A REPORT
ON

RTL DESIGN & VERIFICATION OF A SMALL-SCALE SPIKING NEURAL NETWORK USING SV-UVM METHODOLOGY

BY

Gaurang Brijbhushan Pandey
BITS ID: 2023HT80003

AT

Eximietas Design
Ahmedabad, Gujarat, INDIA

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
APRIL 2025

A REPORT
ON

RTL DESIGN & VERIFICATION OF A SMALL-SCALE SPIKING NEURAL NETWORK USING SV-UVM METHODOLOGY

BY

Gaurang Brijbhushan Pandey
BITS ID: 2023HT80003
M.Tech Microelectronics

Prepared in fulfilment of the
WILP Dissertation
Course No: S2-24_MELZG628T

AT

Eximietas Design
Ahmedabad, Gujarat, INDIA

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
APRIL 2025

Acknowledgement

I would like to express my heartfelt gratitude to my manager, **Mr. Rushang Shah**, for allowing me to pursue this course along with the professional commitments.

I would like to express my heartfelt gratitude to my mentor, **Mr. Kishankumar Kalavadiya** for dedicating his invaluable time and efforts to help me understand complex concepts, thoroughly reviewing my work, and ensuring everything is aligned throughout the course of this project.

I extend my sincere thanks to **Prof. Gangadharaiah S L** for conducting the project reviews and guiding me on the right approach. His critical observations and constructive feedback have helped steer the project in the right direction.

I also wish to extend my warmest thanks to **my family** for constantly motivating me to accept this challenge and supporting me throughout the process. Their encouragement and understanding ensured I could dedicate the time and focus needed to complete this work.

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (RAJASTHAN)
WILP Division

Organization: Eximietas Design **Location: Ahmedabad, Gujarat**
Duration: 4-months **Date of Submission: 24-04-2025**

Title of the Project: RTL DESIGN & VERIFICATION OF A SMALL-SCALE SPIKING NEURAL NETWORK USING SV-UVM METHODOLOGY

ABSTRACT

The project focuses on the design and verification of Spiking Neural Network (SNN) architecture at the Register Transfer Level (RTL) using System Verilog. It involves developing a scalable multi-layer SNN core and creating a robust UVM-based testbench to verify its functionality, scalability, and performance. This work lies at the intersection of neuromorphic computing and digital design verification.

Gaurang Pandey Kishan Kalavadiya

Signature of the Student: **Signature of the Supervisor:**

Name: Pandey Gaurang Brijbhushan **Name: Kishankumar Kalavadiya**
Date: 24-04-25 **Date: 24 -04 -25**
Place: Ahmedabad **Place: Ahmedabad**

CERTIFICATE

This is to certify that the Dissertation entitled **RTL DESIGN & VERIFICATION OF A SMALL-SCALE SPIKING NEURAL NETWORK USING SV-UVM METHODOLOGY** and submitted by **Pandey Gaurang Brijbhushan** having **BITS ID - 2023HT80003** for the fulfilment of the requirements of M. Tech Microelectronics degree of BITS, embodies the Bonafide work done by him/her under my supervision.

Kishan Kalavadiya

Place: Ahmedabad, Gujarat

Signature of the Supervisor

Date: 24-04-2024

Kishankumar Kalavadiya,
Senior Technical Leader
Eximietas Design,
Ahmedabad, Gujarat, India

Contents

List of Tables

List of Figures

List of Abbreviations / Acronyms

Abbreviation	Full Form
SNN	Spiking Neural Network
RTL	Register Transfer Level
SV	SystemVerilog
UVM	Universal Verification Methodology
LIF	Leaky Integrate-and-Fire
DUT	Device Under Test
CSR	Control and Status Register
APB	Advanced Peripheral Bus
RAL	Register Abstraction Layer
TB	Testbench
TLM	Transaction Level Modeling
SPIKE_WINDOW	Time window for spike encoding
MEMBRANE_WIDTH	Width of the neuron's membrane potential
WEIGHT_WIDTH	Width of the synaptic weight
LEAK_WIDTH	Width of the leak factor
THRESHOLD	Neuron firing threshold
RESET_VAL	Neuron reset value
CLK	Clock
RST_N	Active-low Reset
SCB	Scoreboard
FPGA	Field-Programmable Gate Array
STDP	Spike-Timing Dependent Plasticity

1. Broad Area of Work

Digital Design and Verification in VLSI Systems

The project focuses on the design and verification of Spiking Neural Network (SNN) architecture at the Register Transfer Level (RTL) using System Verilog. It involves developing a scalable multi-layer SNN core and creating a robust UVM-based testbench to verify its functionality, scalability, and performance. This work lies at the intersection of neuromorphic computing and digital design verification.

2. Background

Neuromorphic computing is an emerging field inspired by biological neural networks, where neurons communicate through discrete events called spikes. Spiking Neural Networks (SNNs) are a key component of this paradigm and are increasingly utilized in applications such as edge computing, pattern recognition, and IoT due to their energy efficiency and asynchronous processing capabilities.

Designing hardware-efficient SNNs at the RTL level is challenging due to the complexity of spike-based computations and inter-neuron communication. Ensuring the correctness of these designs requires a robust verification methodology. UVM, as an industry-standard verification framework, provides the tools and structure to verify these complex systems rigorously. This project aims to design a multi-layer SNN core and validate its functionality using a UVM-based verification environment

3. Objectives

- To design a multi-layer Spiking Neural Network (SNN) core at the RTL level that supports key neuromorphic features such as:
    - Spike-based computations.
    - Synaptic weight updates.
    - Multi-layer propagation of spikes.
- To develop a comprehensive UVM-based testbench for verifying the SNN core, ensuring:
    - Functional correctness of neuron behavior (spike generation, thresholding, reset).
    - Accurate inter-layer spike propagation.
- To measure the performance and coverage of the SNN design through verification metrics, including latency, correctness, and functional coverage (optional).
- To explore and document the potential of SNNs for real-world applications such as pattern recognition and low-power computation.

4. Scope of Work

- **Design**
    - Develop an RTL model for a configurable multi-layer SNN core.
    - Implement key components like neurons, synapses, and spike routing mechanisms.
    - Integrate layers.
    - Features such as configurable weights, threshold levels, and support for multiple

- **Verification**
    - Build a modular UVM-based testbench for functional and coverage-driven verification.
    - Create sequences and tests for scenarios including edge cases, normal operation, and stress conditions.
    - Design a scoreboard to compare DUT outputs with expected results.
    - Implement functional coverage models for spike generation, potential accumulation, and layer-wise propagation. (This is Optional due to limited time constraint)

- **Analysis:**
    - Verify design correctness using UVM and analyse simulation results.
    - Measure performance metrics like latency & throughput efficiency.
    - Evaluate scalability by testing larger network configurations.

- **Documentation**
    - Provide a detailed project report with design details, verification methodology, results, and conclusions.

5. Spiking Neural Network RTL Design Development Technical Architecture

5.1 Introduction

The technical architecture of a Spiking Neural Network (SNN) implemented in System Verilog RTL. The SNN is designed for pattern recognition, feature extraction and includes a configurable structure with input processing, hidden layers, and output layers. The design implements leaky integrate-and-fire (LIF) neuron models and includes control and status registers accessible via an APB interface.

5.2 System Overview

The spike neural network (SNN) design consists of:

- An input layer that converts pixel values to spike trains
- A hidden layer with configurable neurons
- An output layer that produces classification results
- Control and status registers (CSRs) accessible via APB interface
- Clock division and reset synchronization

The design operates using a spiking neuron model where signals are communicated through binary spikes rather than continuous values, mimicking biological neural networks.

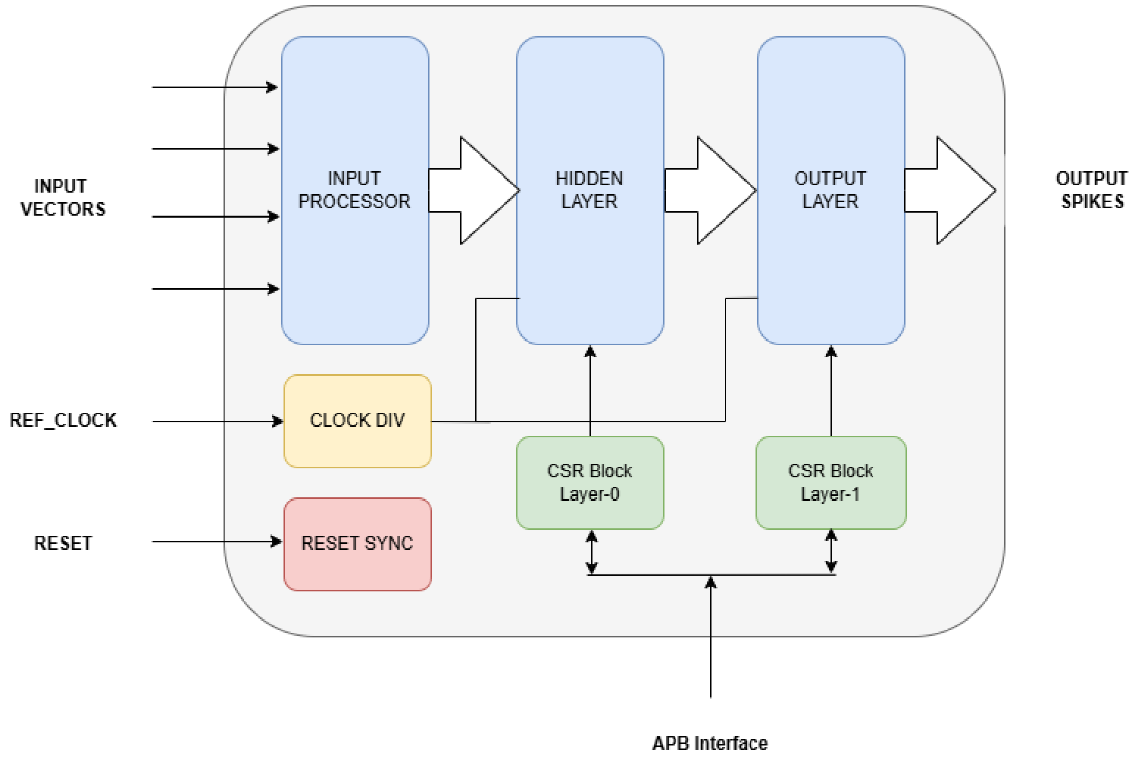
5.3 Spiking Neural Network Module

The **spike_neural_network** module serves as the top-level design, integrating all submodules, including neurons, synapses, input processing, and control logic. It defines the overall connectivity between layers and ensures data flows correctly between different components. This module is responsible for orchestrating the processing of input spikes through hidden and output layers.

Table 1: SNN Top Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Main clock signal
rst_n	Input	1	Active-low reset signal
pixel_input	Input	PIXEL_WIDTH[INPUT_SIZE-1:0]	Pixel values for input image
leak_factor	Input	8	Configurable leak factor for all neurons
digit_spikes	Output	OUTPUT_SIZE	Output spike trains representing classification
psel	Input	1	APB select signal
penable	Input	1	APB enable signal
pwrite	Input	1	APB write signal
paddr	Input	16	APB address bus
pwdata	Input	32	APB write data bus
prdata	Output	32	APB read data bus
pready	Output	1	APB ready signal

5.3.1 Block Diagram



5.3.2 Functionality

- Incorporates two layers of neurons (hidden and output)
- Processes pixel inputs through the input processor to generate input spikes
- Configures network parameters through APB-accessible CSRs
- Uses a divided clock for neuron operation
- Synchronizes resets across clock domains
- Produces output spikes representing for example - digit classifications, pattern, feature

5.4 Input Processor Module

The **input_processor** module is responsible for converting 8-bit values into spike trains. Each input value, ranging from 0 to 255, is processed to generate a corresponding spike pattern based on an encoding mechanism. This module ensures that the input layer of the neural network receives spikes in a format suitable for further processing by synapses and neurons.

Table 2 : SNN Input Layer Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Main clock signal
rst_n	Input	1	Active-low reset signal
clk_cnt	Input	1	Slower counter clock input
pixel_value	Input	PIXEL_WIDTH[INPUT_SIZE-1:0]	Pixel values for input image
spike_out	Output	INPUT_SIZE	Generated spike trains

5.4.1 Functionality

- Implement rate coding where pixel intensity determines spike frequency
- Uses counters to control spike generation timing
- Computes dynamic thresholds based on pixel values
- Generates output spikes when counter values are below thresholds

5.5 Leaky Neuron Module

The lif_neuron module implements the Leaky Integrate-and-Fire (LIF) neuron model. It accumulates incoming weighted spikes, applies a decay factor to simulate biological leakage, and compares the resulting membrane potential with a predefined threshold. If the potential exceeds the threshold, the neuron generates an output spike and resets its state. This module models the core computational unit of the neural network.

Table 3: Neuron Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Clock signal
rst_n	Input	1	Active-low reset signal
input_spike	Input	1	Input spike from synapse
leak_factor	Input	LEAK_WIDTH	Configurable leak factor
threshold	Input	32	Threshold for generating spikes
output_spike	Output	1	Output spike when threshold is reached

5.5.1 Functionality

- Maintains an internal membrane potential
- Integrates incoming spikes by increasing the potential
- Applies a configurable leak to the potential overtime
- Generates an output spike when potential exceeds threshold
- Resets potential after firing

5.6 Synapse Module

The synapse module performs spike-weight multiplication and transmits the weighted spike signals to connected neurons. It stores synaptic weights and applies them to incoming spikes to determine their impact on target neurons. This module plays a key role in learning and adaptation by adjusting weights during operation.

Table 4: Synapse Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Clock signal
rst_n	Input	1	Active-low reset signal
pre_spike	Input	1	Incoming spike train
weight	Input	weight_t	Synaptic weight
threshold	Input	weight_t	Threshold for generating spikes
weighted_spike	Output	1	Output weighted spike train

5.6.1 Functionality

- Accumulates weighted input spikes
- Compares accumulated value to a threshold
- Generates output spikes when threshold is exceeded
- Resets accumulation after a configurable spiking window

5.7 Neuron Layer Module

The neuron_layer module groups multiple lif_neuron & synapses instances to form a functional layer in the network. It handles parallel spike processing and passes the output spikes to the next layer. This module facilitates structured connectivity between layers, ensuring efficient spike propagation.

Table 5: Neuron Layer Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Clock signal
rst_n	Input	1	Active-low reset signal
input_spikes	Input	INPUT_COUNT	Input spikes from previous layer
leak_factor	Input	8	Configurable leak factor
weight_reg	Input	weight_t[INPUT_COUNT-1:0] [NEURON_COUNT-1:0]	Synaptic weights
spike_threshold	Input	weight_t[INPUT_COUNT-1:0] [NEURON_COUNT-1:0]	Thresholds for synapse spike generation
neuron_threshold	Input	16[NEURON_COUNT-1:0]	Thresholds for neuron spike generation
output_spikes	Output	NEURON_COUNT	Output spikes from neurons

5.7.1 Functionality

- Creates a fully connected layer between inputs and neurons
- Instantiates synapses for each input-neuron connection
- Aggregates weighted spikes for each neuron
- Instantiates LIF neurons to process aggregated inputs
- Produces output spikes from each neuron

5.8 CSR Blocks

The snn_csr_apb module implements an APB interface for accessing control and status registers (CSRs) of the neural network. It enables external configuration of synaptic weights and neuron thresholds by interfacing with the system bus. This module allows dynamic updates to network parameters without modifying the hardware design.

Table 6 : CSR Block Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Clock signal
rst_n	Input	1	Active-low reset signal
psel	Input	1	APB select signal
penable	Input	1	APB enable signal
pwrite	Input	1	APB write signal
paddr	Input	16	APB address bus
pwdata	Input	32	APB write data bus
prdata	Output	32	APB read data bus
pready	Output	1	APB ready signal
weight_reg	Output	32[((INPUT_SIZE * OUTPUT_SIZE)) - 1:0]	Weight registers
spike_threshold	Output	32[((INPUT_SIZE * OUTPUT_SIZE)) - 1:0]	Spike threshold registers
neuron_threshold	Output	32[OUTPUT_SIZE-1:0]	Neuron threshold registers
cntrl_status_csr	Output	32	Control and status register

5.8.1 Functionality

- Handles APB protocol transactions
- Provides memory-mapped access to network parameters
- Supports configuration of weights, thresholds, and control registers
- Maintains register state between transactions

5.9 Clock Divider Module

The clock_divider module generates a slower clock signal from the system clock to synchronize different operations in the neural network. Given that spike processing may require timing control different from the main clock, this module ensures that neuron updates and spike propagations happen at a controlled rate.

Table 7: Clock Divider Interface Signal Details

Signal	Direction	Width	Description
clk_in	Input	1	Input clock
rst_n	Input	1	Active-low reset signal
clk_out	Output	1	Divided clock output

5.9.1 Functionality

- Divides the input clock by a parameterizable factor
- Generates a 50% duty cycle output clock
- Provides synchronized reset capabilities

5.10 Reset Synchronization Module

The reset_synchronizer module ensures a stable and glitch-free reset signal across different clock domains. It synchronizes the reset signal with the active clock, preventing metastability issues that could arise due to asynchronous resets. This module is crucial for maintaining reliability in the design.

Table 8: Reset Synchronizer Interface Signal Details

Signal	Direction	Width	Description
clk	Input	1	Destination clock domain
enable	Input	1	Output enable
rst_n_async	Input	1	Asynchronous active-low reset
rst_n_sync	Output	1	Synchronized active-low reset

5.10.1 Functionality

- Implements a two-flip-flop synchronizer for reset signals
- Prevents metastability when crossing clock domains
- Provides conditional enabling of the synchronized reset

5.11 Neuron Design Parameter Package

Contains parameters and type definitions for LIF neuron implementation.

Table 9: Neuron Parameters

Parameter	Value	Description
MEMBRANE_WIDTH	32	Membrane potential width in bits
WEIGHT_WIDTH	32	Synaptic weight width in bits
LEAK_WIDTH	8	Leak factor width in bits
THRESHOLD	16’h1000	Default firing threshold
RESET_VAL	16’h0000	Default reset value after spike

Table 10 : Neuron Type Definitions

Type	Definition	Description
membrane_t	logic [MEMBRANE_WIDTH-1:0]	Membrane potential type
weight_t	logic [WEIGHT_WIDTH-1:0]	Synaptic weight type
leak_t	logic [LEAK_WIDTH-1:0]	Leak factor type

5.12 SNN Design Parameter Package

Contains parameters for the overall neural network configuration.

Table 11 : SNN Design parameters

Parameter	Value	Description
CLK_PERIOD	10	Clock period in ns (100 MHz)
CLOCK_DIVIDER_VAL	8	Clock divider value
INPUT_SIZE	64	Number of input neurons
HIDDEN_SIZE	32	Number of neurons in hidden layer
OUTPUT_SIZE	16	Number of output neurons
PIXEL_WIDTH	8	Width of pixel values in bits
SPIKE_WINDOW	16	Time window for rate coding
LAYER_WEIGHT_BASE_ADDR_U0	16’h0000	Base address for layer 0 weights
LAYER_SPIKE_THRESH_BASE_ADDR_U0	16’h2000	Base address for layer 0 spike thresholds
LAYER_NEURON_THRESH_BASE_ADDR_U0	16’h4000	Base address for layer 0 neuron thresholds
CONTROL_STATUS_BASE_ADDR_U0	16’h6000	Base address for layer 0 control/status
LAYER_WEIGHT_BASE_ADDR_U1	16’h8000	Base address for layer 1 weights
LAYER_SPIKE_THRESH_BASE_ADDR_U1	16’hB000	Base address for layer 1 spike thresholds
LAYER_NEURON_THRESH_BASE_ADDR_U1	16’hC000	Base address for layer 1 neuron thresholds
CONTROL_STATUS_BASE_ADDR_U1	16’hE000	Base address for layer 1 control/status

6. Spiking Neural Network (SNN) Verification Testbench Architecture

6.1 Introduction

This document provides a comprehensive overview of the verification testbench architecture for a Spiking Neural Network (SNN) design. The verification environment is developed using the Universal Verification Methodology (UVM) in System Verilog. The purpose of this environment is to functionally verify the correct implementation of the SNN's behavior, connectivity, and configurability using stimulus patterns and score boarding mechanisms.

The SNN design under test (DUT) mimics biological neurons and processes 8x8 pixel image inputs. It includes an input layer, hidden layer, and output layer composed of Leaky Integrate-and-Fire (LIF) neurons. The design integrates custom CSRs for configuring weights, thresholds, and other neural parameters.

6.2 SNN Top-Level Testbench Overview

The SNN testbench is structured hierarchically under the top module SNN_TOP, which encapsulates the complete verification environment for a Spiking Neural Network design. The test bench comprises the SNN_TEST module, containing the main environment module SNN_ENV.

6.2.1 Components of the Testbench

- **SNN_ENV**: The main environment which houses the verification components including agents, scoreboard, register model, and predictors/adapters.
- **SNN_AGENT**: Manages the SNN-specific protocol transactions. It includes:
    - SNN_SEQR: Sequence generator for SNN stimulus.
    - SNN_DRIVER: Drives transactions to the DUT.
    - SNN_MONITOR: Captures and reports DUT behaviour.
    - Uses a VIRTUAL SNN_INTERFACE to connect with the DUT.
- **APB_AGENT:** Manages APB bus protocol interactions with the DUT. It includes:
    - APB_SEQR: Sequence generator for APB transactions.
    - APB_DRIVER: Drives APB signals to the DUT.
    - APB_MONITOR: Monitors APB activity.
    - Uses a VIRTUAL APB_INTERFACE for communication.
- **SNN SCOREBOARD:** Collects and checks DUT outputs against expected values.
- **SNN RAL MODEL:** Represents the register abstraction layer, used for register-level verification. It connects to:
    - APB REG_ADAPTER: Translates APB transactions to RAL transactions.
    - APB REG_PREDICTOR: Predicts expected register states based on observed transactions.
- **Interfaces:** The testbench interacts with the DUT through:
    - CLK_RESET_INTERFACE
    - SNN_INTERFACE
    - APB_INTERFACE

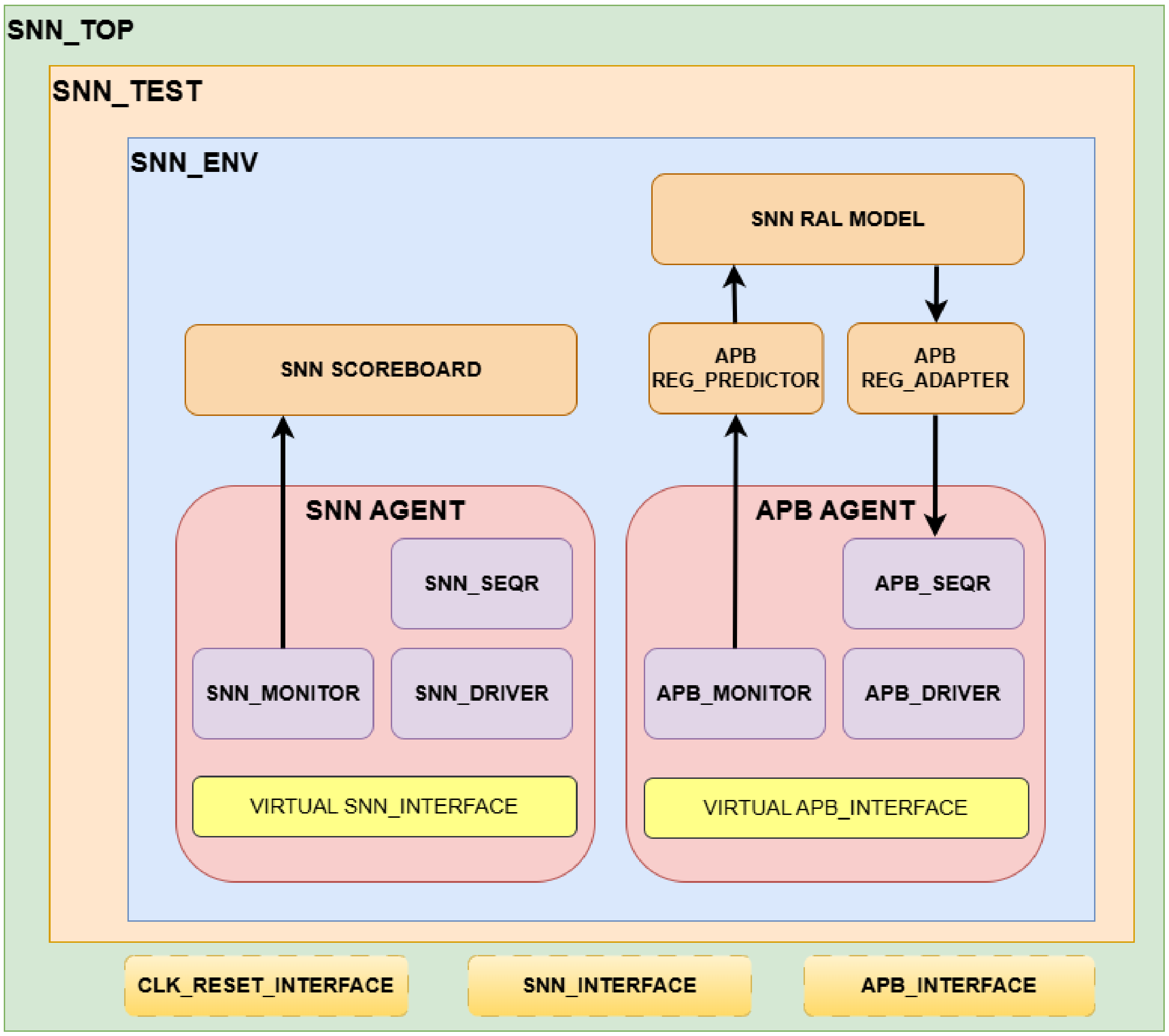


Figure 1: SNN TB Top Level Block Diagram

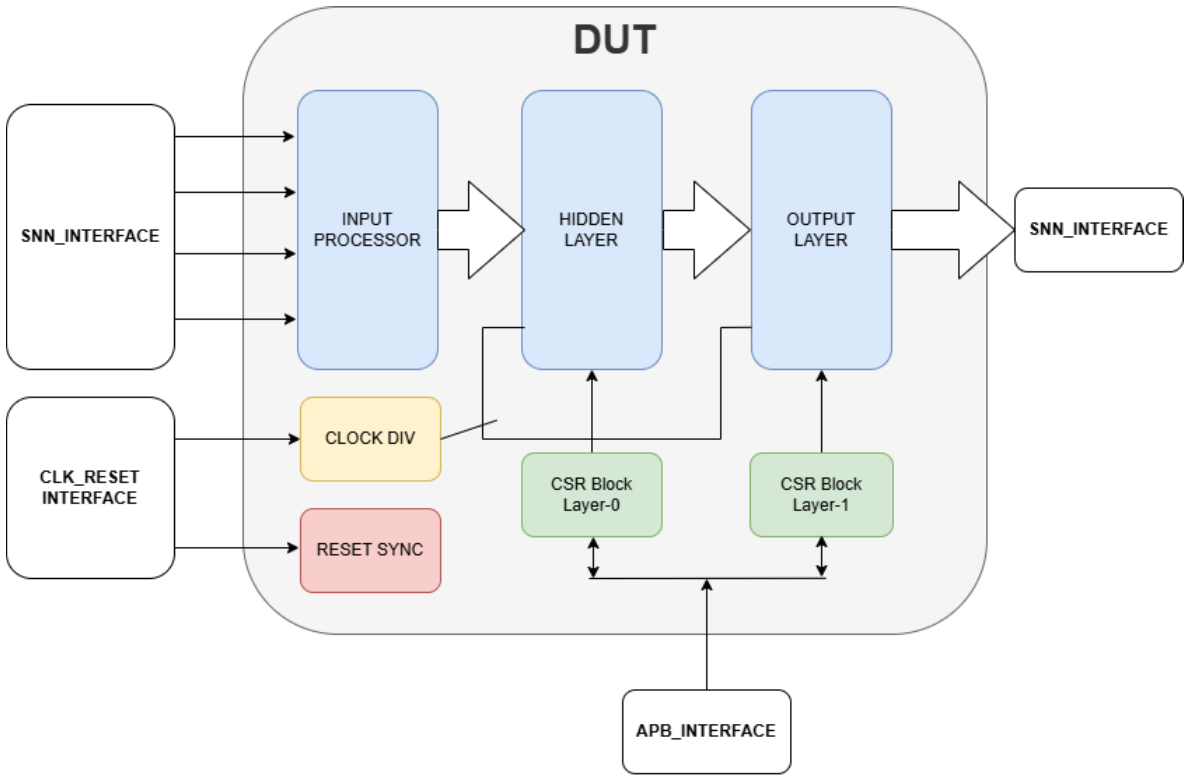


Figure 2: Interfaces connection with DUT

6.3 Interface Layer

The interface layer provides a structured connection between the DUT and verification components:

The following are the SV interfaces present in the current SNN testbench.

1. Clock & Reset Interface
    - Contains clock and reset signal
    - Generate periodic clock & initial reset to DUT
2. SNN Interface
    - Encapsulates input pixels, spike outputs, and config inputs.
    - Used by SNN Driver and Monitor
3. APB Interface
    - Encapsulates APB protocol signals
    - Used by APB Driver and Monitor for CSR write-read

6.4 APB Agent

The APB agent follows standard UVM methodology and provides a complete verification infrastructure for APB protocol-based transactions.

The APB agent serves as a protocol-specific verification component that drives APB-based interface for CSR accesses within the SNN design under test.

6.4.1 Component Structure

The APB agent consists of the following key components:

- **APB Agent**: Top-level component that manages driver, sequencer, and monitor instances.
- **APB Driver**: Converts sequence items to pin-level activity on the APB interface.
- **APB Monitor**: Observes APB bus activity and converts to transaction objects.
- **APB Sequencer**: Manages and distributes sequence items to the driver.
- **APB Interface**: Defines protocol-specific signals and clocking blocks.
- **APB Register Adapter**: Converts register operations to APB transactions.
- **APB Register Predictor**: Updates register model based on observed transactions.

6.4.2 Configurability

The APB agent can operate in either active or passive mode:

- **Active Mode:** Full agent with driver, sequencer, and monitor
- **Passive Mode:** Monitor-only configuration for analysis purposes

6.4.3 APB Transaction Model

The transaction model defines the data structure for APB operations:

Key Fields

- Address (paddr): 32-bit APB address
- Write control (pwrite): Determines read or write operation
- Write data (pwdata): 32-bit data for write operations
- Read data (prdata): 32-bit data captured during read operations
- Ready signal (pready): Status from slave device
- Operation type: Enumerated type (READ/WRITE)

Constraints

- Address alignment constraints ensure proper memory access
- Operation consistency constraints link transaction type to control signals

6.4.4 Integration with SNN Environment

- The APB agent is integrated within the SNN verification environment.
- The APB agent connects to a register model that provides:
- Address map for register access
- Field definitions and access policies
- Reset values and default configurations

6.4.5 Connection Details

Key connections within the environment include:

1. APB agent's monitor to register predictor
2. Register model to APB sequencer via adapter
3. APB monitor to scoreboard for transaction checking
4. Virtual sequencer to APB and SNN sequencers

6.5 SNN Agent

The SNN agent follows standard UVM methodology and provides a comprehensive verification infrastructure for SNN-based transactions.

6.5.1 Component Structure

The SNN agent consists of the following key components:

- **SNN Agent**: Top-level component that orchestrates driver, sequencer, and monitor instances
- **SNN Driver**: Drives pixel data and control signals to the SNN interface
- **SNN Monitor**: Observes activity on the interface and captures spike outputs
- **SNN Sequencer**: Manages and distributes SNN transaction items to the driver
- **SNN Interface**: Defines signal connections and clocking blocks

6.5.2 Configurability

The SNN agent supports configurability through:

- Active/passive mode selection (UVM_ACTIVE or UVM_PASSIVE)
- Dynamic creation of components based on configuration

6.5.3 SNN Transaction Model

The transaction model encapsulates the data structure for SNN operations:

Key Fields

- **Pixel Input Array**: Dynamic array of 8-bit pixel values representing input data
- **Leak Factor**: 8-bit control value influencing neuron potential decay
- **Spike Window**: Timing parameter controlling stimulus application duration
- **Digit Spikes**: Output array representing neural network classification results

Constraints

- Array size constraints ensure consistency with network package parameters
- Pixel value constraints enforce valid 8-bit ranges (0-255)
- Leak factor constraints control neuron behaviour
- Spike window constraints set appropriate duration for network evaluation

6.5.4 SNN Driver Implementation

Functionality

The SNN driver is responsible for:

- Resetting the device under test (DUT)
- Driving pixel input values to the SNN interface
- Setting appropriate control signals (pixel_valid, leak_factor)
- Maintaining timing relationships based on spike window requirements

Operation Sequence

The driver follows a defined sequence:

1. Reset the DUT through interface signals
2. Receive transaction items from the sequencer
3. Drive pixel input array values to the interface
4. Assert pixel_valid for the duration of spike_window cycles
5. De-assert pixel_valid to complete the transaction

Reset Operation

The reset operation initializes the DUT by:

- Asserting the active-low reset signal (rst_n)
- Clearing all pixel inputs to zero
- Setting default leak factor value
- Maintaining reset for five clock cycles
- De-asserting reset to enable normal operation

6.5.5 SNN Monitor Implementation

Functionality

The SNN monitor performs:

- Continuous observation of SNN interface signals
- Detection and capture of neuron spike events
- Creation and population of transaction objects
- Transmission of observed transactions to the scoreboard

Monitoring Process

The monitor implements a state-based monitoring approach:

1. Detect valid pixel inputs (pixel_valid assertion)
2. Capture spike outputs throughout the active period
3. Accumulate spike information until pixel_valid is de-asserted
4. Create transaction objects containing accumulated spike data
5. Transmit completed transactions through analysis port

6.5.6 Integration with SNN Environment

Environment Architecture

The SNN agent is integrated within the comprehensive SNN verification environment alongside the APB agent

Connection Details

Key connections within the environment include:

1. SNN agent's monitor to scoreboard for output verification
2. SNN sequencer to virtual sequencer for coordinated test sequences
3. SNN driver to SNN interface for stimuli generation

Virtual Sequencer Integration

The SNN sequencer is connected to the virtual sequencer to enable:

- Coordinated test sequences between APB and SNN operations
- Synchronized configuration and stimulus application
- Cross-domain verification scenarios

Scoreboard Integration

The SNN agent's monitor provides transaction information to the scoreboard for:

- Output spike pattern verification
- Neural network classification accuracy assessment
- Functional coverage collection

6.6 SNN Register Model (RAL model)

The SNN testbench includes a comprehensive register model built using UVM's Register Abstraction Layer (RAL). This model enables controlled and observable access to DUT configuration and status registers through both front door (via APB) and backdoor paths. It ensures test scenarios can manipulate the design registers in a protocol-independent and reusable manner.

6.6.1 Register Model Construction

- The register model is defined in snn_reg_pkg.sv, using UVM macros and base classes:
- Fields: Each field corresponds to configurable elements like:
- Synaptic weights (per input node to hidden neuron)
- Hidden-to-output layer weights
- Neuron-specific thresholds

6.6.2 Register Blocks

- snn_reg_block: Top-level block instantiating all register sets.
- Layered weight registers are grouped hierarchically by destination neuron index.
- Address Mapping: Registers are aligned with the APB memory map.

6.6.3 Integration in Testbench

- The register model is instantiated and configured in snn_env.sv:
- uvm_reg_block snn_reg_blk; is created in the environment class.
- It is connected to the apb_reg_adapter and the register predictor.
- The APB monitor is used to track register access and predict internal model state.

6.6.4 Access Methods

- Front door Access - All tests and sequences use the register model APIs such as write() and read() to configure registers via the APB agent.
- Mirror and Predict - The register values are mirrored and compared with the DUT state for consistency checking.

6.6.5 RAL-Integrated Sequences

- snn_reg_bitbash_sequence: Flips all bits and checks correct reflection.
- snn_reg_rand_sequence: Random register writes to test robustness.
- snn_reg_layer_sequence: Writes a full layer’s weight registers in a single transaction series.

6.6.6 Benefits of RAL

- Register model abstracts low-level details and simplifies test writing.
- Enables reusability across tests and makes coverage easier to collect and analyze.
- Reduces chances of testbench-DUT misalignment through consistent predictive modeling.

6.7 SNN Scoreboard

6.7.1 Scoreboard Overview

The SNN scoreboard (snn_scoreboard) provides comprehensive result checking and validation for the SNN design. It compares actual SNN outputs against expected results based on configured parameters.

6.7.2 Key Features

The scoreboard implements the following capabilities:

- Analysis ports for SNN and APB transactions
- Statistical tracking of matches and mismatches
- Detailed comparison reporting

6.7.3 Transaction Processing

- **APB Transactions:** The write_apb method processes APB transactions
  - Capturing register configuration operations
  - Updating internal configuration flags if any
- **SNN Transactions:** The write_actual method handles SNN transactions by:
  - Incrementing transaction counters
  - Generate expected packet & Invoking comparison logic

6.7.4 Comparison Logic

The scoreboard implements a deterministic comparison strategy:

1. Creates expected spike pattern based on neuron_num parameter
2. Compares each output spike against expected values
3. Records match or mismatch status
4. Generates detailed comparison report
5. Updates statistical counters

6.8 SNN Environment

6.8.1 SNN Environment Architecture Overview

The SNN environment (snn_env) integrates multiple verification components into a coherent testbench structure. It implements the standard UVM environment paradigm while incorporating specialized components for SNN and APB protocol verification.

6.8.2 Component Hierarchy

The environment contains the following key components:

- **SNN Agent**: Manages SNN-specific stimulus generation and response monitoring
- **APB Agent**: Handles APB bus transactions for register access
- **Virtual Sequencer**: Coordinates test sequences across multiple interfaces
- **Register Model**: Provides abstraction for DUT register access
- **Register Adapter**: Converts register operations to APB transactions
- **Register Predictor**: Updates register model based on observed bus activity
- **Scoreboard**: Validates SNN functionality and results

6.8.3 Configuration Parameters

The environment supports configuration through:

- neuron_num: Specifies the target neuron for expected classification
- scb_en: Controls scoreboard instantiation and connection

6.8.4 Environment Implementation Details

Build Phase: During the build phase, the environment:

1. Creates agent instances (SNN and APB)
2. Instantiates the virtual sequencer
3. Conditionally creates the scoreboard based on scb_en
4. Creates register adapter and predictor components
5. Builds and configures the register model
6. Makes the register model globally accessible via configuration database

Connect Phase: The connect phase establishes the following connections:

1. Virtual sequencer to individual agent sequencers
2. Register model to APB sequencer through the adapter
3. APB monitor to register predictor
4. Agent monitors to scoreboard analysis ports (when enabled)

6.8.5 Register Model Integration

The register model is:

1. Created and built during the build phase
2. Reset to initialize default values
3. Locked to prevent structural modifications
4. Configured for automatic read checking
5. Connected to the APB infrastructure through the adapter and predictor

6.9 SNN Testbench Top Module

The **snn_tb_top** module is the top-level testbench module for the Spiking Neural Network (SNN) design.

6.9.1 Module Instantiations

Clock and Reset Interface:

- Instantiates clk_rst_if as clk_rst_vif which generates:
    - 100 MHz clock signal (toggling every 5ns)
    - Reset signal (rst_n) that starts at 0 and transitions to 1 after 100ns
- SNN Interface:
    - Instantiates snn_if as snn_vif for SNN-specific signals
- APB Interface:
    - Instantiates apb_if as apb_vif with clock and reset connections for register access

DUT (Device Under Test):

- Instantiates the spike_neural_network module as dut with the following connections:
    - Clock and reset signals
    - Neural network signals (pixel_input, leak_factor, digit_spikes)
    - APB interface signals (paddr, psel, penable, pwrite, pwdata, prdata, pready)

Signal Interconnections

- Clock and Reset:
    - clk is connected to clk_rst_vif.clk
    - rst_n is connected to clk_rst_vif.rst_n
- SNN Interface Connections:
    - snn_vif.clk is connected to dut.clk_cnt
    - snn_vif.rst_n is connected to dut.u_reset_sync.rst_n_sync
    - SNN signals (pixel_input, leak_factor, digit_spikes) are connected directly to the DUT
- APB Interface Connections:
    - All APB signals are directly connected between the interface and the DUT

6.9.2 Testbench Function

The testbench performs these primary functions:

Environment Setup:

- Registers the virtual interfaces with the UVM configuration database:
    - snn_vif registered as "vif" for SNN-specific components
    - apb_vif registered as "vif" for APB components
    - clk_rst_vif registered as "vif" for clock/reset components

Test Execution:

- Launches the UVM test by calling run_test("snn_base_test")
- This initiates the UVM testing infrastructure to run the specified test

The testbench serves as the connection point between the UVM verification environment and the actual SNN design, providing the necessary interfaces and stimulus to verify the functionality of the Spiking Neural Network implementation.

6.10 SNN Sequences

The following are the list of the sequences, and which is used to generate various flavors of stimulus.

Sequence File	**Purpose / Description**
snn_base_sequence.sv	Base class for all sequences. Contains instances for RAL model, SNN interface, and Clock & Reset interface.
snn_init_sequence.sv	Performs a sanity test on the DUT. Writes to cntrl_status_csr to propagate resets to all neuron layers.
snn_reg_rand_sequence.sv	Performs random writes followed by reads on every register. Ensures basic RAL model integration and data path integrity.
snn_reg_bitbash_sequence.sv	Applies bit-level toggles on all SNN registers. Helps confirm register correctness and bit-level implementation compliance.
snn_reg_layer_sequence.sv	Programs weights and threshold for **one** output neuron. Other output neurons are set to zero. Helps validate output behaviour for selected neurons.
snn_random_sequence.sv	This sequence generates random values at all the 64-inputs of the SNN DUT. The values ranges from 0 to 255.
snn_digit_sequence.sv	This sequence generates digit-like (0-9) patterns to the input of the DUT.
snn_gradient_sequence.sv	This sequence generates gradually increasing pixels.
snn_checkered_sequence.sv	This sequence generates alternating high-low pixels.
snn_stripe_sequence.sv	This sequence generates horizontal/vertical stripe inputs.
snn_radial_sequence.sv	This sequence generates radial gradient input test.

6.11 SNN Tests

The following are the list of the test, and which is used to generate test scenarios by calling one or more of the SNN sequences

Test	**Description**
snn_base_test.sv	Base class for all sequences. Contains instances for RAL model, SNN interface, and Clock & Reset interface.
snn_sanity_test.sv	Performs a sanity test on the DUT. Writes to cntrl_status_csr to propagate resets to all neuron layers.
snn_reg_rand_wr_rd_test.sv	This test calls the snn_reg_rand_sequence Performs randomized writes and reads to all CSR.
snn_reg_bitbash_test.sv	This test calls the snn_reg_bitbash_sequence.
snn_random_sequence_test.sv	This test generates random stimulus to SNN 64-bit input array Programs the weights & thresholds of each neuron with random values Drives random input values and expect random spikes on all DUT outputs
snn_reset_test.sv	Ensures correct behavior post-reset. Generates reset in-between on-going input patterns
snn_digit_sequence_test.sv	Classifies digits from input patterns.
snn_gradient_sequence_test.sv	Gradient value test.
snn_checkered_sequence_test.sv	Alternating pattern test.
snn_stripe_sequence_test.sv	Stripe-specific pattern test.
snn_radial_sequence_test.sv	Test with radial pattern.

6.12 Test Simulation Flow

The simulation of the Spiking Neural Network (SNN) is carried out through multiple systematic UVM phases, ensuring correctness, configurability, and output validation. Each phase has specific responsibilities that contribute to a comprehensive verification flow.

Build Phase

- All UVM components (agents, environment, scoreboard, register model) are constructed.
- Register model (snn_reg_block) is created and mapped to APB addresses.
- Configuration objects and virtual sequencer handles are propagated to sub-components.

Connect Phase

- TLM connections are established between sequencers and virtual sequencer.
- Scoreboard receives analysis ports from both SNN and APB monitors.
- The register model is bound to the adapter and predictor.
- APB monitor is connected to the register predictor for state tracking.

Elaboration Phase

- All configuration and factory overrides are finalized.
- Optional UVM report configurations (verbosity, severity actions) are completed.

Start of Simulation Phase

- Initial diagnostic messages and testbench settings are displayed.
- Waveform dump is initiated if enabled via waves.tcl.

Run Phase

- This is the main simulation execution phase and is time-consuming.
- Reset DUT - The DUT is reset through the interface signals (rst_n).
- Optionally verified using the snn_reset_test.
- Register Programming - Using apb_agent and the register model, CSRs are initialized with: Synaptic weights, Neuron thresholds & Leak factor
- Virtual sequences such as snn_init_sequence handle register setup.
- Stimulus Application - Pixel values are streamed through the snn_driver.
- Output Collection- The snn_monitor collects output spike signals.
- Transactions are sent to the scoreboard for comparison.

Check Phase

- The scoreboard compares actual spikes with expected spikes.
- Failures are reported through the UVM report server.

Report Phase

- A final simulation summary is printed.
- Pass/fail criteria, assertion statistics, and error counts are listed.
- Coverage summary (if enabled) is displayed.
- A waveform file and log are saved for post-simulation analysis.

6.13 Coverage Strategy

- Code coverage enabled via coverage.ccf config.
- Running full random testcases and CSR programming to achieve 100% code coverage
- Developed specific scenarios such as reset to achieve the higher toggle coverage
- There are codes which is by default tied to specific values and will not toggle
- Exclusions are added to the .vRefine file (Cadence Flow)
- Functional coverage optional for future.

6.14 Compilation and Simulation Scripts

6.14.1 File Compilation Order

The files.f file lists all SystemVerilog and UVM components in the correct order of compilation. This ensures dependencies are resolved correctly and all components are recognized during simulation. It includes the following types of files:

- Interface definitions (clk_rst_if.sv, snn_if.sv, apb_if.sv)
- Package files (snn_tb_pkg.sv, apb_pkg.sv, etc.)
- Component files (agents, monitors, drivers, scoreboards, etc.)
- Environment and testbench top (snn_env.sv, snn_tb_top.sv)
- Virtual sequences and tests
- Register model files

Maintaining the correct order in files.f is essential for successful elaboration by the simulator.

6.14.2 Simulation Script

The **run_cmd.sh** script is a unified driver to execute simulation runs. It supports test selection, conditional coverage collection, and user-defined runtime variables. It accepts up to four arguments:

The following is the command to run any of the developed uvm_test :

```
./run.sh <test_name> <enable_coverage: 1|0> <neuron_num> <scb_enable>
```

6.14.3 Script Workflow

- **+UVM_TESTNAME**: Specifies the UVM test to run (default: base_test).
- **+COV_EN:** If set to 1, includes the coverage.ccf file for functional and code coverage.
- **+NEURON_NUM** & **+SCB_EN**: Parameters to configure test-specific DUT behavior and scoreboard usage.
- **Working Directory Setup**:
    - Create a **run** folder if absent.
    - Creates a unique sub-directory for the test (run/<test_name>).
- **Simulation Execution**: Uses Cadence xrun with the following options:

- **-uvmhome** to set UVM library path
    - **-f** to include files.f
    - **-access +rwc** for runtime access to signals and variables
    - **-timescale 1ns/1ps** to set timing resolution
    - **-input** to source waveform dump (waves.tcl)
    - **+UVM_TESTNAME, +NEURON_NUM, +SCB_EN** for runtime macros
    - Conditional appending of **-covfile** when coverage is enabled

This script ensures reproducibility, organized logs, and scalable simulation handling.

6.15 Simulation Results and Code Coverage Statistics

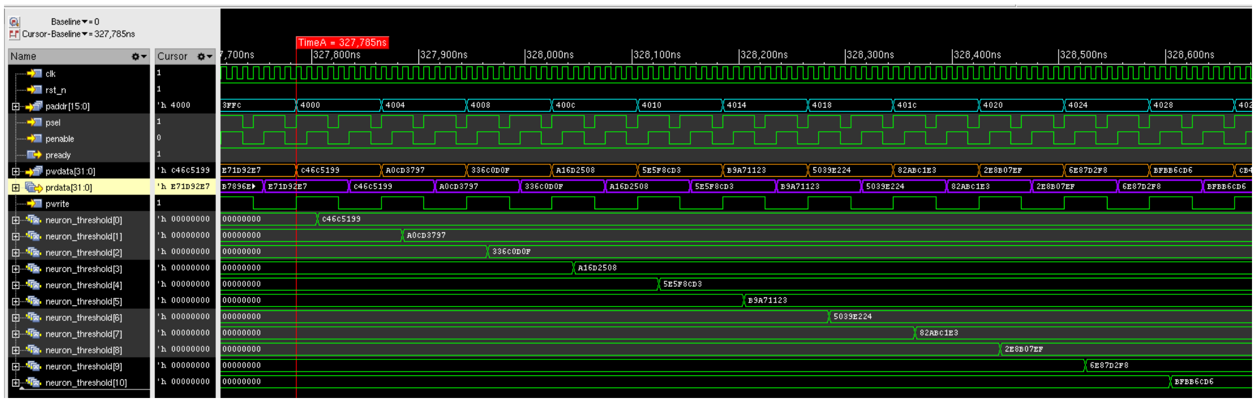


Figure 3: APB-CSR Write Read Operation

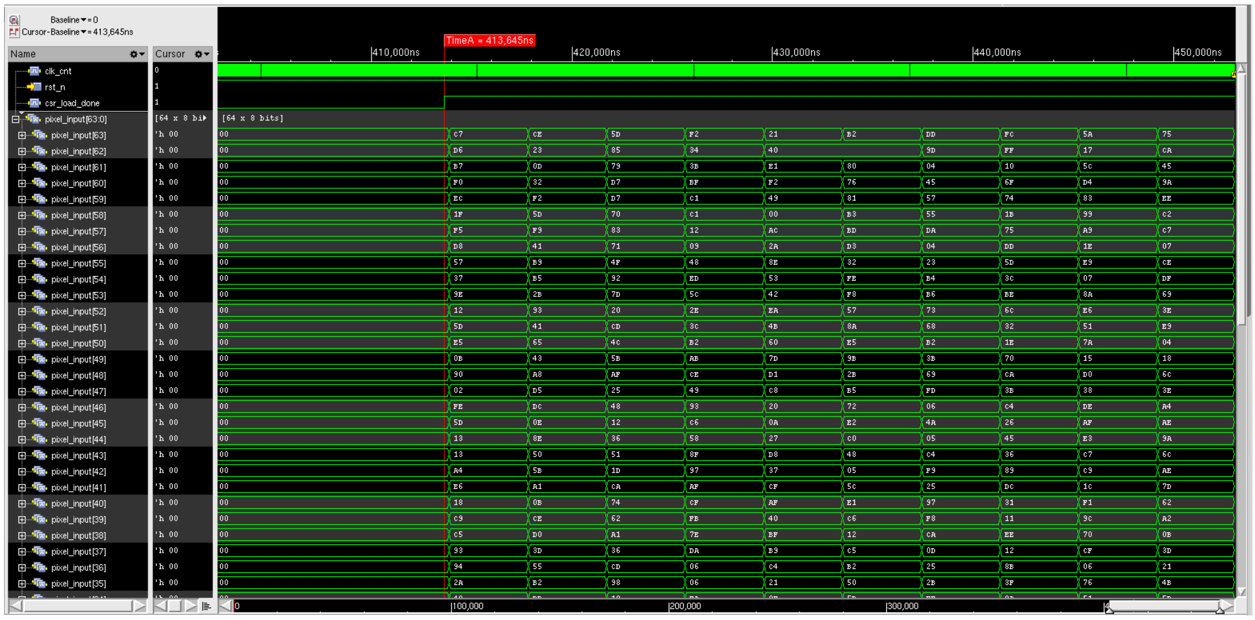


Figure 4: Driving 64-bit input array (pixel value) after CSR Load done.

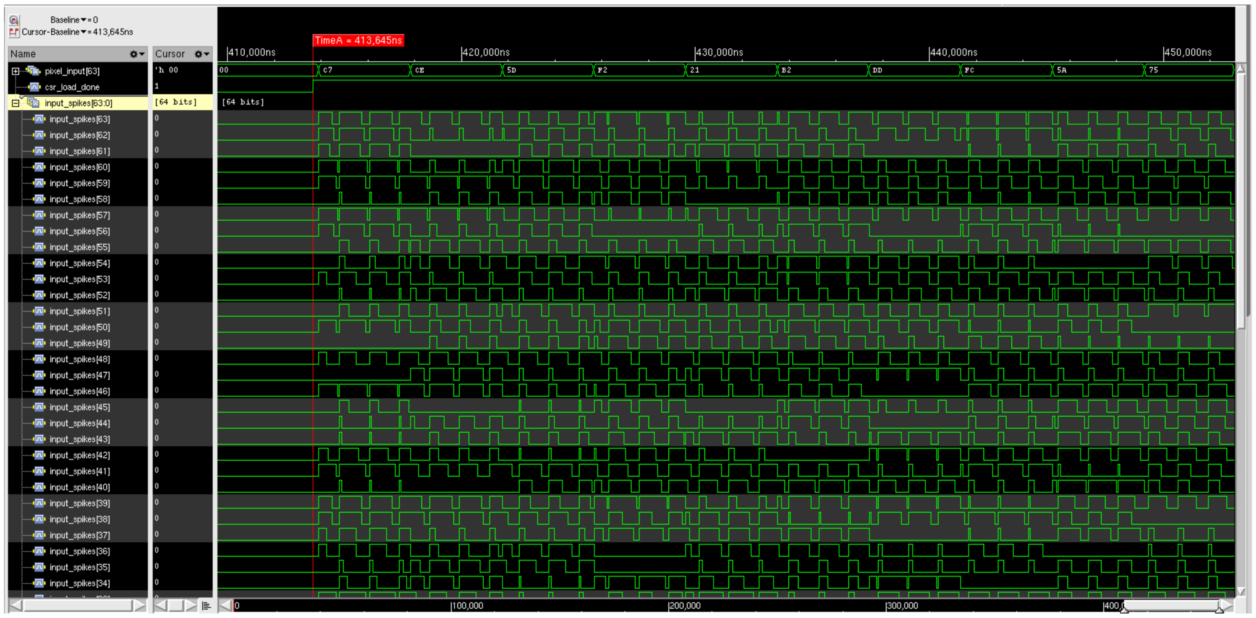


Figure 5: Pixel values getting converted to spike-trains with varying duty-cycle

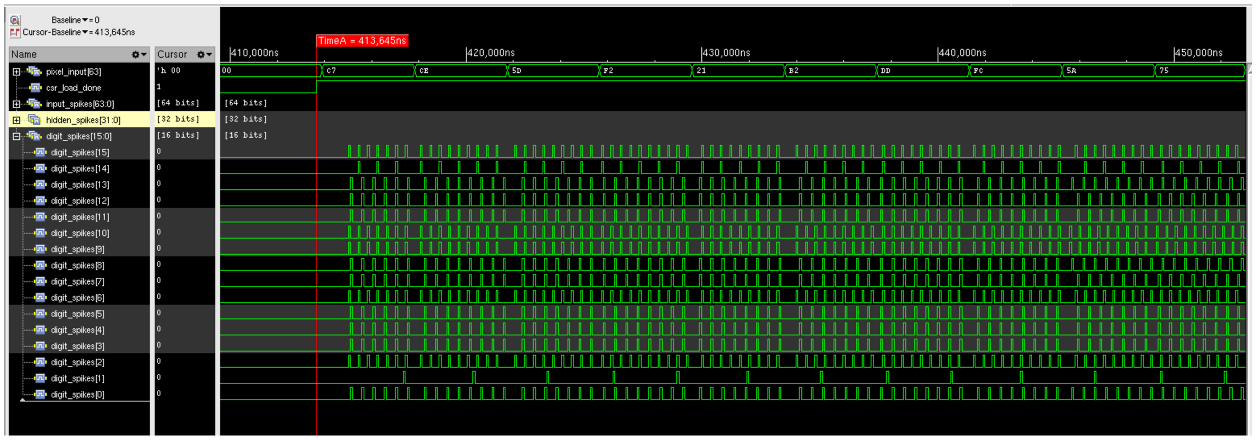
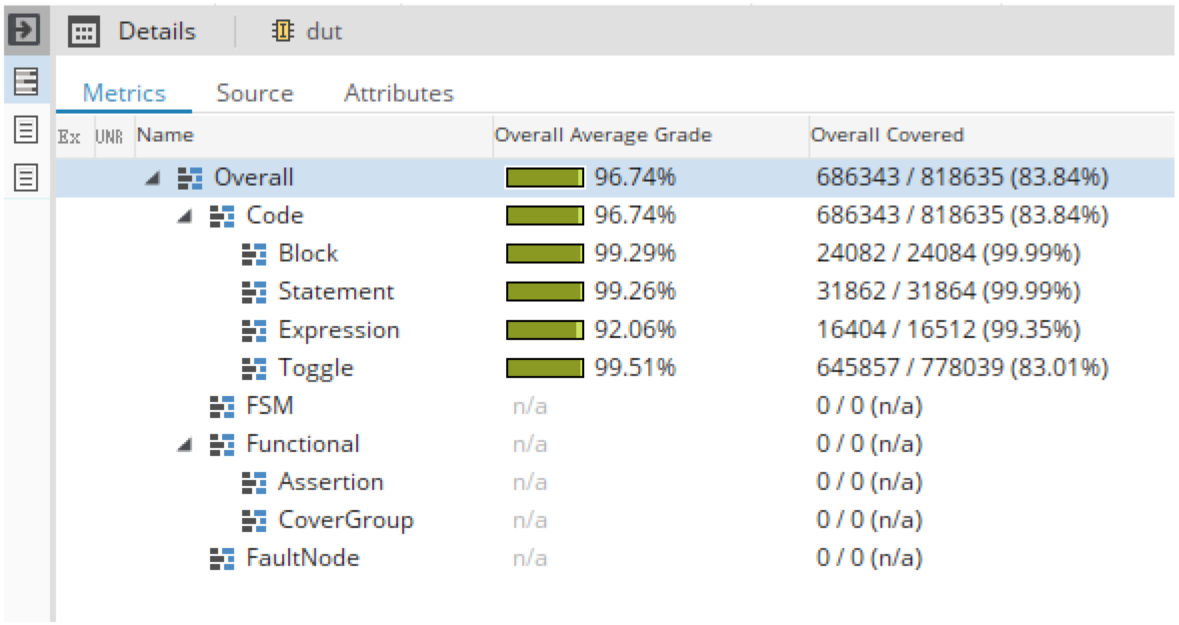
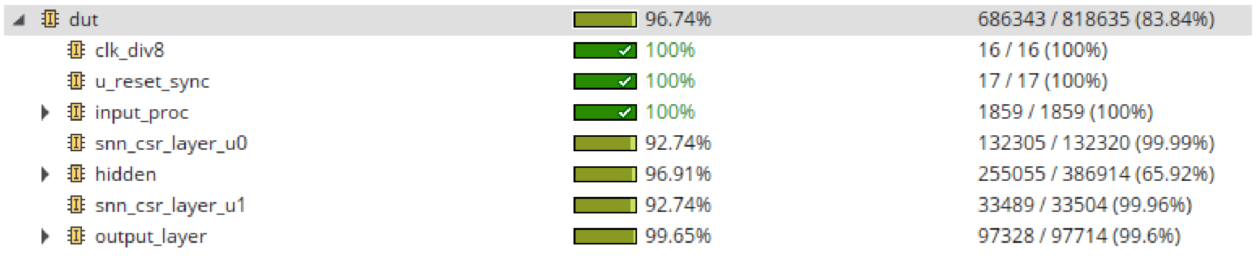


Figure 6: Output spikes generated based on the combination of random inputs, weights and threshold



Ex	UNR	Name	Overall Average Grade	Overall Covered
		Overall	96.74%	686343 / 818635 (83.84%)
		Code	96.74%	686343 / 818635 (83.84%)
		Block	99.29%	24082 / 24084 (99.99%)
		Statement	99.26%	31862 / 31864 (99.99%)
		Expression	92.06%	16404 / 16512 (99.35%)
		Toggle	99.51%	645857 / 778039 (83.01%)
		FSM	n/a	0 / 0 (n/a)
		Functional	n/a	0 / 0 (n/a)
		Assertion	n/a	0 / 0 (n/a)
		CoverGroup	n/a	0 / 0 (n/a)
		FaultNode	n/a	0 / 0 (n/a)

Figure 7: Overall DUT Code Coverage



dut	96.74%	686343 / 818635 (83.84%)
clk_div8	✓ 100%	16 / 16 (100%)
u_reset_sync	✓ 100%	17 / 17 (100%)
input_proc	✓ 100%	1859 / 1859 (100%)
snn_csr_layer_u0	92.74%	132305 / 132320 (99.99%)
hidden	96.91%	255055 / 386914 (65.92%)
snn_csr_layer_u1	92.74%	33489 / 33504 (99.96%)
output_layer	99.65%	97328 / 97714 (99.6%)

Figure 8: SNN Design Sub-blocks code coverage

7. Summary

This dissertation presented the RTL design and UVM-based verification of a small-scale Spiking Neural Network (SNN), designed to mimic biological neurons using digital logic. The project successfully developed a two-layer SNN comprising input, hidden, and output layers, using the Leaky Integrate-and-Fire (LIF) neuron model and synaptic connections with configurable weights and thresholds. The design was verified using a modular and reusable SystemVerilog UVM testbench, ensuring correctness across various input patterns and neuron configurations.

The verification environment incorporated agents for both APB and custom SNN protocols, a comprehensive register model, randomized and directed stimulus sequences, and a scoreboard for output comparison. Simulation results demonstrated the correct functionality of spike generation, weighted summation, and spike propagation through layers. The scoreboard and coverage reports validated the robustness and reusability of the testbench.

Advantages of the Current Design

- **Biologically Inspired Design**: Implements energy-efficient spike-based processing using LIF neuron models, suitable for edge AI and pattern recognition applications.
- **Modular RTL Architecture**: Clearly defined neuron, synapse, CSR, and spike processing modules allow for easy scaling and maintenance.
- **UVM-Based Verification**: Fully functional, layered UVM environment ensures systematic and exhaustive verification with reuse-friendly components.
- **Scalability and Configurability**: The use of parameterized System Verilog modules and APB-accessible CSRs allows easy reconfiguration of network parameters like weights, thresholds, and leak factors.

Limitations

- **Fixed Network Size**: The current design supports a fixed 64-32-16 (input-hidden-output) neuron configuration, limiting its use in applications requiring deeper or more complex topologies. Also, the leak factor is not supported in the current design
- **No On-Chip Learning**: Synaptic weights are static and configured externally via CSRs; no support for real-time weight adaptation or training is implemented.
- **Lack of Power Analysis**: While the architecture is inspired by low-power neuromorphic computing, the design has not undergone power or area optimization or post-synthesis analysis.
- **Functional Coverage Not Implemented**: Due to time constraints, functional coverage models were planned but not implemented, leaving some verification opportunities unexplored.

Future Expansion

- **Support for On-Chip Learning**: Integration of Spike-Timing-Dependent Plasticity (STDP) or Hebbian learning could enable adaptive learning capabilities directly within hardware.
- **Multi-Layer Generalization**: Refactor the architecture to support deeper networks with configurable number of layers and neurons per layer.
- **Power and Area Optimization**: Perform RTL-to-Gate-level synthesis and power estimation to explore design trade-offs for edge deployment.
- **Functional Coverage Addition**: Develop covergroups and coverpoints to capture stimulus and DUT state coverage metrics, enhancing verification completeness.
- **Integration with AI Toolchains**: Create a Python or MATLAB interface to allow seamless mapping of trained weights and datasets from standard SNN training libraries.
- **FPGA Prototyping**: Map the design to an FPGA (e.g., Zynq-7000) for real-time testing and deployment in embedded neuromorphic use-cases.

8. Plan of Work

Table 12: Plan of Work

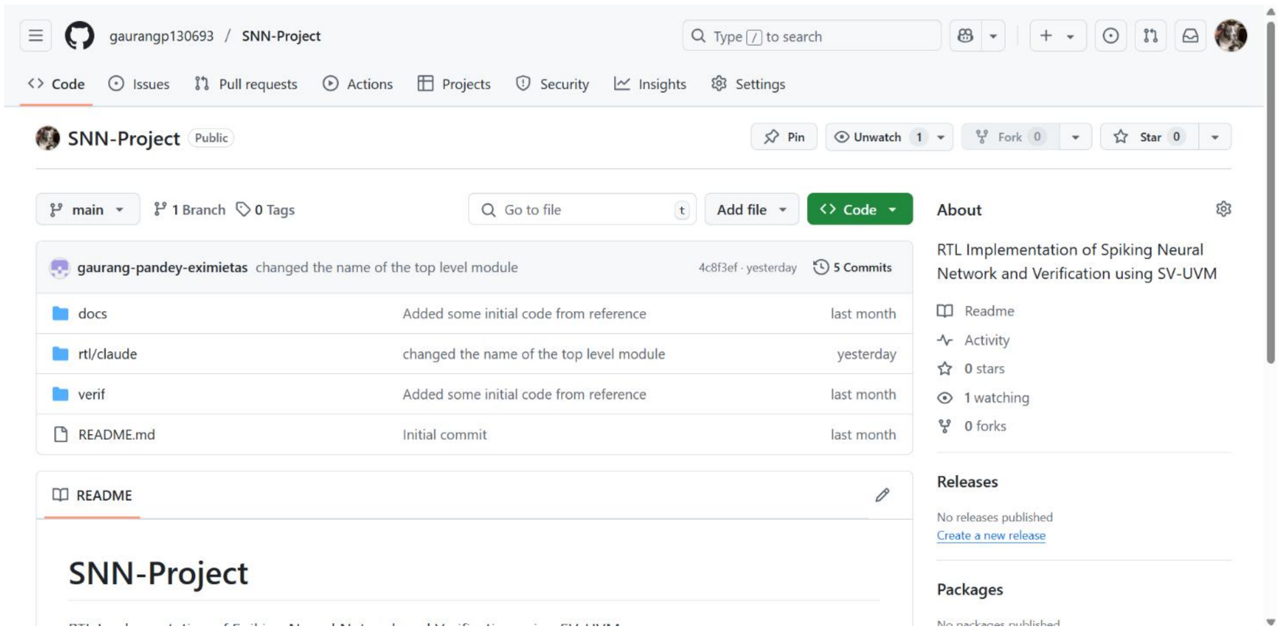
Phases	Start Date-End Date	Work to be done	Status
Dissertation Outline	18th Jan - 25 Jan 2025	Literature Review and prepare Dissertation Outline	COMPLETED
Design & Development	1st Feb – 19th Mar 2025	Design & Development Activity	COMPLETED
Mid-Semester Progress report	20th Mar – 27th Mar 2025	Mid-Semester progress report on Viva Portal	COMPLETED
Design Verification	20th March – 06th Apr 2025	Implement review points from BITS Faculty and Complete the Verification	COMPLETED
Dissertation Review	07th Apr – 16th Apr 2025	Submit Dissertation to Supervisor & Additional Examiner for review and feedback	COMPLETED
Submission	17th Apr– 24th Apr 2025	Final Review and submission of Dissertation	COMPLETED

9. Literature References

- Books:
  - T. C. Stewart et al., "Introduction to Spiking Neural Networks," Springer, 2021.
  - Chris Spear and Greg Tumbush, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features," Springer, 2022.
- Papers and Articles:
  - Maass, Wolfgang, "Networks of Spiking Neurons: The Third Generation of Neural Network Models," Neural Networks, 1997.
  - Schuman, Catherine D., et al., "A Survey of Neuromorphic Computing and Neural Networks in Hardware," Neurocomputing, 2017.
  - S. Furber et al., "The SpiNNaker Project," Proceedings of the IEEE, 2014.
- Industry Standards:
  - Accellera Systems Initiative, "UVM User Guide Version 1.2," Accellera Systems Initiative, 2014.
- Web Resources:
  - Xilinx: Neuromorphic Computing Overview
  - SystemVerilog UVM Resources

10. Github Link for Code Repository

Link : https://github.com/gaurangp130693/SNN-Project



gaurangp130693 / SNN-Project

Type / to search

Code · Issues · Pull requests · Actions · Projects · Security · Insights · Settings

SNN-Project Public

Pin · Unwatch 1 · Fork 0 · Star 0

main · 1 Branch · 0 Tags · Go to file · Add file · Code

gaurang-pandey-eximietas changed the name of the top level module		4c8f3ef · yesterday · 5 Commits
docs	Added some initial code from reference	last month
rtl/claude	changed the name of the top level module	yesterday
verif	Added some initial code from reference	last month
README.md	Initial commit	last month

README

SNN-Project

About

RTL Implementation of Spiking Neural Network and Verification using SV-UVM

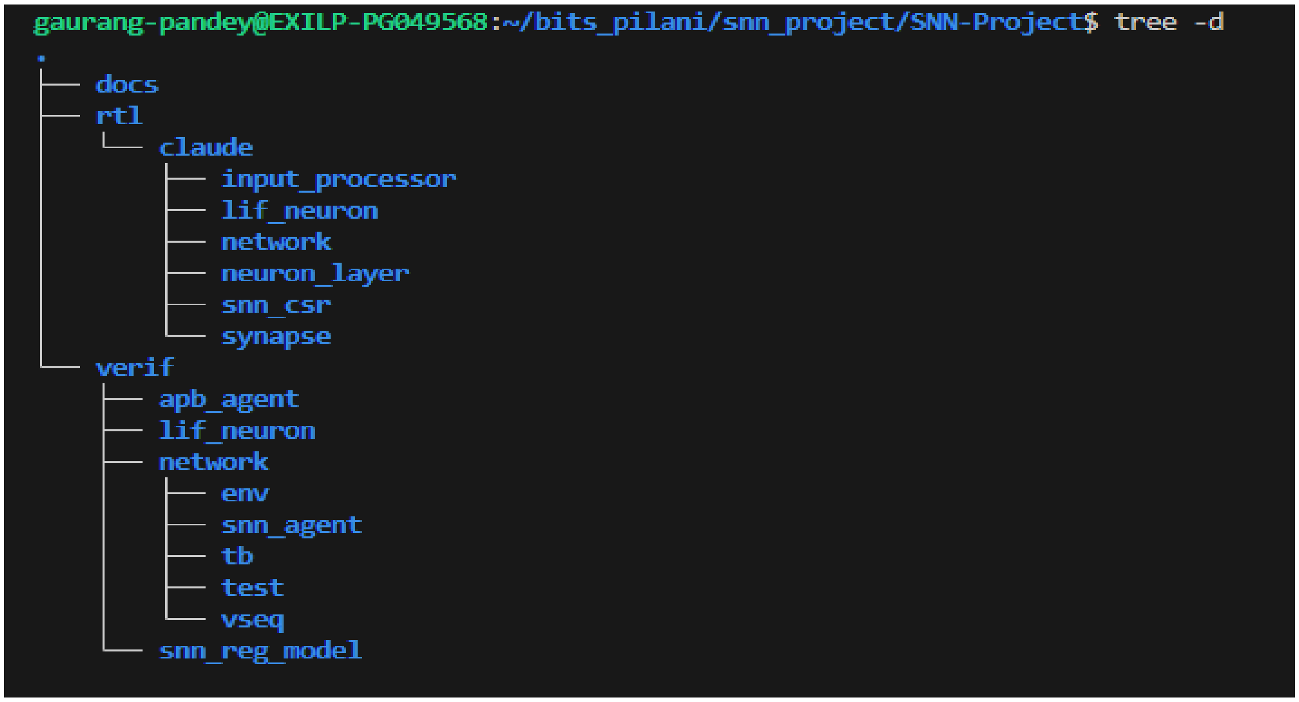
Readme
Activity
0 stars
1 watching
0 forks

Releases

No releases published
Create a new release

Packages

Figure 9: Github Code Repository Snapshot



```
gaurang-pandey@EXILP-PG049568:~/bits_pilani/snn_project/SNN-Project$ tree -d
.
├── docs
├── rtl
│   └── claude
│       ├── input_processor
│       ├── lif_neuron
│       ├── network
│       ├── neuron_layer
│       ├── snn_csr
│       └── synapse
└── verif
    ├── apb_agent
    ├── lif_neuron
    ├── network
    │   ├── env
    │   ├── snn_agent
    │   ├── tb
    │   ├── test
    │   └── vseq
    └── snn_reg_model
```

Figure 10 : Project Directory Structure

11.Checklist of Items for the Final Dissertation

This checklist is to be duly completed, verified and signed by the student.

1.	**Is the final report neatly formatted with all the elements required for a technical Report?**	**Yes**
2.	Is the Cover page in proper format as given in Annexure A?	**Yes**
3.	Is the Title page (Inner cover page) in proper format?	**Yes**
4.	(a) Is the Certificate from the Supervisor in proper format? (b) Has it been signed by the Supervisor?	**Yes** **Yes**
5.	Is the Abstract included in the report properly written within one page? Have the technical keywords been specified properly?	**Yes** **Yes**
6.	Is the title of your report appropriate? **The title should be adequately descriptive, precise and must reflect scope of the actual work done.** Uncommon abbreviations / Acronyms should not be used in the title	**Yes**
7.	Have you included the List of abbreviations / Acronyms?	**Yes**
8.	Does the Report contain a summary of the literature survey?	Yes
9.	Does the Table of Contents include page numbers? (i). Are the Pages numbered properly? (Ch. 1 should start on Page # 1) (ii). Are the Figures numbered properly? (Figure Numbers and Figure Titles should be at the bottom of the figures) (iii). Are the Tables numbered properly? (Table Numbers and Table Titles should be at the top of the tables) (iv). Are the Captions for the Figures and Tables proper? (v). Are the Appendices numbered properly? Are their titles appropriate	**Yes** **Yes** **Yes** **Yes** **Yes** **Yes**
10.	Is the conclusion of the Report based on discussion of the work?	**Yes**
11.	Are References or Bibliography given at the end of the Report? Have the References been cited properly inside the text of the Report? Are all the references cited in the body of the report	**Yes** **Yes** **Yes**
12.	Is the report format and content according to the guidelines? The report should not be a mere printout of a PowerPoint Presentation, or a user manual. Source code of software need not be included in the report.	**Yes**

Declaration by Student:
I certify that I have properly verified all the items in this checklist and ensure that the report is in proper format as specified in the course handout.

Gaurang Pandey

Place: Ahmedabad, Gujarat **Signature of the Student**

Date: 24-04-2024 **Name: Gaurang Brijbhushan Pandey**

ID No.: 2023HT80003