Imperial College London

Real-time Drift Prediction and DNA Detection through on-chip AI Processor

Author
Xuyang Cui
CID: 02256317

Supervised by
Prof.Pantelis Georgiou

A Thesis submitted in fulfillment of requirements for the degree of
Master of Science in Analogue and Digital Integrated Circuit Design

Department of Electrical and Electronic Engineering
Imperial College London
2023

Abstract

This paper implemented two dedicated hardware systems of Recursive Neural Network (RNN) computation in [1]. Based on the framework, a finely grained and optimized computation platform has been established with Xilinx Intelligence Property (IP)s. As parallelism has been deeply and effectively applied with the focus on this particular network and the hardware system, this version has finally reached an increase in speed for 67.6% comparing to the hardware platform generated from High Level Synthesis (HLS) in The work of [1]. Also, the effectiveness has been improved in term of hardware resource: these is a decrease of 77.2%, 74.6%, 38.8%, 37.5% on LookUp Table (LUT), Flip Flop (FF), Digital Signal Processing (DSP) and Block Random Access Memory (BRAM) resources respectively which remaining an accuracy of 7.7×10^{-5} Root-mean-square Error (RMSE) in 100 test cases compared to groundtruth.

Besides, this paper also established fully-customized and completely proprietary IPs of float point adder, multiplier and Coordinate Rotation Digital Computer (CORDIC) implementation of $Tanh$ and $Sigmoid$ function, etc that would replace Xilinx's computation IPs in first version, expanding its opportunity to be used in commercial application and further Application Specific Integrated Circuit (ASIC) implementation. Similarly, this version has reached an increase in speed for 67.9% comparing to [1]'s work and shows a decreased accuracy of 1.54×10^{-1} on RMSE error in 100 test cases compared to groundtruth. And 39.4%, 76.8%, 40% of LUT, FF and BRAM resources has been saved while no DSP resources is used as this version is targeted to a further ASIC implementation.

With senior student's help, an ASIC implementation using Taiwan Semiconductor Manufacturing Company (TSMC) 65nm Low Power technology has been established. For this version, the chip is of 2583213 μm^2 area, 98.9% module density and within the timing constraint of 100MHz operating clock.

To sum up, this paper presented two version of optimization for the work of [1] on both performance and hardware resources. While the first version has minor improvement on the accuracy, the second one shows the increased computation error which could be further optimized. Finally the ASIC implementation for the layout of second version is done, unveiling the promising potential

on both the significance and application field of this paper's work.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced.

Copyright Declaration

Acknowledgments

I would like to express my gratitude to my project supervisor Prof.Pantelis Georgiou who offer me the chance to conduct research on such an interesting topic where I could be able to practically combine what I have learnt during MSc degree with the application that is meaningful in real life world.

I would also like to thank to Lei Kuang who together with Prof.Pantelis Georgiou sacrificed valuable time in offering me tremendous help on both the ASIC and hardware design that contributes to the final success and completeness of my project.

Contents

List of Acronyms

AI Artificial Intelligence

IP Intelligence Property

ANN Artificial Neural Network

RNN Recursive Neural Network

CNN Convolution Neural Network

ISFET Ion-Sensitive Field-Effect Transistors

LoC Lab-on-Chip

MLP Multi-Layer Perceptron

CMOS Complementary Metal-oxide Semiconductor

GRU Gated Recurrent Unit

FPGA Field Programmable Gate Array

HLS High Level Synthesis

BRAM Block Random Access Memory

FF Flip Flop

LUT LookUp Table

DSP Digital Signal Processing

ASIC Application Specific Integrated Circuit

RMSE Root-mean-square Error

CORDIC Coordinate Rotation Digital Computer

TSMC Taiwan Semiconductor Manufacturing Company

NLP Natural Language Processing

SRN Simple Recurrent Networks

HTF Hyperbolic Tangent Function

GRU Gated Recurrent Unit

LSTM Long Short-Term Memory

NLP Natural Language Processing

LiGRU Light Gated Recurrent Unit

BN batch normalization

ReLU Rectified Linear Unit

CPU Central Processing Unit

GPU Graphics Processing Unit

II Iteration Interval

ROM Read Only Memory

LSB Least Significant Bit

MSB Most Significant Bit

FSM Finite State Machine

DUT Design Under Test

STA Static Timing Analysis

WNS Worst Negative Slack

RHC Rotation Mode of Hyperbolic CORDIC

VLC Linear Vector Mode of CORDIC

RAM Random Access Memory

DRC Design Rule Check

LVS Layout Versus Schematic

MAC Multiply and Accumulate

List of Figures

1 Introduction

Contents

1.1 Motivation

Recently, Lab-on-Chip (LoC) method[2], [3] has shown great potential in minimizing the size of biomedical instrumentation platform while remaining or even increasing the detection accuracy and speed at the same time. Meantime, the possibility to fabricate ISFET sensors, which could be used in electr-ochemical sensing application like detection of DNA application[4], with unmodified Complementary Metal-oxide Semiconductor (CMOS) technology on LoC chip made this especially promising for its cost-effectiveness, convenience and real-time processing speed[5].

However, apart from advantages, unpleasant system features could affect its accuracy and therefore limits its application field. Chief among these features is the drift phenomenon, which would limit the dynamic range and hence instrument accuracy of the sequencing system, of ISFET sensors caused by the hydration of sensing surface[6].

Some prior works has shown the possibility to utilize AI technology such as RNN or Convolution Neural Network (CNN) network to detect and cancel this drift. The work of [7], [8] and [9] uses RNN, MLP and Artificial Neural Network (ANN) respectively to compensate the drift and all

1

achieves a RMSE error less than 1e-5 in pH detection. However, for low-frequency electro-chemical measurement, the ionic interactions could only be vaguely spotted[1]. The work of [1] uses RNN network with GRU to overcome this and finally reaches 89.71% accuracy in reaction classification.

However, despite these progresses, the work of [1] could still be further optimized in the following three aspects: Firstly, as they only implemented the compensation inference network on Field Programmable Gate Array (FPGA) platform, the poor portability and inconvenience indicates this method could not be directly used in real life. Secondly, as their work only uses HLS for software to hardware projection, this preliminary tool would result to a poorly optimized projected hardware architecture, meaning redundant hardware resource and slow processing speed which would not meet the requirement of real-time processing. Thirdly, the use of Xilinx built-in function library such as 'hls::exp' etc in HLS would cause IP issues, limiting its application field only in academic uses and therefore preventing it to be widely promoted to commercial world where the most applications are.

1.2 Contribution

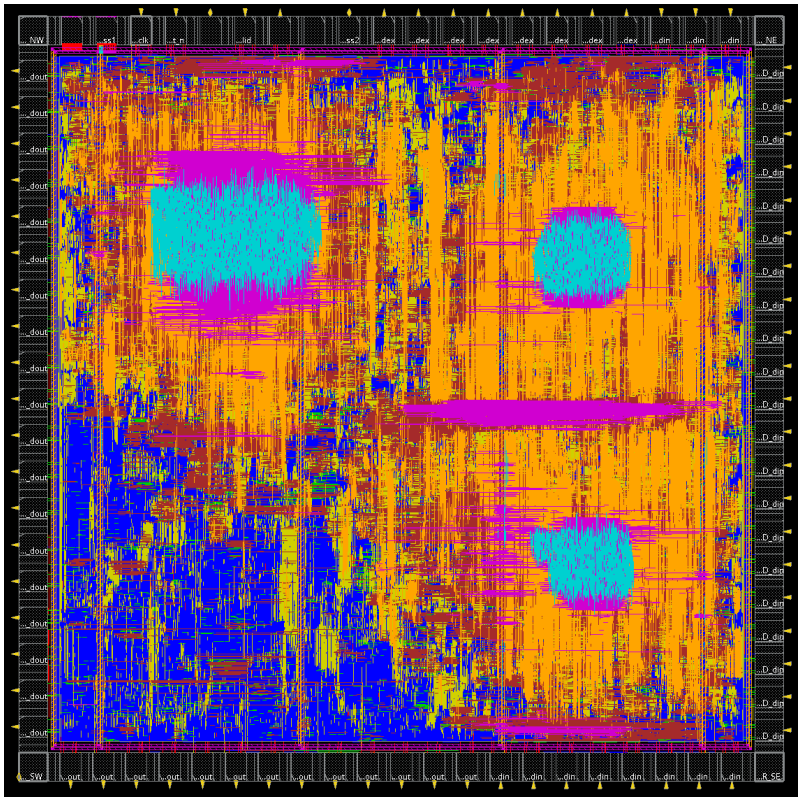


Figure 1.1: Layout of implemented ASIC chip using TSMC 65nm Low Power Technology

This paper has further optimized the work of [1] in three aspects mentioned in last section.

Firstly, this paper implemented a dedicated hardware architecture of RNN network in [1]. Based on the framework, a finely grained and optimized computation platform has been established in Vivado with Xilinx IPs. As parallelism has been deeply and effectively applied with the focus on this particular network and hence the hardware system, this version has finally reached an increase in speed for 67.6% (using 100MHz clock, which is the same with the work of [1]) comparing to the hardware platform generated from HLS in the work of [1]. Also, the effectiveness has been improved in term of hardware resource: these is a decrease of 77.2%, 74.6%, 38.8%, 37.5% on LUT, FF, DSP and BRAM resources respectively which remaining an accuracy of 7.7×10^{-5} RMSE in 100 test cases compared to groundtruth.

Secondly, this paper established fully-customized and completely proprietary IPs of float point adder, float point multiplier and CORDIC implementation of *Tanh* and *Sigmoid* function, etc that would replace Xilinx's computation IPs in first version, expanding its opportunity to be used in commercial application. Similarly, this version has reached an increase in speed for 67.9% (using 100MHz clock, which is the same with the work of [1]) comparing to the hardware platform generated from HLS in the work of [1] and shows a decreased accuracy of 1.54×10^{-1} on RMSE error in 100 test cases compared to groundtruth. And 39.4%, 76.8%, 40% of LUT, FF and BRAM resources has been saved while no DSP resources is used as this version is targeted to a further ASIC implementation.

Thirdly, with senior student's help, an ASIC implementation using TSMC 65nm Low Power technology has been established whose layout is shown as in 1.1. For this version, the chip is of 2583213 μm^2 area, 98.9% module density and coherent in timing which fitting 100MHz operating clock, fulfilling all the requirements in the customized version.

1.3 Thesis Outline

The Chapter 1 introduced the overview of whole project including the motivation and contribution. Chapter 2 and 3 respectively investigated the related literature and launched the analysis from the DNA detecting method to RNN architecture and finally to the hardware realization. The Chapter 4 and 5 implemented two different version of hardware system as the optimized work of [1]. Then the Chapter 6 demonstrated the implementation of ASIC for the second version hardware. The Chapter 7 would compare the performance, resource and accuracy of two proposed hardware with

the work of [1]. Finally the Chapter 8 will re-look the work of the project and point out the further optimization work while the Chapter Conclusion summed up the overall contribution and work of the project.

2

Background

Contents

As the goal of this project is to implemented an AI processor for drift prediction and DNA detection, the RNN structure and its architecture then becomes important as it determines the performance and resource of the system.

In this chapter, basic RNN architecture and inner structure and mechanism is firstly introduced. Then the architecture and inner scheme of widely used GRU unit which is particular dominant for this project is analyzed. Finally the common optimization methodology and features of acceleration system is introduced and relevant real-world implementation examples are presented.

2.1 Recurrent Neural Network

2

Recurrent Neural Network is a type of recurrent neural network that takes sequence data as input, iterates in the evolutionary direction of the sequence, and all the nodes (recurrent units) are connected in a chain[10].

Recurrent Neural Networks have memory, parameter sharing and Turing completeness, so they have some advantages in learning nonlinear features of sequences[11]. Recurrent Neural Networks have applications in Natural Language Processing (NLP), such as speech recognition, language modeling, machine translation, etc, and are also used for various types of time series forecasting. Recurrent neural networks constructed with CNNs are introduced to deal with computer vision problems that contain sequential inputs.

2.1.1 Recurrent Unit

Internal Computation

The core part of an RNN is a directed graph, whose chained elements in the unfolding of a directed graph are called RNN cells[10], [12]. In general, the chained connections of a recurrent cell can be compared to the hidden layer in a feedforward neural network, but in different accounts, the "layer" of an RNN may refer to the recurrent cell of a single time step or to all recurrent cells[13]. Given sequential input learning data $X = \{X_1, X_2, ..., X_T\}$, the length of the RNN expansion is τ. The sequence to be processed is usually a time sequence, and the evolution direction of the sequence is called "time-step". For a time-step, the cyclic unit of RNN has the representation as in 2.1:

$$h^{(t)} = f\left(s^{(t-1)}, X^{(t)}, \theta\right) \tag{2.1}$$

where h is the system status of RNN system.

In the dynamical systems view, the system state describes the change of all points in a given space over time steps[10] and s is the internal status and is related directly to system status where $s = s\,(h, X, y)$.

Since solving for the current system state requires the use of the internal state of the previous time step, the computation of the cyclic unit involves recursion. In the tree-structured viewpoint,

all recurrent units from previous time steps are parents of the recurrent unit at the current time step.

In 2.1, f is either an excitation function or an encapsulated feedforward neural network where the former one corresponding to Simple Recurrent Networks (SRN)s and the latter one is to gating algorithms and some deep algorithms[10]. Common choices of excitation function include logistic function and Hyperbolic Tangent Function (HTF).

The θ in equation 2.1 are weight coefficients within the recurrent unit, which is independent of the time step for a set of learning samples, the RNN computes the output for all time steps using shared weights[10].

On the other side, an RNN consisting only of cyclic units is theoretically feasible, but RNNs usually have another output node, which is defined as a linear function as in 2.2:

$$o^{(t)} = vh^{(t)} + c \tag{2.2}$$

where v, c is the weight coefficient.

Depending on the structure of the RNN, the computational results of one or more output nodes can be obtained as output values after passing through the corresponding output function $\hat{y} = g\left(o\right)$. For example, for the classification problem, the output function can be a normalized exponential function (softmax function) or a classifier built by other machine learning algorithms[10], [11].

Connectivity

There are three common used connectivity topology:

1. Cyclic cell-cyclic cell connection: also known as "hidden-hidden connection" or full connection, in which the state of each cyclic cell at the current time step is determined by the inputs of the time step and the state of the previous time step as shown in equation 2.3:

$$h^{(t)} = f\left(uh^{(t-1)} + wX^{(t)} + b\right) \tag{2.3}$$

   where u, w is the weight of the cyclic node and the former is called the state-state weights while the latter is called state-input weights[11].

2. Output node-cyclic unit connection: the state of the cyclic unit in this connection is determined by the inputs of that time step and the outputs but not the state of the previous time step as shown in 2.4:

$$h^{(t)} = f\left(uo^{(t-1)} + wX^{(t)} + b\right) \tag{2.4}$$

Due to the underlying assumption that the output node of the previous time step is able to characterize the state of all previous time steps, the RNN with output node-loop unit connection is not Turing-complete and has a lower learning capacity than fully connected networks. However, it has the advantage of being able to use Teacher Forcing for fast learning[10].

3. Context-based connection: this connection is also known as closed-loop connection because of the closed-loop structure presented in the view of the graph network, where the system state of the loop unit introduces the true value, $y^{(t-1)}$, of its previous time step. An RNN using context-based connection is a generative model that approximates the probability distribution of the learning objective since the real values of the learning samples are used as inputs during training. There are various forms of context-based connections, a common type of which uses the inputs of the moment, the state of the previous moment, and the true values as shown in equation 2.5 .

$$h^{(t)} = f\left(uX^{(t-1)} + wh^{(t-1)} + Ry^{(t-1)}\right) \tag{2.5}$$

Other types may use fixed-length inputs, use the output of the previous moment instead of the true value, or not use the input of that moment[10].

2.1.2 Output Mode

Over the establishment of output nodes, RNNs can have the following three commonly used output modes.

1. Sequence-classifier: the model is suitable for machine learning problems with sequential inputs and a single output like text categorization (sentiment classification). Given learning data and classification labels $X = \{X_1, X_2, ..., X_T\}, y \in \{1, ..., C\}$, the output nodes of the loop units in a sequence-classifier pass directly through the classifier, a common choice is to use the output node of the last time step $\hat{y} = g\left(o^{\tau}\right)$, or $\hat{y} = g\left[o\left(\overline{h}\right)\right]$, the mean of all system states in a recursive computation[11]. Also, common sequence-classifiers use fully connected structures.

2. Sequence-sequence output: Each time step of the sequence corresponds to an output where the inputs and outputs are of the same length[11]. Given a learning objective $y = \{y_1, ..., y_\tau\}$, the output mode of sequence-sequence outputs results $y^{(t)} = g\left(o^{(t)}\right)$ at each time step.

3. Encoder-decoder: When both the input data and the learning target are sequences of variable length, they can be modeled using two coupled contextually connected RNNs like encoder-decoders. Take the example of machine translation, given the embedded original text and the translated text $X = \{X_1, X_2, ..., X_{\tau 1}\}, y = \{y_1, ..., y_{\tau 2}\}, \tau 1 \neq \tau 2$, the encoder processes the original text at work and outputs $o^{(\tau 1)}$ or $h^{(\tau 1)}$ to the decoder, which generates a new sequence based on the encoder's output[10].

2.2 Gated Recurrent Unit

GRU is a commonly used macro architecture for RNN who is an improved version of LSTM structure as it reduces network weight parameter by omitting the output gate layer while still has similar performance with LSTM network. Though simpler structure, the GRU can also solve the issue of not being able to perform long-term memory and gradient disappearance in back propagation of RNN, which showing promising potential in tasks of polyphonic music modeling, speech signal modeling and NLP as LSTM does[14], [15].

The GRU structure itself has two variations: the full version gated unit with gating done using previous hidden state and three combination of network bias, and a simplified structure minimal gated unit which has even less gate[16].

2.2.1 Fully Gated Unit

The state function of fully gated unit is described by equation 2.6 to 2.9 with initial condition $t = 0, h_0 = 0$:

$$z_t = \sigma\left(W_z x_t + U_z h_{t-1} + b_z\right) \tag{2.6}$$

$$r_t = \sigma\left(W_r x_t + U_r h_{t-1} + b_r\right) \tag{2.7}$$

$$\hat{h}_t = \phi\left(W_h x_t + U_h\left(r_t \odot h_{t-1}\right) + b_h\right) \tag{2.8}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{2.9}$$

where variables:

- x_t : input vector at time t
- h_t : output vector at time t
- $\hat{h}_t$: candidate activation vector at time t
- z_t : update date vector at time t
- r_t : reset date vector at time t
- W, U and b : parameter matrices and vector

where activation functions:

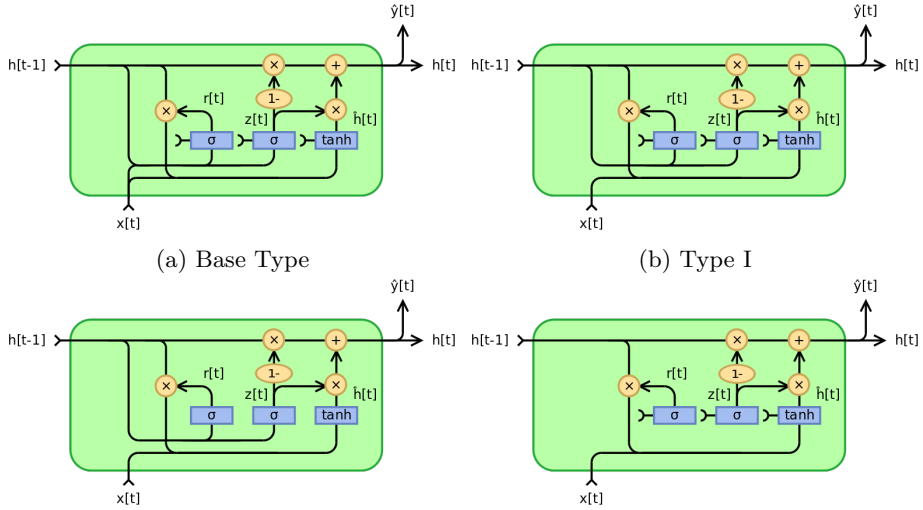
- σ : Originally to be logistic function, but in practice usually choose Sigmoid function $\frac{1}{1+e^{-x}}$, where $L = 1, k = 1, x_0 = 0$ in logistic function.
- ϕ : Originally to be hyperbolic tangent function $tanh$
- $\odot$: the Hadamard product

Equation2.7 describes the reset gate behaviour of GRU, determining how the GRU would combine the newly input vector with previous memory which would help to capture the short-term dependency in time sequence. As r_t becoming bigger, $U_h\,(r_t \odot h_{t-1})$ term of equation 2.8 which is the next output vector would also increase, meaning a tighter combination of input vector x_t and previous memory. In special cases, if this term equals to 0, it means a discard of all previous history memory, and reversely means preserving all the previous hidden states.

The equation 2.6 describes the update gate behaviour of GRU, determining how the GRU would take previous states into current states, which would help to control the degree of transmission of fast information to future. The z getting bigger means the term $(1 - z_t) \odot h_{t-1}$ in equation 2.9 getting small and term $z_t \odot \hat{h}_t$ getting bigger, indicating more current information would be remembered and previous hidden states would be deprecated.

And the equation 2.8 and 2.9 describes how the output vector is formed and how it is updated in every time step.

The figure 2.1 shows the structure of fully gated unit and figure 2.1a is described by equation 2.6 to 2.9. The type I as in figure 2.1b omits the Wx term in equation 2.6 and 2.7; type II as in figure 2.1c omits the Wx and b term; type III as in figure 2.1d only have the b term. And the equation 2.8 and 2.9 remains the same for derivatives.



(a) Base Type (b) Type I

(c) Type II (d) Type III

Figure 2.1: Base type and three variation of fully gated unit[17]

2.2.2 Minimal Gated Unit

The minimal gated unit is a derivative of fully gated unit, who merges update and reset gate into one forget gate of term f_t as shown in equation[18]:

$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \tag{2.10}$$

$$\hat{h}_t = \phi\left(W_h x_t + U_h\left(f_t \odot h_{t-1}\right) + b_h\right) \tag{2.11}$$

$$h_t = (1 - f_t) \odot h_{t-1} + f_t \odot \hat{h}_t \tag{2.12}$$

Which further decreases the network size and parameter dimension space.

2.2.3 Light Gated Recurrent Unit

The Light Gated Recurrent Unit (LiGRU) structure discards the reset gate of fully gated unit, and alter the ϕ activation function from hyperbolic *tanh* to Rectified Linear Unit (ReLU) while also added the batch normalization (BN)[14]. Equation 2.10 to 2.12 describes its behaviour.

$$z_t = \sigma\left(\mathrm{BN}\left(W_f x_t\right) + U_z h_{t-1}\right) \tag{2.13}$$

$$\hat{h}_t = \mathrm{ReLU}\left(\mathrm{BN}\left(W_h x_t\right) + U_h h_{t-1}\right) \tag{2.14}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \hat{h}_t \tag{2.15}$$

2.3 AI Processor and Accelerator

2.3.1 AI Processor

AI processor is a specialized microprocessor designed with the goal of efficiently executing the calculations necessary for a range of artificial intelligence tasks. These tasks commonly encompass intricate mathematical operations, pattern recognition, and data analysis that form the core of machine learning, deep learning, and similar AI algorithms.

Differing from general-purpose processors like Central Processing Unit (CPU)s and Graphics Processing Unit (GPU), AI processors are uniquely tailored for the specific attributes of AI workloads. This optimization enables AI processors to manage substantial data volumes and execute computations in a parallel manner, a critical requirement for the effective training and implementation of neural networks and other AI models.

2.3.2 AI Accelerator

AI accelerator is a more dedicated AI processor who refers to a specialized hardware component explicitly designed and optimized for accelerating AI tasks[19]. Unlike AI processors which can handle a relatively wide range of tasks, dedicated AI accelerators focus on efficiently performing one particular AI network and its derivatives.

Dedicated AI accelerators are designed to provide a balance between high performance and energy efficiency, catering to the intense computational demands of AI tasks while minimizing power consumption. These accelerators are often designed with features that are well-suited to the characteristics of neural networks, which are common in AI applications.

2.3.3 Key Features and Inner Optimization

In order to achieve dedicated AI framework optimization, these systems are commonly using following four methodologies:

- Matrix operations: Many AI computations involve matrix multiplications and vector operations. Dedicated accelerators or processor are optimized to efficiently handle these operations, which are prevalent in tasks like training and inference in neural networks.

- Parallel processing: In order to achieve high performance, AI accelerators are designed to execute multiple operations in parallel, taking advantage of the inherent parallelism present in neural network computations. This significantly speeds up the overall processing time of network inference.
- Reduced precision arithmetic: Many AI workloads can tolerate lower precision, like 16-bit or even 8-bit arithmetic, without sacrificing much accuracy. Taking advantage of this, dedicated AI accelerators often support reduced precision arithmetic, which helps improve performance and reduce memory and bandwidth requirements.
- On-chip optimizations: These accelerators or processors often include specialized on-chip memory, caches, and data storage structures to minimize data transfer bottlenecks and improve overall performance.

2.3.4 Common Implementation Type

To fully utilize the optimization methods and implemented a practical system in real world, four possible solutions are presented:

- GPU: While initially developed for rendering graphics in video games, GPUs have become a popular choice for AI tasks due to their parallel processing capabilities. They can handle a large number of calculations simultaneously, making them suitable for training and running neural networks.
- FPGA: FPGAs are re-configurable hardware devices that can be programmed to perform specific tasks. They offer high energy efficiency and can be customized for particular AI workloads, making them versatile for various applications. The inner parallelism especially making it suitable for high performance computing in AI task.
- ASIC: ASICs are custom-designed chips optimized for specific tasks. They offer the highest performance and energy efficiency but lack the flexibility of FPGAs. Companies design ASICs specifically for AI tasks to achieve the best possible performance.
- Neuromorphic chips: These chips are inspired by the structure and function of the human brain, using interconnected neurons to perform computations. They aim to replicate the brain's efficiency in processing sensory information and performing cognitive tasks.

2

3 Literature Review

Contents

3.1 Drift Prediction and DNA Detection with ISFETs

Recently, LoC method[2], [3] has shown great potential in minimizing the size of biomedical instrumentation platform while remaining or even increasing the detection accuracy and speed at the same time. Meantime, the possibility to fabricate ISFET sensors, which could be used in electro-chemical sensing application like detection of DNA application[4], with unmodified CMOS technology on LoC chip made this especially promising for its cost-effectiveness, convenience and real-time processing speed[5].

As the sensitivity of measurement system would increase as the amount of targeting objects also increases, the diagnostic instruments would be benefited from increasing concentration of target pathogenic DNA strands when applied to medical field, and this process would be manually accelerated by launching amplification reactions of DNA [20]. Then this feature would be suitable for applying ISFET sensors to detect because of its intrinsic high sensitivity of hydrogen ions that would dissociate into the environment during the amplification process and nurtures the charges

on the sensor surface (process shown as in the figure 3.1) and thus cause the electrical property of sensor changes which could be measured with peripheral circuit[21].

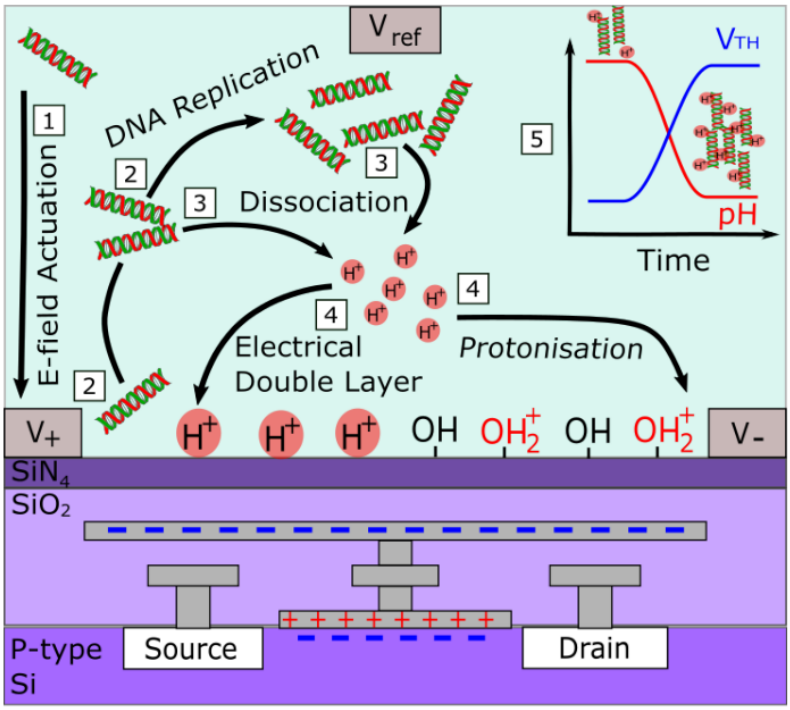


Figure 3.1: Detection of DNA by ISFET sensor with ion-nurtured charges[20]

Then this makes the detection and processing of DNA amplification process related data possible since the bio-signal can be made to be converted to the electrical signal[22]. And some prior works has established the system to sampling and processing the DNA related data with ISFETs. One of the structure by Nicolas Moser et al is shown as in figure 3.2.

The basic ISFET pixel inside the sensor array would firstly sense the ion-changes towards the location. Then it would send the event to the column and row logic to decode the changes. The column logic would generate the PWM wave according to the sensor's data, and calibrate the Digital core after decoding into a 16 bits wide signal. The Digital Core would then adjust the controlling parameter and send the coordinate to the Row and Column Logic to stimulate the array and control the Column Logic in a close-loop way. The system has the SPI unit as interface to be communicated with outside computers.

Although this detecting method performs well in most of the situation, the drift property, which would limit the dynamic range and hence instrument accuracy of the sequencing system, of ISFET sensors caused by the hydration of sensing surface would introduce intense error to this sort of detecting system[6].

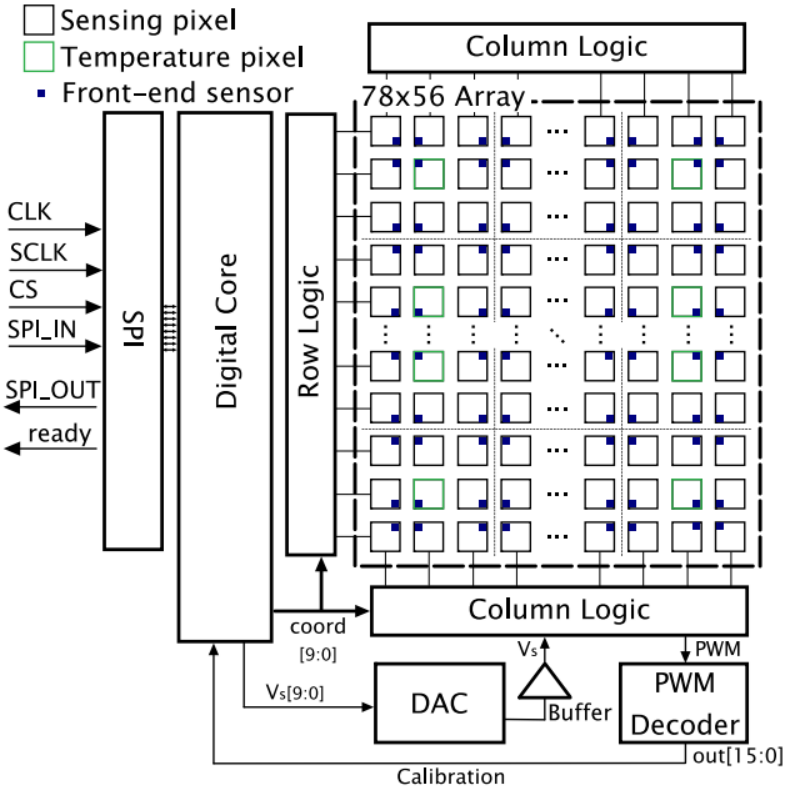


Figure 3.2: DNA detecting and processing system with ISFET sensors[22]

Some prior works has shown the possibility to utilize AI technology such as RNN or CNN network to detect and cancel this drift. The work of [7], [8] and [9] uses RNN, MLP and ANN respectively to compensate the drift and all achieves a RMSE error less than 1e-5 in pH detection. However, for low-frequency electro-chemical measurement, the ionic interactions could only be vaguely spotted[1]. The work of [1] uses RNN network with GRU to overcome this and finally reaches 89.71% accuracy in reaction classification and their processing structure is shown as in figure 3.3.

The ISFET pixel output would be send to the buffer after smoothing, and the buffer would send the timing sequence to the GRU Model to predict the chemical reaction label. If the label is greater than a threshold value, the chemical reaction is predicted to have happened in this pixel, and the estimator would predict the drift according to the delayed data in the buffer. Then the result would be subtracted with smoothed data to have the compensated chemical reaction signal.

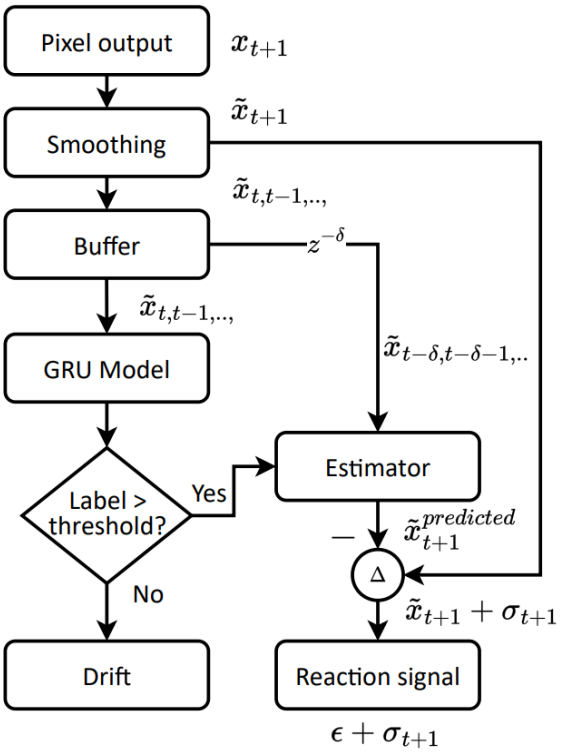


Figure 3.3: Drift compensation network structure in the work of [1]

3.2 RNN Implementation on Hardware

The hardware implementation of RNN network two main methods where the analog one uses capacitor in analog circuit to stimulate and memorize the information and the digital method would uses register to memorize and peripheral unit to approximate the network in a numeric way.

3.2.1 Analog Approach

In term of analog method, Kamilya Smagulova et al proposed a way to implement the recurrent neutral network with memristor-based memory circuit[23].

To solve the operation of matrix vector multiplication (Hadamard product multiplication), [23] proposed the circuit shown as in figure 3.4 to compute product in a parallel way. The cross bar of column and row in the left side of the unit would result to number of intersection point, and a memristor is applied in the each point to connect the crossed line. This makes the output row be the multiplication of conductance of memristor and the input column vector[24], implementing the vector dot operation if the conductance is programmed to be the same as weight matrix in differential mode.

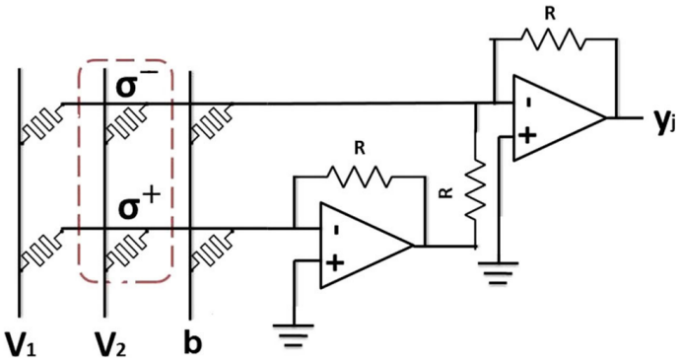


Figure 3.4: Memristor neuron unit

In term of activation function, the system proposed to use activation circuit as shown in figure 3.5 to generate the *Sigmoid* and *Tanh* signals by using the structure of memristor-invertor.

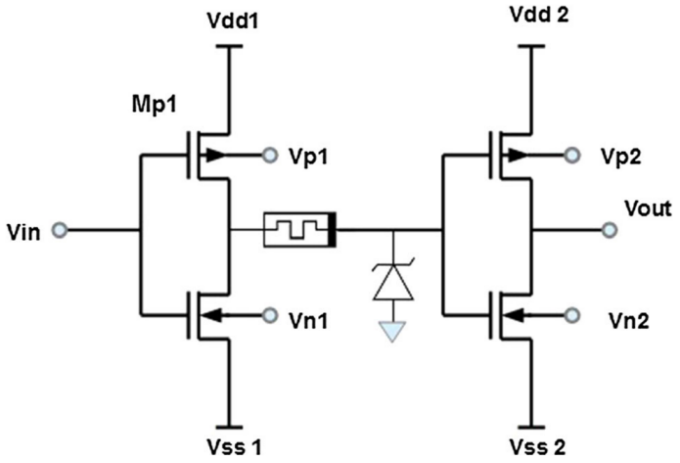


Figure 3.5: Activation circuit

This method could be operated in a extremely high speed since the whole circuit is made up with analog components where there is no clock to constrain the computation speed. Also the power would be low compared to digital approach since the circuit do not possess dynamic consumption when doing analog operation. However, as the value of memristor would need to be set to the value of weight matrix, the accuracy and flexibility would be limited. Any re-deployment of the network would results to the re-calculation and re-fabrication of the circuit, introducing huge amount of cost.

3.2.2 Digital Approach

Considering the intrinsic drawbacks of analog method, the digital approach is considered to be used. Chang et al[25] proposed an configurable system to compute LSTM network based on

FPGA platform. The overall architecture is shown as in figure 3.6.

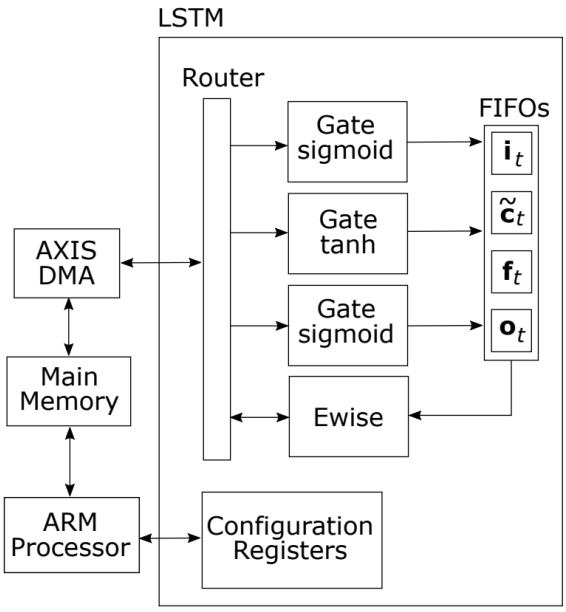
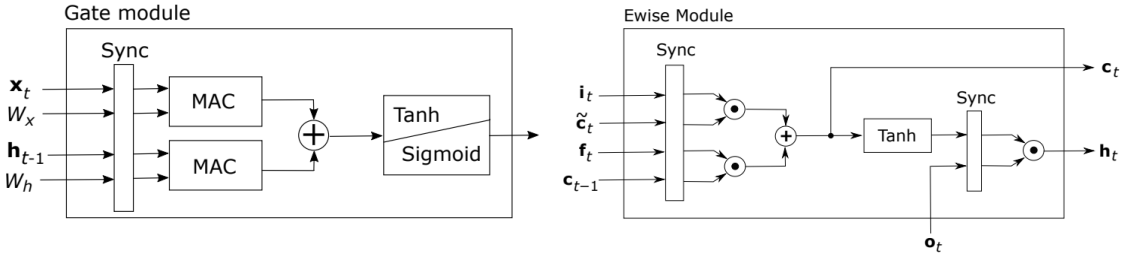


Figure 3.6: Overall architecture for the LSTM system[25]

The whole system is divided into two part: the PL side is for the hardware computation acceleration; and the PS side stands for the overall control and data transition. Inside the PL side, the transferred input would be sent to the router which allocates the data into sub-computation modules. The structure of basic arithmetic units are shown as in figure 3.7. The gate module shown in the figure 3.7a could be configured to execute *Tanh* or *Sigmoid* operation; and it would use the Multiply and Accumulate (MAC) unit to compute the linear-ed result of input matrix and memory matrix. After sending the result to synchronize FIFOs, the Ewise module shown in the 3.7b would compute the update output and memory matrix using parallel multiplication and addition.

The system has been tested to have a desiring performance of a roughly 23 times increase on speed comparing to run the LSTM model with the ARM A9 CPU. As the the basic model



(a) Gate module (b) Ewise module

Figure 3.7: Structure of basic unit in implemented LSTM system[25]

of LSTM network is similar with GRU of this paper, the hardware implementation methodology could be considered as reference when conducting the architecture design. However, since the system has chosen to quantize all the computation unit into 8 bit fixed point number, its accuracy and flexibility to migrate to other model would be limited.

Besides, Zaghloul et al[26],has implemented the GRU unit on FPGA platform and resulting to a 24.7 times speed-up compared to ARM A9 CPU. Their basic computation structure are the same with Chang et al which uses two channel MAC unit to compute the linear result and uses the same structure of Ewise to update the memory matrix. Also, Sicheng et al proposed a way to expand the RNN structure into stages where each represent a time-step and compute them in parallel[27], which is also insightful.

3

4 RNN Implementation with Vivado IP

Contents

This chapter implemented the proposed RNN system with Vivado IP. Firstly the RNN model has been analyzed and constructed in C++. Then the C++ model has been projected into hardware to have an overall framework. After applying fine-grained pipelining and other parallelism methodology, arithmetic IPs that constitute the system has been constructed. Finally the IP has been scheduled both in IP and system level to optimize the overall latency and Iteration Interval (II). The testbench is also established to test the system.

4.1 RNN Projection and Hardware Architecture

4.1.1 C++ Projection

As the network of [1] is a simple combination of RNN network with two-layer GRU structure and MLP with full connection layer, the architecture could be projected to two cascaded parts which perform GRU and MLP function respectively.

4

GRU Unit Construction

Based on equation 2.6 to 2.9 which describing the behaviour of fully gated unit in the work of [1], the C++ model of first GRU unit in RNN network that could detect the chemical reaction is established as follow:

```
void gru_layer1(float input, float h_tm1[32], float h_tm1_result[32])
    {
    int i,j;
    float x_z [32];
    float x_r [32];
    float x_h [32];
    float re_z [32];
    float re_r [32];
    float re_h [32];

    // warr computation
    for(i=0;i<32;i++)
        x_z[i] = input*reaction_first_gru_warr[i]+reaction_first_gru_ibis[i];
    for(i=32;i<64;i++)
        x_r[i-32] = input*reaction_first_gru_warr[i]+reaction_first_gru_ibis[i];
    for(i=64;i<96;i++)
        x_h[i-64] = input*reaction_first_gru_warr[i]+reaction_first_gru_ibis[i];

    // uarr computation
    for(i=0;i<32;i++)
    {
        re_z[i] = 0;
        for(j=0;j<32;j++)
            re_z[i] += h_tm1[j]*reaction_first_gru_uarr[i*32+j];
        re_z[i] += reaction_first_gru_rbis[i];
    }
    for(i=32;i<64;i++)
    {
        re_r[i-32] = 0;
        for(j=0;j<32;j++)
            re_r[i-32] += h_tm1[j]*reaction_first_gru_uarr[i*32+j];
        re_r[i-32] += reaction_first_gru_rbis[i];
```

```
33      }
34      for(i=64;i<96;i++)
35      {
36          re_h[i-64] = 0;
37          for(j=0;j<32;j++)
38              re_h[i-64] += h_tm1[j]*reaction_first_gru_uarr[i*32+j];
39          re_h[i-64] += reaction_first_gru_rbis[i];
40      }
41
42      // Activation function computation
43      float z [32];
44      float r [32];
45      float hh [32];
46      for(i=0;i<32;i++)
47          z[i] = 1/(1+expf(-1.0*(x_z[i]+re_z[i])));
48      for(i=0;i<32;i++)
49          r[i] = 1/(1+expf(-1*(x_r[i]+re_r[i])));
50      for(i=0;i<32;i++)
51          hh[i] = tanhf((re_h[i]*r[i]+x_h[i]));
52
53      // Output updating
54      float output1[32];
55      float output2[32];
56      for(i=0;i<32;i++)
57      {
58          h_tm1_result[i] = z[i]*h_tm1[i] + (1-z[i])*hh[i];
59          output1[i] = z[i]*h_tm1[i];
60          output2[i] = (1-z[i])*hh[i];
61      }
62  }
```

As both the reset gate and update gate have term of multiplying input vector x or h_{t-1} with corresponding weight matrix W and U, they have been scheduled to be separately computed for an easier hardware projection, and all variable with subscript z representing update gate related computation, r for reset gate and h for output vector. As the h and $\hat{h}$ is consist of 32 number, all results within GRU unit is also of 32-value vector.

The line 12 to 17 implements the W related computation. After expanding the matrix dot operation, which multiplies the weight and adds the bias, the three result, x_z, x_r and x_h could be used as term of $W_z x_t$, $W_r x_t$, $W_h x_t$ in the equation. Similarly, line 20 to 40 is of the computation of U related computation, and they have the same subscript naming methodology as W have.

Then line 46 to 51 conducts the activation behaviour of σ and ϕ in the equation. By choosing function of σ and ϕ as *Sigmoid* and *Tanh* respectively, output vector h and $\hat{h}$ can be computed as shown in line 58 to 60 with expanded Hardamad product.

And for second GRU unit, it is mostly the same with first layer, excepting it would process W related matrix dot with similar methodology of U and dense layer, which would sum the weighted input to have one value of output instead of directly output weighted result as W term in the first layer has. Besides, the output of first layer would be connected to the input of second layer, and they would be interleaved and looped for specified iteration (196 times for this project).

4

MLP Construction

Then the MLP structure, which have been framed as three dense layers with ReLU output in [1] could be implemented with following:

```
void dense_l1(float input[32], float result[32])
{
    int i,j;
    for(i=0;i<32;i++)
    {
        result[i] = 0;
        for(j=0;j<32;j++)
            result[i] += input[j]*reaction_dense_1_weights[j+i*32];
        result[i] += reaction_dense_1_bias[i];
        //relu output
        if(result[i]>=0)
            result[i] = result[i];
        else
            result[i] = 0;
    }
}

void dense_l2(float input[32], float result[16])
{
    int i,j;
    for(i=0;i<16;i++)
    {
        result[i] = 0;
        for(j=0;j<32;j++)
            result[i] += input[j]*reaction_dense_2_weights[j+i*32];
        result[i] += reaction_dense_2_bias[i];
        //relu output
        if(result[i]>=0)
            result[i] = result[i];
        else
            result[i] = 0;
    }
}

void dense_l3(float input[16], float result[1])
{
    int j;
    result[0] = 0;
```

```
39      for(j=0;j<16;j++)
40          result[0] += input[j]*reaction_dense_3_weights[j];
41      result[0] += reaction_dense_3_bias[0];
42  }
```

First dense layer *dense_l*1 takes the sum of the product of corresponding weight matrix and input vector (output of second GRU layer after 196 times of iteration) and adds it with bias and operates ReLU operation to have one single value of output. By iterating 32 times, the layer dense the 32-value vector input to a 32-value vector output.

The second layer *dense_l*2 is similar to the first layer, while takes different weight and bias matrix and iterates 16 times which dense 32-value vector to a 16-value vector. The third layer *dense_l*3 would dense 16-value vector input to a 1-value output, while not performing ReLU operation.

Finally after performing the *Sigmoid* operation of output of third dense layer, the output reaction percentage could be computed as final result.

4.1.2 Hardware Projection

The most rudimentary projection of hardware is to project every sentences in C++ code to one or more particular hardware units. With this methodology, as loop like "$for(i = 0; i < 16; i++)$" could not be directly synthesized and implemented in hardware level, all these loops need to be projected as unit that runs particular iteration and conducts the operation inside the loop body every iteration.

The accumulation operation "+ =" would be projected as hardware accumulator which add right side expression every clock cycle and gets the final results. And the weight and bias array access would be projected as memory read behaviour, which is usual done with BRAM unit.

In consideration of accelerating the design, these hardware unit could be optimized to have a better performance: as the original accumulator takes N-1 cycles to sum N values (perform add operation and have the results every cycle), it could be accelerated to an add-tree structure to reduce the overall cycle to $log_2(N)$ cycles at an expense of extra hardware.

And the iteration of loop could be considered to be unrolled after decoupling the operation in loop body to have an acceleration factor of N (N is the unroll factor). As loop being expanded, the

array access brings a higher demands of higher memory bandwidth. And this could be solved by partition the weight and bias matrix and instantiate multiple BRAMs to offer a higher data read ability.

4

4.1.3 Hardware Architecture

After the construction and analysis of C++ projection of RNN network, the hardware architecture could be built as shown in figure 4.1 with the goal of accelerating the design.

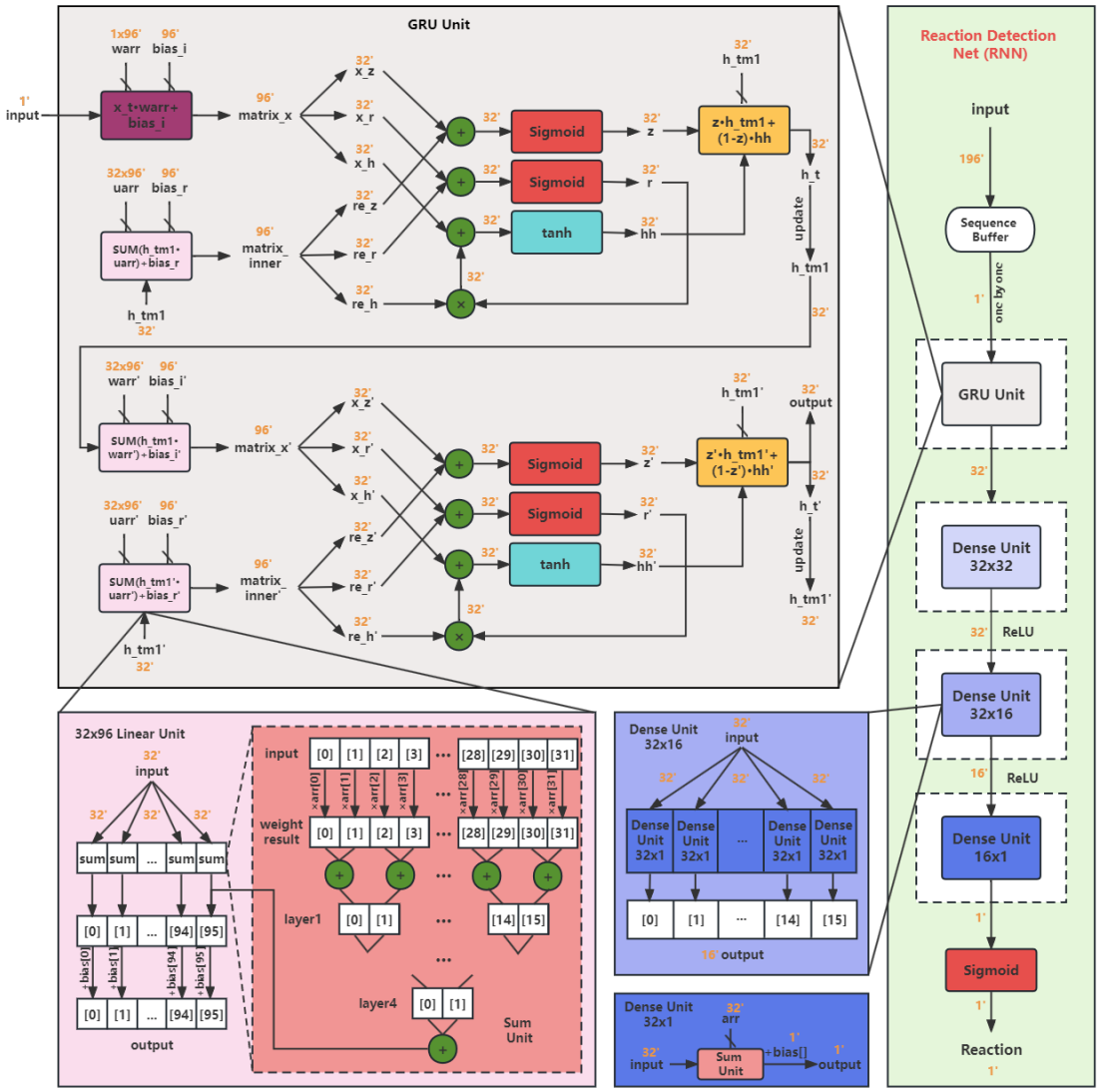


Figure 4.1: Hardware architecture of RNN nerwork for reaction prediction

Overall Architecture

The right **Light Green** block diagram shows the overall hardware data-path of RNN network. Firstly the input vector, which consist of 196 IEEE-754 single float point number, would be sent to the sequence buffer.

Then the buffer would take out one single value sequentially when previous value is computed. Buffer's output would be connected to the **Gray** block (GRU unit), and the GRU's result will be condensed with MLP which is consisted of three cascaded Dense Unit.

Finally after *Sigmoid* operation of 1-sized dense result, one single buffered value in Sequence Buffer finished its computation. And after all the 196 of buffered data has been computed with these process, the system done signal would be pulled to high for one cycle, indicating all the computation is done. Then the RNN's final output is the reaction percentage of 196 iteration.

GRU Unit

The **Gray** block shows the GRU unit, which consist of two interleaved GRU layers in C++ model. Once the unit received the input value from buffer, it would firstly start the first GRU layer with buffed value as input, and then starts the second layer when the first layer is done and set the output of first layer as the input of the second layer. And the output of second layer would be sent to the next dense unit. As to realize the memory behaviour of GRU unit, the h and $\hat{h}$ value, which is h_tm1 and h_tm1' value of first and second layer in graph would be preserved from previous input vector computation and update it whenever layer computation is done instead of reset it to zero every time.

Inside the GRU unit, the first layer takes the 1-size vector as input, and **Violet block (1x96 Linear Unit)** would weight and bias it with 1x96 and 1x96 size matrix respectively to perform W related computation. Also, the **Pink block (32x96 Linear Unit)** would firstly weight the preserved 32-value h_tm1 with 32x96 matrix and sum it with **Deep Pink block (Sum Unit)**, then add the bias which is of 1x96 matrix to have one of the out vector of unit to perform U related computation. And this weight-sum-add (32x1 Linear) process need to be executed for 96 times to have all the output matrix (x_z, re_z, etc).

Inside **32x96 Linear Unit**, the 32-sized input vector need to be processed for 96 iterations of 32x1 linear operation to have all the output matrix. And the core of 32x1 linear operation is the

Sum Unit, which firstly weight the 32-sized input value with shifted 32x96 matrix (has 96 groups of 32-sized weight elements, and weight input value with different 32-sized element determined by iteration number), and then use the 5-layer add-tree to sum the weighted results to have the fasted processing speed. Finally, the output of **Sum Unit** would be added with particular bias in a 1x96 matrix.

All the results of after **1x96 Linear Unit** and **32x96 Linear Unit** would be grouped separately. The x_z and re_z will be sent to the **Green Block(Float-add Unit)**, and so does x_r and re_r. The re_h and r would be sent to **Float-multiply Unit** and then add with x_h in **Float-add Unit**.

Then the two **Red block (*Sigmoid* Unit)** and one **Blue block (*Tanh* Unit)** would perform the activation operation which implement the σ and ϕ function in equation 2.6 to 2.9 to have the 32-sized update gate variable z, reset gate variable r and candidate output vector $hh\left(\hat{h}\right)$ respectively.

The r would be sent back to **Float-multiply unit** as mentioned before to generate the input of ***Tanh* Unit**, and z and hh would be connected to the input of **Orange block (Combination Arithmetic Unit)** with previous state h, which is represented as h_tm1 in the figure, to calculate the updated h value h_t. Inside the **Combination Arithmetic Unit**, the z and h_tm1, $1-z$ and hh would perform matrix dot and add to have a 32-size output vector. So in hardware level this module is the combination of existing Float-multiply and Float-add unit and it will be assembled into these IPs in the succeeding section.

Finally the h_t would update the preserved state vector h_tm1 to current state, and send it out as the layer output.

The structure of second layer GRU is similar with the first one. However, as the second layer takes a 32-sized vector as input instead of 1, the **1x96 Linear Unit**, which conducts W related computation, has been replaced by a **32x96 Linear Unit**. Besides, it has its personal preserved h_tm1', updating every iteration.

MLP

As the MLP has been modeled as three cascaded full connection layers, its hardware structure could be easily projected as three seperate and cascaded module (shown as three Purple blocks after **GRU Unit**).

The Light Purple block (Dense Unit 32x32) would dense a 32-sized input to a 32-sized output. Similar as 32x96 Linear Unit, it would dense the input vector with **Deep Purple block (Dense Unit 32x1)**, which conducts weight-sum-add (linear) operation to have 1-sized result, in 32 iteration with iteration dependent weight and bias matrix to have all the 32 results. Finally, the module would perform ReLU operation to all the result element separately to have the module output.

And the **Purple block (Dense Unit 32x16)** is the next unit cascaded behind Dense Unit 32x32. It dense the 32-sized input vector to a 16-sized vector, down-scaling the vector size by only taking 16, instead of 32 in Dense Unit 32x32, iteration of linear operation with **Dense Unit 32x1**, and output ReLU of computed results.

The third **Deep Purple block (Dense Unit 16x1)** dense the 16-sized input to only 1 sized output. And in term of hardware efficiency, this unit is the simplified version of **Dense Unit 32x1** by set the last 16 value of its input vector to zero.

Finally, the 1-sized result of **Dense Unit 16x1** would not conduct ReLU operation but directly send to one *Sigmoid* **Unit** to have the final network output.

4.2 Arithmetic IP Configuration

4.2.1 System Data Path and Optimization

As the analysis in previous section, as the critical path is started from 32x96 linear unit, ideally the system performance would be the best if all the iteration within 32x96 linear unit are unrolled and output all the element of 96-sized vector while all the afterward units need to process 32-sized vector at same time and outputs all the result elements together. This brings the requirement to have the fastest 32x1 basic linear unit which constitutes the 32x96 linear unit. And the structure of 32 Float-multiply units, a 5-layer float point add-tree and one final bias Float-add unit as *SumUnit* shown in the figure 4.1 would be the fastest processing micro-architecture.

However, higher processing speed would bring an even higher hardware resource burden: if keep the 32x1 linear unit the fastest architecture, it would need to instantiate 32 of float-multiply IPs and 32 of float-add IPs (31 from 5-layer add-tree, 1 from final bias adder); And to make the processing speed fastest in system level, the all the individual unit need have an II equals to 1 to maximize the throughput. This configuration in Vivado IP takes 3 DSP for multiply unit and 2

for the adder, indicating the fastest architecture of 32x1 linear unit needs to consume 320 DSP units at least.

And this considerable consumption of resource would become even greater if accelerating 32x96 linear unit. If it is fully unrolled to have 96 paralleled processing path, a single unit would then takes 30720 DSP units and 92160 for two GRU layers, surpassing the resource limit of 6840 DSP resource on the most advanced VCU118 FPGA board by far, which is not possible in practice, and the unfeasible high memory bandwidth demands it required would be another problem. Then a smaller unroll factor is to be explored and compared. After pre-estimating the relationship between resource and unroll factor and trade-offs between speed and resource, the final unroll factor is considered to be 2 in the design inspired by the work of [1].

Then the data path of system could be determined: the 32x96 and 1x96 linear module would unroll the loop with factor 2, which output 2 elements of output vector per cycle, and these elements will then be grouped respectively (x_z with re_z, x_r with re_r and etc) and sent to the Float-add unit. However, as x_z, x_r and x_h are theoretically needed to be grouped and added at the same time but previous linear module could only output 2 elements, their processing order would also need to be examined.

The critical path of the system can be clearly analyzed to be the hh path. As z, r and h all need results from previous linear module to be computed, the hh path, which needs to add the r_h and the product of r and r_h and then send to $Tanh$ unit, has a longer path since it not only need the linear results, but also require other results that also uses linear result to be ready. So the r path is arranged to be the first priority, and the linear module would firstly compute the x_r and re_r which is used for r computation, and then compute x_h and re_h to generate the hh value while x_z and re_z which is used for z generation. Finally the Combination Arithmetic Unit would update the h_t vector.

4.2.2 Arithmetic IP Configuration

As analysis in previous section, the interface definition of units within the network is shown as in table 4.1.

The linear 1x96 unit take the buffed 1-sized output from sequence buffer as input, and generates 2 elements of results per cycle, 96 elements in total. And similarly linear 32x96 unit takes the h_tm1, the result of first GRU layer which constains 32 elements as input, and output 2 elements

Table 4.1: Interface Definition of Arithmetic IPs

Unit	Interface	In/Out	Function
Linear_1x96	[31:0] din[i]	In	RNN input
	[31:0] dout [1:0]	Out	Dual output value (2 elements)
Linear_32x1[ii]	[31:0] din [31:0]	In	Unit input (32 elements)
	[31:0] dout	Out	output value
Linear_32x96	[31:0] din [31:0]	In	h_tm1 input (32 elements)
	[31:0] dout [1:0]	Out	Dual output value (2 elements)
Dense_32x32	[31:0] din [31:0]	In	Unit input (GRU layer2 output, 32 elements)
	[31:0] dout [1:0]	Out	Dual output value (2 elements)
Dense_32x16	[31:0] din [31:0]	In	Unit input (Dense 32x32 output, 32 elements)
	[31:0] dout [1:0]	Out	Dual output value (2 elements)
Dense_16x1	[31:0] din [15:0]	In	Unit input (Dense 32x16 output, 16 elements)
	[31:0] dout	Out	Single output value
Arithmetic[iii]	[31:0] din1[iv]	In	First channel input
	[31:0] din2[iv]	In	Second channel input
	[31:0] dout1	Out	First channel output value
	[31:0] dout2	Out	Second channel output value

[i] As all data in system are using IEEE-754 single float point format, every element need 32 bits of width to be represented in hardware.
[ii] The unit is consist of one Sum unit (Deep pink block in figure 4.1) and one bias adder.
[iii] The unit includes dual channel float-add, float-multiply, *Sigmoid* and *Tanh* sub-units.
[iv] As float-add and float-multiply needs two operands each channel, this port would contain 2 elements ([31:0] din [1:0]) for float-add and float-multiply and 1 for *Sigmoid* and *Tanh*.

of result per cycle, 96 elements in total.

The Dense unit 32x32, 32x16 and 16x1 would take the output of previous layer as input (32 elements, 32 elements and 16 elements respectively) and all generates 2 elements of results per cycle, but have 32, 16, 2 (in order to be better aligned with other dual port unit, 1 zero result has been padded) elements in total respectively.

The dual channel arithmetic unit would directly take two groups of input (add and multiply need two port per channel as they need two operands) and processing them in parallel, outputting the results in two separate channels.

4.2.3 Linear 1x96 Unit

Overall Architecture

The architecture of Linear 1x96 Unit is shown as in figure 4.2.

The input of unit is one element vector with 32 bits of width in hardware, and it would be copied for 32 times to form a 32-element array *CopiedArray*. As all the unit within GRU has been targeted as dual-channel module, the original 1x96 unit has been partitioned to iterate 16 times

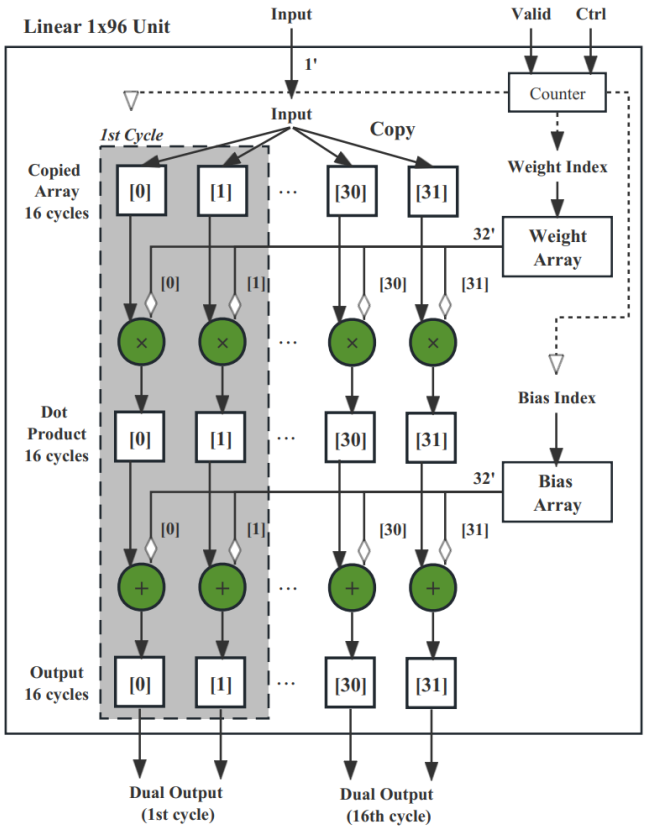


Figure 4.2: Architecture of Linear 1x96 Unit

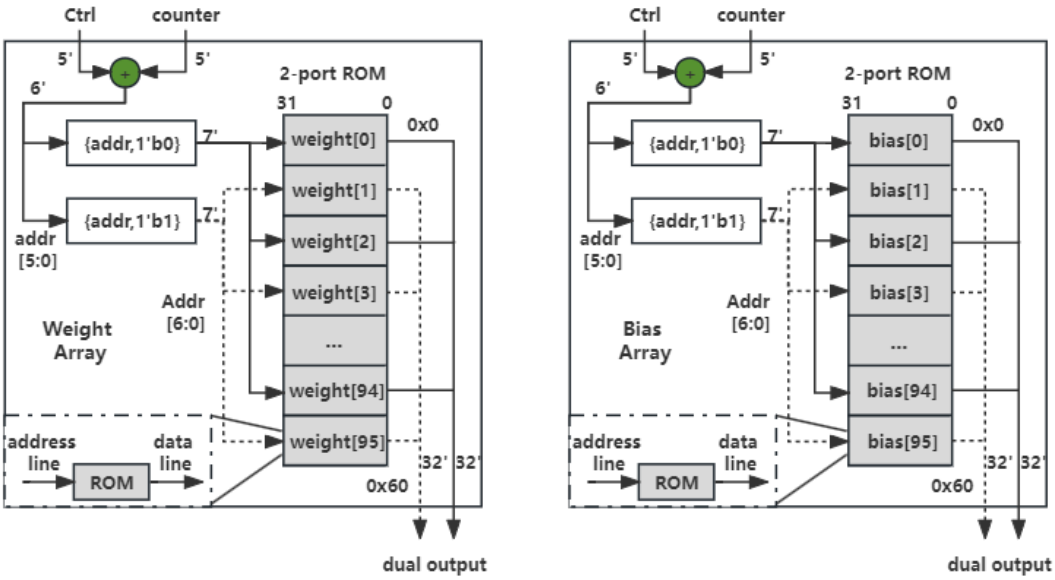
to compute all the 32-element array x_r then x_h and finally x_z.

In hardware level, each cycle the unit would perform the computation within the gray box, multiplying the corresponding 2-element weight and then add the 2-element bias to output the results. Also, an internal counter would count the iteration number from 0 to 15 (16 times), and control the index of weight array and bias array to provide correct weight and bias in each iteration. This counter is controlled by external $Valid$ signal and $Ctrl$ signal: the counter starts to count when $Valid$ is high, and output the weight and bias index according to external $Ctrl$.

Memory Array

In term of weight array and bias array, as the unit only needs two weight value and bias value per working cycle, two 2-port Read Only Memory (ROM) would be capable for this requirement (one for each), and each of the ROM holds 96x32 bits of space. The storing methodology of weight and bias matrix in their array is shown as in figure 4.3.

The bias and weight value are storing in sequence in the ROM memory space. The $Ctrl$ and



(a) Weight Array structure (b) Bias Array structure

Figure 4.3: Structure of Weigh and Bias Array inside 1x96 Linear Unit

counter would be added to have *addr* of 6-bit width inside the module when unit valid signal is high, and then the *addr* is used to form addressing index for odd and even spacing number, which is for output dual-value result, by extending a 1-bit 0 and 1 in the Least Significant Bit (LSB) (equivalent to multiply *addr* by 2 and add 0 and 1 respectively. As counter counts 0 to 15 when module is valid, this makes extended 7-bit *Addr* able to index all the 96 elements according to *Ctrl* in odd-order or even-order).

Hardware Resource

Then the unit's hardware is all module inside the gray block, which would only need to instantiate two float-point adder, two float-point multiplier and two 2-port ROM Vivado IP. In order to achieve full-pipelined structure and thus provide highest throughput, all the arithmetic IP like float-point adder and multiplier have been configured with II=1 and latency=1 to make the whole GRU layer has lower latency as possible, and they would consume 2 and 3 DSP resource respectively. The 2-port ROM would then instantiate 2 BRAM for each ROM, as basic unit of BRAM in Xilinx FPGA is BRAM18k, which hold 1k storing space and each space has a width of 18-bit, but the *Array* needs a space of 32-bit so two BRAM18k would be concatenated to provide a width of 36-bit.

Unit Timing

The proposed timing of Linear 1x96 unit is shown as in figure 4.4.

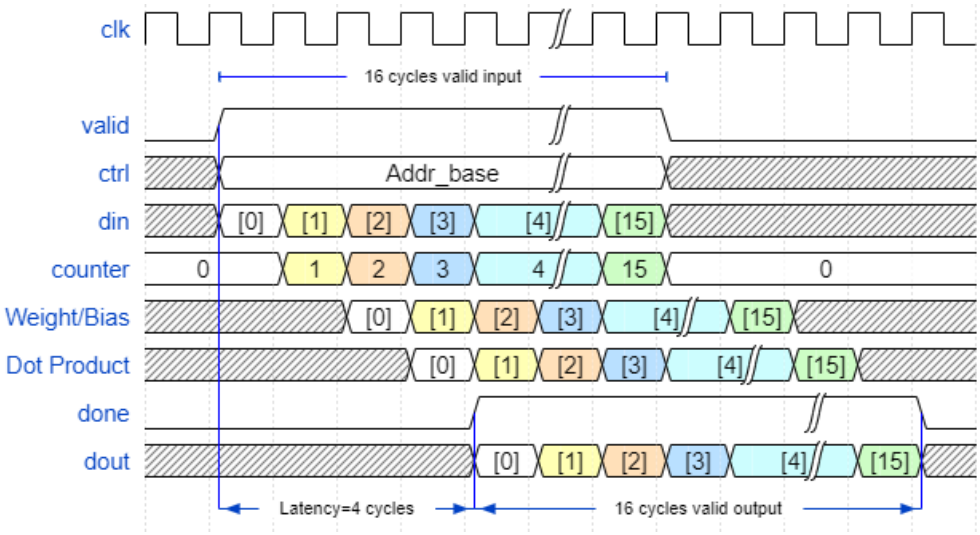


Figure 4.4: Timing graph of Linear 1x96 Unit

The *valid* signal and *ctrl* and *din* are proposed to arrive synchronously to the module, and the *valid* signal needs to be high for 16 cycles to compute 32 results using the unit while the *ctrl* signal indicate the base address for 2-port ROM indexing to the weight and bias matrix (0x0 for computing x_z, 0x20 for x_r, 0x40 for x_h as they are both 32-bit in size so the address increment is 32 which is 0x20 hex-decimal) should keep no-change during the valid session.

In this process, when the counter detects the high valid, it would start to count, and the corresponding address would be concatenated and shifted to have the index for weight and bias. Then as the BRAM has been configured to have read latency=1 cycle, the weight and bias value would be available after one cycle.

The two-cycle buffered *din* data (valid weight data is two cycles behind the valid *din*, so a shift register with depth of 2 has been applied to align the signal) would then start to multiply with the weight in two channels, and the one-cycle buffered bias (valid bias data is being read out the same cycle with weight, so is one cycle above the valid *DotProduct*, so a shift register with depth of 1 has been applied to align the signal) would be added after the computation. Finally the *done* signal would be high for the same high cycle of input *valid*, and output the computed *dout* in dual channel (first for odd index result, and other for even index) with overall latency=4 cycles.And as unit is full-pipelined structure, the unit's II=1.

Simulation

The unit simulation result is shown as in figure 4.5 whose stimulus dual-channel inputs are all 1-sized value of 0.7 and weight and bias are the same parameter of GRU layer1 W related matrix. The unit result has been compared with C++ simulation result and showed a good match while the module latency is tested to be 4 cycles with II=1, indicating the unit is functioning correctly.

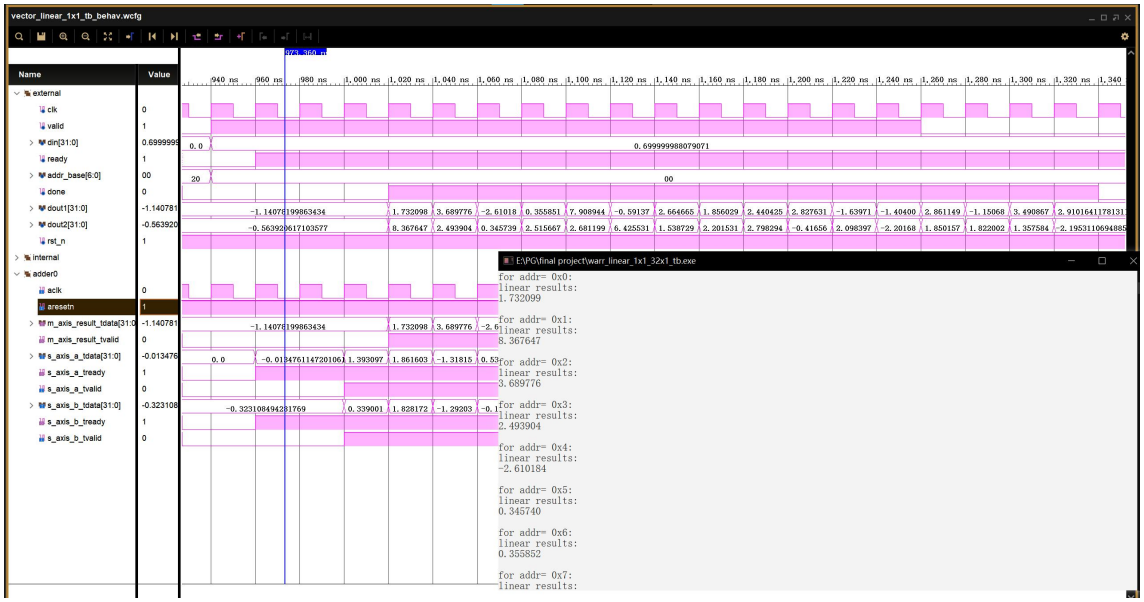


Figure 4.5: Simulation of Linear 1x96 Unit

4.2.4 Linear 32x96 Unit

Overall Architecture

The architecture of Linear 32x96 Unit is shown as in figure 4.6

The input of the unit contains 32 elements array which each element is IEEE-754 single float point number format of 32 bits. Similar with Linear 1x96 Unit, this input array will be copied into two channels each cycle (gray block in the figure), and each channel will be processed with Linear 32x1 unit (white square block inside the gray block) to have one result each channel and two in total. After 16 iteration, all the 32-sized output would be computed, and the order of re_r, re_h and re_z could be configured with $Ctrl$ (0x0, 0x20, 0x40 respectively).

Inside the Linear 32x1 Unit, the 32 elements array will multiply 32 different weight values in parallel with 32 float-point multiplier units, and then in order to sum the 32 elements array in

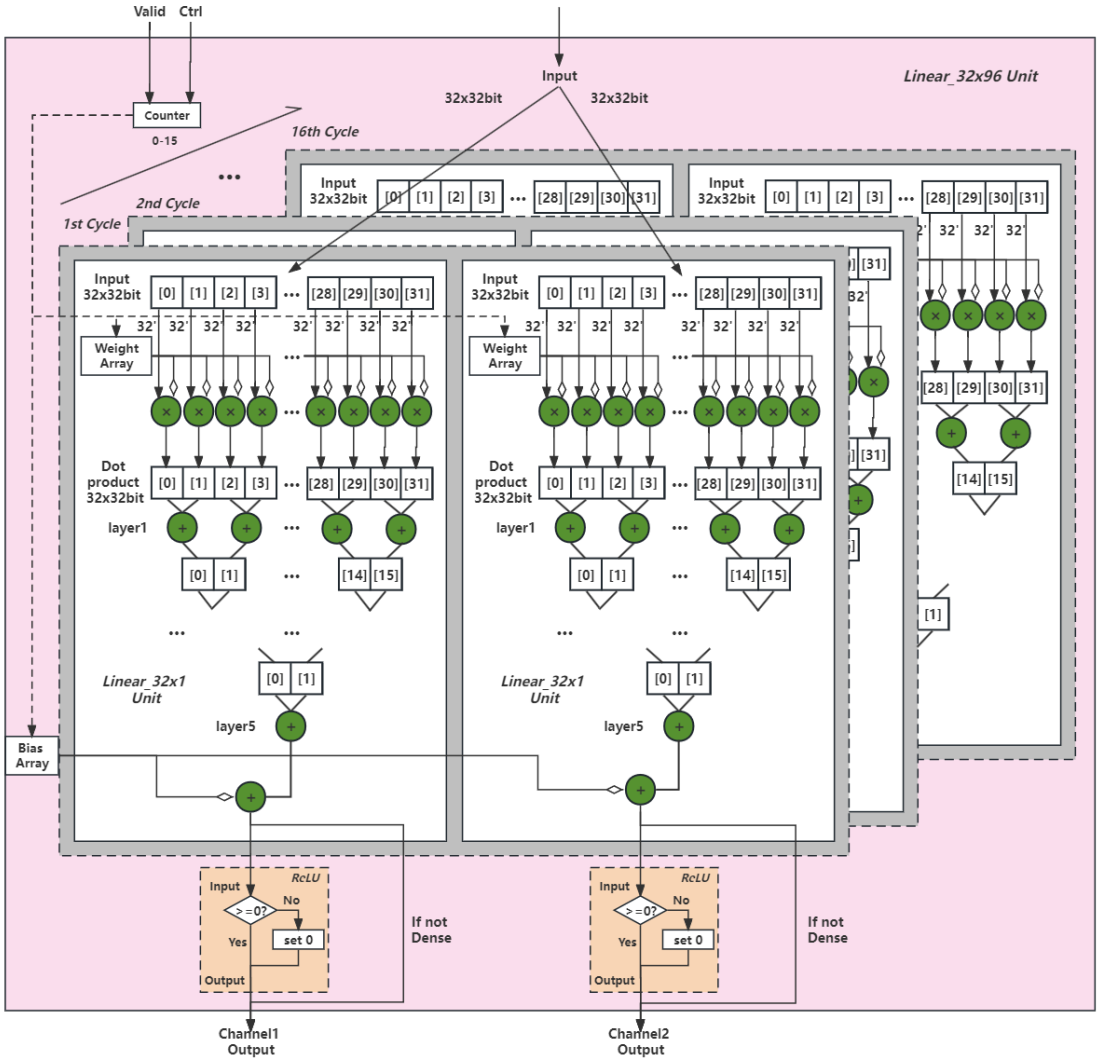


Figure 4.6: Architecture of Linear 32x96 Unit

the fastest way, the array would be sent to a 32-input Add Tree as shown in figure 4.7 instead of an serial float-point accumulator nor parallel adder because of the dependency between the summation and the remaining values.

In the add tree, the first layer uses $\frac{32}{2}$ number of add unit to add all the input with 16 groups of odd and even index separately to have a 16-sized results, and the second layer recursively uses $\frac{16}{2}$ number of add unit to add the 8 groups of input to have a 8-sized results; Until the fifth layer, the two results of fourth would be added to have the final result of the summation of 32 input numbers.

As the add tree structure only has log_2N layers for N-sized input, its overall latency would be $Latency_{adder} * log_2N$ and II=$Latency_{adder}$ if fully pipelined structure has been used. But an

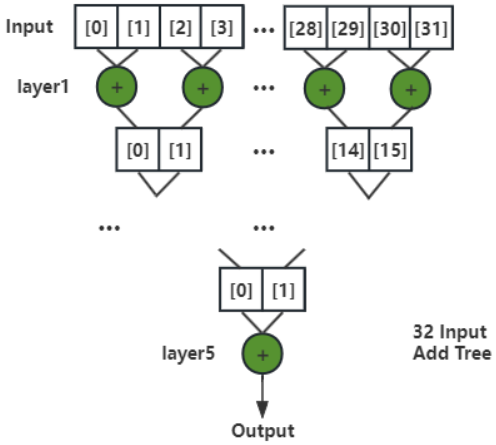


Figure 4.7: Structure of 32 input value Add-tree

serial accumulator would have a overall latency and II equal to $Latency_{adder} * (N-1)$ for N-sized input, which is slower than the add tree structure. However, in term of hardware utilization, the accumulator only need one adder to finish the computation while the add tree structure needs $N-1$ number of adder. In other words, in order to have the highest computation speed, hardware resource has been sacrificed to utilize the parallelism of adder.

Then the summation of 32-sized $DotProduct$ would add with the bias and bypass the ReLU unit, which sets all the negative number to 0 and remains the positive number, to have the output of Linear 32x1 Unit, in linear operation (extended feature for dense operation). Finally the results of two channel Linear 32x1 module constitute the outputs of Linear 32x96 module.

Similarly, an internal counter would count the iteration number from 0 to 15 (16 times), and control the index of weight array and bias array to provide correct weight and bias in each iteration. This counter is controlled by external $Valid$ signal and $Ctrl$ signal: the counter starts to count when $Valid$ is high, and output the weight and bias index according to external $Ctrl$.

Memory Array

In term of matrix array, as the two channel Linear 32x1 Unit only needs two valid bias value per cycle, only one 2-port ROM would be capable for this so the two channel bias array would be combined to one array and using one 2-port ROM for bias storing.

However, in each channel, as 32-sized weight array is needed per cycle, 32 ports, which is 16 of 2-port ROM, is needed to provide enough memory bandwidth. Additionally, as weight array

is different per channel and per iteration, the storing memory cannot be shared across channel, indicating two channel of Linear 32x1 Unit in this case would needs two weight array but not combined one as bias array. The storing methodology of weight matrix in its array is shown as in figure 4.8.

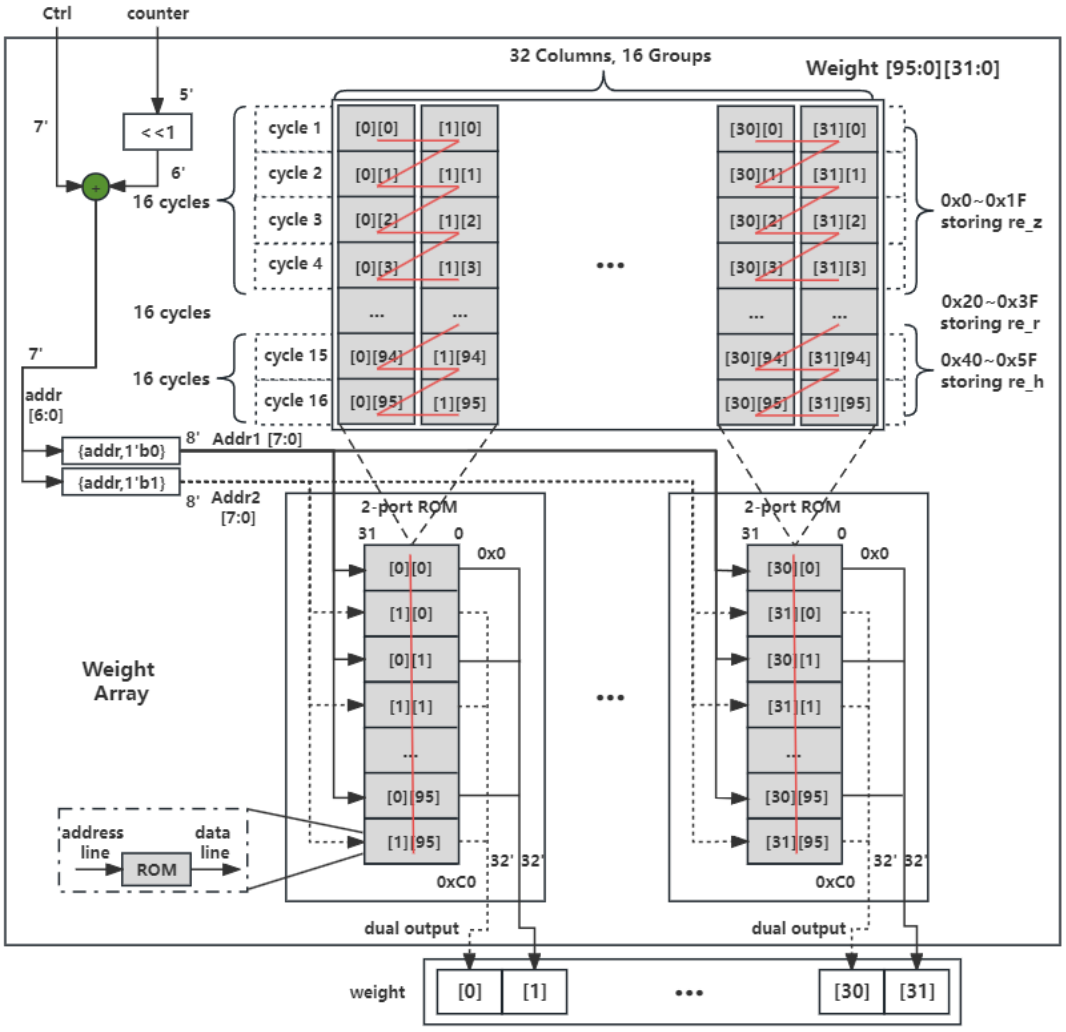


Figure 4.8: Structure of Weight Array Unit inside Linear 32x96 Unit

The weight array of Linear 32x96 Unit is originally to be a 32x96 one-dimension array. To make it processing friendly, the one-dimension array has been firstly resized to two-dimension array with 96 columns and 32 rows whose each row represents the 32-element weight vector that Linear 32x96 Unit needed per cycle, and stores the weights for re_z, re_r, re_h computation in sequence (0x0-0x1F for re_z, 0x20-0x3F for re_r, 0x40-0x5F for re_h).

Then intuitively the two-dimension array would be partitioned in column as each rows are needed per cycle. Considering the micro unit ROM has two ports per unit, if only storing one

column per ROM, half of the port bandwidth would be wasted. For this reason, the odd index column data has been grouped with even index data, and in each cycle they would be read out in parallel while 16 ROMs with this pattern would then provide the needed 32-element data.

The storing method of grouped column is in *ZigZag* way, which storing the first row in sequence and the the second row in sequence, etc. Under this methodology, the odd index address for ROM is for odd index column searching, and even index for even column. And each 2-port ROM would then consume 2*96, which is 192 storing slot which each slot is 32-bit in width.

As external counter counts 0 to 15 for 32-element computation, it would be firstly left shift by one, and then add the *Ctrl* signal to control computing *re_z*, *re_r* or *re_h* (0x0, 0x20, 0x40 respectively) to have the rudimentary 7-bit *addr*. Finally, to index the adjacent odd and even address, the *addr* would be expanded with 0 and 1 in the LSB for odd and even index respectively.

The structure of bias array, which is shown as in figure 4.9, is similar to the one of 1x96 Unit.

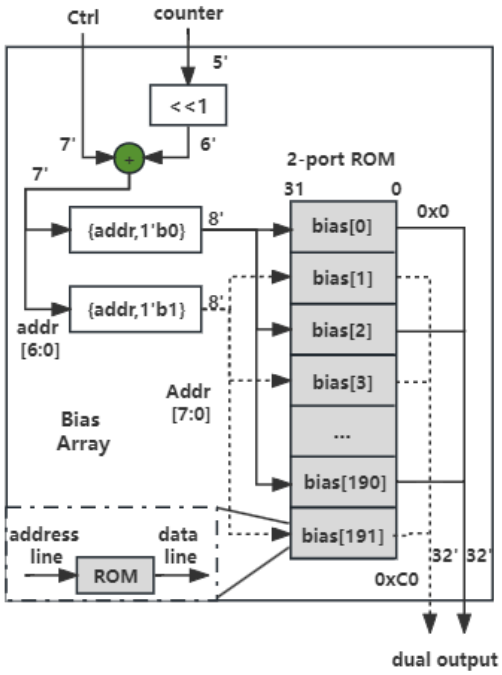


Figure 4.9: Structure of Bias Array Unit inside Linear 32x96 Unit

As mentioned before to save space for bias array, the dual channel bias array has been concatenated in the way shown in the figure, which odd slot stores the bias needed in odd channel,and even slot stores the even channel data. Then similarly the counter would be left shifted by 1 and add with the *Ctrl* control signal and extended 0 and 1 in the LSB to index odd and even address.

The ROM stores 2*96, which is 192, elements and each element consume 32 bits of width.

Hardware Resource

The unit's hardware is all module inside the gray block in figure 4.6, which is consist of two channels of the Linear 32x1 Unit. As each Linear 32x1 Unit needs to operate multiply operation in parallel, firstly 32 unit of float-point multiplier is required. Then the 32-input add-tree structure would consume 32-1, which is 31, units of float-point adder. Finally each channel would need a float-point adder to add the bias.

Under this architecture, the hardware would need 32 float-point multiplier and 32 float-point adder in total. Similarly, in order to achieve full-pipelined structure and thus provide highest throughput, all the arithmetic IP like float-point adder and multiplier have been configured with II=1 and latency=1 to make the whole GRU layer has lower latency as possible, and they would consume 2 and 3 DSP resource respectively.

In term of ROM, as weight array needs 16 groups of 2-port ROM which each of them holds 192x32 bits, two BRAM18k units are need to concatenate together to provide the storing slot of 32 bits. As the storing depth is 192 in each 2-port ROM, the 1k space of BRAM18k is capable for the requirement. The bias array only needs one 2-port ROM so only two BRAM18k are needed. In total the unit consumes 34 number of BRAM18k resource.

Unit Timing

The proposed timing of Linear 32x96 unit is shown as in figure 4.10.

The *valid* signal and *ctrl* and *din* are proposed to arrive synchronously to the module, and the *valid* signal needs to be high for 16 cycles to compute 32 results using the unit while the *ctrl* signal indicate the base address for 2-port ROM indexing to the weight and bias matrix (0x0 for computing x_z, 0x20 for x_r, 0x40 for x_h as they are both 32-bit in size so the address increment is 32 which is 0x20 hex-decimal) should keep no-change during the valid session.

In this process, when the counter detects the high valid, it would start to count, and the corresponding address would be concatenated and shifted to have the index for weight and bias. Then as the BRAM has been configured to have read latency=1 cycle, the weight and bias value would be available after one cycle.

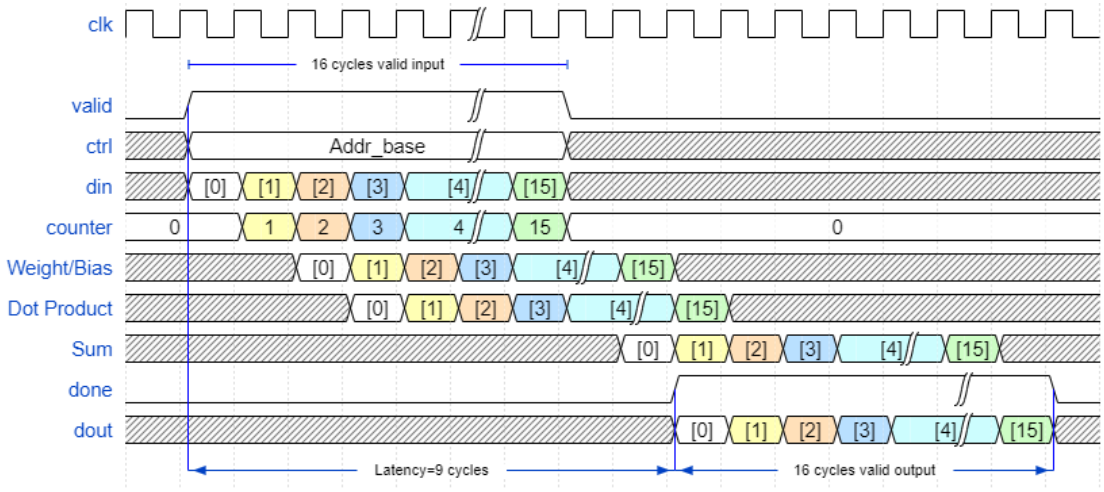


Figure 4.10: Timing graph of Linear 32x96 Unit

The two-cycle buffered *din* data (valid weight data is two cycles behind the valid *din*, so a shift register with depth of 2 has been applied to align the signal) would then start to multiply with the 32-sized weight in parallel. Then the result will be sent to the 32-input add-tree, whose 5-layer structure contribute a 5-cycle latency. Finally the adder would add the summation result and five-cycle buffered bias (valid bias data is being read out the same cycle with weight, so is one cycle above the valid *DotProduct*, so a shift register with depth of 1 has been applied to align the signal) together to have the final output.

Finally the *done* signal would be high for the same high cycle of input *valid*, and output the computed *dout* in dual channel (first for odd index result, and other for even index) with overall latency=9 cycles. And as unit is full-pipelined structure, the unit's II=1.

Simulation

The unit simulation result is shown as in figure 4.11 whose dual-channel input are both 32-sized vector of all 0.7 and weight and bias are the same parameter of GRU layer2 W related matrix. The unit result has been compared with C++ simulation result and showed a good match while the module latency is tested to be 9 cycles with II=1, indicating the unit is functioning correctly.

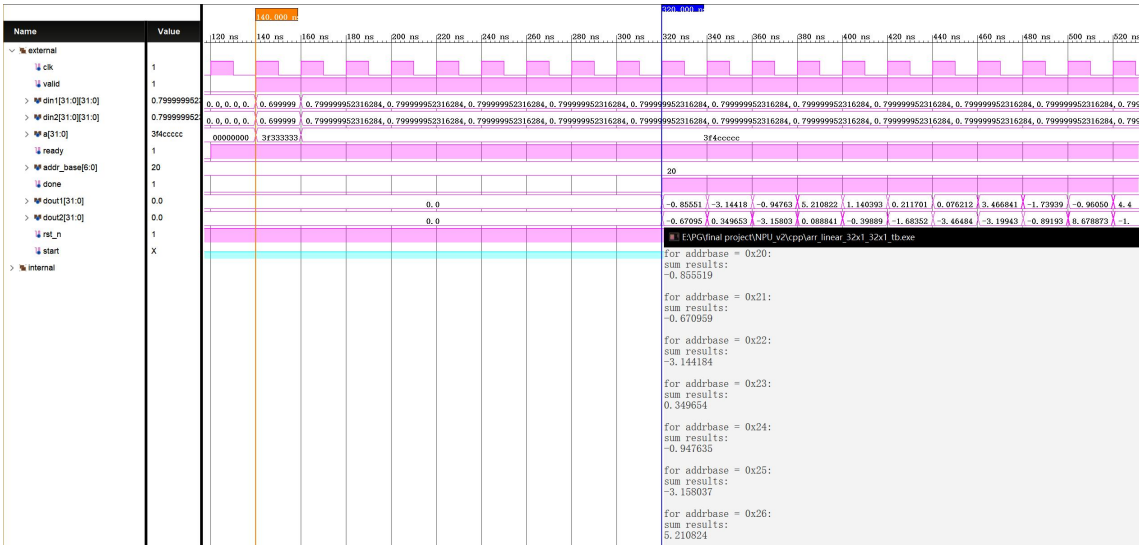


Figure 4.11: Simulation of Linear 32x96 Unit

4.2.5 Dense Unit

Overall Architecture

The Dense unit in the network contains three types of operation: 32x32, 32x16 and 16x1, which Dense 32-sized input to 32-sized output, 32-sized input to 16-sized output and 16-sized input to 1-sized output respectively. During the process, the input array would be firstly be multiplied with a same-sized weight array and then sum the vector and finally add the bias and output ReLU result. Notably, as the weight-sum-bias process is the same with linear operation, and previous Linear 32x1 Unit is able to support 32-sized input at maximum, hardware resource reused is possible who reuses existed Linear 32x1 Units inside W and U related computation module to save the hardware resource.

Considering all the Dense operation is conducted after the GRU layer where all existed Linear 32x1 Units located, there is no resource conflict across computing process. Besides, as Linear 32x96 Unit itself already contains the control logic to support the dual-channel iteration of Linear 32x1 Unit in 16 times (32 iteration in total if combined all channels together), the transplant of Dense operation with iteration of 32, 16 and 1 would be feasible which means not excessive redundant hardware. However, as its inner $WeightArray$ and $BiasArray$ are dedicated to store the Linear matrix, another way to store and read the Dense needs to be examined.

Intuitively, there are two solutions where the first one is to append the corresponding matrix at the end of Linear matrix and another is to instantiate more 2-port ROMs to store matrix separately

and mux the memory output. The first solution has the advantage of saving BRAM resources by filling the unused space in existing BRAM as there is no read-out conflict in time dimension compared to the second one. However, this solution needs to modify the address indexing logic and module control logic. In this section, the second solution which instantiate extra ROM has been applied to make this version transplant friendly which could be modified easily into full-customized IP version.

The overall reused Dense/Linear Unit architecture is shown as in figure 4.12.

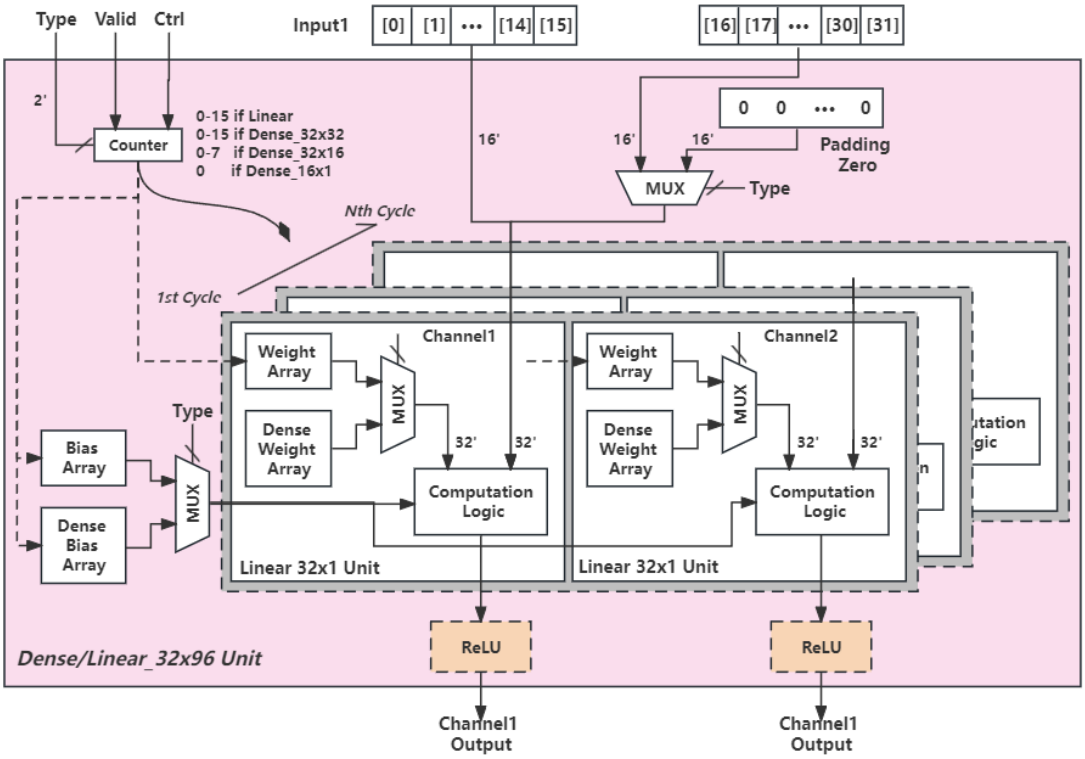


Figure 4.12: Architecture of reused Dense/Linear Unit

An extra 2-bit control signal of *Type* has been added to control the unit conduct its original Linear 32x96 operation, or choose to operate Dense 32x32/32x16/16x1 function (0 for Linear, 1 for Dense 32x32, 2 for Dense 32x16 and 3 for Dense 16x1). On the input side, as the Dense 16x1 operation only have valid 16 elements input vector, the higher 16 elements module input needs to be set to 0 in case it interferes the succeeding multiplication and thus summation. A muxer controlled by *Type* has been applied to mux the higher 16 element input of the Dense/Linear 32x96 Unit to be original input or the 16-element padded zero array.

And the counter would also be controlled by *Type* to operate within correct iteration cycles

(0-15 when conduct Linear or Dense 32x32 as iterates 32 times, 0-8 when Dense 32x16 as iterates 16 times and 0 when Dense 16x1 as only iterates one time), and control the index of Bias and Weight Array of Linear and Dense Unit. Also, two muxers have been applied to select the valid weight and bias input of parallel multiplier and adder inside the computation logic according to *Type*. Finally, the ReLU Unit will be used if Dense operation has been detected while be bypassed if Linear detected.

4

Memory Array

The weight storing methodology is shown as in figure 4.13 which takes the Dense 32x32 operation as example.

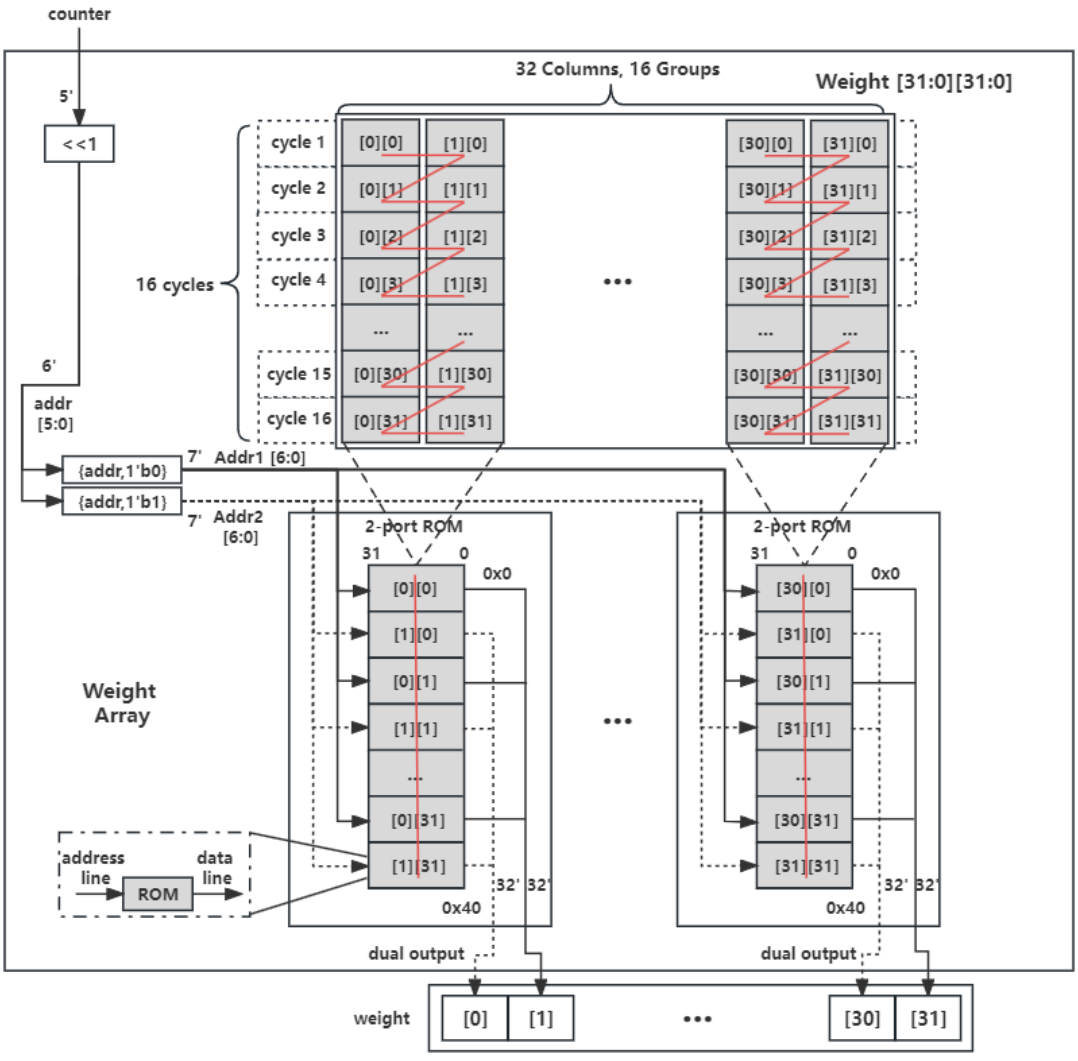


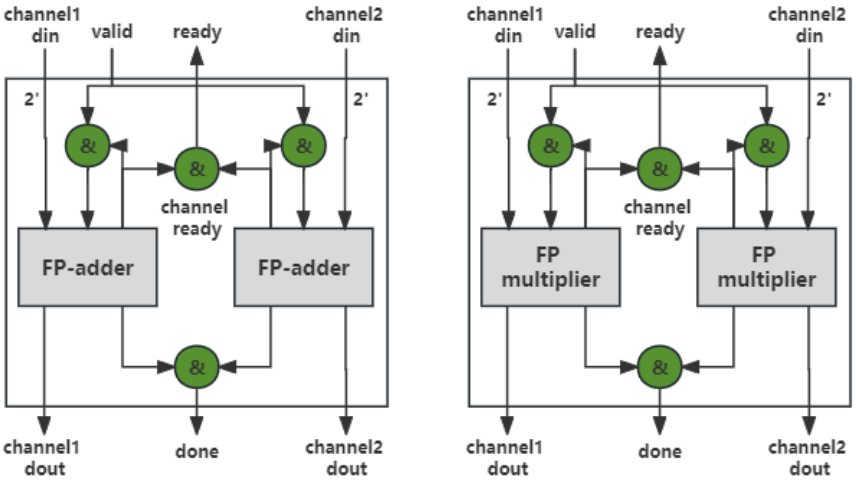
Figure 4.13: Structure of Dense Weight Array Unit of Dense/Linear 32x96 Unit

The Dense 32x32 and Dense 32x16 array is partitioned similarly with the structure in 4.8 who also have 32 columns and each row is needed per cycle. Sixteen of 2-port ROM unit, which each of them stores a group of matrix in ZigZag way, has been instantiated with storing slot of 32-bit width but storing depth of 64 as they only have 32 and 16 rows respectively.

In term of Dense 16x1 array, as only 16 weights are need per cycle, they are divided into 8 groups and instantiated 8 of 2-port ROM unit with depth of 2. For bias unit, the memory for two channels would also be concatenated together and stored in one 2-port ROM. In total, 2*16*2+2*8+3, which is 51 number of BRAM18k has been used.

4.2.6 Float-point Adder and Multiplier

The structure of dual channel float point adder and multiplier is shown as in figure 4.14.



(a) Dual channel float point adder (b) Dual channel float point multiplier

Figure 4.14: Structure of dual channel Float-point adder and multiplier

The gray block indicates the instantiated Vivado float point adder or multiplier IP. To make the unit and system has the highest throughput, the II and latency of these IP have been configured to 1 with the cost of consuming 2 and 3 DSP resource respectively.

In term of control interface, the *valid* signal would directly start two IP in synchronous, and the unit's valid input would operate logic-and with its ready signal to finish the hand-shake. And the unit's output *ready* and *done* signal is the logic-and result of ready and done signal of all channels respectively. Besides, the module is constructed in a extensible way whose channel number could be configure by the interface parameter.

4.2.7 Sigmoid Unit

The structure of dual channel *Sigmoid* Unit is shown as in figure 4.15.

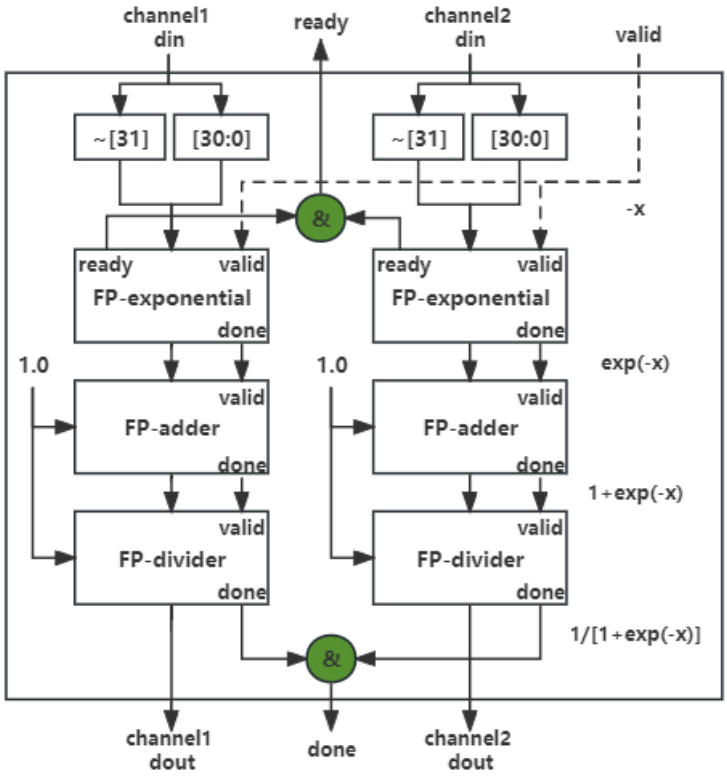


Figure 4.15: Structure of dual channel Sigmoid unit

The 32-bit input of each channel would be firstly reverse the Most Significant Bit (MSB) bit (defined to be the sign bit in IEEE-754 protocol and 0 represent positive number, 1 for negative number) and concatenate with the remaining bits to have term $-x$. Then it will be sent to the float point exponential IP to compute the term e^{-x}. In hardware level, this is done by instantiating a Vivado float point exponential IP, and to make system capable for 100MHz clock while be as fast as possible, the II of IP has been configured to be 1 and latency to be 4 (4 pipeline stages) which would consume 7 DSP units. After this, the result will be sent to a float-point adder with another operand of 1.0 to have $1 + e^{-x}$, and the adder has been configured to have II=1 and latency=1. Finally the output of adder would be connected to a float-point divider who would divide the result by 1.0, resulting a $\frac{1}{1+e^{-x}}$, which is the *Sigmoid* function, and output the result.

As the float point exponential, adder and divider unit are cascaded, their valid and done signals are also connected in a cascaded way which the done of previous unit would connect to the valid of

this stage unit, and start the unit when its ready is also high by operating logic-and with the ready signal. This would make the unit automatically shake hand in each active cycle which securing the signal's validity. In term of control signal, the unit *valid* signal would control the start of each channel, and the ready and done would be the logic-and operation of all channel's personal signal.

4.2.8 Tanh Unit

Computation Algorithm

Traditionally there are two classic ways to calculate the $Tanh$: one is to expand the $Tanh$ into equation 4.1:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.1}$$

where another one is to calculate it using equation 4.2.

$$tanh(x) = \frac{sinh(x)}{cosh(x)} \tag{4.2}$$

The first one do not need other hyperbolic computation unit but need to have 4 exponential units and one float point divider. As analysis in the previous section, a highest throughput exponential unit would need 7 DSPs and four cycles to finish the computation which demands a intensive hardware resources. For this reason, this project intends to compute the hyperbolic tangent function using $\frac{sinh}{cosh}$ whose hyperbolic function generated by CORDIC unit and then pass to a divider to have the final result.

Hardware Architecture

The structure of dual channel $Tanh$ Unit is shown as in figure 4.16.

As the CORDIC IP in Vivado has following features: 1)the input must be 32-bit fixed point format with 3 bits of integer-bit; 2) the input should be located within ± 1.1 otherwise the algorithm would not converge and thus output invalid result; 3) the output is also in fixed point format with 32bits in width and 2 bits of integer-bit, there needs some peripheral circuits to support its operation.

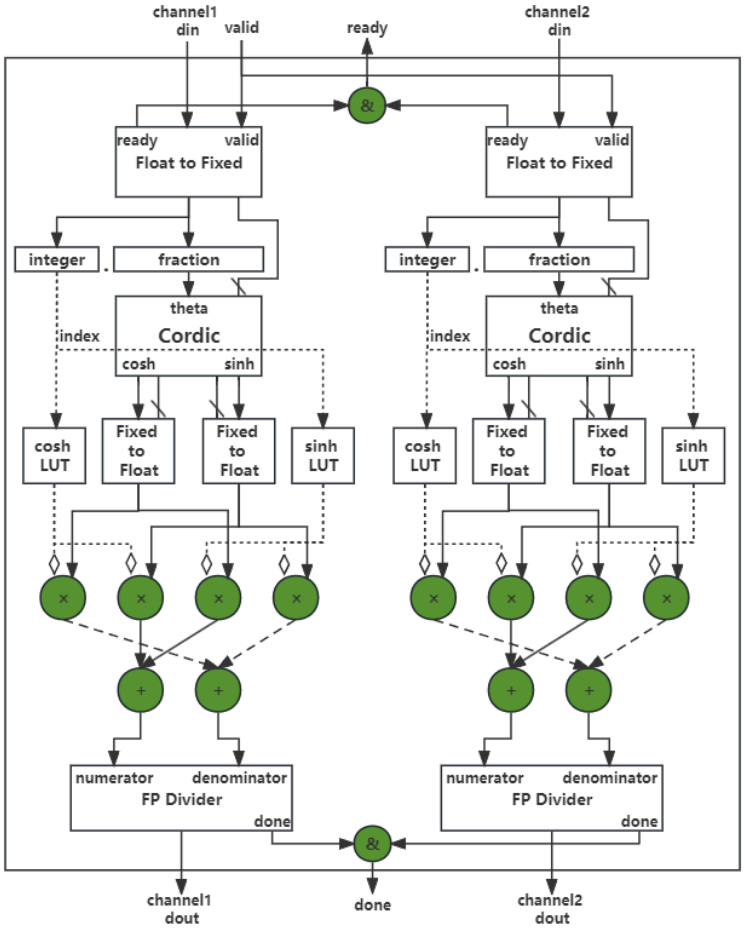


Figure 4.16: Structure of dual channel Tanh unit

Firstly, a float to fix point converter is instantiated in each channel to convert the 32-bit single float point number to a 32-bit fixed number with 3 integer-bit and convert the fixed point of output port back to the float point number with a fix to float point converter. To make the system has highest throughput, two converters per channel has been configured to have II=1 and latency=1.

Then as the input of CORDIC is limited with ± 1.1 but the input number of $Tanh$ unit in the system are mostly located within ± 10, the IP's converge range, which is the valid input range, should be extended.

Two ways are possible for this extension: the first one directly set any number greater than converge range to its infinite value (if greater than 1.1, set to $tanh(\text{inf})$ which is 1.0 while if less than -1.1 then set to $tanh(-\text{inf})$ which is -1.0). However, considering the $tanh(1.1) = 0.8005$, if applying this deprecation method, there would be an error of 0.2 across all the number greater than the convergence range. As these part of number hold major percentage in data space and

this error would accumulate within the 196 times of iteration which would exaggerate during the process, this method is not applicable considering the data integrity.

The second method is applied by expanding $tanh(x)$ with equation 4.2 and dissemble x into integer part int and fractional part $frac$. Then the function could written as equation 4.3

$$tanh(x) = \frac{sinh(int + frac)}{cosh(int + frac)} = \frac{sinh(int)cosh(frac) + cosh(int)sinh(frac)}{cosh(int)cosh(frac) + sinh(int)sinh(frac)} \tag{4.3}$$

where the $sinh(frac)$ and $cosh(frac)$ could be computed with the CORDIC IP since its range is within ± 1.0. And the $sinh(int)$ and $cosh(int)$ could be stored in a look-up-table in advance and index the table using integer part value. Considering the range of practical input number is not infinite, the integer number could be enumerated easily before use.

In the project, second method is selected as the trade-off between hardware resource and speed. The converted CORDIC output would be connected to 4 float point multiplier and compute the dot product in equation 4.3. Then after two float-point adder, the product would be added together and finally send to the float-point divider to have the unit's output. Similarly, the II and latency of float point adder and multiplier has been set to 1 to increase the throughtput. And as the complexity of float point divider, it has only be configured to have latency=5 and II=1.

In term of control logic, similar as other dual-channel IP, the unit's *valid* signal would control all the channel to start synchronously, and the *ready* and *done* output is the logic-and operation of all channel's personal signal. Besides, all the previous stage done is connected to next stage's valid to configure the auto hand-shake between layers.

4.3 System Temporal Scheduling and IP Control

In previous section, the architecture of RNN network has been defined whose major computation unit is cascaded GRU and MLP. As inner structure of GRU and MLP has been analyzed and required IP has been constructed, the only left work is to scheduling these IP and assemble them together to build the GRU and MLP unit and moreover to build the whole system. In this section, GRU unit would be firstly scheduled and established, and after the construction of MLP, the assembling and scheduling of the whole system would be done.

4.3.1 GRU Unit Scheduling and Control

Scheduling

The required IPs for the construction of GRU Unit are shown as in Table 5.4.

Table 4.2: Timing of system IPs

IP (Dual channel)	Latency (cycles)	II (cycles)	I/O Timing
Float-point Add	1	1	N[i]cycle high
Float-point Multiply	1	1	N cycle high
Linear 1x96	4	1	16[ii]cycle high
Linear 32x96	9	1	16 cycle high
Sigmoid	10	1	N cycle high
Tanh	45	1	N cycle high

[i] N could be any number of cycle.
[ii] Regulated 16 cycles is for a batch of computation (32 element output), and M batch of input should have M*16 cycle of high *valid*.

Since the Combination Arithmetic Unit compute the expression $z*h\_tm+(1-z)*hh$, it could be dissembled to the combination of float point adder and multiplier. Firstly the adder would compute $1+(-z)$, then two multiplier would have the $z*h\_tm$ and $(1-z)*hh$ and finally an adder would add two product together to have the updated h_t.

As the analysis in previous sections, the critical path of GRU layer is the hh path. As z, r and h all need results from previous linear module to be computed, the hh path, which needs to add the r_h and the product of r and r_h and then send to $Tanh$ unit, has the longest path since it not only need the linear results, but also require other results that also uses linear result to be ready. So the r path is arranged to be the first priority, and the linear module would firstly compute the x_r and re_r which is used for r computation, and then compute x_h and re_h to generate the hh value while x_z and re_z which is used for z generation. Finally the Combination Arithmetic Unit would update the h_t vector.

Under this methodology, the IP scheduling within the GRU could be drafted as in figure 4.17 (take GRU layer1 as example) where one Linear 32x96 Unit and 1x96 Unit, three float point adder, two float point multiplier and one Sigmoid and Tanh Unit has been instantiated to have a balance between performance and hardware resources.

As three "x" prefixed 32-element variables (x_z, x_r, x_h) are relay on the Linear 1x96 Unit and "re" prefixed variables are depends on Linear 32x96 Unit, their processing order needs to be determined first considering each Linear Unit would be instantiated once in each layer. As hh path

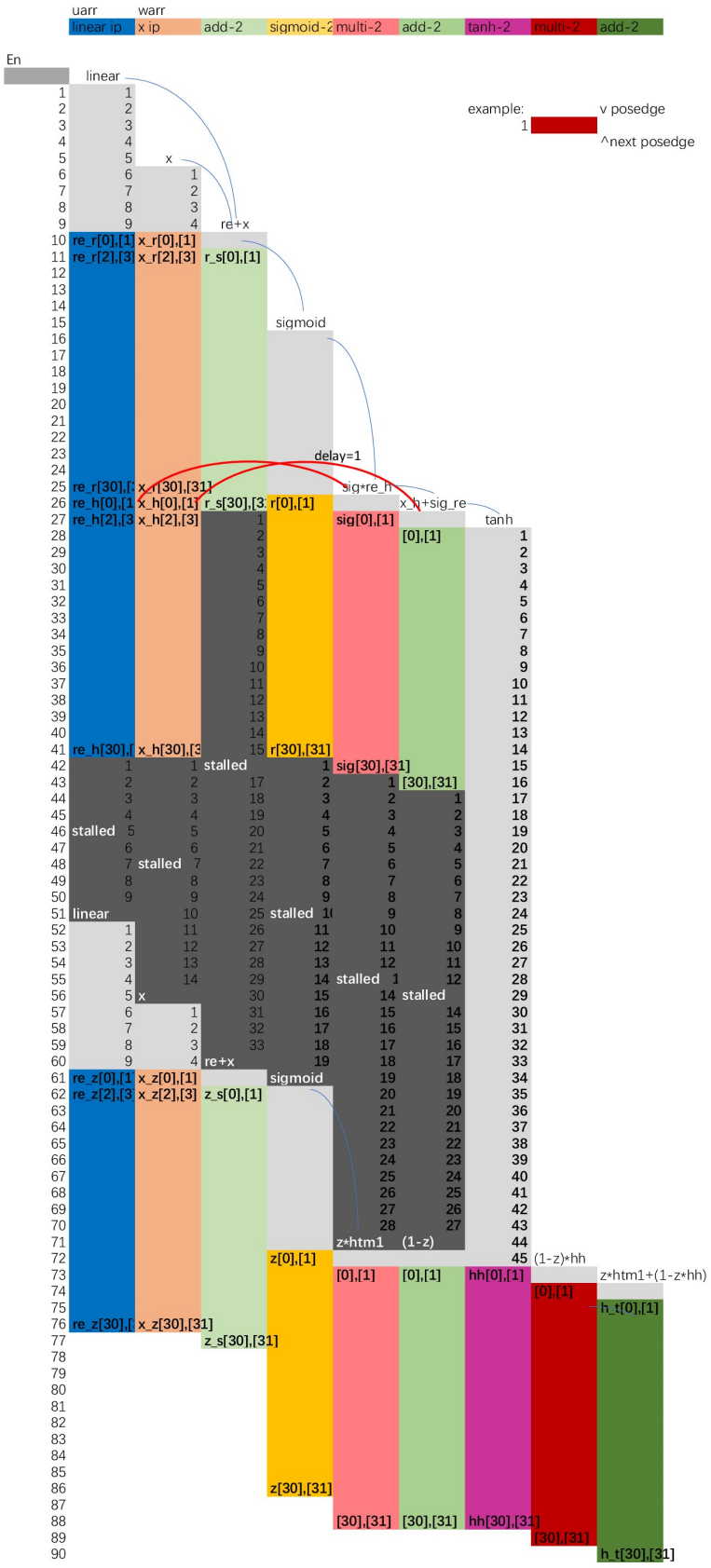


Figure 4.17: System timing schedule of arithmetic IPs where blue pool indicate the timing of Linear 32x96 Unit, orange for Linear 1x96 Unit, green for the three float point adder, red for the two float point multiplier and yellow for the Sigmoid unit and purple for the Tanh Unit

is critical path which depends on r, the r related x_r and re_r would be computed firstly, and then the x_h and re_h and finally the x_z and re_z. Since Linear 1x96 Unit has a latency of 4 cycles but Linear 32x96 Unit has 9, Linear 32x96 Unit would be started ahead at clock cycle 1 and then the Linear 1x96 Unit would be started after 5 cycles, which is at cycle 5, to align the x_r and re_r output whose first result is valid at clock cycle 10. As all IPs are constructed in a dual-channel way, 16 cycles of input is enough for the "_r" computation. Then it comes with the choice of whether stop the Linear Unit or keep it running to calculate other data, and the answer of this depends on the following data dependency and would be rearranged after alignment.

After this, the first float point adder (light green) would starts to add the output of two Linear Unit at the cycle of 10. Then after 1 cycle, the output of the first adder would be ready at the cycle of 11. And the scheduling of Sigmoid Unit becomes the problem since it only takes 10 cycles to generate the first output. As the computation of $re\_h * r$ depends on both the r and re_h, they should be aligned together to be both valid when sending to the multiplier. However, the first output of Sigmoid Unit would be valid at cycle of 21, which is 5 cycles ahead of first valid re_h at cycle of 26. Then the Sigmoid Unit is considered to have a delayed start-up for 5 cycles behind who will now start at the cycle of 16 to make its first valid output at the cycle of 26 which is aligned with re_h, and this makes the Linear Unit to continue computing after finishing the re_r and x_r.

However, since the first valid output of adder would be 5 cycles ahead of the start of the Sigmoid Unit and the adder will still be working when Sigmoid's output is ready, adder's results would be deprecated considering full-pipelined structure has been applied as each cycle would update a new value within the working cycle and the old one would be deprecated. Then a shift register with depth of 5 would be instantiated as in figure 4.18 to store the output of adder and after 5 cycles of shifting, its result would be sent to the input of Sigmoid Unit.

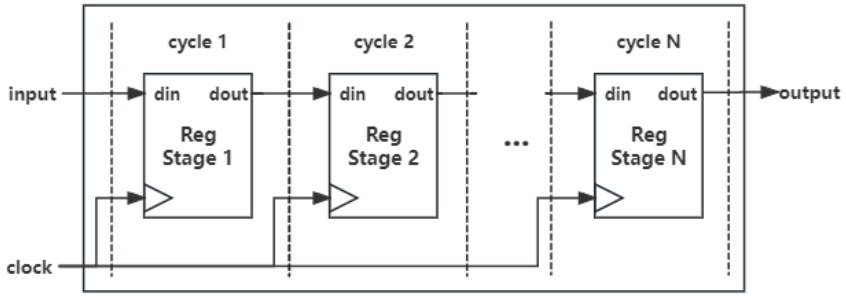


Figure 4.18: Structure of N stage shift register

Then the output of Sigmoid Unit would connect to the first float point multiplier and multiply with aligned re_h to have the r. Then the r would be added with x_h to generate the input of Tanh Unit. Similarly, the x_h and re_h are aligned together whose first output is valid at cycle of 26, but x_h would be needed at cycle of 27 as the latency of multiplier is 1. Then another shift register with depth of 1 is needed to delay the x_h and align it to the output of first multiplier. Besides, as the calculation of Tanh's input overlaps the caculation of Sigmoid input on working cycle, they cannot be scheduled to have hardware reuse and two adder unit is needed.

After 45 cycles of delay, the first valid output of Tanh Unit would be at cycle of 73. As the latency of Tanh is extremely high, and final Combination Arithmetic Unit needs its result of hh to be aligned with z, the computation of z do not need to start immediately after Linear Units finishing re_h and x_h and therefore could be stalled for several cycles to avoid unnecessary shift-register of their output. Then their start time would be backward computed for the alignment of z and hh.

As the Combination Arithmetic Unit has been dissembled to $z * h\_tm + (1 - z) * hh$, firstly the term $z * h\_tm$ and $(1 - z) * hh$ would be calculated and added together. As z could be valid ahead of hh, these two terms should be aligned with hh as reference. Then the $1 - z$ term needs to be aligned with hh at the cycle of 73. Considering it has a latency of 1, the adder should be started at cycle of 72.

Also, the start of $(1 - z)$ means that z is needs to be valid at cycle of 72 which constrains the start time of Sigmoid to cycle of 62. In addition, as h_tm vector is always valid across each iteration in the GRU unit, the computation of $z * h\_tm$ could also be started at cycle of 72 for z is valid. Under the same methodology, the re-start time of Linear 32x96 Unit could be computed to at the cycle of 52 and 1x96 Unit to be at 57 and succeeding adder to be at 61.

In term of hardware, the output of Sigmoid would also need a 5-cycled delay which is proposed to reuse the shift-register. And an extra shift-register would be needed to delay the $z * h\_tm$ for 1 cycle (latency of adder) to be aligned with $(1-z)*hh$. After analysis of this part overlap, one more adder and multiplier are needed to compute $z * h\_tm + (1 - z) * hh$ and $(1 - z) * hh$ respectively. The $z * h\_tm$ term could use previous multiplier to compute and $(1 - z)$ could be use the second adder.

Control

As the scheduling of IPs in previous section is accurate in cycle-level and there is multiple state to control, a Finite State Machine (FSM) is considered to use to achieve this. The control information of these IPs is shown as in Table 5.5

Table 4.3: Control information of system IPs

IP	Start Time	End Time	Source
Linear 1x96[i]	5 56	37 72	*input*
Linear 32x96	0 51	32 67	*h_tm*
1st Adder[ii]	9 60	25 76	Linear 1x96, Linear 32x96
2nd Adder	26 71	42 87	1st Multiplier, Linear 1x96[iii] 1.0, Sigmoid
3rd Adder	73	89	1st Multiplier[iv],2nd Multiplier
1st Multiplier	25 71	41 87	Sigmoid, Linear 32x96 *h_tm*, Sigmoid
2nd Multiplier	72	88	Tanh, 2nd Adder
Sigmoid Unit	15 61	31 77	1st Adder[v]
Tanh Unit	27	43	2nd Adder

[i] Second line indicate the second round start and end time of unit
[ii] The float point multiplier and adder has two oprand so they have two input sources.
[iii] Need 1 cycles of delay during first round operation
[iv] Need 1 cycles of delay
[v] Need 5 cycles of delay during first round operation

And for the convenience of transplanting the system, the start and end time of IPs could be defined with parameters of delay as following:

```
1   `define WARR_L1_DELAY   4       // only for warr_l1
2   `define ARR_DELAY       9       // for uarr_l1, warr_l2, uarr_l2
3   `define ADD_DELAY       1
4   `define DIV_DELAY       5
5   `define EXP_DELAY       4
6   `define SIGMOID_DELAY   (`EXP_DELAY+`ADD_DELAY+`DIV_DELAY)
7   `define MULTIPLY_DELAY  1
8   `define TANH_DELAY      (40+`DIV_DELAY)
9
10  // defined timing point for ff update. If for combinational logic, add 1
11  // latency to all point.
12  `define ARR_START_1             0
13  `define ARR_END_1               (`ARR_START_1+32)
14
15  `define WARR_L1_START_1         (`ARR_DELAY-`WARR_L1_DELAY)
16  `define WARR_L1_END_1           (`WARR_L1_START_1+32)
17
18  //depends on warr and uarr, suppose delay: uarr >= warr
19  `define ADD_UNIT1_START_1       (`ARR_START_1+`ARR_DELAY)
```

```
20      `define ADD_UNIT1_END_1          (`ADD_UNIT1_START_1+16)
21
22      //depends on add unit1
23      `define SIGMOID_START_1          (`ARR_START_1+`ARR_DELAY+16-`SIGMOID_DELAY)
24      `define SIGMOID_END_1            (`SIGMOID_START_1+16)
25
26      //depends on sigmoid
27      `define MULTIPLY_UNIT1_START_1   (`SIGMOID_START_1+`SIGMOID_DELAY)
28      `define MULTIPLY_UNIT1_END_1     (`MULTIPLY_UNIT1_START_1+16)
29
30      //depends on multiply unit1
31      `define ADD_UNIT2_START_1        (`MULTIPLY_UNIT1_START_1+`MULTIPLY_DELAY)
32      `define ADD_UNIT2_END_1          (`ADD_UNIT2_START_1+16)
33
34      //depends on add unit2
35      `define TANH_START               (`ADD_UNIT2_START_1+`ADD_DELAY)
36      `define TANH_END                 (`TANH_START+16)
37
38      //depends on tanh and add unit2 (should be aligned with first tanh result)
39      `define MULTIPLY_UNIT2_START     (`TANH_START+`TANH_DELAY)
40      `define MULTIPLY_UNIT2_END       (`MULTIPLY_UNIT2_START+16)
41
42      //depends on multiply unit2 and multiply uni1 (need ff delay buffer)
43      `define ADD_UNIT3_START          (`MULTIPLY_UNIT2_START+`MULTIPLY_DELAY)
44      `define ADD_UNIT3_END            (`ADD_UNIT3_START+16)
45
46      // second time start-up of add unit2 and multiply unit1 should be aligned
47      // with first result of tanh, so the start time is calculated in backward//
48
49      //depends on sigmoid, and should be aligned with first tanh result
50      `define ADD_UNIT2_START_2        (`TANH_START+`TANH_DELAY-`ADD_DELAY)
51      `define ADD_UNIT2_END_2          (`ADD_UNIT2_START_2+16)
52
53      //depends on sigmoid, and should be aligned with first tanh result
54      `define MULTIPLY_UNIT1_START_2   (`TANH_START+`TANH_DELAY-`MULTIPLY_DELAY)
55      `define MULTIPLY_UNIT1_END_2     (`MULTIPLY_UNIT1_START_2+16)
56
57      //depends on add unit1
58      `define SIGMOID_START_2          (`ADD_UNIT2_START_2-`SIGMOID_DELAY)
59      `define SIGMOID_END_2            (`SIGMOID_START_2+16)
60
61      //depends on warr and uarr, suppose delay: uarr >= warr
62      `define ADD_UNIT1_START_2        (`SIGMOID_START_2-`ADD_DELAY)
63      `define ADD_UNIT1_END_2          (`ADD_UNIT1_START_2+16)
64
65      `define WARR_L1_START_2          (`ADD_UNIT1_START_2-`WARR_L1_DELAY)
66      `define WARR_L1_END_2            (`WARR_L1_START_2+16)
67
68      `define ARR_START_2              (`ADD_UNIT1_START_2-`ARR_DELAY)
69      `define ARR_END_2                (`ARR_START_2+16)
```

The scheduling of second GRU layer is mostly the same with the first one, excepting changing the Linear 1x96 Unit to the Linear 32x96 Unit to take the output of previous layer, h_tm, as the

input and conducts the operation. Then the start time and end time of two Linear 32x96 Unit would be the same and they will be controlled together.

In term of IP input resource, the Linear 1x96, Linear 32x96, 1st Adder, 3rd Adder, 2nd Multiplier, Sigmoid and Tanh Unit have fixed input source so they will be directly wired to the source as shown in Table 5.5 and wired to the shift register if the source need to be delayed. The 2nd Adder, 1st Multiplier have multiple input source during different round of computation, so muxers are used to mux out the input source according to the FSM state which reflecting the computation round.

Then the timing of GRU Unit could be defined as in figure 4.19.

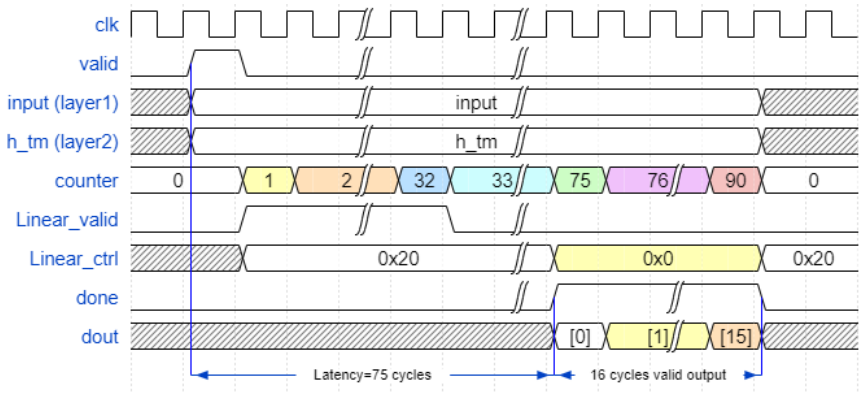


Figure 4.19: Timing of proposed GRU Unit

The *valid* signal of unit is the high pulse wave for one cycle, and the 1-sized input of layer1 or 32-sized *h_tm* of layer2 are required to be valid with the *valid* signal and remains unchanged during the computation of GRU Unit. Then after the valid is detected to be high, a inner counter would starts to count whose number would be considered as the state of FSM.

The IPs inside the Unit would start and end according to the current counter number (state). For example, the valid signal of Linear 32x96 would be set high since counter is 1 and keep for 32 cycles, resetting when counts to 32. Also, the *ctrl* signal of Linear Unit would be changed synchronously and keeps unchanged across unit's working cycle. Finally after 75 cycles, the first two output would be valid and the done signal would be high since then for 16 cycles. As all the results have been transmitted, the unit's inner register like *ctrl* or counter would be reset to prepare for the next round operation. According to the timing graph, the overall II of GRU Unit is 90 cycles and latency is also 90 cycles

4.3.2 MLP Unit Scheduling and Control

Scheduling

As the analysis in the previous section, the realization of Dense layer of MLP is by the reuse of Linear Unit. And since the three layer of Dense operation is cascaded itself and they are all behind the GRU computation, the Linear Unit could be started again as Dense Unit without any temporal crash. To gain the easy control of Linear Unit, only the Linear 32x96 in GRU layer 2 which computing the W related matrix has been modified to support the Dense computation. This is reasonable considering the GRU Unit would take at least 196*91, which is 17836 cycles, but the Dense operation only takes 16+8+1, which is 25 cycles, if only utilizing one dual-channel Linear Unit. The acceleration of Dense layer would only save 22 cycles at most but would waste 2 extra complicated control logic and make all the Linear 32x96 terrible in time constraint.

The timing of MLP Unit is shown as in figure 4.20.

Firstly the system would enable the Linear Unit for 16 cycles and mux the Memory output to operate Dense 32x32 operation; Then the system would enable Linear Unit for 8 cycles and 1 cycles to finish the calculation of Dense 32x16 and Dense 16x1 layer. And the result would be valid for 16, 8, 1 cycles for Dense 32x32, 32x16 and 16x1 respectively.

During the computation process, the input of Linear is required to be unchanged and the *ctrl* of Linear Unit has been set to 0x0 as the Memory Array of Dense layer only stores one-time weights and do not need to be partitioned in to $\_z$, $\_r$, etc.

Control

In term o hardware control, as the Unit's *valid* starts the internal counter, the $Linear_valid$ signal is one cycle behind which would result to a extra delay of 1 on the bases of Linear Unit. And the counter would recognize the layer type of Dense operation and automatically reset the value (reset when reach 26, 18, 11 for 32x32, 32x16, 16x1 respectively) when it comes to the end of computation to make the module ready for the next round calculation.

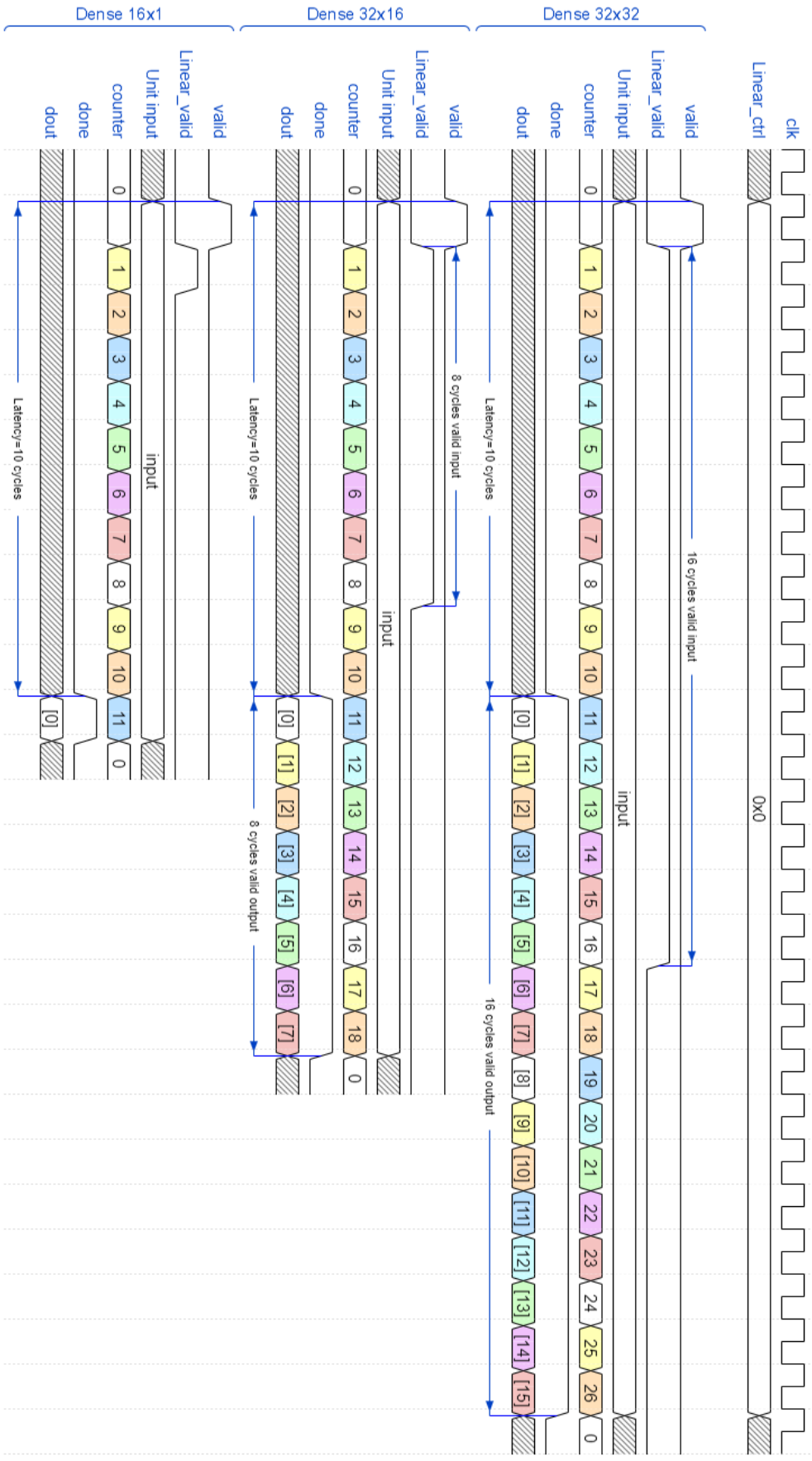


Figure 4.20: Timing of proposed MLP Unit

4.3.3 System Scheduling and Control

Originally the GRU Unit has a cascaded structure shown as the above part in the figure 4.21. In each iteration the GRU layer1 would firstly compute with the 1-sized input, and then send its 32-sized result to GRU layer2. Only when layer2 finished its computation, has one iteration been done and would start the next round of calculation. In this way, the total delay of 196 iteration times is 2*196*GRU_DELAY.

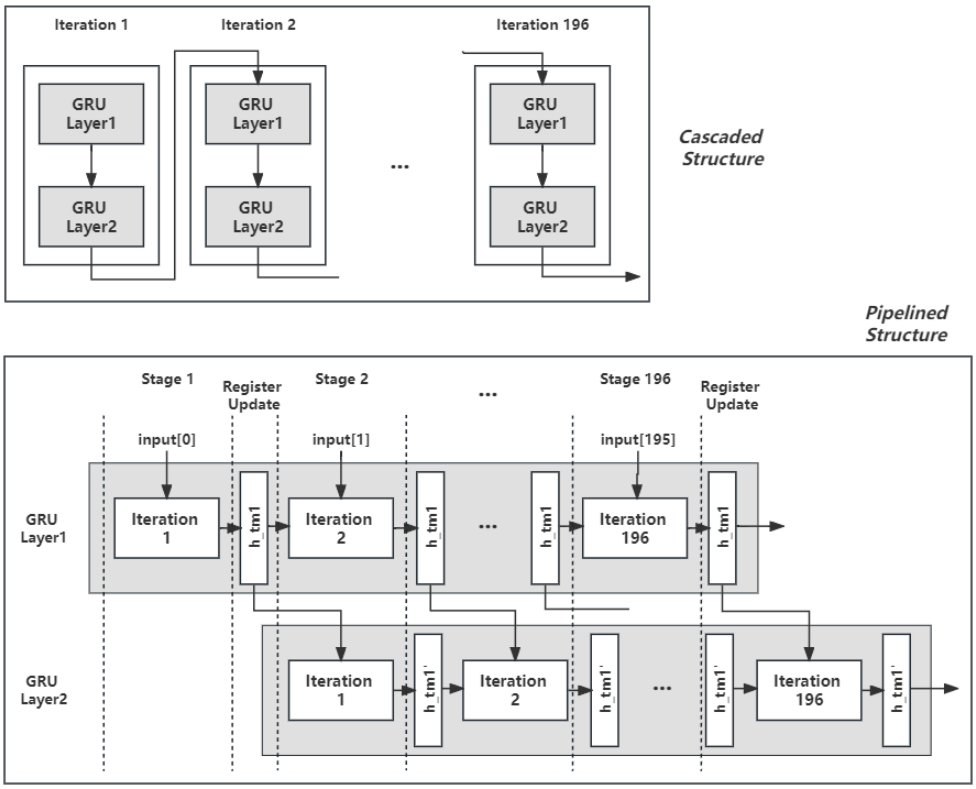


Figure 4.21: Two structures of proposed GRU Unit.

However, as each GRU layer has been instantiated separately and their Array Memory is stored independently inside the Unit, the two cascaded layer could be expanded to a Pipelined structure as shown in the bottom part in figure 4.21. Under this methodology, two layers are executing in parallel while the first layer calculates the workload of this iteration stage and the second layer computes the data of previous iteration stage.

By inserting register group between pipeline stage, the parallelism of these two layers is guaranteed since previous and current state would not be crash (previous state stored in pipeline register and only be updated after computation is done, but current state is generating within the unit). Under this method, the total delay of 196 iteration times would be decreased to 197*GRU_DELAY,

roughly half of the time of cascaded structure.

Then the system timing could be constructed as shown in figure 4.22.

GRU Unit

The GRU layer1 would firstly to be started to make it one iteration ahead of layer2. After it finished its first iteration, the *layer*1*_done* signal would be high for one cycle, and the pipeline register would update the storing data of this iteration results. Then the layer1 would start second round of iteration, and the layer2 would be aligned with it to have module *valid* high pulse at same cycle.

Another control counter would count the iteration number ever time layer1 is done, and would end the GRU layer when all the 196 iteration is done and layer2 drained out its pipeline. Finally the *GRU_done* signal would be high for one cycle to indicate all the GRU computation is done and ready to conduct MLP computation.

MLP Unit

As the MLP Unit reuse the Linear 32x96 Unit inside the GRU layer2, the layer2 would be start again after *GRU_done* is done. To control the layer to support Dense operation, a configure signal would be set to indicate whether to conduct Dense 32x32, 32x16 or 16x1 computation. Then the counter would have different threshold value to reset according to the Dense operation type (26, 18, 11 for Dense 32x32, 32x16 and 16x1 respectively). When the last Dense layer of 16x1 is done, the MLP_done signal would be high for one cycle to indicate the MLP layer is done. Finally a Sigmoid Unit would compute the final reaction rate with MLP layer's result and set the system's *done* to high.

4.4 Simulation

The structure of testbench is shown as in figure 4.23.

A trace unit is firstly inserted into the Design Under Test (DUT) to trace all the intermediate variables, including output of Tanh, Simoigd Unit, updated h_tm1 and h_tm1' value after each

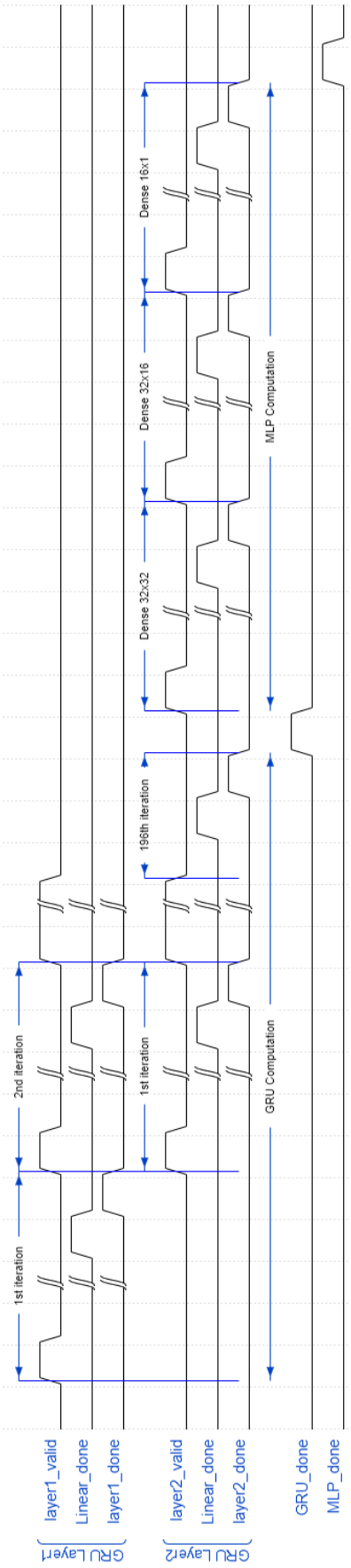


Figure 4.22: Timing of overall system

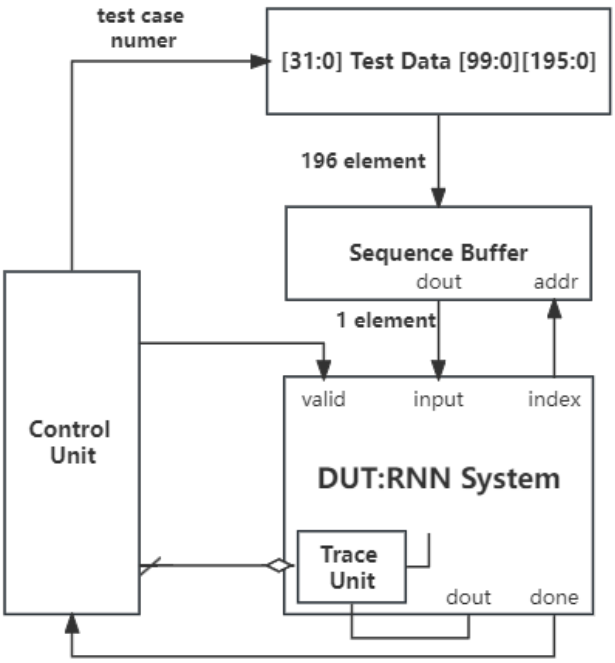


Figure 4.23: Structure of testbench for proposed RNN system

iteration, the final output of RNN system, etc. The the tracer would record these signal into local file to be evaluated in the Evaluation Chapter.

All the 100 cases of test data are stored in an array in testbench where each cases contains 196 32-bit elements. The sequence buffer would take in a case according to the test case number and start to send out values element by element in the test case to the DUT.

Also, a inner counter of RNN system is instantiated to record the iteration number and ask outer memory to provide corresponding 1-sized input data in each iteration. The counter has an output port of *index*, which would be connected to the sequence buffer to provide correct data.

A control unit is then to control all the testing process from the test number counting and boundary check, and enable and disable the DUT and control the trace unit.

The simulation result of DUT under this architecture is shown as in figure 4.24. The 1e-4 level deviation between C++ model (result in the black box) and DUT output (around the yellow cursor) suggesting the proposed RNN system functions correctly. The more timing and resource as well as output accuracy analysis could be seen in Chapter Evaluation.

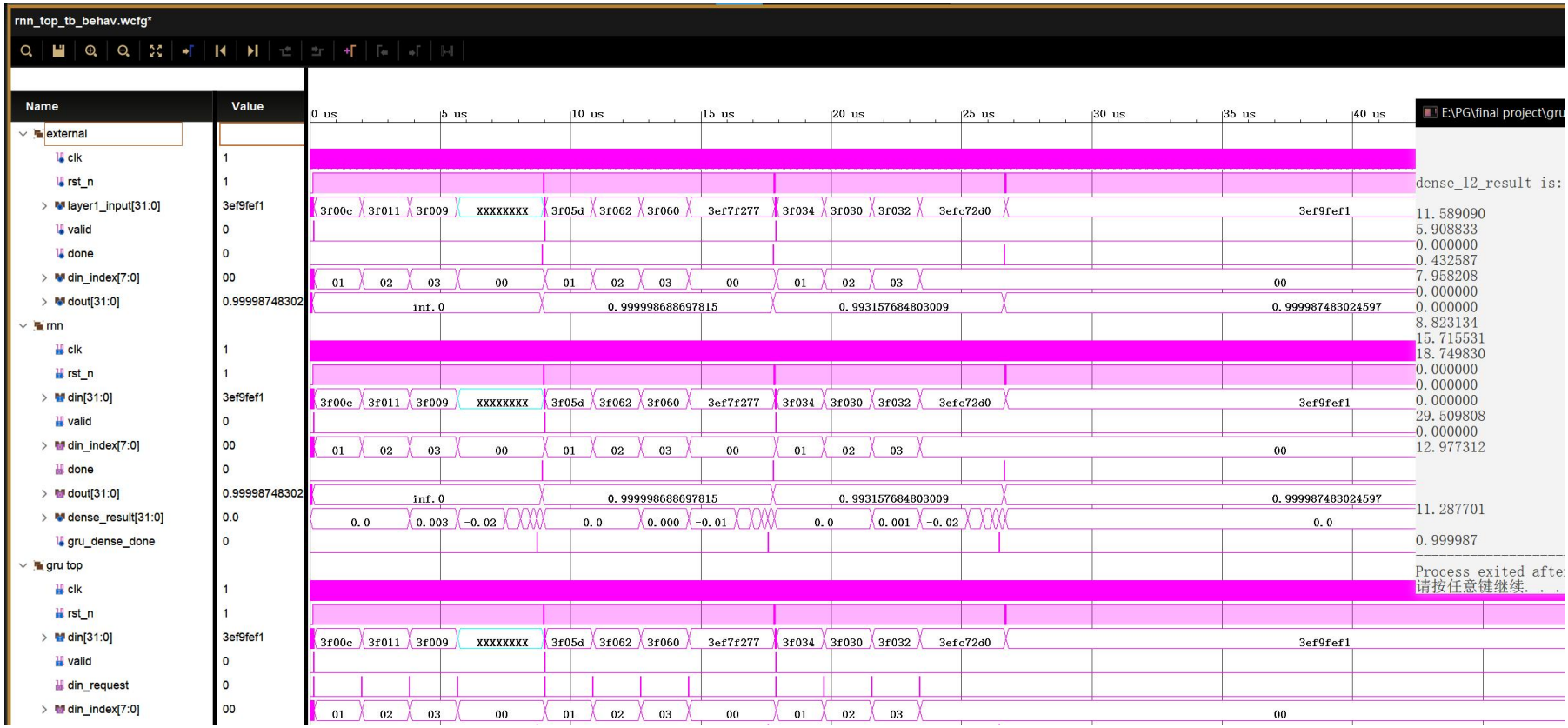


Figure 4.24: Simulation of proposed RNN system (GRU iteration time has been set to 3 to have fast simulation)

4

5

RNN Implementation with Customized IP

Contents

The previous chapter has established the framework and infrastructure of proposed RNN system with commercial Vivado IPs to predict the chemical reaction rate. However, the commercial property constrains it to be only able to run on the Xilinx platform, preventing it to be implemented into ASIC. To this point, this chapter aims to firstly construct the complete customized and proprietary arithmetic IPs to substitute the Vivado IPs. Then with all the available unit, the system would be re-analyzed and scheduled under the framework of previous Vivado Version. Finally the timing is presented and the simulation is done with the same testbench platform, validating the system to be operating correctly.

5.1 Transplantation from Vivado Version

As C++ model and hardware project has been done in the previous chapter, the overall architecture of RNN system is proposed and would remain unchanged if the network structure keeps the same.

Then the transplantation from Vivado Version could directly applying the same framework. However, as the basic arithmetic units are based on Vivado's IP, they should be firstly substituted with completely proprietary customized IP. Then the timing of IPs needs to be re-analyzed and the re-scheduling is expected within the system.

Finally after the extra alteration of IP and system's peripheral circuit, the transplantation is done, switching to a commercial-IP-free version that could be implemented in ASIC.

5.2 Arithmetic IP Configuration

5.2.1 Float-point Adder

The processing flow of proposed float point adder which follows the IEEE-754 float point arithmetic procedure is shown as in figure 5.1 and the architecture details is shown in the figure 5.2.

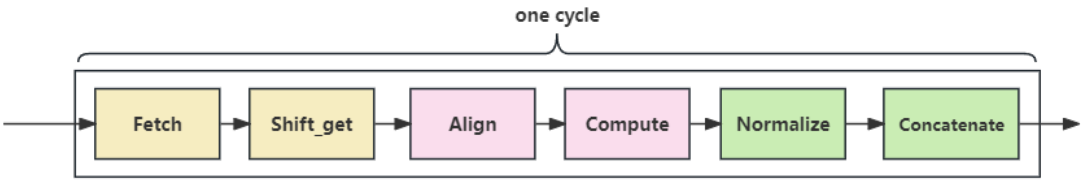


Figure 5.1: Computation flow of Float-point Adder

The whole float add operation could be discomposed to six stages: Fetch, Shift_get, Align, Compute, Normalize and Concatenate, and they are connected in a cascaded way.

Fetch

The standard IEEE-754 single float point number is consist of three parts, the sign bit which is the 31th (MSB) bit, the exponential bit which is 30th to 23th bit, and the fractional bit which is 22th to 0th bit. The sign bit contains the sign information of the number where 1 represent a negative

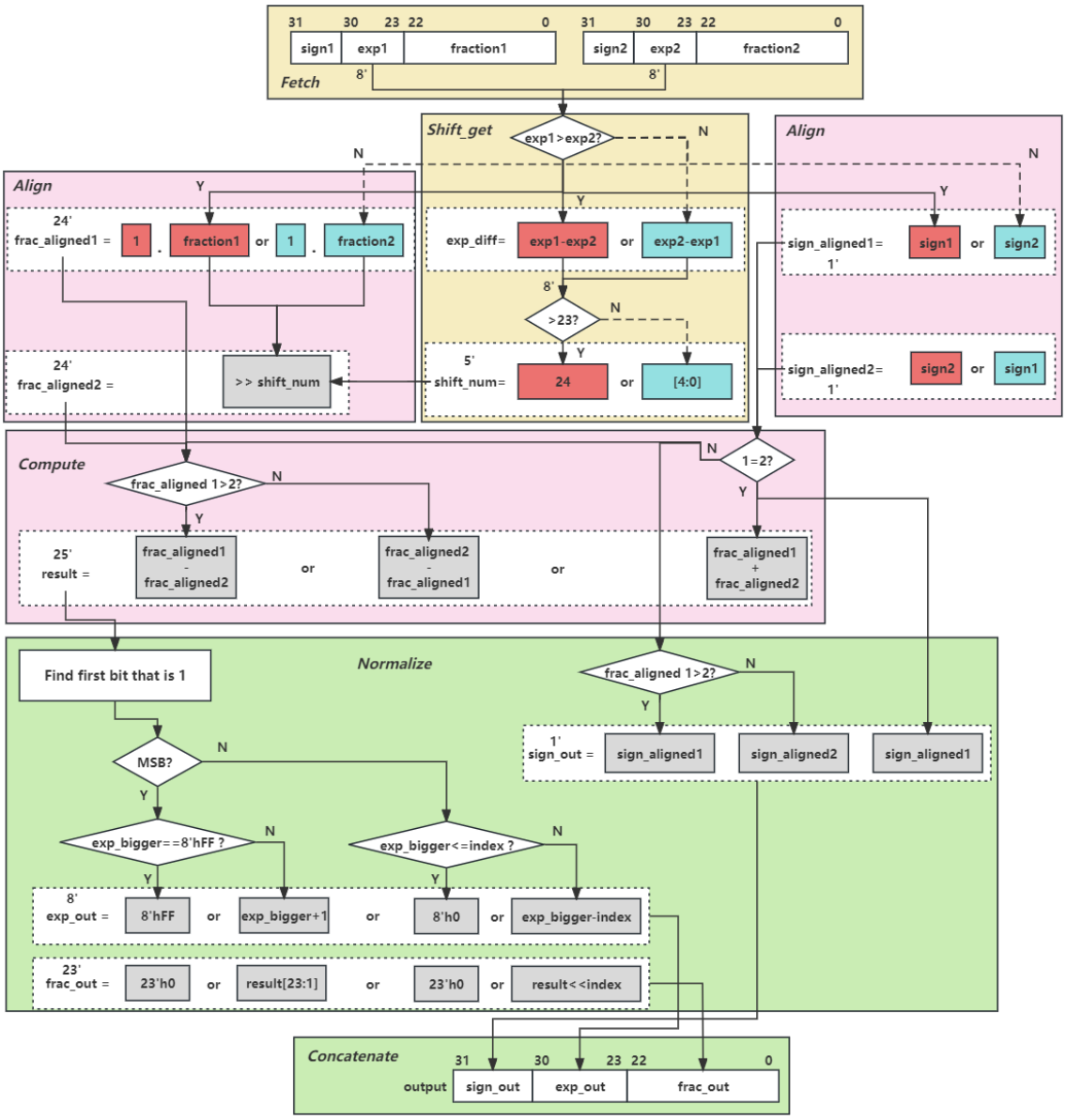


Figure 5.2: Architecture of Float-point Adder

number while 0 indicate a positive one. The exponential bit stores how many bit the fractional part would be shifted to recover the original number, and it is from 0 to 255 for a normalized number (as there is a 127 offset, the real shift number is from -127 to 128). Then the fractional part contains the fractional part of original number, and to save bits, there is always a hidden 1 ahead of the MSB.

As these bits are concatenated together in a float number, these parts of information should be disassembled to be analyzed in the following stages. In the Fetch stage, the two float number would be both analyzed and the sign bit, exponential bit and fractional bit would be decoded from the original 32-bit input in a parallel way. In hardware, a wire connection of pure combination

logic would be assigned to finish this part of job.

Shift_get

Then it comes to the Shift_get part which compute the shift number for the next Align stage. Firstly the exponential bit of two input number will be compared, and a exp_diff signal would record the difference of these bit. Considering if only distract the two number directly, the result could be a negative number, and then it needs to convert the 2-complement code of negative which add an extra stage to the circuit and increase the delay. Then the two cases of which $exp1 - exp2$ and $exp2 - exp1$ are computed in parallel and a muxer would mux them out according to the relative size of two exponential numbers.

After this, as the fractional bit is only have 24 of width if includes the hidden 1, any logic shift operation greater than 24 will all make the final result a 24 bit of zero. However, as the exp_diff is of 8 bit width, in hardware level this would be projected to a shift register that could shift right for $2^7 = 128$ bits in one cycle, which would require a huge amount of resource to be realized, but any shift number great than 24 would have the same result and waste majority of the resource. Then the $shift\_num$ of 5-bit width is established to fetch the low 5 bit in exp_diff, and set its value to 24 if the number is any number greater than 23. This makes the $shift\_num$ ranging from 0 to 24 with all the number represent a unique meaningful shift cases.

Align

Considering although one fractional bit may be greater than another, the shifted original number would be smaller the exponential bit of another one is bigger. Then a Align stage would be needed to align the fractional part of two original number. This is done by right shifting the fractional bit with smaller exponential bit of $shift\_num$ that indicates the difference of two exponential part in Shift_get stage. However, the fractional bit would be firstly extended to 24 bit with the hidden 1, and then store the one with large exponential bit in the $frac\_aligned1$, and shifted smaller one to the $frac\_aligned2$. Also, the corresponding sign bit will also be replaced when the larger and smaller fractional part changed their place.

Compute

Then the fractional part of original number has been aligned and the addition of two number could be operated. However, considering the fractional bit is not in 2's complement format if it is a negative number, the positive one and negative one could not be added together directly to have the final result. So the unit would firstly compared the sign bit of two numbers: if both positive or negative, they could be added together directly as the final result will have the same sign with them; but if they have different sign, the result would have the same sign with the bigger one, and the result is the bigger one subtract the smaller one. Finally considering the two 24-bit number addition could cause a overflow at the MSB, the *result* has been set to 25-bit to store the overflow bit. And the final result's exponential bit would be the bigger exponential bit of the two input number.

Normalize

As the IEEE-754 single float point number stipulates that the fractional part must have a hidden 1 ahead of the MSB of 23-bit fraction bit but the addition result of two number in Compute stage might not fulfill that, the Normalize stage is needed to normalize the output of addition following the representation rules of IEEE. This is done by a priority decoder that finds the first bit of one from MSB to LSB and then adjust the exponential bit of proposed output number.

The *result* in the previous stage could be represented as $xx.xxxxx$ which has two integer bit and 23 fractional bit, and the goal of normalization could then be converted to shifting this number to make the integer part become 01. Then the 23 bit fractional part could directly become the fractional bit of the proposed output result by considering the 1 in the integer part as the hidden 1 in the fractional bit. The the operation could be determined under this methodology: shift the *result* right with 1 bit if it is $1x.xxxxx$ and add the exponential bit of 1; shift the *result* left with *index* number where the *index* is the first bit of 1 from 24th bit to 0th bit and also takes the fractional part as the fractional bit with the hidden 1 in the integer part ahead and subtract the greater exponential bit with *index* value. This would expand the lower bit of fractional bit with all 0, which also choose the direct rounding across the rounding strategy.

However, the overflow in the normalization should also be considered: the IEEE-754 stipulate that any number with exponential bit of 0 and fractional bit also 0 would represent the zero float number; and any one with exponential bit 255 (all 1 for the 8 bits) and fractional bit of 0 would

indicate the ∞ number. But the addition and subtraction in previous operation could make the result flow over to 255 or below 0 and results to a wrong value. Then the protection is set that set all possible overflow result with exponential bit of 255 to the ∞, and any one with exponential bit of 0 to the float zero.

Concatenate

Finally after all previous stages, every components of the output float number has been determined. The last Concatenate stage would concatenate the previous components in sequence to assemble the output number to the output register, and pull up the valid signal, indicating the overall computation is done.

Timing

Originally the Unit is proposed to be finished in one clock cycle. However, when evaluate on FPGA with the targeted 100MHz clock, the Static Timing Analysis (STA) suggest that the module has timing violation of setup time which resulting to a negative Worst Negative Slack (WNS) from input to output. As the module only have one stage of register, this indicates that the combination logic of computing is too long and it could not be finished within one cycle which is 10ns.

Then the Unit is considered to be cut with pipeline stages to optimize the timing which is a general method to repair the setup time violation. Considering to make each pipeline stage with nearly even load and thus delay time, the structure of pipeline is show as in figure 5.3.

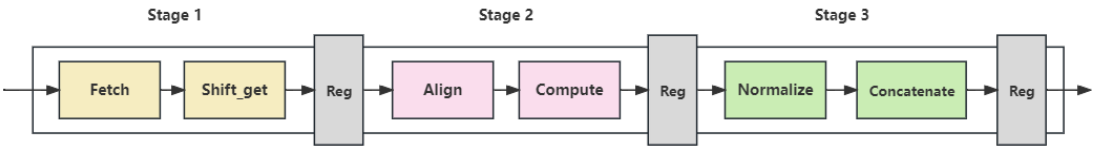


Figure 5.3: Pipeline structure of proposed Float-point Adder

The original path has been cut to three stages: Fetch and Shift_get constitute the first stage; the Align and Compute is of second stage; and the Normalize and Concatenate is for the third stage. Between each stage a register group is inserted to memorize the information of previous state while the inside generates the current stage information. And when applying the golden module rule, all the unit in the system would only have output register, and the input is buffered during the computing process since the input is connected to the output port which is already

in register within other module. After this, the timing of unit has fulfilled the 10ns requirement, resulting a positive WNS of 2.853ns.

Accuracy

In order to test the accuracy of proposed float point adder, the testbench is constructed as shown in figure 5.4.

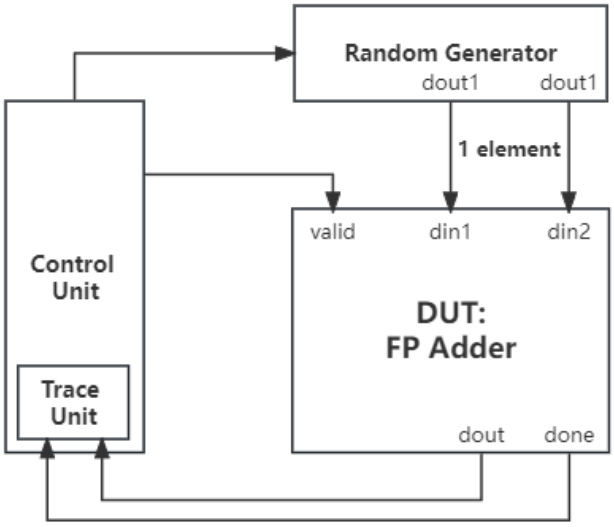


Figure 5.4: Testbench structure of proposed Float-point Adder

A control Unit is instantiated to control all the test process and test boundary check. The trace unit inside would then trace the output of the DUT float point adder and record the result. The random generator is used to generate two random float point input to the DUT as stimulus. Considering the real number inside the system would be float number with absolute value less than 100, the range of generated input has been set to $[-\frac{1M}{101234.5}, +\frac{1M}{101234.5}]$ with minimal resolution of $\frac{1}{101.0} = 9.88\times10^{-6}$. After 20000 number of test cases, the RMSE error compared to C++ IEEE float number of proposed adder is to be 4.33×10^{-7}, fulfilling the accuracy requirement.

5.2.2 Float-point Multiplier

The processing flow of proposed float point multiplier which follows the IEEE-754 float point arithmetic procedure is shown as in figure 5.5 and the architecture details is shown in the figure 5.6.

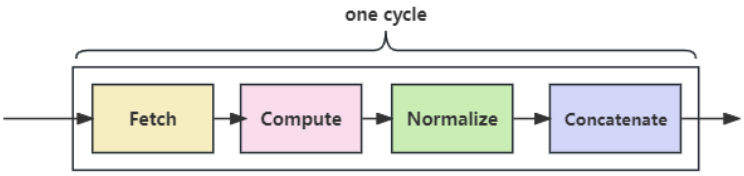


Figure 5.5: Computation flow of Float-point Multiplier

The whole float add operation could be discomposed to six stages: Fetch, Compute, Normalize and Concatenate, and they are connected in a cascaded way.

Fetch

The standard IEEE-754 single float point number is consist of three parts, the sign bit which is the 31th (MSB) bit, the exponential bit which is 30th to 23th bit, and the fractional bit which is 22th to 0th bit. The sign bit contains the sign information of the number where 1 represent a negative number while 0 indicate a positive one. The exponential bit stores how many bit the fractional part would be shifted to recover the original number, and it is from 0 to 255 for a normalized number (as there is a 127 offset, the real shift number is from -127 to 128). Then the fractional part contains the fractional part of original number, and to save bits, there is always a hidden 1 ahead of the MSB.

As these bits are concatenated together in a float number, these parts of information should be disassembled to be analyzed in the following stages. Similar with the Fetch stage of the float point adder, the two float number would be both analyzed and the sign bit, exponential bit and fractional bit would be decoded from the original 32-bit input in a parallel way. In hardware, a wire connection of pure combination logic would be assigned to finish this part of job.

Compute

Considering the float number could be represented as $num = sign \times 1.frac \times 2^{exp-127}$, the multiplication of two float point number could have the result with the form in equation 5.1.

$$\begin{aligned} num_1 \times num_2 &= sign_1 \times 1.frac_1 \times 2^{exp_1-127} \times sign_2 \times 1.frac_2 \times 2^{exp_2-127} \\ &= (sign_1 \times sign_2) \times (1.frac_1 \times 1.frac_2) \times 2^{exp_1+exp_2-127-127} \end{aligned} \tag{5.1}$$

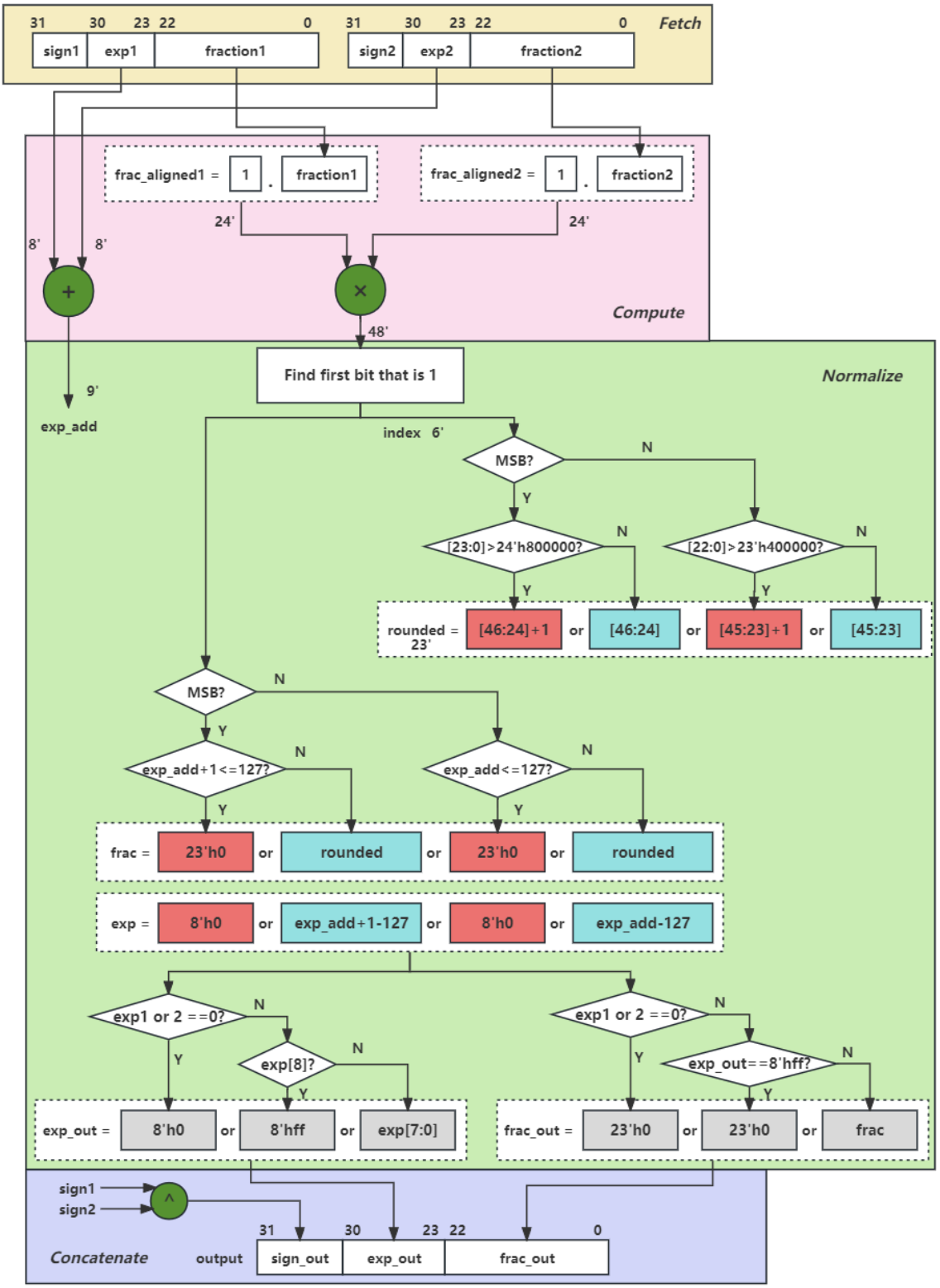


Figure 5.6: Architecture of Float-point Multiplier

The multiplication could be disassembled into three parts: multiplication of sign bit, the extended fractional bit and the exponential bit respectively, and multiplied them together after the computation of smaller part to have the overall result. And in the project this method would be applied to compute the float point multiplication.

Firstly the two fractional part would be extended with the hidden 1 ahead of MSB to have the 24-bit $frac_aligned$. The two $frac_aligned$ would be sent to the integer multiplier to have the 48-bit result. This operation is reasonable since the fixed point multiplication would not change the place of dot; In this case, the multiplication of two 24-bit fixed point number with the dot at the 23th index would have the a result of 48-bit number, and if considering the dot place is at 23+23=46th, the result would be the correct which has two integer bit and 46 fractional bit.

As the term of $2^{exp_1+exp_2-127-127}$ represent the shift operation with $exp_1 + exp_2 - 127 - 127$ bit in fixed point system, the multiplication of this term would results to the replacement of dot in the result but not affect the fractional number. For this reason, the exponential bit of two input would be simply added to calculate the exponential part of the output. And the sign bit would be multiplied which is implemented as $\oplus$ operation in hardware level.

Normalize

As the IEEE-754 single float point number stipulates that the fractional part must have a hidden 1 ahead of the MSB of 23-bit fraction bit but the multiplied result of two number in Compute stage might not fulfill that and have longer bit length, the Normalize stage is needed to normalize the output of addition following the representation rules of IEEE. This is done by a priority decoder that finds the first bit of one from MSB to LSB and then adjust the exponential bit of proposed output number.

After this, the unit would firstly judge the round condition to find the rounding result of the fractional part. As the final fractional part has 24 bits (hidden 1 included) but the multiplied result have 48 bits, it means that 24 of the bits would be deprecated. Then the direct rounding strategy is applied: if the deprecated number is greater than half of its maximum value, the LSB of rounded result would have to add 1 or other wise the quantization error will not have a even distribution across Nth bit to N-1th bit.

Theoretically the rounded result would be the 24 bit of number started from the first 1 bit to LSB. However, as the location of first 1 bit is changing across different number, the truncation

would be hard to realize in the hardware level. Then the unit propose that two conditions are considered: 1) the MSB is one, where the truncation will happend to preserve the 46th to 24th bit and rounding the 23th to 0th bit. 2) the first 1 bit is lower than the 46th bit, and the higher 23 bits from 45th to 23th would be preserved since the bits below 23th is too small to be considered.

Also, the middle $frac$ and exp variable is used to store and update the rounded result. Then judge whether the exponential is too small to be represented (approximate to zero) and set the $frac$ and exp to zero according to IEEE-754 protocol when the add result of exponential is less tha 127-1 (the MSB is the first 1-bit situation, since the fraction would right shift by one bit and add the 1 to exponential. The value would be 127 if the shift is toward left.).

Finally, the unit would judge whether any of the input number is float point zero since any multiplication with zero would have a result of zero. Besides, the overflow control is also done: if the added exponential result is greater than maximum value that 8-bit number could represent, the unit would reset it the maximum value (255, 8'hff) or else the lower 8 bits would be truncated and used as exponential output.

Concatenate

After all previous stages, every components of the output float number has been determined. The last Concatenate stage would concatenate the previous components in sequence to assemble the output number to the output register, and pull up the valid signal, indicating the overall computation is done.

Timing

Considering the complex architecture of multiplication for the float point number, theoretically the unit would have to be cut into pipeline structure. However, the STA tool show that the timing of the unit could fulfill the constraints, indicating all the combination computation logic could be done within one cycle of 100MHz clock. Then the module will not need to be pipelined and have an overall latency of 1 and II=1 with the WNS of 3.534ns and uses no DSP resources.

Accuracy

The testbench of unit has the same structure with the one of float point adder as in the figure 5.4 except the DUT in the system has been changed to the float point multiplier unit.

Considering the real number inside the system would be float number with absolute value less than 100, the range of generated input has been set to $[-\frac{1M}{101234.5}, +\frac{1M}{101234.5}]$ with minimal resolution of $\frac{1}{101.0} = 9.88\times10^{-6}$. After 20000 number of test cases, the RMSE error compared to C++ IEEE float number of proposed multiplier is to be 2.05×10^{-6}, fulfilling the accuracy requirement.

5.2.3 Activation Functions

The Vivdao version RNN System uses CORDIC IP to calculate the $Tanh$ function and exponential IP to compute the $Sigmoid$ function. However, this is a compromise considering these are the only way that existing Vivado IP could finish the operation, and it would consume intense DSP units and other resources while the speed is also not very desirable. Then in this section, as the fully proprietary IP is needed, these functions could be approximated by constructing dedicated CORDIC IP core.

Many prior works have explore the possibility to use CORDIC method to approximate the $Tanh$ and $Sigmoid$ function directly due to its high accuracy and simple structure as well as balanced trade-off between speed and area. Among them, [28]'s work has shown great potential in computing these functions with highly flexible and configurable structure. Firstly they convert the $Tanh$ function into the form of $Sigmoid$ as shown in equation 5.2:

$$Tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 1 - \frac{2}{1 + e^{2x}} = 1 - 2Sigmoid(-2x) \tag{5.2}$$

Then the goal of approximating the two function is converted into compute the $Sigmoid$ function, saving three exponential block in the hardware level. And the uses the RHC and VLC module to compute the $Sigmoid$ function with some alteration for the $Tanh$ function.

CORDIC

In 1959, Volder and Jack E invented the CORDIC algorithm to compute and approximate the trigonometric function, multiplication and division[29]. After that, the Walther and John Stephen further extended the application of CORDIC, enabling them to compute the logarithm, exponential and squared root[30].

The principle of two mode of application of CORDIC, RHC and VLC analyzed as following.

RHC Mode

The iteration functions of RHC mode variable are shown as in equation 5.3 to 5.5.

$$x_{i+1} = x_i + sign(z_i)(2^{-i}y_i) \tag{5.3}$$

$$y_{i+1} = y_i + sign(z_i)(2^{-i}x_i) \tag{5.4}$$

$$z_{i+1} = z_i - sign(z_i)arctanh(2^{-i}) \tag{5.5}$$

where $i \geq 0$ and $i \in \mathbf{Z}$. And the iteration $k = 4, 13, 40, ..., n$ needs one more iteration with the same coefficient otherwise the algorithm would not converge.

The RHC mode uses the rotation of vector on a hyperbolic line to approximate the *sinh* and *cosh* function. Equation 5.3 and 5.4 describe the x and y coordinate of the rotation vector, and z is for the angle difference. Then after several iterations, the three variables would be converged to the value as shown in equation 5.6 to 5.8:

$$x_n = \Gamma(x_0 \cdot cosh(z_0) + y_0 \cdot sinh(z_0)) \tag{5.6}$$

$$y_n = \Gamma(y_0 \cdot cosh(z_0) + x_0 \cdot sinh(z_0)) \tag{5.7}$$

$$z_n = 0 \tag{5.8}$$

where Γ is scale-factor which computed by $\prod_{i=1}^{n}(\sqrt{1 - 2^{-2k}})$ and the term of $i = 4, 13, 40...$ needs to be multiplied twice.

VLC Mode

The iteration functions of RHC mode variable are shown as in equation 5.9 to 5.11.

$$x_{i+1} = x_i \tag{5.9}$$

$$y_{i+1} = y_i - sign(y_i)(2^{-i}x_i) \tag{5.10}$$

$$z_{i+1} = z_i + sign(y_i)2^{-i} \tag{5.11}$$

where $i \geq 0$ and $i \in \mathbf{Z}$.

The VLC mode uses the rotation of vector on a linear line to approximate the linear variables. Equation 5.9 and 5.10 describe the x and y coordinate of the rotation vector, and z is for the angle difference. Then after several iterations, the three variables would be converged to the value as shown in equation 5.12 to 5.14:

$$x_n = x_0 \tag{5.12}$$

$$y_n = 0 \tag{5.13}$$

$$z_n = z_0 + \frac{y_0}{x_0} \tag{5.14}$$

Construction of Activation Function

It is easy to be observed that the Γ is the function of iteration times, and if the iteration number is fixed, the Γ could be considered as a constant coefficient. This make the result of the RHC mode controllable: if deliberately set the initial value of variables to equation 5.15,

$$x_0 = \frac{1}{\Gamma}, y_0 = 0, z_0 = x \tag{5.15}$$

then the output of RHC unit could be converge to equation 5.16.

$$x_n = cosh(x), y_n = sinh(x), z_n = x \tag{5.16}$$

and if deliberately set the initial value of variables of VLC to equation **??**,

$$x_0 = x, y_0 = y, z_0 = 0 \tag{5.17}$$

then the output of VLC unit could be converge to equation 5.18.

$$x_n = x, y_n = 0, z_n = \frac{y}{x} \tag{5.18}$$

Then the $cosh(x)$ and $sinh(x)$ could be computed with RHC Unit, and the VLC Unit could be used as a divider, indicating the $cosh(x)$ and $sinh(x)$ could be divided and results to a $Tanh$. However, as the convergence range of VLC mode is $\frac{y}{x} \leq 1$[31] but the relative size of $cosh(x)$ and $sinh(x)$ is changing across the angle, this method could not be used and need to convert to the variation of $Sigmoid$ function.

But if considering the expression of $Sigmoid$ function which is $\frac{1}{1+e^{-x}}$, as the term e^{-x} is always a positive number, the nominator will always less than the denominator and fulfill the converge requirement. This meaning that the combination of RHC and VLC unit could be used in computing the $Sigmoid$ function by using the equation $cosh(x) + sinh(x) = e^{x}$. Shown as in figure 5.7, if set the input value $z_0 = -x$ in RHC and connect the plus one addition result of x_n and y_n to the x_0 port of VLC unit and set its $y_0 = 1, z_0 = 0$, the output z_n after several iteration could be the $Sigmoid(x)$.

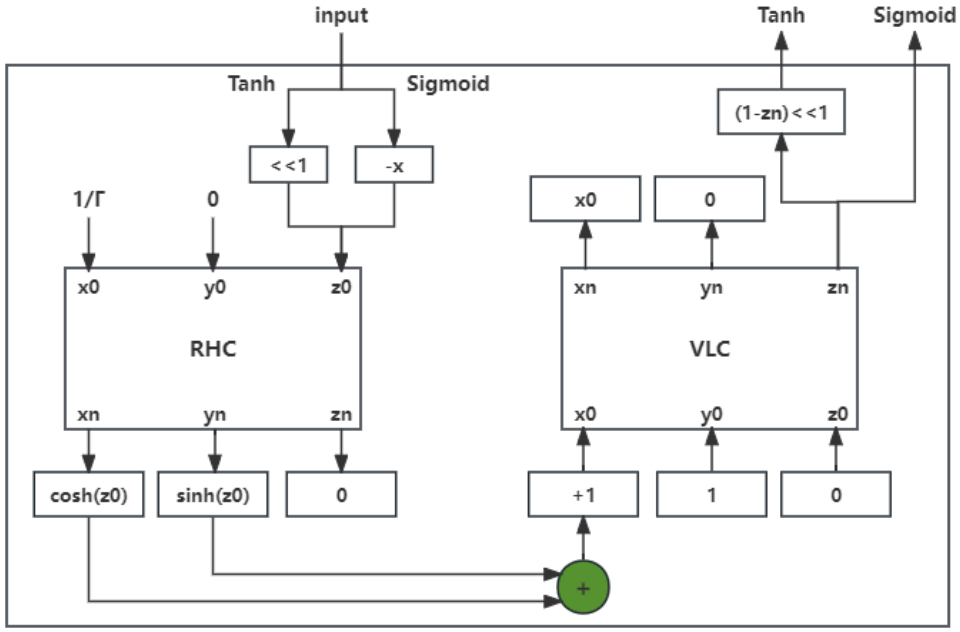


Figure 5.7: Architecture of CORDIC computation

Expanding the Convergence Range

As the analysis in previous section, under the architecture of 5.7, the VLC unit will always converge. However, the RHC unit has its convergence requirement of having $|z_0| \leq 1.1182$[31], which would directly constrain the input range of x to less than 1.1182. Considering the real number that would be used to computed with $Sigmoid$ Unit has majority part that greater than this part, the input

range of *Sigmoid* Unit needs to be expanded to comply with this.

In 1991, X. Hu, R.G. Harber and S.C. Bass discovered a way to expand the convergence range of RHC mode CORDIC to nearly infinite[31]. By expanding the iteration index in the RHC $i = 0, 1, ..., n$ to $i = -m, -m+1, ..., 0, 1, ..., n$, the convergence angle could be extended to 5.19.

$$\theta_{max}^{(m,n)} = \sum_{i=-m}^{0} arctanh(1 - 2^{-2^{-i+1}}) + \sum_{i=1}^{n} arctanh(2^{-i}) \tag{5.19}$$

where the iteration $k = 4, 13, 40, ..., n$ time should be accumulated twice.

The first summation part greatly extended the max angle of input to near infinity: even when $m = 5$, the inside of $arctanh$ would be $1 - 2^{-2^{5+1}} = 1 - 2^{-64} \to 1$ where the $arctanh \to \infty$. The converge range of choosing different m and $n = 14$ is shown as in Table 5.1.

Table 5.1: Convergence range of *Sigmoid* and $Tanh$ function against m value with $n = 14$

m	$\theta_{max}^{(m,14)}$	$Sigmoid(x)$[i]	$Tanh(x)$[i]
0	2.091	[-2.091,2.091]	[-1.045,1.045]
1	3.808	[-3.808,3.808]	[-1.904,1.904]
2	6.926	[-6.926,6.926]	[-3.463,3.463]
3	12.818	[-12.818,12.818]	[-6.409,6.409]
4	24.255	[-24.255,24.255]	[-12.128,12.128]
5	$\to \infty$	$\to \infty$	$\to \infty$

[i] Indicate the input range of target function

Then the iteration equation of the RHC unit is changed to equation 5.20 to 5.22.

$$x_{i+1} = x_i + sign(z_i)((1 - 2^{-2^{-i+1}}) \cdot y_i) \tag{5.20}$$

$$y_{i+1} = y_i + sign(z_i)((1 - 2^{-2^{-i+1}}) \cdot x_i) \tag{5.21}$$

$$z_{i+1} = z_i - sign(z_i)arctanh(1 - 2^{-2^{-i+1}}) \tag{5.22}$$

where $i = -m, -m+1, ..., 0$ and m is a constant. And when i is positive integer number, the expression is the same with equation 5.3 to 5.5.

RHC Hardware Implementation

Even when $m = 4$, the input range of both function could be very high. From a practical point of view and easiness of cutting pipeline, the project finally choose $m = 3$ to compute the activation

function. Also, from [28]'s work as shown in figure , when choosing $m = 0, p = 16$ where p is the iteration time of VLC unit, the RMSE error of proposed unit is below 1×10^{-6}. Considering the small computation error, the project finally choose the $n = 14$ to compute activation functions.

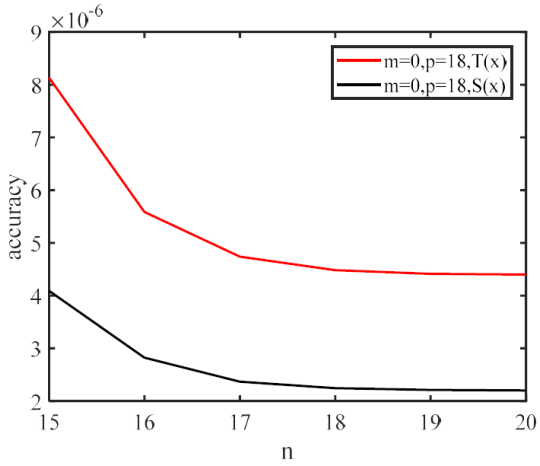


Figure 5.8: Accuracy of CORDIC for $Sigmoid$ and $Tanh$ function against n[28]

Then the architecture of RHC unit is shown as in figure 5.9.

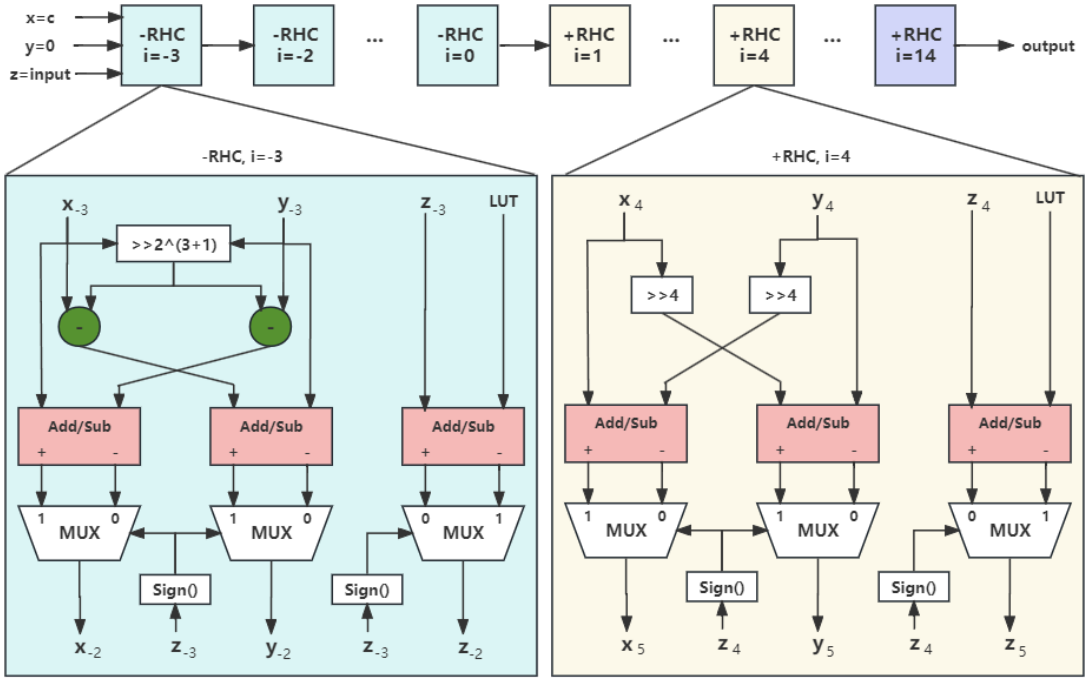


Figure 5.9: Architecture of RHC Unit with $n = 14$, $m = 3$

As the iteration equation is in different form for negative and positive stage of iteration, basic stage unit has two different structure where the "-RHC" unit implements the negative stage ($i = -m, -m+1, ..., 0$) and "+RHC" unit implements the positive stage. As the project has chosen

$m = 3$ and $n = 14$, there are 4 negative stages in total and 16 positive stages as the iteration would be repeated when $i = 4$ and $i = 13$, which has been projected to a repeated +RHC unit with same stage number.

In term of computing data flow, firstly the initial x, y, z are sent to the $i = -3$ stage's input, and their values are selected to be as shown in the equation 5.23 to 5.25.

$$x_{-3} = \frac{1}{\Gamma} = \frac{1}{\prod_{i=1}^{i=14}(\sqrt{1 - 2^{-2i}})} \approx 10754 \tag{5.23}$$

$$y_{-3} = 0 \tag{5.24}$$

$$z_{-3} = input \tag{5.25}$$

Then the $i = -3$ stage -RHC unit would iterate it to have the x_{-2}, y_{-2}, z_{-2} and send it to the $i = -2$ stage. Recursively, until the $i = 0$ stage the negative part of RHC iteration could be done.

The structure of -RHC unit is shown as the blue block in the figure 5.9. The input x and y would be firstly shift right by 2^{3+1} bits (take $i = -3$ as example), and the x and y would subtract shifted number to implement the "$((1 - 2^{-2^{-i+1}}) \cdot x_i/y_i)$" in the iteration equation. Then the results would be cross-subtracted with the input (x subtract y term and vice versa). After this, multiplexer would be used to mux the result to have the output x and y according to the sign of input z_{-3} where this parallel way of processing could help to increase the hardware processing speed.

Finally the z could be updated using the same structure as x and y, excepting the sign has reversed multiplexer channel. Notably, the z would add or subtract with a *arctanh* function of i related term where the none-linear function is hard to implement in hardware. However, as it is only related to i, which is known for every iteration, this part of number could be pre-calculated and be stored in a array and be read out as coefficient. Then the *arctanh* term in the equation 5.22 could be replaced by a LUT number (shown as in Table 5.2) in the substantiation stage of hardware.

The structure of +RHC unit is shown as the yellow block in the figure 5.9. Based on the equation 5.3 and 5.4, the input x and y would be firstly right shift by the stage number, and be cross-subtracted with the input (x subtract y term and vice versa). Then the same parallelism has been applied as in the -RHC unit: a multiplex would mux out the result of cross-subtraction based

Table 5.2: LUT value for RHC negative stage when $m = 3$

i	$arctanh(1 - 2^{-2^{-i+1}})$	Quantization[i]
3	5.891747220047717	49423557
2	3.118184795101852	26157229
1	1.716993602242573	18403186
0	0.972955074527656	8161738

[i] Quantization is done by converting the float point coefficient to fixed point number with 23-bit fractional bit and shown in the decimal format (equivalent to right shift by 23 bits)

on the sign of input z.

Similarly, the *arctanh* term in the positive iteration has also been pre-calculated and stored in a LUT which would be used to add or subtract with input z and be muxed to have output z. The stored value is shown in Table 5.3.

Table 5.3: LUT value for RHC positive stage when $n = 18$

i	$arctanh(2^{-i})$	Quantization[i]
1	0.549306144334055	4607913
2	0.255412811882995	2142557
3	0.125657214140453	1054089
4	0.062581571477003	524972
5	0.031260178490667	262229
6	0.015626271752052	131082
7	0.007812658951540	65537
8	0.003906269868397	32768
9	0.001953127483533	16384
10	0.000976562810441	8192
11	0.000488281288805	4096
12	0.000244140629851	2048
13	0.000122070313106	1024
14	0.000061035156326	512
15	0.000030517578134	256
16	0.000015258789064	128
17	0.000007629394531	64
18	0.000003814697266	32

[i] Quantization is done by converting the float point coefficient to fixed point number with 23-bit fractional bit and shown in the decimal format (equivalent to right shift by 23 bits)

VLC Hardware Implementation

The architecture of VLC unit is shown as in figure 5.10.

As the iteration equation of VLC unit is uniform across $i = 0$ to $i = p$, the basic stage unit has the same structure. Since the project has chosen $p = 16$, there are 16 stages in the VLC unit and no repeat iteration number according to the recursive function.

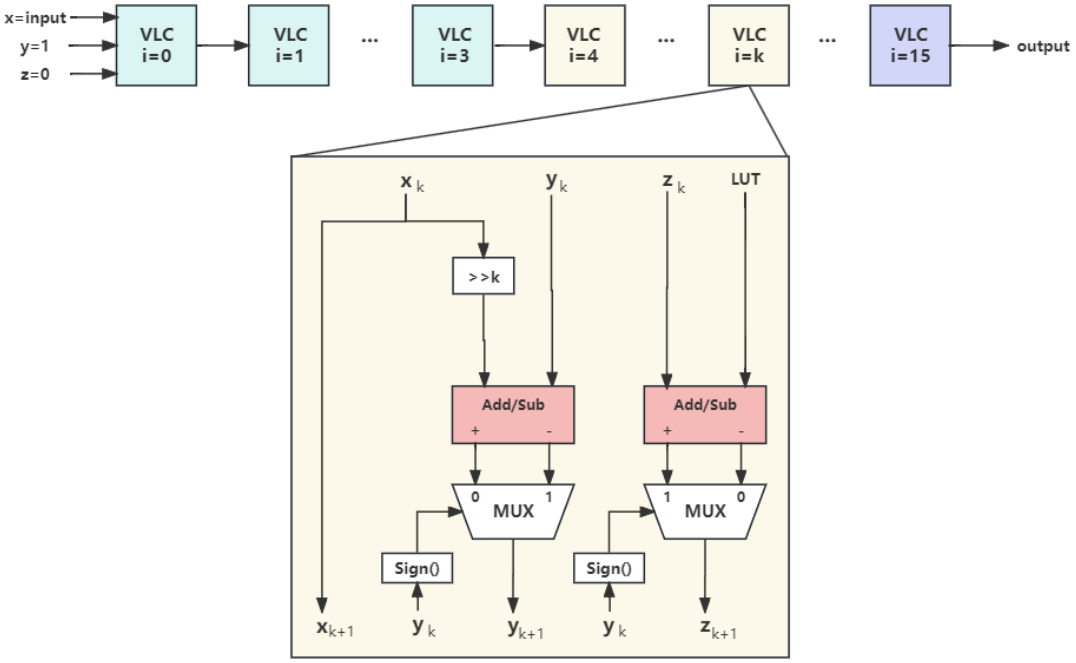


Figure 5.10: Architecture of VLC Unit with $p = 16$

In term of computing data flow, firstly the initial x, y, z are sent to the $i = -3$ stage's input, and their values are selected to be $input, 1, 0$ respectively to operate as a divider.Then the $i = 0$ stage VLC unit would iterate it to have the x_1, y_1, z_1 and send it to the $i = 1$ stage. Recursively, until the $i = 15$ stage the full iteration of RHC could be done and output the divided result.

The structure of VLC unit is shown as the yellow block in the figure 5.10. The input x would be firstly shift right by k bits for k stage. Then the result would be add and subtract with input y in parallel, where a multiplexer would be used to mux the result to have the output y according to the sign of input y_k. And the output x would be the same as the input x.

Finally the z could be updated using the same structure as y, excepting the sign has reversed multiplexer channel. Notably, the z would add or subtract with the 2^{-i} term. This could be both implemented with the constant number shifter or the LUT method as in the RHC mode. Considering the hardware efficiency, the constant number shifter method has been applied since it do not need an extra LUT to store the coefficient and the shifted constant shifted number could be compiled as constant logic value when synthesis, which would not waste the hardware resource to synthesis a physical shifter.

Quantization

Theoretically, all the computation of RHC and VLC are using the single float point number format in default. However, the addition and subtraction inside the basic unit has the requirement of instantiating the float point add IP, which has a latency of 3 as built in the previous section. And this would make the individual latency for each basic unit to 3 cycles, making the overall of RHC or VLC to 60 and 48 cycles which is unacceptable in a practical way.

To solve this, the computation within the RHC and VLC is considered to quantize to fixed point number. Considering the IEEE-754 single float point number has 23 bit of fractional part, any wider fractional part for the fixed point number would not increase the computation accuracy since they will be always added to be 0. Then the project proposed to use 23-bit fractional fixed point to quantize the computation.

Along with the quantization of in-process computation variables, the values in the LUT of *arctanh* term of RHC should also be quantized and the after result is shown as in the right column in Table 5.3 and 5.2. Considering the integer part of fixed point number should be able to store the processing result of exponential, then the integer width of negative RHC is set to 16 and the one of positive to 8 bits respectively, which constrains the input range of *Sigmoid* to have integer part no greater than 8 and $Tanh$ of 4.

Float to Fixed Point Converter

As the computation within the CORDIC module has now been all set to fixed point, there would need the float to fixed and fixed to float point converter since the input and output of the CORDIC unit is in the IEEE-754 single float point format which has 32-bit of width.

The structure of float to fixed point converter that convert the single point float number to fixed point number of DWIDTH bits is shown as in figure 5.11.

Firstly the converter would compare whether the input float number is in format of zero, and if it is, the unit would output WIDTH bit of 0 as the result. And if not, the unit would start the procedure to convert to the fixed point number. The exponential part would then be extracted to judge its relative value (the exponential bit minus 127), and if it is positive, the extended fractional part, which has its hidden 1 added and extra padding bits of 0 added, would need to be shift left with the difference of exponential and 127 to recover the original number and vice versa. As the

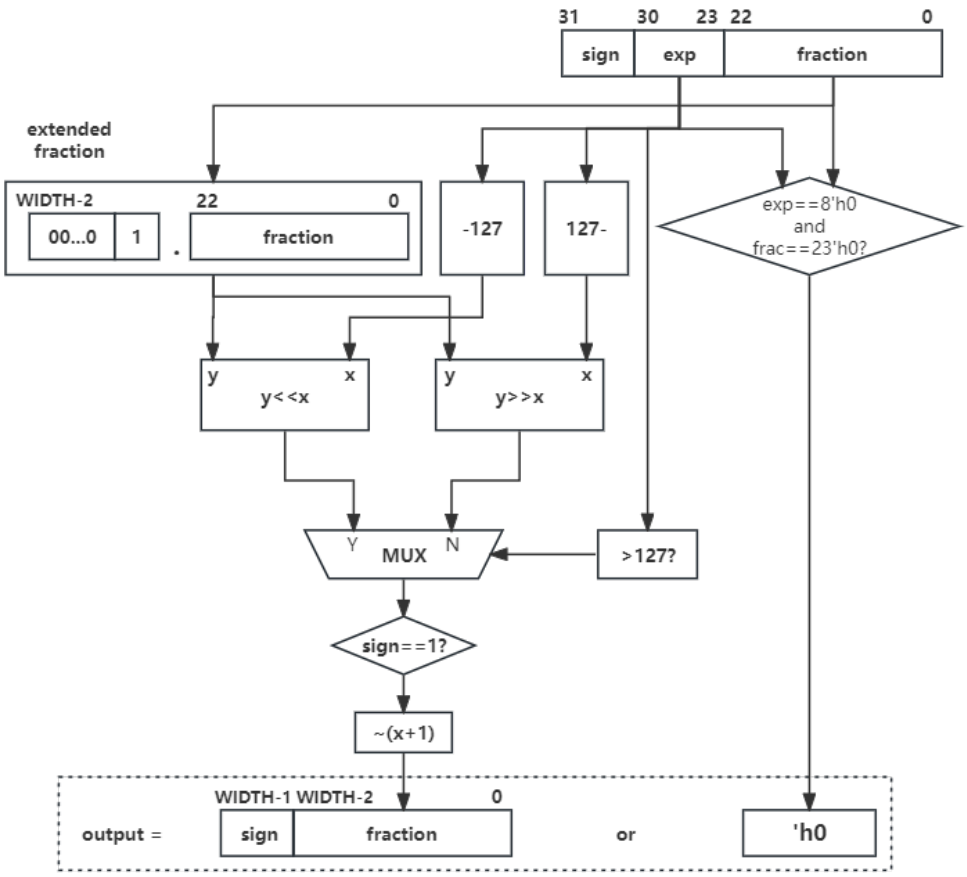


Figure 5.11: Structure of float to fixed point (DWIDTH bit wide) number converter

fractional part of fixed point number is stipulated to be 23-bit width, the width of extended extra padding bits would be WIDTH-23-1.

And considering the computation is avoiding any variable in 2's complement format as it would make the shift operation has negative possibility which would results to a much more complicated hardware structure, the relative size of exponential bit and 127 is firstly compared and the difference would be the bigger one minus smaller one, ensuring the result would be in the positive number format. As the negative number is not in 2's complement format in the float point number, it would need to convert to this format since the fixed point number fully utilizes this mechanism to operate addition and subtraction of negative number. Finally after concatenating with the sign bit, the result of converted fixed point number of WIDTH would be output.

In term of control signal, the unit valid signal would be pulled up and and output register would be updated. As the timing has fulfill the requirement of 100MHz, the unit has a latency of 1 and II=1.

Fixed to Float Point Converter

The structure of fixed to float point converter that convert the fixed point number of DWIDTH bits to single point float number is shown as in figure 5.12.

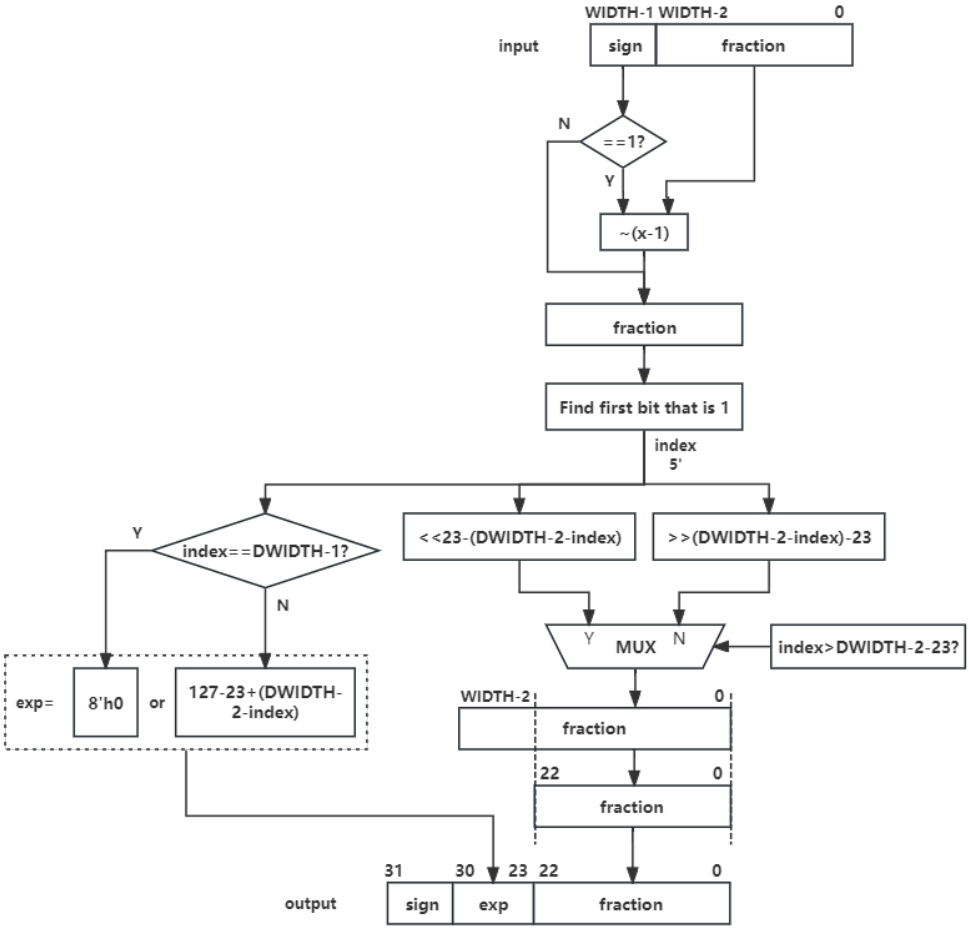


Figure 5.12: Structure of fixed (DWIDTH bit wide) to float point number converter

As the fixed point number is in 2's complement format, firstly the converter would compare the sign bit of the input number. If it is a negative number whose sign bit is 1, the converter would subtract 1 to 1 and reverse bit-wise to convert 2'complement code to true form. Then the unit would find the first bit that is one from its MSB to LSB as the IEEE protocol need to have the fractional bit have a hidden one ahead of them. Then the index would be used to compute the shift number of fractional part to align them to 23 bits.

Similarly, considering the computation is avoiding any variable in 2's complement format as it would make the shift operation has negative possibility which would results to a unnecessary complicated hardware structure, the relative size of *index* and WIDTH-2-23 is firstly compared

and the difference would be the bigger one minus smaller one, ensuring the result of shift number would be in the positive number format.

Then the unit could shift the fractional part to align its dot to the 23th place. Since the WIDTH-2-23 represent the relative distance of first 1 bit and the MSB, a higher value means this bit is away from MSB and is near the lower bit and vice versa. Then for the fractional part of float point number, if it is *index* is greater than this value, it means the index of first 1 bit is less than 23th and close to LSB if it is great and the number needs to shift left 23-(WIDTH-2-*index*) to align it to the 23th bit where the padding of zero is applied. And if index is smaller, the fraction needs to shift right as the actual dot place is greater than 23th bit where the deprecation of lower bits is happened. After this, the fractional bit of the output could be the lower 23 bit of the shifted results.

In term of exponential bit, the converter would detect whether the input has no bit of one (all zero) and set the exponential bit to all zero to make the final float point is zero under the IEEE protocol (the fractional bit should also be all zero and it is already done as the shift number would shift left 24 bits and padding zero to all the 23 lower bits which results to the final fraction part to be all zero). If the input number is not a zero, then it would compensate the shifted distance into the exponential bit. After added a offset of 127 and convert the exponential part to a range of 0-255, the exponential bit could be output and concatenate with the fractional and sign bit.

In term of control signal, the unit valid signal would be pulled up and and output register would be updated. As the timing has fulfill the requirement of 100MHz, the unit has a latency of 1 and II=1.

Then after all the trade-off, the input range of $Sigmoid$ operation has been set to no greater than 8 and the $Tanh$ has been set to 4. Any input value greater than this would be set to the infinite value of the function (for example the input 20 of $Sigmoid$ would have a result of 1.0). This is reasonable since the $1 - Tanh(4) \approx 0.0006707$ and $1 - Sigmoid(8) \approx 0.0003354$, indicating the introduced error of this is at a 1×10^{-4} level at maximum, and would be quickly converge to 0.

Timing

As the RHC and VLC are pure combination logic and so does its peripheral circuit (excepting the float to fixed and fixed to float converter), ideally it could be made finished in one cycle. However,

the STA shows that there would be serious negative time slack when applying 100MHz clock. Then both the of the RHC and VLC unit would have to be made in to pipeline structure and cut the critical path with register.

The pipeline structure of RHC unit is shown as in figure 5.13. Considering the convenience of cutting the system, every four stage of iteration has been grouped to one pipeline stage. As for the positive iteration, the stage of 4 and 13 needs to be repeated twice and they are proposed to be realized by instantiating a new +RHC unit. Then the RHC unit would have 20 stages in total where they have been divided into five groups as shown in the figure.

Also, register groups are inserted between every stages to memorized the buffered data of previous stages and keep the current stage and thus the pipeline running. Then STA tool shows that when computing each stage with the 100MHz clock, the timing would be fulfilled and there would be no negative time slack. As the pipeline stage is 5, the overall latency for the RHC unit would be 5 cycles and the II would be 1 since the fully pipeline structure is used.

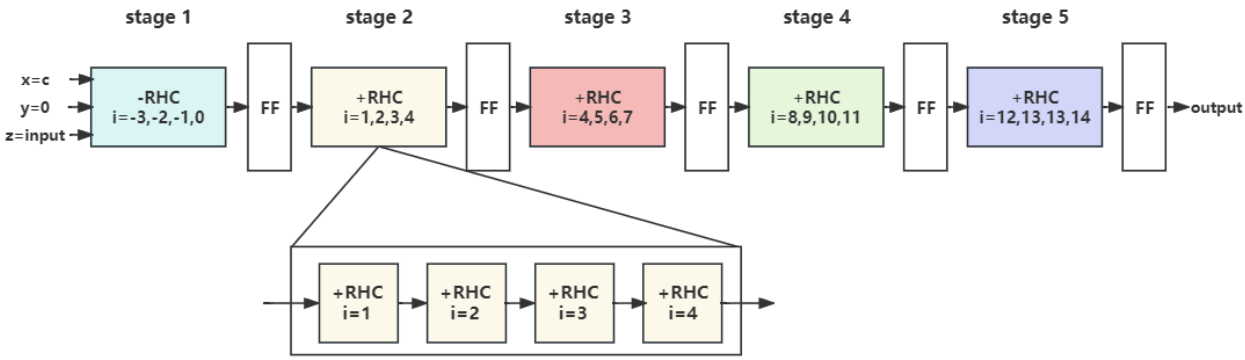


Figure 5.13: Pipeline structure of proposed RHC unit

The pipeline structure of VLC unit is shown as in figure 5.14. Similarly, considering the convenience of cutting the system, every four stage of iteration has been grouped to one pipeline stage. As there is no iterations needed to be repeated, the total 16 stage has been divided into 4 groups as shown in the figure and each group would be finished in one clock cycle.

Also, register groups are inserted between every stages to memorized the buffered data of previous stages and keep the current stage and thus the pipeline running. Then STA tool shows that when computing each stage with the 100MHz clock, the timing would be fulfilled and there would be no negative time slack. As the pipeline stage is 4, the overall latency for the RHC unit would be 4 cycles and the II would be 1 since the fully pipeline structure is used.

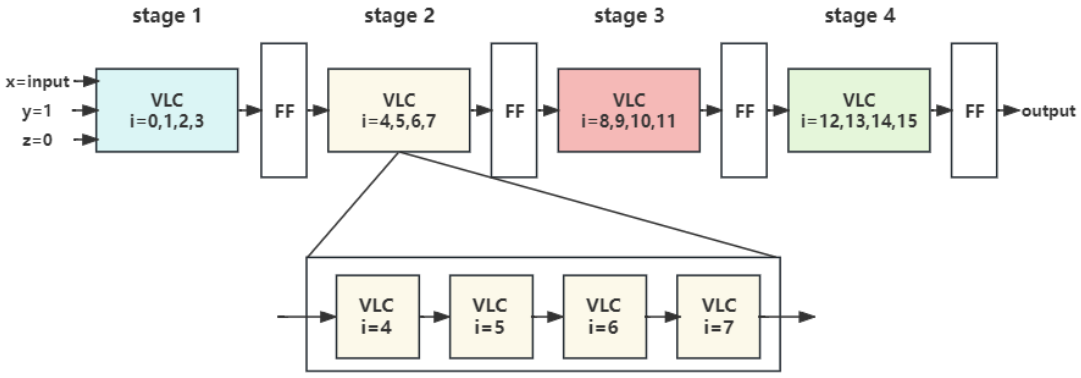


Figure 5.14: Pipeline structure of proposed VLC unit

Then the timing of overall CORDIC unit is shown as in figure 5.15. As the float to fixed point converter in input stage and the fixed to float convert in the output stage will both need one cycle to be finished, the overall module latency would be 1+5+4+1=11 cycles. And the done signal of previous stage is used as the valid signal of current stage to realize auto-handshake during the computation to ensure the validity of the processing data. Since the unit is of full-pipelined structure, the overall II of unit is equals to 1.

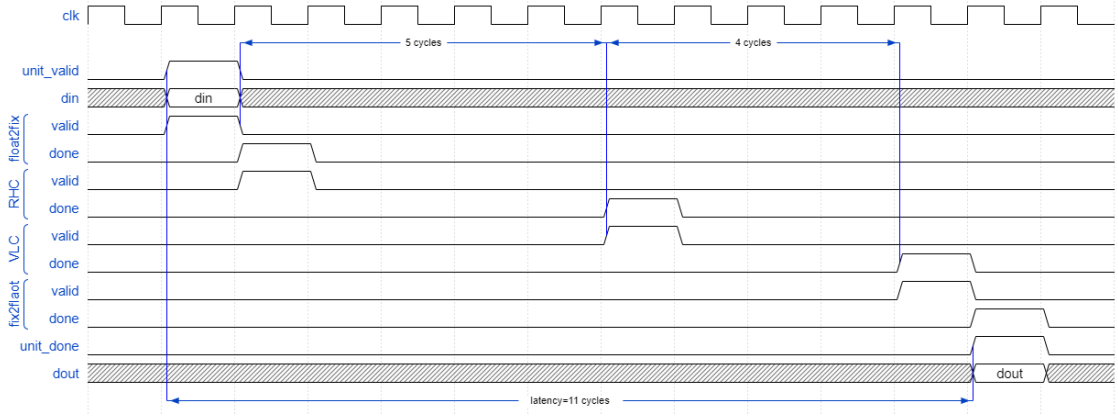


Figure 5.15: Timing of CORDIC unit

Accuracy

The testbench of unit has the same structure with the one of float point adder as in the figure 5.4 except the DUT in the system has changed to the CORDIC unit and control unit will also control the unit to whether operating *Sigmoid* function or the *Tanh*.

Then for the input random float point number from -8.0 to 8.0, the error of unit in *Sigmoid* mode would be less than 1×10^{-4}, and the outsider number would have a error of 3.354×10^{-4} at

most. For the random float point number from -4.0 to 4.0, the error of unit in $Tanh$ mode would also be less than 1×10^{-4}, and the outsider number would have a error of 6.707×10^{-4} at most. Finally the output error of the unit would be at around 10^{-4} level, which is acceptable since the CORDIC algorithm itself with $n = 14, m = 3, p = 16$ has an intrinsic error of 10^{-4} level, and this expanding method has not exaggerate the situation.

5.2.4 Linear Unit

Linear 1x96 Unit

As only the basic IP of float point multiplier and adder has changed to customized version, the overall framework of computing Linear 1x96 Unit need not to change and it still the architecture shown as in figure 4.2. However, since the float point adder has increased latency from 1 to 3 cycles, the overall latency for the linear module has increased to 5 cycles as shown in the figure 4.4.

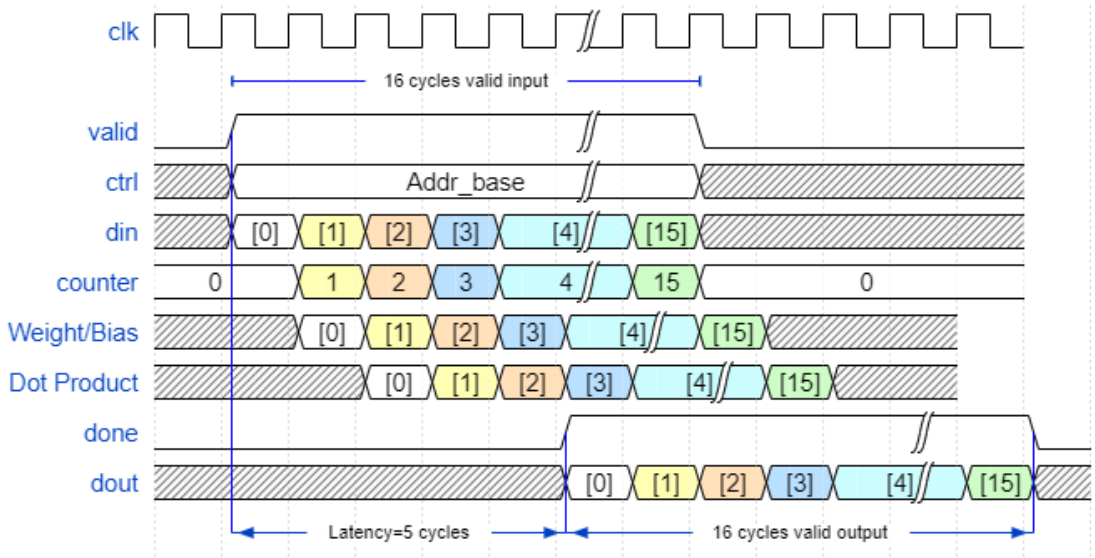


Figure 5.16: Timing of Linear 1x96 unit with customized arithmetic units

With the optimization of configuring the memory array, the value could be read-out just after the counter has updated while keeping its original architecture and storing array. As the cascaded module are using auto-handshake methodology, the inside control signals are not needed to be changed and could automatically adopt to the new version. Since the customized float point add and multiply unit is of full pipelined structure with II=1, the overall Linear 1x96 Unit is also fully pipelined with II=1.

The simulation result of the unit is shown as in figure A.1 where the stimulus are all 1-sized value of 0.7 and weight and bias are the same parameter of GRU layer1 W related matrix. The unit result has been compared with C++ simulation result and showed a good match while the module latency is tested to be 5 cycles with II=1, indicating the unit is functioning correctly.

Linear 32x96 Unit

Similarly, the previous framework of Linear 32x96 Unit could also be used with some alteration regarding the float point adder, and its timing is shown as in figure 5.17.

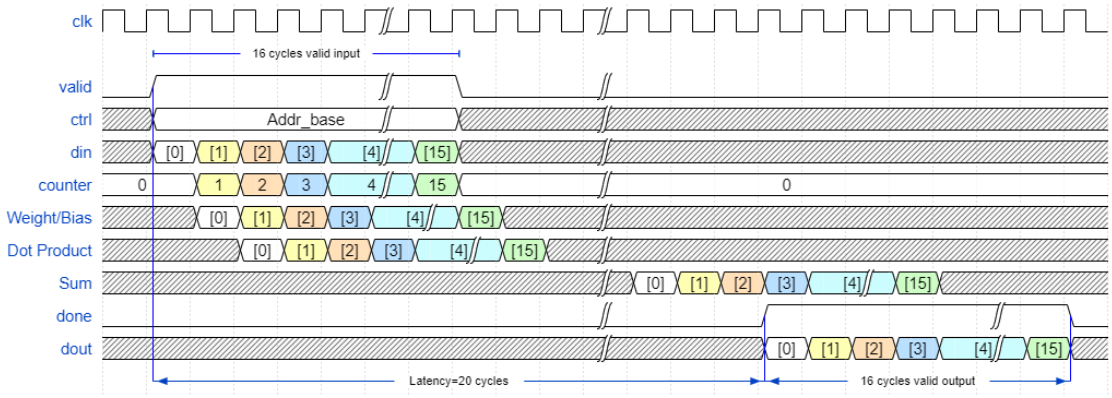


Figure 5.17: Timing of Linear 32x96 unit with customized arithmetic units

Similarly with the optimization of configuring the memory array, the value could be read-out just after the counter has updated while keeping its original architecture and storing array. And the cascaded unit are also auto-handshaking since the same control methodology has applied. However, since the float point adder has increased latency from 1 to 3 cycles, the overall latency for the linear module has increased 6*(3-1) cycles since the increased latency of 5-layer add tree and bias adder cannot be hided. Besides, as the customized float point add and multiply unit is of full pipelined structure with II=1, the overall Linear 32x96 Unit is also fully pipelined with II=1.

The simulation result of the unit is shown as in figure A.2, A.3 and A.4 where the stimulus are all 1-sized value of 0.7 and is weight and bias are used for W matrix computation in GRU layer2, U matrix computation in GRU layer1 and layer2 respectively. The unit result has been compared with C++ simulation result and showed a good match while the module latency is tested to be 20 cycles with II=1, indicating the unit is functioning correctly.

5.2.5 Dense Unit

As the Dense Unit reuses the Linear Unit inside the Linear 1x96 Unit and Linear 32x96 Unit, no control logic is needed to be modified since the valid and done signal of these Linear Unit are capsuled into a black-box where the outside only need to input 16 cycles of valid and wait to receive the done signal and corresponding data.

5.3 System Temporal Scheduling and IP Control

In previous section, the required customized IPs have been constructed and tested to fulfill the requirement. Although their functions are the same with the Vivado IP version, their delay has changed and therefore the previous scheduling on both GRU Unit and System level are no longer valid and needs to be re-constructed. However, since the basic controlling logic are similar, the overall architecture of previous RNN network could remained and still has IPs running in their place.

5.3.1 GRU Unit Scheduling and Control

The required IPs for the construction of GRU Unit are shown as in Table 5.4.

Table 5.4: Timing of system IPs for customized version

IP (Dual channel)	Latency (cycles)	II (cycles)	I/O Timing
Float-point Add	3	1	N[i]cycle high
Float-point Multiply	1	1	N cycle high
Linear 1x96	5	1	16[ii]cycle high
Linear 32x96	20	1	16 cycle high
Sigmoid	11	1	N cycle high
Tanh	11	1	N cycle high

[i] N could be any number of cycle.
[ii] Regulated 16 cycles is for a batch of computation (32 element output), and M batch of input should have M*16 cycle of high *valid*.

Similarly, the Combination Arithmetic Unit would be dissembled to the combination of float point adder and multiplier. Firstly the adder would compute $1 + (-z)$, then two multiplier would have the $z * h\_tm$ and $(1 - z) * hh$ and finally an adder would add two product together to have the updated h_t.

As the C++ computation model remains the same, the critical path of GRU layer is still the hh path. And the r path is arranged to be the first priority. Then the linear module would firstly

compute the x_r and re_r which is used for r computation, and then compute x_h and re_h to generate the hh value while x_z and re_z which is used for z generation. Finally the Combination Arithmetic Unit would update the h_t vector.

Under this methodology, the IP scheduling within the GRU could be drafted as in figure 5.18 (take GRU layer1 as example) where one Linear 32x96 Unit and 1x96 Unit, three float point adder, two float point multiplier and one Sigmoid and Tanh Unit has been instantiated to have a balance between performance and hardware resources.

The general idea of scheduling it to prioritize the computation of critical path. And since the critical path remains the same with the previous version, the computation order is also remains the same. The r related x_r and re_r would be computed firstly, and then the x_h and re_h and finally the x_z and re_z. Since Linear 1x96 Unit has a latency of 5 cycles but Linear 32x96 Unit has 20, Linear 32x96 Unit would be started ahead at clock cycle 1 and then the Linear 1x96 Unit would be started after 15 cycles, which is at cycle 16, to align the x_r and re_r output whose first result is valid at clock cycle 21. As all IPs are constructed in a dual-channel way, 16 cycles of input is enough for the "_r" computation. Then it comes with the choice of whether stop the Linear Unit or keep it running to calculate other data, and the answer of this depends on the following data dependency and would be rearranged after alignment.

After this, the first float point adder (light green) would starts to add the output of two Linear Unit at the cycle of 21. Then after 3 cycle, the output of the first adder would be ready at the cycle of 24. The scheduling of Sigmoid unit apply the same strategy which aligns its output with the result of second round of computation for re_h in Linear 32x96. However, as the previous float point add has two more cycles of latency, the Sigmoid Unit would only need to have 2 cycles of delay. Considering to preserve the results of 1st adder in case they are deprecated in the full pipeline structure, a shift register with depth of 2 would be instantiated as in figure 4.18 to store the output of adder and after 2 cycles of shifting whose result would be sent to the input of Sigmoid Unit.

Then the output of Sigmoid Unit would connect to the first float point multiplier and multiply with aligned re_h to have the r. Then the r would be added with x_h to generate the input of Tanh Unit. Similarly, the x_h and re_h are aligned together whose first output is valid at cycle of 37, but x_h would be needed at cycle of 38 as the latency of multiplier is 1. Then another shift register with depth of 1 is needed to delay the x_h and align it to the output of first multiplier. Besides, as the calculation of Tanh's input overlaps the caculation of Sigmoid input on working

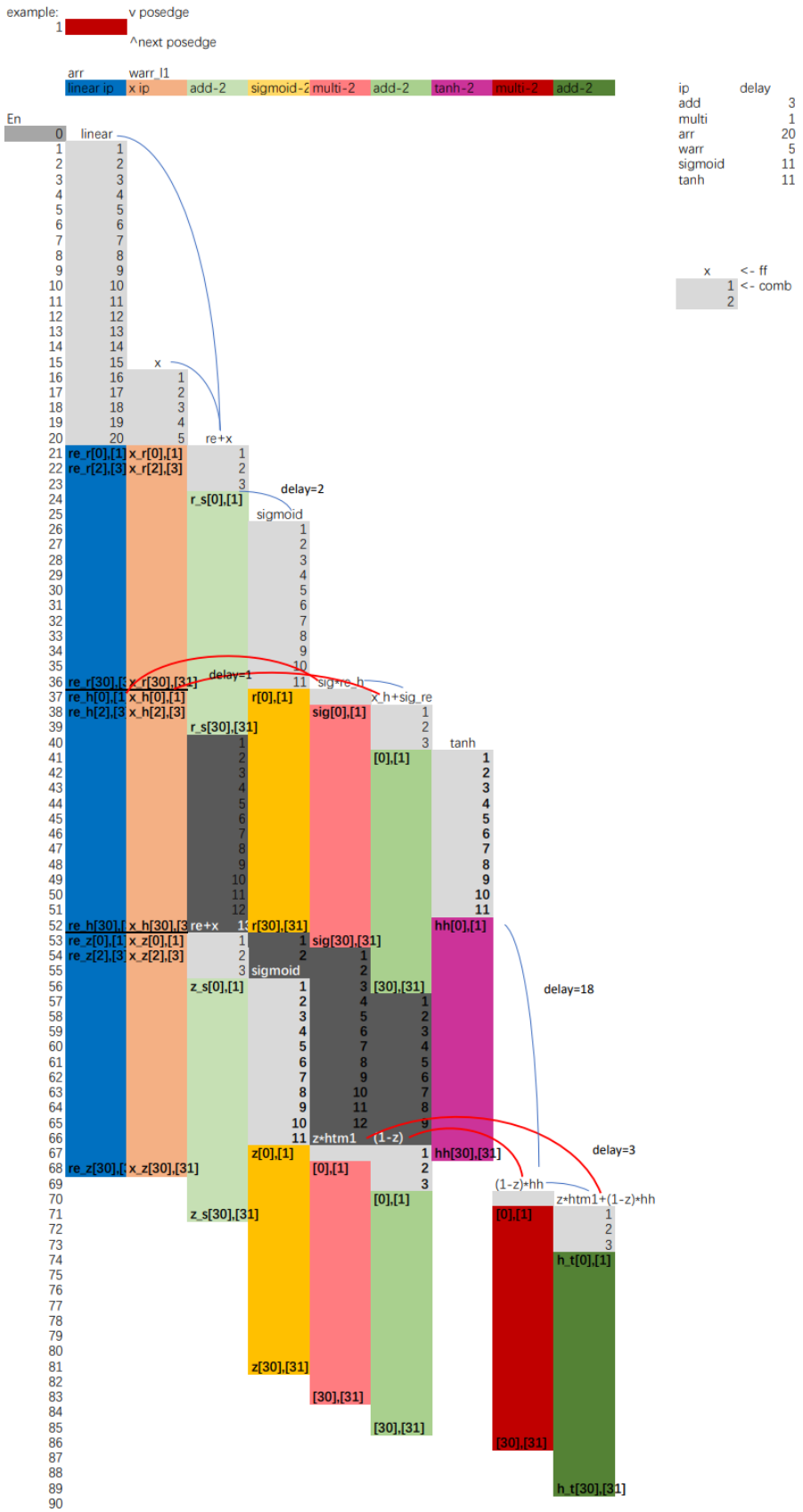


Figure 5.18: Customized version system timing schedule of arithmetic IPs where blue pool indicate the timing of Linear 32x96 Unit, orange for Linear 1x96 Unit, green for the three float point adder, red for the two float point multiplier and yellow for the Sigmoid unit and purple for the Tanh Unit

cycle, they cannot be scheduled to have hardware reuse and two adder unit is needed.

After 11 cycles of delay, the first valid output of Tanh Unit would be at cycle of 52, which is much ahead compared to Vivado version. This makes the system to be rescheduled considering when the first output of Tanh Unit comes out, the Linear Units are still computing the second round of re_h and x_h. This requires the output of Tanh Unit to be delayed until the $1-z$ term is ready to be computed. To accelerate the computation of $1-z$, its origin re_z and x_z needs to be computed as soon as possible. Then the Linear module is proposed to continuely run the third round of computation without stall and re-start. As the third round result would come out at cycle of 53, the 1st adder would have these as input and compute the result to output with delay of 3 at cycle of 56. Then the Sigmoid Unit would takes the output of 1st adder without any delay.

In this way, the z, which is the output of Sigmoid Unit would be ready from the cycle of 67. Then the 1st multiplier and 2nd adder would take this value and starting computing $z * htm1$ and $1-z$ respectively. Since multiplier and adder have different delay and the result of 2nd adder is firstly needed, the 2nd multiplier would immediately start the computation after 2nd adder is done at the cycle of 70. This makes the output of Tanh to delay 18 cycles which is implemented with a shift register with depth of 18. After this, the 3rd adder would add the result of 1st and 2nd multiplier together to have the final result. However, as the output of 1st multiplier is 3 cycles ahead of the one of 2nd multiplier, it would need a 3-depth shift register to delay it for 3 cycles.

Control

As the scheduling of IPs in previous section is accurate in cycle-level and there is multiple state to control, a FSM is considered to use to achieve this. The control information of these IPs is shown as in Table 5.5

And for the convenience of transplanting the system, the start and end time of IPs could be defined with parameters of delay as following:

```
1  `define MEM_DELAY       1
2  `define ADD_DELAY       3//1
3  `define MULTIPLY_DELAY  1
4
5  // only for warr_l1
6  `define WARR_L1_DELAY   (`MEM_DELAY+`MULTIPLY_DELAY+`ADD_DELAY)
7  // for uarr_l1, warr_l2, uarr_l2
8  `define ARR_DELAY       (`MEM_DELAY+`MULTIPLY_DELAY+`ADD_DELAY*(5+1))
```

Table 5.5: Control information of system IPs

IP	Start Time	End Time	Source
Linear 1x96	15	63	*input*
Linear 32x96	0	48	h_tm
1st Adder[i,ii]	20 52	36 68	Linear 1x96, Linear 32x96
2nd Adder	37 66	53 82	1st Multiplier, Linear 1x96[iii] 1.0, Sigmoid
3rd Adder	70	86	1st Multiplier[iv]2nd Multiplier
1st Multiplier	36 66	52 82	Sigmoid, Linear 32x96 h_tm, Sigmoid
2nd Multiplier	69	85	Tanh,[v] 2nd Adder
Sigmoid Unit	25 55	41 71	1st Adder[vi]
Tanh Unit	40	56	2nd Adder

[i] Second line indicate the second round start and end time of unit
[ii] The float point multiplier and adder has two oprand so they have two input sources.
[iii] Need 1 cycles of delay during first round operation
[iv] Need 3 cycles of delay
[v] Need 18 cycles of delay
[vi] Need 2 cycles of delay during first round operation

```
9
10  //`define DIV_DELAY       5
11  //`define EXP_DELAY       4
12  `define SIGMOID_DELAY   11//(`EXP_DELAY+`ADD_DELAY+`DIV_DELAY)
13  `define TANH_DELAY      11//(40+`DIV_DELAY)
14
15  // defined timing point for ff update.
16  // If for combinational logic, add 1 latency to all point.
17  `define ARR_START_1             0
18  `define ARR_END_1               (`ARR_START_1+48)
19
20  `define WARR_L1_START_1         (`ARR_DELAY-`WARR_L1_DELAY)
21  `define WARR_L1_END_1           (`WARR_L1_START_1+48)
22
23  `define ARR_CHANGE_ADDR         (`ARR_START_1+32)
24  `define WARR_L1_CHANGE_ADDR     (`WARR_L1_START_1+32)
25
26  //depends on warr and uarr, suppose delay: uarr >= warr
27  `define ADD_UNIT1_START_1       (`ARR_START_1+`ARR_DELAY)
28  `define ADD_UNIT1_END_1         (`ADD_UNIT1_START_1+16)
29
30  //depends on add unit1
31  `define SIGMOID_START_1         (`ARR_START_1+`ARR_DELAY+16-`SIGMOID_DELAY)
32  `define SIGMOID_END_1           (`SIGMOID_START_1+16)
33
34  //depends on sigmoid
35  `define MULTIPLY_UNIT1_START_1  (`SIGMOID_START_1+`SIGMOID_DELAY)
36  `define MULTIPLY_UNIT1_END_1    (`MULTIPLY_UNIT1_START_1+16)
37
38  //depends on multiply unit1
39  `define ADD_UNIT2_START_1       (`MULTIPLY_UNIT1_START_1+`MULTIPLY_DELAY)
40  `define ADD_UNIT2_END_1         (`ADD_UNIT2_START_1+16)
41
```

```
42      //depends on add unit2
43      `define TANH_START              (`ADD_UNIT2_START_1+`ADD_DELAY)
44      `define TANH_END                (`TANH_START+16)
45
46      `define ADD_UNIT1_START_2       (`ARR_START_1+`ARR_DELAY+32)
47      `define ADD_UNIT1_END_2         (`ADD_UNIT1_START_2+16)
48
49      //depends on add unit1
50      `define SIGMOID_START_2         (`ADD_UNIT1_START_2+`ADD_DELAY)
51      `define SIGMOID_END_2           (`SIGMOID_START_2+16)
52
53      //depends on sigmoid
54      `define MULTIPLY_UNIT1_START_2  (`SIGMOID_START_2+`SIGMOID_DELAY)
55      `define MULTIPLY_UNIT1_END_2    (`MULTIPLY_UNIT1_START_2+16)
56
57      //depends on sigmoid
58      `define ADD_UNIT2_START_2       (`SIGMOID_START_2+`SIGMOID_DELAY)
59      `define ADD_UNIT2_END_2         (`ADD_UNIT2_START_2+16)
60
61      //depends on tanh and add unit2 (should be aligned with first tanh result)
62      `define MULTIPLY_UNIT2_START    (`ADD_UNIT2_START_2 + \
63                                      ((`ADD_DELAY>`MULTIPLY_DELAY)?`ADD_DELAY: \
64                                      `MULTIPLY_DELAY))
65      `define MULTIPLY_UNIT2_END      (`MULTIPLY_UNIT2_START+16)
66
67      //depends on multiply unit2 and multiply uni1 (need ff delay buffer)
68      `define ADD_UNIT3_START         (`MULTIPLY_UNIT2_START+`MULTIPLY_DELAY)
69      `define ADD_UNIT3_END           (`ADD_UNIT3_START+16)
70
71      `define GRU_DELAY               (`ADD_UNIT3_START+16+`ADD_DELAY)
72
73      `define ADD1_SIGMOID_DELAY      (`SIGMOID_START_1+1 - \
74                                      (`ADD_UNIT1_START_1+1+`ADD_DELAY))
75      `define WARR_ADD2_DELAY         (`ADD_UNIT2_START_1 - \
76                                      ((`ARR_START_1+`ARR_DELAY+16)))
77      `define TANH_MULTIPLY2_DELAY    (`MULTIPLY_UNIT2_START - \
78                                      (`TANH_START+`TANH_DELAY))
79      `define MULTIPLY1_ADD3_DELAY    (`ADD_UNIT3_START - \
80                                      (`MULTIPLY_UNIT1_START_2+`MULTIPLY_DELAY))
```

The scheduling of second GRU layer is mostly the same with the first one, excepting changing the Linear 1x96 Unit to the Linear 32x96 Unit to take the output of previous layer, h_tm, as the input and conducts the operation. Then the start time and end time of two Linear 32x96 Unit would be the same and they will be controlled together.

In term of IP input resource, the Linear 1x96, Linear 32x96, 1st Adder, 3rd Adder, 2nd Multiplier, Sigmoid and Tanh Unit have fixed input source so they will be directly wired to the source as shown in Table 5.5 and wired to the shift register if the source need to be delayed. The 2nd Adder, 1st Multiplier have multiple input source during different round of computation, so muxers

are used to mux out the input source according to the FSM state which reflecting the computation round.

5.3.2 MLP Unit Scheduling and Control

As the float point adder has two more cycles latency while other remains the same, the latency of Dense Unit has changed to 20 cycles due to the lower speed of add tree and bias adder. However, since the outside unit controls the Dense Unit in a blackbox way, the changing latency would not influence the control scheduling of external unit like valid signal. Then the timing of the MLP Unit would be the same with the Vivado version as shown in figure 4.20, excepting the latency has changed to 20 cycles for all the Dense Unit.

5.3.3 System Scheduling and Control

As the functionality of GRU Unit and MLP Unit remains the same, the system scheduling would be the same as the Vivado version. The GRU Unit is made into two interleaved GRU layers running in a pipeline way in hardware level as shown in figure 4.21 to increase the hardware utilization rate across the time and minimize the system processing latency. When the GRU layers iterate for 196 times, it would output the 32-sized result to the Dense Unit to connect each extracted features via MLP layer. Then the three layer of 32x32, 32x16 and 16x1 would dense the features to 1-sized result and after the Sigmoid operation it would be the predicted reaction rate of system.

GRU Unit

The GRU layer1 would firstly to be started to make it one iteration ahead of layer2. After it finished its first iteration, the *layer*1_*done* signal would be high for one cycle, and the pipeline register would update the storing data of this iteration results. Then the layer1 would start second round of iteration, and the layer2 would be aligned with it to have module *valid* high pulse at same cycle.

Another control counter would count the iteration number ever time layer1 is done, and would end the GRU layer when all the 196 iteration is done and layer2 drained out its pipeline. Finally the *GRU*_*done* signal would be high for one cycle to indicate all the GRU computation is done and ready to conduct MLP computation. Then the timing of GRU Unit could be defined as in figure 5.19.

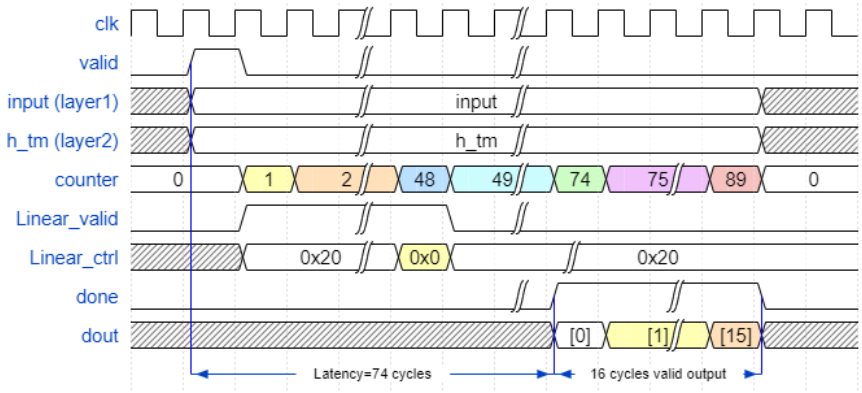


Figure 5.19: Timing of proposed GRU Unit

The *valid* signal of unit is the high pulse wave for one cycle, and the 1-sized input of layer1 or 32-sized *h_tm* of layer2 are required to be valid with the *valid* signal and remains unchanged during the computation of GRU Unit. Then after the valid is detected to be high, a inner counter would starts to count whose number would be considered as the state of FSM.

The IPs inside the Unit would start and end according to the current counter number (state). For example, the valid signal of Linear 32x96 would be set high since counter is 1 and keep for 48 cycles, resetting when counts to 48. Also, the *ctrl* signal of Linear Unit would be changed synchronously and reset to 0x0 when count to 32 and return to 0x20 then valid is done. Finally after 74 cycles, the first two output would be valid and the done signal would be high since then for 16 cycles. As all the results have been transmitted, the unit's inner register like *ctrl* or counter would be reset to prepare for the next round operation. According to the timing graph, the overall II of GRU Unit is 89 cycles and latency is also 89 cycles.

MLP Unit

As the MLP Unit reuse the Linear 32x96 Unit inside the GRU layer2, the layer2 would be start again after *GRU_done* is done. To control the layer to support Dense operation, a configure signal would be set to indicate whether to conduct Dense 32x32, 32x16 or 16x1 computation. Then the counter would have different threshold value to reset according to the Dense operation type (36, 28, 21 for Dense 32x32, 32x16 and 16x1 respectively). When the last Dense layer of 16x1 is done, the *MLP_done* signal would be high for one cycle to indicate the MLP layer is done. Finally a Sigmoid Unit would compute the final reaction rate with MLP layer's result and set the system's *done* to high.

5.4 Simulation

The testbench of the system uses the same platform as the Vivado version whose structure is shown as in figure 4.23. The trace unit is preserved to trace all the intermediate variables, including output of Tanh, Simoigd Unit, updated h_tm1 and h_tm1' value after each iteration, the final output of RNN system, etc. And the the tracer would record these signal into local file to be evaluated in the Evaluation Chapter.

Similarly, all the 100 cases of test data are stored in an array in testbench where each cases contains 196 32-bit elements. The sequence buffer would take in a case according to the test case number and start to send out values element by element in the test case to the DUT.

Also, a inner counter of RNN system is instantiated to record the iteration number and ask outer memory to provide corresponding 1-sized input data in each iteration. The counter has an output port of *index*, which would be connected to the sequence buffer to provide correct data.

A control unit is then to control all the testing process from the test number counting and boundary check, and enable and disable the DUT and control the trace unit.

The simulation result of DUT under this architecture is shown as in figure 4.24. The 0.012 deviation between groundtruth (the number in the online converter 0.7725399. The C++ model is also compared whose result is in the black box) and DUT output (around the yellow cursor) suggesting the proposed RNN system functions correctly. The more timing and resource as well as output accuracy analysis could be seen in Chapter Evaluation.

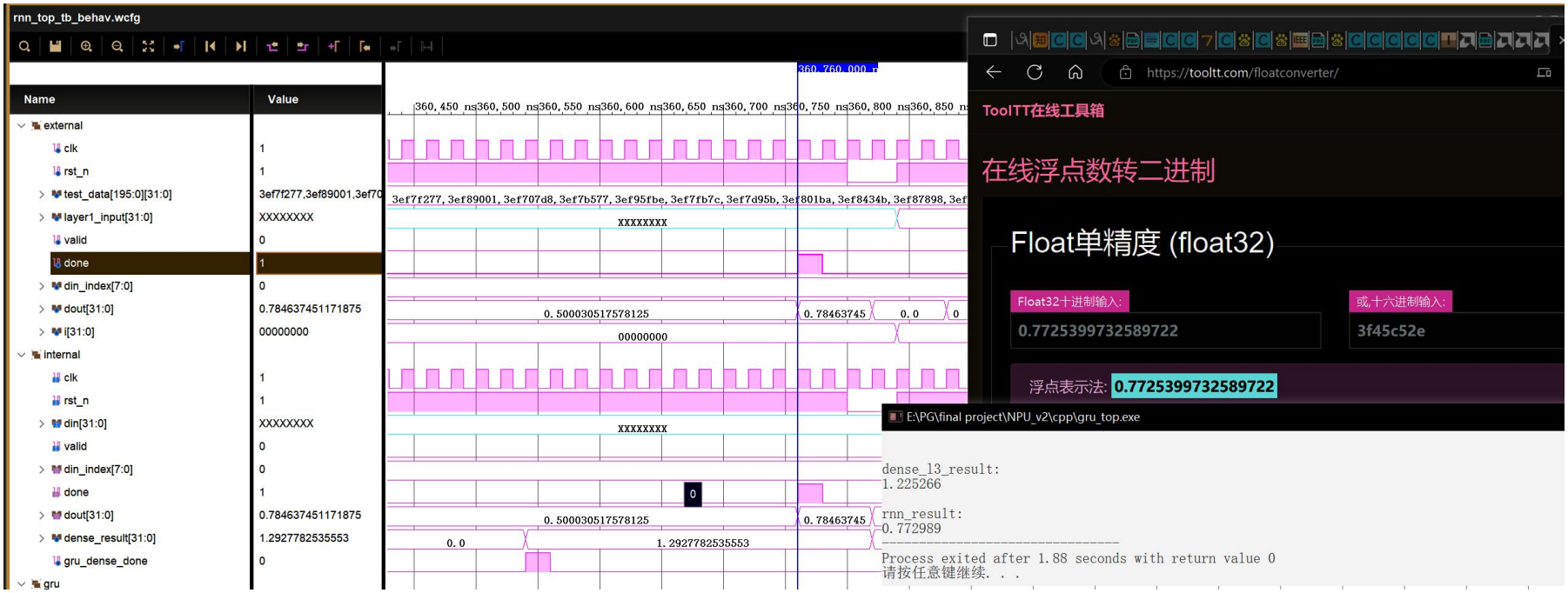


Figure 5.20: Simulation of proposed RNN system using customized IP

6

ASIC Implementation

Contents

In Chapter 5, the Vivado commercial IPs has been substituted to the fully customized IPs and the usage of FPGA dedicated DSP resource has been expanded to the combination logic. This makes the ASIC implementation possible to establish dedicated integrated circuit. Then in this chapter, the RNN system would be implemented into ASIC with TSMC 65nm low power technology under the help of senior student.

6.1 Transplantation from Fully Customized IP Version

Although the previous chapter has substitute the Vivado IP with the customized IP that enables the system could work outside the Vivado environment, the previous system is targeted to be implemented on FPGA platform. Since the principle of running FPGA is the mapping from hardware Verilog code to the the combination and connection of onboard pre-existed resources like LUT, FF, etc, the transplantation toward the ASIC targeted Verilog design is needed.

In the previous design, as the DSP resource has been expanded to the combination logic that uses LUT and FF, only three types of dedicated resources have been used in the design (LUT, FF and BRAM). The transplantation then starts from mapping these units into the ASIC units.

Considering the LUT is intrinsically a tiny ram that stores all the possible results of input signal to implement combination logic in FPGA implementation, the ASIC tools would directly mapping these logic circuit in Verilog to the combination of logic gate which do not need any further extra operations.

In term of FF, as its basic structure is the Flip-Flop, these would then be synthesized and mapping to the Flip-Flop unit in the standard cell library when doing the ASIC, which also indicates these part of resource do not needs to pay extra attention.

However, as for the BRAM resource has no direct mapping in the standard library, either it needs to be implemented with Random Access Memory (RAM) IP, or assemble the memory using register and readout circuit. As the use of other source IP would again cause the property issue, second solution has been selected to map the BRAM.

Then in the RAM, the data is memorized in the register, which is implemented as a register group ([DWIDTH-1:0] mem [DEPTH-1:0] for storing DEPTH of DWIDTH wide data). And the input control signal would be used to index the register group, and readout the memorized data to the output register. Beside, as the initial sequence used in FPGA implementation is not synthesizeable in the ASIC tools, the initialization of the memory files has changed to the enumeration of assignment to each element in the array and a C++ tool has been developed to automatically mapping the hex file to the list of assignment to finish this job.

After all these mapping of resource, the hardware design has been finally transplanted to the ASIC version and could be able to implemented with ASIC tools.

To capsuled the system into a chip, the following structure has been used as shown in figure 6.1.

A *TOP* module would wrap the RNN system and connect its input and output port out and then connecting to the corresponding pad cell on the chip die. Also, the power line of the system has been added and also connect to the pad cells. Then in the following ASIC implementation the *TOP* module would be set as the top level design.

6.2 Synthesis

The synthesis of the system is based on the Genus platform from Cadence company and to simplify the process, the script is used during the procedure. Firstly the script would read all the Verilog

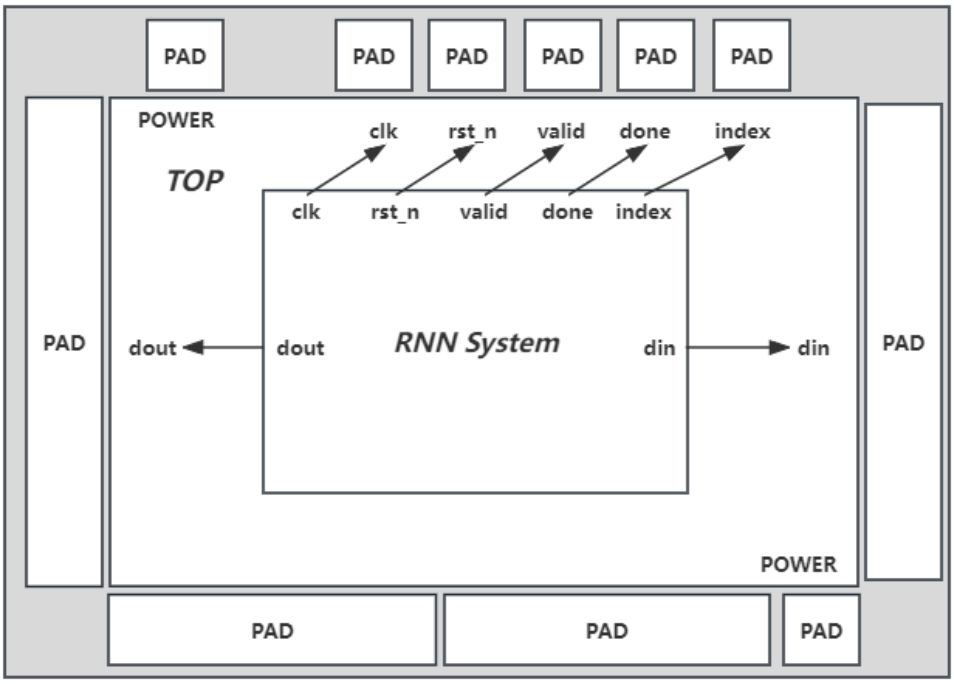


Figure 6.1: Structure of proposed RNN system in ASIC level

design file and parsing and synthesize them with loaded library. The clock in the synthsis has been set to 100MHz; the VDD and VSS power pin has been connected to the standard cells.

For the constraints, a model of 1pF capacitance and 1ns time has been set; the clock has been configured to have 1ns latency and 0.2ns transition time with 0.2ns uncertainty in setup time and 0.05ns in hold time; then the I/O port has been set to have maximum 2ns delay and minimum 0.2ns delay; drive and load of the circuit is configured to have $1k\Omega/pF$ at maximum and $0.01k\Omega/pF$ at least. To make the system friendly for the STA, the asynchronous path of rst_n signal and the power signal has been set to false path. Then the overall hardware would be synthesized and the system netlist will be generated.

6.3 Implementation

The implementation part is based on the Innovus platform from Cadence company and to simplify the process, the script is used during the procedure. Firstly the floorplan of the system is stipulated to be as in figure 6.2.

As the *din* and *dout* all have 32 bits and *index* has 8 bits, top bar would first arrange the

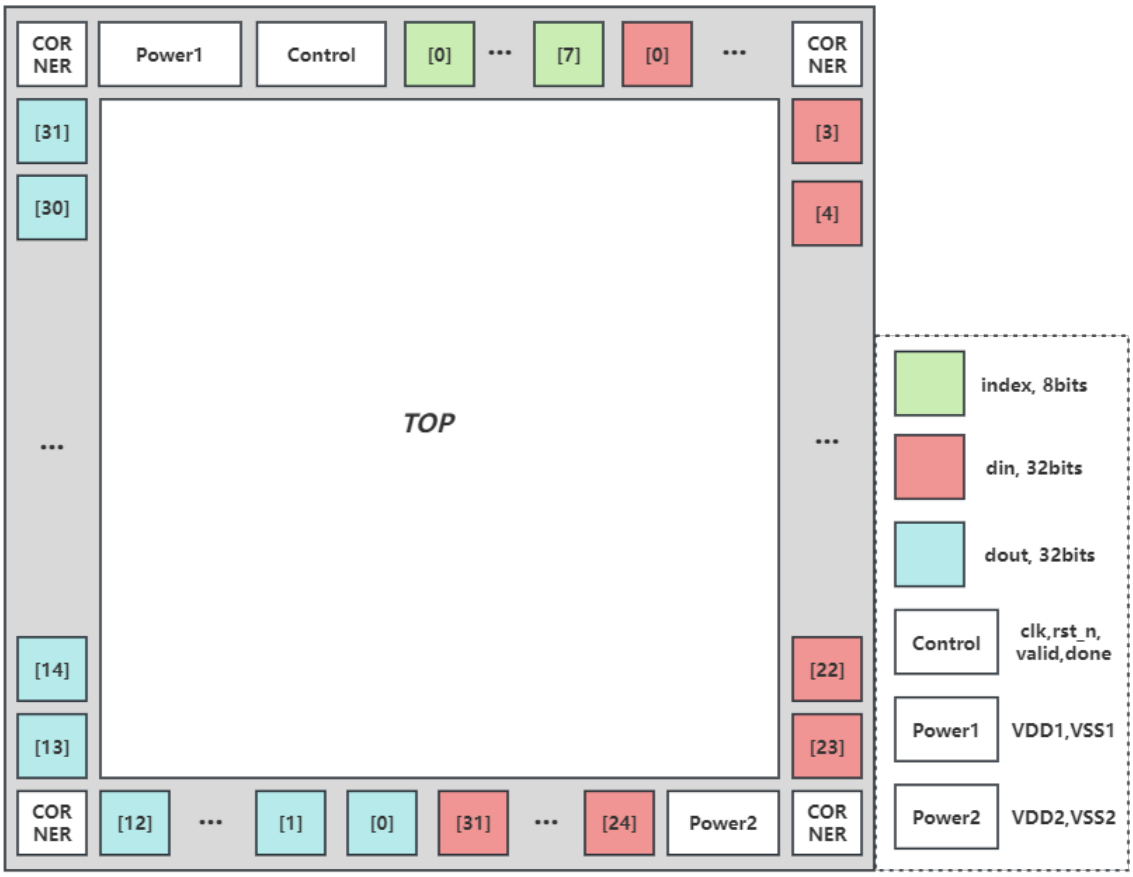


Figure 6.2: ASIC floorplan of proposed RNN system

Power1 (VDD1 and VSS1), control signals (*clk*,*rst_n*,*valid*,*done*), *index* ([0] to [7]) and the *din* ([0] to [2]) in sequence from left to right. The right would have *din* ([3] to [23]) in sequence from top to bottom; The bottom bar would have Power2 (VDD2 and VSS2), *din* ([24] to [31]) and *dout* ([0] to [12]) in sequence from right to left; The left bar would have *dout* ([13] to [31]) in sequence from bottom to top. Then the chip could be made to be squared and the length has been set to be 2000 micro meters and would be adjusted automatically according to the tool.

After this, the I/O filler, power rings, tap cells would be added to enhance the electrical feature of the layout. Then the tool would map the hardware to the standard library cell of TSMC 65nm low power techonolgy and running the routing of the unit to connect them together based on the netlist provided in the Synthesis step. Finally the tool would check the design rule and timing of the system while trying to optimize the system. If the design fails any of them, the tool would literately run the process until the requirements are all fulfilled.

The final elaborated layout of the system is shown as in figure 6.3.

The setup and hold timing of the system is shown as in Table 6.1 and6.2 respectively, all

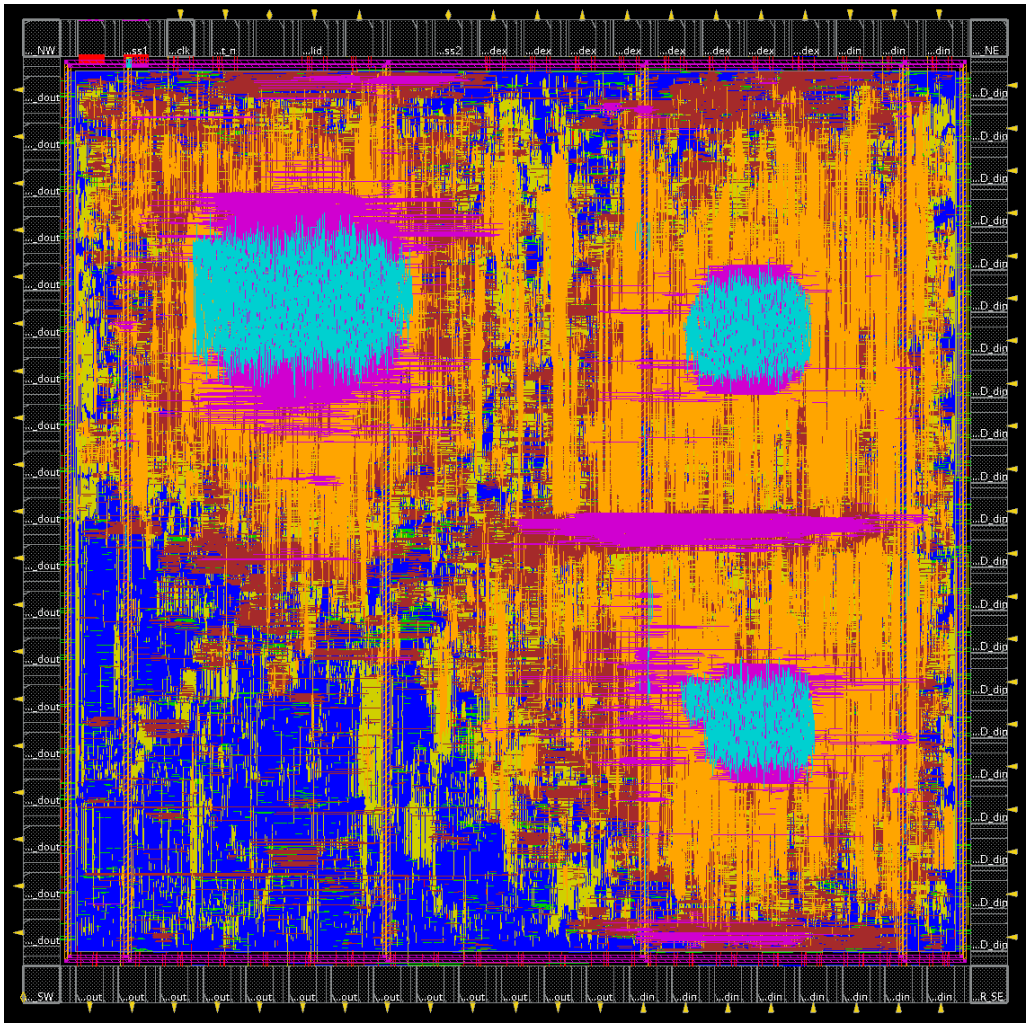


Figure 6.3: Eleborated ASIC layout of proposed RNN system using TSMC 65nm Low Power Technology

indicating the system has fulfill the constraints of the 100MHz clock.

Table 6.1: Timing of Setup mode of the RNN system

Setup Mode	**all**	**reg2reg**	**in2reg**	**reg2out**	**in2out**
WNS(ns)	0.226	0.226	4.376	3.737	N/A
TNS(ns)	0.000	0.000	0.000	0.000	N/A
Violeting Paths	0	0	0	0	0

Finally the area of chip die is reported to be 2583213μm^2 while the module density is to be 98.9%, indicating the floorplan is tit enough to fully utilize nearly all the space on the die.

The report for power consumption is shown in the figure 6.4.

The total energy consumption of the system is 391.9W when running with 100MHz clock. Among them, the combination logic take 64.89% of the total consumption and the second is

Table 6.2: Timing of Hold mode of the RNN system

Hold Mode	all	reg2reg	in2reg	reg2out	in2out
WNS(ns)	0.042	0.042	0.624	1.294	N/A
TNS(ns)	0.000	0.000	0.000	0.000	N/A
Violeting Paths	0	0	0	0	0

Group	Internal Power	Switching Power	Leakage Power	Total Power	Percentage (%)
Sequential	79.45	7.158	0.0729	86.68	22.12
Macro	0	0	0.002857	0.002857	0.0007289
IO	21.26	17.19	0.003015	38.46	9.812
Physical-Only	0	0	4.589e-05	4.589e-05	1.171e-05
Combinational	131	122.9	0.4049	254.3	64.89
Clock (Combinational)	2.485	9.999	0.003908	12.49	3.186
Clock (Sequential)	0	0	0	0	0
Total	234.2	157.3	0.4876	391.9	100

Figure 6.4: Power consumption of proposed RNN system

the Sequential logic which take 22.12%. However, the Internal Power of both sequential and combination logic is high, and this is considered to be the power consumption of memory cell that uses the flip-flop to store the information and would cause a majority of energy. Also, the low power design methodology are considered to be applied more to decrease the switching power, which also takes considerable proportion.

In term of Design Rule Check (DRC) and Layout Versus Schematic (LVS) check of layout, as the server of ASIC tool is too slow to be able to put the pad cell manually due to the limitation of hardware, this part of job would be considered to be finished in the future.

7 Evaluation

Contents

In previous sections, the FPGA implementation of proposed RNN system has been established with both Vivado IPs and fully customized IPs. In this section, the performance, hardware usages and accuracy of these two version of design will be compared and analyzed with the HLS work of [1].

7.1 Performance

When simulating the two version of design using the 100MHz clock, the II as well as the overall latency (and II) of two designs are measured to be 18197 (Vivado Version) and 18031 (Customized IP Version) cycles respectively when conducting 196 iteration for the GRU unit (system latency is measured as the interval between the posedge of the valid signal and the posedge of the done signal). Although the Dense Unit in the customized version has 10 more cycles latency than the Vivado version due to the increased latency of float point adder, the decrease 1 cycle of latency in the GRU Unit across 196 iterations makes the overall latency even smaller.

Comparing the HLS design of [1] whose overall latency (and II) is 56254 cycles with 100MHz clock estimated by HLS software, the Vivado Version has a 67.7% increase on processing speed (both the latency and II) and the customized version has a 67.9% increase on processing speed.

As both of the version has a major increase on the processing speed, the performance could be considered has been further optimized.

7.2 Hardware Usage

The comparison of three versions of hardware implementation regarding hardware usage is shown as in Table 7.1.

Table 7.1: Comparison of three versions of hardware implementation regarding hardware usage

Version	**LUT**		**FF**		**DSP**		**BRAM**	
	Number	[i]*Drop%*	*Number*	*Drop%*	*Number*	*Drop%*	*Number*	*Drop*
work in [1]	465238	N/A	233394	N/A	1908	N/A	307	N/A
Vivado IP	**100169**	**77.2%**	58610	74.6%	1167	38.8%	192	37.5%
Custom IP	266467	39.4%	**52385**	**76.8%**	**0**	**100%**	**184**	**40.0%**

[i] The drop percentage is compared with the [1]'s HLS design

In term of LUT resource, the Vivado and customized version uses 100169 and 266467 units respectively and have a drop on resource of 77.2% and 39.4% compared to the work of [1]. Although both of the version has a decrease on resource, the Custom version uses almost double amount of LUT. However, considering the complete disassemble of DSP unit in customized version, this is the necessary trade-off for going towards the ASIC design.

For the FF and BRAM resource, the customized version has a higher decrease of 76.8% and 40% compared to Vivado version who has dropped 74.6% and 37.5%. Also, the DSP resources has been fully constituted with other resources in customized version while the Vivado version still used 1167 units of DSP but still have 38.8% drop. To sum up, both version has a major decrease on hardware resources where the Vivado version has less LUT but more DSP resources than the customized version who uses no DSP units.

7.3 Accuracy

The accuracy of the system is tested by running 100 test cases on the testbench of figure 4.23 where each case contains 196 number of float point number to be iterated 196 times. The tracer in the testbench would trace the updated value of output in each layer for every iteration and the system final output and store it in the recording file whose format is shown as in figure 7.1 (M=100, N=196).

	Head file name	Test case 1			…	Test case M			Total Number
		Iter 1	…	Iter N		Iter 1	…	Iter N	
GRU Layer1	gru1_sigmoid_data_din.h	64	…	64	…	64	…	64	64*N*M
	gru1_sigmoid_data_dout.h	64	…	64	…	64	…	64	64*N*M
	gru1_tanh_data_din.h	32	…	32	…	32	…	32	32*N*M
	gru1_tanh_data_dout.h	32	…	32	…	32	…	32	32*N*M
GRU Layer2	gru2_sigmoid_data_din.h	64	…	64	…	64	…	64	64*N*M
	gru2_sigmoid_data_dout.h	64	…	64	…	64	…	64	64*N*M
	gru2_tanh_data_din.h	32	…	32	…	32	…	32	32*N*M
	gru2_tanh_data_dout.h	32	…	32	…	32	…	32	32*N*M
Dense Layer	dense_sigmoid_data_din.h	1			…	1			1*M
	dense_sigmoid_data_dout.h	1			…	1			1*M
Layer Output	htm_l1.h	32	…	32	…	32	…	32	32*N*M
	htm_l2.h	32 … 32	32	16 2	…	32 … 32	32	16 2	(32*N+32+16+2)*M
Result	rnn_results.h	1			…	1			1*M

Figure 7.1: Store format of recorded tracing data

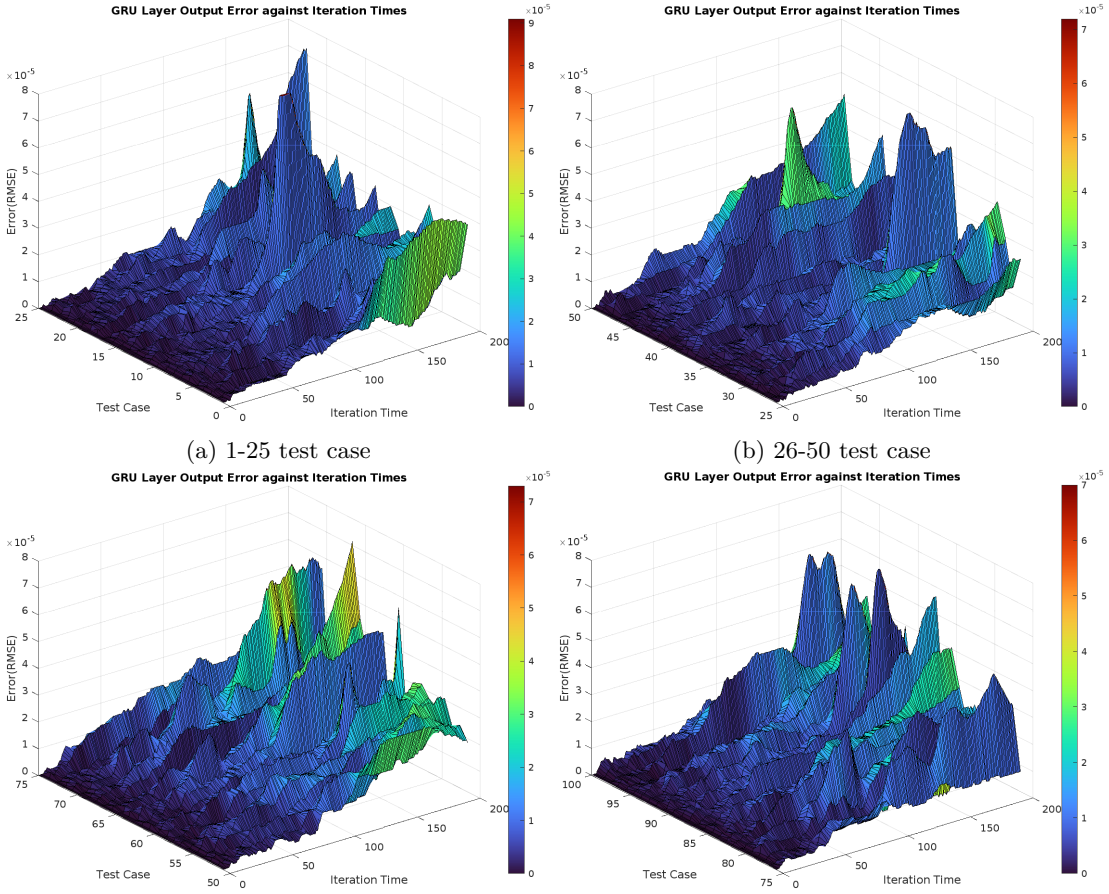
To analyze the result and visualize the trend, peripheral C++ tools are developed to calculate the output error of GRU layer compared with C++ model. Then the output error of GRU Unit compared to groundtruth across the iteration time in RMSE form for the 100 test cases is shown as in figure 7.2 and 7.3 of Vivado and customized version respectively.

It could be observed that the both version have a increase on the error across the iteration time, and in some cases the error would be obviously greater than other cases but still stays at the same order of magnitude.

However, the customized version's error accumulates faster than the Vivado's which results to a considerable difference after 196 iteration (1×10^{-1} level and 1×10^{-4} level). This results to a final RMSE error difference for the 100 test cases where the Vivado version has 7.7×10^{-5} and customized version of 1.54×10^{-1} while the HLS version has the error of 1.08×10^{-4}. The possible reason is analyzed as following.

Firstly, the DSP resource has 48 bits of operation width which could provide a higher resolution when computing but the IP core used in customized version only have 24 bits of width from the point of trade-off who has an RMSE error of 4.33×10^{-7} and 2.05×10^{-6} for float point addition and multiplication respectively. In addition, since the memory information in the layers will not be cleared for each iteration, any tiny error would accumulate to be a considerable deviation across a large number of iteration time like 196 and no need to say more error would be added multiple times in each iteration.

Besides, the activation function which computed using the CORDIC IP in the customized version would also introduce substantial error because of its structure. Since the CORDIC has change the input range of *Sigmoid* to no greater than 8 and *Tanh* to no greater than 4 to make the module converge, any input number greater than this value would be set to the infinite value of the function. And this would introduce an error at most 6.707×10^{-4} and 1×10^{-4} if inside the



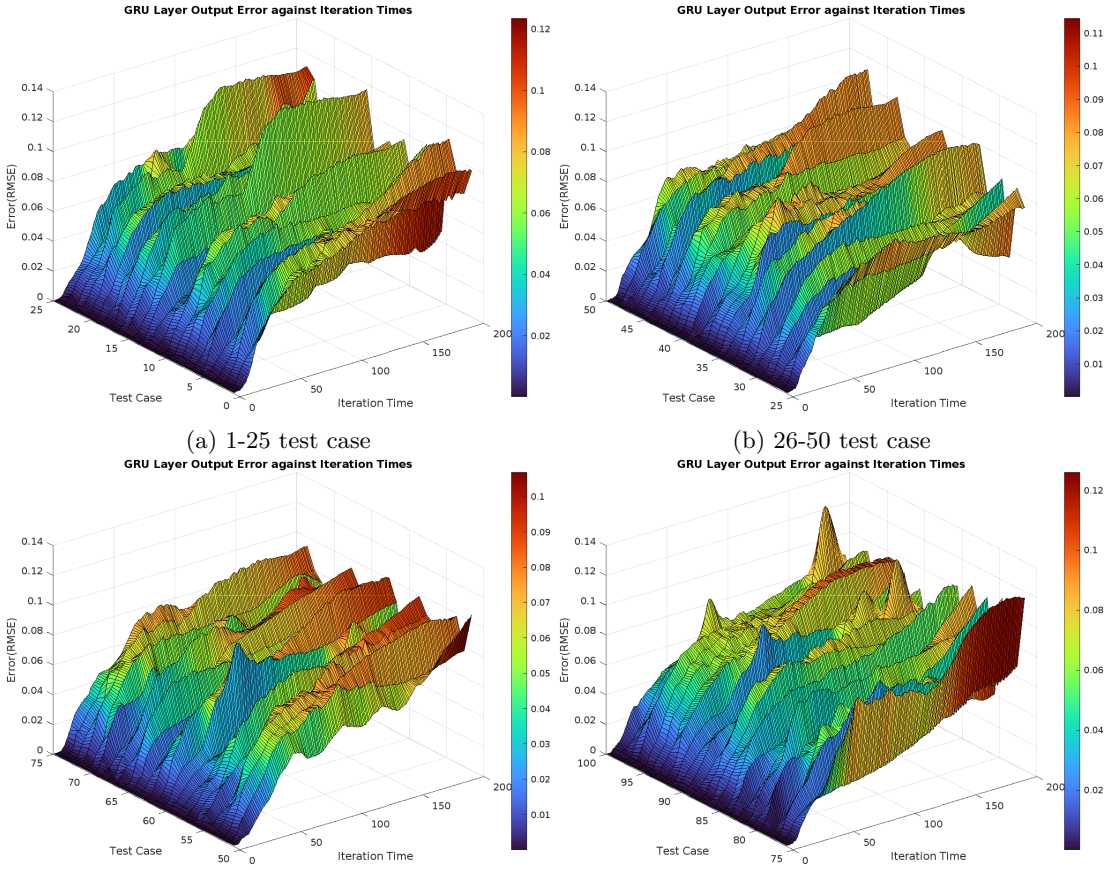
(a) 1-25 test case

(b) 26-50 test case

(c) 51-75 test case

(d) 76-100 test case

Figure 7.2: Output error in RMSE of GRU in Vivado IP version against iteration time



(a) 1-25 test case

(b) 26-50 test case

(c) 51-75 test case

(d) 76-100 test case

Figure 7.3: Output error in RMSE of GRU in Customized IP version against iteration time

range. Similarly as the float point addition and multiplication, these error would be introduced three times per iteration, which could finally influence the output result.

Also, the weight and structure of the model provided in the work of [1] could be intrinsically sensitive to the operation errors across the iteration considering its recursive structure.

In conclusion, the accuracy of the Vivado version has a minor increase than the the work of [1] while the customized version has a increased error at the expense of substituting the DSP resources and other commercial IPs.

8 Future Work

Contents

8.1 Proposed Work

Previous chapters has demonstrated two optimized versions of hardware implementation for the work of [1], and results to a desirable optimization on both performance and hardware resources. In term of accuracy, the Vivado version shows a minor improvement while the developing customized version has an increased computation error as trade-off. Besides, the promising potential of the work of [1] encourage the second version to be implemented into ASIC, which results to a favorable outcome on the chip die size and the module density.

8.2 Next Generation Implementation

In next generation of implementation, the deficiencies of previous Vivado and customized version could be further optimized.

Firstly, the ASIC implementation shows a nearly 400W power consumption requirement even

using the low power technology. This could be improved in the top level by changing the architecture of computation framework and applying the low-power design methodology.

Power gating and gated clock could be used to the unit when stalling but not just stop the unit in behavior level by latching up the output logic and keep the flip-flop switching which would constantly introduce dynamic consumption. The multi-voltage technology would be used to compute short path slowly by applying a lower voltage and long path quicker with higher voltage.Also, the memory cell could also be optimized like commercial IPs to slightly slows down the read-out speed but greatly increase the energy efficiency. And the channel number and the IP size could be re-designed to grind the operation path even finely and fully exploit the performance of device

Secondly, since error of customized version is considerably greater than the proposed work, the error source needs to be further deeply analyzed and peeled out. This could be done by applying more accurate rounding strategy and increase the computation width of float point addition and multiplication. In term of activation function, more negative stages in RHC units could be added to further expand the converge range and more positive stages of VLC and RHC are considered. Also, another way that choose to disassemble the activation function into exponential form could be compared to have the best accuracy-resources balance.

Finally, the ASIC implementation could be further pushed to check the electrical rules and design rules (LVS and DRC) to validate the design of layout, which pave the path for the further optimization and the succeeding wafer-processing.

Conclusions

Firstly, this paper implemented a dedicated hardware architecture of RNN network in [1]. Based on the framework, a finely grained and optimized computation platform has been established in Vivado with Xilinx IPs. As parallelism has been deeply and effectively applied with the focus on this particular network and further the hardware system, this version has finally reached an increase in speed for 67.6% (using 100MHz clock, which is the same with the work of [1]) comparing to the hardware platform generated from HLS in the work of [1]. Also, the effectiveness has been improved in term of hardware resource: these is a decrease of 77.2%, 74.6%, 38.8%, 37.5% on LUT, FF, DSP and BRAM resources respectively which remaining an accuracy of 7.7×10^{-5} RMSE in 100 test cases compared to groundtruth.

Secondly, this paper established fully-customized and completely proprietary IPs of float point adder, float point multiplier and CORDIC implementation of *Tanh* and *Sigmoid* function, etc that would replace Xilinx's computation IPs in first version, expanding its opportunity to be used in commercial application. Similarly, this version has reached an increase in speed for 67.9% (using 100MHz clock, which is the same with the work of [1]) comparing to the hardware platform generated from HLS in the work of [1] and shows a decreased accuracy of 1.54×10^{-1} on RMSE error in 100 test cases compared to groundtruth. And 39.4%, 76.8%, 40% of LUT, FF and BRAM resources has been saved while no DSP resources is used as this version is targeted to a further ASIC implementation.

Thirdly, with senior student's help, an ASIC implementation using TSMC 65nm Low Power technology has been established. For this version, the chip is of 2583213 μm^2 area, 98.9% module density and within the timing constraint of 100MHz operating clock.

To sum up, this paper presented two version of optimization for the work of [1] on both performance and hardware resources. While the first version has minor improvement on the accuracy, the second one shows the increased computation error which could be further optimized. Finally the ASIC implementation for the layout of second version is done, unveiling the promising potential on both the significance and application field of this paper's work. Furthermore, since the IP of both version is designed with adjustable performance and features and the overall architecture

is designed in a parametric way, the whole system is extremely flexible and could be able to be re-configured to adopt other GRU network with minor adjustment, providing a universal hardware acceleration of full gated GRU network.

A

Appendix-Simulation Result of IP in Project

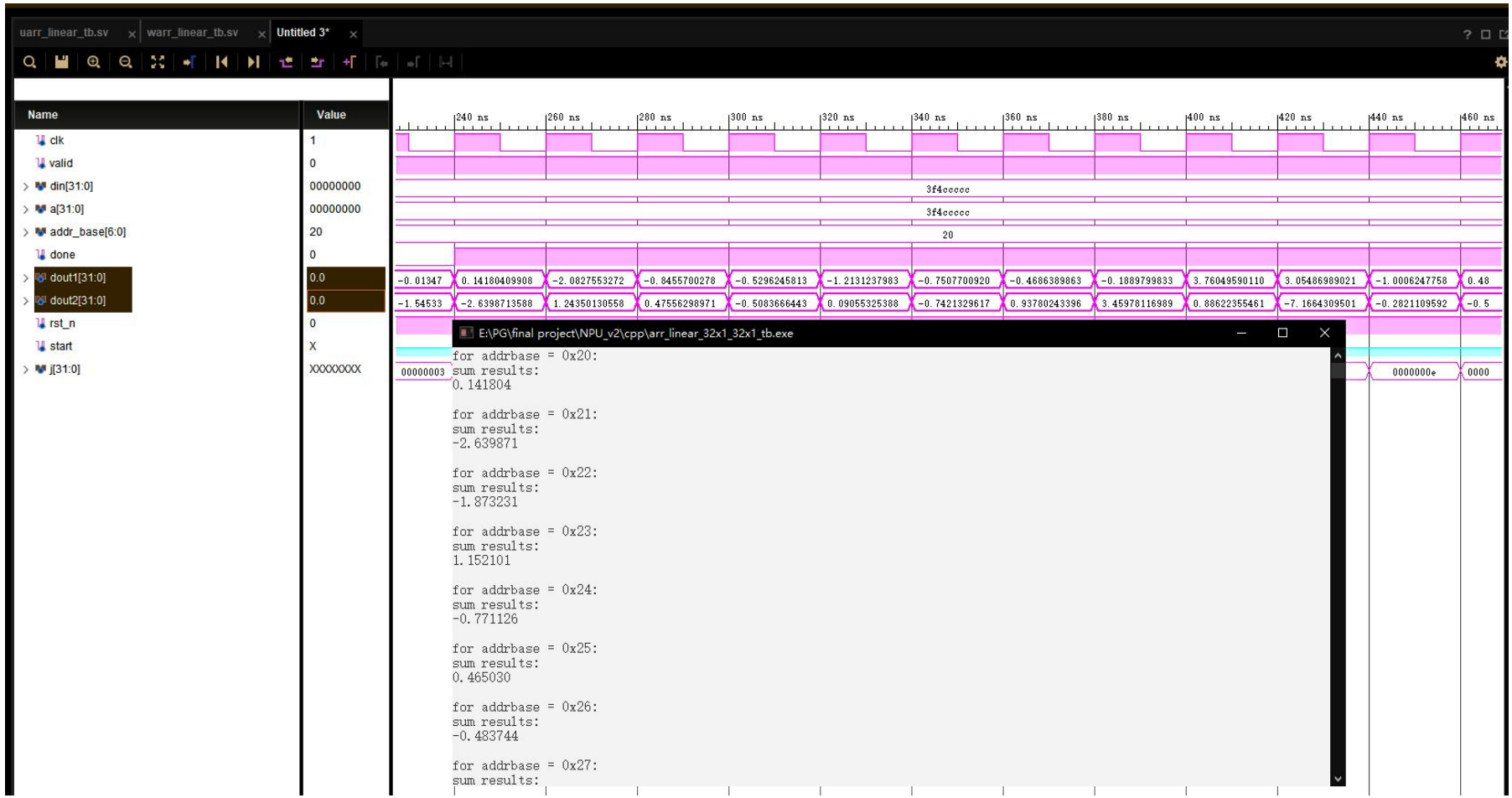


Figure A.1: Simulation result of Linear 1x96 Unit computing W related matrix in GRU layer1 with comparison of C++ results

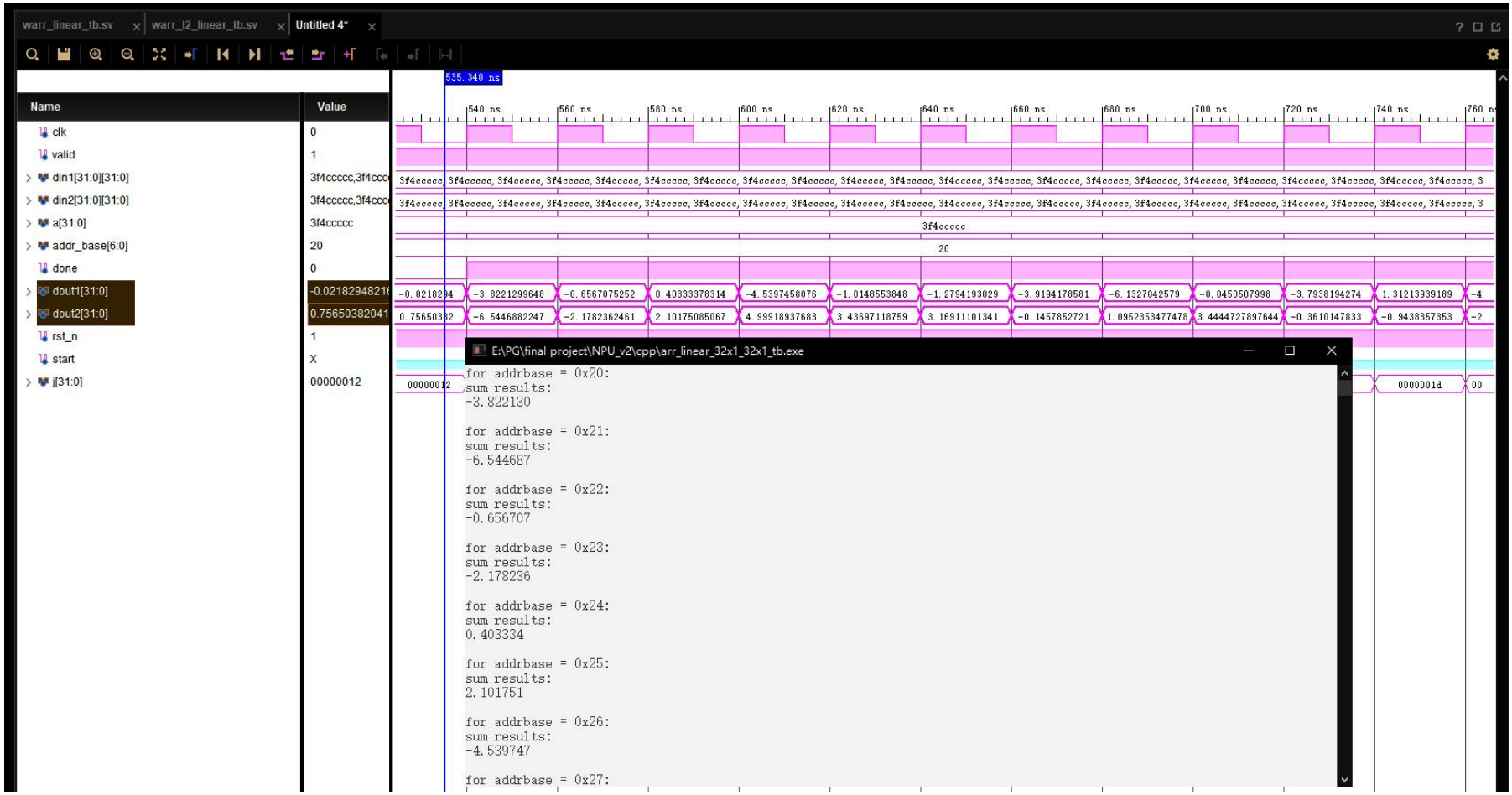


Figure A.2: Simulation result of Linear 32x96 Unit computing W related matrix in GRU layer2 with comparison of C++ results

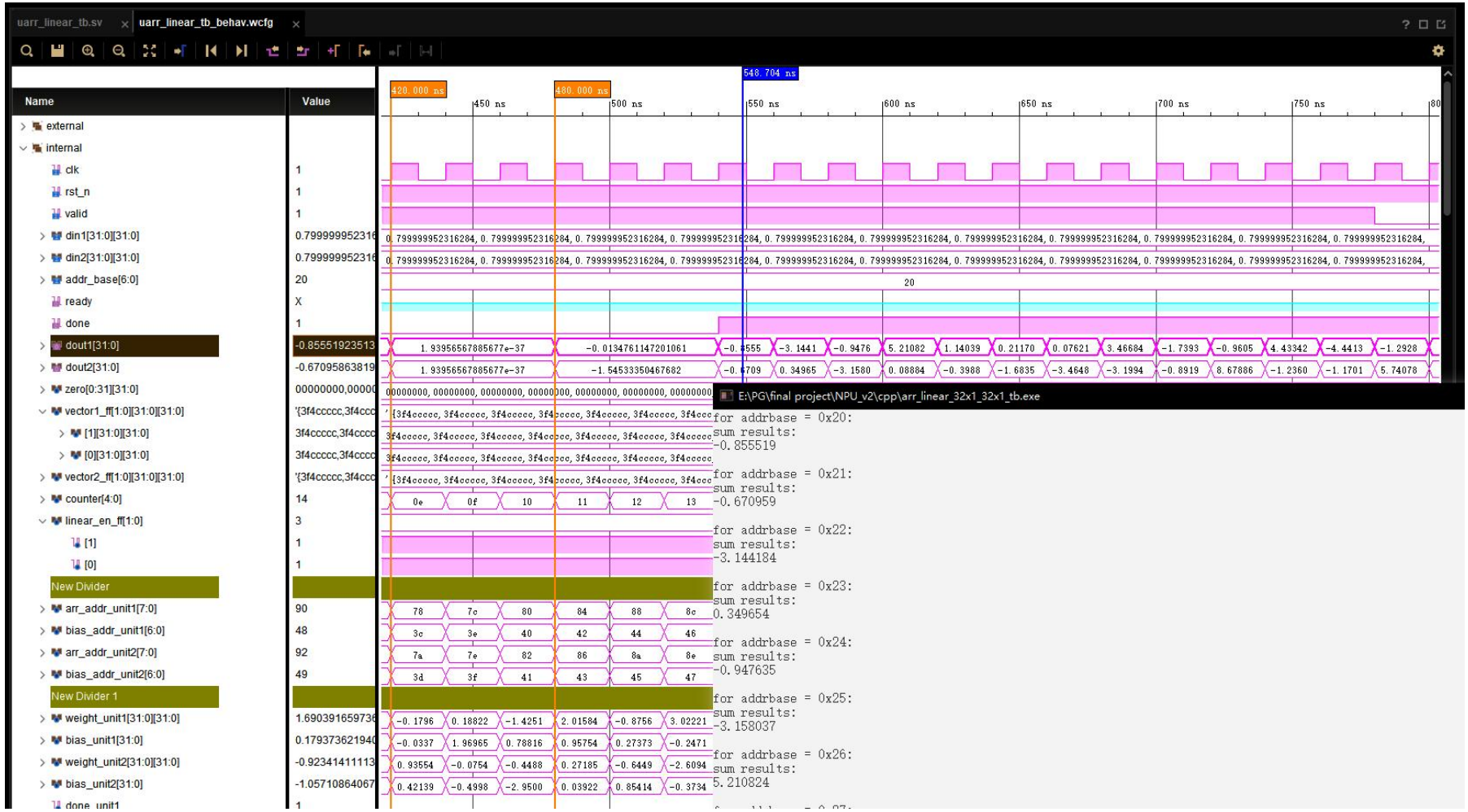


Figure A.3: Simulation result of Linear 32x96 Unit computing U related matrix in GRU layer1 with comparison of C++ results

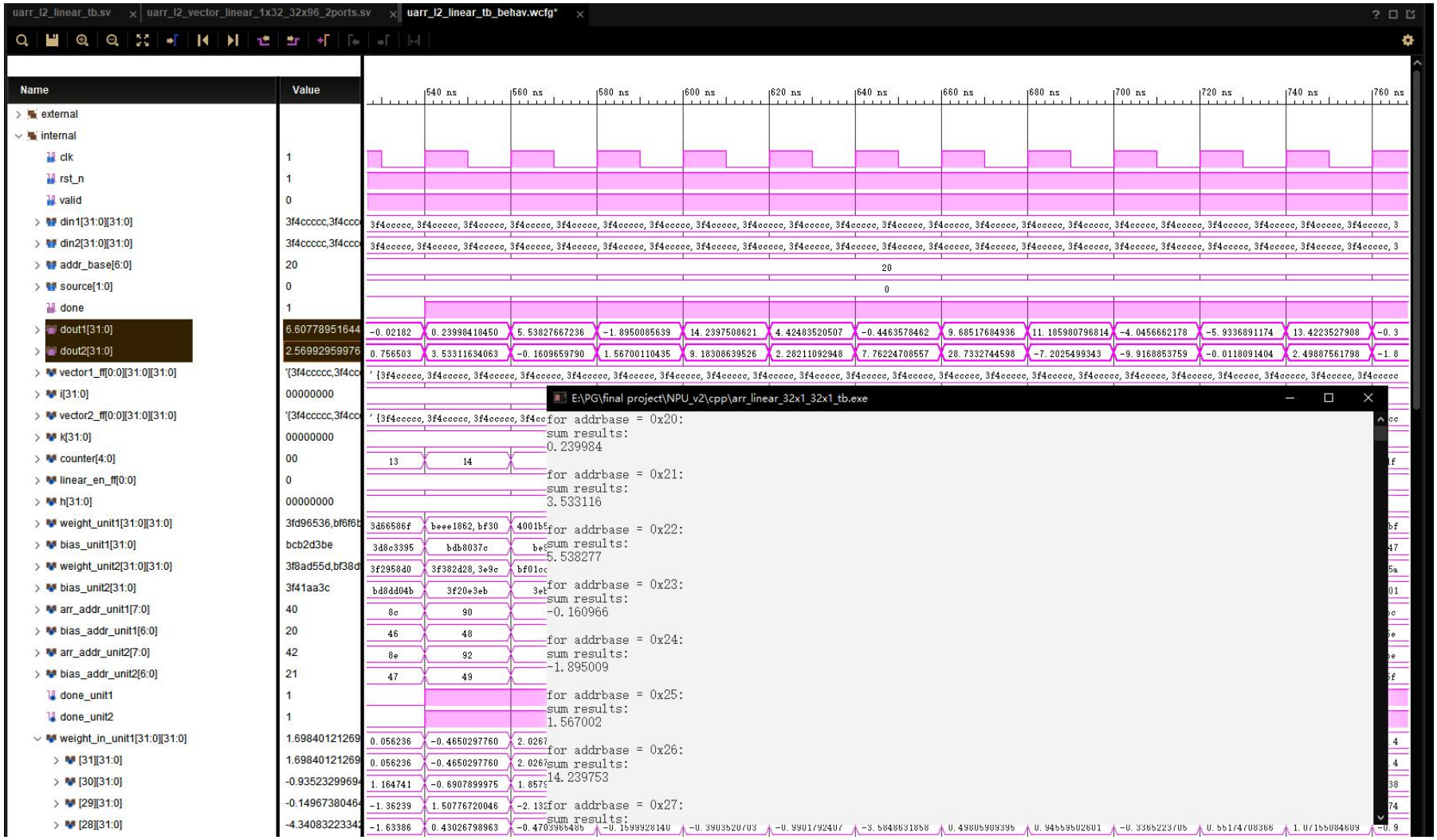


Figure A.4: Simulation result of Linear 32x96 Unit computing U related matrix in GRU layer2 with comparison of C++ results

A

Bibliography

[1] Y. Xu, L. Kuang, T. Zhu, J. Zeng, and P. Georgiou, "Drift prediction and chemical reaction identification for isfets using deep learning," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10181442.

[2] K. Malpartida-Cardenas, N. Miscourides, J. Rodriguez-Manzano, *et al.*, "Quantitative and rapid plasmodium falciparum malaria diagnosis and artemisinin-resistance detection using a cmos lab-on-chip platform," *Biosensors and Bioelectronics*, vol. 145, p. 111 678, 2019, ISSN: 0956-5663. DOI: https://doi.org/10.1016/j.bios.2019.111678. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0956566319307572.

[3] M. Kalofonou, K. Malpartida-Cardenas, G. Alexandrou, *et al.*, "A novel hotspot specific isothermal amplification method for detection of the common pik3ca p.h1047r breast cancer mutation," *Scientific Reports*, vol. 10, no. 1, p. 4553, Mar. 2020, ISSN: 2045-2322. DOI: 10.1038/s41598-020-60852-3. [Online]. Available: https://doi.org/10.1038/s41598-020-60852-3.

[4] J. M. Rothberg, W. Hinz, T. M. Rearick, *et al.*, "An integrated semiconductor device enabling non-optical genome sequencing," *Nature*, vol. 475, no. 7356, pp. 348–352, Jul. 2011, ISSN: 1476-4687. DOI: 10.1038/nature10242. [Online]. Available: https://doi.org/10.1038/nature10242.

[5] C. Toumazou, L. M. Shepherd, S. C. Reed, *et al.*, "Simultaneous dna amplification and detection using a ph-sensing semiconductor system," *Nature Methods*, vol. 10, no. 7, pp. 641–646, Jul. 2013, ISSN: 1548-7105. DOI: 10.1038/nmeth.2520. [Online]. Available: https://doi.org/10.1038/nmeth.2520.

[6] S. Jamasb, S. Collins, and R. L. Smith, "A physical model for drift in ph isfets," *Sensors and Actuators B: Chemical*, vol. 49, no. 1, pp. 146–155, 1998, ISSN: 0925-4005. DOI: https://doi.org/10.1016/S0925-4005(98)00040-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925400598000409.

[7] N. Sahu, R. Bhardwaj, H. Shah, R. Mukhiya, R. Sharma, and S. Sinha, "Towards development of an isfet-based smart ph sensor: Enabling machine learning for drift compensation in iot

applications," *IEEE Sensors Journal*, vol. 21, no. 17, pp. 19 013–19 024, 2021. DOI: 10.1109/JSEN.2021.3087333.

[8] R. Bhardwaj, S. Majumder, P. K. Ajmera, *et al.*, "Temperature compensation of isfet based ph sensor using artificial neural networks," in *2017 IEEE Regional Symposium on Micro and Nanoelectronics (RSM)*, 2017, pp. 155–158. DOI: 10.1109/RSM.2017.8069141.

[9] A. Harrak and S. E. Naimi, "Design and simulation of ph-isfet readout circuit for low thermal sensitivity applications through an automatic selection of an isothermal point," *Sensing and Imaging*, vol. 23, no. 1, p. 10, Mar. 2022, ISSN: 1557-2072. DOI: 10.1007/s11220-022-00378-2. [Online]. Available: https://doi.org/10.1007/s11220-022-00378-2.

[10] A. C. Ian Goodfellow Yoshua Bengio, *Deep learning*. Cambridge MIT Press, 2016, vol. 1, pp. 367–415.

[11] X. Qiu, *Neural Networks and Deep Learning (in Chinese)*. Beijing: China Machine Press, 2020, ISBN: 9787111649687. [Online]. Available: https://nndl.github.io/.

[12] Y. B. M. Andrew Ng Kian Katanforoosh, "Sequence models," in *Deep learning*, Coursera and deeplearning.ai.

[13] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[14] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, "Light gated recurrent units for speech recognition," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, pp. 92–102, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:4402991.

[15] Y. Su, Y. Huang, and C.-C. J. Kuo, "On extended long short-term memory and dependent bidirectional recurrent neural network," *Neurocomputing*, vol. 356, pp. 151–161, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:3675055.

[16] K. Cho, B. Van Merriënboer, C. Gulcehre, *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[17] *Recurrent neural network (rnn) – part 5: Custom cells*.

[18] J. C. Heck and F. M. Salem, "Simplified minimal gated unit variations for recurrent neural networks," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, IEEE, 2017, pp. 1593–1596.

[19] N. Gupta, "Chapter one - introduction to hardware accelerator systems for artificial intelligence and machine learning," in *Hardware Accelerator Systems for Artificial Intelligence and Machine Learning*, ser. Advances in Computers, S. Kim and G. C. Deka, Eds., vol. 122, Elsevier, 2021, pp. 1–21. DOI: https://doi.org/10.1016/bs.adcom.2020.07.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245820300541.

[20] L. Keeble, N. Moser, J. Rodriguez-Manzano, and P. Georgiou, "A combined isfet-electric field actuation system for enhanced detection of dna: A proof-of-concept," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9181097.

[21] N. Moser, J. Rodriguez-Manzano, T. S. Lande, and P. Georgiou, "A scalable isfet sensing and memory array with sensor auto-calibration for on-chip real-time dna detection," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, no. 2, pp. 390–401, 2018. DOI: 10.1109/TBCAS.2017.2789161.

[22] M. H. M. Abdulwahab, N. Moser, J. Rodriguez-Manzano, and P. Georgiou, "A cmos bio-chip combining ph sensing, temperature regulation and electric field generation for dna detection and manipulation," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351342.

[23] K. Smagulova, O. Krestinskaya, and A. P. James, "A memristor-based long short term memory circuit," *Analog Integrated Circuits and Signal Processing*, vol. 95, pp. 467–472, 2018.

[24] K.-H. Kim, S. Gaba, D. Wheeler, *et al.*, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2012.

[25] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on fpga," *arXiv preprint arXiv:1511.05552*, 2015.

[26] Z. S. Zaghloul and N. Elsayed, "The fpga hardware implementation of the gated recurrent unit architecture," in *SoutheastCon 2021*, 2021, pp. 1–5. DOI: 10.1109/SoutheastCon45413.2021.9401819.

[27] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2015, pp. 111–118.

[28] H. Chen, L. Jiang, Y. Luo, *et al.*, "A cordic-based architecture with adjustable precision and flexible scalability to implement sigmoid and tanh functions," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180864.

[29] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.

[30] J. S. Walther, "The story of unified cordic," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 25, no. 2, pp. 107–112, 2000.

[31] X. Hu, R. Harber, and S. Bass, "Expanding the range of convergence of the cordic algorithm," *IEEE Transactions on Computers*, vol. 40, no. 1, pp. 13–21, 1991. DOI: 10.1109/12.67316.