POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Spiker-V: bringing Neuromorphic Intelligence at the edge through the optimized integration of a SNN Hardware Accelerator with a Low-Power RISC-V Processor

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Dott. Alessio CARPEGNA

Candidate

Renato BELMONTE

October 2024

Summary

Spiking Neural Networks (SNNs) represent the latest advancement in Artificial Neural Network development, designed to realistically mimic brain behavior. The innovative aspect of this technology lies in its use of spike signals, unlike traditional neural networks that rely on continuous signals, to encode data and the artificial implementation of a model that emulates membrane potential, which helps extract information from the incoming data. A spike signal can be digitally represented by a single bit indicating, for instance, that an event occurred. Various models implement the membrane potential using different approaches.

This thesis explores the integration of a Spiking Neural Network hardware accelerator into a RISC-V processor to enable energy-efficient computation. The goal is to develop a system optimized for low-power consumption while maintaining real-time data processing capabilities. Neuromorphic computing, inspired by the brain's event-driven nature, offers an alternative to traditional deep learning, which is computationally expensive and energy-intensive. By mimicking biological neurons that communicate through spikes, SNNs promise improved energy efficiency, especially for edge computing devices. Furthermore, the open-source nature of the processor allows hardware customization to meet specific requirements for size, performance, and power consumption.

This thesis presents a new framework to demonstrate the efficiency of this solution, using an FPGA that hosts both the RISC-V processor and dedicated hardware to implement the Spiking Neural Network accelerator. This setup enables an easily programmable and controllable platform, which allows direct debugging of the processor core, enabling rapid prototyping and testing. Additionally, the open-source nature of the RISC-V processor allows extensive hardware customization, making it adaptable to specific requirements such as size, processing performance, and power consumption. This level of flexibility is crucial in developing specialized, low-power computing systems.

The framework uses an AMD FPGA, specifically the Zynq UltraScale+ MP-SoC ZCU102 Evaluation Kit, which supports all major peripherals and interfaces, enabling development for a wide range of applications. This type of device contains both a dedicated system for data processing and communication handling, as well as a programmable component. The programmable logic of the device can be directly programmed through a software suite exploiting a hardware description of the IP. As such, the PULP platform was chosen as the microcontroller to be implemented within the FPGA. More specifically, *PULPissimo*, a 32-bit open-source microcontroller that can support various types of RISC-V cores, is implemented in the FPGA. Finally, inside the microcontroller SoC a peripheral acting as an adaptor between the core and the hardware accelerator is allocated. This module consist of two interfaces, the first one is exploited for reading the incoming spikes from the memory and provides the data to the accelerator, while the second one is used to write the accelerator results to the memory. In addition these interfaces are in charge of the management of the controls signals used to master the accelerator.

Along this thesis work a full framework is described in its fundamental elements showing strengths and weakness of this solution. The goal is to prove that this kind of solution can guarantee at the same time a good performance and a cost-effective implementation mainly focused on power saving. The final outcome is a low-power system, flexible and reconfigurable, able to enhance the RISC-V domain by the power of neuromorphic computing.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

CPU

Central processing unit

RISC

reduced instruction set architecture

AI

Artificial intelligence

ANN

artificial neural network

SNN

spiking neural network

IoT

Internet of things

FPGA

Field programmable gate arrays

PS

Processing system

PL

Programmable logic

RTL

Register transfer level

PLD	Programmable logic device
IC	Integrated circuit
ASIC	Application specific integrated circuit
CAD	Computer-aided design
SPICE	Simulation Program with Integrated Circuit Emphasis
CPLD	Complex programmable logic device
FF	Flip-flop
LUT	Look-Up Table
SRAM	Static random access
CLB	Configurable Logic Block
LE	Logic element
VLSI	Very-large-scale integration
HLL	High-level languages

ISA

Instruction set architecture

PULP

Parallel Ultra Low Power

uDMA

micro Direct Memory Access

SDK

Software Development kit

HWPE

Hardware Processing Engine

MPSoC

Multi-processor System on Chip

HDL

Hardware description language

TCL

Tool Command Language

IP

Intellectual property

TCDM

Tightly Coupled Data Memory

AXI

Advanced Extensible Interface

GVSoC

Generic Virtual System on Chip

GUI

Graphic USer INterface

ELF	Executable and Linkable Format
EDA	Electronic Design Automation

Chapter 1

Introduction

According to [1] it is possible to define an Artificial Neural Network (ANN) as a try to emulate the architecture and the information handling system typical of the biological nervous system. Further advances in the field of neuromorphic intelligence have been brought about by the spiking neural network (SNN). The introduction of time information represents a meaningful step forward regarding the parallelism with the actual biological model. Then, a hardware accelerator is tasked to handle the actual implementation of the theoretical model through a custom design.

This chapter presents a brief overview of artificial neural networks to understand better the reasons that have led to the choice of a hardware accelerator. Firstly, the neural network working mechanism is explained. Then, the approach of spiking signals is focused, highlighting the motivations behind this choice. Finally, the use of a hardware accelerator and RISC-V processor is motivated.

1.1 Motivations

The common architecture of an Artificial neural network is characterized by a set of nodes and interconnections between them. The purpose of this approach is to efficiently solve complex problems, such as prediction or modeling, by a *divide and conquer* technique. The nodes are in charge of receiving inputs and producing outputs according to a certain function. These computational units act like the neurons inside the brain that receive signals, also known as action potentials, through synapses situated on the dendrites. If the signal surpasses a defined threshold, the neuron is activated and a new signal is propagated towards other nerve cells. On the other hand, the connections between nodes determine how the information flows in the network. This linking behaves like the axons that connect the neurons. As a result, the model of artificial neurons and interconnections highly abstracts the real neuron complexity.

Over the years, ANNs reached a higher level of similarity to the real nervous system and also the computational power is increased [2]. The third-generation ANN, known as the Spiking neural network, represents significant progress due to the introduction of time-space information analysis, but can further improve the technologies already implemented into the previous generation, as it has been demonstrated in [3]. This improvement is clearly shown in Figure 1.1 where the improvement among the different generations from multilayer perceptron [4] is represented. Indeed, at each time step, if the input reaches a certain threshold the neuron generates a spike signal [5]. Then, both input and output can be seen as spike trains. This information can also be read as binary data:

- (1) = In the current time step a spike signal is generated
- (0) = In the current time step no spike signal is generated

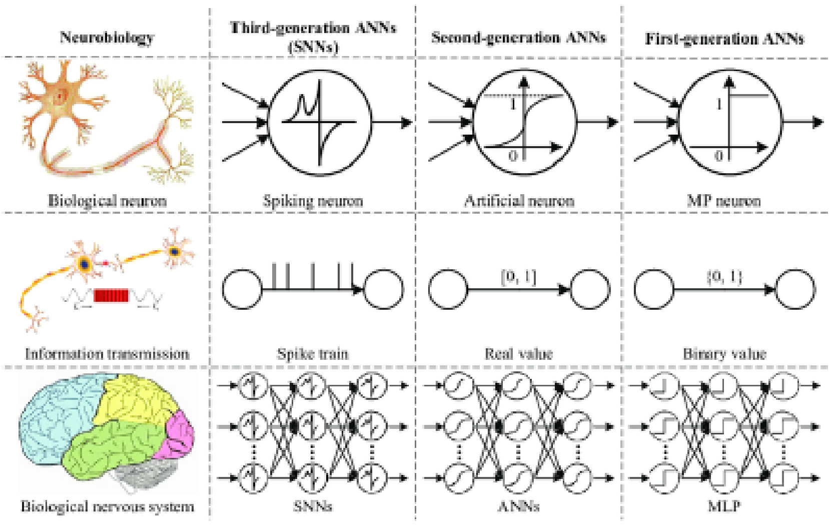


Figure 1.1: Supervised learning in spiking neural networks: A review of algorithms and evaluations [6]

In order to efficiently develop an architecture that is accurately shaped by the biological model, a custom hardware implementation can be the better solution, instead of a software-based answer. This kind of implementation is called *hardware accelerator* because the primary purpose is to enhance performance achieving also an acceptable power consumption. Then, to handle the entire platform that hosts the implementation of an SNN a RISC- V core is customized. The dominant reason behind this solution is the possibility of entirely modifying the open-source hardware to optimize the final result.

1.2 State-of-the-art

As stated by [7], [8] and [9], dedicated hardware for general-purpose spiking neural networks is hardly procurable due to the heterogeneous requirements in terms of network architectures, encoding methods, and neuron models.

Another approach, followed in the past, consists of implementing a software-based network. This kind of implementation is not suitable in the case of a few power computational units, as explained in [10], despite features like high parallelism and event-driven computation due to the sparse nature of the SNNs. SpiNNaker is a noteworthy example of large network models that aspire to achieve a massively parallel million-core computation and according to [11] is suited to the modeling of large-scale spiking neural networks in biological real time. This scope is achieved through 18 ARM968 processor nodes, which guarantee also good energy management sacrificing some performance.

Although significant computational power can be achieved by exploiting a massive amount of resources, design tools and simulators are not well-suited for implementing real-time systems. Additionally, it is crucial to consider that Internet of Things (IoT) and wearable devices require compact size and efficient power management. As a result, custom hardware designed for a specific task can become the most viable solution.

Field Programmable Gate Arrays (FPGAs) are programmable logic devices composed of configurable logic blocks and a connecting grid. By modifying these connections, FPGAs can perform a wide range of digital functions. FPGAs offer not only cost reduction but also greater flexibility. Additionally, they are valuable in embedded system applications because they can concurrently handle both hardware and software development.

In order to integrate the hardware accelerator inside the FPGA there are two ways:

- Exploiting the Processing System (PS) to handle the communication interfaces towards the external world, while the Programmable Logic (PL) is exclusively committed to the accelerator engine.
- The design is entirely implemented inside the PL, in this situation, an additional core is needed to manage the data and control the accelerator.

The last alternative has the benefit of allowing the customization of the design at the Register transfer level (RTL). Thus, this solution enable the realization of optimized interfaces between the core, the memory and the accelerator. In addition, it is possible to include unique instructions to configure and manage the hardware proficiently. Specifically, a custom RISC-V processor can be synthesized through the programmable logic to meet the SNN requirements and save area.

1.3 Contribution

Precisely in this perspective, the purpose of this thesis is to present the potential of a fully implemented framework for an SNN. The features of the project insist on achieving better results in terms of specialized power computation with the minimum effort in terms of area occupied and power consumption. In order to fulfil this purpose the following means are employed:

- **FPGA:** All the inner components are implemented inside the PL of the FPGA to exploit to the maximum the potentiality in terms of customization.
- **RISC-V-based microcontroller:** In order to handle the communications to the outside world some peripherals surround the core. In particular, these modules are used to receive data and, after the processing, can send the computation result.
- **RISC-V core:** It is in charge of programming other modules according the instructions, feeding with data, and receiving results.
- **Hardware accelerator:** This unit is responsible for receiving data, performing computations, and giving results.
- **SNN engine:** Inside this module, the data are used in the neural network to produce outcomes.

In chapter 2 an overview of these subjects is detailed, while in chapter 4 the setup of the environment is delineated. Thus, the hardware accelerator is focused on in chapter 5 and its results are presented in 6.

Chapter 2

Background

This chapter provides a brief overview of the main topics that are required to fully understand not only the potentiality of the framework but also the limitations. Indeed, firstly it depicts the creation and growth of FPGA technology highlighting the motivation behind a progressive rise. Then, the eternal fight between RISC and CISC architectures will be discussed from the RISC point of view. After that, an emerging solution for the realization low power devices known as the PULP platform will be explained. Lastly, the Spiking neural network will be discussed.

2.1 FPGA

FPGAs are commercial programmable devices that allow the realization of custom hardware implementation thanks to a flexible platform and ensure a low development cost. These devices are a subset of a family of logic devices known as Programmable logic devices (PLDs) that embrace other solutions such as Programmable array logic (PAL) and Programmable logic array (PLA).

2.1.1 History

The first device that it is possible to recognize as a forerunner of the modern FPGAs was produced by Xilinx in 1984 and is known as XC2064 [12]. At that time, before the adoption of the fabless model, only an integrated device manufacturer could create its chips. As a consequence, in addition to the design requirements also huge investments in facilities and skilled staff were indispensable to entry into IC business or to comply with Moore's Law. This empirical relationship between the number of transistors and the time stated that the number of in an integrated circuit (IC) doubles about every two years. A representation of these phenomena is shown in figure 2.1. The changeover began in the mid-1980s when the engineer

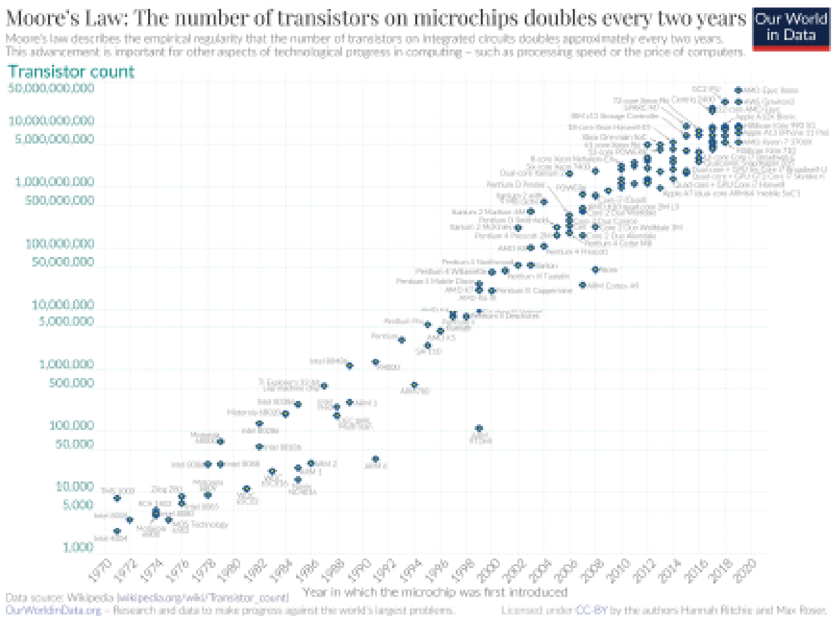


Figure 2.1: A semi-log plot of transistor counts for microprocessors against dates of introduction, nearly doubling every two years [13]

Ross Freeman formerly working at Zilog, the microelectronic company founded by Federico Faggin and Ralph Ungermann was sure that each function realized through an Application-Specific Integrated Circuit (ASIC) could be replaced by the same piece of silicon. Freeman was able to persuade both Jim Barnett and Bernie Vonderschmitt to leave Zilog and found a new start-up, named Xilinx.

In pursuit of Vonderschmitt's vision, Xilinx focused exclusively on its best result: the design of the best ICs on the market. On the other hand, Xilinx had to partner with other actors to gain access to skills and assets within the chip manufacturing area. Thanks to Vonderschmitt's ability to persuade his friend Saburo Kusama, fellow fab-management executives at Japan's Seiko Corp., Xilinx succeeded in delegating their IC fabrication to Seiko.

Another former Zilog employee, Bill Carter, was in charge of actually designing the first functional FPGA. Carter has to figure out a way to deal with challenging situations as the realization of a never-designed-before IC with an IC fab on the other side of the Pacific, overcoming barriers of language and culture. Vonderschmitt knew that the unique fabless approach united with a first-of-its-kind chip could scare off customers, then regularly advised Carter to keep risk to a minimum and not try anything "too clever or exotic"[14].

In this perspective, XC2064 would be a 1,000-ASIC-gate equivalent with a working frequency of 18 MHz. As a consequence, Carter required that Seiko completely characterize its process to provide minimum feature widths to include all the FPGA features as closely as possible.

To deal with Vonderschmitt's concerns about risks, Carter realized the implementation through one modular configurable logic block (CLB) and one modular I/O block, with slight variations in some specific positions. This decision allowed the designers to manually verify the logic design instead of relying on premature and unreliable versions of computer-aided design (CAD). Most of the time was spent verifying the few unconventional blocks through Simulation Program with Integrated Circuit Emphasis (SPICE). An example of this attitude was Carter's decision to employ fewer p-channels and more n-channels in his CMOS design in order to improve performance and save space.

After the manufacturing by Seiko, 25 FPGA wafers were delivered to Carter to be carefully tested by means of probes and a homemade debugger. Unfortunately, only one wafer did not show dead short, although it suffered a significant current draw. On the other hand, this single wafer was useful to understand that an insufficient etching was the main failure cause. Moreover, the usable chips were utilized to successfully run a simple bit stream implementing an inverter.

Finally, after working hard to solve the etching and other problems, Xilinx and Seiko made XC2064 the first commercially available FPGA in the world. This remarkable step was just the beginning of the fruitful relationship between the two companies that changed the history of IC manufacturing and more.

2.1.2 Working principle

Nowadays, programmable logic technology represents an advantageous trade-off between high per-unit costs and, on the other side, outstanding performances and power consumption. As a result, custom digital hardware represents the best choice for low-volume design and prototyping, because the same design can be used for different applications. At the same time, the unit cost is kept low thanks to a large volume of production.

To understand the key advantages of choosing an FPGA as PLD and how it works, it is possible to briefly evaluate Complex programmable logic devices (CPLD). This top-down approach stems from exploiting programmable array logic, in other words, this basic block has a programmable AND plane and a fixed OR plane. A schematic example of PAL is shown in figure 2.2, where there are 3 inputs and 2 outputs resulting from how the interconnections are programmed. In addition, feedback connection can increase the complexity of the implemented logic functions at the cost of consuming more resources and introducing higher delays. The main

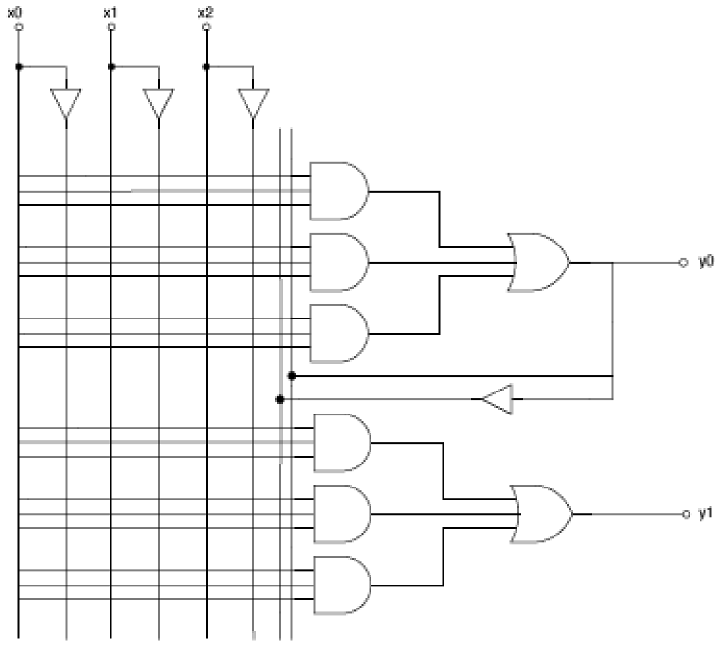


Figure 2.2: Programmable logic device schematic

disadvantage of this choice is the lack of solutions to implement sequential functions. A feasible workaround can be a macrocell shown in figure 2.3, a configurable logic circuit made of PLA where each output is coupled with a flip-flop and a multiplexer allows to select among different functions:

- Combinational bypass (direct and inverted)
- Latched output (direct and inverted)

Therefore, a collection of macrocells makes a macroblock to implement more advanced features. With the same approach, a complex programmable logic device is set up by some macrocell and increases the number of inputs and outputs. As a consequence, the complexity of routing signals and programming interconnections increases and the number of flip-flops can become unaffordable.

In order to satisfy the large storage demand for temporary data, FPGA is a

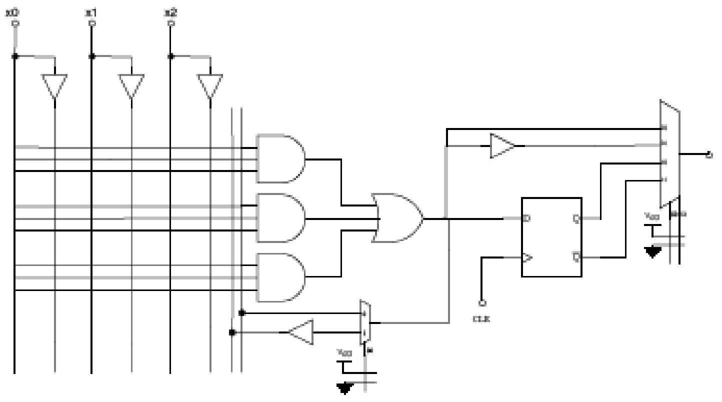


Figure 2.3: Macrocell schematic

better choice because they are programmable logic devices based on Look-Up Tables (LUT). A LUT is a small memory used to implement combinational logic functions with a limited number of inputs and outputs. Nowadays, the most suitable technology to implement these devices is SRAM, static random access memory. LUTs and optional FFs are grouped in Configurable Logic Block (CLB) or Logic Element (LE). Then, A collection of many CLBs or LEs, organized as a matrix array of rows and columns, with configurable I/O blocks to connect to the external world create an FPGA. Moreover, programmable interconnections allow the route signals from one CLB to another. The LE architecture depicted in figure

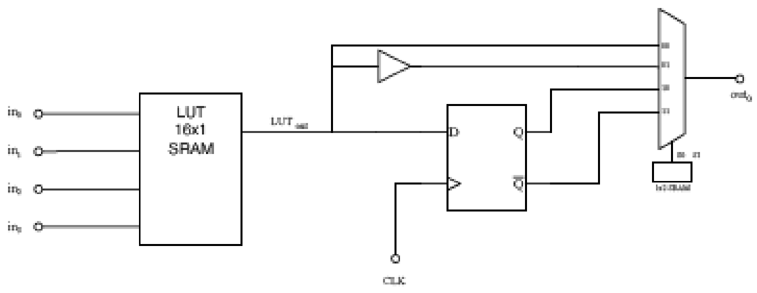


Figure 2.4: Logic element schematic

2.4 is similar to a macrocell, but the 4-input 1-output LUT, written only during configuration, substitutes the logic gates.

The other innovative logic element of the FPGA design is the switch matrix drawn in figure 2.5 The purpose of these pass transistors is to prevent conflicts. Specifically, if a transistor is not connected, adjacent segments may belong to

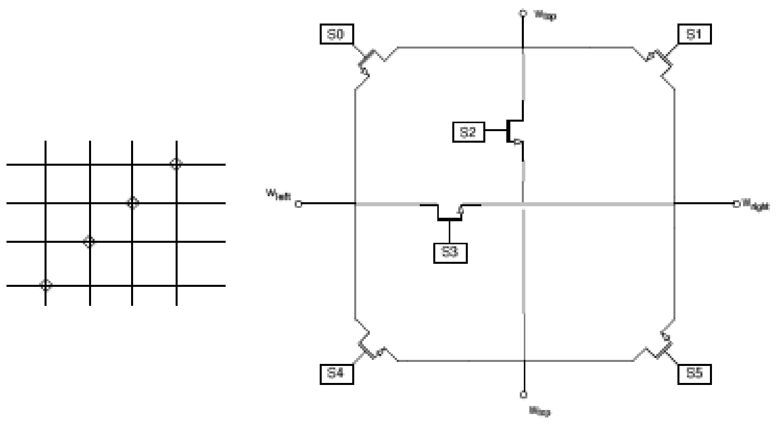


Figure 2.5: Switch matrix: on the right there is the detailed view of a switch

different interconnections. This approach enhances flexibility and simplifies routing. However, the switch matrix also increases both delay and the area occupied. Therefore, it is crucial to assess the optimal trade-off between cost, performance, and flexibility.

2.2 Reduced Instruction Set Computer

Increasing the processor bandwidth is no longer sufficient to improve the number of instructions processed in a given time due to the physical limitations of current technology, united with the increasing challenges of heat dissipation. Then, alternative technologies and architectures can be explored to use the actual knowledge more efficiently. RISC architecture represents a significant example of this approach because cutting-edge modules can be used to greatly increase performance.

2.2.1 More complex more power?

In the 1980s and 1990s chip area and processor design complexity were the primary constraints, these factors have still a considerable impact on the hardware cost-effectiveness of computer architects. As stated by David A. Patterson and David R. Ditzel in [15], the trend that relates the additional complexity with a profitable increase in performance is not always cost-effective. They proposed to take the opposite path: a Reduced instruction set computer, in other words, reduce the complexity of the architecture to enhance the performance.

The first proof in support of Patterson and Ditzel's approach was given by W.G. Alexander and D.B. Wortman in [16]. In this paper, the authors demonstrated that for a specific compiler, IBM 370, 10 instructions accounted for 80% of all instructions executed and 99% can be reached with only 30 instructions. Then, is clear that the entire architecture is not fully exploited in most of the cases. In the same direction, Andrew S. Tanenbaum in 1978 [17] empirically demonstrated that for a particular architecture, a smaller instruction can often do the same work as longer and more complex instructions.

Nevertheless, the complexity of computers continues to increase. In their paper, Patterson and Ditzel attempted to explain the motivation behind this phenomenon. Memory was a major concern at the time, as designers had to address both the imbalance between CPU and memory speeds, as well as prohibitively high costs. As a result, increasing code density to save space and using single instructions to perform complex operations emerged as the most cost-effective solution. However, from another perspective, increasing complexity to achieve higher code density can be a double-edged sword. The savings from increased complexity are negated if the resulting CPU becomes prohibitively expensive.

In addition, it is important to emphasize that the primary goal of a computer company is not necessarily to design the most cost-effective computer, but rather to maximize profits. From this perspective, adding new and more powerful instructions has a positive impact on potential customers, who may not be able to objectively assess cost-effectiveness. Furthermore, the easiest way to maintain upward compatibility is to add features rather than making fundamental changes to the architecture.

2.2.2 RISC: changing of approach

With the rise of Very-large-scale integration (VLSI) focusing the research effort on cutting-edge architecture implementation became even more critical. First of all, a low-complexity implementation needs a smaller area to be implemented, then also in the 80s, a single chip would have been able to contain the entire CPU design. For the same reason keeping up with Moore's law became more feasible because that design time is considerably reduced. The possibility of exploiting in a more efficient way the same area enables the implementation of techniques like pipelining to exploit instruction-level parallelism and caches to take advantage of spatial and temporal locality.

In particular, pipeline implementation was an innovative way to think about the CPU working. About this subject, Shlomo Weiss and James E. Smith in [18] studied a variety of hardware pipeline scheduling in order to find the possible trade-offs to achieve high scalar performance. In a pipelined computer, instruction processing is broken into segments, and processing proceeds in an assembly line fashion with the execution of several instructions being overlapped. A schematic example is shown in figure 2.6 where a five-stage pipeline is represented:

- Instruction fetch (IF): Instructions are read from the memory using the program counter (PC), a special register, to retrieve the address.
- Instruction decode (ID): Instruction bits are decoded in simple combinational logic to produce control signals.
- Execution (EX): The Execute stage is where the actual computation occurs.
- Memory (MEM): During this stage is possible to access the memory.
- Write back (WB): The result is written into the register file.

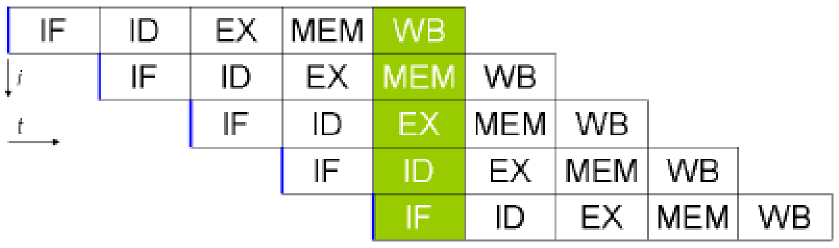


Figure 2.6: Basic five-stage pipeline in a RISC machine [19]

This approach theoretically can achieve high throughput, but hazards and memory bottlenecks can reduce the effectiveness unless proper countermeasures are taken. As effectively demonstrated after around 10 years by [20] and [21] the advantages of the RISC approach would have led to significant advancements in computer architectures. Thus, even the most skeptical designers were convinced that the implementation benefits of RISC concepts could overcome the advantages of complex instructions.

2.2.3 How it works?

Then, it is important to highlight how the RISC concepts were implemented and the following improvements. In order to briefly give an overview of an actual implementation it is possible to bring up the Berkeley RISC I as a case in point [22]. First of all, some ideas were taken as cornerstones:

- One clock cycle instructions. Then, micro instructions become useless saving area committed to decoder and increasing performance.
- All instructions are the same size.
- Only load and store instructions access the memory. Getting rid of complex addressing modes makes the design more simple.
- Ensure high-level language (HLL) support with a special consideration for C and Pascal.

Regarding the last point, the frequency and impact of HLL statements were evaluated to determine which required hardware support to ensure optimal performance. The analysis identified the CALL procedure as the most time-consuming operation, primarily due to the reliance on subroutines to replace complex instructions. To address this, a register window structure was implemented to accelerate the CALL function and minimize memory accesses. Instead of relying on a single set of registers for all operations, the CPU utilizes multiple sets of registers, known as register windows, with each function or procedure call assigned its own window.

In other words, the first ten registers (from *r0* to *r9*) are known as *global* registers and they are not saved or restored. Then, the other registers (from *r10* to *r31*) are broken in three part:

- 26 - 31 (HIGH): are reserved to parameters from the calling procedure.
- 16 - 25 (LOCAL): are reserved to the local scalar storage.
- 10 - 15 (LOW): are reserved to parameters to the called procedure.

To preserve information during a CALL procedure, the lower registers of the calling frame are hardware-overlapped with those of the called frame. Additionally, a specific routine must be executed when no free registers are available.

To implement these architectural improvements, the initial instruction set had to be limited in size, comprising only 32 instructions, which can be categorized into the following types:

- Arithmetic and logic operations performed on the registers.
- Memory access instructions like *load, store* to move data between registers and memory.
- Branch instructions like *call, return*, conditional and unconditional jumps.
- Miscellaneous.

Regarding data and addresses size, the first version of RISC architecture allows 32-bit addresses and 8-, 16- and 32-bit data, then all the registers are 32-bit wide. In particular figure 2.7 shows the format for register-to-register instructions:

- OPCODE: Operation code field keep the information about instruction to be performed.
- SCC: Determines if the condition codes are set.
- DEST: Select one of the 32 registers as destination of the result of the operation.
- SOURCE 1: Hold the source register used in the operation.
- IMM: If it is higher then one, it represents the immediate value used instead of register, otherwise, indicates that SOURCE2 has a special function.
- SOURCE 2:
  - If IMM = 0: The five least significant bits indicate the second source register.
  - if IMM = 1: Express a sign extended 13-bit constant.

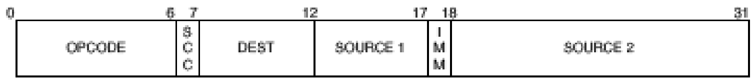


Figure 2.7: Instruction format

2.2.4 RISC-V

RISC development began to experience healthy growth and innovation in the 1980s, becoming a key area of focus due to its groundbreaking concepts. Then, universities like Stanford and Berkeley decided to pursue research in this field, but also major companies like IBM believed in RISC's potential. However, when new improvements stagnated, it could underperform compared to competitors, causing its impact to diminish and receive less attention in the broader context.

Arm and Sun (now part of Oracle) have been the main players in the RISC scenery since the 1980s. In particular, Arm, which stands for Advanced RISC Machines, focused its research on developing low-costs and low-power processors to establish itself as the main solution for portable devices and embedded systems. On the other hand, SPARC architecture was developed to power computer workstations and servers as happened for Sun-4. However, despite their contributions, each of these architectures faced challenges in maintaining their competitive edge, particularly as technology evolved and new competitors emerged.

After several years, in 2008, began a new five-year project based on RISC architecture to advance parallel computing, the Par Lab. This project was based at Berkeley under the supervisor of Professor Krste Asanović and Professor David Patterson as chair of the Computer Science Division and carried on by graduate students Yunsup Lee and Andrew Waterman. The interest in this project brought generous funding from Intel, Microsoft, and also the Defense Advanced Research Projects Agency (DARPA). Although, all the projects in the Par Lab were open source using the Berkeley Software Distribution (BSD) license [23]. The purpose of the project has been carried on by the RISC-V Foundation which has worked to build an open, collaborative community of software and hardware innovators based on the RISC-V ISA since 2015.

The foundations of this project are summarized inside the instruction set architecture (ISA)[24], the abstract model where all the information required to use a CPU is defined. First of all, the purpose and the principles of their work are declared, and then a technical review is presented. It is important to highlight the decision to keep the standard open, allowing both scientists to work free of charge and both companies to develop their versions without fees.

Moving on the technical side, the member of the project settled to design a small general-purpose ISA that could support up-to-date features such as the revised 2008 IEEE 754 floating-point standard and both 32-bit and 64-bit address space variants. Moreover, their architecture should support both 32-bit, known as RV32, and 64-bit, known as RV64, address space variants and efficient dense instruction encoding to address performance, power consumption, and code density issues. In addition to these features, also user-level ISA extensions had to be guaranteed[25].

The register set consists of 31 general-purpose registers holding fixed-point 32- or 64-bit values (from x1 to x31), while the register x0 is fixed to 0. Floating-point values can be expressed with single or double precision and are stored inside 32 registers f0-f31 64-bit wide. In addition, the *pc* register holds the address of the current instruction, while *fsr* holds the address of the current instruction.

As mentioned earlier, RISC-V is designed to work with 32-bit instructions, but it also supports both compressed instructions and extensions, provided that are aligned on 16-bit. The basic instructions can be grouped in six sets:

- R-type: has 2 source registers and an optional 10-bit function field. Integer computational instructions are part of this group.

- R4-type: the only has 3 source registers, then only five bits are available for the function field. This format is only used by some floating-point instructions.
- I-type: instead of a second source register, an 11-bit immediate value is provided, then only three bits are reserved for function field.
- B-type: it is similar to I-type group, but differs about the immediate encoding. Conditional branches follow this encoding.
- L-type: only the destination register is specified, while the others bits are dedicated to the upper immediate to be loaded. In fact, *lui* is the only instruction with this format.
- J-type: the main field holds the target address stored as the offset from program counter. In fact, this format encoded the unconditional jumps.

In the table 2.1 is shown how the 32-bit are filled according to the organization of the micro-architecture.

RISC-V is a byte-addressable load-store architecture so only Load and Store instructions can access memory, while other instructions have to use CPU registers. In particular, Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format, and stores are B-type. The effective byte address is obtained by adding register *rs1* to the sign-extended immediate. Loads write to register *rd* a value in memory. Stores write to memory the value in register *rs2*.

31 27	26 22	21 17	16 12	11 10	9 7	6 0	
rd	rs1	rs2	funct10			opcode	R-type
rd	rs1	rs2	rs3	funct5		opcode	R4-type
rd	rs1	imm [11:7]	imm [6:0]		funct3	opcode	I-type
imm [11:7]	rs1	rs2	imm [6:0]		funct3	opcode	B-type
rd	LUI immediate [19:0]					opcode	L-type
jump offset [24:0]						opcode	Jtype

Table 2.1: RISC-V base instruction formats

2.3 Parallel Ultra Low Power Platform

In the wake of RISC-V growing, the Integrated Systems Laboratory (IIS) of ETH Zürich and the Energy-efficient Embedded Systems (EEES) group of the University of Bologna started together to explore and develop new and efficient computing architectures based on the RISC-V open ISA. The Parallel Ultra Low Power (PULP) Platform was born in 2013 with the aim of realize low-power devices, but it has now realizing high-performance devices and also multi-core systems. In less than ten years PULP became one of the most well-known open-source projects worldwide, as proved by more than 50 ASICs realized and the successful projects stemmed from this work like Ariane or Zeroriscy [26].

An overview of the environment is proposed in picture 2.8 where it is possible to distinguish the following main section:

- Core: this is the hear of each system and specific solutions are available, even a 64-bit version known as Ariane.
- Peripherals: this group includes the interfaces towards the outside world enabling both communication and control of the device.
- Interconnect: different kinds of solution are implemented according to the aim, for example reach a custom module or address a peripheral.
- Platform: assembling the previous components with a proper memory system it is possible to build from single core to Multi-cluster implementation.
- Accelerator: this component is not mandatory, but nowadays is one of the most useful part.

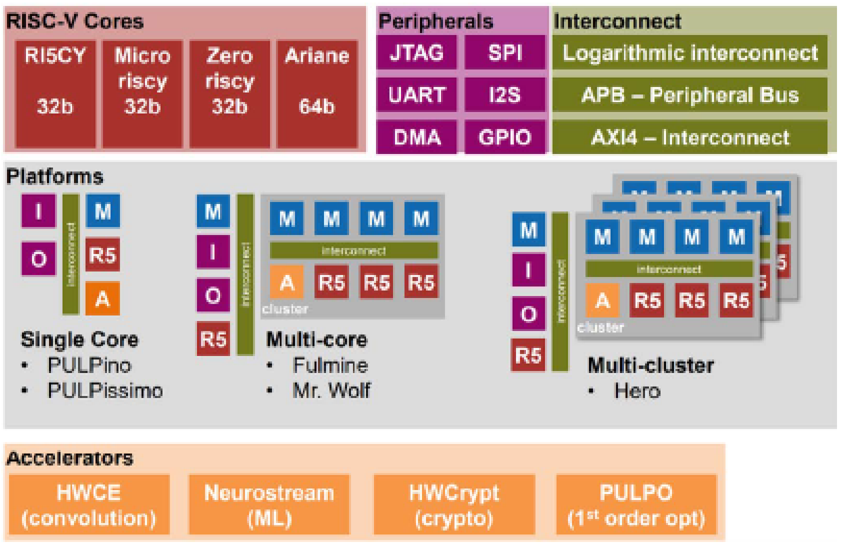


Figure 2.8: PULP platform overview

The choice of the right processor is strictly related to the purpose of the system. In fact, RI5CY, also known as CV32E40P, is a four-stage 32-bit core supporting instruction set extension for digital signal processing operation and a 32-bit floating point unit used to enhance performance. On the other hand, Zero and Micro riscy are aimed at achieving a minimal area. Ariane was designed to reach a critical path length of about 20 gate delays.

The simplest PULP-based systems are single-core microcontrollers called PULPino and PULPissimo. Cluster-based systems are implemented to increase performance and basically consist of some cores and memories, but also a SoC that houses a larger second-level memory, peripherals for input and output, and a complete PULPissimo microcontroller for power management and basic operations. The most powerful solution is implemented through a compound of clusters connected to a regular computing node.

An in-depth look at the PULPissimo microcontroller is provided to highlight the platform 's typical features. First of all, PULPissimo is a single-core microcontroller, so it combines all the necessary elements of a microcomputer system onto a single piece of hardware so the architecture as shown in figure 2.9 includes:

- Main core like RI5CY or Micro riscy (Ibex).
- Micro Direct Memory Access controller (uDMA) to make the Input/Output system autonomous.
- Memory subsystem.
- Simple interrupt controller.
- Peripherals.
- Support for Hardware Processing Engine (HWPE).
- Software Development Kit.

On one side, RISCY is an in-order, single-issue core and thanks to a four-stage pipeline stage it can reach about one instruction per clock cycle. It supports the base integer instruction set (RV32I), compressed instructions (RV32C), and multiplication instruction set extension (RV32M). In addition, it can be configured to have a single-precision floating-point instruction set extension (RV32F). It implements several ISA extensions such as hardware loops, post-incrementing load-and-store instructions, bit-manipulation instructions, MAC operations, support fixed-point operations, packed-SIMD instructions, and the dot product. It has been designed to increase the energy efficiency of ultra-low-power signal processing applications.

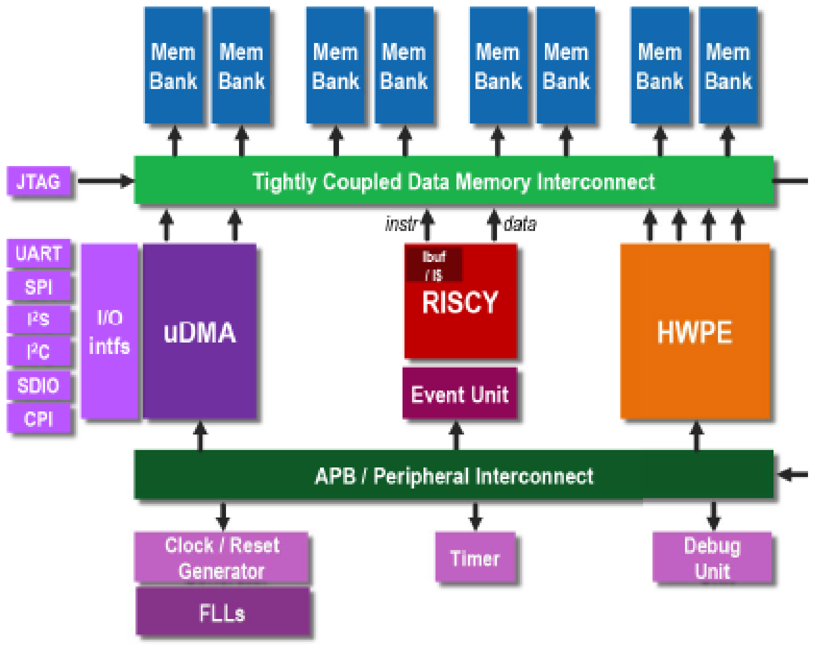


Figure 2.9: PULPissimo architecture [27]

On the other side, Ibex, formerly Zero-riscy, is an in-order, single-issue core with two pipeline stages. It has full support for the base integer instruction set (RV32I version 2.1) and compressed instructions (RV32C version 2.0). It can be configured to support the multiplication instruction set extension (RV32M version 2.0) and the reduced number of registers extension (RV32E version 1.9). Ibex was originally designed at ETH to target ultra-low-power and ultra-low-area constraints.

2.3.1 Software development kit

From the software point of view, it is crucial to have the possibility to test the entire hardware implementation with powerful debug tools and to save the significant amount of time needed by the syntheses. This scope is even more urgent when the main actor is a lightweight and flexible micro-controller as PULPissimo in order to break the speed and design effort bottlenecks. Among the several solutions powered by the PULP platform summarized in figure 2.10, the most powerful solution is represented by the Software development kit distributed through a GitHub repository at [29]. It provides tools, libraries, and APIs for writing, compiling, and debugging software on various PULP-based chips. The organization is schematized in figure 2.11 where the three steps of the process are highlighted.

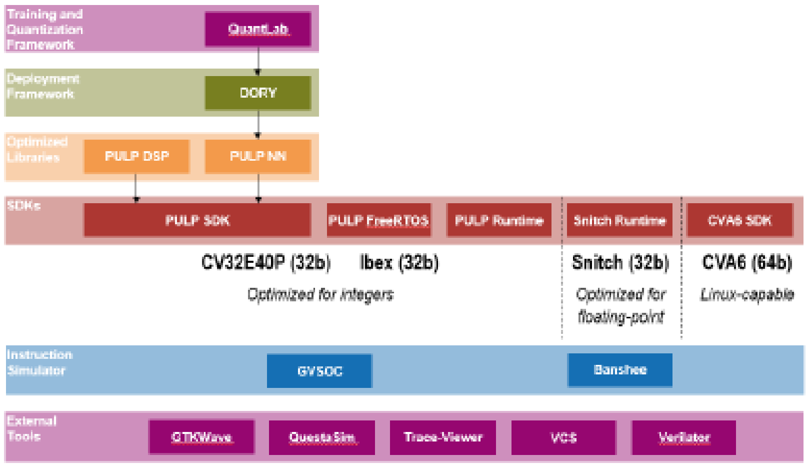


Figure 2.10: PULP software tools [28]

Firstly, the toolchain itself should be configured according to the desired platform through the use of JSON configuration files, which define the specific parameters and settings for the target architecture. Once configured, the toolchain must be built in order to generate all the necessary tools based on the RISC-V ISA for the development process, including compilers, linkers, and libraries tailored to the specific PULP-based platform.

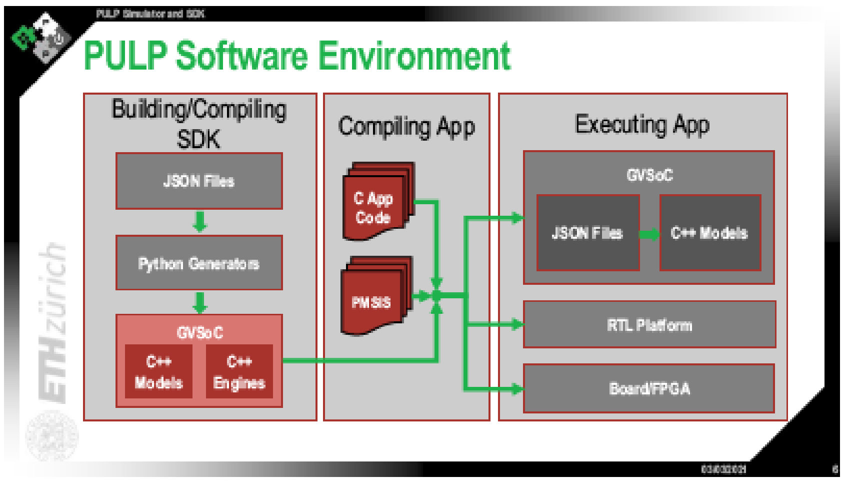


Figure 2.11: PULP SDK organization [28]

The heart of the SDK engine is the Generic Virtual System on Chip (GVSOC), an open-source simulator specifically designed for simulating PULP architectures.

GVSOC is capable of simulating complex full-platform systems, offering developers a virtual environment that accurately mimics the behavior of the hardware. As noted by the developers in [30], the primary advantages of this simulator lie in its ability to provide a highly configurable and timing-accurate simulation, utilizing an event-driven model. This approach ensures that the simulator can capture the intricacies of hardware execution in a manner that is both realistic and flexible.

GVSOC achieves this by leveraging both Python and C++ in its architecture. Python is primarily used for the high-level configuration and control of the simulation environment, while C++ handles the low-level, performance-critical parts of the simulation, ensuring efficient execution of the simulation process. This combination allows developers to fine-tune and customize the simulation according to their needs while maintaining the necessary performance for accurate event-driven simulations.

The process of executing a program in this simulated environment involves translating the C code into instructions that the PULP platform can execute. This is accomplished through the PMSIS (PULP Microcontroller Software Interface Standard), which provides a set of low-level drivers that facilitate communication between the software and the hardware or, in this case, the simulated platform. PMSIS acts as the bridge that allows developers to write high-level code, which can then be translated into instructions for the PULP architecture.

Once the code is translated, it is executed within the GVSOC environment. In addition to providing simulation capabilities, GVSOC also includes a debugging interface that allows developers to step through their code, set breakpoints, and monitor the behavior of their applications. This integrated debugging environment is crucial for identifying and resolving issues early in the development process, ensuring that the software runs correctly when eventually deployed on actual hardware.

If only a standard simulation of the implementation is needed, using the stand-alone PULP routine would be a time-saving alternative. This solution can turn out to be effective for situations like the simulation of a simple application implemented on the PULP platform. Unlike the SDK toolchain, this tool mainly relies on the GNU compiler.

2.4 Spiking Neural Network

In this era dominated by remarkable progress in the vast field of Artificial Intelligence (AI) energy efficiency is one of the most relevant concerns. As stated in [31] larger models translate to greater computing demands and, by extension, greater energy demands. Indeed, on one hand, deep learning models succeeded in various aspects of an ordinary and extraordinary life, from video games to medical tasks. On the other hand, the trade-off between power consumption and performance still elevates the human brain as the best player in computational effort. Thus, the most reasonable conclusion is to be inspired by biology to realize the most efficient way to perform computation, in other words, Neuromorphic computing[32].

2.4.1 Biological network model

The way used by the body to sense the external world is by imitated creating artificial sensors inspired by biological sensors like retina or cochlear, as outlined in 2.12. The eclectic signal generated from the sensor in response to an event is known as a spike and carries the information. Then, the observation of phenomena can be coded through a spike signals train, where the single bit represents if an event happens, so the bit is equal to 1, or not. The models developed with this approach are called Spiking neural networks and, according to the spike characteristic, are focused on the evolution over time despite the intensity. From the hardware point of view, using a spike means having less data movement, which can lead to increased power and reduced latency concerning the same task on conventional hardware.

Once understand the general idea behind the word SNN, this section will be presented a brief overview of the actual model of SNN. One of the main actors in the human brain is the spike signal, which is an electrical impulse, known as action potentials, of approximately 100 mV in amplitude. These signals move around the body thanks to the nerve fibers, also known as *axons*, that are long, slender projections of a nerve cell. At the end of the link, the signals are received by the cell body, a bulbous, non-process portion of the neuron called *soma*.

This mechanism is implemented in an all-or-nothing way using only one bit and weight in order to make the routing and computation task easier. In spite of using one bit instead of continuous values, SNNs are highly different from binary networks. Indeed, the spike timestamp is used to transfer information and can be implemented using clock signals that are already distributed across a digital circuit. This approach shows one great gain concerning the memory occupation. In fact, the biological neurons are rarely activated, while they rest for most of the time, leading to a large amount of zeros between two ones. Thus, it is much more

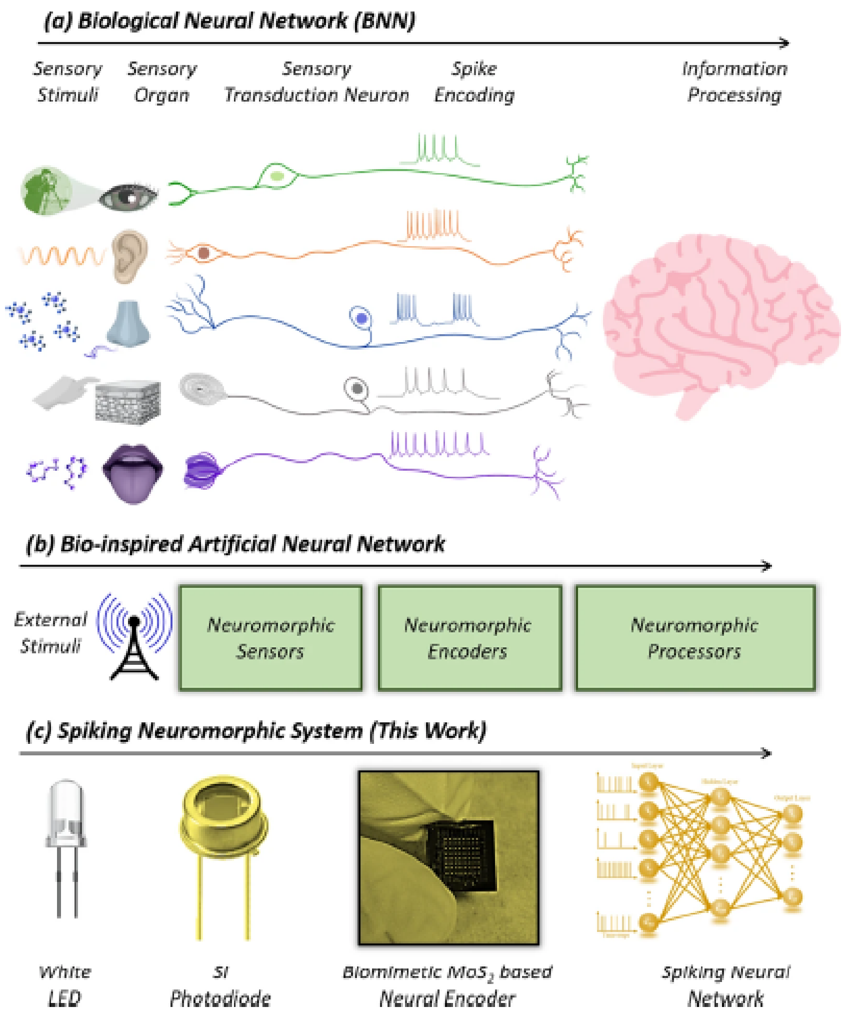


Figure 2.12: Schematic of biological neural network[33]

convenient in terms of memory occupation, and then cost, to store the activation information through the moment in which it happens assuming that in the other instants, there are no events. This model results in a fast response system capable of suppressing static events.

2.4.2 Neuron model

In order to translate the interactions that happen inside the human body into precise electrical circuits a mathematical model is needed. The leaky-integrate-and-fire (LIF) model provides a first approximation of how the membrane potential behaves. The purpose behind this model developed by Louis Lapique is to treat the membrane of the neuron as a leaky capacitor, in other words, an ideal capacitor with resistance in parallel. To emulate the input signals coming from the synapses it is used a current source, the entire starting circuit is depicted in figure 2.13. Thus,

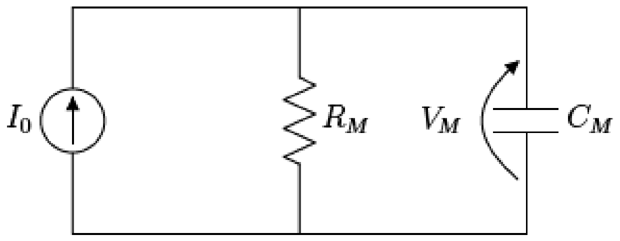


Figure 2.13: Leaky integrate and fire equivalent circuit

as a consequence of a spike pulse train, the capacitor can increase its potential over a certain threshold only if the period between the signals is short enough. This behavior likely simulates how the membrane potential is overcome when the number of incoming signals is sufficiently high.

Intending to making the mathematical analysis easier without changing the final result, it is possible to consider the capacitor as ideal removing the resistance as in figure 2.14. Thus, it is immediate to write the equation of this circuit 2.1a, but can be also viewed from the Voltage point of view as in 2.1b highlighting the evolution along the time.

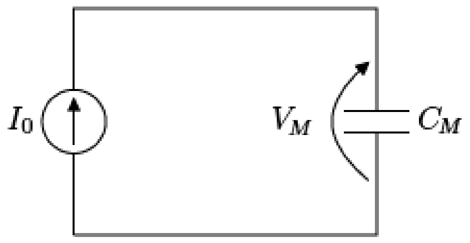


Figure 2.14: Integrate and fire equivalent circuit

$$I_{(t)} = \frac{dV_{(t)}}{dt} C_M \tag{2.1a}$$

$$V_{(t)} = \frac{1}{C_M} \int_{t_0}^{t} I_{(t)} \mathrm{d}t \tag{2.1b}$$

From the equation describing the voltage change across time it is possible to see how applying a constant current, the response is expected to increase linearly until the threshold is reached and then it is reset.

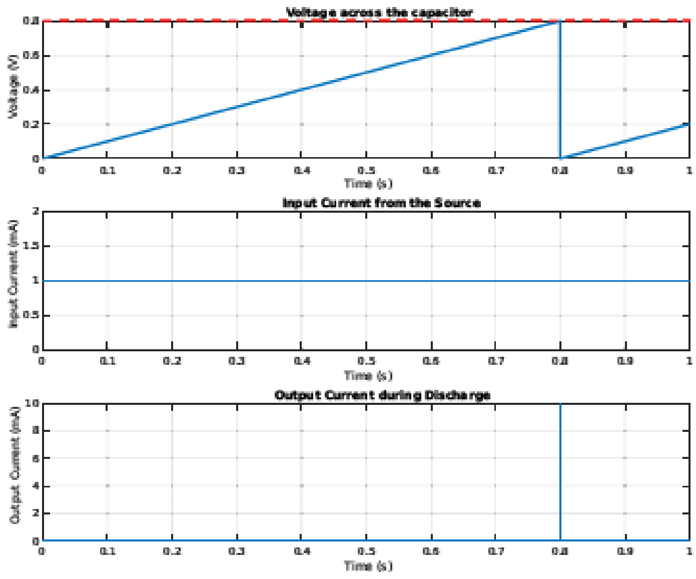


Figure 2.15: Ideal behavior of the integrate and fire circuit with constant current

This description depicted in figure 2.15 can represent a realistic situation for the discharging phase, but it is not suitable due to the input current behavior. In fact, the most accurate kind of signal to describe the input synapses is a spike impulse. The reaction of the membrane when the potential reaches the threshold is shown in figure 2.16. In this scenario the input current $I_{in(t)}$ is the sum of all the signal coming to the neuron, while the output current $I_{out(t)}$ represents the eventual response at the stimuli.

Thus, it is possible to focus on the Leaky integrate-and-fire model to have a better representation of the neuron. In fact, the figures 2.17 shows how the currents and voltages change when a leakage, emulated by a resistance, is considered in the model. Thus, the model represented in the figure 2.18 obtained with a spiking current, can adapt the artificial neuron to the the membrane potential that decreases when it is not stimulated.

The LIF neuron is the most widely used spiking neuron model in the context of deep learning. This popularity stems from its balance between biological realism and computational simplicity, making it efficient for simulating neural activity while maintaining manageable computational costs. However, the LIF neuron is just one model among many available in the family of spiking neuron models, each

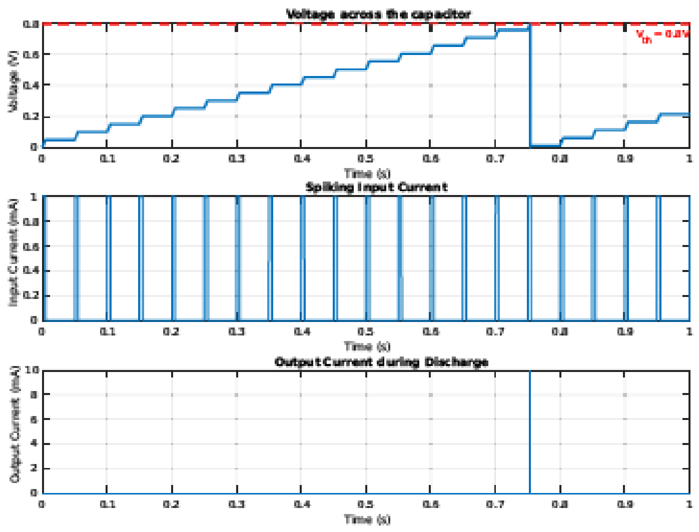


Figure 2.16: Ideal behavior of the integrate and fire circuit with spike current

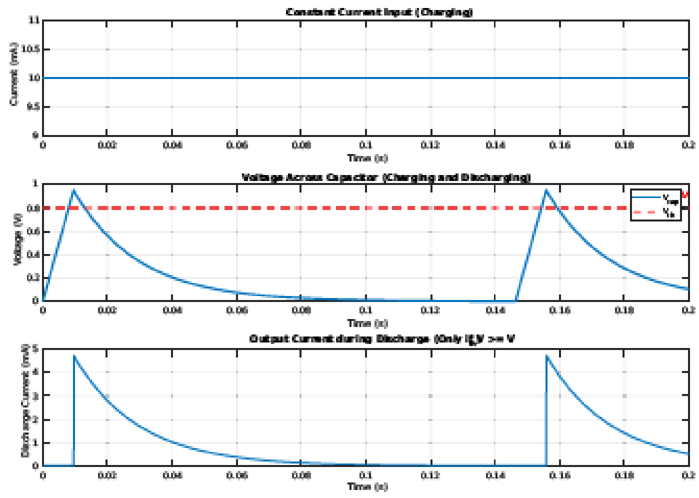


Figure 2.17: Output current with constant current source

with different characteristics and capabilities.

Other models can be employed to fulfill the needs of specific neural network tasks, depending on the desired level of biological realism, complexity, and the network's functional requirements. For example, the Integrate-and-Fire (IF) neuron

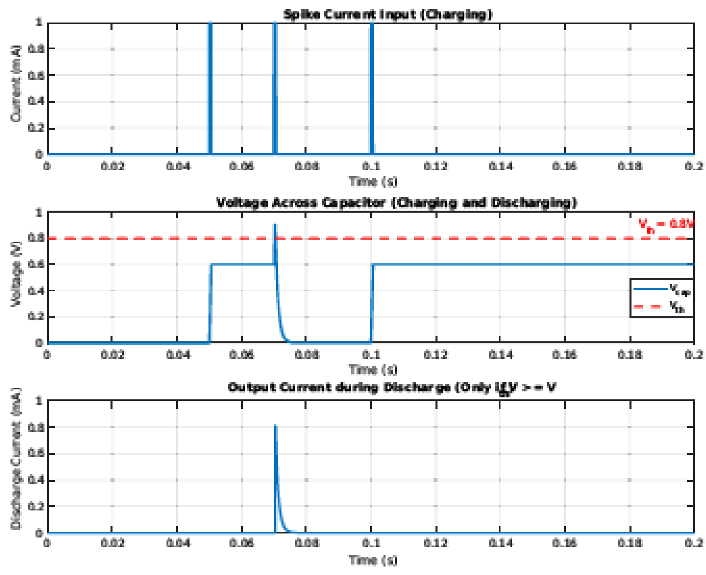


Figure 2.18: Output current with constant spike source

is a simpler variant that excludes the leaky term, making it even less computationally demanding, though at the expense of less biological accuracy. On the other end of the spectrum, more complex models like the Hodgkin-Huxley (HH) [34] neuron model simulate the detailed ionic mechanisms underlying action potential generation, making it highly accurate for biological neural activity but far more computationally intensive.

Additionally, the Izhikevich model [35] offers a middle ground between biological realism and computational efficiency. It is capable of reproducing a wide variety of spiking behaviors seen in biological neurons, such as bursting, regular spiking, and fast spiking, while remaining computationally lighter than the Hodgkin-Huxley model. Similarly, the FitzHugh-Nagumo [36] [37] model provides a simplified representation of neuronal activity, capturing essential features of spiking and excitation without requiring the full complexity of ionic channel dynamics.

In summary, while the LIF neuron is commonly used due to its simplicity and efficiency in the deep learning context, alternative spiking neuron models like the IF, Hodgkin-Huxley, Izhikevich, and FitzHugh-Nagumo models offer varying levels of complexity and biological fidelity, allowing neural networks to be tailored to the specific demands of different tasks and applications.

Chapter 3

Related work

In recent years, there has been a rapidly growing interest in developing low-power embedded systems capable of efficiently handling deep-learning computations. This surge is driven by the increasing need for AI and machine learning applications to be deployed in resource-constrained environments such as IoT devices, mobile platforms, and edge computing systems. Traditional deep-learning models require significant computational power and energy, which is not feasible for embedded systems that often operate on limited battery life and processing capabilities. As a result, researchers and engineers are focusing on optimizing neural network architectures and developing energy-efficient hardware accelerators to reduce the computational complexity and power consumption of deep-learning tasks on embedded platforms.

One of the first efforts in achieving a hardware accelerator IP for Neural network was presented at the 2018 Embedded System Week by Conti et al. [38] which earned them the Best Paper award. That work aimed to design and Binary neural network accelerator, the *XNOR Neural Engine*, tightly integrated within a microcontroller and evaluate its integration in a simple, yet powerful, microcontroller system. The IP block, even if compact, is designed to overcome several limitations typically associated with software-based implementations. One of the key challenges in software-based implementations is the high storage requirements, which stem from the need to handle large amounts of data, weights, and intermediate results during computation. However, by incorporating optimized hardware buffers within the IP block, data management is handled more efficiently, significantly reducing the overall storage requirements. These buffers allow for faster data access and minimize the need to constantly read and write data from external memory, which can be slow and power-hungry. Additionally, software-based implementations often require the addition of complex instructions to the ISA, not needed for a hardware accelerator.

Another key component actively explored by researchers to achieve optimized management of resources and data in embedded deep-learning systems is the microcontroller. The growing demand for studying more complex phenomena - such as vibrations, audio signals, video processing, or biological data - has led to the necessity of handling larger datasets and performing more intricate computations in real time. Traditional methods, where computations were centralized in a network-based solution, are increasingly inadequate due to the sheer volume of data being transmitted. These centralized approaches require sending raw data from edge devices to a central server or cloud platform, leading to excessive bandwidth usage, higher latency, and potential bottlenecks in communication.

A more effective approach to addressing this problem involves shifting the computation towards the "edge" - moving data processing closer to where the data is generated, at the end nodes, such as sensors or microcontrollers. In this scenario, the key challenge is to retain the low-power characteristics of a microcontroller while simultaneously handling the heavy computational workloads typically managed by digital signal processors or parallel processors.

To address this issue, Pullini et al. developed Mr. Wolf [39], a state-of-the-art microcontroller that integrates a fabric controller powered by a highly programmable parallel processing engine, specifically designed for flexible multi-sensor data analysis. The system-on-chip (SoC) architecture relies on a power-optimized processor and an I/O subsystem, meticulously designed to efficiently and rapidly move large volumes of data between components. This careful design ensures that data can be processed with minimal delays and energy consumption. Additionally, the system features eight fully programmable processors that are equipped with digital signal processing (DSP) extensions and a shared floating-point unit (FPU), allowing it to handle complex computational tasks - such as signal filtering, image processing, or machine learning inference - while maintaining low power consumption.

Additionally, the use of Spiking Neural Networks has sparked significant interest in the field of edge computing applications due to their low power consumption and sparse data processing characteristics. One implementation that covers several aspects already discussed in this chapter is in given in [7]. In this review Barocci et al. present an integration of the *ReckOn* digital neuromorphic processor with the PULPissimo RISC-V single core microcontroller to enable edge IoT applications. The choice of using *ReckOn*, one of the open-source spiking recurrent neural network processors developed by Frenkel and Indiveri [40], lies in a greater energy efficient joined with a higher number of neurons and synapses per core.

Chapter 4

Preliminary work

Moving on to the practical aspect of this thesis, the first step in implementing the framework is to outline the workflow. In this type of activity, it is essential to verify, step by step, that each component can effectively interact with the others. It is also crucial to maintain a comprehensive view of the entire development process so that incompatible components can be identified and replaced as needed. The strategy applied consisted of the following steps:

- Evaluating of the available choices in terms of working platform and developing boards.
- Building a dummy version of the framework to test the components compatibility.
- Powering the device with the hardware accelerator.

Before beginning the hands-on work, the PULP platform was selected to host the framework, given that its implementation was feasible with some of the FPGAs available at the university laboratory. Within the PULP environment, several options were considered, but the most reasonable choice needed to be both stable and capable of delivering acceptable performance, particularly for AI applications. As a result, the optimal trade-off appeared to be the PULPissimo microcontroller. The next step involved verifying its compatibility with the available board.

As stated in the GitHub repository of PULPissimo [41], inside the *target/fpga* section, there are ready-made scripts for Synthesis and Implementation for Xilinx Vivado for the following development boards:

- Digilent Genesys2
- Xilinx ZCU104

- Xilinx ZCU102
- Xilinx VCU108
- Digilent Nexys Board Family
- ZedBoard

Considering that the ZCU102 board has all the features needed to host the framework, it appeared to be the best choice. After the preliminary conclusions regarding the working environment, it is possible to move to fieldwork programming the FPGA.

4.1 FPGA programming

First of all, it is useful to briefly introduce the development board, the Zynq UltraScale+ MPSoC ZCU102 shown in picture 4.1.

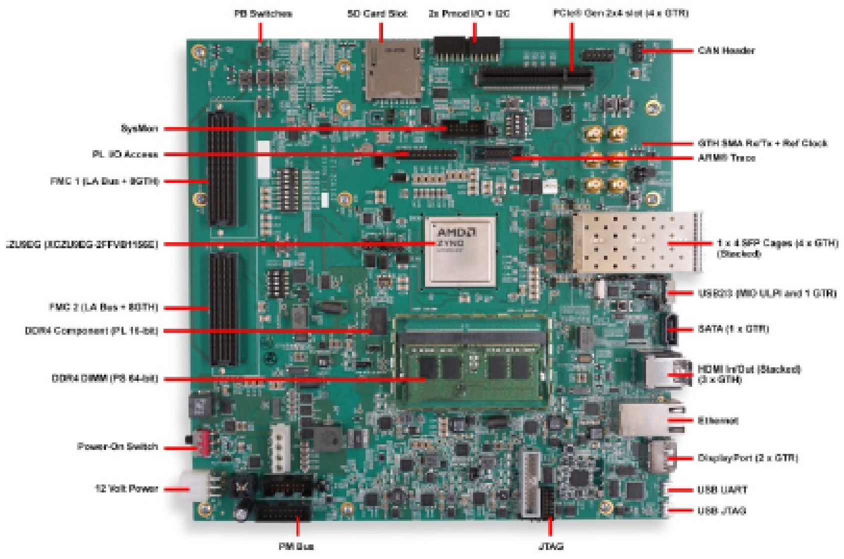


Figure 4.1: ZCU102 [42]

4.1.1 ZCU-102

This board with the rest of the Zynq line has been produced by the historic company Xilinx until 2022 when AMD completed the acquisition, then all the products are branded under AMD. The Zynq™ UltraScale+™ provides a Multi-processor System on Chip (MPSoC) with a powerful PS and user PL into the same device.

The PS features a quad-core or dual-core processor as an Accelerated processing unit (APU), a dual-core real-time processor (RPU), and a graphical processor unit (GPU).

In particular, the ZCU102 whose block diagram is shown in 4.2 is powered by the XCZU9EG-2FFVB1156E MPSoC as concerns the PS side, while the PL can exploit up to 600 hundred of LUTs. The board configuration is carried out by onboard JTAG, through USB, ARM 20-pin header, or PC4 header, but also SD card and QSPI can be used for programming. In addition, communication protocols like UART and I/O ports shuch as HDMI, SATA, and Ethernet are integrated with the board. The PL can be accessed and then programmed using the software suit

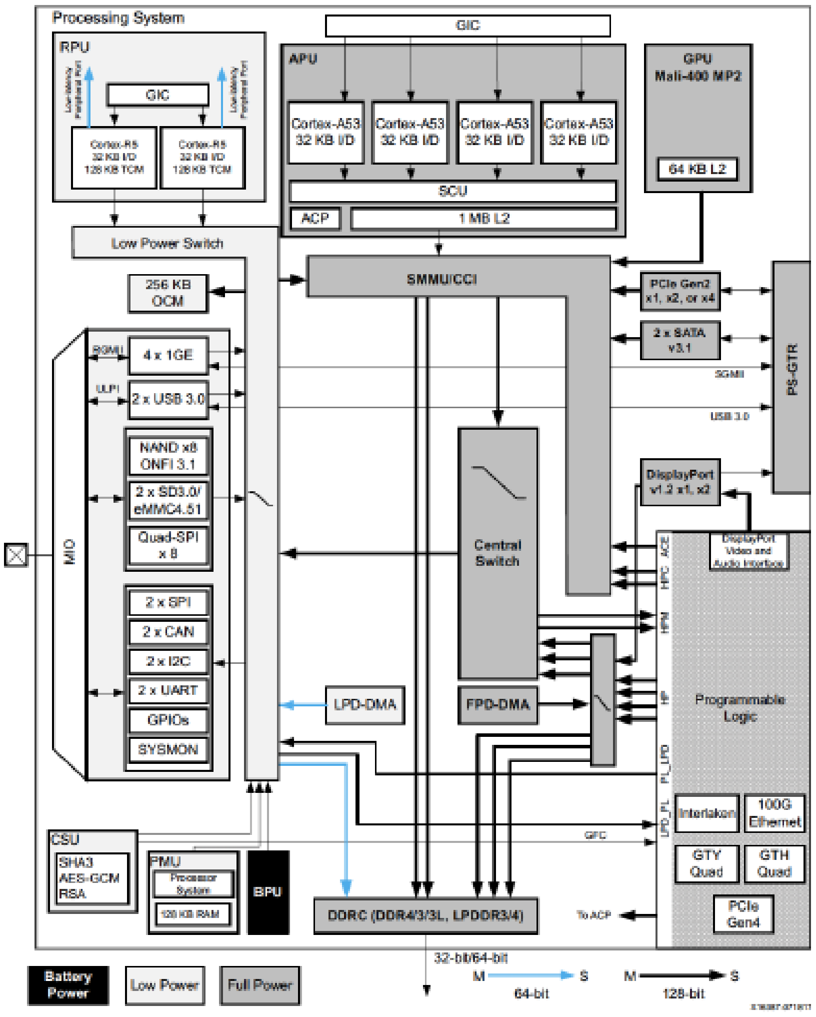


Figure 4.2: ZCU102 block diagram

Vivado, developed by AMD. This powerful tool allows to synthesis and analysis of hardware description language (HDL) devices thanks to a toolchain that converts C code into programmable logic. The suite has four components:

- Vivado High-Level Synthesis compiler: Enables C, C++ and SystemC to be used inside Xilinx devices without the need to manually create RTL.
- Vivado Simulator: It is a compiled-language simulator that supports mixed-language, TCL scripts, encrypted IP and verification.
- Vivado IP integrator: allows to quickly integrate and configure IP.
- Vivado TCL store: can be used for developing add-ons to Vivado.

To work with Vivado suite it is possible to download on your own computer with GNU-Linux or Microsoft operating system, even if it is easier to exploit scripting inside the Linux environment. In addition, it is important to clarify that the evaluation boards contain a license that is necessary to download the enterprise version, on the contrary, the license has to be bought.

4.1.2 PULPissimo repository

A briefly overview of the PULPissimo hierarchy is given to introduce the structure of the platform and in particular the SoC is detailed. The module *pulpissimo.sv* is the

```
pulpissimo.sv
 |__pulp_domain.sv
 |   |__pulp_soc.sv
 |       |__fc_subsystem.sv
 |       |   |__fc_hwpe.sv
 |       |__soc_interconnect_wrap.sv
 |       |   |__soc_interconnect.sv
 |       |       |__interleaved_crossbar.sv
 |       |       |__contiguous_crossbar.sv
 |       |__dmi_jtag.sv
 |       |__tcdm_arbiter_2x1.sv
 |       |__apb2per.sv
 |__padframe_adapter
 |__addr_decode
 |__rst_gen
 |__clk_gen
```

Figure 4.3: PULPissimo project organization

default top-level entity of PULPissimo used as the basis for ASIC implementations and RTL simulation. It instantiates the clock generation and pad multiplexing IP as well as the *soc_domain* (which wraps almost the entire logic of pulpissimo from the external pulp_soc repository) it is instantiated through the *pulp_domain* module. In addition, also the default features of the microcontroller are defined:

- *CORE_TYPE*: define which core using
  - 0: CV32E40P (default)
  - 1: IBEX RV32IMC (formerly ZERORISCY)
  - 2: IBEX RV32EC (formerly MICRORISCY)
- *USE_XPULP*: default 1, means that extension is enabled.
- *USE_FPU*: only for CV32E40P, include the FPU inside the design.
- *USE_HWPE*: enable the Hardware processing engine.

Inside the top-level entity, *pulp_domain* module is instantiated and all the connections are made, while it is described separately in *pulp_domain.sv*. This module aims to tie off unnecessary signals and expose only the signals required for PULPissimo, then *pulp_soc* retrieved from another repository is instantiated.

The module *pulp_soc* sketched in 4.4 is the actual heart of the PULPissimo SoC and contains the key modules inside the architecture:

- *fc_subsystem*: includes all the IP related to the fabric controller and particularly the HWPE [38] that will be covered in 4.2.
- *soc_interconnect_wrap*: encloses the *soc_interconnect* module, but also the interfaces to do connect memory, core and peripherals.
- *dm_top*: Top-level of debug module. This is an AXI-Slave.
- There are also an APB-to-Peripheral interconnect protocol adapter and Tightly Coupled Data Memory (TCDM) multiplexer.

Inside *soc_interconnect_wrap* there are Advanced extensible interface AXI and TCDM buses, but the main module is *soc_interconnect*. This module handles the different memory regions but also peripherals. The first stage of interconnection is a demultiplexer dedicated to *L2* memory, where the first slave port routes to the *axi crossbar*, the second slave port routes to the contiguous crossbar and the third slave port connects to the *interleaved crossbar*.

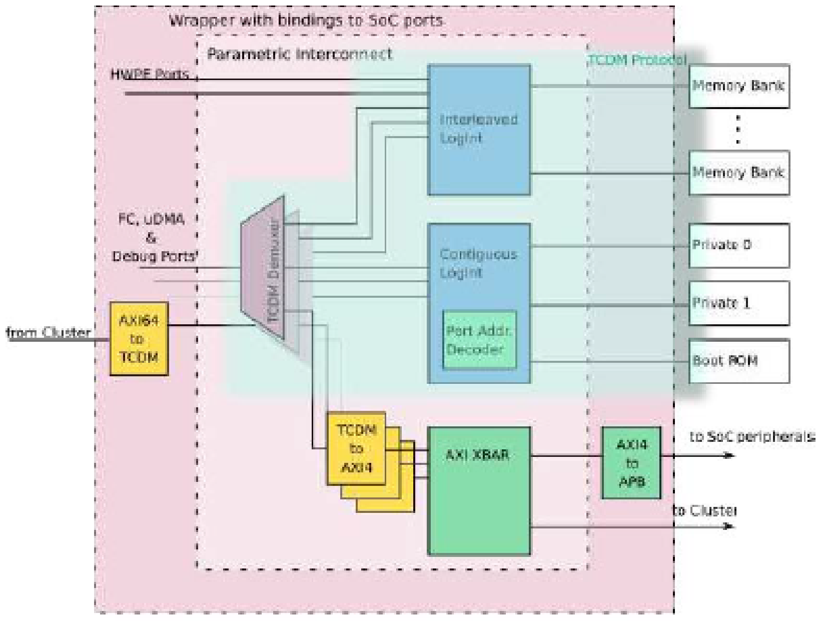


Figure 4.4: PULP System on Chip [43]

The AXI crossbar accepts a set of address-to-slave port mapping rules and decodes the transaction address accordingly. Illegal addresses that do not map to any defined address space are answered with a decode error and Read Responses contain the data '0xBADCAB1E'. A *continuous crossbar* is a fully connected *bus* with combinational arbitration (logarithmic Interconnect). Internally, there is an address decoder that matches each master port address against a number of address range to output port mapping rules. Addresses not matching any of the address mapping rules will end up on a default port that always grants the request and in the case of a read access, responds with the word '0xBADACCE5'. Interleaved Crossbar is a fully connected crossbar such as the contiguous one. It arbitrates from the master ports from the L2 demultiplexer and the interleaved-only master ports (ports that do not have access to the other address spaces) to the TCDM slaves with address interleaving. That is, the least significant bits of the address are used to select the slave port. This results in a more equal load on the SRAM banks when the master sequentially accesses memory regions. Slaves that cannot respond within a single cycle must appropriately delay the assertion of the *gnt* (grant) signal.

4.1.3 Environment setup

The primary goal is to integrate the PULP microcontroller into the FPGA, and there are two approaches available for this purpose. The first approach is the simpler option, as it involves downloading the bitstream from the repository and loading it into the Programmable Logic. This method provides a ready-to-use version of the environment for running programs but does not allow modifications, making it unsuitable for implementing a hardware accelerator. The second approach, known as the full flow, is detailed in the repository's README page and involves locally building the sources before synthesizing the device. Although this approach is more complex, it allows for full customization of the device, including the addition of custom modules and configuration changes. The integration of new intellectual properties (IPs) is streamlined by using Bender, as described in Appendix A.

To begin, one can either download a release version of PULPissimo or fork and clone the most up-to-date version. The PULP platform provides Makefile targets to facilitate standard tasks such as building, synthesis, and bitstream generation. Once the process is complete, the bitstream file for the JTAG configuration of the FPGA can be loaded using Vivado's hardware manager. The purpose of the Makefile target is to verify the correct version of each IP required by the top-level module and then traverse the hierarchy to resolve all dependencies. This task is managed by the Bender software, which relies on independent GitHub repositories to ensure that source files are either up-to-date or properly archived.

Alternatively, the simulation platform for the PULPissimo platform can also be built to conduct simulations. This can be achieved using the Makefile targets, but it requires simulation tools such as Questasim or Xcelium. Similar to the FPGA process, the source files must first be verified and updated if necessary, based on the standard files associated with each IP. Once this verification is complete, the simulation platform can be built, leveraging the scripts provided by Bender to streamline the process.

4.2 Hardware designing

Once understood how the FPGA can be programmed to host the PULP platform it is time to customize PULPissimo with, at first, a dummy module to understand the integration process as suggested in [43]. First of all, a GitHub repository can be the right choice to keep the source under control and, at the same time, exploit the powerful features provided by Bender. The repository can be organized as in figure 4.5, the purpose of the modules inside the *rtl* directory is just to notify the success of the integration procedure. Thus, the top-level entity can be instantiated

```
dummy_ip/
├── rtl/
│   ├── dummy_top.sv
│   └── dummy_submodule.sv
├── include/
│   ├── dummy_header.svh
│   └── dummy_ip/
└── Bender.yml
```

Figure 4.5: Repository organization

within the *pulp_soc* module without requiring any special connections. Within the sub-module, common cells are instantiated to ensure the proper declaration of dependencies, as managed by Bender software according to the configuration file *Bender.yml*. Finally, the hardware can be verified through software simulation, as detailed in Section 4.3. Although this initial approach is relatively straightforward, it provides valuable experience in managing IPs, particularly with Bender. Additionally, the integration process clarifies the role of key components in PULPissimo, such as the *pulp_soc* module.

4.2.1 Wide ALU example

After the first integration experience it is possible to increase the level of the training in order to realize a realistic but simple module, following the indications presented in [44]. The purpose of this module is to accept two numbers up to 256-bit wide and perform some arithmetic or logic operation. In this example is needed to move data in order to reach the hardware module, so an AXI bus is used to perform the link. In addition, the tool *regtool.py*, detailed in the appendix B, is used to describe the data structure required by the module instead of manually connecting each port.

First of all, inside the *pulp_soc* module an additional AXI bus is instantiated an then connected on the *axi_slave* port of the *wide_alu_top* IP.

```
// MY WIDE ALU IP
AXI_BUS #(
    .AXI_ADDR_WIDTH ( 32                ),
    .AXI_DATA_WIDTH ( 32                ),
    .AXI_ID_WIDTH   ( AXI_ID_OUT_WIDTH   ),
    .AXI_USER_WIDTH ( AXI_USER_WIDTH    )
) s_wide_alu_bus ();

```

```

wide_alu_top #(
     .AXI_ADDR_WIDTH ( AXI_ADDR_WIDTH    ),
     .AXI_ID_WIDTH   ( AXI_ID_OUT_WIDTH  ),
     .AXI_USER_WIDTH ( AXI_USER_WIDTH    )
) i_wide_alu (
     .clk_i          ( soc_clk_i         ),
     .rst_ni         ( soc_rstn_synced_i ),
     .test_mode_i    ( dft_test_mode_i   ),
     .axi_slave      ( s_wide_alu_bus    )
  );
```

In the *soc_interconnect_wrap* module there is the configuration of the AXI crossbar rules, needed to correctly address the data. In addition, one extra master port reserved to the *wide_alu_slave* is added, then the port is included also inside the *soc_interconnect* module.

```
module soc_interconnect_wrap
  import pkg_soc_interconnect::addr_map_rule_t;
  #(...)(
AXI_BUS.Master wide_alu_slave // MY WIDE ALU IP
  )
  ...
  localparam NR_RULES_AXI_CROSSBAR = 3;
  localparam addr_map_rule_t [NR_RULES_AXI_CROSSBAR-1:0]
  AXI_CROSSBAR_RULES = '{
    '{ idx: 0, start_addr: `SOC_MEM_MAP_AXI_PLUG_START_ADDR,
         end_addr: `SOC_MEM_MAP_AXI_PLUG_END_ADDR},
    '{ idx: 1, start_addr: `SOC_MEM_MAP_PERIPHERALS_START_ADDR,
         end_addr: `SOC_MEM_MAP_PERIPHERALS_END_ADDR}},
    '{ idx: 2, start_addr: `SOC_MEM_MAP_WIDE_ALU_START_ADDR,
         end_addr: `SOC_MEM_MAP_WIDE_ALU_END_ADDR}};
    ...
AXI_BUS #(
     .AXI_ADDR_WIDTH ( 32                                     ),
     .AXI_DATA_WIDTH ( 32                                     ),
     .AXI_ID_WIDTH   ( pkg_soc_interconnect::AXI_ID_OUT_WIDTH ),
     .AXI_USER_WIDTH ( AXI_USER_WIDTH                         )
  ) axi_slaves[3]();

  `AXI_ASSIGN(axi_slave_plug, axi_slaves[0])
  `AXI_ASSIGN(axi_to_axi_lite_bridge, axi_slaves[1])
  `AXI_ASSIGN(wide_alu_slave, axi_slaves[2])
```

The *wide_alu_top* module is used to wrap the input and out signals, but also instantiate the required interfaces between the AXI bus and the registers, and configure the register file defined separately. The core of the IP is the *wide_alu* entity, where an three stages finite state machine (FSM) is exploited to handle the operations.

In addition to the hardware description, this example gives the opportunity to practise also the use of the software level. Indeed, in order to perform the desired operation through the *wide_alu* a driver in needed and has to be compiled within the main application.

4.3 Software developing

The software side of the implementation can be achieved by exploiting the C language and the header file produced by *regtool* to easily address the memory mapped-peripherals. Firstly, inside the *main.c* file the memory is reserved for the input and output fields, and then two values are set to prepare an example.

```c
uint32_t a[32];
uint32_t b[32];
uint32_t result[64];

memset(a, 0, sizeof (a));
memset(b, 0, sizeof (b));
memset(result, 0, sizeof (result));

a[0] = 3;
b[0] = 5;
```

Thus, with the aim of show how it is possible to configure the hardware, a decelerator factor is set with the function *set_delay* exploiting again the register interface produced by *regtool*.

```c
void set_delay(uint8_t delay)
{
uint32_t volatile * ctrl2_reg = (uint32_t*)(
   WIDE_ALU_CTRL2_REG_OFFSET + WIDE_ALU0_BASE_ADDR);

uint32_t ctrl2_old_value;

//Read old value
ctrl2_old_value = *ctrl2_reg;

//Overwrite operation bits
*ctrl2_reg = ctrl2_old_value | (delay & WIDE_ALU_CTRL2_DELAY_MASK)
   ;
}
```

Inside the *wide_alu* code it is useful to highlight the role of the function *pool_done* where it is used the *status* register to know if the hardware is idle, pending or shows some errors.

```c
int poll_done(void)
{
uint32_t volatile * status_reg = (uint32_t*)(
    WIDE_ALU_STATUS_REG_OFFSET + WIDE_ALU0_BASE_ADDR);
uint32_t current_status;

do {
  current_status = (*status_reg)&WIDE_ALU_STATUS_CODE_MASK;
} while(current_status == WIDE_ALU_STATUS_CODE_VALUE_PENDING);

if (current_status == WIDE_ALU_STATUS_CODE_VALUE_IDLE)
  return 0;
else
  return current_status;
}
```

Thus, similarly, the result is written by the hardware an saved in memory to be printed as a confirm of the right behavior of the implementation.

In order to evaluate the performance of the device and check the accuracy of the result it is possible to exploit the PULP runtime routine to compile and simulate the module inside the PULPissimo microcontroller. This operation is performed sourcing the *makefile* in the directory *<PULP_RUNTIME_HOME>/install/rules/pulp_rt.mk* specifying the parameters like the desired input-output method, e.g. UART. This file includes another file specific to the target, PULPissimo, and defines the parameters used during the compilation and simulation. In the end, the application is run according to the desired specification.

4.4 RISC-V programming

After the software programming, the last section is entirely focused on the way to access, program, and debug the core from the computer interface. The first step, as said previously, is to load the bitstream exploiting the Vivado hardware manager either by the graphic user interface (GUI) or the batch mode. This action consists of effectively programming the hardware to execute the desired functionality. Thus, the next crucial step is to establish communication between the FPGA and the computer. This can be achieved through various interfaces, such as JTAG.

The ZCU102 evaluation board can be connected to the host computer exploiting the *Digilent JTAG_2_NC* connector to load the bitstream, while the *PMOD GPIO header* it is used to reach directly the RISC-V, these pins are mapped to the core during the syntheses. This configuration is available thanks to the JTAG chain depicted in figure 4.6.

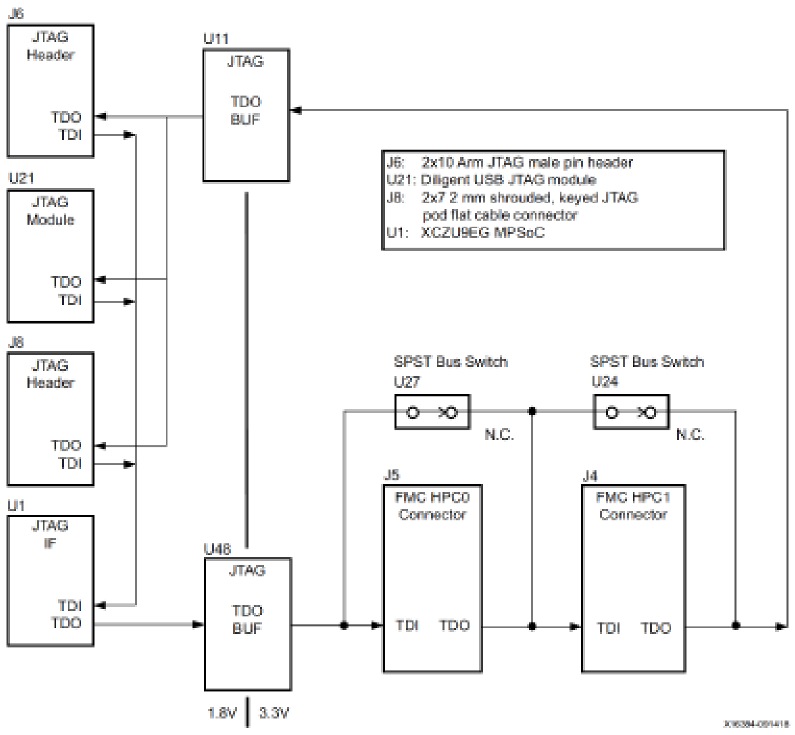


Figure 4.6: JTAG Chain block diagram [45]

With the FPGA programmed and communication established, debugging can begin. As previously said 2.3.1, a modified RISC-V GNU toolchain is included to power the framework with a tool used to generate an Executable and Linkable Format (ELF) file, from the desired application. Thus, the Open On-Chip Debugger (OpenOCD), created by Dominic Rath, is exploited in conjunction with GDB to communicate with the internal RISC-V debug module. PULP platform provides several standard configuration files for some boards in order to ease the use of the debugger interface. The configuration file is related to the programming device used to link the host computer with the board. In the listing 4.4 it is shown the settings used to communicate with the HS2 programming cable, a high-speed programming solution for Xilinx FPGAs.

```
adapter_khz     1000

# Digilent JTAG-HS1
interface ftdi
ftdi_vid_pid 0x0403 0x6014
ftdi_channel 0
ftdi_layout_init 0x00e8 0x60eb

set _CHIPNAME riscv

jtag newtap $_CHIPNAME unknown0 -irlen 5 -expected-id 0x5fffedb3
jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x50001db3

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME -
    coreid 0x3e0

gdb_report_data_abort enable
gdb_report_register_access_error enable

riscv set_reset_timeout_sec 120
riscv set_command_timeout_sec 120

# prefer to use sba for system bus access
riscv set_prefer_sba on

# dump jtag chain
scan_chain


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

In a separate terminal, it is possible to launch the customized version of GDB provided by the PULP platform passing the ELF file as an argument. In a third terminal launch a serial port client on Linux is used to redirect the UART output. In conclusion, it is possible to debug the program using the following GDB command:

```
target remote localhost:3333
load
continue
```

Chapter 5

Methods

A detailed description of the architecture can be introduced through the schematic view presented in figure 5.1. Firstly, as introduced in Chapter 2, the only interface

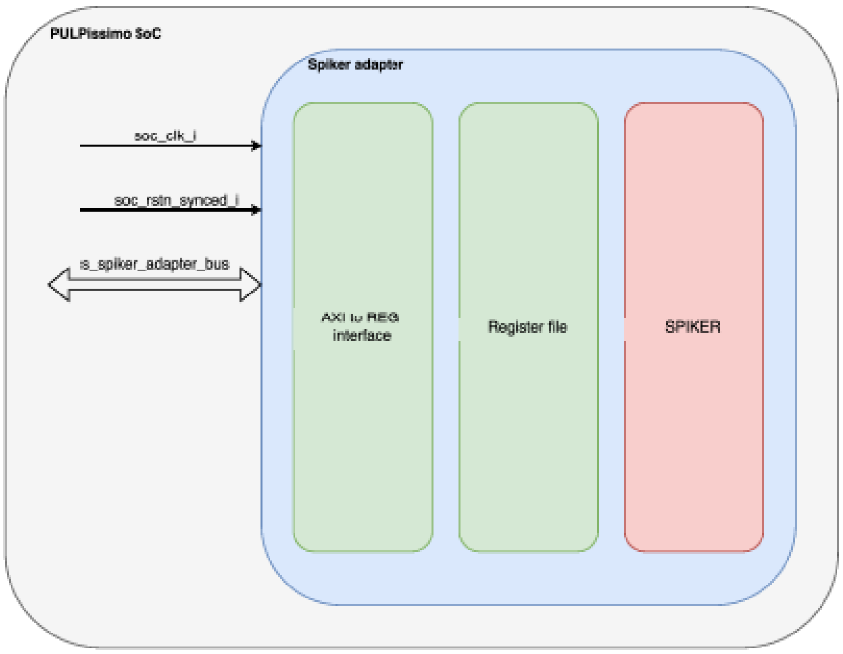


Figure 5.1: System on Chip schematic

between the IP and the rest of the SoC is represented by the AXI bus, referred to as the *s_spiker_adapter_wbus*. The other two signals are the clock and reset signals. The AXI bus enables connection to the memory and facilitates both requests and responses handled by the CPU. Additionally, the overall organization of the *Spiker adapter* module can be seen, featuring its three main components:

- **AXI to REG interface**: This component contains several macro and interfaces used to pack and unpack the data according to the AXI protocol, minimizing unnecessary signals.
- **Register file**: As previously introduced, it is generated exploit the *regtool* software.
- **Spiker**: The main component of the IP, which includes both the modules responsible for managing data and control movement, as well as the computation engine.

Figure 5.2 presents an exploded view of the bridge between the AXI slave input and the register file to highlight the connections between the different modules. [ht] The first module called the *AXI to REG interface*, adapts the AXI protocol

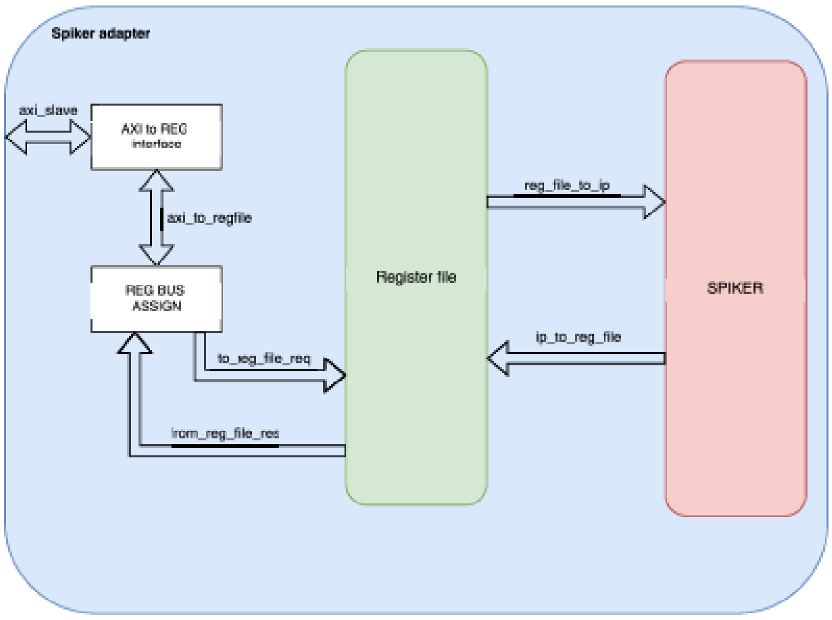


Figure 5.2: Spiker adapter detailed schematic

to the register file structure while preserving the interface's nature. Specifically, the conversion from the *interface* to two separate *structs*, which allows easy access to the internal fields, is handled by the *REG BUS ASSIGN*. Finally, these are connected to the auto-generated register, while two other structs perform a similar function for the core of the IP, *Spiker* which is further exposed in figure 5.3. Thus, the module *Spiker Reader* is responsible for unrolling the registers that hold the incoming data and presenting all the bits in parallel to the input port of the network. Moreover, it is in charge of notifying the *Spiker* engine whenever a new sample is ready to be processed through the *Sample Ready* signal, while the signal

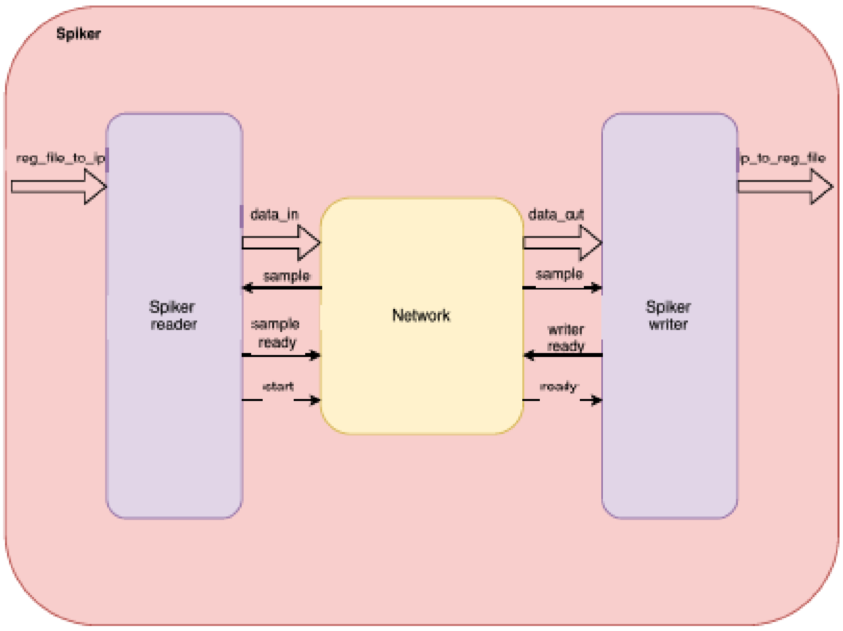


Figure 5.3: Spiker schematic

called *Start* is used at the beginning of the task. On the other side of the *Spiker* schematic, the *Spiker Writer* module is implemented to divide the computation result into multiple registers, while the control signals allow the *Network* to know when it is ready to receive new data. The core of the computation is the *Network*

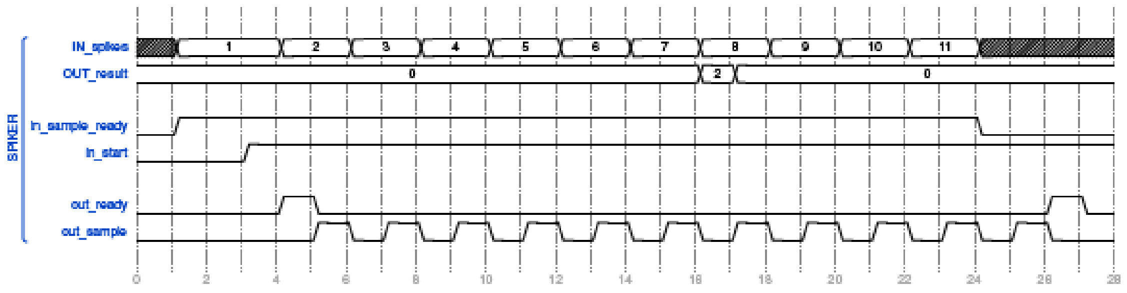


Figure 5.4: Spiker waves

module, which contains all the hardware responsible for analyzing incoming spikes and processing responses. The behavior of this component is illustrated in Figure 5.4, showing the input and output signals. Specifically, when the *Network* module is started, if the *sample ready* signal has already been received, it notifies the rest of the framework that it is ready. For each incoming data point, the *sample* signal acknowledges that the information has been correctly received. At the end of the burst, when the *sample ready* signal is deactivated, the *ready* signal confirms the successful completion of the exchange.

The accelerator consists of two layers used to process incoming and outgoing signals between the spiking core of the network and the external modules. Additionally, a clock-driven mechanism updates the membrane potential at each clock cycle, providing a purely event-driven architecture. There is also an arbitrary number of hidden layers connected in a feed-forward structure, designed to perform computations organized into temporal steps. When the inner architecture is ready, it notifies the CU, which sends a new set of spikes and initiates the computation. Similarly, when the computation ends, the CU is notified again, allowing a new computation to begin.

Chapter 6

Results

In order to verify the correctness of the implemented model, a step-by-step verification has been performed. Firstly, the *Network* module has been tested separately from the rest of the architecture with the aim of finding an input value suitable to be used as a test case to prove the implementation. Then, a software-based simulation performed through the Questasim software and provided by the Pulp platform is exploited to check the behavior of the entire IP at the register-transfer level. In conclusion, the bitstream is downloaded inside the FPGA in order to simulate the hardware implementation, accessing the core thanks to GDB.

6.1 Software simulation

To demonstrate how the *Network* module should realistically function, the mock-up testbench shown in listing 6.1 is used to stimulate the module and observe the results. The testbench achieves its goal by changing the value of the input signal at each sample according to the protocol expected by the module. Additionally, the simulation begins only when the *ready* signal indicates that the *Network* is ready to receive the samples. At the end of the simulation, the *sample_ready* signal is set to zero to stop the operation. Meanwhile, a monitor is used to display useful debugging information for the architecture.

```
    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 100 MHz clock
    end

    // Stimulus generation
    initial begin
        // Initialize signals
        rst_n = 0;
        input_signal = 4'hF;
        sample_ready = 1;
        start = 0;

        // Apply reset
        #10
        rst_n = 1;
        @(posedge ready);
        start = 1;

        // Wait for sample to go high
        @(posedge sample);
        input_signal = 4'he;

        @(posedge sample);
        input_signal = 4'hd;

        @(posedge sample);
        input_signal = 4'hc;

        @(posedge sample);
        input_signal = 4'hf;
        @(posedge sample);
        input_signal = 4'hf;
        @(posedge sample);
        input_signal = 4'hf;

        #10
        @(posedge ready);
        sample_ready = 0;

    end

    // Monitor outputs
    initial begin
        $monitor("At time %t, start = %h, input_signal = %h,
   output_signal = %h, sample_ready = %h, ready = %h, sample = %h"
   , $time, start, input_signal, output_signal, sample_ready,
   ready, sample);
    end
```

As a result, the waveforms in Figure 6.1 illustrate the internal behavior of the accelerator and its response to the stimuli provided by the testbench. The desired proof, which demonstrates that the input signals have been processed and produce a valid non-zero output, is obtained after 625 µs, when the *output_signal* equals 2. Subsequently, the *Network* module is integrated into the Pulp SoC to validate the entire architecture within a simulated environment built using Questasim

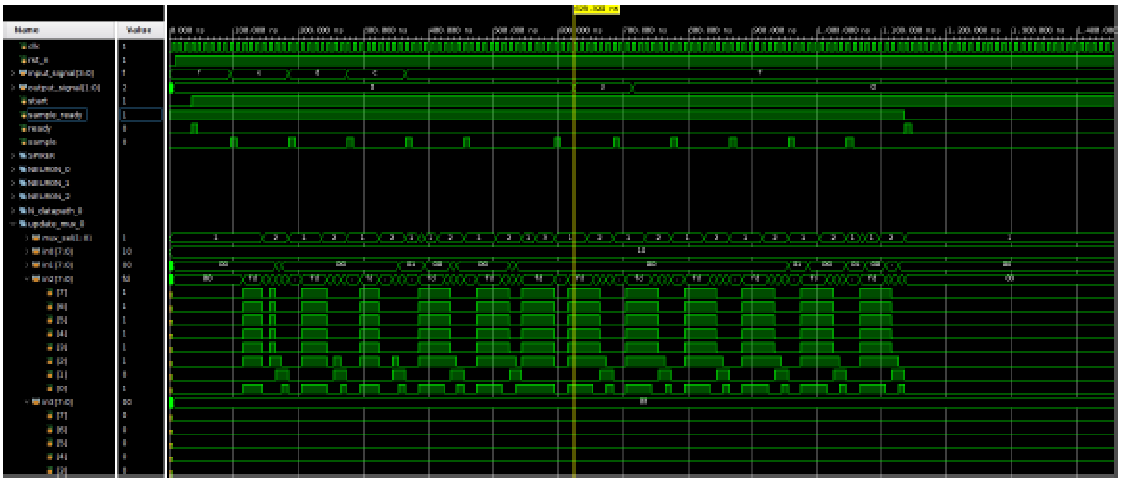


Figure 6.1: Network simulation wave graph

software and driven by C-based source code. This simulation aims to replicate the same stimuli as in the previous test, but with the microcontroller managing the configuration and operation of the modules based on instructions provided by the source code reported in listing 6.1.

```c
#include <stdint.h>
#include <stdio.h>
#include "pulp.h"
#include "spiker_adapter.h"

#define SPIKER_ADAPTER_BASE_ADDR  0x1A400000
#define SPIKER_ADAPTER_CTRL1_MASK 0x1

int __rt_fpga_fc_frequency = 20000000;
int __rt_fpga_periph_frequency = 10000000;
```

Firstly, the base address for the memory mapped peripheral called *Spiker adapter* and the mask used for the fields inside the control register are defined. Then, the two weekly defined variable related to the fabric controller and the peripheral frequency are redefined according to the values set during the FPGA implementation.

```c
int main()
{
    printf("Hello World!\n");

    uint32_t buffer[SPIKER_ADAPTER_SPIKES_MULTIREG_COUNT];
    memset(buffer, 0, sizeof (buffer));
    buffer[0] = 0x89ABCDEF;
    printf("Buffer[0] content is %x\n", buffer[0]);

    printf("The address of buffer is %x\n", &buffer);
```

Subsequently, the buffer used to hold the stimuli for the simulation is initialized and can be immediatly checked during the debug.

```
    // Set up the pointers to the registers
    uint32_t volatile *spiker_adapter_ctrl1 = (uint32_t *)(
   SPIKER_ADAPTER_BASE_ADDR + SPIKER_ADAPTER_CTRL1_REG_OFFSET);
    uint32_t volatile *spiker_adapter_status = (uint32_t *)(
   SPIKER_ADAPTER_BASE_ADDR + SPIKER_ADAPTER_STATUS_REG_OFFSET);

    // Set up the pointers to the DATA registers
    uint32_t volatile *spiker_adapter_reg = (uint32_t *)(
   SPIKER_ADAPTER_BASE_ADDR + SPIKER_ADAPTER_SPIKES_0_REG_OFFSET);
    uint32_t volatile *spiker_adapter_res = (uint32_t *)(
   SPIKER_ADAPTER_BASE_ADDR +
   SPIKER_ADAPTER_SPIKES_RESULT_0_REG_OFFSET);
```

The base address already mentioned and the offsets defined in the auto-generated header file called *spiker_adapter.h* are used to address the fields of the register file for both control signals and data.

```
    // Write to the accelerator interface (spiker_reg)
    for (size_t i = 0; i < SPIKER_ADAPTER_SPIKES_MULTIREG_COUNT -
   23; i++)
    {
        spiker_adapter_reg[i] = buffer[i];
        asm volatile ("": : : "memory");
    }

    // SAMPLE_READY <= 1 (Accelerator can read the data)
    uint32_t old_ctrl1 = *spiker_adapter_ctrl1;
    *spiker_adapter_ctrl1 = old_ctrl1 | ( 1 <<
   SPIKER_ADAPTER_CTRL1_SAMPLE_READY_BIT);
    printf("Samples are ready (sample_ready) ctrl1 = %x\n", *
   spiker_adapter_ctrl1);

    // START <= 1
    old_ctrl1 = *spiker_adapter_ctrl1;
    *spiker_adapter_ctrl1 = old_ctrl1 | (1 <<
   SPIKER_ADAPTER_CTRL1_START_BIT);
    printf("I've started the accelerator (start) ctrl1 = %x\n", *
   spiker_adapter_ctrl1);
```

After the configuration phase, the buffer is used to write the register file, including also an optional memory break to further ensure the right behavior. Thus, the *Network* module can be started and produce a result. Lastly, the result is read again checked through the debugger.

```
    // CHECK STATUS OF THE ACCELERATOR WAITING FOR READY
    while ((*spiker_adapter_status & 0x1) != 1)
    {
        printf("Waiting for the accelerator to be ready\t status =
    %x\n", *spiker_adapter_status);
    }

    // READ FROM MEMORY
    for (size_t i = 0; i < 4; i++)
    {
        printf("Check: reg[%i] = %x\n", i, spiker_adapter_res[i]);
    }

    printf("JOB DONE\n");

    return 0;
}
```

The source code is subsequently compiled and used as the basis for hardware simulation, replacing the testbench. This approach mirrors the one used during FPGA implementation, reliably emulating the implemented solution. The resulting waveform is shown in figure 6.2, highlighting how the hardware accelerator's result is written into the register file and then becomes available for reading by the rest of the architecture.

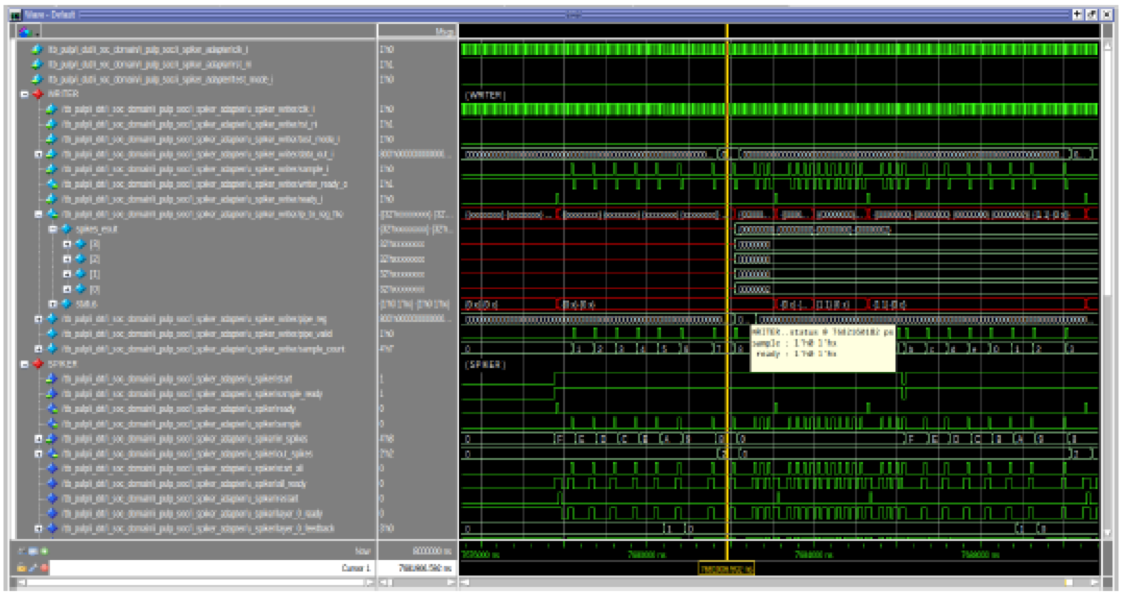
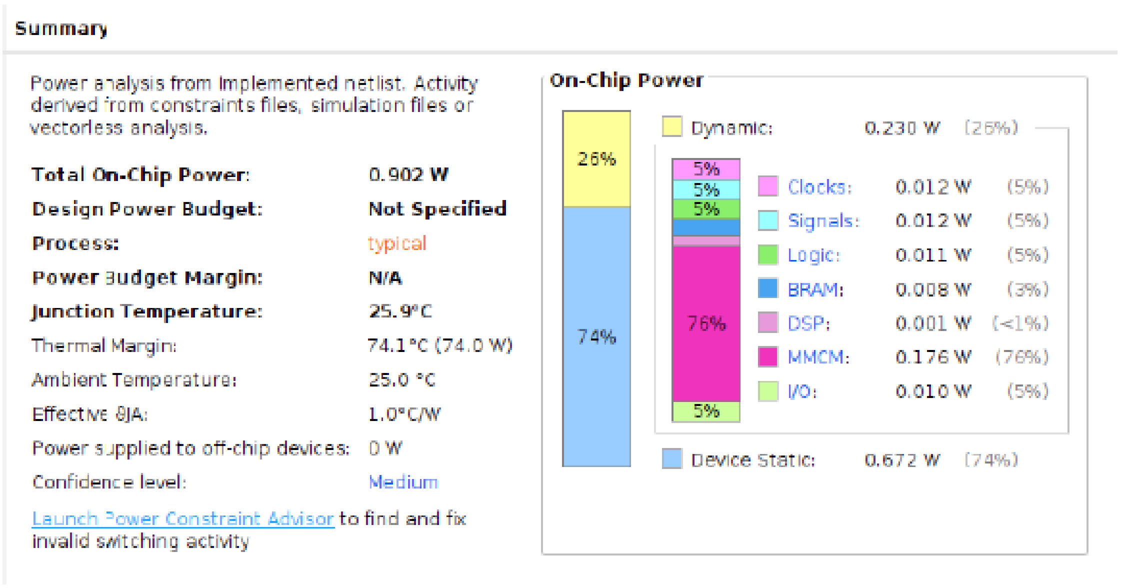


Figure 6.2: Intel Questasim simulation

6.2 FPGA implementation

The final step is carried out to verify that this implementation can effectively provide a reliable solution to reduce the impact of network computation on power consumption. A realistic implementation of the network, powered by a 748-bit input and producing a 10-bit output, is verified with the goal of obtaining accurate results. The Vivado suite is once again utilized to generate the power report starting from the implementation performed for the ZCU102 board, as shown in figure 6.3. Notably, the overall power consumption of the microcontroller aligns with the expectations for a low-power neuromorphic application, measuring just 902 mW. In addition, Figure 6.4 presents a representation of the FPGA area utilization in both graphical and textual formats. This result demonstrates that the choice of the ZCU102 Evaluation Kit provides a reasonable safety margin, although a smaller development board could also be sufficient.



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

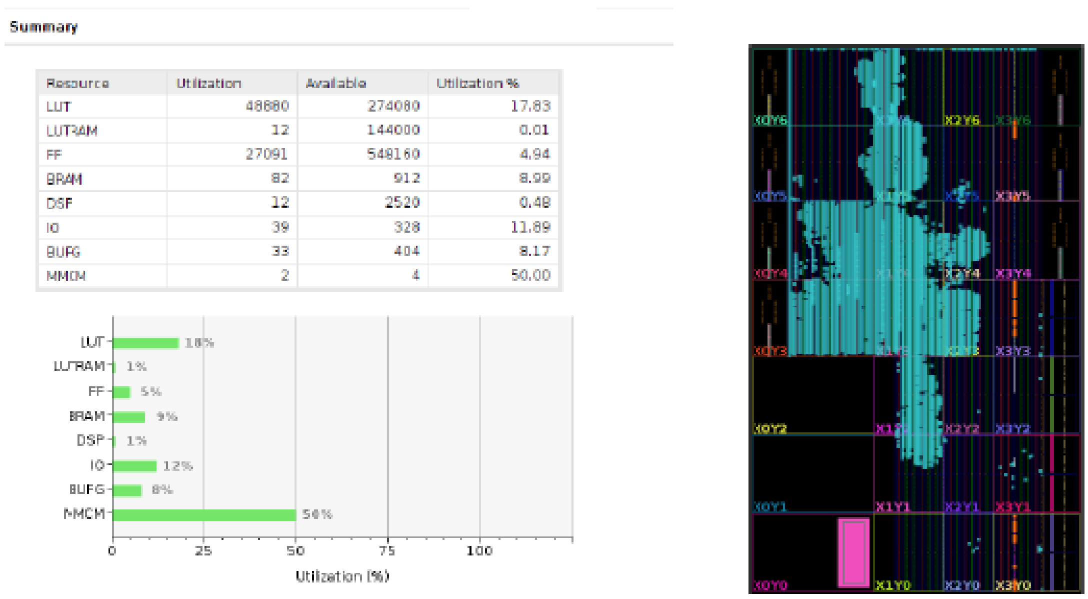
Total On-Chip Power:	**0.902 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.9°C**
Thermal Margin:	74.1°C (74.0 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

🟨	Dynamic:	0.230 W	(26%)
🟪	Clocks:	0.012 W	(5%)
🟦	Signals:	0.012 W	(5%)
🟩	Logic:	0.011 W	(5%)
🟦	BRAM:	0.008 W	(3%)
🟪	DSP:	0.001 W	(<1%)
🟥	MMCM:	0.176 W	(76%)
🟩	I/O:	0.010 W	(5%)
🟦	Device Static:	0.672 W	(74%)

Figure 6.3: Power report



Summary

Resource	Utilization	Available	Utilization %
LUT	48880	274080	17.83
LUTRAM	12	144000	0.01
FF	27091	548160	4.94
BRAM	82	912	8.99
DSP	12	2520	0.48
IO	39	328	11.89
BUFG	33	404	8.17
MMCM	2	4	50.00

Figure 6.4: Report FPGA utilization

Appendix A

Bender

Bender is a dependency management tool designed by PULP platform with the aim of provide an efficient way to define and ensure the right dependency between different IPs in the same hardware device. On the other hand, Bender offers other features to verify that the source files are valid for EDA tools without any assumptions about the specific software.

First, Bender collects all the source files of a hardware IP, maintaining the required order and supporting both SystemVerilog and VHDL. It also manages include directories. Additionally, Bender tracks the Git hash of each dependency and stores this information in the *Bender.lock* file, enabling reproducible builds.

```
package:
  name: spiker_adapter
  authors:
    - "Renato Belmonte <renato.belmonte@studenti.polito.it"
dependencies:
  common_cells:
  { git: "https://github.com/pulp-platform/common_cells.git",
   version: 1.21.0 }
  axi:
  { git: "https://github.com/pulp-platform/axi.git", version:
   0.39.2 }
  register_interface:
  { git: "https://github.com/pulp-platform/register_interface.git"
   , version: 0.4.1 }
  spiker:
  { git: "https://github.com/RenatoBelmonte/spiker.git", rev: "
   main" }
sources:
  #spiker_adapter
  defines:
  files:
    - gen_sv/spiker_adapter_reg_pkg.sv
    - gen_sv/spiker_adapter_reg_top.sv
    - rtl/spiker_writer.sv
    - rtl/spiker_reader.sv
    - rtl/spiker_adapter.sv
  vlog_opts:
    - -L register_interface_lib
```

Listing A shows the Bender file used in the *Spiker adapter* repository. After an initial section containing package information, the dependencies are listed in the dependencies section of the package manifest. These sources are retrieved from the specified GitHub repositories, and can be referenced using either a version or a target, such as *main*. The *source* section defines the source files or groups of source files used in the design, with the option to specify which files or groups to include or exclude.

More information, including the source code, examples and a comprehensive list of available commands, can be accessed from the official documentation [46]

Appendix B

Regtool

The register tool is part of OpenTitan, the open source secure silicon ecosystem administered by lowRISC CIC. It is used to construct register documentation, register RTL and header files. Specifically, the standalone version relies on python3 to read configuration and register descriptions in a variant of the JavaScript Object Notation (JSON) format, HJSON, and generate various output formats. A more comprehensive explanation of the tool it is provided by OpenTitan on the official website [47]

Specifically, the example in B demonstrates how the register file for the *Spiker adapter* module is generated. The primary information, such as the clock name and the interface used, is declared at the beginning of the file. The registers are then described either individually or in groups, known as *multireg*, using a combination of optional and mandatory fields. The optional fields provide significant customization of the register behavior, allowing, for example, enhanced security or reliability of the stored data. To this end, the *swaccess* and *hwaccess* fields control access to the information within the registers.

```
{
    name: "spiker_adapter",
    clock_primary: "clk_i",
    reset_primary: "rst_ni",
    bus_interfaces: [{ protocol: "reg_iface", direction:
    "device"}],

    regwidth: "32",
    registers: [
        { multireg:
          { name: "SPIKES",
            desc: "Subword of Spikes",
            count: "25",
            cname: "SPIKES",
            swaccess: "rw",
            fields: [
                { bits: "31:0"
                }
            ],
          }
        },
        { multireg:
          { name: "SPIKES_RESULT",
            desc: "Subword of results.",
            count: "4",
            cname: "SPIKES_RESULT",
            swaccess: "ro",
            hwaccess: "hwo",
            hwext: "true",
            fields: [
                { bits: "31:0"
                }
            ],
          }
        },
        { name: "CTRL1",
          desc: "Controls handshaking signal of the accelerator.",
          swaccess: "rw",
          hwaccess: "hro",
          fields: [
              { bits: "0", name: "SAMPLE_READY",
                desc: "Signals the presence of a new sample."
              }
              { bits: "1", name: "START",
                desc: "Signals that SPIKER can start."
              }
          ]
        },
        { name: "STATUS",
          desc: "Contains the current status of the accelerator.",
          swaccess: "rw",
          hwaccess: "hwo",
          fields: [
              { bits: "0", name: "SAMPLE",
                desc: "Signals that SPIKER is ready for the next sample."
              }
              { bits: "1", name: "READY",
                desc: "Signals that SPIKER is has a new result."
              }
          ]
        }
    ],
}
```

Bibliography

[1] S. S. Udpa and L. Udpa. «NDT Techniques: Signal and Image Processing». In: *Encyclopedia of Materials: Science and Technology*. Ed. by K. H. Jürgen Buschow, Robert W. Cahn, Merton C. Flemings, Bernhard Ilschner, Edward J. Kramer, Subhash Mahajan, and Patrick Veyssière. Oxford: Elsevier, Jan. 2001, pp. 6033–6035. ISBN: 978-0-08-043152-9. DOI: 10.1016/B0-08-043152-6/01064-0. URL: https://www.sciencedirect.com/science/article/pii/B0080431526010640 (visited on 07/30/2024) (cit. on p. 1).

[2] Samanwoy Ghosh-Dastidar and Hojjat Adeli. «Third Generation Neural Networks: Spiking Neural Networks». en. In: *Advances in Computational Intelligence*. Ed. by Wen Yu and Edgar N. Sanchez. Berlin, Heidelberg: Springer, 2009, pp. 167–178. ISBN: 978-3-642-03156-4. DOI: 10.1007/978-3-642-03156-4_17 (cit. on p. 2).

[3] Alberto Dequino, Alessio Carpegna, Davide Nadalini, Alessandro Savino, Luca Benini, Stefano Di Carlo, and Francesco Conti. «Compressed Latent Replays for Lightweight Continual Learning on Spiking Neural Networks». In: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. ISSN: 2159-3477. July 2024, pp. 240–245. DOI: 10.1109/ISVLSI61997.2024.00052. URL: https://ieeexplore.ieee.org/abstract/document/10682744 (visited on 10/11/2024) (cit. on p. 2).

[4] Xiangwen Wang, Xianghong Lin, and Xiaochao Dang. «Supervised learning in spiking neural networks: A review of algorithms and evaluations». In: *Neural Networks* 125 (May 2020), pp. 258–280. DOI: 10.1016/j.neunet.2020.02.011 (cit. on p. 2).

[5] Sanaullah, Shamini Koravuna, Ulrich Rückert, and Thorsten Jungeblut. «Exploring spiking neural networks: a comprehensive analysis of mathematical models and applications». English. In: *Frontiers in Computational Neuroscience* 17 (Aug. 2023). Publisher: Frontiers. ISSN: 1662-5188. DOI: 10.3389/fncom.2023.1215824. URL: https://www.frontiersin.org/journals/computational-neuroscience/articles/10.3389/fncom.2023.1215824/full (visited on 07/31/2024) (cit. on p. 2).

[6] *Three generations of artificial neural networks (ANNs). MLP, multilayer... | Download Scientific Diagram*. en. URL: `https://www.researchgate.net/figure/Three-generations-of-artificial-neural-networks-ANNs-MLP-multilayer-perceptron-MP_fig2_339481763` (visited on 07/31/2024) (cit. on p. 2).

[7] Michelangelo Barocci, Vittorio Fra, Enrico Macii, and Gianvito Urgese. «Review of open neuromorphic architectures and a first integration in the RISC-V PULP platform». In: *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. ISSN: 2771-3075. Dec. 2023, pp. 470–477. DOI: `10.1109/MCSoC60832.2023.00076`. URL: `https://ieeexplore.ieee.org/document/10387816` (visited on 08/04/2024) (cit. on pp. 3, 29).

[8] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks». In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2022, pp. 14–19. DOI: `10.1109/ISVLSI54635.2022.00016` (cit. on p. 3).

[9] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. *Spiker+: a framework for the generation of efficient Spiking Neural Networks FPGA accelerators for inference at the edge*. Jan. 2024. DOI: `10.48550/arXiv.2401.01141`. URL: `http://arxiv.org/abs/2401.01141` (visited on 01/26/2024) (cit. on p. 3).

[10] Dario Padovano, Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «SpikeExplorer: Hardware-Oriented Design Space Exploration for Spiking Neural Networks on FPGA». en. In: *Electronics* 13.9 (Jan. 2024), p. 1744. ISSN: 2079-9292. DOI: `10.3390/electronics13091744`. URL: `https://www.mdpi.com/2079-9292/13/9/1744` (visited on 09/06/2024) (cit. on p. 3).

[11] Germain Haessig, Francesco Galluppi, Xavier Lagorce, and Ryad Benosman. «Neuromorphic networks on the SpiNNaker platform». In: *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. Mar. 2019, pp. 86–91. DOI: `10.1109/AICAS.2019.8771512`. URL: `https://ieeexplore.ieee.org/document/8771512` (visited on 08/04/2024) (cit. on p. 3).

[12] Peter Alfke, Ivo Bolsens, Bill Carter, Mike Santarini, and Steve Trimberger. «It's an FPGA!» In: *IEEE Solid-State Circuits Magazine* 3.4 (2011). Conference Name: IEEE Solid-State Circuits Magazine, pp. 15–20. ISSN: 1943-0590. DOI: `10.1109/MSSC.2011.942449`. URL: `https://ieeexplore.ieee.org/document/6069771` (visited on 08/10/2024) (cit. on p. 5).

[13] *Moore's law.* en. Page Version ID: 1238717785. Aug. 2024. URL: https://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=1238717785 (visited on 08/10/2024) (cit. on p. 6).

[14] Daniel Nenni, Paul McLellan, and Cliff Hou. *Fabless: The Transformation of the Semiconductor Industry.* Inglese. www.SemiWiki.com, Apr. 2014. URL: https://semiwiki.com/books/Fabless%202019%20Version%20PDF.pdf (visited on 08/12/2024) (cit. on p. 6).

[15] David A. Patterson and David R. Ditzel. «The case for the reduced instruction set computer». In: *SIGARCH Comput. Archit. News* 8.6 (1980), pp. 25–33. ISSN: 0163-5964. DOI: 10.1145/641914.641917. URL: https://dl.acm.org/doi/10.1145/641914.641917 (visited on 08/18/2024) (cit. on p. 10).

[16] W.G. Alexander and D.B. Wortman. «Static and Dynamic Characteristics of XPL Programs». In: *Computer* 8.11 (Nov. 1975). Conference Name: Computer, pp. 41–46. ISSN: 1558-0814. DOI: 10.1109/C-M.1975.218804. URL: https://ieeexplore.ieee.org/document/1649280 (visited on 08/19/2024) (cit. on p. 11).

[17] Andrew S. Tanenbaum. «Implications of structured programming for machine architecture». In: *Commun. ACM* 21.3 (Mar. 1978), pp. 237–246. ISSN: 0001-0782. DOI: 10.1145/359361.359454. URL: https://dl.acm.org/doi/10.1145/359361.359454 (visited on 08/18/2024) (cit. on p. 11).

[18] Weiss and Smith. «Instruction Issue Logic in Pipelined Supercomputers». In: *IEEE Transactions on Computers* C-33.11 (Nov. 1984). Conference Name: IEEE Transactions on Computers, pp. 1013–1022. ISSN: 1557-9956. DOI: 10.1109/TC.1984.1676375. URL: https://ieeexplore.ieee.org/document/1676375 (visited on 08/19/2024) (cit. on p. 12).

[19] *Classic RISC pipeline.* en. Page Version ID: 1191069658. Dec. 2023. URL: https://en.wikipedia.org/w/index.php?title=Classic_RISC_pipeline&oldid=1191069658 (visited on 08/19/2024) (cit. on p. 12).

[20] J.E. Smith and A.R. Pleszkun. «Implementing precise interrupts in pipelined processors». In: *IEEE Transactions on Computers* 37.5 (May 1988). Conference Name: IEEE Transactions on Computers, pp. 562–573. ISSN: 1557-9956. DOI: 10.1109/12.4607. URL: https://ieeexplore.ieee.org/document/4607 (visited on 08/19/2024) (cit. on p. 12).

[21] Dileep Bhandarkar and Douglas W. Clark. «Performance from architecture: comparing a RISC and a CISC with similar hardware organization». In: *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems.* ASPLOS IV. New York, NY, USA: Association for Computing Machinery, Apr. 1991, pp. 310–319. ISBN:

978-0-89791-380-5. DOI: 10.1145/106972.107003. URL: https://dl.acm.org/doi/10.1145/106972.107003 (visited on 08/19/2024) (cit. on p. 12).

[22] David A. Patterson and Carlo H. Sequin. «RISC I: A Reduced Instruction Set VLSI Computer». In: *Proceedings of the 8th annual symposium on Computer Architecture*. ISCA '81. Washington, DC, USA: IEEE Computer Society Press, 1981, pp. 443–457. (Visited on 08/19/2024) (cit. on p. 12).

[23] *History – RISC-V International*. en-US. URL: https://riscv.org/about/history/ (visited on 08/26/2024) (cit. on p. 15).

[24] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html (cit. on p. 15).

[25] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UCB/EECS-2011-62. May 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html (cit. on p. 15).

[26] Frank K Gürkaynak and Luca Benini. «10 years of making PULP chips». en. In: () (cit. on p. 17).

[27] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. «Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX». In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145 (cit. on p. 19).

[28] *Getting started with PULP: SW point of view*. URL: https://pulp-platform.org/pulp_sw.html (visited on 09/30/2024) (cit. on p. 20).

[29] *pulp-platform/pulp-sdk*. original-date: 2018-02-07T12:22:18Z. Sept. 2024. URL: https://github.com/pulp-platform/pulp-sdk (visited on 09/23/2024) (cit. on p. 19).

[30] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. «GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors». In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. arXiv:2201.08166 [cs, eess]. Oct. 2021, pp. 409–416. DOI: 10.1109/ICCD53106.2021.00071. URL: http://arxiv.org/abs/2201.08166 (visited on 09/24/2024) (cit. on p. 21).

[31] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. *Carbon Emissions and Large Neural Network Training*. arXiv:2104.10350 [cs]. Apr. 2021. DOI: `10.48550/arXiv.2104.10350`. URL: `http://arxiv.org/abs/2104.10350` (visited on 09/04/2024) (cit. on p. 22).

[32] Jason K. Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. *Training Spiking Neural Networks Using Lessons From Deep Learning*. en. Sept. 2021. URL: `https://arxiv.org/abs/2109.12894v6` (visited on 09/07/2024) (cit. on p. 22).

[33] «Fig. 1: Biomimetic neuromorphic computing. | Nature Communications». en. In: (). URL: `https://www.nature.com/articles/s41467-021-22332-8/figures/1` (visited on 09/07/2024) (cit. on p. 23).

[34] A. L. Hodgkin and A. F. Huxley. «A quantitative description of membrane current and its application to conduction and excitation in nerve». In: *The Journal of Physiology* 117.4 (Aug. 1952), pp. 500–544. ISSN: 0022-3751. URL: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/` (visited on 09/24/2024) (cit. on p. 27).

[35] E.M. Izhikevich. «Simple model of spiking neurons». In: *IEEE Transactions on Neural Networks* 14.6 (Nov. 2003). Conference Name: IEEE Transactions on Neural Networks, pp. 1569–1572. ISSN: 1941-0093. DOI: `10.1109/TNN.2003.820440`. URL: `https://ieeexplore.ieee.org/document/1257420` (visited on 09/24/2024) (cit. on p. 27).

[36] Richard FitzHugh. «Mathematical models of threshold phenomena in the nerve membrane». In: *The bulletin of mathematical biophysics* 17 (1955). Publisher: Springer, pp. 257–278 (cit. on p. 27).

[37] Jinichi Nagumo, Suguru Arimoto, and Shuji Yoshizawa. «An active pulse transmission line simulating nerve axon». In: *Proceedings of the IRE* 50.10 (1962). Publisher: IEEE, pp. 2061–2070 (cit. on p. 27).

[38] Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. *XNOR Neural Engine: a Hardware Accelerator IP for 21.6 fJ/op Binary Neural Network Inference*. en. July 2018. DOI: `10.1109/TCAD.2018.2857019`. URL: `https://arxiv.org/abs/1807.03010v1` (visited on 10/05/2024) (cit. on pp. 28, 34).

[39] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. «Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing». In: *IEEE Journal of Solid-State Circuits* 54.7 (July 2019). Conference Name: IEEE Journal of Solid-State Circuits, pp. 1970–1981. ISSN: 1558-173X. DOI: `10.1109/JSSC.2019.2912307`. URL: `https://`

ieeexplore.ieee.org/document/8715500/citations#citations (visited on 10/05/2024) (cit. on p. 29).

[40] Charlotte Frenkel and Giacomo Indiveri. «ReckOn: A 28nm Sub-mm2 Task-Agnostic Spiking Recurrent Neural Network Processor Enabling On-Chip Learning over Second-Long Timescales». In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 65. ISSN: 2376-8606. Feb. 2022, pp. 1–3. DOI: 10.1109/ISSCC42614.2022.9731734. URL: https://ieeexplore.ieee.org/document/9731734 (visited on 10/05/2024) (cit. on p. 29).

[41] *pulp-platform/pulpissimo*. original-date: 2018-02-09T10:24:02Z. Aug. 2024. URL: https://github.com/pulp-platform/pulpissimo (visited on 08/31/2024) (cit. on p. 30).

[42] *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit*. en. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html (visited on 08/31/2024) (cit. on p. 31).

[43] PULP Platform. *A Deep Dive into HW/SW Development with PULP - Part 1*. Mar. 2021. URL: https://www.youtube.com/watch?v=B7BtaYh3VqI (visited on 09/03/2024) (cit. on pp. 35, 36).

[44] PULP Platform. *A Deep Dive into HW/SW Development with PULP - Part 2*. Mar. 2021. URL: https://www.youtube.com/watch?v=0GdsS2hq0zM (visited on 09/03/2024) (cit. on p. 37).

[45] «ZCU102 Evaluation Board User Guide». en. In: (2023). URL: https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd (cit. on p. 41).

[46] *pulp-platform/bender*. original-date: 2018-05-17T08:24:21Z. Sept. 2024. URL: https://github.com/pulp-platform/bender (visited on 10/06/2024) (cit. on p. 55).

[47] *reggen & regtool: Register Generator - OpenTitan Documentation*. URL: https://opentitan.org/book/util/reggen/ (visited on 10/06/2024) (cit. on p. 56).