University of Arkansas, Fayetteville
ScholarWorks@UARK

Theses and Dissertations

5-2021

Low-Power and Reconfigurable Asynchronous ASIC Design Implementing Recurrent Neural Networks

Spencer Nelson
University of Arkansas, Fayetteville

Follow this and additional works at: https://scholarworks.uark.edu/etd

Part of the Artificial Intelligence and Robotics Commons, Computer and Systems Architecture Commons, Digital Circuits Commons, and the OS and Networks Commons

Citation
Nelson, S. (2021). Low-Power and Reconfigurable Asynchronous ASIC Design Implementing Recurrent Neural Networks. *Theses and Dissertations* Retrieved from https://scholarworks.uark.edu/etd/4033

This Dissertation is brought to you for free and open access by ScholarWorks@UARK. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks@UARK. For more information, please contact ccmiddle@uark.edu.

Low-Power and Reconfigurable Asynchronous ASIC Design Implementing Recurrent Neural Networks

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Engineering

by

Spencer Nelson
University of Arkansas
Bachelor of Science in Computer Engineering, 2016

May 2021
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council.

____
Jia Di, Ph.D.
Dissertation Director

____
James P. Parkerson, Ph.D.
Committee Member

____
Qinghua Li, Ph.D.
Committee Member

____
Jingxian Wu, Ph.D.
Committee Member

ABSTRACT

Artificial intelligence (AI) has experienced a tremendous surge in recent years, resulting in high demand for a wide array of implementations of algorithms in the field. With the rise of Internet-of-Things devices, the need for artificial intelligence algorithms implemented in hardware with tight design restrictions has become even more prevalent. In terms of low power and area, ASIC implementations have the best case. However, these implementations suffer from high non-recurring engineering costs, long time-to-market, and a complete lack of flexibility, which significantly hurts their appeal in an environment where time-to-market is so critical. The time-to-market gap can be shortened through the use of reconfigurable solutions, such as FPGAs, but these come with high cost per unit and significant power and area deficiencies over their ASIC counterparts. To bridge these gaps, this dissertation work develops two methodologies to improve the usability of ASIC implementations of neural networks in these applications.

The first method demonstrates a method for substantial reductions in design time for asynchronous implementations of a set of AI algorithms known as Recurrent Neural Networks (RNN) by analyzing the possible architectures and implementing a library of generic or easily altered components that can be used to quickly implement a chosen RNN architecture. A tapeout of this method was completed using as few as 112 hours of labor by the designer from RNN selection to a DRC/LVS clean chip layout ready for fabrication.

The second method develops a flow to implement a set of RNNs in a single reconfigurable ASIC, offering a middle ground between fully reconfigurable solutions and completely application-specific implementations. This reconfigurable design is capable of representing thousands of possible RNN configurations in a single IC. A tapeout of this design was also completed, with both tapeouts using the TSMC 65nm bulk CMOS process.

TABLE OF CONTENTS

1 Introduction 1

1.1 Problem Statement 1

1.2 Dissertation Statement 1

1.3 Dissertation Organization 2

2 Background 3

2.1 Multi-Threshold NULL Convention Logic (MTNCL) 3

2.2 Recurrent Neural Networks (RNNs) 6

3 Quick Put-Together Research Work 11

3.1 RNN Architecture Analysis 11

3.2 Generic Units 17

3.3 Non-Generic Units 28

3.4 Case Study – Four Layer FastGRNN 32

4 Reconfigurable RNN Research Work 35

4.1 Reconfigurable RNN Design 35

4.2 Reconfigurable Unit Adaptations 36

4.3 Reconfigurable MTNCL Completion Detection 42

4.4 Reconfigurable RNN Top Level Design 44

5 Results and Analysis 50

5.1 Quick Put-Together RNN Results 50

5.2 Reconfigurable RNN Results 52

6 Conclusion and Future Work 55

7 References **57**

LIST OF TABLES

Table 1: Fundamental NCL gates [2] 4

Table 2: LSTM Equations 8

Table 3: GRU Equations 9

Table 4: FastGRNN Equations 10

Table 5: Sample Block Circulant Matrix 27

Table 6: Reconfigurable Variables 36

LIST OF FIGURES

Figure 1: MTNCL Pipelined Architecture [1] 6

Figure 2: Basic Element-Wise Implementation 13

Figure 3: Basic Vector-Matrix Multiply Implementation 15

Figure 4: Generic Element-Wise Component 19

Figure 5: Generic Multiply-Accumulate Component 21

Figure 6: Vector Spooling Component 22

Figure 7: Vector Un-Spooling Component 25

Figure 8: SRAM and ROM Control 30

Figure 9: FastGRNN Gate Architecture 33

Figure 10: FastGRNN Layer Architecture 33

Figure 11: Case Study Top Level Architecture 34

Figure 12: Reconfigurable Adder Architecture 37

Figure 13: Reconfigurable Vector Spooling Architecture 38

Figure 14: Reconfigurable Vector Un-Spooling Architecture 39

Figure 15: Reconfigurable SRAM 41

Figure 16: Reconfigurable MTNCL Completion Detection 44

Figure 17: Reconfigurable Layer – Worst Case 45

Figure 18: Reconfigurable Layer – GRU Only 46

Figure 19: Reconfigurable Layer – Final 47

Figure 20: Complete Reconfigurable Design 49

Figure 21: Quick Put-Together Physical Layout 50

Figure 22: Reconfigurable Physical Layout 52

1 Introduction

1.1 Problem Statement

With the huge rise in the utilization of machine learning algorithms, the varying demands for the implementations of such algorithms encompass a wide array of possibilities. These demands include reducing the power consumption, area, and cost per unit requirements to fit into Internet-of-Things (IoT) applications, as well as improving the range of usability of a single implementation through reconfigurable solutions. A fully reconfigurable solution would be the utilization of Field Programmable Gate Arrays (FPGAs); but FPGAs come with significant costs in the aspects of power, area, and cost per unit. Application Specific Integrated Circuit (ASIC) implementations would better fit the requirements of IoT devices, but generally come with high non-recurring engineering costs, longer time-to-market, and lack of flexibility.

Ideal solutions to these implementation requirements would include either rapidly developed low power ASICs or some middle ground between FPGAs and ASICs which would allow for low area and power while also giving some degree of reconfigurability.

1.2 Dissertation Statement

This dissertation research focuses on the development of two methodologies for improving the design and implementation feasibility of neural network applications on ASICs, utilizing gated Recurrent Neural Networks (RNNs) to demonstrate the methodologies. The Quick Put-Together (QPT) methodology demonstrates the analysis and implementation of RNNs with asynchronous generic components, reducing non-recurring engineering cost for the development of ASIC implementations and the time-to-market for similar designs. The other methodology demonstrates the methods for the development of a reconfigurable RNN architecture, utilizing reconfigurable ASICs as a feasible middle ground between fully reconfigurable solutions such as

FPGAs and completely fixed ASICs for neural network applications. Both methodologies utilize the low power, asynchronous Multi-Threshold NULL Convention Logic (MTNCL) design paradigm.

1.3 Dissertation Organization

This dissertation provides the necessary background knowledge for RNNs and MTNCL design theory before presenting the methods for developing the QPT and reconfigurable methods. Chapter 2 details background information regarding MTNCL design as well as RNN concepts. Chapter 3 describes the QPT methodology and covers the implementation of a sample RNN utilizing the QPT methods. Chapter 4 discusses the development and implementation of a reconfigurable RNN covering a large set of possible RNN configurations. Chapter 5 presents and analyzes the results of the completion of a full chip tape-out of both the QPT and reconfigurable designs. Finally, Chapter 6 summarizes the concepts and lessons learned from this dissertation research and discusses possible future work.

2 Background

2.1 Multi-Threshold NULL Convention Logic (MTNCL)

Multi-Threshold NULL Convention Logic (MTNCL) [1] is a quasi-delay insensitive (QDI) asynchronous design methodology which combines the principles of NULL Convention Logic (NCL) [2] with multi-threshold power gating. NCL, the precursor to MTNCL, is a QDI design paradigm that uses a dual rail encoding scheme and a set of fundamental NCL gates known as "threshold gates" to complete self-timed operations. NCL gates are defined by a threshold value and the number of inputs. The threshold and number of inputs are used to denote each gate through the naming convention, formatted as "THmn" where *n* is the number of inputs and *m* is the threshold. The threshold denotes how many of the inputs must be asserted before the output of the gate will be asserted. For example, a TH34 gate requires 3 of the 4 inputs to be asserted at the minimum for the output to assert. This can be any of the three inputs and will also assert when all four inputs are asserted, as the threshold has been met in any of these cases.

Additionally, NCL gates can have weights associated with its inputs, included in the naming convention by appending a "*w*", followed by any weights above 1, starting with input *A* and proceeding in order. For example, input *A* in the TH33w2 gate has a weight of 2. Asserting input *A* (weight 2) and either input *B* or *C* (weight 1 each) would assert the output of this gate by meeting the threshold of 3 (2+1). Table 1 lists the 27 fundamental NCL gates, the set of gates utilized to form any NCL design. One important characteristic of NCL gates is that once an output is asserted, all inputs must be de-asserted for the output to de-assert. This functionality, known as hysteresis, is essential to NCL and is one of the key differences between NCL and MTNCL gates. MTNCL gates do not require hysteresis, instead having a "*sleep*" input on each gate that forces the output to de-assert.

Table 1: Fundamental NCL gates [2]

NCL Gate	Boolean Function
TH12	*A + B*
TH22	*AB*
TH13	*A + B + C*
TH23	*AB + AC + BC*
TH33	*ABC*
TH33w2	*AB + AC*
TH14	*A + B + C + D*
TH24	*AB + AC + AD + BC + BD + CD*
TH34	*ABC + ABD + ACD + BCD*
TH44	*ABCD*
TH24w2	*A + BC + BD + CD*
TH34w2	*AB + AC + AD + BCD*
TH44w2	*ABC + ABD + ACD*
TH34w3	*A + BCD*
TH44w3	*AB + AC + AD*
TH24w22	*A + B + CD*
TH34w22	*AB + AC + AD + BC + BD*
TH44w22	*AB + ACD + BCD*
TH54w22	*ABC + ABD*
TH34w32	*A + BC + BD*
TH54w32	*AB + ACD*
TH44w322	*AB + AC + AD +BC*
TH54w322	*AB + AC + BCD*
THxor0	*AB + CD*
THand0	*AB + BC + AD*
TH24comp	*AC + BC + AD + BD*

The set of fundamental gates also applies to MTNCL, but due to removing the hysteresis condition and having the *sleep* signal in each gate, the gates have slight differences to their internal logic, resulting in a separate set of fundamental MTNCL gates with the same naming convention, followed by the letter *m* to denote that the gate is an MTNCL gate, such as a TH34m gate. NCL and MTNCL utilize a dual-rail data encoding scheme. Designing logic within the dual-rail scheme utilizes two wires, referred to as rails. These rails are denoted as $RAIL^0$ and $RAIL^1$, representing a single logical bit *D*. By selecting which of $RAIL^0$ and $RAIL^1$ are asserted or de-asserted, D is made to represent any value from the set {DATA0, DATA1, NULL}. When

$RAIL^0$=1, $RAIL^1$=0, this corresponds to the state DATA0, which is equivalent to logic 0 or Boolean FALSE in single-rail paradigms. When $RAIL^0$=0, $RAIL^1$=1, this corresponds to the state DATA1, which is equivalent to logic 1 or Boolean TRUE in single-rail paradigms. *D* enters a NULL state when $RAIL^0$ and $RAIL^1$=0 meaning the value of *D* is not yet available. The state $RAIL^0=1$, $RAIL^1$=1 is not used in logic and should never occur, so it is considered an illegal state. This also means that $RAIL^0$ and $RAIL^1$ must be mutually exclusive in NCL/MTNCL logic.

NCL and MTNCL accomplish pipelining via localized handshaking and alternating DATA/NULL wavefronts. In NCL, a DATA wave is always followed by a full NULL spacer, to ensure that all gates can return to zero. This NULL spacer must propagate through all the gates in the pipeline stage such that all the outputs of the stage return to NULL. In MTNCL, this complete NULL spacer is somewhat replaced by putting all gates in a pipeline stage into the sleep state. This maintains the characteristic alternating DATA/NULL waves of NCL but reduces the impact of the NULL waves by eliminating the time needed to propagate the NULL spacer through all the gates in the stage. It also makes use of the multi-threshold aspect of the MTNCL gates. The sleeping of the MTNCL pipeline stages is accomplished through handshaking signals generated by completion detection components, which check the input to a pipeline stage to ensure the complete set of inputs have all reached DATA. These handshaking signals are used in a specific pipelining architecture. This pipelining architecture, called Slept Early Completion Register Input-Incomplete (SECRII), is shown in Figure 1.

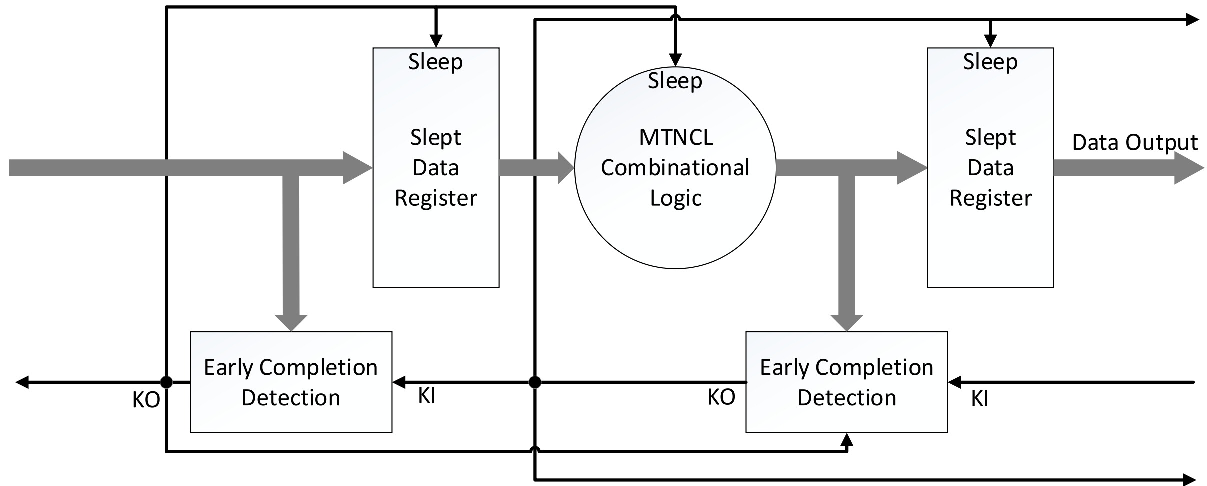

Figure 1: MTNCL Pipelined Architecture [1]

In the figure, each combinational MTNCL logic unit is followed by an associated slept early completion detection unit and MTNCL register. The associated completion detection unit has an output signal *ko* which is responsible for three functions: requesting for DATA/NULL from the previous completion detection unit and its associated register/pipeline stage, sleeping its associated register, and sleeping the next set of combinational logic. In this way, when the *ko* for a stage is 1, the current register has not yet received a complete DATA wave, so it and the logic that would receive data from that register are slept, reducing the power consumption while the logic is not in use while also giving NULL to the next pipeline stage. When the *ko* is 0, the completion unit is requesting for NULL, informing the preceding completion detection unit that the data in the preceding register is no longer needed, and unsleeping the current register and subsequent logic, so that the next stage's computation can begin. To simplify future figures, the completion detection components will be included as part of the MTNCL registers, and the handshaking and sleep signals will be omitted unless there is a particular complication to how they operate.

2.2 Recurrent Neural Networks (RNNs)

Recurrent neural networks are a form of neural network [3] that primarily operate on sequences of data, adjusting computation over time based on the results from each value in the sequence of data. This adjustment over time is done by maintaining a hidden state, referred to as *h,* which is both the output and used as an input for the next calculation. The main input, referred to as *x,* is a sequence of inputs, one for each timestep *t.* The result is generally used for predictive or classification functions common in machine learning fields. Both *h* and *x* are vectors which vary in length, but *x* in particular can vary in two aspects – the length of the value at each timestep and the total number of values, equal to the total number of timesteps. At each timestep, the RNN receives a new input value x_t, and a new hidden state h_t is calculated as $h_t = f(x_t, h_{t-1})$, where *f* is some non-linear activation function and h_{t-1} is the previous timestep's hidden state (with some initial value for $t=0$). Although this function can be as simple as an element-wise sigmoid function, the RNNs discussed in this dissertation are larger, with similar architectures to Long Short-Term Memory (LSTM) [4]. After some number of timesteps, the hidden state can be taken as output from the RNN or followed by a small additional computation, after which all hidden states are reset to their initial state.

LSTM is an early form of a class of RNNs known as gated RNNs. Gated RNNs are primarily constructed from their namesake gates, which are discrete mathematical units. A typical gate equation would be $g_t = \sigma(W_g x_t + U_g h_{t-1} + b_g)$. The values of W and U are "weight" values and the values of *b* are "bias" values. In neural networks, weight and bias values are trained utilizing another system for deciding what values perform better for different classification and prediction objectives and can be changed with additional training data and results. The non-linear function in the example case, σ, can also be changed out for a variety of other non-linear functions, such as the hyperbolic tangent function, *tanh.* While RNNs can be

made of a single gate, many architectures use multiple gates. If there are multiple gates in an RNN, there will also be additional weight matrices and bias vectors, as well as additional computations used to calculate the hidden state from the results of the gates. Table 2 describes an LSTM with four gates (*f, i, o,* $\tilde{c}$), and the element-wise operations that result in the hidden state.

Table 2: LSTM Equations

$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$
$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$
$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$
$\tilde{c}_t = \tanh(W_f x_t + U_f h_{t-1} + b_f)$
$c_t = f_t \bullet c_{t-1} + i_t \bullet \tilde{c}_t$
$h_t = o_t \bullet \tanh c_t$

The size of the data required to compute a result is a common complicating factor for neural networks. In the case of LSTM in Table 2 and RNNs like it, the *U* matrix is a two-dimensional matrix, where both dimensions are equal to the size of h_t. The *W* matrix is also a two-dimensional matrix, but with one dimension equal to the size of x_t and the other equal to the size of h_t. The bias vectors are also the size of h_t. Overall, the number of weights can be calculated as $N = g(|h_t|^2 + |h_t| * |x_t| + |h_t|)$ where *g* is the number of gates. Additionally, individual RNNs can be chained to one another in sequence to form arbitrarily long networks composed of multiple RNNs, with each independent RNN comprising a single "layer" of a the overall RNN. The number of layers can increase prediction and classification performance at a significant cost to data size, training time, and computation time. The selection of these

parameters (e.g., number of layers, layer types, hidden state vector length, input vector length) greatly alter the overall functionality, as well as the efficacy, of the RNN as a whole.

Since LSTM was developed, several attempts have been made to reduce the size of the RNNs while improving the accuracy or reducing the training time. Two of these attempts include GRU [5] and FastGRNN [6], which were selected as the primary focus of the reconfigurable portion of this dissertation in Chapter 4.

The equations that make up the GRU algorithm are shown in Table 3. The objective of GRU was to create a method to eliminate short-term or irrelevant data from the hidden state while preserving long-term data dependencies by adaptively resetting or updating the current hidden state. This is done by having two primary gate units, the reset and update gates. The reset gate is intended to act as a computational switch, either allowing or disallowing the previous hidden state to contribute to the new hidden state. This is intended to resolve the short-term dependencies by removing no longer relevant information from the new hidden state. The reset gate is unique in terms of the gates in LSTM due to it being composed of two gate units connected in series rather than all in parallel. The update gate is intended to capture the longer-term dependencies by allowing some amount of the previous hidden state to affect the current hidden state. GRU has been shown to outperform LSTM in the majority of applications [7].

Table 3: GRU Equations

$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$
$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$
$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \bullet h_{t-1}) + b_h)$
$h_t = z_t \bullet h_{t-1} + (1 - z_t) \bullet \tilde{h}_t$

FastGRNN attempts to minimize the overall size of the network while still maintaining the accuracy of the network. Table 4 shows the equations of a FastGRNN network, which contains only two gates. The primary aspect of FastGRNN over other gated RNN architectures is that the *W* and *U* matrices are actually shared between the two gates, reducing the data size, as well as the computational cost, significantly. To reduce the size even further, the original FastGRNN work also discusses a version utilizing a byte quantization methodology, reducing the weights in *U* and *W* to a single byte in length. Overall, FastGRNN is capable of having a significantly smaller footprint that still matches or exceeds the prediction accuracies of LSTM or GRU [6].

Table 4: FastGRNN Equations

$z_t = \sigma(Wx_t + Uh_{t-1} + b_z)$
$\tilde{h}_t = \tanh(Wx_t + U\, h_{t-1} + b_h)$
$h_t = z_t \bullet h_{t-1} + (\zeta(1 - z_t) + \nu) \bullet \tilde{h}_t$

3 Quick Put-Together Research Work

While ASICs generally have the best-case performance, smallest size, lowest power consumption, or some combination of the three, their development is subject to long time-to-market and high non-recurring engineering costs. The first principal contribution of this dissertation is an analysis of gated RNN architectures which facilitates the rapid implementation of asynchronous gated RNNs.

In order to best facilitate the rapid construction of fully custom ASIC RNN configurations, this work makes use of the "correct-by-construction" aspect of MTNCL to form a library of generic reusable components that can be utilized to implement a wide variety of RNN configurations. Since MTNCL is asynchronous and quasi-delay insensitive, there is no need to verify timing requirements, which is a substantial concern and time cost in the development of synchronous ASIC designs. The primary objective is to make as much of the design generic as possible, while making the non-generic portions be easily altered or extended. The completion of this objective allows for drastic reductions in design time for the logic implementation of ASIC RNNs.

3.1 RNN Architecture Analysis

RNN implementations are composed of a set of component operations and dataflow, called the architecture, which dictates the computation performed to produce the hidden state. Example architectures include the LSTM, GRU, and FastGRNN algorithms. While the architecture describes the operations, there are several variables that the primary architecture does not specify. These variables are defined as the network's configuration. While additional architectures may contain other configuration variables, an analysis of the equations of the

LSTM, GRU, and FastGRNN algorithms, as well as the ability to add additional layers in series, yields five configuration variables:

1. The number of individual RNNs connected serially, referred to as layers
2. The architecture of each layer (LSTM, GRU, or FastGRNN)
3. The size of the hidden state of the layers
4. The size and number of input vectors of the layers
5. The data representation used in each layer (bitwidth of data, fixed-point or floating-point number representation, etc.)

This list of variables is used to determine which of these variables apply to the reusable components designed as part of establishing the QPT methodology for the architectures, so that they can be made generic, or at least easily altered manually, with respect to that variable. Looking at the equations presented in Tables 2, 3, and 4, it can be seen that these architectures consist of five mathematic operations:

1. Vector-Matrix Multiplication
2. Element-Wise Addition
3. Element-Wise Non-Linear Function (implemented as a look-up table in ROM)
4. Element-Wise Multiplication
5. "1-Minus" Operation (implemented as Element-Wise Subtraction)

With the exception of the vector-matrix multiplication, each of these operations are element-wise operations, performing the same operation on each value of a set of input vectors (just one input vector in the case of the non-linear function). In every case of these operations, the width of the vector(s) being operated on is equal to the size of the hidden state vector. These operations can most easily be implemented with $|h_t|$ copies of the respective individual operation.

For example, element-wise addition could be implemented with 128 parallel adders in the case where the size of the hidden state is 128. This solution is the fastest both in speed and time to implement, but also easily the least efficient, in terms of area and power, of handling these operations and should be reserved for applications where area is no issue and speed must take absolute precedence. For applications where area and power are greater concerns, an optimization solution is necessary and has been developed. Since element-wise operations perform the same operation on each value of the input vectors, they can be implemented as single-element operations that accept a streaming input, operating on each element in the vector one at a time. The assumption of a streaming input allows the design to reduce each component to their simplest version, e.g., a single adder for element-wise addition. This solution also allows for parallelism, allowing a balance to be struck between speed, area, and power for the chosen application. Figure 2 shows the basic implementation for these element-wise operations.

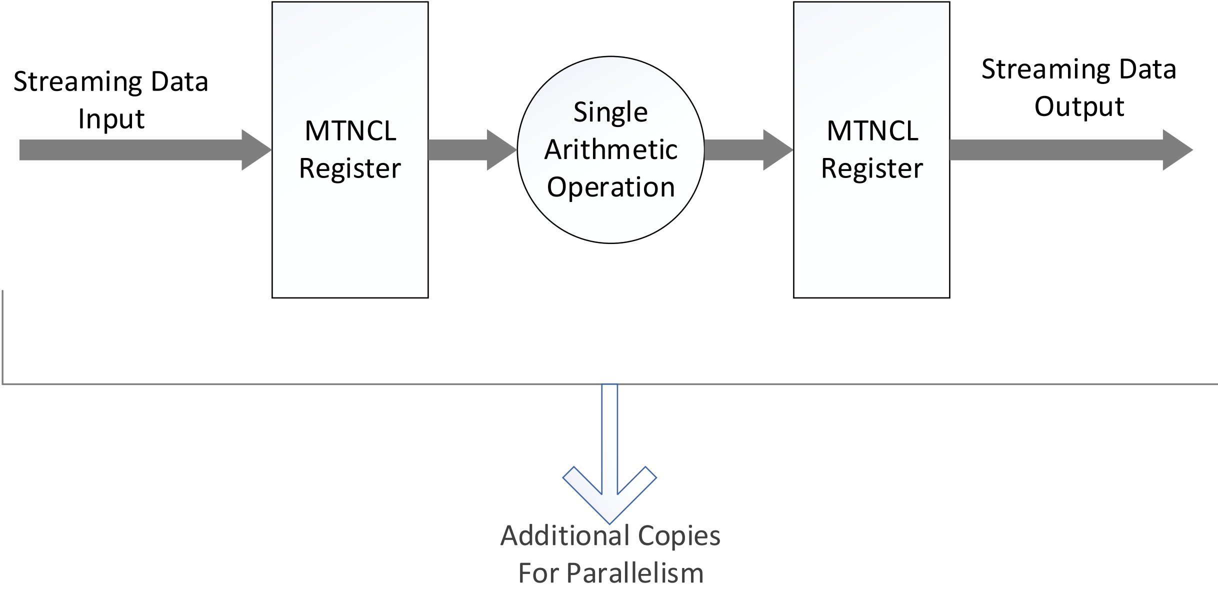

Figure 2: Basic Element-Wise Implementation

The assumption of a streaming input for the element-wise operations significantly simplifies them; but to satisfy this condition, a unit must be capable of producing these streaming

values as its output. The vector-matrix multiplication, the most complex unit in the list, is adapted to this purpose. The nature of vector-matrix multiplication is that each row of the matrix operates with the vector in the same way, multiplying the values of the row with the elements of the vector in an element-wise fashion and summing the results. This means that each row of the matrix produces exactly one value of the result vector. With this in mind, vector-matrix multiplication can be implemented as a single Multiply-Accumulate unit, iterating over the entire vector once for each row of the matrix, producing a single result at a time, creating the streaming output needed to satisfy the streaming input assumption of the element-wise units. This implementation also maintains the possibility for parallelism to match with the element-wise units if area and power constraints allow for it. Figure 3 illustrates the basic implementation of the vector-matrix multiplication operation.

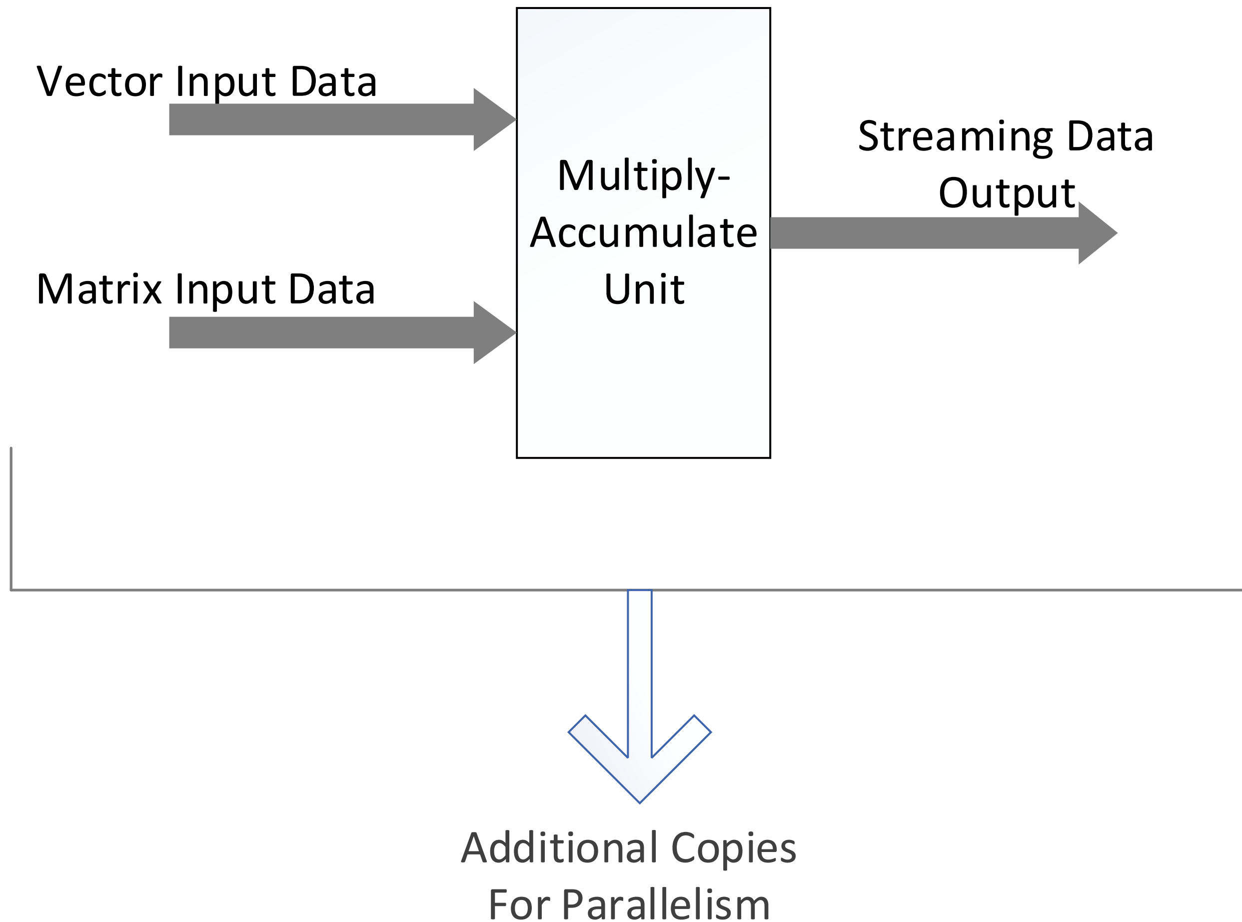

Figure 3: Basic Vector-Matrix Multiply Implementation

Due to its nature, the vector-matrix multiplication does require the entire vector to be present before any output values can be produced. Additionally, since the weight matrices represent a significant amount of data which is operated on many times, having the weight data come from off-chip represents a very high I/O cost. Instead, the values are loaded one time and stored on-chip, which also has its own significant cost. These factors lead to a set of five data management components. These components either store or control the flow of data, but do not perform any operations on the data. These components include:

1. Vector Spooling Unit
2. Vector Un-Spooling Unit

3. Weight Storage (SRAM)
4. SRAM Read Unit
5. Data Loading Unit

The Vector Spooling Unit, also called the Vector Spooling Register, is responsible for collecting all the values of a vector into a single register, so that the whole vector can be output at the same time. This is necessary due to adapting the element-wise operations into streaming operations. The primary use of this component is converting the streaming output from the element-wise units into the full final hidden state vector at the output of a layer. It can also be used in cases such as the GRU, where the reset gate's characteristic pair of gates in sequence require the completion of the entirety of the computation of the first gate before the second gate can produce any values.

The Vector Un-Spooling Unit, also called the Vector Un-Spooling Register, signifies the reverse of the Vector Spooling Unit, taking in a full vector of values as input, and producing the values of the vector one at a time in order. This component's primary purpose is to facilitate the usage of the input and previous hidden state vectors as the input to the Multiply-Accumulate components that perform the vector-matrix multiplication. Besides its primary purpose, it is also used in cases where the previous hidden state vector is used as part of an element-wise operation to convert the hidden state vector into a streaming input to the element-wise unit. To further clarify their function, the Vector Spooling and Un-Spooling units combined in that order would form a first-in first-out (FIFO) queue but are separated into two units for cases where a vector needs to be spooled but not immediately used in a queue-like manner or where different vectors need to be combined as part of the same output queue.

The weight storage is implemented as SRAM generated by the TSMC 65nm Memory Compiler tool. Due to this, the inner workings of the SRAM itself will not be discussed, but the function, size, number of units, and the method for including them into the design is covered in the following sections. Along with the SRAM comes a need to write the needed values to them as well as read the values from them, leading to the SRAM Read unit and the Data loading unit. In total, the analysis of these two algorithms resulted in ten components to analyze and implement as generic units where possible or in an easily extendable structure where not.

3.2 Generic Units

Of the ten units described in Section 3.1, seven of them are implemented as generic components. These are:

1. Vector-Matrix Multiplication
2. Element-Wise Addition
3. Element-Wise Multiplication
4. Element-Wise Subtraction
5. Vector Spooling Unit
6. Vector Un-Spooling Unit
7. SRAM Read Unit

While the element-wise non-linear function is not able to be made generic due to its implementation as a ROM, the implementation of the remaining computation units can be done generically. For these components, the computations occur entirely within the layer itself, and do not need to be sensitive to the number of layers or the architecture of the layer. This leaves only the size of the hidden state vector, size of the input vector, and the data representation to affect the units. Among these, the element-wise arithmetic functions are the easiest to make generic.

As described earlier, each element-wise component operates on a streaming input to form vector element-wise operations out of the simplest version of the computation. This serves the additional purpose of allowing each unit to be entirely unaffected by the sizes of the hidden state and input vectors. Instead, each of these units are only sensitive to the data representation. Making these arithmetic operations generic with respect to the bitwidth of the input values is already common in the field of digital design and is effectively trivial, but the numeric representation is more complex. The numeric representation refers to the way the bits of a binary value represent numeric values. Examples of different numeric representations include whether the data is signed or unsigned, if the values are integers or contain fractional values, and how the fractional values are represented such as fixed-point or floating-point formats. This aspect represents significant complexity, with the potential to add large amounts of logic to where might usually be a simpler function. In the QPT section of this dissertation, floating-point representations are not addressed and will be considered as part of the future work for this dissertation.

To enable the element-wise components to be generic with respect to the numeric representation, a generic sub-component was developed for each of the arithmetic operations which allows for integer or fixed-point, signed or un-signed values to be performed by appending the sub-component to the end of each arithmetic component. This unit, referred to as an Arithmetic Helper unit, is made generic by defining the output of the arithmetic components by how they need to be corrected. In fixed-point and integer systems, the number of bits and what each bit represents is fixed, allowing these units to correct arithmetic outputs by truncating the results into the correct output representation. Figure 4 shows the architecture for the complete generic element-wise operations.

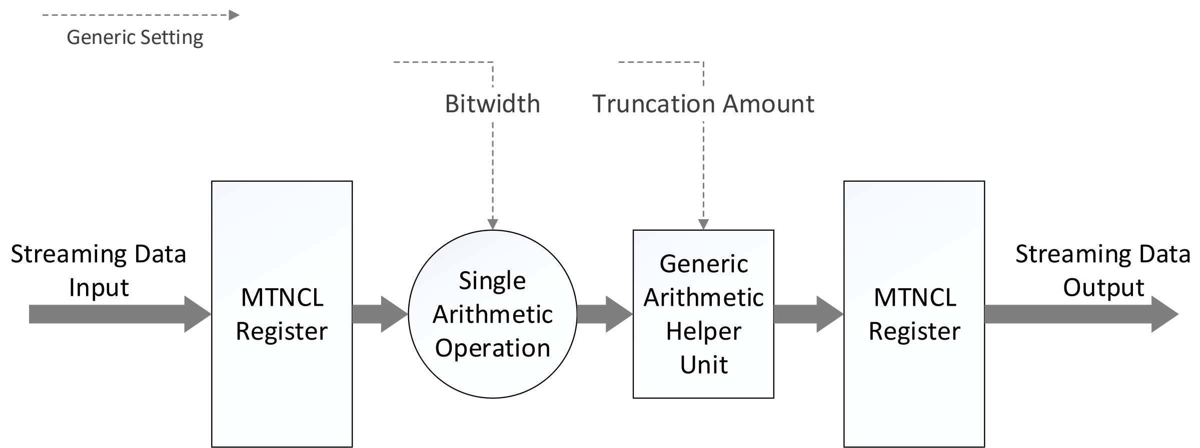

Figure 4: Generic Element-Wise Component

To demonstrate the function of the Arithmetic Helper unit, the example of the element-wise multiplier is used. This example multiplier is using the data representation of fixed-point 13-bit values with 3 integer bits and 13 fractional bits. The output of the multiplication will produce a 32-bit value with 6 integer bits and 26 fractional bits. The Arithmetic Helper must truncate the three most significant bits and the 13 least significant bits to return the value to the original data representation. This can be made generic by setting the size of the input and the number of bits to be truncated from each size. From a mathematical perspective, this also adds considerations for how to handle cases where the result of the multiplication is outside the range of what can be represented in the data representation. The decisions related to data overflowing their allocated data range must be handled at the algorithmic level and could change depending on the algorithmic requirements. In the case of this research, values that fall outside of the set of values that can be represented are set to the maximum value of the same sign as the unconverted result.

As it is not an element-wise function, the vector-matrix multiplication must not only be aware of the data representation for the multiplication but also the total number of values required for each element in the operation. By implementing the vector-matrix multiplication as

a Multiply-Accumulate unit, the generic multiplication unit covers the multiplication aspect, but additional logic is needed for the accumulation. The accumulate portion is covered by the generic adder unit along with registers to store the current accumulated value, and generic supporting logic that allows the unit to accumulate a programmable number of values before outputting the result and resetting the accumulated value. This generic supporting logic is another sub-component and is critical to several components in the overall design.

This sub-component is described as a Self-Resetting Counter, a generic counter which has two controlling input ports: a Maximum Counter Value and a Count Enable port. The Maximum Counter Value declares the highest values the counter will count to. Once this value has been reached, the counter will reset to zero, and also raise its output control signal, the Counter Reset signal, that declares that the counter has reached its maximum value, which can be used to enable the output of the Multiply-Accumulator and reset the accumulated value. It is important to note that the Maximum Counter Value is a set of ports and not a generic variable, so these ports are tied to constants on chip. This allows the resetting portion of the Self-Resetting Counter to be implemented with a simple comparator. The other controlling input port, the Count Enable port, is an input that decides whether or not the counter accumulates on each DATA cycle. In the Multiply-Accumulator, this value is tied to logic high such that the counter accumulates on every DATA cycle but is used in other components to only count under specific circumstances.

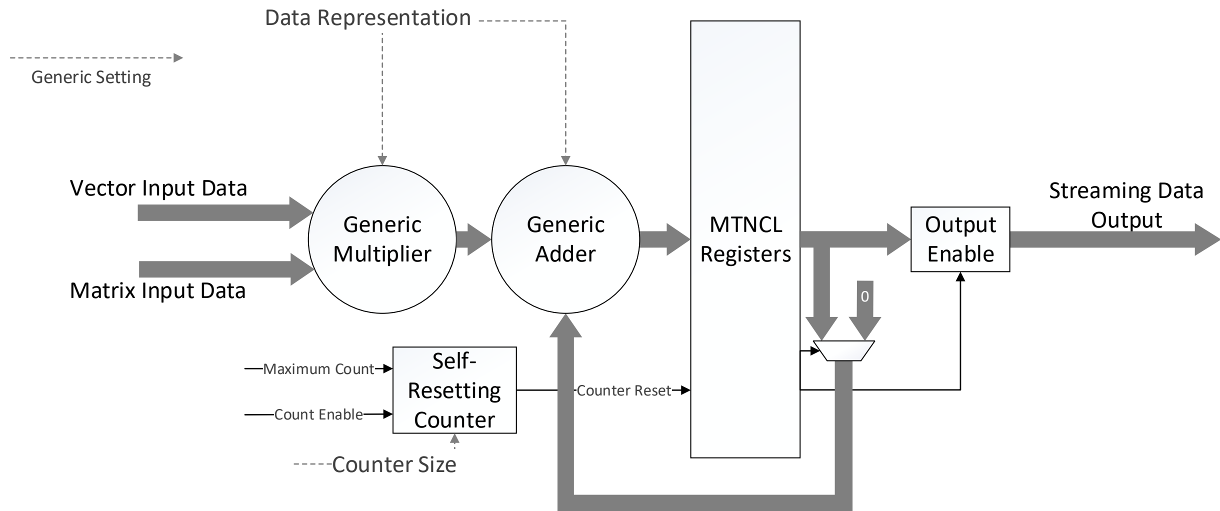

Figure 5: Generic Multiply-Accumulate Component

This counter represents one of the major design philosophies utilized in the QPT methodology, which is to localize as much control logic to the specific component requiring the control where possible. Control logic is often complex and difficult to make generic, so the localization of control is used to reduce the need for mid-level or global control units as much as possible. In this case, controlling the number of accumulations within the Multiply-Accumulate unit from within the component itself prevents the need for a higher-level control unit that monitors and controls the state of the Multiply-Accumulate components throughout the layer. This concept is further demonstrated in the data management components.

Of the five data management components, three can be made generic: The Vector Spooling, Vector Un-Spooling, and SRAM Read components. The two vector management components, the Vector Spooling and Un-Spooling units, have similar requirements due to the nature of their operations.

The Vector Spooling unit needs to be sensitive to the number of bits in each data value it will be storing and the total number of values in the vector being stored. Additionally, due to the Vector Spooling unit's use as the primary output unit of each layer, putting the final hidden state

into a single register as the output of the layer, this unit must also be capable of keeping track of the number of times it has been used, such that it can be aware of how many times the hidden state has been completed. This allows it to enable resetting the hidden state to its initial value once a set number of computations, specified by the total number of input vectors, have been completed. Figure 6 shows the main implementation details of the Vector Spooling unit. Note that the Hidden State Feedback with Reset logic is optional, and is only included when this unit is used in its main application as the primary output of an RNN layer.

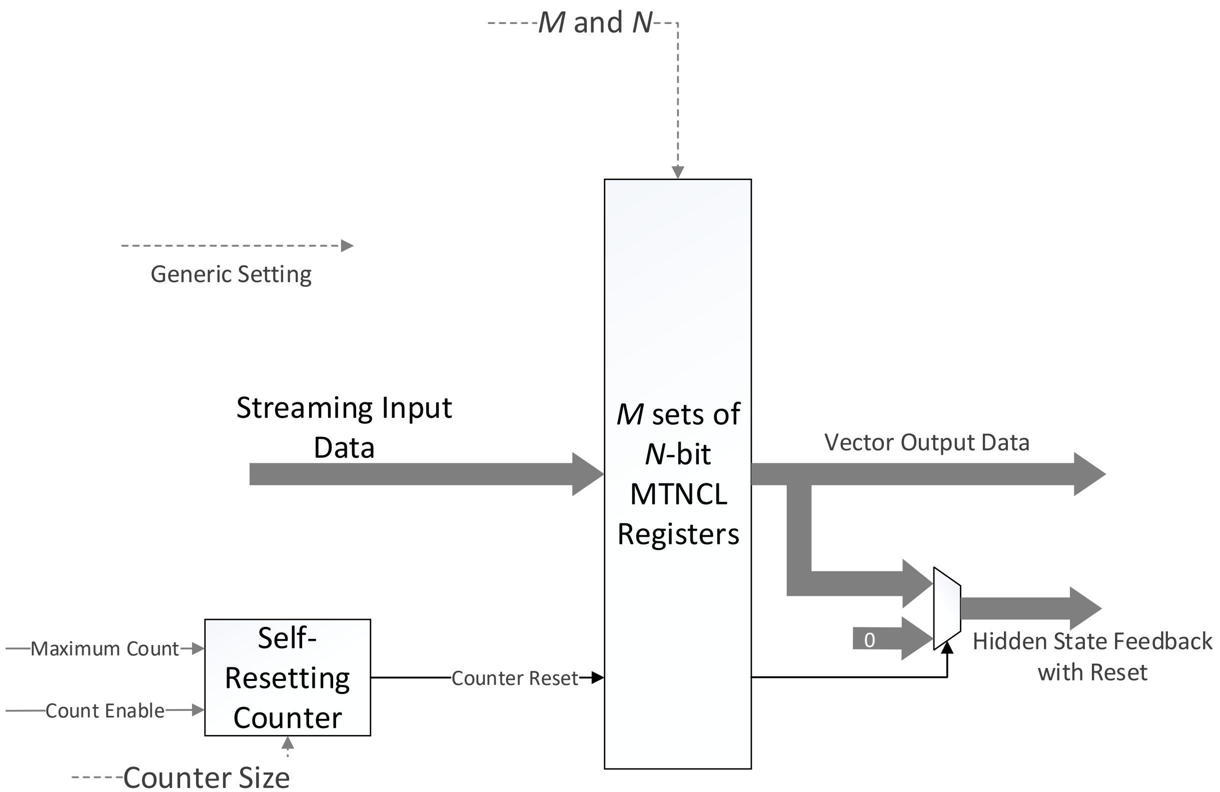

Figure 6: Vector Spooling Component

The Vector Un-Spooling unit needs to be sensitive to the number of bits in each data value it will be storing and the total number of values in the vector being stored, which is the same as the Vector Spooling unit. However, this unit is not responsible for resetting the hidden state, and so does not need to be aware of how many times it has been used. The Vector Un-Spooling unit

is used primarily to store a vector that represents the combined hidden state and input vectors and provide these vectors as the input to the Multiply-Accumulate units. As discussed earlier, the vector-matrix multiplication that the Multiply-Accumulate units represent requires that the vector be used in its entirety once for each row in the matrix. This means that they require each value in the combined vector to be provided in the same order a number of times equal to the size of the hidden state. Therefore, the Vector Un-Spooling unit must not only output the values in order but also be able to do so a programmable number of times.

Both the Vector Spooling and Vector Un-Spooling units utilize the Self-Resetting Counter sub-component previously discussed. For the Vector Spooling unit, there needs to be a counter keeping track of how many times the Vector Spooling unit has been used. Once it has reached the maximum number, the Counter Reset signal is used to reset the hidden state.

The Vector Un-Spooling unit has the more complex use case for the Self-Resetting Counter. The Vector Un-Spooling unit must keep track of two values that are incrementing over time. The first is the number of values that have been output for the current vector. The second is how many times the entire vector has been output. This is implemented with two copies of the Self-Resetting Counter, a lower level counter and an upper level counter. The lower level counter increments with every value that is output and resets once the total number of values in the stored vector has been reached. The upper level only counts each time the entire vector has been output. This case is where the functionality of the Count Enable control port is utilized. When the lower level count reaches its maximum value, its Counter Reset signal is raised. This Counter Reset signal for the lower level is used as the Count Enable signal for the upper level counter, causing the upper level counter to count once each time the lower level counter reaches its maximum value.

Unlike the Vector Spooling unit, which outputs a complete vector all at once, making its output function within the MTNCL pipeline as expected, the Vector Un-Spooling unit only outputs one of its values at a time. To perform this function, the lower level counter's value is used with a multiplexor, selecting the correct value to be output from the internal register with the lower level count until the count reaches the maximum value, at which point the upper level count increments by one. Once both counters have reached their maximum values, the Vector Un-Spooling unit can return to the NULL state as part of obeying the MTNCL pipeline architecture. Figure 7 shows the primary details of the Vector Un-Spooling unit's implementation. Because this unit adds complications to the MTNCL pipeline, the handshaking signal *KO* from the subsequent design unit is not omitted to illustrate its use to request DATA/NULL from the two counters, controlling the output of the register set, as well as its interaction with the output of the two counters to determine when the register set should receive a request for NULL.

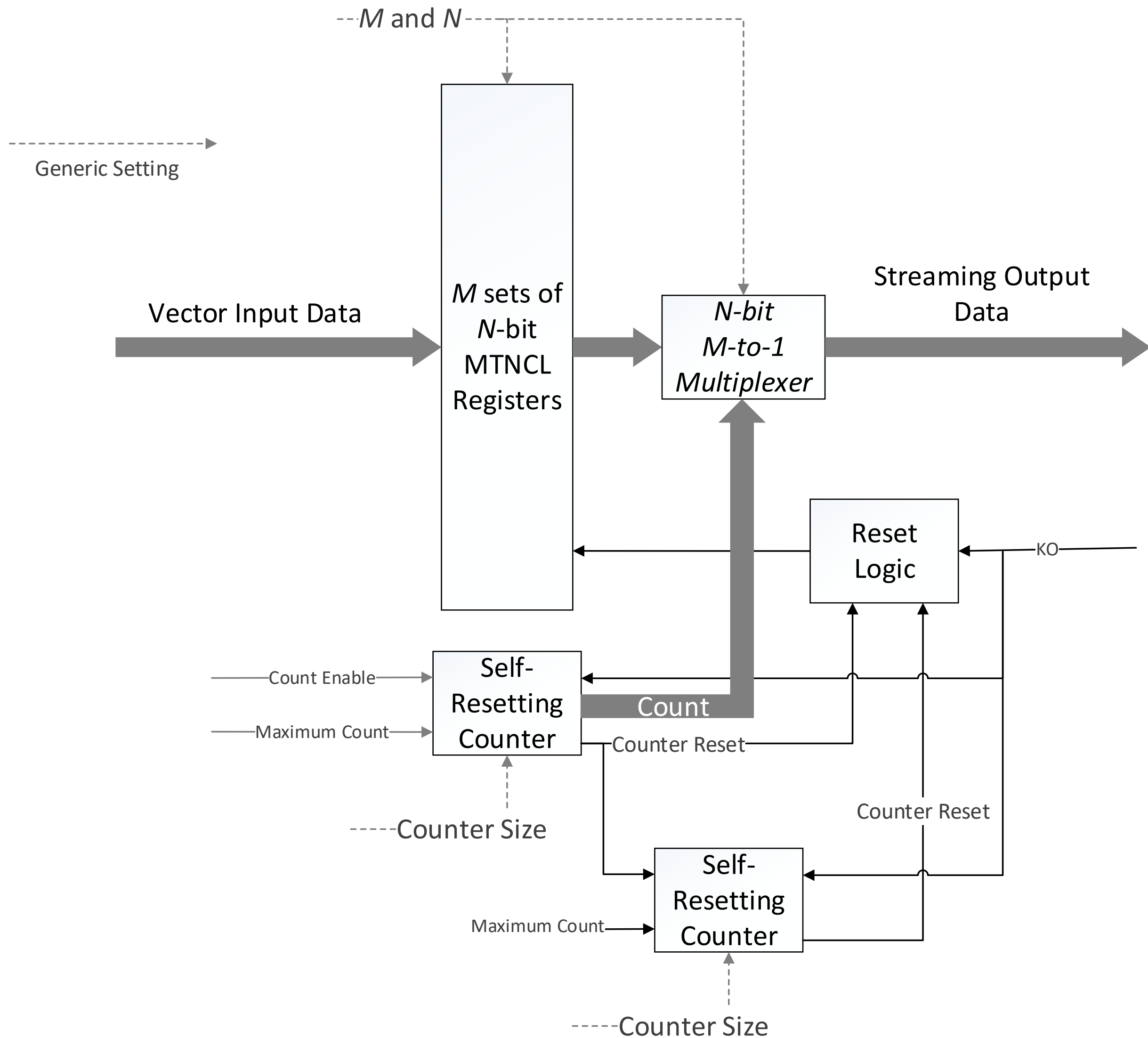

Figure 7: Vector Un-Spooling Component

To better clarify the functionality of the two connected counters described above, this two counter architecture would work as a single large counter in the case where the lower level counter's maximum value follows the form $2^n - 1$, resulting in a total number of values that are exactly a power of two; but due to the possibility of non-power of two values, the counters are split. The Vector Un-Spooling unit also adds the current counts (lower and upper) as an output, so that subsequent units can be aware of the current value being operated on, which greatly simplifies the SRAM Read Unit.

The SRAM Read Unit is responsible for ensuring the correct value is read out of SRAM for the current Multiply-Accumulate operation by generating the correct read address for the SRAM. It is important to note that this unit is distinct from the control of the SRAM itself, which is discussed as part of the SRAM implementation in Section 3.4. The address of the correct value is inherently dependent on the organization of the memory itself. In this case, an architectural decision needed to be made on how many individual memory units would be used to store the values. There are three main choices: single monolithic memory, single memory per layer, and individual memory per gate. The individual memory per gate was chosen, due to the significant reduction in complexity over the other two solutions, despite having the highest overhead. The reason the complexity is so much lower is due to the nature of vector-matrix multiplication. Each value in the matrix needs to be used exactly once to complete a full vector-matrix multiplication, so by localizing the matrix values to a single SRAM, the address generation can be as simple as a counter. The single monolithic memory would have resulted in the lowest power and least area overhead but with it comes a significant increase in the complexity of the control, as well as limiting the speed of the overall computation dramatically. The single memory per layer offers some tradeoffs between the two but adds significant complexity over the single memory per gate while also reducing the speed of each layer significantly. Given this choice, the SRAM Read unit can be as simple as using the count output from the preceding vector un-spooling unit for the address. However, the SRAM size is a significant limiting factor in terms of the size of the overall design, so methods for reducing the size of the matrices result in significant reductions in area and power, even if these methods add complexity to the SRAM Read component.

In this dissertation, a method for reducing the size of the weight matrices is utilized, dividing the number of values in most matrices by eight. This methodology, which makes the

weight matrices block circulant [8], reduces the size of the matrices by reusing weights in a specific pattern. In this implementation, only the first row out of every eight rows needs to be stored, replacing the values for the remaining seven rows with values from the stored row. This results in a substantial reduction in area and power expenditure, while actually maintaining the computational power of the network. Table 5 demonstrates how a 16×16 matrix is implemented in a block circulant fashion, utilizing only 32 values to implement the entire matrix. In the table, the first and eight rows would be actually stored in SRAM. Each of the four 8×8 squares in the matrix get their values by shifting the values from the previous row to the right, wrapping around to the beginning. The values 0-7 are repeatedly used in the top-left 8×8 block and the values 8-15 are repeatedly used in the top-right 8×8 block. The values 16-23 are used to form the bottom-left 8×8 block and the values 24-31 are used to form the bottom-right 8×8 block, resulting in a set of two 16 entry rows that can be used in place of a 16×16 matrix. Since the matrices in the actual RNN implementations are much larger, such as the 256×128 matrices used as part of the implementation discussed in Section 3.4, this reduction is extremely significant, especially in networks with multiple layers or gates per layer.

Table 5: Sample Block Circulant Matrix

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
7	0	1	2	3	4	5	6	15	8	9	10	11	12	13	14
6	7	0	1	2	3	4	5	14	15	8	9	10	11	12	13
5	6	7	0	1	2	3	4	13	14	15	8	9	10	11	12
4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11

Table 5 (Cont.)

3	4	5	6	7	0	1	2	11	12	13	14	15	8	9	10
2	3	4	5	6	7	0	1	10	11	12	13	14	15	8	9
1	2	3	4	5	6	7	0	9	10	11	12	13	14	15	8
16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
23	16	17	18	19	20	21	22	31	24	25	26	27	28	29	30
22	23	16	17	18	19	20	21	30	31	24	25	26	27	28	29
21	22	23	16	17	18	19	20	29	30	31	24	25	26	27	28
20	21	22	23	16	17	18	19	28	29	30	31	24	25	26	27
19	20	21	22	23	16	17	18	27	28	29	30	31	24	25	26
18	19	20	21	22	23	16	17	26	27	28	29	30	31	24	25
17	18	19	20	21	22	23	16	25	26	27	28	29	30	31	24

3.3 Non-Generic Units

As the SRAM is generated individually by a memory compiler for the exact size needed, it cannot be considered a generic unit but can be put together very quickly. The element-wise non-linear function units were also implemented as look-up tables in ROM, generated by the same memory compiler as the SRAM. The primary problem surrounding these units is the fact that the memory compiler used generates synchronous memory designs which also means these designs lack an output that denotes when an operation has been completed. This results in the need to develop a small amount of logic to act as a wrapper for the SRAM and the ROM such that they behave correctly as part of the MTNCL pipeline. This functionality is divided into two segments: Input Control and Output Control.

Input Control is responsible for providing the address, control signals, and data (in the case of writing to SRAM) to the SRAM or ROM before activating it in order to obey setup time restrictions. Output Control is responsible for ensuring enough delay occurs for the SRAM or ROM to complete its operation before the single-rail outputs are converted to dual-rail signals and output into the next stage of the MTNCL pipeline. The primary functionality in both cases is the addition of arbitrarily long delays to satisfy the setup time and execution time restrictions. These delays are introduced using buffer chains, which can be made arbitrarily long generically, so that a proper characterization of the delay per buffer allows the wrapper's delays to be generated to be longer than the worst-case setup time or execution time for the memory component.

To follow the MTNCL pipeline, these delays are introduced to the inverse of the *sleep* signal that would normally control the MTNCL logic for this pipeline stage. This way, when the pipeline stage containing the SRAM or ROM is unslept, the input values to the SRAM or ROM are ready and after the setup delay, the inverted sleep signal enables the memory. Then after the remaining delay for the worst-case execution time for the SRAM or ROM, the single-rail output of the memory is converted into dual rail signals and output to the next pipeline stage. The implementation of the SRAM/ROM with the necessary controls is shown in Figure 8.

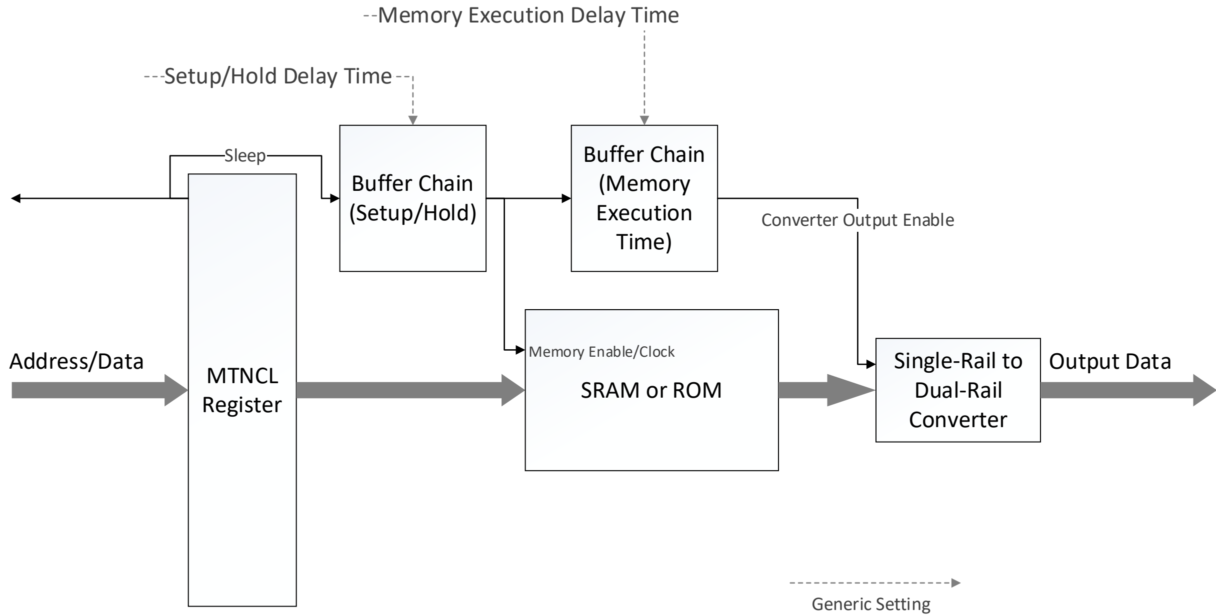

Figure 8: SRAM and ROM Control

Other than the memory components, the only non-generic unit is the Data Loading unit. The Data Loading unit can also be considered the only true global control unit and is the primary finite state machine (FSM) of the entire design. Its responsibility is to load all constant values for the RNNs configuration into the correct locations. The reason the Data Loading unit was chosen as non-generic is not that it is necessarily impossible, but because doing so would reduce its flexibility while requiring the highest amount of design time, complexity, and verification time of any of the components. For the Data Loading unit to work under the chosen distributed SRAM architecture, the Data Loading unit must be generic to all five aspects of an RNN configuration listed in Section 3.2, as these variables determine the total number of values and the destination of each of those values.

Putting the variables into the perspective of the Data Loading component, it must be aware of the number of layers, how many SRAMs are in each layer (layer architecture), how many values each SRAM should receive (size of the hidden state vector and the size of the input

vector), the bitwidth of the data. Since this unit is responsible for reading in all constant values to the chip, it must also be aware of notable exceptions, such as the two constant values in the FastGRNN layers, ζ and ν. It would also need to be able to handle possible reductions in the weight matrices, such as the block circulant method discussed previously. Instead of creating and verifying a generic design of this complexity, the solution chosen was to create an easily extendable unit that is generic to only the bitwidth of the data. To that end, the memory write unit is an FSM with $N+1$ states, where N is the total number of data storages in the design. Data storages include SRAMs as well as registers storing constants like ζ and ν. Each state except the final state handles exactly one data storage. Each state writes a programmable number of values to the location it is connected to before transferring control to the next state. The final state acts as a start signal, signaling that all the data is loaded, and the circuit can begin computation.

While it would be possible to make all of this into a generic unit, the primary reason this unit was not made generic is due to its nature as the only unit that is external to the layers of the RNN. In this design, it is assumed that due to I/O constraints, the overall design cannot afford to have an individual set of I/O ports dedicated to each data storage location. If that were the case, then a generic Data Loading unit would be extremely simple, and each data storage would be allocated a separate Data Loading unit. Under this assumption, this design utilizes a single set of I/O ports for all of the constant data to come through (connected to a flash memory unit in practice).

With only a single set of I/O ports for all constant data, there must exist a single design that is external to the individual layers able to route the data to the correct data storage locations. In addition to that, there are a few exceptions to the rule that this unit is in charge of handling. One example is the ζ and ν variables in the FastGRNN architecture. Each FastGRNN layer requires

two additional values not stored in the SRAM, instead being loaded into a register as a constant specific to the current set of weights. These values would also come through some external interface, so the Data Loading unit is also used for this data. Another example would be the Fully Connected Layer, an extra layer sometimes added to the end of an RNN configuration that reduces the final output down to a small number of values. In this dissertation, this is a single vector-matrix multiplication, with a number of rows in the matrix equal to the number of desired outputs. Since some unit must be aware of any exceptions like these, it was decided that the most effective solution would be to create an easily extendable FSM, rather than an extremely complex fully generic unit that may not be able to adapt to all exceptions.

3.4 Case Study – Four Layer FastGRNN

To demonstrate the Quick Put-Together methodology, an RNN configuration was selected, designed, and implemented as an MTNCL ASIC. The chosen configuration was a four-layer RNN, where each layer utilized the FastGRNN architecture. Each layer is defined with a hidden vector size of 128 values, and the input size of the initial layer is 1 value while the rest utilized the 128-value provided from the hidden state of the preceding layer. The data was represented as 16-bit fixed-point values.

As discussed earlier, one of the key points of the FastGRNN algorithm is that the weight matrices are shared between the two gates, with only biases and non-linearity being different. The architecture for each layer is identical, with only the initial layer differing in regard to the size of the weight matrices and number of values used in the vector-matrix multiplication. Figure 9 shows the architecture for the FastGRNN gate and Figure 10 shows the architecture for the layers. Figure 9 represents a FastGRNN gate, where a standard gate in LSTM or GRU

architectures would contain only one of the non-linear units (σ or *tanh*) as well as only one of the adder and Vector Un-Spooling units responsible for the biases.

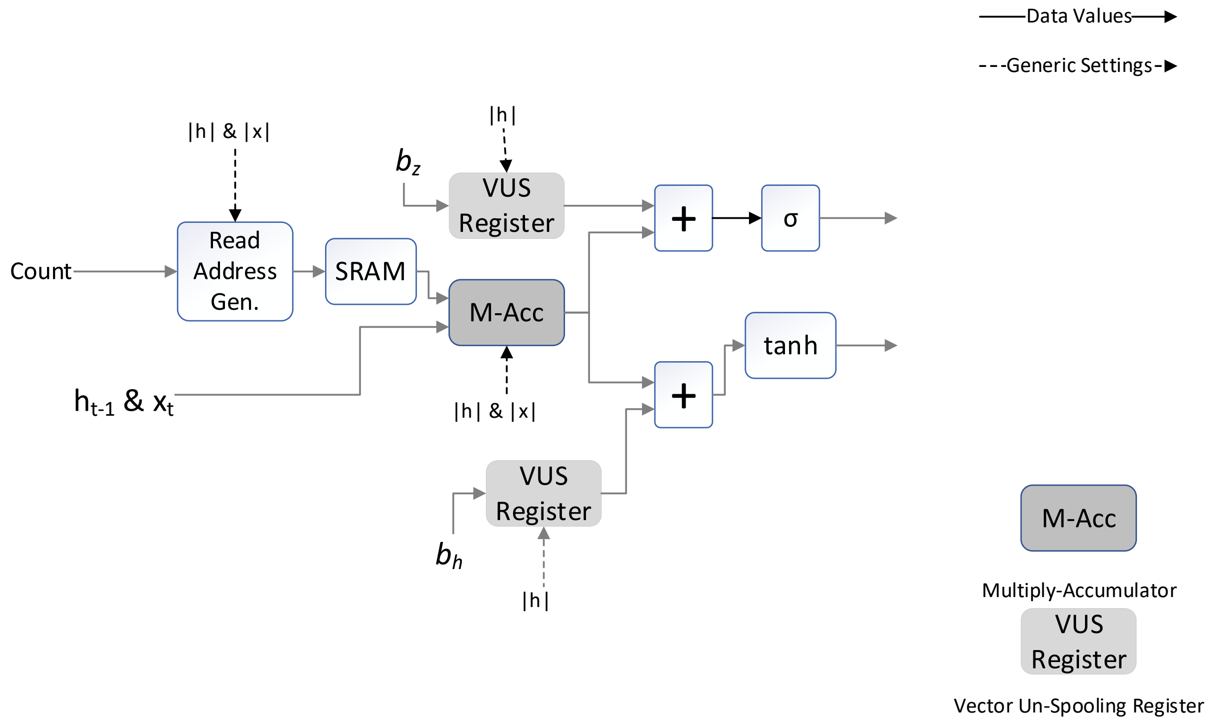

Figure 9: FastGRNN Gate Architecture

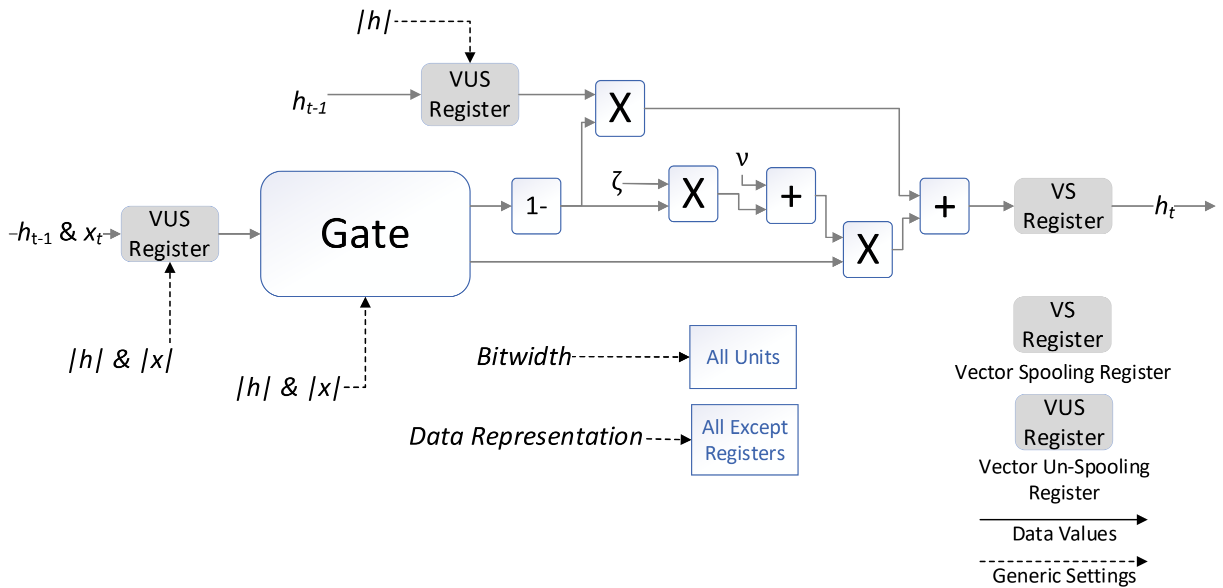

Figure 10: FastGRNN Layer Architecture

The main Vector Un-Spooling unit, shown at the far left of Figure 10 takes in 129 16-bit values in the initial layer, and 256 16-bit values in the other layers, corresponding to the sum of the sizes of the input vector (1 for the initial layer and 128 for the others) and hidden vectors (128 for all). The Multiply-Accumulate unit computes 128 16-bit results, performing 129 multiplications per result in the initial layer and 256 multiplications per result in the other layers. The following element-wise units operate on each of the 128 resulting values one at a time, with the final result being the new hidden state which is loaded one value at a time into the Vector Spooling register.

The hidden state of each layer is then used as the input of the next layer, shown in Figure 11. The output of the final layer is used as the input to a fully connected layer, which computes a vector-matrix multiplication against a matrix with two rows, producing two final results. This result is only calculated after a set number of input values have been operated on, determined as part of the size of the input vector. In this implementation, the input vector is specified as a set of 256 1-value vectors. This means that all four layers will compute a new hidden state 256 times before a final output is computed by the fully connected layer, at which time all the hidden states are reset to their initial values.

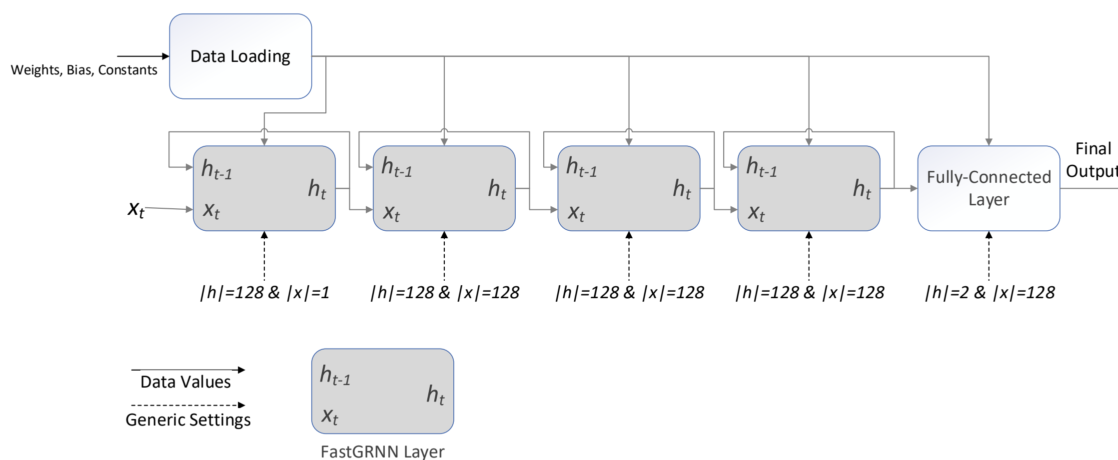

Figure 11: Case Study Top Level Architecture

4 Reconfigurable RNN Research Work

The second contribution of this dissertation is the development of a flow for the design and implementation of reconfigurable gated RNNs, which can take the place of a wide variety of gated RNN configurations in a single ASIC. This offers a middle ground solution between fully application-specific and fully reconfigurable solutions.

4.1 Reconfigurable RNN Design

The development of the reconfigurable RNN design makes use of the architectural analysis discussed in Section 3.2. The reconfigurable components significantly resemble the generic components discussed in Section 3, therefore many of the reconfigurable components will be discussed from the perspective of altering their related generic counterparts. The reconfigurable RNN is implemented as a single RNN layer that can be used sequentially to implement multi-layer networks. This results in the primary work for this design consisting of two parts: Reconfigurable Layer and Layer Wrapper. First, the development of the Reconfigurable Layer is completed by implementing a set of reconfigurable components. Second, the Reconfigurable Layer is supported by designing an external Layer Wrapper for the Reconfigurable Layer, adding components that support the Reconfigurable Layer by maintaining the hidden states of the RNN layers not currently being computed by the Reconfigurable Layer, as well as performing any conversions that may need to occur between RNN layers.

It is important to note the distinction between the Reconfigurable Layer and the RNN layers. The Reconfigurable Layer refers to the set of physical components designed to compute the result of an arbitrary RNN layer, while the RNN layers refer to the set of layers in the chosen RNN configuration. For example, an RNN configuration could have 4 RNN layers, which will all be computed individually by the single Reconfigurable Layer.

To design an ASIC that is reconfigurable, the specification of what variables need to be implemented in a reconfigurable manner is critical. Table 6 lists each parameter and the set of possible values for each parameter.

Table 6: Reconfigurable Variables

Parameter	Value Set
Layer Architecture	{GRU, FastGRNN}
Number of Layers	2-4
Hidden Vector Size	{64, 128}
Input Vector Width	{1,64,128}
Number of Input Vectors	2-256
Data Representation	{16-bit fixed-point, 8-bit floating-point}

4.2 Reconfigurable Unit Adaptations

The ten units listed for the Quick Put-Together methodology are still needed, but these units need to be designed to be reconfigurable to the same variables that needed to be generic in the QPT method. The four element-wise units: adder, subtractor, multiplier, and non-linear unit are only sensitive with respect to the data representation. In order to facilitate a variety of data representations, there are two options. The optimal solution would be to create fully reconfigurable units which can facilitate any data representation. This would come with greatly increased complexity for each of the units, a long design time, and significant on-chip overhead. For the purposes of this dissertation, there are only two possible data representations: 16-bit fixed-point and 8-bit floating-point. With only two representations, it was decided that the overhead of implementing these units by using separate computation units for each representation and multiplexing the output would be the optimal solution, while reconfigurable designs covering more representations may want to pursue the fully reconfigurable unit path for these units. Figure 12 shows the reconfigurable addition unit, as an example of how these four units were implemented.

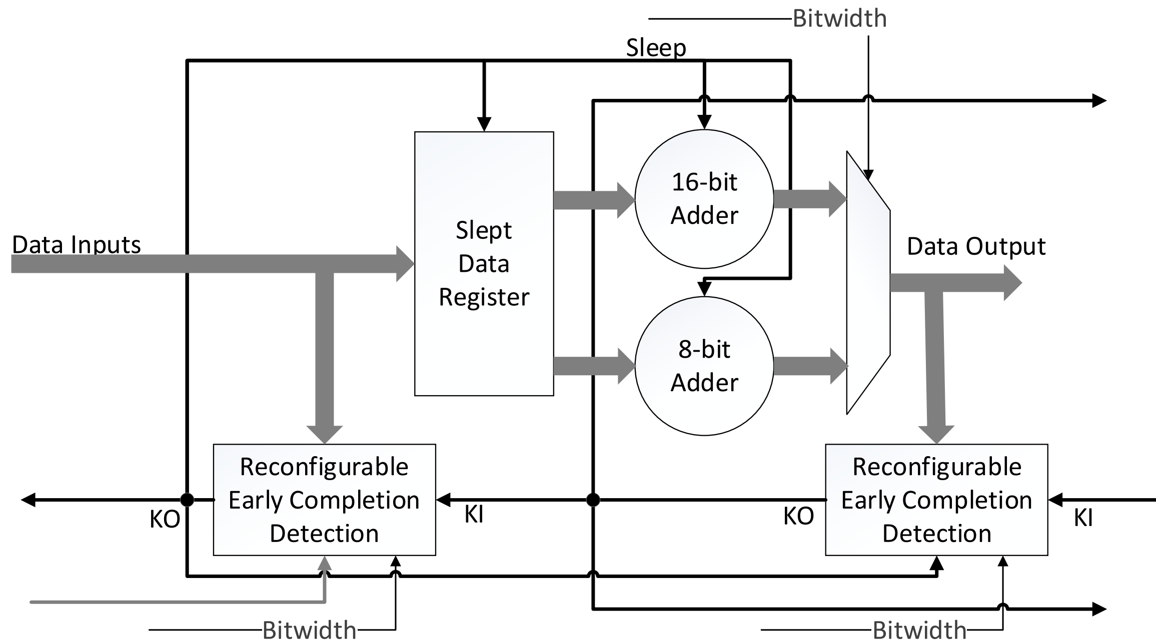

Figure 12: Reconfigurable Adder Architecture

The vector-matrix multiplication unit (Multiply-Accumulate unit) originally was designed to be generic with respect to the number of values it must operate on. For the reconfigurable version, the unit is implemented as the worst-case version (able to compute the maximum number of computations that could be required). The existing Self-Resetting Counter design from the QPT methodology facilitates this aspect already as it already possesses a control port dictating the maximum count. Previously, this port was tied to constants on chip, but for the reconfigurable design, these ports can be used directly to modify the maximum count during runtime. The Vector Spooling and Vector Un-Spooling units require similar changes, implementing these units at the maximum possible size to accommodate the worst case, with the Self-Resetting Counter acting to fulfill the reconfigurable aspect.

Once these units are able to be reconfigured, they are optimized for the set of allowed configurations. The Vector Spooling unit is used in the architecture exclusively to store the hidden state, or some vector of the same length as the hidden state, placing the total amount of

values as either 64 or 128. The unit must be able to hold 128 values for the worst case, so exposing all of the bits specifying how many values to accumulate would result in an additional 7 control pins on each Vector Spooling unit, where in reality this implementation only requires a single bit to specify between 64 and 128. Figure 13 shows the architecture for the reconfigurable Vector Spooling unit. The primary changes from the generic version are that the Maximum Count value for the Self-Resetting Counter has been changed to an input to the design, along with the new input for the Hidden State Size that controls how many values the unit will store before outputting the vector.

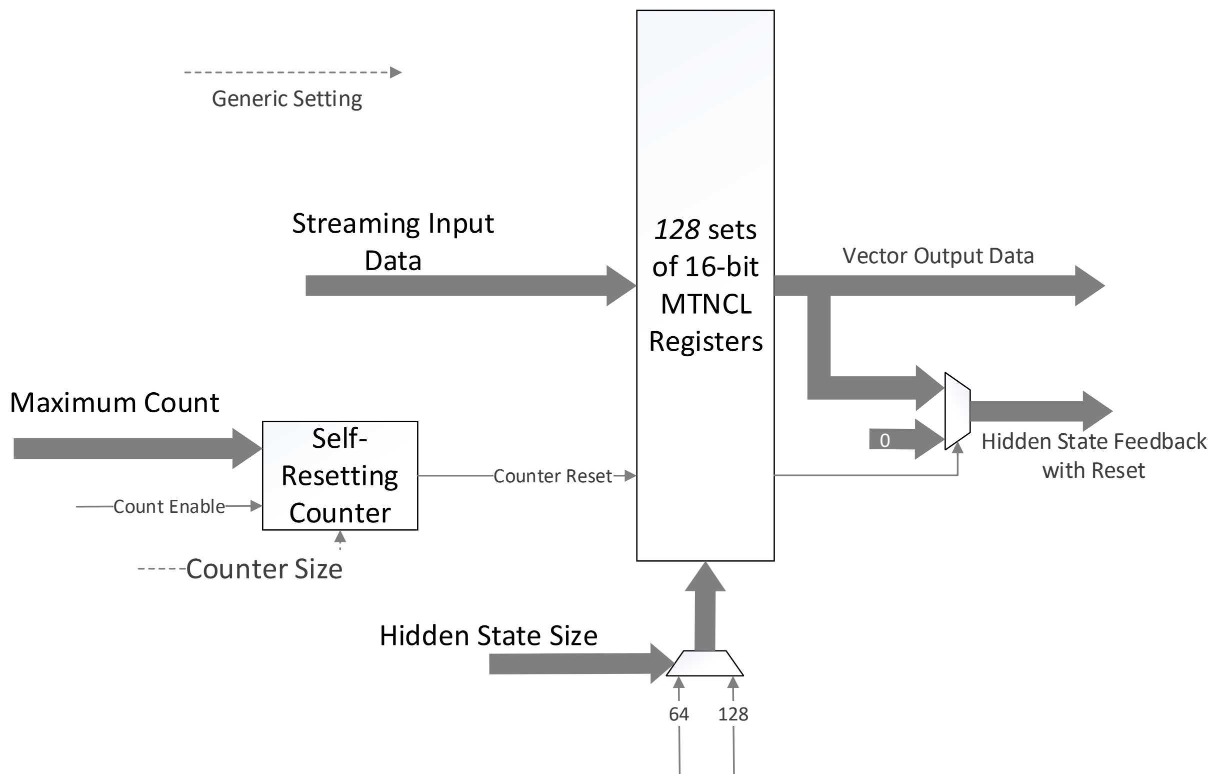

Figure 13: Reconfigurable Vector Spooling Architecture

The Vector Un-Spooling unit is used in two ways: to store and produce the hidden state, or some vector of the same size, or to store and produce the previous hidden state and the current input vector. This results in the Vector Un-Spooling unit being required to be capable of storing

and producing the amount of values in the set {64, 65, 128, 129, 192, 256}. The worst-case value, 256, would require an additional 8 control pins on each vector un-spooling unit, but since there are only six possible values, only three control pins are needed to cover the set of possible values. The unit must also have the ability to produce that number of values either 64 or 128 times depending on the size of the hidden state vector. As was the case with the Vector Spooling unit, this adds a single additional control pin.. The reconfigurable implementation of the Vector Un-Spooling unit is detailed in Figure 14.

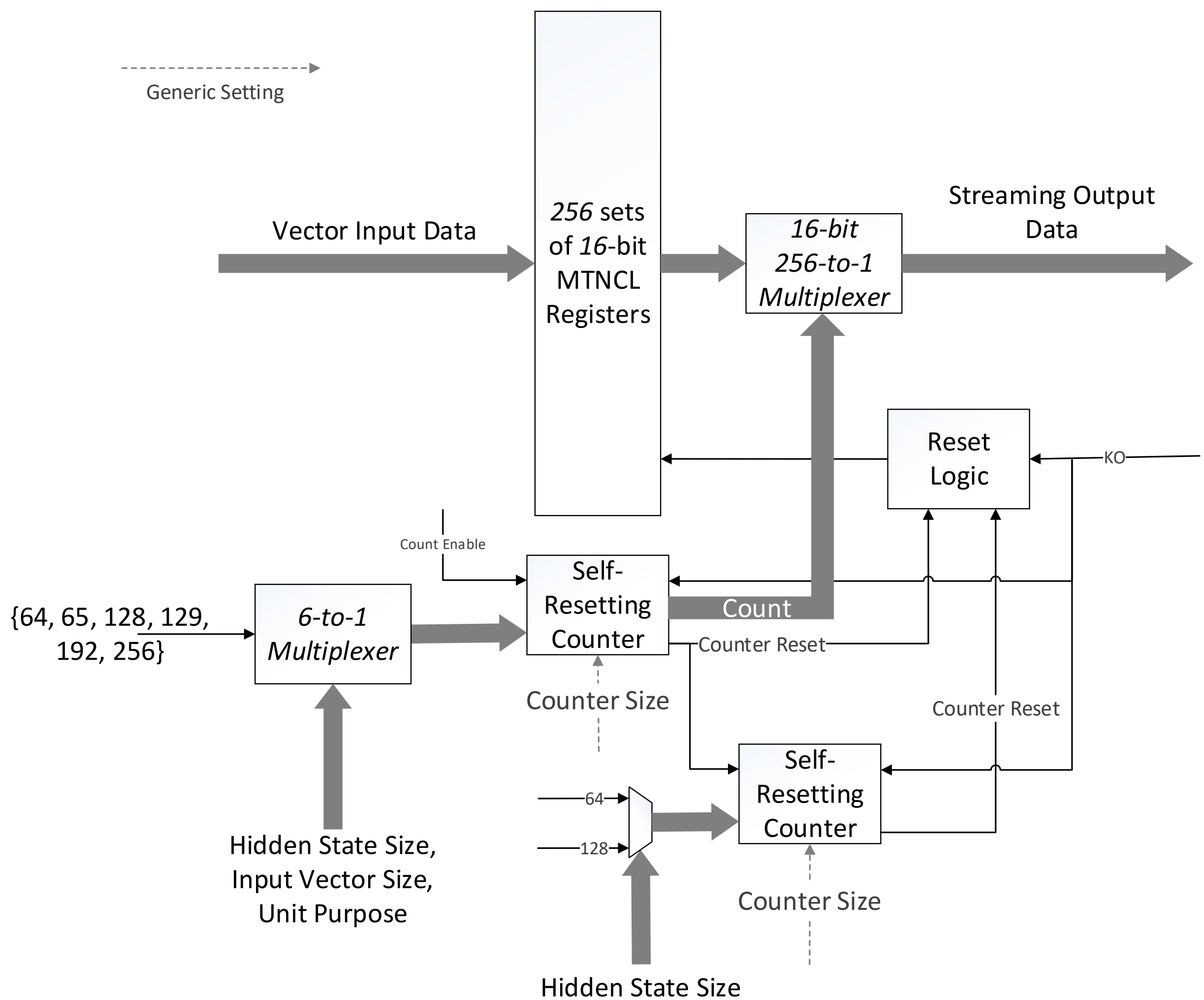

Figure 14: Reconfigurable Vector Un-Spooling Architecture

For the data storage aspects, there must be SRAMs able to hold all the weights for all the layers. To accomplish this, each SRAM in the layer must have a separated memory bank for each possible RNN layer, with additional address bits that select which bank is currently in use based on the current RNN layer being executed. This causes the SRAM Read unit to have the most complex change. The SRAM storage is treated as a concatenation of the two weight matrices into a single larger matrix, used to perform the two vector-matrix multiplications and the addition of the resulting vectors in a single Multiply-Accumulate operation. This concatenated matrix has a number of rows equal to the size of the hidden state (64 or 128 in this system), and a number of columns equal to the size of the hidden state plus the size of the input (65, 128, 129, 192, 256). This is a very similar selection set to that of the Vector Un-Spooling unit and requires the same number of resulting control pins (3). As in the Quick Put-Together design, the SRAM Read unit uses the Vector Un-Spooling's current count as the base address and performs a mapping of the count to the block circulant addresses of the SRAM. Figure 15 shows the SRAM storage.

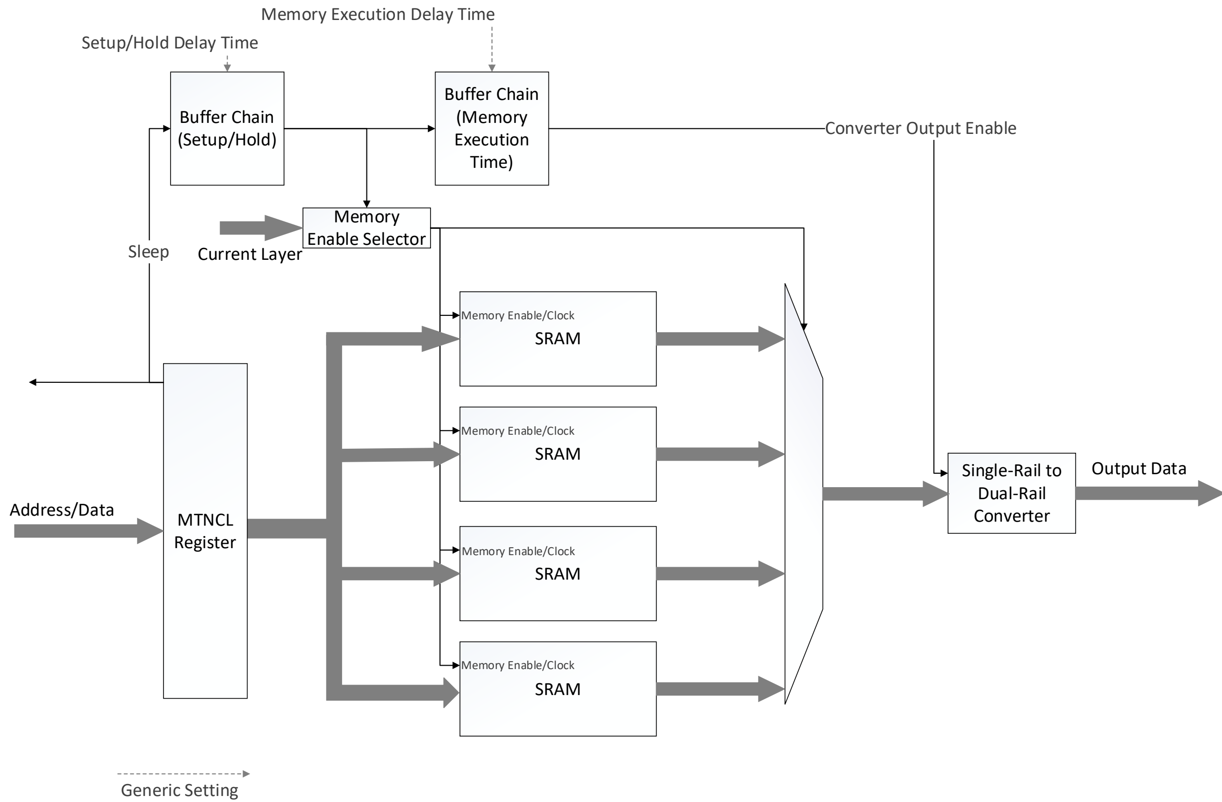

Figure 15: Reconfigurable SRAM

This does raise the question of how to handle the writing to the SRAM. For example, in the case when the layer has a hidden state size and an input vector size of 128, the weight matrix will have 128 rows of 256 values before block circulant reductions, with 16 rows of 256 values after, placed into addresses 0-4095 in that order (each value in the row in order before moving to the next row). In smaller cases (e.g., 128 rows of 128 values), the values are still kept in a contiguous set of memory addresses starting from 0 to ensure the read address generation remains the same, just using less bits from the base address to perform the address mapping.

While not actually part of the Reconfigurable Layer, the Data Loading unit is still a required part of the overall functionality of the Reconfigurable Layer. As before, it is a finite state machine with a state for each location that requires storing trained values, with a generic value determining how many values are stored for each state. The easiest option for

reconfigurability would be to expose the count for each state as a port, but this leads to a large surge in reconfiguration control pins (~6 pins per state on average). However, the Data Loading unit runs only at the very beginning of execution and writes to every element of each storage location exactly once, it can remain entirely unaffected by the configuration. This comes with a tradeoff of longer startup time for smaller configurations (as unneeded locations are written to) as well as some additional energy consumption but reduces the size and complexity of this unit by a significant margin.

The I/O for the configuration pins also influenced the decision between these two options. In the initial design phase, it was decided that the configuration pins would get dedicated I/O pads on chip, to avoid adding additional overhead in the form of another piece of I/O logic. With the number of pins that would be required to make the Data Loading unit reconfigurable, that decision would no longer have been feasible, and an additional component (with no reconfigurable aspects) would have been required to load in the chip's configuration at startup. Overall, it was decided that leaving the Data Loading write unit as non-reconfigurable was the suitable choice for this design, while designs with different I/O requirements or stricter startup timing and power requirements may wish to opt for the additional logic.

4.3 Reconfigurable MTNCL Completion Detection

In order to complete the design of the Reconfigurable Layer, one additional component is required; a reconfigurable MTNCL completion detection unit. As discussed in Section 2.1, the MTNCL architecture requires completion detection, where every bit of dual-rail output from a pipeline stage must reach a DATA value before the stage can return to the NULL state. The design is required to be able to perform with different bitwidths (16-bit and 8-bit in particular), which presents a problem for the standard MTNCL register/completion detection unit pair. If a

16-bit register receives only 8 bits of data, the completion detection unit will not detect that as a complete DATA wave. Initially, this was solved by having every data unit fill unused bits with zero values, but this added logic to every unit and consumed power transferring unused data. Instead, the design requires a Reconfigurable Completion Detection unit.

The MTNCL Completion Detection unit uses a tree structure using a set of THxor0m gates to check each bit in sets of two (with an additional TH12 gate if the number of bits is odd), and then using an MTNCL AND tree to check that every bit has reached DATA. For the reconfigurable version, the design needs to be able to effectively disable bits from contributing to the overall completion detection. This is a case where MTNCL is particularly more useful for this case than NCL. In NCL, all inputs to an NCL gate must return to logic 0 before the NCL gate can return to logic 0, but that is not true of MTNCL, and due to this, each bit that can be disabled of the MTNCL completion detection unit can contribute the result of the Boolean equation *data1+data0+disabled* to the MTNCL AND tree and the overall result will still be correct for the bits that are enabled. Additionally, this can be adjusted based on how fine-grained the reconfigurable aspect needs to be by performing the OR function with the disable signal at higher levels in the and tree. In the case of this design, only two options are needed, 8-bit and 16-bit. By selecting that the 8 most significant bits of each register is able to be disabled, a single additional gate is added to the standard MTNCL completion detection unit, a TH12 gate, which takes the result of the most significant 8 bit’s AND tree and the disable signal, and contributes that result to the AND tree for the 8 least significant bits. With that, if the register is in 8-bit mode, the top 8 bits will always register as complete and only the non-disabled bits will measure current completion status, while in 16-bit mode all 16 bits will still contribute to the complete

status. This unit replaces the completion detection unit of every 16-bit register in any component in the design. Figure 16 demonstrates the design of this unit.

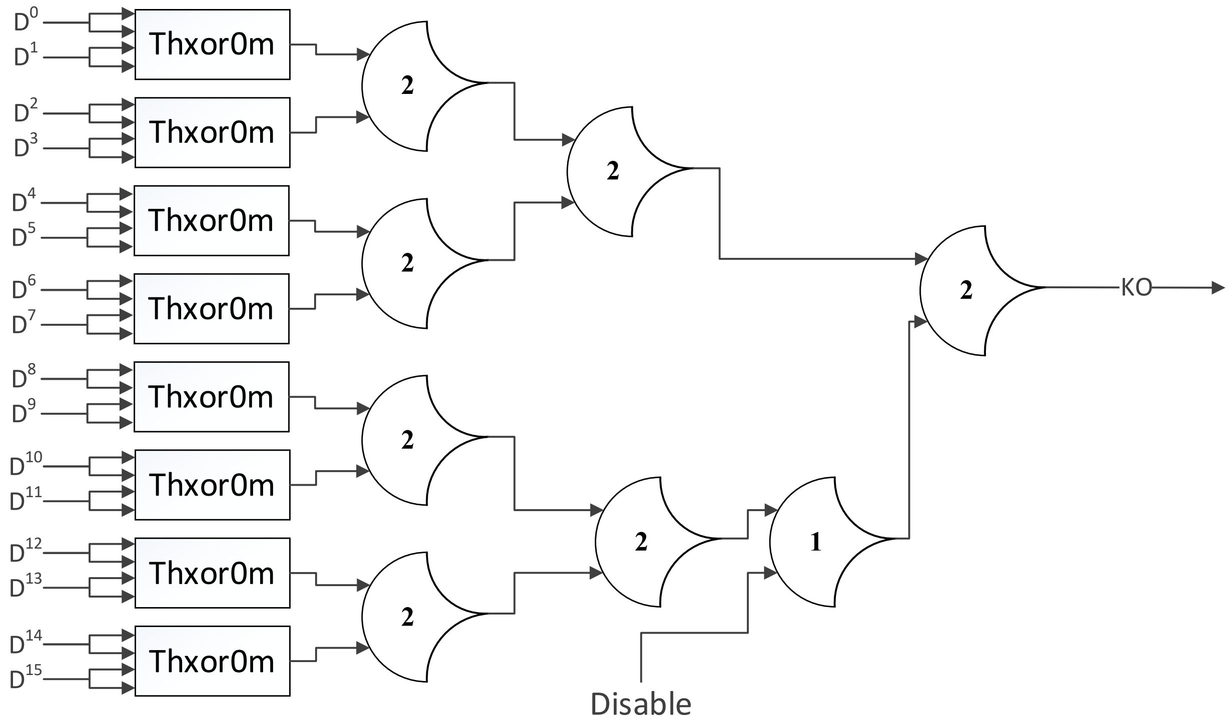

Figure 16: Reconfigurable MTNCL Completion Detection

4.4 Reconfigurable RNN Top Level Design

Designing the architecture for the complete implementation of the reconfigurable RNN begins with how to handle the layers. Since the targeted design space includes between two and four layers in a given configuration, the RNN must be designed with the worst case (4 layers) in mind. The simplest solution would be to have four copies of the reconfigurable layer design, using only the number of layers needed for the current configuration. While this solution is the simplest to implement, the result is very large and extremely inefficient, as half the available circuitry is completely unused in the case of a two-layer network. Instead, the design was architected with the goal of reducing the amount of unused logic in any configuration. To accomplish this, a single reconfigurable RNN layer was implemented and used repeatedly in

place of each layer in the configuration. This Reconfigurable Layer is the primary computation unit of the design.

The reconfigurable components discussed in Section 4.2 allow the layer to handle all of the reconfigurable variables except for two: the number of layers and the architecture of each layer. The primary focus of the Reconfigurable Layer's design must be able to handle both RNN layer architectures, GRU and FastGRNN. Tables 3 and 4 show the equations for each of the architectures. The initial implementation, shown in Figure 17, contains all the logic necessary to compute either layer, selecting which layer architecture is in use with a multiplexor while the other is placed in the sleep state to conserve power.

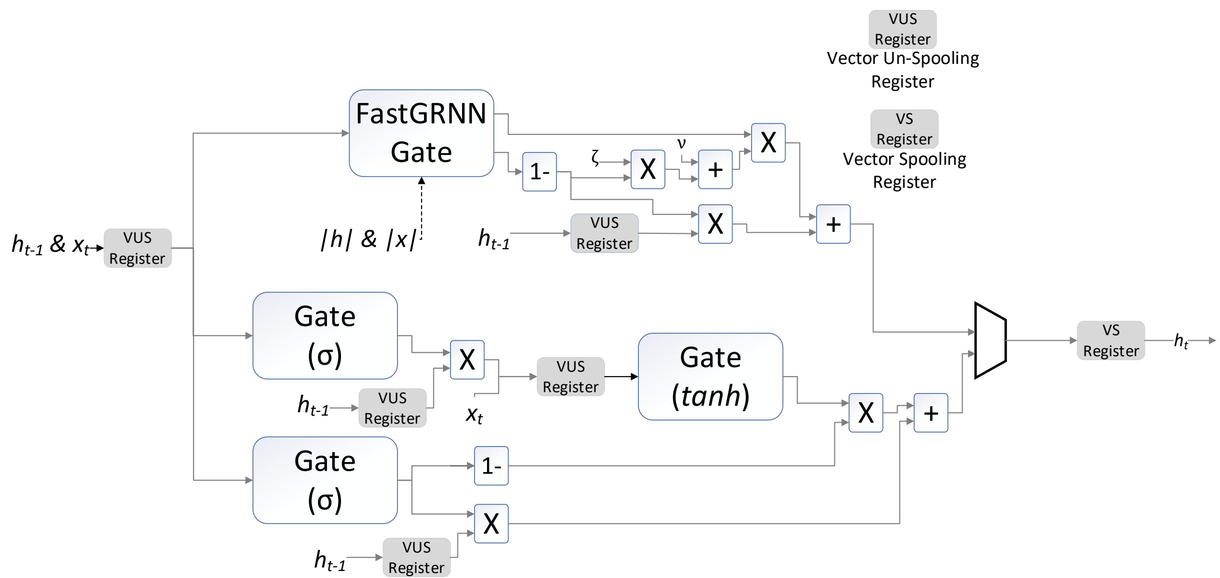

Figure 17: Reconfigurable Layer – Worst Case

As was the case with the decision between using a single layer repeatedly over multiple separate layers, this solution is the easiest, but leaves significant room for optimization. Instead, the Reconfigurable Layer implements the worst-case architecture in full, which is the GRU architecture in this case, and then the FastGRNN architecture is analyzed to find how it can be

included in the design using already existing logic as effectively as possible. Figure 18 shows the implementation of the layer with only the GRU architecture.

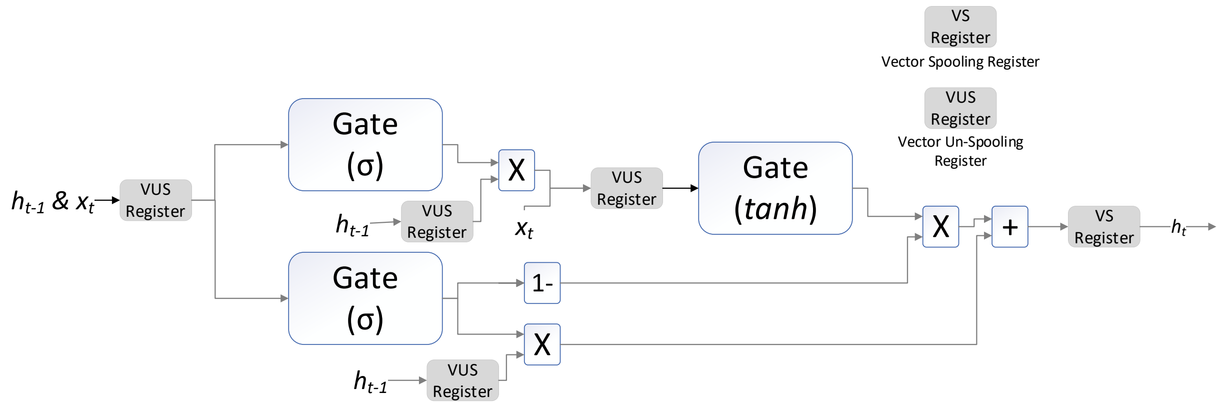

Figure 18: Reconfigurable Layer – GRU Only

The key aspect of the FastGRNN algorithm is that although it contains two gates, these gates share weights. This allows the FastGRNN's vector-matrix multiplication to be performed just once, with the result going to separate adders to add the bias values, followed by separate non-linear units. Thus, FastGRNN requires a Multiply-Accumulator for the vector-matrix multiplication with the output branching into two adders for the addition of the bias vectors with one adder followed by a *sigmoid* non-linear unit and the other followed by a *tanh* non-linear unit. These components are enough to complete the FastGRNN layer's gate computations. All of these computations can overlap those in GRU. In Figure 18, the Multiply-Accumulator for the z_t gate is used to compute the vector-matrix multiplication. The output can then branch, with one branch using the bias adder and sigmoid component of the same gate, and the other using the bias adder and tanh component of the $\tilde{h}_t$ gate, completing the FastGRNN gate computations with only an additional multiplexor to select the input to the bias adder of the $\tilde{h}_t$ gate. The remaining computations for FastGRNN are very similar to GRU. The only difference in the two equations after the gate computations are the two constant operators in FastGRNN: ζ and ν. ζ is multiplied

by the result of the already present $1 - z_t$ calculation, and ν is added to the result of that multiplication. Overall, FastGRNN can be included into the GRU layer with only an additional multiplier, adder, and two multiplexors. The final layer design is shown in Figure 19.

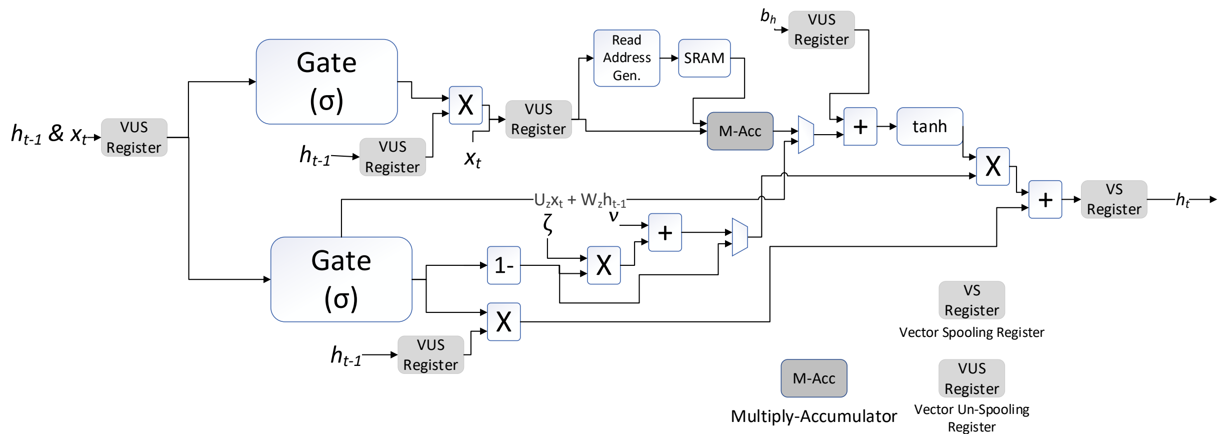

Figure 19: Reconfigurable Layer – Final

The Reconfigurable Layer is the primary computation unit, but it requires a supporting structure which handles both the inputs to the layer and the outputs from the layer, operating as a wrapper for the layer unit. This wrapper has four key functions:

1. Maintain the current state of execution
2. Store the previous hidden state of each layer
3. Input the correct hidden state and input vector
4. Perform conversions between data representations

Maintaining the current state of execution refers to keeping track of which layer is currently being calculated and controlling the configuration pins of the layer to match the current layer's configuration. Hidden state storage must also occur outside of the layer, and with that comes the need to control when the hidden state is used again as input. Finally, as the input vector of a layer is the hidden state of the previous layer (or the chip input in the case of the first layer), the wrapper must control the dataflow to take the output from the layer as the input, and

convert to the current layer's data representation if necessary. From these four functions, the wrapper's architecture can be broken down into three components: the Control Register, Hidden State Storage, and Input Vector Management.

First, the Control Register is created from the Vector Un-Spooling Register. This component contains a set of layer configuration values for each layer in the design (up to four in the case of this design), and outputs each set one at a time in order to set the layer into the current state. It also outputs the current count the same way the generic Vector Un-Spooling unit does, acting as the address for the currently needed hidden state from the hidden state storage. This function is repeated until reset, so the functionality that resets the Vector Un-Spooling Register after a set number of complete outputs is disabled in this case.

The Hidden State Storage is implemented as a set of large registers, one for each possible RNN layer. The output of these registers is addressed by the layer counter from the Control Register, and whichever register's output is used also receives the next output of the layer. Finally, the input vector origin needs to be selected and in some cases converted between possible numeric representations.

The Input Vector Management unit receives the output of the layer and that layer's data representation information. When the next layer's control bits are output from the control register, the Input Vector Manager checks if the next layer to be computed is the initial layer. If so, the Input Vector Manager takes the input from the input vector ports on the chip. Otherwise, the Input Vector Manager compares the data representation from the previous layer and the new layer and determines if conversion is necessary, performing the conversion computation if needed. This conversion unit is only necessary if it is allowed for a configuration to differ in the data representation aspect, such that every layer of the RNN network is not required to have the

same data representation. If so, this computation is set at an architectural level, and must be created specifically for the data representation conversions specified.

As the name implies, the Reconfigurable Layer is completely encased within the Layer Wrapper, with all of the Reconfigurable Layer's inputs and outputs controlled by the Layer Wrapper. With both of these units designed, the final architecture of the Reconfigurable RNN is shown in Figure 20.

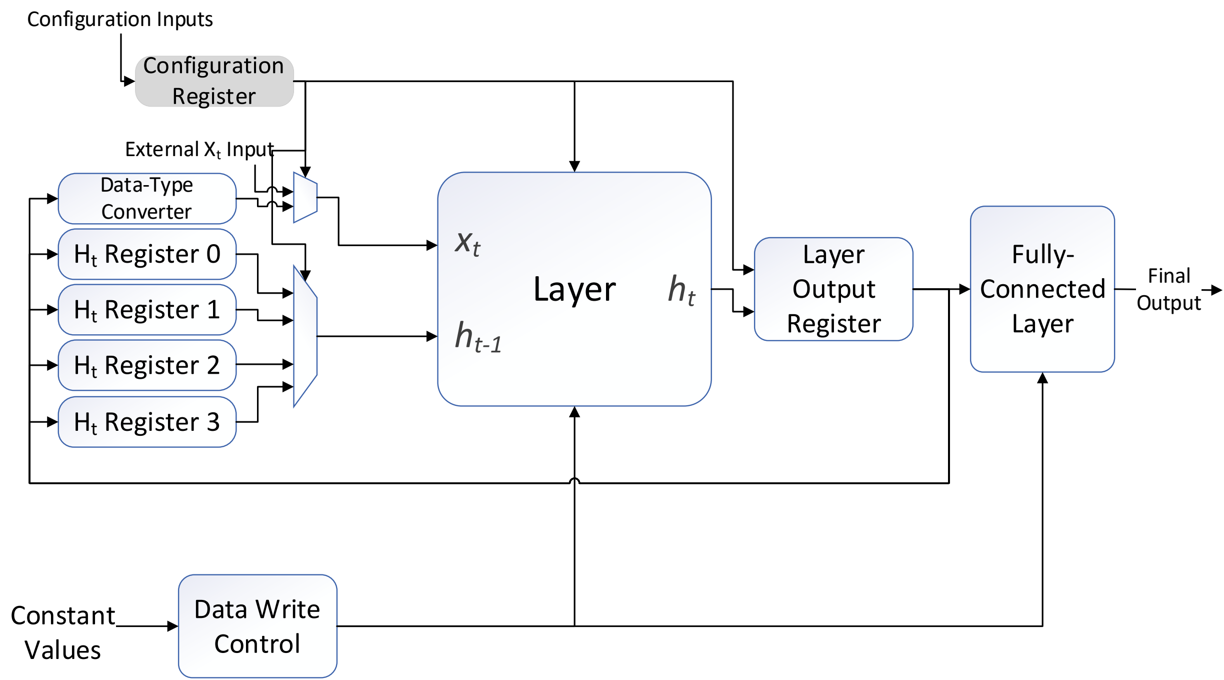

Figure 20: Complete Reconfigurable Design

5 Results and Analysis

The designs discussed in Sections 3.4 and 4.4 were both implemented into the TSMC 65nm bulk CMOS process.

5.1 Quick Put-Together RNN Results

The Quick Put-Together case study was implemented onto a 3mm × 2.9mm integrated circuit. Figure 21 shows the full chip layout. The design is currently under fabrication at TSMC.

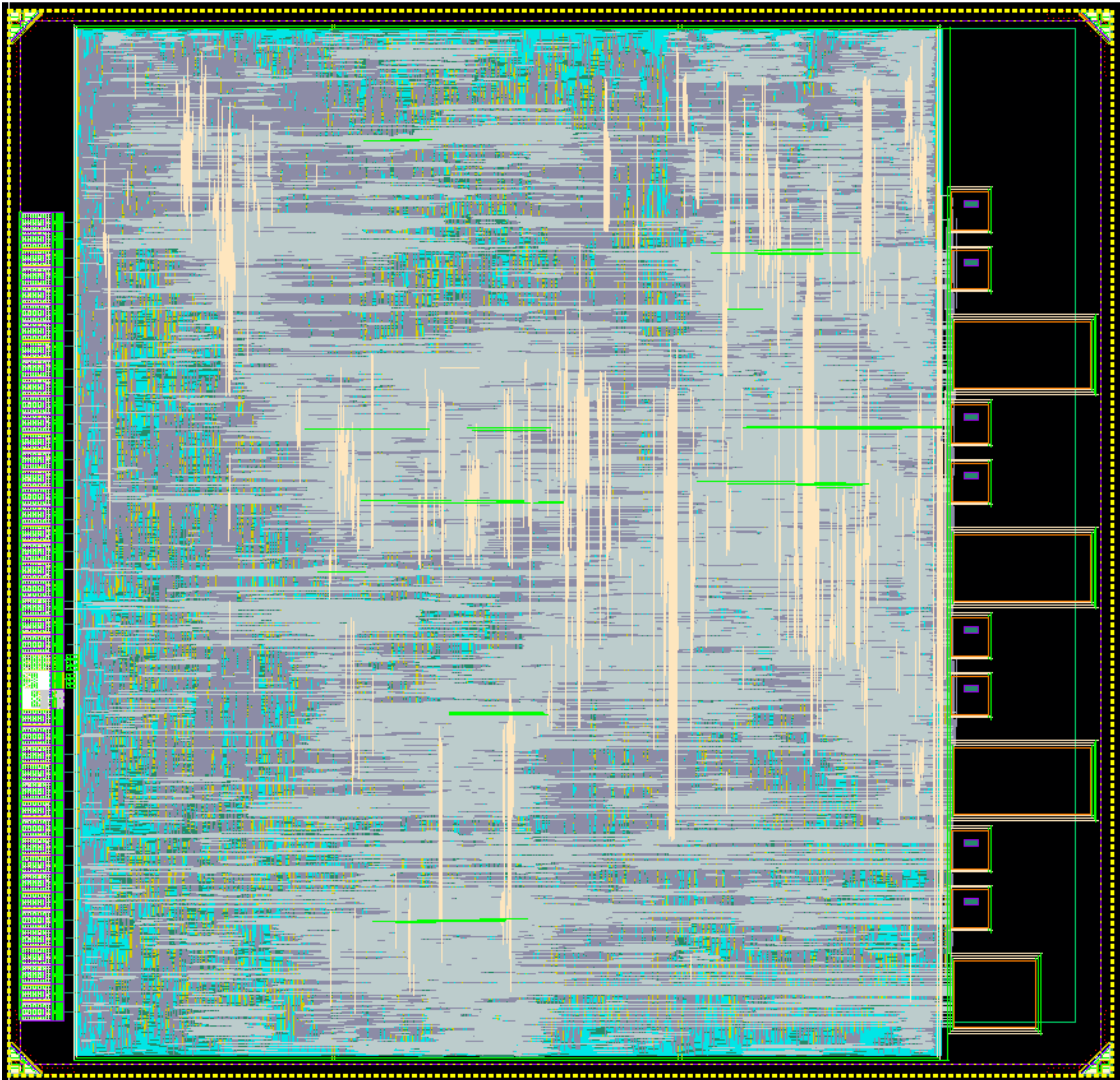

Figure 21: Quick Put-Together Physical Layout

The primary objective of the Quick Put-Together methodology was to show that with an

in-depth analysis of RNN architectures, a set of generic and extensible designs would allow for an asynchronous low power ASIC implementation to be completed in a short period of time. The logic design for this chip was performed twice, separately by two individuals whose experience with the methodology differed greatly. Individual 1 was the same individual that performed the RNN architecture analysis and implemented almost all of the generic designs, as well as having a previous experience implementing an LSTM ASIC. Individual 2 had no prior experience implementing RNNs and helped implement only a few of the generic designs. This disparity allows the results of the implementations to show an approximate range of the amount of labor that can be expected from future implementations following this methodology. Individual 1 completed the logic design and verification in 30 hours of labor, while individual 2 required 123 hours of labor. While this is a fairly wide range, the maximum value of 123 hours to complete a logical implementation of this size and complexity is quite fast, and with additional experience the number of hours for implementation drops up to 75.6%. The physical design was performed only once, as the time improvement for this methodology comes primarily from the reduction in the verification time that would be required for synchronous times to verify timing requirements. The physical design took 82 hours of labor, putting the total labor to complete an entire asynchronous ASIC implementation to between 112 and 155 hours of labor in total. Preliminary power and speed data were gathered by performing transistor level simulations which showed the following results. The design required 97.7 ms to complete a full characterization, utilizing 10.1 mJ of active energy on average. When idle, the design utilizes 1.30 mW of standby power.

The speed of the design is primarily limited by the vector-matrix multiplication. The non-input layers each compute 32,768 Multiply-Accumulate functions to complete a single hidden state vector (input layer performs 16,512), and this must be performed 256 times to get a single

output. Overall, the speed is not extremely slow compared to the size of the calculation to be performed and can easily be improved significantly by adding parallelism to the Multiply-Accumulation at the cost of increased power utilization.

5.2 Reconfigurable RNN Results

The Reconfigurable RNN case study was implemented onto a 3mm × 3.9mm integrated circuit. Figure 22 shows the full chip layout. The design is currently under fabrication at TSMC.

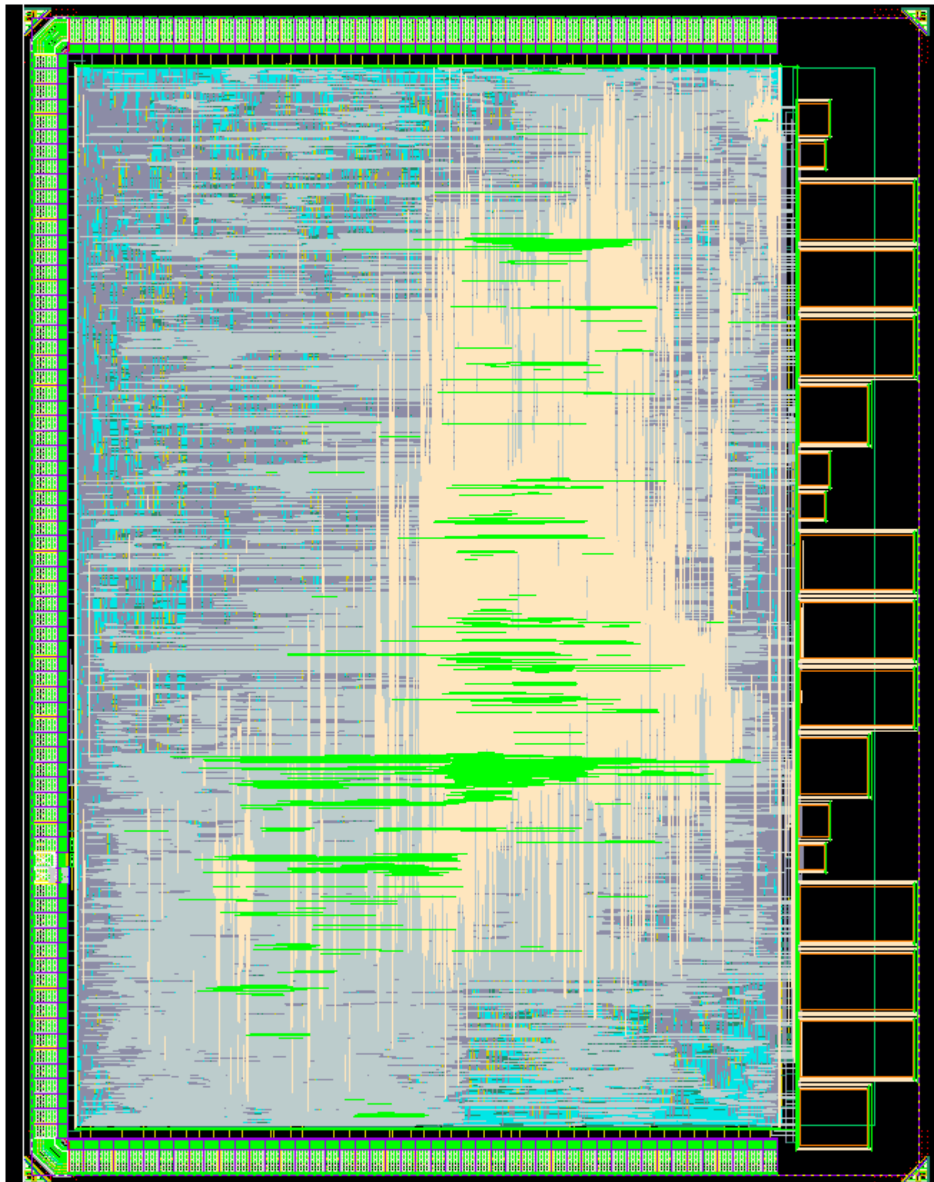

Figure 22: Reconfigurable Physical Layout

The primary objective of the Reconfigurable RNN methodology was to utilize asynchronous MTNCL design methods to implement a wide array of gated RNN configurations in a single ASIC, to demonstrate the feasibility of incorporating reconfigurable ASIC implementations of machine learning algorithms into designs with power, cost, and security requirements that would not allow for fully reconfigurable solutions such as FPGAs. The chosen design covers thousands of possible gated RNN configurations in a single ASIC. Preliminary power and speed data were gathered by performing transistor level simulations which showed the following results. The design consumes 1.89 mW of idle power, but the time and energy for a complete characterization vary widely by configuration. The worst case, a 4-layer network with GRU layers and 16-bit data and 256 input vectors consumed 30.3 mJ of active energy on average while requiring 262.0 ms to complete a full characterization. The best-case results would be for a 2-layer network with both layers using the FastGRNN architecture, 8-bit data, and only 2 input vectors. This configuration consumed 10.5 μJ of active energy on average and required 15.5 ms to complete a full characterization.

A large portion of the time required, particularly for the best case, is consumed by the weight data loading process. This is primarily due to the external flash memory chosen to be used in conjunction with this design runs at 50 MHz and loads one bit of data per cycle, causing the data loading interface to be the slowest aspect of the design, consuming 14.5 ms and 9.75 μJ on average at the startup of any configuration. Additionally, since the design uses the single Reconfigurable Layer to compute each layer, the worst-case version has the Reconfigurable Layer perform a full execution 1024 times. Having multiple layers or increasing in-layer parallelism would significantly reduce the execution time, especially for larger networks. A large portion of the energy consumption, as well as the overall area, comes from the data storage. The

hidden state storages for all the layers requires 1 KB of storage for the worst case, and other large registers, such as the vector spooling and un-spooling registers continue to compound the large number of registers needed. Even after the block circulant method for reducing the weight matrix size, the input layer requires three sets of 4.25 KBs SRAM, and the three possible non-input layers each require three sets of 8 KBs SRAM, for a total SRAM storage of 84.75 KBs of SRAM. Moving the weight storage into some off-chip interface that is faster than the flash chosen for this application would significantly reduce the power and area of the chip itself, as well as almost completely eliminating the setup time required before the network can begin computation. While this solution simply moves the burden of the area and power to some arbitrary off chip component, this would also eliminate a significant amount of complexity from the overall application, as the chip would no longer require the Data Loading component or the SRAM Reading component, instead just requiring some off-chip unit send data to the chip in a predefined order.

6 Conclusion and Future Work

In conclusion, this dissertation work focuses on developing two methodologies for improving the feasibility of the design and implementation of neural network applications as ASICs, utilizing RNNs to demonstrate the methodologies. The first methodology demonstrates a method for analyzing and implementing RNNs with asynchronous generic components to reduce non-recurring engineering cost of ASIC development and the time-to-market for similar designs. The second methodology shows the development for a reconfigurable RNN architecture, demonstrating the feasibility of reconfigurable ASICs as a middle ground between fully application specific ICs and fully reconfigurable solutions for neural network applications.

Analysis of several architectures of RNN was performed, and the result of the analysis is used to develop a set of generic components to cover a wide array of possible RNN architectures and configurations. The analysis of these architectures was also used to develop an architecture of a single RNN layer and supporting logic to implement a single asynchronous ASIC that can be reconfigured to a wide variety of RNN configurations. The Quick Put-Together methodology is used to complete a tape-out in the TSMC 65nm bulk CMOS process to show the effectiveness of the methodology in reducing time spent on design and implementation after completing the architecture analysis and generic component implementation. A full tape-out was also completed for the Reconfigurable architecture, also in the TSMC 65nm bulk CMOS process.

Future work in the Quick Put-Together methodology should include adding additional architectures to the set of generic components, as well as an analysis of the costs and benefits of making the generic components less fine-grained, such as a complete layer that is generic to all the aspects relevant to the layer. Future work for the Reconfigurable methodology should focus on the costs and benefits to making the non-reconfigurable units in this design fully

reconfigurable, such as arithmetic units that are reconfigurable to a wide array of data representations. Due to the high computation costs and large data sets of RNNs, there is a very large number of ways to tune the power, area, and speed to the application in question, opening a significant amount of work to be done to understand the best way to handle these architectures in ASIC implementations, but this dissertation has contributed significant work to the wider usage of ASIC neural networks more easily and in a wider array of applications.

7 References

[1] L. Zhou, R. Parameswaran, F. Parsan, S. Smith, J. Di, “Multi-Threshold NULL Convention Logic (MTNCL): An Ultra-Low Power Asynchronous Circuit Design Methodology,” Journal of Low Power Electronics and Applications, pp. 81-100, 2015.

[2] S. C. Smith and J. Di, “Designing Asynchronous Circuits using NULL Convention Logic (NCL)”, Morgan Claypool Publishers, 2009.

[3] J. Schmidhuber, “Deep Learning in Neural Networks: An Overview,” Technical Report IDSIA-03-14, October 2014.

[4] S. Hochreiter, and J. Schmidhuber. "Long short-term memory." Neural computation 9.8, pp. 1735-1780, 1997

[5] K. Cho, B. Van Merri.nboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." In arXiv preprint arXiv:1406.1078, 2014

[6] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma. “FastGRNN: a fast, accurate, stable and tiny kilobyte sized gated recurrent neural network.” In Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18), pp. 9031 9042, 2018

[7] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” in Proceedings of the 32nd International Conference on Machine Learning, pp. 2342–2350, 2015

[8] C. Ding, S. Liao, Y. Wang, et al., “CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices,” Proceedings of MICRO-50, 2017