Copyright

by

Jim Xavier

2020

The Report Committee for Jim Xavier
Certifies that this is the approved version of the following Report:

RTL Design and Analysis of Softmax Layer in Deep Neural Networks

APPROVED BY
SUPERVISING COMMITTEE:

Lizy Kurian John, Supervisor

Nur Touba

RTL Design and Analysis of Softmax Layer in Deep Neural Networks

by

Jim Xavier

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2020

Acknowledgements

I would like to express my gratitude towards Dr Lizy Kurian John for valuable guidance and supervision during the course of this work. I would also like to thank Dr Nur Touba for agreeing to be the reader for my report. I also want to thank Dr Earl Swartzlander for inputs regarding implementation specifics.

Abstract

RTL Design and Analysis of Softmax Layer in Deep Neural Networks

Jim Xavier, M.S.E

The University of Texas at Austin, 2020

Supervisor: Lizy Kurian John

Deep neural networks (DNNs) are widely used in modern machine learning systems in the big data era for their superior accuracy. These artificial neural networks suffer from high computational complexity. The structure of DNN layers vary depending on the nature of training and inference tasks. Softmax Layer is a critical layer in DNNs and is usually used as the output layer in multi-category classification tasks. Softmax layer involves exponentiation and division, thereby resulting in high computational complexity and long critical paths. This report focuses on frontend implementation of an efficient microarchitecture of Softmax layer, which tries to address some of the problems associated with a simple, direct implementation. Techniques like pipelining are employed to boost the performance of the complex datapath logic. Error analysis of the hardware is performed with software results from MATLAB. Synthesis of the RTL code is performed on Xilinx Artix-7 FPGA, resulting in a clock frequency of 274.3 MHz.

Table of Contents

List of Tables vii

List of Figures viii

1. INTRODUCTION 1

  1.1. Softmax Layer 2

  1.2. Direct Implementation 2

    1.2.1 Division Problem 3

    1.2.2 Exponent Problem 3

    1.2.3 Overflow Problem 3

2. ARCHITECTURE 5

  2.1. Exponent Calculation Unit 6

    2.1.1 Taylor Series Architecture 6

    2.1.2 Linear Interpolation with 1 LUT 8

    2.1.3 Linear Interpolation with 2 LUTs 10

  2.2. Adders 11

  2.3. Multipliers 13

  2.4. Divider 15

  2.5. AXI Interface 16

    2.5.1 AXI Read Transaction 18

    2.5.2 AXI Write Transaction 18

3. RESULTS 20

  3.1. Error Analysis of Exponent Calculation Unit 20

  3.2. Error Analysis of Softmax Layer 22

  3.3. Synthesis Results 23

4. CONCLUSION 25

References 26

List of Tables

Table 3.1: Error Analysis for Exponent Calculation Unit 22

Table 3.2: Error Analysis for Softmax Layer 23

Table 3.3: Area and Speed of Exponent Calculation Unit designs............24

List of Figures

Figure 1.1: Basic model of a neural network with activation function 1

Figure 2.1: Hardware Architecture for Softmax Function 5

Figure 2.2: Exponential Function using Taylor Series Expansion 7

Figure 2.3: Block Diagram of Linear Interpolation with 1 LUT 9

Figure 2.4: Block Diagram of Linear Interpolation with 2 LUTs 10

Figure 2.5: Stages of operation of parallel prefix adders 12

Figure 2.6: Layout and Schematic of Kogge Stone Adder 12

Figure 2.7: Stages of Multiplication 13

Figure 2.8: Wallace Tree Multiplication with 16-bit x 16-bit numbers 14

Figure 2.9: Pipeline Diagram of divider unit 15

Figure 2.10: AXI Channels 17

Figure 2.11: Timing Diagram of AXI Read Transaction 18

Figure 2.12: Timing Diagram of AXI Write Transaction 19

Figure 3.1: Error Analysis of Exponent Calculation Unit 21

Figure 3.2: Error Analysis of Softmax Layer 23

1. INTRODUCTION

Deep Learning is a rapidly expanding research field with immense potential for application in image classification, natural language processing and artificial intelligence. A Deep Neural Network (DNN) is an artificial neural network with multiple layers between input and output layers. Deep neural networks contain a large number of interconnected networks, contrary to traditional neural networks. Softmax function is an activation function typically used in the final layer of DNN classification tasks. The function turns numbers into logic probabilities that sum to one. Softmax function outputs a vector that represents the probability distributions of a list of possible outcomes. Softmax function contains expensive mathematical functions, thereby causing the hardware design to suffer from high complexity, long critical paths and overflow problems. Traditional activation functions can be accomplished by simple addition and multiplication operations.

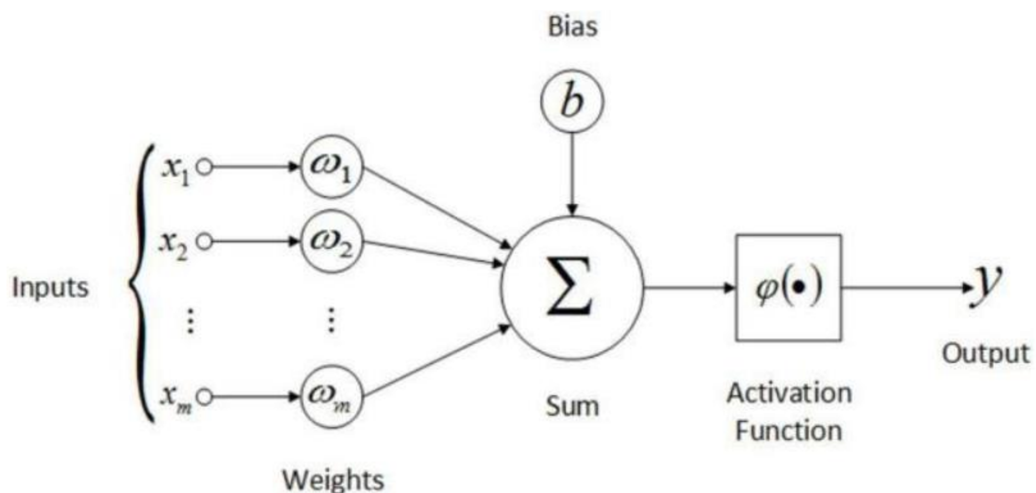

Figure 1.1: Basic model of a neural network with activation function

1.1. Softmax Layer

Different from traditional activation functions that can be used in any position of deep neural networks, softmax layer is typically used as the output layer of the entire neural network. Softmax layers are especially useful in multi-category classification tasks, which is a very fundamental need in many practical pattern detection tasks such as digit recognition and character recognition. Specific hardware architecture designs for machine learning applications have gained significant popularity in recent times. Customized hardware accelerators can provide significant improvements in training times, inference times and power consumption across multiple workloads. Specialized hardware designs for pooling layer, fully connected layer and convolutional layers have significantly improved the computation efficiency of deep neural networks. The hardware architecture of the above-mentioned layers can be mapped directly to simple mathematical operations like multiplication, addition and shifting.

In general, for a N-category classification task, the number of neurons in the softmax layer is N. Taking a set of input numbers, the natural exponent of each input is calculated first and the output of each input is the ratio of its natural exponent to the sum of all natural exponents. Mathematically, the calculation of the *i*-th neuron in the softmax layer can be represented as

$$\frac{e^{xi}}{\sum_{j=1}^{N} e^{xj}} \tag{1}$$

1.2. Direct implementation

Direct mapping of Softmax function to simple mathematical operations is not possible because of some complications.

1.2.1. Division Problem:

Dividers are usually implemented using a large Lookup table (LUT) technique that occupies large chip area and long critical paths. The LUT based Divider is sensitive to the accuracy of the denominator. The quantization error of the denominator may cause high accuracy loss to the output of the divider. This makes the division operation a bottleneck in the high-performance design of Softmax Layer.

1.2.2. Exponent Problem:

A straightforward design requires N instantiations of exponent calculation unit. Depending on the implementation style of exponent function, the exponent calculation module will also occupy a large fraction of chip area. The accuracy loss introduced in the exponent calculation module is also capable of propagating through the design serially, resulting in larger errors at the end output.

1.2.3. Overflow Problem:

The range of inputs to the exponent calculation module can be very large because they can be possible inner products of intermediate signals. In practical designs, the quantization precision and range of inputs to the exponent calculation module is constrained by hardware resources. This leads to a potential overflow problem in the design of Softmax Layer function.

An efficient and practical implementation of Softmax hardware must take care of factors such as data preprocessing, efficiency of exponent and division operations, as well as storage of division and exponent calculation results.

2. ARCHITECTURE

The microarchitecture of reference for this project report is depicted in Figure 2.1. The input and ou[1]tput data are in 16-bit fixed point format. The exponent calculation results are stored in 32-bit fixed point format. The summation of all exponent result values is represented in 52-bit fixed point format for high precision and truncated to 36 bits. The truncation is performed by taking the higher 36 bits of the result. 4096 input values are used for analysis and a 16kB output FIFO would be required to store the results from the Exponent Calculation Unit. Two small asynchronous FIFOs are used at the input side to account for stalls encountered from external memory side and to implement ping-pong operation while streaming data.

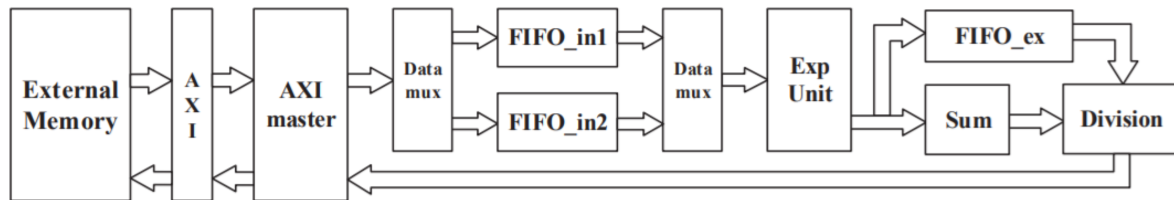

Figure 2.1: Hardware Architecture for Softmax function

AXI master-slave interface is used for transmitting data between the external memory and the input side FIFOs. AXI interface enables high performance and bandwidth without any loss of data. Data demultiplexers are used to split the data and store them between FIFO_in1 and FIFO_in2. A Data multiplexer selects the data between

[1] Portions of the chapter have been previously published in [1]. This architecture is used as the reference implementation for my design with further optimizations and improvements included

FIFO_in1 and FIFO_in2 in an interleaved manner and feeds inputs to the Exponent calculation unit. The Exponent calculation unit calculates the exponent of the input and feeds it to the FIFO_ex and Summation modules. FIFO_ex is used to store the calculated exponent results of the inputs. The Summation unit accumulates the results of all exponents calculated till the N-*th* iteration. Upon calculating the exponent value for all input data, the final sum is obtained and division operation commences to calculate the Softmax output. The Divider output is of 16-bit width. The AXI master-slave interface is responsible for sending the Softmax output into the external memory.

2.1. Exponent Calculation Unit

The Exponent Calculation Unit is reponsible for the exponent function e^x, which is a vital component of softmax implementaton. Three different architectures are evaluated for Exponent function computation. The architecture giving the lowest error values as compared to the software model of exponent would be chosen for the final softmax layer implementation.

2.1.1. Taylor Series Architecture

Taylor series [3] is a series expansion of a function about a point. A one-dimensional Taylor series is an expansion of a real function f(x) about a point x=a. It is given by

$$f(x) = f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2!} + f'''(a)\frac{(x-a)^3}{3!} + \cdots f^n(a)\frac{(x-a)^n}{n!} + \cdots \quad (2)$$

The exponential function e^x can be represented using Talyor series expansion for the first 8 terms as below:

$$e^x = e^a(1 + (x-a) + \frac{1}{2}(x-a)^2 + \frac{1}{6}(x-a)^3 + \frac{1}{24}(x-a)^4 + \frac{1}{120}(x-a)^5 + \frac{1}{720}(x-a)^6 + \frac{1}{5040}(x-a)^7 \tag{3}$$

By factoring out (x-a) from all terms and rearranging their order, we get:

$$e^x = e^a\left[1 + (x-a)\left(1 + (x-a)\left(\frac{1}{2} + (x-a)\left(\frac{1}{6} + (x-a)\left(\frac{1}{24} + (x-a)\left(\frac{1}{120} + \frac{1}{720}(x-a)\right)\right)\right)\right)\right)\right] \tag{4}$$

The hardware for designing exponential function using Talyor series expansion is designed using Equation (4). The design is depicted in Figure 2.2 and it uses 7 adders and 7 multipliers. Two's complement arithmetic is used for fixed point subtraction. 32-bit adders and 24-bit multipliers are used for the implementation. The specifications of the adders and multipliers used for the design are discussed later in this report.

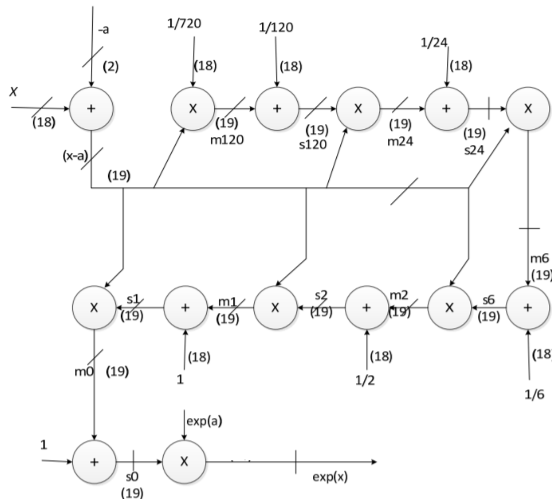

Figure 2.2: Exponential Function using Taylor Series Expansion

2.1.2. Linear Interpolation with 1 LUT

Linear interpolation based architecture [3] is implemented to accomplish the exponent function. The architecture requires the use of Lookup Table (LUT) to store the pre-calculated values. The starting point of the implementation of exponent using linear interpoloation is the below equation:

$$y(n) = y_b(n) + \left(\mathrm{x(n)} - \mathrm{x_b(n)}\right) * \frac{(y_b(n+1) - y_b(n))}{(x_b(n+1) - x_b(n))} \tag{5}$$

Since the Softmax layer implementation already involves division operation, it would be better to avoid divider logic in the exponent calculation module. Assuming there are fixed number of intervals in the architecture, the denominator in equation (5) is always a fixed fractional number. This will help turn the division operation into a multiplication operation. Assuming the denominator to be a fraction *v*,

$$y(n) = y_b(n) + \left(\mathrm{x(n)} - \mathrm{x_b(n)}\right) * v * (y_b(n+1) - y_b(n)) \tag{6}$$

If we substitute $\left(\mathrm{x(n)} - \mathrm{x_b(n)}\right) * v$ as f(n) in equation (6), we get

$$y(n) = y_b(n) + f(n) * (y_b(n+1) - y_b(n)) \tag{7}$$

Equation (7) is used for the implementation of exponential function using linear interpolation architecture. The design takes x as input and y(n) as output. The design involves the usage of 1 LUT, adders and multipliers.

In this architecture, a single LUT is used to store 256 intervals of exponent function. To index into the LUT, 8 bits of the input signal is required. Each location of the LUT stores 18 bits of each datapoint. Figure 2.3 shows the block diagram of the

implementation of the architecture using 1 LUT. The values of the intermediate intervals are computed using MATLAB and stored in the LUT.

From the input word x of word length 16 bits, 8 most significant bits are used fetch the base value and the next base value. The next base value is fetched out to evaluate the gradient difference. The gradient difference is subsequently multiplied with the 8 least significant bits of word x. The base value and gradient difference are evaluated using a single LUT in this architecture. The hardware consists of a 256 x 18 LUT, 2 adders and a multiplier.

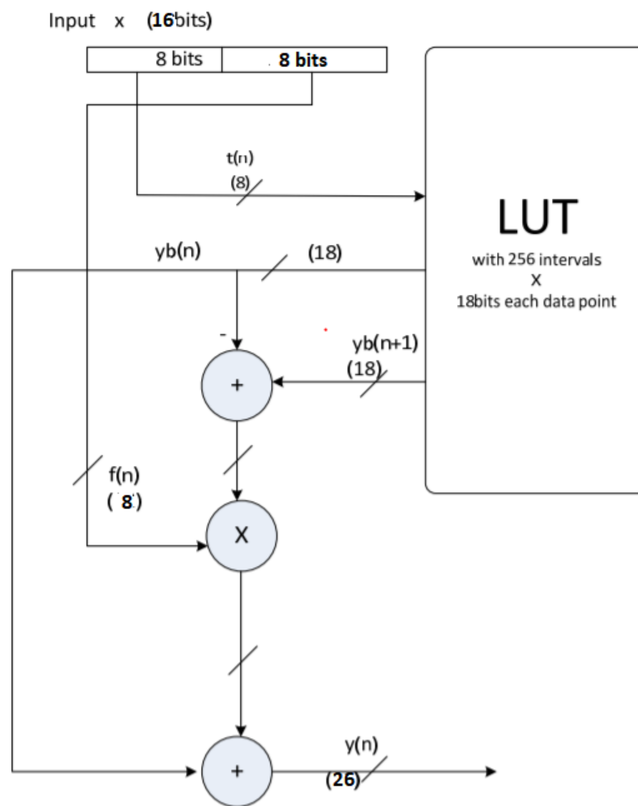

Figure 2.3: Block diagram of Linear interpolation with 1 LUT

2.1.3. Linear Interpolation with 2 LUTs

To reduce the latency of the architecture and also improve precision, two LUTs were designed to separately index the base value and gradient difference.

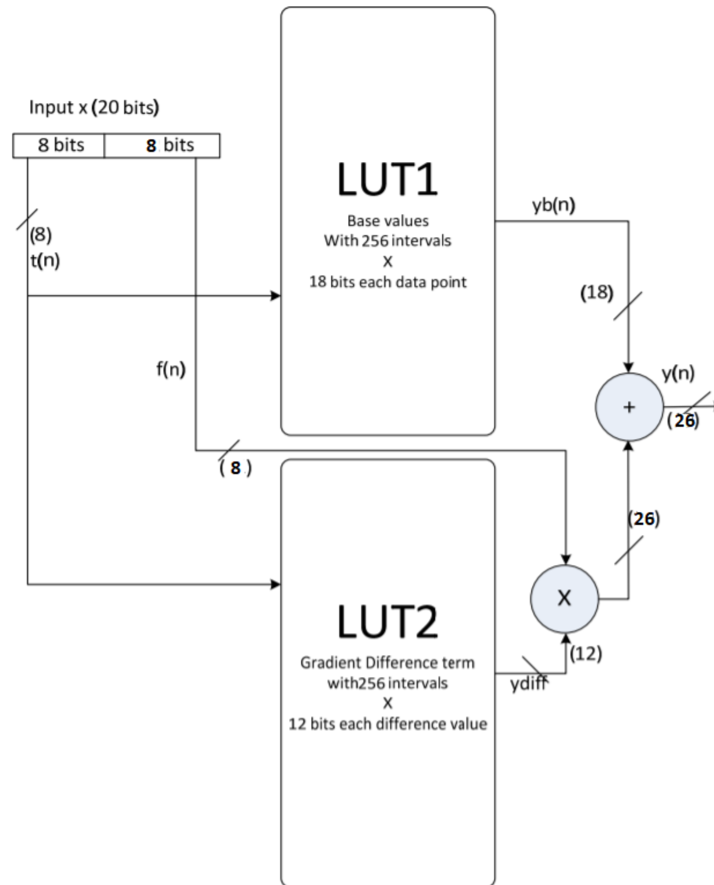

Figure 2.4: Block diagram of Linear interpolation with 2 LUTs

- In the first LUT, base value is stored. Each value has a precision of 18 bits.
- In the second LUT, gradient difference is stored. Each value is stored with a precision of 12 bits.

As shown in Figure 2.4, the implementation involves the use of 2 LUTs, 1 multiplier and 1 adder. The input word x has a word length of 16 bits. The most significant 8 bits are used to index into 2 LUTs to address the base value and gradient difference. The remaining 8 bits are used to multiply with the gradient coefficient to generate gradient difference, which has a word length of 19 bits. The base value is added to the gradient difference to form the final result. The word length of the output is 26 bits. This architecture uses 1 less adder in the critical path as compared to the linear interpolation architecture with a single LUT. Hence the critical path delay is reduced, and the design is capable of achieving higher clock frequency.

2.2. Adders

The adder falls in the critical path of Exponent Calculation Unit as evident from Figure 2.3 and Figure 2.4. For the purpose of this project, a high-speed parallel prefix adder unit is chosen for the adder design. Parallel prefix adders typically operate in 3 stages as shown in Figure 2.5

- Pre-processing stage
- Carry generation stage
- Post-processing stage

In the pre-processing stage, the carry propagate and generate stages are evaluated. Using the evaluated P_i and G_i signals, the carry generation stage evaluates the carry required for post-processing using Black-cell and Gray-cell logic. The post processing

stage calculates the Sum output and Carry output using the signals generated from the previous stages.

The parallel-prefix adder of choice for this project tis Kogge Stone Adder. It is a parallel prefix form carry look-ahead adder (CLA) and is one of the fastest adder architectures available. The Kogge-Stone Adder [6] typically takes more area to implement than other parallel-prefix adder architectures, but has a lower fanout at each stage, thereby boosting the performance in terms of critical path delay. The schematic and layout of the Kogge Stone adder designed is depicted in Figure 2.6.

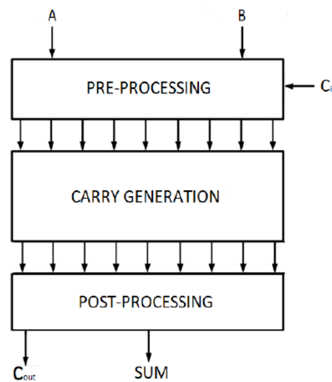

Figure 2.5: Stages of operation of Parallel prefix adders

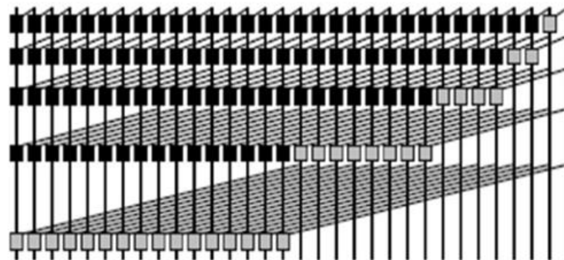

Figure 2.6: Layout and Schematic of Kogge Stone Adder

2.3. Multipliers

Multiplier also falls in the critical path of the Exponent Calculation Unit. Hence, there arises a need to design a high-speed multiplier to ensure high performance of Exponent Calculation Unit. The multiplier of choice for the project is Wallace Tree Multiplier [7] with tree reduction employed. Fast multipliers like Wallace tree mulitplier typically operate in 3 steps (shown in Figure 2.7)

- Partial Product generation
- Partial Product Accumulation
- Final Addition

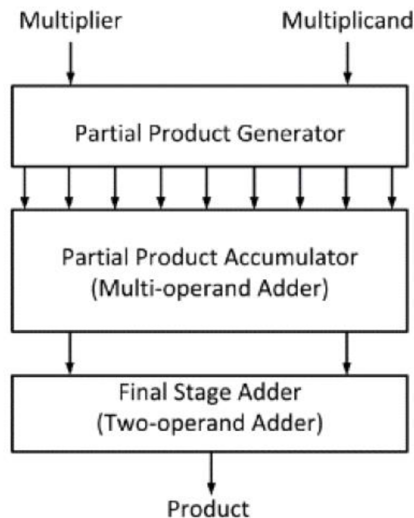

Figure 2.7: Stages of Multiplication

In the first stage, each bit of one operand is ANDed with the bits of the other operand to generate the partial products. Using tree of Half Adders and Full Adders,

partial product accumulation is performed as shown in Figure 2.8. The partial products are basically reduced to two by using the layers of Half Adders and Full Adders. The algorithm for partial product accumulation works as follows-

1. Take any three wires with the same weights and input them to a full adder, since a full adder can perform 3-bit addition. The result will be an output wire of the same weight and an output wire with a higher weight for each of the three input wires.
2. If there are two wires with the same weights, input them to a half adder.
3. If there is just one wire left, connect it to the next layer.

In the Final addition Stage of Wallace Tree multiplier, all the output wires are grouped into two numbers are added with a conventional adder. In this project, the conventional adder used for the final stage of multiplication is a Kogge-Stone Adder.

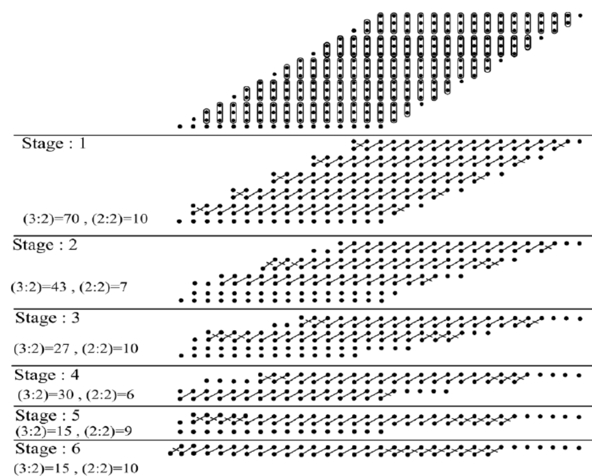

Figure 2.8 : Wallace Tree Multiplication of 16-bit x 16-bit numbers

2.4. Divider

The divisor for the divider module is a 52-bit number, which is the summation of all the exponents and the dividend is the 32-bit exponent that is read from the FIFO_ex module. Since the dividend is smaller than the divisor, it is safe to set the output to be a 16-bit signal. The division algorithm used for the design is described in [1], which is an eight-stage pipelined design. This algorithm is an enhanced form of other standard binary division algorithms like SRT Division [4], Restoring Division and Non-Restoring Division [5]. There also exist traditional division algorithms which involve left shifting the dividend and performing division operation by bit-shifting. To reduce the hardware resource usage without deteriorating accuracy, the architecture right-shifts both the dividend and divisor by 16 bits.

$$\frac{\{a[31:0], 16'b0\}}{b[51:0]} \approx \frac{a[31:0]}{b[51:16]} \tag{8}$$

The design is pipelined to enhance the performance of the Softmax layer as a whole. The pipeline stages of the divider are depicted in Figure 2.9.

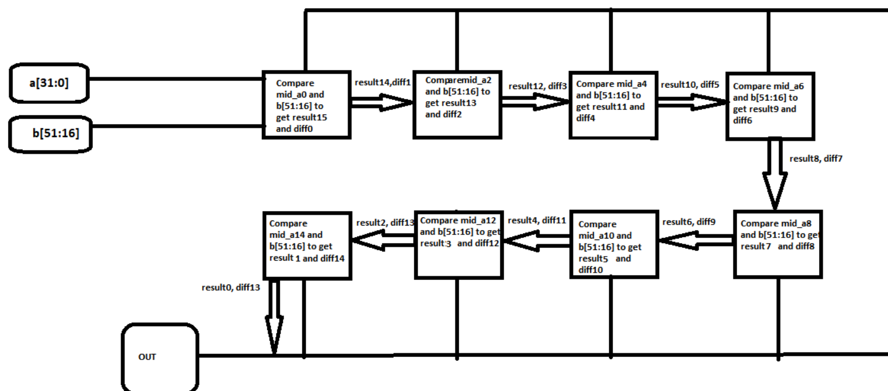

Figure 2.9: Pipeline diagram of divider unit

The divider unit is designed using comparators and shifters as basic building blocks. Each stage compares and shifts the values of mid_a* and the 36 bits of the divisor b[51:16]. This computes the result and diff bits corresponding to that stage and propagates the result and diff bits required for the next pipeline stage. This is continued for the 8 stages and the final result is a concatenation of the results from individual stages. This result is stored in OUT register.

2.5. AXI Interface

The Advanced Extensible Interface (AXI) is an important protocol present as part of the AMBA specification for System on Chip Design. AXI uses multiple, dedicated channels for performing READ and WRITE operations between Masters and Slaves. AXI is capable of handling burst-based data transfers through the channels. AXI also includes several additional features like out-of order transactions, unaligned data transfers, cache support signals and a low power interface.

There are five independent channels in AXI between Master and Slave:

- Read Address channel
- Read Data channel
- Write Address channel
- Write Data channel
- Write Response channel

The address channels are used to send address and control information while performing a basic handshake between the Master and Slave. The data channels are

where the information to be exchanged is placed. A Master reads data from and writes data to a Slave. Read Response channel is placed on the Read Data channel, whereas Write Response has an additional dedicated channel. This dedicated channel is used by the Master to verify if the transaction completed successfully.

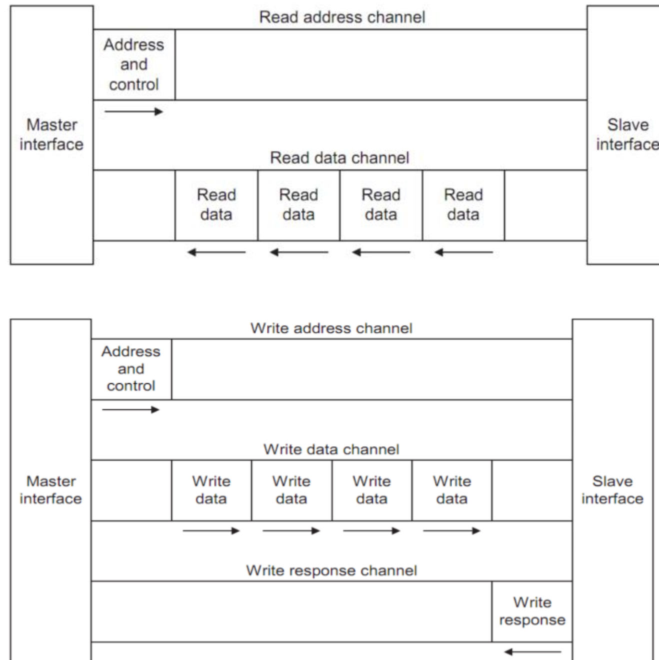

Figure 2.10: AXI channels

An AXI Interconnect is used to manage transactions between AXI Master and Slave. This project designs an AXI Interconnect and an AXI Master interface to control transactions between the Softmax Layer and External Memory. The Interface designed is a basic protocol and is not equipped to handle burst transfers or out of order transactions. The Interconnect needs to handle just one Master and Slave according to the requirements.

2.5.1 AXI Read Transaction

Figure 2.11 shows the basic timing diagram of a Read transaction between an AXI Master and Slave. To start the transaction, the Master places the Slave's address on ARADDR line and asserts that there is a valid address on the line (ARVALID). After time T1, the Slave asserts the ready signal (ARREADY). For a transfer to occur, both ARVALID and ARREADY need to be asserted. This happens on the read address channel, with the address transfer completing on the rising edge of cycle T2. When the Master is ready to accept data from the Slave, it asserts the RREADY signal on the read data channel. The Slave then places the data on the RDATA line and asserts that there is a valid data on the line RVALID. For a read transaction, the slave is the source and master is the receiver.

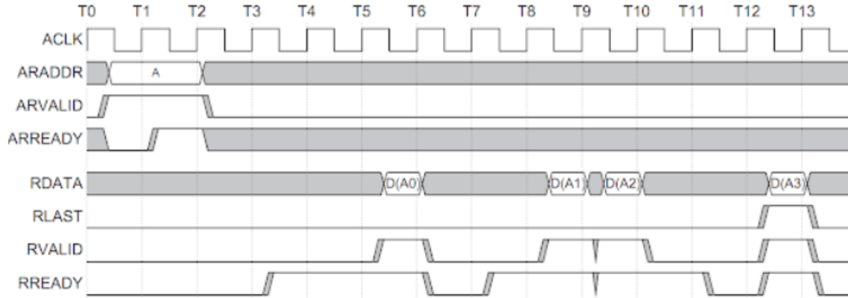

Figure 2.11: Timing Diagram of AXI Read transaction

2.5.2 AXI Write Transaction

Figure 2.12 shows the timing diagram of a Write transaction between an AXI Master and Slave. The addressing sequence is similar to a Read transaction. A master places an address ARADDR and asserts a valid signal ARVALID. The slave asserts

ARREADY to handshake that it is ready to receive the address and the address is transferred. After this is completed, the master places data on the bus in the Write Data channel and asserts the data valid signal WVALID. When the slave is ready to receive data, it asserts the WREADY signal and the data transfer begins.

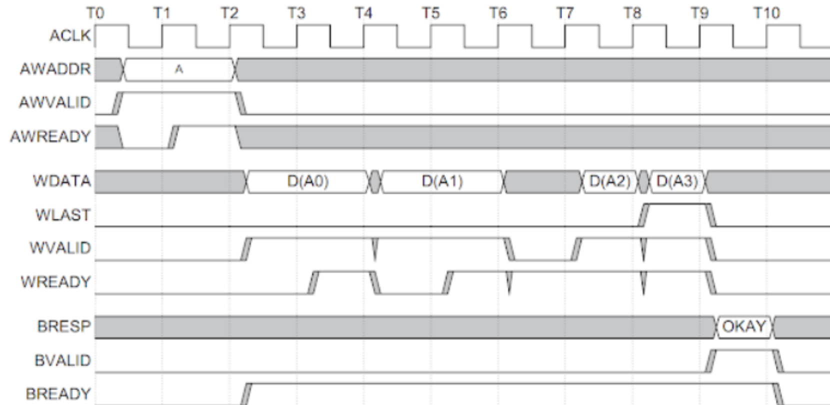

Figure 2.12: Timing Diagram of AXI Write Transaction

The Master asserts the WLAST signal when the last data is being transferred. In contrast to Reads, Writes use a Write Response Channel where the slave can assert that the write transaction has completed successfully. The AXI arbiter and synchronizer designed in the interconnect takes care of the protocol and the data exchange between the External Memory and Softmax Layer.

3. RESULTS

Verilog HDL is used to implement the arithmetic components of Softmax Function and the AXI Interface modules. Verilog Testbenches are used for functional verification of individual modules. A set of experiments are conducted to gauge the precision and accuracy of the designs. Error analysis is performed on the Exponent Calculation Unit and the Softmax Layer, which instantiates the Exponent Calculation Unit for computation purpose. The Hardware results are collected from MODELSIM simulator and the software results are collected from MATLAB for some values of x (between 0 and 1).

3.1. Error Analysis of Exponent Calculation Unit

Some common causes of precision loss are

- Precision loss introduced by low data width when preprocessing the input data
- Precision loss introduced by low data width for the lookup tables
- Precision loss introduced by the truncation after multiplication

The Error analysis for the three architectures used for implementing the Exponent Calculation Unit are depicted in Figure 3.1.

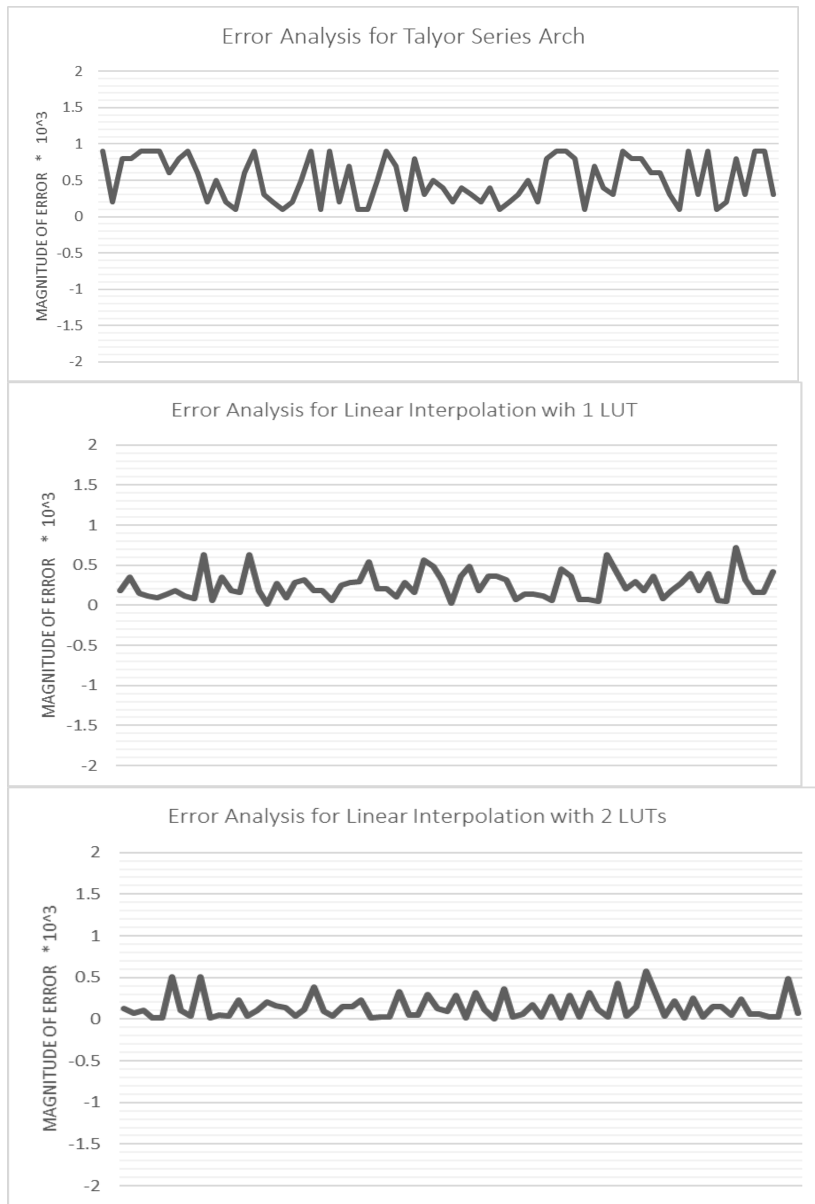

Figure 3.1: Error Analysis of Exponent Calculation Unit

	Taylor architecture	Linear Interpolation 1	Linear Interpolation 2
Avg Error	$0.509772 * 10^{-3}$	$0.247222 * 10^{-3}$	$0.223889 * 10^{-3}$
Max Error	$0.802371 * 10^{-3}$	$0.465733 * 10^{-3}$	$0.465123 * 10^{-3}$

Table 3.1: Tabulation of Max Error and Avg Error for Exponent module

Linear Interpolation architecture with 2 LUTs is giving the lowest Error magnitudes (Max and Avg) for exponent function. Hence, this architecture is used for the design of Exponent Calculation Unit for the Softmax Layer.

3.2. Error Analysis of Softmax Layer

Software result of Softmax function is obtained from MATLAB for 4096 values of input data. For the same input data, hardware simulation results are collected from MODELSIM simulator using the Verilog testbench environment. The output data from hardware is of 16-bit width. These values are compared for each value of x and magnitude of error is calculated using the magnitude of difference between the software and hardware results. The results are plotted in Figure 3.2. The average error observed is $2.7 * 10^{-5}$, which is within acceptable limits for Softmax layer of DNN applications.

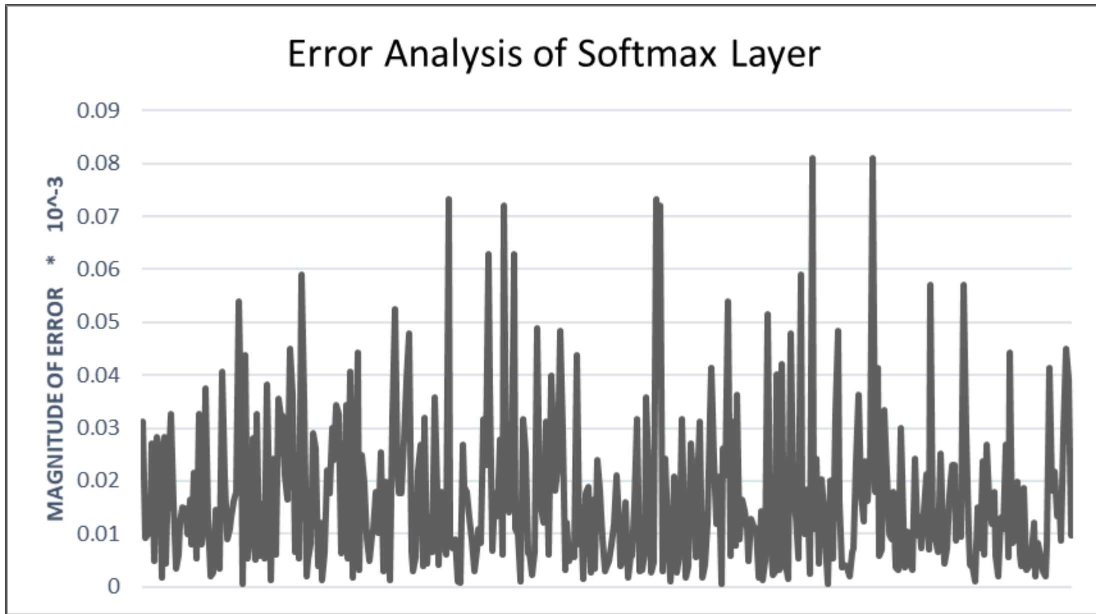

Figure 3.2: Error Analysis of Softmax Layer

	Softmax Layer
Avg Error	$2.7 * 10^{-5}$
Max Error	$8.2 * 10^{-5}$

Table 3.2: Tabulation of Max Error and Avg Error of Softmax layer implementation

3.3. Synthesis Results

All three designs for Exponent Calculation Unit were synthesized on Artix-7 FPGA model XC7A35TCPG236-1. Comparison of the 3 different exponent calculation architectures is tabulated in Table 3.3. Taylor series architecture falls behind the other two architectures because of the large number of adders and multipliers present in the

datapath. It is unexpected that one implementation will fare poorly on all 3 fronts (area, frequency, accuracy). However, the results obtained are consistent with the observations in [3].

Design	Area (Number of Slice LUTs)	Clock Frequency (MHz)
Taylor Series	8143	98
Linear Interpolation 1	5378	323
Linear Interpolation 2	4292	456

Table 3.3: Area and Speed of Exponent Calculation Unit designs

The Softmax Layer design is synthesized on Artix-7 FPGA model XC7A35TCPG236-1. The synthesized netlist uses 1342 slice registers and 7783 slice LUTs on the FPGA. The clock frequency achieved at Synthesis phase for Softmax Layer is 274.3 MHz.

4. CONCLUSION

Efficient hardware implementation of Softmax Layer is essential for enabling embedded DNN applications. The project targets implementation of a high precision, high performance Softmax Layer microarchitecture. For the exponentiation function which generally consumes a lot of chip area, a linear interpolation architecture using LUTs and arithmetic components like Adders and Multipliers is used. This decision is made after performing experiments with alternate implementation options. To boost the performance of Addition and Multiplication operations, high performance datapath components like Kogge-Stone Adder and Wallace-Tree Multiplier are designed. A modified shift-compare divider is implemented, which operates in 8 pipeline stages. For high performance data transfers with external memory, a simple AXI Interconnect and AXI Interface is designed adhering to ARM AMBA AXI specification. The maximum error reached with software comparison is in 10^{-5} scale, thereby adhering to requirements of Softmax function in DNNs. The synthesized netlist achieved a clock frequency of 274.3 MHz on Xilinx Artix-7 FPGA.

References

[1] Zhenmin Li, Henian Li, Ziange Jiang, Bangyi Chen, Yue Zhang and Gaoming Du, "Efficient FPGA Implementation of Softmax Function for DNN applications," Proceedings of IEEE, Nov 2018.

[2] Bo Yuan, "Efficient Hardware Architecture for Softmax Layer in Deep Neural Network," Proceedings of IEEE, Nov 2016.

[3] Ateeq Ur Rahman Shaik, "Hardware Implementation of the Exponential Function using Taylor Series and Linear Interpolation", Master's Thesis, Lund University, May 2014

[4] D. Harris, S. Oberman, M. Horowitz, "SRT division: Architectures, models and implementations," Tech Rep, 1998.

[5] J.E. Robertson, "A new class of digital division methods," IRE Trans, of Elec Comp, Vol. EC-7, No.3 (Sept 1958), pp.218-222.

[6] P. Kogge et al., IEEE Trans. Computers, vol. C-22, p-786 (1973).

[7] C.S. Wallace, "A Suggestion for a Fast Multiplier," IEEE Trans on Electronic Computers, March 1964.

[8] ARM, "AMBA AXI and ACE Protocol Specification," 2019.