IMPLEMENTATION OF AN INTEGRATED ARTIFICIAL NEURAL NETWORK TRAINED WITH BACK PROPAGATION ALGORITHM

Thesis submitted in the partial fulfillment of requirement for the award of degree of

Master of Technology

in

VLSI Design

Submitted by:

MOHIT JOSHI

Roll No : 601061029

Under the guidance of:

Dr. RAVI KUMAR

Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

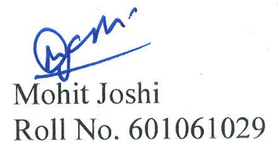
(Established under the section 3 of UGC Act, 1956)

PATIALA – 147004 (PUNJAB)

DECLARATION

I, Mohit Joshi, hereby certify that the work which is being presented in this thesis entitled "Implementation of an integrated artificial Neural Network trained with Back Propagation Algorithm" by me in partial fulfillment of the requirements for the award of degree of Master of Technology in VLSI Design from Thapar University (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision of Dr. Ravi Kumar.

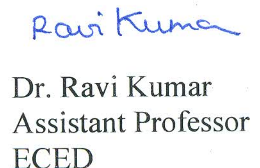
The matter presented in this thesis has not been submitted in any other University / Institute for the award of any other degree.



Date: 13/07/2012

Mohit Joshi
Roll No. 601061029

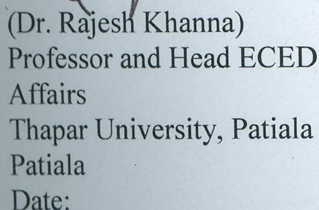
It is certified that the above statement made by the student is correct to the best of my knowledge and belief.



Date: 13/07/12

Dr. Ravi Kumar
Assistant Professor
ECED

Countersigned by:



(Dr. Rajesh Khanna)
Professor and Head ECED
Thapar University, Patiala
Date:

(Dr. S.K. Mohapatra)
Dean of Academic Affairs
Thapar University, Patiala
Date:

// ACKNOWLEDGEMENT

First of all, I would like to express my gratitude to **Dr. Ravi Kumar, Assistant Professor,** Electronics and Communication Engineering Department, Thapar University, Patiala for his patient guidance and support throughout the work. I am truly very fortunate to have the opportunity to work with him. I found this guidance to be extremely valuable.

I am also thankful to **Dr. Rajesh Khanna, Professor & Head,** Electronics and Communication Engineering Department, entire faculty and staff of the department and the friends who devoted their valuable time and helped me in all possible ways towards successful completion of this work. Also I would like to thank **Mr. Arpit Midha**, Consultant, Cadence Design Systems, for his support. I thank all those who have contributed directly or indirectly to this work.

Lastly, I would like to thank my grandparents and parents for their years of unyielding love for constant support and encouragement. They have always wanted the best for me and I admire their determination and sacrifice.

Date: (Mohit Joshi)

Place: Patiala

ABSTRACT

Artificial Neural Network (ANN) is a mathematical model that is inspired by the structureand/or functional aspects of biological neural networks.A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. This thesis is an effort towards the implementation of an integrated ANN trained with backpropagation algorithm. This work discusses the motivations behind the development of ANNs and describes the basic biological neuron and the artificial computational model. It presents ASIC (semi-custom) and FPGA implementation of the network for solving the XOR problem using Fixed-point format (FXP) for representing real numbers. Implementation of squashing function has also been achieved using appropriate approximation techniques. The thesis concludes with a comparison of results obtained for ASIC and FPGA.

CONTENTS

	DECLARATION	i
	ACKNOWLEDGEMENT	ii
	ABSTRACT	iii
	CONTENTS	iv
	LIST OF FIGURES	vi
	LIST OF TABLES	viii
1	**Introduction**	**1**
	1.1. Motivation	1
	1.2. Biological Vs Artificial Neural Network	2
	1.3. Backpropagation Algorithm	7
	1.4. Design Challenges	10
	1.5. Novel aspects of the thesis	15
	1.6. Literature Survey	16
2	**Basic Requirements for ANN Design**	18
	2.1. Optimization of Generic Topology	18
	2.2. Numeric Representation	20
	2.3. General Structure	21
	2.4. Squashing Function	22
3	**Implementation of squashing function**	23
	3.1. Types of squashing function	23
	3.1. Piece-Wise Linear (PWL)	25
4	**Implementation of Main Neural Block**	28
5	**Results and Discussions**	31
	5.1. Functional Simulation	31
	5.2. FPGA Implementation	36
	5.2.1. Synthesis	36
	5.2.2. Translation	40
	5.2.3. MAP	42
	5.2.4. PAR (Place and Route)	43
	5.2.5. STA (Static Timing Analysis)	44
	5.2.6. Power analysis	46
	5.3. ASIC Implementation	48
	5.3.1. Synthesis	48
	5.3.1.1. Reading in the Design	48
	5.3.1.2. Elaborating the Design	50
	5.3.1.3. Constraining the Design	50
	5.3.1.4. Synthesizing the Design	51
	5.3.1.5. Export Design	52
	5.3.2. Placement and Routing	53

6 Conclusions & Future Scope 60

6.1. Conclusion 60

6.2. Future Scope 60

References

LIST OF FIGURES

1	Biological neuron	1
2	Neuron structure and Synapse	3
3	Mathematical Model of Neuron	4
4	Decision boundaries constructed for XOR	6
5	a Architectural graph of network for solving the XOR problem.	6
	b Signal-flow graph of the network	6
6	a Decision boundary constructed by hidden neuron 1 of the network	7
	b Decision boundary constructed by hidden neuron 2 of the network	7
	c Decision boundaries constructed by the complete network	7
7	Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals	8
8	Illustrating error-correction learning	9
9	Implementation options for digital systems	11
10	The general synthesis flow of an FPGA-based and ASIC design	12
11	The general RTL synthesis flow	13
12	The general flow of physical synthesis	14
13	Illustrating error-correction learning	19
14	IEEE standard 754-1985 format for single precision	20
15	Format of an FXP format	21
16	General structure of ANN	21
17	2:2:1 topology used for solving XOR problem	22
18	Types of activation functions	24
19	PWL function implemented	26
20	Symbol generated for PWL and its differential	27
21	Symbol generated for the main neural block	28
22	Architecture of the network used	29
23	PWL implementation for non-linear activation function	31
24	Simulation result for the PWL module	32
25	Simulation result for the DIFFPWL module	33
26	(a) Simulation result for the NEURAL_BLOCK_1 module using testbench	34
	(b) Simulation result for the NEURAL_BLOCK_1 module	35
27	XST Design Flow.	37
28	HDL analysis report	38
29	HDL synthesis report obtained from XST showing the total number of design building blocks required after HDL synthesis	39
30	Device utilization summary	39
31	Report after low level optimization	40
32	NGDBuild design flow	41

33 Translation report of the design 41
34 MAP design flow 42
35 Device utilization summary after mapping the design to the target FPGA 43
36 PAR flow 44
37 XPower results summary 47
38 Input and output files for RTL Copiler 48
39 RTL Compiler work flow 49
40 Input and output files for First Encounter 53
41 Generic flow of First Encounter 54
42 Floorplanning and power planning done 57
43 Design placed (Physical view) 57
44 Design placed (Amoeba View) 58
45 Buffers and inverters added during CTS 58
46 Design routed 59
47 Timing analysed of the design 59

LIST OF TABLES

1 IEEE 754 binary formats 20
2 Inferred blocks for each design unit 38
3 STA Results 45
4 Hierarchical division of power among different modules 46
5 Synthesis results for the design 52
6 Timing results for STA done at various stages 54
7 General Design Information 55
8 Netlist Information 55
9 Power Information 55
10 Floorplan/Placement Information 55
11 Area of Power Net Distribution 56
12 Wire Length Distribution 56

1. INTRODUCTION

1.1. Motivation

Human brain is the most extraordinary and complex creation in the universe. It has made human beings stand apart from the animal kingdom. The human brain, being the most intelligent device on the earth, drives us being the ever-progressive species on the planet.

The advantage of human brain is its massive parallelism, the highly parallel computing structure. The human brain is a collection of approximately $\mathbf{10^{11}}$ computing elements called neurons (shown in figure 1). Neurons are living cells with axons (single long fibre) and dendrites (treelike networks of nerve fibres) that form interconnections through electro-chemical synapses, with a density of approximately $\mathbf{10^{4}}$synapses per neuron. Signals are transmitted through the cell body (soma), from the dendrite to the axon as an electrical impulse, by raising or lowering the electrical potential inside the body of the receiving cell. If the potential reaches a threshold, a pulse is sent through axon and the cell is sad to have 'fired'.

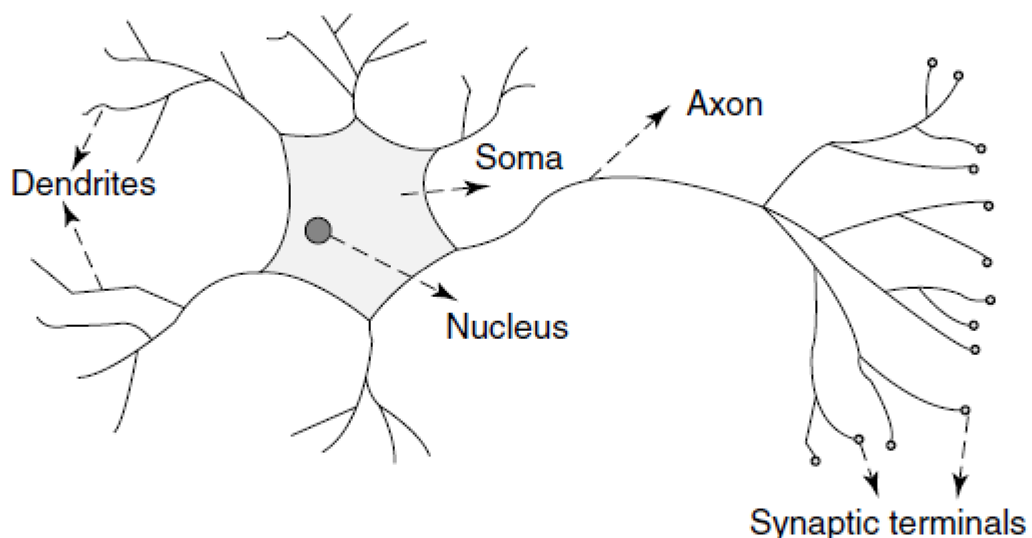


Figure 1.Biological neuron [1].

Man always tried to make machines that could do intelligent job processing, and take decisions on its own. The result was Computer. Even though it could perform millions of calculations every second, display incredible graphics and 3-dimentional animations, play

audio and video but it made the same mistake every time. Practice could not make it perfect. So the question for making more intelligent device continued. Then the idea of initiating human brain stuck the designers who started their researches, giving rise to Artificial Neural Networks.

Synthetic networks that emulate the biological neural networks found in living organisms are called Artificial Neural Networks. Artificial neural networks have undoubtedly been biologically inspired, but the close correspondence between them and real neural systems is still rather weak. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. Modern neural networks are non-linear statistical data modelling tools. They are usually used to model complex relationships between inputs and outputs or to find patterns in data.

The above mentioned properties of an ANN serve as a primary motivation for their on-chip implementation. This work comprehensively summarizes the efforts towards the implementation of individual ANN modules.

1.2. Biological Vs. Artificial Neural Network

Biological Neural Network

A biological Neural Network is a series of interconnected biological neurons. A biological neuron receives inputs from other sources, combines them in some way, performs a generally nonlinear operation on the result, and then output the final result. Output can be excited or not excited, subject to attenuation in the synapses, which are junction parts of the neuron. Incoming signals from other neurons determine if the neuron shall excite ("fire"). Figure 2 shows neuron structure.

The facts about Biological Neural Networks which motivated humans to implement architecture similar to them

- The number of neurons in the human brain: 10^{11}
- The average number of connections of each neuron: 10^{4}
- Highly parallel computation

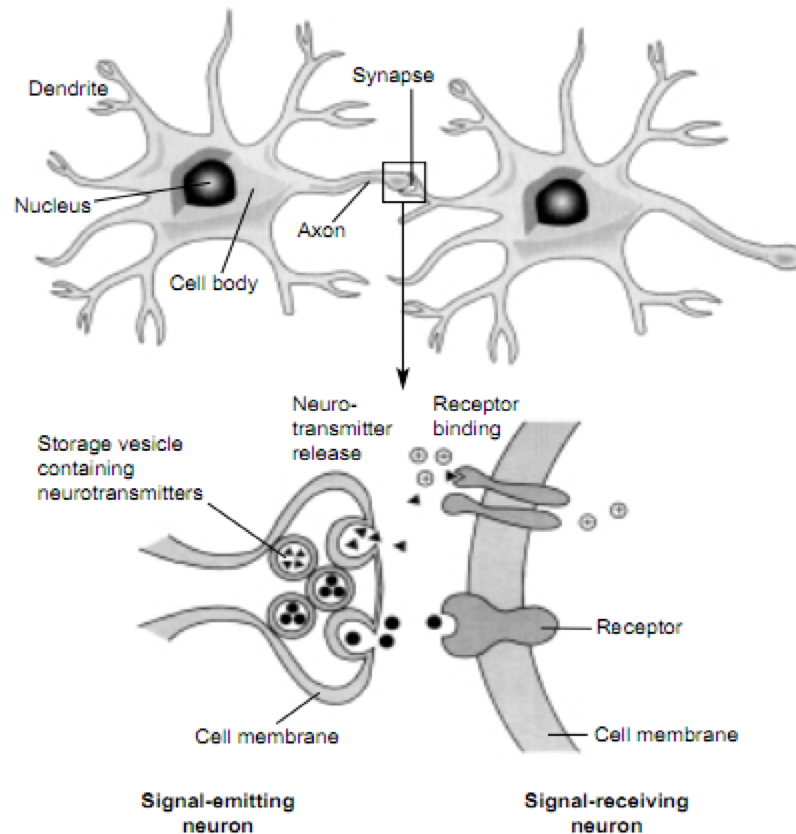


Figure 2. Neuron structure and Synapse [2]

Few terms related to the Biological Neural Network:

- **Neuron**: Electrically excitable cell that processes and transmits information by electrical signalling.
- **Dendrites**: Branches of neurons that receive signals from other neurons and pass the signals into the soma.
- **Soma**: Cell body of the neuron.
- **Axons**: The interface through which neurons interact with their neighbouring neurons
- **Synapse**: Electrochemical contact between Neurons.

Hebb's Rule: The synapse resistance to the incoming signal can be changed during a "learning" process, following quoted by **Donald Olding Hebb** in his book "The Organization Behavior" [1949], later known as Hebb's Rule:

Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.... When an axon

of cell A is near enough to excite cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased [3].

Artificial Neural Network

Artificial Neural Network (ANN) is a mathematical model that is inspired by the structure and/or functional aspects of biological neural networks. A neural network consists of an interconnected group of artificial neurons, and it processes information using a connectionist approach to computation. The block diagram of Figure 3 shows the mathematical model of a neuron, which forms the basis for designing ANNs. Here we identify three basic elements of the neuronal model:

1. A set of ***Synapses*** or connecting Links, each of which is characterized by a ***Weight*** or ***Strength*** of its own. Specifically, a signal $\mathbf{X_i}$ at the input of synapse $\boldsymbol{i}$ connected to neuron $\boldsymbol{k}$ is multiplied by the synaptic weight $\mathbf{W}_{\boldsymbol{ki}}$.
2. An ***Adder*** for summing the input signals, weighted by the respective synapses of the neuron.
3. An ***Activation Function*** for limiting the amplitude of the output of a neuron. The activation function is also referred to as a ***Squashing Function*** in that it squashes (limits) the permissible amplitude range of the output signal to some finite value.

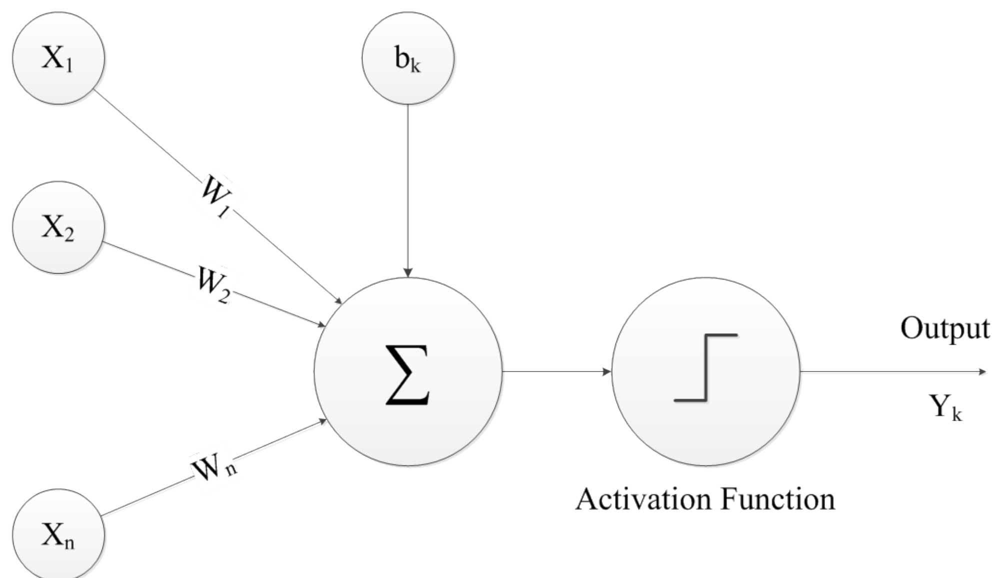


Figure 3. Mathematical Model of Neuron.

The neuronal model of Figure 3 also includes an externally applied ***Bias***, denoted by $\boldsymbol{B_k}$. The bias $\boldsymbol{B_k}$ has the effect of increasing or lowering the net input of the activation function, depending on whether it is positive or negative, respectively.

In mathematical terms, we may describe a neuron k by writing the following pair of equations:

$$u_k = \sum_{i=1}^{n} w_{ki} \times x_i + b_k \qquad 1.1$$

and

$$y_k = \phi(u_k) \qquad 1.2$$

where $\boldsymbol{X_i}$ are the input signals; $\boldsymbol{W_{ki}}$ are the synaptic weights of neuron k; $\boldsymbol{B_k}$ is the bias; $\boldsymbol{U_k}$is the adder output, $\varphi(\cdot)$ is the *activation function*; and Y_k is the output of the neuron. The use of bias $\boldsymbol{B_k}$ has the effect of applying an affine transformation to the output $\boldsymbol{U_k}$ of the linear combiner in the model of Figure 3.

The XOR Problem

In the single-layer perceptron there are no hidden neurons. If a classification is linearly separable (as in the case of AND/OR/NAND/NOR), we can use single-layer perceptron. XOR is not linearly separable as can be seen in the Figure 4, consequently it cannot be implemented using single layer network; a three layered network (Figure 5) is required to solve the problem. The XOR can be classified into two groups as:

$$\left.\begin{array}{l} 0 \oplus 0 = 0 \\ 1 \oplus 1 = 0 \end{array}\right\} \textbf{Class 0}$$

$$\left.\begin{array}{l} 0 \oplus 1 = 1 \\ 1 \oplus 0 = 1 \end{array}\right\} \textbf{Class 1} \qquad 1.3$$

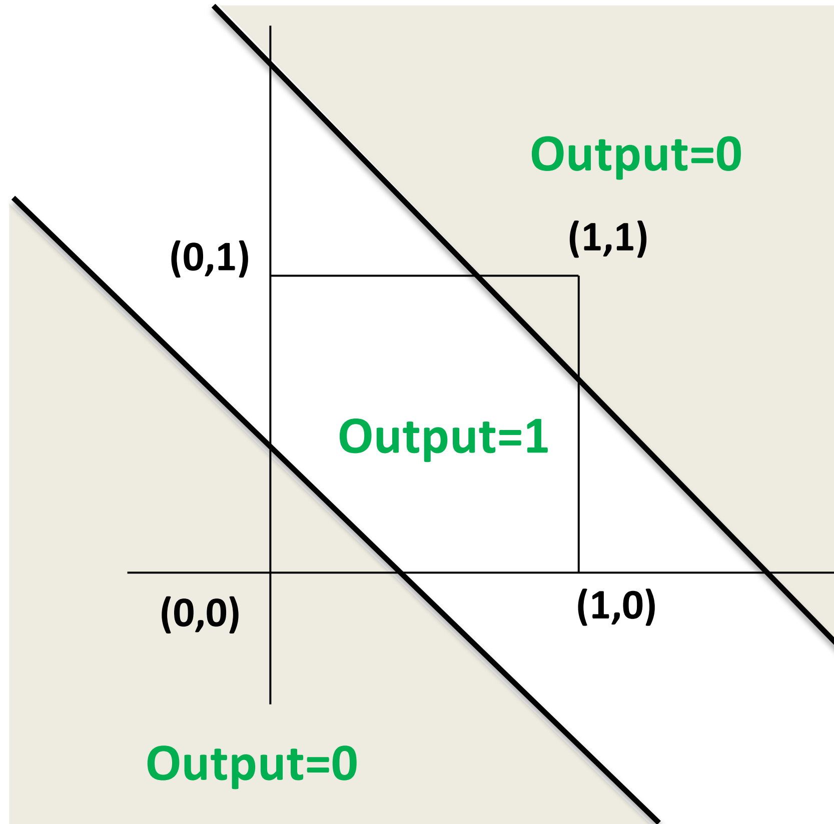


Figure 4. Decision boundaries constructed for XOR.

Apart from the solution given in Figures 5, and 6; other solutions can be implemented like

$a \oplus b = (\overline{a \wedge b}) \wedge (a \vee b)$, $\quad a \oplus b = (a \vee b) \wedge (\overline{a \wedge b})$, ….. [4]

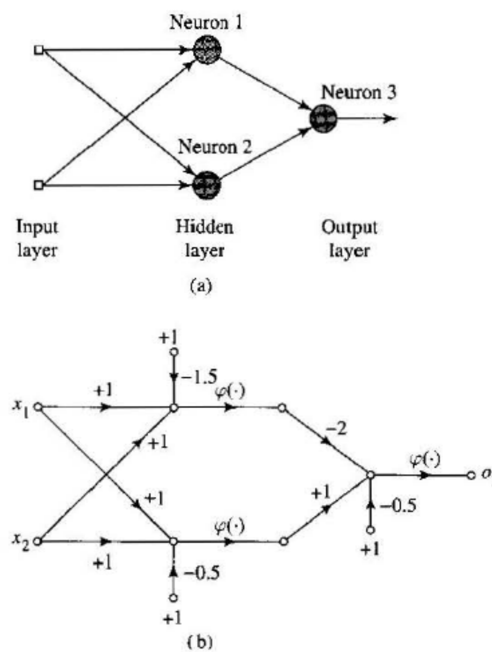


Figure 5. (a) Architectural graph of network for solving the XOR problem.
(b) Signal-flow graph of the network[3].

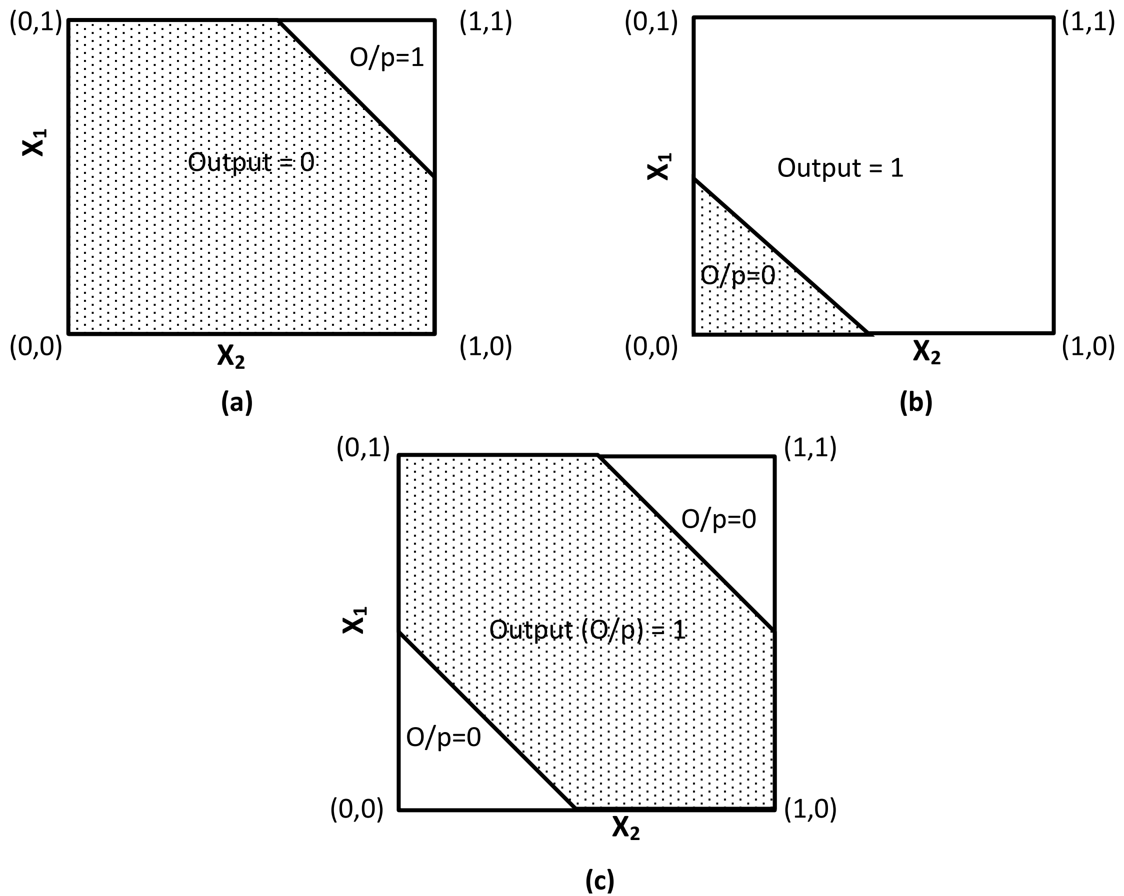


Figure 6.(a) Decision boundary constructed by hidden neuron 1 of the network in Figure 5.
(b) Decision boundary constructed by hidden neuron 2 of the network.
(c)Decision boundaries constructed by the complete network[2].

1.3. Backpropagation Algorithm

The Backpropagation is a common method of training artificial neural networks. It is an error correction learning method (explained below), and is a generalization of the delta rule. Error data at the output layer is "back propagated" to the previous layer of neuron, thus allowing the updation of the weights of these layers.

The algorithm has two passes for error correction, they are:

1. Forward Pass

a) Error is calculated from outputs

b) Used to update output weights

2. Backward pass

a) Error at hidden nodes is calculated by back propagating the error at the outputs through the new weights

b) Hidden weights updated

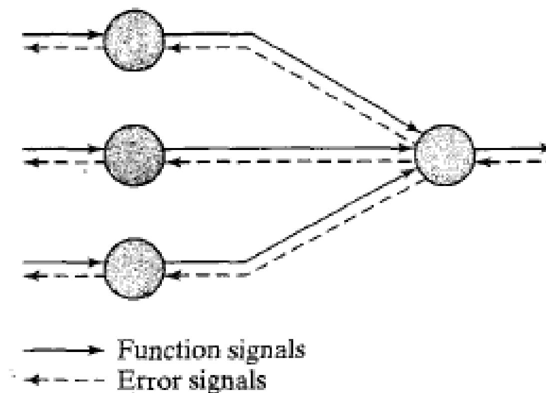


Figure 7 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals.

Figure 7, shows the forward pass of the signal and backward pass of the error signal.

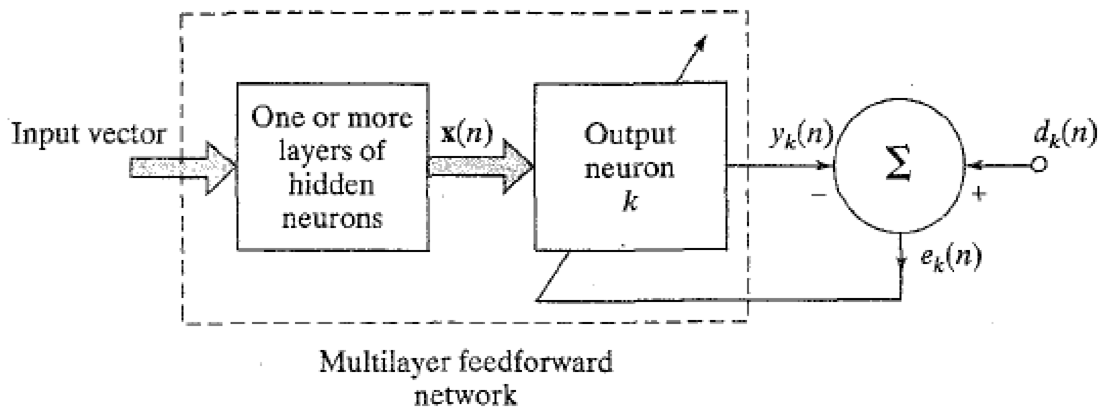
Error-Correction Learning

To illustrate the error correction learning rule, consider the simple case of a neuron k constituting the only computational node in the output layer of a feed-forward neural network, as depicted in Figure 8. Neuron *k* is driven by a *signal vector X(n)* produced by one or more layers of hidden neurons, which are themselves driven by an input vector (stimulus) applied to the source nodes (i.e., input layer) of the neural network. The argument *n* denotes discrete time, or more precisely, the time step of an iterative process involved in adjusting the synaptic weights of neuron k. The output signal of neuron k is denoted by $y_k(n)$. This output signal, representing the only output of the neural network, is compared to a desired response or target output, denoted by $d_k(n)$. Consequently, an error signal, denoted by $e_k(n)$, is produced. By definition, we thus have

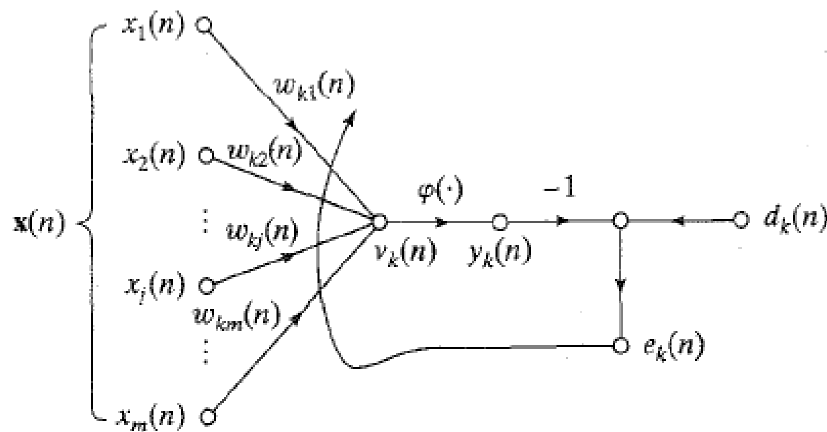
$$e_k(n) = d_k(n) - y_k(n) \quad 1.4$$

The error signal $e_k(n)$ actuates a *control mechanism*, the purpose of which is to apply a sequence of corrective adjustments to the synaptic weights of neuron k. The corrective adjustments are designed to make the output signal $y_k(n)$ come closer to the desired response $d_k(n)$ in a step-by-step manner. This objective is achieved by minimizing a *cost function or index of performance*, $\xi(n)$, defined in terms of the error signal $e_k(n)$ as:

$$\xi(n) = \frac{1}{2} e_k{}^2(n) \qquad 1.5$$



(a) Block diagram of a neural network, highlighting the only neuron in the output layer



(b) Signal-flow graph of output neuron

Figure 8.Illustrating error-correction learning [3].

That is, $\xi(n)$is the *instantaneous value of the error energy*. The step-by-step adjustments to the synaptic weights of neuron *k* are continued until the system reaches a steady state (i.e., the synaptic weights are essentially stabilized).At that point the learning process is terminated.

The learning process described herein is obviously referred to as *error-correction learning.* In particular, minimization of the cost function *ξ(n)* leads to a learning rule commonly referred to as the *delta rule* or *Widrow-Hoff rule,* named in honour of its originators (Widrow and Hoff, 1960). Let $w_{kj}(n)$ denote the value of synaptic weight w_{k_j} of neuron *k* excited by element $x_j(n)$ of the signal vector *x(n)* at time step *n.* According to the delta rule, the adjustment $\Delta w_{kj}(n)$ applied to the synaptic weight w_{k_j} at time step *n* is defined by

$$\Delta w_{kj}(n) = \eta e_k(n) \cdot x_j(n) \qquad 1.6$$

where η is a positive constant that determines the *rate of learning* as we proceed from one step in the learning process to another. It is therefore natural that we refer to η as the *learning-rate parameter*. In other words, the ***delta rule*** may be stated as:

The adjustment made to a synaptic weight of a neuron is proportional to the product of the error signal and the input signal of the synapse in question.

1.4. Design Challenges

Real numbers are not synthesizable in digital systems. Also analog systems consume lesser power compared to their digital counterparts. Analog systems can achieve higher speed, and they are also more area-efficient than their digital counterparts. Analog systems can be directly interfaced with the real world (data convertors not needed unlike in digital systems). Also the nonlinear activation function is easy to implement in analog systems. But the analog system have certain disadvantages like – storage of weights, they are more susceptible to temperature and power supply variations, crosstalk. The major issue is obtaining linear multiplier over a wide range of operation.

The digital system can be implemented either as an ASIC or an FPGA. Both have certain advantages and disadvantages listed below. According to the need of the design, the designer must look for a trade-off between the parameters.

ASICs have higher speed when compared to FPGA. As ASICs are designed for specific application they can be optimized to achieve maximum speed, hence we can have high speed in ASIC designs. FPGAs contain lots of LUTs, and routing channels which are

connected via bit streams (program). As they are made re-usable and made for general purpose. They are in-general larger designs than corresponding ASIC design. Also FPGA consume much more power when compared to ASICs. This can be explained as – unused circuitry contributes in leakage power. So ASICs permit us to optimize power to the maximum. ASICs are cost effective for very high design volumes are fabricated, for research purposes FPGAs serve as a better option.

On the other hand FPGAs serves some purposes better than ASICs, these are – faster time-to-market (can be contributed to elimination of the complex and time-consuming floorplanning, place and route, timing analysis, and mask/re-spin stages of the project since the design logic is already synthesized to be placed onto an already verified, characterized FPGA device in the FPGA design flow). No upfront non-recurring expenses (NRE) in FPGA (NRE refers to the one-time cost of researching, designing, and testing a new product, which is associated with ASICs).Simpler design cycle can be contributed to the software that handles much of the routing, placement, and timing. FPGAs are Field reprogramable i.e., A new bitstream can be uploaded, while ASICs are only one time programmable [5].

VLSI (digital) system implementations can be classified into three classes: *ASICs, µP/DSP systems, and field programmable devices*, as shown in figure 9. Implementations based on HDL can be synthesized as either FPGA or cell based ASIC. The general synthesis flow of an FPGA-based and ASIC design is shown in figure 10.

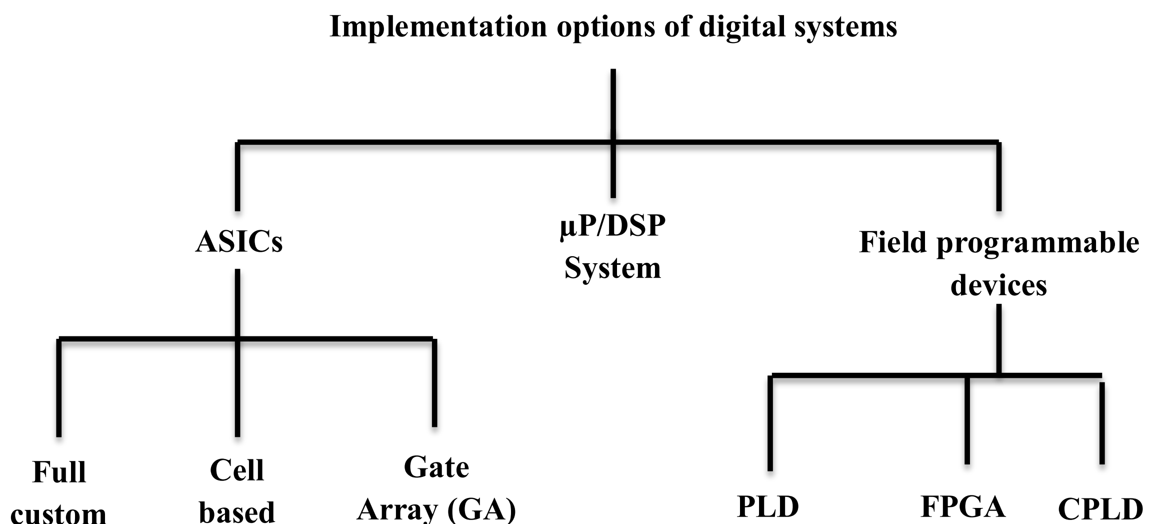


Figure 9.Implementation options for digital systems [5].

In figure 10, we have divided the synthesis flow into two major parts: *front end* and *back end.* The front end is target-independent and contains three phases, starting from product requirement, behavioral/RTL description and ending with RTL synthesis, and generates a gate-level netlist. The back end is target-dependent and mainly comprises the physical synthesis, which accepts the structural description of a gate-level netlist and generates a physical description. In other words, the RTL synthesis is at the heart of the front-end part and the physical synthesis is the essential component of the back-end part.

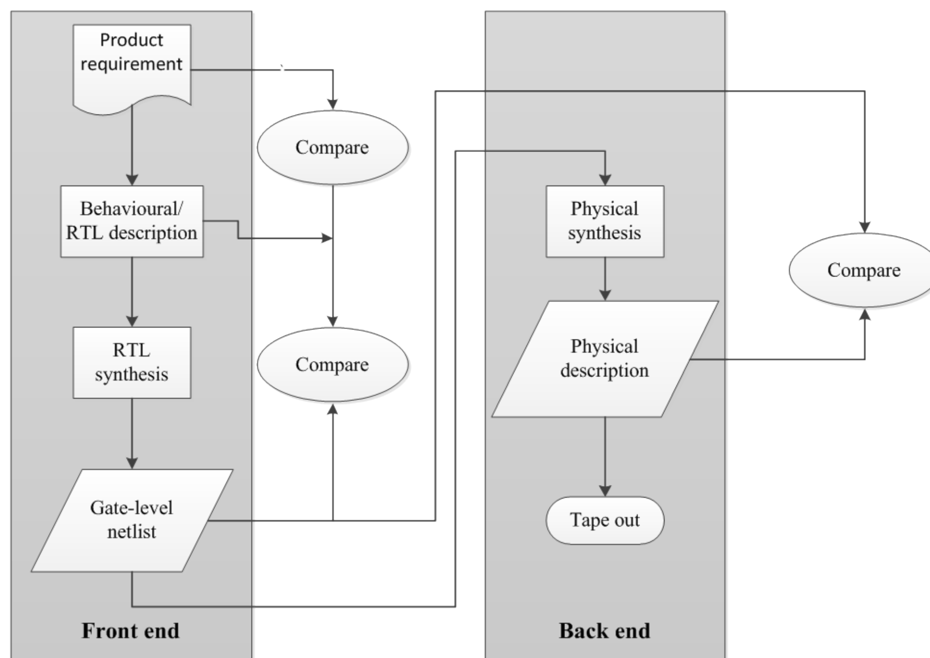


Figure 10.The general synthesis flow of an FPGA-based and ASIC design [5].

RTL synthesis flow: The general RTL synthesis flow is shown in figure 11. The RTL synthesis flow begins with design specification, which is then described with an RTL code (either in VHDL or Verilog HDL). The results are then verified by using a set of test benches written in HDL. This verification process is known as *RTL functional verification*. The functional verification ensures that the function of the design entry is correct and conforms to the specification, in addition to check the design for syntactical errors.

The RTL description is synthesized by a logic synthesizer after functional verification. This process is termed as *RTL synthesis* or *logic synthesis*. The essential operation of logic synthesizer is to convert an RTL description into generic gates and registers, and then optimize the logic to improve speed and area. In addition, datapath optimization and

power optimization can also be performed at this stage. A logic synthesizer accepts three inputs: *RTL code, technology library,* and *constraints*, and generates a gate-level netlist.

After the gate-level netlist is generated, it is again verified by test benches used in the functional verification stage, to check whether they produce the same results.

The next three steps often used in ASIC but not in FPGA-based design are *scan-chain logic insertion, resynthesis*, and *verification*, as shown in figure (shaded block B) .the scan-chain (or test logic) insertion step is to insert or modify logic and registers to aid in the manufacturing test. *Automatic test pattern generation* (ATPG) and *built-in self-test* (BIST) are generally used in ASIC designs.

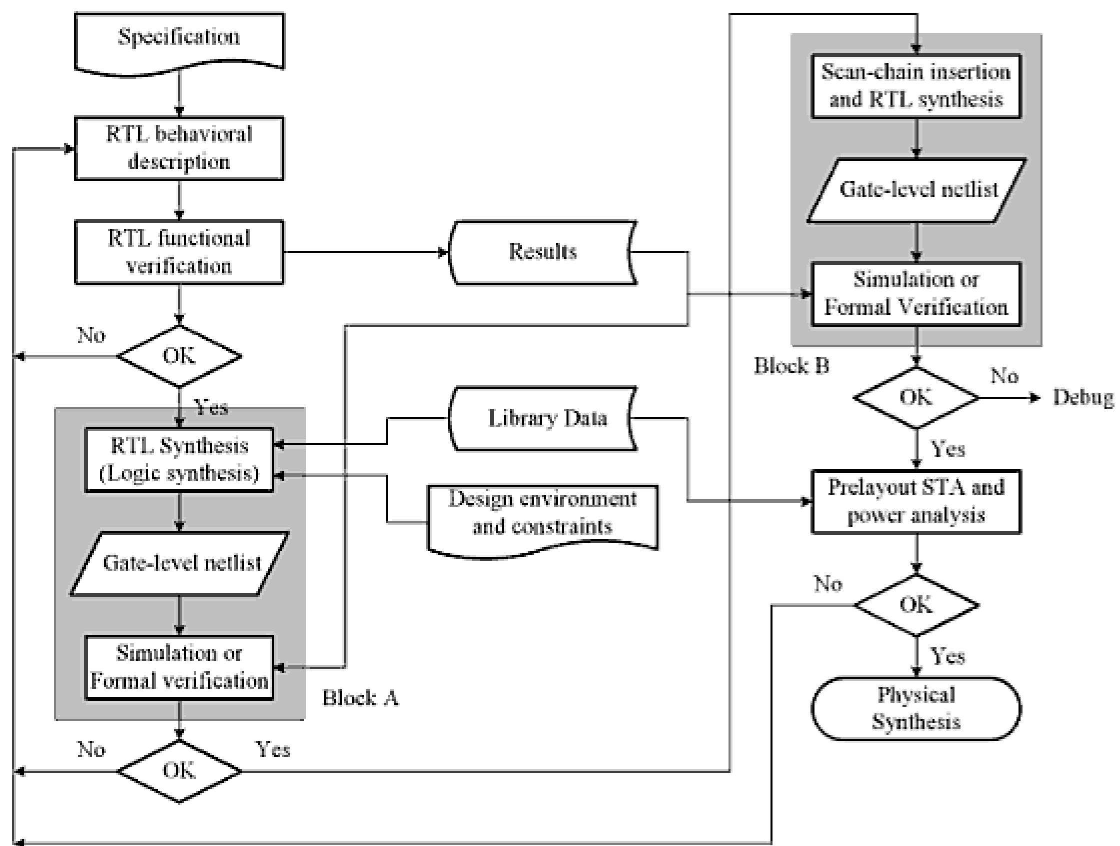


Figure 11. The general RTL synthesis flow [5].

The final stage of RTL synthesis flow is the pre-layout *static timing analysis* (STA) and power dissipation analysis. The STA is a timing analysis alternative to the *Dynamic timing analysis* (DTA), which is performed by simulation, by analyzing the timing paths of the design without carrying out any actual simulation. Through detailed STA, many

timing problems can be corrected and system performance might also be optimized. Power analysis estimates the power dissipation of the circuit.

Physical synthesis flow: The second part of the synthesis flow of an FPGA-based or ASIC system is the physical synthesis. In this part we have to choose a target (either FPGA or a cell library). Regardless of the FPGA-based or ASIC system, the physical synthesis can further be subdivided into two major stages: *placement* and *routing*, as shown in figure 12. Physical synthesis is generally called *place and route* (PAR / P&R) in CAD tools.

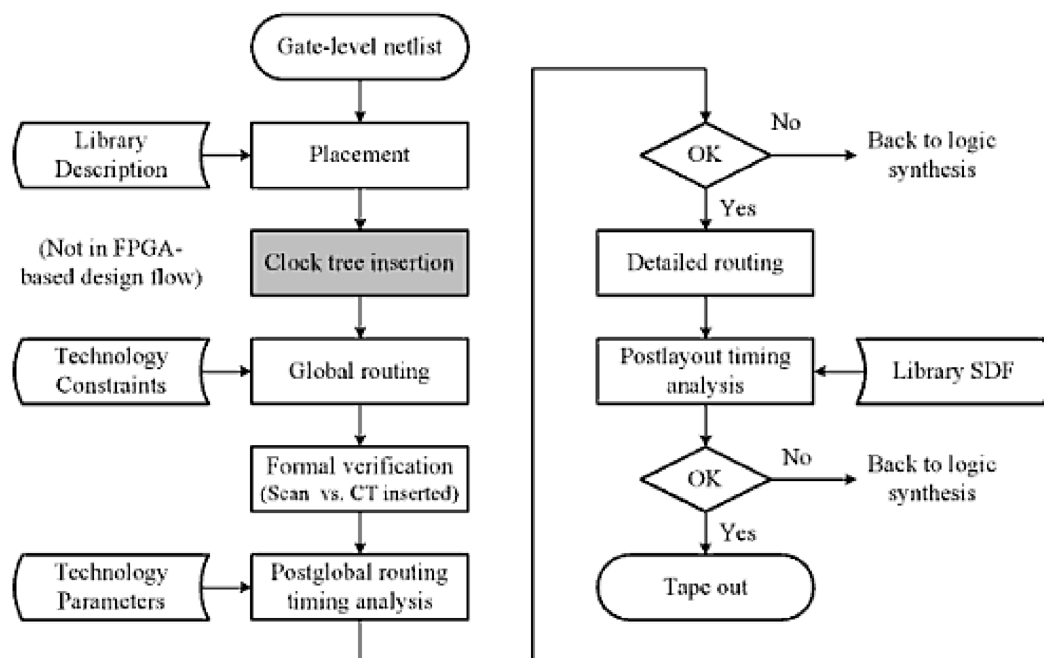


Figure 12.The general flow of physical synthesis [5].

In the placement stage, logic cells are placed at fixed positions to minimize the total area and wire lengths. In other words, the placement stage defines the location of logic cells on a chip and sets aside the space for the interconnect of each logic cell. This stage is generally a mixture of three operations: *partitioning, floorplanning*, and *placement*.

Partitioning divides the circuit into parts such that the sizes of the components are within prescribed ranges and the number of connections between components is minimized. Floorplanning determines the appropriate location of each module in a rectangular chip area. Placement finds the best position of each module on the chip such that the total chip area is minimized or the total length of wires is minimized.

After placement, a clock tree is inserted in the design (only for ASIC designs, as clock distribution network is already fixed in FPGAs), also known as *clock tree synthesis* (CTS). In this step, a clock tree is generated and routed coupled with the required inverters and buffers. A clock tree is generally placed before the main logic placement and routing is completed to minimize the clock skew.

The next stage is known as *routing*, which is used to complete the connections of the signal nets among the cell modules placed in the previous stage. This stage is subdivided into two substages: *global routing* and *detailed routing*. Global routing decomposes a large routing problem into small manageable sub-problems, by finding a rough path for each net to reduce chip size, shorten wire lengths, and evenly distribute the congestion over the routing area. Detailed routing carries out the actual connections of signal nets among the modules.

After both global and detailed routing, a separate STA for each of these two steps is performed. These timing analyses rerun the timing analysis with the actual routing loads placed on the gates to check whether the timing constraints are still valid.

The final tape-out stage has different meanings for ASICs and FPGA-based syntheses. For ASICs, the tape-out stage generates the photomasks so that the resulting designs can be “programmed” in an IC. For FPGA-based syntheses, the tape-out stage generates the programming file to program the device.

1.5. Novel aspects of the thesis

- The present work focuses on hardware implementation of Artificial Neural Network, which is capable of resolving paradigms that linear computing cannot.
- Real numbers cannot be synthesized to hardware. Two alternate formats are available for representing real numbers they are fixed point and floating point formats. So in the work I choose fixed point format (available in VHDL 2008) which is precise, faster, and fewer complexes, than floating point. But none of the Xilinx XST or Cadence Encounter supports VHDL 2008. So I’ve to search for a VHDL 93 compatible fixed package which doesn’t support division so I’ve used multiplication in place of division.

- ASIC and FPGA are the two different flows available for implementing digital circuits, each having some merits over other. I've implemented the network using both the flows.
- The network is generally implemented in separate modules for error generator, weight update, and synapse. I've integrated all these units into a single module.
- The present work also suggests the use dual FXP format for the future work.

1.6. Literature Survey

The year 1943 is often considered the watershed in the development of artificial neural systems. McCulloch and Pitts (1943) outlined the first formal model of an elementary computing neuron. The model included all necessary elements to perform logic operations, and thus it could function as an arithmetic logic computing element. The implementation of its compact electronic model, however, was not technologically feasible during the era of bulky vacuum tubes. The formal neuron model was not widely adopted for the vacuum tube computing hardware description, and the model never became technically significant. However, the ***McCulloch-Pitts neuron model*** laid the groundwork for future developments.

Donald Hebb (1949) [3] first proposed a learning scheme for updating neuron's connections that we now refer to as the ***Hebbian learning rule***. He stated that the information can be stored in connections, and postulated the learning technique that had a profound impact on future developments in this field. Hebb's learning rule made primary contributions to neural networks theory.

The neuron-like element called a ***perceptron*** was invented by Frank Rosenblatt in 1958 [6]. It was a trainable machine capable of learning to classify certain patterns by modifying connections to the threshold elements (Rosenblatt 1958). The idea caught the imagination of engineers and scientists and laid the groundwork for the basic machine learning algorithms that we still use today.

The first VLSI realisation of Neural Networks was done by Carver A. Mead and M. A. Mahowald [7]. They created the first neurally-inspired chips, including the ***silicon retina*** and chips that learn from experience. Later Mahowald M. and Douglas R. implemented ***silicon neuron*** on analog chip [8].

There have been several attempts to build custom application specific integrated circuits (ASICs) for the network that include multiple parallel processing units [13] – [15]. However the network implemented on ASICs were constrained by the non-reconfigurability of ASICs unlike FPGA. So more recently, the focus on implementing ANN hardware shifted towards reconfigurable hardware of which FPGAs are the most preferred among them. Thus FPGA implementation allows more flexibility of the constraints like network size, type, and topology [16] – [19]. FPGAs provide similar logic density as that of ASICs with the flexibility of quick design and test cycles, making them preferred choice for research purposes.

When implementing the BPN on FPGAs the design poses few challenges like weight precision and activation function implementation [20]. Weight precision issue is related to the choice of format used for numeric representation. Higher weight precision means fewer quantization errors, while a lower precision leads to simpler designs, higher speed, area, and power reductions [21]. One must find "minimum precision" required for the problem in order to resolve the trade-off between the constraints discussed also termed as *area versus precision* design trade-off. Non-linear activation function implementation in digital design is also a great challenge. Sigmoid function directly cannot be directly implemented in digital system, there are two practical approaches discussed in the literature to approximate non-linear sigmoid function. They are **Piece-wise linear (PWL)** approximation and **Lookup tables.**

2. BASIC REQUIREMENTS FOR ANN DESIGN

2.1. Optimization of Generic Topology

Figure 13, shows the general layout and interconnections of data and control in the network. The layout consists of four major units: the forward stage, BP stage, weight update stage, and the controller.

1. *Forward Stage:* The forward stage module consists of neurons for both hidden and output layers. It evaluates output for each neuron. In Figure 13, the outputs of all neurons are marked OUTPUTS, and the first derivatives of those outputs are marked OUTPUTS´.

2. *BP Stage:* The BP stage module calculates the error between the final output and the desired output. Following error calculation, a delta value is calculated for each of the output neurons. Later this delta is back-propagated to the hidden neurons based on the output deltas and the associated weights in the output layer and further deltas calculated for the input layers also based on the hidden deltas and associated weights with the hidden layers.

3. *Update Stage:* The update stage adjusts the network's weights according to the deltas, the learning rate, and the input to the corresponding layer. The adjustment value is added to the existing weight to produce a new weight for the next cycle of the forward stage.

4. *Controller Unit:* The controller unit is used for data routing and timing during operation of the three previous stages. The controller has a signal for each stage.

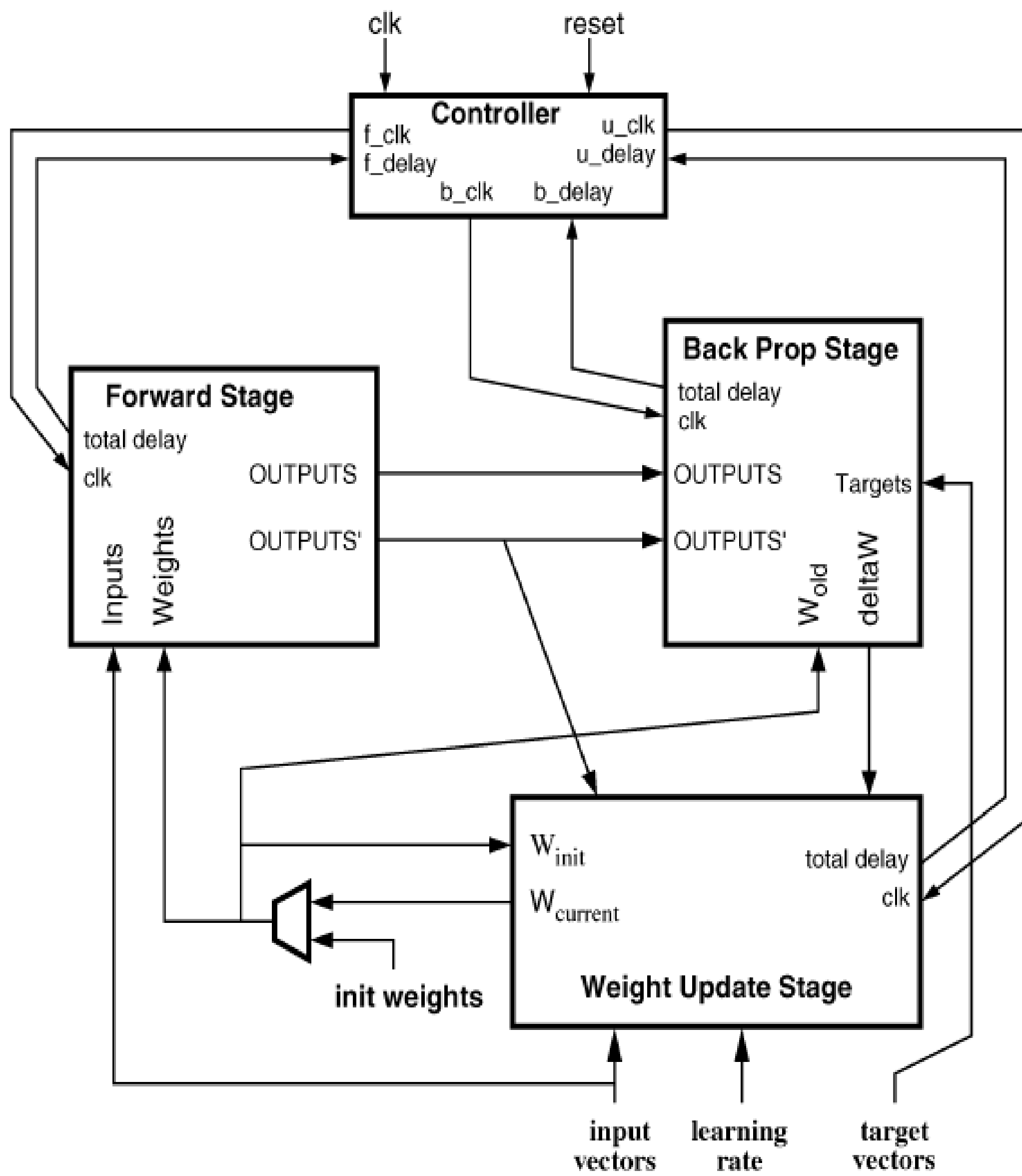


Figure 13.Illustrating error-correction learning [22].

In the present work, we have integrated all the four units in the main module.

2.2. Numeric Representation

VHDL supports binary, integers for synthesis while real numbers can be used for simulation purposes only, they are not synthesizable. Fixed-point format (FXP) and floating-point (FLP) format are both methods of representing real numbers. So for digital signal processing, FLP and FXP are used. Because fixed-point and floating-point operations can produce results that have more bits than the operands, there is possibility for information loss.

1. *Floating-point Format:* In general, while using FLP represented approximately to a fixed number of significant digits and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. The typical number that can be represented exactly is of the form:

$$\pm \boldsymbol{d.dd \ldots d \times \beta^{e}} \qquad 2.1$$

More precisely $\pm \boldsymbol{d_0 d_1 \ldots d_{p-2} d_{p-1} \times \beta^{e}}$ represents the number

$$\pm(\boldsymbol{d_0} + \boldsymbol{d_1} \times \boldsymbol{\beta^{-1}} + \cdots + \boldsymbol{d_{p-1}} \times \boldsymbol{\beta^{-(p-1)}}) \times \boldsymbol{\beta^{e}}, \qquad 2.2$$

where β represents the base (which is always assumed to be even), represents the exponent, and is the precision expressed as number of significant digits or bits forβ = 2. One of the most common FLP is the single precision IEEE 754-1985 format shown in figure 14.

The *IEEE* has standardized the computer representation for binary floating-point numbers in IEEE 754. IEEE 754-1985 is of the form:

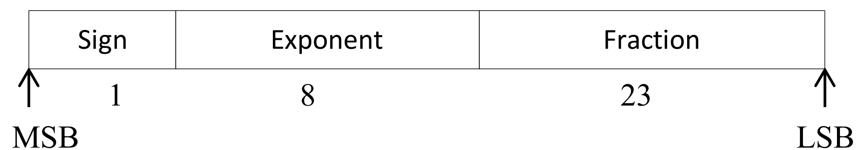


Figure 14.IEEE standard 754-1985 format for single precision.

Table 1. IEEE 754 binary formats

Type	Sign	Exponent	Significand	Total bits
Half	1	5	10	16
Single	1	8	23	32
Double	1	11	52	64
Double extended	1	15	64	80
Quad	1	15	112	128

2. *Fixed-point (FXP) Format:* Fixed-point format is also a representation for real data type. It is used for a number that has a fixed number of digits after (and sometimes also before) the radix point. FXP format is illustrated in figure 15. There are two parts in an FXP number. The first is the integer part, the second is the fractional part. FXP can be signed or unsigned. If we are using the signed fixed-point format, the first bit of the integer part represents the sign bit.

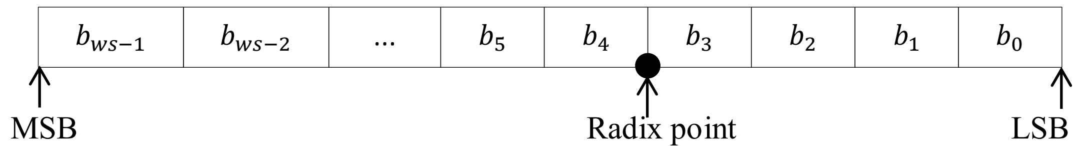


Figure 15.Format of an FXP format.

FLP has an advantage that it can support a much wider range of values for same number of bits when compared with the FXP format. The FXP architecture is always smaller in area, as compared to FLP architecture with similar precision. The FXP is also faster than its FLP counterpart.

2.3. General Structure

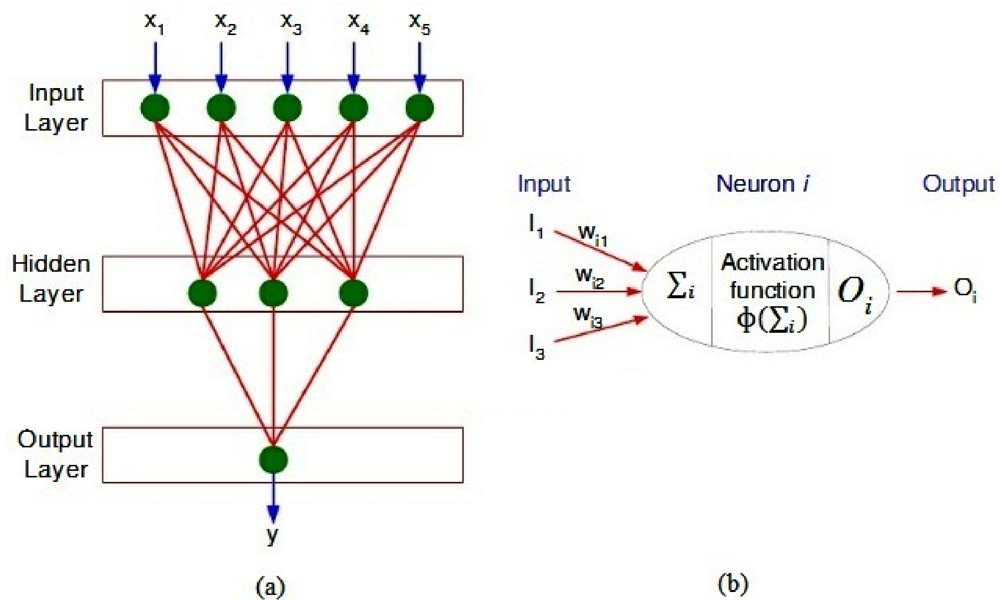


Figure 16.General structure of ANN.

ANN is an interconnected structure of neurons. It is a highly parallel structure. In figure 16(a), a 5:3:1 fully connected network is shown with 5 inputs viz. x1, x2, x3, x4, and x5

fed to the input layer, and a single output viz. y. In figure 16(b), the structure of a neuron, which is the basic block of the neural network, is shown.

For solving the XOR problem, a 2:2:1 fully connected topology is an optimum solution. The topology used for the present work is as shown in figure 17. The activation function used is implemented by a PWL (similar to hyperbolic tangent function).

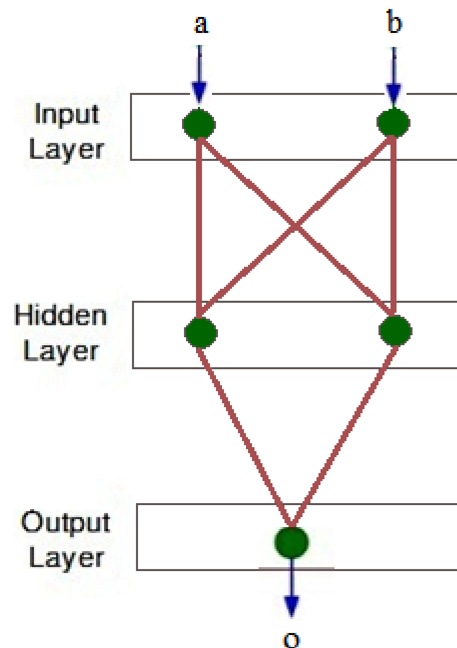


Figure 17. 2:2:1 topology used for solving XOR problem.

2.4. Squashing Function

The squashing function is an important component of an ANN. It bounds the output of a neuron. Squashing function is also termed as activation function. The squashing function important between the summed output of a neuron and the input of the next neuron because the output of a neuron may not be in the range acceptable as an input to the next neuron (i.e., out of bound input).

3. IMPLEMENTATION OF SQUASHING FUNCTION

3.1. Types of Squashing Function

The squashing function, also called activation function, denoted by $\varphi(v)$, defines the output of a neuron in terms of the induced local field v. Here we identify three basic types of activation functions:

a) ***Threshold Function*** For this type of activation function, described in Figure 18(a), we have

$$\phi(v) = \begin{cases} 1 \ \ if \ \ v > 0 \\ 0 \ \ if \ \ v \leq 0 \end{cases} \qquad 3.1$$

Such a neuron is referred to in the literature as the *McCulloch-Pitts model,* in recognition of the pioneering work done by McCulloch and Pitts (1943). In this model, the output of a neuron takes on the value of 1 if the induced local field of that neuron is non-negative and 0 otherwise. This statement describes the *all-or-none property* of the McCulloch-Pitts model.

b) ***Piecewise-Linear Function*** For the piecewise-linear function described in Figure 18(b)we have

$$\phi(v) = \begin{cases} 1 & if \quad v \geq +\frac{1}{2} \\ v & if \quad -\frac{1}{2} > v > +\frac{1}{2} \\ 0 & if \quad v \leq -\frac{1}{2} \end{cases} \qquad 3.2$$

where the amplification factor inside the linear region of operation is assumed to be unity. This form of an activation function may be viewed as an approximation to a non-linear amplifier.

The piecewise-linear function reduces to a threshold function if the amplification factor of the linear region is made infinitely large.

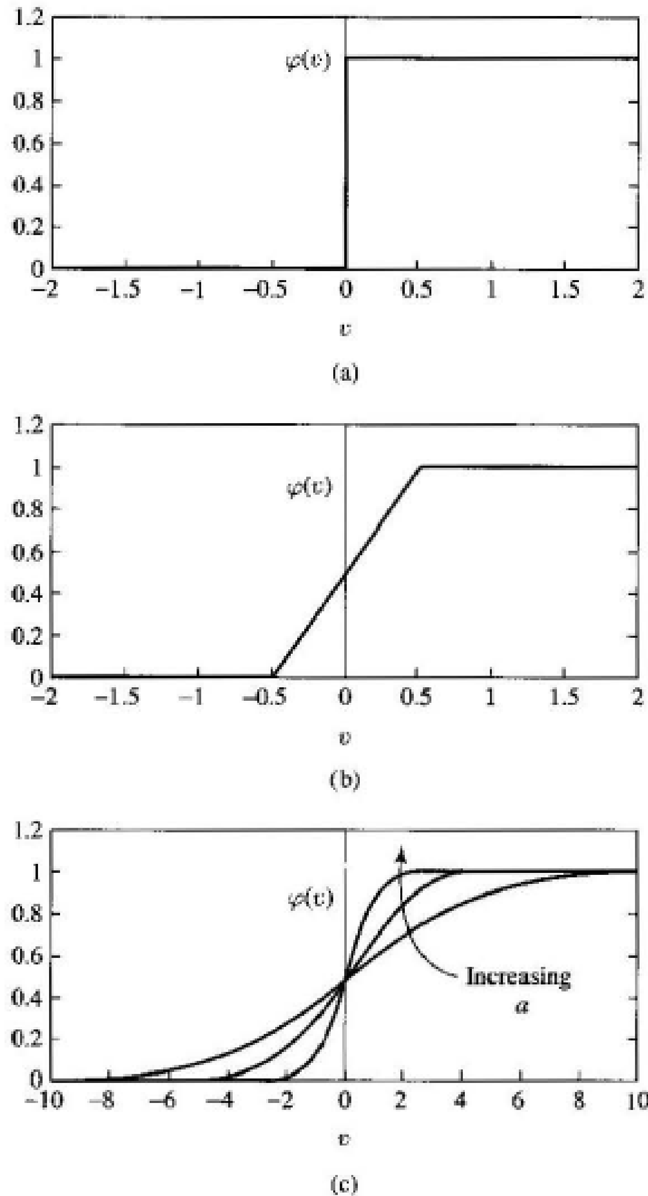


Figure 18.Types of activation functions (a) Threshold function.(b) Piecewise-linear function. (c) Sigmoid function for varying slope parameter *a* [3].

c) ***Sigmoid Function*** The sigmoid function, whose graph is s-shaped, is by far the most common form of activation function used in the construction of ANNs. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behaviour. An example of the sigmoid function is the logistic function, defined by

$$\phi(v) = \frac{1}{1+e^{(-av)}} \quad 3.3$$

where a is the slope parameter of the sigmoid function. By varying the parameter a, we obtain sigmoid functions of different slopes, as illustrated in Figure 18(c). In fact, the slope at the origin equals a/4. In the limit, as the slope parameter approaches infinity, the sigmoid function becomes simply a threshold function. Whereas a threshold function assumes the value of 0 or 1, a sigmoid function assumes a continuous range of values from 0 to l. Note also that the sigmoid function is differentiable, whereas the threshold function is not.

The activation functions defined above range from 0 to +l. It is sometimes desirable to have the activation function range from -1 to +1, in which case the activation function assumes an anti-symmetric form with respect to the origin; that is, the activation function is an odd function of the induced local field. Specifically, the threshold function of is now defined as

$$\phi(v) = \begin{cases} +1 & if \quad v > 0 \\ 0 & if \quad v = 0 \\ -1 & if \quad v < 0 \end{cases} \qquad 3.4$$

which is commonly referred to as the ***signum function***. For the corresponding form of a sigmoid function we may use the hyperbolic tangent function, defined by

$$\phi(v) = \tanh(v) \qquad 3.5$$

3.2. Piece-Wise Linear (PWL)

The implementation of a high-precision squashing function needs large area, but in FPGAs we have limited area. Thus for implementing in FPGA we need to find a trade-off between both the parameters. So we must implement the squashing function either using PWL (Piece-Wise Linear) or LUT (Look up table) [23]. Here we choose PWL because there is high precision loss using LUT. Also an LUT itself is a memory, it is thus undesirable when implementing an LUT based squashing function in FPGA, since FPGAs have limited internal memory which has other purposes also then serving only as a storage for the squashing function. Also sharing an LUT approximation for squashing function among all the neurons reduces speed.

The PWL function used is similar to the hyperbolic tangent function. Figure 19, shows the PWL implemented as squashing function. The curve is similar to the hyperbolic tangent curve. The non-linear curve is broken into eleven linear pieces for implementation as shown in figure 19.

The PWL function implemented as squashing function is:

$$o = \begin{cases} -1; & i < -8.0 \\ \frac{i+4.0}{64.0} - 0.9375; & i < -4.0 \\ \frac{i+2.0}{32.0} - 0.875; & i < -2.0 \\ \frac{i+1.0}{8.0} - 0.75; & i < -1.0 \\ \frac{i+0.5}{2.0} - 0.5; & i < -0.5 \\ i; & i < +0.5 \\ \frac{i-0.5}{2.0} + 0.5; & i < +1.0 \\ \frac{i-1.0}{8.0} + 0.75; & i < +2.0 \\ \frac{i-2.0}{32.0} + 0.875; & i < +4.0 \\ \frac{i-4.0}{64.0} + 0.9375; & i < +8.0 \\ 1; & i > +8.0 \end{cases} \quad 3.6$$

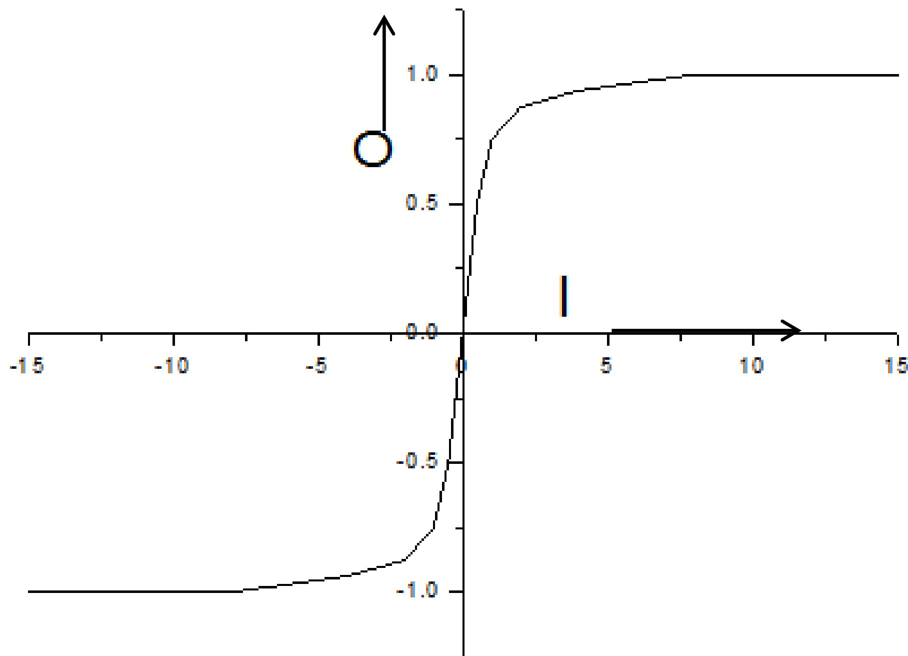


Figure 19.PWL function implemented.

The differential of the squashing function is also needed to evaluate the adjustment in the weights. The symbol generated by Xilinx ISE for the PWL function and its differential are shown in figure 20.

The differential of the PWL is:

$$o = \begin{cases} 0.00000; & i < -8.0 \\ 0.01563; & i < -4.0 \\ 0.03125; & i < -2.0 \\ 0.12500; & i < -1.0 \\ 0.50000; & i < -0.5 \\ 1.00000; & i < +0.5 \\ 0.50000; & i < +1.0 \\ 0.12500; & i < +2.0 \\ 0.03125; & i < +4.0 \\ 0.01563; & i < +8.0 \\ 0.00000; & i > +8.0 \end{cases} \quad 3.7$$

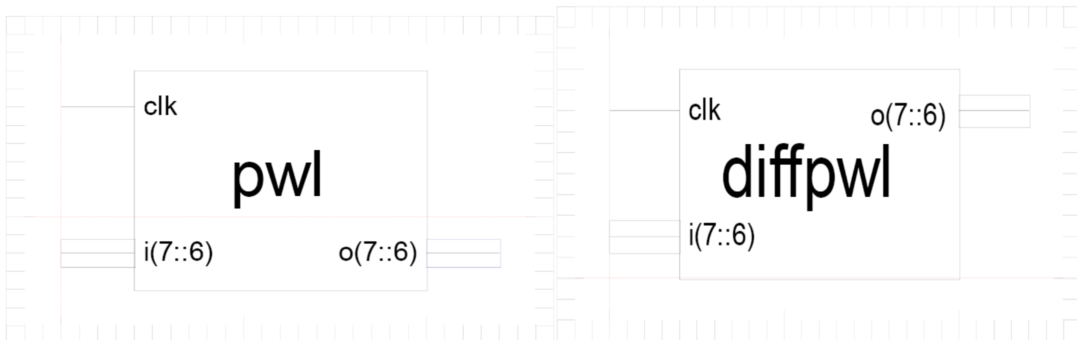


Figure 20.Symbol generated for PWL and its differential.

4. IMPLEMENTATION OF MAIN NEURAL BLOCK

In the present work, I’ve suggested a 2:2:1 network using signed fixed numeric representation to fulfil the requirement of using real number (which is not synthesizable for hardware implementation). The symbol for main neural block is shown in figure 21. This is the hierarchical top module of the design. It consists of pwl and diffpwl as sub-modules. Forward stage, Backpropagation stage and weight update stage, all are integrated in this top module.

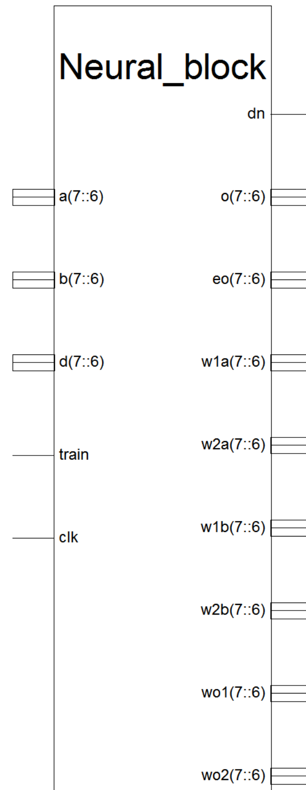


Figure 21. Symbol generated for the main neural block.

The main neural block has *a, b, d* as 14-bit signed fixed (7::-6) inputs (input layer); *train* as single bit input for initiating training; *clk* as a single bit input clock; *o* as 14-bit signed fixed (7::-6) output; *eo* as 14-bit signed fixed (7::-6) error in output compared to the desired output given by *d*; *dn* as I single bit output to denote that one iteration of back-propagating the error is complete; six weights *w1a,w2a, w1b, w2b, wo1, wo* as 14-bit signed fixed (7::-6) inout pins.

The main neural block instantiate three pwl, and three diffpwl modules for each of the hidden and output neurons. The architecture of the block is as shown in figure 22. In the figure, a and b are the inputs, n_a, and n_b represents the input neurons, n_{h1}, n_{h2}, and n_o represents the hidden and output neurons respectively; w_{1a}, w_{1b}, w_{2a}, w_{2b}, w_{o1}, and w_{o2}represents weights connecting neurons (w_{ij} is the weight for the signal path from neuron j to neuron i. The output at neuron n_{h1} is given by $\phi(a \times w_{1a} + b \times w_{1b})$, where $\phi(.)$ is the squashing function implemented as pwl. Similarly outputs of all the neurons can be calculated.

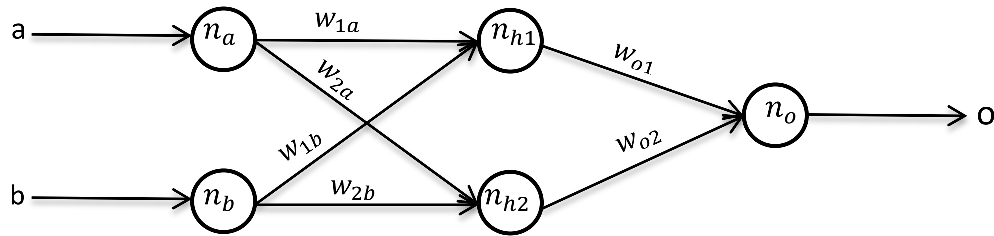


Figure 22. Architecture of the network used.

The final output *o,* thus is evaluated as:

$$o_{h1} = \phi(a \times w_{1a} + b \times w_{1b}) \quad 4.1$$

$$o_{h2} = \phi(a \times w_{2a} + b \times w_{2b}) \quad 4.2$$

$$O = \phi(o_{h1} \times w_{o1} + o_{h2} \times w_{o2}) \quad 4.3$$

where o_{h1}, and o_{h2} are the outputs of the hidden neurons n_{h1}, and n_{h2} respectively.

The error in the final output is thus calculated as:

$$e_o = d - o \quad 4.4$$

Next we find the delta at the output node as

$$\delta_o = e_o \times \phi^{'}(o), \quad 4.5$$

where $\phi^{'}(o)$ is the differential of the final output.

Now, the deltas at the hidden nodes can be calculated as:

$$\delta_{h1} = \delta_o \times w_{o1} \times \phi^{'}(o_{h1}), \quad 4.6$$

$$\delta_{h2} = \delta_o \times w_{o2} \times \phi^{'}(o_{h2}). \quad 4.7$$

Next we have to find the adjustments to be made to weights:

$$\Delta w_{o1} = \eta \times \delta_o \times o_{h1}, \quad 4.8$$

$$\Delta w_{o2} = \eta \times \delta_o \times o_{h2}, \quad 4.9$$

$$\Delta w_{1a} = \eta \times \delta_{h1} \times a, \quad 4.10$$

$$\Delta w_{2a} = \eta \times \delta_{h2} \times a, \quad 4.11$$

$$\Delta w_{1b} = \eta \times \delta_{h1} \times b, \quad 4.12$$

$$\Delta w_{2b} = \eta \times \delta_{h2} \times b, \quad 4.13$$

The adjustments evaluated above are added to the original weights in the next stage (weight updation stage), the updated weights are:

$$w'_{1a} = w_{1a} + \Delta w_{1a}, \quad 4.14$$

$$w'_{1b} = w_{1b} + \Delta w_{1b}, \quad 4.15$$

$$w'_{2a} = w_{2a} + \Delta w_{2a}, \quad 4.16$$

$$w'_{2b} = w_{2b} + \Delta w_{2b}, \quad 4.17$$

$$w'_{o1} = w_{o1} + \Delta w_{o1}, \quad 4.18$$

$$w'_{o2} = w_{o2} + \Delta w_{o2}. \quad 4.19$$

After updating the weights, the *dn* bit goes high. Again the error in the output is calculated. If this error is greater than 0.03125, the block continues training with new set of inputs, otherwise the network is trained with the data set.

5. RESULTS AND DISCUSSIONS

This chapter is divided into three parts: *functional simulation results*, *FPGA synthesis results*, and *ASIC synthesis results*. The first part summarizes the results obtained during functional simulation of the main neural block done with ModelSim. Then follows the FPGA synthesis results using Xilinx XST and the synthesis results of ASIC system contributes the last part.

5.1. Functional Simulation

The design is divided into three modules. They are:

1) PWL, the activation function
2) DIFFPWL, differential of the activation function (required to calculate the error)
3) NEURAL_BLOCK_1, the main neural computational block (2:2:1 network).

NEURAL_BLOCK_1 is the top module in the design.

The PWL implemented is shown in figure 23.

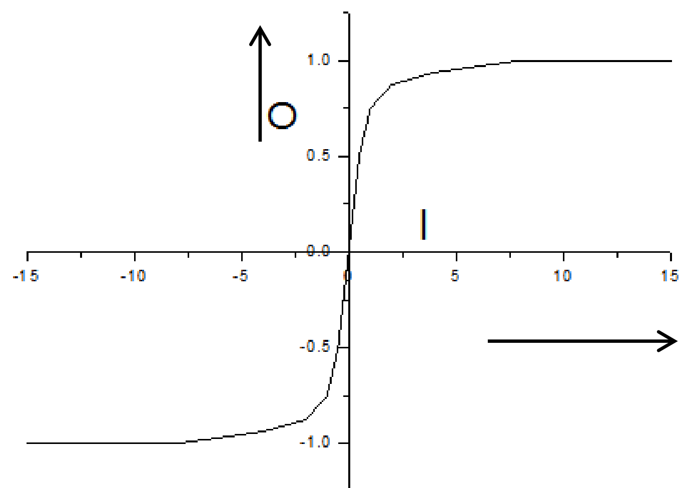


Figure 23.PWL implementation for non-linear activation function.

Simulation results for the PWL, DIFFPWL, and NEURAL_BLOCK_1 modules are shown in Figures 24, 25, and 26 respectively.

Figure 24. Simulation result for the PWL module

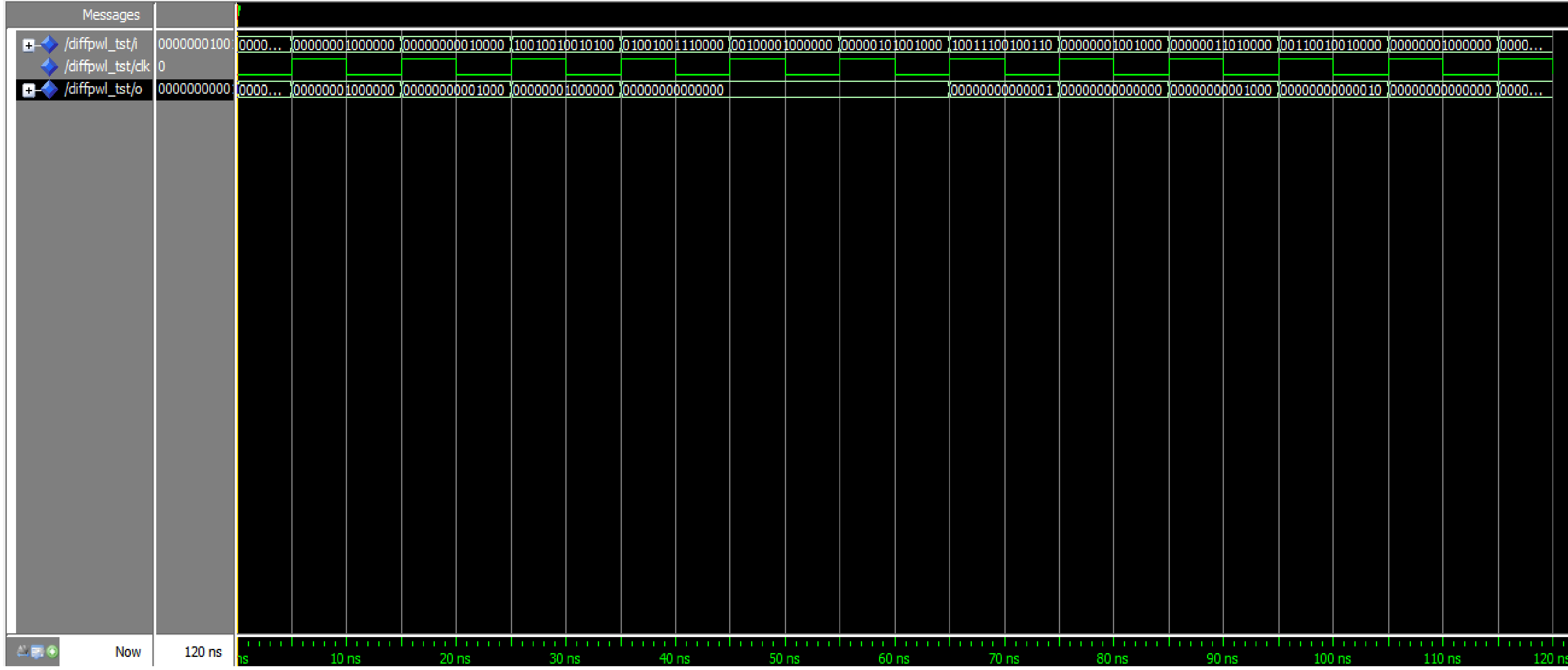


Figure 25. Simulation result for the DIFFPWL module

Figure 26 (a). Simulation result for the NEURAL_BLOCK_1 module using testbench

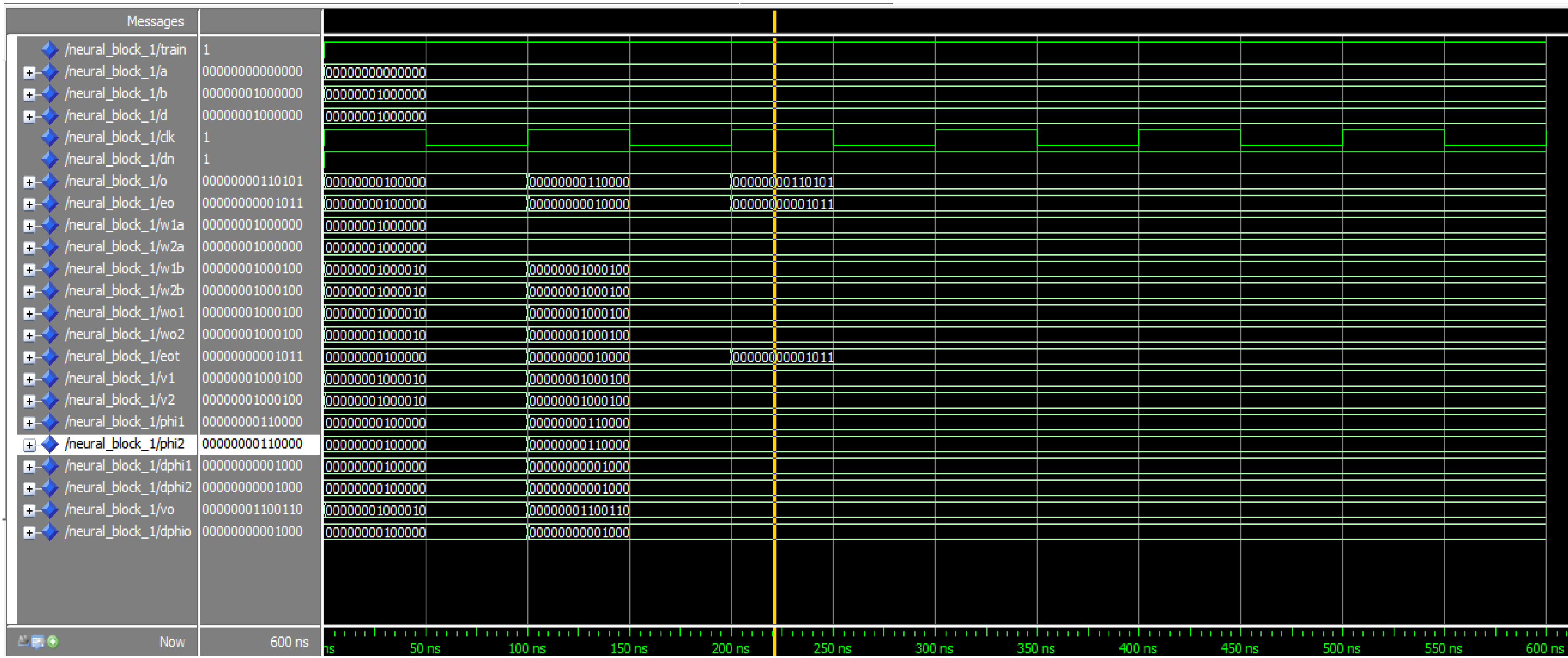


Figure 26(b). Simulation result for the NEURAL_BLOCK_1 module

5.2. FPGA Implementation

FPGA implementation is done using Xilinx ISE 13.4 for the thesis work. The implementation step is divided in following steps:

5.2.1. Synthesis:

During synthesis, the HDL files are translated into gates and optimized for the target architecture. Here the VHDL code is synthesized for Xilinx Spartan-3E starter kit using Xilinx ISE 13.4. The Xilinx Synthesis Tool (XST) uses the design's HDL code and generates a supported netlist (NGC) for the Xilinx implementation tools.

Processes available for synthesis using XST are as follows:

a) *View RTL Schematic*
   Generates a schematic view of the RTL netlist. Pre-optimization of the HDL code.
b) *View Technology Schematic*
   Generates a schematic view of the technology netlist. Post-synthesis view of the HDL design mapped to the target technology.
c) *Check Syntax*
   Verifies that the HDL code is entered properly.
d) *Generate Post-Synthesis Simulation Model*
   Creates HDL simulation models based on the synthesis netlist.

Figure 27. shows each of the steps that take place during XST synthesis.

XST generates NGR from the register transfer level (RTL) netlist. RTL Viewer opens the NGR file, and you can select a block to view as a schematic. The RTL Viewer does not generate output files. It only allows you to view, not save, NGR files. XST also generates an NGC file, which is the netlist file with constraint information.

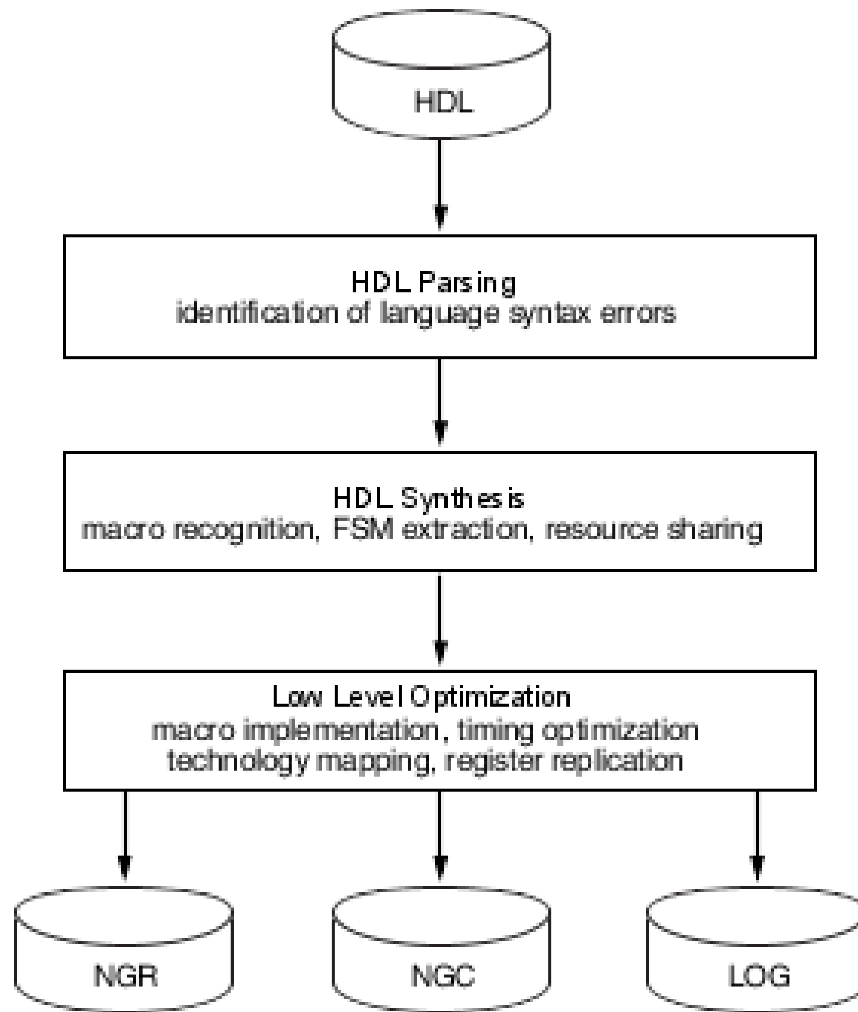


Figure 27. XST Design Flow.

Following section describe each step in detail with results for the design.

1) HDL Parsing

During HDL parsing, XST checks whether your HDL code is correct and reports any syntax errors. During this step, the XST first compiles each of the design files in the specified libraries followed by building design hierarchy. And finally analyses the design files. Analysis report of the design files:

```
==========================================================================
*                             HDL Analysis                                *
==========================================================================
Analyzing Entity <Neural_block_1> in library <work> (Architecture
<behavioral>).
WARNING:Xst:819 - "E:/my_designs/thesis/xilinxIse13.1/BPN1/nn1.vhd" line
173: One or more signals are missing in the process sensitivity list. To
enable synthesis of FPGA/CPLD hardware, XST will assume that all
necessary signals are present in the sensitivity list. Please note that
the result of the synthesis may differ from the initial design
specification. The missing signals are:
   <eot>, <dphio>, <phi1>, <phi2>, <wo1>, <wo2>, <dphi1>, <dphi2>, <w1a>,
<w1b>, <w2a>, <w2b>
Entity <Neural_block_1> analyzed. Unit <Neural_block_1> generated.

Analyzing Entity <pwl> in library <work> (Architecture <beh>).
Entity <pwl> analyzed. Unit <pwl> generated.

Analyzing Entity <diffpwl> in library <work> (Architecture <beh>).
Entity <diffpwl> analyzed. Unit <diffpwl> generated.
```

Figure 28. HDL analysis report.

The warning Xst: 819, occur if an input signal of a process block is not listed in the sensitivity list of that block.

2) HDL Synthesis

During HDL synthesis, XST analyses the HDL code and attempts to infer specific design building blocks or macros (such as MUXes, RAMs, adders, and subtractors) for which it can create efficient technology implementations. To reduce the amount of inferred macros, XST performs a resource sharing check. This usually leads to a reduction of the area.

Table 2. Inferred blocks for each design unit

Design Unit	Inferred Blocks
PWL	10 Adders/Subtractors 10 Comparators
DIFFPWL	10 Comparators
Neural_block_1	22 Adder/Subtractors. 20 Multipliers

Figure 29, shows the synthesis report showing total number of design building blocks required after HDL synthesis.

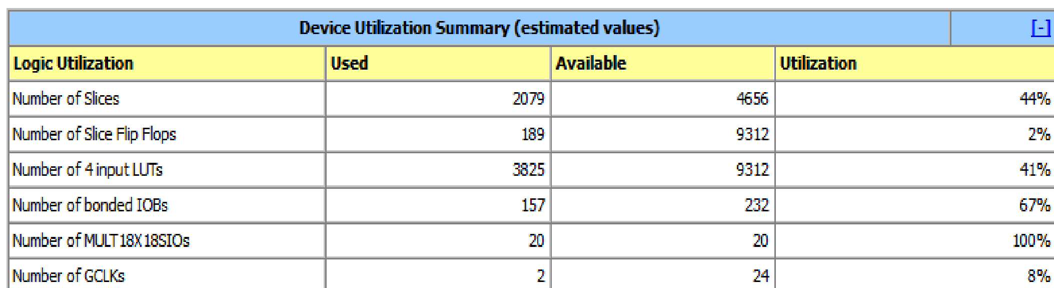
```
=========================================================================
HDL Synthesis Report

Macro Statistics
# Multipliers                                          : 20
 14x14-bit multiplier                                  : 12
 28x14-bit multiplier                                  : 8
# Adders/Subtractors                                   : 52
 15-bit adder                                          : 33
 15-bit subtractor                                     : 13
 29-bit adder                                          : 3
 30-bit addsub                                         : 3
# Latches                                              : 13
 1-bit latch                                           : 1
 14-bit latch                                          : 9
 30-bit latch                                          : 3
# Comparators                                          : 60
 14-bit comparator less                                : 60
# Xors                                                 : 15
 1-bit xor2                                            : 15

=========================================================================
```

Figure 29. HDL synthesis report obtained from XST showing the total number of design building blocks required after HDL synthesis.

3) Low Level Optimization

During low level optimization, XST transforms inferred macros and general glue logic into a technology-specific implementation. Also the redundant blocks are trimmed.

Device utilization summary of the design for the selected device is shown in figure 30. These are the estimated values during synthesis. The actual values are available after mapping the design to the target FPGA.



Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	2079	4656	44%
Number of Slice Flip Flops	189	9312	2%
Number of 4 input LUTs	3825	9312	41%
Number of bonded IOBs	157	232	67%
Number of MULT18X18SIOs	20	20	100%
Number of GCLKs	2	24	8%

Figure 30. Device utilization summary.

The final report after low level optimization is shown in figure 31.

```
==========================================================================
*                              Final Report                              *
==========================================================================
Final Results
RTL Top Level Output File Name     : Neural_block_1.ngr
Top Level Output File Name         : Neural_block_1
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : No

Design Statistics
# IOs                              : 157

Cell Usage :
# BELS                             : 7888
#      GND                         : 1
#      INV                         : 62
#      LUT1                        : 101
#      LUT2                        : 988
#      LUT2_D                      : 12
#      LUT2_L                      : 22
#      LUT3                        : 561
#      LUT3_D                      : 79
#      LUT3_L                      : 65
#      LUT4                        : 1704
#      LUT4_D                      : 142
#      LUT4_L                      : 89
#      MULT_AND                    : 252
#      MUXCY                       : 1829
#      MUXF5                       : 55
#      VCC                         : 1
#      XORCY                       : 1925
# FlipFlops/Latches                : 189
#      LD                          : 105
#      LDE                         : 84
# Clock Buffers                    : 2
#      BUFG                        : 1
#      BUFGP                       : 1
# IO Buffers                       : 156
#      IBUF                        : 43
#      OBUF                        : 113
# MULTs                            : 20
#      MULT18X18SIO                : 20
==========================================================================
```

Figure 31. Report after low level optimization

5.2.2. Translation

Translation is the first step of the back end design implementation. ISE uses NGDBuild tool during translation. NGDBuild takes the synthesized netlist (NGC) (from the front end tool – XST) and constraints files as inputs and creates a Xilinx® Native Generic Database (NGD) file that contains a logical description of the design in terms of logic elements, such as AND gates, OR gates, LUTs, flip-flops, and RAMs. It also creates a BLD file which is build report file contains information about the NGDBuild run.

Figure 32. shows the NGDBuild design flow.

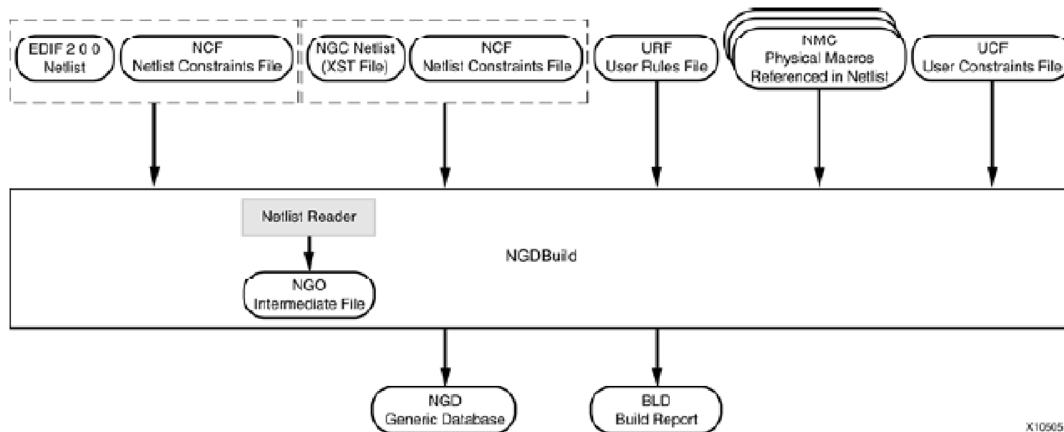


Figure 32. NGDBuild design flow.

The NGD file contains both a logical description of the design reduced to Xilinx primitives and a description of the original hierarchy expressed in the input netlist. The output NGD file can be mapped to the desired device family.

Figure 33. shows the translation report.

```
Release 13.4 ngdbuild O.87xd (nt)
Copyright (c) 1995-2011 Xilinx, Inc.  All rights reserved.

Command Line: C:\Xilinx\13.4\ISE_DS\ISE\bin\nt\unwrapped\ngdbuild.exe -intstyle
ise -dd _ngo -nt timestamp -uc Neural_block.ucf -p xc3s500e-fg320-4
Neural_block_1.ngc Neural_block_1.ngd

Reading NGO file "E:/my_designs/thesis/xilinxIse13.1/BPN1/Neural_block_1.ngc"
...
Gathering constraint information from source properties...
Done.

Annotating constraints to design from ucf file "Neural_block.ucf" ...
Resolving constraint associations...
Checking Constraint Associations...
Done...

Checking expanded design ...

Partition Implementation Status
-------------------------------

  No Partitions were found in this design.

-------------------------------

NGDBUILD Design Results Summary:
  Number of errors:     0
  Number of warnings:   0

Total memory usage is 120032 kilobytes

Writing NGD file "Neural_block_1.ngd" ...
Total REAL time to NGDBUILD completion:  14 sec
Total CPU time to NGDBUILD completion:   9 sec

Writing NGDBUILD log file "Neural_block_1.bld"...
```

Figure 33. Translation report of the design.

5.2.3. MAP

The MAP program maps a logical design to a Xilinx® FPGA. The input to MAP is an NGD file, generated by the NGDBuild program. Depending on the options used, MAP places the design.

MAP first performs a logical DRC (Design Rule Check) on the design in the NGD file. MAP then maps the design logic to the components (logic cells, I/O cells, and other components) in the target Xilinx FPGA.

The output from MAP is an NCD (Native Circuit Description) file a physical representation of the design mapped to the components in the targeted Xilinx FPGA. The mapped NCD file can then be placed and routed using the PAR program.

Figure 34. shows the MAP design flow.

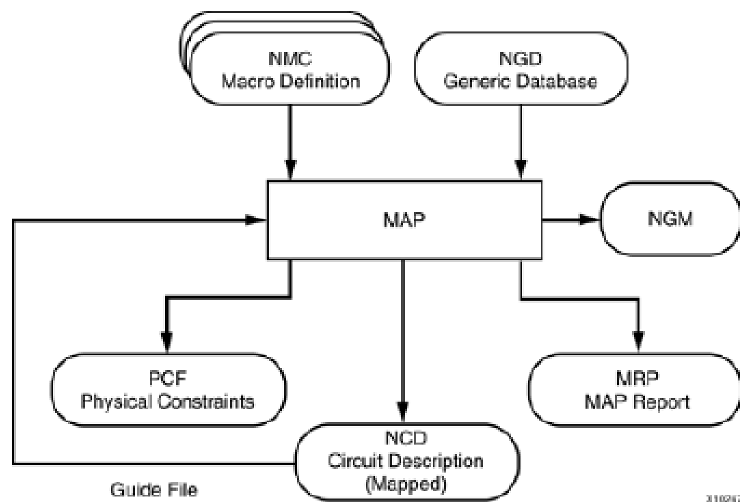


Figure 34. MAP design flow.

Figure 35. shows the device utilization summary post-MAP. In the report, related logic is defined as being logic that shares connectivity - e.g. two LUTs are "related" if they share common inputs. When assembling slices, Map gives priority to combine logic that is related. Doing so results in the best timing performance.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Latches	189	9,312	2%	
Number of 4 input LUTs	3,826	9,312	41%	
Number of occupied Slices	2,054	4,656	44%	
Number of Slices containing only related logic	2,054	2,054	100%	
Number of Slices containing unrelated logic	0	2,054	0%	
Total Number of 4 input LUTs	3,950	9,312	42%	
Number used as logic	3,826			
Number used as a route-thru	124			
Number of bonded IOBs	157	232	67%	
Number of BUFGMUXs	2	24	8%	
Number of MULT18X18SIOs	20	20	100%	
Average Fanout of Non-Clock Nets	2.49			

Figure 35. Device utilization summary after mapping the design to the target FPGA.

Unrelated logic shares no connectivity. Map will only begin packing unrelated logic into a slice once 99% of the slices are occupied through related logic packing.

Note that once logic distribution reaches the 99% level through related logic packing, this does not mean the device is completely utilized. Unrelated logic packing will then begin, continuing until all usable LUTs and FFs are occupied. Depending on your timing budget, increased levels of unrelated logic packing may adversely affect the overall timing performance of your design.

5.2.4. PAR (Place and Route)

After creating the Native Circuit Description (NCD) file with the MAP program, placement and routing the design file using PAR can be done. PAR accepts a mapped NCD file as input, places and routes the design, and outputs an NCD file to be used by the bitstream generator (BitGen).

PAR is done in following two steps:

- *Placing:* The PAR placer executes multiple phases of the placer. PAR writes the NCD after all the placer phases are complete.

  During placement, PAR places components into sites based on factors such as constraints specified in the PCF file, the length of connections, and the available routing resources.

- *Routing:* After placing the design, PAR executes multiple phases of the router. The router performs a converging procedure for a solution that routes the design to completion and meets timing constraints. Once the design is fully routed, PAR writes an NCD file, which can be analysed against timing.

  PAR writes a new NCD as the routing improves throughout the router phases.

Figure 36. shows PAR flow.

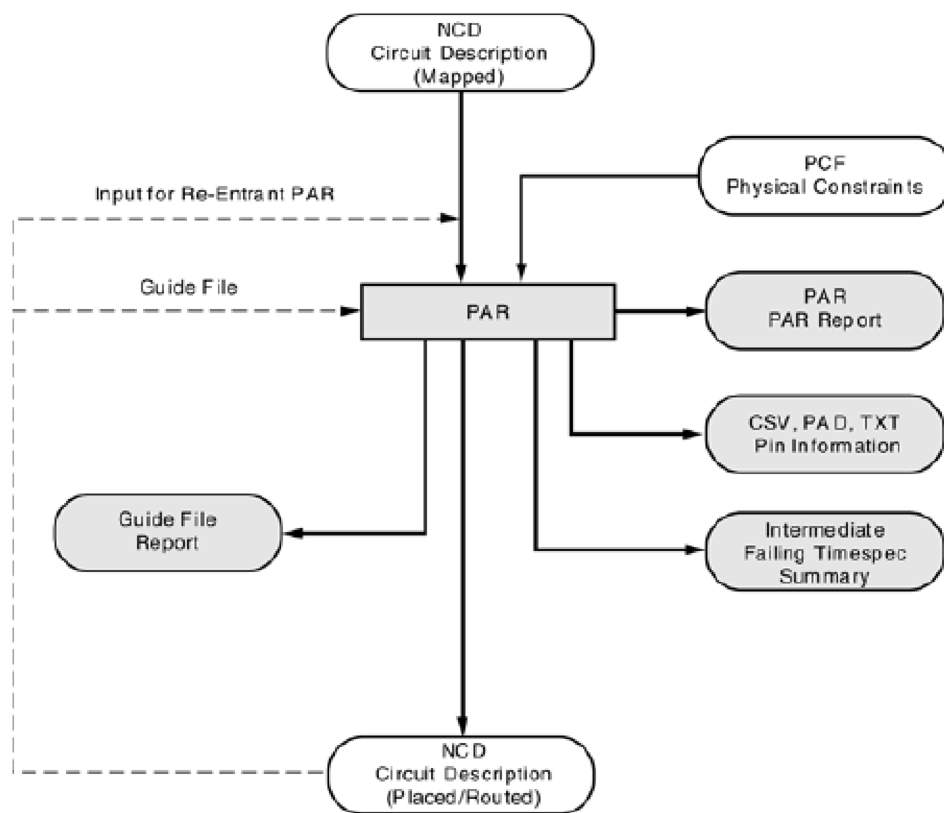


Figure 36. PAR flow.

PAR is done with successfully without errors.

5.2.5. STA (Static Timing Analysis)

Static timing analysis is an important step in analysing the performance of a design. Generally, static timing analysis is much faster than timing-driven gate-level simulation and does not require stimulus vector generation. Therefore, unlike dynamic analysis, the quality of the static approach is independent of the quality of stimulus vectors. However, proper functionality of the design cannot be checked in static analysis.

An accurate and efficient static timing analysis has many benefits, such as providing quick and efficient information to enhance the design performance and easing the design debugging procedure. This application note presents the most important concepts and techniques of static timing analysis and contains practical examples.

In FPGA flow, STA is done at two steps:

1) *Post-Map STA:* The timing report generated after mapping uses the estimated delay information. Accurate timing report can be obtained once the PAR is done. It is also referred to as pre-route STA.
2) *Post-Place and Route STA:* The actual timing report generated after routing is done. This gives the actual timing report .

Table 3. STA Results

	Pre-route STA	Post-route STA
Setup slack (minimum)	72.303 ns	44.145 ns
Hold slack (minimum)	1.442 ns	1.724 ns
Component switching limits slack (minimum)	148.404 ns	148.348 ns
Minimum Period	77.697 ns	105.855 ns
Maximum operation frequency	12.871 MHz	9.447 MHz

The slack associated with each connection is computed as the difference between the required arrival time (RAT) and the actual arrival time (AAT). Positive slack indicates that timing is met – the signal arrives before it is required – while negative slack indicates that timing is violated – the signal arrives after its required time.

$$\textbf{Setup slack} = (\textbf{requirement} - (\textbf{data path} - \textbf{clock path skew} + \textbf{uncertainty}))$$

$$\textbf{Hold slack} = (\textbf{requirement} - (\textbf{clock path skew} + \textbf{uncertainty} - \textbf{data path}))$$

5.2.6. Power analysis

Xilinx ISE provides XPower tool for power analysis. XPower provides power and thermal estimates after PAR, for FPGA designs. XPower does the following:

- Estimates how much power the design will use
- Identifies how much power each net or logic element in the design is using
- Verifies that junction temperature limits are not exceeded.

Hierarchical division of power among different modules is as shown in table 4. and figure 37. shows the XPower results.

Table 4. Hierarchical division of power among different modules.

Name	Power	Logic Power	Signal Power	#FFs	#LUTs	#MULTs
Hierarchical Total	0.00285	0.00136	0.00149	189	3724	20
Neural_block_1	0.00202	0.00096	0.00105	84	2693	20
Inst_diffpwl1	0.00007	0.00001	0.00006	5	32	0
Inst_diffpwl2	0.00008	0.00001	0.00007	5	32	0
Inst_diffpwl3	0	0	0	5	40	0
Ist_pwl1	0.00031	0.00017	0.00014	30	300	0
Ist_pwl2	0.00031	0.00017	0.00014	30	299	0
Ist_pwl3	0.00007	0.00004	0.00003	30	328	0

Device	
Family	Spartan3e
Part	xc3s500e
Package	fg320
Temp Grade	Commercial
Process	Typical
Speed Grade	-4

Environment	
Ambient Temp (C)	25.0
Use custom TJA?	No
Custom TJA (C/W)	NA
Airflow (LFM)	0

Characterization	
PRODUCTION	v1.2,06-23-09

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.001	2	---	---
Logic	0.001	3848	9312	41
Signals	0.001	5542	---	---
MULTs	0.000	20	20	100
IOs	0.009	157	232	68
Leakage	0.081			
Total	0.094			

Thermal Properties	Effective TJA (C/W)	Max Ambient (C)	Junction Temp (C)
	26.1	82.6	27.4

Supply Summary		Total	Dynamic	Quiescent
Source	Voltage	Current (A)	Current (A)	Current (A)
Vccint	1.200	0.029	0.003	0.026
Vccaux	2.500	0.018	0.000	0.018
Vcco25	2.500	0.005	0.003	0.002
		Total	Dynamic	Quiescent
Supply Power (W)		0.094	0.013	0.081

Figure 37. XPower results summary

5.3. ASIC Implementation

Cadence tools are used for ASIC implementation. We have used Encounter RTL Compiler for synthesizing the design and First Encounter for back end implementations (Placement and Routing).

5.3.1. Synthesis

Cadence Encounter RTL Compiler (RC) is used for synthesising the HDL code to netlist. RTL Compiler takes the HDL code (Verilog/VHDL), design constraints, and target library as inputs, and creates an optimized netlist (Verilog), and design constraints (for back end tools).

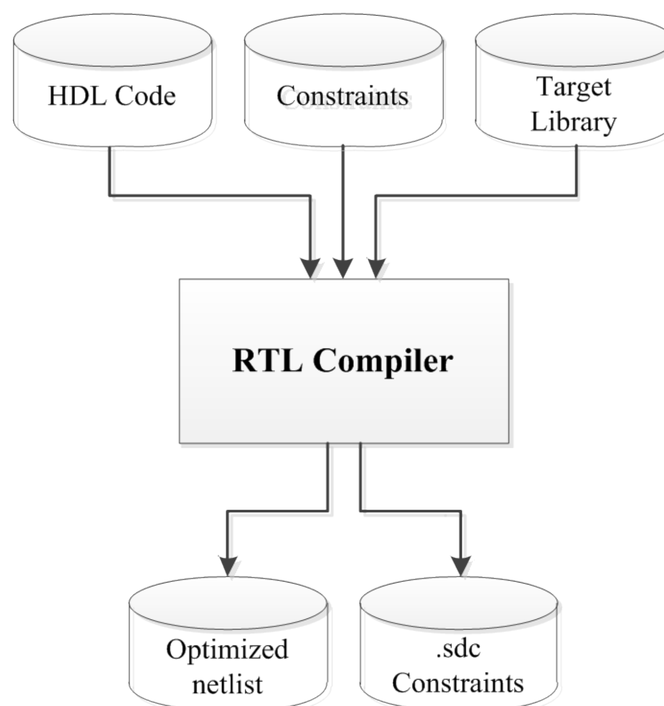


Figure 38.Input and output files for RTL Compiler.

The work flow for RTL Compiler is shown in figure 38. Each step is explained below:

5.3.1.1. Reading in the Design

Before giving the inputs files to the RTL Compile, we have to specify the search path for libraries, and Design files (HDL Code) using the commands:

rc:/> set_attribute lib_search_path *path* /

rc:/> set_attribute hdl_search_path *path* /

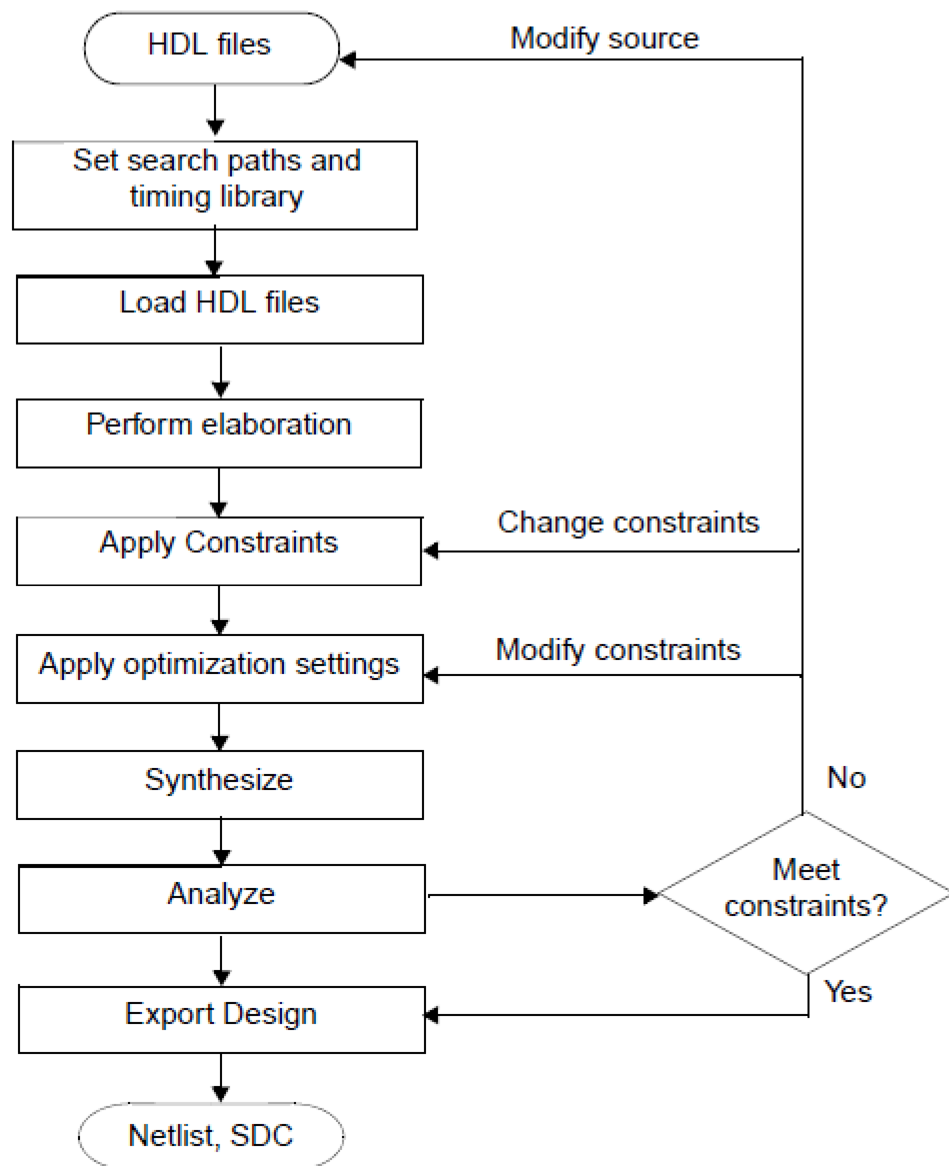


Figure 39. RTL Compiler work flow.

Now we have to specify the target library using command:

$$\text{rc:/>}\ \ \text{set_attribute library}\ lib_name.lib$$

The next step is loading HDL files using the command:

$$\text{read_hdl}\ \{\ file1.v\ file2.v\ file3.v\ \}$$

The above command loads Verilog files by default. To load VHDL files we have to use –vhdl switch. For loading files into desired library, first we have to create the library and load the design files to that library using –library switch.

In the design, we needed to load the package files to a new library named IEEE_proposed, and compile the package files in the new library. The commands used to create library IEEE_proposed and load fixed package into IEEE_proposed are:

hdl_create library IEEE_proposed

read_hdl -vhdl -library IEEE_proposed {fixed_float_types_c.vhdl fixed_pkg_c.vhdl}

The design files pwl.vhd, diffpwl.vhd, nn.vhd are read using:

read_hdl -vhdl {pwl1.vhd diffpwl.vhd nn.vhd}

5.3.1.2. Elaborating the design

Elaboration translates the design into a technology-independent design. Elaboration is only required for the top-level design. The elaborate command automatically elaborates the top-level design and all of its references. During elaboration, RTL Compiler performs the following tasks:

- Builds data structures
- Infers registers in the design
- Performs high-level HDL optimization, such as dead code removal
- Checks semantics

After elaboration, RTL Compiler has an internally created data structure for the whole design so now we can apply constraints and perform other operations.

During elaboration, RTL Compiler removed unused registers from the design.

5.3.1.3. Constraining the Design

After loading and elaborating your design, constraints must be applied to the design.

The constraints include:

- Operating conditions
- I/O timing
- Clock waveforms

5.3.1.4. Synthesizing the Design

Synthesis is the process of transforming the HDL design into a gate-level netlist, given all the specified constraints. In RTL Compiler, synthesis involves the following four processes:

- *RTL Optimization:* During RTL optimization, RTL Compiler performs optimizations like datapath synthesis, resource sharing, speculation, mux optimization, and carry save arithmetic (CSA) optimizations. After this step, RTL Compiler performs logic optimizations like structuring and redundancy removal.

- *Global Focus Mapping:* RTL Compiler performs global focus mapping at the end of the RTL technology-independent optimizations.
  This step includes restructuring and mapping the design concurrently, including optimizations like splitting, pin swapping, buffering, pattern matching, and isolation.

- *Remapping:* After Global Focus Mapping, RTL Compiler performs synthesis remapping. During this phase, RTL Compiler only performs global sizing of cells. There are actually multiple remapping phases: some are targeted at area optimization while others at timing optimization.

- *Incremental optimization:* The final optimization RTL Compiler performs is incremental optimization. Optimizations performed during IOPT improve timing and area and fix DRC violations.

Synthesis is performed in three steps:

- Synthesizing the design to generic logic (RTL optimizations are performed in this step).

- Mapping to the technology library.

- Performing incremental optimizations.

5.3.1.5. Export design

After completing synthesis, the gate level netlist and constraint file needed for back end tools are created using commands:

$$\text{write_hdl} > filename.v$$

$$\text{write_sdc} > filename.sdc$$

Synthesis Results

The results of the synthesized design are summarized in table 5. The negative slack creates timing violations. To remove the negative slack, we have partitioned large blocks and inserted latches in the critical path to reduce the critical path delay thus reducing the actual arrival time (because slack is the difference between the Required Arrival Time (RAT) and Actual Arrival Time(AAT)).

Table 5. Synthesis results for the design.

	Generic	Mapped	Incremental
Total Power (nW)	5811060.238	3925052.064	3700373.48
Leakage Power (nW)	400607.899	593615.443	590776.664
Dynamic Power (nW)	5410452.463	3331436.622	3309596.81
Area	NA	106609.104	106216.085
Timing Slack (ps)	-13364	-2556	1083
Number of cells	48882	20136	19940

5.3.2. Placement and Routing

We used Cadence First Encounter back end tool for placement and routing. The routed netlist can be exported to GDSII stream.

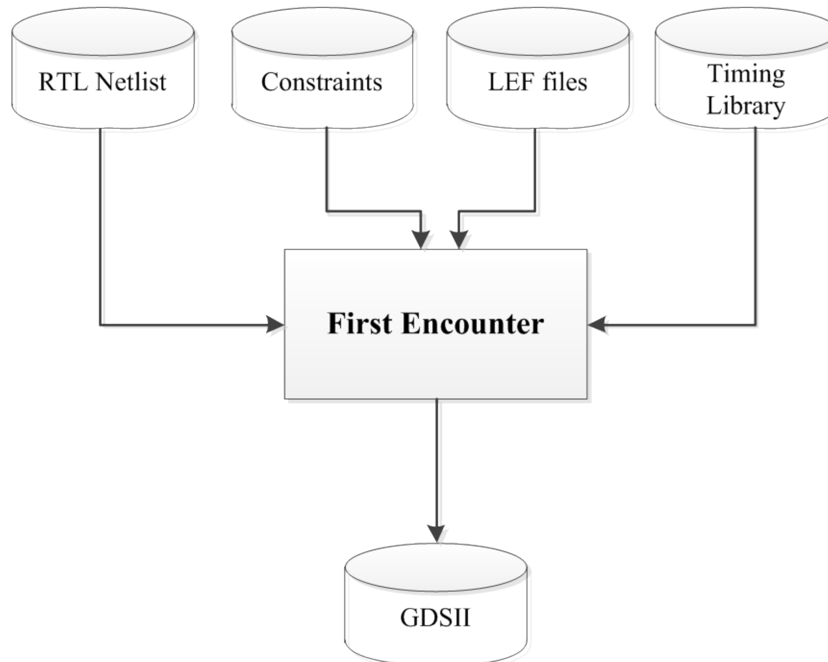


Figure 40.Input and output files for First Encounter.

First we have to read all the input files. Then the design is to be floorplanned. Floorplanning is the first point for physical layout. Floorplanning is a step in design that gives the designer some control over the chip. It allows the user to set specific sizes of the core and move around chip objects. The next step is Power planning. In this step, Power Rings and Power Stripes are added to the chip for supplying VDD and VSS. Once the Power planning is done, the design can be placed. During placement, Encounter attempts to accommodate the Floorplan given for the design. It uses the hierarchy and connectivity along with the other constraints given and attempts to automatically place the standard cells. The next step is to perform STA in the placed design to check for timing violations. The next step is Clock Tree Synthesis (CTS), which is to add clock trees to the design. Before CTS, the clock is ideal, so during timing analysis setup violations are checked, and Post CTS, hold violations are checked. Followed by CTS, timing analysis is to be done. If there are no timing violations in the design, the design is then routed. The design is then again analysed for timing violations. RC extraction is to

be done once post-route timing analysis is done in the design. The last step is to verify the design for any error (geometry, connectivity, metal density) and export file.

Figure 41. shows the generic flow for Cadence First Encounter.

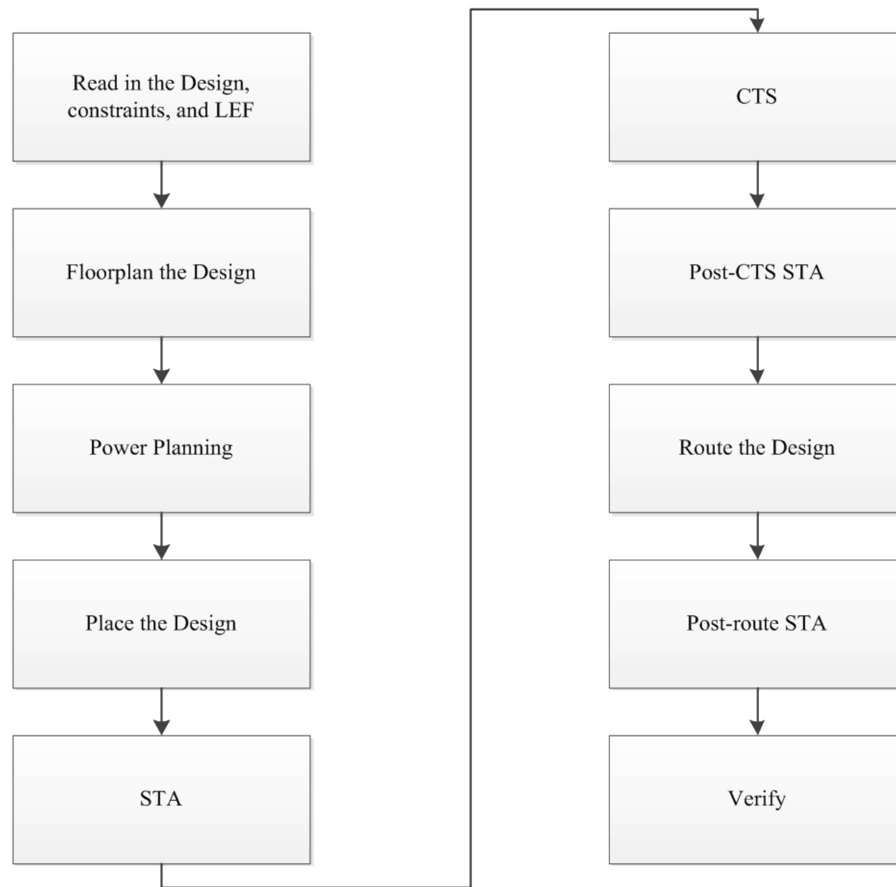


Figure 41. Generic flow of First Encounter

Encounter Results

Table 6. Timing results for STA done at various stages.

	WNS (ns)	TNS (ns)	Violating Paths	All Paths
Pre-CTS STA	6.538	0.000	0	92
Post-CTS STA	4.385	0.000	0	79
Post-route STA	4.401	0.000	0	79

Table 7. General Design Information

Design Status	Routed
Design Name	Neural_block_1
# Instances	18034
# Hard Macros	0
# Std Cells	18034
# Pads	0
# Net	22537
# Special Net	2
# IO Pins	157
# Pins	61380

Table 8. Netlist Information

No of Nets	21842
No of Connections	39506
Total Net Length (X)	2.58E+05
Total Net Length (Y)	2.53E+05
Total Net Length	5.11E+05

Table 9. Power Information

Internal Power	39.43 mW
Switching Power	33.53 mW
Leakage Power	0.01011 mW
Total Power	72.97 mW

Table 10. Floorplan/Placement Information

Total area of Standard cells	412699.795 μm^2
Total area of Core	594953.050 μm^2
Total area of Chip	794960.621 μm^2
Effective Utilization	6.95E-01
% Pure Gate Density	69.37%
% Core Density	69.37%
% Chip Density	51.91%

Table 11. Area of Power Net Distribution

Layer Name	Area of Power Net	Routable Area	Percentage
Metal1	0	594953.0496	0.00%
Metal2	0	594953.0496	0.00%
Metal3	0	594953.0496	0.00%
Metal4	50899.2	594953.0496	8.56%
Metal5	17168.4	594953.0496	2.89%
Metal6	16966.4	594953.0496	2.85%

Table 12. Wire Length Distribution

Total Metal1 wire length	33842.6650 um
Total Metal2 wire length	205622.2650 um
Total Metal3 wire length	232318.6450 um
Total Metal4 wire length	88852.4200 um
Total Metal5 wire length	31676.0300 um
Total Metal6 wire length	4372.9600 um
Total wire length	596684.9850 um
Average wire length/net	26.4758 um

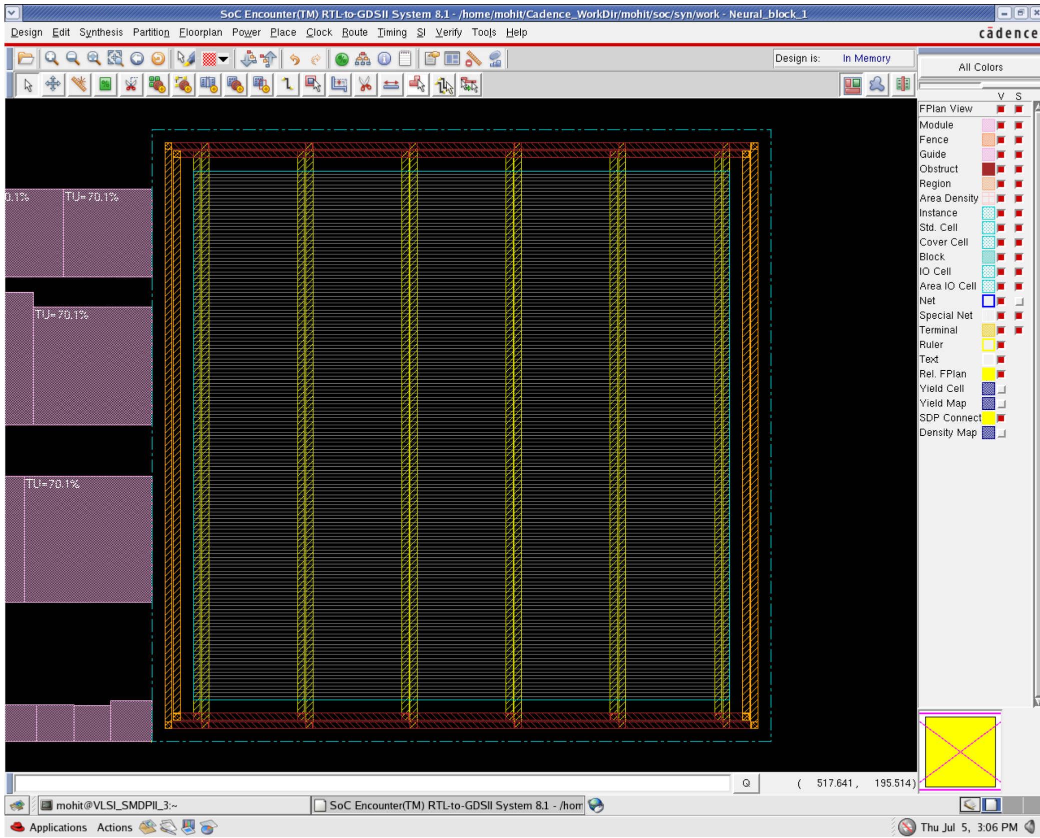


Figure 42. Floorplanning and power planning done.

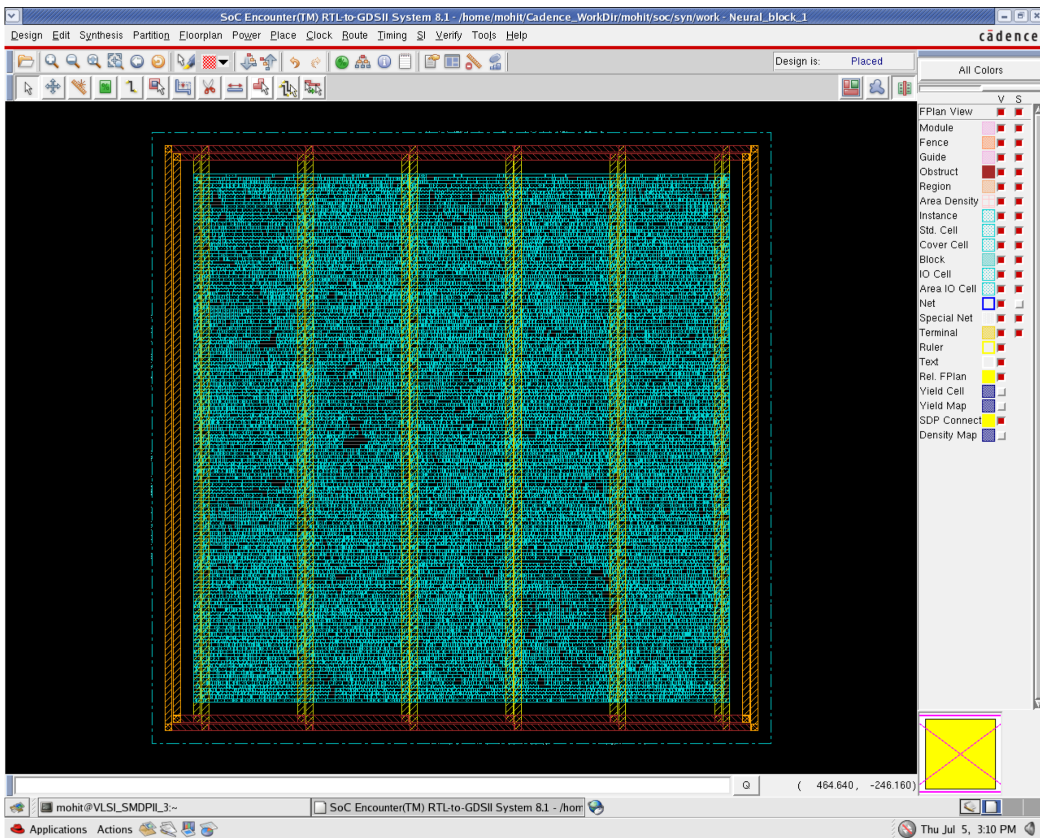


Figure 43. Design placed (Physical view).

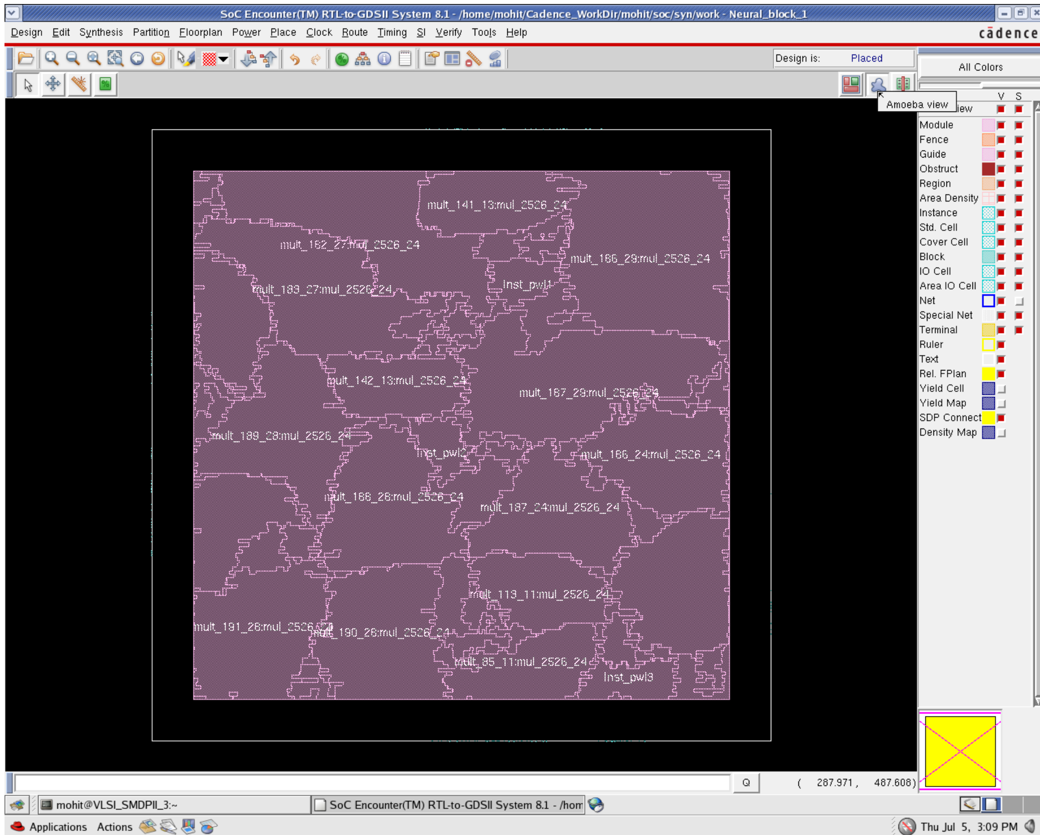


Figure 44. Design placed (Amoeba View).

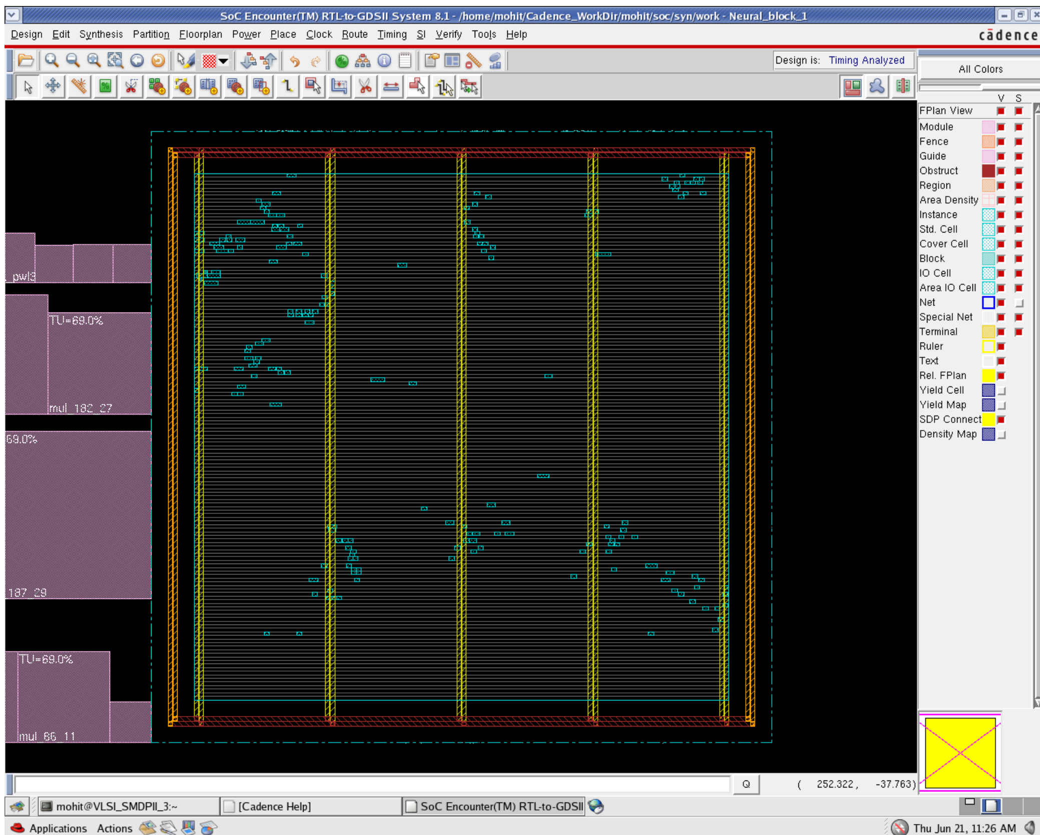


Figure 45. Buffers and inverters added during CTS.

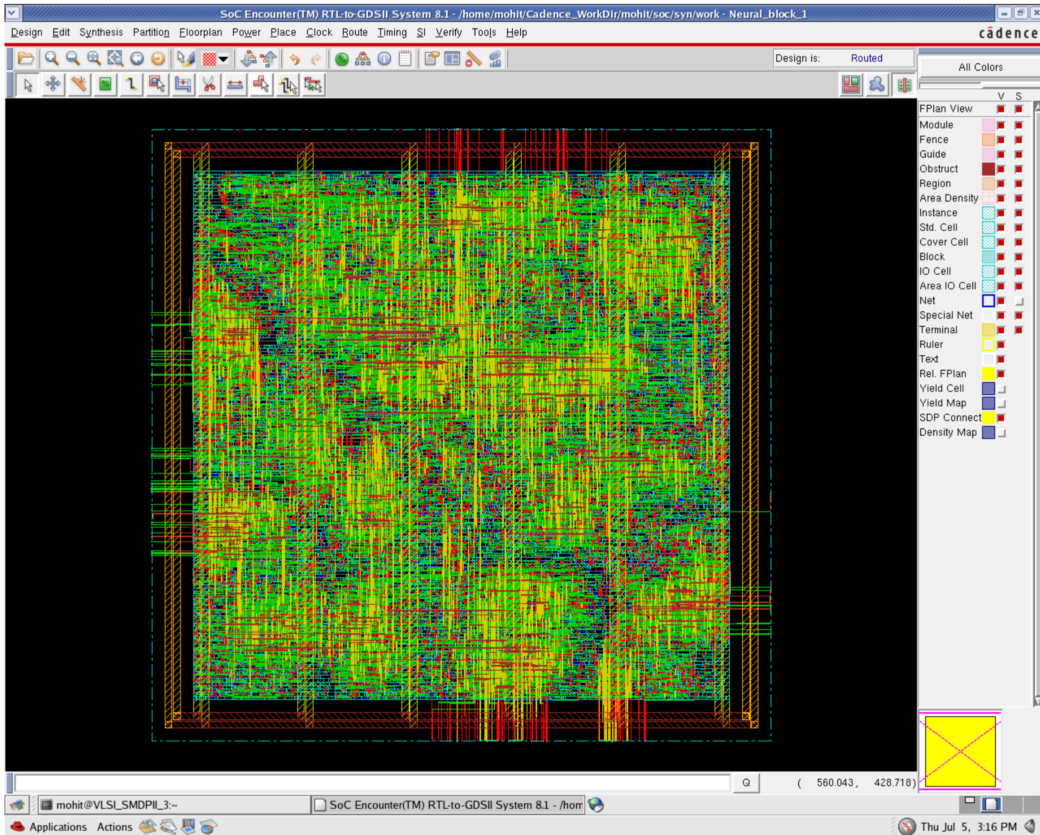


Figure 46. Design routed.

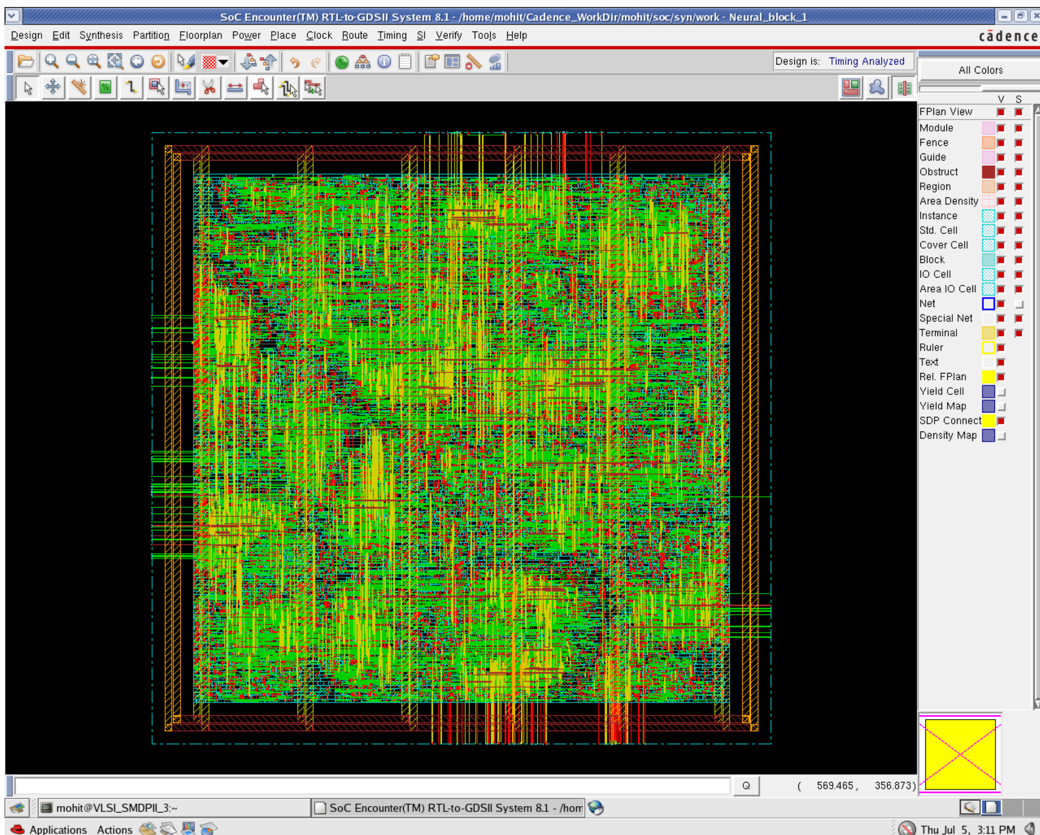


Figure 47. Timing analysed of the design.

6. CONCLUSIONS & FUTURE SCOPE

6.1. Conclusion

The present work focuses on digital implementation of an integrated artificial Neural Network trained with Back Propagation Algorithm. Both the ASIC (semi-custom) and FPGA implementations are done with the same design (possible for an HDL code).

ASIC implementation is more promising compared to its FPGA counterpart in terms of speed, area, power consumption (most of the power is due to leakage in FPGAs like in the design, leakage contributes 86.17% of the total power for FPGA). But when time-to-market and/or field reprogrammability is the requirement, FPGAs are the better choices.

Fixed-point format (FXP) and Floating-point format (FLP) serves purpose of implementing real numbers. FXP is to be used when we need high precision. While FLP is used when precision needs to be compromised due to wide range of inputs/outputs.

6.2. Future Scope

- To achieve higher speed of operation, digital blocks needs to be made parallel. But for large designs, the cost of implementing massively parallel architecture becomes too high. Thus analog implementations are suggested for future work.
- The use of dual Fixed-point format (dual FXP) is suggested for implementing the neural network.

References

1. Abraham A. Artificial Neural Networks. *Handbook for Measurement Systems Design*. John Wiley and Sons Ltd, 2005: 901-908.
2. Goodlett CR, Horn KH. Mechanisms of alcohol-induced damage to the developing nervous system. Alcohol Res Health. 2001; 25(3): 175-84.
3. Haykin, S. Neural Networks: A Comprehensive Foundation. Ed. Martia Horton. Prentice Hall, 2005.
4. Rojas R. Neural Networks: A Systematic Introduction, Springer-Verlag, Berlin, 1996.
5. Lin, M. B. Introduction to VLSI Systems: A Logic, Circuit, and System Perspective, CRC Press, 2008.
6. Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, Vol 65(6), Nov 1958, 386-408.
7. Mead C. A. and Mahowald M. A. A Silicon Model of Early Visual Processing. *Neural Networks,* Pergamon, 1988.
8. Mahowald M. & Douglas R. A silicon neuron, Nature, Vol. 354, pp. 515-518, 1991.
9. Ravi Kumar, R.R. Das, V.N. Mishra, R. Dwivedi, Fuzzy Entropy Based Neuro-Wavelet Identifier-Cum-Quantifier for Simultaneous Identification and Quantification of Odors/Gases IEEE Sensors Journal, Vol.11, Issue 7, May 2011, pp.1548-1555.
10. Ravi Kumar, R.R. Das, V.N. Mishra, R. Dwivedi, Wavelet Coefficient Trained Neural Network Classifier for Improvement in Qualitative Classification Performance of Oxygen-Plasma Treated Thick Film Tin Oxide Sensor Array Exposed to Different Odors/gases, IEEE Sensors Journal Vol.11, Issue 4, April 2011, pp.1013-1018.
11. Ravi Kumar, R.R. Das, V.N. Mishra, R. Dwivedi, A Neuro Fuzzy Classifier cum Quantifier for qualitative and quantitative analysis of Individual Odor Using Responses of Thick-Film Tin-Oxide Sensors, IEEE Sensors Journal, Vol.10, Issue 9, September 2010, pp. 1461-1468.
12. Ravi Kumar, R.R. Das, V.N. Mishra, R. Dwivedi, A Radial Basis Function Neural Network Classifier for the Discrimination of Individual Odor Using Responses of Thick-Film Tin-Oxide Sensors, IEEE Sensors Journal, Vol.9, Issue 10, October 2009, pp. 1254-1261.
13. H. McCartor, "A highly parallel digital architecture for neural network emulation," VLSI Artif. Intell. Neural Netw., pp. 357–366, 1991.
14. U. Ramacher, W. Raab, and J. A. et al., "Multiprocessor and memory architecture of the neurocomputers SYNAPSE-1," in Proc. 3rd Int. Conf. Microelectron. Neural Netw., 1993, pp. 227–231.
15. S. Tam, B. Gupta, H. Castro, and M. Holler, "Learning on an analog VLSI neural network chip," in Proc. IEEE Int. Conf. Syst., Man Cybern., Nov. 1990, pp. 701–703.
16. J. G. Eldredge, "FPGA density enhancement of a neural network through run-time reconfiguration," M.S. thesis, Dept. Elect. Comput. Eng., Brigham Young Univ., Provo, UT, 1994.
17. D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: Theory, algorithm, and FPGA implementation," IEEE Trans. Neural Netw., vol. 14, no. 5, pp. 993–1009, Sep. 2003.
18. H. Ng and K. Lam, "Analog and digital FPGA implementation of brin for optimization problems," IEEE Trans. Neural Netw., vol. 14, no. 5, pp. 1413–1425, Sep. 2003.
19. Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation," IEEE Trans. Neural Netw., vol. 16, no. 6, pp. 1664–1672, Nov. 2005.
20. J. Zhu and P. Sutton, "FPGA implementations of neural networks—a survey of a decade of progress," in Proc. Conf. Field Programm. Logic, Lisbon, Portugal, 2003, pp. 1062–1066.
21. J. Holt and T. Baker, "Backpropagation simulations using limited precision calculations," in Proc. Int. Joint Conf. Neural Netw. (IJCNN-91), Seattle, WA, Jul. 1991, vol. 2, pp. 121–126.

22. A. Savich, M. Moussa, and S. Areibi, “The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study,” IEEE Trans. Neural Netw., vol. 18, no. 1, pp. 240–252, Jan. 2007.
23. V. Havel and K. Vlcek, “Computation of a nonlinear squashing function in digiral neural networks”, 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp. 1-4, 2008.