ECE532 Final Report (Group)
Smart Network Switch

Group 9

Team members:
Mark Bugaisen
David Nguyen
Mahmoud Kharsa

Table of Contents

Overview

Background/Motivation

The Internet is made up of many interconnected devices that communicate with each other. To facilitate this communication, there is a large infrastructure covering the globe. One component of this infrastructure is the cables, satellites, etc. that transmit internet traffic. But this traffic needs to be routed to reach the right destination. This routing is done by devices called routers and network switches. Endpoint devices are usually connected to a local network, which may or may not be connected to the Internet. Routers are responsible for routing traffic between local networks, while switches route traffic within a local network.

Modern networks can operate on fibre optic cables carrying a bandwidth of 100 Gb/s [1] while having very strict latency constraints. This means that switches need to have routing functionality accelerated in hardware to meet these requirements. Modern switches also offer a multitude of security and network management features. Switches can also be connected to each other to create larger local networks and divide endpoint devices into groups of virtual local networks.

Switches make decisions on where to send traffic based on the MAC address table. This table contains a list of MAC addresses and corresponding switch port numbers. Upon receiving a packet, a lookup of the destination MAC is done. If a corresponding table entry is found, the packet is sent to the port. Otherwise, the packet is sent to all ports, and the table is updated once a response arrives. The table also allows for static entries to be added [2].

Our team decided on this project because we found it interesting and it matched the project's requirements. Networking is a subject that we are interested in, and we thought that doing this project is a great way to learn more about it. Also, building a device that facilitates communication rather than an endpoint device is a more unique experience. The performance requirements present in network switches means that we need to use hardware acceleration as part of our project. The switch itself is turned into an IoT device by adding remotely accessed features.

Goals

Our goal with this project was to design and build an IoT smart network switch. First, this switch needed to have the basic function of routing packets implemented in hardware. For performance, we set a minimum throughput of 1Mb/s and a maximum latency of 100ms as our worst-case targets. Next, we wanted the switch to allow for both ethernet and wireless connections, with support for at least 3 connected devices. For the IoT functionality, we wanted the switch to support at least two kinds of remote access, one read and one write. Some of the remote access features that we considered were received/dropped packets counters, read/update address table, and the ability to remotely disable ports.

Block Diagram

A block diagram of the smart network switch is shown in **Figure 1** which shows the major modules of the design. In addition, the block diagram from the Vivado IP Integrator is shown in **Figure 2** with key IP blocks labelled.

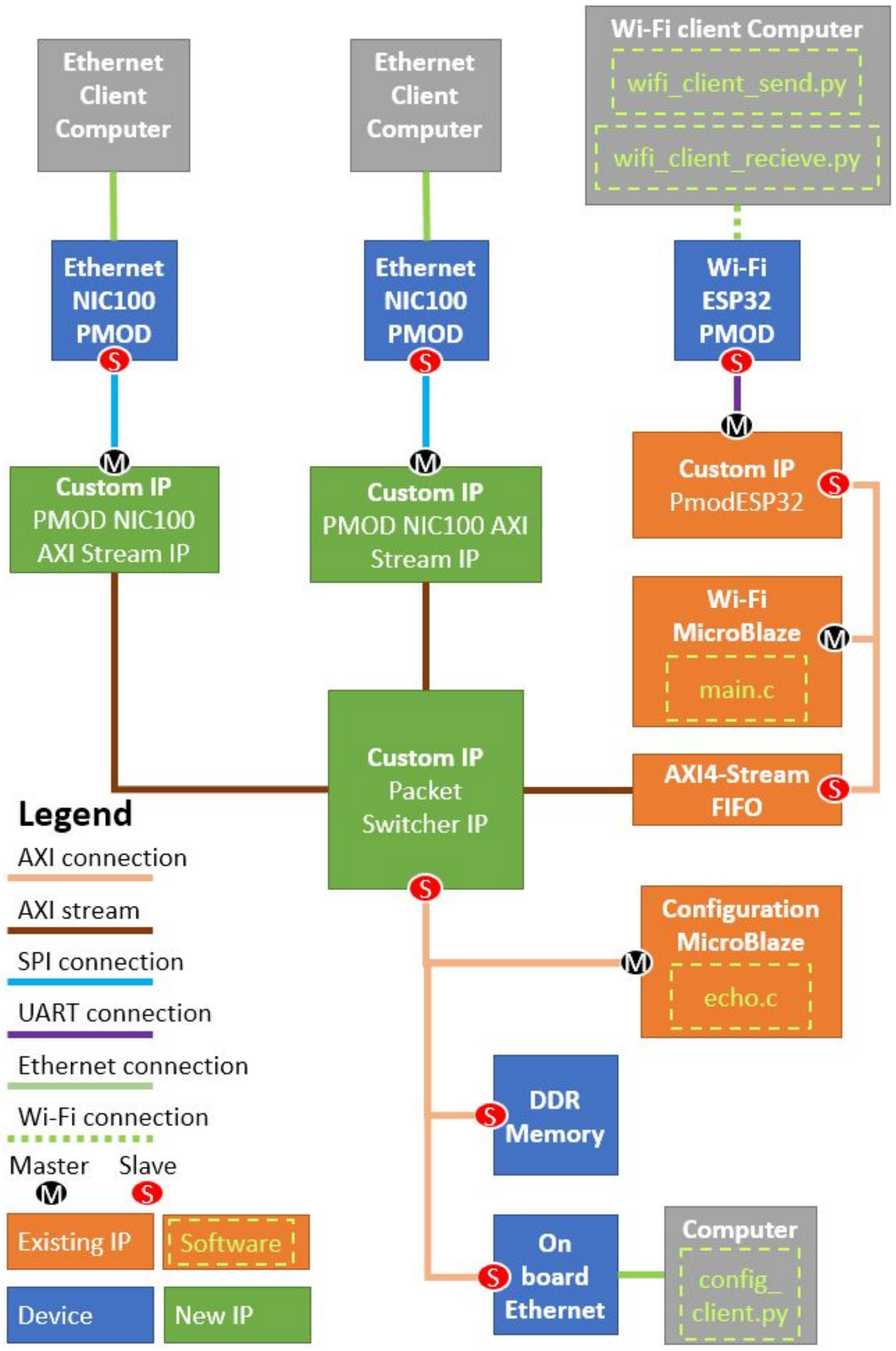
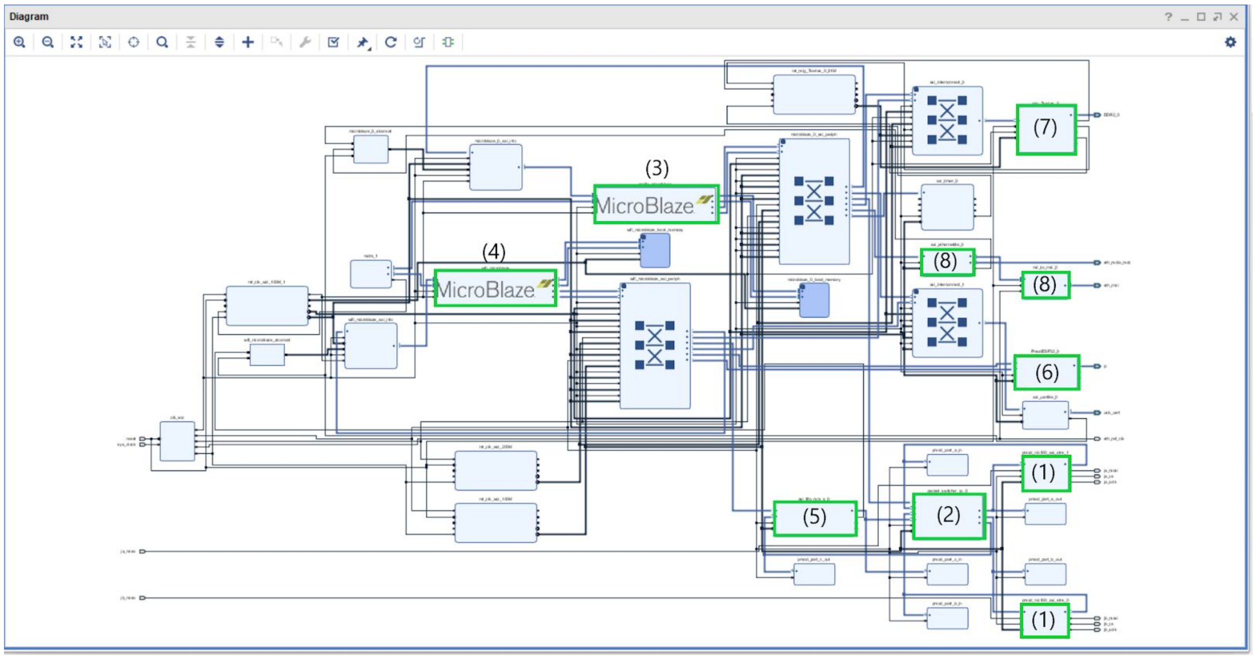


Figure 1: System-level Block Diagram



(1) 2x PMOD_NIC100_axi_stream_ip
(2) Packet_switcher_ip
(3) Configuration Microblaze
(4) Wifi PMOD Driver Microblaze
(5) AXI-Stream FIFO
(6) ESP32 PMOD
(7) MIG 7 Series (DDR)
(8) AXI Etherlite & Ethernet PHY MII to Reduced MII (Local Ethernet)

Figure 2: Block diagram from Vivado IP Integrator

Brief Description of IP

A description of each of the IP blocks used in the project is shown in **Table 1**. Other auto-generated IPs needed to support the Microblaze, DDR memory, and onboard ethernet have been omitted for clarity.

Table 1: IP Blocks used in the project

Module	Description	Origin
FPGA IPs		
pmod_nic100_axi_stream_ip (2x)	Custom IP: Manages the communication with the NIC100 PMOD.	Github[3] - Modified
packet_switcher_ip	Custom IP: Routes packets between 3 ports, where each port is 2 AXI-stream interfaces in opposite directions.	Original
-input_queue (3x)	Sub-module: Takes in an input AXI-stream, buffers incoming packets into a data FIFO and extracts packet destination IP address.	Original
--axis_meta_fifo	Xilinx IP: AXI4-Stream Data FIFO, used to buffer incoming packets.	Xilinx
--axis_data_fifo	Xilinx IP: AXI4-Stream Data FIFO, used to buffer incoming packets' destination addresses.	Xilinx
-packet_arbiter	Sub-module: Routes packets based on destination addresses and arbitrates between ports.	Original
--ip_port_mapper (3x)	Sub-module: Given an IP address, outputs a one-hot encoding of which port to use.	Original
--arbiter_scheduler (3x)	Sub-module: Round Robin arbiter for each output port	Original
--port_controller (3x)	Sub-module: State machine that manages sending data to the pmod_nic100_axi_stream_ip module.	Original
config_microblaze	MicroBlaze processor: manages the onboard ethernet and remote configuration feature.	Xilinx
-main.c	C code: Part of auto-generated echo server code.	Xilinx
-echo.c	C code: Based on the auto-generated echo server code. Modified to interpret and perform commands.	Xilinx - Modified

wifi_microblaze	MicroBlaze processor: manages communication with a Wifi device.	Xilinx
-main.c	C code: Based on the Digilent ESP32 PMOD example. Modified to configure PMOD as a wireless access point and facilitate AXI-stream communication with packet_switcher_ip module.	Digilent - Modified
axi_fifo	Xilinx IP: AXI4-Stream FIFO, used to convert between AXI-Lite interface (wifi_microblaze module) and AXI-Stream interface (packet_switcher_ip module).	Xilinx
PmodESP32	Custom IP: Digilent IP that manages communication with the ESP32 PMOD.	Digilent
Connected Devices		
2x Ethernet PMOD	PMOD NIC100: Ethernet interface with a built-in controller.	Digilent
Wifi PMOD	PMOD ESP32: wireless communication module that supports both Wifi and Bluetooth.	Digilent
Onboard DDR	Used to provide memory for the config_microblaze.	Digilent
Onboard ethernet	Used to connect to a computer/network to facilitate remote configuration.	Digilent
Remote Devices		
Computer connected to Wifi PMOD	Due to the functionality of the Wifi PMOD, devices connected to it must use the following scripts to send/receive packets.	N/A
-wifi_client_recieve.py	Receives UDP packets from the Wifi PMOD and extracts the payload.	Original
-wifi_client_send.py	Generates and sends UDP packets to the Wifi PMOD.	Original
Computer connected to onboard Ethernet	Computer used for remote configuration.	N/A
-config_client.py	Python script: Takes in commands and sends them to the config_microblaze	Original

Outcome

Final Results

We were successful in meeting all of the proposed requirements of the network switch, a table summarizing the requirements is shown in **Table 2**, while a screenshot showing the total utilization of the network switch on the Nexys DDR board is shown in **Figure 3**.

Our network switch is able to route IPv4 packets between three devices, two of which are connected through Ethernet while the third is connected through Wi-Fi. Due to limitations with the Wi-Fi PMOD only sending the packet payload and not the full packet, we had to implement a workaround where a wrapper script is run on the device connected by Wi-Fi to wrap its data inside a UDP packet. Therefore when the packet goes through the PMOD, it will treat the UDP wrapper as the payload and transmit the entire packet across the network switch.

In addition, our network switch is also able to be remotely configured through on-board Ethernet. This allows us to view statistics of the switch such as how many packets were sent or dropped on each port as well as change the mappings in the switch's static IP table to disable and enable ports accordingly.

Table 2: Outcome in achieving the proposed requirements

Requirement	**Achievement**
FPGA routes packets to the correct destination device (in hardware)	Completed
Support two devices connected across Ethernet	Completed
Support one device connected across Wi-Fi	Completed*
Support IPv4	Completed
Remotely configurable	Completed
Has monitoring capabilities	Completed
Minimum Throughput: 1 Mbps	Completed
Maximum Latency: 100 ms	Completed

* Workaround had to be implemented for completion

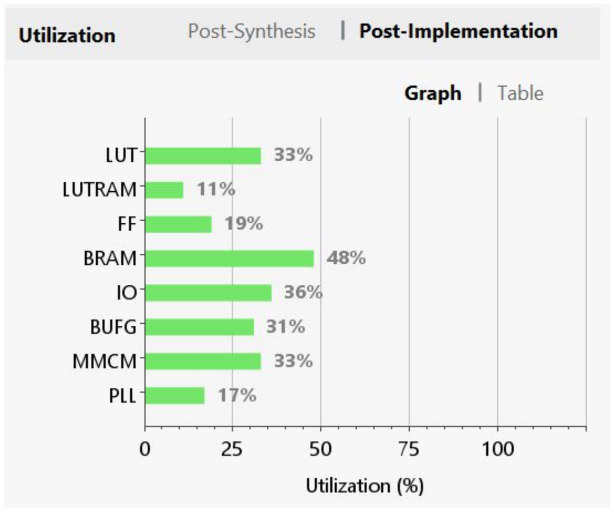


Figure 3: Device utilization of project on the FPGA

For throughput, the packet switcher IP operates on a 10MHz clock and has 8-bit wide buses for each interface. This means the packet switcher is capable of processing 80 Mbps for each port. The PMOD NIC100 IP operates on a 10MHz clock and uses the 1-bit SPI interface. This equals 10 Mbps for each ethernet port.

For latency, the design of the packet switcher IP means that a packet's latency is dependent on its length. This is due to the fact that we buffer the full packet before forwarding it. If a port is forwarded more throughput then it can handle, it will eventually overflow thus having infinite latency. The worst-case bound latency is when max size packets are sent to the same port from all other ports without the above happening. So in our case, we will have 1522 byte packets sent from two ports to the third. Thus the longest latency is:

$$\frac{(1522 * 2) * 8\ b}{10\ Mbps} = 2.4\ ms$$

Note: a few more cycles are spent in the PMOD NIC100 IP, but they have minimal effect on latency.

Improvements/Next Steps

There are a number of improvements that could be made to the network switch. In particular, communication with the ESP32 Wi-Fi PMOD uses a somewhat inelegant workaround due to limitations with the PMOD working at a UDP/TCP level instead of IP. We learned that the PMOD itself can be re-programmed to run customized commands, this meant that in theory, we could have programmed the PMOD to operate at the IP level instead, removing the need for the workaround. However, we could not attempt this due to the limited time remaining as this limitation was discovered in Milestone 6. We chose to avoid re-programming the device since it presented a high degree of risk. If we had the opportunity to restart this project, we would have made more of an effort in obtaining the PMOD and working with it in order to uncover this issue much earlier in the project. That way, we would potentially have had time to actually program the PMOD.

Apart from improving the Wi-Fi PMOD communication, we list a few other minor improvements that can be made if someone else were to take over our project;

- Support more devices, can theoretically fit five on one board
- Support IPv6 protocol in addition to IPv4
- Modify IP table to support DHCP
- Add a means of tracking/estimating individual packet latency

Project Schedule

During the course of developing the packet switcher, we made numerous changes to our project schedule. A side-by-side comparison of the original milestones initially proposed and the actual weekly accomplishments is shown in **Table 3**. Milestones relating to module completion are colour-coded accordingly for easier comparison.

Table 3: Table comparing proposed and achieved milestones

Milestone #	Milestones Proposed	Milestones Achieved
1	➔ Set up tools ➔ Research PMODs	➔ Set up tools ➔ Research PMODs
2	➔ Verilog for packet switching IP done ➔ Testbench for packet switching IP done ➔ **Communicate between Ethernet PMOD and computer** ➔ Configure Wi-Fi PMOD as an access point	➔ Initialized Ethernet PMOD ➔ Created design plan for the packet switching IP
3	➔ Add hardware counters to packet switching IP ➔ **Packet switching IP fully verified in simulation** ➔ **Communicate between Wi-Fi PMOD and computer**	➔ **Communicate between Ethernet PMOD and computer** ➔ Tried setting Wi-Fi PMOD as access point, ran into problems ➔ Finished Verilog of input buffer portion of packet switching IP
Mid-Project Demo	➔ Integrate Ethernet PMODs with packet switching IP	➔ **Packet switching IP Verilog completed and fully verified in simulation**
5	➔ **Add remote configuration using the hardware counters and a Microblaze**	➔ Configure Wi-Fi PMOD as an access point ➔ **Add remote configuration using the hardware counters and a Microblaze** ➔ Integrate Ethernet PMODs with packet switching IP
6	➔ Integration of all modules ➔ Prepare devices with scripts to send/receive packets	➔ **Communicate between Wi-Fi PMOD and computer** ➔ Prepare devices with scripts to send/receive packets
Final Demo	➔ Full project working	➔ Integration of all modules ➔ Full project working

Looking at the originally proposed milestones, we initially planned to finish all of the PMODs and the packet switching IP by the mid-project demo, while dedicating the latter half of the project to minor features and spare debug time in case any delays occurred. As expected of any project though, we experienced numerous issues and delays causing most of our initial milestones to be completed later than expected.

For the Ethernet PMOD, we had to spend additional time to set the PMOD in promiscuous mode and to interface it with the packet switching IP via AXI Stream, so this module was not completed until Milestone 3. The packet switching IP also required more time since we decided to be more thorough in planning and simulating the custom IP, therefore it was not completed until the Mid-Project Demo. The ESP32 Wi-Fi PMOD was initially not available and had to be ordered, so we did not receive the PMOD until around Milestone 3. Due to issues and bugs found with the Wi-Fi PMOD driver code and backlogging from prioritizing the completion of the packet switching IP for the Mid-Project Demo, the Wi-Fi PMOD was not completed until Milestone 6.

Despite the delays in finishing most of the modules, we were able to get the full project working as originally planned. This is largely due to us using up the spare time initially proposed in the latter half of the project to finish up the rest of the modules and to iron out any issues in the final integration. While some headaches in the last 2 weeks could have been mitigated had we uncovered some underlying issues with the Wi-Fi PMOD earlier in the project, ultimately our project planning worked out in our favour.

Detailed Description of IP Blocks

ESP32 Wi-Fi PMOD Communication

The ESP32 Wi-Fi PMOD communication block consists of three major modules, which are the *PMODESP32*, *wifi_microblaze*, and *axi_fifo* modules. A brief diagram showing how these modules interface with one another is in **Figure 4**.

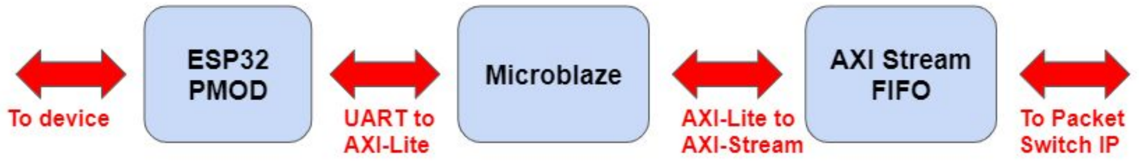


Figure 4: Diagram showing interfaces between Wi-Fi modules

The *PMODESP32* module is a custom IP by Digilent that provides a UART interface for communicating with the ESP32 PMOD. Interaction with the PMOD is done through sending various commands through UART that the PMOD will subsequently respond to. Through these commands the PMOD can be configured to act as an access point for devices to connect to wirelessly and establish UDP/TCP connections with these devices for packet transfer.

The *wifi_microblaze* module is a Microblaze processor that runs code to manage communication with the PMOD, the driver code from the Digilent PMOD library [4] was used as a starting template. The starter code provides a means of communicating with the PMOD by typing in commands through a serial terminal, however many modifications were made to the code to have it automatically send the required commands needed to configure the PMOD properly and to establish packet communication with its associated devices. Since the PMOD responses were sent serially through UART, the code also had to be modified to parse the raw packet data from the serialized PMOD output.

The *axi_fifo* module consists of the AXI Stream FIFO provided as an IP by Xilinx, this IP converts between an AXI-Lite interface that the Microblaze uses to an AXI-Stream interface that the Packet Switching Custom IP uses. Additional code was added to the Microblaze to relay packets between the PMOD and the custom IP through updating the appropriate registers in the AXI Stream FIFO to facilitate the transfer of packet data.

The functionality of this block was tested by connecting the AXI Stream FIFO in a loopback configuration. Therefore packets that are transmitted from a device through the Wi-Fi PMOD will eventually be received again. Since this block is primarily software-based, both the debug print statements from the Microblaze code as well as Wireshark were used to verify that packets were able to travel along both directions in the block.

NIC100 Ethernet PMOD Communication

The PMOD NIC100 AXI Stream IP Block was implemented to interface with the NIC100 Ethernet PMOD. Since neither Xilinx nor Digilent provided their own IP to interface with this device we have implemented our own. For our implementation, we found a very useful Github implementation which demonstrated how to drive the SPI interface to send and receive packets [3]. The Github source followed the specifications outlined for the ENC424J600 Microchip to properly initialize the device and communicate commands [5].

However, the Github source utilized a block RAM to store the received or transmitted packet and custom handshaking signals to issue the communication. As a result, we implemented our own version in Verilog (as the git source was in VHDL) so that we could meet the requirements of our project. To connect this IP to our Packet Switcher IP this module was required to send/receive all packets over two AXI Stream interfaces (one for each direction). To achieve this we utilized the same initialization code as the Github source but we added additional commands to configure the device into promiscuous mode. This mode essentially disables most of the filters so that it accepts all packets. To interface with AXI-Stream, the control flow logic was modified, a detailed list of all the connections to the PMOD NIC100 AXI Stream IP is shown in **Table 4**.

Table 4: List of signal connections to the PMOD NIC100 AXI Stream IP

Name	Interface	Signals	Description
s_axis	AXI Stream (Slave)	s_axis_tdata [31:0] (input) s_axis_tlast (input) s_axis_tvalid (input) s_axis_tready (output)	For Transmitting packets. See below for a description of how to drive these signals
m_axis	AXI Stream (Master)	m_axis_tdata [31:0] (output) m_axis_tlast (output) m_axis_tvalid (output) m_axis_tready (input)	For Receiving packets. See below for a description of how to drive these signals
pmod_miso	SPI (Master)	pmod_miso (input)	Master Input Slave Output connected to NIC100 Ethernet PMOD
pmod_mosi	SPI (Master)	pmod_mosi (output)	Master Output Slave Input connected to NIC100 Ethernet PMOD
pmod_ss	SPI (Master)	pmod_ss (output)	Slave Select connected to NIC100 Ethernet PMOD
pmod_sck	SPI (Master)	pmod_sck (output)	Serial Clock connected to NIC100 Ethernet PMOD
axis_aclk	AXI Stream, SPI	axis_aclk (input)	Clock for both AXI Stream interfaces and derives pmod_ss
axis_resetn	AXI Stream, SPI	xis_resetn (input)	Reset low for all logic

To transmit a packet, the *s_axis_tvalid* must be set to high along with valid data on *s_axis_tdata* (packet byte) to signal that a transmission is desired. When the IP is ready to service the request, it begins issuing commands to the PMOD then once it is ready to stream the data it begins by driving *s_axis_tready* high. After, the signal is asserted high once every 8 cycles. This is because over the SPI interface only one bit can be transmitted at a time. The stream is ended when the *s_axis_tlast* signal is driven high.

To receive a packet, the *m_axis_tready* must be set high to signal that a receive is desired and remains high until the IP responds back. When the IP is ready to service the request, it begins issuing commands to the PMOD. If there are no packets in the PMOD's buffer then there is no response from the IP. However, if there is then when it is ready to stream the data it begins by driving *m_axis_tvalid* high along with valid data on *m_axis_tdata* (packet byte). Then after, the signal is asserted high once every 8 cycles along with valid data on *m_axis_tdata*. The stream is ended when the *m_axis_tlast* signal is driven high.

The functionality of this IP was tested by creating a small project making use of an AXI-Stream FIFO and a MicroBlaze. This project was loaded onto a board and a laptop was connected to the NIC100 Ethernet PMOD. To test sending, the MicroBlaze wrote a predefined packet into the AXI-Stream FIFO and Wireshark was used to verify that the laptop had received the packet. To test receiving, the MicroBlaze also read from the AXI-Stream FIFO and printed out the contents of each packet. Once again, Wireshark was used to verify that the correct packets were received.

Packet Switcher

The Packet Switcher IP Block was implemented to direct the packet traffic between the PMOD connected devices. This IP was implemented entirely from scratch with one exception: the input FIFOs are Xilinx AXI Data FIFOs. The Packet Switcher IP supports up to three connected devices however it can be easily modified to handle more. The AXI Stream interface protocol was chosen as the standard for sending/receiving packets across all the PMOD devices. Additionally, there are AXI Lite Slave registers for configuring or viewing the static IP table and viewing statistics about how many packets were processed or dropped. A detailed list of all the connections to the Packet Switcher IP is shown in **Table 5** while a detailed list of the AXI Lite Slave registers used is shown in **Table 6**.

This IP consists of two major modules: the *input_queue* and the *packet_arbiter*. The input_queue utilizes AXI Stream Data FIFOs (*axis_data_fifo*) in each port to buffer all incoming packets. As the packets are pushed into the FIFO, metadata information is extracted from the IP header. The IP is checked to make sure that it is either ARP or IPv4. If it is neither then the packet is assigned an invalid IP address and dropped in the arbitrator. The IP address is recorded and added to a separate AXI Stream Data FIFO (*axis_meta_fifo*) that is present in each port.

Table 5: List of signal connections to the Packet Switcher IP

Name	**Interface**	**Signals**	**Description**
pmod_a_axis_s	AXI Stream (Slave)	pmod_a_axis_s_tdata [31:0] (input) pmod_a_axis_s_tlast (input) pmod_a_axis_s_tvalid (input) pmod_a_axis_s_tready (output)	For incoming packets from port A.
pmod_b_axis_s	AXI Stream (Slave)	pmod_b_axis_s_tdata [31:0] (input) pmod_b_axis_s_tlast (input) pmod_b_axis_s_tvalid (input) pmod_b_axis_s_tready (output)	For incoming packets from port B.
pmod_c_axis_s	AXI Stream (Slave)	pmod_c_axis_s_tdata [31:0] (input) pmod_c_axis_s_tlast (input) pmod_c_axis_s_tvalid (input) pmod_c_axis_s_tready (output)	For incoming packets from port C.
pmod_a_axis_m	AXI Stream (Master	pmod_a_axis_m_tdata [31:0] (output) pmod_a_axis_m_tlast (output) pmod_a_axis_m_tvalid (output) pmod_a_axis_m_tready (input)	For outgoing packets to port A.
pmod_b_axis_m	AXI Stream (Master)	pmod_b_axis_m_tdata [31:0] (output) pmod_b_axis_m_tlast (output) pmod_b_axis_m_tvalid (output) pmod_b_axis_m_tready (input)	For outgoing packets to port B.
pmod_c_axis_m	AXI Stream (Master)	pmod_c_axis_m_tdata [31:0] (output) pmod_c_axis_m_tlast (output) pmod_c_axis_m_tvalid (output) pmod_c_axis_m_tready (input)	For outgoing packets to port C.
config_axi_s	AXI Lite (Slave)	All defined by the AXI Lite interface protocol	For reconfiguration of IP address table and reading accepted or dropped packet counters
clk	AXI Stream, AXI Lite	clk (input)	Clock for all AXI Stream interfaces and AXI Lite Slave
resetn	AXI Stream, AXI Lite	resetn (input)	Reset low for all logic

Table 6: AXI Lite Slave registers accessible from config_axi_s

Slave Register	**Category**	**Description**
slv_reg0	IP address table	Static IP address assigned to port A (default: 0xC0A8010A which is 192.168.1.10)
slv_reg1	IP address table	Static IP address assigned to port B (default: 0xC0A8010B which is 192.168.1.11)
slv_reg2	IP address table	Static IP address assigned to port C (default: 0xC0A8010C which is 192.168.1.12)
slv_reg4	Packets processed	Number of incoming packets processed by port A
slv_reg5	Packets processed	Number of incoming packets processed by port B
slv_reg6	Packets processed	Number of incoming packets processed by port C
slv_reg8	Packets dropped	Number of incoming packets dropped from port A due to unknown destination IP address or unknown protocol. Note: packets with their destination set to port A are also dropped (Replay packets are not accepted)
slv_reg9	Packets dropped	Number of incoming packets dropped from port B due to unknown destination IP address or unknown protocol. Note: packets with their destination set to port B are also dropped (Replay packets are not accepted)
slv_reg10	Packets dropped	Number of incoming packets dropped from port C due to unknown destination IP address or unknown protocol. Note: packets with their destination set to port C are also dropped (Replay packets are not accepted)

Once a full packet is buffered within one of the *axis_data_fifo* FIFOs, the arbiter reads from the metadata FIFO (*axis_meta_fifo*) to determine where to send the packet. Within the arbiter is a static IP address table which assigns an IP address for each destination port. The address table is configurable through the AXI Lite Slave registers. If the destination IP address matches one of the destination ports then a transfer is scheduled. If there is a conflict where multiple sources request to send a packet to the same destination port, then it is resolved in a round-robin fashion. In the case where the destination IP does not match any entries in the IP address table or if the destination IP address is the same as the source IP address then the packet is simply dropped. Statistics about the number of packets processed or dropped are collected within this module. A description of which slave register to access for statistics or for viewing/modifying the IP address table is shown in **Table 6**.

Testbenches were created to simulate the *input_queue* and *packet_arbiter* modules. The *input_queue_tb* testbench was used to verify that the input_queue module properly extracts metadata information. The *packet_arbiter_tb* testbench was used to verify that the arbiter properly routed packets and resolved conflicts. The *Description of Design Tree* section of the report contains information on how to access these files.

Remote Configuration

The remote configuration feature allows for a device connected to the onboard ethernet to configure the switch. The TCP communication between the connected device to the *config_microblaze* component is based on the Digilent "*Nexys 4 DDR - Getting Started with Microblaze Servers*" tutorial [6]. This tutorial uses the Xilinx LwIP Echo Server template. This template generates a *main.c* file and an *echo.c* file. The *process_message* function inside *echo.c* was modified such that instead of simply echoing a message, it processed and responded to the message. The function can process the following commands;

- "status": Lists the packet received/dropped counters for every port.
- "iptable -l": Lists the current address table entries.
- "iptable -s *port address*": Sets the IP address associated with a port.
- Invalid commands: A response is sent indicating that the command was invalid.

The Packet Switcher IP Block was implemented with AXI slave registers that support the remote configuration, a list of these registers can be seen in **Table 6**. The *process_message* function first checks the start of the received message. If it is "status" then all 6 AXI slave registers that contain the packet counters are read, formatted, and sent back over the TCP connection. If the message is "iptable -l" then the 3 AXI slave registers with IP address table entries are read, formated, and sent back. If the message is "iptable -s" then the port is checked to be "a", "b", or "c", and the corresponding AXI slave register is updated with the new address.

Finally, a client python script was created to make sending the commands more convenient. The client establishes a TCP connection to the configuration MicroBlaze. It sends commands entered over TCP, and prints the replies.

Description of Design Tree

This section outlines the structure of the github repository available at https://github.com/nguy1442/ece532 which contains all IPs along with source code necessary to build the project and reproduce the final result.

- *docs/*
    - Contains all documentation. This includes the group final report, the presentation slides, and a link to the video demonstrating the project
- *full_project/*
    - *proj/*
        - Contains the tcl scripts which create the project and build the block diagram
    - *src/*
        - Contains the source code to the LWIP server for remote configuration and Wi-Fi PMOD driver
        - Contains the constraints file for the project
        - Contains python scripts to issue commands to the LWIP server and scripts for sending/receiving udp packets from the Wi-Fi connected device
- *packet_switcher/ip_repo/packet_switcher_ip_1.0/*
    - Contains the packaged Packer Switcher Custom IP with source code in the *hdl/* folder
- *pmod_nic100/ip_repo/pmod_nic100_axi_stream_ip_1.0/*
    - Contains the packaged NIC100 AXI Stream Custom IP with source code in the *hdl/* folder
- *pmods/*
    - Contains the Digilent provided IPs and driver code needed to communicate with the ESP32 Wi-Fi PMOD
- *testbenches/*
    - Contains testbenches used to simulate the *input_queue* and *arbitrator* modules from the Packer Switcher Custom IP

The top level of the repository contains a *README.md* file that has a description of how to build the full project.

Tips and Tricks

Having just finished our network switch project, we list some useful tips and tricks below. Most of these tips were developed from past experience, so we believe that they may help future students working on similar projects to avoid the same pitfalls that we went through.

Work with PMODs ASAP
If your project involves any PMODs, you should prioritize working with them first in the project. Not all of the PMODs will behave in the way you initially expect, so you may have to find workarounds or alternatives. You do not want to discover this late in the project where the chances of you procuring a different PMOD or implementing a workaround in the time remaining is low.

If the Microblaze code fits in BRAM, run it in BRAM
Avoid putting Microblaze code in the DDR memory unless absolutely necessary, (i.e, the code cannot fit in BRAM) extra latency is incurred if the code is run in DDR memory vs. local BRAM. In our case, this added latency caused the ESP32 PMOD driver code from the Digilent library to not function properly. In addition, you can lower the memory footprint of the Microblaze code by using/creating rudimentary versions of string manipulation functions such as ‘printf’ or ‘atoi’ if they are used in the code.

Do not underestimate integration
Reserve at least 1-2 weeks in our project schedule entirely for integration of all of the modules. Do not expect everything to work on the first attempt as you will inevitably find bugs in the final project that can take hours/days to isolate and debug, ILAs are your best friend in this case. As a corollary, thorough module unit-testing can eliminate most of these bugs before the final integration.

Use Wireshark to track packets
If your project involves network-related components such as Ethernet or Wi-Fi, Wireshark or other related tools can be very helpful to learn as they allow you to track all incoming packets. Viewing these packets can help you debug network communication as they allow you to determine if packets were formatted properly or if they were dropped at certain devices.

References

[1] https://standards.ieee.org/standard/802_3bm-2015.html
[2] https://www.routeralley.com/guides/switching_tables.pdf
[3] https://github.com/carljohnsen/Pmod-NIC100
[4] https://github.com/Digilent/vivado-library/tree/master/ip/Pmods/PmodESP32_v1_0
[5] http://ww1.microchip.com/downloads/en/DeviceDoc/39935b.pdf
[6]https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-getting-started-with-microblaze-servers/start