TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

IEE70LT

Silvestr Knuut 144022IVEM

EEG SIGNAL PATTERN RECOGNITION USING AN ARTIFICIAL NEURAL NETWORK

Master's Thesis

Supervisor: Ants Koel
PhD
Lecturer

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

IEE70LT

Silvestr Knuut 144022IVEM

MUSTRITUVASTUS EEG SIGNAALIST KASUTADES TEHISNÄRVIRAKKUDE VÕRKU

Magistritöö

Juhendaja: Ants Koel
Doktorikraad
Lektor

Tallinn 2016

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Silvestr Knuut

06.06.2016

Abstract

This paper proposes the use of an artificial neural network to identify a combination of electroencephalography signals of the human brain. The result is a system that reacts to the brain activity faster than a human makes a conscious decision to be actualized as action.

This thesis is written in English and is 65 pages long, including 59 chapters, 61 figures.

Annotatsioon

MUSTRITUVASTUS EEG SIGNAALIST KASUTADES TEHISNÄRVIRAKKUDE VÕRKU

See töö käsitleb tehis-närvirakkude võrgu kasutamist elektro-entsefalograafia signaalide kombinatsioonist tegevust ennetava mustri tuvastamiseks. Tulemuseks on süsteem, mis on võimeline reageerima aju aktiivsuse peale kiiremini, kui inimene saab teha teadliku otsusse realiseerida registreeritud aktiivsuse konkreetse tegevusena.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 65 leheküljel, 59 peatükki, 61 joonist.

List of abbreviations and terms

TUT	Tallinn University of Technology
EEG	Electroencephalography
ANN	Artificial Neural Network
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
ALU	Arithmetic Logic Unit
DPU	Data Path Unit
IP	Intellectual Property
LUT	Look Up Table

Table of contents

List of figures

1 Introduction

The goal is to realize an Artificial Neural Network system that identifies signals that have a certain meaning from Electroencephalography.

Everything that a person feels, thinks, sees or plans to do, conscious or unconscious happens in the brain, so it follows from this that it is possible that this data is reflected in the EEG signal of the brain.
Benjamin Libet has conducted an experiment that shows how the brain activity rises before a person makes a conscious decision to make a movement. The experiment shows that conscious decision to act is preceded by an unconscious build-up of readiness potential to be actualized as a movement by approximately 300mS and another 200mS before the conscious decision is actualized as movement.
Totalling to approximately 500mS delay starting from the rise of the brain activity to the initiation of the action

The signals that indicated this build-up of readiness potential were captured and recorded by and EEG and came to called: “Bereitschaftspotential” [1].
Since there is a way to read this readiness potential of the brain, then it is possible to build a system that reacts faster than the person is able to for example in critical situations, where a fast response is needed, or a system that actually reads the mind of a physically impaired person and controls a wheelchair for instance.

However EEG signals are difficult to read because of numerous “Biological Artifacts”.
From Wikipedia:
“Electrical signals detected along the scalp by an EEG, but that originate from non-cerebral origin are called artefacts. EEG data is almost always contaminated by such artefacts. The amplitude of artefacts can be quite large relative to the size of amplitude of the cortical signals of interest. This is one of the reasons why it takes considerable experience to correctly interpret EEGs clinically. Some of the most common types of biological artefacts include:

Eye-induced artefacts (includes eye blinks, eye movements and extra-ocular muscle activity)
ECG (cardiac) artefacts
EMG (muscle activation)-induced artefacts
“Gloss kinetic artefacts”

Because of the presence of artifacts, classifying the EEG signals in real time by using conventional means is either impossible or takes a considerable amount of time and expertise. Therefore, I propose the use the Artificial Neural Networks.
In recent years a number of studies were conducted, where ANNs of different forms and sizes were implemented to read and classify EEG signals of the brain. Majority of the studies focus on epilepsy prediction and driver fatigue. Prediction of conscious decision to be actualized as action has not been widely investigated yet. The closest studies to the current thesis include: predicting driving fatigue [8], classification of EEG signals [9] and brain-computer interface [10].
Artificial neural networks include a list of traits like: massive parallelism, distributed representation and computation, learning ability, generalization ability, inherent contextual information processing, fault tolerance that give them the ability to derive meaning from complicated or imprecise data [7].
ANNs are used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. They are usually used to model complex relationships between inputs and outputs or to find patterns in data.
The goal of this work is to realise in VHDL a simple artificial neural network, that reacts by sending a discreet “true” or ”false” signal when it receives a unique combination of discrete input signals in form of 16bit vectors.

2 The Artificial Neuron model

A formal artificial neuron model is shown on Figure 1.

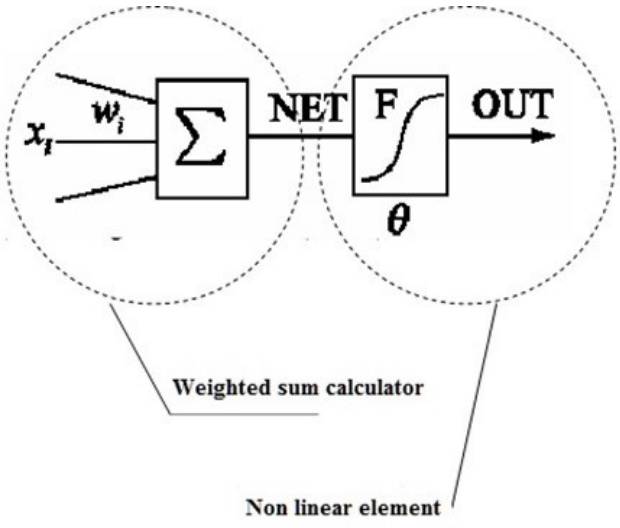


Figure 1. Artificial Neuron model

An artificial neuron is a mathematical function conceived as a crude model, or abstraction of biological neurons. It receives input from some other units, or perhaps from an external source. Each input has an associated weight w, which can be modified so as to model synaptic learning. The output can serve as input to other units.
The artificial neuron model consists of two elements:

1. The weighted sum calculator
2. Nonlinear element

The "Weighted sum calculator" is described in Equation (1)

$$NET = W * X = [w_1 \dots w_n] * \begin{bmatrix} x_1 \\ \dots \\ x_2 \end{bmatrix} = w_1 x_1 + \dots + w_n x_n \quad (1)$$

Where x_n are the input signals, a plurality of neuron input signals forms a vector X,
w_n - are the weight coefficients that for the vector W,

NET is the weighted sum that is passed on to the nonlinear element.

F – is the nonlinear element called the "Activation function"

The output OUT of the neuron is described by Equation (2) [3].

$$OUT = F(NET) \tag{2}$$

2.1 Types of activation functions

Over the years many different types of activation functions have emerged, functions described here are the ones used most commonly.

2.1.1 Step Function

The step function is shown on Figure 2.

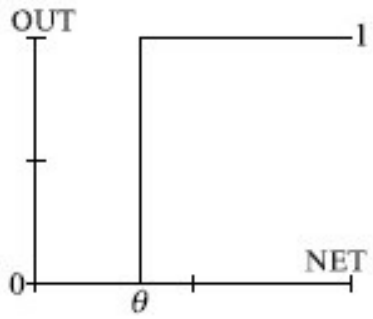


Figure 2. Step function

The step function is used in the classical neuron model. Easily realised in code or hardware, neurons with this type of nonlinear activation function don't use much resources. This function is extensively simplified and does not allow modelling of systems that have to work with continuous signals. The absence of the first derivative makes it difficult to implement gradient descent based learning algorithms.

The step function is described by Equation (3) [3].

$$OUT = \begin{cases} 0, NET < \theta \\ 1, NET \geq \theta \end{cases} \tag{3}$$

2.1.2 Sigmoid Function

The sigmoid function is shown on Figure 3.

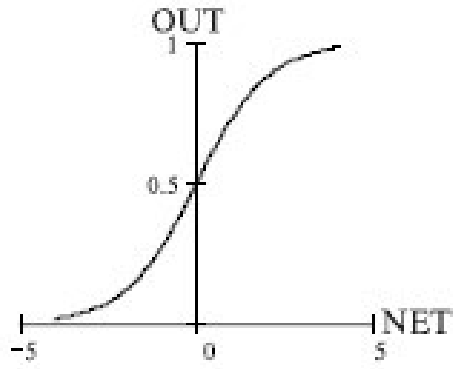


Figure 3. Sigmoid function

The sigmoid function is very often used in multilayers perceptrons and networks that work with continuous signals.

Its important qualities are: smoothness and continuity. Continuity of the first derivative allows the application of gradient descent learning algorithms, like the backpropagation algorithm [11].

The function is symmetrical relatively to the point (NET=0, OUT=1/2), this makes OUT=0 and OUT=1 equitable, which is important for the operation of the network. However the range of the input values from 0 to 1 is not symmetrical and because of that, the learning phase takes a considerable amount of time.

The sigmoid function is described by Equation (4) [3]

$$OUT = \frac{1}{1+e^{-NET}} \tag{4}$$

2.1.3 Hyperbolic Tangent Function

The Hyperbolic tangent function is shown on Figure 4.

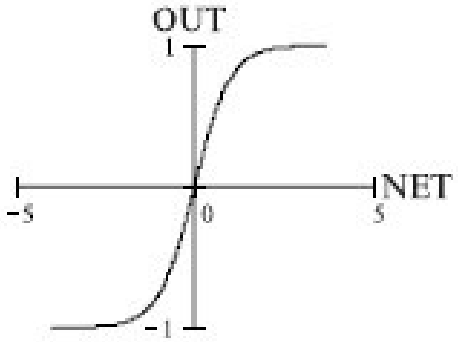


Figure 4. Hyperbolic Tangent function

The hyperbolic tangent function is also widely used. The function is symmetrical in relation the point (0:0) this gives it an advantage over the sigmoid function when it comes to the time spent in learning phase.

The hyperbolic tangent function is described by Equation (5) [3]

$$OUT = th(NET) = \frac{e^{NET} - e^{-NET}}{e^{NET} + e^{-NET}} \quad (5)$$

2.1.4 Ramp function

The Ramp function is shown on Figure 5.

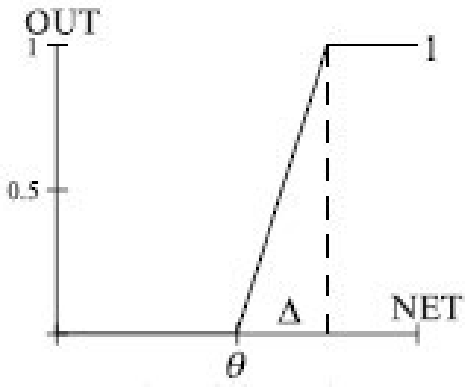


Figure 5. Ramp function

Easily computable, but has a discontinuous first derivative at the points

NET = θ, NET = $\theta + \Delta$, which makes the learning algorithm difficult.

The ramp function is described by Equation (6) [3]

$$OUT = \begin{cases} 0, NET \leq \theta \\ \frac{(NET-\theta)}{\Delta} \quad , \theta \leq NET < \theta + \Delta \\ 1, NET \geq 0 + \Delta \end{cases} \tag{6}$$

2.1.5 Gaussian function

The Gaussian function is shown on Figure 6.

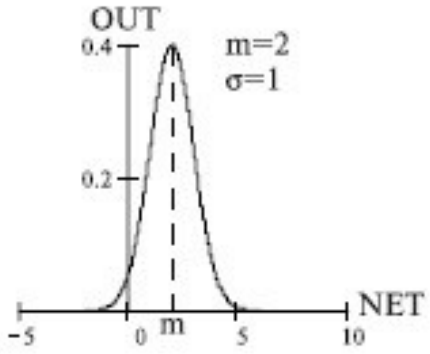


Figure 6. Gaussian function

The Gaussian activation function is used when the response of the neuron has to be maximal for a some specific value of NET.

The Gaussian function is described by Equation (7)

$$OUT = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(NET-m)}{2\sigma^2}} \tag{7}$$

As it stands today, there are no algorithms that help with the choice of the activation function. The choise usually depends on:

• The specific task (input and output value and physical interpretations)

• The method of realization of NN (on a computer or in a shape of an electronic circuit)

• The learning algorithm in use [3]

Normally the activation function in the neurons is linear in the input and output layer of the NN, while the functions of the hidden layers may vary. Often, these are hyperbolic tangent functions.

3 A typical ANN structure

As with the choise of the activation function, there are no algorithms that assist in putting the structure of the ANN together. The amount of neurons is arbitrary as is the way they are connected. The ANN can have any number of layers, the multi-layer perceptron is typically referred as an example of an ANN, shown on Figure 7.

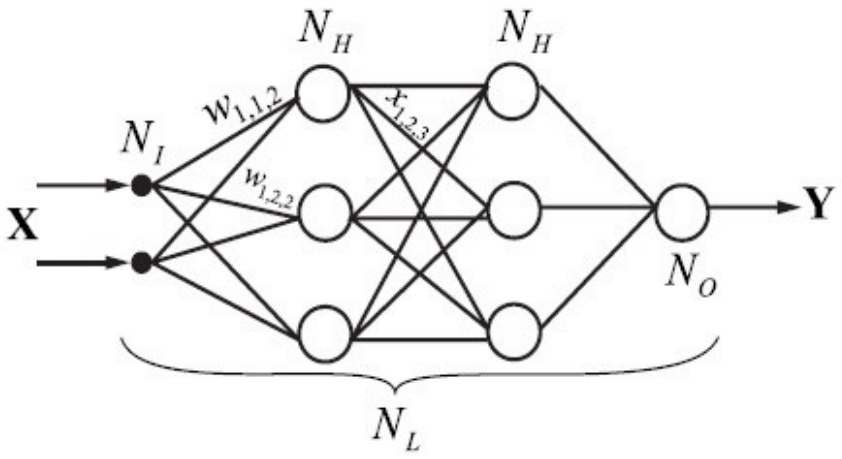


Figure 7. A multi-layer perceptron

The multi-layer perceptron has $N_L = 4$ layers where:

1. The input layer has N_L number of neurons
2. Two hidden layers has N_H number of neurons in each of them
3. The output layer has N_O number of neurons

X is the input signal vector and Y is the output signal vector. It is often debated whether the input layer should or should not be taken into account, as in this case its function is only to distribute the input signals X [3].

In my case I rely on the task at hand to determine the structure of the network as well as the activation functions in use.

4 ANN structure to be realised

As the ultimate goal is to realize an ANN that works with the information coming from a EEG signal, I turn to the waveform of the EEG signal. The EEG waveform is subdivided into 6 bandwidths that are known as: alpha, beta, delta, gamma, theta and Mu where:

Delta is in the frequency range up to 4Hz and is normally observed in adults in shallow sleep, thus is of no use to the purpose of this project Figure 8.

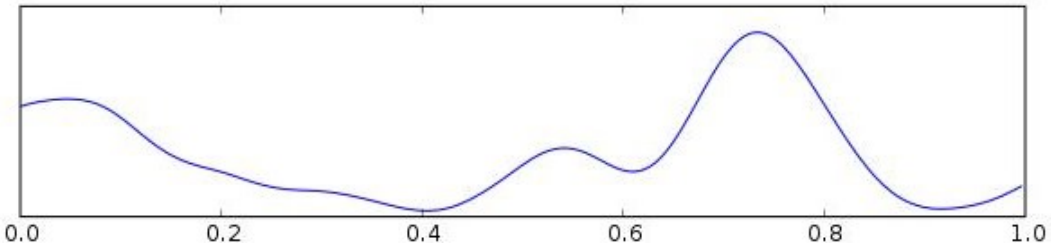


Figure 8. Delta wave

Theta is in the frequency range of 4Hz to 7Hz and is normally observed in drowsiness in children and meditation in adults. On the contrary this range has been associated with reports of relaxed, meditative, and creative states Figure 9 [4].

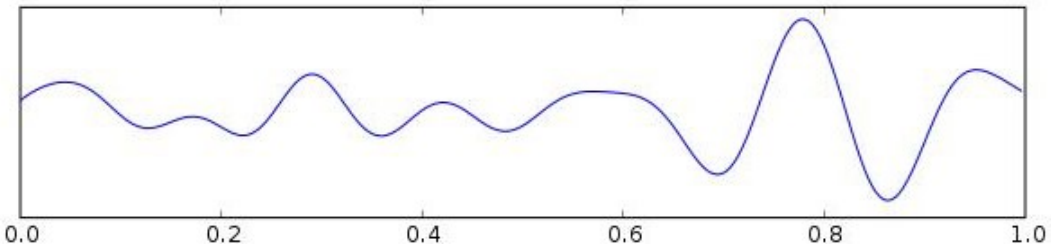


Figure 9. Theta wave

Alpha is in the frequency range of 7 Hz to 14Hz. It emerges when a subject is relaxed with the closing of the eyes and it attenuates with mental stress or/and opening of the eyes Figure 10 [5].

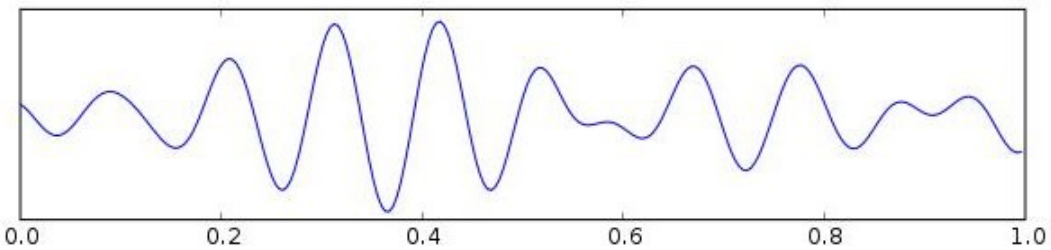


Figure 10. Alpha wave

Beta is in the frequency range of 15 Hz to 30Hz. Beta wave is closely linked to motor behaviour and is generally attenuated during active movements. Low amplitude beta with multiple and varying frequencies is often associated with active, busy or anxious thinking and active concentration. It is the dominant rhythm in people who are alert or anxious Figure 11 [6].

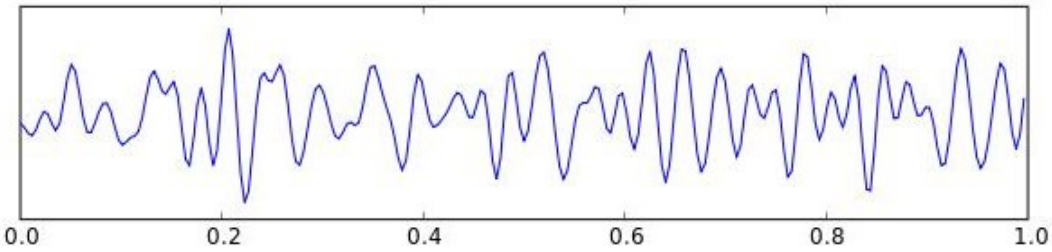


Figure 11. Beta wave

Gamma is in the frequency range of approximately 30Hz to100 Hz. Gamma rhythms are thought to represent binding of different populations of neurons together into a network for the purpose of carrying out a certain cognitive or motor function Figure 12.

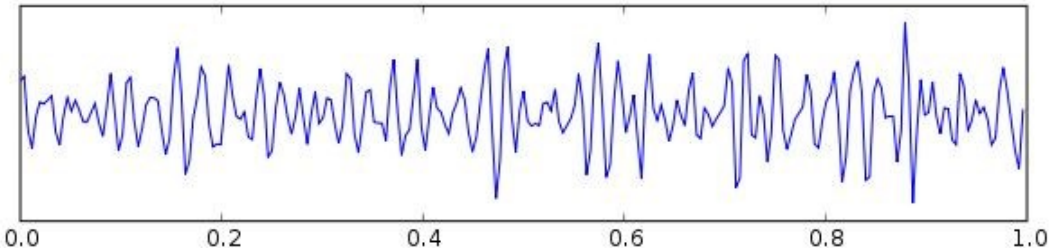


Figure 12. Gamma wave

Mu is in the frequency range of 8Hz to 13Hz and partly overlaps with other waves. It reflects the synchronous firing of motor neurons in rest state Figure 13.

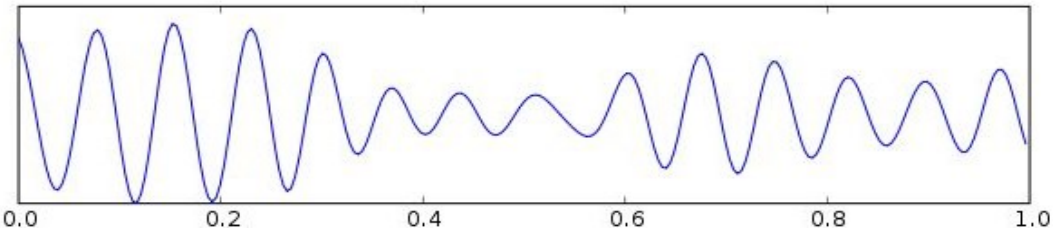


Figure 13. Mu wave

[2]

Based on the information about the waves, I choose to look for pattern combinations of theta, alpha, beta, gamma and Mu waves.
Since the highest frequency that the waves are presented in is 100Hz, sampling with modern techniques presents no challenge. Therefore assuming that, the sampling will be handled by an analogue-digital converter that can be set to give a 16bit vector as output per sample. I choose to have 10 inputs (one input per sample) on each of the neurons of the input layer. Since there are 5 waveforms, but the number of electrodes set on the scalp to detect the waves can be greater than the number of waves. I assume that 10 virtual electrodes will be sufficient for purpose of the simulation. Thus there will be 10 neurons on the input layer and one neuron with 10 inputs on the output layer.
I choose to use the sigmoid as the activation function for the input layer neurons, as for the output layer neuron, since I want my ANN to give an output signal of digital "HIGH" or "LOW" and I am not going to implement any learning algorithms for my design, but rather hard-code it to identify a set of values that represent samples taken from a signal, I decide to use the step activation function.
The structure and operation is displayed on Figure 14.

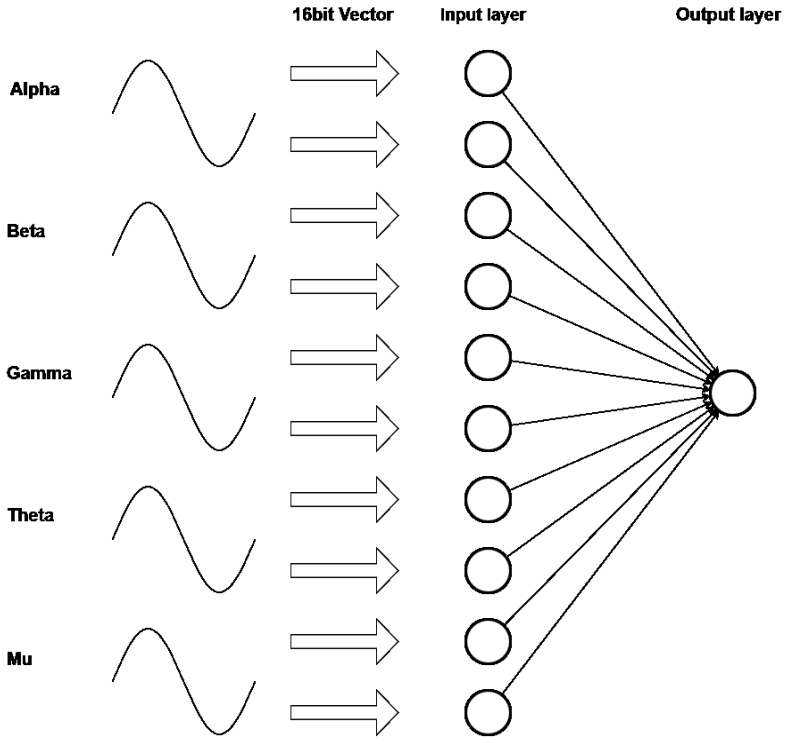


Figure 14. ANN structure

5 ANN Realization

When selecting the appropriate means to realize my ANN, I look for the presence of the following properties: size for portability, scalability to accommodate possible ANN growth, peripherals or the possibility to implement signal filtering as I have different frequency domains to work with, availability on the free market, price and power consumption.

After taking all of the above into consideration I choose to realize my design using VHDL.
VHDL allows my design to be synthesised on a FPGA chip. An FPGA may not be the most cost efficient piece of hardware as prototyping boards like the “Nexys 4 Artix-7 FPGA Trainer Board” or “Zybo Zynq-7000 ARM/FPGA SoC Trainer Board” that have all the peripherals necessary for a complete product, may cost from 189 USD to 320 USD.
However, since at such an early stage of the development of my ANN it is difficult to judge how much hardware recourse it will require, I look upon scalability as the defining factor in realizing my design.
To write the code and simulate the design I will use the Vivado Design Suite - HLx Edition - 2016.1. This software is uses for synthesis and analysis of hardware description language designs.

5.1 VHDL based ANN design

5.1.1 Weighted sum calculator

Every single neuron that will be implemented in my design has one thing in common: the “Weighted sum calculator” that is described in Equation (1).
Thus I begin by describing the operation of the weighted sum calculator in VHDL code Figure 15 to Figure 18.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity SKan_DELTA is
    Port ( SKanDELTA_EN : in  STD_LOGIC;
           RGin_x1 : in std_logic_vector (15 downto 0);
           RGin_x2 : in std_logic_vector (15 downto 0);
           RGin_x3 : in std_logic_vector (15 downto 0);
           RGin_x4 : in std_logic_vector (15 downto 0);
           RGin_x5 : in std_logic_vector (15 downto 0);
           RGin_x6 : in std_logic_vector (15 downto 0);
           RGin_x7 : in std_logic_vector (15 downto 0);
           RGin_x8 : in std_logic_vector (15 downto 0);
           RGin_x9 : in std_logic_vector (15 downto 0);
           RGin_x10 : in std_logic_vector (15 downto 0);
           SKanDELTA_NET : out STD_LOGIC_VECTOR (31 downto 0);
           clk : in STD_LOGIC);
end SKan_DELTA;
architecture Behavioral of SKan_DELTA is
-----------------------Real Value convert--------------------
function real_convert (real_in: real) return std_logic_vector;
function real_convert (real_in: real) return std_logic_vector is
variable std_out: std_logic_vector(15 downto 0);
begin
std_out := std_logic_vector(to_signed(integer(real_in/0.000153),16));
return std_out;
end real_convert;
type state is (multiplyXW1, multiplyXW2, multiplyXW3, multiplyXW4,
multiplyXW5, multiplyXW6, multiplyXW7, multiplyXW8, multiplyXW9,
multiplyXW10, addALL);
signal current_state : State := multiplyXW1;
signal next_state: State;
type alu_select is (multiply1, multiply2, multiply3, multiply4, multiply5,
multiply6, multiply7, multiply8, multiply9, multiply10, add);
constant RG_w1: std_logic_vector(15 downto 0):= real_convert(-0.5);
constant RG_w2: std_logic_vector(15 downto 0):= real_convert(0.4);
constant RG_w3: std_logic_vector(15 downto 0):= real_convert(0.5);
constant RG_w4: std_logic_vector(15 downto 0):= real_convert(0.3);
constant RG_w5: std_logic_vector(15 downto 0):= real_convert(-1.1);
constant RG_w6: std_logic_vector(15 downto 0):= real_convert(1.0);
constant RG_w7: std_logic_vector(15 downto 0):= real_convert(0.52);
constant RG_w8: std_logic_vector(15 downto 0):= real_convert(0.2);
constant RG_w9: std_logic_vector(15 downto 0):= real_convert(0.07);
constant RG_w10: std_logic_vector(15 downto 0):= real_convert(0.8);
signal RG1_EN, RG2_EN, RG3_EN, RG4_EN, RG5_EN,
       RG6_EN, RG7_EN, RG8_EN, RG9_EN, RG10_EN, RGadd_EN: std_logic ;
signal alu_operations : alu_select;
signal ALUresultM1, ALUresultM2, ALUresultM3, ALUresultM4, ALUresultM5,
       ALUresultM6, ALUresultM7, ALUresultM8, ALUresultM9, ALUresultM10,
       ALUresultADD: signed (31 downto 0);
```

Figure 15. Weighted sum calculator part 1.

```
signal Mult_result_1, Mult_result_2, Mult_result_3, Mult_result_4,
Mult_result_5, Mult_result_6, Mult_result_7, Mult_result_8, Mult_result_9,
Mult_result_10,NET: signed (31 downto 0);
begin
process (clk)
  begin
   if clk'event and clk = '1' then
    current_state <= next_state;
   end if;
end process;
process (current_state, SKanDELTA_EN )
begin
next_state <= current_state;
alu_operations <= multiply1;
case current_state is
when multiplyXW1 =>
        if SKanDELTA_EN = '1' then
        alu_operations <= multiply1;
        RG1_EN <= '1';
        next_state <= multiplyXW2;
        elsif SKanDELTA_EN = '0' then
        next_state <= multiplyXW1;
        RG1_EN <= '0';
        end if;
when multiplyXW2 =>
        alu_operations <= multiply2;
        RG2_EN <= '1';
        next_state <= multiplyXW3;
when multiplyXW3 =>
        alu_operations <= multiply3;
        RG3_EN <= '1';
        next_state <= multiplyXW4;
when multiplyXW4 =>
        alu_operations <= multiply4;
        RG4_EN <= '1';
        next_state <= multiplyXW5;
when multiplyXW5 =>
        alu_operations <= multiply5;
        RG5_EN <= '1';
        next_state <= multiplyXW6;
when multiplyXW6 =>
        alu_operations <= multiply6;
        RG6_EN <= '1';
        next_state <= multiplyXW7;
when multiplyXW7 =>
        alu_operations <= multiply7;
        RG7_EN <= '1';
        next_state <= multiplyXW8;
```

Figure 16. Weighted sum calculator part 2.

```
when multiplyXW8 =>
        alu_operations <= multiply8;
        RG8_EN <= '1';
        next_state <= multiplyXW9;
when multiplyXW9 =>
        alu_operations <= multiply9;
        RG9_EN <= '1';
        next_state <= multiplyXW10;
when multiplyXW10=>
        alu_operations <= multiply10;
        RG10_EN <= '1';
        next_state <= addALL;
when addALL =>
        alu_operations <= add;
        RGadd_EN <= '1';
        next_state <= multiplyXW1;
end case;
end process;

process ( alu_operations, RGin_x1, RGin_x2, RGin_x3, RGin_x4, RGin_x5,
RGin_x6, RGin_x7, RGin_x8, RGin_x9, RGin_x10, Mult_result_1, Mult_result_2,
Mult_result_3, Mult_result_4, Mult_result_5,Mult_result_6, Mult_result_7,
Mult_result_8, Mult_result_9, Mult_result_10)
begin
case alu_operations is
 when multiply1 => ALUresultM1 <= (SIGNED(RGin_x1) * SIGNED(RG_w1));
 when multiply2 => ALUresultM2 <= (SIGNED(RGin_x2) * SIGNED(RG_w2));
 when multiply3 => ALUresultM3 <= (SIGNED(RGin_x3) * SIGNED(RG_w3));
 when multiply4 => ALUresultM4 <= (SIGNED(RGin_x4) * SIGNED(RG_w4));
 when multiply5 => ALUresultM5 <= (SIGNED(RGin_x5) * SIGNED(RG_w5));
 when multiply6 => ALUresultM6 <= (SIGNED(RGin_x6) * SIGNED(RG_w6));
 when multiply7 => ALUresultM7 <= (SIGNED(RGin_x7) * SIGNED(RG_w7));
 when multiply8 => ALUresultM8 <= (SIGNED(RGin_x8) * SIGNED(RG_w8));
 when multiply9 => ALUresultM9 <= (SIGNED(RGin_x9) * SIGNED(RG_w9));
 when multiply10 => ALUresultM10 <= (SIGNED(RGin_x10) * SIGNED(RG_w10));
 when add => ALUresultADD <= (SIGNED(Mult_result_1) + SIGNED(Mult_result_2)
+ SIGNED(Mult_result_3) + SIGNED(Mult_result_4) + SIGNED(Mult_result_5) +
SIGNED(Mult_result_6) + SIGNED(Mult_result_7) + SIGNED(Mult_result_8)
+ SIGNED(Mult_result_9) + SIGNED(Mult_result_10));
end case;
end process;
```

Figure 17. Weighted sum calculator part 3.

```
process (clk)
begin
if clk'event and clk = '1' then
if RG1_EN = '1' then
Mult_result_1 <= ALUresultM1;
end if;
if RG2_EN = '1' then
Mult_result_2 <= ALUresultM2;
end if;
if RG3_EN = '1' then
Mult_result_3 <= ALUresultM3;
end if;
if RG4_EN = '1' then
Mult_result_4 <= ALUresultM4;
end if;
if RG5_EN = '1' then
Mult_result_5 <= ALUresultM5;
end if;
if RG6_EN = '1' then
Mult_result_6 <= ALUresultM6;
end if;
if RG7_EN = '1' then
Mult_result_7 <= ALUresultM7;
end if;
if RG8_EN = '1' then
Mult_result_8 <= ALUresultM8;
end if;
if RG9_EN = '1' then
Mult_result_9 <= ALUresultM9;
end if;
if RG10_EN = '1' then
Mult_result_10 <= ALUresultM10;
end if;
if RGadd_EN = '1' then
NET <= ALUresultADD;
end if;
end if;
end process;
SKanDELTA_NET <= STD_LOGIC_VECTOR(NET);
end Behavioral;
```

Figure 18. Weighted sum calculator part 4.

In the code above I implement a DPU unit (Figure 20), which has 11 ALU units and 11 registers. The DPU is controlled by a Moore type FSM (Figure 19), whose outputs are determined only by its current state. States in sequence 1 through 10 of the FSM send a signal to the appropriate ALU unit, to perform a multiplication operation of the input signals by the corresponding stored value that represents the weight coefficient. At the same time the FSM activates the appropriate register, where the multiplication result is

stored. Once the multiplication operations are done, state 11 of the FSM instructs the 11^{th} ALU unit to perform an addition operation of the results stored in registers 1 to 10 and store the addition result in register 11.

The result stored in register 11 is a 32bit vector that is presented as the output or NET value. Operation of the FSM is displayed on Figure 19.

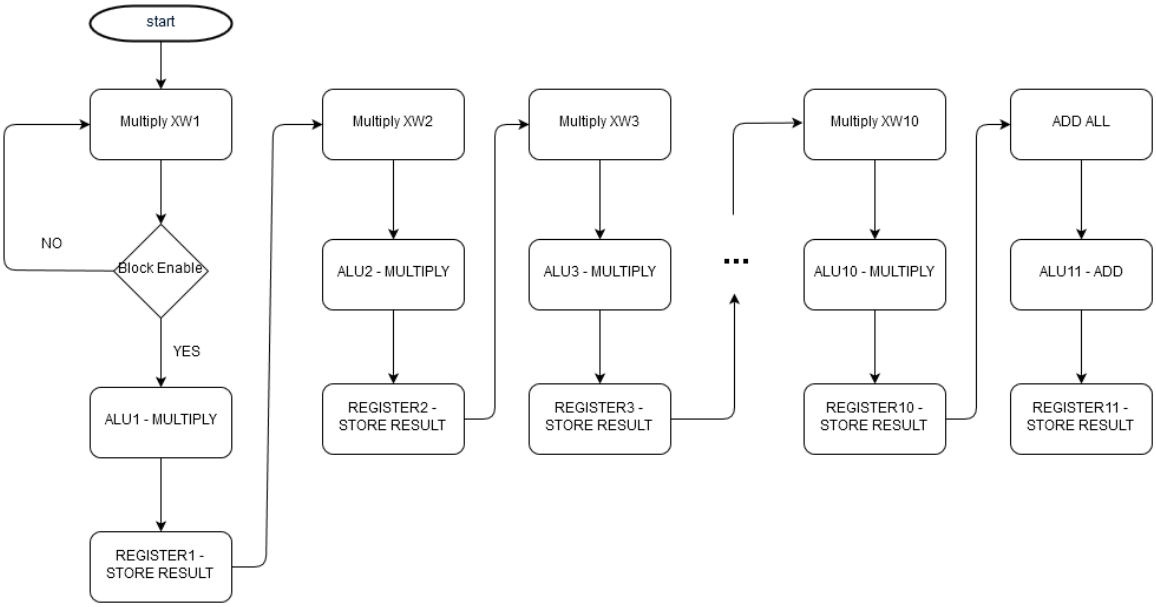


Figure 19. FSM operation

The operation of the DPU unit is displayed on Figure 20.

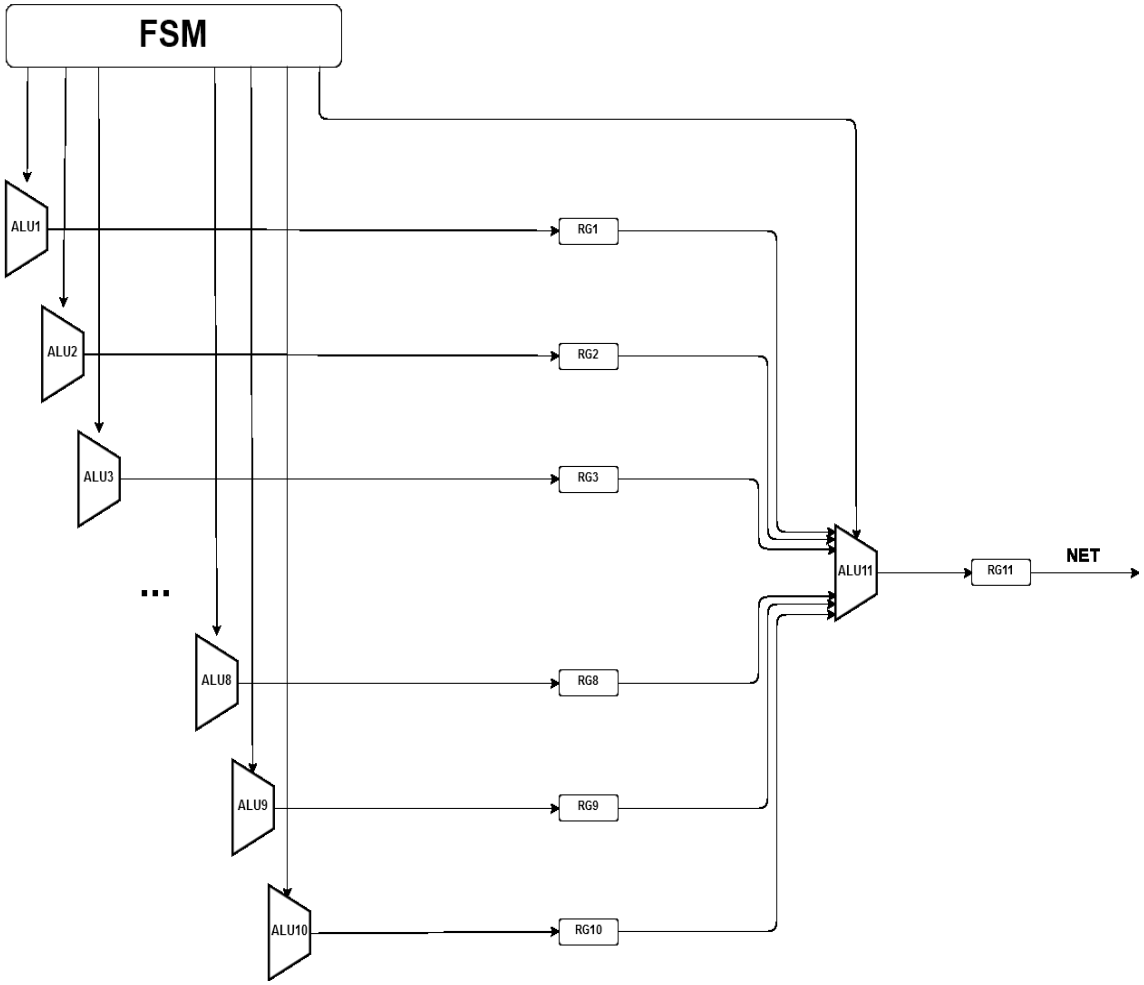


Figure 20. DPU unit

To test the VHDL code, I have written a test bench code where I use the same random values as weight coefficients for inputs: (-0.5, 0.4, 0.5, 0.3, -1.1, 1.0, 0.52, 0.7, 0.07, 0.8). The test bench code is displayed on Figure 21 to Figure 23.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity TB2 is
end TB2;
architecture Behavioral of TB2 is
COMPONENT SKan_DELTA
PORT (
           SKanDELTA_EN : in  STD_LOGIC;
           RGin_x1 : in std_logic_vector (15 downto 0);
           RGin_x2 : in std_logic_vector (15 downto 0);
           RGin_x3 : in std_logic_vector (15 downto 0);
           RGin_x4 : in std_logic_vector (15 downto 0);
           RGin_x5 : in std_logic_vector (15 downto 0);
           RGin_x6 : in std_logic_vector (15 downto 0);
           RGin_x7 : in std_logic_vector (15 downto 0);
           RGin_x8 : in std_logic_vector (15 downto 0);
           RGin_x9 : in std_logic_vector (15 downto 0);
           RGin_x10 : in std_logic_vector (15 downto 0);
           SKanDELTA_NET : out STD_LOGIC_VECTOR (31 downto 0);
           clk : in STD_LOGIC
             );
END COMPONENT;
-----------------------Real Value convert--------------------
function real_convert (real_in: real) return std_logic_vector;
function real_convert (real_in: real) return std_logic_vector is
variable std_out: std_logic_vector(15 downto 0);
begin
std_out := std_logic_vector(to_signed(integer(real_in/0.000153),16));
return std_out;
end real_convert;
-------------------Vector Convert-------------------------
function std_convert (std_in: STD_LOGIC_VECTOR) return real;
function std_convert (std_in: STD_LOGIC_VECTOR) return real is
variable real_out: real;
begin
real_out := 0.000153 * 0.000153 * real(to_integer(SIGNED(std_in)));
return real_out;
end std_convert;
signal out_real: real;
```

Figure 21. Test bench code part 1.

```
--inputs
signal SKanDELTA_EN :  std_logic;
signal RGin_x1 : std_logic_vector (15 downto 0);
signal RGin_x2 : std_logic_vector (15 downto 0);
signal RGin_x3 : std_logic_vector (15 downto 0);
signal RGin_x4 : std_logic_vector (15 downto 0);
signal RGin_x5 : std_logic_vector (15 downto 0);
signal RGin_x6 : std_logic_vector (15 downto 0);
signal RGin_x7 : std_logic_vector (15 downto 0);
signal RGin_x8 : std_logic_vector (15 downto 0);
signal RGin_x9 : std_logic_vector (15 downto 0);
signal RGin_x10 : std_logic_vector (15 downto 0);
signal clk : STD_LOGIC;
--outputs
signal SKanDELTA_NET : STD_LOGIC_VECTOR (31 downto 0);
constant clk_period : time := 10 ns;
begin
--instantiate the unit under test (UUT)
uut: SKan_DELTA PORT MAP (
        clk => clk,
        RGin_x1 =>  RGin_x1 ,
        RGin_x2 =>  RGin_x2 ,
        RGin_x3 =>  RGin_x3 ,
        RGin_x4 =>  RGin_x4 ,
        RGin_x5 =>  RGin_x5 ,
        RGin_x6 =>  RGin_x6 ,
        RGin_x7 =>  RGin_x7 ,
        RGin_x8 =>  RGin_x8 ,
        RGin_x9 =>  RGin_x9 ,
        RGin_x10 => RGin_x10 ,
        SKanDELTA_EN => SKanDELTA_EN,
        SKanDELTA_NET => SKanDELTA_NET);
```

Figure 22. Test bench code part 2.

```
out_real <= std_convert(SKanDELTA_NET);
clk_process : process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
   end process;
stim_proc: process
begin
wait for 50 ns;
RGin_x1 <= real_convert(-0.5);
RGin_x2 <= real_convert(0.4);
RGin_x3 <= real_convert(0.5);
RGin_x4 <= real_convert(0.3);
RGin_x5 <= real_convert(-1.1);
RGin_x6 <= real_convert(1.0);
RGin_x7 <= real_convert(0.52);
RGin_x8 <= real_convert(0.2);
RGin_x9 <= real_convert(0.07);
RGin_x10 <= real_convert(0.8);
SKanDELTA_EN <= '1';
end process;
end Behavioral;
```

Figure 23. Test bench code part 3.

According to the Equation 1, the output NET must be equal to:

(-0.5*-0.5) + (0.4*0.4) + (0.5*0.5) + (0.3*03) + (-1.1*-1.1) + (1.0*1.0) + (0.52*0.52) + (0.7*0.7) + (0.07*0.07) + (0.8*0.8) = 3.9153 (NET)

After running the simulation (Figure 24), we can see the highlighted signals, where "SKanDELTA_NET" is the 32bit binary representation of the output value NET and the signal "out_real" is the binary NET value converted to a "real" type number, we can also see that due to the implemented conversion the output is correct to the one thousandth.

The simulation result also shows that it took 155nS for the code to calculate the result. This time frame consists of a waiting period of 50nS and 11 clock cycles, since the FSM has 11 states, changing them sequentially on every "high" clock event.

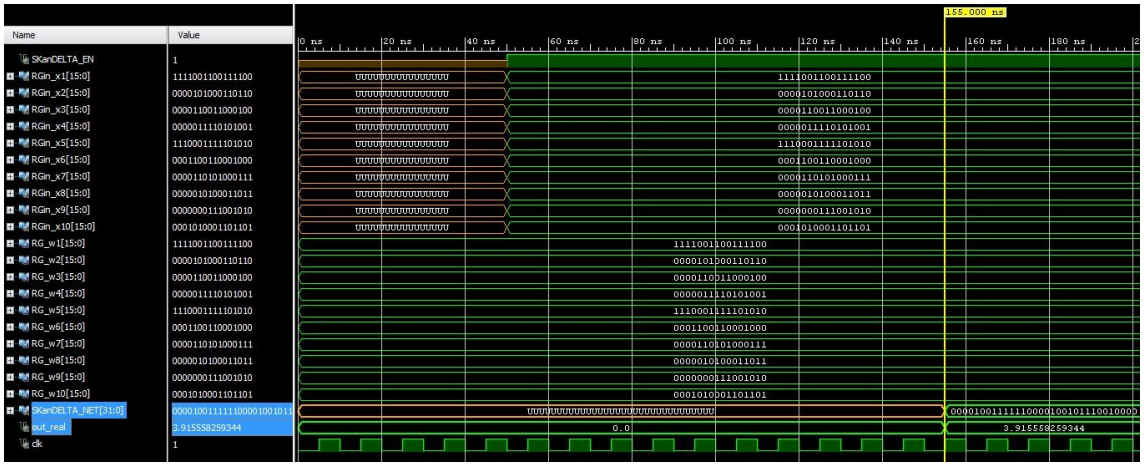


Figure 24. Weighted sum calculator simulation result.

To finalise the design of the weighted sum calculator and to be able to use it more conveniently in the further, I use the Vivado IP packager feature to make a building block (Figure 25), out of the VHDL code.

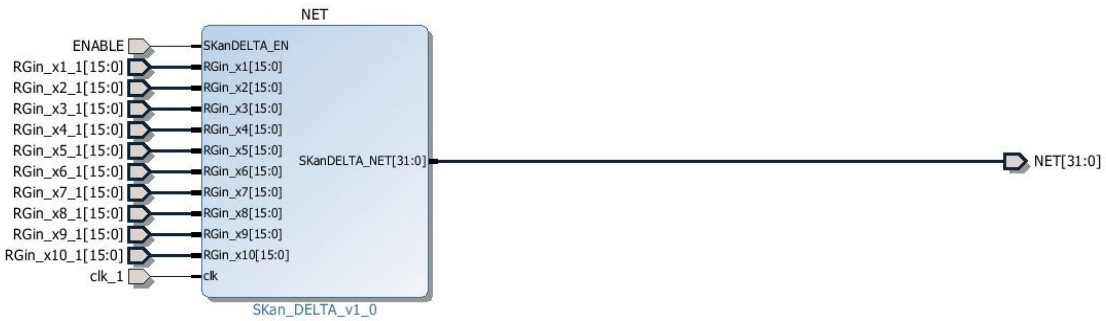


Figure 25. Weighted sum calculator IP block.

5.1.2 The sigmoid function realization

To realize the sigmoid function in VHDL, I choose to use a LUT of a sigmoid function generated in Matlab.

To do so I use the Matlab code displayed on Figure 26.

```
x=-5:0.2:5;
y=logsig(x);
  plot(x,y)
```

Figure 26. Matlab sigmoid generation function

The resulting plot can be seen on Figure 27.

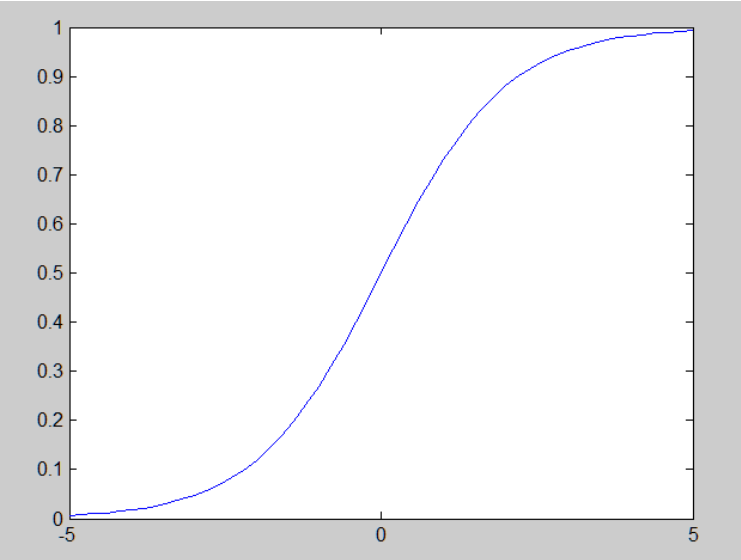


Figure 27. Matlab sigmoid plot

The code in Figure 26 shows that the "x" values are in range from -5 to 5 with a step of 0.2. This step was selected with the goal to make the VHDL realization more simplified by reducing the volume of the LUT. The "logsig" function is used to plot the "y" values in the range from 0 to 1, so the entire plot corresponds to the sigmoid activation plot on Figure 3.

To be able to see the resulting plot more clearly, I slightly adjust the Matlab code (Figure 28).

```
x=-5:0.2:5;
y=logsig(x);
 stairs(x,y)
```

Figure 28. Adjusted Matlab sigmoid code.

From the adjusted code I get the following plot (Figure 29).

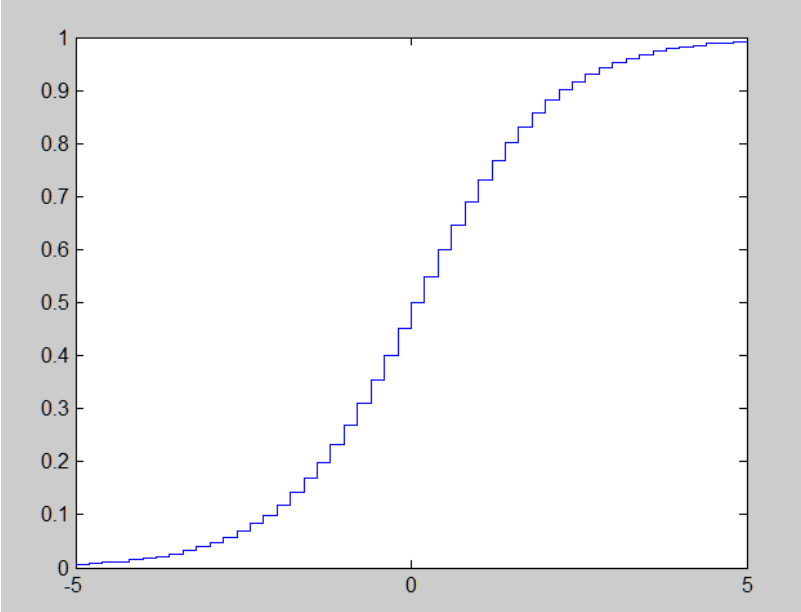


Figure 29. Matlab sigmoid "stairs" plot.

From this plot I extract the "x" and "y" values (Figure 30). That I will use to implement a LUT of a sigmoid function on Figure 29 in VHDL.

X	Y		
-5	0,00971	0,2	0,598058
-4,8	0,01051	0,4	0,685695
-4,6	0,011412	0,6	0,757248
-4,4	0,012434	0,8	0,812348
-4,2	0,013597	1	0,853553
-4	0,014929	1,2	0,884111
-3,8	0,016463	1,4	0,906867
-3,6	0,018241	1,6	0,923999
-3,4	0,020317	1,8	0,937079
-3,2	0,02276	2	0,947214
-3	0,025658	2,2	0,955183
-2,8	0,029129	2,4	0,961538
-2,6	0,033327	2,6	0,966673
-2,4	0,038462	2,8	0,970871
-2,2	0,044817	3	0,974342
-2	0,052786	3,2	0,97724
-1,8	0,062921	3,4	0,979683
-1,6	0,076001	3,6	0,981759
-1,4	0,093133	3,8	0,983537
-1,2	0,115889	4	0,985071
-1	0,146447	4,2	0,986403
-0,8	0,187652	4,4	0,987566
-0,6	0,242752	4,6	0,988588
-0,2	0,401942	4,8	0,98949
0	0,5	5	0,99029

Figure 30. "x" and "y" extracted values.

The following VHDL code is the result of implementing the sigmoid activation function as a LUT and is displayed on Figure 31 to Figure 35.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library UNISIM;
use UNISIM.VComponents.all;
entity SKan_Sigmoid_TB is
        Port (
               SIGin : in STD_LOGIC_VECTOR (31 downto 0);
               SIGout : out STD_LOGIC_VECTOR (31 downto 0)
              );
end SKan_Sigmoid_TB;
architecture Behavioral of SKan_Sigmoid_TB is
---------------------Vector to Integer Convert-------------
function int_convert (std_in: signed) return integer;
function int_convert (std_in: signed) return integer is
variable int_out: integer;
begin
int_out := to_integer(signed(std_in));
return int_out;
end int_convert;
-----------------------Real Value convert--------------------
function real_convert (real_in: real) return integer;
function real_convert (real_in: real) return integer is
variable int_out: integer;
begin
int_out := integer(real_in/(0.000153 * 0.000153));
return int_out;
end real_convert;
constant real_number: real:=3.915558;
constant real_number_to_int: integer:=real_convert(real_number);
constant real_number_to_signed: signed (31 downto
0):=to_signed(real_number_to_int,32);
```

Figure 31. VHDL sigmoid LUT code part 1.

```
constant n5_0: integer:= real_convert(-5.0);
constant n4_8: integer:= real_convert(-4.8);
constant n4_6: integer:= real_convert(-4.6);
constant n4_4: integer:= real_convert(-4.4);
constant n4_2: integer:= real_convert(-4.2);
constant n4_0: integer:= real_convert(-4.0);
constant n3_8: integer:= real_convert(-3.8);
constant n3_6: integer:= real_convert(-3.6);
constant n3_4: integer:= real_convert(-3.4);
constant n3_2: integer:= real_convert(-3.2);
constant n3_0: integer:= real_convert(-3.0);
constant n2_8: integer:= real_convert(-2.8);
constant n2_6: integer:= real_convert(-2.6);
constant n2_4: integer:= real_convert(-2.4);
constant n2_2: integer:= real_convert(-2.2);
constant n2_0: integer:= real_convert(-2.0);
constant n1_8: integer:= real_convert(-1.8);
constant n1_6: integer:= real_convert(-1.6);
constant n1_4: integer:= real_convert(-1.4);
constant n1_2: integer:= real_convert(-1.2);
constant n1_0: integer:= real_convert(-1.0);
constant n0_8: integer:= real_convert(-0.8);
constant n0_6: integer:= real_convert(-0.6);
constant n0_4: integer:= real_convert(-0.4);
constant n0_2: integer:= real_convert(-0.2);
constant n0_0: integer:= real_convert(-0.0);
constant p0_0: integer:= real_convert(0.0);
constant p0_2: integer:= real_convert(0.2);
constant p0_4: integer:= real_convert(0.4);
constant p0_6: integer:= real_convert(0.6);
constant p0_8: integer:= real_convert(0.8);
constant p1_0: integer:= real_convert(1.0);
constant p1_2: integer:= real_convert(1.2);
constant p1_4: integer:= real_convert(1.4);
constant p1_6: integer:= real_convert(1.6);
constant p1_8: integer:= real_convert(1.8);
constant p2_0: integer:= real_convert(2.0);
constant p2_2: integer:= real_convert(2.2);
constant p2_4: integer:= real_convert(2.4);
constant p2_6: integer:= real_convert(2.6);
constant p2_8: integer:= real_convert(2.8);
constant p3_0: integer:= real_convert(3.0);
constant p3_2: integer:= real_convert(3.2);
constant p3_4: integer:= real_convert(3.4);
constant p3_6: integer:= real_convert(3.6);
constant p3_8: integer:= real_convert(3.8);
```

Figure 32. VHDL sigmoid LUT code part 2.

```
constant p4_0: integer:= real_convert(4.0);
constant p4_2: integer:= real_convert(4.2);
constant p4_4: integer:= real_convert(4.4);
constant p4_6: integer:= real_convert(4.6);
constant p4_8: integer:= real_convert(4.8);
constant p5_0: integer:= real_convert(5.0);
constant net0_009: signed(31 downto 0):= to_signed(real_convert(0.009),32);
constant net0_010: signed(31 downto 0):= to_signed(real_convert(0.010),32);
constant net0_011: signed(31 downto 0):= to_signed(real_convert(0.011),32);
constant net0_012: signed(31 downto 0):= to_signed(real_convert(0.012),32);
constant net0_013: signed(31 downto 0):= to_signed(real_convert(0.013),32);
constant net0_014: signed(31 downto 0):= to_signed(real_convert(0.014),32);
constant net0_016: signed(31 downto 0):= to_signed(real_convert(0.016),32);
constant net0_018: signed(31 downto 0):= to_signed(real_convert(0.018),32);
constant net0_020: signed(31 downto 0):= to_signed(real_convert(0.020),32);
constant net0_022: signed(31 downto 0):= to_signed(real_convert(0.022),32);
constant net0_025: signed(31 downto 0):= to_signed(real_convert(0.025),32);
constant net0_029: signed(31 downto 0):= to_signed(real_convert(0.029),32);
constant net0_033: signed(31 downto 0):= to_signed(real_convert(0.033),32);
constant net0_038: signed(31 downto 0):= to_signed(real_convert(0.038),32);
constant net0_044: signed(31 downto 0):= to_signed(real_convert(0.044),32);
constant net0_052: signed(31 downto 0):= to_signed(real_convert(0.052),32);
constant net0_062: signed(31 downto 0):= to_signed(real_convert(0.062),32);
constant net0_076: signed(31 downto 0):= to_signed(real_convert(0.076),32);
constant net0_093: signed(31 downto 0):= to_signed(real_convert(0.093),32);
constant net0_115: signed(31 downto 0):= to_signed(real_convert(0.115),32);
constant net0_146: signed(31 downto 0):= to_signed(real_convert(0.146),32);
constant net0_187: signed(31 downto 0):= to_signed(real_convert(0.187),32);
constant net0_242: signed(31 downto 0):= to_signed(real_convert(0.242),32);
constant net0_314: signed(31 downto 0):= to_signed(real_convert(0.314),32);
constant net0_401: signed(31 downto 0):= to_signed(real_convert(0.401),32);
constant net0_500: signed(31 downto 0):= to_signed(real_convert(0.500),32);
constant net0_598: signed(31 downto 0):= to_signed(real_convert(0.598),32);
constant net0_685: signed(31 downto 0):= to_signed(real_convert(0.685),32);
constant net0_757: signed(31 downto 0):= to_signed(real_convert(0.757),32);
constant net0_812: signed(31 downto 0):= to_signed(real_convert(0.812),32);
constant net0_853: signed(31 downto 0):= to_signed(real_convert(0.853),32);
constant net0_884: signed(31 downto 0):= to_signed(real_convert(0.884),32);
constant net0_906: signed(31 downto 0):= to_signed(real_convert(0.906),32);
constant net0_924: signed(31 downto 0):= to_signed(real_convert(0.924),32);
constant net0_937: signed(31 downto 0):= to_signed(real_convert(0.937),32);
constant net0_947: signed(31 downto 0):= to_signed(real_convert(0.947),32);
constant net0_955: signed(31 downto 0):= to_signed(real_convert(0.955),32);
constant net0_961: signed(31 downto 0):= to_signed(real_convert(0.961),32);
constant net0_966: signed(31 downto 0):= to_signed(real_convert(0.966),32);
```

Figure 33. VHDL sigmoid LUT code part 3.

```
constant net0_970: signed(31 downto 0):= to_signed(real_convert(0.970),32);
constant net0_974: signed(31 downto 0):= to_signed(real_convert(0.974),32);
constant net0_977: signed(31 downto 0):= to_signed(real_convert(0.977),32);
constant net0_979: signed(31 downto 0):= to_signed(real_convert(0.979),32);
constant net0_981: signed(31 downto 0):= to_signed(real_convert(0.981),32);
constant net0_983: signed(31 downto 0):= to_signed(real_convert(0.983),32);
constant net0_985: signed(31 downto 0):= to_signed(real_convert(0.985),32);
constant net0_986: signed(31 downto 0):= to_signed(real_convert(0.986),32);
constant net0_987: signed(31 downto 0):= to_signed(real_convert(0.987),32);
constant net0_988: signed(31 downto 0):= to_signed(real_convert(0.988),32);
constant net0_989: signed(31 downto 0):= to_signed(real_convert(0.989),32);
constant net0_990: signed(31 downto 0):= to_signed(real_convert(0.990),32);
SIGNAL SIGout_signed : signed (31 downto 0);
signal SIGin_integer: integer;
begin
SIGin_integer <= to_integer(SIGNED(SIGin));
process (SIGin_integer)
begin
case SIGin_integer is
    when n5_0 to n4_8-1 => SIGout_signed <= net0_009;
    when n4_8 to n4_6-1 => SIGout_signed <= net0_010;
    when n4_6 to n4_4-1=> SIGout_signed <= net0_011;
    when n4_4 to n4_2-1=> SIGout_signed <= net0_012;
    when n4_2 to n4_0-1=> SIGout_signed <= net0_013;
    when n4_0 to n3_8-1=> SIGout_signed <= net0_014;
    when n3_8 to n3_6-1=> SIGout_signed <= net0_016;
    when n3_6 to n3_4-1=> SIGout_signed <= net0_018;
    when n3_4 to n3_2-1=> SIGout_signed <= net0_020;
    when n3_2 to n3_0-1=> SIGout_signed <= net0_022;
    when n3_0 to n2_8-1=> SIGout_signed <= net0_025;
    when n2_8 to n2_6-1=> SIGout_signed <= net0_029;
    when n2_6 to n2_4-1=> SIGout_signed <= net0_033;
    when n2_4 to n2_2-1=> SIGout_signed <= net0_038;
    when n2_2 to n2_0-1=> SIGout_signed <= net0_044;
    when n2_0 to n1_8-1=> SIGout_signed <= net0_052;
    when n1_8 to n1_6-1=> SIGout_signed <= net0_062;
    when n1_6 to n1_4-1=> SIGout_signed <= net0_076;
    when n1_4 to n1_2-1=> SIGout_signed <= net0_093;
    when n1_2 to n1_0-1=> SIGout_signed <= net0_115;
    when n1_0 to n0_8-1=> SIGout_signed <= net0_146;
    when n0_8 to n0_6-1 => SIGout_signed <= net0_187;
    when n0_6 to n0_4-1 => SIGout_signed <= net0_242;
    when n0_4 to n0_2-1 => SIGout_signed <= net0_401;
    when n0_2 to n0_0-1 => SIGout_signed <= net0_401;
```

Figure 34. VHDL sigmoid LUT code part 4.

```
when p0_2-1 downto p0_0 => SIGout_signed <= net0_500;
    when p0_4-1 downto p0_2 => SIGout_signed <= net0_598;
    when p0_6-1 downto p0_4 => SIGout_signed <= net0_685;
    when p0_8-1 downto p0_6 => SIGout_signed <= net0_757;
    when p1_0-1 downto p0_8 => SIGout_signed <= net0_812;
    when p1_2-1 downto p1_0 => SIGout_signed <= net0_853;
    when p1_4-1 downto p1_2 => SIGout_signed <= net0_884;
    when p1_6-1 downto p1_4 => SIGout_signed <= net0_906;
    when p1_8-1 downto p1_6 => SIGout_signed <= net0_924;
    when p2_0-1 downto p1_8 => SIGout_signed <= net0_937;
    when p2_2-1 downto p2_0 => SIGout_signed <= net0_947;
    when p2_4-1 downto p2_2 => SIGout_signed <= net0_955;
    when p2_6-1 downto p2_4 => SIGout_signed <= net0_961;
    when p2_8-1 downto p2_6 => SIGout_signed <= net0_966;
    when p3_0-1 downto p2_8 => SIGout_signed <= net0_970;
    when p3_2-1 downto p3_0 => SIGout_signed <= net0_974;
    when p3_4-1 downto p3_2 => SIGout_signed <= net0_977;
    when p3_6-1 downto p3_4 => SIGout_signed <= net0_979;
    when p3_8-1 downto p3_6 => SIGout_signed <= net0_981;
    when p4_0-1 downto p3_8 => SIGout_signed <= net0_983;
    when p4_2-1 downto p4_0 => SIGout_signed <= net0_985;
    when p4_4-1 downto p4_2 => SIGout_signed <= net0_986;
    when p4_6-1 downto p4_4 => SIGout_signed <= net0_987;
    when p4_8-1 downto p4_6 => SIGout_signed <= net0_988;
    when p5_0-1 downto p4_8 => SIGout_signed <= net0_989;
    when others => SIGout_signed <= (others => '0');
end case;
end process;
SIGout <= std_logic_vector(SIGout_signed);
end Behavioral;
```

Figure 35. VHDL sigmoid LUT code part 5.

In order to test the VHDL code, I wrote a test bench code (Figure 36). This test bench code contains test values (3.91) and (-4.8) that the sigmoid function would get as an input from the weighted sum calculator block displayed on Figure 25.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity TB2 is
end TB2;
architecture Behavioral of TB2 is
COMPONENT SKan_Sigmoid_TB
PORT ( SIGin : in STD_LOGIC_VECTOR (31 downto 0);
       SIGout : out STD_LOGIC_VECTOR (31 downto 0));
END COMPONENT;
-- inputs
signal SIGin : STD_LOGIC_VECTOR (31 downto 0):=(OTHERS =>'0');
--outputs
signal SIGout : STD_LOGIC_VECTOR (31 downto 0);
constant clk_period : time := 10 ns;
function real_convert (real_in: real) return STD_LOGIC_VECTOR;
function real_convert (real_in: real) return STD_LOGIC_VECTOR is
variable std_out: STD_LOGIC_VECTOR(31 downto 0);
begin
std_out :=
std_logic_vector(signed(to_signed(integer(real_in/(0.000153*0.000153)),32)));
return std_out;
end real_convert;
-------------------Vector Convert-------------------------
function std_convert (std_in: signed) return real;
function std_convert (std_in: signed) return real is
variable real_out: real;
begin
real_out := 0.000153 * 0.000153 * real(to_integer(std_in));
return real_out;
end std_convert;
signal SIGin_real, SIGout_real: real;
begin
--instantiate the unit under test (UUT)
uut: SKan_Sigmoid_TB PORT MAP (SIGin => SIGin,
                               SIGout => SIGout);
stim_proc: process
begin
wait for 50 ns;
SIGin <= real_convert(3.91);
wait for 50 ns;
SIGin <= real_convert(-4.8);
wait;
end process;
SIGin_real <= std_convert(SIGNED(SIGin));
SIGout_real <= std_convert(SIGNED(SIGout));
end Behavioral;
```

Figure 36. VHDL sigmoid function test bench code.

After running the simulation I get the following result displayed on Figure 37.

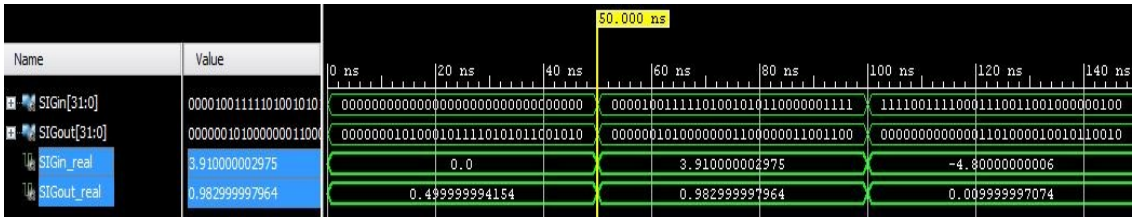


Figure 37. VHDL sigmoid simulation result.

In the simulation result we can see that, during the delay of 50nS, the input value of the highlighted signal “SIGin_real” that represents the NET value is equal to binary “0”, which in turn according to the sigmoid function (Figure 3), output equals to 0.5 that can be seen on the highlighted signal “SIGout_real”.

After the waiting period of 50nS the output value is calculated according to the input value (3.91) and after another 50nS according to the input value (-4.8).

If we compare the result to the “x” and “y” extracted values table on Figure 30. We can see that the output values of the highlighted signal “SIGout_real” on Figure 37 are close to the ones on the table in Figure 30.

The slight difference in precision, is present again, due to the binary to real conversion method used in code to display the values in the simulation result and are with in tolerable range of error.

As before, I use the IP packager feature of Vivado to finalize the design and make a building block (Figure 38), from the VHDL code that, I will be able to use later in the project.

Figure 38. Sigmoid IP block

5.1.3 Single Neuron realization

Now that I have the weighted sum calculator and the sigmoid function IP blocks, I can use them to put together a single neuron model for simulation.

The single neuron model is displayed no Figure 39.

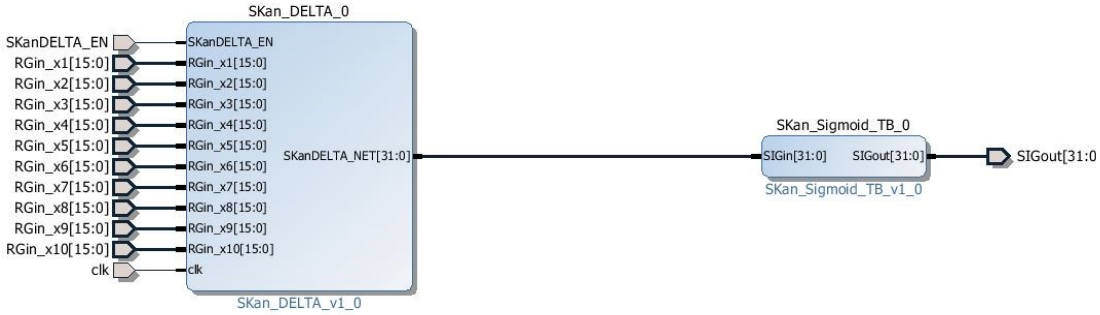


Figure 39. Single neuron IP block.

To simulate this IP block I slightly modify the test bench that was used to simulate the weighted sum calculator. The new test bench is displayed on Figure 40.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity TB2 is
end TB2;
architecture Behavioral of TB2 is
COMPONENT SKan_DELTA_BLOCK2
PORT (SKanDELTA_EN : in  STD_LOGIC;
           RGin_x1 : in std_logic_vector (15 downto 0);
           RGin_x2 : in std_logic_vector (15 downto 0);
           RGin_x3 : in std_logic_vector (15 downto 0);
           RGin_x4 : in std_logic_vector (15 downto 0);
           RGin_x5 : in std_logic_vector (15 downto 0);
           RGin_x6 : in std_logic_vector (15 downto 0);
           RGin_x7 : in std_logic_vector (15 downto 0);
           RGin_x8 : in std_logic_vector (15 downto 0);
           RGin_x9 : in std_logic_vector (15 downto 0);
           RGin_x10 : in std_logic_vector (15 downto 0);
           SIGout : out SIGNED (31 downto 0);
           clk : in STD_LOGIC );
END COMPONENT;
function real_convert (real_in: real) return std_logic_vector;
function real_convert (real_in: real) return std_logic_vector is
variable std_out: std_logic_vector(15 downto 0);
begin
std_out := std_logic_vector(to_signed(integer(real_in/0.000153),16));
return std_out;
end real_convert;
function std_convert (std_in: signed) return real;
function std_convert (std_in: signed) return real is
variable real_out: real;
begin
real_out := 0.000153 * 0.000153 * real(to_integer(std_in));
return real_out;
end std_convert;
signal sig_out_real: real;
--inputs
signal SKanDELTA_EN :  std_logic;
signal RGin_x1 : std_logic_vector (15 downto 0);
signal RGin_x2 : std_logic_vector (15 downto 0);
signal RGin_x3 : std_logic_vector (15 downto 0);
signal RGin_x4 : std_logic_vector (15 downto 0);
signal RGin_x5 : std_logic_vector (15 downto 0);
signal RGin_x6 : std_logic_vector (15 downto 0);
signal RGin_x7 : std_logic_vector (15 downto 0);
signal RGin_x8 : std_logic_vector (15 downto 0);
signal RGin_x9 : std_logic_vector (15 downto 0);
signal RGin_x10 : std_logic_vector (15 downto 0);
signal clk : STD_LOGIC;
```

Figure 40. Test bench code for neuron IP part 1.

```
--outputs
signal SIGout :  SIGNED (31 downto 0);
constant clk_period : time := 10 ns;
begin
--instantiate the unit under test (UUT)
uut: SKan_DELTA_BLOCK2 PORT MAP (
         clk => clk,
         RGin_x1 =>  RGin_x1 ,
         RGin_x2 =>  RGin_x2 ,
         RGin_x3 =>  RGin_x3 ,
         RGin_x4 =>  RGin_x4 ,
         RGin_x5 =>  RGin_x5 ,
         RGin_x6 =>  RGin_x6 ,
         RGin_x7 =>  RGin_x7 ,
         RGin_x8 =>  RGin_x8 ,
         RGin_x9 =>  RGin_x9 ,
         RGin_x10 => RGin_x10 ,
         SKanDELTA_EN => SKanDELTA_EN,
         SIGout => SIGout);
sig_out_real <= std_convert(SIGout);
clk_process : process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;
--Stimulus Process
stim_proc: process
begin
wait for 50 ns;
RGin_x1 <= real_convert(-0.5);
RGin_x2 <= real_convert(0.4);
RGin_x3 <= real_convert(0.5);
RGin_x4 <= real_convert(0.3);
RGin_x5 <= real_convert(-1.1);
RGin_x6 <= real_convert(1.0);
RGin_x7 <= real_convert(0.52);
RGin_x8 <= real_convert(0.2);
RGin_x9 <= real_convert(0.07);
RGin_x10 <= real_convert(0.8);
SKanDELTA_EN <= '1';
end process;
end Behavioral;
```

Figure 41. Test bench code for neuron IP part 2.

The simulation results displayed on Figure 42, show that the neuron behaves as expected.

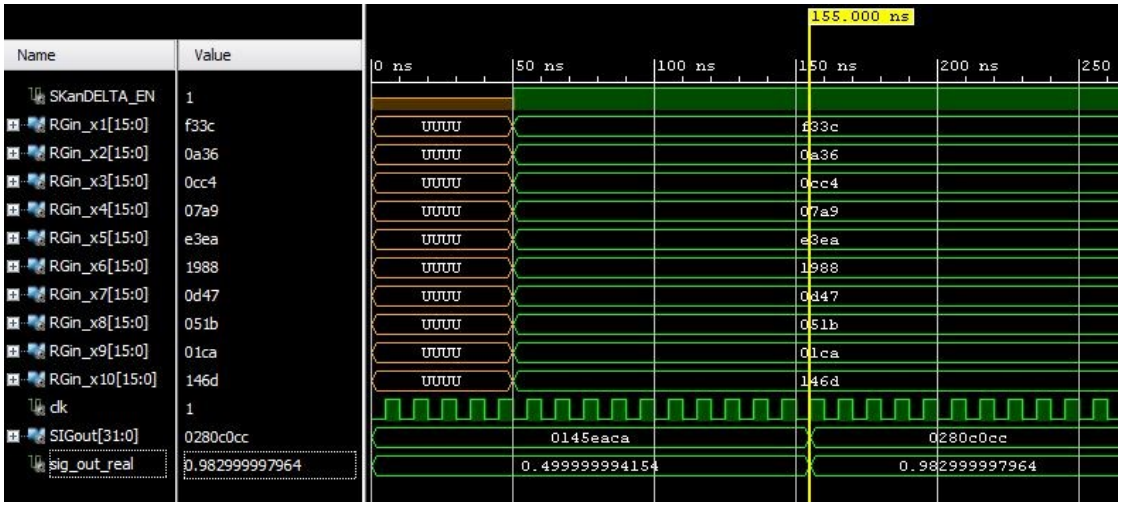


Figure 42. Neuron IP simulation result

In the simulation result we can see that the output value displayed by signal "sig_out_real" is calculated after the delay time of 50nS and the following 11 clock cycles, during which the FSM in the weighted sum calculator block is changing states sequentially.

5.1.4 Output layer neuron realization

The output layer neuron differs from the rest only by having a step activation function described in paragraph 2.1.1.

From the hardware realization point of view it is done by simply comparing the NET value to a threshold value. For the purpose of simulation I select the threshold value to be:$\theta = 2.5$

In order to realize the output layer neuron I use the HVDL code described in paragraph 5.1.1. I make the necessary adjustments by adding a compare function, the result of which is going to be a discrete "1" or "0" depending on the NET value.

The new code is displayed in Figure 43 to Figure 46.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_unsigned.all;
entity SKan_NET64 is
    Port ( SKanDELTA_EN : in  STD_LOGIC;
           ANOUT : out std_logic;
           RGin_x1 : in std_logic_vector (31 downto 0);
           RGin_x2 : in std_logic_vector (31 downto 0);
           RGin_x3 : in std_logic_vector (31 downto 0);
           RGin_x4 : in std_logic_vector (31 downto 0);
           RGin_x5 : in std_logic_vector (31 downto 0);
           RGin_x6 : in std_logic_vector (31 downto 0);
           RGin_x7 : in std_logic_vector (31 downto 0);
           RGin_x8 : in std_logic_vector (31 downto 0);
           RGin_x9 : in std_logic_vector (31 downto 0);
           RGin_x10 : in std_logic_vector (31 downto 0);
           SKanDELTA_NET : out std_logic_vector (63 downto 0);
           clk : in STD_LOGIC);
end SKan_NET64;
architecture Behavioral of SKan_NET64 is
function real_convert (real_in: real) return std_logic_vector;
function real_convert (real_in: real) return std_logic_vector is
variable std_out: std_logic_vector(31 downto 0);
begin
std_out :=
std_logic_vector(to_signed(integer(real_in/(0.000153*0.000153)),32));
return std_out;
end real_convert;
function std_convert (std_in: signed) return real;
function std_convert (std_in: signed) return real is
variable real_out: real;
begin
real_out := 0.0000023536 * real(to_integer(SIGNED(std_in(63 downto 32))));
return real_out;
end std_convert;
signal abc: real;
type state is (multiplyXW1, multiplyXW2, multiplyXW3, multiplyXW4,
multiplyXW5,
               multiplyXW6, multiplyXW7, multiplyXW8, multiplyXW9,
multiplyXW10,
               addALL);
signal current_state : State := multiplyXW1;
signal next_state: State;
type alu_select is (multiply1, multiply2, multiply3, multiply4, multiply5,
                    multiply6, multiply7, multiply8, multiply9, multiply10,
add);
constant RG_w1: std_logic_vector(31 downto 0):= real_convert(-0.5);
constant RG_w2: std_logic_vector(31 downto 0):= real_convert(0.4);
```

Figure 43. Output layer neuron VHDL realization part 1.

```
constant RG_w3: std_logic_vector(31 downto 0):= real_convert(0.5);
constant RG_w4: std_logic_vector(31 downto 0):= real_convert(0.3);
constant RG_w5: std_logic_vector(31 downto 0):= real_convert(-1.1);
constant RG_w6: std_logic_vector(31 downto 0):= real_convert(1.0);
constant RG_w7: std_logic_vector(31 downto 0):= real_convert(0.52);
constant RG_w8: std_logic_vector(31 downto 0):= real_convert(0.2);
constant RG_w9: std_logic_vector(31 downto 0):= real_convert(0.07);
constant RG_w10: std_logic_vector(31 downto 0):= real_convert(0.8);
signal RG1_EN, RG2_EN, RG3_EN, RG4_EN, RG5_EN,
       RG6_EN, RG7_EN, RG8_EN, RG9_EN, RG10_EN, RGadd_EN: std_logic ;
signal alu_operations : alu_select;
signal ALUresultM1, ALUresultM2, ALUresultM3, ALUresultM4, ALUresultM5,
       ALUresultM6, ALUresultM7, ALUresultM8, ALUresultM9, ALUresultM10,
       ALUresultADD: signed (63 downto 0);
signal Mult_result_1, Mult_result_2, Mult_result_3, Mult_result_4,
Mult_result_5,Mult_result_6, Mult_result_7, Mult_result_8, Mult_result_9,
Mult_result_10,NET: signed (63 downto 0);
begin
process (clk)
  begin
   if clk'event and clk = '1' then
    current_state <= next_state;
   end if;
end process;
process (current_state, SKanDELTA_EN )
begin
next_state <= current_state;
alu_operations <= multiply1;
case current_state is
when multiplyXW1 =>
        if SKanDELTA_EN = '1' then
        alu_operations <= multiply1;
        RG1_EN <= '1';
        next_state <= multiplyXW2;
        end if;
when multiplyXW2 =>
        alu_operations <= multiply2;
        RG2_EN <= '1';
        next_state <= multiplyXW3;
when multiplyXW3 =>
        alu_operations <= multiply3;
        RG3_EN <= '1';
        next_state <= multiplyXW4;
when multiplyXW4 =>
        alu_operations <= multiply4;
        RG4_EN <= '1';
        next_state <= multiplyXW5;
```

Figure 44. Output layer neuron VHDL realization part 2.

```
when multiplyXW5 =>
        alu_operations <= multiply5;
        RG5_EN <= '1';
        next_state <= multiplyXW6;
when multiplyXW6 =>
        alu_operations <= multiply6;
        RG6_EN <= '1';
        next_state <= multiplyXW7;
when multiplyXW7 =>
        alu_operations <= multiply7;
        RG7_EN <= '1';
        next_state <= multiplyXW8;
when multiplyXW8 =>
        alu_operations <= multiply8;
        RG8_EN <= '1';
        next_state <= multiplyXW9;
when multiplyXW9 =>
        alu_operations <= multiply9;
        RG9_EN <= '1';
        next_state <= multiplyXW10;
when multiplyXW10=>
        alu_operations <= multiply10;
        RG10_EN <= '1';
        next_state <= addALL;
when addALL =>
        alu_operations <= add;
        RGadd_EN <= '1';
        next_state <= multiplyXW1;
end case; end process;
process ( alu_operations, RGin_x1, RGin_x2, RGin_x3, RGin_x4, RGin_x5,
RGin_x6, RGin_x7, RGin_x8, RGin_x9, RGin_x10, _result_1, Mult_result_2,
Mult_result_3, Mult_result_4, Mult_result_5, Mult_result_6, Mult_result_7,
Mult_result_8, Mult_result_9, Mult_result_10)
begin case alu_operations is
 when multiply1 => ALUresultM1 <= (SIGNED(RGin_x1) * SIGNED(RG_w1));
 when multiply2 => ALUresultM2 <= (SIGNED(RGin_x2) * SIGNED(RG_w2));
 when multiply3 => ALUresultM3 <= (SIGNED(RGin_x3) * SIGNED(RG_w3));
 when multiply4 => ALUresultM4 <= (SIGNED(RGin_x4) * SIGNED(RG_w4));
 when multiply5 => ALUresultM5 <= (SIGNED(RGin_x5) * SIGNED(RG_w5));
 when multiply6 => ALUresultM6 <= (SIGNED(RGin_x6) * SIGNED(RG_w6));
 when multiply7 => ALUresultM7 <= (SIGNED(RGin_x7) * SIGNED(RG_w7));
 when multiply8 => ALUresultM8 <= (SIGNED(RGin_x8) * SIGNED(RG_w8));
 when multiply9 => ALUresultM9 <= (SIGNED(RGin_x9) * SIGNED(RG_w9));
 when multiply10 => ALUresultM10 <= (SIGNED(RGin_x10) * SIGNED(RG_w10));
 when add => ALUresultADD <= (SIGNED(Mult_result_1) + SIGNED(Mult_result_2)
+ SIGNED(Mult_result_3) + SIGNED(Mult_result_4) + SIGNED(Mult_result_5) +
SIGNED(Mult_result_6) + SIGNED(Mult_result_7) + SIGNED(Mult_result_8) +
SIGNED(Mult_result_9) + SIGNED(Mult_result_10));
end case;
end process;
```

Figure 45. Output layer neuron VHDL realization part 3.

```
process (clk)
begin
if clk'event and clk = '1' then
if RG1_EN = '1' then
Mult_result_1 <= ALUresultM1;
end if;
if RG2_EN = '1' then
Mult_result_2 <= ALUresultM2;
end if;
if RG3_EN = '1' then
Mult_result_3 <= ALUresultM3;
end if;
if RG4_EN = '1' then
Mult_result_4 <= ALUresultM4;
end if;
if RG5_EN = '1' then
Mult_result_5 <= ALUresultM5;
end if;
if RG6_EN = '1' then
Mult_result_6 <= ALUresultM6;
end if;
if RG7_EN = '1' then
Mult_result_7 <= ALUresultM7;
end if;
if RG8_EN = '1' then
Mult_result_8 <= ALUresultM8;
end if;
if RG9_EN = '1' then
Mult_result_9 <= ALUresultM9;
end if;
if RG10_EN = '1' then
Mult_result_10 <= ALUresultM10;
end if;
if RGadd_EN = '1' then
NET <= ALUresultADD;
end if;
end if;
end process;
SKanDELTA_NET <= STD_LOGIC_VECTOR(NET);
abc <= std_convert(NET);
process (abc)
begin
if (abc >= 2.5) then ANOUT <= '1';
else
ANOUT <= '0';
end if;
end process;
end Behavioral;
```

Figure 46. Output layer neuron VHDL realization part 4.

In the simulation result on Figure 47 we see the highlighted signal “ANOUT” that represents the output of the neuron, switches it state once the signal “out_real” that represents the NET output, switches its value from 0.0 to 3.9153618768. This operation shows that the neuron is working as expected, changing the output to logical “HIGH” once the NET value is above the threshold value $\theta = 2.5$

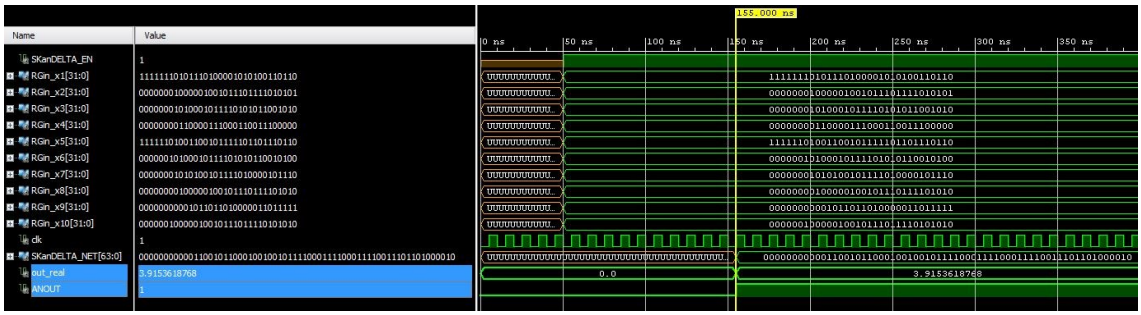


Figure 47. Simulation result of the output layer neuron

Again to finalize the design I make an IP block of the output layer neuron by using Vivado IP packager displayed on Figure 48.

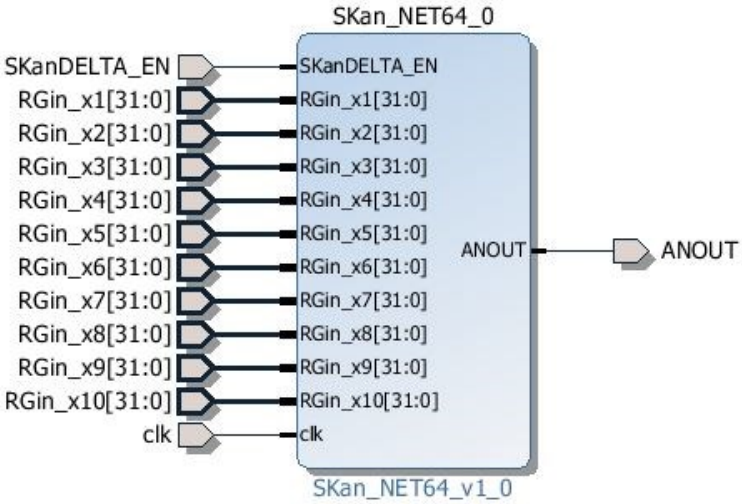


Figure 48. Output layer neuron IP block.

5.2 The full ANN realization

Now that I have all the necessary IP blocks, I can use them to realize the full ANN that is displayed on Figure 49.

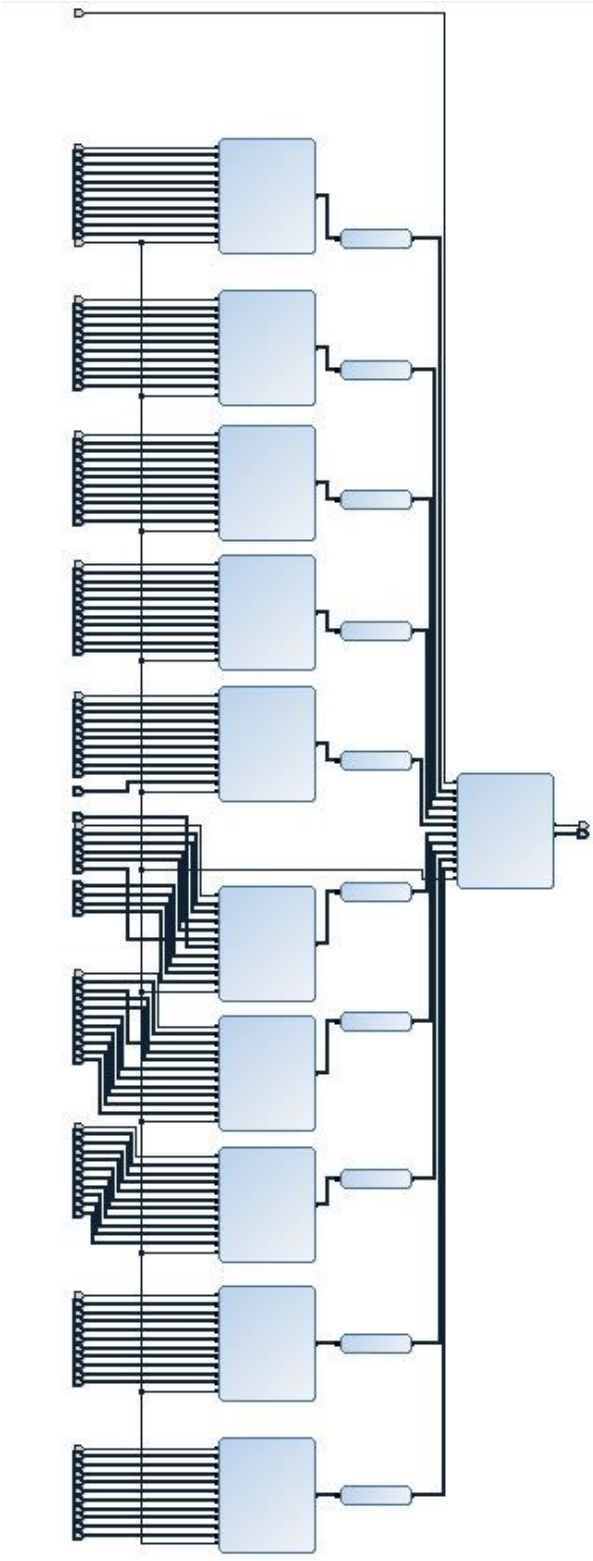


Figure 49. Finalized ANN IP.

By applying the test bench presented on Figure 51 to Figure 61 to simulate the operation of this ANN, I get the result displayed on Figure 50.

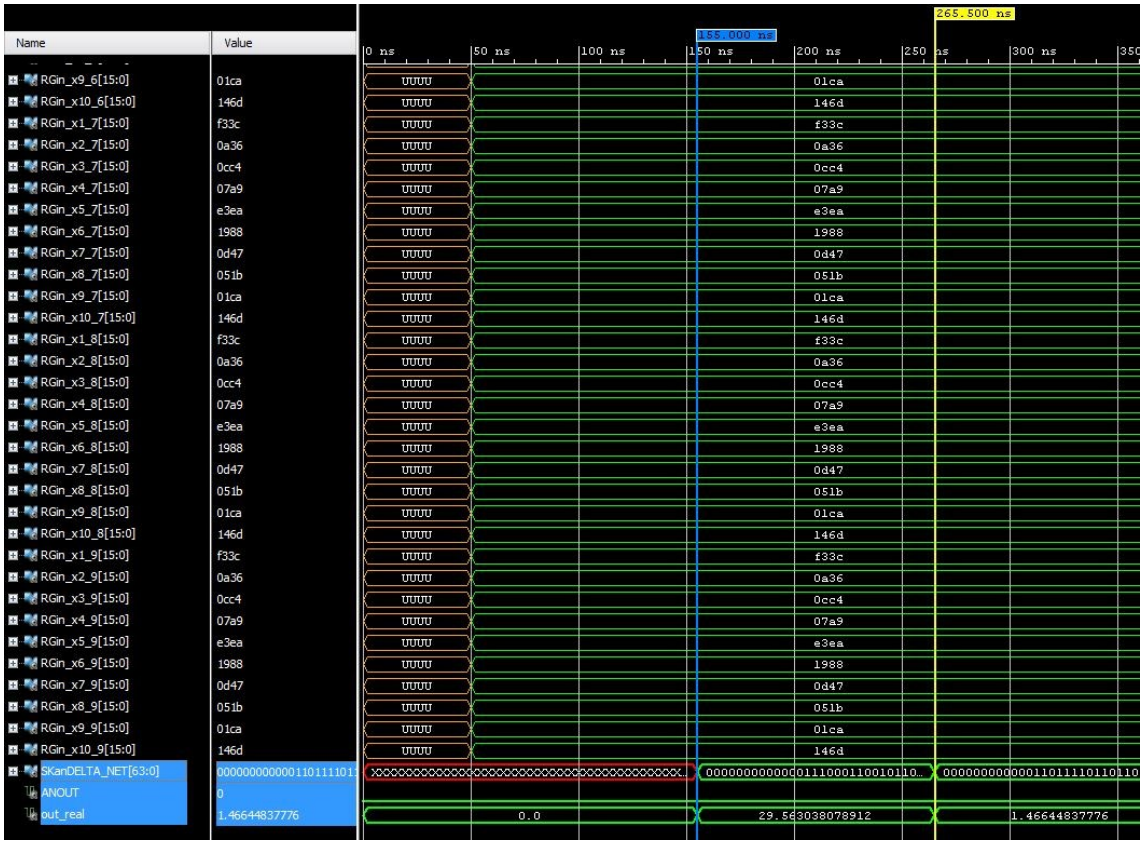


Figure 50. Full ANN simulation.

From the simulation we see that the entire process of computation takes 265nS.

This time consists of the waiting time of 50nS plus 11 clock cycles for the computation done by all the neurons of the input layer in parallel and additional 11 clock cycles for the computation done by the single neuron in the output layer.

The “ANOUT” signal remains at “LOW” state during the entire time, since the comparison takes place after 265nS when the NET value displayed by the “out_real” signal is 1.46644837776 which is below the previously established threshold value $\theta = 2.5$

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity TB2 is
end TB2;
architecture Behavioral of TB2 is
COMPONENT SKan_complete_network
PORT (     SKanDELTA_EN_1 : in  STD_LOGIC;
           RGin_x1 : in std_logic_vector (15 downto 0);
           RGin_x2 : in std_logic_vector (15 downto 0);
           RGin_x3 : in std_logic_vector (15 downto 0);
           RGin_x4 : in std_logic_vector (15 downto 0);
           RGin_x5 : in std_logic_vector (15 downto 0);
           RGin_x6 : in std_logic_vector (15 downto 0);
           RGin_x7 : in std_logic_vector (15 downto 0);
           RGin_x8 : in std_logic_vector (15 downto 0);
           RGin_x9 : in std_logic_vector (15 downto 0);
           RGin_x10 : in std_logic_vector (15 downto 0);
           ------------------------------------------
           SKanDELTA_EN_2 : in  STD_LOGIC;
           RGin_x1_1 : in std_logic_vector (15 downto 0);
           RGin_x2_1 : in std_logic_vector (15 downto 0);
           RGin_x3_1 : in std_logic_vector (15 downto 0);
           RGin_x4_1 : in std_logic_vector (15 downto 0);
           RGin_x5_1 : in std_logic_vector (15 downto 0);
           RGin_x6_1 : in std_logic_vector (15 downto 0);
           RGin_x7_1 : in std_logic_vector (15 downto 0);
           RGin_x8_1 : in std_logic_vector (15 downto 0);
           RGin_x9_1 : in std_logic_vector (15 downto 0);
           RGin_x10_1 : in std_logic_vector (15 downto 0);
           ------------------------------------------
           SKanDELTA_EN_3 : in  STD_LOGIC;
           RGin_x1_2 : in std_logic_vector (15 downto 0);
           RGin_x2_2 : in std_logic_vector (15 downto 0);
           RGin_x3_2 : in std_logic_vector (15 downto 0);
           RGin_x4_2 : in std_logic_vector (15 downto 0);
           RGin_x5_2 : in std_logic_vector (15 downto 0);
           RGin_x6_2 : in std_logic_vector (15 downto 0);
           RGin_x7_2 : in std_logic_vector (15 downto 0);
           RGin_x8_2 : in std_logic_vector (15 downto 0);
           RGin_x9_2 : in std_logic_vector (15 downto 0);
           RGin_x10_2 : in std_logic_vector (15 downto 0);
           ------------------------------------------
           SKanDELTA_EN_4 : in  STD_LOGIC;
           RGin_x1_3 : in std_logic_vector (15 downto 0);
           RGin_x2_3 : in std_logic_vector (15 downto 0);
           RGin_x3_3 : in std_logic_vector (15 downto 0);
           RGin_x4_3 : in std_logic_vector (15 downto 0);
           RGin_x5_3 : in std_logic_vector (15 downto 0);
```

Figure 51. Full ANN test bench code part 1.

```
RGin_x6_3 : in std_logic_vector (15 downto 0);
RGin_x7_3 : in std_logic_vector (15 downto 0);
RGin_x8_3 : in std_logic_vector (15 downto 0);
RGin_x9_3 : in std_logic_vector (15 downto 0);
RGin_x10_3 : in std_logic_vector (15 downto 0);
-----------------------------------------
SKanDELTA_EN_5 : in  STD_LOGIC;
RGin_x1_4 : in std_logic_vector (15 downto 0);
RGin_x2_4 : in std_logic_vector (15 downto 0);
RGin_x3_4 : in std_logic_vector (15 downto 0);
RGin_x4_4 : in std_logic_vector (15 downto 0);
RGin_x5_4 : in std_logic_vector (15 downto 0);
RGin_x6_4 : in std_logic_vector (15 downto 0);
RGin_x7_4 : in std_logic_vector (15 downto 0);
RGin_x8_4 : in std_logic_vector (15 downto 0);
RGin_x9_4 : in std_logic_vector (15 downto 0);
RGin_x10_4 : in std_logic_vector (15 downto 0);
-----------------------------------------
SKanDELTA_EN_6 : in  STD_LOGIC;
RGin_x1_5 : in std_logic_vector (15 downto 0);
RGin_x2_5 : in std_logic_vector (15 downto 0);
RGin_x3_5 : in std_logic_vector (15 downto 0);
RGin_x4_5 : in std_logic_vector (15 downto 0);
RGin_x5_5 : in std_logic_vector (15 downto 0);
RGin_x6_5 : in std_logic_vector (15 downto 0);
RGin_x7_5 : in std_logic_vector (15 downto 0);
RGin_x8_5 : in std_logic_vector (15 downto 0);
RGin_x9_5 : in std_logic_vector (15 downto 0);
RGin_x10_5 : in std_logic_vector (15 downto 0);
-----------------------------------------
SKanDELTA_EN_7 : in  STD_LOGIC;
RGin_x1_6 : in std_logic_vector (15 downto 0);
RGin_x2_6 : in std_logic_vector (15 downto 0);
RGin_x3_6 : in std_logic_vector (15 downto 0);
RGin_x4_6 : in std_logic_vector (15 downto 0);
RGin_x5_6 : in std_logic_vector (15 downto 0);
RGin_x6_6 : in std_logic_vector (15 downto 0);
RGin_x7_6 : in std_logic_vector (15 downto 0);
RGin_x8_6 : in std_logic_vector (15 downto 0);
RGin_x9_6 : in std_logic_vector (15 downto 0);
RGin_x10_6 : in std_logic_vector (15 downto 0);
-----------------------------------------
SKanDELTA_EN_8 : in  STD_LOGIC;
RGin_x1_7 : in std_logic_vector (15 downto 0);
RGin_x2_7 : in std_logic_vector (15 downto 0);
RGin_x3_7 : in std_logic_vector (15 downto 0);
RGin_x4_7 : in std_logic_vector (15 downto 0);
RGin_x5_7 : in std_logic_vector (15 downto 0);
RGin_x6_7 : in std_logic_vector (15 downto 0);
```

Figure 52. Full ANN test bench code part 2.

```
            RGin_x7_7 : in std_logic_vector (15 downto 0);
            RGin_x8_7 : in std_logic_vector (15 downto 0);
             RGin_x9_7 : in std_logic_vector (15 downto 0);
            RGin_x10_7 : in std_logic_vector (15 downto 0);
            -----------------------------------------
            SKanDELTA_EN_9 : in  STD_LOGIC;
            RGin_x1_8 : in std_logic_vector (15 downto 0);
            RGin_x2_8 : in std_logic_vector (15 downto 0);
            RGin_x3_8 : in std_logic_vector (15 downto 0);
            RGin_x4_8 : in std_logic_vector (15 downto 0);
            RGin_x5_8 : in std_logic_vector (15 downto 0);
            RGin_x6_8 : in std_logic_vector (15 downto 0);
            RGin_x7_8 : in std_logic_vector (15 downto 0);
            RGin_x8_8 : in std_logic_vector (15 downto 0);
            RGin_x9_8 : in std_logic_vector (15 downto 0);
            RGin_x10_8 : in std_logic_vector (15 downto 0);
            -----------------------------------------
            SKanDELTA_EN_10 : in  STD_LOGIC;
            RGin_x1_9 : in std_logic_vector (15 downto 0);
            RGin_x2_9 : in std_logic_vector (15 downto 0);
            RGin_x3_9 : in std_logic_vector (15 downto 0);
            RGin_x4_9 : in std_logic_vector (15 downto 0);
            RGin_x5_9 : in std_logic_vector (15 downto 0);
            RGin_x6_9 : in std_logic_vector (15 downto 0);
            RGin_x7_9 : in std_logic_vector (15 downto 0);
            RGin_x8_9 : in std_logic_vector (15 downto 0);
            RGin_x9_9 : in std_logic_vector (15 downto 0);
            RGin_x10_9 : in std_logic_vector (15 downto 0);
            SKanDELTA_EN : in  STD_LOGIC;
            SKanDELTA_NET : out std_logic_vector (63 downto 0);
            ANOUT : out std_logic;
            clk_1 : in STD_LOGIC);END COMPONENT;
-----------------------Real Value convert--------------------
function real_convert (real_in: real) return std_logic_vector;
function real_convert (real_in: real) return std_logic_vector is
variable std_out: std_logic_vector(15 downto 0);
begin
std_out := std_logic_vector(to_signed(integer(real_in/0.000153),16));
return std_out;
end real_convert;
-------------------Vector Convert-------------------------
function std_convert (std_in: signed) return real;
function std_convert (std_in: signed) return real is
variable real_out: real;
begin
real_out := 0.000153 * 0.000153 * real(to_integer(std_in));
return real_out;
end std_convert;
signal out_real: real;
```

Figure 53. Full ANN test bench code part 3.

```
--inputs
signal clk_1 : STD_LOGIC;
signal SKanDELTA_EN :  std_logic;
signal SKanDELTA_EN_1 :  std_logic;
signal SKanDELTA_EN_2 :  std_logic;
signal SKanDELTA_EN_3 :  std_logic;
signal SKanDELTA_EN_4 :  std_logic;
signal SKanDELTA_EN_5 :  std_logic;
signal SKanDELTA_EN_6 :  std_logic;
signal SKanDELTA_EN_7 :  std_logic;
signal SKanDELTA_EN_8 :  std_logic;
signal SKanDELTA_EN_9 :  std_logic;
signal SKanDELTA_EN_10 :  std_logic;
signal RGin_x1 : std_logic_vector (15 downto 0);
signal RGin_x2 : std_logic_vector (15 downto 0);
signal RGin_x3 : std_logic_vector (15 downto 0);
signal RGin_x4 : std_logic_vector (15 downto 0);
signal RGin_x5 : std_logic_vector (15 downto 0);
signal RGin_x6 : std_logic_vector (15 downto 0);
signal RGin_x7 : std_logic_vector (15 downto 0);
signal RGin_x8 : std_logic_vector (15 downto 0);
signal RGin_x9 : std_logic_vector (15 downto 0);
signal RGin_x10 : std_logic_vector (15 downto 0);
signal RGin_x1_1 : std_logic_vector (15 downto 0);
signal RGin_x2_1 : std_logic_vector (15 downto 0);
signal RGin_x3_1 : std_logic_vector (15 downto 0);
signal RGin_x4_1 : std_logic_vector (15 downto 0);
signal RGin_x5_1 : std_logic_vector (15 downto 0);
signal RGin_x6_1 : std_logic_vector (15 downto 0);
signal RGin_x7_1 : std_logic_vector (15 downto 0);
signal RGin_x8_1 : std_logic_vector (15 downto 0);
signal RGin_x9_1 : std_logic_vector (15 downto 0);
signal RGin_x10_1 : std_logic_vector (15 downto 0);
signal RGin_x1_2 : std_logic_vector (15 downto 0);
signal RGin_x2_2 : std_logic_vector (15 downto 0);
signal RGin_x3_2 : std_logic_vector (15 downto 0);
signal RGin_x4_2 : std_logic_vector (15 downto 0);
signal RGin_x5_2 : std_logic_vector (15 downto 0);
signal RGin_x6_2 : std_logic_vector (15 downto 0);
signal RGin_x7_2 : std_logic_vector (15 downto 0);
signal RGin_x8_2 : std_logic_vector (15 downto 0);
signal RGin_x9_2 : std_logic_vector (15 downto 0);
signal RGin_x10_2 : std_logic_vector (15 downto 0);
signal RGin_x1_3 : std_logic_vector (15 downto 0);
signal RGin_x2_3 : std_logic_vector (15 downto 0);
signal RGin_x3_3 : std_logic_vector (15 downto 0);
signal RGin_x4_3 : std_logic_vector (15 downto 0);
signal RGin_x5_3 : std_logic_vector (15 downto 0);
```

Figure 54. Full ANN test bench code part 4.

```
signal RGin_x6_3 : std_logic_vector (15 downto 0);
signal RGin_x7_3 : std_logic_vector (15 downto 0);
signal RGin_x8_3 : std_logic_vector (15 downto 0);
signal RGin_x9_3 : std_logic_vector (15 downto 0);
signal RGin_x10_3 : std_logic_vector (15 downto 0);
signal RGin_x1_4 : std_logic_vector (15 downto 0);
signal RGin_x2_4 : std_logic_vector (15 downto 0);
signal RGin_x3_4 : std_logic_vector (15 downto 0);
signal RGin_x4_4 : std_logic_vector (15 downto 0);
signal RGin_x5_4 : std_logic_vector (15 downto 0);
signal RGin_x6_4 : std_logic_vector (15 downto 0);
signal RGin_x7_4 : std_logic_vector (15 downto 0);
signal RGin_x8_4 : std_logic_vector (15 downto 0);
signal RGin_x9_4 : std_logic_vector (15 downto 0);
signal RGin_x10_4 : std_logic_vector (15 downto 0);
signal RGin_x1_5 : std_logic_vector (15 downto 0);
signal RGin_x2_5 : std_logic_vector (15 downto 0);
signal RGin_x3_5 : std_logic_vector (15 downto 0);
signal RGin_x4_5 : std_logic_vector (15 downto 0);
signal RGin_x5_5 : std_logic_vector (15 downto 0);
signal RGin_x6_5 : std_logic_vector (15 downto 0);
signal RGin_x7_5 : std_logic_vector (15 downto 0);
signal RGin_x8_5 : std_logic_vector (15 downto 0);
signal RGin_x9_5 : std_logic_vector (15 downto 0);
signal RGin_x10_5 : std_logic_vector (15 downto 0);
signal RGin_x1_6 : std_logic_vector (15 downto 0);
signal RGin_x2_6 : std_logic_vector (15 downto 0);
signal RGin_x3_6 : std_logic_vector (15 downto 0);
signal RGin_x4_6 : std_logic_vector (15 downto 0);
signal RGin_x5_6 : std_logic_vector (15 downto 0);
signal RGin_x6_6 : std_logic_vector (15 downto 0);
signal RGin_x7_6 : std_logic_vector (15 downto 0);
signal RGin_x8_6 : std_logic_vector (15 downto 0);
signal RGin_x9_6 : std_logic_vector (15 downto 0);
signal RGin_x10_6 : std_logic_vector (15 downto 0);
signal RGin_x1_7 : std_logic_vector (15 downto 0);
signal RGin_x2_7 : std_logic_vector (15 downto 0);
signal RGin_x3_7 : std_logic_vector (15 downto 0);
signal RGin_x4_7 : std_logic_vector (15 downto 0);
signal RGin_x5_7 : std_logic_vector (15 downto 0);
signal RGin_x6_7 : std_logic_vector (15 downto 0);
signal RGin_x7_7 : std_logic_vector (15 downto 0);
signal RGin_x8_7 : std_logic_vector (15 downto 0);
signal RGin_x9_7 : std_logic_vector (15 downto 0);
signal RGin_x10_7 : std_logic_vector (15 downto 0);
signal RGin_x1_8 : std_logic_vector (15 downto 0);
signal RGin_x2_8 : std_logic_vector (15 downto 0);
signal RGin_x3_8 : std_logic_vector (15 downto 0);
signal RGin_x4_8 : std_logic_vector (15 downto 0);
```

Figure 55. Full ANN test bench code part 5.

```
signal RGin_x5_8 : std_logic_vector (15 downto 0);
signal RGin_x6_8 : std_logic_vector (15 downto 0);
signal RGin_x7_8 : std_logic_vector (15 downto 0);
signal RGin_x8_8 : std_logic_vector (15 downto 0);
signal RGin_x9_8 : std_logic_vector (15 downto 0);
signal RGin_x10_8 : std_logic_vector (15 downto 0);
signal RGin_x1_9 : std_logic_vector (15 downto 0);
signal RGin_x2_9 : std_logic_vector (15 downto 0);
signal RGin_x3_9 : std_logic_vector (15 downto 0);
signal RGin_x4_9 : std_logic_vector (15 downto 0);
signal RGin_x5_9 : std_logic_vector (15 downto 0);
signal RGin_x6_9 : std_logic_vector (15 downto 0);
signal RGin_x7_9 : std_logic_vector (15 downto 0);
signal RGin_x8_9 : std_logic_vector (15 downto 0);
signal RGin_x9_9 : std_logic_vector (15 downto 0);
signal RGin_x10_9 : std_logic_vector (15 downto 0);
--outputs
signal SKanDELTA_NET : STD_LOGIC_VECTOR (63 downto 0);
signal ANOUT : std_logic;
constant clk_period : time := 10 ns;
constant real_number: real:=4.99;
constant real_number_to_std: std_logic_vector(15 downto
0):=real_convert(real_number);
constant real_number_restored: real:=std_convert(signed(real_number_to_std));
constant real_number_square: signed(31 downto
0):=signed(real_number_to_std)*signed(real_number_to_std);
constant real_number_sqaured: real:=std_convert(real_number_square);
begin
uut: SKan_complete_network PORT MAP (
         clk_1 => clk_1,
         ANOUT => ANOUT,
         RGin_x1 =>  RGin_x1 ,
         RGin_x2 =>  RGin_x2 ,
         RGin_x3 =>  RGin_x3 ,
         RGin_x4 =>  RGin_x4 ,
         RGin_x5 =>  RGin_x5 ,
         RGin_x6 =>  RGin_x6 ,
         RGin_x7 =>  RGin_x7 ,
         RGin_x8 =>  RGin_x8 ,
         RGin_x9 =>  RGin_x9 ,
         RGin_x10 => RGin_x10 ,
         RGin_x1_1 =>  RGin_x1_1 ,
         RGin_x2_1 =>  RGin_x2_1 ,
         RGin_x3_1 =>  RGin_x3_1 ,
         RGin_x4_1 =>  RGin_x4_1 ,
         RGin_x5_1 =>  RGin_x5_1 ,
         RGin_x6_1 =>  RGin_x6_1 ,
         RGin_x7_1 =>  RGin_x7_1 ,
         RGin_x8_1 =>  RGin_x8_1 ,
```

Figure 56. Full ANN test bench code part 6.

```
RGin_x9_1 =>  RGin_x9_1 ,
        RGin_x10_1 => RGin_x10_1 ,
        RGin_x1_2 =>  RGin_x1_2 ,
        RGin_x2_2 =>  RGin_x2_2 ,
        RGin_x3_2 =>  RGin_x3_2 ,
        RGin_x4_2 =>  RGin_x4_2 ,
        RGin_x5_2 =>  RGin_x5_2 ,
        RGin_x6_2 =>  RGin_x6_2 ,
        RGin_x7_2 =>  RGin_x7_2 ,
        RGin_x8_2 =>  RGin_x8_2 ,
        RGin_x9_2 =>  RGin_x9_2 ,
        RGin_x10_2 => RGin_x10_2 ,
        RGin_x1_3 =>  RGin_x1_3 ,
        RGin_x2_3 =>  RGin_x2_3 ,
        RGin_x3_3 =>  RGin_x3_3 ,
        RGin_x4_3 =>  RGin_x4_3 ,
        RGin_x5_3 =>  RGin_x5_3 ,
        RGin_x6_3 =>  RGin_x6_3 ,
        RGin_x7_3 =>  RGin_x7_3 ,
        RGin_x8_3 =>  RGin_x8_3 ,
        RGin_x9_3 =>  RGin_x9_3 ,
        RGin_x10_3 => RGin_x10_3 ,
        RGin_x1_4 =>  RGin_x1_4 ,
        RGin_x2_4 =>  RGin_x2_4 ,
        RGin_x3_4 =>  RGin_x3_4 ,
        RGin_x4_4 =>  RGin_x4_4 ,
        RGin_x5_4 =>  RGin_x5_4 ,
        RGin_x6_4 =>  RGin_x6_4 ,
        RGin_x7_4 =>  RGin_x7_4 ,
        RGin_x8_4 =>  RGin_x8_4 ,
        RGin_x9_4 =>  RGin_x9_4 ,
        RGin_x10_4 => RGin_x10_4 ,
        RGin_x1_5 =>  RGin_x1_5 ,
        RGin_x2_5 =>  RGin_x2_5 ,
        RGin_x3_5 =>  RGin_x3_5 ,
        RGin_x4_5 =>  RGin_x4_5 ,
        RGin_x5_5 =>  RGin_x5_5 ,
        RGin_x6_5 =>  RGin_x6_5 ,
        RGin_x7_5 =>  RGin_x7_5 ,
        RGin_x8_5 =>  RGin_x8_5 ,
        RGin_x9_5 =>  RGin_x9_5 ,
        RGin_x10_5 => RGin_x10_5,
        RGin_x1_6 =>  RGin_x1_6 ,
        RGin_x2_6 =>  RGin_x2_6 ,
        RGin_x3_6 =>  RGin_x3_6 ,
        RGin_x4_6 =>  RGin_x4_6 ,
        RGin_x5_6 =>  RGin_x5_6 ,
        RGin_x6_6 =>  RGin_x6_6 ,
        RGin_x7_6 =>  RGin_x7_6 ,
```

Figure 57. Full ANN test bench code part 7.

```
RGin_x8_6 =>  RGin_x8_6 ,
         RGin_x9_6 =>  RGin_x9_6 ,
         RGin_x10_6 => RGin_x10_6,
         RGin_x1_7 =>  RGin_x1_7 ,
         RGin_x2_7 =>  RGin_x2_7 ,
         RGin_x3_7 =>  RGin_x3_7 ,
         RGin_x4_7 =>  RGin_x4_7 ,
         RGin_x5_7 =>  RGin_x5_7 ,
         RGin_x6_7 =>  RGin_x6_7 ,
         RGin_x7_7 =>  RGin_x7_7 ,
         RGin_x8_7 =>  RGin_x8_7 ,
         RGin_x9_7 =>  RGin_x9_7 ,
         RGin_x10_7 => RGin_x10_7,
         RGin_x1_8 =>  RGin_x1_8 ,
         RGin_x2_8 =>  RGin_x2_8 ,
         RGin_x3_8 =>  RGin_x3_8 ,
         RGin_x4_8 =>  RGin_x4_8 ,
         RGin_x5_8 =>  RGin_x5_8 ,
         RGin_x6_8 =>  RGin_x6_8 ,
         RGin_x7_8 =>  RGin_x7_8 ,
         RGin_x8_8 =>  RGin_x8_8 ,
         RGin_x9_8 =>  RGin_x9_8 ,
         RGin_x10_8 => RGin_x10_8,
         RGin_x1_9 =>  RGin_x1_9 ,
         RGin_x2_9 =>  RGin_x2_9 ,
         RGin_x3_9 =>  RGin_x3_9 ,
         RGin_x4_9 =>  RGin_x4_9 ,
         RGin_x5_9 =>  RGin_x5_9 ,
         RGin_x6_9 =>  RGin_x6_9 ,
         RGin_x7_9 =>  RGin_x7_9 ,
         RGin_x8_9 =>  RGin_x8_9 ,
         RGin_x9_9 =>  RGin_x9_9 ,
         RGin_x10_9 => RGin_x10_9,
         SKanDELTA_EN => SKanDELTA_EN,
         SKanDELTA_EN_1 => SKanDELTA_EN_1,
         SKanDELTA_EN_2 => SKanDELTA_EN_2,
SKanDELTA_EN_3 => SKanDELTA_EN_3,
SKanDELTA_EN_4 => SKanDELTA_EN_4,
SKanDELTA_EN_5 => SKanDELTA_EN_5,
SKanDELTA_EN_6 => SKanDELTA_EN_6,
         SKanDELTA_EN_7 => SKanDELTA_EN_7,
         SKanDELTA_EN_8 => SKanDELTA_EN_8,
         SKanDELTA_EN_9 => SKanDELTA_EN_9,
         SKanDELTA_EN_10 => SKanDELTA_EN_10,
        SKanDELTA_NET => SKanDELTA_NET);
out_real <= std_convert(SIGNED(SKanDELTA_NET));
clk_process : process
    begin
        clk_1 <= '0';
```

Figure 58. Full ANN test bench code part 8.

```
wait for clk_period/2;
        clk_1 <= '1';
        wait for clk_period/2;
    end process;
--Stimulus Process
stim_proc: process
begin
wait for 50 ns;
------------------------------SKan_DELTA_0-------------------------------
RGin_x1 <= real_convert(-0.5);
RGin_x2 <= real_convert(0.4);
RGin_x3 <= real_convert(0.5);
RGin_x4 <= real_convert(0.3);
RGin_x5 <= real_convert(-1.1);
RGin_x6 <= real_convert(1.0);
RGin_x7 <= real_convert(0.52);
RGin_x8 <= real_convert(0.2);
RGin_x9 <= real_convert(0.07);
RGin_x10 <= real_convert(0.8);
SKanDELTA_EN_1 <= '1';
------------------------------SKan_DELTA_1-------------------------------
RGin_x1_1 <= real_convert(-0.5);
RGin_x2_1 <= real_convert(0.4);
RGin_x3_1 <= real_convert(0.5);
RGin_x4_1 <= real_convert(0.3);
RGin_x5_1 <= real_convert(-1.1);
RGin_x6_1 <= real_convert(1.0);
RGin_x7_1 <= real_convert(0.52);
RGin_x8_1 <= real_convert(0.2);
RGin_x9_1 <= real_convert(0.07);
RGin_x10_1 <= real_convert(0.8);
SKanDELTA_EN_2 <= '1';
------------------------------SKan_DELTA_2-------------------------------
RGin_x1_2 <= real_convert(-0.5);
RGin_x2_2 <= real_convert(0.4);
RGin_x3_2 <= real_convert(0.5);
RGin_x4_2 <= real_convert(0.3);
RGin_x5_2 <= real_convert(-1.1);
RGin_x6_2 <= real_convert(1.0);
RGin_x7_2 <= real_convert(0.52);
RGin_x8_2 <= real_convert(0.2);
RGin_x9_2 <= real_convert(0.07);
RGin_x10_2 <= real_convert(0.8);
SKanDELTA_EN_3 <= '1';
------------------------------SKan_DELTA_3-------------------------------
RGin_x1_3 <= real_convert(-0.5);
RGin_x2_3 <= real_convert(0.4);
RGin_x3_3 <= real_convert(0.5);
RGin_x4_3 <= real_convert(0.3);
```

Figure 59. Full ANN test bench code part 9.

```
RGin_x5_3 <= real_convert(-1.1);
RGin_x6_3 <= real_convert(1.0);
RGin_x7_3 <= real_convert(0.52);
RGin_x8_3 <= real_convert(0.2);
RGin_x9_3 <= real_convert(0.07);
RGin_x10_3 <= real_convert(0.8);
SKanDELTA_EN_4 <= '1';
------------------------------SKan_DELTA_4-------------------------------
RGin_x1_4 <= real_convert(-0.5);
RGin_x2_4 <= real_convert(0.4);
RGin_x3_4 <= real_convert(0.5);
RGin_x4_4 <= real_convert(0.3);
RGin_x5_4 <= real_convert(-1.1);
RGin_x6_4 <= real_convert(1.0);
RGin_x7_4 <= real_convert(0.52);
RGin_x8_4 <= real_convert(0.2);
RGin_x9_4 <= real_convert(0.07);
RGin_x10_4 <= real_convert(0.8);
SKanDELTA_EN_5 <= '1';
------------------------------SKan_DELTA_5-------------------------------
RGin_x1_5 <= real_convert(-0.5);
RGin_x2_5 <= real_convert(0.4);
RGin_x3_5 <= real_convert(0.5);
RGin_x4_5 <= real_convert(0.3);
RGin_x5_5 <= real_convert(-1.1);
RGin_x6_5 <= real_convert(1.0);
RGin_x7_5 <= real_convert(0.52);
RGin_x8_5 <= real_convert(0.2);
RGin_x9_5 <= real_convert(0.07);
RGin_x10_5 <= real_convert(0.8);
SKanDELTA_EN_6 <= '1';
------------------------------SKan_DELTA_6-------------------------------
RGin_x1_6 <= real_convert(-0.5);
RGin_x2_6 <= real_convert(0.4);
RGin_x3_6 <= real_convert(0.5);
RGin_x4_6 <= real_convert(0.3);
RGin_x5_6 <= real_convert(-1.1);
RGin_x6_6 <= real_convert(1.0);
RGin_x7_6 <= real_convert(0.52);
RGin_x8_6 <= real_convert(0.2);
RGin_x9_6 <= real_convert(0.07);
RGin_x10_6 <= real_convert(0.8);
SKanDELTA_EN_7 <= '1';
------------------------------SKan_DELTA_7-------------------------------
RGin_x1_7 <= real_convert(-0.5);
RGin_x2_7 <= real_convert(0.4);
RGin_x3_7 <= real_convert(0.5);
RGin_x4_7 <= real_convert(0.3);
RGin_x5_7 <= real_convert(-1.1);
```

Figure 60. Full ANN test bench code part 10.

```
RGin_x6_7 <= real_convert(1.0);
RGin_x7_7 <= real_convert(0.52);
RGin_x8_7 <= real_convert(0.2);
RGin_x9_7 <= real_convert(0.07);
RGin_x10_7 <= real_convert(0.8);
SKanDELTA_EN_8 <= '1';
-----------------------------SKan_DELTA_8-------------------------------
RGin_x1_8 <= real_convert(-0.5);
RGin_x2_8 <= real_convert(0.4);
RGin_x3_8 <= real_convert(0.5);
RGin_x4_8 <= real_convert(0.3);
RGin_x5_8 <= real_convert(-1.1);
RGin_x6_8 <= real_convert(1.0);
RGin_x7_8 <= real_convert(0.52);
RGin_x8_8 <= real_convert(0.2);
RGin_x9_8 <= real_convert(0.07);
RGin_x10_8 <= real_convert(0.8);
SKanDELTA_EN_9 <= '1';
-----------------------------SKan_DELTA_9-------------------------------
RGin_x1_9 <= real_convert(-0.5);
RGin_x2_9 <= real_convert(0.4);
RGin_x3_9 <= real_convert(0.5);
RGin_x4_9 <= real_convert(0.3);
RGin_x5_9 <= real_convert(-1.1);
RGin_x6_9 <= real_convert(1.0);
RGin_x7_9 <= real_convert(0.52);
RGin_x8_9 <= real_convert(0.2);
RGin_x9_9 <= real_convert(0.07);
RGin_x10_9 <= real_convert(0.8);
SKanDELTA_EN_10 <= '1';
-----------------------------SKan_NET64---------------------------------
SKanDELTA_EN <= '1';
end process;
```

end Behavioral;

Figure 61. Full ANN test bench code part 11.

6 Summary

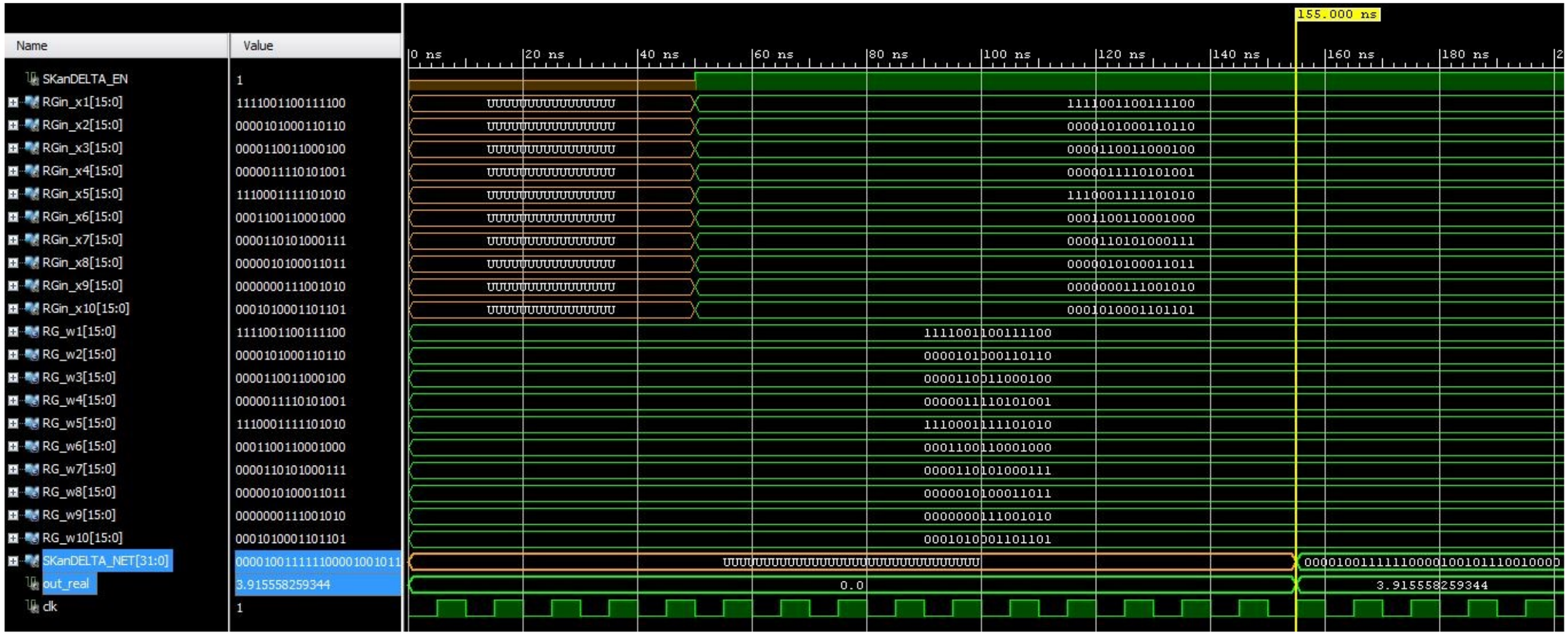
The realization of an ANN in VHDL has proven to be successful. With the implementation of the latest Xilinx software Vivado High Level Synthesis should be possible to synthesise this design on a FPGA board like the Nexys 4 Artix-7 FPGA Trainer Board. With the Nexys 4 having internal clock speeds exceeding 450MHz, the ANN design on the board should have no problem operating at its current speed of 100MHz. The high clock speed together with the parallel structure of the ANN allow my design to react to the rising brain activity faster than a human can actualize this activity into motion.

The next step is to realize EEG signal filtering on the FPGA and a backpropagation learning algorithm.

References

1. **Keim, Brandon (April 13, 2008). "Brain Scanners Can See Your Decisions Before You Make Them". Wired News (CondéNet). Retrieved 2008-04-13.**
2. **Niedermeyer E. and da Silva F.L. (2004). "Electroencephalography: Basic Principles, Clinical Applications, and Related Fields". Lippincot Williams & Wilkins. ISBN 0-7817-5126-8**
3. **Zaentsev I.S. (1999). "Neural networks: basic models". Voronezh: Voronezh State University, 1999, Textbook**
4. **Cahn, B. Rael; Polich, John (2006). "Meditation states and traits: EEG, ERP, and neuroimaging studies". Psychological Bulletin 132 (2): 180–211.**
5. **Niedermeyer, E (1997). "Alpha rhythms as physiological and abnormal phenomena". International Journal of Psychophysiology 26 (1–3): 31–49.**
6. **Pfurtscheller, G.; Lopes Da Silva, F.H. (1999). "Event-related EEG/MEG synchronization and desynchronization: Basic principles". Clinical Neurophysiology 110**
7. **Anil K. Jain; Jianchang Mao; K.M. Mohiuddin (1996) "Artificial Neural Networks: A Tutorial". Computer March 1996: 31-44.**
8. **Yu-Ting Liu; Yang-Yin Lin, Member, IEEE; Shang-Lin Wu, Member, IEEE; Chun-Hsiang Chuang; Chin-Teng Lin, Fellow, IEEE (2016) "Brain Dynamics in Predicting Driving Fatigue Using a Recurrent Self-Evolving Fuzzy Neural Network". IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, VOL. 27, NO. 2, FEBRUARY 2016**
9. **Turker Tekin Erguzel; Serhat Ozekes; Oguz Tan; Selahattin Gultekin (2015)." Feature Selection and Classification of Electroencephalographic Signals: An Artificial Neural Network and Genetic Algorithm Based Approach". Clinical EEG and Neuroscience 2015, Vol. 46(4) 321–326**
10. **Rifai Chai, Student Member, IEEE, Sai Ho Ling, Senior Member, IEEE, Gregory P. Hunter, Member, IEEE, Yvonne Tran, and Hung T. Nguyen, Senior Member, IEEE (2014). "Brain–Computer Interface Classifier for Wheelchair Commands Using Neural Network With Fuzzy Particle Swarm Optimization". IEEE Journal of biomedical and health informatics, vol. 18, no. 5, September 2014**
11. **R. Rojas (1996). "Neural Networks: A systematic introduction". Springer-Verlag, Berlin 1996**

Appendix 1



Weighted sum calculator simulation result.