Verification Workflow For Neuromorphic Digital Hardware On FPGAs

Bryson Gullett
University of Tennessee
Knoxville, Tennessee, USA
bgullet1@vols.utk.edu

Jonathan Ting
University of Tennessee
Knoxville, Tennessee, USA
jting@vols.utk.edu

Abstract

Neuromorphic computing is a branch of computing that focuses on processing systems inspired by the structure and behavior of the biological brain. The TENNLab Neuromorphic Computing research group at the University of Tennessee studies Spiking Neural Networks (SNNs), which are a type of neural network that more closely resembles the brain's neurons interconnected by synapses compared to traditional neural networks. The TENNLab has recently completed an FPGA implementation of custom neuromorphic hardware called RAVENS, which is designed to run SNNs efficiently; however, it has yet to be rigorously tested. Further, the infant state of neuromorphic research means that there are no neuromorphic hardware-in-the-loop testing tools currently available. The goal of this project is to design an automated verification workflow for the TENNLab RAVENS neuromorphic processor FPGA implementation. This testing framework is written using a software engineering unit testing framework, hardware automation code, and TENNLab-specific simulation and deployment software. The RAVENS FPGA verification framework has successfully uncovered bugs in the TENNLab RAVENS SNN deployment pipeline and shown that the RAVENS digital hardware behaves exactly like its C++ simulator golden model in all cases thus far. Though more in-depth verification remains, this project has reinforced the TENNLab's confidence in the correct functionality of digital RAVENS hardware, and it represents a major step forward in terms of hardware-in-the-loop unit testing of neuromorphic processors.

Keywords: Neuromorphic, FPGA, Spiking Neural Networks, Digital Hardware, Hardware Verification, Software Testing

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Bryson Gullett and Jonathan Ting. 2024. Verification Workflow For Neuromorphic Digital Hardware On FPGAs. In . ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The TENNLab Neuromorphic Computing research group at the University of Tennessee does its research in building applications, algorithms, architectures, and devices that can create and run Spiking Neural Networks (SNNs). SNNs are more biologically inspired by the brain compared to traditional Artificial Neural Networks (ANNs). Unlike ANNs, SNNs perform processing using sparse neuronal firing ("spike") events instead of dense matrix multiplications. SNNs often do not have strict network architectures and are often small, highly recurrent "hairballs". SNNs have parameters besides weights that need to be set in order to model biological neurons and synapses. Unfortunately, the complexity of SNNs means that they lack standardization and an ideal training algorithm like ANNs' backpropagation with gradient descent. Some key advantages of running SNNs on dedicated neuromorphic hardware compared to existing computing systems are:

- **Event-Based Computing**: Since SNN processing revolves around neurons firing and transferring data to other neurons via synapses, a neuron only needs to spike if there is a significant event that occurs. Otherwise, it can remain idle. The processing and transfer of data only when necessary (known as event-based computing) is extremely efficient both in terms of speed and power.
- **Collocated Processing and Memory**: Having all the processing and memory live inside an SNN as neuronal potentials, spikes, and charges moving along synapses prevents costly memory transfers to and from a centralized compute unit. Once again, this SNN feature is efficient in terms of both speed and power.
- **Massive Parallelism**: Activity in an ideal SNN is processed completely in parallel, which allows for quick, low-latency computation.

To reap most of SNNs' benefits, the user should run them on dedicated neuromorphic hardware; however, little commercial neuromorphic hardware is currently available as the technology requires continued research before successful

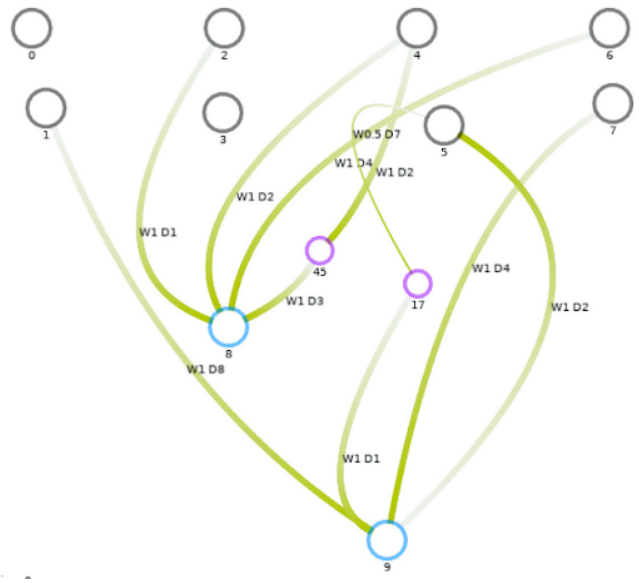

Figure 1. An example of a small, recurrent Spiking Neural Network (SNN) trained using an evolutionary algorithm in the TENNLab Neuromorphic Computing group.

commercialization. To contribute towards neuromorphic processor research, the TENNLab group designed the RAVENS (Reconfigurable and Very Efficient Neuromorphic System) architecture to run SNNs [1]. RAVENS has clusters of neuromorphic cores that efficiently process an entire SNN's activity in parallel. As of today, RAVENS currently has the following implementations:

- C++ Simulator
- Microcontroller in Embedded C
- Field-Programmable Gate Array (FPGA) in SystemVerilog
- Digital Application Specific Integrated Circuit (ASIC)
- Analog ASIC Using Memristive Devices for Synapses

The RAVENS C++ simulator is part of the TENNLab neuromorphic computing framework and has been extensively verified throughout years of being used for a variety of research projects. However, the RAVENS SystemVerilog used for all RAVENS hardware implementations has been completed much more recently (summer 2023). As a result, the process of deploying SNNs from simulation to hardware RAVENS lacks maturity, and hardware RAVENS design itself requires thorough testing to ensure it matches expected behavior demonstrated by the RAVENS C++ simulator.

This project focuses on the automated verification of the digital RAVENS FPGA implementation to ensure it behaves exactly the same as the RAVENS C++ simulator. In addition to just the SystemVerilog HDL that defines the behavior of the RAVENS hardware, this project also automates and verifies the software tools used to take SNNs from simulation and deploy them to RAVENS hardware. The RAVENS FPGA implementation was chosen for the following reasons:

- FPGAs' reconfigurability allows for quick programming of new SNNs during testing.
- The SystemVerilog used to program RAVENS SNNs onto FPGAs is the same HDL used to design the digital RAVENS ASIC.
- The RAVENS ASICs are currently being fabricated and are not yet available for TENNLab testing.
- Consumer FPGA development boards have many interfacing options that allow for a host machine to easily connect to the RAVENS SNN FPGA accelerator.

2 Methods

This project's automated RAVENS verification framework is built upon a large set of tools, both new and previously existing. A single test case requires functions that automatically perform each of the following:

- Randomly generate one SNN
- Randomly generate spikes to apply to the SNN as input data
- Simulate the SNN with the input spike data using the RAVENS C++ simulator and write output spike raster data to a file
- Compile the SNN from the TENNLab JSON file representation to RAVENS SystemVerilog and configuration files
- Convert input spike data into RAVENS packets
- Create and setup a Vivado FPGA project for the SNN
- Go through the Vivado FPGA workflow to program RAVENS with the desired SNN onto the FPGA
- Send the input spike data from the host computer to the RAVENS SNN on the FPGA
- Receive output spike data from the RAVENS SNN on the FPGA
- Convert output spike data to spike raster data and write it to a file
- Diff the ground truth C++ simulator spike raster file with the hardware spike raster file and ensure they perfectly match

Figure 2 visualizes all of the automated RAVENS verification tools working together to complete a single RAVENS test in the form of a block diagram. Notice that each piece of functionality in the testing framework can fit into one of the following categories:

- Top-level testing
- Test case generation using the TENNLab software framework
- Obtaining ground truth data from the TENNLab software framework's RAVENS C++ simulator
- Compiling test cases for RAVENS hardware
- Hardware integration and automation

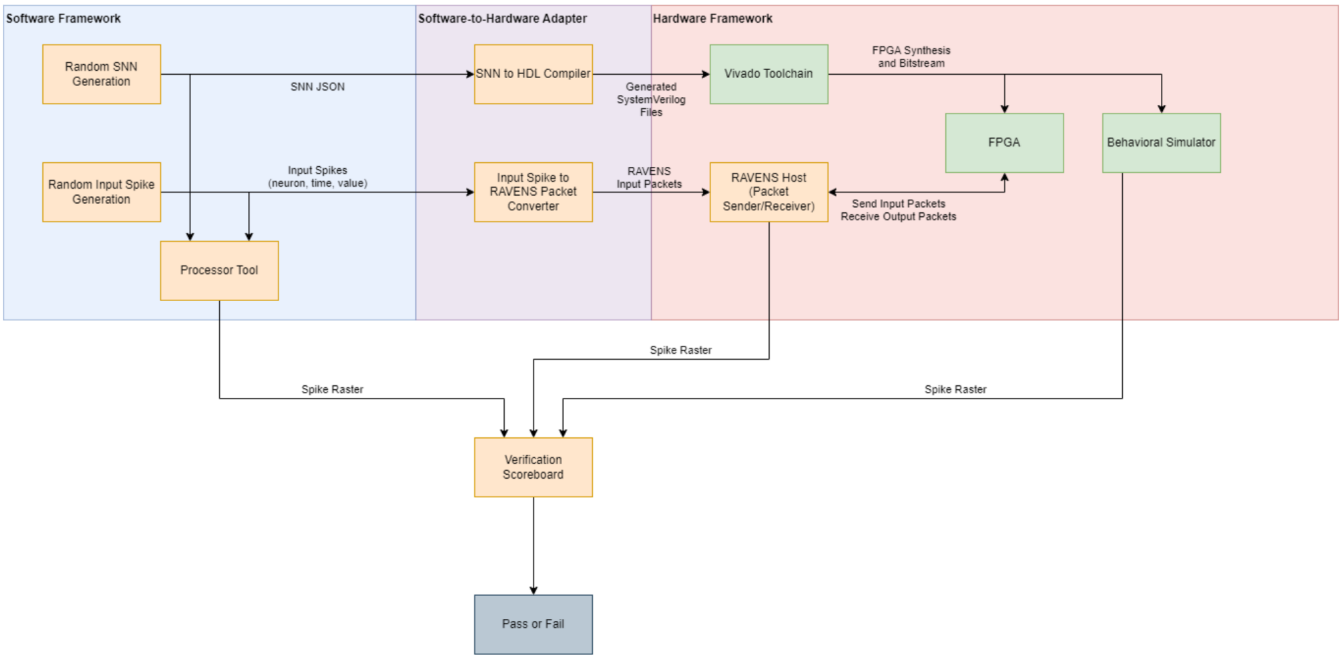

Figure 2. A block diagram that visualizes the functionality of the automated RAVENS verification project.

- Verifying RAVENS hardware outputs with ground truth simulation outputs

2.1 Top-Level Test Script

The top-level test script is the component of the automated RAVENS verification framework that is most inspired by advanced software engineering workflows. Modern software developers often use a Test-Driven Development (TDD) approach where they write tests to establish behavioral requirements before writing the code that accomplishes the desired behavior. pytest is the testing framework of choice for today's Python programmer that follows TDD's test-focused practices. Consequently, this project adopts the widely used pytest framework to perform what is effectively RAVENS hardware-in-the-loop Python unit tests.

The top-level test script uses pytest, which provides a structured, robust, and formalized framework to unit test the RAVENS hardware alongside the tools used to deploy SNNs from simulation to the hardware. The script has a single function that wraps and calls all of the other aforementioned functions required for a single test. This single testing function has parameters that help define the:

- Size and connectivity of the random SNN
- Target neuromorphic processor (a reduced version of RAVENS is also available)
- Duration the neuromorphic processor is run
- Arrangement of hardware neuromorphic cores
- Seed for random number generation

The top-level test script takes advantage of pytest's parametrize feature, which allows users to define multiple separate tests that use the same function using a list of tuples, where each tuple defines the arguments to the function's parameters for the corresponding individual test. The parametrize feature allows the project to use a single RAVENS test function where a list of function arguments can be generated, depending on what kind of SNN test the user would like to perform.

For the sake of testing this automated RAVENS verification workflow, tests were created with function arguments that were all identical (besides the random number generation seed) to SNNs that have been previously tested on RAVENS hardware with success. Also, for performance characterization purposes, another set of tests were created that only vary the sim_time parameter, which adjusts the duration in which the neuromorphic processor is run and how many input spikes it gets between resets.

2.2 Test Case Generation

Test case generation involves two different components that were written in Python for this project: random SNN generation and random input spike data generation. Random SNN generation is done by generating a random directed graph with the rustworkx Python package that is primarily written in Rust to maximize performance. Then, the code uses the TENNLab framework to create an SNN, loops through every node in the randomly generated graph, adds a corresponding neuron to the TENNLab SNN with randomized properties,

loops through every edge in the graph, and adds a corresponding synapse to the TENNLab SNN with randomized properties. The code then saves the TENNLab SNN as a JSON file.

The TENNLab SNN is created using a template JSON file that acts as an empty template SNN. The user can change the minimum and maximum values for neuron and synapse properties using this empty SNN JSON file if desired.

Random input spike data generation is done by sampling from a binomial distribution to determine how many SNN input neurons will spike in a single discrete timestep. SNN input neurons are sampled without replacement this many times to obtain all of the input spike data for a timestep. This process is repeated for every timestep that the neuromorphic processor will be run.

2.3 Obtaining Ground Truth Data

The previously existing TENNLab framework's RAVENS C++ simulator is used to obtain ground truth spike raster data. A spike raster is a 2D binary matrix where each row corresponds to a single neuron, each column corresponds to a discrete timestep running the neuromorphic processor, and each element represents whether or not the given neuron fired in the given timestep. For this project, a test's randomly generated SNN is loaded via Python into the TENNLab RAVENS simulator and run with the randomly generated input spike data. The simulator has a simple function to get the resulting spike raster for the SNN's output neurons. This spike raster is written to a file for comparison later on in the verification workflow.

2.4 Compiling Test Cases For RAVENS Hardware

This project adapts two previously existing TENNLab tools used for compiling SNNs and SNN data to RAVENS hardware representations. The first tool compiles RAVENS SNNs. SNNs in the TENNLab group are created in simulation and stored as JSON files. They can be almost any directed graph with the only rule being that there cannot be any parallel (multiple) edges. Therefore, an SNN graph must be mapped from JSON to the neuromorphic cores and clusters of RAVENS hardware. The TENNLab has an existing Python script to do SNN compilation from JSON to SystemVerilog and configuration files; this project iterated upon the script by fixing discovered bugs and by organizing the code into framework-friendly functions.

The second compilation tool used in this project is focused on taking RAVENS input spike, run, and reset instructions used by a TENNLab neuromorphic processor simulation tool and converting them to 32-bit packets that can be sent to RAVENS hardware. Like with the SNN compilation script, this script is also written in Python and incorporated into this project by being organized into functions. The project uses the converted 32-bit RAVENS packets by writing them to a file for later use.

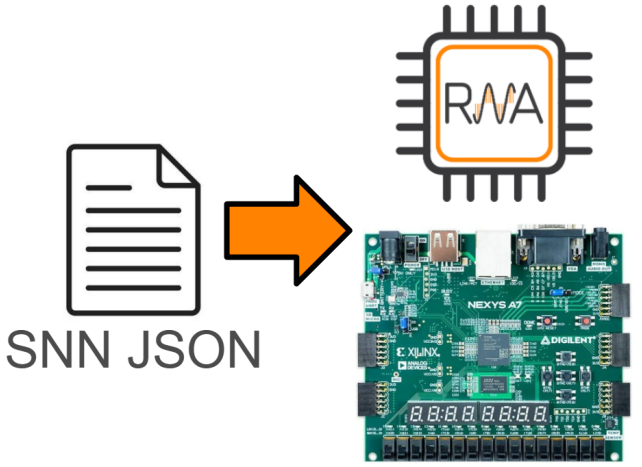

Figure 3. TENNLab Spiking Neural Networks (SNNs) must be compiled from JSON files specifically to run on RAVENS hardware.

2.5 Hardware Integration and Automation

This project is designed to be compatible with any core designed in Verilog, SystemVerilog, or both. For a custom core to be supported, the core must contain a Universal Asynchronous Receiver/Transmitter (UART) interface that must be communicable through any FPGA's built-in UART ports. UART was chosen over other interfaces such as SPI due to its ease of use, which does not require custom hardware or drivers. Most computers have serial ports that can communicate over UART, with the only hardware needed being a USB cable connection between the computer running the framework and the FPGA running the core.

The UART driver on the core sends data 1 byte at a time. However, input packets are encoded with a 32-bit width, meaning the core will concatenate 4 received input bytes together to form a packet. The framework on the computer will also do an equivalent operation, concatenating output bytes into 32-bit width packets.

Different cores will need to have different handling of the output received from the core by the framework. RAVENS input packets encode its input data in three different types of packets:

- Program packets initialize the network.
- Run packets run the network on the core for a specified amount of time.
- Reset packets reset the network to delineate between different observations.

The input packet type is denoted by the output type field, which uses the three most significant bits of the input packets to encode the type.

The output packet has two main types:

- Output packets encodes information on the output spike.

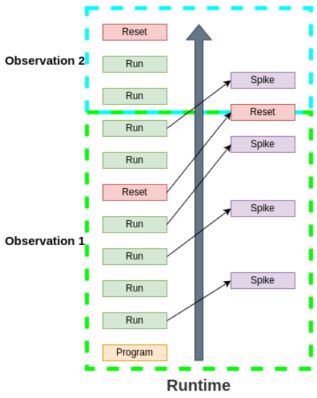

Figure 4. A typical RAVENS core UART communication flow.

- Reset packets denotes that a reset has occurred. Used by the framework for synchronization when delineating between observations.

It also has three main fields in which it encodes the output spike information:

- Output type field indicates whether it is an output packet or reset packet.
- Time step field indicates the timestep at which the spike occurred during the network run.
- Neuron field encodes the internal address of the output neuron where the spike occurred.

```
Observation 3
20 : 34 : 00000111001111101101
21 : 18 : 00011000111000000000
22 : 4  : 00000000000000000000
23 : 2  : 00000000000000000000

Observation 4
20 : 42 : 00000000000100000001
21 : 19 : 00000000000000000000
22 : 6  : 00000000000000000000
23 : 2  : 00000000000000000000
24 : 1  : 00000000000000000000
```

Figure 5. A spike raster generated from parsed FPGA output packets.

These output packets are then used to create the spike raster, where each output packet denotes a single spike, which is indicated in the "1's" on the spike raster strings. The spike raster serves as the final output of each test, which can then be compared against the software simulation equivalent.

	Vivado Project-based Workflow	New Script-based Workflow
Initial Setup Steps	Import core design files into Vivado project	Install neuromorphic framework
	Configure project settings	Import core design files into script project
	Create block diagram	
Steps Per Network	Generate and change out network-specific design files	Set parameters
	Generate bitstream in Vivado	Run pytest command in project directory
	Run program script to program core	
	Run neuromorphic framework to produce results to compare against	
	Run validate script to compare FPGA results with framework results	

Table 1. Manual steps for Vivado project-based workflow vs. new script-based workflow

2.6 Verifying RAVENS Hardware Outputs

3 Results

The two main goals of this project were to:

1. Verify hardware and software implementations of neuromorphic architecture through the creation of an automated framework that generates random stimulus.
2. Automate the setup process and speed up the time it takes to run and verify a single unit test.

Based on the results, our new framework has successfully accomplished both goals, improving the accuracy of the neuromorphic core and streamlining the process of testing the neuromorphic core.

To test the accuracy of the core, 25 randomized networks were generated and tested on the RAVENS core. Since the framework automates the generation, programming, and verification of networks, the only human input necessary was changing the parameter for networks to test. All 25 networks were found to have matching spike rasters in both the FPGA implementation and software implementation. The value of random network testing will be more useful in further development, catching any regressions in accuracy after breaking changes have been implemented.

The new script-based workflow also reduced the amount of human oversight and time needed to run a series of verification tasks on the neuromorphic core. Both the old Vivado project-based workflow and the new script-based workflow have an initial setup time, however, the script-based workflow only requires a few commands to be run, compared to the knowledge necessary to setup a Vivado project. The greatest advantage of the script-based workflow is running all 6 of the steps to validating a network with just one command. The script can also queue up multiple networks to be tested, meaning the command only needs to be run once for a session.

The script-based workflow also results in significant time-saving per verification session. Notwithstanding the speedup in initial setup, which is negligible after the one-time setup is complete, a significant speedup was observed after running the same 5 network tests in the old workflow versus the

new workflow. After averaging the 5 runs, the following results were observed. The runtime of generating bitstream was reduced from 10.4 minutes to 9.65 minutes. This can be attributed to how executing the Vivado GUI adds cycles of overhead to the operations. Programming the core took the same amount of time since both use the same Python script. Validating results is marginally faster, since the old workflow involved manual diff checking of spike rasters. The new workflow can automate this, reducing this runtime by half a minute. Lastly, the old workflow required manual network design file swapping in Vivado, while the new workflow does this automatically, which saves about a minute. The result is approximately 2.22 minutes saved per network, which results in a 13% speedup in runtime of test per network compared to the old workflow.

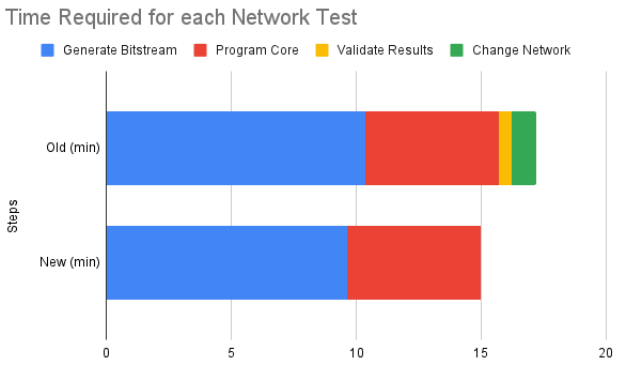

Figure 6. Runtime of old workflow vs. new workflow per test.

4 Conclusion

This project successfully creates an automation and verification framework used to test the FPGA implementation of the TENNLab RAVENS neuromorphic processor and its necessary deployment software. The project required the development of new hardware-in-the-loop unit testing software, the adaptation of existing deployment tools, and the streamlining and automation of hardware workflows. The running of this RAVENS verification framework uncovered bugs in the RAVENS deployment pipeline while also demonstrating that the hardware functions identically to the TENNLab RAVENS C++ simulator golden model thus far. In the future, this project will be used to do more rigorous RAVENS hardware verification that allows test arguments to be more radically varied.

Since neuromorphic technology across the globe is still in its infancy and existing research is extremely diverse, neuromorphic hardware testing methodology is severely lacking. The hope for this project's verification workflow for digital neuromorphic hardware is for it to not only act as a useful tool for the TENNLab to ensure their existing and future neuromorphic processor hardware and deployment software behave as expected, but also for the framework to inspire better neuromorphic testing tool chains down the road. Testing is an integral component to success.

5 Author Contributions

5.1 Bryson Gullett

Bryson Gullett primarily worked on the software side of this project. He wrote the top-level test script, random SNN generation, random spike input data generation, and ground truth spike raster generation parts of the verification framework. Further, he edited the existing input spikes-to-RAVENS packets and SNN-to-RAVENS hardware compilation scripts to integrate them into the project.

5.2 Jonathan Ting

Jonathan Ting primarily worked on the hardware side of this project. He wrote the hardware-interfacing scripts, communication designs, Vivado TCL script templates, Verilog core communication design code, and hardware input sender/output parser scripts. He also created the benchmarks for the new framework and ran the verification tests.

References

[1] Adam Z Foshie, James S Plank, Garrett S Rose, and Catherine D Schuman. 2023. Functional specification of the ravens neuroprocessor. *arXiv preprint arXiv:2307.15232* (2023).