POLITECNICO DI TORINO

Master Degree In Computer Engineering

Master Thesis

Interfacing a Neuromorphic Coprocessor with a RISC-V Architecture

Supervisors
PhD Gianvito Urgese
Dr. Evelina Forno

Candidate
Andrea Spitale

Academic Year 2020-2021

"Intelligence is the ability to adapt to change"

— Stephen Hawking

Summary

The concept of *neural network* is nowadays spread everywhere. Commonly meant as a mean to provide what is called artificial intelligence, neural network constitute one of few state of the art technologies which are constantly growing in complexity and efficiency to overcome new challenges, providing fascinating services and features, ranging from image classification and manipulation, up to speech recognition and many more that are still being explored. A neural network is a computing system which takes inspiration from the human brain, exploiting its parallel interconnections to solve complex data problems, modifying its internal parameters (training phase) in order to recognize unknown input data with higher accuracy (inference phase). *Spiking Neural Networks* (SNN) represent an emerging class of neural networks, coming from neuroscience research and aiming at accurately reproducing both static and dynamic behaviours of human brain neurons. Initially conceived to help neuroscientists enrich their knowledge on the human brain structure and working principles, SNNs have gathered attention in the computer science community, for several reasons: power efficiency, continuous learning, natural support for spatio-temporal input.
Internet of Things (IoT) oriented applications, running on smart devices capable of analyzing data in real time with limited power resources, are predicted to be those that will benefit a lot from the adoption of SNN based solutions. Indeed, since spikes are sparse in time and space, the network is characterized by very low activity, thus the overall power consumption is greatly reduced if compared to other solutions, such as *Convolutional Neural Networks*.
To enhance the computing capability of an IoT device during the execution of SNN based algorithms, the thesis describes the design of an interfacing solution for controlling a reconfigurable SNN-accelerated coprocessor, named ODIN, by means of a RISC-V based System on Chip (SoC), without any remote controller from the cloud. The designed architecture exploits the serial peripheral interface (SPI) to let the RISC-V core configure the accelerator parameters, thus offloading the SNN task on ODIN.
The capability of the system to work as a standalone device has been validated by configuring a synfire chain simulation without the intervention of an host computer. A synfire chain is a particular arrangement of spiking neurons, which are connected

in a loop, firing in series once a proper input stimuli is provided. The synfire chain has been used to benchmark the proposed architecture because of its predictable behaviour, which can be monitored through Register Transfer Level (RTL) simulation. The network behaviour is tested on a few case scenarios, where a number of parameters are changed, and the results are matched against the expected ones. Thus, demonstrating that the proposed solution could avoid the usage of a host computer to control and configure the SNN based accelerator.

Contents

Chapter 1

Introduction

Neuromorphic engineering, of which you might have heard as "neurmorphic computing", represents one of the most interesting yet challenging interdisciplinary field for all computer science enthusiasts, both from academic and industrial communities. Conceived in the 80s by Carver Andress Mead, an American engineer that is considered as one of the pioneers of modern microelectronics and that originally coined Gordon Moore's prediction as "Moore's Law", it involves the usage of very large scale integrated (VLSI) circuits to assemble systems that strive to reproduce the behaviour of the human brain, which is capable of processing and transmitting information at a really high speed, due to dense connectivity and high degree of parallelism, while consuming a very low amount of power ($\approx$ 25 W [22]). The key aspect to be taken in consideration when building such a system is the way neurons and synapses are interconnected, since this strongly influences the computations the system will be able to perform, how robust the architecture is with respect to external and undesired inputs and conditions, and whether and how the interconnections will adapt to changes. Neuromorphic engineering goes hand in hand with neural networks, those widespread objects that are involved in the most fascinating electronic systems we deal with everyday, ranging from AI driven cameras on smartphones up to circuits which are provided to robots so that they can react and improve their responses to external events.

According to [50], there are *10 main reasons* that make neuromorphic hardware so interesting in the upcoming years.

1. *Faster*. The Von Neumann architecture had been devised, together with Harvard one, to have machines process information at a way faster rate with respect to the average one with which humans process it. However, if one compares the computational speed of such an architecture with that of the human brain, it becomes evident there are limits and defects that have to be overcome if one wants to reach the efficiency of that biological system. In particular, researchers working in the 80s stated that neural networks computation could have benefited a lot from the development of custom, dedicated

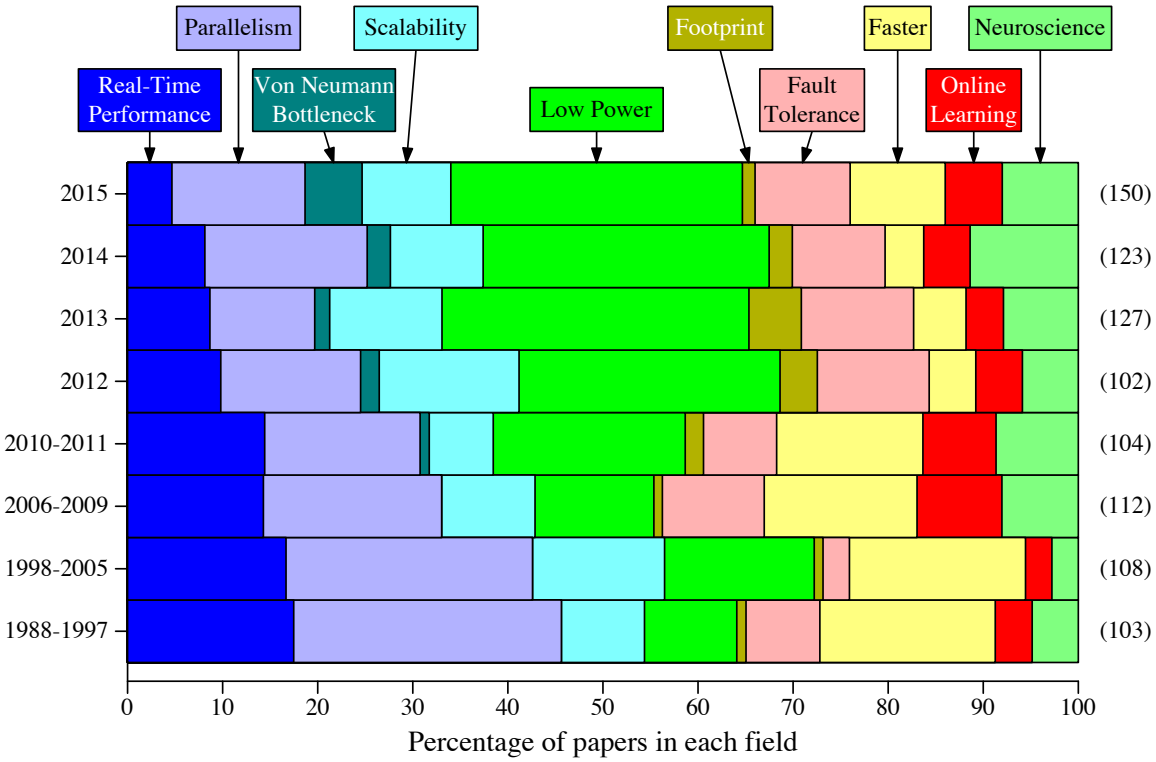

Figure 1.1: A graph showing 10 main motivations for which neuromorphic computing research was conducted. On the bottom, the percentage of papers citing a particular motivation is shown. On the left side the year to which each istogram refers is reported. Finally the right side gives, within round parenthesis, the number of papers on neuromorphic computing published during the years that row refers to. Picture taken from [50].

chips, foreseeing the arrival of an era in which neuromorphic accelerator would have become a fundamental device for machine learning tasks.

2. *Parallelism.* Biological brain performs lots of computations in parallel, achieving a really high throughput, so neuromorphic hardware architectures are tailored to have an high degree of parallelism, providing simple yet efficient computing units, called *neurons*, with condensed interconnections between them, named *synapses*, that exploit some encoding protocol to process information and make it possible for humans to give it a useful meaning. These characteristics make neuromorphic hardware unique among all architectures scientists are aware of.

3. *Von Neumann Bottleneck.* Modern microprocessor based systems can process data *way faster* with respect to accessing it from main memory. This is known

as the Von Neumann Bottleneck. Overall throughput is reduced and the CPU spends most of the operating time in idle state, wasting resources and power. The problem is quite evident nowadays, with CPU operating frequencies and memory sizes that have increased so faster with respect to the speed with which memory and CPU communicate.

4. *Real Time Performance.* Neuromorphic devices find large interests in all those domains for which real time tasks are to be handled. Real time implies reacting within precise deadlines, that must be satisfied in order to prevent undesired consequences from happening. Consequences may be acceptable in case of *soft* real time, for which the system performance may degrade if those deadlines are not respected, or they may be catastrophic in case of *hard* real time, ultimately leading to a total system failure. Applications requiring such performance may be related to digital image reconstruction or autonomous robot control.

5. *Low Power.* This stands firm as the main reason behind neuromorphic hardware development. Nowadays Internet of Things (IoT) and many embedded systems related devices are needed to perform complex computations, yet the power resources are limited. This is strictly related to Moore's law apparently fading away and Dennard scaling. Dennard observed that power density of a given device stays the same as the transistors reduced in size, because current and operating voltage reduce consequently. Thus number of transistors on a given area could be doubled. This allowed the semiconductor industry to have microprocessors which operated at always increasing frequencies during the years, while maintaining acceptable power consumption and density. However two issues rose in the past 20 years, them being dark silicon and leakage current impact on the more refined dynamic power consumption equation $P_{dyn} = \alpha C f V^2 + V I_{leak}$, which ultimately lead to modern high performance processors stop at around 4 GHZ operating frequency, and making designers move to a multi core approach.

6. *Scalability.* As it happens for classic Von Neumann computers, which benefit a lot from having multiple cores scattered across the entire architecture, neuromorphic devices are devised with an high degree of scalability, making it possible to have multiple devices running in parallel and communicating with each other, sharing information and computational resources.

7. *Footprint.* Depending on the application domain, a given design may be too large to fit within given area constraints. Neuromorphic hardware helps in keeping area cost as low as possible.

8. *Fault Tolerance.* Neuromorphic architectures are intrinsically fault tolerant.

This is due to the exploited information encoding and the self healing capability of such devices. Moreover they may help in reducing the impact of process variation in the fabrication of devices, that often lead to degraded performance or unacceptable errors.

9. *Online Learning*. As explained in 2.1.2, is to be intended as the ability of a given neuromorphic device to adapt to input stimuli, refining the efficiency of the tasks they perform, without any external intervention. Given the recent demands for systems that may classify and process data in an unsupervised manner, online learning algorithms have to be developed and successfully implemented in such systems.

10. *Neuroscience*. Neuromorphic computing was firstly conceived to help neuroscientists, that is people who devote their career to the study of of nervous system, trying to delineate its behaviour and working principles. Indeed, simulating neuronal behaviour on classical supercomputers is simply not feasibly, due to scale, speed, and associated power consumption. Neuromorphic architectures are fundamental to perform neuroscience simulations within acceptable processing times.

Neuromorphic computing and devices are widely adopted to perform tasks which belong to various domains, as illustrated in Figure 1.2. A few hints are given in Table 1.1 and Table 1.2.
There is the need for technologies and algorithms which may enable the development of more powerful computing edge devices, while keeping power consumption as low as possible. To this extent, since neuromorphic computing seems to represent a key factor in enabling the transition from a cloud oriented computing environment to one which exploits small devices to perform intensive tasks, this thesis focuses on the development and validation of an architecture that puts a RISC-V based System on Chip (SOC) with a Spiking Neural Network (SNN) oriented hardware accelerator.

Application Domain	**Category**	**Description**
Imaging	Edge Detection	identify edges in a digital image, by identifying points at which brightness rapidly changes. An example is described in [24].
Imaging	Compression	minimizing image size in bytes, possibly without reducing image overall quality. An example is presented in [19].
Imaging	Filtering	changing an image properties to highlight certain features or hide some others. An example is presented in [35].
Imaging	Segmentation	typically used in medical imaging, to search for tumors or similar illnesses, it aims at changing the image representation according to new schemes, which make it easier to analyze. An example is analyzed in [11].
Imaging	Feature Extraction	algorithms used to reduce the number of variables or characteristics associated to an input dataset, in order to speed up the processing phase without losing meaningful information [14].
Imaging	Classification Or Detection	probably the most widespread application, it consists in analyzing an image, to either detect a certain class of objects and/or classify the image and associate it to a precise category, according to the objects being detected. An example is described in [40].
Speech	Word Recognition	technologies and algorithms devoted to analyze spoken language to recognize words and translate them into machine readable format and text. Some recent advancements have been made, like algorithms that can generate images according to a given text based description, as reported in [41].
Data Analysis	Classification	the scope of this kind of this category of algorithms is to analyze input data and associate them to a specific class, through the assignment of a label. An example could consist in having a system that determines whether a given email is purely spam or not. An example is given in [49].
Imaging, Control, Security	Anomaly Detection	algorithms to identify events or data that deviate from a given dataset normal behaviour. An example is presented in [9].
Neuroscience Research	Simulation	neuromorphic architectures are way better suited to provide scientists with meaningful simulations that can be conducted in a reasonable amount of time. An example is reported in [10].
Visual Systems		architectures tailored to run algorithms that improve or repair tissues associated to sense of sight. An example is presented in [3].
Auditory Systems		architectures tailored to run algorithms that improve or repair tissues associated to sense of hearing. An example is described in [34].
Olfactory Systems		architectures tailored to run algorithms that improve or repair tissues associated to sense of smell. An example is given in [27].
Somatosensory Systems		architectures tailored to run algorithms that improve or repair tissues associated to sense of touch. An example is analyzed in [33].

Table 1.1: Machine Learning Applications. Taken from [50]

Application Domain	**Category**	**Description**
Medical Treatment		as discussed in [53], machine learning is changing the healthcare sector around the world. Indeed, sophisticated technologies are being used to create new drugs to cure patients more quicky, provide wearable devices that manage blood sugar levels in people affected by diabetes, increase speed and accuracy of breast cancer diagnosis.
Brain-Machine Interface		A Brain Machine Interface (BCI) between a wired brain and a custom device, which enables the user to translate its brain activity into messages that can be recognized by the custom device and used to execute some sort of task The aim of such system is to enhance or assist human cognitive or motor functions [57].
Robotics	Imitation Learning	like toddlers, also robots and alike can imitate some behaviours of a given target, by replicating the observed patter. These patterns are typically known as Bayesian patterns [12].
Central Pattern Generation (CPG)		Central Pattern Generators are biological circuits that produce basic rhythmic patterns without any external rhythmic input. They constitute the fundamentals by which walking, slimming and other basic motor behaviours are possible. Such circuits are often built by exploiting neuromorphic hardware. An example is described in [15].
Control	Machine Learning Control (MLC)	subset of machine learning methods to improve control systems, making them optimal. Examples of such techniques include altitude control of satellites and feedback turbulence control in aerodynamics. An example is presented in [16].

Table 1.2: Machine Learning Applications - Part 2. Taken from [50]

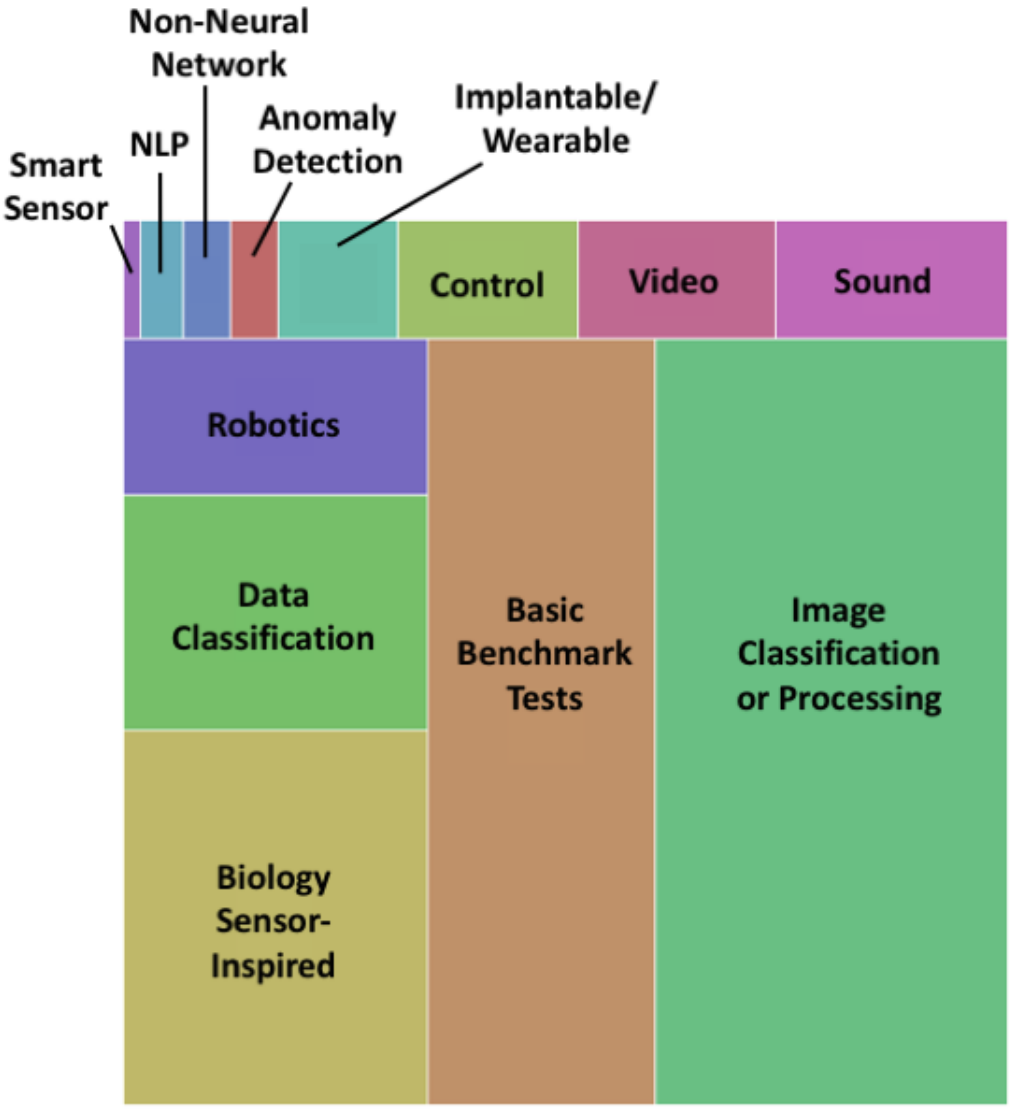

Figure 1.2: A graph showing application domains for which neuromorphic devices have been developed. The size of boxes is proportional to the number of papers that have been published for that domain. Picture taken from [50].

Chapter 2

Technical Background

This chapter covers all concepts and aspects that are needed to understand the thesis workflow, from what machine learning is, how and why it is exploited nowadays, up to employed Instruction Set Architecture and the digital hardware architectures involved.

2.1 Machine Learning and Applications

2.1.1 Machine Learning

The term *Machine Learning* refers to the possibility of building up systems that are able to learn and adapt their parameters, fine tuning their behaviour in order to fulfill a given scope, without being programmed to do so. Then there is *deep learning*, which is a subset of tools taken from the machine learning domain [54], such as neural networks, organized in layers, and brain inspired algorithms, to sample and elaborate information with as many details as possible as the information proceeds from a layer to another [58].

2.1.2 Neural Networks

An Artificial Neural Network (ANN) is a biologically inspired computational model that is used to solve complex tasks. The term *neural network* was coined back in 1944, when Warren McCullough and Walter Pitts, former researchers belonging to the University of Chicago, proposed the first ever computational model of a human brain neuron [37].

As depicted in Figure 2.1, a neuron structure computes a linear combination of inputs, all of which are either 0 or 1, called g, which value is then fed to function

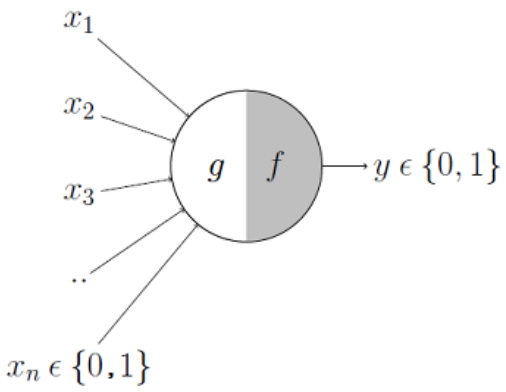

Figure 2.1: McCulloch & Pitts Neuron Model. Picture taken from [8].

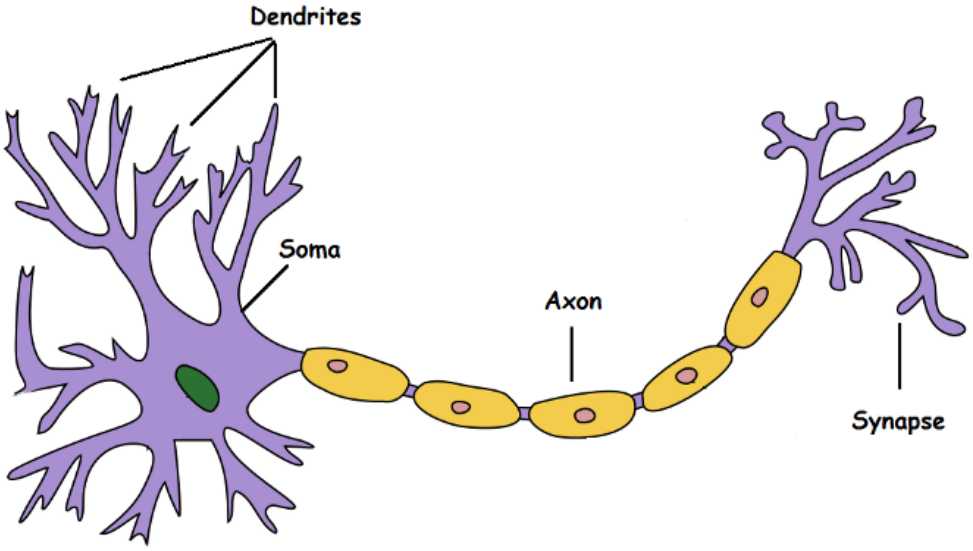

Figure 2.2: Biological Neuron Structure Picture taken from [8].

f, that will determine the outcome y, according to the equations 2.1 and 2.2

$$g(x) = (x_1, x_2, ..., x_n) = \sum_{i=1}^{n} x_i \tag{2.1}$$

$$\begin{cases} f(g(x)) = 1, & \text{if } g(x) \geq \theta \\ 0, & \text{if } g(x) < \theta \end{cases} \tag{2.2}$$

The aforementioned inputs are to be intended as parameters that function f needs to determine whether the neuron has to "fire", where firing is to be nowadays intended as informing all neurons connected to y output that something has

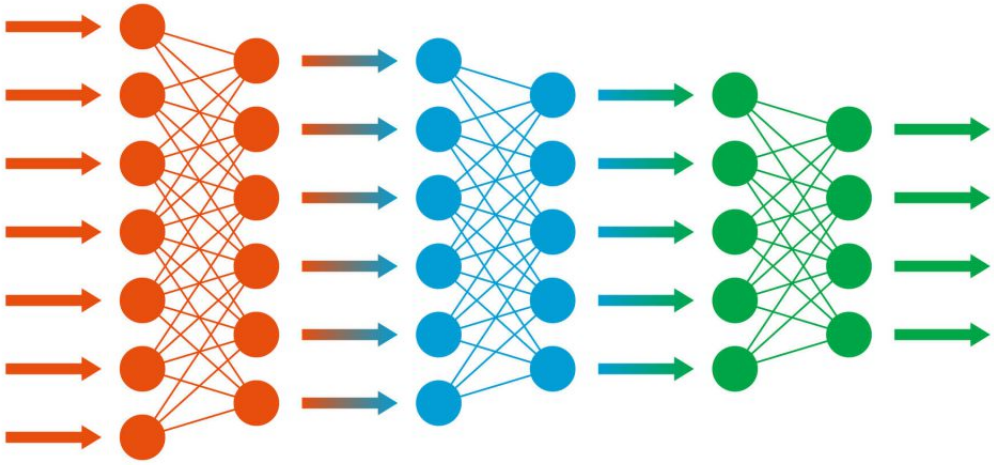

Figure 2.3: Generic Neural Network

happened, typically passing the value of function g as input to them, by means of the synapses. This process takes inspiration from real biological neurons, which internal architecture is briefly sketched in Figure 2.2. As it can be seen, the *dendrites* represent the input parameters, processed by the *soma*, resembling the g function which output is transmitted by means of the axon to a given number of neurons through synapses, that act as interconnection points. As said in Section 2.1.1, a common neural network is organized in layers, composed of a multitude of neurons, which change in number according to the selected layer and are connected according to a given scheme through synapses. As illustrated in Figure 2.3, each layer has the scope of processing a certain aspect of the input information, identify some feature or pattern, and inform the subsequent layer with refined data, up to the point in which the system is able to correctly compute output features from neurons in the rightmost layer. In this regard, it is necessary to introduce the following concepts related to the network learning process

- *parameterized* : a learning algorithm that is characterized by a predetermined number of parameters, which can only change in value during the learning process, so they are independent from the data being fed in.

- *unparameterized* : a learning algorithm that is characterized by a non fixed number of parameters, which can increase or decrease as the learning process goes on, so they are determined by data being fed in.

- *supervised* : direct imitation of a pattern between two datasets, so that the

system feeds in data that is known in order to let the neural network output data the system would like to predict, transforming one dataset into another.

- *unsupervised* : transforms one dataset into another, as if happens in the aforementioned supervised learning, but in this case input data or its characteristics are unknown. This implies that an unsupervised learning algorithm typically clusters input data into groups, according to some specific features it finds during the processing phase.

- *offline* : the input examples set is fixed in size, so the neural network is fed with one example at a time and synapses weights are changed according to a cost function. Once the global cost function is minimized, learning is interrupted and the neural network is deployed. It will just perform inference on the incoming input data, leaving synapses weight fixed for the whole operational time of the device.

- *online* : the neural network model learns and adjust its interconnection after one input data is processed, so the model continuously learn and adapt. Thus, changes made to synaptic weights solely depend on the actual input data and possibly on the current model state.

Let's make an example ([54]). Let's imagine having a set of objects to be fed into geometrically shaped holes (e.g. squares, triangles). babies would probably take an object, and try to put it inside any hole, until it perfectly fits; this is an example of parameterized learning. Teenagers, instead, would probably count the number of sides of each object and look for an hole with that number of sides, before trying to feed the object it; this is unparameterized learning. In most cases, one could say that parameterized learning is all about trial and error, whereas unparameterized one is about counting features. Finally let's have a little digression on the evolution history of Artificial Neural Networks (ANNs).

Spiking Neural Networks

While first and second generation of ANNs have inputs to each neuron consisting of the sum of the incident values multiplied by some weight, third generation ANNs inputs consist of pulse spikes arriving randomly, which indeed are values of arriving spikes multiplied by the weights of the preceptors, hence the name spiking neural networks. First emerged during research to model human brain behaviour, they are still being explored and are not widespread, due to

- lack of efficient supervised learning algorithms

- a so called "killer" application is yet to be identified. A killer application is a task for which SNNs would outperform any other kind of neural network, not only from a power consumption point of view.

Still, there are a few advantages that make them stand out among the possible neural network models

- power consumption. The fact that computation in SNNs consists of spikes which are sparse in time and space and that converting Convolutional Neural Networks (CNNs), which represent state of the art architectures in the computer vision domain, into SNNs provides similar computational capabilities at lower power cost, constitute another reason for which the neuromorphic scientists push towards the widespread of SNNs in domains like that of Internet of Things (IoT), where power capabilities are tightly constrained [52].

- encoding capabilities. As reported in 2.3.7, SNNs allow for the implementation of biologically inspired learning algorithms, which allow for the so called online learning, meaning that synaptic interconnections are reinforced or weakened according to the *cause-effect* relationship that incurs between presynaptic and postsynaptic neurons firing events in the network. Moreover, some algorithms have been developed to provide *unsupervised* online learning, which gave the possibility for reaching state of the art recognition rates when dealing with MNIST database [36]. That said, another advantage comes from the possibility of encoding inputs and outputs either in time or rate of the spikes that characterize neurons.

2.2 The RISC-V Instruction Set Architecture (ISA)

Originally conceived at Berkeley EECS department, RISC-V represents the most discussed Instruction Set Architecture (ISA) among both academic and industry communities, and there are several valid reasons for it to be so:

1. completely open, meaning that specification are publicly available and no royalty fees are due to the RISC-V Foundation, which is a non profit foundation which shall keep the ISA stable, preventing it from being abandoned, as it happened for other closed source ones [42].

2. avoids any dependence on a particular microarchitecture or specific fabrication technology

3. easily extensible to support variants that are more suitable to the desired domain of application, starting from a base yet complete integer ISA (RV32I/RV64I), which is usable as standalone for customized accelerators or educational purposes. This means it should suit all kind of hardware, be it a small microcontroller or a powerful supercomputer.

4. support for highly parallel multicore implementations, such as heterogeneous multiprocessors, optional variable length instructions to both provide a denser instruction encoding and expand available instruction encoding space

5. provides support for hypervisor and privileged instructions, for application domains in which these are required

The name RISC originates from the acronym of the Reduced Instruction Set Computer project, led in the same university by David A. Patterson and Carlo H. Sequin, which aim was exploring an alternative to the general trend toward computers with increasingly complex instruction sets, commonly identified as CISC, which further evolution were accompanied by an always increasing hardware complexity. The RISC approach reduces the instruction set and the available addressing modes, causing larger code size but also reducing design time, number of design errors, and execution time of individual instructions in terms of required clock cycles, being overall advantageous with respect to CISC. The latter family are nowadays maintained as compatibility shell of modern processors, which instead internally rely on a RISC one to improve performance. The **V** in RISC-V underlines the main goal of this ISA, being **variations**, apart from being the fifth instalment in the RISC family developed at Berkeley, whose predecessor were RISC-I, RISC-II, SOAR and SPUR.

A core, that is any component including an independent instruction fetch unit (IFU), serves one or more threads, identified as harts within RISC-V, by means of multithreading of some kind. A core might provide support for additional specialized instructions, either directly or through a so called coprocessor, which is in strict communication with the RISC-V core, performing some domain specific tasks. There are 4 base instruction sets families, each characterized by the width of the integer registers and the corresponding size of the address space and the number of integer registers. Integer registers width is indicated as XLEN. Each one uses two's complement representation for signed integer values. Every ISA version has a RV prefix, followed by the width of employed data and a sequence of letters, among

- I → integer instructions
- E → integer instructions, but registers are restricted to be only 16 (x0-x15). Used for low end implementations for which register files occupy a significant amount of chip area.
- M → integer multiply and divide instructions
- A → 32 bit address space and integer instructions
- F → single precision (32 bit) floating point instructions

- D → double precision (64 bit) floating point instructions
- Q → quad precision (128 bit) floating point instructions. Requires both F and D extensions implementation.
- C → 16 bit compressed integer instructions

Treating the 4 base ISAs as distinct ones has advantages and disadvantages : indeed, it leads more complicate hardware needed to emulate one ISA on another (e.g. RV32I on RV64I), but it allows for an higher degree of optimization for each specific ISA, without the need for each one to support all operations of the one on which it is built upon.
In order to support customization, through the addition of any extension to the chosen basic ISA, three instruction set categories have been outlined: standard, reserved, custom. The former are those defined by the RISC-V Foundation, so instructions which encodings are not interfering with each other; reserved instructions are those that will be used in the future, still undefined. The latter are those that use only custom encodings and extensions, or non conforming extensions, meaning that it uses any standard or reserved encoding space, and are generally available for vendor specific extensions. Any RISC-V implementation must include at least one base integer subset (RV32I, RV32E, RV64I) and any of the above listed extensions. Instructions in the base ISA have the lowest 2 bits set to 11. Compressed C instruction sets have their lowest 2 bits set to 00,01,10, as depicted in Figure 2.4. Instruction sets using more than 32 bits have additional low order bits set to 1 (48 bit r → 011111, 64 bit → 0111111, 80-176 bits → xnnnxxxxx0111111, reserved for at least 192 bits → x111xxxxx1111111).

Figure 2.4: RISC-V Instruction Length Encoding Schemes. Picture taken from [47].

Base ISAs use either little endian or big endian memory systems, with the

privileged architecture further defining bi-endian operations, but the IRAM is filled by 16 bit little endian parcels, regardless of memory system endianess, to ensure that length encoding bits always appear first in halfword address order, allowing the length of a variable length instruction to be quickly determined by a IFU that examines only the first few bits of the first 16 bit instruction parcel.

2.2.1 RV32I - Base Integer ISA

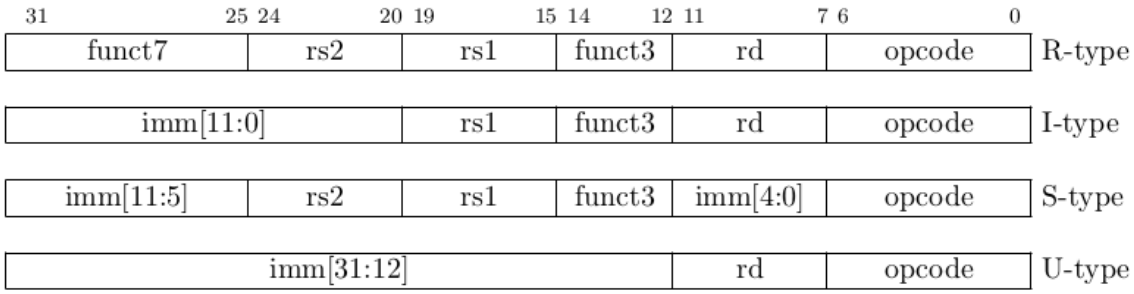

Figure 2.5: RV32I Instruction Formats. Picture taken from [47].

RV32I is the most basic instruction set that is provided and that must be implemented in any case. It was designed to reduce the hardware needed by the most basic application and to support modern operating systems. It handles 32 32-bit registers, from x0, that always contains 0_{10} and can be used to discard an instruction result, to x31, plus the program counter `pc`, all of which are XLEN wide, with XLEN representing the data width in a given ISA version, so 32 bit in this case. Although there is no precise indication on which register must hold the return address of a given subroutine or the actual stack pointer, that is the address of the stack holding the passed subroutine variables from top down, the standard software calling convention uses x1 as link/return register and x5 as alternative one, plus x2 as stack pointer. The instruction formats were designed so to ease the decoding phase as much as possible by:

- keeping source rs1, rs2 and destination rd registers numbers positions fixed between an instruction class and another.
- having immediate MSB always at position 31 (leftmost) to speed up the sign extension operation which involves immediates of all instructions, all located to the left end of the instruction array. The sign extension is not performed on Control Status Register (CSR) instructions. This organization has been chosen because instruction decoding is often part of critical paths in microprocessors, so it helps in reducing them at the expense of moving immediate bits from an instruction format to another, a property already

seen in RISC-IV, also known as SPUR. Zero extension for immediate values is not available as the authors did not find any real advantage in providing it in nowadays applications.

There are a total of four main formats R, I, S, U, plus two variants names SB and UJ, which differ from S and U for the immediate encoding scheme, respectively.

2.2.2 Choices and Consequences

In this part, technical analysis is conducted on the consequences originating from a series of choices that characterize the RISC-V ISA, which ultimately led to choosing it as the target ISA for this work.

- **Cost** : history showed that most companies chose to make their Instruction Set Architectures grow over time, following an approach that is known as *incremental ISA*. It provides for having an instruction set that increases its instruction count over time, always keeping instruction that had been previously developed. The reason is that they are eager to maintain *binary compatibility*, that is the possibility of running very old software on modern processors, no matter the consequences and the strain that derive from such choice. As [42] reports, the consequence for Intel has been to have a tremendous amount of assembly instructions (about 3000), which include a few to support the old fashioned and long abandoned ones to support Binary Coded Decimal (BCD) arithmetic, at the expense of power and occupied area, the latter influencing the *die* cost quadratically. Moreover one should consider that *yield*, that is the number of dies per wafer that are working, decreases as the area of the production wafer increases. On the contrary, the RISC-V architects decided to have a single and basic integer ISA, yet complete for most applications, called *RV32I*, which is guaranteed to be stable, so it won't ever change in the future. To make the ISA suitable for other applications, be it power constrained ones or requiring very high computing capacities, they provide other modular extensions, that can be attached on top of RV32I, keeping them *optional*, and labeling the target ISA according to the extensions being used (e.g. RV32IM indicates the support for multiply instructions), if any. This choice makes it possible for compilers to produce efficient code that better suits the hardware it runs on and guarantees that new instructions will be added by the RISC-V foundation only if there are valid technical reasons that justify the introduction of such instructions, after they are discussed by a dedicated commission.

- **Simplicity** : coming up with complicate instructions, trying to have one macro instruction doing multiple smaller ones , doesn't really pay. Indeed,

most compilers are optimized so to exploit simpler yet effective instructions whenever possible, so they could translate a given program portion in ways one could not expect, as if they neglect specialized instructions that wouldn't carry any advantage with respect to using multiple simpler instructions that serve the same scope.

- **Performance** : although one could think that fewer instructions could imply lower execution time, truth is this is seldom true. Indeed, it is true that a program compiled on a processor supporting a simple, RISC oriented, ISA often leads to more instructions being necessary, but it is also true that either instructions will be executed at a faster rate, because of the higher clock frequency achievable by the underlying processor, or the machine is able to execute more simpler instructions per clock cycle, so it has Cycles Per Instruction (CPI) that is lower with respect to that of the machine exploiting a complex ISA. Moreover one should consider that in the end simple instructions are the most used ones, thus making simplicity a key factor in the choice of an ISA for an application.

- **Isolation of Architecture from Implementation** : an hardware architect should not add features that are particularly helpful in a precise context rather than in another, nor should neglect the consequences that those features might have on other kind of application domains. An example coming from [42] concerns the *delay slot* or *delayed branch* that originates from architects of MIPS-32 ISA. First, it is important to state what jump or branch instructions are, why they are needed and what is the main issue concerning their execution. A jump instruction makes the processor alter the execution flow, bypassing the instruction which comes just after the jump one, and "jumping" to the one which is located at address indicated by the jump instruction. A branch instruction works similarly to a jumping one, but it runs only if a given condition, specified in the instruction, together with the target address, holds true when the branch instruction is fetched and processed. However, the condition outcome can be determined only in a phase that comes after the fetching stage, and it depends on whether the variables (i.e. registers in this case) involved in the condition check are already filled with the needed value or not. This situation may lead to a stall, meaning that the instructions being fetched after the jump or branch one may be stopped for a number of clock cycles that cannot be determined at compile time and strongly depends on the factors discussed above. A delayed branch serves the purpose of having an useful instruction placed right after a branch instruction (i.e. instructions that alter the program flow, so either a branch or a jump), so that the stall that would originate from the execution of the jump or branch instruction is compensated by running an instruction which would be executed in any case, regardless of the branch condition outcome. As the

authors note, this feature helps when there is the need for loading multiple values from the memory in common single instruction fetch processors, but could impact on the performance of superscalar ones, that rely on scheduling of such load instructions in parallel to improve the throughput of the system.

- **Program Size** : a program size, in terms of occupied bytes, is one of major concerns of system architects. Indeed, smaller programs require smaller area on a chip and lower power, as on chip SRAM access needs lower energy with respect to off-chip DRAM, and it often leads to fewer chances of cache misses. As previously discussed, RISC-V standard dictates that instructions are 32 bits wide, although 64 bit and 128 bit extensions are provided, the latter being in ongoing development. [42] states that RISC-V typically leads to programs that are about 10% larger with respect to those written in x86-32 ISA when all instructions are 32 bits long, but the latter proves to be inefficient when dealing with compressed instructions, giving programs that are about 26% larger than RISC-V ones.

- **Ease of Programming** : variables and temporary data are stored into registers, as they are faster in access with respect to common memory. To this extent, RISC-V provides 32 integer registers, whereas other proprietary ones such as ARM-32 or x86-32 have 16 and 8, respectively. Having 32 registers has been proved to be enough for modern applications, although one must note that the cost associated with having an higher number of registers back in the days in which MIPS and ARM were born was too high to make 32 registers affordable. This makes it easier for programmers and compilers to handle complex programs. Moreover, RISC-V instructions typically take one clock cycle, if one ignores cache misses and some other amenities, whereas complex instructions from Intel x86-32 may take way more cycles, and this rises an issue when programming embedded devices, as programmers may want to have more or less precise timing in such applications. Least but not last, modern applications benefit a lot from support for Position Independent Code (PIC). PIC means having a block of code that can be correctly executed regardless of the *absolute address*, that is the compiler assigned a specific and explicit starting address to the block of code. This is generally possible thanks to Program Counter (PC) relative addressing, and it is often used to support *shared libraries*.

- **Room for Growth** : back in the 70s, Gorgon Moore, co-founder of Intel and Fairchild Semiconductor, predicted that *the number of transistors being integrated on a single chip would double every year*, and this remained true until very recently. Back in those days, designers were concerned with squeezing the number of instructions required per program, to reduce the execution time of that program, according to the equation $\frac{instructions}{program} \cdot CPI \cdot f_{ck}$, with f_{ck}

being the processor clock frequency. Today, Moore is slowly fading, mainly due to technological issues that hinder the die manufacturing process. Given this, RISC-V architects chose to make the ISA as modular as possible, having enough room for optional and custom extensions that would help the implementation and usage of domain specific accelerators, such as the one involved in this work. In this regard, a large part of opcode space for base RV32I ISA has been reserved for the integration of specialized coprocessors, that may not need any extension but the basic RV32I and a few specialized instruction to perform the tasks they were designed for [46].

2.3 ODIN : A Spiking Neural Network Coprocessor

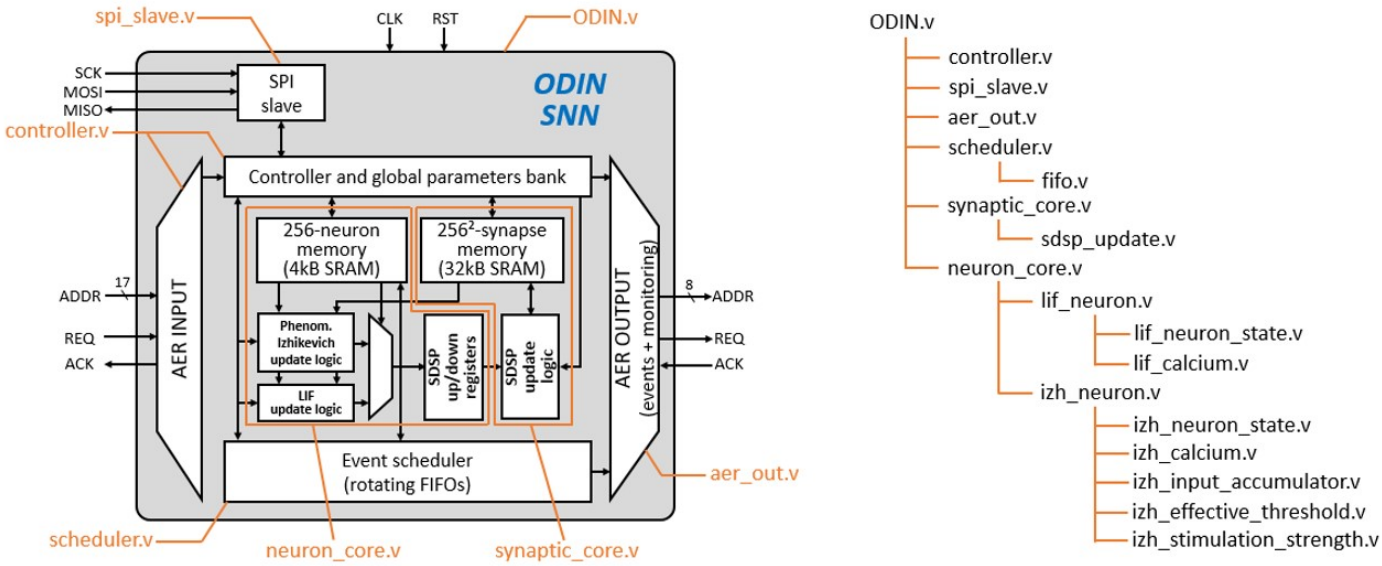

Figure 2.6: ODIN Architecture. Picture taken from [7].

ODIN stands for Online-learning DIgital spiking Neuromorphic processor. It is a neurosynaptic core which supports up to 256 neurons with the possibility o The concept of *neural network* is nowadays spread everywhere. Commonly meant as a mean to provide what is called artificial intelligence, neural network constitute

one of few state of the art technologies which are constantly growing in complexity and efficiency to overcome new challenges, providing fascinating services and features, ranging from image classification and manipulation, up to speech recognition and many more that are still being explored. A neural network is a computing system which takes inspiration from the human brain, exploiting its parallel interconnections to solve complex data problems, modifying its internal parameters (training phase) in order to recognize unknown input data with higher accuracy (inference phase). *Spiking Neural Networks* (SNN) represent an emerging class of neural networks, coming from neuroscience research and aiming at accurately reproducing both static and dynamic behaviours of human brain neurons. Initially conceived to help neuroscientists enrich their knowledge on the human brain structure and working principles, SNNs have gathered attention in the computer science f all-to-all synaptic interconnections, for a total of $2^8 * 2^8 = 64k$ synapses, thus emulating a crossbar which allows every neuron to be connected to every other.

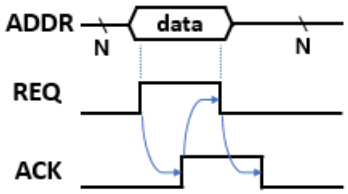

Figure 2.7: AER general working scheme

The communication between ODIN and other external modules happens by means of an input and an output interface that implements the Address Event Representation (AER) protocol. AER stands for Asynchronous Event Representation and it is a protocol originally conceived for neuromorphic devices to allow exchanging information on asynchronous events between computing devices running operations in parallel. As depicted in Figure 2.7, an AER transaction consists of

- *ADDR*: an N bit wide bus providing an event "address", which is made up of control and data bits.

- *REQ*: a single line that is asserted once *ADDR* is ready and stable, in order to request attention for that event to the module it is connected to. When the *ACK* signal goes to logic 1, then the *REQ* line is lowered to logic 0.

- *ACK*: a single line that is asserted once the requested event has been received and completely processed. As soon as the *REQ* signal is de-asserted, this line can be driven to logic 0 as well.

The lines can be an input or an output, depending on whether they are used to receive data from external devices or to provide it to other modules, as shown in Figure 2.6.
Although the standard [55] dictates that every involved unit must be identified through a unique address, in this work ODIN is present as the only AER compatible device, so there is not really need to define such address. An address event consists in packing up information of a given spiking event and transmitting it over to ODIN. The message packet contains various information, depending on the event being represented and trasmitted, as will be detailed in subsections concerning AER OUTPUT and CONTROLLER modules. The coprocessor architecture is illustrated in 2.6 and consists of the following modules

2.3.1 Supported Neurons Models

ODIN supports two neuron models, the former being well known and simple, whereas the latter is a custom model that is able to reproduce main Izhikevich behaviours while being way more computationally efficient with respect to standard Izhikevich or Hodgkin & Huxley ones, due to the fact that the neuron is event driven, and the state is updated only when presynaptic neurons fire or time references trigger it [21].

Leaky Integrate & Fire Model

Originally described by Louis Lapicque, this model is an extension of the classical Integrate & Fire (IF), which adds a term $\frac{V_m(t)}{R_m}$ to $C_m\frac{dV_m(t)}{dt} = I_s(t)+I_{inj}(t)-\frac{V_m(t)-V_0}{R_m}$ equation [6], with V_m being the membrane voltage, R_m is the membrane resistance, $I_s(t)$ indicates the membrane input synaptic current, $I_{inj}(t)$ is a current injected into the neuron by an intracellular electrode, and V_0 is the membrane resting potential. That leaky term takes into account the leaky phenomenon which characterize the neuron membrane, that is not a perfect insulator, leading to its potential decaying over time. The neuron increases its potential from time to time, according to the stimuli given by I_s, until it reaches a given threshold V_{th}, after which the membrane potential is reset to V_0 and the neuron "fires", that is an impulse is emitted at the output (i.e. through the axon) and transmitted to all neurons that are connected to it, according to the synaptic weights. Since the models has just one equation, it cannot exhibit dynamics such as bursting or phasic spikes, plus it only fires spikes with a fixed latency, given by the V_{th} threshold ([31]). Section 3.3.2 at [7] contains a complete description of parameters to be set up for this neuron model.

Custom Phenomenological Izhikevich Inspired Model

A custom neuron model has been designed in order to *bridge the area gap between digital and analog implementations of neuron models* [21]. It allows for reproducing

20 Izhikevich behaviours [31], and it is event driven, reducing the computational effort and resources needed to handle the Hodgkin-Huxley and Izhikevich neuron models equations. At the time being, details on its parameters and working principles are not yet openly published by the author of ODIN, but some effort has been put into extracting some information from the verilog files. Although the documentation inside [7] is sufficient to the extent of the present work, a table listing all parameters of the custom phenomenological Izhikevich based neuron model, which is reported in Tab 2.1 and Tab 2.2 for convenience and future usage.

2.3.2 SPI Slave

The Serial Peripheral Interface (SPI) slave unit is responsible for properly setting ODIN configuration registers, which are listed in Table 2.3, writing to and reading from either neurons or synapses memories. The communication protocol strictly adheres to SPI standard, according to [59], and described on page 29. Both CPOL and CPHA are set to 0, so that transmitted data can change only at falling edge of the clock, whereas it is sampled at the rising edge. In order to have a working transmission and ensure that the system operations have sufficient time to elaborate sampled data, ODIN requires the SCK clock frequency to be at least four times lower than that of the ODIN main clock source, since some operations, such as that needed to handle neurons and synapses memory operation, lasts 4 clock cycles overall (e.g. WAIT → W_NEUR (memory addressing phase) → W_NEUR (memory write phase) → W_SPIDN).

Serial Peripheral Interface

The Serial Peripheral Interface, commonly indicated as SPI, is a serial communication protocol originally conceived by **Motorola Inc.** in 1972. This protocol, as well as I^2C, can be applied only to short interconnection (e.g. same board, or between different boards that are very close together), because of the skew that limits the maximum performance. Also known as 3-wire serial interface, it is a full duplex transmission, i.e. it allows for a concurrent communication between master and slave and between slave and master allowing for only one master and multiple slaves, typically up to 5, because of costs. There 4 types of wire:

- Master Out Slave In (MOSI), that is the wire for data from master to slaves
- Master In Slave Out (MISO), that is the wire for data from slaves to master. There is only one MISO, so there must be a way to avoid conflicts because of multiple wires trying to drive it.
- SPICLK, the clock wire, always driven by the master

Bits	Parameter/State Variable	Name	Width	Description
0	Parameter	lif_izh_sel	1 bit	Selects the neuron model. 0 for Custom Izhikevich, 1 for LIF.
7-1	Parameter	leak_str	7 bits	Defines the amount by which the membrane potential is decreased in the case of time reference l leakage event. The membrane potential cannot go negative.
8	Parameter	leak_en	1 bit	Enables the leakage mechanism.
11-9	Parameter	fi_sel	3 bits	accumulator depth parameter for fan-in configuration.
14-12	Parameter	spk_ref	3 bits	number of spikes per burst.
17-15	Parameter	isi_ref	3 bits	inter-spike-interval in burst.
18	Parameter	reson_sharp_en	1 bit	inter-spike-interval in burst.
21-19	Parameter	thr	3 bits	neuron firing threshold.
24-22	Parameter	rfr	3 bits	neuron refractory period.
27-25	Parameter	dapdel	3 bits	delay for spike latency or for DAP parameter.
28	Parameter	spklat_en	1 bit	spike latency enable.
29	Parameter	dap_en	1 bit	DAP enable.
32-30	Parameter	stim_thr	3 bits	simulation threshold (phasic,mixed,etc).
33	Parameter	phasic_en	1 bit	phasic behaviour enable.
34	Parameter	mixed_en	1 bit	mixed mode behaviour enable.
35	Parameter	class2_en	1 bit	class 2 excitability enable.
36	Parameter	neg_en	1 bit	negative state enable enable.
37	Parameter	rebound_en	1 bit	rebound behaviour enable.
38	Parameter	inhin_en	1 bit	inhibition-induced behaviour enable.
39	Parameter	bist_en	1 bit	bistability behaviour enable.
40	Parameter	reson_en	1 bit	resonant behaviour enable.
41	Parameter	thrvar_en	1 bit	threshold variability and spike frequency adaption behaviour enable.
42	Parameter	thr_sel_of	1 bit	selection between O (0) and F (1) behaviors parameter (according to Izhikevich behavior numbering).
46-43	Parameter	thrleak	3 bits	threshold leakage strength.
47	Parameter	acc_en	1 bit	accommodation behaviour enable (requires threshold variability enabled).
48	Parameter	ca_en	1 bit	Calcium concentration enable.
51-49	Parameter	thetamem	3 bits	Defines the SDSP threshold on the membrane potential.
54-52	Parameter	ca_theta1	3 bits	Defines the first SDSP threshold on the Calcium variable.
57-55	Parameter	ca_theta2	3 bits	Defines the second SDSP threshold on the Calcium variable.

Table 2.1: Custom Izhikevich Neuron Parameters - Part 1

Bits	Parameter/State Variable	Name	Width	Description
60-58	Parameter	ca_theta3	3 bits	Defines the third SDSP threshold on the Calcium variable.
65-61	Parameter	ca_leak	5 bits	Defines the Calcium variable leakage time constant: a number of time reference events must be accumulated for the Calcium variable to be decremented. If set to 0, disables Calcium leakage.
66	Parameter	burst_incr	1 bit	amount by which effective threshold and calcium are incremented at every burst event.
69-67	Parameter	reson_sharp_amt	3 bits	sharp resonant behaviour time constant.
80-70	State	inacc	11 bits	input accumulator state.
81	State	refrac	1 bit	tells whether neuron is in refractory period or not.
85-82	State	core	4 bits	contains the current value of the membrane potential.
88-86	State	dapdel_cnt	3 bits	dapdel counter state.
92-89	State	stim_thr	4 bits	stimulation strength state.
96-93	State	stim_str_tmp	4 bits	temporary stimulation strength state.
97	State	phasic_lock	1 bit	phasic lock state.
98	State	phasic_lock	1 bit	mixed lock state.
99	State	spkout_done	1 bit	tells whether in current time reference interval a spike was fired or not.
101-100	State	stim0_prev	2 bits	zero stimulation monitoring state.
103-102	State	inhexc_prev	2 bits	inhibitory/excitatory stimulation monitoring state.
104	State	bist_lock	1 bit	bistability lock state.
105	State	inhin_lock	1 bit	inhibition-induced lock state.
107-106	State	reson_sign	2 bits	resonant sign state.
111-108	State	thrmod	4 bits	threshold modificator state.
115-112	State	thrleak_cnt	4 bits	threshold leakage state.
118-116	State	calcium	3 bits	Contains the current value of the Calcium variable.
123-119	State	caleak_cnt	5 bits	Contains the current value of the Calcium variable leakage counter (cfr. ca_leak parameter).
124	State	burst_lock	1 bit	burst lock state.
127	Parameter	neur_disable	1 bit	Disables the neuron.

Table 2.2: Custom Izhikevich Neuron Parameters - Part 2

Register Name	Address <15:0>	Width	Description
SPI_GATE_ ACTIVITY	0	1 bit	Gates the network activity and allows the SPI to access the neuron and synapse memories for programming and readback.
SPI_OPEN_ LOOP	1	1 bit	Prevents spike events generated locally by the neuron array from entering the scheduler, they will thus not be processed by the controller and the scheduler only handles events received from the input AER interface. Locally-generated spike events can still be transmitted by the output AER interface if the SPI_AER_SRC_CTRL_nNEUR configuration register is de-asserted.
SPI_SYN_ SIGN	2..17	256 bits	Configures each of the 256 ODIN neurons as either inhibitory (1) or excitatory (0), i.e. all of their downstream synapses either take a negative (1) or positive (0) sign.
SPI_ BURST_ TIMEREF	18	20 bits	Defines the number of clock cycles required to increment the scheduler inter-spike-interval counter. Useful only if the neuron bursting behavior is used (i.e. Izhikevich model only), otherwise this register should be set to 0 to save power in the scheduler.
SPI_AER_ SRC_CTRL_ nNEUR	19	1 bit	Defines the source of the AER output events when a neuron spikes: either directly from the neuron when the event is generated (0) or from the controller when the event is processed (1). This distinction is of importance especially if SPI_ OPEN_LOOP is asserted.
SPI_OUT_ AER_ MONITOR_ EN	20	1 bit	Enables automatic neuron and synapse state monitoring through the AER output link (Refer to 2.3.4).
SPI_ MONITOR_ NEUR_ ADDR	21	8 bits	Neuron address to be monitored if SPI_OUT_AER_ MONITOR_EN is asserted.
SPI_ MONITOR_ SYN_ADDR	22	8 bits	Synapse address of the post-synaptic neuron SPI_ MONITOR_NEUR_ADDR to be monitored if SPI_ OUT_AER_MONITOR_EN is asserted.
SPI_ UPDATE_ UN- MAPPED_ SYN	23	1 bit	Allows SDSP online learning to be carried out in all synaptic weights, even in synapses whose mapping table bit is disabled (Refer to Section SDSP ONLINE LEARNING).
SPI_PROP- AGATE_ UN- MAPPED_ SYN	24	1 bit	Allows all the synaptic weights to be propagated to their post-synaptic neuron, independently of the mapping table bit value.
SPI_SDSP_ ON_SYN_ STIM	25	1 bit	Enables SDSP online learning for synapse events (Refer to Section 2.3.3).

Table 2.3: SPI Module Configuration Registers

- SPISS, that is the slave select wire, connected to the Chip Select input of slave and driven by the master. Of course there might be multiple SPISS,one per each slave, or they may be absent in case of daisy chain connections. A daisy chain network has a master and all slaves are connected in series, so that the MOSI of one slave becomes the MISO of the following one, up to the point in which the last slave MISO is connected to the master. An example of daisy chain configuration is depicted in Figure 2.8.

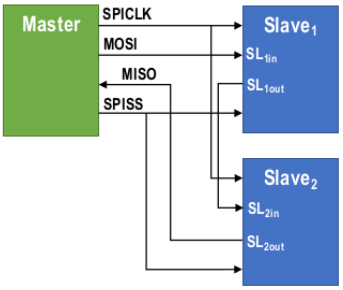

Figure 2.8: SPI with daisy chain configuration

Note that master and slaves are always both transmitters and receivers; indeed, a master is just a block that initiates the transmission, but it doesn't mean it cannot act as a receiver. Furthermore, if one data direction wire for a block is not needed, either MOSI or MISO can be removed, but internal registers will need to have input fixed either to high or low logic value. The slave select of the target slave is active during the whole transmission, while others must have their SPISS deactivated and their MISO in high impedance, ignoring whatever happens on SPICLK and MOSI. The master will generate n clock cycles, transmitting data starting from MSB; data is generated on a precise clock edge and is sampled on the opposite edge, according to the following parameter

- CPHA=0 → data change on trailing (second) edge, and is sampled on the leading (first) one
- CPHA=1 → data change on leading (first) edge, and is sampled on the trailing (second) one

Once the LSB is received, the slave SPISS is deactivated, and SPICLK returns to idle level.

Master and slaves must agree on the number n of bits to be transferred, which is typically equal to 8. One bit is transferred per each clock cycle, exploiting NRZ-L encoding. The SPICLK is idle while there is nothing to be transmitted, according to the value of parameter CPOL

- CPOL=0 → clock is low while idle

CPOL	CPHA	Idle CLK	Data output	Sampling
0	0	0	Falling	Rising
0	1	0	Rising	Falling
1	0	1	Rising	Falling
1	1	1	Falling	Rising

Figure 2.9: CPOL and CPHA define three parameters of the connection

- CPOL=1 → clock is high while idle

So in the end, before setting up a SPI interconnection, master and slaves shall agree on number n of transmitted bits, clock frequency, CPOL and CPHA. CPOL and CPHA define the idle clock level, the active edge for data output and the active edge for data sampling, according to the table in Figure 2.9. SPI doesn't need arbitration mechanism, as there is only one master; addressing is obtained through chip select signals and bit synchronization is achieved since the clock is only generated by the master. However, a slow slave cannot stop the master, so the SPICLK must be set to a lower frequency in order to make things work, and there is no error checking protocol. Furthermore, the only shared wire is MISO, which can be driven by multiple slaves, for which conflict should be avoided through careful addressing, making sure that only one SPISS is active at a given time. Finally, performance is in the order of tens of Mbit/s, and it can be implemented only on interconnections up to tens of centimeters (e.g. SD card memory).

2.3.3 Controller

The whole system is administered by a controller. It is a Moore Finite State Machine (FSM), as its outputs solely depends on the current state, that moves from a state to another, depending on the ongoing operations. States are listed hereby

- WAIT : this state makes ODIN waiting until the AEROUT bus is freed, as it is transmitting a number of AER transactions.
- W_NEUR : a write operation has been requested on neurons memory. It lasts two clock cycles, as in the first one the memory is enabled through the Chip Select (CS), then the write request is granted by asserting the Write Enable (WE) signal of that memory.
- R_NEUR : a read operation has been requested on neurons memory. It lasts two clock cycles, as in the first one the memory is enabled through the Chip Select (CS), then the memory latches the and gives back the word at that address in the subsequent clock cycle.

- W_SYN : a write operation has been requested on synapses memory. It lasts two clock cycles, as in the first one the memory is enabled through the Chip Select (CS), then the write request is granted by asserting the Write Enable (WE) signal of that memory.
- R_SYN : a read operation has been requested on synapses memory. It lasts two clock cycles, as in the first one the memory is enabled through the Chip Select (CS), then the memory latches the address and gives back the word at that address in the subsequent clock cycle.
- TREF : a "time reference" event has been detected coming through input AER bus, and will take two clock cycles, plus the one in which the state changes to TREF, in order to be completed.
- BIST : a "bistability" event has been detected coming through input AER bus, and will take two clock cycles, plus the one in which the state changes to BIST, in order to be completed.
- SYNAPSE : a "single synapse" event has been detected coming through input AER bus, and will take two clock cycles, plus the one in which the state changes to SYNAPSE, in order to be completed.
- PUSH : the controller has detected an incoming AER event, that is either "virtual" or "neuron". In case any of these events must be handled, their information is pushed onto the scheduler in one clock cycle and will be processed according to the scheduler algorithm, so not necessarily in the subsequent cycle.
- POP_NEUR : this state is reached either when the scheduler has at least one event yet to be processes, or it cannot hold any more events due to the limited FIFOs memory available. If the FSM transitions to POP_NEUR, then a "neuron spike" event has to be processed and removed from the scheduler FIFOs once it has been successfully handled. It takes N*2 clock cycles, for the same reasons detailed for states W_NEUR or W_SYN, where N is the number of ODIN supported neurons (i.e. N = 256 by default).
- POP_VIRT : this state is reached either when the scheduler has at least one event yet to be processes, or it cannot hold any more events due to the limited FIFOs memory available. If the FSM transitions to POP_NEUR, then a "neuron spike" event has to be processed and removed from the scheduler FIFOs once it has been successfully handled. It takes two clock cycles, for the same reasons detailed in states W_NEUR or W_SYN.
- WAIT_SPIDN : this is the state in which the controller moves when any memory operation related to neurons or synapses has been taken care of.

Indeed, each of these operations lasts 40 SPI clock cycles, so the controller leaves the state only as soon as the last bit of data on SPI bus has been correctly loaded.

- WAIT_REQDN : this is the state in which the controller moves after an input AER event has been either loaded into the scheduler (i.e. previous state is PUSH) or it has just ended (e.g. BIST event), so the controller shall wait for input AER REQ signal to be driven low before moving on.

The controller also handles AER requests coming from the input AER signals, according to events listed in Table 2.4 and Table 2.5.

Address <16>	**Address <15:8>**	**Address <7:0>**	**Event Type**	**Number of Cycles**	**Description**
1	pre_neur <7:0 >	pre_neur <7:0 >	Single Synapse	2	Stimulates neuron at address post_neur <7:0 >with the synaptic weight associated to pre-synaptic neuron address pre_neur <7:0 >. Ignores the value of the mapping table bit.
0	neur <7:0 >	0xFF	Single Neuron Time Reference	2	Activates a time reference event for neuron neur <7:0>only.
0	Don't Care	0x7F	All Neurons Time Reference	2 * 256	Activates a time reference event for all neurons.
0	neur <7:0 >	0x80	Single Neuron Bistability	128	Activates a bistability event for all synapses in the dendritic tree of neuron neur <7:0>.
0	Don't Care	0x00	All Neurons Bistability	128 * 256 = 32k	Activates a bistability event for all synapses in the crossbar array.

Table 2.4: AER Input Supported Events - Part 1

2.3.4 AER Output

This module handles all outgoing AER transactions, in order to communicate with another neuromorphic processor or let external modules receive information on ODIN neurons and synapses status.

Address <16>	**Address <15:8>**	**Address <7:0>**	**Event Type**	**Number of Cycles**	**Description**
0	pre_neur <7:0 >	0x00	Neuron Spike	1+ (2 * 512)	Stimulates all neurons with the synaptic weight associated to pre-synaptic neuron pre_neur <7:0>. Takes the mapping table bit into account. Neuron spike events go through the scheduler (1 cycle for a push to the scheduler, 512 cycles for processing when the event is popped from the scheduler).
0	neur <7:0 >	{w<2:0>, s,l,001}	Virtual Spike	1+2	Stimulates a specific neuron with weight w<2:0>and sign bit s (1: inhibitory, 0: excitatory), without activating a physical synapse. If the leak bit l is asserted, weight information is ignored and a time reference l leakage event is triggered in the neuron instead. Virtual events go through the scheduler (1 cycle for a push to the scheduler, 2 cycles for processing when the event is popped from the scheduler).

Table 2.5: AER Input Supported Events - Part 2

2.3.5 Scheduler

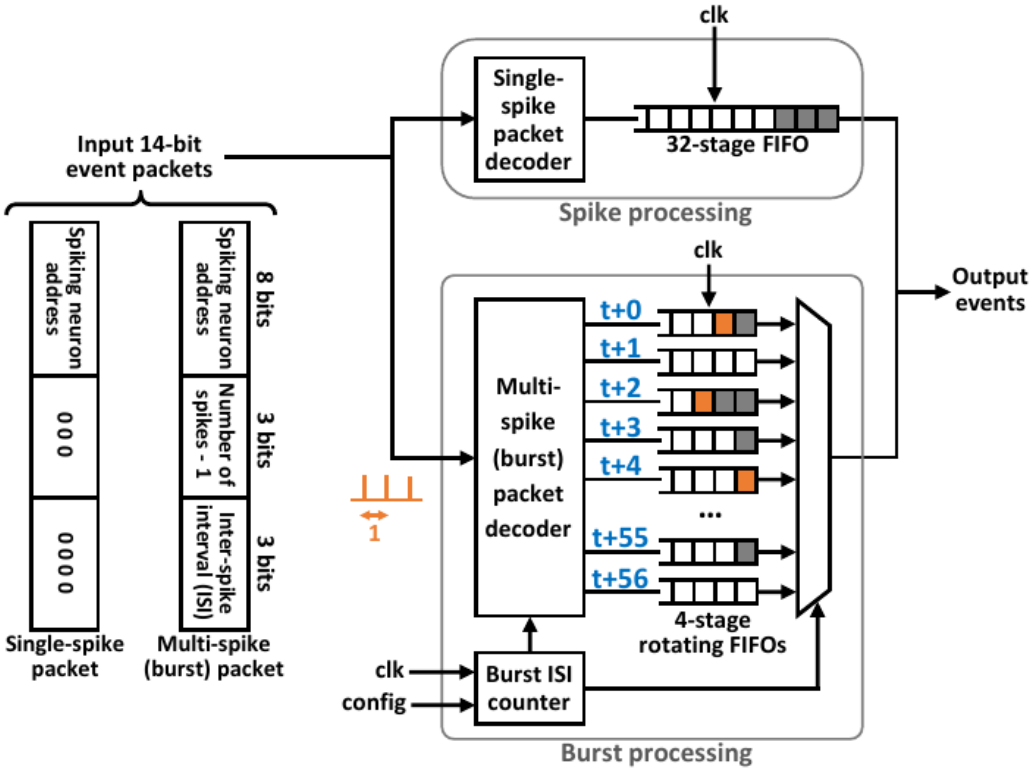

Figure 2.10: ODIN Scheduler. Picture taken from [21].

The scheduler represents one of two crucial ODIN components, the other one being the FSM based controller, and is sketched in Figure 2.10. It is inspired to priority based ordered First In First Out memory structures [39]. The scheduler is responsible for handling spiking and bursting (i.e. multi ple spikes in sequence) events, distinguishing whether they come from ODIN neurons or from other neuromorphic devices through the input AER interface. Every spiking event is encoded into a 14 bit wide packet, consisting of

- *spiking neuron address*
- *number of spikes - 1*
- *inter spike interval (ISI)*

All single spike events are stored into the main FIFO, which can accommodate 32 events, each one storing the address of the neuron that fires. Moreover the main FIFO has the highest priority, so its events are served as soon as they are available, even if there are some bursting events. The latter are stocked into 57 secondary

FIFOs, each one having space for 4 events, each one storing the corresponding neuron address. For an exhaustive description, please refer to Section 2.2.1.3 of [21].

2.3.6 Neuron Core

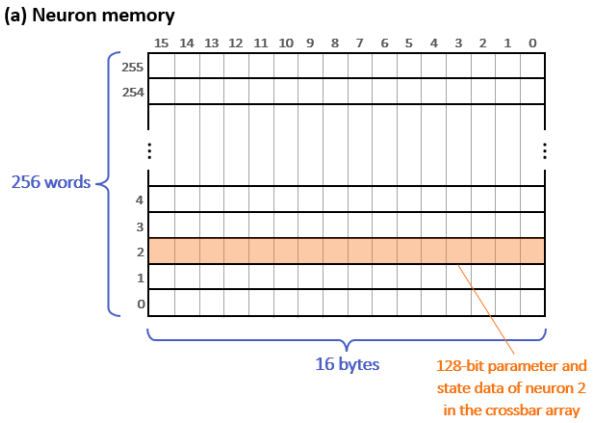

Figure 2.11: ODIN Neurons SRAM Organization. Picture taken from [7].

The neuron core handles events related to neurons state change. In particular it provides information taken from the neurons state SRAM to the neurons update logic blocks, so that it can determine whether the neuron has to fire or not, and updates the SRAM accordingly. The neurons state SRAM is made of 2^8 128 bits wide words, for a total of 4 kBytes of memory. The memory layout is depicted in Figure 2.11. The memory input address corresponds to the 8 bit wide neuron address, whereas the byte to be written or read is selected according to a 4 bits wide selection address which is give through SPI when an operation on neurons memory is requested.

2.3.7 Synaptic Core

The synaptic core is strongly bound to information taken from and given to neuron core. Indeed it provides proper synaptic weight to the latter and receives data needed to determine whether the synapse associated to the event being handled in the system is going to grow, as it made the neuron membrane potential increase, or shrink, as it made the potential decrease. The memory is organized as illustrated in Figure 2.12. The address is formed by intersecting the 8 bits presynaptic neuron address PRE_NEUR[7:0] with the 8 bit postsynaptic one POST_NEUR[7:0], taking all bits from the presynaptic one and concatenating them with bits [7:3] of postsynaptic one, forming a 13 bit wide address; bits [2:1] of postsynaptic neuron address select one of the 4 available word bytes, and the least significant bit

Bitwise location in word	Parameter/State	Name	Width	Description
0	Parameter	lif_izh_sel	1 bit	Selects the neuron model. 0 for Custom Izhikevich, 1 for LIF.
7-1	Parameter	leak_str	7 bits	Defines the amount by which the membrane potential is decreased in the case of time reference l leakage event. The membrane potential cannot go negative.
8	Parameter	leak_en	1 bit	Enables the leakage mechanism.
16-9	Parameter	thr	8 bits	Defines the firing threshold: a spike is issued and the neuron is reset when the membrane potential reaches the *thr* value.
17	Parameter	ca_en	1 bit	Enables the Calcium variable and SDSP online learning in the dendritic tree of the current neuron.
25-18	Parameter	theta_mem	8 bits	Defines the SDSP threshold on the membrane potential.
28-26	Parameter	ca_theta	3 bits	Defines the first SDSP threshold on the Calcium variable.
31-29	Parameter	ca_theta2	3 bits	Defines the second SDSP threshold on the Calcium variable.
34-32	Parameter	ca_theta3	3 bits	Defines the third SDSP threshold on the Calcium variable.
39-35	Parameter	ca_leak	5 bits	Defines the Calcium variable leakage time constant: a number ca_leak of time reference events must be accumulated for the Calcium variable to be decremented. If set to 0, disables Calcium leakage.
77-70	State	core	8 bits	Contains the current value of the membrane potential.
80-78	State	calcium	3 bits	Contains the current value of the Calcium variable.
85-81	State	caleak_cnt	5 bits	Contains the current value of the Calcium variable leakage counter.
127	Parameter	neur_disable	1 bit	Disables the neuron if set to 1.

Table 2.6: ODIN Neuron State

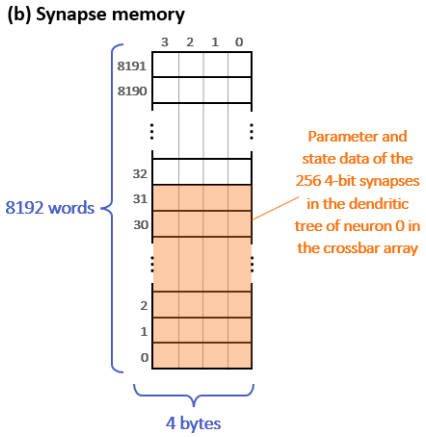

Figure 2.12: ODIN Synapses SRAM Organization. Picture taken from [7].

determines whether the upper or lower half of the selected byte is to be written or read, according to the requested memory operation. However it seems this is not true in the implemented architecture, as the mechanism to choose either the upper half or lower part of the selected byte is based on a byte of masking bits, as described in 2.3.2. In order to benefit from the usage of high density SRAMs, each word consists of 32 bits, storing a total of 8 synapses data blocks. Indeed, each synapse is characterized by a 3 bit wide weight value and a so called *mapping table* bit, which serves the purpose of allowing a given synapse to exploit online SDSP learning mechanism or not, thus being static in value.

Synaptic Plasticity and other Synaptic Features

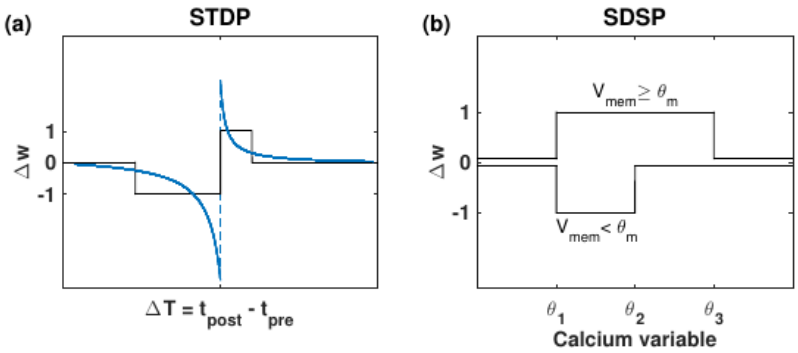

Figure 2.13: STDP vs SDSP Learning. Picture taken from [21].

In the neuroscience domain, the term *plasticity* refers to the possibility for biological systems to modify synaptic strengths from time to time, according to

neurons activity. Spike Timing Dependent Plasticity (STDP) is a popular synaptic learning algorithm that improves or weakens synaptic interconnections between neurons according to firing events. As illustrated in Figure 2.13, the learning rule increases the synaptic weight if the postsynaptic neuron fires after the presynaptic one has fired, or decreases it if the posysynaptic one fires before the presynaptic one does. Thus, the process tends to favor communications between couples of neurons where the presynaptic one contribute to the possibility of making the postsynaptic one fire. Equations 2.3 describe the STDP behaviour in updating synaptic weights when needed

$$\begin{cases} w \to w + A_+ e^{-\frac{\Delta T}{\tau_+}}, & \text{if } \Delta T > 0 \\ w \to w + A_- e^{-\frac{\Delta T}{\tau_-}}, & \text{if } \Delta T < 0 \end{cases} \tag{2.3}$$

with A_+, A_- being amplitudes that can be modified to scale the effects of the learning phase, ΔT is $t_{postsynaptic} - t_{presynaptic}$ and τ_+, τ_- are time constants for which details can be found in [5]. The Spike Driven Synaptic Plasticity (SDSP) is a simplification of STDP. It consists in evaluating the new state of the postsynaptic neuron at the time in which the presynaptic one fires, according to equations 2.4

$$\begin{cases} w \to w + A_+, & \text{if } V_{mem}(t_{presynaptic}) \geq \theta_m, \theta_1 \leq Ca(t_{presynaptic}) < \theta_3 \\ w \to w + A_-, & \text{if } V_{mem}(t_{presynaptic}) < \theta_m, \theta_1 \leq Ca(t_{presynaptic}) < \theta_2 \end{cases} \tag{2.4}$$

The equations concerning the Ca variable refer to a rule to limit the learning process to run under well established conditions, as described in [5]. Indeed, the Ca variable indicates the calcium concentration in the neuron and gives some insights into recent firing activity of the neuron. If it exceeds the ranges delimited by $\theta_1, \theta_2, \theta_3$, then the learning process could lead to a phenomenon called *overfitting*, which means that the employed neural network state evolved so that is it absolutely able to correctly predict when fed with input samples on which it was trained during the learning process, but the prediction accuracy lowers a lot when the neural network handles previous unseen data samples. This implies that the model actually memorized the input examples, rather than the correlation that exists between inputs and outputs. The rule formulated by [5] belongs to the *early stopping* methods, a modern approach to prevent overfitting from happening. Last but not least, the author implemented a set of rules to handle bistable synapses in such networks. The solution is originally described in [29] and furthermore modified in [5]. Indeed, Complementary Metal Oxide Semiconductor (CMOS) technology is ubiquitous when dealing with digital designs, but is not really suited for storing analog values for a relatively long amount of time, due to parasitic capacitances and leaky currents that affect such structure. Thus, stored values may substantially change over time and this is highly unacceptable. The adopted solution provides for a comparing circuit which matches the actual synaptic weight against a fixed threshold and either increase or decreases the weight if the actual value is above or below

that threshold, respectively. Please note that bistability and membrane potential leakage mechanisms (the latter being commonly indicated as leakage mechanism in [7]) are not automatically handled from any synaptic related logic, but specific events have to be generated and sent to ODIN AER input interface, according to Tab 2.4 and Tab 2.5.

2.4 Chipyard Hardware Design Framework

Figure 2.14: The Chipyard project logo

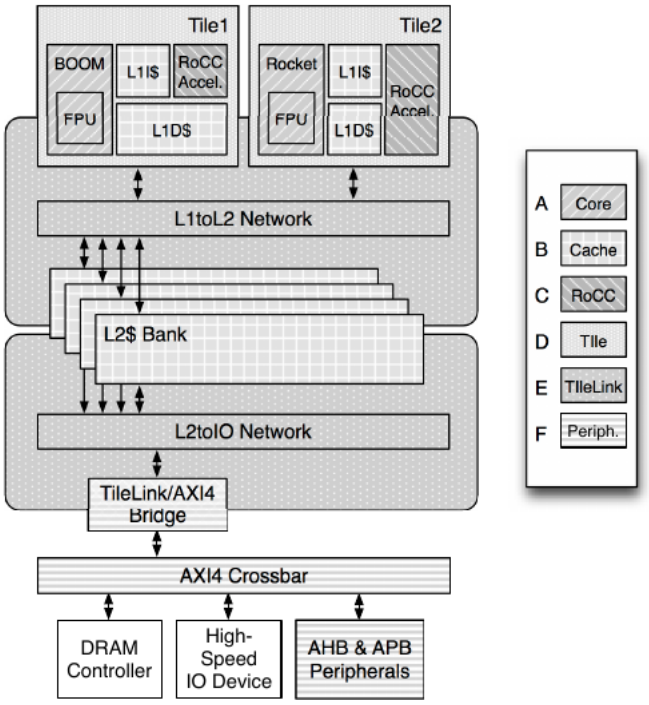

Figure 2.15: An example of two Rocket tiles, one with Berkeley Out of Order Machine (BOOM) core and the other with a standard Rocket core, inside a complete SoC. [2]

Chipyard is the name people at Berkeley Architecture Research gave to an hardware design framework which enables for composing a RISC-V based architecture as complex as you may think of. Starting from the previously conceived RocketChip generator, which allows for producing a 5 stage in order RISC-V core with a great degree of customization, chipyard makes a clear statement: "As the world around us keeps growing its demand for new features, devices responsible for providing those ones are too complex to be designed at the RTL level. There is the need for something which eases this task, among others, and lets companies and research groups alike build at an higher abstraction level, delegating the task of writing clunky HDL code to an autonomous system, of which chipyard just represents one of the many more to come".

2.4.1 Rocket Core

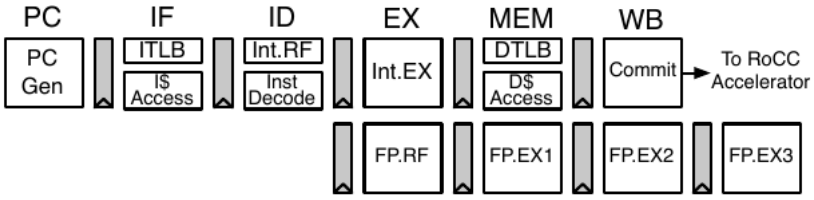

Figure 2.16: The Rocket core pipeline [2]

Rocket is an in order microprocessor that supports RV64GC, with G meaning that it supports base integer instruction with 32 registers all 32 bits wide(32I), integer multiply and divide (M), atomic ones (A), and both single and double precision floating point operations (F and D, respectively). A Rocket tile, which is the basic microprocessor being instantiated in Chipyard, is composed of both L1 and L2 data and instruction caches. Moreover, it consists of a configurable number of Translation Lookaside Buffers (TLBs), pipelined floating point units, a Memory Management Unit (MMU) with support for virtual page based memory and branch prediction mechanisms, the latter being provided through Branch Target Buffer (BTB), Branch History Table (BTH) and Return Address Stack(RAS). The whole core pipeline is shown in Figure 2.16and a complete overview of rocket tiles connection inside the generated SoC is represented in Figure 2.15.

TileLink Coherent Bus Protocol

TileLink [51] is the name of the open source RISC-V oriented interconnect standard used in Chipyard. It provides support for other ISAs, multiple masters, meaning that multiple units can initiate a transfer operation at the same time, and is conceived with the purpose of using it with specialized accelerators, not general purpose

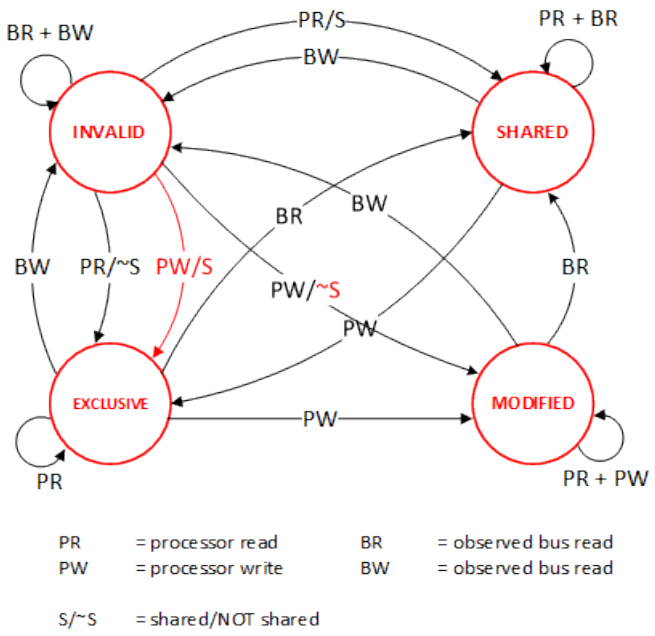

Figure 2.17: MESI Cache Coherence Protocol - State Diagram. Picture taken from [32].

multicore processors only. It provides a physically addressed, shared memory system, with coherent access to both memories and slave device. Memory coherence indeed refers to *cache coherency*. Indeed, as long as the cores do not have data caches, if one writes some data to the shared memory, the others will read the most updated value, so coherency is guaranteed. However, if one or more cores have some cache, this might not be the case. In fact, if one core updates a given chunk of data inside its cache, there is no guarantee that it will update the same data address into the shared memory, so other cores might read that address and get an old value [48]. This is known as *cache incoherency*. There are a number of protocols used to tackle with this issue, many based on the "Invalidate" pattern, that is when a given core updates the value of a piece of data, all other copies of that data in other cores caches are marked as "non valid". This solution reduces the buses data traffic, and is enhanced with the usage of MESI protocol, the one upon which TileLink developed its own. MESI is an acronym deriving from the four states into which a given core cache might reside.

- **M***odified* : cache line present only in current cache, and it differs from the value stored into main shared memory, so it is a so called **dirty** value. The value will have to be stored back into the main memory before or then, and the cache line originally holding it will be labeled as *Exclusive* once that happens.

- **E***xclusive* : cache line present only in current cache, the stored value matches that hold by shared main memory, thus it is a so called *clean* value. The state of the cache line may switch to *Modified* is the core holding that cache modifies the line, or to *Shared* if it detects that any other core requests that data.

- **S***hared* : cache line has a *clean* value and might be actually stored in other cores caches as well, so data can be also retrieved elsewhere, and matches the value stored in main shared memory. The line may switch its state to *Invalid*, once the same data is modified in any other core.

- **I***nvalid* : line value not valid anymore, as some other core modified the value elsewhere.

A state diagram illustrating the MESI states is depicted in Figure 2.17 to give a clearer overview.

2.4.2 Tools & Toolchains

The frameworks consists of tools and toolchains [25] that all together make it possible to customize the System on Chip you may want to build. In the following I'll list all of them, giving a deeper description for those that are actually deeply involved in my thesis. Please note that only RISCV-Tools and ESP-Tools refer to toolchains, whereas the others are simpler tools.

- Chisel: Chisel is an hardware design language, which aims to ease hardware modules and interconnections description while giving the user powerful features other Hardware Description Languages (HDL) like Verilog do not offer.

- FIRRTL: Chisel teams up with another language, named Flexible Intermediate Representation for RTL (FIRRTL), which role consists in taking what has been produced by the Chisel compiler and optimize it (e.g. removing unused signals) before writing an equivalent yet more efficient verilog description of the user design.

- Barstools: a collection of common FIRRTL transformations used to manipulate a digital circuit without changing the generator source RTL.

- Dsptools: a Chisel library for writing custom signal processing hardware, as well as integrating custom signal processing hardware into an SoC (especially a Rocket-based SoC).

- Dromajo: a RV64GC emulator primarily used for co-simulation and was originally developed by Esperanto Technologies

- RISCV-Tools: a collection of software used to develop, compile and run programs based on RISC-V Instruction Set Architecture (ISA). It inclused a functional ISA simulator, named Spike, the berkeley boot loader (BBL) to have linux OS running on a chipyard generated SoC, and other amenities. In this work it is used to have the RISC-V Front End Server (FESVR) running on the chosen software emulator to power up the SoC and feed it with proper commands.

- ESP-Tools: a fork of riscv-tools, designed to work with the Hwacha non-standard RISC-V extension. This fork can also be used as an example demonstrating how to add additional RoCC accelerators to the ISA-level simulation (Spike) and the higher-level software toolchain (GNU binutils, riscv-opcodes, etc.)

2.4.3 Simulators

The chipyard environment can take advantage of three different choices when it comes to simulators.

Verilator

Verilator is a cycle-accurate verilog and systemverilog compiler, not really a simulator. It is a software package that can convert all synthesizable, and even some behavioural, verilog and systemverilog constructs into either C++ or SystemC based models. Once the model is ready, a user written wrapper must be written; it should include a main function, instantiate the top level "verilated" model and describe the operations that should be taken during simulation. With the wrapper ready, a C++ compiler such as gcc can be used to create the simulation executable, according to a series of parameters, enabling for Value Change Dump (VCD) waveform traces to be produced at user request, similarly to what one would pretend from a commercial hardware language simulator. Finally note that this software is open source, so no fees are due to use it. Moreover the authors state that is often outperforms some commercial competitors.

VCS

VCS is a functional simulator made by Synopsys ®. Provided that the user has a valid product license, chipyard allows them to have wrappers to build VCS based simulators from the given Scala and Chisel files, together will all features that are mandatory for such tools, including faster compilation times and VCD trace support.

FireSim

FireSIM is an open source hardware simulation platform that runs on EC2 F1 FPGAs hosted on the Amazon Web Services (AWS). Its primary target is allowing anyone to evaluate the performances of an hardware design at FPGA specific speeds, among other possibilities, such as simulating datacenters and having profiling tools.

2.5 Parallel Ultra Low Platform (PULP)

PULP is an acronym for Parallel Ultra Low Power, and represents a family of processors being developed as joint effort by ETH Zurich and Università di Bologna, which aims at supporting a wide variety of IoT (Internet of Things) related applications, striving to achieve performance comparable to ARM Cortex-M family but with lower power cost. The idea is to exploit parallel calculus as much as possible, organizing multiple smaller cores as clusters, up to a maximum of 16 cores per cluster. A cluster contains a shared instruction cache and a shared scratchpad memory, which provides recent data. The project started from Open RISC, then has gradually moved to RISC-V ISA, adding instructions that serve this scope and possibly accelerators, working with a "near threshold" technique, which aims at using the core with the lowest possible operating voltage which still meet the performance they expect. Many designs were taped out [60]

- Main : PULPv1,PULPv2,PULPv3,PULPv4
- PULP Experiments : Artemis, Hecate, Imperio, Fulmine
- RISC-V Based: Honey Bunny
- Mixed-signal : VivoSoC,EdgeSoc
- Approximate computing : Diego, Manny, Sid

Since the design space they are exploring is pretty huge, the open source list started with a simple yet effective core named PULPino.

2.5.1 PULPino

Small microcontroller with no caches, no memory hierarchy and no DMA. All IPs from PULP projects, working on FPGA. As one can see from Figure 2.18, the core is connected to two separated single port instruction and data RAMs, each one accessible in a single clock cycle, with no wait states, which all contribute to provide small power consumption. An AXI4 interconnect is there to provide connection to the two RAMs and between other peripherals, through an APB bridge, which allows for high flexibility as long as one uses components suited for

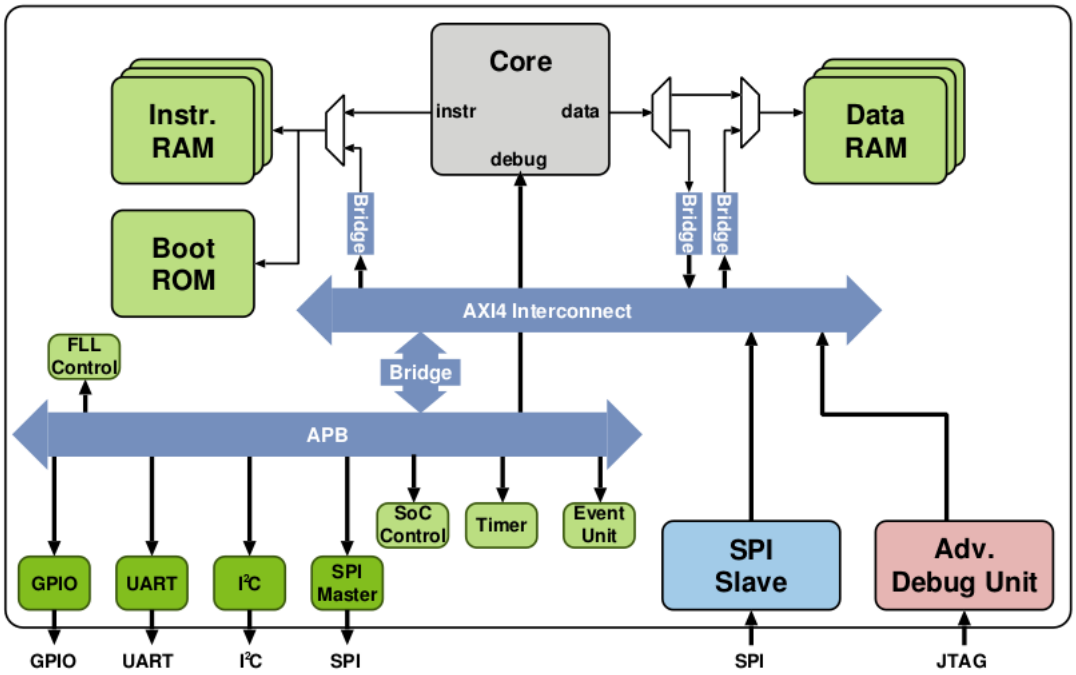

Figure 2.18: PULPino Architecture

it or AXI. The peripherals on the bottom left part, namely GPIOs, UART, I^2C and SPI master are used to communicate with external devices and are fine grained clock gated, meaning that they can be shut down whenever not needed. The core is in a low power state whenever there is nothing to be done, and a simple event unit, which waits for an interrupt or an event caused by a peripheral, inhibits the clock gating on the core and wakes it up. Furthermore a SPI slave is provided to allow external devices to access the entire memory map of pulpino and an advanced debug unit, accessible via JTAG, allows for debugging. Finally a BOOT ROM has been integrated in order to allow user to use PULPino as standalone system, simply loading a bootloader into the core through an external SPI flash memory. Moving from OPEN RISC to RISC-V was justified by the need for easily extensible ISA, the possibility of having less and eventually compressed instructions, which overall helps in power consumption reduction. A couple of extensions which are worth mentioning are

1. Hardware Loops : since a loop is typically a performance impact for a simple core, mostly because of the branch evaluation at the end of every iteration, hardware loops allow to set up the number of overall iterations before the loop starts, removing that cost.

2. Post Incrementing Load and Store : load and store are often part of patterns where the target addresses are repeatedly incremented, but this is done

through a separate instruction. It can be avoided by specifying that the address has to be incremented, so that in a single execution cycle memory gets updated or accessed for a read and the target address is updated.

2.5.2 PULPissimo

PULPissimo is the most recent and advanced single core platform of the PULP project. Although it hosts only one core, it is used to as the main controller for applications in which there are multiple cores, thereby providing

- Autonomous I/O Subsystems, called uDMA (micro Direct Memory Access) [43]. Once the main core configures it, the uDMA takes care of the peripherals transfers.
- new memory subsystem devised to improve both performance and power consumption
- support for Hardware Processing Engines (HWPEs), that is custom accelerators/coprocessors
- new interrupt controller (INTC): up to 32 requests from peripherals, with possibility of masking events, have pending interrupts, and much more.
- new peripherals (QuadSPI, Camera Interface, I^2C, I^2S, JTAG, GPIOs, BootROM)
- new Software Development Kit (SDK): contains tools and runtime support for all PULP microcontrollers, with procedurs for setting up cores and peripherals, so that application developers can exploit their full potential.

RI5CY

It is an in order single instruction core, made up of 4 pipelined stages, with full support for RV32I, RV32C and RV32M RISC-V instructions. It can be configured to support single precision floating point instructions, that is RV32F extension. Moreover it provides a number of specific features

- Interrupts
- Events, allowing the core to sleep and wait for an event (as seen for PULPino)
- Exceptions
- performance counters, telling the number of compressed instructions and so on
- Debug through software breakpoints and access to all registers

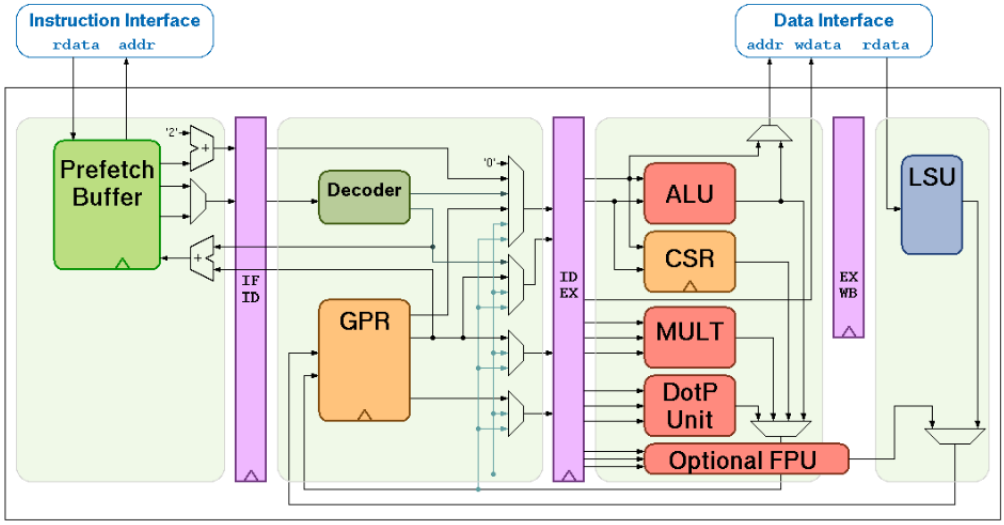

Figure 2.19: RI5CY Datapath Pipeline Stages

- Hardware loops and post incrementing load/store, as seen for PULPino

Further details provided in [18].

ZERO-RISCY

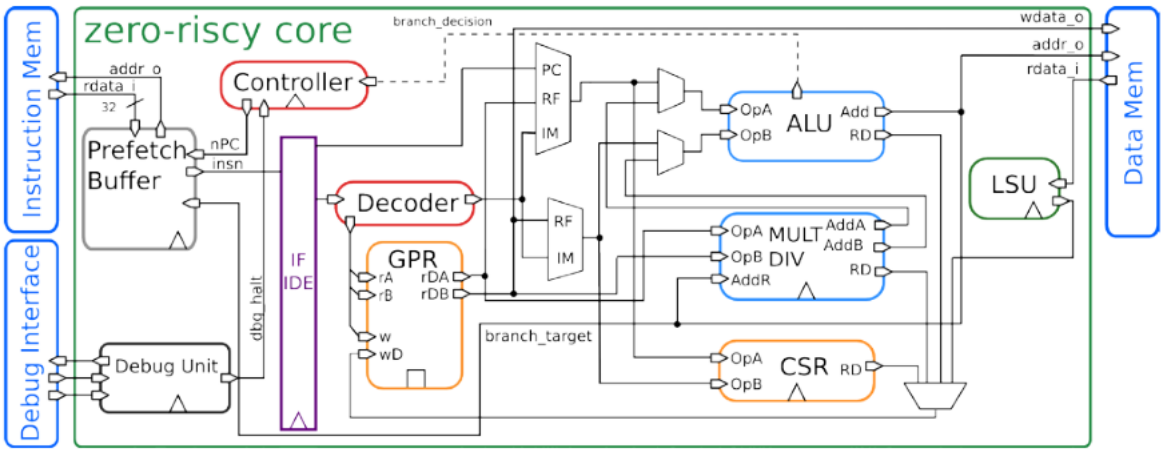

Figure 2.20: ZERO RISCY Datapath

A block diagram is shown in Figure 2.20. Derived from RI5CY, it is a 2 stage in order core, designed to be small, efficient, and configurable through 2 parameters to support one or multiple of the following extensions

- RV32I: base integer ISA, must be always implemented.

- RV32E: reduced version of RV32I, targeting embedded systems.
- RV32C: 16 bit instructions.
- RV32M: integer multiply and divide instructions.

It has been designed to target application domains for which area and power are strongly constrained. Further details are available at [13]. RI5CY has recently being taken over by lowRISC foundation, which provides full documentation at [17].

Chapter 3

Methods

This chapter provides a deep dive into the thesis workflow, giving details on each and every step needed to have a working simulation environment and a properly set up architecture consisting of ODIN and RocketChip interfaced by means of SPI. Chipyard has been selected as RISC-V SoC for the thesis, because it felt easier to use and offered a very high degree of configuration choices. First, ODIN and Chipyard will be properly downloaded and configured, then the accelerator will be integrated inside the Chipyard compilation flow so that it can be instantiated in the final architecture. Finally, the configuration procedure of ODIN will be explained and applied to a specific neural network. The steps of the workflow are summarized in the diagram of Figure 3.1. All steps up to ODIN Configuration Setup will be outlined in this chapter, whereas RTL Simulation and Synthesis will be carried out in Chapter 4.

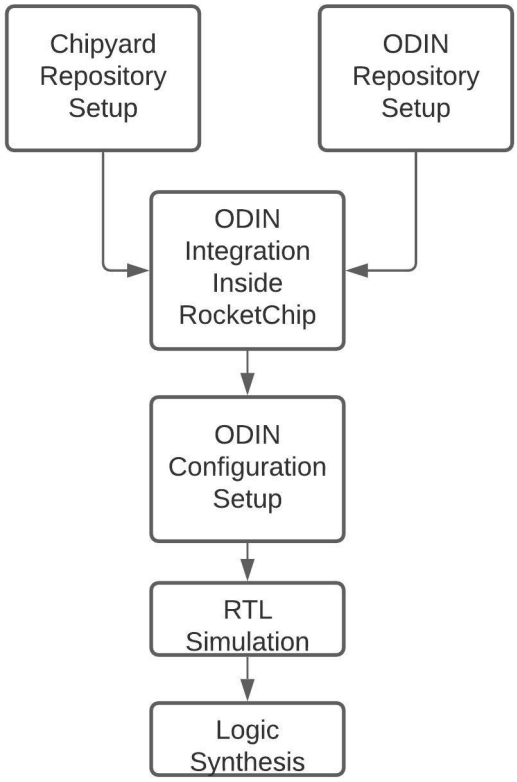

Figure 3.1: Thesis Workflow

3.1 Setting up ODIN and Chipyard Environments

The ODIN files and documentation are publicly available at [20]. The source files of the architecture, listed on the right part of Figure 2.6, are located inside *src* folder. In order to use ODIN, one should create a folder and type *git init*, followed by **git remote add origin <odin_url>**, where odin_url is the one at [20]. Finally execute command *git pull origin master* to let the Git system download everything that's on the master branch of the repository. No further work is needed. The files of Chipyard project are available at [45], whereas the documentation is hosted at [44]. Considering that the thesis workflow has been tested on a Linux distribution, most of the software dependencies listed in that documentation are already satisfied, so that section may be skipped, and can go on with setting the chipyard installation. First, one should create a folder and type **git init**, followed by **git remote add origin <chipyard_url>**, where chipyard_url is again [45]. Finally execute command **git pull origin master** to let the Git system download everything that's on the master branch of the repository. In order to make chipyard fully operational,

one should then execute the **init-submodules-no-riscv-tools.sh** script,which will initialize all project submodules (e.g. rocketchip, sifive-blocks etc), then the **build-toolchains.sh** one; the latter will take a while, as it builds all required toolchains that are needed to compile and run RISC-V programs on any architecture you may compose with Chipyard. To speed up the process, it is possible to type **export MAKEFLAGS=-j<cores>**, where **cores** is an integer indicating the number of cores of the CPU on which the host system is running, before executing the toolchain building script.

3.2 ODIN Integration Inside Chipyard

Companies and researchers are used to exploit designs which someone else build and provides, either freely or by paying for a license, in order to reach their projects goal, often within a strict deadline. Members of Berkeley Architecture Research (BAR) group are aware of this necessary routine, so Chipyard offers the possibility of integrating those design, provided they are written in **Verilog** or SystemVerilog, or describe their architectures using the Chisel language, if necessary. Given a complete, syntactically correct and synthesizable design, the first step to take consists in copying all .sv or .v source files of ODIN coprocessor inside **generators/chipyard/src/main/resources/vsrc**. The top level entity, that is **ODIN.v** in this specific case, should be renamed to **ODINBlackBox.v** before proceeding.
The next move is to define a Scala file named **ODIN.scala** in any folder inside **generators/chipyard/src/main/scala**; In this work, it has been placed inside the **example** folder, as it was done for the Greatest Common Divider (GCD) example of Verilog design integration illustrated in the chipyard documentation. The file **ODIN.scala** serves the purpose of declaring a Chisel specified ODIN blackbox module, that acts as a wrapper for the underlying verilog design. Every chisel blackbox consists of

- an **io** field, which bundles all input and output signal of ODIN top level entity
- a constructor parameter
- a listing of all verilog resources that must be loaded to make the design work

Chisel offers the possibility of having parameterized blocks, and this is absolutely fundamental when one or more blocks have to be adaptable to different domains of application.

```
1 case class ODINParams(
2   address: BigInt = 0x2000,
3   N: Int = 256,
4   M: Int = 8)
```

Listing 3.1: ODIN Parameters

Let's analyze the code contained in **ODIN.scala**. In Listing 3.1, the parameters to build ODIN are declared. In this simple case, there are only three parameters; N and M indicate the number of neurons and number of bits needed to represent the neuron address, respectively, whereas address is the address that is to be associated to ODIN in the memory map of the SoC.

```
case object ODINKey extends Field[Option[ODINParams]](None)
```

Listing 3.2: ODINKey

In Listing 3.2, a Key is declared as Option type; a key is used by Chisel to specify which parameters are needed to customize the functionalities of a given module. They are to be declared as **Option**, with default value being **None**, meaning that in case the user doesn't explicitly call ODINKey with parameters, the default values specified in ODINParams are to be selected.

```
class ODINIO(val N: Int, val M: Int) extends Bundle {
  val CLK = Input(Clock())
  val RST = Input(Bool())
  val SCK = Input(Clock())
  val MOSI = Input(UInt(1.W))
  val MISO = Output(UInt(1.W))
  val AERIN_ADDR = Input(UInt((2*M+1).W))
  val AERIN_REQ = Input(UInt(1.W))
  val AERIN_ACK = Output(UInt(1.W))
  val AEROUT_ADDR = Output(UInt(M.W))
  val AEROUT_REQ = Output(UInt(1.W))
  val AEROUT_ACK = Input(UInt(1.W))
}
```

Listing 3.3: ODINIO

In Listing 3.3, the top level signals of ODIN are declared, with one bit signals being of Bool type and multi bit ones being declared as unsigned integers, which width is parameterized according to the value of M. The ".W" expression serves the purpose of converting the Scala integer value inside the parenthesis to Chisel Width type.

```
trait HasODINIO extends BaseModule {
  val N: Int
  val M: Int
  val io = IO(new ODINIO(N,M))
}
```

Listing 3.4: ODINIO

In Listing 3.4 the IO bundle required by the blackbox integration is declared. Notice that top level parameters are indicated as well, as they are needed for the input/output definition.

```
class ODINBlackBox(val N: Int, val M: Int) extends BlackBox(Map("N" -> IntParam(N), "
    M" -> IntParam(M))) with HasBlackBoxResource
  with HasODINIO
```

```
{
  addResource("/vsrc/ODINBlackBox.v")
  addResource("/vsrc/aer_out.v")
  addResource("/vsrc/controller.v")
  addResource("/vsrc/fifo.v")
  addResource("/vsrc/izh_calcium.v")
  addResource("/vsrc/izh_effective_threshold.v")
  addResource("/vsrc/izh_input_accumulator.v")
  addResource("/vsrc/izh_neuron_state.v")
  addResource("/vsrc/izh_neuron.v")
  addResource("/vsrc/izh_stimulation_strength.v")
  addResource("/vsrc/lif_calcium.v")
  addResource("/vsrc/lif_neuron_state.v")
  addResource("/vsrc/lif_neuron.v")
  addResource("/vsrc/neuron_core.v")
  addResource("/vsrc/sdsp_update.v")
  addResource("/vsrc/spi_slave.v")
  addResource("/vsrc/synaptic_core.v")
  addResource("/vsrc/scheduler.v")
}
```

Listing 3.5: ODINBlackBox

In Listing 3.5, the blackbox class is declared. Notice that the parameters are mapped as integers, and that traits HasBlackBoxResource and HasODINIO are appended to the class declaration. The body of the class just lists all the verilog files needed for the design description.

```
trait ODINModule extends HasRegMap{

 val io: ODINTopIO
  implicit val p: Parameters
  def params: ODINParams
  val clock: Clock
  val reset: Reset

  val impl = Module(new ODINBlackBox(params.N,params.M))

  val aerin_ack = RegInit(0.U(1.W))
  val aerout_ack = RegInit(0.U(1.W))
  val aerin_req = RegInit(0.U(1.W))
  val aerout_req = RegInit(0.U(1.W))
  val aerin_addr = RegInit(0.U((2*params.M+1).W))
  val aerout_addr = RegInit(0.U((params.M).W))
  impl.io.CLK := clock
  impl.io.RST := reset.asBool
  impl.io.SCK := clock

  impl.io.AEROUT_ACK := aerout_ack
  aerout_addr := impl.io.AEROUT_ADDR
  aerout_req := impl.io.AEROUT_REQ
```

```
24  impl.io.AERIN_REQ := aerin_req
25  impl.io.AERIN_ADDR := aerin_addr
26  aerin_ack := impl.io.AERIN_ACK
27
28
29  io.odin_busy := impl.io.RST
30
31  regmap(
32    0x00 -> Seq(
33      RegField.r(1, aerout_req)), // a read-only register capturing aerout_req
34    0x04 -> Seq(
35      RegField.r(params.M, aerout_addr)),
36  0x08 -> Seq(
37    RegField.w(1, aerout_ack)),
38  0x0C -> Seq(
39    RegField.w(1, aerin_req)),
40  0x10 -> Seq(
41    RegField.w(2*params.M+1, aerin_addr)),
42  0x14 -> Seq(
43    RegField.r(1, aerin_ack))) // read-only, aerout_addr
44
45 }
```

Listing 3.6: ODIN Interconnections

In Listing 3.6 a trait is needed to describe the interconnections for ODIN top level signals. Let's analyze this trait. An *impl* val is declared to instantiate the ODIN module, then all required input and output signals are declared as well, making sure that those related to input AER and output AER are declared as registers. Every *impl* signal is configured so that it appears on the left side of the := assignment operator if the signal is an input to ODIN, whereas it appears to the left side of the operator if it is a signal coming out from ODIN module. The AER specific signals must be declared as registers because ODIN will be integrated as a MMIO Peripheral. A MMIO Peripheral is a device which makes it possible for the processor to communicate with it through memory mapped registers, exploiting TileLink interconnects.
Once ODIN signals are properly interconnected, it is necessary to inform Chipyard toolchain on how those registers are to be inserted into the SoC memory map. In particular, *aerout_req* will be a read-only register, 1 bit wide, placed at offset 0x00, with respect to base address of ODIN module, which is indicated by *address* parameter in ODINParams class, whereas *aerout_ack* will be a write-only register, 1 bit wide, placed at offset 0x0C. Similar considerations apply for other ODIN signals, as illustrated into Listing 3.6 at lines 31-43.

```
1 class ODINTL(params: ODINParams, beatBytes: Int)(implicit p: Parameters)
2   extends TLRegisterRouter(
3     params.address, "odin", Seq("ucvlouvain,odin"),
4     beatBytes = beatBytes)(
```

```
    new TLRegBundle(params, _) with ODINTopIO)(
    new TLRegModule(params, _, _) with ODINModule)
```

Listing 3.7: ODIN TileLink

In Listing 3.7 a class named ODINTL is declared and filled in order to properly set ODIN to take advantage of TileLink interconnection. This is done by having the class extend the TLRegisterRouter; the first argument **(params.address, "gcd", Seq("ucbbar,gcd"), beatBytes = beatBytes)** determines in which place of the global memory map ODIN will be placed and what information will be provided in the device tree, that is a data structure used by an operating system kernel to detect and manage attached peripherals; the second one indicates which signals will be seen by TileLink and the last one provides the proper constructor to have ODIN properly connected to TileLink bus. Now it is time to make Chipyard recognize ODIN when needed.

```
trait CanHavePeripheryODIN { this: BaseSubsystem =>
  private val portName = "odin"

  // Only build if we are using the TL (nonAXI4) version
  val odin = p(ODINKey) match {
    case Some(params) => {
      val odin = LazyModule(new ODINTL(params, pbus.beatBytes)(p))
      pbus.toVariableWidthSlave(Some(portName)) { odin.node }
      Some(odin)
    }
    case None => None
    }

  }
```

Listing 3.8: ODIN Periphery

In Listing 3.8 a LazyModule trait is shown, containing indications on what must be initialized before all hardware is elaborated and instantiated. In particular, since this is a memory mapped peripheral, it is sufficient to have ODIN TileLink specific node connected to the MMIO crossbar. Every RegisterRouter has a tilelink node named "node", that can be connected to the main core peripheral bus, in order to add the address to the memory map and all device tree info that have been provided in Listing 3.7.

```
trait CanHavePeripheryODINModuleImp extends LazyModuleImp {
  val outer: CanHavePeripheryODIN
  val odin_busy = outer.odin match {
    case Some(odin) => {
      val busy = IO(Output(Bool()))
      busy := odin.module.io.odin_busy
      Some(busy)
    }
    case None => None
```

```
  }
}
```

Listing 3.9: ODIN Top level implementation

Listing 3.9 shows the top level trait needed to have top level ports visible. A few things still need to be addressed before compiling the chipyard project and have the synthesizable verilog ready.

```
class WithODIN extends Config((site, here, up) => {
  case ODINKey => Some(ODINParams(N = 256, M = 8))
})
```

Listing 3.10: ODIN Config Fragment

Listing 3.10 describes the config fragment needed to put ODIN in whatever configuration class. A more configurable version would be

```
class WithODIN(N: Int, M: Int) extends Config((site, here, up) => {
  case ODINKey => Some(ODINParams(N = N, M = M))
})
```

but the one shown in 3.10 is modified in order to have ODIN instantiated with default values suggested by its author, as the note on ODIN documentation states that the crossbar scales according to N and M, but other ODIN modules do not automatically scale and proper modifications are needed if one changes either M,N or both.

```
class DigitalTop(implicit p: Parameters) extends System
  with testchipip.CanHaveTraceIO // Enables optionally adding trace IO
  with testchipip.CanHaveBackingScratchpad // Enables optionally adding a backing
     scratchpad
  with testchipip.CanHavePeripheryBlockDevice // Enables optionally adding the block
     device
  with testchipip.CanHavePeripherySerial // Enables optionally adding the TSI serial-
     adapter and port
  with sifive.blocks.devices.uart.HasPeripheryUART // Enables optionally adding the sifive
     UART
  with sifive.blocks.devices.gpio.HasPeripheryGPIO // Enables optionally adding the sifive
     GPIOs
  with sifive.blocks.devices.spi.HasPeripherySPIFlash // Enables optionally adding the sifive
     SPI flash controller
  with icenet.CanHavePeripheryIceNIC // Enables optionally adding the IceNIC for FireSim
  with chipyard.example.CanHavePeripheryInitZero // Enables optionally adding the initzero
      example widget
  with chipyard.example.CanHavePeripheryGCD // Enables optionally adding the GCD
     example widget
  with chipyard.example.CanHavePeripheryODIN // Enables optionally adding the ODIN
     example widget
  with chipyard.example.CanHavePeripheryStreamingFIR // Enables optionally adding the
     DSPTools FIR example widget
  with chipyard.example.CanHavePeripheryStreamingPassthrough // Enables optionally
     adding the DSPTools streaming-passthrough example widget
```

```
15  with nvidia.blocks.dla.CanHavePeripheryNVDLA // Enables optionally having an NVDLA
16 {
17  override lazy val module = new DigitalTopModule(this)
18 }
```

Listing 3.11: Chipyard DigitalTop

There are still two steps to take before ODIN can be utilized in Chipyard framework. The first one consists in locating the file DigitalTop.scala in **generators/chipyard/src/main/scala** and modify the class DigitalTop as shown in Listing 3.11, thus adding the string *with chipyard.example.CanHavePeripheryODIN*.

```
1 class DigitalTopModule[+L <: DigitalTop](l: L) extends SystemModule(l)
2  with testchipip.CanHaveTraceIOModuleImp
3  with testchipip.CanHavePeripheryBlockDeviceModuleImp
4  with testchipip.CanHavePeripherySerialModuleImp
5  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
6  with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
7  with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
8  with icenet.CanHavePeripheryIceNICModuleImp
9  with chipyard.example.CanHavePeripheryGCDModuleImp
10  with chipyard.example.CanHavePeripheryODINModuleImp
11  with freechips.rocketchip.util.DontTouch
```

Listing 3.12: Chipyard DigitalTop part 2

The last one consists in modifying the very same file, this time adding the string **with chipyard.example.CanHavePeripheryODINModuleImp** to the class DigitalTopModule, as depicted in Listing 3.12.

```
1 class ODINRocketConfig extends Config(
2  new chipyard.iobinders.WithUARTAdapter ++                    // display UART with a
     SimUARTAdapter
3  new chipyard.iobinders.WithTieOffInterrupts ++               // tie off top-level
     interrupts
4  new chipyard.iobinders.WithBlackBoxSimMem ++                 // drive the master AXI4
      memory with a blackbox DRAMSim model
5 //  new chipyard.iobinders.WithSimAXIMem ++
6  new chipyard.iobinders.WithTiedOffDebug ++                   // tie off debug (since we
     are using SimSerial for testing)
7  //new chipyard.iobinders.WithSimDebug ++
8  new chipyard.iobinders.WithSimSerial ++                      // drive TSI with SimSerial for
      testing
9  new testchipip.WithTSI ++                                    // use testchipip serial offchip link
10  new chipyard.example.WithODIN ++
11  new chipyard.config.WithBootROM ++                          // use default bootrom
12  new chipyard.config.WithUART ++                             // add a UART
13  new chipyard.config.WithL2TLBs(1024) ++                     // use L2 TLBs
14
15  new freechips.rocketchip.subsystem.WithNoMMIOPort ++        // no top-level MMIO
     master port (overrides default set in rocketchip)
```

```
16  new freechips.rocketchip.subsystem.WithNoSlavePort ++          // no top-level MMIO
        slave port (overrides default set in rocketchip)
17 /* new freechips.rocketchip.subsystem.WithInclusiveCache ++      // use Sifive L2 cache*/
18  new freechips.rocketchip.subsystem.WithNExtTopInterrupts(0) ++ // no external
        interrupts
19  new freechips.rocketchip.subsystem.WithNBigCores(1) ++         // single rocket-core
20  new freechips.rocketchip.subsystem.WithCoherentBusTopology ++  // hierarchical buses
        including mbus+l2
21  new freechips.rocketchip.system.BaseConfig)                    // "base" rocketchip system
```

Listing 3.13: Chipyard RocketConfigs

Now it is time to create a new configuration inside file RocketConfigs.scala located in **generators/chipyard/main/scala/config**, simply by having a new class, that extends the Config base class, with all components one wants to put. Listing 3.13 shows such modification. As the Chipyard documentation suggests, the design can be compiled by executing the *make CONFIG=SmallSPIFlashODINRocketConfig BINARY=../../tests/spiflashread.riscv verilog -j4* command on a Terminal, making sure that the working directory is *sims/verilator*, located inside the Chipyard folder that has been created in 3.1. Once compilation starts, the software will determine whether they are compatible with each other or not. As soon as the process ends, the architecture verilog files are available in **sims/verilator/generated-src/chipyard.TestHarness.SmallSPIFlashODINRocketConfig** folder.

```
1  module ODINTL( // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig.fir@
       284084.2]
2  input         clock, // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig.fir@
       284085.4]
3  input         reset, // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig.fir@
       284086.4]
4  output        auto_in_a_ready, // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig
       .fir@284087.4]
5  input         auto_in_a_valid, // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig.
       fir@284087.4]
6  input  [2:0]  auto_in_a_bits_opcode, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
7  input  [2:0]  auto_in_a_bits_param, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
8  input  [1:0]  auto_in_a_bits_size, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
9  input  [6:0]  auto_in_a_bits_source, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
10 input  [13:0] auto_in_a_bits_address, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
11 input  [7:0]  auto_in_a_bits_mask, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
12 input         auto_in_a_bits_corrupt, // @[:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284087.4]
13 input         auto_in_d_ready, // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig.
       fir@284087.4]
```

```
14  input      spi_slave_sck,
15  input      spi_slave_mosi,
16  output         auto_in_d_valid, // @[:chipyard.TestHarness.SmallSPIFlashODINRocketConfig.
        fir@284087.4]
17  output [2:0]   auto_in_d_bits_opcode, // @[:chipyard.TestHarness.
        SmallSPIFlashODINRocketConfig.fir@284087.4]
18  output [1:0]   auto_in_d_bits_size, // @[:chipyard.TestHarness.
        SmallSPIFlashODINRocketConfig.fir@284087.4]
19  output [6:0]   auto_in_d_bits_source, // @[:chipyard.TestHarness.
        SmallSPIFlashODINRocketConfig.fir@284087.4]
20  output [63:0] auto_in_d_bits_data // @[:chipyard.TestHarness.
        SmallSPIFlashODINRocketConfig.fir@284087.4]
21 );
```

Listing 3.14: ODINTL changes

```
1 assign impl_SCK = spi_slave_sck; // @[ODIN.scala 115:15:chipyard.TestHarness.
      SmallSPIFlashODINRocketConfig.fir@284141.4]
2 assign impl_MOSI = spi_slave_mosi;
```

Listing 3.15: ODINTL SCK and MOSI assignments

```
1  ODINTL odin ( // @[ODIN.scala 128:30:chipyard.TestHarness.
       SmallSPIFlashODINRocketConfig.fir@284671.4]
2    .clock(odin_clock),
3    .reset(odin_reset),
4    .auto_in_a_ready(odin_auto_in_a_ready),
5    .auto_in_a_valid(odin_auto_in_a_valid),
6    .auto_in_a_bits_opcode(odin_auto_in_a_bits_opcode),
7    .auto_in_a_bits_param(odin_auto_in_a_bits_param),
8    .auto_in_a_bits_size(odin_auto_in_a_bits_size),
9    .auto_in_a_bits_source(odin_auto_in_a_bits_source),
10   .auto_in_a_bits_address(odin_auto_in_a_bits_address),
11   .auto_in_a_bits_mask(odin_auto_in_a_bits_mask),
12   .auto_in_a_bits_corrupt(odin_auto_in_a_bits_corrupt),
13   .auto_in_d_ready(odin_auto_in_d_ready),
14   .auto_in_d_valid(odin_auto_in_d_valid),
15   .auto_in_d_bits_opcode(odin_auto_in_d_bits_opcode),
16   .auto_in_d_bits_size(odin_auto_in_d_bits_size),
17   .auto_in_d_bits_source(odin_auto_in_d_bits_source),
18  .auto_in_d_bits_data(odin_auto_in_d_bits_data),
19   .spi_slave_sck(qspi_0_0_sck),
20   .spi_slave_mosi(qspi_0_0_dq_1_i)
21
22  );
```

Listing 3.16: ODINTL instantiated block changes

There are some of modifications that should be performed on the file

```
1 sims/verilator/generated-src/chipyard.TestHarness.SmallSPIFlashODINRocketConfig/
      chipyard.TestHarness.SmallSPIFLashODINRocketConfig.top.v
```

Indeed, one should modify the module *ODINTL* as illustrated in Listing 3.14, then look for *impl_SCK* and *impl_MOSI*, which lie inside the ODINTL module, and connect them according to Listing 3.15. Yet another modification should be applied to the instantiated ODIN module; look for "ODINTL odin" inside the file, and modify it according to Listing 3.16. These changes are necessary to properly interface the SPIFlash Master with ODIN. Finally, run *make CONFIG=SmallSPIFlashODINRocketConfig BINARY=../../tests/spiflashread.riscv run-binary-debug -j4* again on the terminal, still with sims/verilator as working directory, in order to recompile the whole architecture and produce the simulation executable. You'll notice that an error has been detected, as shown in the following code snippet

```
[0] \% Error: plusarg\_file\_mem.sv:50: Assertion failed in TOP.TestHarness.spi\_mem\_0.
    memory
```

This error arises due to the fact that the simulator doesn't know where to get the file that SPIFlash module should read to feed ODIN with configuration patterns. To fix the error, one should issue the following command

```
/home/andrea/Documents/SNN_Gianvito/Papers/RocketChip/chipyard/sims/verilator/
    simulator-chipyard-SmallSPIFlashODINRocketConfig-debug +permissive +dramsim
    +spiflash0=/home/andrea/Documents/SNN_Gianvito/Papers/RocketChip/chipyard/
    tests/spiflash_odin.img +verbose -v spiflashread.chipyard.TestHarness.
    SmallSPIFlashODINRocketConfig.vcd +permissive-off ../../tests/spiflashread.riscv </
    dev/null 2> >(spike-dasm > spiflashread.chipyard.TestHarness.
    SmallSPIFlashODINRocketConfig.out) | tee spiflashread.chipyard.TestHarness.
    SmallSPIFlashODINRocketConfig.log
```

Listing 3.17: Command to run ODIN + ROCKET simulation.

You'll notice this is almost the same command issued before the error occurred. Indeed the binary file to be read by SPIFlash master module has been specified with a path given through variable *spiflash0*, as reported in Listing 3.17. If you launch the command shown in that code portion, you'll launch the software RTL simulation and, after a given amount of time, the process will end, providing you with three files

- *spiflashread.chipyard.TestHarness.SmallSPIFLASHODINRocketConfig.out* : this is a long listing of all RISC-V assembly instructions being executed, starting from the base address of the bootrom, up to the end of spiflashread program. An excerpt is shown in Listing 3.18. That string tells the instruction that is being executed (i.e. auipc, 0x0), that it refers to the 19th simulation cycle and core hart #0 (i.e. C0), that r10 is going to be filled with a new value and that, a few cycles before, when the instruction was located into decode stage, it made the processor read two registers, in this case being both r0. Indeed each string in this file refers to the write-back stage of the 5 stage pipeline of rocketchip, that is the last stage of execution of a given instruction, during which the result is written back into the register file, if any.

```
1 C0:         19 [1] pc=[0000000000010040] W[r10=0000000000010040][1] R[r 0=
    0000000000000000] R[r 0=0000000000000000] inst=[00000517] auipc   a0, 0x0
```

Listing 3.18: First line of .out file given by ODIN + ROCKET simulation, running spiflashread.riscv program.

- *spiflashread.chipyard.TestHarness.SmallSPIFLASHODINRocketConfig.log* : this is a log of the entire simulation, listing output of the modules involved in the simulation. In this work it just lists debug information coming from the program being run for simulation, that is *spiflashread.riscv*, consisting in a few printf functions being called in specific points, so to inform the user about the status of the simulation.

- *spiflashread.chipyard.TestHarness.SmallSPIFLASHODINRocketConfig.vcd* : this is the Value Change Dump (VCD) file, that is a dump of the waveforms involved in the simulation, which can be read by any VCD reader, such as the open source GTKWave.

3.3 ODIN Parameters Definition

As one can imagine, manually configuring ODIN can be challenging, both because it is quite a complex piece of hardware and it is configurable through standard Serial Peripheral Interface (SPI) protocol, which is not complex, but can lead to a long and error prone configuration phase, due to its characteristics. To ease this phase, a C program has been developed, allowing for simple yet effective configuration of ODIN SPI internal registers, and loading neuron and synapse Static Random Access Memories (SRAMs). The program is called *odin_configurator* and can be executed by simply calling it by a terminal on Linux. Once executed, a menu will be provided to the user, as shown in Figure 3.5. The proposed choices are

0. Exit. Simply terminates the program.

1. Set SPI Configuration Registers. Once selected, a list of possible registers will be shown, as depicted in Figure 3.6. In example of Figure 3.7, the value 4 is given for the least significant bits of SPI_SYN_SIGN registers, which determines whether synapses of a given neuron are inhibitory or excitatory, according to Table 2.3.

2. Add Synapse. Once selected, the routine asks for presynaptic and postsynaptic neuron numbers, that is the neurons one wants to connect through the synapse that is going to be set up. Then, it asks whether to set up the mapping_table_bit and which weight value to assign to the established synapse. This process is shown in Figure 3.3 and Figure 3.4.

```
Type a number in range [0,255] to specify the neuron you want to customize:
5
Type 1 if you want a LIF neuron, 0 for custom Izhikevich: 1
Type a number in range [0,127] for  defining the leakage mechanism strenght: 0
Type 1 if you want to enable the leakage mechanism, 0 otherwise: 0
Type a number in range [0,255] to define the membrane firing threshold: 1
Type 1 if you want to enable the calcium mechanism, 0 otherwise: 0
Type a number in range [0,255] to define the SDSP threshold on membrane potential: 0
Type a number in range [0,7] to define the first SDSP threshold on calcium variable: 0
Type a number in range [0,7] to define the second SDSP threshold on calcium variable: 0
Type a number in range [0,7] to define the third SDSP threshold on calcium variable: 0
Type a number in range [0,31] to define the calcium variable leakage time constant: 0
Type a number in range [0,255] to define the starting value of membrane potential: 0
Type a number in range [0,7] to define the starting value of calcium variable: 0
Type a number in range [0,31] to define the starting value of calcium leakage counter: 0
Type 1 if you want to disable the neuron, 0 otherwise: 0
Neuron configuration[0] : 01 (LEAK_STR[6:0] + LIF_IZH_SEL)
Neuron configuration[1] : 02 (THR[6:0] + LEAK_EN)
Neuron configuration[2] : 00 (THETAMEM[5:0] + CA_EN + THR[7])
Neuron configuration[3] : 00 (CA_THETA2[2:0] + CA_THETA1[2:0] + THETAMEM[7:6])
Neuron configuration[4] : 00 (CA_LEAK[4:0] + CA_THETA3[2:0])
Neuron configuration[5] : 00 (0x00)
Neuron configuration[6] : 00 (0x00)
Neuron configuration[7] : 00 (0x00)
Neuron configuration[8] : 00 (CORE[1:0] + 000000)
Neuron configuration[9] : 00 (CALCIUM[1:0] + CORE[7:2])
Neuron configuration[10] : 00 (00 + CA_LEAK_CNT[4:0] + CALCIUM[2])
Neuron configuration[11] : 00 (0x00)
Neuron configuration[12] : 00 (0x00)
Neuron configuration[13] : 00 (0x00)
Neuron configuration[14] : 00 (0x00)
Neuron configuration[15] : 00 (NEUR_DISABLE + 0000000)
little-endian
```

Figure 3.2: An example of neuron configuration using odin_configurator application

```
Type the presynaptic neuron number [0,255]: 0
Type the postsynaptic neuron number [0,255]: 1
Synaptic memory address: 000
Byte address : 00
Synapse address spi : 60000
Type 1 to set the mapping table bit, 0 otherwise: 1
Type a number in range [0,7] to set the synapse weight: 1
Synapse data: 00090
little-endian
```

Figure 3.3: An example of synapse configuration using odin_configurator application. Note that since the postsynaptic neuron number is odd, 4 bit configuration data is moved left by 4 positions.

3. Add Neuron. When this option is selected, all parameters listed in Figure 3.2 have to be tuned. First it ask for the neuron number one wants to customize, then all LIF specific parameters can be specified. Once determined, the program gives back the values that were specified, together with a brief indication on the parameters to which each byte refers.

```
Type the presynaptic neuron number [0,255]: 1
Type the postsynaptic neuron number [0,255]: 2
Synaptic memory address: 020
Byte address : 01
Synapse address spi : 62020
Type 1 to set the mapping table bit, 0 otherwise: 1
Type a number in range [0,7] to set the synapse weight: 1
Synapse data: 00009
little-endian
```

Figure 3.4: An example of synapse configuration using odin_configurator application.

```
1 - Set Configuration Register
2 - Add synapse
3 - Add neuron
4 - Synfire chain configuration
0 - Exit.
Type a number:
```

Figure 3.5: ODIN configurator menu.

```
0 - SPI_GATE_ACTIVITY
1 - SPI_OPEN_LOOP
2 - SPI_SYN_SIGN for neurons in range [15,0]
3 - SPI_SYN_SIGN for neurons in range [31,16]
4 - SPI_SYN_SIGN for neurons in range [47,32]
5 - SPI_SYN_SIGN for neuons in range [63,48]
6 - SPI_SYN_SIGN for neurons in range [79,64]
7 - SPI_SYN_SIGN for neuons in range [95,80]
8 - SPI_SYN_SIGN for neurons in range [111,96]
9 - SPI_SYN_SIGN for neuons in range [127,112]
10 - SPI_SYN_SIGN for neurons in range [143,128]
11 - SPI_SYN_SIGN for neuons in range [159,144]
12 - SPI_SYN_SIGN for neurons in range [175,160]
13 - SPI_SYN_SIGN for neuons in range [191,176]
14 - SPI_SYN_SIGN for neurons in range [207,192]
15 - SPI_SYN_SIGN for neuons in range [223,208]
16 - SPI_SYN_SIGN for neurons in range [239,224]
17 - SPI_SYN_SIGN for neuons in range [255,240]
18 - SPI_BURST_TIMEREF
19 - SPI_AER_SRC_CTRL_nNEUR
20 - SPI_OUT_AER_MONITOR_EN
21 - SPI_MONITOR_NEUR_ADDR
22 - SPI_MONITOR_SYN_ADDR
23 - SPI_UPDATE_UNMAPPED_SYN
24 - SPI_PROPAGATE_UNMAPPED_SYN
25 - SPI_SDSP_ON_SYN_STIM
```

Figure 3.6: List of SPI configuration registers that can be set up through odin_configurator application.

One may notice that some pictures concerning odin_configurator application report the string "little-endian" on the terminal. This is due to the fact that the program can handle either little or big endian systems, even if most desktop systems run

```
25 - SPI_SDSP_ON_SYN_STIM
Type a number: 2
Type a value in range [0,65535]: 4
```

Figure 3.7: Configuation of SPI_SYN_SIGN for neurons in range [15,0]. Value 4 means that neuron number 2 will have inhibitory synapses, whereas all others in that range will have excitatory synapses.

little endian processors. Please note that *little and big endian system support* is available for synapses and neurons configurations *only* at this time being.

Chapter 4

Results & Discussion

4.1 RTL Simulation

The final designed architecture, comprising ODIN and RocketChip interfaced through a SPIFlash Master module is depicted in Figure 4.1.

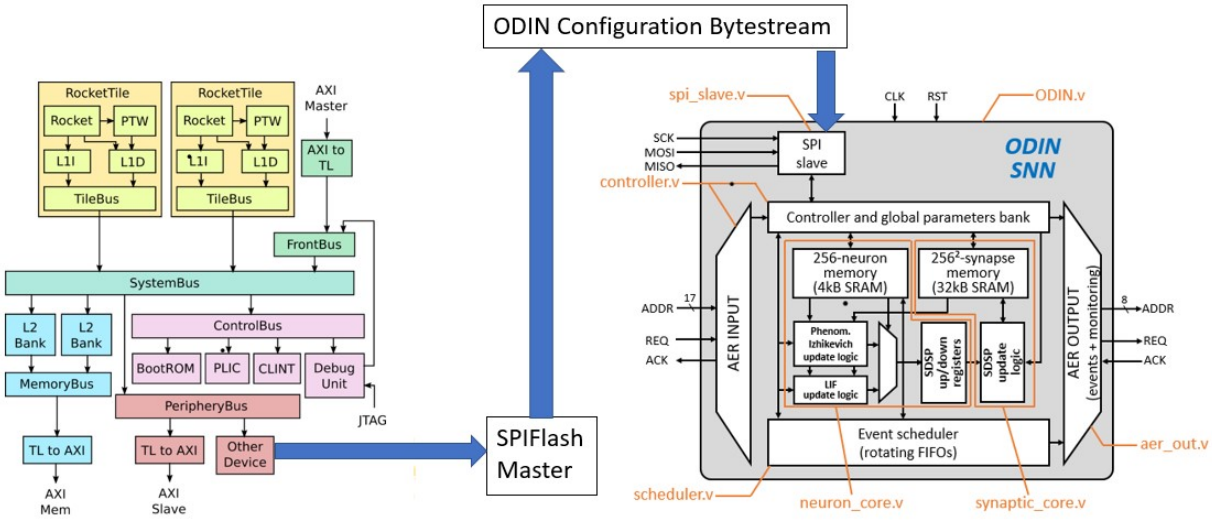

Figure 4.1: Rocket and ODIN SPI Interconnection

A Register Transfer Level (RTL) simulation has been performed to determine

whether the *synfire chain* is correctly stimulated and working. A *synfire chain* is a feed-forward (i.e. the neural network neurons are arranged so that there are no cycles) consisting of many layers of neurons. All synapes are excitatory, meaning that the postsynaptic neuron membrane potential *increases* whenever one or more presynaptic neurons fire. Once the first layer of neurons is characterized by some firing activity, all the subsequent layers are excited and fire, giving birth to a volley of spikes synchronously propagating from a layer to another [1]. The synfire chain has been chosen as a benchmark to validate the proposed architecture because of its simple and predictable behaviour. An example of synfire chain comprising 8 neurons, which constitute the network that is going to be configured in the simulation, is shown in Figure 4.3.

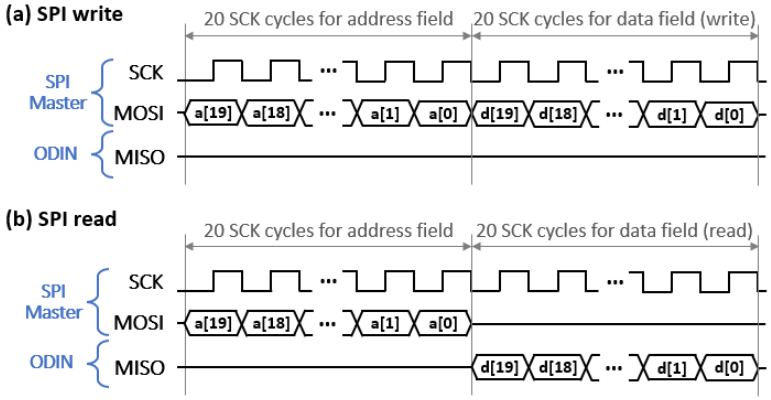

Figure 4.2: ODIN SPI Slave - Transmission Details. Every write or read operation consists of 40 clock cycles, first 20 devoted to operation address transmission, the subsequent 20 are instead needed to transfer data that should be written somewhere, according to Tab 4.2, or it consists of data that someone requested from the outside.

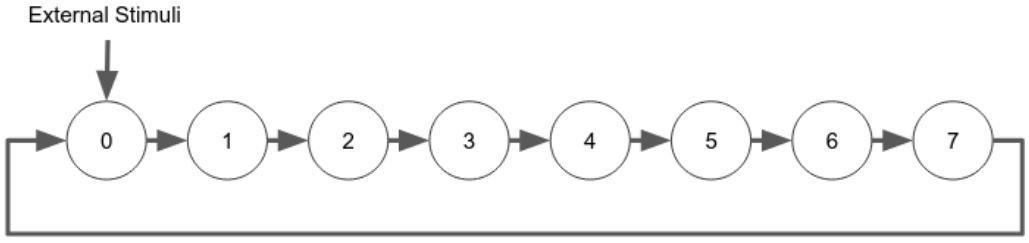

Figure 4.3: Synfire Chain Network composed of 8 neurons. This is the network that serves as validating example for the final architecture comprising ODIN and RocketChip.

Before running the simulation, issuing command in Listing 3.17, a C program, shown in Listing 4.1, has been written to correctly initialize SPIFlash master device,

let it configure ODIN, and gather the results from output AER interface of the ODIN coprocessor. The original program was developed by the authors of Chipyard framework and retained its original filename. Initially conceived to configure the SPI master and tinker with its parameters to read data from a specific area of memory, it has been modified to suit the needs of ODIN.

```
#include <stdlib.h>
#include <stdio.h>

#include "mmio.h"
#include "spiflash.h"
#define ODIN_AEROUT_REQ 0x2000
#define ODIN_AEROUT_ADDR 0x2004
#define ODIN_AEROUT_ACK 0x2008
#define ODIN_AERIN_REQ 0x200C
#define ODIN_AERIN_ADDR 0x2010
#define ODIN_AERIN_ACK 0x2014
#define SYNFIRECHAIN_NEURONS 8
int main(void)
{
  int i;

  spiflash_ffmt ffmt;
  uint8_t neurons[SYNFIRECHAIN_NEURONS];

  ffmt.fields.cmd_en = 1;
  ffmt.fields.addr_len = 4; // Valid options are 3 or 4 for our model
  ffmt.fields.pad_cnt = 0; // Our SPI flash model assumes 8 dummy cycles for fast reads, 0 for slow
  ffmt.fields.cmd_proto = SPIFLASH_PROTO_SINGLE; // Our SPI flash model only supports single-bit commands
  ffmt.fields.addr_proto = SPIFLASH_PROTO_SINGLE; // We support both single and quad
  ffmt.fields.data_proto = SPIFLASH_PROTO_SINGLE; // We support both single and quad
  ffmt.fields.cmd_code = 0x13; // Slow read 4 byte
  ffmt.fields.pad_code = 0x00; // Not used by our model


  printf("Initiating ODIN configuration...\n");
  configure_spiflash(ffmt);
  test_spiflash(0x0,0x2c6,0);
  //test_spiflash(0x0,0xabe, 0); //32 neuroni
  //test_spiflash(0x0,0x551e,0); 256
  //test_spiflash(0x0,0x36B,0); // 10 neuroni
  /** ADATTARE IN BASE A QUANTI NEURONI E SINAPSI CONFIGURO*/
printf("ALL NEURONS CONFIGURED!\n");
  reg_write32(ODIN_AERIN_ADDR,0x0021);

  reg_write32(ODIN_AERIN_REQ,1);
```

```
41  while ((reg_read16(ODIN_AERIN_ACK)) == 0);
42  reg_write32(ODIN_AERIN_REQ,0);
43
44
45  for (i = 0; i < SYNFIRECHAIN_NEURONS; i++)
46  {
47    while ((reg_read16(ODIN_AEROUT_REQ) & 0x1) == 0);
48    neurons[i] = reg_read16(ODIN_AEROUT_ADDR);
49    reg_write16(ODIN_AEROUT_ACK,1);
50    while ((reg_read16(ODIN_AEROUT_REQ) & 0x1) == 1);
51    reg_write16(ODIN_AEROUT_ACK,0);
52  }
53  printf("ALL NEURONS FIRED!\n");
54
55  /*for (i = 0; i < 10; i++) printf("Ordine %d : %d.\n", i, neurons[i]);*/
56  for (i = 0; i < SYNFIRECHAIN_NEURONS; i++)
57  {
58    reg_write32(0x89000000+(i*4),neurons[i]);
59
60  }
61
62
63  return 0;
64
65 }
```

Listing 4.1: Example program spiflashread.c.

Here are the changes that have been applied

1. observing lines 6-11, one can notice that a few preprocessor directives have been added to define memory offsets to be used to access to ODIN memory mapped registers, which store AER related signals. The offsets are those defined in 3.10.

2. line 12 shows a constant that indicates the number of neurons that constitute the configured synfire chain, so it depends on the number of neurons that have been provided to odin_configurator program. It should be changed accordingly before any simulation, if needed.

3. line 20-27 show the configuration of parameters for the SPIFlash module. Line 20 enables the command, line 21 determines the number of bytes that make up the address field, line 22 refers to the number of dummy (i.e. no operations) SPI cycles that have to be performed between a complete operation and another. Lines 23 to 25 show configuration of protocols to be employed for commands, addresses, and data transmission, making it possible to use standard mono bit SPI or the more recent one, called quad SPI, which sends 4 bits per transmission on 4 serial channels. Please note that command transmission doesn't support quad SPI mode, but standard one only. Finally, line

27 can be ignored, as it is a parameter not currently supported, whereas line 26 specifies the command to be sent to the SPI flash master. A list of SPI model commands is shown is Table 4.1.

4. line 31 calls the function devoted to set internal spi flash master module parameters according to the values in *ffmt* structure, whereas line 32 effectively starts the SPI transmission, for all bytes in range 0x0-0x2c6.

5. line 38-42 sets memory mapped input AER registers to values needed to request a VIRTUAL SYNAPSE event targeting neuron 0. Note that the AERIN_REQ signal is lowered only after AERIN_ACK is asserted.

6. line 45-52 handles output AER events as soon as neuron i fires. In particular, the software continuously (i.e. polling technique, since no interrupts are available for this peripheral) reads the memory mapped ODIN_AEROUT_REQ register, until its value becomes logic 1. This means that a neuron fired, and the neuron address is stored into neurons[i]. Then the software sends an acknowledge through register ODIN_AEROUT_ACK and waits for ODIN_AEROUT_REQ to be lowered in response to the acknowledge. Finally the acknowledge register is cleared so that ODIN can handle subsequent events. These operations are repeated for all neurons in range [0,SYNFIRECHAIN_NEURONS].

7. lines 56-60 properly store addresses of neurons that fired into external DRAM, which covers the 0x80000000-0X8FFFFFFF range in the Soc memory map, as stated into sims/verilator/generated-src /chipyard.TestHarness.SmallSPIFlashODINRocketConfig /chipyard.TestHarness.SmallSPIFlashODINRocketConfig.dts device tree source file. Thus, the defalt configuration for external DRAM make it able to store up to 4096 Mebibyte (0x10000000 possible addresses) of data, which correspond to a little more than 4 Gigabytes (GB).

Let's proceed with a step by step description of RTL simulation of a synfire chain composed of 8 neurons, namely neurons with addresses ranging from 0 to 7, that start with a zeroed membrane potential and their threshold firing voltage set to 1. Membrane potential leakage and calcium leakage are disabled. All neurons but those that are part of the neural network are disabled as well.
Figure 4.4 shows the first step. In this picture

1 mem_req_valid indicates whether the SPI transition is valid or not. Every low to high transition signals that the transmission is being executed and data is being sent to ODIN.

Command	**Further Modifications**	**Description**
cmd_code = 0x13	pad_cnt = 0 and addr_len=4 and addr_proto= SPIFLASH_PROTO_SINGLE and data_proto= SPIFLASH_PROTO_SINGLE	Slow read 4 byte address
cmd_code = 0x03	pad_cnt = 0 and addr_len=3 and addr_proto= SPIFLASH_PROTO_SINGLE and data_proto= SPIFLASH_PROTO_SINGLE	Slow read 3 byte address
cmd_code = 0x0B	pad_cnt = 8 and addr_len=3 and addr_proto= SPIFLASH_PROTO_SINGLE and data_proto= SPIFLASH_PROTO_SINGLE	Fast read 3 byte address.
cmd_code = 0x0C	pad_cnt = 8 and addr_len=4 and addr_proto= SPIFLASH_PROTO_SINGLE and data_proto= SPIFLASH_PROTO_SINGLE	Fast read 4 byte address.
cmd_code = 0x6B	pad_cnt = 8 and addr_len=3 and addr_proto= SPIFLASH_PROTO_SINGLE and data_proto= SPIFLASH_PROTO_QUAD	Fast read 3 byte address, with quad data.
cmd_code = 0x6C	pad_cnt = 8 and addr_len=4 and addr_proto= SPIFLASH_PROTO_SINGLE and data_proto= SPIFLASH_PROTO_QUAD	Fast read 4 byte address, with quad data.
cmd_code = 0xEB	pad_cnt = 8 and addr_len=3 and addr_proto= SPIFLASH_PROTO_QUAD and data_proto= SPIFLASH_PROTO_QUAD	Fast read 3 byte address, with quad data and quad address.
cmd_code = 0xEC	pad_cnt = 8 and addr_len=4 and addr_proto= SPIFLASH_PROTO_QUAD and data_proto= SPIFLASH_PROTO_QUAD	Fast read 4 byte address, with quad data and quad address.

Table 4.1: SPI Flash Module Commands. Note that cmd_proto is always set to SPIFLASH_PROTO_SINGLE and pad_code is always set to 0, as it is unused.

Figure 4.4: Synfire chain with 8 neurons: setup of ODIN SPI slave configuration registers.

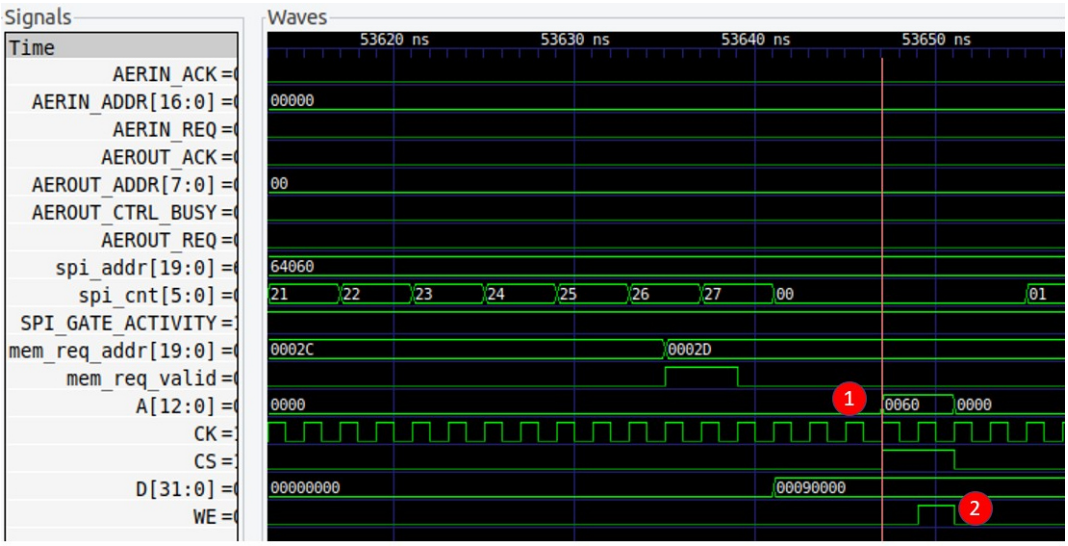

Figure 4.5: Synfire chain with 8 neurons: configuration of a synaptic interconnection by performing a write operation into the synapses SRAM.

2 mem_req_addr indicates the byte, coming from the configuration file, that spi master is reading and sending to ODIN.

3 spi_addr consists of first 20 bits sent through SPI master. The address determines the operation that is to be executed by ODIN SPI slave module, as summarized in Table 4.2.

4 spi_cnt counts the internal clock cycles, in order to distinguish incoming address bits from data related ones. Indeed, the first 20 SPI clock cycles are devoted to the transmission of the operation address, whereas the remaining 20 are dedicated to data transmission, as depicted in Figure 4.2.

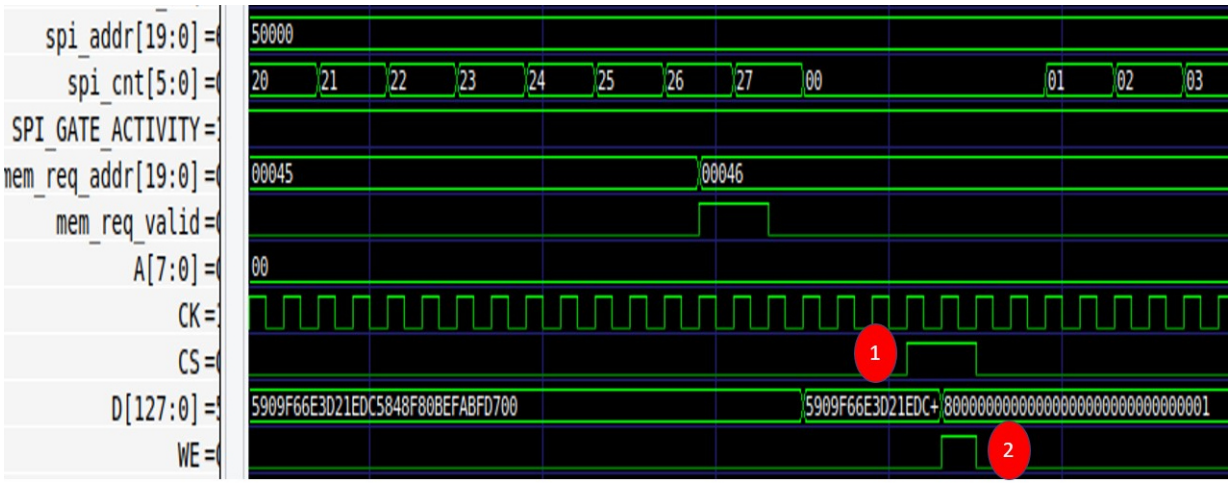

Figure 4.6: Synfire chain with 8 neurons: configuration of least significant byte of neuron 0, by performing a write operation into the neurons SRAM.

Figure 4.7: Synfire chain with 8 neurons: configuration phase ends as soon as SPI_GATE_ACTIVITY is lowered. This lets ODIN run the synfire chain.

Figure 4.8: Synfire chain with 8 neurons: neuron 0 is stimulated by a virtual synapse event (points 1-3), then every neuron of the synfire chain fires in sequence (points 4-7).

5 SPI_GATE_ACTIVITY is one of the configuration registers that is properly set during the simulation. Since it enables write and read operations on neurons and synapses SRAMs, it must be set to logic 1 before providing data to be written in those memories.

Figure 4.5 shows the second step. In this picture signals from A[12:0] down to WE refer to *neurons* SRAM.

1 A[12:0] is the address referring to the synapses SRAM. Indeed, spi_addr[19:0] contains the value 64060, which means that synapse memory must be written. In particular, synapse between neuron 3 and neuron 4 must be modified, as the byte address is 010.

2 once synapses SRAM Chip Select (CS) signal is asserted, the target address is latched into A[12:0]. In the subsequent clock cycle the Write Enable (WE) is asserted as well and data provided through D[31:0], that is ready once spi_cnt is equal to $27_{hexadecimal} = 39_{decimal}$, is written into the memory.

Figure 4.6 shows the third step. In this picture signals from A[7:0] down to WE refer to *neurons* SRAM.

1 A[7:0] is the address referring to the neurons SRAM. Indeed, spi_addr[19:0] contains the value 50000, which means that neurons memory must be written. In particular, byte 0 of neuron 0 must be modified. Once neurons SRAM Chip Select (CS) signal is asserted, the target address is latched into A[7:0].

2 In the subsequent clock cycle the Write Enable (WE) is asserted as well and data provided through D[127:0], that is ready once spi_cnt is equal to $27_{hexadecimal} = 39_{decimal}$, is written into the memory.

Figure 4.7 shows the fourth step. In this picture signals from A[7:0] down to WE refer to *neurons* SRAM.

1 A[7:0] is the address referring to the neurons SRAM. Indeed, spi_addr[19:0] contains the value 50000, which means that neurons memory must be written. In particular, byte 0 of neuron 0 must be modified. Once neurons SRAM Chip Select (CS) signal is asserted, the target address is latched into A[7:0].

2 In the subsequent clock cycle the Write Enable (WE) is asserted as well and data provided through D[127:0], that is ready once spi_cnt is equal to $27_{hexadecimal} = 39_{decimal}$, is written into the memory.

Figure 4.8 shows the last step. In this picture

1 AERIN_REQ is asserted to trigger the ODIN controller and let it accomodate the incoming AER event request if AEROUT_CTRL_BUSY is low.

2 AERIN_ADDR indicates the type of AER event that is being requested and information about it. In this case a VIRTUAL SYNAPSE EVENT, for which details can be found in 2.5, is being handled and is targeted to neuron 0.

3 AERIN_ACK is asserted once the virtual synapse event has been correctly handled and operations are complete.

4 Once the virtual synapse event processing comes to an end, neuron 0 is excited so that its membrane potential matches the configured threshold, it fires and the event is sent over output AER interface, first asserting the AEROUT_REQ and AEROUT_CTRL_BUSY signals.

5 AEROUT_ADDR is filled with the neuron number that has just fired and generated the event.

6 AEROUT_ACK is a software programmed acknowledge which informs ODIN that the latest firing event has been correctly received from the RISC-V CPU.

7 AEROUT_CTRL_BUSY indicates that the accelerator is currently unable to process other requests, either coming from internal modules or through input AER interface, and is lowered only when an external AEROUT_ACK is received.

In the end a all neurons in the range {0,7} fire in sequence, and their output AER events are correctly received and an acklowedge is sent to ODIN for each one of them. As soon as neuron 7 fires and its event is correctly handled, the firing sequence starts again from neuron 0, since neuron 7 is connected to it, and the simulation is stopped. Other simulation experiments, using larger synfire chains, have been conducted, but the one presented here was selected due to its simplicity and shorter simulation time. Simulation of larger networks, such as 256 neurons wide synfire chains, work similarly.

Read (spi_addr[19])	**Write (spi_addr[18])**	**Command (spi_addr [17:16])**	**spi_addr[15:0]**	**Description**
-	-	00	config_reg_addr[15:0]	Configuration register write at address config_reg_addr.
1	0	01	{-, byte_addr[3:0], word_addr[7:0]}	Read to the neuron memory (256 128-bit words). Byte byte_addr[3:0] from word word_addr[7:0] is retrieved.
0	1	01	{-, byte_addr[3:0], word_addr[7:0]}	Write to the neuron memory (256 128-bit words). Byte byte_addr[3:0] from word word_addr[7:0] is written.
1	0	10	{-, byte_addr[1:0], word_addr[12:0]}	Read to the synapse memory (8192 32-bit words). Byte byte_addr[1:0] from word word_addr[12:0] is retrieved.
0	1	10	{-, byte_addr[1:0], word_addr[12:0]}	Write to the synapse memory (8192 32-bit words). Byte byte_addr[1:0] from word word_addr[12:0] is written.

Table 4.2: SPI Slave Configuration Commands

4.2 Synthesis Results

This section serves the purpose of conducting a feasibility study on the entire design synthesis on Field Programmable Gate Array (FPGA), namely Xilinx PYNQ Z2, comparing timing information and occupied area on that FPGA against those obtained by ECE department of University of Virginia, for which one can find more details here, on a Xilinx Zedboard. Before proceeding, the design comprising ODIN and RocketChip must be evicted from all unnecessary components and signals which make it impossible to synthesize it on PYNQ Z2 board, due to the fact that it requires too many I/O pins (i.e. the outermost input and output wires). As shown in Listing 4.2, the following modules have been removed or modified

- no UART adapter, simply removing the chipyard.iobinders.WithUARTAdapter config fragment
- removed SiFive L2 cache. InclusiveCache config fragment has been removed, so the design exploits TileLink proprietary broadcast system. To perform this modification, just remove freechips.rocketchip.subsystem.WithInclusiveCache.
- SPI Flash master has been changed to read-only peripheral, meaning that it can only read from a given file. This is done by giving the *true* argument to chipyard.iobinders.WithSimSPIFlashModel(true) config fragment. The chipyard.config.WithSpiFlash(0x100000) fragment should be inserted as well, with the parameter between parenthesis being the size of addressable space for the SPI controller.
- external DRAM removed (no need to save ODIN results outside, as in Verilator simulation). To do so, remove the chipyard.iobinders.WithBlackBoxSimMem fragment.
- TileLink (TL) monitors removed. Simply add freechips.rocketchip.sybsystem.WithoutTLMonitors.
- smallest RISC-V RocketChip core available has been put in place of standard core. To achieve this result, remove freechips.rocketchip.system.BaseConfig and freechips.rocketchip.subsystem.WithNBigCores config fragments, then add the freechips.rocketchip.system.TinyConfig one, which will instantiate the smallest core available, use an incoherent bus interconnect rather than a coherent one, and remove all interconnects needed for external memories.

```
1 class TinyRDOnlySPIFlashODINRocketConfig extends Config(
2   // no uart
3   // no sifive l2 cache
4   // read only spi flash
5   // small core
```

```
// tiny config
// no tlmonitors
// no external DRAM
// bufferless broadcast
new chipyard.iobinders.WithTieOffInterrupts ++
//new chipyard.iobinders.WithBlackBoxSimMem ++
//new chipyard.iobinders.WithTiedOffDebug ++
//new chipyard.iobinders.WithSimSerial ++
new chipyard.iobinders.WithSimSPIFlashModel(true) ++     // add the SPI flash model in the harness (read-only)
new chipyard.example.WithODIN ++
//new testchipip.WithTSI ++

new chipyard.config.WithBootROM ++
new chipyard.config.WithSPIFlash(0x100000) ++            // add the SPI flash controller (1 MiB)
new chipyard.config.WithL2TLBs(1024) ++
//new freechips.rocketchip.subsystem.WithBufferlessBroadcastHub ++
new freechips.rocketchip.subsystem.WithNoMMIOPort ++
new freechips.rocketchip.subsystem.WithoutTLMonitors ++
new freechips.rocketchip.subsystem.WithNoSlavePort ++
new freechips.rocketchip.subsystem.WithNExtTopInterrupts(0) ++
//new freechips.rocketchip.subsystem.WithNSmallCores(1) ++
//new freechips.rocketchip.subsystem.WithCoherentBusTopology ++
new freechips.rocketchip.system.TinyConfig)
```

Listing 4.2: First line of .out file given by ODIN + ROCKET simulation, running spiflashread.riscv program.

4.2.1 Area

The synthesis result are shown in Tab 4.3 and Tab 4.4. As one can see, the resource utilization on PYNQ Z2 board is really low, accounting for 15.99 % of LUTs slices and 11.07 % of Block RAMs (BRAMs), the latter being instantiated for ODIN neurons and synapses states, RocketChip data and instruction caches. The number of I/O pins is 8

1. clock: main clock source
2. reset: global synchronous reset
3. SCK: SPIFlash Master and ODIN SPI Slave clock source
4. CS: SPIFlash Master Chip Select
5. 4 in/out pins for quad spi data transmission

and is the minimum needed to have a working design. The PYNQ Z2 provides up to 125 user programmable I/O pins (known as IOBs), so there more than enough to integrate other peripherals or systems.

Site Type	**Scope**	**Type**	**Used**	**Available**	**Utilization %**
Slice LUTs			8506	53200	15.99
	Logic		7928	53200	14.90
	Memory		578	53200	1.09
		Distributed RAM	578		
		Shift Register	0		
Slice Registers			4317	106400	4.06
		Flip Flop	4317		4.06
		Latch	0		
F7 Muxes			179	26600	0.67
F8 Muxes			34	13300	0.26

Table 4.3: ODIN + RocketCore Synthesis - Slices

Site Type	**Scope**	**Type**	**Used**	**Available**	**Utilization %**
BRAM Tile			15.5	4140	11.07
	RAMB36/FIFO		15	140	10.71
		RAMB36E1	15		
	RAMB18		1	280	0.36
		RAMB18E1	1		

Table 4.4: ODIN + RocketCore Synthesis - RAM

Chapter 5

Conclusions

This work was intended to establish a first contribution towards seamless integration of neuromorphic technologies with state of the art processors, which cannot really handle operations deep learning algorithms require to achieve the results they were conceived for. The architecture should take the best from both domains; on one side there are advantages coming from using a "standard" SoC system, as the principles behind programming such devices are known and it is for sure easier than directly interacting with a neuromorphic architecture like that on which ODIN is built, plus it exploits an open source ISA that is gaining more and more attention due to its modularity, as the base ISA is stable, minimal, and won't be discontinued in terms of support, and simplicity, plus all positive points cited in 2.2; on another side it allows for efficient execution of above mentioned algorithms due to the nature of networks involved, which further pushes down power consumption, and this is mandatory when dealing with IoT devices. All in all there is so much room for improvements, a few of which are provided in the following

1. make RocketChip able to reconfigure ODIN at run time. This means that proper hardware and sotware (APIs) support should be implemented and provided to the end user.

2. FPGA porting. Chipyard leverages a boot ROM, containing the instructions to run when the SoC is powered on, together with all details concerning the SoC modules through the Device Tree Binary structure. The SoC runs those instructions, then run a wait-for-interrupt (WFI) instruction, waiting for the RISC-V Frontend Server (FESVR) to load the program and wake up the core through an external interrupt. If one wants to deploy and run the system on a FPGA, the booting process should be changed, removing the need for an external interrupt and let the user program run as soon as the boot loader completely set up the SoC modules.

3. exploit ROcket Custom Coprocessor interface instead of MMIO/SPI, first evaluating pros and cons deriving from the possibility of exploiting custom

coprocessor specific instructions. A RoCC accelerator exploits a custom communication protocol and reserved non-standard ISA instructions, having a specific format that is detailed in [25]. This change might help in reaching the functionality of point 1.

4. change SPIFlash module to allow the user to specify separated input and output data files/streams, as at the moment it can only read and write, depending on the configuration parameters, from and to one file only.

5. change SPI slave interface of ODIN to support quad spi, thus accelerating the configuration phase. This would imply not only modifying the external interface of the SPI slave module, but also change the internal behaviour so that the controller can process 4 bits at a time, rather than one.

6. have a cluster of ODIN modules talking one to each other, so to increase the computing capabilities of such systems.

7. modify ODIN so to handle bistability and membrane potential leakage mechanism automatically, removing the dependency from external events.

Acknowledgements

This work established the first step towards the start of my professional career. Lots of difficulties showed up and I was surrounded by a multitude of people that supported me without even asking for. I'd start with a few words on my parents, Marco and Milena. They truly believed in my desire to pursue a career in engineering, although my dad initially thought I'd be better with studying law, and provided me with everything a son might ask for, love above all, and I felt encouraged in leaving home to move to Turin. This city changed my way of being, and acting, and fortified my personality while providing me with opportunities to grow up. It is also the city that gave me friendships I'd never live without, and they are countless, yet I'd like to list a few of them : Vito, Sofia, Cosimo, Samuel, Marco. A special mention goes to Francesco, who spent hundreds of hours in the past months listening to my concerns, providing hints on how I could face certain problems. I could not forget to mention my best friend Aurelio, who has always been present, no matter the distance we were at or the time that passes since the last time we had to hear each other. I'd like to thank my uncle Carlo, the man that stands as my lighthouse in my engineering path. Least but not last, I don't know how to express the gratitude I have towards Giorgia, my girlfriend since two years, who truly understood the malaise I went through and always had kind words to relieve it.

Bibliography

[1] M. Abeles. Synfire chains. *Scholarpedia*, 4(7):1441, 2009. doi: 10.4249/scholarpedia.1441. revision #150018.

[2] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[3] S. Barzegarjalali and A. C. Parker. Neuromorphic circuit modeling directional selectivity in the visual cortex. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 6130–6133, 2016. doi: 10.1109/EMBC.2016.7592127.

[4] N. Boldrini. Deep learning, cos'è l'apprendimento profondo, come funziona e quali sono i casi di applicazione, 2019. URL https://www.ai4business.it/intelligenza-artificiale/deep-learning/deep-learning-cose/.

[5] F. S. Brader JM, Senn W. Learning real-world stimuli in a neural network with spike-driven synaptic dynamics.. (English). *Neural Computation*, 19(11), 2007. doi: https://doi.org/10.1162/neco.2007.19.11.2881.

[6] Burkitt A.N. A review of the integrate-and-fire neuron model: I. homogeneous synaptic input. *Biological Cybernetics*, 95(1):1–19, 2006. doi: 10.1007/s00422-006-0068-6.

[7] J.-D. L. C.Frenkel, M.Lefebvre and D. Bol. A 0.086-mm^2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28-nm CMOS. (English). *IEEE Transactions on Biomedical Circuits and Systems*, 13(1):145–158, 2019. doi: https://doi.org/10.1109/TBCAS.2018.2880425.

[8] A. L. Chandra. Mcculloch-pitts neuron — mankind's first mathematical model of a biological neuron, 2018. URL https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1.

[9] Q. Chen, Q. Qiu, H. Li, and Q. Wu. A neuromorphic architecture for anomaly detection in autonomous large-area traffic monitoring. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 202–205, 2013. doi: 10.1109/ICCAD.2013.6691119.

[10] K. Cheung, S. R. Schultz, and W. Luk. Neuroflow: A general purpose spiking neural network simulation platform using customizable processors. *Frontiers in Neuroscience*, 9:516, 2016. ISSN 1662-453X. doi: 10.3389/fnins.2015.00516. URL https://www.frontiersin.org/article/10.3389/fnins.2015.00516.

[11] G. Crebbin. Image segmentation using neuromorphic integrate-and-fire cells. In *2005 5th International Conference on Information Communications Signal Processing*, pages 305–309, 2005. doi: 10.1109/ICICS.2005.1689056.

[12] Daniel Faggella. Machine learning in robotics – 5 modern applications, 2020. URL https://emerj.com/ai-sector-overviews/machine-learning-in-robotics/.

[13] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017. doi: 10.1109/PATMOS.2017.8106976.

[14] DeepAI. Feature extraction. URL https://deepai.org/machine-learning-glossary-and-terms/feature-extraction.

[15] E. Donati, F. Corradi, C. Stefanini, and G. Indiveri. A spiking implementation of the lamprey's central pattern generator in neuromorphic vlsi. In *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*, pages 512–515, 2014. doi: 10.1109/BioCAS.2014.6981775.

[16] J. Dungen and J. . Brault. Simulated control of a tracking mobile robot by four avlsi integrate-and-fire neurons paired into maps. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 695–699 vol. 2, 2005. doi: 10.1109/IJCNN.2005.1555936.

[17] l. ETH Zurich, University of Bologna. Ibex documentation, 2021. URL https://ibex-core.readthedocs.io/en/latest/index.html.

[18] o. G. ETH Zurich, University of Bologna. Cv32e40p documentation, 2021. URL https://cv32e40p.readthedocs.io/en/latest/.

[19] W. . C. Fang, B. J. Sheu, O. T. . C. Chen, and J. Choi. A vlsi neural processor for image data compression using self-organization networks. *IEEE Transactions on Neural Networks*, 3(3):506–518, 1992. doi: 10.1109/72.129423.

[20] C. Frenkel. Odin spiking neural network (snn) processor. URL https://github.com/ChFrenkel/ODIN.

[21] C. Frenkel. Bottom-up and top-down neuromorphic processor design: Unveiling roads to embedded cognition, 2020. URL http://hdl.handle.net/2078.1/226494.

[22] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown. Overview of the spinnaker system architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, 2013. doi: 10.1109/TC.2012.142.

[23] W. Gerstner and W. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002. URL http://lcn.epfl.ch/~gerstner/SPNM/SPNM.html.

[24] B. Glackin, J. Harkin, T. M. McGinnity, L. P. Maguire, and Q. Wu. Emulating spiking neural networks for edge detection on fpga hardware. In *2009 International Conference on Field Programmable Logic and Applications*, pages 670–673, 2009. doi: 10.1109/FPL.2009.5272339.

[25] U. B. A. R. Group. Chipyard's documentation, 2021. URL https://chipyard.readthedocs.io/en/latest/index.html.

[26] L. Hardesty. Explained: Neural networks, 2017. URL https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

[27] N. Imam, T. Cleland, R. Manohar, P. Merolla, J. Arthur, F. Akopyan, and D. Modha. Implementation of olfactory bulb glomerular-layer computations in a digital neurosynaptic core. *Frontiers in Neuroscience*, 6:83, 2012. ISSN 1662-453X. doi: 10.3389/fnins.2012.00083. URL https://www.frontiersin.org/article/10.3389/fnins.2012.00083.

[28] G. Indiveri and S. C. Liu. Memory and information processing in neuromorphic systems. *Proceedings of the IEEE*, 103(8):1379–1397, Aug 2015. ISSN 0018-9219. doi: 10.1109/JPROC.2015.2444094.

[29] G. Indiveri, E. Chicca, and R. Douglas. A vlsi array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity. *IEEE Transactions on Neural Networks*, 17(1):211–221, 2006. doi: 10.1109/TNN.2005.860850.

[30] G. Indiveri, F. Stefanini, and E. Chicca. Spike-based learning with a generalized integrate and fire silicon neuron. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 1951–1954, 2010. doi: 10.1109/ISCAS.2010.5536980.

[31] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004. doi: 10.1109/TNN.2004.832719.

[32] J. Jones. Mesi cache coherency protocol, 2017. URL https://www.scss.tcd.ie/Jeremy.Jones/VivioJS/caches/MESIHelp.htm.

[33] J. Joshi, A. C. Parker, and T. Celikel. Neuromorphic network implementation of the somatosensory cortex. In *2013 6th International IEEE/EMBS Conference on Neural Engineering (NER)*, pages 907–910, 2013. doi: 10.1109/NER.2013.6696082.

[34] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Sivilotti, and D. Gillespie. Silicon auditory processors as computer peripherals. *IEEE Transactions on Neural Networks*, 4(3):523–528, 1993. doi: 10.1109/72.217193.

[35] J. . Lee and B. J. Sheu. Parallel digital image restoration using adaptive vlsi neural chips. In *Proceedings., 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 126–129, 1990. doi: 10.1109/ICCD.1990.130181.

[36] D. Marković. Spiking neural networks : What makes these biologically realistic neurons interesting for computing?, 2020. URL https://towardsdatascience.com/spiking-neural-networks-558dc4479903.

[37] McCulloch and Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, (5):115–133, Dec 1943. doi: 10.1007/BF02478259.

[38] C. Mead. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636, Oct 1990. ISSN 0018-9219. doi: 10.1109/5.58356.

[39] S. . Moon, K. G. Shin, and J. Rexford. Scalable hardware priority queue architectures for high-speed packet switches. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 203–212, 1997. doi: 10.1109/RTTAS.1997.601359.

[40] W. Murphy, M. Renz, and Q. Wu. Binary image classification using a neurosynaptic processor: A trade-off analysis. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1342–1345, 2016. doi: 10.1109/ISCAS.2016.7527497.

[41] NVIDIA Developer. Top 5 ai speech applications using nvidia's gpus for inference, 2019. URL https://news.developer.nvidia.com/top-5-ai-speech-applications-using-nvidias-gpus-for-inference/.

[42] D. Patterson and A. Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 1st edition, 2017. ISBN 0999249118.

[43] A. Pullini, D. Rossi, G. Haugou, and L. Benini. udma: An autonomous i/o subsystem for iot end-nodes. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017. doi: 10.1109/PATMOS.2017.8106971.

[44] U. B. A. Research. Chipyard documentation, . URL https://chipyard.readthedocs.io/en/latest/.

[45] U. B. A. Research. Chipyard framework, . URL https://github.com/ucb-bar/chipyard.

[46] RISC-V Foundation. Extending risc-v, 2019. URL https://five-embeddev.com/riscv-isa-manual/latest/extensions.html.

[47] RISC-V Foundation. The risc-v instruction set manual - volume i : Unprivileged isa, 2019. URL https://riscv.org/technical/specifications/.

[48] Roger Shepherd. What is coherent interconnect in arm soc design?, 2017. URL https://www.quora.com/What-is-Coherent-Interconnect-in-ARM-SoC-Design.

[49] M. Schmuker, T. Pfeil, and M. P. Nawrot. A neuromorphic network for generic multivariate data classification. *Proceedings of the National Academy of Sciences*, 111(6):2081–2086, 2014. ISSN 0027-8424. doi: 10.1073/pnas.1303053111. URL https://www.pnas.org/content/111/6/2081.

[50] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank. A survey of neuromorphic computing and neural networks in hardware. *CoRR*, abs/1705.06963, 2017. URL http://arxiv.org/abs/1705.06963.

[51] SiFive Inc. Sifive tilelink specification, 2019. URL https://www.sifive.com/documentation.

[52] M. Sorbaro, Q. Liu, M. Bortone, and S. Sheik. Optimizing the energy consumption of spiking neural networks for neuromorphic applications, 2020.

[53] M. Thomas. 15 examples of machine learning in healthcare that are revolutionizing medicine, 2020. URL https://builtin.com/artificial-intelligence/machine-learning-healthcare.

[54] A. W. Trask. *Grokking Deep Learning*. Manning, 2019. URL https://www.manning.com/books/grokking-deep-learning.

[55] S. University. Extended address event representation draft standard v0.4, 2002. URL https://web.stanford.edu/group/brainsinsilicon/documents/methAER.pdf.

[56] Wikipedia Community. Artificial neural network — Wikipedia, the free encyclopedia, 2021. URL https://en.wikipedia.org/wiki/Artificial_neural_network.

[57] Wikipedia Community. Brain-computer interface — Wikipedia, the free encyclopedia, 2021. URL https://en.wikipedia.org/wiki/Brain\OT1\textendashcomputer_interface.

[58] Wikipedia Community. Approfondimento profondo — Wikipedia, the free encyclopedia, 2021. URL https://it.wikipedia.org/wiki/Apprendimento_profondo.

[59] Wikipedia Community. Serial peripheral interface — Wikipedia, the free encyclopedia, 2021. URL https://en.wikipedia.org/wiki/Serial_Peripheral_Interface.

[60] E. Zurich and U. of Bologna. Pulp silicon proven designs, 2021. URL https://pulp-platform.org/implementation.html.