The Bioscan AN INTERNATIONAL QUARTERLY JOURNAL OF LIFE SCIENCES 20(2): S2: 4-10, 2025 www.thebioscan.com

A PROOF-OF-CONCEPT DIGITAL LEAKY INTEGRATE-AND-FIRE NEURAL NETWORK - THE CAT DESIGN

Chilukuri Jagadeesh Reddy[1],Battula Kesava Srinivas[2], P. Saleem Akram[3]

Koneru Lakshmaiah Education Foundation, (Deemed to be University), Guntur, A.P, India

*saleemakramp@gmail.com

DOI: https://doi.org/10.63001/tbs.2025.v20.i02.S2.pp04-10

KEYWORDS
Neural Network,
Digital Neural Nets,
Leaky Integrate-and-Fire,
VLSI,
Address- Event Representation (AER),
Neuromorphic.

Received on:

12-02-2025

Accepted on:

10-03-2025

Published on:

07-04-2025

ABSTRACT

Digital neural networks are an alternative computing structure to the traditional von Neumann architecture and offer enhanced performance when many inputs need to be processed in parallel. This occurs in complex data sets or data sets composed of sensory (e.g. visual or auditory) information. Neural networks inspired by biological systems offer efficient solutions for real-time signal processing, pattern recognition, and neuromorphic computing. This paper presents a proof-of-concept digital implementation of a Leaky Integrate-and-Fire (LIF) neural network, designed and tested in hardware using FPGA-based digital logic. The proposed architecture, termed CAT Design, leverages a scalable and modular approach to implement LIF neurons with configurable parameters such as membrane potential decay, threshold-based firing, and synaptic weight adjustments. By employing efficient digital arithmetic and parallel processing techniques, the design achieves low-latency spike-based computation, making it suitable for energy-efficient neuromorphic applications. We evaluate the performance of the system in terms of computational efficiency, hardware resource utilization, and real-time processing capabilities. The results demonstrate the feasibility of implementing spiking neural networks in digital hardware, paving the way for future developments in brain-inspired computing systems.

This paper presents the results from testing a small proof-of-concept digital neural network designed in VLSI hardware.

INTRODUCTION

The field of neuromorphic computing has grown out of prolific research done on neurotransmission and the recognition that, while the clock speed of modern computers may be orders of magnitude faster than the rate at which neurons are capable of firing, the brain is still far ahead of computers in terms of its ability to process complex information. Recognizing this, scientists developed neural networks: models of the brain's processing scheme. Neural networks were developed by studying the organic brain's ability to parse and process a myriad of inputs in parallel and produce meaningful cognition from the data. Contrary to a traditional von Neumann model, which uses a series of sequential instructions, neural networks rely on thousands to millions or billions of neurons - the basic processing unit of the brain - connected through synapses to form a series of layers and loops. A change in potential across a neuron membrane causes a spike to occur, propagating out to each connected neuron. The spike is the currency of neural computation, while the strength of the connection between neurons is the programmable framework.

Much like a biological brain can form new or stronger memories as it learns and adapts, digital neural networks can be 'trained' to accomplish some tasks, by strengthening or weakening these connections. These tasks can vary from deciphering hand-written text, or predicting the outcome of a sports match given any number of parameters, such as time of day, month, weather conditions, etc. One advantage of neural networks is they can process information in parallel as a system rather than the sequential method of traditional processors

This advantage can be fully realized by constructing a neural network model in software. However, while simpler to build, neural networks become extremely inefficient when built-in software and run on traditional processors because the traditional von Neumann instructions must be used to simulate every potential change sequentially. The required time can be increased by using multi-core processors or supercomputers, but it still grows out of hand as the number of neurons reaches the millions or billions required for significant neural computation. Indeed, it recently took the K supercomputer in Japan, which contains 82,944 processors, 40 minutes to simulate 1 second of real-time computation of a network containing only around 1% of the neurons in the brain. However, this is a popular approach because of the ease of implementation and works well for simpler applications. For example, the Qualcomm Zeroth is another example of a deep-learning device capable of learning

and adapting to its surroundings in real time. The Zeroth project achieves this solely in software, however, meaning the processing all takes place on Qualcomm's multicore Snapdragon processor, and the 'spikes' produced in a biological system are simulated in software. The Zeroth project focuses its research on the fluid incorporation of the software into existing systems, giving them the ability to learn things like recognizing and translating handwriting into text and analyzing the contents of images.

Instead of neural networks in software, neuromorphic engineers are striving to build native hardware for neural networks which will allow the efficient implementation of these systems An additional advantage of implementing these systems in hardware is that memory becomes dynamic - stored in the synapses - so there is no need to institute an expensive (in time, power, and space) memory system. Additionally, because neural networks use spikes rather than constant high or low lines, they require much less power. The ability of neural networks to a) efficiently process so many inputs in parallel, b) effectively eliminate the processor-memory bottleneck, and c) perform these operations in a low-power framework will lead to an entirely new field of computing, one where systems will efficiently harness neural nets to vastly increase computing productivity for certain tasks.

Neuromorphic computing was first developed by Carver Mead in the 1980s at Caltech. Mead recognized that the brain's entire communication system was based significantly on the propagation of electric signals and set out to recreate what he observed in hardware, specifically focusing on the visual system . Since then, several other companies and re- 2 search labs have taken on the endeavor of designing cognitive computing platforms inspired by biological neural networks. In neuromorphic computing, several different parameters can be used in system design. The first is the selection of which neuron model to use. Neuron models range from the complex Hodgkin-Huxley model which encompasses every aspect of neuron behavior to the perceptron which models simple binary behavior. Engineers must choose between a model which allows simple implementation, while also yielding enough complexity to form a variety of systems.

One of the most popular models for neural computation is the Leaky Integrate and Fire (LIF) model, which mimics the capacitive-like buildup and discharge between biological synapses in an organic brain but can still have a relative implementation. For example, IBM just

created the True North chip, a clockless, 4,096 core processor, with each core handling 256 neurons, and in turn, each neuron creating 256 synapses. This works out to be just over four billion artificial LIF neurons and 1 trillion synapses. Several other researchers, including Gert Cauwenberghs at UCSD, and Kawabena Boahen at Stanford have chosen the LIF neuron model as well although their implementations are slightly different. Meanwhile, other researchers, such as Alice Parker at USC are using carbon nanotubes to mimic neuron models that replicate many more of the complex behavioral patterns. Fig. 1 shows an example of a digital implementation of a LIF model. Fig. 2 shows an equivalent representation in analog.

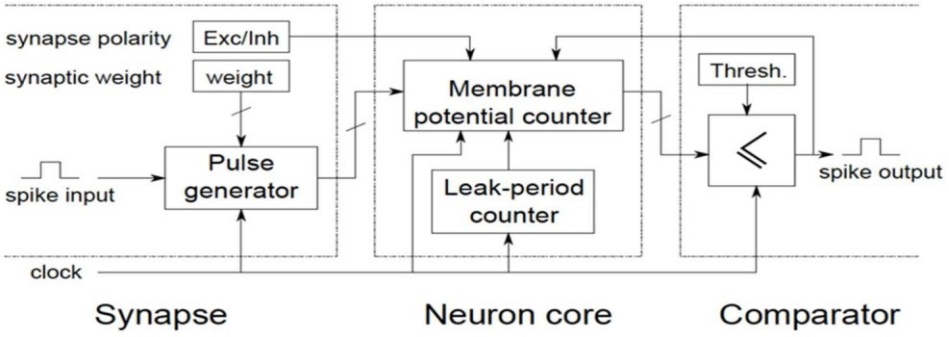

Fig. 1. Show Example of digital implementation of an LIF neuron.

The above Figrue 1, represent The number of neurons implemented, True North and other networks require an extremely complex learning algorithm, because there are no traditional programming languages that can be applied to these chips . As one IBM fellow said, "Given this unconventional computing paradigm, compiling C++ to True North is like using a hammer for a screw" . Rather neural networks must "learn" how to respond. They are given sets of sample data and then they learn how to respond to this data with the hope that this training algorithm will apply to future data sets as well. Each neuron is connected to another neuron with a specific strength or weight and this weight determines the magnitude of the jump in membrane potential. The training algorithm adjusts these weights to adjust the behavior of the network. One popular training algorithm for neural networks, backpropagation, involves the computation of the error between the desired and actual output. This adjustment is performed on the weights of the last layer and then propagated backward to adjust the next layer.

- The design contains 16 separate neural processing units that replicate a LIF neuron.
- Custom interconnection system based on published AER systems for transferring spikes.
- Programmable leak and threshold values.

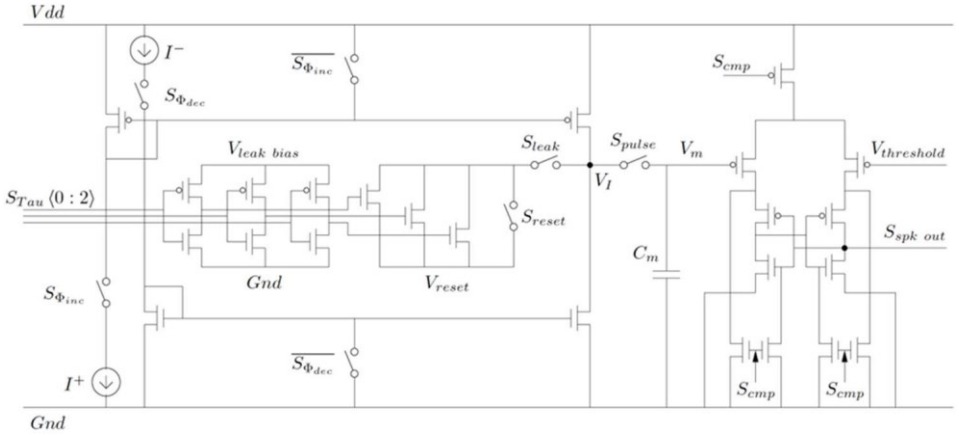

Fig. 2. show Example analog implementation of an LIF neuron.

The above Figrue 2, represent However, the largest problem for neuromorphic systems is finding ways to deal with the complex interconnection schemes of neurons. Human brains contain over a trillion synapses (neural connections) and even for smaller-scale systems attempting to replicate this is difficult with the limited metal layers in CMOS technology (one reason Dr. Parker has moved to carbon nanotubes). This is especially true for the process used in this project, which contains only 3 metal layers. Therefore, one solution is to take advantage of the speed of silicon and develop a system that time-multiplexes the neural spikes on one bus. This means that a single bus is implemented as a communication system for each neuron. Spikes are passed on the bus as they occur controlled by an arbiter. This allows the system to function effectively without the headache of all the wires. This implementation is typical for neuromorphic systems and has become the standard for cross-chip communication interfaces and is known as Address Event Representation (AER). This is because each packet of information contains the destination address of the event so that the routing system knows where to send the chip.

Because of the number of neurons implemented, True North and other networks require an extremely complex learning algorithm, because there are no traditional programming languages that can be applied to these chips . As one IBM fellow said, ”Given this unconventional computing paradigm, compiling C++ to True North is like using a hammer for a screw” . Rather neural networks must ”learn” how to respond. They are given sets of sample data and then they learn how to respond to this data with the hope that this training algorithm will apply to future data sets as well. Each neuron is connected to another neuron with a specific strength or weight and this weight determines the magnitude of the jump in membrane potential. The training algorithm adjusts these weights to adjust the behavior of the network. One popular training algorithm for neural networks, backpropagation, involves the computation of the error between the desired and actual output. This adjustment is performed on the weights of the last layer and then propagated backward to adjust the next layer. However, this model of programming is a) this is a completely separate paradigm of programming which is difficult for many programmers to learn after a lifetime of programming in sequential languages and b) this method requires a long amount of time due to the numerous simulations that must be run.

What follows is the description of an attempt to implement many of the above principles on a digital VLSI chip. This paper will discuss the design and implementation of that chip. Several features of that design include:

- The design contains 16 separate neural processing units that replicate a LIF neuron.
- Custom interconnection system based on published AER systems for transferring spikes.
- Programmable leak and threshold values.
- Theoretically scalable if new chips were to be fabricated.
- Off-chip EPROM for storing weights.
- Input can either come from a second EPROM or just a wired signal (e.g. a different chip).
- Output can also either come from a second (or third) EPROM or to a wired signal. Note that the output lines for the input and output lines are wired together for efficient pin implementation. This implementation has been named the CAT design by the authors.

This paper will go over the details of the implementation and the final layout and design. First, the design of the individual neuron and the neuron communication system will be presented. Second, the external interface will be discussed. Finally, the testing protocol will be described.

3.0 METHODS

Neuron

In this design, a LIF model was implemented based on its simplicity. In Izhikevich’s 2004 paperanalyzing the cost and biological realism of various models the LIF model was found to be

headache of all the wires. This implementation is typical for neuromorphic systems and has become the standard for cross-chip communication interfaces and is known as Address Event Representation (AER). This is because each packet of information contains the destination address of the event so that the routing system knows where to send the chip.

Because of the number of neurons implemented, True North and other networks require an extremely complex learning algorithm, because there are no traditional programming languages that can be applied to these chips . As one IBM fellow said, ”Given this unconventional computing paradigm, compiling C++ to True North is like using a hammer for a screw” . Rather neural networks must ”learn” how to respond. They are given sets of sample data and then they learn how to respond to this data with the hope that this training algorithm will apply to future data sets as well. Each neuron is connected to another neuron with a specific strength or weight and this weight determines the magnitude of the jump in membrane potential. The training algorithm adjusts these weights to adjust the behavior of the network. One popular training algorithm for neural networks, backpropagation, involves the computation of the error between the desired and actual output. This adjustment is performed on the weights of the last layer and then propagated backward to adjust the next layer. However, this model of programming is a) this is a completely separate paradigm of programming which is difficult for many programmers to learn after a lifetime of programming in sequential languages and b) this method requires a long amount of time due to the numerous simulations that must be run.

What follows is the description of an attempt to implement many of the above principles on a digital VLSI chip. This paper

will discuss the design and implementation of that chip. Several features of that design include:
the model with the lowest possible implementation cost. Since space was a large concern, we chose to use this model. The implementation here also uses a digital design. Most neuromorphic chips today (a principal exception being the IBM TrueNorth chip) use analog circuits since they require lower power and are more space-efficient. However, we chose a digital design because of the ease of construction and the testing and fabrication process available.
The diagram of our design is shown in Fig. 3. We have an enabled register that communicates with the weight bus to acquire signals that are sent from other neurons. We then have an accumulator unit that contains the current membrane potential. It checks the register output every cycle and adds the (signed) value to the current counter if there is a non-zero output detected. It then sends a reset signal to the register so this value is not added again the next clock cycle. If there is no register output then the leak is subtracted. It also checks if this potential is over the threshold for every clock signal. If it is, it will tell the communication block to begin communicating with the Arbiter. This design was based on the one shown above in Fig. 1 (and first presented in).

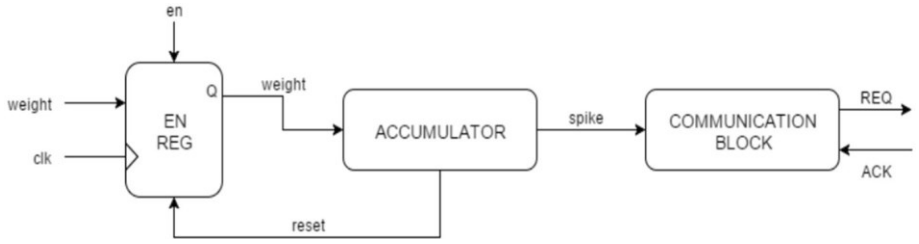

Fig. 3. Show Team CAT LIF neuron implementation

The above Figrue 3, represent Several modifications were made to the standard LIF model to make it more biologically realistic. For example, we have a programmable threshold and leak values. These values come in on the standard weight bus from the input line. Two programming lines tell the neuron to place these values in the leak or threshold values rather than add them to the accumulator. The only other inputs to the neuron are a global clock and a reset line.

Communication System

In this case, a time-multiplexed design system was implemented. This system was chosen because the final size of the LIF neuron was unknown during the design process but a total area limit of 2300μm ×2300μm was placed on the chip fabrication. As it was impossible to know how many neurons would be placed on the final chip during planning an easily scalable communication system was required. It was also anticipated that this would help constrain the design size and increase the number of neurons implemented as wiring a data line between each set of neurons would have been exceedingly difficult given only 3 layers of metal available. Finally, AER is an industry-standard neuromorphic engineering protocol and while the CAT team hoped to create a unique design, they also wanted to take advantage of the work done before them. The CAT design is not exactly AER standard but mirrors the methodology. The principle of AER is that neural communications are most often in the range of hundreds to thousands of Hz, whereas silicon operates in MHz Neuromorphic systems can take advantage of this speedup by multiplexing spikes across time. Neurons will still function the same but now a single bus can be used for all signals. The CAT design implemented a token-based arbitration system to control this bus. The design is shown in Fig. 4. There are three I/O buses to the neuron sets - request, acknowledge, and enable. The request line is a one-hot line to each neuron which tells whether that neuron has fired since the last check. Once the arbiter has processed this request line it sets the corresponding acknowledge line high to indicate to that neuron that it can pull down its request line. The enable line is also one-hot encoded and is sent to each neuron- enabled register to indicate when that neuron should be reading the bus. The request lines are checked sequentially, that is there is a token that is cycled through each neuron. This token is incremented every 16 clock cycles to allow a neuron to send out a signal to each If the neuron's request line is not high then the Arbiter still uses those 16 clock cycles to send out the input data, but the weight EPROM is turned off. If the neuron that is currently being checked does have a request line active then the weight EPROM will be activated and the address will be calculated from the index of the sending and receiving neuron. The signed index is returned and added to the input before being sent out on the weight bus. This system of logically cycling through the sending and receiving neurons allows the input to be set up sequentially so the user knows exactly which neuron they are sending each weight to. It also means that a fair token system is implemented where each neuron is given equal priority.

External Interface and Scalability

Fig. 5 shows the external interface to the CAT chip. The main component of the external interface is an EPROM where the weights are stored and 2-bit program input to tell the network what mode to be in. The EPROM interface consists of an 8-bit bus and a 10-bit address line. The EPROM has a14 bit address line, but the top segment of the address space is not used so these pins should be tied low. Every clock cycle, if a spike occurs in the neuron currently being checked by the arbiter, the chip will send out an address consisting of a 1 followed by four bits containing the index of the receiving neuron and then the index of the sending neuron. If no spike occurs an address of 0 is sent out, meaning a 0 should always be stored in address 0. If one is trying to program 4

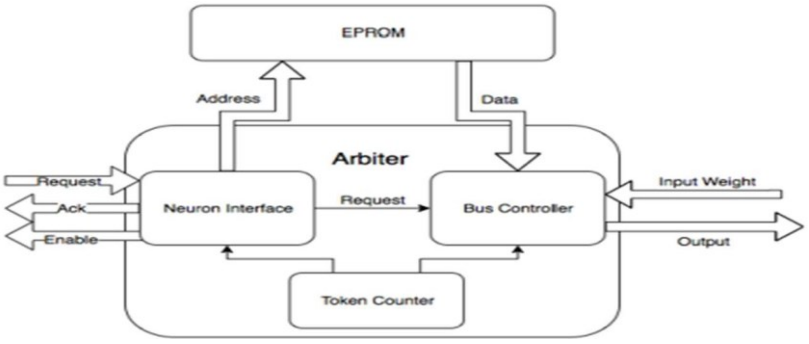

Fig.4. Show Team CAT token-based bus arbitration system

The above Figrue 4, represent Team CAT token-based bus arbitration system the threshold of the neurons, the programming bits can be set to 'b01, and the chip will send out an address of 1 to the EPROM, so the desired threshold should be stored there. The desired leak can be stored in address 2 of the EPROM and can be programmed by setting the programming bits to 'b10. The threshold and leak values can be set to a default value of 60 and 2 respectively by setting the programming bits to 'b11. The programming bits should be set to 'b00 for normal operation. Note that the weights will not overwrite these values because of the leading 1 on those addresses.

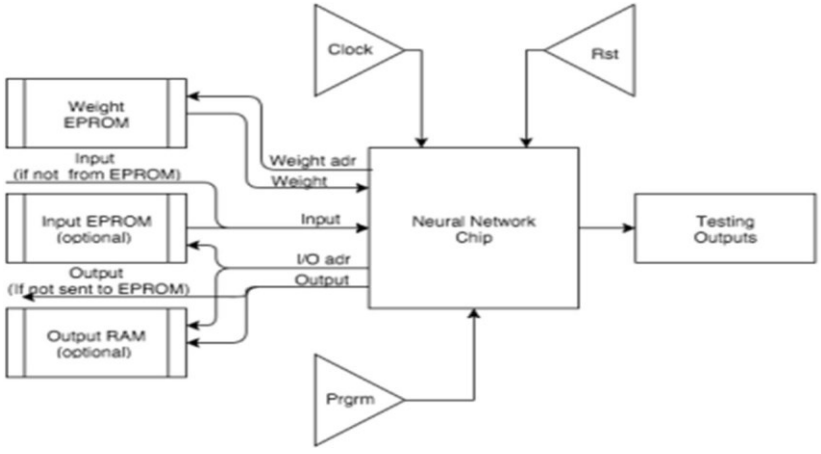

Fig. 5. Show External interface to the CAT chip

The above Figrue 5, represent External interface to the CAT chip the other I/O consists of a clock, reset the input and output lines, another address line, and debugging signals. The input signal is applied to whichever signal is currently being checked by the arbiter. It is simple to control which signal is applied to which neuron, by simply using another EPROM and storing the values sequentially through the address space. The second address line just increments with every clock cycle sequentially after a global reset. The output line can be wired up similarly, it can either be read by an external program or debugger, or wired up to an external RAM with the second address to store the output of the program for later viewing and analysis. In the case that either the input or output is stored or loaded from external memory, the enable lines of those chips should be tied active. Alternatively, the CAT system was designed to be scalable with other CAT chips. Most biological neural networks, contain millions or billions of neurons, meaning that our system of 16 neurons will not be able to recreate anything on a biological scale. Therefore, the output of one CAT line can be easily tied to the input line of another chip for a simple scalable design. However, they are not scalable in the sense that a neuron on one chip can target a neuron on another chip, merely that one chip can be built to extend and interpret the output of a previous chip. With this sort of design, a first chip could take care of low-level processing and pass it on for more high-level processing on the second chip. The debugging outputs consist of:

- The request lines of each neuron to see when they are spiking
- The token counter in the Arbiter to ensure that it is properly looping through the neurons
- The accumulator value of neuron 0 to ensure that a neuron is functioning properly.

VLSI Implementation

The CAT design will be fabricated in a 600nm process. The final layout of the chip is shown in Fig. 6. The layouts for the neuron and the arbiter were also developed during the debugging process. However, to save space in the final design we implemented all 16 neurons and the arbiter in one top module.

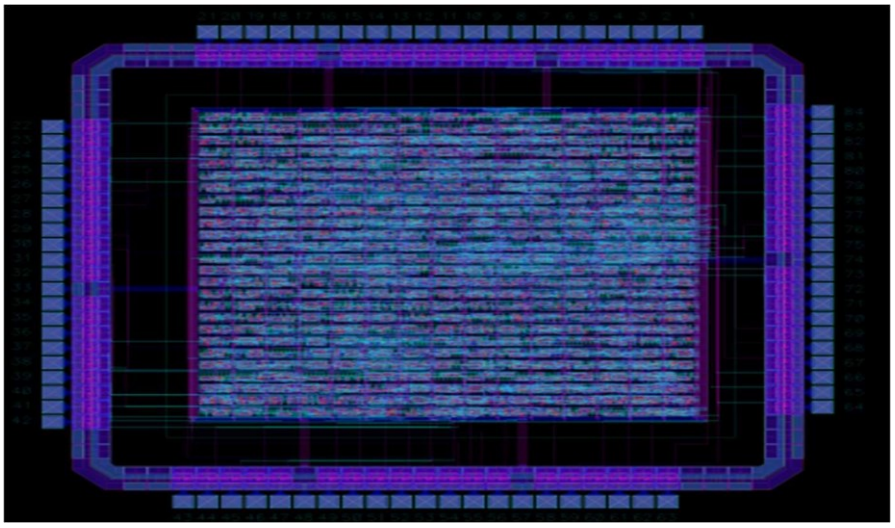

Fig. 6. Show Final chip layout of the CAT chip

The above Figure 6, represents The design was based on a custom standard cell library. The full specifications of this library can be seen in the library specifications document. However, in brief, the following cells were designed:

- **ADDER1X:** 1 bit adder 5
- **AOI 22:** And or Inverter with 4 inputs
- **BUF (X4, X8):** Buffer cell
- **DFF:** D flip flop with asynchronous reset
- **FILLER (1, 2, 4, 8):** Filler cell for extra space
- **INV (X1, X2, X4, X8):** Inverter cell
- **MUX2 INVX1**: Inverting multiplexer
- **NAND2(X1, X2), NAND3X1**: Various NAND cells
- **NOR2X1:** 2 input NOR gate
- **TIEHI:** Cell to tie lines high
- **TIELO**: Cell to tie lines low
- **XOR2X1**: 2 input XOR cell

The decisions for which cells to implement in our library came from logical analysis as well as the characterization and synthesis of various other libraries. The designers knew a DFF and basic
logic cells such as NAND, INV, and NOR were going to be needed because we had register files as well as various logic. The FILLER, TIEHI, and TIELO cells are standard library cells for most chip designs. The ADDER was implemented because the designers anticipated the neuron would need one for the accumulator. The other cells were chosen after synthesizing the Verilog code used to build the design with several other libraries. The Foo and OSU libraries were both provided by the University of Utah and these libraries contained various other cells. Originally, the Verilog code for a counter, a decoder an adder, a mux, a simple state machine, and the neuron and arbiter were synthesized using the 2 libraries and the other cells were chosen based on analysis of the space and timing of the resulting structural Verilog.

Testing

Several Verilog test benches were designed for the project. For more details on these testbenches see the design specification document. Individual simulations were run for the arbiter and neuron to ensure the proper functionality of both pieces. For the neuron, the simulation tested that the neuron could correctly add incoming weights, send off a spike when the weight went over the threshold, and be reset. For the arbiter the testbench ensured that the arbiter correctly cycled through the neurons (both receiving and sending), correctly acknowledged the neurons that had a request line high, correctly interfaced with the EPROM, correctly combined the EPROM and input data for the data bus, output the correct data, and enabled the correct neuron to receive data from the bus. The overall testbench tested all of the above as well as ensuring that the entire system could correctly interpret an input (i.e. a high input created a sequence of spikes in the neurons and a negative input correctly inhibited the neurons). All of these variables were checked by analyzing waveforms. While it is important to eliminate human error by developing self-checking testbenches, in this case, it was judged that the development cycle was short enough, and the chip's nature was complex and non-intuitive enough that a self-checking testbench would have been more hindrance than help.

Instead of developing a self-checking testbench, the designers implemented an idealized version of the network in Python. This version was developed and tested and then the hardware version was tested for reliability against the network by passing both programs the same input and weight files and analyzing the output. This method of testing also allows the users to use the idealized Python version of the network to develop an algorithm to set the weights of the network for different operations.

4.0 RESULTS

A. Verilog Testing

The network successfully passed each Verilog test that was applied to it. It was able to successfully set weights and leaks, interpret input, update neurons, and propagate spikes through the network exactly as we expected.

B. Python Module Comparison

The network successfully replicated the Python module. The same inputs and weight matrix were applied to both modules and spikes occurred at the same time on the same neurons. This indicates that the network works as expected and is ready for operation. During the development process, various versions of the Python module were also used for training the network. The network was trained with a simple genetic algorithm that tried a series of weighting configurations, simulated the network, compared the output of each, and then took the best network and used it as the starting point for the next series of iterations of random changes to the weighting scheme.

C. Potential Applications

As mentioned above neural networks are ideal for a variety of applications, but especially interpreting natural stimuli such as auditory or visual signals, or interpreting data that contains a variety of unconnected inputs. One potential application envisioned is the isolation of a specific frequency component from a signal. So, if a signal is passed to the network, it would respond if a specific frequency component was in that signal, but not if that frequency was not there. Another application that

is envisioned is the prediction of sports data. The network could be passed a series of data points on specific players and the network would predict which players would be the best during the upcoming season. This would allow the user to have better data during a fantasy football draft. However, the exciting part of neural network computing is that, like a standard processor, it can be applied to any number of applications depending on how it is programmed. For a standard processor this is the set of instructions. For a neural network, it is the scheme of interconnecting weights.

CONCLUSION

This paper describes the development and design of a 16-neuron neural network. The network was based on research done into current neuromorphic engineering technology and designs and based on the LIF neuron model and the AER communication system. The network was developed with a custom standard cell library and will be fabricated using a 6 600nm chip. The network was tested with a series of Verilog test benches as well as an idealized Python version of the code.

This design is an implementation of a novel way of computing. While powerful, this method of computing is non-intuitive and requires a completely different paradigm of thinking. Yet the authors believe that the design presented here is a workable implementation of this paradigm and are excited to implement various systems on the chip. With that said, several things could be improved in a second-generation chip. First of all, the authors hope to implement many more neurons in the design, if possible, as the LIF neuron implemented in this design was far larger than other published designs. Increasing the efficiency of used space could increase the number of neurons available and the computational power. The authors also hope to implement an overflow comparison and handling which was not placed in this design. This will still allow the network to function effectively as long as the network is operated with inputs and weights that are less than half of the bus capacity. Finally, the authors learned to prioritize designs that used lower metal layers rather than those that were compact. While compact designs may be effective for processes with more metal layers, they just serve as a blockage for cells and cause problems for the router. Cells that take a little more space, but are all metal1, end up requiring much less space when considering interconnection wiring.

REFERENCES

- W. Maass, Networks of spiking neurons: The third generation of neural network models, IEEE Trans. Neural Netw. 10(9) (1997) 1659–1671.
- D. Z. Jin, Spiking neural network for recognizing spatiotemporal sequences of spikes, Phys. Rev. E 69 (2004) 021905–021918.
- A. L. Hodgkin and A. F. Huxley, Resting and action potentials in single nerve fibres, J. Physiol. 104(2) (1945) 176–195.
- A. L. Hodgkin and A. F. Huxley, The components of membrane conductance in the giant axon of Loligo, J. Physiol. 116(4) (1952) 473–496.
- M. Pospischil, M. Toledo-Rodriguez, C. Monier, Z. Piwkowska, T. Bal, Y. Fregnac, H. Markram and A. Destexhe, Minimal Hodgkin-Huxley type models for different classes of cortical and thalamic neurons, Biol. Cybern. 99(4-5) (2008) 427–441.
- E. M. Izhikevich, Simple model of spiking neurons, IEEE Trans. Neural Netw. 14(6) (2003) 1569–1572.
- E. M. Izhikevich, Which model to use for cortical spiking neurons?, IEEE Trans. Neural Netw. 15(5) (2004) 1063–1070.
- D. Yudanov and L. Reznik, Scalable multi-precision simulation of spiking neural networks on GPU with OpenCL, in Proc. 2012 Int. Joint Conf. Neural Networks (Brisbane, Australia, 2012),
pp. 1–8.
- P. Arena, L. Fortuna, M. Frasca and L. Patane, Learning anticipation via spiking networks: application to navigation control, IEEE Trans. Neural Netw. 20(2) (2009) 202–216.
- A. N. Burkitt, A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input, Biol. Cybern. 95(1) (2006) 1–19.
- M. J. Chacron, K. Pakdaman and A. Longtin, Interspike interval correlations, memory, adaptation, and refractoriness in a leaky integrate-and-fire model with threshold fatigue, Neural Comput. 15(2) (2003) 253–278.
- Y.-H. Liu and X.-J. Wang, Spike-frequency adaptation of a generalized leaky integrate- and-fire model neuron, J. Comput. Neurosci. 10(1) (2001) 25–45.
- C. R. Huyck and R. V. Belavkin, Counting with neurons: Rule application with nets of fatiguing leaky integrate and fire neurons, in Proc. 7th Int. Conf. on Cognitive Modelling (Trieste, Italy, 2006), pp. 142–147.
- E. Nichols, L. J. McDaid and N. H. Siddique, Case Study on A Self-Organizing Spiking Neural Network for Robot Navigation, Int. J. Neural Syst. 20(6) (2010) 501–508.
- T. J. Strain, L. J. McDaid, T. M. McGinnity, L. P. Maguire and H. M. Sayers, An STDP Training Algorithm for A Spiking Neural Network with Dyanmic Threshold Neurons, Int. J. Neural Syst. 20(6) (2010) 463–480.
- J. L. Rossello, V. Canals, A. Morro and A. Oliver, Hardware Implementation of Stochastic Spiking Neural Networks, Int. J. Neural Syst. 22(4) (2012) 1250014-1–1250014-11.
- J. Iglesias and A. E. P. Villa, Emergence of Preferred Firing Sequences in Large Spiking Neural Networks During Simulated Neuronal Development, Int. J. Neural Syst. 18(4) (2008) 267–277.
- W. Gerstner and W. M. Kistler, Spiking neuron models: Single neurons, populations, plasticity. (Cambridge Univ. Pr., Cambridge, United Kingdom, 2002).
- A. L. Hodgkin and A. F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve, J. Physiol. 117(4) (1952) 500–544.
- S. Schliebs, N. Nuntalid and N. Kasabov, Towards spatio-temporal pattern recognition using evolving spiking neural networks, in Proc. 17th Conf. Neural Information Processing. Theory and Algorithms eds. K. Wong, B. S. Mendis and A. Bouzerdoum (Sydney, Australia, 2010), pp. 163–170.

- J. L. Rossello, V. Canals, A. Morro and A. Oliver, Hardware Implementation of Stochastic Spiking Neural Networks, Int. J. Neural Syst. 22(4) (2012) 1250014-1–1250014-11.
- J. Iglesias and A. E. P. Villa, Emergence of Preferred Firing Sequences in Large Spiking Neural Networks During Simulated Neuronal Development, Int. J. Neural Syst. 18(4) (2008) 267–277.
- W. Gerstner and W. M. Kistler, Spiking neuron models: Single neurons, populations, plasticity. (Cambridge Univ. Pr., Cambridge, United Kingdom, 2002).
- A. L. Hodgkin and A. F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve, J. Physiol. 117(4) (1952) 500–544.
- S. Schliebs, N. Nuntalid and N. Kasabov, Towards spatio-temporal pattern recognition using evolving spiking neural networks, in Proc. 17th Conf. Neural Information Processing. Theory and Algorithms eds. K. Wong, B. S. Mendis and A. Bouzerdoum (Sydney, Australia, 2010), pp. 163–170.