Fault Handling FSM using Verilog HDL

Candidate: Rahul Shibu

Institution: Ramaiah Institute of Technology

Submission To: Arys Garage Private Limited

Date: September 2025

Abstract

This project presents the design of a fault-tolerant system using a Verilog-based finite state machine (FSM), an important approach in modern embedded electronics. The FSM monitors four key fault conditions: **Over-Voltage (OV), Under-Voltage (UV), Over-Temperature (OT),** and **Under-Current (UC).** Based on these inputs, the system transitions between four operating states: **NORMAL, WARNING, FAULT,** and **SHUTDOWN.**

To ensure stable operation, short-lived transient faults are filtered using persistence counters, while masking options allow certain fault inputs to be disabled if required. The design was implemented and tested using Icarus Verilog, with signal behaviour examined in GTKWave. Further analysis of simulation data was carried out through Python-based plotting. The results confirm that the FSM responds reliably, showing strong robustness in handling different fault scenarios.

Content

Tools and AI Usage

The project was carried out using a structured workflow that combined open-source EDA tools with selective AI-based support. Icarus Verilog was employed to compile and simulate the FSM design, while GTKWave was used to visualize waveform activity and validate timing behavior. For post-processing of results, simulation logs were exported into CSV format and analyzed using Python with Pandas and Matplotlib, enabling clear visualization of fault-event trends. GitHub was adopted for version control, ensuring that development followed incremental updates with well-documented commits.

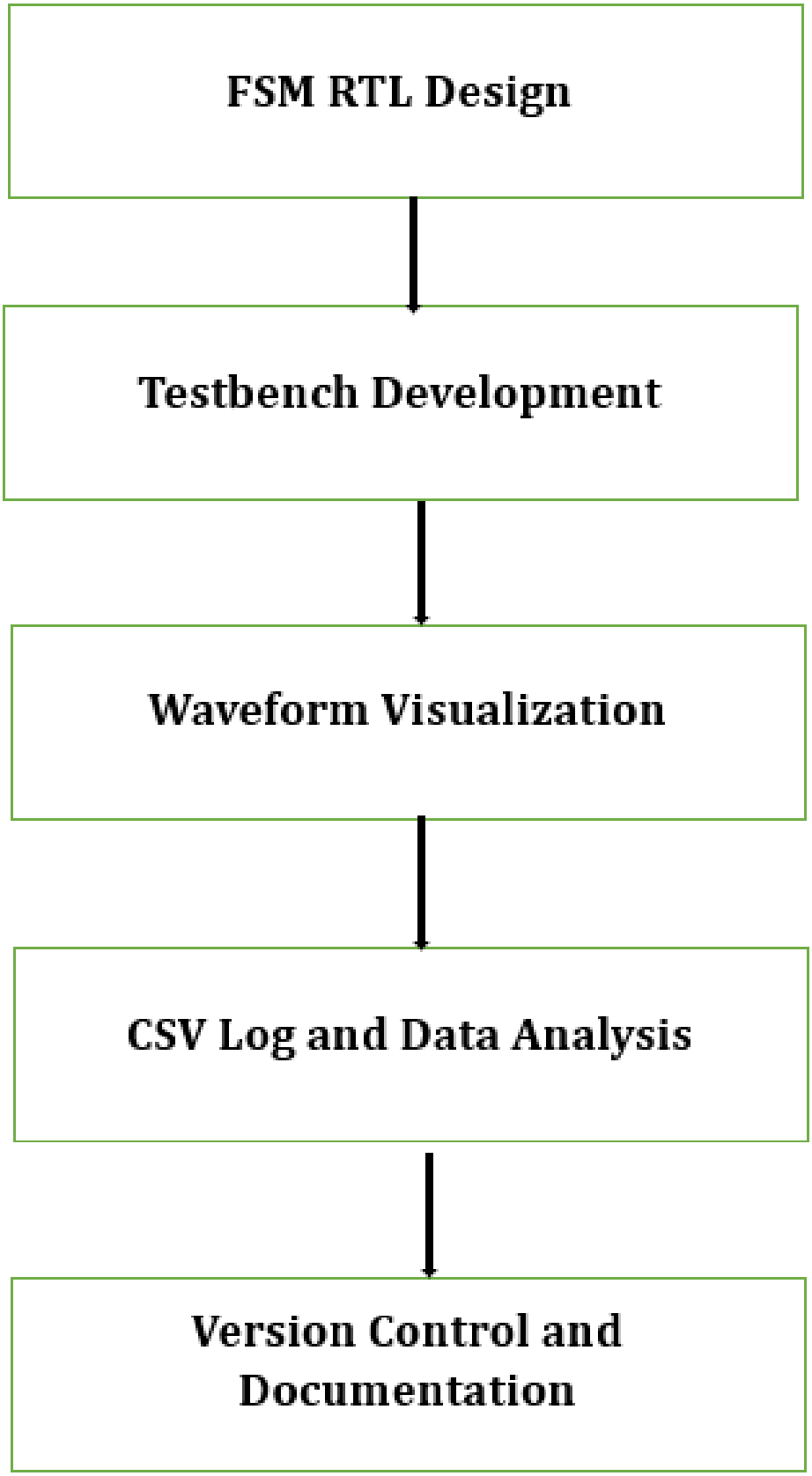
AI-assisted tools were used in a limited and supportive capacity, primarily for accelerating repetitive tasks such as preparing boilerplate Verilog templates, drafting portions of the testbench, and organizing CSV logging scripts. All AI-generated material was carefully reviewed, refined, and validated through simulations before being integrated into the project. The core design decisions, coding of the state machine, debugging, and experimental validation were performed manually, with AI used only as a productivity aid rather than a substitute for engineering judgment.

Design and Methodology

The proposed fault-tolerant finite state machine (FSM) is designed around four operating states, each encoded in binary for efficient hardware implementation: NORMAL (00), WARNING (01), FAULT (10), and SHUTDOWN (11). The FSM transitions between these states based on the persistence and severity of the detected fault signals.

All four monitored fault inputs—Over-Voltage (OV), Under-Voltage (UV), Over-Temperature (OT), and Under-Current (UC)—are tracked using dedicated persistence counters. These counters ensure that short-lived or transient glitches do not trigger unnecessary state transitions. For instance, if a fault remains active beyond the threshold count, the system transitions from NORMAL to WARNING, indicates sustained deterioration by moving to FAULT, and, if unresolved, escalates to the SHUTDOWN state to ensure system safety.

A masking mechanism is implemented to provide flexibility by selectively disabling certain fault inputs when they are either irrelevant to a given application or under diagnostic testing. Additionally, an operator-controlled clear_warning input allows recovery from transient or resolved faults without full system shutdown, improving robustness in dynamic conditions.


FSM RTL Design
Testbench Development
Waveform Visualization
CSV Log and Data Analysis
Version Control and Documentation

Implementation Details

The finite state machine (FSM) was implemented as a parameterized Verilog module, allowing flexible configuration for different application needs. Three persistence thresholds were defined—P_WARN, P_FAULT, and P_SHUT—which governed how long a fault must remain active before the system transitions into the WARNING, FAULT, or SHUTDOWN states respectively. This parameterization makes the design reusable across various embedded systems with different tolerance levels.

The module accepts multiple fault inputs (Over-Voltage, Under-Voltage, Over-Temperature, and Under-Current), along with associated mask signals that allow selective disabling of these faults when necessary. An operator-controlled clear_warning input was also incorporated to give external control for fault recovery in case of transient issues or after a reset procedure.

On the output side, the FSM provides four key signals:

- **warn** – indicates the system has detected a persistent but recoverable issue.
- **fault** – indicates a serious condition requiring attention but not yet critical.
- **shutdown** – signals that the system must be disabled for safety reasons.
- **active_fault_id** – identifies which fault is currently active, aiding in diagnostics.

Testbench Development

To thoroughly validate the FSM, a Verilog testbench was designed to simulate a wide range of fault conditions. Four representative

scenarios were chosen to highlight the behavior of persistence counters, masking, and recovery features:

1. **Transient fault (UC glitch):** A short-lived under-current fault that disappears before crossing the persistence threshold. The FSM correctly ignores this glitch, demonstrating robust filtering.
2. **Persistent OV fault:** An over-voltage condition that persists long enough to first raise a WARNING, later escalate into a FAULT, and eventually clear after removal of the fault. This confirmed the correct functioning of persistence counters and fault clearance.
3. **UC fault leading to SHUTDOWN**: A sustained under-current condition that exceeded all thresholds, pushing the FSM into the SHUTDOWN state. This tested the most critical safety response of the system.
4. **Masked UC fault:** In this case, the under-current input was intentionally masked. Despite fault activity, the FSM ignored it as expected, demonstrating proper handling of masking logic.

Simulation and Analysis

The design was compiled and executed using Icarus Verilog (iverilog and vvp). Simulation dumps were then examined in GTKWave, which provided a detailed visual timeline of how the system transitioned between states based on different fault conditions. The plots confirmed that persistence counters were behaving as designed, filtering out transient glitches while escalating real faults.

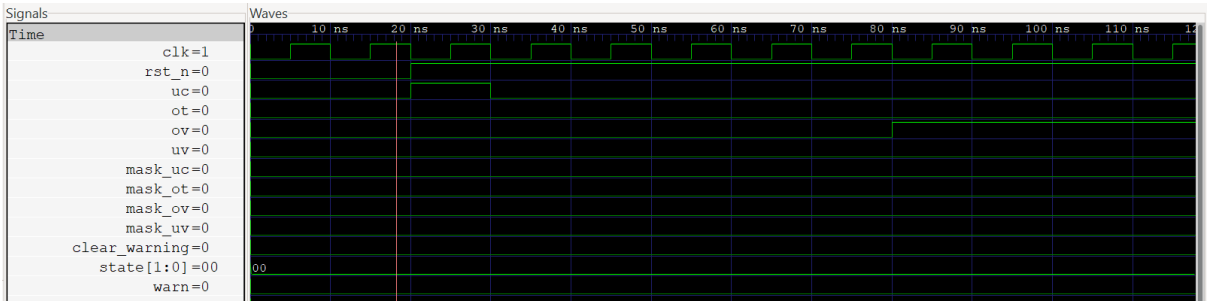
To gain further insight, all FSM outputs and persistence counter values were logged into CSV files during simulation. These logs were processed using Python (Pandas for data handling and Matplotlib for visualization), producing clear time series plots.

These visualizations made it easier to study how fault signals influenced state transitions, especially in cases involving masking and recovery.

Overall, the combination of parameterized design, systematic testbench scenarios, and thorough simulation analysis demonstrated that the FSM is both robust and flexible. It reacts appropriately to a range of operating conditions, ensuring safety while avoiding unnecessary shutdowns due to transient events.

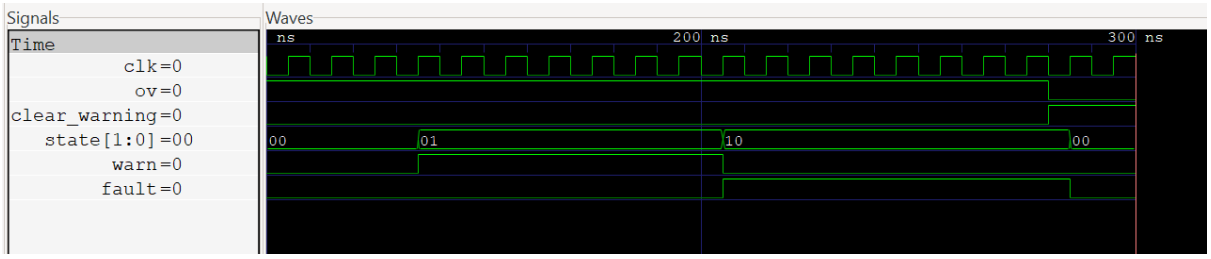
Results

Case 1: Transient UC Fault



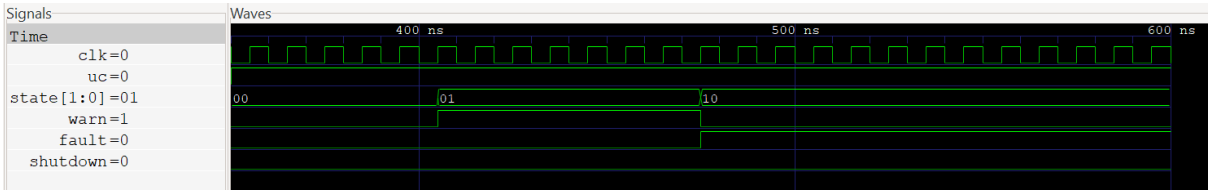
FSM remained in NORMAL state, proving glitch filtering.

Case 2: Persistent OV Fault



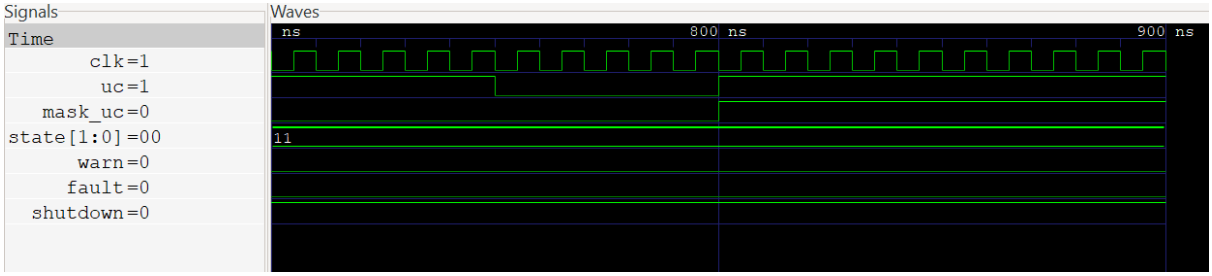
FSM transitioned to WARNING, then FAULT, before returning to NORMAL when cleared.

Case 3: Persistent UC Fault

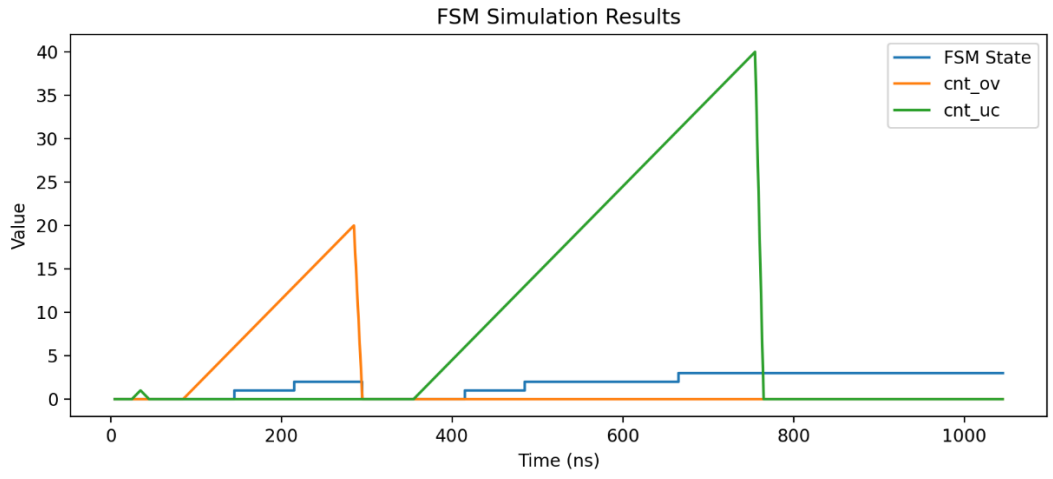


FSM escalated to SHUTDOWN, demonstrating critical fault handling.

Case 4: Masked UC Fault



FSM ignored UC input when mask enabled.



Challenges and Limitations

Challenges

- Toolchain Setup: Installing and configuring Icarus Verilog and GTKWave on Windows required resolving PATH conflicts and ensuring compatibility with Python libraries.
- Threshold Tuning: Designing persistence counters was tricky. Low thresholds falsely detected glitches as faults, while high thresholds delayed detection.
- State Machine Debugging: Managing smooth transitions between NORMAL, WARNING, FAULT, and SHUTDOWN was complex, especially during overlapping faults.
- Data Logging: Combining $fwrite-based CSV logs with $dumpvars initially caused file issues, fixed after multiple iterations.
- Git Workflow: Moving from bulk commits to disciplined, incremental commits improved clarity and traceability.

Limitations

- Simulation-Only: The FSM was tested in Icarus Verilog but not synthesized on FPGA/ASIC, so hardware behavior was not validated.
- Limited Faults: Only four faults (OV, UV, OT, UC) were included; real systems require coverage of additional conditions.
- Single-Fault Handling: The design responds to one active fault at a time, without prioritization for simultaneous events.

Conclusion

The design and simulation of the Fault Handling FSM show that even a compact HDL-based approach can reliably detect and respond to different system faults. Thanks to persistence counters, the FSM is able to ignore short, random glitches and only react when a fault lasts long enough to be meaningful. This ensures that issues are escalated step by step—from WARNING to FAULT, and finally to SHUTDOWN—just as required in real-world safety systems.

Such behavior makes the FSM highly relevant to applications like Battery Management Systems (BMS), automotive ECUs, and power electronics, where timely and accurate fault handling is critical to protect both hardware and users. The addition of masking logic provides flexibility for testing or temporary overrides, while the clear_warning option mirrors how faults are acknowledged and cleared during actual maintenance procedures.

Although this version was validated only in simulation, it lays a solid foundation for further work. Future steps could include FPGA prototyping, hardware-in-the-loop testing, and integration with real sensor inputs. With extensions like handling multiple faults at once and adaptive threshold tuning, the FSM has the potential to grow into a practical, production-ready solution for fault-tolerant embedded systems.

Appendix

Git Log

- add GTKWave screenshots for all FSM test cases
- changes to fault_fsm and test bench
- add FSM, testbench, adc2flag, and can_encoder modules
- Update README.md
- Initial commit

Run Instructions

Environment Setup

- OS: Windows 10/11
- Tools Installed:
    - Icarus Verilog (iverilog + vvp)
    - GTKWave (waveform viewer)
    - Python 3.12 with Pandas + Matplotlib
    - GitHub Desktop (for version control)

Steps to compile, Simulate and visualize

1. Navigate to source folder

cd src

2. Compile FSM and Testbench

iverilog -o sim_fault_fsm fault_fsm.v tb_fault_fsm.v

3. Run simulation (generates VCD + CSV log)

vvp sim_fault_fsm

4. Open GTKWave for waveform analysis

gtkwave tb_fault_fsm.vcd

5. Move CSV log to simulations/

mv sim_log.csv ../sims/

6. Generate Python plot

python plot_sim.py