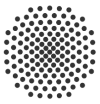

University of Stuttgart
Institute of Computer Architecture and Computer Engineering

Institut für Technische Informatik
Hardwareorientierte informatik
Pfaffenwaldring 47, 70569 Stuttgart

Master Thesis Nr. 051720-019

Implementation and Comparison of Masked and Unmasked LSTM on FPGA.

Rithvik Narayana Swamy

Course of Study:	Information Technology
Examiner:	Prof. Dr. rer. nat. habil. Ilia Polian
Supervisor:	Dipl.-Inf. Devanshi Upadhyaya
Commenced:	November 22, 2023
Completed:	May 22, 2024

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, May 22, 2024, Rithvik Narayana Swamy

Contents

1 Introduction **11**
- 1.1 Neural networks . . . 11
- 1.2 Side-Channel Attacks . . . 16
- 1.3 Technical Tools . . . 17
- 1.4 Goals . . . 19

2 Background **21**
- 2.1 Long Short-Term Memory (LSTM) Networks . . . 21
- 2.2 Modified National Institute of Standards and Technology database . . . 23
- 2.3 ModuloNet: Enhancing LSTM Security . . . 23

3 Methodology **25**
- 3.1 Complexity . . . 25
- 3.2 Flowchart . . . 28

4 Masking **33**
- 4.1 Software Masking . . . 33
- 4.2 Hardware Masking . . . 34
- 4.3 Output . . . 36

5 Comparative Analysis Of LSTM Models **37**
- 5.1 Model 1 . . . 37
- 5.2 Model 2 . . . 38
- 5.3 Model 3 . . . 38
- 5.4 Model 4 . . . 39

6 Software Implementation **41**
- 6.1 Using Tensorflow . . . 41
- 6.2 Unrolled Neural Network using Python . . . 42
- 6.3 Comparison between different software Implementations . . . 45

7 Hardware Implementation **47**
- 7.1 Computational Aspects . . . 47
- 7.2 Parallelization, Pipelining and using Memory . . . 49
- 7.3 Optimization Techniques . . . 53
- 7.4 Implementation of masked LSTM Network on Hardware . . . 56

8 Evaluation **59**
- 8.1 Analysis of Results . . . 59
- 8.2 Comparison with Existing Solutions . . . 61

9 Conclusion and Future Work **63**
9.1 Conclusion 63
9.2 Future Work 63

Bibliography **65**

List of Figures

1.1 A simple CNN architecture 12
1.2 A simple FCNN architecture. 14
1.3 Unrolled Recurrent Neural Network. 15

2.1 Long Short-Term Memory. 22

3.1 Complete data flow of LSTM Neural Network 28
3.2 Input data (digit) fed to the LSTM neural network 29
3.3 Input data represented in binary format 29
3.4 LSTM layer output 30
3.5 Comparison Between Input and LSTM Output 31
3.6 FCNN Neuron 32
3.7 FCNN Layer 1 and Layer 2 Output for digit 9 32

4.1 ModuloNet Masking 34
4.2 Masking in Hardware 35
4.3 Calculations 36

5.1 Block Diagram of Model 1. 37
5.2 Block Diagram of Model 2. 38
5.3 Block Diagram of Model 3. 39
5.4 Block Diagram of Model 4. 40

6.1 Long Short-Term Memory. 42
6.2 Unrolled LSTM Output. 45

7.1 LSTM Simulation Output. 49
7.2 Sequential LSTM Simulation Output. 54
7.3 Sequential LSTM Simulation Output. 55
7.4 Sequential LSTM Simulation Output. 55
7.5 Hardware Implementation of LSTM 58
7.6 Overall Pictue of Hardware Implementation of LSTM 58

8.1 Device Utilization summary Of Different Models. 60
8.2 Camparison between Actual Output against Computed output 61

List of Tables

6.1 Comparison of TensorFlow code and unrolled LSTM code with accuracy and other metrics. 45
6.2 Comparison of different size LSTM neural networks with accuracy and other metrics. 46

7.1 Advanced HDL Synthesis Report Before parallelization,pipelining and memory. 50
7.2 Advanced HDL Synthesis Report after parallelization,pipelining and memory utilization 56
7.3 Device utilization summary 57

8.1 Comparison of Unmasked and Masked LSTM Networks 59
8.2 Comparison of Unmasked and Masked LSTM Networks with Device utilization metrics 60

1 Introduction

Ride-sharing applications [SSB23], intelligent sorting in Gmail and speech recognition [HKM+17] are a few examples of how neural networks have improved people's quality of life. The most innovative technology in neural network is that once trained they start learning by themselves. safeguarding sensitive data processed by neural networks has become a paramount concern. 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 [CDGK21]. 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) can enhance the security of LSTM networks against side-channel attacks [DCA20]. FPGAs offer a flexible and efficient platform for deploying neural networks, allowing rapid prototyping and implementation.

This report presents a novel approach which integrates software masking and hardware making technique to secure neural network inference with an LSTM neural network. In particular, Masking 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 .

LSTM frameworks can incorporate and apply two step masking to provide a new standard for the safe 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

Inspired by the organic neural networks seen in animal brains, neural networks are a fundamental building block of artificial intelligence (AI) and machine learning[KBBM16]. 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 networks 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 security systems, for example, face recognition software uses neural networks to identify and validate individuals with precision [SK15]. 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[LZZZ20]. 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. 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[CLB+21]. Figure 1.1 and Figure 1.2 are extracted from [L24].

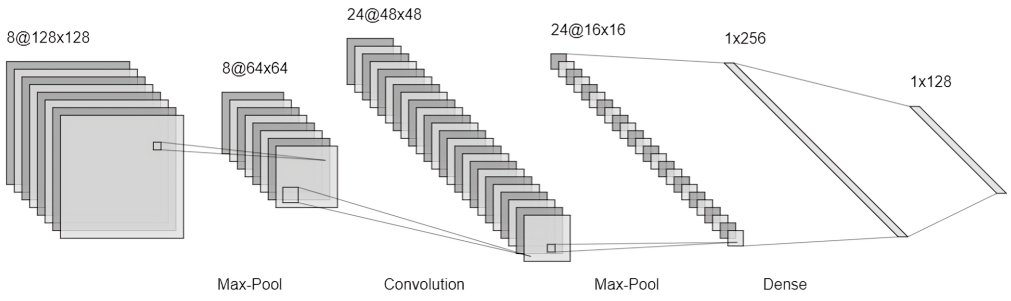

Figure 1.1: A simple CNN architecture .

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. In turn, the activation function determines whether or not a neuron may be triggered[P24].

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. As seen in Figure 1.1, neurons in a fully linked layer are coupled to every activation in the layer before them. 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: It is the initial layer of the network, where each node (shown by a circle) is a single 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: Between the input and output layers are intermediary layers known as "hidden layersßince 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 $\in \mathbb{R}^{10}$").

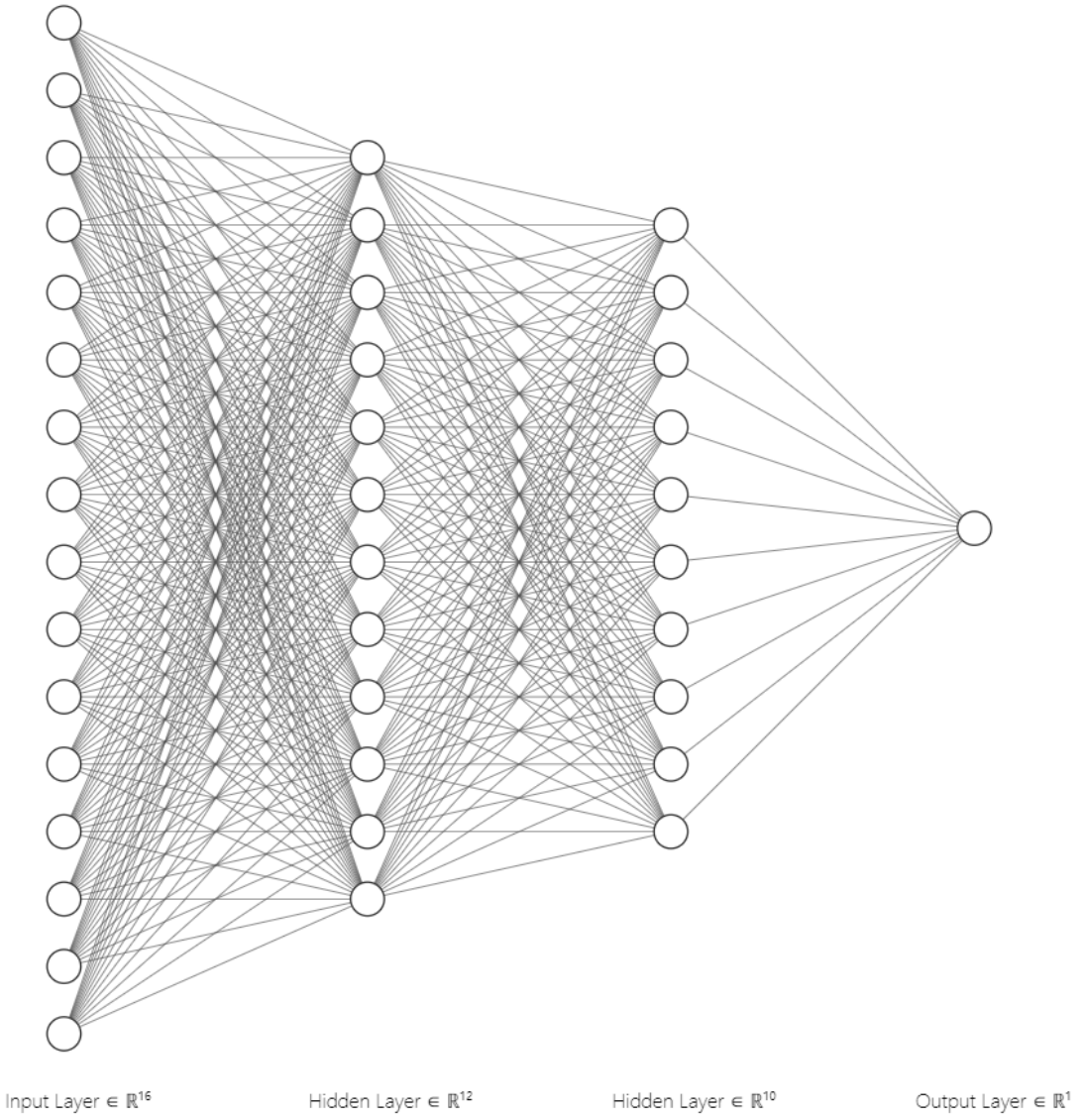

Figure 1.2: A simple FCNN architecture.

Output Layer: In a network, this is the last tier. 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 $\in \mathbb{R}^1$"), This implies that either a binary classification goal or a regression task is what the network is intended to do.

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. To obtain the intended output, the output of each iteration is calculated using the input of the subsequent iteration. Figure 1.3 depicts how RNN is unfolded in time. 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].

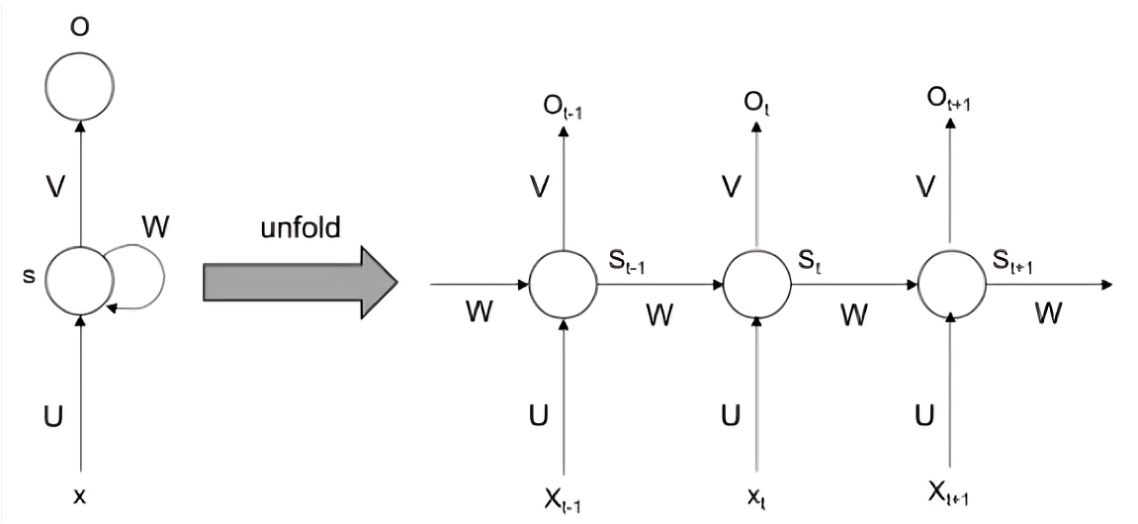

Figure 1.3: Unrolled Recurrent Neural Network.

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: applied weight matrix 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. During this period 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] . For more information on basics of side channel attacks you can refer to [H22] .

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 happened in 1980s, FPGAs was used 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 crucial components. The first element comprise logic blocks which can be assigned to do predefined logical functions. This logic blocks are the basis of FPGA, they are called as Configurable Logic Block (CLB). CLBs can execute complex logic functions. It is also used to implement memory functions, and synchronize code on the FPGA. A number of Look up tables (LUTs) in many FPGAs are used to build the circuit components. The existence of Block Random Access Memory (BRAM) on the FPGA makes it more dynamic [BFRV92].

Apart from this, the trace of the design is made using Spartan-6 LX-75 FPGA from Xilinx given in [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 you can refer to the Table 8.2 on page 60 to get a more idea on FPGA device utilisation summary .

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 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.
- **Dynamic Elaboration**: Three distinct situations lead to dynamic elaboration when sequential constructs in a model are executed. Before executing the enclosed statements of a loop, the loop parameter is assessed and its discrete range is elaborated in a loop statement using a for iteration method. Second, the parameter interface list is expanded during a subprogram call, generating formal parameters and linking them to real parameters. If the subprogram is a protected type method, elaboration may not proceed until exclusive access to the object is obtained. If the subprogram does not have a 'FOREIGN attribute, elaboration is dependent on the implementation. Finally, before allocating the new object, the subtype indication is elaborated when assessing an allocator with one[00].
- **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. Furthermore, the Place and Routing stage is responsible for verifying such ones with regard to the physical arrangement of the components on the FPGA and their connections.

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 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 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, four kinds of LSTM models are dicussed, first one is the unmasked LSTM, followed by LSTM with compressed data. Then the third model consists of software masking and the final model consists of both hardware and software masking. It is, in fact, making the masked LSTM of the project using a two step masking approach and using it along with the unmasked counterpart of the LSTM to compare the results .

2 Background

2.1 Long Short-Term Memory (LSTM) Networks

RNN includes LSTM networks. LSTMs are robust in contrast to traditional RNNs since they are made to handle sequential input, which is difficult to model long-term dependencies with issues like vanishing and expanding gradients. LSTM contains two main memory blocks ,short memory cells and long memory cells can memorize the data for a very long time, using gates which influence the data flow. 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: forget gate, input gate and output gate. 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 neurons, which controls the message flow. The input gate computes newly introduced information with the stored information, the forget gate decides which data can be throwed away [GSC00], the output gate represents that what recorded data should be used to generate it into the output signal at each step. This mechanism of LSTMs gates make them critical in choosing which information is appropriate for long sequences with a "vanishing gradient"[FSC00] .

LSTM models sre 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.

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 $\sigma\left(\left(W_f H_{t-1} + X_t + b_f\right)\right)$, where σ is the sigmoid function, W_f is the forget gate weight matrix, and b_f contains the bias value of forget gate.

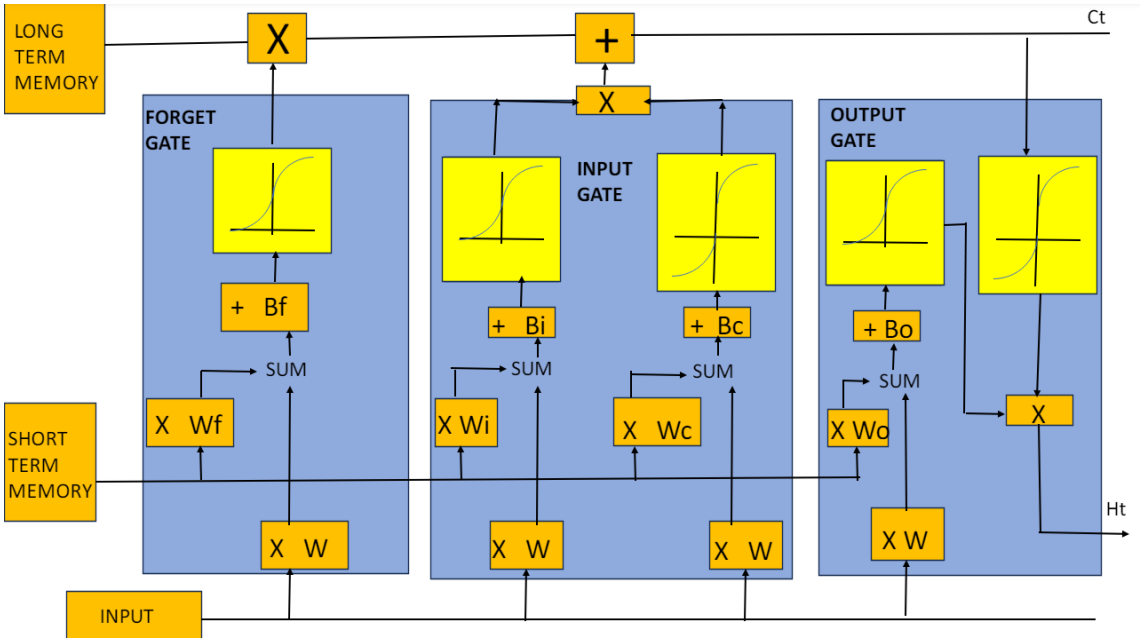

Figure 2.1: Long Short-Term Memory.

- **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 $H_t = o_t * \tanh(C_t)$.

The sigmoid (σ) and hyperbolic tangent (tanh) functions are the activation functions used in LSTM. The sigmoid and tanh activation functions are essential for controlling information flow and preserving stability in long short-term memory (LSTM) networks. The input, forget, and output gates use the sigmoid function, whose output range is (0, 1), to control the amount of information that flows through by making judgments that resemble binary operations, which is in line with the gating method. Stable gradient-based optimization is further supported by its smooth and differentiable character. The tanh function, outputting values between -1 and 1, is utilized for updating the cell state and hidden state, offering a rich representation of both positive and negative values and ensuring balanced activations. With this combination, LSTMs can solve the vanishing/exploding gradient issues that RNNs have and efficiently learn long-term dependencies.

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. After

that, a fully connected layer using a softmax function processes the vector to provide probability values for each class digit. In order to ensure accurate handwritten digit categorization, the LSTM's weights (W) and biases (b) on the gates are first tuned during the training phase.

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 [LCB24]. 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 [20].

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 [20].

Many 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.With packages like Deep Lake, the data is easily accessible and loaded into Python, allowing for the rapid development of script files for various uses, such as testing and training.

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. A greater degree of confidentiality and integrity can be achieved by including modular arithmetic operations into neural network processes.

Fundamental Idea: The key point in ModuloNet concept is that it replaces integer arithmetic in neural network by 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 neural networks from scratch, ModuloNET initializes the modular arithmetic instead of normal addition framework and leverages it for a secure implementation of the process. The method of ModuloNET increases the computational calculations of the network but it eliminates the risk of side attacks, and in result, gives a more secure neural network [DAP+21]

Masking Techniques: In modeling it, ModuloNET applies maskings for components like comparator, activation function, multiplier to compute the results 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[DAP+17].

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

Disadvantage: Modulo folding results in the FCNN layer's negative output scores wrapping around, which leads to inaccurate predictions with high confidence scores. The activation function can no longer be used to inject nonlinearity into the circuit because all activations are now limited to a smaller range. In batch normalization (BN), the modulo operation shifting negative makes things challenging. Blocking BN from normalizing activations to a Gaussian distribution with a zero center of gravity and pushing activations to the positive side. Due to overflows, the modulo operation results in sudden changes in representations and information loss[DAP+21].

3 Methodology

3.1 Complexity

Consider 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. Here, we're utilizing fixed point notation, where the first bit denotes a signbit, the next seven bits denote an integer portion, followed by fractional part in the last 8 bits.

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 .

- 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 = $30 + 30 + 10 + 10 = 80$.

Here's the VHDL code for multiplication:

```
  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 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. 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)

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;

```

- 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 [11c]. This RAM values can convert any value in the range -128 to 127.999 to their approximated activation function value .

3.2 Flowchart

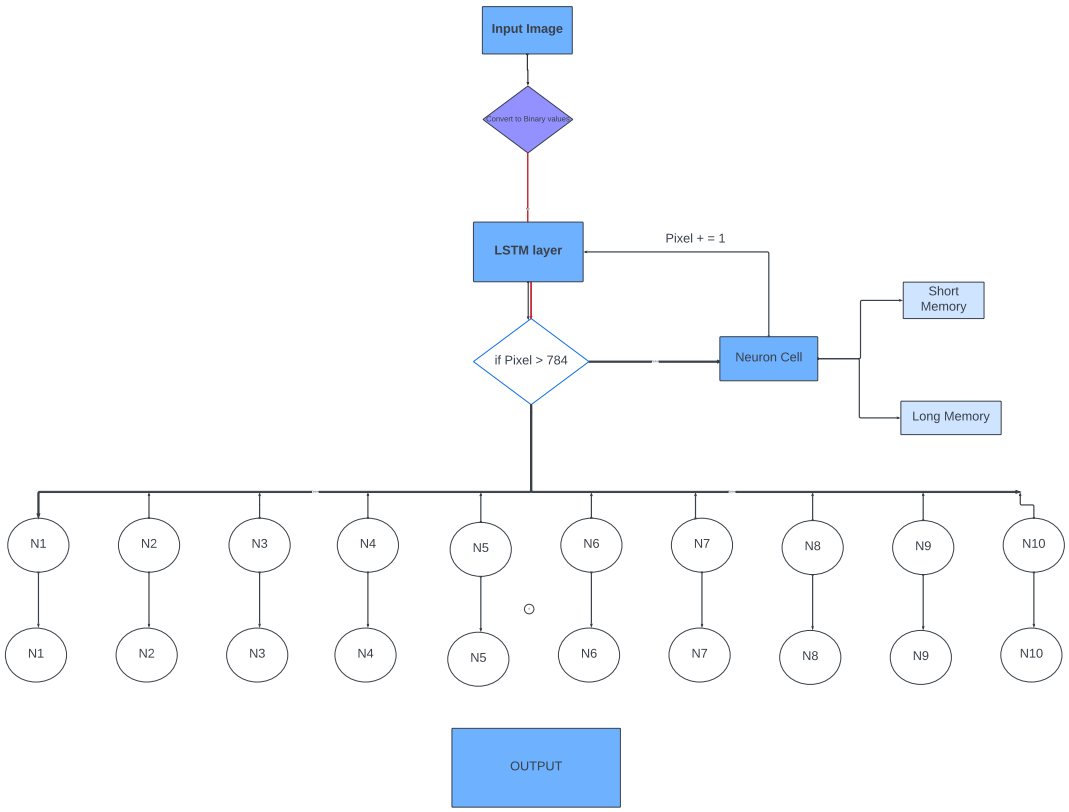

Figure 3.1: Complete data flow of LSTM Neural Network

The Figure 3.1 represents an LSTM Network with 2 fully connected layers. Both FCNN layer's comprises of ten 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 determines which neuron in the final layer has the maximum value and outputs the value's index. This represents the network's output.

For instance, suppose the number is eight, as in Figure 3.2. The max value should be present in the neuron N8, so the weights and biases are computed in a way to get this desired output. The next part goes over each flowchart block's specific method in depth.

3.2.1 STEP 1

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

Let's start with the input digit as shown in Figure 3.2. 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. These integer values are converted to binary after being normalized to ensure they don't go above the limit.

3.2.2 STEP 2

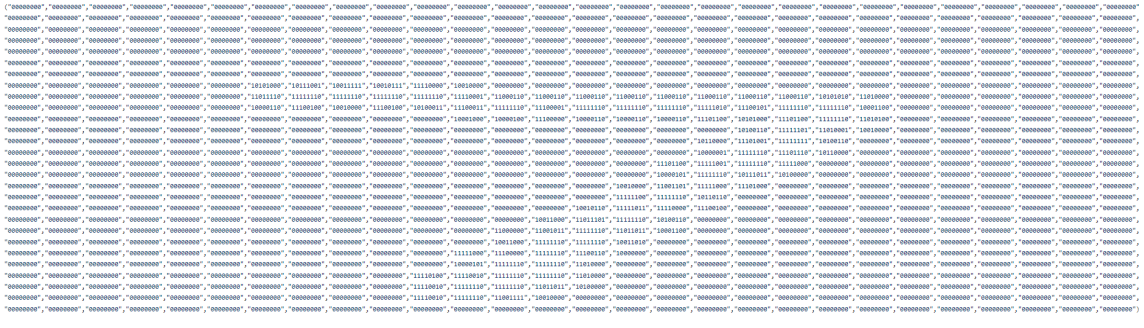

Figure 3.3: Input data represented in binary format

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. The procedure then proceeds as illustrated on Figure 3.1 on page 28. Following the initial iteration, you obtain the short and long-term memory, which are then utilized to calculate the subsequent input. After 784 iterations all the values complete their computations and the memory elements are set to zero for the next input.

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, 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, 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 0 or modulated to 1. As seen in Figure 3.4, zero represents the Background and one indicates the picture. 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. Additionally, we can utilize "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 . The graph below provides an Overview of The LSTM output. You can observe that for all the pixel Input spike, Lstm reacts precisely. The green

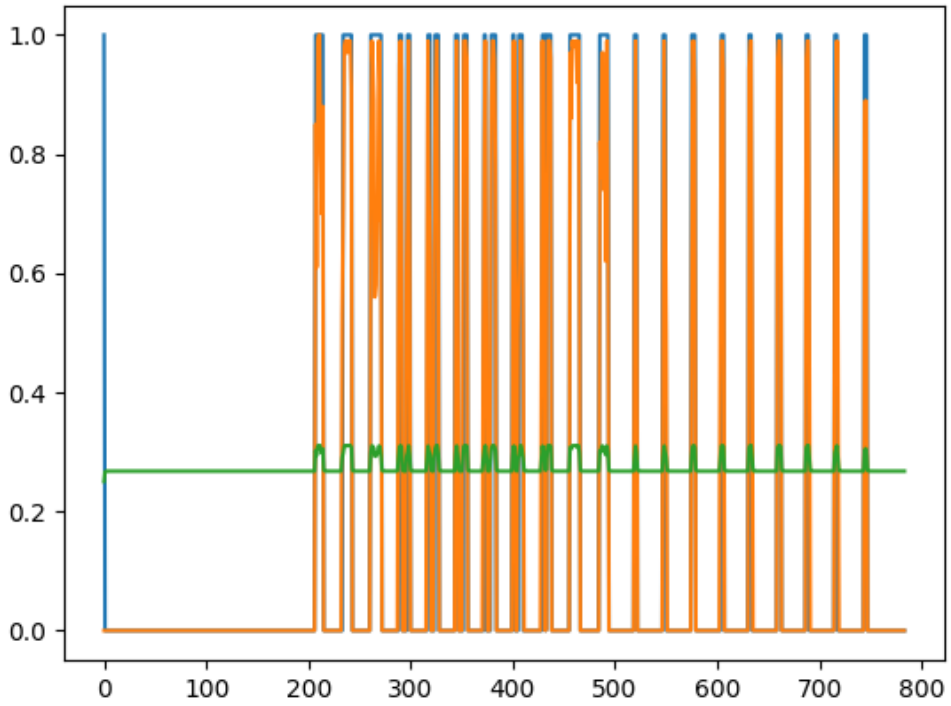

Figure 3.5: Comparison Between Input and LSTM Output

signal is actual LSTM output which is provided to the feedback circuit. But for output computations we have to modulate the signal to 1. Now the Modulated signal is almost similar to the Input signal. The FCNN layer receives this output as shown in Figure 3.4.

3.2.4 STEP 4

Keeping the complexities into count, the FCNN layer is designed. The single neuron in action is depicted in Figure 3.6. 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.

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 following layer of the FCNN receives the output of the sigmoid function. The sigmoid function output is always in the range from 0 to 1.

The Figure 3.7 represents output of layer 1 and layer 2 of fully connected layer. here the classified digit is 9. The values of the second layer are so small as you can see in the grapgh, this is due to the values have been passed through sigmoid twice. Therefore the difference between the value in the final output is so small, which may lead to wrong classification sometimes .

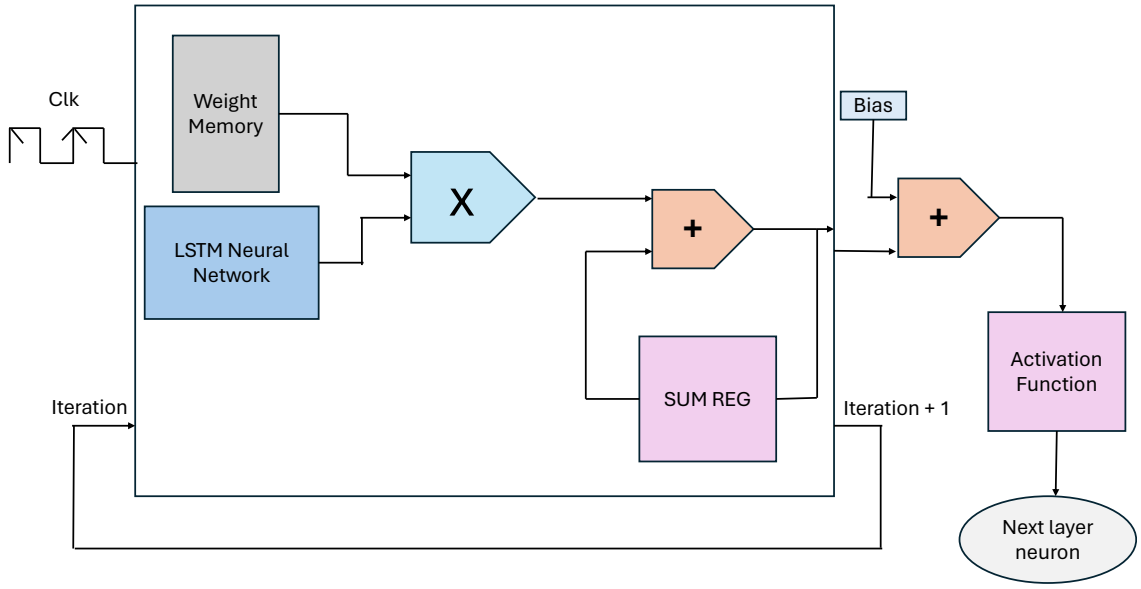

Figure 3.6: FCNN Neuron

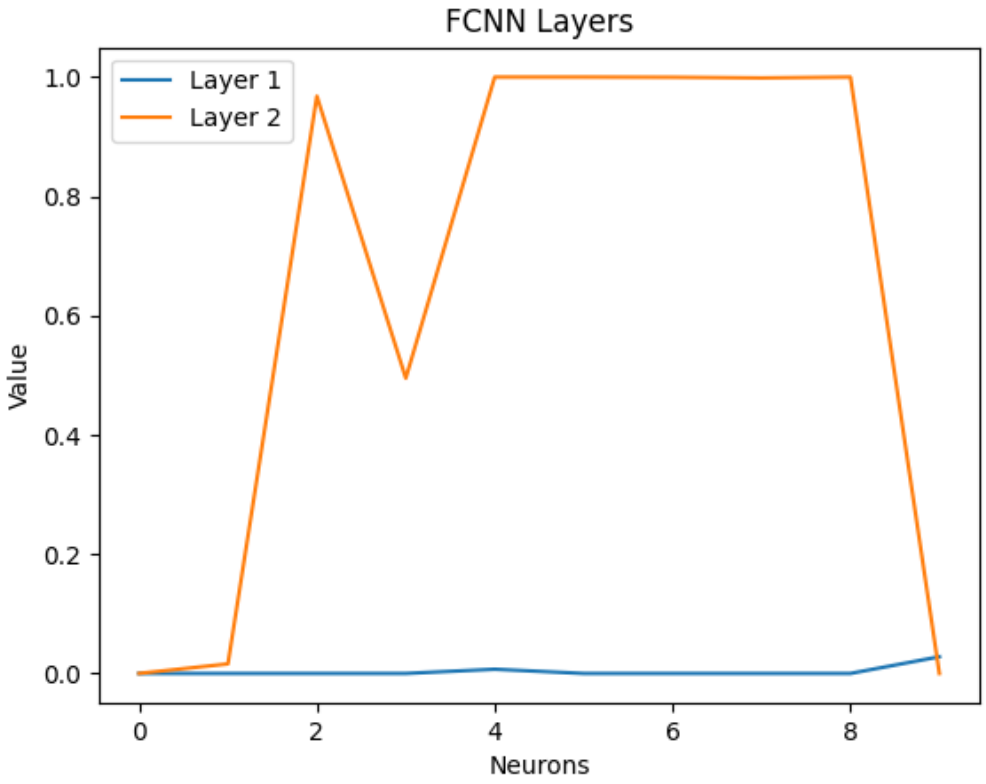

Figure 3.7: FCNN Layer 1 and Layer 2 Output for digit 9

4 Masking

This thesis primary goal was to retrace the development of ModuloNet 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 ModuloNET technique because the output of sigmoid lies between 0 and 1 and also ISE 14.7 does not 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. Because the weights in our model are the secret, building the model requires more components in this method.
- Modulo operations include data loss and abrupt representational shifts brought on by overflows.
- Application of LSTM becomes challenging. Tanh activations that are negative are moved to the positive side by modulo operation. 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 .
- Negative output scores from the Lstm and FCNN layers wrap around as a result of modulo folding. This leads to inaccurate forecasts.

Keeping this problems in mind we propose a different modified version of masking with the help of ModuloNet and DOM. 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 .

4.1 Software Masking

In this approach the weights are passed to the block integer to binary converter, here the input is converted to 16 bit binary number where first bit reprsents the sign bit, next 7 bits represent integer part and the last 8 bits represent fractional part following fixed point representation. Next the 16 bit binary value is sent to compressor which produces an output of 8 bit binary value. This binary values are used as weights in the $RAM0$ to $RAM9$ as shown in the above VHDL code. The Figure 4.1 represents the Software Masking, in which the weights are converted to a 8 bit value. This process is completely performed using python. The next step is to introduce a random value and append it to each of the values. This addition of value provides us an 16 bit masked value ,the final masked output values are stored in memory .

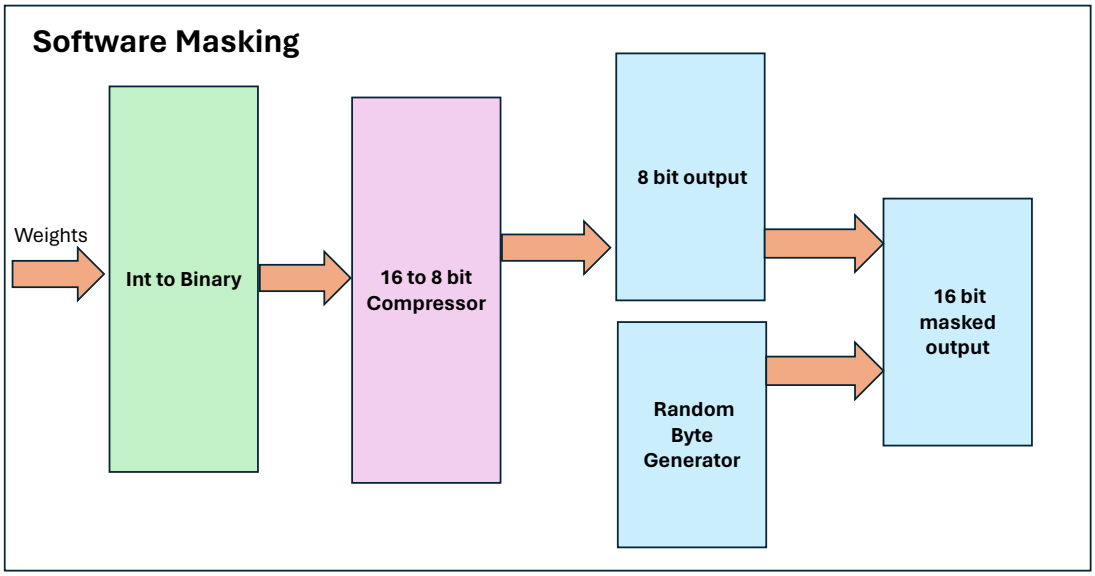

Figure 4.1: ModuloNet Masking

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

```
1
2
3     architecture syn of ram_0 is
4
5     type ram_type is array (0 to 783) of std_logic_vector(7 downto 0);
6     signal ram : ram_type := (
7   "00000000","00000000","00000000",.....
8   "00000000","00000001","00000000","00000000","00000000","00000000");
9   begin
10
```

4.2 Hardware Masking

Once the weights are placed on to the memory as referenced in [10a] ,the hardware consists of a Linear feedback shift register for random byte generator as shown in Figure 4.2. 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 first 4 bits followed by bit 9 to bit 12 from masked weights and 8 bits from random LFSR. The second share contains bit 5 to bit 8 followed by last 4 bits and appended by LFSR Figure 4.3 on page 36. This two share are computed independently with the LSTM output.

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(15 downto 0);
    signal ram : ram_type := (
"0000000101000010","0000000101000001","1000000010010110",
"0000000001000111".........."1000000001100100");
begin

```

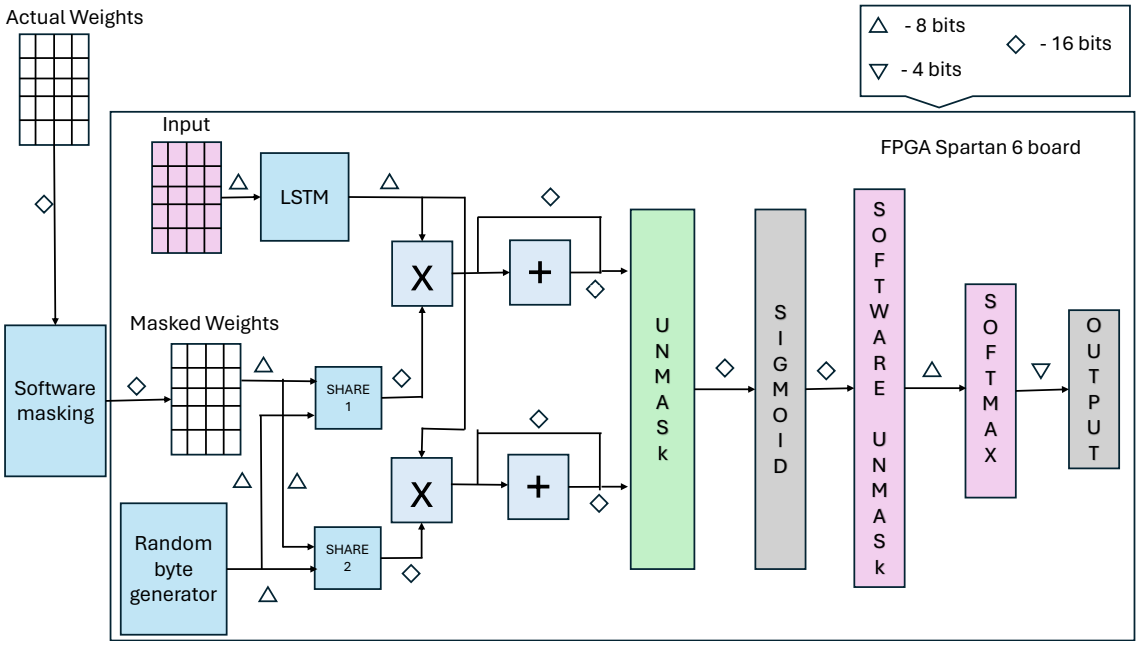

Figure 4.2: Masking in Hardware

After that, the neuron computation starts, as seen in Figure 3.6 on page 32. 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.

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.

4.3 Output

The first level of Unmasking is done before the value is fed to the sigmoid block.Here the last eight bits are removed from each shares and then they are concatinated to form a 16 bit binary number.Due to this process, the accuracy of the system reduces because there can be changes in the values after unmasking , The first 4 bits are the imporant bits in the independent share 1 beacuse the Look up table of Sigmoid function is designed in a way to handle this randomness introduced in software.The first bit represnts sign for sigmoid.Then the next 3 bits represent integer value followed by 12 bits fractional part. Now even if the random values which are located at the last 8 bits after concatination are added it does not change the first 4 bits in most of the cases.

Lets Discuss this with an example, the weights is -3.0625 which is represented by a 8 bits. Next, the software adds a random value. This masked value is stored in the memory. The harware generates two random bytes using linear feedback Shift register (LFSR) which is appended as shown in Figure 4.3. Then after the computations and first level of unmasking you get the 16 bit value which is fed to the sigmoid function. After all the computations with 784 values, the share 1 first 4 values and share 2 first 4 values are used to compute the sigmoid values. The sigmoid LUT is designed in a way to capture this values and produce a output. The output of the sigmoid is unmasked to get the final value for a single layer neural network, If you have more than 1 FCNN layer. You can continue with the masked sigmoid output and provide the value as a input to the next FCNN layer .

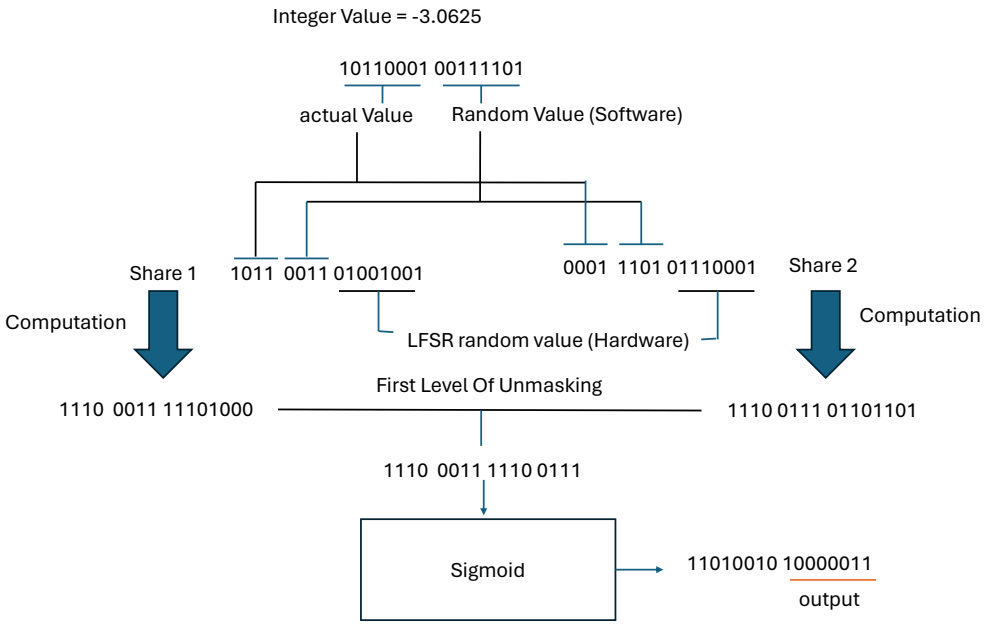

Figure 4.3: Calculations

5 Comparative Analysis Of LSTM Models

In this section let's see how different LSTM models are simulated and synthesised. All the model follows similar approach as explained in methodology. The comaprison of accuracy and other metrics for different model can bre seen in Table 8.1 on page 59. the Device utilisation summary for different models is shown in Table 8.2 on page 60.

5.1 Model 1

The first model is the simple LSTM model, The weights are converted to fixed point binary numbers and stored in the memory. The input image pixels are also stored in memory, at each clock cycle the value is read from input pixel and fed to the LSTM block, this block computes the input as shown in Figure 2.1 on page 22. The short term memory and the long term memory are updated then the output is modulated before giving it to the FCNN stage. The modulated output is multiplied with the FCNN weights on the same clock cycle and then adds the value to the sum register. After 784 clock cycles the output is obtained from this model. In this model the 16 bits are used to represent the weights .

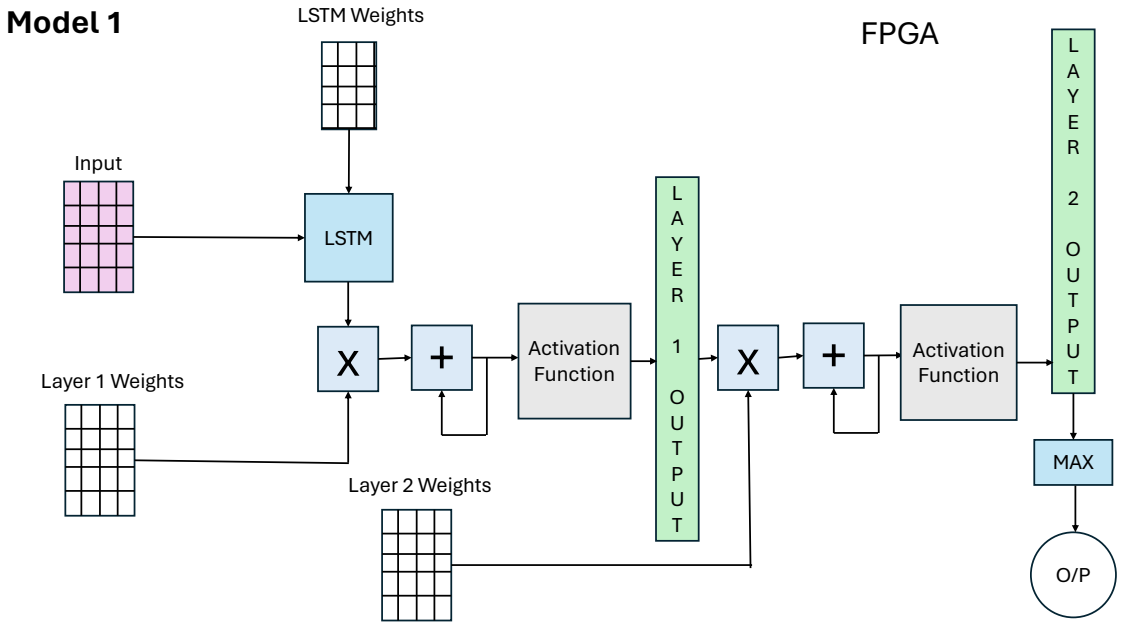

Figure 5.1: Block Diagram of Model 1.

5.2 Model 2

In this model we compress the weights to 8 bit binary value in software and then stored in the memory. This make the model effecint due to 50% reduction of bits in memory. Considering one unit cell LSTM layer followed by 10, 10 FCNN layer the total number of weights are around 8000. each weights are 16 bits with a total of 128kbits, hence by reducing the weights to 8 bits, the memory storage will be equal to 64kbits. This model is energy effecient and matches resource constraints. Therefore it can be implemented on Spartan 6 FPGA board.

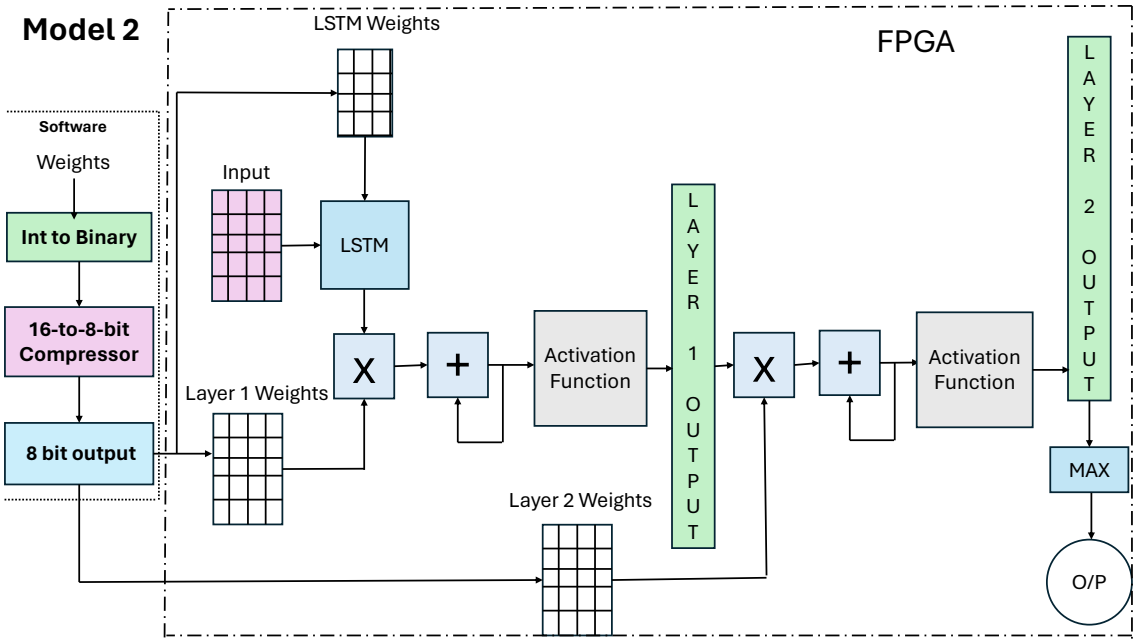

Figure 5.2: Block Diagram of Model 2.

5.3 Model 3

Model 3 introduces the software masking where the compressed 8 bits are appended with a random byte to form a 16 bits masked value and stored in the memory. The working of the other blocks are similar to the other model. The only change is in the Sigmoid Look Up table, where the values are computed with respect to Figure 4.3 on page 36. The output is provided to max function, which gives the final output. The accuracy of the model reduces due to the masking process but provides better security than Model 1 and Model 2 .

Model 3 also faces the same problem as Model 1, where the resources needed to compute the model is more than the available resource. Hence these two models can be simulated and synthesised but cannot be mapped onto the hardware. The mapping error is limited to spartan 6 FPGA board. This Model can be mapped on to the higher version FPGA board.

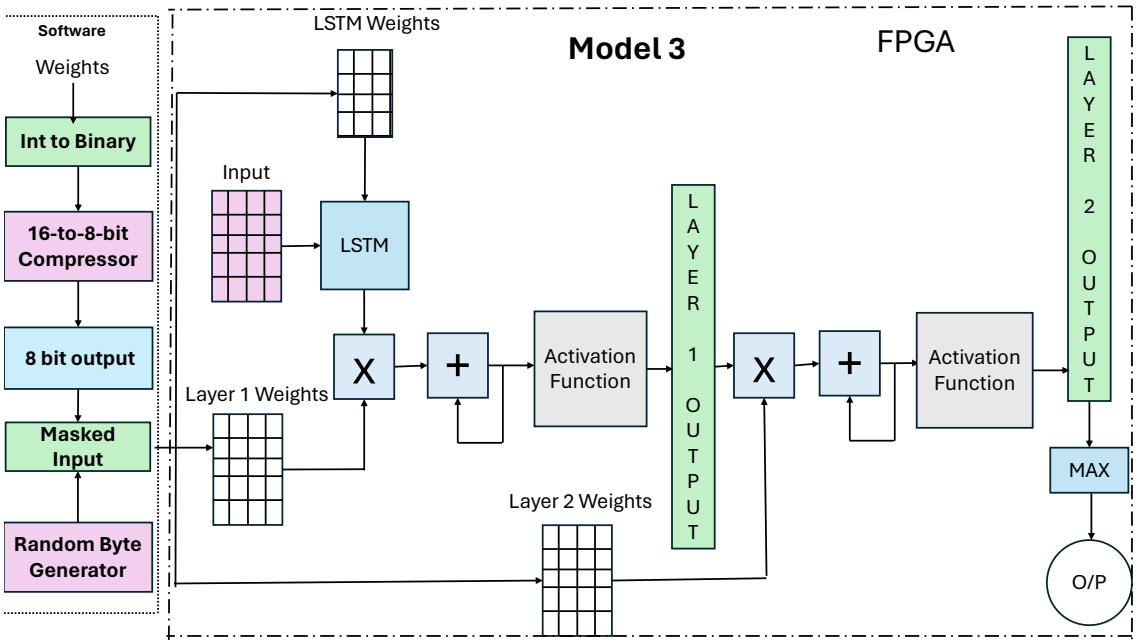

Figure 5.3: Block Diagram of Model 3.

5.4 Model 4

Coming to the final Model, Here a two step masking is done. The first level of masking is performed in software and the second level of masking is performed in hardware. This model is similar to the Figure 4.1 on page 34. The advantage of this model is two level security for the weights. The model consist of a single FCNN layer with 10 neurons. Compared to the other three types, this considerably lowers the accuracy.

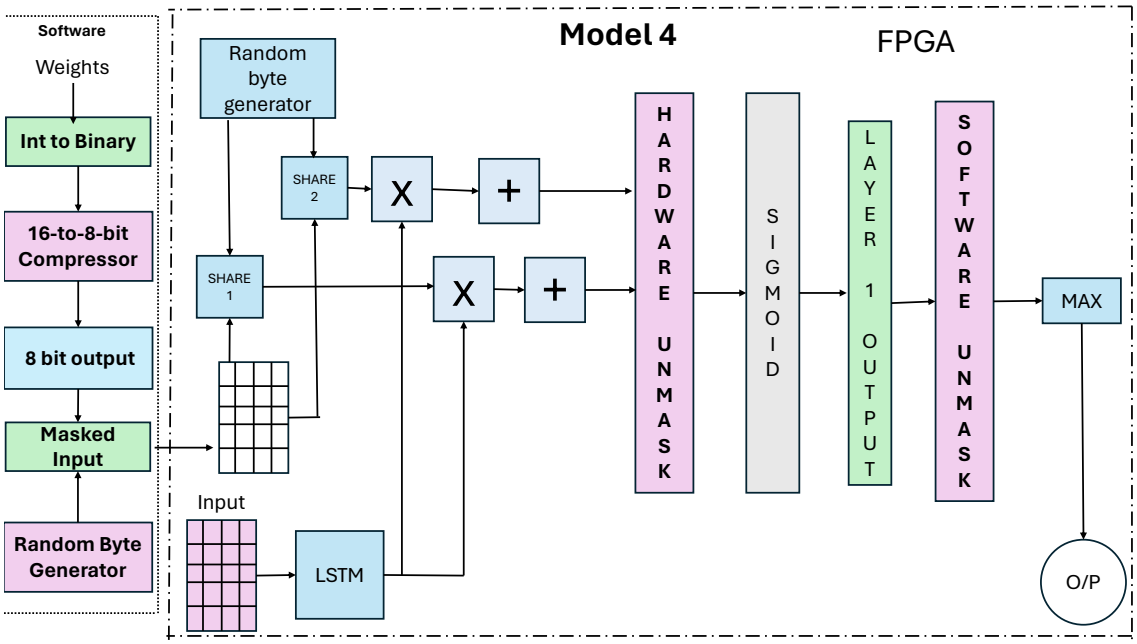

Figure 5.4: Block Diagram of Model 4.

6 Software Implementation

LSTM neural networks in 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 deep learning framework such as TensorFlow, PyTorch, or Keras, these frameworks has a deep rooted aptitude to construct and train LSTM models. After choosing the framework, you must choose the number of LSTM layers, the quantity of memory cells in each layer, and the activation functions that will be applied. 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. For this reason, it's critical to use the data in the appropriate 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 Stochastic gradient descent (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.

6.1 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(Tensors are the central data abstraction in PyTorch), 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 .

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 .

This iteration's outcome is shown in Figure 6.1. The tensorflow code modifies the weight and bias parameters to improve the model's accuracy and decrease its loss. 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 6.1: Long Short-Term Memory.

6.2 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:

- First, clearly outlining the layers, number of neurons in each layer, and activation functions of the deployed neural network. Moreover, initialize biases and weights for every neuron in every layer. 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 .

```
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 6.1 Python code Using Tensorflow to Implement LSTM and obtain the weights .

The accuracy obtained from the Tensorflow code is 96.5 %

- 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 .
- The last layer's output is computed using the softmax function. The index of the neuron having maximum value is provided as output .

Figure 6.2 displays the unrolled LSTM code's output. 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 %.

https://github.com/Rithvikns/lstm_neural_network_on_fpga/blob/main/python are the links where you can find the code.

```
    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 6.2 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 6.2: Unrolled LSTM Output.

6.3 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 6.1: 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 significant to note that the rounding off of the floating point weights and biases is what causes the accuracy to decrease although the code stays the same in both cases.

	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 6.2: 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.

7 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/Sequential_lstm. In the technical portion of this section, we will carefully go through each step so that understanding it clearly becomes possible.

7.1 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 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. The simulation model is designed with the help of [22b], You can see the simulation output in Figure 7.1 on page 49, 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 .

Combinational circuits provide the primary benefit of eliminating the need to wait for the output signal. Because of its nature, the output is available right away, but the biggest drawback is the need for a massive amount of resources in order to compute everything concurrently. 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 resembles the single port RAM code used in the sigmoid activation algorithm. The memory array's 784-item capacity and each item's 8-bit length, however, are the main modifications. The read operation of single port RAMs are asynchronous, meaning that only the write operation is time-dependent.

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:

```
    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. This concurrent code is synthesised and simulated in vivado [22c] to check the model can be mapped on to different FPGA board .

7.1.1 Routing Challenges in FPGA Design

Routing Congestion: As designs become more complex and demand higher resource utilization, routing congestion can occur. This congestion can lead to difficulties in completing the design routing efficiently, potentially resulting in incomplete routing [10b].

Resource Utilization: Increasing resource utilization within the FPGA, such as logic, block RAM, and DSP resources, can lead to routing challenges. Efficiently utilizing these resources while managing routing congestion is crucial for optimal design performance[10b].

Design Complexity: As designs become more intricate and demand higher bandwidth, the need for efficient routing becomes paramount. Routing issues can arise in complex designs with several interconnected modules and high-speed interfaces, which must be resolved at the design stage[10b].

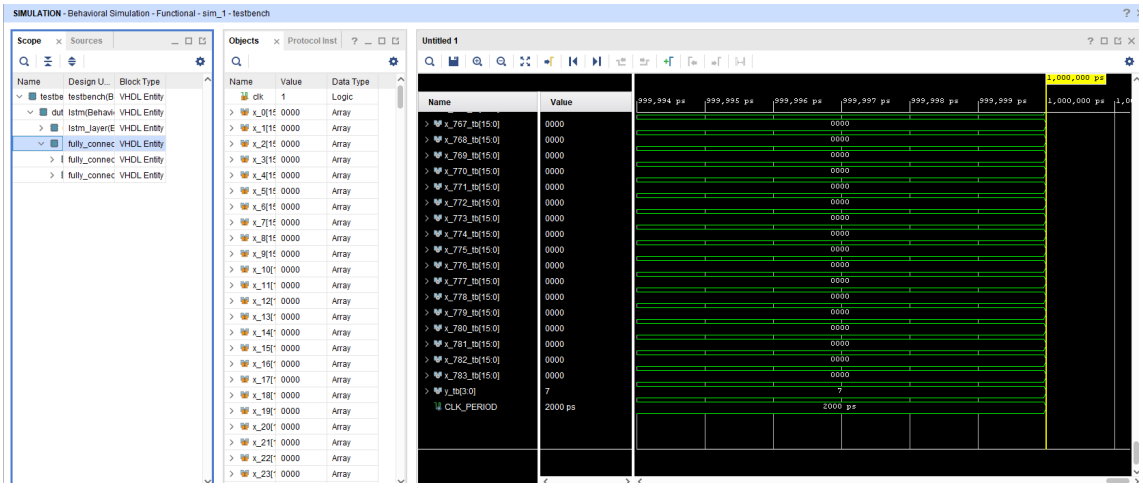

Figure 7.1: LSTM Simulation Output.

7.2 Parallelization, Pipelining and using Memory

In order to minimize resource utilisation of the neural network, 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 current phase and fetching data for the successive phases so that the time duration of the model 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. 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

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

representing each neuron weights, i.e., 784×10 as discussed before. Each RAM has 784 items, and each item is 16 bits long or 8 bits long depending on the model 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 output of the LSTM layer is y is obtained in the same time step. The process of a fully connected layer now starts. 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 is realised using shift registers. The same logic works for the summation part where the sum register is updated with the value of product reg at each time stamp .This mechanisms realises a feedback circuit .

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:

```
    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;


```

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 a parallel block which executed during each 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 the sum registers .

- `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";`

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

```
    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;


```

- `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";`
- `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 .

7.3 Optimization Techniques

Using these techniques the mapping of VHDL code was possible on to the spartan 6 FPGA board. The outcomes are displayed in Table 7.2 .

- **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 [DHH+18].
- **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 [30]. 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 .
- **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, according to the 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 .
- **Hierarchical Design:** Creating an elemental design that has hierarchical components defined to and supports scalability, reusability, and maintainability. lowering the number of levels in the hierarchical architecture compared to the earlier concurrent LSTM code, making it more straightforward and efficient. Figure 7.2 displays the output of the simulation, 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 7.3 and Figure 7.4 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 7.2: Sequential LSTM Simulation Output.

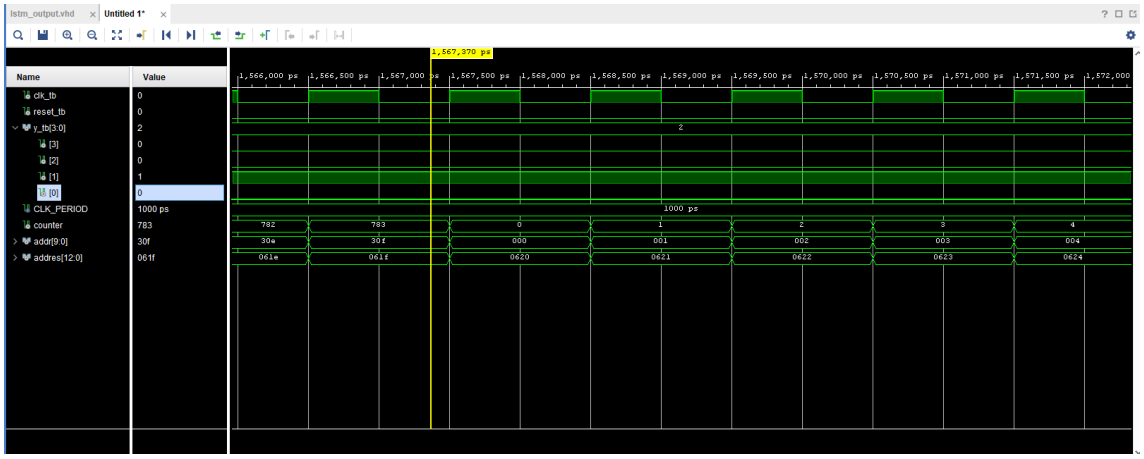

Figure 7.3: Sequential LSTM Simulation Output.

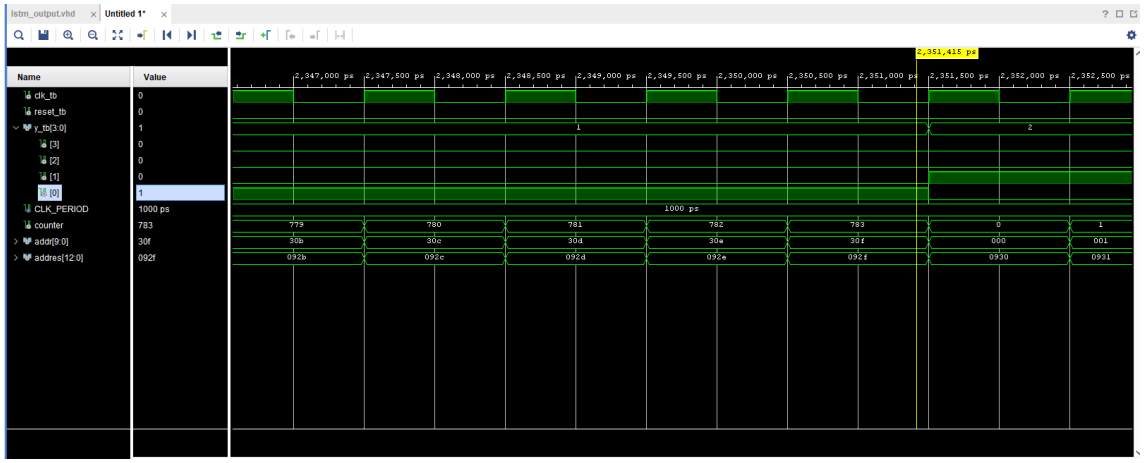

Figure 7.4: Sequential LSTM Simulation Output.

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 .

To understand the sequential LSTM module in detail let's look at Figure 7.5 on page 58. At each time step input X_i is obtained from the input RAM. The neuron receives X_i, after computation the memory elements are updated with new values 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 1. Next step is to send this values to the next layer and compute the final value by using max function. Figure 7.6 on page 58 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 7.6 refers to the Figure 2.1 on page 22 .

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

7.4 Implementation of 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 7.3: 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. By storing the data in memory, the LSTM network uses the masked weights as their true weights. This minimizes the possibility of side-channel assaults. 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 software masking. This two step masking plays a major role in increasing the security of the LSTM Network. The unmasking of the weights is done in different stages. The bias

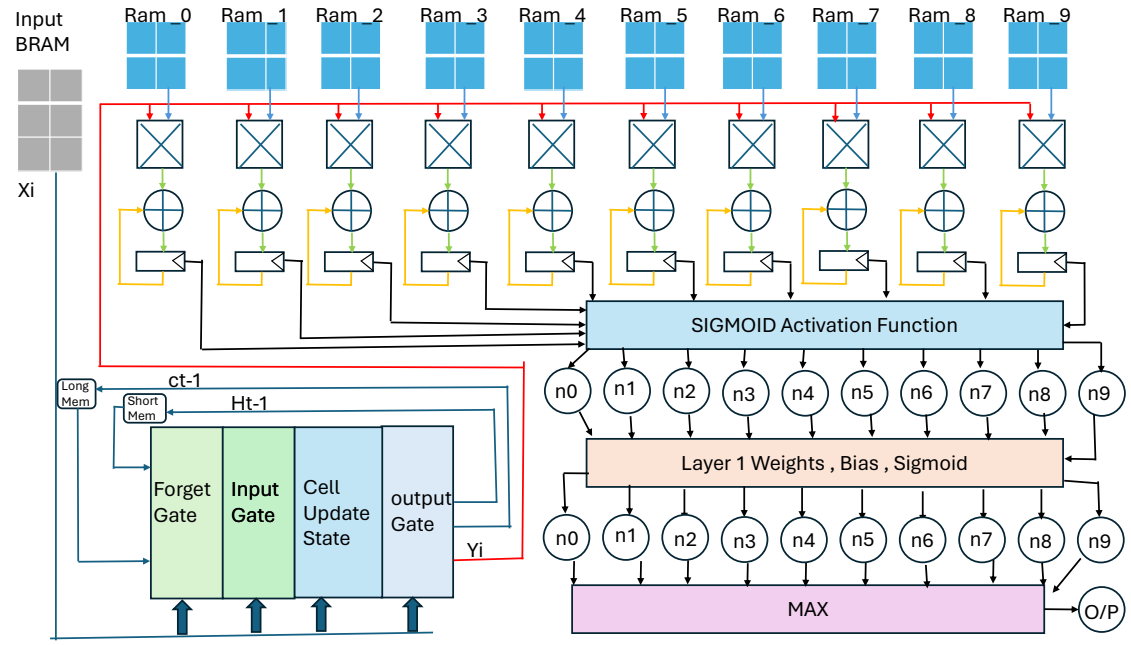

Figure 7.5: Hardware Implementation of LSTM .

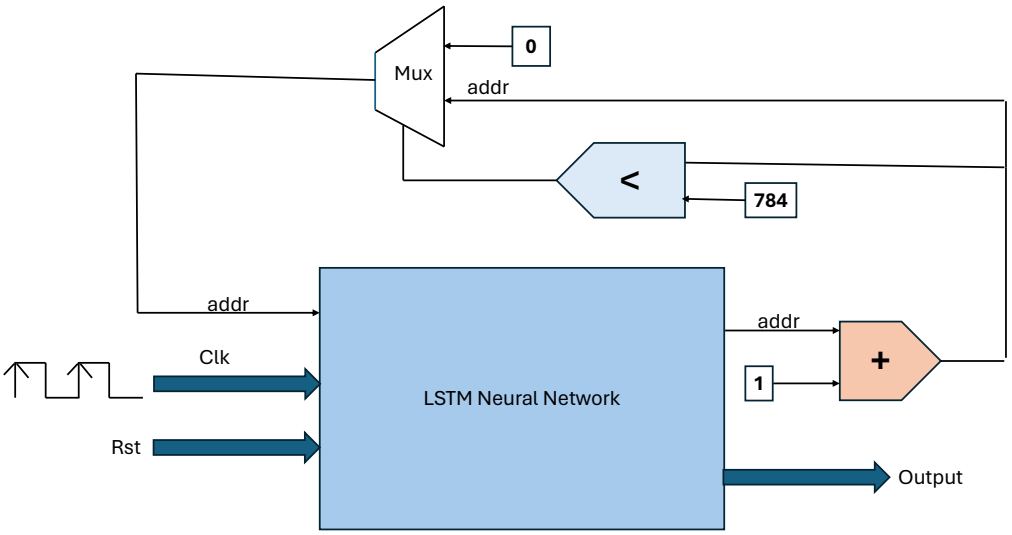

Figure 7.6: Overall Pictue of Hardware Implementation of LSTM .

value 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 unmasking id done as discussed in methodology .

8 Evaluation

In designing a masked and unmasked LSTM network, the results show the balancing act of security versus performance versus resource constraints. Initially, the evaluation of the performance of all the 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.

8.1 Analysis of Results

Metric	Model 1	Model 2	Model 3	Model 4
Accuracy	85%	82%	78%	64%
Testing Time	784 ns	784 ns	784 ns	784 ns
Side Channel Attack Mitigation	Low	Low	Medium	High
Resource Consumption	High (>100%)	Medium	High(>100%)	High(>100%)
Resistance to Adversarial Attacks	low	low	Medium	high
Energy Efficiency	Low	Medium	Low	Low
Scalability	High	Medium	Medium	Low
Deployment Overhead	High	Low	Highest	High

Table 8.1: Comparison of Unmasked and Masked LSTM Networks

The Table 8.1 aim for in-depth comparison of four models achieving it through performance metrics, security measures etc. Model 1 demonstrates its superiority amongst model 2, model 3, and model 4 by having an accuracy of 85% as against their accuracies of 82%, 78%, and 64%, respectively. All the four models indicate the exact test time of 784 nanoseconds, however. Along the side channel attack mitigation, Model 4 is the one that got the highest rating, while Models 1 and 2 got, a lower rating, and Model 3 has medium.

When dealing with resource consumption, Models 1, 3, and 4 are considered as being high consumers, when it comes to resources, as they exceed 100% of available resource on the board. this is specific to spartan FPGA 6 board. Any higher version can accomidate the code. whereas Model 2 is in a medium level of resource consumption.

Metric	Model 1	Model 2	Model 3	Model 4
Number of Slice Registers	223 (1%)	147 (1%)	208 (1%)	208 (1%)
Number of Slice LUTs	38,941 (83%)	31,943 (68%)	113668 (243%)	51319(110%)
Number used as logic	28,601 (61%)	23,349 (50%)	109330 (234%)	49586 (106%)
Number used as Memory	10,340 (93%)	6,253 (56%)	4338 (39%)	1733(15%)
Number of bonded IOBs	6 (1%)	6 (1%)	6 (1%)	6 (1%)
Number of BUFG/BUFGMUXs	1 (6%)	1 (6%)	6 (1%)	6 (1%)
Number of DSP48A1s	116 (87%)	116 (87%)	116 (87%)	6 (4%)

Table 8.2: Comparison of Unmasked and Masked LSTM Networks with Device utilization metrics

The detailed resource usage for each model is displayed in Table 8.2. The Figure 8.1 graph also compares the same metrics as the table. It is clear that masking increases number of computations and resource overhead but with some more approximations the model can be mapped on to the device. The simulation works as expected for all the four models .

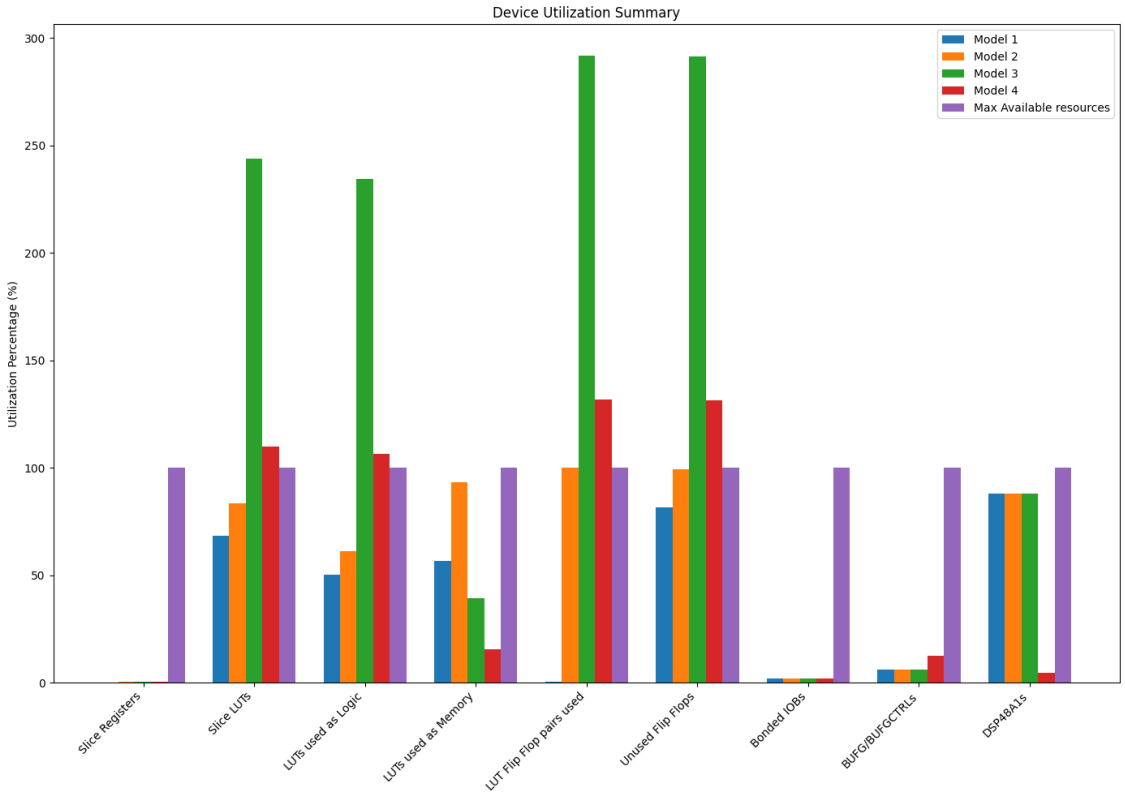

Figure 8.1: Device Utilization summary Of Different Models.

In Figure 8.2 the blue line represents the value of the classified digit which is obtained from the MNIST testing data. The orange line is the values computed from the Model 3 LSTM. You can see the majority of the value coincides with each other providing an accuracy of 78%.

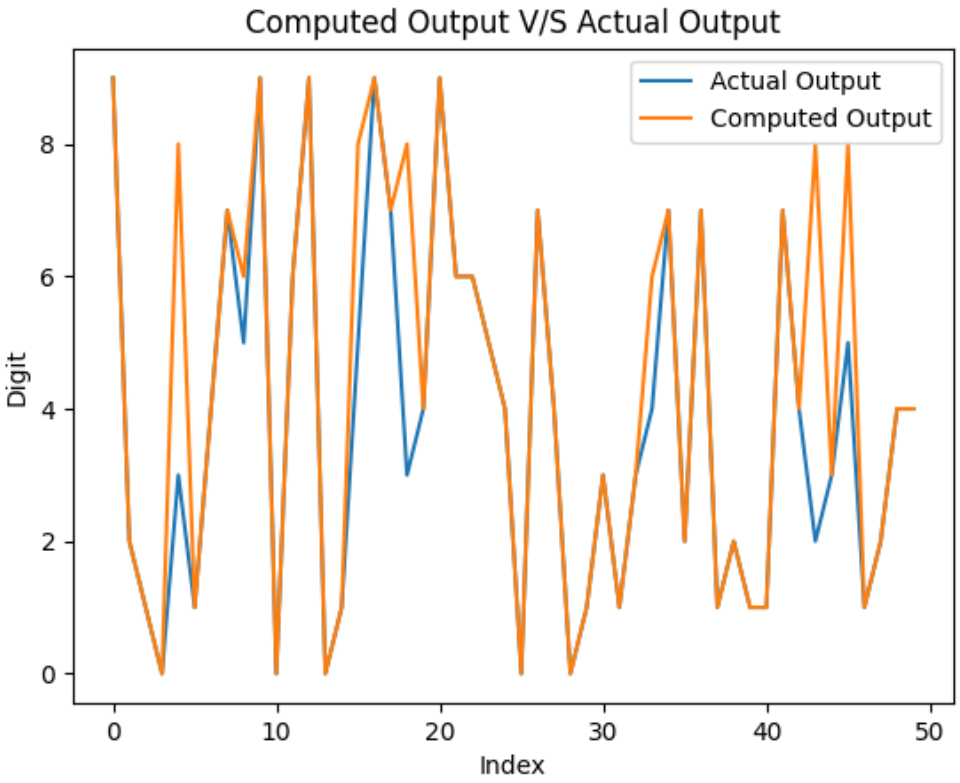

Figure 8.2: Camparison between Actual Output against Computed output .

The accuracy of all the models can be incresed by using floating point numbers instead of using fixed point numbers. Floating points can be used to represent values in higher degree of accuracy, ranging from very small to very large and there-by overcoming the problem of round-off error. These benefits proved to be extremely helpful in this field that demanded high dynamic range, such as LSMT computations, FCNN Convolutions, and activation function.

8.2 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 process[DAP+21].

The suggested method directly applies masking to the weights. The neural network model's weights are the key to our model's mystery. These problems are resolved by directly masking the weights, which reduces resource overhead in comparison to the masking used 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 two layer 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.

9 Conclusion and Future Work

9.1 Conclusion

Both masked and unmasked LSTM neural networks were implemented and analyzed in this thesis through tasks related to the identification of digits. Two primary methodologies were explored: an unmasked LSTM and a masked LSTM that is meant to improve or rather enhance privacy and security while the information is being processed.

When scores in accuracy and efficiency were analyzed, the raw unmasked LSTM cell was found to be superior to its masked counterpart. But the results of using the masked LSTM is still quite feasible in addressing the problem of data leakage especially in cases where privacy is an important issue. The LSTM was also observed to result in some extra computation associated with the masking operations.

One of the main issues that arose during the work was achieving both a high predicting capability and a high level of privacy preservation. The masking technique though useful in boosting security proved to reduce the capacity of the model to generate accurate details about conditions within the data set. The performance of the masked LSTM was severely lacking when compared to its unmasked counterpart.

Masked LSTM was a more complicated sequence model that introduced numerous changes to the typical LSTM. This entailed the addition of software masking layers and hardware masking layers. Although these changes are necessary to protect privacy, they also increase the amount of resources used and the amount of time needed to train the models.

The unmasked LSTM is ultimately shown to be the superior choice when it comes to processing jobs that only focus on raw performance results, while the masked LSTM is still a useful model when it comes to data privacy, following a comparison of the various models in this study. This work provides the foundation for future research in secure machine learning and addresses the feasibility and opportunities of using privacy-preserving methods for building secure neural network architectural designs.

9.2 Future Work

The implementation that is being provided creates opportunities for more study in the area of LSTM security. There are two broad categories into which this research can be divided. On one side, further work could be done on the LSTM implementation of the scheme to make the model less complex and more effecient. On the other side, there are still some unanswered concerns about the scheme's security.

The main concern with implementation of LSTM is how to minimize the quantity of resources needed. Since the weights occupy the majority of the device memory and the compuations of the weights. The next major issue was regarding implementation of activation function. Due to lack of in built exponential function block in ISE 14.7, the walk around also needs storage space. When the code is implemented on updated boards, using exponential function and creating the formula for activation would be the way to go. Instead of using 65535 long LUT. Therefore further research in this area may be represented by optimizing the modules. The use of external memory can also be a solution, using communication protocols like SPI, I2C and Uart, the FPGA can get the data stored in external memory than storing all the data in the BRAM .

The area to focus next is on security, it is particularly interesting since it involves a lot of randomness and primitive operations in its construction. In a strict sense, the masking LSTM would continue to operate as a effective research field in future period. The risk of machine learning attacks could be looked into from a security perspective. The development of machine learning in computing has continued despite side-channel assaults and and is now a real security risk. There are multiple documented attacks of this kind [HGG20] nevertheless, two potential methods are described. The first strategy makes use of machine learning to execute improved template attacks, while the second strategy goes straight after the masking method.

The possible approach to implement modulonet masking could be done using a Relu activation function for the FCNN neural network. This is possible because of the lack of feedback in the FCNN circuit and can check the effeciency of Modulonet on FCNN. Using a state machine model is an additional approach to implementing the LSTM model. State machine optimization allows for the implementation of mapping on FPGA. The complete execution will take more cycles but will use less resources .

Bibliography

[00] IEEE Standard VHDL Language Reference Manual.IEEE Std 1076,page no 157 - 172, 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.

[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).

[CDGK21] H. Chabanne, J. Danger, L. Guiga, and U. Kühne, SSide channel attacks for architecture extraction of neural networks,"*CAAI Trans. Intell. Technol.*, vol. 6, pp. 3-16, 2021. URL: https://doi.org/10.1049/CIT2.12026

[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.

[CLB+21] L. Chen, S. Li, Q. Bai, J. Yang, S. Jiang, and Y. Miao. "Review of Image Classification Algorithms Based on Convolutional Neural Networks."*Remote Sens.*, 13 (2021): 4712. URL: https://doi.org/10.3390/rs13224712.

[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.

[DCA20] A. Dubey, R. Cammarota, and A. Aysu, "BoMaNet: Boolean Masking of an Entire Neural Network,"*2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1-9, 2020. URL: https://doi.org/10.1145/3400302.3415649

[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*, pg - 9, 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.

[HGG20] Hettwer, B., Gehrer, S., Güneysu, T. “Applications of machine learning techniques in side-channel attacks: a survey.” In: Journal of Cryptographic Engineering 10 (June 2020). DOI: 10.1007/s13389-019-00212-8 (cit. on p. 117).

[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/.

[KBBM16] R. Kruse, C. Borgelt, C. Braune, S. Mostaghim, and M. Steinbrecher. Ïntroduction to Neural Networks."(2016): 9-13. URL: https://doi.org/10.1007/978-1-4471-7296-3_2.

[L24] Lenail, A. *AlexNet Architecture*. 2024. Available at: https://alexlenail.me/NN-SVG/AlexNet.html.

[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.

[LZZZ20] X. Li, H. Zhong, B. Zhang, and J. Zhang. Ä General Chinese Chatbot Based on Deep Learning and Its Application for Children with ASD."*International Journal of Machine Learning and Computing*, 10 (2020): 519-526. URL: https://doi.org/10.18178/ijmlc.2020.10.4.967.

[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. *Activation Functions Neural Networks a quick and complete guide*. 15 Mar, 2024. URL: https://www.analyticsvidhya.com/blog/2021/04/activation-functions-and-their-derivatives-a-quick-complete-guide/.

[P24] L.Panneerselvam. 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.

[SK15] D.Surya and R. Krishnaveni. Ä novel method for face recognition using neural networks with optical and infrared images."(2015): 1-5. URL: https://doi.org/10.1109/ICIIECS.2015.7192998.

[SSB23] Shinde, A., Bhoir, P., Shinde, S., Shaikh, B. *An Efficient Ridesharing Model using Machine Learning Based on Riders Reviews*. In: 4th International Conference for Emerging Technology (INCET), Belgaum, India, May 26-28, 2023. Available at: https://example.com/ridesharing-model-paper.

[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.