POLITECNICO DI TORINO

Corso di Laurea in Computer Engineering

Master Degree Thesis

Preliminary integration of a neuromorphic coprocessor in the Risc-V Pulp platform

Relatore
Gianvito Urgese

Supervisor
Michelangelo Barocci

Candidato
Jonathan Damone

April 2023

Contents

List of Figures

Abstract

The aim of this thesis is to offer a comprehensive guide for constructing the proposed SOC, consisting of a traditional processor and a neuromorphic coprocessor. To achieve this goal, I conducted a thorough analysis of the state-of-the-art PULP platforms and the latest neuromorphic processors. Following this, I identified the most suitable candidates that meet high standards of power and flexibility, ultimately selecting PULPissimo and Reckon, interconnected through the uDMA module.

To verify the functionality of the proposed configuration, I conducted an initial test by launching data from memory serially to Reckon, and subsequently validating the transmission through uDMA with the PULP toolchain. The results of this test provided valuable insights into how to simulate various modules using QuestaSim, and a detailed guide on how to effectively operate the PULP platform.

CHAPTER 1

Introduction

Neuromorphic computing is a relatively new field of study that combines the principles of neuroscience with traditional computing in order to create a more efficient and powerful computing systems miming the human brain with a neuron-synaptic model. This approach to computing has the potential to achieve higher energy efficiency, faster processing speeds, and greater adaptability to new situations, which could lead to breakthroughs in fields such as robotics, artificial intelligence, and cognitive computing.

One of the key benefits of neuromorphic system is its potential for energy efficiency, indeed the power consumption of this system is really low compared to traditional one that are power-hungry and, doing so,will limit their potential for use in mobile and other battery-powered devices. Neuromorphic systems, on the other hand, can be designed to operate on very low power, allowing for more widespread deployment of intelligent systems in a variety of settings.

In this context the aim of this thesis is to prepare the ground for placing neuromorphic processors alongside to traditional processors. By moving in this direction we have the opportunity to harness the power of neuromorphic computing along with the universality and versatility of traditional systems. As a first approach we obviously need to see, for both neuromorphic and traditional, which systems are most suitable for our purposes. Once they have been selected the next step is to find a method of connecting and finally integrating them. This preliminary research must be done through both software models and hardware specifications.

CHAPTER 2

Background

In this chapter, we will delve into a comprehensive exploration of traditional RISC-V and neuromorphic processors, examining their current state of the art. Additionally, we will analyze the potential communication bridges between these two types of processors. Finally, we will compare and contrast the two and determine the optimal combination that achieves both low power consumption and high versatility.

2.1 Neuromorphic architecture

2.2 Odin

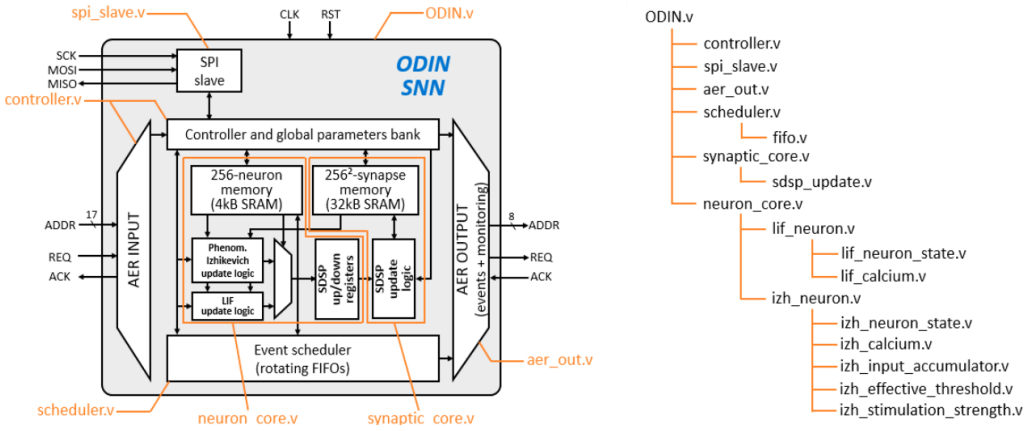

Figure 2.1: Odin [1]

ODIN is a digital online-learning spiking neuromorphic processor published in 2019 . ODIN is based on a single 256-neuron 64k-synapse crossbar neurosynaptic core [1]. The principal feature that this architecture has is the possibility to configure the network as the user prefer thanks to synaptic plasticity (SDSP) and the possibilities to individually reach out and activate a specific neuron. ODIN was conceived to increase the density of neurons in the silicon and to try to minimize the power consumption in comparison of other solutions that was proposed in SNNs domain ,that could be seen on the table on 2.5. From the purely neuron management point of view, Odin has an infrastructure called a 2d mesh topology architecture, which is great for prototyping but energy-intensive.
In fact already other authors have identified that 2d mesh is a power hungry solution and multiple authors have suggested to perform a replacement with a Mixed-mode hierarchical-mesh routing architecture that would bring ODIN even closer to the goal of being a lightweight

circuit [12]

Neuron Core

Each neuron is associated to a 128-bit word in the neuron memory, which contains its parameters and state information that are stored in single-port SRAM, The most-significant bit (MSB) of each word allows individually disabling neurons (1: disabled, 0: enabled).

The least-significant bit (LSB) of each word allows individually choosing the model for each neuron.
These parameters are preloaded during the initial SPI configuration of ODIN. For setting the various parameters ODIN uses an SPI interface to write on the SRAM.

Neurons can be individually configured as an 8-bit leaky integrate-and-fire (LIF) model or as a custom phenomenological Izhikevich (IZH) neuron model, that allows for considerable flexibility in which model you want to rely on. LIF and IZH neuron models were designed to be entirely event-driven, so the activation Is done through the stimulation of neurons.// Neurons can be stimulated in 7 different way:

- Single-synapse event: Two log N-bit addresses are provided: source neuron i and the address of destination j. This event is handled similarly to an AER neuron spike event, but only the neuron j of ODIN will be updated, together with a single SDSP weight update to synapse i of neuron j. in other word the source do not change but the destination does.
- Single-neuron time reference event:Activates a time reference event for neuron
- All-neurons time reference event: Activates a time reference event for all neurons.
- Single-neuron bistability event:Activates a bistability event for all synapses in the dendritic tree of neuron
- All-neurons bistability event:Activates a bistability event for all synapses in the crossbar array.
- Neuron spike event: stimulates all neurons with the synaptic weight associated to pre-synaptic neuron
- Virtual event:Stimulates a specific neuron with weight without activating a physical synapse

Synaptic Core

The synapses are charged with 8192 word sized on 32-bit in an SRAM memory. Each synapse occupies 4 bits, so 8 synapses are accessed per word. The first MSB is used for mapping table used to enable on-line learning on the synapses and the other three LSBs are used to set the synaptic weight. The main role of this core is to connect the various neurons to each other through 2d programmable mesh connections that allow the creation of communication infrastructures prioritized with the respect to the established history of connection frequency.
Through the manipulation of the weight with the synaptic SRAM.

crossbar

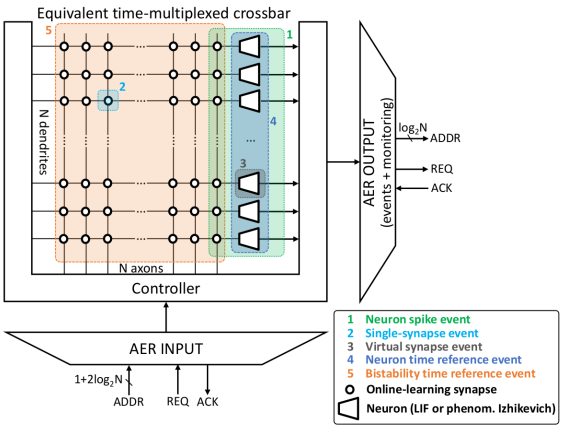

Figure 2.2: crossbar [1]

Once we have described the neurons, which represent the computational engine, and once we understand what synapses are, the next step is to describe the communicative bridges between this two and understand how this two component are linked inside ODIN.
As can be seen from the figure 2.2.
The trigger that will activate the SNN is the arrivals of the AER packets direct to the neurons network.
From the picture 2.2 it is possible to see how the controller plays a key role in the harmony of the information distribution process through the crossbar.

Controller

To control the whole process ODIN uses a Moore's FSM with 14 possible stages:

- W_NEUR: write on Neuron SRAM
- W_SYN: write on synapses's memory
- R_NEUR: read on Neuron SRAM
- R_SYN: read on synapsis's memory
- TREF : from Input AER interface is detected a time reference event
- BIST : from Input AER interface is detected a reference bistability
- SYNAPSE : from Input AER interface is detected a single synapse
- PUSH: AER event that should be handled onto the scheduler
- POP_NEUR: if scheduler have an event to be processed
- POP VIRT: if scheduler have an event to be processed, AER must be virtual event
- WAIT : wait that output AER interface is free for transmitting new data
- WAIT_SPIDN: SPI write parameters in SRAM memory it took 40 SPI clock cycles
- WAIT REQDN: ODIN receives an input from AER input interface and the IC have to manage that.
- RST : Reset the system

Scheduler

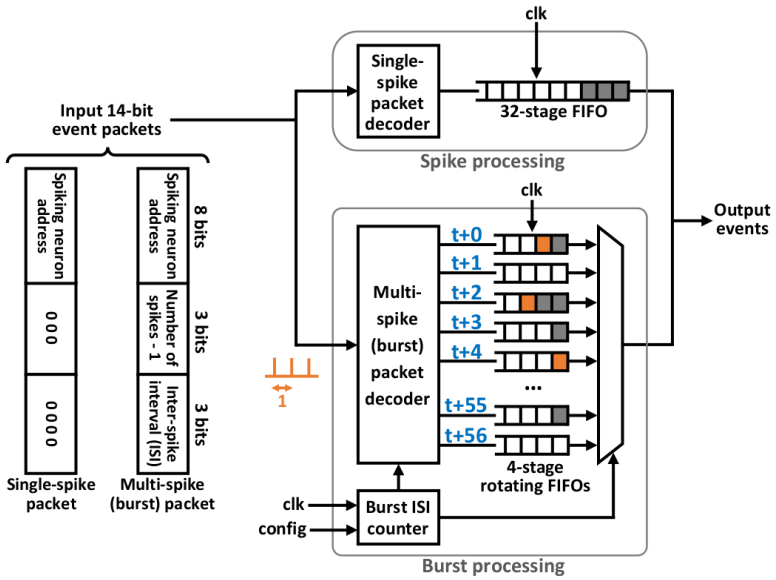

Figure 2.3: scheduler [1]

The scheduler structure is shown in 2.3 and can be thought as a priority-based FIFO. The need of this component is related to the problem that ODIN is a system that works in parallel; so, when we perform for example an output burst operation, it is necessary to develop a buffer component that allows optimal scheduling to connect the processed information and sequence it at the right time. In detail spiking and bursting neurons of ODIN send 14-bit event packets to the scheduler. The packets contain:

- 8-bit address of the source neuron
- 3-bit field indicating the number of spikes to
- 3-bit field quantifying the ISI be generated

while all single-spike events are stored in the scheduler, if the ODIN would like to perform a multi-spike events, the system will send the burst packet into a decoder. The role of the decoder is to divide the burst into multiple single-spike events and spread among buffers that have the role to send AER data sequentially to the output event. In the ODIN documentation they describe this process with the FIG. 2.3 : In first place arrives a neuron event packet and the packet, and it is divided if this packet is a single or multiple type. Then Three single-spike events that will be generated by the burst packet decoder if the packet is a multi packet toward FIFOs associated. The FIFO has a sequentially time steps that starts from l+0 (will be processed immediately by the controller) and will continue until it reaches 56; Each Buffer unit keep 8-bit address of the source spiking neuron.

Simulation configuration

ODIN is an open source project, so is provided the verilog source that could be simulated in a behavioral simulation whit any tool(Modelsim or Questasim).
For the implementation and the synthesis the behavioral SRAMs need to be replaced with the Block RAM that it is used for FPGA implementations (as Vivado).

Specifications

the most important thing in systems is energy consumption:
P = P_leak + P_idle x f_clk + E_SOP

- P_leak: leakage power without clock activity
- P_leak + P_idle x f_clk :power consumption of ODIN without any activity in the network with clock
- E_SOP :includes the contributions of reading and updating the synaptic weight according to the SDSP learning rule, reading and updating the associated neuron state, as well as the controller and scheduler overheads, in other words the neuromorphic system.

With the measurement done directly to ODIN on the FPGA at 0.55 V with a P_leak of 27.3 uW and P_idle of 1.78 uW/MHz plus the energy of SOP OF 8.43 pJ, the overall measured power of ODIN is 477 uW

Metric	measured
Implementation	Digital
Technology	28nm FDSOI
Neurosynaptic core area [mm 2]	0.086
Izhikevich behaviors	20
neurons per core	256
Synaptic weight storage	4 bit
Embedded online learning	SDSP
synapses per core	64k
Time constant	BIO
Neuron core density [neur/mm 2]	3.0k
Synapse core density [syn/mm 2]	741k
Supply voltage	0.55V 1.0V
interface	SPI and AER
Leakage_power	27.3uW at 0.55V
Idle power	1.78uW/MHz at 0.55V
Total_power	477 uW

2.3 ReckOn

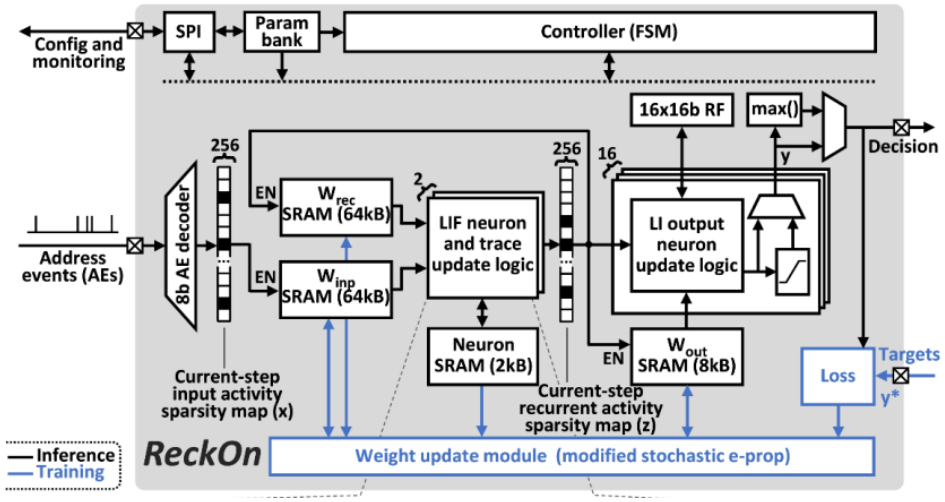

Figure 2.4: ReckOn [2]

ReckOn is a spiking recurrent neural network (RNN) processor that is able to perform on-line learning [2]. It was prototyped and measured in 28-nm FDSOI CMOS and published at the 2022 and it has two main features:

- end-to-end on-chip learning in a milliseconds temporal resolution
- Low-cost solution and low memory usage

The Target of Reckon is to overcome the memory requirements problem that conventional neural network training algorithms has. This Memory usage lead to power and cost problem. Most of the architecture proposed use external SRAMs whit only one access point that bring problems related to Von neumann bottlenecks to the processor and breaks down the computational capabilities and turn down the ability to learn in short timescale.
To get this result Reckon drastically has reduced the memory requirements, power and area budgets for tasks related to gesture recognition and navigation.

2.4 shows the system diagram and the architecture of ReckOn, which implements the spiking RNN topology and computes the network dynamics in a time-stepped fashion thanks FSM controller. The communication is interfaced by using an input AER produced by asynchronous neuromorphic peripheral equipment with an 8-bit 4-phase-handshake AE decoder.

Controller

To control the whole process ODIN uses a Moore's FSM with 6 possible stages:

- IDLE: is the non-working state in which all outputs are to zero and ReckOn is waiting for an internal or external signal.
- PROP : hidden neurons for learning are activated
- STEP: AER input writing state, neurons are updated according to the assigned parameters
- SDONE : Input received, is forwarded to the process or output states
- CONFIG: Configuration of synaptic and neuronal parameters through the SPI port

- EPROP:Activation of the Eprop algorithm for resolution
- SEND:output the result of the computation into the AER output port

Testing features

One of the most emerging advantages of this architecture is the ready-to-test datasets that accompany this processor for demonstrating on-line learning. This demonstration hold data contains two single 50-sample batches (one for training, one for test) of the delayed-supervision navigation task, whose size and complexity are suitable for RTL simulations in most common simulator.

Simulation configuration

Reckon is an open source project, so it is provided the verilog source that could be simulated in a behavioral simulation whit any tool(Modelsim or Questasim).
For the implementation and the synthesis the behavioral SRAMs need to be replaced with the Block RAM that it is used for FPGA implementations (as Vivado).

Specifications

	This work	VLSI'15 [1]	VLSI'17 [7]	ISSCC'18 [2]	VLSI'18 [3]	ISSCC'19 [4]	ISSCC'18 [5]
Technology Implementation Core area Memory Energy metric	28nm Digital 0.45mm² 138kB 5.3pJ/SOP [a]	65nm Digital 1.8mm² 37.6kB 5.7pJ/pix	40nm Mixed-signal 1.3mm² N/A 48.9pJ/pix	65nm Mixed-signal (IMC) 0.8mm² 16kB 0.32pJ/OP	10nm Digital 1.72mm² 896kB 3.8pJ/SOP	65nm Digital 10.1mm² 353kB 0.29pJ/OP	55nm Mixed-signal 3.1mm² 0.4kB 0.32pJ/OP
Network type # Neurons # Synapses (width)	Spiking RNN (256)-256-16 132k (8-bit)	Spiking LCA [b] 4x64 83k (4,5,14-bit)	Spiking LCA [b] 8x64 N/A	SVM 128 8k (16-bit)	Multicore SNN 64x64 1M (7-bit)	Binary NN (784)-200-200-10 194k (14-bit)	ANN (3) - 84 - 3 0.5k (6-bit)
On-chip learning - algorithm - multilayer - dynamics	✓ Mod. stoch. e-prop ✓ Few ms to seconds	✓ SGD ✗ ✗	✓ N/A ✗ ✗	✓ SGD ✗ ✗	✓ STDP ✗ Few ms	✓ Mod. SD [c] ✓ ✗	✓ SGD ✓ ✗
Task	Hand gesture classif. Keyword spotting Navigation	Image classif.	Image classif.	Image classif.	Image classif.	Image classif.	Obstacle avoid.
Dataset	IBM DVS Gestures [f] Spiking Heidelberg Digits [g] Delayed cue integration [h]	MNIST	MNIST	MIT CBCL [d]	MNIST [e]	MNIST	Custom autonomous robot
Average input data depth	Gest: 1318 steps @Δt=5ms KWS: 104 steps @Δt=5ms Nav: 2250 steps @Δt=1ms	1 frame	1 frame	1 frame	1 frame	1 frame	N/A (1-step decisions)
Accuracy with on-chip training	Gest: 87.3% @10classes KWS: 90.7% @1word Nav: 96.4% @2decisions	84%-90%	88%	91.6%	89%	97.8%	N/A
Power (infer / learn)	Gest: 77µW / 135µW [a] KWS: 79µW / 150µW [a] Nav: 62µW / 114µW [a]	268mW / 526mW	87mW / N/A	1.3mW / 3.1mW	6.2mW / N/A	23.6mW / 23.1mW	690µW / N/A
Energy per step (infer / learn)	Gest: 35nJ / 85nJ [a] KWS: 42nJ / 178nJ [a] Nav: 0.6nJ / 1.5nJ [a]	27-162nJ / 94.7µJ	50.1nJ / N/A	42pJ / 150pJ	1.0µJ / N/A	236nJ / 254nJ	0.69nJ / 1.5nJ

[a] At 0.5V, 13MHz, accelerated-time [b] Locally-competitive algorithm [c] Segregated dendrites algorithm [d] Downscaled to 11x11 [e] Pre-processed with Gabor filters
[f] From [8], downscaled to 16x16, 10 classes [g] From [9], English digits 0-9, channel subsampling 1:3, target vs. filler word ratio 1:1 [h] As specified in [6]

Figure 2.5: Comparison table [2]

The table above allows you to see how compared to all the other developed architectures this one is the most efficient and versatile. For all tasks the supply voltage can be scaled down to 0.5V, which allows a power budgets of 150uW. With a core area of 0.45 mm^2 with 138 Kbyte of SRAM storage compared with other works without external memory nor pre-training it is possible to see that this architecture has an energy values of 1.5-178nJ during learning process at 0.5V.

Metric	measured
Implementation	Digital
Technology	28nm FDSOI
Neurosynaptic core area [mm 2]	0.45
ODIN and Synaptic weight storage	8 bit
Embedded on-line learning	EPROP
synapses per core	132k
Time constant	BIO
Neuron core density [neur/mm 2]	0.57k
Synapse core density [syn/mm 2]	293k
Supply voltage	0.5V 1.0V
interface	SPI and AER
Leakage_power	13uW at 0.55V
Idle power	-
Total_power	150 uW

Specifications ReckOn

2.4 RANC

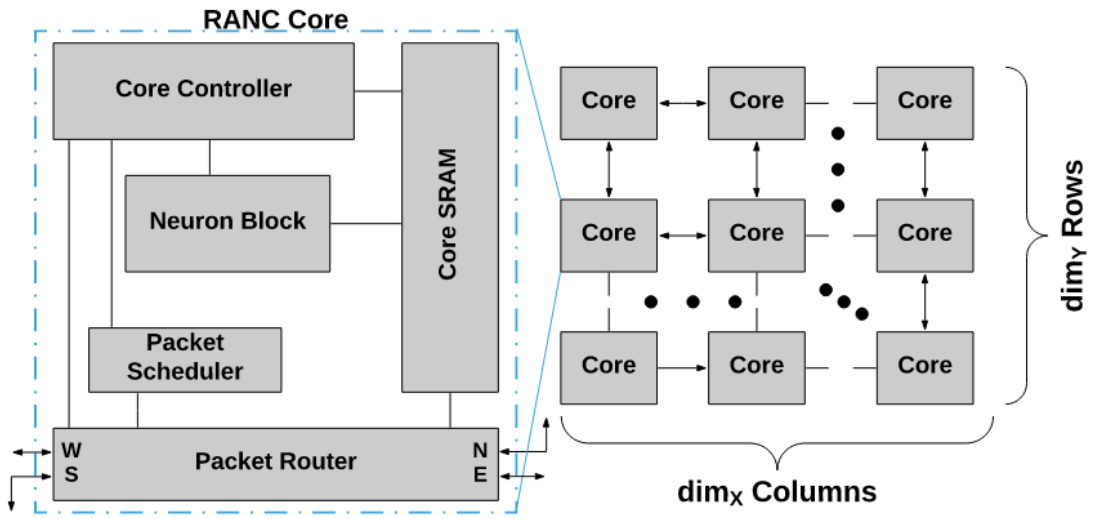

Figure 2.6: RANC [3]

The configurable Architecture for neuromorphic Computing is an open source board able to be deployed in several FPGA [3].
RANC mimes the behavior of biological neurons in the brain using spike model. The engine architecture is based on 2D mesh network-on-chip composed of cores that have the same structure: neuron block, core controller, core SRAM, packet router, and packet scheduler.
This structure allows to build a Network like behaviors that are implemented via the neuron block that it is coupled, with a controller and an SRAM, that will support memory and processing operation.
The output of the neurons are routed between neuron blocks thanks to routing network via the packet scheduler. Should be noticed that no performance value has been provided because this architecture it was used only in researching platform.

Neuron Block

The neuron block emulates a crossbar connected to the output, called neurons, and input, called axon, that is able to emulate a synaptic connection.
The neurons weight are stored in a hardcoded index inside the SRAM. In the reset moment, the neuron block, contains a basic datapath that will be updated for mimicking the voltage characteristics of an LIF neuron model.

This model allows the board to put each input spike that a neuron receives on one of its axons; This signal will be weighted with the current neuron potential that it is stored in the SRAM. The main idea is that a neuron has a certain potential value that should be exceeded for trigger an activation event.
Once this event occurs may appear a decay of potential if it is parameterized by the user in the configuration phase.
At the end of its computation cycle the neuron's final potential value will be written into the SRAM core .

Core controller

The core controller applies a digital force to the neuron block datapath to coordinate memory accesses and data transfers using FSM. The role of core controller is the continued iteration

with the input axons to checking if the axon-neuron crossbar contains a neural connection or not .
If, for example, connection is present, the weight associated with that axon is sent to the neuron block datapath for accumulation into the neuron potential. Once all input spikes are processed for a given neuron, the controller checks if this neuron has produced an output spike. If it has, the controller sends a handshake signal to the router for enqueue spike to target destination.

Core SRAM

It contains configuration of each neuron using a matrix which column has the number of bits required to encode all parameters for a single neuron. All parameters for each neuron (weights, connections, current potential, reset values, thresholds, leak value, destination of generated spikes) are stored as a single word in the core SRAM. After the computation it is done the potential value is updated and core SRAM is committed back to memory in a single write.

Packet Router

The packet router is in charge to carry information to neurons destination to be used as an axon input around the chip using XY algorithm.

Packet Scheduler

When a packet arrives to destination the role of core's scheduler is scheduling. This process is done using the decompression of upcoming packet in base of how long this input spike should wait before being processed by the core and the destination axon. Both values are used to point to an index pointer to an auxiliary memory that is used in as FIFO .

Simulation configuration

In order to stream data, RANC needs an FPGA with at least two arm cores.
For this reason was used the ZYNQ Ultra-Scale MPSOC using Xilinx Vivado 2018.2. For exporting the hardware it was used Xilinx SDK. In other word all the project was based on Xilinx's platform

2.5 SENeCA

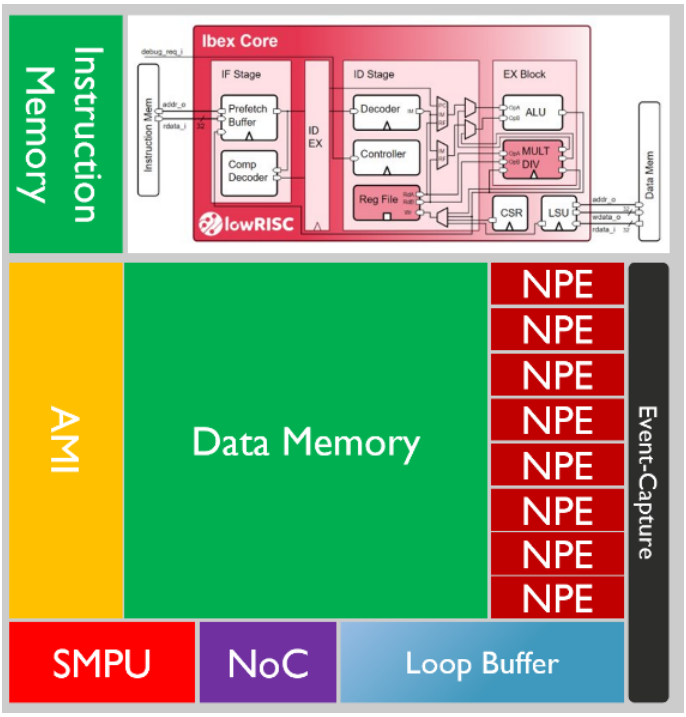

Figure 2.7: SENeCA [4]

SENeCA it is the latest architecture that was released in 2022 and, declared as open source, it will be uploaded freely really soon. Based on 64 Neuron Compute Cluster (NCC) that contain a RISC-V core (Ibex), instruction and data memory, Axon Message Interface (AMI), Network on Chip (NoC), Share Memory Pre-fetch Unit (SMPU), and several other units. [4]

Ibex core

RISC-V Ibex core is used as a controller for the NCC. 2-stage pipeline and uses the RV32IMC instruction set.
As declared the choice of this unit was done due performance efficiency Flexible memory allocation. this structure will be studied later in Pulp section.

Axon Massages Interface

AMI is a programmable accelerator that manages the events as filtering and control. AMI has the power to send interrupts signal to the Ibex when there are enough events in the input queue to be processed and there are no more processing space.

Shared Memory Pre-fetch Unit

SENeCA use hierarchy memory to be able to share memory between a few NCCs to be able to go beyond the area limitations of SRAM for memory-intensive applications. Using shared memory is optional and only required when the internal memories are insufficient to store the application parameters, and at the same time, it is not feasible to tile more SENeCA chips for the target application

Neuron Co-Processor

Neural Co-Processor have the role to execute the primary neural operations emulating silicon neurons by accelerating the most common neuromorphic instructions. This accelerator has a Neuron Processing Elements which it is used for primary operation, and it has also a small memory made from register files and a processing unit that executes a category of most common neuromorphic instructions that can be executed one per clock cycle. Inside this processor there is also the Event-Capture Unit, able to takes the input spike vector and converts them to the form of Address Event Representation (AER)and a Loop Buffer used to update several hundreds of neurons.

Network on Chip

To connect the NCCs and deliver the spike events SENECA has implemented a 2D matrix with some features that supports multicasting on cluster and also supports variable-length packets. Multicasting is implemented by using source-based addressing. In this method, the packet contains the source address instead of the destination address in conventional NoCs. This method avoids the need of building hierarchical structure. The multicasting was done to reduce the amount of data communication over the NoC by for improving the NoC performance

Simulation configuration

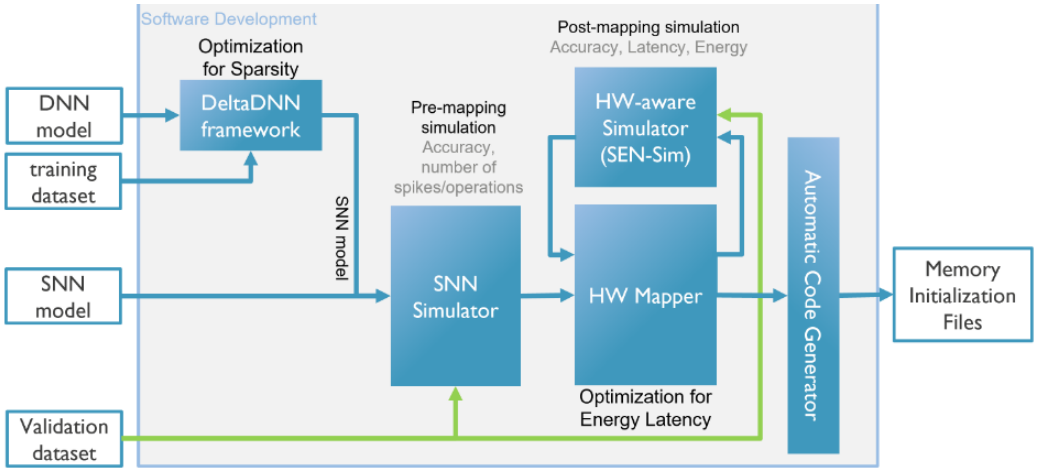

Figure 2.8: SDK architecture for SENeCA [4]

SENeCA will be Integrated with a specific SDK that will set up all the environment. Since it is A 2022 architecture the SDK it is not already ready to be published. Regarding the implementation seneca was implemented with a Xilinx Virtex-7 FPGA (XCVU35p, in 50MHz) that use 4 Block-RAM (instruction memory) and 8 Ultra-RAMs (data memory).

2.6 Ariane

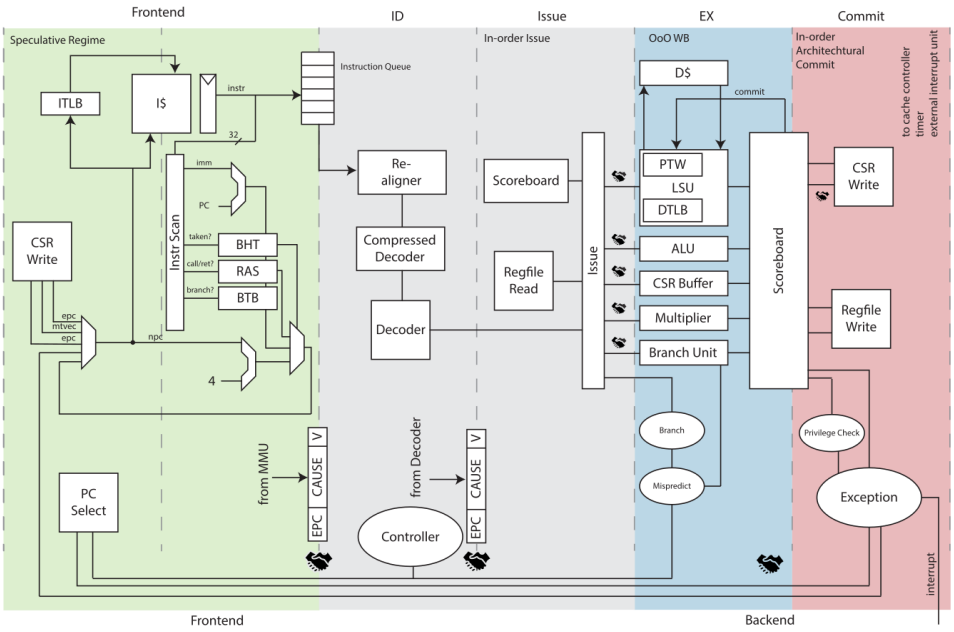

Figure 2.9: Ariane pipeline [5]

Ariane is a RISC-V architectures owned by PULP opensorce that is able to run Linux.
It is a 64 bits architecture with the ability to perform integer, IEEE compliant Floating Point Unit and atomic memory operations. In terms of capability Ariane is able to perform also multiplication and division operation.
Like all RISC5 processors, Ariane also has a staged structure as shown in the following image. 2.9 The pipeline is divided through Frontend and Back-end :one is in charge to receiving information from the SRAM and the other execute the instruction stream.

- **PC Generation** :This stage has the role to select the next Program Counter by increment the previous one. From here there is an exception handler called Control and Status Registers will be call when inside the pipeline there is some stall behaviors as exception, debug interface or mispredicted branch

- **Instruction Fetch** translates the virtual address into the physical one to be trodden from the instruction cache.
  This stage is a pre-decode logic that has the power to guide the branch-prediction by Address Translation and Cache pipelining. Address Translation is performed by fetching from the previous stage the page-offset (12 LSB) and the virtual one (12 to 39 bit) that will load Introduction that are stored inside the cache. While Cache pipelining is when cache's data arrays is processed before being pre-decoded in the next stage

- **Instruction Decode**:The aim of this stage is to re-aligns potentially unaligned instructions with the use of a specific unit called re-aline and perform a decompression to decodes them instruction in the right order that will be putted in an issue queue in issue stage ready to be launched.

- **Issue Stage**: Issue stage has the role to read operands, in integer or float register, and detect possible dependency conflicts. This is done with the Re-order Buffer (ROB) and the scoreboard (Dependencies are tracked in the scoreboard and operands are forwarded from the ROB if necessary)

- **Execute Stage**: is where operation are performed, using ALU, multiplier/divider, FPU and the load/store unit (LSU).if some instruction is not ables to be executed Ariane uses a retired out-of-order buffer that will be loaded again in the execution unit when Write-back conflicts are resolved through the ROB.
- **Commit Stage**:authorizes the result to be written to the SRAM or in the data cache, Stores and atomic memory operations

Scoreboard

One of the most atypic structure is inside the Issue Stage : The scoreboard.
It is made with a first in and first out queue that are sourced with the decoded stage. Instruction that are stopped by a stall operation, that will be required to be inserted in the execution unit, will be placed in this component. Once the stall has been completed and the preceding instructions have been committed, the scoreboard push in the execution unit the delayed instruction.

Functional Units

Ariane contains six functional units that it is used to perform classic RISCV operation:

- **ALU**: Covers most of the RISC-V base ISA, including branch target calculation
- **LSU**: Manages integer and floating-point load/stores as well as atomic memory operations
- **FPU**: IEEE compliant floating-point
- **Branch unit**:extension to the ALU which handles branch prediction and branch correction.
- **CSR**: atomic instruction processor
- **Multiplier**: fully pipelined two-stage multiplier.
- **Divider**: bit-serial divider with input preparation from 2 to 64 cycles

Hardware Control Interfaces

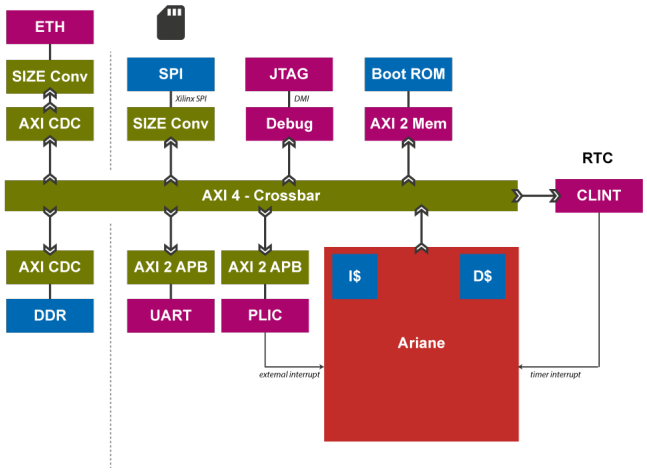

Figure 2.10: Ariane Hardware Control and general purposes Interfaces. [5]

One of the most important parts for this thesis is to evaluate which architecture has a more suitable configuration for integrating an external interfaces. So it becomes clear that the interfaces available for this architecture it is a discriminatory key.

AXI

The core contains a single Advanced extensible Interface (AXI) with five-master port as well as four interrupt sources from the slave unit.
The advanced extensible Interface is a parallel on-chip bus with multi-master and multi-slave communication interface. It was chosen this one because AXI has Separate address/control and data phases and because it is flexible. This protocol it is adopted by Xilinx as primary communication bus for their products [13] and also for the ZYNQ model so it is a welcome feature. In the figure 2.10 you can see the various interfaces and how Ariane could be inserted in a SOC.

UART and SPI

For the communication from outside to inside, Ariane uses the classic serial communication buses. when a certain sequence is intercepted on these input serial ports Ariane triggers a handler routine to stall the process according to the operations that should be performed. Note that this block can be used with a UART or SPI to configure the architecture.
To be more specific UART and SPI are both serial communication protocols; While UART is a full-duplex protocol, which means it can send and receive data simultaneously, SPI is a half-duplex protocol, which means it can only send or receive data at any given time.
UART is typically used for short-range communication between two devices, while SPI is used for communication between multiple devices. UART requires two wires for communication, while SPI requires four.
UART is generally used for transmitting data over short distances, while SPI is used for transmitting data over longer distances.

Specifications

The specifications that we are going to fetch are come form the references paper [5]. As pointed before Ariane can communicate in a SoC via a full-duplex 64-bit address AXI interconnection and 16 kB of instruction cache data.
The SoC that was used it was developed with a small memory that contains 520 kB of on-chip memory interfaced with several peripherals such as HyperRAM, SPI, UART, and I2C.
Regarding the power Ariane is able to produce 192 mW in 22nm FDSOI

Simulation configuration

The simulation of Ariane it is possible to do with verilator and also Questa Sim. For the FPGA emulation it is used Vivado 2018.2 with Xilinx Genesys 2. For debug and program the FPGA it is done using OpenOCD interfaced and attached to the FTDI 2232 USB-to-serial chip on the Genesys 2 board.

2.7 Pulpino

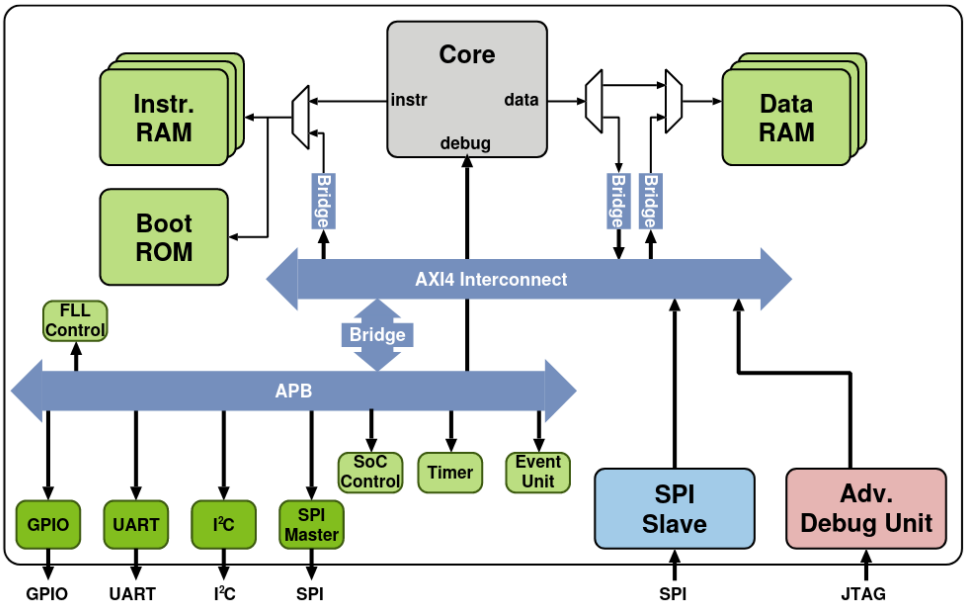

Figure 2.11: Pulpino pipeline

The PULPino is a single-core System on Chip developed by the Integrated Systems Laboratory (IIS) of ETH Zürich and Energy-efficient Embedded Systems (EEES) of the University of Bologna. At the top is possible to see the block diagram of the PULPino [15].
the core communicates directly with the memories through the on-chip bus AXI4, this architecture based on three masters: core, IP SPI Slave and the IP Advanced Debug Unit. while the other memories are accessed as a AXI slave as GPIOs, UART, I2C and SPI. In the top figure is also possible to see that inside the peripheral there is a JTAG input;
The advanced debug unit AXI'S master interface allows to perform the debug operation of the system from outside, but the most important feature is that any interface could be used for this target as SPI or any other interface. In terms of memories is possible to see that there are a BOOT ROM, instruction RAM, and memories RAM.

GPIO, UART, I²C

GPIO

The PULPino input, output and general purpose peripheral (GPIO) has nine 32-bit registers that are useful to perform operation outside the PULPino Board . In those nine the most Notable are the PADDIR, PADIN and PADOUT registers: The first controls the data direction of each of the GPIO pins, the second is used for the input pins and the last one for the output pin. using this register it is possible to connect this board to an external board, here below the GPIO interface

Signal	**Direction**	**Description**
gpio_in[31:0]	**input**	Transmit Data
gpio_out[31:0]	**output**	Receive Data
gpio_dir[31:0]	**output**	Request to Send
gpio_padcfg[5:0][31:0]	**output**	Pad Configuration
interrupt	**output**	Interrupt (Rise or Fall or Level)

GPIO Signals

UART

Another important feature that PULPino has is the serial interface UART with this signal, below there is the table with the UART interface:

Signal	Direction	Description
uart_tx	**output**	Transmit Data
uart_rx	**input**	Receive Data
uart_rts	**output**	Request to Send
uart_cts	**input**	Clear to send
uart_dtr	**output**	Data Terminal Ready
uart_dsr	**input**	Data Set Ready

External UART Signals

I²C

I²C is an open-drain signaling protoco, that mean that high logic values are obtained by using a pull-up resistor in the SDA and SCL lines.Table below shows the I²C signals interface.

Signal	Direction	Description
scl_pad_i	**input**	SCL Input
scl_pad_o	**output**	SCL Output (always 0)
scl_padoen_o	**output**	SCL Pad Direction
sda_pad_i	**input**	SDA Input
sda_pad_o	**output**	SDA Output (always 0)
sda_padoen_o	**output**	SDA Pad Direction
interrupt_o	**output**	Event/Interrupt

I²C Signals

core

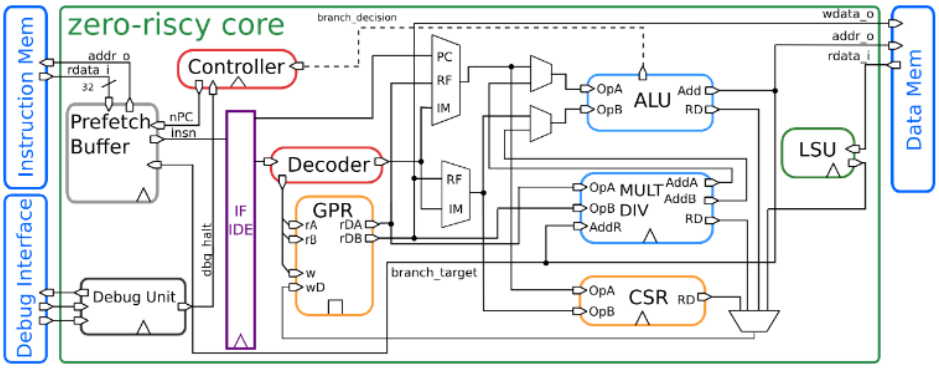

Figure 2.12: Ariane pipeline

ZERO-RISCY

This core was an upgrade from RI5CY,and is more light because provides only 2 stage. This choice was taken because the SOC's need a small and low latency ISA architectures. The reduction of the stages from fives DLX to 4 of RI5CY and at the end with ZERO-RISCY at 2 means that in comparison with a ancient DLX, 100 instruction with no dependency are

processed in 500 clk cycle for DLX, 400 clk for RI5CY (20% clock cycles saved), 200 clk for ZERO-RISCY (60% clock cycles saved).

PULPino supports both the RISC-V RI5CY and the RISC-V zero-riscy ISA because both have The same external interface, but the inner is different. The difference between this two is only in the operation that actually they can perform. RI5CY has more internal fragmentation with the separate multiplication and division unit, while the zero-riscy has a more compact architecture by combining the DIV/MUL unit in the same functional block. below some detail

RI5CY

The architecture RI5CY is a 4 stage pipeline structure that is able to perform integer and floating point operation.
Another important feature that RI5CY brings is the Interrupts routines,Exceptions, and the management of events that allow the core to go in IDLE set when the core is not called to perform an operation

Simulation configuration

The toolchain is ModelSim with a versions greater of 10.2c for simulation proposes. PULPino can be synthesized and deployed on a ZedBoard from the Xilinx, so the synthesis software is Vivado.

2.8 Pulpissimo

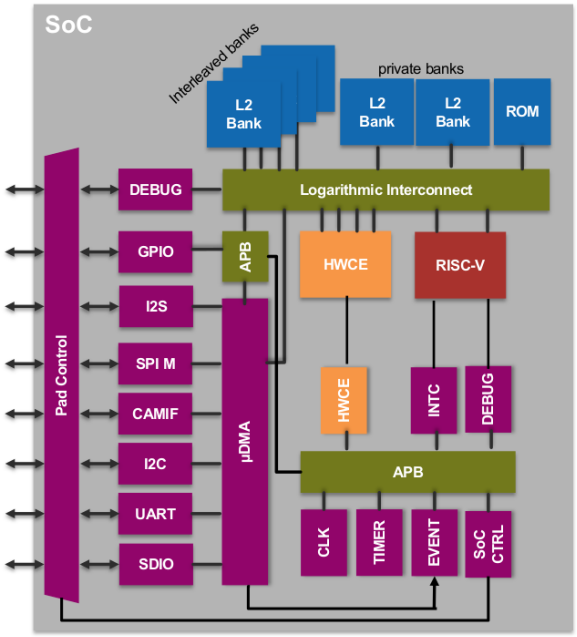

Figure 2.13: PULPissimo

PULPissimo is a 32 bit RI5CY single-core System-on-a-Chip.
Differently from the PULPino, PULPissimo is able to perform more complex memory manipulation and organization and it has an autonomous I/O subsystem and new peripherals. Having inherited the structure directly from its predecessor, this platform also can be configured at design stage to use the RISC-V or zero-riscy core. Regarding the peripherals those are connected to the uDMA that is in charge to transfers the date to the memory subsystem in a more efficient and clever way.

As PULPino also here we have the JTAG and the the AXI interface to access to the SOC propriety. Also the debug unit is present and it is used to access to system and core registers, memories and memory-mapped IO via JTAG. In particular, in the focus of this thesis we can see that the presence of the AXI on-bus interface can be used to extend the PULPino with a co-multi-core cluster or an accelerator.

FLL

The frequency-locked loop is a compensator circuit that has the main task to compare the actual frequency of an oscillator and intervene in such way that the frequencies will be adjusted to a certain reference that the FLL has stored in a register.
In other words the frequencies automatically raises or lowers with in input the oscillator until the output of FLL matched the reference. PULPissimo contains 3 FLL that are used for generating the correct clock.

- FLL that is act in the peripheral domain
- FLL that is employed to the core domain (core, memories, event unit)
- FLL that is employed in the the cluster domain

But is important to understand that FLL is only a feature that could be used because all the three FLLs can be bypassed by adding an external clock that take control over this domain.

μDMA Autonomous I/O Subsystem

The μDMA is a special peripheral bus that is able to have two ports that are linked to memory area. Those direct connections limit the μDMA to access only the system memory and do not allow direct transfers to the processing subsystem or to other peripheral mapped on the APB bus.
The ports towards the memory are 32-bit wide while the single channel interface toward the peripherals may have smaller widths from 8 to 32 bits due peripheral configuration.

The data transfer on TX and RX channels are completely decoupled and not synchronized, this is done to allow to reads and writes data at the same time with no conflict.

In this structure the master's channels will transfer data to/from the memory. At the end of each transaction μDMA generates a dedicated event to notify that is possible to queue another transaction.
When it is happens each channel can have only one transaction running and one enqueued; the last one will be automatically started at the end of the running transaction to support continuous data flow based on a double buffering scheme.

HWPE

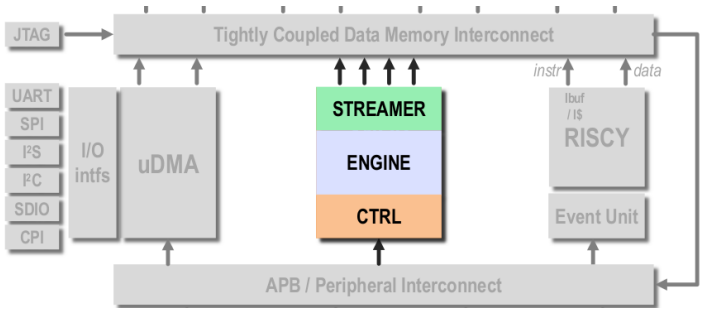

Figure 2.14: Hardware Processing Engines

The presence of a hardware accelerator inside this system is to be taken into account precisely because the coprocessor that we are going to implement serves exactly for this purpose, even if the structure is no longer digital, but the presence of this component, HWPE, accentuates the fact that this system is more suitable because is able to manage a neural network hardware oriented information system.
The architecture of HWPE is divided in three stages: the first one that start the execution of an accelerations process is a control phase that main role to handshake the activation signal that it comes from the APB peripheral bus and, this phase, will set up all the initial parameters. when the control phase is finished it start the ENGINE phase that have the role to execute the operation of accelerations of Neural network.
The last phase is the Data stream, is like an write back where all the data that was produced in the ENGINE phase will be available in the Tightly Coupled Data Memory Interconnect, and then in the memory unit.

Peripherals list

FLL, APB GPIO, Event Controller, Interrupt Controller, APB Timer, uDMA (IIC,SPI,UART,CAMIF,I2S, AXI).

Simulation configuration

The toolchain is based on Questasim exclusively for RTL base and for the simulation on higher level is need only Pulp-sdk. PULPissimo has been implemented on FPGA for the various Xilinx FPGA boards as Digilent Genesys2, Xilinx ZCU104, Xilinx ZCU102, Xilinx VCU108, Digilent Nexys Board Family and ZedBoard

B

2.9 CORE-V X-Interface

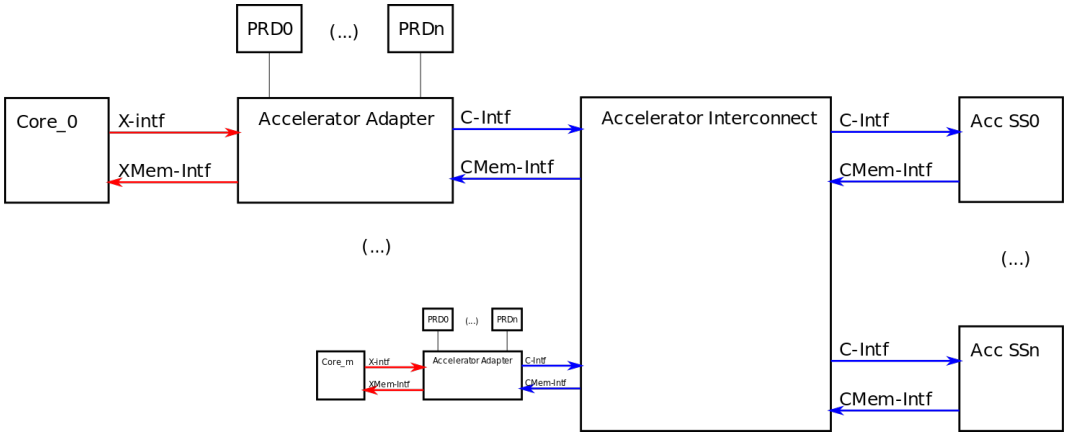

Figure 2.15: CORE-V-xif

RISC-V eXtension interface is a generalized framework suitable to extend ISA of a RISC-V Processor offloading instructions using the not used ISA and writeback the results in the same Bus that the processor is lied [14]. In the documentation available it is possible to se that a processor is extended whit a coprocessor for reducing the task that the main processor have to do and delegate the work to this last; in particular this extensor was used for Bit Manipulation, Integer Multiplication and Division, Single-Precision or Double-Precision Floating Point or implement custom extensions(what we're interested in). All this operation it is done outside the processor using the memory inusage, in that way it is possible to avoid to modify the processor itself, in other words extending instructions without the need to change the RTL using opcodes which are not used by the processor ISA adding Custom instructions that will be performed by The coprocessor as load/store or ALU .

The most important feature that this extension has it is possible to resume as follow:

- Minimal requirements on extension instruction encoding.
- Support for dual writeback instructions (optional, based on X_DUALWRITE).
- Support for dual read instructions (per source operand) (optional, based on X_DUALREAD).
- Support for ternary operations.(three source operands.)
- Support for instruction speculation.

Interfaces

We generally refer to a type x interface only as the one that connects the processor to the CORE-V X-Interface. while we will talk about c interface all others

X-intf

The X-Interface defines in total of four independent channels of communication between the accelerator adapter and the offloading processor core. The X-Request and X-Response channels s and and

- X-Request: route instruction offloading request
- X-Response: writeback responses FROM accelerator
- XMem-Request: memory transaction requests

- XMem-Response:memory transaction responses

This four channel are subject to working in this four handshake :

- The offloading core asserts an ‘valid‘ status in the decode stage and initialized a transaction.
- ‘ q_instr_data ‘ must remain stable.
- check if no pending writeback in the destination register.
- Asserts a ‘*q_ready*‘if the instruction is accepted by any of the connected pre decoders
- The instruction is not accepted by any of the connected pre decoders.
- When both ‘q_valid‘ and ‘q_ready‘ are high, the transaction is successful

C-intf

The C-Interface implements signal routing from and to the accelerator units whit four independent channels for communication between the accelerator adapter and the accelerator units:

- C-Request: route instruction offloading
- C-Response: writeback responses
- CMem-Request: memory transaction requests
- CMem-Response:memory transaction responses

This four channel are subject to working in this four handshake :

- The initiator asserts an ‘valid‘ status.
- Once ‘valid‘ has been asserted all data must remain stable.
- The receiver asserts ‘ready‘ whenever it is ready to receive the transaction. Asserting ‘ready‘ by default is allowed. While ‘valid‘ is low, ‘ready‘ may be retracted at any time.
- When both ‘valid‘ and ‘ready‘ are high the transaction is successful.

Accelerator adapter

The accelerator adapter module implements accelerator-agnostic instruction offloading from the CPU core to the accelerator interconnect. The core-side connection implements the instruction offloading using X-interface while the connection through accelerator is performed by the C-interface. The adapter module operates in conjunction with an array of accelerator-specific This module is the first interface whit the Core 0, and it is really versatile for the most common ISA architecture (32,64,128) and also if it is supported dual-writeback instructions and ternary operations to be offloaded in the external engine.

Accelerator interconnect

The accelerator interconnect module implements the interconnect connection on each level and the interconnect hierarchy. It comprises a crossbar for routing requests and responses from a generic number of requesting units as well as a bypass-path to forward requests from and to a higher hierarchy level.
All in and output ports implement the C-interface.

Specifics on CORE-V X-Interface

at the theoretical level this interface allows to build a system that can rely on more coprocessors and thus decrease considerably more the rendering time and performance, in the practical act, instead many parts of the open-source code was deleted by the authors probably because they wished to effect a more recent release that however was never done, so this interface will not be able to be used

2.10 Hardware Processing Engines

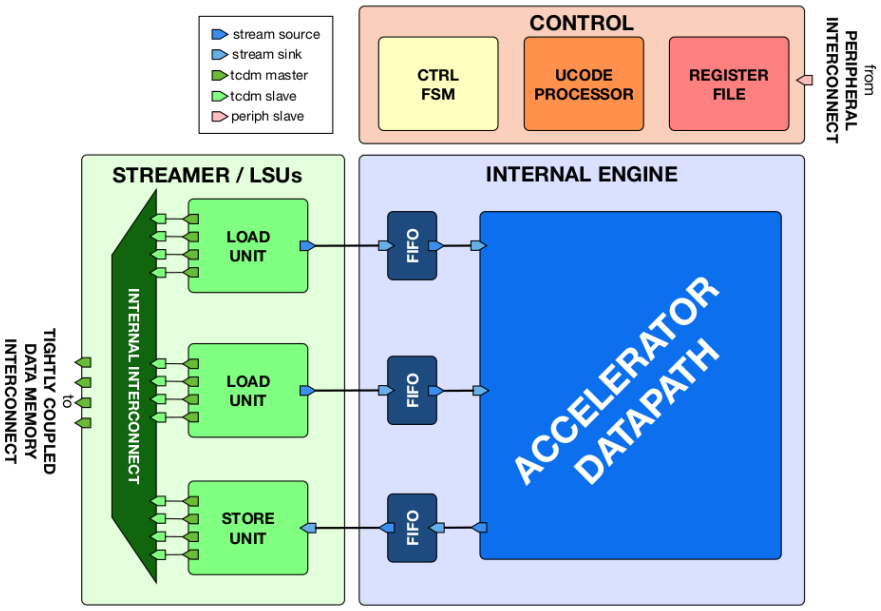

Figure 2.16: Template of a Hardware Processing Engine (HWPE)

Hardware Processing Engines (HWPEs) are special-purpose, memory-coupled accelerators that can be inserted in the SoC or cluster of a PULP system to amplify its performance and energy efficiency in particular tasks. the HWPE is support multicore and do not rely on DMA architecture but operate directly on the same memory that is shared by other elements in the PULP system whit a limited number of pointer and parameters. The HWPE is a stream of entity's that pass towards memory system, a control/peripheral interface used for programming it and an engine containing the actual datapath of the accelerator. So it possible to imagine that solution like an 3 stage out processor component.

HWPE-Stream protocol

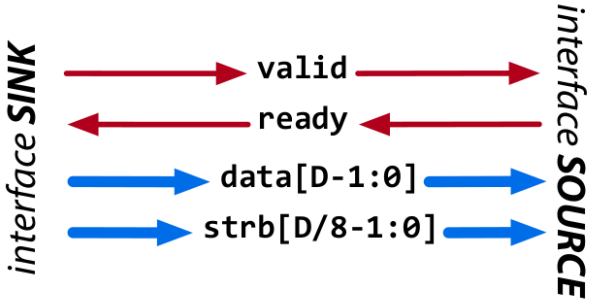

Figure 2.17: HWPE-Stream protocol [6]

The HWPE-Stream protocol is a simple protocol designed to move data between the various sub-components of an HWPE. As HWPEs are memory-based accelerators, streams are typically generated and consumed internally within the accelerator between fully synchronous devices. HWPE-Stream can cross between two clock domains using dual-clock FIFOs; handshakes still have to happen in a fully synchronous way. There is the need of only

4 channels, two for handshakes and 2 for payloads. This protocol could be summarized in these steps

- valid and ready are asserted
- data and strb can change their value when valid is de asserted or the cycle following a handshake

HWPE-Mem

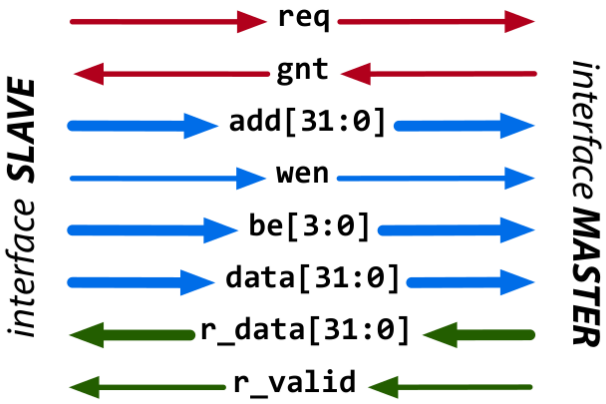

Figure 2.18: HWPE-Stream protocol [6]

HWPEs are the protocol that are connected to external L1/L2 shared-memory, using a request/grant handshake.
The protocol used is called HWPE Memory (HWPE-Mem) protocol, and it is essentially similar to the protocol used by cores and DMAs operating on memories in standard PULP clusters. It supports neither multiple outstanding transactions nor bursts, as HWPEs using this protocol are assumed to be closely coupled to memories.
It uses a two signal channel handshake and carries two phases, a request and a response. The HWPE-Mem protocol is used to connect a master to a slave. For summarize the total number of channel is 8. This protocol could be summarized in these steps

- req and gnt are asserted
- r_valid must be asserted in the cycle after a valid read handshake.
- r_data must be valid on the same cycle
- transition complete

HCI-Core

HCI-Core (Heterogeneous Cluster Interconnect Core) is a protocol designed as a light weight extension of HWPE-Mem better suited for the needs of accelerators, and specifically of cluster-coupled HWPEs. This novel interface protocol support bursts,and in-order multiple outstanding transactions.
Differently from HWPE-Mem, HCI-Core uses a two signal handshake but also includes an load signal to support load back pressure on the response phase.
HCI-Core carries two phases, a request and a response,p arithmetic width; hci_parameters reports the parameters used by the HCI IPs; while hci_signals. In this protocol the two phases of HCI-Core transactions can be treated as two separate channels, so HCI-Core transactions can be latency insensitive and support multiple in-order outstanding transactions Request and response phases are organized to be treated like HWPE-Stream streams. But this protocol it is suitable for clusters architecture

Controller

This component it is the most important one and inside it is possible to see that there is an FSM, a memory-mapped register file to store parameters that should not change and and parameters for base addresses for each type of data and a microcode processor that support write and read registers and also other registers to save parameters

Engine

the motor entity is a generic component that has to be programmed by us, in our case it could be seen as the co-processor so the motor is simply the operational offloading unit and this can be of any type, so also neuromorphic

2.11 Design choices

After all the parts that will make up our SOC have been analyzed at a high level, now it is time to choose the best combination of these elements to achieve a configuration that comes closest to our goal, obviously the parameters of choice will be functional and specific to each architecture presented.

RISCV

The decision about which platform to choose to use analyzed several criteria, also taking into consideration the goals that we want to archive.
the candidates are Ariane,PULPino and PULPissimo.

The main criterion is to choose a processor that is lightweight but also powerful that can accommodate neuromorphic solutions and easy extensibility for offloading. Ariane presents an excessively classical architecture with an AXI-based communication protocol. If we were to choose this processor we will have the limitation that our processor fits the axi architecture leading to a bottleneck. So the choice falls between PULPino and PULPissimo. should we choose PULPino we will have a higher degree of abstraction through the APB protocol . The obvious solution is PULPissimo precisely because it has a higher versatility and orientation to neural networks than previous Of course, integration can be done in two ways: through the DMA as a common peripheral or, as we will probably do, through replacing the HWPE by including the neuromorphic processor as an engine. In this way we do not have the need to change the risk architecture since the coprocessor itself becomes part of the Risc architecture, since it is treated as such

Intra-processor interface

now that the choice has fallen on PULPissimo the next step is to choose how to integrate the processor and coprocessor, the choice obviously falls on HWPE precisely because,being the only complete and working one anyway, it also has the role of being the protocol that is used in PULPissimo

Neuromorphic coprocessor

In this section we are going to compare the two architectures that was just seen: Odin and ReckOn. the reason for that is because this architecture are a stabile and well-know, neuromorphic oriented architecture while Seneca and Ranc are too new for to be considered. the benchmarks will only be qualitative by checking in particular the computing power .

From the point of view of documentation it can be said that ODIN is clearly superior, this is because the purpose of that research was to create a first working prototype so the documentation that was produced is exhaustive and complete, on the other hand RECKON has poor documentation and also the code is poorly commentated.

This leads to a greater difficulty. On the other hand, from the point of view of the technology used, the two processors use the same base, this means that, since the RECKON area is half of ODIN'S one and the lower consumption of RECKON, cit is possible to get at the conclusion that an ODIN is approximately two RECKONs but a RECKON can perform the same operations as an ODIN with a net lower time and power.
Aside for documentation an similarly RECKON wins on all other parameters : less area, less power, enhanced synapse programming from 4 to 8 bits, innovative algorithm to handle neuronal operations, this is why RECKON is our neuromorphic Coprocessor.

Metric	ODIN	**ReckOn**
Implementation	Digital	Digital
Technology	28nm FDSOI	28nm FDSOI
Neurosynaptic core area [mm 2]	0.086	**0.45**
Izhikevich behaviors	20	-
neurons per core	256	256
Synaptic weight storage	4 bit	**8 bit**
Embedded online learning	SDSP	**EPROP**
synapses per core	64k	**132k**
Time constant	BIO	BIO
Neuron core density [neur/mm 2]	3.0k	**0.57k**
Synapse core density [syn/mm 2]	**741k**	293k
Supply voltage	0.5V 1.0V	0.5V 1.0V
interface	SPI and AER	SPI and AER
Leakage_power	27.3uW at 0.5V	**13uW at 0.5V**
Idle power	1.78uW/MHz at 0.55V	-
Total_power	477 uW	**150 uW**

Specifications ODIN and ReckOn

Proposed SOC architecture

We decide to structuring the system on chip whit PULPissim as RISCV, the ReckOn as Neuromorphic coprocessor and the HWPE as intra-processor interface

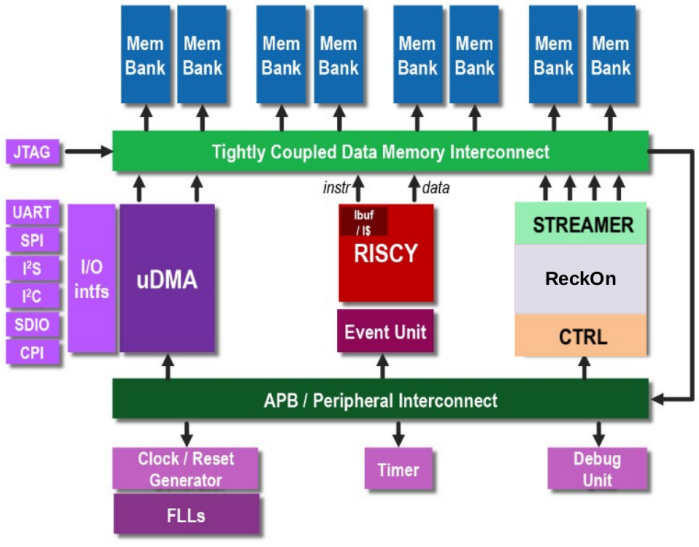

Figure 2.19: SOC Implementation [7]

CHAPTER 3

Materials and method

In this chapter, we will embark on a journey of discovery, exploration, and experimentation. Our focus will be on testing and demonstrating the choices we have made so far. By doing so, we hope to gain a deeper understanding of the world of pulp and its many derivations.

Our first task will be to explore the world of pulp and its various forms. This will involve carrying out tests that will show us how it works and what its properties are. We will look at different types of pulp, such as mechanical pulp, chemical pulp, and recycled pulp, and examine their characteristics, strengths, and weaknesses.

Through these tests, we will gain a better understanding of the properties of pulp and how they can be leveraged to create different products. We will also explore how pulp can be processed to create different forms, such as paper, cardboard, and packaging materials.

Next, we will turn our attention to the rekon test. This test will help us to assess the performance of our chosen communication platform, HWPE. We will examine the features and capabilities of HWPE and evaluate its effectiveness in meeting our needs.

Once we have completed our study of HWPE, we will propose integration solutions that will allow us to integrate the platform with our existing systems. We will evaluate these solutions based on their effectiveness, feasibility, and cost, and select the one that best meets our requirements.

In summary, this chapter will be a comprehensive exploration of the world of pulp and its many derivatives. We will carry out tests, study communication platforms, propose integration solutions, and ultimately make informed choices that will enable us to achieve our goals. We are excited to embark on this journey and look forward to the insights and knowledge we will gain along the way.

3.1 PULP Simulation

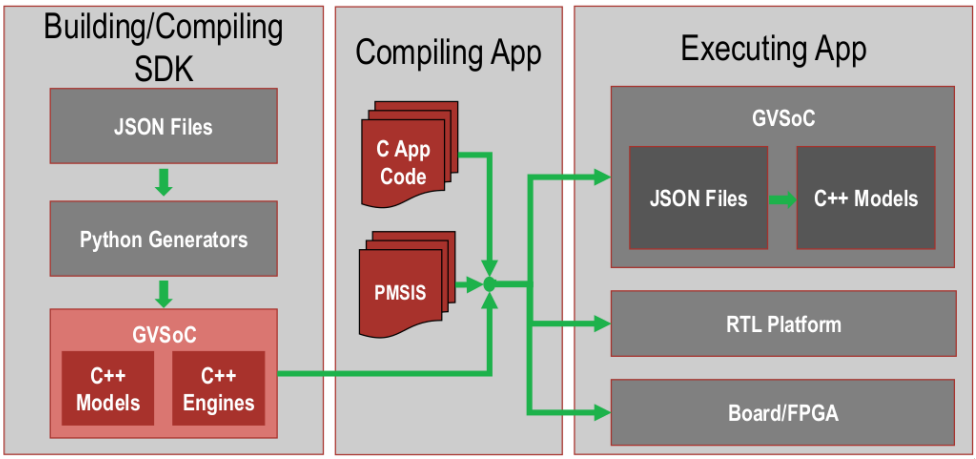

Figure 3.1: PULP Software Environment [8]

The next step it is done to familiarize with the pulp platform using a virtual board simulator that allows you to perform accurate simulations of the behavior that we will expect. This operation is achieved through a SDK that pulp has constituted based on several SDK platform, the most important are the one developed entirely by Pulp,one by GreenWaves-Technologies as GVSoC, and others . i will guide the reader through all the steps First, before proceeding, I will report the specifications of linux that PULP's we have used: Ubuntu 20.04 LTS.

```
Distributor ID: Ubuntu
Description:  Ubuntu 18.04.5 LTS
Release:  18.04
Codename: bionic
```

in fig3.1 it is possible to see the software structure of pulp SDK,at the end of all this step it is possible to see our target that we are going to activate : GVSoc.

installation

This process it is really long if you follow the official guide, be aware that the documentation on PULP-SDK it is written really bad and, if you want to try over there will appeared a lot errors due G++ and Gcc problems. At first time i've installed thought the use of the official guide in ubuntu 16.04 , 18.04 and finally In 20.04 but in any case, once i've tried to install everything correctly step by step, there are some internal compiler error as depicted here below:

```
  CXX src/trace/lxt2.cpp
g++: internal compiler error: Killed (program cc1plus)
Please submit a full bug report,
with preprocessed source if appropriate.
See <file:///usr/share/doc/gcc-7/README.Bugs> for instructions.
Makefile:59: recipe for target '/home/pulp/opt/riscv/pulp-sdk/build/gvsoc/
    engine/vp.o' failed
make[2]: *** [/home/pulp/opt/riscv/pulp-sdk/build/gvsoc/engine/vp.o] Error 4
make[2]: *** Waiting for unfinished jobs....
make[2]: Leaving directory '/home/pulp/opt/riscv/pulp-sdk/tools/gvsoc/common/
    engine'
Makefile:24: recipe for target 'build' failed
make[1]: *** [build] Error 2
make[1]: Leaving directory '/home/pulp/opt/riscv/pulp-sdk/tools/gvsoc/common'
make[1]: Entering directory '/home/pulp/opt/riscv/pulp-sdk/tools/gvsoc/pulp/
    models'
```

```
14 CXX pulp/fll/fll_ctrl_impl.cpp
15 make[1]: *** No rule to make target '/home/pulp/opt/riscv/pulp-sdk/install/
       workstation/lib/libpulpvp.so', needed by '/home/pulp/opt/riscv/pulp-sdk/
       build/gvsoc/models/pulp/fll/fll_ctrl_impl.so'.  Stop.
16 make[1]: Leaving directory '/home/pulp/opt/riscv/pulp-sdk/tools/gvsoc/pulp/
       models'
17 rules/gvsoc.mk:29: recipe for target 'gvsoc.build' failed
18 make: *** [gvsoc.build] Error 2
```

This is done because there are some bug in gpp and gcc dependency. The correct way to setup all the environment is to follow this steps: The first step is to install some package prerequisites for SDK and the toolchain. Among the most important ones appear gtkwave, useful to graph the results of the query to pulp, pip from Python3 ,that we will need to install modules later, and doxygen, that it is a standard tool for generating documentation from annotated C++ sources and other popular programming languages.

```
1 $ sudo apt-get install -y build-essential git libftdi-dev libftdi1
2  doxygen python3-pip libsdl2-dev curl cmake libusb-1.0-0-dev
3  scons gtkwave libsndfile1-dev rsync autoconf automake texinfo libtool
4  pkg-config libsdl2-ttf-dev
5
6 $ sudo apt-get install  autotools-dev curl libmpc-dev libmpfr-dev
7   libgmp-dev gawk build-essential bison flex texinfo gperf  patchutils bc
        zlib1g-dev
```

The SDK also requires the argcomplete, Python's extensible command line for managing bash and shell, and ,and pyelftools a library for parsing and analyzing ELF files and DWARF debugging information

```
1 $ pip install --user argcomplete pyelftools
```

In the next step it is required to install the basic toolchain that have the role to be a RISC-V and C++ cross-compiler supporting generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain compiler. opening a new terminal in the home it is possile to start

```
1   $ pwd
2     /home/jonathan
3   $ git clone https://github.com/pulp-platform/pulp-sdk.git
4   $ wget https://iis-nextcloud.ee.ethz.ch/s/aYESyR5W9FrHgYa/download/riscv-nn-
        toolchain.zip
5   $ unzip riscv-nn-toolchain
```

once the two software are avaliable the next move, that is not neccesary but it is comfortable, it is To rename the two folder one as pulp-sdk and the riscv-nn-toolchain as pulp-riscv-gnu-toolchain. now we are ready to run our Hello word. **IT IS IMPORTANT** That in the next environmental context that the latest directory point in the folder that there is the BIN directory of the pulp-riscv-gnu-toolchain.
rembeber also the initial backslach at the beggining,in my case like that.
also important is that in sudo mode all the previous settings are lost, so you have to redone.

```
1   $ pwd
2     /home/jonathan
3   $ cd pulp-sdk
4   $ export PULP_RISCV_GCC_TOOLCHAIN=/home/jonathan/pulp-riscv-gnu-toolchain
5   $ source configs/pulp-open.sh
6   $ make all
7    [...]
8    make[2]: uscita dalla directory <</home/jonathan/pulp-sdk/tools/gvsoc/
         common/models>>
9    make -C models build ARCHI_DIR= -j 4
10   make[2]: ingresso nella directory <</home/jonathan/pulp-sdk/tools/gvsoc/
         common/models>>
11   find /home/jonathan/pulp-sdk/install/workstation/python -type d -exec touch
         {}/__init__.py \;
```

```
12    make[2]: uscita dalla directory <</home/jonathan/pulp-sdk/tools/gvsoc/
          common/models>>
13    make[1]: uscita dalla directory <</home/jonathan/pulp-sdk/tools/gvsoc/
          common>>
14    make[1]: ingresso nella directory <</home/jonathan/pulp-sdk/tools/gvsoc/
          pulp/models>>
15    find /home/jonathan/pulp-sdk/install/workstation/python -type d -exec touch
           {}/__init__.py \;
16    make[1]: uscita dalla directory <</home/jonathan/pulp-sdk/tools/gvsoc/pulp/
          models>>
```

if the compilation was successful with the make log as in the upper code then you can proceed ahead in this way.
Next let's invoke the hello world :

```
1   $ pwd
2     /home/jonathan/pulp-sdk
3   $ cd tests/hello/
4   $ make clean all run
```

once you send the last command the output will appear:

```
1   RM   /home/jonathan/pulp-sdk/tests/hello/BUILD/PULP/GCC_RISCV/
2   CC   test.c
3   CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/fll-v1.c
4   CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/freq-domains.c
5   CC   /home/jonathan/pulp-sdk/rtos/pulpos/pulp/kernel/chips/pulp/soc.c
6   CC   /home/jonathan/pulp-sdk/rtos/pmsis/pmsis_bsp/fs/read_fs/read_fs.c
7   CC   /home/jonathan/pulp-sdk/rtos/pmsis/pmsis_bsp/fs/host_fs/semihost.c
8   CC   /home/jonathan/pulp-sdk/rtos/pmsis/pmsis_bsp/fs/host_fs/host_fs.c
9   CC   /home/jonathan/pulp-sdk/rtos/pmsis/pmsis_bsp/fs/fs.c
10  CC   /home/jonathan/pulp-sdk/rtos/pmsis/pmsis_bsp/bsp/pulp.c
11  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/lib/libc/minimal/io.c
12  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/lib/libc/minimal/fprintf.c
13  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/lib/libc/minimal/prf.c
14  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/lib/libc/minimal/sprintf.c
15  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/lib/libc/minimal/semiho.c
16  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/init.c
17  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/kernel.c
18  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/device.c
19  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/task.c
20  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/alloc.c
21  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/alloc_pool.c
22  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/irq.c
23  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/soc_event.c
24  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/log.c
25  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/time.c
26  CC   /home/jonathan/pulp-sdk/rtos/pulpos/pulp/drivers/uart/uart-v1.c
27  CC   /home/jonathan/pulp-sdk/rtos/pulpos/pulp/drivers/udma/udma-v3.c
28  CC   /home/jonathan/pulp-sdk/rtos/pulpos/pulp/drivers/cluster/cluster.c
29  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/crt0.S
30  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/irq_asm.S
31  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/task_asm.S
32  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/time_asm.S
33  CC   /home/jonathan/pulp-sdk/rtos/pulpos/common/kernel/soc_event_v2_itc.S
34  CC   /home/jonathan/pulp-sdk/rtos/pulpos/pulp/drivers/cluster/pe-eu-v3.S
35  LD   /home/jonathan/pulp-sdk/tests/hello/BUILD/PULP/GCC_RISCV//test/test
36  gapy --target=pulp --platform=gvsoc --work-dir=/home/jonathan/pulp-sdk/tests
        /hello/BUILD/PULP/GCC_RISCV/ --config-opt=cluster/nb_pe=8     run --image
         --binary=/home/jonathan/pulp-sdk/tests/hello/BUILD/PULP/GCC_RISCV//test
        /test
37  gapy --target=pulp --platform=gvsoc --work-dir=/home/jonathan/pulp-sdk/tests
        /hello/BUILD/PULP/GCC_RISCV/ --config-opt=cluster/nb_pe=8     run --flash
         --binary=/home/jonathan/pulp-sdk/tests/hello/BUILD/PULP/GCC_RISCV//test
        /test
38  gapy --target=pulp --platform=gvsoc --work-dir=/home/jonathan/pulp-sdk/tests
        /hello/BUILD/PULP/GCC_RISCV/ --config-opt=cluster/nb_pe=8     run --exec-
```

```
        prepare --exec --binary=/home/jonathan/pulp-sdk/tests/hello/BUILD/PULP/
        GCC_RISCV//test/test
39  Hello from FC
```

I voluntarily included the log precisely because from the logs you can see all the internal systems of our pulp-open with their interfaces and internal components, the final result will be Hello from FC

Hello world cluster

now that we have the demonstration that it works the next step is to call the pulp cluster,

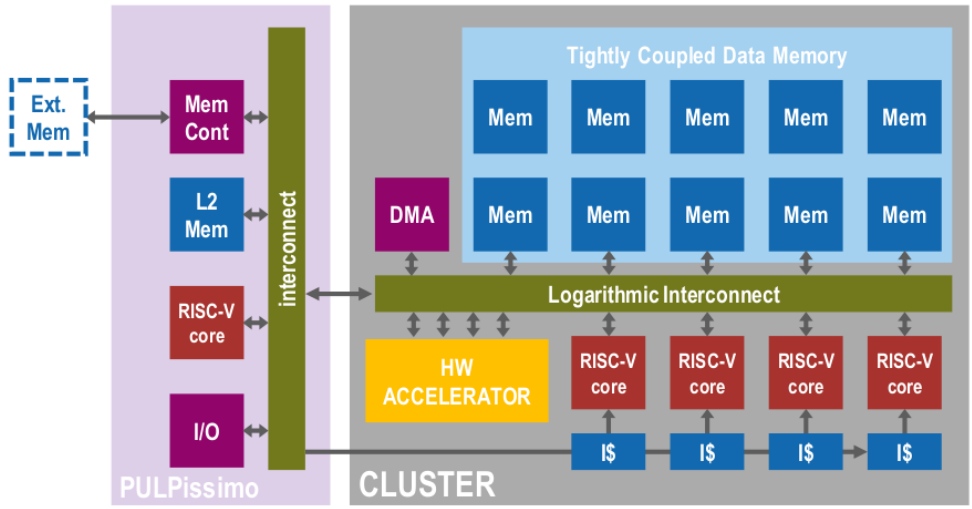

Figure 3.2: Pulp cluster Architecture [9]

```
1 cd /home/jonathan/pulp-sdk/tests/hello/
2 make clean all run
```

Now that the project has been compiled we can check what it does the previous process was based on the same architecture in the figure below only with pulpissimo as a reference. in the next step we are going to call the cluster with this command. this command re-specifies the same considerations made for pulp hello, so just send the command in sucession to the previous section here below the code of the hello word cluster:

```
1  /*
2   * Copyright (C) 2017 ETH Zurich, University of Bologna and GreenWaves
        Technologies
3   * All rights reserved.
4   *
5   * This software may be modified and distributed under the terms
6   * of the BSD license.  See the LICENSE file for details.
7   *
8   * Authors: Germain Haugou, ETH (germain.haugou@iis.ee.ethz.ch)
9   */
10
11 #include "pmsis.h"
12
13
14 #if defined(CLUSTER)
15 void pe_entry(void *arg)
16 {
17     printf("Hello from cluster_id: %d, core_id: %d\n", pi_cluster_id(),
          pi_core_id());
18 }
19
```

```
20  void cluster_entry(void *arg)
21  {
22      pi_cl_team_fork((NUM_CORES), pe_entry, 0);
23  }
24  #endif
25
26  static int test_entry()
27  {
28  #if defined(CLUSTER)
29      struct pi_device cluster_dev;
30      struct pi_cluster_conf cl_conf;
31      struct pi_cluster_task cl_task;
32
33      pi_cluster_conf_init(&cl_conf);
34      pi_open_from_conf(&cluster_dev, &cl_conf);
35      if (pi_cluster_open(&cluster_dev))
36      {
37          return -1;
38      }
39
40      pi_cluster_send_task_to_cl(&cluster_dev, pi_cluster_task(&cl_task,
            cluster_entry, NULL));
41
42      pi_cluster_close(&cluster_dev);
43  #endif
44  #if !defined(CLUSTER)
45      printf("Hello from FC\n");
46  #endif
47
48      return 0;
49  }
50
51
52
53  int main()
54  {
55      return pmsis_kickoff((void *)test_kickoff);
56  }
```

The first include it is used for include all the runtime basic functions and libraries and architecture definition and other parameter that will be usefoul for set pulp correctly, this file it is in ./rtos/pulpos/common/ folder of Pulp-sdk. Since this is made for cluster architecture the ETH have insert an if definition that will be activated only if the target is an cluster. this allows to query to each riscv element to process in parallel way thanks the callable function void pe_entry that will print the core_id and in which cluster this is called, the rest of the Code it is used to set up all the parameters. For compile and run this instruction we need to declare the number of cluster and the relative cores. In this example we are going to use only 1 cluster with 8 RISCV processors, and we're doing so if we launch this command:

```
1  make clean all run USE_CLUSTER=1 NUM_CORES=8
```

and the relative output is printed below:

```
1  hello from cluster_id:0, core_id:2
2  hello from cluster_id:0, core_id:0
3  hello from cluster_id:0, core_id:3
4  hello from cluster_id:0, core_id:4
5  hello from cluster_id:0, core_id:6
6  hello from cluster_id:0, core_id:1
7  hello from cluster_id:0, core_id:5
8  hello from cluster_id:0, core_id:7
```

Now with the following command it is possible to estract the dissasembled in a .s file

```
1  $ make dis > test.s
```

Multiplication

The next operation is to try a simple vector multiplication and see in how many clock cycle the PUlp-open will complete the task.
Here below the structure.

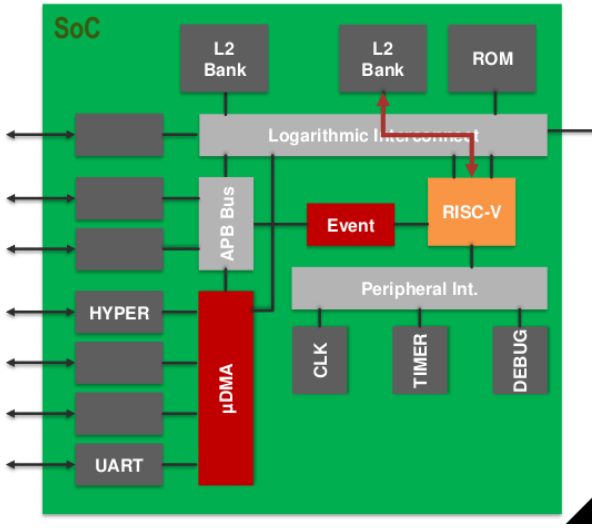

Figure 3.3: Pulp Vector multiplication [9]

Here below i will plot the code for vectors multiplication,

```
#include "pmsis.h"
#include "stats.h"
#define N 128

// global variables
int A[N];
int B[N];
int tempC[N];

// matrix functions
void task_initMat()
{
    for(int i=0;i<N;i++){
        A[i] = i;
        B[i] = i;
    }
}

void task_VectProdScalar(int scalarA, int* matB, int * matC, int dim)
{
    for(int i=0;i<dim;i++){
        matC[i] = scalarA * matB[i];
    }
}

void print_matrix(int * mat, int dim)
{
    for(int i=0;i<dim;i++){
        for(int j=0;j<dim;j++){
            printf("%02d ", mat[i*dim+j]);
        }
        printf("\n");
    }
}

int main()
{
```

```
38      // initialize matrix operands
39      task_initMat();
40
41 #ifdef PRINT_MATRIX
42      printf("\n\nThis is the Matrix A\n");
43      print_matrix(A, N);
44      printf("\n\nThis is the Matrix B\n");
45      print_matrix(B, N);
46 #endif
47 #ifndef STATS
48      //initialize performance counters
49      pi_perf_conf(
50          1 << PI_PERF_CYCLES |
51          1 << PI_PERF_INSTR
52      );
53
54      // measure statistics on matrix operations
55      pi_perf_reset();
56      pi_perf_start();
57 #else
58      INIT_STATS();
59
60      PRE_START_STATS();
61      START_STATS();
62 #endif
63
64      for(int i=0; i<N;i++){
65          task_VectProdScalar(A[i], B, tempC, N);
66      }
67
68 #ifndef STATS
69      pi_perf_stop();
70      uint32_t instr_cnt = pi_perf_read(PI_PERF_INSTR);
71      uint32_t cycles_cnt = pi_perf_read(PI_PERF_CYCLES);
72
73      printf("Number of Instructions: %d\nClock Cycles: %d\nCPI: %f%f\n",
74          instr_cnt, cycles_cnt, (float) cycles_cnt/instr_cnt);
75 #else
76      STOP_STATS();
77 #endif
78 }
```

In this code it is possible to see the simple vector application. on the first part there is the filling of the vector with some number with the init function as:

```
1   for(int i=0;i<N;i++){
2     A[i] = i;
3     B[i] = i;
4     }
```

then we will start to multiply with this operation:

```
1     for(int i=0;i<dim;i++){
2         matC[i] = scalarA * matB[i];
3     }
```

once we build we can see that in pulp open a total of 49812 instuction it is performed in 82582 for a 128 to 128 vector multiplication with a Clock Cycles per instruction by to 2.

```
1   Number of Instructions: 49812
2   Clock Cycles: 82582
3   CPI: 1.6578740.000000
```

DORY

Now that we have seen how these processors are used the next step is to perform a more complex test. What we are going to develop is a part of a certain memory allocation path

for vision applications. The aim is to point to a three memory hierarchy for tiling of weights and activations of nodes The operation of DORY is organized in three steps, performed offline before network deployment of NEMO. First, the ONNX receives as input a QNN graph using the Open Neural Network Exchange (ONNX format).
Then as second step the layer run a tiling loop with data movement.
Third the network parser merges information from the whole network to memory buffer.

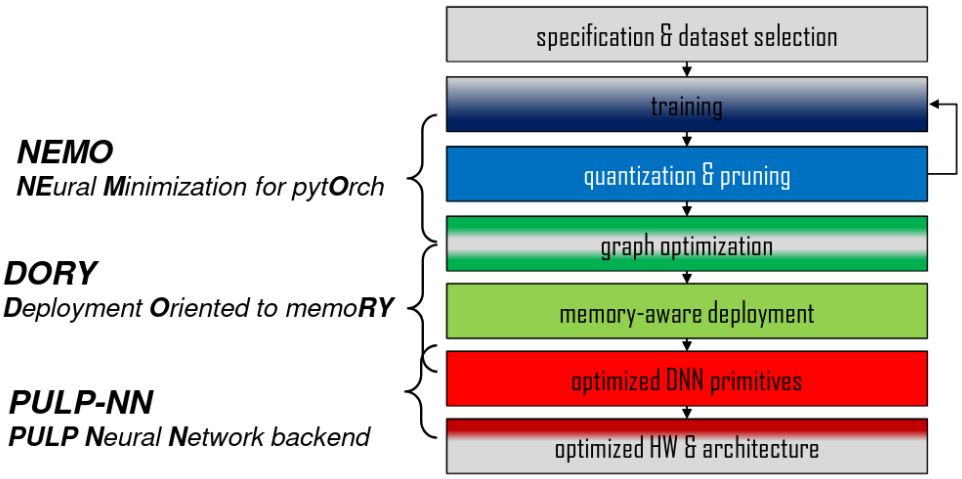

Figure 3.4: PULP Dory flow [10]

As is possible to see on figure the process is divided in there stages:

- NEMO: used for build neural network from a specific dataset (as images)
- DORY: used to deploy the DNN data in memory bank
- PULP-NN: the implementation in sw-hw

in this demonstration we are going to reading of the ONNX output that comes from NEMO process,perform a Layer-by-Layer tiling,Layer template compilation and at least Network compilation.

ONNX Decode and control

The first operation performed by DORY is decoding the input from Nemo that are a an already quantized DNN.
After the data it is fetched Dory process perform a reallocation in a set of layers. Layer it is defined as a sequence of operations performed by distinct ONNX graph nodes. Every layer contain a Linear pooling operation, optional Batch-Normalization operation and a Quantization/Activation operation, all of this layer use quantized inputs, outputs, and weight for processing data.
After a convolutional operation there are some Graph parsing operation with standard operation. After the convolutional node it is established it is analyzing and, if it is recognized as batch mode it is merged back on the conv. step otherwise it will append as new child. After this first step we will generate list of propriety that each node have:

- Layer name
- Convolutional and linear parameters
- Linear Batchnorm
- Status of operation performed
- Network topology parameters

Layer tilling

The next step it is to look inside each node that was described before and generate a memory requirement. if the memory space it is enough it do not perform the tilling, if is not enough but handable it is performed tilling and if it is too big it will send on external memory. after the data it is minimized

load in hardware

The next step is the implementation, as before in the SDK we have this example of Dory. Once everything it is set as before we can call the SDK function as this

```
  $ cd pulp-sdk
  $ export PULP_RISCV_GCC_TOOLCHAIN=/home/jonathan/pulp-riscv-gnu-toolchain
  $ source configs/pulp-open.sh
  $ make all
  $ cd MobileNetV1/
  $ make clean all run
  L3 Buffer alloc initial @ 3388608:  Ok

L3 Buffer alloc initial @ 2388608:  Ok

L3 Buffer alloc initial @ 1388608:  Ok
Layer 0  : Checksum = 120996      , FLASH 120996      , Check OK
Layer 1  : Checksum = 53504       , FLASH 53504       , Check OK
Layer 2  : Checksum = 303730      , FLASH 303730      , Check OK
Layer 3  : Checksum = 117611      , FLASH 117611      , Check OK
Layer 4  : Checksum = 1103930     , FLASH 1103930     , Check OK
Layer 5  : Checksum = 234609      , FLASH 234609      , Check OK
Layer 6  : Checksum = 2236755     , FLASH 2236755     , Check OK
Layer 7  : Checksum = 204746      , FLASH 204746      , Check OK
Layer 8  : Checksum = 4266224     , FLASH 4266224     , Check OK
Layer 9  : Checksum = 524077      , FLASH 524077      , Check OK
Layer 10 : Checksum = 8552435     , FLASH 8552435     , Check OK
Layer 11 : Checksum = 451595      , FLASH 451595      , Check OK
Layer 12 : Checksum = 17038175    , FLASH 17038175    , Check OK
Layer 13 : Checksum = 1057921     , FLASH 1057921     , Check OK
Layer 14 : Checksum = 33824230    , FLASH 33824230    , Check OK
Layer 15 : Checksum = 986273      , FLASH 986273      , Check OK
Layer 16 : Checksum = 34152523    , FLASH 34152523    , Check OK
Layer 17 : Checksum = 945904      , FLASH 945904      , Check OK
Layer 18 : Checksum = 28531734    , FLASH 28531734    , Check OK
Layer 19 : Checksum = 981254      , FLASH 981254      , Check OK
Layer 20 : Checksum = 27547438    , FLASH 27547438    , Check OK
Layer 21 : Checksum = 950986      , FLASH 950986      , Check OK
Layer 22 : Checksum = 27160429    , FLASH 27160429    , Check OK
Layer 23 : Checksum = 899062      , FLASH 899062      , Check OK
Layer 24 : Checksum = 70825648    , FLASH 70825648    , Check OK
Layer 25 : Checksum = 2086584     , FLASH 2086584     , Check OK
Layer 26 : Checksum = 98869725    , FLASH 98869725    , Check OK
Layer 28 : Checksum = 130510192   , FLASH 130510192   , Check OK

L2 Buffer allocated initial @ 0x1c021c38: Ok
L1 Buffer allocated initial @ 0x10006420: Ok

Checksum in/out Layer : Ok
Layer 0 ended
Checksum in/out Layer : Ok
Checksum in/out Layer : Ok
[...]
[0] : num_cycles: 123309034
[0] : MACs: 186400768
[0] : MAC/cycle: 1.511655
[0] : n. of Cores: 1
```

This result it is applied to different buffer information that it is contained in 28 layer. Each layer it is similar to others but what change,of course, is the data but the template

produced by pytorch it is the same.
Here i will report only the layer 0 as example, it is possible to see all the parameter that was extracted.

```
1   sdk                          pulp_sdk
2   dma_parallelization          8-cores
3   optional_type                8bit
4   func_name                    layerConvBNRelu0
5   flag_DW                      0
6   optional                     conv
7   FLAG_BATCHNORM               1
8   has_bias                     0
9   FLAG_RELU                    1
10  test_location                L3
11  type                         char
12  nof                          32
13  factor                       1
14  g                            1
15  nif                          3
16  conv_overlap1                1
17  conv_overlap2                1
18  padding_top                  1
19  padding_bottom               1
20  padding_left                 1
21  padding_right                1
22  stride                       2
23  x_h                          128
24  x_w                          128
25  x_data_size_byte             8
26  x_tile_size_nif              3
27  x_tile_size_h                97
28  x_tile_size_w                17
29  x_tile_size_byte             4947
30  x_tile_size_nif_byte         3
31  x_stride_w_byte              384
32  x_stride_c_byte              3
33  y_h                          64
34  y_w                          64
35  y_data_size_byte             8
36  act_dim_bit                  32
37  y_tile_size_nof              32
38  y_tile_size_h                48
39  y_tile_size_w                8
40  y_tile_size_byte             12288
41  y_stride_w_byte              2048
42  y_stride_c_byte              32
43  y_tile_size_nof_byte         32
44  tile_dim_h                   2
45  tile_dim_w                   8
46  tile_dim_nof                 1
47  tile_dim_nif                 1
48  fs1                          3
49  fs2                          3
50  W_data_size_byte             8
51  W_tile_size_nof              32
52  b_size_byte                  32
53  W_tile_size_nif              3
54  W_tile_size_byte             864
55  W_stride_nof_byte            27
56  W_stride_hw_byte             3
57  W_tile_nif_byte              3
58  l2_off_k                     864
59  l2_off_lambda                992
60  k_tile_size_byte             256
61  lambda_tile_size_byte        256
62  k_size_byte                  128
63  lambda_size_byte             128
64  k_tile_size_byte_transfer    128
```

```
65   lambda_tile_size_byte_transfer 128
66   l1_x_offset                    0
67   l1_y_offset                    9898
68   l1_W_offset                    34478
69   l1_k_offset                    36210
70   l1_lambda_offset               36470
71   x_tile_size_nif_last           3
72   x_tile_size_nif_byte_last      3
73   x_tile_size_h_last             33
74   x_tile_size_w_last             17
75   W_tile_size_nof_last           32
76   W_tile_size_nif_last           3
77   W_tile_size_nif_byte_last      3
78   y_tile_size_nof_last           32
79   y_tile_size_h_last             16
80   y_tile_size_w_last             8
81   y_length_nof_byte_last         32
```

cheat sheet

```
1  $ cd pulp-sdk
2  $ export PULP_RISCV_GCC_TOOLCHAIN=/home/jonathan/pulp-riscv-gnu-toolchain
3  $ source configs/pulp-open.sh
4  $ make clean all run
```

3.2 Reckon implementation

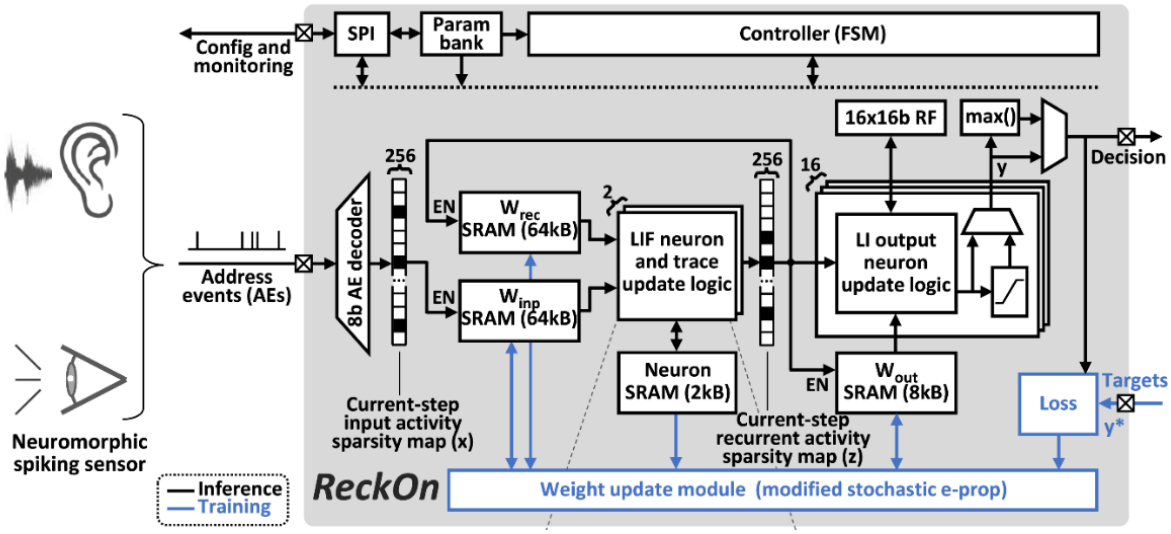

Figure 3.5: Reckon

Inside this section we are going to discuss the simulation and implementation of the reckon as an standalone core. At the first part we are going to deal with the RTL simulation done with Questasim. After the simulation will go as expected we are going to move In physical implementation to end in the deployment.

Overall view

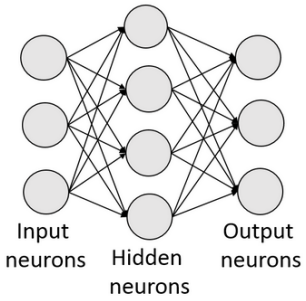

Figure 3.6: SNN

The overall HDL description it is organized in two modules: the first one is the SPI port, that is in charge to connect the outside word with the accelerator for programming target; The second is the neuromorphic core and have the role to be the engine for SNN resolution processor.

The aim of the Reckon is to train and use a SNN to implement a pattern recognition inside hardware. The outside word it is modeled using the AER protocol indeed, the testing's input called interference, are sent through the ReckOn using this protocol while the output are managed as a digital label. The inner structure of this coprocessor is the neural spiking network that is composed of different internal layer that have several fans in and fan out to configure the

correct behavior as could be seen in the figure above. The most important feature is the SPI that have the principal role to be a milestone for configuration proposes.

SPI

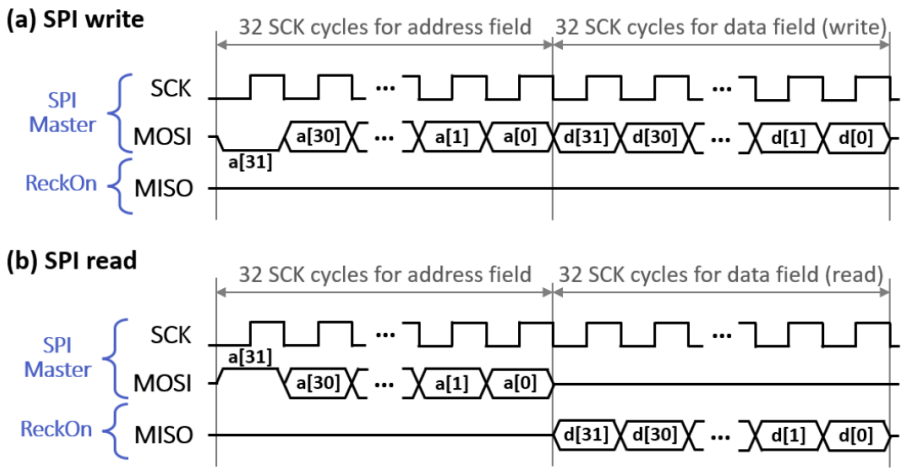

Figure 3.7: 32-bit SPI timing diagram for (a) write and (b) read operations. [9]

Between the many protocols available in the IC world one that has been massively adopted for communication between microcontrollers and external peripherals is SPI (Serial Peripheral Interface).
The SPI protocol is a serial protocol that allows bi-directional synchronous transmission using master and slave channels. This protocol requires a master to control the clock and a slave to validate the correct receiving of a datum .
One of the bigger strength of SPI are the bidirectional way: the master can send or receive information from the slaves, but it will decide who sends and who receives among them.

In Reckon it is possible to see that this architecture it is need to configure the inner registers on the SRAM on chip to update correctly datum and weight for the network.
The slave configuration, based on 32 bit, it works in a precise sequence: in the first stage the Master send through MoSi the address that he want to write and, after the address phase , he will send the data on the same wire of 32 bit wise. For reading the procedure is the same but the output of the address request it is sent on the MiSo wire. In the testing file the sending process it is built inside a sending task where it is possible to see that working method:

```
1   task automatic spi_send (
2         input  logic [31:0] addr,
3         input  logic [31:0] data,
4         input  logic        MISO, // not used
5         ref    logic        MOSI,
6         ref    logic        SCK
7     );
8         integer i;
9
10        for (i=0; i<32; i=i+1) begin
11            MOSI = addr[31-i];
12            wait_ns(`SCK_HALF_PERIOD);
13            SCK  = 1'b1;
14            wait_ns(`SCK_HALF_PERIOD);
15            SCK  = 1'b0;
16        end
17        for (i=0; i<32; i=i+1) begin
18            MOSI = data[31-i];
19            wait_ns(`SCK_HALF_PERIOD);
20            SCK  = 1'b1;
21            wait_ns(`SCK_HALF_PERIOD);
22            SCK  = 1'b0;
23        end
24    endtask
```

If we wan to take as example that we want to perform an write operation in the address 0.1.0.8 (IPv4 notation), in the testbench it is possible to access to this address using the previews task:

```
1 \\ with the value of zero $(SPI\_RST\_MODE = 0x00)$:
2 spi_send (.addr({1'b0,3'b000,12'd1,16'd8  }),
3           .data(`SPI_RST_MODE), .MISO(MISO),
4           .MOSI(MOSI), .SCK(SCK)
5          );
```

programming the parameters

Once the description of the SPI it is closed we can focus now on the next phase: The programming phase of the ReckOn's neural network:

```
1  spi_send (.addr({1'b0,3'b000,12'd1,16'd0  }), .data(1
2  spi_send (.addr({1'b0,3'b000,12'd1,16'd1  }), .data(`SPI_RO_STAGE_SEL
3  spi_send (.addr({1'b0,3'b000,12'd1,16'd2  }), .data(`SPI_GET_CLKINT_OUT
4  spi_send (.addr({1'b0,3'b000,12'd1,16'd8  }), .data(`SPI_RST_MODE
5  spi_send (.addr({1'b0,3'b000,12'd1,16'd9  }), .data(`SPI_DO_EPROP
6  spi_send (.addr({1'b0,3'b000,12'd1,16'd10 }), .data(`SPI_LOCAL_TICK
7  spi_send (.addr({1'b0,3'b000,12'd1,16'd11 }), .data(`SPI_ERROR_HALT
8  spi_send (.addr({1'b0,3'b000,12'd1,16'd12 }), .data(`SPI_FP_LOC_WINP
9  spi_send (.addr({1'b0,3'b000,12'd1,16'd13 }), .data(`SPI_FP_LOC_WREC
10 spi_send (.addr({1'b0,3'b000,12'd1,16'd14 }), .data(`SPI_FP_LOC_WOUT
11 spi_send (.addr({1'b0,3'b000,12'd1,16'd15 }), .data(`SPI_FP_LOC_TINP
12 spi_send (.addr({1'b0,3'b000,12'd1,16'd16 }), .data(`SPI_FP_LOC_TREC
13 spi_send (.addr({1'b0,3'b000,12'd1,16'd17 }), .data(`SPI_FP_LOC_TOUT
14 spi_send (.addr({1'b0,3'b000,12'd1,16'd18 }), .data(`SPI_LEARN_SIG_SCALE
15 spi_send (.addr({1'b0,3'b000,12'd1,16'd19 }), .data(`SPI_REGUL_MODE
16 spi_send (.addr({1'b0,3'b000,12'd1,16'd20 }), .data(`SPI_REGUL_W
```

```
17 spi_send (.addr({1'b0,3'b000,12'd1,16'd21 }), .data(`SPI_EN_STOCH_ROUND
18 spi_send (.addr({1'b0,3'b000,12'd1,16'd22 }), .data(`SPI_SRAM_SPEEDMODE
19 spi_send (.addr({1'b0,3'b000,12'd1,16'd23 }), .data(`SPI_TIMING_MODE
20 spi_send (.addr({1'b0,3'b000,12'd1,16'd25 }), .data(`SPI_REGRESSION
21 spi_send (.addr({1'b0,3'b000,12'd1,16'd26 }), .data(`SPI_SINGLE_LABEL
22 spi_send (.addr({1'b0,3'b000,12'd1,16'd27 }), .data(`SPI_NO_OUT_ACT
23 spi_send (.addr({1'b0,3'b000,12'd1,16'd30 }), .data(`SPI_SEND_PER_TIMESTEP
24 spi_send (.addr({1'b0,3'b000,12'd1,16'd31 }), .data(`SPI_SEND_LABEL_ONLY
25 spi_send (.addr({1'b0,3'b000,12'd1,16'd32 }), .data(`SPI_NOISE_EN
26 spi_send (.addr({1'b0,3'b000,12'd1,16'd33 }), .data(`SPI_FORCE_TRACES
27 spi_send (.addr({1'b0,3'b000,12'd1,16'd64 }), .data(`SPI_CYCLES_PER_TICK
28 spi_send (.addr({1'b0,3'b000,12'd1,16'd69 }), .data(`SPI_KAPPA
29 spi_send (.addr({1'b0,3'b000,12'd1,16'd70 }), .data(`SPI_THR_H_0
30 spi_send (.addr({1'b0,3'b000,12'd1,16'd74 }), .data(`SPI_H_0
31 spi_send (.addr({1'b0,3'b000,12'd1,16'd79 }), .data(`SPI_LR_R_WINP
32 spi_send (.addr({1'b0,3'b000,12'd1,16'd80 }), .data(`SPI_LR_P_WINP
33 spi_send (.addr({1'b0,3'b000,12'd1,16'd81 }), .data(`SPI_LR_R_WREC
34 spi_send (.addr({1'b0,3'b000,12'd1,16'd82 }), .data(`SPI_LR_P_WREC
35 spi_send (.addr({1'b0,3'b000,12'd1,16'd83 }), .data(`SPI_LR_R_WOUT
36 spi_send (.addr({1'b0,3'b000,12'd1,16'd84 }), .data(`SPI_LR_P_WOUT
37 spi_send (.addr({1'b0,3'b000,12'd1,16'd85 }), .data(`SPI_SEED_INP
38 spi_send (.addr({1'b0,3'b000,12'd1,16'd86 }), .data(`SPI_SEED_REC
39 spi_send (.addr({1'b0,3'b000,12'd1,16'd87 }), .data(`SPI_SEED_OUT
40 spi_send (.addr({1'b0,3'b000,12'd1,16'd88 }), .data(`SPI_SEED_STRND_NEUR
41 spi_send (.addr({1'b0,3'b000,12'd1,16'd89 }), .data(`SPI_SEED_STRND_ONEUR
42 spi_send (.addr({1'b0,3'b000,12'd1,16'd90 }), .data(`SPI_SEED_STRND_TINP
43 spi_send (.addr({1'b0,3'b000,12'd1,16'd91 }), .data(`SPI_SEED_STRND_TREC
44 spi_send (.addr({1'b0,3'b000,12'd1,16'd92 }), .data(`SPI_SEED_STRND_TOUT
45 spi_send (.addr({1'b0,3'b000,12'd1,16'd94 }), .data(`SPI_NUM_INP_NEUR
46 spi_send (.addr({1'b0,3'b000,12'd1,16'd95 }), .data(`SPI_NUM_REC_NEUR
47 spi_send (.addr({1'b0,3'b000,12'd1,16'd96 }), .data(`SPI_NUM_OUT_NEUR
48 spi_send (.addr({1'b0,3'b000,12'd1,16'd98 }), .data(`SPI_REGUL_F0
49 spi_send (.addr({1'b0,3'b000,12'd1,16'd99 }), .data(`SPI_REGUL_K_INP_R
50 spi_send (.addr({1'b0,3'b000,12'd1,16'd100}), .data(`SPI_REGUL_K_INP_P
51 spi_send (.addr({1'b0,3'b000,12'd1,16'd101}), .data(`SPI_REGUL_K_REC_R
52 spi_send (.addr({1'b0,3'b000,12'd1,16'd102}), .data(`SPI_REGUL_K_REC_P
53 spi_send (.addr({1'b0,3'b000,12'd1,16'd103}), .data(`SPI_REGUL_K_MUL
54 spi_send (.addr({1'b0,3'b000,12'd1,16'd104}), .data(`SPI_NOISE_STR
```

To be clear here below we're report the documentation explanation what we have done in the last configuration. Reader should be aware that this configuration it is made only for a specific case and for other type of configuration it is need more work on the data parameterization.

- SPI_EN_CONF : Enables access to the network internal state through SPI and ensures the control FSM goes into a safe state to do so, which will be signalled through the SPI_RDY pin.
- SPI_RO_STAGE_SEL : Selects the stage of the ring-oscillator-based local clock generator (not used in the released HDL code as technology-specific blocks, incl. clock gen and frequency divider, were removed).
- SPI_GET_CLKINT_OUT : Enables a frequency-divided copy of the locally generated clock to be displayed on the SPI MISO pin for monitoring purposes (not used in the released HDL code as technology-specific blocks, incl. clock gen and frequency divider, were removed).
- SPI_GET_TAR_REQ_OUT : Enables the target request signal to be displayed on the MISO pin.
- SPI_RST_MODE : Selects the spike reset mode of LIF neurons (1: reset to zero, 0: reset by subtraction).
- SPI_DO_EPROP : Enables e-prop updates (bit 0: input weight updates, bit 1: recurrent weight updates, bit 2: output weight updates). Input/recurrent/output weights can be independently configured in any plastic/frozen configuration.
- SPI_LOCAL_TICK : Enables local generation of timestep ticks (see SPI_CYCLES_PER_TICK for the timestep duration). If configured to 0, timestep ticks are provided externally through the TIME_TICK pin.
- SPI_ERROR_HALT : Enables halting the network operation if a timing error takes place (i.e. a timestep tick occurred before the global FSM finished processing the current timestep) for debugging purposes. A network reset will be necessary.
- SPI_FP_LOC_WINP : Input weight scaling parameter. The stored 8-bit input weights are sign-extended to 16 bits and left-shifted by the value of SPI_FP_LOC_WINP before being added to the neuron membrane potentials.

- SPI_FP_LOC_WREC : Recurrent weight scaling parameter. The stored 8-bit recurrent weights are sign-extended to 16 bits and left-shifted by the value of SPI_FP_LOC_WREC before being added to the neuron membrane potentials.
- SPI_FP_LOC_WOUT : Output weight scaling parameter. The stored 8-bit output weights are sign-extended to 16 bits and left-shifted by the value of SPI_FP_LOC_WOUT before being added to the neuron membrane potentials.
- SPI_FP_LOC_TINP : Radix point location of input traces (left-shifted by the value of SPI_FP_LOC_TINP).
- SPI_FP_LOC_TREC : Radix point location of recurrent traces (left-shifted by the value of SPI_FP_LOC_TREC).
- SPI_FP_LOC_TOUT : Radix point location of output traces (left-shifted by the value of SPI_FP_LOC_TOUT).
- SPI_LEARN_SIG_SCALE : Learning signals scaling parameter, which are left-shifted by the value of SPI_LEARN_SIG_SCALE.
- SPI_REGUL_MODE : Selects the weight regularization mode (bit 0: multiplicative regularization, bit 1: additive regularization). If bit 2 is asserted, regularization is enabled during all timesteps, not only when the TARGET_VALID pin is asserted (for use only with additive regularization).
- SPI_REGUL_W : Enables weight regularization (bit 0: input weights, bit 1: recurrent weights). Input/recurrent weights can be independently configured in any regularized/non-regularized configuration.
- SPI_EN_STOCH_ROUND : Enables stochastic rounding in the eligibility traces and neuron membrane potentials.
- SPI_SRAM_SPEEDMODE : Configuration of the SRAM macro speed modes (not used in the released HDL code as technology-specific blocks were removed).
- SPI_REGRESSION : Should be programmed to 1 for regression tasks and 0 for classification tasks.
- SPI_SINGLE_LABEL : Should be programmed to 1 for classification tasks in order to provide the classification label only once per sample, instead of at every timestep.
- SPI_NO_OUT_ACT : Disables the hard-sigmoid non-linearity applied to the membrane potential of output neurons. SPI_SEND_PER_TIMESTEP Enables sending the network output (format conditioned by SPI_SEND_LABEL_ONLY) at every timestep instead of once at the end of the sample. Typically for use in regression tasks.
- SPI_SEND_LABEL_ONLY : Configures the network output contents sent over the output bus (1: winning neuron label, 0: membrane potential values of all enabled output neurons). Typically configured to 1 for classification tasks and 0 for regression tasks.
- SPI_NOISE_EN : Enables the addition of random noise to membrane potential updates of LIF neurons (noise magnitude configured with SPI_NOISE_STR).
- SPI_FORCE_TRACES : Forces eligibility trace computation even if e-prop updates are disabled (for monitoring purposes).
- SPI_CYCLES_PER_TICK :Number of clock cycles per locally generated timestep tick (used only if SPI_LOCAL_TICK is enabled).
- SPI_ALPHA_CONF : Each bit of SPI_ALPHA_CONF selects the 4 MSBs of the 16-bit leakage decay factors alpha associated to every pair of two LIF neurons, which consist of a single-bit integer part and a 15-bit fractional part (1: the integer part bit is 1 and the three MSBs of the fractional part are 3'b000, 0: the integer part bit is 0 and the three MSBs of the fractional part are 3'b111). The 12 LSBs of alpha's for every pair of two LIF neurons are defined in the neuron memory.
- SPI_KAPPA : Defines the value of the 8-bit leakage factor kappa shared among all output LI neurons, which consists of a single-bit integer part and a 7-bit fractional part.
- SPI_THR_H_0 :Defines the membrane potential threshold separating the first and second segments of the straight-through-estimator (STE) function.
- SPI_THR_H_1 : Defines the membrane potential threshold separating the second and third segments of the straight-through-estimator (STE) function.
- SPI_THR_H_2 : Defines the membrane potential threshold separating the third and fourth segments of the straight-through-estimator (STE) function.
- SPI_THR_H_3 : Defines the membrane potential threshold separating the fourth and fifth segments of the straight-through-estimator (STE) function.
- SPI_H_0 : Defines the value of the first segment of the straight-through-estimator (STE) function.
- SPI_H_1 :Defines the value of the second segment of the straight-through-estimator (STE) function.
- SPI_H_2 : Defines the value of the third segment of the straight-through-estimator (STE) function.
- SPI_H_3 : Defines the value of the fourth segment of the straight-through-estimator (STE) function.
- SPI_H_4 : Defines the value of the fifth segment of the straight-through-estimator (STE) function.

- SPI_LR_R_WINP : Input weight update probability scaling parameter (applies a right shift by the value of SPI_LR_R_WINP).
- SPI_LR_P_WINP : Input weight update probability scaling parameter (applies a left shift by the value of SPI_LR_P_WINP).
- SPI_LR_R_WREC : Recurrent weight update probability scaling parameter (applies a right shift by the value of SPI_LR_R_WREC).
- SPI_LR_P_WREC : Recurrent weight update probability scaling parameter (applies a left shift by the value of SPI_LR_P_WREC).
- SPI_LR_R_WOUT : Output weight update probability scaling parameter (applies a right shift by the value of SPI_LR_R_WOUT).
- SPI_LR_P_WOUT : Output weight update probability scaling parameter (applies a left shift by the value of SPI_LR_P_WOUT).
- SPI_SEED_INP : Seed of the unfolded LFSR generating random numbers for stochastic input weight updates.
- SPI_SEED_REC : Seed of the unfolded LFSR generating random numbers for stochastic recurrent weight updates.
- SPI_SEED_OUT : Seed of the unfolded LFSR generating random numbers for stochastic output weight updates.
- SPI_SEED_STRND_NEUR : Seed of the unfolded LFSR generating random numbers for stochastic rounding of the LIF neuron membrane potentials.
- SPI_SEED_STRND_ONEUR : Seed of the unfolded LFSR generating random numbers for stochastic rounding of the output neuron membrane potentials.
- SPI_SEED_STRND_TINP : seed of the unfolded LFSR generating random numbers for stochastic rounding of the input eligibility traces.
- SPI_SEED_STRND_TREC : Seed of the unfolded LFSR generating random numbers for stochastic rounding of the recurrent eligibility traces.
- SPI_SEED_STRND_TOUT : Seed of the unfolded LFSR generating random numbers for stochastic rounding of the output eligibility traces.
- SPI_SEED_NOISE_NEUR : Seed of the unfolded LFSR generating random numbers for the configurable amount of noise added to the LIF neuron membrane potentials.
- SPI_NUM_INP_NEUR : Number of input neurons enabled in the network (should be configured to the target number of neurons -1).
- SPI_NUM_REC_NEUR : Number of recurrent neurons enabled in the network (should be configured to the target number of neurons -1).
- SPI_NUM_OUT_NEUR : Number of output neurons enabled in the network (should be configured to the target number of neurons -1).
- SPI_REGUL_F0 : Value of the post-synaptic recurrent eligibility traces above which regularization on the pre-synaptic weights is turned on.
- SPI_REGUL_K_INP_R : Input weight additive regularization scaling parameter (applies a right shift by the value of SPI_REGUL_K_INP_R).
- SPI_REGUL_K_INP_P : Input weight additive regularization scaling parameter (applies a left shift by the value of SPI_REGUL_K_INP_P).
- SPI_REGUL_K_REC_R : Recurrent weight additive regularization scaling parameter (applies a right shift by the value of SPI_REGUL_K_REC_R).
- SPI_REGUL_K_REC_P : Recurrent weight additive regularization scaling parameter (applies a left shift by the value of SPI_REGUL_K_REC_P).
- SPI_REGUL_K_MUL : Input and recurrent weight multiplicative regularization scaling parameter (applies a right shift by the value of SPI_REGUL_K_MUL).
- SPI_NOISE_STR : Neuron noise scaling parameter. Noise is generated as pseudo-random 16-bit words to be added to the LIF neuron membrane potentials, right-shifted by the value of SPI_NOISE_STR.

After the network is setup and the weights are imposed the next phase is to verify that all the network is set up correctly with this phrase: Done verification of programmed SNN parameters.
When this is done the network it is implemented, and we are going to perform the network testing.

Testing phase and training

As an explanatory example the next section we are going to see the process of testing. The basic idea of what actually we are going to do, it is possible to see in the figure below. The figure is auto explanatory, indeed, when an object is encoded in the axon the SNN will produce a result looking for the training that he has done earlier. The output of this process it is a decision variable, or label, that brings the object determination.

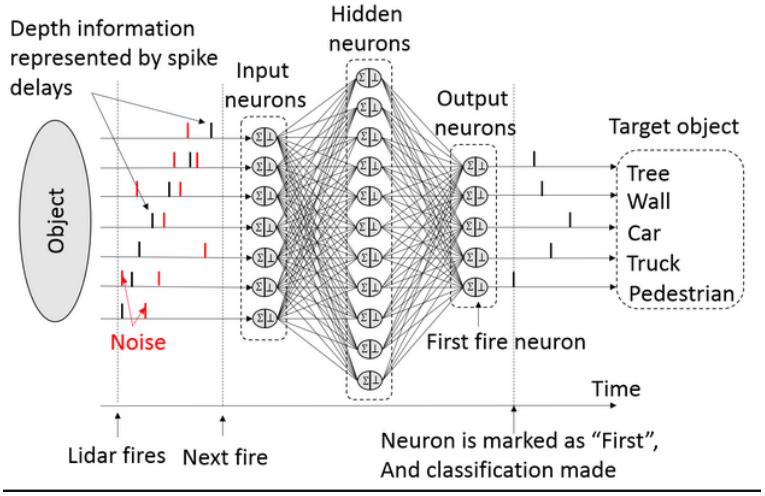

Figure 3.8: SNN example [11]

When the first spike reach the SNN the results are compared with the neuron threshold. If the result is less than the threshold, the neuron will not fire and cumulate next spike's result until the sum of result is larger than the threshold. Once the neuron of subsequent layer fire, that neuron will not receive any spike until the network is reset, and a new input pattern is presented.

Therefore, when the SNN finish a pattern recognition, it maybe just needs to input a few spikes not all spikes. This allows the SNN to have faster responded. Furthermore, We set the classification through the first fire neuron of output layer, which is beneficial to the acceleration of results. So, the trained SNN could process directly the AER input form its port.

FAST DELAYED-SUPERVISION NAVIGATION E-PROP BENCHMARKING

After the SNN it is set up, now we have a brand-new system ready to work. Now we will start the training phase.
This operational part it is performed with several loops that are in charge to call a specific dataset to train and configuring the overlay architecture.
After the set-up phase we are starting the delayed-supervision navigation benchmarking (e-prop enabled on random weights) opening the input weights with the dataset provided (path_to_init_inp_weights.dat).

```
1  fd = $fopen(path_to_init_inp_weights,"r");
2  fork begin
3      for (i=0; i<40; i=i+1) begin
4          spi_half_w(.data({1'b0,3'h3,12'd25,{2'b0,i[7:0],6'd0}}), .MISO(MISO)
               , .MOSI(MOSI), .SCK(SCK));
5          // Header of a multi-write transaction starting at postsynaptic
               address
6          //0 of pre-synaptic neuron i (25 32-bit writes = 100 post-synaptic
               weights)
7          for (j=0; j<25; j=j+1) begin
8              prog_val32 = 32'b0;
```

```
9                for (k=0; k<4; k=k+1) begin
10                   status = $fscanf(fd,"%d",weight);
11                   assert(status == 1) else $fatal(0, "A problem occured while
                         processing input weights file.");
12                   prog_val32 = prog_val32 | ({24'b0,weight[7:0]} << (8*k));
13               end
14               spi_half_w(.data(prog_val32), .MISO(MISO), .MOSI(MOSI), .SCK(SCK
                     ));
15           end
16       end
17   end join
```

The piece of code that I've reported here it is extracted from the phase where reckon are testing and train the snn.
In all the stages the structure remain the same but what it is change is only the dataset that we are going to open as listed below.

- dataset_cueAcc_test.dat: testing the snn
- dataset_cueAcc_train.dat :training snn data
- init_rand_weights_inp_cueAcc.dat :Initializing input weights
- init_rand_weights_out_cueAcc.dat :Initializing output weights:
- init_rand_weights_rec_cueAcc.dat : Initializing recurrent weights

Once one at the time it is called, the next step is to call the training data, and we will display on terminal the log of training that we are processing. For this step it is required a lot of time because the epoch is 10 and the sample that are going to analyze is a lot (near 50). For speed up the process we have reduced the epoch to 2 and the sample to 30.
If no error appear the output will be like something like that:

```
1    ------ Programming SNN parameters...
2  # ------ Starting verification of programmed SNN parametersa
3  #        Initializing neurons (with SPI)...
4  #        Start training for          10 epochs...
5  #        --- Epoch           1:
6  #              Sample  0: inference is  0, label is  0
7  #              Sample  1: inference is  0, label is  1
8  #              Sample  2: inference is  1, label is  1
9  #              Sample  3: inference is  0, label is  0
10 #              Sample  4: inference is  0, label is  0
11 #              Sample  5: inference is  0, label is  1
12 #              Sample  6: inference is  1, label is  1
13 #              Sample  7: inference is  1, label is  0
14 #              Sample  8: inference is  1, label is  1
15 #              Sample  9: inference is  1, label is  1
16 #              Sample 10: inference is  1, label is  0
17 #              Sample 11: inference is  0, label is  0
18 #              Sample 12: inference is  0, label is  0
19 #              Sample 13: inference is  0, label is  0
20 #              Sample 14: inference is  0, label is  0
21 #              Sample 15: inference is  0, label is  0
22 #              Sample 16: inference is  0, label is  0
23 #              Sample 17: inference is  0, label is  1
24 #              Sample 18: inference is  0, label is  1
25 #              Sample 19: inference is  1, label is  1
26 #              Sample 20: inference is  1, label is  1
27 #              Sample 21: inference is  1, label is  1
28 #              Sample 22: inference is  1, label is  0
29 #              Sample 23: inference is  1, label is  0
30 #              Sample 24: inference is  0, label is  0
31 #              Sample 25: inference is  0, label is  1
32 #              Sample 26: inference is  0, label is  0
33 #              Sample 27: inference is  0, label is  1
```

```
34 #               Sample 28: inference is   1, label is   1
35 #               Sample 29: inference is   1, label is   0
36 # Score while training is 21/50 (  42 percents)!
37 #               Sample  0: inference is   1, label is   1
38 #               Sample  1: inference is   1, label is   1
39 #               Sample  2: inference is   0, label is   0
40 #               Sample  3: inference is   0, label is   1
41 #               Sample  4: inference is   1, label is   1
42 #               Sample  5: inference is   1, label is   1
43 #               Sample  6: inference is   0, label is   0
44 #               Sample  7: inference is   0, label is   0
45 #               Sample  8: inference is   0, label is   0
46 #               Sample  9: inference is   0, label is   0
47 #               Sample 10: inference is   1, label is   1
48 #               Sample 11: inference is   1, label is   1
49 #               Sample 12: inference is   0, label is   0
50 #               Sample 13: inference is   0, label is   0
51 #               Sample 14: inference is   0, label is   0
52 #               Sample 15: inference is   0, label is   0
53 #               Sample 16: inference is   0, label is   0
54 #               Sample 17: inference is   0, label is   0
55 #               Sample 18: inference is   0, label is   0
56 #               Sample 19: inference is   0, label is   0
57 #               Sample 20: inference is   1, label is   1
58 #               Sample 21: inference is   1, label is   1
59 #               Sample 22: inference is   0, label is   0
60 #               Sample 23: inference is   0, label is   0
61 #               Sample 24: inference is   0, label is   1
62 #               Sample 25: inference is   1, label is   1
63 #               Sample 26: inference is   0, label is   0
64 #               Sample 27: inference is   1, label is   1
65 #               Sample 28: inference is   1, label is   1
66 #               Sample 29: inference is   0, label is   0
67 #  Score on test set is 28/50 (          56 percents)!
68 # ------ Ending delayed-supervision navigation benchmarking.
69 # ** Note: $finish    : tbench.sv(491)
70 #    Time: 1785324992 ns  Iteration: 0  Instance: /tbench
71 # End time: 22:52:28 on Dec 29,2022, Elapsed time: 1:49:46
```

For understanding the meaning of the log it is important to understand that the interference is actually the sample that was sent, and the label is actually the output of the Rekon. If the interference and the label are target in the SNN have processed correctly otherwise the result it is not correct. As it is possible to see the Score on test it will go better every time the sample are greater .

AER

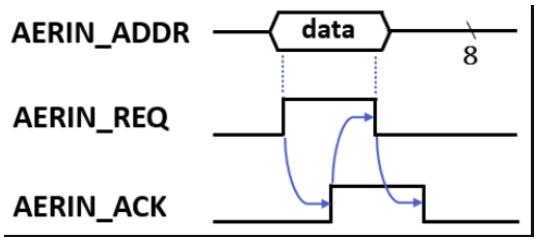

Figure 3.9: Input AER four-phase handshake timing diagram.

Once a lot o data it is sent and the snn it is almost trained the next step is the possibility to send real data through AER.
For each sample that was sent through the recurrent phase there are an AER that simulate the behavior when the Reckon receive data from outside.

The testing source that we are using also have embedded AER sending task as below.

```
[...]
AERIN_TAR_EN = 1'b1;
aer_send(.addr_in(evt_target[7:0]), .addr_out(AERIN_ADDR), .ack(AERIN_ACK), .
    req(AERIN_REQ));
wait_ns(200);
AERIN_TAR_EN = 1'b0;
wait_ns(100);
TARGET_VALID = 1'b1;
[...]
```

So inside a while there are some sending data to see if the label recognizer work properly. As it is possible to see from previews script we are using aer_send:

```
 /*************************
 AER send event
 *************************/

 task automatic aer_send (
     input  logic [`M-1:0] addr_in,
     ref    logic [`M-1:0] addr_out,
     ref    logic          ack,
     ref    logic          req
 );
     while (ack) wait_ns(1);
     addr_out = addr_in;
     wait_ns(1);
     req = 1'b1;
     while (!ack) wait_ns(1);
     wait_ns(1);
     req = 1'b0;
 endtask
```

This task is able to send a AER message in the AER port, it is important to remember that in this steep the input $inputlogic[7:0]addr\_in$ it is the event in which we are going to step the stimuli process.

Module port

As last point we are going to analyze better the port configuration.
This point, even though at first it may be unnecessary, it is crucial because allows the neuromorphic coprocessor configuration to be visualized in plain text.
As can be seen from the code the most important ports are obviously the spi port and also the AER communication port and the fundamental output port called and named with the prefix out

```
 module reckon #(
    parameter N = 256,
    parameter M = 8
)(
    // Global inputs   ------------------------------------
    input  wire          CLK_EXT,
    input  wire          CLK_INT_EN,
    input  wire          RST,

    // SPI slave       ------------------------------------
    input  wire          SCK,
    input  wire          MOSI,
    output wire          MISO,

    // Input bus and control inputs ---------------------
    input  wire [M-1:0] AERIN_ADDR,
    input  wire          AERIN_REQ,
    output wire          AERIN_ACK,
    input  wire          AERIN_TAR_EN,
```

```
20      input  wire          SAMPLE,
21      input  wire          TIME_TICK,
22      input  wire          TARGET_VALID,
23      input  wire          INFER_ACC,
24
25      // Output bus and control outputs ------------------
26      output wire          SPI_RDY,
27      output wire          TIMING_ERROR_RDY,
28      output wire          OUT_REQ,
29      input  wire          OUT_ACK,
30      output wire [  7:0] OUT_DATA
31  );
```

innovus

vivado

For running the vivaldo we used the EDA server and the command to load the modules is *moduleloadtools/xilinx/all* then run the basch vivado. after it is load we need to fix a point in the file spi_slave.v near line 313:

```
1   //SPI_RO_STAGE_SEL - 9 bits - address 1
2   always @(posedge SCK, posedge RST_async)
3   if       (RST_async)  SPI_RO_STAGE_SEL <= 9'd0; // to be add
4   else if      (~|spi_addr[31:28] && (spi_addr[15:0] == 16'd1 ) && &spi_cnt
        [4:0] && |spi_cnt[16:5])      SPI_RO_STAGE_SEL <= {spi_shift_reg_in[7:0],
         MOSI};
```

in that way it is possible to run the synthesis correctly and we can open the elaborated design section after we set up correctly the Board target: XCZU9EG-FFVB1156-1-e with 1156 bidirectional pins and 912 blocks of RAM.

After a while it show up the schematics.

Xlinix deployment

3.3 HWPE

In this section we are going to deal in more detail about HWPE .

Data organization

The test that we are going to do is an standalone SOC environment that is focused on HWPE performing multiply-accumulate on a vector of fixed-point Values. For simulation proposes the structural description it is implemented with a dummy memories and the usage of a zero-riscy core to execute tests written directly in C and other boot assembly script and linker script (link.ld)

installation

here we are going to see how we could install this test on linux distro. It is important to point out that the software Prerequisites it is pointed on Questasim 10.6 and Modelsim it is not supported.

```
1 git clone https://github.com/pulp-platform/hwpe-tb.git
2 cd hwpe-tb
3 make update-ips
4 make build-hw
5 cd hw/sim
6 make
7 vsim
```

The testbench

Since the memory is not syntetizable it is used a dummy memory for simulation proposes. The architecture is straightforward since the communication between the Riscv and the coprocessor it is done using only the TCDM bus interconnection interface namely hwpe_stream_intf_tcdm

```
    logic        req;
    logic        gnt;
    logic [31:0] add;
    logic        wen;
    logic [3:0]  be;
    logic [31:0] data;
    logic [31:0] r_data;
    logic        r_valid;

    modport master (
      output req, add, wen, be, data,
      input  gnt, r_data, r_valid
    );
    modport slave (
      input  req, add, wen, be, data,
      output gnt, r_data, r_valid
    );
    modport monitor (
      input  req, add, wen, be, data, gnt, r_data, r_valid
    );
```

from this interface it is possible to have a slave ,a master and a monitor. For our SOC architectureit is declaredin that way :

```
hwpe_stream_intf_tcdm instr[0:0]  (.clk(clk_i));
hwpe_stream_intf_tcdm stack[0:0]  (.clk(clk_i));
hwpe_stream_intf_tcdm tcdm [MP:0] (.clk(clk_i));

mac_top_wrap #(
    .N_CORES          ( NC ),
    .MP               ( MP ),
    .ID               ( ID )
  ) i_hwpe_top_wrap (
    .clk_i            ( clk_i          ),
    .rst_ni           ( rst_ni         ),
    .test_mode_i      ( 1'b0           ),
    .tcdm_add         ( tcdm_add       ),
    .tcdm_be          ( tcdm_be        ),
    .tcdm_data        ( tcdm_data      ),
    .tcdm_gnt         ( tcdm_gnt       ),
    .tcdm_wen         ( tcdm_wen       ),
    .tcdm_req         ( tcdm_req       ),
    .tcdm_r_data      ( tcdm_r_data    ),
    .tcdm_r_valid     ( tcdm_r_valid   ),
    .periph_add       ( periph_add     ),
    .periph_be        ( periph_be      ),
    .periph_data      ( periph_data    ),
    .periph_gnt       ( periph_gnt     ),
    .periph_wen       ( periph_wen     ),
    .periph_req       ( periph_req     ),
    .periph_id        ( periph_id      ),
    .periph_r_data    ( periph_r_data  ),
    .periph_r_valid   ( periph_r_valid ),
    .periph_r_id      ( periph_r_id    ),
    .evt_o            ( evt            )
  );



  zeroriscy_core #(
    .N_EXT_PERF_COUNTERS ( 0 ),
    .RV32E               ( 0 ),
```

```
39      .RV32M                ( 1 )
40    ) i_zeroriscy (
41      .clk_i                ( clk_i        ),
42      .rst_ni               ( rst_ni       ),
43      .clock_en_i           ( 1'b1         ),
44      .test_en_i            ( 1'b0         ),
45      .core_id_i            ( '0           ),
46      .cluster_id_i         ( '0           ),
47      .boot_addr_i          ( '0           ),
48      .instr_req_o          ( instr_req    ),
49      .instr_gnt_i          ( instr_gnt    ),
50      .instr_rvalid_i       ( instr_rvalid ),
51      .instr_addr_o         ( instr_addr   ),
52      .instr_rdata_i        ( instr_rdata  ),
53      .data_req_o           ( data_req     ),
54      .data_gnt_i           ( data_gnt     ),
55      .data_rvalid_i        ( data_rvalid  ),
56      .data_we_o            ( data_we      ),
57      .data_be_o            ( data_be      ),
58      .data_addr_o          ( data_addr    ),
59      .data_wdata_o         ( data_wdata   ),
60      .data_rdata_i         ( data_rdata   ),
61      .data_err_i           ( data_err     ),
62      .irq_i                ( evt[0][0]    ),
63      .irq_id_i             ( '0           ),
64      .irq_ack_o            (              ),
65      .irq_id_o             (              ),
66      .debug_req_i          ( '0           ),
67      .debug_gnt_o          (              ),
68      .debug_rvalid_o       (              ),
69      .debug_addr_i         ( '0           ),
70      .debug_we_i           ( '0           ),
71      .debug_wdata_i        ( '0           ),
72      .debug_rdata_o        (              ),
73      .debug_halted_o       (              ),
74      .debug_halt_i         ( '0           ),
75      .debug_resume_i       ( '0           ),
76      .fetch_enable_i       ( fetch_enable ),
77      .ext_perf_counters_i  ( '0           )
78    );
79
80
81    tb_dummy_memory #(
82      .MP           ( MP+1          ),
83      .MEMORY_SIZE  ( MEMORY_SIZE   ),
84      .BASE_ADDR    ( BASE_ADDR     ),
85      .PROB_STALL   ( PROB_STALL    ),
86      .TCP          ( TCP           ),
87      .TA           ( TA            ),
88      .TT           ( TT            )
89    ) i_dummy_memory (
90      .clk_i        ( clk_i         ),
91      .randomize_i  ( randomize_mem ),
92      .enable_i     ( enable_mem    ),
93      .stallable_i  ( busy          ),
94      .tcdm         ( tcdm          )
95    );
96
97    tb_dummy_memory #(
98      .MP           ( 1             ),
99      .MEMORY_SIZE  ( MEMORY_SIZE   ),
100     .BASE_ADDR    ( BASE_ADDR     ),
101     .PROB_STALL   ( 0             ),
102     .TCP          ( TCP           ),
103     .TA           ( TA            ),
104     .TT           ( TT            )
105   ) i_dummy_instr_memory (
106     .clk_i        ( clk_i ),
```

```
107     .randomize_i ( 1'b0   ),
108     .enable_i    ( 1'b1   ),
109     .stallable_i ( 1'b0   ),
110     .tcdm        ( instr )
111   );
112
113   tb_dummy_memory #(
114     .MP          ( 1           ),
115     .MEMORY_SIZE ( MEMORY_SIZE ),
116     .BASE_ADDR   ( BASE_ADDR   ),
117     .PROB_STALL  ( 0           ),
118     .TCP         ( TCP         ),
119     .TA          ( TA          ),
120     .TT          ( TT          )
121   ) i_dummy_stack_memory (
122     .clk_i       ( clk_i ),
123     .randomize_i ( 1'b0  ),
124     .enable_i    ( 1'b1  ),
125     .stallable_i ( 1'b0  ),
126     .tcdm        ( stack )
127   );
```

From this structural architecture it is possible to note that there are the most important configuration operand $Test\_mode\_i = 1'b0$),.
This input port have the role to set up the engine between a simple multiplication mode that takes two 32bit streams (vectors) a, b and computes the multiplication : $d = a * b$.
or a scalar product that takes three 32bit fixed-point streams a, b, c and computes the operation as here: $d = dot(a, b) + c$.
input Once the 64 multiplication is done the vectors are separated by an iteration stride and normalized with a configurable shift factor.
The four streams a, b, c, d are connected to four separate ports on the external memory interface.
Indeed inside the top_mac it is possible to see the integration of the HWPE modules of the data bus in and out from the engine :

```
1   hwpe_stream_intf_stream #(.DATA_WIDTH(32)) a (.clk ( clk_i ) ); //input
2   hwpe_stream_intf_stream #(.DATA_WIDTH(32)) b (.clk ( clk_i ) ); //input
3   hwpe_stream_intf_stream #(.DATA_WIDTH(32)) c (.clk ( clk_i ) ); //input
4   hwpe_stream_intf_stream #(.DATA_WIDTH(32)) d (.clk ( clk_i ) ); //output
```

this interface will be called later on the script when we want to connect the real datapath of the engine that it is embedded inside the mac_engine i_engine

```
1   mac_engine i_engine (
2       .clk_i          ( clk_i          ),
3       .rst_ni         ( rst_ni         ),
4       .test_mode_i    ( test_mode_i    ),
5       .a_i            ( a.sink         ),
6       .b_i            ( b.sink         ),
7       .c_i            ( c.sink         ),
8       .d_o            ( d.source       ),
9       .ctrl_i         ( engine_ctrl    ),
10      .flags_o        ( engine_flags   )
11    );
```

Since the testing was done only for the simple multiplication what we need is only the stream a b and d. the figure 3.10 show the simulation of the multiplication.

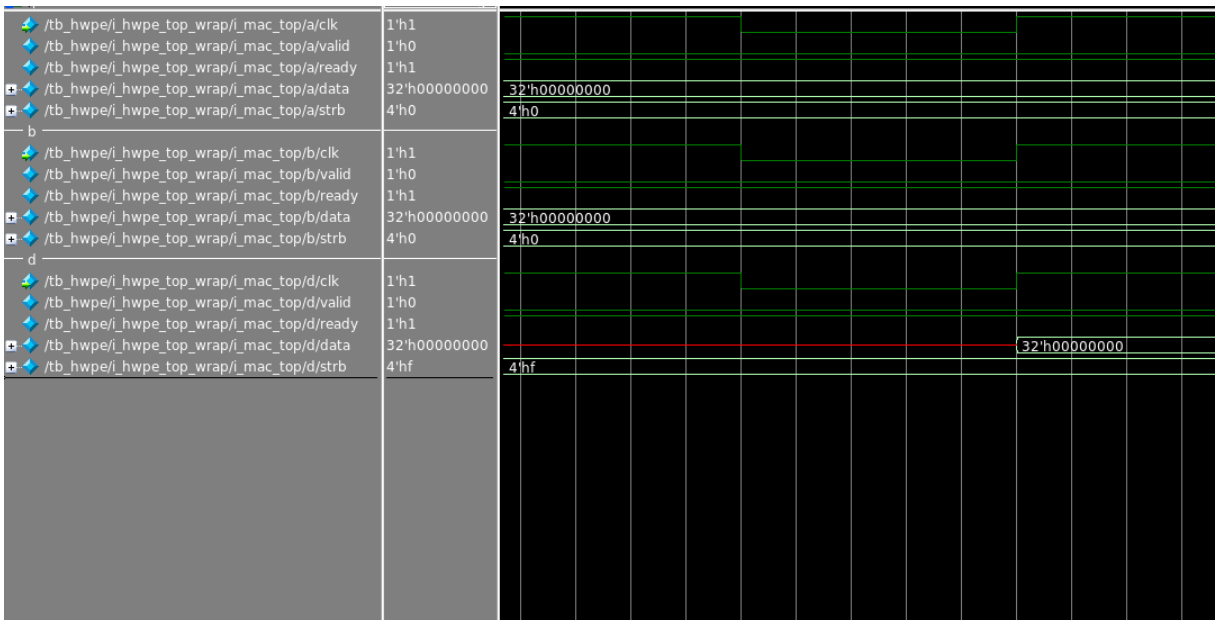

Figure 3.10: hwpe result of testing

then this result will be saved on the memory as possible to see here

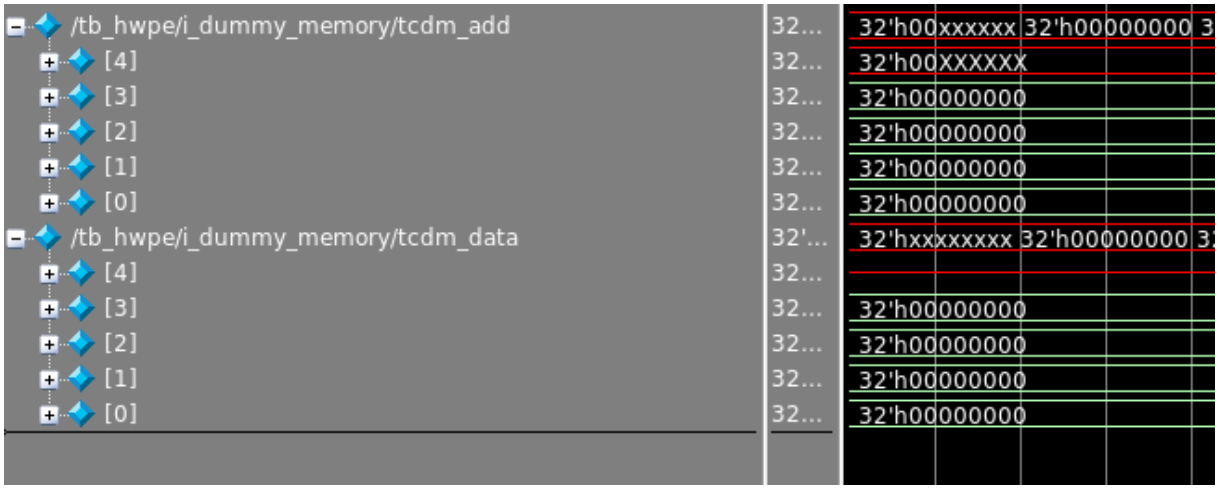

Figure 3.11: HWPE result of testing

First Idea

All the analyses down until now have brought me to create a first implementation. The basic idea was to tailor the modules as an emergency SOC. The idea it is pretty simple based on Frankel result: The coprocessor it is always turned on, when there are a recognition of a certain situation inside the SOC will flow a trigger signal that perform a specific interrupt routine from the zero riscy.
Here below it is possible to see the figure of that idea :

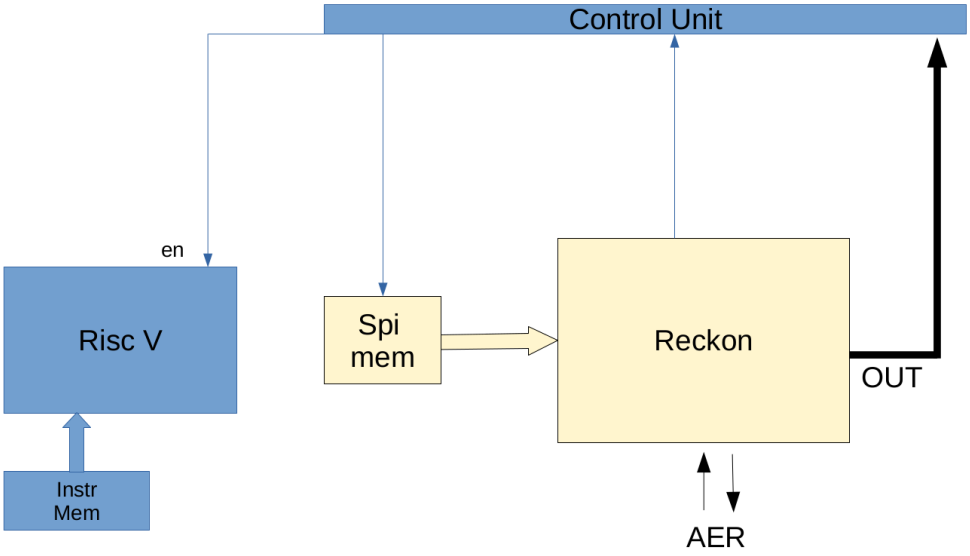

Figure 3.12: uDMA

The warning structure it was deployed but the target of this implementation proposed was centered in the Reckon while the true task is to engage the processor zero riscy.
But for be able to have a landscape of all the possible combination and utilization It will present also this idea, although it was not developed because it is out of target but could be presented for a further implementation for security camera or something more centered on the peripheral and not on the central unit.

Anyway, before going deep below it is reproduced the wave output:

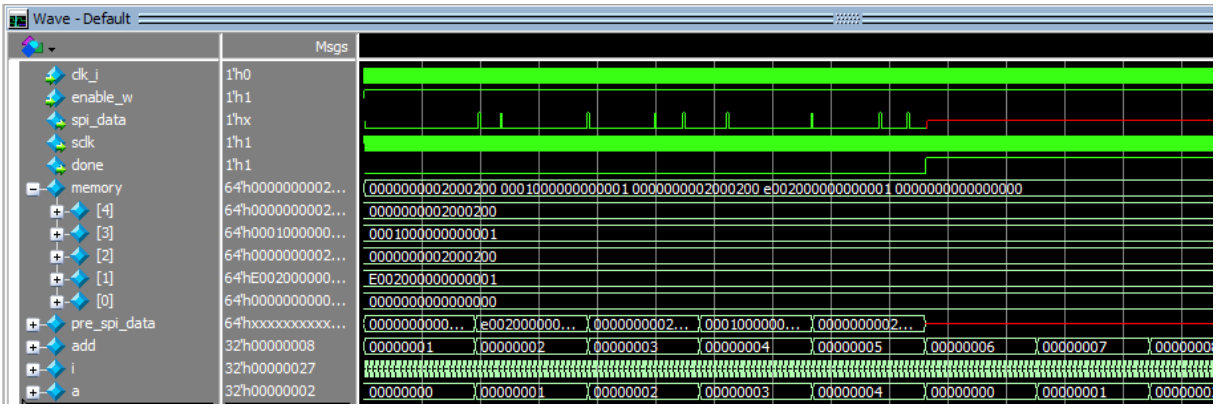

Figure 3.13: wave of security method

The picture shows all the running structure.
For time reason I have sized the Rom SPI mem only with 5 register but could be extended with a more huge sized memory bank. This SPI mem will be sent on the SPI bus on SPI

Data wire that it is attached directly on Reckon's SPI. The memory was design for be able to send on the control unit a signal that inform the system that all the parameters are sent correctly, and it is named as Done. Done have a 0 for default and 1 when we reach the bottom of the RAM.

```
// vsim -voptargs=+acc work.tbench_spi_mem
timeunit 1ns;
timeprecision 1ns;
`define CLK_HALF_PERIOD             10
`define SCK_HALF_PERIOD             50
`define EPOCHS              10
`define SPI_RO_STAGE_SEL            9'b0
`define SPI_GET_CLKINT_OUT          1'b0
`define SPI_RST_MODE                1'b0
`define SPI_DO_EPROP                3'b000
`define SPI_LOCAL_TICK              1'b0
`define SPI_ERROR_HALT              1'b1
`define SPI_FP_LOC_WINP             3'b011
`define SPI_FP_LOC_WREC             3'b011
`define SPI_FP_LOC_WOUT             3'b011
`define SPI_FP_LOC_TINP             3'b011
`define SPI_FP_LOC_TREC             3'b100
`define SPI_FP_LOC_TOUT             3'b110
`define SPI_LEARN_SIG_SCALE         4'b0101
`define SPI_REGUL_MODE              3'b010
`define SPI_REGUL_W                 2'b11
`define SPI_EN_STOCH_ROUND          1'b1
`define SPI_SRAM_SPEEDMODE          8'b00000000
`define SPI_TIMING_MODE             1'b0
`define SPI_REGRESSION              1'b0
`define SPI_SINGLE_LABEL            1'b1
`define SPI_NO_OUT_ACT              1'b0
`define SPI_SEND_PER_TIMESTEP       1'b0
`define SPI_SEND_LABEL_ONLY         1'b1
`define SPI_NOISE_EN                1'b0
`define SPI_FORCE_TRACES            1'b0
`define SPI_CYCLES_PER_TICK         32'b0
`define SPI_ALPHA_CONF              128'h0
`define SPI_KAPPA                   8'h79
`define SPI_THR_H_0                 $signed(16'd205)
`define SPI_THR_H_1                 $signed(16'd205)
`define SPI_THR_H_2                 $signed(16'd205)
`define SPI_THR_H_3                 $signed(16'd205)
`define SPI_H_0                     $signed(5'd0)
`define SPI_H_1                     $signed(5'd0)
`define SPI_H_2                     $signed(5'd0)
`define SPI_H_3                     $signed(5'd0)
`define SPI_H_4                     $signed(5'd1)
`define SPI_LR_R_WINP               5'b0
`define SPI_LR_P_WINP               5'd8
`define SPI_LR_R_WREC               5'b0
`define SPI_LR_P_WREC               5'd8
`define SPI_LR_R_WOUT               5'b0
`define SPI_LR_P_WOUT               5'd14
`define SPI_SEED_INP                25'hF0F0
`define SPI_SEED_REC                25'hF1F1
`define SPI_SEED_OUT                22'hF2F2
`define SPI_SEED_STRND_NEUR         30'h3F3FF3F3
`define SPI_SEED_STRND_ONEUR        15'hF4F4
`define SPI_SEED_STRND_TINP         30'h3F5FF5F5
`define SPI_SEED_STRND_TREC         30'h3F6FF6F6
`define SPI_SEED_STRND_TOUT         30'h3F7FF7F7
`define SPI_SEED_NOISE_NEUR         17'h00FF0
`define SPI_NUM_INP_NEUR            8'd39
`define SPI_NUM_REC_NEUR            8'd99
`define SPI_NUM_OUT_NEUR            4'd1
`define SPI_REGUL_F0                12'd160
```

```
63 `define SPI_REGUL_K_INP_R            5'b0
64 `define SPI_REGUL_K_INP_P            5'd10
65 `define SPI_REGUL_K_REC_R            5'b0
66 `define SPI_REGUL_K_REC_P            5'd10
67 `define SPI_REGUL_K_MUL              5'd0
68 `define SPI_NOISE_STR                4'b0000
69
70
71
72 module tb_dummy_memory
73 #(
74   parameter MEMORY_SIZE ,//1024
75   parameter MEMORY_word = 64,
76   parameter ADD_word = 8
77 )
78 (
79   input  logic                  clk_i,
80   input  logic                  enable_w,
81   //OUTPUT
82   output logic spi_data ,
83   output logic sclk ,
84   output logic done = 1'b0
85   //interface
86   //spi_int.slave
87
88 );
89
90   logic [MEMORY_SIZE-1:0][MEMORY_word-1:0] memory;
91   logic [MEMORY_word-1:0] pre_spi_data;
92   integer add  = 0;
93   int i = 0;
94   int a = 0;
95  initial begin
96   memory[0] <= '0 ;
97   memory[1] <= {1'b1,3'b111,12'd1,16'd0,'0,32'd1};
98   memory[2] <= {1'b0,3'b000,12'd1,16'd1 ,`SPI_RO_STAGE_SEL }; //
       SPI_RO_STAGE_SEL
99   memory[3] <= {1'b0,3'b000,12'd1,16'd0 ,1                                  }; //
       SPI_EN_CONF
100  memory[4] <= {1'b0,3'b000,12'd1,16'd1 ,`SPI_RO_STAGE_SEL                  }; //
       SPI_RO_STAGE_SEL
101  memory[5] <= {1'b0,3'b000,12'd1,16'd2 ,`SPI_GET_CLKINT_OUT                }; //
       SPI_GET_CLKINT_OUT
102  memory[6] <= {1'b0,3'b000,12'd1,16'd8 ,`SPI_RST_MODE                      }; //
       SPI_RST_MODE
103  memory[7] <= {1'b0,3'b000,12'd1,16'd9 ,`SPI_DO_EPROP                      }; //
       SPI_DO_EPROP
104  memory[8] <= {1'b0,3'b000,12'd1,16'd10,`SPI_LOCAL_TICK                    }; //
       SPI_LOCAL_TICK
105  memory[9] <= {1'b0,3'b000,12'd1,16'd11,`SPI_ERROR_HALT                    }; //
       SPI_ERROR_HALT
106  memory[10] <= {1'b0,3'b000,12'd1,16'd12,`SPI_FP_LOC_WINP                  }; //
       SPI_FP_LOC_WINP
107  memory[11] <= {1'b0,3'b000,12'd1,16'd13,`SPI_FP_LOC_WREC                  }; //
       SPI_FP_LOC_WREC
108  memory[12] <= {1'b0,3'b000,12'd1,16'd14,`SPI_FP_LOC_WOUT                  }; //
       SPI_FP_LOC_WOUT
109  memory[13] <= {1'b0,3'b000,12'd1,16'd15,`SPI_FP_LOC_TINP                  }; //
       SPI_FP_LOC_TINP
110  memory[14] <= {1'b0,3'b000,12'd1,16'd16,`SPI_FP_LOC_TREC                  }; //
       SPI_FP_LOC_TREC
111  memory[15] <= {1'b0,3'b000,12'd1,16'd17,`SPI_FP_LOC_TOUT                  }; //
       SPI_FP_LOC_TOUT
112  memory[16] <= {1'b0,3'b000,12'd1,16'd18,`SPI_LEARN_SIG_SCALE              }; //
       SPI_LEARN_SIG_SCALE
113  memory[17] <= {1'b0,3'b000,12'd1,16'd19,`SPI_REGUL_MODE                   }; //
       SPI_REGUL_MODE
114  memory[18] <= {1'b0,3'b000,12'd1,16'd20,`SPI_REGUL_W                      }; //
```

```
        SPI_REGUL_W
115     memory[19] <= {1'b0,3'b000,12'd1,16'd21,`SPI_EN_STOCH_ROUND          }; //
        SPI_EN_STOCH_ROUND
116     memory[20] <= {1'b0,3'b000,12'd1,16'd22,`SPI_SRAM_SPEEDMODE          }; //
        SPI_SRAM_SPEEDMODE
117     memory[21] <= {1'b0,3'b000,12'd1,16'd23,`SPI_TIMING_MODE             }; //
        SPI_TIMING_MODE
118     memory[22] <= {1'b0,3'b000,12'd1,16'd25,`SPI_REGRESSION              }; //
        SPI_REGRESSION
119     memory[23] <= {1'b0,3'b000,12'd1,16'd26,`SPI_SINGLE_LABEL            }; //
        SPI_SINGLE_LABEL
120     memory[24] <= {1'b0,3'b000,12'd1,16'd27,`SPI_NO_OUT_ACT              }; //
        SPI_NO_OUT_ACT
121     memory[25] <= {1'b0,3'b000,12'd1,16'd30,`SPI_SEND_PER_TIMESTEP       }; //
        SPI_SEND_PER_TIMESTEP
122     memory[26] <= {1'b0,3'b000,12'd1,16'd31,`SPI_SEND_LABEL_ONLY         }; //
        SPI_SEND_LABEL_ONLY
123     memory[27] <= {1'b0,3'b000,12'd1,16'd32,`SPI_NOISE_EN                }; //
        SPI_NOISE_EN
124     memory[28] <= {1'b0,3'b000,12'd1,16'd33,`SPI_FORCE_TRACES            }; //
        SPI_FORCE_TRACES
125     memory[29] <= {1'b0,3'b000,12'd1,16'd64,`SPI_CYCLES_PER_TICK         }; //
        SPI_CYCLES_PER_TICK
126     memory[30] <= {1'b0,3'b000,12'd1,16'd69,`SPI_KAPPA                   }; //
        SPI_KAPPA
127     memory[31] <= {1'b0,3'b000,12'd1,16'd70,`SPI_THR_H_0                 }; //
        SPI_THR_H_0
128     memory[32] <= {1'b0,3'b000,12'd1,16'd71,`SPI_THR_H_1                 }; //
        SPI_THR_H_1
129     memory[33] <= {1'b0,3'b000,12'd1,16'd72,`SPI_THR_H_2                 }; //
        SPI_THR_H_2
130     memory[34] <= {1'b0,3'b000,12'd1,16'd73,`SPI_THR_H_3                 }; //
        SPI_THR_H_3
131     memory[35] <= {1'b0,3'b000,12'd1,16'd74,`SPI_H_0                     }; //
        SPI_H_0
132     memory[36] <= {1'b0,3'b000,12'd1,16'd75,`SPI_H_1                     }; //
        SPI_H_1
133     memory[37] <= {1'b0,3'b000,12'd1,16'd76,`SPI_H_2                     }; //
        SPI_H_2
134     memory[38] <= {1'b0,3'b000,12'd1,16'd77,`SPI_H_3                     }; //
        SPI_H_3
135     memory[39] <= {1'b0,3'b000,12'd1,16'd78,`SPI_H_4                     }; //
        SPI_H_4
136     memory[40] <= {1'b0,3'b000,12'd1,16'd79,`SPI_LR_R_WINP               }; //
        SPI_LR_R_WINP
137     memory[41] <= {1'b0,3'b000,12'd1,16'd80,`SPI_LR_P_WINP               }; //
        SPI_LR_P_WINP
138     memory[42] <= {1'b0,3'b000,12'd1,16'd81,`SPI_LR_R_WREC               }; //
        SPI_LR_R_WREC
139     memory[43] <= {1'b0,3'b000,12'd1,16'd82,`SPI_LR_P_WREC               }; //
        SPI_LR_P_WREC
140     memory[44] <= {1'b0,3'b000,12'd1,16'd83,`SPI_LR_R_WOUT               }; //
        SPI_LR_R_WOUT
141     memory[45] <= {1'b0,3'b000,12'd1,16'd84,`SPI_LR_P_WOUT               }; //
        SPI_LR_P_WOUT
142     memory[46] <= {1'b0,3'b000,12'd1,16'd85,`SPI_SEED_INP                }; //
        SPI_SEED_INP
143     memory[47] <= {1'b0,3'b000,12'd1,16'd86,`SPI_SEED_REC                }; //
        SPI_SEED_REC
144     memory[48] <= {1'b0,3'b000,12'd1,16'd87,`SPI_SEED_OUT                }; //
        SPI_SEED_OUT
145     memory[49] <= {1'b0,3'b000,12'd1,16'd88,`SPI_SEED_STRND_NEUR         }; //
        SPI_SEED_STRND_NEUR
146     memory[50] <= {1'b0,3'b000,12'd1,16'd89,`SPI_SEED_STRND_ONEUR        }; //
        SPI_SEED_STRND_ONEUR
147     memory[51] <= {1'b0,3'b000,12'd1,16'd90,`SPI_SEED_STRND_TINP         }; //
        SPI_SEED_STRND_TINP
148     memory[52] <= {1'b0,3'b000,12'd1,16'd91,`SPI_SEED_STRND_TREC         }; //
```

```
        SPI_SEED_STRND_TREC
149   memory[53] <= {1'b0,3'b000,12'd1,16'd92,`SPI_SEED_STRND_TOUT          }; //
        SPI_SEED_STRND_TOUT
150   memory[54] <= {1'b0,3'b000,12'd1,16'd93,`SPI_SEED_NOISE_NEUR          }; //
        SPI_SEED_NOISE_NEUR
151   memory[55] <= {1'b0,3'b000,12'd1,16'd94,`SPI_NUM_INP_NEUR             }; //
        SPI_NUM_INP_NEUR
152   memory[56] <= {1'b0,3'b000,12'd1,16'd95,`SPI_NUM_REC_NEUR             }; //
        SPI_NUM_REC_NEUR
153   memory[57] <= {1'b0,3'b000,12'd1,16'd96,`SPI_NUM_OUT_NEUR             }; //
        SPI_NUM_OUT_NEUR
154   memory[58] <= {1'b0,3'b000,12'd1,16'd98,`SPI_REGUL_F0                 }; //
        SPI_REGUL_F0
155   memory[59] <= {1'b0,3'b000,12'd1,16'd99,  `SPI_REGUL_K_INP_R           };
        //SPI_REGUL_K_INP_R
156   memory[60] <= {1'b0,3'b000,12'd1,16'd100,`SPI_REGUL_K_INP_P            };
        //SPI_REGUL_K_INP_P
157   memory[61] <= {1'b0,3'b000,12'd1,16'd101,`SPI_REGUL_K_REC_R            };
        //SPI_REGUL_K_REC_R
158   memory[62] <= {1'b0,3'b000,12'd1,16'd102,  `SPI_REGUL_K_REC_P           };
        //SPI_REGUL_K_REC_P
159   memory[63] <= {1'b0,3'b000,12'd1,16'd103,`SPI_REGUL_K_MUL              };
        //SPI_REGUL_K_MUL
160   memory[64] <= {1'b0,3'b000,12'd1,16'd104,`SPI_NOISE_STR                };
        //SPI_NOISE_STR
161   memory[65] <= {1'b1,3'b111,12'd1,16'd0,'0};
162 end
163
164   always @(posedge clk_i)
165
166   begin : dummy_proc
167       if (enable_w) begin
168       for ( a=0 ; a < 5; a++) begin
169           add <= add + 1'b1;
170           pre_spi_data <= memory[add];
171         for (i=0 ; i<64; i=i+1) begin
172             spi_data  <= pre_spi_data[63-i];
173             //tbench.try  = MOSI;
174             #30;
175             sclk   <= 1'b1;
176             #30;
177             sclk  <= 1'b0;
178         end
179         i = 0;
180
181         end
182         done = 1'b1;
183       end
184
185     end
186
187
188
189
190
191
192
193 endmodule // tb_dummy_memory
194 $
```

Here is possible to see the core, as said before is not complete because it was not developed a slave interface to store data inside the memory bank but there is a preloaded SPI configuration. With the central task of the dummy process.
Then we have configured a structural using the two module of the reckon and the other module as the CU and the spi

```
1 // vsim -voptargs=+acc  work.tbench
```

```
2  timeunit 1ns;
3  timeprecision 1ns;
4
5  `define CLK_HALF_PERIOD              10
6  `define SCK_HALF_PERIOD              30
7
8  //sim:/tbench/tc/MOSI_prefifo/data
9
10 module tbench ();
11
12 logic          clk;
13 logic          sclk;
14 logic          RST = 1'b0;
15 logic          SPI_RDY;
16 logic          TIMING_ERROR_RDY;
17 logic          OUT_REQ;
18 logic          OUT_ACK;
19 logic   [7:0]  AERIN_ADDR ;
20 logic          AERIN_REQ;
21 logic          AERIN_ACK;
22 logic          AERIN_TAR_EN;
23 logic          SAMPLE;
24 logic          TIME_TICK;
25 logic          TARGET_VALID;
26 logic          INFER_ACC;
27 logic clear_i;
28 logic rk_ISR_o;
29 //sim:/tbench/RST
30
31
32 hwpe_stream_intf_stream  #(8) MOSI      ();
33 hwpe_stream_intf_stream  #(8) MISO      ();
34 hwpe_stream_intf_stream  #(8) OUT_DATA ();
35 hwpe_stream_intf_stream  #(8) MOSI_prefifo ();
36 logic       try;
37 logic    [7:0]  out_rec;
38
39
40 //
41 logic enable_w ;
42 logic spi_data ;
43 logic done ;
44 //
45
46
47 initial begin
48
49         RST = 1'b0;
50         # 100 ;
51         RST = 1'b1;
52         # 100 ;
53         RST = 1'b0;
54         # 100 ;
55         AERIN_ADDR   =  'd0;
56         AERIN_REQ    = 1'b0;
57         AERIN_TAR_EN = 1'd0;
58         TARGET_VALID = 1'b0;
59         INFER_ACC    = 1'b0;
60         SAMPLE       = 1'b0;
61         TIME_TICK    = 1'b0;
62         OUT_ACK      = 1'b0;
63         # 100 ;
64         enable_w = 1'b1;
65     end
66
67
68
69
```

```
70
71
72
73
74
75  mac_engine tv (    //(.N(256),.M(8))
76
77      // Global inputs   ---------------------------------
78      .CLK_EXT(clk),
79      .CLK_INT_EN(1'b0),
80      .RST(RST),
81
82      // SPI slave       ---------------------------------
83      .SCK(sclk),
84      .MOSI(spi_data),
85      .MISO(MISO.source),       //input
86
87       // Output bus and control outputs ------------------
88        .SPI_RDY    (SPI_RDY),
89        .TIMING_ERROR_RDY   (TIMING_ERROR_RDY),//output
90        .OUT_REQ   (OUT_REQ),//output
91        .OUT_ACK   (OUT_ACK),//INPUT
92      // AER
93        .AERIN_ADDR(AERIN_ADDR)  ,   //[M-1:0]
94        .AERIN_REQ(AERIN_REQ),
95        .AERIN_ACK(AERIN_ACK),
96        .AERIN_TAR_EN(AERIN_TAR_EN),
97        .SAMPLE(SAMPLE),
98        .TIME_TICK(TIME_TICK),
99        .TARGET_VALID(TARGET_VALID),
100       .INFER_ACC(INFER_ACC),
101       // Output bus and control outputs
102       .OUT_DATA(OUT_DATA.source ) //output
103 );
104
105
106
107 tb_dummy_memory #(
108   .MEMORY_SIZE(65),
109   .MEMORY_word(64),
110   .ADD_word(8))
111 dut (
112   .clk_i(clk),
113   .enable_w(enable_w),
114   //OUTPUT
115   .spi_data(spi_data) ,
116   .sclk(sclk) ,
117   .done (done)
118
119 );
120
121
122 cu cu
123 (
124   .clk_i(clk),
125   .rk_out_data_i(OUT_DATA.sink.data),
126   .rk_ISR_o(rk_ISR_o)
127
128 );
129
130   always
131
132   begin
133     # `CLK_HALF_PERIOD clk = 1;
134     # `CLK_HALF_PERIOD clk = 0;
135   end
136
137
```

```
138
139 endmodule
```

The result of this structural could be seen in the 3.13

3.4 Plans review

Once the presentation of the plan was made to the Pulpissimo and HWPE authors they have suggested to change the configuration and the idea in one more robust using the Pulpissimo modules already ready and tested, here below we report the proposed configuration by the authors.

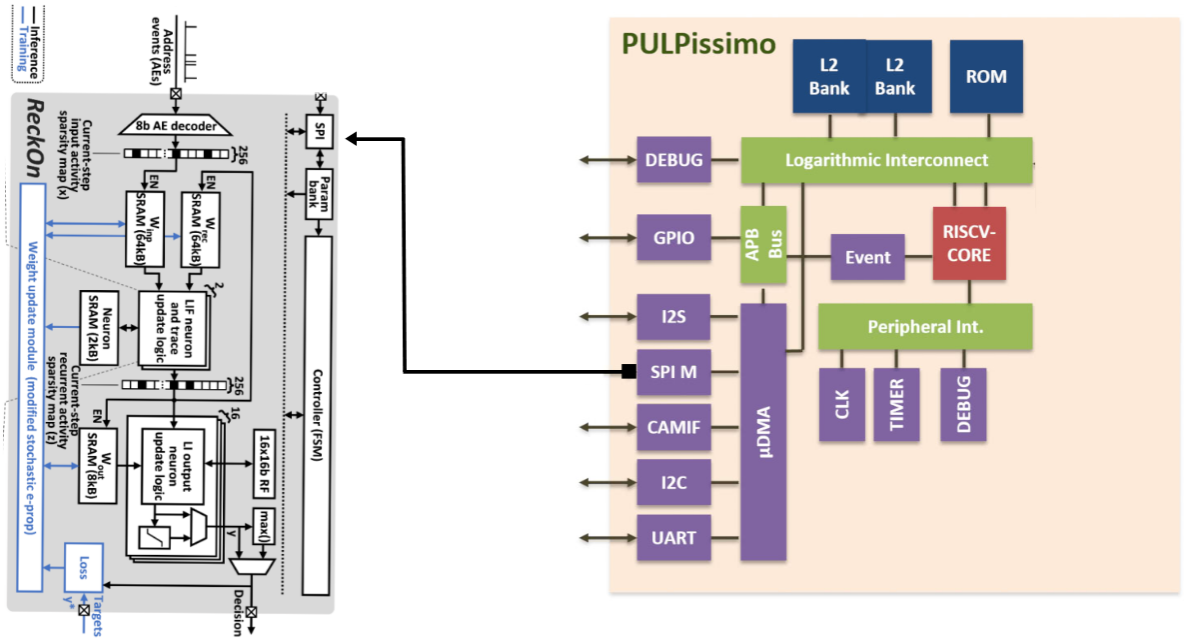

Figure 3.14: Final configuration

As it is possible to see the idea is straightforward: we use the uDMA for sending SPI data, and we are using the L2 bank of memory to store all the possible configuration In this way we could have multiple SNN configuration that could be loaded to Reckon whenever we want.
Doing so we'll be able to exclude the SOC from the pc and using independently for embedded system.

Data wire that it is attached directly on Reckon's SPI. The memory was design for be able to send on the control unit a signal that inform the system that all the parameters are sent correctly and it is named as Done. Done have a 0 for default and 1 when we reach the bottom of the RAM.

3.5 uDMA

uDMA (micro Direct Memory Access) and DMA (Direct Memory Access) are both technologies used to transfer data between different devices in a SOC without involving the central processing unit CPU or request processor. However, there are some key differences between them:

- Scale: uDMA is designed to be used in smaller especially for embedded systems domain, while DMA is typically used in larger complex, systems for support heavy memory translation.

- Complexity: uDMA is generally simpler and has less functionality due his nature compared to DMA, making it more suitable for smaller systems.
- Performance: Since DMA it was not designed for be less restricted it is generally faster than uDMA due to its more advanced features and support for multiple channels and multi peripheral structure. Indeed DMA allows for multiple data transfers to occur simultaneously.
- Cost: uDMA is generally less expensive than DMA due to its simpler design and smaller scale.

In summary, the choice between uDMA and DMA depends on the specific requirements and constraints of a system; In Pulpissimo, embedded system oriented design, the choice of uDMA was obvious.
Here we are going to see the central part about the Pulpissimo system in the soc domain.
In the architecture there are two separate part: the pad and the soc environment. The environment could be enclose in various sub module that have multiple functionality and, linked one to other, will produce the final result.

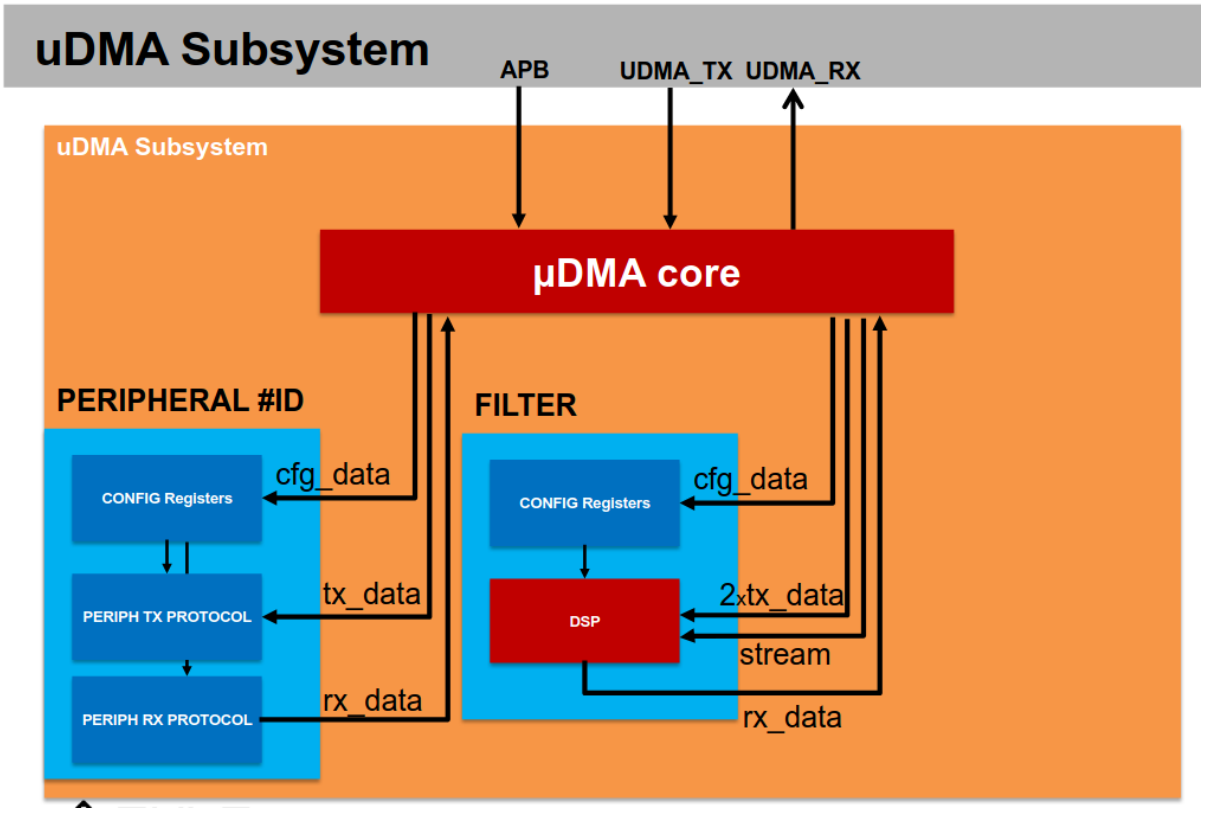

Figure 3.15: uDMA [9]

In the original project there are several unit that we are going to discover a little:

Top module

The uDMA subsystem is composed by a initial declaration that has the role to set up all the parameter inside the generate statement embedded in the peripheral modules. Indeed, the L2 parameter have the role to fix the value of the memory bank size in terms of address and data width. Later the most important setting is refereed to the number of peripheral port that we are going to have in the SOC.
This data it is important because will tailor the receiver and the sender bus based on the number of port. In terms of interface we are going to see that there are a interface to receive

data in memory with the L2_ro prefix and L2_wo for sending data in and from memory. This interface it is casted to be an HWPE standard.

```
module udma_subsystem
#(
    parameter L2_DATA_WIDTH  = 32,
    parameter L2_ADDR_WIDTH  = 19,   //L2 addr space of 2MB
    parameter CAM_DATA_WIDTH = 8,
    parameter APB_ADDR_WIDTH = 12,  //APB slaves are 4KB by default
    parameter TRANS_SIZE     = 20,  //max uDMA transaction size of 1MB
    parameter N_SPI          = 4,
    parameter N_UART         = 4,
    parameter N_I2C          = 1,
    parameter N_HYPER        = 1,

    localparam N_PERIPH_MAX  = 32
)
(
    output logic                          L2_ro_wen_o     ,
    output logic                          L2_ro_req_o     ,
    input  logic                          L2_ro_gnt_i     ,
    output logic                  [31:0]  L2_ro_addr_o    ,
    output logic  [L2_DATA_WIDTH/8-1:0]   L2_ro_be_o      ,
    output logic    [L2_DATA_WIDTH-1:0]   L2_ro_wdata_o   ,
    input  logic                          L2_ro_rvalid_i  ,
    input  logic    [L2_DATA_WIDTH-1:0]   L2_ro_rdata_i   ,

    output logic                          L2_wo_wen_o     ,
    output logic                          L2_wo_req_o     ,
    input  logic                          L2_wo_gnt_i     ,
    output logic                  [31:0]  L2_wo_addr_o    ,
    output logic    [L2_DATA_WIDTH-1:0]   L2_wo_wdata_o   ,
    output logic  [L2_DATA_WIDTH/8-1:0]   L2_wo_be_o      ,
    input  logic                          L2_wo_rvalid_i  ,
    input  logic    [L2_DATA_WIDTH-1:0]   L2_wo_rdata_i   ,
```

Always in the same module declaration we could see that there is some configuration with the idea of using two clock signal and a reset trigger.

```
    input  logic                          dft_test_mode_i,
    input  logic                          dft_cg_enable_i,
    input  logic                          sys_clk_i,
    input  logic                          sys_resetn_i,
    input  logic                          periph_clk_i,
```

Then we have the interface that comunicate with the APB bus. This is done because we could expect interrupt or other control signal from the rest of component

```

    input  logic  [APB_ADDR_WIDTH-1:0]  udma_apb_paddr,
    input  logic                  [31:0]  udma_apb_pwdata,
    input  logic                          udma_apb_pwrite,
    input  logic                          udma_apb_psel,
    input  logic                          udma_apb_penable,
    output logic                  [31:0]  udma_apb_prdata,
    output logic                          udma_apb_pready,
    output logic                          udma_apb_pslverr,
    output logic             [N_PERIPH_MAX*4-1:0]  events_o,
    input  logic                          event_valid_i,
    input  logic                   [7:0]  event_data_i,
    output logic                          event_ready_o,
```

Here come the most important element for our task: the SPI.
Here is possible to see that there are a standard logic bus of spi. of course the Sclk is unique but the data and the control signal are sized in view of the number of SPI that we want to integrate in our system with the parametric N_SPI

```
1     // SPIM
2     output logic      [N_SPI-1:0]       spi_clk,
3     output logic      [N_SPI-1:0] [3:0] spi_csn,
4     output logic      [N_SPI-1:0] [3:0] spi_oen,
5     output logic      [N_SPI-1:0] [3:0] spi_sdo,
6     input  logic      [N_SPI-1:0] [3:0] spi_sdi,
```

Then we do not report the HDL for other port (I2C,UART,SDIO,I2S,HYPERBUS) interfaces because will not be used for the moment but for completeness we will do a little description, so later we are able to valuate a greater future substitution of SPI:

I2C is a serial communication protocol used for extra chip communication formed by two-wire interface that uses a clock signal, controlled by master, and a data signal to transmit data between devices (slave). The transition speed is from 100 kbps to 5 Mbps and Support multiple slave recognized with incremental or arbitrary unique address on 7-bit and 10-bit with bursting control and data information .Is important to underline that I2C are mostly spread in sensors and EEPROMs memory.

Also, the UART have some similar feature: a transit line (TX) and a reception line (RX). The master stream down data on TX for RX, and they are clocked internal for the speed asynchronous. UART is Simple and easy to implement and also low cost and low power consumption used commonly for long distance inclined for the total separation between the SOCs.

Another peripheral that is implemented is the SDIO (Secure Digital Input Output) used for high-speed communication between the host device and the peripheral device from 25 Mbps up to 104 Mbps.
Since it is streaming media and video oriented, due high-bandwidth data transfer, this allows a fast burst of data on the peripheral output with also a low power consumption so ideal for battery-powered devices such as drones or unnamed devices.
Another unit is serial communication protocol the Inter-IC Sound used to transmit digital audio signals between integrated circuits over a single data line with separate clock and control signals. I2S is commonly used in audio applications such as digital signal processing, digital-to-analog converters (DACs), and audio codecs. It supports high-quality audio with low noise and low distortion and is widely used in consumer and professional audio equipment.

The last in the list is the HYPERBUS a new high-speed, low-power interface protocol that was thought for fast communication between a host processor and high-performance memory devices based on 12-pin interface and supports data transfer rates of up to 400 MB/s. Since it is really fast it is used to high-performance embedded applications that require fast data transfer, such as automotive infotainment systems or industrial control systems so that applications that require high-bandwidth memory access.
The SPI and I2C could not compete and probably will be the next implementation for reckon if we want a top edge SOC.

signal and parameter setting

May seem that next part it is useless to be reporeted but this parameter and those signal reppresent the backbone of the entire system:

```
1  localparam DEST_SIZE = 2;
2
3      localparam L2_AWIDTH_NOAL = L2_ADDR_WIDTH + 2;
4
5      localparam N_I2S      = 1;
6      localparam N_CAM      = 1;
7      localparam N_CSI2     = 0;
8      localparam N_SDIO     = 1;
9      localparam N_JTAG     = 0;
10     localparam N_MRAM     = 0;
```

```
11      localparam N_FILTER    = 1;
12      localparam N_CH_HYPER = 8;
13      localparam N_FPGA      = 0;
14      localparam N_EXT_PER   = 0;
15
16      localparam N_RX_CHANNELS =    N_SPI + N_HYPER + N_MRAM + N_JTAG + N_SDIO +
            N_UART + N_I2C + N_I2S + N_CAM + 2*N_CSI2 + N_FPGA + N_EXT_PER +
            N_CH_HYPER;
17      localparam N_TX_CHANNELS = 2*N_SPI + N_HYPER + N_MRAM + N_JTAG + N_SDIO +
            N_UART + 2*N_I2C + N_I2S + N_FPGA + N_EXT_PER + N_CH_HYPER;
18
19      localparam N_RX_EXT_CHANNELS =    N_FILTER;
20      localparam N_TX_EXT_CHANNELS = 2*N_FILTER;
21      localparam N_STREAMS         =    N_FILTER;
22      localparam STREAM_ID_WIDTH   = 1;//clog2(N_STREAMS)
23
24      localparam N_PERIPHS = N_SPI + N_HYPER + N_UART + N_MRAM + N_I2C + N_CAM +
             N_I2S + N_CSI2 + N_SDIO + N_JTAG + N_FILTER + N_FPGA + N_EXT_PER +
            N_CH_HYPER;
```

At this point we are going to define the parameter using the number of component inside the system. At the next we are going to develop the standard logic size of the rx and tx bus

```
1       // Currently s_events is designed for N_PERIPH=32. If we change this then
2       // make sure all the events are correctly mapped and connected.
3       if (N_PERIPHS > N_PERIPH_MAX)
4           fatal(1, "number of events is desigend for at most %d peripherals",
                N_PERIPH_MAX);
5
6       // TX Channels
7       localparam CH_ID_TX_UART     = 0;
8       localparam CH_ID_TX_SPIM     = N_UART;
9       localparam CH_ID_CMD_SPIM    = CH_ID_TX_SPIM  + N_SPI   ;
10      localparam CH_ID_TX_I2C      = CH_ID_CMD_SPIM + N_SPI   ;
11      localparam CH_ID_CMD_I2C     = CH_ID_TX_I2C   + N_I2C   ;
12      localparam CH_ID_TX_SDIO     = CH_ID_CMD_I2C  + N_I2C   ;
13      localparam CH_ID_TX_I2S      = CH_ID_TX_SDIO  + N_SDIO  ;
14      localparam CH_ID_TX_CAM      = CH_ID_TX_I2S   + N_I2S   ;
15      localparam CH_ID_TX_HYPER    = CH_ID_TX_CAM   + N_CAM   ;
16      // Tx Ext Channel
17      localparam CH_ID_TX_EXT_PER = CH_ID_TX_HYPER + N_HYPER + N_CH_HYPER;
18
19
20      // RX Channels
21      localparam CH_ID_RX_UART     = 0;
22      localparam CH_ID_RX_SPIM     = N_UART;
23      localparam CH_ID_RX_I2C      = CH_ID_RX_SPIM  + N_SPI   ;
24      localparam CH_ID_RX_SDIO     = CH_ID_RX_I2C   + N_I2C   ;
25      localparam CH_ID_RX_I2S      = CH_ID_RX_SDIO  + N_SDIO  ;
26      localparam CH_ID_RX_CAM      = CH_ID_RX_I2S   + N_I2S   ;
27      localparam CH_ID_RX_HYPER    = CH_ID_RX_CAM   + N_CAM   ;
28      // Rx Ext Channel
29      localparam CH_ID_RX_EXT_PER = CH_ID_RX_HYPER + N_HYPER + N_CH_HYPER;
30
31      // Stream Channel
32      localparam STREAM_ID_FILTER = 0;
33
34      localparam CH_ID_EXT_TX_FILTER = 0;
35      localparam CH_ID_EXT_RX_FILTER = 0;
36
37      localparam PER_ID_UART     = 0;
38      localparam PER_ID_SPIM     = PER_ID_UART    + N_UART    ;
39      localparam PER_ID_I2C      = PER_ID_SPIM    + N_SPI     ;
40      localparam PER_ID_SDIO     = PER_ID_I2C     + N_I2C     ;
41      localparam PER_ID_I2S      = PER_ID_SDIO    + N_SDIO    ;
42      localparam PER_ID_CAM      = PER_ID_I2S     + N_I2S     ;
43      localparam PER_ID_FILTER   = PER_ID_CAM     + N_CAM     ;
44      localparam PER_ID_HYPER    = PER_ID_FILTER + N_FILTER  ;
```

```
45      localparam PER_ID_EXT_PER = PER_ID_HYPER  + N_HYPER  + N_CH_HYPER;
```

core

Before going deep witH the SPI logic we are going to see how the core is connected with the other external peripheral. Each component have an independent interface that will be used to control the stream of data for and inside the core

```
1   udma_core #(
2      .L2_AWIDTH_NOAL     ( L2_AWIDTH_NOAL     ),
3      .L2_DATA_WIDTH      ( L2_DATA_WIDTH      ),
4      .DATA_WIDTH         ( 32                 ),
5      .N_RX_LIN_CHANNELS  ( N_RX_CHANNELS      ),
6      .N_TX_LIN_CHANNELS  ( N_TX_CHANNELS      ),
7      .N_RX_EXT_CHANNELS  ( N_RX_EXT_CHANNELS  ),
8      .N_TX_EXT_CHANNELS  ( N_TX_EXT_CHANNELS  ),
9      .N_STREAMS          ( N_STREAMS          ),
10     .STREAM_ID_WIDTH    ( STREAM_ID_WIDTH    ),
11     .TRANS_SIZE         ( TRANS_SIZE         ),
12     .N_PERIPHS          ( N_PERIPHS          ),
13     .APB_ADDR_WIDTH     ( APB_ADDR_WIDTH     )
14     ) i_udmacore (
15     .sys_clk_i            ( sys_clk_i           ),
16     .per_clk_i            ( periph_clk_i        ),
17
18     .dft_cg_enable_i      ( dft_cg_enable_i     ),
19
20     .HRESETn              ( sys_resetn_i        ),
21
22     .PADDR                ( udma_apb_paddr      ),
23     .PWDATA               ( udma_apb_pwdata     ),
24     .PWRITE               ( udma_apb_pwrite     ),
25     .PSEL                 ( udma_apb_psel       ),
26     .PENABLE              ( udma_apb_penable    ),
27     .PRDATA               ( udma_apb_prdata     ),
28     .PREADY               ( udma_apb_pready     ),
29     .PSLVERR              ( udma_apb_pslverr    ),
30
31     .periph_per_clk_o     ( s_clk_periphs_per   ),
32     .periph_sys_clk_o     ( s_clk_periphs_core  ),
33
34     .event_valid_i        ( event_valid_i       ),
35     .event_data_i         ( event_data_i        ),
36     .event_ready_o        ( event_ready_o       ),
37
38     .event_o              ( s_trigger_events    ),
39
40     .periph_data_to_o     ( s_periph_data_to    ),
41     .periph_addr_o        ( s_periph_addr       ),
42     .periph_data_from_i   ( s_periph_data_from  ),
43     .periph_ready_i       ( s_periph_ready      ),
44     .periph_valid_o       ( s_periph_valid      ),
45     .periph_rwn_o         ( s_periph_rwn        ),
46
47     .tx_l2_req_o          ( L2_ro_req_o         ),
48     .tx_l2_gnt_i          ( L2_ro_gnt_i         ),
49     .tx_l2_addr_o         ( L2_ro_addr_o        ),
50     .tx_l2_rdata_i        ( L2_ro_rdata_i       ),
51     .tx_l2_rvalid_i       ( L2_ro_rvalid_i      ),
52
53     .rx_l2_req_o          ( L2_wo_req_o         ),
54     .rx_l2_gnt_i          ( L2_wo_gnt_i         ),
55     .rx_l2_addr_o         ( L2_wo_addr_o        ),
56     .rx_l2_be_o           ( L2_wo_be_o          ),
57     .rx_l2_wdata_o        ( L2_wo_wdata_o       ),
58
```

```
59      .stream_data_o        ( s_stream_data       ),
60      .stream_datasize_o    ( s_stream_datasize   ),
61      .stream_valid_o       ( s_stream_valid      ),
62      .stream_sot_o         ( s_stream_sot        ),
63      .stream_eot_o         ( s_stream_eot        ),
64      .stream_ready_i       ( s_stream_ready      ),
65
66      .tx_lin_req_i          ( s_tx_ch_req           ),
67      .tx_lin_gnt_o          ( s_tx_ch_gnt           ),
68      .tx_lin_valid_o        ( s_tx_ch_valid         ),
69      .tx_lin_data_o         ( s_tx_ch_data          ),
70      .tx_lin_ready_i        ( s_tx_ch_ready         ),
71      .tx_lin_datasize_i     ( s_tx_ch_datasize      ),
72      .tx_lin_destination_i ( s_tx_ch_destination  ),
73      .tx_lin_events_o       ( s_tx_ch_events        ),
74      .tx_lin_en_o           ( s_tx_ch_en            ),
75      .tx_lin_pending_o      ( s_tx_ch_pending       ),
76      .tx_lin_curr_addr_o    ( s_tx_ch_curr_addr     ),
77      .tx_lin_bytes_left_o   ( s_tx_ch_bytes_left    ),
78      .tx_lin_cfg_startaddr_i  ( s_tx_cfg_startaddr   ),
79      .tx_lin_cfg_size_i       ( s_tx_cfg_size        ),
80      .tx_lin_cfg_continuous_i ( s_tx_cfg_continuous  ),
81      .tx_lin_cfg_en_i         ( s_tx_cfg_en          ),
82      .tx_lin_cfg_clr_i        ( s_tx_cfg_clr         ),
83
84      .rx_lin_valid_i          ( s_rx_ch_valid        ),
85      .rx_lin_data_i           ( s_rx_ch_data         ),
86      .rx_lin_ready_o          ( s_rx_ch_ready        ),
87      .rx_lin_datasize_i       ( s_rx_ch_datasize     ),
88      .rx_lin_destination_i    ( s_rx_ch_destination  ),
89      .rx_lin_events_o         ( s_rx_ch_events       ),
90      .rx_lin_en_o             ( s_rx_ch_en           ),
91      .rx_lin_pending_o        ( s_rx_ch_pending      ),
92      .rx_lin_curr_addr_o      ( s_rx_ch_curr_addr    ),
93      .rx_lin_bytes_left_o     ( s_rx_ch_bytes_left   ),
94      .rx_lin_cfg_startaddr_i  ( s_rx_cfg_startaddr   ),
95      .rx_lin_cfg_size_i       ( s_rx_cfg_size        ),
96      .rx_lin_cfg_continuous_i ( s_rx_cfg_continuous  ),
97      .rx_lin_cfg_stream_i     ( s_rx_cfg_stream      ),
98      .rx_lin_cfg_stream_id_i  ( s_rx_cfg_stream_id   ),
99      .rx_lin_cfg_en_i         ( s_rx_cfg_en          ),
100     .rx_lin_cfg_clr_i        ( s_rx_cfg_clr         ),
101
102     .rx_ext_addr_i           ( s_rx_ext_addr        ),
103     .rx_ext_datasize_i       ( s_rx_ext_datasize    ),
104     .rx_ext_destination_i    ( s_rx_ext_destination ),
105     .rx_ext_stream_i         ( s_rx_ext_stream      ),
106     .rx_ext_stream_id_i      ( s_rx_ext_stream_id   ),
107     .rx_ext_sot_i            ( s_rx_ext_sot         ),
108     .rx_ext_eot_i            ( s_rx_ext_eot         ),
109     .rx_ext_valid_i          ( s_rx_ext_valid       ),
110     .rx_ext_data_i           ( s_rx_ext_data        ),
111     .rx_ext_ready_o          ( s_rx_ext_ready       ),
112
113     .tx_ext_req_i            ( s_tx_ext_req         ),
114     .tx_ext_datasize_i       ( s_tx_ext_datasize    ),
115     .tx_ext_destination_i    ( s_tx_ext_destination ),
116     .tx_ext_addr_i           ( s_tx_ext_addr        ),
117     .tx_ext_gnt_o            ( s_tx_ext_gnt         ),
118     .tx_ext_valid_o          ( s_tx_ext_valid       ),
119     .tx_ext_data_o           ( s_tx_ext_data        ),
120     .tx_ext_ready_i          ( s_tx_ext_ready       )
121
122     );
```

Since the other interface for the moment will not be taken in consideration we are going to see something more interesting about the SPI interface.

The interface it was developed for generic structure of interface and with a generic recurrent statement it is possibe to cast a multi SPI ports.

```
1
2   //PER_ID 1
3     generate
4         for (genvar g_spi=0;g_spi<N_SPI;g_spi++)
5         begin : i_spim_gen
6  assign s_events[4*(PER_ID_SPIM+g_spi)+0] = s_rx_ch_events[CH_ID_RX_SPIM+g_spi
       ];
7  assign s_events[4*(PER_ID_SPIM+g_spi)+1] = s_tx_ch_events[CH_ID_TX_SPIM+g_spi
       ];
8  assign s_events[4*(PER_ID_SPIM+g_spi)+2] = s_tx_ch_events[CH_ID_CMD_SPIM+g_spi
       ];
9  assign s_events[4*(PER_ID_SPIM+g_spi)+3] = s_spi_eot[g_spi];
10
11 assign s_rx_cfg_stream[CH_ID_RX_SPIM+g_spi] = 'h0;
12 assign s_rx_cfg_stream_id[CH_ID_RX_SPIM+g_spi] = 'h0;
13 assign s_rx_ch_destination[CH_ID_RX_SPIM+g_spi] = 'h0;
14 assign s_tx_ch_destination[CH_ID_TX_SPIM+g_spi] = 'h0;
15 assign s_tx_ch_destination[CH_ID_CMD_SPIM+g_spi] = 'h0;
16             udma_spim_top
17             #(
18 .L2_AWIDTH_NOAL       ( L2_AWIDTH_NOAL                              ),
19 .TRANS_SIZE           ( TRANS_SIZE                                  )
20             ) i_spim (
21 .sys_clk_i            ( s_clk_periphs_core[PER_ID_SPIM+g_spi]       ),
22 .periph_clk_i         ( s_clk_periphs_per[PER_ID_SPIM+g_spi]        ),
23 .rstn_i               ( sys_resetn_i                                ),
24 .dft_test_mode_i      ( dft_test_mode_i                             ),
25 .dft_cg_enable_i      ( dft_cg_enable_i                             ),
26 .spi_eot_o            ( s_spi_eot[g_spi]                            ),
27 .spi_event_i          ( s_trigger_events                            ),
28 .spi_clk_o            ( spi_clk[g_spi]                              ),
29 .spi_csn0_o           ( spi_csn[g_spi][0]                           ),
30 .spi_csn1_o           ( spi_csn[g_spi][1]                           ),
31 .spi_csn2_o           ( spi_csn[g_spi][2]                           ),
32 .spi_csn3_o           ( spi_csn[g_spi][3]                           ),
33 .spi_oen0_o           ( spi_oen[g_spi][0]                           ),
34 .spi_oen1_o           ( spi_oen[g_spi][1]                           ),
35 .spi_oen2_o           ( spi_oen[g_spi][2]                           ),
36 .spi_oen3_o           ( spi_oen[g_spi][3]                           ),
37 .spi_sdo0_o           ( spi_sdo[g_spi][0]                           ),
38 .spi_sdo1_o           ( spi_sdo[g_spi][1]                           ),
39 .spi_sdo2_o           ( spi_sdo[g_spi][2]                           ),
40 .spi_sdo3_o           ( spi_sdo[g_spi][3]                           ),
41 .spi_sdi0_i           ( spi_sdi[g_spi][0]                           ),
42 .spi_sdi1_i           ( spi_sdi[g_spi][1]                           ),
43 .spi_sdi2_i           ( spi_sdi[g_spi][2]                           ),
44 .spi_sdi3_i           ( spi_sdi[g_spi][3]                           ),
45
46 .cfg_data_i           ( s_periph_data_to                            ),
47 .cfg_addr_i           ( s_periph_addr                               ),
48 .cfg_valid_i          ( s_periph_valid[PER_ID_SPIM+g_spi]           ),
49 .cfg_rwn_i            ( s_periph_rwn                                ),
50 .cfg_data_o           ( s_periph_data_from[PER_ID_SPIM+g_spi]       ),
51 .cfg_ready_o          ( s_periph_ready[PER_ID_SPIM+g_spi]           ),
52
53 .cmd_req_o            ( s_tx_ch_req[CH_ID_CMD_SPIM+g_spi]           ),
54 .cmd_gnt_i            ( s_tx_ch_gnt[CH_ID_CMD_SPIM+g_spi]           ),
55 .cmd_datasize_o       ( s_tx_ch_datasize[CH_ID_CMD_SPIM+g_spi]      ),
56 .cmd_i                ( s_tx_ch_data[CH_ID_CMD_SPIM+g_spi]          ),
57 .cmd_valid_i          ( s_tx_ch_valid[CH_ID_CMD_SPIM+g_spi]         ),
58 .cmd_ready_o          ( s_tx_ch_ready[CH_ID_CMD_SPIM+g_spi]         ),
59
60 .data_tx_req_o        ( s_tx_ch_req[CH_ID_TX_SPIM+g_spi]            ),
61 .data_tx_gnt_i        ( s_tx_ch_gnt[CH_ID_TX_SPIM+g_spi]            ),
```

```
62  .data_tx_datasize_o   ( s_tx_ch_datasize[CH_ID_TX_SPIM+g_spi]        ),
63  .data_tx_i            ( s_tx_ch_data[CH_ID_TX_SPIM+g_spi]            ),
64  .data_tx_valid_i      ( s_tx_ch_valid[CH_ID_TX_SPIM+g_spi]           ),
65  .data_tx_ready_o      ( s_tx_ch_ready[CH_ID_TX_SPIM+g_spi]           ),
66
67  .data_rx_datasize_o   ( s_rx_ch_datasize[CH_ID_RX_SPIM+g_spi]        ),
68  .data_rx_o            ( s_rx_ch_data[CH_ID_RX_SPIM+g_spi]            ),
69  .data_rx_valid_o      ( s_rx_ch_valid[CH_ID_RX_SPIM+g_spi]           ),
70  .data_rx_ready_i      ( s_rx_ch_ready[CH_ID_RX_SPIM+g_spi]           ),
71
72  .cfg_cmd_startaddr_o  ( s_tx_cfg_startaddr[CH_ID_CMD_SPIM+g_spi]     ),
73  .cfg_cmd_size_o       ( s_tx_cfg_size[CH_ID_CMD_SPIM+g_spi]          ),
74  .cfg_cmd_continuous_o ( s_tx_cfg_continuous[CH_ID_CMD_SPIM+g_spi]    ),
75  .cfg_cmd_en_o         ( s_tx_cfg_en[CH_ID_CMD_SPIM+g_spi]            ),
76  .cfg_cmd_clr_o        ( s_tx_cfg_clr[CH_ID_CMD_SPIM+g_spi]           ),
77  .cfg_cmd_en_i         ( s_tx_ch_en[CH_ID_CMD_SPIM+g_spi]             ),
78  .cfg_cmd_pending_i    ( s_tx_ch_pending[CH_ID_CMD_SPIM+g_spi]        ),
79  .cfg_cmd_curr_addr_i  ( s_tx_ch_curr_addr[CH_ID_CMD_SPIM+g_spi]     ),
80  .cfg_cmd_bytes_left_i ( s_tx_ch_bytes_left[CH_ID_CMD_SPIM+g_spi]    ),
81
82  .cfg_tx_startaddr_o   ( s_tx_cfg_startaddr[CH_ID_TX_SPIM+g_spi]      ),
83  .cfg_tx_size_o        ( s_tx_cfg_size[CH_ID_TX_SPIM+g_spi]           ),
84  .cfg_tx_continuous_o  ( s_tx_cfg_continuous[CH_ID_TX_SPIM+g_spi]     ),
85  .cfg_tx_en_o          ( s_tx_cfg_en[CH_ID_TX_SPIM+g_spi]             ),
86  .cfg_tx_clr_o         ( s_tx_cfg_clr[CH_ID_TX_SPIM+g_spi]            ),
87  .cfg_tx_en_i          ( s_tx_ch_en[CH_ID_TX_SPIM+g_spi]              ),
88  .cfg_tx_pending_i     ( s_tx_ch_pending[CH_ID_TX_SPIM+g_spi]         ),
89  .cfg_tx_curr_addr_i   ( s_tx_ch_curr_addr[CH_ID_TX_SPIM+g_spi]       ),
90  .cfg_tx_bytes_left_i  ( s_tx_ch_bytes_left[CH_ID_TX_SPIM+g_spi]      ),
91
92  .cfg_rx_startaddr_o   ( s_rx_cfg_startaddr[CH_ID_RX_SPIM+g_spi]      ),
93  .cfg_rx_size_o        ( s_rx_cfg_size[CH_ID_RX_SPIM+g_spi]           ),
94  .cfg_rx_continuous_o  ( s_rx_cfg_continuous[CH_ID_RX_SPIM+g_spi]     ),
95  .cfg_rx_en_o          ( s_rx_cfg_en[CH_ID_RX_SPIM+g_spi]             ),
96  .cfg_rx_clr_o         ( s_rx_cfg_clr[CH_ID_RX_SPIM+g_spi]            ),
97  .cfg_rx_en_i          ( s_rx_ch_en[CH_ID_RX_SPIM+g_spi]              ),
98  .cfg_rx_pending_i     ( s_rx_ch_pending[CH_ID_RX_SPIM+g_spi]         ),
99  .cfg_rx_curr_addr_i   ( s_rx_ch_curr_addr[CH_ID_RX_SPIM+g_spi]       ),
100 .cfg_rx_bytes_left_i  ( s_rx_ch_bytes_left[CH_ID_RX_SPIM+g_spi]      )
101   );
```

3.6 SPI

In the next section we are going to see the most important component that is central for our target: the SPI module. This component has a specific integration that would be taken in consideration.
Befor going further i will introduce better the SPI. SPI is a synchronous serial communication protocol that is widely used to transfer data between modules and Soc that allows devices to communicate with each other using a minimum of wires or interconnection. The master-slave architecture it is designed to be monolithic and centralized : one device acts as the master and controls the clock signal, while the other device(s) act as slaves and receive or transmit data as directed by the master in a data transferred bit-by-bit in full duplex mode. The SPI protocol supports multiple slave devices, which can be connected to a single master and addressed individually, This is really important because in the application that we are developing we could create a cluster of different neuromorifc agent for example. Since there is only one wire and the synchronization of the data is not required this protocol is suitable for applications that require fast data transfer using high rate of the clock. Regarding the most important signal we have in general the Chip Select (CS) signal to select the slave device from the master, but will not be used for our proposes.
Regarding the Data frame format SPI protocol is typically used in 8 bits but in this application is sized to 32

Despite its advantages, the SPI protocol it is limited in its support for long-distance communication, and it may not be the best choice for applications that require large amounts of data to be transferred over long distances and supports a limited number of devices (usually up to 8) on a single bus, which may not be sufficient for large-scale systems. These two disadvantages are not affecting our design because we have only one Coprocessor.
A problem that could be however affecting our process is the data transfer rate of the SPI protocol limited by the speed of the clock signal if not well-designed and electronically efficient. Anyway the SPI is great for an initial prototype but for a more sophisticated board we will need a faster protocol that could be used online and offline in writing process to Reckon while it is running. Here the interface code

```
1   module udma_spim_top
2   #(
3       parameter L2_AWIDTH_NOAL = 12,
4       parameter TRANS_SIZE     = 16,
5       parameter REPLAY_BUFFER_DEPTH = 6
6   )
7   (
8  input  logic                          sys_clk_i,
9  input  logic                          periph_clk_i,
10 input  logic                          rstn_i,
11 input  logic                          dft_test_mode_i,
12 input  logic                          dft_cg_enable_i,
13 output logic                          spi_eot_o,
14 input  logic                   [3:0]  spi_event_i,
15 input  logic                  [31:0]  cfg_data_i,
16 input  logic                   [4:0]  cfg_addr_i,
17 input  logic                          cfg_valid_i,
18 input  logic                          cfg_rwn_i,
19 output logic                  [31:0]  cfg_data_o,
20 output logic                          cfg_ready_o,
21 output logic [L2_AWIDTH_NOAL-1:0]     cfg_cmd_startaddr_o,
22 output logic      [TRANS_SIZE-1:0]    cfg_cmd_size_o,
23 output logic                          cfg_cmd_continuous_o,
24 output logic                          cfg_cmd_en_o,
25 output logic                          cfg_cmd_clr_o,
26 input  logic                          cfg_cmd_en_i,
27 input  logic                          cfg_cmd_pending_i,
28 input  logic [L2_AWIDTH_NOAL-1:0]     cfg_cmd_curr_addr_i,
29 input  logic      [TRANS_SIZE-1:0]    cfg_cmd_bytes_left_i,
```

```
30
31 output logic [L2_AWIDTH_NOAL-1:0] cfg_rx_startaddr_o,
32 output logic      [TRANS_SIZE-1:0] cfg_rx_size_o,
33 output logic                       cfg_rx_continuous_o,
34 output logic                       cfg_rx_en_o,
35 output logic                       cfg_rx_clr_o,
36 input  logic                       cfg_rx_en_i,
37 input  logic                       cfg_rx_pending_i,
38 input  logic [L2_AWIDTH_NOAL-1:0] cfg_rx_curr_addr_i,
39 input  logic      [TRANS_SIZE-1:0] cfg_rx_bytes_left_i,
40
41 output logic [L2_AWIDTH_NOAL-1:0] cfg_tx_startaddr_o,
42 output logic      [TRANS_SIZE-1:0] cfg_tx_size_o,
43 output logic                       cfg_tx_continuous_o,
44 output logic                       cfg_tx_en_o,
45 output logic                       cfg_tx_clr_o,
46 input  logic                       cfg_tx_en_i,
47 input  logic                       cfg_tx_pending_i,
48 input  logic [L2_AWIDTH_NOAL-1:0] cfg_tx_curr_addr_i,
49 input  logic      [TRANS_SIZE-1:0] cfg_tx_bytes_left_i,
50
51 output logic                       cmd_req_o,
52 input  logic                       cmd_gnt_i,
53 output logic                [1:0] cmd_datasize_o,
54 input  logic               [31:0] cmd_i,
55 input  logic                       cmd_valid_i,
56 output logic                       cmd_ready_o,
57
58 output logic                       data_tx_req_o,
59 input  logic                       data_tx_gnt_i,
60 output logic                [1:0] data_tx_datasize_o,
61 input  logic               [31:0] data_tx_i,
62 input  logic                       data_tx_valid_i,
63 output logic                       data_tx_ready_o,
64
65 output logic                [1:0] data_rx_datasize_o,
66 output logic               [31:0] data_rx_o,
67 output logic                       data_rx_valid_o,
68 input  logic                       data_rx_ready_i,
69
70 output logic                       spi_clk_o,
71 output logic                       spi_csn0_o,
72 output logic                       spi_csn1_o,
73 output logic                       spi_csn2_o,
74 output logic                       spi_csn3_o,
75 output logic                       spi_oen0_o,
76 output logic                       spi_oen1_o,
77 output logic                       spi_oen2_o,
78 output logic                       spi_oen3_o,
79 output logic                       spi_sdo0_o,
80 output logic                       spi_sdo1_o,
81 output logic                       spi_sdo2_o,
82 output logic                       spi_sdo3_o,
83 input  logic                       spi_sdi0_i,
84 input  logic                       spi_sdi1_i,
85 input  logic                       spi_sdi2_i,
86 input  logic                       spi_sdi3_i
87   );
```

Beside the verbosity of this module it is important to understand that could be compressed logically in 4 sub parts:

- Generic: it is not directly part of communication as clock or reset and other configuration type signal .
- Cfg: The configuration signal is made to fix the behavior in the register and in the governing state machines.

- Cmd: It is the resultant of a specific action that impose an action using the control signal produced by the FSM.
- Data: It is the actual stream
- SPI : Is the SPI interface for controlling an SPI modules

Then inside the architecture declaration it is possible to see that there are some module

- udma_spim_reg_if
- udma_clkgen
- udma_dc_fifo
- io_tx_fifo
- udma_dc_fifo
- io_tx_fifo
- udma_dc_fifo
- udma_spim_ctrl
- udma_spim_txrx
- edge_propagator

All this structural module are functional to transmit data down to an SPI wiring.
Indeed, in the udma_spim_reg_if is the most important component because control the behavior of the system with this command specification:

Command name	**Command field**	**Description**
SPI_CMD_CFG	0x0	Sets the configuration for the SPI Master IP
SPI_CMD_SOT	0x1	Sets the Chip Select (CS)
SPI_CMD_SEND_CMD	0x2	Transmits up to 16bits of data sent in the command
SPI_CMD_DUMMY	0x4	Receives a number of dummy bits
SPI_CMD_WAIT	0x5	Waits an external event to move to the next instruction
SPI_CMD_TX_DATA	0x6	Sends data (max 256Kbits)
SPI_CMD_RX_DATA	0x7	Receives data (max 256Kbits)
SPI_CMD_RPT	0x8	Repeat the commands until RTP_END for N times
SPI_CMD_EOT	0x9	Clears the Chip Select (CS)
SPI_CMD_RPT_END	0xA	End of the repeat loop command
SPI_CMD_RX_CHECK	0xB	Checks up to 16 bits of data against an expected value
SPI_CMD_FULL_DUPL	0xC	Activate full duplex mode
SPI_CMD_SETUP_UCA	0xD	Sets address for uDMA tx/rx channel
SPI_CMD_SETUP_UCS	0xE	Sets size and starts uDMA tx/rx channel

All the hexadecimal command could be sent through the spi command from outside the SPI core. This register are placed and controlled by the uDMA core.

CHAPTER 4

Results and Discussion

We will now implement the proposed configuration and analyze the results obtained from simulations and tests to evaluate its effectiveness. The implementation involves designing and configuring the system, integrating components, and testing for functionality and performance. The analysis includes evaluating the system's performance, reliability, and scalability. The discussion of results highlights strengths, weaknesses, and recommendations for future improvements. Overall, the implementation and analysis provide valuable insights into the proposed configuration's potential applications and limitations.

4.1 Results

Pulpissimo simulation toolchain

Now that almost all the single module are studied analyzed and simulated it comes the time to spend some effort to the final procedure to simulate and build the simulation.
Since this operation was made in a remote Ubuntu 18.04 server that was configured whit module feature we need in first place call the modules that we need to load the correct specification.

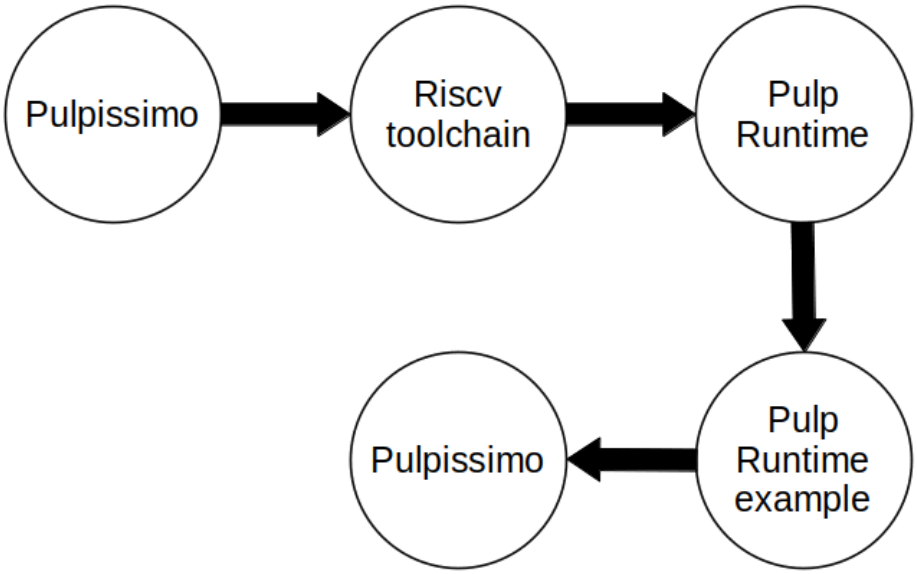

Figure 4.1: Pulpissimo Toolchain

The relative bash code could be seen here below

```
module load tools/mentor/all
module unload tools/mentor/catapult/2022-23

export PULP_RISCV_GCC_TOOLCHAIN=/home/damone/riscv-nn-toolchain
export PATH=$PULP_RISCV_GCC_TOOLCHAIN/bin:$PATH



cd pulpissimo
make checkout
source setup/vsim.sh
make build
cd

cd pulp-runtime
source configs/pulpissimo.sh
cd




# SPI regression test
cd regression_tests/peripherals/spim_flash
make clean all
make run gui=1 bootmode=fast_debug
cd
```

4.1.1 Hello pulpissimo

Once the last command is sent the Questasim will load the example contained in the runtime example. Here below the content:

```
#include <stdio.h>

int main()
{
  printf("Hello Pulpissimo!\n");

  return 0;
}
```

And the result is linear with the output, indeed write the printf on the Questasim transcript. This introductory example is not enough to show the power of PULP platform we need something that go down on the uDMA

4.1.2 UART pulpissimo

```
#include <stdio.h>
#include <pulp.h>
#include <stdint.h>

#define BUFFER_SIZE 256

uint8_t tx_buffer[BUFFER_SIZE];


int main()
{
  printf("Entering test\n");

  if (uart_open(0, 115200))
    return -1;


```

```
18      for (int i=0; i<BUFFER_SIZE; i++)
19      {
20        tx_buffer[i] = i;
21      }
22
23      uart_write(0, tx_buffer, BUFFER_SIZE);
24
25      uart_close(0);
26
27      return 0;
28    }
```

This snippet demonstrates how we can use PULP Platform's UART interface.
To fully understand the flow we need to go in deep on the description of what the code does: On first move the code includes the header files for using the PULP Platform as well as the standard C library . Then we will declare BUFFER_SIZE with a value of 256 and declares an array namely tx_buffer to hold the data that will be sent over the UART interface. Then we are moving in the main with a check if the uart is opend in that case we call an error. In next we are going to call a for loop used to fill the tx_buffer array with consecutive numbers from 0 to 255 and later we call the uart_write() function to send the contents of tx_buffer over the UART interface down to uDMA.
Regarding the procedure the most important and central function is related to the uart_write() function: the first argument is the ID of the UART interface that we want to use and the second argument is a pointer to the data to be sent (tx_buffer), and the third argument is the number of bytes to be sent. This simple example is designed to show how the Pulpissimo manage the uDMA.

For demonstration reason we have reduced the data send from 256 to only 10 and the result could be seen in next two figure

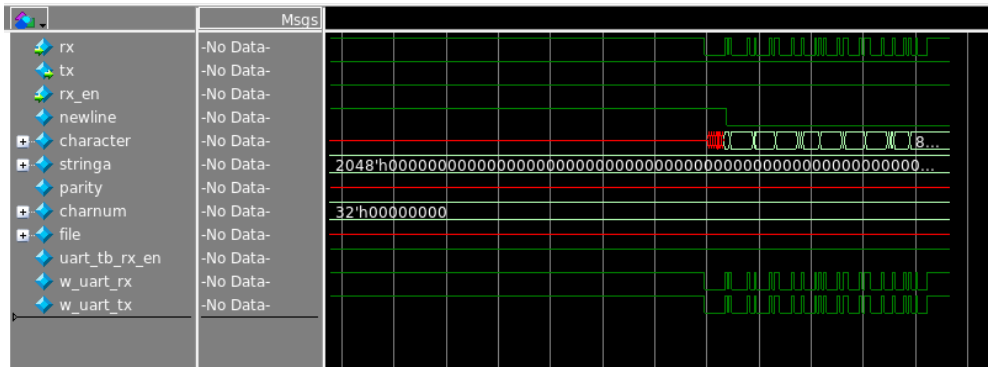

Figure 4.2: UART sending

4.1.3 Regression test of SPI

The next section we are going to view a more complex example that was pointed by the authors: The regression test example. here below we print the bash commands

```
1  module load   tools/mentor/all
2  module unload tools/mentor/catapult/2022-23
3
4  export PULP_RISCV_GCC_TOOLCHAIN=/home/damone/riscv-nn-toolchain
5  export PATH=$PULP_RISCV_GCC_TOOLCHAIN/bin:$PATH
6
7
8
9  cd pulpissimo
10 make checkout
11 source setup/vsim.sh
12 make build
```

```
cd

cd pulp-runtime
source configs/pulpissimo.sh
cd




# SPI regression test
cd regression_tests/peripherals/spim_flash
make clean all
make run gui=1 bootmode=fast_debug
cd
```

And here the code++

```
  #include <stdio.h>
  #include "pulp.h"
  #include "flash_page.h"

  #define REG_PADFUN0_OFFSET  0x10
  #define REG_PADFUN1_OFFSET  0x14
  #define REG_PADFUN2_OFFSET  0x18
  #define REG_PADFUN3_OFFSET  0x1C
  #define REG_PADCFG0_OFFSET  0x24
  #define REG_PADCFG1_OFFSET  0x28
  #define REG_PADCFG2_OFFSET  0x2C
  #define REG_PADCFG3_OFFSET  0x30
  #define REG_PADCFG4_OFFSET  0x34
  #define REG_PADCFG5_OFFSET  0x38
  #define REG_PADCFG6_OFFSET  0x3C
  #define REG_PADCFG7_OFFSET  0x40
  #define REG_PADCFG8_OFFSET  0x44
  #define REG_PADCFG9_OFFSET  0x48
  #define REG_PADCFG10_OFFSET 0x4C
  #define REG_PADCFG11_OFFSET 0x50
  #define REG_PADCFG12_OFFSET 0x54
  #define REG_PADCFG13_OFFSET 0x58
  #define REG_PADCFG14_OFFSET 0x5C
  #define REG_PADCFG15_OFFSET 0x60

  #define OUT 1
  #define IN  0

  #define BUFFER_SIZE 16

  #define TEST_PAGE_SIZE 256

  int pad_fun_offset[4] = {REG_PADFUN0_OFFSET,REG_PADFUN1_OFFSET,
      REG_PADFUN2_OFFSET,REG_PADFUN3_OFFSET};
  int pad_cfg_offset[16] = {REG_PADCFG0_OFFSET,REG_PADCFG1_OFFSET,
      REG_PADCFG2_OFFSET,REG_PADCFG3_OFFSET,REG_PADCFG4_OFFSET,
      REG_PADCFG5_OFFSET,REG_PADCFG6_OFFSET,REG_PADCFG7_OFFSET,
      REG_PADCFG8_OFFSET,REG_PADCFG9_OFFSET,REG_PADCFG10_OFFSET,
      REG_PADCFG11_OFFSET,REG_PADCFG12_OFFSET,REG_PADCFG13_OFFSET,
      REG_PADCFG14_OFFSET,REG_PADCFG15_OFFSET};

  static inline void wait_cycles(const unsigned cycles)
  {
  [...]
  }



  uint32_t configure_gpio(uint32_t number, uint32_t direction, uint32_t
      alternate){
    uint32_t which_reg_fun = number / 16; //select the correct register
```

```
45      uint32_t address = ARCHI_APB_SOC_CTRL_ADDR + pad_fun_offset[which_reg_fun
            ];
46
47      //--- set alternate 1/2/3 on GPIO
48      uint32_t value_wr = pulp_read32(address);
49      value_wr |= ((alternate & 0x00000003) << ((number - which_reg_fun*16)*2));
50      pulp_write32(address, value_wr);
51
52      //--- set GPIO
53      if(number < 32)
54      {
55        if (direction == OUT)
56        {
57          [...]
58        }
59      }else{
60        if (direction == OUT)
61        {
62          [...]
63
64        }
65      }
66
67      while(pulp_read32(address) != value_wr);
68
69    }
70
71    uint32_t set_gpio(uint32_t number, uint32_t value){
72      uint32_t value_wr;
73      uint32_t address;
74      if (number < 32)
75      {
76        address = ARCHI_GPIO_ADDR + GPIO_PADOUT_OFFSET;
77        value_wr = pulp_read32(address);
78        if (value == 1)
79         {
80          value_wr |= (1 << (number));
81         }else{
82          value_wr &= ~(1 << (number));
83         }
84        pulp_write32(address, value_wr);
85      }else{
86        address = ARCHI_GPIO_ADDR + GPIO_PADOUT_32_63_OFFSET;
87        value_wr = pulp_read32(address);
88        if (value == 1)
89         {
90          value_wr |= (1 << (number % 32));
91         }else{
92          value_wr &= ~(1 << (number % 32));
93         }
94        pulp_write32(address, value_wr);
95      }
96
97      while(pulp_read32(address) != value_wr);
98    }
99
100   uint32_t get_gpio(uint32_t number){
101     uint32_t value_rd;
102     uint32_t address;
103     if (number < 32)
104     {
105       address = ARCHI_GPIO_ADDR + GPIO_PADIN_OFFSET;
106       value_rd = pulp_read32(address);
107     }else{
108       address = ARCHI_GPIO_ADDR + GPIO_PADIN_32_63_OFFSET;
109       value_rd = pulp_read32(address);
110     }
111     return value_rd & (1 << (number % 32));
```

```
112   }
113
114   int main()
115   {
116
117   int error = 0;
118
119   //--- refer to this manual for the commands
120   //--- https://www.cypress.com/file/216421/download
121
122   //--- command sequence
123   int tx_buffer_cmd_program[BUFFER_SIZE] = {SPI_CMD_CFG(1,0,0),
124                                             SPI_CMD_SOT(0),
125                                             SPI_CMD_SEND_CMD(0x06,8,0),
126                                             SPI_CMD_EOT(0,0),
127                                             SPI_CMD_SOT(0),
128                                             SPI_CMD_SEND_CMD(0x12,8,0),
129                                             SPI_CMD_TX_DATA(4,4,8,0,0), //---
                                                  write 4B addr to the addr
                                                  buffer (first 4 bytes of the "
                                                  page" array)
130                                             SPI_CMD_TX_DATA(TEST_PAGE_SIZE,
                                                  TEST_PAGE_SIZE,8,0,0), //---
                                                  write 256B page data to the
                                                  page buffer
131                                             SPI_CMD_EOT(0,0)};
132
133   int addr_buffer[4] = {0x00,0x00,0x00,0x00}; //--- reading address
134   int tx_buffer_cmd_read[BUFFER_SIZE]     = {SPI_CMD_CFG(1,0,0),
135                                             SPI_CMD_SOT(0),
136                                             SPI_CMD_SEND_CMD(0x13,8,0), //---
                                                  read command
137                                             SPI_CMD_TX_DATA(4,4,8,0,0), //---
                                                  send the read address
138                                             SPI_CMD_RX_DATA(TEST_PAGE_SIZE,
                                                  TEST_PAGE_SIZE,8,0,0),
139                                             SPI_CMD_EOT(0,0)};
140
141   int rx_page[TEST_PAGE_SIZE];
142   int tx_buffer_cmd_read_WIP[BUFFER_SIZE] = {SPI_CMD_CFG(1,0,0),
143                                              SPI_CMD_SOT(0),
144                                              SPI_CMD_SEND_CMD(0x07,8,0),
145                                              SPI_CMD_RX_DATA(1,1,8,0,0),
146                                              SPI_CMD_EOT(0,0)};
147
148   int u = 0; //--- select spi0
149
150
151   printf("[%d, %d] Start test flash page programming over qspi %d\n",
          get_cluster_id(), get_core_id(),u);
152
153   configure_gpio(28,OUT,1); //--- using this GPIO as CS for the flash
154   set_gpio(28,1);
155
156   //--- enable all the udma channels
157   plp_udma_cg_set(plp_udma_cg_get() | (0xffffffff));
158
159   //--- get the base address of the SPIMx udma channels
160   unsigned int udma_spim_channel_base = hal_udma_channel_base(UDMA_CHANNEL_ID(
          ARCHI_UDMA_SPIM_ID(u)));
161   set_gpio(28,0);
162   printf("uDMA spim%d base channel address %8x\n", u,udma_spim_channel_base);
163   set_gpio(28,1);
164
165   //--- write the flash page
166   plp_udma_enqueue(UDMA_SPIM_TX_ADDR(u) , (int)page           ,TEST_PAGE_SIZE
          *4 + 4*4, UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
167   plp_udma_enqueue(UDMA_SPIM_CMD_ADDR(u), (int)tx_buffer_cmd_program , 68,
```

```
      UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
168
169   //--- wait until the page is written (we could use the WIP bit instead of
          waiting)
170   wait_cycles(50000);
171
172   //--- try to read back data
173   plp_udma_enqueue(UDMA_SPIM_RX_ADDR(u) , (int)rx_page      , TEST_PAGE_SIZE
          *4, UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
174   plp_udma_enqueue(UDMA_SPIM_TX_ADDR(u) , (int)addr_buffer     , 4*4,
          UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
175   plp_udma_enqueue(UDMA_SPIM_CMD_ADDR(u), (int)tx_buffer_cmd_read , 26,
          UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
176
177   wait_cycles(50000);
178
179   for (int i = 0; i < TEST_PAGE_SIZE; ++i)
180   {
181     printf("read %8x, expected %8x \n",rx_page[i],page[i+4]);
182     if (rx_page[i] != page[i+4])
183
184     if (error == 0)
185     {
186       printf("TEST SUCCEDED\n");
187     }else{
188       printf("TEST FAILED with %d errors\n", error);
189     }
190
191     return error;
192   }
```

This code defines several memory-mapped registers and provides a function called "configure_gpio" that sets up GPIO pins on the pulp to be able to produce the relative output . The register offsets are stored in arrays called "pad_fun_offset" and "pad_cfg_offset" and are used for control the overall configuration of the pins regarding direction and configuration.

wait_cycles method

```
1   static inline void wait_cycles(const unsigned cycles)
2   {
3     register unsigned threshold;
4     asm volatile("li %[threshold], 4" : [threshold] "=r" (threshold));
5     asm volatile goto("ble %[cycles], %[threshold], %l2"
6             : /* no output */
7             : [cycles] "r" (cycles), [threshold] "r" (threshold)
8             : /* no clobbers */
9             : __wait_cycles_end);
10    register unsigned i = cycles >> 2;
11  __wait_cycles_start:
12    // Decrement `i` and loop if it is not yet zero.
13    asm volatile("addi %0, %0, -1" : "+r" (i));
14    asm volatile goto("bnez %0, %l1"
15            : /* no output */
16            : "r" (i)
17            : /* no clobbers */
18            : __wait_cycles_start);
19  __wait_cycles_end:
20    return;
21  }
```

The purpose of the function is to introduce a delay of a specified number of clock cycles declared in the argument of this function.

As first thing we are setting a threshold to 4 to ensure that the loop is not less then a basic 4 clok instruction. if the number of clock cycles that is less than 4, as entering the loop with an insufficient number of cycles will cause an underflow on the first subtraction

operation. Later we branch to a label (__wait_cycles_end) if the cycles argument is less than or equal to threshold: if the branch is not taken, the function initializes a local variable i to cycles shifted right by 2 bits (which is equivalent to dividing cycles by 4) if is taken will proceed to a loop.
The central loop controlled by the variable named "i" will branches to __wait_cycles_start if i is not yet zero and ,if so, the loop terminates and the function returns.

GPIO config

```
1  //--- set GPIO
2  if(number < 32)
3  {
4    if (direction == OUT)
5    {
6      //--- enable GPIO
7      address = ARCHI_GPIO_ADDR + GPIO_GPIOEN_OFFSET;
8      value_wr = pulp_read32(address);
9      value_wr &= ~(1 << number);
10     pulp_write32(address, value_wr);
11     //--- set direction
12     address = ARCHI_GPIO_ADDR + GPIO_PADDIR_OFFSET;
13     pulp_write32(address, value_wr);
14   }else if (direction == IN){
15     //--- enable GPIO
16     address = ARCHI_GPIO_ADDR + GPIO_GPIOEN_OFFSET;
17     value_wr = pulp_read32(address);
18     value_wr |= (1 << number);
19     pulp_write32(address, value_wr);
20     //--- set direction
21     address = ARCHI_GPIO_ADDR + GPIO_PADDIR_OFFSET;
22     pulp_write32(address, value_wr);
23   }
24 }else{
25   if (direction == OUT)
26   {
27     //--- enable GPIO
28     address = ARCHI_GPIO_ADDR + GPIO_GPIOEN_32_63_OFFSET;
29     value_wr = pulp_read32(address);
30     value_wr &= ~(1 << (number-32));
31     pulp_write32(address, value_wr);
32     //--- set direction
33     address = ARCHI_GPIO_ADDR + GPIO_PADDIR_32_63_OFFSET;
34     pulp_write32(address, value_wr);
35   }else if (direction == IN){
36     //--- enable GPIO However this example was taken in consideration
           because it shows us how is possible to load a firmware
37     using pulp method.
38     address = ARCHI_GPIO_ADDR + GPIO_GPIOEN_32_63_OFFSET;
39     value_wr = pulp_read32(address);
40     value_wr |= (1 << (number-32));
41     pulp_write32(address, value_wr);
42     //--- set direction
43     address = ARCHI_GPIO_ADDR + GPIO_PADDIR_32_63_OFFSET;
44     pulp_write32(address, value_wr);
45   }
46 }
```

This code is for setting the direction GPIO pin, and it is sensible to the GPIO pin number and the direction to be set. If the GPIO pin number is less than 32 we will use the lower range of GPIO registers to set the direction while If the number is greater than or equal to 32 we are going to use the upper range of GPIO registers. In the first part we enable the GPIO by reading the GPIO enable register, we are modifying the appropriate bit to enable the selected GPIO pin, and then writing the modified value back to the register. When all this step is done we are going to sets the direction of the GPIO pin by writing the enable

register value to the appropriate GPIO direction register.

Central sending unit

This part that appear on the bottom is the core of this example and it is formed by three element:

- tx_buffer_cmd_program
- tx_buffer_cmd_read

in detail:

```
1 int tx_buffer_cmd_program[BUFFER_SIZE] = {SPI_CMD_CFG(1,0,0),
2                                           SPI_CMD_SOT(0),
3                                           SPI_CMD_SEND_CMD(0x06,8,0),
4                                           SPI_CMD_EOT(0,0),
5                                           SPI_CMD_SOT(0),
6                                           SPI_CMD_SEND_CMD(0x12,8,0),
7                                           SPI_CMD_TX_DATA(4,4,8,0,0), //---
                                                write 4B addr to the addr buffer
                                                 (first 4 bytes of the "page"
                                                array)
8                                           SPI_CMD_TX_DATA(TEST_PAGE_SIZE,
                                                TEST_PAGE_SIZE,8,0,0), //---
                                                write 256B page data to the page
                                                 buffer
9                                           SPI_CMD_EOT(0,0)};
```

This code defines an array tx_buffer_cmd_program that contains a sequence of SPI for programming the memory L2. The commands are defined using macros that take in various parameters, such as the opcode, data size and transmission settings in this sequence :

- Configure the SPI controller for single I/O mode.
- Start of transmission.
- Send a write enable command (opcode 0x06).
- End of transmission.
- Start of transmission.
- Send a page program command (opcode 0x12).
- Transmit the 4-byte address of the page to be programmed.
- Transmit the page data.
- End of transmission.

All this setting are written to the tx_buffer_cmd_program array in the order that they are intended to be executed by the SPI controller. This code is setting up a buffer for SPI commands to read data from an SPI flash memory device.

```
1 int addr_buffer[4] = {0x00,0x00,0x00,0x00}; //--- reading address
2 int tx_buffer_cmd_read[BUFFER_SIZE]     = {SPI_CMD_CFG(1,0,0),
3                                            SPI_CMD_SOT(0),
4                                            SPI_CMD_SEND_CMD(0x13,8,0), //---
                                                 read command
5                                            SPI_CMD_TX_DATA(4,4,8,0,0), //---
                                                 send the read address
6                                            SPI_CMD_RX_DATA(TEST_PAGE_SIZE,
                                                 TEST_PAGE_SIZE,8,0,0),
7                                            SPI_CMD_EOT(0,0)};
```

The tx_buffer_cmd_read buffer is used to store a sequence of commands that will be sent to the SPI flash memory to read data from a specific address in the memory. The buffer is initialized with the following commands:

- SPI_CMD_CFG(1,0,0) configures the SPI interface to have 1 bit per transfer and sets the Chip Select (CS) to be active low.
- SPI_CMD_SOT(0) indicates that the start of transfer signal should not be sent.
- SPI_CMD_SEND_CMD(0x13, 8, 0) sends the read command to the memory device, which is represented by the hex value 0x13 and the 8 indicates that the command is 8 bits long, and the final 0 indicates that there is no delay after the command is sent.
- SPI_CMD_TX_DATA(4, 4, 8, 0, 0) sends the address to read from, which is stored in the addr_buffer array. The first 4 specifies the length of the data being sent (4 bytes), the second 4 specifies the number of bits per transfer (4 bytes x 8 bits per byte = 32 bits), and the final 0, 0 indicate that there is no delay after the data is sent.
- SPI_CMD_RX_DATA(TEST_PAGE_SIZE, TEST_PAGE_SIZE, 8, 0, 0) receives the data from the SPI flash memory device. TEST_PAGE_SIZE is the size of the data being read, and is likely set to 256 bytes (the size of a page on many SPI flash memory devices). The 8 specifies the number of bits per transfer, and the final 0, 0 indicate that there is no delay after the data is received.
- SPI_CMD_EOT(0,0) indicates the end of the transfer and that there is no delay after the end of transfer signal is sent.

```
1 int tx_buffer_cmd_read_WIP[BUFFER_SIZE] = {SPI_CMD_CFG(1,0,0),
2                                            SPI_CMD_SOT(0),
3                                            SPI_CMD_SEND_CMD(0x07,8,0),
4                                            SPI_CMD_RX_DATA(1,1,8,0,0),
5                                            SPI_CMD_EOT(0,0)};
```

This block is in charge to checking the status of the Write In Progress (WIP) bit in a flash memory device . To be more detailed the WIP bit indicates whether the flash memory device is currently busy with a write operation or not: If the WIP bit is set to 1, the device is busy and a write or erase operation cannot be initiated. Using a read command (opcode 0x07) to read the status register of the L2 we will first configure the SPI interface with a clock frequency of 1 MHz (SPI_CMD_CFG), then sends the read command (SPI_CMD_SEND_CMD), receives one byte of data (SPI_CMD_RX_DATA) and at the ends the transaction (SPI_CMD_EOT) performed.

main

Once all the register are setted we could Start test flash page programming over qspi inside the memory using this GPIO as CS for the flash and later we will enable all the uDMA channels and get the base address of the SPIMx udma channels that in our case is zero. On next process we have to

```
1 //--- write the flash page
2 plp_udma_enqueue(UDMA_SPIM_TX_ADDR(u), (int)page ,TEST_PAGE_SIZE*4
      + 4*4, UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
3 plp_udma_enqueue(UDMA_SPIM_CMD_ADDR(u), (int)tx_buffer_cmd_program , 68,
      UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32);
```

These two lines are used to enqueuing data to be sent over SPI using the pulp-rt driver.

The first plp_udma_enqueue is used to to add a new SPI transaction for transmitting with four arguments to the function:

- UDMA_SPIM_TX_ADDR(u): This is the address of the buffer to be transmitted. The UDMA_SPIM_TX_ADDR macro returns the base address of the TX buffer of the specified SPI device.
- (int)page: This is a pointer to the data to be transmitted, which in this case is the contents of the page buffer.
- TEST_PAGE_SIZE*4 + 4*4: This is the number of bytes to be transmitted. TEST_PAGE_SIZE*4 represents the size of the page buffer in bytes, and 4*4 represents the size of the addr_buffer in bytes.
- UDMA_CHANNEL_CFG_EN | UDMA_CHANNEL_CFG_SIZE_32: This configures the UDMA channel for the SPI transaction. The UDMA_CHANNEL_CFG_EN bit is set to enable the channel, and UDMA_CHANNEL_CFG_SIZE_32 indicates that the data is 32 bits wide.

The second line is enqueuing a SPI command transaction for programming the flash memory and have a similar behavior.

on last, we have the final check if the process have work correctly:

```
1   for (int i = 0; i < TEST_PAGE_SIZE; ++i)
2   {
3     printf("read %8x, expected %8x \n",rx_page[i],page[i+4]);
4     if (rx_page[i] != page[i+4])
5     {
6       error++;
7     }
8   }
9
10    if (error == 0)
11    {
12      printf("TEST SUCCEDED\n");
13    }else{
14      printf("TEST FAILED with %d errors\n", error);
15    }
16
17    return error;
18  }
```

This last part of the function is used to check if the data was sent correctly

A for loop is used to iterate over each element of the rx_page array as long as we reach all the data sent. The if statement checks whether rx_page[i] is equal to page[i+4]. If they are not equal, error is incremented.

After the loop, the function checks the value of error. If it is zero, the function prints "TEST SUCCEDED". Otherwise, it prints "TEST FAILED with the number of test failed. in short us a comparison test that is used to verify that some received data matches the expected 3data counting the number of errors encountered during the test, which can be used to determine whether the test was successful or not.

Below is possible to see the result on transmitting data using SPI.

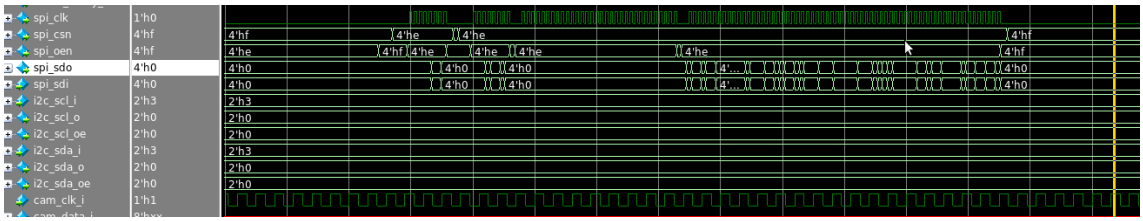

Figure 4.3: SPI result

However this example was taken in consideration because it shows us how is possible to load a firmware that is able to write data on L2 and then invoke the pulp_enqueue method to send through SPI the data

4.2 Discussion

The results just obtained reflect an aptitude of our system on the pulpissimo versality to any kind of application, although it is only a starting point the results are encouraging because you can clearly see how the modules work properly. Using generators it will be possible to build load models through the jtag to enter the parameters in the L2 and then be called on reckon when it is more appropriate

CHAPTER 5

Conclusion

This discussion was carried out in order to make an analytical study of the state of the art of neuromorphic processors and the state of the art of SOC platforms that could host them. After carefully choosing those components we ventured into the verification and inspection of most of the internal modules to discover how they are thought. All this process was done because we want to have a comprehensive and panoramic landscape of the technologies that are currently available in research area. Have in mind the initial goals we were able to choose the right combination (Reckon + pulpissimo) in order to obtain a SOC capable of being autonomous and power efficient.

The discovery and the work done in this thesis have realized that these technologies are the real future due low power consumption, robustness, dynamism and high level of performance. If we take in view the traditional processors it is possible to affirm that is not able to compete with this brand-new technologies and, if so, in a few years they will be an industry a new standard.

This preliminary work underlines how strong the Reckon model and pulpissimo are; However also demonstrates what are the points of improvement for a future work. In particular we are referring that in a possible upgrade the programming phase with a replacement of the SPI with a more performing communication model that allows a faster configuration of the neurons. One possible improvement is the usage of HyperBus as communication protocol: Compared to aged SPI, HyperBus supports higher data transfer rates and has a more efficient command structure, which can translate in our case in a faster data access to memory and improved system performance.
In addition, HyperBus, allows a direct memory access which reduces CPU overhead and can improve overall system efficiency.

Overall all these improvements make HyperBus a more efficient and faster communication protocol compared to SPI; Indeed this memory oriented protocol made possible the data transfer from L2 to the two internal SRAM more in a faster way.
In conclusion we could affirm that this preliminary work have show how the integration of these two processors could brings many benefits for applications that relies on battery consumption.
In summary this starting point will allow future studies to create a stable and dynamic platform capable of outclassing all the implementations created so far in the field of edge computing on embedded system platform.
In conclusion We conducted an extensive exploration of the various hardware modules integrated into the Pulp platform and Reckon, with particular attention paid to the uDMA within Pulpissimo. Our thesis work was focused on developing a robust and well-documented toolchain for launching and managing SPI communication on Pulpissimo.

Bibliography

[1] Charlotte Frenkel, Martin Lefebvre, Jean-Didier Legat, and David Bol. A 0.086-mm 12.7-pj/sop 64k-synapse 256-neuron online-learning digital spiking neuromorphic processor in 28-nm cmos. *IEEE transactions on biomedical circuits and systems*, 13(1):145–158, 2018.

[2] Charlotte Frenkel and Giacomo Indiveri. Reckon: A 28nm sub-mm2 task-agnostic spiking recurrent neural network processor enabling on-chip learning over second-long timescales. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3. IEEE, 2022.

[3] Joshua Mack, Ruben Purdy, Kris Rockowitz, Michael Inouye, Edward Richter, Spencer Valancius, Nirmal Kumbhare, Md Sahil Hassan, Kaitlin Fair, John Mixter, et al. Ranc: Reconfigurable architecture for neuromorphic computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(11):2265–2278, 2020.

[4] Amirreza Yousefzadeh, Gert-Jan Van Schaik, Mohammad Tahghighi, Paul Detterer, Stefano Traferro, Martijn Hijdra, Jan Stuijt, Federico Corradi, Manolis Sifalakis, and Mario Konijnenburg. Seneca: Scalable energy-efficient neuromorphic computer architecture. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 371–374. IEEE, 2022.

[5] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, 2019.

[6] F. Conti, P. D. Schiavone, and L. Benini. Xnor neural engine: A hardware accelerator ip for 21.6-fj/op binary neural network inference. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[7] Pasquale Davide Schiavone and The PULP team. Understanding and working with pulp. *Integrated sytem laboratory*, 13.06.2019.

[8] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 409–416, 2021.

[9] Patrick Schiavone et al. Risc-v microcontroller workshop - zurich 2019. `https://pulp-platform.org/docs/riscv_workshop_zurich/schiavone_wosh2019_tutorial.pdf`, 2019.

[10] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, pages 1–1, 2021.

[11] Xiaoqing Chang et al. Deep learning-based ultra-fast demodulation for high-speed optical communication systems. *arXiv preprint arXiv:1810.12436*, 2018.

[12] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE transactions on biomedical circuits and systems*, 12(1):106–122, 2017.

[13] Inc. Xilinx. Axi - advanced extensible interface. `https://www.xilinx.com/products/intellectual-property/axi.html`, 2022.

[14] openhwgroup. core-v-xif. `https://github.com/openhwgroup/core-v-xif`, 2022.

[15] Davide Rossi, Daniele Jahier Pagliari, Carlo Sau, Andrea Bartolini, and Luca Benini. Pulpino: an open-source ultra-low-power risc-v based microcontroller. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1579–1582. IEEE, 2015.

[16] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: an ultra-low-power pulpissimo soc in 22nm fdx. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, 2018.

[17] Andrea Bartolini, Antonio Pullini, Alessandro Capotondi, and Luca Benini. The pulp runtime: A task-based execution framework for parallel ultra-low-power computing. *IEEE Transactions on Computers*, 68(2):190–203, 2019.

[18] Cypress Semiconductor. Psoc 6 mcu: Psoc 63 with ble datasheet, 2019.

[19] GreenWaves Technologies. *GAP8 Hardware Reference Manual*. GreenWaves Technologies, Grenoble, France, 2021.

[20] IEEE Standards Association. *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE, Piscataway, NJ, USA, 2017.

[21] PULP Platform. PULP runtime examples, 2022.

[22] GreenWaves Technologies. Greenwaves technologies: Spi api documentation, 2022.

Acknowledgment

I would like to take this opportunity to express my gratitude to Gianvito and Yvan for their invaluable support throughout the course of my thesis.
I am hopeful that this work will serve as an excellent starting point for future research in this field.