DaFPGASwitch

Lauren Chin lmc2265 Fathima Hakeem fh2486 Teng Jiang tj2488
Ilgar Mammadov im2703 Irfan Tamim it2304

Wednesday 3rd April, 2024

Contents

List of Figures and Tables

1 | Overview

The primary objective of this project is to implement a hardware-based network switch using an FPGA (Field-Programmable Gate Array).

A network switch is a dedicated hardware component tasked with connecting multiple computers and networking devices within a computer network. The primary function of a network switch is to route incoming data packets to the correct destination ports, guided by the destination addresses encoded within each packet.

This process facilitates the transfer of data across a Local Area Network (LAN) or Wide Area Network (WAN). For our project, we aim to replicate the functionality of a network switch using an FPGA, operating with a simplified, yet streamlined packet structure for simulation purposes.

For this project, we simulate a 4-input, 4-output switch, and conduct performance evaluations.

2 | Block Diagram

Conceptually, our system runs in the following fashion: The software generates the packets and sends them to the according ingress modules. The ingress modules decide which packet gets sent onto the crossbar (switch fabric). The egress modules receive data from the crossbar and send it back to the software.

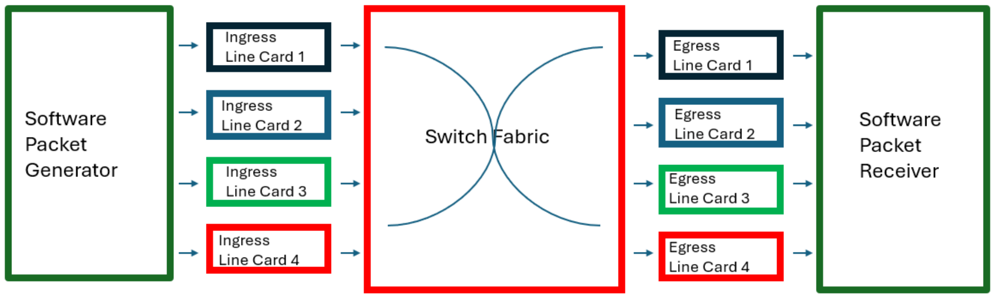


Figure 2.1: Block Diagram of the whole system.

As we proposed in the proposal, we're going to maintain virtual output queues per egress port, as illustrated below, inside the ingress port module. So most of the functionalities are inside each ingress port.

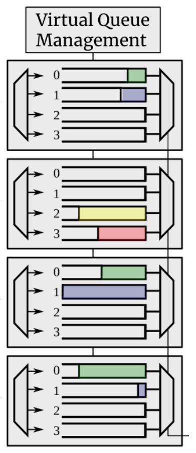


Figure 2.2: Virtual Output Queues of four input modules. Each of the queues inside one module corresponds to one output module.

Here's the illustration of 2 ingress ports.

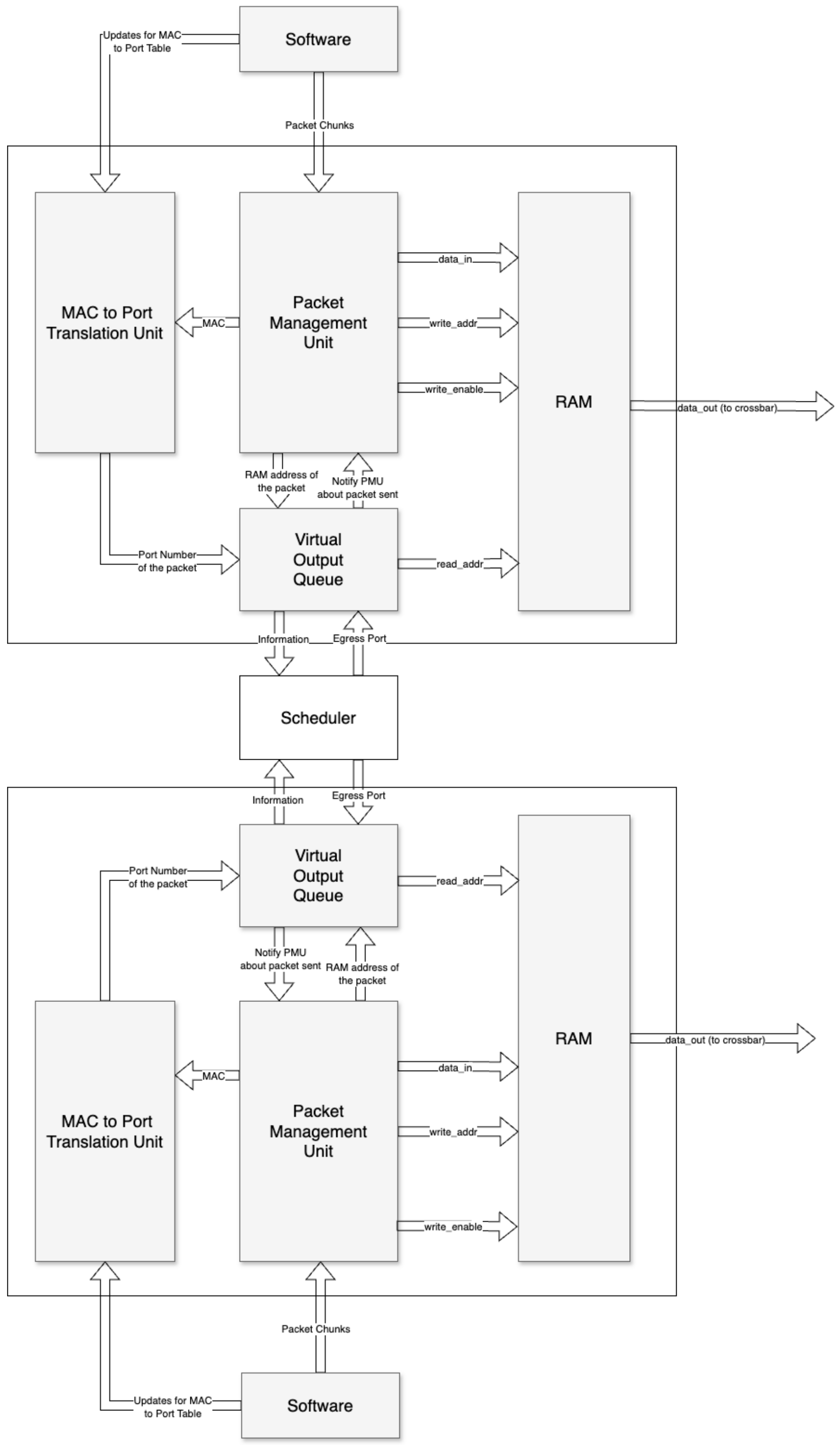


Figure 2.3: Block Diagram of 2 Ingress Ports for illustration.

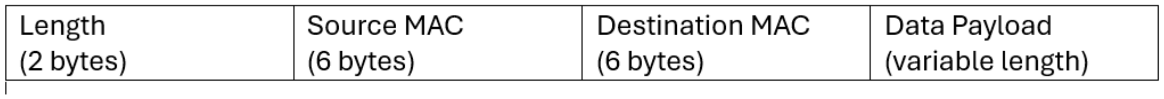
Packets are sent to the Packet Management Unit (PMU) in chunks. The PMU extracts the MAC address from the `source` field, and forwards the MAC address to the MAC-to-port translation unit. The PMU also picks an available slot in memory to store it in RAM. The port number and packet addresses are sent to virtual output queues (VOQ) to enqueue the packet, while VOQs also control the dequeuing decision about which packet to send to the crossbar. The software can also interact with the MAC-to-port unit to update the MAC-to-port table.

3 | SW-HW Interface & Protocols

3.1 Packet definition

The packet will include 4 main segments: Length, Source MAC address and Destination MAC address, and Data Payload. As the maximum length can be 1500 bytes in the Ethernet data frame, the ”Length” segment should be 2 bytes to accommodate ”1500” as a length. The Source MAC address will be 48 bits long and does not have any role in the functionality of the switch, but we keep it for realistic implementation. The Destination MAC address will be 48 bits long and it will be used for MAC-to-Port translation. Lastly, the data part will be variable length and the maximum overall length of the packet will be 32 bytes long.

We need to know how many memory chunks we have to assign to the packet, so the ”Length” part will be first in the data frame. Overall, the format can be illustrated as below:



Length (2 bytes)	Source MAC (6 bytes)	Destination MAC (6 bytes)	Data Payload (variable length)

Figure 3.1: Packet Format

3.2 The SW-HW Interface

The software that we are going to utilize for our switch system will have 2 predominant parts.

Packet Generator

The first part of the software is a packet generator. The main segments that must be included in the packet are source and destination MAC addresses, length, and data payload:

In order to make our project more realistic, we prefer to stick to variable-length packet generation. The ”length” segment included in the data format will be used for that purpose and its length will be 2 bytes.

Destination MAC addresses will be used in the MAC-to-Port translation functionality, which will be discussed below. As mentioned above, we aim to make our project as realistic as possible, so the MAC address will be 48 bits (8 bytes).

Once the packets are simulated in the software, through the software-hardware interface, they will be passed to the hardware. The rest of the system, till the packets are exported from output ports will be in the hardware. Once a packet is sent through the crossbar to the appropriate output memory, the second part of the software comes into play which is a packet receiver.

The software pulls the packets from the output memory and we can again check the destination port that the packet was addressed using MAC-to-Port translation in the software and determine if our switch performs accurately.

We also aim to measure the speed of our FPGA switch using the time difference that it was generated and accepted.

Determining the next packet transfer

As we decided to have a variable-length data payload, and we can send only 32 bits at a time, we need some mechanism to identify the beginning and end of a packet. To achieve that there are several methods, such as sending start and stop bits from software or using the ”length” segment included in the packet. We decided to do that in hardware using the second method which is about the ”length” field. We call these signals ”Strobe” and ”EndOfpacket”, and we decided to block the possibility that both signals are True at the same time.

4 | Components

4.1 MAC-to-Port Translation

We are going to use 4 input and 4 output ports, and the destination port will be defined using MAC-to-Port translation tables. Firstly, we will generate the table in the software, which can be described as shown below:

MAC Address	Output Port #
`00:11:22:33:44:55`	1
`00:11:22:33:44:56`	2
`00:11:22:33:44:57`	3
`00:11:22:33:44:58`	4

Table 4.1: Example MAC address to port translation

Then when the program is initialized, we first send each MAC address to the hardware through the software-hardware interface (with `ioctl()`) and store them in appropriate registers. Then we can extract the MAC addresses from the incoming packets in the hardware and use comparators to determine the queue to which the memory index of the packet will be stored.

A tabulated and stylized form of the MAC-to-port translation unit would look like the following:

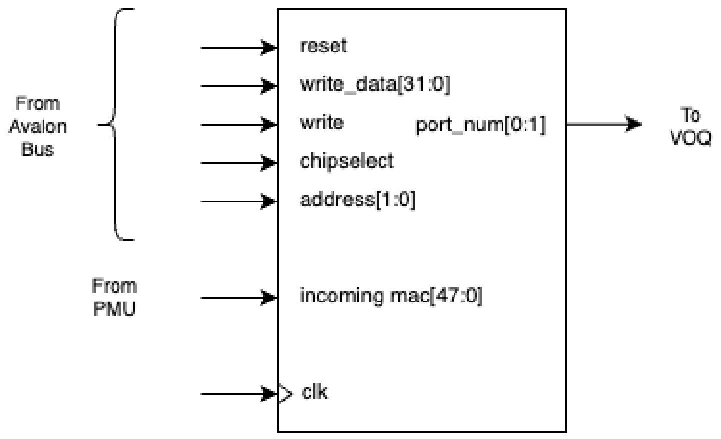


Figure 4.2: Mac-to-Port Translation Unit

4.2 Packet Management Unit (PMU)

There are four ingress ports and each port has one associated memory buffer managed by the Packet Management Unit (PMU). Each PMU will be made of two parts: the data buffer and the control buffer.

Following the minimum length packet size of 32 bytes, the data buffer will be broken down into 32 byte blocks. Each block will be used to store only packet information. The control buffer will be broken into 6 byte blocks. The first bit will indicate if the corresponding data block has already been allocated or not. Next 20 bits will store the address of the start of associated data block. The next 8 bits will contain the address of the next free control block if the incoming packet is larger than 64 bytes. This leaves 3 unused bits that can be used to store other information. The address of each control block will be 8 bits long, resulting in 256 control blocks, and 256 data blocks.

Therefore, each ingress port will need approximately (256(4bytes + 32bytes)) = 9216 bytes of memory for each ingress buffer, or 40 KB for all ingress buffers combined.

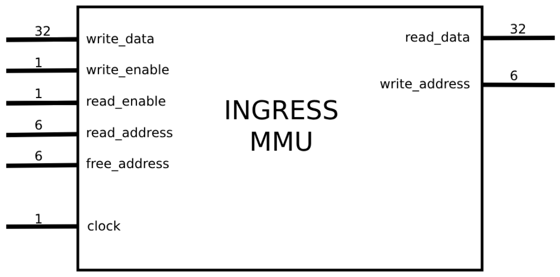


Figure 4.3: Block diagram for the ingress PMU module.

The packet management unit for each ingress port gets 32 bits at a time from software over the Avalon bus. The first 16 bits of the packet will contain packet length information, which will stored in the PMU. Every cycle, the PMU will receive new packet data and write that data to the buffer. The MAC address of the packet is stored in the next 48 bits or 6 bytes. The PMU will hold this information in a register and send over to the MAC-to-Port translation module.

The ingress buffer's PMU will store the following data in registers:

- **Packet length:** self-explanatory.
- **MAC address:** stores the MAC address of the packet and sends to the MAC-to-Port translation module
- **Free block:** a pointer to the first free data block within the control block
- **Block size:** how many blocks are currently being utilized to keep track of total available memory

PMU capabilities

- UPDATE FREE ADDRESS: takes a control block address. Checks if the block is free

or not. If not, continuously checks the next control block until a free block is found and updates the pointer to the next free block.
- STORE: Stores a maximum of 32 bytes to the first free data block 32 bits at a time. Calls UPDATE FREE ADDRESS with the associated control block to find the next free control block.
- WRITE: Checks if incoming packet length can fit into the total available memory. If it can, then continuously calls the STORE function to store the entire packet in memory 32 bits at a time. Returns the address to the first utilized memory block.
- RETRIEVE: Takes a control block address and retrieves the 32 bytes packet information from the associated data block.
- READ: Takes a control block address and length of packet. Continuously calls the RETRIEVE function until all packet information is returned.
- DELETE: Takes a control memory address. Sets the first bit to 0 to indicate that the block is no longer allocated. Returns the address of the next associated data block and sets them to 0.
- FREE: Takes a control memory address. Continuously calls the delete function to free all associated control blocks that were used to hold that particular packet information. Calls UPDATE FREE ADDRESS function.

4.3 Egress Buffer

There are four egress ports and each of them will have an associated ring buffer. Like typical ring buffers, the memory will be connected end to end using. Two pointers, one head and one tail, will chase each other to read and write at the correct addresses. The buffers will read out data in FIFO manner. The egress buffer will inform the software when new packets arrive. Software will request the egress buffers to read out the packet 32 bits at a time.

4.4 Virtual Output Queues (VOQ)

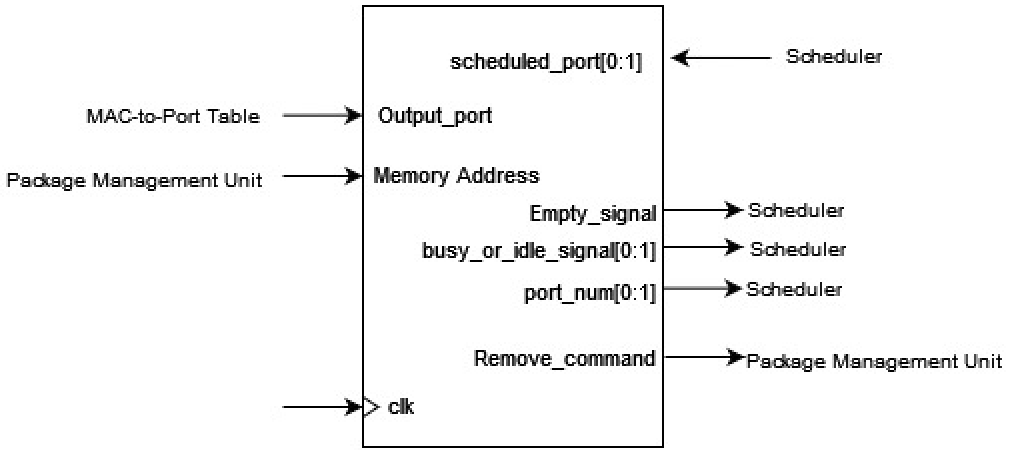


Figure 4.4: The VOQ interface

We will have one virtual output queue (VOQ) for each ingress port containing 4 RAM modules. Each time a packet is put into a queue, the memory address of this packet and the corresponding output port are sent from the PMU and MAC-to-Port module.

In terms of the interaction of the VOQ and the Scheduler, VOQ sends the output port of the current packet to the scheduler and receives the scheduled port from the scheduler, and it sends the remove command to PMU to remove that specific packet from memory.

Empty and busy_or_idle signals are used for interaction between the VOQ and the scheduling algorithm.

4.5 Scheduling Algorithm

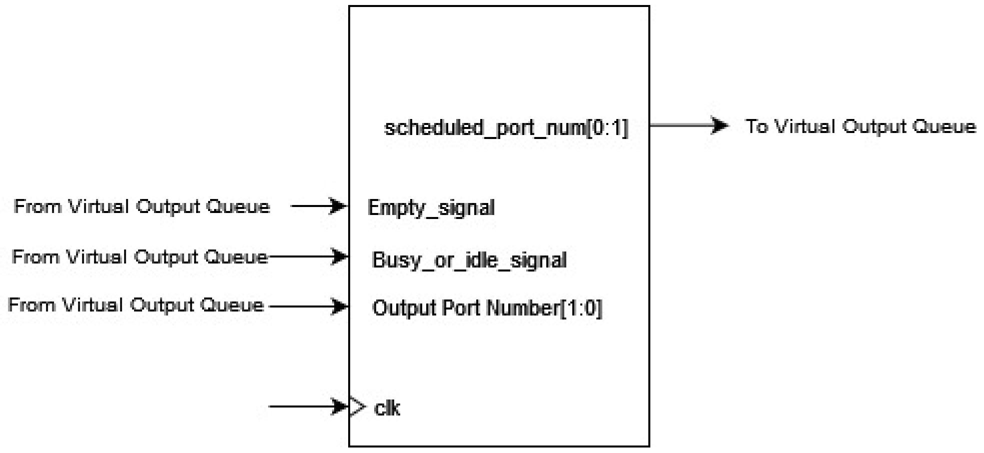


Figure 4.5: The scheduler interface

In our proposed network switch design, we employ a round-robin scheduling algorithm coupled with a busy signal mechanism to efficiently manage packet transmission of variable sizes across multiple input and output ports.

Round-robin scheduling is a well-established method for fairly distributing resources among multiple entities in a cyclic manner. For example, in a network switch with four input ports, the scheduler sequentially selects packets from each input port in a predetermined order: Port 1, Port 2, Port 3, Port 4, and then back to Port 1. In the context of our network switch, this scheduling algorithm ensures that each input port receives equal opportunities to transmit packets to the output ports, preventing any single port from monopolizing network resources.

4.5.1 Busy_or_idle Signal

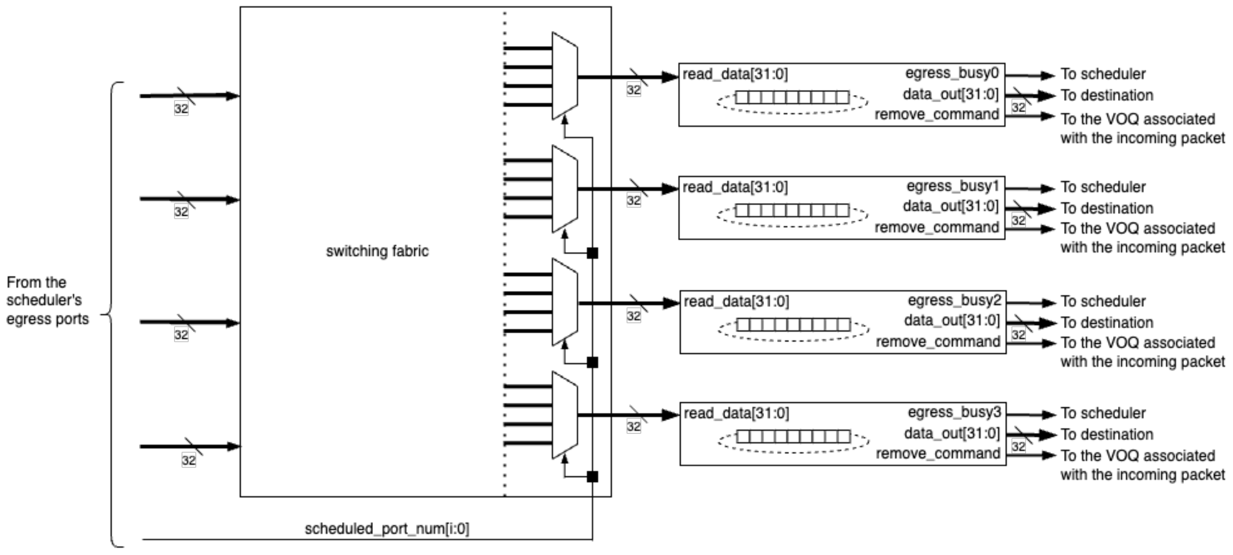
Since packets can be multiples of 32 bytes, and each block is 32 bytes, the scheduler needs to handle variable packet lengths efficiently. To handle variable-length packets, each output port is equipped with a busy signal. This signal indicates whether the port is currently busy and the round-robin scheduling should bypass that port in the current cycle.

4.5.2 Mechanism

1. At the beginning of each scheduling cycle, the scheduler checks the busy signals of all output ports.
2. Output ports with an active busy signal are excluded from the round-robin selection process for that cycle.
3. The scheduler proceeds with round-robin selection only among the non-busy output ports.
4. If all output ports are busy, packets in the input queues wait until at least one output port becomes available.
5. Once a packet transmission completes for an output port, its busy signal is reset, indicating that the port is available for scheduling in the next cycle.

4.6 Networking Fabric: Crossbar and buffers

A crossbar, also known as a switching fabric, is a network topology that consists of a grid of intersecting buses, enabling direct and exclusive connections such that at any given time, each output port is connected to only one input port. Crossbar switches are a critical component of any network switch, as it improves throughput by allowing multiple, non-interfering data connections to be in use at any given time. Since crossbars provide a single unique path from an input to an output, only one packet must be chosen from the buffers at the input.



÷