POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

# Development of a high-performance Linux device driver for a custom SNN accelerator for Xilinx FPGA boards

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Dott. Alessio CARPEGNA

Candidate

Mauro LANZA

October 2022

# Summary

Spiking Neural Networks (SNN) are an emerging type of Artificial Neural Network (ANN), which takes direct inspiration from the behavior of a biological brain, trying to reach its extraordinary energy efficiency. In particular, similarly to what is observed in biology, neurons exchange information in form of spikes. There are many possible implementations of SNN, from software simulations, to dedicated analog circuits, made of hardware components that mimic the behavior of biological elements. In this scenario there is a full research branch that studies the translation of Spiking Neural Network models into dedicated digital hardware accelerators.
In order to reproduce the behaviour of spikes signals between neurons an Artificial Neural Network uses voltage thresholds that when are passed trigger the firing of the spike. The SNN used for this thesis has a total of 784 inputs and 400 neurons, eahc neuron is dedicated to recognize a particular number. At the end of the calculation, the sum of all the spikes of each neuron is taken and a cross examination is performed: the neurons that was used to recognize a number that had the most number of spikes indicates what number the network recognized.

One possible target platform for such kind of accelerators is a Field Programmable Gate Array (FPGA). This is already present on many systems and doesn't require the integration of a new dedicated component.

This thesis aims to develop a high-performance software driver, developed in C, to interface a custom accelerator for an SNN, implemented on a Xilinx FPGA. A Xilinx ZC702 evaluation board © is used for the development. It hosts a dual-core ARM Cortex-A9 based Processing System (PS) and a Xilinx Artix-7 FPGA based Programmable Logic (PL).
All the hardware development of the board, from the interface to the physical address mapping was done using Vivado©, a tool offered by Xilinx to create custom logics and implement them on the FPGA of the board. The board allows to simulate a complete system, placing custom accelerators side by side to the CPU. It includes the possibility to boot a Linux operating system from an external SD-card. This OS can be customized in almost every way or form. The most interesting part is the possibility to customize the Kernel in order to implement an external Linux Kernel Module inside.

In fact, the final goal of this work is to embed the developed driver inside the Linux kernel, in order to make the usage of the accelerator completely straightforward for user applications.

The Processing System and the Programmable Logic can be interfaced through an Advanced eXtensible Interface (AXI) connection, exploiting an Advanced Microcontroller Bus Architecture (AMBA) communication protocol. This work includes the development of the complete interface required to obtain a communication between the PS and the custom accelerator. Starting from the bare accelerator, the interface was modified in order to efficiently fit an AXI4-lite connection using 2 32-bits registers for the interface.

Then, the component was embedded into an AXI4-lite slave peripheral, automatically mapped in the physical memory to be accessed by the PS. Although, this doesn't allow the peripheral to be accessed by a Linux Kernel Module (LKM) which is the final step. In order to achieve such a goal the address has to be first mapped into the virtual memory of the device so that it can be accessed via c-pointers structures. The final step to complete the communication between Processing System and Programmable Logic was to create a Linux Kernel Module that could map the physical address into a virtual memory and create a character device accessible from the user application to communicate with it.

At last, a C application was developed to practically use the custom accelerator to perform inference on a set of input data, exploiting the developed interface to communicate with it. First, a target application is selected. In particular the MNIST dataset was chosen and the SNN was trained offline using a software simulator. The user application then feeds to the LKM all the parameters that it needs to initialize the SNN (all given by the user): numbers of inputs, numbers of neurons, number of cycles, voltage reset's value, seed for the LFSR. After the network is initialized, the user decides what to do with the MNIST dataset, he can choose to: recognize an image, check if a random image recognized is correct, check the accuracy of the whole network by feeding to it all the 60000 images present in the dataset.

# Acknowledgements

Vorrei cominciare con una premessa: non sono mai stato in grado di scrivere nulla in un italiano quantomento accettabile, quindi chiedo a te, che stai leggendo questo, di prenderne il contenuto e mi auguro che quest'ultimo sia abbastanza comprensibile.
Vorrei dapprima ringraziare i miei genitori e la mia famiglia per essermi stati vicini in questo percorso: non troppo da soffocarmi ma neanche troppo poco da sentirli distanti. Tutto ciò è stato essenziale per permettermi di fare le mie scelte senza dovermi sentire in colpa o obbligato a farne di altre.
Grazie alla mia fidanzata Ludovica per avermi ascoltato blaterare per ore su problemi tecnici che avevo durante dei progetti e grazie per avermi insegnato quanto la forza di volontà di una persona sia in grado di superare ogni ostacolo. E' anche d'obbligo un grazie per tutti quei video di cani e gatti che mi mandi per tirarmi su il morale.
Un grazie speciale va anche al dottorando con cui ho affrontato questo ultimo scoglio, Alessio e al professor Di Carlo grazie al quale ho potuto, per la prima volta in 5 anni di politecnico, eseguire un progetto ingegneristico degno del suo nome.
Un grazie a Massimiliano per essere quell'amico che ha talmente tanti impegni da dimenticarsi pure che si chiama Paolo ma che quando ho avuto bisogno è sempre stato in grado di ritagliarsi 5 minuti per me.
Un grazie a Gianluca per tutte quelle notti passate a guardare film orribili dopo giornate infinite e per essere quell'amico che è sempre dalla tua parte.
Un grazie ad Ahmed Ashraf Ahmed Ibrahim Ahmed (e no, non ho avuto uno spasmo sulla tastiera) per tutte quelle discussioni in cui pur avendo idee diverse siamo riusciti a capirci ed io, personalmente, ad imparare molto.
Un grazie a Fabio per essere stato il miglior collega che potessi desiderare per un lavoro per il quale non eravamo neanche lontanamente qualificati.
Un grazie a Gianluca, Antonio, Davide e Lorenzo per essere stati i miei compagni in questa avventura.
Grazie a Matteo per essere l'"Amico della pale" senza che io vada in palestra (giuro che ci torno prima o poi).
Grazie a tutti i membri del Primo Gruppo: Lorenzo, Stefano, Emanuele, Fabio, Murru, Riccardo, Federico, Daniele.

Grazie a Samuele per avermi ispirato con il suo spirito competitivo.
Un ultimo grazie va a te che hai deciso di leggere tutti questi ringraziamenti. Spero che in mezzo al marasma di persone ti sia trovat*, nel caso non fosse così, sappi che mi sono dimenticato ma in buona fede.

Grazie.

*“If you can laugh in the face of adversity, you are bulletproof”*

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**AI**

Artificial Intelligence

**SNN**

Spiking Neural Network

**ANN**

Artificial Neural Network

**CNN**

Convolutional Neural Network

**DNN**

Deep Neural Network

**AMBA**

Advanced Microcontroller Bus Architecture

**FPGA**

Field Programmable Gate Array

**OS**

Operating System

**I/O**

Input/Output

**SoC**

System-on-Chip

**PL**

Programmable Logic

**PS**

Processing System

**SD**

Secure Digital

**UART**

Universal Asynchronous Receiver-Transmitter

**USB**

Universal Serial Bus

**AHB**

Advanced High-performance Bus

**APB**

Advanced Peripheral Bus

**GPIO**

General Purpose Input-Output

**DMA**

Direct Memory Access

**AXI**

Advanced eXtensible Interface

**MSB**

Most Significant Bit

**LSB**

Least Significant Bit

**LFSR**

Linear Feedback Shift Register

| | |
|---|---|
| **HDL** | Hardware Description Language |
| **HDF** | Hardware Description File |
| **FSBL** | Fist Stage Boot Loader |
| **MNIST** | modified National Institute of Standards and Technology database |
| **LKM** | Linux Kernel Module |
| **FAT** | File Allocation Table |
| **LUT** | Look-Up Table |

# Chapter 1

# Introduction

A device driver is a program that allows the communication between an external device connected to the Operating System. In the case of our ZC702 board, the external device is the SNN mounted on the Field Programmable Gate Array. There are 4 different type of device driver

- BIOS: it's the first program that gets booted when the OS is turned ON

- Motherboard: these drivers are the ones responsible for basic devices that needs to be connected with the OS such as the USB driver or the Input/Output for keyboards and mouses.

- Hardware: these kind of drivers are the ones this theses focuses on. They are the drivers that allow external hardware devices to be interconnected with the OS.

- Virtual Device: these kinds of drivers are used to emulate the function of an external hardware without it being really connected to the PC.

This thesis aims to create a driver of type "Hardware" that manages the exchange of information between the SNN and the ZynQ processor of the ZC702 board. The images that will be sent to the Network will be in an *IDX* format, the file is structured in the following way

| Offset | Type | Value | Description |
|---|---|---|---|
| 0000 | 32 bit integer | 2051 | Magic number |
| 0004 | 32 bit integer | 60000 | Number of images |
| 0008 | 32 bit integer | 28 | Number of rows |
| 0012 | 32 bit integer | 28 | Number of columns |
| 0016 | unsigned byte | - | Pixel Value |
| xxxx | unsigned byte | - | Pixel Value |

**Table 1.1:** IDX Train Images format[1]

| Offset | Type | Value | Description |
|---|---|---|---|
| 0000 | 32 bit integer | 2051 | Magic number |
| 0004 | 32 bit integer | 60000 | Number of items |
| 0008 | unsigned byte | - | Label |
| 0012 | unsigned byte | - | Label |
| 0016 | unsigned byte | - | Label |
| xxxx | unsigned byte | - | Label |

**Table 1.2:** IDX Train Images Labels format[1]

The C application will open the IDX file and extract the Magic Number, Number of images, number of rows and columns in order to set its parameters for the scan of the images inside of it.
The magic number is set in the following way:

- The 2 bytes (start counting form the MSB) are always 0.
- The 3rd byte represents the type of data
  - 0x08 - unsigned byte
  - 0x09 - signed byte
  - 0x0B - short (2 bytes)
  - 0x0C - int (4 bytes)
  - 0x0D - float (4 bytes)
  - 0x0E - double (8 bytes)
- The 4th byte represents the number of dimensions of the vector/matrix

The pixels, as the Table 1.1 shows, are ordered one pixel per 4 of offset. Since each pixel has a value that goes from 0 to 255 in a scale of grays (0 means white, 255

means black), only 8 bits of the DATA signal will be used.
In this case we have 28 number of rows and 28 number of columns. Scanning this file just means using a for loop that goes from 0 to $28 * 28 - 1 = 783$. Each cycle of the loop will send to the SNN a different pixel on a different address.
The Kernel module will have the task to get the physical address where the AXI4-Lite peripheral is located and map it into a virtual address, accessible via pointer.
The whole project will have the following parts:

- 1) An interface used to connect the SNN to the AXI bus peripheral
- 2) A Kernel module to open the communication between the AXI peripheral and the C application
- 3) A C application accessible by the user to test the SNN

# Chapter 2

# Brief overview of the ZC702 board

The ZC702 evaluation board provides an hardware environment to test designs that targets the Zynq ® XC7Z020-1CLG484C SoC. This board provides many features, the one in which we are interested are:

- Zynq ® XC7Z020-1CLG484C
- 1GB DDR3 memory component
- USB JTAG
- Programmable Logic
- Status LEDs to check the FPGA status
- SD connector
- USB to Universal Asynchronous Receiver-Transmitter (UART) bridge

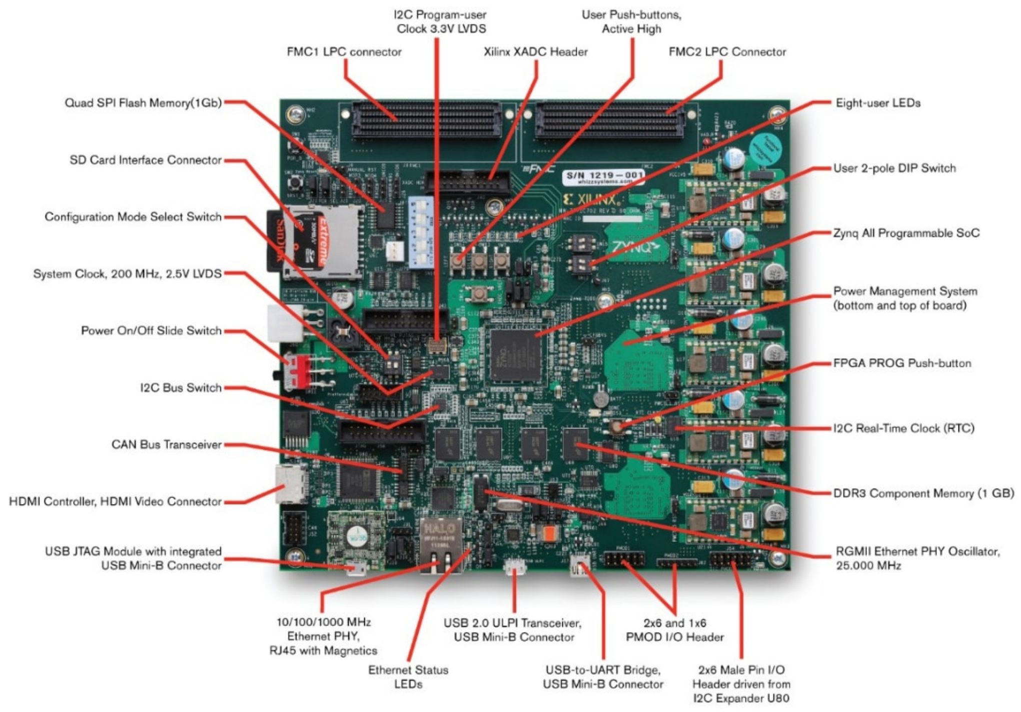

I2C Program-user Clock 3.3V LVDS
User Push-buttons, Active High
FMC1 LPC connector
Xilinx XADC Header
FMC2 LPC Connector
Quad SPI Flash Memory(1Gb)
Eight-user LEDs
SD Card Interface Connector
User 2-pole DIP Switch
Configuration Mode Select Switch
Zynq All Programmable SoC
System Clock, 200 MHz, 2.5V LVDS
Power Management System (bottom and top of board)
Power On/Off Slide Switch
FPGA PROG Push-button
I2C Bus Switch
I2C Real-Time Clock (RTC)
CAN Bus Transceiver
DDR3 Component Memory (1 GB)
HDMI Controller, HDMI Video Connector
USB JTAG Module with integrated USB Mini-B Connector
RGMII Ethernet PHY Oscillator, 25.000 MHz
10/100/1000 MHz Ethernet PHY, RJ45 with Magnetics
USB 2.0 ULPI Transceiver, USB Mini-B Connector
2x6 and 1x6 PMOD I/O Header
Ethernet Status LEDs
USB-to-UART Bridge, USB Mini-B Connector
2x6 Male Pin I/O Header driven from I2C Expander U80

**Figure 2.1:** ZC702 Evaluation Board (courtesy image from Mouser®)

## 2.1 XC7Z020-1CLG484C SoC

The XC7Z020-1CLG484C SoC consists of a Processing System and a Programmable Logic integrated on a sigle die.
The PS integrates two ARM® Cortex-A9 Core, to be able to use this kind of processor all the C application has to be cross-compiled.
By using a combination of the 5 switches located near the SD card interface the board can be booted in various mode.

| BOOT Mode | Code |
|---|---|
| JTAG Mode | 00000 |
| Independent JTAG Mode | 10000 |
| Quad SPI Mode | 00010 |
| SD Mode | 00110 |

**Table 2.1:** BOOT Modes of the ZC702[2]

Of those 4 modes only the JTAG and the SD have been used.
In particular:

- JTAG: used to debug the applications without loading them into the SD card. this task can be accomplished by simply connecting both the JTAG and the USB-to-UART to the PC and using a terminal to access the board.
- SD Mode: used in the final stages of the project

The next figure represents the High Level Block Diagram of the SoC. In particular we can see that the Processing System comprehends the AMBA interface which we will access from our driver to move the image pixel's data to the neural network placed onto the FPGA of the board.

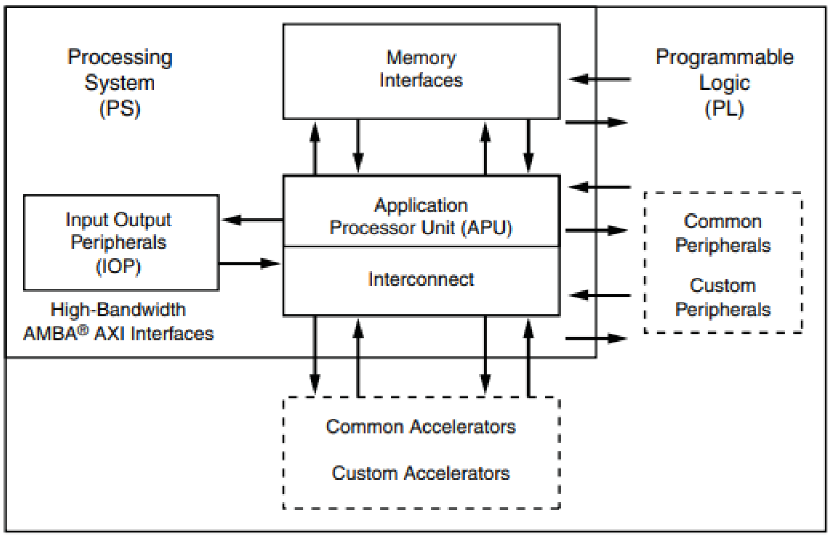


**Figure 2.2:** XC7Z020 SoC High Level Block Diagram[2]

## 2.2 The ZC702's FPGA

A Field Programmable Gate Array (FPGA) is a type of pre-fabricated sylicon device that can be electrically programmed to implement almost any kind of digital system[3]. In addition to such functional flexibility one part of the FPGA can be programmed while another section is still running a previously loaded digital circuit.

An FPGA normally comprehends various parts such as:

- Programmable logic blocks - implement logic functions
- Programmable routing - connect the logic functions
- I/O blocks that allow various external connections to occur with the logic blocks

The status of the FPGA of the ZC702 can be easily checked by looking at the status LED.
The FPGA can also be seen as a Programmable Logic (PL) which can be accessed from the Processing System (PS) via the AXI bus. This interface will be further explored in chapter 4.

## 2.3 USB Connections - JTAG

There are 2 main USB connectors on the Xilinx ZC702 Board:

- USB-to-UART bridge
- USB JTAG Module

In this section we will briefly explore both type of connectors and see how they have been used and to what extend in this thesis project.

### 2.3.1 USB-to-UART bridge

The Universal Asynchronous Receiver-Transmitter (UART) is one of the most used device-to-device type of connection protocol. It works by connecting the Receiver (RX) end of one UART device to the Transmission (TX) end of another UART device. To fully understand how the UART protocol works we also need to understand at what speed does this interface work. This speed is called *Baud Rate* and represents the number of Bits trasmitted per second. Two UART interfaces that needs to interact must be working at closely the same Baud Rate (with a maximum of 10% error).
The connection on the considered platform works with a Baud Rate of 115200.
The Baud Rate isn't the only thing that has to be the same between the two devices communicating, besides that they need to have the same:

- Number of N bits inside a character sent
- Presence of parity: None, Even, Odd, Mark or Space.
- Number of STOP bits used to signal the end of the transmission (either 1 or 2)

In this type of connection protocol both the devices are connected to the same clock, the peculiarity is that the clock is used only once for the synchronization of the two: in particular it sets the phase shift between active edges. This type of

clock signals are called "isofrequential independent clocks".
One point to be careful on is the fact that the devices are different and so they are subjected to some tolerance issues due to their nature. It may happen (without any prevision possible) that both of them, after being synchronized, go out of phase. To solve this issue a periodic synchronization has to be performed, in order to know how frequently this synchronization is needed one has to study the eye diagram of the devices.

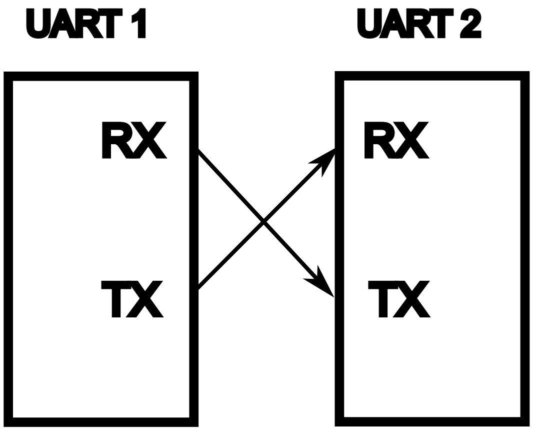


**Figure 2.3:** UART Protocol scheme

The USB used for this bridge is of type Mini-B. It's a USB 2.0 so its speed goes up to $50Mb/s$, often it is limited to $35Mb/s$. This kind of USB has 5 pins described as seen in the Table 2.2

| Pin Number | Signal | Description |
|---|---|---|
| 1 | VBUS | Connected to 5V |
| 2 | D- | Pin for the Data- |
| 3 | D+ | Pin for the Data+ |
| 4 | ID | Pin left unconnected |
| 5 | GND | Ground |

**Table 2.2:** USB Mini-B Pin Description

As we can see the 2 protocols are different, that's why the interface between them is called "USB-to-UART bridge", which means it translates data between the two interfaces to allow a device to send/receive information on one interface and receive/send the information on the other one. When the data is received, it is stored in the memory waiting for the other interface to be ready. To check if any type of data is being received the *polling* mechanism is used. This type of method is one of the most expensive in terms of power since it consists in an infinite loop that keeps checking the ports (both the USB and UART) for a new information. The only way to stop this loop is either from the user by forcingly close it or by disconnecting the USB cable.

# Chapter 3

# Introduction to SNN

A Neural Network is defined as a computation model created in order to try mimicking the functionality of a biological brain. In order to do such a task, it is composed by Artificial Neurons which takes an input, pass it through a function and elaborates an output.
*"The spike-based computations have a high potential of achieving cost-effective inferencing with their low-precision data representations, simple neuron operations, and new parallelization opportunities"[4]* that's why the SNN have been chosen to perfmorm the task of hardware accelators.

## 3.1 Brief Overview of SNN

In history if we classify them for their computational methods, 3 type of Neural Networks have been used the following list divides them by generation [5]

- First Generation: can only give digital output
- Second Generation: these networks use an "activation function" with a continuous set of possible output values to a weighted sum of the inputs
- Third Generation: this generation is the one which implemented the Spiking Neurons. Their job is not to transmit information at each cycle but rather to use spikes in the voltage that are greater than a threshold given in order to send those informations.

The SNN are the natural evolution of the ANN (Artificial Neural Network) being more efficient in terms of power and more easy to implement on hardware. The reason for moving from the ANNs to the SNNs was the attempt to emulate as better as possible the behaviour of the human brain. The information between

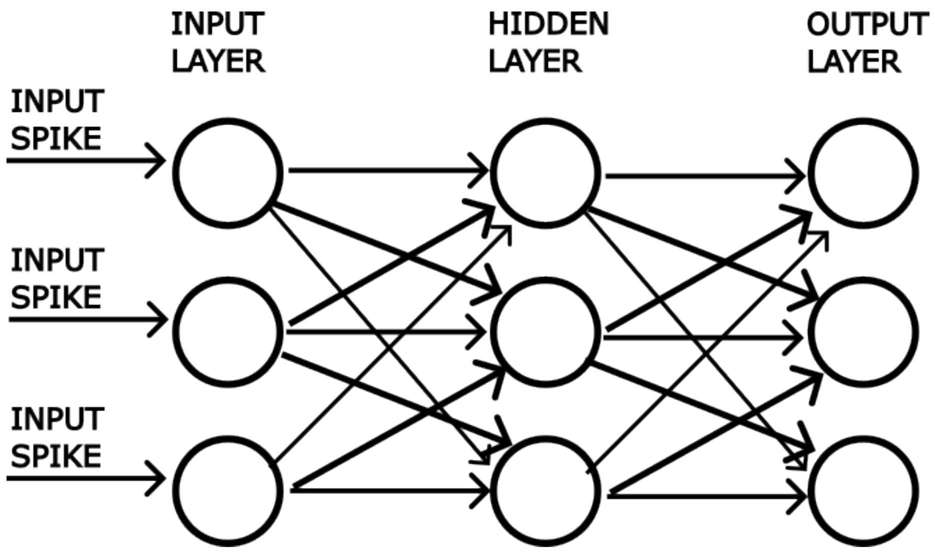


**Figure 3.1:** Generic shape of an SNN

biological neurons propagates via a train of spikes. These individual spikes are scattered in time but with a uniform amplitude (around $100mV$)[6]. So in an SNN it is fundamental that the delays as well as the rates are covered as nicely as possible.
Studies have shown that biological neurons don't always fires (meaning it will generate a signal) in a reliable manner, only under certain conditions that is verified. In order to reproduce such a behaviour *W. Maass* in 1996 proposed a stochastic model for the SNN. This means that a neuron may or may not fire in the time interval that is set for it to perform such action. Each network has a response and a threshold function used to understand the probability and the thresholds at which the neurons will fire.
Of course the simpler mathematical model is the one where the firing of the neurons are deterministic. Each time a Neural Network is used we feed it $N$ number of Neurons and $I$ number of inputs as we will see later.

## 3.2 Utilities of SNN

If SNN are used via dedicated platforms that allow it to recognize audio signals as well as object recognition [7].
As it is possible to see from Figure 3.1, an SNN is an oriented graph, where each neuron is a system with a mixture of inputs that produces a series of spikes. So, at each time $t$ the neuron will emit a spike that has a value of *1*.
The usage of SNN for image recognition is the one applied by the network that

is used in this thesis. Its learning method is called "unsupervised". The main difference between supervised and unsupervised learning is the labeling for the training set of data.

- Supervised Learning: use input and output data which are labeled
- Unsupervised Learning: use unlabelled data

The database used to perform the training is called MNIST (modified National Institute of Standards and Technology database) which is very common. It consists on a series of images of number written by hand.

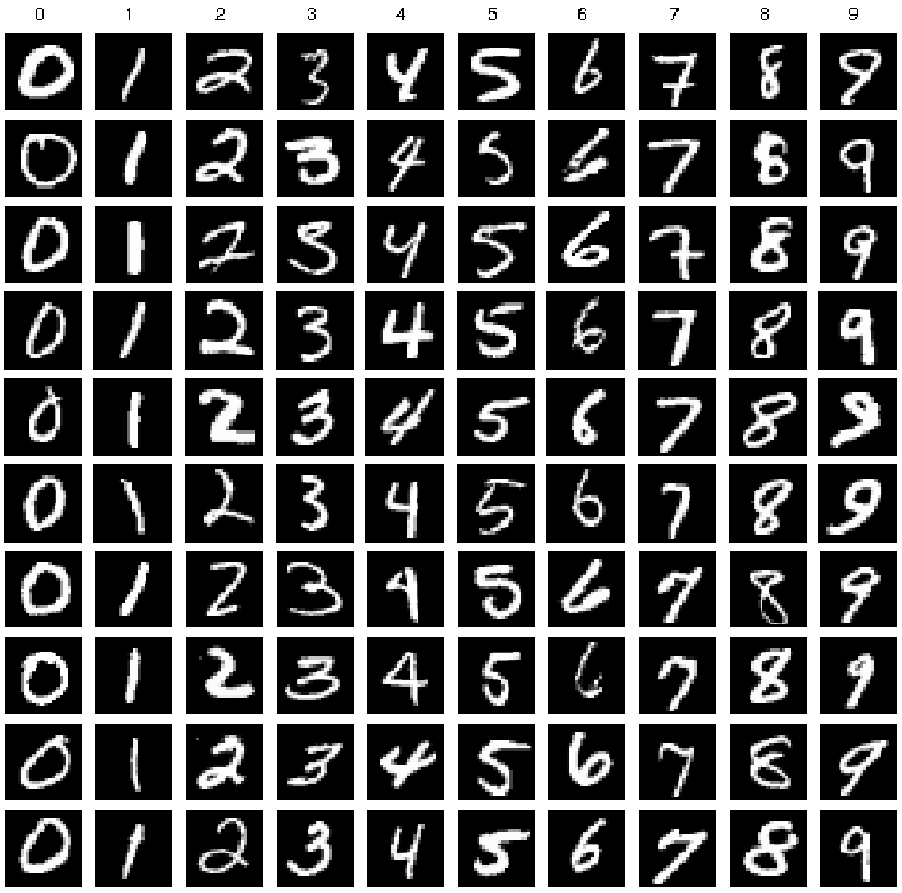


**Figure 3.2:** MNIST Examle

The data is stored in a very simple file format designed for storing vectors and multidimensional matrices. The training set [1] used for our Spiking Neural Network is composed of 60000 images. Its format is called IDX which is a simple format for vectors and multidimensional matrices of various numerical types.

## 3.3 Multi-layer SNN

SNNs are a good way to create ultra-energy-efficient hardware but their performance isn't the best with respect to the traditional methods. By using multiple layers is possible to decrease this gap. We are going to analyze what happens when an interval is set between the firing of each neuron. In particular this method leads to state-of-the-art classification rates on the MNIST dataset (98.60%) and the Faces/Motorbikes dataset (99.46%).
The gap cited above is in part due to the training method used for these types of networks: even tho it is efficient for them, is worse than the one used for traditional networks.
In order to achieve a better performance on SNNs a key thing to set the attention on are the parameters, in particular the voltage threshold. This is what set how hard it is for a SNN to fire a spike. A way to increase the performance from this side is to have a time $t_{target}$ at which a neuron must fire.[8]
It is necessary to create a function for encoding the signal that the neuron send via spikes and also a function to decode spike trains at the output of the network. This is called *Neural Coding* and can be mathematicaly described as:

$$f : [0,1] \longrightarrow R_{+}^{N_x}$$

$$x \longrightarrow (t_0, t_1, t_2, ...t_{N_x})$$

There are 2 types of coding:

- Frequency: uses spike frequency to encode values
- Temporal: uses timestamps of spikes to encode values

One of the most used method is the latency coding where early spikes encode the largest values, while late spikes encode the lowest values. The mathematical description of this latency coding is the following one:

$$t = T_{START} + (1 - x) \times (T_{END} - T_{START})$$

- $T_{END}, T_{START}$: time range of the sample
- $t$: timestamp of the generated spiker
- $x$: input value in range [0,1]

## 3.4 Spiker - VHDL Custom Neural Network

Spiker is the name given to a SNN created by Carpegna Alessio for his thesis in 2021 in Politecnico of Turin. The aim of this thesis is to create an interface, a driver and a C application to pilot Spiker. Although, firstly we need to understand how this networks actually perform its operations.

| Name | Description |
|---|---|
| CLK | Clock for the network |
| RST_N | Reset signal active-LOW |
| RDEN | Used to enable the register |
| START | Start the calculation |
| LOAD | Used to index the data passed to the right part |
| ADDRESS | Address of the BRAM |
| SELECT | Neuron selector |
| DATA | Input Data |
| CNT_OUT | Output of a neuron |
| Ready | Used to check if the network is ready |

**Table 3.1:** Signals needed by Spiker

The CLK signal is the only one that isn't piloted by the interface, all the other are passed through 2 registers of 32 bits each as it is explained in Table 4.12.
The first thing to do is loading the following parameters:

- Number of Inputs
- Number of Neurons
- Number of Cycles to perform
- LFSR seed
- Inhibitor weight

After that, we have to load the weight for each neuron and the voltage threshold that if is surpassed is gonna produce a spike.
Once all the signals are loaded, in each input is loaded the value of a pixel of the image. Since the images are $28 \times 28$ the total number of inputs are 784 so a cycle of 784 iterations is performed to load all the pixels.
Since the images are stored using the IDX format each pixel is in a scale of gray with each value going in the range $0 - 255$, 8 bits are enough to cover this range.
After loading all the data the start signal is given and the network starts computing

the output. The finishing signal is given when Ready goes to 1. The software checks via polling when this happens and starts the scan for the output values. Checking the output values means checking the counter $CNT\_OUT$ by switching the value of the SEL signal. All the neurons have a specific weight and label that specify which number that neuron is "looking for". For instance: if the neuron is checking the pixels for the number "1" it will produce spikes (and so increment the counter) if the number "1" is recognize. Once the computation ended the check of all the neurons will produce the sum of all the spikes generated by all the neurons checking for their number. By looking at which label has the most number of spikes produces we can see what is the number recognized by the network and subsequently compare it with what number it really is in order to check the efficiency.

Chapter 4

# The Advanced Microcontroller Bus Architecture

## 4.1 Introduction

The AMBA is a type of bus created by ARM and dedicated to trasmits data over an SoC device. It can work in 3 different ways:

- Synchronous
- Semi-synchronous
- Asynchronous

Each transmission has 3 (or 4) phases:

- Arbitration (Arb)
- Addrissing (Add)
- Transmission (Data)
- Acknowledge (Ack) - this is the only one which is optional

By taking into consideration onlythe 3 needed phases is possible to create 2 main scenarios:

- No-Pipeline: The latency is $3T_{ck}$ since the phases occur one after the other and so the throughput is $\frac{1}{3T_{ck}}$

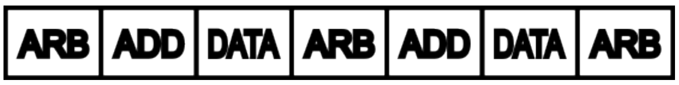


**Figure 4.1:** AMBA communication protocol without a pipeline architecture

- Pipeline: The latency remains $3T_{ck}$ but the throughput is $\frac{1}{T_{ck}}$

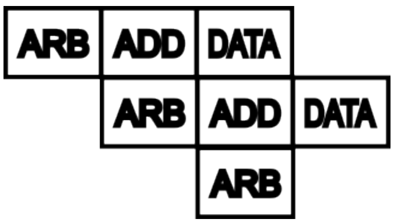


**Figure 4.2:** AMBA communication protocol with a pipeline architecture

There are 2 main interfaces of the AMBA protocal:

- AHB - Advanced High-performance Bus
- APB - Advanced Peripheral Bus

## 4.2 AHB & APB

This 2 types of busses connect different parts of the board.
The AHB is the main bus used by the processor to interface it with its memories and with the DMA (Direct Memory Access).
The APB is used to connect external peripherals such as timers and GPIO ports.

### 4.2.1 Advanced High-performance Bus

The AHB has the most signals between the two of them.

| **Signal** | **Description** | **Source** |
|---|---|---|
| CLK | Clock signal | Master |
| Reset | Reset signal | Master |
| TRANS | Type of transfer to perform | Master |
| WRITE | Write signal (active-HIGH) | Master |
| SIZE | Size of the data to transmit (it must be a power of 2) | Master |
| BURST | Burst size | Master |
| WDATA | Signal that carries the bits to transmit | Master |
| ADDR | Signal that carries the address | Master |
| SEL | Chip select signal | Master |
| RDATA | Data sent back to be read | Slave |
| ACK | Acknowledge bit used to signal that the peripheral is ready | Slave |
| RESP | Transfer response | Slave |
| BUSREQ | Signal to request the use of the bus | Slave |

**Table 4.1:** Table of the AHB signals

There are 4 main blocks in this type of interface that receives these signals and elaborates them:

- Master
- Slave
- Arbiter
- Decoder

While the Master and the Slave are the one that send and receive the data, the other two are control blocks used to allow this transfers without any issues.
In particular the Arbiter receives as input the request, address and control signals. Its outputs are used to grant access to the peripheral in question. The Decoder has a more simple structure receiving the address as an input and sending the SEL signal as output.
Both the Arbiter and the Decoder are used to driver a series of multiplexer to do two main tasks:

- Avoid any conflict on the bus
- Select the signal change based on the bus phase

The connection between those blocks can be seen in Table 2.2

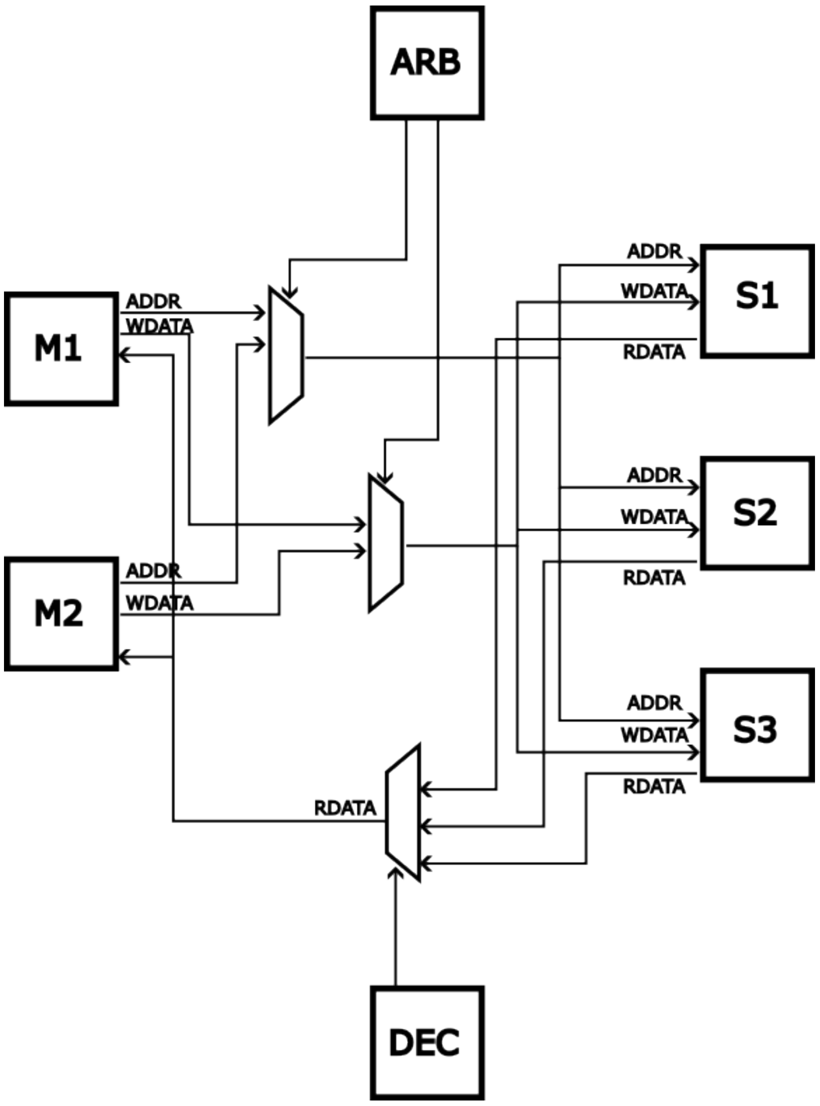


**Figure 4.3:** AHB connection

The read and write cycles have 2 phases:

- Address Phase: lasts 1 clock cycle and provides information about the address and the type of transaction (burst, ...)
- Data Phase: the bits are sent/received, this phase may last for more than 1 clock cycle

The Arbitration phase has to be performed before any transaction occurs. Here the master request (via the BUSREQ signal) the bus and waits for it to send the GRANT signal. Once this happen, the master waits for the READY signal to

become HIGH. To check which of the Masters has the bus occupied we can look at the MASTER signal.
An error that can occur during the transmission of data between a Master and a Slave is caused by the difference of speed between them. If the Slave is slower it can request the to suspend a bus operation. Depending on whether the master keeps or not his priority this operation can take 2 different names:

- SPLIT: master has no priority
- RETRY: master keeps his prioprity

### 4.2.2 Advanced Peripheral Bus

This type of bus is used for the peripheral, such as timers or GPIO, and so it has a slower clock frequency as well as fewer wires connected. It has no pipeline, no arbitration and has a single Master.

| **Signal** | **Description** | **Source** |
|---|---|---|
| CLK | Clock signal | Master |
| Reset | Reset signal | Master |
| Addr | Address signal | Master |
| WR | Write/Read control | Master |
| SEL | Chip select signal | Master |
| ENABLE | Transfer Enable | Master |
| WDATA | Signal that carries the bits to transmit | Master |
| RDATA | Data sent back to be read | Slave |

**Table 4.2:** Table of the APB signals

All the transmission lasts for 2 clock cycles that's why it isn't necessary to have a pipelined architecture.

- First CC:
  - ADDR signal contains the address of the Slave register
  - SEL signal of the slave is set to 1 while the others remain to 0
  - WR signal is set to the operation that we want to perform
  - ENABLE remains at 0
- Second CC:
  - ADDR keeps the previous value

  - SEL keeps the previous value
  - WR keeps the previous value
  - ENABLE is set to 1
  - WDATA or RDATA is used to perform the wanted operation

This interface is much simplier to use than the AHB one

## 4.3 The AXI Protocol

The Advanced eXtensible Interface (AXI) is a communication bus protocol developed by ARM for the AMBA 3 and 4, released in 2003 for the AMBA 3.
This type of protocol is very useful to our project since it provides flexibility in the implementation of the interconnect architectures. It also has many features that makes it a perfect fit for us, in particular it has [9]:

- separate address/control and data phases
- support for unaligned data transfers, using byte strobes
- separate read and write data channels, that can provide low-cost Direct Memory Access (DMA) - this is a great addition to the bus since our Spiking Neural Network has a high cost in terms of power consumed at the moment
- support for issuing multiple outstanding addresses

This protocol is burst-based and to use it we only need a few core channel signals:

- Read Data
- Write Data
- Read Address
- Write Address
- Write Response (optional)

In the case of our Vivado project there are 3 different types of AXI: AXI4-Lite, AXI4 and AXI4-Stream. Independentely of which one of these interface we chose (we will see them more in details later) they all wrok with the same concepts.
Two signals, a VALID and a READY signal are exchanged between the MASTER and the SLAVE as an handshake between them.
The SLAVE outputs to the READY signal to the MASTER to indicate him that

he is ready to accept the address and the control signals for the write of the data. The MASTER outputs the VALID signal to the SLAVE to indicates that the write address and control informations are both valid.
For each of the 5 (or 4 without counting the Write Response) channel signals we con have multiple different signals each of the them used to either control the address or data validity or to send them and check the correctness of thi last step.

### 4.3.1 AXI4

The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between Masters and Slaves components.

- *AWx* - Write Address Channel Signals
- *Wx* - Write Data Channel Signals
- *Bx* - Write Response Channel Signals
- *ARx* - Read Address Channel Signals
- *Rx* - Read Data Channel Signals

Following a description of all the signals used in the AXI4 interface:

| **Signal** | **Description** | **Source** |
|---|---|---|
| ACLK | Clock signal | Clock Source |
| ARESETn | Reset signal active-low | Reset Source |
| AWID | Identification tag for a write transaction | Master |
| AWADDR | Write Address signal | Master |
| AWLEN | Burst Leght for a write transaction | Master |
| AWSIZE | Number of bytes in each data transfer | Master |
| AWBURST | Burst type | Master |
| AWLOCK | Atomic characteristic of a write transaction | Master |
| AWCACHE | Write transaction requirement to go through a system | Master |
| AWPROT | Privilege/security level of the write transaction | Master |
| AWQOS | Quality of Service identifier for a write transaction | Master |
| AWREGION | Region Identicator for a write transaction | Master |
| AWUSER | User-defined extension for a write address channel | Master |
| AWVALID | Validity of address and control signals | Master |
| AWREADY | Slave is ready to accept the Address and Control signals | Slave |
| WID | ID tag of the write data transfer (only on AXI3) | Master |
| WDATA | Write Data | Master |
| WSTRB | Which byte lanes holds the valid data | Master |
| WLAST | Indicates if this is the last data transfer in a transaction | Master |
| WUSER | User-defined extension for a write data channel | Master |
| WVALID | Validity of DATA and STRB | Master |
| WREADY | Slave is ready to accept the write data | Slave |
| BID | Identification tag for a write response | Slave |
| BRESP | Status of the write transaction | Slave |
| BUSER | User-defined extension for a write response channel | Slave |
| BVALID | Validity of Response | Slave |
| BREADY | Master can accept write response | Master |
| ARID | Identification tag for a read transaction | Master |
| ARADDR | Read Address signal | Master |
| ARLEN | Burst Leght for a read transaction | Master |
| ARSIZE | Number of bytes in each data transfer | Master |
| ARBURST | Burst type | Master |
| ARLOCK | Atomic characteristic of a read transaction | Master |
| ARCACHE | Read transaction requirement to go through a system | Master |
| ARPROT | Privilege/security level of the read transaction | Master |
| ARQOS | Quality of Service identifier for a read transaction | Master |
| ARREGION | Region Identicator for a read transaction | Master |
| ARUSER | User-defined extension for a read address channel | Master |
| ARVALID | Validity of address and control signals | Master |
| ARREADY | Slave is ready to accept the Address and Control signals | Slave |
| RID | Identification tag for Read Data | Slave |
| RDATA | Read Data | Slave |
| RRESP | Status of the read transaction | Slave |
| RLAST | Indicates if this is the last data transfer in a transaction | Slave |
| RUSER | User-defined extension for a read data channel | Slave |
| RVALID | Validity of Data | Slave |
| RREADY | Master is ready to accept the write data | Master |

**Table 4.3:** AXI4 Signals Description[9]

### 4.3.2 Handshake

In this part we will call the two communicating devices "Source" and "Destination" since both the Master and the Slave have their own same signals to handshake one another.
The two signals that we are looking for are the **VALID** and the **READY** signal. The VALID is used, as written above, to signal the Destination that the data that is about to be sent is complete, while the READY signal is used to signal the Source that the Destination is able to receive the data about to be sent.
There can be 3 types of behaviour

- VALID emitted before READY
- READY emitted before VALID
- VALID and READY emitted at the same time

In all of those cases the handshake occurs only when the next rising edge of the clock samples both the signals to 1. The following 3 images describe the different behaviours from a waveform point of view.

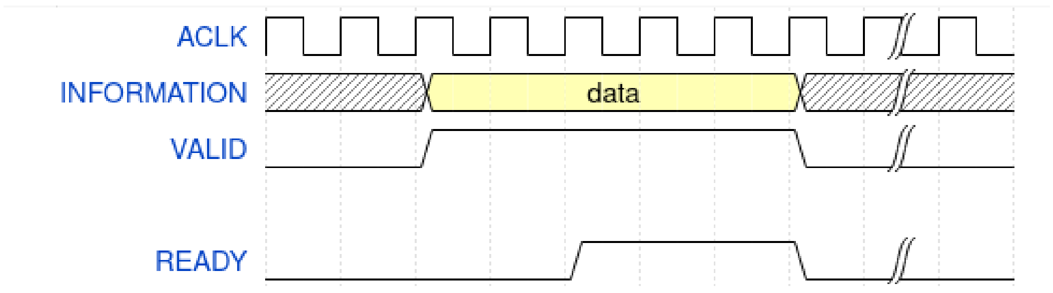


**Figure 4.4:** AXI4 Handshake Waveform VALID before READY

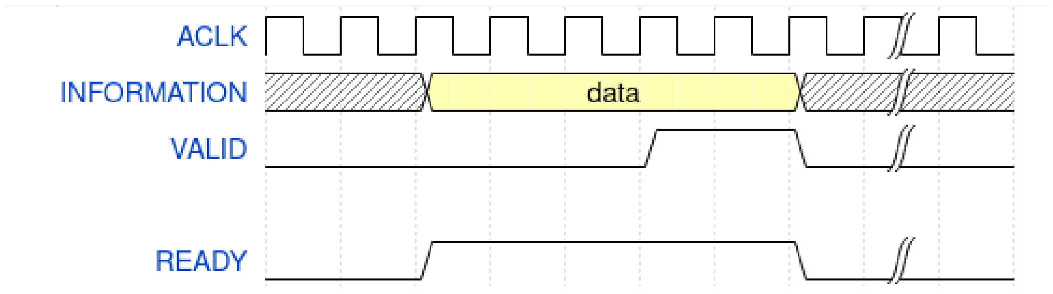


**Figure 4.5:** AXI4 Handshake Waveform READY before VALID

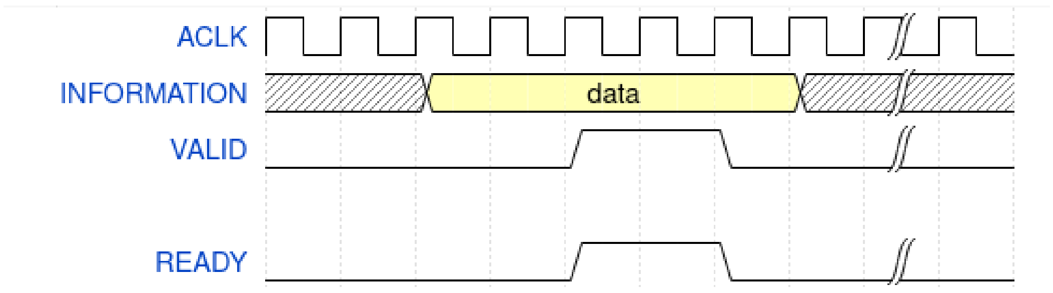


**Figure 4.6:** AXI4 Handshake Waveform READY and VALID at the same time

To implement this handshake there are 2 possible solutions:

- 1) We set the signals to 1 as soon as possible
- 2) We wait for one of the two (VALID or READY) signal before asserting the other. This can create more efficient designs.

In the AXI4 peripheral this 2 solutions are mixed, depending on the transaction we are either forced to use the first or we can choose to use the second.
In the following graphs the dependencies between the signals are shown, on the arrows there is the solution that is used for that specific correlation. For instance the arrow that goes from ARVALID to ARREADY has "1/2" which means that they can be asserted in whichever order we like.
RVALID receives 2 arrows with both denomination being to "1", which means that both those signals needs to be asserted before we can assert it.

$$\begin{tikzcd}
\text{ARVALID} \arrow[dd, "1/2"] \arrow[dr, "1"] & & \\
& \text{RVALID} \arrow[r, "1/2"] & \text{RREADY} \\
\text{ARREADY} \arrow[ur, "1"'] & &
\end{tikzcd}$$

**Figure 4.7:** AXI4 Handshake Read Transaction Dependencies

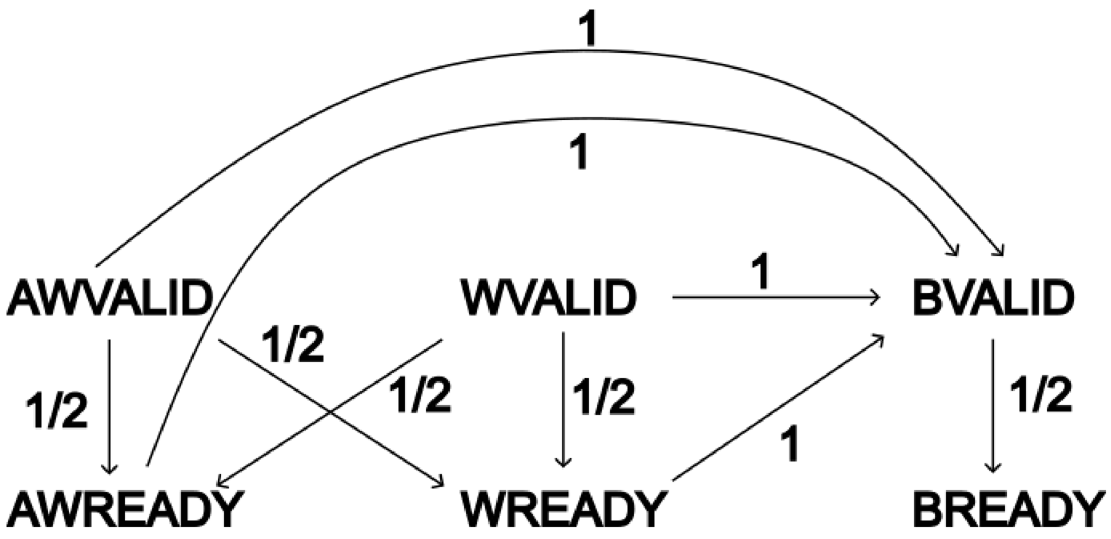


**Figure 4.8:** AXI4 Handshake Write Transaction Dependencies

### 4.3.3 Transaction Status

#### Address Structure

The AXI protocol is burst-based. The Master begins each burst by driving control information and the address of the first byte in the transaction to the Slave. As the burst progresses, the Slave must calculate the addresses of subsequent transfers in the burst. To avoid crossing the boundaries between two Slaves, the burst must not exceed $4KB$.
The **Burst Lenght** is specified by two signals of 8 bits:

- AWLEN[7:0] for write transfers
- ARLEN[7:0] for read transfers

The AXI protocol follow this sequence of rules to use the burst:

- For wrapping bursts, the burst length must be 2, 4, 8, or 16. - if the lenght exceeds 16, it can be divided into smaller bursts.
- A burst must not cross a 4KB address boundary
- Early termination is not supported

The **Burst Size** is specified by two signals of 3 bits:

- AWSIZE[3:0] for write transfers
- ARSIZE[3:0] for read transfers

The rule is simple:

$$Bytes\_in\_Transfer = 2^{AxSize}$$

| **AxSIZE** | **Bytes in Transfer** |
|---|---|
| 000 | 1 |
| 001 | 2 |
| 010 | 4 |
| 011 | 8 |
| 100 | 16 |
| 101 | 32 |
| 110 | 64 |
| 111 | 128 |

**Table 4.4:** Burst Size's Encoding

The **Burst Type** is specified by 2 signals of 2 bits:

- ARBURST[1:0] for read transfers
- AWBURST[1:0] for write transfers

The comination of 2 bits produces $2^2$ combinations described as below

| **AxBURST** | **Burst Type** |
|---|---|
| 00 | FIXED |
| 01 | INCR |
| 10 | WRAP |
| 11 | reserved |

**Table 4.5:** AXI4 Burst Types

We can descrive each of the type as follows

- FIXED - the address is always the same for every burst
- INCR - the address of each burst is incremented by one from the address of the previous bus
$$Burst\_Lenght = AxLEN + 1$$
- WRAP - similar to an incrementing burst, except that the address wraps around to a lower address if an upper address limit is reached.

Even tho there are many options for the burst transaction some interfaces only uses a few a of them. If the Master of a system only abilitates few of those options, the Slave can be designed with a simpler logic in order to reduce the space needed to implement it.

### Data Structure

To check where the bytes that are being sent are present, a signal called **WSTRB** is used. This is called "Write Strobe" and specify the byte lanes of the data bus that contain valid information. Since $1\_BYTE = 8\_BITS$, there is one strobe per 8 bits of data. A Master must ensure that the WSTRB goes to 1 only for the lane that contains the valid data.
This type of bus can also access memory spaces that have different type on endian decoded inside. First let's check what are the 2 types of endian that we can find:

- BIG ENDIAN - The Most Significant Bit is stored in first in the memory

| **Memory Address** | **Data Saved** |
|---|---|
| 0 | A1 |
| 1 | B2 |
| 2 | C3 |
| 3 | D4 |

**Table 4.6:** Big Endian Example for a memory (1 byte per cell) for the 32 bit data: 0xA1B2C3D4

- LITTLE ENDIAN - The Least Significant Bit is stored in first in the memory

| **Memory Address** | **Burst Type** |
|---|---|
| 0 | D4 |
| 1 | C3 |
| 2 | B2 |
| 3 | A1 |

**Table 4.7:** Little Endian Example for a memory (1 byte per cell) for the 32 bit data: 0xA1B2C3D4

Since we need to know whether the memory is Big Endian of Little Endian for storing the data accordingly to its specification a method called Byte-invariant endianess is used. It has 2 main characteristics:

- The element uses the same continuous bytes of memory, regardless of the endianness of the data
- Any byte transfer to an address passes the 8 bits of data on the same data bus wires, to the same address location, regardless of the endianness of any larger data element that it is a constituent of.

In this way we make sure that whatever is the memory we are accessing we never have problems with overlap of informatin inside of it.

### Response Structure

The AXI protocol provides two responses one for reading and one for writing. The two signals of 2 bits that carry this information are:

- RRESP[1:0]
- BRESP[1:0]

The comination of 2 bits produces $2^2$ combinations described as below

| **xRESP** | **Response Type** |
|---|---|
| 00 | OKAY |
| 01 | EXOKAY |
| 10 | SLVERR |
| 11 | DECERR |

**Table 4.8:** AXI4 Response Types

- **OKAY**: this response isn't very common and indicates either the success of a normal access or the failure of an exclusive access.
- **EXOKAY**: indicates the success of an exclusive access
- **SLVERR**: indicates an unsuccessful transaction
- **DECERR**: indicates that the interconnect cannot successfully decode a Slave access.

### Memory Types

The **AxCACHE** contains the information about which kind of memory is in our device. Depending on the AXI version the encoding of the AxCACHE signal may differ.

The corresponant signals are on 4 bits. The "x" inside the binary code means that in that spot the equivalent bit can be either 1 or 0.

| ARCACHE | AWCACHE | Memory Type |
|---|---|---|
| 0000 | 0000 | Device Non-bufferable |
| 0001 | 0001 | Device Bufferable |
| 0010 | 0010 | Normal Non-cacheable Non-bufferable |
| 0011 | 0011 | Normal Non-cacheable Bufferable |
| 1010 | 0110 | Write-Through No-Allocate |
| x110 | 0110 | Write-Through Read-Allocate |
| 1010 | 1x10 | Write-Through Write-Allocate |
| 1110 | 1110 | Write-Through Read and Write-Allocate |
| 1011 | 0111 | Write-Back No-Allocate |
| x111 | 0111 | Write-Back Read-Allocate |
| 1011 | 1x11 | Write-Back Write-Allocate |
| 1111 | 1111 | Write-Back Read and Write-Allocate |

**Table 4.9:** AXI4 Memory Types given from AxCACHE

Each of this memory types is described by a different behaviour of the memory. A specific memory region defined by one of the types above cna be substitute by another one of different, incompatible type. In order to do that, all the peripherals must stop accessing the memory, only one Master access it and perform the required operations, at last the memory is restarted and all the peripherals starts accessing her once again with the new attribute (new value of AxCACHE).

### Access Permission

To protect against illegal transactions, the AXI protocol provides 2 signals on 3 bits:

- ARPROT[2:0]: access permission for read transactions
- AWPROT[2:0]: access permission for write transactions

The following table describes the possible level of privileges depending on the value of the AxPROT signal

| AxPROT | Value | Function |
|---|---|---|
| 0 | 0 | Unprivileged access |
| | 1 | Privileged access |
| 1 | 0 | Secure access |
| | 1 | Non-secure access |
| 2 | 0 | Data access |
| | 1 | Instruction access |

**Table 4.10:** AXI4 Access Privileges

### Identifier

The last type of signal of the AXI4 peripheral that we are going to see it's the ID type. This type of signal "xID" is used by the master to identify separate transactions and to understand in which order they need to be returned.
The advantage of having an identifier (in contrast to not having it as we'll see later in the AXI4-Lite part) is that the Master can issue multiple transactions without waiting for anyone to finish. This would be impossible without an ID since it would'nt know in which order send them back.
Of course this signal is completely optional since both the Slave and the Master can easily manage all the signal, issueing them one at the time.
As we've seen before the following table recaps the xID type of signals:

| xID | Channel |
|---|---|
| AWID | Write address |
| WID | Write data |
| BID | Write response |
| ARID | Read address |
| RID | Read data |

**Table 4.11:** AXI4 Transaction IDs

The Slave must ensure that the RID matches the ARID of the address that it is responding to.
The Master must issue all the WID in the same order it transmits the AWID in order to avoid confilicts.

### 4.3.4 AXI4-Lite

The AXI4-Lite is a more simple version of the AXI4 peripheral seen before. It has the following functionalities:

- Way less signals in the control sections
- All transactions burst are of lenght 1 (no signal needed to specify this)
- The bus can be either 32 or 64 bits and all the data access uses the full width of it.
- The are no exclusive accesses

Given these premises we can now check the difference between the AXI4 and AXI4-Lite peripherals. The signals that are still present on the AXI4-Lite peripherals are:

- ACLK
- ARESETn
- AWVALID, AWREADY, AWADDR, AWPROT
- WVALID, WREADY, WDATA, WSTRB
- BVALID, BREADY, BRESP
- ARVALID, ARREADY, ARADDR, ARPROT
- RVALID, RREADY, RDATA, RRESP

To have a more clear understanding of these signals check Table 4.3.
Of them, two are modified from their previous version: RRESP and BRESP since the EXOKAY response isn't supported, this is due to the fact that there are no exclusive access so it'd be completely pointless to have a signal that could point out a correct execution of that task.

#### AXI4-Lite Interface for Spiker

This type of peripheral is the one used in the project. The reasion why we've chosen this is because having a more complex AXI4 peripheral would have been way too much. The interface for the SNN only needs us to write a specific combination of bits on a register that will be later read by the SNN in order to perform its operations.

The AXI4-Lite interface on Vivado, provides 4 to 512 32-bits registers to use in order to store the data that are sent on the bus. For our project we only need to use 2 of them.
Since the register and the AXI4-Lite uses mostly a 32-bit interface (a 64-bit interface is used only in case of components having a 64-bit atomic accesses), we had to perform some operations. The signals that we need to pass to the SNN are the following one:

| Signal Name | N° Bits | Register Assigned |
|---|---|---|
| CLK | 1 | - |
| RDEN | 1 | SLV_REG1(30) |
| RST_N | 1 | SLV_REG1(29) |
| START | 1 | SLV_REG1(28) |
| LOAD | 4 | SLV_REG1(27 downto 24) |
| SEL | 10 | SLV_REG1(23 downto 14) |
| ADDR | 10 | SLV_REG1(13 downto 4) |
| DATA(36 downto 32) | 4 | SLV_REG1(3 downto 0) |
| DATA(31 downto 0) | 32 | SLV_REG0 |

**Table 4.12:** AXI4-Lite Spiker Interface Signals

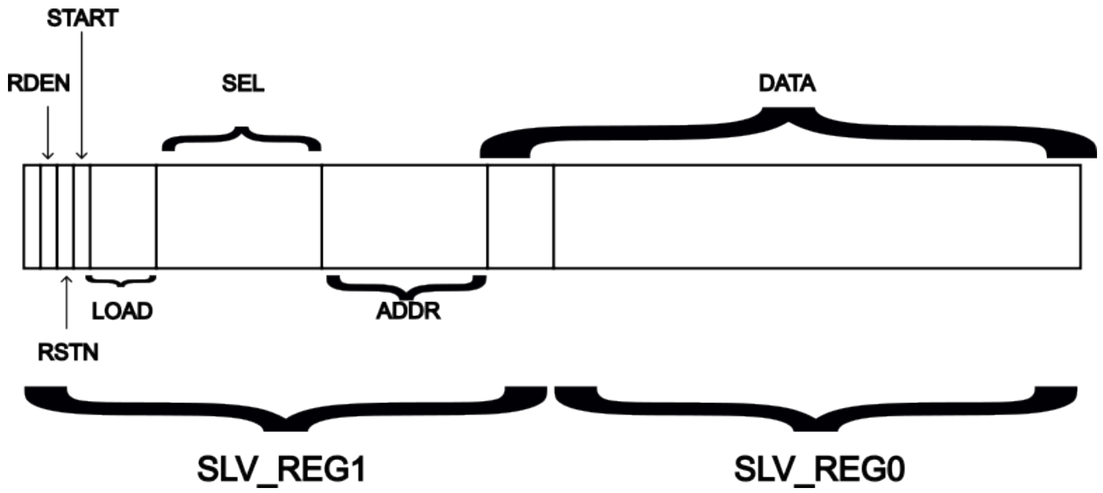


**Figure 4.9:** Distribution of signals in the AXI registers on Vivado

As we can see, the total number of bits required to our Spiking Neural Network is 63 so we had to find a way to use 64 bits synchronized while passing 32 bits at a time.
The way we decided to do so, is by passing the first 32 LSBs and then the next 32

(or 31) MSBs. We connected all the above signals to the correspondant registers SLV_REG0 and SLV_REG1 inserting the first 32 bits of the DATA into the SLV_REG0 and the subsequent 4 bits of the DATA plus the 27 bits for the control and address signals into the SLV_REG1.
After setting all the registers, a signal that synchronizes the transmission of those data to the SNN is needed, without it each time we either update the LSB or the MSB what happens is that the Network will be updated each time, instead of only once (when all the 64 bits are set). In order to do such a thing a third register has been used: SLV_REG3. When there is a rising edge on the first bit of this register all the 64 bits are updated, otherwise the signals maintain their previous values. A flowchart of this is shown below.

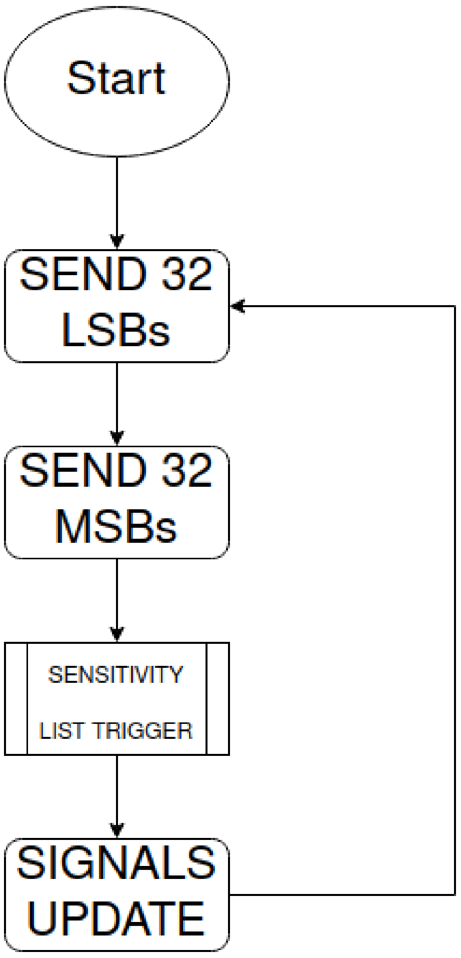


**Figure 4.10:** Flowchart of the AXI4-Lite Spiker Interface

Although the concept is very simple, this allowed us to use 64 bits each time we

want to communicate with the SNN.
From the list above, the two output signals (READY and CNT_OUT) are missing. This is because they are connected to an output register which is updated each time one of those two signals changes.

Another issue to face was the START signal. Since we are not directly giving the data to the SNN but we are calling a function to do so a couple of problems arised:

- We have to wait before the function finishes for having the data
- We have to wait until the next transaction writes into the register to change the value provided to the SNN

Both of those "problems" aren't critical for all the signals but the START, this is because the SNN expects the START to go to 1 only for one clock cycle but this can't happen. The execution of the function *Xil_Out32* that writes 32 bits onto a memory address isn't performed in 1 clock cycle. As it is known the process to decompile a c function is the following one:

- PREPROCESS: the source code (.c file) gets extended and passed to the compiler
- COMPILE: the compiler, converts this code into assembly code
- ASSEMBLER: the assembler converts the code into object code
- LINKER: the linker connects the liberary used by the C code to our object code.

This process creates from a simple line of code in C, multiple lines of code in ASSEMBLY which are the one that are executed 1 per clock cycle (if the are no stalls). It has been calculated that to execute this simple line of code

$$Xil\_Out32(Address, Data)$$

the processor needs around 200 clock cycles, which means that when we give the START signal the value '1' to switch on our SNN it will stay on for atleast 200 clock cycle until it is set to '0'. This created problems with the neural network.
To solve this issue a simple circuit has been described in VHDL, its behaviour is of sampling the current signal and doing a check to see if it is different from the signal of the previous clock cycle. The circuit is the following one

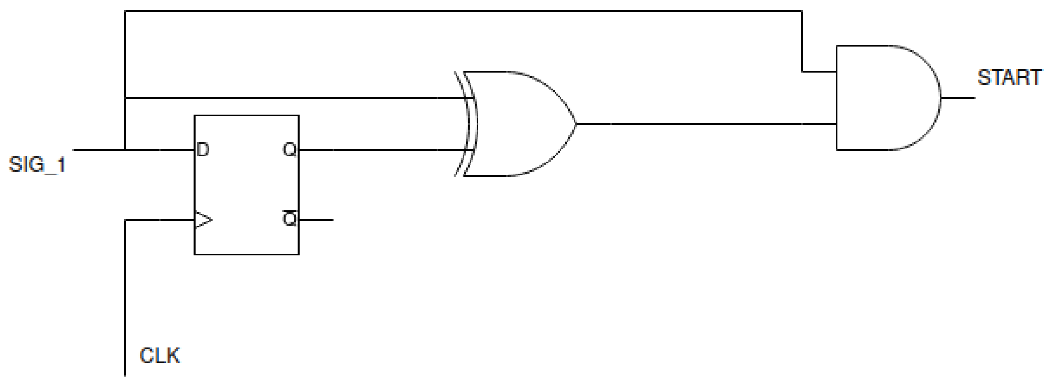


**Figure 4.11:** Circuit for the START signal

The signal SIG_1 is the one sent from the driver. The truth table is the following one

| Current Signal | Previous Signal | START |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Table 4.13:** START signal truth table

As we can see from the Table 4.13 the signal START goes to 1 if and only if the current signal is 1 and the previous signal is 0.

# Chapter 5

# Vivado's Synthesis & Drivers

In this chapter we are going to analyze what were the results given from the Vivado's synthesis ad what were the steps and result given from the implementation of the device on board.

## 5.1 Vivado's Features

Vivado is a software produced and distributed by Xilinx for the synthesis, analysis, test and implementation of HDL (Hardware Description Language) designs. It is written in C++ for Windows and Linux systems.
This software allows you to create a block design of your architecture and connect it to the processor and peripheral of the real board you are going to implement it on.

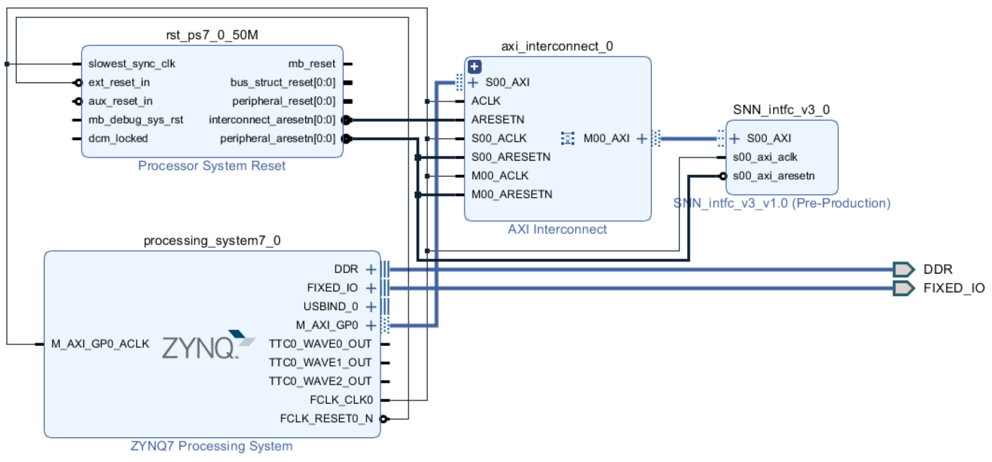


**Figure 5.1:** Vivado's Block Design of Spiker Interface

Vivado also supports the creation of custom peripherals that interface with the Advanced Microcontroller Bus Architecture bus (see chapter 4), this was used in our case to create the AXI peripheral that connects our custom SNN to the processor.
Once the peripheral is created another important feature of this software is that it automatically generates physical addresses for the peripheral just created in order to be able to access it via software (debug).
After the creation of the whole block design and the assignments of all the addresses for the peripherals, it is possible to perform the synthesis of the design and also the implementation. This steps are fundamental to understand whether is possible or not to implement the design on a board.
The final useful feature to us is the possibility to extract and export an HDF (Hardware Description File) and a Bitstream. These are used to perform the following tasks:

- Hardware Description File: Used to setup the configuration of the device inside the kernel (*device-tree*) of the operating system.

- Bitstream: Used to setup the Field Programmable Gate Array of the board with our custom block design.

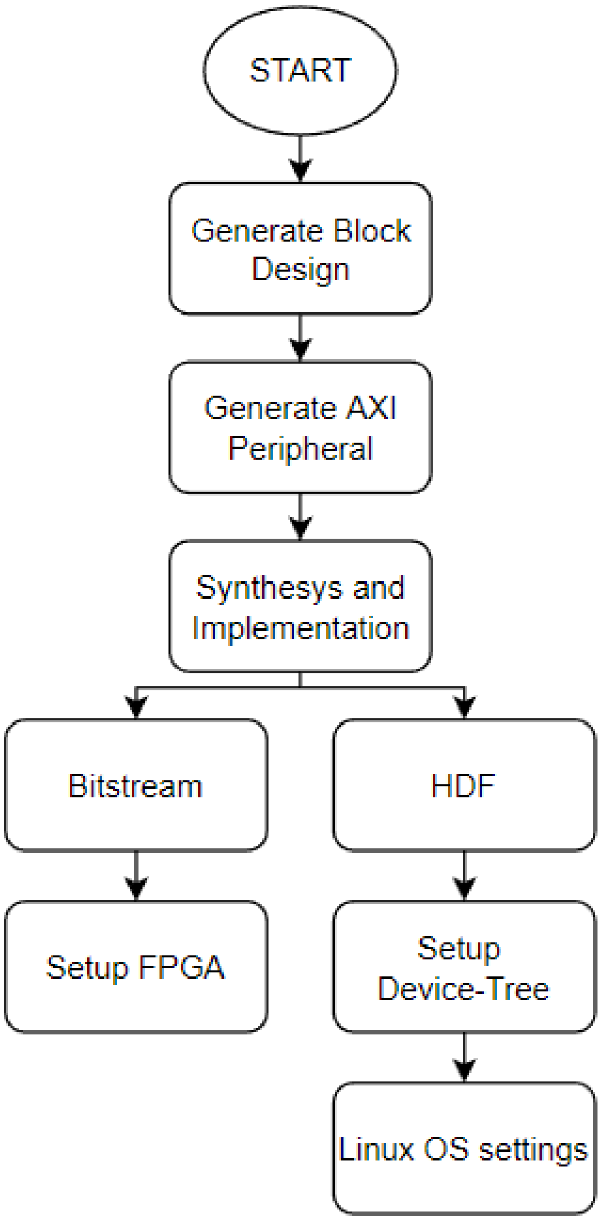


**Figure 5.2:** Vivado project flowchart

## 5.2 SNN Interface

As we already discussed in a previous section (section 4.3.4) the interface to connect Spiker to the processor of the Zc702 was done using an AXI peripheral. The whole interface project looks like this

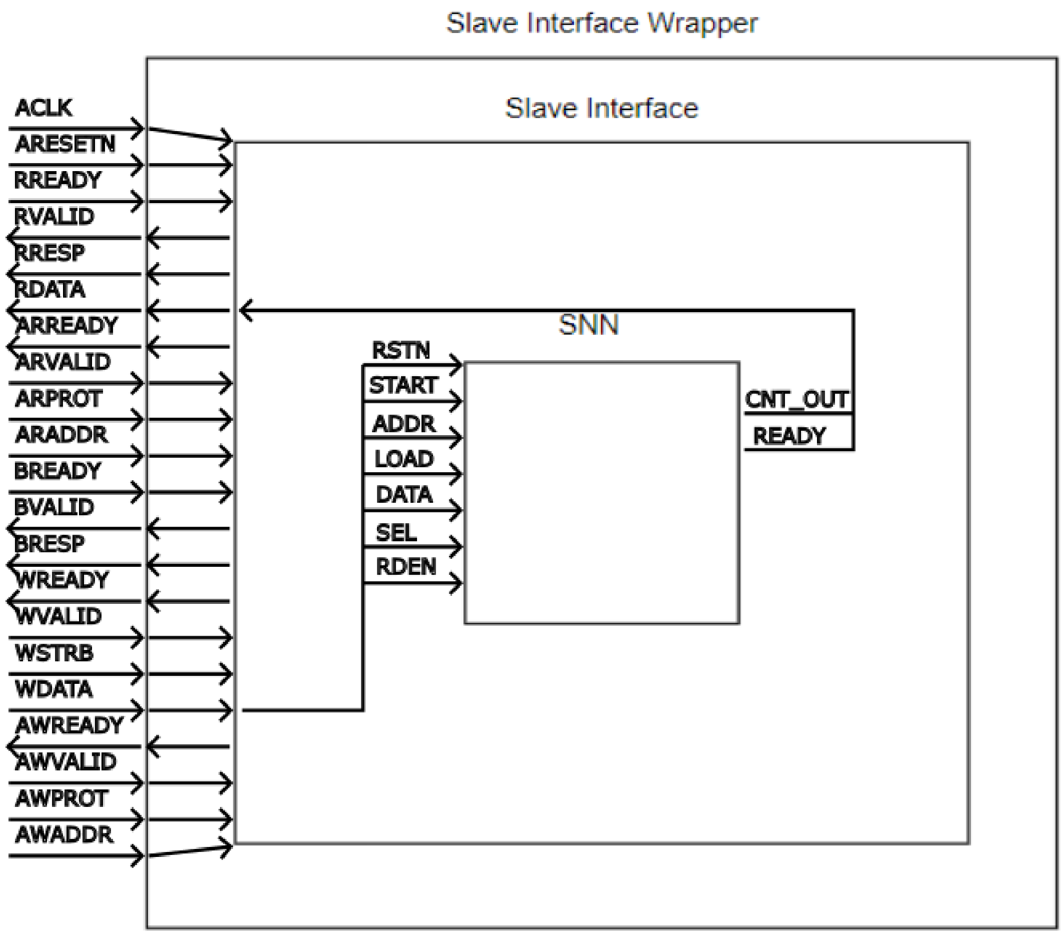


**Figure 5.3:** Internal scheme of the SNN Interface

In the Figure 5.3 we can see that the connection is performed inside a standard AXI4-Lite peripheral. As it is possible to see from Figure 5.1 the connection between the Interface and the rest of the block design is the following one:

- A Master to Slave cable used to connect the interface to the AXI_Interconnect block.

- The clock signal provided by the processor.

If we explode that single cable, as we can see from Figure 5.4 all the signals described in subsection 4.3.4 are seen

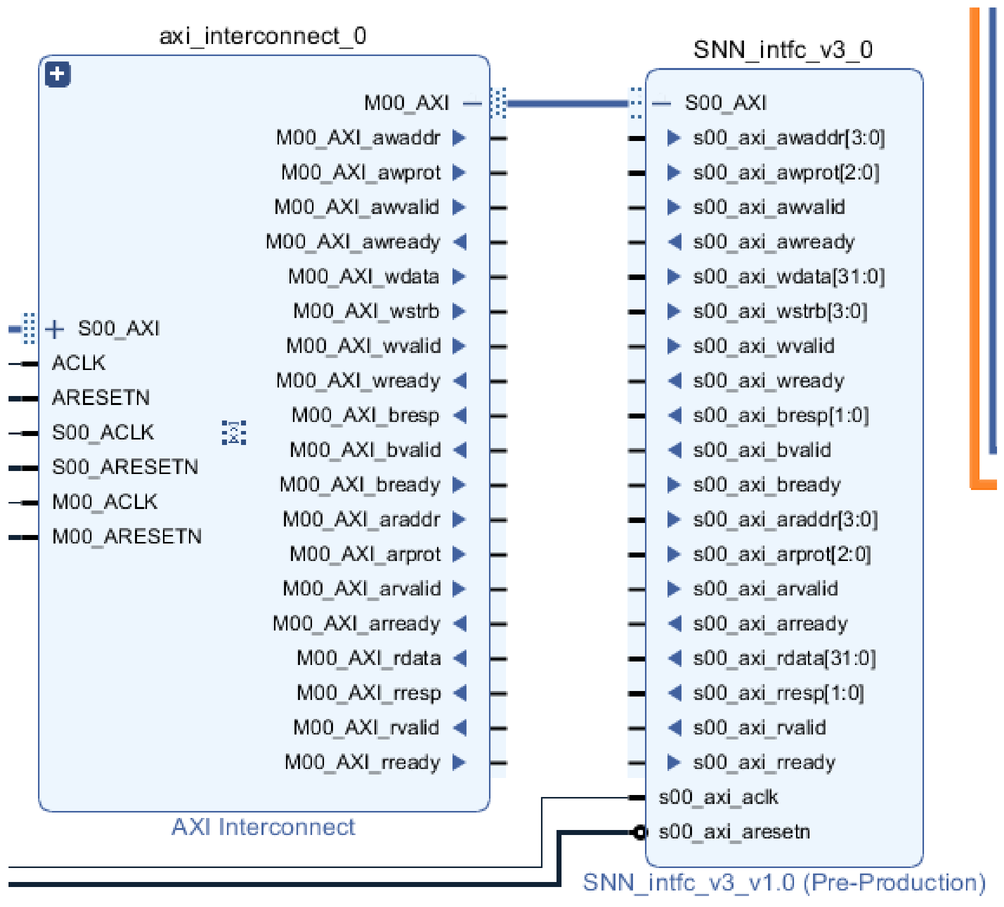


**Figure 5.4:** Zoom-in on the AXI signals of the peripheral

Most of these signals are piloted automatically from the processor, the only two that we can access from our driver are the WDATA and RDATA signals. In order to perform such a task one have to write or read on the Physical Address provided by the tool.
Of course this has to be done if the aim is to access it externally via software: via Testbench is possible to access and change all the signals to see the behaviour of the interface in all the use cases. It is important to add that performing read/write operation on a Physical Address directly from an application inside an operating system is impossible. This will be better explained later on subsection 5.3.1.

As it is possible to see from Figure 5.3 inside the block of the interface there are various other sub-blocks that compose it, the most notable are:

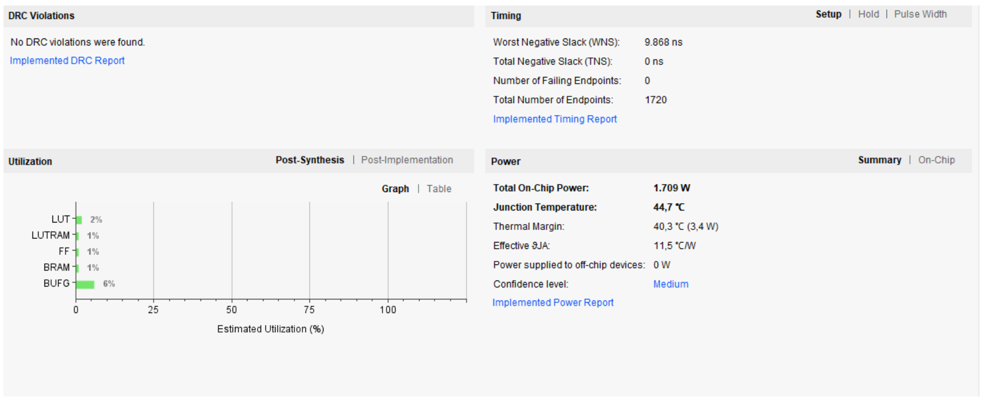
- The interface with the AXI slave peripheral
- The spiker accelerator

- The circuit for the start signal analyzed in Figure 4.11

All these blocks are then wrapped in a unique VHDL file used from the main project to synthesize and implement the custom circuit on the FPGA of the board.

## 5.3 Results and Software Implementation

The synthesis and implementation produced the following graphs of the resources allocated.


**DRC Violations**

No DRC violations were found.
Implemented DRC Report

**Timing** — **Setup** | Hold | Pulse Width

| | |
|---|---|
| Worst Negative Slack (WNS): | 9.868 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1720 |

Implemented Timing Report

**Utilization** — **Post-Synthesis** | Post-Implementation — **Graph** | Table

| | |
|---|---|
| LUT | 2% |
| LUTRAM | 1% |
| FF | 1% |
| BRAM | 1% |
| BUFG | 6% |

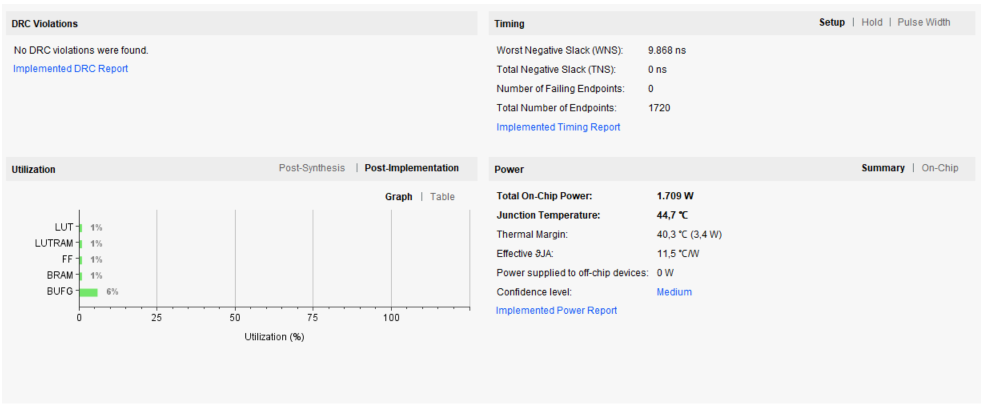
Estimated Utilization (%)

**Power** — **Summary** | On-Chip

| | |
|---|---|
| **Total On-Chip Power:** | **1.709 W** |
| **Junction Temperature:** | **44,7 °C** |
| Thermal Margin: | 40,3 °C (3,4 W) |
| Effective θJA: | 11,5 °C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Medium |

Implemented Power Report

**Figure 5.5:** Synthesis Result of the project


**DRC Violations**

No DRC violations were found.
Implemented DRC Report

**Timing** — **Setup** | Hold | Pulse Width

| | |
|---|---|
| Worst Negative Slack (WNS): | 9.868 ns |
| Total Negative Slack (TNS): | 0 ns |
| Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 1720 |

Implemented Timing Report

**Utilization** — Post-Synthesis | **Post-Implementation** — **Graph** | Table

| | |
|---|---|
| LUT | 1% |
| LUTRAM | 1% |
| FF | 1% |
| BRAM | 1% |
| BUFG | 6% |

Utilization (%)

**Power** — **Summary** | On-Chip

| | |
|---|---|
| **Total On-Chip Power:** | **1.709 W** |
| **Junction Temperature:** | **44,7 °C** |
| Thermal Margin: | 40,3 °C (3,4 W) |
| Effective θJA: | 11,5 °C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Medium |

Implemented Power Report

**Figure 5.6:** Implementation Result of the project

So is possible to see that the implementation on the board can be done without any issue.
The data on both Figure 5.5 and Figure 5.6 can be summarized and confronted.

| | Implementation | Synthesis |
|---|---|---|
| LUT | 2% | 1% |
| LUTRAM | 1% | 1% |
| FF | 1% | 1% |
| BRAM | 1% | 1% |
| BUFG (Global Buffers) | 6% | 6% |

**Table 5.1:** Post Implementation and Synthesis confront table

The difference between the two LUT depends on the fact that the Post-Synthesis elaboration is done with the estimated resources allocated in the board.
In order to boot Linux with a custom block on the FPGA of the board it is necessary to have 2 things:

- A *BOOT.bin* file that containt the Bitstream of the FPGA as well as the Fist Stage Boot Loader

- An *Image.ub* file that contains the Image of the kernel

Along with these two files a *ROOT* folder is also needed.
All of the files listed have then to be placed inside an SD-card inside specific partitions.

- FAT32: for the BOOT and the Image of the kernel

- ext4: for the ROOT directory

The board then has to be set according to Table 2.1 in order to set the BOOT option to SD-card. The connection via serial USB-to-UART (see subsection 2.3.1) has to be set to 115200 Baud rate.

### 5.3.1 Kernel Module

The Kernel module is an essential part to create a first step in the communication between the hardware and the user.
As it was stated before Vivado give us a first step in the software implementation by assigning a physical address to the peripheral. This physical address is inaccessible via C application or via Kernel module directly. In order to be able to access it it needs to be mapped from the Physical Memory where it is, to the Virtual Memory.

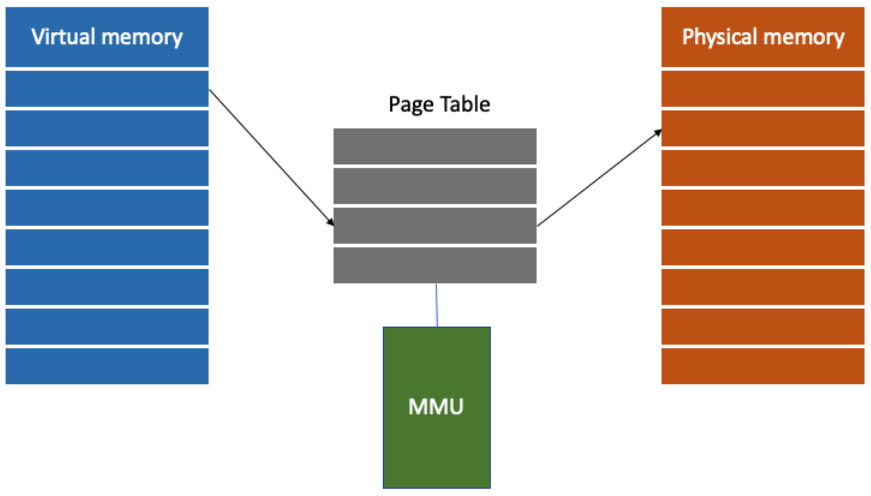


**Figure 5.7:** Physical Memory to Virtual Memory mapping

This is done by the Kernel module with the following function

$$ioremap\_nocache(unsigned\,long\,phys\_addr,\ unsigned\,long\,size)$$

where

- *phys_addr* is the physical address to map
- *size* is the byte of the memory region

The LKM (Linux Kernel Module), besides the mapping of the physical memory into the virtual one, also allocates the memory for the character device that will serve as a "bridge" between the C application and the module. The connection between those components is shown in Figure 5.8. Both the C Application and the LKM can access the Device in read/write. Whatever is written on the device is written in the form of

$$char\_*$$

This means that it will have to be converted both ways to be used by the C Application and also to be sent to the Spiking Neural Network. This device needs 4 function to be declared:

- Open: called when the device is opened
- Release: called when the device is closed

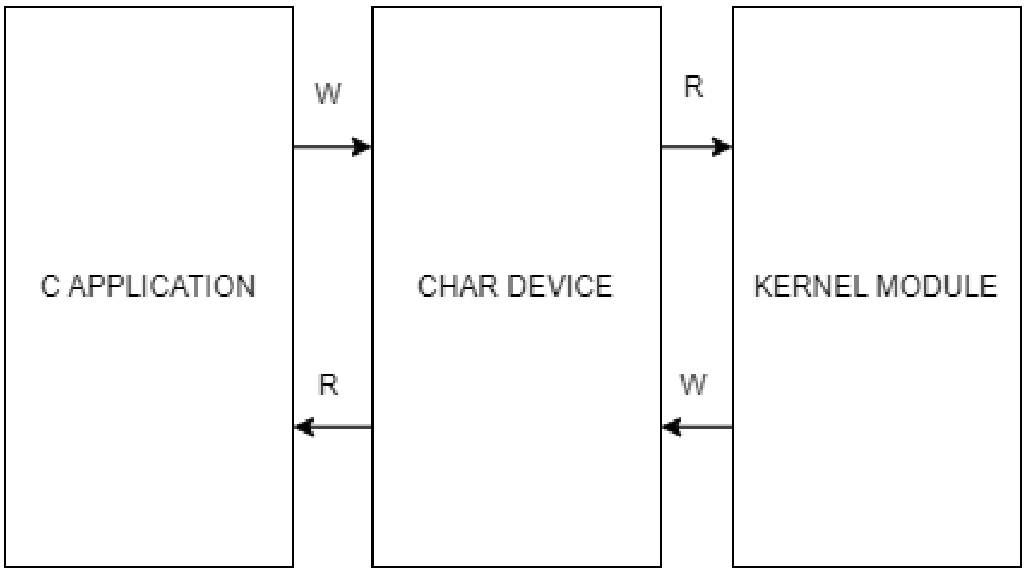


**Figure 5.8:** Interconnection LKM - Character Device - C Application

- Read: used to read from the device

- Write: used to write on the device

As it is shown in Figure 5.8, the LKM writes on the device the output of the SNN while the C Application write on the device the data to send to the SNN.

### 5.3.2 C Application

The C application is used to allow the user to open the files and send the string of bytes to the device.
The following flowchart describes how this application works

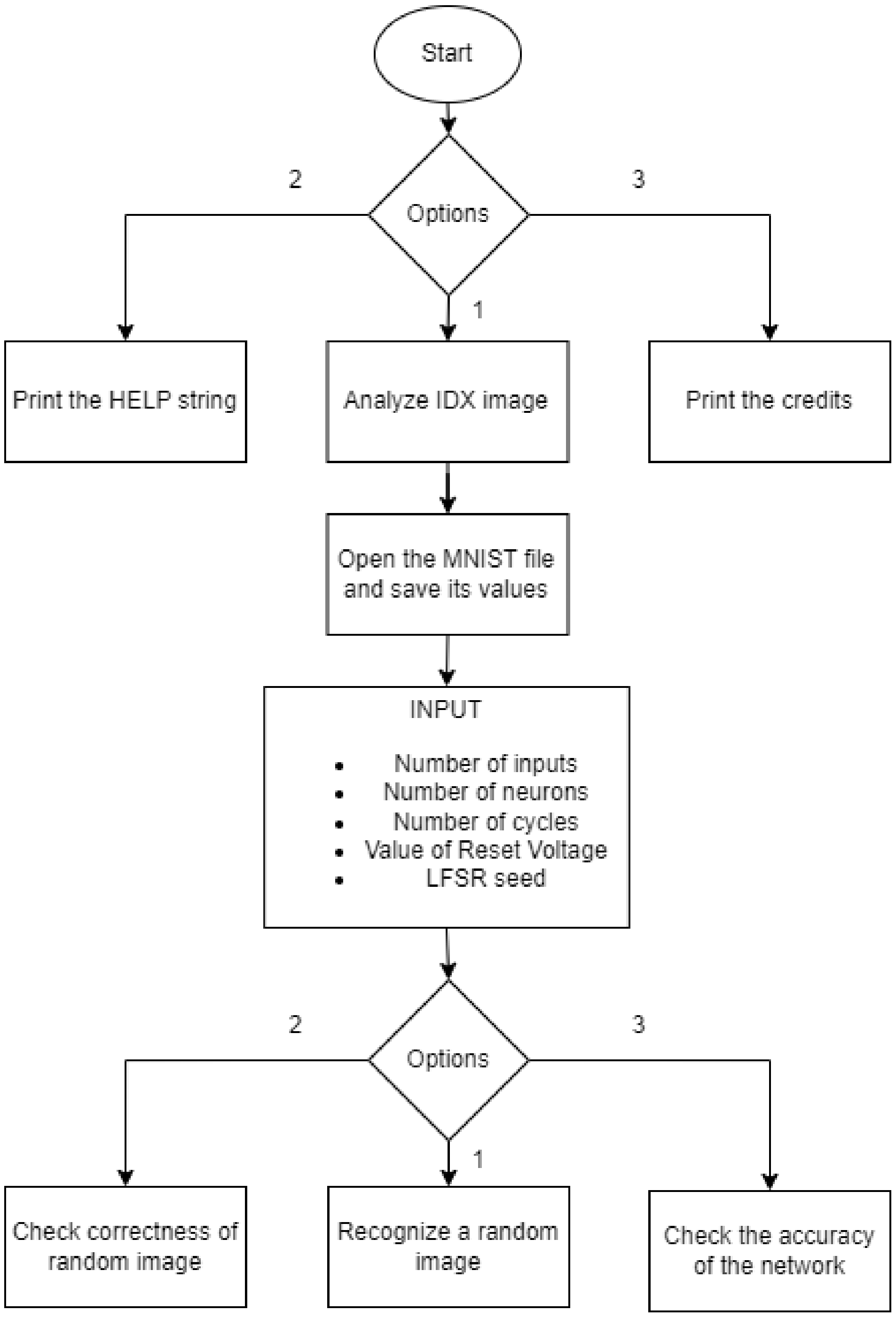


**Figure 5.9:** Flowchart of the C Application

This application takes in input 5 parameters

- 1) IDX Train Set: file that contains the image set
- 2) Labels: file that contains the set of labels of the images
- 3) Inhibitor Weights: file that contains the list of the inhibitor weights for each BRAM of each Neuron
- 4) Voltage Thresholds: file that contains the voltage thresholds of each input
- 5) Neuron Labels: file that contains the list of labels for each neuron, used to see what number they are trying to recognize

Once the program is launched the user will be able to choose between 3 options:

- 1) Check the IDX file: if he wants to use the Spiker interface
- 2) Help: if he wants a quick instruction to what this application does
- 3) Credits: if he want to check who developed each part of the hardware, driver and software.

If the *Option 1* is selected the Network is loaded and the user will be asked to insert the following parameters in it:

- 1) Number of Inputs
- 2) Number of Neurons
- 3) Number of Cycles
- 4) Value of Voltage Reset
- 5) Seed of the LFSR

After that the whole network is ready to be used and the user will be asked to choose between these 3 options:

- 1) Recognize a random image: the software will generate a random number from 1 to the maximum number of images and will pass it to the SNN to let it try to recognize it
- 2) Check if a random image is correct: this option is the same as *Option 1* but it also open the label file to see if the output produced by the SNN is the correct one
- 3) Check the accuracy on all images: this repeats *Option 2* iteratively on all images and will produce a percentage as output to see how many images it correctly recognized.

# References

[1] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *MNIST Database of Handwritten Digits* (cit. on pp. 2, 12).

[2] Xilinx. *UG850 ZC702 Evaluation Boards for the Zynq-7000 XC7Z020 SoC*. Mar. 2019 (cit. on pp. 5, 6).

[3] Zied Marrakchi & Habib Mehrez Umer Farooq. *Tree-based Heterogeneous FPGA Architectures*. May 2012, pp. 7–8 (cit. on p. 6).

[4] Hunjun Lee Chanmyeong Kim Seungho Lee Eunjin Baek Jangwoo Kim. «An accurate and fair evaluation methodology for SNN-based inferencing with full-stack hardware design space explorations». In: (Sept. 2021) (cit. on p. 10).

[5] Wolfgang Maass. «Networks of Spiking Neurons: The Third Generation of Neural Network Models». In: (Nov. 1996) (cit. on p. 10).

[6] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. «Deep Leaning in Spike Neural Networks». In: (Jan. 2019) (cit. on p. 11).

[7] Hyeryung Jang, Nicolas Skatchkovsky, and Osvaldo Simeone. «Spiking Neural Networks-Part I: Detecting Spatial Patterns». In: (June 2021) (cit. on p. 11).

[8] Pierre Falez, Ioan Marius Bilasco Pierre Tirilly, Philippe Devienne1, and Pierre Boulet. «Multi-layered Spiking Neural Network with Target Timestamp Threshold Adaptation and STDP». In: (July 2019) (cit. on p. 13).

[9] ARM. *AMBA AXI and ACE Protocol Specification*. Oct. 2011 (cit. on pp. 21, 23).