An FPGA-based Network Switch using SPI Protocol

Andrea Pinto
Electrical and Computer Engineering
Saint Louis University
St. Louis, USA
andrea.pinto.1@slu.edu

***Abstract*—This work presents the results of the Hardware (HW) and Software (SW) co-design class. This project aimed to build an FPGA filter/switch . The available hardware allowed to follow the HW and SW co-design approach by creating a splitting of the functionalities. The idea is that the HW can quickly decide on the forwarding and filtering logic. The SW would make it easier to use libraries to implement the communication logic between the FPGA and the Ethernet (ETH) ports based on the SPI protocol. Finally, a testbed is executed to test the configuration.**

***Index Terms*—FPGA, switch, filtering, nic100, pmod, spi, HW-SW co-design**

I. Introduction

As Network Softwarization becomes more prevalent, the role of networking switches is becoming increasingly important when it comes to performance. Implementing an FPGA switch would allow a potential network manager to take advantage of all the benefits of an FPGA, such as the lowest possible latency and fastest possible performance, while keeping renewal costs low. An FPGA switch is used to implement special purpose switching, bridging, and monitoring applications in general.

In the Network Softwerization context, SDN (Software-Defined Networking) is a networking design that simplifies network management and increases network programmability. Separation of control and data planes, logically centralized network control, and a flexible and open interface to program underlying network architecture are its key features. The southbound interface is responsible for the interplay of network states between the control and data planes in the SDN architecture. It also defines the SDN data plane's forwarding abstraction.

The data plane must enable means to introduce new network protocols, header formats, and functions while still forwarding traffic as quickly as possible, as SDN gives some hope for rapid prototyping and deployment. The most extensively used standard SDN southbound interface is OpenFlow, which is a vendor-agnostic interface to switching elements. As a result, the majority of SDN switches are OpenFlow SDN switches.

The SDN data plane can be implemented in one of two ways: hardware or software. Customized ASIC switching chips are used in some SDN hardware switches made by HP, NEC, and Arista. While the ASIC method can achieve excellent forwarding performance, it is difficult to extend it to additional protocols and functionalities. Programmable hardware switches can give flexibility at the cost of a significant number of hardware resources, such as costly TCAMs for flow entry.

SDN software switches are the most adaptable when it comes to adding new network services, protocols, and functions. Unlike TCAMs in hardware SDN switches, software SDN switches have plenty of memory for supporting flow rules. As a result, they are the first-hop virtual switches of choice for SDN researchers in laboratories and have been widely used in data centers as first-hop virtual switches. However, a completely software-based approach will struggle to meet most modern networks' stringent performance and line-speed security requirements.

It would be possible to attempt to utilize the benefits of FPGA (Field Programmable Gate Array) on processing packets as the capacity and computational power of FPGA (Field Programmable Gate Array) continue to improve. Microsoft has designed an FPGA fabric that is attached to each server in order to accelerate large-scale data center services with tailored function logic.

The goal of this work is to propose a special purpose switch architecture to filter packets via HW, which is in line with the course's goal of HW&SW co-design. Specifically, the SW part will handle the peripheral connections and the HW part will handle the filtering logic.

In particular a demonstration of how to set up a SW driver to allow the FPGA board to connect with all peripherals and, lastly, a filtering/switching behavior in HW. The FPGA board utilized is a Basys3, and the ethernet devices attached to it via the external bus are NIC 100. Figure 1 illustrates the final setup. The FPGA will not have a flash memory with stored code for this project. Plugging in each ethernet cable, generating the code with SDK, retrieving the bitstream, then downloading it through the JTAG cable would be required to run the code. Without an OS we don't have dynamic memory. The provided solution consist of three main steps:

- Defining the soft processor (Microblaze) and the modules attached to it. The main added modules to the built Microblaze are axi_quad_spi modules and a customized module to manage a shared hardware memory,
- studying the SPI communication protocol to setup the communication between the FPGA and the peripherics,
- Implementing a communication driver between the peripherics using XSPI libraries via SW and an FPGA behaviour to perform switching/filtering decisions via HW.

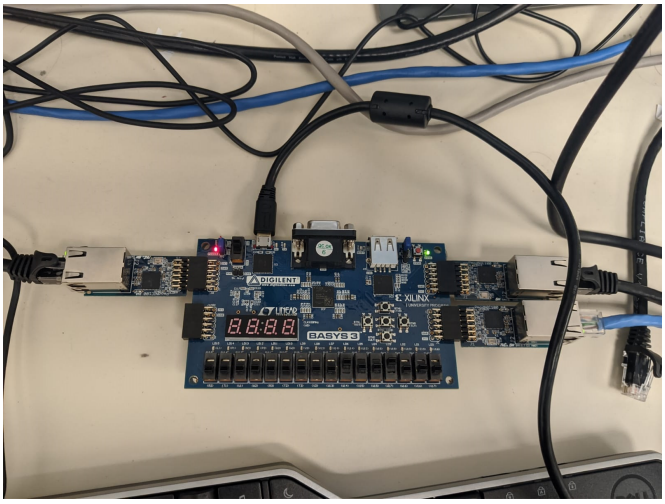

Fig. 1: Basys3 attacched to 3 Pmod NIC 100

Following that, a background section will be presented to show the context and current state of the art in this field. The project design, workflow of the implemented solution, and testbed analysis will all be shown in the implementation section. Finally, a conclusion section will summarize the project's core principles and provide further work samples.

II. Background and Related Work

As internet performance expectations are growing more and more, it is needed to have high performances hardware to meet high-level performances. This project's purpose was an entry-level switch/filter whose aim was not to perform high-level performances but to follow HW and SW co-design principles. Thus it was hard to find some related work or lecture addressing an FPGA filter/switch behavior. Many of the research available on the internet is on building FPGA switches for data centers in three main categories: special-purpose switching, bridging, and monitoring applications.

Some of the compelling lectures found are analyzing the slow results of Software switches, and they try to implement customized solutions to enhance the performances in Software Defined Networking SDN. The lecture FAS: Using FPGA to Accelerate and Secure SDN Software Switches aims to use FPGAs to the previously explained aim and improve the performance and the capacity against malicious traffic attacks of SDN software switches by offloading some functional modules.

More industry-related research is being conducted to build high-performance FPGA-based switches, such as new wave dv 32-Port Programmable Switch FPGA-based Network Switch, which aims to have a high port-density, completely FPGA-based network switch built for special purpose switching, bridging, and monitoring applications.

III. Design and Work Performed

This section will discuss the workflow used to achieve the project goal outlined in the introduction. It's first mentioned in the design section to help you comprehend the testbed you've chosen. After that, an examination of the hardware connections and protocols is performed. Finally, a results section is supplied to show how the tests were completed.

A. Design

It was necessary to set up different pieces of hardware consisting of a Basys3 FPGA board and three Pmod NIC 100 ethernet ports. Note that this device uses the ENC424J600 processor. The software used to program the FPGA board is Vivado version 2018.3. This software is used to flash the design into the FPGA memory using the JTAG protocol. The Vivado software allows the user to build a microprocessor, called MicroBlaze, on the FPGA. It is possible to attach to this microprocessor different personalized blocks to perform the needed logic and behavior. This microprocessor is very interesting since it allows the developer to write software executable. Moreover, it is possible to add extra blocks to run customized Hardware behaviors. It was possible to realize the filtering logic by combining these three main logics. The following section will describe the Software and Hardware implementation steps to show how the functional decomposition have been designed.

B. Implementation

Learning how to connect the board to the Ethernet ports using a communication protocol was required to continue with the configuration. SPI was the used communication protocol. The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems. The interface was developed by Motorola in the mid-1980s and has become a de-facto standard. SPI devices communicate in full-duplex mode using a master-slave architecture The SPI bus specifies four logic signals:

- SCLK: Serial Clock (output from master)
- MOSI: Master Out Slave In (data output from master)
- MISO: Master In Slave Out (data output from slave)
- CS /SS: Chip/Slave Select (often active low, output from master to indicate that data is sent).

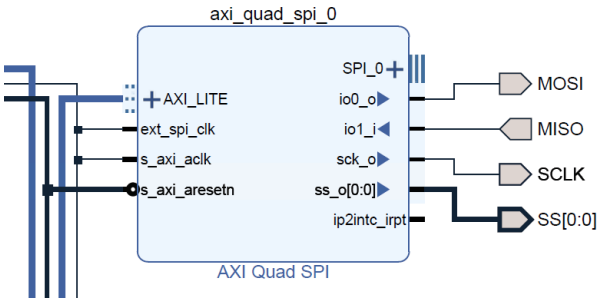

Fig. 2: axi_quad_spi

A first step to initialize the communication between these devices was to set up the MicroBlaze processor creating a new design and by adding its block, attaching different IP modules such as a clock wizard (100Mhz), axi_quad_spi logic

```
set_property IOSTANDARD LVCMOS33    [get_ports clk]
set_property PACKAGE_PIN W5         [get_ports clk]

set_property IOSTANDARD LVCMOS33    [get_ports reset]
set_property PACKAGE_PIN R2         [get_ports reset]

set_property IOSTANDARD LVCMOS33    [get_ports MISO]
set_property PACKAGE_PIN B15        [get_ports MISO]

set_property IOSTANDARD LVCMOS33    [get_ports MOSI]
set_property PACKAGE_PIN A16        [get_ports MOSI]

set_property IOSTANDARD LVCMOS33    [get_ports SCLK]
set_property PACKAGE_PIN B16        [get_ports SCLK]

set_property IOSTANDARD LVCMOS33    [get_ports {SS[0]}]
set_property PACKAGE_PIN A14        [get_ports {SS[0]}]
```

Fig. 3: Constraints file

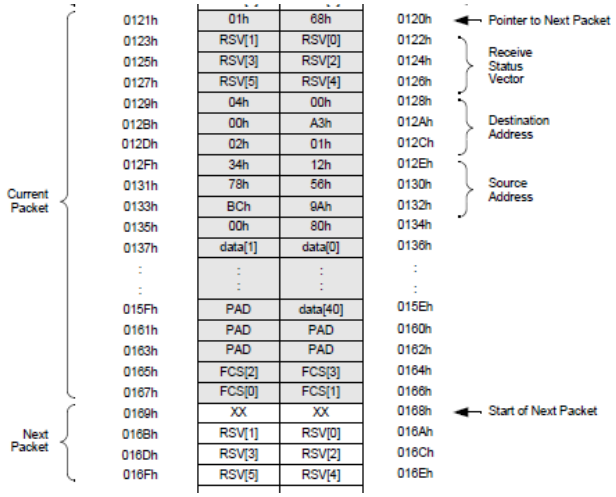

Fig. 4: Receiver Buffer Logic

modules to connect the Ethernet port to the board and finally a custom block to share memory between HW and SW. Since three Ethernet adapters were available, the design has three axi_quad_spi modules, and each of these modules is connected to external ports called MISO, MOSI, SS, SCLK as described by the SPI standard shown in Figure 2.

The Ethernet adapters are connected to the board using the Pmod ports. Pmod port pins have been programmed to implement the SPI behavior described. It was required to write an association in the constraints file, assigning a pin to each defined signal to enable the reported SPI communication with the device. Constraints file is reported in Figure 3.

When a packet is received, it is stored inside a buffer as shown in Figure 4. It is possible to notice that there is a pointer referring to two dummy bytes (a read fo these first two bytes can automatically move the pointer to the next value). Then there is a vector called RSV and from this vector it was possible to understand the size of the received frame (the RSV[0] and RSV[1] indicates the following fields total size: destination address, source address, type/length, data, padding and CRC). In this way it was possible to calculate how to move the tail pointer to the next header once a lecture was completed.

a) Software functionalities:

After this first setup configuration, it was possible to write the Software driver to enable the connection, receive and forward packets using the Xilinx SDK software. It was possible to configure AXI bus calls to access the external memory of the SPI ethernet 100 by using the Xilinx Xspi library. The primary function of this library is called *XSpi_Transfer*. This function reads a specific location from the stack of the peripheric connected via SPI and eventually gives back a specific amount of bytes. The inputs are defined in the following function: *int XSpi_Transfer(XSpi * InstancePtr,u8 * SendBufPtr,u8 * RecvBufPtr,unsigned int ByteCount)*. It is needed to pass the instance pointer, send and receive buffer, and byte count.

The configuration described in the ENC424J600 data sheet was followed to appropriately interact with the ETH module. For example, it was possible to read the MAC address contained in the MAADR register.

Each address register is identified with a mnemonic name. The banked addressing method was used to access a specific register. A function called *bankSelect* was implemented to select first the bank of a specific address register. Then it was possible to call another function depending on the needed operation to perform, read or write. Implemented functions to write are *net100Writer* to write a generic amount of data to the specific register, *net100WriteWord* to write a word to the specific register. These two methods return a null value since the write operation does not need to return any value. Implemented functions to read from the SPI peripheric buffer are *net100ReadByte* that returns a single byte in the unsigned int format and receives as input the address to write at, *net100ReadWord* similarly to the previous one returns an unsigned int but with a word content pre-formatted in a little-endian format. The code snippet for the read word function is in Figure 5.

```
unsigned int net100ReadByte(XSpi *SpiInstancePtr, unsigned char add){
    unsigned char ReadBuffer[1];
    for(int i=0; i<2; i++) {
        ReadBuffer[i] = 0x00;
    }
    XSpi_Transfer(SpiInstancePtr, (unsigned char []){add}, ReadBuffer, 2);
    return ReadBuffer[1];
}
```

Fig. 5: Read Word function

Just with this set of functions, it was possible to go through the setup steps to activate a single port communication. These procedures are contained inside the method *net100Init* that runs every time a new NIC100 needs to be initialized. It consists of several phases, such as performing a reset, setting up the buffer, initializing MAC and PHY filters, and establishing the connection through ETH. It was then needed to configure the receiving and forwarding packets methods.

The ENC424J600 memory is organized as shown in figure 6, and it consists of two main sections, a receiving and a sending buffer. Both have a circular logic and use pointers

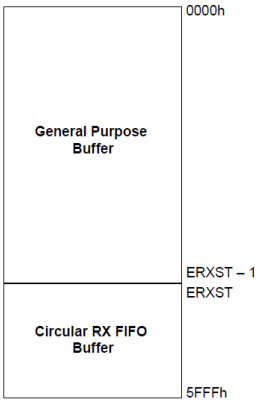

Fig. 6: SRAM Buffer

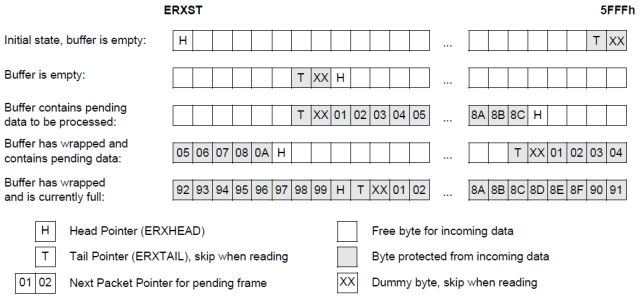

Fig. 7: Buffer head and tail management

to access the tail and the header of the memory. The push operation happens in the queue head, and its relative pointer is automatically updated. The pop operation is managed manually by the SW implemented to avoid the overwrite of packets present in the stack. Once the received packet is handled, the header is taken to the following packet header. Figure 7 describes all the empty and full stack cases adequately. If the stack is full, packets are dropped. Figure 8 shows how to read the dummy byte and the RSV part. Moreover since the memory is word aligned if an odd numbyte is received an extra byte have to be addedd manually added.

```
//2. Begin reading at address pointed to by the application variable, NextPacketPointer
// Set ERXRDPT to desired location (Receive Buffer Read Pointer)
do {
    net100WriteWord(SpiInstancePtr, WRXRDPT, add);
    valW = net100ReadWord(SpiInstancePtr, RRXRDPT);
    xil_printf("ERXRDPT: 0x%04x\r\n", valW);
} while(valW != add);

//3. Read the first two bytes of the packet,
//which are     the address of the next packet and write to NextPacketPointer.

//first read is a dummy word
net100ReadWord(SpiInstancePtr, RRXDATA);

//next packet start address
int newTail = net100ReadWord(SpiInstancePtr, RRXDATA) - 2; //subtract the dummy byte?

//4. Read the next six bytes, which are the Receive Status Vector (RSV).
XSpi_Transfer(SpiInstancePtr, (unsigned char []){RRXDATA}, RdBuf, 7);
memcpy(RSV, &RdBuf[1], 6);

numBytes = RSV[1]*256 + RSV[0]; //includes the dst , src , type/length, data, padding and CRC.


// if numBytes is odd add one
if((numBytes & 0x01) == 1) {
    numBytes += 1;
}
xil_printf("RSV numBytes: 0x%04x\r\n",numBytes);
```

Fig. 8: RSV read function

The receiving packet behavior is implemented in the function receive packet. Two of these functions have been implemented. The first, called *receivePkt*, aims to receive packets and read the content, distinguishing between an arp packet, the IP protocol, and the inner transport UDP and TCP data. A second function, called *filterReceivePkt*, stores the received packet in the local memory. It will then be available inside a local memory to be forwarded to other ethernet peripherals using the send packet function. This function also decides how to forward the next packet by copying specific fields in the FPGA memory. This memory is contained in an IP design block 10 that checks specific conditions to make the forwarding decision. In our case, the implemented solution was to broadcast arp packets from a source to the other two available ETH ports, and the other packets were split on an even or odd address logic to the other two ETH ports. Finally, this function calls the send packet function based on the forwarding decision taken by hardware. The forward function is called *filterSendPkt*. It runs every time it is needed to send a pkt from the NIC100 *sndBuff*. This function will copy the packet from the local memory and store it in the relative Ethernet output port. In particular the send packet function will define how to treat the specific packet. In our case the packet is entirely copied in the send buffer except for the CRC that is automatically calculated. The Figure 9 shows how this mode is configured.

```
//Inputs are Destination Address, Source Address, Protocol, Data.
//MAC insertion disabled (TXMAC = 0) (ECON2<13>).
bankSelect(SpiInstancePtr, BANK3);
net100Writer(SpiInstancePtr,(unsigned char []){BFC+ECON2H, 0x20}, 2);
//Automatic padding no (PADCFG<2:0> = 000) (MACON2<7:5>)
bankSelect(SpiInstancePtr, BANK2);
net100Writer(SpiInstancePtr,(unsigned char []){BFC+MACON2L, 0xA0}, 2);
//CRC generation enabled (TXCRCEN = 1)  (MACON2<4>)
net100Writer(SpiInstancePtr,(unsigned char []){BFS+MACON2L, 0x10}, 2);
```

Fig. 9: Send packet configuration

The described behavior is performed in polling mode. It means that the main function is continuously checking for the number of packet condition higher than zero. If one or more packets can be read from the receive buffer, the function reads and forwards the packet as described in the previous section.

b) Hardware functionalities:

Since the XSPI library easily allowed communication via SPI between the ETH external ports and the FPGA, the implementation was straightforward in Software. The central part of the proposed solution was implemented in Software, but this section will describe the main Hardware contribution.

As mentioned in the previous sections, the Hardware aims to decide on the packet forwarding. The Figure 10 shows the logic block attached to the Microblaze to implement the Hardware functionalities.

The implemented behavior is shown in the Figure 11. The filter block has 16 registers of 4 bytes each. One of the challenges was to store Ethernet addresses from SW to HW. Since each ETH address is 6 bytes, it was necessary to split the address into two different registers and store it in this way: 4 bytes in the first register and 2 bytes in the other register.

Register 14 contains the type of protocol to test if the received one is an ARP packet. The implementation uses register 12 as ready status. If the value is equal to 1, the Hardware can test one of the required conditions. If the destination address terminates with an odd value, the packet is sent to the interface 2. Instead if it is even it is sent to the destination 3. The only time the packet is broadcasted is when the type is ARP Once the decision is taken, the result is written in the temporary variable myReg, and the logic will proceed to copy it in register 15 that the Software will read to forward the packet. The Software will check if the register 12 value is 0, then proceed to forward it.

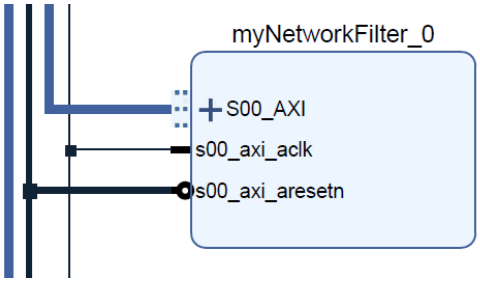

Fig. 10: Filter module

```
process(S_AXI_ACLK)
      begin
          if rising_edge(S_AXI_ACLK) then
              if(slv_reg12(7 downto 0) = "00000001") then
                  if(slv_reg4(11 downto 0) = "100000000110") then
                      myReg <=  (C_S_AXI_DATA_WIDTH-1 downto 0=> '1');
                  elsif(slv_reg1(1 downto 0) = "10") then
                      myReg <=  (0 => '1', others => '0');
                  else
                      myReg <=  (1 => '1', others => '0');
                  end if;
                  myReg1 <= (2 => '1');

              end if;
              slv_reg12 <= (C_S_AXI_DATA_WIDTH-1 downto 0 => '0') ;
          end if;
      end process;
```

Fig. 11: Filtering behaviour VHDL implementation

C. Test

It was needed to set up the following testbed to test the filtering switching behavior. The testbed was composed of 3 computers running Wireshark and connected to the board through ETH cables. The board has 3 Pmod nic 100 establish the connection to each host as shown in Figure 1. In particular, it was considered a single ETH traffic source, and all its traffic going through that wire was finally managed as described in the previous section. After programming the board correctly using the Xilinx SDK software, connecting the board with the JTAG mode, connecting all of the 3 available Ethernet ports to build the testbed, it was possible to run the implemented code and obtain the output in the console log as shown in Figure 12.

The traffic generated by the source computer to the ETH connection was sufficient to test the implemented behavior since there were always arp packets and odd or even ETH

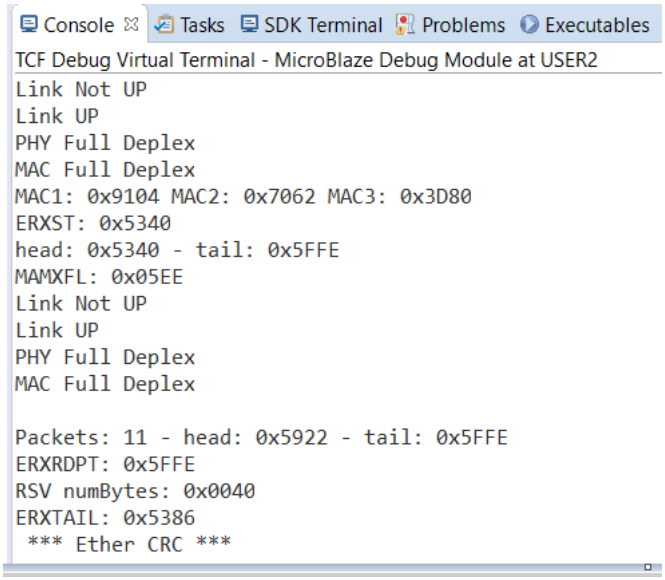

Fig. 12: Board initialization setup

addresses. In this way, it was possible to test the implemented logic. As a result, it was possible to notice that the output of the Wireshark packet trace was compliant with the behavior described and implemented. Figure 13 and 14 shows that applying the same ARP filter to the wireshark incoming traffic both the computer connected to ETH interfaces 1 and 2 were receiving the same ARP request from the computer attached to ETH0. This behavior is compliant witht he traffic described before.

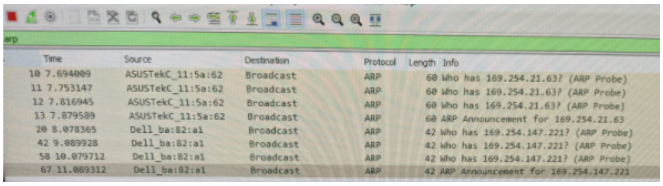

Fig. 13: Wireshark PC 1 attached to ETH1

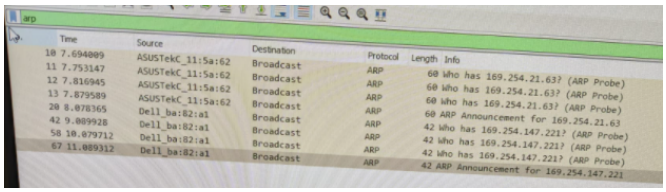

Fig. 14: Wireshark PC 2 attached to ETH2

IV. Conclusions

This work aimed to implement a filtering / switching logic using the FPGA hardware described in the previous sections. The project's purpose was to follow the HW and SW co-design principles. It was needed to divide the logic into two parts. The software part takes care of the configuration setup and the receiving and forwarding logic. The hardware implements the filtering / switching logic. Overall the implemented project

works as shown in the simulation section. There are advantages on implementing a Switch using FPGAs but this project aim was not to obtain good performances but to follow the logic defined by the HW and SW co-design.

The code complexity reached a good result with this implementation since the software code is around 800 lines of code and the hardware part is around 100 lines of code. The design complexity can be improved trying to implement customized SPI communication modules and to work on the single Master with multiple slaves mode. The project is the basement for further updates and implementations, and it would allow conduct studies about the internet softwarization on FPGA boards to study their performances.

V. References

- git repo: https://github.com/pintauroo/SPIswitch.git
- course material
- Digilent pmod nic100 datasheet.
- Basys 3 datasheet
- High-Performance FPGA Network Switch Architecture
- FAS: Using FPGA to Accelerate and Secure SDN Software Switches