**EE533: Network Processor Design & Programming**

**Lab #5: Debugging and Testing Custom NetFPGA Design**

Instructor: Prof. Young Cho, PhD

Team Number:**#3**

***Project Partners:***

*Member #1:* **Sarthak Jain**

*Member #2:* **Archit Sethi**

*Member #3:* **Justin Santos**

*Designed, Created, and Submitted by Team #3*

***University of Southern California***

***Los Angeles, CA 90007***

Team #4 GitHub Repository Link:[*Team#3 Lab-5 Link*](https://github.com/EE533-TEAM3/EE533_LAB_PROJECTS/tree/main/LAB5)

1. **Introduction**
   1. **Overview of Lab Objectives & Explanation of NetFPGA-based mini IDS**

In this lab#5, we aim to debug and test a custom NetFPGA -based Intrusion Detection System (mini-IDS). The objective is to implement and verify the functionality of a passthrough module, integrate a logic analyzer, and incrementally test our mini-IDS design.

The system will be validated using packet injection, pattern matching, and performance analysis under high-speed network conditions.

The primary task in this lab include:

* Integrating and testing a passthrough module for baseline validation
* Implementing a logic analyzer to capture and analyze internal signals
* Incrementally adding and testing mini-IDS components
* Modifying a register access script (idsreg) to interact with hardware registers
* Testing the pattern matching mechanism using traffic generation tools (iperf)
* Verifying correct packet blocking behavior tcpdump

The final goal is to ensure that the mini-IDS functions correctly, detects predefined string patterns in network traffic, and performs efficiently without introducing excessive latency or packet drops.

1. **Procedure**
   1. **Passthrough Implementation**
      1. **Description of adding passthrough.v to the NetFPGA reference NIC/Router**

To ensure that our NetFPGA is set up correctly, a passthrough module (ids.v) was inserted into the NetFPGA reference NIC/router package. This module simply connects input to output using wire type variables, ensuring unmodified packet forwarding.

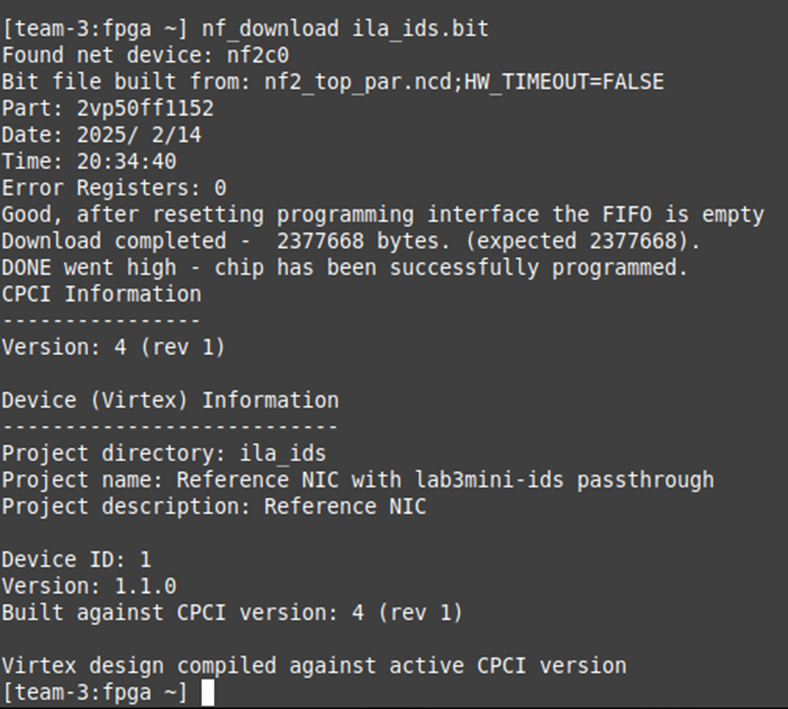


Fig.1. Passthrough ids with ila plkaced to mintor output data of ids

* + 1. **Steps for compiling and generating the bitfile, then uploading the bitfile to the FPGA using *‘scp’* and reconfiguring it**
       1. Implemented passthrough ids.v, which directly connects inputs to outputs.
       2. Compiled the design using the same method as we did in Lab#4 to generate the bitfile
       3. Uploaded the bitfile to the NetFPGA using the Linux scp command
       4. Reconfigured the NetFPGA with the generated bitfile and validated that it behaves the same as the reference NetFPGA bitfile

Expected behavior is that the passthrough module should forward all packets without modification. We verified by pinging between the nodes to confirm correct packet forwarding.

* 1. **Logic Analyzer Development**
     1. **Implementation of a logic analyzer using block RAM**

A Logic Analyzer (ILA) was developed using block RAM (BRAM) to capture internal signals for debugging purposes. The ILA was controlled through a register interface, allowing it to record data and export it for analysis.

* + 1. **Description of register interface usage for recording internal signals**

We designed a register-mapped logic analyzer inside the FPGA. Then, modified the register interface to control the analyzer’s behavior. Implemented a script to extract recording signals from the FPGA and send them to a server. Verified that internal signals were being captured correctly.

A diagram of a computer program

AI-generated content may be incorrect.

Fig. This is the Logic-Block diagram for custom ILA

The logic analyzer should record selected internal signals and allow data retrieval via the register interface.

* 1. **Incremental Design Testing**
     1. **Adding mini-IDS components step-by-step**

To ensure reliability, mini-IDS components were added one at a time with separate compilation and testing at each stage.

* + 1. **Loading and testing the bitfile using *‘nf\_download’***

Steps:

1. Integrated individual IDS components incrementally.
2. Verified correctness at each step before moving forward.
3. Used ‘nf\_download’ to load the bitfile and ensured proper interface configuration.
4. Conducted connectivity tests by ensuring that NetFPGA nodes could ping each other.

Each newly added module should function correctly without affecting each other part of the system.

* 1. **Register-Based Testing**
     1. **Explanation of modifying *idsreg* Perl script**

The perl script *idsreg* was used to interact with NetFPA registers. The script was modified to match the assigned register addresses and to facilitate testing of register-based data processing.

Steps:  
1. Copied idsreg script to the FPGA node using scp.

2. Analysed the script to understand how it interacts with registers.

3. Modified script:

3.1. Use correct NetFPGA compiler-assigned register addresses.

3.2. Support and writing custom register values.

3.3. Test data transformations within the hardware.

Register reads/writes will work correctly allowing the FPGA to process data and return results.

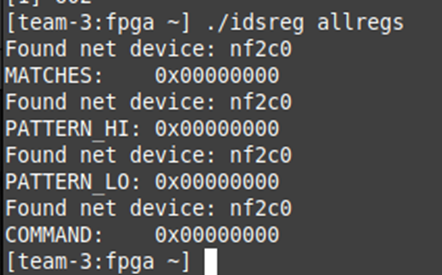


Fig.2. Successfully edited idsreg script to have the correct addresses of our ids registers. I.e. deadbeef is not shown anymore

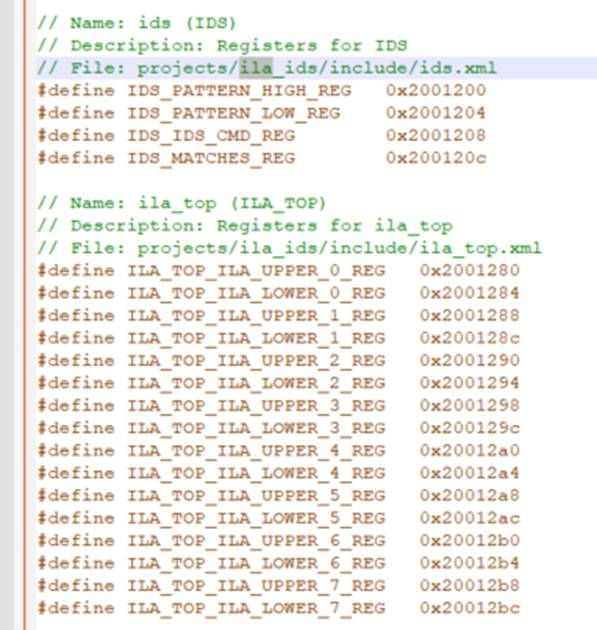
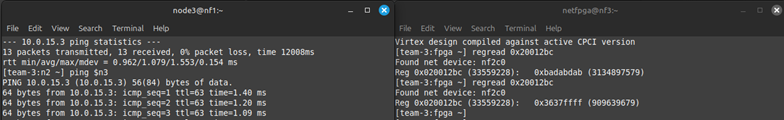
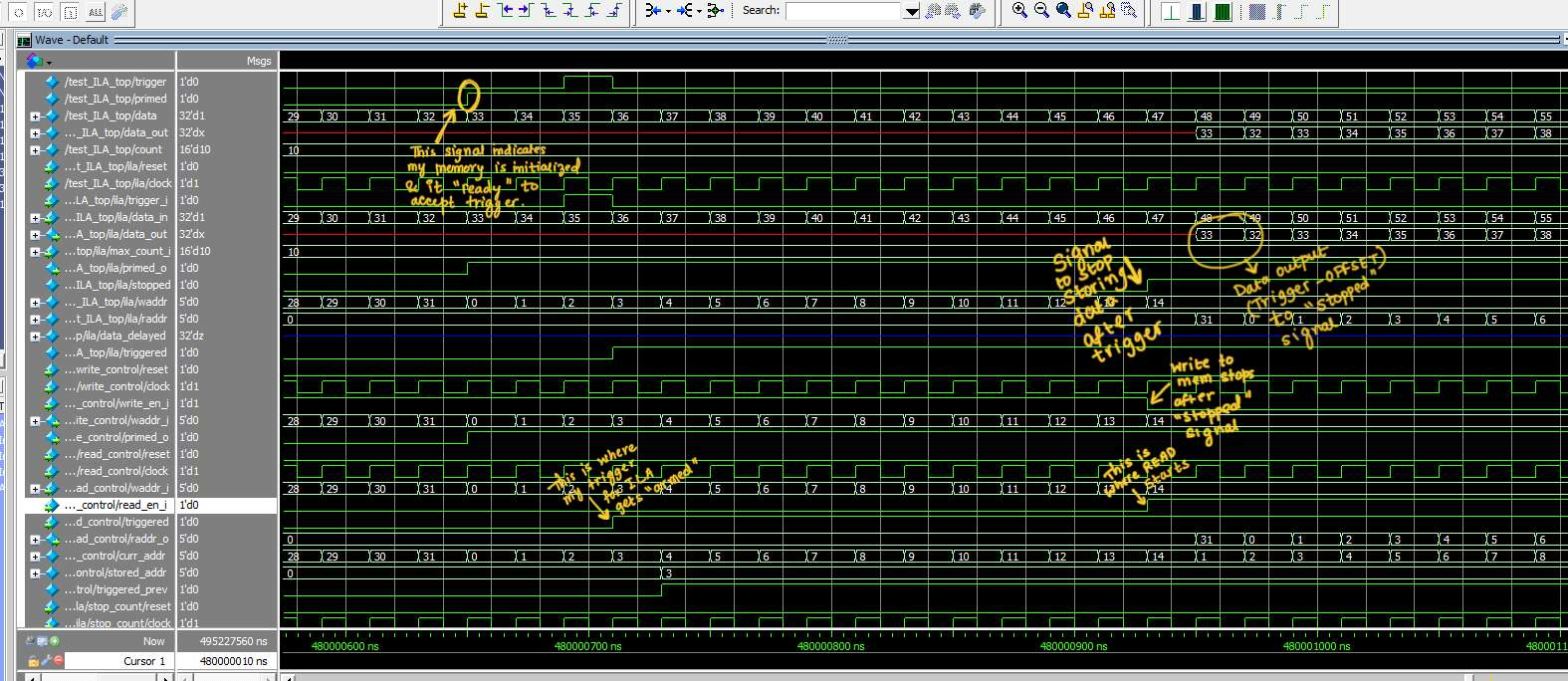


Fig.3. Successfully placed ila registerse into the memory map

1. We did regread 0x20012bc which is the last ila register and read 0xbadabdab which is our initialization value.
2. Then I started ping $n3 on n2
3. After pinging, we did regread 0x20012bc again, and this time. We got a new value 0x3637ffff indicating that our ila is updating properly and we are able to read from it.



* 1. **Traffic Analysis with iperf**
     1. **Running iperf servers/clients on each node**
     2. **Sending:**
        1. **“Good” packets (without the specific string)**
        2. **“Bad” packets (with the designated string)**
     3. **Observing behavior in TCP and UDP modes**
     4. **Using tcpdump to verify packet blocking**
  2. **Final Testing**
     1. **Running idsreg matches to verify the count of dropped packets**

Please note the above-mentioned waveform that has been uploaded:

The *primed* signal indicated that our memory is initialized and it is ready to accept the trigger signal.

The *data out* signal is DATA-OUTPUT (Trigger -Offset) to ‘stopped’ signal

The write address (waddr) is used to stop storing data after the trigger

The write enable (wr\_en\_i) writes to memory stops after ‘stopped’ signal

The read enable (rd\_en\_j) reads where READ starts

The trigger signal (triggered) is where our trigger for ILA gets armed.

1. **Answering Report Questions**
   1. **Identifying a Bug at High-Speed Traffic**
      1. **Explanation of any discovered issues when the mini-IDS operates near 1 Gbps**

At high-speed network traffic (~1Gbps) a performance issues:

* + - 1. Bug: The mini-IDS drops legitimate packets or introduces delays
    1. **Possible causes (e.g. buffer overflow, processing delays)**
       - 1. FIFO/Buffer Overflow: Limited memory can’t handle high-speed traffic.
         2. Pattern Matching delay: The IDS takes long time to analyze packets
         3. Synchronization issues: Packet order may be incorrect under heavy loads.

Solutions:

Increase buffer size to handle traffic bursts.

Optimize pattern matching logic for high-speed performance.

Parallelize packet processing to improve throughput.

* 1. **How the mini-IDS works**
     1. **Summary of the mini-IDS operation and how it interacts with other NetFPA modules**

Mini-IDS is a hardware-based intrusion detection system implemented on NetFPGA.

Functionality:

Receives network packets via NetFPGA.

Extracts payloads and checks for a predefined pattern.

Drops malicious packets if they contain the pattern.

Forwards legitimate traffic to the destination.

* + 1. **Explanation for a technical audience (NetFPGA developers)**

user\_data\_path.v → Manages network packet routing.

ids.v → Implements pattern matching logic.

Register Interface → Allows software-based IDS configuration.

* 1. **Pattern Matching Algorithm**
     1. **Explanation of the system detects patterns in network traffic**

Reads packet payload data.

Extracts a substring from the payload.

Compares it against the predefined pattern (e.g., "ABCDEFG").

If a match is found, the packet is dropped.

Else, it is forwarded normally

* + 1. **Algorithm efficiency and performance considerations**

Use hardware-based comparators to speed up matching.

Implement parallel string comparison to process packets efficiently