EE533 Lab 4 Report

NetFPGA Bitfile Generation and Using NetFPGA

Team Members:
Jeremy Cai (yujiecai@usc.edu, 5988192173)
Prathamesh Hirekodi (hirekodi@usc.edu, 5404695932)
Yanzhe Li (yanzheli@usc.edu, 6884027371)

Date: February 5, 2026

1. Introduction

The objective of Lab 4 is to familiarize the team with the NetFPGA development environment and toolchain. The goals are to:

- set up the NetFPGA tool environment using VirtualBox;
- compile a reference NetFPGA design into a *.bit* file using Fedora VM;
- program the NetFPGA card with reference NIC bitfiles and then evaluate routing performance using *ping* and *iperf;*
- integrate the Lab 3 Mini-IDS hardware into the NetFPGA project and to validate that normal traffic continues to pass while “bad” traffic containing a chosen string pattern is dropped.

2. NetFPGA Tool VM Setup and Reference Project Build

2.1 NETFPGA TOOL VM SETUP

We use the provided NetFPGA compiler virtual machine *Fedora 14* and open a terminal, then switch to root (*su*) in order to compile designs into a NetFPGA bitfile.

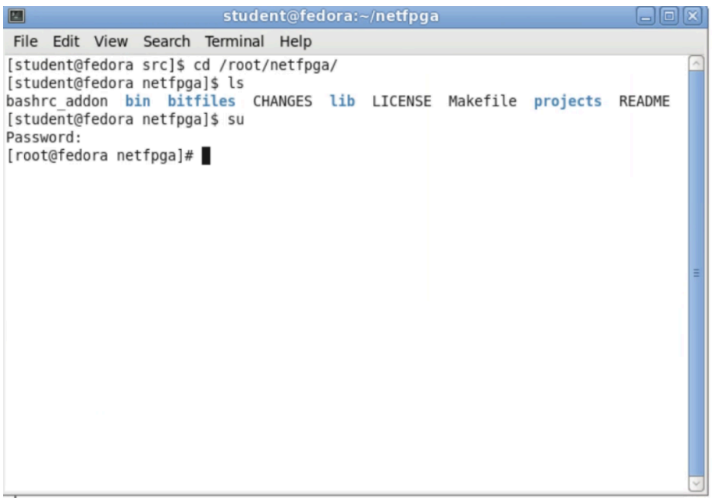


Figure 1. Fedora VM terminal showing *su* and the *root* prompt

2.2 BUILDING THE *REFERENCE_ROUTER* PROJECT BITFILE

Inside the VM, we move to the NetFPGA projects directory and copied the *reference_router* project into our own Lab 4 project folder. Then we enter the *synth* directory and run *make*, which starts compilation and generates the *.bit* file output.

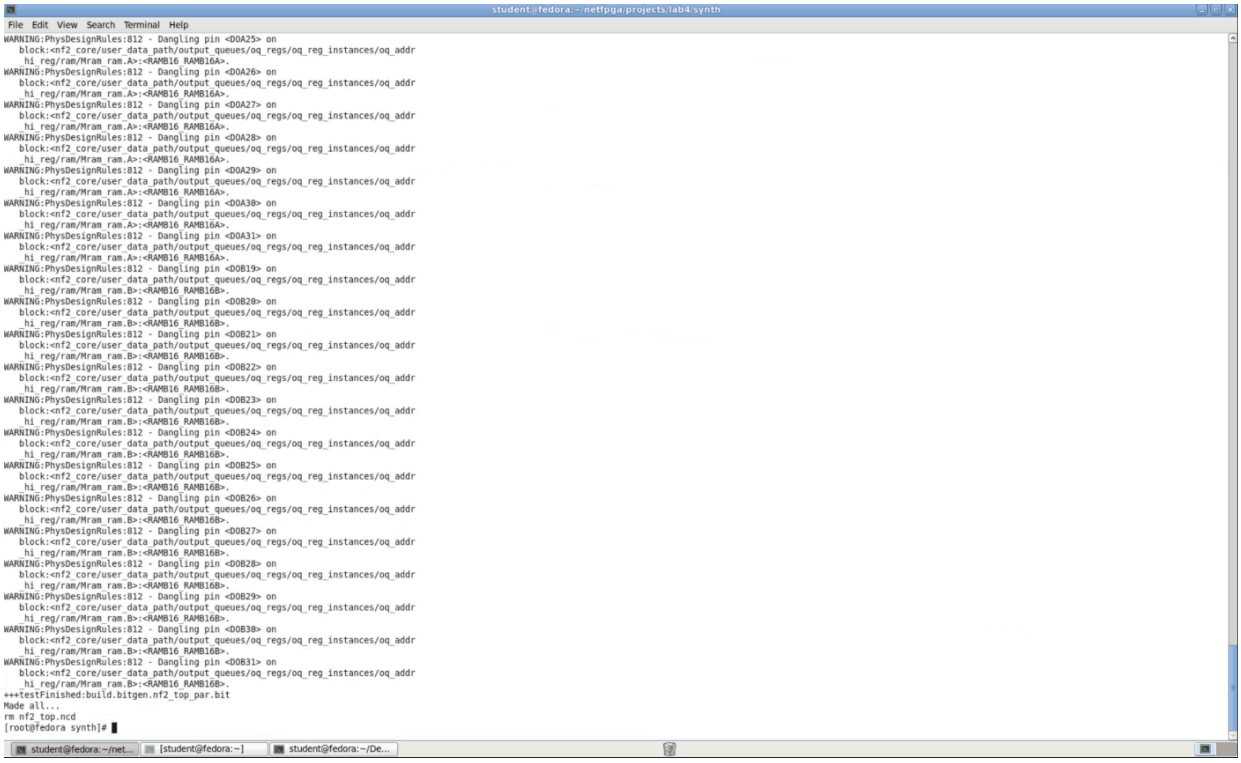


Figure 2. Terminal showing the build completing successfully

3. Network Access and Testbed Environment

3.1 VPN + SSH ENVIRONMENT

To access the NetFPGA machines remotely, we connect through USC VPN and used SSH with legacy-compatible settings as required by the lab environment. We open multiple terminals to monitor FPGA and host nodes simultaneously. Please note the the openterm file mentioned in the lab manual was not provided initially and we wrote our own file which has the save functionality as described in the lab manual.

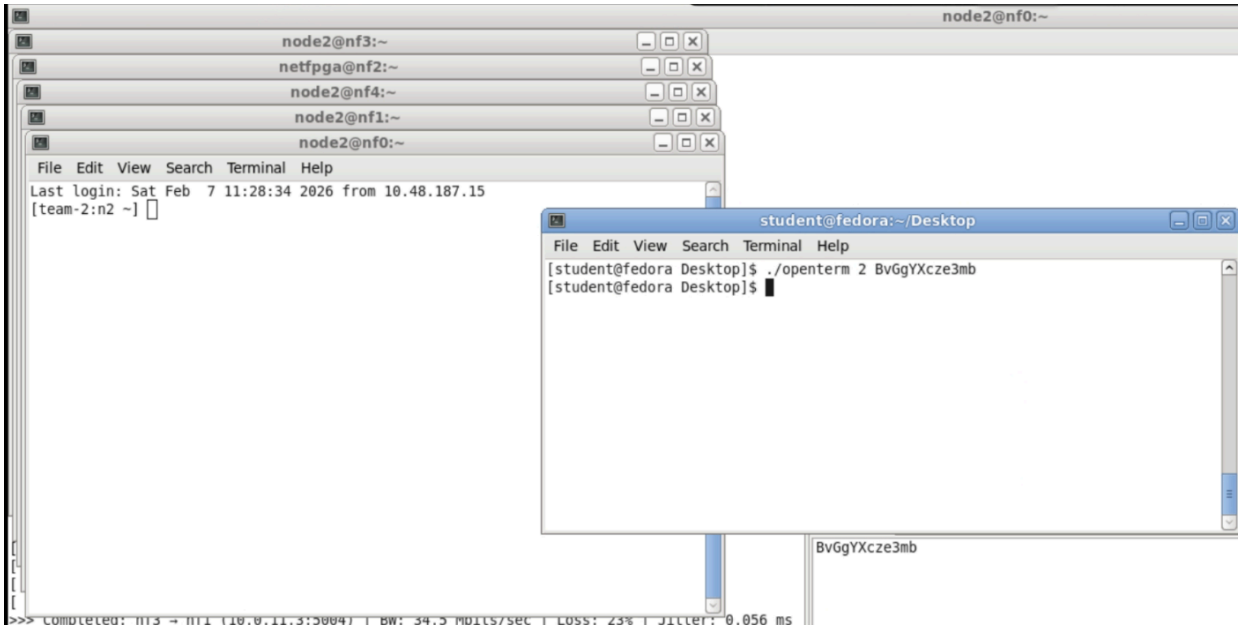


Figure 3. VPN connected status and SSH terminal windows opened

3.2 IDENTIFYING NODE IP ADDRESSES

After logging into the remote nodes, we check the IP addresses using the provided environment variables for nodes 0–3 to confirm correct addressing.

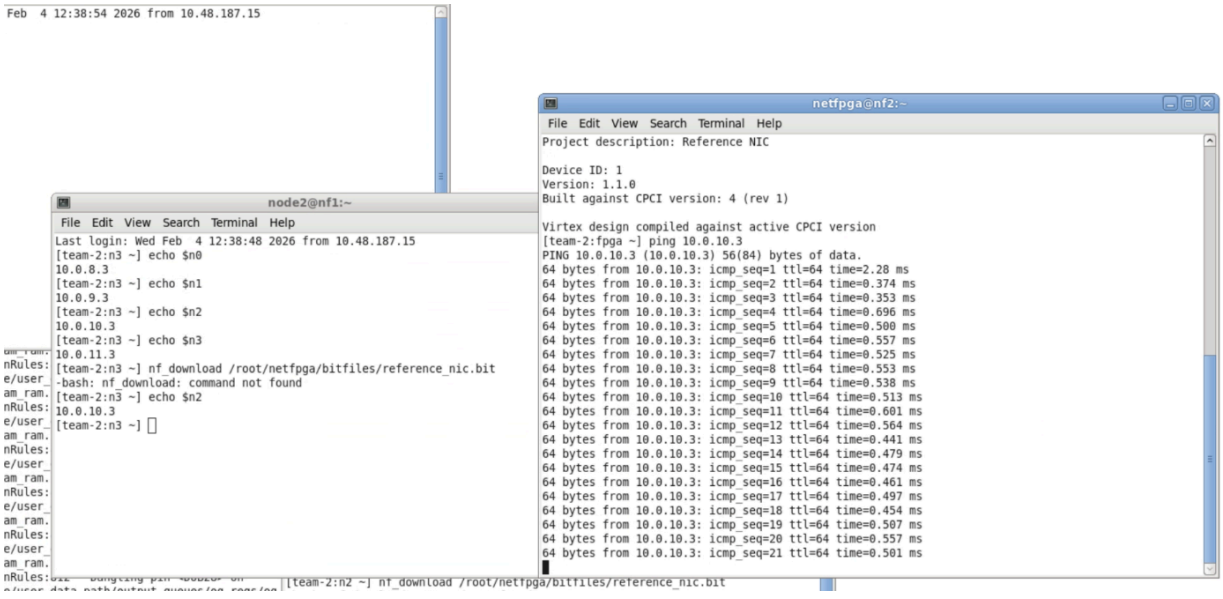


Figure 4. Terminal showing the output of echo $n0 ... $n3

4. Reference NIC Test (Linux Kernel IP Router Behavior)

4.1 PROGRAMMING THE NETFPGA WITH REFERENCE NIC

On the FPGA node, we load the reference NIC bitfile using *nf_download*. The expected output shows the NetFPGA device (nf2c0) and confirms successful programming.

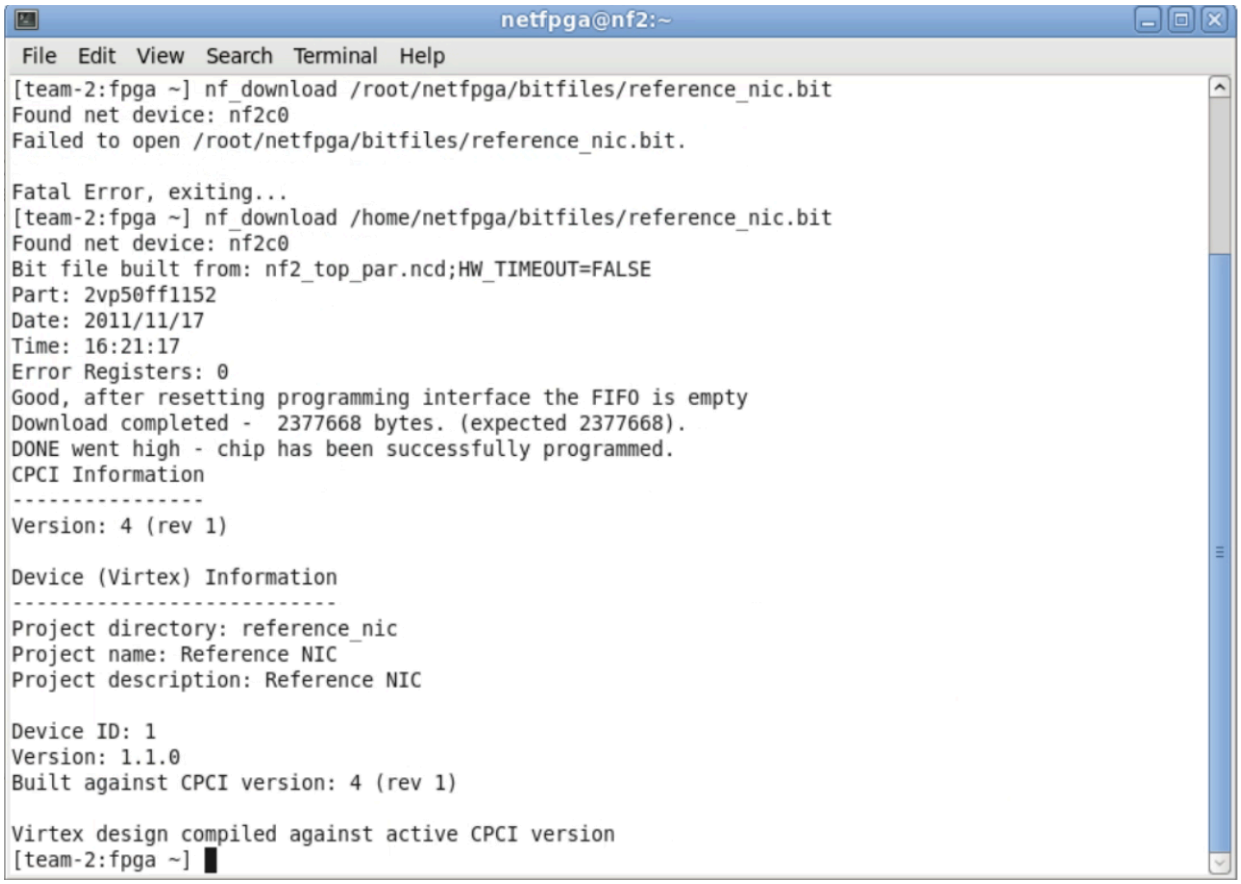


Figure 5. Output confirming programming success *(reference_nic)*

4.2 PING CONNECTIVITY ACROSS NODES

With Linux routing behavior under the reference NIC design, all nodes should be able to ping each other through the testbed topology.

```
[team-2:fpga ~] ping 10.0.8.3
PING 10.0.8.3 (10.0.8.3) 56(84) bytes of data.
64 bytes from 10.0.8.3: icmp_seq=1 ttl=64 time=0.593 ms
64 bytes from 10.0.8.3: icmp_seq=2 ttl=64 time=0.594 ms

--- 10.0.8.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.593/0.593/0.594/0.024 ms
[team-2:fpga ~] ping 10.0.9.3
PING 10.0.9.3 (10.0.9.3) 56(84) bytes of data.
64 bytes from 10.0.9.3: icmp_seq=1 ttl=64 time=0.566 ms
64 bytes from 10.0.9.3: icmp_seq=2 ttl=64 time=0.276 ms
64 bytes from 10.0.9.3: icmp_seq=3 ttl=64 time=0.500 ms

--- 10.0.9.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.276/0.447/0.566/0.125 ms
[team-2:fpga ~] ping 10.0.10.3
PING 10.0.10.3 (10.0.10.3) 56(84) bytes of data.
64 bytes from 10.0.10.3: icmp_seq=1 ttl=64 time=0.562 ms
64 bytes from 10.0.10.3: icmp_seq=2 ttl=64 time=0.572 ms
64 bytes from 10.0.10.3: icmp_seq=3 ttl=64 time=0.486 ms

--- 10.0.10.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.486/0.540/0.572/0.038 ms
[team-2:fpga ~] ping 10.0.11.3
PING 10.0.11.3 (10.0.11.3) 56(84) bytes of data.
64 bytes from 10.0.11.3: icmp_seq=1 ttl=64 time=0.589 ms
64 bytes from 10.0.11.3: icmp_seq=2 ttl=64 time=0.518 ms
64 bytes from 10.0.11.3: icmp_seq=3 ttl=64 time=0.510 ms
64 bytes from 10.0.11.3: icmp_seq=4 ttl=64 time=0.501 ms

--- 10.0.11.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.501/0.529/0.589/0.041 ms
[team-2:fpga ~]
```

Figure 6. Ping tests showing successful replies between nodes

4.3 BASELINE PERFORMANCE TEST WITH *IPERF*

We run a baseline throughput test using iperf to measure achievable bandwidth before enabling NetFPGA router acceleration. This baseline is used for later comparison.

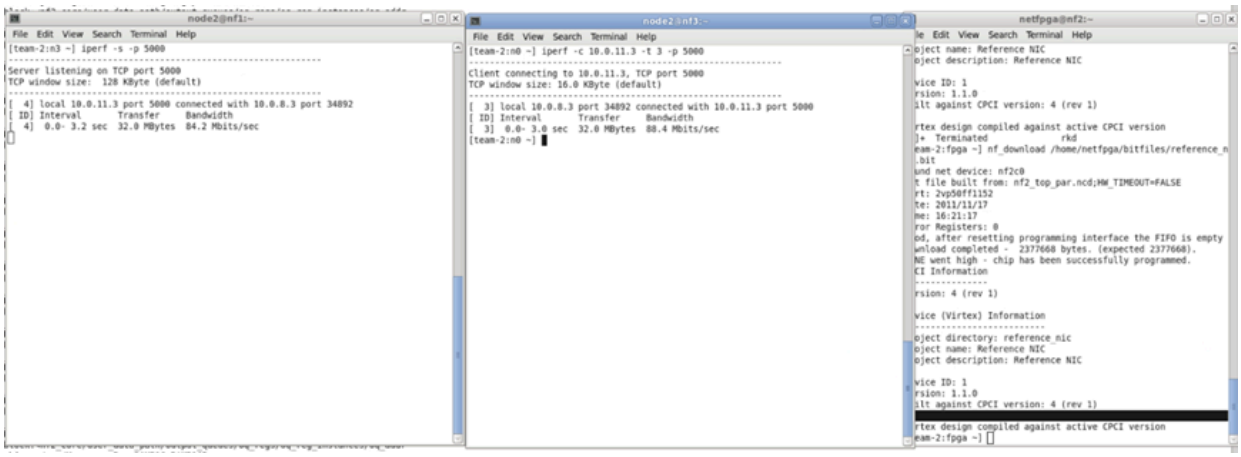


Figure 7. Baseline throughput using Reference NIC (Software Routing)

5. Reference Router Test with RKD Hardware Acceleration

5.1 PROGRAMMING THE NETFPGA WITH REFERENCE ROUTER

We load the reference router bitfile on the FPGA node using *nf_download*. After loading the router bitfile, we start the *rkd* daemon in the background so that routing table entries are inserted into the NetFPGA hardware.

```
[team-2:fpga ~] nf_download /home/netfpga/bitfiles/reference_router.bit
Found net device: nf2c0
Bit file built from: nf2_top_par.ncd;HW_TIMEOUT=FALSE
Part: 2vp50ff1152
Date: 2011/11/17
Time: 17:49:43
Error Registers: 0
Good, after resetting programming interface the FIFO is empty
Download completed -  2377668 bytes. (expected 2377668).
DONE went high - chip has been successfully programmed.
CPCI Information
----------------
Version: 4 (rev 1)

Device (Virtex) Information
---------------------------
Project directory: reference_router
Project name: Reference router
Project description: Reference IPv4 router

Device ID: 2
Version: 1.0.0
Built against CPCI version: 4 (rev 1)

Virtex design compiled against active CPCI version
[team-2:fpga ~] rkd &
[1] 7317
```

Figure 8. Output confirming programming success *(reference_router)*

5.2 PERFORMANCE COMPARISON (BASELINE VS. ROUTER ACCELERATION)

We re-run *iperf* tests after enabling *rkd* and compared throughput results. In general, hardware acceleration reduces CPU involvement in routing and can increase achievable throughput, especially under heavier traffic loads.

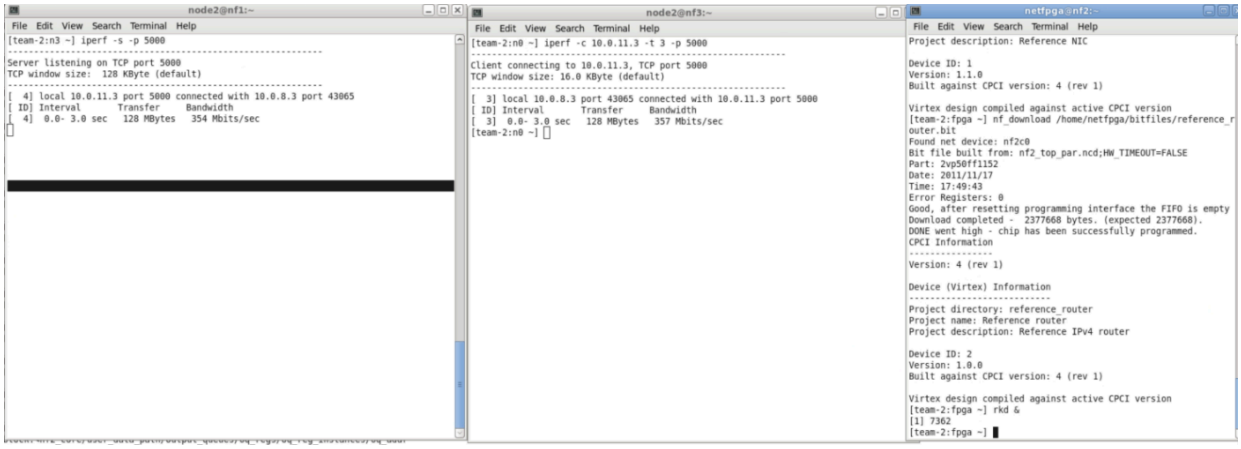


Figure 9. Accelerated throughput using Reference Router *(Hardware Routing)*

5.3 SMALL-PACKET STRESS TEST (UDP, 512B PACKETS, ≥30 SECONDS)

We stress per-packet processing overhead by running UDP iperf with 512-byte packets for at least 30 seconds per run. First, we developed a shell script to automate the testing process. This script launches multiple iperf clients in the background such that each NetFPGA port carries close to 1 Gbps traffic in each direction, and start them in the background to make flows begin approximately simultaneously. Then we repeat the experiment multiple times and compute the average throughput from server outputs. From the server-side results, we compute the total aggregate bandwidth through the NetFPGA by summing throughputs across ports/directions:

```
--- iperf result: nf0 → nf1 (10.0.11.3:5004) ---
------------------------------------------------------------
Client connecting to 10.0.11.3, UDP port 5004
Sending 512 byte datagrams
UDP buffer size: 2.00 MByte (default)
------------------------------------------------------------
[  3] local 10.0.10.3 port 57720 connected with 10.0.11.3 port 5004
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec  58.3 MBytes   244 Mbits/sec
[  3]  2.0- 4.0 sec  57.3 MBytes   240 Mbits/sec
[  3]  4.0- 6.0 sec  56.9 MBytes   239 Mbits/sec
[  3]  6.0- 8.0 sec  59.6 MBytes   250 Mbits/sec
[  3]  8.0-10.0 sec  60.7 MBytes   254 Mbits/sec
[  3] 10.0-12.0 sec  60.3 MBytes   253 Mbits/sec
[  3] 12.0-14.0 sec  59.6 MBytes   250 Mbits/sec
[  3] 14.0-16.0 sec  59.2 MBytes   248 Mbits/sec
[  3] 16.0-18.0 sec  58.6 MBytes   246 Mbits/sec
[  3] 18.0-20.0 sec  58.1 MBytes   244 Mbits/sec
[  3] 20.0-22.0 sec  57.3 MBytes   240 Mbits/sec
[  3] 22.0-24.0 sec  56.6 MBytes   238 Mbits/sec
[  3] 24.0-26.0 sec  60.9 MBytes   256 Mbits/sec
[  3] 26.0-28.0 sec  60.6 MBytes   254 Mbits/sec
[  3] 28.0-30.0 sec  60.1 MBytes   252 Mbits/sec
[  3]  0.0-30.0 sec   884 MBytes   247 Mbits/sec
[  3] Sent 1810616 datagrams
[  3] Server Report:
[  3]  0.0-30.0 sec   314 MBytes  87.7 Mbits/sec   0.649 ms 1167719/1810615 (64%)
[  3]  0.0-30.0 sec  6 datagrams received out-of-order
>>> Completed: nf0 → nf1 (10.0.11.3:5004) | BW: 87.7 Mbits/sec | Loss: 64% | Jitter: 0.649 ms

Stopping iperf server on nf1:5004
```

Figure 10. Small-packet stress test (UDP, 512B packets, ≥30 seconds)

Condition	Packet size	Total aggregate BW (Mbps)	Performance Impact
NIC	512B	157.1	**~42% decrease** vs Large Packets
NIC	1472B	272.4	Baseline
ROUTER	512B	286	**~63% decrease** vs Large Packets
ROUTER	1472B	765	**~2.8x speedup** vs NIC

Table 1. Per-run results and the average (Mbps/Gbps)

Based on the repeated UDP iperf runs in our four logs (NIC vs. Router+RKD, and 512B vs. 1472B payloads), we evaluated the system performance in two ways:

(1) Total Aggregate Throughput: We estimated the total observable throughput through the NetFPGA by summing the bandwidth of simultaneous flows across the aggregate test phases (Unidirectional and Bidirectional);

(2) Individual Flow Baselines: Additionally, we characterized the network topology by testing all 12 unique cross-node flows sequentially to verify individual link connectivity and baseline capacity.

The test results clearly demonstrate the performance trade-offs between hardware/software routing and packet sizes. In the Reference NIC mode (software routing), the total observable bandwidth was approximately 272.4 Mbps with large packets, but this dropped significantly to 157.1 Mbps when using small (512-byte) packets. The Reference Router (hardware accelerated) showed superior performance, achieving 765 Mbps with large packets—a nearly 3x improvement over the software baseline. However, even the hardware router experienced a performance drop to 286 Mbps with small packets.

- Why small packets stress the system?

Using small packets (512 bytes) drastically reduces throughput because network processing costs are primarily incurred **per packet**, not per byte. Every packet requires a fixed amount of overhead for interrupt handling (in software mode), header parsing, route lookup, and framing regardless of its payload size:

(1) In Software (NIC Mode): The CPU must process an interrupt and traverse the network stack for every single packet. Sending the same amount of data using smaller packets multiplies the number of CPU interrupts, overwhelming the processor and causing the throughput to plummet (from 272 Mbps to 157 Mbps);

(2) In Hardware (Router Mode): While the FPGA pipeline is much faster, it still has a finite Packet Per Second (PPS) limit. When the packet size is small, the data rate (Mbps) required to saturate the link corresponds to a much higher PPS rate. If the arrival rate of these small packets exceeds the processing frequency of the router pipeline or the arbiter, the effective bandwidth decreases, as seen in the drop from 765 Mbps to 286 Mbps.

- Is it fair to say that the NetFPGA can route IP traffic bi-directionally at line speed for a total of 4Gbps?

Based strictly on the empirical data collected in this experiment, it is not fair to claim we observed line-speed routing, as our measured throughput (286 Mbps for small packets, and 765 Mbps for large packets) falls short of the theoretical 4 Gbps capacity. However, this result does not imply the NetFPGA is incapable of line-speed routing. The limitation observed here is a bottleneck in the testbed environment (Host CPU and PCI bus) rather than the NetFPGA hardware itself. The host computers

used as traffic generators cannot generate small (512-byte) UDP packets fast enough to saturate the Gigabit links. Therefore, while the NetFPGA architecture is designed for 4Gbps wire-speed forwarding, our test setup prevents us from fully demonstrating that capability with small packets.

6. Initial Attempt in Synthesizing & Testing the Mini-IDS NetFPGA

6.1 BITFILE GENERATION AND FPGA PROGRAMMING

After integrating the Mini-IDS into the NetFPGA project, we generate a new bitfile through the standard NetFPGA synthesis flow. The design is then programmed onto the NetFPGA board using *nf_download*. The console output confirmed that the device *nf2c0* was found and that programming completed successfully. The downloaded design is identified with project metadata such as Project directory: lab4_dummy, Project name: IDS, and Project description: IDS Router, indicating that our custom IDS-enabled router image is running on the FPGA.

```
[team-2:fpga bitfiles] nf_download nf2_top_par.bit
Found net device: nf2c0
Bit file built from: nf2_top_par.ncd;HW_TIMEOUT=FALSE
Part: 2vp50ff1152
Date: 2026/ 2/ 5
Time: 22:31:41
Error Registers: 0
Good, after resetting programming interface the FIFO is empty
Download completed -  2377668 bytes. (expected 2377668).
^[[A^[[ADONE went high - chip has been successfully programmed.
^[[A^[[ACPCI Information
---------------
Version: 4 (rev 1)

Device (Virtex) Information
--------------------------
Project directory: lab4_dummy
Project name: IDS
Project description: IDS Router

Device ID: 102
Version: 0.1.0
Built against CPCI version: 4 (rev 1)

Virtex design compiled against active CPCI version
[team-2:fpga bitfiles] ping 10.0.8.3
PING 10.0.8.3 (10.0.8.3) 56(84) bytes of data.

--- 10.0.8.3 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 4999ms

[team-2:fpga bitfiles] rkd &
[2] 18587
[team-2:fpga bitfiles] ping 10.0.8.3
PING 10.0.8.3 (10.0.8.3) 56(84) bytes of data.
64 bytes from 10.0.8.3: icmp_seq=1 ttl=64 time=2.19 ms
64 bytes from 10.0.8.3: icmp_seq=2 ttl=64 time=0.315 ms
64 bytes from 10.0.8.3: icmp_seq=3 ttl=64 time=0.337 ms
64 bytes from 10.0.8.3: icmp_seq=4 ttl=64 time=0.302 ms
64 bytes from 10.0.8.3: icmp_seq=5 ttl=64 time=0.308 ms
64 bytes from 10.0.8.3: icmp_seq=6 ttl=64 time=0.348 ms
64 bytes from 10.0.8.3: icmp_seq=7 ttl=64 time=0.633 ms

--- 10.0.8.3 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6004ms
rtt min/avg/max/mdev = 0.302/0.633/2.194/0.646 ms
[team-2:fpga bitfiles] 
```

Figure 11. FPGA programmed with IDS Router bitfile

6.2 INITIAL CONNECTIVITY ISSUE AFTER PROGRAMMING

During the first connectivity check, we observe that ICMP connectivity could fail even though programming succeeds. Specifically, *ping 10.0.8.3* showed 100% packet loss, suggesting that routing state has not been populated into the NetFPGA router tables yet.

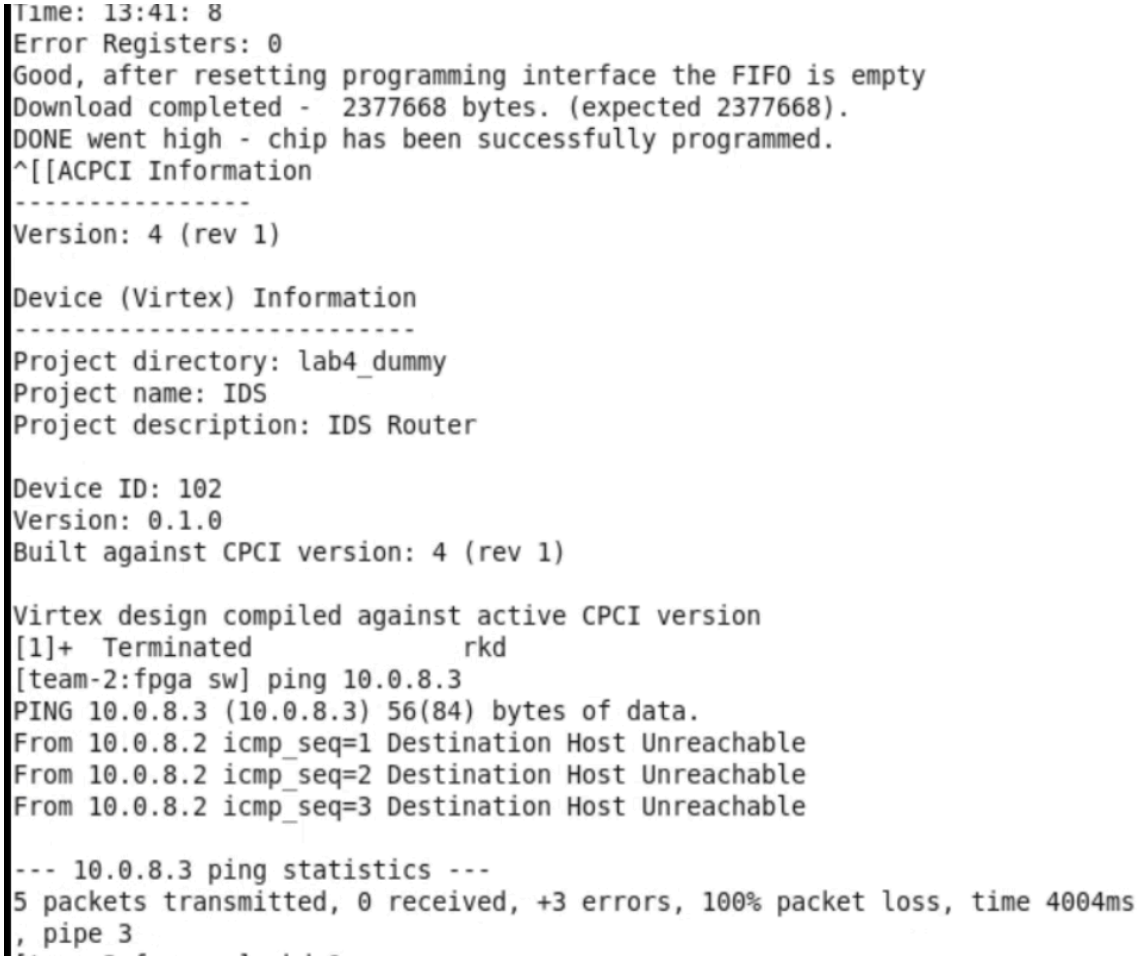


Figure 12. Ping failure immediately after programming

6.3 RECOVERY AND DEPENDENCY ON *RKD*

After starting the route kernel daemon (*rkd &*), the same ping test became successful with 0% packet loss, demonstrating that the datapath works correctly once routing table updates are actively pushed to hardware. We also confirmed the reverse condition: if *rkd* is terminated or not running, the node can report “Destination Host Unreachable,” and connectivity breaks again. This indicates that our IDS-enabled router design relies on *rkd* to synchronize Linux routing information into the NetFPGA hardware routing tables.

```
[team-2:fpga sw] rkd &
[1] 23832
[team-2:fpga sw] ping 10.0.8.3
PING 10.0.8.3 (10.0.8.3) 56(84) bytes of data.
64 bytes from 10.0.8.3: icmp_seq=1 ttl=64 time=1.67 ms
64 bytes from 10.0.8.3: icmp_seq=2 ttl=64 time=0.365 ms
64 bytes from 10.0.8.3: icmp_seq=3 ttl=64 time=0.370 ms
64 bytes from 10.0.8.3: icmp_seq=4 ttl=64 time=0.352 ms
64 bytes from 10.0.8.3: icmp_seq=5 ttl=64 time=0.370 ms
64 bytes from 10.0.8.3: icmp_seq=6 ttl=64 time=0.314 ms
64 bytes from 10.0.8.3: icmp_seq=7 ttl=64 time=0.501 ms
64 bytes from 10.0.8.3: icmp_seq=8 ttl=64 time=0.525 ms
64 bytes from 10.0.8.3: icmp_seq=9 ttl=64 time=0.529 ms
64 bytes from 10.0.8.3: icmp_seq=10 ttl=64 time=0.482 ms
^C64 bytes from 10.0.8.3: icmp_seq=11 ttl=64 time=0.466 ms

--- 10.0.8.3 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 10002ms
rtt min/avg/max/mdev = 0.314/0.540/1.670/0.365 ms
[team-2:fpga sw]
```

Figure 13. Connectivity restored after starting *rkd*

7. Register Interface Integration and *idsreg* Configuration

7.1 INITIAL PASSTHROUGH IDS MODULE INTEGRATION

Before enabling pattern matching and drops, we first integrate a passthrough *ids.v* into the *reference_router* pipeline and confirm that the design still forwards packets correctly. This stage validates file placement, build integration, and module interfaces while keeping behavior identical to the reference router.

7.2 REGISTER BLOCK INSTANTIATION WITH *GENERIC_REGS*

To enable runtime control of the Mini-IDS, we integrate the IDS block into the NetFPGA register framework using *generic_regs*. In our design, the IDS register block is instantiated with a dedicated tag and address width parameters so it appears as a standard memory-mapped NetFPGA peripheral. The instantiation shows *NUM_SOFTWARE_REGS = 3* (used for IDS command and signature pattern registers) and *NUM_HARDWARE_REGS = 1* (used for a hardware-visible status/counter register), matching the intended workflow of software configuration plus hardware reporting.

7.3 ADDRESS MAP PARAMETERS FOR IDS REGISTERS

We verify that the IDS addressing parameters match the generated NetFPGA address map. In the definitions, *IDS_BLOCK_ADDR* is assigned (e.g., 19'h0000c), and related widths (*IDS_BLOCK_ADDR_WIDTH*, *IDS_REG_ADDR_WIDTH*) are defined consistently. This is important because the *idsreg* script must access the correct base address; otherwise, reads/writes would not reach the IDS hardware.

7.4 SOFTWARE REGISTER PACKING AND BUS WIDTH RELATIONSHIP

The code snippets also show how the software register vector is constructed and sized, including the connection *.software_regs({ids_cmd, pattern_low, pattern_high})*. Another snippet highlights the sizing rule for *SOFTWARE_REGS_WIDTH*, which depends on *NUM_SOFTWARE_REGS* and the NetFPGA register bus data width (*CPCI_NF2_DATA_WIDTH*, shown as 32). The *generate* logic demonstrates that when *NUM_SOFTWARE_REGS > 0*, the *generic_sw_regs* module is instantiated and the *software_regs* signals are expanded onto the register interface.

```
`define DRAM_BLOCK_ADDR_WIDTH                1
`define DRAM_REG_ADDR_WIDTH                  24
`define IDS_BLOCK_ADDR_WIDTH                 19
`define IDS_REG_ADDR_WIDTH                   4
`define IN_ARB_BLOCK_ADDR_WIDTH              17
`define IN_ARB_REG_ADDR_WIDTH                6
```

```
`define STRIP_HEADERS_BLOCK_ADDR  17'h00001
`define IN_ARB_BLOCK_ADDR         17'h00002
`define IDS_BLOCK_ADDR            19'h0000c
`define OQ_BLOCK_ADDR             13'h0001
`define DRAM_BLOCK_ADDR           1'h1
```

```
`define CPCI_NF2_ADDR_WIDTH                  27

// CPCI--Virtex data bus width
`define CPCI_NF2_DATA_WIDTH                  32

// DMA data bus width
`define DMA_DATA_WIDTH                       32
```

```
generic_regs
#(
   .UDP_REG_SRC_WIDTH   (UDP_REG_SRC_WIDTH),
   .TAG                 (`IDS_BLOCK_ADDR),          // Tag -- eg. MODULE_TAG
   .REG_ADDR_WIDTH      (`IDS_REG_ADDR_WIDTH),     // Width of block addresses -- eg
   .NUM_COUNTERS        (0),                 // Number of counters
   .NUM_SOFTWARE_REGS   (3),                 // Number of sw regs
   .NUM_HARDWARE_REGS   (1)                  // Number of hw regs
) module_regs (
   .reg_req_in       (reg_req_in),
   .reg_ack_in       (reg_ack_in),
   .reg_rd_wr_L_in   (reg_rd_wr_L_in),
   .reg_addr_in      (reg_addr_in),
   .reg_data_in      (reg_data_in),
   .reg_src_in       (reg_src_in),

   .reg_req_out      (reg_req_out),
   .reg_ack_out      (reg_ack_out),
   .reg_rd_wr_L_out  (reg_rd_wr_L_out),
   .reg_addr_out     (reg_addr_out),
   .reg_data_out     (reg_data_out),
   .reg_src_out      (reg_src_out),

   // --- counters interface
   .counter_updates  (),
   .counter_decrement(),

   // --- SW regs interface
   .software_regs    ({ids_cmd,pattern_low,pattern_high}),
```

```
    NUM_COUNTERS > 0 ? NUM_COUNTERS * INSTANCES : INSTANCES,
  parameter SOFTWARE_REGS_WIDTH =
    NUM_SOFTWARE_REGS > 0 ? NUM_SOFTWARE_REGS * `CPCI_NF2_DATA_WIDTH * INSTANCES : INSTANCES,
  parameter HARDWARE_REGS_WIDTH =
    NUM_HARDWARE_REGS > 0 ? NUM_HARDWARE_REGS * `CPCI_NF2_DATA_WIDTH * INSTANCES : INSTANCES
)
```

```
// --- SW regs interface
output [SOFTWARE_REGS_WIDTH - 1 : 0]   software_regs, // signals from the software
```

```
generate
if (NUM_SOFTWARE_REGS > 0) begin
   generic_sw_regs
     #(.UDP_REG_SRC_WIDTH   (UDP_REG_SRC_WIDTH),
       .TAG                 (TAG),
       .REG_ADDR_WIDTH      (REG_ADDR_WIDTH),
       .NUM_REGS_USED       (NUM_SOFTWARE_REGS * INSTANCES),
       .REG_START_ADDR      (REG_START_ADDR + NUM_COUNTERS * INSTANCES))
   generic_sw_regs
     (
      .reg_req_in        (cntr_reg_req_out),
      .reg_ack_in        (cntr_reg_ack_out),
      .reg_rd_wr_L_in    (cntr_reg_rd_wr_L_out),
      .reg_addr_in       (cntr_reg_addr_out),
      .reg_data_in       (cntr_reg_data_out),
      .reg_src_in        (cntr_reg_src_out),

      .reg_req_out       (sw_reg_req_out),
      .reg_ack_out       (sw_reg_ack_out),
      .reg_rd_wr_L_out   (sw_reg_rd_wr_L_out),
      .reg_addr_out      (sw_reg_addr_out),
      .reg_data_out      (sw_reg_data_out),
      .reg_src_out       (sw_reg_src_out),

      .software_regs     (software_regs_expanded),

      .clk               (clk),
      .reset             (reset));
end
else begin
```

Figure 14. Verilog snippets for IDS register integration and address mapping

7.5 IMPLEMENTATION AND TESTING USING THE PASSTHROUGH LOGIC

After we understand the design logic of the interface of creating the hardware and software interface, we implement the passthrough logic by applying a reverse of given pattern. As shown in below code block, we reduce the number of software register to 2 (ids_cmd and ids_sw_reg):

```
////////////////////////////////////////////////////////////////////////////////
// vim:set shiftwidth=3 softtabstop=3 expandtab:
// $Id: module_template 2008-03-13 gac1 $
//
```

```
// Module: ids.v
// Project: NF2.1
// Description: Defines a simple ids module for the user data path.  The
// modules reads a 64-bit register that contains a pattern to match and
// counts how many packets match.  The register contents are 7 bytes of
// pattern and one byte of mask.  The mask bits are set to one for each
// byte of the pattern that should be included in the mask -- zero bits
// mean "don't care".
//
////////////////////////////////////////////////////////////////////////////
module ids
   #(
      parameter DATA_WIDTH = 64,
      parameter CTRL_WIDTH = DATA_WIDTH/8,
      parameter UDP_REG_SRC_WIDTH = 2
   )
   (
      input  [DATA_WIDTH-1:0]             in_data,
      input  [CTRL_WIDTH-1:0]             in_ctrl,
      input                               in_wr,
      output                              in_rdy,
      output [DATA_WIDTH-1:0]             out_data,
      output [CTRL_WIDTH-1:0]             out_ctrl,
      output                              out_wr,
      input                               out_rdy,

      // --- Register interface
      input                               reg_req_in,
      input                               reg_ack_in,
      input                               reg_rd_wr_L_in,
      input  [`UDP_REG_ADDR_WIDTH-1:0]    reg_addr_in,
      input  [`CPCI_NF2_DATA_WIDTH-1:0]   reg_data_in,
      input  [UDP_REG_SRC_WIDTH-1:0]      reg_src_in,
      output                              reg_req_out,
      output                              reg_ack_out,
      output                              reg_rd_wr_L_out,
      output  [`UDP_REG_ADDR_WIDTH-1:0]   reg_addr_out,
      output  [`CPCI_NF2_DATA_WIDTH-1:0]  reg_data_out,
      output  [UDP_REG_SRC_WIDTH-1:0]     reg_src_out,
      // misc
      input                                reset,
      input                                clk
   );
   // Define the log2 function
   // `LOG2_FUNC
   //------------------------ Signals-------------------------------


   // software registers
   wire [31:0]                   ids_sw_reg;
   wire [31:0]                   ids_cmd;
   // hardware registers
   reg [31:0]                    ids_hw_reg;

   generic_regs
```

```
    #(
       .UDP_REG_SRC_WIDTH   (UDP_REG_SRC_WIDTH),
       .TAG                 (`IDS_BLOCK_ADDR),          // Tag -- eg. MODULE_TAG
       .REG_ADDR_WIDTH      (`IDS_REG_ADDR_WIDTH),     // Width of block addresses -- e
       .NUM_COUNTERS        (0),                 // Number of counters
       .NUM_SOFTWARE_REGS   (2),                 // Number of sw regs
       .NUM_HARDWARE_REGS   (1)                  // Number of hw regs
    ) module_regs (
       .reg_req_in       (reg_req_in),
       .reg_ack_in       (reg_ack_in),
       .reg_rd_wr_L_in   (reg_rd_wr_L_in),
       .reg_addr_in      (reg_addr_in),
       .reg_data_in      (reg_data_in),
       .reg_src_in       (reg_src_in),
       .reg_req_out      (reg_req_out),
       .reg_ack_out      (reg_ack_out),
       .reg_rd_wr_L_out  (reg_rd_wr_L_out),
       .reg_addr_out     (reg_addr_out),
       .reg_data_out     (reg_data_out),
       .reg_src_out      (reg_src_out),
       // --- counters interface
       .counter_updates  (),
       .counter_decrement(),
       // --- SW regs interface
       .software_regs    ({ids_cmd,ids_sw_reg}),
       // --- HW regs interface
       .hardware_regs    (ids_hw_reg),
       .clk              (clk),
       .reset            (reset)
     );
    //------------------------ Logic-------------------------------

     assign out_data = in_data;
     assign out_ctrl =in_ctrl;
     assign out_wr = in_wr;
     assign in_rdy = out_rdy;
     always@(posedge clk)
     begin
     //match is set to the reverse order of the pattern.
       ids_hw_reg <= {ids_sw_reg[7:0],ids_sw_reg[15:8],ids_sw_reg[23:16],ids_sw_reg[31:
     end
endmodule
```

Meanwhile, we change the definition in ids.xml for defining proper registers:

```
<?xml version="1.0" encoding="UTF-8"?>
<nf:module xmlns:nf="http://www.NetFPGA.org/NF2_register_system" xmlns:xsi="http://www
    <nf:name>ids</nf:name>
    <nf:prefix>ids</nf:prefix>
    <nf:location>udp</nf:location>
    <nf:description>Registers for IDS</nf:description>
    <nf:blocksize>64</nf:blocksize>
    <nf:registers>
        <nf:register>
```

```
          <nf:name>ids_sw_reg</nf:name>
            <nf:description>ids software register for pattern</nf:description>
            <nf:type>generic_software32</nf:type>
        </nf:register>
        <nf:register>
          <nf:name>ids_cmd</nf:name>
            <nf:description>Command Register</nf:description>
            <nf:type>generic_software32</nf:type>
        </nf:register>
            <nf:register>
          <nf:name>ids_hw_reg</nf:name>
            <nf:description>Register for storing reverse pattern</nf:description>
            <nf:type>generic_hardware32</nf:type>
        </nf:register>
    </nf:registers>
</nf:module>
```

By applying the changes in the previous design, we are able to obtain the definition in registers.v, which is automatically generated

```
// Name: ids
// Description: Registers for IDS
// File: projects/lab4_dummy/include/ids.xml
`define IDS_IDS_SW_REG  4'h0
`define IDS_IDS_CMD     4'h1
`define IDS_IDS_HW_REG  4'h2
```

and the correct corresponding address definition in header file:

```
// Name: ids (IDS)
// Description: Registers for IDS
// File: projects/lab4_dummy/include/ids.xml
#define IDS_IDS_SW_REG_REG   0x2000300
#define IDS_IDS_CMD_REG      0x2000304
#define IDS_IDS_HW_REG_REG   0x2000308
```

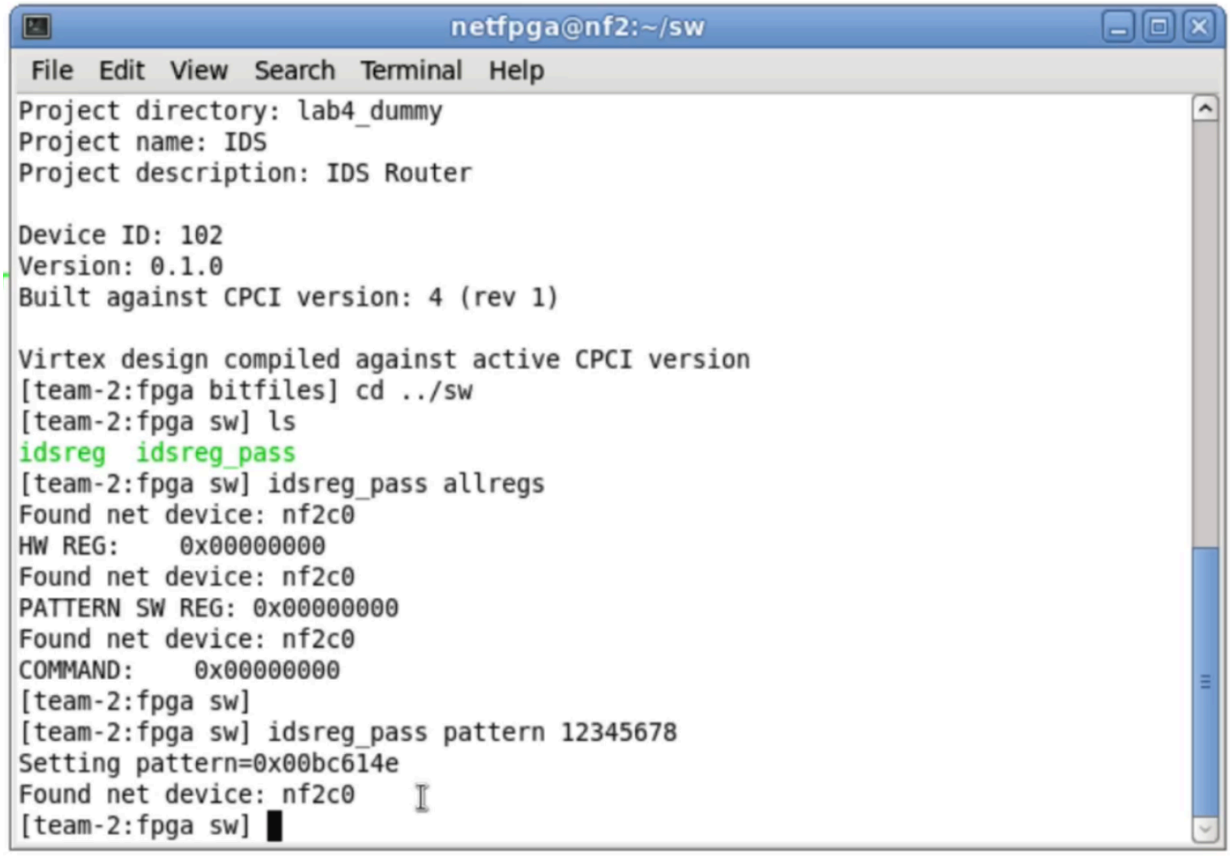
Also we write our own idsreg (idsreg_pass) for the passthrough design

```
#!/usr/bin/perl -w
use lib "/usr/local/netfpga/lib/Perl5";
use strict;

my $IDS_SW_REG = 0x2000300;
my $IDS_COMMAND_REG = 0x2000304;
my $IDS_HW_REG = 0x2000308;

sub regwrite {
   my( $addr, $value ) = @_;
   my $cmd = sprintf( "regwrite $addr 0x%08x", $value );
   my $result = `$cmd`;
   # print "Ran command '$cmd' and got result '$result'\n";
}
```

```
sub regread {
   my( $addr ) = @_;
   my $cmd = sprintf( "regread $addr" );
   my @out = `$cmd`;
   my $result = $out[0];
   if ( $result =~ m/Reg (0x[0-9a-f]+) \((\d+)\):\s+(0x[0-9a-f]+) \((\d+)\)/ ) {
      $result = $3;
   }
   return $result;
}

sub idsreset {
   regwrite( $IDS_COMMAND_REG, 0x1 );
   regwrite( $IDS_COMMAND_REG, 0x0 );
}

sub write_pattern {
   my($pattern) = @_;
   regwrite($IDS_SW_REG, $pattern);
}


sub usage {
   print "Usage: idsreg <cmd> <cmd options>\n";
   print "  Commands:\n";
   print "    reset                resets the matches counter\n";
   print "    pattern <value>      set a pattern in hex\n";
   print "    allregs              displays the value of all registers\n";
}

my $numargs = $#ARGV + 1;
if( $numargs < 1 ) {
   usage();
   exit(1);
}
my $cmd = $ARGV[0];
if ($cmd eq "reset") {
   idsreset();
} elsif ($cmd eq "pattern") {
   if ($numargs < 2) {
      usage();
      exit(1);
   }
   my $str = $ARGV[1];
   my $pattern = $str;
   printf("Setting pattern=0x%08x\n", $pattern);
   write_pattern($pattern);
}  elsif ($cmd eq "allregs") {
   print "HW REG:    ", regread( $IDS_HW_REG ), "\n";
   print "PATTERN SW REG: ", regread( $IDS_SW_REG ), "\n";
   print "COMMAND:    ", regread( $IDS_COMMAND_REG ), "\n";
} else {
   print "Unrecognized command $cmd\n";
   usage();
   exit(1)
```

```
}
```

With all been set, we flash the image to the netFPGA and we are getting the expected behavior.



Now we are confident about our methodology of create or modify the design registers with proven evidence of workable bitfile.

8. Initial Hardware Validation Results (Connectivity + IDS Register Sanity)

8.1 HARDWARE DATAPATH SANITY: FORWARDING DEPENDS ON *RED*

From the terminal results, the IDS router bitfile can be successfully programmed, but simple ICMP connectivity is not immediately guaranteed. The screenshots show a clear transition: ping fails right after programming (100% loss), but becomes successful after launching *rkd &*. When *rkd* is terminated, connectivity fails again with “Destination Host Unreachable.” This confirms that, in our initial test, the design behaves correctly only when rkd is running and the system is in a consistent routing/forwarding state.

8.2 IDS REGISTER ACCESS SANITY USING *IDSREG* (PATTERN PROGRAMMING + READBACK)

We then verify that IDS registers can be programmed and read back in hardware. Using *idsreg pattern hello*, the tool prints the pattern split into two words: hi = 0x7c68656c and lo = 0x6c6f0000. Running *idsreg allregs* shows readable register outputs including *MATCHES: 0x00000001 (ignore for this part)*, *PATTERN_HI: 0x7c68656c*, *PATTERN_LO: 0x6c6f0000*, and *COMMAND: 0x000004ae*. This confirms end-to-end register communication (software tool → NetFPGA register interface → IDS registers → software readback) works in our initial hardware test.

```
[team-2:fpga sw] idsreg pattern hello
Setting pattern hi=0x7c68656c, lo=0x6c6f0000
Found net device: nf2c0
Found net device: nf2c0
[team-2:fpga sw] idsreg allregs
Found net device: nf2c0
MATCHES:    0x00000001
Found net device: nf2c0
PATTERN_HI: 0x7c68656c
Found net device: nf2c0
PATTERN_LO: 0x6c6f0000
Found net device: nf2c0
COMMAND:    0x000004ae
```

Figure 15. idsreg Register Write and Read Test using "*idsreg allregs*"

9. Full Mini-IDS Functional Validation (Good vs Bad Traffic)

9.1 EXPERIMENT SETUP (PATTERN + RESET + TRAFFIC ROLES)

We first reset the IDS state and configure the detection pattern on the FPGA using *idsreg reset* followed by *idsreg pattern <STRING>*, so the hardware matcher is ready before traffic injection. According to the lab procedure, we then start an *iperf* server on each of the three host nodes and generate cross-node traffic by launching three *iperf* clients on each node sending to the other nodes. For each node, two client processes generate "good" traffic whose payload does not contain the IDS string, while the third client generates "bad" traffic by using *iperf -F <FILENAME>* to inject the same signature string or string contains the pattern into the payload. This traffic-role setup (2 good + 1 bad per node) is used consistently for both TCP and UDP experiments to validate that only signature-containing traffic is filtered while normal traffic continues to pass.

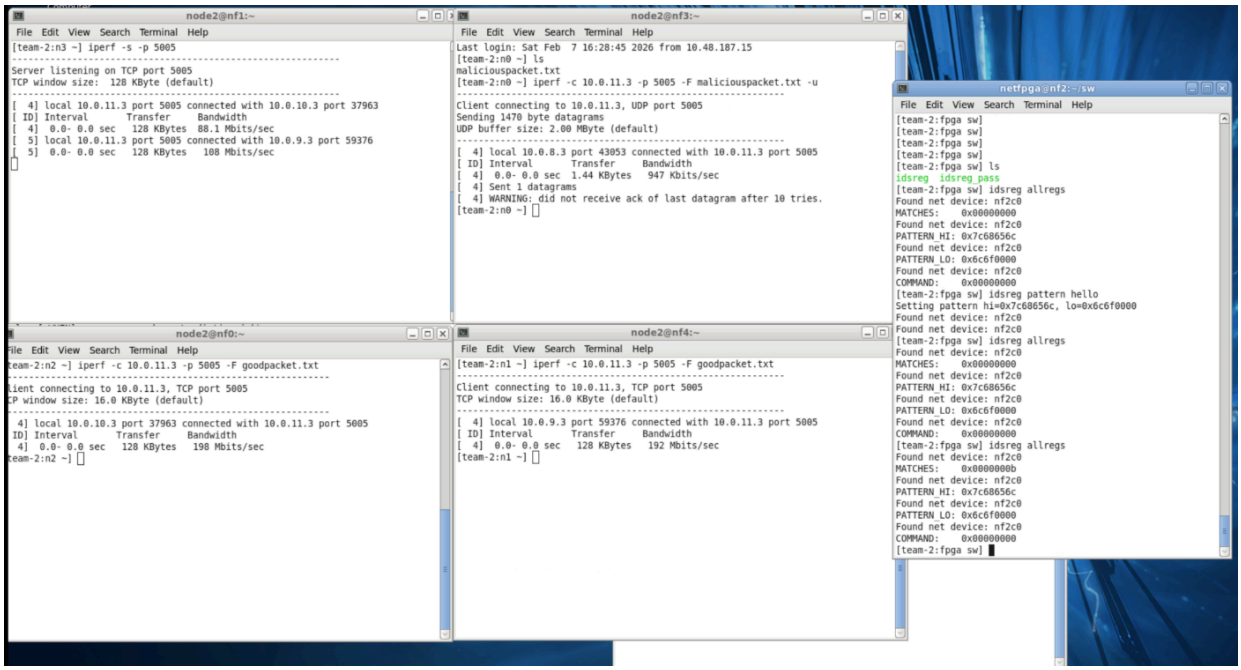


Figure 16. IDS functional test topology and client roles (2 good + 1 bad per node)

9.2 EXPECTED BEHAVIOR AND TCP VS UDP DIFFERENCE

The lab requires us to observe that “bad” traffic does not successfully traverse the network and to explicitly show the different expected outcomes in TCP versus UDP modes. In TCP mode, the bad client may still complete connection establishment (e.g., a TCP handshake), but the IDS should prevent signature-bearing payload packets from reaching the server, so the server-side TCP *iperf* output should show missing/zero application data associated with the bad stream compared with the good streams. In UDP mode, the expected behavior is stricter: the server should not receive any UDP packets from the bad client at all, so the UDP *iperf* server output should reflect contributions only from the good streams while the bad stream is absent. Demonstrating this protocol-dependent observation is essential because TCP can establish control-plane connection state even when data-plane payload is being filtered, whereas UDP visibility depends entirely on received datagrams.

Figure 17. TCP-mode behavior: bad client connects but no payload arrives

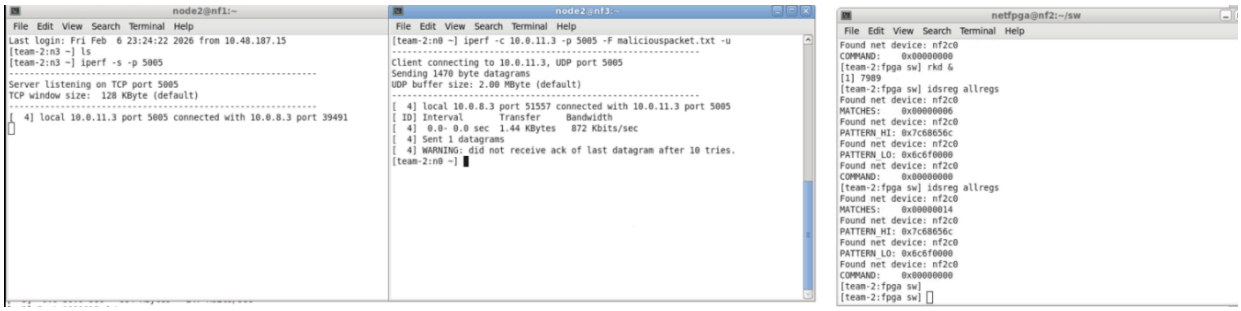


Figure 18. UDP-mode behavior: no packets reach server from bad client

9.3 DROPPED-PACKET EVIDENCE USING IDSREG MATCHES

Finally, to quantify IDS filtering and confirm the hardware counter path, we read the drop/match counter on the FPGA node using *idsreg matches*, which the lab specifies as the required indicator of dropped packets. We record a baseline *idsregmatches* value before starting the traffic, then run the full traffic experiment (including the bad clients injecting the signature string), and read *idsreg matches* again afterward. A clear increase in the counter provides direct evidence that the IDS is detecting the signature in the data stream and actively dropping packets, which complements the server-side *iperf* behavior and the *tcpdump* observation by proving that the loss is caused by the IDS rather than unrelated network issues.

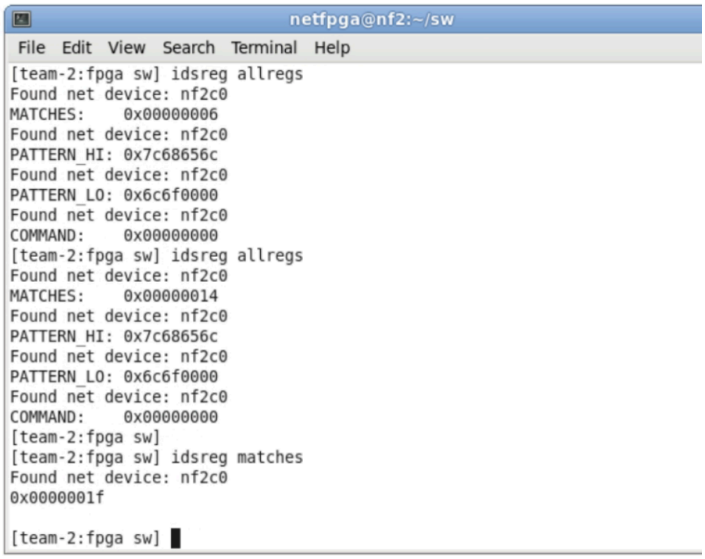


Figure 20. IDS drop counter evidence (idsreg matches before/after)

10. Conclusion

In this lab, we follow the NetFPGA workflow to compile and deploy both reference designs and our custom Mini-IDS router bitfile. We confirm that hardware routing acceleration depends on RKD to synchronize Linux routing information into the NetFPGA routing tables, and that ICMP connectivity becomes stable only after *rkd &* is running.

We integrate the Mini-IDS control plane through the NetFPGA register framework, validating address-map consistency and end-to-end register access using *idsreg* (pattern programming and readback). We then perform the required functional IDS validation using three iperf clients per node (two good + one bad using *-l*), verify with tcpdump that each server only receives packets from the two good clients, and confirm that the FPGA-side *idsreg matches* counter reflects dropped packets caused by bad traffic.

Overall, our results provide sufficient evidence that we successfully completed the required tasks and demonstrated an IDS-enabled NetFPGA router capable of filtering signature-matching flows while maintaining normal forwarding for benign traffic.

11. GitHub Repository and Team Contributions

GitHub Repository: https://github.com/jeremycai99/EE533-Labs/tree/main/Lab4

Contribution Summary:

- Jeremy Cai: Work on project builds and initial performance testing using simple CLI
- Prathamesh Hirekodi: Develop scripts for large scale UDP performance test
- Yanzhe Li: Performance test data analysis and reporting