CHAPTER 5

HARDWARE IMPLEMENTATION OF PATTERN RECOGNITION SYSTEM

5.1 INTRODUCTION

The pattern recognition unit can be chosen between software and hardware implementation depending upon the requirement of the application. Traditionally, neural networks are implemented in software by training and simulating on sequential Von Neumann type computers. The main limitation in the software implementation of pattern recognition system is that the simulation is executed sequentially. Moreover, for software realisation as it requires huge size equipment like desktop, recognition system such as diagnostic equipment cannot be made portable and the speed of software simulation acts as a constrain if the neural network is of large size (Liu 2005). A few milliseconds delay can be considered important for application like medical diagnosis and the main reason for using hardware approach is to obtain portable diagnostic devices. With advancement in technology computationally intensive applications like pattern recognition for medical diagnosis can be configured as hardware due to its high speed, low cost and short time to market. Current generation of Field Programmable Gate Arrays (FPGAs) contain a very high number of Configurable Logical Blocks (CLB) and are becoming more feasible for implementing massively parallel distributed processing systems. This chapter deals with implementation of pattern recognition of E.coli with its life stage detection system in hardware. For identifying E.coli with its life stage, the prototype model (Figure 5.1)

Figure 5.1 Prototype model of E.coli detection along with its life stage

Presently, the sensory array used in this work is fabricated with MOS technology and the pattern recognizer is software that runs in the computer. It will be efficient if the software part is replaced with hardware as it offers too many advantages for medical diagnosis. Firstly, by making the Artificial Neural Network (ANN) part as neuro-chip, portable device for diagnosis can be developed, which can be used as a point of care instrument. Secondly, inherent parallelism and accuracy can be achieved by hardware implementation. In this chapter, an attempt is made to implement the pattern recognizer part in hardware to enhance the performance and to design a hand held device prototype model, retaining required specificity and sensitivity.
5.2 JUSTIFICATION FOR HARDWARE IMPLEMENTATION

Implementation of ANN is the procedure of mapping the neural model into hardware in an efficient form. Such implementations fall in two categories: general purpose neuro computers for emulating wide range of neural network models and special purpose Very Large Scale Integrated (VLSI) neural systems which are dedicated to specific application with neural network model. Though general purpose neuro computers offer the flexibility and a high degree of observability into the inner working of neural algorithms, it does not provide high speed and compactness for real-time applications.

The special-purpose VLSI chips allow overcoming neural network implementation problems in real-time applications with enhanced processing speed and compactness. But due to fixed mapping, its flexibility is limited and also it is time consuming to develop such chips. With the introduction of field programmable gate arrays (FPGAs), it is feasible to provide custom hardware for application specific computations design. FPGAs are chosen for implementing ANNs due to the reasons that they can implement a wide range of logic gates starting with tens of thousands up to few millions gates. They can be reconfigured to change logic function while resident in the system. FPGAs have short design cycle that leads to fairly inexpensive logic design and have parallelism in their nature. Thus, they have parallel computing environment and allow logic cycle design to work parallel. They also have powerful design, programming and synthesis tools (Nasri 2009, Khalil 2009).

Thus not surprisingly, recent FPGAs comfortably occupy a niche between full custom chip (fast, inflexible, high cost) and pure software realisation (slow, flexible, low development cost) (Ormondi 2006). FPGA based reconfigurable computing architectures are well suited to implement ANNs as they exploit concurrency and can rapidly be reconfigured in
accordance with the weights and topologies of an ANN (Blake 1997, Yamina 1998, Hikawa 2003). Thus, for making wearable medical diagnostic device, FPGAs can be an alternative for replacing the software part of the pattern recognizer as a neuro chip, as they are becoming popular over the years due to its flexibility, great performance and reasonable developing cost.

5.3 DESIGN FLOW AND TOOLS

In this work, VHDL (Very High Speed Integrated Circuits Hardware Description Language) was used as programming language. For simulation and synthesis, tools like Modelsim and Xilinx 8.2i were used. The synthesised code was targeted in Virtex2PXC2VP50FF1152-7.

5.3.1 Overview of VHDL

VHDL is an acronym for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. As a standard description of digital systems, VHDL is used as input and output to various simulation, synthesis and layout tools. The language provides the ability to describe systems, networks and components at a very high behavioural level as well as very low gate level.

The general format of a VHDL program is built around the concept of BLOCKS, which are the basic building units of a VHDL design. Within these design blocks a logic circuit of function can be easily described. A VHDL design begins with an ENTITY block that describes the interface for the design. The interface defines the input and output logic signals of the circuit being designed. The ARCHITECTURE block describes the internal operation of the design. Within these blocks are numerous other functional blocks used to build the design elements of the logic circuit being created. VHDL can be used to take three different approaches to describe the
hardware. These three different approaches are the structural, data flow and behavioural methods of hardware description (Roth 2006, Bhasker 1998).

5.3.2 Design Process using VHDL

The design process can be broken into seven steps as definition of the design requirements, description of the design in VHDL, simulation of the design model, synthesis, Optimization and fitting (Placement and Routing) the design, simulation of the Post-layout design and programming the device (Figure 5.2).

Figure 5.2 VLSI design flow
1. Definition of the design requirements

Before writing the code for the design the design objectives and requirements were clearly studied. This was regarding the function of design, the maximum operating frequency and clock to output times. After having a clear idea of the requirements, the design methodology and device architecture were chosen to initially synthesize the design.

2. Description of design in VHDL

Describing the design in VHDL refers to formulating a design and then coding it (Roth 2006, Bhasker 1998, Navabi 2008).

a. Formulating the design

After having defined the design requirements, the design methodology was chosen to be top-down approach, which divides the design into functional components, each component having specific inputs and outputs and performing a particular function.

b. Coding the design

In this work all blocks were coded using Register Transfer Logic (RTL) modelling (combination of dataflow and behavioural). The syntax and semantics were taken care while coding and it was more hardware specific.

3. Simulation of the source code

After describing the design in VHDL, each block was functionally simulated and the functionality of the corresponding block was verified.
4. **Synthesis, optimisation and fitting the design**

Before synthesis the design was checked for syntax errors and then synthesized to a lower level circuit representation (netlist). As synthesizer is technology specific, Virtex2P XC2VP50FF1152-7 was chosen as target device. The optimisation process depends on three things: the form of Boolean expressions, the types of resources available and automatic or user applied synthesis directives. In this work the netlist was optimized for FPGA and every logic was expressed in the forms other than the sum of products based on device specific resources and directive driven optimization goals.

5. **Placement and routing**

The logic produced by synthesis and optimisation process was placed into the target device and routed. Placement and routing tools have a large impact on the performance of designs. Propagation delays can depend significantly on routing delays. A good placement and route will place critical portion of the circuit close together to eliminate routing delays. Place and route tools use algorithms, directives, user applied constraints and performance estimates to choose an initial placement. Algorithms can then iterate on small changes to the placement to approach a layout that is expected to meet performance requirement. Routing may then begin, with global routing structures or signals that must route over large distances. Local routing structures may then be used to route local inter logic cell and I/O signal paths. The mapping, translation, place and routing were done by Xilinx ise.

6. **Simulation for a Post Synthesis model**

After completion of place and route the design was simulated again after it had been fitted or placed and routed. From post layout simulation both
functionality of the design and timing, such as setup, clock-to-output and register-to-register times were verified. Care was taken that if the design was unable to meet the design objectives; the design was re-synthesized or recoded until it meet the design objectives.

7. Programming the device

After post route simulation verification, the code was downloaded into the target device and physically verified for its functionality.

5.4 HARDWARE REALISATION

The optimised software solution leads generally to build extremely complex circuits depending on the size of the data. In order to develop realistic devices with a convenient accuracy and a reasonable hardware size the following are considered in this section: Neuron model, data representation, logic of activation function implementation and parallelism of NN architecture.

5.4.1 Neuron Model

As neuron is the basic for neural network architecture, initially hardware model was coded for neuron. The neuron model architecture (Figure 5.3) gets all input and the inputs are 'weighted', the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and fed to the activation function to produce the output. The input signal to a given neuron is calculated as follows: The outputs of the preceding neurons of the network (i.e., x₁, x₂ etc.) are multiplied by their respective synapse weights (i.e., w₁, w₂ etc.). These results are added, resulting in the value $y_{in}$
that is delivered to the given neuron. Now the neuron calculates activation function by applying a threshold function for the input value \( y_{in} \) to produce the output value \( Y \) (Sarich 2007). The activation function is usually nonlinear and must be chosen critically, as the performance of the neural network heavily depends on it.

\[ f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \]

**Figure 5.3 Architecture of neuron**

The top-down block represents the architecture of hardware neuron (Figure 5.4). The external RAM acts as input layer since the input layer performs with identity activation function. The hidden layer and output layer are implemented in similar way except in the activation function. In this implementation each neuron is controlled by neural processor (np). The input vector consists of 12 inputs representing the sensor feature vectors, which are stored in external RAM and the 12 weights are stored in the internal ROM. The np is enabled by the signal “ENABLE” in the top down block (Figure 5.4) and is set to logic ‘1’.

Depending upon rising edge of clock, the np places address in the address bus simultaneously for input RAM and weight ROM block. Once it retrieves \( x_1 \) and \( w_1 \), np places in the ALU block where multiplication is
performed and stored in a register. Similarly it gets $x_2 \& w_2$ and the multiplication operation is performed and added with previously stored $x_1w_1$ product value. This final value $x_1w_1 + x_2w_2$ is now stored in register. This process is repeated until the last value is retrieved from RAM and ROM block. After the final computation, np gives the value that is stored in the register to the activation function block. Depending upon the activation function the neuron gives its output. The neurons are interconnected to get the final architecture. Thus the primary function is to implement activation function block together with neural processor and ALU block.

![Figure 5.4 Top-Down design of hardware neuron](image)

### 5.4.2 Data Representation

Any advanced technology gains supremacy over its peers due to its relative increase in speed and accuracy. The innumerable technological innovations like disease diagnosis made accuracy in terms of sensitivity as a
mandatory factor. The manipulation of data collected is done by the arithmetic unit which is more precise in performing the floating point arithmetic operations. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of “sliding window” of precision appropriate to the scale of the number which allows it to represent numbers from $1,000,000,000,000$ to $0.0000000000000001$ with ease and provides results of great accuracy. A few attempts have been made to implement ANNs in FPGA hardware with floating point arithmetic but it was unsuccessful due to the requirement of resources of FPGA (Nichols Moussa 2002, Zhu 2005). But if the arithmetic lacks precision, it can severely limit ANN ability to learn and solve a problem.

While applying Neural Networks in practical applications, definitely there are real values (e.g, -1.3, -0.005, 1.006 etc.). In order to represent the real values, standard formats are needed. Some of the standard formats are Q formats and IEEE754 standard formats. In this work, single precision IEEE754 format is used for representing the values which is one of the most commonly used standards. In single precision format the maximum value that can be represented is $10^{38}$ and the minimum value that can be represented is $10^{-38}$.

Single precision format number consists of 32 bits. The sign of the number is given in the first bit followed by a representation for the exponent (to the base 2) of the scale factor. Instead of the signed exponent $E$, the value actually stored in the exponent field is an unsigned integer ($E = e + 127$).
Hence this is called excess-127 format. Thus E is in the range 0 and 255. The end values of the range 0 and 255 are used to represent some special values.

The binary value for single precision format (Figure 5.5) is given as

\[ \text{Value} = N = (-1)^s \times 2^{E-127} \times (1.M) \]  
(5.1)

Value represented = ±1.M \times 2^{E-127}  
(5.2)

<table>
<thead>
<tr>
<th>S (Sign)</th>
<th>E (Signed Exponent)</th>
<th>M (Mantissa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>8 bit</td>
<td>23 bit</td>
</tr>
<tr>
<td>0 signifies '+'</td>
<td>in excess -127 representation</td>
<td>In fraction</td>
</tr>
<tr>
<td>1 signifies '-'</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.5 Single precision format**

An example of single precision number is illustrated as Figure 5.6

\[ 0 00101000\cdot 0010000000000000000000 \]

Value represented = 1.0010 \times 2^{-87}

**Figure 5.6 Example of a single precision**

### 5.4.3 IEEE754 Floating Point Adder and Subtractor

A floating point algorithm for addition and subtraction takes two inputs with p bits of precision and returns p bits result. The algorithm computes the addition and rounds up the final value to p bits by IEEE754 round to nearest mode algorithm. The IEEE754 addition operation can be explained in four stages. In the first stage, the exponents of the two inputs are
compared by the comparator. If $E_1$ is less than $E_2$ the operands are swapped. This ensures the difference between the exponent to be a positive value. If the signs differ, the 2’s complement of $M_2$ is performed. After $M_2$ is shifted through $(E_1 - E_2)$ positions to the right, thereby aligning the binary point. In the second stage $M_1$ and $M_2$ are added. If the result is negative it should be replaced by its 2’s complements. In third stage, the result is shifted to the left until it is normalised. The normalisation is needed so that the output will not have any carry. The exponent is adjusted according to the number of positions shifted. Finally the result is rounded.

In stage four, the number is rounded using round to nearest mode, which adds 1 to the least significant bit according to the rule. If the rounding causes an over flow, the mantissa is replaced with zeros and “1” is added to the exponent. Then the sign bit is determined. The top level design of adder and subtractor is given in Figure 5.7 and 5.8 and algorithm is illustrated by examples in Table 5.1 and 5.2 respectively.

![Figure 5.7 IEEE754 adder](image)

**Figure 5.7 IEEE754 adder**
Figure 5.8 IEEE754 subtractor

Table 5.1 Example for IEEE754 floating point addition

<table>
<thead>
<tr>
<th>Variable</th>
<th>Decimal</th>
<th>Hex</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>74</td>
<td>42940000</td>
<td>0</td>
<td>10000101</td>
<td>00101000000000000000000000</td>
</tr>
<tr>
<td>B</td>
<td>59.59</td>
<td>426E6666</td>
<td>0</td>
<td>10000100</td>
<td>110111001100110011001101</td>
</tr>
<tr>
<td>C</td>
<td>133.6001</td>
<td>430599A</td>
<td>0</td>
<td>10000110</td>
<td>0000101100110011001101110</td>
</tr>
</tbody>
</table>

Table 5.2 Example for IEEE754 floating point subtraction

<table>
<thead>
<tr>
<th>Variable</th>
<th>Decimal</th>
<th>Hex</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>34</td>
<td>42080000</td>
<td>0</td>
<td>10000100</td>
<td>00100000000000000000000000</td>
</tr>
<tr>
<td>B</td>
<td>58.4</td>
<td>4269999A</td>
<td>0</td>
<td>10000100</td>
<td>110100111001100110011011</td>
</tr>
<tr>
<td>C</td>
<td>-24.4</td>
<td>C1C33333</td>
<td>1</td>
<td>10000001</td>
<td>1000011001100110011001100</td>
</tr>
</tbody>
</table>

The final output of IEEE754 adder and subtractor is got by the following Equation

Result of adder = $X + Y = (X_m \times 2^{X_e}) + (Y_m \times 2^{Y_e})$  \hspace{1cm} (5.3)

Result of subtractor= $X - Y = (X_m \times 2^{X_e}) - (Y_m \times 2^{Y_e})$  \hspace{1cm} (5.4)
The adder and subtractor are coded and their functionalities are verified by behavioural simulation (Figure 5.9, 5.10). The propagation delay was found to be 25 ns.

<table>
<thead>
<tr>
<th>Now: 1000 ns</th>
<th>100</th>
<th>200</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_{[31:0]})</td>
<td>3...</td>
<td>32h41000000</td>
<td>32h40100000</td>
</tr>
<tr>
<td>(b_{[31:0]})</td>
<td>3...</td>
<td>32h41000000</td>
<td>32hC1000000</td>
</tr>
<tr>
<td>(c_{[31:0]})</td>
<td>3...</td>
<td>32h40C00000</td>
<td>32hC0B20000</td>
</tr>
</tbody>
</table>

401C0000 = 2.4375  
C1000000 = -8  
C0B20000 = -5.5625

**Figure 5.9 Behavioural simulation for single precision IEEE754 addition**

<table>
<thead>
<tr>
<th>Now: 1000 ns</th>
<th>100</th>
<th>200</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_{[31:0]})</td>
<td>3...</td>
<td>32h41000000</td>
<td>32h3FA60000</td>
</tr>
<tr>
<td>(y_{[31:0]})</td>
<td>3...</td>
<td>32h40A00000</td>
<td>32hD0100000</td>
</tr>
<tr>
<td>(z_{[31:0]})</td>
<td>3...</td>
<td>32h42900000</td>
<td>32h40900000</td>
</tr>
</tbody>
</table>

C1100000 = -9  
C0400000 = -3  
C0C00000 = -6

**Figure 5.10 Behavioural simulation for single precision IEEE754 subtraction**

**5.4.4 IEEE754 Floating Point Multiplier**

The floating point multiplier has several stages. It also takes two inputs with p bits of precision and returns p bits result. In the first stage, the mantissa part is multiplied by ordinary integer multiplication method because the multiplier deals only with unsigned numbers. In the second stage, the
obtained result is rounded. If the mantissas are unsigned $p$-bit numbers, then the product can have as many as $2^p$ bits and are rounded to a $p$-bit number. The third stage computes the new exponent. The exponent part is added and the sign bit is calculated by logical XOR operation of two sign bits. The floating point multiplication block diagram is shown in Figure 4.11 and its illustration by an example is given in Table 5.3.

![Figure 5.11 IEEE754 multiplier](image)

**Table 5.3 Example for IEEE754 floating point multiplication**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Sign</th>
<th>Exponent</th>
<th>mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>74</td>
<td>42940000</td>
<td>0</td>
<td>10000101</td>
<td>001010000000000000000000000000000</td>
</tr>
<tr>
<td>B</td>
<td>24.4</td>
<td>41C33333</td>
<td>0</td>
<td>10000011</td>
<td>1000011001100110011001100111111</td>
</tr>
<tr>
<td>C</td>
<td>1805.6</td>
<td>44E1B333</td>
<td>0</td>
<td>10001001</td>
<td>1100001101100110011001100110111</td>
</tr>
</tbody>
</table>

The final output is got by the following Equation

$$\text{Result} = X \times Y = (-1)^{X_s} (X_m \times 2^{X_e}) \times (-1)^{Y_s} (Y_m \times 2^{Y_e}) \quad (5.5)$$
The output is also a 32 bit floating point number. For this purpose normalization is needed. Figure 5.12 shows the simulated output for multiplication.

3FA00000 = 1.25  
3FE00000 = 1.75  
3400C000 = 2.1875

Figure 5.12 Behavioural simulation for single precision IEEE754 multiplication

5.4.5 IEEE754 Floating Point Divider

For floating point divider, if the divisor is zero, infinity is returned at the output else if both the inputs are zero a ‘Not a Number’ is returned to the output. For division operation to be performed always the dividend should be greater than the divisor. The sign bit of the result is calculated as the above mentioned procedures. The exponent is got by simply subtracting the exponents of the inputs and adding the bias. Since the division is a complex task the mantissa parts are subtracted using booth division method in which a quotient register and remainder register are used and shifting operation is done to find the value of the result. The result is normalised if needed by shifting the mantissa left and by decrementing the resultant exponent. It is then checked for overflow or underflow. Exponent overflow is returned if the result is larger than the maximum exponent. Similarly exponent underflow is returned if the result is smaller than the minimum exponent (Figure 5.13).
The floating point division is performed by

\[
\text{Result } X / Y = (-1)^{X_s} (X_m \times 2^{X_e}) / (-1)^{Y_s} (Y_m \times 2^{Y_e}) \quad (5.6)
\]

where \( e \) is exponent, \( m \) is mantissa and \( s \) is sign.

The simulated output for IEEE754 division is shown in the Figure 5.14.

\[
41C00000 = 24 \\
40800000 = 4 \\
40C00000 = 6
\]
5.4.6 Activation Function

The behaviour of an ANN depends on both the weights and the input-output function (activation function) that is specified for the units. In this work, all neurons in the hidden layer are subjected to bipolar sigmoid activation function and all neurons at output layer are subjected to soft max activation function. For sigmoid units, the output varies continuously but not linearly as the input changes and they have greater resemblance to real neurons. A softmax function assumes the value of 0 or 1, whereas a sigmoid function assumes a continuous range of values from 0 to 1. The sigmoid function is differentiable, whereas the softmax function is not.

For softmax unit, the function is defined by the Equation (5.7), where $i$ is the value of an output node, $j$ is the net input to an output node and $n$ is the number of output nodes. It ensures all the output values between 0 and 1 and that their sum is 1.

$$y_i = \frac{e^{y_i}}{\sum_{j} e^{y_j}} \quad \text{(5.7)}$$

For sigmoidal activation function as defined by Equation (5.8), whose graph is s-shaped and is by far the most common form of activation function used in the construction of artificial neural networks (Aoyama 2002). It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. An example of the sigmoid function is the logistic function, defined by

$$y = \frac{1}{1 + e^{-\gamma n}} \quad \text{(5.8)}$$

In the software implementation there is no major drawback in computing Equation (5.8). However, from the hardware point of view, sigmoidal function cannot be implemented easily as it contains exponential
series and gives out real value. The real value need to be represented in any one of the standard formats for further implementation. Many digital implementations appeal to some sort of table lookup, for implementing the value given out of the exponential series (Xinsheng Yu 1994, Chalhoub 2006, Eu-Ttum 2002, Muthuramalingam 2002, Zhu 1999). But this approach still demands a great amount of hardware resources and also gives output only for the values present in the lookup table. Direct implementation for non linear sigmoid transfer function is very expensive. In order to overcome this, approximated sigmoidal functions were used (Nordstrom 1992, Chen 2002, Nelson 2002, Vitabile 2005, Yongsoon 2006, Xiaobin 2003, Economou 1994, Ferrer 2004, Hikawa 2003, Wolf 2001 and Kwan 1992, Rossmann 1996). The value obtained by the approximated sigmoid function is tabulated and compared with actual value (Table 5.4).

Precision of activation function is used to tradeoff the capabilities of the realised ANN against the design complexity. A higher precision means fewer quantization errors in the final implementation while a lower precision leads to simpler designs, greater speed and reduction in area requirements and power consumptions.

**Table 5.4 Comparison between real and approximated sigmoid function**

<table>
<thead>
<tr>
<th>Values</th>
<th>Actual Sigmoidal function</th>
<th>Approximated Sigmoidal function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2.00</td>
<td>0.1192</td>
<td>0.1667</td>
</tr>
<tr>
<td>-1.25</td>
<td>0.2227</td>
<td>0.2222</td>
</tr>
<tr>
<td>-1.00</td>
<td>0.2689</td>
<td>0.2500</td>
</tr>
<tr>
<td>-0.02</td>
<td>0.4950</td>
<td>0.4920</td>
</tr>
<tr>
<td>0.00</td>
<td>0.5000</td>
<td>0.5000</td>
</tr>
<tr>
<td>0.02</td>
<td>0.5049</td>
<td>0.5098</td>
</tr>
<tr>
<td>1.00</td>
<td>0.7311</td>
<td>0.7500</td>
</tr>
<tr>
<td>1.25</td>
<td>0.7773</td>
<td>0.7778</td>
</tr>
<tr>
<td>1.50</td>
<td>0.8175</td>
<td>0.8000</td>
</tr>
<tr>
<td>2.00</td>
<td>0.8808</td>
<td>0.8333</td>
</tr>
</tbody>
</table>
For applications like medical diagnosis of infectious diseases which require high precision, this approximated functions or look up table cannot be used for activation function implementations. This work calculates the real value of the sigmoidal activation function where high precision is maintained for meeting accuracy in terms of sensitivity and specificity.

In this work, the exponential function is initially coded by using table – driven implementation method used by Ping Tak Peter Tang. It consists of three main stages. The input value is first reduced to a certain working range. A shifted exponential function is then estimated using known polynomial approximations. Finally, the exponential function of the original input is reconstructed using a certain formula which is explained in Appendix 2. The algorithm includes operations like single precision based IEEE754 based addition, multiplication, division by 32, rounding to the nearest integer, modulo 32, comparison and powers of 2 (Hung 1999). These modules are used as building blocks to construct the floating point exponential function.

The values obtained by table driven method for exponential function are compared with expected values and are tabulated in Table 5.5. From that it is noted that it gives value in high precision and can be extended for sigmoidal and tanh activation functions. For implementing, it further requires IEEE754 adder and divider. The equation of bipolar sigmoidal function is given in the Equation (5.9).

\[ y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \]  

(5.9)
Table 5.5 Comparison of expected and approximated exponential function

<table>
<thead>
<tr>
<th>Values</th>
<th>Expected exponential function</th>
<th>Table driven IEEE754 method</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2.00</td>
<td>0.135335283</td>
<td>0.135335296</td>
</tr>
<tr>
<td>-1.00</td>
<td>0.367879441</td>
<td>0.36787945</td>
</tr>
<tr>
<td>-0.5</td>
<td>0.606530659</td>
<td>0.606530666</td>
</tr>
<tr>
<td>0.00</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.5</td>
<td>1.648721271</td>
<td>1.648721218</td>
</tr>
<tr>
<td>1.00</td>
<td>2.718281828</td>
<td>2.718281745</td>
</tr>
<tr>
<td>2.00</td>
<td>7.389056099</td>
<td>7.389055728</td>
</tr>
</tbody>
</table>

From the Figures 5.15 and 5.16, it can be noted that the post route simulation gives exact value as from Behavioural simulation. Thus this exponential code was used for further implementation of activation function.

Figure 5.15 Behavioural simulation for single precision IEEE754 exponential function

Figure 5.16 Post route simulation for single precision IEEE754 exponential function
5.4.7 **Input RAM Block**

The input RAM block has clock, ram_address (4 bit) bus, ram_in (32 bit) data bus, ram_out (32 bit) data bus, ram_rd (1 bit) read and ram_write (1 bit) signal. For every positive edge of the clock the RAM block is accessed. The read operation is possible only when control unit activates ram_rd signal. According to the address placed in the address bus, the value is placed on the ram_out data bus. Similarly, the write operation is possible only when the control unit activates ram_write signal as active high. According to the ram_address and ram_in bus, the value is stored in the RAM. It is possible to write as well as read simultaneously, within one clock period if both ram_read and ram_write signals are activated (Figure 5.17).

![Figure 5.17 Behavioural simulation for RAM](image)

5.4.8 **Weight ROM Block**

The weight ROM block has clock, np_weight_address (4 bit) bus, np_weight_ out (32 bit) data bus and np_weight_rd (1 bit) read signal. For every positive edge of the clock the ROM block is accessed. The read operation is possible only when control unit activates np_weight_rd signal. According to the address placed in the np_weight_address bus, the value is placed on the ram_out data bus. It is possible to read within one clock period if np_weight_rd is activated (Figure 5.18).
5.4.9 MLP-FF Neural with Off-line Learning

The neuron described above along with the weights, inputs and activation function can be interconnected to form a neural architecture that can be designed for any specific or general applications. This work designs architecture with an input layer, hidden layer and an output layer (Figure 5.19). Here the neuron is trained in offline method. By the method of trial and error at first the neuron is trained and is tested for the next set of inputs and finally when the weights are fixed it is then incorporated into the ROM. In this architecture the first topmost layer is the input layer. The architecture is designed such that only the top module, the RAM is visible as the outermost layer. This design is made such that there is a RAM containing 12 inputs that is to be tested and internal ROM containing 12 weight values that are obtained previously by training. The testing of neuron is performed offline. The inputs and weights are multiplied and then summed up to provide the input to the next hidden layer. In this layer, there are eight neuron modules with the activation function as bi-polar sigmoid function or tanh function that manipulates with the inputs and produces a floating point output. The final output is calculated at output layer.
The output layer utilises softmax activation function and the weights are taken from separate internal ROM. Here the output can be got according to the way in which the neuron is trained.

5.4.10 Central Control Block

The control unit plays a central role in the whole system of processing. This unit’s output controls all other units. The operation of the unit is described as a state flow machine. It maintains parallelism between the input layer and middle layer of NN and also it maintains parallelism between middle layer and output layer of NN. It utilises 60 states for calculation of yin and 12 states for calculation of tanh activation function i.e, y. Similarly for output unit it takes 40 states for calculation of yin and 6 states for calculation of softmax activation function. In the output, a threshold value is set to 0.5. If
the output neuron \( y \) is greater than 0.5 then the control unit gives “1” or else “0”. The simulation of neuron function is verified by simulation (Figure 5.20).

![Figure 5.20 Functional simulation of Neuron](image)

### 5.5 TEST BENCH

Test bench is a model that is used to exercise and verify the correctness of a hardware model. This model generates stimulus for simulation and applies this stimulus to the entity under test and collects the output responses. Here the external input RAM is connected to the neural network architecture. The inputs are stored in the RAM and the output is verified through simulation (Figure 5.21).
5.6 VALIDATION

The function of neural network was tested using XOR problem using neural network which consisted of two neurons in the input layer and one hidden layer with two neurons and one output neuron. The design was simulated synthesised and verified by post route simulation. The result suggests that the coding works properly and can be enhanced with larger number of neurons.