LSTM HARDWARE ACCELERATOR

Project Book

Abstract

Project book of an in-depth design of an LSTM hardware accelerator

Adam Shacham & Etay Sela

Contents

1. Introduction

1.1 Introduction of RNN LSTM

Long-Short Term Memory Recurrent Neural Network (RNN LSTM) is a commonly used subset of an Artificial Neural Network (ANN). ANN is a computing system inspired by the biological neural networks in the human brain. In the following chapter we will describe RNN LSTM and its place within the growing field of artificial intelligence.

Definitions of artificial intelligence (AI) and machine learning (ML)

The generally accepted definition of AI is the study of the design of intelligent agents. An intelligent agent is any system whose actions are appropriate to a changing environment and to its goal. The agent learns from past experiences and makes decisions based on them and its computational abilities[1].

ML is a type of AI that allows software applications to become more accurate at predicting outcomes without being explicitly programmed to do so. Machine learning algorithms use historical data as input to predict new output values[2].

Common uses of machine learning include, among many others, well known search engines such as google and YouTube, who use ML to personalize recommendations to their users, as well as self-driving cars, who use ML to identify their surroundings.

A partial list of milestones in the development of ML:

- 1952 - Arthur Samuel creates a program to help an IBM computer get better at checkers the more it plays.
- 1959 - MADALINE becomes the first artificial neural network applied to a real-world problem: removing echoes from phone lines.
- 1985 - Terry Sejnowski's and Charles Rosenberg's artificial neural network taught itself how to correctly pronounce 20,000 words in one week.
- 1997 - IBM's Deep Blue beat chess grandmaster Garry Kasparov.
- 1999 - A CAD prototype intelligent workstation reviewed 22,000 mammograms and detected cancer 52% more accurately than radiologists did.
- 2012 - An unsupervised neural network created by Google learned to recognize cats in YouTube videos with 74.8% accuracy.
- 2014 - A chatbot passes the Turing Test by convincing 33% of human judges that it was a Ukrainian teen named Eugene Goostman.

1.2 Types of machine learning

There are two main approaches for machine learning: deep learning, and shallow learning. Shallow Learning is the process of extracting information using manually defined and predetermined features, while in deep learning the feature extraction is algorithmically computed without manual human intervention.

For example, while trying to teach a machine to differentiate between images of cats and non-cats, a shallow learning approach would define a-priori criteria with which to make a decision, such as ears, whiskers and paws. A deep learning algorithm will itself find points of interest to use in its classification of the images.

Independently, it is generally accepted to distinguish between the following methods: supervised learning, unsupervised learning, and reinforcement learning. The type of algorithm that data scientists choose to use depends on what type of data they want to predict.

Supervised learning: In this type of machine learning, data scientists supply algorithms with labeled training data and define the variables they want the algorithm to assess for correlations. Both the input and the output of the algorithm are specified. The machine evaluates a function called the loss function which describes the difference between the defined output and the received output in order to make better predictions. The goal of this method is to train the model so that it can predict the output when it is given new data. The method is used to tackle 2 types of problems[3]:

- *Classification* uses an algorithm to accurately assign test data into pre-determined categories. It recognizes specific entities within the dataset and attempts to draw some conclusions on how those entities should be labeled or defined. The previously described problem of differentiating between cats and non-cats is an example of a classification problem.
- *Regression* is used to understand the relationship between dependent and independent variables. It is commonly used to make projections, such as for sales revenue for a given business. An example of such a problem would be an estimation of growth of a cat population, given the current and previous populations.

Unsupervised learning[4]: This type of machine learning involves algorithms that train on unlabeled data. The algorithm scans through data sets looking for any meaningful connections and does not use any feedback in the process as only input data is provided. The data that algorithms train on as well as the predictions or recommendations they output are predetermined. Problems who use this method include:

- *Clustering* is the process of organizing objects into groups whose members are similar in some way. An example would be receiving an input of many unlabeled cat

images with no expected output and receiving a result of the cat images grouped according to cat's breeds.

- Association mining: Identifying sets of items in a data set that frequently occur together. For example, given a large sample of cat images, the output {triangular head, almond shaped eyes} → {long neck} would indicate that a triangular head and almond shaped eyes strongly suggest a long-necked cat.

Reinforcement learning[5]: Data scientists typically use reinforcement learning to teach a machine to complete a multi-step process for which there are clearly defined rules. Data scientists program an algorithm to complete a task and give it positive or negative cues as it works out how to complete it. For the most part however, the algorithm decides on its own what steps to take along the way.

- Robotics: Robots can learn to perform tasks in the physical world using this technique.
- Video gameplay: Reinforcement learning has been used to teach bots to play several video games.
- Resource management: Given finite resources and a defined goal, reinforcement learning can help enterprises plan out how to allocate resources.

1.3 Artificial Neural Networks (ANN)

ANN [6]is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives a signal, processes it and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function (called activation function) of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer) to the last layer (the output layer), possibly after traversing the layers multiple times.

1.4 Recurrent Neural Network (RNN)

Recurrent Neural Network [7] is a neural network that has an internal memory in the form of feedback from its output back to itself. A simple block diagram depicting an RNN, and its internal memory can be seen in figure 1.1:

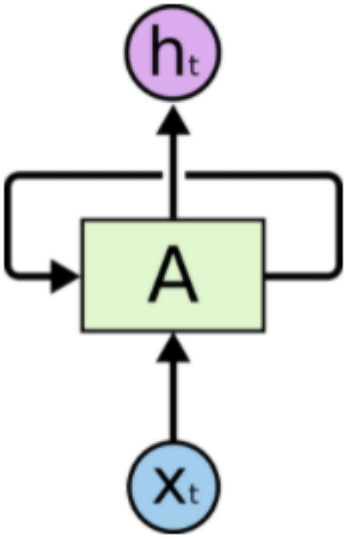

Figure 1.1 - a block diagram of an RNN where x_t is the input, h_t is the output and A being the network

To acquire a better understanding of the network, we can look at each time step individually. This process is called unrolling. A simple unrolled RNN is shown in figure 1.2:

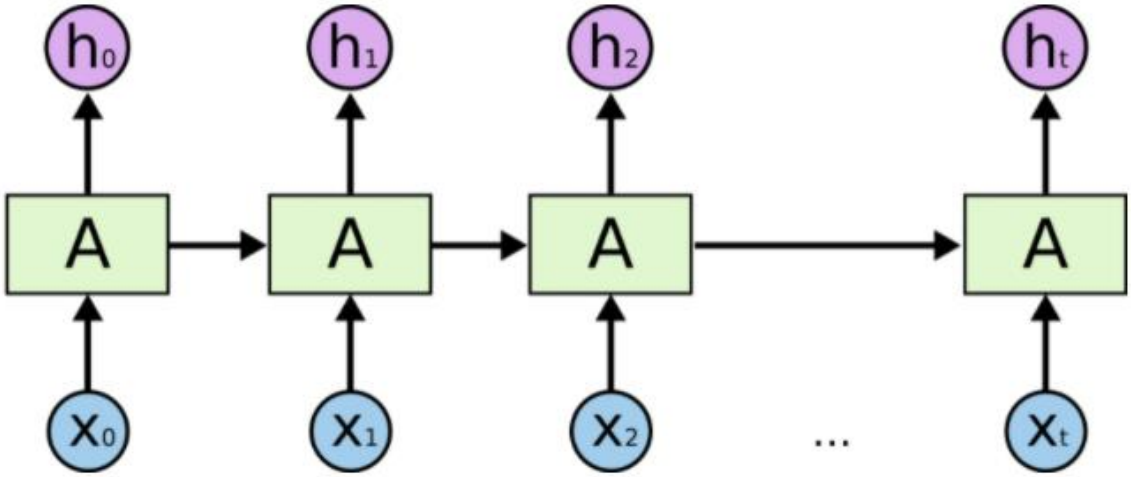

Figure 1.2 - a block diagram of an unrolled RNN, x_i is the i-th input, h_i is the i-th output and A is called an RNN cell and represents the different operation done on the input to achieve the output and the feedback at a given time.

The basic RNN cell at time t is nothing but a tanh function applied to the vector that is the result of the concatenation of the input at time t and the output of time $t-1$. This simple RNN cell model is shown in figure 1.3:

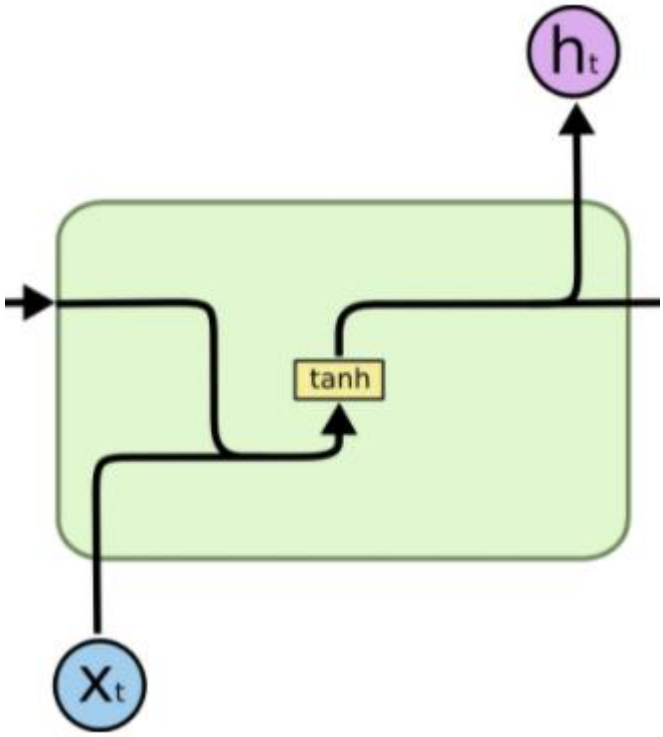

Figure 1.3 - a basic RNN cell

RNN is recurrent in nature as it performs the same function for every input of data while the output of the current input depends on the past one computation. After producing the output, it is copied and sent back into the recurrent network. To decide, the network considers the current input and the output that it has learned from the previous input through a process called backpropagation – an algorithm which computes the gradient of the loss function with respect to the weights of the network.

Thus, RNNs can use their internal state (memory) to process sequences of inputs. This makes them better applicable to solving tasks such as unsegmented, connected handwriting recognition or speech recognition compared to other ANN's. In other ANN's, all the inputs are independent of each other but in RNN, all the inputs are related to each other.

However, a drawback with RNNs is that as time passes by and they get fed more and more new data, they start to **"***forget***"** about the previous data they have seen, as it gets diluted between the new data, the transformation from activation function, and the weight multiplication. This means they have a good *short-term memory*, but a slight problem when trying to remember things that have happened a while ago (data they have seen many time steps in the past). The more time steps there

are, the more chance we have of backpropagation gradients vanishing down to nothing. This problem is commonly referred to as “The vanishing gradient problem”[8]. In addition, the analogues issue emerges as well – information that propagates through many different layers may be multiplied by a factor bigger than one at many iterations and therefore grow uncontrollably. This is referred to as “The exploding gradient problem”.

1.5 LSTM[9]

Long Short-Term Memory (LSTM) networks are a modified version of recurrent neural networks, which makes it easier to remember past data in memory, thus solving the vanishing gradient problem described above. LSTM is well-suited to classify, process and predict time series given time lags of unknown duration. It trains the model by using backpropagation. It is useful to envision LSTM as an augmentation of RNN in which the content of the cell varies. Figure 1.4 shows a basic RNN LSTM cell:

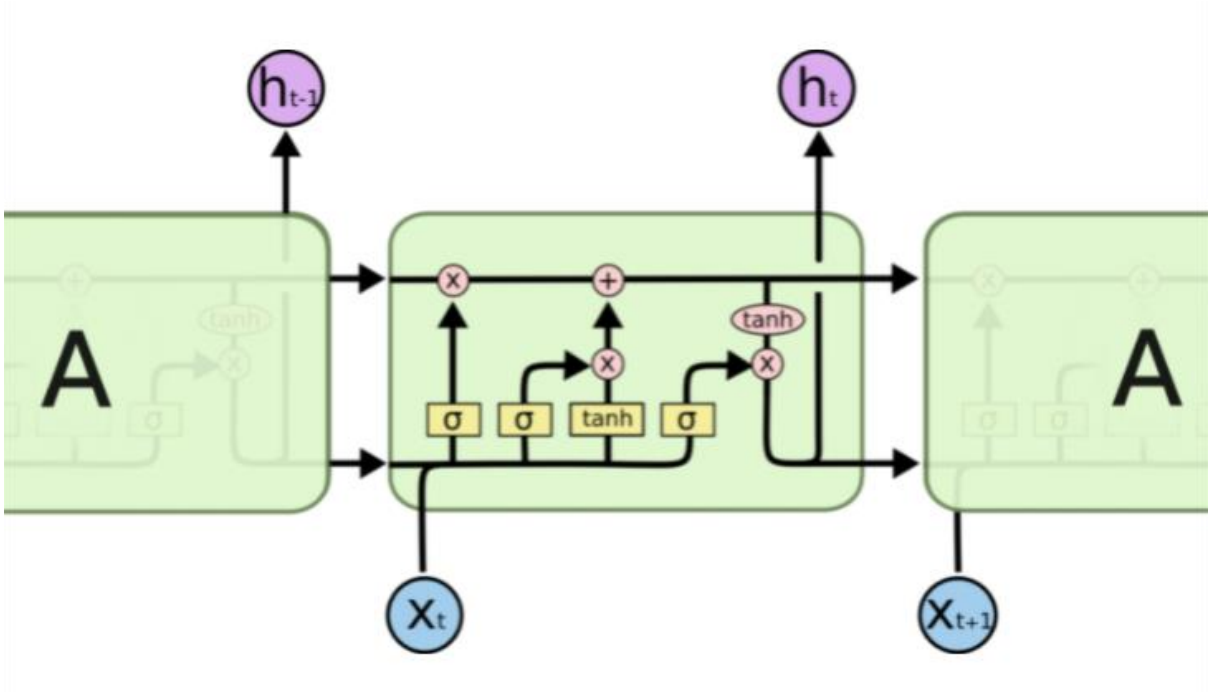

Figure 1.4 - an RNN LSTM cell

The intersections where different routes converge represent a concatenate operation, and the intersections where a route diverges represent a copy operation. The pink circles signify bitwise operation, and the yellow rectangles signify neural network layers and therefore, among other traits, have weights that adjust as the system learns. The sigma represents the sigmoid activation function which matches real numbers to a value between 0 to 1 as seen in figure 1.5:

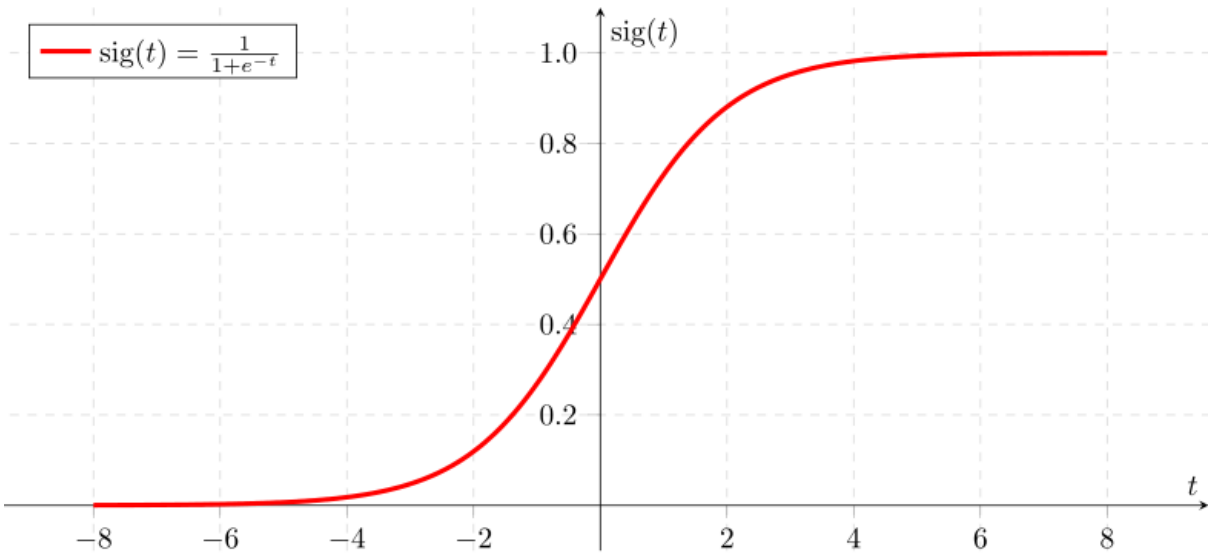

Figure 1.5: sigmoid function

The LSTM cell is comprised of 3 gates. Each gate serves a different function in the cell:

- *Forget gate* decides which memory details should be discarded, by using an activation function.

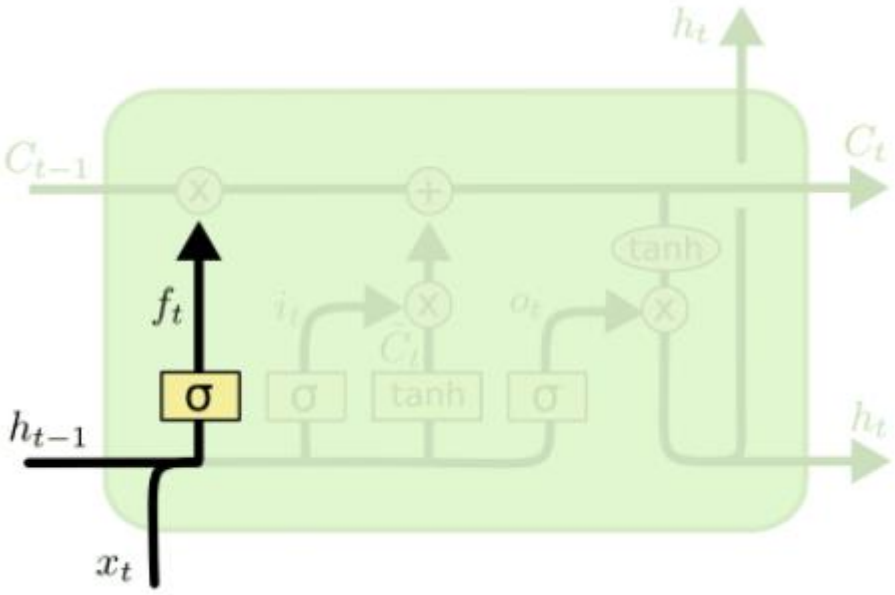

Figure 1.6: forget gate in an LSTM cell

The concatenation of the input and the previous output go through a sigmoid layer into bitwise multiplication with the previous cell state $C(t-1)$. The result of that multiplication can be considered as the previous cell state with the irrelevant information filtered out.

- *Input gate* decides which input values should be used to modify the memory.

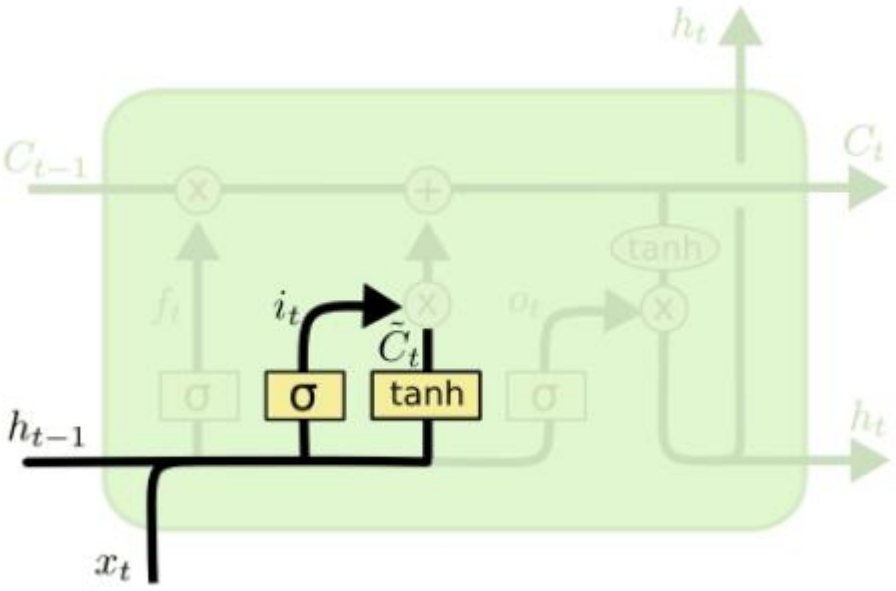

Figure 1.7: input gate in an LSTM cell

The concatenation of the input and the previous output go through a sigmoid layer and parallelly through a tanh layer. The tanh layer is used to create a vector of new candidate values and the sigmoid layer is used as a filter which, via bitwise multiplication with the new candidate vector, filters out the irrelevant new data and leaves only valuable information.

- *Output gate* uses both input and memory values to determine the output. Works similarly to the input gate.

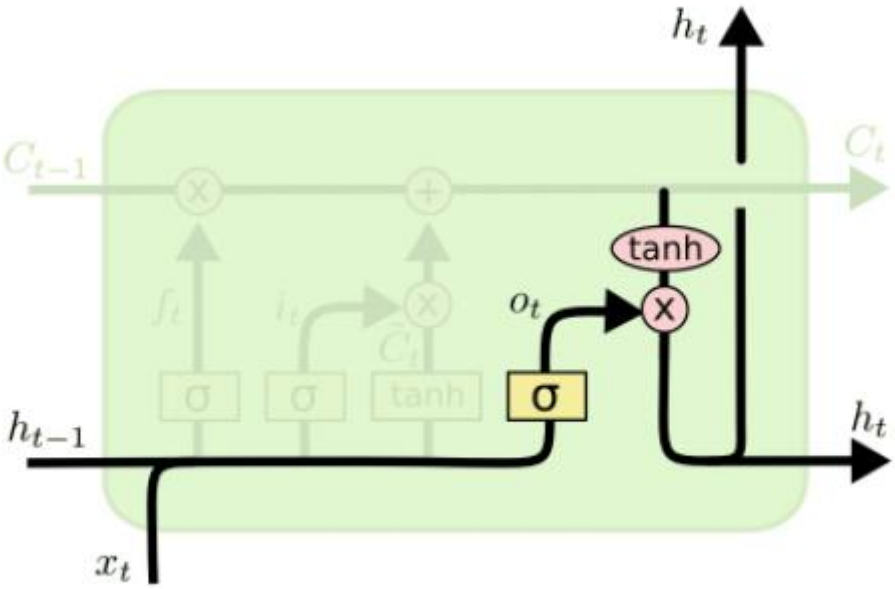

Figure 1.8: output gate in an LSTM cell

The concatenation of the input and the previous output go through a sigmoid layer which serves as a filter that ensures only information that is relevant to the output goes through from the cell state. This architecture allows LSTM to overcome the vanishing gradient problem. By remembering the state of any cell individually and separately from the output and changing its content only with carefully regulated gates, the system removes the long dependencies that previously inhibited RNNs from processing long sequences.

1.6 AMBA APB

1.6.1 AMBA Protocol family

Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. It facilitates the development of multi-processor designs with large numbers of controllers and components with a bus architecture. Today, AMBA is widely used on a range of Application-specific integrated circuits (ASIC) and SoC parts including application processors used in modern portable mobile devices like smartphones.

1.6.2 About AMBA APB

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family. It defines a low-cost interface that is optimized for minimal power consumption and reduced interface complexity. The APB protocol is used to connect to low-bandwidth peripherals. The APB protocol is a synchronous communication protocol in which every transfer lasts at least two clock cycles. The protocol defines all communication as communication between a bridge (that connects to the rest of the system) and a slave. i.e., memory devices, IOs, and other low-bandwidth communication protocols such as UART.

1.6.3 AMBA APB Signals

The AMBA APB protocol defines the following signals:

#	Signal name	Source	Description
1	PCLK	Clock source	Clock. The rising edge of PCLK times all transfers of the protocol
2	PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal
3	PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
4	PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a PSELx signal for each slave.

5	PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
6	PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
7	PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when PWRITE is HIGH. This bus can be up to 32 bits wide.
8	PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
9	PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when PWRITE is LOW. This bus can be up to 32-bits wide.
10	PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the PSLVERR pin. This is true for both existing and new APB peripheral designs. Peripherals are not required to support this pin and we will support this in our project meaning it will be tied low.

Table 1.1: AMBA APB Signals

- Notice that the AMBA APB protocol uses two different data buses, one for read data and one for write data. Each bus can be up to 32 bits wide. Data transfers can happen on one bus or the other at any given time (and not on both simultaneously).
- This table refers only to signals defined by the v1.0 protocol specification (APB3). There are newer versions and different signals which are not relevant for the purposes of this paper.

1.6.4 AMBA APB Transfers

WRITE Transfer

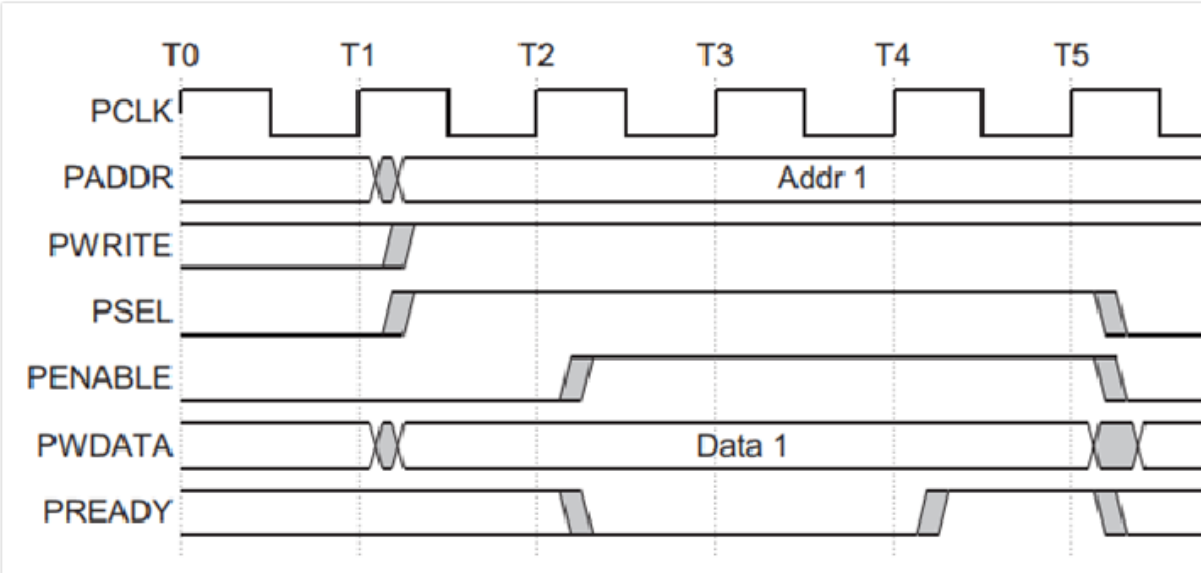

Figure 1.9: AMBA APB Write transfer timing diagram

- At T_1, a write transfer with address PADDR, PWDATA, PWRITE and PSEL starts.
- They will register at the next rising edge of PCLK, T_2γ
- This is the Setup Phase of the transfer.
- After T_2, PENABLE is asserted at the rising edge of PCLK.
- When asserted, PENABLE indicates starting of ACCESS Phase
- During the ACCESS Phase, when PENABLE is high, the slave may extend the transfer by driving PREADY low (the wait state).
- The PADDR, PWRITE, PSEL, PENABLE, PWDATA signals should remain unchanged while PREADY is low.
- When asserted, PREADY indicates that slave can complete the transfer at the next rising edge of PCLK.
- PADDR, PDATA and control signals all should remain valid till the transfer completes at T_5.
- PENABLE signal will be de-asserted at the end of transfer.
- PREADY can take any value when PENABLE is low.
- It is recommended that the address and write signals are not changed immediately after a transfer but remain stable until another access occurs.
- PSEL is also de-asserted if the next transfer is not to the same slave.

READ Transfer

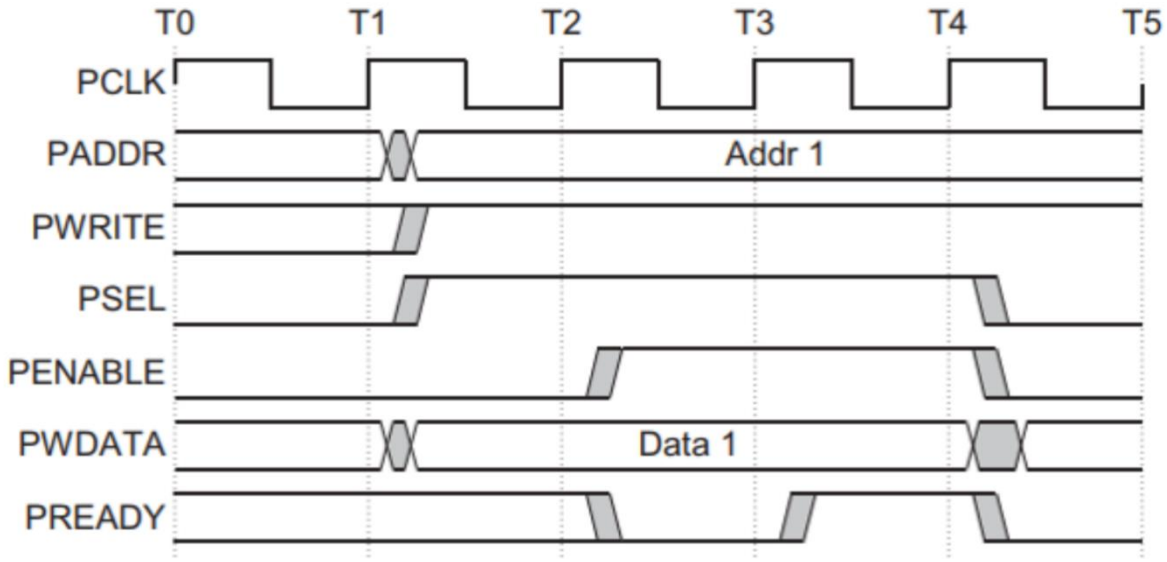

Figure 1.10: AMBA APB Read transfer timing diagram

- At T_1, a READ transfer with address PADDR, PWRITE and PSEL starts.
- They will be registered at the rising edge of PCLK.
- This is the SETUP Phase of the transfer.
- After T_2 PENABLE is asserted at the rising edge of PCLK.
- When asserted, PENABLE indicates the starting of ACCESS phase.
- During the ACCESS Phase, when PENABLE is high, the slave may extend the transfer by driving PREADY low (the wait state).
- The PADDR, PWRITE, PSEL, PENABLE signals should remain unchanged while PREADY is low.
- When asserted, PREADY indicates that slave can complete the transfer at next rising edge of PCLK by providing the data on PRDATA.
- Slave must provide the data before the end of read transfer. i.e., before T_4.

1.6.5 AMBA APB Operating States

The APB Protocol operates in three operating states as shown below.

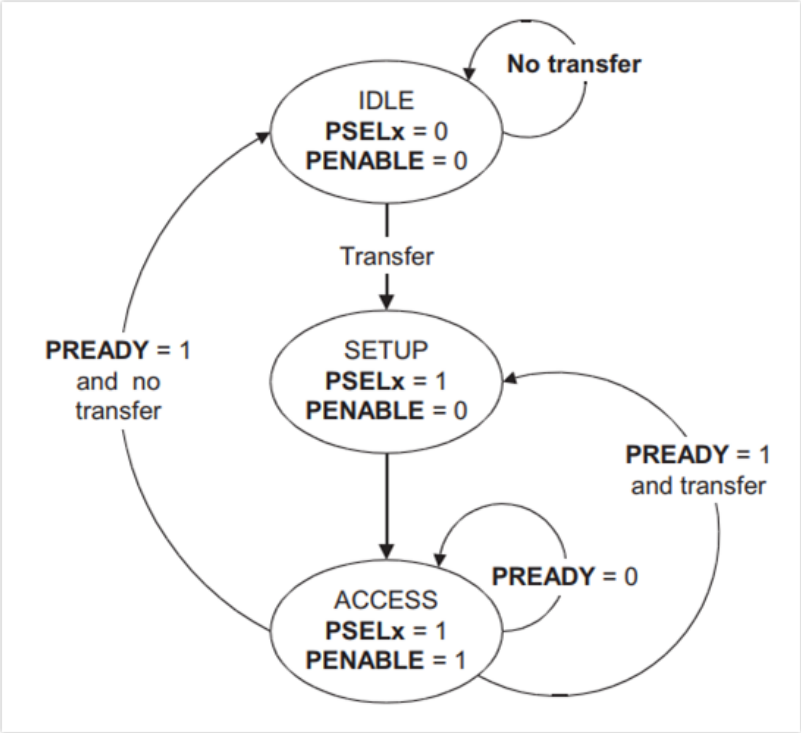

Figure 1.11: AMBA APB state diagram

IDLE: This is the default state of the APB protocol.

SETUP: When a transfer is required, PSELx is asserted and the bus moves into SETUP state. The Bus only remains in SETUP for one clock cycle and always moves to ACCESS state on the next rising edge of the clock. Thus, the slave must be able to sample the Address and control information in the SETUP cycle itself.

ACCESS: PENABLE is asserted to enter the ACCESS state. The PADDR, PWRITE, PSELx and PWDATA signals must remain stable during the ACCESS state.

1.7 Review of "Design and implementation of LSTM accelerator based on FPGA"

1.7.1 Intro

FPGA implementation of LSTM has gained in popularity during the past several years due to performance, flexibility, and efficiency advantages over more standard hardware implementations such as CPU, GPU and SoC.

In an article published on the 2020 IEEE 20th international conference on communication technology, Weifeng Zhang, Fen Ge, Chenchen Cui, Ying yang, Fang Zhou and Ning Wu describe the architecture and implementation of an LSTM machine on a XILINX Virtex-7 VC707 FPGA. Then, the authors compare their design's performance with other LSTM implementations on different Hardware Architectures.

This section shall cover the hardware architecture of the system and the results as achieved in the article, as the workings of the LSTM machine have already been addressed in this paper.

1.7.2 LSTM Architecture

The system is mainly composed of three parts: the storage unit, control unit, and LSTM acceleration unit. The storage unit uses a multi-level storage strategy, consisting of an offchip DRAM, input output and weight buffers. The control unit manages the read/write signals of the storage unit, along with the operation process. The acceleration unit consists of three modules: a matrix-vector multiplication module, an element-wise module, and an activation function module.

1.7.3 LSTM acceleration unit

Matrix-vector multiplication module: The input vector and the weight data are multiplied and accumulated to get the output vector. The input vector X is a concatenation of the current input and the previous output. The implementation of this module is done in a pipeline manner, using several parallel processing units called PEs.

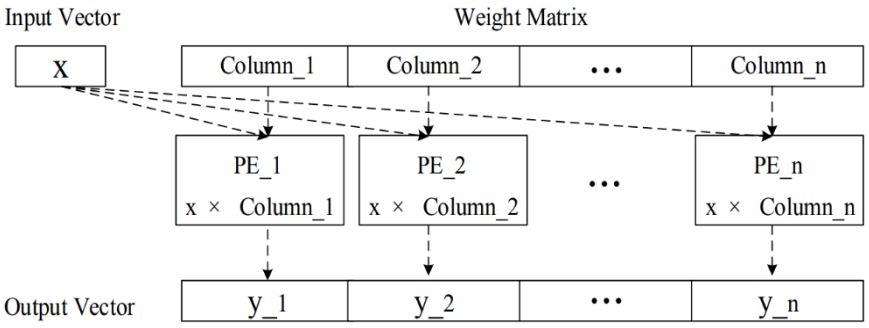

Figure 1.12: The parallel operation of PE

The elementwise module: processes the output vector of the matrix-vector multiplication module, to complete vector-element multiplication or addition. The module is split into three different states, to maximize resource usage - each state uses at most one bitwise operator of the same kind.

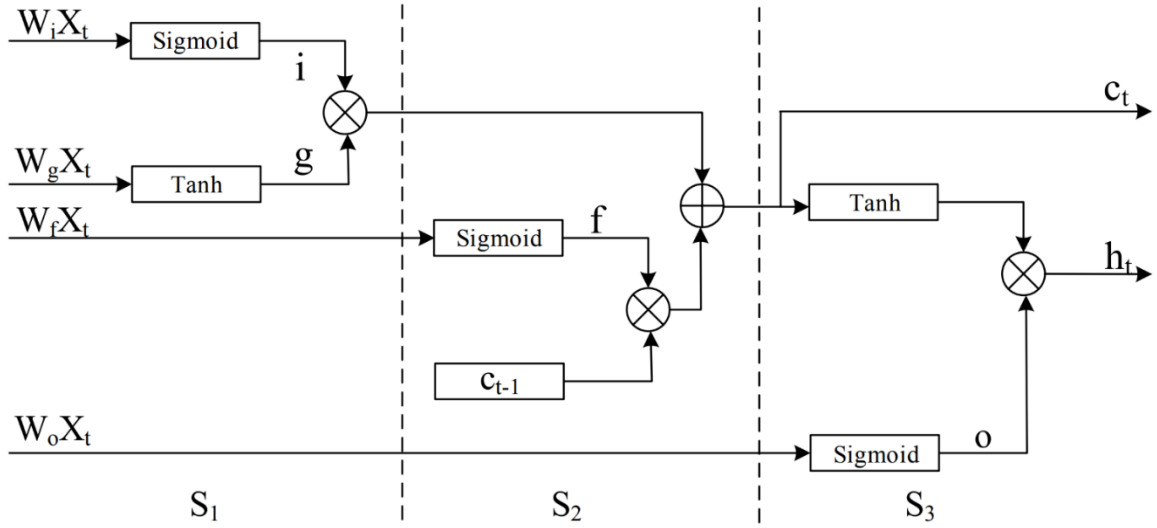

Figure 1.13: The structure of the Element-wise module

The activation function module: there are two types of activation functions, sigmoid and tanh. Each one is implemented using multipliers and adders. The values with which the functions are evaluated are stored in RAM.

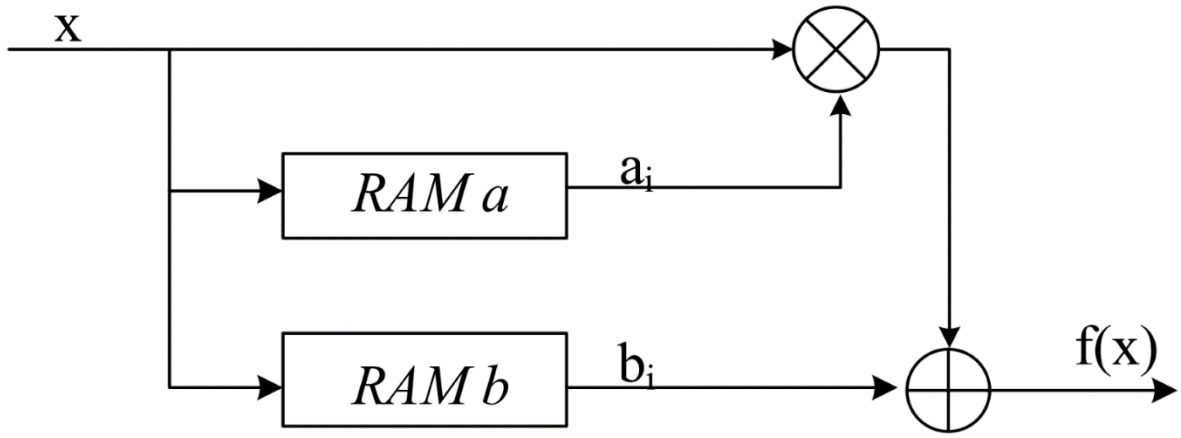

Figure 1.14: The design of activation function

1.7.4 Experimental results

The last section of the article compares the results of testing done with the proposed model, to previously proposed LSTM implementations. The accelerator is implemented on Virtex-7 VC707. The

FGPA contains a maximum bandwidth of 12.8GBps. The capacity of 1GB of DDR3 SDRAM off chip memory, and an operating frequency of 200MHz. The number of PEs is set to 32, and the parallelism of the element wise module to 4.

The article compared the running time, and power consumption of the LSTM module on the FPGA to the module on CPU and GPU, as well as analyzing the resource utilization of the FPGA (see table 1.2). Compared with CPU and GPU, the accelerator achieves 43 times and 3 times faster run time respectively. compared with GPU, FPGA power consumption is only 11% of GPU power consumption (see table 1.3).

Resource	LUT	DSP	BRAM	FF
Used	23473	142	165	21389
Total	303600	2800	1030	607200
Utilization	7.73%	5.07%	16.07%	3.52%

Table 1.2: The resource utilization of the accelerator

Platform	Time(ms)	Frequency(Hz)	Power(W)
CPU	4.788	3.20G	8.20
GPU	0.308	1785M	13.5
FPGA	0.111	200M	1.52

Table 1.3: Performance on different platforms

The throughput of the accelerator can reach 10.9 GOP/s, and the energy efficiency ratio can reach 7.1 GOPS/W. This performance is better than previous works (see table 1.4), mainly due to the

parallel structure and multi-level strategy which significantly improve the performance of the proposed accelerator.

Works	FPGA	Frequency	Performance
[9]	XC7VX485	150M	7.26GOP/s
[12]	XCZU6EG	238M	7.64GOP/s
[15]	XC7Z045	142M	454MOP/s
[16]	XC7Z020	140M	4534MOP/s
Our	XC7VX485	200M	10.90GOP/s

Table 1.4: Performance comparison with previous works

1.8 Systolic Arrays

1.8.1 Definition

The design of the system relies heavily on efficient matrix multiplication implementation. To achieve said efficiency, the authors have suggested an architecture based on systolic arrays. Systolic arrays are hardware structures built for fast and efficient operation of regular algorithms that perform the same task with different data at different time instants. Systolic arrays replace a pipeline structure with an array of processing elements that can be programmed to perform a common operation.

1.8.2 Benefits

Regularity, reconfigurability and scalability are some of the features of systolic design. Systolic architectures offer the competence to uphold the high-throughput capacity requirement. A major benefit of systolic arrays is that all operand data and partial results are stored within (passing through) the processor array. There is no need to access external buses, main memory or internal caches during each operation as is the case with Von Neumann or Harvard sequential machines. The sequential limits on parallel performance dictated by Amdahl's Law also do not apply in the same way, because data dependencies are implicitly handled by the programmable node interconnect and there are no sequential steps in managing the highly parallel data flow.

1.8.3 Applications

Multi-dimensional image processing algorithms, pattern recognition, video streaming, nonlinear optimization problems and decision-based algorithms are a few of many algorithms that are computationally demanding and can be benefited by implementing systolic arrays.

2. RNN LSTM Accelerator Implementation

2.1 Block Diagram

The top-level block diagram of the accelerator is depicted in figure 2.1:

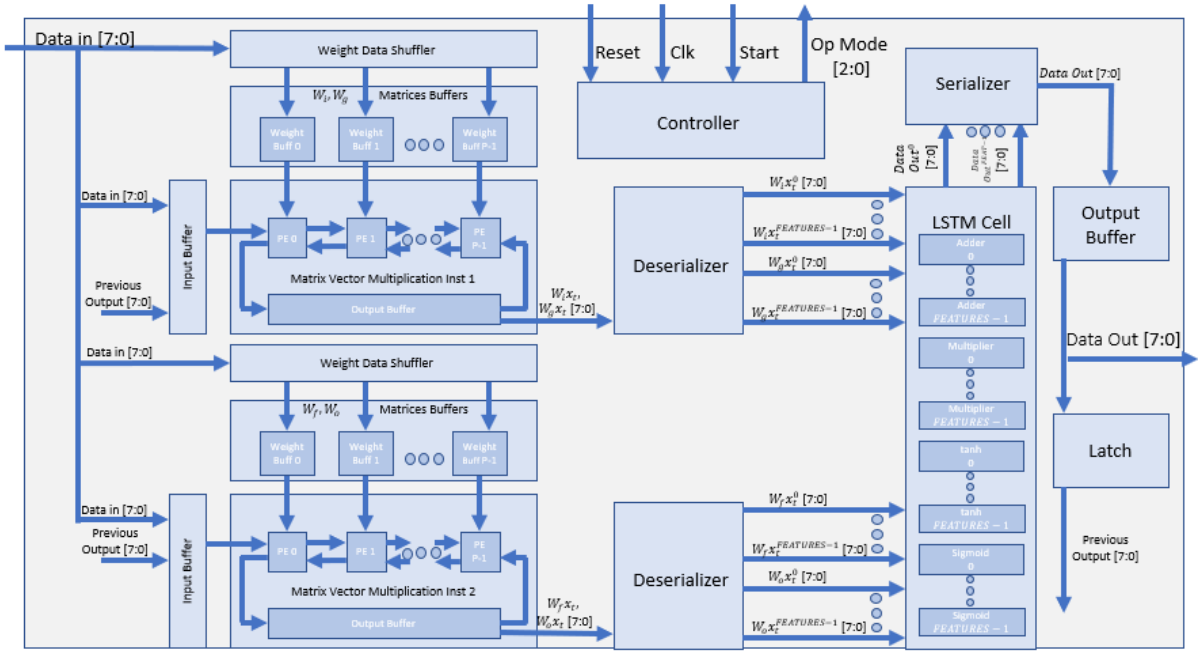

Figure 2.1: top-level block diagram

The left side of figure 2.1 shows the components of the main part of the system – the matrix vector multiplication module. It is comprised of a systolic array (Matrix Vector Multiplication Inst 1,2), see 1.8 for general definition and 2.6.1 for a detailed description, 2 data buffers, one for the input data (input buffer) and one for the constant weight data (matrices buffers), and a weight data shuffler which is responsible of ensuring data is fed into the systolic array in correct order. The aforementioned setup is instantiated twice, each instantiation in charge of two results to total 4 different results at a time, each with different weights (matrices). In the middle of the figure, the reader can see a deserializer which is responsible for feeding data into the next part of the system in a parallel manner. To the right of the deserializers, there is the LSTM Cell. This block manipulates the input data in a unique manner using activation functions and other arithmetic element-wise operations. Please refer to 1.5 for general definitions and 2.6.2 for an explanation regarding the specific implementation used in this project.

2.2 Block Description

Upon initialization, weight data matrices:

$$(1)\quad W_i^{(xy)}, W_g^{(xy)}, W_f^{(xy)}, W_o^{(xy)} \mid x = 1,2 \dots FEATURES\ ;\ \ y = 1,2 \dots 2 \cdot FEATURES$$

are ordered to fit the systolic array algorithm and are saved in dedicated DPRs in the *Matrix Vector Multiplication Module* ($W_i, W_g\ Matrix\ Buffers$ in figure 2.1). During this step, System mode (from here onwards $op\_mode[2:0]$) toggles between $INIT\_W1$ and $INIT\_W2$ depending on whether W_i, W_g or W_f, W_o are being loaded. Once this step is done, the accelerator performs the following steps in each operating cycle t:

a. Current input data is read from outside the system during R_IN state of $op\_mode[2:0]$:

$$(2)\quad x_t^{(i)} \mid i = 1,2 \dots FEATURES$$

Is concatenated to previous output data:

$$(3)\quad h_{t-1}^{(i)} \mid i = 1,2 \dots FEATURES$$

b. $CALC$ mode begins. The vector created in a. is fed into the Matrix Vector Multiplication Module where the results:

$$(4)\quad W_X X_t^{(k)} = \sum_{j=1}^{2 \cdot FEATURES} W_X^{(kj)} \cdot [x_t, h_{t-1}]^{(j)} \quad | \quad X = i, g, f, o \; ; \; k = 1,2 \dots FEATURES$$

are calculated and saved in a dedicated DPR.

c. The four results of equation (4) are read from the dedicated DPR and fed element-by-element into a deserializer. The output of the deserializer is four $[8 * FEATURES - 1:0]$ logics which hold the relevant results in 8-bit floating point format which is used throughout the system.

d. LSTM Cell utilizes the results of step c. as well as the previous cell state:

$$(5)\quad C_{t-1}^{i} \mid i = 1,2 .. FEATURES$$

To calculate:

$$(6)\quad f_t^i = \sigma\left(W_f x_t^i\right) \quad | \quad i = 1,2 \dots FEATURES$$

$$(7)\quad i_t^j = \sigma\left(W_i x_t^j\right) \quad | \quad j = 1,2 \dots FEATURES$$

$$(8)\quad g_t^i = tanh\left(W_g x_t^i\right) \quad | \quad i = 1,2 \dots FEATURES$$

$$(9)\quad o_t^i = \sigma\left(W_o x_t^i\right) \quad | \quad i = 1,2 \dots FEATURES$$

$$(10)\quad C_t^j = f_t^j \cdot C_{t-1}^j + i_t^j \cdot g_t^j \quad | \quad j = 1,2 \dots FEATURES$$

$$(11)\quad h_t^j = tanh\left(C_t^j\right) \cdot o_t^j \quad | \quad j = 1,2 \dots FEATURES$$

e. h_t calculated in (11) is serialized and written to an Output buffer. $CALC$ mode is done.

f. During W_OUT mode, the data saved in the output buffer is being written both outside the system and back to the $matrix\ vector\ multiplication$ module to be concatenated to the next input.

Flow Chart

A flow chart of the above operation is shown in figure 2.2:

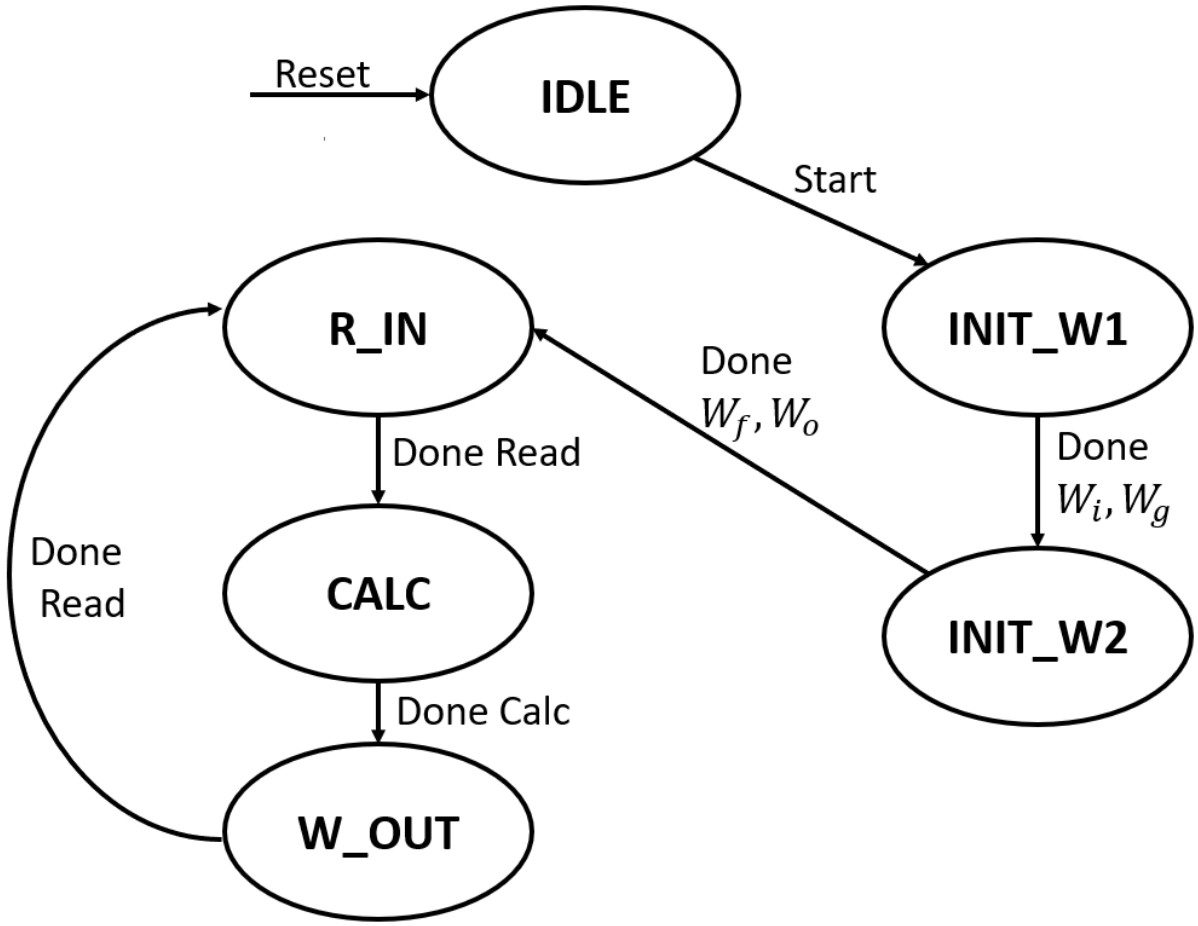

Figure 2.2: top-level flow chart

The system begins initialization upon receiving and external start signal. The initialization phase consists of 2 states, “INIT_W1”, “INIT_W2”. In each state, data is being read from an external memory into the “weight data shuffler” and then stored in the “matrices buffers” (see figure 2.1). Once “INIT_W2” is done, the system enters a the “R_IN” stage. In this stage input data is being read from an external memory and stored in “input buffer” (see figure 2.1). After “R_IN”, “CALC” begins, during this state the system performs all the calculations needed (including “matrix-vector multiplication” and propagation through the “LSTM Cell”). Once “CALC” is done, the output data is ready, and the system writes it out to the user during “W_OUT” phase. “R_IN”, “CALC” and “W_OUT” are executed in a loop until an external reset is applied from the user.

2.3 Pins Description

The accelerator’s pin description is depicted in table 2.1:

#	Signal name	Source (Accelerator POV)	Description
1	clock	Input	Clock. Times all data transfers, source of all internal clocks.

2	reset_n	Input	Reset, active LOW. This signal flushes all internal RAM and sets the accelerator to IDLE mode.
3	start	Input	Single clock cycle pulse, active high. This signal initiates standard operation.
4	data_in[7:0]	Input	Accelerator's input data in 8-bit floating point format. Transaction significance is determined by op_mode[2:0] signal: 1. $IDLE \rightarrow INIT\_W1$: transaction of W_i, W_g 2. $INIT_{W1} \rightarrow INIT\_W2$: transaction of W_f, W_o 3. $(INIT_{W2} \rightarrow W\_IN)$ *or* $(R_{OUT} \rightarrow W\_IN)$: transaction of x_t 4. $Otherwise:$ no transaction
5	data_out[7:0]	Output	Accelerator's output data in 8-bit floating point format. Transaction significance is determined by op_mode[2:0] signal: 1. $CALC \rightarrow R\_OUT$: transaction of h_t 2. $Otherwise:$ no transaction
6	op_mode[2:0]	Output	Accelerator's operational mode: 1. $IDLE$ $(3'b000)$: system is idle, enter by driving reset_n low and leave via start rising edge. 2. $INIT\_W1$ $(3'b001)$: initialization of W_i, W_g 3. $INIT\_W2$ $(3'b010)$: initialization of W_f, W_o 4. W_IN $(3'b011)$: transaction of input data x_t 5. $CALC$ $(3'b100)$: calculation of h_t as a function of x_t, h_{t-1} 6. R_OUT $(3'b101)$: transaction of output data h_t

Table 2.1: Pin description

2.4 Clocks and Resets

clk

System's main clock, times all data transfers, source of all internal clocks.

pe_clk

Internal clock. synchronizes data flow in the *Matrix Vector Multiplication Module.* To meet timing requirements of the floating-point multiplication and addition this clock satisfies:

$$(12)\quad f_{pe\ clk} = \frac{1}{16} f_{clk}$$

$cell\_clk$

Internal clock. synchronizes data flow in the *LSTM Cell.* To meet timing requirements of the floating-point multiplication and addition this clock satisfies:

$$(13)\quad f_{cell\ clk} = \frac{1}{16} f_{clk}$$

$cell\_clk$'s frequency is equal to pe_clk's frequency in this case is the same but each clock is in charge of timing a different module and can work independently from its counterpart.

$reset\_n$

Reset, active LOW. This signal flushes all internal RAM and sets the accelerator to IDLE mode.

$start$

Single clock cycle pulse, active high. This signal initiates standard operation. If system detects positive edge of this signal and $op\_mode[2:0] = IDLE$ then $op\_mode[2:0]$ will change to a valid $INIT\_W1$ by the next clk positive edge.

2.5 Interfaces description

Data transaction is done via $data\_in[7:0]$ input \ $data\_out[7:0]$ output, timed by clk signal and initiated utilizing $op\_mode[2:0]$ value as depicted in table 2.2:

Previous $op\_mode[2:0]$	Current $op\_mode[2:0]$	Direction	Description
$IDLE$	$INIT\_W1$	In	transaction of W_i, W_g
$INIT\_W1$	$INIT\_W2$	In	transaction of W_f, W_o
$INIT\_W2$ || R_OUT	W_IN	In	transaction of x_t
$CALC$	R_OUT	Out	transaction of h_t

Table 2.2: direction and description of transaction initiated as a function of change in $op\_mode[2:0]$

$INIT\_W1$ transaction

Upon change in $op\_mode[2:0]$ which initiates weight data transaction, the following sequence begins:

1. clk Rising edge #0 : current $op\_mode[2:0]$ valid value signifies a change in a manner which triggers weight data transaction as seen in table 2.2. $data\_in[7:0]$ value is not valid and not used by the accelerator.
2. clk Rising edge #1 through clock rising edge #$(2 \cdot FEATURES)^2$: valid data transaction. W_i data should be inserted in a serial manner from first to last followed immediately by W_g data from first to last.

3. clk Rising edge #$((2 \cdot FEATURES)^2 + 1)$: transaction is terminated automatically by the system. $data\_in[7:0]$ value is not valid and not used by the accelerator.

Example of such a transaction is shown in figure 2.3:

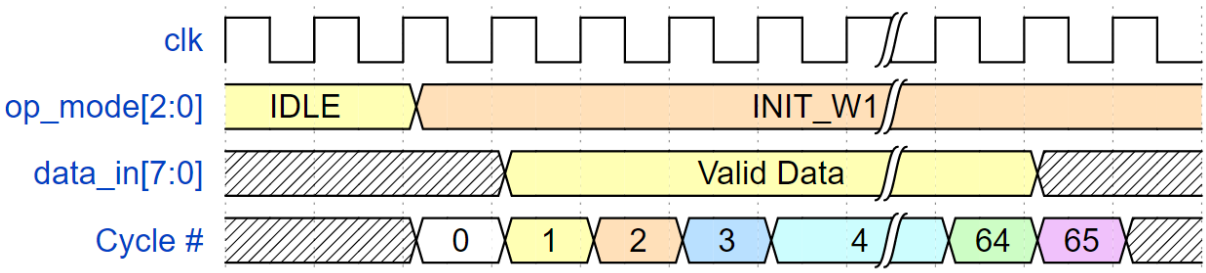

Figure 2.3: $INIT\_W1$ transaction for $FEATURES = 4$

$INIT\_W2$ transaction

Upon change in $op\_mode[2:0]$ which initiates weight data transaction, the following sequence begins:

4. clk Rising edge #0 : current $op\_mode[2:0]$ valid value signifies a change in a manner which triggers weight data transaction as seen in table 2.2. $data\_in[7:0]$ value is not valid and not used by the accelerator.
5. clk Rising edge #1 through clock rising edge #$FEATURES^2$: valid data transaction. W_f data should be inserted in a serial manner from first to last followed immediately by W_o data from first to last.
6. clk Rising edge #$(FEATURES^2 + 1)$: transaction is terminated automatically by the system. $data\_in[7:0]$ value is not valid and not used by the accelerator.

Example of such a transaction is shown in figure 2.4:

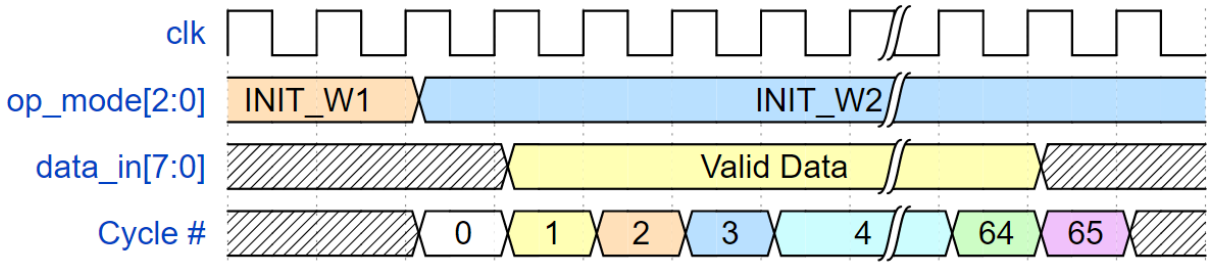

Figure 2.4: $INIT\_W1$ transaction for $FEATURES = 4$

x_t transaction

Upon change in $op\_mode[2:0]$ which initiates an input data transaction, the following sequence begins:

7. clk Rising edge #0 : current $op\_mode[2:0]$ valid value signifies a change in a manner which triggers an input data transaction as seen in table 2.2. $data\_in[7:0]$ value is not valid and not used by the accelerator.
8. clk Rising edge #1 through clock rising edge #$FEATURES$: valid data transaction. Input data should be inserted in a serial manner from first to last.

9. clk Rising edge #$(FEATURES + 1)$: transaction is terminated automatically by the system. $data\_in[7:0]$ value is not valid and not used by the accelerator.

Example of such a transaction is shown in figure 2.5:

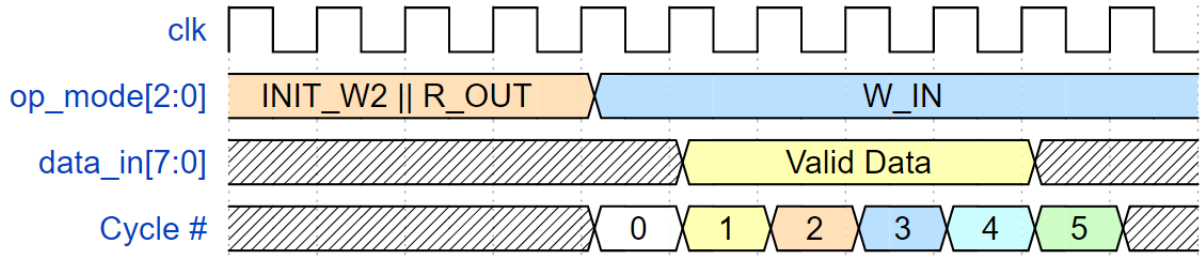

Figure 2.5: x_t transaction for $FEATURES = 4$

h_t transaction

Upon change in $op\_mode[2:0]$ which initiates an output data transaction, the following sequence begins:

10. clk Rising edge #0 : current $op\_mode[2:0]$ valid value signifies a change in a manner which triggers an output data transaction as seen in table 2.2. $data\_out[7:0]$ value is not valid and not used by the accelerator.
11. clk Rising edge #1 through clock rising edge #$FEATURES$: valid data transaction. Output data is inserted in a serial manner from first to last.
12. clk Rising edge #$(FEATURES + 1)$: transaction is terminated automatically by the system. $data\_in[7:0]$ value is not valid and should not be used.

Example of such a transaction is shown in figure 2.6:

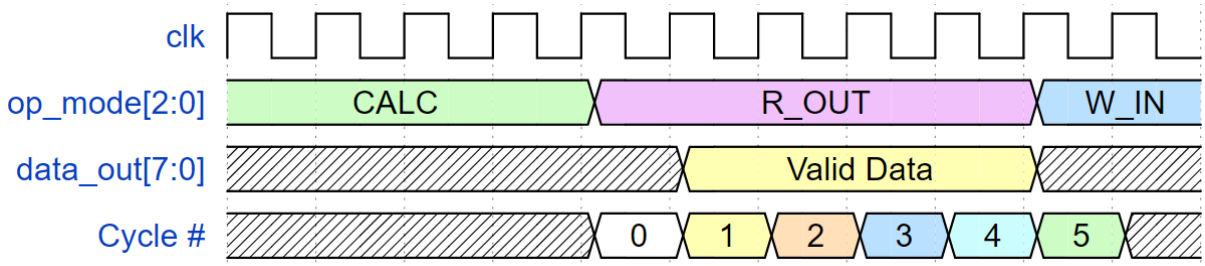

Figure 2.6: h_t transaction for $FEATURES = 4$

2.6 Sub-units' description

2.6.1 Matrix vector multiplication module

This section describes the main block of the system. It conducts matrix-vector multiplication in a systolic manner. The following passages elaborate on the definitions, algorithm, implementation and verification of said unit. The inputs of this unit are weight data (matrix) and input data (vector) which are fed into the system from an external memory through $data\_in[7:0]$ or $Previous\ Output\ [7:0]$ and the outputs are the four data vectors $W_g x, W_i x, W_o x, W_f x$ which are the result of the matrix-vector multiplication of the inputs. It mainly consists of a "weight data shuffler", "input buffer", "matrices buffers" and a systolic array "matrix vector multiplication inst i" which can be seen in figure 2.1.

Motivation and problem definition

The four basic definitions of the gates in an LSTM cell as they were defined in equations $(6)-(9)$ of this chapter. Deriving these expressions requires many matrix multiplication operations. Thus, the efficient implementation of this module is in our best interest.

Say that our input $\vec{\imath}$ is of size i and our output $\vec{h}$ is of size h such that the vector $\vec{b}$ is a concatenation of $\vec{\imath}$ and $\vec{h}$ the size of $h+i=n$. In addition, for each gate we can define a weight matrix which is a concatenation of the weights of the input and the output for said gate:

$$A = (W_{x*}, W_{h*})^T \quad of\ size\ \ nXa$$

We can now rewrite the problem as:

$$(14)\ \ i_t = sigmoid\left(A_i \cdot \vec{b} + b_i\right)$$

$$(15)\ \ f_t = sigmoid\left(A_f \cdot \vec{b} + b_f\right)$$

$$(16)\ \ o_t = sigmoid\left(A_o \cdot \vec{b} + b_o\right)$$

$$(17)\ \ g_t = tanh\left(A_g \cdot \vec{b} + b_g\right)$$

Let us now look at the brute-force approach to conduct such a calculation of $A_* \cdot \vec{b}$ from the perspective of estimated throughput: to calculate $(A * b)_i$ we will need to preform n operations and we must do so a times. All in all, we reach a time complexity of $O(na)$. And for $a \approx n$ we can write $O(n^2)$.

Systolic array algorithm

This section describes the module which its input is a vector $\vec{b}$ the size of n and a matrix A the size of aXn, and its output is a vector $\vec{c} = A\vec{b}$ the size of a. A detailed block diagram can be seen in figure 2.7:

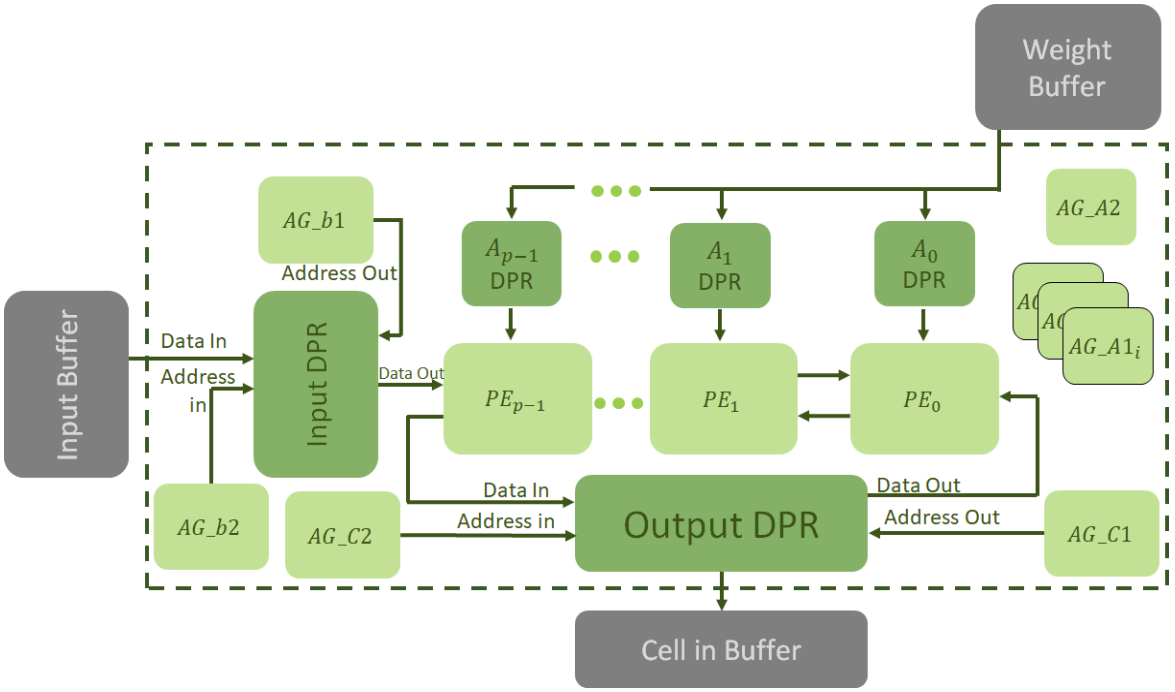

Figure 2.7: block diagram of matrix vector multiplication module

In this figure, the set of "A_i" DPRs comprise the "matrices buffers" in figure 2.1 and the set of address generators "AG_A2" and "AG_A1_i" comprise the weight data shuffler in figure 2.1. the set of "PE_i" comprise the systolic array or "Matrix Vector Multiplication inst i" in figure 2.1. this section will use the more elaborate naming convention of figure 2.7.

The inputs to this module are:

1. $n\ X\ n$ Weight matrix – static, drawn from the weight buffer and stored in A_i DPRs once at the beginning of system operation (as it does not change during operation). This matrix is comprised of 2 different weight matrices of size $\frac{n}{2}\ X\ n$ which correspond to W_i, W_g in case of "Matrix Vector Multiplication Inst 1" or to W_f, W_o in case of "Matrix Vector Multiplication Inst 2".
2. $n\ X\ 1$ Input vector – $\frac{n}{2}$ data elements are drawn from the input buffer and $\frac{n}{2}$ data elements are drawn from the output of the "LSTM Cell"'s last output.

The output of this module Is an $n\ X\ 1$ cell-in vector – drawn from the output DPR and stored in Cell in Buffer. The first $\frac{n}{2}$ data elements correspond to $W_i x_t$ in case of "Matrix Vector Multiplication Inst 1" or to $W_f x_t$ in case of "Matrix Vector Multiplication Inst 2" and the last $\frac{n}{2}$ data elements correspond to $W_g x_t$ in case of "Matrix Vector Multiplication Inst 1" or to $W_o x_t$ in case of "Matrix Vector Multiplication Inst 2".

Inside the module, $p \leq \frac{m}{2}$ processing elements (PEs) are being used to perform parallel computation, each PE is composed of a multiplier, adder and 2 latches as shown in figure 2.8:

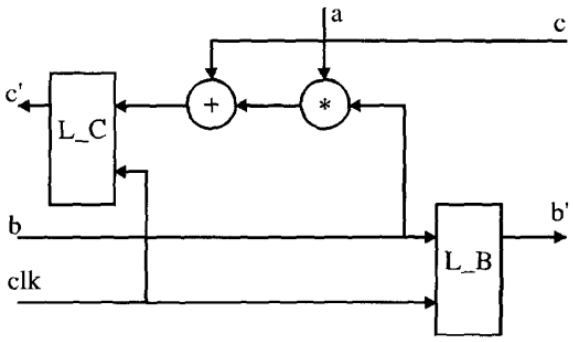

Figure 2.8: internal structure of a PE

Where the circles indicate an element-wise operation, and the rectangles are latches. In each iteration, the a and b inputs to the PE are multiplied, added to the c input and latched in L_C. The input b is latched as is in L_B for further computation.

In addition to the PEs, different Address generator modules oversee organizing the data flow from and to the DPRs. Finally, DPR sizes of the module are shown in table 2.3:

DPR	Rows	Columns
Input	1	$n+1$
Output	1	n
A_i	$\frac{n^2}{p}$	1

Table 2.3: DPR sizes of the systolic array module

To simplify the description of the module, some notations are in order:

$$(18) \quad m = \begin{cases} n, & n\ is\ odd \\ n+1, & n\ is\ even \end{cases}$$

$$(19) \quad \gamma = \left\lceil \frac{m}{p} \right\rceil, \qquad r = 2m - p$$

For a given vector $\vec{x} = [x_0, x_1 \dots, x_{m-1}]^T$ and $k = \left\lceil \frac{(m-1)}{2} \right\rceil$ the *perfect ordering* is defined as:

$$(20) \quad \vec{x}^* = [x_0, x_{k+1}, x_1, x_{k+2}, \dots]^T$$

A shifted vector by that is a result of a t element shift of vector $\vec{x}$ will be denoted as:

$$(21)\quad (\vec{x})_t = shift\_by\_t(\vec{x})$$

For a given matrix M the *perfect repartitioning* is defined as the repartitioning of the matrix into m quasidiagonals $\overrightarrow{d_i}$ such that:

$$(22)\quad \overrightarrow{d_i} = \left[a_{0,i}, a_{1,(i+1)mod(m)}, \dots, a_{k,(k+1)mod(m)}, \dots., a_{m-1,(m+i-1)mod(m)}\right]$$

And M_i is a block matrix of order mXm that contains p quasidiagonals such that M_0 contains the first p quasidiagonals $\overrightarrow{d_i}$, $\quad i = 0, \dots, p-1$ and so on up until $M_{\gamma-1}$ which contains quasidiagonals $\vec{d}_{(\gamma-1)p+i}$, $\quad i = 0, \dots p-1$. If $\frac{m}{p}$ is not an integer, then $\overrightarrow{d_m} = \vec{0}$. Note that as a result from this repartitioning we can write M as:

$$(23)\quad M = \sum_{i=0}^{\gamma-1} M_i$$

Applying the *perfect repartitioning* to the input matrix A allows as to write the output vector in the following manner:

$$(24)\quad \vec{c} = \sum_{i=0}^{\gamma-1} A_i\, \vec{b}$$

Elements of the resulting vector $\vec{c}$ enter the array by PE_0 in the distribution shown in figure 2.9:

$$PE_0 \longleftarrow \left| \quad \vec{c}^{\,*} \quad \right| \left(\vec{c}^{\,*}\right)_r \left| \cdots \right| \left(\vec{c}^{\,*}\right)_{(\gamma-1)r} \Big|$$
$$\left|1.\ \text{iter.}\right|\ 2.\ \text{iter.}\ \left| \cdots \right| \quad \gamma\ \text{iter.} \quad |$$

Figure 2.9: output vector $\vec{c}$ order of entrance to the systolic array

The output vector C is stored in an $m\ by\ k$ size dual-port RAM (DPR) named DPR_C in a chronological order (i.e. c_0 is stored in address 0, c_1 is stored in address 1, etc.). PE_0 and PE_{p-1} read from said DPR using addresses generated in dedicated modules named AG_0, AG_(P-1). Both modules are built the same way but AG_(P-1)'s output is delayed by p time units. The AG modules generate the addresses using the following equation:

$$(25)\quad adr = mod_2\big(mod_m(i + j \cdot r)\big) \cdot \left(\frac{m-1}{2} + 1\right) + div_2\big(mod_m(i + j \cdot r)\big)$$

Where $i = 0,1 \dots m-1$ is representative of the time step and $j = 0,1 \dots \gamma - 1$ is representative of the iteration. When it comes to the hardware implementation of formula (25), there are a few things we must keep in mind:

g. The div2 operation, which is power of 2, on unsigned numbers such as addresses, is simply a logical shift of the bits to the right. Since the divisor is a constant 2, the shift operation is a hard-wired shift. In short, there is no circuit involved, simply a rearrangement of the wires in the bus is required.

h. No circuit is necessary at all in the implementation of the mod2 operation. The modulus of a number is found by simply discarding the MS bits and keeping the LS bit which represents the range of the modulo arithmetic. This is effectively a masking operation in which the MS bits are masked off, leaving just the LS bit intact.

i. The result of mod2 operation is 0 or 1, the second multiplication can be implemented by a multiplexer.

A block diagram of the resulting AG module can be seen in figure 2.10:

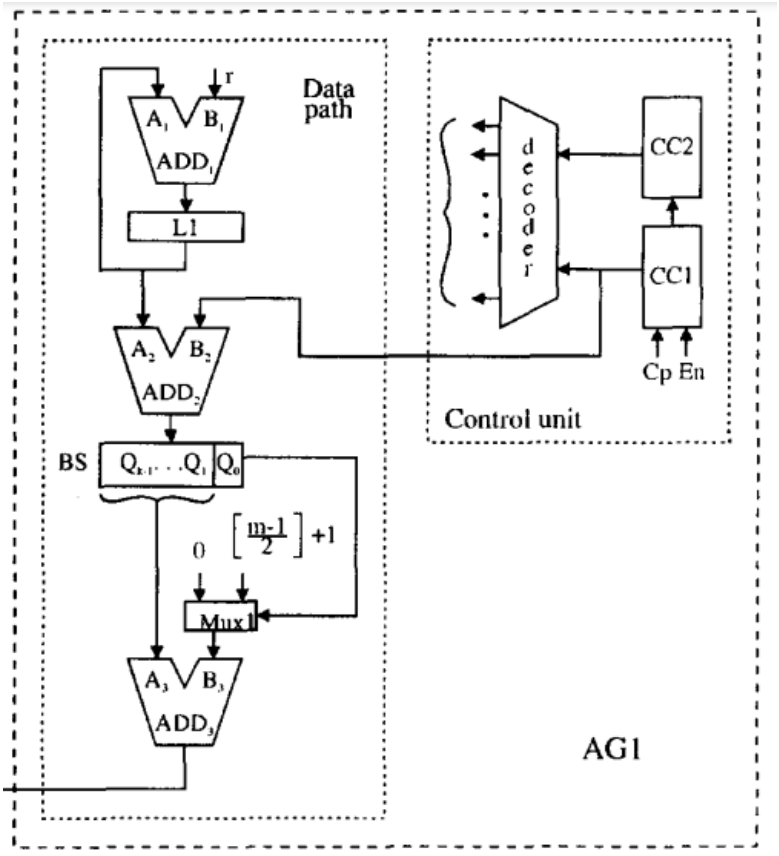

Figure 2.10: AG architecture

The AG constituents of: two modulo m adders, ADD1 and ADD2, one full adder, ADD3, a latch, L1, a barrel shifter, BS, and a multiplexor, MUX1. Blocks ADD1 and L1 perform modulo m add-with-accumulation operation. Initially, L1 is set to zero. At the beginning of each iteration the content of L1 is incremented for the shifting factor r. The ADD2 performs the modulo m addition. The BS performs two operations simultaneously: mod2, by keeping the right most bit position of the ADD2 output, and div2 by shifting the ADD2 output one position to the right and appending zero to the

left-most bit position. The result of mod2 operation is used as a select signal for MUX1. MUX1 then implements multiplication of term [(m - 1)/2] + 1 by 0 or 1. Finally, ADD3 generates the address for accessing data elements in DPR_C. A table showing the resulting address generation for $p = 2$, $m = 5$ can be seen in figure 2.11:

i	j	$(i + j \cdot r)$	$(i + j \cdot r) \bmod m$	$((i + j \cdot r) \bmod m) \bmod 2$	Adr
0	0	0	0	0	0
1	0	1	1	1	3
2	0	2	2	0	1
3	0	3	3	1	4
4	0	4	4	0	2
0	1	1	1	1	3
1	1	2	2	0	1
2	1	3	3	1	4
3	1	4	4	0	2
4	1	5	0	0	0
0	2	2	2	0	1
1	2	3	3	1	4
2	2	4	4	0	2
3	2	5	0	0	0
4	2	6	1	1	3

Figure 2.11: address generation for $p = 2,\ m = 5$

The algorithm proposed calculates each element of the sum in equation (24) in its turn. After applying the *perfect repartitioning* to A, we apply the *perfect shuffling* to each quasidiagonal $\vec{d}_i$ and shift each element of $\vec{d}_i$ by t where t is the index of A_t that $\vec{d}_i$ belongs to. The result than enters the systolic array in the manner shown in figure 2.12:

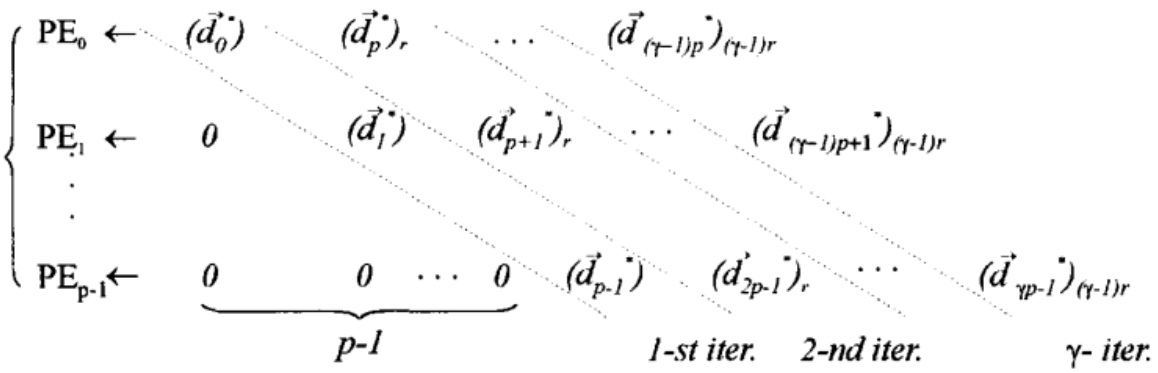

Figure 2.12: the ordering of matrix A entrance to the systolic array

Where the shuffling and shifting operations are notated as defined in equations (20) and (21). Elements of A are stored in p identical DPRs named DPR_A_i. The above shuffling and shifting is implemented using an AG_A block which is divided into 2 stages:

a. AGA_H which reads a diagonal element from the main memory and stores it in a temporary buffer, Temp_buff.
b. AGA_DPR which accepts elements from Temp_buff and writes it into the corresponding DPR_Ai, $i = 0, \ldots, p-1$.
c. AG_SA which performs a shuffle according to equation (25).

Block AGA_H generates addresses for accessing diagonal elements of matrix A stored in the main memory. Elements of the i-th diagonal, i = 0,..., m - 1, are read from the addresses in equation (26):

$$(26)\ adr\_diag = m \cdot j + mod_m(i+j), \quad j = 0,1 \ldots m-1$$

To implement the computation given in (26) we need: a multiplier, a modulo m adder, and a full adder. The structure of AGA_H is shown in Figure 9. The first term in (26) is again implemented as an add-with-accumulation operation using one full adder, SUM1, and one latch, L_A1. The term $(i + k)\ mod\ m$ is obtained at the output of modulo m adder, denoted as SUM2, and finally, the address Adr_ diag for accessing elements of matrix A located in main memory is generated at the output of full adder SUM3. A block diagram depicting the above AGA_H module can be seen in figure 2.13:

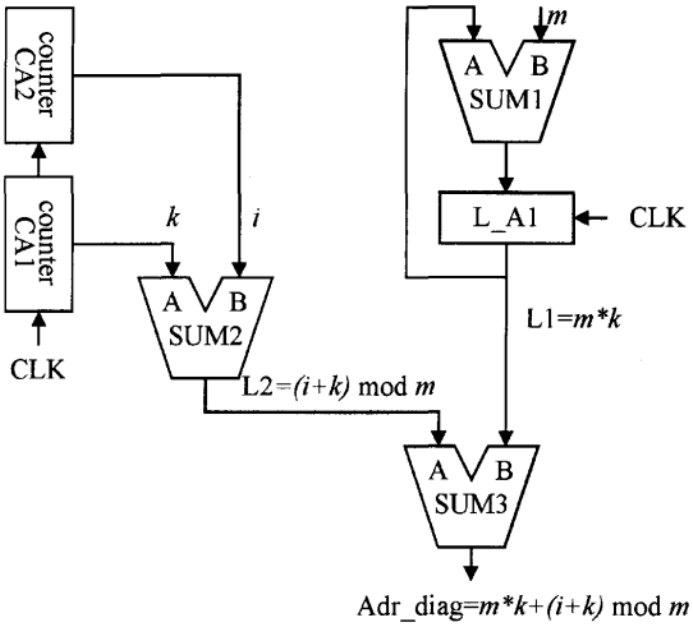

Figure 2.13: adr_diag generation module

Data is fed into the PE array from the DPR_Ai i = 0,...,p-1, according to the addresses generated by the address generator AG_SA. The AG_SA performs perfect shuffle and shifting permutations over diagonal elements of matrix A during address generation. Its structure is like the structure of AG0 shown in Figure 2.10.

Elements of the input vector $\vec{b}$ always enter the systolic array as $\vec{b}^*$. It is read from the main memory and stored in DPR_B after being repartitioned and reshuffled by module AG_B according to equation (25) with the exception that $r = 0$ and the systolic array reads from DPR_B using the AG_SA module which is simply a counter. The block diagram depicting the data flow of b elements to the array is shown in figure 2.14:

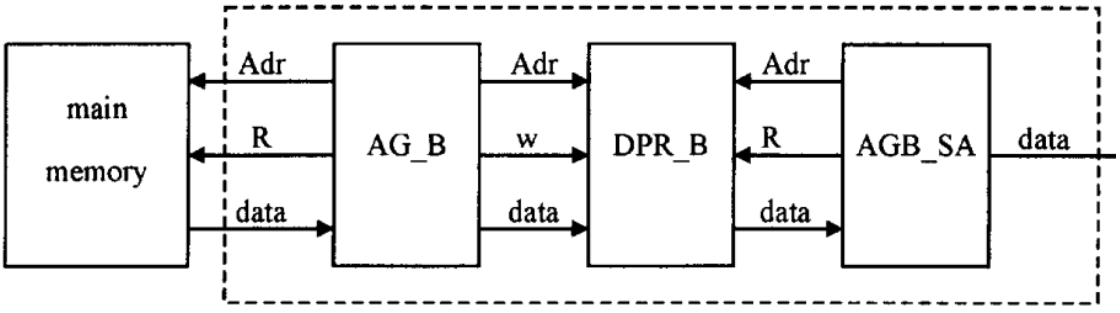

Figure 2.14: data flow of b to the systolic array.

The outcome of the described algorithm for $p = 2,\ m = 5$ can be seen in figure 2.15:

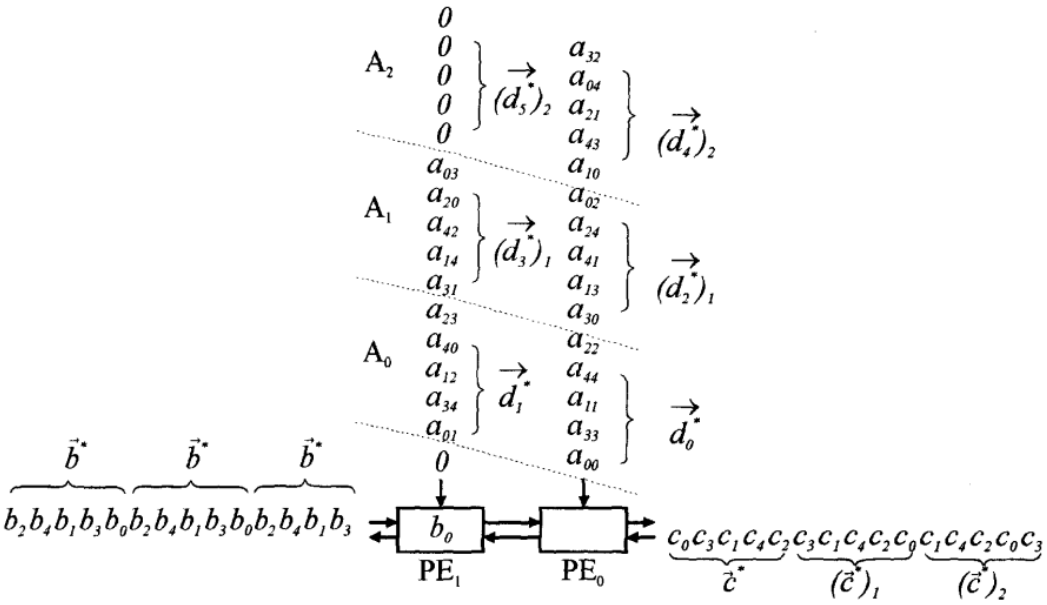

Figure 2.15: data flow through a systolic array where $p = 2,\ \ m = 5$

2.6.2 LSTM Cell

Definition

This module comprises the LSTM cell described in section 5 of chapter 1, located in the right side of the main block diagram in figure 2.1. It receives 4 input vectors, which are the outputs of the matrix-vector multiplication module described in section 1 of this chapter: $W_i X_t, W_g X_t, W_f X_t, W_o X_t$. these vectors are the previous cycles output vector h_t concatenated with the current cycles input vector x_t multiplied by the relevant weights at the matrix-vector multiplication module (see section 2.6.1). this module's outputs are the vectors c_t, h_t, namely the cell state, which allows the module to remember the cells value long-term, and the output vector, sent to the main memory. These are calculated using the relations presented in equations $(10), (11)$. The unit mainly consists of activation functions and element-wise arithmetic operations. The former will be explained in section 2.6.3 while the latter will be explained in 2.6.4. This section (2.6.2) will explain the interconnections of the two as well as the overall pipeline-structure.

Functionality Description

There are several steps in the implementation of the module. First, the module uses the forget gate f_t to calculate which values should be discarded, using the sigmoid activation function to determine the degree to which the information is to be forgotten. Second, the module uses the input gate i_t to decide which values should be updated, while also generating candidate values to update with, g_t. The first vector is passed through sigmoid while the second is passed through tanh, to tackle the vanishing gradient problem, as they are then being multiplied. Third, the cell state (long-term memory) is updated. The previous state c_{t-1} is multiplied by f_t, then added the multiplication of

$i_t \otimes g_t$ calculated in step 2, according to formula (10). The fourth step is to calculate the output vector h_t. The output gate is used to decide which values should be outputted, and these values are then multiplied with the squashed recently calculated cell state c_t, as described in formula (11). A block diagram of the LSTM cell can be seen in figure 2.16:

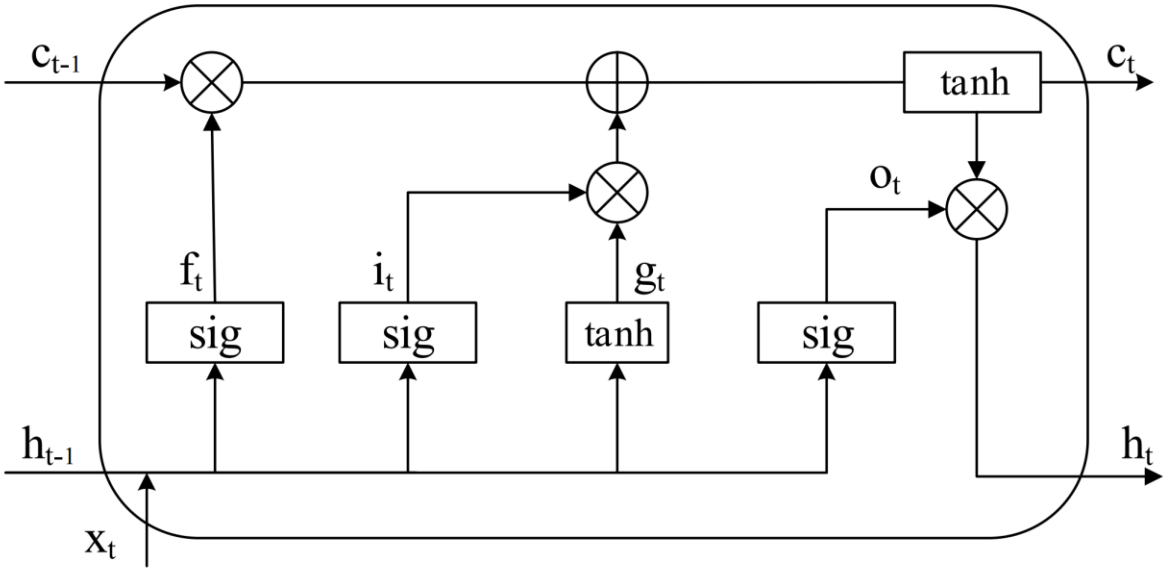

Figure 2.16: the LSTM cell architecture

This functionality was implemented in a pipeline manner which is comprised of 3 stages exactly as described in section 1.7:

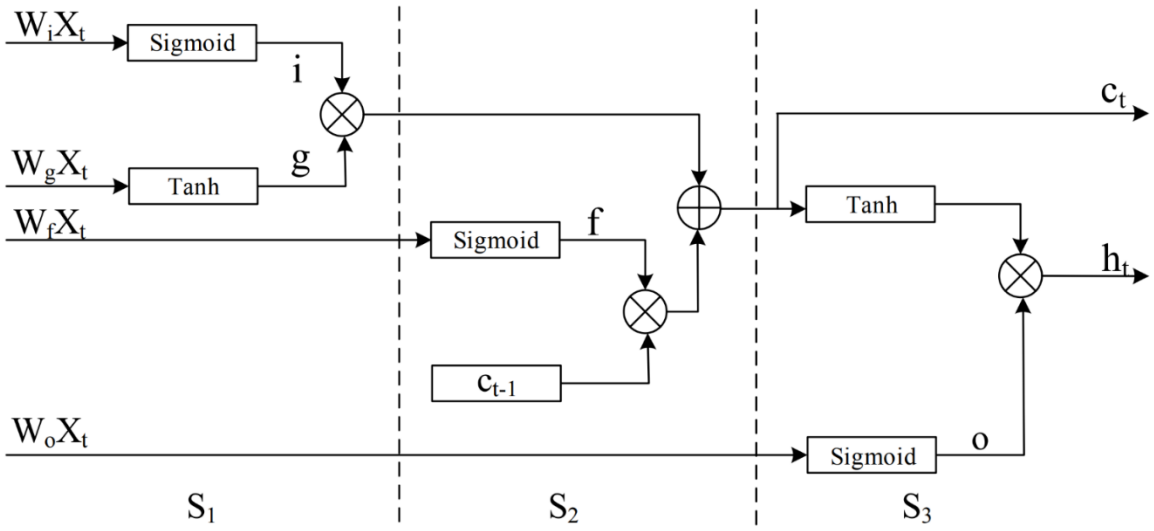

Figure 2.17 : LSTM Cell pipeline

2.6.3 Activation function module

This unit is a part of the LSTM Cell (2.6.2) and performs sigmoid or tanh operations on single data elements (8-bit, floating point numbers).

Theoretical solution

When designing a recurrent neural network, two major problems that require addressing are the **exploding gradient problem** and the **vanishing gradient problem**. In short, both problems arise from the flow of data backward and forward through time, known as **propagation** and **backpropagation**. As the propagated values advance from one layer to another, they are being multiplied constantly, and so if they are bigger than 1, they tend towards zero (known as “exploding”), and if smaller they tend towards 0 (known as “vanishing”).

A common method to solve these problems is the use of an **activation function**, which effectively “squashes” the values into a selected range. Two common activation functions, which will be used in this LSTM module, are the **tanh** (Hyperbolic Tangent) function (39), which produces values $x \in (-1,1)$, and the **sigmoid** function (38), which produces values $x \in (0,1)$. However, the two functions are non-linear, and are therefore difficult to efficiently implement in a hardware system.

$$(38) \quad sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$(39) \quad \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

The approach is therefore to **approximate** the activation function values and store them in a lookup table (LUT). The function will be split into sections. Each table entry will store the actual $\tanh(x)$ value of the corresponding section's left most point using a 32-bit signed number.

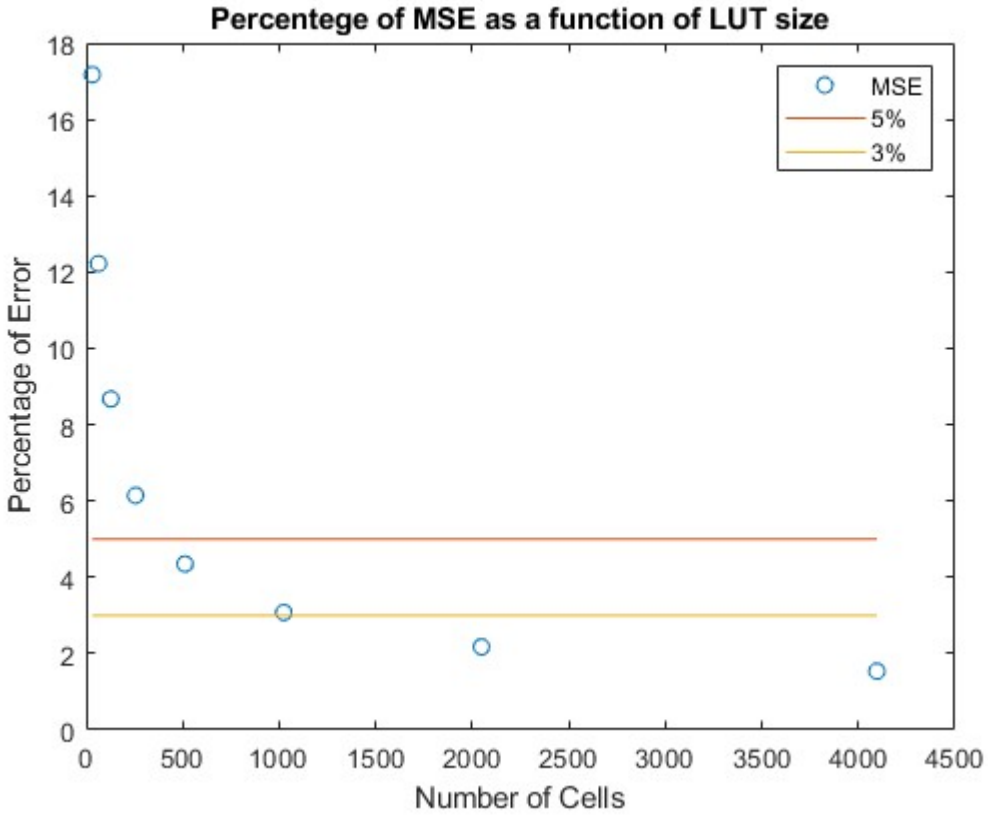

Figure 2.18: MSE as a function of LUT size

It is important to note that even though this project was eventually implemented using an 8-bit architecture, preparation and thought were also put towards possible future expansions to architectures with more bits, and this work is presented in the following chapter. The actual, 8-bit appropriate implementation used in this project shall then be described.

The general requirement is therefore to strike a balance between a reasonable approximation error and LUT size. As seen in figure 1 above, in order to reach a desirable 3-5% error rate for (for example) 32-bit systems, a 1024 cells large LUT is chosen. It amounts to $1024 \cdot 32b = 4KB$ LUT size and delivers at the worst a 3.07643% error rate, and both parameters are acceptable.

The function describing the modules activation function implementation is (40):

$$(40) \quad y = \begin{cases} sign(x), |x| \geq 4 \\ LUT[i], x \in [x_i, x_{i+1}] \end{cases}$$

The proposed design will be implemented on an FPGA or similar technology. These have 2 types of on-chip memory: distributed RAM and block RAM; Distributed RAM uses existing logic units as storage, and usually holds small amounts of data, while Block RAM uses a separate specified area of the chip and normally holds up to $4Kb$. Since the LUT size is $4KB$, distributed RAM will not suffice, and block RAM must be used.

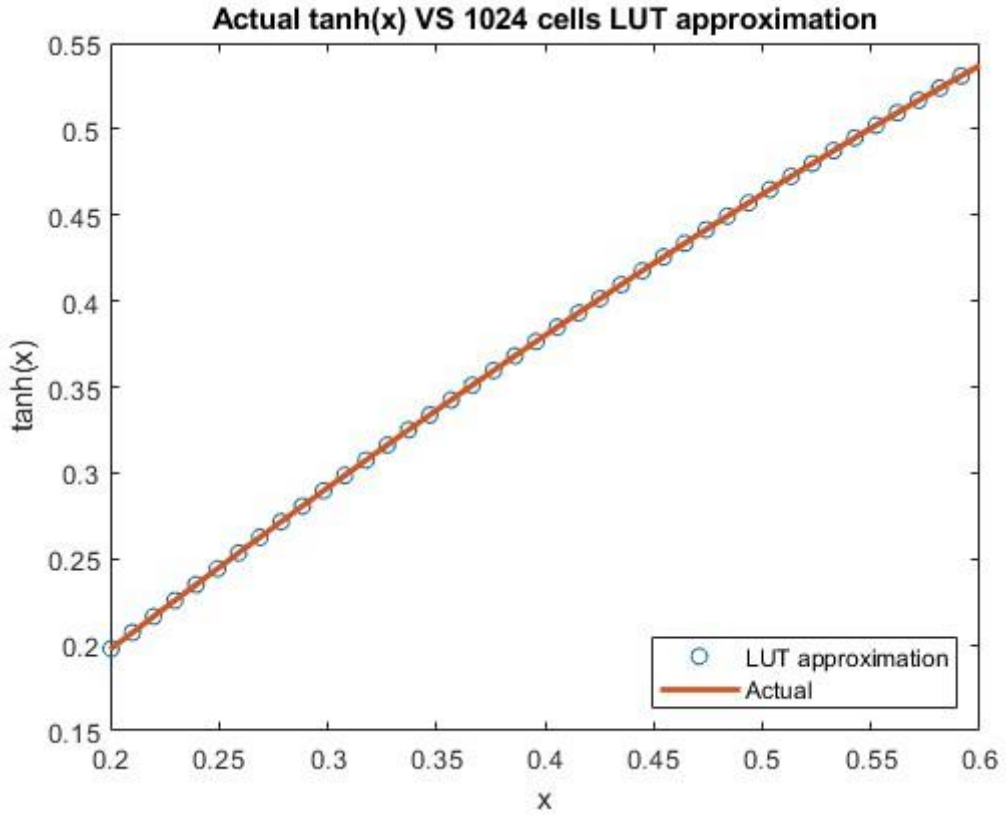

Figure 2.19: actual tanh(x) vs the module's approximation. between the dots the left one is chosen.

While the LUT size chosen was 1024 cells, several properties of each function can be used to improve the table's resolution. With tanh, property (41) enables storing only the values between $x \in [0,4]$, effectively doubling the table's resolution. Property (42) is used to better approximate the values for $x \in \left[\frac{0,1}{4}\right]$, also making more cells available which otherwise would have been used to approximate this range of inputs. With sigmoid, property (43) allows storing only the positive values of range $x \in [0,4]$, again effectively doubling the number of stored values.

$$(41) \quad \tanh(-x) = -\tanh(x)$$

$$(42) \quad \tanh(x) \approx x, x \in \left[0, \frac{1}{4}\right]$$

$$(43) \quad \sigma(-x) = 1 - \sigma(x)$$

When approaching the algorithm to find the correct LUT index for an input x, one must keep in mind that the main goal of the LUT is to approximate the non-linear functions as quickly as possible, in to maximize speed as accuracy has already been somewhat compromised. Hence, finding the right index in minimal time is acutely important. This project proposes an algorithm to finding the right LUT index using a hardware-oriented algorithm, which would be implemented purely with combinational logic, taking minimal toll on the time of the overall operation.

C++ simulations were designed for each function to find and test the algorithms to find the correct LUT index i given input x. The simulations used only bit shifts, additions, subtractions, and comparisons and are thus well suited to be implemented on hardware. For a floating point represented number x, with sign bit s, exponent part e and mantissa part m, the algorithms proposed are as follows:

tanh/Sigmoid LUT algorithm

1. Calculate unbiased exponent, $ube = e - 127$
2. split to cases according to ube. Each value of ube represents a range of numbers.
3. For each range, set i to a predetermined value, that being the range's left most value's LUT index.
4. Shift m by a different value for each range. These values were determined empirically.
5. $i \mathrel{+}= shifted\_m$
6. Return $LUT[i]$.
    - For tanh: the sign bit is set as that of the output value.
    - For sigmoid: if the sign bit is 1 the value returned is $1 - LUT[i]$

Each activation function algorithm varies slightly with different range splits, and within several ranges the most-significant-bits of the mantissa were used to further split into sub-ranges and refine the index addition.

Mean square error (MSE) was used to evaluate the algorithms and help refine them when needed. They are as follows:

$$MSE(tanh) = 0.0005$$
$$MSE(sigmoid) = 0.0018$$

8-bit version

Specifically for an 8-bit floating point architecture, there are only $2^8 = 256$ representable numbers. Therefore, the most efficient method would be to handwrite a single precalculated output value for each one of the 256 available input values. This way a single $8b \cdot 256 = 256B = 2Kb$ LUT is enough for each activation function.

2.6.4 Element-wise modules

This unit is a part of the LSTM Cell (2.6.2) and performs element-wise addition and multiplication on single data elements. The activation functions image is some number $x \in [-1,1]$. Therefore, both element-wise addition and multiplication are designed to receive a floating point 8-bit number. Those element-wise addition and subtraction are not the focus of this work and therefore were not optimized performance-wise. In addition, it should be noted that 8-bit floating-point arithmetic adds significant error to the calculation which is resulted by the limited expressiveness of the model. Future work can be done with regards to transitioning to IEEE single or double precision floating point formats which should decrease error marginally.

Addition

Andrew Ingram and Ronalee Lo's 8-bit floating-point adder\subtractor[1] was used as design-ware for this project with some minor adaptations. It receives $in\_a[7:0]$, $in\_b[7:0]$, $mode$ and outputs $res\_[7:0]$. $mode$ is used to select between normal SV $'+'$ operation to the implemented floating-point addition. The algorithm consists of 5 main stages:

1. Extracting the sign of the result.
2. Subtracting the 2 exponents: $E_{diff} = |E_a - E_b|$.
3. Shifting the smaller number's mantissa by E_{diff} bits.
4. Add the mantissa of the larger number with the shifted mantissa of the smaller one.
5. Normalizing the result back to 8-bit FP representation.

This block's verification was done manually, and its performance was found to be satisfactory for the purpose of this work.

Multiplication

The multiplication block was written in-house. Its inputs are $in\_a[7:0]$, $in\_b[7:0]$, $mode$ and its output is $res\_[7:0]$. $mode$ is used to select between normal SV $'*'$ operation to the implemented floating-point multiplication. The algorithm consists of 5 main stages:

1. Extracting the sign of the result S_{res}.
2. Adding the two exponents: $E_{sum} = E_a + E_b - bias$
3. Multiply the two mantissas: $M_{prod} = M_a * M_b$
4. Normalize the result by shifting M_{prod} back to 8-bit floating point representation while subtracting E_{sum} accordingly.
5. Output $res = \{S_{res}, E_{sum}, M_{prod}\}$

This block's verification was done manually, and its performance was found to be satisfactory for the purpose of this work.

[1] https://pages.hmc.edu/harris/class/e158/01/proj01/fpadd2.pdf

2.7 Performance

2.7.1 Vector matrix multiplication

The number of cycles it takes for the algorithm proposed to update the output vector $\vec{c}$ is $O(\gamma n)$ operations. Using the maximum number of PEs $p = \frac{m}{2}$ we can achieve $O(\gamma n) = O\left(\frac{m}{p}n\right) = O(n)$ operations. This is marginally better than the trivial implementation which results in $O(n^2)$. In comparison with the algorithm proposed in the article summarized in 1.7, it is evident that the systolic array provides a few advantages:

1. Scalability: while the parallel implementation mentioned in the article requires a number of PEs which is linearly dependent on the size of the input, the systolic array algorithm works on any given input size using a fixed number of PEs.
2. Flexibility: the systolic array implementation can achieve linear time complexity as analyzed above while also functioning properly for any given p which allows the user to maximize the tradeoff between resource usage and time complexity in an efficient and easy manner.

Given $FEATURES = 4, P = 4$ we get $M = 9, \gamma = 3$. In this case, the number of cycles needed to perform this algorithm is:

$$(1) \quad \#Cycles = INIT + \gamma \cdot m + FIN = 3 + 27 + 4 = 34 \ \ Cycles$$

Where $INIT = P - 1$ is the number of cycles needed to insert the first $P - 1$ input elements to the array and FIN is the number of cycles needed to flush out all relevant output data from the array.

Each cycle's propagation delay was bound by pe_clk which times the data flow of the module. Referring to section 2.4 one finds that this module should take no less than the following:

$$(2) \quad \frac{f_{clk}}{f_{pe\ clk}} \cdot \#Cycles \cdot (time\ units\ per\ cycle) = 8 \cdot 34 \cdot 10 = 2720[time\ units]$$

This is a significant improvement on the trivial, serial multiplication which should take roughly:

$$(3) \quad \frac{f_{clk}}{f_{pe\ clk}} \cdot \#Cycles \cdot (time\ units\ per\ cycle) = 8 \cdot 64 \cdot 10 = 5120[time\ units]$$

Simulation results with the aforementioned $FEATURES, P$ agree with our analysis as can be seen by figure 2.20:

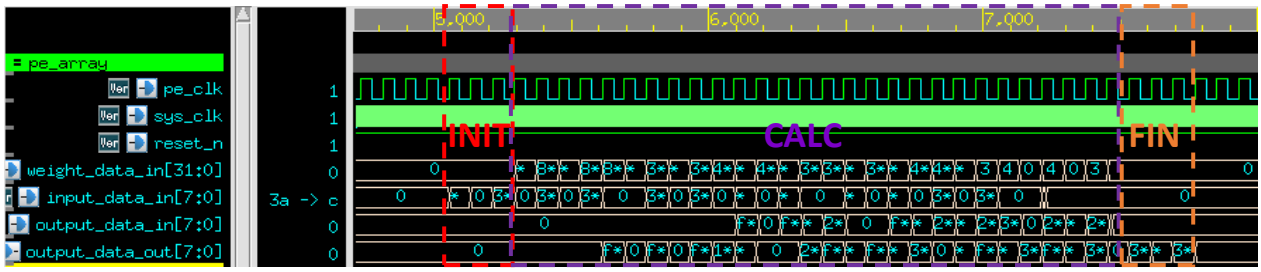

Figure 2.20: simulation results of matrix vector multiplication module with $FEATURES = 4, P = 4$

The $INIT$ phase occurs between $t = 5054$ to $t = 5294$, the calculation ends by $t = 7454$ and the FIN phase ends by $t = 7774$ which comes to a total of exactly $7774 - 5054 = 2720$ time units. For $f_{clk} = 200MHz$ we get $t_{pd_{sys}} = 0.005\mu s \cdot 272 = 1.36\mu s$.

2.7.2 Deserializer

This Block's input is the serial output data read from the vector matrix multiplication module output buffer. The block's output is the same data held in a parallel manner. The process should take no less than $2 \cdot FEATURES\ \ clk$ cycles to complete. simulation results for $FEATURES\ = 4$ can be seen in figure 2.21:

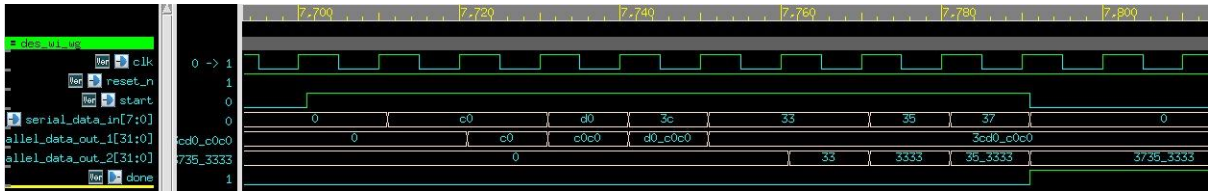

figure 2.21: simulation results of deserializer block for $FEATURES\ = 4$

As can be seen from the above figure there is a $9\ clk$ cycle gap between the start and done signals. For $f_{clk} = 200MHz$ we get $t_{pd_{deserializer}} = 0.005\mu s \cdot 9 = 45ns$.

2.7.3 LSTM Cell

LSTM Cell consists of 3 stages. In each stage some combination of addition, multiplication and activation is applied to the data as explained in 2.6.2. Hence, each stage's propagation delay can be bound by a single $cell\_clk$ cycle. In addition, the module must write output data to an output buffer which should take $FEATURES\ clk$ cycles. Given this calculation, LSTM Cell latency cannot precede $\frac{f_{cell\ clk}}{f_{clk}} \cdot stages + FEATURES = 8 \cdot 3 + 4 = 28[clk\ cycles]$.

Actual simulation results are shown in figure 2.22:

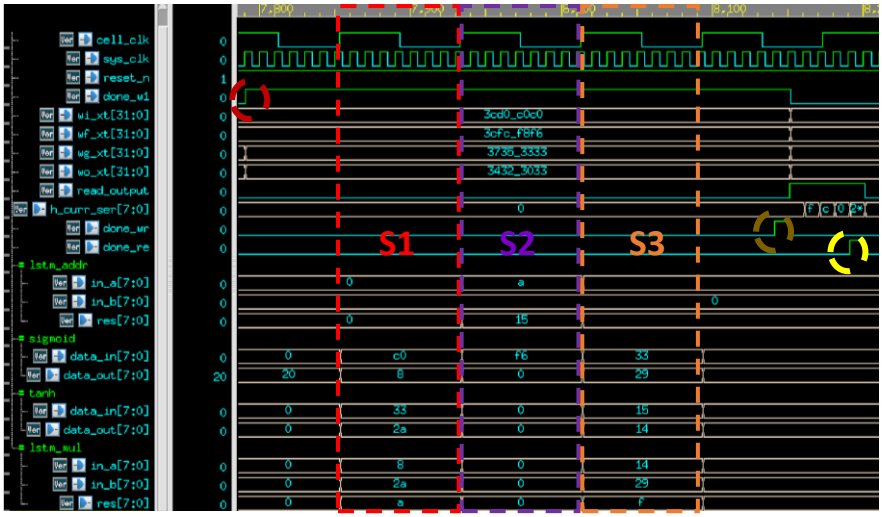

Figure 2.22: simulation results of the LSTM cell module

The first circle (7791) indicates that the previous block (deserializer) is done. In the next cell_clk rising edge (7853), the LSTM cell begins operation. It toggles between 3 stages as already specified in section FILL HERE. This process is finished after 3 cell_clk cycles (8093). FEATURES+1 (5) clk cycles later the done_wr signal is asserted, signifying that the output data has been written to the output buffer, indicated by the brown circle (8141). FEATURES+2 (6) clk cycles later, the done_re signal is asserted, signifying that the output data was read from the output buffer, as indicated by the yellow circle (8191). The execution requires a total of $8141 - 7853 = 288$ time units which are equivalent to roughly 29 clk cycles. The simulation results are within a clock cycle of the theoretical analysis and probably could be optimized in future work. For $f_{clk} = 200MHz$ we get $t_{pd_{cell}} = 0.005\mu s \cdot 29 = 145ns$.

2.7.4 Full Cycle

$INIT\_W1, INIT\_W2$ each occurs once in standard operation (at initialization) and therefore not analyzed in this section. Hence, the full cycle is defined as one runthrough of operation modes $W\_IN, CALC, R\_OUT$. W_IN and R_OUT should each take $FEATURES$ clk cycles as in each mode $FEATURES$ data elements are read into or out of the system. Combining the results of 2.7.1, 2.7.2, 2.7.3, one can conclude that the $CALC$ stage should take $272 + 9 + 29 = 310$ clk cycles. In actuality, there are some wasted clock cycles in the transition of data between blocks that are timed with clk to blocks that are timed with pe_clk or $cell\_clk$ as can be seen for example in figure 2.22 - the deserilizer signals that the data is ready for the LSTM Cell but the cell can begin S1 only in the next $cell\_clk$ rising edge. simulation results for $FEATURES = 4$ can be seen in figure 2.23:

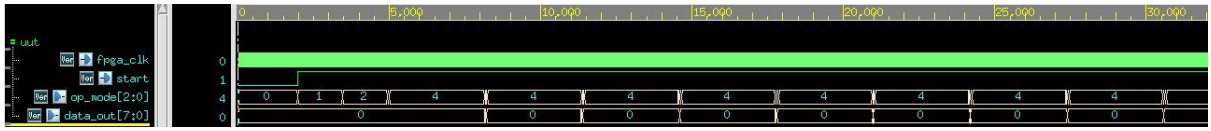

Figure 2.23: simulation results with LSTM top as uut, for $FEATURES = 4$

As can be seen in the above figure, each cycle's duration is $320\ clk$ cycles (equivalent to 3200 time units) and for $f_{clk} = 200MHz$ we get $t_{cycle} = 0.005\mu s \cdot 320 = 1.6\mu s$.

2.7.5 Summary

The summary of the system's latency can be seen in table 2.5:

Module name	**Simulated Latency $[\mu s]$**
Vector matrix multiplication	1.36
Deserializer	0.045
LSTM Cell	0.145
Total	1.6

Table 2.5: Summary of the simulated latency of the system

Since every module's input is dependent on the last module's output (including a dependency between LSTM Cell output at time t and the input of Vector matrix multiplication module at $t + 1$), the system in its whole does not operate in a pipeline manner and its throughput is simply:

$$(4)\ TP_{tot} = 0.625MOps$$

Where Ops stands for operations per second.

2.8 Synthesis

2.8.1 Technology and constraints

The technology used in this synthesis is Tower's $0.18\mu m$. The constraints of this synthesis amounted to a $5ns$ clock cycle which is equivalent to the $200MHz$ clock frequency used in the article in section 1.7.

2.8.2 Floor Plan

The entire floorplan can be seen in figure 2.24:

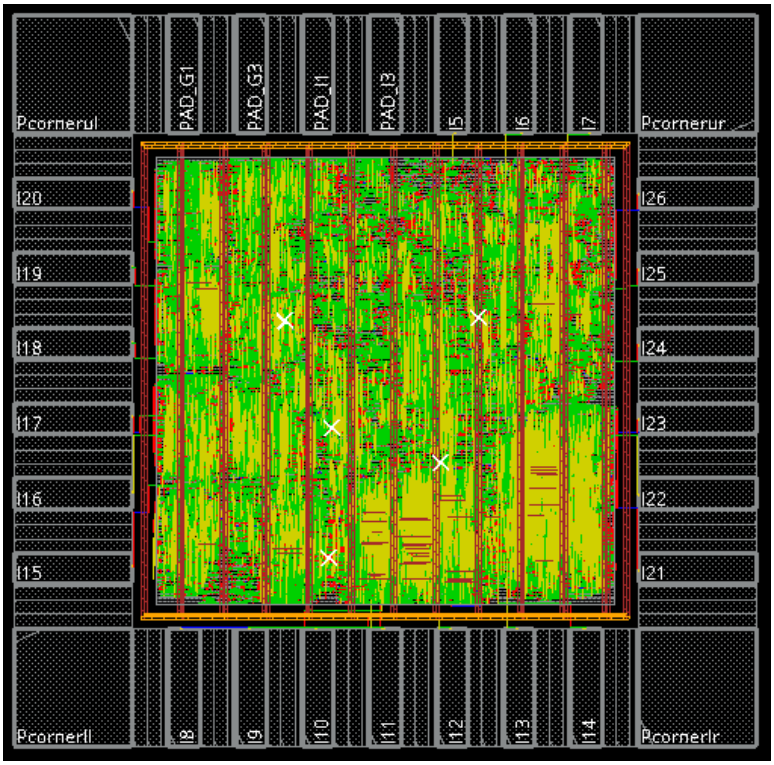

Figure 2.24: Floorplan of the entire system

Figure 2.25 highlights the areas reserved for the weight matrices RAM and its peripheral shuffling method:

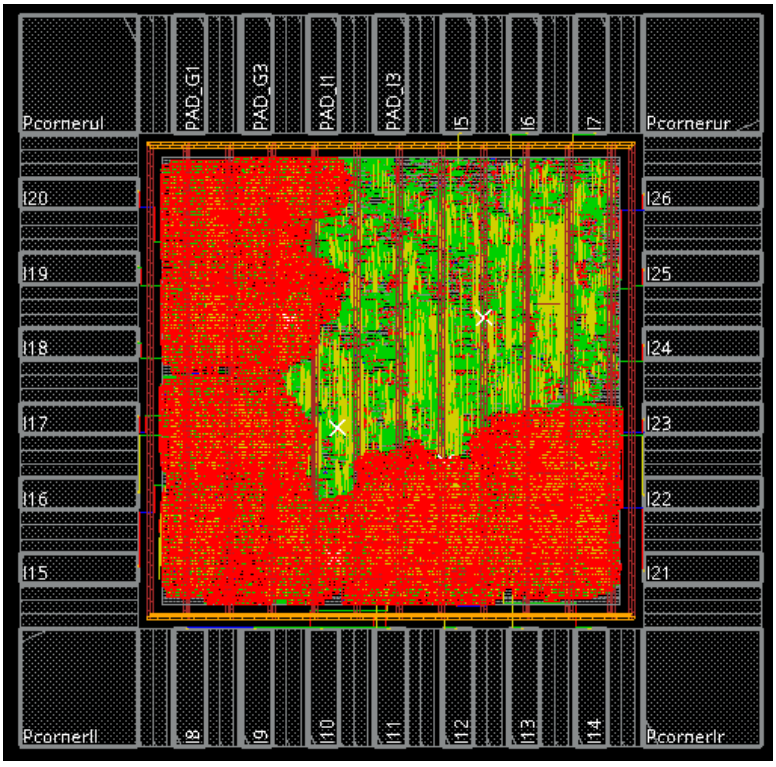

Figure 2.25: weight matrices buffers and weight data shufflers highlighted in red

Figure 2.26 highlights the areas reserved for the two PE arrays:

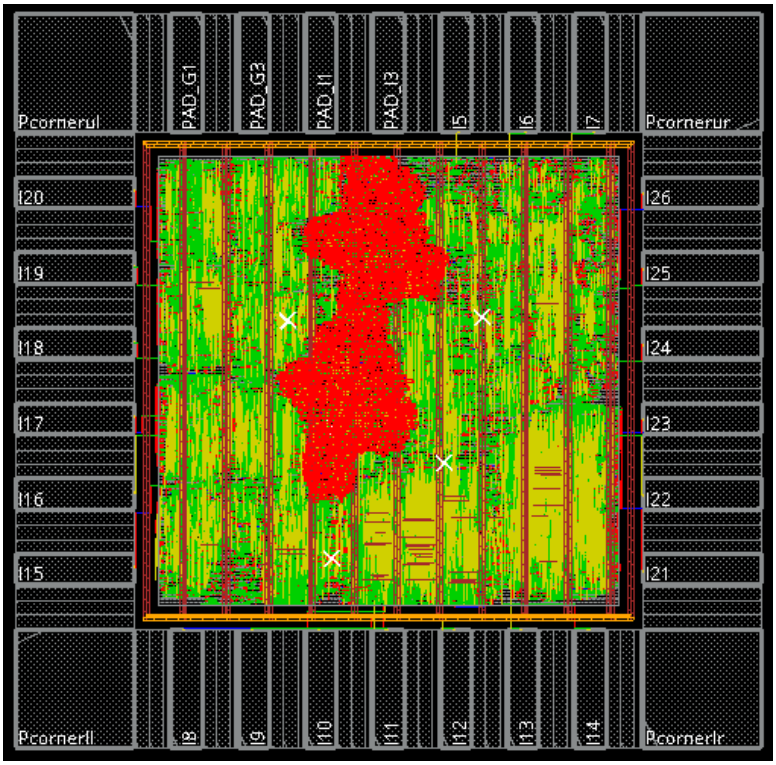

Figure 2.26: PE arrays highlighted in red

Finally, figure 2.27 highlights the location reserved for the LSTM Cell:

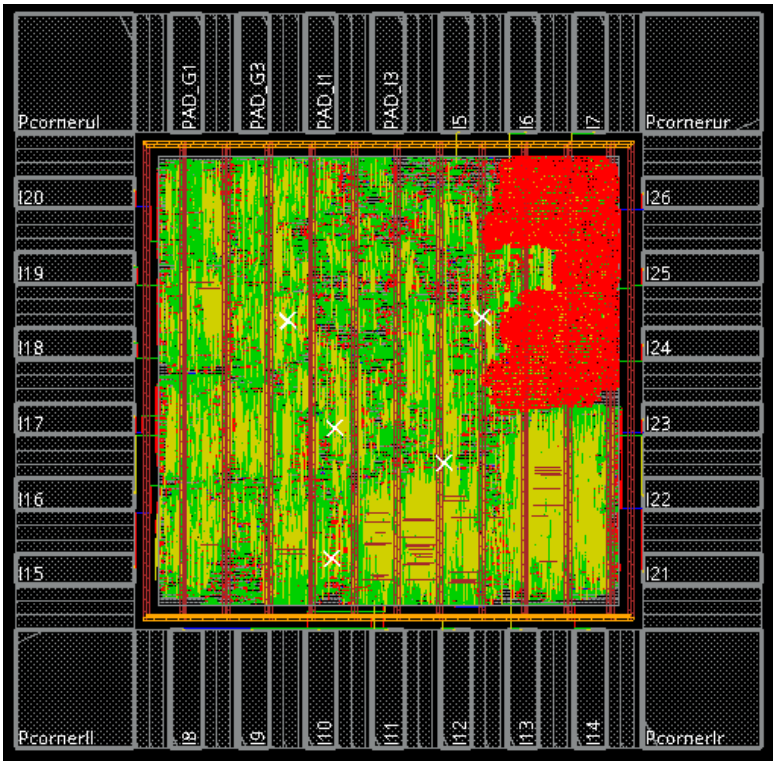

Figure 2.27: LSTM Cell Highlighted in red

2.8.3 Area and Static Power

Area

Area usage is specified in units of dual-input NAND gates and can be seen in table 2.6:

Category	NAND Gates Count
Combinational area	16575.75
Buf/Inv area	3477.25
Non-combinational area	30406.00
Net Interconnect area	19498.63
Total cell area	46981.75
Total area	66480.38

Table 2.6: Area usage of design

Power

The power usage of the design can be viewed in table 2.7:

Power Group	Internal Power	Switching Power	Leakage Power	Total Power
Clock network	0.0	0.9484	268.3600	0.8915
register	14.2153	1.0170e-02	6.1200e+05	14.2261
sequential	1.5191e-02	3.8281e-06	6.0086e+04	1.5255e-02
combinational	0.1885	1.2106	2.9998e+05	1.3994
Total	14.3621 mW	2.1692 mW	97.233 μW	16.5323 mW

Table 2.7: Power usage of design

2.8.4 Timing – Worse Slack Path

Using Synopsys analysis tools, we ran an analysis of worse slack path. The worse path had a slack of $1.71ns$ out of the entire $5ns$ clock cycle which is roughly 34% of the cycle. The following is a detailed description of this path:

1. **Start-point:** matrix_vector_multiplication_1/output_top/mode_reg[1].
2. **End-point:** matrix_vector_multiplication_1/output_top/output_dpr/data_out_reg[0].
3. **Description:** This path begins at the mode register - the mode of operation for the block in charge of the output of the matrix_vector_multiplication module. Since the output RAM is used both to propagate data through the PE array and to write data outside to the next part of the system (can be seen in figure 2.1), it utilizes a mode register to choose which external data_out it should be connected to at any given time.
4. **Detailed Path:**

Point	Increment	Total Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
wf_wo_arr/output_top/mode_reg[1]/CP (dfcrq1)	0.00	0.00
wf_wo_arr/output_top/mode_reg[1]/Q (dfcrq1)	0.30	0.30
wf_wo_arr/output_top/U37/ZN (inv0d0)	0.29	0.60
wf_wo_arr/output_top/U36/ZN (nr02d1)	0.26	0.86
wf_wo_arr/output_top/U8/Z (aor22d1)	0.26	1.12
wf_wo_arr/output_top/output_dpr/U32/ZN (inv0d0)	0.00	1.12
wf_wo_arr/output_top/output_dpr/U31/ZN (nd02d1)	0.21	1.33
wf_wo_arr/output_top/output_dpr/U19/ZN (inv0d0)	0.32	1.65
wf_wo_arr/output_top/output_dpr/U127/ZN (aoi22d1)	1.04	2.69
wf_wo_arr/output_top/output_dpr/U126/ZN (aoi21d1)	0.08	2.77
wf_wo_arr/output_top/output_dpr/U125/ZN (oan211d1)	0.20	2.97
wf_wo_arr/output_top/output_dpr/U124/ZN (oan211d1)	0.05	3.02

wf_wo_arr/output_top/output_dpr/data_out_reg[0]/D	0.15	3.17

5. **Slack Analysis:**

Path Time	3.17
Library Setup Time	0.12
Total Delay	3.17+0.12 = 3.29
Clock Cycle	5
Slack	5-3.29 = 1.71ns

2.9 Verification

2.9.1 Methodology

This section briefly describes the verification methodology used in this project. The verification process was performed bottom-up, meaning each individual block was designed along side a designated test-bench. Each test-bench verified the normal operation of a single block before connecting it to the rest of the design. The systolic array of the matrix-vector multiplication module warranted additional actions – a *c*-based software simulator, alongside an excel macro were designed specifically to follow, visualize, and verify the data propagation through the array. The code for this simulator is attached to this book as appendix FILL_HERE. After making sure that both the Matrix-vector multiplication module and LSTM Cell functioned properly, connections were drawn alongside peripheral components such as the deserializer, and the entire unit was tested using a designated test-bench.

2.9.2 Unit-level Verification

2.9.2.1 Matrix-Vector multiplication

In the verification of this module, we used regular arithmetic multiplication and addition as opposed to the 8-bit floating point units. The reason for that is that 8-bit floating point calculation presents rounding errors to the calculation which are hard to evaluate a-priori. Rounding errors decrease marginally when moving to higher bit count representation of floating-point numbers, however, this was not the purpose of this paper and therefore was not pursued.

Verification of this module was done manually and compared step-by-step to an excel spreadsheet built to demonstrate the algorithm's process. Presented here is a single simulation for a non-trivial case of the following characteristics:

$$FEATURES = 4 \quad (27)$$

$$W_i = \begin{pmatrix} 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \end{pmatrix} \quad (28)$$

$$W_g = \begin{pmatrix} 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \end{pmatrix} \quad (29)$$

$$(30) \quad [x_t, h_{t-1}] = \begin{pmatrix} 5 \\ 6 \\ 7 \\ 8 \\ 16 \\ 6 \\ 6 \\ 5 \end{pmatrix}$$

The output of this module in this case is:

$$(31) \quad \left[W_i \cdot [x_t, h_{t-1}], W_g \cdot [x_t, h_{t-1}]\right] = (236 \quad 236 \quad 203 \quad 177 \quad 177 \quad 118 \quad 118 \quad 118)^T$$

Where the first 4 elements corelate to the product of the input vector with W_i and the last 4 elements corelate to the product with W_g. Full view of the simulation results for this specified test case can be viewed in figure 2.28:

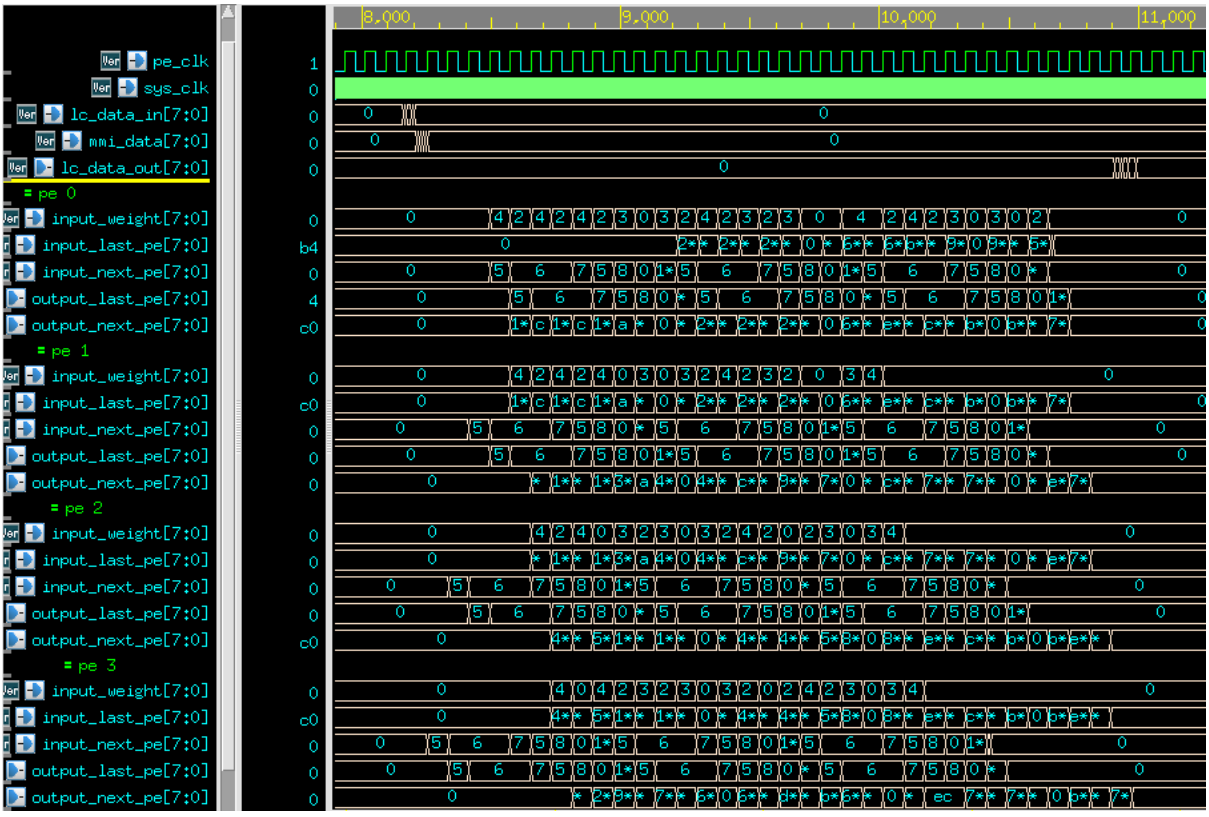

Figure 2.28: full view of systolic array simulation

Roughly at the $8.2K$ mark, input data enters the array, first the signal $lc\_data\_in[7:0]$ transmits the $FEATURES = 4$ elements of h_{t-1} and then mmi_data holds the x_t input. The input data enters the PE array from PE 3's $input\_next\_pe[7:0]$ signal and flows through the array visiting each PE's $input\_next\_pe[7:0]$ for exactly one cycle until finally leaving the array through PE 0's $output\_last\_pe[7:0]$. Output data is read from the output buffer and enters through PE 0's

$input\_last\_pe[7:0]$ flowing through the array and leaving through $output\_next\_pe[7:0]$ of $PE\ 3$'s where it is written to the relevant address of the output buffer. Other than the external interfaces described above, Adjacent PEs are connected such that $input\_last\_pe[7:0]$ of $PE\ i$ is connected to $output\_next\_pe[7:0]$ of $PE\ i-1$ etc.

Enlargement of a couple of cycles of the process can be seen in figure 2.29:

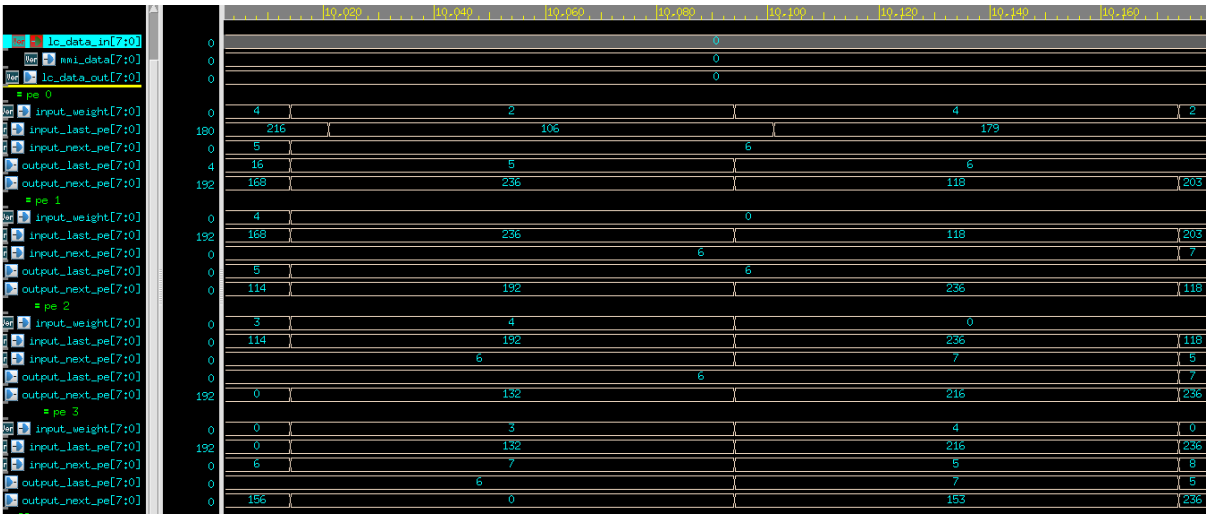

Figure 2.29: zoom of 2 cycles of systolic process

The connections between the PE's can be seen in the above figure more clearly. In addition, a comparison to the excel spreadsheet is made in figure 2.30:

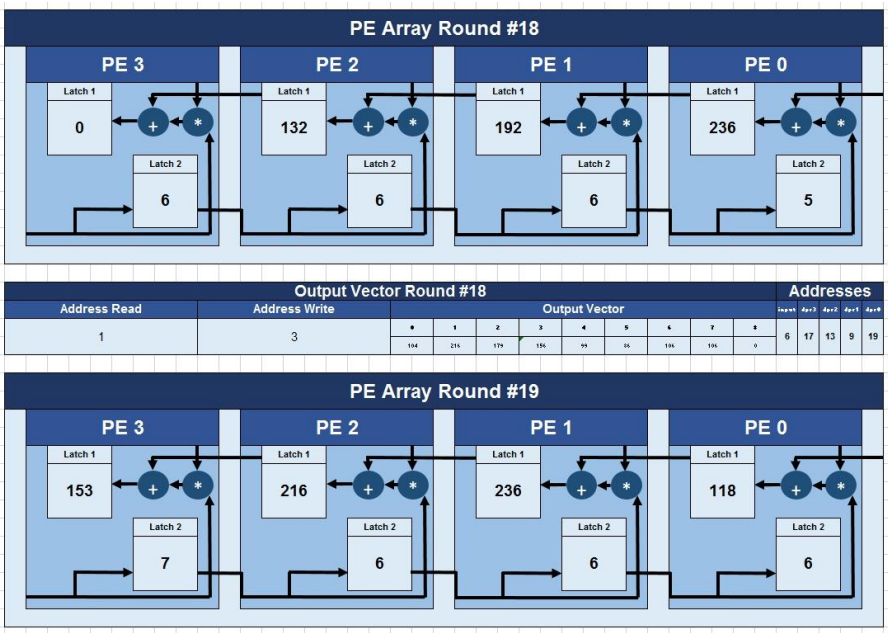

Figure 2.30: Same 2 cycles as in figure 2.29 shown in excel spreadsheet

Finally, we show that the $lc\_data\_out[7:0]$ signal shown to be changing roughly at the $11{,}000$ mark in figure 2.31 holds the relevant output data:

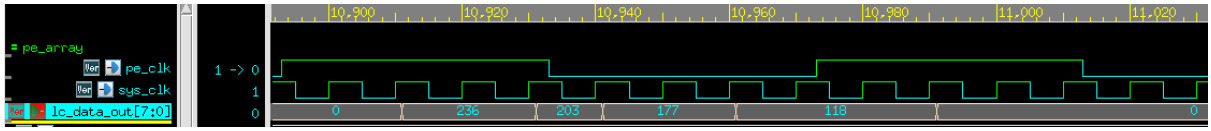

Figure 2.31: output data of systolic array simulation

Please note that this signal is timed by sys_clk (clk) and not pe_clk as there is no need for the slower clock used to time the multiplications in the array. Therefore, it is evident that the output is indeed as stated in equation (31).

2.9.2.2 LSTM Cell

Verification of this module was done manually and compared step-by-step to expected output, using the Verdi software. Since all $FEATURES$ data elements go through the same process parallelly, the following paragraph follows a single element's flow through the cell. In addition, it should be noted that some error is expected to be inserted to the process due to the 8-bit floating point arithmetic's limitations (please refer to section 2.6.4 for elaboration).

The verification was done for some chosen cycle where $FEATURES = 4$. The full operation of the cell in the chosen cycle can be seen here:

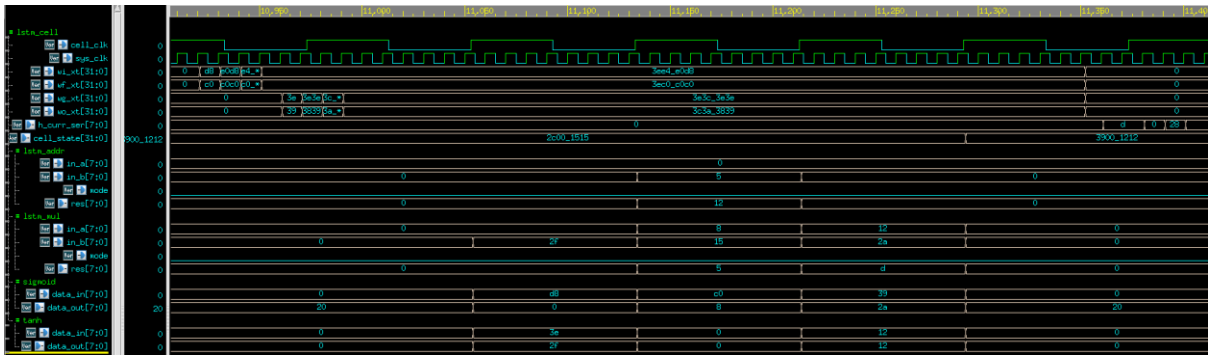

Figure 2.32: Verification of LSTM Cell, full operation view, one feature

Over the next paragraphs, the path of the first data element will be followed in detail.

Input

The first data element (which is corelated to the first feature) enters the LSTM cell as:

Name	Hexadecimal Value	Decimal (given FP Representation)
$W_i x_t[7:0]$	$d8$	-6.0
$W_f x_t[7:0]$	$c0$	-2.0
$W_g x_t[7:0]$	$3e$	1.875
$W_o x_t[7:0]$	39	1.5625
$C_{t-1}[7:0]$	15	0.328125

Table 2.4: Verification of LSTM Cell, inputs table

The inputs are ready around the 11,000 timeline and can be seen in the full view figure or the following:

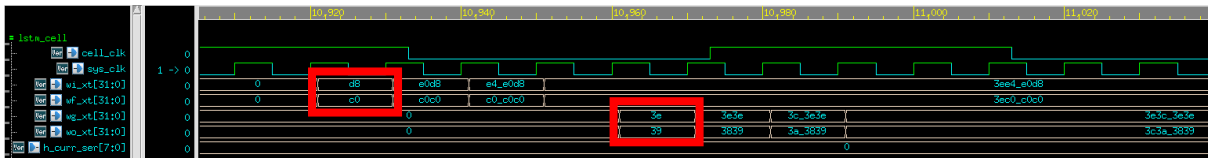

Figure 2.33: Verification of LSTM Cell, inputs to the cell

S1

In this stage, the following result should be calculated:

$$(32) \quad res_{s1} = \sigma(W_i x_t[7:0]) \cdot \tanh\left(W_g x_t[7:0]\right)$$

In the above case:

$$(33) \quad res_{s1} = \sigma(-6.0) \cdot \tanh(1.875) = 0.0025 \cdot 0.954 = 0.0024$$

Since the smallest number in our 8-bit FP representation is 0.015, we would expect res_{s1} to be 0. In the following figure, we can see the actual simulation results of S1 in detail:

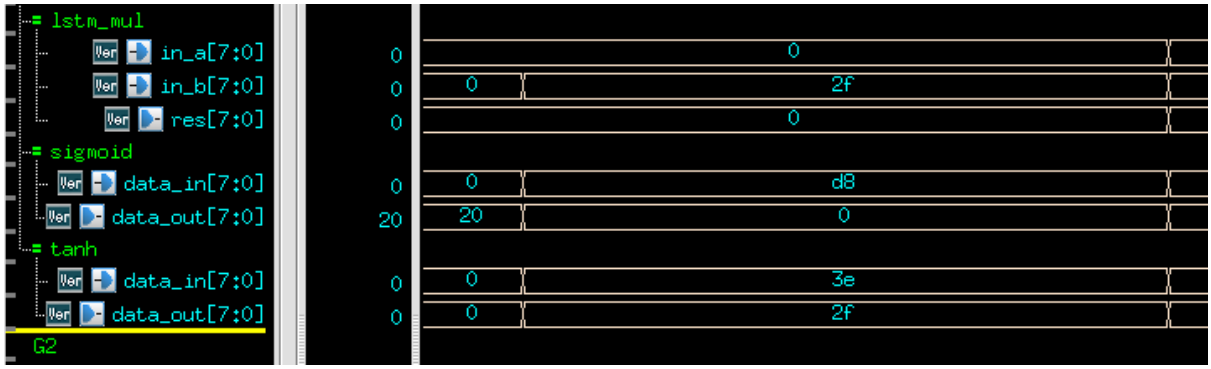

Figure 2.34: Verification of LSTM Cell, S1

It is evident that:

a. $W_i x_t[7:0] = (d8)_{hex} = (-6.0)_{dec}$ is the input of the sigmoid and the output is 0.

b. $W_g x_t[7:0] = (3e)_{hex} = (1.875)_{dec}$ is the input of the sigmoid and the output is $(2f)_{hex} = (0.96875)_{dec}$. This is the closest number to the actual 0.954 expected in the FP representation.

c. The inputs of the multiplier are 0 and $(2f)_{hex}$ and the output is 0.

S2

In this stage, the following result should be calculated:

$$(34) \quad res_{s2} = \sigma\left(W_f x_t[7:0]\right) \cdot C_{t-1}[7:0] + res_{s1}$$

In the above case:

$$(35) \quad res_{s2} = \sigma(-2.0) \cdot 0.328125 + 0 = 0.1192 \cdot 0.328125 = 0.03908$$

Actual simulation results:

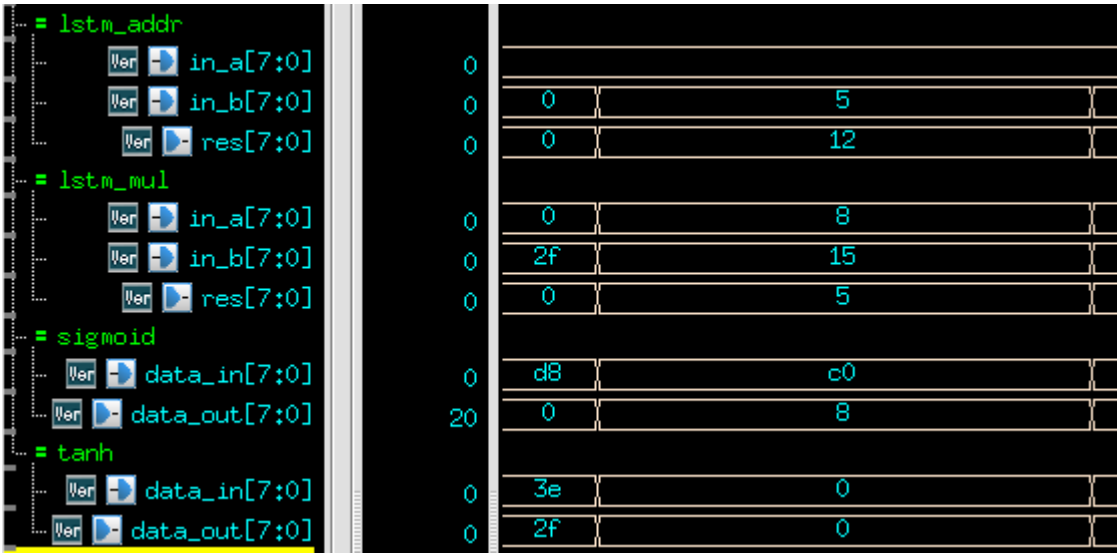

Figure 2.35: Verification of LSTM Cell, S2

One can see from the above figure that the input of the sigmoid in this stage is indeed $c0$ and the output is $(8)_{hex} = (0.125)_{deci}$. in addition, the inputs of the multiplier are $(8)_{hex}$ and $(15)_{hex}$ while the output is $(5)_{hex} = 0.078125$. Finally, you can see that $res_{s1} = 0$ and $(5)_{hex}$ are the inputs to the adder block as intended. $res_{s2} = (12)_{hex} = 0.28125$.

S3

In this stage, the following result should be calculated:

$$(36) \quad res_{s3} = \tanh(res_{s2}) \cdot \sigma(W_o x_t[7:0])$$

In the above case:

$$(37) \quad res_{s3} = \tanh(0.28125) \cdot \sigma(1.5625) = 0.2741 \cdot 0.8267 = 0.2266$$

Actual simulation results:

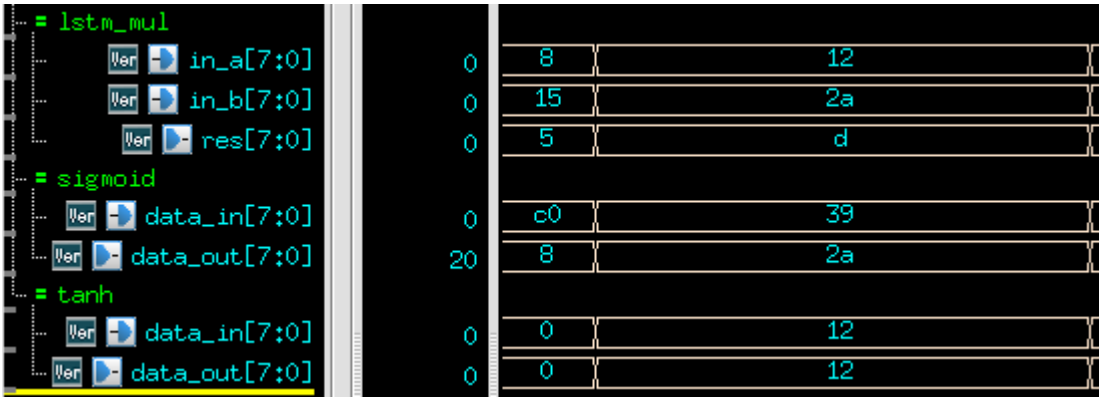

Figure 2.36: Verification of LSTM Cell, S3

It is evident that:

a. The input of tanh is res_{s2} and its output is $(12)_{hex} = (0.28125)_{deci}$.

b. The input of sigmoid is $(39)_{hex}$ and the output is $(2a)_{hex} = 0.8125$.

c. The inputs of the multiplier are the results of a and b and its output is $(d)_{hex} = (0.203125)_{deci}$.

Output

The module updates the $cell\_state[7:0]$ to be res_{s2} and $h\_curr\_ser[7:0]$ to be res_{s3} as can be seen in the following figure:

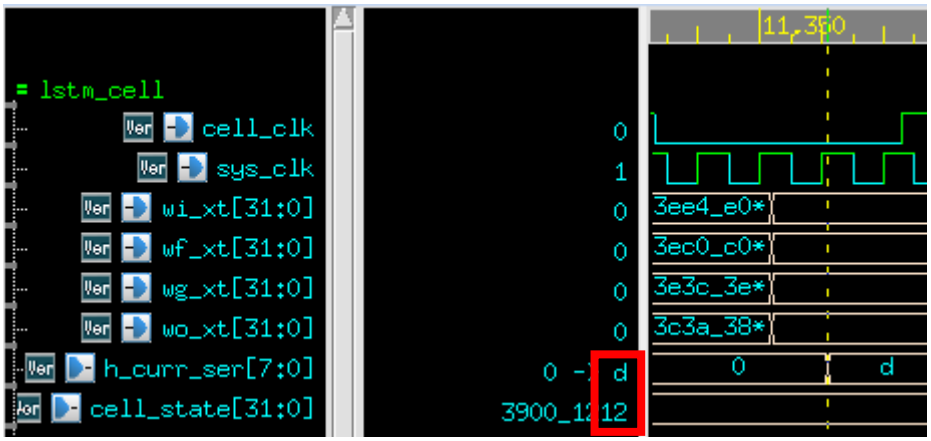

Figure 2.37: Verification of LSTM Cell, Output

2.9.3 System-level Verification

In this section we will show that the seams between the modules function properly. We will assume that both the Matrix-Vector Multiplication module and the LSTM Cell function as intended (please refer to 2.9.2 for elaboration).

2.9.3.1 Matrix-Vector multiplication to Deserializer

This section proves that the connection between "$output\_top$" – the block that contains the output RAM of the systolic array and "des_wi_wg" which is the deserializer which connects to the RAM and in charge of propagating the data forward in a parallel manner, is functioning properly. In the following figure, the reader can see the connection:

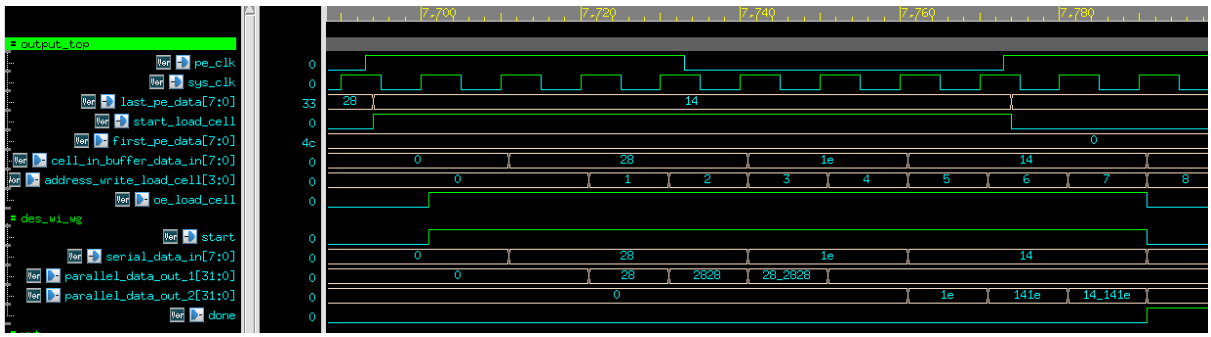

Figure 2.38: connection between matrix-vector multiplication module and deserializer

The data enters the RAM through "$last\_pe\_data$" (out of order, please refer to 2.6.1 for elaboration). The data is then read out of the RAM through "$cell\_in\_buffer\_data\_in$" which is connected to "$serial\_data\_in$" of the deserializer. The reader can see that the entering data is then held in a parallel manner in the logics "$parallel\_data\_out\_1$" and "$parallel\_data\_out\_2$".

2.9.3.2 Deserializer to LSTM Cell

This section proves that the connection between "des_wi_wg" which is the deserializer which is connected to the output of the matrix-vector multiplication module and the LSTM Cell, which is its output, is functioning properly. In the following figure, the reader can see the connection:

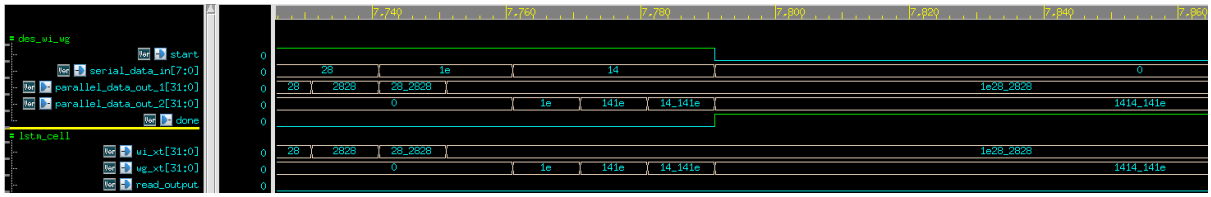

Figure 2.39: Connection between deserializer and LSTM Cell

The deserializer output "$parallel\_data\_out\_1$" and "$parallel\_data\_out\_2$" are the LSTM Cell's inputs "wi_xt" and "wg_xt" accordingly.

2.9.3.3 LSTM Cell to output

This section proves that the connection between LSTM Cell's output and the output of the system is functioning properly. In the following figure, the reader can see the connection:

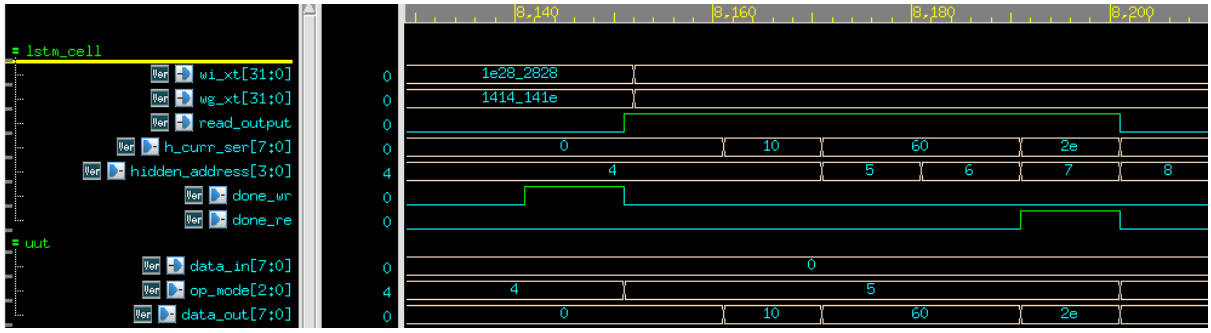

Figure 2.40: Connection between LSTM Cell and output

LSTM Cell's output data is delivered outside through "h_curr_ser" and the system's data out "$data\_out$" updates accordingly.

2.9.3.4 LSTM Cell to input

This section proves that the connection between LSTM Cell's output and the matrix-vector multiplication input is functioning properly – this connection represents the "memory" of the LSTM system, please see 1.5 for elaboration. In the following figure, the reader can see the connection:

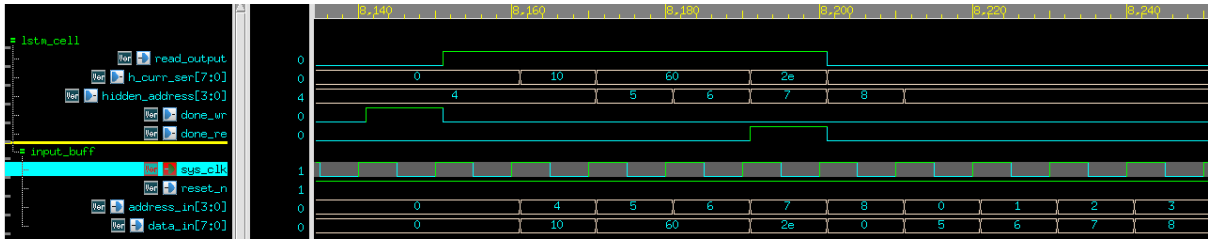

Figure 2.41: Connection between LSTM Cell and output

LSTM Cell's output data is delivered outside through "h_curr_ser". This data enters "$input\ buff$" which is the RAM that holds the input data to the matrix-vector multiplication module. The reader can see that the data enters addresses 4-7 out of 7 by looking at "$address\_in$". When "$address\_in$" is at 8 no data is being written into the RAM and right after that the addresses 1-3 are being written from the external memory. No data has been lost in the process.

2.9.3.5 Summary and multi-cycle simulation

In summation, we have shown each module functions properly on it's own in 2.9.2 and we have shown in this section (2.9.3) that the modules interconnects are in order. The following final figure depicts the top block's signals throughout many operation cycles:

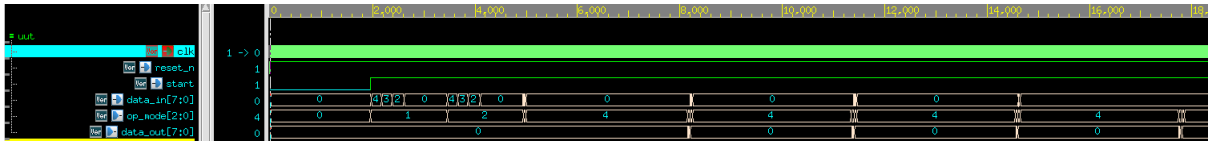

Figure 2.42: Multi-cycle figure

The reader can see that right after $start$ signal rises, $INIT\_W1$ stage begins ($op\_mode = 1$). When this is done, $INIT\_W2$ begins ($op\_mode = 1$). Right after that, the system enters a loop: constantly changing between the rather short W_IN and W_OUT ($op\_mode = 3{,}5$ accordingly) and the very long $CALC$ ($op\_mode = 3$) during which the entire calculation is being performed.

2.10 Programmer's Guide

When using the device, one should pay attention to the following:

2.10.1 Hardware Parameters

The project uses hardware parameters to make the code more readable and enable one to easily adjust these parameters. This is done from the top hierarchy of each of the modules. The entire code, excluding the floating-point element wise modules, is adjustable in this manner. The parameters, by module, are as follows:

$FEATURE\_BITS$: The number of bits required to represent the number of features (for example, in our 8-feature implementation, 4 bits are required.)

$ELEMENT\_BITS$: The number of bits in an element.

$FEATURES$: the number of features in an input vector.

P: The number of PEs in the systolic array module.

M: A parameter used to simplify systolic array calculations (and pad the input vector). It is defined like so:

$$m = \begin{cases} n & if\ n\ is\ odd \\ n+1, & if\ n\ is\ even \end{cases}$$

Where n is the number of elements in the input vector.

$GAMMA$: Is used to count semi-circles of each element through the array during the multiplication, and is defined using m & p:

$$\gamma = \left\lceil \frac{m}{p} \right\rceil$$

2.10.2 Memory Requirements

During both initialization and calculations, the unit uses information stored in an outer memory source. For the data to be processed correctly, it must adhere to the following limitations:

Input: each input vector should be comprised of $FEATURES$ elements. Each element shall be comprised of $ELEMENT\_BITS$ bits. Input vector should be fed into the system from first element to last, one after the other according to 2.5 x_t transaction instructions.

Output: each output vector should be comprised of $FEATURES$ elements. Each element shall be comprised of $ELEMENT\_BITS$ bits. Output vector should be read from the system from first element to last, one after the other according to 2.5 h transaction instructions.

Weight matrix: each weight matrix should be comprised of $FEATURES\ \ X\ \ 2 \cdot FEATURES$ elements. Each element shall be comprised of $ELEMENT\_BITS$ bits. Weight matrices should be fed into the system in pairs, W_i with W_g and W_f with W_o, from first element to last with no intermission in between paired matrices. Please refer to 2.5 $INIT\_W1$, $INIT\_W2$ transaction instructions.

2.10.3 APB Registers

The following table shows the APB registers in the system, their function, and initial values (init):

Name	Address	Function	Init
PSEL	0x00	User generated, signals the system is selected	0
PENABLE	0x01	User generated, active from the second cycle of transfer	0
PWDATA	0x02	User generated, active when user write transfer is selected	0
PREADY	0x03	System generated, signals an extension of transfer	0

Table 2.8: APB registers

2.10.4 Driver Guidelines

The following section describes in general points the driver one is required to write to be able to communicate with the system.

1. Start-point: op_mode signal is IDLE, system is at rest.
2. Write initial values to the APB registers as shown in table 2.8.
3. Hold $start$ signal at '1' for at least $5ns$.
4. Transfer of weight data:
    a. Write '1' to $PSEL$ and $PWDATA$
    b. Wait for $5ns$
    c. Write $'1'$ to $PENABLE$ and write valid weight data to $data\_in[7:0]$.
        - See 2.9.2 for an elaboration
    d. Wait for $PREADY$ to be '1' – write '0' to $PENABLE$.
5. Transfer input data:
    a. Wait $5ns$
    b. Write '1' to $PENABLE$ and write valid input data to $data\_in[7:0]$.
    c. Wait for $PREADY$ to be '1' – write '0' to $PENABLE$, $PWDATA$ and $PSEL$.
6. Read Output data:
    a. Wait for op_mode to be W_OUT
    b. Write '1' to $PSEL$
    c. Wait for $5ns$
    d. Write $'1'$ to $PENABLE$ and read valid output data from $data\_out[7:0]$.

e. Wait for $PREADY$ to be '1' – write '0' to $PENABLE$.

7. Go back to stage 5.

2.11 Alternate Designs

In the following section, the main alternative designs considered will be explained, a discussion managing the tradeoff between each of the designs is conducted, and finally, the reasoning for the writer's choice is explained.

2.11.1 Time Scale

To conduct the discussion, defining a time scale is required. Given $ELEMENT\_BITS = 8$, number of bits per element, as we have previously shown in this chapter, the most time costly indivisible operations are multiplications which can be bound by $O(ELEMENT\_BITS^2)$ clock cycles. Using the FPGA chosen in the article in 1.7, this number is equivalent to approximately:

$$t\_basic\_operation = \frac{k^2}{f} = \frac{8^2}{200 \cdot 10^6} = 0.32\mu[s]$$

To free the discussion from this parameter, we will from now on consider a new time axis where this time unit is equal to $1[\tilde{t}]$.

2.11.2 Design Principles

The design of the pipeline can take many forms. To limit the discussion to a select few, some guidelines are in order:

1. The number of matrix-vector multiplication modules is critical – it is the most resource costly module of the design.
2. With equivalent importance, the throughput of the pipeline should be considered.
3. Some element-wise calculation must be done after the matrix-vector multiplication $W_o X_t$ and before starting a new matrix-vector multiplication (since the next operation $W_* X_{t+1}$ depends on the output h_{t+1}). Every design reviewed in the following paragraphs will aspire to minimize the time it takes to conduct this calculation as during this time the matrix-vector multiplication modules are in idle. For convenience this time will be denoted in the following as $\delta\tilde{t}$. Minimizing this time means using enough resources such that the time is not dependent on input size n.

2.11.3 Alternate Designs Description

Following the guidelines in section 2.11.2, the following will analyze 4 main designs:

1. **D1 -** one Matrix-Vector multiplication modules
2. **D2 -** two Matrix-Vector multiplication modules
3. **D3 -** three Matrix-Vector multiplication modules
4. **D4 -** four Matrix-Vector multiplication modules

For each design, a pipeline diagram, resource utilization, and throughput analysis are shown in this section. A summary of said analysis can be seen in table 2.9.

The Y-axis of each diagram is indicative of the resource being used where:

- MV Mul #num – matrix vector multiplication module number #num.
- Sigmoid, tanh, multiplier – element wise operations.

The X-axis is consistent with time and is represented in units of $\frac{n}{3}[\tilde{t}]$ where n is the size of input and $\tilde{t}$ was defined in section 2.11.1 as a rough estimation of the top boundary of time needed to perform the largest indivisible operation (element-wise multiplication).
Each block in the diagram is representative of an operation that corresponds to its X-axis location being performed at a time which corresponds to its Y-axis location and is a part of a calculation of the output at a time which corresponds to the block's text.

2.11.4 Option D1

A pipeline of this design is shown in figure 2.43:

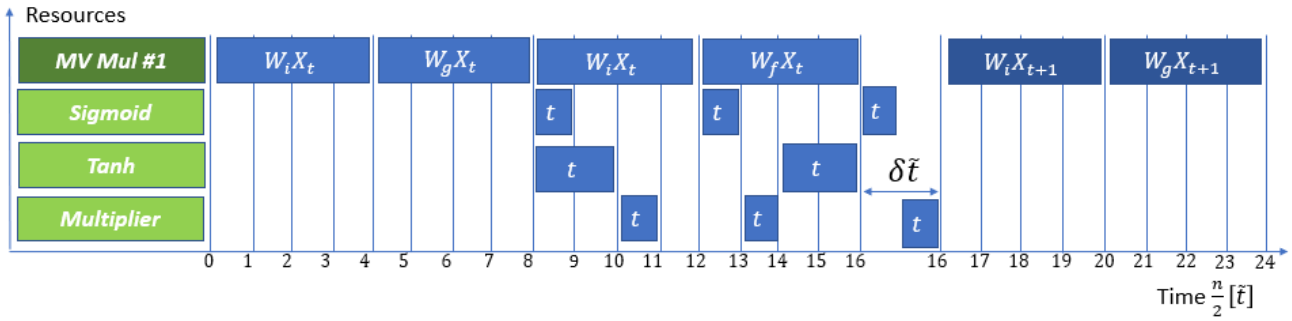

Figure 2.43: pipeline of design D1

The throughput is $\frac{1}{8n\tilde{t}+\delta\tilde{t}}$ where the time frame $\delta\tilde{t}$ contains two basic indivisible operations meaning $\delta\tilde{t}_{D1} = 2\tilde{t}$ meaning $TP_{D1} = \frac{1}{(8n+2)\tilde{t}}$. the resources required are 1 matrix vector multiplication module which is roughly 100% utilized, 1 tanh activation module which is roughly 25% utilized and n of sigmoid and element-wise multiplier modules which are roughly 0% utilized.

2.11.5 Option D2:

A pipeline of this design is shown in figure 2.44:

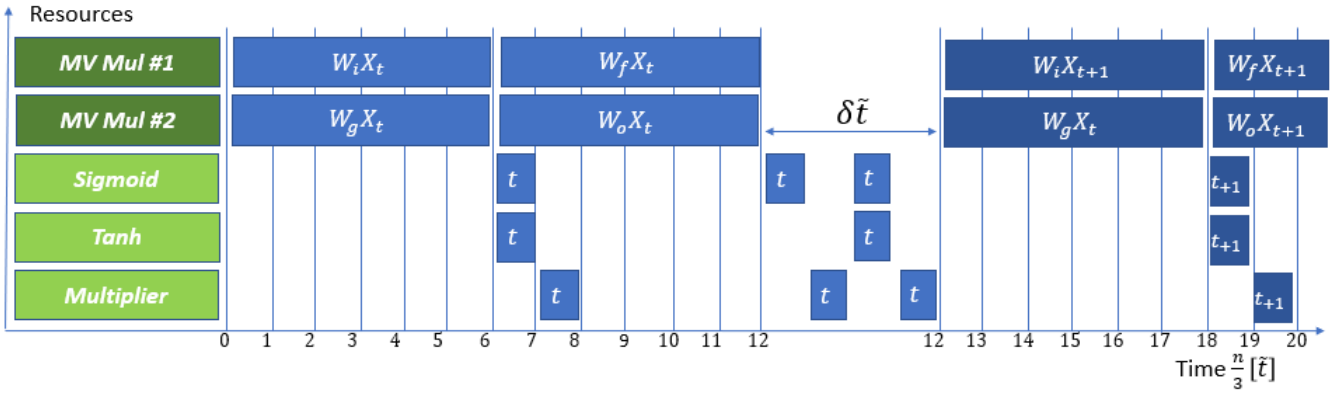

Figure 2.44: pipeline of design D2

The throughput is $\frac{1}{4n\tilde{t}+\delta\tilde{t}}$ where the time frame $\delta\tilde{t}$ contains 4 cycles of basic indivisible operations meaning $\delta\tilde{t}_{D2} = 4\tilde{t}$ meaning $TP_{D2} = \frac{1}{(4n+4)\tilde{t}}$. the resources required are 2 matrix vector multiplication module which are roughly 100% utilized, and n of sigmoid, tanh and element-wise multiplier modules which are roughly 0% utilized.

2.11.6 Option D3:

A pipeline of this design is shown in figure 2.45:

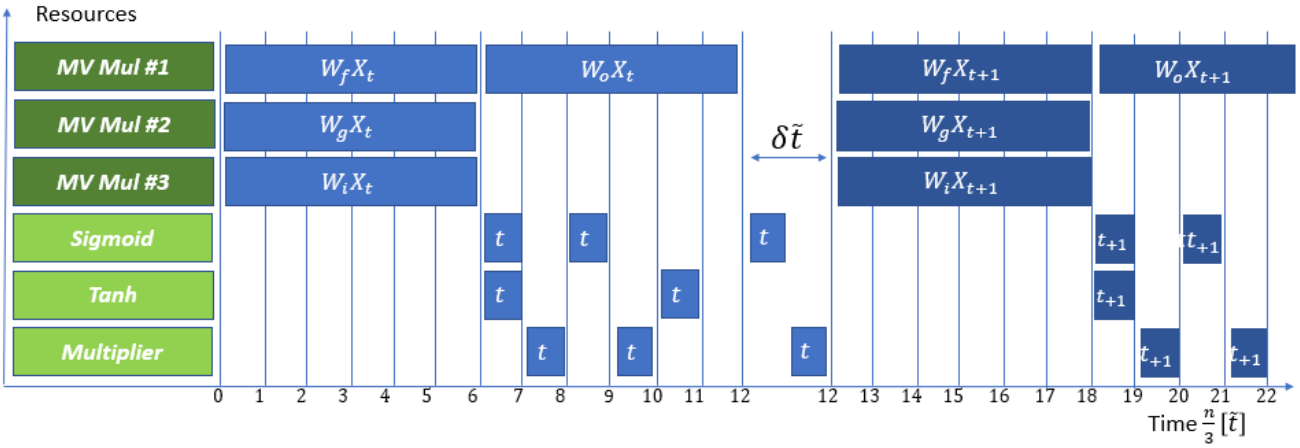

Figure 2.45: pipeline of design D3

The throughput is $\frac{1}{4n\tilde{t}+\delta\tilde{t}}$ where the time frame $\delta\tilde{t}$ contains 2 cycles of basic indivisible operations meaning $\delta\tilde{t}_{D3} = 2\tilde{t}$ meaning $TP_{D3} = \frac{1}{(4n+2)\tilde{t}}$. the resources required are 3 matrix vector multiplication module which are roughly 67% utilized, 1 tanh activation module which is roughly 50% utilized and n of sigmoid and element-wise multiplier modules which are roughly 0% utilized.

2.11.7 Option D4:

A pipeline of this design is shown in figure 2.46:

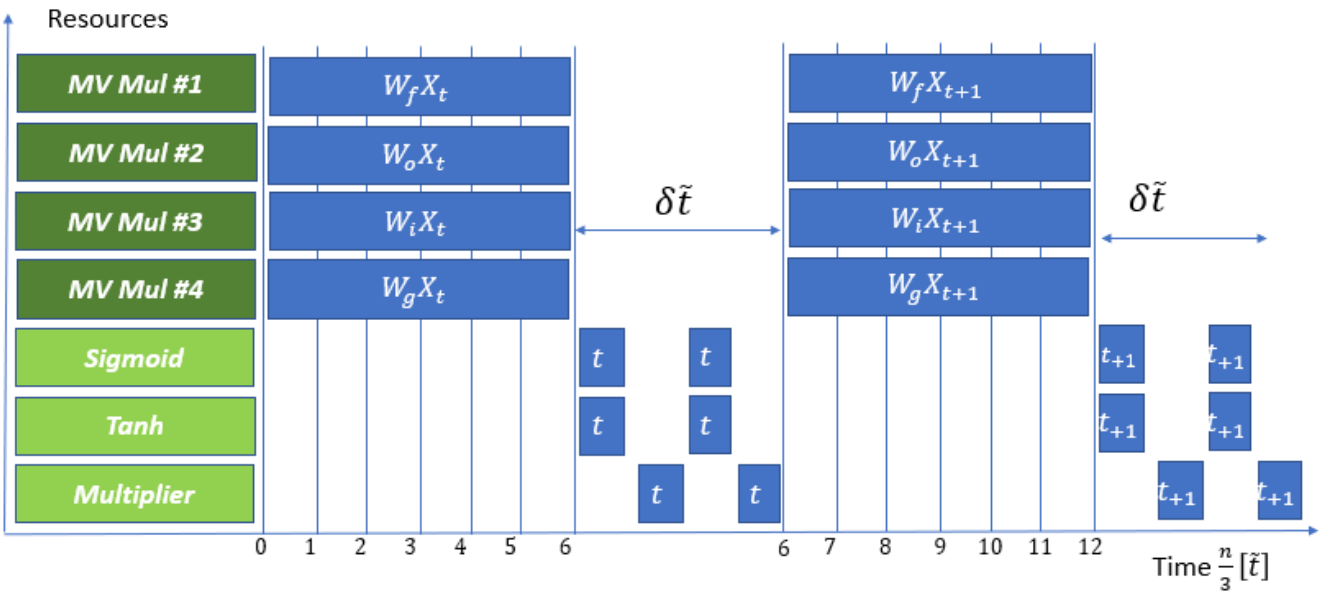

Figure 2.46: pipeline of design D4

The throughput is $\frac{1}{2n\tilde{t}+\delta\tilde{t}}$ where the time frame $\delta\tilde{t}$ contains 4 cycles of basic indivisible operations meaning $\delta\tilde{t}_{D4} = 4\tilde{t}$ meaning $TP_{D2} = \frac{1}{(2n+4)\tilde{t}}$. the resources required are 4 matrix vector multiplication module which are roughly 100% utilized, $2n$ of sigmoid and element-wise multiplier modules which are roughly 0% utilized and finally n of tanh activation function modules which are roughly 0% utilized. Alternatively, $D4^*$ design is implemented using n of sigmoid, tanh and element-wise multiplier modules which are roughly 0% utilized in which case $\delta\tilde{t}_{D4^*} = 6\tilde{t}$.

2.11.8 Summary of analysis

Method	Sigmoid		Tanh		Element-wise Multiplier		Matrix-vector multiplier		Latency$[\tilde{t}]$
	Number of modules	%Utilization	Number of modules	%Utilization	Number of modules	%Utilization	Number of modules	%Utilization	
D1	n	0	1	25	n	0	1	100	$8n+2$
D2	n	0	n	0	n	0	2	100	$4n+4$
D3	n	0	1	50	n	0	3	67	$4n+2$
D4	$2n$	0	n	0	$2n$	0	4	100	$2n+4$
D4*	n	0	n	0	n	0	4	100	$2n+6$

Table 2.9: Summary of analysis of alternative designs

The above table clearly shows that there are 2 viable options - D_2 and D_4^*. Both utilize the same number of element-wise modules while the former utilizes only half the matrix multiplication modules of the latter and provides twice the latency. In this project, D_4, the faster, more resource demanding option was chosen.

2.11.9 $\delta\tilde{t}$ Design

This section will provide a more specific description of the $\delta\tilde{t}$ time frame design given the chosen D_4^* option. The element-wise operations required in this time frame are divided into 3 stages according to figure 1.13 precisely like the article in 1.7. a detailed look at the resulting pipeline for a single element can be seen in figure 2.47:

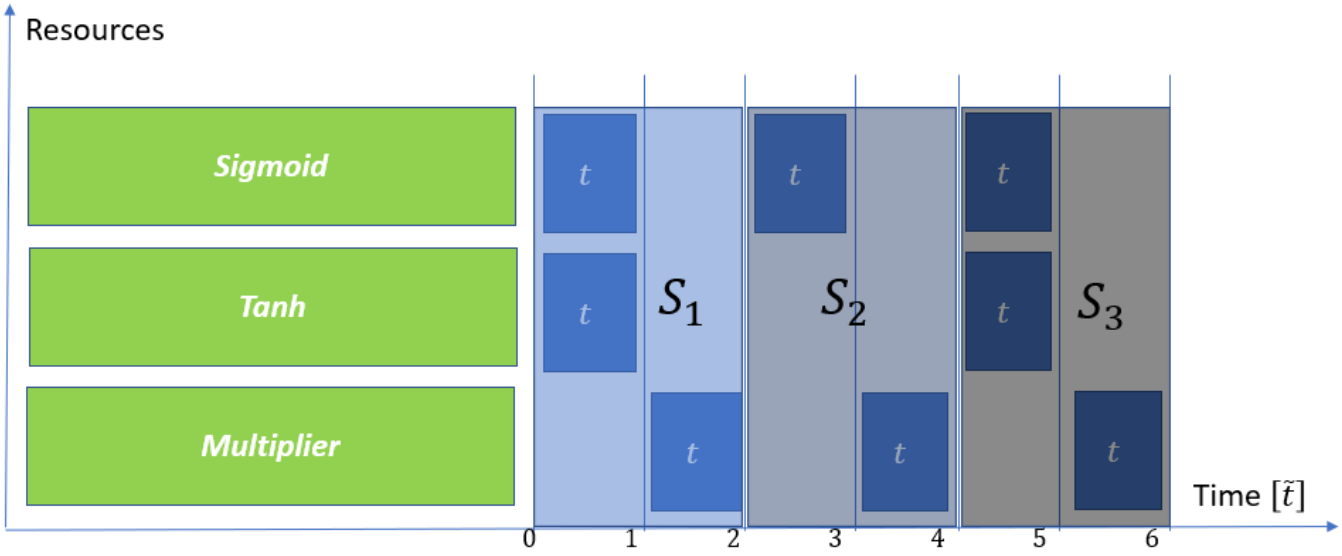

Figure 2.47: $\delta\tilde{t}$ time frame design

As described in the previous section, this architecture requires 1 of each element-wise multiplier, tanh and sigmoid and there are n identical copies of it in the final design (one for each element of the input vector).

3. Conclusion

The following section will review the project from the authors' perspective. We will elaborate on the process we went through, our main takeaways, the improvements to the original design (Section 1.7) we have implemented and some of the future work we recommend pursuing.

Before starting the architectural design of our system, we took some time to deepen our understanding of basic terms and definitions in AI and ML as well as to learn the structure of a basic LSTM. In addition, we also learned about the APB interface. Chapter 1 of this project is a summary of what we learned during this phase of the work. This part of the project took us 3 weeks to complete.

Approaching the architectural design, we had the baseline described in the article reviewed in section 1.7 to rely on. Upon further investigation of different designs of matrix multiplication blocks, we decided to go in a different direction from the article and implemented a systolic array. The reason for going with this design choice is two-fold. The first reason is technical. We wanted the advantages the systolic array offered, mainly the flexibility between time and physical-space and an option of a set physical-size for any number of features. The second reason is didactic. We were drawn to the option of implementing a more complex design and challenging ourselves in the flexible time frame a project like this offers. The fact that this particular design choice is used in many deep-learning applications is a nice added bonus. Designing the entire system took us roughly 7 weeks.

After finishing the design of the system, we began the implementation. Working bottom-up, we started with the smallest block in the matrix-vector multiplication module and built it from the ground up, integrating each tiny block to its neighbor before moving forward. Only after the matrix-vector multiplication module was fully integrated and verified, we moved to the next main block in the design – the LSTM cell. Finally, we integrated all the parts together, built a controller to handle the entire system and built the external interfaces. The process described in the above paragraph was the longest phase of this project, taking us a combined 12 weeks with some intermissions.

Once the project was implemented and verified, we synthesized the design and performed the layout of the chip using Tower's "design vision" and "innovus" tools. In doing so we learned about those phases in the VLSI path and encountered some of the common problems a design might face in the transition from Verilog simulation to physical components. This process took us 4 weeks.

Finally, we completed the writing and editing of this project book. The task did not consume much time as it was done throughout the project, taking a total of 3 weeks.

Designing, implementing, and verifying such a design, which is much more complex than everything we have attempted before, taught us many things regarding VLSI in particular and project management in general. Firstly, we learned the importance of solid architectural foundation. We believe that the time it took us to implement and verify the design could have been greatly shortened had we done a more meticulous job in the architectural design. For example, some of the problems we encountered were simply a matter of ill-defined interfaces between blocks regarding dimensions or timing. Secondly, we understood more deeply the importance of organization in managing such a project. Pre-defining hardware parameters and definition of done for each of the blocks are just a couple of the helpful tips we take away from this experience. Lastly, we greatly deepened our understanding both of VLSI, LSTM, and systolic architecture, giving us a solid foundation to rely on in future endeavors.

Although this was a great didactive experience conducted in a flexible time frame, there is still much work to be done. First, we recommend integrating into the system an efficient 32-bit FP arithmetic unit. This was not one of the project goals and so was not our focus, but we believe that the 8-bit FP arithmetic implemented here is not nearly accurate enough for practical work. Secondly, for the 8-bit representation, we settled for implementing our $tanh, sigmoid$ activation functions with logic gates instead of RAM. We made this in the interest of simplifying our synthesis but in future work we recommend considering transitioning to RAM technology. Lastly, we would suggest improving on the basic LSTM Cell design offered here, optimizing it for a specific task and testing its performance. Some already proved features, such as peepholes or coupling the forget and input gate, could be an easy and efficient way to boost performance.

Finally, we would like to thank the VLSI lab of the faculty of electrical engineering in the Technion for the opportunity to pursue this project. Special thanks goes to Goel Samuel for providing and supporting the developmental environment and to Shahar Gino for the mentoring and great help throughout this project.