# *From Brain-Inspiration to Silicon-Realisation*: SNN-based Smart Watchdogs for RISC-V Fault Detection

**PhD Student:** David Simpson

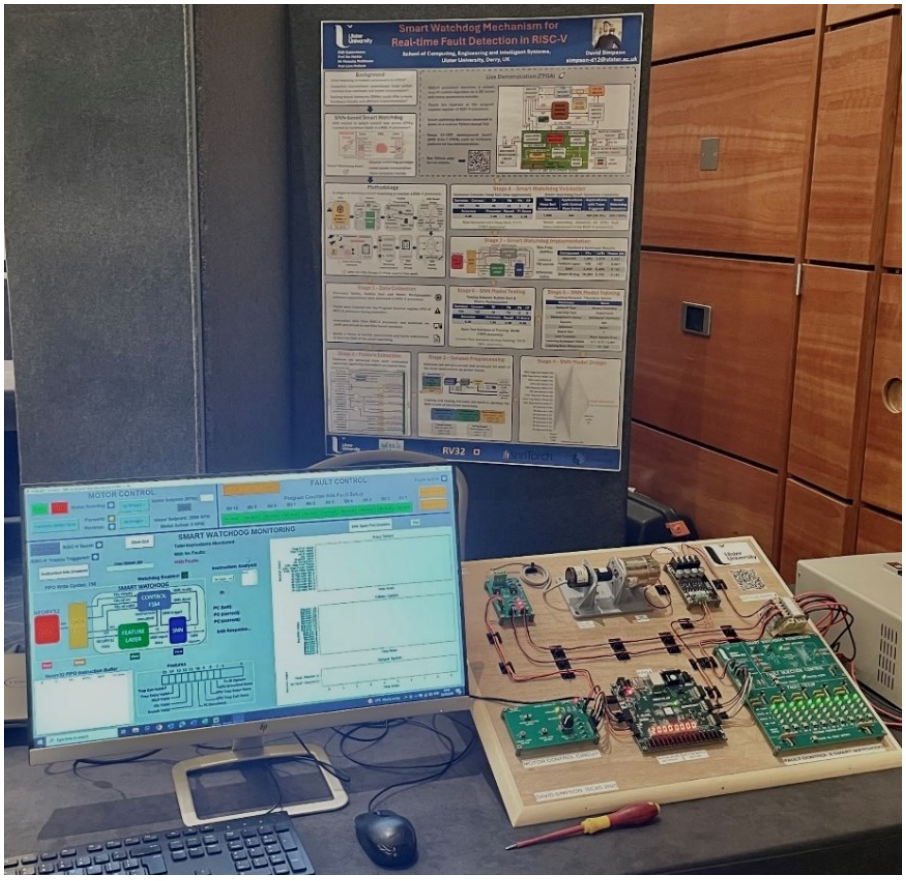
**Supervisor Name:** Professor Jim Harkin

**University:** Ulster University

**Team Number:** AOHW25_620

**YouTube Video:**

**Github Repo:** DS-567/AMD-AOHW25_620: AMD Open Hardware Design Competition 2025 Contribution: SNN-based Smart Watchdogs for RISC-V Fault Detection




**Email:** simpson-d12@ulster.ac.uk

**LinkedIn:** David Simpson | LinkedIn

## Table of Contents

## Abstract

This project developed the first known Spiking Neural Network (SNN) model capable of detecting faults in a modern embedded processing architecture, referred as smart watchdogs. This research went from brain-inspiration to silicon-realisation, implementing the smart watchdog on AMD FPGAs for validating proof of concept during live, real-time execution monitoring of a physical RISC-V processor (Neorv32). Capable of distinguishing between normal execution and control flow errors with 98% coverage regardless of the application software ran on the RISC-V enables the smart watchdog to be re-used for different compiled software without requiring any hardware changes and can detect errors that the RISC-V internal fault detection mechanisms fail to. A complete demonstrator highlights the re-usability of the smart watchdog component, deployed to monitor a realistic motor control task on Neorv32 with additional functionality enabling user-friendly interaction and powerful data visualisation at instruction level. Benchmarking the smart watchdog against other watchdogs is difficult, however this work may serve as an initial benchmark after highlighting the fault detection capability of SNNs in an embedded processing architecture.

In this PhD, a custom hardware architecture was created to gather real-world RISC-V instruction data to enable training a SNN model. Additionally, VHDL implementation of the core SNN components and a framework of porting software models to FPGA hardware, enables straightforward prototyping of smart watchdogs from concept to silicon – scalable to other non-watchdog tasks. This design submission shares this methodology with the research community via a Github repository.



## Acknowledgements

Grateful for the scholarship provided by the Department for the Economy (DfE) to support this PhD research project.

Special thanks to Stephan Nolting for his expertise on Neorv32 and the RISC-V architecture and the PhD supervisors for their time, effort and advice.

## List of Figures



## List of Equations



## List of Tables

# 1. Introduction

## 1.1 Motivation

The continuous shrinking of semiconductor geometries has enabled more complex and power-efficient silicon designs through increased transistor density. Despite higher performance, modern transistor-dense silicon architectures, such as processors, are becoming increasingly more vulnerable to hardware faults as a result of down-scaling [1]. Google [2], Meta [3-4] and Alibaba [5] all recently reported occurrences of Silent Data Corruption (SDC) from manufacturing defects in their large scale CPU server fleets, despite devices passing post-fabrication testing. ARM also expressed similar reliability concerns in the embedded domain, conducting studies on radiation-induced soft faults such as Single Event Upsets (SEUs) in their micro-architectures [6]. As the sub-nanometre era brings new reliability challenges, particularly for embedded processors deployed in safety-critical applications such as space exploration, fault detection is now more crucial than ever.

System designers face strict power, area, and performance constraints, all of which are negatively impacted by traditional fault detection mechanisms [7]. Hardware redundancy techniques like Triple Module Redundancy (TMR) and dedicated watchdog processors provide effective fault coverage but add significant overheads. More importantly, a reliance is placed on voting and checker circuits [8-11], which share the same silicon vulnerabilities and failure of checker circuits is a direct threat to the reliability of the system.

In contrast to silicon, the biological brain operates not through immense computational power, but instead by extreme specialisation and efficiency, and demonstrates all desirable watchdog traits, i.e. intelligence, low power, and reliability. Recent research has explored Machine Learning (ML) and Artificial Neural Networks (ANNs) for fault detection in processors with promising results [12-17]. Neural networks are robust structures due to their distributed and densely connected architectures – faults are unlikely to cause catastrophic failure but may still influence inference [18]. Spiking Neural Networks (SNNs) capture spiking neural dynamics and resemble the human brain most closely, offering more power-efficient and hardware friendly computing compared with traditional ANN models [19]. Further inspired by the brain's self-repairing capabilities through neuro-glial cells, i.e. astrocytes [20-22], this PhD introduces *smart watchdogs* – a novel and innovative concept that leverages SNNs to provide a more efficient and dependable watchdog mechanism for fault detection in embedded CPUs.

## 1.2 Smart Watchdog Concept

Conceptually, the SNN acts as the brain of the smart watchdog, responsible for decision-making (figure 1.1). Mimicking biology, instruction-level information from the monitored CPU is received through a digital representation of dendrites (inputs), which are first encoded into spikes. These spikes are processed by the SNN and based on offline training, can distinguish between normal and faulty execution patterns. The SNN decisions can be observed by decoding spikes on the axons (outputs) to ultimately detect faults within the CPU.

The SNN is the key decision-making component within the smart watchdog and main contribution in this work. The aim is to realise a watchdog that operates independently of the CPU and is ultimately low power, hardware friendly and more dependable. SNNs operating on custom analog or asynchronous neuromorphic platforms can truly leverage sparsity and event-driven computation, only demanding power when necessary. Dependability through integration of astrocytes provides the SNN the ability to detect when inference is corrupted by hardware faults and modulate internal model parameters to realise self-repair. For safety-critical tasks, a watchdog that can tolerate soft errors itself is a significant advantage.

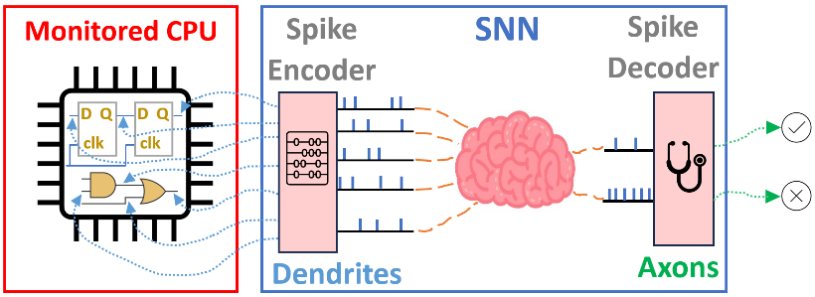



***Figure 1.1** – Conceptual SNN for CPU monitoring.*

There are multiple key circuits in a processing architecture (e.g. instruction fetch, execute, memory accesses). As opposed to training a single large, deep network, multiple smaller smart watchdogs could be developed to monitor these key circuits, similar to how the human brain is understood to be arranged as separate, specialised functional units (i.e. regions). This could simplify the significant task of large model development into simpler, dedicated, robust watchdog circuits.

For clarity in this report, embedding astrocytes is a major motivation behind selecting SNNs as the decision-maker in the smart watchdog and remains future work. The focus of this PhD was to firstly evaluate the plausibility of fault detection within a processor architecture.

## 1.3 Project Overview

This PhD represents the first known instance of applying SNNs to detect faults in a processor architecture. In this work, a software SNN model was trained and further implemented on FPGA as a fully functional prototype for proof of concept. Full details of the PhD work can be found in peer reviewed IEEE publications:

- ISCAS 2025 conference paper [23].
- TCAS II: Express Briefs (invited) journal paper [24].

As an additional stage, a hardware demonstrator [25] was developed and presented in the ISCAS 2025 live demo session, showcasing the smart watchdog deployed to monitor a RISC-V CPU (Neorv32 [26]) executing a realistic motor-control task. Given the hardware focus of this design competition, this report and submission concentrates on the design and implementation of both the smart watchdog and the ISCAS demonstrator, which the latter highlights the reusability aspect of the watchdog. Beyond the publications, this report further discusses the feasibility of integrating smart watchdogs into real-world processors, offering additional novelty.

## 1.4 Report Outline

Chapter 1 has set the motivation behind smart watchdogs in the PhD project. Chapter 2 briefly summarised the 8-stage methodology used to develop the first prototype smart watchdog. Chapter 3 then outlines the design and hardware implementation of the prototype smart watchdog, with extended focus on two different hardware implementations of the SNN model. Chapter 4 discusses the implementation of the ISCAS demonstrator that deploys the smart watchdog. Chapter 5 presents the performance results of the smart watchdog in addition to a discussion about the future of smart watchdogs integrated with processing architectures. And finally chapter 6 describes the custom data collection framework that was produced in the beginning of the PhD, including it within the design submission.

## 2. Methodology

Control flow monitoring is a common watchdog task. At compile time, instructions are placed in memory and the CPU fetches instructions dynamically based on the Program Counter (PC) register, to execute the program algorithm. Hardware faults can corrupt this sequence of instructions executed, causing the processor to deviate away from the correct program flow, known as a Control Flow Error (CFE). The smart watchdog developed in this work is trained to detect CFEs in the RISC-V architecture at instruction-execution level, following an 8-stage methodology (figure 2.1), depicting a complete ML workflow.

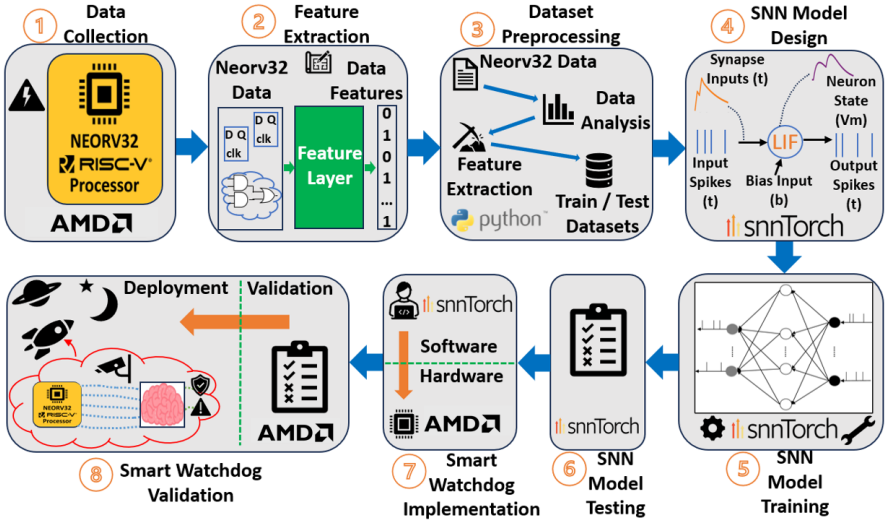



***Figure 2.1** – 8-stage methodology for developing smart watchdog.*

**Stage 1 – Data Collection**: A custom hardware architecture and framework was developed to inject faults and collect SNN training data from a modern RISC-V CPU (Neorv32) during execution of three different C software benchmarks.

**Stage 2 – Feature Extraction**: Features were determined to extract meaningful information from instructions in the base 32I RISC-V architecture relating to control flow.

**Stage 3 – Dataset Preprocessing**: For every instruction collected from Neorv32, features were extracted and stored in datasets used to develop the SNN model.

**Stage 4 – SNN Model Design**: SNN model design parameters were selected such as network architecture, neuron type, training algorithm, loss functions etc.

**Stages 5 and 6 – SNN Model Training and Testing**: The SNN model was trained and tested with the application datasets using SNNTorch [27], [28].

**Stage 7 – Smart Watchdog Implementation**: Moving further than a software SNN model, a framework was created to accelerate the process of implementing a hardware model (FPGA).

**Stage 8 – Smart Watchdog Validation**: The smart watchdog underwent a final hardware validation stage where it performed real-time fault monitoring instantiated beside the RISC-V CPU during a live fault injection experimentation. Results were extracted off-FPGA to process with Python scripts.

While the target platform for the smart watchdog is neuromorphic, this PhD project showcases the power of FPGAs, and in particular, the effectiveness of AMD FPGAs and their software toolchains for prototyping and researching novel AI approaches. The smart watchdog is not just simulated, but validated in real hardware with a space-experienced RISC-V core (Neorv32).

## 3. Smart Watchdog Design & Implementation

The smart watchdog is the key novelty of this submission. It was previously trained and implemented on FPGA during the PhD work, and has been extended with minimal logic to support the demonstrator. Figure 3.1 depicts the smart watchdog component integrated with the RISC-V CPU, highlighting its three main components: the **SNN**, **feature extraction layer** and **control FSM**. During execution, selected RISC-V instruction data is written to an AMD FIFO IP [29] buffer every clock cycle. The control FSM manages valid data transfer from the FIFO to the feature extraction layer, extracting 16 pre-determined binary features per instruction. These 16 features are inputs to the SNN, triggered by the FSM. Once inference completes (i.e. SNN is done), one of the two output classes are asserted based on the inference and the process repeats until all data in the FIFO has been processed (i.e. FIFO is empty).

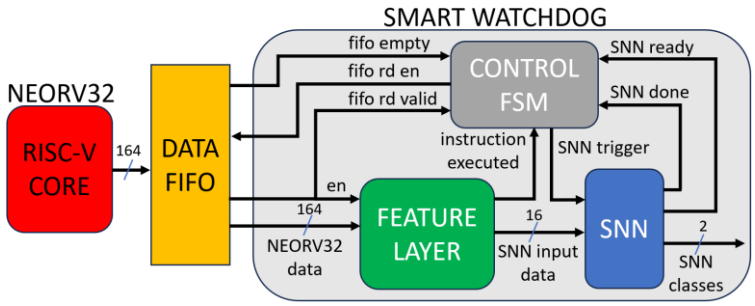



***Figure 3.1*** *– Block diagram of the smart watchdog with main components.*

Integrating the smart watchdog to monitor control flow of a RISC-V processor in real hardware requires consideration. As Neorv32 is a multi-cycle architecture, the watchdog must detect when an instruction has been executed to check control flow. Figure 3.2 illustrates this process with a waveform showing three sequential instructions with respect to the PC, instruction register (IR) and the Execute FSM state. In this architecture, the dispatch state issues the next instruction and the execute state begins executing the instruction. Other CPU states such as *branched* provide context to internal CPU operations (e.g. taking a branch). The watchdog will check if the updated PC value is consistent based on the previous PC, the instruction and other CPU information. In figure 3.2, two instructions (1 and 2) are highlighted as examples of how the watchdog will monitor and classify two instructions.

**Instruction 1**: The PC incremented by 4 bytes (0x324 → 0x328).

**Instruction 2**: A branch was taken (0x328 → 0x2D4) and the signed offset of -84 encoded as an immediate value in the instruction was added to the PC.

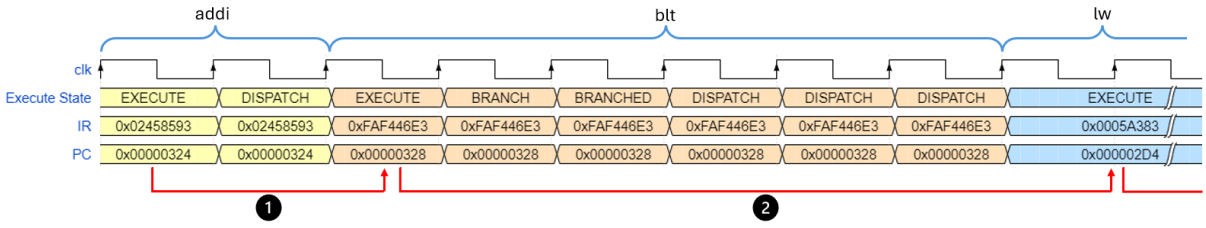



***Figure 3.2*** *– Neorv32 execution waveforms.*

### 3.1 Feature Extraction Layer

16 binary features were selected for this smart watchdog to capture key information related to control flow within the RISC-V architecture. The features are extracted from six registers in Neorv32 (instruction data) as shown in the high level block diagram in figure 3.3 (a). As Neorv32 is multi-cycle, all data for each instruction is sequentially passed through the feature

extraction layer, distilling all 164 bits down to 16 binary features, with each feature capturing single piece of information relating to control flow within the RISC-V core. Combined, these 16 features enable accurate monitoring of RISC-V control flow.

The feature layer in hardware consists of flip flops, a reduced sign-extended RISC-V instruction offset decoder, 2 adders and six combinational 32-bit comparators. A 2-stage flip-flop pipeline captures the current and last values of the PC and IR enabled by the execute state when new instruction data arrives (figure 3.3b). The state comparators (yellow) are realised with synchronous latches while the remaining comparators (orange) are asynchronous logic. After processing each instruction, the 16 extracted features are latched into 16 flip-flops for SNN input, and the instruction executed flip-flop and latches are reset.

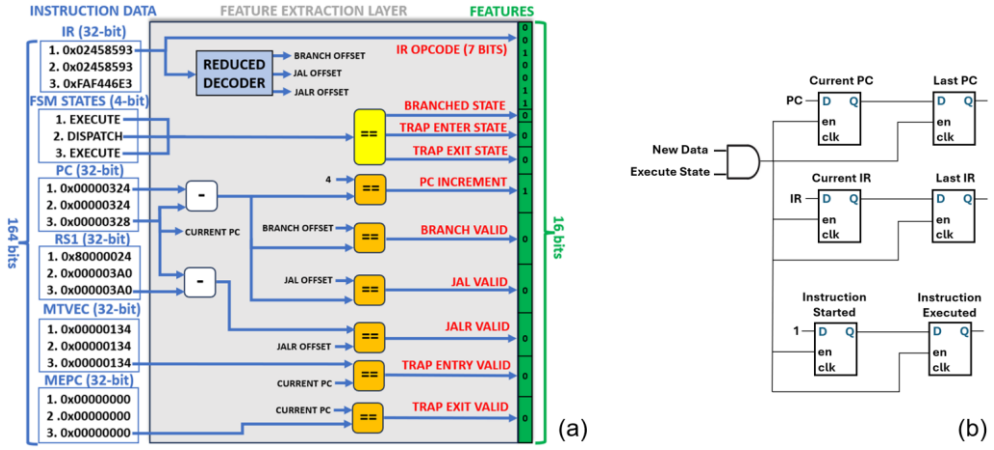



***Figure 3.3*** *– Feature layer component within the smart watchdog:*
*(a) high level block diagram of the feature layer. (b) 2-stage pipeline of PC and IR, and instruction executed logic.*

## 3.2 SNN

The SNN is the key decision-making element of the smart watchdog to classify the features extracted from RISC-V instruction data into two classes; normal execution or control flow error.

### 3.2.1 Model Overview

The SNN model is a binary classifier with a fully-connected, three-layer architecture of 16, 20, and 2 neurons (Figure 3.4). The input layer encodes 16 binary features into spike trains, while hidden and output layers process the spikes. Output neuron spiking patterns determine the inferred class. LIF neurons in the hidden and output layers provide a balance between model complexity and hardware efficiency [30]. Inference completes in 10 timesteps.

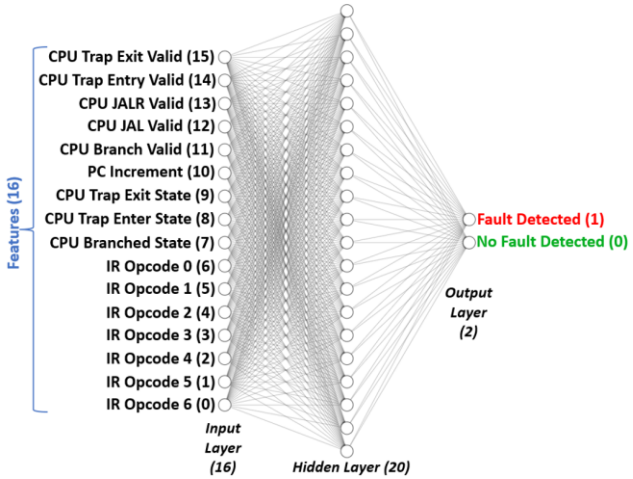



***Figure 3.4*** *– SNN model architecture.*

The SNN was trained with SNNTorch using supervised learning. Features were extracted from RISC-V instruction data during the stage 1 – data collection (figure 2.1) and labelled as either normal execution or CFE. The model was developed with three software C benchmarks (Fibonacci series, bubble sort, and matrix multiplication) over 400 epochs with a standard mean square error loss function, Adam optimizer and learning rate scheduler. Full details of the SNN design and development stages are provided in the publications from this work [23], [24]. The standard binary classifier performance metrics were used to evaluate the SNN. Overall, the trained model achieved ~98% accuracy on testing data, and ~96% on features that were unseen during training, demonstrating strong generalization.

### 3.2.2 Model Framework

A custom framework for porting SNNTorch software models to FPGA hardware was developed, as shown in figure 3.5.

1) A Python script accesses and extracts the pre-trained SNN model parameters, i.e. weights, biases and thresholds.
2) Parameters are pre-processed, quantized and stored as text files in a specified directory i.e. named “setup text files”.
3) A pre-designed SNN component is instantiated by the smart watchdog component that takes generics, i.e. layer sizes and dimensions, beta, timesteps and the file path of the setup text files, and automatically builds the model during bitstream generation.

**Note**, two Python scripts for the two SNN implementations, as described in the next section.


***Figure 3.5*** *– SNNTorch software models to FPGA hardware models (framework).*

### 3.2.3 Model implementation

The individual components of the SNN hardware component were designed, implemented, verified, and instantiated according to the SNNTorch model and generics. Figure 3.6 depicts a top level block diagram of the SNN (a), and its main components (b). Once triggered, the input data sample is registered, encoded into spikes by the input layer, and subsequently processed by layers for the specified number of timesteps. The output neuron spikes are counted and decoded into inference classes based on the decoder design.

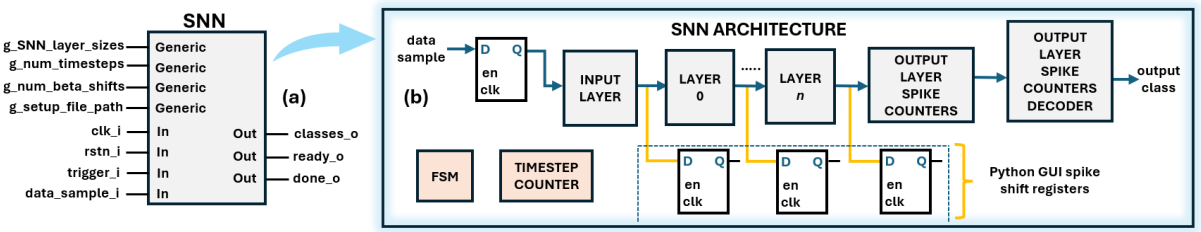



***Figure 3.6*** *– SNN component: (a) Top level block diagram. (b) SNN architecture.*

**Note**, that for clarity in this report, two different hardware implementations of the SNN software model were developed in this PhD. A first implementation based on time-multiplexing was designed for any network size (named ***SNN***), and a second implementation was more optimised for experimenting towards lower-latency inference (named ***Fast SNN***). Both HDL source codes can be found in the Github repository and the Fast SNN is deployed in the demonstrator. The remainder of this section describes inference at component level, highlighting the key differences between the SNN and the fast SNN implementations.

**Input Layer**: The SNN communicates with the external world via spikes, and the input layer functions purely as a spike encoder. The extracted binary features are encoded into spiketrains using a custom rate-based scheme (figure 3.7a), where features with logic values of 0 and 1 are represented as spiketrains of 2 and 8 spikes out of the 10 timesteps, respectively. This scheme offers adequate distinction between logic 0 and 1 features, as well as ensuring that sufficient spikes are present in the network if many features are logic 0. Figure 3.7 (b) shows the input layer hardware design. Each bit of the input sample (i.e. the 16 binary features) selects either the low or high spiking frequency stored in the spiketrain register, i.e. encoding scheme. A timestep counter indexes which spike is output for each bit, and an output register stage added to prevent the encoder's maximum frequency from being restricted.

**LIF Layer**: Each layer instantiates LIF neurons, passes input spikes to neurons for processing, and triggers any subsequent layers. The layers are the first significant difference between the SNN and Fast SNN implementations. In the SNN implementation (figure 3.8a), input spikes are time-multiplexed to neurons at each timestep, orchestrated by a FSM and synapse counter. Along with spikes, the synapse counter and other FSM output control lines are passed to the LIF neurons to handle accumulation one spike at a time.

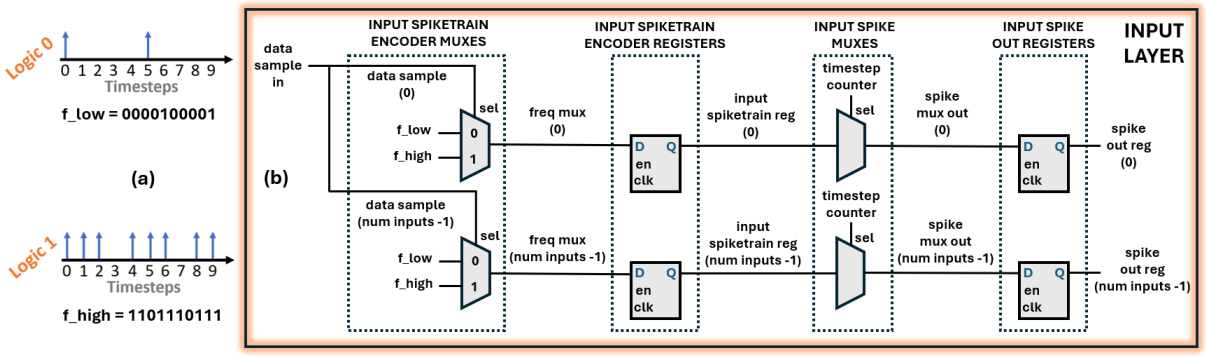



***Figure 3.7*** *– Spike encoding scheme for each binary feature.*

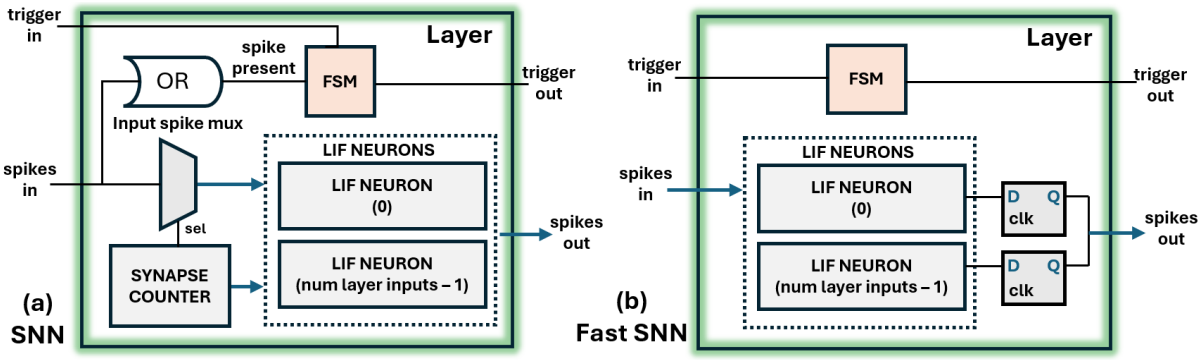



***Figure 3.8*** *– LIF Layer component block diagram: (a) SNN layer. (b) Fast SNN layer.*

In the Fast SNN implementation (figure 3.8b), input spikes are passed directly to the neurons in parallel, along with the FSM output control lines. The layers are pipelined, allowing the next timesteps input spikes to be processed of the next while the output layer is still computing the previous timestep. minimising idle time of layers and increases throughput. Figure 3.9 portrays the difference in layer sequencing between the SNN and Fast SNN implementations.

| Layer Processing Stages (SNN) | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Timestep | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 1 | H/L | O/L | | | | | | | | | | | | | | | | | | |
| 2 | | | H/L | O/L | | | | | | | | | | | | | | | | |
| 3 | | | | | H/L | O/L | | | | | | | | | | | | | | |
| 4 | | | | | | | H/L | O/L | | | | | | | | | | | | |
| 5 | | | | | | | | | H/L | O/L | | | | | | | | | | |
| 6 | | | | | | | | | | | H/L | O/L | | | | | | | | |
| 7 | | | | | | | | | | | | | H/L | O/L | | | | | | |
| 8 | | | | | | | | | | | | | | | H/L | O/L | | | | |
| 9 | | | | | | | | | | | | | | | | | H/L | O/L | | |
| 10 | | | | | | | | | | | | | | | | | | | H/L | O/L |
| Layer Processing Stages (Fast SNN) | | | | | | | | | | | | | | | | | | | | |
| 1 | H/L | O/L | | | | | | | | | | | | | | | | | | |
| 2 | | H/L | O/L | | | | | | | | | | | | | | | | | |
| 3 | | | H/L | O/L | | | | | | | | | | | | | | | | |
| 4 | | | | H/L | O/L | | | | | | | | | | | | | | | |
| 5 | | | | | H/L | O/L | | | | | | | | | | | | | | |
| 6 | | | | | | H/L | O/L | | | | | | | | | | | | | |
| 7 | | | | | | | H/L | O/L | | | | | | | | | | | | |
| 8 | | | | | | | | H/L | O/L | | | | | | | | | | | |
| 9 | | | | | | | | | H/L | O/L | | | | | | | | | | |
| 10 | | | | | | | | | | H/L | O/L | | | | | | | | | |

***Figure 3.9*** *– Layer processing stages in both SNN and Fast SNN designs.*

**LIF Neuron**: LIF neurons were implemented to capture the spiking dynamics and neural behaviour from SNNTorch (equation 3.1), which defines the membrane potential $U_{(t+1)}$ of each neuron at the next timestep. The input weight accumulation $WX_{(t+1)}$ w corresponds to a multiply-accumulate (MAC) operation. The term $\beta U_{(t)}$ captures the membrane potential decay and the reset mechanism, $S_{(t)}thr$, is computed if a threshold is exceeded and an output spike generated. Initially, all weights, biases and LIF neuron membrane potentials are represented as 24-bit fixed point notation with 10 integer parts and 14 fractional parts.

$$U_{(t+1)} = \underbrace{\beta U_{(t)}}_{decay} + \underbrace{WX_{(t+1)}}_{input} + bias - \underbrace{S_{(t)}thr}_{reset} \tag{3.1}$$

Both the SNN and Fast SNN layers have separate LIF neuron implementations to suit, highlighting another difference in the SNN hardware designs. Figure 3.10 shows the data path of the LIF neuron used in the time-multiplexing SNN. A register stores the neuron membrane potential, and the synaptic weights are stored in a tightly coupled, read-only RAM that is initialised by the setup text files during bitstream generation. Biases are stored as constant LUTs in this work, however, to scale to larger architectures, memory such as BRAM can be leveraged (like weights). The synapse counter addresses the weights memory, and the layer FSM orchestrates the LIF neuron as displayed in the state diagram in figure 3.11.

1) **Neuron decay**: $\beta U_{(t)}$ multiplication is handled with a hardware-friendly right-shift and subtraction operation. Consider $U_{(t)} = 4$ and $\beta = 0.75$.

   **1.** $4 \times 0.75 = 3$, **2.** $4 \gg 2 = 1\ (shift\ result)$, **3.** $4 - 1\ (shift\ result) = 3$

   A signed right shift component along with the adder/subtractor core controlled via the add/sub multiplexor handles the operation. The multiplexor selects are zero, subtracting the shifted membrane potential and writing the result back into the register.
2) **Weight accumulation**: When no input spikes are present (OR gate output low), the hardware skips accumulation and proceeds to **3)**. To accumulate weights, each input spike controls the select line of a 2-to-1 weight multiplexor, which chooses between zero or the corresponding weight from the weight memory (addressed by the synapse

counter), which is then added to the neuron's membrane potential. An extra sign bit is added to the weight (MSB) to determine whether to add or subtract the weight, representing excitatory or inhibitory synapses.

3) **Bias**: The bias is then added to the neuron's membrane potential with the bias MSB sign bit signalling an excitatory or inhibitory bias in the same manner as the weights.
4) **Spike**: The output spike flip flop is updated with the result of the signed greater than (>) comparator, that checks for the membrane potential exceeding the threshold.
5) **Reset mechanism**: Finally if a neuron spiked, the threshold is subtracted from the membrane potential. If the neuron didn't spike, this is essentially skipped by subtracting zero from the membrane potential.

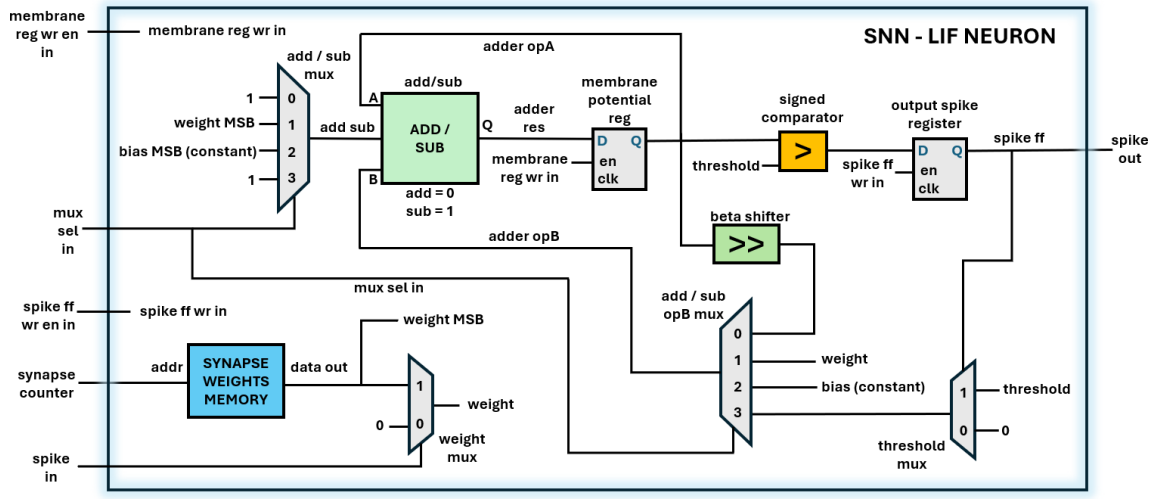



***Figure 3.10*** *– SNN LIF neuron data path schematic.*

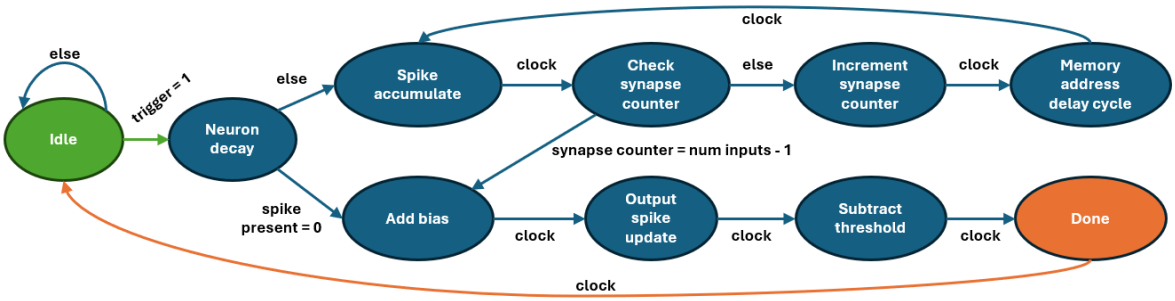



***Figure 3.11*** *– Layer FSM flow diagram for sequencing time-multiplexing LIF neurons.*

The time-multiplexing approach allows for large network sizes to be implemented practically, at the cost of slow latency. As a first step to exploring latency improvements alongside pipelining, the Fast SNN LIF neuron utilises dedicated adder trees to accumulate the weights at the synapse. Figure 3.12 and 3.13 presents the LIF neuron implementation and the layer FSM flow diagram, respectively.

Instead of processing input spikes sequentially, the neuron can process all spikes. The neuron body is split into two processing stages. Weights are now stored locally in the neuron as constant LUTs rather than in memory, eliminating RAM latency. They are initialized from setup text files in two's complement format to simplify arithmetic in the adder trees. Generic-ness sacrifices in the Fast SNN implementation due to the custom, experimental adder trees.

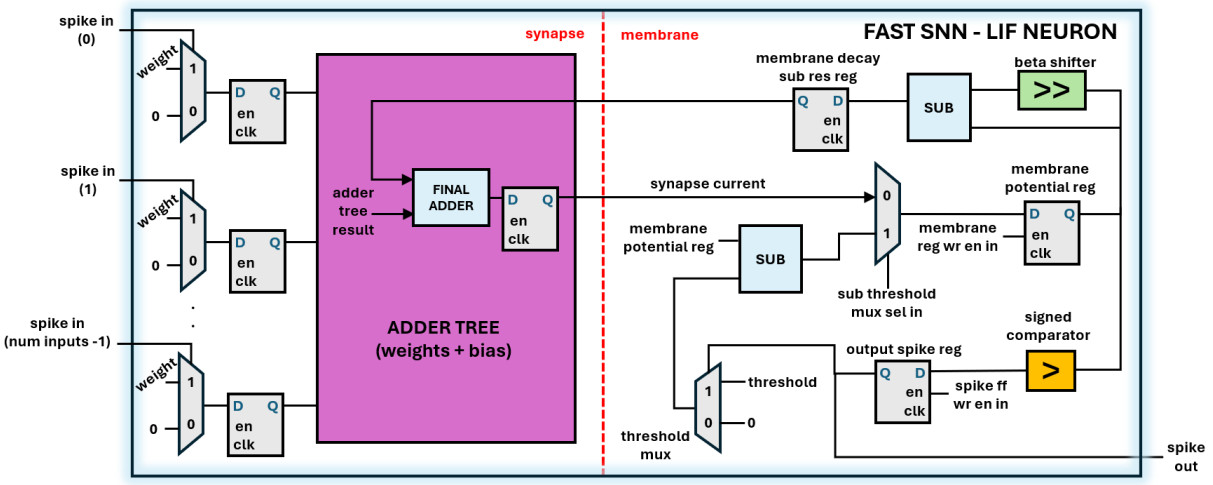



***Figure 3.12*** *– Fast SNN: LIF neuron data path schematic.*

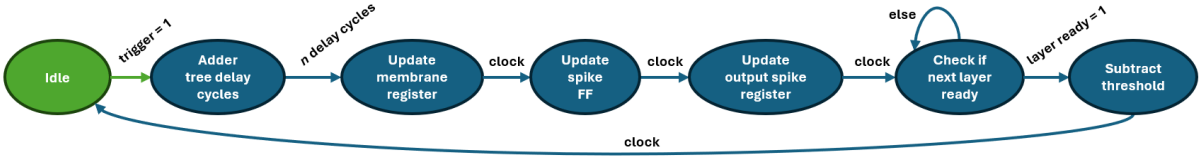



***Figure 3.13*** *– Layer FSM flow diagram for sequencing adder tree-based LIF neurons.*

1) **Weight accumulation**: A 2-to-1 weight multiplexor and a register for each input spike are placed before the adder tree, selecting either zero or the corresponding weight to feed into the adder tree. The number of clock cycles required to sum all inputs, including the bias, depends on the input vector width. The final adder stage sums the signed membrane decay to produce the synapse current for the timestep. All adder tree stages are registered.
2) **Update membrane**: The synapse current is stored in the membrane potential register.
3) **Spike Update**: The output spike FF is updated with the signed membrane potential and threshold comparison result.
4) **Spike**: The output spike FF's are registered for the next layer.
5) **Wait for next layer**: The downstream layer is triggered when ready.
6) **Reset mechanism**: Finally if a spike occurred, the threshold is subtracted from the membrane potential.

## Output Layer Spike Counters and Decoding:

Output spike decoding takes the highest output neuron spike counter as the class winner for the inference. There are two different designs for the SNN and FAST SNN implementations. The time-multiplexing SNN utilises time to decode outputs spike counts with the hardware circuit described in figure 3.14. Each output spike count of the vector array is indexed (multiplexor) and passed into the decoder. A register keeps track of both the max value and its index in the array (neuron index). After iterating through the entire output spike count array, the class decoder sets the winning class high (if valid). As this SNN only has two output neurons, the Fast SNN uses logic-based comparators to decode spike counts in one clock cycle, latching the winning class register, as depicted in figure 3.15.

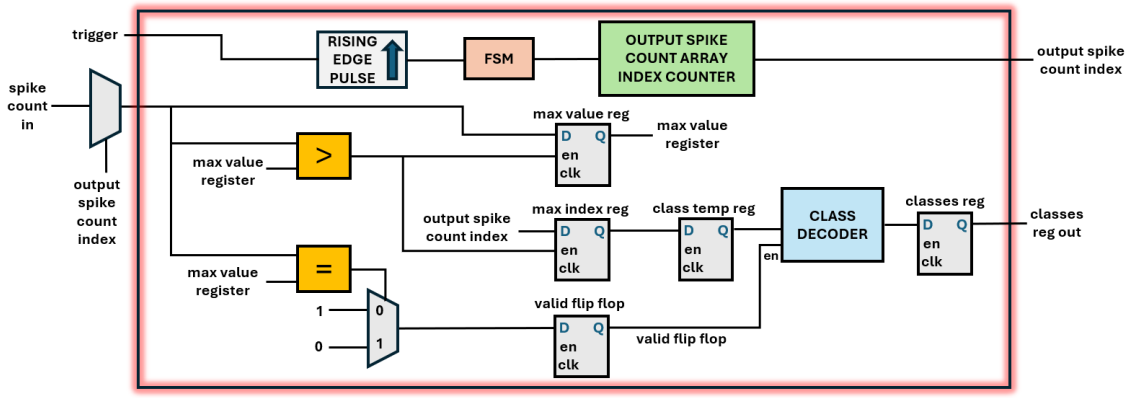



***Figure 3.14*** *– Output spike decoding component for SNN.*

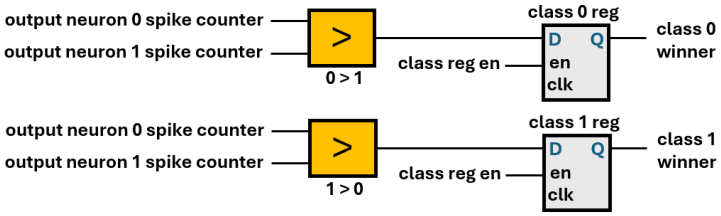



***Figure 3.15*** *– Output spike decoding component for Fast SNN.*

### 3.2.4 Control FSM

The control FSM implements the basic orchestration of the smart watchdog.

1) The FSM remains idle until RISC-V instruction data is written into the FIFO.
2) Provided the watchdog is enabled, the FSM ensures valid reads data from the FIFO to the feature layer until an instruction has been executed.
3) The 16 binary features from the feature layer are latched as input data to the SNN and the SNN ready status is checked. If ready, the FSM triggers the SNN.
4) The FSM and feature layer continues extracting features from the remaining RISC-V instruction data as described in **2)**, while the SNN is classifying the previous instruction.
5) Once all RISC-V instruction data has been processed (i.e. FIFO is empty), the FSM resets the feature layer and finish off this…

**Note**: The control FSM for the demonstrator in this submission contains additional state logic necessary to communicate the RISC-V instruction data, extracted features and the SNN class result with the Python GUI for visualisation.

# 4. Demonstrator Design & Implementation

This section describes the ISCAS demonstrator, deploying the RISC-V CPU to execute realistic motor control software, and the smart watchdog is instantiated to monitor control flow.

In the physical demonstrator setup (figure 4.1), a 24V DC motor is controlled by a Cytron H-bridge PWM motor driver [31]. A Nidec 1,000 pulses-per-revolution quadrature rotary encoder [32] is directly coupled to the motor to provide speed feedback. Its signals are conditioned and level-shifted by a custom motor encoder circuit PCB. The custom motor control circuit PCB provides buttons, a switch and a rotary encoder used to start/stop, change direction and speed setpoint of the motor respectively, and three LEDs indicate the running and direction status of the motor. Actual motor speed can be verified by a handheld digital tachometer [33] by measuring motor shaft rotation via the reflective tape. The two 7-segment displays on the Nexys A7-100T development board [34] show the setpoint and actual motor speed in RPM.

The final custom fault injection control and smart watchdog monitoring PCB allows various faults (bit flips and stuck at 0/1) to be setup and injected on bits 1 to 10 of the RISC-V PC to potentially cause control flow errors. The smart watchdog fault detection response can be observed with the green and red LEDs (normal execution and fault detected, respectively). Two additional switches reset the RISC-V processor to a known state after fault injection and allow the smart watchdog to be disabled. The final component is an AMD Artix-7 FPGA [35] that implements the smart watchdog, RISC-V core (Neorv32) and other required logic. All FPGA designs are written in VHDL using AMDs Vivado toolchain [36] and clocked at 100MHz. The FPGA hardware is the key innovation of the design and is detailed in the next section.

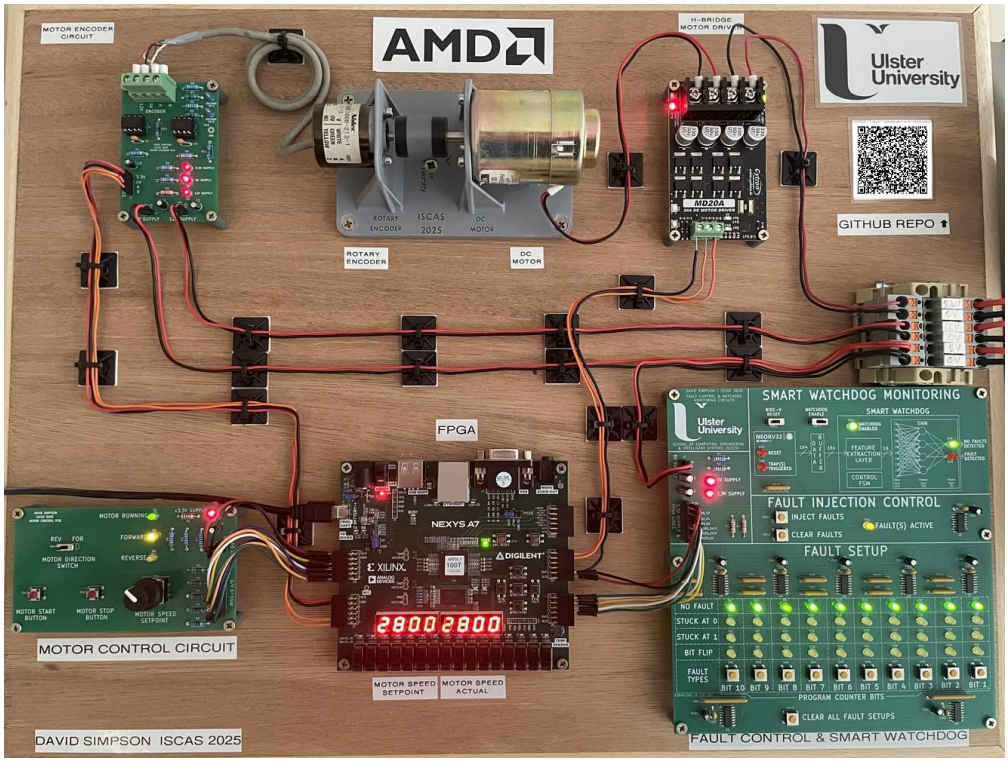



***Figure 4.1*** *– Physical hardware setup of demonstrator.*

## 4.1 Hardware Implementation

This section details the FPGA design (VHDL) at component levels. A simplified view of the demonstrator architecture is shown in Figure 4.2. The hardware has core three functions: **motor control**, **fault injection control**, and **smart watchdog monitoring**.

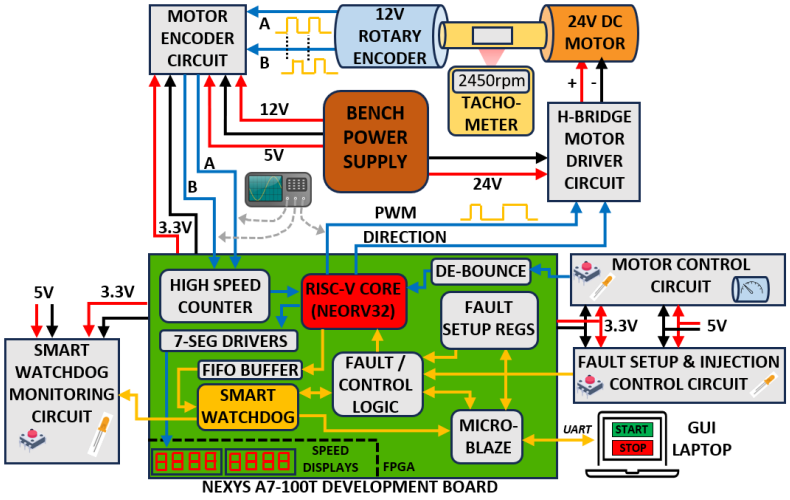



***Figure 4.2*** *– Block diagram of the hardware design.*

### 4.1.1 Motor Control

The motor control task aims to create a realistic and complex workload for the RISC-V processor and is designed independently of the smart watchdog, highlighting its non-invasiveness. FPGA parallelism is leveraged by dedicating logic to handle input de-bouncing and multiplexing logic for the two on-board 7-segment displays used for motor speed indication.

#### 4.1.1.1 Neorv32 (RISC-V CPU)

Neorv32 is configured as a complete micro-controller platform with GPIO, timer, PWM and interrupt peripherals enabled or the task (figure 4.3). Motor control commands are received via general purpose inputs from both the motor control PCB and the Python GUI, along with a 32-bit counter to count encoder pulses from motor rotation. General purpose outputs write to a motor control/status register and two 32-bit registers containing the calculated motor setpoint and actual speeds (in RPM and BCD format). The PWM is outputted directly from Neorv32.

Modifications made to Neorv32 include routing the selected instruction data registers to the FIFO for the smart watchdog as previously discussed, and implementing 4-to-1 multiplexor based fault injection logic (FIL) to inject faults at bits 1 to 10 of the PC register (figure 4.3). The multiplexers are controlled via 20-bit fault injection select lines, which are detailed in Section 2.1.3 – Fault Injection Control.

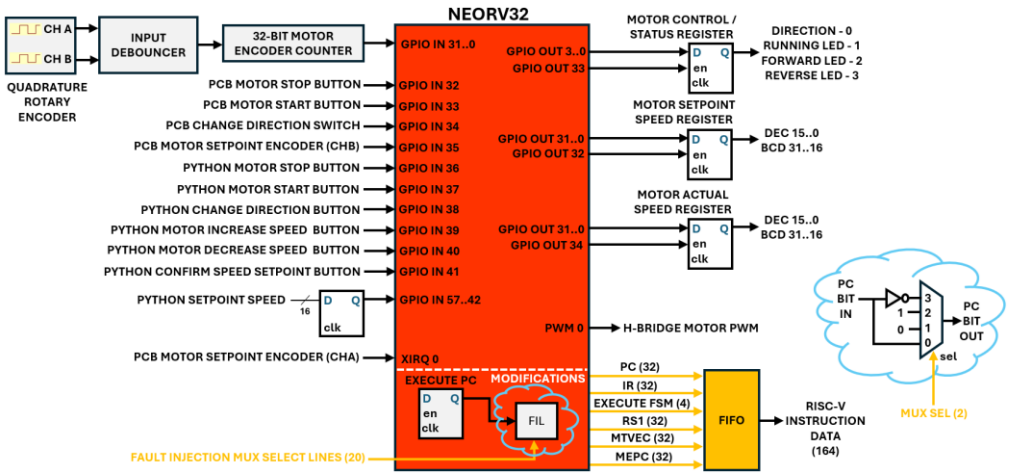



***Figure 4.3*** *– Block diagram of Neorv32 for the motor control application.*

#### 4.1.1.2 7-Segment Display Controller

The 32-bit BCD motor speeds register stores the setpoint speed in the upper 16 bits and the actual speed in the lower 16 bits, with each 16-bit half arranged as four 4-bit nibbles representing the corresponding digits. A 7-segment display controller handles BCD decoding and digit multiplexing, refreshing each digit every 1 ms (125 Hz), as shown in figure 4.4.

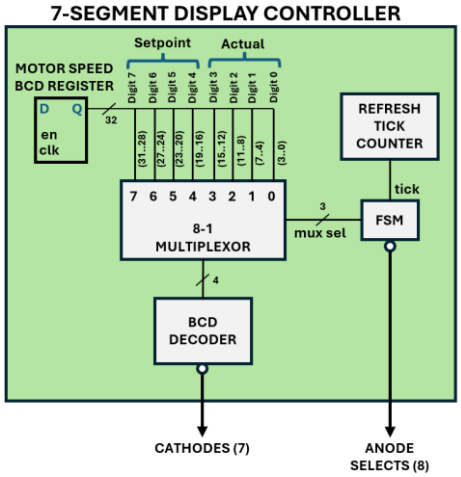



***Figure 4.4*** *- Block diagram of the 7-segment display controller component.*

### 4.1.2 Fault Injection Control

Faults are setup and injected into the Neorv32 PC using either the buttons on the fault injection control PCB or virtual buttons on the Python GUI. The process has two hardware stages: **setup** and **injection**, with each handled separately as described in the following sub-sections.

#### 4.1.2.1 Fault Setup

This demonstrator permits injection of three fault types into the targeted bits 1 to 10 of the PC, i.e. stuck at 0 and stuck at 1 (to model silicon failures), and bit flips (to model SEUs). Each bit's fault type is implemented with a 2-bit synchronous counter, incremented by a button press from the PCB or the Python GUI, allowing the fault types to be cycled through. Counter values are used to update the Python GUI, and a 2-to-4 decoder generates the corresponding LED indication on the PCB, as illustrated in figure 4.5. The *"clear all faults"* command resets all fault type counters through a PCB or GUI button press.

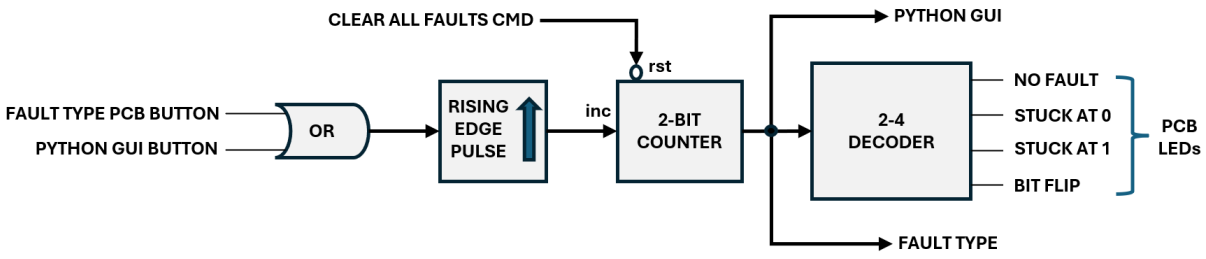



***Figure 4.5*** *– Block diagram of the fault type setup logic.*

#### 4.1.2.2 Fault Injection

The fault control component handles injection of faults (figure 4.6). For simplicity, the sequence of injecting faults is described in numbered stages below:

1) The FSM awaits the ***"inject fault(s)"*** command from the PCB button press or the Python GUI. Provided the watchdog is ready and the FIFO empty, fault injection is triggered.
2) The FSM enables the FIFO write counter, and the FIFO write enable line, allowing RISC-V instruction data to begin buffering in the FIFO on every subsequent clock cycle. A short delay of 16 clock cycles is also initiated to allow the FIFO to fill with normal RISC-V data before any faults are physically injected.
3) After the 16 cycle delay, the FSM enables the fault controllers for bits 1 to 10, and any setup faults will be injected during the write stage of Neorv32 PC. This is to ensure that faults are injected at a critical time to cause CFEs for demonstration purposes. Triggering fault injection with no faults setup allows for visualising the smart watchdog behaviour under normal RISC-V executing conditions.
4) Once the specified amount of RISC-V instruction data is stored in the FIFO (i.e. ***FIFO write counter >= FIFO write cycles***), the FIFO write enable is disabled and no more RISC-V instruction data is written to the FIFO.
5) Finally the FSM awaits the ***"clear fault(s)"*** command from the PCB button press or the Python GUI, and the FIFO write counter and FIFO pointers are reset, ready to repeat.

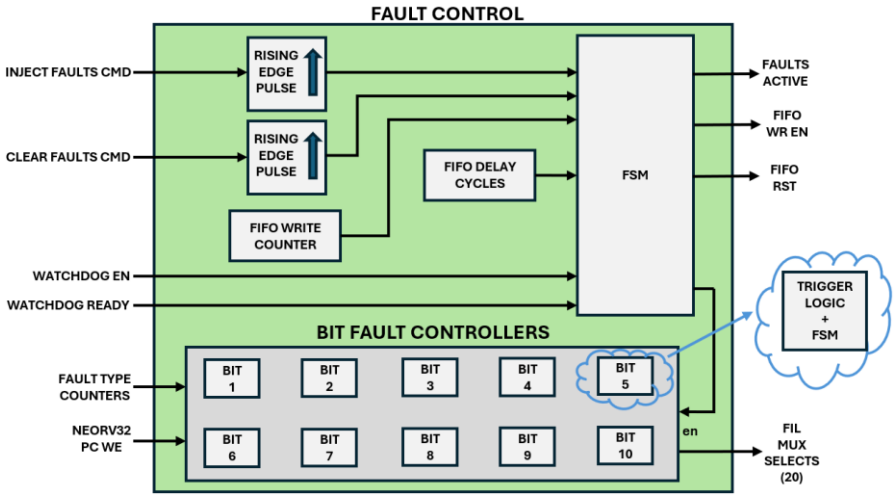



***Figure 4.6*** *– Block diagram of the fault control component.*

#### 4.1.2.3 PCB I/O Controller

Given the large number of I/Os (15 buttons and 46 LEDs) on the fault injection control and smart watchdog monitoring PCB, all inputs and outputs are routed through shift register ICs: parallel-to-serial (74HC165) for buttons and serial-to-parallel (74AHCT595) for LEDs. A simple serial I/O controller on the FPGA manages read and write sequences, as simplified in figure 4.7. A delay counter triggers the serial input control component every 25ms handling debouncing internally, after which, the serial output control is triggered and refreshed.

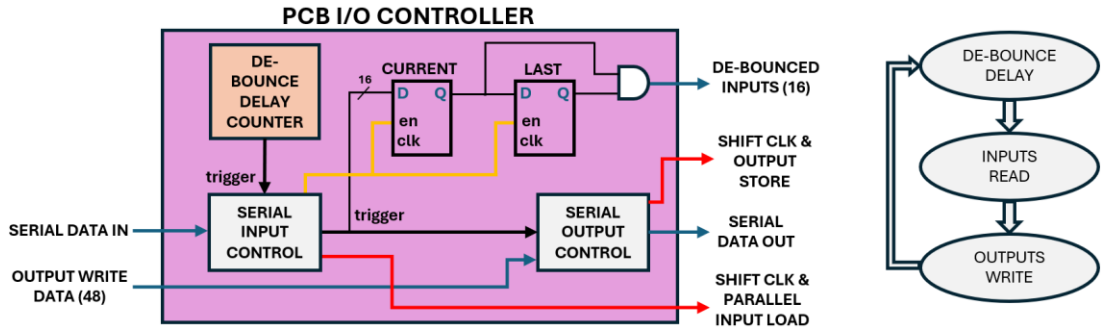



***Figure 4.7*** *– Block diagram PCB I/O controller component with functional flowchart.*

### 4.1.3 Packaging Hardware as IP

The Python GUI provides enhanced usability and powerful data visualisation of inside the FPGA architecture. Previous sections have discussed the motor control application, fault setup and injection, and the hardware implementation of the smart watchdog. To simplify integration between the RTL architecture and the GUI, the design is packaged into a modular IP block with an AXI4-Lite front end interface, enabling a straightforward AXI peripheral software approach. Communication between hardware and software via a 32-bit MicroBlaze softcore processor with an AXI-Uartlite peripheral (230,400 baud rate), provides a flexible and reliable transmission link. Figure 4.8 from Vivado shows the hardware IP instantiated alongside the MicroBlaze and other necessary components. The following section details the Python GUI software, the MicroBlaze embedded C code, and the Neorv32 C code used for the motor control task.

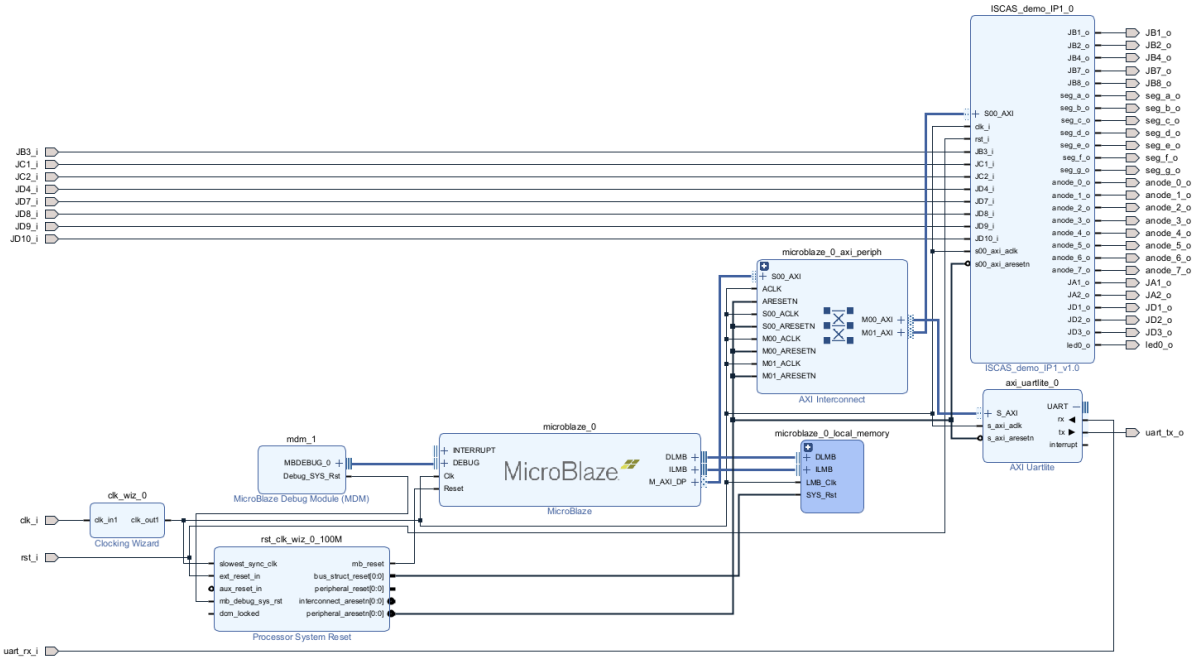



***Figure 4.8*** *– Vivado hardware design block diagram.*

### 4.2 Software Implementation

There are three main softwares running in this demonstrator all written from scratch by the PhD student; C software on the **RISC-V** and the **Microblaze** softcores, and the **Python GUI** script running locally on a computer.

#### 4.2.1 RISC-V Embedded C Software

The Neorv32 software framework is utilised to control the motor speed and direction based on rotary encoder feedback and user input commands from the motor control PCB and Python GUI. During initialisation, the software configures GPIO, PWM, and external interrupt peripherals, and enables timer interrupts at 1ms intervals. The main code structure is organised into five procedures:

**1. Reading inputs**: Every 1ms, the 64-bit GPIO input port is read. Button presses are detected on rising edges by comparing with the previous read state. A logical OR operation is required where multiple instances of buttons exist, e.g. PCB motor start and Python GUI motor start button. Setpoint speed parameters are range checked, with values capped between 2,000rpm and 4,000rpm and modified in 50rpm steps.

**2. Motor control:** A FSM cycles between motor idle, motor forward, motor reverse and motor stop states, transitioning based on inputs and current state.

**3. Motor speed calculation:** All rotary encoder edges are counted in hardware. Every 1ms, the actual motor speed is calculated by reading the encoder counter connected to GPIO and computed as shown in equation 4.1.

$$\frac{(motor\ encoder\ count - last\ motor\ encoder\ count)\ \times \left(\frac{60}{1 \times 10^{-3}}\right)}{ENCODER\ PPR} \tag{4.1}$$

**4. PI control**: Speed is regulated using a Proportional-Integral (PI) controller with a setpoint-relative error. Integral ranges are checked to prevent anti-windup and the limited PI output is scaled to an 8-bit PWM duty cycle according to equation 4.2.

$$\left(\frac{(PI\ output - \min RPM\ error)\ \times (\max PWM\ duty\ cycle - \min PWM\ duty\ cycle)}{(\max RPM\ error - \min RPM\ error) + \min PWM\ duty\ cycle}\right) \tag{4.2}$$

**5. BCD algorithm:** A digit-by-digit repeated subtraction algorithm converts the setpoint and actual motor speeds into BCD for display. For a stable display, the actual motor speed is filtered by a running average over 333ms before conversion to BCD.

**Note**, in the event of traps caused by injected hardware-faults (exceptions), the software does not provide any correction other than incrementing the PC by 4. Trap triggers are highlighted on the demonstrator for users to observe the RISC-V fault detection mechanism response.

### 4.2.2 Microblaze Embedded C Software

The MicroBlaze (μB) CPU [37] is programmed in C using the Vitis IDE and framework [38], with the exported Vivado bitstream. The μB handles the two-way communication between the FPGA hardware IP and the Python GUI.

**Python GUI → Hardware**: The μB polls the UartLite receive FIFO, reading bytes from the Python GUI into a buffer until a newline character denotes the end of a message. The string is then null-terminated and compared against valid command identifiers.

- Most commands from the Python GUI indicate a command identifier (e.g. motor_start) and a state (e.g. pressed or released). Based on the command received, the μB sets or clears the corresponding bit in the corresponding AXI slave register (figure 4.9a), allowing the hardware IP act accordingly (e.g. motor start button pressed).

- Other commands include an identifier with a numeric parameter, such as setting the motor setpoint on the Python GUI (figure 4.9b).

"**motor_start_p\n**" → Set AXI reg bit

"**motor_start_r\n**" → Clear AXI reg bit

(a)

"**motor_setpoint_3000\n**" → Write parsed setpoint to AXI reg

(b)

***Figure 4.9*** *– Example UART messages from the Python GUI to the MicroBlaze: (a) Python GUI motor start button example. (b) Python GUI motor setpoint speed example.*

**Hardware → Python GUI:** The μB polls the AXI slave registers cyclically to track motor speeds, fault type counters, fifo write cycles and other data relating to motor status and LEDs. To minimise UART transmission, only changed values are sent to the GUI. The μB also checks the smart watchdog *"ready/done"* status bits via AXI slave registers. Once the watchdog begins monitoring RISC-V execution, i.e. instruction data in FIFO from *"inject faults" command*, the μB enters a dedicated loop to extract and stream the watchdog data off the FPGA to the Python GUI for visualization.

- **Instruction info disabled**: The μB counts the number of instructions classified by the smart watchdog and tracks the classification results (i.e. no faults and faults detected). When the watchdog monitoring is complete, the counters are sent to the Python GUI.

- **Instruction info enabled**: More detailed information is streamed from the FPGA in real time. For each instruction classified, the μB handshakes with the watchdog control FSM to read the RISC-V instruction data, extracted features and SNN classification result via AXI slave registers and sends the data over UART to the Python GUI.

- **Spike plot enabled**: When enabled in addition to instruction info, the μB also reads spiking activity captured in shift registers during inference, making it available to the GUI for spike raster visualisation.

### 4.2.3 Python GUI Software

This Python GUI was developed from scratch by the PhD student and utilises common libraries such as serial, matplotlib, tkinter. Its primary purpose is as a interaction and data visualization tool for the demonstrator. Functionally, the GUI manages UART communication with the µB, processes received messages, and runs the GUI on a separate thread. This section focuses on the GUI's behaviour and role within the system, rather than its internal software design. The GUI (figure 4.10) is divided visually into three main areas; **motor control**, **fault control** and **smart watchdog monitoring**, reflecting the physical setup.

- **Motor Control:** The GUI replicates the motor control PCB functionality in parallel, allowing the user to start, stop, change direction, and adjust motor speed. Additional functionality enables the direct entry of motor setpoints in 50 rpm increments. The interface includes graphical LED indicators for motor status and text displays for setpoint and measured speeds.

- **Fault Control:** Fault setup and injection functionality are functionally identical to the PCB on the physical setup, providing a parallel interface for configuring and applying faults through the GUI.

- **Smart Watchdog Monitoring:** This area is the most valuable addition of the GUI, enabling detailed visualization of smart watchdog behaviour. While the PCB offers only a high-level LED indication of whether a CFE was detected, the GUI provides much more detail.

    - **Instruction info disabled:** The total number of instructions monitored is displayed, including a breakdown of normal execution and CFEs detected.
    - **Instruction info enabled**: The GUI also now displays the RISC-V instruction data, extracted features, and the SNN inference classification for every monitored instruction. Users can select individual instructions for analysis, revealing their opcode, the last PC, current PC, and the correct PC update to validate watchdog results.
    - **Spike plot enabled**: The GUI additionally plots spiking activity captured from the SNN hardware during inference, offering a detailed view of how the smart watchdog model processes real execution data.

A key strength of this demonstration lies not only in deploying the smart watchdog on hardware to monitor and classify real-world RISC-V instruction data from a space-qualified CPU, but also in the GUI's ability to reveal the FPGA's internal behaviour at a fine level of detail, providing users with an environment for observing and understanding the watchdog's operation.

**Note**, while the µB and Python GUI software successfully achieve the desired functionality, it could be further optimized for scalability and maintainability. For example, replacing large if–else decision trees with hash tables or dictionaries.

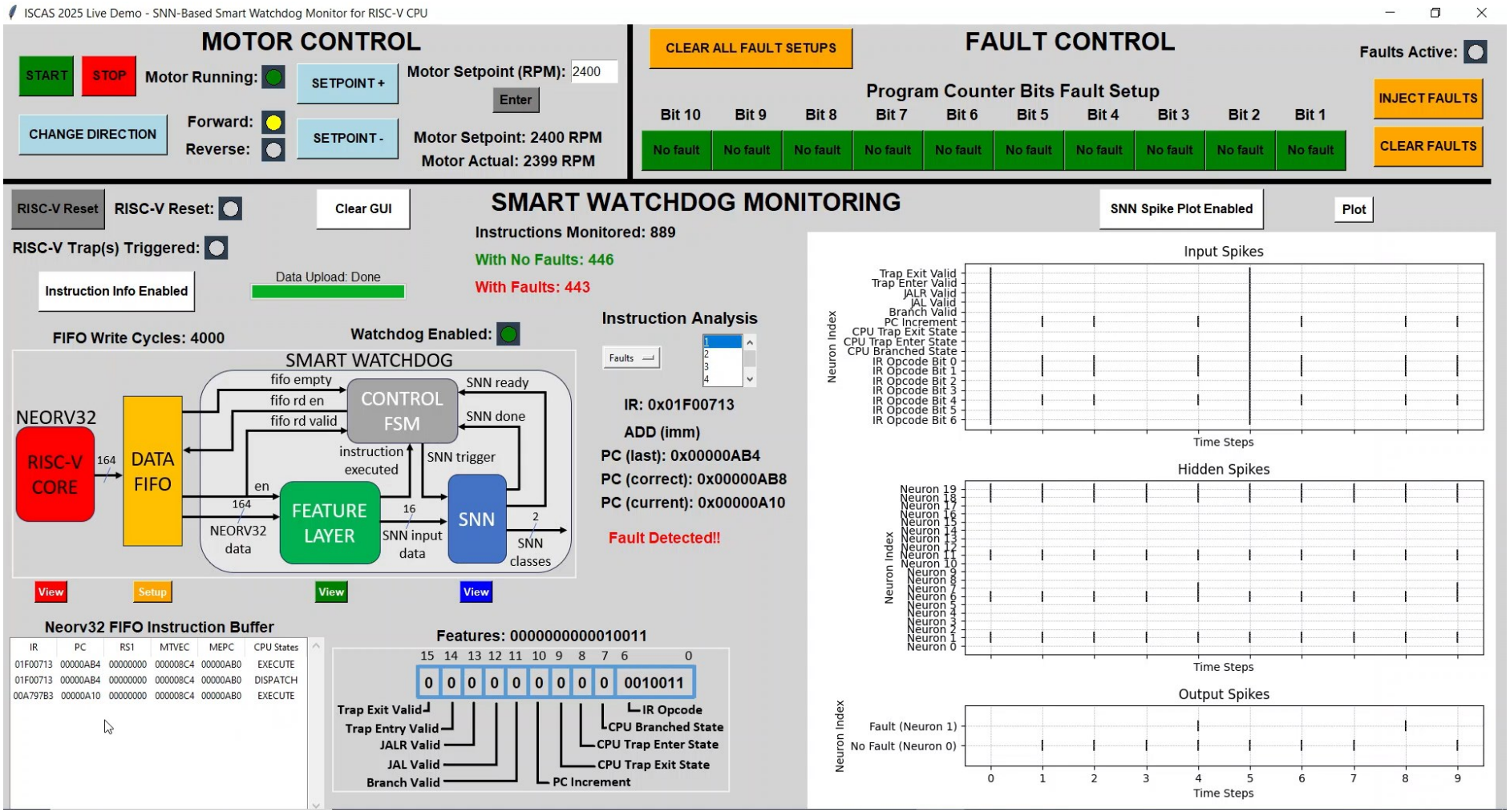



***Figure 4.10*** – *Python GUI.*

# 5. Results

The demonstrator effectively showcases the smart watchdogs real-time monitoring and fault detection capability, through powerful data visualization. To complement this report, an additional YouTube video is provided to give a narrated walkthrough of the demonstrator, describing motor control, fault injection, and smart watchdog monitoring in detail: ISCAS 2025 Live Demo - Smart Watchdog Mechanism for Real time Fault Detection in RISC-V.

The following sub-sections highlight the reusability of the PhD work, and present the fault detection and hardware implementation results of the smart watchdog. A detailed discussion then explores the challenges of deploying the smart watchdog hardware model in its current form. The report then concludes with a summary of findings and an outline of future work.

## 5.1 Design Reuse

Re-usability was a core design objective of the smart watchdog. In this PhD, the watchdog was trained to learn control flow of a modern embedded processor (RISC-V), based solely on spiking patterns of extracted instruction features. This enables any C software to be compiled and executed on the RISC-V processor without requiring any modifications to the smart watchdog component, as it operates at the instruction level independently of application software. This independence is one of the watchdog's key functional benefits, and the demonstrator highlights the design's inherent reusability. In practice, the smart watchdog component was instantiated alongside the Neorv32 while motor control software was compiled and executed. And although not yet verified, the smart watchdog, particularly the SNN model, is expected to be directly applicable to other RISC-V processors.

Design reuse extends beyond the watchdog itself. The methodology developed in this PhD (Figure 2.1) can be applied to create new smart watchdogs targeting other circuits (e.g. instruction fetch), leveraging the same hardware design (SNN components) and software stack (SNNTorch training scripts). Similarly, the modular design of the fault setup and injection framework enables it to be adapted for other hardware targets. The data collection framework, built around the MicroBlaze and UART interface, can also be reused for training data acquisition, with the option of substituting UART for a faster protocol such as Ethernet and exploiting on-board DDR for larger datasets.

Finally the generic, time-multiplexing SNN hardware model should be re-usable for other non-watchdog tasks by replacing the fixed spike encoder with a task-specific encoder while reusing the same software-to-hardware deployment framework (figure 3.5). A final point is that the smart watchdog has been deployed on two different AMD FPGA platforms i.e. Artix-7 and Virtex-7, demonstrating device portability to an extent.

**Open-Source**

There are three main contributions in this design submission, found in the Github repository:

1) **Demonstrator:** All HDL source code for the complete smart watchdog component is shared including both SNN implementations (***time-mux SNN*** and ***Fast SNN***). The demonstrator has build instructions, although can't be fully replicated without PCBs.

2) **Data Collection:** All HDL source code for the data collection framework including the µB C software and fault injection hardware is shared, allowing others to re-use it for data collection in RISC-V with build instructions.

3) **Model SW to HW:** The framework for porting software models trained using SNNTorch to FPGA hardware is also included, allowing it to be re-used for other trained SNNTorch models. Build instructions are also provided.

## 5.2 Fault Detection Results

For completeness in this report, the hardware validation results from stage 8 – smart watchdog validation, of the 8 stage methodology (figure 2.1) are briefly summarised. A heap sort C algorithm was executed on Neorv32 while faults were injected into the PC. The smart watchdog performed real-time, in-circuit classification of RISC-V instruction data and results were extracted off-FPGA using the MicroBlaze, UART and AXI interface approach as used in the demonstrator. The smart watchdog classification results were evaluated in Python, and **~2.4 million instructions** during 1,000 heap sort executions were monitored, with only **two** misclassifications. A binary feature dataset was created producing 100 unique, recurring features with label 0 and 1 breakdowns of 52 and 48. Results are shown in Table 5.1. Overall the SNN classified **98%** of RISC-V instruction data correctly. More specifically, the **100% precision** shows that when the smart watchdog signals a CFE, it was always correct. While the **96% recall** shows two CFEs instances were mistaken as normal execution. This performance metrics highlights an advantageous trait of the smart watchdog. All negative samples (i.e. normal CPU execution) were classified correctly. If the SNN failed to recognise normal CPU execution, it would face an immediate barrier to practical deployment, as incorrect classifications would undermine the model's validity and waste precious execution time in correcting states that are, in reality, fault-free.

***Table 5.1*** *– Smart watchdog hardware validation results.*

| Samples | Correct | TP | TN | FN | FP |
|---|---|---|---|---|---|
| 100 | 98 | 46 | 52 | 2 | 0 |
| **Accuracy** | | **Precision** | **Recall** | **F1 Score** | |
| 0.98 | | 1.00 | 0.96 | 0.98 | |

Given modern concerns regarding SDC in processors, additional results (table 5.2) are presented to evaluate the fault detection benefit of integrating the smart watchdog with the RISC-V processor. Across the 1,000 heap sort executions, 840 of the applications contained at least one CFE from injected faults. 490 (58.3%) of these applications raised a trap in Neorv32, with the remaining 350 application executions being corrupted unknowingly, potentially resulting in SDC. The smart watchdog detected 100% of these silent CFEs that failed to raise a trap in the RISC-V CPU, highlighting a significant benefit in fault coverage.

***Table 5.2*** *– Smart watchdog silent CFE detection results.*

| Total Applications | Applications with CFEs | Applications with Trap(s) | Smart Watchdog CFE Detections |
|---|---|---|---|
| 1,000 | 840 | 490 (58.3%) | 350 (100%) |

## 5.3 Hardware Implementation Results

Resource utilization and power estimates from Vivado (table 5.3) are presented for the key components discussed in the report. Typical parameters for the base RISC-V core (Neorv32) alongside the demonstrator instantiation with peripherals enabled (GPIO, PWM, timer, and interrupts) allow for evaluating the hardware overheads incurred from the smart watchdog. The feature layer is negligible, and the first observation is that for both implementations, the SNN is the significant part of the smart watchdog. The second observation is that the time-multiplexing SNN is comparable with Neorv32 in terms of power and resources with the

addition of the weights memory (implemented as DRAM). The final observation is that the Fast SNN requires significantly more power and resources than Neorv32, as expected.

***Table 5.3** – Hardware synthesis results.*

| Component | FF's | LUT's | LUTRAM | BRAM | Power (W) |
|---|---|---|---|---|---|
| **Neorv32 (base)** | 1,993 | 1,874 | 24 | 6 | 0.027 |
| **Neorv32 (peripherals)** | 2,336 | 2,183 | 24 | 8 | 0.032 |
| **Feature Layer** | 149 | 157 | 0 | 0 | 0.001 |
| **Time-mux SNN** | 1,791 | 1,953 | 500 | 0 | 0.037 |
| **Fast SNN** | 9,308 | 6,635 | 0 | 0 | 0.084 |
| **Smart Watchdog (Time-mux SNN)** | 1,944 | 2,119 | 500 | 0 | 0.038 |
| **Smart Watchdog (Fast SNN)** | 9,469 | 6,817 | 0 | 0 | 0.085 |

***Table 5.4** – Throughput comparison between SNN hardware implementations.*

| SNN Model | Clock Cycles | Max Frequency | Latency |
|---|---|---|---|
| **Time-mux SNN** | ~ 1,433 | 100MHz[1] | ~14.22µs[2] |
| **Fast SNN** | 153 | 350MHz | ~438ns |

[1] The time-mux SNN was not optimized for timing, therefore max frequency was not determined.

[2] Latency is therefore calculated at the default frequency of 100MHz.

**Note**, the estimated power consumption of the smart watchdog with the Fast SNN in the demonstrator is lower than estimated during the PhD validation stage (0.085W compared to **0.143W**), despite a similar resource utilization. Variations in power estimates could potentially be attributed to differences in FPGA platforms (Artix-7 vs Virtex-7), project configurations in Vivado, and modifications introduced in the demonstrator, all of which may influence routing and related factors in the power estimation process.

The Fast SNN was specifically optimized for throughput by pipelining, dedicated hardware for weight accumulation (adder trees) and register stages to accommodate higher clock speeds. As highlighted in Table 5.4, these optimizations yielded significant performance improvements compared to the time-multiplexed SNN, which was designed primarily for generality. The number of clock cycles required for a single inference was reduced by almost 10x, to just 153 cycles. Moreover, the Fast SNN meets timing closure at a max frequency of 350MHz, enabling a total inference latency of just under 438ns including spike encoding and decoding.

## 5.4 Discussion

Smart watchdogs and in particular, SNNs, have proven highly effective at fault detection in the RISC-V architecture when integrated with an actual, in-silicon CPU. However the real-world feasibility of the smart watchdog model depends on more than just classification accuracy, as safety-critical embedded designs often face stringent power and area constraints. Hardware models must therefore be optimized for efficiency, silicon-area and latency to be considered as a watchdog mechanism. Michaela Blott emphasised during the eFutures Edge AI 2024 event at AMD, Citywest, Dublin, that designers should "never stop optimizing" their models.

### 5.4.1 Power and Area

The SNN is the significant hardware component of the smart watchdog and has been estimated to require more power and resources than the RISC-V processor. To truly realise efficient SNNs, specialised custom analog or neuromorphic hardware are required to fully leverage sparsity and event-driven, low power operation which FPGA can't achieve. Research has found that LIF neurons implemented in analog resulted in 5×area and 20×power

reductions compared to digital implementations [39]. However exploring optimisations through FPGA prototyping can have a direct carry over to ASIC.

To reduce the SNN power and area footprint, bit quantisation must be applied. Currently, neuron membrane potentials represented with 24 bits, which stemmed from a challenge encountered during implementation where the hardware neuron outputs did not match the software model for the same input data. An analysis of neuron membrane potentials stored during model training revealed that certain neurons experience extreme hyperpolarisation and depolarisation for specific input data, causing the hardware neuron membrane potential register to wrap around and disrupt thresholding conditions. With this knowledge, limiting the range of neuron's membrane potential in hardware would enable the use of a smaller bit width without degrading performance, providing significant power and area savings.

Weights and biases are currently represented with 24 bits, and is likely excessive. Research indicates that SNNs can achieve comparable performance with significantly fewer bits, often as low as 12 or less, resulting in substantial power and area reductions, particularly in the Fast SNN's adder tree. Additionally, some hidden neurons remain in a hyperpolarized state and rarely fire. These neurons contribute little to inference, yet still consume resources and power every clock cycle and could therefore be pruned or removed entirely. As neurons with adder tree synapses are computationally expensive, pruning would provide significant savings. Interestingly, training data also shows that output neurons require less bit-width than hidden neurons as membrane potentials remain tighter. Beyond hardware, longer inference times (i.e. model latency), requires more clock cycles and thus higher computational cost.

### 5.4.2 Latency

In this watchdog monitoring task, inference time is important, as excessive latency can render the model unusable. Latency, power and area are all closely correlated, as highlighted by the integration of adder trees in the Fast SNN – latency is reduced at the cost of efficiency. Despite the optimisations, the current hardware model still faces latency challenged (figure 5.1a). Instruction data from the RISC-V is written to the FIFO every clock cycle at 100MHz, with the minimum interval between instructions requiring control-flow checking on Neorv32 ranging from 30 ns (3 clock cycles), up to 80 ns (8 clock cycles) for memory accesses or taken branch instructions. Given that the Fast SNN requires 438 ns for inference, further latency reductions are necessary to prevent the FIFO buffer from eventually becoming full.

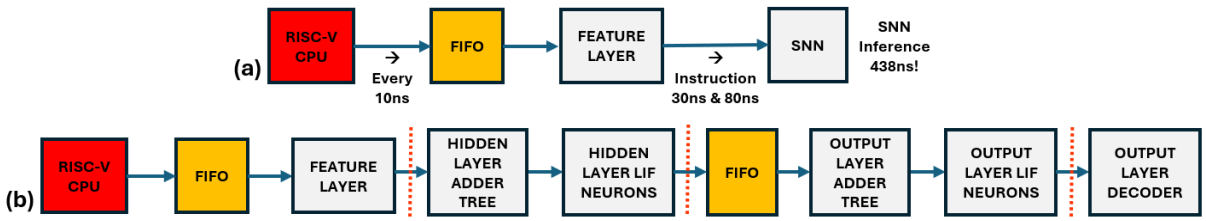



***Figure 5.1** – SNN full pipelining optimisation:*
*(a) current SNN coupled pipeline. (b) proposed decoupled SNN pipeline.*

The slowest part of the SNN is weight accumulation. In the current smart watchdog, the critical path is the carry propagation in the pipelined adder tree. Quantizing weights and biases would reduce propagation delays and could effectively eliminate this critical path. Further acceleration could be achieved by replacing the standard adder trees with specialized versions such as Wallace or Carry-Save Adder (CSA) trees. Offloading the SNN to dedicated AI platforms such as AMD's Versal AI engines with DSP58 primitives and parallelism [40], would also provide significant speedups.

A closer examination of the current design reveals that the throughput of the adder tree pipeline is not fully utilized. For each timestep in the LIF layers, weight propagation requires seven clock cycles to reach the adder tree output, resulting in a total of 70 cycles for ten timesteps. Instead, by processing all input spikes for the ten timesteps in a single burst, weight accumulation could be reduced to only 17 cycles for the same workload. Further, by decoupling the layers and fully pipelining the SNN, once the pipeline is filled, inference time could be reduced from approximately 150 cycles to the longest pipeline stage (figure 5.1b).

Ultimately, however, to fully leverage the efficiency of SNNs, the focus is on neuromorphic implementations, with FPGA serving primarily as a prototyping proof-of-concept platform. With that in mind, the number of timesteps ultimately determines inference latency. Research has shown that in certain cases, only a single timestep is sufficient for effective inference [41]. Re-training the SNN software model in SNNTorch with fewer timesteps, or even reducing to just one timestep per instruction could substantially reduce latency and power consumption without potentially affecting classification accuracy. This would also provide additional headroom for running the monitored CPU at faster clock speeds.

## 5.5 Conclusion and Future Work

This PhD project proposed a novel and innovative concept of leveraging brain-inspired SNNs for more modern fault detection paradigms, aspiring towards more efficient, intelligent and robust watchdog mechanisms. To explore proof of concept, complete ML workflow had to be developed from scratch and incrementally followed. A comprehensive FPGA-based fault injection and data collection platform with a RISC-V processor was developed to collect training data, and a Python framework put in place to preprocess datasets, extract features and train a SNN software model. Thereafter a hardware model of the complete smart watchdog component was validated in hardware for real-time monitoring of a RISC-V CPU and a demonstrator was created to showcase the effective fault detection capability of the smart watchdog, which was the focus of this report. A final feasibility discussed the performance and challenges of integrating the smart watchdog with an embedded RISC-V processor.

Despite the depth of this PhD project, it lays the foundation for a wide range of future work. Investigating the hardware optimisations such as bit quantization and more effective adder trees utilisation to address silicon and latency limitations. Additionally, looking into temporal spike encoding over rate-based schemes could further improve efficiency through sparsity although noise tolerances could introduce reliability concerns. Exploring integration of astrocyte functionality into the SNN to then realise a model that can internally detect and self-repair faults to ensure reliable monitoring. Finally expanding the development of smart watchdogs to harden other vital circuits like instruction fetch, and verifying generalization to RISC-V and other architectures such as ARM could transform watchdog mechanisms for safety-critical applications where dependability is paramount.

## 6. Data Collection

A significant milestone in the PhD project was the creation of a custom framework that enabled live data collection from a RISC-V softcore (Neorv32). Given the importance of access to training data for developing the SNN, a succinct description of the hardware architecture and software framework is provided in this section. Majority of the hardware in this initial design was re-used by the demonstrator and has already been discussed in detail.

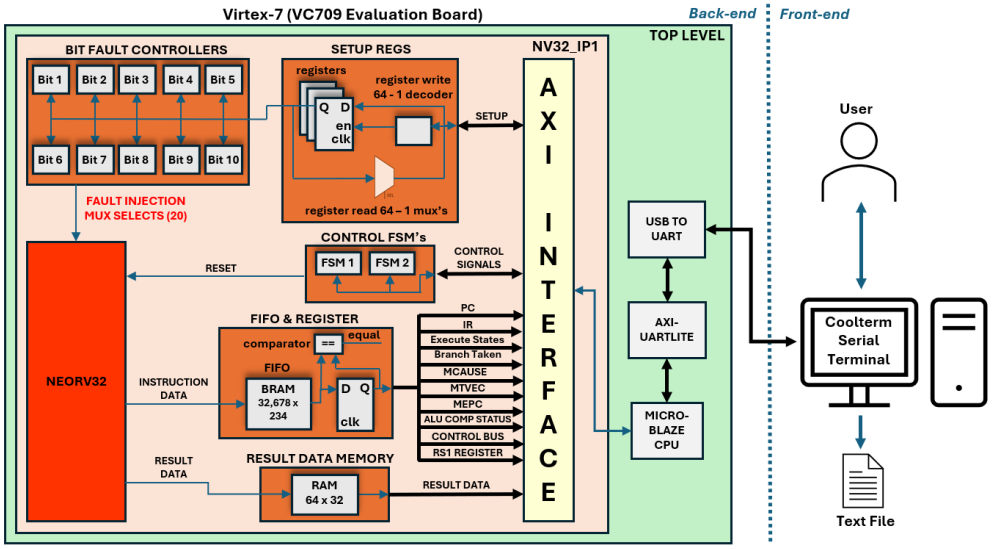



***Figure 6.1*** *– Data collection hardware architecture block diagram.*

The hardware is placed in an IP (***nv32_ip1***) with an AXI interface with the ***µB*** as the communication link. Principally as previously described, the ***RISC-V CPU*** executes the compiled C software while faults are injected into the PC. During execution, the RISC-V instruction data is stored in a BRAM ***FIFO buffer*** every clock cycle. Additionally the result of the C application, i.e. a sorting algorithm or matrix multiplication, is stored in a ***result data RAM***. Capturing the result data enables the effects of the injected faults on the C to be evaluated.

After the RISC-V runs the C application software for a specified number of clock cycles, the processor is then held in reset while the ***µB*** accesses and streams both the RISC-V instruction data and result data off-FPGA to a listening serial terminal over UART. To configure parameters relating to fault injection are configured by the µB writing and reading to the ***setup registers***, which are used by the bit fault controllers and the ***control FSMs***.

The C software running on the ***µB*** orchestrates the hardware IP to repeat each RISC-V C application multiple times with different fault configurations, extracting the RISC-V instruction data and result data after each single application run. The framework supports single and multiple fault injection and can be used to target other areas of the RISC-V architecture with minor hardware and software modifications.

This is data collection framework aims to be published, including an analysis on the RISC-V result data from fault injection. This data collection framework is also included as part of the submission due to it being a crucial stepping stone to developing the SNN model of the smart watchdog deployed in the demonstrator.

## References


[1] J. Li, S. Zhang, and C. Bao, “DuckCore: A Fault-Tolerant Processor Core Architecture Based on the RISC-V ISA,” *Electronics (Basel)*, vol. 11, no. 1, p. 122, Dec. 2021, doi: 10.3390/electronics11010122.

[2] P. H. Hochschild *et al.*, “Cores that don’t count,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, New York, NY, USA: ACM, Jun. 2021, pp. 9–16. doi: 10.1145/3458336.3465297.

[3] H. D. Dixit *et al.*, “Silent Data Corruptions at Scale,” 2021. doi: https://doi.org/10.48550/arXiv.2102.11245.

[4] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, “Detecting silent data corruptions in the wild,” 2022. doi: https://doi.org/10.48550/arXiv.2203.08989.

[5] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo, “Understanding Silent Data Corruptions in a Large Production CPU Population,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, New York, NY, USA: ACM, Oct. 2023, pp. 216–230. doi: 10.1145/3600006.3613149.

[6] X. Iturbe, B. Venu, and E. Ozer, “Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU,” in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, Sep. 2016, pp. 91–96. doi: 10.1109/DFT.2016.7684076.

[7] J. R. Azambuja *et al.*, “A Fault Tolerant Approach to Detect Transient Faults in Microprocessors Based on a Non-Intrusive Reconfigurable Hardware,” *IEEE Trans Nucl Sci*, vol. 59, no. 4, pp. 1117–1124, Aug. 2012, doi: 10.1109/TNS.2012.2201750.

[8] T. Ç. Köylü, C. R. W. Reinbrecht, S. Hamdioui, and M. Taouil, “RNN-Based Detection of Fault Attacks on RSA,” in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, Oct. 2020, pp. 1–5. doi: 10.1109/ISCAS45731.2020.9180708.

[9] S. Zhang, P. Dai, N. Li, and Y. Chen, “A Radiation-Hardened Triple Modular Redundancy Design Based on Spin-Transfer Torque Magnetic Tunnel Junction Devices,” *Applied Sciences*, vol. 14, no. 3, p. 1229, Feb. 2024, doi: 10.3390/app14031229.

[10] T. Arifeen, A. Hassan, and J.-A. Lee, “A Fault Tolerant Voter for Approximate Triple Modular Redundancy,” *Electronics (Basel)*, vol. 8, no. 3, p. 332, Mar. 2019, doi: 10.3390/electronics8030332.

[11] T. Ç. Köylü, C. R. Wedig Reinbrecht, M. Brandalero, S. Hamdioui, and M. Taouil, “Instruction flow-based detectors against fault injection attacks,” *Microprocess Microsyst*, vol. 94, p. 104638, Oct. 2022, doi: 10.1016/j.micpro.2022.104638.

[12] H. Selg, M. Jenihhin, P. Ellervee, and J. Raik, “ML-Based Online Design Error Localization for RISC-V Implementations,” in *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, Jul. 2023, pp. 1–7. doi: 10.1109/IOLTS59296.2023.10224864.

[13] L. Gaber, A. I. Hussein, and M. Moness, “Fault Detection based on Deep Learning for Digital VLSI Circuits,” *Procedia Comput Sci*, vol. 194, pp. 122–131, 2021, doi: 10.1016/j.procs.2021.10.065.

[14] S. Dutto, A. Savino, and S. Di Carlo, “Exploring Deep Learning for In-Field Fault Detection in Microprocessors,” in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, Feb. 2021, pp. 1456–1459. doi: 10.23919/DATE51398.2021.9474120.

[15] A. Vishnu, H. van Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie, “Fault Modeling of Extreme Scale Applications Using Machine Learning,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, May 2016, pp. 222–231. doi: 10.1109/IPDPS.2016.111.

[16] K. Khalil, O. Eldash, A. Kumar, and M. Bayoumi, “Machine Learning-Based Approach for Hardware Faults Prediction,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 11, pp. 3880–3892, Nov. 2020, doi: 10.1109/TCSI.2020.3010743.

[17] B. K. S. V. L. Varaprasad, R. Anilkumar, B. A. Prasad, S. P. Bondapalli, and K. Padmapriya, “Design of Watchdog Circuit using Decision Trees for Detection of Single Event Upsets in Processor,” in *2021 Third International Conference on Inventive Research in Computing Applications (ICIRCA)*, IEEE, Sep. 2021, pp. 1306–1311. doi: 10.1109/ICIRCA51532.2021.9544797.

[18] C. Torres-Huitzil and B. Girau, “Fault and Error Tolerance in Neural Networks: A Review,” *IEEE Access*, vol. 5, pp. 17322–17341, 2017, doi: 10.1109/ACCESS.2017.2742698.

[19] M. Dampfhoffer, T. Mesquida, A. Valentian, and L. Anghel, “Backpropagation-Based Learning Techniques for Deep Spiking Neural Networks: A Survey,” *IEEE Trans Neural Netw Learn Syst*, vol. 35, no. 9, pp. 11906–11921, Sep. 2024, doi: 10.1109/TNNLS.2023.3263008.

[20] J. Liu *et al.*, “Exploring Self-Repair in a Coupled Spiking Astrocyte Neural Network,” *IEEE Trans Neural Netw Learn Syst*, vol. 30, no. 3, pp. 865–875, Mar. 2019, doi: 10.1109/TNNLS.2018.2854291.

[21] J. Liu, J. Harkin, L. P. Maguire, L. J. McDaid, and J. J. Wade, “SPANNER: A Self-Repairing Spiking Neural Network Hardware Architecture,” *IEEE Trans Neural Netw Learn Syst*, vol. 29, no. 4, pp. 1287–1300, Apr. 2018, doi: 10.1109/TNNLS.2017.2673021.

[22] S. Matinizadeh, S. Johari, A. Mohammadhassani, and A. Das, “Towards Biology-Inspired Fault Tolerance of Neuromorphic Hardware for Space Applications,” in *2024 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, Oct. 2024, pp. 1–7. doi: 10.1109/DFT63277.2024.10753531.

[23] D. Simpson, J. Harkin, M. McElholm, and L. McDaid, “Smart Watchdog Mechanism for Fault Detection in RISC-V,” in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, May 2025, pp. 1–5. doi: 10.1109/ISCAS56072.2025.11044018.

[24] D. Simpson, J. Harkin, M. McElholm, and L. McDaid, “Smart Watchdog for RISC-V: A Novel Spiking Neural Network Approach to Fault Detection,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–1, 2025, doi: 10.1109/TCSII.2025.3583042.

[25] D. Simpson, J. Harkin, M. McElholm, and L. McDaid, “Live Demonstration: Smart Watchdog Mechanism for Real-time Fault Detection in RISC-V,” in *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, May 2025, pp. 1–1. doi: 10.1109/ISCAS56072.2025.11044164.

[26] S. Nolting, “The NEORV32 RISC-V Processor - Datasheet,” https://stnolting.github.io/neorv32/.

[27] J. Eshraghian, “SNNtorch,” https://github.com/jeshraghian/snntorch.

[28] J. K. Eshraghian *et al.*, “Training Spiking Neural Networks Using Lessons From Deep Learning,” *Proceedings of the IEEE*, vol. 111, no. 9, pp. 1016–1054, Sep. 2023, doi: 10.1109/JPROC.2023.3308088.

[29] AMD, “FIFO Generator.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/intellectual-property/fifo_generator.html

[30] E. Z. Farsa, A. Ahmadi, M. A. Maleki, M. Gholami, and H. N. Rad, “A Low-Cost High-Speed Neuromorphic Hardware Based on Spiking Neural Network,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 9, pp. 1582–1586, Sep. 2019, doi: 10.1109/TCSII.2019.2890846.

[31] Cytron, “20Amp 6V-30V DC Motor Driver.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.cytron.io/p-20amp-6v-30v-dc-motor-driver?srsltid=AfmBOorcNniOtluYu9doa06g5LbaDaSUamLKH17cRu69OtKezMJns16b

[32] Nidec, “Rotary encoder RE30E.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.nidec-components.com/eu/product/detail/00000064/

[33] RS PRO, “RS PRO Tachometer Best Accuracy ±0.05 % - Non Contact LCD.” Accessed: Aug. 25, 2025. [Online]. Available: https://uk.rs-online.com/web/p/tachometers/1938687

[34] Digilent, “Nexys A7.” Accessed: Aug. 25, 2025. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/start?srsltid=AfmBOorccE3tWx1S0dduFCdIhY-7tgZFw5szYPXLhDhRpyWzBXnHvsN-

[35] AMD, “AMD ArtixTM 7 FPGAs.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/artix-7.html

[36] AMD, “AMD VivadoTM Design Suite.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html

[37] AMD, “AMD MicroBlazeTM Processor.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/microblaze.html

[38] AMD, “AMD VitisTM Unified Software Platform.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html

[39] A. Joubert, B. Belhadj, O. Temam, and R. Heliot, “Hardware spiking neurons design: Analog or digital?,” in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jun. 2012, pp. 1–5. doi: 10.1109/IJCNN.2012.6252600.

[40] AMD, “AMD VersalTM AI Edge Series.” Accessed: Aug. 25, 2025. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/ai-edge-series.html#tabs-246a8928d1-item-e1a47a9c87-tab

[41] “Sayeed Shafayet Chowdhury,” “Nitin Rathi,” and “Kaushik Roy,” “One Timestep is All You Need: Training Spiking Neural Networks with Ultra Low Latency,” *Arxiv Preprint arXiv:2110.05929*, Oct. 2021.