Universidad del País Vasco / Euskal Herriko Unibertsitatea

ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Bachelor Final Thesis
Degree in Electronic Engineering

# Design of a configurable neural network architecture for efficient FPGA implementation using VHDL

Author:
Iñigo Poncela Vicente

Supervisor:
Inés del Campo Hagelstrom

Leioa, 18th June 2021

# Contents

# Chapter 1

# Introduction and objectives

The digital electronics field is one of the most fast changing sectors of research and industry. Since the invention of the transistor back in 1947 the electronic circuit's scale of integration has grown swiftly. Indeed, in 1965 Gordon Moore set the globally known Moore's Law [1], that has ruled the plans of this field for years.

One of the greatest advances in digital electronics was the invention of the programmable devices, which allowed to reuse an electronic circuit without wasting resources. This opened a door that was quickly explored and exploited. Among all the inventions and discoveries that sprouted in those early years, field programmable gate arrays (FPGAs) appeared, arriving to the industry in the late 90s after a fast growth [2]. Nowadays, they are still the preferred option for many digital implementation applications, due to their flexibility, high parallelism and good cost-performance relation [3].

On a completely different evolution process, although almost parallel in time, neural networks (NN) made their appearance thanks to an extensive experimentation during the 20th century [4]. From the original Perceptron network in the late 50s to the all around use of today, NNs have evolved to become a cutting-edge research topic. For example, it is being used in the autonomous driving tests that many research groups and companies are carrying out [5], [6], as well as, in finance [7] and medical applications [8], among others.

At the same time that FPGAs and NNs grew, a hardware description language (HDL), called VHDL (Very high speed HDL), was created to aid in the development of the increasingly common integrated circuits (IC). In the present, after various updates and reworks, VDHL is a language that can describe the performance and behavioral of digital circuits. It is widely used in the configuration of several programmable devices such as FPGAs and CPLDs (Complex Programmable Logic Device) [9].

Nevertheless, VHDL is not the only hardware description language, Verilog is another example. Indeed, both of them are IEEE (Institute of Electrical and Electronics Engineers) standards, which means that they can describe any electronic circuit without incompatibilities between companies or countries. This description includes the possibility of simulating the working procedures of the circuit and testing its performance, becoming great design tools.

## Objectives

The first objective of this work is to delve deeper into the use of FPGAs, taking into account what was learned in the Digital electronics and Digital systems design subjects. Starting with its theoretical background and finishing with its configuration in a physical board. In addition, advanced digital techniques will be applied with the aim of simplifying the hardware implementation, for example, using powers of two as numbers that turn costly term products into simple bit shifts.

Related with the previous objective, learning how to design digital systems with an optimal speed/cost tradeoff depending on the target application is also pursued. This can be done by selecting a low latency architecture with a high use of resources (in a parallel structure) or a higher latency one with a lower cost (a serial design).

A second objective is to serve as an introduction to the machine learning (ML) techniques, focusing more in depth in the the neural network algorithms. Specifically, a type of NN is of particular interest, extreme learning machines (ELM). Thus, their distinctive characteristics are to be studied and its difference with the rest of NNs understood.

Another of the main objectives is to learn from the start how a hardware implementation application is carried out. The complete process of planning the project, coding the modules in VHDL, simulating each of them, and in the end, implementing the design in a FPGA is of great interest.

Finally, this work also aims to perform an exercise that gathers all the previous steps, in this case, the classification of different kinds of soils in a multispectral image obtained by a Landsat satellite. The reasons for choosing it are related to the common use of FPGAs in satellite installations, due to the high quantity of data they can handle in parallel processing steps. Additionally, the device configuring bitstreams can be easily sent to the satellite.

The following chapters are organised as follows: Chapter 2 will give the theoretical basis of ML algorithms, neural networks, and in particular, ELMs. Moreover, it will also include an introduction to FPGAs, with the main resources and characteristics of the Artix 7 family, and an explanation of the common design flow in a FPGA configuration project. Chapter 3 will present an explanation of the basic blocks that form a common NN, as well as a comparison between two of the main methods to implement an ELM. Furthermore, Chapter 4 includes some simplifying measures for the already showed basic blocks, together with various designs that include these measures. To conclude, Chapter 5 and Chapter 6 contain a final classification application with a new database and the conclusions of this work respectively.

# Chapter 2

# Theoretical basis

In this chapter, the theoretical background of this work will be presented in order to understand the followed process. First of all, machine learning techniques will be briefly described together with a more detailed explanation of a particular type of structure known as extreme learning machines (ELM). Secondly, field programmable gate arrays (FPGAs) will be addressed since, in the end, the designed architecture will be implemented in one of them. Finally, a summary of the main characteristics of the FPGA family that will be used will be given, together with an outline of a common design flow.

## 2.1 Machine learning

Machine learning (ML) is a field of computational science that develops dynamic algorithms capable of making decisions based upon the previously received data, this contrasts with classical programming instructions that rely on static algorithms. One of the objectives of ML is enabling computer programs to learn and improve their performance at some tasks through experience [10].

ML is already widely spread, and it has been used in several fields such as, finance [7], medical applications [8], pattern recognition [11] and computer vision [12], among others. It usually works with big databases, in order to have the best learning process and achieve the highest generalization as possible. Generalization is a concept in ML that states how well a model performs on unseen data [13].

The adaptation of the computational algorithm is commonly called training, during this process it will be fed with input data, changing according to it. Depending on the type of algorithm the training process will be different: some weights can be adjusted, the internal network of computational pathways can be rearranged or the probability distributions that predict the outputs can change [14].

Depending on the input data, ML can be divided into three categories: supervised, unsupervised and semi-supervised learning. In the former, the desired output comes along with the data and is correctly labelled, it is commonly used in classification and regression problems. The objective is to estimate an output from new data based on previously given samples, where the output was known. During the training process the algorithm will configure itself according to the known inputs [14].

On the other hand, in unsupervised learning, the input data does not have any labelled output targets, so the training will be done after knowing the difference between the given and the desired output. It is used in clustering and estimation of probability density functions. Finally, semi-supervised learning is a combination of both methods, with only part of the data labelled that will be used to infer the remaining portion. Its main use are the text and image retrieval systems [14].

## 2.1.1 Neural networks

A typical NN is an effective distribution of different types of neurons and its interconnections, as shown in Figure 2.1. These neurons are organized in a parallel structure called layer, that can fulfil different goals depending on their position in the network. There are three types of layers: input, hidden and output ones.

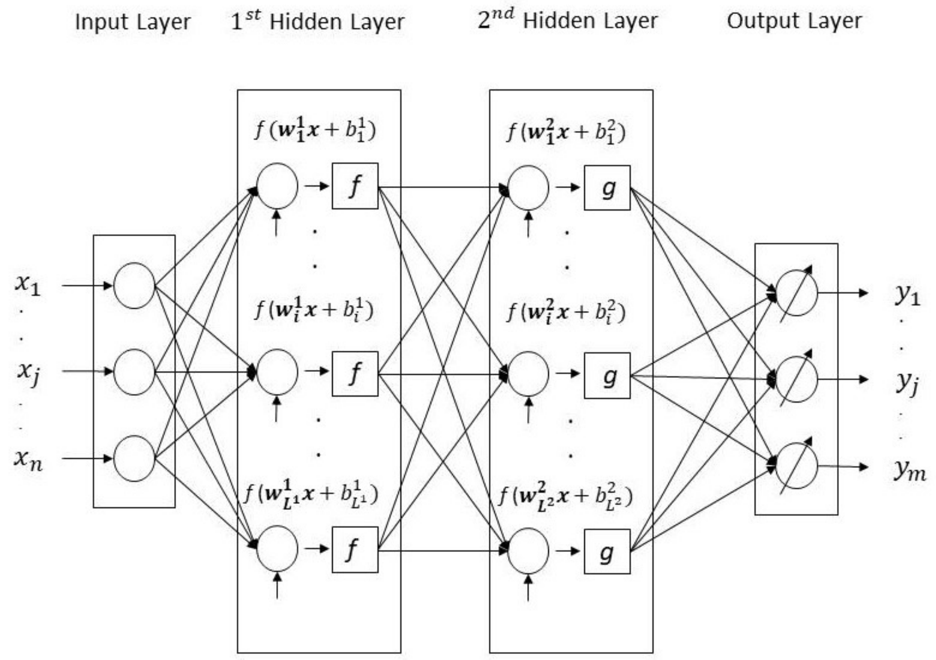
Input Layer
$1^{st}$ Hidden Layer
$2^{nd}$ Hidden Layer
Output Layer
$f(\boldsymbol{w}_1^1 x + b_1^1)$
$f(\boldsymbol{w}_i^1 x + b_i^1)$
$f(\boldsymbol{w}_{L^1}^1 x + b_{L^1}^1)$
$f(\boldsymbol{w}_1^2 x + b_1^2)$
$f(\boldsymbol{w}_i^2 x + b_i^2)$
$f(\boldsymbol{w}_{L^2}^2 x + b_{L^2}^2)$
$f$
$g$
$x_1$
$x_j$
$x_n$
$y_1$
$y_j$
$y_m$

Figure 2.1: Example of a multilayer neural network topology.

One of the parameters that will vary depending on the type of layer is the number of neurons in each one. While the input layer has as many as input attributes, the output layer is defined by the number of outputs of the network. On the other hand, for the hidden one it is a decision taken by the designer.

Furthermore, those neurons are different depending also on the type of layer. The most complex case is the hidden neuron, that counts with a multiplier, an adder and a transfer function. Its input attributes, that come from the neurons in the input layer, are organized into a vector $\mathbf{x} = (x_1, x_2, ..., x_n)$ and multiplied by another vector of the same dimension with the weights $\mathbf{w}$, and then added to a bias $b$. Both parameters are unique for each neuron, and thus, the result of one is different from the rest. This process is shown in Figure 2.1. Both the weights and bias are tuned during the training process.

The last parameter of the neuron is the transfer or activation function *f*, that unlike the previous variables initially set at random, is chosen by the designer according to the desired output. Some of the most common functions are: the linear, the hard limit and the log-sigmoid functions [4]. Its input is given by the already mentioned sum of products seen in Equation 2.1.

$$h_i = f(\mathbf{w}_i \mathbf{x} + b_i) \tag{2.1}$$

Where $h_i$ is the scalar output of neuron $i$. It is important to note that while the internal parameters, $\mathbf{w}_i$ and $b_i$, of a neuron in a layer are particular to it, the transfer function is the same for all of them.

The other two types of neurons have a more straightforward functioning. While the input ones receive the input attributes to distribute them to all the hidden neurons, the output neurons usually have a proper transfer function depending on the application, commonly a linear function [4].

Putting it all together, a multilayer distribution with a finite number of layers and a variable quantity of neurons in each one can be built, as showed in Figure 2.1. This type of topology is specially interesting because it allows to combine different transfer functions, one per hidden layer, enabling the resolution of a great number of problems.

Another decision the designer must make is the selection of the NN training method. This training process tunes the value of the internal parameters of the network, that were initialised randomly, comparing the outputs computed by the network with the targets provided in the training data. In addition, during this process the NN also optimises its internal processes to minimise the analysis and processing time [4].

The main training methods are derived from the backpropagation algorithm, that is based on the generalisation of the Widrow-Hoff learning rule. It starts with random weights, and the goal is to adjust them to reduce this error until it reaches a certain value [15]. The most common methods are: the Levenberg-Marquardt algorithm, the scaled conjugate gradient and the gradient descent. In the three of them the gradient is computed by the backpropagation algorithm.

### 2.1.2 Extreme Learning Machine

Extreme learning machine (ELM) is a particular type of NN, with a high-speed machine learning method based on a simple tuning-free algorithm. In other words, it depends less on the decisions taken by the designer compared to more common ML techniques such as, the explained backpropagation neural networks (NN) or support vector machines (SVM) [16]. ELM also does not present local minima or overfitting during the training period, which avoids common training problems.

An ELM consists of a feed-forward neural network with just a hidden layer. This architecture is specially interesting since the weights and biases of the hidden layer are not only generated randomly, as it is common in other NNs, but they remain unchanged, preventing them from being adjusted with every iteration. Thus, the learning time of an ELM is shorter than more common training algorithms, which opens a door for multiple real time response applications.

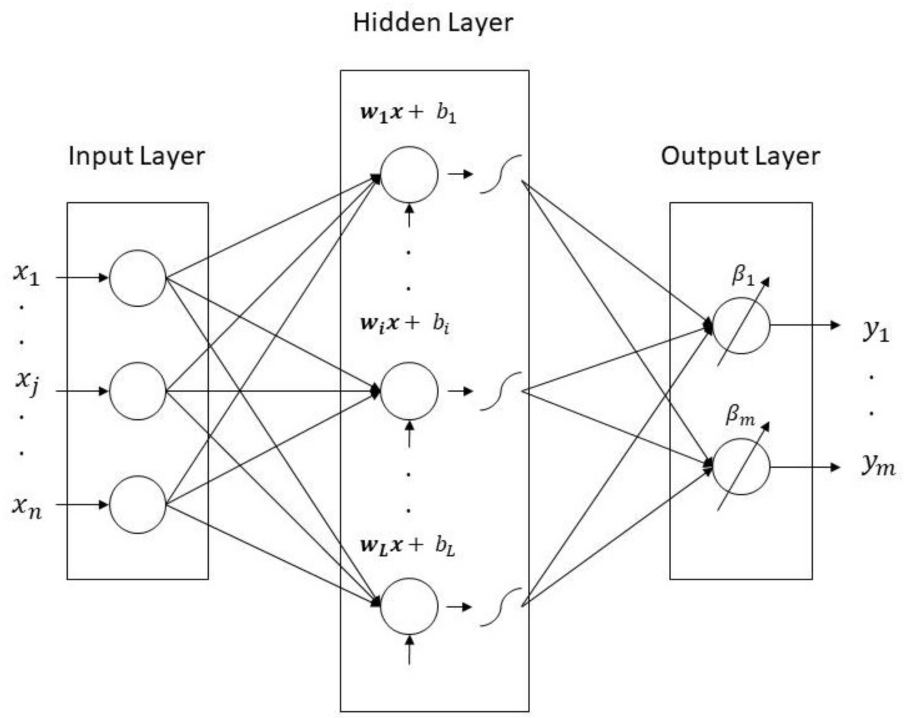

Figure 2.2: Outline of an ELM architecture.

In Figure 2.2 an outline of the ELM architecture with $n$ inputs, $L$ neurons in the hidden layer and $m$ outputs can be seen. The output vector $\boldsymbol{y}$ will be computed as follows:

$$\mathbf{y}(\mathbf{x}) = \sum_{i=1}^{L} h_i(x)\beta_i = \mathbf{h}(\mathbf{x})\boldsymbol{\beta} \tag{2.2}$$

Where $\boldsymbol{\beta}$ is the vector of beta weights that multiplies the output of each neuron of the hidden layer, indicated by the vector $\mathbf{h(x)}$. This output is expressed by Equation 2.1, where $f$ is the activation function, $\mathbf{w}_i$ the random weights and $b_i$ the bias of the $i$th neuron. Precisely, these last two parameters will be the ones set at random during the initialisation and that will remain untouched. Furthermore, the activation function will be chosen by the designer, the most common ones are the sigmoid function, the hardlimit and the tangential function.

The learning process in an ELM algorithm is based on computing the output weights $\boldsymbol{\beta}$ of Equation 2.2. Everything starts by taking a set of $K$ training samples, each one with $n$ input attributes, and its target vector $\boldsymbol{t}$, of dimension $m$. The relation between them is expressed by Equation 2.3:

$$\mathbf{T} = \mathbf{H}(\mathbf{x})\mathbf{B} \tag{2.3}$$

Where $\mathbf{H}(\mathbf{x})$ is the output matrix of the hidden layer. Those matrices can be expressed as follows:

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1 \\ \vdots \\ \mathbf{t}_K \end{bmatrix}_{K \times m} \qquad \mathbf{B} = \begin{bmatrix} \boldsymbol{\beta}_1 & \cdots & \boldsymbol{\beta}_m \end{bmatrix}_{L \times m} \tag{2.4}$$

$$\mathbf{H}(\mathbf{x}) = \begin{bmatrix} \mathbf{h}(\mathbf{x}_1) \\ \vdots \\ \mathbf{h}(\mathbf{x}_K) \end{bmatrix} = \begin{bmatrix} h_1(\mathbf{x}_1) & \cdots & h_L(\mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ h_1(\mathbf{x}_K) & \cdots & h_L(\mathbf{x}_K) \end{bmatrix}_{K \times L} \tag{2.5}$$

Solving Equation 2.3, that shows a system of linear equations, the output weights $\boldsymbol{\beta}$ can be calculated:

$$\boldsymbol{\beta} = \mathbf{H}^{\dagger}\mathbf{T} \tag{2.6}$$

Being $\mathbf{H}^{\dagger}$ the Moore-Penrose generalized inverse of matrix $\mathbf{H}$ [16], [17]. When designing the complete classifying algorithm there will be one output neuron, with its own $\boldsymbol{\beta}$ weights, for each classification class. In this explanation, the dimension of the target vector $m$ is the one stating that number of classes, and thus, the dimensions of the $\boldsymbol{\beta}$ matrix in the previous equation.

In addition, the $n$ input attributes of each sample will determine the number of neurons of the input layer. On the other hand, the number of hidden neurons is not fixed by any external parameter but by the designer, that can choose depending on the target application.

Finally, once the training stage is finished and all the parameters of the network are set, Equation 2.3 can be used to calculate the outputs of the test database, that will generate a new $\mathbf{H}$ matrix ($\mathbf{H}_{test}$).

This fast training process, without recursive operations, enables the possibility of using ELMs in high speed implementation applications. Indeed, the great parallelism inherent to this architecture, that is shown in Figure 2.2, encourages the use of hardware devices with a high parallelization capability. For example, field programmable gate arrays (FPGAs), that present this characteristic and are commonly used in these cases.

## 2.2 Field Programmable Gate Arrays

A field programmable gate array (FPGA) is a semiconductor device constituted of several logic cells, generally called configurable logic blocks (CLB), that are interconnected by a matrix of horizontal and vertical communication channels. By arranging these interconnecting wires (with programmable elements) and configuring the CLBs, different complex designs can be implemented. FPGAs can be reprogrammed to adjust to the desired application, which entails its main characteristic [18]–[20].

FPGAs are the next step from CPLDs (Complex Programmable Logic Device), since its logic blocks offer a better functionality than the set of term products and macrocells of the CPLDs. These CLBs usually consist of multiplexers (MUX), look-up tables (LUT) and flip-flops (FF), although each manufacturer does its logic blocks with certain differences between them. In Figure 2.3 an outline of a common architecture of a FPGA can be seen, with the interconnecting channels linking the I/O (Input/Output) blocks and the CLBs.

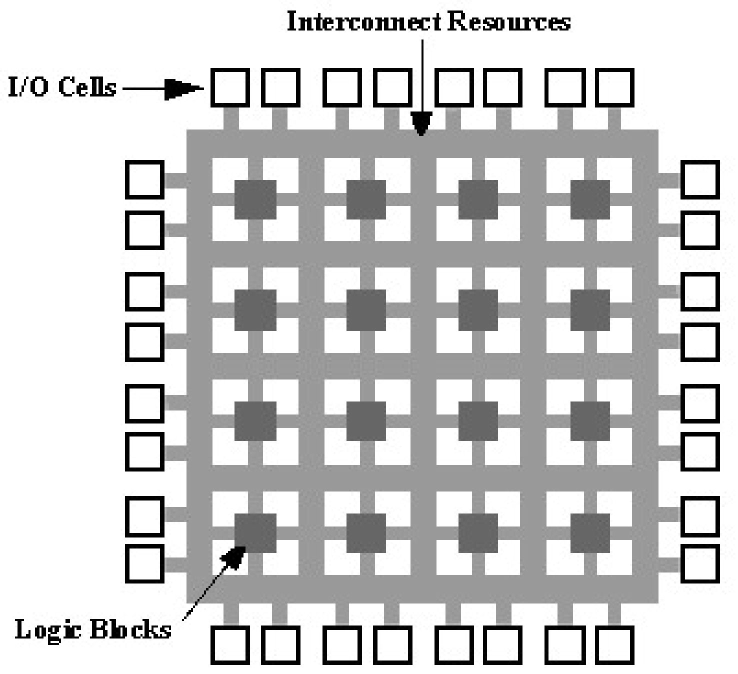

Figure 2.3: Scheme of a common architecture of a FPGA (Image from Wikimedia Commons under the Creative Commons Attribution 2.5 Generic License).

Moreover, FPGAs also offer several benefits that other devices such as ASICs (Application Specific Integrated Circuit), more focused on overall performance for niche applications, cannot match. First of all, FPGAs offer the possibility of implementing a higher parallelism in the designs, enabling a faster computational speed. Actually, they have certain embedded resources, DSP (Digital Signal Processing) slices and RAM (Random Access Memory) memories among others, that allow FPGAs to be used as algorithm accelerators, thanks to the level of parallelism they can reach [19].

Secondly, the flexibility that FPGAs offer, enabling its reprogramming if the application changes, makes them more appealing for certain cases than more specific devices such as the mentioned ASICs. This characteristic also has an economic impact, since

FPGAs allow the user to perform processes that in other cases will be done by designing companies, which makes them more profitable [18].

Furthermore, due to the possibility of skipping manufacturing steps and their lower cost, FPGA technology provides a faster path for prototyping and testing designs. Finally, they can be remotely configured, which makes them attractive for installations in hardly accessible platforms such as, satellites [21].

### 2.2.1 Architecture and resources of 7-Series FPGAs

One of the main manufacturers of FPGAs in the world is Xilinx [22], who has many device families each one with different characteristics: amount of resources, processing speed or number of I/O pins. In this work, the different designs will be implemented on an Artix-7 Xilinx FPGA. This device belongs to the 7-Series of the company. Therefore, a general overview of the architecture of this family will be carried out.

Firstly, it has to be noted that all the FPGAs have the resources already mentioned in the previous section. Secondly, the 7-Series [23] has many families (Artix, Kintex, Virtex and Zynq) where the Artix is the one with the lowest power and cost. Anyway, all the families have the same architecture showed in Figure 2.4, where it can be seen the distribution of the different elements [24].

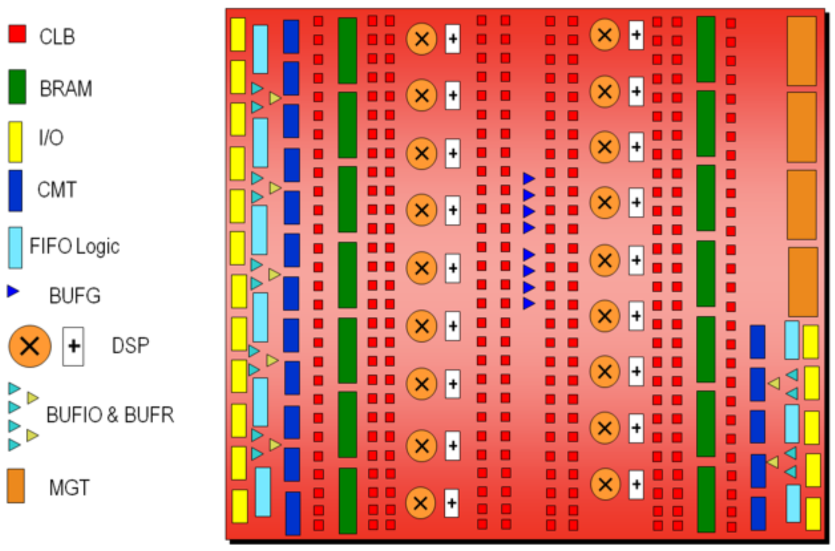

Figure 2.4: 7-Series architecture overview (courtesy of Xilinx).

To begin with, the CLBs count with two slices where the MUXs, FFs and LUTs are organised so that their use is maximised. In addition, there are two types of slices, with the difference that while the LUTs of one of them can only be used as logic, in the other they can also act as memory.

Continuing with the I/O blocks, it is very interesting to note that the 7-Series FPGAs can work at a single data rate (SDR) or a double data rate (DDR). Which means that the data can be transferred with each rising or falling edge of the clock (SDR) or with both of them (DDR). Moreover, these families have various power reduction features as well as a digitally controlled impedance [24].

Furthermore, the FIFO logic blocks link these I/O modules with the clock management tiles (CMT) and the memory block RAMs (BRAM), all of them can be seen in the architecture overview. Indeed, all the memory blocks will have a synchronous operation [24].

Finally, the DSP blocks, that are able of performing sums of products such as Equation 2.1 at a very high speed, are embedded in the architecture. Which allows the different CLBs to use them rapidly. In fact, these blocks are of great importance when a digital circuit that involves the already mentioned sums of products is implemented, since there is no other resource in the FPGA capable of replicating the calculations at a similar speed. For example, they are widely used in the neuron modules of a NN.

### 2.2.2 Nexys A7 board

The old Nexys4 DDR [25], now renamed as Nexys A7, is a prototyping board that features an Artix 7 FPGA. In particular, the XC7A100T-1CSG324C part number. It is optimised for a high performance logic, and has more capacity and resources than earlier designs. Indeed, it is the second highest capacity device of the Artix 7 family, with large external memories and several I/O ports (USB and Ethernet among others) to increase it if desired. The main features of the device are:

- 15,850 logic slices with 63,400 LUTs (4 each) and 126,800 flip flops (8 each) in total.
- 240 DSP slices.
- 4,860 Kbits of fast block RAM.
- Six clock management tiles (CMT)
- Internal clocks with speeds exceeding 450 MHz.
- On-chip analog-to-digital converter (XADC).

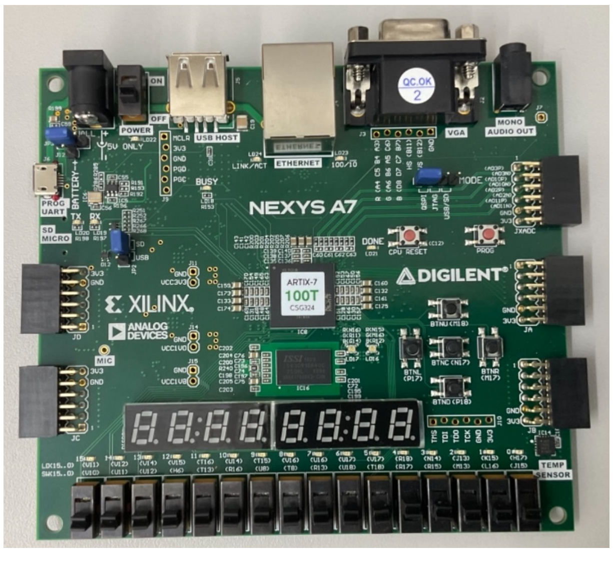

Figure 2.5: Image of the Nexys A7 board.

In addition, the Nexys A7 board, shown in Figure 2.5, has various peripherals that can be used to test the implemented applications: switches, LEDs, an accelerometer, a temperature sensor, 7-segment displays and a microphone, apart from the several I/O ports and memories already mentioned.

### 2.2.3 FPGA design steps

The usual FPGA design flow is showed in Figure 2.6, and as every other project, it starts with a planning stage. Once this is done, the first step of the design is the coding of the different modules that will form the complete project, as well as determining their relations. This code can be written in any of the standard hardware description languages, being VHDL and Verilog the most important ones.

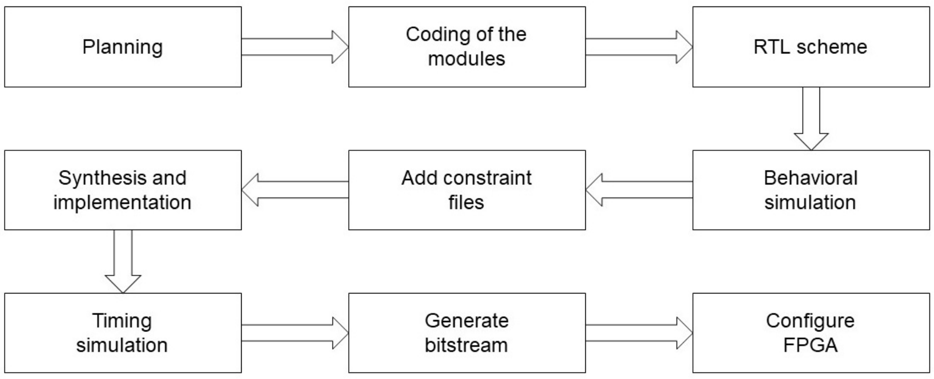

Figure 2.6: Design flow of a VHDL project.

The next step is elaborating a register transfer logic (RTL) scheme, where the connections between modules are easily seen. In addition, a behavioral simulation, where the timing delays and the common glitches are not taken into account, can be done to complete the first revision of the design. Before launching this simulation it will be necessary to create a testbench archive, also written in VHDL or Verilog, that controls the inputs required by the top module of the project.

Then, constraint files must be added, their purpose is to fix the assignments of the peripherals of the FPGA board to every input and output of the designed module. Moreover, they also serve to determine the width of the system clock pulses. These files are very important to guide the routing of resources done in the synthesis and implementation processes. Later, another simulation can be carried out, in this case taking into consideration the timing limitations of the device. It is called timing simulation.

Finally, the bitstream that will configure the FPGA can be generated. After that, the design will be loaded and the device ready to be used in the target application.

It is important to remark that although Figure 2.6 shows a linear design flow, it is possible to go back to any step whenever is needed. For example, this may happen if the proposed design does not fulfil the target problem specifications.

In this work, Xilinx's Vivado Design Suite [26] will be used to carry out all the explained steps of a FPGA design flow. It is worth noting that Intel (Altera in the past), who is the other big manufacturer of the sector, has also its own designing tool called Quartus Prime [27].

# Chapter 3

# ELM architecture design

In this chapter, a complete design of an ELM architecture will be done step by step, starting from the basic blocks (the hidden neuron, the activation function and the output neuron) up to a serial or parallel architecture. The objective is to implement in hardware an algorithm that allows different levels of parallelism, being able to select an architecture over another depending on the resources and speed parameters specified by the target application.

A generic VHDL code will be developed that can be particularised for different examples and training datasets by changing the configurable parameters, such as the number of neurons in each layer or the signals data format.

The ELM will be trained in Matlab using the code proposed by Guang-Bin Huang in [28], then, the different weights and biases will be translated to a fixed point fractional data format and stored in ROM memories using Vivado. After that, the results obtained with the hardware implementation will be compared with the ones obtained by Matlab and with the real targets.

As an example, the Breast Cancer Wisconsin dataset from the UCI machine learning repository [29] will be used to train the network. Each sample has 9 input attributes ($n$) and can be classified in 2 classes ($m$), benign and malign. Therefore, the input layer will have 9 neurons and the output one, two. For the hidden layer $L = 10$ was selected, based on a low resource implementation design published in [30], [31] that used the same database. Anyway, this example is only for a qualitative comparison between the architectures that will be presented.

## 3.1 Basic blocks

As it was explained in Section 2.1.1 there are two types of computing neurons, the ones in the hidden layer, that implement Equation 2.1, and the output neurons with their $\boldsymbol{\beta}$ weights. This difference is also showed in Figure 2.2. In this section, the design of both modules together with an example of a transfer function will be exposed.

### 3.1.1 Hidden neuron block

#### General distribution

This block will be in the hidden layer seen in Figure 2.2, its objective is to receive the input attributes $\boldsymbol{x}$ and multiply them by the random weights $\mathbf{w}$ before adding the bias $b$.

In Figure 3.1 the RTL scheme of the designed block can be seen, with the multiplier module (MultWX_neuron) and the accumulator (FeedbackSum). The latter adds the results of the former with the bias, storing the sum from one step to another. This process will be done as many times as attributes has the input data. The code of these modules can be seen in Appendixes B and C.

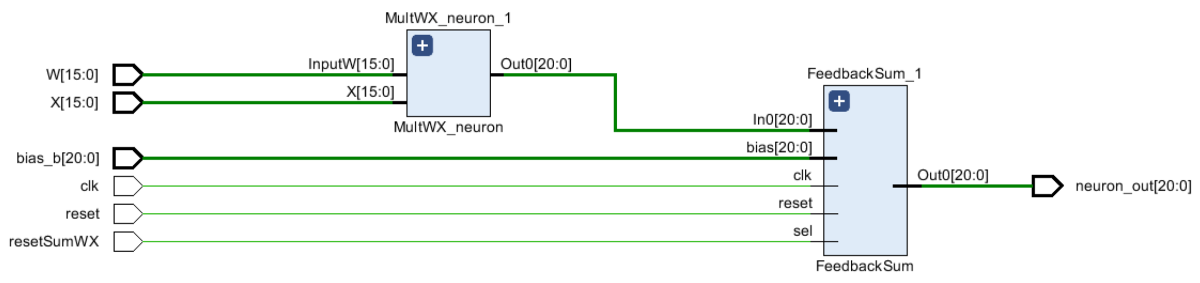

Figure 3.1: RTL schematic of the hidden neuron block generated by Xilinx Vivado software.

## Data format

As it can be observed, both the input attributes and weights are encoded in a 16-bit format, although there is a difference between them. The values of $\boldsymbol{x}$ are scaled in the [0, 1] range, and thus, all the 16 bits are fractional. On the other hand, weights $\mathbf{w}$ range from -1 to 1, needing 1 bit as the sign to determine whether the number is positive or not. The output of the multiplier is encoded in 21 bits, with a bit for the sign and 20 for the fractional part.

It is important to remark that when a term product is performed, the result requires of as many bits as both terms together to avoid loosing accuracy. Meaning, that in this design the multiplier resizes the output from 32 bits to 21, loosing 11 bits of accuracy. The same thing happens when adding $z$ different terms, the result will need $z$ additional bits. Therefore, vectors have to be truncated to a fixed length to avoid an increasing use of resources, since the result signals of a calculation will be used in another one, accumulating extra bits with each process.

The bias $b$ is also in the [-1, 1] range, with a 1.20 format (1 bit for the sign and 20 for the fractional part). In order to avoid overflow, the sums were carried out reserving 4 bits for the integer part, plus the sign bit, that leaves 16 for the fractional part (a 5.16 format). With this format, a precision up to the fourth decimal is reached, as showed in the following expression:

$$\text{Precision: } log_{10}(2^{16}) = 4.81 \Rightarrow 4 \text{ decimals of full precision} \tag{3.1}$$

The dynamic range stated above is valid for the example designs, but it will not be for a generic problem. Indeed, the number of bits reserved for the integer part has to be estimated depending on the maximum and minimum values that a series of operations can take. Thus, for an exercise with more input attributes and hidden neurons the proposed dynamic range will have to be changed.

### Simulation

Once all the signals are correctly defined and distributed, it is important to carry out a simulation in order to see how they change. In addition, it is also useful for checking possible errors. Figure 3.2 shows a behavioral simulation, run with a 100 MHz clock, of the hidden neuron block where the final value (0.95579) is accurate just in the first three decimals when compared with the Matlab calculations, done using 64 bit floating point vectors. This decreasing in the precision of the result is explained with the cumulative error due to finite resolution. In other words, the accumulator computes sums of elements with a certain precision, which ends in a bigger aggregated error.

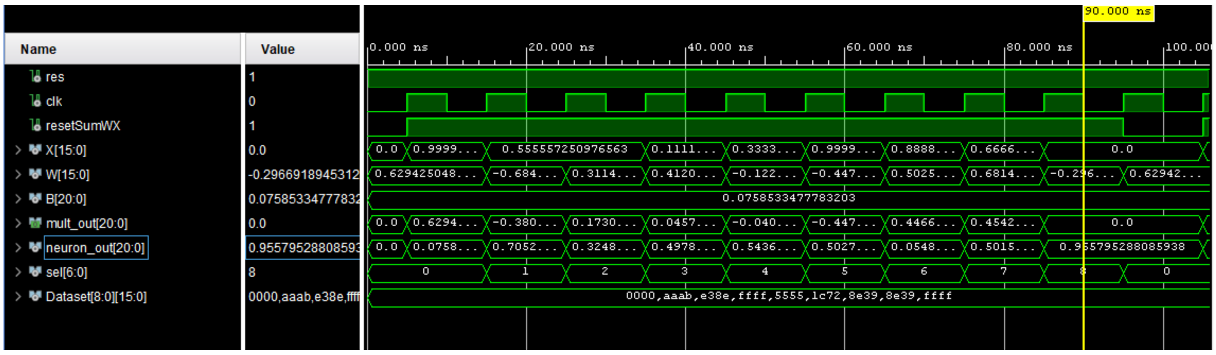

Figure 3.2: Behavioral simulation of the neuron block for the example problem generated by Xilinx Vivado software.

### Resource use and timing performance

Finally, the design can be implemented, obtaining the results showed in Table 3.1. The timing parameters have been generalised with the number of attributes in each input, $n$, that fixes the number of clock cycles needed to obtain the answer. This can be seen in the previous simulation with the 9 attributes of the example. The total time needed to obtain the final result will be the multiplication of the latency with the inverse of the frequency, 26.4 ns for the example problem. In addition, it is interesting to explain that the DSP is only used in the multiplier block.

| LUTs | Flip Flops | MUXs | DSPs | Latency (cycles) | Max. Frequency (MHz) |
|---|---|---|---|---|---|
| 22 | 21 | 0 | 1 | $n = 9$ | 341 |

Table 3.1: Resource demand and timing parameters of the hidden neuron block for the example problem.

### 3.1.2 Activation function block

The activation or transfer function block comes after the neuron module, actually, its input will be the output of the hidden neuron as it can be seen in Figure 2.2. There are many types of activation functions, but the sigmoid is one of the most common, and in fact, it is the one that has been implemented for the example. The main reason for this selection was its recurrent use and its higher accuracy in the final results when compared to other transfer functions.

The sigmoid, showed in Equation 3.2, has a symmetry around the value $x = 0$. This symmetry, that can be seen in Figure 3.3, enables the possibility of implementing just half of the sigmoid while still being able to obtain all the values of the full range.

$$f(x) = \frac{1}{1+e^{-x}} = 1 - f(-x) \tag{3.2}$$

It is interesting to note that the function is defined in the $[-\infty, \infty]$ range in the X axis and between 0 and 1 in the Y axis. Thus, when implementing the sigmoid in a VHDL module, it is of great importance to define its limits in the abscissa axis.

### Hardware implementation

There are many methods to implement this function in hardware, such as, bit level mapping, piecewise linear methods or Taylor series [32], [33]. But in this design, the sigmoid has been sampled and its outputs stored in a memory. There are two main reason for choosing this method over the rest: first of all, it is simple and extremely fast, and secondly, it only uses LUTs, that are abundant in FPGAs.

The matrix that stores the sampled values has two key parameters: the number of words and their length. For the latter, vectors of 16 bits have been selected, as the sigmoid is always positive of module inferior to one, all the length is reserved for the fractional part (unsigned 0.16 format).

The other parameter will be determined by the length of a selection vector, working as a pointer in the memory, that depends on the desired accuracy. To make the election, it has to be taken into account that the input of the sigmoid block, that will form the selection vector, is a 21 bit vector with a sign bit, 4 bits for the integer part and 16 for the fractional. In addition, if just half of the function is sampled, lets say the positive part, the sign bit can be ignored and the precision is doubled since the same number of samples are concentrated in half of the function.

In the end, a 10 bit vector was picked, which gave a memory of 1024 samples, from zero to the desired positive range. This range is another tunable parameter and its choice responds to a practical fact, it will depend on the selected number of bits of the inputs integer part. In other words, the sigmoid will be sampled in ranges: [0, 2), [0, 4), [0, 8) and [0, 16); based on whether 1, 2, 3 or the 4 bits of the integer part are chosen respectively. For example, if two integer bits are selected the maximum number that can fit is 3.99...

Initially, a range up to 8 was chosen, thus, 3 bits of the selection vector were integers and 7 fractional. This filled the last position of the array with the value corresponding to $f(x = 7.9922) = 0.9997$. Which gives an accuracy error of: $1 - f(x) = 3.38 \times 10^{-4}$, ensuring that the results are reliable up to the third decimal [33].

However, there is another error that has to be considered, the one caused by the truncation of the exact value of the sigmoid in a 16 bit binary number. The situation is the same as in Equation 3.1, which in theory will commit a smaller error than the previous one. Anyway, this can be checked by calculating the difference between both values for each of the sampled points, the results are showed in Figure 3.3, where the error is multiplied by $1 \times 10^4$.

It can be easily verified that this error is smaller than the previous one, therefore, the real accuracy of the calculations is set in the third decimal number.

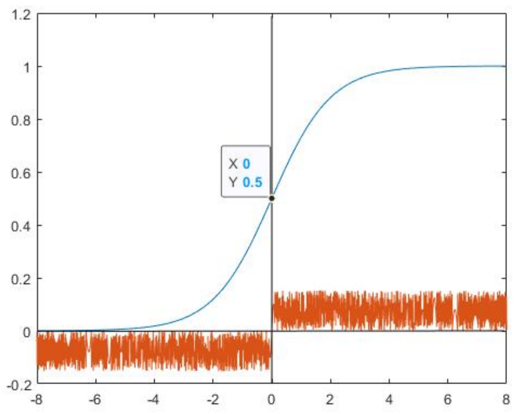

Figure 3.3: Sigmoid function (blue line) and the committed truncation error multiplied by $1 \times 10^4$.

On the other hand, in case that a negative input had to be redirected, the formula expressed in Equation 3.2 was used, calculating the output for the two's complement of the input and subtracting the result to 1.

The code of this block, that shows these calculations, can be seen in Appendix D.

Finally, it is important to state that just 243 LUTs are needed to implement this design, that is exclusively combinational since no flip flops are used.

### 3.1.3 Output neuron block

#### General distribution

This block will receive the output of the activation function, expressed by the vector $\mathbf{h}(\mathbf{x})$ in Equation 2.2, and will multiply them by the $\boldsymbol{\beta}$ weights. Its output will be the sum of these multiplications.

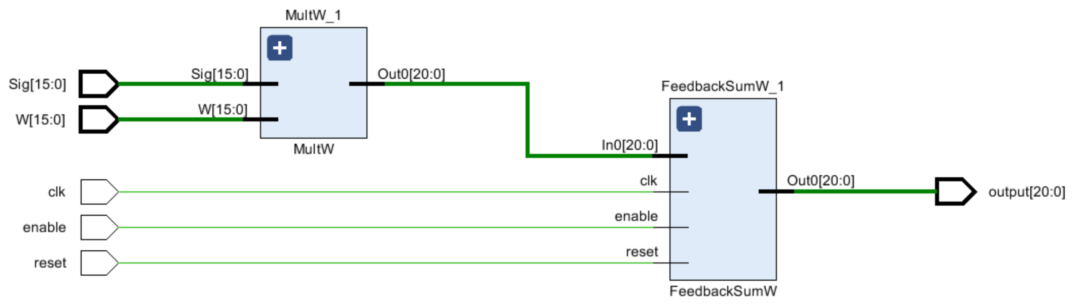

Figure 3.4: RTL schematic of the output neuron block generated by Xilinx Vivado software.

The outline of the complete block, showed in Figure 3.4, has the same modules as the hidden neuron (Figure 3.1) but in this case, there is no bias to be added to the final result. Indeed, that is the only difference from the hidden neuron modules code showed in Appendixes A, B and C.

**Data format**

As for the previous case, both inputs of the block are vectors of 16 bits, the already explained output of the sigmoid function and the $\boldsymbol{\beta}$ weights, with just a sign bit in a 1.15 format within the [-1, 1] range. The multiplier and the accumulator work as the ones in the hidden neuron block. The former, giving a 21 bit vector with the most significant one as the sign bit, and the latter, returning an output of the same length but in a signed 5.16 format.

**Resource use and timing performance**

Table 3.2 shows the results obtained after implementing the design. As before, the timing parameters have been generalised, but in this case the important parameter is $L$, the number of hidden neurons. Although, the most striking thing when looking at the tables is that the resources used by both designs are equal, something that could be expected after seeing their RTL schematics.

| **LUTs** | **Flip Flops** | **MUXs** | **DSPs** | **Latency (cycles)** | **Max. Frequency (MHz)** |
|---|---|---|---|---|---|
| 22 | 21 | 0 | 1 | $L = 10$ | 403 |

Table 3.2: Resource demand and timing parameters of the output neuron block for the example problem.

### 3.1.4 Output comparator block

If a classifier is going to be implemented, the highest result among the output neurons must be selected. Therefore, a module that computes the comparison between two inputs and returns the largest of them can be done. In fact, if there are more than two output classes a binary tree can be designed, this tree structure will take the results of previous comparators and will use them as inputs of the next set of them, determining at the end, the highest answer. The code of this block can be seen in Appendix E.

On the other hand, if the NN is going to be used in a regression problem, there is no need of selecting the largest result. In these examples, all the outputs are part of the final answer.

### 3.1.5 Memory blocks

The general architectures that will use the previous blocks will also need of memories from which to take the weights and biases for their calculations. These memories will be similar to the one that stores the sampled sigmoid, although this time, the number of saved values will be determined by $n$ and $L$. The random weight memory will have $n \times L$ values, while the bias and $\boldsymbol{\beta}$ ones only $L$.

## 3.2 Serial/parallel architectures

After explaining the basic blocks, it is time to put them together and form a complete ELM design. Two different ways of arranging these modules will be presented: a serial architecture, where there is only one block of each type that is reused constantly, and a fully parallel one, that is the most straightforward implementation of the schematic showed in Figure 2.2 with one block per neuron. Mixed models will not be taken into account in this section.

### 3.2.1 Serial architecture

#### General distribution

As mentioned before, this model is based on the reusing of blocks. In fact, the neuron and the sigmoid modules will work $L$ times with the same $\boldsymbol{x}$ inputs but with different weights and biases, emulating the $L$ different neurons. These weights $\mathbf{w}$ and bias $b$ are stored in the memories explained before and can be seen in Figure 3.5 prior to the hidden neuron block.

The output neuron requires of $m$ different $\boldsymbol{\beta}$ weight memories, one per output class of the problem. This forces to repeat completely the process of the previous blocks. In summary, the neuron and the sigmoid modules will be used $L \times m$ times per set of input attributes, while the output neuron and the $\mathbf{w}$ and $b$ memories will need $m$ runs. Finally, the $\boldsymbol{\beta}$ weights memories will only be used once.

The selection of the weights and biases will be controlled from the outside and then given as an entrance to the ELM module, as seen in Figure 3.5. The code of this architecture is not showed in the Appendix, since it is a direct use of the basic blocks.

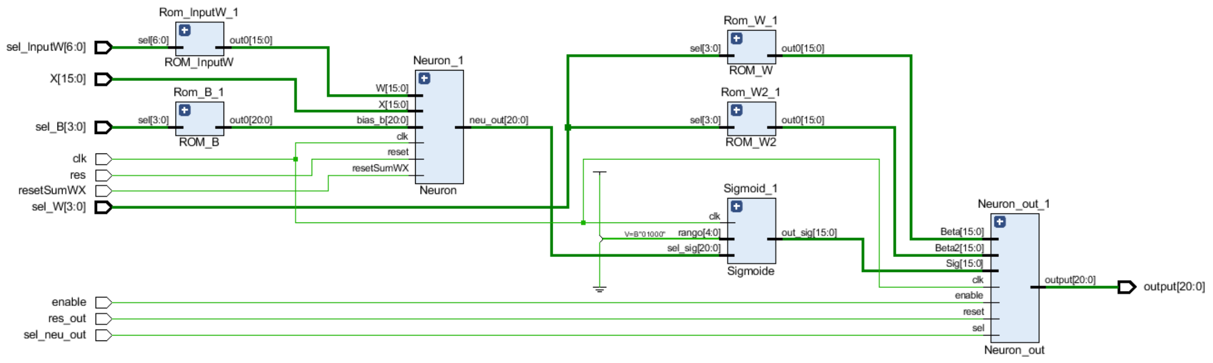

Figure 3.5: RTL schematic of the ELM serial architecture generated by Xilinx Vivado software.

#### Data format

The format of the signals will remain unchanged compared to the basic blocks explanations. Therefore, the output should be accurate up to the third decimal, but, as well as each module alone, the complete implementation also suffers from a cumulative error due to finite resolution. So, if the precision went from four decimal numbers to three in each block, in this case the accuracy decreases up to the second decimal.

Even so, the implemented design works perfectly, since the objective of the algorithm is to select the largest output between those given by the neurons in the last layer.

## Simulation

If a simulation is carried out, the evolution of the output signal can be easily followed. In this case, Figure 3.6 only shows the last cycles before a valid output is given, that will be marked by the signal called *ready*. Its purpose is to point out that the final result, with the highest answer among all the output neurons, is ready.

Moreover, the simulation shows two more signals that are worth of an explanation, *save* and *out_aux*. Both will be updated every time one of the $m$ sweeps to all the architecture is done, but while the former indicates when the result of a certain output neuron has to be saved the latter stores the highest one of the already computed neurons.

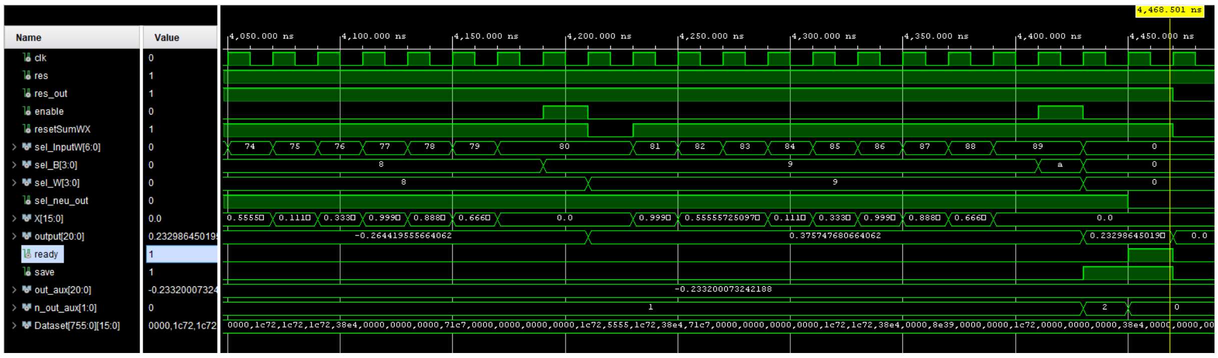

Figure 3.6: Behavioral simulation of the serial architecture for the example problem generated by Xilinx Vivado software.

## Resource use and timing performance

The main positive characteristic of this architecture is the small amount of resources that needs compared with other models, as it can be seen in Table 3.3. It is interesting to note that the multiplexers did not appear in the basic blocks resource tables because they are usually needed for the selection of the weights and biases of each memory, something that was not necessary before.

| LUTs | Flip Flops | MUXs | DSPs | Latency (cycles) | Max. Frequency (MHz) |
|---|---|---|---|---|---|
| 301 | 42 | 84 | 2 | $(L\cdot m)\cdot(n+1) + m\cdot(L+1) = 222$ | 68 |

Table 3.3: Resource demand and timing parameters of the serial architecture for the example problem.

On the other hand, it is a recursive architecture, which means that the final output will need much more time to be computed. This time will be determined by the timing parameters seen in Table 3.3: the latency, calculated following the explanations given in the "General distribution" section, and the maximum working frequency. For the example problem there is a 222 clock cycle latency, and thus, the correct result is obtained in 3.26 $\mu$s.

## 3.2.2 Parallel architecture

### General distribution

Unlike the previous model, in this case the objective is not a low resource design but a fast response architecture. Therefore, a direct implementation of Figure 2.2 will be done, devoting an individual module for each neuron. The RTL parallel structure is shown in Figure 3.7.

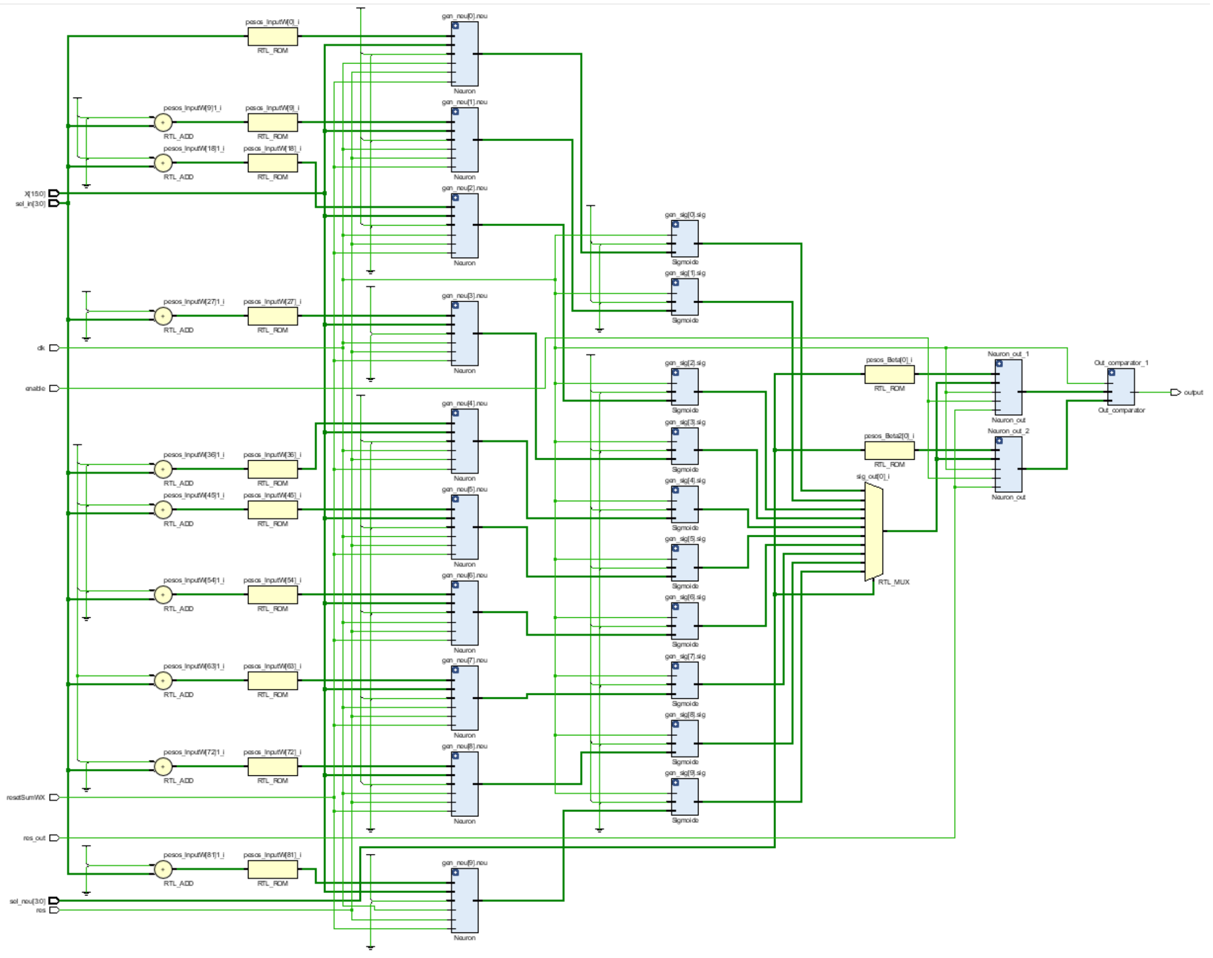

Figure 3.7: RTL schematic of the ELM parallel architecture generated by Xilinx Vivado software.

It can also be seen that in this design there are no general memories storing all the weights and the bias of every neuron, instead, each module has its parameters loaded in an individual memory. Although this measure increases drastically the number of LUTs needed, the controlling algorithm is simplified and the calculations accelerated, because the weights are closer to the DSPs, as it can be seen in Figure 2.4. The top module of this architecture, where all the parallel blocks are generated, is showed in Appendix F.

### Simulation

The simulation of this design, showed in Figure 3.8, has many differences with the serial case. First of all, since there is no need of repeating any type of process nor having to reuse any block, because each result is computed independently of the rest, the computation of the final answer is straightforward. The outputs of every neuron in the last layer are directly compared, giving just the index of the neuron with the largest value. For the example seen in the simulation it will be ’0’ or ’1’. This output will only be saved when the *ready* signal is ’1’, indicating the end of the calculations.

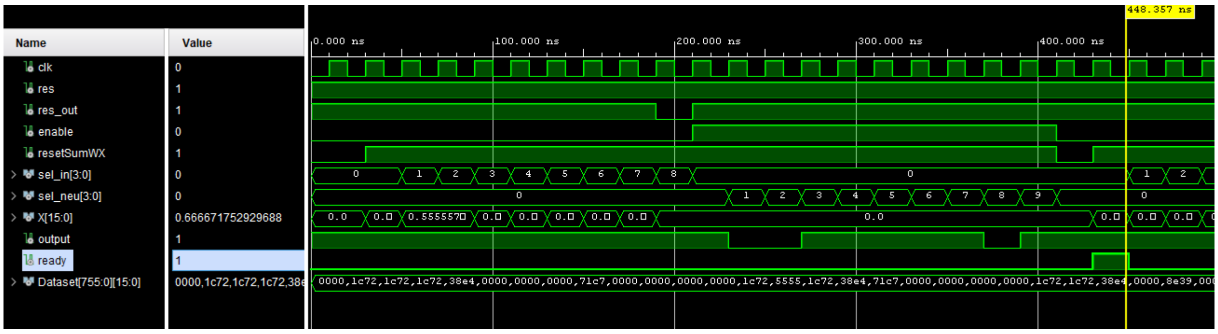

Figure 3.8: Behavioral simulation of the parallel architecture for the example problem generated by Xilinx Vivado software.

### Resource use and timing performance

As it can be seen in Table 3.4, this design needs of a DSP per neuron block ($L+m = 12$), which gives an example of its dependency on the number of neurons in each layer.

For this architecture, the total latency is calculated adding the individual latencies of every block, given that the sigmoid block, the comparator module and delivering the final output need of a clock cycle each. If the example is taken, it will require just 22 cycles, 10 times less than the serial architecture, returning the result in 0.34 $\mu$s. Therefore, although the maximum working frequency is slightly lower the computing speed is increased by the great change in the latency.

| LUTs | Flip Flops | MUXs | DSPs | Latency (cycles) | Max. Frequency (MHz) |
|---|---|---|---|---|---|
| 2407 | 252 | 823 | 12 | $n+1+(L+2) = 22$ | 62 |

Table 3.4: Resource demand and timing parameters of the parallel architecture for the example problem.

## 3.2.3 Comparative of architectures

As showed above, the differences between both architectures are very clear. The serial model is focused on reducing the amount of resources needed to implement the algorithm, thus, becoming the best option for applications where there is a lack of them. For example, this situation can be reached using a low resource device. However, the reusing of blocks can lead to a high response time, making the parallel architecture a better choice, since it centres its attention in reducing it without taking into account the resources used.

Another parameter that should also be considered is the final test classification accuracy that is obtained with both architectures. For the example explained in the introduction of this chapter, the ELM algorithm proposed by Guang-Bin Huang in [28] obtained an average overall accuracy (OA) of 94.05% after 10 retries.

The proposed designs accomplished the same results after being given the weights and the bias retrieved from the training previously done in Matlab. The test database, obtained from [31] and used in both cases included 84 samples, each one with its 9 attributes. To sum up, Tables 3.5 and 3.6 gather all the implementation results of both designs, including the resource used, the response time and the final accuracy.

| **Architectures** | **LUTs** | **Flip Flops** | **MUXs** | **DSPs** |
|---|---|---|---|---|
| **Parallel** | 2407 | 252 | 823 | 12 |
| **Serial** | 301 | 42 | 84 | 2 |

Table 3.5: Comparative of the resource demand for the example problem between the parallel and the serial architectures.

| **Architectures** | **Latency** | **Max. Frequency** | **Time needed** | **Overall Accuracy** |
|---|---|---|---|---|
| | **(cycles)** | **(MHz)** | **($\mu$s)** | **(%)** |
| **Parallel** | 22 | 62 | 0.34 | 94.05 |
| **Serial** | 222 | 68 | 3.26 | 94.05 |

Table 3.6: Comparative of the timing parameters for the example problem between the parallel and the serial architectures.

### 3.2.4 Implementation and test of the parallel architecture

If the FPGA design flow explained in Section 2.2.3 is followed, it can be seen that the last steps are related to the configuration of the device. For that purpose, the already explained test database can be stored in a memory connected to the ELM parallel top module. Then, the bitstream can be generated and loaded into the device board.

Once this is done, the control signals can be introduced and modified, using the available I/0 ports, and the output showed in the various LEDs. The relation of the board peripherals with every signal will be defined by the constraint files.

## 3.3 Chapter conclusions

First of all, it has to be said that the presented basic blocks were programmed separately in different Vivado projects, following the VHDL design flow explained in Section 2.2.3. Then, those modules were arranged so that they suited the desired architecture, either a serial or parallel one. That way, every block could be tested and their implementation results obtained independently.

As said before, the serial and parallel designs are just two types of architectures of the many that can be used to implement a NN such as the ELM. Those two were selected because each one represents a completely different approach to the same problem.

Nevertheless, sometimes there is no need of such a low resource implementation as the serial one, or a fast response answer as the parallel architecture. Therefore, a mixed model might be a better idea for certain applications. For example, general memories can be maintained or the neuron block reused if the number of sweeps is low, while the output is paralleled to avoid too many repetitions when the quantity of classes increases.

# Chapter 4

# Design of a simplified architecture for ELM

As it has been explained in the previous chapter, it can be of great importance the quantity of resources used by a hardware implemented design. The reasons can be diverse, from the necessity to use a very low-cost resource device to the possibility of the application being just a small part of a bigger project that has to be implemented together.

In addition, sometimes it is not enough to use an architecture such as the serial one. Although it reduces the demand of resources by reusing the hardware blocks, other measures have to be taken into account. The following two sections, that will explain some of these measures, are based on the already mentioned low resource implementation design published in [30], [31]. Basically, the characteristics of the binary digital arithmetics are exploited. Finally, an analysis and comparison of different architectures that include these simplified designs will be done.

## 4.1 Modified random weights w

One of the characteristics of the ELM algorithm is that the weights and biases of the hidden neurons are set at random. Moreover, they remain unchanged during the training phase because they serve to calculate the $\boldsymbol{\beta}$ weights of the output neurons. Thanks to this quality, these parameters can be defined at will by the designer.

Therefore, the weights $\mathbf{w}$ can be selected in a way that the multiplier seen in Figure 3.1 is not needed, hence, avoiding the use of the DSP that is a more scarce resource than others. The authors in [30], [31] define these weights as a power of two, following Equation 4.1:

$$w_{ij} = s \cdot 2^{-k} \tag{4.1}$$

Being $i$ the neuron number in the hidden layer, $1 \leq i \leq L$, $j$ the input attribute, $1 \leq j \leq n$, $s$ the sign of a random number stored in a memory and $k$ a random integer in the range [0, $R$] being $R$ a parameter selected by the designer. Following what the authors proposed, $R$ was chosen to be 7, which sets the minimum absolute value for the weights in 0.0078.

Defining the $\mathbf{w}$ weights in this way exploits a property of the binary numbers: when dividing or multiplying a binary number by a power of two, the operation can be made directly by shifting the binary vector a number of positions marked by the exponent of the power to the right or left respectively. In the case of Equation 4.1, the multiplied input attributes $x_j$ will be shifted $k$ positions to the right, completing the vector with '0's from the left.

The sign $s$ is then used as a selection bit for a multiplexer that will decide between the explained $wx$ product or its complement. The complete outline of the this hidden neuron module is showed in Figure 4.1. In addition, the code of these new modules can be seen in Appendix G.

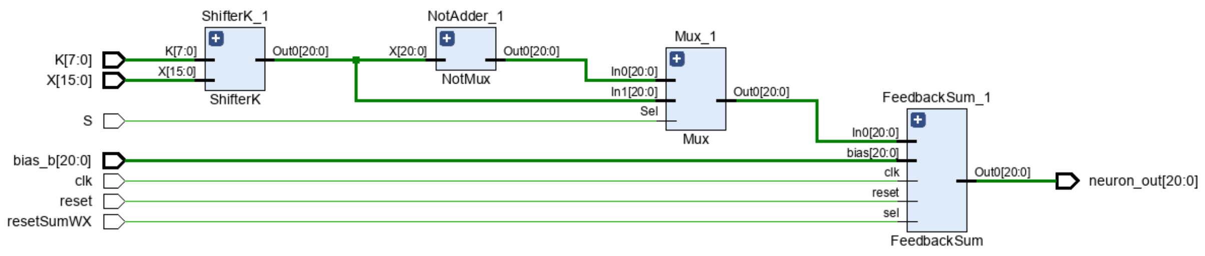

Figure 4.1: RTL schematic of the simplified hidden neuron block generated by Xilinx Vivado software.

### Data format

The signals seen in the schematic are encoded in a 21 bit vector with a constant format: a sign bit, 4 bits for the integer part and 16 for fractional. This format ensures that there will not be overflow in the sums carried out by the accumulator, since the inputs $x \in$ [0, 1] and the weights $w \in$ [-1, 1].

### Resource use and timing performance

Both the integers $k$ and the signs $s$ are stored in memories, they will be loaded after being randomly generated in Matlab, guaranteeing that the random weights are completely arbitrary. Although this proposal of a simplified hidden neuron eliminates the need of a multiplier, avoiding the use of a DSP, it increases the number of LUTs since it replaces the random weights memory with two of the same size, for the $k$ and $s$ values.

Anyway, for certain devices with low DSP resources this proposal is a better option. On the other hand, this module is slightly slower than the original hidden neuron, although the latency remains the same. The implementation results can be seen in Table 4.1:

| **LUTs** | **Flip Flops** | **MUXs** | **DSPs** | **Latency (cycles)** | **Max. Frequency (MHz)** |
|---|---|---|---|---|---|
| 65 | 21 | 0 | 0 | $n = 9$ | 232 |

Table 4.1: Resource demand and timing parameters of the simplified hidden neuron block for the example problem.

Looking at the first of them, it stands out the zero multiplexers used by the design, specially when one of the modules is a MUX. This can be explained taking into account that the Vivado software implements that module using LUTs to avoid wasting resources, since the multiplexers of the working FPGA are too large fot it.

## 4.2 Election of a different activation function

Apart from the sigmoid block explained in section 3.1.2, the activation function can be of different types, such as, a hardlimit or a logaritmic function. Each one has its advantages and disadvantages, but for a simplified implementation model the former becomes the best option.

The hardlimit function is defined as follows:

$$f(x) = \begin{cases} 1, & \text{if } x > 0 \\ -1, & \text{if } x \leq 0 \end{cases} \tag{4.2}$$

As it can be seen, it is a very simple expression, since it only takes into account if the input is positive or not. Therefore, the results of the calculations performed with this function will be less accurate than with more complex elections. Nevertheless, the ELM classification algorithm that is being used only considers the largest result given by the different output neurons. Thus, the variation between answers will be set by the $\boldsymbol{\beta}$ weights, allowing to change the transfer function without loosing too much accuracy.

In Figure 4.2 an outline of the proposed output neuron block is showed, where it is interesting to note that it includes the activation function module as well, integrated in the data path.

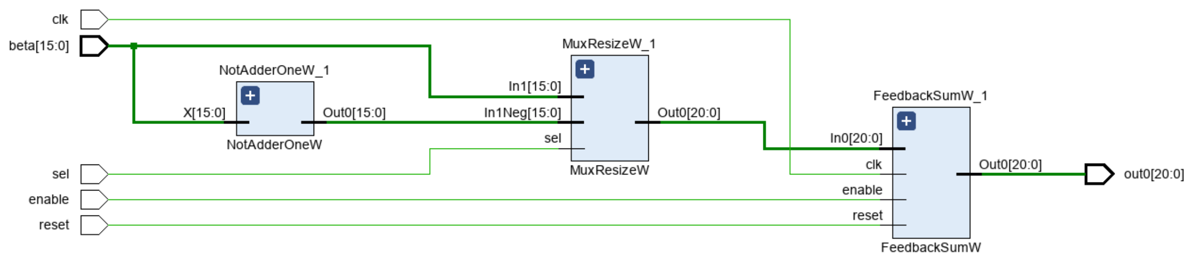

Figure 4.2: RTL schematic generated by Xilinx Vivado software of the simplified output neuron block with the hardlimit function integrated.

### Data format

The hardware implementation of Equation 4.2 is more straightforward than the sigmoid case. First of all, instead of taking the complete 21 bit vector of the hidden neuron output, only the sign bit is taken. Then, this bit will be used as the selector of the multiplexer that will decide between the $\boldsymbol{\beta}$ weights or its two's complement, simulating a multiplication between the weight and 1 or -1. In addition, the MUX module also resizes the signal, giving a 21 bit vector with a sign bit, 5 bits for the integer part and 15 for the fractional. The final result of this block has the same format, loosing one fractional bit of accuracy compared to the original output neuron design. The code of the new modules can be checked in Appendix H.

**Resource use and timing performance**

This design offers a great reduction in the use of resources, first of all, the 243 LUTs needed in the sigmoid function block will not be necessary. Moreover, the DSP used for the multiplier of the original output neuron module will also be unnecessary. On the other hand, the maximum frequencies of both modules are very similar, as well as the latency.

| LUTs | Flip Flops | MUXs | DSPs | Latency (cycles) | Max. Frequency (MHz) |
|---|---|---|---|---|---|
| 37 | 21 | 0 | 0 | $L = 10$ | 401 |

Table 4.2: Resource demand and timing parameters of the simplified output neuron block for the example problem.

In summary, it can be said that the proposed design reduces considerably the amount of resources needed while maintaining the original data path speed. Although, it may present a decrease in the final classification accuracy, something that should be kept in mind.

## 4.3 Analysis of simplified architectures

Once these measures are explained, it is time to develop a complete ELM implementation. First of all, it has to be remarked that the objective of these new architectures is to reduce the use of resources. Therefore, a serial outline will be followed, since it would not make much sense to apply a resource reduction measure in a high use architecture type.

So, in this section, designs for all the combinations of both simplifying measures with the original modules will be presented. In addition, a comparison of resources and speed will be made between them and with the original serial architecture.

### 4.3.1 Modified random weights with a sigmoid activation function

As a preliminary study, and based on the analysis of the previous section, this design will be just slightly different from the original serial architecture. Basically, the multiplier of the neuron block will be eliminated at the cost of introducing an extra memory. Although it may seem that is not worth it, depending on the target application, eliminating that DSP can be of great importance. Anyway, its usefulness will rely on maintaining or improving the existing classification test accuracy and computing speed.

Figure 4.3 shows the outline of the design where the necessary 6 memories, including the sigmoid function storage, can be seen.

**Resource use and timing performance**

Comparing Table 4.3 with its analogous of the original serial architecture, Table 3.3, the initial estimations are confirmed, there are few changes between both designs. The main ones are the elimination of a DSP and the small difference in the LUT number even with an extra memory. The latter can be explained knowing that the Vivado software is

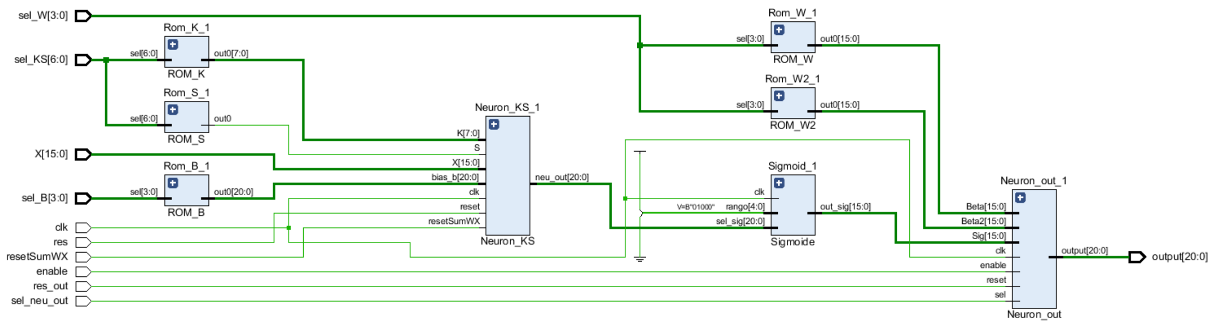

Figure 4.3: RTL schematic generated by Xilinx Vivado software of a simplified serial architecture with modified random weights and the sigmoid function integrated.

programmed to route the resources so that the least amount of them is used. Moreover, the timing situation is almost identically the same.

| **LUTs** | **Flip Flops** | **MUXs** | **DSPs** | **Latency (cycles)** | **Max. Frequency (MHz)** |
|---|---|---|---|---|---|
| 308 | 42 | 73 | 1 | $(L{\cdot}m){\cdot}(n{+}1) + m{\cdot}(L{+}1) = 222$ | 67 |

Table 4.3: Resource demand and timing parameters of the simplified serial architecture with modified random weights and a sigmoid activation function for the example problem.

### 4.3.2 Original weights with a hardlimit transfer function

In this case, a major change is performed, since the activation function largely determines the network performance. It will also simplify the internal functioning of the network, as well as reducing greatly the LUTs used in memories. Indeed, Figure 4.4 shows the simplicity of this architecture, with just two memories for each of the neuron types.

#### Resource use and timing performance

In total, 189 look up tables, 68 MUXs and 1 DSP are saved, becoming a great option for low-cost applications. On the other hand, although the latency does not change with respect to the other serial models, the result will be computed in just 0.75 $\mu$s, due to the high maximum working frequency of 298 MHz. Both, the low resource use and the high frequency, are characteristics of the hardlimit transfer function.

| **LUTs** | **Flip Flops** | **MUXs** | **DSPs** | **Latency (cycles)** | **Max. Frequency (MHz)** |
|---|---|---|---|---|---|
| 112 | 42 | 16 | 1 | $(L{\cdot}m){\cdot}(n{+}1) + m{\cdot}(L{+}1) = 222$ | 298 |

Table 4.4: Resource demand and timing parameters of the simplified serial architecture with a hardlimit activation function for the example problem.

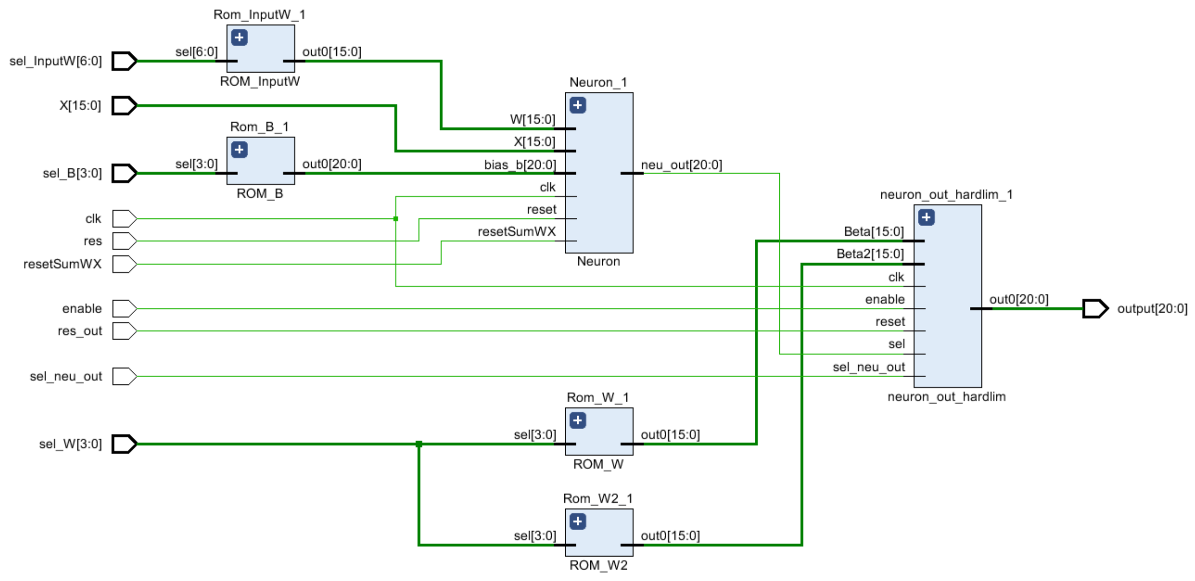

Figure 4.4: RTL schematic generated by Xilinx Vivado software of a simplified serial architecture with the hardlimit transfer function integrated.

## 4.3.3 Modified random weights with a hardlimit activation function

This design includes both simplifying measures, which should turn it into the most resource friendly architecture until now. Its outline, that is very similar to the previous model, is shown in Figure 4.5 with the only difference of an extra memory and the change of hidden neuron block type.

### Resource use and timing performance

Once implemented, it can be seen from Table 4.5 that this is the architecture with the lowest resource demand of the already presented ones, although there is a slight increase in the LUTs compared to the previous model because of the extra memory.

| LUTs | Flip Flops | MUXs | DSPs | Latency (cycles) | Max. Frequency (MHz) |
|---|---|---|---|---|---|
| 124 | 42 | 3 | 0 | $(L{\cdot}m){\cdot}(n{+}1) + m{\cdot}(L{+}1) = 222$ | 303 |

Table 4.5: Resource demand and timing parameters of the simplified serial architecture with the modified random weights and the hardlimit transfer function for the example problem.

Furthermore, the hardlimit activation function not only saves resources but also increases greatly the maximum working frequency, computing the final result in just 0.74 $\mu$s, very similar to the previous case.

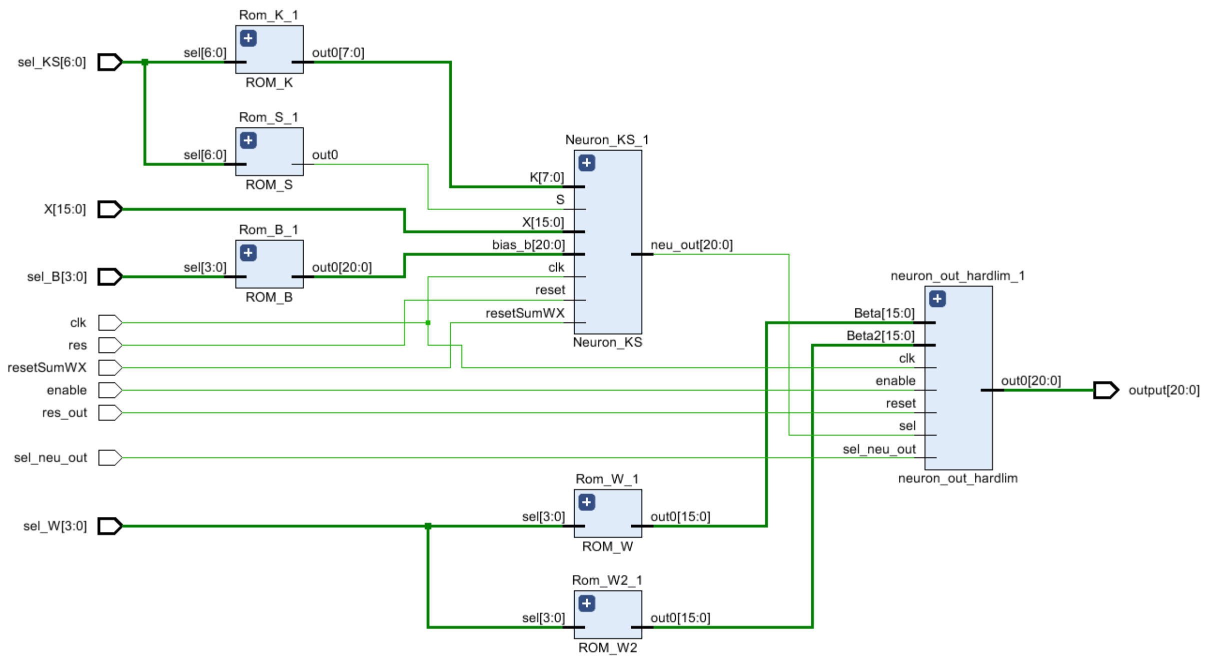

Figure 4.5: RTL schematic generated by Xilinx Vivado software of a simplified serial architecture with the modified random weights and the hardlimit transfer function integrated.

## 4.3.4 General comparative with the original designs

Although a partial comparative of each architecture with the serial design has already been done, it is interesting to put everything in context, not only between the serial models but also with the parallel case. In addition, the total test classification accuracy or overall accuracy (OA) has to be taken into account, since it is a very important parameter to bear in mind when selecting an architecture over another.

Therefore, each model was initialised randomly and trained in Matlab 10 times with the example data, for each retry the memories of the weights and the bias in Vivado were updated with the new parameters. At last, the network was implemented and simulated with the same test database, averaging its overall accuracy. The final results of all the designs are showed in Tables 4.6 and 4.7.

| **Architectures** | **LUTs** | **Flip Flops** | **MUXs** | **DSPs** |
|---|---|---|---|---|
| **Parallel: Random w / Sigmoid** | 2407 | 252 | 823 | 12 |
| **Serial: Random w / Sigmoid** | 301 | 42 | 84 | 2 |
| **Serial: Power-of-2 w / Sigmoid** | 308 | 42 | 73 | 1 |
| **Serial: Random w / Hardlimit** | 112 | 42 | 16 | 1 |
| **Serial: Power-of-2 w / Hardlimit** | 124 | 42 | 3 | 0 |

Table 4.6: Comparative of the resource demand for the example problem between various architectures.

| **Architectures** | **Latency** | **Max. Frequency** | **Time needed** | **Overall Accuracy** |
|---|---|---|---|---|
| | **(cycles)** | **(MHz)** | **($\mu$s)** | **(%)** |
| **Parallel: Random w / Sigmoid** | 22 | 62 | 0.34 | 94.05 |
| **Serial: Random w / Sigmoid** | 222 | 68 | 3.26 | 94.05 |
| **Serial: Power-of-2 w / Sigmoid** | 222 | 67 | 3.32 | 94.53 |
| **Serial: Random w / Hardlimit** | 222 | 298 | 0.75 | 86.78 |
| **Serial: Power-of-2 w / Hardlimit** | 222 | 303 | 0.74 | 86.07 |

Table 4.7: Comparative of the timing parameters for the example problem between various architectures.

As it can be seen, each model has its strengths and its weak points, and the selection of a design over another has to be made once the characteristics of the target application are set. However, it is interesting to note the importance of the transfer function in the final accuracy, since the difference between the models with the sigmoid function and the hardlimit is notorious. On the other hand, the hardlimit function enables to obtain the result in a shorter time, only the parallel architecture with its high use of resources is able to improve it.

## 4.4 Chapter conclusions

It is very common to try to reduce the resource demand in hardware implementation applications, since they are limited. Therefore, various simplifying measures can be applied with that objective, such as the two different ideas presented in this chapter. Nevertheless, it is important to bear in mind the inherent trade-off between speed, resource demand and accuracy that every architecture has.

For example, if a very low resource implementation is needed, independently of the rest of the parameters, the hardlimit models become the best options, specially with the modified weights. They also give a fast response. However, if the key parameters of the target application are time and accuracy, the parallel architecture needs to be used. Simplifying methods have not been implemented for this type of structure, since it will not make much sense to try to reduce the amount of resources used in a model that does not care about them.

Moreover, the serial architecture with the sigmoid function, independently of the type of random weights, gives the slowest response but with a high accuracy and a medium-low use of resources. Then, between both models, the one with the modified weights is clearly a better option, since it maintains the accuracy and speed but saves some multiplexers and a DSP. This is due to the way the ELM algorithm is defined, as explained in Section 2.1.2, the $\mathbf{w}$ weights are set at random and remain unchanged. Later on, they serve to determine the $\boldsymbol{\beta}$ weights, letting the designer define them at will. This allows to take advantage of a property of binary numbers that verifies that a multiplication of a power of two can be computed as a shift.

# Chapter 5

# Development of an FPGA-based ELM for satellite images classification

Nowadays, satellites are widely used in multiple applications, from weather forecasting to Earth's surface element detection. In fact, this chapter is related to the former, since many of the hardware implementations done to classify the satellite images use FPGAs. This is mainly because of the high parallelism that these devices allow, together with the possibility of changing remotely, partially or totally, its internal configuration [21].

Many of the satellite imaging systems use sensors that are able of capturing light from wavelengths outside of the visible range. These systems are called multispectral or hyperspectral, depending on the quantity and width of the bands from which information is acquired [34]. Usually, light from the visible (VIS) and the near infrared (NIR) ranges is captured.

These spectral systems are of great interest since they are able of detecting elements invisible to other types of sensors, for example, internal bruises and defects in food inspection procedures [35]. Therefore, they can become a very powerful tool when trying to distinguish and classify different constituents in a sample or area. Indeed, one of the industry sectors where these images are of great interest is agriculture, because they can be used to obtain information from soil degradation or crop chlorophyll content for example [36].

In this chapter, a satellite image classification will be conducted applying what has been explained throughout the work. First of all, the satellite image that will serve as train and test databases will be selected. Then, one of the architectures already presented will be chosen and configured, setting the number of neurons of the hidden layer. Finally, an analysis of resource use, timing performance and classification accuracy will be done.

## 5.1 The data

In this application, a Landsat satellite image obtained from [28] will be used. This website provides various databases of different types, and has a particularity, the train and test datasets are randomly generated every time it is downloaded. Anyway, the original repository from where the data is taken can be found in [37]. There, it can be seen that the image has been acquired using a multispectral sensor that obtained information from four different spectral bands, two in the visible range (green and red) and the other two in the NIR.

Each sample of the database has 36 input attributes, corresponding to a 3x3 square, formed by the target pixel and its first neighbours, in every of the four bands. In addition, the target or central pixels are classified in 6 different classes: red soil, cotton crop, grey soil, damp grey soil, soil with vegetation stubble and very damp grey soil. In total, the acquired data has 6435 samples, 3217 for train and 3218 for test. On the other hand, since the datasets are formed in a random order and certain pixels have been removed, it is not possible to reconstruct the original image, but an example of a Landsat image can be seen in Figure 5.1.

Figure 5.1: Image of the Ebro delta taken by Landsat 8 on January 31, 2018 (Courtey of NASA's Earth Observatory).

As an interesting fact, the Landsat program was created by the United States of America, and from 1965 has gathered Earth's information from several remote sensing missions. First in airborne flights but since 1972 using satellite platforms [38]. The program has launched 9 different satellites in total, being Landsat 5 the most famous one. Since it holds the Guinness World Record for the longest operating Earth observation satellite, after delivering data for 28 years and 10 months [39].

## 5.2 Architecture selection

The next step is the architecture selection of the classifying neural network, which will be an ELM, as in the examples given in previous chapters. When choosing between the options available, a serial or parallel design, or even a mixed model, it is necessary to set which is the most important characteristic for the target application. In other words, a decision between speed and low resource use will have to be taken.

For this problem, speed has been prioritised selecting a parallel architecture, because if the NN is going to be implemented in a FPGA mounted in a satellite a fast response is needed. Moreover, the Nexys A7 FPGA board that will be used has enough resources available to implement the desired design. On the other hand, the activation function of the hidden layer neurons must also be chosen. Two different options have been presented in Chapters 3 and 4, the sigmoid and hardlimit functions respectively. Each one has its particular advantages and disadvantages, but following the reasoning exposed before, the former was selected. Preferring a higher classification accuracy over a resource saving.

Lastly, the number of neurons in the hidden layer must be set. Therefore, a neuron sweep from 1 to 500, calculating the average test classification accuracy over 10 retries, was done. The results, seen in Figure 5.2, show a slow growing tendency after the first hundred neurons. Hence, selecting an early value of the almost horizontal slope might be the best decision.

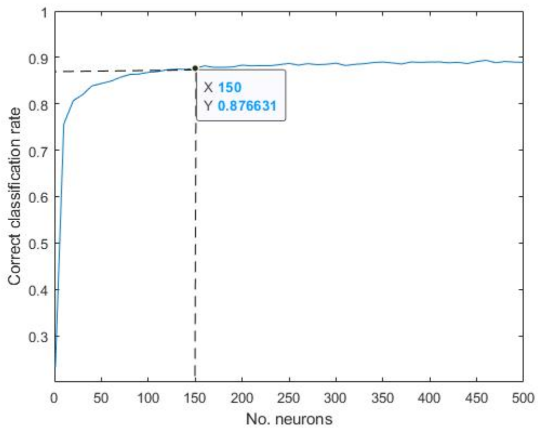

Figure 5.2: Average test classification accuracy of the ELM network depending on the number of hidden neurons.

In the end, a 150 hidden neuron configuration was chosen, obtaining an average accuracy of 87.66%, just 1% lower than the 500 neuron case. In this way, many resources can be saved without an important loss in the final accuracy.

So, in summary, a parallel architecture with a sigmoid activation function and 150 neurons in the hidden layer will be implemented. In addition, the input and output layers will have 36 and 6 neurons respectively.

On the other hand, the dynamic range of the data signals has to be checked, due to the change in the number of neurons in the different layers. Until now, a signed 5.16 format has been used, as shown in the modules of the Appendix, with 4 bits for the integer part. But with 36 input attributes and 150 hidden neurons, the accumulators will compute values that might overflow the previous range. Therefore, a signed 7.14 format will be adopted in the hidden neuron blocks and a signed 9.12 in the output ones, avoiding this possible problem.

To finish with the architecture design, the selection of the highest result among the output neurons must be done. A very effective way to compute this calculation is to implement a binary tree, that is to say, a two input comparator scheme. It is based on a VHDL module, showed in Appendix E, that will give the largest value between two inputs. This way, a tree structure can be defined, with the outputs of two comparators being the inputs of the next one until the highest result is determined.

## 5.3 Analysis of the implementation results

Once the ELM architecture is fixed, the training phase begins, where the first of the mentioned datasets is used to generate the weights and the bias of every neuron. This process is done in Matlab before loading the parameters in different memories in Vivado.

After that, the second dataset will serve to test the performance of the proposed design. Finally, the NN can be implemented in a FPGA and its results analysed.

| LUTs | Flip Flops | MUXs | DSPs | Latency (cycles) | Max. Frequency (MHz) | Time needed ($\mu$s) |
|---|---|---|---|---|---|---|
| 36224 (57.14%) | 3276 (2.58%) | 12405 (26.09%) | 156 (65%) | 190 | 51 | 3.73 |

Table 5.1: Resource demand and timing parameters of the parallel architecture for the satellite image classification.

The usage rates showed in Table 5.1 can be compared with the available resources of the Nexys A7 board presented in Section 2.2.2. This way, it can be seen that the total number of hidden neurons in a parallel architecture will be limited by the 240 DSP slices of the FPGA. Therefore, the selected network could have had more, until a maximum of 234 hidden and 6 output neurons.

Nevertheless, there is another device in the Artix 7 family with higher capacity, the XC7A200T part number packages, with up to 740 DSPs. Indeed, Figure 5.3 shows a comparison of usage rates between both FPGAs. Moreover, the Virtex family has a device with 3360 DSPs, being the highest value for the complete 7-Series [40].

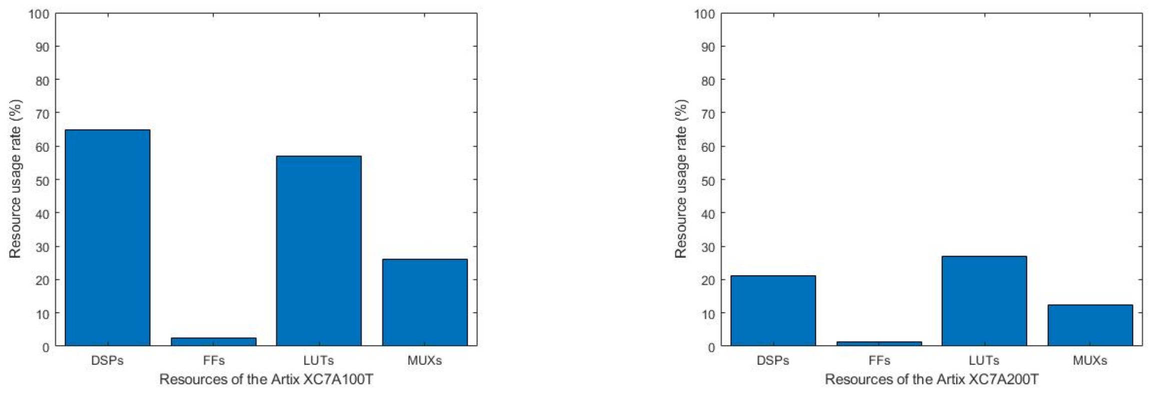

Figure 5.3: Usage rates of the main resources of the used FPGA (XC7A100T) and the highest capacity one in the Artix 7 family (XC7A200T) for the final application.

On the other hand, the design gives an answer in just 3.73 $\mu$s, fulfilling the speed expectations. It has to be noted that Table 5.1 shows a 190 clock cycle latency, obtained by computing the generic expression of Table 3.4 and adding an extra cycle due to the delay introduced by the binary tree that selects the maximum output.

Furthermore, the classification test accuracy of the network was of 87.32%, near the average presented in Figure 5.2.

To conclude, the alternative of implementing a serial architecture can be analysed. This way, more neurons could be added without taking into account the resource demand, increasing the classification accuracy. However, the response time will scale considerably. In fact, if the 150 hidden neuron configuration was to be implemented in a serial design, the latency will be of 34206 clock cycles according to the expression shown in Table 3.3. Which supposing a maximum working frequency of 68 MHz, even though it will likely be lower, means that the answer for just an input sample will come after 503 $\mu$s.

# Chapter 6

# Conclusions

First of all, it has to be said that designing every VHDL module separately, following the design flow explained in Section 2.2.3, gives the possibility of achieving different specifications for the same application. This can be easily seen when comparing the architectures that implement both of the presented hidden neuron blocks, the original and the simplified one. With the later accomplishing similar classification results but with a certain resource saving, as showed in Tables 4.6 and 4.7.

When setting together the different modules several structures can be constructed, depending on the characteristics of the target application. Indeed, knowing its specifications is fundamental to select a network architecture over another. For example, if a short respond period is pursued, a parallel design must be implemented, once its inherent resource cost is accepted. On the other hand, if the objective is a low resource use application a serial model will fit better.

In addition, a high classification accuracy can also be demanded, forcing to select a sigmoid transfer function over a hardlimit, that uses less resources. A complete comparative of some designs can be seen in Tables 4.6 and 4.7, but there are several other options. Indeed, mixed models have not been taken into account in this work, and having the possibility of paralleling part of the computing process while letting the rest as a serial structure might be of great interest in some cases.

In summary, when designing a network for a new application three main decisions must be taken: the type of architecture, the number of neurons in the hidden layers and the transfer function. These decisions have been addressed in Chapter 5 for a satellite image classification process, in that case, time and accuracy were prioritised over the resource demand. Therefore, a parallel model with a sigmoid function was selected.

As a future work, this design can still be further optimised if a pipeline structure is implemented, which will reduce the latency greatly. However, this configuration needs of a more sophisticated controlling algorithm to ensure that every signal reaches its destination when it has to. Moreover, the lengths of the signals of every module can also be generalised, so that the user can set them.

Finally, it is worth noting that this work has led to a better understanding of the capabilities of FPGAs, completing what was learned in various subjects throughout the degree: Digital electronics, Digital systems design and Electronic devices, among others. Furthermore, the potential of the NNs has been perceived, specially because of their adaptability to very diverse applications, ranging from a function regression to an image classification.

# Bibliography

[1] E. Mollick, “Establishing moore’s law,” *IEEE Annals of the History of Computing*, vol. 28, no. 3, pp. 62–75, 2006. DOI: `10.1109/MAHC.2006.45`.

[2] C. Maxfield, *The Design Warrior’s Guide to FPGAs: CD-ROM*, ser. The Design Warrior’s Guide to FPGAs: Devices, Tools and Flows. Newnes, 2004.

[3] J. J. Rodriguez-Andina, M. D. Valdes-Pena, and M. J. Moure, “Advanced features and industrial applications of fpgas—a review,” *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853–864, 2015. DOI: `10.1109/tii.2015.2431223`.

[4] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús, *Neural Network Design.* 2014.

[5] K. Basterretxea, V. Martinez, J. Echanobe, J. Gutiérrez-Zaballa, and I. Campo, “Hsi-drive: A dataset for the research of hyperspectral image processing applied to autonomous driving systems,” *32nd IEEE Intelligent Vehicles Symposium (IV21), (accepted)*, July 2021.

[6] M. Lan, Y. Zhang, L. Zhang, and B. Du, “Global context based automatic road segmentation via dilated convolutional neural network,” *Information Sciences*, vol. 535, pp. 156–171, 2020. DOI: `10.1016/j.ins.2020.05.062`.

[7] L. Györfi, G. Ottucsák, and H. Walk, *Machine learning for financial engineering.* World Scientific, 2012, vol. 8.

[8] T. J. Cleophas, A. H. Zwinderman, and H. I. Cleophas-Allers, *Machine learning in medicine.* Springer, 2013, vol. 9.

[9] A. Grewal, “Review report on vhdl (vhsic hardware description language),” *language*, vol. 27, p. 28, 2018.

[10] K. El Bouchefry and R. S. de Souza, “Chapter 12 - learning in big data: Introduction to machine learning,” in *Knowledge Discovery in Big Data from Astronomy and Earth Observation*, P. Škoda and F. Adam, Eds., Elsevier, 2020, pp. 225–249. DOI: `10.1016/B978-0-12-819154-5.00023-0`.

[11] C. M. Bishop, *Pattern recognition and machine learning.* springer, 2006.

[12] B. Apolloni, A. Ghosh, F. Alpaslan, and S. Patnaik, *Machine learning and robot perception.* Springer Science & Business Media, 2005, vol. 7.

[13] G. Shobha and S. Rangaswamy, “Chapter 8 - machine learning,” in *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications*, ser. Handbook of Statistics, V. N. Gudivada and C. Rao, Eds., vol. 38, Elsevier, 2018, pp. 197–228. DOI: `10.1016/bs.host.2018.07.004`.

[14] I. El Naqa and M. J. Murphy, “What is machine learning?” In *Machine Learning in Radiation Oncology: Theory and Applications*, I. El Naqa, R. Li, and M. J. Murphy, Eds. Cham: Springer International Publishing, 2015, pp. 3–11. DOI: 10.1007/978-3-319-18305-3_1.

[15] M. Puig-Arnavat and J. C. Bruno, “Chapter 5 - artificial neural networks for thermochemical conversion of biomass,” in *Recent Advances in Thermo-Chemical Conversion of Biomass*, A. Pandey, T. Bhaskar, M. Stöcker, and R. K. Sukumaran, Eds., Boston: Elsevier, 2015, pp. 133–156. DOI: 10.1016/B978-0-444-63289-0.00005-3.

[16] I. del Campo, M. V. Martinez, J. Echanobe, E. Asua, R. Finker, and K. Basterretxea, “A versatile hardware/software platform for personalized driver assistance based on online sequential extreme learning machines,” *Neural Computing and Applications*, vol. 31, Dec. 2019. DOI: 10.1007/s00521-019-04386-4.

[17] O. Ertugrul and Y. Kaya, “A detailed analysis on extreme learning machine and novel approaches based on elm,” *American Journal of Computer Science and Engineering*, vol. 1, pp. 43–50, Dec. 2014.

[18] V. A. Chandrasetty and S. M. Aziz, “Chapter 6 - hardware implementation of ldpc decoders,” in *Resource Efficient LDPC Decoders*, V. A. Chandrasetty and S. M. Aziz, Eds., Academic Press, 2018, pp. 69–104. DOI: 10.1016/B978-0-12-811255-7.00006-X.

[19] J. Tiete, F. Domínguez, B. da Silva, A. Touhafi, and K. Steenhaut, “8 - mems microphones for wireless applications,” in *Wireless MEMS Networks and Applications*, ser. Woodhead Publishing Series in Electronic and Optical Materials, D. Uttamchandani, Ed., Woodhead Publishing, 2017, pp. 177–195. DOI: 10.1016/B978-0-08-100449-4.00008-7.

[20] P. Wilson, “Chapter 6 - digital circuits,” in *The Circuit Designer’s Companion (Fourth Edition)*, P. Wilson, Ed., Fourth Edition, Newnes, 2017, pp. 259–320. DOI: 10.1016/B978-0-08-101764-7.00006-2.

[21] D. Mesquita, F. Moraes, J. Palma, L. Möller, and N. Calazans, “Remote and partial reconfiguration of fpgas: Tools and trends,” vol. 0, Apr. 2003, 177a.

[22] Xilinx, *Xilinx’s website*, (Accessed June 16, 2021). [Online]. Available: https://www.xilinx.com/.

[23] Xilinx, *7-series product selection guide*, (Accessed June 16, 2021). [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf.

[24] Xilinx, *7-series architecture overview*, (Accessed June 16, 2021). [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html?resultsTablePreSelect=xlnxdocumenttypes:SeeAll#trainingSupport.

[25] Digilent, *Nexys4 ddr™ fpga board reference manual*, (Accessed June 16, 2021). [Online]. Available: https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys4DDR/documentation/Nexys4-DDR_rm.pdf.

[26] Xilinx, *Vivado design suite*, (Accessed June 16, 2021). [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[27] Intel, *Intel quartus prime software suite*, (Accessed June 16, 2021). [Online]. Available: https://www.intel.es/content/www/es/es/software/programmable/quartus-prime/overview.html.

[28] G.-B. Huang, *Extreme learning machines*, (Accessed June 16, 2021). [Online]. Available: `https://personal.ntu.edu.sg/egbhuang/elm_codes.html`.

[29] D. Dua and C. Graff, *UCI machine learning repository*, (Accessed June 16, 2021). [Online]. Available: `https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)`.

[30] C. Gianoglio, F. Guastavino, E. Ragusa, A. Bruzzone, and E. Torello, "Hardware friendly neural network for the pd classification," Oct. 2018, pp. 538–541. DOI: `10.1109/CEIDP.2018.8544825`.

[31] E. Ragusa, C. Gianoglio, P. Gastaldo, and R. Zunino, "A digital implementation of extreme learning machines for resource-constrained devices," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 8, pp. 1104–1108, 2018. DOI: `10.1109/TCSII.2018.2806085`.

[32] A. Armato, L. Fanucci, E. Scilingo, and D. De Rossi, "Low-error digital hardware implementation of artificial neuron activation functions and their derivative," *Microprocessors and Microsystems*, vol. 35, no. 6, pp. 557–567, 2011. DOI: `10.1016/j.micpro.2011.05.007`.

[33] I. del Campo, R. Finker, J. Echanobe, and K. Basterretxea, "Controlled accuracy approximation of sigmoid function for efficient fpga-based implementation of artificial neurons," *Electronics Letters*, vol. 49, pp. 1598–1600, Dec. 2013. DOI: `10.1049/el.2013.3098`.

[34] T. Adão, J. Hruška, L. Pádua, J. Bessa, E. Peres, R. Morais, and J. Sousa, "Hyperspectral imaging: A review on uav-based sensors, data processing and applications for agriculture and forestry," *Remote Sensing*, vol. 9, no. 11, p. 1110, 2017. DOI: `10.3390/rs9111110`.

[35] G. ElMasry and D.-W. Sun, "Chapter 1 - principles of hyperspectral imaging technology," in *Hyperspectral Imaging for Food Quality Analysis and Control*, D.-W. Sun, Ed., San Diego: Academic Press, 2010, pp. 3–43. DOI: `10.1016/B978-0-12-374753-2.10001-2`.

[36] R. Sahoo, S. Ray, and M. R, "Hyperspectral remote sensing of agriculture," *Current science*, vol. 108, pp. 848–859, Mar. 2015.

[37] D. Dua and C. Graff, *UCI machine learning repository*, (Accessed June 16, 2021). [Online]. Available: `https://archive.ics.uci.edu/ml/datasets/Statlog+(Landsat+Satellite)`.

[38] NASA, *Landsat program history*, (Accessed June 16, 2021). [Online]. Available: `https://landsat.gsfc.nasa.gov/about/history`.

[39] USGS.gov, *Landsat satellite missions*, (Accessed June 16, 2021). [Online]. Available: `https://www.usgs.gov/core-science-systems/nli/landsat/landsat-satellite-missions?qt-science_support_page_related_con=0#qt-science_support_page_related_con`.

[40] Xilinx, *7-series architecture data sheet*, (Accessed June 17, 2021). [Online]. Available: `https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`.

# Appendix

## A

```vhdl
--------------------------------------------------------------------------------
                    -- Module Name: Hidden Neuron Block --
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity Neuron is
    Port ( reset:in STD_LOGIC;
           clk :in STD_LOGIC;
           resetSumWX : in STD_LOGIC;                       -- Will sum the bias when low
           X : in STD_LOGIC_VECTOR (15 downto 0);           -- Input
           InputW : in STD_LOGIC_VECTOR (15 downto 0);      -- Random weight
           bias_b : in STD_LOGIC_VECTOR (20 downto 0);      -- Bias b
           neuron_out : out STD_LOGIC_VECTOR (20 downto 0));-- Signed 5.16 Format
end Neuron;

architecture Behavioral of Neuron is

COMPONENT MultWX_neuron
    Port ( X : in STD_LOGIC_VECTOR (15 downto 0);
           InputW: in STD_LOGIC_VECTOR(15 downto 0);
           mult_out : out STD_LOGIC_VECTOR (20 downto 0)
           );
end COMPONENT;

COMPONENT FeedbackSum
  Port (  reset:in STD_LOGIC;
          clk :in STD_LOGIC;
          sel :in STD_LOGIC;
          In0: in STD_LOGIC_VECTOR(20 downto 0);
          bias: in STD_LOGIC_VECTOR(20 downto 0);
          Out0: out STD_LOGIC_VECTOR(20 downto 0)
        );
end COMPONENT;

signal OutMultWX   : STD_LOGIC_VECTOR( 20 downto 0);

begin

 MultWX_neuron_1:MultWX_neuron PORT MAP (
           X => X,
           InputW => InputW,
           mult_out => OutMultWX
           );

 FeedbackSum_1: FeedbackSum  PORT MAP (
         reset=> reset,
         clk => clk,
         sel => resetSumWX,
         bias => bias_b,
         In0 => OutMultWX,
         Out0=> neuron_out );

end Behavioral;
```

## B

```
-------------------------------------------------------------------------------
                    -- Module Name: Neuron Block Multiplier --
-------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity MultWX_neuron is
Port ( X : in STD_LOGIC_VECTOR (15 downto 0);          -- Unsigned 0.16 Format
       InputW: in STD_LOGIC_VECTOR(15 downto 0);       -- Signed 1.15 Format
       mult_out : out STD_LOGIC_VECTOR (20 downto 0)); -- Signed 1.15 Format
end MultWX_neuron;

architecture Behavioral of MultWX_neuron is

signal X_aux: STD_LOGIC_VECTOR(15 downto 0);
signal out_mult: STD_LOGIC_VECTOR(31 downto 0);
begin

-- The signed multiplication is allowed by appending a sign bit '0' to the 15 most signifi-
-- cative bits of the input X. 1 bit of precision is lost.
X_aux <= '0' & X(15 downto 1);

-- The result of the multiplication will have a signed 2.30 format, by multipliying two sig-
-- ned 1.15 numbers. Therefore, just the least significative sign bit is important (it is
-- the one that sets if the result is positive or not). In addition, just the 20 most signi-
-- ficative fractional bits will be taken, loosing some precision.
out_mult <= InputW * X_aux;
mult_out <= out_mult(30 downto 10);

end Behavioral;
```

# C

```
---------------------------------------------------------------------------------
                  -- Module Name: Neuron Block Accumulator --
---------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity FeedbackSum is
  Port (  reset:in STD_LOGIC;
          clk :in STD_LOGIC;
          In0: in STD_LOGIC_VECTOR(20 downto 0);  -- Output of the multiplier
          bias: in STD_LOGIC_VECTOR(20 downto 0); -- Bias b
          sel: in STD_LOGIC;                      -- Selector between the bias and the accumulated value
          Out0: out STD_LOGIC_VECTOR(20 downto 0) -- Signed 5.16 Format
        );
end FeedbackSum;

architecture Behavioral of FeedbackSum is

signal registerAccumulatorOld: STD_LOGIC_VECTOR  (20 downto 0):=(others =>'0');
signal outMux:  STD_LOGIC_VECTOR (20 downto 0):=(others =>'0');
signal registerAccumulatorNew:  STD_LOGIC_VECTOR(20 downto 0):=(others =>'0');
signal sign_fill_In0: STD_LOGIC_vector(4 downto 0);
signal sign_fill_bias: STD_LOGIC_vector(4 downto 0);
signal bias_format: STD_LOGIC_VECTOR(20 downto 0);
signal In0_format: STD_LOGIC_VECTOR(20 downto 0);
begin

-- The dynamic range of the accumulating operation will be of 4 integer bits, together with the sign bit
-- and the 16 fractional bits. With this range, overflow can be avoided in the example problem.
-- NOTE: If the target application changes this dynamic range should be reviewed.

-- The input vectors have to be restructured to the dynamic range before computing any operation.
-- This is done with a sign extension.
sign_fill_bias <= (others => bias(20));
bias_format <= sign_fill_bias & bias (19 downto 4);

sign_fill_In0 <= (others => In0(20));
In0_format <= sign_fill_In0 & In0 (19 downto 4);

with sel select
outMux <= registerAccumulatorOld when '1',
          bias_format when others;
-- outMux will be in the desired format

registerAccumulatorNew <=  In0_format +  outMux;

process(reset, clk)
begin
if(reset = '0') then
        registerAccumulatorOld <= (others =>'0');
else
if( clk='1' AND clk'EVENT ) then
        registerAccumulatorOld<=registerAccumulatorNew;
    end if;
end if;
end process;

Out0<=registerAccumulatorOld;

end Behavioral;
```

## D

```
---------------------------------------------------------------------------
        -- Module Name: Global Package for the Sigmoid Module --
---------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
package Globals is
   constant int_sel_len: integer := 10; -- Defines the length of the selection vector.
   constant int_out_len: integer := 16; -- Defines the length of the output.
end package;
```

```
------------------------------------------------------------------------------------------------------------------------
                              -- Module Name: Sigmoid Activation Function Block --
------------------------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.Globals.all;

entity Sigmoid is
    Port (sel : in  STD_LOGIC_VECTOR (20 downto 0);    -- Output of the hidden neuron (Signed 5.16 Format)
          range_sig: in STD_LOGIC_VECTOR (4 downto 0); -- Sampling range of the sigmoid
          clk: in STD_LOGIC;
          out0 : out  STD_LOGIC_VECTOR (15 downto 0)); -- Unsigned 0.16 Format
end Sigmoid;

architecture Behavioral of Sigmoid is

signal aux: STD_LOGIC_VECTOR(int_sel_len-1 downto 0);
signal sign : STD_LOGIC;
signal neg: STD_LOGIC_VECTOR(20 downto 0);
signal range_aux: STD_LOGIC_VECTOR (2 downto 0);
signal one : STD_LOGIC_VECTOR (int_out_len-1 downto 0):=(others=>'1');

-- The matrix containing the sampled values is filled, its number of elements will be determined by the length of the
-- selection vector (int_sel_len).
type memory is array (0 to 2**int_sel_len-1) of STD_LOGIC_VECTOR (int_out_len-1 downto 0);
constant val_sig: memory:= ("1000000000000000",
"1000000010000000",
"1000000100000000",




"1111111111101001",
"1111111111101010");

begin

-- The complemented input is calculated, it will be used to compute the result of a negative input.
sign <= sel(20);
neg <= not(sel) + "000000000000000000001"; -- Two's complement

-- The sampling range will determine the amount of integer bits of the input that will be taken.
with range_sig select
range_aux <= "001" when "00010", --Precision 1*10^-1
             "010" when "00100", --Precision 1*10^-2
             "011" when "01000", --Precision 1*10^-3
             "100" when "10000", --Precision 1*10^-4 --> Maximum precision with a signed 5.16 format.
             "000" when others;

-- The bits of the selection vector are chosen, they are determined by the length of that vector (int_sel_len).
with sign select
aux <= sel(15 + to_integer(unsigned(range_aux)) downto 16 + to_integer(unsigned(range_aux)) - int_sel_len) when '0',
       neg (15 + to_integer(unsigned(range_aux)) downto 16 + to_integer(unsigned(range_aux)) - int_sel_len) when others;

-- The output is obtained from the matrix.
with sign select
out0 <= val_sig(to_integer(unsigned(aux))) when '0',
        one - val_sig(to_integer(unsigned(aux))) when others; -- f(-x) = 1 - f(x)

end Behavioral;
```

## E

```
------------------------------------------------------------------------------
                   -- Module Name: Output Comparator --
------------------------------------------------------------------------------
-- This module receives two inputs and returns the highest one. It can be used  to implement
-- a selection binary tree.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Out_comparator is
  Port ( int1 : in STD_LOGIC_VECTOR(20 downto 0);
         int2 : in STD_LOGIC_VECTOR(20 downto 0);
         clk : in STD_LOGIC;
         max_output : out STD_LOGIC_VECTOR(20 downto 0); -- Highest input
         ind_out : out STD_LOGIC);                       -- Index of the highest input
end Out_comparator;

architecture Behavioral of Out_comparator is

signal aux: STD_LOGIC;
begin

process (clk) -- A process is done to match these calculations with the clock edges.
begin

-- The index of the highest input is determined.
if (int1 > int2) then
    aux <= '0';
else
    aux <= '1';
end if;
ind_out <= aux;
end process;

-- The highest input is selected using the index previously determined.
with aux select
max_output <= int1 when '0',
              int2 when others;

end Behavioral;
```

# F

```
--------------------------------------------------------------------------------
        -- Module Name: Parallel Architecture Top Module --
--------------------------------------------------------------------------------
```

```
-- Vector matrices are generated for the outputs of each hidden neuron and sigmoid block.
type n_out is array (0 to 9) of STD_LOGIC_VECTOR (20 downto 0);
signal neu_out: n_out;

type sigmoide_out is array (0 to 9) of STD_LOGIC_VECTOR (15 downto 0);
signal sig_out: sigmoide_out;
signal w_mux: sigmoide_out;

signal int_sig: STD_LOGIC_VECTOR (15 downto 0);
signal int_beta1: STD_LOGIC_VECTOR (15 downto 0);
signal int_beta2: STD_LOGIC_VECTOR (15 downto 0);
signal out1: STD_LOGIC_VECTOR (20 downto 0);
signal out2: STD_LOGIC_VECTOR (20 downto 0);
signal max_out: STD_LOGIC_VECTOR(20 downto 0);

begin

-- The random weights of each hidden neuron are picked from a general memory.
mux_gen: for i in 0 to 9 generate
    w_mux(i) <= pesos_InputW(i*9 + to_integer(unsigned(sel_in)));
end generate;

-- The hidden neurons are generated. In this example L = 10.
gen_neu: for i in 0 to 9 generate
    neu: Neuron Port Map( reset => res,
                          clk => clk,
                          resetSumWX => resetSumWX,
                          X => X,
                          W => w_mux(i),
                          bias_b => pesos_b(i),
                          neu_out => neu_out(i)  );
end generate;

-- The sigmoid blocks for each hidden neuron are generated.
gen_sig: for i in 0 to 9 generate
    sig: Sigmoide Port Map(sel_sig => neu_out(i),
                           rango => rango_sig,
                           clk => clk,
                           out_sig => sig_out(i) );
end generate;

-- The Beta weights and the output of each sigmoid block are selected sequentially.
int_sig <= sig_out(to_integer(unsigned(sel_neu)));
int_beta1 <= pesos_Beta(to_integer(unsigned(sel_neu)));
int_beta2 <= pesos_Beta2(to_integer(unsigned(sel_neu)));

Neuron_out_1: Neuron_out Port Map(reset => res_out,
                                  clk => clk,
                                  enable => enable,
                                  Sig => int_sig,
                                  Beta => int_beta1,
                                  output => out1   );

Neuron_out_2: Neuron_out Port Map(reset => res_out,
                                  clk => clk,
                                  enable => enable,
                                  Sig => int_sig,
                                  Beta => int_beta2,
                                  output => out2   );

-- The binary comparator module is used to obtain the highest result.
Out_comparator_1: Out_comparator Port Map(int1 => out1,
                                          int2 => out2,
                                          clk => clk,
                                          max_output => max_out,
                                          ind_out => output  );

end Behavioral;
```

**G**

```
--------------------------------------------------------------------------------------------
            -- Module Name: Shifter of the modified Hidden Neuron Block --
--------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ShifterK is
    Port ( X : in STD_LOGIC_VECTOR (15 downto 0);        -- Inputs of the hidden neuron block
           K: in STD_LOGIC_VECTOR(7 downto 0);           -- Output of the k memory
           Out0 : out STD_LOGIC_VECTOR (20 downto 0));   -- Unsigned 5.16 Format
end ShifterK;

architecture Behavioral  of ShifterK is

signal out0_hid : STD_LOGIC_VECTOR (7 downto 0);
begin

   -- The input vector X is shifted K positions to the right, the number is divided by a power of two with an exponent k.
   -- Then, it is resized to a 21 bit vector, filling the new positions and the shifted bits with '0's. Therefore, the
   -- result will be unsigned, as well as the original X input.
   Out0 <= std_logic_vector(resize(unsigned(std_logic_vector(shift_right(unsigned(X), to_integer(unsigned(K))))),21));

end Behavioral ;
```

```
--------------------------------------------------------------------------------
             -- Module Name: Not module of the modified Hidden Neuron Block --
--------------------------------------------------------------------------------

-- This module complements the output of the shifter.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity NotMux is
    Port ( X : in STD_LOGIC_VECTOR (20 downto 0);     -- Output of the shift module
           Out0 : out STD_LOGIC_VECTOR (20 downto 0) -- Signed 5.16 Format
          );
end NotMux;

architecture Behavioral of NotMux is

signal notX: STD_LOGIC_VECTOR (20 downto 0);
begin

Out0<= not(X);
end Behavioral;
```

```
--------------------------------------------------------------------------------
             -- Module Name: Multiplexer of the modified Hidden Neuron Block --
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux is
    Port ( In0 : in STD_LOGIC_VECTOR (20 downto 0);     -- Output of the Not module
           In1 : in STD_LOGIC_VECTOR (20 downto 0);     -- Output of the shifter
           Sel : in STD_LOGIC;                           -- Sign s, it comes from a memory
           Out0 : out STD_LOGIC_VECTOR (20 downto 0)); -- Signed 5.16 Format
end Mux;

architecture Behavioral of Mux is
begin

-- If the sign s is '0' the shifter output, without being complemented, is selected.
Out0<= (In1) when Sel='0' else In0;

end Behavioral;
```

# H

```
---------------------------------------------------------------------------------------
      -- Module Name: Not module of the modified Output Neuron Block --
---------------------------------------------------------------------------------------

-- This module complements the Beta weights of the output neuron.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity NotAdderOneW is
    Port ( X : in STD_LOGIC_VECTOR (15 downto 0);    -- Beta weights (Signed 1.15 Format)
           Out0 : out STD_LOGIC_VECTOR (15 downto 0) -- Complemented Beta weights (Same format)
          );
end NotAdderOneW;

architecture Behavioral of NotAdderOneW is

signal notX: STD_LOGIC_VECTOR (15 downto 0);
begin

-- The two's complement of the Beta weights is computed.
notX<= not(X);
Out0 <= std_logic_vector(to_signed(to_integer(signed( notX )) + 1, 16)); --Complemento a 2

end Behavioral;
```

```
----------------------------------------------------------------------------------------------------
                -- Module Name: Multiplexer of the modified Output Neuron Block --
----------------------------------------------------------------------------------------------------

-- This module selects between the Beta weights and its complement, then, the result is resized.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MuxResizeW is
  Port ( In1: in STD_LOGIC_VECTOR(15 downto 0);      -- Beta weights
         In1Neg : in STD_LOGIC_VECTOR(15 downto 0); -- Complemented Beta weights
         sel : in STD_LOGIC;                         -- Sign bit of the output of the hidden neuron
         Out0 : out STD_LOGIC_VECTOR(20 downto 0)    -- Signed 6.15 Format
        );
end MuxResizeW;

architecture Behavioral of MuxResizeW is

signal IXPlus :  STD_LOGIC_VECTOR(15 downto 0);
begin

-- The beta weights or their complements are selected depending on the sign bit of the output of the
-- hidden neuron block.
IXPlus<= In1 when Sel='0' else In1Neg;

-- The selected weight is resized to a 21 bit vector in a signed 6.15 format.
Out0 <= std_logic_vector(resize(signed(IXPlus), 21));

end Behavioral;
```