
Politecnico di Torino

Master's Degree in
Electronic Engineering

# A RISC-V based accelerator for NFC Signal Processing

**Supervisors**
Prof. Guido Masera
Dr. Stefan Brennsteiner

**Candidate**
Francesco Babbaro
S287465

December 2022

*To you, if you have stuck with*
*me until the very end.*

*A te, se sei rimasto con*
*me fin proprio alla fine.*

# Abstract

Near Field Communication (NFC) is a ubiquitous technology with many applications ranging from identification to ticketing, from mobile payment to logistical solutions and, just recently, wireless charging.

Every modem unit must be able to support multiple communication standards in different operative conditions. In many cases, this is achieved employing an extensive use of custom Digital Signal Processing (DSP) logic at the expense of functional flexibility.

One way to balance chip resources with the growing need for flexibility is to consider the adoption of microprocessors able to perform DSP operations. They provide the wanted flexibility at the cost of potentially higher resource usage and clock rates in order to meet the end-application performance.

In this context, starting from an open-source RISC-V based core by the OpenHW group, a DSP hardware accelerator for NFC decoding has been designed and tested. The work has been carried out in parallel with a colleague who focused on the firmware/algorithmic part to be run on the developed hardware. From the interaction, the two parts have been improved and tailored during the months.

The accelerator implements an architecture for the efficient execution of digital filtering, which turned out to be the main limit within the considered application. The keyword is *flexibility*, giving the user deep customization possibilities, without impacting significantly on the efficiency. The accelerator is implemented in a coprocessor that is coupled to the main core through a standardized interface natively provided by the chosen OpenHW core. This approach allows an easier Instruction Set Architecture (ISA) extension.

This work represents the starting point for a more ambitious project that will be carried out in the next years. The outcome is promising, resulting in performance close to the target and reasonable area-power requirements.
However, assumptions and limitations have been considered in this stage, therefore a lot of investigations and work are needed to get a working system eventually.

# Contents

# Introduction

**Near Field Communication (NFC)** is a ubiquitous technology with many applications ranging from identification to ticketing, from mobile payment to logistical solutions and, just recently, wireless charging. A considerable amount of analog and digital signal processing is required on all NFC integrated circuits to achieve high performance and reliable communication while keeping engineering and manufacturing costs low.
Every modern NFC controllers product must support multiple communication standards in different operative configurations and environmental conditions. In many cases, this is achieved by employing extensive use of **custom Digital Signal Processing (DSP) logic**, which is tuned and constrained to meet the final product requirements, at the expense of full functional flexibility. In such a context, the estimated engineering effort, burdened by a very high pre-silicon verification effort, can be **unacceptable from the business perspective** and limit the end product evolution.

One way to balance chip resources with the growing need for flexibility is to consider the adoption of **microprocessors** able to perform DSP operations. As opposed to more specialized signal processing made with custom digital logic, microprocessors provide the wanted **flexibility** at the cost of potentially higher resource usage: above all, program and data memory. Additionally, higher clock rates might be required to meet the end-application performance, which impacts the total operative power consumption.
Performance could be not suitable for time-critical applications such as the NFC decoding, where the system shall respect very strict time requirements. In these cases, **accelerators** to support microprocessor operations could be a solution. In addition, accelerators guarantee a very short execution time, lowering energy power consumption. As a drawback, the power absorption peaks could be much higher and careful design has to be considered. In modern SoCs, the trend of developing accelerators is growing and chip photos from different electronic companies show that in some cases more than half of the die area is used by blocks other than the CPUs [1].

The hardware platform identified as suitable for the described application is an **open-source RISC-V based core by OpenHW group**. After a preliminary analysis,

the core alone proved not suitable to meet the application requirements. To improve the performance, a DSP hardware **accelerator** for **NFC decoding** has been developed and tested to face the main bottleneck identified, namely **digital filtering**. The work has been carried out in parallel with a colleague who focused on the firmware/algorithmic part to be run on the developed hardware. From the interaction, the two parts have been improved and tailored during the months.

The accelerator **extends** the Instruction Set Architecture (ISA) of the main core with custom instructions. Its integration has been simplified by a standardized interface developed by the OpenHW group, called *eXtension interface*, that allows the implementation of custom instructions without acting on the internal architecture of the core. In other words, it allows the development of an independent accelerator, coupling it to the main core through the interface. This greatly reduces the design and verification effort.
During the development, **flexibility** has been given to the system by defining pre-synthesis parameters (e.g. bitwidth, filter length, desired latency) and runtime configuration opportunities. From a functional perspective, the accelerator allows efficient execution of digital filtering, both FIR and IIR structures. Two intrinsic issues have been identified: **arithmetic operations** to compute the result and **memory access** to fetch the samples.
To face the arithmetic aspects, a Multiply-Accumulate (MAC) arithmetic unit has been implemented. The developed architecture allows the exploitation of Single-Instruction Multiple-Data (SIMD) parallelization to further increase the performance. As for memory access, a buffering mechanism has been introduced to reduce memory load operations. It is based on a series of shift registers organized in banks, in which each bank maintains the status of a specific FIR or IIR filter defined by the user.
The accelerator supports 2's complement **integer** and **fixed-point numbers**. A configurable **rounding unit** is also present to manage fixed-point notation. The filtering samples and coefficients can be taken from the core Register File (RF) or the data memory.

The **testing-verification** of the developed hardware has been considered a priority. A **reference model** to resemble the hardware expected behaviour has been developed using C programming language together with a **random instruction generator**. During the verification flow, random instructions are generated and the expected output from the reference model is compared with the actual output from the RTL simulation. The instruction generation and the reference model execution are performed outside the UVM environment, while the comparison is done exploiting UVM capabilities. To make the verification process more manageable, some custom features have been added to the pre-existing environment by the OpenHW group.
In addition, a software-dependent verification has been performed by the firmware colleague, comparing the output of the DSP algorithm with a MATLAB reference model. This ensured that the software was correct and, to some extent, it added further validity to the hardware testing.

The **results** after the software execution and testing are encouraging. Exploiting the accelerator, the number of instructions per sample has passed from the initial value of 152 to 41. The estimations show that the performance is **close to real-time processing**.
As for the hardware profiling, the **area** occupied by the core-only is 26 kGates. The total estimated area (core+accelerator) spans from 62 kGates to 140 kGates according to the chosen settings for the accelerator. The **power** has been estimated only normalized to the power of the core: the power of the entire system spans from $(2.5 \cdot P_{core})$ to $(3.5 \cdot P_{core})$.

This work represents the starting point for a more ambitious project that will be carried out in the next years by NXP. The outcome at this stage is promising, but a deeper investigation is needed to reach the final aim to integrate the microprocessor into the existing NXP implementation.

This thesis is divided into 6 chapters, plus an appendix:

- ***chapter 1 - RFID background*:** some information about the Radio Frequency IDentification (RFID) systems is summarized. It provides background information on the workings of RFID systems at the physical and data link level, aiding the reader to contextualize the presented work in the following chapters.
  However, the reader who is only interested in accelerator development can safely skip it.

- ***chapter 2 - OpenHW group's cores*:** available open-source RISC-V based cores developed by OpenHW group are presented, with all their features and related tools.

- ***chapter 3 - Preliminary analysis*:** an analysis about the improvement room is performed. It starts from the performance obtained by running the algorithm on the processor with the base RISC-V ISA. Then, the chapter moves to a literature review to identify different alternatives to improve the performance and meet the requirements.

- ***chapter 4 - Accelerator design*:** accelerator design process is presented. This part is a sort of documentation for the accelerator usage and it lists all the custom instructions implemented, their structure and working principle, giving the programmer a clear idea of how to use the accelerator, without going too deep into its architecture (see *Appendix B* for architectural details).

- ***chapter 5 - Accelerator testing-verification*:** here it is described how the existing verification environment has been extended for the verification-testing of the accelerator. The general approach and some details on the technical realization of the different components are shown.

- ***chapter 6 - Results*:** obtained results, both on the software and hardware

sides, are reported and commented on.

- **Appendices:**
  - ***Appendix A - RFID Physical Principles*:** physical principles of inductively remote-coupled systems are explained.
  - ***Appendix B - Architectural details*:** all the MAC design details and choices are reported. Notice that this part is **not** self-contained, but it is the prosecution of ***Accelerator design***.

# Chapter 1

# RFID background

All the information reported in this chapter is taken from [2].

In recent years, automatic identification systems (*Auto-ID*) have become widespread; they were developed to provide information about goods, people, animals.
In these systems, a *transponder* and a *reader* are present: the former is the data-carrying device whose identity has to be checked, while the latter is in charge of checking that identity.

The pioneer was the *barcode label*, representing an omnipresent element in automatic identification. Unfortunately, it is being found to be not sufficient for some applications, due to some intrinsic issues (e.g. low storage capabilities, absence of re-programmability features).

A good solution found was to store the data needed for the identification in a piece of silicon, able to store a large amount of data in a negligible physical space. The first applications of this concept (some of them still used today) need a mechanical coupling between the reader and the transponder, which is not possible in all the use cases and brings issues related to reliability.

For the reasons explained above, contactless approaches have been developed and they are incredibly popular nowadays. Since in these systems the data transfer occurs by exploiting electromagnetic fields, they are called *Radio Frequency IDentification (RFID) systems*.
Some examples are the contactless cards used in a lot of fields from financial (e.g. credit cards) to industrial (e.g. employers' badges), anti-theft tags for goods, NFC implemented in smartphones and many many others.

NFC is a RFID system allowing communication between two devices, usually smartphones and tablets. It allows for having a reliable and flexible protocol, suitable for various applications such as electronic payments, access authentication and easy de-

vice pairing.
Our attention will be focused on NFC systems, but most of the concepts reported can be applied also to other RFID systems. According to the ISO/OSI model, we will focus on the physical and data link layers. The latter, in particular, is the aspect in which we are more interested since the digital logic that will be developed in the remaining chapters of this thesis will work on that layer.

## 1.1 Differentiation

Countless variants of RFID systems exist able to fulfil a wide variety of applications. The main differentiating features are described in the following lines.

### Operating procedures and data quantity

We mainly distinguish between *full/half-duplex* systems and *sequential* systems.

For the *full/half-duplex* systems, the response of the transponder is broadcasted while the field of the reader is continuously switched on, without interruptions.
If the communication is *Half Duplex (HDX)*, the reader broadcasts an unmodulated field (i.e. purely sinusoidal) when the transponder is responding and a modulated field when the transponder is receiving.
In *Full Duplex (FDX)* communications, the field coming from the reader is always modulated, since the two nodes exchange data at the same time, exploiting frequency multiplexing.
This approach opens up to the possibility of having completely passive transponders, supplied by the field of the reader, which send data through load modulation (see *subsection 1.2.1*). It is not relevant if the field is always modulated (full-duplex) or not (half-duplex), since the transponder can draw energy anyway from that field. On the other hand, the fact that the field of the reader is always switched on during the communication requires appropriate design and procedures for the analog front-end (e.g. subcarrier modulation, see *subsection 1.3.2*), since the signal of the transponder is typically very weak if compared to the one of the reader.

In contrast, in *Sequential (SEQ)* systems the field of the reader is switched off at regular intervals, during which the transponder can send its data. The main disadvantage consists of the loss of power during the pauses of the reader which makes the implementation of fully passive transponders difficult.

*Figure 1.1* shows a representation of what happens in the different cases.

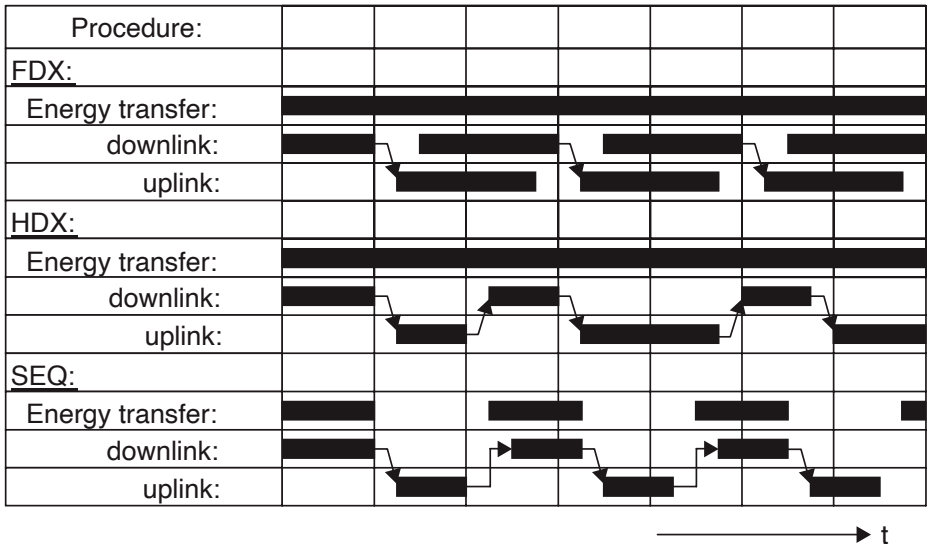



**Figure 1.1:** *Representation of Full Duplex (FDX), Half Duplex (HDX) and Sequential (SEQ) systems over time. Data transfer from the reader to the transponder is termed down-link, while data transfer from the transponder to the reader is termed up-link. Taken from [2].*

The amount of data an RFID system can exchange depends on the specific implementation. It is straightforward that the more the amount of data to be exchanged, the more performance and complexity are required for the hardware.

## Programmability

The possibility of writing into the memory of the transponder and, in general, of easily modifying its behaviour without hardware modifications are desired features in some applications.
Some data must be read-only for security reasons, e.g. a unique serial number for identification, but other ones can be considered useful to be modified during the lifetime of the device. The memory can be managed by simple Finite State Machine (FSM) that are hard-coded into the hardware giving no flexibility after the design, or by microprocessors which make devices more flexible and simple to be updated if necessary. In addition, the same microprocessor can be used for very different tasks changing only the software.
The drawback is that the power needed by a general-purpose architecture could be too high for some applications where passive and low-power transponders are required.

## Power supply

The power supply for the transponder is a very important feature of RFID systems. We can distinguish between *passive* and *active* transponders.

*Passive* transponders do not have their own power supply, but the reader supplies the required energy to the transponder through its field.

*Active* transponders have their own power supply (e.g. a battery) providing energy to the internal chip and to memories for storage retention. The advantage is that the range is extended since these devices can be able to detect weaker signals from the reader as well. It is important to underline that even active transponders are **not** able to generate a field on their own, but the communication principles are the same as the passive transponders.

### Operating frequency, range and coupling methods

The operating frequency is probably the most important criterion to differentiate RFID systems since it widely influences the applications for which that device is suitable. RFID systems are operated at widely differing frequencies, spanning from 135 kHz to 5.8 GHz.

About the achievable range, we can distinguish three main categories, even though the boundaries between them are blurred:

- **close-coupling systems:** today their role is less important on the market, being used only in applications in which strict security requirements are present. They can act in a range up to 1 cm, exploiting *magnetic or electric field* with a frequency from *DC* to 30 MHz, since the communication does not rely on a wave propagation (which is not possible at very low frequencies), but on induction. Examples of applications are *electronic door locking systems* and *contactless smartcards*, even though most of them are operated in remote-coupling today.

- **remote-coupled systems:** they represent the 90% of all the RFID systems sold today and the systems with a range up to 1 m belong to this category. Almost all of them rely on *inductive coupling*, but there are some exceptions relying on *capacitive coupling*. An example of application are *contactless smartcards* we use everyday (ISO/IEC 14443 [3][4][5][6]).

- **long-range systems:** these systems can range up to 15 m, operating at *UHF* and *microwave* exploiting *backscatter* mechanisms and, in some cases, Surface Acoustic Wave (SAW).

## 1.1.1 NFC systems

NFC is a wireless data interface between devices, exactly as *Bluetooth*, but operating with an alternating magnetic field at 13.56 MHz. The typical maximum operating range is 20 cm since the transponder needs to be in the near-field region of the transmitting antenna. An NFC device has both a Transmitter (TX) and an Receiver (RX) chain, sharing a common antenna, that is a large-surface coil or a conductor loop. The NFC interface is active since the energy is supplied by the device containing it.

At first sight, NFC systems could not be strictly considered RFID since the reader/transponder (initiator/target) role is not defined and each device can act either way. However, its characteristics are of interest in relation to RFID systems.

There are two different ways of working for NFC systems:

- **active mode:** the device that wants to start the communication acts as initiator generating an alternating magnetic field (modulated according to the protocol) that is detected and demodulated by the target device. Then, if the target device has to answer, the roles are exchanged and the initiator switches to receiving mode (becoming the new target) and the target switches to transmitting mode (becoming the new initiator). This approach is very different from the one of the RFID systems where the transponder does not generate its own field but acts directly on the one of the reader. Moreover, the roles of initiator/target can not be exchanged in RFID systems.

- **passive mode:** also in this case the communication starts when one of the devices (initiator) starts to generate an alternating magnetic field, but now the field is not turned off and the target can answer to the initiator through load modulation (see *subsection 1.2.1*). This means that the communication is the same as remote-coupled systems relying on magnetic coupling, such as ISO/IEC 14443 [3][4][5][6].
  This opens different useful practical scenarios where the initiator/target roles can be negotiated before the starting of the communication in such a way, for example, that the device with the weakest power supply acts as a target (transponder), exploiting load modulation to transmit the data.
  In addition, NFC devices can interact not only with other NFC interfaces, but they can also interact with passive transponders compliant with the same standard since the physical principle of communication is the same. This way of working is called *reader-emulation mode* since the NFC interface acts as the reader for the transponder in that case.
  Symmetrical, the NFC interface can exploit load modulation to act as a transponder for a reader placed nearby. A practical example is paying contactless with your phone exploiting NFC instead of using directly your physical credit card.

## 1.2 Physical principles

The systems in which we are interested are **half-duplex inductively coupled**, belonging to the category of the **remote-coupled systems**. The fact that the system is half-duplex or full-duplex is not so relevant, since there are differences in the analog front-end and digital controlling logic, but the same working principle applies. The important thing to take into account is that in this kind of system, the field of the reader is always switched on and the transponder can be passive, draining energy from the reader field and sending data through load modulation (see *subsection 1.2.1*).
An inductively coupled system is made by a chip (analog+digital) needed to elaborate

the signal and an antenna consisting of a large-area coil or a conductor loop.

About antenna and electromagnetic field generation by the reader, we can distinguish two main space regions:

- **near-field ($r < \lambda/2\pi$):** in this region the electromagnetic field has not generated a propagating wave, yet, therefore it can be treated as a simple alternating magnetic field coupled with the transponder antenna. Moreover, it can retroact towards the generating antenna, giving the possibility to have a mutual induction between the generating antenna (reader) and the receiving one (transponder);

- **far-field ($r > \lambda/2\pi$):** in this region the field starts to separate from the antenna, becoming a propagating wave. The field can not be seen as a simple magnetic field anymore, but it behaves as a wave, with all the related effects (e.g. reflections). The most important aspect is that this field can not retroact towards the antenna, therefore no inductive coupling can occur between the reader and transponder.

**Inductively coupled** systems work in the *near-field* region, exploiting the inductive coupling. *Long-range systems*, for instance, work at UHF or microwave frequencies, in the *far-field* region, relying on effects such as reflections to communicate.

### 1.2.1 Load modulation

Working in near-field region, inductively coupled systems can exploit the so-called *load modulation* for data exchange. If a transponder is placed in the near-field region of a reader and if the system is correctly designed, the transponder can draw energy from the reader's field. Being in the near-field, magnetic retroaction on the reader antenna is given by the transponder. In other words, according to how the transponder draws energy from the reader field, an equivalent varying impedance $Z_T(t)$ is seen by the reader. This allows having passive transponders that send data varying their energy absorption, without generating their own field.

A more detailed explanation is reported in the appendix, at *section A.3*.

## 1.3 Coding and Modulation

Coding and modulation are the most important part of this background chapter since the digital logic that will be developed in the remaining chapters of this thesis are based on these aspects.

The high-level block diagram of a communication system is shown in *Figure 1.2*.

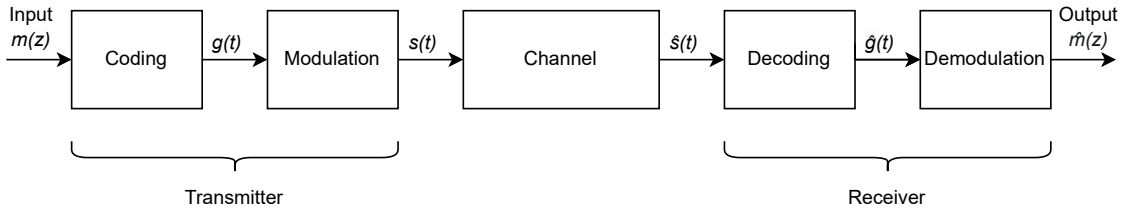



**Figure 1.2:** *High-level block diagram of a communication between two nodes.*

The considered system relies on digital data, therefore all the following considerations are valid for digital transmissions only, while for analog ones things are a bit different. Therefore, the input $m(z)$ is a sequence of bits that has to be delivered to the receiver.

The system consists of five basic blocks:

- **TX - Coding:** the bit sequence $m(z)$ is modified to improve the Bit Error Rate (BER). Then, the bit sequence is translated into a continuous-time signal according to the channel needs. It is important to highlight that the obtained signal $g(t)$ is a base-band signal that is not yet suitable to be sent on the channel.
- **TX - Modulation:** a high-frequency unmodulated signal called *carrier* is altered (amplitude, frequency and/or phase) according to $g(t)$. The resulting signal $s(t)$ is a high-frequency signal whose spectrum is centered around the carrier frequency $f_T$.
- **Channel:** physical medium on which the signal propagates. It introduces noise superimposed to the signal usually modelled as Additive White Gaussian Noise (AWGN) plus additional non-linear effects such as distortion. For these reasons, the output signal is $\hat{s}(t)$ and not $s(t)$.
- **RX - Demodulation:** it implements the symmetric operation with respect to its TX counterpart. The obtained signal is $\hat{g}(t)$ and not $g(t)$ due to the influence of the channel non-idealities.
- **RX - Decoding:** it implements the symmetric operation with respect to its TX counterpart. The bit string is $\hat{m(z)}$ and not directly $m(z)$ because of the channel non-idealities. In other words, the obtained bit string could be different with respect to the transmitted one and (in case) countermeasures are taken according to the protocol policies.

### 1.3.1 Coding

With coding, the ones and zeros are translated into continuous-time signals with different features (e.g. bandwidth) according to the chosen translation scheme. The obtained signal $g(t)$ is a baseband signal.

Mathematically:

$$g(t) = f(m(z)) \tag{1.1}$$

The different translation schemes are called *line coding*. Common line codes for RFID systems are shown in *Figure 1.3*. The ones which we are more interested in are the *Manchester coding* and the *modified Miller coding* that are used in the ISO/IEC 14443 Type-A protocol [4].

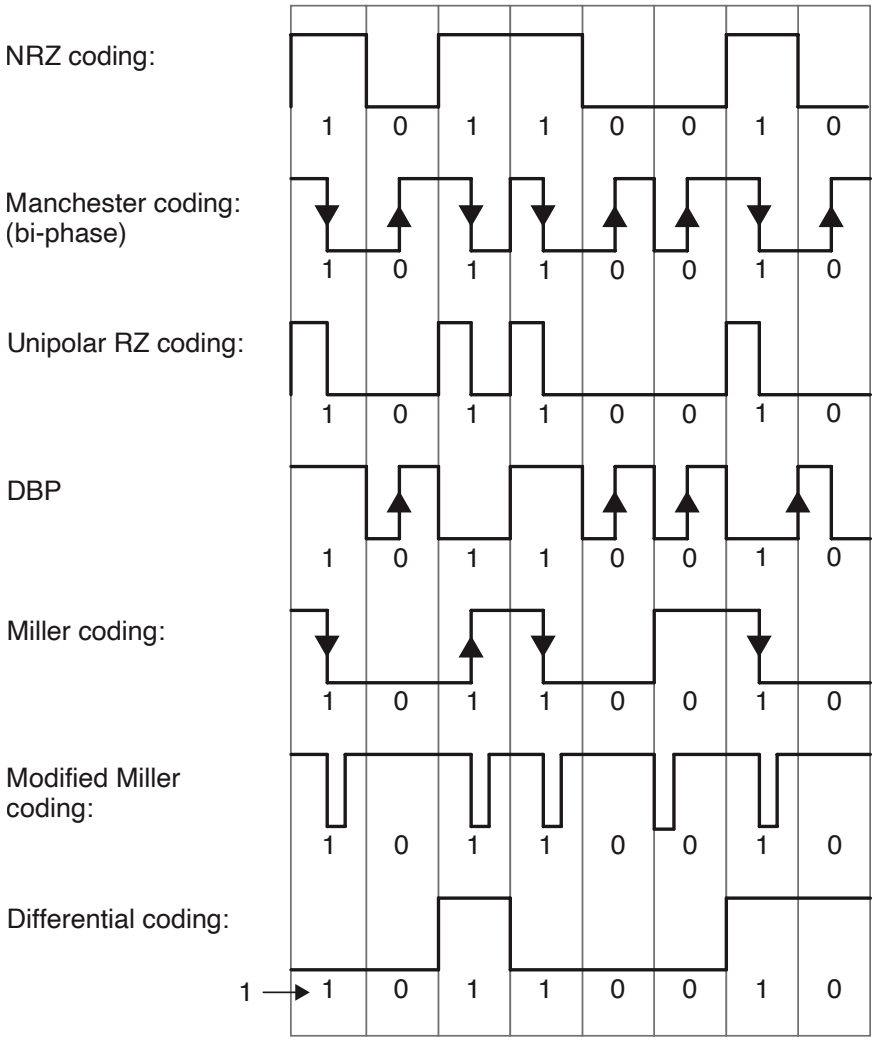



**Figure 1.3:** *Line codings frequently used in RFID systems. Taken from [2].*

### 1.3.2 Modulation

$g(t)$ is a baseband signal that is not suitable for direct transmission on a channel like the magnetic induction considered before. Another point is the fact that the channel could be shared among multiple devices, even not RFID systems, that could operate in the same frequency region, needing a frequency multiplexing to avoid interference.

In the modulation stage, $g(t)$ is used to modify the parameters (amplitude, phase, frequency) of a carrier $c(t)$, which is an unmodulated electromagnetic wave (i.e. a sinusoid). In other words, the information contained in $g(t)$ is embedded into the carrier $c(t)$: the former brings the information while the latter makes it transferable on a physical channel.

There are infinite possibilities for this process, but the simplest one is represented by the so-called *binary modulated signalling*. Three modulations belong to this category and they are Amplitude-Shift Keying (ASK), Phase-Shift Keying (PSK) and Frequency-Shift Keying (FSK). According to $g(t)$, one parameter of the carrier is changed (respectively amplitude, phase or frequency).

More complex modulations exist and they are used in communication systems where a high bitrate needs to be achieved (e.g. Quadrature Amplitude Modulation (QAM), Quadrature Phase-Shift Keying (QPSK)). However, they are not used for RFID, since in these systems the required performance is very limited and maintaining the complexity low is preferred.
For this reason, the equations shown below are not general and they cannot be applied to more complex modulation schemes.

The modulation used in ISO/IEC 14443 Type-A protocol is the *ASK* [4], that is the purely digital version of Amplitude Modulation (AM). The amplitude of the carrier is switched between two states $u_0$ and $u_1$ according to $g(t)$. Assuming a sinusoid with amplitude 1 as carrier, $u_0$ and $u_1$ are respectively the upper and lower values that $s(t)$ assume after the modulation.
Mathematically:

$$s(t) = [d \cdot g(t) + 1 - d]\, c(t) \qquad with \qquad d = \frac{u_0 - u_1}{u_0 + u_1} \tag{1.2}$$
$$c(t) = \sin(\omega_T t)$$

where $d$ is called *duty factor*.

An example of ASK signalling is shown in *Figure 1.4*.

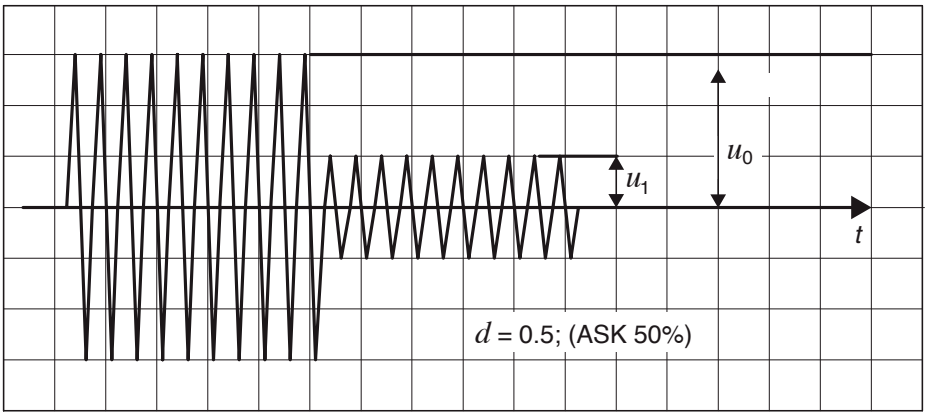



**Figure 1.4:** *ASK working principle. Taken from [2].*

Working into the frequency domain, the operation done with ASK is:

$$s(t) = (d \cdot g(t) + 1 - d)\, c(t) = x(t)\, c(t) \quad \Rightarrow \quad S(\omega) = X(\omega \pm \omega_T) \tag{1.3}$$

that returns the $x(t)$ spectrum translated and centered around $\omega_T$.

At the receiver side, the signal can be translated to baseband through dedicated analog devices called *mixers* and then the information can be recovered.

However, transponders in inductively coupled RFID systems often rely on two subsequent frequency translations, with a subcarrier and a carrier respectively. The reason is that, at the reader side, the signal coming from the transponder (load modulation, see *subsection 1.2.1*) has a magnitude that is much lower than the signal from the reader itself, making it difficult to correctly detect it. This issue is mitigated by using a subcarrier, since in that case the signal coming from the transponder can be easily filtered out, being the reader/transponder spectra not superimposed.

This technique can be mathematically expressed as:

$$\begin{aligned} v(t) &= x(t)\, \cos(\omega_S t) = x(t)\, \cos(2\pi f_S t) \\ s(t) &= v(t)\, \cos(\omega_T t) = v(t)\, \cos(2\pi f_T t) \end{aligned} \tag{1.4}$$

where $f_S$ is the frequency of the subcarrier and $f_T$ is the frequency of the carrier.

A representation of subcarrier modulation is shown in *Figure 1.5*.

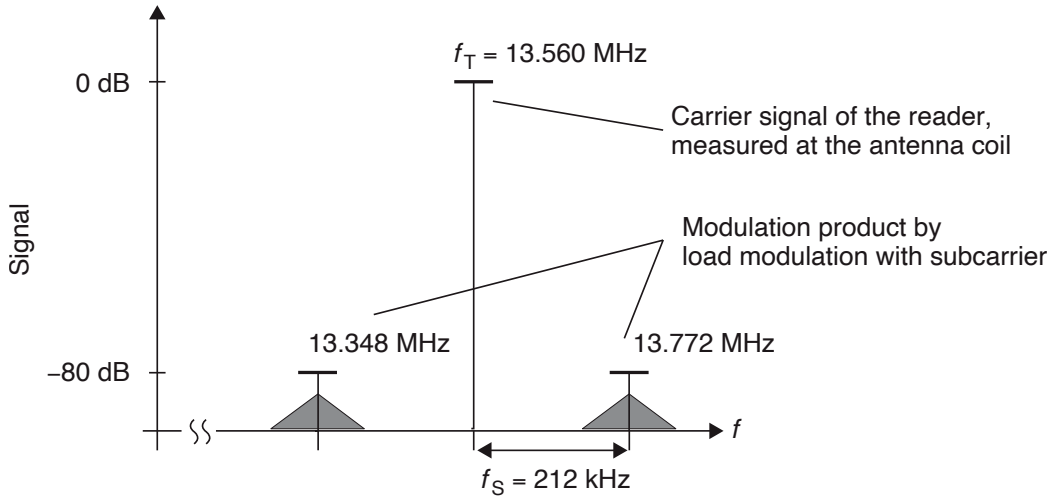



**Figure 1.5:** *Subcarrier+carrier modulation. Taken from [2].*

The subcarrier modulation approach can be used by transponders in the case of inductively coupled RFID systems that are based on the physical principles explained in *Appendix A*. Those communications are half-duplex, therefore only one of the two nodes transmits at a time.

For the reader→transponder communication, the former does not need a subcarrier because of the sufficient power available. Therefore, it transmits on a single carrier (as stated in *Equation 1.3*) using an active approach (analog front-end terminates with a power amplifier), providing power supply and data to the transponder.

About the transponder→reader communication, the transponder is usually passive and it relies on load modulation for the transmission. The variations given by the load modulation are usually very weak, needing subcarrier modulation. Referring to *Figure A.5* and *Figure A.4* and according to the *Equation A.8*, the change in the impedance $Z_T$ due to transponder load modulation can be detected at the reader side measuring the voltage $u_{meas}$. As said, a subcarrier is present and this means that the FET is not switched on and off by a baseband-coded signal, but a low-frequency subcarrier is first modulated by the baseband-coded signal. The modulated subcarrier is then used to drive the FET.
A principle scheme is shown in *Figure 1.6*.

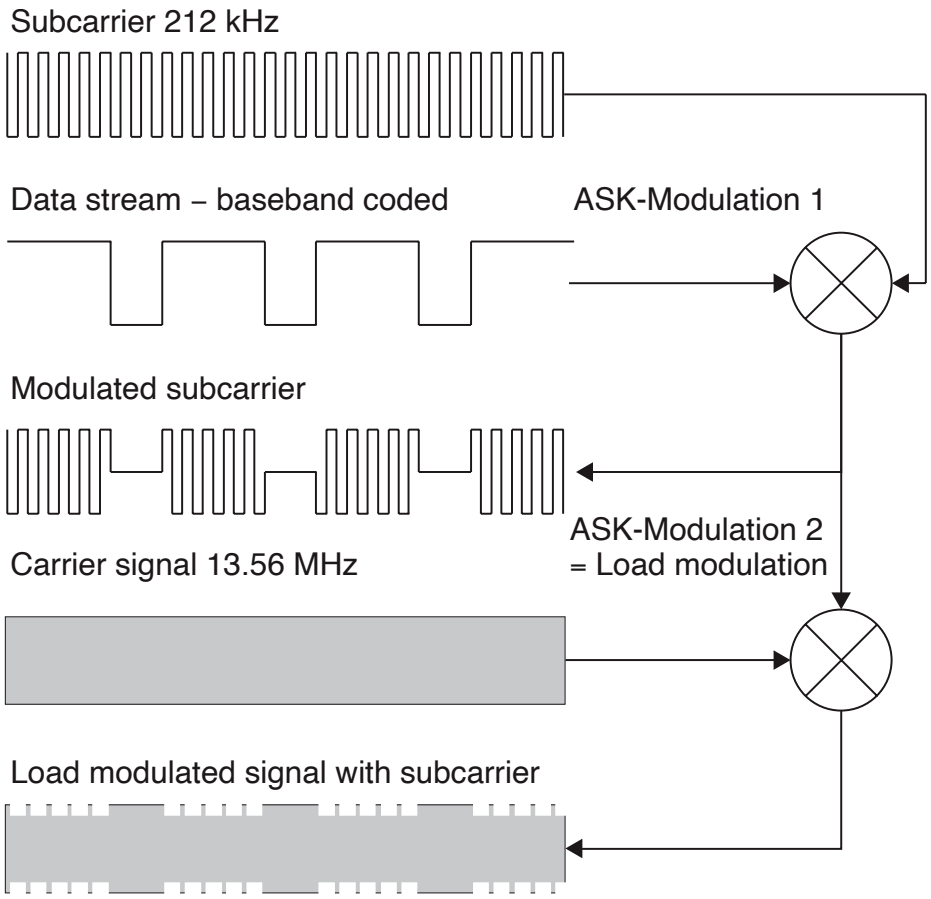



**Figure 1.6:** *Principle scheme of carrier+subcarrier generation. Taken from [2].*

## 1.4 ISO/IEC 14443

The RFID systems in which we are interested rely on the ISO/IEC 14443 protocol, entitled *"Cards and security devices for personal identification — Contactless proximity objects"*, used for contactless proximity-coupling smart cards [3][4][5][6].

Considering the differentiation made in *section 1.1*, the systems compliant to that protocol are half-duplex and the transponders are passive (if they are NFC systems, they operate in passive mode).
They are inductively coupled systems with an operating frequency of 13.56 MHz and a range of 7 to 15 cm.

Two different variants have been defined, called respectively *Type-A* and *Type-B*. The working principle and the physical layer are the same and all the differences are at higher levels (e.g. coding, error detection), allowing the same device to work in both modes changing a few parameters of the control part.
The variant used in the application that we will discuss in the rest of this thesis is the *Type-A*.

In Type-A, for the communication reader→transponder, ASK modulation with $d = 1$ (100%) and modified Miller coding is used.

For the transponder→reader, a load modulation procedure with a subcarrier at frequency $f_S$ =847 kHz (13.56 MHz/16) is used. The modulation of the subcarrier is made through ASK modulation with Manchester coding. The admitted baud rates for the transmission are various, but the considered application has $f_{baud}$ =106 kHz. Here, the ASK duty factor $d$ depends on the coupling coefficient $k$ (see *section A.1*) between the reader and the transponder since the load modulation effect is more effective if $k$ is high (refer to *Equation A.8* for a mathematical expression).
In *Figure 1.7* an example of the reader's and transponder's waveforms is reported.

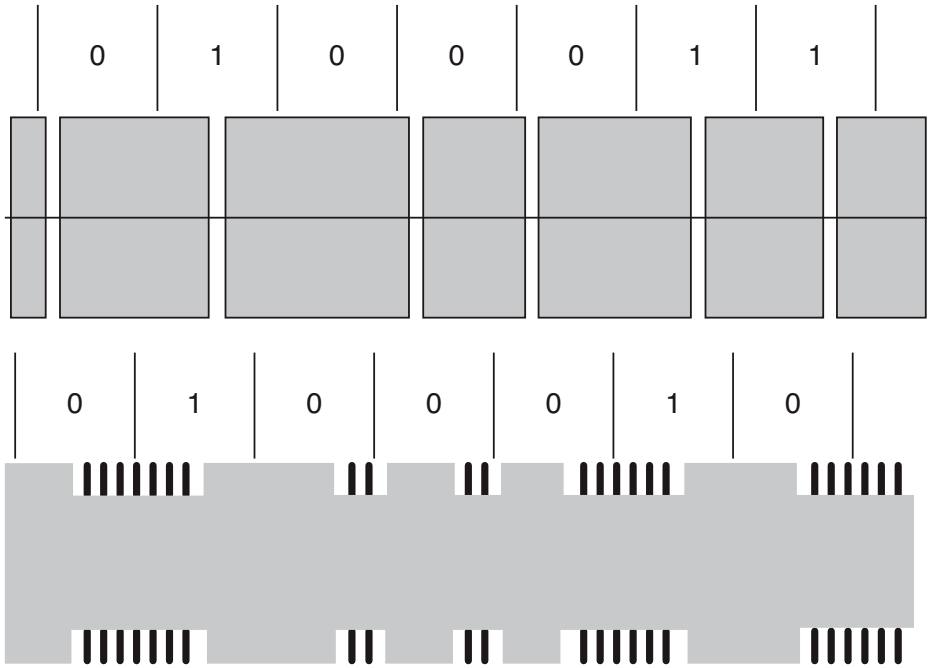



**Figure 1.7:** *Example of ISO/IEC 14443 Type-A waveforms.*
*First image: down-link (voltage at the reader antenna).*
*Second image: up-link (voltage at the transponder coil). Taken from [2].*

The protocol is very detailed on the aspects discussed so far and it goes on with the upper layers (e.g. initialization and anticollision, transmission protocol). Within this thesis, a demodulator for the transponder→reader communication will be considered, therefore it would be not useful to go on with further details and the protocol deepening is left to who is reading, if interested.

# Chapter 2

# OpenHW group's cores

## 2.1 Why RISC-V?

In the field of microprocessors, one of the main distinction parameters is the Instruction Set Architecture (ISA). The ISA defines aspects such as the supported data types, which instructions the processor is able to execute, the registers and how the hardware manages main memory. The ISA can be extended by adding instructions or other features.
Two main design philosophies about ISA exist: RISC and CISC.

According to Reduced Instruction Set Computer (RISC), the CPUs are designed to execute really basic instructions, relying on simple and efficient hardware architectures. This means that to execute a certain program a large number of assembly instructions could be needed; this aspect is counterbalanced by the efficiency (in terms of clock frequencies and power) given by the hardware architecture.

On the other hand, the Complex Instruction Set Computer (CISC) approach is based on more expensive architectures able to execute more complex instructions. As consequence, the number of assembly instructions needed to execute the same program is less than RISC, but the architecture is less efficient and a lower clock frequency and higher power is reached.

In [7], an analysis of the RISC vs CISC debate is reported. Traditionally, RISC ISA was used for low-power applications (e.g. mobile) while CISC for high-performance (e.g. server). Nowadays, the computing landscape is different, since the growth of mobile devices running RISC ISA is surpassing that of the desktops and servers running CISC ISA. Moreover, high-performance RISC and low-power CISC applications are arising and in some cases CISC processors embed some RISC parts and vice-versa, making the distinction quite blurred. From the analysis of various processors emerged that RISC and CISC are two engineering design points and there is nothing fundamentally more efficient in one ISA class or the other.

However, that analysis focuses on general-purpose microprocessors, ARM Cortex-A8/Cortex-A9 and Intel Atom/Sandybridge, that are not suitable for the application considered in this thesis.

In [8], a core for IoT applications with DSP ISA extensions is presented. That application is more similar to the one this thesis is focused on and it is based on the RISC-V platform.

RISC-V is an open standard RISC ISA that has been developed at the University of California, Berkeley, since 1981. It is an open-source, royalty-free ISA that can be freely used by anyone as a base block to build their own solutions and services, even if commercial [9].

The main reason behind the RISC-V choice for the application presented in [8] is the open-source nature of the project that allows avoiding the dependency on IP provider, cutting the cost and giving freedom for application-specific instruction extensions.

In addition to this, RISC-V project is 30 years old and it has been widely explored; this means that it is likely to be mature now. Other symptoms of its maturity are the variety of RISC-V based implementations that are arising in the last years and the great attention that the companies are dedicating to it. In fact, leading companies in the semiconductor field, such as NXP, are investing in RISC-V.

## 2.2 OpenHW group

The development of the project has been carried out starting from open source RISC-V based cores by the OpenHW group. As one can read from the OpenHW group website [10]:

> *OpenHW Group is a not-for-profit, global organization driven by its members and individual contributors where hardware and software designers collaborate in the development of open-source cores, related IP, tools and software. OpenHW provides an infrastructure for hosting high-quality open-source HW developments in line with industry best practices.*

They provide a family of permissively licensed RISC-V based cores called *Core-V*, with different features and intended applications.

OpenHW group's cores are becoming popular for different reasons. They offer a great variety of implementations suitable for different applications and the quality is comparable to industrial products. In addition, it is not so difficult to switch from one core implementation to another, since they share the same development approach and verification environment.

These advantages add up to the ones of RISC-V, determining OpenHW group's core as a very good choice for the considered DSP application.

In *Figure 2.1* the road map for the different cores is shown. It can be seen that some cores are in advanced status, while others are in development or just a concept.

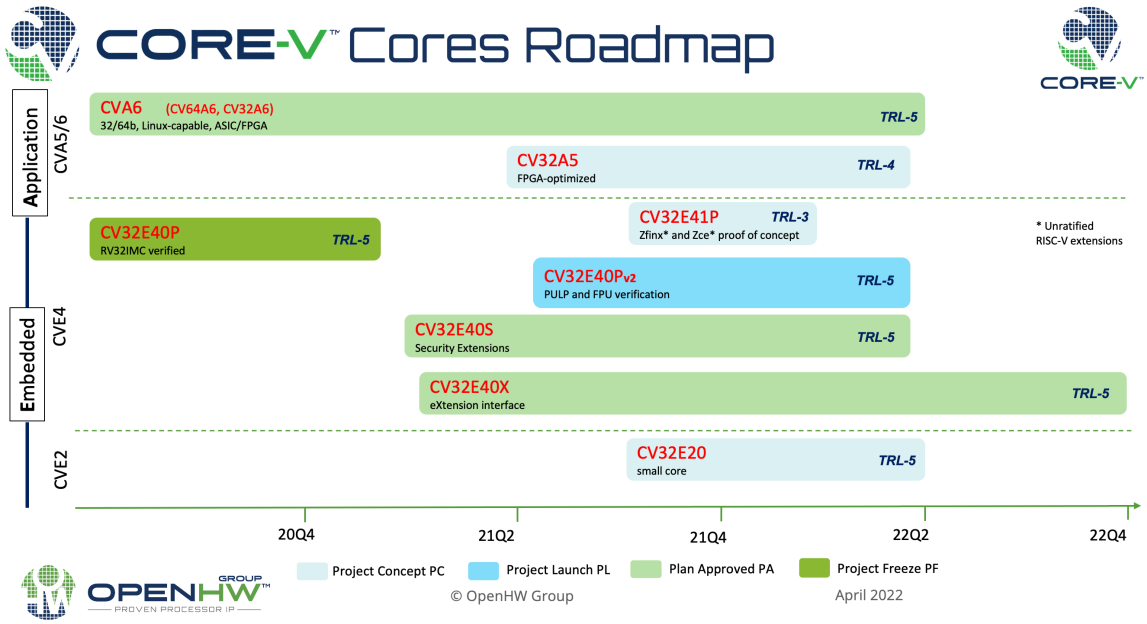



**Figure 2.1:** *Core-V family cores road-map updated to April 2022. Taken from OpenHW repo [11].*

The following brief descriptions are taken from [11].

## Application cores

- **CVA6**: CVA6 is a 32-bit/64-bit 6-stage, in order, single-issue core, implementing `RV32GC` or `RV64GC` extensions with three privilege levels. In addition, it has features to support Unix-like operating systems.

- **CVA5**: CVA5 is a 32-bit RISC-V processor designed to support FPGAs. Multiply/Divide and Atomic extensions (`RV32IMA`) are present. The datapath has been developed to support parallel, variable-latency execution units and to be easily extended with custom ones.

## Embedded cores

- **CV32E40P**: CV32E40P is a 32bit, 4-stage core that implements `RV32I[M][F]C[Xpulp]` ISA. Among the optional extensions, there are the FPU, multiplication/division support. In addition, custom instructions are implemented for DSP operations.

- **CV32E40X**: CV32E40X is a 32-bit small and efficient, in-order RISC-V core with a 4-stage pipeline that implements the `RV32[I,E][M|Zmmul][A][Zca_Zcb_Zcmb_Zcmp_Zcmt][Zba_Zbb_Zbs|Zba_Zbb_Zbc_Zbs][Zicntr][Zihpm][Zicsr][Zifencei][X]` ISA. It is aimed at compute-intensive applications and it provides an extension interface to reduce the effort for custom instructions implementation.

- **CV32E40S**: CV32E40S is a 32-bit small and efficient, in-order RISC-V core with a 4-stage pipeline that implements the `RV32[I,E][M|Zmmul][Zca_Zcb_Zcmb_Zcmp_Zcmt][Zba_Zbb_Zbs|Zba_Zbb_Zbc_Zbs][Zicsr][Zifencei][Xsecure]` ISA. It is aimed at security applications.

- **CV32E41P**: CV32E41P is a 32-bit small and efficient, in-order RISC-V core with a 4-stage pipeline. It is a fork of the CV32E40P core, supporting the same ISA plus the official RISC-V `Zfinx` and `Zce` extensions.

### The considered alternatives: CV32E40P and CV32E40X

Among the cores described above, the ones interesting for the considered DSP application are the *embedded cores* since the complexity of an *application core* is not needed. Among the four available, the CV32E40S has security features that are not needed, while the CV32E41P development status is far to be usable. The CV32E40P and the CV32E40X are good candidates because their development is advanced and they have unique useful features that will be discussed in the following sections.

## 2.3 CV32E40P main features

It is a 32-bit 4-stage core compliant with the RISC-V RV32I base integer ISA and with other RISC-V standard and custom extensions. The block diagram is shown in *Figure 2.2*. We are not interested in focusing on the different blocks working principles (for more information, refer to [12]).

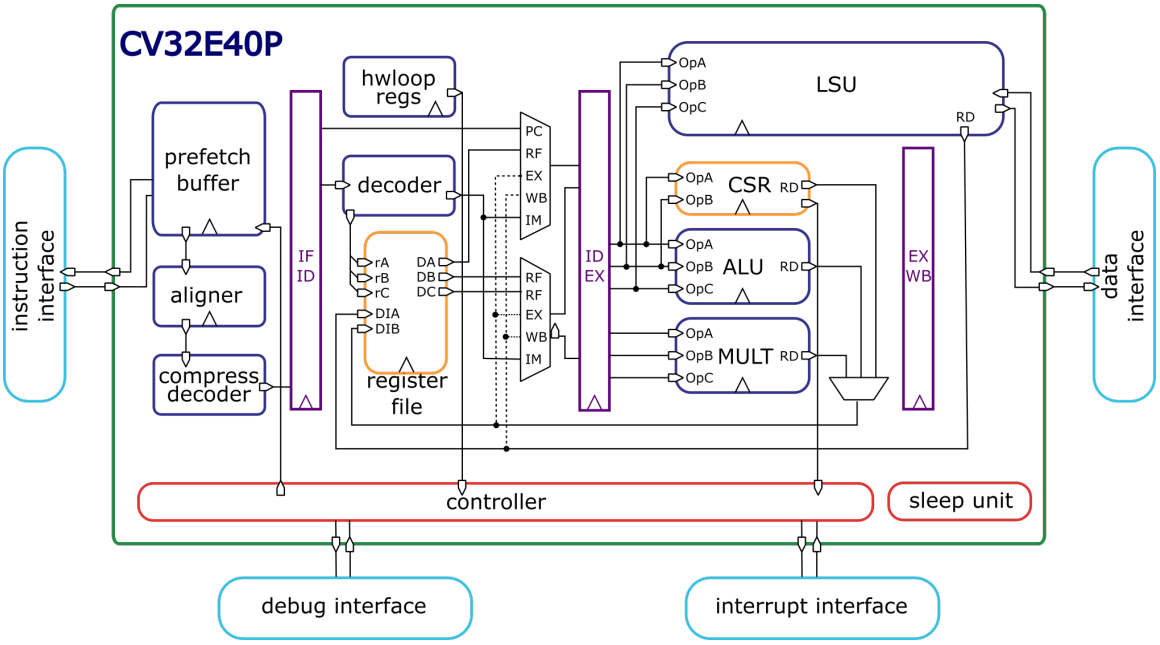



**Figure 2.2:** *CV32E40P block diagram [12].*

The RISC-V **standard ratified** extensions available are:

- **I (v2.1, always enabled):** basic instruction set;
- **C (v2.0, always enabled):** compressed instructions;
- **M (v2.0, always enabled):** integer multiplication and division;
- **Zicntr (v2.0, always enabled):** performance counters;
- **Zicsr (v2.0, always enabled):** Control and Status Registers related instructions;
- **Zifencei (v2.0, always enabled):** instruction-fetch fence;
- **F (v2.2, optionally enabled):** single-precision floating-point arithmetic operations using dedicated F-registers;
- **Zfinx (v1.0, optionally enabled):** single-precision floating-point arithmetic operations using integer X-registers.

The RISC-V **standard not ratified** extensions available are:

- **Zicntr (v2.0, always enabled):** performance counters.

**Custom** instructions available are:

- Post-Incrementing load and store;
- Hardware Loop extensions;
- ALU extensions;
- Multiply-Accumulate extensions;
- Single-Instruction Multiple-Data (SIMD) extensions.

## 2.4 CV32E40X main features

It is a 32-bit 4-stage core, compliant with the RISC-V RV32I base integer ISA and with other RISC-V standard extensions. The CV32E40X block diagram is shown in *Figure 2.3*. We are not interested in focusing on the different blocks working principles (for more information, refer to [13]).

It is similar to the CV32E40P since it is derived from it, but it has different ISA extensions implemented and a standard interface called *eXtension interface* is present to simplify the custom ISA extension through a coprocessor (look at *subsection 2.4.1*).

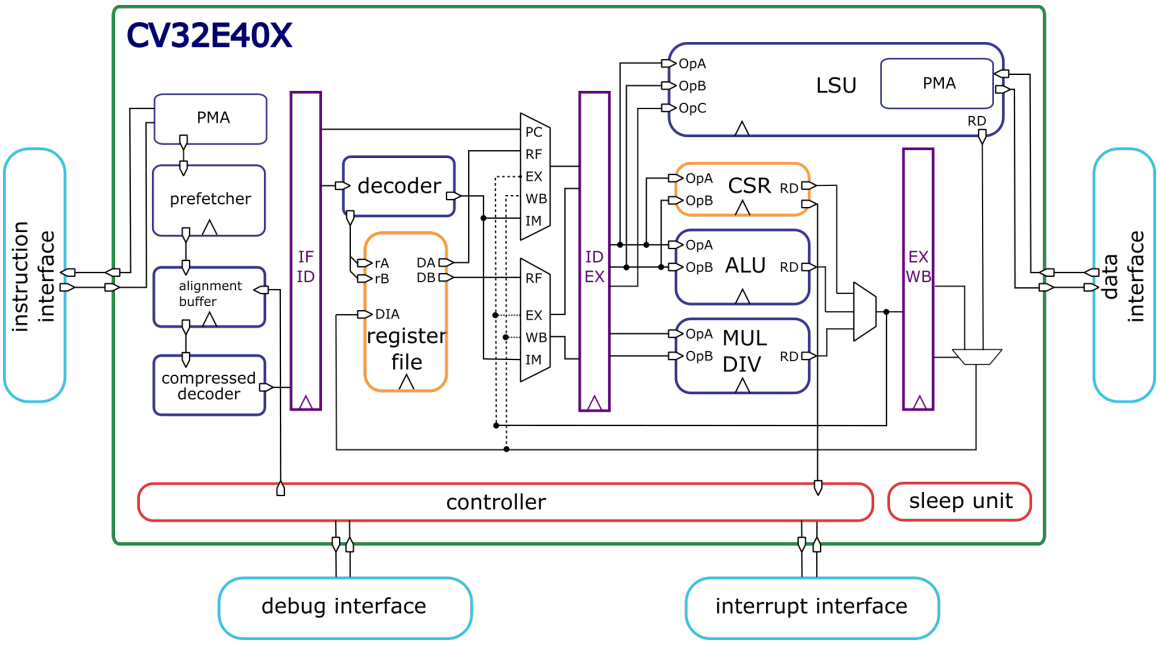



**Figure 2.3:** *CV32E40X block diagram [13].*

**Notice that the specifications listed below are updated to the period in which this thesis has been written (Q3'22) and, for now, the CV32E40X core is not RTL freeze. This means that these instructions are the final aim of the OpenHW project, but for now, some of them could be not**

**yet implemented or their implementation modified in the future. In addition, some instructions are not even ratified by the RISC-V international association, therefore they are more likely to be changed.**

The RISC-V **standard ratified** extensions available are:

- **I (v2.1, always enabled):** basic instruction set;
- **C (v2.0, always enabled):** compressed instructions;
- **M (v2.0, optionally enabled):** integer multiplication and division;
- **Zicsr (v2.0, always enabled):** Control and Status Registers related instructions;
- **Zifencei (v2.0, always enabled):** instruction-fetch fence;
- **A (v2.1, optionally enabled):** atomic instructions;
- **Zba / Zbb / Zbc / Zbs (v1.0.0, optionally enabled):** bit manipulation;
- **Zkt (v1.0.0, always enabled):** cryptography-related extensions;
- **Zbkc (v1.0.0, optionally enabled):** cryptography-related extensions;
- **Zmmul (v1.0.0, optionally enabled):** subset of the standard *M* extension.

The RISC-V **standard not ratified** extensions available are:

- **Zicntr / Zihpm (v2.0, always enabled):** counters;
- **Zc* (v0.70.4, always enabled):** subset of the standard *C* extension.

### 2.4.1 eXtension Interface (CV32E40X only)

The CV32E40X core implements and supports a specific interface, called *eXtension interface*, to simplify the implementation of custom instructions. It is not supported by the CV32E40P core.

The related documentation updated to v0.2.0, which is the one used in this project, can be downloaded from [14].

The eXtension InterFace (XIF) is an interface developed by the OpenHW group to have a standardized manner to extend the core with custom instructions, without the need to modify the core internal RTL. It can be used to implement standard RISC-V extensions (floating-point support, SIMD, bit manipulation etc...) or fully custom extensions.
Custom ALU type, load/store type or CSR related extensions can be implemented.

Custom instructions related to instruction flow control (e.g. branches, jumps) are **not** supported through the interface.

The basic idea is to have an external *coprocessor* to be coupled to the main core through the XIF. The coupling is loose since the coprocessor cannot directly access the core datapath. The coprocessor shall implement the desired custom instructions and the logic to manage the communication core-coprocessor. The interface is based on handshaking (control) and data signals divided among six interfaces (*compressed*, *issue*, *commit*, *memory*, *memory result*, *result*), according to the purpose of each signal.

A bunch of parameters can be set to use the interface with cores with different features (e.g. memory width addressing, number of reading ports of the core register file). Almost all of these parameters are core dependent, therefore they are fixed for the CV32E40X and not significant for the coprocessor development.

### Working principle

CPU will attempt to offload every (compressed or non-compressed) instruction that it does not recognize as a legal instruction itself. In the case of a compressed instruction, the coprocessor must first provide the core with a matching uncompressed (i.e. 32-bit) instruction using the *compressed interface*. This non-compressed instruction is then attempted for offload via the *issue interface*.

The offloading of all the uncompressed instructions is performed through the *issue interface*. Together with the instruction, the core provides the required Register File (RF) operands (i.e. `rs1` and `rs2`), which are always encoded in the instruction bits `[19:15]` and `[24:20]` respectively. The coprocessor shall signal for each offloaded instruction if that instruction is accepted or rejected and it shall inform the core about the type of instruction (e.g. load/store, writeback to RF is needed).

The core can speculatively offload instructions to the coprocessor. This means that the core can offload an instruction without knowing if it shall be committed or killed (e.g. due to a taken branch). The communication about the committing/killing is performed through the *commit interface*. In case of killing, the coprocessor shall ensure that the instruction is completely discarded with no remaining traces.

In the case of load/store instructions, the memory accesses by the coprocessor are made indirectly through the *memory interface* and the *memory result interface*. Through the former, the coprocessor requests memory access (both read/write), while on the latter the core returns the requested value (in case of a load) or a simple acknowledgement (in case of a store).

At the end of the execution of an offloaded accepted instruction, the coprocessor shall signal its completion through the *result interface*. In case a writeback to the core RF is needed, the value to be stored and the register index is sent through this

interface. If a writeback is not needed, a simple acknowledgement by the coprocessor is signalled.

In short: from a functional perspective, it should not matter whether an instruction is handled inside the core or inside a coprocessor. In both cases, the instructions need to obey the same instruction dependency rules, memory consistency rules, load/store address checks, etc.

## 2.5 CV32E40P and CV32E40X comparison

The two cores are pretty similar since the X-version has been derived from the P-version, but there are some differences, as shown in *Table 2.2*.

| | **CV32E40P** | **CV32E40X** |
|---|---|---|
| **ISA** | Useful standard/custom extensions already present (e.g. FPU, SIMD) | Limited extensions present (no FPU, no SIMD) |
| **ISA extension** | Directly in the EX-stage of the core → extension could be tricky | eXtension interface → easily extendable |
| **Dev status** | Frozen development → stable | Advanced but WIP development → bugs can be present |

**Table 2.2:** *Summary of the main differences between CV32E40P and CV32E40X.*

## 2.6 Verification environment

The verification environment is provided by OpenHW and it can be found on their repository [15]. Within this thesis, details will be avoided and only a brief overview will be given. Additional features have been implemented to support this project and they are reported in *chapter 4*.

The environment is based on the Universal Verification Methodology (UVM) framework to guarantee flexibility and compliance with respect to industrial working practices. It is supporting all the OpenHW group's cores, with some elements that are common to all the cores (e.g. UVM libraries) and others that are core-dependent (e.g. assertion modules). Different simulation tools (e.g. Cadence Xcelium, Metrics Dsim) are supported.
Among the core-dependent components, there are the testcases: many of them have been defined by the OpenHW group for each core to verify their features. To start using the environment, it is sufficient to download a GCC toolchain supporting RISC-V instructions (e.g. RISC-V Embecosm toolchain [16]) and slightly modify the makefiles (look at the OpenHW docs for more information).

An open-source bus called *Open Bus Interface (OBI)* is used within the testbench for core-memory communication. Memory behaviour is emulated through some models.

For the core verification, an Instruction Set Simulator (ISS) is needed. They are reference models emulating the RTL behaviour of the core exploiting high-level programming languages. Even though it can be disabled in the testbench, without a reference model the RTL executes the code without any checking on the correctness of the execution.
There are two popular ISSs:

- **Spike:** it is an open-source ISS for RISC-V CPUs only, supporting the base ISA and a lot of extensions. In addition, it can be easily extended with custom instructions (even though some limitations are present);

- **OVPsim:** it supports a lot of different CPUs models, from different vendors. It was born as fully open-source ISS, but during the years it has become a hybrid: it can be used freely if no commercial aims are present (personal, hobby, academic), while a fee is applied if it is aimed to be used for commercial projects.

At the moment, OVPsim supports the OpenHW group's core and it is integrated into the verification environment, while Spike is not supporting the OpenHW group's core specifically, but only general RISC-V CPUs. For OVPsim, we decided to not use it to avoid possible licensing legal issues. As for Spike, a not negligible effort would be needed to adapt it to support the OpenHW group's cores and to integrate it into the verification environment. For these reasons, we decided to disable the ISS support in the verification environment and to perform a basic verification of the developed accelerator working externally, as discussed in *chapter 4*.

# Chapter 3

# Preliminary analysis

## 3.1 Starting point

### 3.1.1 System overview

An NFC system compliant with the standard ISO/IEC 14443 is considered. For a high-level overview of a communication system, see *Figure 1.2*.

Focusing on the RX chain, a very basic scheme is shown in *Figure 3.1*. It is made by:

- **analog front-end:** it receives the radio frequency field and it performs the carrier demodulation (i.e. the spectrum centered around $\omega_T$ is translated to baseband) and an Analog-to-Digital Converter (ADC) is exploited for sampling. Its output is the I/Q samples to feed the digital part. It works on the *physical layer* of the ISO/OSI model.
- **digital baseband demodulator:** it is in charge of the baseband demodulation starting from the I/Q samples provided by the analog front-end. Its output is a bitstring that should match with the one sent by the TX. It works on the *data link* layer of the ISO/OSI model.
- **additional processing:** it performs additional operations on the bitstring provided by the demodulator, such as collision detection and error correction. It works on the *data link* layer and above.

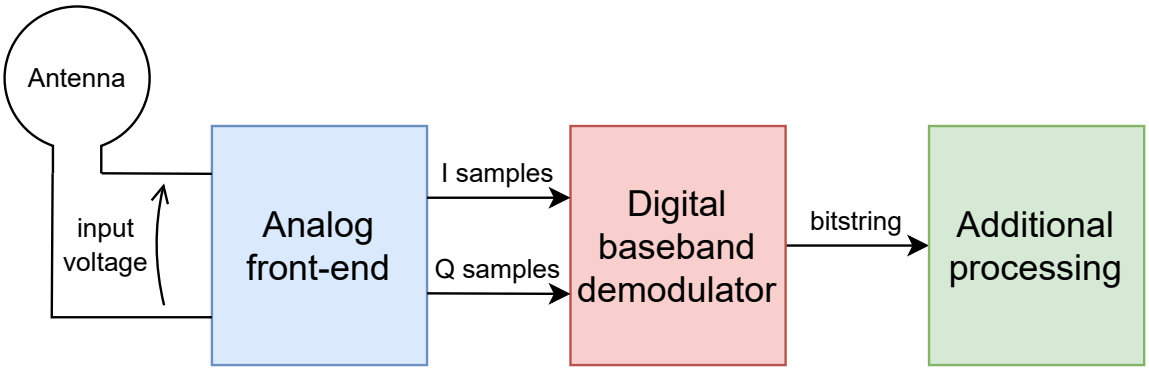



**Figure 3.1:** *Principle scheme of the RX chain.*

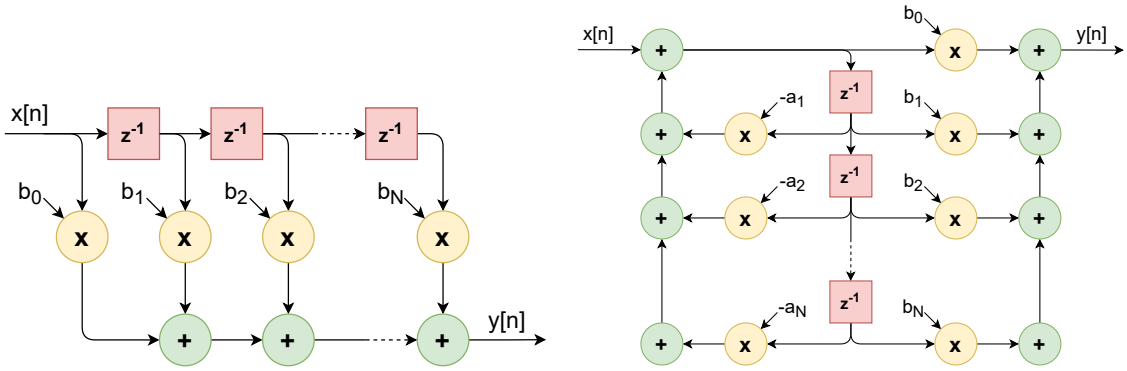
The block this work is focused on is the **digital baseband demodulator**. The ISO/IEC 14443 standard supports many configurations, therefore one of them has been chosen: reader mode, Type-A, 106 kHz baud rate [4].

This setting has been chosen as a starting point because it is the simplest case to be managed, but this is not a limit since the project developed can be extended to different use cases. Indeed, for some use cases, the hardware developed is perfectly compatible and only some software changes are required, while for others some hardware extensions could be needed. The fact that many configurations and use cases exist is the main reason why the hardware needs to be as flexible as possible.

The existing hardware and algorithms are NXP's Intellectual Property (IP), therefore the details can not be explained in this work.
What is relevant for the following discussions is that the system is a chain of various digital filters, each of them executing different tasks and having a different number of taps and a different bitwidth.
In the end, the main operation to be performed is **digital filtering**.

### 3.1.2 Digital filtering basics

Generally speaking, two basic digital filter structures exist: *Finite Impulse Response (FIR)* and *Infinite Impulse Response (IIR)*. The general diagram is shown in *Figure 3.2*, while the corresponding equations are respectively shown in *Equation 3.1* and *Equation 3.2*.




**(a)** *FIR filter.* **(b)** *IIR filter.*

**Figure 3.2:** *General structure of FIR and IIR filters.*

FIR equation:

$$y[n] = \sum_{i=0}^{N-1} b_i \, x[n-i] \tag{3.1}$$

IIR equation:

$$y[n] = \sum_{i=0}^{N-1} b_i \, x[n-i] - \sum_{j=1}^{N-1} a_j \, y[n-j] \tag{3.2}$$

where:

- $x[n-i]$: input at the discrete time instant $i$;
- $y[n-j]$: output at the discrete time instant $j$;
- $(N-1)$: order of the filter;
- $b_i$: feedforward filter coefficients;
- $a_i$: feedback filter coefficients.

The arithmetic operation made by a filter is a Multiply-Accumulate (MAC). There are data coming in, shifting through the delay line, which are multiplied by fixed coefficients and then all the multiplication results added up.

### 3.1.3 Results exploiting RISC-V base ISA

#### 3.1.3.1 General filter expected results

If one wants to implement a filter using a processor:

- the input samples $x[n-i]$ are usually taken from the data memory;
- the coefficients $b_i$ and $a_j$ could be taken from the data memory or provided as immediate packed into the instruction;
- the output samples $y[n-j]$ have to be computed by the ALU and then maintained into the core registers (if feasible) or stored/loaded into/from the data memory.

Consider the FIR case (refer to *Equation 3.1*) and assume that the coefficients are passed as immediate (i.e. no memory load needed for them). In a first approximation, to compute a single value of $y[n-j]$ the instructions needed are:

- `lw`: $N$ times ($N$ input samples $x$ have to be loaded from the data memory);
- `muli`: $N$ times ($N$ input samples $x$ have to be multiplied by $N$ coefficients $a$, $b$);
- `add`: $(N-1)$ times (results of the previous multiplications ($N$ elements) have to be added together).

#### 3.1.3.2 Considered use case results

Focusing on the considered use case, an indicator of the system performance can be obtained by comparing the number of instructions needed per sample with the maximum number of instructions that the processor can execute in a sampling period. The sampling period is the key part of the analysis. To be real-time, the processor must be able to execute all the instructions needed for a sample in a time lower (or equal) than the sampling period.

Assuming:

- a sampling frequency of 13.56 MHz, with a 2x downsampling, the time interval between two subsequent samples is:
$$t_s = \frac{1}{13.56\,\text{MHz}} \cdot 2 = 147.5\,\text{ns} \tag{3.3}$$
- a CPU clock frequency $f_{clock} = 540\,\text{MHz}$, that is a reasonable value for the processors we are considering, the number of clock cycles at the core disposal between two subsequent samples is:
$$n_{cc,max} = \frac{t_s}{t_{clock}} = t_s \cdot f_{clock} = 79 \tag{3.4}$$

As for the number of instructions needed per each sample, running the demodulation algorithm (NXP's IP) exploiting the RISC-V base ISA, **152 instructions** per sample are needed.

Simulating on the RTL through the verification environment, the number of clock cycles needed per sample is larger than the number of instructions, since the relation (1 instruction $\Longleftrightarrow$ 1 clock cycle) is not valid due to events such as pipe stalling and memory accesses. This worsens the performance, taking the configuration further away from the requirements.

However, even with the assumption of 1 instruction executed in 1 clock cycle, the performance is 2 times less than the required!

#### 3.1.3.3 Issues

Two main issues (and the corresponding solutions) can be identified:

1. **arithmetic:** the number of `add` and `muli` needed per each output sample grows with the number of taps of the filter $N$ and they could limit the performance if $N$ is large.
   A solution could be the development of a dedicated ALU block to compute MAC efficiently.

2. **memory accesses:** also for the memory accesses the dependence is on the number of taps of the filter $N$. Neglecting compiler optimizations, `lw` should be called $N$ times per output sample. Compiler optimizations (e.g. reuse input samples for more than one output sample without reloading them from the memory) are not always feasible since the number of internal registers available to the user in a RISC-V based processor is not so high (31 R/W registers, not all of them dedicated to the user).
   However, if we assume that all the samples used to compute $y[n]$ are available for the ALU (i.e. they have been loaded from the memory to some registers), only an additional sample $x[n+1]$ is needed to compute $y[n+1]$, with only one additional memory load. We can exploit this *memory buffering* to reduce the memory load operations.

## 3.2 Related work

To improve performance, accelerators could be designed.

Accelerators are key components in many of modern SoCs and they are used to compute intense and specific tasks with lower latency and energy consumption [1]. RISC-V has becoming an affordable choice in such systems since it is open-source, efficient and has been widely explored over the years. One can opt for a basic RISC-V implementation with low area-power, implementing very basic features and acting

as a sort of control unit for accelerators, that are in charge of efficiently executing dedicated tasks.

An example is *LACore* [17] , a programmable accelerator for general-purpose linear algebra applications. It is based on a very complex architecture allowing to bring many of the linear algebra computing capabilities typically reserved for supercomputers to usual SoCs form factor.
In [18] an integration of *NVIDIA Deep Learning Accelerator (NVDLA)* into a RISC-V SoC is presented. *NVDLA* is an accelerator for deep neural networks by NVIDIA, originally designed for its Xavier SoC, then made open source in 2018.
In [19] a coprocessor for matrix multiplication in AI applications has been developed, while in [20] an accelerator for DSP applications generated through hardware generation algorithms is presented.

The implementations above are impressive from the complexity and functional perspective, being out-of-scope for this thesis work and, probably, too much and "oversized" also for the considered application.

More modest, but interesting implementations of core+accelerator architectures are presented in [1] and [21]. They implement the multiply-accumulate operation, but some desired features, such as memory buffering and the IIR support, are missing.

Besides high-level architectures, some improvements could be gained from the exploration of optimization techniques at a lower level, acting on the purely arithmetic part of the design.
Focusing on the MAC optimization, many papers are present and very efficient implementations could be achieved [22][23][24][25][26].
One could also act on the filter structure optimization, as shown in [27] and [28].
Unfortunately, these approaches are often too tailored to very specific structures, resulting very strict and not allowing the needed flexibility.

## 3.3 Improvement alternatives

To improve the performance of **arithmetic** there are different alternatives, some of them already ratified by the RISC-V organization.

- ***MAC*:** Multiply-Accumulate arithmetic unit;
- ***SIMD* (Single-Instruction Multiple-Data):** it consists of packing multiple data in a single register, processing them in parallel;
- ***Vector processing*:** it is based on the same principle of SIMD, but it works at a higher level of abstraction, bringing much more flexibility and higher complexity.

As for as **memory buffering**, things could be not straightforward.
Assuming one wants to implement a memory buffering mechanism using the existing RISC-V ALU, it is difficult to define how the new memory structures (i.e. additional registers) could be used by the RISC-V ALU. The standard Register File in a RISC-V CPU is made by 32 registers and the instructions are defined according to this limitation.

A good approach could be the design of a new and, overall, independent ALU block including both an efficient arithmetic part (e.g. MAC, SIMD) and registers implementing memory buffering. This is the idea that will be followed in this project.

### 3.3.1 MAC

The Multiply-Accumulate (MAC) arithmetic operation is the most common in DSP since it is the foundation of digital filtering.

Mathematically:

$$y = \sum_{i=0}^{N-1} c_i \, x_i \tag{3.5}$$

Porting the general idea to the special case of digital filtering, $c_i$ are the coefficients, while $x_i$ are the samples (feedforward and feedback).

Considering the basic RISC-V ISA, this operation would require $N$ multiplications and $(N-1)$ additions, while a dedicated MAC could be optimized to be much more efficient.

### 3.3.2 SIMD

Single-Instruction Multiple-Data (SIMD) consists of packing multiple data in a single register and performing the same operation on them in parallel.

For instance, consider the usual 32-bit Register File (RF) of a RISC-V processor and that one wants to perform an addition between two registers. With SIMD, each register can be split in $M$ equal parts (usually $M$ is a power of 2) consisting of $32/M$ bits each. On each of the $M$ parts, the same operation can be executed (on $32/M$ bits) and the result can be packed in the same way as the input. In other words, with a single instruction, $M$ operations are made in parallel, incrementing the throughput.

Of course, the data must have a bitwdith suitable to fit in those sub-parts. Usually, in DSP operations, the samples have a quite low bitwidth, taking great advantage of this kind of processing.

In *Figure 3.3* the working principle with $M = 2$ is shown.

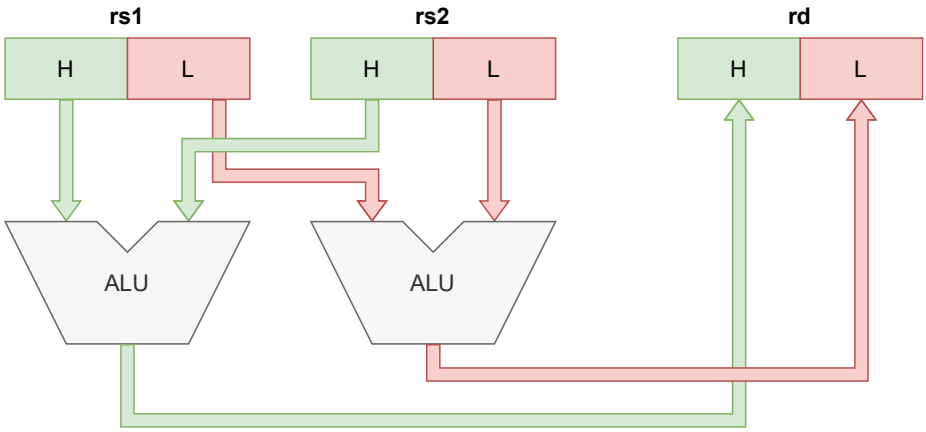



**Figure 3.3:** *Working principle of SIMD ($M = 2$).*

A standardization proposal by the RISC-V organization is present (extension is named $P$), even though it is not ratified. Many of the defined instructions are tailored to DSP applications.

The SIMD instructions are quite simple to be implemented and they do not require much more additional logic.
The main drawback is that these instructions are entirely hardware-dependent, bringing difficulties in code porting. For instance, different CPUs could have different RF sizes and this needs the writing of multiple versions of the same code to operate correctly on the different processors. In addition, the number of instructions to be implemented into the ISA increases with the RF size. These drawbacks can be avoided by working with Vector processing.

### 3.3.3 Vector processing

*Vector processing* is based on the same principle of SIMD (i.e. parallelization), but this strategy is based on a more flexible (and complex) hardware, allowing the programmer to work at a higher level of abstraction.
In SIMD, for the same operation (e.g. `ADD`), multiple instructions to select the right packing are coded (e.g. `ADD8`, `ADD16`), bringing to a large ISA size, even if the effective number of operations defined is quite low. In addition, porting is almost impossible among CPUs with different features.
In vector processing, only one instruction for each operation is defined (e.g. `VADD`) and the selection of the packing, data alignment and other parameters is made by configuring Control-Status Register (CSR). This decreases the ISA size and allows to have a portable code among cores with different specifications since the instructions are abstracted and a different CSRs configuration is enough to make the code work on the different cores.

Vector instructions have a RISC-V organization's proposal, as well, and they have

been ratified (i.e. frozen) on November 2021 [29]. It adds 32 vector registers (`v0-v31`) and seven additional unprivileged CSRs. An astonishing number of different instructions and CSR configurations are possible, bringing great flexibility to the system.
One can set a CSR to choose the number of bits for each sub-part of the vector registers (e.g. 16 bits for each sub-part) and one can even work with multiple vector registers together. For example, considering vector registers of 32 bits each, one can set the usage of two of them together (64 bits total) split into elements of 8 bits (8x8bit elements) for each operation: the hardware will implement the needed logic to perform this operation with a single instruction.
Instructions for vector-vector, vector-scalar or vector-immediate operations are implemented and integer, fixed-point and floating-point numbers are supported, with the possibility to have different rounding schemes.

It is clear that this kind of extension is really powerful and brings a lot of advantages in terms of flexibility and portability, but the hardware required to implement it could be really expensive. In the case we are considering, in which the code developed is firmware for specific hardware and tasks, the porting features are not so useful. For this reason, the vector processing extension is probably too much for our purposes and a SIMD-like approach could be more suitable.

### 3.3.4 Dedicated accelerator

Since we are working with time-critical applications and the core has to be used as an embedded product integrated into special-purpose hardware, the requirements for the project is to achieve better performance with respect to the base RISC-V ISA, but without needing too many different instructions. For this reason, a custom accelerator could be a better solution: the advantage is that only the strictly needed custom instructions are developed (and the corresponding hardware) and the flexibility can be tailored to fit the considered application.

The hardware could be flexible enough to support different kinds of filters, with different orders etc, meanwhile reducing at minimum the overhead that a general-purpose extension would bring. Moreover, we can implement additional features that the extensions cited before do not provide, such as the *memory buffering*, which is one of the bottlenecks of the system.

#### 3.3.4.1 General filter expected results

We replicate the same analysis of *subsubsection 3.1.3.1*, but considering a coprocessor able to:

- compute the MAC operation in 1 clock cycle, despite the number of taps of the filter;
- buffer all the samples needed to compute $y[n]$ (i.e. storing them in internal registers) and using them to compute $y[n+1]$, loading only an additional input sample $x[n+1]$ from the memory.

As in *subsubsection 3.1.3.1*, consider the FIR case (refer to *Equation 3.1*) and assume that the coefficients are already available to the ALU (i.e. no memory load needed for them).
In a first approximation, to compute a single value of $y[n+1]$ the instructions needed are:

- `lw`: 1 time (only 1 additional input sample $x[n+1]$ is needed);
- `mac`: 1 time (the MAC arithmetic unit is able to compute all the Multiply-Accumulate operations needed in one instruction).

We pass from $(3N-1)$ of the RISC-V base ISA to 2 instructions only! Notice that **ideally** the number of instructions needed exploiting the accelerator does **not** depend on the number of taps of the filter $N$. However, this is not completely true, since higher $N$ means higher hardware complexity (more memory for the buffering, more logic for the MAC unit) and a trade-off could be needed.
The improvement is significant anyway.

#### 3.3.4.2 Considered use case expected results

Here the same analysis of *subsubsection 3.1.3.2* is resembled.

As said there, assuming $f_{clock} = 540\,\text{MHz}$, the maximum number of clock cycles available per sample is **79 clock cycles** and the base ISA is not able to reach such a performance.

Defining a dedicated accelerator with the specifications that will be explained in *chapter 4*, the expected number of instructions needed by a sample spans from **29 to 45 instructions** (no details are reported here due to NXP's IP). It is more than 3 times less than the 152 instructions needed with the base ISA!

Of course, the condition (1 instruction $\iff$ 1 clock cycle) is not always satisfied, therefore more clock cycles are needed due to some non-idealities such as pipe stalls and core-accelerator interfacing. These losses could be mitigated through additional hardware and investigations (*subsection B.1.1* is an example).

Despite these limitations, the improvement that could be reached is promising.

The final results after the coprocessor development obtained by the RTL simulations are reported in *section 6.2*.

# Chapter 4

# Accelerator design

In the end, the development of a custom coprocessor/accelerator (both terms will be used throughout this document) was the choice. The development has been greatly simplified by the eXtension InterFace (XIF) by OpenHW group, allowing to couple a coprocessor to the main core exploiting a series of standardized interfaces, as explained in *subsection 2.4.1*. The only OpenHW group's core supporting this interface so far is the CV32E40X and it is the one that has been used in this project.

In this part, the high-level specifications are reported. For more details, refer to *Appendix B* (look at *Introduction* for information about the two parts' purpose).

## 4.1 Working principle

The general architecture is shown in *Figure 4.1* and some elements are worth being explained.

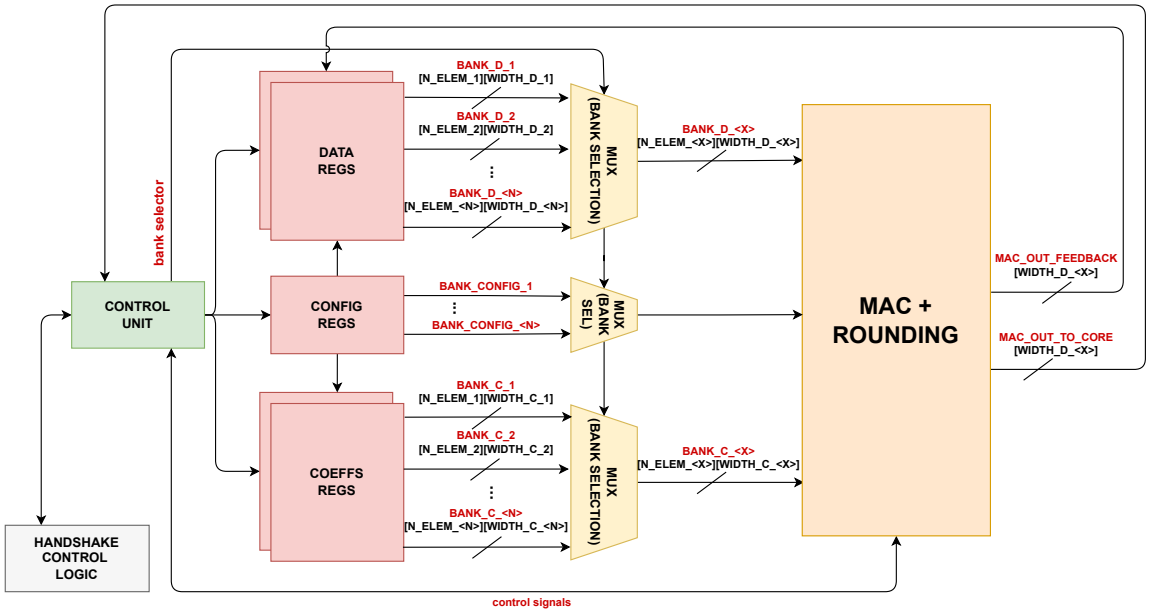



**Figure 4.1:** *ALU MAC architecture.*

The system works with **integer/fixed point 2's complement** numbers. The usage of fixed-point numbers is made possible thanks to a configurable rounding unit present in the design.

The system is tailored to optimize digital filtering, therefore the notation reported in *subsection 3.1.2* will be also used in this section.

There are two *sets* of shift registers (`DATA_REGS` and `COEFFS_REGS`) and they represent the operands of the MAC arithmetic unit. The number of feedforward and feedback elements to be stored can be set, allowing the implementation of FIR and IIR filters. The `DATA_REGS` store the input samples $x[n-i]$ and, eventually, the feedback samples $y[n-j]$.
The `COEFFS_REGS` store the coefficients $b_i$ and, eventually, $a_j$.

Both `DATA_REGS` and `COEFFS_REGS` are made up of different independent *banks* that are identified by an index that will be named `bank_index` throughout this document. A *bank* is a series of memory locations (i.e. an array of multi-bit values) in which data are loaded in a shift-register manner.
Each *data bank* has its corresponding *coeffs bank*, having the same number of elements (i.e. length of the shift register) and indexed together. In other words, each `bank_index` denotes the pair of *data bank* and *coeffs bank*: for each `bank_index`, each

sample in a *data bank* has its corresponding coefficient in the *coeffs bank*.

Each *bank* should be ideally devoted to a single filter, in such a way that each *bank* stores the state of that filter (samples+coefficients) and that state is not modified by operations on other filters. However, one can choose also to dedicate the same *bank* to more than one filter, accepting some performance loss.
We call `N_ELEM_<X>` the number of elements in the *data bank* and *coeffs bank* identified with `bank_index=X`.

`N_ELEM_<X>` can be set differently for each *bank* as RTL parameter: it will be fixed after the synthesis and can not be changed at runtime. If not all the memory locations are needed in the software, one can set the corresponding coefficients to 0.

The bitwidth of each element can be set independently for each *bank* and it can be different between *data banks* and *coeffs banks*.
We call `WIDTH_D_<X>` the bitwidth of the samples stored into the ***data*** *bank* identified with `bank_index=X`.
We call `WIDTH_C_<X>` the bitwidth of the coefficients stored into the ***coeffs*** *bank* identified with `bank_index=X`.
Take into account that the core RF and the data memory are on 32 bits: if `WIDTH_*_<X>` is lower than 32, MSBs truncation will occur during the storing of new data into the registers. This is a wanted behaviour if the values you are storing have a bitwidth lower than 32, as often occurs in DSP applications.

Each *bank* contains `N_ELEM_<X>` elements (that are samples for `DATA_REGS` and coefficients for `COEFFS_REGS`), where:

- `(n_feed_<X>)` are the feedback elements;
- `(n_forw_<X>=N_ELEM_<X>-n_feed_<X>)` are the feedforward elements.

With this structure, one can split the whole `N_ELEM_<X>` size between the feedforward and the feedback elements as desired.
E.g. If one has a filter with 32 feedforward elements and 3 feedback elements, 35 total elements are needed.

In *Figure 4.2* the logical structure of a single bank is shown. The orange line separates the part of the bank dedicated to the feedforward elements from the one related to the feedback ones. The position of the orange line (i.e. the number of feedback elements) can be set by the programmer at runtime.

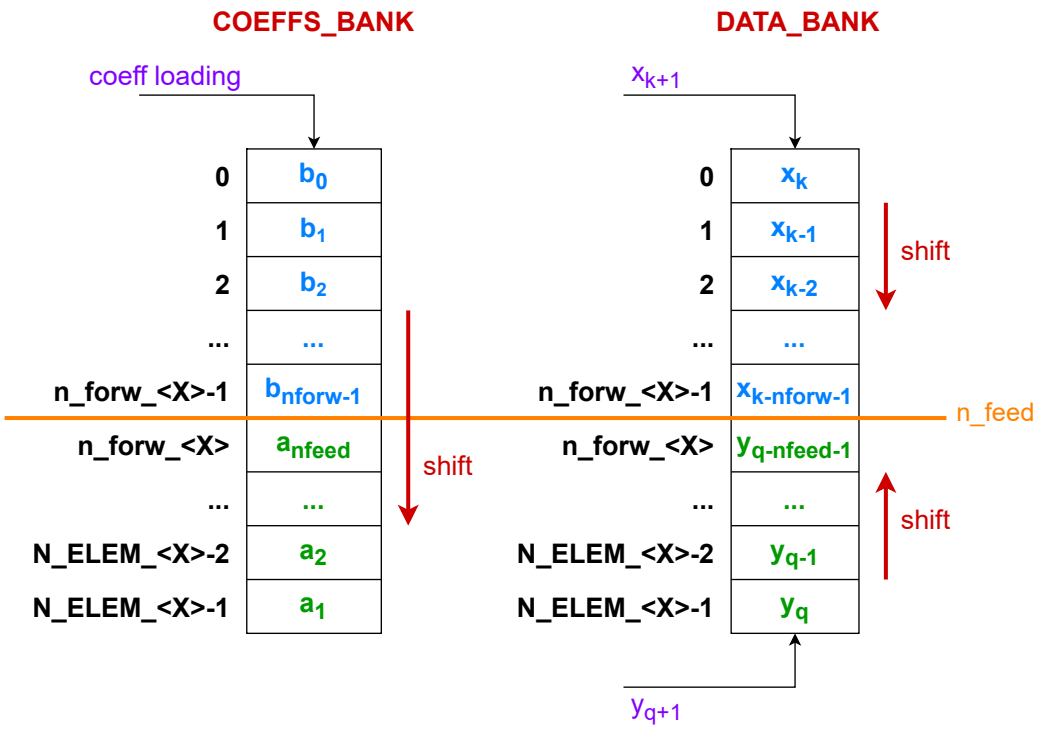



**Figure 4.2:** *Logical structure of a data bank (on the right) and the corresponding coeffs bank (on the left).*

As said, those memory locations behave as a shift register, but there are some differences among the *data banks* and the *coeffs banks*.
The *coeffs banks* are truly shift-registers, with a single input and a single shifting direction (from top to bottom in *Figure 4.2*).
As for the *data banks*, it can be seen as a couple of two independent shift registers, one shifting downward (for the feedforward elements) and another one shifting upward (for the feedback elements). The size of each of the two parts can be set by the programmer at runtime, setting `n_feed_<X>`. In case `n_feed_<X>=0`, only feedforward elements are present, i.e. a single downward shift register. The other limit case is when `n_feed_<X>=(N_ELEM_<X>-1)`: in that case, a single upward shift register is present.
The programmer can load elements only to the feedforward part of the *data banks*, while the feedback part is managed by the hardware which automatically loads the output result of the MAC arithmetic part after each execution.

Once the data/coefficients are loaded, the result of the Multiply-Accumulate operation can be obtained through a dedicated instruction and it is stored in the feedback part (if `n_feed_<X>`$>0$) of the *data bank* on which the operation has been executed. A configurable rounding unit is also present.

The operation can be executed in a standard way or in a packed SIMD fashion, according to a parameter that can be set at runtime. If SIMD is enabled for that

*bank*, each element of the *data bank* and *coeffs bank* is seen as two packed elements and two parallel executions are carried out (see *subsubsection 4.3.1.2*).

## 4.2 Pre-synthesis parameters

Since different configurations have to be supported, the project aimed to have a flexible architecture, without impacting significantly the performance-power-area aspects. To this purpose, the possibilities offered by SystemVerilog and by modern synthesizers have been largely used. The obtained design can be customized through static pre-synthesis parameters, which modify the architecture (e.g. bitwidth, number of registers) without introducing overhead.
There are some parameters to be set in the top-level module of the design (`coproc.sv`) and here their meaning is explained.

| **Parameter name** | **Type** | **Description** |
|---|---|---|
| `N_BANKS_MAC` | *int* | Number of banks for `DATA_REGS` and `COEFFS_REGS`. |
| `N_ELEMENTS_BANK_MAC` | *int*[`N_BANKS_MAC`] | Number of elements for each *bank*. E.g.: `N_ELEMENTS_BANK_MAC[2]` is the number of elements of the *bank* identified with the index `bank_index=2`. |
| `N_ELEMENTS_BANK_MAX_MAC` | *int* | Maximum value into the `N_ELEMENTS_BANK_MAC` vector. |
| `WIDTH_DATA_MAC` | *int*[`N_BANKS_MAC`] | Bitwidth for each *data bank*. E.g.: `WIDTH_DATA_MAC[2]` is the number of elements of the *data bank* identified with the index `bank_index=2`. |
| `WIDTH_DATA_NOSIMD_MAX_MAC` | *int* | Maximum data bitwidth for the no-SIMD operations. |
| `WIDTH_DATA_SIMD_MAX_MAC` | *int* | Maximum data bitwidth for the SIMD operations. |
| `WIDTH_DATA_MAX_MAC` | *int* | Maximum value into the `WIDTH_DATA_MAC` vector. |

| | | |
|---|---|---|
| WIDTH_COEFFS_MAC | *int*[N_BANKS_MAC] | Bitwidth for each *coeffs bank*. E.g.: WIDTH_COEFFS_MAC[2] is the number of elements of the *coeffs bank* identified with the index bank_index=2. |
| WIDTH_COEFFS_NOSIMD_MAX_MAC | *int* | Maximum coeffs bitwidth for the no-SIMD operations. |
| WIDTH_COEFFS_SIMD_MAX_MAC | *int* | Maximum coeffs bitwidth for the SIMD operations. |
| WIDTH_COEFFS_MAX_MAC | *int* | Maximum value into the WIDTH_COEFFS_MAC vector. |
| SIMD_MAC | *bit* | If 1, enable the SIMD support. The SIMD can be then enabled for the desired *banks* through the dedicated instruction at runtime. If SIMD is not needed on any *bank*, set it to 0 to reduce the hardware complexity. |
| N_BIT_NFEED_MAC | *int* | Bitwidth of n_feed. In other words, one can set at runtime the number of feedback elements at a maximum equal to $2^{\texttt{N\_BIT\_NFEED\_MAC}} - 1$. |
| FEEDBACK_UNROUNDED_MAC | *bit* | If 1, the feedback result from the MAC arithmetic block to be stored into the feedback registers of the selected *data bank* will be unrounded. Otherwise, it will be rounded according to the rounding unit configuration. **Notice that the result stored into the core RF will be always rounded.** |

| | | |
|---|---|---|
| `N_OPERANDS_MAC` | *int* | Number of operands that the MAC arithmetic block can manage in parallel. If `N_OPERANDS_MAC` <`N_ELEMENTS_BANK_MAC[bank_index]` the execution on the bank with index `bank_index` will be performed with more than one iteration. In other words, by lowering this parameter, the hardware complexity is lower (look at *section B.3* to know more), but the latency is higher. |
| `BITS_CUT_MAX_MAC` | *int* | Maximum number of bits that the rounding unit can round. |

Some notes:

- parameters such as `N_ELEMENTS_BANK_MAX_MAC` and `WIDTH_DATA_MAX_MAC` could be obtained simply using a `max()` function since they are the maximum values stored into the corresponding vectors (`N_ELEMENTS_BANK_MAC` and `WIDTH_DATA_MAC` respectively).
  Unfortunately, such a function in SystemVerilog cannot be used for this kind of parameters, therefore a manual definition for them is needed.

- `WIDTH_DATA_SIMD_MAX_MAC` and `WIDTH_COEFFS_SIMD_MAX_MAC` are related to SIMD operations only (see *subsubsection 4.3.1.2*). These parameters have to be set with the entire bitwidth of the SIMD packed elements.
  E.g. if you assume to have SIMD packed elements made by two 13-bit values, you have to set 26 here.

## 4.3 Custom instructions

Twelve custom instructions have been implemented. They can be divided into five groups according to their function:

- **Configuration and clearing:** configure the MAC architecture and clear the registers.
- **Loading coefficients:** load the coefficients (from core RF or from data memory).
- **Loading data:** load the samples (from core RF or from data memory).
- **Executing MAC:** execute the MAC operation and store the result into the core RF.
- **Loading data and execute MAC:** load a new sample and execute the MAC operation.

The coding has been chosen to be compliant with the RISC-V standard ISA. To this purpose, specific opcodes officially provided by *RISC-V International* can be used, ensuring no conflicts with the current and future RISC-V standard extensions. The chosen opcode is `0101011`.

The chosen opcode does not belong to an existing RISC-V instruction, therefore it defines a completely new encoding space (*greenfield extension*). This encoding space has been used to code the twelve instructions developed with a *brownfield extension* approach. This means that the instructions share the same opcode, being differentiated by other fields (`funct3` and `funct7` in our coding).

### 4.3.1 Configuration and clearing

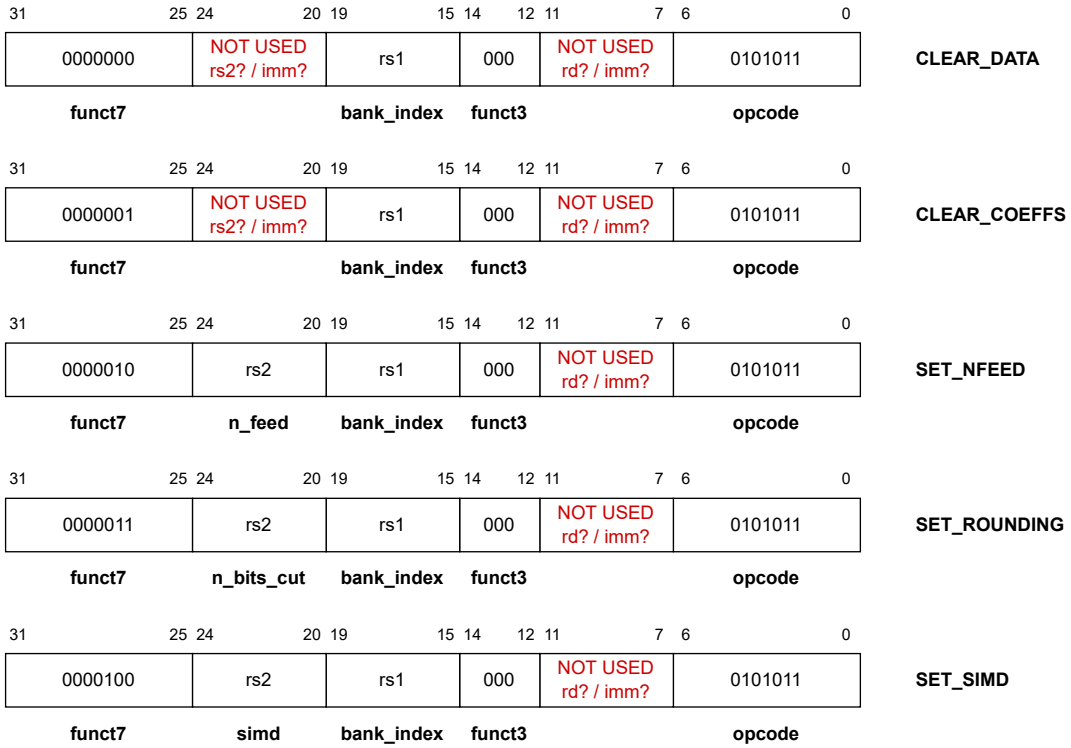



**Figure 4.3:** *Configuration and clearing instructions coding.*

These instructions are coded with the same *opcode*, same *funct3* field and they are differentiated by the *funct7* field. There are many more *funct7* codes that can be used for future extensions. The fields shown in red in *Figure 4.3* are not used for now.

The purpose of each instruction is:

- `CLEAR_DATA`: the *data bank* identified with the index `bank_index` (passed through the GPR `rs1`) is cleared to `0` (feedback part included).
- `CLEAR_COEFFS`: the *coeffs bank* identified with the index `bank_index` (passed through the GPR `rs1`) is cleared to `0`.
- `SET_NFEED`: the number of registers to be used for the feedback elements is set to `n_feed` (passed through the GPR `rs2`) for the bank identified with the index `bank_index` (passed through the GPR `rs1`).
- `SET_ROUNDING`: the rounding unit is set to round a number of bits equal to `n_bits_cut` (passed through the GPR `rs2`) for the bank identified with the index `bank_index` (passed through the GPR `rs1`). See *subsubsection 4.3.1.1*.

- (only available if `SIMD_MAC=1`)
  `SET_SIMD`: if `simd=1` (passed through the GPR `rs2`), the MAC is set to use SIMD for the bank identified with the index `bank_index` (passed through the GPR `rs1`). If `simd=0`, the MAC will not use SIMD for that bank. See *subsubsection 4.3.1.2*.

#### 4.3.1.1 Rounding unit details

Two types of rounding unit have been developed and one can easily switch between them acting on the RTL modules instantiation in the SystemVerilog files.

With the `SET_ROUNDING` instruction, one can set the number of bits (LSBs) to round.

Consider a fixed-point number and assume we want to round this number to an integer. Schematically:

$$(x_{(k-1)}\, x_{(k-2)} \ldots x_1\, x_0\, x_{-1}\, x_{-2} \ldots x_{(-l+1)}\, x_{-l}) \quad \rightarrow \quad (x_{(k-1)}\, x_{(k-2)} \ldots x_1\, x_0)$$

where each $x_i$ is a bit, $k$ is the number of integer bits and $l$ the number of fractional bits. The format of the number on the left can be expressed using the following notation `Q<k>.<l>`.

We call *Unit of Least Precision (ulp)* the LSB of the rounded output, that is $x_0$ in the example. It defines the precision after the rounding.

The two rounding units developed are:

- **round to nearest:** it rounds towards the nearest number; when exactly in the middle, always round-up.
  It is the most common rounding scheme and it ensures that the maximum rounding error is 1/2 *ulp*. However, on average, a slight positive bias is present (since it always rounds up when in the middle), which can be harmful in rare cases for feedback systems (e.g. IIR).
  Examples:
  12.3 → 12
  12.8 → 13
  12.5 → **13**
  13.5 → **14**

- **round to nearest even:** it rounds towards the nearest number; when exactly in the middle, round towards the nearest even number.
  The error is 1/2 *ulp* as well, but the advantage is that on average no bias is present. It is suggested for feedback systems in which using the *round to nearest* scheme, a drift during the execution is present.
  Examples:
  12.3 → 12

$12.8 \rightarrow 13$
$12.5 \rightarrow \mathbf{12}$
$13.5 \rightarrow \mathbf{14}$

An example can be considered:

- 8 operands (data-coefficients pairs) for Multiply-Accumulate operations;
- *data* ($x_i$) represented as fixed-point `Q10.6`;
- *coefficients* ($c_i$) represented as fixed-point `Q3.2`;
- *output* ($y$) wanted in fixed-point `Q10.6`;
- filter designed to be stable and to have an output always fitting on an integer part of 10 bits.

The operation to be executed is a Multiply-Accumulate:

$$y = \sum_{i=0}^{7} c_i \, x_i \tag{4.1}$$

Therefore:

1. for each $c_i \, x_i$ multiplication (namely *partial term*), the number of bits is given by the sum of the number of bits of $c_i$ and $x_i$ (in the example, each partial term will be in the form `Q(10+3).(6+2)=Q13.8`).
2. adding up all the 8 partial terms $c_i \, x_i$, the number of bits of the integer part increases by $\log_2(8) = 3$, giving `Q(13+3).8=Q16.8`.

Since we assumed that the filter is designed to be stable and to have an output always fitting on an integer part of 10 bits, the result can be expressed with the form `Q10.8`. Two bits of the fractional part have to be cut off to have the desired output format `Q10.6`.

In the example above, the rounding unit has to be set with `n_bits_cut=2`, since two bits have to be cut off.

#### 4.3.1.2 SIMD details

The `simd` parameter can be set for each bank to `0` (SIMD off) or `1` (SIMD on) through the dedicated instruction `SET_SIMD`.

With the SIMD enabled for a bank, the *data* and the *coeffs* bitwidth of that bank is virtually split into two parts and two data/coefficients are packed into one register

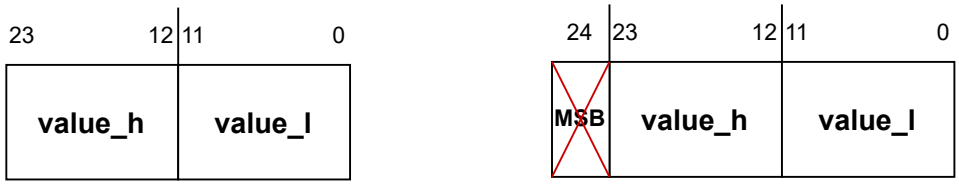
location.

The packing for the **data** contained into the *data banks* is made according to the `WIDTH_DATA_SIMD_MAX` pre-synthesis parameter.
The packing for the **coefficients** contained into the *coeffs banks* is made according to `WIDTH_COEFFS_SIMD_MAX` pre-synthesis parameter.
The packing for the **output** after a Multiply-Accumulate execution is made according to `WIDTH_DATA_SIMD_MAX` pre-synthesis parameter.
In this way, feedback storing into *data banks* if `(n_feed_<X>)>0` can occur without issues since the packing is compliant.

If `WIDTH_DATA_SIMD_MAX_MAC` or `WIDTH_COEFFS_SIMD_MAX_MAC` is odd, the MSB is discarded and not considered.
If `(WIDTH_DATA_SIMD_MAX_MAC<WIDTH_DATA_MAX_MAC)` or `(WIDTH_COEFFS_SIMD_MAX_MAC<WIDTH_COEFFS_MAX_MAC)`, the exceeding MSBs are discarded and not considered.
In the end, the only parameter controlling the SIMD packing is `WIDTH_*_SIMD_MAX` (where `*` can be both `DATA` or `COEFFS`).

In *Figure 4.4*, an example of packing with `WIDTH_*_SIMD_MAX=24` and `WIDTH_*_SIMD_MAX=25` is shown. `WIDTH_*_MAX_MAC` can assume whatever value, the packing will not change and the exceeding bits will be discarded and not considered.

With SIMD turned on, the Multiply-Accumulate operation is made on both values in parallel, speeding up the execution.




**(a)** *WIDTH_*_SIMD_MAX=24* **(b)** *WIDTH_*_SIMD_MAX=25*

**Figure 4.4:** *SIMD packing.*

### 4.3.2 Loading coefficients

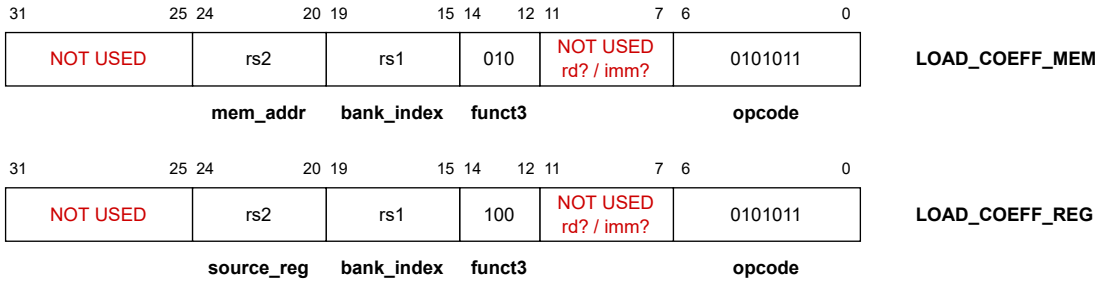



**Figure 4.5:** *Coefficients loading instructions coding.*

One can load **one** new **coefficient** into the *coeffs bank* identified with the index `bank_index`. The loading is performed in a shift register fashion (refer to *Figure 4.2*) and the coefficient can be loaded from the memory or from core GPR.

- `LOAD_COEFF_MEM`: the value stored into the data memory at the memory address `mem_addr` (passed through the GPR `rs2`) is loaded into the *coeffs bank* identified with the index `bank_index` (passed through the GPR `rs1`).
- `LOAD_COEFF_REG`: the value stored into the GPR `rs2` is loaded into the *coeffs bank* identified with the index `bank_index` (passed through the GPR `rs1`).

All the coefficients (i.e. `N_ELEM_<X>` coefficients) should be loaded to avoid undefined values. There are `n_feed_<X>` feedback coefficients and `n_forw_<X>` feedforward coefficients, where (`n_feed_<X>` + `n_forw_<X>` = `N_ELEM_<X>`).
These instructions only load **one** value in a shift register fashion, therefore one has to perform the operation `N_ELEM_<X>` times to set all the coefficients.
If one *bank* per filter is used, one needs to set the coefficients for each filter only at the beginning of the execution, without reloading them every time, since they remain stored during the execution.

One must follow a precise order for the coefficients in order to load them correctly.

The order must be:

1. (skip this part if `n_feed_<X>=0` for the considered bank)
   feedback coefficients starting from the first one (i.e. from $a_1$ to $a_{\texttt{nfeed}}$);
2. feedforward coefficients starting from the last one (i.e. from $b_{(\texttt{nforw}-1)}$ to $b_0$).

### 4.3.3 Loading data

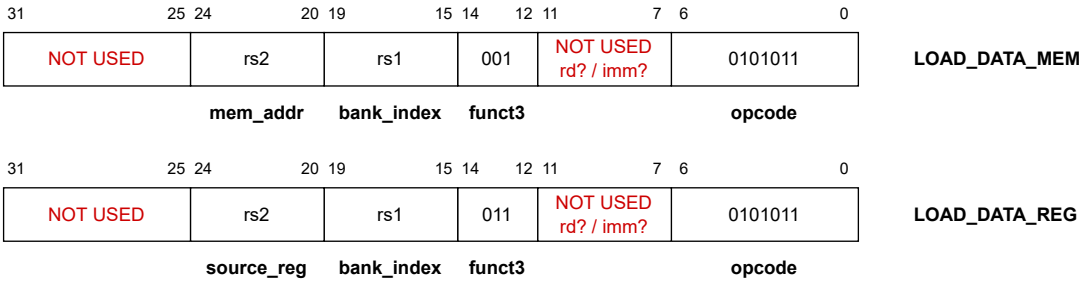



**Figure 4.6:** *Data loading instructions coding.*

One can load **one** new **data** into the *data bank* identified with the index `bank_index`. The loading is performed in a shift register fashion (refer to *Figure 4.2*) and the data can be loaded from the memory or from a core GPR.

- `LOAD_DATA_MEM`: the value stored into the *data memory* at the memory address `mem_addr` (passed through the GPR `rs2`) is loaded into the *data bank* identified with the index `bank_index` (passed through the GPR `rs1`).
- `LOAD_DATA_REG`: the value stored into the GPR `rs2` is loaded into the *data bank* identified with the index `bank_index` (passed through the GPR `rs1`).

Referring to *Figure 4.2*, the new value $x[n]$ is loaded from the top input. The bottom input $y[n]$ is up to the internal hardware logic and cannot be loaded manually, but it is taken by the MAC output.

Only one value is loaded per each instruction call. In the beginning, one needs to fill the buffer with a number of values determined by the taps of the filter, then the buffering can be exploited, by loading only one new $x[n+1]$ to get $y[n+1]$.

### 4.3.4 Executing MAC

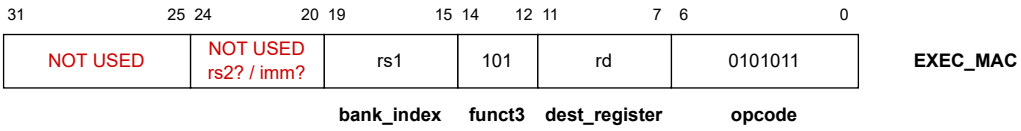



**Figure 4.7:** *Multiply-accumulate execute instruction coding.*

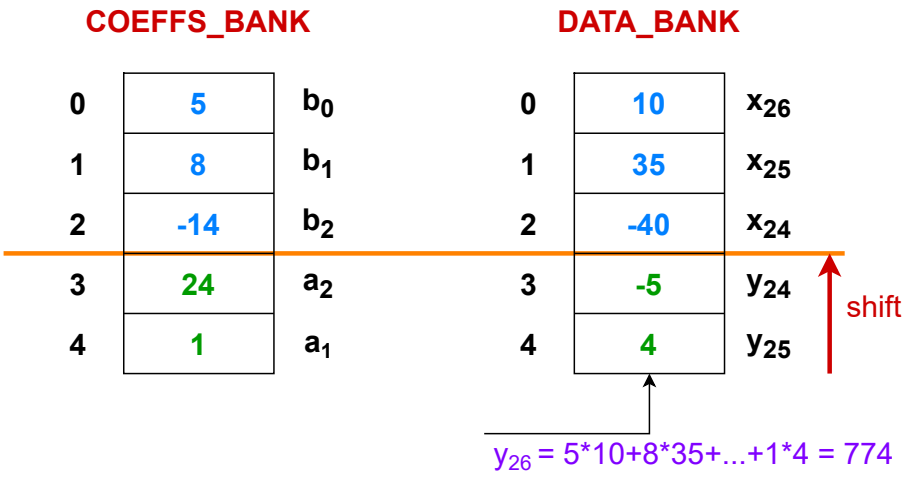
MAC is executed on the values stored in *data banks* and *coeffs banks* identified with the index `bank_index` (passed through the GPR `rs1`).

Taking as reference the *Figure 4.2*, the operation executed is:

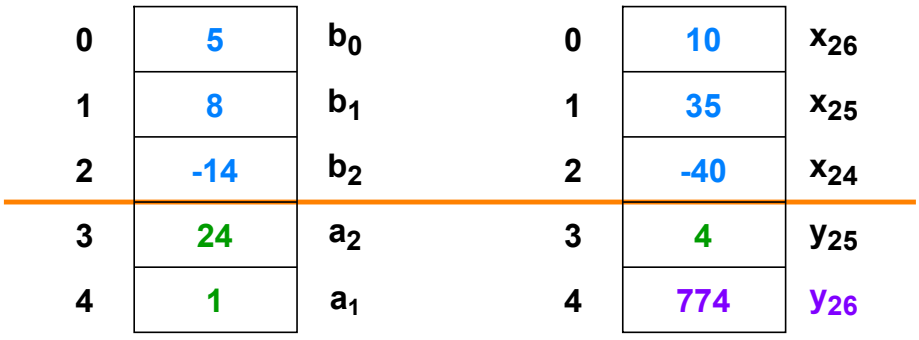
$$y_{q+1} = \sum_{m=0}^{(\texttt{N\_ELEM\_<X>}-1)} \texttt{COEFFS\_BANK}[m] \cdot \texttt{DATA\_BANK}[m]$$

The value $y_{q+1}$ is stored in the GPR `dest_register` (`rd`) and in the feedback part of the *data bank* identified with the index `bank_index` (passed through the GPR `rs1`).

An example of how the state of the register changes in case of a no-SIMD operation is shown in *Figure 4.8*.




**(a)** *Phase 1: MAC computation.*




**(b)** *Phase 2: feedback output loaded.*

**Figure 4.8:** *`EXEC_MAC` working example (`simd=0` for the considered bank).*

If SIMD is enabled for the selected bank, the working principle is the same, but each *data*/*coeffs* location is treated as made by two packed values and the Multiply-

Accumulate output will be made by two packed values as well, as explained in *sub-subsection 4.3.1.2*.

### 4.3.5 Loading data and execute MAC

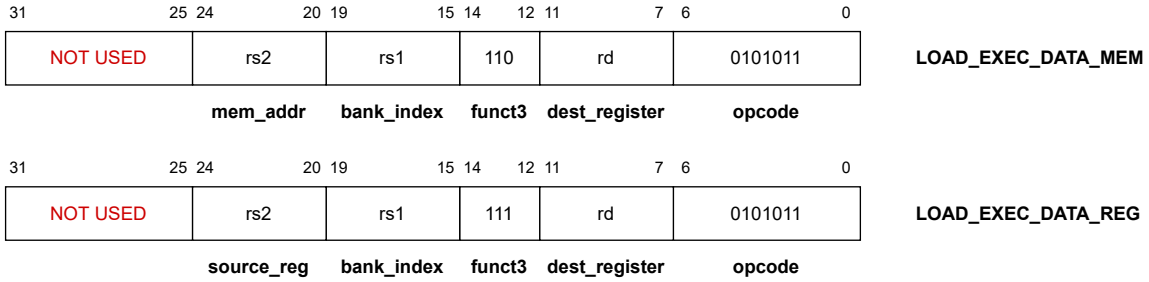


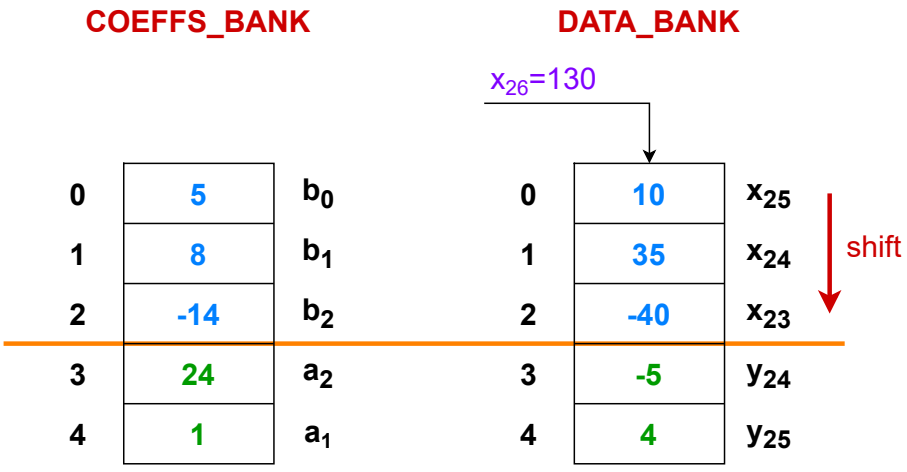

**Figure 4.9:** *Data loading + Multiply-Accumulate execution instructions coding.*

These instructions can be seen as the subsequent execution of `LOAD_DATA_[MEM/REG]` and `EXEC_MAC`.

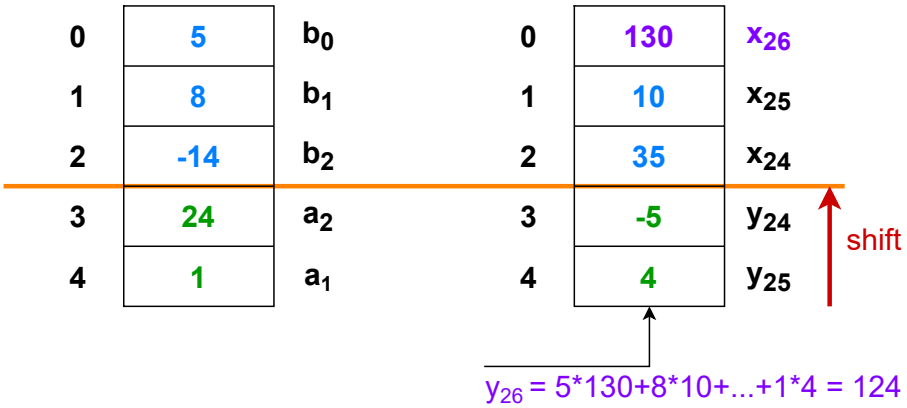
The decision to define these instructions comes from the acknowledgement that loading a new sample $x[n+1]$ and, subsequently, computing the new output $y[n+1]$ is the most common operation in digital filtering. Defining these two instructions allows saving some clock cycles with respect to calling a straight `LOAD_DATA_[MEM/REG]` and then a straight `EXEC_MAC` since in that case there is an additional overhead (e.g. decoding, core-coprocessor handshaking). The working principle and the parameters are the same as the two instructions taken individually.

An example of how the state of the register changes is shown in *Figure 4.10*.

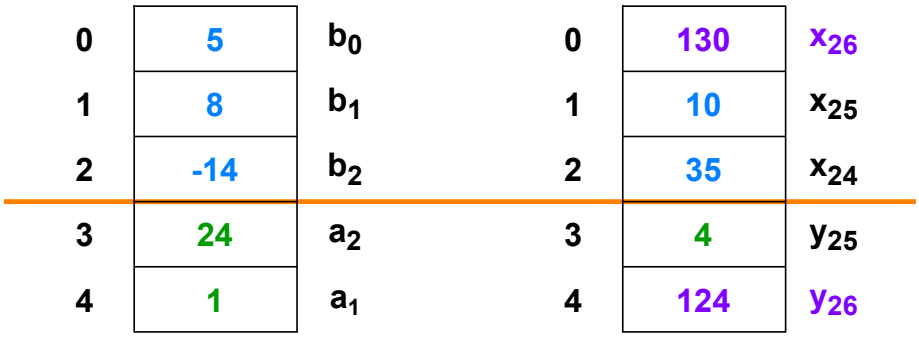
Also for these instructions, the same SIMD considerations made for the straight `EXEC_MAC` apply.




(a) *Phase 0: initial state.*




(b) *Phase 1: new sample loaded and MAC computation.*




(c) *Phase 2: feedback output loaded.*

**Figure 4.10:** *LOAD_EXEC_DATA_[MEM/REG] example.*

# Chapter 5

# Accelerator testing-verification

After the design, a phase of verification has been carried out to assess the correct functionality of the designed system. Starting from the verification environment developed by the OpenHW group, additional features have been added and a custom verification mechanism has been developed and integrated.

The core verification by OpenHW is not completed, yet, but it is ongoing. Some small bugs have been discovered and fixed during this thesis project, even though its stability and usability are really good.
However, for the coprocessor testing, it was impossible to carry out an exhaustive verification (e.g. assertions, formal methods, ISS), but a trade-off has been chosen. An auto-checking testing mechanism, more advanced than the simple "*write some code and test the execution manually*", was needed. Firstly, because the core-coprocessor system is quite complex and it is impossible to excite manually a good number of cases to test, and then because the high flexibility of the coprocessor makes the verification much more complex since there are potentially infinite parameter combinations.

Two layers of testing have been considered:

- **software independent testing:** during this process, the hardware has been tested with random stimuli not related to the specific DSP algorithm to be run. This process is explained in this chapter.

- **software dependent testing:** after the software independent testing, the DSP algorithm has been run on the system and the outcome compared with a MATLAB reference model. This ensured that the software was correct and, to some extent, it added further validity to the hardware testing. This process is software related and it will not be treated within this thesis.

## 5.1 Verification environment extension

The OpenHW group's verification environment is UVM-based and its main features have been described in *section 2.6*.

To simplify the algorithm testing and verification, the environment has been extended with the possibility to pre-load data taken from a file into a region of the data memory and to dump a memory region onto an output file. The base address, the number of words and the filenames can be independently chosen both for the data to pre-load and for the ones to dump. In addition, they can be independently disabled or enabled.
All these parameters can be set as variables in one of the makefiles, resulting in completely integrated into the OpenHW verification environment.

For their implementation, the SystemVerilog testbench has been slightly modified introducing a few additional lines; in particular, the SystemVerilog standard functions `$writememh()` and `$readmemh()` have been exploited.
With the former, the content of the memory can be dumped onto an output file in hexadecimal form, while the latter performs the opposite operation, transferring the content of a file into the memory. The input file can be a usual `.txt` and the data must have a specific format (refer to the official documentation for more information).

## 5.2 Verification approaches

An auto-checking verification process was needed instead of a manual approach due to the system's (i.e. core+coprocessor) complexity and flexibility. Another important aspect is the randomization of the testcases since manual definition rarely excites the corner cases.

There are different approaches to verification, with different levels of effectiveness and complexity:

- **exploit the UVM features of the verification environment:** this means going on with assertions, formal methods and complex reference models. This is the approach followed in the industry and by the OpenHW group, but it requires a very good knowledge of verification methodologies and it is time-consuming.

  Moreover, as explained in *section 2.6*, the reference model used is an *Instruction Set Simulator (ISS)* called *OVPsim*, already integrated into the environment and already implementing the CV32E40X features [15]. A good approach could have been to extend it with the coprocessor features and exploit it for verification. With an ISS, all the internal registers (GPR, CSRs), the memory reading/writing and, in general, almost all the aspects of the system could be checked in a very exhaustive way.
  Unfortunately, extending an ISS could be tricky, requiring a great effort. In

addition, we had to abandon the idea since *OVPsim* has licensing policies that could have been not compatible with the fact that this thesis has been carried out in a company.
The corresponding fully open-source ISS alternative was *Spike*, but it has some limitations and it is not yet integrated into the verification environment [30].

- **work outside the verification environment:** a simpler, but limited, approach is to work outside the verification environment and then join and check the correctness of the results somehow. The advantage is that one does not need to face the complexity of the already present testbench, understand how it is implemented and extend it. However, internal signal checking is less effective and this approach is often less flexible.

### 5.2.1 Our approach to the verification

Working within the testbench exploiting UVM and ISS for the verification was not feasible for the reasons explained above. Therefore, the choice was to work outside the verification environment and to join the results in the end, checking their correctness.

The verification scheme is shown in *Figure 5.1*.

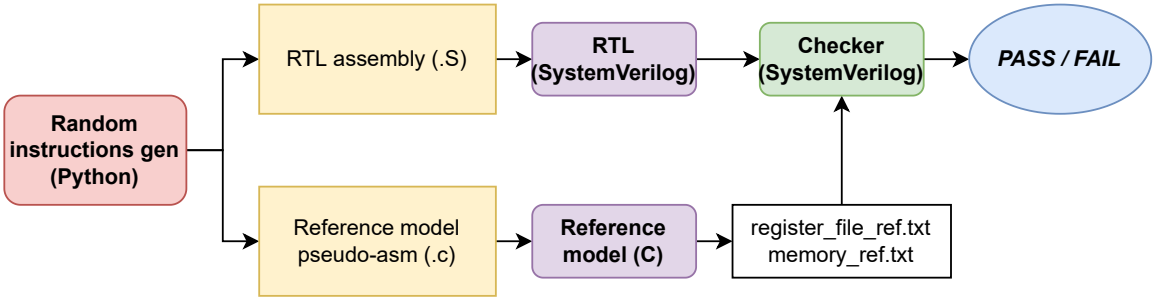



**Figure 5.1:** *Verification scheme.*

It is based on two parallel flows, one related to the RTL simulation and the other one to the reference output generation:

- A C reference model implementing the basic RISC-V instructions (e.g. arithmetic, branch, jump) and the accelerator instructions has been developed. It implements the Assembly (ASM) instructions as C functions resembling the system's expected behaviour.

- A Python script generates random instructions packing them into two different files: one assembly file (`.S`), to be compiled through the toolchain for the RTL simulation, and the corresponding pseudo-ASM code in C, for the reference model.

- The reference model executes the pseudo-ASM and generates two files with the

expected behaviour of Register File and memory. These two files will be taken as a reference by the checker.

- A SystemVerilog checker integrated into the testbench checks the correctness of the result. The register file content is checked every time it is written (i.e. `rf_write_enable=1`), while the memory is checked only at the very end of the execution for efficiency reasons. A `PASS` or a `FAIL` are then generated.

Each component is configurable through some parameters: most of them are the RTL pre-synthesis parameters (see *section 4.2*) to match the exact accelerator behaviour, while others are related to some features of the verification flow.

### Random instructions generator

This is a Python script able to generate random assembly instructions for the RTL (`.S`) and pseudo-ASM for the reference model (`.c`). Headers, footers and all the needed lines for the correct compilation plus some comments within the code are also generated by this script, making the files compilable without additional manual modifications.

Some RTL pre-synthesis parameters need to be set here (e.g. `N_BANKS_MAC`, `SIMD_MAC`) in order to control the instruction generation. For instance, if `SIMD_MAC=0`, the SIMD-related instructions (`SET_SIMD`) are not available for the RTL and they are not generated in the code.

This unit generates the following instruction types:

- RISC-V basic **arithmetic** instructions, both register-based and immediate-based (e.g. `add/addi`, `or/ori`, `slt/slti`);
- RISC-V **flow control** instructions (e.g. `beq`, `bge`, `j`);
- RISC-V **load/store from/to memory** (`lw`, `sw`);
- all the **accelerator** custom instructions.

All the parameters are randomly generated (e.g. immediate, register index). However, for the instructions related to load/store, the indexed memory region is randomly generated within specific address intervals set by the user through dedicated variables in the program. This is crucial to avoid unwanted writing in memory regions reserved for the system, which could cause issues during the code execution.

At the beginning of the generated programs, the register file and the memory regions are initialized at random values and the MAC is configured with random but valid settings.

The most tricky aspect of the script is the branch/jump generation. This is a very

relevant aspect since this kind of instruction is the most error-prone to be executed in a processor, mainly due to control hazards. Therefore, an extensive test of them was needed.
For this reason, the Python script allows the generation of nested branch/jump that makes the generated ASM program more suitable for an extensive test of the control structures of the system. For each conditional branch in ASM, an `if...else` block is generated for the C pseudo-code feeding the reference model and they can be nested until a configurable maximum nesting value is reached. In other words, the number of nested branches one can have is limited by a parameter that the user can set.
Also, the maximum number of instructions one can have within an `if...else` block can be customized by the user.
Experimentally, reasonable values are `max_nesting_index=3` and `max_instr_within_branch=20`.

Parts of the two randomly generated files are shown in *Source Code 5.1* and *Source Code 5.2*.
The `print_rf_to_file()` function in the C code is used for the RF reference file generator (see *Reference model*).

```
1415                srai x20, x9, 23
1416                add x24, x8, x8
1417                // load_data_reg rs2: x6, rs1: x9
1418                .word 0x5e64bb2b
1419                beq x7, x6, start_else_3
1420                    sltu x25, x29, x28
1421                    slli x23, x6, 21
1422                    li x31, 0x91012c
1423                    lw x23, 0(x31)
1424                    sll x26, x23, x8
1425                    srl x18, x9, x30
1426                    j end_else_3
1427                start_else_3:
1428                    srl x23, x18, x30
1429                    andi x18, x6, 1144
```

**Source Code 5.1:** *Part of a randomly generated ASM program.*

```
1207                SRAI(23, 9, 20);
1208                print_rf_to_file("hex", file_rf);
1209                ADD(8, 8, 24);
1210                print_rf_to_file("hex", file_rf);
1211                load_data_reg(6, 9);
```

```
1212                    if (!(register_file[7] == register_file[6])) {
1213                        SLTU(28, 29, 25);
1214                        print_rf_to_file("hex", file_rf);
1215                        SLLI(21, 6, 23);
1216                        print_rf_to_file("hex", file_rf);
1217                        LUI(0x910, 31);
1218                        print_rf_to_file("hex", file_rf);
1219                        ADDI(300, 31, 31);
1220                        print_rf_to_file("hex", file_rf);
1221                        // starting lw (we need these lines to mimic the
    ↪  HW behavior)
1222                        LW(0, 31, 23);
1223                        lw_temp_value = register_file[23];
1224                        register_file[23] = 0;
1225                        print_rf_to_file("hex", file_rf);
1226                        register_file[23] = lw_temp_value;
1227                        print_rf_to_file("hex", file_rf);
1228                        // ending lw
1229                        SLL(8, 23, 26);
1230                        print_rf_to_file("hex", file_rf);
1231                        SRL(30, 9, 18);
1232                        print_rf_to_file("hex", file_rf);
1233                    }
1234                    else {
1235                        SRL(30, 18, 23);
1236                        print_rf_to_file("hex", file_rf);
1237                        ANDI(1144, 6, 18);
1238                        print_rf_to_file("hex", file_rf);
```

**Source Code 5.2:** *Part of a randomly generated pseudo-ASM C code for the reference model.*

### Reference model

The C reference model implements the ASM instructions listed before as C functions. The main purpose of a reference model is implementing functionalities that could be very complex in HW using high-level constructs that simplify the development and reduce the possibilities of bugs. In other words, it is convenient that the reference model is as simple as possible and for this reason, high-level programming languages are preferred.

There are different alternatives, such as Python, C and even SystemVerilog, which is a really powerful language not only for HW development.
However, during the definition of the specifications a C reference model was developed to allow to the colleague developing software to integrate it into the algorithm. Its

purpose was to test the high-level working principle of the accelerator before the actual RTL development, to discover if it was suitable for the application. This high-level model was developed in C because the DSP software algorithm is written in C as well.
For this reason, after the RTL development, that reference model has been taken as starting point and extended to perfectly mimic the HW behaviour, without starting from scratch its development in a different programming language.

Thinking about it, SystemVerilog would have been a better choice, allowing a tighter integration with the testbench. Moreover, SystemVerilog would have been better for the definition of the arithmetic operations, since in C the programmer is forced to use variables with power of 2 number of bits (8/16/32/64/128 bits) that are not so intuitive to be used to represent RTL signals with a custom bitwidth.
As for the register file and the memories, they have been implemented as arrays in C, making their reading/writing extremely easy.
Also for the reference model, all the RTL pre-synthesis parameters are present (see *section 4.2*) to correctly mimic the HW.

The list of the available C pseudo-ASM instructions is shown in *Source Code 5.3*.

```
89  /*
90  INSTRUCTIONS THAT CAN BE CALLED BY THE PROGRAMMER IN ASSEMBLY
91  */
92  // BASIC RISCV INSTRUCTIONS
93  // they are written in upper case to avoid conflicting with C++
    ↪  keywords
94  void LUI(int32_t imm, uint32_t rd);
95  void LW(int32_t imm, uint32_t rs1, uint32_t rd);
96  void SW(int32_t imm, uint32_t rs1, uint32_t rs2);
97  void ADDI(int32_t imm, uint32_t rs1, uint32_t rd);
98  void SLTI(int32_t imm, uint32_t rs1, uint32_t rd);
99  void SLTIU(int32_t imm, uint32_t rs1, uint32_t rd);
100 void XORI(int32_t imm, uint32_t rs1, uint32_t rd);
101 void ORI(int32_t imm, uint32_t rs1, uint32_t rd);
102 void ANDI(int32_t imm, uint32_t rs1, uint32_t rd);
103 void SLLI(int32_t imm, uint32_t rs1, uint32_t rd);
104 void SRLI(int32_t imm, uint32_t rs1, uint32_t rd);
105 void SRAI(int32_t imm, uint32_t rs1, uint32_t rd);
106 void ADD(uint32_t rs2, uint32_t rs1, uint32_t rd);
107 void SUB(uint32_t rs2, uint32_t rs1, uint32_t rd);
108 void SLL(uint32_t rs2, uint32_t rs1, uint32_t rd);
109 void SLT(uint32_t rs2, uint32_t rs1, uint32_t rd);
110 void SLTU(uint32_t rs2, uint32_t rs1, uint32_t rd);
111 void XOR(uint32_t rs2, uint32_t rs1, uint32_t rd);
```

```
112 void SRL(uint32_t rs2, uint32_t rs1, uint32_t rd);
113 void SRA(uint32_t rs2, uint32_t rs1, uint32_t rd);
114 void OR(uint32_t rs2, uint32_t rs1, uint32_t rd);
115 void AND(uint32_t rs2, uint32_t rs1, uint32_t rd);
116 // COPROCESSOR
117 void clear_data(uint32_t rs1_bank_index);
118 void clear_coeffs(uint32_t rs1_bank_index);
119 void set_nfeed(uint32_t rs2_nfeed, uint32_t rs1_bank_index);
120 void set_rounding(uint32_t rs2_nbits_cut, uint32_t rs1_bank_index);
121 void set_simd(uint32_t rs2_simd, uint32_t rs1_bank_index);
122 void load_coeff_mem(uint32_t rs2_mem_addr, uint32_t rs1_bank_index);
123 void load_coeff_reg(uint32_t rs2, uint32_t rs1_bank_index);
124 void load_data_mem(uint32_t rs2_mem_addr, uint32_t rs1_bank_index);
125 void load_data_reg(uint32_t rs2, uint32_t rs1_bank_index);
126 void exec_mac(uint32_t rs1_bank_index, uint32_t rd);
127 void load_exec_data_mem(uint32_t rs2_mem_addr, uint32_t
    ↪ rs1_bank_index, uint32_t rd);
128 void load_exec_data_reg(uint32_t rs2, uint32_t rs1_bank_index,
    ↪ uint32_t rd);
```

**Source Code 5.3:** *Header of the defined C pseudo-ASM instructions.*

The reference model executes the program file generated by the Python script and it generates in turn the two reference text files that will be used by the checker.
The two files generated are:

- `register_file_ref.txt`
  It is a list of the RF snapshots taken every time a writing operation is performed (i.e. `rf_write_enable=1`). The checker will compare each snapshot with the corresponding RTL behaviour.

- `memory_ref.txt`
  It stores the very last state of the user memory region after the execution of all the instructions. It does not store all the intermediate snapshots, as for the RF, since the memory region could be very large, impacting the efficiency of the checker.

### Checker

The checker is implemented in SystemVerilog and integrated into the existing testbench. It is a module instantiated only for the coprocessor verification (some variables are defined in the makefiles to enable it), being disabled if verification has not to be performed.

When a RF writing operation is occurring during the RTL execution, it compares the actual RF snapshot taken from the RTL with the `register_file_ref.txt` given

by the reference model. If the snapshots coincide, it goes on with the execution waiting for another writing operation. Otherwise, it throws a UVM error and stops the simulation, signalling the index of the register for which there is no compliance with the reference model.
A log file with all the snapshots taken from the RTL is generated for debugging purposes.

A similar approach is used for the memory, but the checking occurs only one time at the very end of the execution.

# Chapter 6

# Results

As explained in *chapter 3*, the considered use case is the software implementation of an **NFC digital baseband demodulator**, compliant with the ISO/IEC 14443 standard.

## 6.1 RTL parameters for the considered use-case

The HW configurations used for the considered use case are shown in *Table 6.1* and *Table 6.2*.

Two different configurations for the considered use case have been defined:

- **no-SIMD:** in this configuration, SIMD feature of the accelerator is not used, but all the computations are made in a conventional form (i.e. one register → one value);
- **SIMD:** in this configuration, some of the defined banks are managed with no-SIMD computations (i.e. one register → one value), while other ones with SIMD computations enabled (i.e. one register → two values).
  The banks for which SIMD is enabled have been defined according to the features of each filter. E.g. the SIMD is particularly useful in the case of I/Q components that some filters have to manage.

The RTL pre-synthesis parameters (see *section 4.2*) set for the no-SIMD and SIMD configurations are shown respectively in *Table 6.1* and *Table 6.2*.

| **Parameter name** | **Value** |
|---|---|
| N_BANKS_MAC | 8 |
| N_ELEMENTS_BANK_MAC | [2, 2, 34, 34, 4, 4, 34, 2] |
| N_ELEMENTS_BANK_MAX_MAC | 34 |
| WIDTH_DATA_MAC | [16, 16, 16, 16, 16, 16, 21, 23] |
| WIDTH_DATA_NOSIMD_MAX_MAC | 23 |
| WIDTH_DATA_SIMD_MAX_MAC | (not relevant, since SIMD_MAC=0) |
| WIDTH_DATA_MAX_MAC | 23 |
| WIDTH_COEFFS_MAC | [8, 8, 8, 8, 8, 8, 8, 8] |
| WIDTH_COEFFS_NOSIMD_MAX_MAC | 8 |
| WIDTH_COEFFS_SIMD_MAX_MAC | (not relevant, since SIMD_MAC=0) |
| WIDTH_COEFFS_MAX_MAC | 8 |
| SIMD_MAC | 0 |
| N_BIT_NFEED_MAC | 2 |
| FEEDBACK_UNROUNDED_MAC | 1 |
| N_OPERANDS_MAC | 5 or 7 or 12 or 17 or 34 *(synthesis variable)* |
| BITS_CUT_MAX_MAC | 8 |

**Table 6.1:** *no-SIMD configuration RTL parameters.*

| Parameter name | Value |
|---|---|
| N_BANKS_MAC | 6 |
| N_ELEMENTS_BANK_MAC | [2, 34, 4, 4, 34, 2] |
| N_ELEMENTS_BANK_MAX_MAC | 34 |
| WIDTH_DATA_MAC | [32, 32, 16, 16, 21, 23] |
| WIDTH_DATA_NOSIMD_MAX_MAC | 23 |
| WIDTH_DATA_SIMD_MAX_MAC | 32 |
| WIDTH_DATA_MAX_MAC | 32 |
| WIDTH_COEFFS_MAC | [16, 16, 8, 8, 8, 8] |
| WIDTH_COEFFS_NOSIMD_MAX_MAC | 8 |
| WIDTH_COEFFS_SIMD_MAX_MAC | 16 |
| WIDTH_COEFFS_MAX_MAC | 16 |
| SIMD_MAC | 1 |
| N_BIT_NFEED_MAC | 2 |
| FEEDBACK_UNROUNDED_MAC | 1 |
| N_OPERANDS_MAC | 5 or 7 or 12 or 17 or 34 *(synthesis variable)* |
| BITS_CUT_MAX_MAC | 8 |

**Table 6.2:** *SIMD configuration RTL parameters.*

## 6.2 Software profiling

In this section, software profiling is reported. Unfortunately, due to NXP's IP, the software details can not be shown. The only allowed parts are explained in *chapter 3* and *chapter 4*.

### 6.2.1 Codesize

The profiling is needed to check how much of the instruction/data memory is needed for the code and all the related elements.

The instruction memory is read-only for the processor, while the data memory is read/write.

For our code, there are four parts to be considered:

- **.text:** it is the section containing the code (that is converted in ASM instructions and then in binary by the compiler). It is stored into the instruction memory.
- **.data:** it is the section where the initialized variables are stored. It is stored partially into the instruction memory (e.g. the initialization values) and partially into the data memory (e.g. the actual value of the variables that can be modified in the code).
- **.rodata:** it is the section containing the constant data (read-only). It is stored into the instruction memory.
- **.bss:** it is the section dedicated to storing the uninitialized variables.

The size obtained for each of the four parts described above is shown in *Figure 6.1* and they are reported in `byte`. The considered case is the **no-SIMD** version of the algorithm, but the profiling differences with the **SIMD** one are negligible.

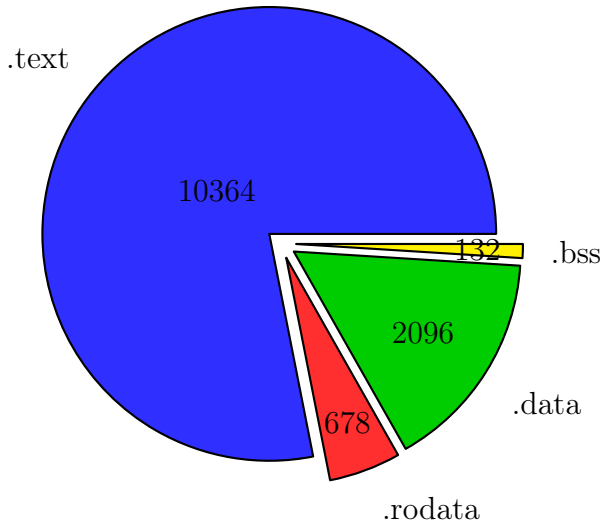



**Figure 6.1:** *Codesize (values are in `byte`).*

### 6.2.2 Obtained results

The expected results with the base RISC-V ISA (i.e. without the accelerator) are shown in *subsubsection 3.1.3.2*. The outcome is that **152 instructions** are needed per sample, that are far away from the computed upper limit that is **79 clock cycles** per sample (assuming $f_{clock} = 540\,\text{MHz}$).

With the accelerator, the ideal analysis has been made in *subsubsection 3.3.4.2* and we obtain that **29 to 45** instructions per sample are needed.

Of course, the relation (1 instruction $\Longleftrightarrow$ 1 clock cycle) is not always valid due to some non-idealities, such as pipe stalls and core-coprocessor interfacing. For this reason, an RTL simulation is needed to understand how much these aspects impact the performance.

Unfortunately, also here it is not possible to go deeper into the algorithm due to NXP's IP, but only the final results can be given.

The following results are obtained simulating through the OpenHW verification environment (see *section 2.6*) extended with some custom functionalities (see *section 5.1*).

The code has been written both without and with SIMD support and the hardware settings are reported in *Table 6.1* and *Table 6.2*.
The parameter `N_OPERANDS_MAC=34` has been set to get the maximum performance.

Simulating, the outcome is:

- **no-SIMD:** without using SIMD, the result obtained is that **161 clock cycles** are needed for computing **two** samples. This means that the processor should run at 546 MHz to be real-time capable.
- **SIMD:** using SIMD, the result obtained is that **155 clock cycles** are needed for computing **two** samples. This means that the processor should run at 526 MHz to be real-time capable.

### 6.2.3 Considerations on the results

The results shown above are promising, but they are not yet valid for a real use case.

The project aims to integrate the core+coprocessor into the existing system that now exploits custom hardware. This means that the core+coprocessor has to interface with other devices, sharing buses and memories.

Looking at the memories, in the OpenHW verification environment, they are emulated through a model that makes their behaviour not completely ideal (e.g. some clock cycles are needed to access the data), giving a little more credibility to our results. However, it is not enough and a deeper analysis is needed.
It is a not straightforward aspect to face since it means analyzing the existing NXP system, modelling it somehow, understanding how and where our CPU should be placed and then proceeding with the integration and the final results. Of course, this is out-of-scope for this thesis work.

Therefore, the outcome is that the performance has been greatly improved ($\sim$**3x**), but the system is far away to be ready for integration into the existing system.

## 6.3 Hardware profiling

### 6.3.1 Synthesis details

#### Tools and technological libraries

The synthesis has been performed using *Cadence Genus* as synthesizer.

The technological library used is a 28 nm process by TSMC containing a great variety of cells, with different features.
Besides the strength (i.e. fan-out capability), three different threshold voltages are available: standard $V_T$ ($SVT$), high $V_T$ ($HVT$), ultra high $V_T$ ($UHVT$), to better optimize the netlist from a power perspective. A low threshold voltage is beneficial for cell performance, but it has dramatic effects on the leakage power.
The synthesizer is in charge of choosing the most suitable cell for each path.

#### Some notes and limits about the synthesis

The synthesis performed is just a preliminary one, since a more accurate back-end is needed to have reliable and fully meaningful results.

The main limitations are:

- **no Place and Route (P&R):** without the P&R, the obtained absolute values are not meaningful and only relative ones could be used.

- **no memories:** memories are not instantiated for the synthesis, since an analysis has not been performed on their requirements, yet. Moreover, work is needed to integrate them to work with the OpenHW group OBI.

- **clock gating:** in the OpenHW group's core, clock gating is used to save power. In the current implementation, the clock gating cell is an RTL module in SystemVerilog containing a latch and an `AND` port. A better approach could be the usage of dedicated clock gating cells that are available in the used technological libraries, but it is a task needing further investigation.
  In the end, it is not a big deal, since the clock gating implemented by OpenHW simply turns off the entire core when it is not used, being not fine-grained. In other words, since in our software we always use the core, the clock gating is always disabled.

- **no switching activities back-annotation:** to have a first rough power estimation, the netlist generated by *Cadence Genus* should be simulated (Gate-Level Simulation (GLS)) with a representative testcase.
  Unfortunately, the GLS requires a deep modification of the testbench, since the signal naming changes passing from the RTL to the synthesized netlist, representing a time-consuming task.
  For this reason, no GLS has been performed.

The **area** will be reported in *Gate Equivalent (GE)* that is a technology-independent measure of area, in which the values are normalized to the area of an average logic port, usually the two-inputs strength-one NAND (`NAND2_X1`). Mathematically:

$$GE_{circuit} = \frac{A_{circuit}}{A_{\texttt{NAND2\_X1}}}$$

The **power**, instead, will be **normalized to the core-only power** to have a relative estimation and not absolute values that would be meaningless for the reasons explained above.

The only aspect to be taken into account is that the modules hierarchy is maintained unflatten for some components to have a better profiling insight. If we leave to the synthesizer the possibility to make the whole design flat, the boundaries of the different modules would be lost and all the elements would be treated as "nameless" logic gates. If the synthesizer flats all, one can not have information, for instance, on the arithmetic part of the MAC or on the registers, but only on the total design.

Three sub-designs are defined (i.e. their boundaries are maintained):

1. core;
2. coprocessor registers (*data/coeffs/configuration*);
3. coprocessor arithmetic part (e.g. MAC arithmetic, rounding).

This means that the boundaries of these parts of the design are not fully optimized by the synthesizer, but in exchange we can get profiling information on those groups. For the sake of curiosity, a synthesis with all the design flatten has been performed as well and the obtained differences are negligible.

### Clock frequencies and timing

The chosen clock frequencies are $f_{clock} = 356\,\text{MHz}$ and $f_{clock} = 414\,\text{MHz}$. These two frequencies were identified as suitable for real-time processing in a preliminary stage of the software profiling.

Unfortunately, these values have slightly changed with the last profiling, resulting in 526 MHz and 546 MHz (see *subsection 6.2.2*). However, a new synthesis from scratch has been considered not worthwhile and 356 MHz and 414 MHz have been maintained.

No timing information will be shown in the next plots since the stated clock frequency is always met. Moreover, the critical path does not change among the different tests, but it is always represented by the MAC arithmetic part (MAC + rounding + MUXes).

### 6.3.2 Synthesis results

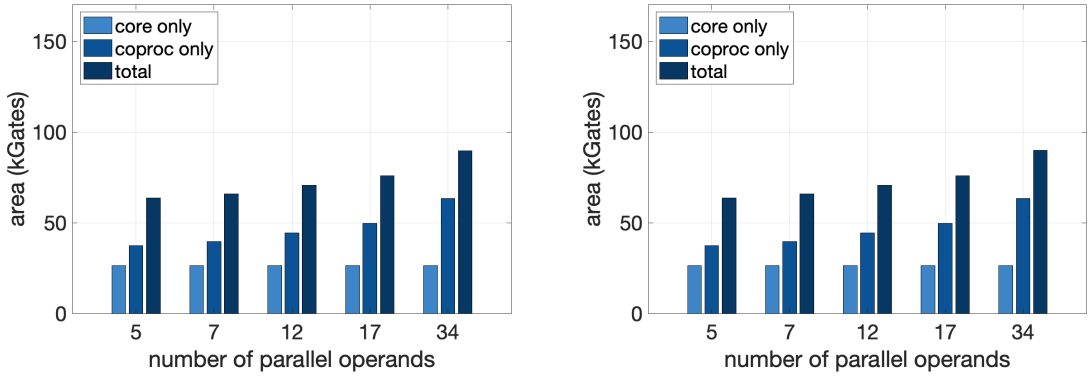
#### 6.3.2.1 Overall results

In this section, the overall results are shown. Area (*blue* bars) and normalized power (*red* bars) are plotted both for no-SIMD (*Figure 6.2*, *Figure 6.4*) and for SIMD configuration (*Figure 6.3*, *Figure 6.5*).

On the x-axis, the number of operands that the MAC arithmetic unit is able to manage in parallel (see *section B.3*) is reported, while on the y-axis the represented quantity.
For each of the x-axis values, three bars are shown (respectively *core only*, *coprocessor only*, *core+coprocessor*), stating which part of the system the quantity is referred to. Each plot is made for two different clock frequencies (see *Clock frequencies and timing*).

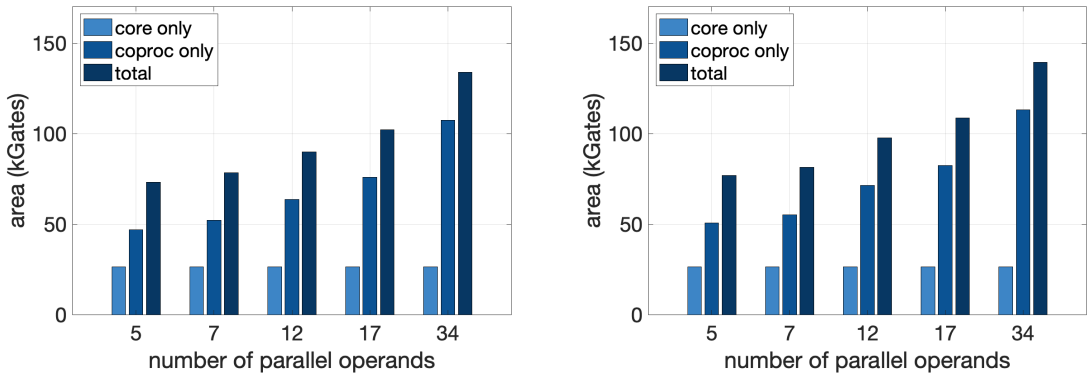
At glance, it can be seen that the core-only area and power are lower than the coprocessor ones. This is due on one hand to the intrinsically expensive structure of the coprocessor (e.g. MAC arithmetic and the registers to store all the buffered values). In addition, the core ISA extensions (e.g. multiplication/division) have been disabled. The control part of the core is much more complex than the coprocessor one, but usually, the control part is less expensive than the arithmetic/memory one.

As for the **area**, the trend is as expected: the higher the number of parallel operands to be managed, the higher the complexity. Of course, the core-only area remains unchanged varying the number of parallel operands, since it is a parameter impacting the coprocessor only.
Reporting some numbers: the area of the core-only is about 26 kGates, while the coprocessor-only area for the no-SIMD configuration spans from 38 kGates to 62 kGates, according to the number of parallel operands.
As expected, the SIMD implementation has a larger area than the no-SIMD one, but the trend is exactly the same.
The entire system (core+coprocessor), considering no-SIMD and SIMD implementations, the different number of parallel operands and the different clock frequencies, can have an area from 62 kGates to 140 kGates.




**(a)** $f_{clock} = 356\,\text{MHz}$. **(b)** $f_{clock} = 414\,\text{MHz}$.

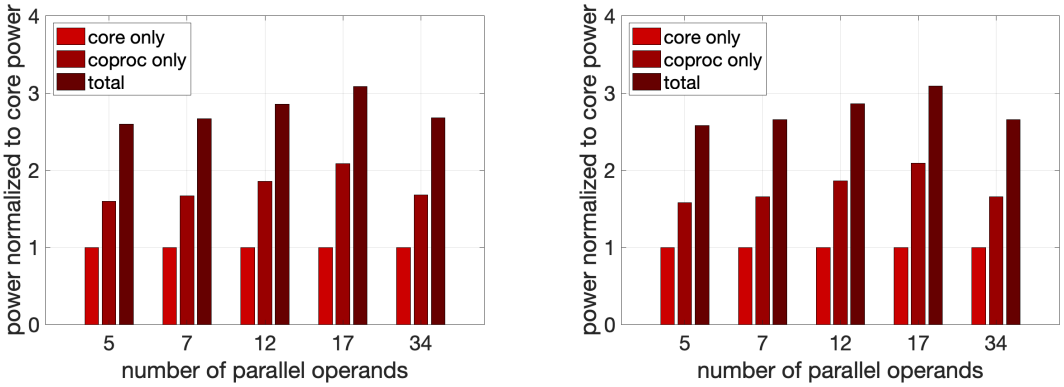
**Figure 6.2:** *Area profiling results for no-SIMD configuration.*




**(a)** $f_{clock} = 356\,\text{MHz}$. **(b)** $f_{clock} = 414\,\text{MHz}$.

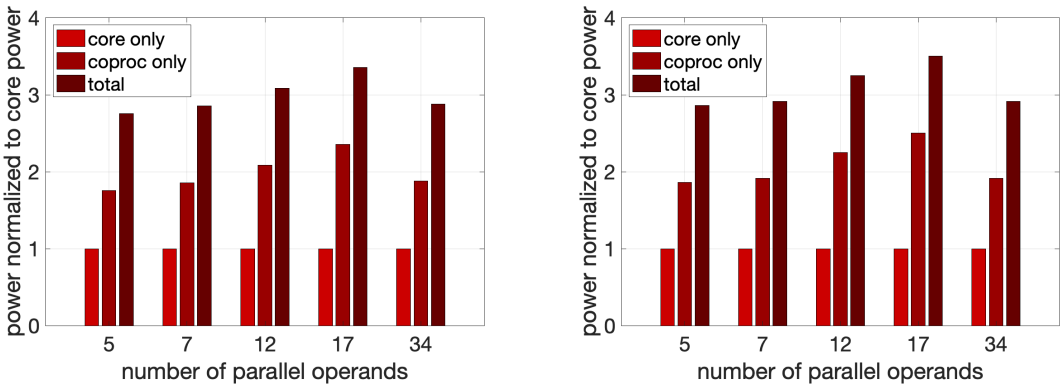
**Figure 6.3:** *Area profiling results for SIMD configuration.*

About the **normalized power**, the trend is the same as the area, but with one exception: passing from 17 parallel operands to 34, the power lowers. This is not so intuitive, but it is due to the fact the hardware overhead needed for an iterative structure (in the cases in which `N_ELEMENTS_BANK_MAX_MAC > N_OPERANDS_MAC`) consumes more power than having a bigger MAC arithmetic unit and no iterations (`N_ELEMENTS_BANK_MAX_MAC = N_OPERANDS_MAC = 34`). See *section B.3* for more information.
Also in this case, SIMD implementation has an higher power consumption than the no-SIMD one.
As for the area, the coprocessor consumes from 1.5 to 2.5 times more than the core.




**(a)** $f_{clock} = 356\,\text{MHz}$. **(b)** $f_{clock} = 414\,\text{MHz}$.

**Figure 6.4:** *Power profiling results for no-SIMD configuration.*



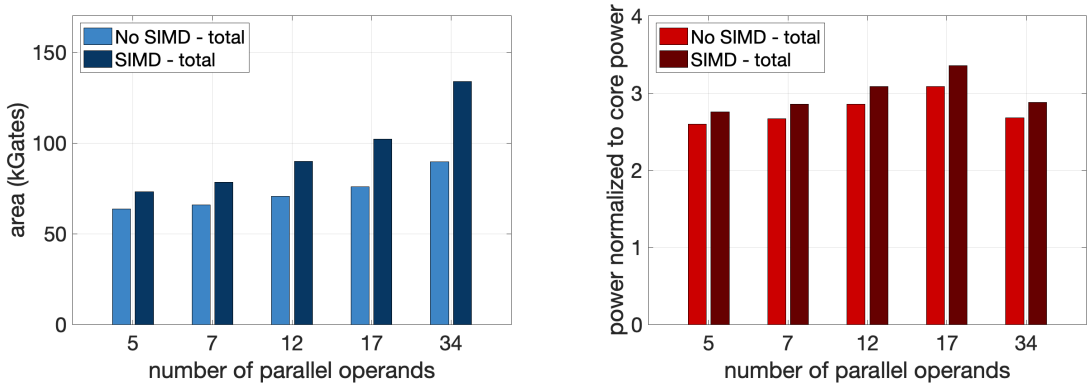

**(a)** $f_{clock} = 356\,\text{MHz}$. **(b)** $f_{clock} = 414\,\text{MHz}$.

**Figure 6.5:** *Power profiling results for SIMD configuration.*

#### 6.3.2.2 No-SIMD vs SIMD comparison ($f_{clock} = 356\,\text{MHz}$)

In *Figure 6.6*, no-SIMD vs SIMD comparison is shown. They are the same data as the plots above (*Figure 6.2*, *Figure 6.3*, *Figure 6.4*, *Figure 6.5*), but arranged to highlight the differences between the two implementations.
Only values for $f_{clock} = 356\,\text{MHz}$ are reported, since for the other frequencies the results are pretty similar and it is not worth showing all of them.

As expected, the SIMD implementation has higher area and power. Considering the **area**, it is 20% higher for 5 parallel operands up to 47% for 34 parallel operands. The **power** increment is more reasonable, being around 6% more for all the parallel operands.




**(a)** *Area profiling comparison between no-SIMD and SIMD configurations* $f_{clock} = 356\,\text{MHz}$.

**(b)** *Power profiling comparison between no-SIMD and SIMD configurations* $f_{clock} = 356\,\text{MHz}$.

**Figure 6.6:** *no-SIMD vs SIMD comparison.*

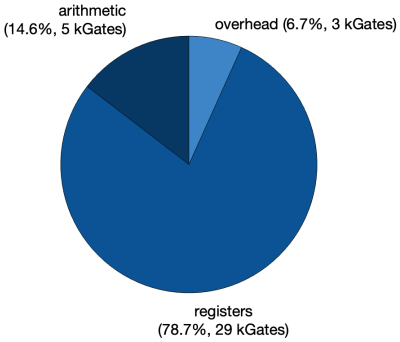
#### 6.3.2.3 Coprocessor insight ($f_{clock} = 356\,\text{MHz}$)

It is also useful to check how the coprocessor area and power are partitioned among *arithmetic*, *registers* and *overhead*.
The *overhead* is made by the coprocessor parts not belonging neither to the arithmetic or the registers. It is mainly represented by the control logic (e.g. FSMs) and by the iterative structures needed if the number of parallel operands that the MAC can manage is lower than needed (see *section B.3*).

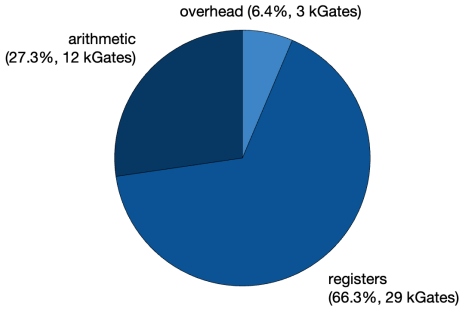
The partitioned area and power results are shown in *Figure 6.7*, *Figure 6.8*, *Figure 6.9*, *Figure 6.10*.

About the **area**, looking at the *registers*, it can be noticed that it is unchanged among the different implementations, being around 30 kGates.
The same can not be said for the *arithmetic* part, which spans from 5 kGates to 75 kGates according to the chosen implementation.
The most interesting part is probably the *overhead*, which is around 3 to 4 kGates for the iterative implementations (number of parallel operands lower than 34), falling to only 1 kGates for the non-iterative implementations (number of parallel operands equal to 34). This means that the iterative structures are more expensive than the whole control logic.

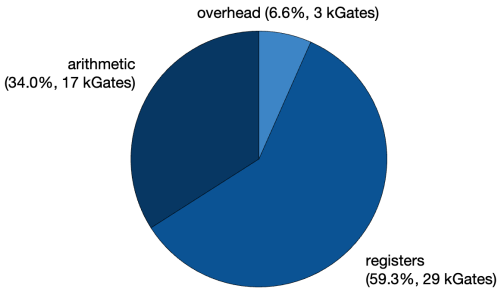
The identical trend applies to the **power**.



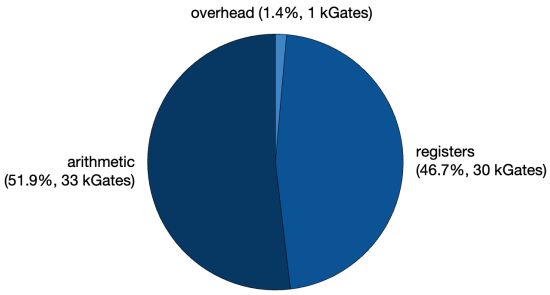

**(a)** *5 parallel operands.*




**(b)** *12 parallel operands.*

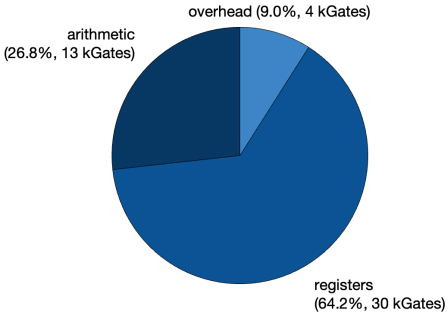



**(c)** *17 parallel operands.*

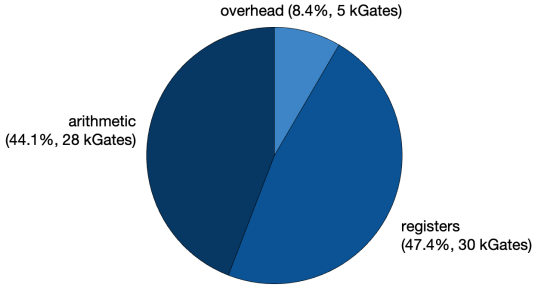



**(d)** *34 parallel operands.*

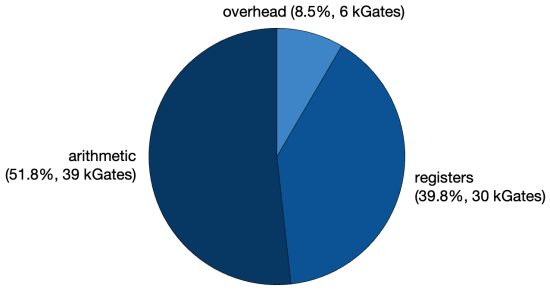
**Figure 6.7:** *Area profiling: coprocessor insight for no-SIMD configuration ($f_{clock} = 356\,\mathrm{MHz}$).*



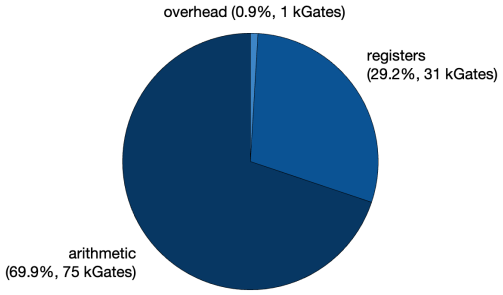

**(a)** *5 parallel operands.*




**(b)** *12 parallel operands.*

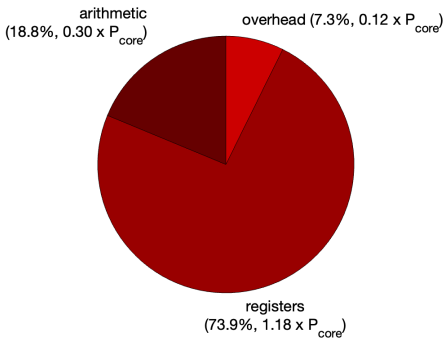



**(c)** *17 parallel operands.*

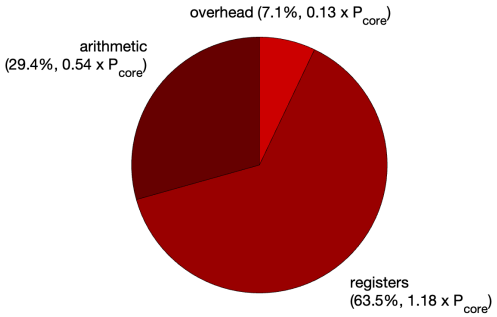



**(d)** *34 parallel operands.*

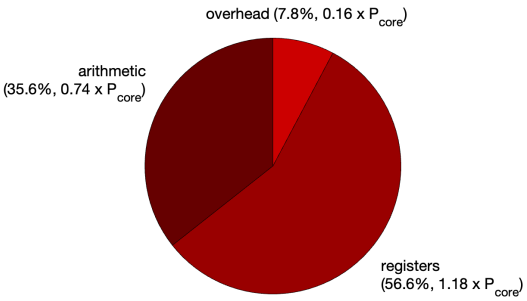
**Figure 6.8:** *Area profiling: coprocessor insight for SIMD configuration ($f_{clock} = 356\,\mathrm{MHz}$).*



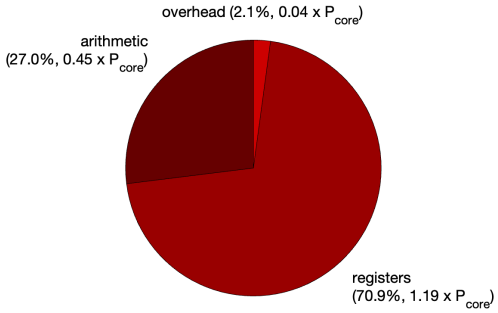

**(a)** *5 parallel operands.*




**(b)** *12 parallel operands.*

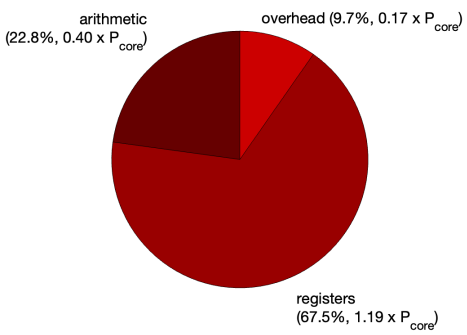



**(c)** *17 parallel operands.*

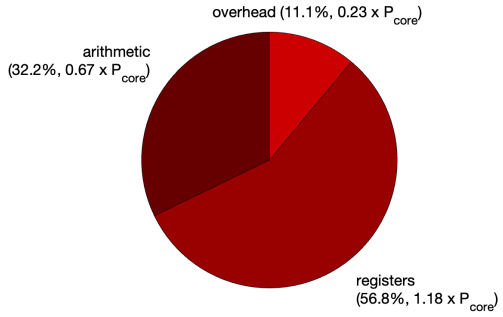



**(d)** *34 parallel operands.*

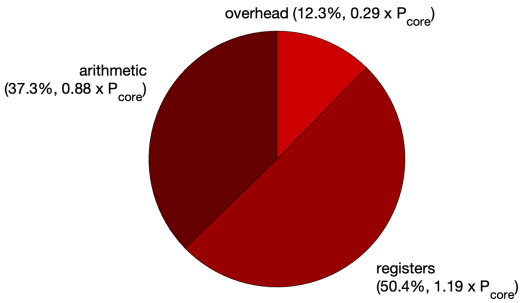
**Figure 6.9:** *Power profiling: coprocessor insight for no-SIMD configuration ($f_{clock} = 356\,\mathrm{MHz}$).*



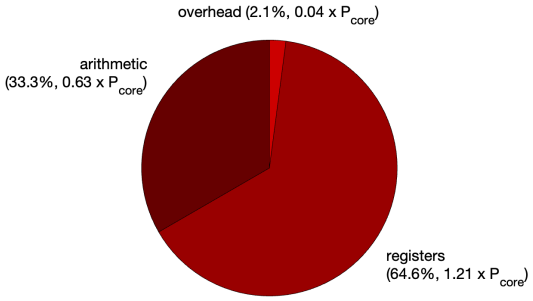

**(a)** *5 parallel operands.*




**(b)** *12 parallel operands.*




**(c)** *17 parallel operands.*




**(d)** *34 parallel operands.*

**Figure 6.10:** *Area profiling: coprocessor insight for SIMD configuration ($f_{clock} = 356\,\mathrm{MHz}$).*

### 6.3.3 Synthesis optimization: registers without asynchronous reset

It can be possible to slightly optimize the synthesis exploiting registers without asynchronous reset.

Usually, flip-flops have asynchronous reset to initialize them to a determined value (usually `'0'`) during the initial reset phase. Having asynchronous reset complicates the microelectronic structure of the flip-flop in a not negligible way, therefore, in some cases, flip-flops without asynchronous reset could be employed.
If the asynchronous reset is not present, the bit stored by the flip-flop at the beginning of the system operation is not defined, but can be `'0'` or `'1'` with the same probability.
Notice that the lack of an asynchronous reset does not imply the absence of a synchronous one, which introduces a lower complexity.

The reset is strictly needed in some cases, such as in the status registers of the FSMs,

in which the value stored into the registers right after the start-up must be defined for the correct operation of the system.

Looking at the developed accelerator, having a determined value at the start-up is not strictly needed for the *data/coeffs/configuration regs*, while it is mandatory for the FSMs.

One of the few disadvantages of this approach is that the programmer can not do assumptions about the value stored into those registers at the beginning of the operation, therefore one has to initialize all the registers to ensure that a defined value is stored. This applies only for the initial operations, while during normal working no differences are present. In addition, the *data/coeffs regs* have a synchronous reset which can be easily triggered through `CLEAR_DATA` and `CLEAR_COEFFS` instructions.

Another drawback is related to module verification. It has to be developed taking into account the undefined initial values of those registers.

#### 6.3.3.1 Results

Of course, almost all the data reported in the previous pages are valid in this case, since they are not changed by this optimization. Some examples are the arithmetic and the control parts, which do not change at all. The *data/coeffs/configuration regs* are the only ones to take advantage of this optimization, being now without the asynchronous reset.

Results are shown in *Table 6.3*.
The reduction on the **area** represents more than 11% of the total registers area and it could be not negligible.
The **power** is normalized to the power of the core-only, as done for all the previous plots, and the obtained improvement can be considered negligible.

| | **Reduction (absolute)** | **Reduction (percentage)*** |
|---|---|---|
| **Area** | ∼3.3 kGates | ∼11.3% |
| **Normalized power** | ∼0.0424 | ∼3.6% |

**Percentages referred to the registers-only area/power before optimization.*

**Table 6.3:** *Improvements using registers without asynchronous reset.*

Looking at the obtained benefit on the entire system, it depends on the ratio between arithmetic/registers.
Two intermediate cases can be considered as representative:

- **No-SIMD, 12 parallel operands:**
  - **Area:** 4.7% less.
  - **Power:** 1.2% less.
- **SIMD, 12 parallel operands:**
  - **Area:** 3.4% less.
  - **Power:** 0.93% less.

Of course, the impact is slightly large on the configurations in which the arithmetic part is smaller and slightly lower when it is larger.

In the end, the obtained improvement on the entire system is not so significant, but since it can be obtained almost for free, it could be worthwhile.

## 6.4 Results summary

In *Table 6.4*, the results are summarized.

The reported results are related to a clock frequency $f_{clock} = 354\,\text{MHz}$.
The coprocessor settings are shown in *Table 6.1* and *Table 6.2*, for the no-SIMD and SIMD versions respectively.
The parameter `N_OPERANDS_MAC=34` has been set.

| | **Instructions per sample** | **Area** | **Normalized power** |
|---|---|---|---|
| **Without accelerator** | 152 | 26 kGates | 1 |
| **With accelerator (no-SIMD)** | 45 | 90 kGates | 2.7 |
| **With accelerator (SIMD)** | 41 | 135 kGates | 2.9 |

**Table 6.4:** *Results summary.*

# Conclusion

This thesis works aims to start analyzing the feasibility of employing a **Software-Defined Radio (SDR)** based on a microprocessor for NFC signal processing. This allows for **enhanced flexibility** with respect to a custom logic implementation, at the cost of potentially **higher resource usage** to meet the requirements. The hardware platform identified as suitable for the purpose has been an open-source RISC-V based CPU by the OpenHW group.
After a preliminary analysis, the **core alone** proved **not suitable** to meet the strict requirements of the considered application. To improve the performance, an **accelerator** has been developed and tested to face the main bottleneck identified, namely **digital filtering**.

The results are encouraging. Exploiting the accelerator, the number of instructions per sample has passed from 152 to 41. The estimations show that the performance is **close to real-time processing**.
As for the hardware profiling, the **area** occupied by the core-only is 26 kGates. The total estimated area (core+accelerator) spans from 62 kGates to 140 kGates according to the chosen settings for the coprocessor. The **power** has been estimated only normalized to the power of the core, spanning between $(2.5 \cdot P_{core})$ and $(3.5 \cdot P_{core})$.

In the current implementation, coprocessor parallelization and pipelining have been denied due to the great complexity introduced. In future work, the performance could be further improved by exploiting them.

Another aspect to be faced in the future is the reliability of the results. They could not be accurate for two main reasons:

- As for the **performance**, the system should be better modelled to have a good estimation. The simulations done within this thesis have been obtained with the core+coprocessor as the only actor in the system. In other words, all the resources (e.g. memories) are always available, but this assumption is not valid for more complex systems.
  The core+coprocessor is meant to be integrated into the existing NFC system by NXP, where other devices are present. This aspect shall be taken into great account since it can degrade significantly the performance.

- As for the **area** and **power**, the back-end flow should be completed (i.e. P&R) to get meaningful estimations.

This thesis represents a starting point for the more ambitious project that NXP is meant to carry on, but it is certainly not enough and further investigations will be needed.

# Appendices

# Appendix A

# RFID Physical Principles

In the following pages, the physical principles of **inductively remote-coupled** systems are explained.

## A.1 Mutual inductance and coupling coefficient

$\Phi$ is the magnetic flux through a single conductor loop with area $A$ generated by a magnetic field $B$. Maintaining the same conditions, but placing $N$ identical conductor loops in series, the corresponding magnetic flux $\Psi$ is:

$$\Psi = N\,\Phi = N\overrightarrow{B}\cdot\overrightarrow{A} \tag{A.1}$$

The *inductance* $L$ is the most important parameter of a coil and it is defined as:

$$L = \frac{\Psi}{I} \tag{A.2}$$

where $\Psi$ is the interlinked flux generated by the current $I$ that arises in an area enclosed by that current.

Assuming to have a first coil through which a current $I_1$ is flowing, if a second coil is present and if it is close enough to the first one, it will be subject to a proportion of the total magnetic flux $\Psi$, namely $\psi_{21}$. The magnitude of $\psi_{21}$ depends on the geometry of the two coils, the relative position and the magnetic properties of the medium.
Similarly to the inductance $L$, *mutual inductance* $M_{21}$ can be defined:

$$M_{21} = \frac{\psi_{21}(I_1)}{I_1} \tag{A.3}$$

The same can be said for the symmetric case in which the second coil influences the first one. The following relation applies:

$$M = M_{12} = M_{21} \tag{A.4}$$

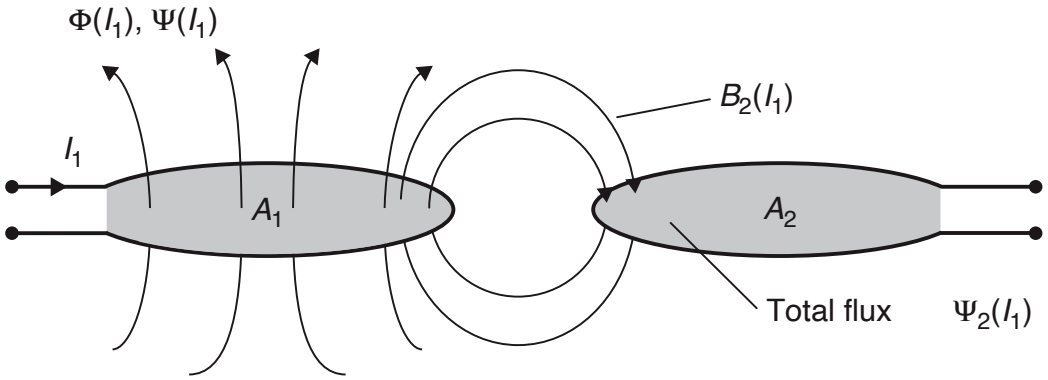



**Figure A.1:** *Graphical representation of the mutual inductance concept. Taken from [2].*

A normalized coefficient $k$ related to the mutual inductance $M$ can be defined and it is called **coupling coefficient**:

$$k = \frac{M}{\sqrt{L_1 L_2}} \in [0, 1] \tag{A.5}$$

Taking as reference the two boundaries, we get:

- $k = 0$: the two coils are completely decoupled. No effects occur in the first coil due to the second and vice-versa.
- $k = 1$: full coupling. The two coils are subject to the same magnetic flux $\Psi$. Transformers are examples of systems in which $k \approx 1$.

## A.2 Faraday's law, resonance and power supply

Considering the two coils $L_1$ and $L_2$ described before, if the magnetic flux $\Psi$ in $L_1$ changes, a voltage is induced into $L_1$ itself (self-inductance) and in $L_2$ (mutual inductance). This effect is explained by the *Faraday's law* (*Equation A.6*).

If the second coil is an open circuit, the voltage induced is:

$$u_{i2} = \frac{d\Psi_{21}(t)}{dt} = M \frac{di_1(t)}{dt} \tag{A.6}$$

The second coil is our transponder and it is not an open circuit since it drains current. An equivalent circuit (even though heavily simplified) is shown in *Figure A.2*. The two coils are represented and $R_2$ is the intrinsic resistance associated with $L_2$ that is not an ideal inductor.

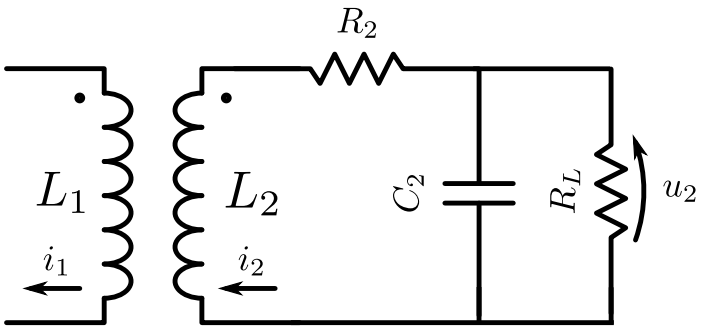



**Figure A.2:** *Equivalent circuit for magnetically coupled conductor loops.*

Writing the voltage Kirchhoff law and considering sinusoidal signals, we get that the voltage across the resistive load, which represents the chip of the transponder, is:

$$u_2(\omega) = u_{i2}(\omega)\frac{\left[\frac{1}{j\omega C_2} \parallel R_L\right]}{\left[\frac{1}{j\omega C_2} \parallel R_L\right] + j\omega L_2 + R_2} = j\omega M\, i_1(\omega)\frac{1}{1 + (j\omega L_2 + R_2)\left(\frac{1}{R_L} + j\omega C_2\right)} \tag{A.7}$$

If $L_2$ and $C_2$ are designed to be in resonance at the operating frequency of the system $\omega_0^2 = \dfrac{1}{L_2 C_2}$, $u_2$ reaches very high voltages allowing to have communications also with low magnetic fields (i.e. large distance between the two coils).

In passive transponders, where no external battery is present, the power supply for the chip is provided by the reader field. As said, the voltage can reach very high voltages through resonance (even much greater than 100 V if the coupling coefficient is high), therefore a voltage regulation is needed since the chip usually works with few volts. Shunt regulators can be used, as shown in the principle scheme in *Figure A.3*, that are designed to tune the equivalent resistance seen by the LC group according to the voltage level.

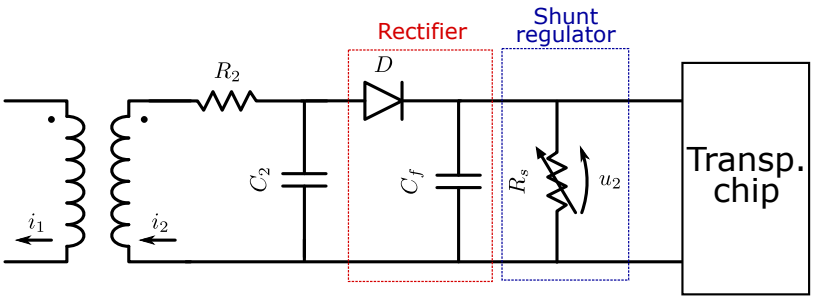



**Figure A.3:** *Operating principle for power supply in the transponder.*

## A.3 Load modulation

As the transponder (coil 2) is influenced by the reader (coil 1), the vice-versa is also true, since the mutual inductance $M$ and, consequently, the coupling coefficient $k$ is symmetric. The logical cause-effect chain is: $i_1$ varies $\rightarrow$ $\Psi_1$ varies $\rightarrow$ $\Psi_2$ varies $\rightarrow$ $u_2$ varies $\rightarrow$ $i_2$ varies $\rightarrow$ $\Psi_2$ varies $\rightarrow$ $\Psi_1$ varies $\rightarrow$ $u_{meas}$ varies $\rightarrow$ $i_1$ varies $\rightarrow$ ...

Of course, the system will reach a steady state condition, but it is important to highlight that the presence of the second circuit (transponder) represents a feedback for the first one (reader).
The equivalent circuit from the reader's point of view is shown in *Figure A.4*.

The resistor $R_1$ is the intrinsic resistance associated with the coil due to non-idealities. The capacitor $C_1$ is chosen in such a way to be a series resonator with $L_1$ at the operating frequency of the system $\left(\omega_0^2 = \frac{1}{L_1 C_1}\right)$, giving $Z_{res} = \frac{1}{j\omega_0 C_1} + j\omega_0 L_1 = 0$.

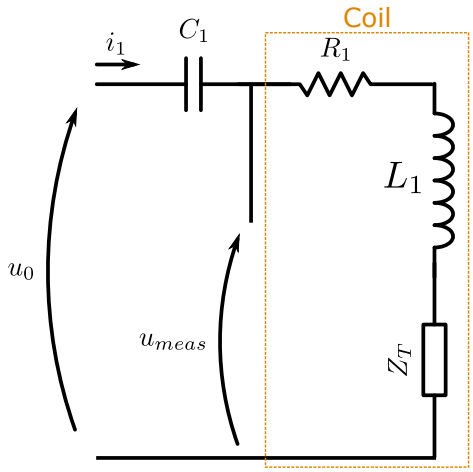



**Figure A.4:** *Equivalent circuit at the reader side.*

The effect of the feedback can be modelled by introducing the impedance $Z_T$, which is called *complex transformed transponder impedance.*

It can be shown that the expression of $Z_T$ is:

$$Z_T = \frac{\omega^2 M^2}{R_2 + j\omega L_2 + \dfrac{R_L}{1 + j\omega R_L C_2}} = \frac{\omega^2 k^2 L_1 L_2}{R_2 + j\omega L_2 + \dfrac{R_L}{1 + j\omega R_L C_2}} \tag{A.8}$$

where the parameters related to the transponder are the same reported in *Figure A.2*.

$Z_T$ depends on plenty of parameters, but the very interesting dependency is $R_L$, which is the equivalent resistance of the transponder chip that can be changed at run-time. The transponder can send data to the reader simply varying $R_L$ that will influence

$Z_T$ and finally $u_{meas}$. This technique is called *load modulation* and it is extensively used by passive transponders to send data to the reader.

Notice that the reader can not directly access $Z_T$ since it is a "virtual" impedance given by the Faraday's law voltage, but the antenna terminals ($R_1$ and $L_1$ represent the reader's antenna) can be accessed and the voltage variation can be detected.

In *Figure A.5*, the complete system with load modulation data transmission is shown. It is a principle scheme since the analog front-end is more complex than this, but all the significant elements so far discussed are present. The load can be modulated through a FET whose gate is controlled by the transponder chip with a suitable modulating signal. That signal is binary code based (i.e. two levels alternating signal), but its characteristics depend on the used protocol (refer to *subsection 1.3.1*).

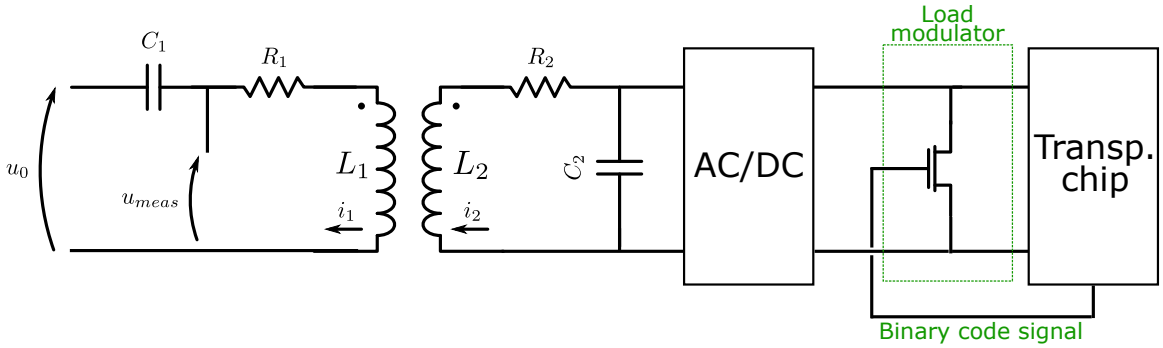



**Figure A.5:** *Complete basic reader+transponder system.*

# Appendix B

# Architectural details

This section is a detailed dissertation on the design choices and techniques used during the development of the MAC accelerator. These details are needed only for those who want to understand the RTL structure of the accelerator, for debugging purposes or just for curiosity. It is not strictly needed for those who want only to use it as a programmer, since the specifications explained in *chapter 4* should be enough.

**Notice** that you should read first the section dedicated to the high-level specifications (*chapter 4*) since in the following sections some of the concepts explained there will be not repeated.
In addition, all the specifications related to the eXtension InterFace (XIF) (e.g. handshaking, signal naming and meaning) will be not explained here, but one can refer to [14].

## B.1 Coprocessor architecture

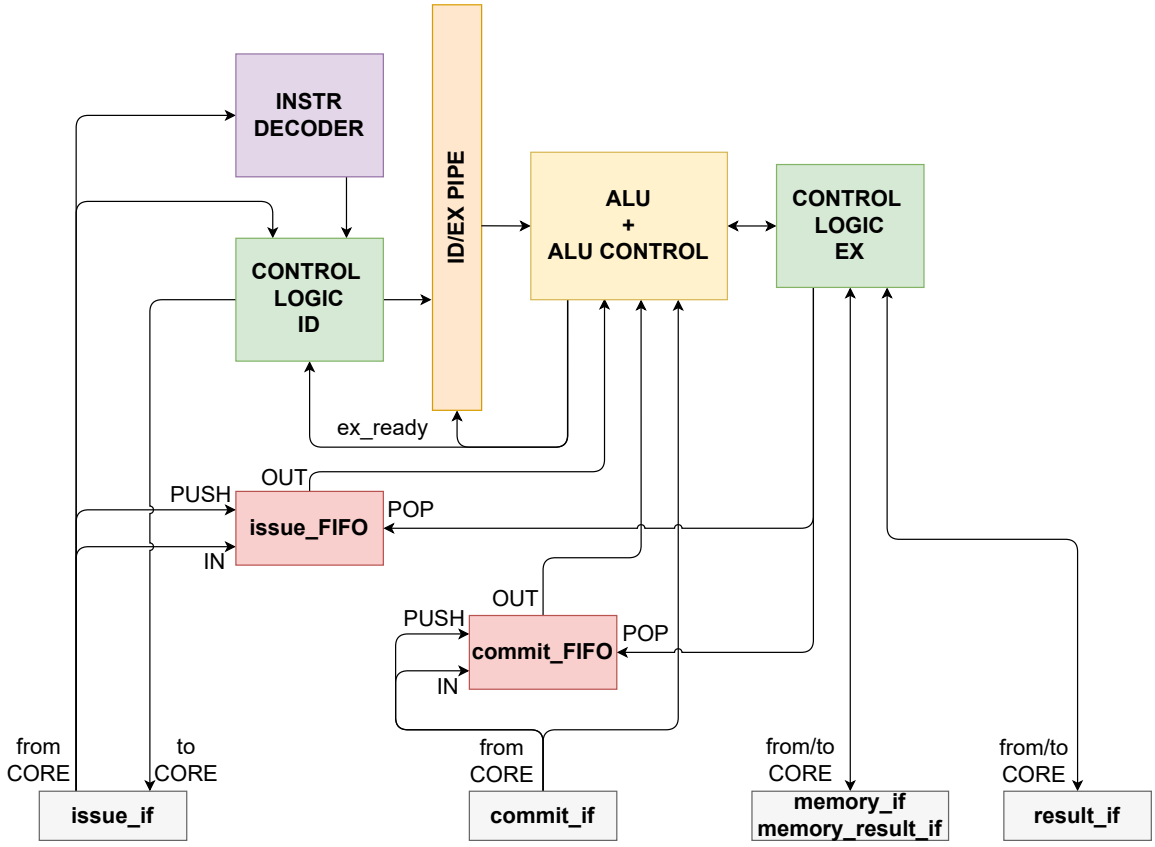



**Figure B.1:** *Coprocessor architecture.*

The high-level architecture of the coprocessor is shown in *Figure B.1.*
It is a 2-stage coprocessor, that is dependent on the core for the instruction offloading, committing, data memory access and result storing (refer to *subsection 2.4.1*).

### B.1.1 Parallelization and pipelining

The core-coprocessor couple behaves as a producer-consumer system, with the core providing instructions to the coprocessor and the coprocessor executing them.

In this stage of the coprocessor development, some simplifications have been made to make the development and testing feasible in a reasonable amount of time. The main one is letting the coprocessor execute **only one instruction at a time and denying pipelining**.
This means that no parallelization has been implemented, using an in-order execution with only one instruction at a time, avoiding more complex approaches (e.g. superscalar-like).
As for pipelining, it is also denied in this phase, since the hazard management would

have been complex. For instance, we have read/write from/to memory and RF, commit/kill signals and possibly different blocks into the coprocessor ALU, which are aspects not easily manageable.
Parallelization and pipelining could be a good starting point for future work.

### B.1.2 Instruction Decoder

It is represented in *purple* in *Figure B.1* and it is in charge of decoding the instructions coming from the *issue interface*, signalling to the *ID-stage Control Logic* if an offloaded instruction is recognized and can be executed or if it is unknown and has to be rejected.

Moreover, all the signals stating the properties of the decoded instruction (e.g. write-back, the number of source registers needed) are generated and sent to the *ID-stage Control Logic*.

### B.1.3 ID-stage Control Logic

It works tightly coupled to the `INSTR DECODER` and it is in charge of managing the core-coprocessor handshaking on the *issue interface*.

In addition, it dispatches to the EX-stage the required information for the correct instruction execution (e.g. type of instruction, `rs` operands, `rd` for the writeback) through the ID/EX pipe.

### B.1.4 EX-stage Control Logic

It manages the *commit interface*, the *memory interface*, the *memory result interface* and the *result interface*, implementing all the handshaking and data signals needed for a correct core-coprocessor communication.

It also generates the control signals for the ALU execution (e.g. instruction kill) and it manages the ones coming from it (e.g. memory accesses, final result providing) acting as a core-ALU intermediary.

### B.1.5 FIFOs

Being a producer-consumer system able to manage only one element (instruction) at a time, 1-element First-In First-Out (FIFO) is needed (shown in *red* in the figure). Two FIFOs are present, one managing the *issue* part and another one the *commit* part and they are connected to the related XIF interfaces.

The issue_FIFO stores:

- coproc_issue.issue_resp.accept: signal stating if the coprocessor has accepted or rejected the offloaded instruction;
- coproc_issue.issue_req.id: *id* of the current serving instruction;
- alu_block_sel: not XIF-related signal, but it is needed by the EX-stage to select the right output among the different ALU blocks present. For now, only the MAC block is present within the ALU and this signal is not used.

The issue_fifo_push signal is the signal to push a new value into the FIFO (if 1, a new data is loaded into the FIFO). It is asserted when (coproc_issue.issue_valid AND coproc_issue.issue_ready)=1, that means "when the core is offloading an instruction and the coprocessor is ready to receive it".

The commit_FIFO stores:

- coproc_commit.commit.commit_kill

The commit_fifo_push signal is asserted when coproc_commit.commit_valid=1.

The fifo_pop signal is the same for both the FIFO and it is generated by the EX-stage at the end of the current instruction execution.

## B.2 MAC general architecture

This section can be seen as a sort of continuation of *chapter 4* and starting from there the design choices will be explained.

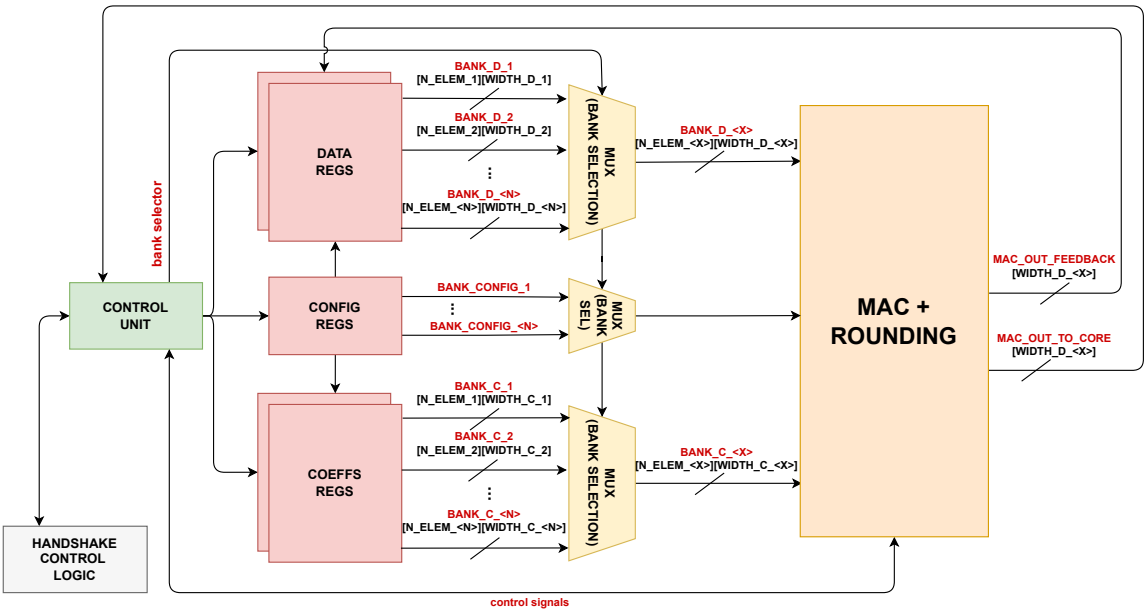



**Figure B.2:** *MAC (general) architecture (same figure as Figure 4.1).*

### B.2.1 Data registers

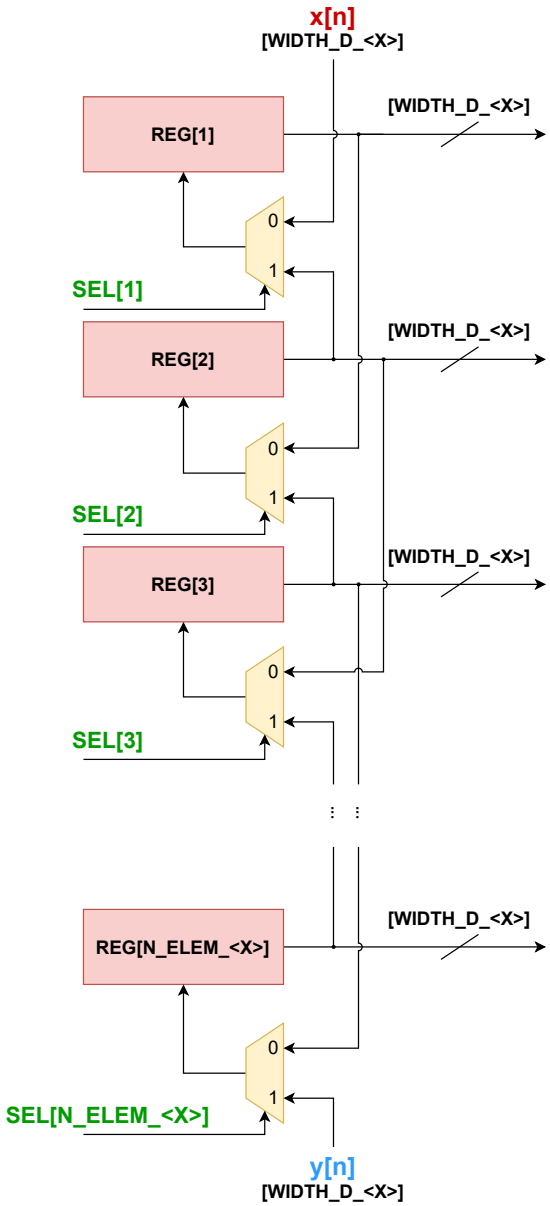



**Figure B.3:** *Data registers architecture.*

Data registers are designed to appear as a couple of two independent shift registers, one shifting downward (for the feedforward elements) and another one shifting upward (for the feedback elements). The size of each of the two parts can be set by the programmer at runtime, setting `n_feed_<X>` (see *chapter 4*).

The architecture of the *data banks* is shown in *Figure B.3*.

The feedforward/feedback partitioning is obtained through a series of MUXes (one per register) allowing one to choose if a register should take the input (next value to store) from above or from below. To split the shift registers into two independent parts, the selection signals `SEL[<N>]` needs to be a series of `0` for the feedforward part (downshifting) and a series of `1` for the feedback part (upshifting).

Consider for instance a *data bank* made by 8 elements and the user wants to have 6 feedforward elements and 2 feedback elements. The selection signals will be `SEL[1:6]=000000` and `SEL[7:8]=11`.

With a more general notation we can write:

- **feedforward part:** `SEL[1:n_forw_<X>]=000...0`;
- **feedback part:** `SEL[(n_forw_<X>+1):(n_forw_<X>+n_feed_<X>)]=111...1`.

The idea is to have configurable selection signals `SEL[<N>]` stored into configuration registers that can be set at runtime through the `SET_NFEED` instruction. The user will provide `n_feed_<X>` (the number of feedback elements wanted) that will be converted into the vector of selection signals `SEL[<N>]` through a decoder.

In addition, some pre-synthesis parameters are present to customize this structure, giving to the user the possibility to instantiate only the strictly needed hardware, avoiding useless overhead.
For instance, `N_ELEMENTS_BANK_MAC[<X>]` defines the number of registers instantiated for each bank (notice that it applies both to *data banks* and *coeffs banks*), while `WIDTH_DATA_MAC[<X>]` defines the bitwidth of the *data bank* with `bank_index=X`.

### B.2.2 Coefficients registers

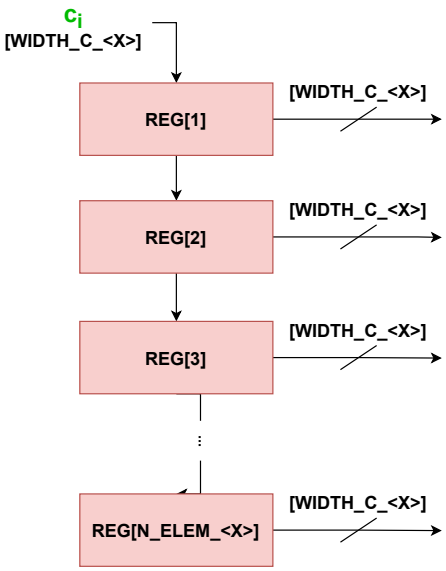



**Figure B.4:** *Coefficients registers architecture.*

The registers devoted to the *coeffs banks* are plain shift-registers. The input is $c_i$ which is a generic coefficient that can be loaded into the registers through the `LOAD_COEFF_[MEM/REG]` instructions.

As explained in *subsection 4.3.2*, the loading order of the feedforward/feedback coefficients is not so intuitive and the user should pay attention to it.
A more intuitive loading order could have been provided to the user making the structure more complex (e.g. having two different instructions to load the feedforward and the feedback coefficients), but our final decision was to give more importance to maintaining low the complexity (i.e. lower area and power) since the coefficients are loaded very few times into the program and paying attention to their order is not dramatic for the programmer.

Also for the *coeffs banks* the number of elements of the shift register (i.e. its length) and the bitwidth can be set for each bank as pre-synthesis parameters (see *section 4.2*).

### B.2.3 Configuration registers

There are three sets of configuration registers:

- **rounding:** they are devoted to storing the rounding unit configuration (i.e. the number of bits to round);
- **nfeed:** they are devoted to store the selector signals `SEL[<N>]` for the *data regs*;
- **simd:** they are devoted to store the SIMD enable/disable configuration (only instantiated if pre-synthesis parameter `SIMD_MAC=1`).

Note that each bank has its own set of configuration registers since these parameters are independently configurable for each bank.

These registers are implemented in HW as normal register files, in which one can write a specific location (one location per bank) through an index (`bank_index`).

### B.2.4 Control Unit

The Control Unit is a quite complex logic since there are many signals to be managed, most of them due to the handshaking with the main core.

It is not worth explaining all the control signals defined, since this would not add useful details to the coprocessor working principle, but it is useful to take a look at the FSMs.
Most of the control signals do not need a FSM, therefore only two FSMs are present: one for the memory transactions with the core and another one for data loading to the internal MAC registers. Mealy's FSMs have been preferred on Moore's ones, since the former brings fewer states and, overall, better performance, allowing saving some clock cycles. The advantage could seem not so relevant, but since we are working in a DSP application where a lot of samples are managed in series, saving some clock cycles per sample could improve the performance significantly. Moreover, the fact that Mealy's FSM does not cut the combinational path is not a problem in this case, since they are not on the critical path.

Unfortunately, it is impossible to indicate the transition signals on the FSM graphs because there are many of them. In addition, being Mealy's FSM, the output is related to the current state and to the input, which would make the graphical representations even more complex.

### B.2.4.1 Memory transactions FSM

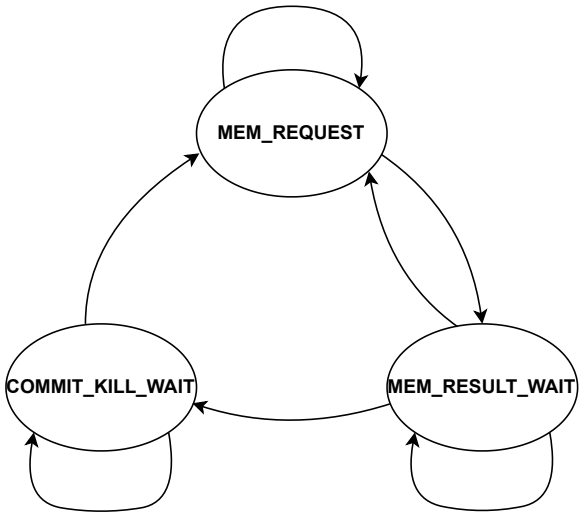



**Figure B.5:** *FSM for the memory transactions with the core.*

In *Figure B.5*, the **FSM for the memory transactions** with the core is reported.

The MAC is able to receive data/coefficients both from the core internal register file and from the data memory.
The operands from the RF are passed on the XIF through the *issue interface* when the instruction from the core is offloaded. They are managed by the ID-stage and passed to the EX-stage through the ID/EX pipe: for this reason, they are available to the MAC from the beginning and without additional effort.
Instead, the data memory access is not so straightforward, because the coprocessor can not directly access the memory, but it has to exploit the *memory interface* and the *memory result interface* to obtain a value from the data memory, with the core acting as an intermediary. Refer to [14] to get the details about the two interfaces, but summing up we can say that the former is devoted to the request of the value to be read by the coprocessor, while the latter is used by the core to providing that value to the coprocessor. Handshaking signals are present on the interfaces and they are managed by this FSM.

The role of the three states is:

- `MEM_REQUEST`:
  This is both the idle state and the state in which the coprocessor requests a value from the data memory through the XIF *memory interface*.
  The next state is:
  - `MEM_REQUEST`: if no memory request has to be performed;
  - `MEM_RESULT_WAIT`: if a memory request has been done correctly and the coprocessor has to wait for the value.
- `MEM_RESULT_WAIT`:
  After a memory request on the XIF *memory interface*, the coprocessor passes in this state to wait for the result that will be provided by the core through the XIF *memory result interface*.
  The next state is:

- `MEM_RESULT_WAIT`: if no value has been received, yet;
- `COMMIT_KILL_WAIT`: if the value has been received, but the core has not signalled, yet, through the *commit interface* if the offloaded instruction has to be committed or killed;
- `MEM_REQUEST`: if the value and the commit from the core have been both received.

- `COMMIT_KILL_WAIT`:
  This state is used to wait for the commit/kill signal from the core, if not provided after the memory request and the reception of the corresponding value.
  If the instruction has to be committed, the received value is stored into the internal registers; otherwise, if the instruction has to be killed, the value must not be stored to preserve the previous state of the registers and the memory transaction is aborted.
  The next state is:
  - `COMMIT_KILL_WAIT`: if the commit/kill on the *commit interface* has not been received, yet;
  - `MEM_REQUEST`: if the commit/kill signal has been received.

### B.2.4.2 Internal registers loading FSM

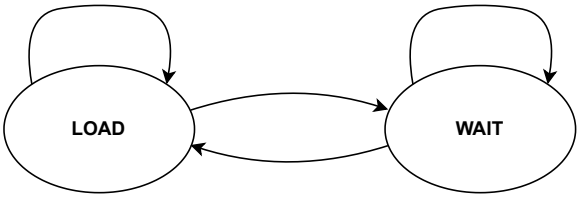



**Figure B.6:** *FSM for the value loading into the internal MAC registers.*

In *Figure B.6*, the second **FSM for the value loading** into the internal MAC **registers** is shown.

This FSM is a Mealy's one, as well. It is simpler than the previous one and it is needed only to ensure that the `write_enable`/`shift_enable` signal of the registers (*data*, *coeffs* or *config regs*) is asserted for a single clock cycle, avoiding unwanted writing/shifting. It is needed since there are some cases in which the `write_enable`/`shift_enable` signal could remain asserted for more than one clock cycle.

Without going too deep with signal naming, the role of the two states is:

- `LOAD`:
  This is both the idle state and the state in which, if the instruction has to perform a load-to-regs and the value to store is ready, `write_enable`/`shift_enable` can be asserted.
  The next state is:
  - `LOAD`: if the instruction does not need to load a value into the registers;
  - `WAIT`: if `write_enable`/`shift_enable` has been asserted during this clock cycle and the MAC is not ready to accept a new instruction (i.e. pipe stalling).
- `WAIT`:
  After `write_enable`/`shift_enable` assertion, the FSM passes in this state in order to de-assert it, ensuring its assertion for only one clock cycle.
  The next state is:
  - `WAIT`: if the MAC is not ready to accept a new instruction (i.e. pipe stalling);
  - `LOAD`: if the MAC is finally ready to accept a new instruction (i.e. the load instruction served so far is over).

## B.3 MAC arithmetic architecture

The arithmetic part of the MAC (that is the *orange* block of *Figure B.2*) is in charge of computing the final result according to the values stored in the *data banks* and *coeffs banks*, including its rounding. It supports SIMD computation that can be set at runtime (see *Figure 4.3*).
It is probably the most customizable part of the whole design with various parameters controlling its structure (see *section 4.2*). Most of the design has been defined behaviorally to ensure the wanted flexibility (e.g. bitwidth).

### B.3.1 General structure

According to two pre-synthesis parameters, `N_ELEMENTS_BANK_MAX_MAC` and `N_OPERANDS_MAC`, two different structures can be generated.

As said, each *bank* can have a different number of elements that can be set through the vector `N_ELEMENTS_BANK_MAC[<X>]`, therefore the MAC arithmetic unit must be able to process them in the worst case, that is represented by the *bank* with the highest number of elements, namely `N_ELEMENTS_BANK_MAX_MAC`.

Unfortunately, there are some cases in which `N_ELEMENTS_BANK_MAX_MAC` is too high to have an efficient arithmetic part managing all those elements in parallel, therefore we could go for an iterative approach.
In these cases, the parameter `N_OPERANDS_MAC`, representing the number of operands that the MAC is able to manage in parallel, can be set to a value lower than `N_ELEMENTS_BANK_MAX_MAC` and an iterative architecture is automatically generated.

Summarizing, two different architectures can be generated:

- `N_ELEMENTS_BANK_MAX_MAC == N_OPERANDS_MAC`:
  the MAC arithmetic block is able to manage all the elements of the *banks* in parallel, therefore it will compute the result in 1 clock cycle and no additional logic is needed (*Figure B.7*).

- `N_ELEMENTS_BANK_MAX_MAC > N_OPERANDS_MAC`:
  the MAC arithmetic block is **not** able to manage all the elements of the *banks* in 1 clock cycle, but only a subset of them. In this case, the execution will be done in more than one clock cycle through additional logic: a counter, MUXes selecting the subset of elements to be computed in the $i^{th}$-iteration, a temporary register to store the temporary result and an adder to accumulate (*Figure B.8*).
  If `N_ELEMENTS_BANK_MAX_MAC` is not a multiple of `N_OPERANDS_MAC` (i.e. their division gives a fractional number), `0`-elements are introduced as padding not influencing the result.

For the *banks* with (`N_ELEMENTS_BANK_MAC[<X>]`$<$`N_ELEMENTS_BANK_MAX_MAC`), `0`-elements are introduced into the missing positions.

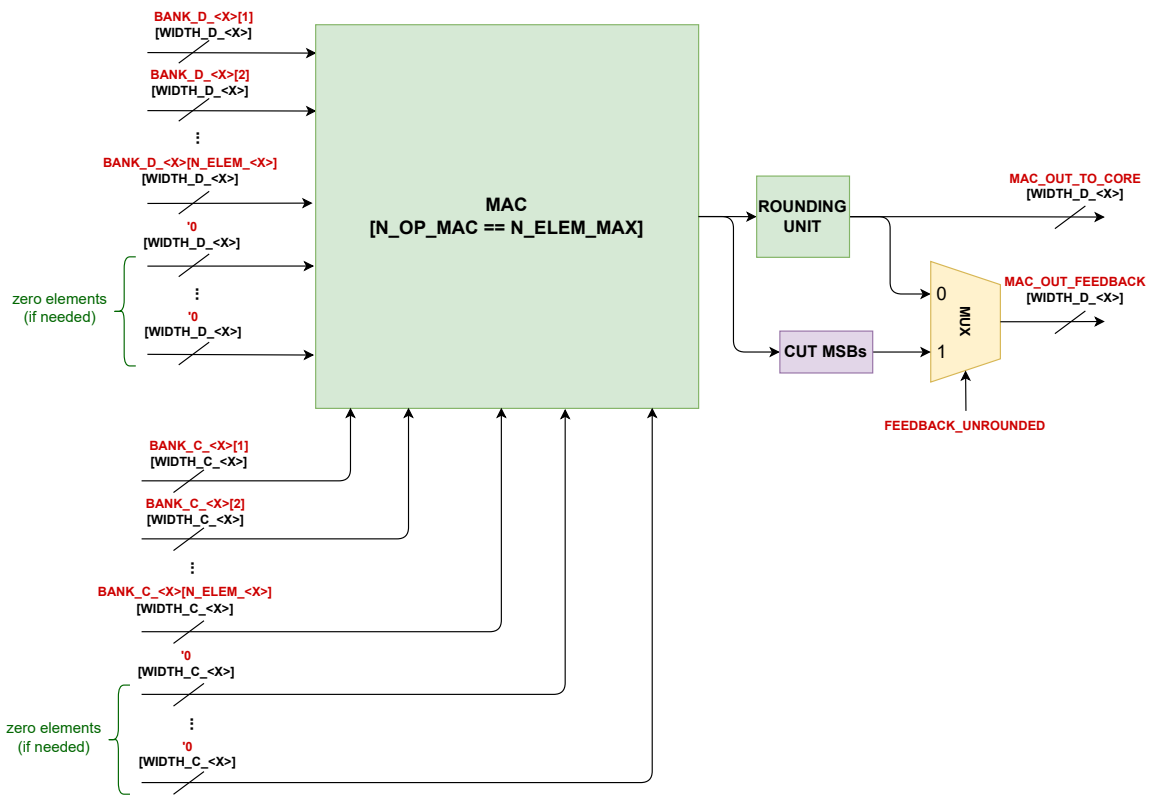



**Figure B.7:** *MAC arithmetic architecture (no iterations needed).*

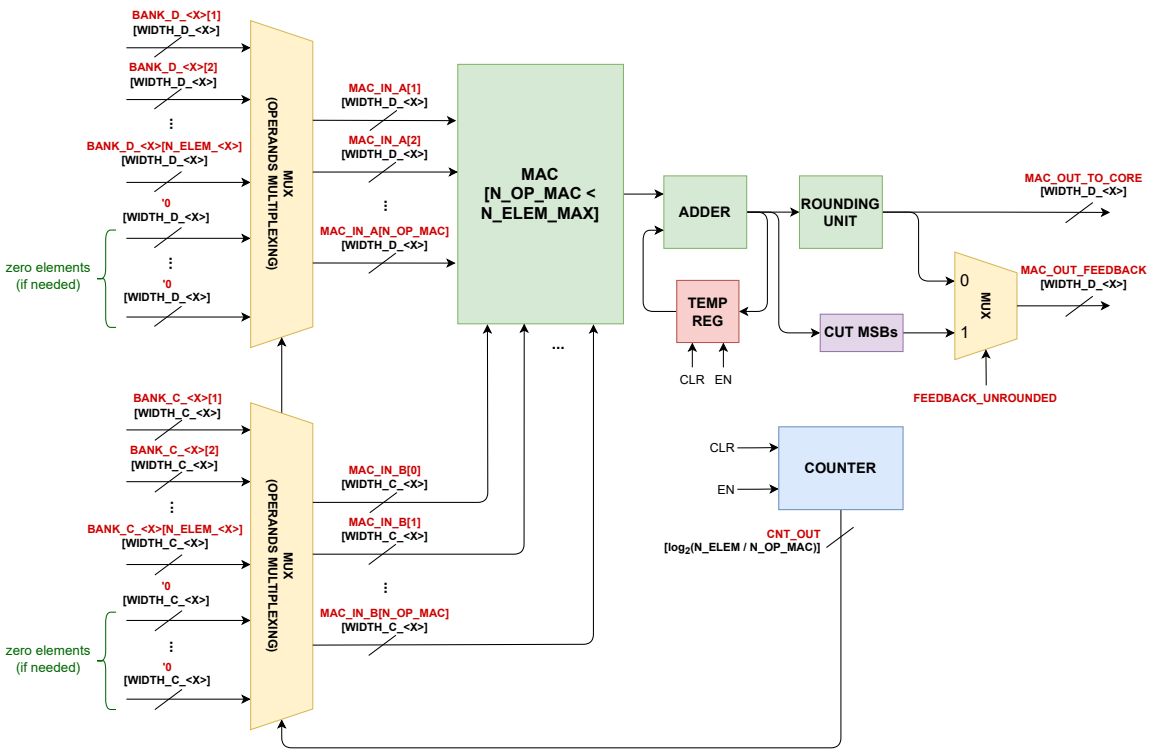



**Figure B.8:** *MAC arithmetic architecture (iterations needed).*

### B.3.2 Notes on the iterative structure

The iterative structure can be chosen if `N_ELEMENTS_BANK_MAX_MAC` is too large to have a Multiply-Accumulate computation in one clock cycle and it gives the possibility to have a simpler MAC arithmetic structure at the price of higher latency. However, the design has been optimized to ensure that the minimum time needed is taken for each operation.

In the worst case, the number of clock cycles needed is `ceil(N_ELEMENTS_BANK_MAX_MAC / N_OPERANDS_MAC)`. However, since the different *banks* could have a different number of elements, the design has been made in such a way that the number of iterations done is changed according to the actual number of elements of each bank. In other words, the number of iterations is not fixed, but it is equal to `ceil(N_ELEMENTS_BANK_MAC[<X>] / N_OPERANDS_MAC)`, according to the selected bank (`bank_index=X` in this expression).

Putting some numbers: if you have three *banks* with, respectively, [3, 15, 8] elements and `N_OPERANDS_MAC=5`, the number of iterations needed for the computation on each *bank* will be respectively 1 clock cycle, 3 clock cycles and 2 clock cycles.

This ensures no performance losses maintaining low the HW complexity to implement such a mechanism, since these are pre-synthesis parameters placed statically into the design after the synthesis, therefore no strange HW is generated (e.g. no division and ceiling).

### B.3.3 MAC

The MAC arithmetic block implements the Multiply-Accumulate operation and it supports SIMD that can be set at runtime. It is fully customizable through pre-synthesis parameters: some of them are transparent to the user and they can be set (see *section 4.2*), while others are not available to the user.

There are many papers on the efficient implementation of arithmetic operations like this (see *section 3.2*), but a good trade-off is opting for a behavioural implementation in HDL, leaving to the synthesizer the freedom to optimize it and giving the user great flexibility. Probably this solution will not bring the best arithmetic architecture in terms of performance-power-area, but it could be not worth going for a manual architecture definition, since that structure could be complex to be designed and not flexible at all.

It is worth discussing the most significant parameters influencing the MAC operation:

- `N_OPERANDS_MAC`:
  it is the number of operands that the MAC arithmetic block is able to process in parallel (as discussed in *subsection B.3.1*).

- `SIMD_MAC`:
  if `1`, the SIMD support for the MAC is enabled. Notice that enabling the SIMD support means that the user *could* use SIMD, but the actual SIMD enable must be done with the dedicated instruction (see *subsection 4.3.1*) and it can be enabled/disabled at runtime, differently for each *bank*.

- `WIDTH_DATA_NOSIMD_MAX_MAC` and `WIDTH_COEFFS_NOSIMD_MAX_MAC`:
  they define the bitwidth used for the no-SIMD operations.
  In the cases in which there are *banks* on which the user wants to enable SIMD and other ones on which it wants to disable it, the bitwidth may be different among the banks. Consider for instance two banks, one configured as SIMD and the other one as no-SIMD, with bitwidth respectively of 32-bit (two 16-bit packed values, SIMD) and 23-bit (a single value, no-SIMD). In a case like this, it makes no sense for the MAC arithmetic unit to manage no-SIMD numbers on 32 bits, but 23 bits are enough. This is the parameter the user can set to implement this behaviour, avoiding useless HW overhead.

- `WIDTH_DATA_SIMD_MAX_MAC` and `WIDTH_COEFFS_SIMD_MAX_MAC`:
  It defined the bitwidth used for the SIMD operations and it also defines the packing of the input values (`N_BIT_OP_*_SIMD` is split into the parts, as explained in *subsubsection 4.3.1.2*).
  The same discussion about bitwidth optimization done above applies also here.

### B.3.4 Rounding

Two kinds of rounding units have been developed implementing *round-to-nearest* and *round-to-nearest even* rounding scheme. The user can switch between them by changing a couple of lines in the RTL code.
They are configurable at runtime and the working principle is explained in *subsubsection 4.3.1.1*, while here the HW implementation is shown.

The number of bits to be cut `n_bits_cut` can be set at runtime through the dedicated instruction `SET_ROUNDING`.

SIMD is supported as well.

Starting from the assumptions made in *subsubsection 4.3.1.2*, with the two rounding schemes, we have:

- **round-to-nearest**: it rounds towards the nearest number; when exactly in the middle, always round-up.
  We need only to look at $x_{-1}$:
  - if $x_{-1} = 0$: round down (i.e. truncate);
  - if $x_{-1} = 1$: round-up (i.e. add $+1$ at the *ulp* position).

  In the end, right shift by `n_bits_cut`.
  Examples:
  1.25 (01.01) $\rightarrow$ 1 ($01 + 0 = 01$)
  1.75 (01.11) $\rightarrow$ 2 ($01 + 1 = 10$)
  1.50 (01.10) $\rightarrow$ 2 ($01 + 1 = 10$)
  2.50 (10.10) $\rightarrow$ 3 ($10 + 1 = 11$)

- **round-to-nearest even**: it rounds towards the nearest number; when exactly in the middle, round towards the nearest even number.
  We need to look at $x_{-1}$ and at $(x_{-2}...x_{-l})$.
  The quantity `OR`$(x_{-2},\ x_{-3},\ ...,\ x_{-l})$ is called *sticky bit* $(S)$.
  - if $x_{-1} = 0$: always round down (i.e. truncate);
  - if $x_{-1} = 1$:
    - if $S = 0$: we are exactly in the middle $\rightarrow$ round towards the nearest even number (i.e. add $x_0$ at the *ulp* position).
    - if $S = 1$: always round up (i.e. add $+1$ at the *ulp* position).

  In the end, right shift by `n_bits_cut`.
  Examples:
  1.25 (01.01) $\rightarrow$ 1 ($01 + 0 = 01$)
  1.75 (01.11) $\rightarrow$ 2 ($01 + 1 = 10$)
  1.50 (01.10) $\rightarrow$ 2 ($01 + 1 = 10$)
  2.50 (10.10) $\rightarrow$ 2 ($10 + 0 = 10$)

The *round-to-nearest even* scheme is more complex, but it ensures that no drift issues related to the rounding in the feedback systems occur (see *subsubsection 4.3.1.1*).

In addition, a further parameter `UNROUNDED_FEEDBACK_MAC` is present and it selects the value to be sent to the feedback part of the *data regs*, among the rounded and the unrounded one. It is a specific feature needed by the software part implemented

by my colleague that does not add complexity, since `UNROUNDED_FEEDBACK_MAC` is a pre-synthesis parameter, therefore the MUX shown on the right in *Figure B.7* and *Figure B.8* is not present after the synthesis.

# List of Figures

# List of Tables

# Acronyms

**ADC** Analog-to-Digital Converter. 28

**ALU** Arithmetic-Logic Unit. 23, 24, 31, 32, 34, 37, 40, 92, 93, 108

**AM** Amplitude Modulation. 13

**ASK** Amplitude-Shift Keying. 13, 14, 16, 108

**ASM** Assembly. 58–63, 68

**AWGN** Additive White Gaussian Noise. 11

**BER** Bit Error Rate. 11

**CISC** Complex Instruction Set Computer. 18

**CPU** Central Processing Unit. 1, 18, 25, 27, 34, 35, 69, 82

**CSR** Control-Status Register. 24, 35, 36, 57

**DSP** Digital Signal Processing. 1, 2, 19–21, 33–35, 41, 56, 62, 97, II

**EX-stage** EXecution stage. 92, 93, 98

**FDX** Full Duplex. 6, 7, 108

**FIFO** First-In First-Out. 92, 93

**FIR** Finite Impulse Response. 2, 29–31, 37, 40, 108

**FPU** Floating-Point Unit. 20, 26

**FSK** Frequency-Shift Keying. 13

**FSM** Finite State Machine. 7, 75, 79, 80, 97, 98, 100, 109

**GE** Gate Equivalent. 71

**GLS** Gate-Level Simulation. 70

**GPR** General-Purpose Register. 47, 48, 51–53, 57

**HDL** Hardware Description Language. 104

**HDX** Half Duplex. 6, 7, 108

**HW** Hardware. 19, 61, 62, 65, 97, 104, 105

**ID-stage** Instruction-Decoding stage. 92, 98

**IIR** Infinite Impulse Response. 2, 29, 30, 33, 40, 48, 108

**IP** Intellectual Property. 29, 32, 37, 67

**ISA** Instruction Set Architecture. 2, 18–21, 23, 26, 27, 32, 34–37, 46, 68, 72, II

**ISS** Instruction Set Simulator. 27, 56–58

**LSB** Least Significant Bit. 48

**MAC** Multiply-Accumulate. 2, 4, 30, 32–34, 37, 40, 42, 44–46, 48–50, 52–55, 59, 71–73, 75, 90, 93, 94, 97, 98, 100–105, 108, 109

**MSB** Most Significant Bit. 41, 50

**NFC** Near Field Communication. 1, 2, 5, 6, 8, 9, 16, 28, 82, II

**NVDLA** NVIDIA Deep Learning Accelerator. 33

**OBI** Open Bus Interface. 27, 70

**P&R** Place and Route. 70, 83

**PSK** Phase-Shift Keying. 13

**QAM** Quadrature Amplitude Modulation. 13

**QPSK** Quadrature Phase-Shift Keying. 13

**RF** Register File. 2, 25, 34, 35, 41, 44, 46, 59, 60, 63, 92, 98

**RFID** Radio Frequency IDentification. 3, 5–9, 12–14, 16, 108

**RISC** Reduced Instruction Set Computer. 1, 3, 18–24, 26, 27, 32–37, 46, 58, 59, 68, 82, II

**RTL** Register Transfer Level. 2, 23, 24, 27, 32, 38, 41, 48, 58, 59, 62–64, 66, 67, 69, 70, 90, 105, 110

**RX** Receiver. 8, 11, 28, 29, 108

**SAW** Surface Acoustic Wave. 8

**SDR** Software-Defined Radio. 82

**SEQ** Sequential. 6, 7, 108

**SIMD** Single-Instruction Multiple-Data. 2, 23, 24, 26, 33–36, 42–45, 48–50, 53, 54, 59, 65–69, 72–79, 81, 97, 101, 104, 105, 108–110

**SoC** System-on-Chip. 1, 32, 33

**TX** Transmitter. 8, 11, 28

**ulp** Unit of Least Precision. 48, 106

**UVM** Universal Verification Methodology. 2, 26, 57, 58, 64

**XIF** eXtension InterFace. 24, 25, 39, 90, 92, 93, 98

# Bibliography


[1] L. Calicchia et al. "Digital Signal Processing Accelerator for RISC-V". In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 703–706. DOI: 10.1109/ICECS46596.2019.8964670.

[2] Klaus Finkenzeller. *RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication, Third Edition*. John Wiley & Sons, Ltd., 2010, p. 480. ISBN: 978-0-470-69506-7.

[3] *Cards and security devices for personal identification — Contactless proximity objects — Part 1: Physical characteristics*. en. Standard ISO/IEC 14443-1:2018. Geneva, Switzerland: International Organization for Standardization, Apr. 2018. URL: https://www.iso.org/standard/73596.html.

[4] *Cards and security devices for personal identification — Contactless proximity objects — Part 2: Radio frequency power and signal interface*. en. Standard ISO/IEC 14443-2:2020. Geneva, Switzerland: International Organization for Standardization, July 2020. URL: https://www.iso.org/standard/73597.html.

[5] *Cards and security devices for personal identification — Contactless proximity objects — Part 3: Initialization and anticollision*. en. Standard ISO/IEC 14443-3:2018. Geneva, Switzerland: International Organization for Standardization, July 2018. URL: https://www.iso.org/standard/73598.html.

[6] *Cards and security devices for personal identification — Contactless proximity objects — Part 4: Transmission protocol*. en. Standard ISO/IEC 14443-4:2018. Geneva, Switzerland: International Organization for Standardization, June 2018. URL: https://www.iso.org/standard/73599.html.

[7] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 1–12. DOI: 10.1109/HPCA.2013.6522302.

[8] Michael Gautschi et al. "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[9] RISC-V International. *RISC-V*. 2022. URL: https://riscv.org/ (visited on 11/10/2022).

[10] OpenHW group. *OpenHW group*. 2022. URL: https://www.openhwgroup.org/ (visited on 09/10/2022).

[11] OpenHW Group. *Core V Cores*. https://github.com/openhwgroup/core-v-cores.

[12] OpenHW Group. *cv32e40p*. https://github.com/openhwgroup/cv32e40p.

[13] OpenHW Group. *cv32e40x*. https://github.com/openhwgroup/cv32e40x.

[14] OpenHW Group. *Core-V-XIF*. https://github.com/openhwgroup/core-v-xif.

[15] OpenHW Group. *Core-V-Verif*. https://github.com/openhwgroup/core-v-verif.

[16] Embecosm. *Embecosm toolchain*. 2022. URL: https://www.embecosm.com/resources/tool-chain-downloads/ (visited on 11/16/2022).

[17] Samuel Steffl and Sherief Reda. "LACore: A Supercomputing-Like Linear Algebra Accelerator for SoC-Based Designs". In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017, pp. 137–144. DOI: 10.1109/ICCD.2017.29.

[18] Farzad Farshchi, Qijing Huang, and Heechul Yun. "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim". In: *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*. 2019, pp. 21–25. DOI: 10.1109/EMC249363.2019.00012.

[19] Jipeng Wang et al. "A Reconfigurable Matrix Multiplication Coprocessor with High Area and Energy Efficiency for Visual Intelligent and Autonomous Mobile Robots". In: *2021 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2021, pp. 1–3. DOI: 10.1109/A-SSCC53895.2021.9634793.

[20] Stevo Bailey et al. "A Generated Multirate Signal Analysis RISC-V SoC in 16nm FinFET". In: *2018 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2018, pp. 285–288. DOI: 10.1109/ASSCC.2018.8579326.

[21] Tanya Gaurav, Amit Bhatt, and Rutu Parekh. "Design and Implementation of low power RISC V ISA based coprocessor design for Matrix multiplication". In: *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*. 2021, pp. 189–195. DOI: 10.1109/ICESC51422.2021.9532933.

[22] Rakesh Warrier, C.H. Vun, and Wei Zhang. "A low-power pipelined MAC architecture using Baugh-Wooley based multiplier". In: *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*. 2014, pp. 505–506. DOI: 10.1109/GCCE.2014.7031169.

[23] Utkarsh Parasrampuria, Chandan Misra, and Sourangshu Bhattacharya. "An Optimized Distributed Recursive Matrix Multiplication for Arbitrary Sized Matrices". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 5798–5800. DOI: 10.1109/BigData50022.2020.9378361.

[24] A. Danysh and D. Tan. "Architecture and implementation of a vector/SIMD multiply-accumulate unit". In: *IEEE Transactions on Computers* 54.3 (2005), pp. 284–293. DOI: 10.1109/TC.2005.41.

[25] Darjn Esposito, Antonio G. M. Strollo, and Massimo Alioto. "Low-power approximate MAC unit". In: *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*. 2017, pp. 81–84. DOI: 10.1109/PRIME.2017.7974112.

[26] Su Yang, Wei Yuechuan, and Zhang Mingshu. "Research and Design of Dedicated Instruction for Reconfigurable Matrix Multiplication of VLIW Processor". In: *2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)*. 2016, pp. 324–327. DOI: 10.1109/INCoS.2016.18.

[27] Abhijit Chandra and Sudipta Chattopadhyay. "Design of hardware efficient FIR filter: A review of the state-of-the-art approaches". In: *Engineering Science and Technology, an International Journal* 19.1 (2016), pp. 212–226. ISSN: 2215-0986. DOI: https://doi.org/10.1016/j.jestch.2015.06.006. URL: https://www.sciencedirect.com/science/article/pii/S2215098615001068.

[28] P. Bougas et al. "Pipelined array-based FIR filter folding". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 52.1 (2005), pp. 108–118. DOI: 10.1109/TCSI.2004.838542.

[29] RISC-V International. *Vector Extension*. https://github.com/riscv/riscv-v-spec.

[30] RISC-V International. *Spike ISS*. https://github.com/riscv-software-src/riscv-isa-sim.

# Acknowledgements / Ringraziamenti

Anche se questa tesi porta il mio nome, essa è frutto di un percorso che non sarebbe stato possibile senza il supporto di alcune persone. Questa sezione è qui proprio per provare a ringraziarle.

Grazie al mio relatore, **prof. Guido Masera**, per il supporto che mi ha fornito durante questo lavoro di tesi. I suoi feedback sono stati illuminanti e di vitale importanza per il progetto e la stesura della tesi. È superfluo commentare la sua immensa competenza tecnica, ma mi ha sorpreso l'aspetto umano del suo modo di fare. Mail dopo mail, meeting dopo meeting, è stato chiaro di avere a che fare con una persona che ama il proprio lavoro, che ha piena consapevolezza dell'estrema importanza del suo ruolo nella vita degli studenti e che si impegna per rendere la vita di questi ultimi un po' meno grigia. L'augurio più grande che si può fare ad uno studente è di incrociare persone come lei durante la propria vita accademica.

Thanks to all the team at NXP which I have had the luck to work in.
Grazie a **Tiberio Fanti** per il tempo trascorso nel team che dirige. Credo lui sia una persona rara, dal punto di vista personale e professionale, incarnando i valori più nobili e puri dell'italianità. La sua simpatia, unita al senso di responsabilità e dedizione al lavoro, rendono l'atmosfera all'interno del team piacevole, ma allo stesso tempo proattiva. Per ogni persona come te che va via dall'Italia, il nostro Paese perde un'occasione per risalire il baratro in cui sta precipitando.
Thanks to **Stefan Brennsteiner** for his great kindness and wisdom in managing the project. I really appreciate the precision and devotion he puts into everything he does and his respect for hearing others' opinions without prejudices. Your support has been essential and it has been a pleasure to work under your supervision.
Grazie a **Luca** per la sua infinita disponibilità, prima, durante e dopo il lavoro di tesi. Si è sempre dedicato senza riserve ad ascoltare idee e problemi, dando un supporto imprescindibile durante l'attività. Alla fine, questa esperienza non sarebbe neanche iniziata se non fosse stato per lui.
Un ringraziamento particolare va a **Valerio** e **Marianna**, persone speciali e amici meravigliosi. Non avrei mai pensato di passare dal grigiore della solitudine delle prime settimane in Austria, alla malinconia dell'ultimo caffè insieme: "quando uno del sud Italia va in Austria piange due volte, quando arriva e quando se ne va"...o forse ricordo male la citazione? Avete reso magico questo periodo in Austria, di cui

conserverò gelosamente tutti i ricordi.
Thanks to **Philipp** and **Marcel**, my project mates. Working with you has been enjoyable and constructive.
A final thanks to all the people I met during this time. All of you have contributed to my personal and professional growth.

Ringrazio **la mia famiglia** per aver affrontato di petto e senza farlo pesare tutti i sacrifici degli scorsi 5 anni e, in particolare, di questo ultimo, turbolento anno. Ognuno sviluppa il proprio carattere e fa emergere il suo personale modo di essere durante la strada della vita, ma avere una famiglia così alle spalle, fa sì che sia più facile trovare la strada. Grazie per avermela indicata sempre con chiarezza e di aver rispettato le mie scelte. Se posso provare ad insegnare io una cosa a voi, è quella di riflettere su dove eravamo un anno fa e dove siamo oggi. È proprio quando tutto sembra perduto e monti che paiono insormontabili si pongono davanti a noi che bisogna alzare la testa e iniziare la scalata. La vetta, prima o poi, arriverà e, guardando indietro, sembrerà tutto più facile di quanto avremmo mai immaginato. L'importante è avere persone accanto a supportarci e io sarò qui, come voi lo sarete per me. La strada per Capodichino, che oramai conoscevamo a memoria, era fatta alternativamente una volta con le lacrime agli occhi e l'altra col sorriso in volto, ma sempre con la speranza di riunirsi definitivamente. Un giorno. Forse è presto per dirlo, ma quel giorno potrebbe essere arrivato.

Grazie a **nonna Anna** e **nonno Francesco** per la loro continua vicinanza. Non ho mai capito quale sia il modo giusto per dimostrare loro il bene che gli voglio: se conviene fare come nonna, che è affettuosa ed emotiva, o come nonno, serio e pragmatico. Forse dovrei trovare un mio modo. Nel dubbio, sappiate che vi voglio bene e che mi dispiace avervi dato pensieri e dispiaceri con la mia partenza. Ora siamo finalmente qui, tutti insieme e, alla fine, tutto sembra essere passato in un attimo.

Un ringraziamento particolare va a mia sorella **Arianna** che, invecchiando, sembra seguire più l'iter dell'aceto che quello del vino...scherzo! Al contrario, credo sia proprio questa profonda differenza caratteriale a tenerci uniti e a spingerci a migliorarci, facendo sì che ognuno prenda il buono dall'altro. Sono sicuro che la sua intelligenza e il suo acume la porteranno lontano e che la sua maturità farà la differenza nell'affrontare i difficili anni che la attendono. Ricorda di godere di ogni singolo momento di gioia che potrai cogliere durante la tua vita e se in certi istanti tutto ti sembrerà buio, non dimenticare che potrai sempre contare su di me. Certo, dovrai accontentarti della pasta al dente e del focolare fatto male, ma, se ne avrai bisogno, io sarò qui per aiutarti a trovare uno spiraglio di luce.
Fatto sta che, probabilmente, l'unica cosa che ci accomuna è l'avere un carattere difficile che poche persone sanno approcciare e sopportare.

Per fortuna, io ne ho trovata una, probabilmente l'unica e sola. La mia. Stare lontano da casa ti chiarisce le idee inequivocabilmente e la nebbia di colpo svanisce.

Per sempre. È passato esattamente un anno da quando tutto ha preso forma, ma a me sembra di non aver mai vissuto senza di te, **Maddalena**. E forse è proprio così.
Grazie per essere stata lì, *ferma a guardare me che scendevo giù dalle scale*, non abbandonando mai la certezza di quello che sarebbe stato.
Grazie per essere stata qui, a supportarmi durante questo viaggio, mettendo da parte i tuoi pensieri e problemi e senza chiedere nulla in cambio.
Hai riempito la mia esistenza e mi hai reso una persona migliore. Non desidero nient'altro dalla vita, se non condividerla con te. Credo che la sensazione di non meritare tutto l'amore che sai darmi non sparirà mai e che dovrò imparare a conviverci. L'unica cosa che posso fare è darti tutto me stesso e un po' di più, sperando basti.

*Te l'avrò detto già tremila volte*
*Se servirà, te lo dirò di più*
*Ti penso pure quando sto alle poste*
*Spedisco lettere ai tuoi occhi blu*

Ti amo.

Grazie a **Rosario** e **Antonio**, la cui amicizia indissolubile mi accompagna da molti anni e sono sicuro durerà per tutta la vita. Quanto dista Graz da Pedaline e Fonte? Ad occhio direi pochi metri, dato che è rimasto tutto immutato, come se non me ne fossi mai andato.
Grazie a **Stefano** e **Cosimo**, che mi fa sempre piacere sentire e vedere per condividere interessi e passioni. Ultimamente gli impegni ci hanno un po' allontanato, ma reputo la loro amicizia preziosa.

Grazie **alla mia band**, Tonino, Carmine, Vittorio, Serena, Carmen e Giovanni, che nell'ultimo anno ha dovuto ripensare alcuni meccanismi che erano rodati e che sembravano imprescindibili. È nei periodi di crisi che le migliori idee vengono fuori, per questo penso sia stato un momento di crescita per scoprire aspetti fino a quel momento sconosciuti. Grazie per aver fatto sì che la musica rimanesse parte integrante della mia vita e farò di tutto affinché questo valga anche per il futuro.

Grazie ad **Alessandra**, **Sara**, **Pier**, **Andrea** e **Filippo**, con cui ho condiviso buona parte del mio percorso a Torino. Siete stati molto più di semplici colleghi e, anche se ognuno di noi imboccherà la propria strada, spero che il rapporto che ci ha legato in questi pochi mesi possa perdurare nel tempo.

Ci sarebbero tante altre persone da citare, nel bene e nel male, quindi grazie anche

*A te, se sei rimasto con*
*me fin proprio alla fine.*