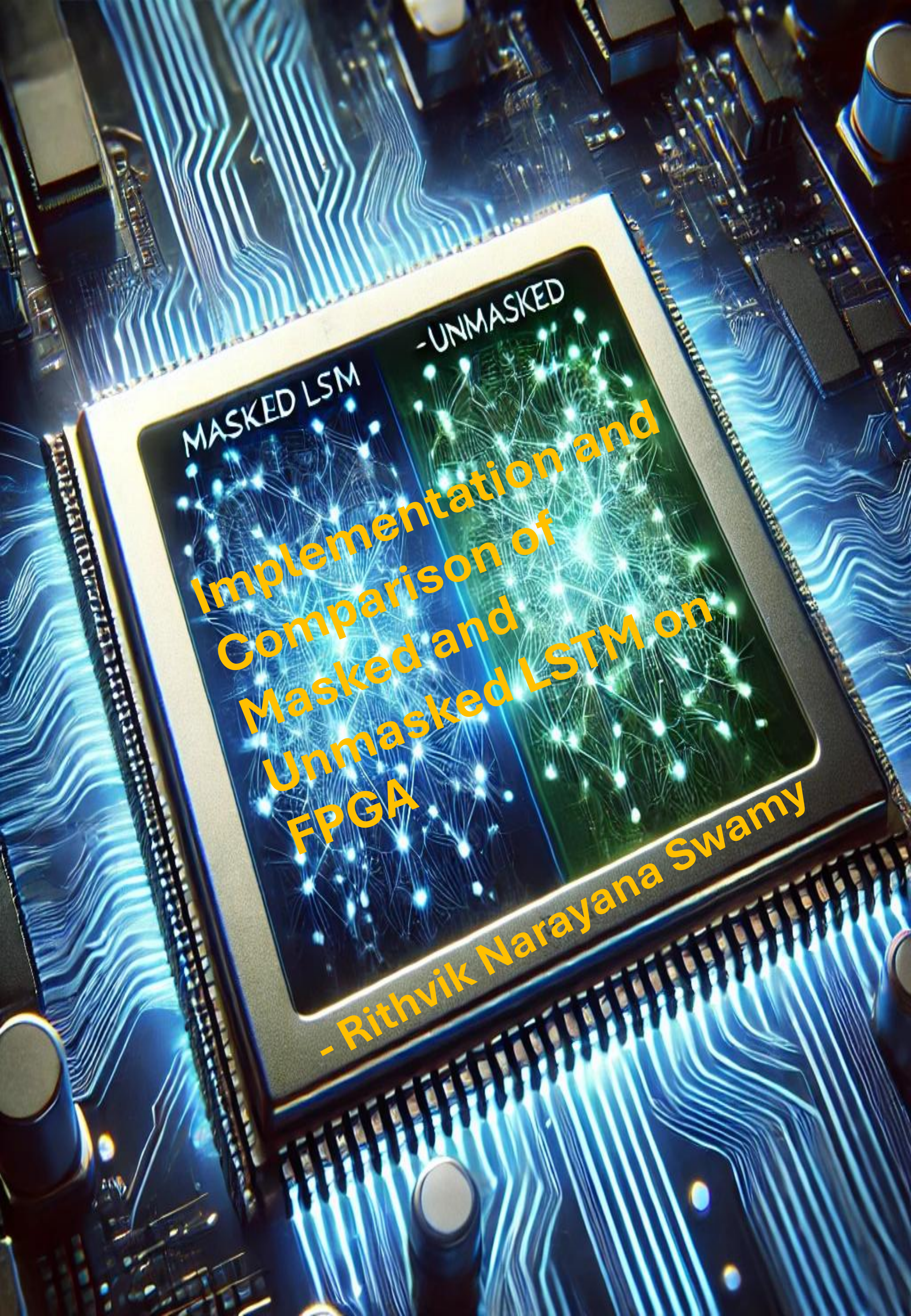
Implementation and Comparison of Masked and Unmasked LSTM on FPGA

- Rithvik Narayana Swamy



Abstract

In the face of escalating side-channel attack threats on neural networks, this thesis is presented with a ModuloNet approach to enhance the security of Long Short-Term Memory (LSTM) networks by building and evaluating specialized masks. This paper aims to develop both masked and unmasked LSTM networks on FPGA platforms, using ModuloNet as basis for achieving low sensitivities to timing, EM and power side-channels attacks. By designing a system that processes MNIST dataset, it makes it possible to evaluate how best the network can perform and secure its operation in processing classifying high-dimensional data. So as to examine the effects of these security measures on performance and resource efficiency of LSTM networks making them more resistant against possible intruders. Different types of software Implemetation and the hardware implementation are explained in detail in this report . The trade-off between masked and unmasked LSTM are provided . Different variants of software implementation , LSTM VHDL code and optimised LSTM VHDL codes are dicussed in detail .

Contents

List of Figures

List of Tables

1 Introduction

Ride-sharing applications, intelligent sorting in Gmail and Amazon recommendations are some of the ways in which neural networks have made life easier[DHH+18] . The most innovative technology is neural network is that once trained they start learning by themselves. safeguarding sensitive data processed by neural networks has become a paramount concern[HKM+17]. Nevertheless, neural networks are vulnerable to side-channel attacks that exploit unintentional information leaks like power consumption, electromagnetic emissions or timing features to infer critical data or model parameters . These attacks represent a significant threat and therefore good countermeasures are necessary.

Traditional defense mechanisms may either result in high computational overheads or partial security, according to recent research. Machine learning model software and hardware implementations have already been successfully attacked through power/EM side channels as shown by several recent studies. It is also true that neural network security extends to Long Short-Term Memory (LSTM) networks,[GDL24] which are recognized for their ability to learn from large and complex datasets. There are various applications that rely on LSTM networks that handle sensitive data such as financial modelling and healthcare analytics. However, in LSTM networks, the confidentiality and integrity of data can be severely compromised by side-channel attacks, which exploit vulnerabilities in hardware implementation to gain access to private information. To tackle this important problem, our work focuses on enhancing the security of LSTM networks based on strong masking techniques that can be used on Field-Programmable Gate Arrays (FPGAs). FPGAs offer a flexible and efficient platform for deploying neural networks, allowing rapid prototyping and implementation.

This report presents a novel approach which combines ModuloNet a technique that uses modular arithmetic to secure neural network inference with an LSTM neural network. In particular, ModuloNet has demonstrated potential in reducing information leakage even in implementations of hidden neural networks . This research applies these state-of-the-art security measures to LSTM networks while evaluating how well they protect against side-channel attacks

This study examines how ModuloNet can be integrated into LSTM frameworks and implemented on FPGA hardware in order to develop a new standard for secure deployment of neural networks. This effort is part of broader discussions around safeguarding machine learning systems against new attacks and enhancing LSTM network security.

1.1 Neural networks

Neural networks, an important foundation that underpins artificial intelligence (AI) and machine learning, are inspired by the biological neural networks found in animal brains. Neural networks consist basically of layers upon layers of interconnected nodes called “neurons” that process input data and apply various connections and transformations to generate an output. By means of this structure, they learn intricate patterns as well as relationships within the material that enhance their performance with access to more information. They are widely applied across industries due to their ability to adapt and learn various things.

Neural nets are used in real life situations where conventional programming approaches cannot solve complex or impractical problems. Computer vision employs neural networks extensively for the purpose of interpreting images and understanding them.

In smartphones as well as security systems, for example, face recognition software uses neural networks to identify and validate individuals with precision. By matching face features from an image or a video with a database, this technology discovers matches and gets better at it the more information it processes. Neural networks are commonly used in natural language processing (NLP) for computers to understand, interpret and produce human language. Chatbots like Siri and Alexa use neural networks to understand voice commands and questions. These systems listen to the user's words, grasp their meaning and intent then provide appropriate responses or actions. Moreover , over time, these neural networks become increasingly proficient in communication as users, who help them learn every time they contact them , teach them to detect subtleties of human language that might be missed out on during training . These examples reveal how pervasive these artificial intelligence systems have become across all the different areas of our lives. changing the way we communicate interact with people using technology.

1.1.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are particularly effective for processing data with a grid-like topology, such as images. CNNs use convolutional layers, pooling layers and fully connected ones to extract and learn hierarchical features from visual inputs. They have revamped the computer vision field allowing significant strides in image and video recognition, image classification among others[GBC16].

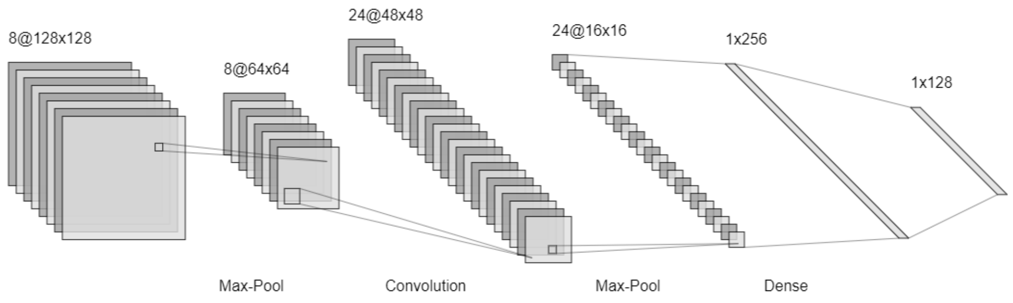


Figure 1.1: A simple CNN architecture.

Here's how a CNN works, step by step:

Input Layer: Input Layer: The first layer of a CNN receives as input the raw pixel data of an image which is usually arranged in height x width x depth format (depth refers to color channels e.g. RGB).

Convolutional Layers: These layers apply a number of filters to the input. Each filter scans the image and computes an activation map that indicates presence of different specific patterns or features at various positions within the inputted framework. These could be simple edges, textures or more complex patterns later on deeper layers of the network .

Activation Function: After a convolutional layer an activation function such as Sigmoid , Tanh and Rectified Linear Unit (ReLU) is used. The activation function allows non-linearity in the network so that it can learn more complicated features.

Pooling Layers: Pooling (usually max pooling) layers reduce the spatial dimensions i.e., width and heights of the next convolutional layer's volume. It also comes with the advantage of making the network have less parameters as well as computation, and a general scheme for what the filters do.

Fully Connected (Dense) Layers: High-level reasoning takes place in the neural network after several convolutional and pooling layers. Neurons in a fully connected layer are connected to all activations in the previous layer as shown in the Figure 1.1. The final fully connected layer combines features learnt by a given network for purposes of classifying images into different categories.

Output Layer: Last fully connected layer contains classification scores for every class in task (e.g., dog, cat, car etc.). To convert scores into probabilities, softmax activation function is usually used here during classification tasks.

1.1.2 Feedforward Neural Networks (FNNs)

Feedforward Neural Networks (FNNs) are the simplest kind of artificial neural network architecture is also known as Multilayer Perceptrons (MLPs). Information always moves one way through an FNN: forward from input nodes, through one or more hidden layers until it reaches the output layer. Simple and versatile as they are, FNNs find wide application across various tasks including classification and regression. A foundational text on FNNs and their applications can be found in this book .[GBC16].

Here's a breakdown of the components shown in the Figure 1.2:

Input Layer: This is the first layer of the network, on which each node (shown as a circular shape) is a representative of one input feature. The raw data that you feed into the network are usually called input features. For instance, when the input data were images, each node could represent the intensity value of a pixel. The label input Layer $\in \mathbb{R}^{15}$" shows there exist 15 input nodes, and hence it implies that this can be done with 15 features.

Hidden Layers: These are intermediate layers between input and output layers and are referred to as "hidden" since they do not have direct contact with external environment (inputs/outputs are part of the environment). Neurons in these layers receive inputs, multiply them by weights and pass them through an activation function to produce an output that passes to the next layer. In this figure there are two hidden layers: The first hidden layer consists of twelve neurons ("Hidden Layer $\in \mathbb{R}^{12}$"), while the second hidden layer has ten neurons ("Hidden Layer $\in \mathbb{R}^{10}$").

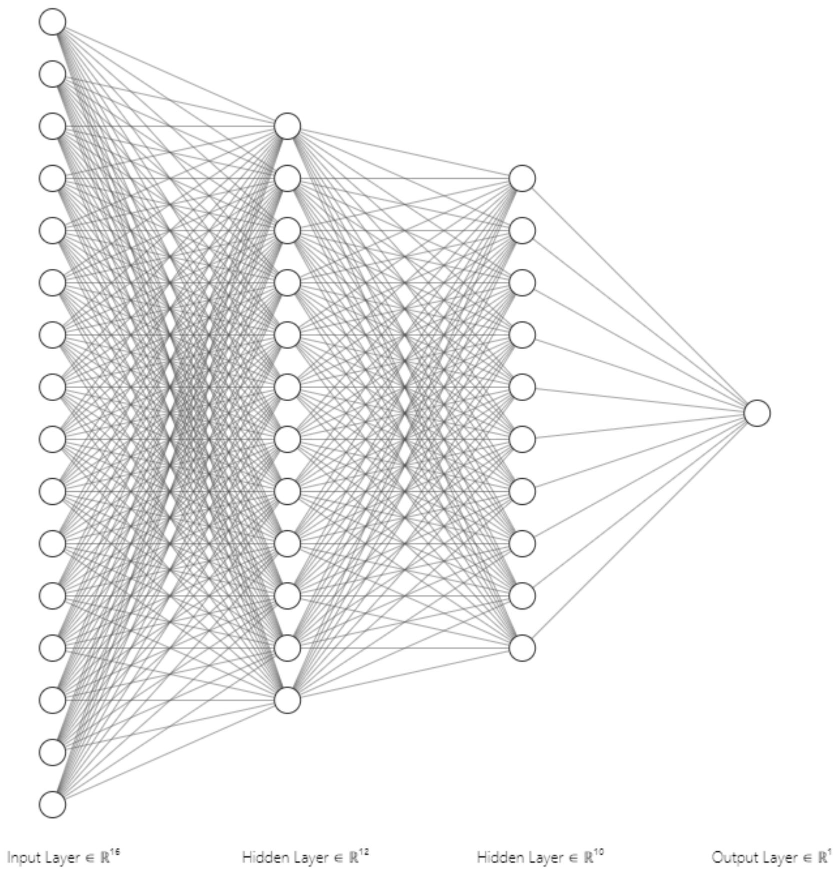


Figure 1.2: A simple FCNN architecture.

Output Layer: This is the last layer in a network. The nodes in this layer would be providing outputs for the network. In numerous tasks, it may be indicated by this diagram as being at a point where it carries out predictions or classifications. It appears from the diagram there is only one output node ("Output Layer $\in \mathbb{R}^1$"), which suggests that the network is designed to perform a regression task or a binary classification task.

The lines connecting the nodes represent the synapses or connections through which data flows from one node to another. Each connection typically has an associated weight, which is adjusted during the training of the network to optimize the network's performance on a given task.

1.1.3 Recurrent Neural Networks (RNNs)

RNN is a kind of Artificial Neural Network which uses the concept of feedback to process the data . The ouput of each iteration is computed with the input of next iteration to get the desired output . The unfolding of time is shown in Figure 1.3 on the facing page. Their architecture makes them ideal for the interpretation of input sequences and the display of temporal dynamic behaviour, so that they can predict time series, recognize voice and model language usage. RNNs differ from the feedforward neural networks as the latter possess memory to process inputs themselves while RNN can absorb information over time. They are distinct from other neural networks due to their ability to learn sequence dependencies[HS97].

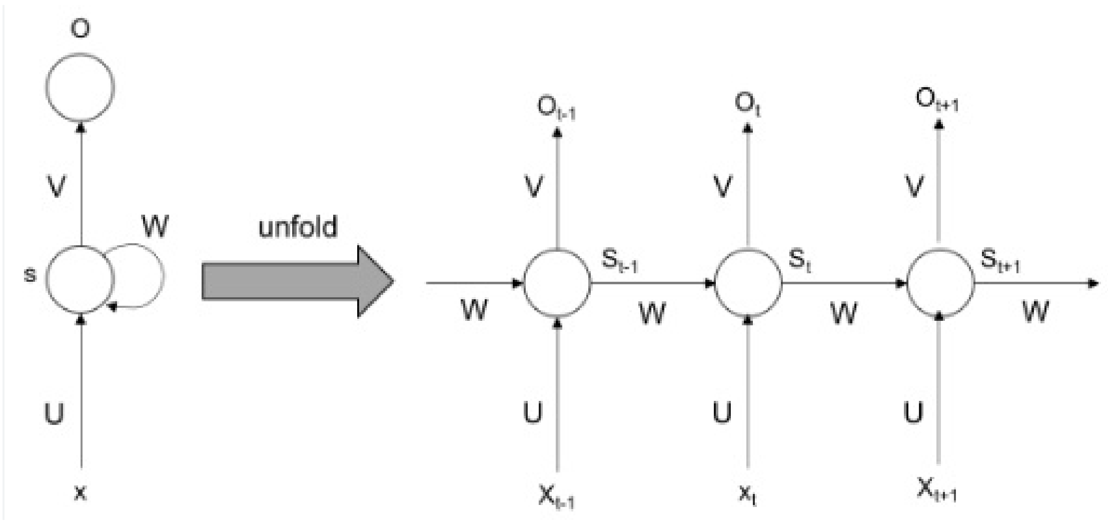


Figure 1.3: Recurrent Neural Network (RNN) unfolded in time.

Nodes:

- x: Input node where the sequence data enters the network.
- s: Hidden state node which represents the memory of the network.
- o: Output node which gives the result after processing the input.

Weights:

- U: Weight matrix applied to the input.
- W: Weight matrix applied to the hidden state (the recurrent connection).
- V: Weight matrix applied to the hidden state to produce the output.

Process: At every time-step t,x_t is fed through together with current hidden state s_t which outputs o_t and updates hidden state for next step s_{t+1}. The network 'unfolds' in time – it is replicated at every time-step having the same weights U, V, W . The previous hidden state s_{t-1} influences the current hidden state s_t through x_t with influence being mediated by weight matrices.

Unfolding:

1. The unraveling arrow indicates that on the right, a single RNN cell is expanded through time (on the right) to process sequences.
2. This is how the RNN maintains memory of internal states till end of a sequence.
3. For this reason, unfolding is undertaken to enable the user visualize how specific neuron in the network processes every element making up an input sequence over time, retaining the history of past inputs in its hidden state.
4. RNNs, unlike CNNs, have another key feature, they can handle input sequences with different lengths and malleable.

Gradient explosion is one key obstacle to training RNNs because when gradient values used during training rise too high, weights are updated very rapidly. This can result into a chaotic training that leads to infinite model weights or NaN fields. Vanishing gradients mean that learning eventually stops when the gradients become too small, unlike gradient explosions.

An alternative type of RNN called Long Short-Term Memory (LSTM) networks was created to address this problem and make it more efficient in handling long term dependencies. Long-term information retention memory cell units are a feature of Long Short-Term Memory Units (LSTMs). By controlling what data enters and exits each cell using gates, LSTMs are able to determine which data in a sequence should be kept and which should be forgotten. Compared with ordinary RNNs ([HS97][G12][SVL14] [CGC14]).

1.2 Side-Channel Attacks

Physical implementation of a computer system is exploited by side-channel attacks rather than its software vulnerabilities , which can do this through indirect measures such as power consumption, electromagnetic emissions or execution times. There are different kinds of these attacks like electromagnetic, acoustic, power, optical timing and memory cache attacks as well as those that exploit hardware weaknesses. For instance electromagnetic attack involves measuring the amount of electromagnetic radiation emitted by the device in order to reconstruct internal signals while in acoustic attack one listens to sounds made by the electrics. Power analysis attack monitors the power consumption to infer what is happening within the system.([B21] [G21]).

The long history of side channel attacks dates back to WWII illustrating their long-standing relevance. This period for example witnessed Bell Labs discovered that encrypted communications could be compromised by electromagnetic spikes produced by encryption devices which prompted significant advancement in countermeasures such as shielding filtering and masking. This historical context also points out that there has been an ongoing struggle between improving side-channel attack methods and the invention of defensive measures[B21] .

The technological advancements have made modern side channel attacks more feasible because of improvements in sensitivity of measurement equipment and computational power, In addition to machine learning applications, these improvements allow attackers to better analyze the data that they have stolen and as a result secure systems can also be invaded. It is worth noting that side-channel attacks are problematic because they are hardly noticeable and do not always leave traces.

1.3 Technical Tools

The following section is the broad description of the implemented technical tools during the stage of the thesis activities. The scope includes designing and synthesizing the function of LSTM neural Network in VHDL, and Xilinx ISE 14.7 Design Suite software package as well as Spartan 6 FPGA(Field-Programmable Gate Array).

1.3.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGA) have seen more improved than it was many long ago. The first application of, that happened in 1980s, FPGAs was as a tools for prototyping digital circuits. Along time, semiconductor designs have been making FPGAs more complex, and therefore they can process a wide range of functions. A widening range of applications many of which have now turned mainstream [MEY14]. FPGAs, among the most reprogrammable devices, after manufacturing, execution of any specific tasks becomes possible with some modifications. This function changed the fate of digital devices with FPGAs getting applied everywhere including in numerous real-life applications [MC07].

The usual FPGA architectural design is made up [RES93] of two essential parts. The first element comprise logic blocks which can be assigned to do predefined logical functions is the logic block. A number of LUTs (205) in many FPGAs are realized via a technique using RAM [BFRV92].

Apart from this, the trace of the design is made using Spartan-6 LX-75 FPGA from Xilinx given that [11b]. Being the LFC 74,637 logic blocks organized as a 11,662 slices. An internal clock oscillator is an additional component by the chip that outputs a clock signal frequency ranging from 40MHz to 50MHz. Furthermore, the 128KB of block RAM inside FPGA is able to store up to 18 kilibits. To be more specific about the consumptive part of the FPGA[M23].

1.3.2 VHSIC Hardware Description Language

VHDL is primarily a text-based hardware description language . VHDL is a flagship technology of the digital design field, and this universal language is used to describe and verify the simulation of advanced digital designs. Arising in the 1980s, VHDL was primarily created by the U.S. Department of Defense in order to streamline the development of dynamic electronic systems [P02]. VHDL is easy and reliable,hence digital logic circuits, embedded FPGA-based systems and digital systems of a variety of industries use this language for their design.

The feature which a significant part of VHDL expertise lies in is its hierarchical nature, letting designers to define reusable modules which can encapsulate complex functionality[S00]. This hierarchical structure makes feasible the modular design concepts and hence gives the opportunity to engineers for effective management of the complexity of large-scale digital systems. In addition, VHDL's compatible model for concurrent and sequential execution allows designers to codify both synchronous and asynchronous flows of their designs, which makes complex control act and data processing algorithms more feasible.

Both VHDL and Verilog are the two hardware description languages which are widely used for digital design which includes FPGA implementing also. Although both of the semi-formal languages perform similar functions, VHDL and Verilog have the differences. When it comes to implementing neural networks on FPGA, VHDL offer's certain advantages[00]:

- **Explicit Modeling**: VHDL's highly expressive syntax and strong type system can be beneficially expressed for constructing complex network architectures, moreover, they guarantee building precise formulations of the design description.
- **Structured Design**: Neural network models coded in VHDL follow the hierarchical modeling style, which provides the possibility for one to establish reusable modules in networks that are easy to maintain and scale.
- **Formal Verification**: VHDL may help check the neural network implementation for errors by including assertions and formal verification methods, which is vital for safety-constrained applications like autonomous cars and airplanes.

1.3.3 Xilinx ISE Design Suite

During the course of this thesis, the Xilinx ISE Design Suite (version 14.7) was extensively employed to design modules, simulate designs, and ultimately develop the final design. With the Xilinx Vivado Design Suite having replaced ISE in the year 2012 as the design tool, the implementation of ISE was still a dominant need because of the hardware constraints. In this suite you will find several utilities that automate the purpose of design simulations and FPGA upload.

In order to synthesize the VHDL design on the FPGA board with ISE environment, there are three crucial things to consider [11a]. At first place we need to write the behavioural code for the hierarchical modules and simulate the model. Once it is working as expected then a netlist is drawn that using a synthesis technique indicating the necessary primitives and their interconnections. Lastly, the design is contained on the specified FPGA board. It starts with translation, expanding netlist with functionally oriented and user-defined constraints [NWA16]. Moving into the following stage the mapping phase assigns FPGA-specific logic resources to primitives. Moreover, confirming such ones in terms of the physical layout of the components on the FPGA and the interconnections between them falls within the scope of the Place and Routing step.

When the design plan for a particular FPGA gets finished, the end product is a .bit file. Through this file, a detailed design configuration is achieved, while the FPGA is specified to follow the same manner. However, there is no universal bitstream on which all vendors can decide and this particular format is hidden by the vendors [EMP20]. Making the FPGA to work with the communication medium, the bit stream file must load on the FPGA telling it to take the specific configuration.

1.4 Goals

This research will explore the practicality of masking techniques, with particular emphasis on ModuloNET and its application for masking Long Short-Term Memory (LSTM) on FPGA platforms in terms of performance, efficiency, and energy consumption. In this work, we intend to appraise whether the ModuloNET masking technique improves the security resistance of LSTM neural networks to side-channel attacks.

we evaluate several software implementations of LSTM networks in the first place to see their accuracy and trade-offs. This assessment will be conducted by comparing the frameworks that have and do not have libraries, presented in terms of throughput, power consumption, and resources utilization. Through assessment of the mentioned factors, we intend to determine the configuration of masked and unmasked LSTM networks most adequate for hybrid in terms of both security and throughput.

Then, two kinds of LSTM networks with masked and unmasked scheme will be designed and implemented by means of FPGA hardware. It is, in fact, making the masked LSTM of the project using ModuloNET-based masking approach and using it along with the unmasked counterpart of the LSTM. Besides, VHDL codes are examples that we will discuss ranging from concurrent and sequential designs, these designs play a big role of being performance, security, and resource efficient when they are applied conversely with ModuloNET masking platform.

1.5 Outline

The thesis's following sections are organized as follows:

- **Chapter 2: Background** - The section 2.1, focuses on things with LSTM network architecture, showing how they function with the help of a block diagram Figure 4.1 on page 39. And then, Section 2.2 brings about the modified National Institute of Standards and Technology database and tells about its being and what it means in the past time research. The Section 2.3 discussing ModuloNet and its essential contribution on optimizing the security of LSTM systems. Chapter 2 lays the groundwork for the rest of the research by critically evaluating the key concepts.
- **Chapter 3: Methodology** - The step by step process used to convert the image to predict a number from the image is explained in detail . In this section you can also witness the complexities involved in implementing different neural networks of different sizes .
- **Chapter 4: Software Implementation** - This sections will explain different methods of software implementation such as using TensorFlow in python and unrolled neural network . Different sizes of neural network are implemented and tested in this section . Trade off between all these LSTM variants are explained .
- **Chapter 5: Hardware Implementation** - This is where you can find the detailed version of VHDL codes . Testing different size neural network using both combinational circuit and sequentially circuit . The simulation is performed on all the varaints and tested out . Due to the board limitations the combinational circuit cannot be mapped into hardware . The

Masking is done using ModuloNET and the masked LSTM is implemented on hardware . various metrics such as device utilisation summary and advanced HDL synthesised are provided in this chapter .

- **Chapter 6: Discussion** - The section 6.1 provides the analysis of Result , different comparison tables . section 6.2 gives the comparison with existing solutions and the last part discusses the trade offs between Security and performance .
- **Chapter 7: Concluding and Prospects for Future Work** - The section closing this thesis looks at the findings that were made and offers future research possibilities pertaining to unusual product masking.

2 Background

2.1 Long Short-Term Memory (LSTM) Networks

Long Short-memory (LSTM) networks are an example of recurrent neural networks (RNN) used in deep learning. In contrary with Traditional RNNs, LSTMs are robust in the sense that they are designed to handle each sequential data which are hard to model long-term dependencies with problems like vanishing and exploding gradients. LSTM contains two main memory blocks ,short memory cells and long memory cells can memorize the data for a very long time, gates which influence the data flow. Using what is learned, LSTMs can make use of the past information, which is one of the features allowing them to successfully resolve time series prediction and natural language processing tasks, as well as tackle speech recognition([HS97] [O15]).

The core components of an LSTM unit are the cell state, and three types of gates: as the overlap crosses the lower bound of the input gate, the gate becomes open and information flows in, then at the bottom of the output gate, the gate is open and the information becomes outputted, and in the top Where the signal from the previous step intersects the forget gate, the gate becomes open and old information is expelled. The cell state behaves like a moving track, it carries vital information which is further working through the means of data. The gates are made up of a deformed neural net layer, which conducts the message flow, and a pointwise multiplication term, which is responsible for the flow of data. The input gate vector being selected as the number of newly introduced information, the forget gate decides which data can be throwed away, the output gate represents that what recorded data should be used to generate it into the output signal at each step. This mechanism of gating IP LSTMs makes them critical in choosing which information is appropriate for long sequences with a "vanishing gradient"[FSC00] .

LSTMs have showed their ability to be used in a wide array of domains, such as speech recognition, handwriting recognition and machine translation, which make them not only versatile but also effective. While applying them for language modeling, machine translation, and sentiment analysis tasks are the common ones in natural language processing. LSTMs have given excellent recommendations as the kind of models to be used in time series prediction in areas such as stock price forecast, weather trend prediction, and health pattern prediction of patients, to mention but a few. LSTMs show their best capacities in the domain of long sequence data processing because they're known for their sophisticated memory and gating mechanisms. They substitute the limitations of the usual RNNs' functioning and open up new possibilities in sequence analysis and prediction.

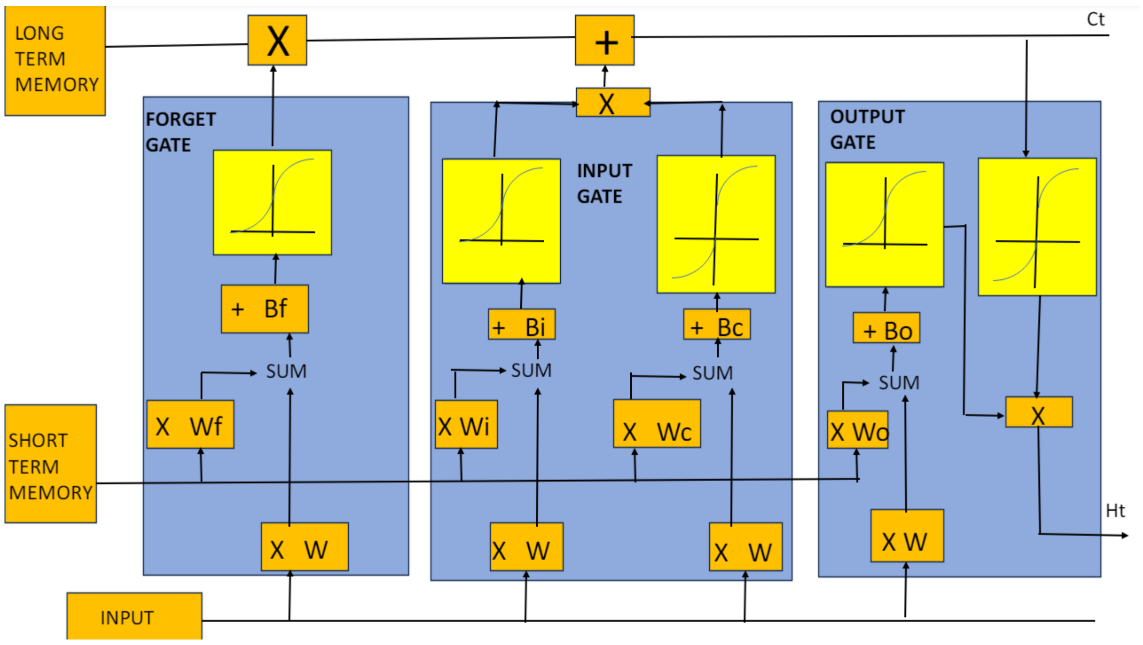


Figure 2.1: Long Short-Term Memory.

The image depicts an LSTM cell comprising of three gates that control the information flow: the forget gate, the input gate, and the output gate. Each gate has a distinct role:

$$
\begin{aligned}
f_t &= \sigma((wf_1 \cdot x) + (uf_1 \cdot h_{t-1}) + bf_1) \\
i_t &= \sigma((wi_1 \cdot x) + (ui_1 \cdot h_{t-1}) + bi_1) \\
c_t &= \tanh((wc_1 \cdot x) + (uc_1 \cdot h_{t-1}) + bc_1) \\
o_t &= \sigma((wo_1 \cdot x) + (uo_1 \cdot h_{t-1}) + bo_1) \\
c_{t-2} &= c_t \cdot f_t + (i_t \cdot c_t) \\
h_{t-2} &= \tanh(c_{t-2}) \cdot o_t
\end{aligned}
$$

- **Forget Gate**: this gate Fg determines what information has to be discarded from the cell state. It looks at the previous short-term memory (H_{t-1}) and the current input (X_t) by the given formula $\sigma\left((W_f H_{t-1} + X_t + b_f)\right)$, where σ is the sigmoid function, W_f is the forget gate weight matrix, and b_f is the forget gate
- **Input Gate**: This gate says whether new information is to be written into the memory state. This is where the decision making system starts. It employs two equations: there exist one to produce $\tilde{C}_t = \tanh(W_C \cdot [H_{t-1}, X_t] + b_C)$, and another one to select updated values $i_t = \sigma(W_i \cdot [H_{t-1}, X_t] + b_i)$.
- **Output Gate**: This gate gives a new hidden state as H_t and externally outputs. Then, it will calculate $o_t = \sigma(W_o \cdot [H_{t-1}, X_t] + b_o)$ and write C_t the cell state. Finally, the new hidden state will be computed as $H_t = o_t * \tanh(C_t)$.

Network is the vector of weights and biases of each gate, which is selected with the help of the sigmoid (σ) and hyperbolic tangent (tanh) functions added for nonlinearity by applying element wise operations to the state of cell.

Once the LSTM is trained using the MNIST dataset (containing hand-written images of digits) it successively processes vertical and horizontal of pixels from each image one at a time as its time steps are walked through. At every move, the LSTM refreshes the memory cells comprising the key and its association value, i.e., the present row or column data and the accumulated value of the cells together, thus the more it unbundles, the more information from the image it retains. Upon seeing the final row or column, the LSTM's output gate creates a feature vector ready for classification. The vector is then sent through a fully connected layer with a softmax function to produce probabilistic values for each class digit. The training process starts with tuning the LSTM's weights (W) and the biases (b) on the gates as well ensuring that they are consistent with accurate handwritten digit classification.

2.2 Modified National Institute of Standards and Technology database

The MNIST data set, is a large archive of handwritten digits that have earned widespread regard amongst the fields of machine learning and computer vision. Containment of 60 000 training images and 10 000 testing images is in them. This data accumulates from a mixture of handwritten materials written by the U.S. Census Bureau agents and high school students when they fill out forms. The one-of-a-kind aspect of this dataset is that each image is resized and centered into a 28-by-28 pixel frame and thus, the standard benchmark for networks is based on these images.

At first, the pictures of shown NIST were size-normalized but adjusted to a fit within a 20x20 pixel both guided by the aspect ratio. This process would bring in light and dark levels as the antialiasing technique employed in the normalization algorithm is used. In order to organize the picture at the data center of 28x28 field, the center of mass of the pixels was calculated, and the image translating was made.

Many Individuals that utilize their hands-on skills like data scientists and machine learning practitioners often use MNIST dataset as a starting point. It is because of the data set's simplicity and the minimal protocol required. It is a reference data set designed for the model development and evaluation. The modern models based on the classical algorithms and even neural networks are among the candidates. The data can be conveniently accessed and loaded into Python through packages like Deep Lake and can be used to quickly set up script files for different purposes like for training and testing.

The MNIST dataset is not just a lifesaver for starters but very popular in testing and research of state of the art methods in field of machine learning. As the dataset gets created, its structure is also improvised and normalized just the way many other datasets have been coded for and known to have contributed greatly to the modern machine learning breakups.

2.3 ModuloNet: Enhancing LSTM Security

To begin with, ModuloNET utilizes modular arithmetic in the application of neural networks for security purposes and to keep the side channel attacks away. By interjecting modular arithmetic operations into neural network operations it is possible to realize a higher level of confidentiality and integrity.

Fundamental Idea: The key point in ModuloNet concept is that it replaces integer arithmetic in neural network inference processes by modern arithmetic, i.e. modular arithmetic. The shift also permits narrowing of the activation function inputs and output layer restrictions by expanding the modular scale. Hence, such a move guarantees that there is no information leak and provides great security that ultimately protects the entire neural network.

Modular Arithmetic in Neural Networks: Rather than changing the neuron networks from scratch, ModuloNET initializes the modular arithmetic framework and leverages it for a secure implementation of the process. The method of ModuloNET brings the computational calculations to a single block controlled problem, it eliminates the risk of side attacks, and in result, gives a more secure neural network

Efficient Masking Techniques: In modeling it, ModuloNET applies compact maskings, namely masked thresholders and comparators, in order to process the activation functions and compute the results at the output layer in a secure way. These measures are incorporated for the purpose to safeguard confidential data and do not allow side-channel attacks to invade the sercurity of the neural network[HKM+17].

Hardware Implementation: ModuloNET combines hardware layouts for both unmasked and fully masked embedded modules. The overall masked fabrication of ModuloNET Malta features various top-level masked parts like block RAMs storing weights and bias. In so doing this practical deployment of modules arithmetic can be integrated into safeguarding neural network operations on the hardware level[DHH+18]

Benefits: ModuloNET integrates neural networks to modular addition, which consequently leads to numerous positive effects. It gives an efficient framework where models can be trained using robust masking technologies, and also offers the capabilities of developing unique mask designs to be used by ML. ModuloNET combines secureness with efficiency securing its position as a scalable and cheaper alternative between it and other masking techniques with increased resiliency against first-order attacks

Summary: All in all, ModuloNET variation of neural networks does in such way that it brings the modern approach of combining arithmetic of modularity into neural networks to ensure privacy, prevent side-channel attacks and enhance model integrity and security. ModuleNET commits to achieving this through use of modular arithmetic and skilled masking methods, thereby making a transition to a more advanced cryptographic technique with an improved protection from leaks of confidential data([HKM+17][DHH+18])

3 Methodology

3.1 Complexity

The Figure 3.1 on page 29 represents a LSTM neural network having one unit cell in each neuron followed by 10, 10 fully connected layers . The Neural network looks simple but the complexity arises while storing weights, handling multiplication and addition . In the fully connected layer each neuron has its own weight which has to be multiplied with the input pixel value . For 784 pixel values each neuron has 784 weights and a bias . Therefore a total of 7840 weights and 10 biases for the first fully connected layer . each weight and bias is 16 bits . 1st bit represents *signbit* , the next seven bits represents integer part and the the last 8 bits represent fractional part . Let's understand the complexity to build a lstm layer with two unit cell followed by 30,30,10,10 fully connected layers . The accuracy for this network is greater than 91% whereas requires four memory storage . Activation functions tanh and sigmoid need exponential function to compute the values , but ise 14.7 does not support exponential function . So to implement activation function we use Look Up Tables (LUT).The look up table contains all the possible values (2**16) and their sigmoid and tanh output . The LUT had 65536 entries in it which becomes too large for the synthesis tool to handle , therefore this is stored in RAM . This RAM values can convert any value in the range -128 to 127.999 to their approximated activation function value .

Here's the VHDL code for multiplication:

```vhdl
begin
    unsigned_A <= unsigned(inputx);
    unsigned_B <= unsigned(inputy);
    unsigned_x <= unsigned_A(14 downto 0);
    unsigned_y <= unsigned_B(14 downto 0);

    mul_result <= to_integer(unsigned_y) * to_integer(unsigned_x);
    mul_out <= std_logic_vector(shift_right(unsigned(to_unsigned(mul_result,
mul_out'length)), 8));
    sign_bit(15) <= inputx(15) xor inputy(15);
    sign_bit(14 downto 0) <= mul_out(14 downto 0);

    output <= "0000000000000000" when ((inputx = "0000000000000000") or (inputy =
"0000000000000000")) else sign_bit;
end architecture Behavioral;

```

Here's the VHDL code for addition:

```
        begin

        process(inputx, inputy)
            variable unsigned_A, unsigned_B : UNSIGNED(14 downto 0);
            variable sum_out                 : INTEGER;
            variable sign_out                 : std_logic ;
        begin
            unsigned_A := UNSIGNED(inputx(14 downto 0));
            unsigned_B := UNSIGNED(inputy(14 downto 0));

            if inputx(15) = inputy(15) then
                sum_out := TO_INTEGER(unsigned_A) + TO_INTEGER(unsigned_B);
                sign_out := inputx(15);
            elsif unsigned_A > unsigned_B then
                sum_out := TO_INTEGER(unsigned_A) - TO_INTEGER(unsigned_B);
                sign_out := inputx(15);
            else
                sum_out := TO_INTEGER(unsigned_B) - TO_INTEGER(unsigned_A);
                sign_out := inputy(15);
            end if;

            output <= sign_out  & std_logic_vector(TO_UNSIGNED(sum_out, 15));
        end process;

    end Behavioral;


```

- Number of weights for fully connected layer = $784 \times 30 + 30 \times 30 + 30 \times 10 + 10 \times 10 = 23520 + 900 + 300 + 100 = 24820$ (each 16 bits long).
- Number of biases for fully connected layer = $30 + 30 + 10 + 10 = 80$.
- Number of multiplications = $24900 + 24900 = 49800$.
- Number of additions around $50,000$.

The code for both LSTM layer having one unit cell followed by 10,10 fully connected network and lstm network with 30,30,10,10 is provided in my github repository . So let's choose a simpler network for understanding . So for the first layer of fully connected network you need to multiply weights and pixel values . That is 7840 weights multiplied by 784 pixel values multiplied by 10 neurons which equals to $7840x7840$. The multiplication has a complexity of $o(n2)$. Multiplying two 16 bit numbers produces a 32 bit number . So after each multiplication the value keeps growing with the power of two . Due to this reason after each multiplication the 32bit output is shifted by 8 (divide the number by 255) to get the fractional part 8 bits , the rest are the integer part and sign bit .

Here's the VHDL code for sigmoid activation function using single port RAM:

```
architecture Behavioral of ram_sigmoid is
type mem_array is array (0 to 65535) of std_logic_vector (15 downto 0);
signal sigmoid : mem_array := (
0 to 4 => "0000000010000000",
5 to 8 => "0000000010000001",
9 to 12 => "0000000010000010",
13 to 16 => "0000000010000011",
17 to 20 => "0000000010000100",
21 to 24 => "0000000010000101",
25 to 28 => "0000000010000110",
29 to 32 => "0000000010000111",
33 to 36 => "0000000010001000",
37 to 40 => "0000000010001001",
41 to 44 => "0000000010001010",
45 to 48 => "0000000010001011",
49 to 52 => "0000000010001100",
53 to 56 => "0000000010001101",
57 to 60 => "0000000010001110",
61 to 64 => "0000000010001111",
65 to 68 => "0000000010010000",
69 to 72 => "0000000010010001",
73 to 76 => "0000000010010010",
77 to 80 => "0000000010010011",
81 to 84 => "0000000010010100",
85 to 88 => "0000000010010101",
89 to 93 => "0000000010010110",
94 to 97 => "0000000010010111",
98 to 101 => "0000000010011000",
102 to 105 => "0000000010011001",
106 to 109 => "0000000010011010",
110 to 113 => "0000000010011011",
114 to 118 => "0000000010011100",
119 to 122 => "0000000010011101",
123 to 126 => "0000000010011110",
.
.
.
.
34187 to 65535 => "0000000000000000");
begin
process (clk)
begin
if (clk'event and clk = '1') then
if (we = '1') then
sigmoid(conv_integer(a)) <= di;
end if;
end if;
end process;
do <= sigmoid(conv_integer(a));
end Behavioral;
```

By doing this we are able to keep the value after multiplication in 16 bits without actually changing the value . we can use xor operation for sign bit because xor truth table matches multiplication truth table .

- positive(0) x positive(0) = positive (0)
- positive(0) x negative(1) = negative(1)
- negative(1) x positive(0) = negative(1)
- negative(1) x negative(1) = positive(0)

The first bit of every number is a sign bit , 0 represents positive and 1 represents negative number . For addition if both the numbers are positive or negetive we add else we subtract . During the subtraction the large number should be subtracted from small number . To check which nuber is large we need a comparator for each addition .

Activation functions tanh and sigmoid need exponential function to compute the values , but ise 14.7 does not support exponential function . So to implement activation function we use Look Up Tables (LUT).The look up table contains all the possible values (2**16) and their sigmoid and tanh output . The LUT had 65536 entries in it which becomes too large for the synthesis tool to handle , therefore this is stored in RAM . This RAM values can convert any value in the range -128 to 127.999 to their approximated activation function value .

3.2 Flowchart

The Figure 3.1 represents an LSTM Network with 2 fully connected layers . Each fully connected layer has 10 neurons . After 784 iterations of LSTM layer the result is given to the fully connected layer to predict the output digit using a softmax function . The softmax function calculates which neuron has the max value at the last layer and prints out the index of the max value . This is the output of the network .

For example if the digit is 8 as shown in Figure 3.3 . The max value should be present in the neuron N9 , so the weights and biases are computed in a way to get this desired output . The detail process of each block in the flowchart is explained in the next section .

3.2.1 STEP 1

Let's start with the input digit as shown in Figure 3.3 . The first step is to convert this digits whose dimensions are 28 $*$ 28 , into a list of 784 bytes. Each byte represents one pixel . The size of each pixel representation can vary , you can also represent each pixel with 16 bits , depending on your data width of your neural network . I have implemented LSTM neural network with both 8 bits and 16 bits to represent each pixel . The Binary values for all the 10000 testing data is available in the Mnist dataset website , but using python you can also import the image and get the integer values . This integer values are normalized so that it does not exceed the limit and further converted to binary .

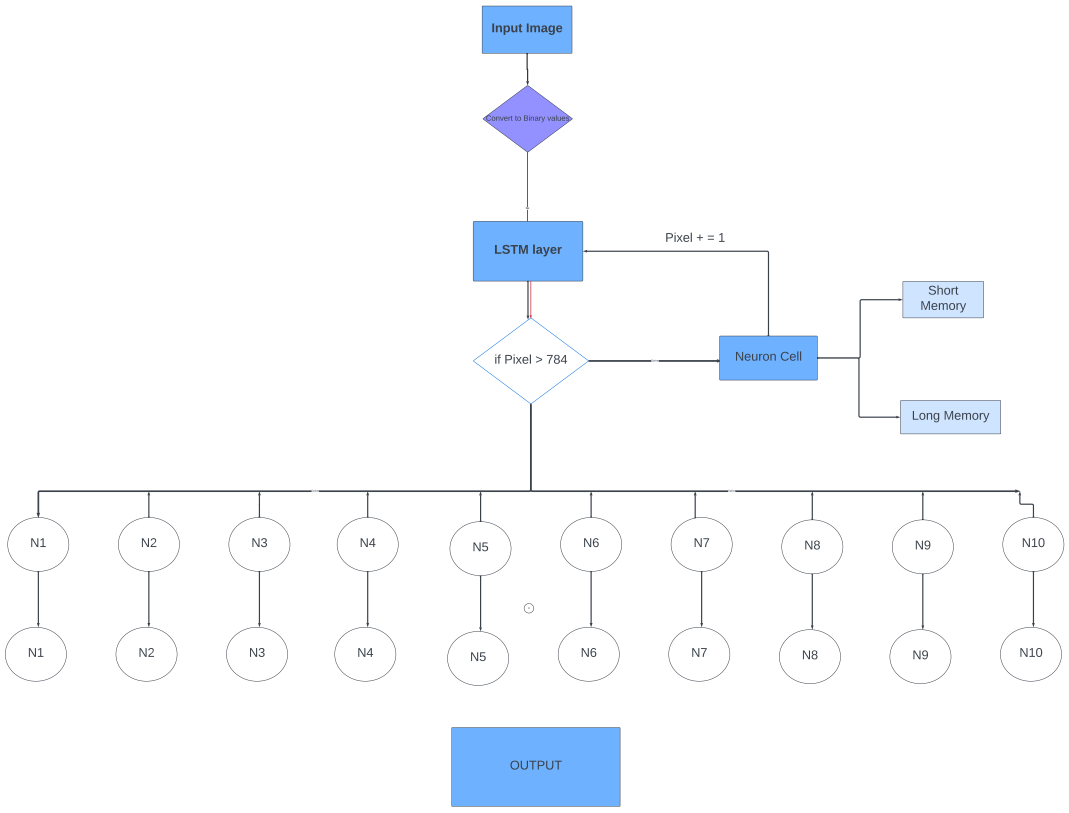


Figure 3.1: Complete data flow of LSTM Neural Network

Figure 3.2: Input data (digit) fed to the LSTM neural network

3.2.2 STEP 2

Now this data is sequentially fed to the lstm layer . The first byte enters the lstm layer , the long term memory and the short term memory are initially set to zero . Then the process follows as shown in Figure 3.1 . Then after first iteration you get the long term memory and short term memory which is used to compute the next input . After 784 iterations all the values complete their computations and the memory elements are set to zero for the next input .

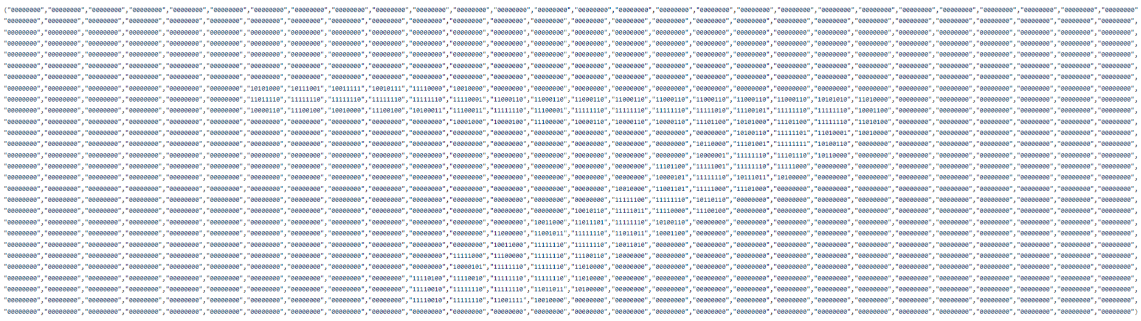
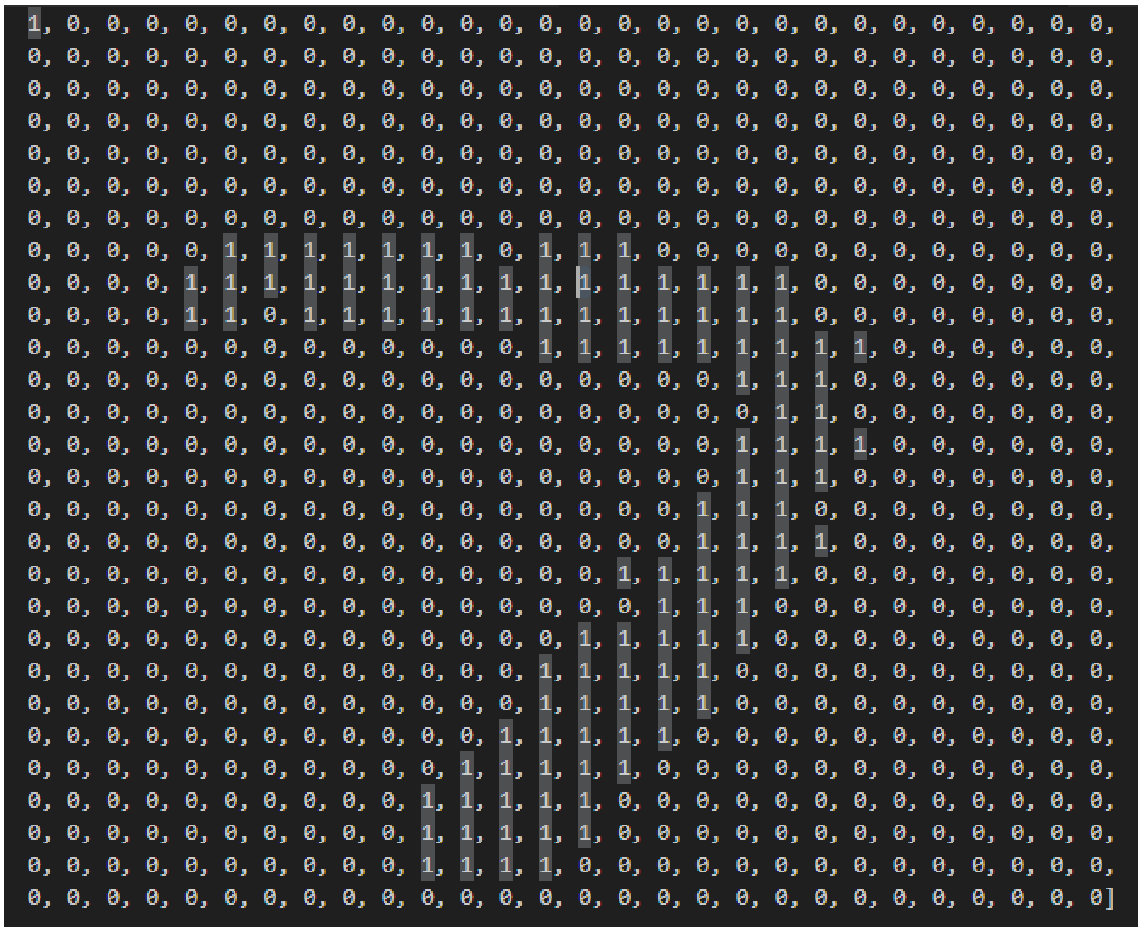


Figure 3.3: Input data represented in binary format

3.2.3 STEP 3



```
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 3.4: LSTM layer output

The output of this lstm network is reduced to give either 0 or 1 . zero represents the Background whereas one represents the image as shown in Figure 3.4 . The output of the LSTM can be 8 bits or 16 bits depending on your FCNN . If it is 8 bit we represent it as "00000001", 16 bit is represented as "0000000011111111". In 8 bit the value follows the normal binary numbers , whereas in 16 bit representation the last 8 bits represnt fractional value . Therefore this value "0000000011111111"tends to 0.999 which almost equal to 1 . We can also use "0000000100000000"to represent 1 . But the actual values are stored in the short term memory and the long term memory to compute the next value .

3.2.4 STEP 4

After the 784 iterartions you get the LSTM output as shown in Figure 3.4 . This output is fed to the fully connected layer . Keeping the complexities into count , the FCNN layer is designed . The single neuron working is shown in Figure 3.5 . Each neuron has it's own neuron memory . In this case each neuron hold 784 values due to the number of image pixels . Then at each iteration the weight is obtained from memory and multiplied with the input .

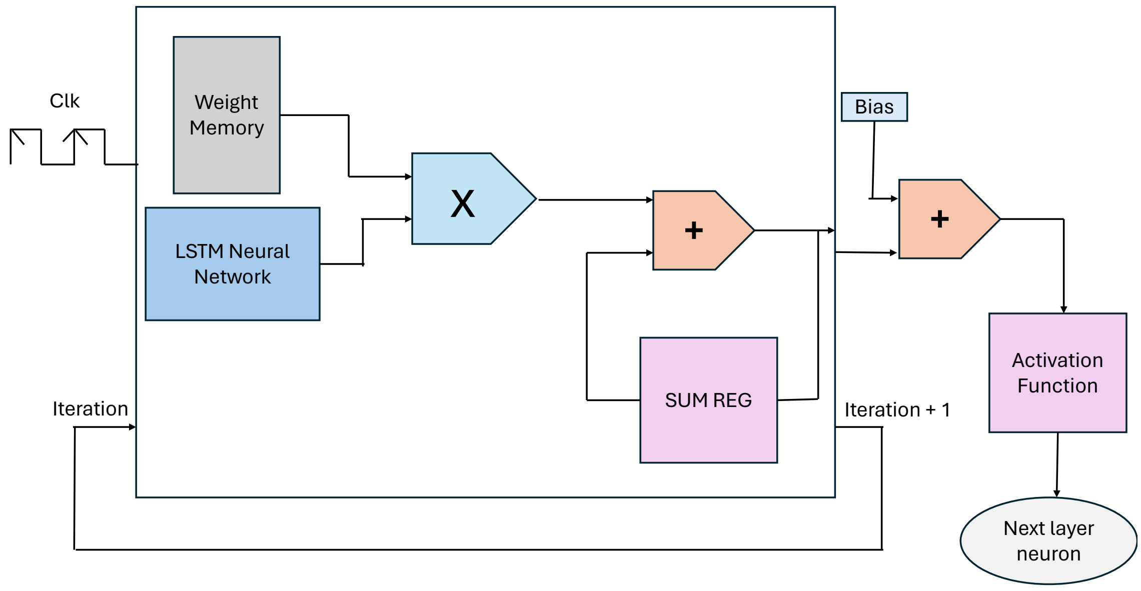


Figure 3.5: FCNN Neuron

Then the summation is performed for all the 784 values . After all the values are added then the bias are added before it is passed through activation function . The activation function is sigmoid function and introduces non linearity in the network . The output of the sigmoid function is fed to the next FCNN layer . The sigmoid function output is always in the range from 0 to 1 .

3.3 Masking

The main focus of this thesis was to follow the ModuloNet making from [DAP+21]. But implementing DOM (Domain oriented masking) and Modulonet on LSTM is not possible due to following problems .

- The sigmoid activation function cannot be masked with mod operation because the output of sigmoid lies between 0 and 1 . Since the ISE 14.7 doenot have a builtin exponential function therefore a look up table is created . Hence ModuloNet masking cannot be performed on Sigmoid function .
- The Report [DAP+21] on page 19 focuses on Masking the input pixels rather than weights . This approach creates a more overhead of components to construct the model .
- Modulo operation entails loss of information and sudden changes in representations due to overflows.
- Application of LSTM becomes challenging. Modulo operation shifts the negative tanh activations to the positive side . Thus, The tanh activation function cannot be masked with mod operation beacuse tanh output is in the range -1 to 1 . If we apply mod operation it becomes 0 to 1 which is equal to sigmoid function hence the neural network will not perform has expected .
- Modulo folding causes negative output scores of the last layer to wrap around. This results in wrong predictions to have high confidence scores .

Keeping this problems in mind we propose a different modified version of masking with the help of ModuloNet . This new approach has two parts of masking , the first part is the masking of weights in software before placing the weights in memory and the second part is the harware masking .

3.3.1 Software Masking

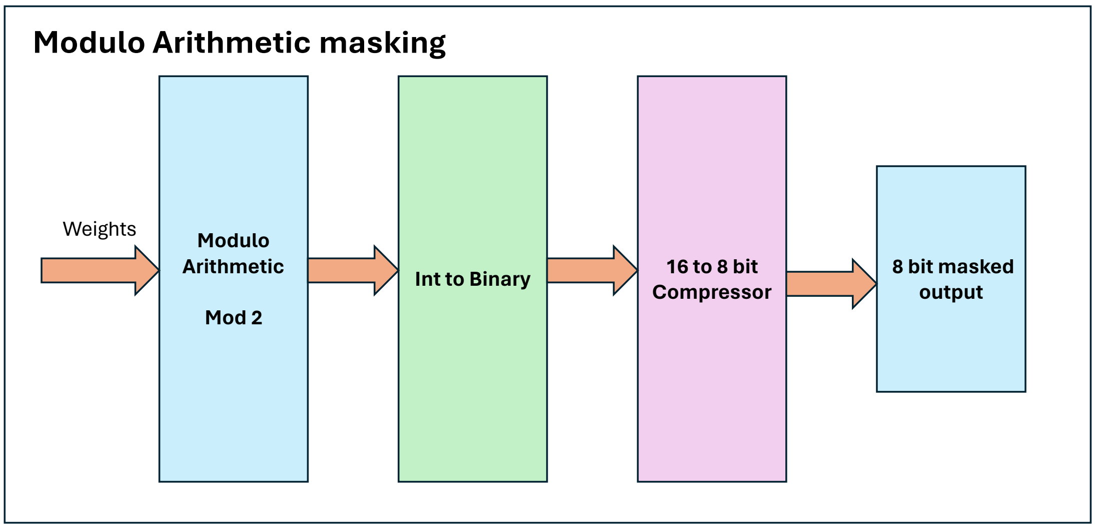


Figure 3.6: ModuloNet Masking

Here's the VHDL code of RAM0 which consists of 784 masked neuron0 weights :

```


  architecture syn of ram_0 is

  type ram_type is array (0 to 783) of std_logic_vector(7 downto 0);
  signal ram : ram_type := (
"00000000","00000000","00000000",......
"00000000","00000001","00000000","00000000","00000000","00000000");
begin

```

The implementation of ModuloNet offers a viable solution for successfully undertaking masking operations in neural networks such as LSTM models and harnesses the peculiarities of modulo arithmetic. The ModuloNet masking consist of three layer masking . In the above code the 16 bit binary number represent the value of the weights in the standard form (1st bit sign bit ,next 7 bit integer part and the last 8 bits represent fractional part).

In this approach the weights are passed to a Mod 2 block . The output of this block will be in the range 0 to 2 . All the negative values and the values greater than 2 will be changed to a value between 0 and 2 . The next block is the integer to binary converter , here the mod 2 output is converted to 16 bit binary number where first 8 bits represent integer part and the last 8 bits represent fractional part . Finally coming to the last part of ModuloNet , the 16 bit binary value is sent to compressor which produces an output of 8 bit binary value . Since the output of mod 2 can have an integer part has 0,1 or 2 , therefore we can use an 8 bit representation to represent the value by compressing a few bits .This binary values are used as weights in the $RAM0$ to $RAM9$ as shown in the above VHDL code . The Figure 3.6 represents the ModuloNet Masking , in which the weights are converted to a 8 bit value . This process is completely performed using python and the final masked output values are stored in memory .

3.3.2 Harware Masking

Once the weights are placed on to the memory the hardware consists of a random byte generator as shown in Figure 3.7 . This component generates random byte in each iteration which is concatinated with the masked weights . This concatinated value is divided into two independent shares . each independentshare has 4 bits from masked weights and 4 bits from random generator . This two share are computed independently with the LSTM output .

Then the computation of neuron begins as shown in Figure 3.5 on page 31 . Since we have two share , each share is computed seperately and the final value of both this shares is added before it is passed throught the activation function block . The output of the activation function is again provided as input to the next layer of the FCNN .

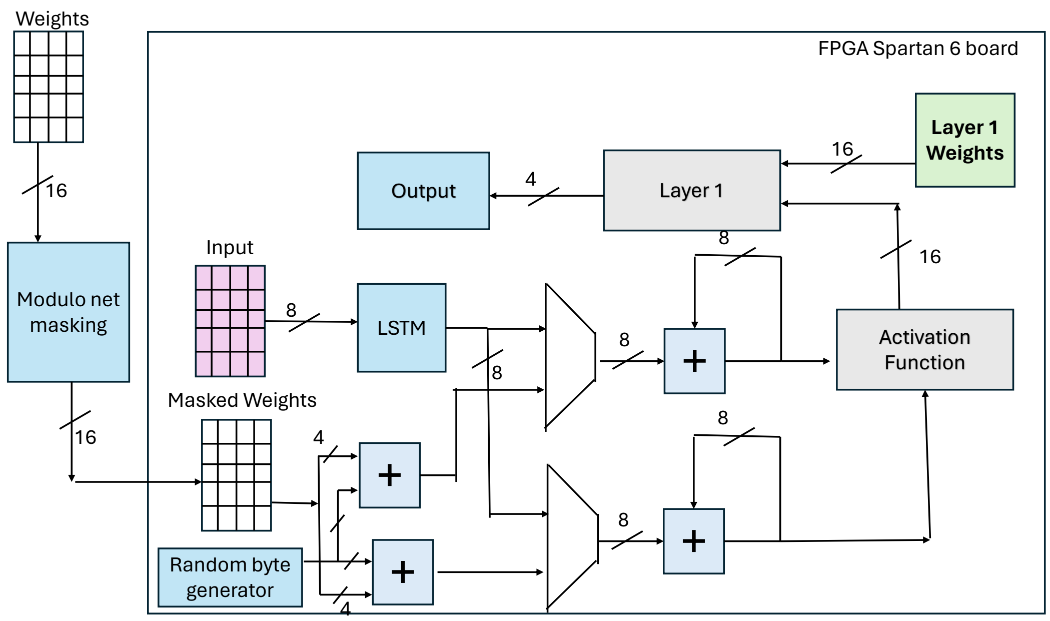


Figure 3.7: ModuloNet Masking in Harware

Here's the actual weights and the masked values for the actual weights :

```

    weights = [0.02091480791568756, 0.08141384273767471, 0.06094793230295181,
-0.07860653102397919, -0.07090934365987778, 1.0332908630371094, 1.980320930480957,
3.005425453186035, 3.060730457305908, 2.844209671020508]

    weights = ['00000000', '00000001', '00000000', '10000001', '10000001',
'00010000', '00000000', '00110000', '00110000', '00101101']


```

This approach has a major advantages to prevent the side channel attacks , since the actual weights are not present on the hardware , it is highly difficult to retrive the actual weights of the neural network . Even if the weights presented on the memory is retrived the attacker has to perform an aditional level of unmasking to retrive the actual weights and the process is highly difficult .

3.4 Output

The software masking is unmasked during summation , The difference between the weighted summation of the actual weights and the masked weights is calculated . For example let's assume 10 weights as shown in the code .

Here's the initial values provided to the summation register :

```vhdl
signal sum_reg_0 : STD_LOGIC_VECTOR(15 downto 0):= "1000000101010101";
signal sum_reg_1 : STD_LOGIC_VECTOR(15 downto 0):= "1000000100100011";
signal sum_reg_2 : STD_LOGIC_VECTOR(15 downto 0):= "1000000100001001";
signal sum_reg_3 : STD_LOGIC_VECTOR(15 downto 0):= "1000000110101010";
signal sum_reg_4 : STD_LOGIC_VECTOR(15 downto 0):= "1000000111011000";
signal sum_reg_5 : STD_LOGIC_VECTOR(15 downto 0):= "1000000011110000";
signal sum_reg_6 : STD_LOGIC_VECTOR(15 downto 0):= "1000001010000100";
signal sum_reg_7 : STD_LOGIC_VECTOR(15 downto 0):= "1000000100010011";
signal sum_reg_8 : STD_LOGIC_VECTOR(15 downto 0):= "1000001000001110";
signal sum_reg_9 : STD_LOGIC_VECTOR(15 downto 0):= "1000001011001111";
```

The summation of the first weight list results to 11.937738083302975 and the summation of the second weight list which is the masked weights is 9.75 . The difference is 2.1877380833029747 . This value is given to the summation register as an initial value .

The same logic works for 784 weights in each of the 10 neurons . The difference weights are calculated using python and provided as an initial value to the summation register . This initial value is difficult to obtain by side channel attack and the value cannot be calculated by the attacker since he has no knowledge about the actual weights .

The next step is to unmask the harware masking . This is done in the activation function block . When both the independent shares meet, the unmasking is done to remove the random byte included at the beginning of the computation . This results in preserving the accuracy of the model and increasing the Security of the neural network .

4 Software Implementation

LSTM neural nets in such software can be set up by taking certain important considerations that should help to construct, train and monitor the model correctly. The primary task to be performed here is to select a complete deep learning framework - TensorFlow, PyTorch, or Keras - which has a deep rooted aptitude to construct and train LSTM models. Once you select the framework you have to declare the number of LSTM layers along with the number of memory cells in every layer as well as the activation functions to be used. Furthermore, a tuning mechanism might be incorporated to control properties like dropout rate or recurrent dropout rate, thereby avoiding overfitting of the model during training.

Data preprocessing is the next thing that shall be done. LSTMs are especially fitted in handling sequential data. Hence, it's important to use the data in the correct format. This practically means transforming the inputs sequences into a fixed-length sequences, adding zeros on the shorter ones, and maybe having a normalization step during training to improve convergence. Processing done,the data is ready for training by LSTM with SGD or Adam using appropriate optimization algorithms. In training, the model undergoes a process of capturing sequential patterns inherent in the data and its parameters grounded them to a minimum loss value. The last stage is taking the trained model to the test which is done on a separate validation or test dataset for evaluating its performance before fine-tuning the hyper parameters as necessary. Specifically, the LSTM architecture modeling must provide an accurate thought to the architecture structure, data preprocessing and training methodologies so as to develop the robust models for sequential data analysis.

4.1 Using Tensorflow

TensorFlow is worth of praise as one of the top 3 deep learning frameworks and all the community appreciates things like its high scalability, modularity, flexibility and community support. TensorFlow has been introduced by Google Brain and it is a tool used by AI researchers and experts to construct, train and deploy models of machine learning algorithms which are useful in domains such as vision computers, language processing, etc. In this architecture, symbols represent the graph of computer topology, where nodes depict the mathematical operation and the edge imply the diffusion of tensors, or multidimensional arrays. This layout not just avails developers to execute codes on CPUs or GPUs but also supports distributed computing across numerous devices, hence it is the choicest one for either development or deployment in production.

To begin with, TensorFlow provides a user-friendly interface to build and train neural networks with little amount of code . In the code you can observe that the MNIST dataset is made sequential and reshaped to 28 x 28 before passing it to the LSTM layer . The number 64 in the line model . add (tf . keras . layers . LSTM(64)) represents number of unit cells present in each LSTM neuron . Then the output is passed through the fully connected layers . The number of layers and the activation function can be changed according to the neural network requirements .

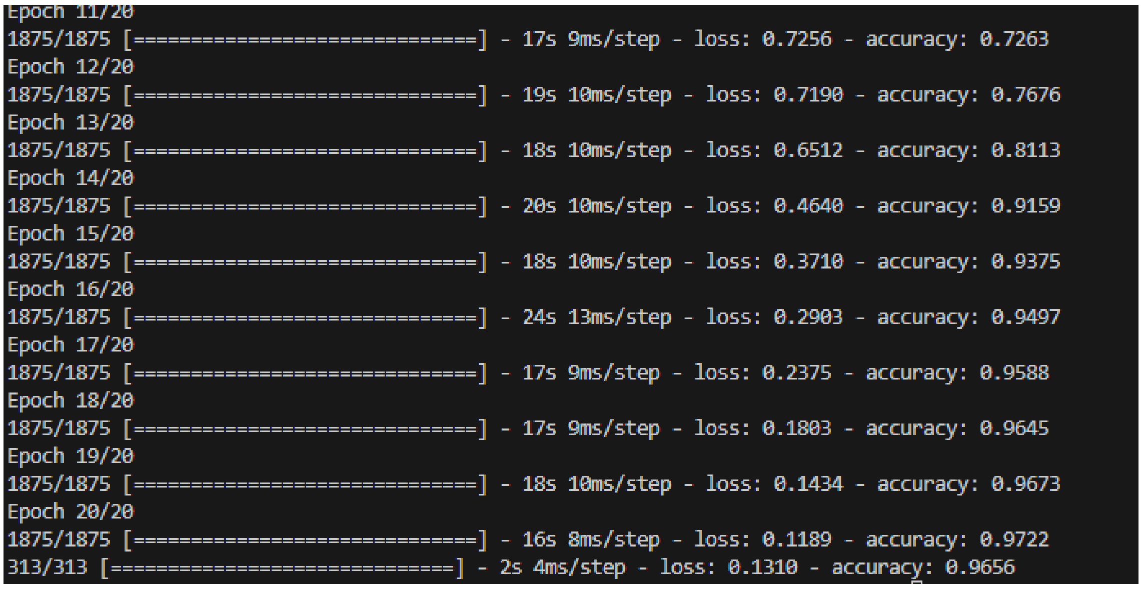
```
import  tensorflow  as  tf
import  json

mnist  =  tf . keras .  datasets  . mnist
( x_train ,   y_train ) ,  ( x_test ,   y_test )  =  mnist . load_data  ()
 x_train   =  tf . keras .  utils  . normalize ( x_train ,   axis =1)
 x_test   =  tf . keras .  utils  . normalize ( x_test ,   axis =1)

model  =  tf . keras . models .  Sequential  ()
model . add ( tf . keras .  layers  . Reshape ((28,   28) ,   input_shape  =(28,   28) ) )
model . add ( tf . keras .  layers  .LSTM(64))  #  Adding  an  LSTM  layer
model . add ( tf . keras .  layers  .Dense (30,    activation  = tf . nn . sigmoid ) )
model . add ( tf . keras .  layers  .Dense (30,    activation  = tf . nn . sigmoid ) )
model . add ( tf . keras .  layers  .Dense (10,    activation  = tf . nn . sigmoid ) )
model . add ( tf . keras .  layers  .Dense (10,    activation  = tf . nn . sigmoid ) )

model . compile ( optimizer = 'adam' ,
                    loss = '    sparse_categorical_crossentropy     ' ,
                    metrics =[ ' accuracy ' ])
model .  fit  ( x_train ,    y_train ,   epochs =20)
( val_loss  ,    val_accuracy  )  =  model .  evaluate  ( x_test ,    y_test  )

 weightList   =  []
 biasList   =  []
for  i  in  range  (1,   len ( model .  layers  ) ) :
       weights   =  model .  layers  [ i ].  get_weights  () [0]
        weightList  . append (( weights  . T) .  tolist  () )
       biases   =  model .  layers  [ i ].  get_weights  () [1]
        biases_list    =  biases .  tolist  ()   #  Convert  NumPy array  to  Python   list
        biasList  . append (  biases_list   )

data   =  { " weights " :   weightList  ,   " biases  " :   biasList  }
with  open ( '  weightsandbiases  . txt ' ,   "w")  as  f :
      json . dump( data ,   f )

#  Assuming  you  have  saved  the  model
model . save ( 'my_model' )
```

Listing 4.1 Python code Using Tensorflow to Implement LSTM and obtain the weights .

The accuracy obtained from the Tensorflow code is 96.5 %

Then using adam optimiser together compile the model , here epochs = 20 means the model runs for 20 iterations and after each iteration the model backtraces and updates the weights and biases to get higher accuracy . This weights and biases are stored in a text file for further usage .

The output of this iterartions can be seen in Figure 4.1 . The tensorflow code updates the values of weights and biases to increase the accuracy and reduce the loss of the model . The output also provide the time required for each iteration to complete . The value 1875 denotes that the model is tested against 1875 testing dataset in each iteration and predicts the output . Depending on the output the accuracy is printed .



```
Epoch 11/20
1875/1875 [==============================] - 17s 9ms/step - loss: 0.7256 - accuracy: 0.7263
Epoch 12/20
1875/1875 [==============================] - 19s 10ms/step - loss: 0.7190 - accuracy: 0.7676
Epoch 13/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.6512 - accuracy: 0.8113
Epoch 14/20
1875/1875 [==============================] - 20s 10ms/step - loss: 0.4640 - accuracy: 0.9159
Epoch 15/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.3710 - accuracy: 0.9375
Epoch 16/20
1875/1875 [==============================] - 24s 13ms/step - loss: 0.2903 - accuracy: 0.9497
Epoch 17/20
1875/1875 [==============================] - 17s 9ms/step - loss: 0.2375 - accuracy: 0.9588
Epoch 18/20
1875/1875 [==============================] - 17s 9ms/step - loss: 0.1803 - accuracy: 0.9645
Epoch 19/20
1875/1875 [==============================] - 18s 10ms/step - loss: 0.1434 - accuracy: 0.9673
Epoch 20/20
1875/1875 [==============================] - 16s 8ms/step - loss: 0.1189 - accuracy: 0.9722
313/313 [==============================] - 2s 4ms/step - loss: 0.1310 - accuracy: 0.9656
```

Figure 4.1: Long Short-Term Memory.

4.2 Unrolled Neural Network using Python

Implementing an unrolled neural network solely via the weight matrix and some simple flow control constructs like if and for loop. Let's outline a basic approach:

- In the first place, defining the structure of rolled out neural network explicitly with layers, number of neurons in each layer, and activation functions. In addition, initializing weights, for each neuron in each layer, and biases. The weights and biases used here are obtained from the tensorflow code .
- Secondly, take each of the layers in turns and multiply them at the time step with the sequence to be constructed of. Instead of this, at every layer you will carry out weighted summation of inputs from the previous layer (or from the input data if it is the first layer) using the weights and the biases. These can be visualized in the code .
- In the next step, I use the summation for obtaining the arithmetic sum. Finally, activate the sum with an activation function to give the network the non-linearity. The activation function is tanh function and sigmoid function .
- When calculating the output of the last layer, softmax function is used . The index of the neuron having maximum value is provided as output .

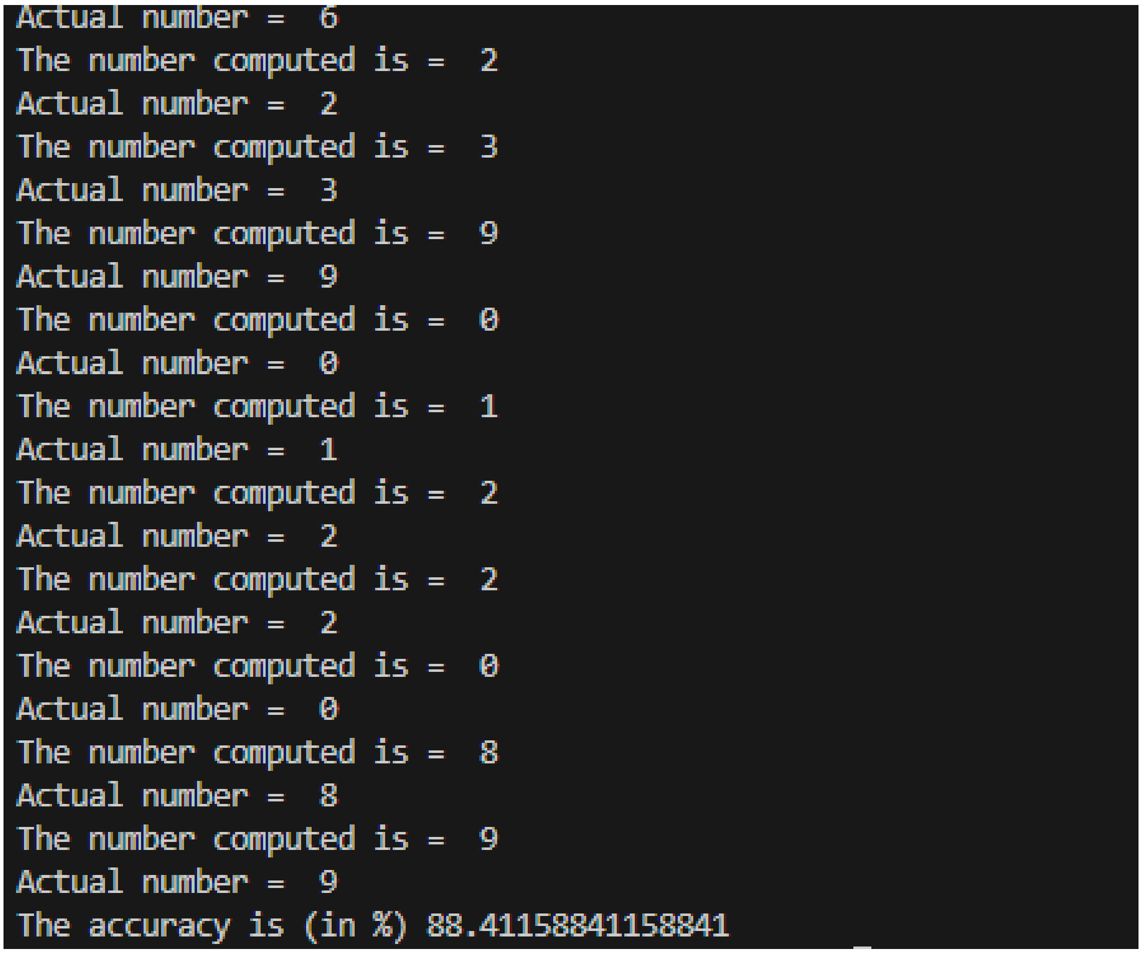
The output of the unrolled LSTM code is shown in Figure 4.2 . For each testing dataset the actual value is included in the list has 785 th element . Therefore it is easy to find that the computed value is equal to the actual value . The accuracy of this model is 88.41 % .

The code can be found in this link: https://github.com/Rithvikns/lstm_neural_network_on_fpga/blob/main/python.

```
    for  t  in  range (28) :

    wf= weights_array  [0]
    wi= weights_array  [1]
    wc= weights_array  [2]
    wo= weights_array  [3]


    for  i  in  range (28)  :

        y= lines [28∗t+i]

        x=( int (y ,2) /255)



        f_t  =  sigmoid ((wf∗ x)+(uf∗  h_t_minus)+bf)
        i_t  =  sigmoid ((wi∗ x)+(ui∗  h_t_minus)+bi)
        c_t  =  tanh ((wc∗x)+(uc∗  h_t_minus)+bc)
        o_t  =  sigmoid ((wo∗x)+(uo∗  h_t_minus)+bo)
        c_t_minus  =  c_t_minus  ∗  f_t  +  (i_t  ∗  c_t)
        h_t_minus  =  tanh (c_t_minus)∗o_t
        output_im . append (h_t_minus)
    output_v  =  [0  if  65  <  round (z  ∗  255)  <  75  else  1  for  z  in  output_im]


for  v  in  range  (0,10) :
    sum_1 = 0
    for  k  in  range  (0,784) :
        sum_1 +=  weights  [0][ v ][ k]∗ output_v [k]
     layer_1_bin  . append (sum_1)
    sum_1 +=  biases  [0][ v]
     layer_1  . append (sigmoid (sum_1))

for  z  in  range  (0,10) :
    sum_2 =0
    for  l  in  range  (0,10) :
        sum_2 = sum_2 +  weights  [1][ z ][ l]∗ layer_1  [l]
    sum_2 +=   biases  [1][ z]
     layer_2  . append (sigmoid (sum_2))
max_index  =  layer_2  . index (max( layer_2 ))
if  int (max_index)  ==  int ( lines  [784],  2):
    loss  =  loss
else :
    loss  =  loss  + 1
print ("The number computed  is  = ", max_index)
print (" Actual  number = ", int ( lines  [784],  2))
 digit_count   += 1

accuracy  =  ((  digit_count   −  loss )  /   digit_count  )∗100
print ("The  accuracy  is  (in  %)",  accuracy )
```

Listing 4.2 Unrolled Python code to Implement LSTM .

The accuracy obtained from the Unrolled LSTM code is 88.41 %



```
Actual number =  6
The number computed is =  2
Actual number =  2
The number computed is =  3
Actual number =  3
The number computed is =  9
Actual number =  9
The number computed is =  0
Actual number =  0
The number computed is =  1
Actual number =  1
The number computed is =  2
Actual number =  2
The number computed is =  2
Actual number =  2
The number computed is =  0
Actual number =  0
The number computed is =  8
Actual number =  8
The number computed is =  9
Actual number =  9
The accuracy is (in %) 88.41158841158841
```

Figure 4.2: Unrolled LSTM Output.

4.3 Comparison between different software Implementations

	Comaparison Table 1	
	Tensorflow Code	**Unrolled LSTM code**
Accuracy (%)	96.56	88.41
Loss	0.131	0.25
Training Time (minutes)	5.41	No Training time needed
Execution Time (minutes)	2.54	2.24
Flexibility	Flexible	Not Flexible

Table 4.1: Comparison of TensorFlow code and unrolled LSTM code with accuracy and other metrics.

The table makes clear the performance differences in the TensorFlow and unrolled LSTM codes for different metrics. With regard to accuracy, TensorFlow code has 96.56% while the unrolled LSTM code has 88.41%. This equality carries over to the loss values of the TensorFlow in the order of 0.131 is significantly lower than that of the LSTM code unrolled 0.25 loss. In particular, TensorFlow features a much shorter training time of 5.41 minutes, as opposed to unrolled LSTM code that does not require any training time by virtue of its internal architecture. Another important consideration is the execution time, where TensorFlow is slower with 2.54 minutes against 2.24 minutes of unrolled LSTM code. Apart from flexibility, TensorFlow provides greater flexibility in its implementations than the hardcoded LSTM code which tends to be rigid in structure.

The unrolled LSTM can be achived only with the help of tensorflow code , since it provides the weights and biases for the model . It is important to notice that the drop in accuracy whereas the code remains the same in both is due to the rounding off of the floating point weights and biases .

	Comaparison Table 2	
	2 Lstm units , 30,30,10,10 layers	**Single Lstm unit, 10,10 layers**
Accuracy (%)	88.41	84.41
Loss	0.25	0.29
Training Time (minutes)	No Training time	No Training time
Execution Time (minutes)	2.24	2.06
Number of weights and biases	24,900	7,960
Complexity	More Complex	Less Complex

Table 4.2: Comparison of different size LSTM neural networks with accuracy and other metrics.

The given table describes differences in performance of two variations of LSTM neural networks that have different configurations. The first Configuration including two LSTM units (with layers 30, 30, 10, 10) that show 88.41% accuracy and 0.25 loss value accordingly. On contrary, the second system that is made by a single LSTM unit with 10,10 layers the accuracy is a little lower 84.41% and a marginally higher loss of 0.29 respectively. Both the types need no time for training, which makes them suitable to be deployed anytime. In terms of duration, the initial setup takes 2.24 minutes, whereas the latter one involves slightly better execution time that completes the same process in only 2.06 minutes. Moreover, the first configuration has 24,900 weight and bias elements with only 7,960 in the second. Consequently, the first one is a lot more expressive.

This particular illustration demonstrates that the network architecture type has a direct effect on the metrics of performance which include accuracy, loss, execution time, and the model complexity. The one with two LSTM units and four-deep layers better depicts the high level of accuracy, however it is also much more expensive in terms of calculations due to the growth of the weights and biases. However, as opposed to the first case, and placing aside the architecture with the few layers and and a simple LSTM block, one loses a bit of the accuracy and the complexity but gains computational efficiency with the lower number of parameters. This is an implication of both model complexity and performance, that the right network architecture needs to be chosen based on the needs of an individual task, after considering resources, desired accuracy, and model interpretability.

5 Hardware Implementation

With the aid of this, now that we have learned about different Neural Network software implementation, let us use LSTM neural network with two unit cells coupled with 10,10 fully connected layers. You can find the VHDL code for this neural network at `https://github.com/Rithvikns/lstm`. Additionally, the code for another neural network variant can be accessed here: `https://github.com/Rithvikns/lstm_neural_network_on_fpga/tree/main/lstm`. In the technical portion of the class, we will carefully go through each step so that understanding it clearly becomes possible.

5.1 Computational Aspects

Contrary to that, rolled-out LSTM networks consist of several components at every time step such as calculating the input, forget and output gates and consequently changing the memory cell and neural network states. These operations are usually carried out in a standardized way through the use of matrix-matrix multiplications, element-wise operations, and non-linear activation functions. Input gates decide the flowing accuracy of information into the memory cell, forget gates are employed to determine the content of information to ejected from the cell state, and output gates regulate the information flow to the next tier or output.

In this combinational circuit each process needs it's own resources , Due to the limitaions of FPGS Spartan 6 board this code can be synthesised and simulated but cannot be mapped on to the board . You can see the simulation output in Figure 5.1 on page 45 , the output is 7 , all the other signals are input signal . the input signal provided to this Neural Network is 784 , but the spatran 6 has only 328 Number of bonded IOBs .

The main advantage of combinational circuits is that you do not have to wait for output signal . Due to it's nature the output is available from the start but the main disadvantage is that you have to provide huge number of resources to compute everything parallely . Because of this limitations we use parallelization , pipelining and memory .

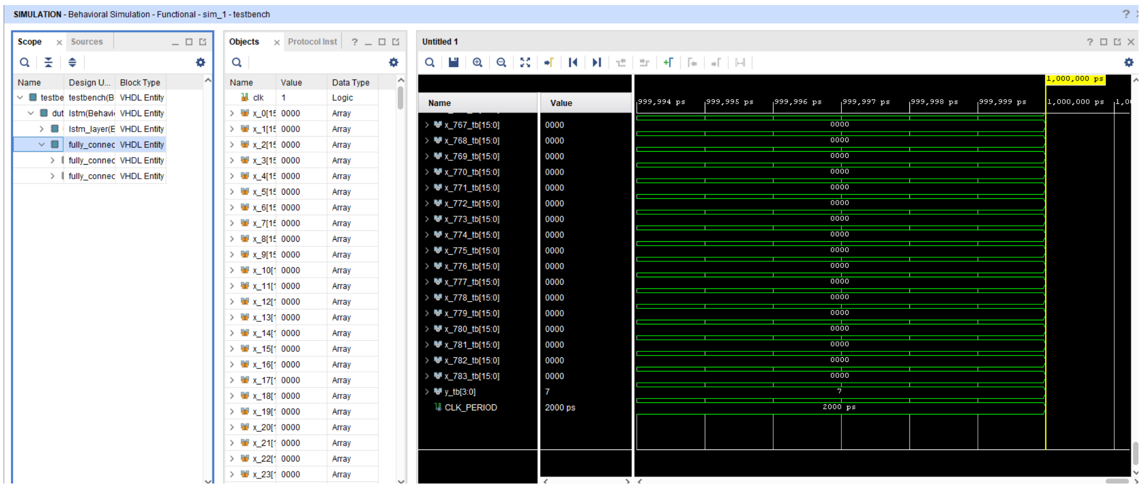
To make the mapping possible , the input data is stored in a single port RAM due to limited IOB . The code looks something similar to the sigmoid activation function using single port RAM code . But the major changes is the size of the memory array is 784 and each item is 8 bits long . The single Port RAM's have asynchronous read , that means the read operation is not dependent on clock only the write operation is dependent on clock .

you can see in the vhdl code for a single neuron computation you need 11 multiplications , 9 additions , 3 sigmoid and 2 tanh functions to run parallely . There are 20 signals wf1 to wf19 used to carry data between each gate and unit cell . The final memory elements are sent out of the module so that the next neuron uses this value for it's computation .

Here’s the VHDL code for a single neuron with two unit cells:

```vhdl
    ut1_nn_multiplication: nn_multiplication port map( x,wf , wf1);
    ut2_nn_multiplication: nn_multiplication port map( uf,h_t_minus_in , wf2);
    ut1_nn_addition: nn_addition port map( wf1,wf2 , wf3);
    ut2_nn_addition: nn_addition port map( wf3,bf , wf4);
    ut1_sigmoid: sigmoid port map( to_integer(unsigned(wf4)),f_t);
    ut3_nn_multiplication: nn_multiplication port map( x,wi , wf5);
    ut4_nn_multiplication: nn_multiplication port map(ui,h_t_minus_in , wf6);
    ut3_nn_addition: nn_addition port map( wf5,wf6 , wf7);
    ut4_nn_addition: nn_addition port map( wf7,bi , wf8);
    ut2_sigmoid: sigmoid port map( to_integer(unsigned(wf8)),i_t);
    ut5_nn_multiplication: nn_multiplication port map( x,wc , wf9);
    ut6_nn_multiplication: nn_multiplication port map( uc,h_t_minus_in , wf10);
    ut5_nn_addition: nn_addition port map( wf9,wf10 , wf11);
    ut6_nn_addition: nn_addition port map( wf11,bc , wf12);
    ut1_tanh : tanh port map( to_integer(unsigned(wf12)),c_t);
    ut7_nn_multiplication: nn_multiplication port map( x,wo , wf13);
    ut8_nn_multiplication: nn_multiplication port map(uo,h_t_minus_in , wf14);
    ut7_nn_addition: nn_addition port map( wf13,wf14 , wf15);
    ut8_nn_addition: nn_addition port map( wf15,bo , wf16);
    ut3_sigmoid: sigmoid port map( to_integer(unsigned(wf16)),o_t);
    ut9_nn_multiplication: nn_multiplication port map( c_t_minus_in ,f_t , wf17);
    ut10_nn_multiplication: nn_multiplication port map( i_t , c_t , wf18);
    ut9_nn_addition: nn_addition port map( wf17,wf18 , c_t_minus_1);
    ut2_tanh: tanh port map(to_integer(unsigned(c_t_minus_1)),wf19);
    ut11_nn_multiplication: nn_multiplication port map( wf19 ,o_t , h_t_minus_1);
    y <= "0000000000000000" when ((unsigned(h_t_minus_1)> "0000000001000001") and
(unsigned(h_t_minus_1)< "0000000001010000")) else "0000000100000000" ;
    h_t_minus_out <=  h_t_minus_1;
    c_t_minus_out <=  c_t_minus_1;

```

since the resources required for a single neuron is 25 arithmetic circuits and there are 784 neurons to be executed concurrently . Therefore with a total of 19600 operations . This process is then followed by a 10,10 fully connected layer which has to multiply all this neuron output with the weights and do a summation of the result , then adding the bias and pass through sigmoid activation function to provide the result for next layer . The next layer neuron performs a similar operation and get the final result of 10 outputs . This is a red hot line coding where each output refers to each number 0 to 9 . Hence using softmax function we are able to get the exact output of the digit , this makes it hard for the FPGA board to accomidate the place .



```
SIMULATION - Behavioral Simulation - Functional - sim_1 - testbench

Scope   Sources
Name      Design U...    Block Type
testbe    testbench(B    VHDL Entity
  dut     lstm(Behavi    VHDL Entity
          lstm_layer(E   VHDL Entity
          fully_connec   VHDL Entity
          fully_connec   VHDL Entity
          fully_connec   VHDL Entity

Objects   Protocol Inst
Name      Value   Data Type
clk       1       Logic
x_0[15    0000    Array
x_1[15    0000    Array
x_2[15    0000    Array
x_3[15    0000    Array
x_4[15    0000    Array
x_5[15    0000    Array
x_6[15    0000    Array
x_7[15    0000    Array
x_8[15    0000    Array
x_9[15    0000    Array
x_10[1    0000    Array
x_11[1    0000    Array
x_12[1    0000    Array
x_13[1    0000    Array
x_14[1    0000    Array
x_15[1    0000    Array
x_16[1    0000    Array
x_17[1    0000    Array
x_18[1    0000    Array
x_19[1    0000    Array
x_20[1    0000    Array
x_21[1    0000    Array
x_22[1    0000    Array
x_23[1    0000    Array

Untitled 1
                                                                                                   1,000,000 ps
Name              Value     999,994 ps  999,995 ps  999,996 ps  999,997 ps  999,998 ps  999,999 ps  1,000,000 ps  1,0
x_767_tb[15:0]    0000                              0000
x_768_tb[15:0]    0000                              0000
x_769_tb[15:0]    0000                              0000
x_770_tb[15:0]    0000                              0000
x_771_tb[15:0]    0000                              0000
x_772_tb[15:0]    0000                              0000
x_773_tb[15:0]    0000                              0000
x_774_tb[15:0]    0000                              0000
x_775_tb[15:0]    0000                              0000
x_776_tb[15:0]    0000                              0000
x_777_tb[15:0]    0000                              0000
x_778_tb[15:0]    0000                              0000
x_779_tb[15:0]    0000                              0000
x_780_tb[15:0]    0000                              0000
x_781_tb[15:0]    0000                              0000
x_782_tb[15:0]    0000                              0000
x_783_tb[15:0]    0000                              0000
y_tb[3:0]         7                                 7
CLK_PERIOD        2000 ps                           2000 ps
```

Figure 5.1: LSTM Simulation Output.

Macro Statistics	
# RAMs	3240 65536x16-bit single-port distributed Read Only RAM
# Multipliers	15024 15x15-bit multiplier
# Adders/Subtractors	41208 16-bit adder 16-bit subtractor
# Comparators	15033 15-bit comparator greater 16-bit comparator greater
# Multiplexers	42512 16-bit 2-to-1 multiplexer 32-bit 2-to-1 multiplexer 4-bit 2-to-1 multiplexer
# Xors	28760 1-bit xor2

Table 5.1: Advanced HDL Synthesis Report Before parallelization,pipelining and memory .

5.2 Parallelization, Pipelining and using Memory

In order to minimize input-output delays on digital devices parallelization and pipelining of operations are employed. The principle underlying parallel execution is to run numerous calculations all at once using concurrently working hardware accelerators such as the GPUs and FPGAs. Pipelining can be defined as a process of breaking the operations into steps and mixing execution of successive phases so that the lack of time during the work progression is minimized. Utilizing the functionality of hardware resources can be achieved by meticulously scheduling the data flow and computation tasks, hence creating vast performance gains and highest efficiency.

Let's start with the parallel, pipelined, and memory process part 1. In this VHDL code, the input x is obtained from a `raminfr` module. This module has 784 inputs, each input is 8 bits long. The `addr` signal provided to this module is the address of which input is required at that time step. In the LSTM layer itself, This code combines the first fully connected layer with the LSTM layer. Therefore, the second `port map` corresponds to the weight of the fully connected layer. There are a total of 10 Single-port RAMs representing each neuron weights, i.e., 784×10 as discussed before. Each RAM has 784 items, and each item is 8 bits long to represent the weights. The same address signal is provided to the `ram_1`, `ram_2`, ..., `ram_9` module to get the exact weights of the corresponding input address.

Once you have the input x, the neuron process is run parallelly, and the memory elements are stored in `c_t_minus_out` and `h_t_minus_2`. Due to the neuron module being a concurrent module, the result y is obtained in the same time step. This result Y is the output of the LSTM layer. Now the fully connected layer process begins. The output y is multiplied with different weights from `ram_0` to `ram_9`.

Coming to the second part of the parallel, pipelined, and memory process VHDL code . The process is sensitive to the clock and reset . If the reset is high all the signals are set to zero . In this code we have to keep track of two signals counter and addr . The address signal is incremented and fed to the all the modules above making the process sequential . In each time stamp a single input is selected from 784 pixels . Then this input goes through the neuron module to obtain y . Then this y is multiplied with 10 different weights and added to 10 different sum registers .

In the code you can observe if the counter is less than 783 , i.e if the counter has not reached the end . the neuron input elements are assigned with the values of previous memory elements . At the end of this clock cycle the value gets updated therefore the new input gets the value of the stored memory . This part of the code acts as a D FF . The same logic works for the summation part where the product reg value is stored in sum ref for each time stamp .

Third part of the parallel, pipelined, and memory process VHDL is very simple the output of the sum reg is fed to the sigmoid module to get the activation function output . This module is run on every time stamp but the actual result is availbale only on the last time step since the summation would have completed for all the 784 values . If you are now thinking about the biases for each fully connected layer is missing . Then you are right . I have added biases as the initial value for sum reg

- `signal sum_reg_0 : STD_LOGIC_VECTOR(7 downto 0) := "10000001";`
- `signal sum_reg_1 : STD_LOGIC_VECTOR(7 downto 0) := "00000001";`
- `signal sum_reg_2 : STD_LOGIC_VECTOR(7 downto 0) := "00000001";`
- `signal sum_reg_3 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";`
- `signal sum_reg_4 : STD_LOGIC_VECTOR(7 downto 0) := "10000001";`
- `signal sum_reg_5 : STD_LOGIC_VECTOR(7 downto 0) := "00000000";`
- `signal sum_reg_6 : STD_LOGIC_VECTOR(7 downto 0) := "00000001";`

Here's the VHDL code for parallel , pipelined and memory process part 1:

```
    ut1_raminfr: raminfr port map(
      clk  => clk,
      we   => '0',
      a => addr,
      di =>  (others => '0'),
      do => x
  );

    ut1_ram_9: ram_9 port map(
          clk  => clk,
        we   => '0',
        a => addr,
        di =>  (others => '0'),
        do => wf9
    );

    ut1_neuron: neuron port map(
     clk => clk,
       x  => x,
       c_t_minus_in  => sig_c_t_minus_in,
       h_t_minus_in  => sig_h_t_minus_in,
       c_t_minus_out => sig_c_t_minus_out,
       h_t_minus_out => sig_h_t_minus_out,
       y => y
    );
ut9_f_c_mul: f_c_mul port map(
    inputx => y,
    inputy => wf9,
    output => bf9
    );
ut10_f_c_add: f_c_add port map(
      inputx => bf9,
      inputy => sum_reg_9,
      output => product_reg_9
      );


```

Here's the VHDL code for parallel , pipelined and memory process part 2:

```vhdl
    addr_counter : process (clk, reset)
    begin
        if reset = '1' then
            addr <= (others => '0');
            counter <= 0;
            sig_c_t_minus_in <= (others => '0');
            sig_h_t_minus_in <= (others => '0');
        elsif rising_edge(clk) then
                    if counter < 783 then
                        addr <= std_logic_vector(unsigned(addr) + 1);
                        sig_c_t_minus_in <= sig_c_t_minus_out;
                        sig_h_t_minus_in <= sig_h_t_minus_out;
                        sum_reg_0 <= product_reg_0 ;
                        sum_reg_1 <= product_reg_1 ;
                        sum_reg_2 <= product_reg_2 ;
                        sum_reg_3 <= product_reg_3 ;
                        sum_reg_4 <= product_reg_4 ;
                        sum_reg_5 <= product_reg_5 ;
                        sum_reg_6 <= product_reg_6 ;
                        sum_reg_7 <= product_reg_7 ;
                        sum_reg_8 <= product_reg_8 ;
                        sum_reg_9 <= product_reg_9 ;
                        counter <= counter + 1;
                    else
                        counter <= 0;
                    end if;
                end if;
    end process addr_counter;


```

- `signal sum_reg_7 : STD_LOGIC_VECTOR(7 downto 0) := "10000001";`
- `signal sum_reg_8 : STD_LOGIC_VECTOR(7 downto 0) := "10000001";`
- `signal sum_reg_9 : STD_LOGIC_VECTOR(7 downto 0) := "00000010";`

Coming to the last part the fully connected layer 1 , all the values are fed to this module . The module is similar to the neuron module whick work concurrently and the output is obtained at the time step 784 . The actual output is obtained before due to the majority of the inputs being zero at the end .

Here's the VHDL code for parallel , pipelined and memory process part 3:

```vhdl
  ut10_f_c_sig: f_c_sig port map(
    num => to_integer(unsigned(sum_reg_9)),
    y => y_9
    );
ut0_fully_connected_layer_1: fully_connected_layer_1 port map(
clk => clk,
x_0 => y_0 ,
x_1 => y_1 ,
x_2 => y_2 ,
x_3 => y_3 ,
x_4 => y_4 ,
x_5 => y_5 ,
x_6 => y_6 ,
x_7 => y_7 ,
x_8 => y_8 ,
x_9 => y_9 ,
y => y_out
    );
end Behavioral;


```

5.3 Optimization Techniques

Using these techniques the mapping of VHDL code was possible on to the spartan 6 FPGA board . The results are shown in the table 5.1 .

- **Algorithm Optimization:** Reducing the code from a concurrent logic to a sequential logic has made the model simple and efficient . The complexity is reduced and the number of components required for the model is also reduced significantly .
- **Parallelism:** The switchover can enhance efficiency by executing the operations in an interlocked way. This can be implemented by pipelining, multiprocessing, or executing parallel execution of independent tasks. By making the LSTM neuron and the fully connected layer work parallelly the effeciency of the neural network has incresed . If these two modules were made sequential then the result will be obtained at 8000 time step but due to this parallelization the result is obtained at 784 time step .
- **Resource Sharing:** Utilization of shared hardware resources for different application functions or other processes could be a source of better resource usage and performance if there were multiple functions or processes. These are common items which include sharing registers, multiplexers, adders, and other components among the chips. This has played a important part in converting the program from concurrent to sequential . Sharing the resources has made possible to implement this LSTM neural network on FPGA . In the VHDL complete code runs single input at once therefore all the resources are shared for each input .

- **State Machine Optimization:** Optimal state machines can be achieved at minimum-state count, eliminating transitions and simplifying state encoding, which results in more efficient designs at lower resource-utilization. To store the memory elements of the neuron and the summation values of the fully connected neuron, state machine played a major role . Using this the number of files for the Concurrent LSTM implemntation is 96 is reduced to 33 vhdl modules to implement the sequential LSTM network .
- **Code Refactoring:** Reworking the code to make it simple, moving towards modularity and superior maintainability. The process is based on dividing long complicated logics into smaller manageable partials, rules, and functions and according to tiered design concept. Combining the lstm layer and the fully connected layer has reduced the LUT and the memory elements . If both the codes were in different modules the result had to be sent to the other module to compute but with the help of code refactoring the code look simpler .
- **Area and Timing Optimization:** The design of the circuit should be broken down to a Space utilization decide and time performance optimize. This implies determining the critical paths, fine-tuning timing constraints and designing the digital circuit as the logic levels are no longer the constraining point. Using a clk and making the design synchronous played a major role in area utilization . The LSTM network is divided according to the space utilization and used RAM and LUT in an equal propotion .
- **Hierarchical Design:** Creating an elemental design that has hierarchical components defined to and supports scalability, reusability, and maintainability. Making the hierarchical design a simple and effective by reducing many layers when compared to the previous concurrent LSTM code . The simulation output is shown in Figure 5.2 , the result is obtained at time step 784 , 1568 ,...., 7840 . If the input for 10 images is provided at once in input memory . The Figure 5.3 and Figure 5.4 represents the output at time step 1568 and 2352 which are the multiples of 784 . the computed value is equal to the actual value in this case .

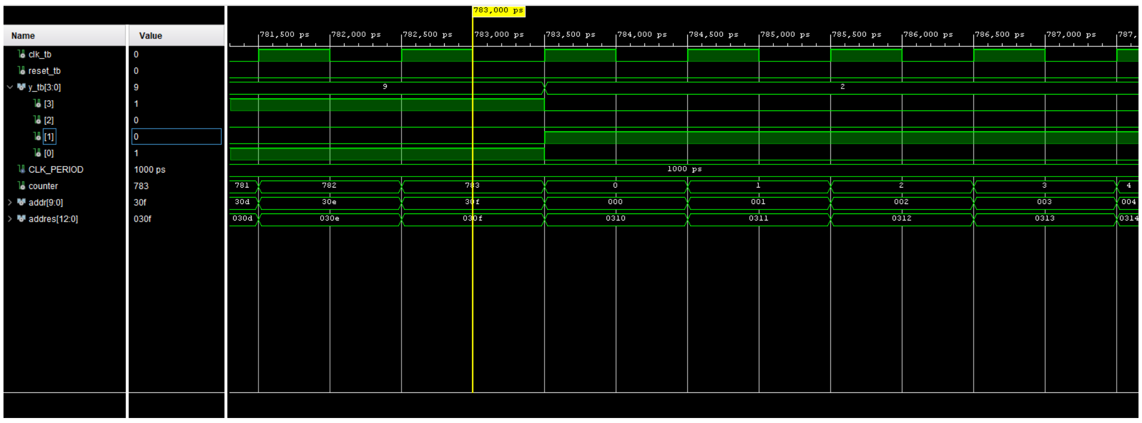


Figure 5.2: Sequential LSTM Simulation Output.

Comparing the number of adders from table 5.1 with table 5.2 . The numbers of adders drastically reduced from 41208 16 bit adders to 238 16 bit adders . you can see 99.5 % reduction . Similar to that number of multiplexers is reduced from 42512 to 5049 . you can see 85 % reduction . The number of comparators is reduced by half . The number of xor gate is reduced by 99.5 % i.e from 28760 to 240 .

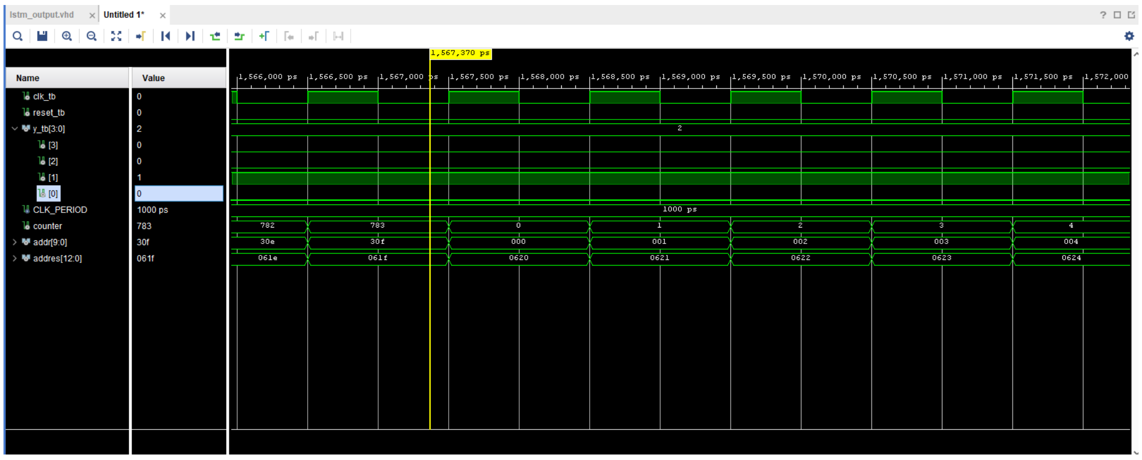


Figure 5.3: Sequential LSTM Simulation Output.

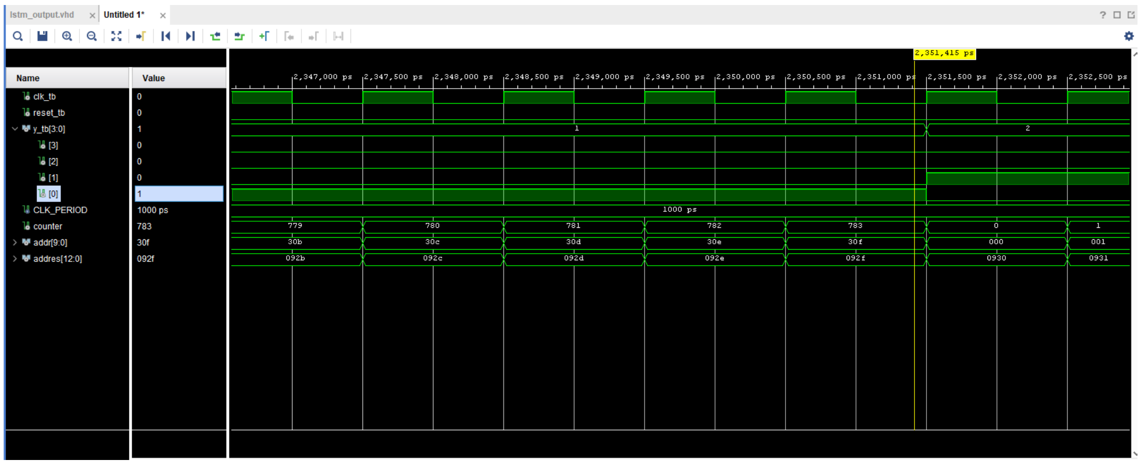


Figure 5.4: Sequential LSTM Simulation Output.

To understand the sequential LSTM module in detail let's look at Figure 5.2 . At each time step input X_i is obtained from the input RAM . Then this value is sent to the neuron , after the neuron computation the memory elements are stored and output y_i is sent to the multipliers . y_i is multiplied with the corrsponding weights from the RAM . Then the summation process takes place and store in a register file , this final value is sent to the sigmoid activation function to get the output values of layer 0 . Next step is to send this values to the next layer and compute the final value by using max function . Figure 5.3 represents the overall working off the sequential LSTM module . the output obtained at the time step 784 is the actual output . The LSTM neural network block in figure 5.3 refers to the figure 5.2 .

Here's the VHDL code of RAM0 which consists of 784 neuron0 weights :

```
    entity ram_0 is
    Port ( clk : in std_logic;
we : in std_logic;
a : in STD_LOGIC_vector(9 downto 0);
di : in std_logic_vector(15 downto 0);
do : out std_logic_vector(15 downto 0)
);
end ram_0;

architecture syn of ram_0 is

    type ram_type is array (0 to 783) of std_logic_vector(15 downto 0);
    signal ram : ram_type := (
"0000000000000101","0000000000010100","0000000000001111",......,"0000000010010101",
"1000000000001100","0000000000010101","0000000000000101","1000000000010010");
begin
process (clk)
begin
if (clk'event and clk = '1') then
if (we = '1') then
RAM(conv_integer(a)) <= di;
end if;
end if;
end process;
do <= RAM(conv_integer(a));
end syn;

```

Here's the VHDL code of RAM0 which consists of 784 masked neuron0 weights with random byte :

```

    architecture syn of ram_0 is

    type ram_type is array (0 to 783) of std_logic_vector(15 downto 0);
    signal ram : ram_type := (
"1010011000000000","1111100000000000","0011011000000000","0100100000000000",
"0001111100000000",...........,"0011110000000000","1011100100000000",
"0100110100000000");
begin

```

Advanced HDL Synthesis Report	
Macro Statistics	
# RAMs	21 100x16-bit single-port distributed RAM 784x8-bit single-port distributed RAM
# Multipliers	121 15x15-bit multiplier 7x7-bit multiplier
# Adders/Subtractors	238 16-bit adder 16-bit subtractor 8-bit adder 8-bit subtractor
# Counters	2 10-bit up counter 11-bit up counter
# Registers	112 Flip-Flops
# Comparators	7788 11-bit comparator greater 15-bit comparator greater 16-bit comparator greater 32-bit comparator greater 32-bit comparator lessequal 7-bit comparator greater
# Multiplexers	5049 1-bit 2-to-1 multiplexer 16-bit 2-to-1 multiplexer 17-bit 2-to-1 multiplexer 4-bit 2-to-1 multiplexer 8-bit 2-to-1 multiplexer 9-bit 2-to-1 multiplexer
# Xors	240 1-bit xor2

Table 5.2: Advanced HDL Synthesis Report after parallelization,pipelining and memory utilization

5.4 Implementation masked LSTM Network on Hardware

Implementing a masked LSTM (Long Short-Term Memory) architecture on hardware involves optimizing the existing hardware setup designed for unmasked LSTM networks with a key alteration: instead of 16 bit weights, a size reduction to 8 bits. This means that memory utilization goes down

Device Utilization Summary	
Selected Device	6slx75csg484-3
Slice Logic Utilization:	
Number of Slice Registers	140 out of 93296 (0%)
Number of Slice LUTs	32404 out of 46648 (69%)
Number used as Logic	26141 out of 46648 (56%)
Number used as Memory	6263 out of 11072 (56%)
Number used as RAM	6263
Slice Logic Distribution:	
Number of LUT Flip Flop pairs used	32433
Number with an unused Flip Flop	32293 out of 32433 (99%)
Number with an unused LUT	29 out of 32433 (0%)
Number of fully used LUT-FF pairs	111 out of 32433 (0%)
Number of unique control sets	3
IO Utilization:	
Number of IOs	6
Number of bonded IOBs	6 out of 328 (1%)
Specific Feature Utilization:	
Number of BUFG/BUFGCTRLs	1 out of 16 (6%)
Number of DSP48A1s	116 out of 132 (87%)

Table 5.3: Device utilization summary

by 50%, in the order of magnitudes, and thus, the masked LSTM becomes more memory efficient. Additionally, the lesser size of the reduced weight renders in the increase of the processing rate of such use case where time factor integrity is essential.

The principles at the heart of the masked LSTM remain equivalent to those of the unmasked LSTM, the only difference is the actual weights are converted and placed in the memory .Throught the use of identical architecture only with different precision levels on multiplication , addition and sigmoid . Due to similar architecture of implementing Masked LSTM it is easy to fit quickly without much adjustments in the hardware .

The use of masked LSTMs on hardware is accompanied by one very strong benefit which is the heightened security that it provides. The LSTM network traffic makes use of the masked weights as their actual weights, by saving the data in the memory. This helps to reduce the risk of side-channel attacks. This approach does not allow adversaries to break the memory reliability by extracting sensitive information, it makes the whole system firm. In fact, masked LSTM deployments not only lead in performance and speed but also put a lot on data security with embedded systems and secure communication networks being some of the many applications.

In case of the masked weights are obtained by side channel attacks it is difficult to retrieve the actual weight, due to the mod 2 block in masking. This ModuloNet Masking plays a major role in increasing the security of the LSTM Network. The unmasking of the weights is done in

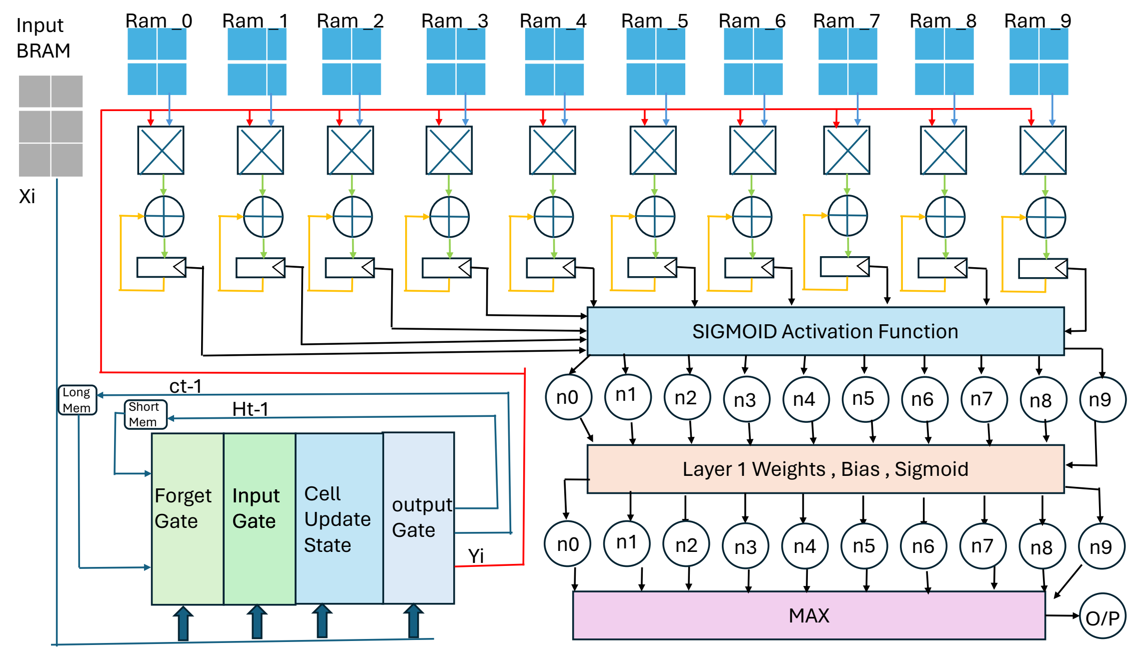


Figure 5.5: Hardware Implementation of LSTM .

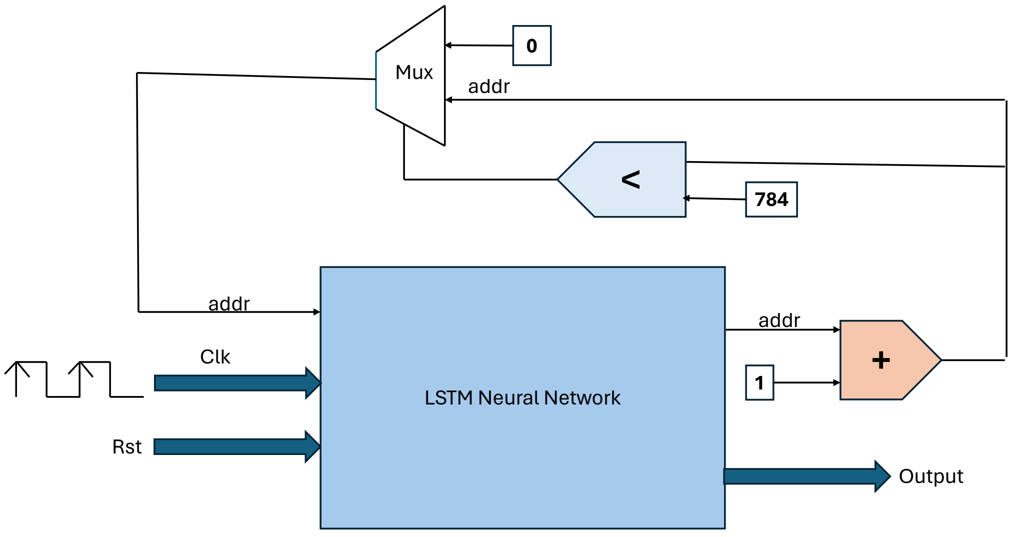


Figure 5.6: Overall Pictue of Hardware Implementation of LSTM .

different stages. The difference between the masked weights and the actual weights are stored in the initial value of sum reg, therefore when the summation is completed for a single neuron that is 784 multiplications and additions. The final value will be equal to the value of the unmasked architecture. Hence the circuit does not need any specific unmasking architecture to retrieve the actual weights. The difference value is calculated beforehand so that it can be stored in the register.

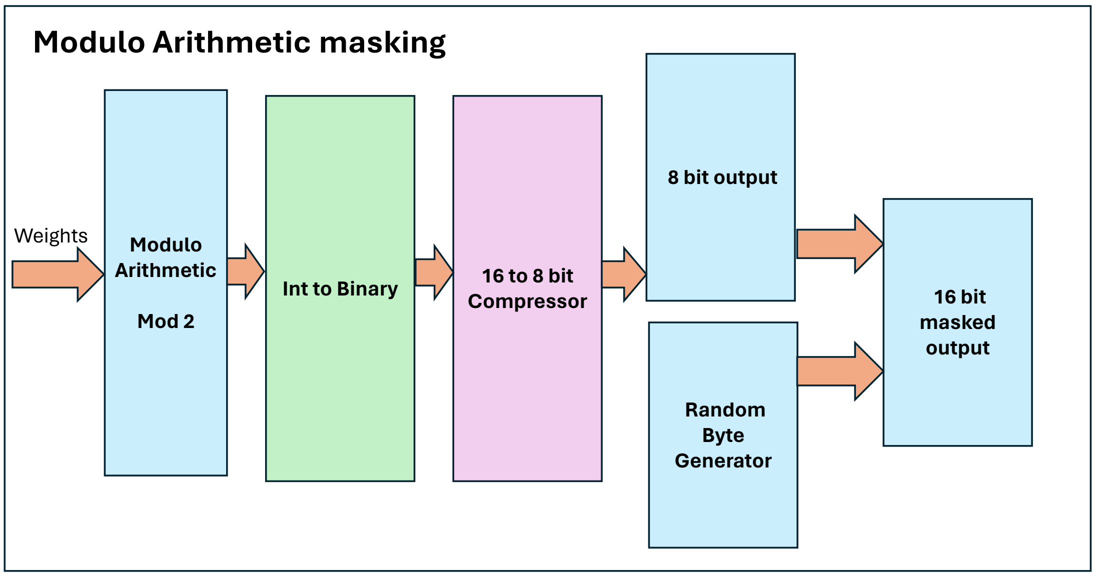


Figure 5.7: ModuloNet Masking Using Random Bits

6 Discussion

In designing a masked and unmasked LSTM network, the results show the balancing act of security issues versus performance problems towards resource constraints. Initially, the evaluation of the performance of both algorithms shows a slightly decreased accuracy of the LSTM network with a mask, along with the improvement of the model against side-channel attacks. It may convey that with additional changes of the model through masking, it has brought stronger defenses against attacks that target sensitive information like weights of the model. As the training and testing time remains same for masked LSTM network and unmasked LSTM network, the accuracy reduces due to compressing the weights to 8 bits in masking technique. Now, despite the fact that minor difference in accuracy,the additional security measures don't seem to hinder the general efficiency of the network appreciably.

6.1 Analysis of Results

Metric	Unmasked LSTM	Masked LSTM
Accuracy	84%	77%
Training Time	10 min	10 min
Testing Time	784 ns	784 ns
Robustness	Vulnerable to side-channel attacks	Improved security against side-channel att
Resource Consumption	High	High
Resistance to Adversarial Attacks	Medium	Medium
Energy Efficiency	Medium	Medium
Generalization	Good	Good
Scalability	Good	Good
Interpretability	Good	Good
Deployment Overhead	Low	Medium
Training Convergence	Stable	Stable

Table 6.1: Comparison of Unmasked and Masked LSTM Networks

The Table 6.1 shows the difference in performance of unmasked and masked LSTM networks through several metrics. Natively, unmasked LSTM acquires an accuracy of 84%, which beats masked LSTM, which has a accuracy of 77%. Both configurations facilitate the same amount of training and test times with training and testing taking 10 minutes and 784 nanoseconds respectively. However, masked LSTM possesses the capability to secure the system against side-channel attacks better as compared to unmasked LSTM. This improved security is vital in cases when the data confidentiality and private information are in jeopardy like the financial transactions and data processing of sensitive information. Through these security enhancements we see the same levels of defense against adversarial attacks and energy efficiency be demonstrated in both masking and non-masking configurations, hinting strongly that the masking approach does not have a significant influence on these characteristics.

Both the unmasked and masked LSTM networks demonstrate similar properties in terms of generalization, scalability, interpretability and training convergence. These moral traits act as a foundation for the actual use and function of LSTM neural networks across a wide range of areas and domains. The stable training convergence provides with reliable model training course, while the good generalization, scalability and interpretability provide with the necessary adaptability and usability of both configurations no matter what kind of dataset it is and what the case. Furthermore, the provision of low deployment overhead in Unmasked LSTM suggests light calculations and resource usage during the deployment stage and makes the integration into current systems or platforms easier and more effective. Medium deployment overhead in masked LSTM suggests slightly more calculations and resource usage during the deployment stage and makes the integration into current systems or platforms more complex . In sum, the comparison table depicts the equipment adjustments that should be applied between unmasked and masked LSTM architectures to choose the optimal one for the task.

Metric	Software Masked	Unmasked LSTM	Masked LSTM	Max Resources
Number of Slice Registers	147 (1%)	147 (1%)	147 (1%)	93,296
Number of Slice LUTs	31,943 (68%)	38,941 (83%)	38,941 (83%)	46,648
Number used as logic	23,349 (50%)	28,601 (61%)	28,601 (61%)	46,648
Number used as Memory	6,253 (56%)	10,340 (93%)	10,340 (93%)	11,072
Number of bonded IOBs	6 (1%)	6 (1%)	6 (1%)	328
Number of BUFG/BUFGMUXs	1 (6%)	1 (6%)	6 (1%)	16
Number of DSP48A1s	116 (54%)	116 (54%)	6 (1%)	132

Table 6.2: Comparison of Unmasked and Masked LSTM Networks with Device utilization metrics

The table 6.2 shows how well masked and unmasked LSTM networks perform in terms of device utilization metrics. It discovers the resource utilization and efficiency of each network A noticeable point is the employment of slice registers and LUTs. With almost the same amount of 147 slice registers, both masked and unmasked LSTMs have a similar amount of overhead in terms of register allocation. On the other hand, the masked LSTM shows a similar employing of slice LUTs (31,943 out of 46,648) against the unmasked LSTM (38,941 out of 46,648). The fact that the masking approach does not significantly decrease the use of slice LUTs, which are responsible for the realization of logic functions inside the FPGA, reveals that.

Similarly, the analysis of the use of logic brings in interesting facts. The masked LSTM uses up 23349 slice LUTs for logic functions which is nearly 50% of the available resources, while the unmasked LSTM uses a higher one which is 61%. The variation shows that the LSTM without the mask needs perhaps more complex logic operations, hence the higher demand of the LUTs. Also, the memory consumption in each case is very different. The masked LSTM uses 6,253 memory resources (56%) in comparison with the unmasked LSTM which uses substantially more (10,340 or 93%). This difference demonstrates that the unmasked LSTM might depend more on its memory resources in order to store computational intermediates or model parameters.

The last comparison of the commissioning's feature set that includes such a block as bonded IOBs, BUFG/BUFGMUXs, and DSP48A1 tells more about the interface and capability of computation processors in the network. Not only the both configurations but also the demonstration of identical utilization of bonded IOBs and BUFG/BUFGMUXs'lower numbers' indice defenite needs to derive from the clock management and input/output interfacing. On the other hand, the masked LSTM comes on top using 116 out of 132 DSP48A1s that counts to 54%, while the unmasked LSTM also utilizes the 116 DSP48A1s (54%). This lead us to an understanding that like memories, the LSTMs have certain parallel matrices of weights which are used in operations such as multiplications and accumulations, the most common operations in LSTMs. The table summarizes in detail the discrepant resource utilization as well as features of the devices between the masked and unmasked LSTM configurations, which enables the decision-making and optimal FPGA design.

6.2 Comparison with Existing Solutions

The contrast with existing solutions represents a detour from the traditional mode where the primary lens applied is using the masking of input pixels, activation function, output layer, comparator , multiplexer to safeguard the model. The offered solution suggests something novel,performing the masking in two stages . Software masking the weights and hardware masking introducing random byte , rather than the input pixels and different components , are masked out in the actual process.

The suggested method directly applies masking to the weights. The secret in our model are the weights of the neural network model. Through directly masking the weights solves these challenges in such a way the overhead of the resources is decreased when compared to masking performed in the paper[DAP+21] which in turn leads to the enhanced practical feasibility to secure machine learning model.

The proposed method provides a better protection in a new way by shifting the focus of security from input pixels to model weights. This creative solution goes beyond solving conventional security issues and additionally leads to the realization of optimal resource management and efficient secure model implementation. By shaking the norms and using the direct way through which data is masked, the proposed solution comes in handy in not only giving users enhanced security but also increasing the machine learning systems' robustness in various applications across different domains.

6.3 Trade-offs between Security and Performance

Unmasked and masked LSTM network comparison reveals that there is a constant room for improvement in terms of both security and efficiency across all metrics. However, the unmasked LSTM excels by providing higher accuracy of 84% rather than the masked LSTM's 77% which provides more side-channel attack resistance. This raised security is so critical because it falls within the brackets of protecting confidential information like financial transactions but at the same time, there is some sacrifice in accuracy. Last but not the least, both architectures exhibit different levels of robustness against adversarial attacks and energy efficiency, suggesting that Differential

Privacy techniques hardly cause long-term damage in these areas of model performance. The trade-off is the fact that hidden LSTM may be more effective when it comes to security, but it is at the cost of accuracy.

From the standpoint of resource consumption, both systems will differ only slightly, though will allocate their resources the same way in general. With fewer items per slice for the masked LSTM as compared to the unmasked LSTM, the implementation can facilitate better resource management, including fewer slice Look Up Tables (LUTs) and slice registers. Nevertheless, the LSTM that is not hidden uses more memory, the trade-off between memory and other sources is reached. These configurations display the similar low issues with deploying and the stable convergence during the training stage which indicates that they can be implemented in real-world without problems. In the end, this comparison of the device resource utilization metrics gives us the answer for the resource needs of each configuration and emphasizes the oppositions from the security and resource consumption.

7 Conclusion and Future Work

The tradeoffs concerning the safety and efficiency across different metrics are obvious as these phenomena occur in the comparison of unmasked and masked LSTM networks. In comparison, the unmasked LSTM outperforms the masked LSTM in accuracy with 84% against 77%, but the former supplies any protection against side-channel attacks. This better level of security is the basis on which we can rely to process our transactions safely and effectively even though the deed may not be 100% accurate. Of which two configurations have almost the same level of resistance to adversarial attacks and energy efficiency, the masking approach is proven to not greatly decrease these elements. Thus the bargain indicates that theming the LSTM in a disguise may increase security but imposes the accuracy sacrifice.

Talking about the resource design,both examples display similar habits but the way resources are used differs slightly. The particle LSTM saves slice LUTs and slice registers by slicing and therefore can more likely use the resources in a more effective way. But the memory-intensive desguized LSTM entails the tradeoff of this memory usage with other used resources though. Even though the two configurations have notable distinctions in terms of no deployment overhead and stability, they still efficiently train, underscoring their value for live applications and utilization. In the end, the analysis of metrics of gadget use gives a glimpse into the resource expenses of every choice and it shows that there is a difference between resource consumption and safety levels.

7.1 Summary of Findings

The comparison of masked and unmasked LSTM networks was very informative about how including security into machine learning models could impact on them in different ways. In the scope of multiple performance parameters such as precision, training time, and testing time, LSTM shows its genuine ability, where it competes on the level of precision as well as training time and testing time with the masked version. On the one word, Masked LSTM wards off side-channel attacks that represent a critical issues for the fields that require utmost security of data confidentiality and privacy. The trade-off between accuracy and security, which perfectly highlights the need for cautious consideration of machine learning models in their design and implementation, brings with it the need to balance performance requirements against the security concerns.

And on top of the things mentioned, the comparison brings into focus generalization, scalability, interpretability, deployment overhead, and training convergence in both of the settings. On the other hand, the unmasked and masked LSTM systems show outstanding training convergence, highly efficient generalization abilities, and low execution cost, which implies their effectiveness in real life usage for many different fields. On the other hand, the adversarial attack resistance and energy consumption remain at the same level, making the masking approach no more significant for this two aspects of model performance. Furthermore, it illustrates LSTM networks to be highly adaptable and capable of controlling as well as handling security issues for different machine learning jobs.

By the end of this analysis, we can understand that, of the unmasked and masked LSTM networks, the trade-offs ensue in the performance and the security of the machine learning models creation. The unmasked LSTM shows accuracy as the best but the masked LSTM provides enhanced security functions that underscores the necessity of combining security with performance needs. It can be seen that both architectures have identical features, such as generalization, scalability, and interpretability, denoting that long short-term memory (LSTM) networks are a successful tool for many different tasks. These discoveries broaden the picture of the peril of including protective characteristics in machine learning models. It serves to direct the building of fault-tolerant and secure algorithms in real life.

7.2 Contributions to Field

Comparing the performances of unmasked and masked LSTM models in terms of machine learning security creates a new area in the field of machine learning security that discloses different security approaches that may be implemented in the neural network architectures. Through the systematic evaluation of the above configurations with numerous marks, this study provides significant findings concerning the security-performance trade-offs. In this regard, the result that the collaborative LSTM brings improved security of side-channel attacks without losing accuracy sums up the fact that security aspects should be taken into account in the process of model design. This makes the security picture in machine learning has to be available for a more complete understanding and explain the importance of robust security measures in neural network structures.

Besides, the illustration indicates it is not a big deal to combine security features with machine learning models without hurting their effectiveness at all. Whilst the accuracy of the masked LSTM configuration shows a slight drop compared to the original unmasked configuration, both configurations determine a similar level of resistance to adversarial attacks, energy efficiency and generalization. Consequently, this supports the implication that security features could be effectively integrated without impacting much on the entire neural network performance. Such appreciation is forever critical to the researchers and practitioners involved in the development of safe and stable machine learning systems for the intended usage.

All in all, the introduction of this study contributes by demonstrating the practicality and efficiency as well as the trade-offs associated with a number of security measures in neural network architectures. Security considerations pose a common challenge while designing the machine learning systems. Through exploring various techniques such as masking algorithms, we inform and guide the development and deployment of secure machine learning systems. Moreover, the result add to ongoing process of the robust and trustworthy AI technologies thus providing a platform for a safe deployment in different sectors that ultimately advance the state-of-the-art in machine learning security.

7.3 Recommendations for Future Research

The machine learning security research of tomorrow which is based on the outcomes of this study can gain the traction to launch further comprehensive investigations of those challenges and opportunities. A further possible line of investigation suggests the use of advanced masking processes with high accuracy and at the same time ensuring the security. When designing side-channel-secure LSTM architectures, designing alternative masking methods or combinations of different techniques may be the appropriate measure to achieve the accuracy loss objection observed in the secret key masked LSTM configurations while still being side-channel resistant. Also, studying the implication of divergent types of threats, for example, model inversion or membership inference, on both masked and unmasked LSTM network will provide further deeper understanding of their security features and will be useful in designing more resilient models.

Moreover, future research efforts could target the formation of security tools as well as methodologies for the analysis of machine learning model security issues. Implementing suitable benchmarks and metrics for evaluating the reliability as well as the cyber security of neural network designs will be vital in allowing more comparisons between systems and ultimately accelerate advancements in the field. One other dimension that should be explored is the scalability and efficiency of security enhanced machine learning models (ML) in a real-world deployment scenarios, which will help to bridge the gap between the theoretical discoveries and practical AI applications. Assessment of security–performance–resource utilization tradeoffs in practical deployment situations is the step that will lead the way in evolving of safe and resource efficient machine learning systems.

In fact, the interdisciplinary nature of machine learning security calls for cooperation between fields of computer science, cybersecurity, and privacy. Drawing on tools and modalities from these disciplines might permit more wide ranging ways to respond to security challenges of machine learning. Moreover, consideration of ethical and societal consequences including privacy and fairness matters regarding security-enhanced machine learning models should be imperative for proper development and application of AI technologies. Answering these research directions can help to enhance the security of machine learning and provide a reliable and resilient AI system.

Bibliography

[00] IEEE Standard VHDL Language Reference Manual.IEEE Std 1076, 2000 Edition URL: https://edg.uchicago.edu/~tang/VHDLref.pdf

[10a] Spartan-6 FPGA Configurable Logic Block. User Guide. Xilinx, Advanced Micro Devices, Inc. February 23, 2010. URL: https://docs.xilinx.com/v/u/en-US/ug384 (cit. on p. 15).

[10b] Virtex-6 FPGA Routing Optimization Design Techniques. Recommendations for Improving Congested Designs. Xilinx, Advanced Micro Devices, Inc. October 28, 2010. URL: https://docs.xilinx.com/v/u/en-US/wp381_V6_Routing_Optimization (cit. on p. 96).

[11a] ISE In-Depth Tutorial. User Guide. Xilinx, Advanced Micro Devices, Inc. October 19, 2011. URL: https://www.xilinx.com/htmldocs/xilinx13_3/ise_tutorial_ug695.pdf (cit. on p. 16).

[11b] Spartan-6 Family Overview. Product Specification. Xilinx, Advanced Micro Devices, Inc. 2011. URL: https://docs.xilinx.com/v/u/en-US/ds160 (cit. on pp. 16, 102).

[11c] Spartan-6 FPGA Block RAM Resources. User Guide. Xilinx, Advanced Micro Devices, Inc. July 8, 2011. URL: https://docs.xilinx.com/v/u/en-US/ug383 (cit. on p. 86).

[20] *MNIST - Machine Learning Datasets*. 2020. Available at: https://datasets.activeloop.ai/mnist. Accessed: 2024-02-26.

[22a] UltraFast Design Methodology Timing Closure Quick Reference Guide. Reducing Congestion. Xilinx, Advanced Micro Devices, Inc. November 30, 2022. URL: https://docs.xilinx.com/v/u/en-US/ug1292-ultrafast-timing-closurequick-reference (cit. on p. 96).

[22b] Vivado Design Suite Tutorial. Logic Simulation. Xilinx, Advanced Micro Devices, Inc. December 23, 2022. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug937-vivado-design-suite-simulation-tutorial.pdf (cit. on p. 16).

[22c] Vivado Design Suite User Guide. Using the Vivado IDE. Xilinx, Advanced Micro Devices, Inc. April 27, 2022. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug893-vivado-ide.pdf (cit. on p. 16).

[30] UltraFast Design Methodology Guide for FPGAs and SoCs. Addressing Congestion. Xilinx, Advanced Micro Devices, Inc. 2022-11-30. URL: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology/Addressing-Congestion (cit. on p. 96).

[B21] Bischoff, P. *What is a Side Channel Attack? (with Examples)*. Article on Comparitech, 2021. Available at: https://www.comparitech.com/blog/information-security/what-is-a-side-channel-attack/.

[BFRV92] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic. *Field-programmable gate arrays*. Vol. 180. Springer Science & Business Media, 1992 (cit. on p. 15).

[CGC14] Chung, J., Gülçehre, Ç., Cho, K., Bengio, Y. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. *arXiv preprint arXiv:1412.3555*, 2014. Available at: https://arxiv.org/abs/1412.3555.

[DAP+21] Dubey, A., Ahmad, A., Pasha, M. A., Cammarota, R., Aysu, A. *ModuloNET: Neural Networks Meet Modular Arithmetic for Efficient Hardware Masking*. The International Association for Cryptologic Research, 2021, (1), 1–40. Available at: https://eprint.iacr.org/2021/1437.pdf.

[DHH+18] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., Pierini, M., Rivera, R., Tran, N., Wu, Z. *Fast inference of deep neural networks in FPGAs for particle physics*. *Journal of Instrumentation*, 13(07), P07027, 2018. DOI: 10.1088/1748-0221/13/07/P07027. Available at: https://doi.org/10.1088/1748-0221/13/07/p07027.

[EMP20] M. Ender, A. Moradi, C. Paar. "The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs." In: 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 1803–1819. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/ender (cit. on p. 17).

[FSC00] Felix A. Gers, Jürgen Schmidhuber, Fred Cummins. *Learning to Forget: Continual Prediction with LSTM*. *Neural Computation*, 12(10), 2451–2471, 2000.

[G12] Graves, A. *Supervised Sequence Labelling with Recurrent Neural Networks*. Springer, 2012. Series: Studies in Computational Intelligence, Volume: 385. ISBN: 978-3-642-24796-5. DOI: 10.1007/978-3-642-24797-2.

[GBC16] Goodfellow, I., Bengio, Y., Courville, A. *Deep Learning*. MIT Press, 2016. ISBN: 0262035618. Available at: http://www.deeplearningbook.org.

[GDL24] Gao, C., Delbrück, T., Liu, S. *Spartus: a 9.4 top/s fpga-based lstm accelerator exploiting spatio-temporal sparsity*. *IEEE Transactions on Neural Networks and Learning Systems*, 35(1), 1098-1112, 2024. DOI: 10.1109/tnnls.2022.3180209. Available at: https://doi.org/10.1109/tnnls.2022.3180209.

[GSC00] Gers, F. A., Schmidhuber, J., Cummins, F. *Learning to Forget: Continual Prediction with LSTM*. *Neural Computation*, 12(10), 2451–2471, 2000.

[G21] Gillis, A. *What is a side-channel attack?*. Article on TechTarget, April 2021. Available at: https://www.techtarget.com/searchsecurity/definition/side-channel-attack.

[HKM+17] Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Song, Y., Wang, Y., Dally, W. J. *Ese*. Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2017. DOI: 10.1145/3020078.3021745. Available at: https://doi.org/10.1145/3020078.3021745.

[HS97] Hochreiter, S., Schmidhuber, J. *Long Short-Term Memory*. *Neural Computation*, 9(8), 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[H22] Hertz, J. *Understanding Side Channel Attack Basics*. Technical Article on All About Circuits, 2022. Available at: https://www.allaboutcircuits.com/technical-articles/understanding-side-channel-attack-basics/.

[LBBH98] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. *Gradient-Based Learning Applied to Document Recognition*. *Proceedings of the IEEE*, 86(11), 2278–2324, 1998. DOI: 10.1109/5.726791.

[LCB24] LeCun, Y., Cortes, C., Burges, C. J. C. *THE MNIST DATABASE of handwritten digits*. 2020. Available at: http://yann.lecun.com/exdb/mnist/. Accessed: 2024-02-26.

[MC07] E. Monmasson, M. N. Cirstea. “FPGA Design Methodology for Industrial Control Systems—A Review.” In: *IEEE Transactions on Industrial Electronics* 54.4 (2007), pp. 1824–1842. DOI: 10.1109/TIE.2007.898281 (cit. on p. 15).

[MEY14] U. Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. 4th ed. Springer Publishing Company, Incorporated, 2014. ISBN: 3642453082 (cit. on p. 15).

[NWA16] M. Naghmash, M. Wali, A. Abdulmajeed. “High Level Implementation Methodologies of DSP Module using FPGA and System Generator.” In: *Engineering and Technology Journal* 34 (Feb. 2016), pp. 295–306. DOI: 10.30684/etj.34.2A.9 (cit. on p. 17).

[O15] Olah, C. *Understanding LSTM Networks*. 2015. Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[P02] Perry, Douglas L. VHDL Programming by Example. McGraw-Hill Education, 2002.

[RES93] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli. “Architecture of field-programmable gate arrays.” In: *Proceedings of the IEEE* 81.7 (1993), pp. 1013–1029. DOI: 10.1109/5.231340 (cit. on p. 15).

[S00] Stephen A. Bailey *IEEE Standard VHDL Language Reference Manual*.IEEE Std 1076, 2000 Edition.ISBN 0-7381-1948-2 SH94817. Available at: https://edg.uchicago.edu/~tang/VHDLref.pdf.

[SVL14] Sutskever, I., Vinyals, O., Le, Q. V. *Sequence to Sequence Learning with Neural Networks*. In: Advances in Neural Information Processing Systems, 27, 3104–3112, 2014. Available at: https://arxiv.org/abs/1409.3215.