UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

DEPARTMENT OF ELECTRONIC ELECTRIC AND COMPUTER ENGINEERING (EECE)

EES 424: RESEARCH PROJECT

EXAMINATION REPORT: IMPLEMENTATION OF SIGMOID ACTIVATION FUNCTION FOR NEURAL NETWORKS ON A FIELD PROGRAMMABLE GATE ARRAY DEVICE.

Name and Surname	Student Number	Signature	% Contribution
M. H. Khan	14346011	[illegible]	100

November 11, 2020

ABSTRACT / SUMMARY

This research project aims to identify a method that can be implemented on an FPGA board to generate a sigmoid activation function. The purpose of these functions is used specifically in the neural network domain. The sigmoid added to neural networks to aid in the learning of complex data patterns. A low-level model is designed in Python using various methods of piecewise functions to approximate the sigmoid as close to an ideal sigmoid as possible. The success of this determines the basis for the low-level implementation using VHDL on an FPGA board. Verification is conducted by transmitting serial data or data points of the sigmoid back to a computer to analyse the results of the two models implemented. The informal verification process is conducted to visually see the shape of the sigmoid. This is done to ensure that indeed the actual shape form of the sigmoid is being plotted. Secondly, formal verification is conducted which test the quality of how successful the sigmoid has been implementing. The metrics used for this include the mean, standard deviation and variance of the error as well as the resource usage between the two models that are implemented. The conclusion drawn from this was that both models are successful. However, model 1 (Base model) requires fewer resources but results in an error of approximately 2.5%. Whilst the improved model 2 (Research model) achieves an error below 0.5%. However, this comes at the cost of approximately double the processing time as well as doubling the resource usage.

PART A:
LITERATURE REVIEW

Submitted as ‘Assignment 1’ in partial fulfillment of the requirements of ECSA Graduate Attribute 4 for the module EES 424 - Research Project.

By submitting this section of the document, I confirm that

- the content in Part A is the latest / final evaluated version of *Assignment 1* that was uploaded to the AMS system,
- the content has not been changed to such an extent that it could be interpreted to mean something different from that as my final evaluated submission.
- the various revision control colour highlighting (changes highlighted in yellow represent a resubmission and changes highlighted in orange/red represent a further special intervention submission) have been untouched and remain intact.

I. Introduction

With the turn of the 21st century artificial intelligence (AI) and artificial neural networks (ANN) are becoming quickly an integral part in our everyday lives from Tesla's self-driving cars to simple smart home IOT appliance. With this surge come along the need for rapid prototyping AI and ANN. One of the core components of these systems is the activation which is represented by the sigmoid function. ANNs form part of a filed in AI that designed to model the behaviour of the human brain which raises the concept of deep learning [1], as well as the multi-layer perception these include but are not limited to speech recognition and image identification and calcification [2]. These systems can be implemented on a software level however research has shown software based neural networks suffer from slower execution time compare to there hardware level counterparts in real time applications [1].

One of the key challenges engineers face when synthesizing artificial neural network based on FPGA is the approximation of the activation function which in this case is the sigmoind as sated prior [3]. There have been multiple studies with regards to the optimum implementation of the sigmoid function. The A-Law approach [4] is a pairwise linear approximation used to implement a pulse code modulation which the authors suggest can be implemented with a using a small number of logic gates. An other approach to the activation is proposed by C.Alippi approximation is base on selecting an integer set breakpoints and setting the y-values of these integers as the power of 2 integers [5][6]. The piecewise linear approximation of a nonlinear function (PLAN) approximation proposed by Amin, Curtis and Hayes–Gill. deploys digital gates to directly transform values from x to y [7]. This approximation uses the x as the in put an y the approximated sigmoidal output.

A piecewise second order approximation was proposed to to approximate the sigmoid function, with this method comes greater performance and lower implementation cost. The method require one multiplication and no lookup table as suggested by the author [8]. The second order approximation scheme substantially improves approximation of higher order piece wise functions. The scheme requires no memory and has significantly low delay and error rate.

This literature study will investigate the methods to implement these methods on a FPGA as well as take a look it the advantages and trade offs of each of the motioned approaches, as well the logical element cost of each of the approaches.

II. Literature Review

Sigmoid activation functions are used in feed forward neutral networks. The sigmoid function is monotonic growing and differentiated.[9]. The sigmoidal function is defined by Eq.1

$$f(x,k,b,T,c) = k + \frac{c}{1+be^{Tx}}, \forall x \in R. \tag{1}$$

$$Where\ k \in R, b \in R^{+}, T, c \in R\backslash\{0\}$$

classical sigmoid function ($k = 0, c = b = 1, T = 1$):

$$f(x) = \frac{1}{1+e^{-x}}. \tag{2}$$

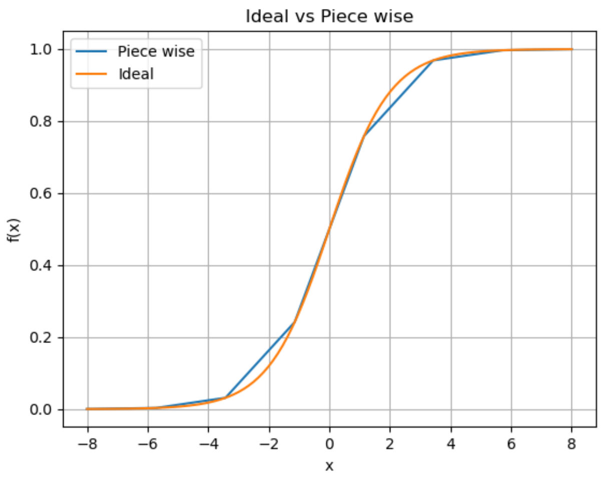


Fig. 1: The graphical representation of Eq.1

A. A-Law Approach

With this proposed method of implementation to achieve the modified sigmoid function each linear segment is implied to be the power of 2 this new curve has only seven segments[3]. Splitting the curve this way allows to replace multipliers with shift registers. The fig.2 below is the hardware inflammation proposed by [10]. It can be seen in fig.2 require 4 shift registers, 2 multiplexers and a single sum block. The MCode block is for compare functions. It is also stated by the author that ”All the used blocks are part of the System Generator library Xilinx Integrated Software Environment”. The results obtained by [10] where as follows, the maximum error introduced by A-law approximation was found be 5.63% and the mean error is 0.6335%.

TABLE I: Break points of the A LAW approximation

x	-8.0	-4.0	-2.0	-1.0	1.0	2.0	4.0	8.0
y	0.0	0.0625	0.12	0.25	0.75	0.87	0.937	1.0

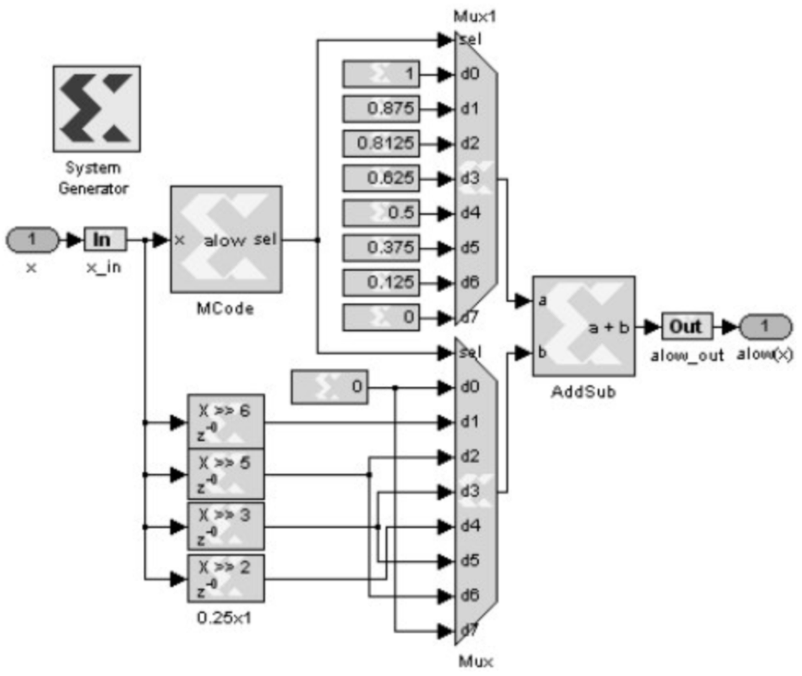


Fig. 2: Hardware architecture of the A Law approximation[10]

B. Alippi and Storti-Gajani model

This approximation of this scheme depends on selecting the set of breakpoints of the first derivative and setting up the function as the sums of powers similar to the A-Law approach. Hence the sigmoid function will be symmetrical at point $x = 0, y = 0.5$, therefore only half the pair need to be determined thanks to the symmetrical nature[5].

$$y_{x>0} = 1 - y_{x\leq 0}. \tag{3}$$

Considering only the negative half of the function on the negative x-axis we can define the integer part of x, n[6] can be written as the following function:

$$n = |(x)| + 1. \tag{4}$$

The decimal part of x can express as the equation below

$$\hat{x} = x + |(x)|. \tag{5}$$

The C.Alippi and G.Storti-Gajani scheme can be implemented with 8 shift registers for dividing $\hat{x}$ into 4 signals for shifting with the number of x bits. 4 Multiplexers, subtraction blocks, 2 slice blocks and 1 cooperator block[10] this implementation can be seen in fig.3. The author stated ”The numerical representation used for the analysis is 32 bits with 16 bits binary point representation induces a maximum of 1.89 %” and a mean error of 1.11%[10]

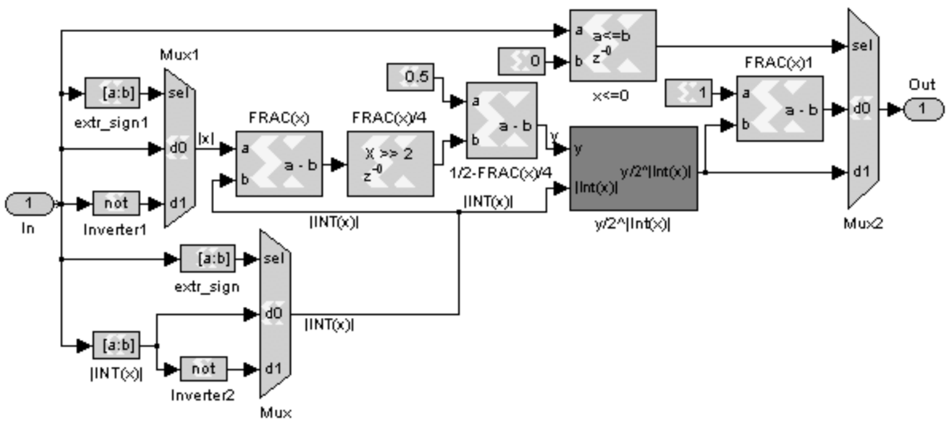


Fig. 3: Hardware architecture of the Alippi function approximation[10]

It should made know to the reader in the above figure $\hat{x} = FRAC(X)$ and $|x| = |INT(x)|$

C. Piecewise Linear Approximation of a Nonlinear function (PLAN)

The PLAN is a efficient piecewise approximation that can be implemented using digital gate to transform from X to Y. This approximation is used in the outputs of neural networks to perform approximation[7]. Due to the nature of the PLAN calculation are only need to be performed on the absolute value of the input x[5]. The one main advantage of the scheme is that instead of multiplication we use shift operation[10]. This due to the PLAN efficiently choosing the gradient of the line to eliminate the need for multipliers[7]. The representation induces a maximum error of 1.89 % and a mean error of 0.63%.

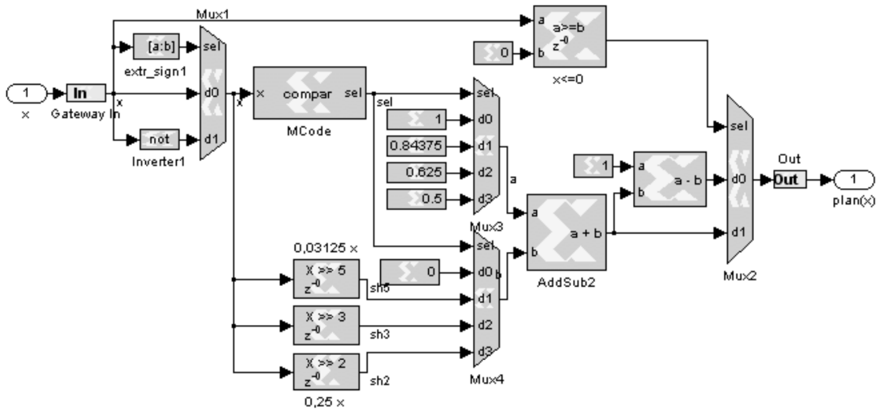


Fig. 4: Hardware architecture of the PLAN function approximation[10]

Piecewise second-order approximation

The sigmoid can also be realised with a oiece wise second order approximation, the follow is formula for this approximation[8].

$$y(x) = c_0 + c_1 * x + c_2 * x^2. \quad (6)$$

When we take a closer look at the above function we can see that on clear drawback the need for multiplication[5]. Zhang suggests scheme requiring one multiplier, in the range [-4,4] that sigmoid is computes as follows:

$$y = \begin{cases} 2^{-1} * \left(1 - \left|2^{-2} * x\right|^2\right) & -4 < x < 0 \\ 1 - 2^{-1} * \left(1 - \left|2^{-2} * x\right|^2\right) & 0 \leq x < 4 \end{cases} \tag{7}$$

From simplification of the above piecewise function, it can be implemeted with one multiplier, two shift registers and two XOR gates this can be seen in fig.5. It has been reported in [10] that there are there way to implement the second order approximation however, these require upto tree time more hardware resources[8].

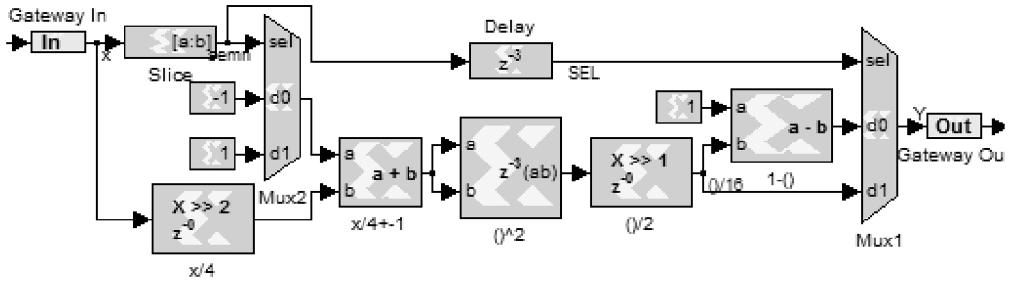


Fig. 5: Hardware architecture of the Zhang function approximation[10]

TABLE II: Erros and resources utilization of each method of approximation [10]

Approximation function	Maximum error (%)	Mean error (%)	Total equivalent gates count for design
A-law	5.63	0.63	411
Allipi	1.89	1.11	877
Plan	1.89	0.63	351
Zhang	2.16	1.10	314

Interpreting the results in the above table we can see that the Plan method has the lowest mean at a rate, followed by A Law which also has the same mean at a rate however has a significantly larger maximum average. Allipi and Plan have the same maximum and rate however plan has a lower mean at a rate and also requires less total gates for the design. Zhang falls in the middle of these results not having a terrible maximum at a rate and also not having good mean at a rate however, Zhang requires the least number of equivalent gates.If we disregard the maximum error percentage for A Law it presents the simplest hardware implementation method from the study. However, if the maximum error is of concern than the second-best implementation from simplicity standpoint would be Zhang approach.

PART B:

RESEARCH PROPOSAL

Submitted as ‘Assignment 2’ in partial fulfillment of the requirements of ECSA Graduate Attribute 4 for the module EES 424 - Research Project.

By submitting this section of the document, I confirm that

- the content in Part B is the latest / final evaluated version of *Assignment 2* that was uploaded to the AMS system,
- the content has not been changed to such an extent that it could be interpreted to mean something different from that as my final evaluated submission.
- the various revision control colour highlighting (changes highlighted in yellow represent a resubmission and changes highlighted in orange/red represent a further special intervention submission) have been untouched and remain intact.

III. Research Statements

A. Problem Statement

There is a wealth of knowledge pertaining to the implementation of a sigmoid activation function of a field programmable gate array (FPGA). These onboard application methods include, however, not limited to the A-Law method proposed by [4]. This proposal will focus on using a piece wise function to potentially implement a sigmoid function on the FPGA as performed by [4] on the Altera Cyclone V SoC FPGA.

B. Research Statements

1. Can the piece wise function be used to implement a sigmoid as an activation function on a FPGA?
2. Can the system be deployed with a maximum error rate below 10% and a mean error rate below 5%?

C. Research Model

1) Model Description: The figure below is the hardware architecture as proposed by [10].

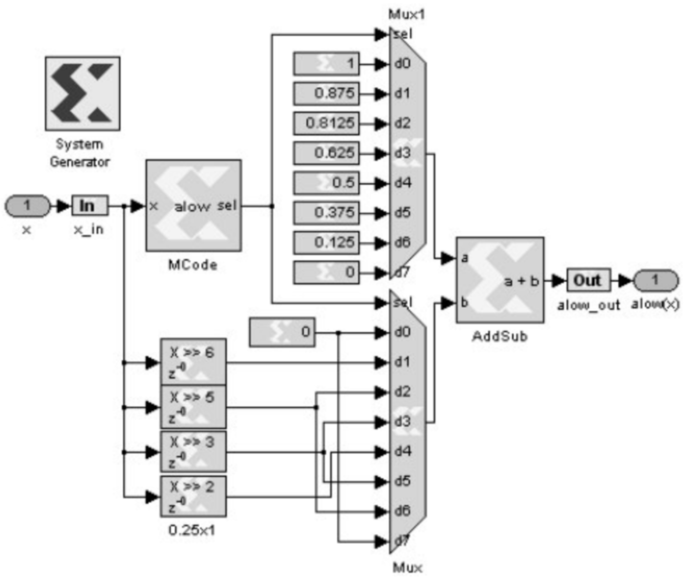


Fig. 6: Hardware architecture of the A Law approximation

The A-law method approximates the sigmoid function by taking the sigmoid function over the range of -8 to 8 and dividing it into 8 piecewise functions. By doing this the gradient of each of those points can be calculated and combined to approximate the sigmoid. However, this approximation will not result in a very closely matched function. An attempt will be made to increase the piecewise function from 8 data points to 16 data points in doing this the potential error should be reduced as the area between the ideal sigmoid and the approximated sigmoid will be reduced. The design will need to functional on the FGPA device supplied. From past experience it well know that working with decimal data can cause problems. One potential solution as suggested in [11] is to scale the decimal values to a large enough value to make the result an integer. This is a potention solution to the floating point values.

TABLE III: Break points of the A LAW approximation

x	-8.0	-4.0	-2.0	-1.0	1.0	2.0	4.0	8.0
y	0.0	0.0625	0.12	0.25	0.75	0.87	0.937	1.0

The table above contains the data points that the sigmoid was partitioned into. There are 8 points that were used to approximate the sigmoid in fig.7. The blue signal is the A-Law Python approximation and the orange signal is an ideal sigmoid with 100 data points.

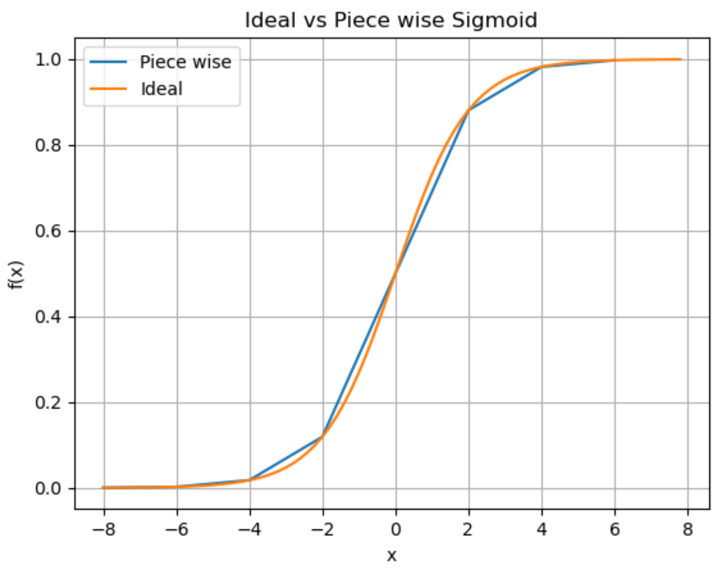


Fig. 7: Ideal sigmoid vs piece wise function of the sigmoid

In the figure above an ideal sigmoid is plotted on the same axis using the data points present in table.V.

One of the more critical areas for success would be the efficient use of the FPGA resources although the device utilisation is not yet know for the hardware the model will be implemented on, one can get a sense of idea looking at the implementation performed by [10]. Device usage was as follows in the table below. The higher the device utilisation the longer it would take the sofware to synthesise the design.

TABLE IV: Resource Allocation for Hardware Implementation of Sigmoid on 4VSX35 FPGA[10]

Slices	LUTs	RAMBs	DSPs	Total equivalent gates count	Max frequency (MHz)	Estimated power consumed
29	31	1	1	66	268.168	607.02

D. *Data collection*

A serial communication link will be established the system deployed and a suitable computer. The serial(RS232) data that has been read will be saved to an appropriate file format for analysis at a later point. The computer will control the number of data points that will need to be read. A code written in Python running on the computer, will send the transmission start and stop command to the system to perform the necessary data acquisition.

E. Analysis Of Acquisitioned Data

To validate the captured data it will need to be compared to set function such as eq.9. The reference curve which is represented by eq.9 will be plotted in Python. The collected data will also be plotted in Python on the same axis.

$$f(x) = \frac{1}{1+e^{-x}}. \tag{8}$$

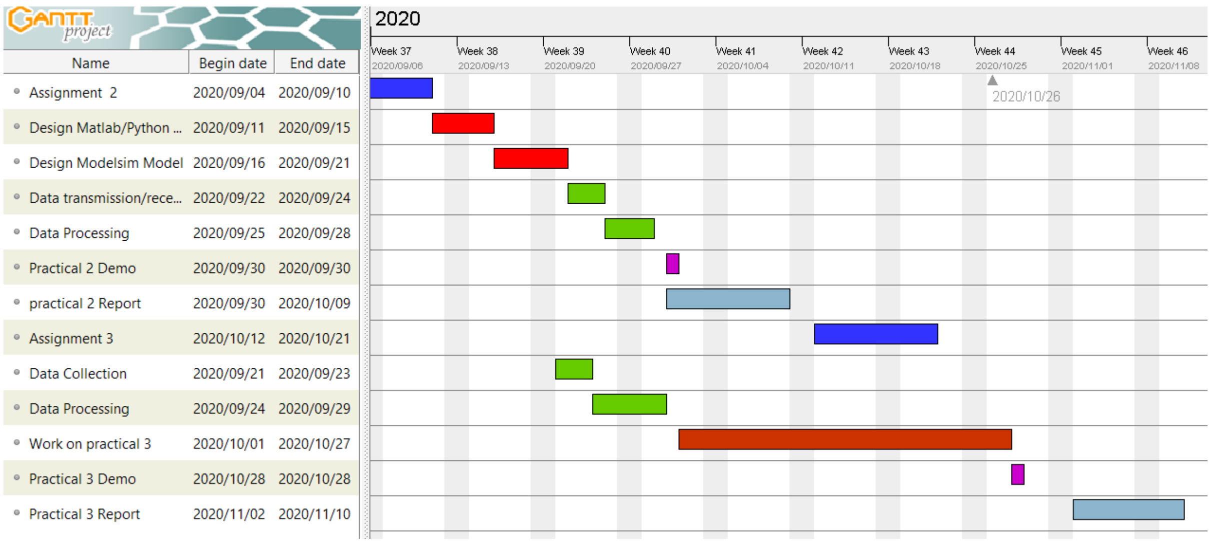
Having both figures plotted on the same axis to make a comparison and as well as perform the necessary adjustments on the piece wise function and ideal function. This will allow for the error between the two functions to be computed. This error will be the area between the actual ideal sigmoid and the generated model within Python and VHDL. This will need to considered for the entire range of all the sigmoid models. A successful implementation would mean absolute error of below 5% for the entire range of the research model as suggested by. [10]

F. Experimental Method

A high level system will be designed in Python to emulate the sigmoid function. This function will be of high accuracy using libraries in Python. The objective to attempted to minimise the error between the ideal sigmoid and the piece wise function. For this ideal representation, 100 points are used to simulate high accuracy while piecewise function is constructed by using 8 points. This figure simply a representation of how the model can be implemented. If the resolution of the system is increased ie: using more points such as 16 points. This would reduce the error between the ideal model and the simulate model. Once satisfied with the high level and low level Python models a low level VHDL model will be designed. A strategy needs to be devised to read float values from the FPGA moreover, how to process the floating point data.

G. Resource Planning

The figure below depicts a potential Gantt chart outlining the time line for the project life cycle.



Name	Begin date	End date
Assignment 2	2020/09/04	2020/09/10
Design Matlab/Python ...	2020/09/11	2020/09/15
Design Modelsim Model	2020/09/16	2020/09/21
Data transmission/rece...	2020/09/22	2020/09/24
Data Processing	2020/09/25	2020/09/28
Practical 2 Demo	2020/09/30	2020/09/30
practical 2 Report	2020/09/30	2020/10/09
Assignment 3	2020/10/12	2020/10/21
Data Collection	2020/09/21	2020/09/23
Data Processing	2020/09/24	2020/09/29
Work on practical 3	2020/10/01	2020/10/27
Practical 3 Demo	2020/10/28	2020/10/28
Practical 3 Report	2020/11/02	2020/11/10

Fig. 8: Gantt chart of the research project.

PART C:
RESEARCH METHODOLOGY

Submitted as 'Practical 2 Report' in partial fulfillment of the requirements of ECSA Exit Graduate Attribute 4 and 5 for the module EES 424 - Research Project.

By submitting this section of the document, I confirm that

- the content in Part C is the latest / final evaluated version of *Practical 2 Report* that was uploaded to the AMS system,
- the content has not been changed to such an extent that it could be interpreted to mean something different from that as my final evaluated submission.
- the various revision control colour highlighting (changes highlighted in yellow represent a resubmission and changes highlighted in orange/red represent a further special intervention submission) have been untouched and remain intact.

IV. Implementation Of The Research Model

The research model that has been implemented as a piecewise sigmoid activation function this function has been fully described in [12]. The base research model is made up of splitting a sigmoid activation function into seven equal parts from a range of -8 to 8 on the x-axis range and 0 to 1 y-axis. This function is described by the following equation:

$$f(x) = \frac{1}{1+e^{-x}}. \tag{9}$$

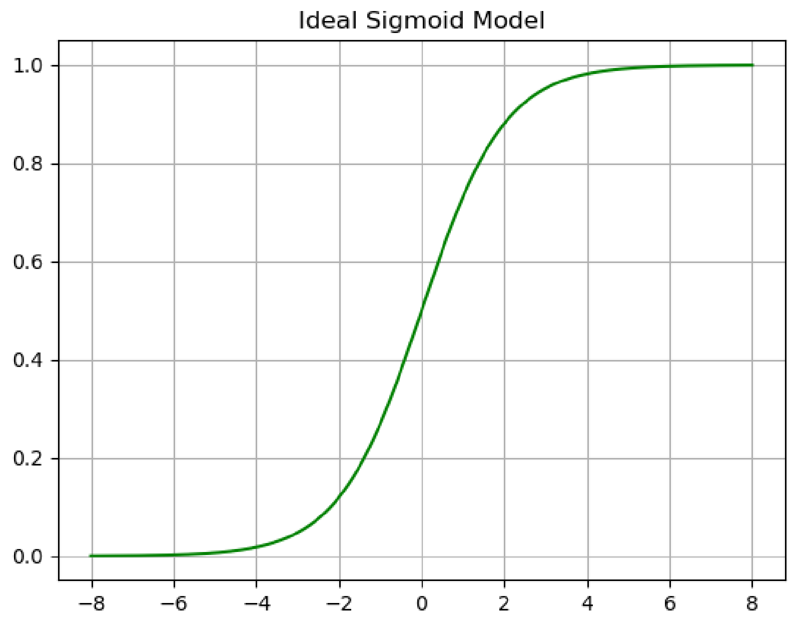


Fig. 9: Ideal sigmoid generated with eq.9

It was noticed that working with floating point values can cause significant amount of trouble. Therefore, all the values in the system were then scaled to very large integers by means of multiplying both axis by 2^{16}. Doing this eliminates the need for keeping account for the position of the decimal point when doing calculations. Once all that data has been captured the scaling values can be removed.

Model 1: Base research model.
Model 2: Improvement upon base research model.

A. *Generation of input data*

The input data ranges from $-524288 = 2^{16} * (-8)$ to $524288 = 2^{16} * (8)$ this results in a total of 1048576 data points that are split in to seven segments as described in in the table below.

TABLE V: Break points of the A LAW approximation[10]

x	-8.0	-4.0	-2.0	-1.0	1.0	2.0	4.0	8.0
y	0.0	0.0625	0.12	0.25	0.75	0.87	0.937	1.0

By taking a each value in the above table and scaling by 2^{16} will result in the data type that will be suitable for the model that has been implemented in VHDL. A variable x is initialised to -524288 and starts to count up to 524288.

B. State machine diagram of the implemented model

Figure.11 below represents the finite state machine diagram of the model that has been implemented in VHDL.

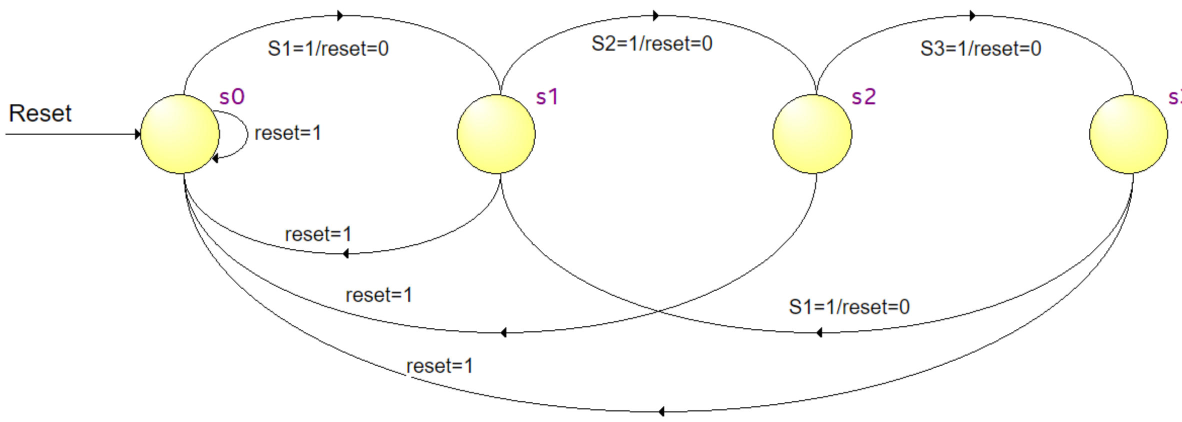
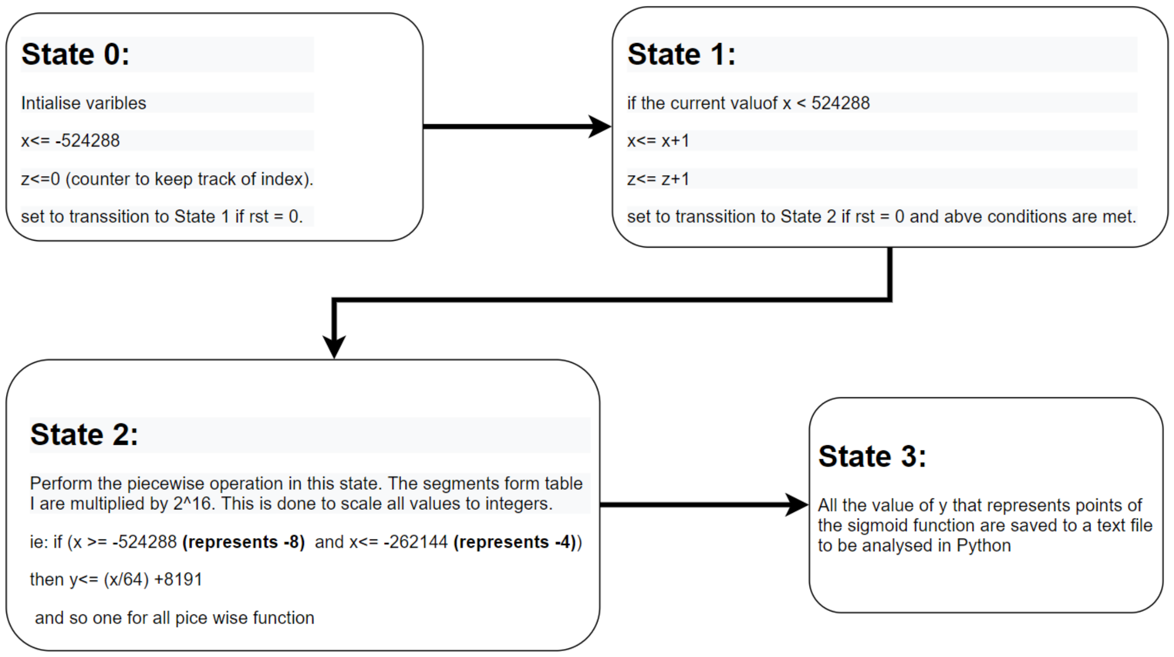


Fig. 10: Finite state machine model

The model consists of four states. These states are as follows:



State 0:

Intialise varibles

x<= -524288

z<=0 (counter to keep track of index).

set to transsition to State 1 if rst = 0.

State 1:

if the current valuof x < 524288

x<= x+1

z<= z+1

set to transsition to State 2 if rst = 0 and abve conditions are met.

State 2:

Perform the piecewise operation in this state. The segments form table I are multiplied by 2^16. This is done to scale all values to integers.

ie: if (x >= -524288 **(represents -8)** and x<= -262144 **(represents -4)**)

then y<= (x/64) +8191

and so one for all pice wise function

State 3:

All the value of y that represents points of the sigmoid function are saved to a text file to be analysed in Python

Fig. 11: Code flow diagram

The figure that follows is a plot in python of the data that has been saved to the text file in state 3 of the VHDL model. The plot below has a degree of resemblance to the result that has been obtained by [4].

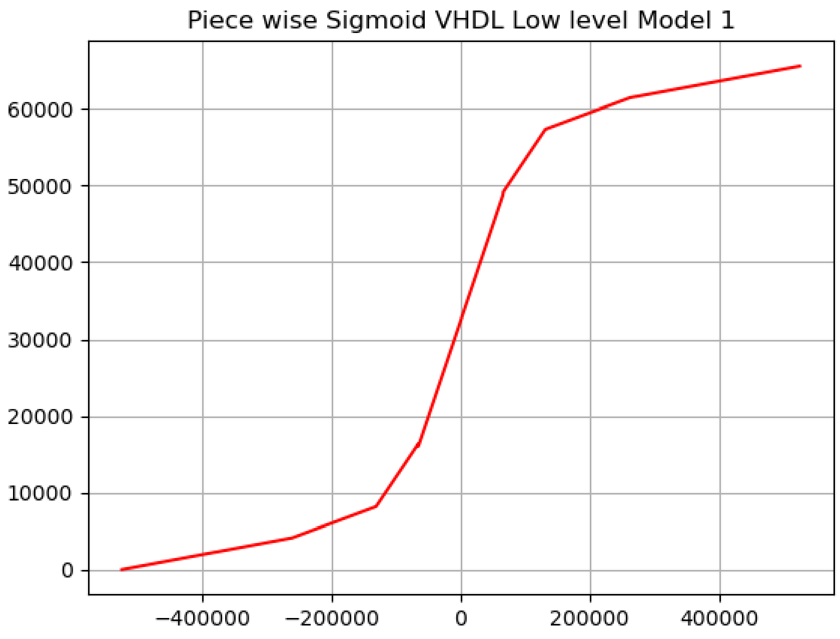
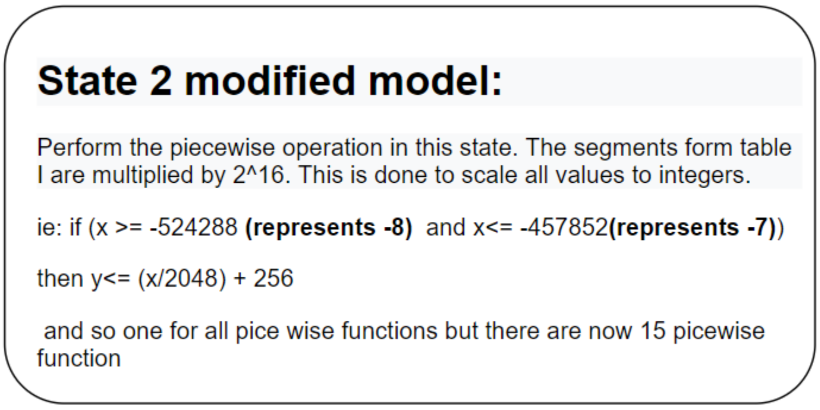


Fig. 12: Low level VHDL model 1.

In this figure the shape of the sigmoid is recognisable. However, this model can be improved to more closely resemble the sigmoid in fig.9. The section that follows will investigate this improved model implementation.

C. *Implementation of the improved model.*

The model was improved upon by increasing the number of piecewise function Steps from seven to fifteen. In doing this the line segments can be more closely represented by the gradient of the ideal sigmoid. Hence, the piecewise function looks a lot closer to that of the ideal sigmoid that was presented in fig.9.



State 2 modified model:

Perform the piecewise operation in this state. The segments form table I are multiplied by 2^16. This is done to scale all values to integers.

ie: if (x >= -524288 **(represents -8)** and x<= -457852**(represents -7)**)

then y<= (x/2048) + 256

and so one for all pice wise functions but there are now 15 picewise function

Fig. 13: State 2 of the improved model

The core functionality of the code remains the same. There are still four states present however, there is a modification made to state 2. The piecewise functions are increased as seen in the above fig.13. Increasing these piecewise functions allowed for the improvement in the model seen in fig.14.

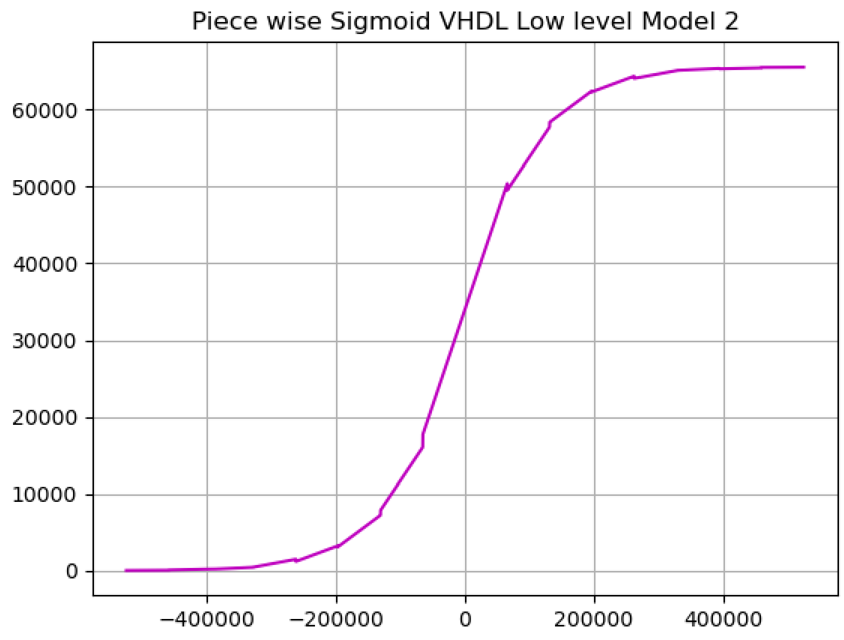


Fig. 14: Low level improved VHDL model.

In the above figure it can be noticed that the new model more closely represents the ideal sigmoid as presented in fig.9. This will result in an improvement in the overall error between the ideal sigmoid and the model. This will be further discussed in the verification methodology section.

D. Verification Methodology

1) Informal verification strategy: The informal verification strategy was implemented by taking the resultant value of the model at a position [i] for both the Python and the VHDL low-level models.

TABLE VI: Tabulated 10 points of both implemented models

i	Python Model 1	VHDL Model 1	Python Model 2 (improved)	VHDL Model 2 (improved)
158350	2474	2474	268	268
158351	2474	2474	268	268
158352	2474	2474	268	268
158353	2474	2474	268	268
158354	2474	2474	268	268
158355	2474	2474	268	268
158356	2474	2474	268	268
158357	2474	2474	268	268
158358	2474	2474	268	268
158359	2474	2474	268	268
158360	2474	2474	268	268

For the above tabulated data points it can been seen that there is zero error between the Python model and the VHDL model for both model 1 and model 2. From informal verification standpoint one can make the assumption that the models in VHDL are identical to the Python counterpart. Hence, for the visual inspection the Python model was plotted against the VHDL data that was saved in the text file from modelSIM. In the figure below this data is plotted on the same axis so as to make a visual confirmation that the two functions are indeed identical to one another.

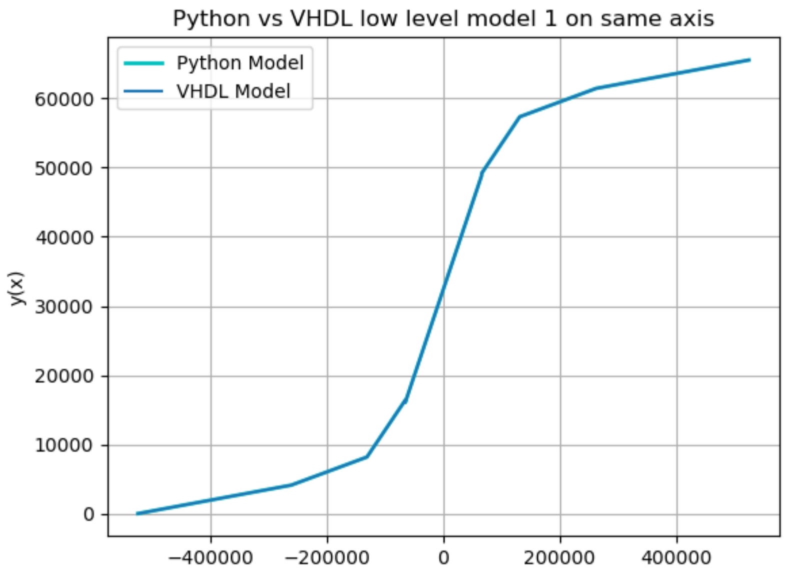


Fig. 15: Python vs VHDL for model 1

The same strategy was applied to the modified model which is depicted in the figure below. It can be seen at the improved model not only closely resembles the ideal sigmoid but also achieved zero error meaning the Python model and the VHDL model are indeed identical.

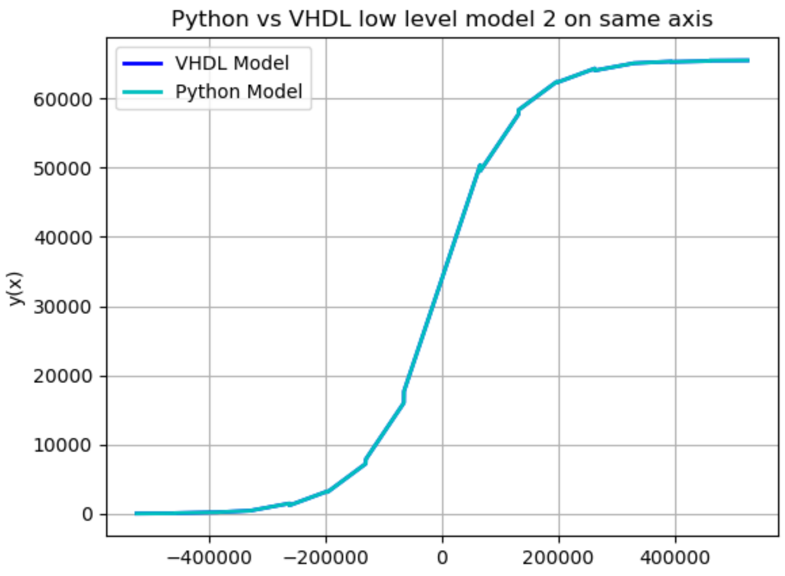


Fig. 16: Python vs VHDL for model 2

2) *Formal verification strategy:* The flow diagram below describes the formal verification process that is done automatically within python. The Python low-level model data is loaded as well as opening the text file in which the VHDL model was written to. Both models are stored into arrays. A loop is then run to test whether the index of array 1 value is equal to the index of array 2 to at

the same point. The difference between the two arrays at the identical index is then plotted. This plot should be a flat line at zero for the entire data range for the for the models to be considered identical and appropriate.

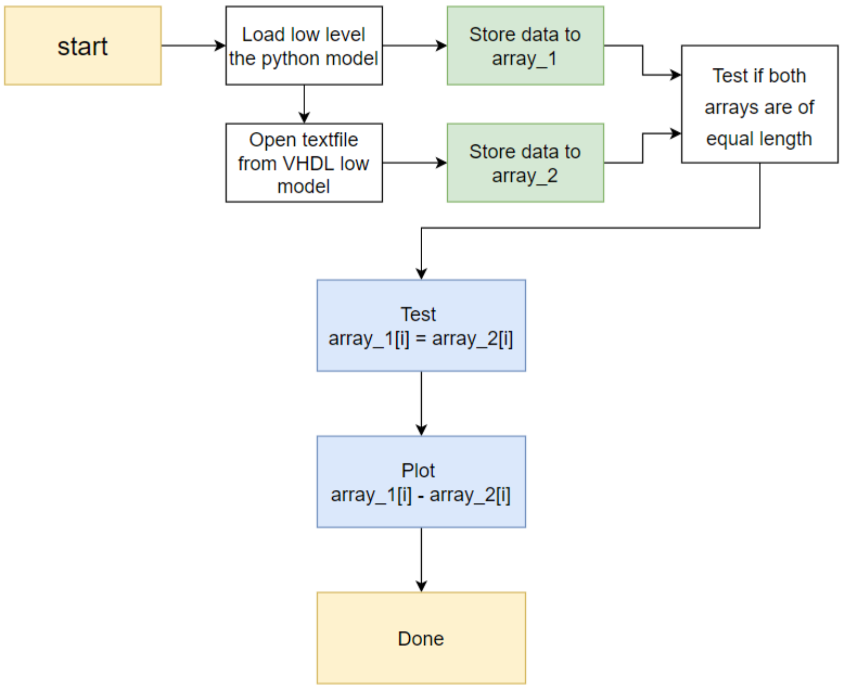

start
Load low level the python model
Store data to array_1
Open textfile from VHDL low model
Store data to array_2
Test if both arrays are of equal length
Test array_1[i] = array_2[i]
Plot array_1[i] - array_2[i]
Done

Fig. 17: Formal verification strategy

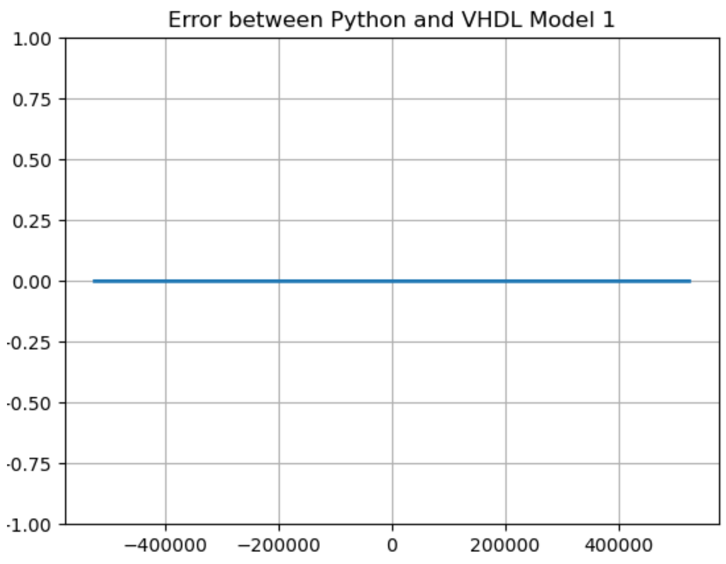


Fig. 18: Difference between VHDL model 1 and Python model 1

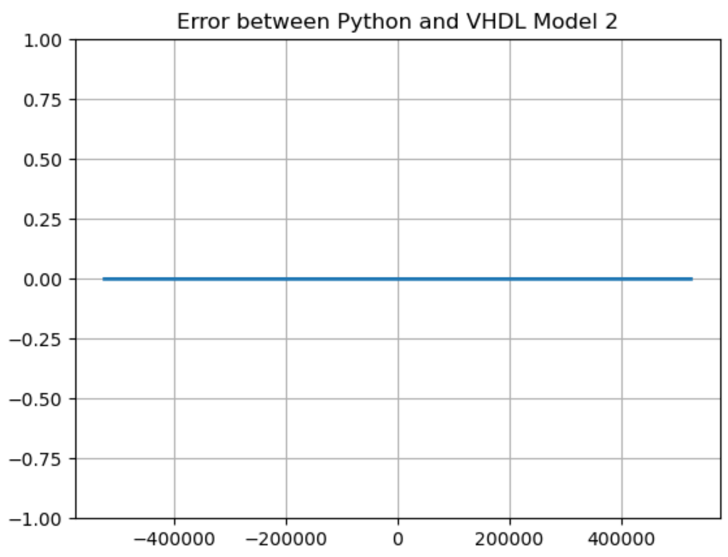


Fig. 19: Difference between VHDL model 2 and Python model 2

E. *Improvements over base model*

In order to test the overall improvement between the base model and improved model. The area between the ideal sigmoid and the base model as well as the ideal sigmoid and the modified model where recorded. The results of this exercise are presented in this section.

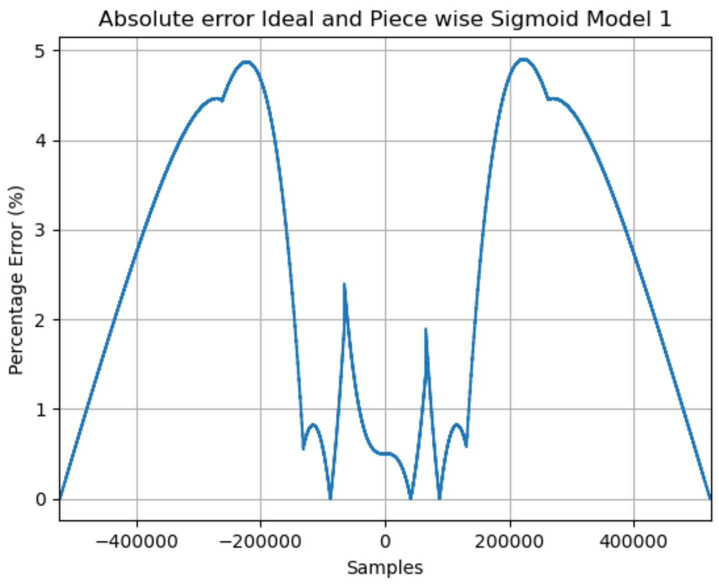


Fig. 20: Absolute error between ideal and the piece wise sigmoid model 1

The absolute error in model 1 was found to be around 4.85$\%$ seen in fig.20. This is in line with what was recorded by the research paper published by [10].

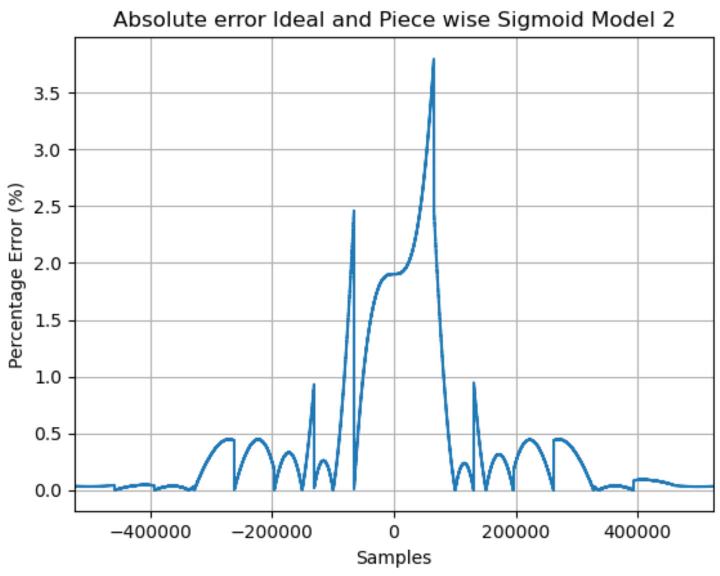


Fig. 21: Absolute error between ideal and the piece wise sigmoid model 2

The absolute error of the improved model was lower to the point of the absolute error in the improve model being 3.58% as presented in fig.21. This absolute error was noticed to be around the zero point of the sigmoid the piecewise functions gradients can be improved to reduce this error.

F. Conclusion

In conclusion both models were successfully implemented in VHDL. An overall bit error between the Python model and the VHDL model for both the base model as well as the modified model will be zero. furthermore with the modified model there was an improvement in absolute error of approximately 1.27%. Taking these factors into account it can be concluded that the system was successfully implemented in VHDL and was successfully and thoroughly verified within Python using both informal as well as formal automated verification processes.

PART D:
EXAMINATION REPORT

Submitted as part of the November examination in partial fulfillment of the requirements of ECSA Graduate Attribute 4 and 5 for the module EES 424 - Research Project.

By submitting this section of the document, I confirm that

- I have familiarised myself with the guidelines and requirements of this section as set out in the report framework document available through the course website,
- this is my own work and that I have conducted this research project in an ethical manner,
- I wish to be evaluated for my examination assignment on this copy of the report. I have double checked to verify that the content is correct, complete and representative of what I wish to be evaluated on.

V. Experimental Methodology

Piecewise sigmoid function was implemented in VHDL using methods described by [4] ans synthesised on the Altera Cyclone V SoC 5CSEMA4U23C6N FPGA. The data was collected for this implementation and a second improved model was implemented in VHDL as well. This was done to test the validity of the research statements. Analysis was conducted on both these results and comparisons will be made to ascertain the advantages and trade-offs of both implementations. The date was collected for this by transmitting the data via to a computer for analysis. For purposes of analysis the mean, maximum error. These function were implemented using both first principles as well as Python NumPy library. Other statistical properties that were looked into include standard deviation of the error as well as variance.

A. Mean

The method used to calculate the mean is done by taking the sum of all differences between the ideal sigmoid function and the model in question. The equation below describes the function implemented.

$$\bar{x} = \sum_{i=-2^{16}}^{N} \frac{f_1(i) - f_2(i)}{N} \tag{10}$$

Where:
f_1 = Ideal sigmoid given by eq.9.
f_2 = Model in question given by fig.15 or fig.16.
N = Total number of data points (1048576).

Having the mean the absolute error can be determined by using eq.11.

$$|error| = |\bar{x}| \cdot \frac{100\%}{2^{16}} \tag{11}$$

The result of equation (11) are given in fig.20 and fig.21.

B. Standard Deviation

Standard deviation gives one the idea of how widespread the data would be for these given models. In this case the standard deviation between the ideal sigmoid function and the models that are being implemented the standard deviation is given by equation below.

$$\sigma = \sqrt{\frac{\sum (x_i - \bar{x})^2}{N}} \tag{12}$$

C. Variance

The variance of a function is the expectation of the squared deviation of the variable that is in question from its mean. This gives a good understanding of how the data is spread from the mean.

$$\sigma^2 = \frac{\sum (x_i - \bar{x})^2}{N} \tag{13}$$

VI. Experimental Verification

A. *Informal verification strategy*

For informal experimental verification both models are implemented in VHDL and the data generated by each piecewise function being model 1 which is the base model and model 2 which is the improved research model. These models are compiled in VHDL and the data from each of the piecewise functions is saved to a text file. This text file can be imported into Python or any other software tool. The VHDL data stored in a text file is visually compared to the picewise function that were modeled in Python. The Results of this verification strategy can be found in the appendix fig.27 to fig.32. These results form purely a visual representation of the models here one can make visual verification to see if the models are are indeed identical to those that have been designed in Python.

The figure below code flow of how the data is processed in order to make visual verification.

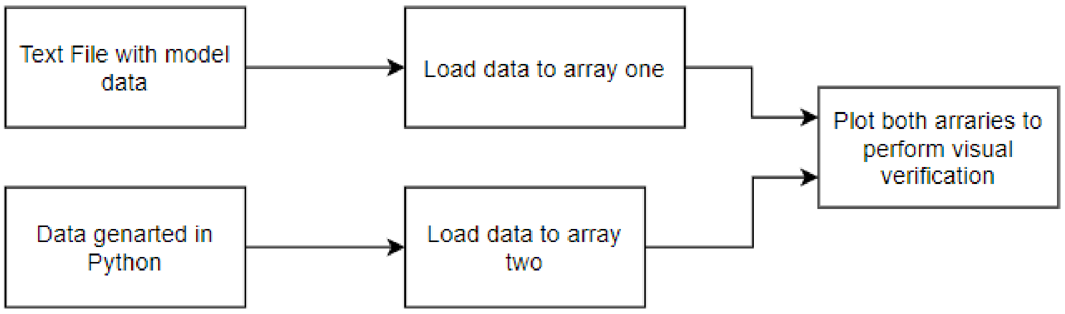

Text File with model data
Load data to array one
Plot both arraries to perform visual verification
Data genarted in Python
Load data to array two

Fig. 22: Visual verification process

B. *Formal verification strategy*

For the formal verification more stringent performance metrics for measuring the data that has been saved from the VHDL models. This data was then imported to Python and tests such as mean, standard deviation and variance were for calculated. These performance metrics provide greater insight into how accurately the sigmoid function has been implemented on the VHDL system. To determine the level of success of the implemented models the ideal sigmoid which has been designed in Python and the serial data that is received from the FPGA board or plotted on the same axis. The area between the two curves is then calculated and save to an array. The data points that are saved in the array are then used to calculate the mean, standard deviation as well as a variance in the error using. These statistical values the success of the system can be measured.

VII. Data Collection

The generation of the sigmoid functions was carried out on the Altera Cyclone V FPGA board. This board has the capability of serial. Data is transmitted using a GPIO (general purpose input output) pins that are located on the board. The Rx and Tx pins on the FPGA are connected to the Tx and Rx pins of an Arduino the behaves as a serial bridge. A clock of 50 MHz was set as a default clock speed this allows to clock down the FPGA to operate at the required speed. 16 bits of data are required to be transmitted therefor data is split into two 8-bit packets and these eight bits are sent at the time. In doing this one would require the need to create two transmission cycles in order to send the entire point of the sigmoid.

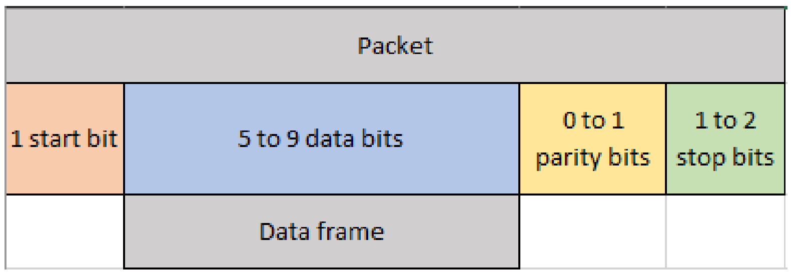


Fig. 23: UART data packet

The data was received in Python the PySerial library was used and this allowed for parameters to be specified for data transmission and reception via serial such as setting the buad rate and the amount of samples that the system would receive. As mentioned earlier and Arduino was used to facilitate a serial Bridge the baud rate on the Arduino, FPGA and Python are all set to the same baud. The data that is received in Python was saved to a text file so that it could be interpreted and visually inspected for any errors or gibberish within the transmission.

Informal verification or visual verification was performed on the data that was received via serial and save to the text file. The data was loaded into a python array and simply plotted to get a visual outline of the shape of the segment that was formed by the data that was received. However, this does not give one great depth on how well the function was implemented. Rather it is a visual tool to allow the reader to see if a sigmoid was implemented indeed.

VIII. Results

A. Results

The table provided below tabulate the results for the mean error maximum and standard deviation and variance for model 1 which is the base model and model to purchase the research model.

TABLE VII: Results using Python NumPy library

	Mean Error %	Maximum Error %	Standard Deviation %	Variance %
Model 1	2.499959697720615	4.8774	1.6467998670503385	2.71194980211701
Model 2	0.44108161631399945	3.79721	0.6870689010864655	0.47206367484016326

TABLE VIII: Results using Python first principals

	Mean Error %	Maximum Error %	Standard Deviation %	Variance %
Model 1	2.4999596977207514	4.87714	1.6467998670504315	2.711949802116900
Model 2	0.4410816163140015	3.79721	0.6870689010865665	0.47206367484016224

The data provided below is the absolute error range for both model 1 and model 2 over the entire data range.This figure was obtained by taking the serial data that was transmitted from the FPGA board into Python and this data was used as the input variable for equations 10 and 11.

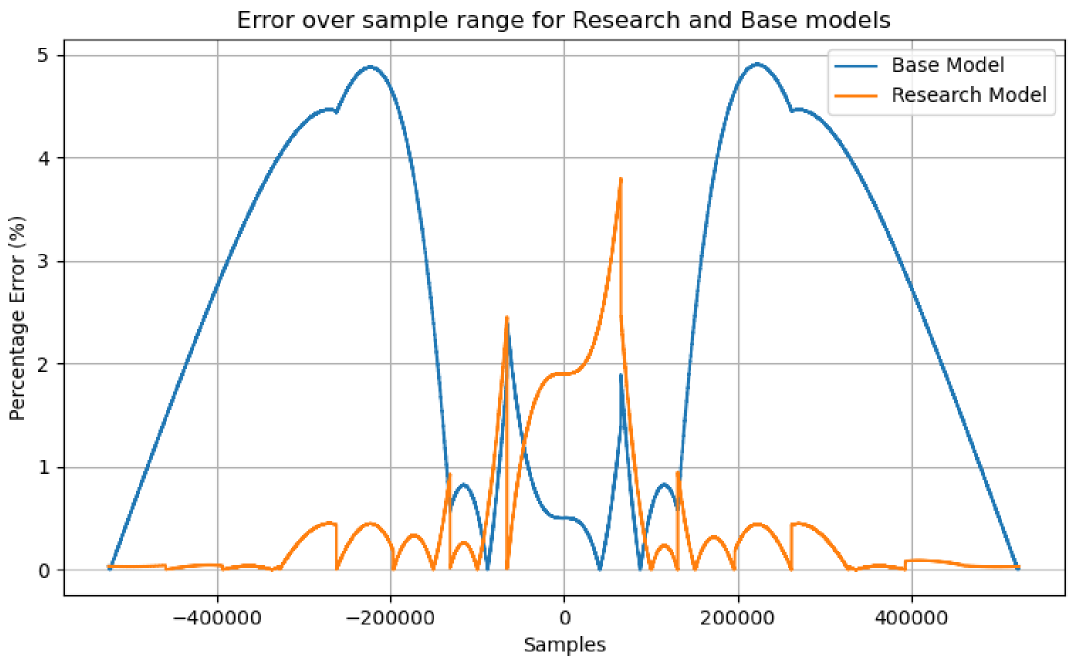


Fig. 24: Research Model vs Base Model

B. Discussion of results

1) Mean of results: This statistical value relays the average value of a set of data. In the case of results presented in the table provided above. The means for the models where calculated using equations 10 and 11. The results from the Numpy functions and the results obtained by implementing the mentioned equations yielded the effectively the same results down to the 10th decimal point. The reason why the mean can not be used as sole metric for success come down to to the fact it does not relay to one how much the error varies over time. The fig.24 above is the absolute error over the entire sample range. The error changes over the sample time and just using a mean does not tell one anything about by how much does the error deviated over the sample space.

2) Standard deviation of results: Standard deviation is a measure of spread that is, how spread out a set of data is. Data that has a low standard deviation tells us that the data is closely clustered around the average fig.26. While a high standard deviation indicates the data is dispersed over a wider range of values fig.25. Standard deviation is used when the distribution of data is approximately normal resembling a bell curve. Standard deviations commonly used to understand whether a specific data point is standard unexpected or unusual and unexpected standard deviation is. In the case of the results that have been tabulated in tables VII and VIII. The standard deviation of model 2 is 1% better. This indicates that the error concentrated closer to the mean. The results between the NumPy library function versus the first principles approach wherein essence identical.

3) Variance of results: The objective of using the variance a as a measure of success allows for one to determine how spread out the data is. If the data is largely spread out this would result in large deviations in the moon this would mean that the points of error would be dispersed far apart. Ideally, for the implementation of the sigmoid function, one would like the average error for the deviation between points to be close to the mean. The variance is a good measure of that in the sense that it describes the spread of the data. Looking at the Gaussian distributions provided one can see that fig.25 the data is a lot more spread out. Whilst, the spread of fig.26 is significantly better than the spread fig.25. This indicated an improvement in the research model from the base model.

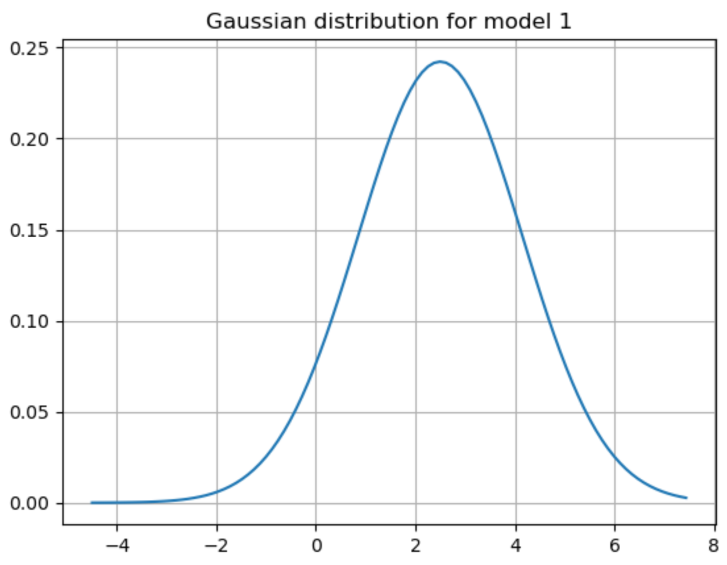


Fig. 25: Gaussian distribution for base model

The figure above represents a Gaussian distribution for the base model. In this figure above it can be seen that the mean value is approximately 2.5%.

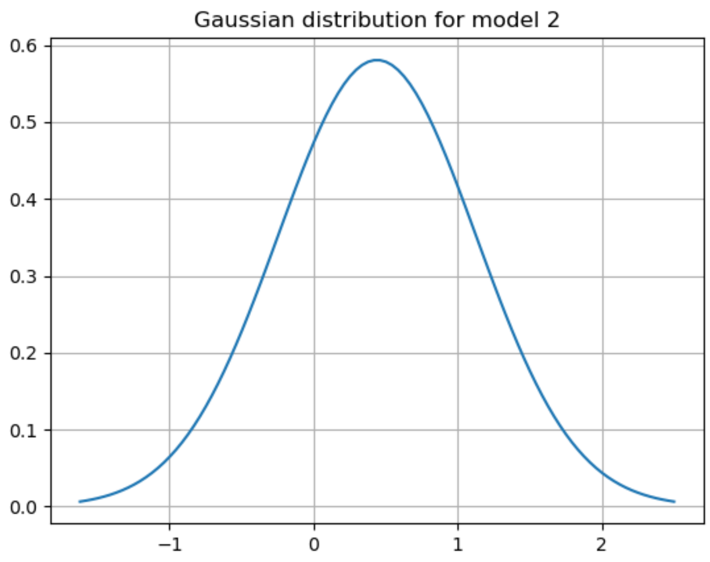


Fig. 26: Gaussian distribution for research model

The figure above represents the Gaussian distribution for the research model and in this figure can be seen that the mean value is 0.4%.

C. Comparison of results

TABLE IX: Table of resource usage on Altera Cyclone V FPGA board

	Logic utilization(ALM needed)	ALUT usage	Dedicated Logic registers	DSP Blocks
Model 1	157	258	57	0
Model 2	297	508	57	0

The resource usage data obtained from Quartus is tabulated above. The resource usage on the resource is almost double that of the base model. The increase in resource usage can also translate to the increase in processing time. The results for the improvements that were discussed prior come at the cost of double the recourse usage as well as a increase in simulated run-time. Model 1 takes $41.94308\mu s$ and model 2 taking $78.89604\mu s$ to complete the entire sigmoid function in ModelSim simulation.

The choice of model will come down to the question. Does the user require an activation function which has a high level of accuracy? or Does the user require a shorter processing time and a lower resource usage?

Model one will be suitable for a device which does not have the 297 logic (ALM) to compute the sigmoid function with a computation time of $41.94308\mu s$. Whilst model two will be more suitable for a device which has a higher level of computing power and this will result in a improvement in the accuracy of the function and achieve an error rate below 1% and the cost of resources and a computation time of $78.89604\mu s$.

IX. Conclusion

In the investigation of the application of a sigmoid function or a sigmoid activation function implemented on an FPGA board was carried out.The question that needed to be addressed. Can the sigmoid function be implemented with a mean error rate of below 5%? Multiple models were constructed using various simulation tools and design aids such as a low-level model in Python vs the low-level model in VHDL. Verification and analysis were conducted on both these models to determine the accuracy of the VHDL system. Further performance parameters were considered such as mean, standard deviation and variance as well as the resource usage and completion time for each of the models. The model has been successfully implemented and synthesized on the FPGA device and has successfully answered the research question. The system can be implemented with a mean error below 5% however there are trade-offs to greater accuracy vs resource usage. The more accurate system has a higher resource usage and computation time than that of the system which has a lower resource usage but also has lower accuracy.

REFERENCES

[1] Z. Li, Y. Huang, and W. Lin. Fpga implementation of neuron block for artificial neural network. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2, 2017.

[2] P. W. Zaki, A. M. Hashem, E. A. Fahim, M. A. Masnour, S. M. ElGenk, M. Mashaly, and S. M. Ismail. A novel sigmoid function approximation suitable for neural networks on fpga. In *2019 15th International Computer Engineering Conference (ICENCO)*, pages 95–99, 2019.

[3] Z. Xie. A non-linear approximation of the sigmoid function based on fpga. In *2012 IEEE Fifth International Conference on Advanced Computational Intelligence (ICACI)*, pages 221–223, 2012.

[4] D. J. Myers and R. A. Hutchinson. Efficient implementation of piecewise linear activation function for digital vlsi neural networks. *Electronics Letters*, 25(24), 1989.

[5] M. T. Tommiska. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings - Computers and Digital Techniques*, 150(6):403–411, 2003.

[6] Cesare Alippi and Giancarlo Gajani. Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning. pages 1505 – 1508 vol.3, 07 1991.

[7] H. Amin, K. M. Curtis, and B. R. Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings - Circuits, Devices and Systems*, 144(6):313–317, 1997.

[8] Ming Zhang, S. Vassiliadis, and J. G. Delgado-Frias. Sigmoid generators for neural computing using piecewise approximations. *IEEE Transactions on Computers*, 45(9):1045–1049, 1996.

[9] I. Tsmots, O. Skorokhoda, and V. Rabyk. Hardware implementation of sigmoid activation functions using fpga. In *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*, pages 34–38, 2019.

[10] Alin Tisan, Stefan Oniga, Daniel Mic, and Buchman Attila. Digital implementation of the sigmoid function for fpga circuits. *ACTA TECHNICA NAPOCENSIS Electronics and Telecommunications*, 50, 01 2009.

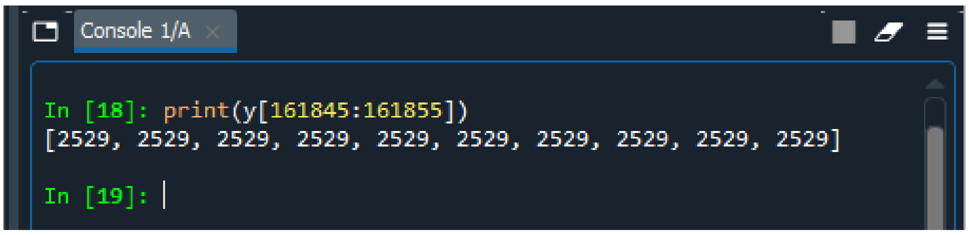
[11] F. Vahid. *Digital Design 2nd Edition with RTL Design, VHDL, and Verilog and Verilog for Digital Design Set*. John Wiley & Sons Canada, Limited, 2010.

[12] M.H. Khan. Research proposal- design of vhdl based function calculation prototype. University of Pretoria, 2020.

APPENDIX A
SOFTWARE TOOL UTILISED

A. *Spyder IDE (Python)*

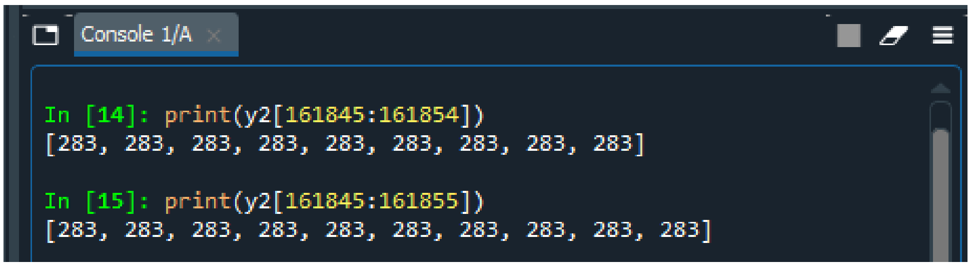
The Spyder Python it was crucial in the development of both low-level models of the sigmoid function . The IDE provides extensive feature as well as ease of use of variable explorer to see values that are being loaded into the variables that are created. This makes debugging and troubleshooting a lot more user friendly as well as reducing downtime looking for bugs. There are various other IDEs that can be used to achieve the required results. An example of an IDE is Microsoft VS Code. However these IDEs are more complex to setup and are not designed for a scientific and or data analysis applications. Secondly Python is selected as the language of choice mainly because of its easy of use and extensive literature available online. A close second language would be Matlab and it can even be said Matlab supersedes Python in some use cases but the high cost of obtaining a user licence prevents its use. The figures below are images of the console printing values of the implemented models for a small value range.



```
In [18]: print(y[161845:161855])
[2529, 2529, 2529, 2529, 2529, 2529, 2529, 2529, 2529, 2529]

In [19]: |
```

Fig. 27: Spyder console for model 1



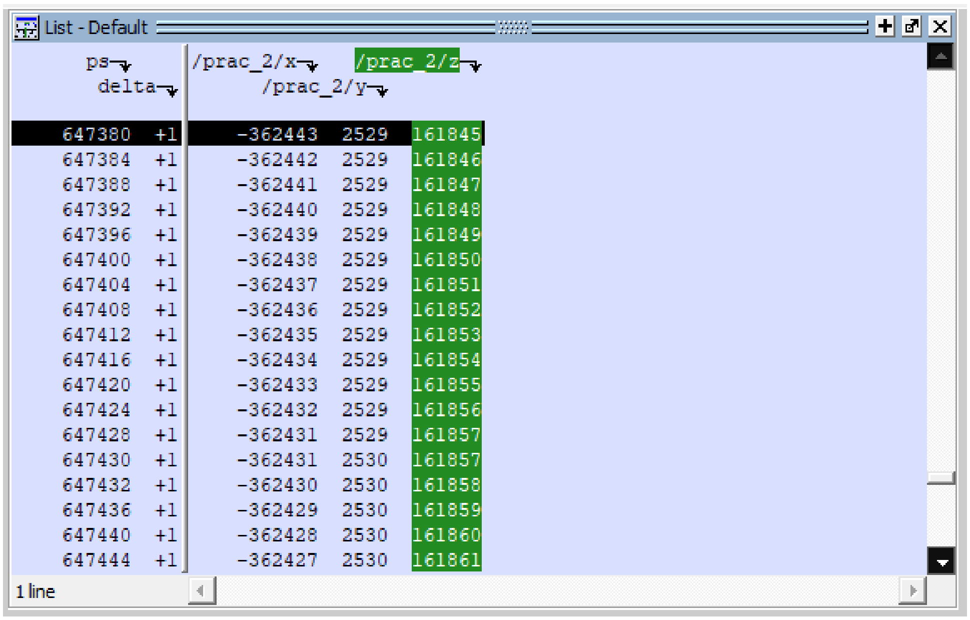
```
In [14]: print(y2[161845:161854])
[283, 283, 283, 283, 283, 283, 283, 283, 283]

In [15]: print(y2[161845:161855])
[283, 283, 283, 283, 283, 283, 283, 283, 283, 283]
```

Fig. 28: Spyder console for model 2

B. *ModelSim/Quartus*

ModelSim is also another integral part in the success of the implementation of the sigmoid function. ModelSim allows the user to run VHDL system test match models within its test environment. In this test environment the user is able to choose different clock frequencies as well as manually assign values to all the variables and signals that have been created in the VHDL Script. The user can print a list of values that are stored at in the signal or variable or the user can plot the waveform of the signals and variables. These data sets can be either represented as binary, hex, decimal or various other data types that are available in the software suite. An alternative to Modelsim is Cadence Xcelium Logic Simulation this software is not very well documented and has a very high cost of entry. Quartus/ModelSim are designed to work with the Cyclone SoC that are embedded on the FPGAs issued. Furthermore, the ModelSim is very well documented as well as many tutorial videos available online . The figures that are presented below are the data values from the range of 161845 to the range 161855. The same range is presented in the figures above which are the two Python console figures. It can be noted from that figure that the models in Python and the models in VHDL all have the exact same value for the same data range.

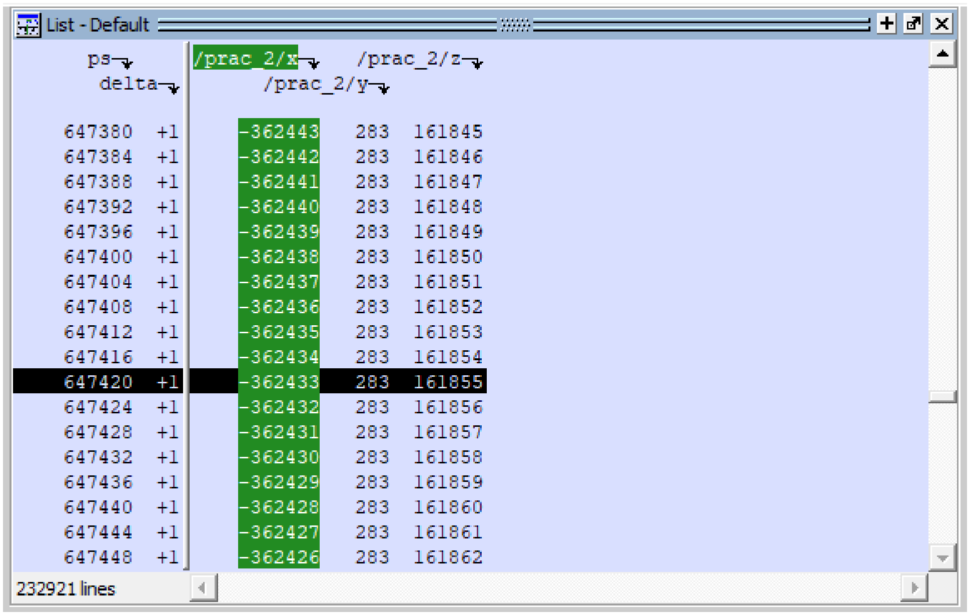


```
       ps    /prac_2/x    /prac_2/z
    delta       /prac_2/y

647380  +1   -362443  2529  161845
647384  +1   -362442  2529  161846
647388  +1   -362441  2529  161847
647392  +1   -362440  2529  161848
647396  +1   -362439  2529  161849
647400  +1   -362438  2529  161850
647404  +1   -362437  2529  161851
647408  +1   -362436  2529  161852
647412  +1   -362435  2529  161853
647416  +1   -362434  2529  161854
647420  +1   -362433  2529  161855
647424  +1   -362432  2529  161856
647428  +1   -362431  2529  161857
647430  +1   -362431  2530  161857
647432  +1   -362430  2530  161858
647436  +1   -362429  2530  161859
647440  +1   -362428  2530  161860
647444  +1   -362427  2530  161861
```

1 line

Fig. 29: ModelSim signal list for model 1

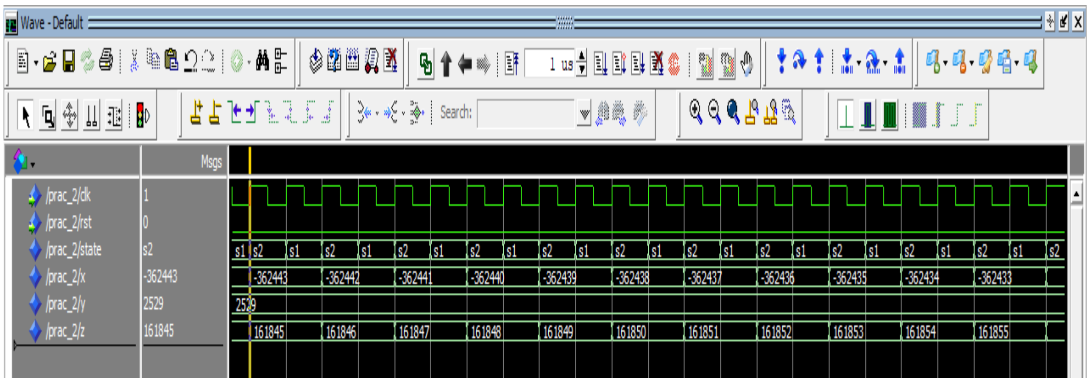


```
       ps    /prac_2/x    /prac_2/z
    delta       /prac_2/y

647380  +1   -362443  283  161845
647384  +1   -362442  283  161846
647388  +1   -362441  283  161847
647392  +1   -362440  283  161848
647396  +1   -362439  283  161849
647400  +1   -362438  283  161850
647404  +1   -362437  283  161851
647408  +1   -362436  283  161852
647412  +1   -362435  283  161853
647416  +1   -362434  283  161854
647420  +1   -362433  283  161855
647424  +1   -362432  283  161856
647428  +1   -362431  283  161857
647432  +1   -362430  283  161858
647436  +1   -362429  283  161859
647440  +1   -362428  283  161860
647444  +1   -362427  283  161861
647448  +1   -362426  283  161862
```

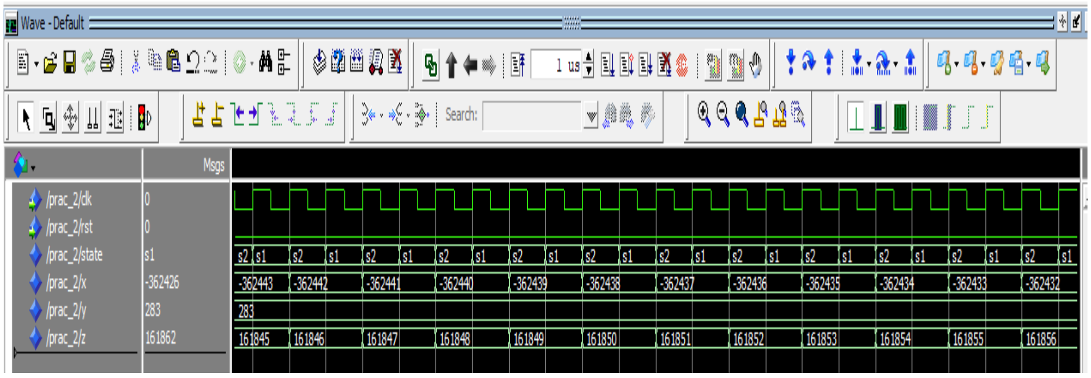
232921 lines

Fig. 30: ModelSim signal list for model 2



```
Wave - Default
                Msgs
/prac_2/clk     1
/prac_2/rst     0
/prac_2/state   s2      s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2
/prac_2/x       -362443 -362443 -362442 -362441 -362440 -362439 -362438 -362437 -362436 -362435 -362434 -362433
/prac_2/y       2529    2529
/prac_2/z       161845  161845 161846 161847 161848 161849 161850 161851 161852 161853 161854 161855
```

Fig. 31: ModelSim waveform for model 1



```
Wave - Default
                Msgs
/prac_2/clk     0
/prac_2/rst     0
/prac_2/state   s1      s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1 s2 s1
/prac_2/x       -362426 -362443 -362442 -362441 -362440 -362439 -362438 -362437 -362436 -362435 -362434 -362433 -362432
/prac_2/y       283     283
/prac_2/z       161862  161845 161846 161847 161848 161849 161850 161851 161852 161853 161854 161855 161856
```

Fig. 32: ModelSim waveform for model 2

C. Outcome from engineering methods, skills and tools

Quartus a very useful tool for the synthesis of VHDL and Verilog models. This tool allows the user to run advanced synthesis at the gate level. The tool also provides the user with a lot of information regarding run time and device resource usage as well as synthesise logic level models. However, if the user just requires the compilation of code to test in a test bench the compilation can take some time. The workaround for this was to code the vVHDL file in notepad++ and compile the file in ModelSim and simulate in ModelSim this takes the compiling time from minutes to just a few seconds. Another benefit of working in notepad++ is the autocomplete feature which can save some time. ModelSim also allows the user to compile multiple VHDL file simultaneously since it is not dependent on a top-level entity where Quartus is.

Both ModelSim and Quartus have their place. However, if the user just requires that quick compilation of the code to detect for any syntax errors writing the code in notepad++ and compiling in ModelSim is a lot faster than synthesising the project in Quartus. But if the user requires more resource metrics then Quartus is the better option.