Full-Auto Bug Buster

AI-Powered RTL Debugging Agent

CogniChip Hackathon
Demonstration

Bob Huang (hh2727@nyu.edu)

Shahran Newaz (sn3507@nyu.edu)

https://github.com/shahran-n/Bug-Buster

Table of Contents

Problem Domain

RTL debugging is still painfully manual:

- Bugs hide across multiple artifacts: RTL source, simulator logs, and waveforms.
- Finding root cause often means lots of back-and-forth:
  - read log ➜ locate signal ➜ inspect VCD ➜ jump to RTL line.
- Early student / prototyping projects rarely have polished infrastructure (coverage, assertions, lint, etc.).
- A fast “first pass” assistant can reduce time-to-insight and teach better RTL habits.

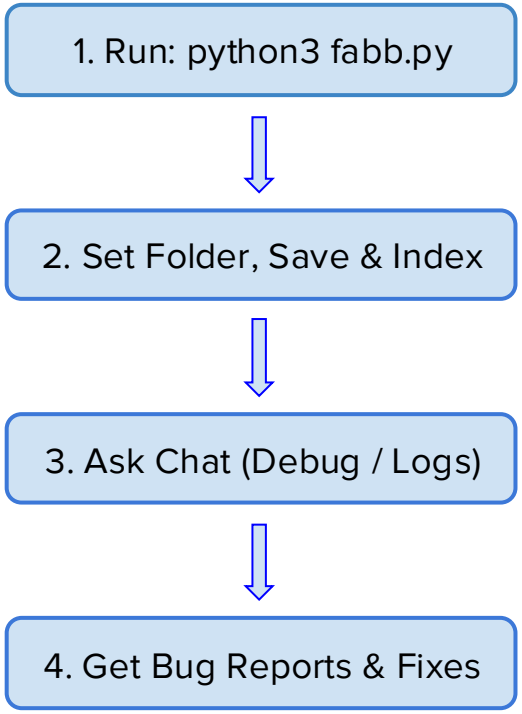
Our Goal:

Make RTL bug-finding feel like chatting with a verification engineer — backed by local parsing of RTL/VCD/logs.

Project Overview - FABB in 1 Slide

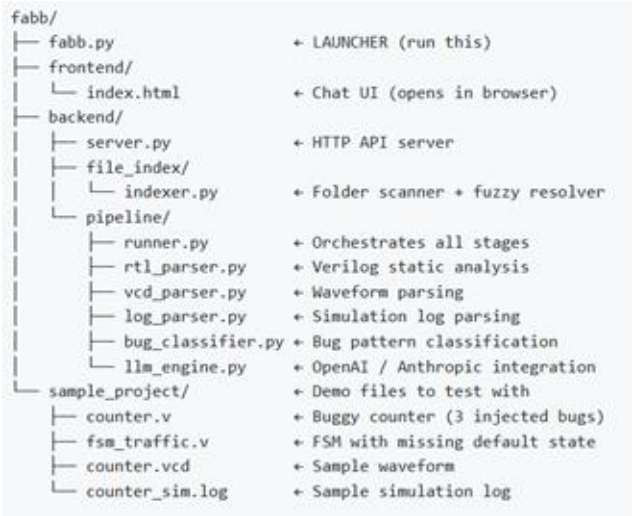
What it does

- Runs locally: Python stdlib backend + static HTML chat UI.
- Indexes a project folder (Verilog/SystemVerilog + VCD + logs).
- Understands prompts like “debug counter.v” or “check latest simulation”.
- Finds likely issues from static RTL patterns + VCD anomalies + log failures.
- Optional LLM layer: richer explanations + patch suggestions when an API key is provided.



Design Methodology

- We utilized CogniChip to develop the foundational framework of our application, as shown on the right.
- Due to its limitations in processing large prompts, we iteratively broke down complex instructions into smaller, manageable inputs.
- The overall architecture was intentionally designed to balance an intuitive, easy-to-navigate user interface with robust backend functionality.



```
fabb/
├── fabb.py                    ← LAUNCHER (run this)
├── frontend/
│   └── index.html             ← Chat UI (opens in browser)
├── backend/
│   ├── server.py              ← HTTP API server
│   ├── file_index/
│   │   └── indexer.py         ← Folder scanner + fuzzy resolver
│   └── pipeline/
│       ├── runner.py          ← Orchestrates all stages
│       ├── rtl_parser.py      ← Verilog static analysis
│       ├── vcd_parser.py      ← Waveform parsing
│       ├── log_parser.py      ← Simulation log parsing
│       ├── bug_classifier.py  ← Bug pattern classification
│       └── llm_engine.py      ← OpenAI / Anthropic integration
└── sample_project/            ← Demo files to test with
    ├── counter.v              ← Buggy counter (3 injected bugs)
    ├── fsm_traffic.v          ← FSM with missing default state
    ├── counter.vcd            ← Sample waveform
    └── counter_sim.log        ← Sample simulation log
```

Application Architecture - Higher Level

Frontend (Browser)

Static HTML chat UI

Settings: folder + API key

Shows structured bug blocks

HTTP

Backend (Python stdlib HTTP Server)

HTTP API

GET /health
GET /api/index
POST /api/chat
GET/POST /api/config

File Index

Scans folder for .v/.sv/.vcd/.log
Fuzzy resolver + “latest file” helper

Pipeline Runner

Loads relevant files ➔ runs RTL/VCD/log parsing ➔ optionally calls LLM ➔ returns: plain text + structured bug blocks

LLM Layer: Provider generates richer explanations and fixes

Application Architecture - RTL/Static

Regex-based RTL parser (no external deps)

Extracts modules, signals/ports, always blocks, reset/clock candidates, FSM state params.

Flags suspicious patterns: missing posedge/negedge, dual-edge sensitivity, possible reset polarity mismatch.

Feeds summary + suspicious lines into the pipeline context so the assistant can reference exact RTL fragments.

Downstream context (for bug finding)

- Loaded file content (first ~3000 chars)
- Parsed structure: module names, signal count, always block count
- Suspicious line list (line #, text, issue)

Frequently Asked Questions

What's the tech stack?

- **Backend:** Python 3 (stdlib HTTP server + parsers)
- **Frontend:** static HTML/CSS/JS chat UI
- **Artifacts:** .v/.sv, .vcd, .log/.txt
- **AI Analyzation:** OpenAI/Anthropic API calls when configured

What bug types can it detect well right now?

Common "student/prototype" bugs such as:

- Incorrect sensitivity lists (missing posedge/negedge)
- Reset polarity mismatches / reset sequencing issues
- Off-by-one comparisons
- Basic FSM next-state issues (e.g., missing default)
- X/Z propagation, stuck signals (from VCD)
- FAIL/ASSERT/MISMATCH evidence (from logs)

Why choose stdlib-only instead of frameworks (Flask/FastAPI) and parsers?

To maximize portability: **clone → run** with minimal setup, fewer dependency conflicts, and easier evaluation in constrained environments.

How does it "understand" RTL if it doesn't use a real Verilog parser?

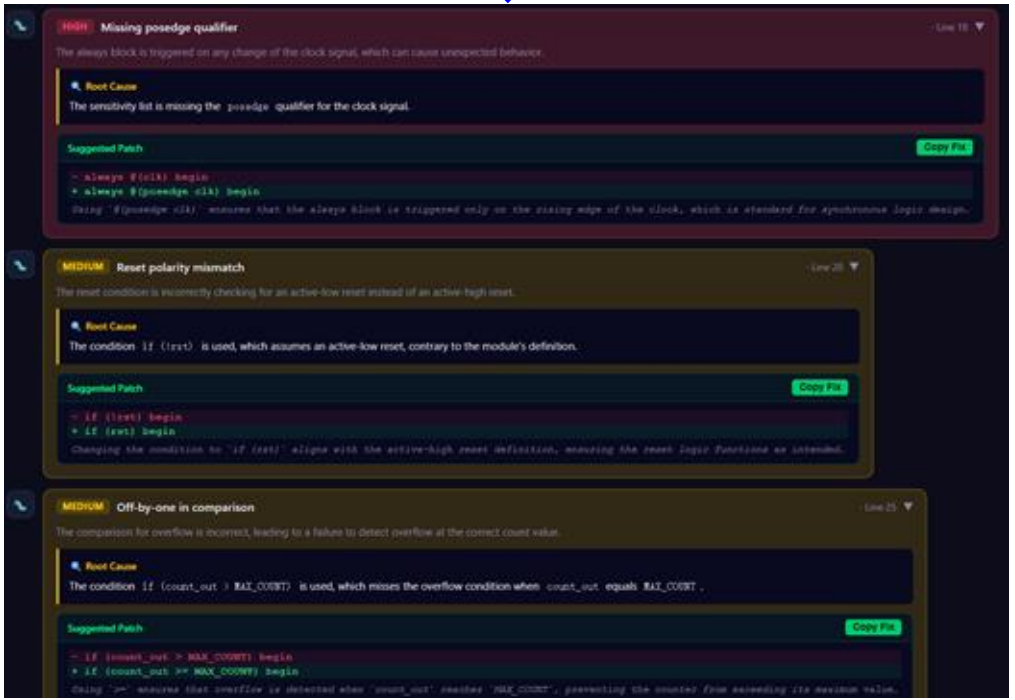
It uses **targeted regex/text heuristics** to extract practical structure (modules, always blocks, reset/clock patterns, FSM hints) and to flag common bug patterns. It's intentionally pragmatic, not fully language-complete.

Is this actually a server or just a script?

It's a real server process, but it's meant for **local single-user use**: a lightweight Python process serving a UI and APIs on **localhost**.

Simulation Results

- Project is tested with a sample project, containing Verilog RTL with **injected errors**
- **Successfully** detected bugs
- **Successfully** analyzed root causes
- **Successfully** generated patch suggestions



Performance Discussions

Advantages:

Local / deterministic costs:

- Indexing: recursive folder scan over supported extensions (.v/.sv/.vcd/.log/.txt)
- Parsing: linear scans (regex-based RTL parse; text scan for log; line scan for VCD)
- Output formatting: bug blocks + summary

Disadvantages:

Variable costs:

- Requires API Key for analysis
  - Due to constraints of prompt limits without a key, detailed responses might not be available.
- Response speed relies on LLM service
  - In some test cases, the patch suggestion stage took longer than expected.

Scalability notes

- Best for small/medium projects where fast heuristics are enough to narrow down suspects.
- For huge repos: add caching, incremental indexing, and smarter file-selection (top-K relevant modules).
- For large VCDs: add time-windowed parsing or pre-filter signals of interest.

Challenges & Lessons

1. Verilog is hard to parse with regex: focus on “useful shortcuts” vs completeness.
2. File matching needs tolerance: fuzzy resolver + “latest” helpers.
3. Make outputs renderable: structured bug blocks are worth the effort.
4. No-dependency constraint (stdlib-only) shapes many design choices.

Future Steps

- Explore possibilities of user access without API keys.
- Implement an executable to improve user experience.
- Better relevance: rank files/modules from prompt and signal names.
- Deeper waveform reasoning: assertions from VCD, temporal patterns.
- Add a test suite.

THANKS FOR LISTENING!

Github Repository Link:
https://github.com/shahran-n/Bug-Buster