Confidential

Adding of a vector co-processor on an open-source RISC-V core for cryptographic algorithms execution

March 2023 - August 2023

Corentin MARAIS

2023 Master's degree in Micro- and nanotechnologies for integrated systems (MNIS)

INSTITUT NATIONAL POLYTECHNIQUE - PHELMA

http://phelma.grenoble-inp.fr

Under the supervision of :

Jean-Roch Coulon (jean-roch.coulon@thalesgroup.com)

Katell Morin-Allory (katell.morin-allory@univ-grenoble-alpes.fr)

Thales Silicon Security Arteparc Meyreuil, Bâtiment D Route de la Côte d'Azur 13590 Meyreuil France

Declaration

I hereby declare that the work contained in this report has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, this work contains no material previously published or written by another person except where due reference is cited.

Corentin MARAIS
ID: 41901519
Master Nanotechnologies for Integrated Systems
Institut National Polytechnique - Phelma

Approval by the Supervisor

This is to certify that **Corentin Marais** has carried out this internship under my supervision, and that he has fulfilled the relevant Academic Ordinance of the Grenoble-INP Phelma, so that he is qualified to submit the following Master Thesis in the application for the degree of Master in Nanotechnologies. Furthermore, the Master Thesis complies with the plagiarism and academic integrity regulation of Grenoble-INP.

Jean-Roch Coulon
Digital Manager
Thales Digital Identity and Security (Invia)

Acknowledgement

First and foremost, I would like to thanks Jean-Roch Coulon, Yannick Casamatta, Andre Sintzoff, Come Allart, Zbigniew Chamski, Guillaume Chauvon, Jules Fauchon, Aziz Frikha, Sebastien Lapeyre for their professionalism and their support during my internship. I also want to thank Thales SAS Invia that gave me the opportunity to progress as an intern for 6 months. The work with the RISC-V team has been challenging while keeping a motivating work environment.

I also thank Ms Katell Morin-Allory for her guidance and all 3 universities, Grenoble-INP Phelma, Politecnico di Torino and EPFL whose engineering training allowed me to carry out this project. Finally, I would like to express our gratitude to Thales DIS for providing me with the necessary facilities to undertake this work and to the Open Hardware Group for their support.

Contents

List of Figures

Glossary

AES Advanced Encryption Standard, is a widely used symmetric encryption algorithm. 5

ASIC Application-Specific Integrated Circuit, custom-designed integrated circuit that is purpose-built for a specific application or function. 6, 8, 10, 17, 31, 49

CI Continuous integration, the Open Hardware Group and Thales use Gitlab and Github as a platform for collaborative work. The platform it set up to ensures that features and progress are non-regressive.. 25

CVA6 a six-stage processor project that can implement the RISC-V ISA. 2

DIS Thales Defense Identity and Security business unit. 1

FPGA Field Programmable Gate Array, versatile component that can be used for various application. 8, 11, 49

FPU Floating point unit, unit which manages floating point instructions. 8

GBU Global Business Unit. 2

ILEN Instruction length.. 8

IP Intellectual properties, it can stands for a code, an hardware product or everything that is protected by Thales. 2

ISA Instruction Set Architecture, represents the specifications and the standardization of instructions to design a processor. 2, 3, 8, 11, 17, 20, 27–29, 39

mnemonic In coding, a mnemonic is a symbolic code or abbreviation used to represent a complex or specific instruction, function, or operation in a more human-readable and memorable form.. 24

OHG Open Hardware Group. 2

RISC-V RISC-V is an open-source Instruction Set Architecture (ISA) for computer processors.. iv, 2, 7, 9, 18, 20, 27–31, 40

RSA Rivest-Shamir-Adleman, is a widely used asymmetric encryption algorithm. 5

RTL Register Transfer Level design is a method of describing digital hardware behavior by specifying how data is transferred between registers using hardware description languages.. 4, 8, 25, 31

SIMD Specific type of computation. The computation is done by splitting a 64 bit elements into smaller subelements of 8, 16 or 32 bits when it is possible.. 25

smart cards Smart cards are electronic devices with embedded microprocessors that store and process data. 1

vcs Verilog Compiled Simulator, a popular tool in the field of electronic design automation (EDA) used for hardware design and verification. Developed by Synopsys, a leading EDA company.. 8

verilator An open-source high-speed Verilog/SystemVerilog simulator. 8

VRF Vector register file. 13

Chapter 1

Introduction

1.1 Thales

Thales is a French multinational electronics company listed on the Paris stock exchange. It has 81,000 employees in 68 countries and the company supplies electronic equipment to five main markets: digital identity and security, land transport, defense and security, aeronautics and space. With the acquisition of Gemalto and Invia, a world leader in data identification and protection, Thales acquires a range of technologies and skills complementary to its own. Among this huge digital identity and security company, ex-Invia still a research and development leader in Thales facilities and for the products such as Smart cards that are sold in bank field or governmental one.

Thales clients are divided in two parts:

- Private companies such as Microsoft, HSBC, Airbus or Google.
- Public clients such as countries, governments, armed forces, cities like Beijing, Paris, New York, Singapore or organizations the European space agency.

In 2020, Thales had a revenue of 17 billion euros and that year, one billion euros was dedicated to self-funded research and development. The internal structure of the company is organized by region and sector of activity. Thales is composed of 5 Business Units which are divided in more than 16 business lines. Thales DIS Design Services SAS formerly INVIA is part of the digital identity and security unit (DIS). This sector produces equipment and services to secure the identity, communication, or data transactions of people and connected objects.

1.2 Invia

The R&D section that hosted my internship is used to be called Invia in reference to the previous name of the team. Within Thales DIS, this section is a microelectronics company that designs secure

Intellectual Properties (IP). The company was founded in 2006 by Robert Leydier and Alain Pomet, two experienced semiconductor engineers. Key members of the company, with more than fifteen years old experience in smart card design they are experts in their fields.

Invia's research and development team is now made up of 60 people, including 17 doctors. The company became a member of Thales group in 2019. Invia's customers are system integrators where security is a necessity. They are present in several field including the military, banking, telecommunications, biomedical and many other ones. The company aims at providing a wide range of IPs to secure integrated circuits but also to supply products to other Thales Global Business Units (GBU). Invia has 70 employees in nine teams. Each team is responsible for part of the NP design, verification or production process. A detailed description of the teams is available in annex.

As a main contributor to the processor open-source development and particularly the RISC-V project, Thales works with international teams such as the Open Hardware Group. Thales is a strong support to this strategic project.

1.3 Open Hardware Group

Open Hardware Group [5] (OHG) is an international non-profit organization that, together with its partners, designs open-source processors, tools and software. The success of open-source software (Linux) has proven the viability of this model. This has been a major inspiration to the project of open-source in hardware which represents a major concern for companies that are looking for independence. Some big companies figure among Open Hardware Group members and partners like Samsung, Bosch, NXP or Thales.

The projects within Open Hardware Group are spread over different fields around the world. There are among 5 processor projects that are conducting in parallel including a project called CVA6 which will be introduce latter.

1.4 Context and RISC-V

Considering that developing a competitive processor is costly in term of human resources, technology, time and thus budget, the open instruction set architecture (ISA) RISC-V represents many interests for microelectronic companies.

Since its creation in 2010, it has become a reference in microelectronic industry to be now a competitor to traditional ISA providers such as ARM. Many international companies have joined the RISC-V project, such as Thales but also NVIDIA, ST Microelectronics, Samsung, IBM and many others. Since 2015, RISC-V is a Swiss-based foundation with more than 2,000 members in over 70 countries.

One of the current problems in microelectronics is that most processors have a proprietary set of instructions. Thus, to use these processors, it is necessary to have the often burdensome and costly permission of the owner. This functioning limits the innovation and possible applications of these processors. RISC-V is then a good alternative to this complex and costly development. The RISC-V ISA has only 47 instructions which is the smallest 32-bit and 64-bit ISA on the market. In addition to this, the RISC-V is modular. Since the instructions are grouped into extensions this provides flexibility and an exceptional innovation potential for performances, size, implementation or architecture of new microprocessors.

Now that many entities are investing in RISC-V and the core IPs are in the public domain, the technology becomes more mature every day and the risks of working with this core decreases. Various studies also show that this set of instructions should take more and more market shares in the coming years. According to Semico Research [1], 80,000 processors which 14% of the market, will be based on RISC-V by 2025.

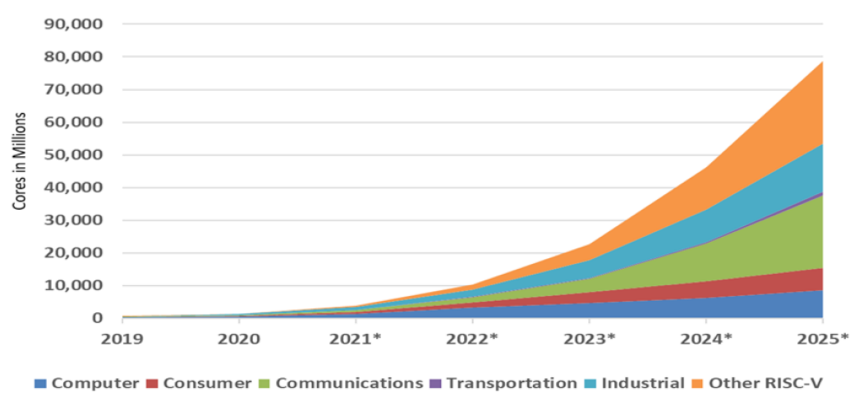

Figure 1.1: Market size progression of RISCV-V cores until 2025 [1].

The support of RISC-V project by an international community is a major asset to simplify the monitoring of cores. Thales proprietary technologies are aging and lack support and the low computing power of the current processors will soon be limiting for future development. As an example, RISC-V processors are already a pertinent solution for the space industry and will replace the old cores. In aeronautics also, single-core processors will no longer be enough. Moreover, this new ISA could be simply a base for development of new cores other than RISC-V.

1.5 Internship project

Are vector co-processors suited for the next generation of Thales Invia's products which involve execution of PQC algorithms?

Arrived at Thales Invia in early March 2023, my work aims at measuring the performances of a vector extension connected to a RISC-V CVA6 core. This evaluation is done by executing a cryptographic algorithm called Keccak. The vector extension Ara is a project conducted by the Open Hardware Group and is currently accessible in open-source. This vector extension was expected to be an accelerator for Artificial Intelligence and cryptography algorithm. Ara/CVA6 structure mainly works as a dual processor and are both connected to an external memory. The relevance of such a co-processor will be based on its performances and the resources it needs.

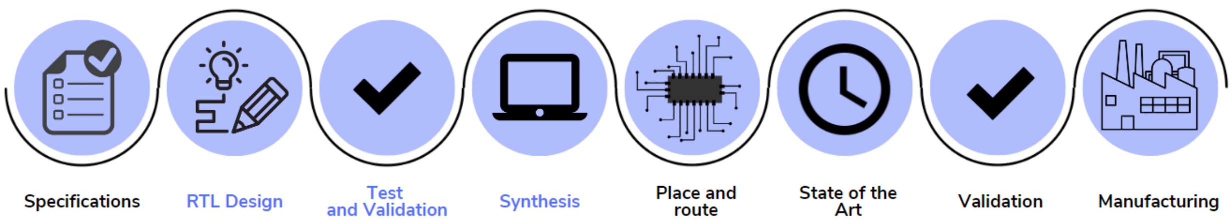

Figure 1.2: Electronic design flow.

My work is focused on the earliest stages of development. I will focus on the three following steps:

- **RTL Design**: By exploring the existing Ara's design, I will have to implement new hardware to optimise the execution of a cryptographic algorithm and compute the performances achievable.
- **Test and validation**: The implementation must be functional thus a validation including small tests is necessary.
- **Synthesis**: This step is critical when designing since it allows the designer to have a first overview of the resources needed by its design.

The cryptographic context and Ara's design will be introduced first. I will then detail the hardware designed to estimate the performances of a mature Ara which is still under development. Eventually I will discuss a perspective of improvement to use such a vector coprocessor in Thales products.

Chapter 2

Dilithium and CVA6 performances

Digitalization of services and companies that started during the second half of the XX century has made the cryptography one of the main concerns of modern security. Devices and data that are now connected by computers need to be protected from cyber criminality and more recently from post-quantum cyberattacks.

Considering this post-quantum challenge, the National Institute of Standards and Technology (NIST), introduced the PQC project in 2016 with the goal of soliciting, evaluating, and standardizing quantum-resistant cryptographic algorithms. The project involves multiple rounds of evaluation and selection, where algorithms are submitted by researchers and industry experts for scrutiny and analysis. After having organized a world competition for secure PQC algorithms, NIST chose 2 finalist Dilithium and Kyber.

Thus, a few years ago, Thales Invia starts working with the help of post-doc researcher on the optimization and the implementation of Dilithium in its products which actually use common cryptography algorithms like AES or RSA.

2.1 Keccak

In Dilithium and more broadly in cryptography algorithms, Keccak is a cryptographic hash function that is used for generating digital fingerprints or fixed-size representations of large data [6]. It was developed by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche, and was selected as the winner of the SHA-3 competition held by the NIST in 2012. Keccak used in several steps of Dilithium is a sponge-based hash function that operates on an input data block of any size and produces a fixed-length output of 224, 256, 384, or 512 bits. The function uses a permutation-based design, which involves applying a sequence of mathematical operations to the input data block and a set of fixed constants. Its flexibility and versatility have made it a popular choice for many different cryptographic applications [7].

Keccak is a computational step in Dilithium and in recent products of Invia, this algorithm is accelerated in an ASIC coprossecor connected to the in-house core. To introduce the subject, a first work was to measure the efficiency of the CVA6 executing Keccak and compare the performances obtained to the ones from the in-house core. The Keccak algorithm is decomposed in 4 steps as depicted in figure 2.1:

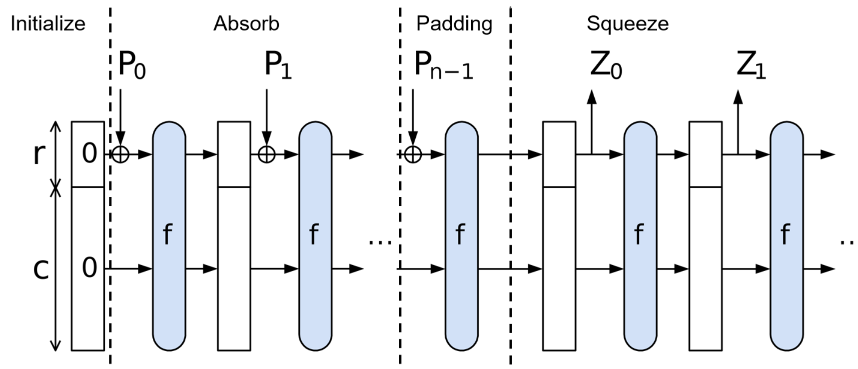

Figure 2.1: Sponge construction of the Keccak algorithm [2]

1 Initialization:

- The input data is here represented by $Input = [P_0 : P_n]$.
- The capacity c and the rate r are initialized.
- The state array (vertical rectangle composed of r and c blocks) is initialized before absorbing the input data.
- It sets the initial values of the state array based on the hash function's parameters (the initial value is 0 in figure 2.1).

2 Absorb:

- The input data is split into blocks of r size, and the blocks $P_i \in [P_i : P_{n-1}]$ are XORed with the first r elements in the state array.
- This step is responsible for diffusing the input data into the state array.
- If the input data is longer than the block size r, the process is repeated for multiple blocks.
- The function f, commonly called **state permutation** function, propagates the absorbed input into the whole state array. In the rest of the master thesis, I will call this function KeccakF1600_StatePermute.

3 Padding:

- The Padding step pads the last block of the input data P_n and completes the absorption process.
- It ensures that the last block is properly formatted and aligned with the block size of the hash function.

4 Squeeze:

- The Squeeze step extracts the final hash value from the state array after the absorption and padding phases.
- Bits are extracted from the state array by group of r to form the desired hash output $(Z_0, Z_1 ...)$.

For the performance test, we decided to implement a loop over the last step (squeeze) of Keccak which is the most used in Dilithium. The test aimed at comparing 3 different cores: a cv32a6 (CVA6 configured for 32-bit processing), a cv64a6 (CVA6 configured for 64-bit processing) and the in-house core which is configured on 32 bits data processing. CVA6 cores are introduced in the next section.

After several trials with Keccaks coded either in 32 or 64 bits, we chose to continue the comparison with a reference Keccak previously coded by a member of the cryptography team. Indeed, I designed a Keccak in 32 bits but it was sub-optimized and the execution on the in-house core was not representative of its performances. For simplicity, in the reference test defined, we also choose to use small data in input: a table of 64 elements of 32 bits equal to zero. Thus, the squeeze step became critical and running it 100 times gave a good average of the time needed for a Keccak execution (on small data).

Together with members of the cryptography team we defined the next specifications for a fair comparison:

- The test should be written in c (no assembly code) and compile with gcc compiler.
- The compiler options should be the same (and compatible with the in-house core).

Compilation refers to the process of translating human-readable source code (here C or C++) into machine-readable instructions, a code that the computer's core can understand and execute (binary file).

2.2 CVA6

As mentioned above, CVA6 is one of the last RISC-V core in development by the Open Hardware Group and Thales Invia. The core is designed in system Verilog and can be implemented in different targets such as FPGA or ASIC. The design is scalable and configurable. The user can choose parameters like the instruction length ILEN to be either 32- or 64-bits. Also, unit of the CVA6 like the Floating Point Unit (FPU) can be enabled or disabled depending on the application domain.

CVA6 is mainly the name of the repository GitHub that contains the source code for few implementations of the core and CV stands for a Core-V family of RISC-V core while A6 indicates that the application class processor contains a 6-stages execution pipeline. Nevertheless, the CVA6 does not implement a specific production core, but it is expected to be a base for numerous different applications. Indeed, the scalability and the availability of this open-source processor created enthusiasm around industries and companies in several fields.

2.3 Benchmark

The following benchmark aims at comparing three cores: cv32a6, cv64a6 and the in-house one. The test flow including compilation is represented in the next figure:

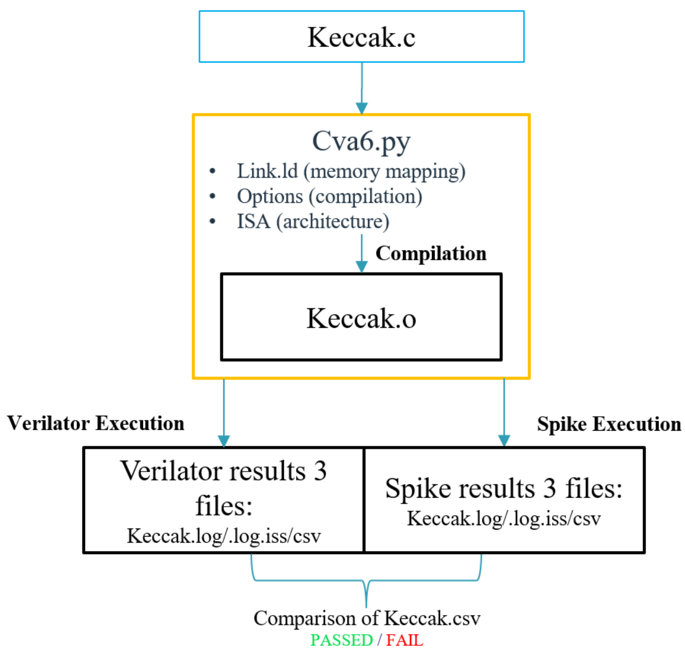

Figure 2.2: Test flow with the **cva6.py** script.

Testing the different CVA6 architecture involves an open-source script called **cva6.py**, developed by Thales Invia's team. Highly parametrizable, the script manages the compilation of different type of code (c, assembly, elf...) and the configuration of several hardware simulator like Vcs or Verilator. The simulators are used to perform a Register Transfer Level (RTL) execution of the compiled code using CVA6 hardware, and compare it with a simulator called Spike. Spike is developed as part of the RISC-V project and is considered as the golden RISC-V ISA model. The comparison between the two executions (spike and a RTL simulator) is done using instruction traces.

An instruction trace, in processor field, is the historic of the tasks done by a core using the codified instruction set architecture. The instructions generated while compiling the Keccak c-code depends on one argument called **–march** that is passed to the compiler [8]. This argument refers to the target architecture or micro-architecture for which the compiled code will be optimized. As an example, **cv64a6_imafc_sv39** corresponds to a compilation which uses (**–march=rv64imafdc**) for a RISC-V 64 bits target that enables the extensions i for integer instruction, m for multiplication on integer, a for atomic instructions, f for single-precision floating-point instructions (decimal numbers) and c for compressed instructions.

Eventually, the status PASS/FAIL of a test depends on the comparison of two instruction traces: one from the simulator and one from Spike.

2.4 Results

The core performances are summarized in the next figure:

Keccak reference (100 squeeze)	GCC version	Optimisations	Bitmanip	Instructions	cycles
target : cv64a6_imafc_sv39	**13,1,0**	\-O3	/	236000	267000
	13,1,0	\-O3	zbb	136000	150000
target : cv32a6_imafc_sv32	**13,1,0**	\-O3	/	630000	769000
	13,1,0	\-O3	zbb	524000	604000
target : recent in-house core	**13,1,0**	\-O3	/	400000	**575000**

Figure 2.3: Core performances executing Keccak Algorithm with 100 squeeze steps.

The notion of cycle is used to compare the performances between two executions since it does not depends on the frequency of the clock used. It basically corresponds to the total execution time divided by the clock period.

The table highlights the internal construction of the algorithm: elements of the state array are coded on 64 bits. In fact, the level of security of a Keccak algorithm is closely related to a figure of merit called entropy which involve the size of the state array. In the Keccak of reference used for the benchmark, this size is 1600 bits. If the states are coded on 64 bits, the state permutation function will do 24 rounds to propagate the input block P_i of size r (figure 2.1) into the 1600 bits state array while if the states are coded on 32 bits, the rounds must be doubled to obtain the same propagation. Doubling the rounds means doubling the number of instructions executed and thus increases the execution time. In fact, this test points out the difference between a 64-bit and a 32-bit based architecture while running Keccak at this level of security.

The biggest gain of performances is achieved when using **zbb** extension. At compilation step, this extension enables the generation of a new instruction called *rol* [8]. *rol* corresponds to a ”rotation left”. In arithmetic operation this instruction groups 3 standard instructions together. Details about *rol* instruction will be given in section 4.4.1.

Nevertheless, more than being a gain in execution time, this extension is also a way to divide the code size by almost 2 when using the cv64a6. Indeed, in the context of embedded product, code size is crucial for efficient memory utilization, cost-effectiveness, or power consumption [9]. It is a way to create more compact, cost-efficient, and energy-efficient devices while enhancing performances. The use of RISC-V cores has been boosted by the creation of extensions like **zbb** are the dynamic around these open-source cores is a major upside in favor of using the cva6 in future products.

Clearly then, **the cv32a6 has almost the same execution speed than the in-house processor**. This test highlights the intrinsic capability of each core to execute a c-compiled algorithm.

In the current products of Thales Invia, Keccak algorithm is accelerated by an ASIC accelerator. This accelerator is able to run the Keccak algorithm in few thousands’ cycles. Thus, this first step of my internship obviously highlights the need to use a 64-bit core to accelerate the execution. By using a 64-bits coprocessor without specific optimisations, the execution time is 3 times better than using a 32 bits CVA6. One possibility to replace the ASIC accelerator would be to use a vector coprocessor operating on 64 bits.

Chapter 3

Ara a vector coprocessor

The vector coprocessor studied in this master thesis is named Ara and have been connected few months ago to a cva6 by the Open Hardware group team using an interface which leaves the full decoding tasks to Ara. In this electronic module, Ara and CVA6 are distinct.

It has been important for my work to understand communication between the two cores and the proper working flow of Ara to perform an in-depth diagnostic over the viability of this solution to replace the in-house accelerator on future products using CVA6.

3.1 Vector processors

Vector processors are specialized processors that implement vector-SIMD (Single Instruction, Multiple Data) instructions. They are time-multiplexed versions of array processors, where several specialized functional units stream micro-operations on consecutive cycles. This configuration allows for dynamic configuration of the vector length, independent of the number of functional units. Unlike packed-SIMD, vector processors do not require subdividing long vectors into fixed-size chunks. Instead, they can be issued using a single vector instruction. As a result, vector processors have the potential to be more energy efficient than equivalent array processors.

Vector processors firstly found their way into Field-Programmable Gate Arrays (FPGAs) as general-purpose accelerators. As an example, ARM already entered the vector processing domain with their an architecture called Scalable Vector Extension (SVE), which allows for different implementations with varying vector lengths (from 128 bits to 2048 bits).

The open RISC-V ISA specification is also making strides in vector processing through its vector extension. This extension, currently at version 1.0, is actively developed and places almost no limits on vector length. It also provides the flexibility to trade off the number of architectural vector registers for longer vectors.

Vector lanes correspond to the execution units and maximizing their utilization is a crucial aspect of vector processing design. Different approaches have been explored, such as sharing a pool of vector units among different threads. Studies have shown that intelligent sharing of vector units on multi-core processors can increase efficiency and throughput compared to multi-core processors with per-core private vector units [3].

3.2 Ara

As introduced above, in the context of my internship the CVA6 is not used alone. The Ara vector coprocessor is connected to the CVA6 core and supports the extended set of instruction: v-instructions. The interface between the CVA6 and Ara is lightweight and leaves the full decoding task to the coprocessor. It thus passes the instruction to the coprocessor if it is not recognized by the CVA6 decoder. Ara raises an exception if the instruction is not included in the v-extension or if a memory issue occurs. The design of Ara is introduced in the next section.

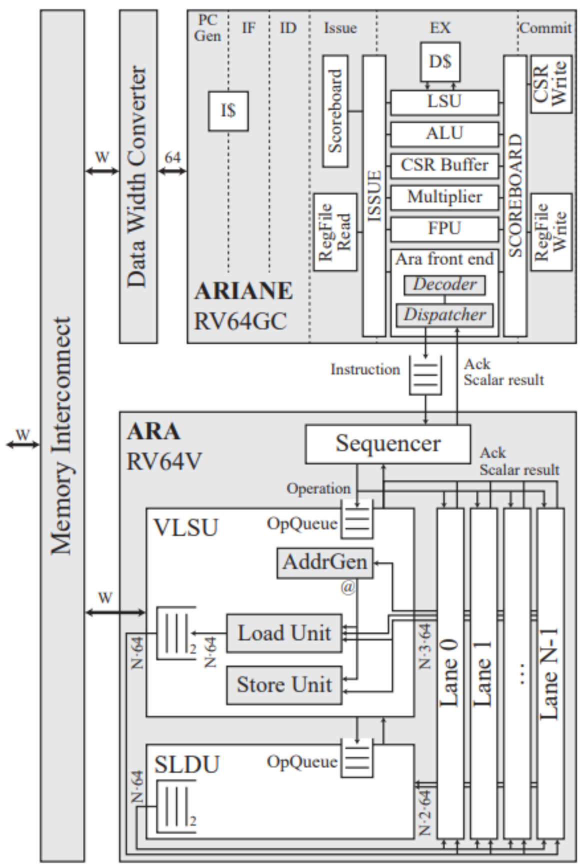

Figure 3.1: CVA6 and Ara design [3].

The design of the vector coprocessor Ara is divided into 4 main blocks.

3.2.1 Sequencer

The sequencer plays a vital role in the operation of Ara, as it manages the execution of vector instructions, coordinating with the different execution units and communicating with the CVA6. It is the unit that maintains a global view of the progress of instruction execution across all execution units: the vector lanes. The sequencer has the capacity to handle up to eight parallel instructions (if the number of lanes is compatible with it), ensuring a continuous flow of instructions for execution.

One of the key responsibilities of the sequencer is to resolve hazards that may arise among pending vector instructions. Structural hazards can occur due to architectural decisions, such as shared paths between the Arithmetic Logic Unit (ALU) and the Single Lane Dispatch Unit (SLDU), or if a functional unit's operation queue reaches its capacity and cannot accept additional instructions. In such cases, the sequencer delays the issuance of vector instructions until the structural hazard is resolved, meaning the conflicting instruction has completed its execution.

3.2.2 SLDU

The Slide Unit (SLDU) is responsible for handling instructions that require simultaneous access to all Vector Register File (VRF) banks. Its main tasks include managing operations like element insertion and extraction in vectors, vector shuffling, and vector sliding (where elements are shifted within a vector, such as:

$$vd[i] = vs[i + slideamount]$$

Additionally, the SLDU can be extended to support basic vector reductions, such as vector addition and internal product. However, the current version of RISC-V's vector extension considers support for vector reductions as an optional feature.

In the design of Ara, the decision made by the Open Hardware group was to not include support for vector reductions in order to maintain simplicity. This is based on the understanding that an $O(n)$ vector reduction can still be achieved by implementing a sequence of $O(logn)$ vector slides and the corresponding arithmetic instruction. By utilizing this approach, the Open Hardware Group team ensured that the vector processor could perform necessary reductions efficiently without the need for dedicated reduction instructions, considering the available options and trade-offs.

Thus, the SLDU is one of the blocks that manages instructions which require simultaneous access to all VRF banks. It facilitates operations like element manipulation, vector shuffling, and sliding, and has the potential to support basic vector reductions if desired. The design choices made regarding vector reductions were driven by considerations of simplicity and the ability to achieve efficient reduction functionality using existing instructions and techniques.

3.2.3 VLSU

Ara incorporates a single memory port, designed to maintain an efficient balance between memory access and computational capabilities. Figure 3.1 provides an illustration of the components involved in memory access within Ara's architecture.

The address generator in Ara is responsible for determining the specific memory address to be accessed. It supports three types of memory operations:

1 Unit-stride loads and stores: These operations access a contiguous chunk of memory. The address generator determines the starting address, and to optimize data transfer efficiency, the unit coalesces unit-stride memory operations into burst requests. This eliminates the need to request individual elements from memory.

2 Constant-stride memory operations: These operations access memory addresses with a fixed offset between them. The address generator calculates the addresses based on the constant stride and initiates the corresponding memory access.

3 Scatters and gathers: These operations involve accessing memory using a vector of offsets, allowing for more general access patterns. The address generator utilizes the vector of offsets to determine the memory addresses to be accessed.

Following the address generation stage, the unit combines the unit-stride memory operations into burst requests. The burst start addresses and burst length are then sent to either the load or the store unit, both of which are responsible for initiating the actual data transfers through Ara's Advanced extensible Interface (AXI) interface. These units handle the communication between Ara and the memory system, ensuring the efficient transfer of data between the processor and memory.

3.2.4 Lane

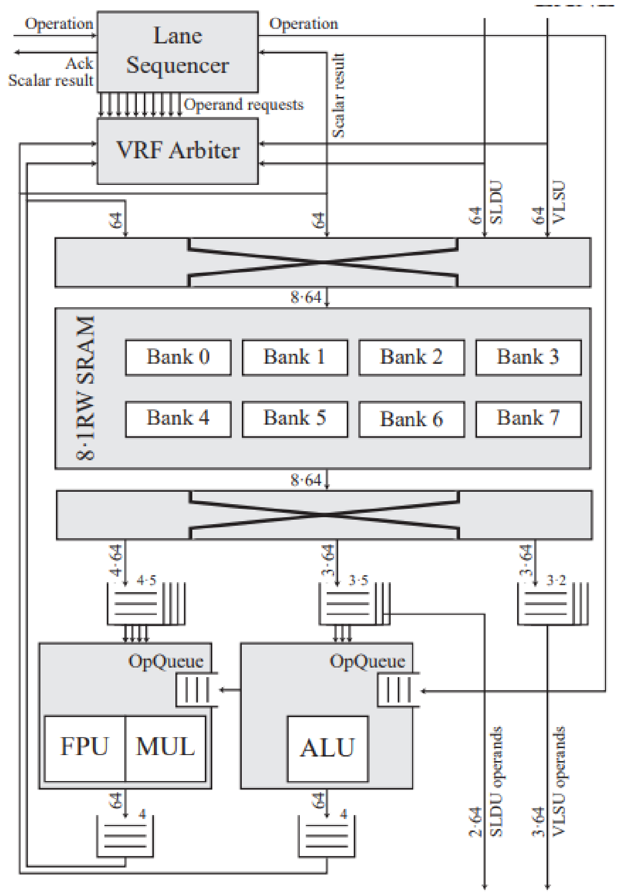

Figure 3.2: Lane design [3].

The number of lanes in the design is configurable. Each lane are identical and have its own architecture (Figure 3.2). It includes various components to support parallel computation.

1 **Lane Sequencer**: The lane sequencer handles the execution of vector instructions within a single lane. The lane sequencer issues vector instructions to the functional units and controls their execution. It initiates requests to read operands from the Vector Register File (VRF) and generates up to ten independent requests. Operand fetch and result write-back are decoupled, and back pressure is used to regulate the operation request rate, ensuring dependent instructions run at the same pace. Data hazards are handled by synchronizing the availability of operands. For instance, if a vector instruction attempts to write to a vector register that is currently being written, the lane sequencer identifies a write-after-write (WAW) data hazard between them. Similar handling applies to read-after-write (RAW) and write-after-read (WAR) hazards.

2 **Vector Register File** (VRF): The VRF is a critical component in every vector processor. In Ara, the VRF is composed of a set of single-ported (1RW) banks. Within Ara, each lane has its own VRF, including eight 64-bit wide 1RW banks. The Vector Register File provides a sufficient throughput to supply operands to the functional units and receive their results. Inter-lane communication occurs through the VLSU and SLDU components. No other unit can manage elements from different lanes. An arbiter per bank resolves banking conflicts dynamically, with lower priority assigned to low-throughput instructions to prevent interference with high-throughput instructions.

3 **Operand Queues**: Banking conflicts can occur when the same VRF bank access is required by different functional units. Each lane involves an operand queue between the VRF and functional units to handle these conflicts. There are ten operand queues in total, with dedicated queues for the FPU/MUL unit, ALU, VLSU, and shared queues. Depending on the latency and throughput of the functional unit the queue depth can vary. The queues absorb banking conflicts on the write-back path to the VRF, allowing for a pipelined execution of vector instructions. The pipeline allows storing or loading data from the VRF while processing the previous data.

4 **Execution Units**: Each lane includes three execution units: an integer ALU, an integer MUL, and an FPU (which is not useful for my study since the Keccak uses only 64 bit integer). All execution units operate on a 64-bit data path. The MUL unit shares operand queues with the FPU, but they cannot be used simultaneously. Vector chaining is allowed between any execution units, except when executing instructions with vector shuffles. The execution units support various data formats, including 1×64, 2×32, 4×16, and 8×8 bit signed or unsigned operands.

Ara's configurable vector processor architecture incorporates multiple identical lanes, and everyone has its own lane sequencer, VRF, operand queues, and execution units. This design enables parallel execution of vector instructions, with support for various data formats and seamless coordination between functional units.

Cv64a6 and Ara were connected by the Open Hardware Group and the interface between the two technologies is not a concern of my study. An introduction of Ara's design was necessary to understand the limitations faced when optimizing the execution of Keccak on Ara. Hardware will also be implemented to compute the performances achievable.

Chapter 4

Keccak execution and performances on Ara

Starting from the c code of Keccak algorithm, the goal was to compute the performance of Ara executing such an algorithm and the viability of this coprocessor to replace the current ASIC accelerator. Thales products have four main concerns to guarantee the competitiveness of their product on the market:

- The resources needed for a technology fix the price of the product on the market and its embedded capability in a pin card.
- The execution performances must be comparable to the one obtained by the main concurrent. In my case, I will consider that the accelerator currently used on the in-house core is competitive. It will thus be my reference for performances.
- The energy consumption which is critical when no power supply is integrated to the product and comes from outside. I will however not take this parameter into account in my study since performances in term of resources and execution time are more critical at early stages of development.
- The security is really important for a product that belongs to be sell in Digital and Identity market.

Despite its efficiency, the current ASIC accelerator connected to the in-house processor lakes of genericity since is not compatible with an ISA and only the Keccak algorithm can be run with it. Thus, a versatile equivalent using the v-ISA with comparable performances would be able to run other algorithms of Dilithium while being more secure.

4.1 New compiler

Considering the current compiler, the v-instruction extension in relatively new. So, the first issue faced was to find a way to compile v-instructions that is to say to translate the c code of Keccak into instructions readable by Ara. Current releases of compilers like **llvm 16** or **gcc 13.1.0** should have the ability to generate automatically v-instruction from c code [10]. To start with the vector extension, I successively had to build both compilers. In this context, I was brought to contribute to the open-source project **core-v-verif** which is the repository dedicated to the verification of CVA6 design.

In a public repository like **core-v-verif** passing from an old compiler **gcc 10.2.0** to the last version **gcc 13.1.0** was needed to access new compiling features and extensions. Issues were reported by different teams around the world when trying to use the new compiler with the public tools. Indeed, the main changes between the two versions was the definition of the **–march** (introduced in section 2.3) and thus the Instruction Set. In **gcc 13.1.0**, some instructions have been grouped in new extensions while they were included by default in a compilation with **gcc 10.2.0**.

Thus, to take these compiler changes into account, Jean-Roch Coulon trusted me to modify the continuous integration benchmark and implement new software including a parameter in **cva6.py** script. **cva6.py** is the main tool for verification as introduced in the section 2.3. My contribution was merged this summer to the Open Hardware Group public repository: **core-v-verif**.

However, few trials have shown that these new compilers were not able to compile v-instructions for Keccak algorithm even after using vector built-ins (c structures dedicated to vector compilation). After having involved non-mature version of compiler like the last commit of the open-source compiler **llvm 16**, coding in assembly was necessary to guarantee the use of v-instructions and thus Ara. Following the RISC-V v-instruction set specifications [11] and some tests already available in the Ara repository, I wrote an equivalent KeccakF1600_StatePermute function (introduced as f in section 2.2) in assembly involving several v-instructions.

4.2 Assembly v-instructions

This extension represents a way to perform large data computing in one instruction. As introduced in the precedent chapter, the architecture of a vector processor is designed to manage various memory accesses that are required by only one instruction. Thus, this extension can reduce drastically the number of instructions executed.

All v-instructions are coded on 32 bits and there are 16 formats of instruction. The one shown in the next figure is the same for all arithmetic integer v-instructions.

Bits	31--25	24--20	19--17	16--12	11--7	6	5--0
Section	category	vd/rd	index	vs1/rs1	vs2/rs2	vm	funct6

Figure 4.1: Format of v-instructions.

The format is almost the same for the other type of instruction. In this example:

- The category, coded on 7 bits, is the termination of the instruction. It specifies the type of the objects managed by this instruction.
- vd/rd, coded on 5 bits, is the destination register (either vector or not).
- index, coded on 3 bits, represents a subcategory of instruction for a specific name and a specific category.
- vs1/rs1 such as vs2/rs2, coded on 5 bits respectively, defines the starting address of the vector/register used in the computation.
- vm, coded on 1 bit, specifies the use of a mask or not.
- The name, coded on 5 bits, must explicitly describe the purpose of the instruction.

Here is a basic instruction using the previous format:

$$vxor.vv \quad vd \quad vs1 \quad vs2 \quad v0.t$$

The name of the instruction is here *vxor*. It specifies that the arithmetic instruction to do is a xor. The *.vv* is the category that contains the arithmetic instructions that involve 2 vectors of integers. The two vectors are respectively called source register 1 *vs1* and source register 2 *vs2*. The *v0.t* is the mask vector, a vector able to block the *xor* operation on elements. This instruction stores the result in the register destination *vd*.

In our case, three categories of v-instructions contain the quasi-totality of instructions needed for KeccakF1600_StatePermute computation. This is an overview to understand the meaning of the term category in v-instruction context.

- *.vv* integer vector-vector instructions: these instructions are defined for vectors that only contain integers. The subcategory *.v* specifies that only one source vector is involved in the instruction.

- *.vx* integer vector-scalar instructions: these instructions are defined for vectors that only contain integers and the computation involves a scalar given as an argument (an example: adding 3 on every element of a vector).

- *.vi* integer vector-immediate instructions: these instructions are defined for vectors whatever the type of the elements. The computation involves an immediate value given as an argument which is not loaded from the registers.

Many more categories of v-instructions are defined in the RISC-V ISA specification but the instructions used are part of these three. Only 11 v-instructions where needed to achieve a Keccak computation:

- *vmv.v.v* which copies a source operand to a vector register group.
- *vsetvli* which allows the user to set the length of a vector and the size of the elements.
- *vluxei.v* used to perform permutation in the vector (i.e. change element order in a vector).
- *vse64.v* and *vse32.v* used to store 32 or 64 bits vectors.
- *vle64.v* and *vle32.v* used to load 32 or 64 bits vectors.
- *vsll.vv* and *vsrl.vv* used to shift left or right elements in a vector.
- *vxor.vv* used to perform a *xor* between integer elements of two vectors.
- *vand.vv* used to perform an and between integer elements of two vectors.
- *vslidedown.vi* used to slide down elements of a vector from a defined scalar.

Since the assembly code is less intuitive than c or python code, macros were useful to simplify the use of common instructions like the ones listed above. A macro is a code that automatically makes a correspondence between a task wanted by the designer and the instructions related. All the macros used were stored in a file called vector_macros.h and you can see an example below that was used to simplify assembly coding:

```
//Vector store
#define VSTORE_U64(vreg) VSTORE(uint64_t,e64,vreg,Ru64)

// Macro to store a vector register into the pointer vec
#define VSTORE(T, storetype, vreg, vec)                                        \
  do {                                                                         \
    T* vec ##_t = (T*) vec;                                                    \
    asm volatile ("vs"#storetype".v "#vreg", (%0)\n" : "+r" (vec ##_t));      \
    MEMORY_BARRIER;                                                            \
  } while(0)
```

Figure 4.2: Macro for assembly v-instructions.

Considering that *v1* is a vector of 64 bit elements, *VSTORE_U64(v1)* was one macro which automatically makes the correspondence between the vector *v1*, the instruction *vse64* and *MEMORY_BARRIER*. *MEMORY_BARRIER* ensures that all vector stores have been committed before continuing with scalar memory operations. The macros are a good way to simplify debug and make the code more readable.

4.3 Keccak and vectors

KeccakF1600_StatePermute function is a state permutation function. Represented by f in figure 2.1, This function is used during the absorb, padding and squeeze steps of Keccak. This function is the computation part of the Keccak algorithm. As specified in its name, the state array (in white on figure 2.1) is 25 states of 64 bits thus 1600 bits in total. Its use aims at propagating fixed size block from the input into the 1600 bit state array in basically 4 steps:

- θ The Theta step provides a high degree of diffusion by introducing dependencies between different parts of the state. It involves computing parity bits for each column of the state array and then applying bitwise *xor* operations to update the state with these parity bits.
- ρ The Rho step involves rotating the bits in the state array to create a spiraling effect. Each lane of the state array is rotated by a predefined offset, determined by a specific mathematical formula.
- π The Pi step performs a permutation of the lanes in the state array. It rearranges the positions of the lanes based on a predefined pattern, thus introducing another level of diffusion in the state.
- χ The Chi step further increases the confusion in the state by applying bitwise operations. For each row of the state array, it performs bitwise AND, bitwise NOT, and bitwise *xor* operations with other elements in the same row.
- ι The Iota step is a special round that provides additional security by introducing a round constant. It performs a bitwise *xor* between a round constant and a specific position in the state array.

At the end of KeccakF1600_StatePermute, every single 1600 output bit of the state array depends on all 1600 input ones. In the c code version used by Thales, we can distinguish two steps: θ on one hand and $\rho\pi\chi\iota$ on the other hand. For a first assembly translation, the main difference between these two steps is the vector representation. θ can be done using vectors of 5 elements while the other step can be done with vectors of 25 elements. The beginning of θ step is shown below:

BCa = state[0] ⊕ state[5] ⊕ state[10] ⊕ state[15] ⊕ state[20]
BCe = state[1] ⊕ state[6] ⊕ state[11] ⊕ state[16] ⊕ state[21]
BCi = state[2] ⊕ state[7] ⊕ state[12] ⊕ state[17] ⊕ state[22]
BCo = state[3] ⊕ state[8] ⊕ state[13] ⊕ state[18] ⊕ state[23]
BCu = state[4] ⊕ state[9] ⊕ state[14] ⊕ state[19] ⊕ state[24]

v1 v2 v3 v4 v5

Figure 4.3: θ preparation.

⊕ operator is a xor. To achieve this first part of θ computation, 20 ⊕/*xor* instructions are executed on a simple core while only 4 are needed for a vector coprocessor as represented below.

V1 [state[0] | state[1] | state[2] | state[3] | state[4]]
⊕
V2 [state[5] | state[6] | state[7] | state[8] | state[9]]
⊕
V3 [state[10] | state[11] | state[12] | state[13] | state[14]]
⊕
V4 [state[15] | state[16] | state[17] | state[18] | state[19]]
⊕
V5 [state[20] | state[21] | state[22] | state[23] | state[24]]
=
[BCa | BCe | BCi | BCo | BCu]

Figure 4.4: θ preparation with vectors.

This representation of data can be used along the Keccak algorithm to group computation and thus instructions. All the steps of Keccak involve arithmetic logical operations, masked operations and permutations. To ease the translation of KeccakF1600_StatePermute in v-instructions I added a test at the end which checks the value of the output by using a reference obtained with a classical execution.

First results

After having obtained a first functional version, the results of executing a single KeccakF1600_StatePermute are summarized in the next figure:

Keccak_F1600_StatePermute	Arithmetic vector instructions	Total instructions	cycles			
			2 lanes	4 lanes	8 lanes	16 lanes
ARA	650	950	13200	11200	10600	10500
cv32a6	0	20400	30000			
cv64a6	0	8760	11000			

Figure 4.5: Performances of KeccakF1600_StatePermute on CVA6 and Ara.

Even if the number of instructions was divided by almost height, the results shows that the CVA6 alone is going 20% faster than Ara for 2 lanes. Regarding the 2 red cases, something went wrong in the design. After having double the number of lanes, the execution is not faster. In fact, this error is due to a critical operation call permutation.

I thus identify two ways of improvement for the Keccak execution on Ara: find a better way to perform permutations and implement **zbb** extension at vector level including the rotation left instruction, *vrol*. Since the implementation of the permutation instruction requires a deep understanding of the design, I implemented first *vrol* that was easier.

4.4 Execution optimizations

4.4.1 *vrol* implementation

As introduced in section 2.3, the extension **zbb** was able to boost the performance of Keccak execution on CVA6 by compiling a *rol* instruction. *vrol* is the equivalent in vector representation. It performs a rotation left on each element of a vector.

This instruction massively used in Keccak is mathematically equivalent to:

$$ROL(a, offset) = [(a << offset) \quad xor \quad (a >> (64 - offset))]$$

Where a is a 64 bits element of a vector, $offset$ is the size of the left part to be rotated left, $<<$ is a shift left, $>>$ is a shift right and 64 is related to the size of a. A simple example illustrates the principle of a rotation:

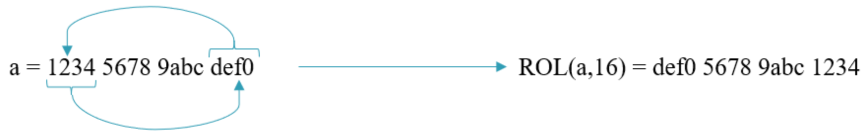

Figure 4.6: ROL(a,16): rotation left of 16 on a 64 bits vector represented in hexadecimal.

Without *vrol*, the compiler translate a rotation in a successive shift left, shift right and xor:

$$\begin{array}{llll} vsll & vd1 & a & offset \\ vsrl & vd2 & a & 64-offset \\ vxor & vd & vd1 & vd2 \end{array}$$

with *vrol*:

$$vrol \quad vd \quad a \quad offset$$

Based on the definition of *vsrl*, *vsll* and *vxor* but also the definition of *rol* in CVA6 design, the implementation of this instruction has been done by modifying a block of Ara design called **simd_alu**. This block performs all the arithmetic instructions and is located in the lanes as described in section 3.2.1. To implement a new instruction, I had modify the hardware rebuild the compiler and adapt the compiler:

1 Compiler: Since the c code is not totally wrote in assembly, a part of the code is compiled by either **gcc** or **llvm** compiler. To translate code into binary, the compiler must be able to recognize the Mnemonic of the instruction either this instruction has been code in c code or assembly. Hopefully, the last commit of **llvm** was able to recognize *vrol* instruction to translate it into binary code.

2 Modifications in the design (Ara here):

- Dispatcher/Decoder: I added a section to decode the instruction and set up the good parameters in the design (set to 1 the flag that indicates the use of $vs1$, of $vs2$, of the mask, of the **simd_alu**...)
- The packages: The package contains few files that reference all the parameters, the instructions and the coding object used in the design. I added an opcode, a format, a category and the name of the instruction to the package (as explained in section 4.2).
- Arithmetic-logic Unit: Since *vrol* is an arithmetic instructions (it performs an arithmetic operation on each elements of a vector), I implemented this instruction in the **simd_alu**

(reference to section 3.2.1-Lane) following the SIMD flow. For genericity, the instruction must manage all type of element (from 8 to 64 bits).

3 Testing: Once the instruction implemented and the design built by a RTL simulator, it was time for debugging. With the help of tests varying the length of the vectors and the size of elements, I adjusted the RTL design implemented and validated the functionality of the instruction. I personally used **GtkWave** for debugging since it was the open-source software used in the testing flow of Ara. A further validation is possible by using the Continuous Integration test bench (CI) and the complete validation is then achieved by a verification team.

3 Use: Once the instruction validated with functional tests, I was able to use it in the v-assembly code of the KeccakF1600_StatePermute.

This first implementation highlighted the challenges to implement an instruction and the architecture of Ara.

4.4.2 Permutation with *vluxei*

As introduced in section 4.3, the second improvement focuses on permutations. Discussions with members of the Open Hardware Group, were necessary to find other possible way of implementation for this specific instruction. A permutation is represented on the next figure:

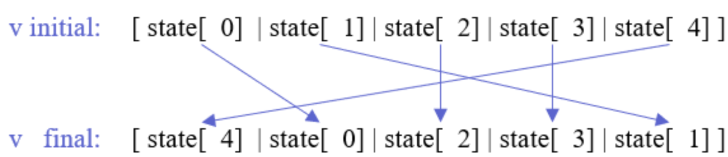

Figure 4.7: Vector permutation.

Permutations are used in every step of the KeccakF1600_StatePermute function. These operations are needed to shuffle elements in a vector, to ensure dependency between elements and at the end, bits. The operation was initially done with two instructions.

In the Ara processor as available now in open-source, permuting elements in a vector involves the successive *vse64* and *vluxei* instructions.

$$\begin{array}{llll} vse64.v & vs1 & (STATE) & \\ vluxei.v & vd & (STATE) & vs2 \end{array}$$

In Ara's hardware, the permutation is done by first storing all the elements of a vector (*vs1*) in the external memory (figure 3.1). To store a vector, the design needs a start address which is specified by the variable $(STATE)$ as used above. The second source vector *vs2* contains the indexes needed to load back the elements from the memory. The loaded back elements are then stored in the vector register files using the destination vector. Mathematically the access is done with the formula:

$$vd[i] = external\_memory[(STATE) + vs2[i]] \quad where \quad i \in [0 : NrElem]$$

The indexes from *vs2* represents the offset with respect to the start address. As the biggest vector size is 25 in Keccak, we can code the indexes on 8 bits. *NrElem* is the number of elements in *vd*. A permutation using *vluxei* is completely dependant of VLSU unit.

The flow involves several blocks and can be described as follow (according to figure 3.1):

vse64

1 Dispatcher: the instruction is decoded through the decoder and flags/parameters are configured to be sent to the rest of the design in this case the instruction in the decoder is *vse64*.

2 Lane: *vse64* is decoded and the information goes directly in the lanes and more precisely in the operand requester.

3 The operand requester manages hazards during the instruction flow like a store and a write requiring the same register access and thus the same bank. Indeed, each lane has 8 memory accesses to its vector register files. Once the eventual hazards solved, in each lane, the operand requester requires an access to VRF to read the elements from *vs1*. Elements are then transmitted to the VLSU in a queue with their respective storing addresses.

4 The VLSU write elements in the external memory through an in/out 128 bits bus.

vluxei

1 Dispatcher: Ara waits until the VLSU is done. Then, *vluxei* is decoded and inserted in the instruction queue.

3 Operand requesters this time requires to read elements from *vs2* that are located in the VRFs. The elements are then transmitted to the VLSU in a queue as indexes.

4 The VLSU read elements from the external memory using the indexes just received from the lanes. The data is splitted into the lanes and stored back in the VRF.

The issue with these instructions to do a permutation, is the use of the external memory and the VLSU unit. Since the external memory is a shared memory between Ara and the CVA6, the VLSU must be done before launching the execution of a new instruction. In fact, when an access to the external memory is required, hazards could occur, and exception could be raised by Ara: Ara needs to wait until the vector is fully loaded into VRFs before starting arithmetic computation. These permutations are the limitations of of the first Keccak execution summarized in figure 4.5.

4.4.3 *vrgather* implementation

In the RISC-V v-instruction specification, the permutation should be achievable by another instruction called *vrgather*. The difference between *vrgather* and *vluxei* is that *vrgather* does not use the external memory. In opposition to *vluxei*, *vrgather* shuffles elements of a vector by only using VRFs. However, in the current design of Ara, *vrgather* is missing. Ara is still under development so an experimental implementation was necessary to compute the potential gain of performance that could be achieved with such an instruction.

Going deeper in the design of Ara, the data coming from the external memory is managed this way:

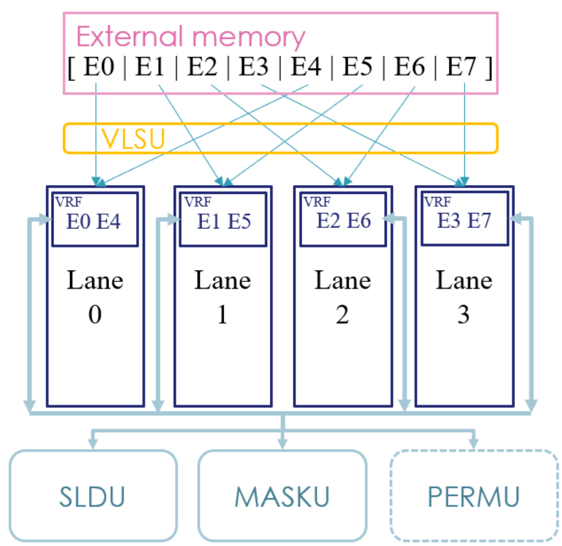

Figure 4.8: VLSU loads and dispatches elements from external memory to the VRF.

Depending on the number of lanes, the data is distributed in the VRF. The downside of having separated registers is that data cannot pass from one lane to another without passing by a block extern to the lanes (connection shows with the grey arrows). For a prototype, there was no need to be RISC-V ISA compliant since the purpose of this implementation was to evaluate the performance that could be achieved. The idea of the experimental *vrgather* is described in the next figure:

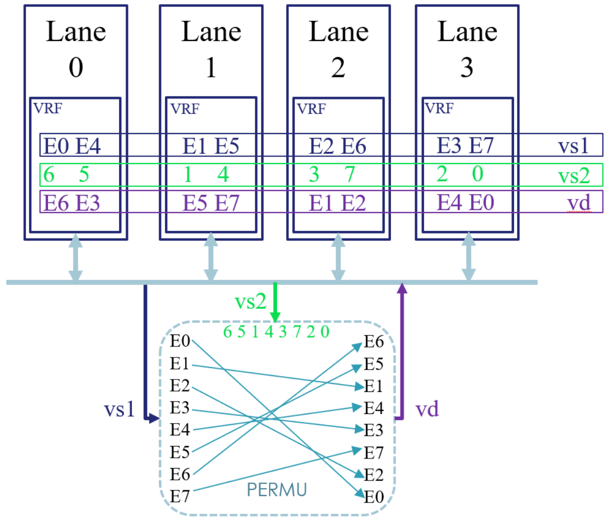

Figure 4.9: Permutation with **permu** block.

The permutation in this case is done by a block called **permu**. This block is located at the same level than the slide, the mask and the vector load/store unit. **permu** block is connected to the VRF and is enabled in a block called **valu** which initially manages the **simd_alu** which performs computation for arithmetic instructions. The block shuffles elements like an address decoder in 2 steps and it follows the arithmetic computation flow (64 bits by 64 bits):

1 Read: the vector including the address and the vector to be permuted are loaded element by element.

2 Decode and write: depending on the value of the address, elements are written in the right vector register file in the right lane.

After being written back in lane's registers, data processing can continue but one issue make this implementation not compatible with the official RISC-V ISA specification: in this prototype, indexes cannot be redundant to avoid overwriting. Since elements are loaded one by one, there is no memory on the previous indexes used.

This choice was easier to implement but it implies a specific declaration of indexes: the indexes at place i in the index vector $vs2$ represents the location that the element at place i in $vs1$ will take in the destination vector vd. At the end:

$$vd[vs2[i]] = vs1[i]$$

To briefly compare, in the RISC-V ISA specification, *vrgather* mathematical formula is:

$$vd[i] = vs1[vs2[i]]$$

Values of $vs2$ corresponds to the index of the value taken from $vs1$. Also, the spec specifies that redundancy of indexes should be managed. The challenges by designing this block were to choose the right interface with the lanes, the VRF and **valu** in order to respect the flow of data organized by the operand requester.

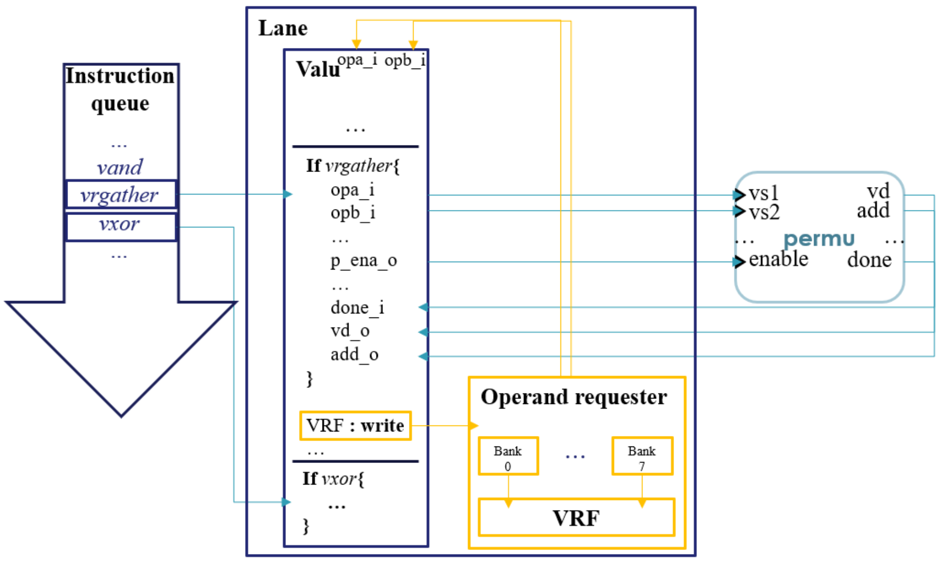

Figure 4.10: Prototype of *vrgather* using **permu** block.

The solution was to implement a if statement in **valu** which enables the use of **permu** thanks to a signal ”p_ena_o” when encountering *vrgather* in the instruction queue. Once the **permu** block enabled, the operand ($vs1$ and $vs2$) requested by the operand requester are sent from **valu** to the **permu**. The permutation block computes two outputs for each lane and more precisely each **valu**. One output is the value ”vd” and the other is an address ”add”. In the simple case of figure 4.9 with 4 executive units the address is 0 or 1 to indicate if the value must be stored in the first or the second position in the VRF. These two signals received by the **valu** are sent to the VRF using the **valu** flow. The instruction was also implemented in the decoder and the package following the design flow explained in the previous section.

The cycles needed for a permutation is perturbed by few dead cycles due to conflict when accessing the VRF with the banks. In fact, if **valu** requires a write to the operand requester while a read is performed at the same time to prepare an operand for the next instruction, arbiters delay from one dead cycles the read which has not the priority with respect to write request.

4.5 Execution time

If I consider that *vrgather* is optimized without dead cycles, the number of cycles for a permutation closely follows the same formula than arithmetic instructions:

$$Cycles = \begin{cases} \frac{SizeElem \times NrElem}{64 \times NrLanes} & \text{if NrElem is even} \\ \frac{SizeElem \times NrElem}{64 \times NrLanes} + 1 & \text{if NrElem is odd} \end{cases}$$

In the second computation part of Keccak, $SizeElem = 64$ and $NrElem = 25$ so:

$$Cycles = \frac{NrElem}{NrLanes} + 1$$

In the next table which summarizes the results of one KeccakF1600_StatePermute execution, **ARA + *vrgather** + *vrol*** are the performances effectively obtained with the vrgather prototype instruction while **ARA + *vrgather* + *vrol*** with the and the orange police are the maximum performances achievable with the previous formula assuming that vrgather is optimized and does not induce dead cycles.

Keccak_F1600_StatePermute	Arithmetic vector instructions	Total instructions	cycles			
			2 lanes	4 lanes	8 lanes	16 lanes
ARA	650	950	13200	11200	10600	10500
ARA + ***vrgather**** + ***vrol***	530	840	8400	5300	3400	2300
ARA + ***vrgather*** + ***vrol***	~530	~840	~7800	~4700	~3100	~2000
cv32a6	0	20400	30000			
cv64a6	0	8760	11000			
in-house accelerator	0	[redacted]	few hundreds			

Figure 4.11: Performances using *vrgather* and different Ara configuration.

As shown in the previous formula, the execution cycles are no more limited by permutations and VLSU block but by the number of executive units in parallel (the number of lanes). Matteo Perotti from Open Hardware Group proposed me to conduct further development of *vrgather* to make it RISC-V compliant and potentially contribute to Ara project. Even if the performances were better than the one obtained with the cv64a6 more development were conditioned by the synthesis of the design.

4.6 Resource optimizations

Performances obtained executing Keccak algorithm on Ara seems promising compared to the one obtained executing on CVA6. However, the number of cycles needed for an execution remains higher than the in-house ASIC accelerator. Indeed, the performances are not comparable since the in-house accelerator involve an ASIC dedicated to the Keccak. However, two properties of Ara remain promising for it use in pin cards.

- The genericity: Ara uses an Instruction set architecture that is specified by the RISC-V community. Thus the accelerator could be used to improve the execution of other algorithms of the PQC Dilithium.
- The security: the vector instruction set is compatible with the use of mask that are very useful to improve the security of the Keccak execution.

Despite these technical upsides, synthesis remains critical for further development of Ara. RTL synthesis is the process of converting high-level hardware descriptions (RTL) into optimized gate-level net lists using a synthesis tool and a technology library. It involves logic optimization, technology mapping, and timing analysis to meet design constraints. Then, the result of synthesis, a gate-level netlist, is used for further verification and physical design stages before chip fabrication. This stage is critical to decide the pertinence to use such an accelerator.

Ara synthesis has been done by adapting the CVA6 synthesis flow. Available on CVA6 open-source repository, the library pd/synth is highly parametrizable to choose the constraints to apply during a synthesis. The synthesis was done with a ST technology.

4.6.1 Area

In Ara design, the resources are largely concentrated into the lanes so the priority is to optimize the lanes by removing the useless parts. One of the biggest unit in Ara's lanes is the Floating point unit. Dilithium algorithm and more broadly cryptography algorithms does not use floating point computation. Only unsigned integers are used in Keccak so this unit is optional. Also, another optimization can be to remove unused instructions from the design. However, optimizing the design bring us closer to a customized accelerator than a generic one. By keeping only the used instructions and removing various arithmetic instructions the differences between Ara and its optimized version are shown in the next table.

Synthesis		Area in **kg** (NAND 326)	
		2 lanes	4 lanes
ARA + ***vrgather***	Lane (without VRF)	906	1759
	permu	1	2
	sequencer	3	3
	SLDU	23	41
	VLSU	53	85
	MASKU	47	60
Total		1033	1950
in-house accelerator		~70	~70

Synthesis		Area in **kg** (NAND 326)	
		2 lanes	4 lanes
ARA Optimized + ***vrgather***	Lane (without VRF)	168	321
	permu	1	2
	sequencer	3	3
	SLDU	23	41
	VLSU	53	85
	MASKU	47	60
Optimized Total		295	512
in-house accelerator		~70	~70

Figure 4.12: Ara synthesis with and without optimizations.

The optimizations concerning the FPU and the instructions represent an area gain of almost 80%. However, the optimized Ara remains 4 times bigger than the recent accelerator and also 10 times slower with 4 lanes in term of execution performances as shown in figure 4.11. The area was computed without the VRF.

After discussing with Jean-Roch, we noted that the area is way too large to be acceptable even if Ara is more generic than the in-house accelerator. Thus, having computed the performances achievable with a vector coprocessor, further development would be to rearrange Ara's design and to find other Dilithium computation that could use v-extension. By grouping resources, it would be possible to obtain area performances that, combined with the upside of genericity, could concurrence the in-house accelerator. The goal of the last part of the master thesis is to find a trade-off between the resources needed, execution performances and genericity to give further perspective of development for an embedded vector accelerator.

Chapter 5

Embedded vector co-processor and conclusion

Currently, there are two main vector co-processors available in open-source, Ara and Vicuna. A last one named Spatz [4], is still under development. According to the above study I will discuss the Spatz co-processor and give further perspectives of development for its design.

5.1 Design

Spatz is a 32 bits co-processor announced in 2022 by the Open Hardware Group as a light and embedded vector coprocessor.

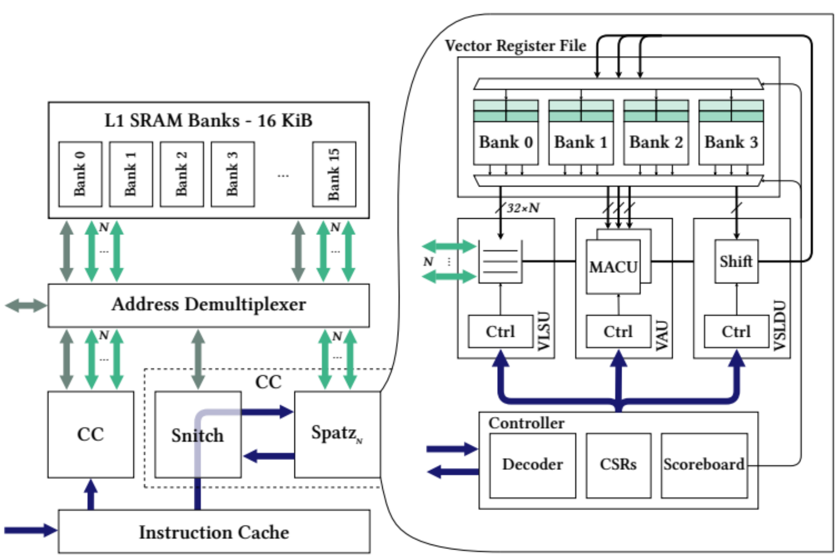

Figure 5.1: Spatz design [4].

Spatz design respect vector processor standards but some points diverge from Ara:

Controller

As introduced with Ara, there are two possibilities for decoding v-instructions: this task is either done by the connected CVA6 (in its decoding unit) or by the vector co-processor. To ease the implementation and have a better control on the implemented instructions the choice for Spatz was also to include the decoding part in the co-processor design. One main difference with Ara is that instead of using arbiters to manage the potential hazards in memory accesses as seen is Ara's design, Spatz uses a Control Status Register. This solution seems to be particularly suited for embedded vector co-processor since it requires less than 16 kg for 4 executive units. There is no equivalent in Ara design since this part groups the sequencer, the arbiters which are in Ara's lanes and operand requester/sequencer.

Vector Register File

In Spatz, the centralized Vector register file (VRF) enables efficient vector permutation instructions like *vrgather* or vector slides using a barrel shifter, avoiding the scalability challenges of lane-based vector architectures like Ara. The VRF proposed here is composed of four 3R1W banks, thus 4 access to the registers are possible in a row. Since we consider that the masks are loaded at the beginning of the computation, the VRF ports can read 2 operands plus a mask to produce one result. Spatz's design proposed 3R1W banks without taking mask into account.

VLSU

In Spatz, the VLSU could handle both external memory interface and VRF interface, supporting unit-strided and constant-strided accesses. An idea would be to have a variable number of 64-bit wide memory interfaces, typically matching the number of MACUs. In Spatz design, the independent and narrow memory interfaces could reuse the same 64-bit wide L1 SPM interconnect as scalar cores, enabling fast constant-strided and potential scatter-gather execution. To ensure ordered memory responses, a Reorder Buffer (ROB) sits between memory interfaces and the VRF, writing responses as ordered words to the VRF, simplifying chaining in the scoreboard.

MACUs

As an example, in Ara's design, a lane optimized represents 80 kg and is the costliest part of the lane in term of resource. Based on this analysis, in a further competitive accelerator only the simd_alu (alu used in the lanes of Ara) will be placed in parallel. All the other units in the lanes should be grouped together. Clearly then, MACUs are the main functional unit in Spatz's coprocessor and contains only the ALUs. To still be compliant with the vector extension, Spatz's MACUs support 8-bit, 16-bit, and 32-bit elements (for mask and reduced integer instruction). As configured now, each MACU processes 32 bits per cycle, employing packed-SIMD execution like Ara. In Spatz, the ALU could feature four data paths, one 64-bit one 32-bit, one 16-bit, and two 8-bit, sharing wide data paths for narrow operations.

VSLDU

The Vector Slide Unit (SLDU) remains the same, it will handle instructions requiring simultaneous access to all Vector Register File (VRF) banks. It manages tasks like element insertion, extraction, vector shuffling, and sliding. Please refer to section 3.2.1-SLDU for more details. Also, as discussed previously, a permutation on a vector with 25 elements of 64 bits can use indexes coded in 8 bits. So, it's important to have data paths with different size that can be accessed at the same time.

MASKU

Not yet implemented in Spatz or Ara, mask unit is still under development and could be particularly useful for cryptographic applications. In fact, as explained in [12], adapted algorithms can use masked instructions to improve their security against attacks that could be conducted by Quantum Computers. Basically, this unit will be able to manage the use of masks in the flow of instructions executed.

There is currently not enough documentation on this part but it would be interesting to investigate on an implementation of a Mask unit in Spatz design.

5.2 Optimizations

Since in Dilithium sub-algorithms performs calculation over 64 bits unsigned integers and regarding to the performance of figure 2.3 it would be interesting to convert Spatz 32 bits into a Spatz 64 bits. It would be also interesting to connect this accelerator to a cv32a6 core to benefit from both the small size of cv32a6 to perform small computation, and the power of a 64 bits coprocessor for long and heavy computation.

Moreover, Spatz is already based on the Zve32x subset of RVV version 1.0 which contains only integer vector instructions. Thus, a designer who wants to have a design operating in 64 bits could use Zve64x as a base. However, this base implements useless instructions that increase the area of the design. In a further development, it would be necessary to choose the instructions to implement to limit the size of the coprocessor. The co-processor must be optimized keeping in mind that the genericity of the RISC v-instruction is the main upside in favor of using this coprocessor as an accelerator but also what increases the area.

Eventually, the last improvement will concern the MACU. This part of the design is considered as a block in the design and its use prevent the design from pipelining arithmetic instructions. As an example, if a MACU is used for a *vand*, the controller will wait for this instruction to be executed before launching the next one. By dividing the MACUs in blocks (one for AND, one for XOR...) it would be possible to chain instructions using different blocks and increase the throughput. However, as introduced with the size of the VRF above, in our case, each independent vector added (1600 bits) to chain instruction will result in an increase of the VRF. With 1kiB and the register computed in the next section, it would be possible to add $16000 - 6300 = 9700$ bits which corresponds to 6 vectors. The schematic below represents this potential improvement:

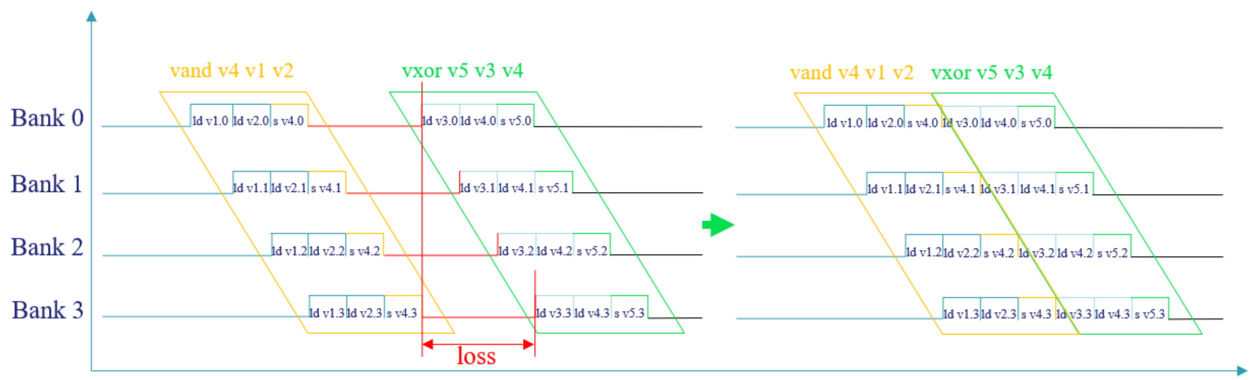

Figure 5.2: Execution with *vxor* inside (left) and outside (right) of the MACU..

This improvement focuses on the execution speed while keeping the genericity of the RVV ISA. It is not an optimization for area since the new block containing the resources to compute a *vxor* must be linked with the rest of the design with buses and connections. In the Keccak algorithm, the long computations mix permutations, slides and arithmetic so this improvement would target other algorithms.

5.3 Performances

5.3.1 Synthesis estimation

A synthesis of Spatz is computed in [4]. It details all the resources needed to build an embedded coprocessor. Considering the results obtained in figure 4.11 and 4.12, adding two lanes almost double the area of the design whereas it does not divide the execution time by two. A good trade-off between area and execution time would be to choose an architecture using 4 lanes.

According to Spatz's paper, the 2kiB-large VRF represents an area of 201kg for an architecture of 4 lanes. However, only the constant of the rotation/permutation and the masks need to be stored in the register to be directly used during the computation. When counting all the bits needed to store these vectors, the result is around 1500 bits. In addition to this, we must add 1600 bits for $vs1$, $vs2$ and vd. The smaller VRF achievable without using the VLSU and the external memory will be 6300 bits. 4 banks and 1kiB will be enough and its associated size is almost 100 kilo-gates. In our Keccak case, adding a register sized for a vector of 1600 bits results in an increase of the VRF. With 1kiB, it would be possible to add $16000 - 6300 = 9700$ bits which corresponds to 6 additional vectors.

The rest of the design such as the MACUs and the controller, or the bus must be converted to 64 bits. This change in the design does not necessarily result in doubling the area. Optimization's techniques can limit the increase in area so it is difficult to properly estimate this.

Synthesis Spatz 64b	Area in **kg** (GlobalFoundries' 22FDX FD-SOI)	
	2 MACUs	4 MACUs
MACUs	<95	<165
VLSU	<30	<55
OTHER	<14	<32
Spatz64 Total (without VRF)	**<150**	**<250**
VRF	100	100
Spatz64 +VRF Total	**<250**	**<350**
in-house accelerator	~70	~70

Figure 5.3: Area estimation based on Spatz design [4].

The major improvement is the size of the executive unit which represent less than 95 kGE (kilo gates equivalent) while in Ara's design, 4 units optimized have an area of 320 kGE. Spatz is thus closer to an embedded vector coprocessor.

The results summarized in the previous figure are only an estimation of what would be the resources needed for such a vector coprocessor. The results are based on Spatz [4], Vicuna [13] and Ara [3] documentation and investigations.

5.3.2 Execution estimation

According to the work done on Ara and the proposal in this chapter, the vector co-processor performances are dependent on the number of processing units that can be run in parallel. A designer carrying further development over vector accelerator should keep in mind the next formula which fix the maximum execution performance achievable with a 64-bit vector coprocessor:

$$TotCycles = (\frac{NrElem}{NrLanes} + 1) \times NrVinstr$$

Where $NrElem = 25$ which represents the number of elements of the vector processed and $NrVinstr$ is the number of vector instructions. I also assume that the size of the element corresponds to the configuration of the vector coprocessor.

5.4 Conclusion

Keccak_F1600_StatePermute	Arithmetic vector instructions	Instructions	cycles		Area kGE	
			2 lanes	4 lanes	2 lanes	4 lanes
ARA (without VRF)	650	950	13200	11200	~290	~510
ARA + ***vrgather**** + ***vrol*** (without VRF)	530	840	8400	5300		
ARA + ***vrgather*** + ***vrol*** (without VRF)	~530	**~840**	~7800	~4700	**~290**	**~510**
Vicuna	~530	**~840**				
Spatz64 (without VRF)	~530	**~840**			**<150**	**<250**
cv32a6	0	20400	30000		<180	
cv64a6	0	8760	11000		<500	
in-house Asic accelerator	0		few hundreds		~70	

Figure 5.4: Summary including the maximum achievable performances of the 3 current vector coprocessors.

The previous table summarizes all the performances discussed above. Obviously performances of Vicuna, Spatz or Ara with a *vrgather* optimized are just estimations.

Depending on the number of execution unit, the performances of Spatz summarized in figure 5.4 are better than the ones obtained with Ara. However, the study shows that optimizing resources and execution performances will bring the coprocessor design closer to a customized accelerator which implies a loss of genericity.

Chapter 6

Conclusion

When designing an accelerator, there are three main concerns to guarantee its integration in Thales products and its competitiveness on the market.

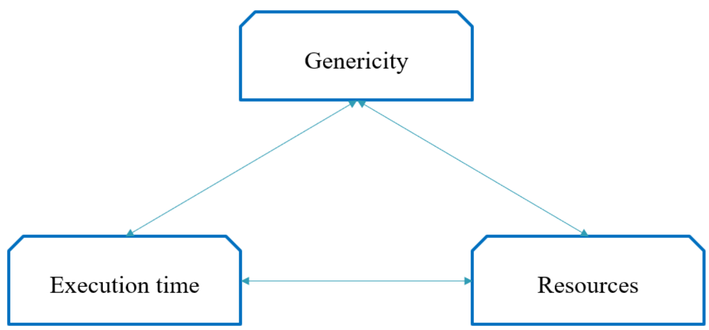

Figure 6.1: Three main figures of merit of harware design.

- The resources of the product and the energy consumption are critical to fix the price of the product on the market and its embedded capability in a pin card.
- The execution performances must be comparable to the one obtained by the main concurrent.
- The genericity is important for an accelerator using standardised ISA. Optimising resources and execution time push the designer to implement only the useful instructions. These optimizations are a loss of genericity and thus a difficulty in reusing the accelerator for other algorithms. Moreover, the genericity is associated to the use of instructions and thus the possibility of using masks. So in cryptography genericity implicitly groups security and adaptability.

Reducing the resources needed is obviously important in a perspective of sustainable development and economy of primary resources. This point may be the main drawback of microelectronic field. Globalization of semiconductor industry has made standardization of sustainable development difficult. Several countries are struggling to remain competitive by adding additional requirements in this area.

Eventually, according to the discussion done in the last section, I proposed few improvements of the existing vector coprocessors. Regarding the performances of the in-house accelerator, I did not find an obvious trade-off between genericity, execution time and resources in favor of using a vector coprocessor as an accelerator for Keccak. Thus **I can conclude that recent vector coprocessors are not yet suited for Thales applications**. Indeed more researches are needed to build a vector co processor able of fulfilling the requirements and the specifications of Thales products. Since v-extension is a specific type of calculation, there may be Ara is sub-optimised in term of area so its used as available in open-source is restricted to application which are not embedded.

This internship was an opportunity to have a deeper knowledge of what is digital design, the organisation of microelectronic industry and the challenges of open-source R&D. I was brought to collaborate with international teams and help my colleagues to undertake parallel tasks. Thanks to the professionalism of the team I was rapidly integrated in the conducted projects what made this final step of my engineer's degree a very formative experience.

My investigations helped Thales SAS Invia to status on the use of a vector coprocessor as an accelerator and my work including tests, implementations, and documentations was designed to be kept and reuse by the team.

Personally, the dynamic around RISC-V cores and the growth of the international community are indicators that let me think that vector cores based on RISC-V ISA may one day reach the performances needed to design competitive embedded products.

Abstract

Research over vector co-processors for cryptographic algorithms execution has been introduced in this master thesis. The work focuses on an algorithm called Keccak, used during the different step of the Post Quantum cryptographic algorithm Dilithium. To evaluate the performance of Keccak execution over a vector co-processor called Ara, there was a need to implement prototype instructions in Ara hardware. This preliminary research work aims at measuring the relevance of such an architecture in the products developed by Thales.

The work conducted during 6 months has shown that Ara co processor, as available now, is not suited for embedded electronic products. If Ara is configured with two execution units, with the permutation instruction, the gain with respect to a cv32a6 core is about 70%.

However this gain is not enough to compete against the ASIC accelerator recently used by the in-house core and moreover, the result of Ara synthesis is that it represents more than 300 kilo gates equivalent based on ST C28SOI technology which is also not suited for embedded electronic products. Eventually, an architecture could be designed by mixing Ara a vector coprocessor named Spatz to fulfill Thales requirements in terms of execution speed and resources. Nevertheless more researches are needed to obtain a mature and competitive technology.

French

Cette thèse de master est un travail d'investigation sur les co-processeurs vectoriels pour l'exécution d'algorithmes cryptographiques. Ce travail se concentre sur un algorithme appelé Keccak, utilisé lors des différentes étapes de l'algorithme cryptographique post-quantique Dilithium. Pour évaluer la performance de Keccak sur un co-processeur vectoriel appellé Ara, des implémentations de protypes d'instructions dans le modèle préexistant ont été nécessaires. Ce travail préliminaire a pour but de mesurer la pertinence d'une telle architecture pour les futures produits de Thales.

Ces 6 mois de stage ont servi à montrer que le processeur Ara tel que disponible actuellement n'est pas adapté pour de l'éléctronique embarqué. Si Ara est configuré avec deux unités d'execution, ainsi que l'instruction qui permette les permutations, le gain par rapport à un processeur cv32a6 est d'environ 70%.

Cependant ce gain n'est pas suffisant pour concurrencer l'accélérateur Asic utilisé dans les produits actuels. Aussi, même en optimisant Ara, sa synthèse l'évalue autour de 300 portes logiques équivalantes en se basant sur une technologie ST C28SOI ce qui est supérieur aux performances attendues. Finalement, un accélérateur vectoriel pourrait être conçue en mélangeant Ara et une autre architecture appelée Spatz pour répondre aux exigences de Thales en terme de rapidité d'exécution et de ressources. D'autres recherches sont néanmoins nécessaires pour obtenir une technologie mature et compétitive.

Italian

Questa tesi di master è un lavoro di indagine sui co-processori vettoriali per l'esecuzione di algoritmi crittografici. Questo lavoro si concentra su un algoritmo chiamato Keccak, utilizzato nelle diverse fasi dell'algoritmo crittografico post-quantistico Dilitio. Per valutare le prestazioni di Keccak su un co-processore vettoriale chiamato Ara, sono state necessarie implementazioni di protypes di istruzioni nel modello esistente. Questo lavoro preliminare ha lo scopo di misurare l'importanza di questa architettura per i futuri prodotti di Thales.

Questi sei mesi di tirocinio sono serviti a dimostrare che il processore Ara attualmente disponibile non è adatto all'elettronica di bordo. Se Ara è configurato con due unità esecutive, così come l'istruzione per le permutazioni, il guadagno rispetto a un processore cv32a6 è di circa il 70%.

Tuttavia, questo vantaggio non è sufficiente per competere con l'acceleratore Asic utilizzato nei prodotti attuali. Anche ottimizzando Ara, la sua sintesi lo valuta intorno a 300 porte logiche equivalenti basate sulla tecnologia ST C28SOI che è superiore alle prestazioni previste. Infine, un acceleratore vettoriale potrebbe essere progettato mescolando Ara con un'altra architettura chiamata Spatz per soddisfare i requisiti di Thales in termini di velocità di esecuzione e risorse. Tuttavia, sono necessarie ulteriori ricerche per ottenere una tecnologia matura e competitiva

Gantt Diagram

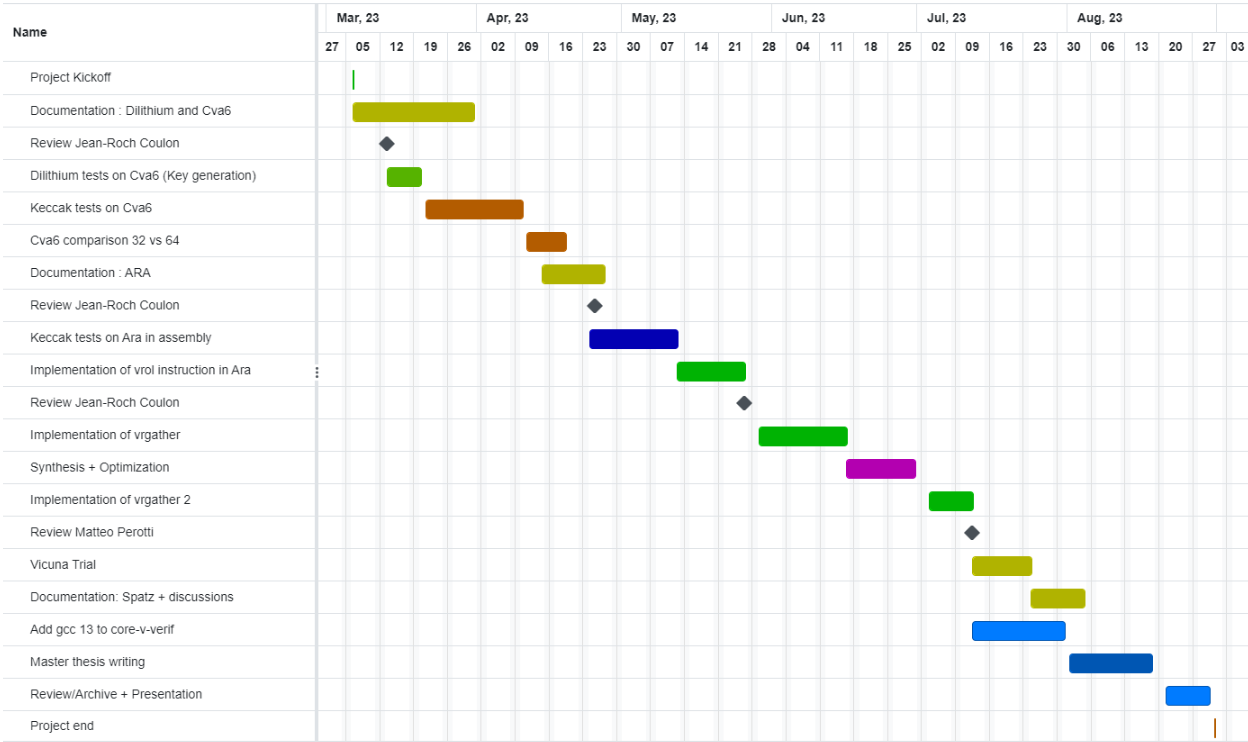

Figure 6.2: Gantt diagram: Vector extension internship.

Archive sheet

Confidential

Adding of a vector co-processor on an open-source RISC-V core for cryptographic algorithms execution

March 2023 - August 2023

Corentin MARAIS

2023 Master's degree in Micro- and nanotechnologies for integrated systems (MNIS)

INSTITUT NATIONAL POLYTECHNIQUE - PHELMA

http://phelma.grenoble-inp.fr

Under the supervision of :

Jean-Roch Coulon (jean-roch.coulon@thalesgroup.com)

Katell Morin-Allory (katell.morin-allory@univ-grenoble-alpes.fr)

Thales Silicon Security Arteparc Meyreuil, Bâtiment D Route de la Côte d'Azur 13590 Meyreuil France

Project description

Research over vector co-processors for cryptographic algorithms execution has been introduced in this master thesis. The work focuses on an algorithm called Keccak, used during the different step of the Post Quantum cryptographic algorithm Dilithium. To evaluate the performance of Keccak execution over a vector co-processor called Ara, there was a need to implement prototype instructions in Ara hardware. This preliminary research work aims at measuring the relevance of such an architecture in the products developed by Thales.

Responsabilities

- Install software, test and run it in autonomy (riscv, core-v-verif, llvm, gcc, vicuna...).
- Execute a cryptographic algorithm on Ara vector extension.
- Implement hardware to estimate the best performances achievable with a mature technology.
- Suggest improvements and further way of development on vector coprocessors.

Resources

The company gave me a complete access to the resources available: a computer, various software tools like vcs, verilator, gtkwave, textbook resources, secured servers and help by every employees when needed. I worked with the other engineers in an open-space.

Bibliography

[1] Semico, “Maket size evolution of risc-v cores..” https://semico.com/content/risc-v-market-momentum-building, 2019.

[2] Wikipedia, “Sponge construction, wikipedia.” https://en.wikipedia.org/wiki/Sponge_function.

[3] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, “Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[4] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini, “Spatz: A compact vector processing unit for high-performance and energy-efficient shared-l1 clusters,” in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[5] OHG, “Open hardware group website.” https://www.openhwgroup.org/.

[6] C. Paar and J. Pelzl, “Sha-3 and the hash function keccak,” *Understanding Cryptography-A Textbook for Students and Practitioners*, 2010.

[7] L. Beckwith, D. T. Nguyen, and K. Gaj, “High-performance hardware implementation of crystals-dilithium,” in *2021 International Conference on Field-Programmable Technology (ICFPT)*, pp. 1–10, IEEE, 2021.

[8] Five-embeddev.com, “The risc-v development forum..” https://five-embeddev.com/riscv-isa-manual/latest/riscv-spec.htm, 2019.

[9] P. Marwedel and G. Goossens, *Code generation for embedded processors*, vol. 317. Springer Science & Business Media, 2013.

[10] G. gnu community, “The gcc 13.1.0 manual..” https://gcc.gnu.org/onlinedocs/13.1.0/, 2023.

[11] Five-embeddev.com, "V-instructions set architecture.." https://five-embeddev.com/riscv-v-spec/draft/v-spec.html, 2022.

[12] T. U. of Munich, "Masked accelerators and instruction set extensions for post-quantum cryptography.." https://eprint.iacr.org/2021/479.pdf, 2021.

[13] M. Platzer and P. Puschner, "Vicuna: a timing-predictable risc-v vector coprocessor for scalable parallel computation," in *33rd euromicro conference on real-time systems (ECRTS 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[14] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, pp. 1–32, 2011.

[15] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v compressed instruction specifications.." https://riscv.org/wp-content/uploads/2015/05/riscv-compressed-spec-v1.7.pdf, 2015.

Annex

Invia

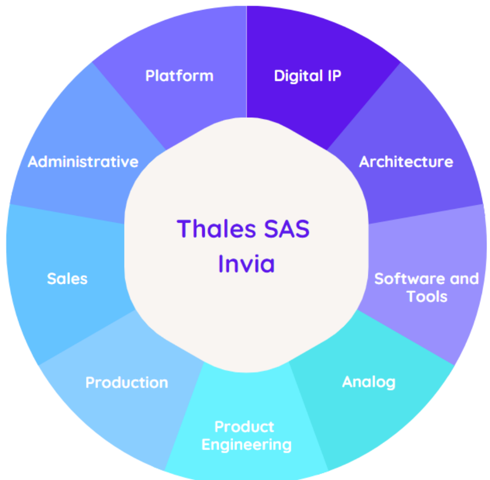

Figure 6.4: Thales SAS Invia organization.

- **Platform**: Assembly and verification of IP design.
- **Digital IP**: IP design.
- **Architecture**: RISC-V related development and specifications.
- **Software and Tools**: firmware development.
- **Analog**: Energy management, high frequency interface and safety related detector.
- **Product Engineering**: Product testing.
- **Production**: Carrying out a test program for production.
- **Sales**: Customer relationship, transforms the customer's need into a product.
- **Administrative**: Administrative tasks and ensures the internal functioning of the company (reception, human resources...).

Open Hardware Group organization

Open Hardware Group projects and activities are managed by two standing committees composed of comity members: the Technical Working Group and the Marketing Working Group. The Technical Working Group aims at steering the technical direction, the road map, and the conduct of projects. The organization is composed of sub-groups which are:

- The Core Task Group develops features and handles processors.
- The Verification Task Group performs core and IP verification.
- The SW Task Group develops and supports software tools, ports operating system and firmware to cores and IP.
- The Hardware Task Group is working on ASIC and FPGA target development and support of cores and IPs.

CVA6

The goal of the CVA6 project is to create a family of production quality, open source, application class RISC-V CPU cores. The CVA6 targets both ASIC and FPGA implementations, although individual cores may target a specific implementation technology. The design is written in SystemVerilog and is heavily parameterizable: parameters can set the ILEN to be either 32- or 64-bits and support for floating point can be enabled/disabled.

CORE-V is the name of the OpenHW Group family of RISC-V cores. CVA6 is the name of a GitHub repository for the source code for a set of application class CORE-V cores. The CV prefix identifies it as a member of the CORE-V family and the A6 indicates that it is an application class processor with a six-stage execution pipeline. However, the CVA6 “as is” is not intended to implement a specific production core. Rather, the CVA6 is expected to be the basis for several application class cores.

The CVA6 is enough powerful to run a Unix operating system like Linux. It is ”single, in-order issue”, the instructions are retrieved in memory (green part of Figure 5), they are decoded and prepared for their execution (grey part) in order and one by one. The CVA6 is ”out-of-order execute” meaning that the execution of the instructions (blue part) can be done in disorder. The ”commit” of the instructions (pink part) is ”in-order”, the result is retrieved in the initial order. Extensions are categories of instructions that can be enabled when choosing the ISA at compilation. The design itself is compatible with all these instructions and does not need a specific configuration to be able to run a set of instruction. To give an overview of the available extension, here are some examples:

- **M extension**: it includes the multiplication and material division of the non-privileged RISC-V specification [14].
- **F extension**: it includes instructions for the computation of floating point numbers.
- **C extension**: it includes the compressed instructions. More than half of the RISC-V instructions have a 16-bit compressed equivalent. This reduces the code size by more than 25%. This extension has a significant impact on performance and energy consumption according to the specification of this extension [15].
- **V extension**: it corresponds to the vector extension. This letter is supposed to enable the generation of v-instructions by the compiler. But considering the early stage of its development, I had to write the assembly code tu use the features of the vector extension.

To be fully compatible with the RISC-V specification, the core can be run in three preferred modes called privilege levels: M for machine, S for supervisor and U for user. Indeed, some instructions require a certain privilege to be executed. Level M, machine is the most privileged while

U is the least privileged. These levels are encoded in one of the registers. The register where these levels are encoded is located in one of the CSRs (Control and Status Registers). If an instruction decoded cannot be executed in the current core privilege level then an exception is raised.

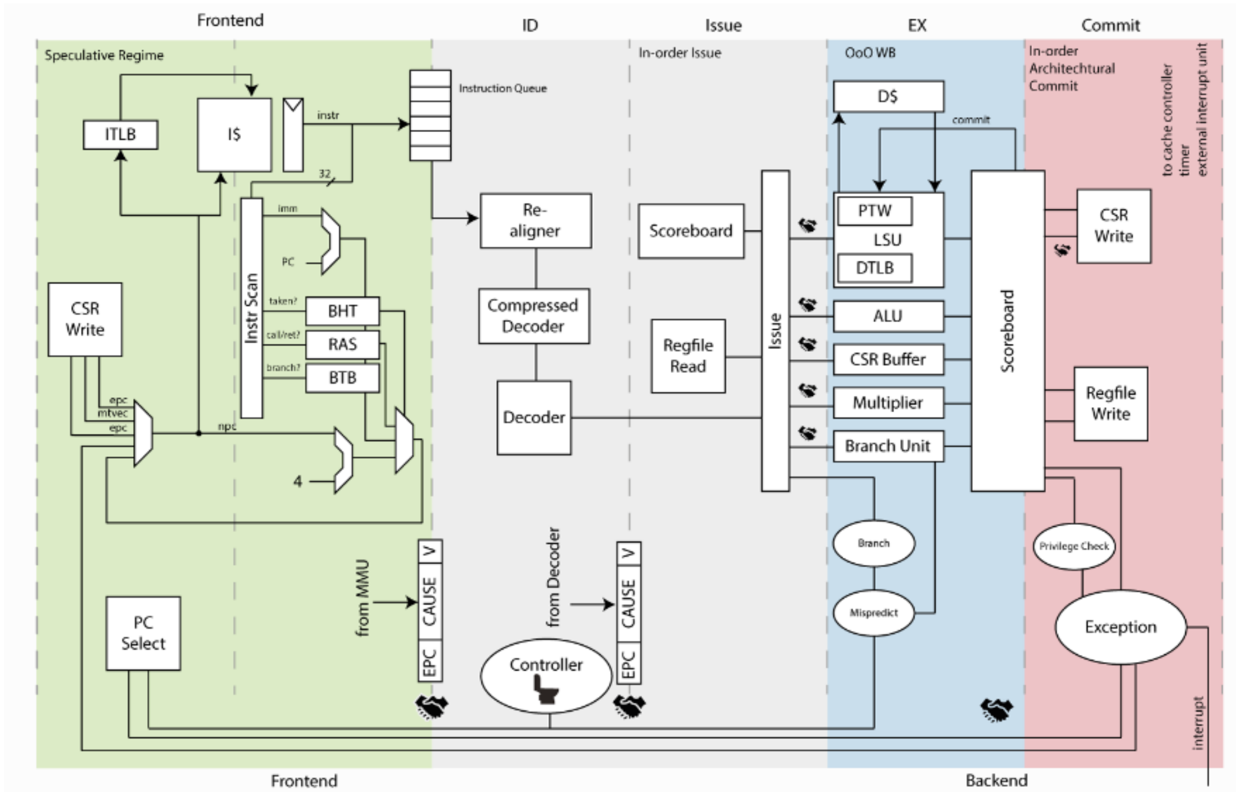

Figure 6.5: CVA6 design with its 6 pipeline stages.

Concerning the architecture, the CVA6 has a six-stage processing chain (pipeline) and a branch predictor for making speculative instructions. In the article[7] the different pipeline stages are described by Florian Zaruba and Luca Benini who are the main contributors for the Cva6 design. The 6 stages pipeline includes:

- **The Program Counter** (PC) generation stage determines the address of the next instruction to be fetched and executed, involving incrementing the PC, calculating branch targets and jump addresses, and handling exceptions. In the CVA6 this stage also includes a branch prediction features which aims at guessing the outcome of a branch before it is determinate. This feature is used to maintain instruction throughput and minimize stalls in the pipeline.

- **The Instruction Fetch** (IF) stage is time critical for the design. It obtains branch prediction information, current PC, and validity from the PC Generation stage, requests address translation from the Memory management unit (MMU), controls the instruction memory interface, manages speculative fetch requests, and stores fetched instructions in a FIFO. The FIFO enables the decoupling of the processor's front-end and back-end.

- **The instruction decode** (ID) stage is responsible for extracting, decoding, and sending instructions from the data stream received from the IF stage to the issue stage, including handling compressed instructions, realignment, branch prediction, and generating the fundamental control structure for further processing in the pipeline.
- **The Issue stage** includes a scoreboard and the instruction queue to retrieve the operands and prepare the execution of the instruction. When everything is ready the instruction is sent to the Execute Stage
- **The Execute stage** includes the execution units which have fixed or variable latency. As an example, the arithmetic and logical unit, the multiplier and divisor, the manager of the ”Control Status Registers” have a fixed latency whereas FPU, Floating Point Unit and Load and Store Unit (LSU) have a variable one. Conflict management to record results is done by the last stage called commit stage. Therefore the execution can be done out-of-order.
- **The Commit stage** is the stage where the results are written in the registers (General Purpose Registers or Floating Point Register). This stage checks the order in which the instructions are ”commited”.