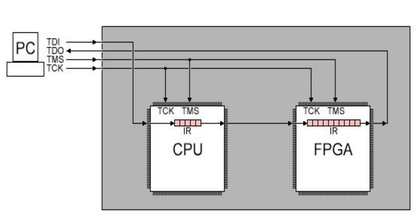
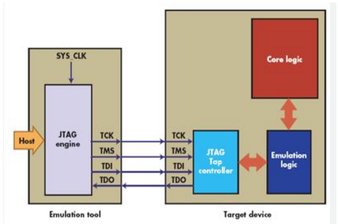
Automated - JTAG Programming for Production Environments

Joint Test Action Group (JTAG)

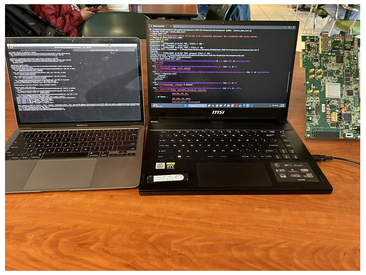
- JTAG is an interface (based on based on the IEEE 1149.1 standard) used for programming, testing, and debugging digital devices such as FPGAs and SoCs.





Objective and Motivation

- To address production bottlenecks, automated programming interface using OpenOCD and a RESTful API allows for remote bitstream uploads, programming, and status tracking
- This reduces human error and enables scalable, remote, and repeatable programming of Microchip FPGAs in high-volume environments.



Requirements

Class / View	HTTP Method	Description
Upload View	POST	Handles bitstream file uploads to the server. Validates and saves .bit or .svf files.
Program View	POST	Starts a background thread to run OpenOCD and program the FPGA with a specified bitstream.
Verify View	POST	Runs OpenOCD to verify a given bitstream. Validates file existence and captures logs.
Cancel View	POST	Terminates the currently running OpenOCD programming process and resets system state.
Status View	GET	Returns the current system state (idle, programming, etc.) as JSON.
GetFiles View	GET	Returns a list of all available bitstream files in the server's directory.
Device View	GET/ POST	Intended to return JTAG programmer hardware info.

Automated FPGA Programming in Production

- Modern high-throughput production environments require rapid and repeatable programming of FPGAs and embedded systems.
- In many cases, engineers must repeatedly program large numbers of devices, a process that is slow and manual.
- This introduces delays and human error, especially at scale.

Script Features

Core Design

- Our API is built using **Flask**, a lightweight Python web framework.
- Each API function is encapsulated in its own class-based view, inheriting from **Flask.views.MethodView**.
  - Each endpoint can support both **GET** and **POST** methods when appropriate.

Multithreading

- Bitstream programming is ran in a separate thread using Python's threading.Thread so that long-running processes do not block the API server.

Logging & Error Handling

- Output from OpenOCD is logged and returned in case of failure.
- Return values include:
  - Success status (200)
  - Client errors (400)
  - Server errors (500)

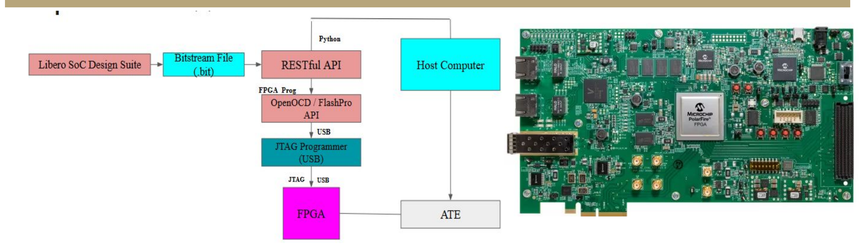
State Management

- In /status/ and /cancel/, a global state variable tracks whether the system is currently idle or programming to prevent collisions and unauthorized termination attempts.

API Security

- request.headers.get("API-Key") is checked before processing critical functions.

Technical Design



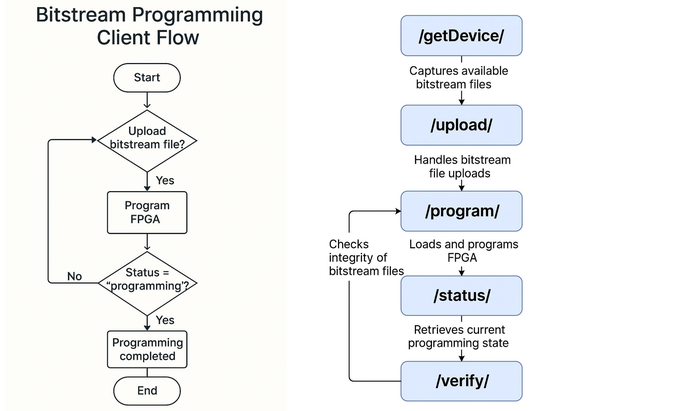
- System is built around a modular client-server architecture
- The client sends HTTP requests to the API using a *request* command in Python
- The server receives user requests via Flask and executes OpenOCD commands to program the FPGA using bitstream files.
  - OpenOCD is launched via Python's subprocess module, and the server tracks state (e.g., idle, programming) to report through /status/.
- Outputs are tracked at every step of the process and in the event of failure, a log of it is recorded to be analyzed.
- Tasks are multithreaded to avoid blocking and is able to perform multiple tasks at once.

Results

- Verified our RESTful API by running client script tests simulating remote FPGA programming. All key endpoints responded successfully, confirming proper functionality.

```
Server response to GET is {'message': 'Hello World! GET'}
Server response to POST is {'message': 'Hello World! POST'}
Server response to POST PROGRAM (raw): 200 {"message":"Verification started for top.bit","status":"verifying"}

Server response to POST PROGRAM (JSON): {'message': 'Verification started for top.bit', 'status': 'verifying'}
Server response to STATUS:  {'message': 'My state is verifying'}
Server response to VERIFY is {'status': 'verifying', 'message': 'Verification started for top.bit'}
```




```
CancelView: No active programming operation
172.20.10.10 - - [12/May/2025 16:39:02] "POST /cancel/ HTTP/1.1" 400 -
172.20.10.10 - - [12/May/2025 16:39:12] "GET /get_device/ HTTP/1.1" 404 -
ExampleView GET
172.20.10.10 - - [12/May/2025 16:41:51] "GET / HTTP/1.1" 200 -
ExampleView POST: {'message': 'post test'}
172.20.10.10 - - [12/May/2025 16:41:51] "POST / HTTP/1.1" 200 -
ProgramView POST: {'bitstream': 'top.bit'}
Programming bitstream: top.bit
[VERIFY] Starting verification for: top.bit
172.20.10.10 - - [12/May/2025 16:41:51] "POST /program/ HTTP/1.1" 200 -
StatusView GET
172.20.10.10 - - [12/May/2025 16:41:51] "GET /status/ HTTP/1.1" 200 -
```

- We tested all major API endpoints via a live client-server demo. The images above show Flask server logs capturing real-time HTTP requests and server-side processing

Future Work

Boundary Scan Testing: Test interconnects (wires, solder joints, pins) on PCBs without using physical test probes
Multi-FPGA Management: Enable programming multiple FPGAs on different ports or daisy chained via JTAG
Expanded Hardware Support: Script currently only supports Microchip Polarfire FPGAs, expanding its features to Altera, Xilinx & other platforms.

ELECTRICAL & COMPUTER ENGINEERING
UNIVERSITY of WASHINGTON

ADVISORS: Rania Hussein, Haidon Shultz, Jeffrey Jancik, Assi Friedman

SPONSOR: Innoflight