Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2016-02

M.Sc. Thesis

Integrating a Neuron Network application into a ZYBO Zynq-7000 development board with an AXI-Bus interface

Mohammad Ahmadinia, B.Sc.

Abstract

In order to verify an application with a simulation, increasing the simulation speed is important. An Inferior Olivary Nucleus (ION) network has been implemented in a SystemC language and its simulation verified by a SystemC testbench. The goal of this thesis is to integrate the ION application into an FPGA in order to increase its simulation speed by using the hardware. In addition, to verify the integrated ION in the FPGA, an ARM processor is used for communication between a PC and the ION. A Zybo development board combines the capability of software programming in an ARM-based processor with the ability of the hardware programming in an FPGA, on a single device. To communicate between the ARM processor and the FPGA, AXI-Bus interface is implemented. The ARM processor executes software in order to send inputs from the PC to the ION and receive results from the ION and show them on the screen on the PC. To verify the ION in the FPGA, software is implemented and the software outputs are compared with the result of SystemC testbench reference model.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science Delft University of Technology

Integrating a Neuron Network application into a ZYBO Zynq-7000 development board with an AXI-Bus interface

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Mohammad Ahmadinia, B.Sc.
born in Malayer, Iran

This work was performed in:

Circuits and Systems Group
Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Integrating a Neuron Network application into a ZYBO Zynq-7000 development board with an AXI-Bus interface "** by **Mohammad Ahmadinia, B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: November 29^{th}, 2016

Chairman: _____
prof.dr.ir. Alle-Jan van der Veen

Advisor: _____
dr. ir. René van Leuken

Committee Members: _____
dr. Carlo Galuzzi

_____
dr. Amir Zjajo

_____
dr. Arjan van Genderen

Abstract

In order to verify an application with a simulation, increasing the simulation speed is important. An Inferior Olivary Nucleus (ION) network has been implemented in a SystemC language and its simulation verified by a SystemC testbench. The goal of this thesis is to integrate the ION application into an FPGA in order to increase its simulation speed by using the hardware. In addition, to verify the integrated ION in the FPGA, an ARM processor is used for communication between a PC and the ION. A Zybo development board combines the capability of software programming in an ARM-based processor with the ability of the hardware programming in an FPGA, on a single device. To communicate between the ARM processor and the FPGA, AXI-Bus interface is implemented. The ARM processor executes software in order to send inputs from the PC to the ION and receive results from the ION and show them on the screen on the PC. To verify the ION in the FPGA, software is implemented and the software outputs are compared with the result of SystemC testbench reference model.

Acknowledgments

I would like to thank my adviser dr. ir. René van Leuken for his assistance during this thesis. He always helped me whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever he thought I needed it. My special gratitude goes to the members of my MSc committee for accepting their role, reading my dissertation, and providing useful feedback and I would like to thank dr. Arian van Genderen for his support and trust for starting my Master study at TUDelft. On a personal note, I would like to thank all of my friends. My special thanks goes to my close friends, Saleh, Samira, Soran and Debarshi.

Last but not least, I am very grateful to dedicate this work to my wife, Somi, whose love and support gave me the motivation I needed to complete this work. I am also thankful to my parents for instilling in me a love for education, and providing me with countless opportunities to learn and grow throughout my lifetime.

Mohammad Ahmadinia, B.Sc.
Delft, The Netherlands
November 29^{th}, 2016

Contents

List of Figures

List of Tables

1 Introduction

This thesis describes an application developed on a Zybo Zynq-7000(xc7z010clg40) [5] development board device which integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA, on a single device. An Inferior Olivary Nucleus (ION) network has been implemented in a SystemC language and its simulation verified by a SystemC testbench [3]. This implementation is called a Neuron Network Application (NNA). The goal of this thesis is to integrate the NNA into an FPGA in order to increase its simulation speed by using the hardware. In addition, to verify the integrated NNA in the FPGA, an ARM processor is used for communication between a PC and the NNA. Using the Zybo board and using different software such as the Vivado Design Suite [6], the ModelSim [7] and the Xilinx System Development Kit (SDK) helped us to create the customized slave IP from integrated NNA with an AXI-bus interface which is called Neuron Network IP-core (NNIP). The NNIP verification is done with writing a VHDL testbench and compare it with a reference SystemC testbench model. After verification of NNIP, a block design is created to connect ARM processor to the NNIP in the FPGA through the AXI interconnect. To verify the block design, writing a software is needed. The software is executed on the ARM processor to send input parameters to the NNIP in the FPGA and return result outputs from the FPGA and show it on the desktop screen.

1.1 Goals

The first goal of this thesis is to increase the simulation speed of ION by integrating ION implementation in the FPGA. To achieve this goal, a customized slave IP (called the Neural Network IP-core (NNIP)) is created to be utilized to integrate the Neuron Network Application (NNA) with the AXI-Bus interface architectures. The NNA model has to be understood, and the AXI-Bus interface architecture and its protocols are crucial concepts to be studied.

The second goal of this thesis is to verify and evaluate the NNIP within a VHDL testbench with the ModelSim tool and also creating a hardware block design in the Xilinx Vivado (2015.1). In order to create a block design, the Zynq-7000 processors should be used as a master to connect with the NNIP and then the block design should be synthesized and the bit-stream will be generated. To evaluate the block design, the Xilinx Software Development Kit (SDK) should be used to write a software driver based on the hardware specification and implementation.

Finally, the software result will be compared with the NNA SystemC testbench result [3] based on the same input parameters and the same conditions.

1.2 Approach

The approach of this thesis for reaching the goals in the previous section begins with the literature studies about the current state-of-the-art of a brain and a neural network. The previous work about designing the Inferior Olivary Nucleus (ION) network on an FPGA device [3] as an initial work for this thesis is investigated. In addition, good understanding about the Advanced Extensible Interface (AXI) bus protocol and the different software and hardware platform are essential.

1.3 Contributions

The contributions of the work presented in this thesis are the following:

- Integrating a Neuron Network Application (NNA) with a new customized slave IP-core (it is called the Neuron Network IP, NNIP) which has the AXI-Bus interface and then creating a package IP.
- Verification and evaluation the NNIP with a VHDL testbench and comparing the result with the reference model(SystemC testbench).
- Using the Vivado to create a block design and adding the Zynq-7000 processors as a master to connect with the NNIP, and finally, creating an HDL wrapper, and then synthesizing, implementing, and creating a bit-stream for the design.
- Exporting the hardware design specification and launching the Xilinx Software Development Kit (SDK) to write a software. This software will be written for Verification and comparing the results of the hardware design with the reference SystemC testbench model.

1.4 Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 Describes the state of the art of the brain, the neural and the Neuron Network Application (NNA). In addition, it presents a brief introduction to the software and the hardware which are used in this study and the introduction of the AXI-Bus for the Xilinx System Development.

Chapter 3 Introduces how the system is designed to integrate the NNA with a new IP-core and communicate through the AXI-Bus interface with the ARM Cortex-A9 processor.

Chapter 4 Shows the system design implementation. It will start with the simulation then the synthesizing, the optimization, the Bit generation and the hardware and the software verification and finally comparing the result with the reference SystemC testbench model.

Chapter 5 The conclusion and the future Work will be discussed and presents how our design can be further improved.

2 State of the Art

This chapter introduces the concept of the *Neurul Network*(NN) in the brain. In addition, the different software and hardware which are used are summarized and their limitation and capability are explained. In response to the software, the Vivado Design Suite [6], the ModelSim [7], the Xilinx System Development Kit (SDK) and the picocom terminal program are used and their contribution is highlighted. Regarding to the hardware, the ZYBO Zynq-7000(xc7z010clg40) [5] development board is explained. Further, the concept of the AXI interface is discussed in this chapter.

2.1 Brain And Neurons

In this section a short introduction to the *Neurul Network*(NN) is given. A top-down method is used: First, the brain itself is discussed. Second, the place of the cerebellum in the brain is covered. Third, the physical structure of the cerebellum is included, also illustrating the underlying cell connections in this section. Subsequently, the cell structure of the *Inferior Olivary Nucleus* (ION) is presented.

2.1.1 The Brain

The brain consists of three main parts, the Cerebrum, the Cerebellum and the Brainstem (see Figure 2.1 for their location in the human brain). The Cerebrum is the outer layer of the biggest part, and it is called the cerebral cortex, it contains about half of all the neurons in the brain[8]. This part is responsible for three main brain functions; specifically (1) perception: the ability to see, feel, taste, hear and smell the environment to maintain a reaction (2) the cognitive functions: thinking, feeling, and intuition and (3) motor control: planning and control of all intentional movements is done by the motor cortex which is in the cerebral cortex [[9], p. 322].

The second part of the brain is the brainstem. It has vital life functions such as maintaining, breathing, heartbeat, and the controlling the blood pressure. Nevertheless, the IONs are located in the brainstem and, take an important role in the motor control[10].

The cerebellum is the third part of the brain which occupies around nine times less volume compared to the Cerebrum, but it contains a roughly the same amount of neurons. Its functionality is well known because it includes only two neuron types: Purkinje cells and Granule cells[11].

2.1.2 The Cerebellum

The cerebellum is a sectors of the brain that takes an important function in the motor control as well as attention and language, pleasure responses [12], and movement related

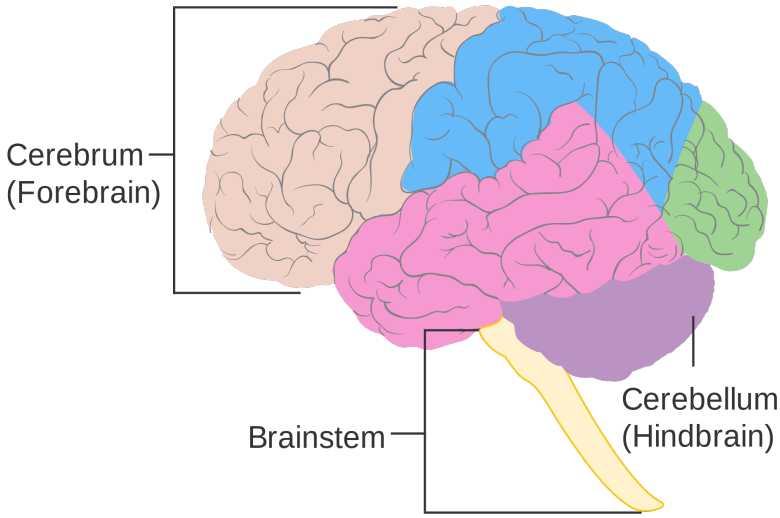

Figure 2.1: Human Brain

functions. The cerebellum does not start the movement but provides the coordination, the precision, and the accurate timing. It collects input from sensory systems of the spinal cord and other parts of the brain and integrates these inputs to the fine-tune motor activity [13].

To control these different motor systems, the cerebellum, receives many input signals such as (1) the somatosensory input from different systems in the body that are sensitive to touch, (2) the spinal cord and (3) the cerebral cortex [14].

2.1.3 Neural networks

The Cells inside the brain consist of the neurons which are used to process and transmit signals. Every neuron mainly consists of three parts (Figure 2.2, A): dendrites,soma, and axon.

The electrochemical stimuli are received by the dendrite as an input from other neurons. This information is given to the soma and after processing and storing as a membrane potential, if this potential reaches a certain threshold, a spike goes via the axon to other neurons (Figure2.2, B).

As a biological function, the *Inferior Olive* (IO) performs within the brainstem. The IO is the part of the Olive Body located in the brainstem and works closely together with the Cerebellum to provide the correct motor control. There are two Olive Bodies in the brainstem, each having their own IO. Both IOs consists of neural cells that will be introduced as an *Inferior Olive Neurons* (ION).

The location of the neurons is defined in the cerebellum and the Inferior Olivary

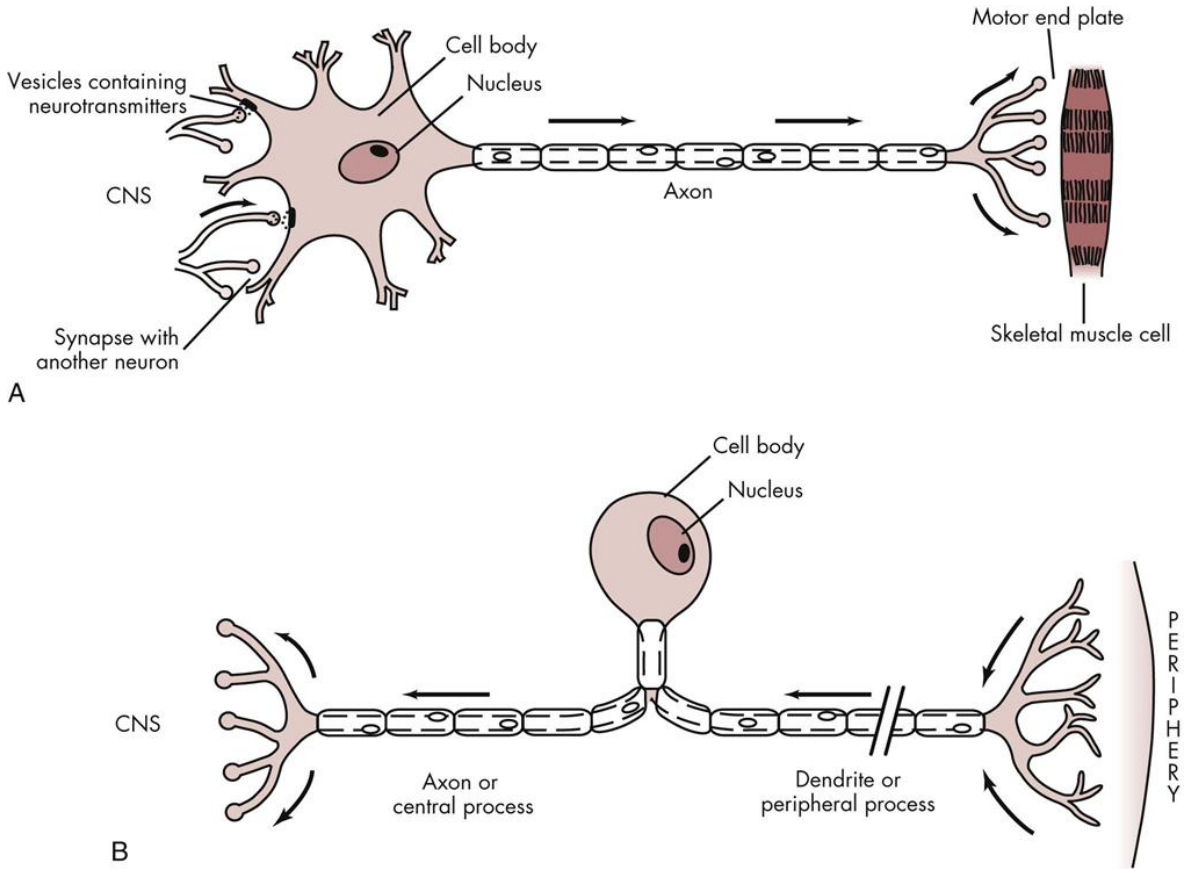

Figure 2.2: Simplified image of a neuron[1]

Nucleus (ION) and, it is essential for the coordination of the body's activities [15].

Each ION is interconnected within synapses, and receives the stimuli through its dendrites although there is a difference between the connected synapse and its dendrite. Together, they transfer the input to the cell body (soma) via the axon and the axon traverses via the ION or towards the Cerebellum ending at the axon terminal, and towards several Synapses (see Figure 2.2, A). The IO works in a close cooperation together with the Cerebellum to provide an accurate motor control. When a person wants to move part of his/her body a request is sent from the Cerebrum. This request is also copied to the cerebellum through the Pons. At the same time, the sensory information received from the body will be fed up via the IO. After receiving this input, the IO reply to the cerebellum, via the output axons (so called Climbing Fibers). Then, the cerebellum compares the command and response, and if it is necessary, the requested movement is adjusted to fit the command. This affects on the brains function to perform reflexes, and quickly generate accurately [16].
The neural networks have been investigated by researchers for several decades. The experiments performed by Hodgkin and Huxley in 1952 [17], gave a way to a clearer understanding and precise modeling of the action potential generation by the neurons and also understanding the importance of bio-potentials in the transmission of infor-

mation in the Central Nervous System (CNS).

Over the years, several NN models have been proposed in [18] and implemented in hardware to verify their accuracy [19]. Due to the high level of parallelism in the neuron networks, reconfigurable hardware, such as a Field Programmable Gate Array (FPGA), can provide the means to properly simulate these intricate and highly parallel networks. However, modern multi-core designs have also proved to be capable of simulating large NN's within a given time. The previous work gave a brief description of how the biochemical neuron was translated into the a logical model that can be implemented on an Integrated Circuit (IC).

2.2 Neuron Network Application (NNA)

In this section Master thesis of three students as the initial work are described.

2.2.1 Introduction

In a work done by [20], by using an extended version of the HH, a neuron can be split in 3 different compartments. Each compartment, models the behaviour of a certain area of the ION (dendrite, some and axon Hillock). The extended HH was first described in a SystemC and then translated to a Synthesizable-SystemC (SSC) [21] model.

A SystemC [22] is a modern derivative of the general purpose C++ programming language, built for the simulating event-driven processes. By designing the model on a special subset of the SystemC, modern tools [6] can create a synthesizable hardware. This in turn made it possible to translate the complex and high level HH calculations into parallel operations that could be implemented on an Field Programmable Gate Array (FPGA).

In the SSC model, by designing a system, all the ION HH calculations were put into a single piece of hardware called a Physical Cell (PhyC), multiple HH calculations could run in parallel. Then, as the time it took for 1 PhyC to finish calculating a single HH response was shorter than the real-time timing constraint (50µs), several iterations of HH calculations could be calculated over a single PhyC. Each iteration represents a single simulated neuron that is referred to as a Simulated Cell (SimC).

To confer the coupling effect between the SimC, and to give an output to the simulated neuron's axon (by the axon hillock), each PhyC is connected to a Wishbone bus [23]. As only 1 PhyC can send information at a time through the bus, a Wishbone arbiter decides which PhyC has access.

2.2.2 Problem Definition

By using the SSC, and by connecting eight PhyCs to the Wishbone bus, up to 48 neurons could be implemented on a Virtex7 FPGA, and simulated within the 50µs time limit. The larger numbers of SimC are not possible due to the limitations of the Wishbone bus.

In the follow up work by [17] the new method has been identified to interconnect the simulated neuron cells in a way that allows for "massive" numbers of cells compared

to the previous approaches. An increase of the number of the cells in the system should not increase the communication time required by the system to handle all the communication between the cells. Furthermore, the system should be expandable to more than one chip as every chip has a certain limit in size. To overcome this limitation, it should be able to connect multiple chips in order to build a larger system.

To accurately simulate the behaviour of a highly parallel cognitive system such as the IO, a bio-physically meaningful model is chosen that closely resembles the biological responses in the human brain. The extended Hodgkin-Huxley model (HH) describes the relation between the electric current to a single neuron membrane, and its capacitance. This relation is translated into nonlinear differential gap functions that describe the responses of three main parts of a neuron (dendrite, soma and axon)[3]. These functions rely a great deal on accurate floating point operations, and in particular the exponent operation, as ionic currents in biological neurons follow a exponent trend. Within the HH the exponent operation is only used 30 times per neuron calculation. Compared to other more often used operations, the exponent operations requires relatively more resources and cycles to complete. Within the NN there exists a high level of connectivity between separate cells. For increasing the complexity, the communication load can increase exponentially the resulting in non-real-time simulation times.

2.2.3 Approach

The Neuron Network (NN) is implemented on a Field Programmable Gate Array (FPGA) with the help of modern tools and, a mix of open and closed source IP. This implementation is called a Neuron Network Application (NNA). Each calculation instance implemented on the FPGA represents a single neuron within the NN. By scheduling the calculation elements around the exponential operation, resources are spared while the required amount of cycles are kept to a minimum. Two approaches are taken to communicate between the neurons. First on a local level the elements are connected to a bus. Secondly these locally connected neurons (cluster) are interfaced in a binary tree network with routers. To verify how accurate the model is, a reference model is used to generate neuron responses for a comparative system.

2.3 Software Tools

In this section, software tools which are used in this thesis will be described. In the chapters 3 and 4, more explanation is given to show how these software help to reach the goal of this thesis.

2.3.1 Modelsim

ModelSim [7] is a multi-language HDL simulation environment by Mentor Graphics, for the simulation of the hardware description languages such as VHDL, Verilog and SystemC [24], and includes a built-in C debugger. ModelSim can be used independently, or in conjunction with the Xilinx ISE [25].

There are multiple editions of ModelSim, such as ModelSim PE, ModelSim SE, and ModelSim XE.

To compare ModelSim SE and PE, ModelSim SE offers high-performance and advanced debugging capabilities, while ModelSim PE is the entry-level simulator for hobbyists and students. ModelSim SE is used in large multi-million gate designs, and it is supported on the Microsoft Windows and Linux, in 32-bit and 64-bit architectures. In this thesis, ModelSim SE is used for the simulation and Verification of the hardware design.

2.3.2 Vivado Design suite

Xilinx [25] provides a software suite called Vivado Design Suite [6] which is for synthesising and analysing of HDL designs. In addition, it is used for superseding Xilinx ISE [25] with additional features for system on a chip [26] development and high-level synthesis.

Vivado give the possibility to developers to synthesize (compile) their designs and perform the timing analysis and do examine RTL diagrams. Vivado simulates a design's reaction to different stimuli, and configure the target device with the programmer. This is a design environment for FPGA [27] products from Xilinx and cannot be used with FPGA products from other vendors.

Xilinx is leading provider of Electronic System Level Design tools for all programmable solutions. The Vivado Design Suite System Edition provides Vivado High-Level Synthesis (HLS) [28] for the C, C++ and SystemC and automatically converts it into the Verilog or the VHDL for further synthesis. In addition, HLS has some interface and optimization settings, which can be configured by TCL scripts. These feature give the ability to high-level IP specifications to be directly synthesized into the VHDL and the Verilog accelerating IP verification. The C function is synthesized into a IP clock with the Xilinx Vivado HLS which is integrated into a hardware system.

2.3.3 Xilinx Software Development Kit (SDK)

The Xilinx Software Development Kit (XSDK)[29] is the Integrated Design Environment (IDE) for creating embedded applications on any of Xilinx microprocessors such as the Zynq-7000 All Programmable SoCs, and the industry-leading MicroBlaze™. The SDK is the first application IDE to deliver true homogeneous and heterogeneous multiprocessor design and debug. In addition, SDK is based on the Eclipse open source platform which is used as a compilation environment and C code editor. This software has been managed as a application build configuration, automatic Makefile generation and error navigation.

In addition to the above native Eclipse provided features, the SDK also provides the following tools for use in the Xilinx embedded software development. It provides Xilinx Microprocessor Debugger (XMD) [30] as a debugging agent to communicate with the Xilinx embedded processors. In addition, it is used to program the Xilinx FPGA with the bitstream.

The XMD facilitates debugging programs and provides a Tool Command Language (TCL) interface. This interface can be used for running complex verification test scripts

to test a complete system as well as command line control and debugging of the target. The XMD console is used as a standard TCL console, where any available TCL commands could be run. Additionally, the XMD console provides command editing convenience, such as a file and a command name auto-fill and a command history.

2.4 Zynq-7000 Development board

2.4.1 Zynq SoC

The Zynq-7000 [5] family is based on the Xilinx All Programmable SoC architecture. It provides a very fast interaction during accelerating programs. Every Zynq SoC included a processing system on PS and a programmable logic on PL. There are feature-rich dual-core or single-core ARM Cortex-A9 on PS and 28 nm Xilinx programmable logic on PL in a single device [31]. There are ARM Cortex-A9 CPUs in the heart of the PS also on-chip memory is included, in addition there are external memory interfaces, and a rich set of peripheral connectivity interfaces.

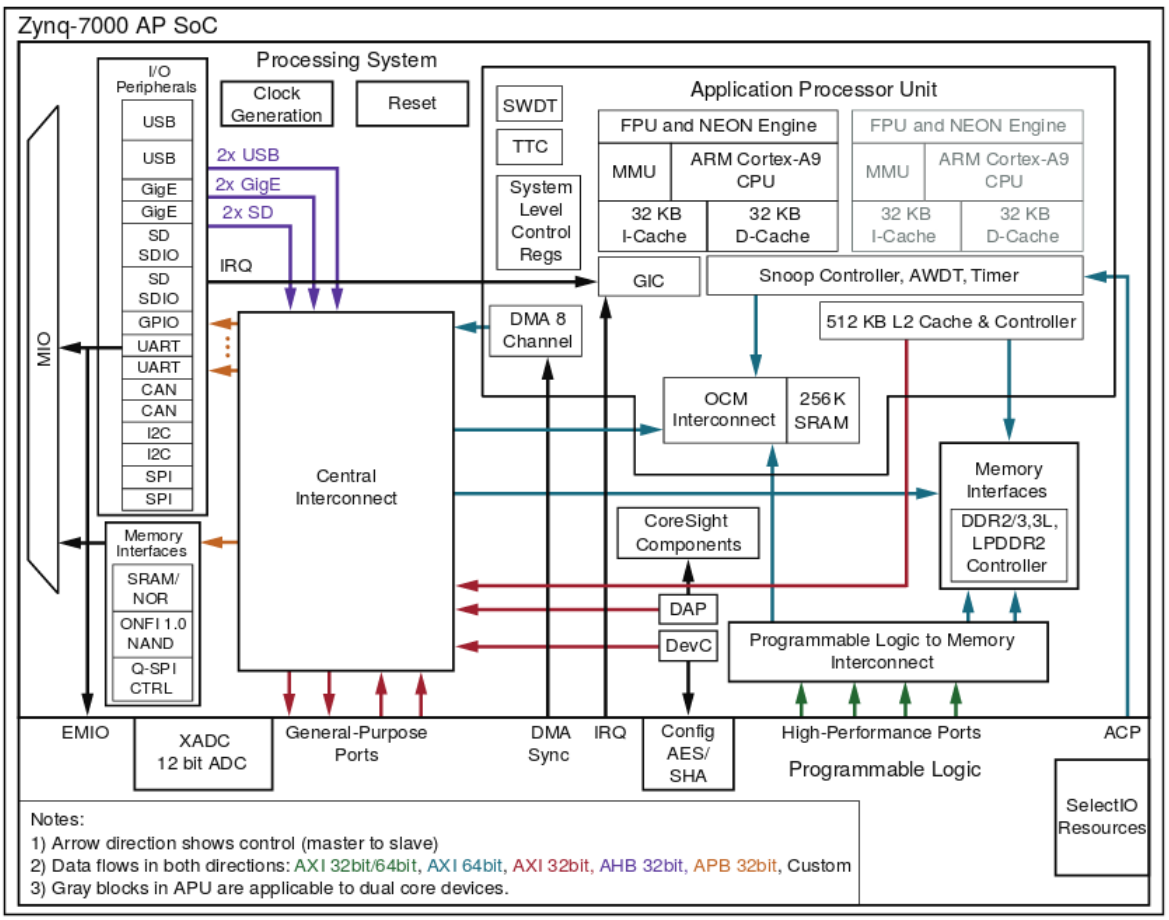

Figure 2.3: Zynq-7000 AP SoC Overview.

The Xilinx Zynq-7000 family offers the flexibility and scalability of an FPGA and

it provides performance, power, and ease of use for the designers. Designers have the choice to use the different range of the Zynq 7000 families which contain the same PS, the PL and I/O resources and on the other hand the Xilinx provides a large number of soft IP for the Zynq 7000 family.

The main functional blocks of the Zynq 7000 Soc are illustrated in the Figure 2.3. As it is mentioned before, the PL and the PS are the main blocks. The PL Consist of an Application Processor unit (APU), Memory interfaces, I/O peripherals (IOP) and Interconnect.

In the section 3.3, the IP integration with the combination of the PS and the PL blocks is described.

2.4.2 Zybo Board

In this study, the Zybo Board is chosen as the development board for our implementation of our accelerator prototype (Figure 2.4). It contains a Xilinx Zynq-7000 All Programmable System on Chip (c7z010-1clg400c), which consists of a dual-core ARM Cortex-A9 MPCore based Processing System (PS) and an Artix-7 FPGA as programmable logic (PL) [5]. The PS includes on-chip memory, external memory interfaces, some of I/O peripherals. The system offers the flexibility and scalability of an FPGA, also will provide performance, power, and ease of use typically associated with ASIC and application specific standard product (ASSP) [31]. The PL makes use of the second version of the Advanced Extensible Interface (AXI4) bus protocol, which is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) [2], [32].

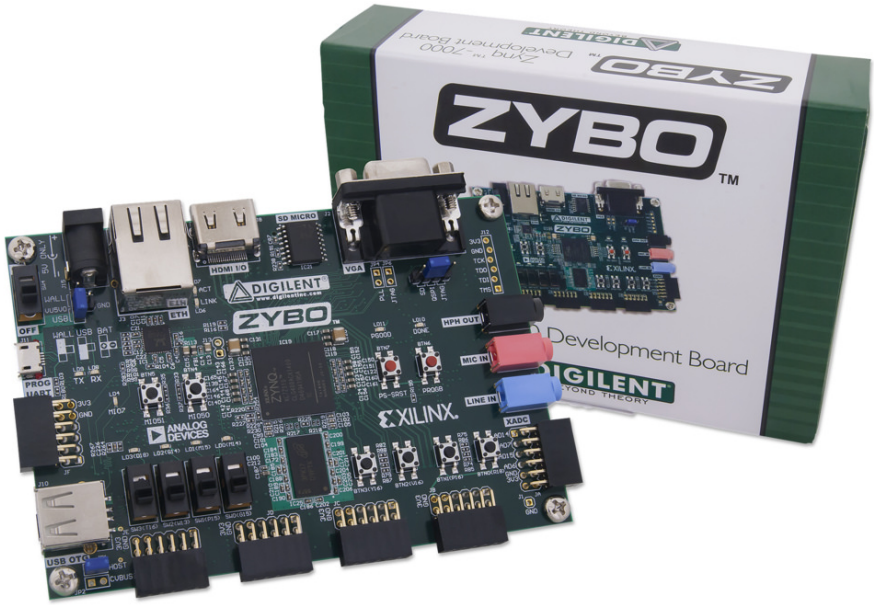

Figure 2.4: The ZYBO Zynq-7000 development board

The Zynq PS and the PL are interconnected via the different interfaces. There are two 32-bit Master AXI ports (PS master) and two 32-bit Slave AXI ports (PL Master),

plus four 32/64-bit Slave High-Performance ports (PL Master). From the PS to the Pl, four clocks are provided, and there are different interrupts between the PS to the PL and vice versa.

The Cortex-A9 processor uses 32-bit addressing and all the PS, and the Pl peripherals are memory mapped to the processor core. In this regards, all the slave PL peripherals are located between 40000000 and 7FFFFFFF (if they are connected to GP0) and 80000000 and BFFFFFFF (if they are connected to GP1).

2.5 Introduction of an Axi-Bus for the Xilinx System Development

The feature that makes the Zybo Board especially well-fitted for hardware acceleration applications is the tight coupling between the PS and the PL. The ARM processors can be connected directly to any component in the PL area through the set of general purpose AXI ports and an Extended Multiplexed I/O (EMIO) port (see Figure 2.3). Besides, there are four high performance (HP) AXI4 ports that PL components can use to access external memory directly. At max capacity, the HP AXI4 ports have a bandwidth of 1200 MB/s.

The second version of the Advanced eXtensible Interface(AXI4) bus protocol is used to make the PL, which is part of the ARM Advanced Microcontroller Bus Architecture (AMBA)[2]. The AXI4 is available in the Xilinx ISE Design Suite which can be applied in the projects. It is suitable for low latency and high bandwidth designs. Differently, the AMBA provides high-frequency operation without the use of bridges. It fits the interface requirements of wide range of components and is suitable for memory controllers.

There are three kinds of AXI4 interfaces:

- AXI4-Full: For high-performance memory-mapped requirements.
- AXI4-Lite: For simple, low-throughput memory mapped communication.
- AXI4-Stream: For high-speed streaming data.

The AXI specification describes an interface between an AXI Master and an AXI Slave. They are connected and using a structure called Interconnect block (Figure 2.5), and in the case of the AXI4-Full and the AXI4-Lite, it is called an AXI Interconnect. It is used for the memory mapped interfaces only while the AXI4-stream interconnect can be utilized for the AXI-4 stream bus implementation.

Any of those interfaces are implementable in the PL. Therefore they can be connected directly to the PS through a set of AXI4 bus ports. In the rest of this section, the AXI specification and the protocols explained in the details.

2.5.1 AXI4-full and AXI4-lite Protocols

The AXI-Full specification proposes a different range of important features such as variable data and address bus widths with high bandwidth burst operations. Also, it

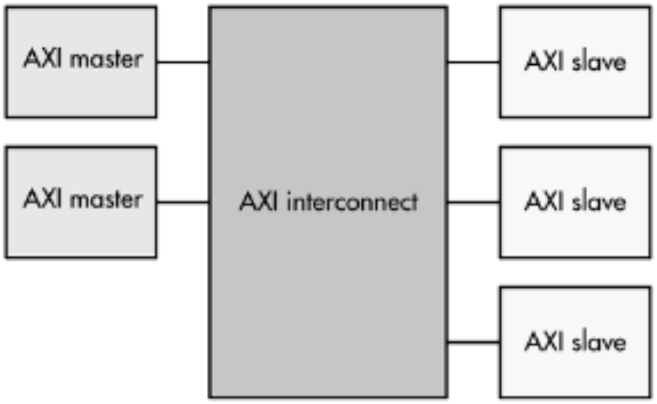

Figure 2.5: AXI-Bus Interconnect [2]

offers advanced caching support and several transaction assurances and access permissions. While these features offer the user flexibility and control, it is often useful to be provided with a much simpler peripheral which consists of only a subset of these functions. For that reason, a reduced feature variant of the AXI4-Full specification exists in the form of the "AXI4-lite".

The AXI4-Lite interconnect provides only necessary interconnect transactions which are required, and high-level capabilities of the interconnect such as burst support, cache support, and variable bit widths for the address and data buses has been removed. The AX4-lite interconnect is suitable for applications where simple control and status monitoring capabilities are required for a custom built IP block.

Both the AXI-Full and the AXI-Lite have five different channels between the Master and the Slave (See Figure 2.6 and 2.7).

Data between the master and the slave can move in both directions simultaneously, and data transfer sizes can be different. The AXI4-Full consists of single address with multiple data with a burst transaction up to 256 data beats, but AXI4-Lite provides only 1 data transfer per transaction with 32 bits data width.

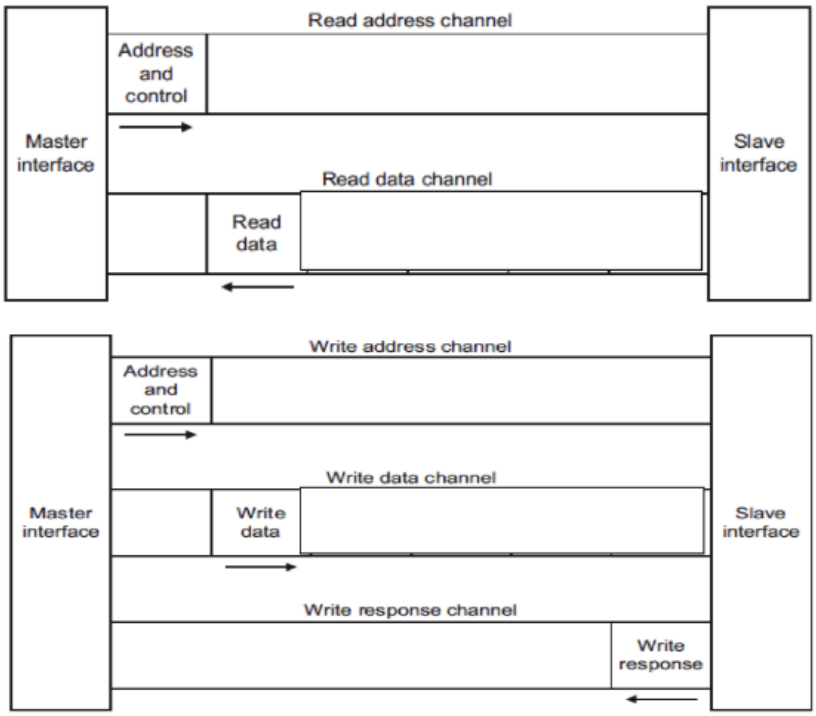

Figure 2.6: AXI-Lite Interface [2]

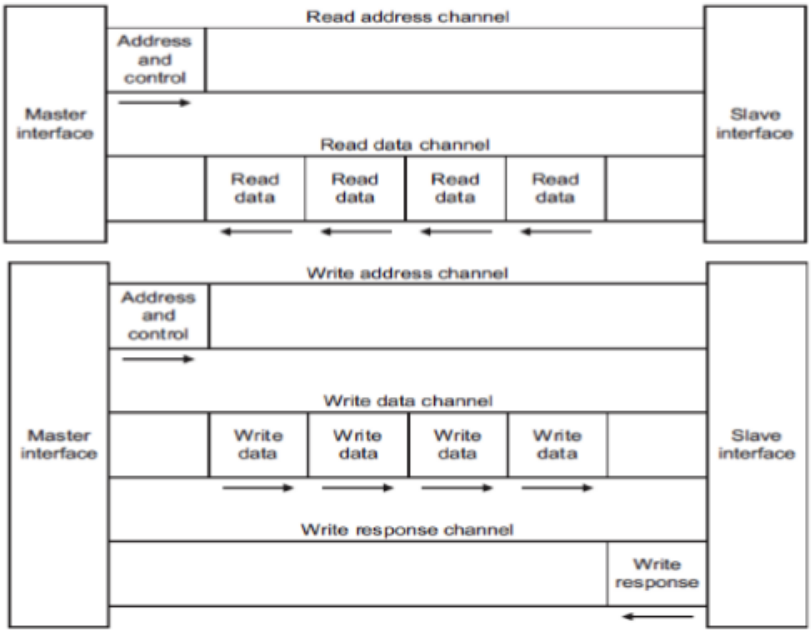

Figure 2.7: AXI-Full Interface [2]

3 System Design

This chapter describes an application developed on the Zynq-7000 All Programmable SoC devices. The capability of the ARM-based processor in the software programming and the FPGA in the hardware programming is considered and integrated into a single device.

The Zynq-7000 All Programmable SoCs (System On Chip) introduce a customizable intelligence into today's embedded systems to suit our application requirements. This family of the FPGA is suitable for high-end application because it has large resources on the single chip. In this regard, it allows the designers to create their custom hardware IP. Therefore, this ability was a driving force of this thesis to create a customized IP based on the previous work [3]. The Zybo board (xc7z010clg4) [33] is chosen as the development board for the implementation of our accelerator prototype which contains a Xilinx Zynq-7000 All Programmable System on Chip (SoC).

The main goal of the current work is to create a Custom Slave IP which can be utilized to integrate the NNA with an AXI-Bus interface architectures, and it is called a Neural Network IP-core (NNIP). Later this NNIP will be verified and evaluated in simulation tool trough a testbench and a writing software.

The integrating and communication of the NNA to the AXI Bus interface is explained in this section as follow. First, the overall view of the design is illustrated to give a wider perspective about this thesis. Second, how structurally the NNA is integrated into a new custom IP design, and finally, the combination of the PS and the PL is covered.

3.1 System Design Overview

In the Figure 3.1, the Zybo board (xc7z010) [33] is shown by dotted lines which consist of many interfaces but it is highlighted only those which is used in this design. Seven main blocks in the figure are shown, and they are covered in this part. There is a Zynq IC (xc7z010-1clg400c) which is mainly divided into the two sections, a Processing System (PS) and a Programmable Logic (PL) [5] which is an FPGA. These two components are connected based on the AXI Bus interconnect. The PS consist of the processor unit, on-chip memory, external memory interfaces, and peripheral connectivity interfaces. The ARM Cortex-A9 and the AXI-Bus interconnect are located in the PS. The AXI specification gives a structure that describes protocols for transferring data between the IPs with using a defined signaling standard. This standard secures that IP can exchange data with each other and that data can be moved across a system. The AXI protocol defines how data is exchanged, transferred, and transformed. It also ensures an efficient, flexible, and predictable means for transferring data.

The PL includes the programmable logic, the configuration logic, and associated

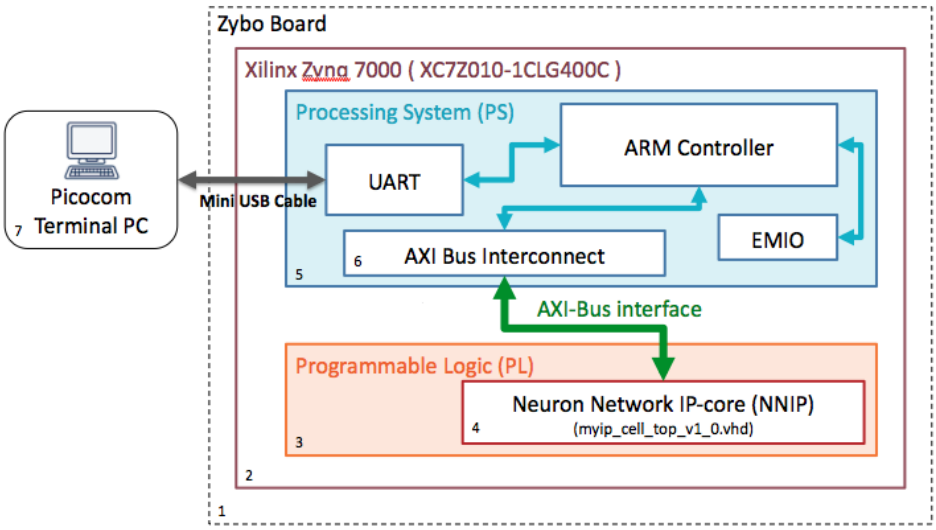

Figure 3.1: System design overview

embedded devices. The main goal of this thesis is integrating the NNA into the custom slave IP in the PL which is called the Neuron Network IP-core (NNIP). The architecture of the NNIP is described in detail in the next subsection. Another component is a UART [33]. When we are using a Xilinx Development Board with a USB-UART port, it uses a mini-B USB cable to connect the USB-UART port on the board to a PC which is equipped by a Picocom terminal emulators. There are various kinds of terminal emulators [34], but the Picocom terminal is used, due to availability on the PC in the lab.

3.2 PL architecture

In order to design and implement the interface for the NNA to communicate with the ARM processor via AXI-bus. The NNIP (see Figure 3.2) is designed and implemented. The NNIP consists of an AXI-Lite slave and an AXI-Full slave which are interfaced by the AXI-Bus interconnect. In below, the characterization of both AXI-Bus interface is discussed.

- AXI-Full salve

  An AXI-Full slave (see Figure 3.2) consists of different parts as follow, a Neuron Network Application (NNA) called a Cell-top.vhd, a Clk Generator and a Memory block (BRAM).

  The NNA has been made based on the different inputs and outputs which are sketched in the figure 3.2. The specification about the inputs and the outputs will be discussed in chapter 4. All the input signals except *S_start*, *Clk* and *Reset* are connected to the the related and allocated registers in the AXI-Lite. The *Clk* and the *Reset* are connected with a system Clk and a system reset. The last

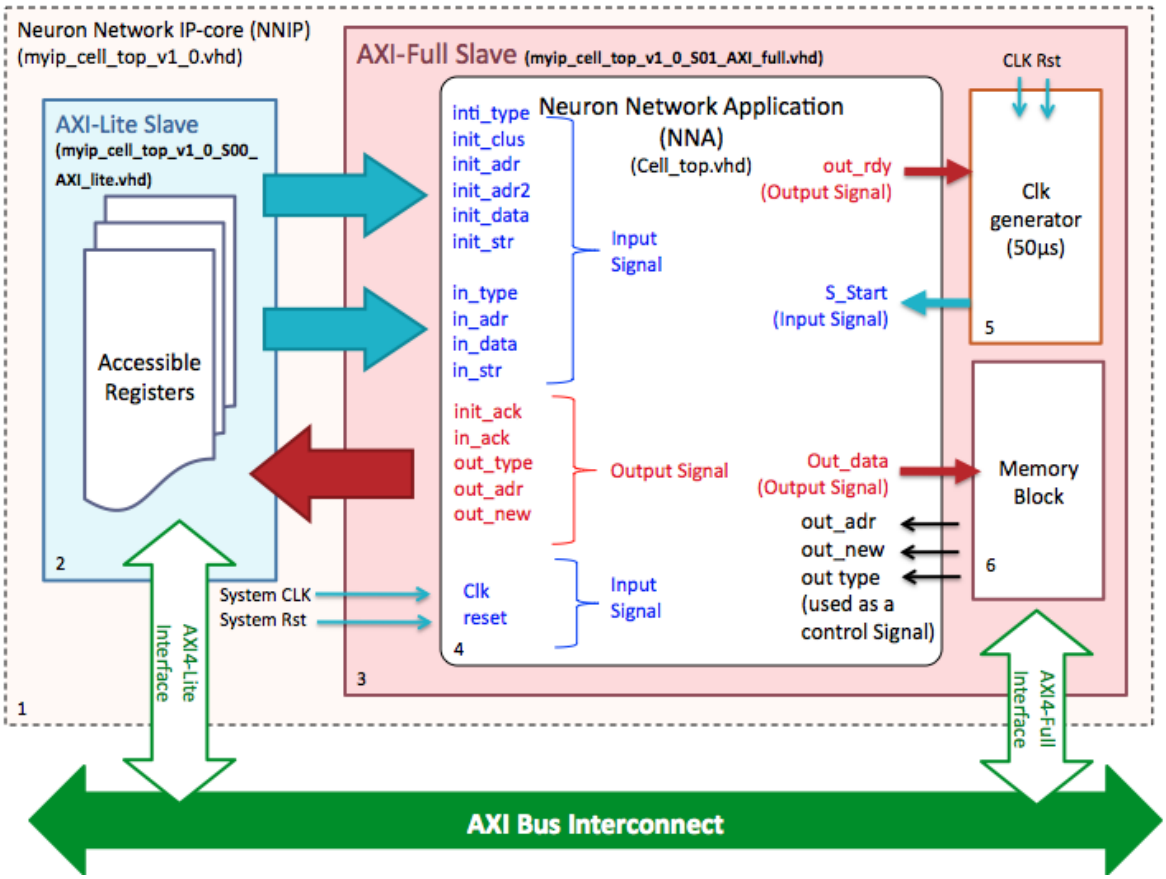

Figure 3.2: PL architecture overview

important input signal is *S_start* which is connected to the output of the Clk generator block. The Clk generator block is responsible for delivering 50µs Clk period to the output based on the signal *Out_rdy*.

All the outputs of the NNA except the *Out_data* vector are stored in allocated AXI-Lite registers. In this regard, all the output signals are accessible through the AXI-Bus interface and the designer can monitor them easier with defined address of the registers.

The memory block with a size of 64k byte (see Figure 3.3) is implemented to be used for storing the *out_data* vector data. This alignment of the data in the memory is based on the *out_adr* signal and control signals such as *out_new* and *out_type*. This memory block is accessible by allocated and configured addresses in the range of 0#7AA00000 to 0#7AA0FFFF through the AXI-Bus interconnect with the 32 bits data word.

- AXI-Lite slave

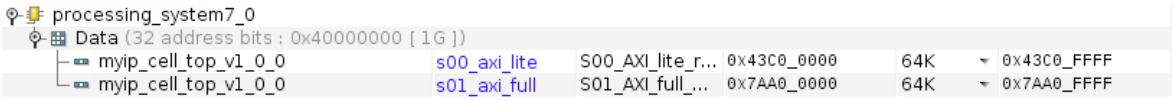

Figure 3.3: Registers addressing

The AXI-Lite slave block (see the Figure 3.2) consists of local registers with the length of 32 bits. Those local registers are defined and used for storing the input data which are required for the AXI-Full inputs, and storing some outputs from the AXI-Full. Those local registers are accessible by allocated and configured addresses in the range of 0#43C00000 to 0#43C0FFFF through the AXI-Bus interconnect (see Figure 3.3). Through the accessing to the registers, it is possible to monitor the inputs signal and some important control outputs signals which are used to verify and proof the correctness of the NNA outputs.

The hardware implementation of AXI-Lite slave and the AXI-Full slave are discussed in a details in chapter 4.

3.3 PL/PS architecture

The Zynq PS and the PL are interconnected via the different interfaces. There are a 32-bit Master AXI port and a 32-bit Slave AXI port, and also some clocks and reset signal from the PS to the Pl are provided.

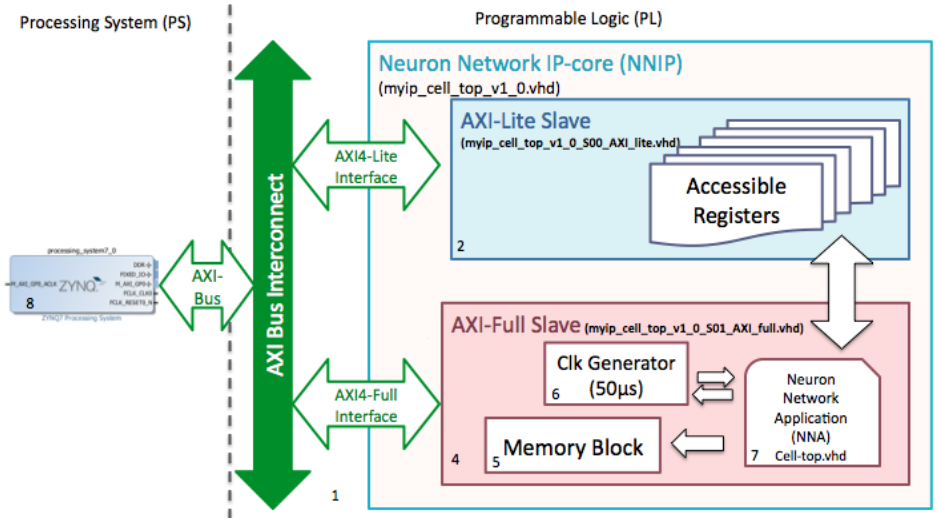

Figure 3.4: PS/PL architecture overview

Figure 3.4 shows how the Neuron Network IP-core (NNIP) is designed to be connected to an embedded processor in Processing System (PS) with an AXI-Lite and an AXI-Full interface. The processor acts as a Master, and the Custom Slave IP acts as a Slave. By accessing the allocated registers via the AXI-Lite interface and the Memory

block via the AXI-Full interface, the processor can control, and read and write the data from and to the NNIP.

For example, to access to the data ports of the NNA, the PS should read from the memory block in the AXI-Full and write to the associated local registers in the AXI-Lite.

The ARM Cortex-A9 processor uses 32-bit addressing, and all the PS and the Pl peripherals are memory mapped to the processor core. In this regards, all slave PL peripherals are located between 0#40000000 and 0#7FFFFFFF (In this study, it is connected to the GP0) and 0#80000000 and 0#BFFFFFFF (if it was connected to the GP1).

4 System Implementation

In this chapter, an integration of the Neuron Network Application (NNA) within a Custom Slave IP (NNIP) and its communication through the AXI-Bus interface are introduced. This chapter mainly is divided into five main sections. Firstly, in order to start with the NNA as an initial application, its SystemC testbench implementation and specification is explained. Secondly, converting the NNA (the SystemC and C implementation) to the VHDL code is shown. Thirdly, simulation of the integrated NNA within the Custom Slave IP (NNIP) block and the AXI-Bus interface (the PL implementation) is illustrated. Fourthly, the synthesizing, optimization and bitstream generation of the block deign (the PS and the PL integrated block) are described and finally, the software implementation is explained.

4.1 Neuron Network Application (NNA)

The aim of this section is to give an overview about the NNA and its testbench.

4.1.1 NNA Overview

Before the testbench explanation, it is necessary to have an overview about the Neuron Network Application (NNA) which has been written in the file called *Cell_ top.vhd*. This application model has been tuned based on 4 parameters (1x1x25x1) such as number of *Physical Cell Clusters* (PCCs), the amount of shared *Exponent Coprocessor* (ExpC) within each PCC, the number of *Physical Cell* (PhyC) in each PCC, and the *Time Sharing Factor*(TSF) for each PhyC. Those parameters and definitions are described in [3], and the reader is strongly advised to understand them before starting this chapter.

To understand better the I/O signals of the NNA, Appendix A and Figure 4.1 are illustrated to show the linked PCCs by Routers and I/O ports of the FPGA. When a PhyC has calculated a new axon voltage or neuron response for a given *Simulated Cell* (SimC), it is sent through the tree network. The axon voltages are only directed to the I/O of the FPGA, while the responses are sent to all of the other PCC's. Finally, if a response needs to be overwritten or a SimC needs to be released from/receive an impulse, a signal can be injected into the tree network. When all responses have been streamed to the I/O of the FPGA and received by the clusters, respectively, the model is ready to start calculating the new SimC states [3]. It is important to mention the timing of computational cycles within a PCC and implementable hardware design timings, which are 50µs for a single PCC.

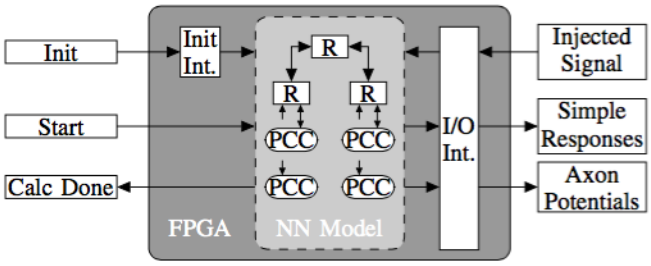

Figure 4.1: General overview of the NNA implementation on an FPGA [3].

4.1.2 Testbench description of the NNA

A SystemC testbench has been done in previous result [3]. In that result, running the hardware simulation model and initialization is briefly described. There is a *Cluster_rdy* signal in each PCC, which has been controlled by 3 input signals such as the *Clock*, the *Reset* and the *Start*.

In the Figure 4.2, there are two operating modes such as an initialization and a normal operation. In the NNA, operationally, the SimCs within the design needs to calculate and communicate simulated ION responses to their neighbors and the axon. To simulate the real-time behavior of the complete Neuron Network (NN), all implemented SimCs must have been carried out these two operations within 50µs [3].

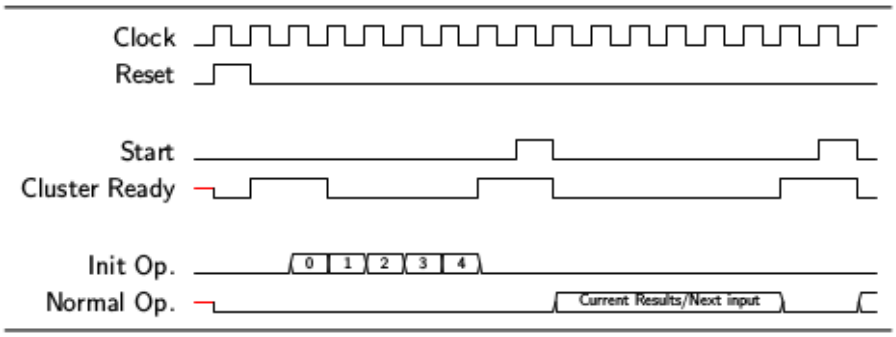

Figure 4.2: Neuron Network Application (NNA) System Control [3]

Every PCC sends one output and receives two inputs data vector besides the status signal and control which are connected to FIFO's. One input vector has been connected to the initialization port and the other input and output vector have been connected to the main communication port. Every port vector has been made with multiple smaller bit vectors which are dependent on the accuracy of the simulation model either 32 bit or 64 bit model, the maximum number of connected SimCs a SimC can have, and the configuration that used to build the system.

Two bridges have been placed to make handshaking between the data ports of the PCC, and the input and the output pins of the FPGA. One bridge has been connected to the main communication port and other to the initialization port [3].

4.1.2.1 Initialise PCCs

Each SimC needs to be initialised, before a simulation is started. When a reset is given (Figure 4.2), the PCC will pull up the *Cluster_ rdy* signal to show that the PCC is ready to receive new initialization data (Init Op). Now it is possible to sent the initialization data.

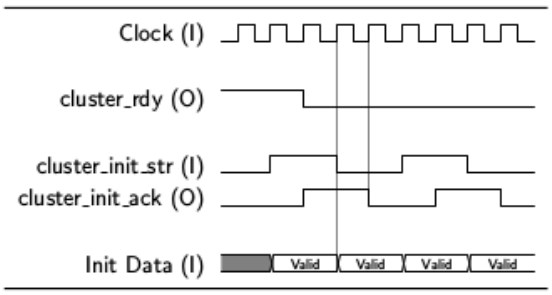

Figure 4.3: Init_data Handshaking [3]

The simple bridge has been designed to promote handshaking between the initialization-port-connected FIFO's and the I/O of the FPGA to initialise the system. In Figure 4.3 the handshaking protocol of the init-data is illustrated. The handshaking has been defined in two steps, Firstly, valid data is located on the Init data input pins, and the strobe signal is inverted. Secondly, when the data is received, it acknowledges this by making the acknowledgment equal to the strobe.

In this thesis, the signals for the handshaking between the FIFOs and the I/O of the FPGA are replaced by the AXI-Bus interface signals. In the next sections, the AXI-Bus interface is explained in detail.

Five different types have been defined for the initialization data. For an accurate system initialization, each type has to be sent. Table 4.1 and Figure 4.4 show the different init vectors and it is not important in which sequence type (0 through 3) is sent, but as soon as a PCC gets a type 4 init data vector, the initialization mode for that PCC will be locked [3].

Table 4.1: Init type table [3]

Number	name
0	Init cluster number
1	Init cell dendrite voltage
2	Init cell parameters
3	Init connectivity matrix
4	Init done
5-7	Not used

A brief description about 5 initialization steps (Table 4.1) are listed as below [3]:

Init cluster number (*init_ type = 0*): Every PCC is initialized with a specific index number (PCC nr) by sending a type 0 init data vector. Based on this number, the PCC calculates the global address (SimC adr) of each SimC.

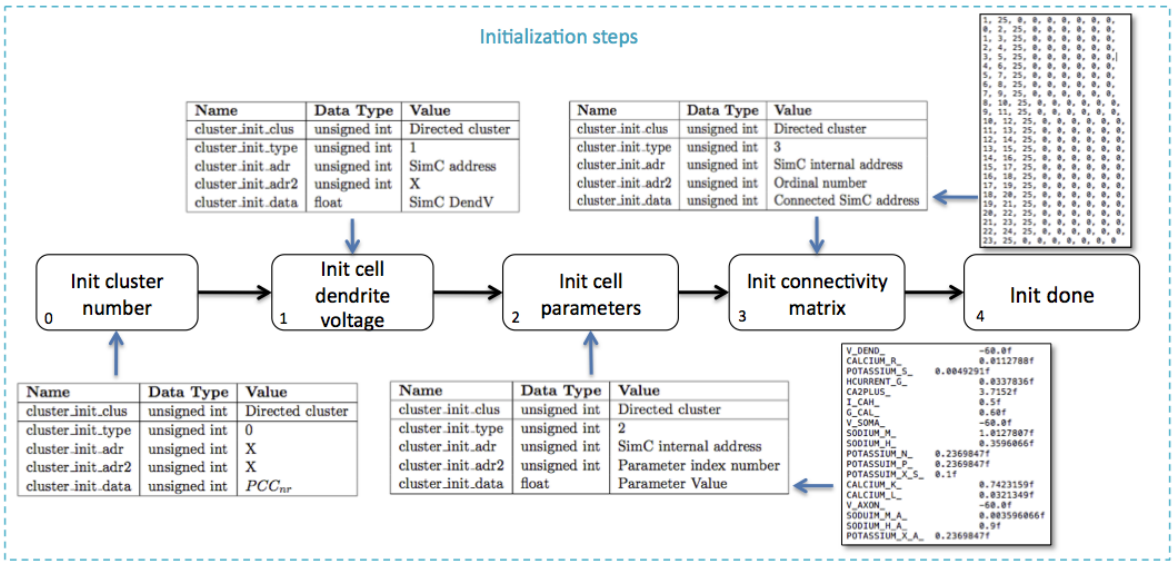

Figure 4.4: Initialization steps for the NNA

Init cell dendrite voltage (*init_type = 1*): A local copy of all the dendrite voltages is stored through every PCC. To initialize the dendrite voltages stored within each PCC controller, each PCC needs to get every dendrite voltage for all SimCs in the design. A single dendrite voltage within a single PCC is initialized after sending a type 1 init data vector to the design. Looping this data vector for each SimC adr on each cluster will setup all dendrite voltages within the design.

Init cell parameters (*init_type = 2*): Every SimC has its individual set of locally stored initial parameters. These are 16 cell properties and three initial cell states (Vsoma and Vaxon, Vdend). A single initial parameter for a single SimC is set with giving a type 2 init data vector to the design. Looping this data vector within every parameter on a SimC with int-adr within a PCC will setup the given SimC. With doing this for all SimCs, each SimC in the design will be setup.

Init connectivity matrix (*init_type = 3*): Every SimC gets many responses from other SimCs about where it is placed in the neuron network. It is initialized with giving a type 3 init data vector to the PCC.

Init done (*init_type = 4*): After a type 4 init data vector is given, the PCC is init locked.

Figure 4.5 shows a waveform of five mentioned initialization steps in the SystemC testbench and, all the inputs and setup parameters are illustrated.

4.1.2.2 Start Simulation

After locking the PCC (see Figure 4.6), the signal *cluster_rdy* is asserted and a simulation can be started. *s_start* is a single control signal to control the simulation. It is the timing of computational cycles within a PCC and implementable hardware

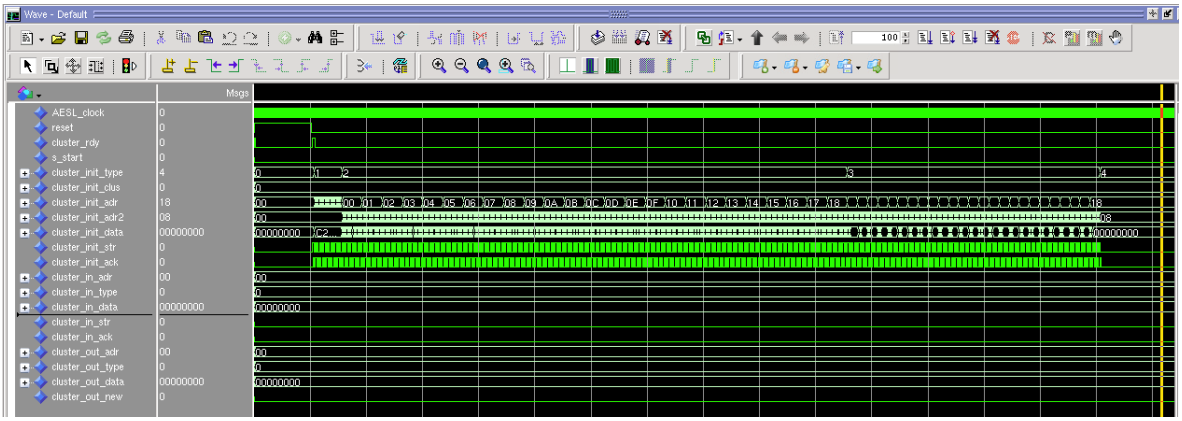

Figure 4.5: Waveform of initialization steps in SystemC testbench

design schedules, which are 50µs for a single PCC. The PCCs will calculate the output dendrite and axon responses during the simulation [3].

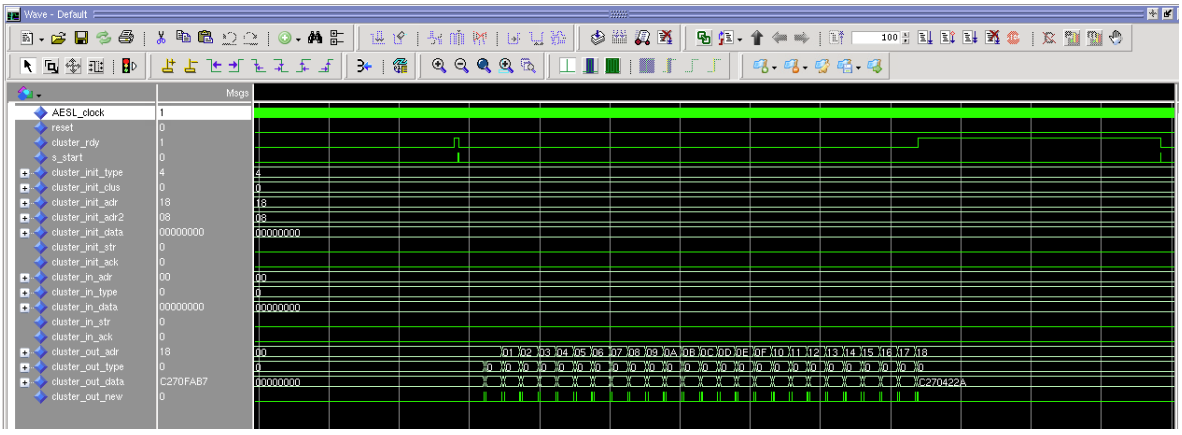

Figure 4.6: Waveform of Start simulation in the SystemC testbench

4.1.2.3 Receive Result

The axon and dendrite voltage responses made by the system will automatically get out of the main communication output port and streamed to the output pins of the FPGA with the bridge. In order to understand the new valid data is presented the bridge will assert the *cluster_out_new* signal. If the new value (*cluster_out_data*) is of type 1, it means axon voltage, it is stored in an array at address *cluster_out_adr* for later comparison with the reference model [3].

The next step is using mentioned information about the NNA and its SystemC testbench, in order to implement the PL with integrating the NNA with the AXI-Bus interface, and later, simulating the PL and comparing the VHDL testbench with the SystemC testbench.

4.2 PL Implementation and Simulation

4.2.1 Simulation and implementing overview

In this Section the PL implementation and its simulation are described as below. Firstly, the Vivado_HLS tool has been used to generate the VHDL code and related TCL files from the NNA implementation (see Figure 4.8). Secondly, the Vivado Suite is used to integrate the generated VHDL code from the NNA within the Neuron Network IP-core (NNIP) and the AXI-Bus interface which is implemented in this study. Thirdly, it is exported as a Package Ip-core and finally, the Modelsim internally in the Vivado is used to run a simulation for the NNIP. The PL Implementation will be verified by a VHDL testbench. The steps are shown in Figure 4.7 by order and it is discussed in detail in this section.

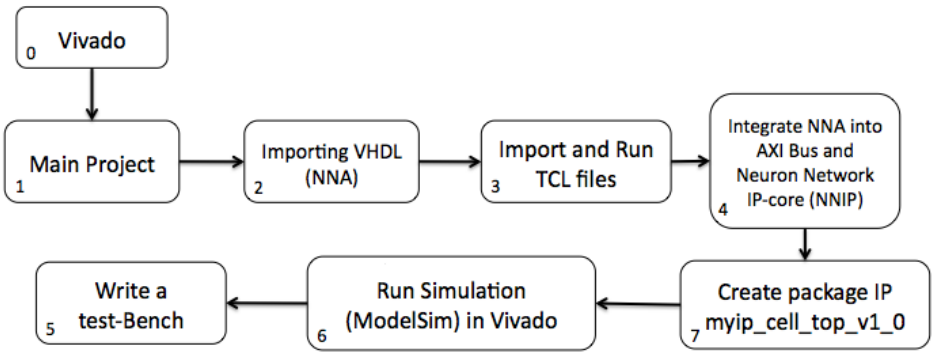

Figure 4.7: PL implementation and simulation overview

4.2.2 Generate VHDL from NNA

The NNA implementation in the previous work [3] models an Inferior Olivary Nucleus (ION) network on an FPGA devices designed with the SystemC and called as a *cell-top.c*. As it is explained in section 2.3.2 about functionality of Vivado HLS, this tool has been used to convert NNA implementation automatically into VHDL for further synthesis (see Figure 4.8).

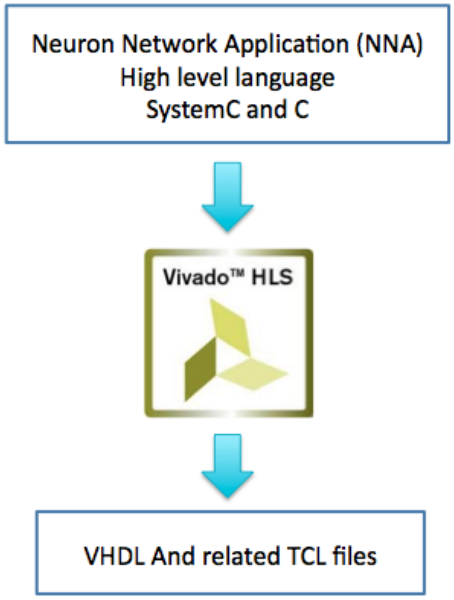

Figure 4.8: Generate VHDL from C and SystemC files

4.2.3 PL Implementation (NNIP Implementation)

The Pl architecture is explained in Chapter 3 and shown in Figure 3.2. In this section, integrating the Neuron Network Application (NNA) into the Neuron Network IP-core (NNIP) with an AXI-Bus interface is explained in detail.

In the section 3.2 is mentioned that there are two AXI-Bus interfaces in the design of NNIP such as the AXI-Lite salve and the AXI-Full slave interface. Their signal names are completely flexible from the point of view of the VHDL design. During the creation of a Xilinx IP block, the Vivado tools are used to map each AXI signal onto the signal name that it is used when creating the IP. However, in order to make the life of the designer much easier, the signal names used here are recommended when designing a custom AXI slave in the VHDL. Using these signal names have allowed the Vivado design tools to automatically detect the signal names during the "create and package IP" step (described later on).

The AXI-Lite salve and the AXi-Full slave channels are described in detail in section 2.5.1. The next important aspect of those AXIs is the handshaking signals which is really important, and it is done in this study. These signals are consistent amongst the five channels and give a powerful way to control all read and write transactions.

Slave AXI-Lite implementation

To implement read and write transaction in the AXI-Lite slave, The signals are defined (signals are listed in the Appendix B) based on a simple *"Ready"* and *"Valid"*

principle. Signal *"Ready"* is used by the recipient to indicate that it is ready to accept a transfer of a data or address value, and *"Valid"* is used to clarify that the data (or address) provided on that channel by the sender is valid so that the recipient can then sample it.

In AXI-Lite Slave, the memory mapped register selection and write logic generation are implemented. The write data is accepted and written to memory mapped registers when *slv_reg_wren* is asserted. It happens when *axi_awready*, *S_AXI_WVALID*, *axi_wready* and *S_AXI_WVALID* are asserted.

Write strobes are used to select byte enables of the slave registers while writing. These registers are cleared when reset (active low) is applied. In Slave register write enable is asserted when the valid address and data are available and the slave is ready to accept the write address and write data.

There is simplified implemented code in Appendix C that shows how the write transaction to a local register is implemented in the AXI-Lite slave.

The reading data is accepted and reading from the memory mapped registers happens when the *axi_aready*, the *S_AXI_ARVALID* and the *axi_rvalid* are asserted. It means, the slave register read enable is asserted when the valid address is available and the slave is ready to accept the read address. An implemented code in Appendix D shows how the write transaction to a local register is implemented in the AXI-Lite slave.

Salve AXI-Full Implementation

In this thesis, a handshaking signals for the AXI-Full slave are implemented in the five channels. They give a powerful way to control all read and write transactions from the memory block in the AXI-Full slave (see figure 3.2). As it is mentioned before the goal of defining this Memory block is storing the output of the NNA (the Axon and the Dendrite voltage) after one iteration based on another NNA output signals such as the *Cluster_out_adr*, the *Cluster_out_Type* and the *Cluster_out_new*.

The example implemented code to access defined logic memory region is shown in Appendix E. The NNA produces the data output (the Axon and the Dendrite Voltage) via the *Cluster_out_data* signal and for every output data, the *Clutser_out_new* will be asserted. The writing data is accepted and writing to the memory mapped registers happens when the *Clutser_out_new* is asserted. In order to organize the Axon and Dendrite voltage data in the the memory block, the *Clutser_out_type* and the *Clutser_out_adr* have the important role. The address location of the output voltage in the memory is given by the *Clutser_out_adr* and the type of the output voltage data is distinguished based on the *Clutser_out_type* (if it is "00", it means the Dendrite Voltage and if it is "01" means the Axon Voltage). It is mentioned in the beginning of this chapter that the NNA has been tuned based on one PCC and 25 PhyCs in a PCC (configured setup 1x1x25x1). With this configuration, 25 Axons voltage and 25 Dendrites voltage as the 32bit output data are expected to be written in the memory with the size of 1.6 KB in every iteration.

A reading data from the Memory block will happened when the *mem_rden* signal is asserted. It means that the *axi_aready* , the *S_AXI_ARVALID* and the *axi_rvalid*

should be asserted. The handshaking implementation code to access and read from the memory region in AXI-Full slave is provided in Appendix E.

4.2.4 Packaging Neuron Network IP-core (NNIP)

The next step of the design process is to package the IP to put it into a format that can be understood by the Xilinx Vivado block diagram GUI. The Vivado Tool is used to create an IP based on the imported NNIP source code. Then "Create and Package IP" menu item is chosen from the "Tools" menu at Vivado. Through this menu option, a tool is started within the Vivado suite called the "IP Packager", which took all of the design sources within that project, and started a design wizard which is provide access to all of the configuration settings needed for the IP to be created. To create an IP, existing source files is used, and all the source files are added in the design sources in the vivado project (see Figure 4.9). There are different configuration wizards that should be followed before create the package IP such as the IP naming and family in the library, the IP customization parameter, the IP ports, the IP interfaces and the IP addressing and memory.

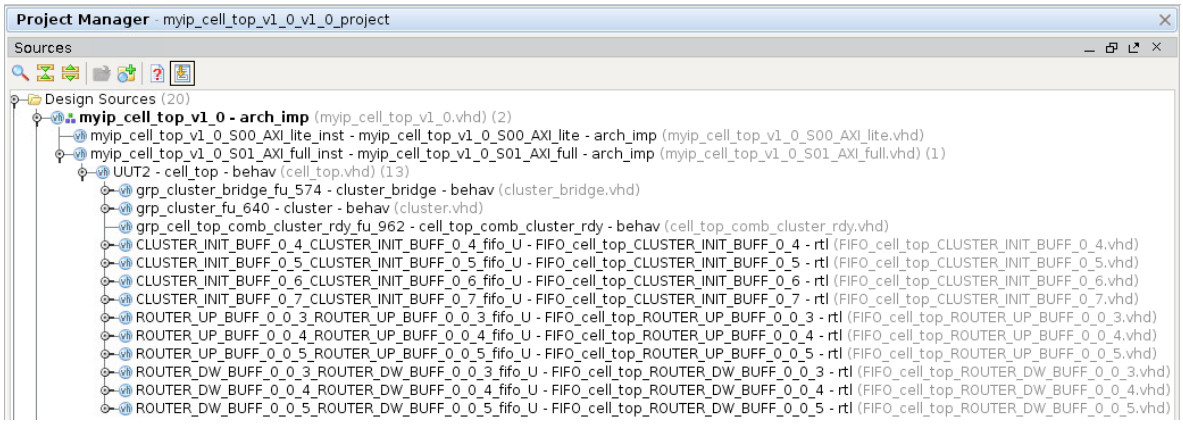

Figure 4.9: Design source overview of Package IP (NNIP)

After setting up all the parameters and creating the package IP, the Neuron Network IP-core (NNIP) is accessible via the Vivado IP catalog with the name of *myip_cell_top_v1_0_1*. It is ready to be used for creating block design (see Figure 4.10), but before using the NNIP to create a hardware block design, it should be verified by a simulation (e.g. Modelsim) by writing a VHDL testbench.

4.2.5 PL Simulation (NNIP simulation)

The next step is to verify the correctness of the packaged IP (NNIP) by writing a VHDL testbench. The ModelSim is well suited to simulate digital components based on the hardware description language VHDL. Generally, a VHDL model consists of several blocks, each defined by an interface (entity) and dedicated architectures. Entities are described by unidirectional and bidirectional ports, architectures can be implemented by a behavioral or a structural description. The behavioral description on an

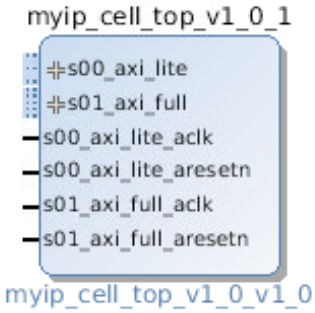

Figure 4.10: Neuron Network IP-core (myip_cell_top_v1_0_0)

abstract VHDL modeling level allows using data types and programming techniques known from other programming languages. On a structural level, a VHDL model mostly consists of basic logic blocks. Both description types can be merged. The NNIP (*myip_cell_top_v1_0*) is simulated by the VHDL testbench which behaves as a Processing System (PS) to sent and received the data through the AXI-Bus interconnect. The simulation overview and its source files overview are shown in Figures 4.11 and 4.12 respectively.

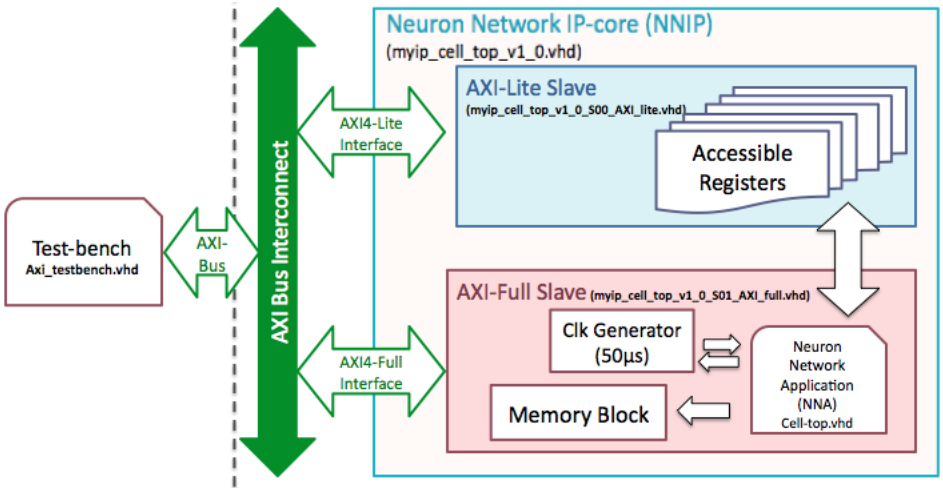

Figure 4.11: Simulation diagram overview of NNIP

Figure 4.12: Simulation Source Files overview of NNIP

4.2.5.1 Initialise PCCs

Before writing the VHDL testbench, it is essential to understand the SystemC testbench which is explained above in section 4.1.2. All the initialization steps before starting the simulation are illustrated in Figure 4.13. As it is mentioned, the initialization data has five different types, and for the correct system initialization, each type has to be sent (these five steps are shown by a signal waveform in Figure 4.14). As soon as a PCC receives a type 4 init data vector, the initialization mode for that PCC will be locked, and the simulation will be started.

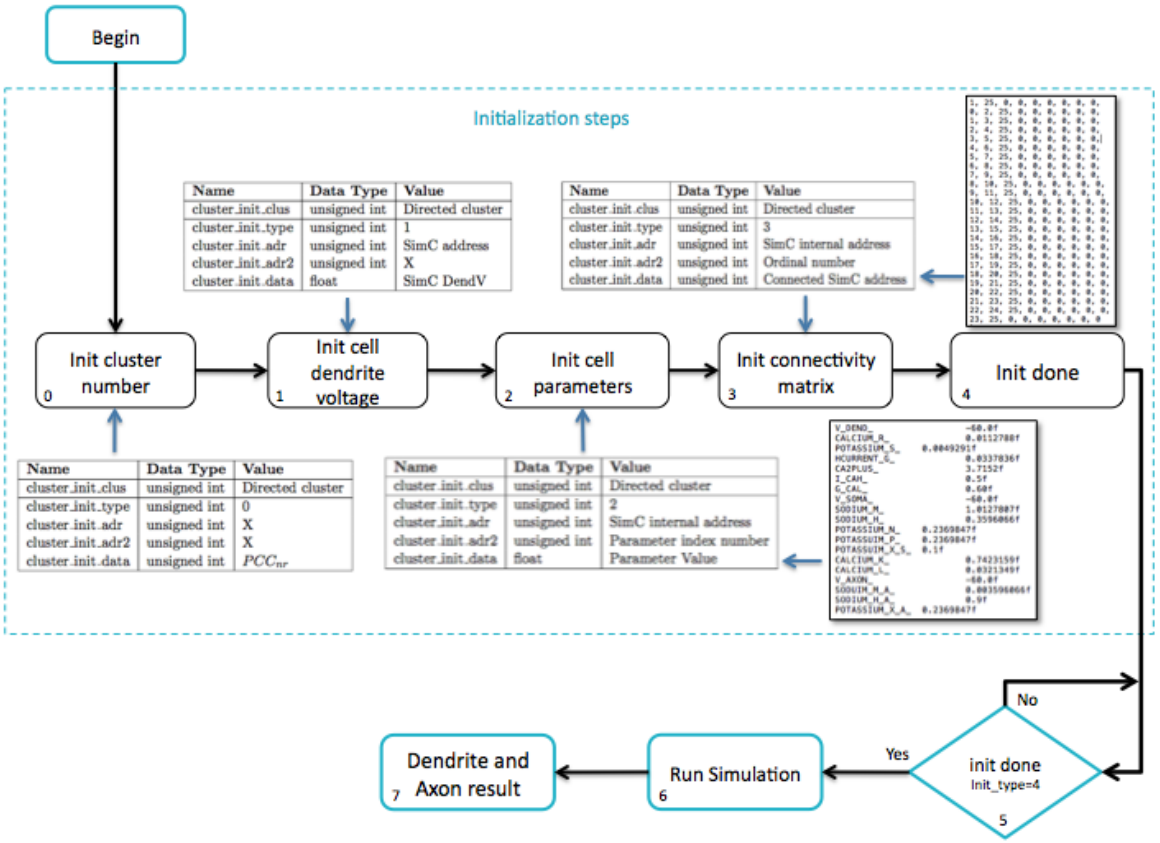

Figure 4.13: Test-Bench overview

4.2.5.2 Start Simulation

Figure 4.15 shows the waveform simulation after the PCCs receive type 4 init data vector and the status signal *cluster_rdy* is pulled up and a simulation can be run. The simulation is controlled by single control signals *s_start* which is the timing of the computational cycles within a PCC and implementable hardware design timings, which are 50µs for a single PCC. During the simulation, the system will output dendrite and the axon responses calculated by the PCCs.

Responses generated by the NNA will automatically be taken out of the main communication output port, and streamed to the output pins of the FPGA by the bridge,

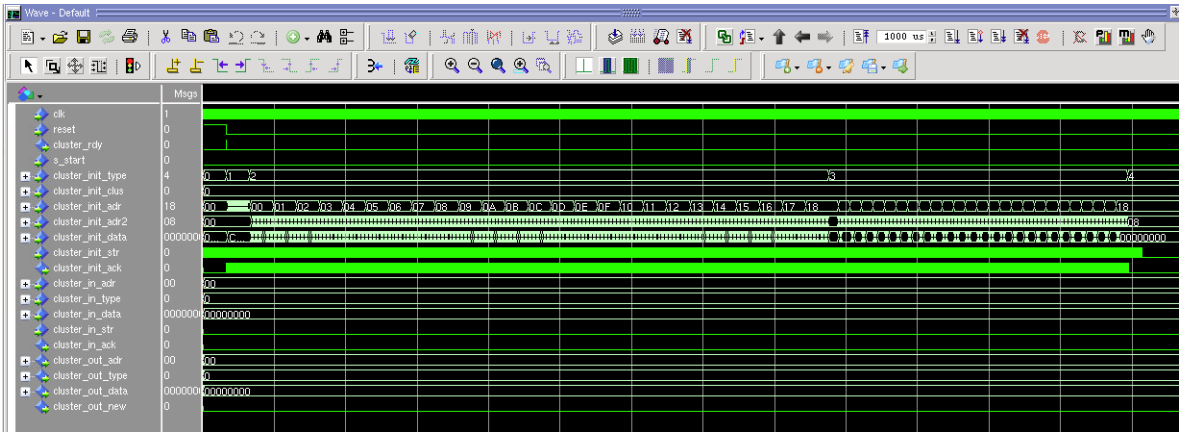

Figure 4.14: Waveform of initialization steps in the VHDL testbench

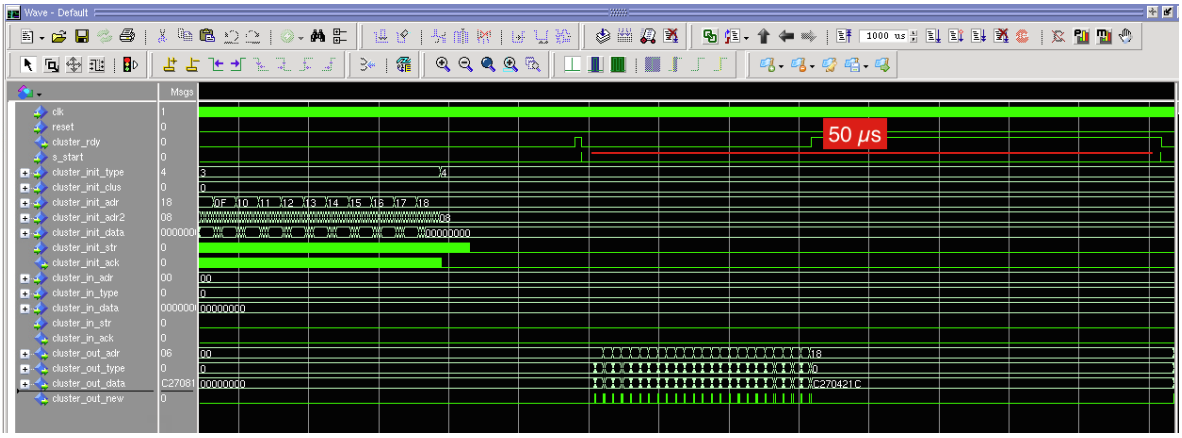

Figure 4.15: Waveform of Start simulation in VHDL testbench

which is internally is connected to the defined memory block and registers. To signify that a new valid data is provided, the bridge will pull up the *cluster_out_new* signal. The system will check if the desired response type is provided. If the new value (*cluster_out_data*) is type 1 (axon) or type 0 (dendrite), they are stored in an memory at a address of *cluster_out_adr* for the later comparison with the reference SystemC testbench model.

4.2.5.3 Simulation Evaluation

In order to validate the correctness of the simulation, the waveform of the VHDL simulation of the NNIP is compared with the reference waveform of the SystemC simulation done by [3] (they are shown above (4.1.2.2) in Figure 4.5 and 4.6). The goal of this simulation is to verify five main capabilities of the NNIP (*myip_cell_top_v1_0*) implementation.

Table 4.2: *Cluster_ out_ data* output of the SYSTEMC simulation vs. the VHDL simulation

SystemC testbench					VHDL testbench				
	Axon Voltage		Dendrite Voltage			Axon Voltage		Dendrite Voltage	
out_adr	Hex	Float	Hex	Float	out_adr	Hex	Float	Hex	Float
0	0xC26FE764	-59.976	0xC270422A	-60.0646	0	0xC270EB32	-60.2297	0xC270421C	-60.0646
1	0xC26FE764	-59.976	0xC270422A	-60.0646	1	0xC27B96FA	-62.8974	0xC270421C	-60.0646
2	0xC26FE764	-59.976	0xC270422A	-60.0646	2	0xC27B96FA	-62.8974	0xC270421C	-60.0646
3	0xC26FE764	-59.976	0xC270422A	-60.0646	3	0xC27B96FA	-62.8974	0xC270421C	-60.0646
4	0xC26FE764	-59.976	0xC270422A	-60.0646	4	0xC27B96FA	-62.8974	0xC270421C	-60.0646
5	0xC26FE764	-59.976	0xC270422A	-60.0646	5	0xC27B96FA	-62.8974	0xC270421C	-60.0646
6	0xC26FE764	-59.976	0xC270422A	-60.0646	6	0xC27B96FA	-62.8974	0xC270421C	-60.0646
7	0xC26FE764	-59.976	0xC270422A	-60.0646	7	0xC27B96FA	-62.8974	0xC270421C	-60.0646
8	0xC26FE764	-59.976	0xC270422A	-60.0646	8	0xC27B96FA	-62.8974	0xC270421C	-60.0646
9	0xC26FE764	-59.976	0xC270422A	-60.0646	9	0xC27B96FA	-62.8974	0xC270421C	-60.0646
10	0xC26FE764	-59.976	0xC270422A	-60.0646	10	0xC27B96FA	-62.8974	0xC270421C	-60.0646
11	0xC26FE764	-59.976	0xC270422A	-60.0646	11	0xC27B96FA	-62.8974	0xC270421C	-60.0646
12	0xC26FE764	-59.976	0xC270422A	-60.0646	12	0xC27B96FA	-62.8974	0xC270421C	-60.0646
13	0xC26FE764	-59.976	0xC270422A	-60.0646	13	0xC27B96FA	-62.8974	0xC270421C	-60.0646
14	0xC26FE764	-59.976	0xC270422A	-60.0646	14	0xC27B96FA	-62.8974	0xC270421C	-60.0646
15	0xC26FE764	-59.976	0xC270422A	-60.0646	15	0xC27B96FA	-62.8974	0xC270421C	-60.0646
16	0xC26FE764	-59.976	0xC270422A	-60.0646	16	0xC27B96FA	-62.8974	0xC270421C	-60.0646
17	0xC26FE764	-59.976	0xC270422A	-60.0646	17	0xC27B96FA	-62.8974	0xC270421C	-60.0646
18	0xC26FE764	-59.976	0xC270422A	-60.0646	18	0xC27B96FA	-62.8974	0xC270421C	-60.0646
19	0xC26FE764	-59.976	0xC270422A	-60.0646	19	0xC27B96FA	-62.8974	0xC270421C	-60.0646
20	0xC26FE764	-59.976	0xC270422A	-60.0646	20	0xC27B96FA	-62.8974	0xC270421C	-60.0646
21	0xC26FE764	-59.976	0xC270422A	-60.0646	21	0xC27B96FA	-62.8974	0xC270421C	-60.0646
22	0xC26FE764	-59.976	0xC270422A	-60.0646	22	0xC27B96FA	-62.8974	0xC270421C	-60.0646
23	0xC26FE764	-59.976	0xC270422A	-60.0646	23	0xC27B96FA	-62.8974	0xC270421C	-60.0646
24	0xC26FE764	-59.976	0xC270422A	-60.0646	24	0xC27B96FA	-62.8974	0xC270421C	-60.0646

1. Verification of the AXI-Bus interface implementation.
2. Verification of reading and writing from/to registers in the AXI-Lite slave (*myip_ cell_ top_ v1_ 0_ S00_ AXI_ lite*).
3. Verification of the NNA integration with the AXI-Full slave (*myip_ cell_ top_ v1_ 0_ S01_ AXI_ full*) and the AXI-Bus interface.
4. Testing the memory bock implementation in the AXI-Full slave (*myip_ cell_ top_ v1_ 0_ S01_ AXI_ full*).
5. Comparing the *Clutser_ out_ data* (the NNA outputs data) with the reference model (the SystemC NNA and simulation). In Table 4.2, the *Clutser_ out_ data* in integrated NNA in the Neuron Network IP-core (NNIP) and the reference model SystemC NNA simulation are compared. The table shows, the axon voltages in the both simulation (the VHDL and the SystemC) are nearly similar and the dendrite voltage is exactly the same.

4.3 PL and PS Implementation on Hardware

Once the VHDL testbench is passed, the packaged IP is completed, and it is accessible in the IP catalog in the Vivado. Figure 4.17 shows the hardware implementation flow as follow. A new Vivado project for target device or a Xilinx target board (a device architecture that was included in the list of supported architectures in IP which is the Zybo Zynq Evaluation board in this thesis) is created, and the new IP core (*myip_cell_top_v1_0*) is added to the IP Catalog. The next step is creating a block design by adding the Zynq-7000 processor as a master and connect it with the *myip_cell_top_v1_0*. Another IP including Processor System Reset and the AXI Interconnect is added to the system design by connection automation via the Vivado. In order to PS accesses to the PL local registers and the allocated memory block, the base and offset addresses of the AXI-Full Slave and the AXI-Lite slave are well defined in the block design (see Figure 4.18).

To be continued the steps in figure 4.17, the block diagram is ready, and the next step in the Vivado is to create an HDL wrapper, and then synthesize, implement, and create a bitstream for the design. Then the hardware specification and implementation is exported to the Software Development Kit (SDK) for preparation to start writing software for the hardware design.

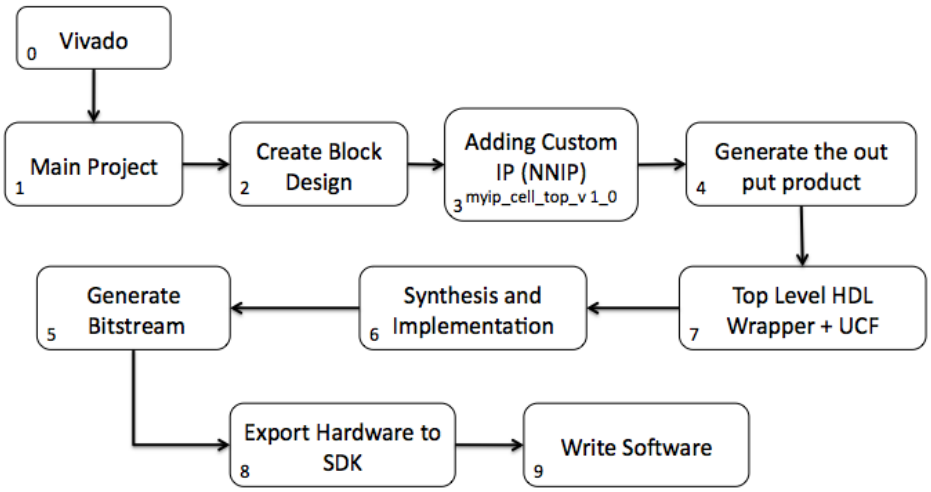

Figure 4.16: Hardware implementation flow

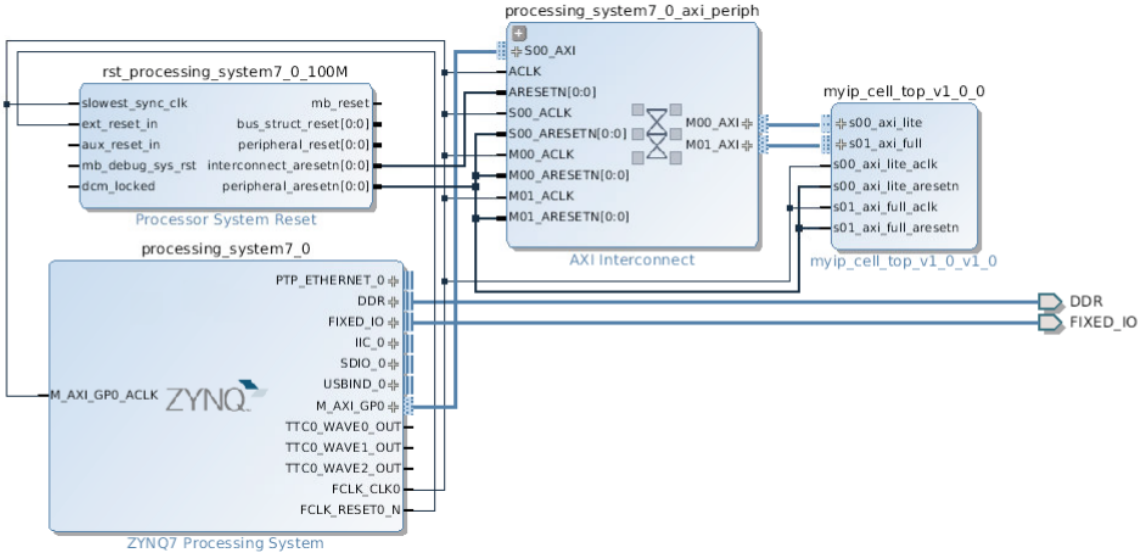

Figure 4.17: Block diagram including *myip_cell_top_v1_0* (PS and PL)

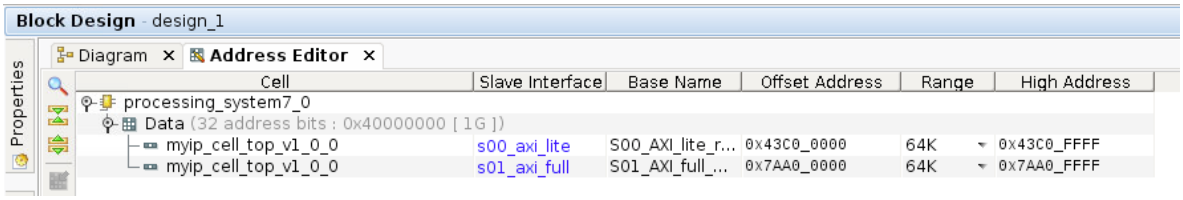

Figure 4.18: Base address and range of the Custom Slave IP Address

4.4 Software

The Xilinx Software Development Kit (XSDK) [29] is the Integrated Design Environment (IDE) for creating embedded applications on any of Xilinx microprocessors such as the Zynq-7000 All Programmable SoCs. The SDK is an Eclipse-based software design environment which enables the integration of hardware and software components through the link inside the Vivado.

Figure 4.19 shows the summary for the proposed, implemented and tested projects in this thesis. In order to write, debug, and deploy software applications for the hardware, the Hardware Platform Specification files are required which are all the information and files from a hardware design. After exporting the hardware and launching the SDK, Vivado creates the Hardware Platform Specification in a directory which is used in the SDK. The main components of this specification are a hardware description, an FPGA Bitstream corresponding to the hardware description, a Block RAM Memory Map (BMM) file corresponding to the Bitstream and the Zynq-7000 APSoC Initialization Files.

The Zynq-7000 AP SoC Initialization Files consist of a *ps7_init.tcl* file, which is the pre-initialization file that is to be sourced and executed before any application is

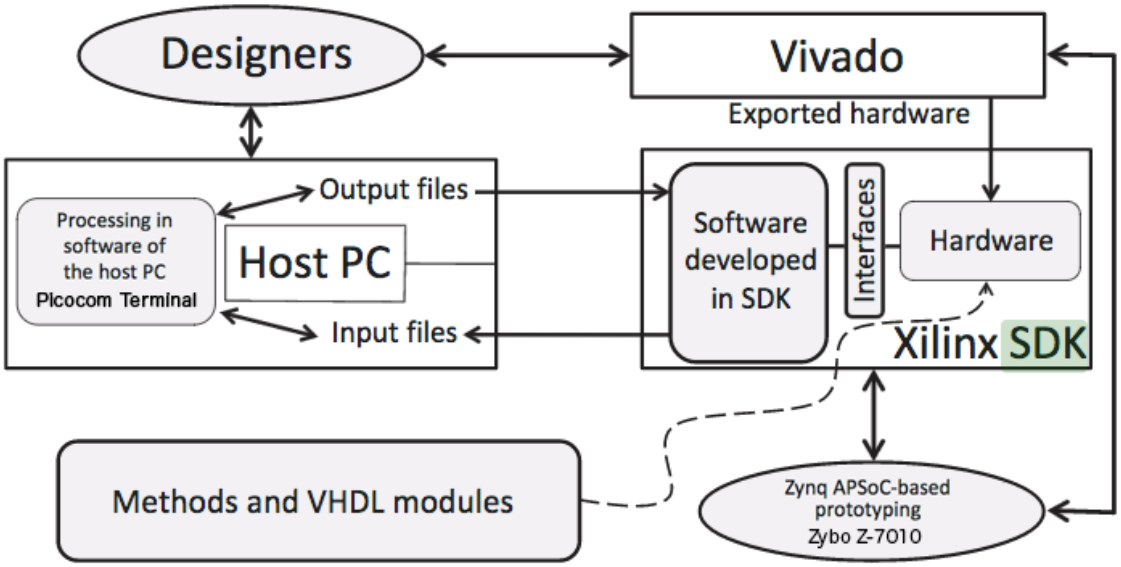

Figure 4.19: Summary for the proposed, implemented and tested projects[4]

downloaded onto the target. This TCL file contains the clock, pll and ddr initialization code. The other required files are a *ps7_init.c* and *ps7_init.h*. These files are the C-equivalent files for the *ps7_init.tcl* file. The SDK is used the Hardware Description file to detect the processor(s) and memory-mapped peripherals present in the hardware [35].

Another Required step before writing the software is creating a Board Support Package (BSP). The SDK creates the BSP when a new software project is created. The BSP is the collective term referring to all of the software components required to match a given operating system (and its environment) to a given hardware design (board). The BSPs commonly contain a low-level operating system and device driver code that is dependent on the hardware and upon which the rest of the operating system layers. In addition, the BSP might also contain other files containing directives, compilation parameters, and hardware parameters that are used to configure the operating system. Since a given Xilinx hardware platform is configurable, the fixed board support packages are not possible. Therefore, the custom board support package must be generated for each hardware design that is created [35].

Figure 4.20 gives an overview about the software implementation files and its structure. In order to understand the software architecture, it is essential to briefly explain related .c and .h files in this Figure.

When the BSP is created, some compilation and the hardware parameters that are used to configure the operating system will be generated. The main important one are listed below:

- *xparameters.h*: This file contains the address definitions for the peripherals attached to the ARM Cortex A9 core.

- *xil_io.h*: This file contains the interface for the general IO component, which

encapsulates the Input/Output functions for processors that do not require any special I/O handling. The most commonly used functions calls are *Xil_Out32()* and *Xil_In32()*, but similar functions exist for 16 bit and 8 bit data transfers.

- *xbasic_types.h*: This file contains the standard type definition of integer and float numbers with signed and unsigned 8,16 and 32 bit.

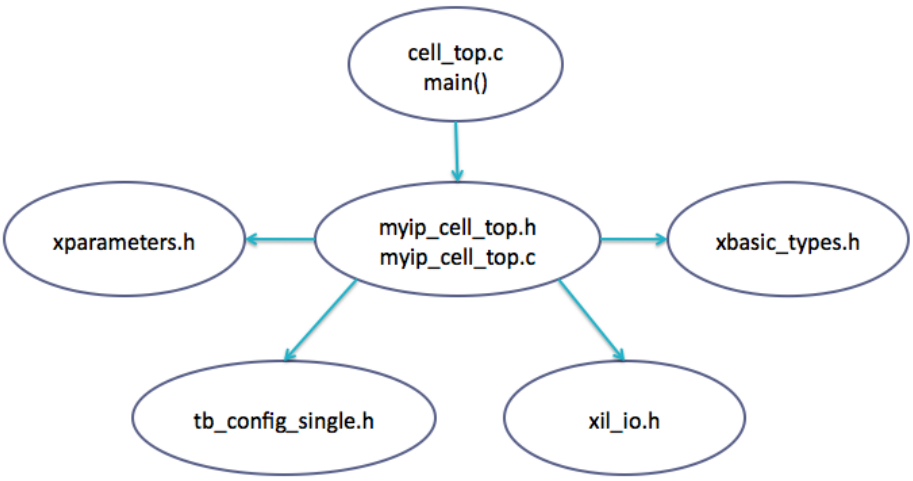

Figure 4.20: Software implementation overview

In the beginning of this chapter (section 4.2.5, Figure 4.13) the initialization steps for the testbench was illustrated. A software is implemented with the same path in the main() file called a *cell_top.c*, and all the needed functions are made in the file called *myip_cell_top.c* and they will be defined in file called *myip_cell_top.h* . Two functions are defined for reading and writing from/to the AXI-Lite slave local registers based on the defended register address and offset. In order to access to the memory block inside the AXI-Full slave, a function is implemented for reading from the memory block based on the address and offset.

And finally all the functions for five steps initialization are implemented to be used in the main() function (*cell_top.c*) for initializing the NNA hardware implementation in the Zynq-7000 APSoC family (ZyBo-xc7z010clg400) with the embedded dual-core ARM-Cortex-A9.

In the main() (*cell_top.c*) function, all the initialization will be started and some control signals are used to check whether the initialization is done and also the simulation is completed. Then all the generated axon and dendrite output data will be read from the memory block and they will be shown through the Picocom emulator terminal on the screen.

4.5 Evaluation the Results

The Hardware Platform Specification file (the Bitstream and the other required files *ps7_init.c/.h*) are ready and the software is compiled, and a .elf file is generated within

the SDK. In order to verify the software and the hardware, the Xilinx Development Board (In this thesis, the ZyBo-xc7z010clg40) is connected with a PC through a USB UART port (with a mini-B USB cable). The board is programmed with the Bitstream file and initialized with other required files (*ps7_init.c/.h*). Then the software can be run in the SDK. The PC is equipped by the Picocom terminal emulators [36], in order to communicate with the hardware and print the software result in terminal. Figure 4.21 shows the different processes in the software from the initialization to end of the simulation and finally, print the axon and dendrite voltage for 25 PCCs from the memory storage.

```
Start of testing My Ip
======= Init 0 done =====
======= Init 1 done =====
======= Init 2 done =====
======= Init 3 done =====
======= Init 4 done =====
-------- initialization DONE ----
number_cluster_rdy of slave lite : FFFFFFF2
number_s_start of slave lite : FFFFFFF2
number_out_new of slave lite : 32
-------- Simulation DONE ----
Reading Axon and Dendrite voltage from memory
memory Axon voltage data for 0 is  : C270EB32
memory Axon voltage data for 4 is  : C27B96FA
memory Axon voltage data for 8 is  : C27B96FA
memory Axon voltage data for 12 is  : C27B96FA
memory Axon voltage data for 16 is  : C27B96FA
memory Axon voltage data for 20 is  : C27B96FA
memory Axon voltage data for 24 is  : C27B96FA
memory Axon voltage data for 28 is  : C27B96FA
memory Axon voltage data for 32 is  : C27B96FA
memory Axon voltage data for 36 is  : C27B96FA
memory Axon voltage data for 40 is  : C27B96FA
memory Axon voltage data for 44 is  : C27B96FA
memory Axon voltage data for 48 is  : C27B96FA
memory Axon voltage data for 52 is  : C27B96FA
memory Axon voltage data for 56 is  : C27B96FA
memory Axon voltage data for 60 is  : C27B96FA
memory Axon voltage data for 64 is  : C27B96FA
memory Axon voltage data for 68 is  : C27B96FA
memory Axon voltage data for 72 is  : C27B96FA
memory Axon voltage data for 76 is  : C27B96FA
memory Axon voltage data for 80 is  : C27B96FA
memory Axon voltage data for 84 is  : C27B96FA
memory Axon voltage data for 88 is  : C27B96FA
memory Axon voltage data for 92 is  : C27B96FA
memory Axon voltage data for 96 is  : C27B96FA
memory Dendrite voltage data for 404 is  : C2704181
memory Dendrite voltage data for 408 is  : C270421C
memory Dendrite voltage data for 412 is  : C270421C
memory Dendrite voltage data for 416 is  : C270421C
memory Dendrite voltage data for 420 is  : C270421C
memory Dendrite voltage data for 424 is  : C270421C
memory Dendrite voltage data for 428 is  : C270421C
memory Dendrite voltage data for 432 is  : C270421C
memory Dendrite voltage data for 436 is  : C270421C
memory Dendrite voltage data for 440 is  : C270421C
memory Dendrite voltage data for 444 is  : C270421C
memory Dendrite voltage data for 448 is  : C270421C
memory Dendrite voltage data for 452 is  : C270421C
memory Dendrite voltage data for 456 is  : C270421C
memory Dendrite voltage data for 460 is  : C270421C
memory Dendrite voltage data for 464 is  : C270421C
memory Dendrite voltage data for 468 is  : C270421C
memory Dendrite voltage data for 472 is  : C270421C
memory Dendrite voltage data for 476 is  : C270421C
memory Dendrite voltage data for 480 is  : C270421C
memory Dendrite voltage data for 484 is  : C270421C
memory Dendrite voltage data for 488 is  : C270421C
memory Dendrite voltage data for 492 is  : C270421C
```

Figure 4.21: Software result

In this Figure, five initialization steps are specified. In addition, to verify and check the simulation, three counters are defined to record and track the process in the hardware.

- number_clutser_rdy

  Counting the rising age of the *cluster_ rdy*

- number_s_start

  Counting the rising age of the *s_ start*

- number_out_new

  Counting the rising age of the *number_ out_ new*

As it is mentioned before about the relation between the *Cluster_ rdy* and the *S_ start* control signals (section4.2.5.2), it is expected to see value 2 as a counter value for both signals in one iteration. To signify that new valid data is provided by the *cluster_ out_ data* vector signal, the *number_ out_ new* is used to count the pulling up of the *cluster_ out_ new* signal. After running the software, a Hexadecimal value 32 (50 decimal value) is expected to be shown, because the software provides 25 axons and 25 dendrites output voltage data through the *cluster_ out_ data* vector signal and these data are read from the memory storage and listed on the screen. The printed axon and dendrite voltages in screen are compared with the ModelSim VHDL simulation result in Table 4.2 and the reference SystemC testbench model, and it shows the similarity between them.

Table 4.3, shows the hardware source usage in different cases. The number 1 is dedicated for hardware source usage of the complete block design. The number 2 is for Neuron Nerwork IP-core (NNIP) with integrated Neuron Network Application (NNA). The number 3 shows the IP design (NNIP) without integrated NNA which is the block with the AXI-Lite Slave and the AXI-Full slave.

Number	Design name	**BRAM (60)**	**DSP48EE (80)**	**FF (35200)**	**LUT (17600)**
1	Complete block design	7.5 (12.5%)	28 (35%)	8674(24.64%)	10696 (60.77%)
2	NNIP Core with integrated NNA	7.5 (12.5%)	28 (35%)	8254 (23.45%)	10027 (56.97%)
3	NNIP Core without integrated NNA	2 (3.33%)	0 (0%)	1054 (2.99%)	696 (3.95%)

Table 4.3: Hardware resource usage for the NNIP core for the Virtex 7 ZyBo-xc7z010clg4 device

5 Conclusion and Future Work

5.1 Conclusion

The proposed system meets all the goals of the thesis presented in Section 1.3. In this thesis, customized slave IP core is designed and implemented which is called the Neuron Network IP-cor (NNIP). The NNIP is compatible with the AXI-Bus interface and it is verified and evaluated based on the reference SystemC testbench model [3].

The main contributions of the proposed system are summarized below:

- I had been provided with a VHDL source code of a Neuron Network Application (NNA). I integrated the NNA into a new customized slave IP-core (it is called the Neuron Network IP-core (NNIP)) with the AXI-Bus interface and then a package IP is created.
- The NNIP in the package IP is verified and evaluated within a VHDL testbench by the Modelsim tool and the results are compared with the reference model(the systemC testbench).
- The Vivado is used to create a block design, and the Zynq-7000 processors as a master is added to connect with the NNIP. Finlay, the HDL wrapper is created, and then synthesize, implement, and bit-stream are generated for the design.
- The hardware design specification and configuration are exported into the Xilinx Software Development Kit (SDK) to write software. This software is written to verify and compare the result of the hardware design with the reference SystemC testbench model and the ModelSim VHDL simulation.
- The software result is compared with the reference SystemC testbench model and the VHDL testbench. It shows that the axon voltages are nearly similar in both simulation (the VHDL and the SystemC) and the dendrite voltages are exactly the same.

This section concludes the work on this thesis. In the next section, possible extensions to the work proposed in this thesis is explained.

5.2 Future work

This section provides ideas for further research interests following from the work on this thesis. These are possibilities to improve the system and furthermore ways to extend the system to increase the functionality.

1. As the NNA application is run just for an iteration, more iterations on the hardware and compare the result with the reference SystemC testbench model are required.

2. The axon and the dendrite voltage after running more iteration need to be stored. Every iteration needs 1.6 KB memory to be stored. Reprogrammable logic equivalent to Artix-7 FPGA in the Zybo board consist of 240KB Block RAM [33]. Then it is possible to store the outputs, just for 150 iteration. But of course more iterations are needed, therefore finding a solution to store the axon and the dendrite voltage after more iteration is needed.

3. Creating a boot image for an application that runs under the standalone from the SD card on the Zybo board.

Appendix

A

Neuron Network Application (NNA) ports

```
1
    clk
    reset
    --Control signals
    s_start
6   cluster_rdy

    --Initialization signals
    cluster_init_type
    cluster_init_clus
11  cluster_init_adr
    cluster_init_adr2
    cluster_init_data
    cluster_init_str
    cluster_init_ack
16
    --Injected signals
    cluster_in_adr
    cluster_in_type
    cluster_in_data
21  cluster_in_str
    cluster_in_ack

    --Axon and Dendrite Voltage
    cluster_out_adr
26  cluster_out_type
    cluster_out_data
    cluster_out_new
```

Appendix B

```
-- ------------------------------------------------------
-- -- Read and write transaction signals of AXI-Lite Slave
-- ------------------------------------------------------
--AXI-Lite signals Name
--Clock and Reset
    S_AXI_ACLK : in std_logic;
    S_AXI_ARESETN: in std_logic;
--Write Address Channel
    S_AXI_AWADDR: in  std_logic_vector(31 downto 0);
    S_AXI_AWVALID: in  std_logic;
    S_AXI_AWREADY: out std_logic;
-- Write Data Channel
    S_AXI_WDATA: in  std_logic_vector(31 downto 0);
    S_AXI_WSTRB: in  std_logic_vector(3 downto 0);
    S_AXI_WVALID: in  std_logic;
    S_AXI_WREADY: out std_logic;
--Read Address Channel
    S_AXI_ARADDR: in  std_logic_vector(31 downto 0);
    S_AXI_ARVALID: in  std_logic;
    S_AXI_ARREADY: out std_logic;
--Read Data Channel
    S_AXI_RDATA: out std_logic_vector(31 downto 0);
    S_AXI_RRESP: out std_logic_vector(1 downto 0);
    S_AXI_RVALID: out std_logic;
    S_AXI_RREADY: in  std_logic;
--Write Response Channel
    S_AXI_BRESP: out std_logic_vector(1 downto 0);
    S_AXI_BVALID: out std_logic;
    S_AXI_BREADY: in  std_logic;
```

Appendix

C

```
-- ------------------------------------------------------------------
-- -- Example code for writing into the local registers on AXI-Lite Slave
-- ------------------------------------------------------------------
slv_reg_wren <= axi_wready and S_AXI_WVALID and axi_awready and
    S_AXI_AWVALID ;

process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then

      slv_reg0 <= (others => '0');
    else
      loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB
          );
      if (slv_reg_wren = '1') then
        case loc_addr is

          when b"000000" =>
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
                -- Respective byte enables are asserted as per write
                    strobes
                -- slave registor 0
                slv_reg0(byte_index*8+7 downto byte_index*8) <=
                    S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
              end if;
            end loop;
          when others =>
            slv_reg0 <= slv_reg0;
        end case;
      end if;
    end if;
  end if;
end process;
```

Appendix

D

```
-- ------------------------------------------------------
-- -- Example code to access registers on AXI-Lite Slave
-- ------------------------------------------------------
slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;

process ( slv_reg0, axi_araddr)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
     case loc_addr is
      when b"000000" =>
    reg_data_out <= slv_reg0;
      when others =>
    reg_data_out  <= (others => '0');
    end case;
end process;

  process( S_AXI_ACLK ) is
begin
  if (rising_edge (S_AXI_ACLK)) then
    if ( S_AXI_ARESETN = '0' ) then
      axi_rdata  <= (others => '0');
    else
      if (slv_reg_rden = '1') then
    -- When there is a valid read address (S_AXI_ARVALID) with
    -- acceptance of read address by the slave (axi_arready),
    -- output the read data
    -- Read address mux
      axi_rdata <= reg_data_out;     -- register read data
      end if;
    end if;
  end if;
end process;
```

Appendix

E

```
-- ------------------------------------------------------
-- -- Example code to access defined logic memory region
-- ------------------------------------------------------

gen_mem_sel: if (USER_NUM_MEM >= 1) generate
begin
  mem_select  <= "1";
  mem_address <= axi_araddr(ADDR_LSB+OPT_MEM_ADDR_BITS downto ADDR_LSB)
      when axi_arv_arr_flag = '1' else
                 axi_awaddr(ADDR_LSB+OPT_MEM_ADDR_BITS downto ADDR_LSB)
                    when axi_awv_awr_flag = '1' else
                 (others => '0');
end generate gen_mem_sel;

-- implement Block RAM(s)
BRAM_GEN : for i in 0 to USER_NUM_MEM-1 generate
  signal mem_rden : std_logic;
  signal mem_wren : std_logic;
begin

  mem_wren <= pm_cluster_out_new  ;
  mem_rden <= axi_arv_arr_flag ;

 BYTE_BRAM_GEN : for mem_byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1)
    generate
   signal byte_ram : BYTE_RAM_TYPE;
   signal data_in  : std_logic_vector(8-1 downto 0);
   signal data_out : std_logic_vector(8-1 downto 0);
 begin

   cluster_data_out_send_toMemory_PROC : process( S_AXI_ACLK ) is
   begin
     if ( rising_edge (S_AXI_ACLK) ) then
       if ( mem_wren = '1' and S_AXI_WSTRB(mem_byte_index) = '1' ) then
         data_in  <= pm_cluster_out_data((mem_byte_index*8+7) downto
             mem_byte_index*8);
       end if;
     end if;

   end process cluster_data_out_send_toMemory_PROC;


   data_out <= byte_ram(to_integer(unsigned(mem_address)));

```

```
    BYTE_RAM_PROC : process( S_AXI_ACLK ) is
    begin
      if ( rising_edge (S_AXI_ACLK) ) then
        if ( mem_wren = '1' and S_AXI_WSTRB(mem_byte_index) = '1' ) then
45
         if (pm_cluster_out_type = "00" ) then
                  byte_ram(to_integer(unsigned(pm_cluster_out_adr))) <=
                     data_in;
               else
                  byte_ram(to_integer(unsigned(pm_cluster_out_adr))+100) <=
                     data_in;
50           end if;
        end if;
      end if;

    end process BYTE_RAM_PROC;
55  process( S_AXI_ACLK ) is
      begin
        if ( rising_edge (S_AXI_ACLK) ) then
          if ( mem_rden = '1') then
            mem_data_out(i)((mem_byte_index*8+7) downto mem_byte_index*8)
               <= data_out;
60
          end if;
        end if;
    end process;

65  end generate BYTE_BRAM_GEN;

  end generate BRAM_GEN;
  --Output register or memory read data

70 process(axi_rvalid , axi_araddr) is
  begin
    if (axi_rvalid = '1') then
      -- When there is a valid read address (S_AXI_ARVALID) with
      -- acceptance of read address by the slave (axi_arready),
75    -- output the read dada
      axi_rdata <= mem_data_out(0);  -- memory range 0 read data
       -- memory range 0 read data
    else
      axi_rdata <= (others => '0');
80  end if;
  end process;
```

Bibliography

[1] Liebgott B: Anatomical basis of dentistry, ed 2, St Louis, 2001, Mosby.

[2] Xilinx, Axi Reference Guide , http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[3] Christiaanse, Gerrit Jan, et al. "A real-time hybrid neuron network for highly parallel cognitive systems." engineering in medicine and biology society (EMBC), 2016 IEEE 38th Annual International Conference of the. IEEE, 2016.

[4] Sklyarov, Valery and Skliarova, Iouliia and Silva, João and Sudnitson, Alexander, Design space exploration in multi-level computing systems, Proceedings of the 15th International Conference on Computer Systems and technologies, 2014.

[5] Xilinx, zynq-7000 product table, http://www.xilinx.com/support/documentation/selection-guides/ zynq-7000-product-selection-guide.pdf.

[6] Vivado Design Suite HLx Editions - Accelerating high level design.

[7] Modelsim Advanced Simulation and debugging , http://model.com/.

[8] Cerebral Hemisperes , http://www.indiana.edu/p1013447/dictionary/cer_hemi.htm, accessed:2013-12-19.

[9] E. J. Fine, C. C. Ionita, and L. Lohr, "The history of the development of the cerebellar examination," Semin Neurol, vol. 22, pp. 375–384, 2002.

[10] D. McDougal, D. van Lieshout, and J. Harting, "Medical Neurosciences." http://www.neuroanatomy.wisc.edu/virtualbrain/brainstem/06olive.html. Accessed: 2014-01-13.

[11] R. Llinas, K. Walton, and E. Lang, The Synaptic Organization of the brain. Oxford: Oxford University Press, 1990.

[12] Wolf U, Rapoport MJ, Schweizer TA (2009). "Evaluating the affective component of the cerebellar cognitive affective syndrome". J. Neuropsychiatry Clin. Neurosci.

[13] Fine EJ, Ionita CC, Lohr L (2002). "The history of the development of the cerebellar examination". Semin. Neurol.

[14] Wisegeek, "What is the somatosensory Cortex." http://www.wisegeek.org/ what-is-the-somatosensory-cortex.htm. Accessed: 2014-01-14.

[15] Dward J. Fine1, Catalina C. Ionita1, and Linda Lohr. "The History of the Development of the Cerebellar Examination". In: Seminars in Neurology (2002), pp. 375 - 384. D O I : 10.1055/s-2002-36759.

[16] J. P. Welsh and R. Llinas, "Some organizing principles for the control of movement based on olivocerebellar physiology," Progress in Brain Research, vol. 114, pp. 449461, 1997.

[17] J. Hofmann, C. Galuzzi, A. Zjajo, and R. van Leuken, "Multi-chip dataflow architecture for massive scale biophysically accurate neuron simulation," ISCAS, 2016 .

[18] E. Izhikevich, "Which model to use for cortical spiking neurons?" Neural Networks, IEEE Transactions on, vol. 15, no. 5, pp. 1063–1070, sept 2004.

[19] G. Smaragdos, S. Isaza, M. F. van Eijk, I. Sourdis, and C. Strydis, "Fpga-based biophysically-meaningful modeling of olivocerebellar neurons," in proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 89–98. [online]. Available: http://doi.acm.org/10.1145/2554688.2554790.

[20] M.F. van Eijk. "Modeling of Olivocerebellar Neurons using Systemc and High-Level Synthesis". MA thesis. Delft, The Netherlands: Delft University of Technology, 2014.

[21] O. S. Initiative et al., "Systemc synthesizable subset draft 1.3," 2009.

[22] IEEE standard for standard Systemc language reference manual, IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), pp. 1–638, Jan 2012.

[23] Opencores, "Wishbone b3, wishbone system-on-chip (soc)interconnection architecture for portable ip cores.".

[24] System-C ,http://accellera.org/downloads/standards/systemc.

[25] Xilinx-ISE design suite ,https://www.xilinx.com/products/design-tools/ise-designsuite.html.

[26] System on a chip, https://en.wikipedia.org/wiki/system_on_a_chip.

[27] Field Programmable Gate Arrays (FPGAs) ,http://www.xilinx.com/training/fpga/fpga-field-programmablegatearray.htm.

[28] Vivado High-Level Synthesis ,http://www.xilinx.com/products/design-tools/vivado/integration/esldesign.html.

[29] Xilinx software development kit (xsdk) , https://www.xilinx.com/products/design-tools/embeddedsoftware/sdk.html.

[30] Xilinx microprocessor debugger (xmd) , http://www.xilinx.com/support/ documentation/sw_manuals/xilinx11/est_rm.pdf.

[31] Xilinx, Xilinx. Zyng-7000 overvie, http://www.xilinx.com/support/ documentation/data_sheets/ds190-zynq-7000-overview.pdf.

[32] D. Dhanalaxmi , V. Roopa Reddy, Sdk and Hardware Platform Specification, Implementation of Secured data Transmission System on Customized Zynq SoC, International Journal of Science and Research (IJSR) ISSN (Online): 2319-7064.

[33] http://www.xilinx.com/support/documentation/university/xup20boards/ xupzybo/documentation/zybo_rm_b_v6.pdf.

[34] https://en.wikipedia.org/wiki/list_of_terminal_emulators.

[35] Sdk and Hardware Platform Specification, https://www.xilinx.com/support /documentation/sw_manuals/xilinx2015_3/sdk_doc/index.html.

[36] Picocom terminal program , https://linux.die.net/man/8/picocom.