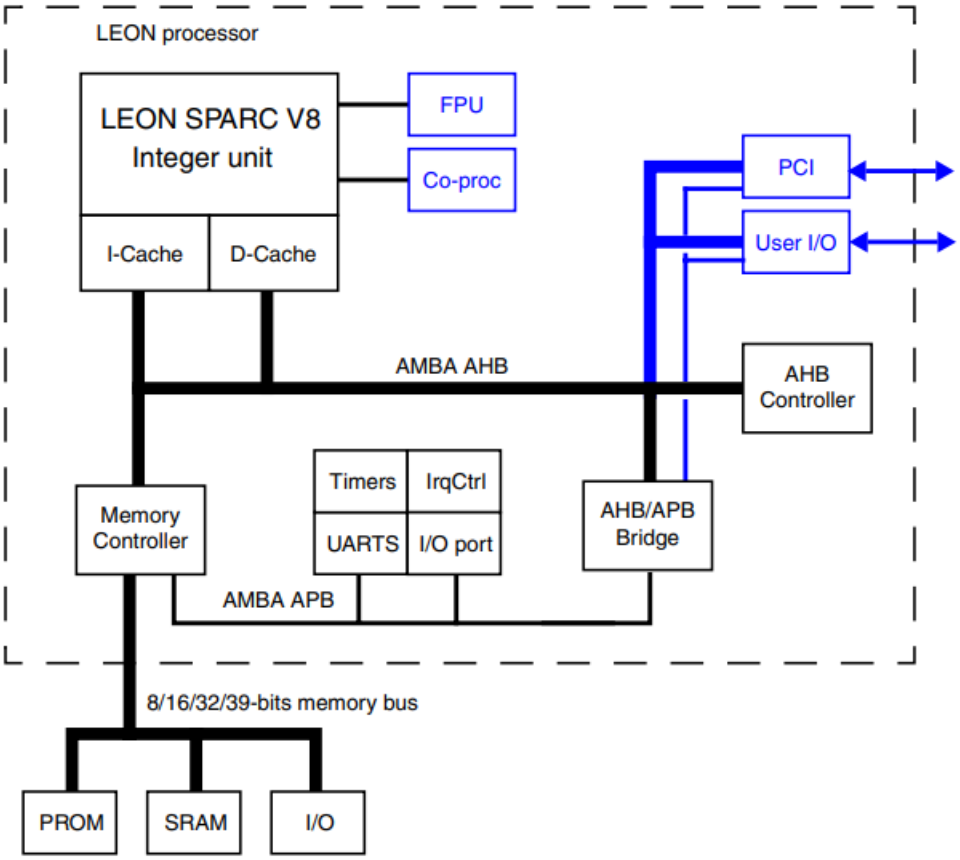


Figure 2: *LEON block diagram [24]*

Aside from the LEON family, other processors incorporated radiation-hardening to be suitable for use in space. An example of this is the RAD750 [25], a space processor that is identical to the PowerPC 750 microprocessor, but adapted for the space environment. The RAD750 processor was used in the Lunar Reconnaissance Orbiter in 2008, a NASA mission to scout the lunar surface in order to prepare future manned missions.

Both the LEON and the RAD750 were general purpose processors, however, in many situations and for certain tasks, the utilization of digital signal processors (DSP), like the TSC21020, was preferred. It provided better performance in the signal processing and data handling applications, but on the other hand lacked programmability and flexibility. Other approaches incorporated parallel computing in the signal processing, such as the eXtreme Processing Platform (XPP) [26]. XPP consisted of a number of Processing Array Elements of 32-bit arithmetic elements. A key characteristic of XPP is that instead of being controlled by the instruction flow, it uses a configuration flow replacing simple ALU operations with complex functions. In Figure 3 we can see a comparison of performance against flexibility for the presented space processors.

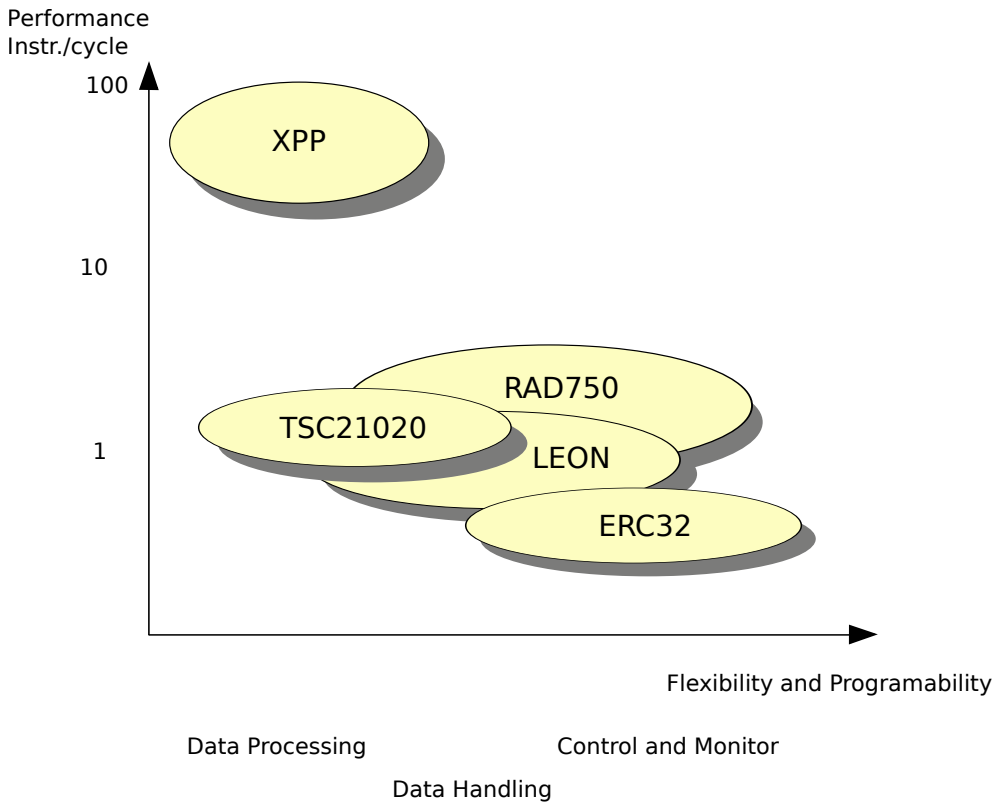


Figure 3: *Space processors performance comparison [26]*

As an alternative to fault-tolerant ASIC cores, FPGA implementations reduce the design cost. In order to be used in space, and thanks to technology improvements, radiation hardened designs are available at the market such as the Xilinx Virtex-5QV or the Microsemi RTG 64. An example of recent use of radiation tolerant FPGA is the Perseverance Mars rover, where Microsemi Rad-Tolerant FPGAs were used [27].

2.3 GRLIB

GRLIB is a collection of IP components distributed by Cobham Gaisler. The company, is one of the world leaders in embedded computer systems for harsh environments, such as space. In fact, Gaisler has a close partnership with the European Space Agency (ESA) for the development and validation of space-grade microcontrollers, such as the LEON3FT (LEON3 Fault-Tolerant). The usage of the provided cores is not limited to space, as they are also used in the automotive sector, for multimedia or as a general SOC platform specially in embedded systems [28].

The GRLIB library is provided under a GNU GPL License, but can also be licensed for commercial use. In the later case, different distributions are available with fault-tolerant support and targeting FPGAs or ASICs. Even with the limitations on the free distribution, GRLIB IP cores are written in VHDL and can be easily reused for different projects. Additionally, full-processor designs are included which can be simulated or synthesised using different tools depending on the requirements of the users. This top-level designs are specially designed for a number of FPGAs and there is no additional support outside of these. However, there is a mailing list available for collaborative troubleshooting for the GPL version in order to help with the utilisation of the library or porting it to new platforms [29].

The library includes cores for communication control such as AMBA AHB/APB, the LEON3, LEON4 and LEON5 processors, the NOEL-V RISC-V processor, and a number of different protocols, memory management and utility cores. The library is extensively documented either for the configuration and initialization of GRLIB [30] or for the documentation of each of the provided IPs [31].

2.3.1 LEON3

The LEON3 [31] is a successor of the LEON processor mentioned in Section 2.1. As such it keeps a few common characteristics while providing better performance and more capabilities. It is a 32-bit processor conforming to the IEEE-1754 (SPARC v8) architecture, it is considered a high-performance core for embedded systems and since it can be enhanced with fault-tolerance, it is widely used in the space domain.

It has a Harvard architecture with a 7-stage pipeline, it can include high-configurable cache for instructions and data which are independent one from the other. Also, it incorporates a hardware multiply and divide units, however, for this operations are costly as they require additional cycles. The core also includes debug support, which allows to evaluate the processor from outside and obtain results regarding the performance or cache utilization.

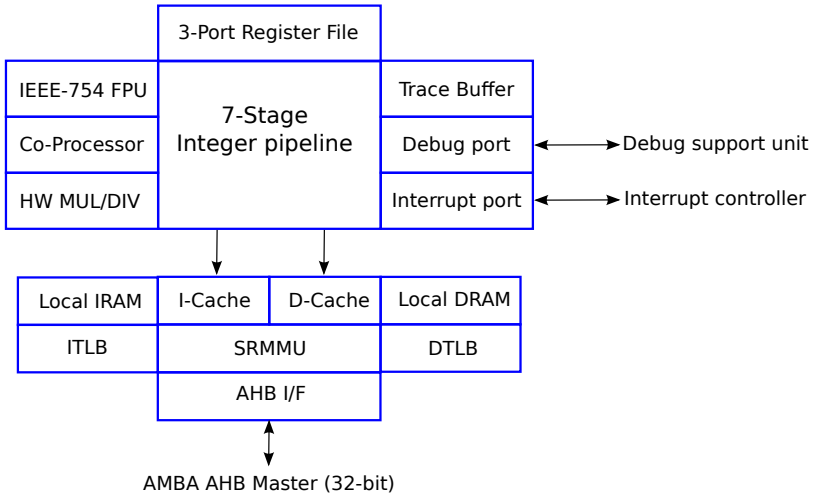


Figure 4: *LEON3 block diagram [31]*

There are different interface ports which allow communications with the outside, through the AMBA AHB protocol. A block diagram for the LEON3 processor is depicted in Figure 4.

LEON3 has a dedicated cache for instructions and one for data, which are easily configurable with 1-4 ways, 1-256 KiB/way and 16 or 32 bytes per line. The number of register windows, which are used in the SPARC standard for low latency function calls, is also configurable, from 2 to 32 register windows. For the support of multiplication and division operations hardware units are included.

An overview on the LEON3 integer pipeline which is the most relevant one for this project, is shown in Figure 5. The 7 pipeline stages are described below:

- **Instruction Fetch (FE):** Fetch the instruction from the instruction cache if present, otherwise fetch it from the AHB bus.
- **Decode (DE):** Decode the instruction and evaluate data dependencies setting the bypass logic or stalling the pipeline, the target addresses of BRANCH and CALL are generated.
- **Register Access (RA):** Prepare the operands for the instruction, whether from the register file or from internal bypasses.
- **Execute (EX):** Perform the ALU, logic and shift operations. For memory operations, JMPL and RETT compute the address.
- **Memory (ME):** Access the data cache if enabled, otherwise the access, read or write, is forwarded to main memory.
- **Exception (XC):** Resolve traps and interrupts. For memory reads it aligns the data.
- **Write-back (WR):** The result obtained from the ALU or memory is written in the corresponding register.

2.3.2 NOEL-V

In contrast to the SPARC based LEON family, Cobham Gaisler released on the Christmas Day of 2020 a RISC-V processor, the NOEL-V [31]. Although there are differences in the architecture, there are many similarities between both processors. Moreover, as it is the case of LEON3, the NOEL-V is highly configurable. There are 4 basic configurations, from a tiny one with minimal components to a high performance version. Furthermore, the processor can be set for 32-bit (RV32) or 64-bit (RV64) and aside from the tiny and minimal configurations, a double-issue pipeline is available. In Figure 6 an overview of the NOEL-V system is shown.

NOEL-V follows a Harvard architecture with separate data and instruction caches, too. The integer unit is also divided in 7 stages and it is very similar with the LEON3 one, and includes multiplication and division logic. In addition, the NOEL-V has late ALU and late BRANCH support to perform said operations in the exception stage which allows to avoid pipeline stalls because of dependencies.

In Figure 7 the integer pipeline of the NOEL-V is presented. Below the description of each stage is provided:

- **Instruction Fetch (FE):** A 64-bit word is fetched from the instruction cache if enabled, otherwise it is fetched from the AHB bus. The 64-bit word contains two to four instructions which are latched in the IU.
- **Decode (DE):** Decode two instructions and evaluate if dual-issue is possible and which instruction goes to each lane. Also set the logic for handling the data dependencies.
- **Register Access (RA):** Prepare the operands for the instruction, whether from the register file or from internal bypasses.
- **Execute (EX):** Perform the ALU, logic and shift operations. For memory operations, JMPL and RETT compute the address.
- **Memory (ME):** Access the data cache if enabled, otherwise the access, read or write, is forwarded to main memory.
- **Exception (XC):** Resolve traps and interrupts. Late ALU and BRANCH functionalities are available to allow delayed dependency resolution.
- **Write-back (WR):** The result obtained from the ALU or memory is written in the corresponding register.

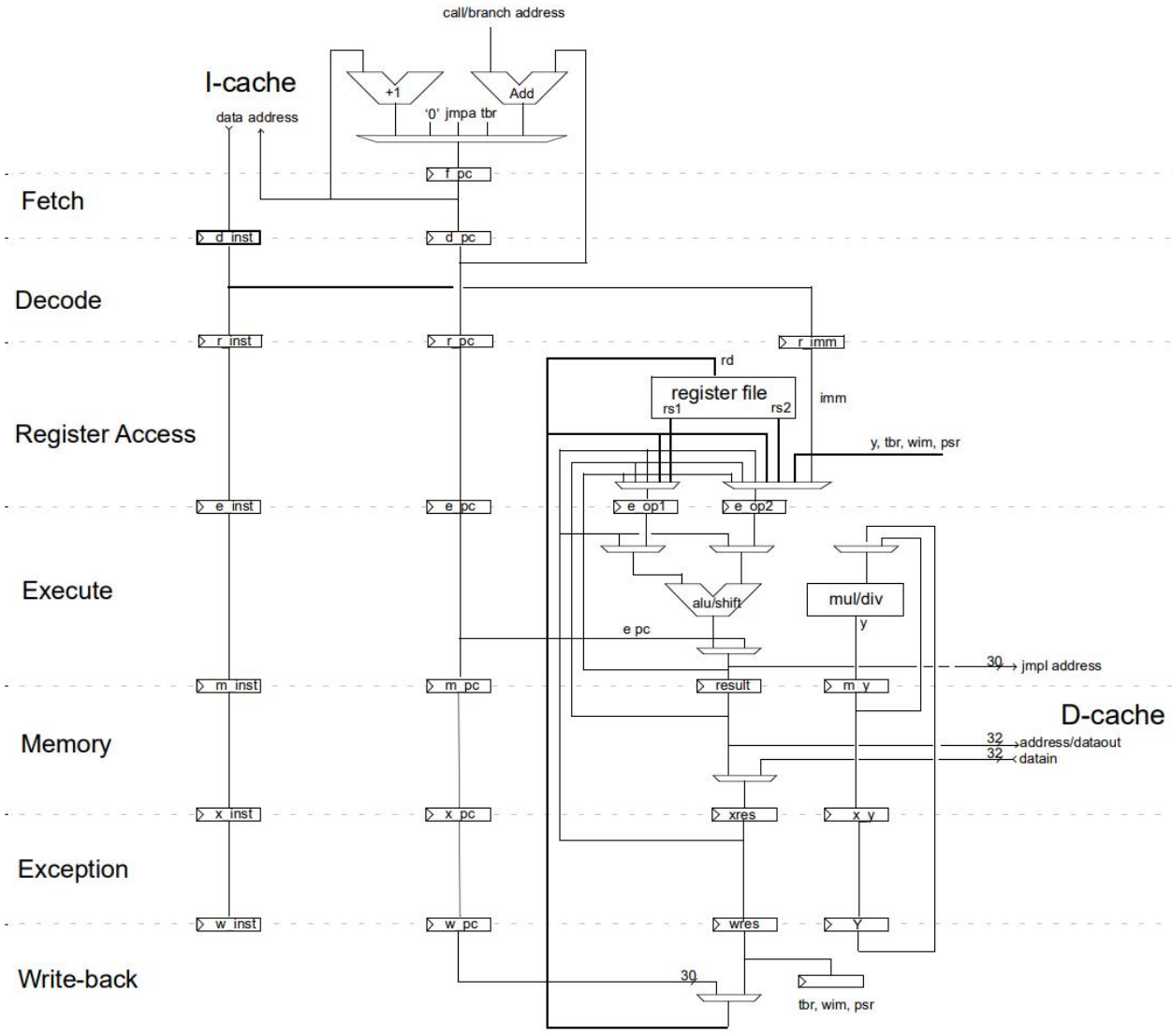


Figure 5: *LEON3 Integer Pipeline [31]*

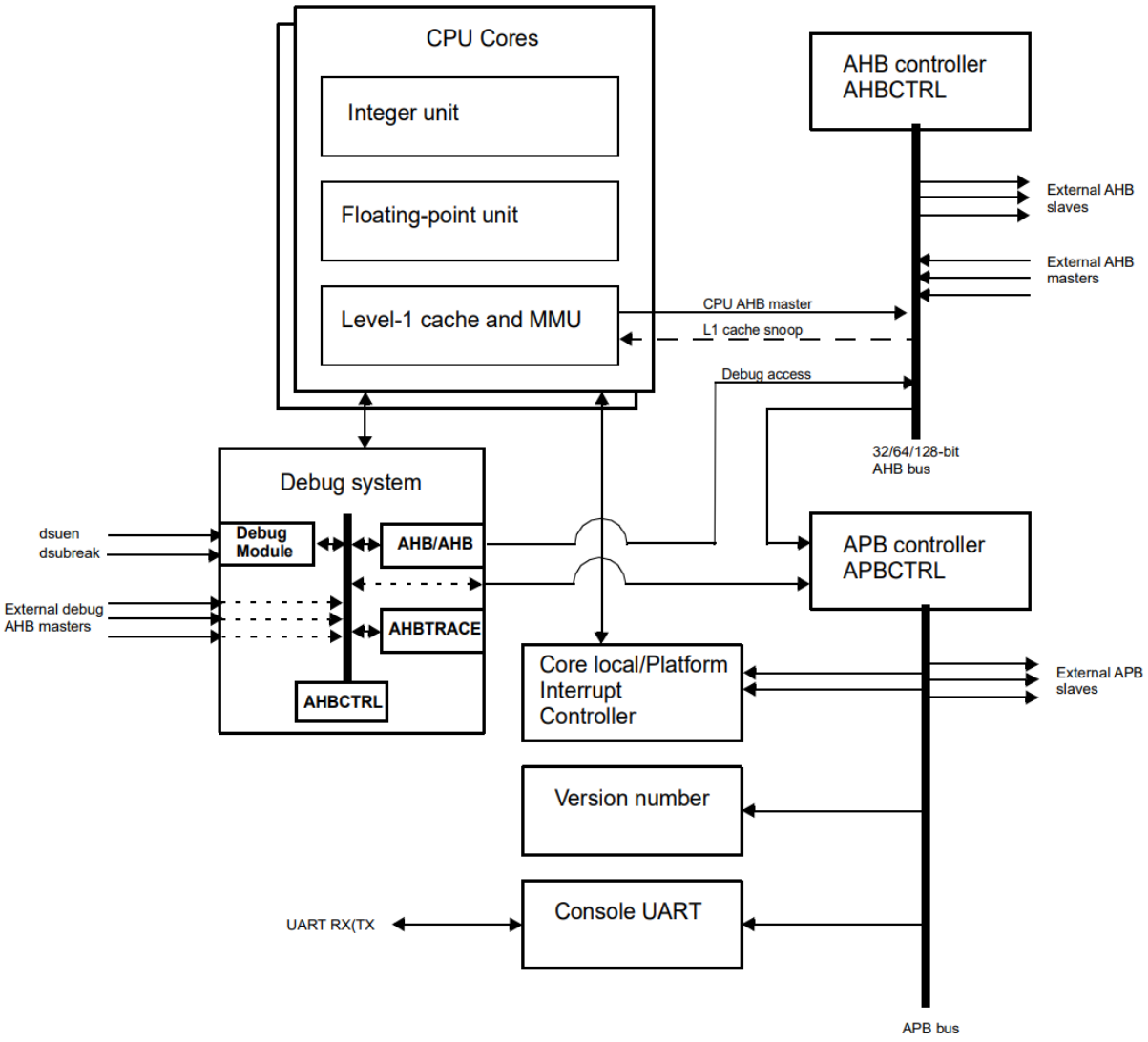


Figure 6: *NOEL-V block diagram [31]*

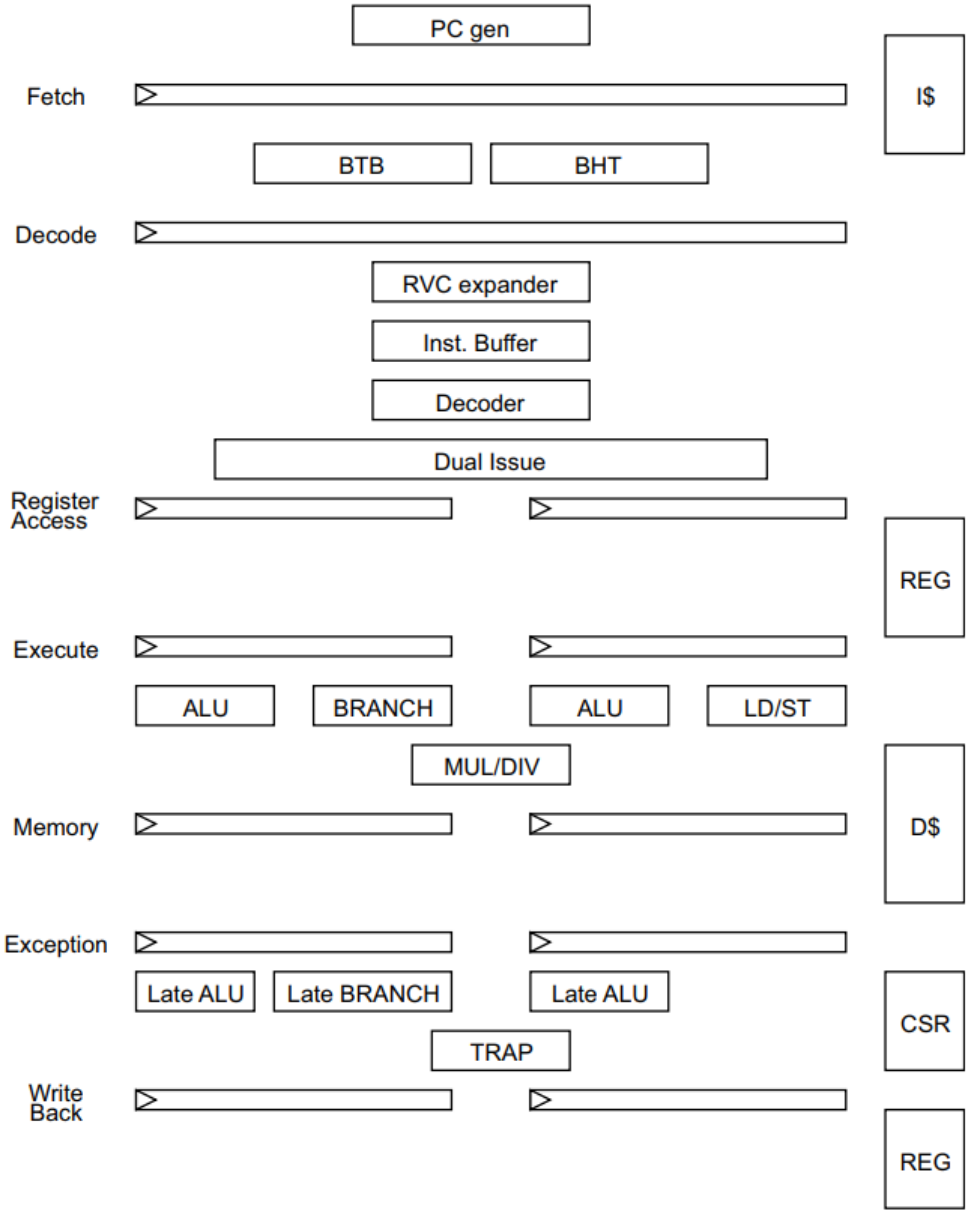


Figure 7: *NOEL-V Integer Pipeline [31]*

3 SPARROW

3.1 Design overview

To satisfy the objectives introduced in Section 1.2 in this project we have designed SPARROW. The name, conveys the idea of its small size, similar to the small bird sparrow and includes the word *arrow* as a reference to speed provided by the acceleration that module offers. Additionally it starts with *SP*, which induces the idea of space.

SPARROW is an extension module for space processors and it is currently implemented for both the LEON3 and NOEL-V processors and adapted to their ISAs, SPARC v8 and RISC-V respectively. More precisely, it extends the integer pipeline with additional short vector operations with focus on AI applications, without any performance cost in the rest of the operations of the base processor. To guarantee this, the cycle time and the pipeline depth is the same as of the unmodified baseline processor. To do so, the module is attached at the execution stage and the result is returned at the exception stage. Since no instruction operated in SPARROW requires to access memory nor it can generate any exception there is no risk nor any additional consideration to be made. With this, not only the conditions for the regular integer pipeline are kept but also for the new SIMD instructions.

A key design decision of SPARROW is the reuse of the integer register file to reduce the area overhead of the design. We notice that in conventional architectures the vector register file consumes a significant portion of the vector design. For example in T0 [8], the vector register file is 3× larger than the baseline MIPS-II processor, and consumes 30% of the overall vector unit area. In VIRAM-1 [13] the vector register file has the same size with the baseline scalar core, and again it accounts for 30% of the total vector additions. In a similar way the vector register file of Samsung's M3 high performance embedded processor [32] consumes an analogous share of the real-estate dedicated to the implementation of the NEON SIMD extensions of the ARMv8 Specification [16]. Therefore, our decision results in a SIMD design which is at least 20-30% smaller than any other vector design targeting embedded processors, as is presented later in Section 5.

Another important element of SPARROW's design is its hardware-software co-design for AI processing. By analyzing the literature it is apparent that one of the most significant operations in ML is the dot product [33], which is primarily used in the matrix multiplication implementation. Matrix multiplication is one of the most common operations in such applications, because it is used both for the implementation of fully connected layers, as well as for convolutions. Thus, the optimization of this operation was the starting point of the design of SPARROW's architecture, resulting in the two stage architecture design which is explained in more detail in the following sections.

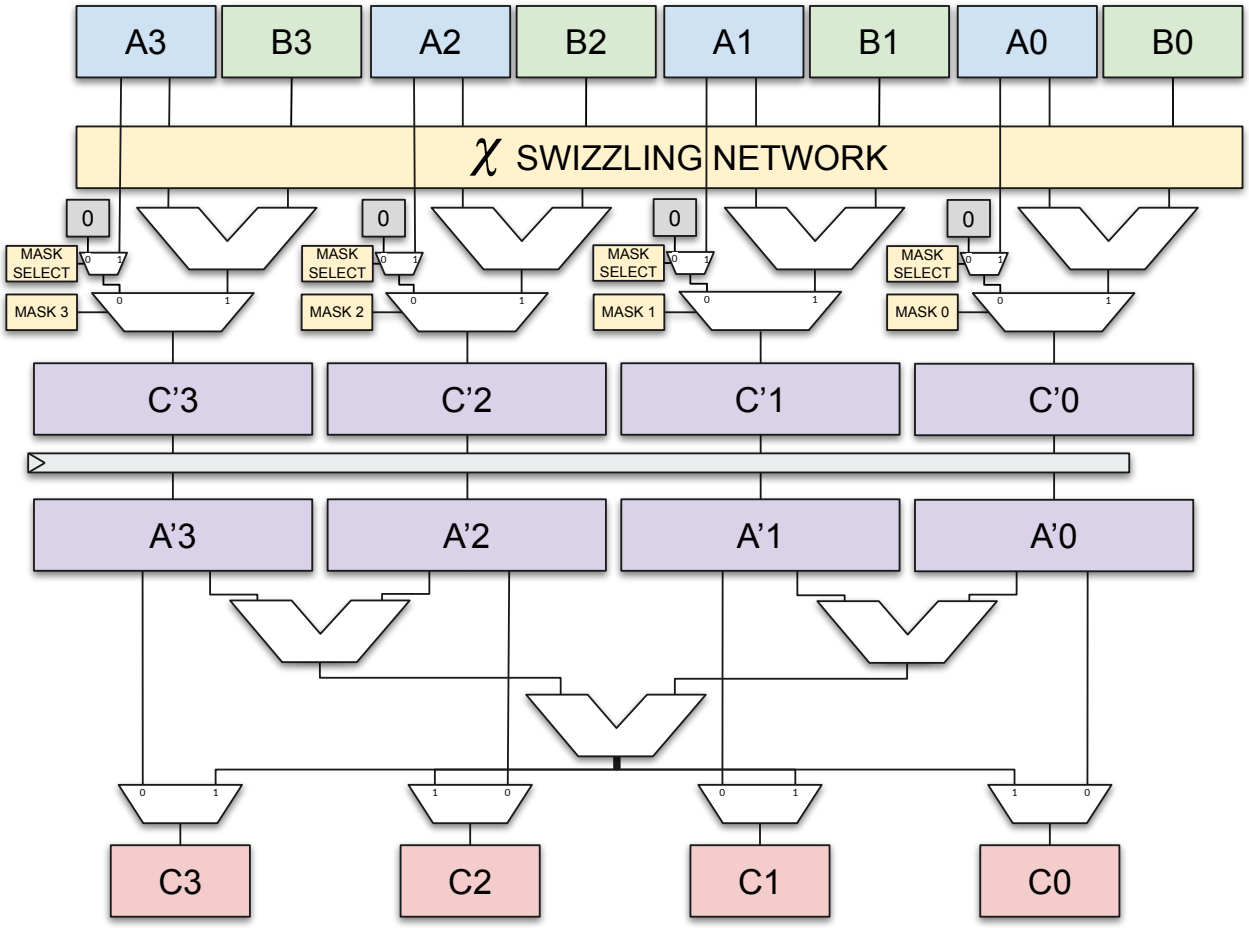


Figure 8: *Outline of the SPARROW module*

In addition, recent studies have shown that reducing the precision of arithmetic operations involved in machine learning inference operations to 8-bit integers provides minimal reduction in the inference accuracy. As consequence, almost every architecture designed for inference workloads nowadays operate with 8-bit integers or even smaller bit widths [34]. This is compatible with the choice to reuse the integer register file, which consists of 32-bit registers and therefore can accommodate 4 values which can be processed in a SIMD manner. In order to avoid the possible overflow introduced when working with 8-bit values, the different operations of SPARROW include a saturation option. Moreover, the decision to work only with 8-bit widths allowed us to take implementation choices that would otherwise be impossible to implement without a significant impact on the cycle time or the area of SPARROW.

An outline of SPARROW is shown in Figure 8, where the clear division of the module in two stages can be seen. In the first stage, the two input registers execute 4 operations in parallel in a conventional SIMD fashion, whilst being subject to traditional vector modifiers, such as swizzling or masking. The four components of the first stage result are either combined in reduction operations to produce up to a 32-bit result or passed to the module output without additional modifications. In addition, in both stages the result can be saturated. Any of the two stages can be disabled acting as a `nop`, in case of the second stage, the value can be bypassed directly to the integer pipeline saving a cycle in case of dependency.

As both baseline processors, SPARROW has been written in VHDL. Furthermore, in order for the design to be consistent with the rest of the GRLIB library the structure of the VHDL file follows the design method presented in [35].

3.2 First Stage: Parallel computing

In the first stage, the two source vectors are modified according to the swizzling configuration, which is further explained in Section 3.4. The resulting reordered vectors, the operand vectors, are computed one against the other at component level. For arithmetic operations a saturation option is available, in which case the result is clipped between 0 and 255 for unsigned data or between -128 and 127 for signed. Otherwise, the 8-bit operands can require an extra bit to avoid overflow and in case of multiplication a 16-bit value is needed. For this reason, the results of the first stage are stored in longer intermediate components, as can be seen in Figure 8.

The result of the first stage operation is then masked. This introduces more versatility to the module and can greatly simplify the management of the data. More details on the mask behaviour are explained in Section 3.4.

With the described design, SPARROW can work with two input registers. Nonetheless, in some cases it is necessary to work with constants which can be costly to set into a register, especially if the higher bits must be set to work in all the register components. The alternative is to use immediates, whose value is replicated in all the components. However, we would require 8 free bits in the instruction to pass the value, but, as shown in Section 3.5 there is not enough space to do so. Instead, we decided to encode in the 5 bits used for the second operand the most common values for ML applications [36]. This is a result of the hardware-software co-design approach which was fundamental for the implementation of this project.

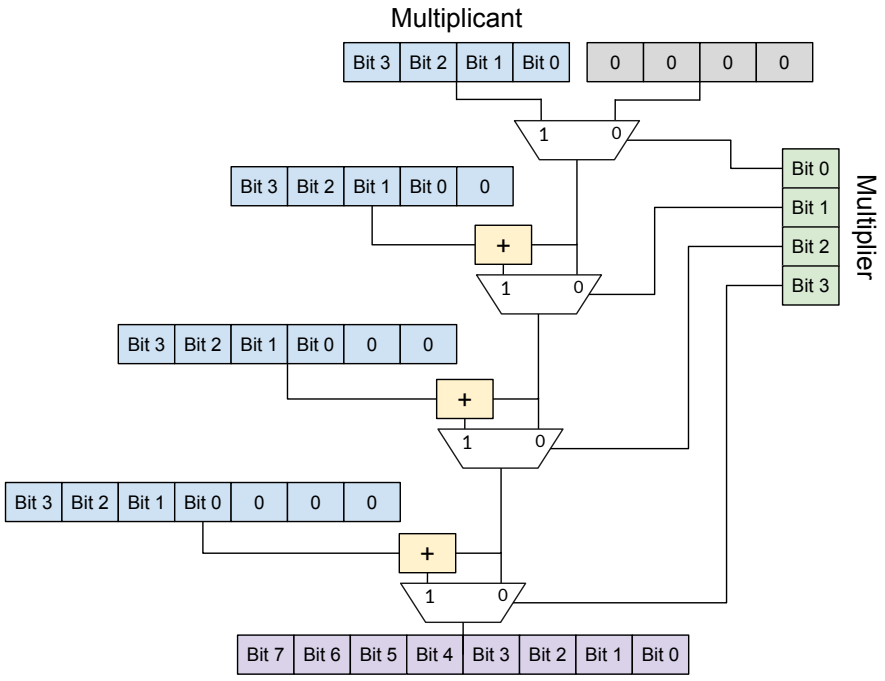


Figure 9: *Multiplication logic implementation in SPARROW with four bits for simplicity*

3.2.1 Multiplication logic

One of the vital design characteristics of SPARROW is the implementation of multiplication in the first stage. Multiplication is traditionally a costly operation which requires multiple cycles to be executed and has a high area cost, as we have seen in the description of the original LEON and NOEL processors in Section 2.3. Implementing a full 4-component SIMD multiplication in a single pipeline stage would impact the frequency of the original design or it would require to be split in multiple pipeline stages. However, since SPARROW works on 8-bit operands the operation fits within a single cycle and does not cause any stall in the pipeline. In fact, in the first implementations of the module we had to reduce the target frequency this was not possible to achieve, since due to a more general implementation, the design did not meet the timing constraints.

In order to solve this, we re-implemented the multiplication logic taking a more explicit approach. Instead of letting the VHDL libraries perform the full multiplication, the logic computes the result by applying a shift and addition for the 8-bits. A reduced version of this design with 4-bit inputs is shown in Figure 9.

For the multiplication of signed numbers this approach is still valid, but only for the first 8 bits of the result. Due to the introduction of longer intermediate values the result cannot be considered as valid and must be somehow fixed. The first alternative would be the sign extension of the operands to 16 bits, however, the logic grows too big and it makes impossible to fit the multiplication in a single processor cycle at the original processor frequency.

For this reason, we decided to operate always with positive values, and later fix the sign of the result. That means that in case of signed multiplication, if the component first bit is set, two's complement inversion is performed. After computing the multiplication, if the original operands had the same sign the result is left as is (positive) whereas if the sign was different an inversion is performed again to produce a negative result. Including this logic does not entail any significant overhead as sign evaluation was already present to compute the saturation. The algorithm to perform the multiplication is as follows:

1. Invert the sign of the operands if they are negative (leftmost bit equal to 1) and the operation is signed.
2. Perform the product.
3. Use a multiplexer to select a result as in the algorithm shown in Figure 10 where overflow means that one of the 8 most significant bits is set (in case of signed, the 9th most significant bit is also checked).

```
if saturation is enabled then
    if operation is signed then
        if Operand 1 sign = Operand 2 sign then
            if overflow then
                result is 0x7F
            end
        else
            if overflow then
                result is 0x80
            else
                result is sign inverted
            end
        end
    else
        if overflow then
            result is 0xFF
        end
    end
else
    if operation is signed then
        if Operand 1 sign ≠ Operand 2 sign then
            result is sign inverted
        end
    end
end
```

Figure 10: *Algorithm for selecting the result of the multiplication*

3.3 Second Stage: Reduction operations

Second stage operations are those that compute a single result by combining all the components of the intermediate vector. These reduction operations also have a saturated and non-saturated version, however, due to limited bits in the instruction encoding the same saturation option as in the first stage is used. Even with this restriction, using two consecutive instructions it's possible to have any result.

Since the result is a single 32-bit value, there is no precision loss in the second case, but even with 32-bit results the saturation option is necessary to retain the validity of the data inside the program. Furthermore, the intermediate operations are performed increasing the data length to avoid overflow. In the saturated case, the clamping is performed only to the final result, to avoid different results depending on the order of the components.

If there is no need for a reduction operation, a `nop` can be specified. In that case the result will be the output of the first stage with no modification whatsoever. Moreover, in this scenario, the first stage result is bypassed to the integer pipeline and can be immediately used with no need to wait for an additional cycle.

3.4 Additional features: SPARROW Control Register

To provide SPARROW with more versatility, we have included additional features like swizzling and masking which as we have mentioned in Section 2.1 are common in some SIMD designs. In order to configure these characteristics, we added in the architecture a special register, the SPARROW (or SIMD) Control Register (SCR).

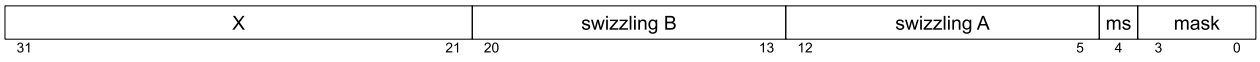


Figure 11: *SPARROW Control Register encoding*

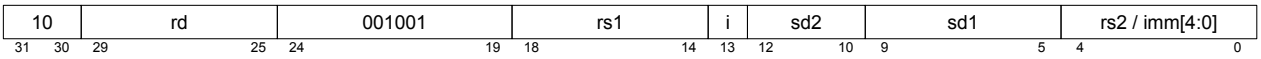
The encoding of the SCR is shown in Figure 11 where:

- **Mask:** Each of the four bits references one of the intermediate vector components, if the bit is set the result from the first stage is applied, otherwise the mask is passed.
- **Mask select (ms):** Selects the mask to be applied. If one, the value is that of the first operand respective component, previous to the swizzling. Otherwise, the passed value is zero.
- **Swizzling:** Sets for operands A and B – first and second – the position of each component. For each of the operands, the ith pair of bits identifies the vector component in the ith position.
- **Free (X):** Empty bits available for future extensions.

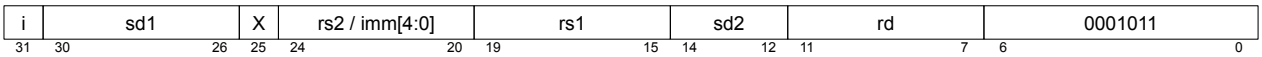
3.5 SPARROW instructions

To instruct the processor to use SPARROW for the desired operations those must be added in the ISA. Since SPARROW uses the integer register file, there is no significant modifications required in the base processor, however, it is necessary to find free opcodes to identify such instructions. Instead of using existing vector extensions, such as the RISC-V Vector extension, we decided to take a custom approach as it allows for higher flexibility, like reusing the integer register file, and also increased portability to different ISAs. Furthermore, existing vector extensions include additional instructions for setting the vectors which is not necessary for SPARROW again because of the reuse of the register file.

Although the module has been implemented for two different processors with different architectures, the implementation of the instructions is not complex proving the portability of the module. Therefore, we had just to decide on the opcodes for SPARC and RISC-V. Ideally they would be the same to simplify the identification, nonetheless, this is impossible due to the differences of the ISAs. However, in both cases we tried to use free opcodes similar to the integer instructions of each respective architecture. In Figure 12a the SPARROW instruction encoding for SPARC v8 is shown, in Figure 12b the same is shown for the RISC-V ISA.



(a) *SPARROW SPARC v8 instruction*



(b) *SPARROW RISC-V instruction*

Figure 12: *SPARROW instruction for the SPARC v8 and RISC-V architectures*

In Figure 12 the specific opcodes for each of the architectures are shown in binary. The rest of fields are described below:

- **rd:** Destination register
- **rs1:** First source register
- **rs2:** Second source register
- **imm:** 5-bit encoded immediate
- **i:** Use immediate instead of register for the second operand
- **sd1:** SPARROW first stage operation code (see Table 1)
- **sd2:** SPARROW second stage operation code (see Table 2)

sd1	Name	Operation		
00000	`nop`	$rd' = rs1$	$rd' \in \mathbb{N}$	
00001	`add`	$rd'_i = rs1_i + rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
00010	`sub`	$rd'_i = rs1_i - rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
00011	`mul`	$rd'_i = rs1_i \times rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
00101	`max`	$rd'_i = max(rs1_i, rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
00110	`min`	$rd'_i = min(rs1_i, rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
00111	`and`	$rd'_i = rs1_i \wedge rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
01000	`or`	$rd'_i = rs1_i \vee rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
01001	`xor`	$rd'_i = rs1_i \oplus rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
01010	`nand`	$rd'_i = \neg(rs1_i \wedge rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
01011	`nor`	$rd'_i = \neg(rs1_i \vee rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
01100	`xnor`	$rd'_i = \neg(rs1_i \oplus rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
01101	`sadd`	$rd'_i = max(-128, min(127, rs1_i + rs2_i))$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
01110	`ssub`	$rd'_i = max(-128, min(127, rs1_i - rs2_i))$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
01111	`smul`	$rd'_i = max(-128, min(127, rs1_i \times rs2_i))$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
10000	`merg`	$rd' = rs2$	$rd' \in \mathbb{N}$	
10001	`shft`	$rd'_i = rs1_i \ (\ll \mid \gg)^1 \ rs2_i/2$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{Z}$
10011	`umul`	$rd'_i = rs1_i \times rs2_i$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
10101	`umax`	$rd'_i = max(rs1_i, rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
10110	`umin`	$rd'_i = min(rs1_i, rs2_i)$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
11001	`sshft`	$rd'_i = max(-128, min(127, rs1_i \ (\ll \mid \gg)^1 \ rs2_i/2))$	$\forall i \in \{0,...,3\}$	$rs1 \in \mathbb{Z}^2$
		$rd'_i = max(0, min(255, rs1_i \ (\ll \mid \gg)^1 \ rs2_i/2))$	$\forall i \in \{0,...,3\}$	$rs1 \in \mathbb{N}^2$
11101	`usadd`	$rd'_i = max(0, min(255, rs1_i + rs2_i))$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
11110	`ussub`	$rd'_i = max(0, min(255, rs1_i - rs2_i))$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$
11111	`usmul`	$rd'_i = max(0, min(255, rs1_i \times rs2_i))$	$\forall i \in \{0,...,3\}$	$rs1, rs2 \in \mathbb{N}$

1. The second operand sign determines the direction of the shift. Also its last bit identifies whether the shift is logic or arithmetic.

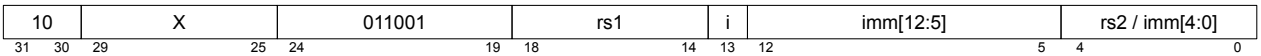
2. If the shift is logical it treats the data as unsigned whereas if it's arithmetic as signed.

Table 1: *SPARROW first stage operation codes*

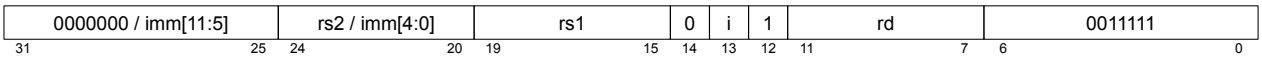
sd2	Name	Operation	
000	`nop`	$rd = rd'$	$rd' \in \mathbb{N}$
001	`sum`	$rd = \sum rd'_i \quad \forall i \in \{0, ..., 3\}$	$rd' \in \mathbb{Z}$
010	`max`	$rd = max(rd'_0, rd'_1, rd'_2, rd'_3)$	$rd' \in \mathbb{Z}$
011	`min`	$rd = min(rd'_0, rd'_1, rd'_2, rd'_3)$	$rd' \in \mathbb{Z}$
100	`xor`	$rd = rd'_0 \oplus rd'_1 \oplus rd'_2 \oplus rd'_3$	$rd' \in \mathbb{N}$
101	`usum`	$rd = \sum rd'_i \quad \forall i \in \{0, ..., 3\}$	$rd' \in \mathbb{N}$
110	`umax`	$rd = max(rd'_0, rd'_1, rd'_2, rd'_3)$	$rd' \in \mathbb{N}$
111	`umin`	$rd = min(rd'_0, rd'_1, rd'_2, rd'_3)$	$rd' \in \mathbb{N}$

Table 2: *SPARROW second stage operation codes*

In Table 1 and Table 2 the different stage instructions are presented. For simplicity the operation does not include the modifications introduced by the mask or the swizzling, but the different sign and saturations options are shown. There are a total of 24 operations in the first stage and 8 in the second, which can be combined to produce 200 unique instructions, which can be further configured with the SCR. In both tables the intermediate vector register is identified with `rd'`, and the data type, whether signed or unsigned (integer or natural) indicates if sign extension is performed when storing the result. In some cases where the same instruction is used for both data types, like `add`, the sign is deduced from the second stage operation.

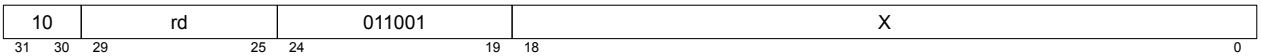


(a) *SPARROW SCR write SPARC v8 instruction*

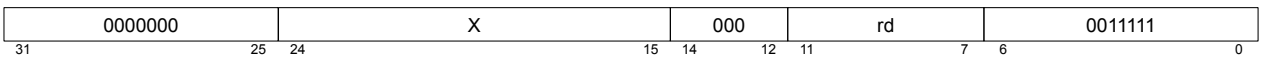


(b) *SPARROW SCR write RISC-V instruction*

Figure 13: *SPARROW SCR write instruction for the SPARC v8 and RISC-V architectures*



(a) *SPARROW SCR read SPARC v8 instruction*



(b) *SPARROW SCR read RISC-V instruction*

Figure 14: *SPARROW SCR read instruction for the SPARC v8 and RISC-V architectures*

In Figure 13 the instructions to set the SCR value are shown. The stored value is the result of the `xor` of the two operands. This simplifies the writing of the value while keeping the possibility of just writing the value of the register when setting the second operand as zero. This operation is performed in the integer pipeline reusing the existing logic to do it. This approach is in fact a common method for setting special registers as is for example the case with SPARC v8. In Figure 14 the instructions shown are those for reading the SCR value, in a more straightforward way, the value is stored in the destination register.

4 Software Support

4.1 SPARROW assembly

An important advantage of SPARROW compared to custom accelerators is the ability to reuse the existing qualified software stack of LEON3 i.e. the RTEMS real-time operating system or bare-metal space applications, which reduces both the cost and the effort of the development of a new compiler from scratch as well as its qualification cost later. For this reason we focused in adding the SPARROW instructions in the SPARC v8 and RISC-V assembly. Although outside of the scope of this project, we plan to implement full compiler support for SPARROW with automatic vectorization and different optimizations.

We added SPARROW support in the two most widely used compilers nowadays, GCC and LLVM. To do so, we modified the `binutils` of Gaisler's *bcc-2.2.0* GCC-derivative compiler and the base *LLVM v13.0*. Having software support for both compilers adds to the idea of having increased portability. Furthermore, it will allow to compare the performance of both approaches and will allow to also evaluate the effort required for working with each of them.

To identify the new instruction we decided to use an underscore as separator between the instruction names, which are those depicted in Table 1 and Table 2. In the case of `nop` it is omitted for the second stage, ending the instruction name with the underscore. Although we would have liked to do the same in the first stage, it is not possible for an instruction to start with an special character in the gnu assembler used by bcc/GCC. We also added aliases such as the dot product, which can be both represented by `mul_sum` or `dot`. As with the SPARROW instruction names an `s` and `u` prefix in the alias denotes saturation and unsigned operation respectively, i.e. `usmul_usum` is `usdot`.

To access the SPARROW control register the instructions implemented differ between SPARC v8 and RISC-V. In the former case, the SCR can be accessed using the `wr`, `rd` and `mov` instructions already present on the ISA for accessing special registers. For the latter, new `scrwr` and `scrrd` were added with the same behaviour as the SPARC v8 instructions.

The addition of assembly instructions allows to program SPARROW in C, using inline assembly instructions as shown in the example of Figure 15 for a saturated vector addition with unsigned 8-bit values using swizzling and masking. As it can be seen, SPARROW can be programmed in a high level way, not very different than vector intrinsics for conventional SIMD extensions such as SSE or NEON.

Another important advantage of reusing the integer register file is that we can use the regular load and store instructions. Inline assembly (and code generation) is only required for the SIMD operations. In order to read and write the four vector components with one instruction, if these components are ordered and aligned in memory, they can be accessed using an integer pointer (lines 14 and 19). Moreover, this means that it's not necessary to specify explicit registers in the inline assembly, nor to modify the compiler register allocator. Notice that by passing the integer variable names to the inline assembly instruction (line 17), the SIMD instruction accesses directly the register in which each of the variable is allocated by the compiler.

```
unsigned char weights[32*32];
unsigned char next_layer[32*32];
unsigned int a, b, result, scr;

/* set the value of the %scr */
scr = 0x0D9D0E; //swizzling_B = 1-2-3-0,
                //swizzling_A = 3-2-2-0,
                //mask_select = 0, mask = 1110

asm("wr %0", %%scr : : "r"(scr));
/* initialise all a components to 0, ie a.xyzw=0 */
a = 0;
/* b.xyzw = weights[0].xyzw */
b = *((unsigned int*) &weights[0]);
asm("nop"); //wait for %scr to commit the write
/* result.xyz = a.xyy + b.zyx */
asm("usadd_ %1, %2, %0": "=r"(result) : "r"(a), "r"(b));
/* next_layer[0].xyzw = result.xyzw */
*((unsigned int*) &next_layer[0])=result;

```

Figure 15: *Example of SPARROW programming in C for SPARC v8 with inline assembly*

4.2 SPARROW support in GCC

In order to include the GCC support for SPARROW we used Gaisler's `bcc-2.2.0` sources which are available open-source in their website [28]. BCC is a GCC-derivative cross-compiler for the LEON processors and the recommended tool for generating code for these processors as it does not require any prior configuration. That been said, there is no support for RISC-V, therefore, this section focuses on the SPARC v8 ISA. Although the GCC compiler is large, in order to just include the assembly instructions for SPARROW only a few files need to be modified. First of all, the encoding of the instructions must be set, and later if any new parameter is used it must added in the machine-code generation and disassembler functions.

In order to set the encoding for the SPARROW instructions, or any that needs to be modified, the first step is to define each instruction fields if not already defined. To do so, in the *bcc-2.2.0/binutils/include/opcode/sparc.h* file, a C-preprocessor macro is used to set the corresponding bits in the instruction to one. For example: `#define SD1(x) (((x) & 0x1f) << 5)` will set the bits in the *sd1* field to the value x.

These definitions can be nested and, for the larger ones which encode all the instruction fields, a negate version is required to validate the instruction generated. That means that for the SPARROW instructions it is necessary to have:

```
#define AI(x, y, z) (OP (2) | OP3(0x09) | SD1(x) | SD2(y) | F3I(z))
```

and

```
#define NAI(x, y, z)(OP (~2) | OP3(~0x09) | SD1(x) | SD2(y) | F3I(z))
```

where *OP* corresponds to bits 30 and 31, *OP3* to the bits 19-24 and *F3I* to the field i as described in Figure 12a.

With the instruction fields defined, in file *bcc-2.2.0/binutils/opcodes/sparc-opc.c* the instructions are defined, in the *sparc_opcode* struct. A new entry must be added for each new instruction, for SPARROW that means to have one for each combination of the SPARROW stages. Each of these entries require the assembly instruction name, the set bits and the negated set bits as defined before, a string encoding the operands and destination, a flag identifying the instruction as an alias or the preferred alias, two flags identifying the hardware requirements, and one for the architecture version. For the hardware and version flags, we have defined identification values for SPARROW, although with the same values as the baseline LEON3 for simplicity.

For reference, the entries for the dot product, with second operand register and immediate, are:

```
{"dot", AI(3, 1, 0), NAI(~3, ~1, ~0), "1,2,d", F_PREF_ALIAS, HWCAP_AISIMD, 0, v8ai}
{"dot", AI(3, 1, 1), NAI(~3, ~1, ~1), "1,X,d", F_PREF_ALIAS, HWCAP_AISIMD, 0, v8ai}
```

where in the 4th parameter, *1* and *2* represent the source registers, *X* a 5-bit immediate and *d* the destination register. Internally, GCC already has defined the bit location of these fields and the encoding for the registers, so there is no need to add those as done for the other fields in the instruction.

Adding support for the SPARROW Control Register just required to copy existing `wr` and `rd` instructions replacing the *OP3* opcode value and using a new character to encode the SCR in the 4th parameter string (`'<'`). This encoding needs later to be resolved in the machine-code generation and disassembler for the compiler to be able to interpret the `%scr` string. The same must be done for the 5-bit immediate as, as explained in Section 3.2, are encoded with common values. This is done in *bcc-2.2.0/binutils/gas/config/tc-sparc.c* and *bcc-2.2.0/binutils/opcodes/sparc-dis.c* files. Lastly, the SCR must be added in the *reg_names* list also in the *sparc-dis.c* file.

4.3 SPARROW support in LLVM

As well as with GCC, Gaisler provides a LLVM-based compiler, however in this case only the binaries are available. Therefore, in order to have a LLVM implementation we used the *LLVM 13.0* distribution [37][38]. This allowed us to provide software support for both SPARC v8 and RISC-V as both are present in the distribution. Since the organization and required modifications for both architectures in LLVM is practically the same and for comparison reasons with the GCC section, we will describe only the SPARC v8 implementation.

However, with the general LLVM distribution we were unable to compile the code as it failed in the linking step. Since all previous steps worked correctly and it was possible to generate an object file with the program, we had to link the object file using the Gaisler-provided GCC, which allowed finally to generate a valid executable. For RISC-V, we encountered an error which caused the program to enter an endless loop when trying to print a value. Thanks to support by the GRLIB Community forum [29], adding a function called *bcc_init70* to initialize a specific memory mapped address for the UART, solved the problem.

As at this point we just required to add the assembly instructions for SPARROW, all the files that need to be modified are in the directory *llvm-project/llvm/lib/Target/Sparc*. To keep the directory organized we created a *SparcInstrSparrow.td* file which is included in the *SparcInstrInfo.td* as the name implies this file contains the definition of all the SPARC instructions. Aside from adding the instructions, and as done for the GCC compiler, the machine-code generation and disassembler are the only other functions that require a modification.

The instructions are described using TableGen format, in which definitions use classes, that like functions, set the instructions bits according to the parameters. Also additional information relevant to the compiler is passed. Classes can be called recursively each one setting different instruction fields furthermore, a *multiclass* allows the definition of multiple instructions. This *multiclass* allows to generate all second stage SPARROW instructions for each of the first stage, avoiding to specifically write all combinations like we had to do in GCC.

We have created two *multiclasses*, one that generates all the combination with the second stage for each first stage definition, and another that generates the register and immediate variations for each of these resulting instructions. Also, like in GCC, we have added aliases for the dot product, however in this case they are not defined as a new instruction with an alias flag, but instead the *InstAlias* class is used.

To include the SCR read and write instructions a new definition needs to be added for each, but they can reuse the generic SPARC v8 class for the `wr` and `rd`. LLVM has a clause option for the instructions, which allows to set what instructions use or define a register. This is used for the SCR which must also be defined in the *SparcRegisterInfo.td* file. The encoding for the 5-bit immediates are found in the *MCTargetDesc/SparcMCCodeEmitter.cpp* and *Disassembler/SparcDisassembler.cpp*.

4.4 SPARROW intrinsics library

With the modifications in the compiler done, it is possible to write code for SPARROW, however there are still some features, like the mask and swizzling which the programmer needs to be aware of. In order to make the setting of the SPARROW Control Register transparent, we have decided to create a library, which is in fact a header file, that contains multiple definitions to simplify working with SPARROW in SIMD intrinsics fashion.

The file contains multiple C-preprocessor macros that convert function-like calls into the inline assembly. For the SCR, a variable is declared which is modified when setting the mask and swizzling and is used to write in the special register. One of the advantages of having a library implemented like this is once again the portability and simplicity. Table 3 shows the existing functions in the SPARROW library.

Function	Description
`__sparrow_readSCR(X)`	Stores the current value of the SPARROW Control Register in the variable `X`
`__sparrow_writeSCR()`	Writes in the SPARROW Control Register the value of `__sparrow_scr`
`__sparrow_set(X,Y)`	Writes in the SPARROW Control Register `X xor Y`
`__sparrow_resetSCR()`	Resets the value of the SPARROW Control Register to the default one with no swizzling nor mask
`__sparrow_setMask(X)`	Sets the mask bits of `__sparrow_scr` to `X`
`__sparrow_setMaskSel(X)`	Sets the mask selection bit of `__sparrow_scr` to `X`
`__sparrow_setSwizzlingA(X,Y,Z,W)`	Sets the first operand swizzling order in `__sparrow_scr` to `X-Y-Z-W`
`__sparrow_setSwizzlingB(X,Y,Z,W)`	Sets the second operand swizzling order in `__sparrow_scr` to `X-Y-Z-W`
`__sparrow_(op1, op2, A, B, C)`	Performs the `op2` reduction on `A op1 B` and stores the value in `C`
`__nop(C, op1, A, B)`	Computes `C = A op1 B`
`__sum(C, op1, A, B)`	Computes a sum over `A op1 B` and stores the result in `C`
`__max(C, op1, A, B)`	Computes the maximum in `A op1 B` and stores the result in `C`
`__min(C, op1, A, B)`	Computes the minimum in `A op1 B` and stores the result in `C`
`__xor(C, op1, A, B)`	Computes a xor reduction over `A op1 B` and stores the result in `C`
`__usum(C, op1, A, B)`	Computes an unsigned sum over `A op1 B` and stores the result in `C`
`__umax(C, op1, A, B)`	Computes the unsigned maximum in `A op1 B` and stores the result in `C`
`__umin(C, op1, A, B)`	Computes the unsigned minimum in `A op1 B` and stores the result in `C`

Table 3: *SPARROW library functions*

In Figure 16 the same code shown in Figure 15 is represented using the SPARROW library. Note that the setting of the SCR, which starts at line 6, requires more instructions, however since those are C-preprocessor macros the compiler can reduce the number of actual generated instructions. On the other hand, although the same behaviour as with the inline assembly could be achieved by using `__sparrow_setSCR(X,Y)`, with this approach the value of each field is more clear and the programmer does not require any knowledge on the SCR organization.

Also, in case of future modifications there is no need to update all the code as just the library would need to be updated. In line 11 it is necessary to include the writing of the SCR as the previous lines were just setting the library internal variable, this is done to reduce the number of accesses which otherwise, would be necessary if each line did the actual write.

```
1  unsigned char weights[32*32];
2  unsigned char next_layer[32*32];
3  unsigned int a, b, result, scr;
4
5  /* set the value of the %scr */
6  __sparrow_setMask(0b1110);
7  __sparrow_setMaskSel(0);
8  __sparrow_setSwizzlingA(3,2,2,0);
9  __sparrow_setSwizzlingB(1,2,3,0);
10
11 __sparrow_writeSCR();
12 /* initialise all a components to 0, ie a.xyzw=0 */
13 a = 0;
14 /* b.xyzw = weights[0].xyzw */
15 b = *((unsigned int*) &weights[0]);
16 asm("nop"); //wait for %scr to commit the write
17 /* result.xyz = a.xyy + b.zyx */
18 __nop(result, a, "usadd", b);
19 /* next_layer[0].xyzw = result.xyzw */
20 *((unsigned int*) &next_layer[0])=result;
21
```

Figure 16: *Example of SPARROW programming in C for SPARC v8 with the SPARROW library*

5 Evaluation

In this Chapter we evaluate SPARROW in multiple aspects. In Section 5.1 we assess the hardware overhead of our design under various configurations of baseline processors, as well as for different FPGAs. Moreover, we use different configurations in order to show that our implementation has an area optimised benefit both for FPGA implementations, as well as for ASICs. In Section 5.2, we evaluate the performance benefit of SPARROW using various benchmarks and comparing it to a similar vector design for micro-controllers.

5.1 Hardware Overhead

5.1.1 LEON3-MINIMAL: Artix-7 FPGA

As mentioned in Section 2.3 there are different top-level designs for the LEON3 provided in the GRLIB library, of those, the LEON3-MINIMAL is designed for the Artix-7 (xc7a100tcsg324-2) FPGA. This design implements the smallest configuration of the LEON3 with 8KB direct-mapped instruction and data caches, clocked at 100MHz. Since at the moment this was the design we had managed to use in simulation, we synthesised the LEON3 with SPARROW for this FPGA in order to evaluate the hardware cost that the module would have.

To do so we used Xilinx Vivado 2020.1, using different synthesis strategies, however, in none of those the SPARROW managed to fit within the timing constraints. For this reason we implemented the changes described in Section 3.2.1 as well as adapted the file structure as explained in [35]. With such optimisations we did not require to reduce the processor frequency and obtained meaningful data for the hardware overhead evaluation. To get more information we also implemented the same design with cache disabled.

	Artix-7	LEON3-minimal		SPARROW	
	Available	**Cache enabled**	**Cache disabled**	**Cache enabled**	**Cache disabled**
LUT	63400	5743 (9.06%)	5333 (8.41%)	7739 (12.21%)	7197 (11.35%)
LUTRAM	19000	266 (1.4%)	266 (1.4%)	2 (0.01%)	2 (0.01%)
FF	126800	2649 (2.09%)	2568 (2.03%)	3082 (2.43%)	2998 (2.36%)
BRAM	135	9 (6.67%)	4 (2.96%)	11 (8.15%)	5 (3.7%)
DSP	240	1 (0.42%)	1 (0.42%)	1 (0.42%)	1 (0.42%)
IO	210	67 (31.9%)	67 (31.9%)	67 (31.9%)	67 (31.9%)
BUFG	32	1 (3.13%)	1 (3.13%)	2 (6.25%)	2 (6.25%)
PLL	6	0 (0%)	0 (0%)	1 (16.67%)	1 (16.67%)

Table 4: *LEON3 resource utilization comparison for the Artix-7 FPGA*

As we can see from Table 4, SPARROW has a very small relative increase over the original LEON3 when it is implemented on the Artix-7. We notice an increment of only 35% over the baseline LEON3-MINIMAL both implemented with caches and without caches. It is worth noting that LEON3 uses block RAM modules available in FPGAs for the implementation of caches and register files, that's why the difference between the resources required with caches enabled or disabled are small. In absolute terms, SPARROW uses only 2000 LUTs and 450 FF, in comparison, Johns and Kazmierski [1] present a vector unit implementation for a RISC-V embedded processor, which doubles the resource utilisation over its baseline. However, unlike SPARROW they implement vectors operations up to 32-bit, not only 8-bit ones. As another indication of the cost of SPARROW, the floating point unit (GRFPU) for LEON3 from Gaisler's GRLIB [39] costs 4600 LUTS and 2 BRAM blocks, and its area optimised one (GRFPUlite) has comparable cost with ours (2000 LUTS and 2 BRAM blocks).

5.1.2 LEON3-ZCU102: Zynq Ultrascale+ FPGA

In order to have the LEON3 working on a FPGA we did not have an Artix-7 FPGA. Instead we needed a design for the Xilinx Zynq Ultrascale+ ZCU102 Evaluation Board (xczu9eg-ffvb1156-2-e) FPGA which is the board we had available at the CAOS research group at Barcelona Supercomputing Center (BSC). Since there is no top design from Gaisler's GRLIB library for the Zynq Ultrascale+, we had to create a top-level design using the minimum components for a functional processor.

To do so we requested help from Mr. David Steenari, who serves as a Technical Officer in the GPU4S project which we participate. Mr. Steenari has supervised the Master thesis [2], in which they also implemented the LEON3 for the Zynq Ultrascale+. Due to confidential IPs used in that thesis, the amount of help we could receive was limited, however with the provided information we managed to have a working design. This design, which is based on the LEON3-MINIMAL, also has 8KB direct-mapped instruction and data caches, and is clocked at 100MHz. Similar to the Artix-7, we also evaluated the design in a cache-less version to compare with.

	Zynq Ultrascale+	LEON3		SPARROW	
	Available	**Cache enabled**	**Cache disabled**	**Cache enabled**	**Cache disabled**
LUT	274080	9333 (3.41%)	8709 (3.18%)	11792 (4.3%)	11251 (4.11%)
LUTRAM	144000	292 (0.2%)	292 (0.2%)	292 (0.2%)	292 (0.2%)
FF	548160	6346 (1.16%)	6145 (1.12%)	6553 (1.2%)	6353 (1.16%)
BRAM	912	9.5 (1.04%)	4 (0.44%)	9.5 (1.04%)	4 (0.44%)
DSP	2520	4 (0.16%)	4 (0.16%)	4 (0.16%)	4 (0.16%)
IO	328	31 (9.45%)	31 (9.45%)	31 (9.45%)	31 (9.45%)
BUFG	404	1 (0.25%)	1 (0.25%)	1 (0.25%)	1 (0.25%)
PLL	8	1 (12.5%)	1 (12.5%)	1 (12.5%)	1 (12.5%)

Table 5: *LEON3 resource utilization comparison for the Zynq Ultrascale+ FPGA*

As we can see from the resource utilization results in Table 5 when implemented in the Zynq Ultrascale+ SPARROW has a small impact with respect to the baseline processor. In this case the increment is only of 26% in the cached design and 30% in the cache-less one. In absolute terms, SPARROW requires 2500 LUTs and 200 FF. As we see the percentage is smaller than in the Artix-7 implementation, that is explained due to the increased resource utilization in the ZCU102 board. The discrepancies in the absolute cost of sparrow between the two FPGAs can be explained with the internal optimizations of the synthesis tool for each board resources.

	LEON3				SPARROW			
	Cache enabled		Cache disabled		Cache enabled		Cache disabled	
	Single RF	**Extra RF**	**Single RF**	**Extra RF**	**Single RF**	**Extra RF**	**Single RF**	**Extra RF**
LUT	15183 (5.54%)	15489 (5.65%)	10252 (3.74%)	10576 (3.86%)	17631 (6.43%)	17939 (6.55%)	12795 (6.67%)	13118 (4.79%)
LUTRAM	4916 (3.41%)	5156 (2.58%)	1572 (1.09%)	1812 (1.26%)	4916 (3.41%)	5156 (3.58%)	1572 (1.09%)	1812 (1.26%)
FF	6705 (1.22%)	6748 (1.23%)	6311 (1.15%)	6354 (1.16%)	6909 (1.26%)	6953 (1.27%)	6517 (1.19%)	6560 (1.20%)
BRAM	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

Table 6: *LEON3 resource utilization for synthesis using LUT-RAM blocks similar to [40] and with an additional register file*

However, radiation-hardened-by-design space FPGAs such as the Xilinx Virtex 5 XQR5V FX130T FPGA (also known as V5QV) only offer radiation hardening for LUT-RAMs, while BRAM blocks only have ECC protection. Therefore, implementing the cache and the register files using LUT-RAMs instead of BRAMs can increase the LEON3 reliability [40]. In order to validate SPARROW's resource overhead in that scenario and to give a rough indication of its relative area overhead in the case of an ASIC implementation, we have evaluated the resource utilization of the same designs using LUT-RAMs in Table 6. In Tables 5 and 6 we show the use of DRAM instead of BRAM for the RAM memory of the processor, to focus on just the processor's LUT-RAM utilization.

We can see that in this case the relative cost of SPARROW over the baseline LEON3 is 16% when the cache is present, and 25% otherwise. Note that our baseline processor is a small LEON3 configuration, so with a larger one, our relative hardware overhead is expected to be even smaller.

Moreover, in order to show the exact hardware savings of SPARROW thanks to the reuse of the integer register file, we implemented a version of both the baseline LEON3 and LEON3 with SPARROW using an extra register file, indicated as "Extra RF" in Table 6. According to the results, the cost of the extra vector register file would be around 310 LUTs, 240 LUT-RAMs and 43 FFs. This cost corresponds to 12% of the SPARROW cost in LUTs and 21% in FFs. Therefore, the hardware savings thanks to this decision are consistent with the overhead of the vector register file in ASIC implemented vector processors [8] [13].

5.1.3 NOELV-ZCU102: Zynq Ultrascale+ FPGA

The situation for the FPGA implementation of the NOEL-V was similar to the LEON3. There was no default support for the Zynq Ultrascale+. In fact, due to the recent release and the larger size of the NOEL-V with respect to the LEON3, there is only support for a few number of high-end boards. Nonetheless, as done previously with the LEON3, and even with a different top-level design and different structure we managed to successfully implement the NOEL-V for the Zynq Ultrascale+ ZCU102. As in the previously shown results the implementation is done using DRAM to focus on the core resource utilization.

As explained in Section 2.3.2 there are four basic configurations available. We decided to test the tiny and general-purpose implementations, both with 32-bits and single-issue. The tiny configuration (TIN32) is forced to have a single-issue pipeline and has the cache disabled, on the other hand, the general-purpose implementation (GPP32) has an optional dual-issue pipeline, which in this case was disabled, and 16KB instruction and data caches. Furthermore the GPP32 configuration includes a memory management unit (MMU), physical memory protection (PMP) and a floating point unit (FPU), which are missing in the TIN32.

	Zynq Ultrascale+	NOEL-V		SPARROW	
	Available	**TIN32**	**GPP32**	**TIN32**	**GPP32**
LUT	274080	22360 (8.16%)	46293 (16.89%)	25258 (9.22%)	48533 (17.71%)
LUTRAM	144000	1591 (1.1%)	1959 (1.36%)	1591 (1.1%)	1959 (1.36%)
FF	548160	19538 (3.56%)	28826 (5.26%)	19815 (3.61%)	28694 (5.23%)
BRAM	912	25.5 (2.8%)	34.5 (3.78%)	25.5 (2.8%)	34.5 (3.78%)
DSP	2520	7 (0.28%)	9 (0.36%)	7 (0.28%)	9 (0.36%)
IO	328	55 (16.77%)	55 (16.77%)	55 (16.77%)	55 (16.77%)
BUFG	404	8 (1.98%)	9 (2.23%)	9 (2.23%)	8 (1.98%)
PLL	8	1 (12.5%)	1 (12.5%)	1 (12.5%)	1 (12.5%)

Table 7: *NOEL-V resource utilization comparison for the Zynq Ultrascale+ FPGA*

The results presented in Table 7 compare the NOEL-V implementation with SPARROW for both configurations with the baseline NOEL-V. The first important consideration when looking at the results is the base difference between the TIN32 and GPP32 configurations, having the GPP32 approximately twice the number of LUTs and a considerable amount of other resources like FF.

Regarding the overhead induced by SPARROW, the absolute cost of the module is consistent with the results obtained for the LEON3, with a 2500 cost in LUTs for both configurations and 300 in FF in the TIN32 implementation. Note that the GPP32 baseline implementation has a higher number of flip-flops than the SPARROW counterpart. This can be explained by the inner optimizations of the synthesis tool. As can be expected, this increment for the TIN32 represents a 13% utilization increase, which due to the baseline size difference is reduced to 5% in the GPP32 implementation.

This results further validate the decision making when designing SPARROW, as the module has a small impact on hardware overhead of the processor. As expected when comparing with larger configurations the relative impact of SPARROW is minimal while still providing the same advantages.

5.2 Performance

5.2.1 LEON3 simulation

During the first stages of the project, all evaluation of SPARROW was done in simulation. Before the addition of the software support, the generation of tests had to be done by manipulating the instruction bits in the memory dump file. This was done by adding a dummy instruction which later with a script would be substituted by the correct one. As can be expected, this approach has many drawbacks as the new instruction needs to encoded by hand with each modification of the code as the compiler could change the used registers.

Even so, at this stage we had no access to an FPGA nor successfully implemented any of the FPGA top level designs. Furthermore, it is simpler to run a simulation than having to program the FPGA and generate the bitstream after each modification, making simulation a great tool for debugging in the earlier stages. On the other hand, simulation is slower and not as accurate as the execution on a real board, therefore, tests executed had to be limited in length and the obtained results had to be later validated in the FPGA implementation.

In order to perform the bigger tests in simulation we used a remote server at the CAOS group at BSC to run the program. This way we could leave the simulation during days and later get the results. Even so, if the print option was set to view the results the execution time was even larger, reason for which the validation of the output was not possible for the bigger sizes. That being said, we could verify that for the smaller versions the program run without issues and provided the expected output.

GRLIB supports different simulation tools and provides a Makefile to easily run each of these. However, some of the simulators require a paid license and many of the design require proprietary IPs which caused a lot of errors. The only working design we were able to simulate without requiring additional modifications was the LEON3-MINIMAL design. We used GHDL for simulation as I had prior experience with it. With GHDL a waveform can be generated to view the signals at each time unit, there are different options: `-vcd`, `-ghw` or `-fst`. Of those we found that the fst file is the most compressed one, a necessary condition for viewing the larger tests waveforms. Additionally, we used the `-read-wave-opt=filename` option to specify the signals we wanted to view, thus reducing greatly the size of the waveform.

For the evaluation of the proposal, we compared the results with similar approaches in the literature targeting resource constrained embedded systems. Since to our knowledge there are no implementations in the literature of other vector accelerator proposals for AI and in particular for space that we can compare against, we compare SPARROW with [1] which is the vector processor design in the literature which is closer to ours.

Although their design supports vector operations up to 32-bit per element (one 32-bit, 2 16-bit or 4 8-bit operations in a single cycle), they only evaluated their proposal with 8-bit programs: matrix multiplication, Greyscale conversion of an RGB image and an edge detection filter. From these operations matrix multiplication and the convolution filter are AI-related, while the greyscale conversion could be an operation that could be applied as a part of an inference processing. This makes their design ideal for comparison with SPARROW.

The authors were kind enough to provide me with information and their software implementations in RISC-V assembly using vector extensions in order to perform a fair comparison. Their design runs at 50MHz which is half of SPARROW's and does not feature a cache. In order to perform a fair comparison, we run the experiments with a LEON3-MINIMAL baseline and a SPARROW implementation without cache.

Program	Data size	LEON3	SPARROW	Speedup
Matrix Mult.	120×120	69.457.937	13.731.096	5.1×
Greyscale	256×256	1.405.970	799.006	1.8×
Filter	256×256	15.945.897	4.993.320	3.2×
Cifar-10	32×32	1.682.684	291.240	5.8×
Polynomial	2048	51.442	13.041	3.94×

Table 8: *Simulation results for the LEON3 with 8KB cache*

Program	Data size	LEON3	SPARROW	Speedup
Matrix Mult.	120×120	709.738.072	138.909.319	5.1×
Greyscale	256×256	15.236.721	4.425.816	3.4×
Filter	256×256	207.739.142	67.870.004	3×
Polynomial	2048	350.265	79.417	4.4×

Table 9: *Simulation results for the LEON3 with cache disabled*

The results are shown in the Table 9. For each experiment we report the number of processor cycles, as well as the obtained speed-up compared to the LEON3-MINIMAL baseline. The resulting speed-ups are similar to those shown in [1] (Matrix multiplication 5.8×, Greyscale 2.7× and Filter 3.2×). The small differences can be explained from the fact that the baseline processors use different ISAs and different implementations. However, SPARROW obtains the same performance boost over these 8-bit workloads using only a fraction of the hardware cost of [1] and it is able to operate at double frequency.

Table 8 shows the speed-ups obtained when using a cache. As expected, the use of the cache reduces almost an order of magnitude the execution time. However, the speed-up compared to the original design remained the same or increased slightly for the two applications that exhibit some data reuse such as the convolution filter and the matrix multiplication. On the other hand, a streaming program such as greyscale benefits more by the introduction of the cache, which prefetches the next 9 pixels of the image in its 32 bit cache lines, than SPARROW's SIMD vector processing unit, in which case it is processing the 3 components of the pixel together.

To have an approximate comparison of our speed-ups with the performance benefit provided by other vector architectures such as ARM's NEON over their scalar baselines, we have computed the 2nd degree polynomial equation presented in [41]. Jie and Kapre [41] mention that this operation executed with NEON with high loop trip counts over uncached 8-bit data provides a speed-up of 3.7× over the ARM scalar code. SPARROW provides a speed-up of 4.4× with respect to the baseline when regular 8-bit instructions are used and 8.32× when saturation arithmetic is used for both designs.

If the cache is enabled, the speed-up drops slightly to 3.94×, similar to the cache of the greyscale. Again probably the reason in this case is that the benefit provided by the instruction cache, which prefetches and retains 4 instructions in each cache-line, and can hold the instructions of this small loop, is slightly higher than the benefit provided by the vector unit, which operates in 4 computations at the time. However, in both cases we obtain a speed-up similar to the one provided by a SIMD vector extensions implemented on an ASIC design, for only a fraction of its relative area over the baseline design.

Finally, although the previously presented benchmarks from [1] can be already considered good examples of ML applications, they don't exhibit any data reuse, which can further show the benefit of SPARROW, nor are relevant for space. For this reason we have ported a complex space relevant inference application based on CIFAR-10 from the open source GPU4S Bench benchmark suite [42] which was recently released [43][44] and which is done at the BSC CAOS group. Interestingly, part of the code for the matrix multiplication and the convolution filter tests that were developed to compare with [1] has been reused to elaborate the layers of this application

Its neural network layers include convolution, relu, max pooling and matrix multiplication over an initial 32×32 input matrix. All of the layers are a good fit for SPARROW as the obtained speed-up is of 5.8×. This impressive result is higher than the individual speed-ups obtained with single layers, thanks to the data reuse from one layer to the other.

5.2.2 LEON3 FPGA implementation

With the Zynq Ultrascale+ available and the compiler support allowing for the generation of SPARROW executable programs we were able to obtain more detailed results and for larger sizes. To be able to measure the performance in the FPGA we used GRMON [45], a tool provided by Cobham Gaisler which allows to connect to the LEON3 core and debug it. The L3STAT module can be used to set counters and breakpoints allowing the measurement of the program execution time among other utilities such as cache misses.

As mentioned in the previous section one of the limitations of simulation was the real time cost of each execution. In the FPGA that is no longer the case, instead the limit is set by the available RAM. We were not able to use the board DRAM and was for this reason we were forced to use BRAM. The maximum BRAM we could fit in the design was 3MB. With the FPGA implementation we were also able to print the output of the larger tests and verify the correctness of their output.

Program	Data size	LEON3	SPARROW	Speedup
Matrix Mult.	120×120	31.550.354	3.653.057	8.64×
Greyscale	256×256	924.204	301.103	3.07×
Filter	256×256	10.242.140	3.033.531	3.37×
Cifar-10	32×32	1.063.695	190.391	5.58×
Polynomial	2048	27.298	6.307	4.33×

Table 10: *FPGA results for the LEON3 with 8KB cache*

Program	Data size	LEON3	SPARROW	Speedup
Matrix Mult.	120×120	117.310.797	12.632.882	9.3×
Greyscale	256×256	3.221.620	876.633	3.67×
Filter	256×256	39.019.342	11.855.218	3.3×
Cifar-10	32×32	3.784.906	649.618	5.83×
Polynomial	2048	88.145	19.537	4.5×

Table 11: *FPGA results for the LEON3 with cache disabled*

With the same experiments as with the simulation, we have achieved the speed-ups shown in Table 10 and Table 11 for the cache and no-cache versions of the processor. In general the results are similar to those obtained before, however are slightly better in the FPGA implementation. More specifically, the matrix multiplication has increased to 9.3× speed-up with cache disabled. Thus further demonstrates the correctness of the AI-targeted design of SPARROW as it demonstrates high benefits, even more when considering the low-cost of its implementation.

Although due to SPARROW's design it would seem that the maximum speed-up is 7× (4 parallel computations in the first stage and three reduction operations in the second) the actual speed-up is higher than this threshold. This can be easily explained thanks to the saturation: in the SPARROW version is not necessary to have any additional cycles to avoid the possible overflow while in the baseline processor this has to be achieved by using conditionals and altering the program flow.

On the other hand, the speed-up of the CIFAR-10 inference application is slightly smaller compared with the simulation, nonetheless the achieved speed-up is still considerable as it is over 5.5×. Also in the FPGA version we were able to compute the speed-up with the disabled cache, which was not possible in simulation, and the resulting speed-up is on pair of what could be expected.

Since the time to perform a test in the FPGA is much smaller than in simulation, we were able to launch different configurations to have more data to work with. We also compared the speed-up when using GCC, which was the compiler used for all previous presented tests, with LLVM. The results are shown in the Tables from 12 to 17, and a summary of these experiments are plotted in Figures from 17 to 22. The charts compare the speed-up for each input data size with GCC and with LLVM, and include also the geometric mean which is shown in the *Geomean* bar.

GCC			
Size	LEON3	SPARROW	Speed-up
4×4	498	243	2.04×
8×8	2420	905	2.67×
16×16	27455	6028	4.55×
32×32	240631	41027	5.86×
64×64	1879345	328034	5.72×
128×128	15856659	4576240	3.46×
256×256	128968634	36562425	3.52×
512×512	1073844043	299990787	3.57×

LLVM			
Size	LEON3	SPARROW	Speed-up
4×4	553	234	2.36×
8×8	2674	932	2.86×
16×16	21023	6809	3.08×
32×32	190960	45371	4.20×
64×64	2153746	346750	6.21×
128×128	18002989	5641198	3.19×
256×256	145942541	45016602	3.24×
512×512	1208848790	367362081	3.29×

Table 12: *Results in cycles for the matrix multiplication program in the LEON3 implemented in the Zynq Ultrascale+*

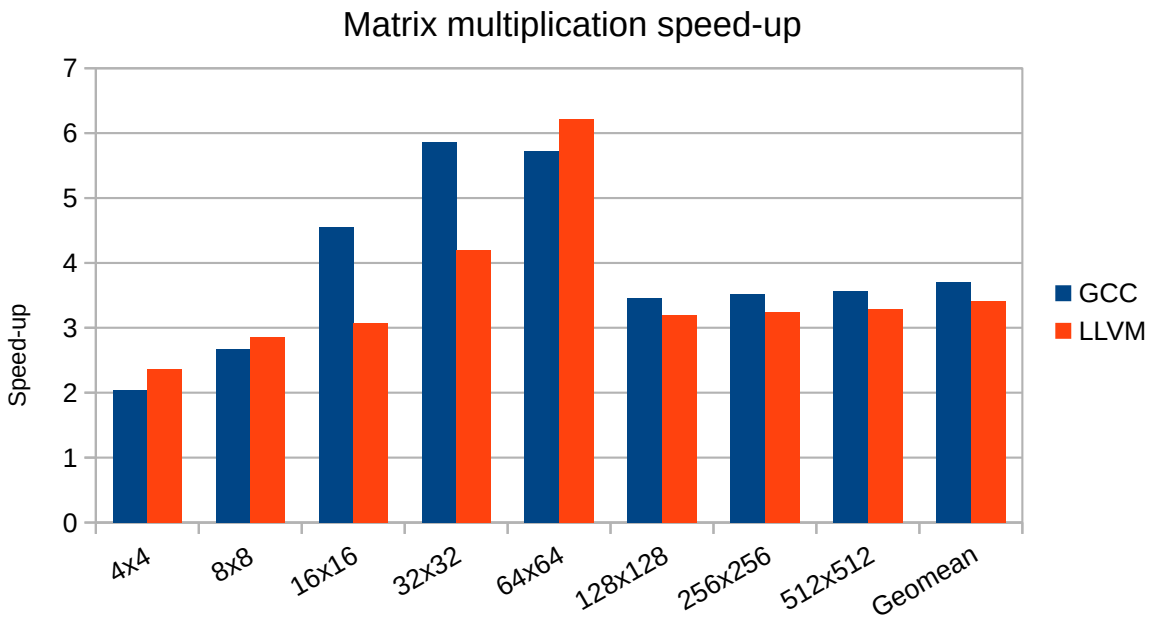


Figure 17: *Matrix multiplication speed-up for the LEON3 implemented in the Zynq Ultrascale+*

GCC			
Size	LEON3	SPARROW	Speed-up
4×4	1181	243	4.86×
8×8	8397	905	9.27×
16×16	66033	6028	10.9×
32×32	628409	41027	15.3×
64×64	4815898	328034	14.6×
128×128	38656531	4576233	8.44×
256×256	308400023	36562437	8.43×
512×512	2498336539	299990829	8.32×

LLVM			
Size	LEON3	SPARROW	Speed-up
4×4	1246	234	5.32×
8×8	6988	932	7.49×
16×16	51940	6809	7.62×
32×32	695384	45371	15.3×
64×64	6160357	346750	17.7×
128×128	52905988	5641198	9.37×
256×256	436788127	45016615	9.70×
512×512	3579246299	367362116	9.74×

Table 13: *Results in cycles for the matrix multiplication program in the LEON3 implemented in the Zynq Ultrascale+ with saturation enabled*

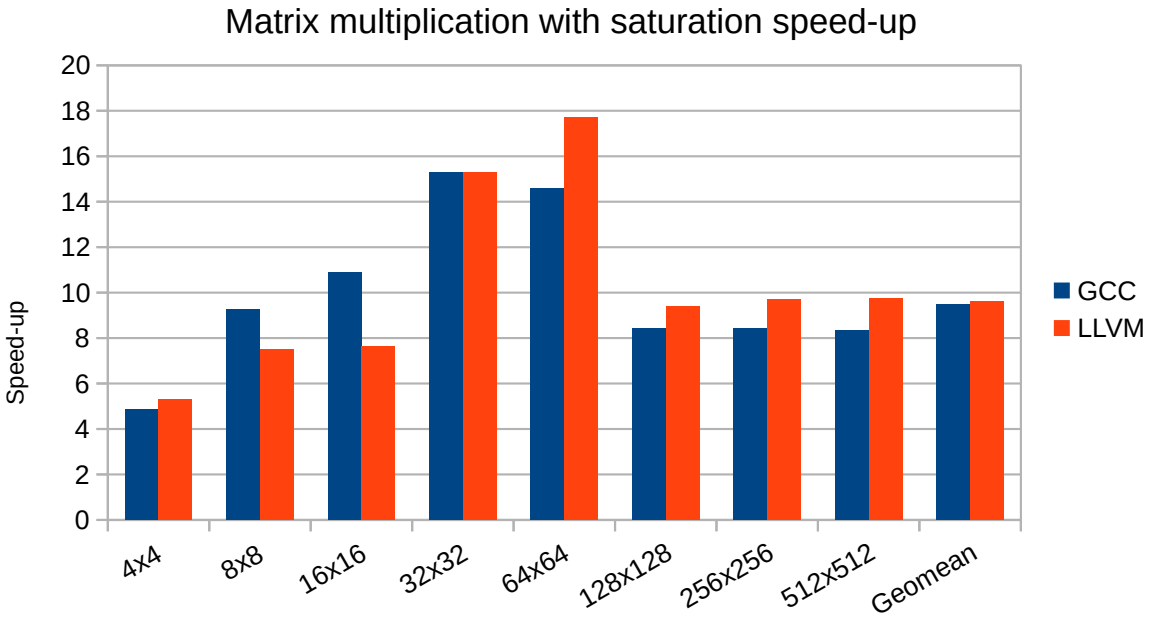


Figure 18: *Matrix multiplication with saturation speed-up for the LEON3 implemented in the Zynq Ultrascale+*

GCC			
Size	LEON3	SPARROW	Speed-up
4×4	304	106	2.86×
8×8	808	351	2.30×
16×16	3010	1005	2.99×
32×32	14752	3852	3.82×
64×64	58284	15000	3.88×
128×128	231724	75823	3.05×
256×256	924206	301103	3.06×
512×512	3691564	1200176	3.07×

LLVM			
Size	LEON3	SPARROW	Speed-up
4×4	302	91	3.31×
8×8	832	296	2.81×
16×16	3032	1023	2.96×
32×32	16735	3886	4.30×
64×64	66341	20139	3.29×
128×128	264229	79659	3.31×
256×256	1054757	316973	3.32×
512×512	4214827	1264684	3.33×

Table 14: *Results in cycles for the grayscale conversion program in the LEON3 implemented in the Zynq Ultrascale+*

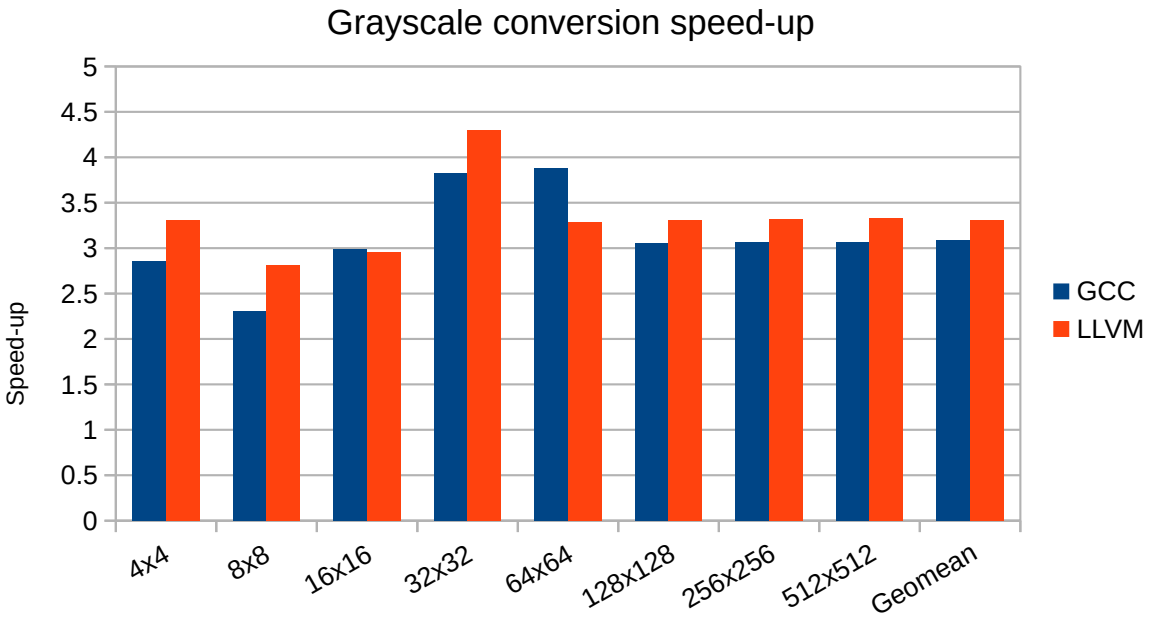


Figure 19: *Grayscale conversion speed-up for the LEON3 implemented in the Zynq Ultrascale+*

GCC			
Size	LEON3	SPARROW	Speed-up
4×4	1531	837	1.82×
8×8	7952	3183	2.49×
16×16	36545	11929	3.06×
32×32	152517	47392	3.21×
64×64	625255	189434	3.30×
128×128	2541092	758075	3.35×
256×256	10242140	3033531	3.37×
512×512	41126135	12137151	3.38×
1024×1024	164818653	48555196	3.39×

LLVM			
Size	LEON3	SPARROW	Speed-up
4×4	1825	786	2.32×
8×8	8814	3778	2.33×
16×16	39821	14869	2.67×
32×32	169741	59471	2.85×
64×64	704193	238261	2.95×
128×128	2866301	954130	3.00×
256×256	11567599	3818898	3.02×
512×512	46475231	15280787	3.04×
1024×1024	186300343	61133989	3.04×

Table 15: *Results in cycles for the edge detection filter program in the LEON3 implemented in the Zynq Ultrascale+*

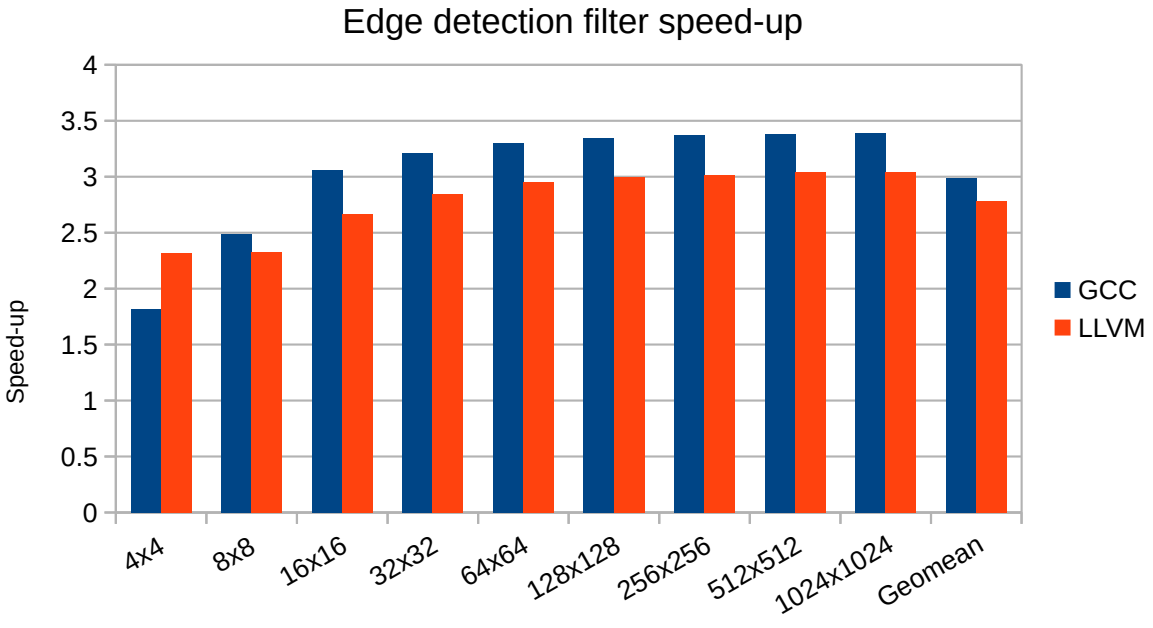


Figure 20: *Edge detection filter speed-up for the LEON3 implemented in the Zynq Ultrascale+*

GCC			
Size	LEON3	SPARROW	Speed-up
4×4	91	35	2.60×
8×8	157	48	3.27×
16×16	293	75	3.90×
32×32	460	143	3.21×
64×64	876	259	3.38×
128×128	1728	417	4.14×
256×256	3432	809	4.24×
512×512	6850	1603	4.27×
1024×1024	13666	3171	4.30×

LLVM			
Size	LEON3	SPARROW	Speed-up
4×4	96	36	2.66×
8×8	166	57	2.91×
16×16	295	82	3.59×
32×32	569	133	4.27×
64×64	941	256	3.67×
128×128	1857	447	4.15×
256×256	3679	862	4.26×
512×512	7338	1705	4.30×
1024×1024	14666	3401	4.31×

Table 16: *Results in cycles for the polynomial program in the LEON3 implemented in the Zynq Ultrascale+*

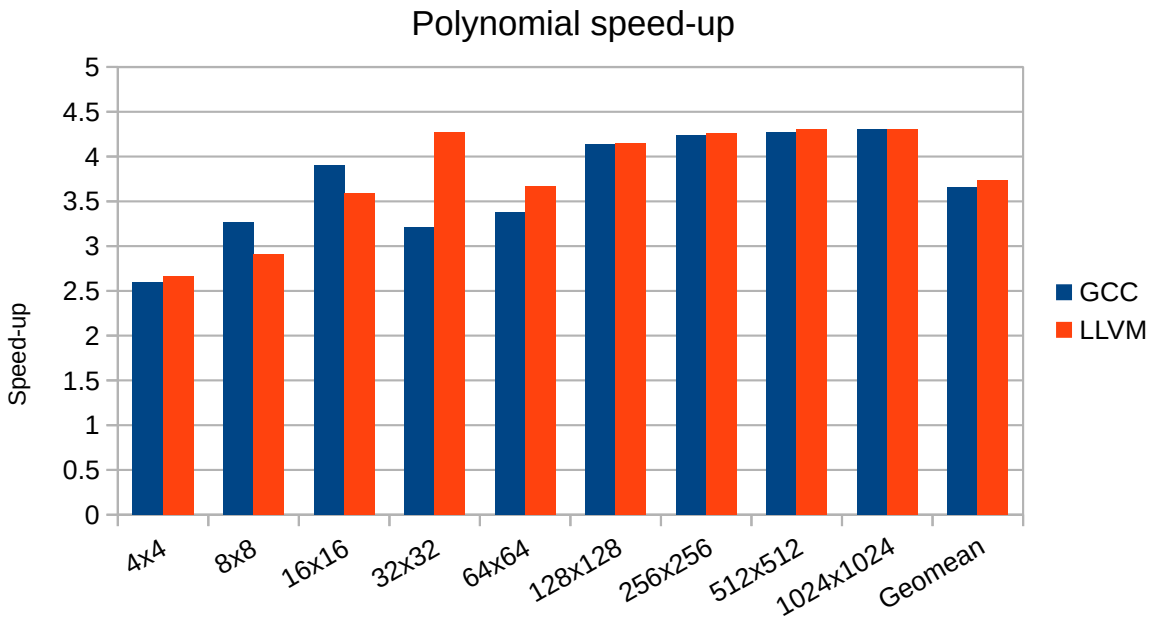


Figure 21: *Polynomial speed-up for the LEON3 implemented in the Zynq Ultrascale+*

GCC			
Size	LEON3	SPARROW	Speed-up
4×4	301	35	8.6×
8×8	586	48	12.2×
16×16	1156	75	15.4×
32×32	1494	143	10.4×
64×64	2907	259	11.2×
128×128	5759	417	13.8×
256×256	11463	809	14.2×
512×512	22863	1603	14.3×
1024×1024	45637	3171	14.4×

LLVM			
Size	LEON3	SPARROW	Speed-up
4×4	326	36	9.1×
8×8	621	57	10.8×
16×16	1192	82	14.5×
32×32	1148	133	8.6×
64×64	2198	256	8.6×
128×128	4309	447	9.6×
256×256	8531	862	9.9×
512×512	16979	1705	10.0×
1024×1024	33893	3401	10.0×

Table 17: *Results in cycles for the polynomial program in the LEON3 implemented in the Zynq Ultrascale+ with saturation enabled*

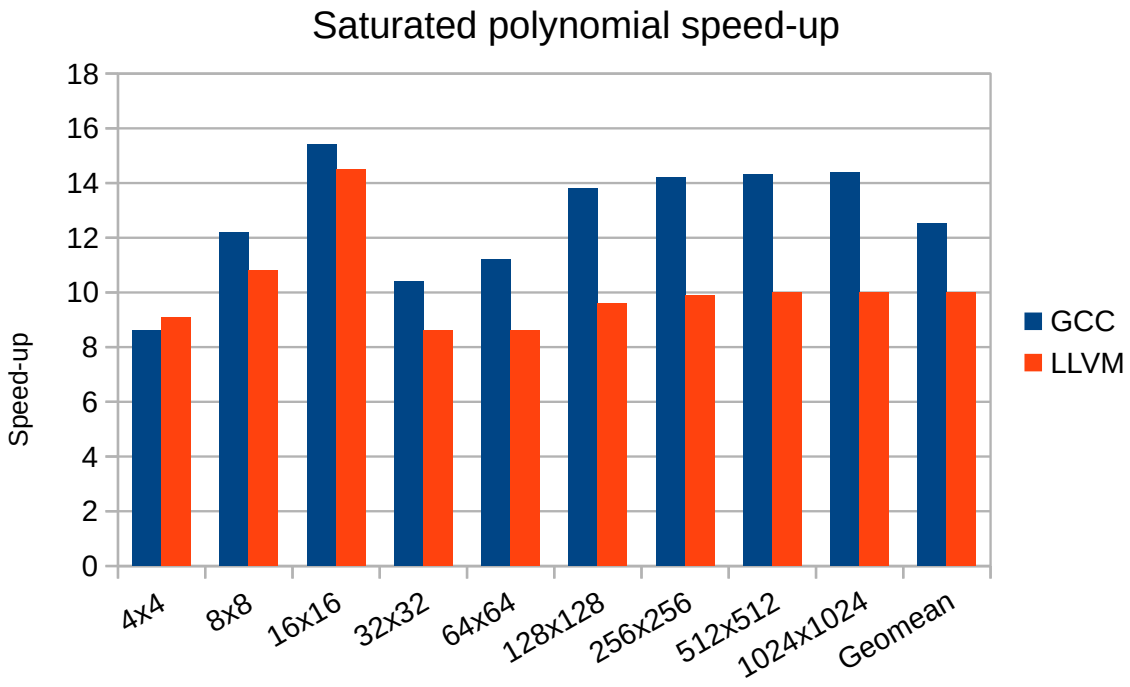


Figure 22: *Saturated polynomial results for the LEON3 implemented in the Zynq Ultrascale+*

The results depicted in Table 13 and Table 17, for matrix multiplication and polynomial computation respectively, evaluate the impact of saturation. We observe that the saturation results in speed-ups over 10× and 10× on overage for various sizes. If the focus is shifted to the matrix multiplication in Tables 12 and 13, it is clear that the highest speed-up is achieved for the 32×32 and 64×64 matrices. Furthermore, the results are close to 6× when no saturation is involved, which is close to the theoretical maximum of 7×.

In order to compare the speed-up from GCC and LLVM, in Figure 23 the geometric means for each program are presented as normalized results. This simplifies the comparison regardless of the actual speed-up. A larger percentage of the blue bar means that GCC provides better performance than LLVM and vice versa. In the majority of the programs both GCC and LLVM have similar speed-ups as they are close to the center. The only notable exception is the saturated polynomial. However, the total geometric mean across the different programs proves that the speed-ups obtained with GCC are marginally better. Furthermore, if the absolute number of cycles is evaluated in the presented tables, it is shown that GCC, both for the baseline LEON3 as with SPARROW, has better performance than LLVM.

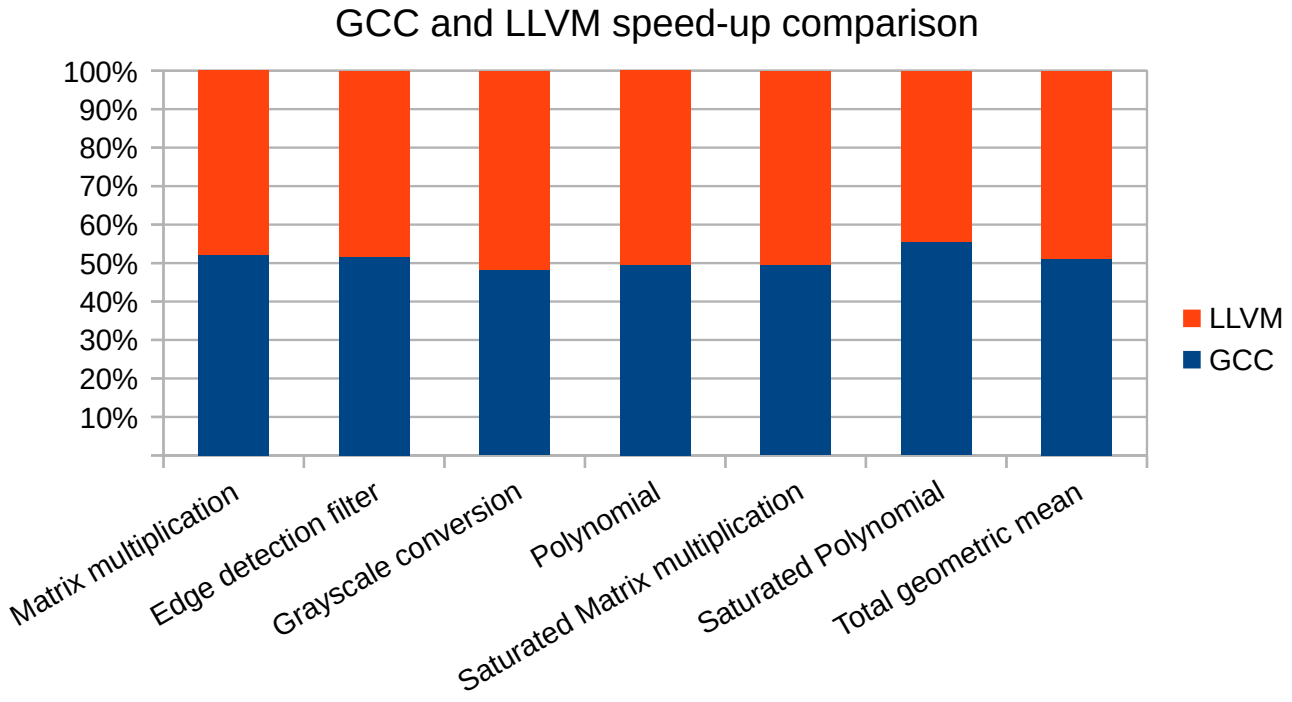


Figure 23: *Comparison of the speed-ups for all the programs in GCC and LLVM*

With the geometric means we have computed the speed-ups of SPARROW, which provides a total 4.96× speed-up in GCC and 4.73× in LLVM. This value, however, includes both saturated and non-saturated programs. For a more meaningful results, the non-saturated programs have a geometric mean of 3.35× in GCC and 3.29× in LLVM. On the other hand, for the two saturated programs the geometric mean is 10.9× and 9.8× for GCC and LLVM respectively.

5.2.3 NOEL-V simulation

As explained before, the NOEL-V design is relatively recent and there are few implementations in the GRLIB library. We decided to use the NOELV-GENERIC design, which is the entity used for all the available FPGA implementations, and therefore it is independent on the proprietary libraries of any FPGA manufacturer.

However, since the design is recent the simulation with GHDL had not been fully supported or at least not in the GPL GRLIB 2021.2 distribution. When trying to execute the simulation in the default GRLIB a list of errors occur, some of them related to missing components that are part of the commercial distribution. Fortunately, none of them is actually mandatory as they can be disabled. By solving all the existing errors we were able to successfully simulate the NOEL-V with GHDL.

For the NOEL-V we decided to execute the simulations in the GPP32 single-issue configuration. The performed tests are the same for the LEON3 in order to compare with [1]. In Table 18 the speed-up for each program is shown. The results demonstrate similar speed-ups as with the LEON3, the edge detection filter has, however, a lower speed-up than expected.

Program	Data size	LEON3	SPARROW	Speedup
Matrix Mult.	120×120	27.945.510	5.529.004	5.05×
Greyscale	256×256	742.721	293.152	2.8×
Filter	256×256	5.099.629	3.841.966	1.33×

Table 18: *Simulation results for the NOEL-V*

In absolute terms, it is notable that NOEL-V has higher performance with respect to the LEON3. This can be explained easily by the architectural differences of both processors such as the late ALU and late BRANCH or the bigger cache which are present in the NOEL-V. Being able to provide similar results in both cases proves the advantage of SPARROW which is the acceleration speed-up relative to the processor performance, therefore providing good speed-ups even to already higher-performant processors.

6 Lessons learnt

In this Chapter we briefly provide our lessons learnt from our experience in the development of this Master thesis from hardware design approaches, as well as by adding software support for SPARROW to two different compiler toolchains.

6.1 Hardware design and VHDL

As the SPARROW module has progressed during the development of the Thesis, in the latter stages new functionalities were added. However, during the implementation of the LEON3 in the FPGA, with the objective of meeting the timing constrains within a cycle, we were forced to re-do part of the module using lower-level descriptions in order to reduce the logic of the functionalities.

However, the results did not improve for the timing analysis, in some cases the critical path was even longer. This made evident one of the most important points we have learned during the development of this project, which is that the synthesis tool has already great logic optimizations. Having a more complex design in order to reduce the resulting logic or number of gates, does not, in general, improve the post-synthesis results. Instead, and as presented in [35] having a clear and organized design is better.

In many cases we were hesitant to use the `integer` type for signals and instead used `std_logic_vector`, but when performing arithmetic operations there is no performance improvement and instead the code becomes unintelligible. Nonetheless, there are exceptions to this rule, as shown in Section 3.2.1, when the computation can be included in already existing logic or when not the full result is required, so that a custom approach can provide more benefits. The same is true if a specific resource is available which the synthesis tool is unaware of.

6.2 GCC vs LLVM

Having worked with both GCC and LLVM for the development of the software support for SPARROW has allowed us to compare, not only the performance, but also the experience when working with each one. It is worth noting that we had no prior experience on working on either of the two toolchains prior to this project. In Section 5.2.2 we already presented a brief performance comparison between the two compilers, a detailed analysis would require more experimentation and the evaluation in different scenarios, which lays outside of the scope of this Thesis. In general, however, it is shown that GCC-compiled executables have lower execution times, at least for the SPARC v8 architecture.

When working to include the SPARROW assembly instructions one of the key advantages of LLVM over GCC was the possibility of defining the instructions in a nested way. This, as is explained in the Section 4.3, simplifies the addition of two-stage instructions allowing a simpler combination of them. However, a new line for each combination must be manually added. On the other hand, the code for adding these instructions is easier to understand in GCC, which can be easily deduced from the existing instructions. Fortunately, LLVM has a great documentation and a large number of tutorials can be found on how to modify it [38].

All in all, both compilers had advantages and disadvantages compared to one another, however they both offer good facilities to implement the required functionality in order to add software support in new hardware designs.

7 Conclusions and future work

7.1 Conclusions

High-performance AI and ML related computing in space is unattainable with the current technologies, and this constrains the performance of the required applications to improve the quality of space missions. In this Master thesis we have introduced SPARROW, a SIMD low-cost accelerator module for artificial intelligence applications. The pivotal point of SPARROW is its hardware/software co-design by analyzing the literature and AI software characteristics, in order to achieve an efficient implementation and obtain a performance improvement with low cost.

We have presented the architecture of SPARROW and the justification behind each design decision, which have later been supported by an extensive set of results. For example, the benefits of reusing the register file have been demonstrated both in terms of decreasing the hardware overhead and the simplification of the programming, since additional load and store instructions were not required.

The presented results show that SPARROW's implementation has up to a 30% relative cost with respect of a small base processor, which is a fraction of conventional SIMD and vector processors. For a larger one, like the NOEL-V this relative increase is down to a 5%. Furthermore, the dual implementation of SPARROW for the LEON3 and the NOEL-V act as a demonstration of the portability of the module with minimal architectural modifications, even for two different ISAs.

In regards to the performance provided by SPARROW, the obtained results confirm the significant speed-up provided by using SPARROW in machine learning workloads with over 5× speed-up in a complex inference application. However, we have also presented the results for other algorithmic building blocks related to the artificial intelligence where a speed-up of over 15× is achieved for specific matrix sizes.

Such results are presented for both GCC and LLVM, the two base compilers we have adapted to include support for the SPARROW module. The programming model is similar to the SIMD intrinsics and is achieved by using inline assembly in C, nonetheless, with a custom library the programming of SPARROW applications is simplified making the module architecture transparent to the programmer.

7.2 Future work

The presented project has not only has met with original objectives, but also exceeded them, since it showcased the project with multiple space processors and compiler toolchains. At the same time, the implementation achieved good performance results. However, there is still work to be done in order to improve the results or provide new approaches, such as the development of full compiler support for the code generation, in order not to need to resort to the inline assembly or our SIMD-like library for programming SPARROW. The support for the compiler could even include auto-vectorization, feature that could later be compared with the manual programming. Support for SPARROW could even be extended to established interfaces and libraries such as TensorFlow.

We would also like to test the module with space-relevant ML benchmarks which are currently in development at BSC in collaboration with ESA in the context of the GPU4S project and the OBPMark open source benchmarking suite, which is built on top of the GPU4S Bench benchcmarking suite. Furthermore, it would interesting to tests our design in orbit using the European Space Agency OPS-SAT FPGA platform, and use actual space ML case studies provided by ESA.

8 Related publications

An early version of the work performed in this thesis was published and presented in the following workshop organised by the European Space Agency.

- **Marc Solé**, Jannis Wolf, Leonidas Kosmidis. *Reliable Machine Learning Acceleration for Future Space Processors and FPGAs: LEON, NOEL-V and TASTE*, ESA/CNES/DLR European Workshop on On-Board Data Processing (OBDP) 2021

Moreover, the various parts of the project have been presented in the following events:

- **Marc Solé**, Leonidas Kosmidis. *Low-Cost SIMD Module for ML Acceleration*, RISC-V Forum: Vector and Machine Learning, September 15, 2021
- **Marc Solé**, Leonidas Kosmidis. *Hardware-Software Co-design for Low-cost AI processing in Space Processors*, Open Hardware 2021 Awards, Xilinx Dublin, Ireland, September 22, 2021
- **Marc Solé**, Leonidas Kosmidis. *Low-cost Hardware/Software co-designed SIMD Unit for AI Acceleration in a Qualified Space Processor*, ACM Student Research Competition (SRC) at International Conference On Computer-Aided Design (ICCAD) 2021
- **Marc Solé**, Leonidas Kosmidis. *Navigating Exotic SIMD Lands with an LLVM Guide*, Student Technical Talk, LLVM Developers' Meeting 2021.

The LEON3 part of the work of this Thesis was submitted in the *Xilinx Open Hardware 2021* competition in the Student Category, where it was awarded with the **1st place** [46]. Both the hardware design and the compiler work have been released as open-source at [47].

References

[1] M. Johns and T. J. Kazmierski. A Minimal RISC-V Vector Processor for Embedded Systems. *Forum for specification and Design Languages (FDL)*, 2020. doi: 10.1109/FDL50818.2020.9232940.

[2] B. Revuelta Fernández. Study of Scalable Architectures on FPGA for Space Data Processors. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2018.

[3] J. Meß, F. Dannemann, and F. Greif. Techniques of Artificial Intelligence for Space Applications - A Survey. *European Workshop on On-Board Data Processing (OBDP)*, 2019.

[4] N. Potter. NASA's Mars Perseverance Rover Should Leave Past Space Probes in the Dust; New mission uses AI to navigate Martian surface three times as quickly, 2021. URL `https://spectrum.ieee.org/nasa-mars-perseverance-rover-should-leave-past-space-probes-in-dust`. [Visited 21/10/2021].

[5] Intel. Intel Powers First Satellite with AI on Board. URL `https://www.intel.com/content/www/us/en/newsroom/news/first-satellite-ai.html`. [Visited 21/10/2021].

[6] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 65–74, 2017. doi: 10.1145/3020078.3021744.

[7] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers (TC)*, 100(8):746–757, 1968.

[8] K. Asanović. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, 1997.

[9] R. Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politècnica de Catalunya, 1997.

[10] M. Mittal, A. Peleg, and U. Weiser. MMX Technology Architecture Overview. *Intel Technology Journal*, 1(3), 1997.

[11] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *COMPCON'95. Technologies for the Information Superhighway*, pages 462–469. IEEE, 1995.

[12] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[13] C. Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, 2002.

[14] C.E. Kozyrakis and D.A. Patterson. Scalable, Vector Processors for Embedded Systems. *IEEE Micro*, 23(6):36–45, 2003. doi: 10.1109/MM.2003.1261385.

[15] ARM Holdings, LTD. ARM Architecture Reference Manual, ARM-v7A and ARMv7-R edition. Technical Report ARM DDI 0406C.d (ID040418), ARM, 2018.

[16] ARM Holdings, LTD. ARM Architecture Reference Manual, ARM-v8, for ARMv8-A architecture profile. Technical Report ARM DDI 0487G.a (ID0411921), ARM, 2021.

[17] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, page 61–70, 2008. doi: 10.1145/1450095.1450107.

[18] P. Yiannacouras, J. G. Steffan, and J. Rose. Fine-Grain Performance Scaling of Soft Vector Processors. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, page 97–106, 2009. doi: 10.1145/1629395.1629411.

[19] P. Yiannacouras, J. G. Steffan, and J. Rose. Portable, Flexible, and Scalable Soft Vector Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(8): 1429–1442, 2012. doi: 10.1109/TVLSI.2011.2160463.

[20] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. Lemieux. VEGAS: Soft Vector Processor with Scratchpad Memory. *International Symposium on Field Programmable Gate Arrays (FPGA)*, page 15–24, 2011. doi: 10.1145/1950413.1950420.

[21] A. Severance and G. Lemieux. VENICE: A Compact Vector Processor for FPGA Applications. *International Conference on Field-Programmable Technology (FPT)*, pages 261–268, 2012. doi: 10.1109/FPT.2012.6412146.

[22] J. Yu, C. Eagleston, C. H. Chou, M. Perreault, and G. Lemieux. Vector Processing as a Soft Processor Accelerator. *ACM Transactions on Reconfigurable Technology and Systems*, 2(2), 2009. ISSN 1936-7406. doi: 10.1145/1534916.1534922.

[23] J. Gaisler. Concurrent error-detection and modular fault-tolerance in a 32-bit processing core for embedded space flight applications. *International Symposium on Fault-Tolerant Computing (FTCS)*, pages 128–130, 1994.

[24] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. *International Conference on Dependable Systems and Networks (DSN)*, pages 409–415, 2002.

[25] D. Rea, D. Bayles, P. Kapcio, S. Doyle, and D. Stanley. PowerPC™ RAD750™-A Microprocessor for Now and the Future. *IEEE Aerospace Conference (AeroConf)*, pages 1–5, 2005.

[26] E. Schueler, M. Syed, and T. Helfers. XPP A High Performance Parallel Signal Processing Platform for Space Applications. *Data Systems In Aerospace (DASIA)*, 532, 2003.

[27] J. N. Maki, D. Gruel, C. McKinney, M. Morales, D. Lee, R. Willson, D. Copley-Woods, M. Valvo, T. Goodsall, J. McGuire, R. G. Sellar, A. E. Johnson, H. Ansari, K. Singh, T. Litwin, R. Deen, A. Culver, N. Ruoff, D. Petrizzo, D. Kessler, C. Basset, T. Estlin, F. Alibay, A. Nelessen, S. Algermissen, M. A. Ravine, J. A. Schaffner, M. A. Caplinger and J. M. Shamah. The Mars 2020 Engineering Cameras and microphone on the perseverance rover: A next-generation imaging system for Mars exploration. *Space Science Reviews*, 216(8):1–48, 2020.

[28] Cobham Gaisler, AB. Gaisler Web, 2021. URL `https://www.gaisler.com`. [Visited 29/09/2021].

[29] Discourse. GRLIB Community, 2021. URL `https://discourse.grlib.community`. [Visited 29/09/2021].

[30] Cobham Gaisler, AB. GRLIB User's Manual, 2021. URL `https://www.gaisler.com/products/grlib/grlib.pdf`. [Visited 29/09/2021].

[31] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc. GRLIB IP core user's manual, Version 2021.1, 2021. URL `https://www.gaisler.com/products/grlib/grip.pdf`. [Visited 29/09/2021].

[32] J. Rupley. Samsung M3 Processor. HotChips 2018, 2018.

[33] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. PuDianNao: A Polyvalent Machine Learning Accelerator. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. doi: 10.1145/2694344.2694358.

[34] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *International Symposium on Computer Architecture (ISCA)*, 2017. doi: 10.1145/3079856.3080246.

[35] J. Gaisler. A structured VHDL design method. *Fault-tolerant microprocessors for space applications*, pages 41–50, 2011.

[36] M. M. Trompouki and L. Kosmidis. Towards general purpose computations on low-end mobile GPUs. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2016. doi: 10.3850/9783981537079_0165.

[37] LLVM. llvm-project, 2021. URL `https://github.com/llvm/llvm-project`. [Visited 18/10/2021].

[38] LLVM. About – LLVM 13 documentation, 2021. URL `https://llvm.org/docs/index.html`. [Visited 18/10/2021].

[39] Cobham Gaisler, AB. GRLIB IP Core Performance and Resource Utilization. URL `https://www.gaisler.com/products/grlib/grlib_area.xls`. [Visited 09/10/2021].

[40] M. W. Learn. Evaluation of the Leon3 Soft-Core Processor Within a Xilinx Radiation-Hardened Field-Programmable Gate Array. Technical Report SAND2012-0454, Sandia National Labs, April 2011.

[41] S. J. Jie and N. Kapre. Comparing Soft and Hard Vector Processing in FPGA-based Embedded Systems. *Conference on Fiel Programmable Logic and Applications (FPL)*, 2014.

[42] I. Rodriguez, L. Kosmidis, J. Lachaize, O. Notebaert, and D. Steenari. GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing. Technical report, Universitat Politecnica de Catalunya, 2019. URL `https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html`. [Visited 11/10/21].

[43] L. Kosmidis, I. Rodriguez, A. Jover, G. Cabo, S. Alcaide, J. Lachaize, O. Notebaert, A. Certain, and D. Steenari. GPU4S (GPUs for Space): Are we there yet? *European Workshop on On-Board Data Processing (OBDP)*, 2021.

[44] D. Steenari, L. Kosmidis, I. Rodrigez, A. Jover, and K. Forster. OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications. *ESA/DLR/CNES European Workshop on On-Board Data Processing (OBDP)*, 2021.

[45] Cobham Gaisler, AB. Grmon3 user's manual, 2021. URL `https://www.gaisler.com/doc/grmon3.pdf`. [Visited 11/10/21].

[46] Xilinx University Program. OPEN HARDWARE - 2021 Results, 2021. URL `http://www.openhw.eu/2021`. [Visited 20/10/2021].

[47] M. Solé Bonet. XOHW_GRLIB_AI_extension, 2021. URL `https://gitlab.bsc.es/msolebon/XOHW_GRLIB_AI_extension`. [Visited 20/10/2021].