Verification Workflow For Neuromorphic Digital Hardware On FPGAs

Jonathan Ting and Bryson Gullett

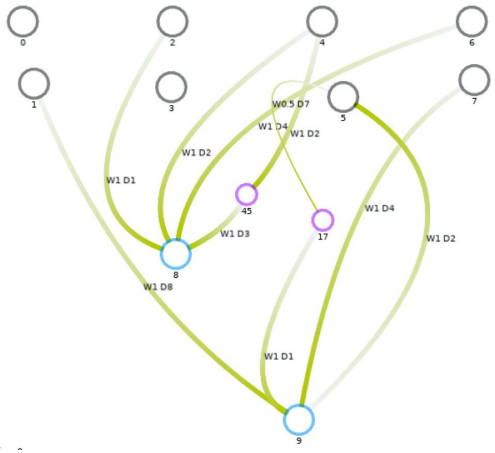
THE UNIVERSITY OF TENNESSEE KNOXVILLE

TENN LAB NEUROMORPHIC ARCHITECTURES. LEARNING. APPLICATIONS.

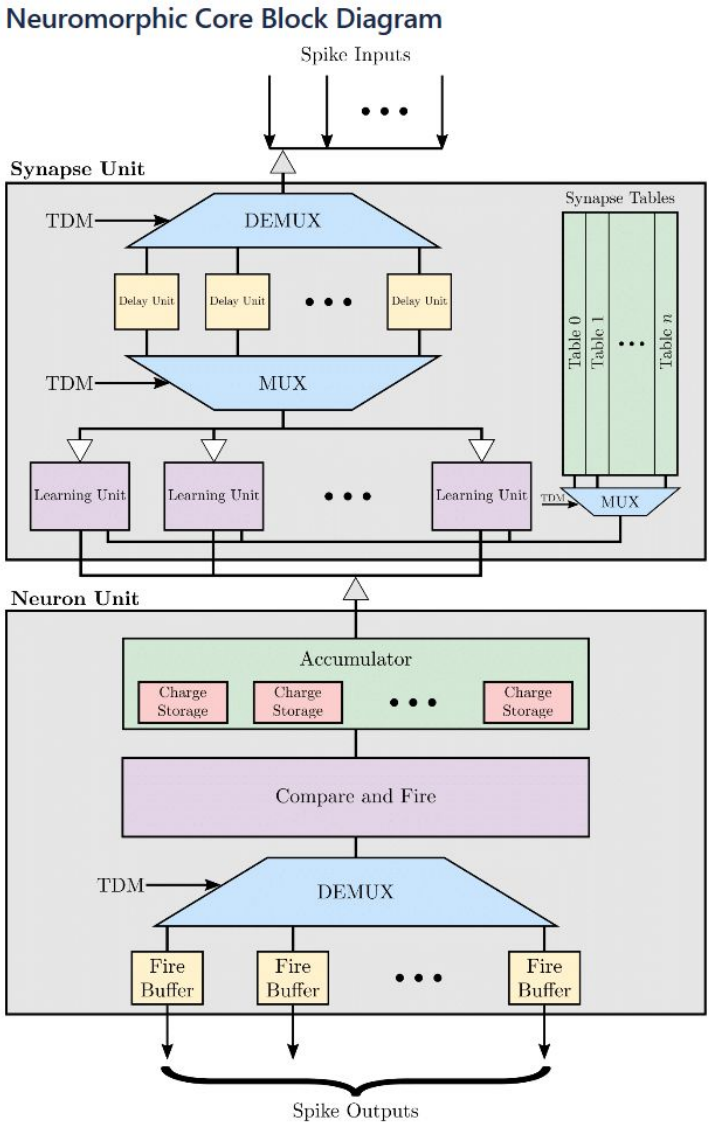
Background and Introduction

Spiking Neural Networks (SNNs)

- Neurons interconnected by synapses like the biological brain
- Key differences from traditional ANNs:
  - Spike events (neuron firings) instead of dense matrix multiplications
  - No strict layer structure (“hairball” networks)
  - More parameters to model biological neurons and synapses
  - Neuron and synapse models are not standardized
  - No perfect training algorithm like backpropagation + gradient descent
- Key advantages:
  - Energy efficiency due to nature of event-based processing
  - Massively parallel
  - Collocated processing and memory (leads to even more efficiency)



RAVENS Digital Neuroprocessor



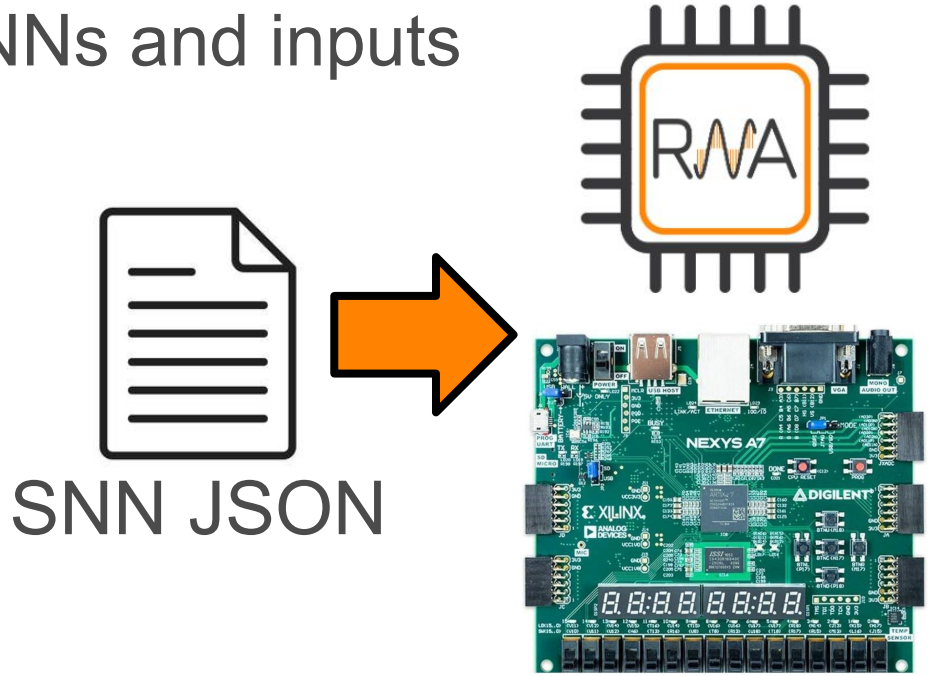
- RAVENS runs SNNs natively on custom digital hardware
  - FPGA and ASIC implementations
- Clusters of neuromorphic cores that process SNN activity in parallel
- Developed by other students in the TENNLab
  - Finished development last summer

Existing RAVENS FPGA Workflow

- Train SNN with existing TENNLab framework tools for desired app
  - SNN stored as simple JSON file
- Map the SNN to RAVENS clusters, cores via a Python script
- Create Vivado project with all sources generated from mapping
  - Target specific FPGA board
- Use Vivado to synthesize, implement, generate bitstream, and program SNN onto FPGA using bitstream
- Finish loading SNN parameters by sending them to FPGA from host computer via UART
- Inputs and outputs communicated between host and FPGA via UART

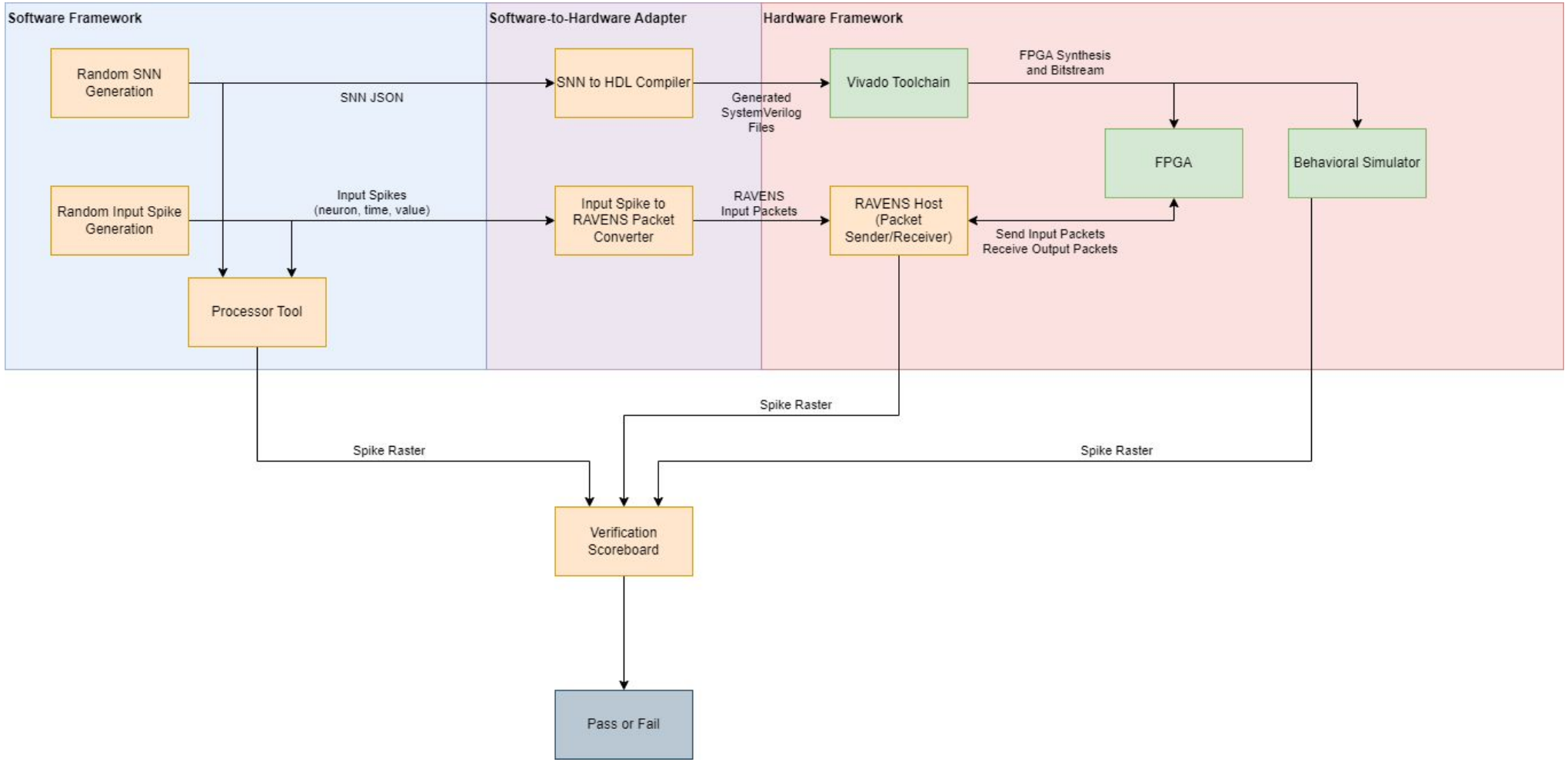
Our Goal: Automated RAVENS Testing

- Ensure FPGA behaves exactly the same as software simulator
- Automate entire RAVENS workflow
- Perform workflow repeatedly for different SNNs and inputs
- Use existing software testing framework
  - Effectively perform a unit test on RAVENS FPGA
- Support different FPGA boards
- Support RISP FPGA as well
  - Reduced version of RAVENS



Approach/Methodology

Structure of a Single RAVENS Test



Verification Components Written

Software Side	Hardware Side
• Top-level pytest test script • Random SNN generator • Random SNN input generator • RAVENS SNN mapping tool (adapted, not fully written)	• UART interface for core communication • Bitstream generation • Spike raster generator • Software vs. hardware validation

Top-Level pytest Test Script

pytest

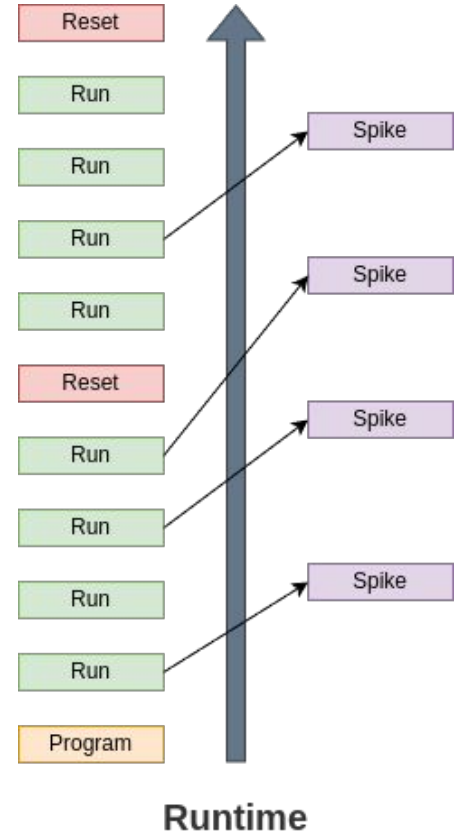
- Function that performs a single test
  - Integrates every component of the verification framework
    - Each component has its own Python function that is called
- Generate list of tests
  - Single test described by arguments to aforementioned function
- pytest parametrize feature used to automatically run test function with all generated tests

Pass or Fail?

- RAVENS FPGA outputs are spikes on SNN output neurons
- **Spike Raster** = 2D matrix where each row corresponds to a neuron and each column corresponds to a RAVENS processing timestep
- Create spike rasters from RAVENS FPGA outputs sent to host machine
  - Written to simple .txt file
- Create spike rasters from TENNLab framework's RAVENS simulator
  - Written to simple .txt file
- Diff hardware and software spike raster .txt files
  - If identical => pass
  - Else => fail

Hardware Communication Overview

- **32-bit** wide input packets are transmitted over **UART**.
- **Input packets** come in 3 forms
  - **Program** - initializes the network
  - **Run** - runs the network for a specified amount of time
  - **Reset** - resets the network for a new observation
- **Output spikes** are received over UART as **32-bit** wide output packets with the following fields
  - **RNA time step** - time step when the spike occurred
  - **AER packet** - address of the spiking neuron



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0
Output Type (3 bits)			RNA Time Step (16 bits)																AER Packet (13 bits)													
																			Core Address (4 bits)				Node Address (4 bits)				Synapse Address (5 bits)				

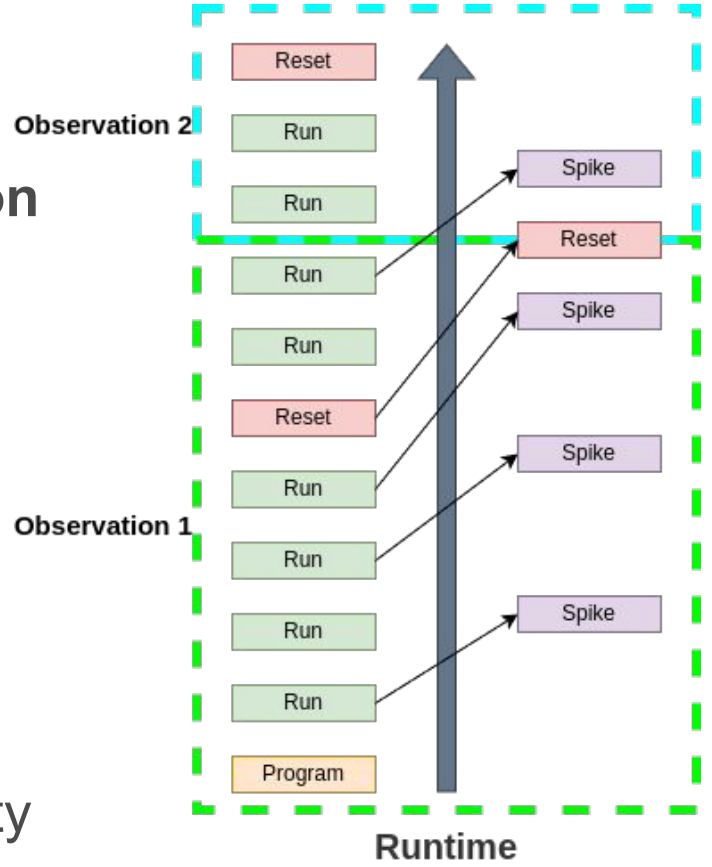
Spike Raster Formation

```
0,00196a00
0,001b4a00
1,000d0a20
1,000d2a20
```

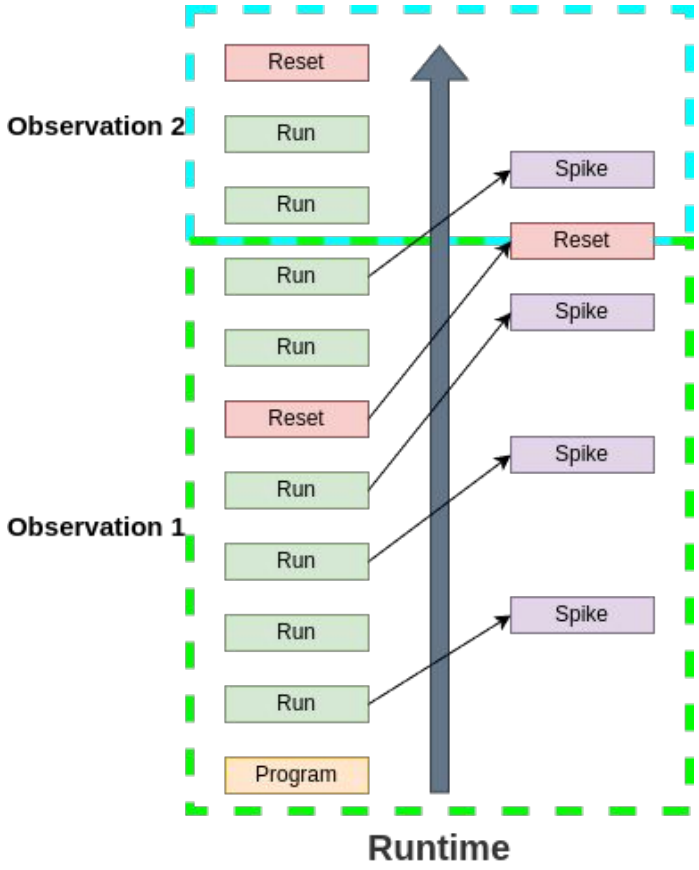
```
Observation 3
20 : 34 : 00000011100111110110
21 : 18 : 00011000011100000000
22 : 4  : 00000000000000000000
23 : 2  : 00000000000000000000

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

- Output is separated into its component fields.
- A string of 0's is initialized for each **output neuron**
  - Left is lowest timestep
  - Right is max timestep
- Based on each packet's time field and neuron number, one of the 0's is flipped to a 1
  - Indicates a **spike** at that timestep and neuron.
- Reset packets delineate between **observations**.
- **Spike rasters** are created for both software and hardware implementations
  - Matching rasters indicate implementation parity



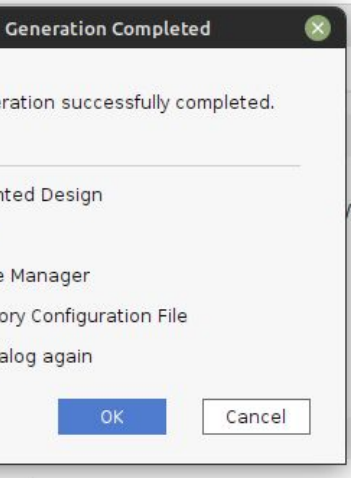
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0
Output Type (3 bits)			RNA Time Step (16 bits)																AER Packet (13 bits)													
																			Core Address (4 bits)				Node Address (4 bits)				Synapse Address (5 bits)				


Observation 2
Observation 1
Reset
Run
Run
Run
Run
Reset
Run
Run
Run
Run
Program
Spike
Reset
Spike
Spike
Spike
Runtime

Results

Old Vivado Project-Based Workflow vs. New Script-Based Workflow

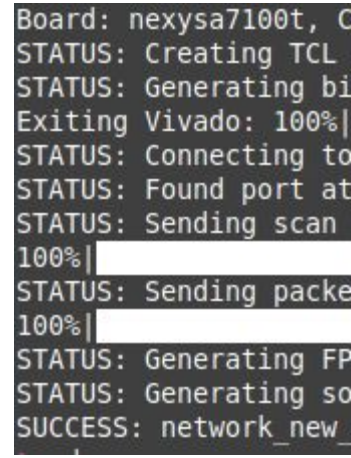
OLD



1. **Initial Setup Steps**
   a. Import core design files into Vivado project
   b. Configure project settings
   c. Create block diagram
2. **Steps Per Network Tested**
   a. Generate and change out network-specific design files
   b. Generate bitstream in Vivado
   c. Run program script to program core
   d. Run neuromorphic framework to produce results to compare against
   e. Run validate script to compare FPGA results with framework results

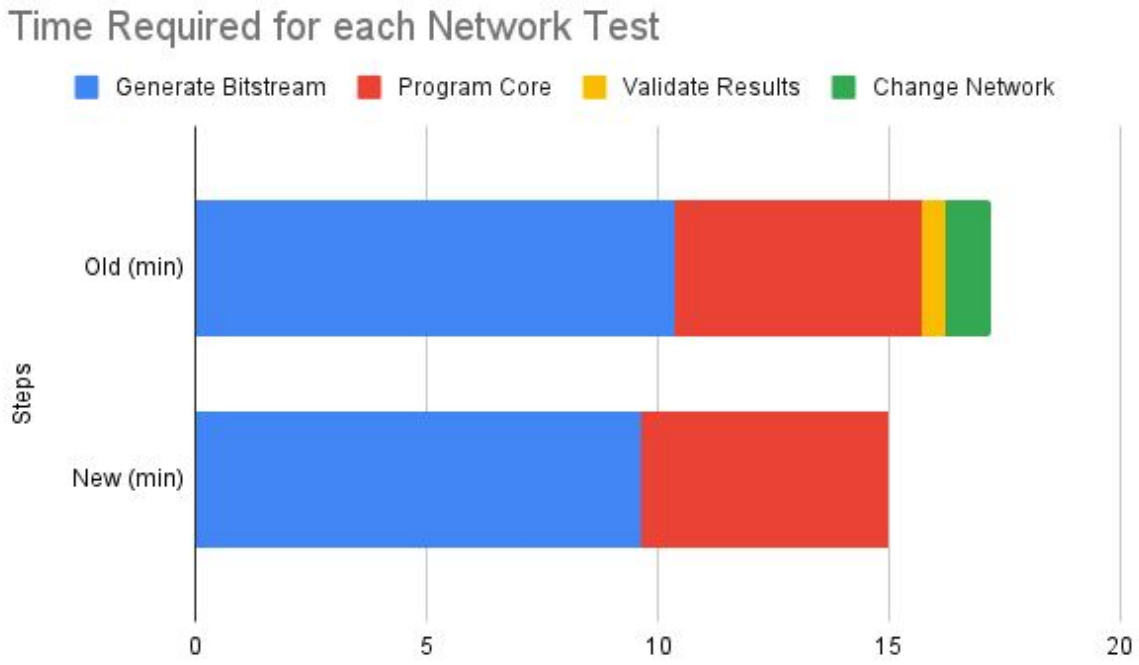
NEW

1. **Initial Setup Steps**
   a. Import core design files into script project
2. **Steps Per Network Tested**
   a. Run pytest command, which runs the following steps automatically
      i. Generate network-specific design files
      ii. Create block diagram
      iii. Generate bitstream
      iv. Program core
      v. Run neuromorphic framework to produce results to compare against
      vi. Compare results



Validation Productivity Improvements

- Approximately a **13% speedup** of time to run test per network.
- Around **2.22 minutes** saved per network
- Time saved can add up significantly when batch testing multiple networks
  - For example, testing **5 networks** results in **11.1 minutes** saved.
- Most important improvement:
  - New method is **fully automated** for batch testing
  - Multiple networks can be tested without human intervention.



Future Work

Future Work

- The FPGA workflow has been successfully integrated into the script-based workflow
  - Currently, behavioral simulation is only available with the old Vivado-based workflow
- Simulation isn't as high priority if the FPGA results match software
  - Simulation primarily used for debugging when the FPGA does not match
- The script-based workflow currently uses stdout and stderr to display progress and errors.
  - Plan on implementing a proper logging framework to improve debugging