Master's Thesis

Network Interface Card and Switch integration

Diptyajit Choudhury

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, May 2014.

LUND
UNIVERSITY

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

MASTER OF SCIENCE THESIS

Network Interface Card and Switch integration

Author: Diptyajit Choudhury

Advisor: Robert Wikander

Examiner: Joachim Rodrigues

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

Printed in Sweden
E-huset, Lund, 2014

Abstract

Today server computers are being stacked in racks. A computer rack consists of 40+ [1] server computers connected to a top of rack switch. This top of rack switch is then connected to other racks. Together they form a large cluster of computers which are used in today's cloud computing. Each server has one or sometimes two Network Interface Cards (NICs) which are connected to the top of rack switch. The functionality of the NIC is to transport the packets from the server's main memory onto a standard such as Ethernet and then transfer them to the network. However today inside a server the protocol to transport packets in and from the main memory is already done by Peripheral Component Interconnect Express (PCIe) protocol. It can therefore be argued that the NIC's function is useless. Since it will only translate PCIe packets to Ethernet packets. Hence the NIC functionality, i.e. going from PCI express to Ethernet can be put into the switch instead.

In this project a NIC is built from scratch to understand how it works and understand networking and networking hardware to achieve this vision. Then the PCIe protocol is studied in detail and a customized PCIe IP is implemented and verified to prove that it can be utilized for the project. Hardware tests are conducted on a FPGA interfaced with a host computer using this protocol. Finally, a theoretical investigation of the costs and feasibility of supporting a new networking protocol called Quantized Congestion Notification (QCN) in a novel NIC-switch hybrid device is provided.

Acknowledgements

I would like to thank my supervisor, Robert Wikander who gave me the opportunity to work on this project. Both he and Per Karlsson at Packet Architects AB have given me ample suggestions on my designs, which allowed me to spot issues early on. I would like to thank my examiner Joachim Rodrigues for suggesting that I apply to the thesis opportunity. His feedback and positive criticism have been instrumental for my progress. Also, I thank the EMEA scholarship programme and the EM2 team at Lund University for the opportunity to come to Lund and study at Lund University. Lastly, to the people I love, thank you for your support and understanding.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AXI Advanced eXtensible Interface
BSCAN Boundary Scan
CPU Central Processing Unit
CRC Cyclic Redundancy Check
DCN Data Center Network
DMA Direct Memory Access
FCOE Fiber Channel Over Ethernet
FIFO First In First Out
FPGA Field Programmable Gate Array
FSM Finte State Machine
HDL Hadrware Descriptive Language
ISR Interrupt Service Routine
JTAG Joint Test Action Group
LIFO Last In First Out
LUT Look Up Table
MSI Message Signaled Interrupts
NIC Network Interface Controller
PCI Peripheral Component Interconnect
PCIe Peripheral Component Interconnect Express
PHY Physical Layer
PLL Phase Locked Loop
QCN Quantized Congestion Notification
QoS Quality of Service
RAM Random Access Memory
ROM Read Only Memory
RTL Register Transfer Level
SATA Serial ATA
SERDES Serializer/Deserializer
SR-IOV Single Root I/O Virtualization
TFD Transmit Flow Descriptor
TQD Transmit Queue Descriptor
VF Virtual Functions

Chapter 1

Introduction

1.1 Traditional Approach

The NIC is a modular hardware unit which is responsible for connecting a computer to a network. From the computer's point of view, the NIC is alike all other I/O devices and from the network's perspective it represents one unique node [2].

The NIC has two interfaces, one is the PCIe standard which connects it to the CPU via the motherboard, and the other is the Ethernet standard which is used to interface with the network traffic.

A switch (Figure 1.1) is a device used in networks to deliver a message from one node to another, for which the message is intended. There are different types of switches which have additional functions as well, but the basic underlying principle is the same.

Figure 1.1: A typical Switch showing Ethernet ports

Source : `http://goo.gl/ynKJjb`

Network traffic means packets of data, which need to be transmitted or received by a device. In the generic case of a computer which is part of a data center, the

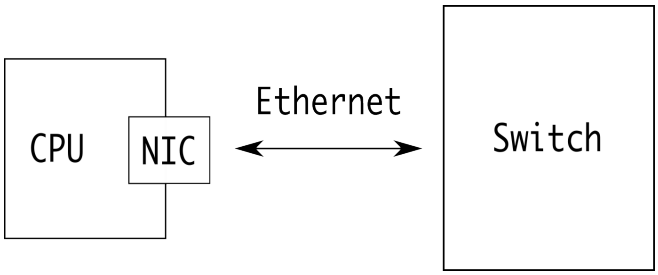


Figure 1.2: The traditional setup

traditional network architecture is shown in Figure 1.2. If one considers transmission of traffic, the CPU's task is to assemble a packet, which is then sent via the NIC to the switch and further transmitted to the packet's destination to the network. For receiving packets, the CPU must allocate space for the incoming traffic and relay this information to the NIC, which receives the data intended for this unique node from the switch.

The setup shown in Figure 1.2 has a problem. All the network devices such as NICs, switches, routers etc. have a small amount of memory (buffer memory) that they use to temporarily store the traffic that they are transferring. A scenario can arise, where programs or applications running on a device on the network attempt to transfer more packets per unit time than what is permissible by the network itself. A real-world example of this event is when a video conference is initiated from the same machine where a file transfer program was already active. This situation is called 'exceeding the bandwidth', and this causes the buffers to get filled up and/or overflow. When this happens, the network gets congested and packets start getting dropped. As a consequence of dropped packets, data transmission protocols such as TCP [3] and others [4] mandate a re-transmission of those same packets which were not delivered, which further increment the congestion in the network and result in throughput degradation.

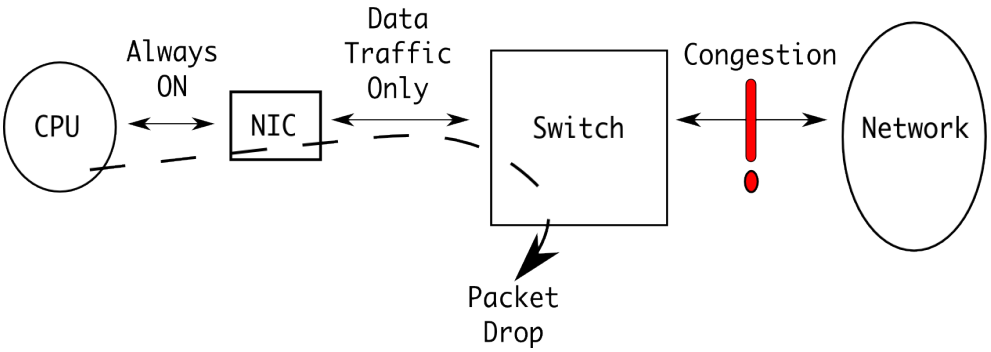


Figure 1.3: Dropping packets due to congestion

Multiple solutions have been proposed by investigating the problem from different

perspectives. For example, there has been previous research on managing buffer overflow in switches [5]. Another approach is highlighted in literature on network congestion avoidance and/or mitigation [6]. Others suggest a combination of the two approaches [7]. This thesis proposes a different viewpoint whereby the setup architecture is identified as the source of the problem. The argument is provided below. The information that the network is congested, is only available to the switch which tries its best to mitigate the problem. But, this information is not relayed to the CPU or endpoint which is causing the problem itself. It would be worthy to envision a means to let the CPU know that a network congestion has occurred and it should halt transmissions now. Instead, the CPU employs a 'fire and forget' approach, whereby it keeps dumping packets to the NIC which further dumps them to the switch and leaves it to deal with the problem. This is shown in figure 1.3.

1.2 Proposed technical solution

DCNs are unique in nature, because they must maintain high QoS and must strive for loss-less transmission of network traffic because of their choice of Fiber optic hardware or FCOE infrastructure does not allow any packets to be dropped [8].

Due to these requirements, the previously discussed generic architecture is unsuitable to perform in DCNs and several work arounds have been proposed. The proposed technical solution is embodied in the principle of an intelligent configurable switch called the *Flexswitch* working in tandem with a DMA engine called the *Hydra*. This proposal is explored in some detail below.

Typically DCNs consist of server racks, multiple computers serially arranged in a cabinet, each having its own unique NIC in its enclosure, connected to a switch using Ethernet [9]. This is shown in Figure 1.4.

The solution proposed in this thesis involves a novel device, the *Hydra* which consists of several *Hydra* NICs. Each *Hydra* NIC has two major components, a PCIe endpoint which connects to the host and a DMA controller which moves data between the host memory and the switch (and the other way around). From a host's point of view the *Hydra* NIC appears as an ordinary NIC, albeit with a somewhat extended feature set. The change in the architecture is shown in the figure 1.5.

Figure 1.4: Individual hosts in a DCN, from CERN.

Source : `http://goo.gl/Ep9ppL`

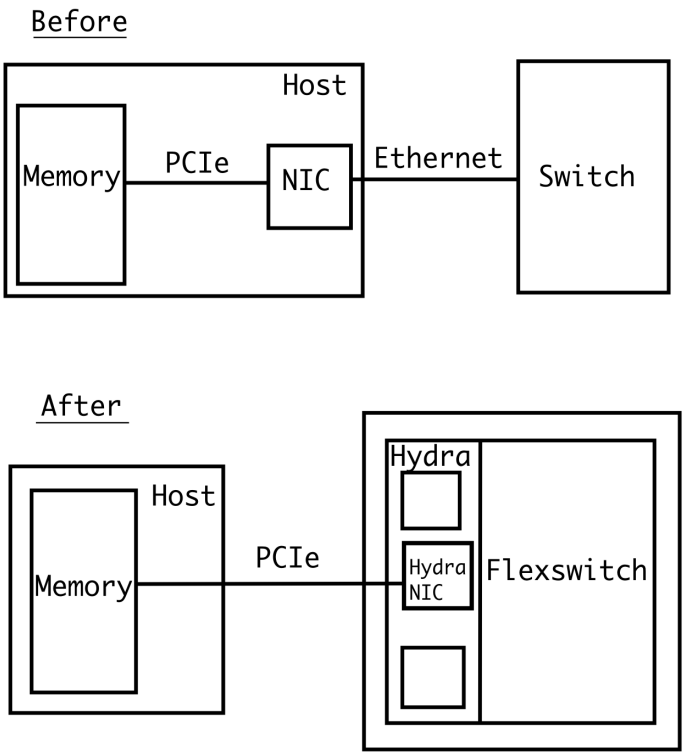


Figure 1.5: Proposed Solution

The possibilities which open up when a shift is made from the traditional setup to the proposed solution above, are listed below.

- PCIe bandwidth (256Gbps) is much higher than that of Ethernet (10Gbps) today, so low latency traffic transfer is handled more efficiently between the host and the network device.
- Power consumption (per host) is reduced because instead of individual NICs on the hosts, a physically separate module incorporating NICs is proposed. This enables the use of other architectural modifications in the *Hydra* which further reduces power consumed.
- Most importantly, since the *Hydra* and the *Flexswitch* are part of the same module, information about network congestion can be exchanged between them and this information can be relayed to the host without causing significant overhead, particularly due to PCIe's large bandwidth.

As can be gathered from the above discussion, the entire proposed solution is a complex device and it supports several features to make it a marketable and viable product. These include

- PCIe 3.0 specification (including backwards compatibility)
- Support for virtualization (virtual machines) using SR-IOV
- Proprietary interface with the flexswitch, which includes configurability according to customer needs
- Novel descriptor management architecture, to support upto 512 hosts (considering PCIe 1x channels for each host)
- Support for QCN which solves certain problems associated with network congestion

Since implementing all these features is a time consuming and intensive task, the purview of the thesis is limited to a subset of these.

1.3 Scope and structure of thesis

In the first part of the project, a NIC is built using Verilog HDL. This step is requisite to gain familiarity with networking and networking hardware. The first chapter discusses the entire design process and presents the internal architecture of a NIC. This is followed by an implementation of this architecture which adopts a modular approach. The testing strategy is then presented, along with the results of the simulation.

The second part of this project is devoted to studying the PCIe interface, assimilating this industry standard protocol and investigating if it can be used to achieve

the features that are part of the proposed solution. Initially, the thesis focusses on understanding the theoretical structure and underlying signaling of the PCIe interface. Once that is completed, the focus shifts to working with the standard in hardware and tailoring it to the solution's needs. Since this is the development phase and rapid redesign is necessary, a Xilinx FPGA board that includes a PCIe port is chosen as the target platform. A generic barebones PCIe soft IP which is bundled with the Xilinx ISE design suite is chosen as a starting point, and several changes and additions are incorporated. This leads to a customized PCIe IP which is now a product that can be reused by the company where the thesis is conducted. Additionally Linux drivers are designed and implemented at each stage of the PCIe exploration to debug the system at a higher level. In addition to that, the Xilinx Chipscope tool is utilized to debug the live hardware. The goals, design procedures and results from this part of the thesis are discussed in the third chapter.

Once it is seen that PCIe can actually be used to achieve the results envisioned with the *Hydra*, it is imperative to conduct a feasibility study on the feature set that must be supported by the final product. This includes evaluation of hardware footprint, optimization of the same and exploration of co-designing with software. The most challenging idea is to study the feasibility of implementing QCN, and how the different trade-offs associated with the same can be managed. This is important because the competition either does not support QCN, dubbing it as "ineffective in today's FCoE networks" [10] or supports a very small subset of what the proposed device is equipped with. This section of the thesis is discussed in the fourth chapter.

Chapter 2

Building a NIC

2.1 Overview of goals and steps

This step of the project is dedicated to building a NIC. The goals of this step are listed below.

- Build a functional DMA engine, to lay the groundwork for the proposed solution (Hydra).
- Build a functional MAC which can handle Ethernet packets.
- Investigate, analyze and implement hardware components such as shared memories, FIFOs and linked lists.
- Identify and point out issues with the traditional setup discussed in the previous chapter.
- Gain familiarity and work with the Ethernet standard in hardware.
- Acquire an overall design experience of networking and networking hardware.

Figure 2.1 shows the basic functionality of the NIC. There are two directions that packets can flow.

The first direction, henceforth called *RxPath* is composed of packets from the network which are received by the host. The NIC accepts packets from the network which are encapsulated in a particular format which is dependent on the network chosen. The NIC stores the packet in the on-chip buffer memory, and then stores it in the main memory of the host through a direct memory access (DMA) engine. The second flow of packets occurs when the host machine needs to transmit packets to the network, and this is called the *TxPath*. The NIC should retrieve those packets from system memory, package them as required by the network and send them through the medium access controller (MAC). At any point of time, both the Rxpath and the Txpath can be active, and maximum efficiency is achieved when

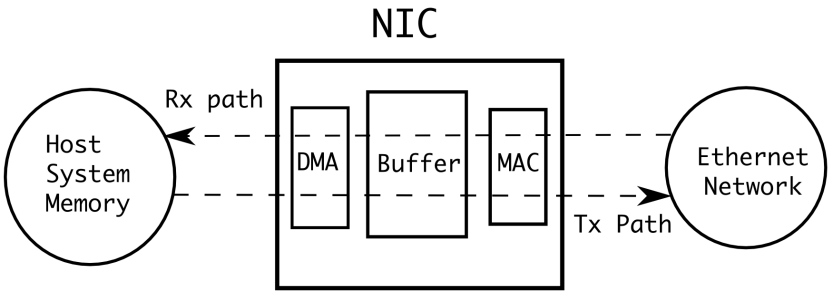


Figure 2.1: NIC functionality

both these flows are at maximum bandwidth. At this high level of abstraction, the NIC built in this thesis is comparable to the RiceNIC [11].

This chapter will first present the interface of the NIC and cover its design specifications according to the interface. Then, it will propose an architecture for achieving the specifications and describe the function of the different modules, justifying design choices when they are encountered. This will be followed by a presentation of the testing strategy and the reasons why the test bench was made in a particular way(s). Then, it will show the results of the behavioral simulations.

2.2 Interfaces and Specifications

The NIC consists of two interfaces, a 10Gbit Ethernet port at the network side and a memory access interface at the host memory side. The interfaces are discussed below.

2.2.1 Ethernet packet format

An Ethernet packet is shown in the Figure 2.2.

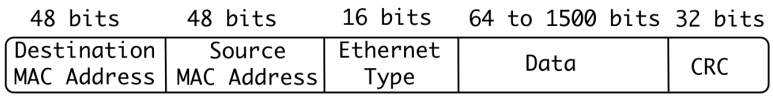


Figure 2.2: Ethernet packet

Essentially the packet contains two distinct parts. The data itself, which is the payload, and various other information regarding the data to be transferred. These include the source and the destination MAC addresses which are used for routing the packets to the correct destination, and then identify from which device they were originally sent. There are also mechanisms by which one can check if the

packet has been damaged since it was sent, and Ethernet packets include a CRC code at the tail of the packet to make such checks if and when required.

2.2.2 Ethernet interface

The actual interface that is used by the NIC on the Ethernet side needs to be discussed. This is shown in figure 2.3 on the right hand side of the NIC.

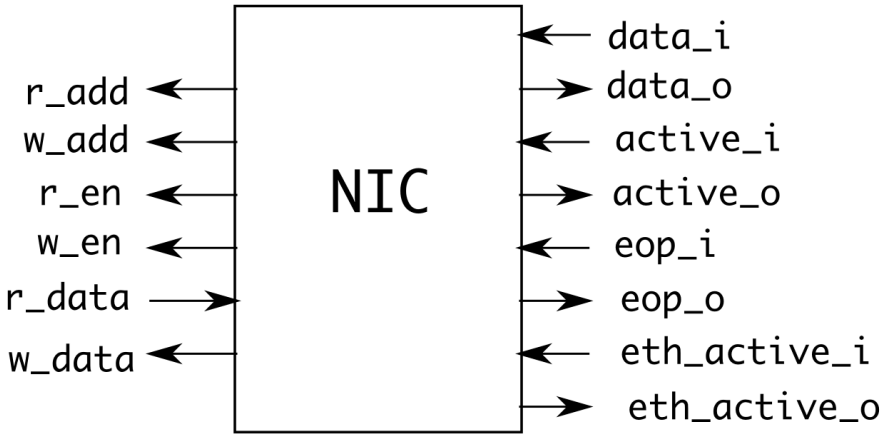


Figure 2.3: NIC interfaces

The data port(s) is a 64 bit port through which the Ethernet packet must be received and transmitted, and there are two data ports on the NIC, one for the Rxpath and the other for the Txpath. The word *data* might be misleading as this has no connection with only the payload of the packet; the entire packet is moved through the data port(s), 64 bits per clock cycle. As long as the *active* signal is high, the data port is either receiving or sending a packet. To mark the end of the packet, the *eop* signal is set to high during the last 64 bits being transferred. This is especially useful to distinguish between two packets which arrive without any inter-frame gap,(i.e the last 8 bytes of a packet are immediately followed by the first 8 bytes of a subsequent packet in the very next clock cycle), and the *active* signal does not go low, because there is no pause in packet traffic. The 3 bit *ethernet_active* signal indicates how many bytes of the current data in the data port is actually valid. This is required to transmit packets of any size, using the same data port. For example if a packet of 65 bytes needs to be transmitted, during the first eight clock cycles, 8 bytes will be transmitted using the data port, while *ethernet_active* remains 000 indicating all bytes of the data are valid. In the last clock cycle, the 65th byte will be padded with 7 bytes of zeroes and transmitted through the data port while the *ethernet_active* will be 001 indicating that only the first byte of this data is actually active.

2.2.3 Interface to main memory

Most NICs in today's computers use either the PCI or the PCIe protocols to connect to the motherboard of the computer [12]. Since these protocols are complex industrial standards implementing these protocols was beyond the scope of this part of the project. Instead, a simple memory interface was used and shown in Figure 2.3. There are separate address buses for reading and writing, coupled with separate read enable and write enable signals. Essentially the module which mimics the main memory of the computer is a dual port RAM and this allows us to adopt this simple interface. The important design choice made at this juncture was the width of the data bus(es) between the main memory and the NIC. The constraint was actually determined by the Ethernet interface discussed in section 2.2.2, and this is elaborated in section 2.4.2.

2.3 Architecture

The architecture of the NIC is presented in Figure 2.4. The approach is to have separate modules for each of the tasks that the NIC must handle simultaneously.

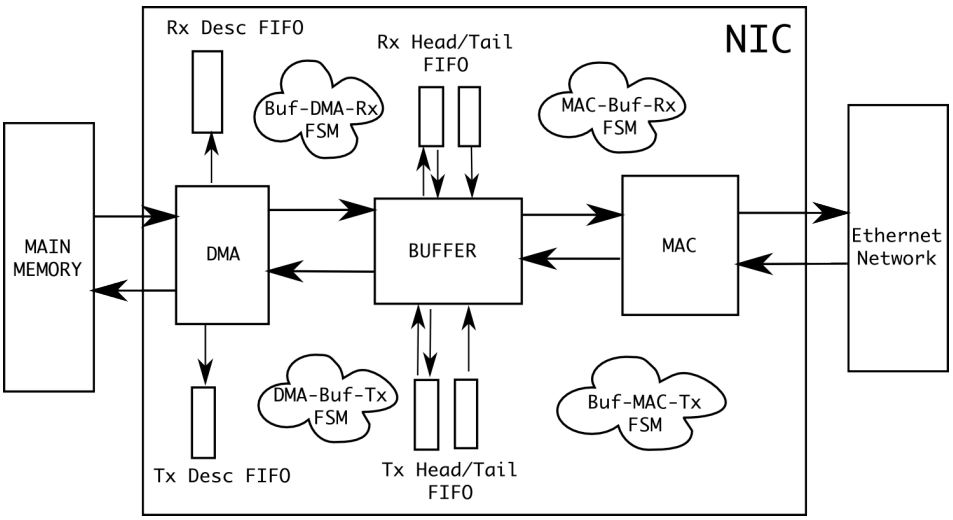


Figure 2.4: NIC architecture

- Firstly, the NIC must be able to receive and send Ethernet packets to and from the external world. This is done by the block called the MAC.
- The next task that the NIC should perform, is to transfer data packets to and from the main memory of the computer. Since the computer itself should not be interrupted for such transfers, the NIC should be able to transfer this data by directly accessing the memory, and hence this task is covered by the DMA block.

- The final task of the NIC, is to store packets when either the DMA or the MAC is busy, and to do this the NIC has an on-board memory. To implement this model and to carry out the NIC's final task, the block 'Buffer' is used in the architecture.

These three blocks constitute the data path of the NIC. The control path is separated from these blocks and is comprised of state machines. In addition to these blocks, there are two *Descriptor* FIFOs and two pairs of *Head-Tail* FIFOs. The *Head-Tail* FIFOs are used to keep track of the order of data packets, for example, packet 'x' received before packet 'x+1' from the Ethernet must be stored in that order in the main memory. Each pair is dedicated for each flow of packets (Rx path and Tx path). The *Descriptor* FIFOs store descriptors, which indicate addresses in the main memory where incoming packets must be stored (for the Rx path) or outgoing packets must be retrieved for transmission (Tx path). Each of these blocks or modules are discussed in detail below. This will be followed by a description of the control logic used in the design.

2.4 Buffer

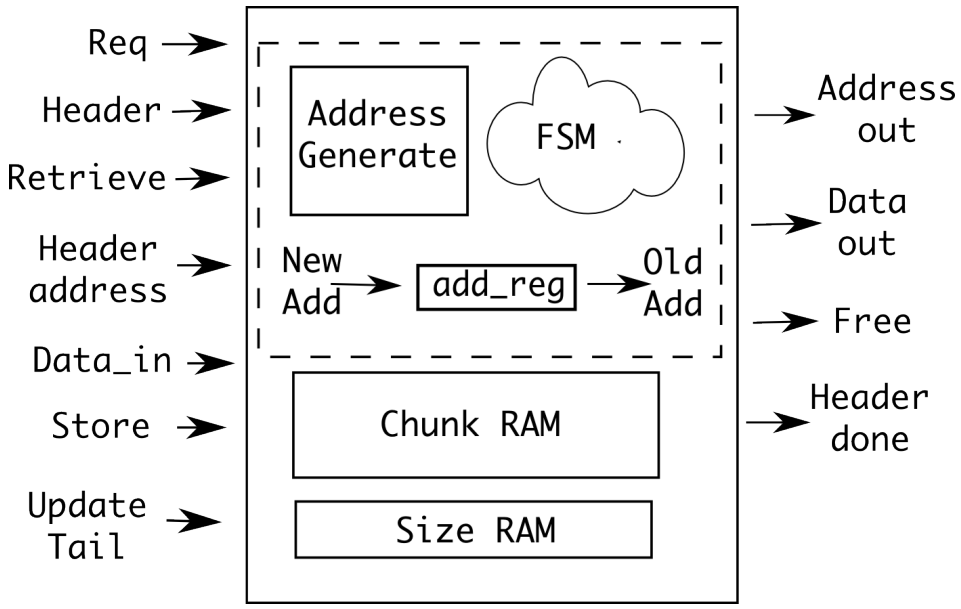


Figure 2.5: The Buffer block

The Buffer block is shown in Figure 2.5 and has three components, the address generator, the chunk memory, and the size memory. The address generator is used to generate new (or reusable) addresses where data will be written. The chunk memory stores the packet data. Every index in the chunk memory has a corre-

sponding index in the size memory which is used to store information regarding how much of the entire chunk is actually valid data. This feature is essential in order to support packets of varying sizes, where only few bytes of an entire chunk might contain the useful information. Each of these components are discussed in detail below.

2.4.1 The Address Generator

At the onset, it is to be established as to why the address generator is required. The memories in the buffer are initially empty. When the MAC or the DMA needs to transfer data to and from the buffer, it is preferable to have the address management logic (for chunk memory and size memory) independent of these blocks so as to simplify the interface between the Buffer block and the MAC as well as the buffer and the DMA. Read operations will be investigated in a later section, but for write operations to the buffer memory, the address generator manages the locations where data is written into. This management is done by using a LIFO data structure, complemented with a counter and is shown in Figure 2.6.

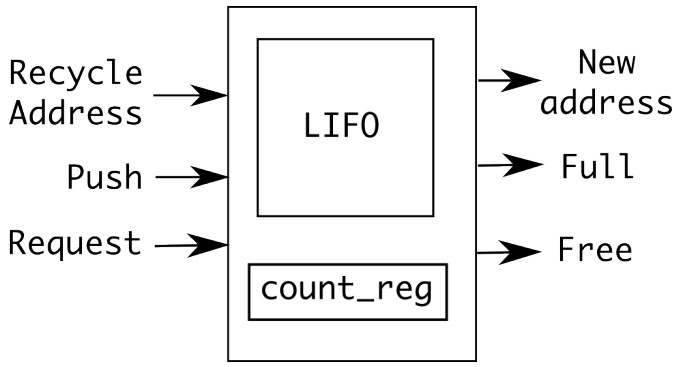


Figure 2.6: The Address Generator

When the device is reset, the counter is initialized to zero, and it is assumed that the buffer memory is empty. The counter can count up to the last location of the memory, which is determined by the (configurable) size of the memory. If there are 'n' locations in the memory, for the first 'n' write requests, the counter will generate the address and the address generator will send this address to the memory for the corresponding write request. Owing to the limited size of the memory to maintain area constraints of the chip, one is bound to run out off these 'n' locations in a finite amount of time. Hence, the addresses must be re-used after the corresponding data has already been read from the memory and there is no need to store the data anymore. This is done by using the LIFO. When a read operation is completed in the buffer block, subsequently the address from which the data was read is now pushed into the LIFO. For the 'n+1' th write request, we check the LIFO, and if it is not empty, an address is popped from it, and this is

forwarded by the address generator to the memory for the relevant write request. In this way addresses are successfully reused for future writes.

When the counter's value is 'n' and the LIFO is also empty, it means that all 'n' locations in the memory are currently occupied by data, and none of these locations have been read so these memory locations cannot be recycled. In other words, at this stage the buffer memory is full and the address generator cannot generate an address for a subsequent write request. Since we don't want to start dropping packet data, we prevent this new write request from being generated by asserting a signal 'Buffer Full' which indicates the control logic to wait till addresses are freed.

2.4.2 The Chunk Memory

The chunk memory is where we store the actual packet data and it has been implemented as a dual port RAM. It has a simple interface, identical to the of the main memory discussed in 2.2.3. Some features of the memory are explored below.

To ensure that maximum throughput is achieved, the memory model must be of a 'shared memory' type. This can be verified using a simple example where we use separate memories of predefined sizes for the separate Rx Path and Tx Path. If due to network congestion or faults, the network traffic consists of just one type of data (say data only in the Rx Path), the predefined (separate) memory for the Rx path will inevitably become full and the device will have to start dropping incoming packets because it has nowhere to store it. Meanwhile, the (separate) memory block for the Tx path is still empty and unused. This leads to a simultaneous drop in performance as well as underutilisation of resources. The better choice is to opt for a shared memory space which can be populated with data packets from either path. This choice does introduce a need for tagging memory locations with the flow (Rx or Tx) that they are associated with (for identification during subsequent read requests), but this is easily achieved by using two separate FIFOs outside the Buffer block. This the tagging mechanism is pursued later in section 2.4.3, and for now, the size of the data ports of the chunk memory is justified.

The specification of the Ethernet port is 10Gbit. So, every second 10G bits will enter(or leave) the device. However as we have already seen in section 2.2.2, the port through which the Ethernet data is transmitted, is only 64 bits wide. Hence, the clock frequency with which the device should operate is given by

$$f = \frac{10 * 1000000000}{64} = 156.25\,MHz$$

To achieve this frequency for the entire design, the memory in the buffer block must be able to handle reads and writes at the same frequency.

Hence, the frequency constraint on the memory means that the number of bytes read from or written to the memory in every cycle must also be chosen correctly. It must be kept in mind that both the DMA and the MAC have read and write access

to the on-chip buffer memory, and since the memory is a shared memory, it has only one write port and one read port. A situation may arise where the DMA has initiated a read request for a packet on the RxPath, and before that is completed, the MAC also issues a read request for another packet on the TxPath. Since only one read port is available, the easiest solution is to queue such requests and handle them sequentially. However this solution will severely affect the throughput of the device, as the NIC and the MAC will no longer be independent of each other. Each block has to wait till the previous block which made the read or write request is serviced, and only then will the current block's request be satisfied. This will result in under-utilization of the external ports, that is, no transmission will be made even when the ports are idle, and the only reason is that a pending read(or write) request.

An alternative solution is to read or write in bursts, where data *chunks* are moved between the memory and the peripheral blocks(the MAC and the DMA) for every request. In this scenario, when the MAC or the DMA issues a read(or write) request, it will receive or send a chunk of data at a time. This will significantly reduce the number of memory accesses that the peripheral blocks must issue to transfer an entire packet. For example, consider a packet in the TxPath, of length 256 bits, that must be read from the buffer memory by the MAC and then be transmitted to the external network. Now, since the data port on the MAC is 64 bits wide (as mandated by the Ethernet interface in section 2.2.2) the logical decision is to transfer 64 bits of the packet from the buffer memory to the MAC in every clock cycle, and transmit them to the network. So, for the current example this will take 4 clock cycles. Now, consider that the DMA was busy transferring a packet for the first two cycles of this operation. At the beginning of the third cycle, the DMA is ready to transfer a new packet on the RxPath and issues a read request to the buffer memory. Since the buffer memory is already processing a read request, this new request will not be serviced immediately, but has to wait for two more cycles till the read from MAC is finished. Now instead of reading only 64 bits at a time, let us assume that the data *chunk size* is fixed at 256 bits. Then, the MAC can read all 256 bits of the packet in one cycle, and the aforementioned conflict with the DMA read request is avoided. After the chunk read, the MAC now needs to store the 256 bits in a register (internal to the MAC), and transmit 64 bits per clock cycle from this, and is thus no longer influencing the speed at which the DMA can read from the buffer memory. This solution does introduce overhead in terms of hardware, but the throughput is increased considerably.

Now that it is clear that the data chunk transfer is the preferred mode of reading or writing to the memory, the next step in the design process is to choose an optimal chunk size. When the size of the chunk is progressively increased, frequency of read and write requests to the memory by the peripheral blocks is progressively reduced. This also decreases the probability of a request conflict. However, a larger chunk size also means larger registers, more chip area, and more power consumption. Thus choosing the size of the chunk is an important design decision. At this junction, it must be recalled that the frequency of operation should be 156.25 MHz and hence the frequency with which chunks are written in or read out from the memory should be roughly equal to this. This chunk rate can be equated

as

$$Chunk\ rate = \frac{Packet\ Rate}{Chunk\ Size}$$

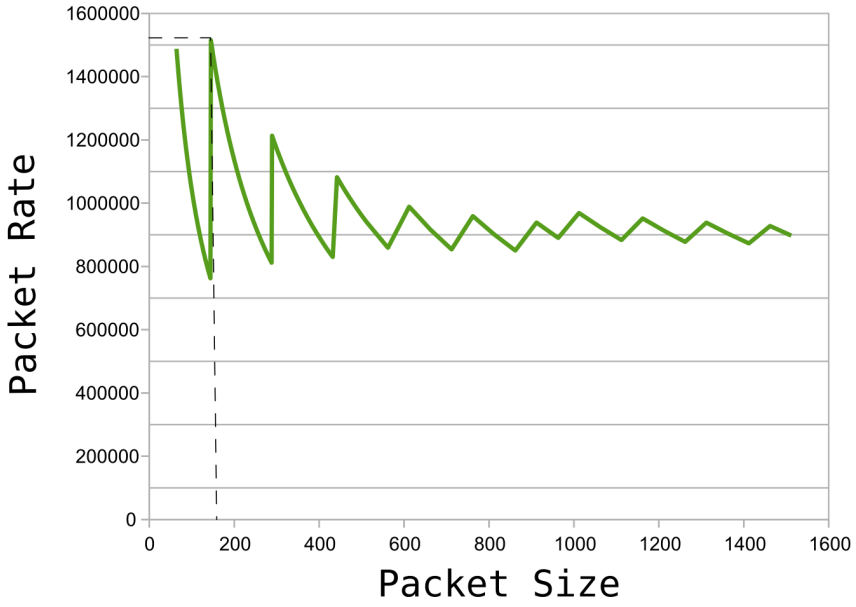


Figure 2.7: Packet rate as a variation of Packet size

For this discussion, refer to Figure 2.7. In the X axis of this graph the packet size has been varied as allowed by the ethernet standard in section 2.2.2. With this variation in mind, the packet rate, which is determined by the specification of the NIC (10Gbit), is plotted on the Y axis. The highest packet rate and its corresponding packet size is marked on the graph with the dotted line. The chunk size is now chosen so that it is equal to the packet size for which the packet rate is maximum. This choice is justified because it can ensure performance even when the NIC is operating at maximum load and maximum throughput. From this calculation and the graph above, it was decided that the chunk size will be 144 bytes (1152 bits). Furthermore, to maintain a uniform design the width of the data ports of the main memory is also kept as 144 bytes.

2.4.3 The Size Memory and the Head-Tail FIFOs

This is another memory in the Buffer block. The size memory contains as many index as the chunk memory so that there is a one-to-one correspondence with it. Since the useful data in a chunk may be less than the maximum size of the chunk, this memory is necessary to keep track of the payload data. The tagging mechanism introduced in section 2.4.2 is explained below. Since the *Size* and *Chunk* memories are both shared memories, it is not mandatory that consequent

address locations contain consecutive fragments of a single packet. What this translates to, is a requirement for keeping track of the memory locations which contain all the fragments of a single packet. The solution to this problem is to use a linked list, and this is where the Rx and Tx Head/Tail FIFOs come into action. Essentially, these FIFOs shown in Figure 2.4 store the memory index of a complete (chunk and size) entry in the buffer block. An example of the entire process is provided. Initially these FIFOs are empty, and once we get a number of Rx packets (so that a chunk can be written) in the MAC, a chunk write is performed. The address generated by the address generator is pushed onto the Rx Head/Tail FIFO and the same address is used to store that chunk and its corresponding size on the chunk memory and size memory respectively. Since its a FIFO, this action can be continued until the DMA is ready to transfer the chunks into main memory (system memory) of the host. When the DMA is ready, the addresses stored in the FIFO is popped, and the chunk which has been residing in the buffer for the longest time (i.e the packets which came in first) will be moved to the DMA first, and then the subsequent packet and so on. The size of the chunk from the *Size* memory is also transferred to the DMA and then forwarded to the main memory because the physical host machine should also be informed about how much of the chunk stored in its system memory is actually useful data.

2.5 MAC

The MAC is the block which interfaces the NIC to the ethernet network. So, it must include the ethernet interface as outlined in section 2.2.2. Almost all leading manufacturers have their own implementation of a MAC available for consumers. Popular choices in this increasingly competitive market included Xilinx's 10GEMAC [13], Arasan's XGMAC [14] and Altera's XAUI [15]. Though the basic functionality explained below is the same for these popular alternatives and the MAC developed here in this thesis, it is clarified that the implemented MAC does not have a PHY module (beyond the scope of the thesis), so it lacks the ability to be hooked up to a physical copper medium and start transmitting packets.

The MAC (shown in Figure 2.8.) is responsible for the interaction with the external Ethernet network. It receives and transmits packets using 64 bit data ports to the Ethernet interface. It is also connected to the buffer block and these two blocks interact by using read and write chunks of 144 bytes. The biggest challenge in this block is to maintain the integrity of the data flow even though data ports on either side of the MAC are so different in size. This challenge is tackled by adopting a fine-grained control of the data flow and the solution is described below. The MAC has two internal registers which are the same size as the determined chunk size (this is configurable as well). These registers allow the MAC to perform serial in parallel out (SIPO) and parallel in serial out (PISO) conversions on the data packets. For example, when packets are being transmitted to the Ethernet network, the MAC will first receive a chunk of data and store it in one of these registers. Then it will serially transmit 8 bytes from this register onto its *eth_data_out* port over 144/8

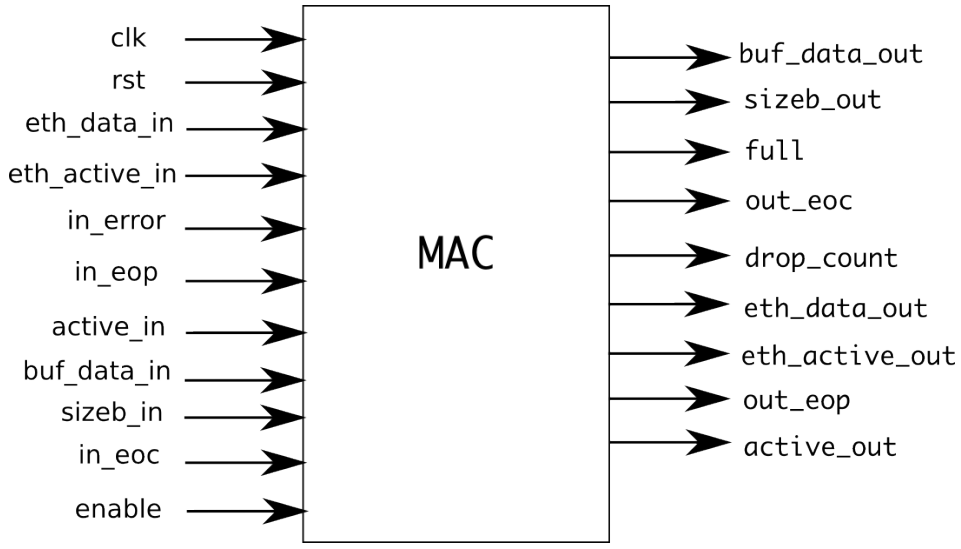


Figure 2.8: The MAC

= 18 clock cycles. And in this way it performs a PISO function. The reverse is applicable for packets on the RX path.

On the RX path, the MAC has to fulfill another function as well. Since the *size* value which we associate with every chunk is required for our internal architecture and is not available from the Ethernet network directly, when the MAC (serially) receives the 64 bits of Ethernet data per clock cycle and stores them in the internal SIPO register (for a future write to the buffer), every cycle, the MAC must use the 3 bit *eth_active_in* signal to ascertain the valid bytes of the incoming 64 bit data. As discussed earlier this value can range from 000 (indicating all bytes are valid) to 111 (indicating the first 7 bytes are valid) and accordingly, the MAC increments the *size* value associated with that particular chunk which it is handling in the SIPO register. When the register is full and the buffer indicates that it is ready for a write, the MAC will write the contents of both the PISO register and this size register into the buffer's chunk memory and size memory respectively.

An additional port named 'full' has been added to the MAC which was not visible in the original sketch of the NIC interfaces in Figure 2.3. This is used to let the external Ethernet network know that the internal buffer of the NIC is full and it should not send any more packets. If the network still persists in sending any more packets when the value of 'full' is high, then the packets will be dropped and since that is an important metric of evaluating a networking device, we have added a 'drop_count' port in the MAC which keeps a record of the packets dropped.

As part of its function to transmit Ethernet packets which are compatible with the external ethernet network, the MAC also maintains appropriate values on the *eth_active_out* and *active_out* signals. To achieve that, when its making the PISO conversion for the TX path, it monitors the 'size' value associated with the

chunk (both the size and chunk have been retrieved from the system memory by the DMA, then stored in the buffer, and finally read by the MAC for transmission), deducts the number of bytes it is transmitting every cycle from this value, and uses a look up table to generate the *eth_ active_ out* for the next data transmission.

2.6 DMA

The DMA block shown in Figure 2.9 is responsible for transferring packets between the buffer and the system memory of the host. To allow this interaction with the system memory, the DMA should be informed about the memory locations (in the system memory) which either store packets which need to be transmitted and the memory locations where incoming packets (from the Ethernet network) on the RX path can be stored. To achieve this, *descriptors* which are addresses pointing to the aforementioned memory locations are used. There are two types of descriptors : RX and TX and in the real world case, its the (host machine) operating system's task to manage these descriptors. Since interfacing with a physical host machine is investigated in a later part of the project, a hardware based descriptor management system consisting of two FIFOs was implemented in the form of the RX descriptor FIFO and the TX descriptor FIFO (refer Figure 2.4).

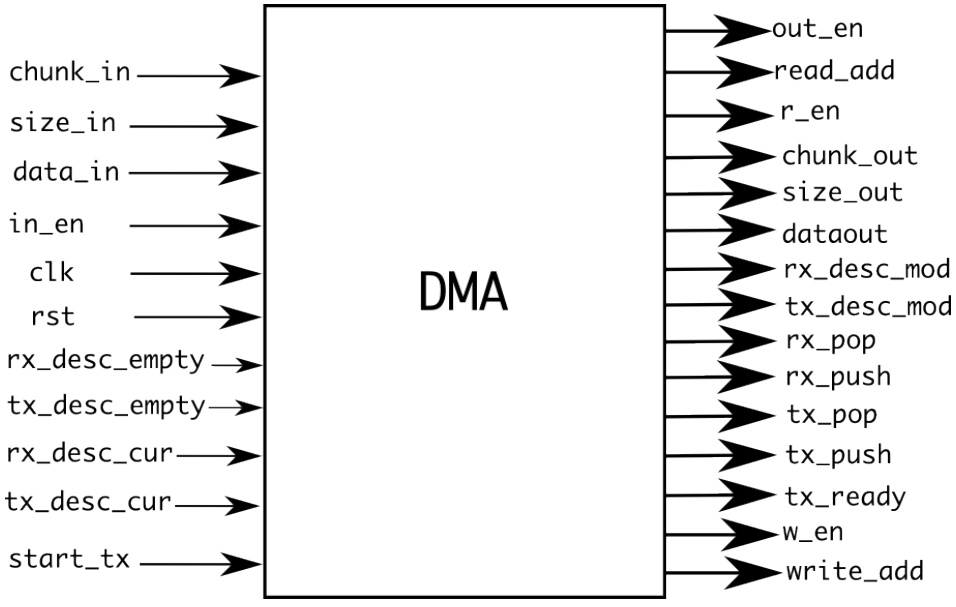


Figure 2.9: The DMA

The descriptor FIFOs are part of the test-bed that will be discussed later, but to understand the DMA's functions an example is presented here. Consider a TX packet is ready to be transferred in the system memory. These are the events that occur in sequence.

- The memory location where this packet is stored, will be pushed to the TX descriptor FIFO.
- As soon as this FIFO is not empty, the control logic will inform the DMA that a descriptor is available, which is pointing to a packet waiting for transmission.
- The DMA will pop the FIFO and get this address. It will use this address to generate a read from the system memory and store the response in an internal register which has the same size as the chunk used in the buffer. It will also retrieve the *size* value associated with this chunk from the system memory.
- Once a chunk is loaded, the DMA will inform the buffer control logic that it has a pending transmission request and it needs to write to the buffer.
- If it is not full, the buffer will generate a new address or reuse one of the previously used addresses (as discussed in section 2.4.1) for its internal memories and indicate to the control logic that it is ready to receive the TX packet from the DMA.
- The DMA will write the chunk to the buffer.
- The DMA will now update the TX descriptor, indicating that the packet has been sent to the buffer and that the host can now reuse this address to store another packet it needs to transmit.

A similar set of steps are followed when a packet is being received, but the data flow is the other way round (from the buffer to the DMA and then to the system memory). A glance at Figure 2.9, reveals a number of ports which were previously not seen in Figure 2.3. These have been added to allow the interaction with the control logic, managing the descriptors and communicating with the descriptor FIFOs.

To maintain the order of these steps and ensure that the DMA reacts as outlined above, FSMs are used within the DMA block in conjunction with the control logic. The TX FSM used inside the DMA is shown in Figure 2.10.

A walk-through of these states is presented to get a clear understanding of how the DMA manages to interact with the data flow and the control logic, and the steps highlighted above are included in the discussion for the sake of clarity.

- **Initial** : In this state the DMA is idle and its is waiting for packets that need to be transmitted from the system memory. As soon as the TX descriptor FIFO is not empty, (indicating a pending transmission) the FSM goes to the next state.
- **Retrieve Descriptor** : In this state the DMA will pop the TX descriptor FIFO to get the descriptor which is pointing to the packet (in system memory) waiting to be transmitted. This will take one clock cycle and then the FSM will proceed to the next state.

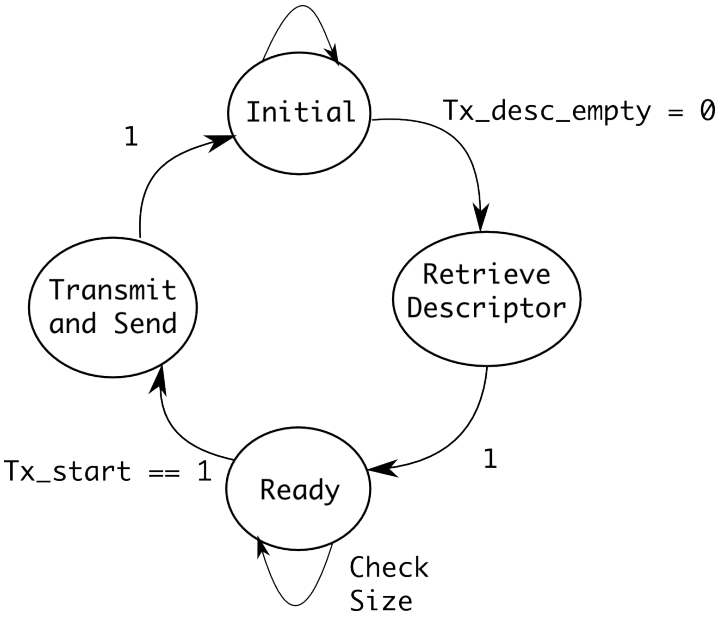


Figure 2.10: The DMA Tx FSM

- **Ready** : The DMA will use the address in the descriptor and fetch the chunk and its size from system memory. This will take one clock cycle. After that, it waits in this state until the buffer indicates that its ready for a write (tx_start=1).
- **Transmit and Send** : The DMA will complete the *transfer* of the chunk to the buffer block. Then it will update the current descriptor and *send* it to the descriptor management system. Once done, the FSM goes back to the idle state, waiting for the next descriptor.

2.7 Control Path

From Figure 2.4 it can be recalled that the control paths are four FSMs which lie between the three principal blocks viz. the MAC, the DMA and the buffer. The naming convention followed (for these FSMs) is based upon which blocks they are between, and the flow (RX or TX) that they are controlling. For example, the control logic FSM between the Buffer and the DMA which controls the Rx path, is called the Buffer_DMA_RX_FSM. These FSMs are responsible for these functions indicated below.

- They ensure that the MAC, DMA and the buffer follow the steps that were highlighted in the above sections.

- They interact with the FIFOs in our architecture making sure that they pop and push as required by the blocks.
- They interact with the internal architecture in the buffer, control the address generator when required.

To get a clearer picture of how they are implemented, the DMA_Buffer_TX_FSM is investigated in detail. It is shown in Figure 2.11.

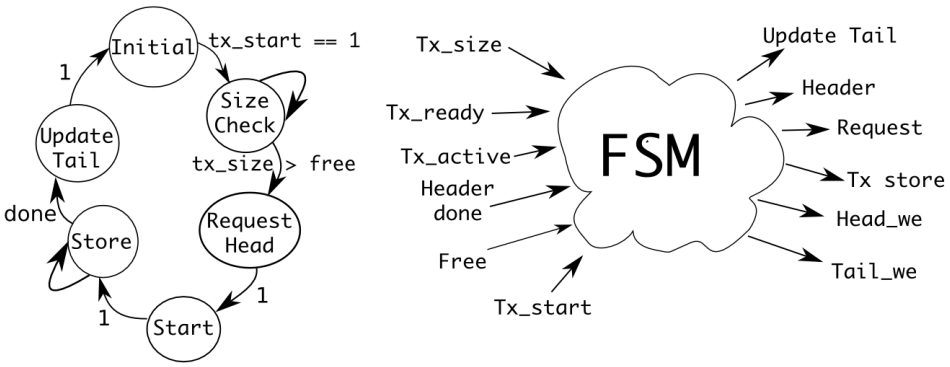


Figure 2.11: DMA Buffer TX control logic

Each state in the FSM is described below.

- **Initial** : This is the idle state, the control logic is in this state until the DMA indicates that it has successfully transferred a packet from the system memory and is waiting to transmit (tx_start=1).
- **Size Check** : Now that the controller knows that the DMA wants to write, it will first check if the buffer is full, whether there is enough space on the buffer to store this write packet (tx_size > free). If not, it waits till the buffer is freed of some memory.
- **Request Head** : The controller prompts the address generator for a free address (it can be a new address or one which can be reused) in the buffer memory.
- **Start** : Now, the controller has a free address to use on the buffer, and the DMA is also ready for the transfer. So, in this state the controller instructs the DMA to start the write to the buffer.
- **Store** : In this state, the DMA finishes its transfer and the data is stored in the buffer.
- **Update Tail** : Now that we have a valid data in the buffer, and the transfer is completed, the address where this transfer was made, is pushed to the TX FIFO so that we know which packets are scheduled to be transmitted and in which order. Once this process is completed, the controller goes back to the initial state.

All four controllers work in a similar fashion.

2.8 Testing

The principal objective of this part of the thesis is to get an in-depth knowledge and hands-on experience of networking and networking hardware. Thus, it is sufficient to have a functional verification of the same, and the NIC was not tested on a physical hardware platform. Instead, a test bed was designed, and this is shown in Figure 2.12.

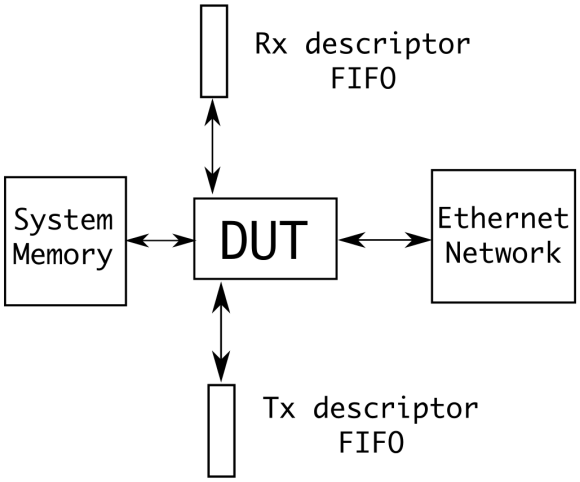


Figure 2.12: The test-bed around the DUT

The blocks around the NIC design under test (DUT) are explained here. The system memory is modeled by a dual port RAM. At the start of the top level test, packets are written into the system memory (mimicking the operating system's role) and these addresses are updated in the Tx descriptor FIFO. Furthermore, a few addresses which are empty in the system memory block, are pushed into the Rx descriptor FIFO for incoming packets.

The Ethernet network is modeled by a register bank which include registers of different widths (64 bits to mimic ethernet data, 3 bits to simulate ethernet active etc.) and these are controlled by a state machine so that all of them change into appropriate values at the correct clock cycles.

The FIFOs are essentially a RAM block with a state machine and internal registers control the addresses where the reads and writes take place. This is shown in Figure 2.13.

The internal registers are called head and tail. Initially they are initialized to zero. When a 'push' operation is requested, the data is written into the address

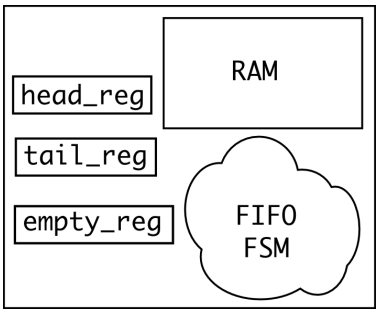


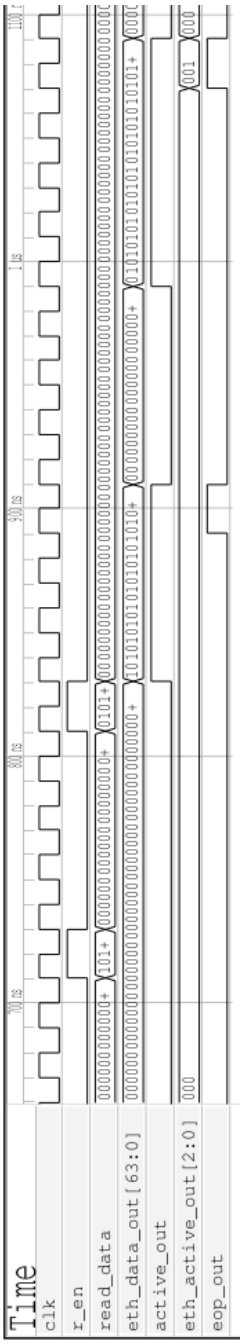
Figure 2.13: The FIFO architecture

indicated by the 'tail' register (denoting the end of the queue). This is followed by an update (increment) of the tail because now the queue has effectively grown longer. The reverse is done in case of a 'pop' operation. The address stored in the 'head' register is used to read from the RAM and then decremented to denote that the head is now pointing to the next data.

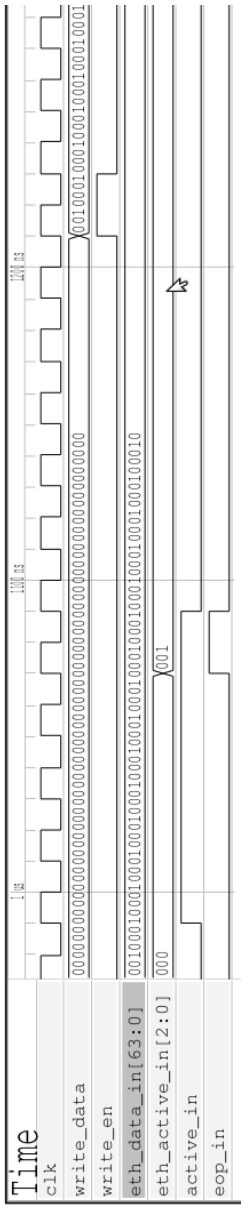
Block level unit testing was carried out on the MAC, DMA and buffer. This was followed by a top level test where the entire setup is tested. It was desirable to test both the Rx path and the Tx path. To mimic a typical scenario where the host machine has packets to transmit, at the onset of the test, packets were loaded into the main memory RAM block. Additionally, the TX descriptor FIFO was updated with TX descriptors which pointed to these packets in the system memory. With this setup in place, the NIC was activated. For the RX flow to be tested, while the NIC was active, a packet mimicking network traffic was sent to it using its ethernet interface. Additionally an empty address in the system memory was pushed to the RX descriptor FIFO where this incoming packet was supposed to be stored. The results from these simulations are shown in Figure 2.14.

The TX flow is verified from the Figure 2.14(a). At the 710 ns mark, it can be seen that the system memory is being read and a packet with the recurring pattern '10' is read from it. This is denoted as packet A. Similarly, another packet is read at 810 ns, and this packet B has a recurring pattern of '01'. At 830 ns, it an be observed that the 'active_out' signal on the Ethernet interface of the NIC goes high, indicating that a packet is now being transmitted to the Ethernet network. And simultaneously one can notice packet A being transmitted by the 'eth_data_out' port of the NIC. Similarly, packet B is seen to be transmitted at 990 ns. The difference in latency between these two packets can be attributed to the fact that packet B is longer than packet A as indicated in Figure 2.14(a).

The RX flow is verified from Figure 2.14(b). At 990 ns, a packet starts arriving at the Ethernet interface as the signal 'active_in' goes high. It has the recurring pattern '0010' and will be denoted as packet C. At 1210 ns, a write operation on the system memory can be observed, and it can be verified from the figure that the data being written is indeed packet C.



(a) TX flow test



(b) RX flow test

Figure 2.14: Behavioral simulations of the NIC

2.9 Results and Discussion

The primary objective of this part of the thesis was to build a NIC in hardware and based on the verification results shown in section 2.8, it can be confirmed that a NIC which can handle both Rx and Tx flows has been designed and implemented using Verilog HDL in this project. In retrospect, the goals listed in section 2.1 are re-evaluated in conjunction with the material presented in this chapter.

2.9.1 Goals achieved

In section 2.6, the architecture of the DMA engine is presented, which has been implemented for this part of the thesis. Block-level tests, in addition to top-level tests have been conducted and positive results from both have confirmed that the DMA engine is indeed functional and ready to be adopted for the future work on the proposed solution.

In section 2.5, the MAC has been presented in detail and from the discussion of its interface it can be inferred that it can handle Ethernet packets. The top level tests have been conducted with packets that resemble Ethernet packet sizes and so, it is imperative to conclude that this part of the project has been successful in achieving this goal.

During the course of this chapter, section 2.4.2 has introduced the need for the shared memory, and when discussing its implementation, additional components such as the linked list and FIFOs have been investigated in 2.4.3. The designer did not have previous experience with building these data structures on hardware, but after this part of the project I have gained experience in implementing these as exemplified in section 2.8.

In this step of the project I have gained familiarity with the ethernet standard which has been discussed in section 2.2.1 and then used in practice during the testing in section 2.8. Furthermore, by designing the entire NIC with all its components I have now attained a working knowledge of networking hardware, and most importantly, encountered the particular issues with the traditional setup (refer section 1.1) during the design process. These issues and the mechanism used in the current design to tackle them are discussed below.

From the Figure 1.2, it is understood that dropping packets is not desirable and it was hypothesized in section 1.1 that the limited buffer memory size is responsible for this situation to arise. In the presentation of the architecture, one needs to recall section 2.4.1 where it became necessary to introduce a 'Buffer Full' signal to indicate that the internal buffer is full when the address generator runs out of free memory locations. By designing the architecture and implementing it, this thesis has been able to point out the exact location of the bottleneck in the NIC hardware. This justifies the proposed technical solution shown in Figure 1.5 and gives the project an idea about the next steps that need to be taken. Though it is not possible to solve this problem while following the traditional NIC approach,

in order to have a fool-proof design, this designer has opted to add the above mentioned signal so as to prevent (high load) data traffic from collapsing the implemented NIC. It can be argued that it is more of a preventive measure which limits the feature set of the NIC, but the purpose of this part of the project is not to come up with a solution at this stage, but to build a working NIC in hardware.

2.9.2 Hardware implementation results

Though testing on hardware has not been pursued for this part of the project, since the NIC is implemented in RTL logic in Verilog, it is synthesized using the Xilinx ISE tool. As such, to get an insight into how the tool has invoked the code, some statistics are reported below.

Table 2.1: Macro statistics from Synthesis log

Hardware unit	**Number of blocks**
FSM	7
8x32 bit ROM	1
Adders and Subtractors	20
Flip-Flops	14082
Comparators	14
Multiplexers	1236

The NIC is a fairly large design, as will be clear in the device utilization summary below in Table 2.2, but from Table 2.1, a few fields are now examined in detail. There are no RAM modules invoked, which means that the chunk memory and the size memory, as well as the internal RAMs used in the FIFO blocks and the LIFO block have all been invoked as register banks. This can explain why the number of flip-flops seems to be disproportionate compared to the other fields in the table. Another point of discussion is the number of multiplexers. The RTL code in the appendix of this report can account for this large number as well. It is due to how the SIPO and PISO registers in the MAC and DMA blocks have been designed. Of course, more efficient implementation of the blocks in the NIC, combined with restraints on the synthesis tool can yield different statistics, but that was not a goal of this project.

The device utilization summary is presented below for the Virtex 5 FPGA as part of the ML507 development board.

In Table 2.2, the number of slices and their utilization gives the reader an idea of the footprint of the design. Since the project uses a FPGA flow, area statistics are not available because the dimensions of the FPGA is fixed, and a proper evaluation of the design's size can be made from the percentage of utilization. A closer look at Table 2.2 yields a disproportionate value in the number of bonded IOBs (Input/Output Buffer). This is not unusual if the reader recalls from Figure 2.3, Figure 2.9 and Figure 2.8, that the external interfaces of the NIC contain a

Table 2.2: Device Utilization Summary

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	14091	44800	31%
Number of Slice LUTs	6558	44800	14%
Number of fully used LUT-FF pairs	3818	16831	22%
Number of bonded IOBs	588	640	92%
Number of BUFG/BUFGCTRLs	2	32	6%

large number of signals which interface it to the host memory and the Ethernet network. As such, this high number of IOBs is indicative of the architecture chosen, and in case one prefers to decrease this number, the external interface can be simplified, or a serial interface may be preferred over the (present) parallel bus based interface. Opting for this alternative, however is likely to introduce a constraint on the speed of the design which is explored below.

Table 2.3 shows the timing information for the NIC, as reported by the Xilinx synthesis and timing tool. It is to be kept in mind that in both cases, the speed grade was set to -1.

Table 2.3: Timing Summary

FPGA	Minimum Period	Maximum Frequency
Virtex 5	3.543ns	282.247MHz
Virtex 7	2.659ns	376.081MHz

Section 2.4.2 presents a calculation of the minimum frequency with which the NIC design must operate in order to fulfill the ethernet specification discussed in section 2.2. Table 2.3 indicates that the actual maximum frequency achieved is higher than this requirement, so that is a positive result. However, the designer was surprised to see the results for the Virtex 5 FPGA, because though Table 2.2 indicates a fairly large area utilization, a minimum period of 3.5 ns is indicative of a very fast design. Consequently, the timing reports were generated for successive runs, and since the results were consistent with what is shown in the first row of Table 2.3, the designer opted to synthesize the design for the Virtex 7 platform as well (row 2 of Table 2.3) and look for discrepancies. It seems that these results depend on the size of the FPGA, and the resources used in the FPGA. As seen in Table 2.3, the larger FPGA (virtex 7) allows an even faster frequency of operation. Further investigation of the timing report for the Virtex 5 platform yield an interesting feature shown in Table 2.4.

Since the NIC is the sole block which is being implemented, the Xilinx tools are able to optimize it for speed, and based on what is shown in Table 2.4, the placement and routing of the components might be responsible for these statistics. In fact, up to 49.7% of the minimum period can be attributed to this factor alone, and since Table 2.3 shows that a larger FPGA gives a faster implemented design, this

Table 2.4: Timing Breakdown

	Time	**Percentage**
Logic	1.782ns	50.3%
Route	1.761ns	49.7%

argument is valid. Additionally, the high number of flip-flops reported in Table 2.1 can also be attributed to these results because from the knowledge gathered in my courses at LTH, an efficient RTL implementation with separate data path and control path, complemented with extensive pipelining in the design should yield a smaller maximum path for delay.

To conclude, in this section of the report, the results from this part of the project has been presented and evaluated both, in terms of the goals and specifications required, and on the basis of results generated by the Xilinx FPGA flow.

Chapter 3

PCIe interfacing with host

3.1 The PCIe interface

In this section of the the project the PCIe standard [16] is investigated in order to achieve the goals in this chapter. PCIe is an industry standard which is used to connect devices such as graphic cards, NICs, solid state storage etc. to the motherboard of the computer. It improves the previous generation of PCI [17] which is a traditional bus based architecture, and introduces a network based architecture, whereby it is implemented as a point-to-point link between devices. PCIe is a network of devices, and communication between them occurs in the form of PCIe packets. This difference is shown below in Figure 3.1. Since the topology of PCIe is more of a serial network rather than a parallel bus (like PCI) there are no performance hindrances such as bus arbitration, operating speed bottlenecks (due to low maximum frequency achievable) etc. associated with the previous generation [16]. A more detailed discussion on the advantages of PCIe over PCI can be found in [18].

Since the proposed technical solution is targeted towards DCNs as seen in section 1.2, its worthy to mention that previous research [19] has evaluated the benefits of using PCIe in such an environment. For the purpose of the project, the theory of the PCIe standard and its implementation has been studied in detail. Since this is an industry standard, there is limited scope for the designer to contribute to the theoretical details and instead of repeating this information which is well documented in [16], [18] and [20], this report will focus mainly on the different types of packets and the protocol followed to establish transfer of data between devices.

3.2 Goals and Overview

After studying the PCIe standard in detail, the designer was presented with a set of objectives that were required to be fulfilled at the end of the project. These

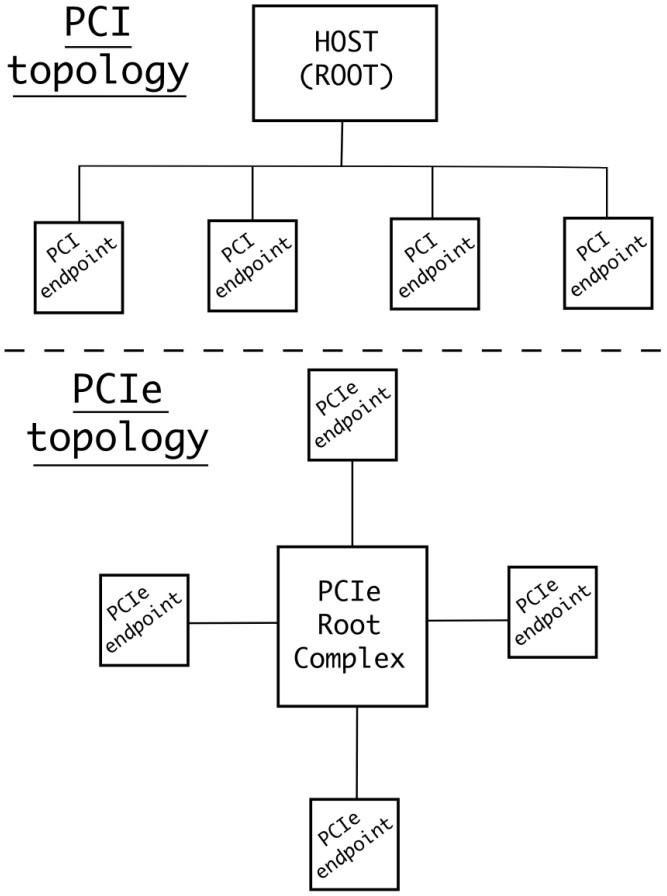


Figure 3.1: Difference between the PCI and the PCIe

goals are presented below.

- Demonstrate that the PCIe standard can be utilized to interact between the host machine and the proposed solution.
- Design a PCIe IP that can be re-used to interface future products to a Linux based system.
- Ensure that the IP is able to handle *DMA* and *Interrupts*.
- Deliver a complete hardware and software module which can serve as the starting point for the proposed solution.
- Verify above goals by testing on a physical hardware platform.

With these goals in mind, the designer now presents an overview of the outcome from this part of the project. The requirements mandate two elements which are, a

hardware device which can connect to the host as a PCIe endpoint and a software driver which can be used to control this device and its interaction with the host machine. The hardware used is discussed here. Since the proposed solution is a proprietary device with specific requirements, the designer did not choose to use an off the shelf PCIe IP such as [21] because it would take more time to interface it with the company specific interface, especially since the IP itself was closed source and did not allow investigation into its internal structure. Instead, this designer adopted a more open ended solution, which is the Xilinx PCIe Block Plus v14 IP [22]. This is a very generic implementation of the standard and the designer built the hardware application on top of the existing PCIe specific PHY and link layer. As a hardware platform to test, evaluate and demonstrate the developed IP, a Xilinx ML507 development board [23] is chosen which has a Virtex 5 FPGA and has support for a physical PCIe 2.0 1x endpoint. The developed IP is loaded on the FPGA and the board is inserted into the motherboard of the host machine.

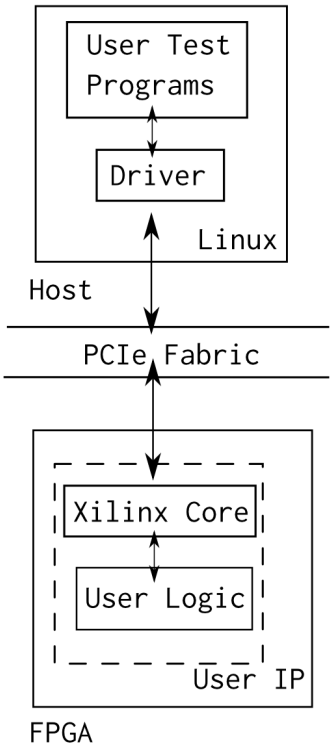


Figure 3.2: PCIe project overview

The developed solution, and its position in the bigger picture as described below can be seen in Figure 3.2. On the software side, another employee at the company helped to design the different versions of the driver needed to control and interact with the hardware from a terminal. All the drivers developed during the course of this project are based on the Linux 2.6 kernel but they are easily ported to include support for the newer 3.x kernels as well. This approach of having a hardware and

software based solution to create a customized PCIe IP on top of the Xilinx PCIe core, has been followed in academia [24], industry [25] and a master thesis [26]. Hence, even though the solution developed at the end of the project is my original contribution, I opted to use a similar approach as them because as a designer I evaluated that it was well suited to satisfy the goals presented above.

3.3 PCIe : Transactions and Packets

Any transfer of data in the PCIe protocol is called a transaction. There are two kinds of transactions, posted and non-posted. For non-posted transactions the device (for example, the host) which initiates the transaction (from now on, called the 'requester') transmits a request packet to another device (for example, the board). As a response the latter device (called the 'completer') returns a completion packet to the requester. In contrast, for posted transactions, the completer does not generate a completion packet in response to the requester's request packet. It seems that posted transactions are geared towards performance and speed because the extra time to send the completion packet is non-existent. For this project, it is worthwhile to know that memory writes are posted transactions while memory reads are non-posted transactions.

The naming convention followed in the PCIe specification is straightforward. Instead of listing all the different types of PCIe packets, the reader is provided Table 3.1 which includes the subset of PCIe packets which were encountered during the project.

Table 3.1: PCIe packets and abbreviations

Packet type	**Abbreviation**	**Purpose**
Memory Read Request	MRd	Initiates a read
Memory Write Request	MWr	Initiates a write
Message without data	Cpl	Interrupts
Completion With Data	CplD	Response to a read

3.3.1 Write to board

The first step is to set up a register on the FPGA and write to this register from the host. As shown in Figure 3.3 this requires a posted MWr transaction.

For debug purposes, the value written to the register is used to light up the user accessible LEDs on the board. Three design initiatives are requisite to achieve this step.

- Writing a driver and a user space program to run on the host, which initiates the Write request.

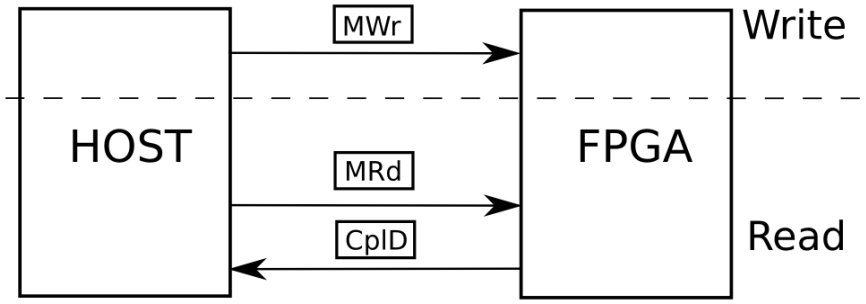


Figure 3.3: PCIe read and write transactions

- Writing a user defined hardware register on the board and interfacing that with the Xilinx PCIe core.
- Decoding the PCIe MWr packet on the host, extracting the payload and using it to drive the LEDs.

For the first item, the driver which Xilinx has provided for the IP was explored. It was found that it was incompatible with the newer kernel and furthermore, it was only designed to calculate the throughput of the PCIe standard. It did not allow user space applications to interact with it, so a driver was written from scratch. It is beneficial to understand how PCIe devices are enumerated at start up to design the driver.

- At boot time, the PCIe root on the motherboard searches the available slots for devices.
- When it finds one, it is able to access the PCIe endpoint's (device's) configuration space.
- This configuration space, besides other information, has a register which lets the PCIe root know how much memory is on board.
- The PCIe root relays this information to the operating system, which allocates the same (or less, depending on size) amount of memory in the host's system memory which the driver for the device can access.

Once the host boots up, I manually insert the driver in the Linux kernel and launch the user space application which is a shell script, which on execution, prompts the user to enter a number using the keyboard.

For the hardware design on the FPGA, I looked into how the Xilinx PCIe IP was implemented. After identifying the requisite signals which are carrying the information of interest, I proceeded to extract it from the protocol. For this purpose, I also looked into the actual structure of a PCIe MWr packet, which gives us the information about which bits need to be extracted to obtain the payload. This is shown in Figure 3.4.

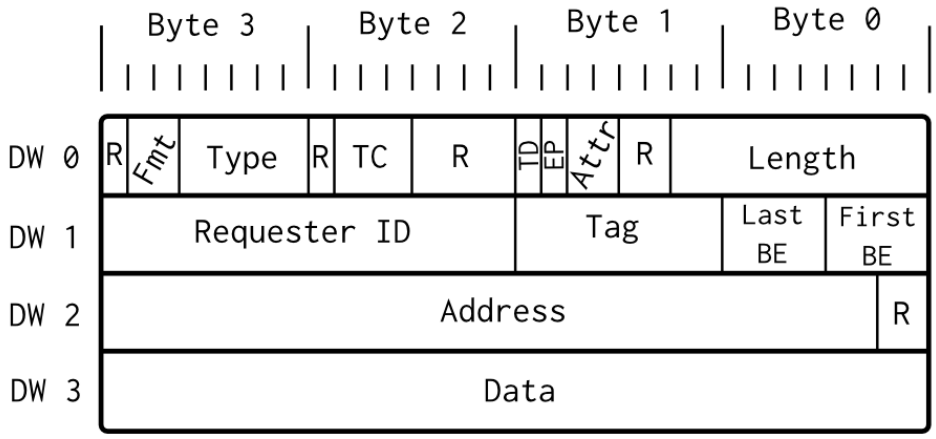


Figure 3.4: PCIe MWr packet

The critical issue about the payload is that by default it is 32 bits long, whereas the register implemented on the hardware, as well as the number of LEDs available, is 8. So, to look into the exact byte that contains the information, I set up the Xilinx Chipscope tool, which enabled me to look into the actual hardware signals, like a virtual logic analyzer. A detailed discussion on this is presented later in section 3.4.

3.3.2 Read from board

To read from the board, I interface a new register to the PCIe IP, and toggle the user accessible switches available on the board to actively input a byte of data from the board into this register.

PCIe MRd is a non-posted transaction, and the driver for this phase initiates a read request. The hardware logic on the board responds with a completion to this read request. This is shown in Figure 3.3.

Now, the Xilinx core does not provide the means to respond with the requisite data when getting such a request. Furthermore, the completion packet needs to have information appended to it, regarding the destination of the packet (in this case, the host).

I needed to implement this completion mechanism on my own, and to do that I looked into the read request from the board (Figure 3.5), in order to extract information about the source of that packet (Requester ID in Figure 3.5), to which the developed IP will send its response. Furthermore, the structure of a PCIe CplD (Figure 3.6) is also investigated so as to ensure that the completion is actually accepted by the host.

Once these packets were assimilated, I identified the signals which allowed me to

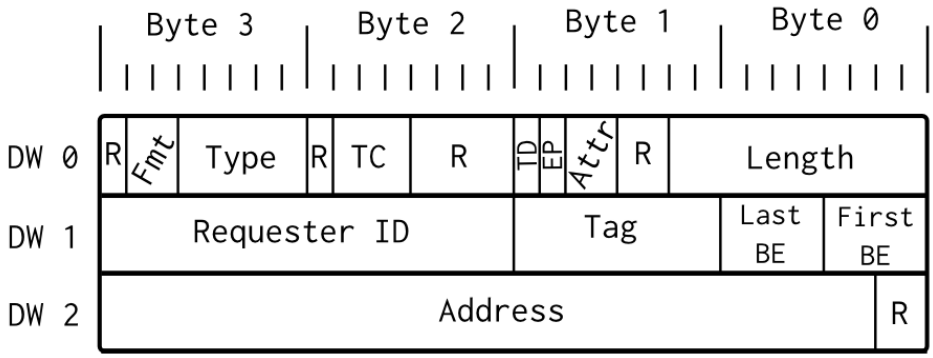


Figure 3.5: PCIe MRd packet

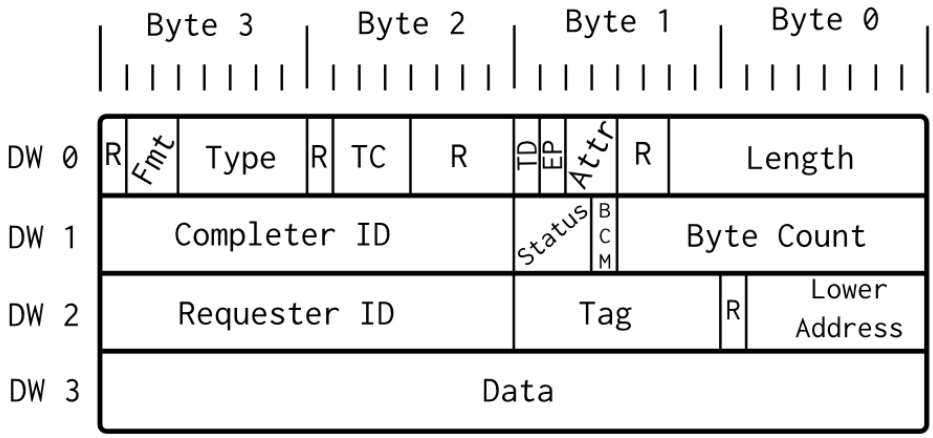


Figure 3.6: PCIe Completion with data

receive and send the corresponding request and completion. The next step was to write a custom hardware module and interface that with the IP to achieve the following steps sequentially.

- Deconstruct the MRd request
- Extract the information about the Requester Id and Packet Tag
- Assemble the CplD response
- And finally, transmit the response to the requester (host machine)

This also required setting correct values to other auxiliary signals which carried information about which byte was valid in the entire payload so that the host can identify the same.

On the software side, a user space application was employed to generate a read request and receive the response. The user space application was another shell

script which generated the request at the press of the return key on the host machine's keyboard.

3.3.3 Interrupt generation

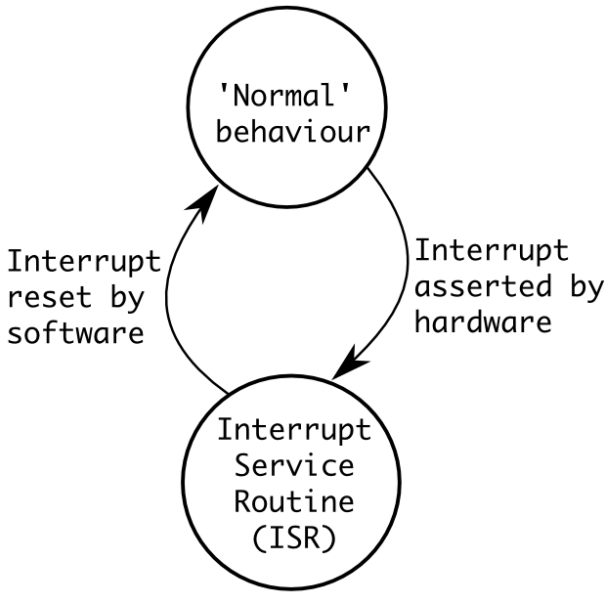


Figure 3.7: How an interrupt works

The concept of interrupting a process can be broken down to three basic steps as shown in Figure 3.7.

- The hardware module that wants to interrupt the software process asserts the interrupt in response to some event. From the hardware point of view this assertion can be done in more than one way, which we will see later.
- Once the interrupt is received by the software, it halts the normal operation of the process, and starts executing an Interrupt Service Routine (ISR) which is unique to that interrupt. In most cases, the ISR is a sequence of predetermined instructions.
- After the ISR has been completed, the state of the interrupt is reset and the process resumes its normal operation.

Before PCIe was introduced as an industry standard, PCI and previous generations of interconnect used the 'legacy' mode of generating an interrupt. This is shown in Figure 3.8.

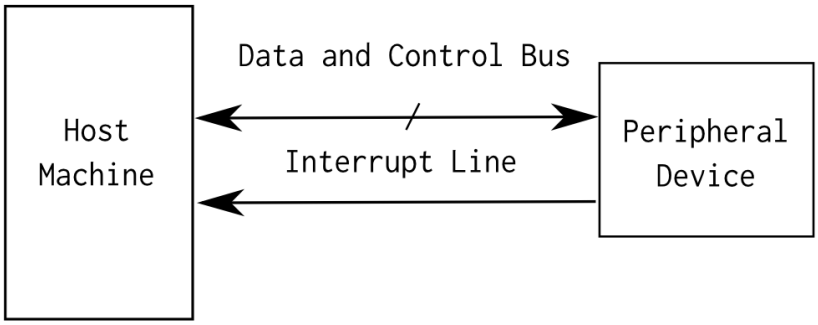


Figure 3.8: Legacy interrupt line architecture

In this generation, it was necessary to have a separate 'interrupt' line which would be set high in order to assert an interrupt. It was level sensitive, so the interrupt was asserted as long as this signal was driven high by the peripheral device.

This arrangement is not necessary for devices which natively support PCIe (version 2.2 and later). This is because PCIe by default uses packets to generate interrupts. Interrupts are propagated to the host using the same physical medium that carries all data and control information in the PCIe interconnect standard. The implementation of message signaled interrupts (MSI) in PCIe devices is achieved by a three fold process.

- An interrupt register in the device's PCIe configuration space is initialized. This contains relevant information like how many interrupts the device is requesting to take control of.
- At system boot, the host operating system will read the configuration space of the external devices and accordingly assign the available interrupts among them.
- During active operation, the external device needs to write a PCIe interrupt message and send it to the host to activate the previously assigned interrupt.

In this project, I enabled MSI functionality and properly configured the Xilinx IP using the Coregen tool of the Xilinx ISE software. To actually generate the interrupt, I assembled a MWr packet and send it from the board to the host machine.

The MSI capability register is the aforementioned *interrupt* register in the configuration space of the PCIe device. The entire MWr packet for the MSI message consists of two parts, the header and the data. I had to access the *read-only* configuration space manually by going through the IP code to copy the information from the appropriate register into our MSI packet and send it to the host.

So far the actual process of generating an interrupt has been discussed. But one also has to consider a real world case where interrupts are actually required. Consider the particular example of a first-in first-out (FIFO) data structure interfaced

with the PCIe IP, where the host writes to and reads from, this FIFO on the FPGA using the PCIe standard. A straight-forward implementation is that the host should always read from the FIFO. But this is not possible when the FIFO is empty. So, to indicate to the host when the 'read-from-FIFO' process can start, it was decided that the board will send an interrupt to the host as soon as a particular event occurs, that is, the FIFO changes from 'empty' state to 'non-empty' state. In other words, this event occurs when at-least one element is in the FIFO. The hardware developed on the board is shown in Figure 3.9.

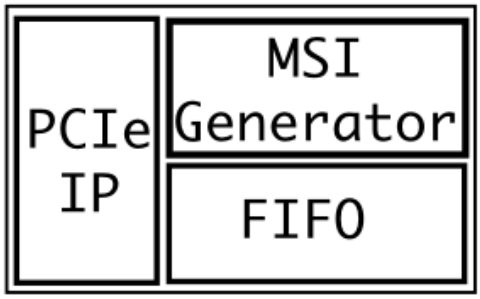


Figure 3.9: Hardware on FPGA for interrupts

On the software side, the existing driver is augmented with the handling of interrupts. The existing write and read test programs from the previous versions of the driver is reused. The write program is used to push data into the FIFO and the read program is used to pop the FIFO and retrieve the data. Additionally, the read program is linked to the interrupt handling mechanism as well. At launch, the read program remains suspended because by default the FIFO is empty. Then as the write program is used to populate the FIFO, an interrupt is generated from the hardware which activates the read program. It remains active till all the contents of the FIFO is popped, after which the read program again goes to the suspend state and waits for the next interrupt.

3.3.4 Direct Memory Access

Considering the functionality of the proposed solution, the most important feature is to transfer packets to and from the system memory without active intervention of the CPU. This is called direct memory access (DMA). When reviewing literature on this issue, I found out that this DMA functionality using the PCIe is the motivation for most research on the subject. Authors in [27] describe a scatter-gather DMA engine using the PCIe express, whereas [28] outlines a DMA engine for the Windows platform and a Dell Optiplex machine. Others such as [29] build and use the PCIe DMA function to achieve active buffer data transmission.

To build and test the DMA functionality, I decided to pursue this sequence of PCIe traffic

- Write a packet from the host to the FPGA.
- Use the payload from the previous action to write another packet from the FPGA to the host.
- Issue a read request from the host to the FPGA.
- Since the requested data is not on the FPGA, issue a read request from the FPGA to the host.
- Wait for the host's completion to the FPGA's read request.
- Extract the payload from the host's response and use the same for the completion from board to host.

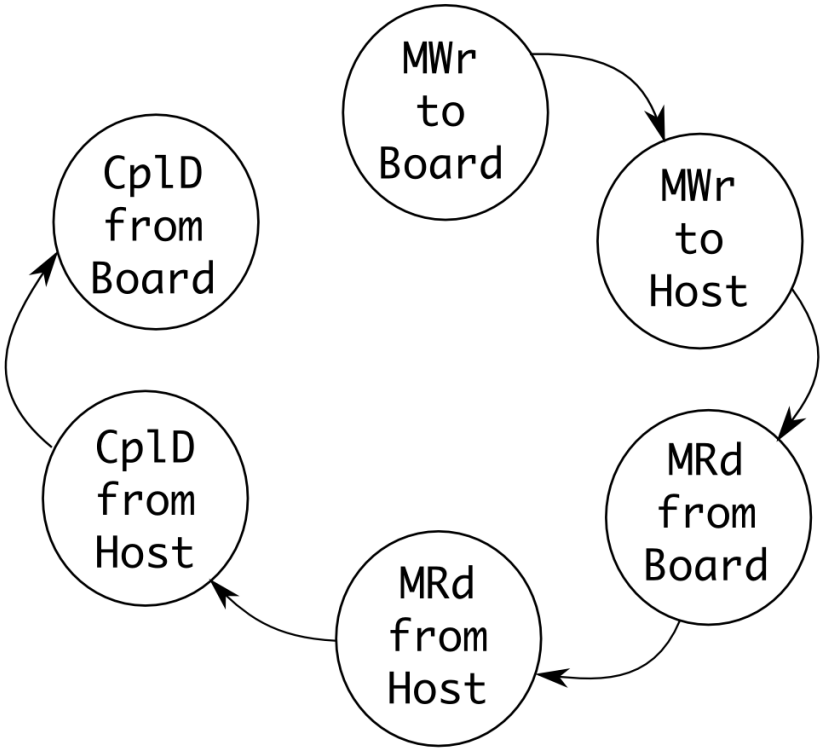


Figure 3.10: PCIe Packets for DMA

It is imperative to point out that in this stage of the project, the physical location of the data being written and read - is now stored only in the system memory, and not on the physical registers in the FPGA. The actual packets which need to be transferred for this DMA engine are shown in Figure 3.10.

Adding DMA functionality to the hardware IP does not imply that the host will recognize this capacity on its own. To solve this problem, the changes in the software driver are presented below. To support DMA, the board must be informed

which memory location (in the system memory) it has access to. It will write and read from this memory location. It is the driver's task to allocate this space in the system memory for the device so it is not overwritten (for example by other applications or devices). Furthermore, its the driver's task to let the board know of this location.

The driver uses the dma_alloc_coherent function available in Linux to allocate a 64 bit address pointing to a byte of space in the system memory. When the external device is enumerated, a MWr packet is sent to it, independent of the user test programs, and this contains the address of the space allocated in system memory where the device can write to and read from.

It is time to look deeper in the hardware IP module to understand the changes and additions implemented at this stage. The different packets that need to be handled by the user module are listed below.

- A read packet MRd from the host.
- A write packet from the host informing about the address.
- A memory write packet from the host.
- A completion packet from the host.

So, immediately it can be seen that there is a need to distinguish between two types of write packets (address write and memory write) from the host. They have the same source (host) and destination (board) and they have the same header type (MWr) because they are both write packets. So the hardware and the software agreed to designate a byte enable field (Figure 3.11) which was used to distinguish between the two write packets.

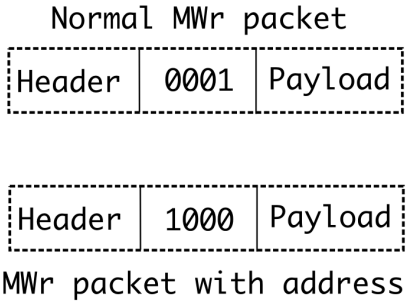


Figure 3.11: Showing the byte enable field

In addition to this, there is a need to handle a new type of packet which has not been encountered before, the completion from the host. The information gathered from section 3.3.3 is reused. After generating the read request to the host, there is a small latency before the hardware receives the response from the host. This latency can be attributed to the PCIe connection between the board and the host. After receiving the completion from the host, the hardware IP decodes the packet,

extracts the payload and then reconstructs another completion packet which the device (FPGA) must send to the host again.

The testing strategy used is also modified. First, the write test is invoked. The program returns the value written to the board as well as the value which is written by the board to the register in system memory. Then, as an extra hidden step, the driver increments the value in that register by one. Next the read test is executed. This returns both the value in the system memory which is read (directly by the driver) and the value actually returned by the board. If at any point, these two pairs are not coherent, the test fails. This high-level software based testing strategy which utilizes parts of all the tests carried out before is shown in Figures 3.12 and 3.13.

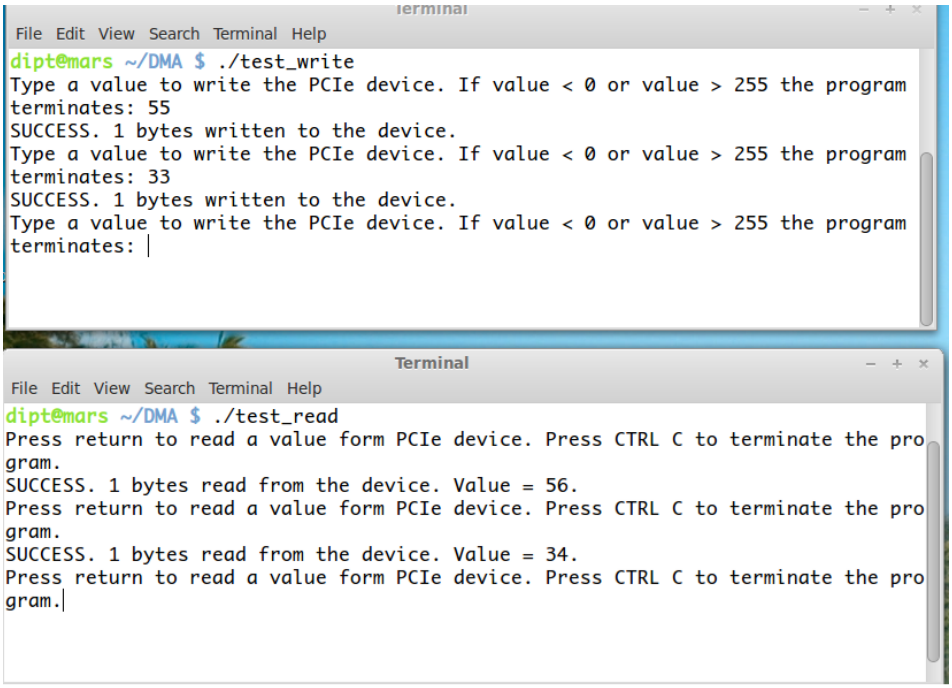


Figure 3.12: Read and Write Test programs

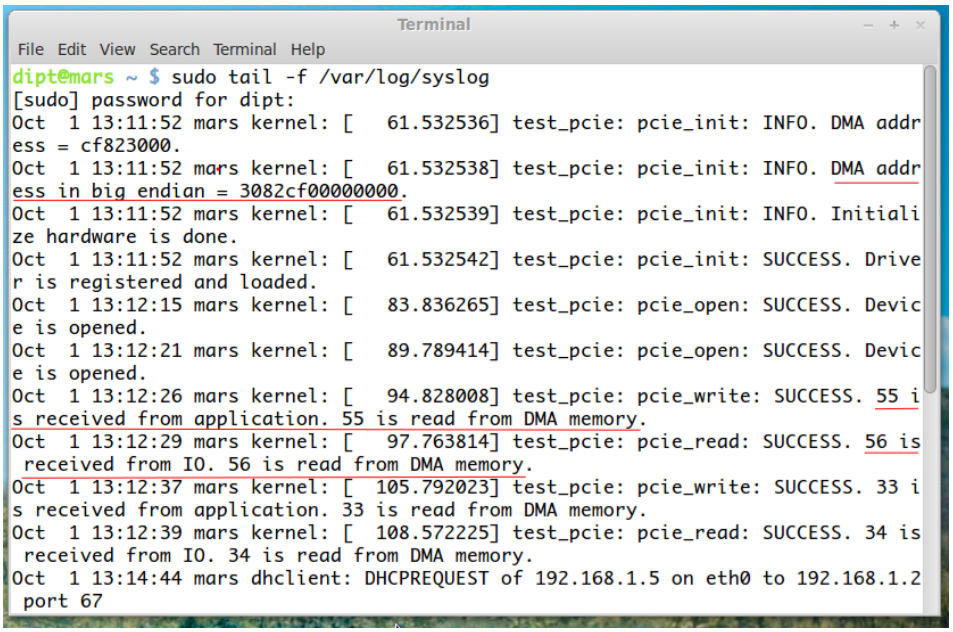


Figure 3.13: The system log displaying messages from the console

3.4 Debug strategy

Once the design is synthesized and routed on the ML507 evaluation platform, high level tests are performed using user written software programs as indicated in Figure 3.2. However, when these tests fail, it is necessary to look into the FPGA design. The ideal approach for this debugging is to use a PCIe logic analyzer and interface that with the board. However these instruments are expensive and are not available to the designer at this phase of the project. Xilinx provides an alternative solution known as Chipscope, which enables debugging at chip-level albeit at the cost of additional area on the FPGA. The idea is to insert additional modules into your existing design, and these modules monitor and capture snapshots of all desired signals (in your existing design) which can be viewed while the FPGA is on-line. This communication is achieved through the JTAG cable provided with the board.

Post synthesis, an Integrated Controller Pro (ICON) core and an Integrated Logic Analyzer Pro (ILA) core are inserted into the design. Then, these cores are configured to monitor the signals for reading and writing to the FPGA's PCIe interface. This is done to manually inspect the actual packets which are being sent or received, to verify that the developed user application (behind the hard IP) is assembling and decoding packets correctly. There is a need to choose a hardware trigger, that is, a signal which is monitored continuously by these cores. When a change on this trigger is detected, the ILA stores a snapshot of all signals and

reports it to the user. This choice of trigger is motivated by which packets (read or write) needs to be inspected. Since there is a typical 'end of packet' signal in the PCIe interface, this signal is used as the trigger signal.

To give an insight into this strategy an the DMA read test from section 3.3.5 is considered as an example.

Signals *trn_reof_n* and *trn_teof_n* are of interest in Figure 3.14 because they indicate the end of packet (and effectively end of transmission) of packets received and packets sent respectively. As indicated by the marker T, the latter is used as a trigger in this example. The ILA is configured to capture a large number of samples after the trigger is activated, and as a result the actual packets that are used for accomplishing the DMA read can be seen. At sample number -3, it is observed that a packet is received by the FPGA. This is the host machine requesting a read. At sample 0, a packet being transmitted by the FPGA can be identified. This is the FPGA issuing a read request to the host's system memory. A latency is observed on the PCIe interface because there are no packets until sample 101. At this point, the completion is received from the host and it carries the data requested by the user application running on the FPGA. Finally at sample 105, the FPGA transmits a completion back to the host. Since this observation is in sync with the acquired knowledge of the PCIe standard, and because the test programs report success as shown in Figure 3.13, it is confirmed that the DMA task is completed.

3.5 Results and Discussion

The primary objective of this section of the project is to build a PCIe IP which is able to handle certain functions and can be used to interface the proposed solution to a live computer. As indicated in the above tests, this has been achieved by the designer. It is imperative to look back at the goals listed in section 3.2 once again at this point in the report.

3.5.1 Goals met

When the project was initialized, the objective of this step was to investigate and learn the PCIe standard. The information available at that juncture was that PCIe is a commonly used industry standard with high throughput that can be used to implement and support the functionality and unique features of the proposed solution which introduced a large overhead in the system and could not be implemented without PCIe support. In section 3.3 it has been shown that the PCIe can actually be used to interact with the host machine and an external FPGA board. Additionally, since the hardware utilization of the developed PCIe IP as indicated in Table 3.2 and Table 3.3 indicates that a majority of the available resources on the FPGA are free, it can be assumed that it is possible to augment the FPGA with a moderately sized standalone hardware project as well. Of course,

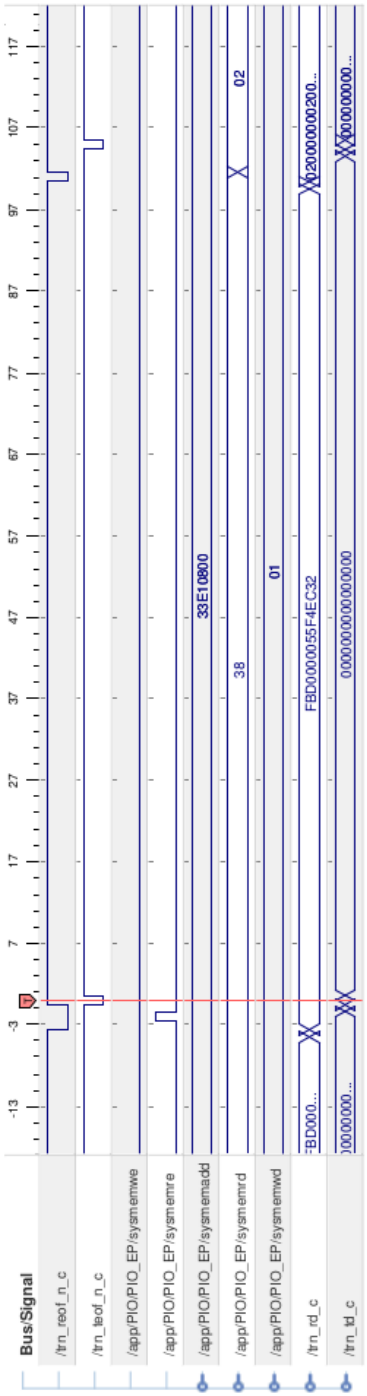


Figure 3.14: Verification using Chipscope

this will require an additional step where this additional project will be register mapped to the internal registers discussed in section 3.3.2 and section 3.3.3. So, the first two goals listed in section 3.2 are fulfilled.

In section 3.3.4 the interrupt handling process has been elucidated and a MSI based interrupt generation has been presented. This developed functionality has been reused during the DMA step, as discussed in section 3.3.5 and it has been verified on hardware that interrupts can properly be generated and handled by the developed IP. In section 3.3.5, the DMA functionality has been explained and the results from hardware testing, using a high level software verification platform, has been presented in Figure 3.12 and Figure 3.13. To supplement these observations, section 3.3 has explained the debug functionality on live hardware, and from Figure 3.14 the DMA functionality has been verified. At the end of this project, the designer has delivered a PCIe IP which has been verified on hardware, and a software driver which interfaces this module to a Linux kernel. So, it can be stated that the rest of the goals of the project, from section 3.2, have also been met.

3.5.2 Hardware implementation results

The device utilization summary for the PCIe design on the ML507 development platform is presented below. Table 3.2 provides the results achieved from synthesis, and Table 3.3 shows the resource utilization after place and route.

Table 3.2: Device Utilization Summary : Synthesis

Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	3305	44800	7%
Number of Slice LUTs	2883	44800	6%
Number of fully used LUT-FF pairs	4521	16831	26.8%
Number of bonded IOBs	8	640	1%
Number of BUFG/BUFGCTRLs	6	32	18%
Number of Block RAM/FIFO	10	148	6%
Number of GTX_DUALs	1	8	12%
Number of PLL_ADVs	1	6	16%

It is interesting to compare Table 3.2 with Table 2.2 because they are both for the same board, and they are both generated by the Xilinx ISE synthesis flow. The immediate observation is the change in the number of logic slices (in rows 1 and 2), which has decreased considerably in the case of the PCIe IP as compared to the NIC. One argument for this is that the NIC might be a larger design than the PCIe IP and the developed user application behind it. But, one must not overlook the additional Block RAM/FIFO row in Table 3.2.

To recall the discussion in section 2.9.2, it was specifically pointed out that the RAMs and FIFOs in the NIC has been synthesized as register banks by the ISE tool. In the PCIe IP case, this does not happen. The specific reason as to why this has happened, can be attributed to the fact that in the PCIe IP project, the RTL for the RAM block has been obtained from a Xilinx example design. So, their synthesis tool has performed better at translating their code into appropriate hardware blocks.

The number of bonded IOBs has decreased significantly as compared to the NIC design. And the reason behind this, is that at the top level, PCIe can be interfaced using a pair of coaxial signals and few additional signals (reset, clock etc). The reader can also notice two new additions in the Table 3.2. The GTX_DUALS are Xilinx's implementation of the SERDES whereas the PLL_ADVs is used to generate multiple clocks that are required for the PCIe IP to function.

Table 3.3: Device Utilization Summary : Place and Route

Logic Utilization	**Used**	**Available**	**Utilization**
Number of BSCANs	1	4	25%
Number of BUFDSs	1	8	12%
Number of BUFGs	6	32	18%
Number of GTX_DUALs	1	8	12%
Number of External IOBs	9	640	1%
Number of External IPADs	4	690	1%
Number of External OPADs	2	32	6%
Number of PCIEs	1	3	33%
Number of PLL_ADVs	1	6	16%
Number of RAMB36SDP_EXPs	2	148	1%
Number of RAMB36_EXPs	29	148	19%
Number of Slices	2067	11200	18%
Number of Slice Registers	4165	44800	9%
Number of Slice LUTS	3250	44800	7%
Number of Slice LUT-Flip Flop pairs	5273	44800	11%

Table 3.3 gives an idea of the post route and placement hardware utilization. Most entries reported in this table are specific to the Xilinx FPGA tool chain, and for further information on each component, the reader is advised to refer to the product documentation of the development platform [23]. To get a clearer interpretation of Table 3.3, certain examples are highlighted. BSCAN is the module used to interface the hardware with the JTAG chain. Since the Chipscope tool is used in this project, the JTAG interface has been used in the implementation, so that the designer can connect the board to the computer and capture signals while the hardware is online. This is useful for the debugging of the hardware, and has been discussed in section 3.3. Again, since the developed PCIe IP is built

upon the Xilinx PCIe core as a starting point, this core shows up in the Table 3.3 as *PCIE*.

The timing information obtained is presented in Table 3.4.

Table 3.4: Timing Summary

Minimum Period	Maximum Frequency
15.744ns	63.516MHz

Comparing Table 3.4 to Table 2.3, the difference in the operating frequency is noticeable. However, it is important to point out that when using specific IPs with preconfigured components, one is limited to the operating frequency of the components used. In this project, the hardware development platform has a PCIe 2.0 1x as indicated in section 3.2 and when the Xilinx PCIe core is configured using the Coregen tool, the operating frequency cannot be changed from the value indicated above.

Finally, the power consumption by the implemented design is reported below. This information is obtained by using the Xpower analyzer tool.

Table 3.5: Dynamic Power Consumption

On-Chip component	Power (mW)
Clocks	84.19
Logic	5.71
Signals	21.88
IOs	16.69
BRAMs	26.09
GTP_DUALS	171.78
PLLs	81.07
PCIEs	305.62
Total Dynamic Power	713.03

There are two key points that one must keep in mind when interpreting the information in Table 3.5. Firstly, Xpower Analyzer is more of an estimation tool rather than a measurement tool. In the case of this project, I have chosen to enable the default toggle rate for the flip-flops (12.5%) and the BRAM write rate is set to default as well (50%). This information is pertinent, because Table 3.5 shows the dynamic power consumption, or rather, an estimation of the dynamic power consumption using this toggle rate and write rate. The actual dynamic power consumption can vary from the values shown above, depending on the load on the IP.

Secondly, since the project has been implemented in a FPGA flow, the value of the power consumed cannot be compared to ASIC solutions. One of the reasons

for this can be directly observed in Table 3.5. The reader's attention is drawn to the fact that there are several components in the table above, which may be considered as auxiliary additions that are necessary for the PCIe IP to work on this FPGA. For example, the PCIe core itself is using 305 mW of power, but the total dynamic power is more than two times that value. Since, the designer is constrained by the design of the hardware platform, removing these supporting blocks is not feasible, because then the designed IP will stop working.

Table 3.6: Total Power Consumption

Power Type	**Value (mW)**
Static Power	1633.66
Dynamic Power	713.03
Total Power	2346.69

Table 3.6 sheds light on the total power consumption, and the issue of interest is the wide gap between static and dynamic power consumption seen here. The reason FPGAs consume a large amount of static power compared to ASIC implementations is that FPGAs use up to 10x the number of transistors compared to an ASIC [30], and the static power is a manifestation of the power lost due to leakage current on all of these transistors. This issue cannot be resolved by an end-user of the FPGA, such as the designer in this project due to the nature of how FPGAs are constructed as discussed in detail in [31].

I have previously come across this information during lectures in Digital IC design and Introduction to Structured VLSI at LTH, and have also come across literature like [32] which specifically explores these issues in the current context. However, by completing this project, I have experienced it myself. So, this discussion is concluded by stating that even though FPGAs allow faster prototyping on hardware, through this project, I have gained a first-hand idea of the power consumption and area consumption challenges as compared to an ASIC solution.

Chapter 4

Investigation of QCN implementation

4.1 Introduction and goals

The goal of QCN is to give switches the capability to notify end hosts of any congestion in the network so that the hosts can respond by decreasing the transmission of packets and therefore alleviate the congestion [10]. QCN is an acronym for Quantized Congestion Notification and is part of the IEEE802.1Qau Ethernet Congestion Notification standard. The algorithm is based on two types of nodes in a network. The Congestion Point (CP) is the node where there is a scope for congestion to occur and is typically a switch. The Reaction Point (RP) is the node which can react to a congestion by controlling the flow of packets, and is typically the source of the packets like a host machine. This part of the project is an investigation of the resources required for supporting this algorithm. The goals of the project are listed below.

- Conduct an analysis of the hardware cost for supporting QCN.
- Discuss and finalize (high-level) descriptor handling architecture for the proposed solution
- Suggest an optimum size of a on-chip memory for the proposed solution.
- Investigate if the PCIe standard can handle the overhead and latency requirements for the proposed solution.
- Estimate the number of client machines that the proposed solution can support.

For this chapter, the term *Hydra* will be used to refer to the proposed technical solution in section 1.2.

4.2 Flows and Queues

To get a better understanding of the architecture one must first be familiarized with certain terminology that will be repeated in the following discussions. A flow, or specifically a packet flow is a stream of packets traveling from one source to a destination. This destination can be another host machine, a server or even multiple machines. The issue of interest is how to distinguish between flows right from the level of software applications, through the operating system and down to the networking hardware. Consider an example, where a host machine is simultaneously connected to a voice call on Skype as well as downloading a file from a ftp server. In this scenario, packets used for the voice call constitute a single flow, which is distinguishable from the connection to the ftp server which is identified as another flow. QCN allows implementation of packet processing on a flow level, which enables application of QoS requirements.

Queues on the other hand exist mostly in the hardware domain of network equipment. Queues are utilized to temporarily store network traffic until it is ready for packet processing (an ingress or incoming queue) or until it is ready for serialization and transportation on a physical medium or wire (an egress or outgoing queue) [33]. Queues are necessary because in any network device like a switch or a router, it is likely that input packets (destined for the same port) are received from several applications, but only one flow can be active (that is, be forwarded through that port) at one time. Hence the device must temporarily store packets, and thus queues are necessary. Multiple flows reside in a single queue. For example, one queue can be deemed to be a *priority* queue and all flows which demand a higher QoS can be placed in that queue. In the event of network congestion, the device will use all the available bandwidth to try and transfer the packets which are in the priority queue and thus maintain high QoS as desired.

This is where QCN becomes interesting. Consider an example where both a Voice Over IP (VOIP) application and a file transfer application are using the same queue. Suppose a network congestion occurs due to which the file transfer's flow is being hampered, i.e. the remote server might be busy or not responding. One of the most popular solutions which currently aim to mitigate this is the Ethernet 'pause frame' [34] which causes the entire queue to be paused. What this means is that now both the flows for the VOIP application (flow 1 in Figure 4.1) and that of the file transfer (flow 2 in Figure 4.1) are paused. This is detrimental for the application which was not facing congestion because it gets paused as well. QCN allows us to implement a similar 'pause' effect at a flow level, allowing other flows in the same queue to continue transmission even if one flow is facing congestion. It only pauses the congested flow. The implementation of this 'pause' effect in QCN is different from [34] but this is one of the reasons why the *Hydra* supports QCN.

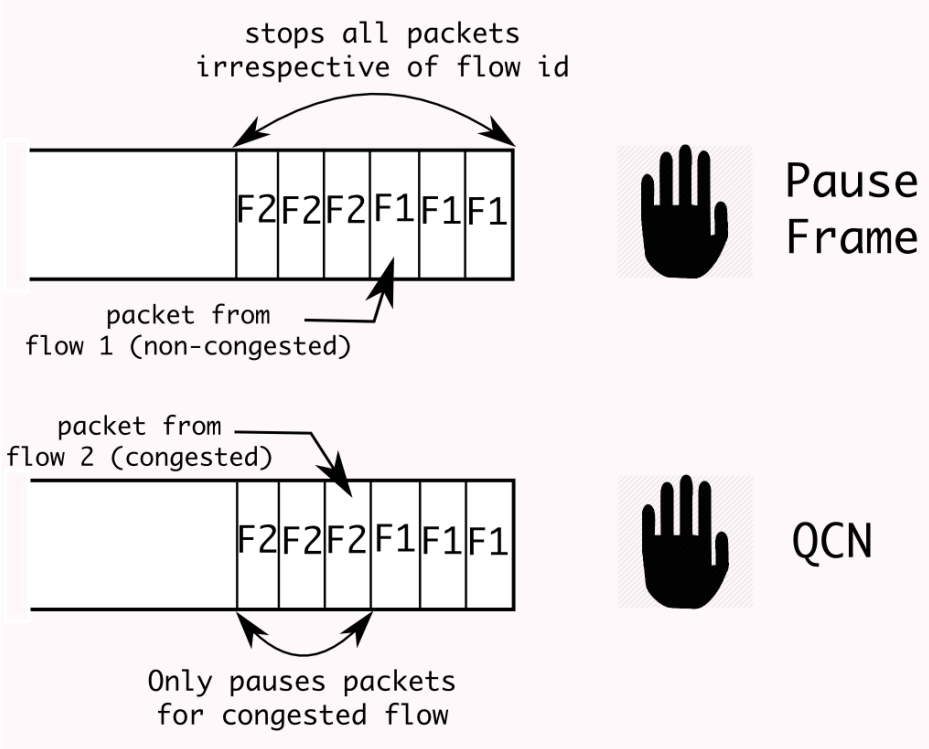


Figure 4.1: Two queues showing how congestion can be handled

4.3 Descriptor Organization

The PacketArc *Hydra* specification [35] states that the *Hydra* supports up to 1024 Virtual functions, each with 8 queues and up to 65536 flows. Virtual functions (VF) can be viewed as individual PCIe endpoints, so essentially each VF is a host machine (physical or virtual). An explanation of how VFs, queues, flows and actual data packets are organized is provided. For the sake of simplicity, consider one host machine. Software applications running on the machine utilize its system memory (RAM) to store packets of data which are being transferred to and from the system memory by the application. *Buffer descriptors* are place holders which are address buffers and point to the relevant packet data. Since packet data is organized in the form of chunks and may or may not be contiguous, complete sets of such buffer descriptors (which point to the packet data) are required.

For a single flow, a *flow descriptor* (another place holder which stores an address) points to a set of buffer descriptors which in turn point to the packet data associated with the flow as described above. To service an active flow, the *Hydra* must have the relevant flow descriptor on its on-chip memory, using which it will first read to get the relevant buffer descriptors and store them on chip. When the flow is

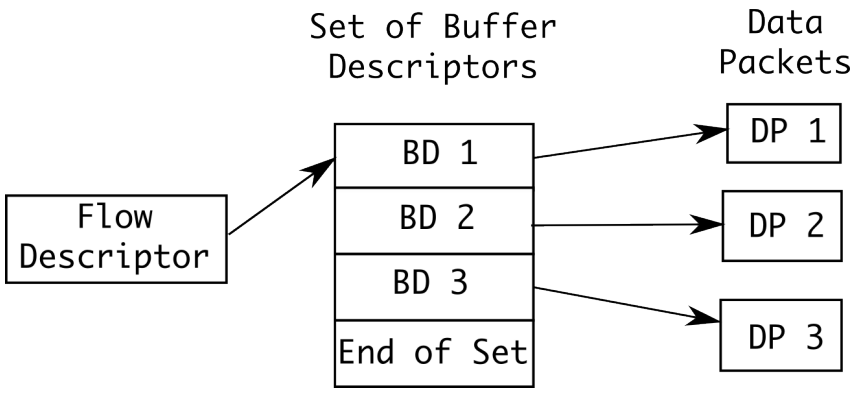


Figure 4.2: Data architecture

scheduled for processing, the *Hydra* will use the on-chip buffer descriptors to locate the packet data in the host's system memory and transfer them by issuing read or write request over the PCIe interface. This is shown in Figure 4.2. Now that the mechanism using which data packets are transferred by the *Hydra* is understood, it is imperative to explore certain options to fully support the specifications. At each step of the calculations, detailed discussions are provided.

4.3.1 Full Hardware solution

In this approach it is assumed that all required flow descriptors are stored directly on the on-chip memory of the *Hydra* device. This removes the latency associated with fetching the flow descriptors from the system memory every time a new flow is scheduled for service. What is gained in speed, is lost in resources required, i.e. the size of memory on-chip needs to be increased, and that is going to cost chip area. Calculations are presented below.

According to [35], 1 Flow Descriptor is 383 bits long. So, the size of on-chip memory is
= Number of VFs * Number of flows per VF * Length of one flow descriptor
= 1024 * 65536 * 383 bits
= 4.2 GB

Since this amount of memory on-chip is not feasible for a system-on-chip, alternatives must be explored. Considering the platform for which the *Hydra* is being developed for, this realistic value of on-chip memory is 2MB. If one must absolutely must choose the full hardware approach, realistically either the support for the number of VFs must be decreased or the number of flows each VF is allowed to handle must be reduced.

4.3.2 Descriptor manager solution

To overcome this problem, utilizing the host machine's system memory to store the entire set of descriptors is proposed. Simultaneously, a descriptor cache resides in the local on-chip memory of the *Hydra*. The solution is based on intelligent prefetching of these descriptors from system memory to the on-chip cache over a PCIe connection.

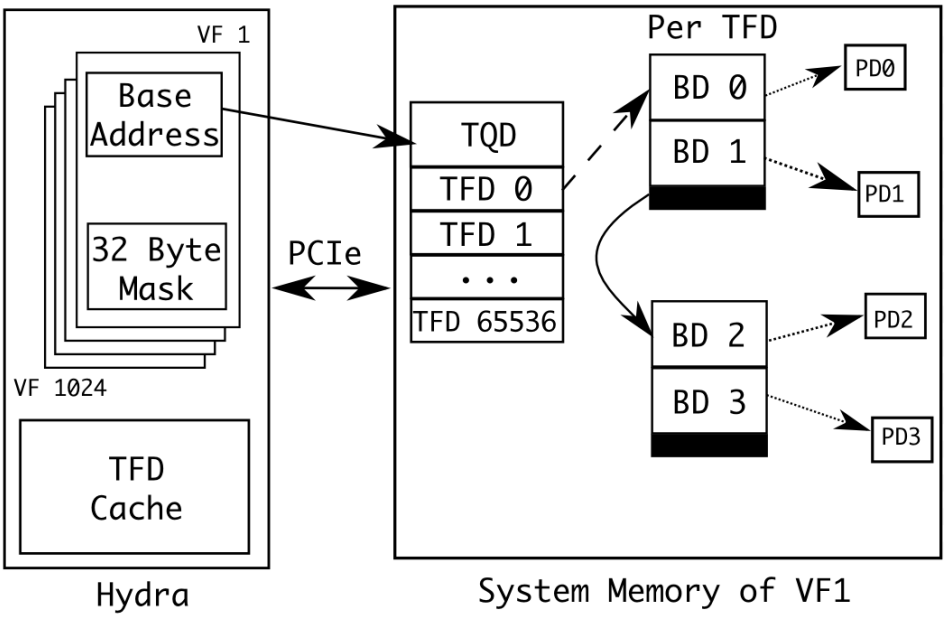


Figure 4.3: Proposed Solution

The architecture depicted in Figure 4.3 is explained below. The block on the left side is the *Hydra* device and the block on the right is representing the system memory of a host machine (VF) connected to the *Hydra*. For each VF there is a base address pointing to the start of a contiguous block allocated in host memory. This block in the system memory, as shown, is divided into a Transmit Queue Descriptor (TQD) area and a number of Transmit Flow Descriptors (TFDs). (One TFD for each flow to be precise.)

The TQD area is a 64kbit block, where each bit corresponds to each flow and indicates whether the flow has packets waiting in host memory. This idea is shown in Figure 4.4.

The TQD area is divided into chunks so that 32 chunks per queue exist. For each of these chunks, a corresponding bit mask is kept on-chip, which tells the *Hydra* about changes that may occur in the TQD area for that particular VF. Since there are 8 queues per VF and 32 bits per queue, 32 Bytes of mask info (per VF) is updated (over PCIe) immediately after servicing each flow, so that the *Hydra* knows which flows need to be scheduled next. The TFDs on the other

1	0	0	1
0	0	1	0
1	0	0	0
0	1	0	1

Figure 4.4: The TQD area

hand, contain all the configuration and status data for the flows including target bandwidth etc. required for the QCN protocol.

Hence, according to this proposal, the on-chip memory needs to store

- Byte masks for all VFs i.e. (32*1024) Bytes = 32 KB.
- The TFD cache, which is shared between all VFs.

This proposed solution aims to reduce the cost of chip area (as compared to the results in section 4.3.1) by using this architecture highlighted above. The critical issue is to control the number of flows that are stored in the on-chip TFD cache, and the next section sheds light on this.

4.4 Design considerations

It is already known that QCN can be implemented in hardware, as seen in [36], but the existing implementations are mostly proof-of-concept in nature. A realistic calculation of the extent to which QCN must be supported for customers in the data center market is required at this point, and this part of the thesis is geared towards the same.

4.4.1 Buffer Descriptor management

Once a flow is scheduled, in order to maintain a steady flow of data according to the requirements (i.e maintaining bandwidth specifications) there is a need to fetch the buffer descriptors from the system memory which point to actual packet data. The process is three fold.

- Schedule a flow X.
- Retrieve all buffer descriptors which are pointed to by the TFD of flow X.

- Transfer packet data pointed to by these buffer descriptors.

To maintain bandwidth, the number of buffer descriptors that need to be on-chip, when a particular flow has been scheduled, is calculated. If the target bandwidth is B_t Bytes/second and if each buffer descriptor points to a fixed size of packet data l_p Bytes, then the number of buffer descriptors that need to be present on chip per unit time to meet B_t requirements is

$$n_b = \frac{B_t}{l_p} \tag{4.1}$$

The transfer of buffer descriptors will be handled by a separate stand-alone module which will be operating using a proprietary 'fire-and-forget' algorithm. This module is beyond the scope of this thesis.

4.4.2 Descheduling of flows

A clarification is made between two concepts viz. descheduling and purging. Descheduling is the process whereby a scheduled or active flow is paused or stopped due to certain conditions, which are elucidated below. Purging on the other hand, is the process by which TFDs are removed from the on-chip cache. Depending on the purge policy, i.e how descriptors are transferred between the on-chip memory and the system memory, descheduling and purging might have the same result for example, in the case where a TFD is purged as soon as it is descheduled.

In such a case, two events occur once a flow is descheduled. Firstly, the control information on the header of the descriptor is updated and written back to the system memory. And secondly, a new TFD is retrieved from the system memory to the on-chip memory (which is waiting to be scheduled). Both these steps will introduce an overhead into the PCIe bandwidth, especially if one considers that per unit time (seconds), the number of flows that will be descheduled can be considerably high.

The quantification of this descheduling process is now presented by investigating the factors which are responsible for descheduling a flow (after it has become active, or scheduled).

- **Maximum burst length (MBL) reached.**
  MBL denotes the amount of data that can be sent as one continuous burst. In more general terms, it determines the duration of time for which a flow can be scheduled as active. For example, if the MBL is 2kB and the currently scheduled flow is transmitting data, as soon as it reaches that threshold, it will be descheduled. If data transfer is occurring at a rate of 2kB/s for this example, and there are 2 flows which are active, the scheduling should look like as shown in Figure 4.5.

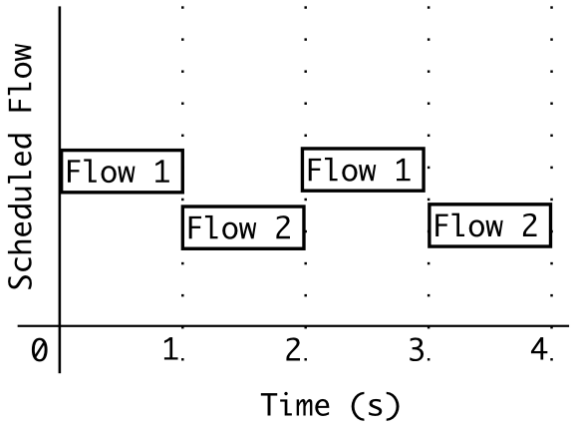


Figure 4.5: Descheduling flows due to MBL reached

- **Target rate exceeded.**
  In the QCN protocol the flows are transmitted using a custom target rate (TR) and a current rate (CR). The target rate sets a limit on the maximum transfer rate for the flow in question, and when a flow is scheduled, with every byte of data that is transferred, the flow's current rate is dynamically updated. If during this process, the flow's current rate exceeds its target rate, then the flow is descheduled because other flows need to be serviced. This process is depicted in Figure 4.6.

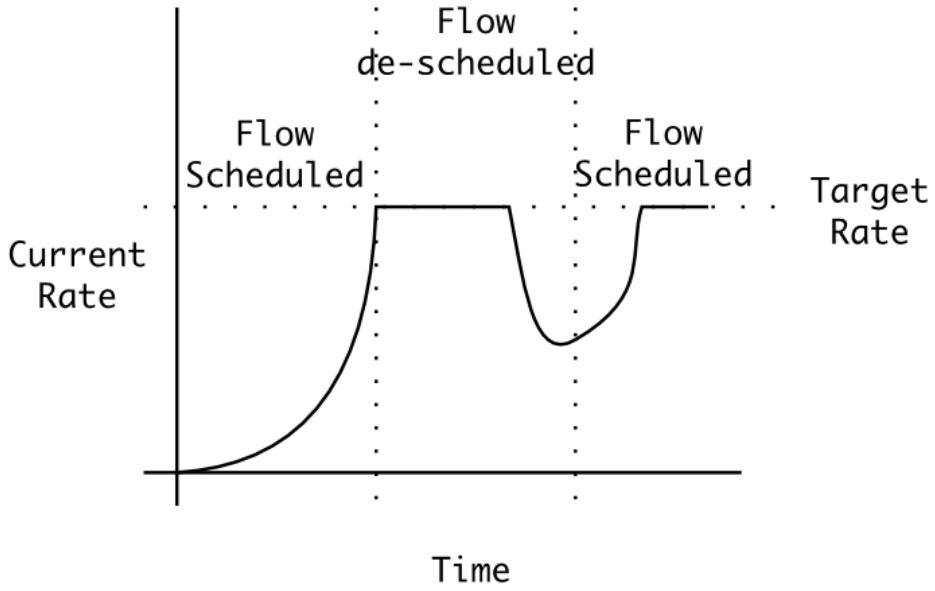


Figure 4.6: Relationship between scheduling and target rate

- **Packet data exhausted**
  This option is only invoked when a scheduled flow has completed transferring all the data packets it was assigned to. Since by definition, QCN is particularly suited for long-lived flows, the occurrence of this event is particularly scarce in frequency.

What must be remembered is that some of the above factors are not completely independent. For example, for a larger MBL, every time a flow is scheduled, it will transfer packets for a longer duration, effectively increasing its CR, which can reach the TR and hence lead to the flow being descheduled before it has transferred the exact amount of data as assigned by the MBL.

4.4.3 Size of on-chip RAM

The *Hydra* design has to be accommodated on a FPGA provided by the interested parties, and since a preliminary idea about the target environment is available, and the hardware footprint of the existing design is also known, it can be calculated that the maximum amount of memory resources available for the *Hydra* is around the 2MB mark. This is actually a good starting point for the calculations which are shown below.

First, an investigation of the relationship between the amount of on-chip RAM (R_c)and the number of TFDs (n_f) that can be stored on chip is made. Since there are 1024 VFs and each has a 32 byte mask on-chip (Figure 4.3),

$$n_f = \frac{R_c - (32 * 1024)B}{l_f} \tag{4.2}$$

$$=> n_f = \frac{R_c - 32kB}{l_f} \tag{4.3}$$

where l_f denotes the length of a TFD.

Now, it is known that l_f is 383 bits from the specifications [35], hence the reader can look at Table 4.1 to get an effective idea of what equation 4.3 suggests.

So, for each VF up to 42 flows can be supported if all 1024 of them are connected. One cannot comment at this point whether this is the optimal distribution of available resources, because in the real use scenario, a client may desire more number of flows and less number of VFs or the other way around. But using the information that is available during the design phase, this is what can be offered, and further calculations will be based on this result.

4.4.4 Time available to fetch descriptors

Once a flow is scheduled, the actual duration before which it gets descheduled is determined by three factors as discussed in section 4.4.2. Assuming that all flows

Table 4.1: Number of flows supported by size of RAM

R_c(kB)	l_f(kB)	n_f	Flows/VF
64	0.048	685	0
128	0.048	2054	2
256	0.048	4792	4
512	0.048	10267	10
1024	0.048	21218	20
2048	0.048	43121	42
4096	0.048	86926	84

are long-lived flows (so they don't run out of packets) this relationship can be represented by the equation

$$t_s = MBL/B_t \tag{4.4}$$

where MBL denotes the maximum burst length, t_s represents the time for which the flow is scheduled and B_t represents the target bandwidth. Since up to 42 flow descriptors can be stored on chip for every VF as shown in Table 4.1, assuming that a cache miss does not occur, i.e. a descriptor which is not on chip is not scheduled (this must be taken care of by the cache purging algorithm which is beyond the scope of this thesis), the time required to schedule all of these 42 flows (T) can be calculated as

$$T = 42 * t_s \tag{4.5}$$

This is the same time available to fetch up to 42 new flow descriptors which will be scheduled in the next round. A clearer estimation of this time is warranted for decisions regarding the design and hence, calculations are presented below. Here, the MBL is kept constant at 200kb, while the target bandwidth supported is varied, and the time available for fetching new descriptors is calculated.

Table 4.2: Time to fetch descriptors as limited by target bandwidth

MBL (Mb)	B_t (Mbps)	T (micro-second)
0.2	1	8400000
0.2	10	840000
0.2	100	84000
0.2	1000	8400
0.2	10000	840
0.2	100000	84

So, from Table 4.2 it can be gathered that to maintain a 10G target bandwidth, 0.84 ms is available to fetch up to 42 new flow descriptors from a single VF.

4.4.5 Latency on PCIe

From the previous sections a clear picture of how many descriptors that can be stored on chip, how many need to be fetched during a particular interval of time, and the actual extent of that interval has been shown. But an important consideration is the latency on the PCIe link because one cannot afford to issue a read request and receive its completion data beyond the times shown in Table 4.2.

PCIe latency depends on factors such as the configuration of the link (number of lanes, generation of PCIe etc.) and the size of the payload (short packets contain a few bytes of useful data, while longer payloads can be up to 4kB). In light of this, the designer suggests running tests on actual physical hardware with such configurations to get a more exact understanding of latency in PCIe, but for this thesis, data from other simulations [37] will be used.

The latency for read packets of size 2kB, (which have been split into smaller requests of 128 bytes or 256 bytes) is shown in Table 4.3 [37, page 9]. Time indicated in the data is for a complete round trip, i.e from the transmission of the read request from the requester to receiving the data from the completer.

Table 4.3: PCIe latency for 2kB packets

Path	**Latency (ns)**	
Max Payload setting	128	256
Number of requested packets	16	8
Read Request		
Tx Application	15	6
Data Link + Transaction layers	15	15
SERDES + PMA + PCS + MAC	20	20
SERDES + PMA + PCS + MAC	30	30
Data Link + Transaction layers	15	15
Rx Application	15	6
Fabric + DRAM cont. + DRAM (open)	51	51
Read Completion		
Tx Application (builds response packet)	63	6
Data Link + Transaction layers	63	15
SERDES + PMA + PCS + MAC	20	20
SERDES + PMA + PCS + MAC	30	30
Data Link + Transaction layers	63	111
Total to get 1st byte of 1st packet back	400	325
Rx application (waits for all bytes at link speed)	608	560
Total: Source->Link->CPU->Link->Sink	**1008**	**885**

As can be observed above, it takes around 1 micro second to complete such a request due to latency on the PCIe. Now, consider the amount of data (D) that

is fetched as a block of descriptors,

$$D = 42 * l_f => D = 42 * 383bits = 2.01kB \tag{4.6}$$

it is found to be comparable to the size of the data for which Table 4.3 is valid. Since even for a 100Gbit target rate, 84 micro seconds are available to do the fetch operation (Table 4.2), and Table 4.3 suggests that latency on the PCIe for packets of that size is significantly less, it can be concluded that the proposed technical solution is immune from failures due to latency on the PCIe.

4.4.6 Overhead due to flow fetch

In section 4.4.2 it has been shown that fetching descriptors from the system memory is likely to introduce an overhead into the PCIe bandwidth. It is now time to investigate these effects in detail.

If the number of flows being fetched is denoted as n_{fetch} then the overhead in bandwidth (O_v) is

$$O_v = (l_f + PCIe_b) * n_{\text{fetch}} \tag{4.7}$$

where $PCIe_b$ denotes the header and CRC of the PCIe packet which is independent of the payload size. In order to equate overhead with the bandwidth, overhead introduced per second O_s, is given by

$$O_s = O_v * f_{\text{fetch}} \tag{4.8}$$

where f_{fetch} is the frequency of such a fetch. Assuming that all active flows have the same B_t and each VF has one flow active per clock cycle, total bandwidth requested by all active flows (B_a) is given by

$$B_a = 1024 * B_t \tag{4.9}$$

Equation 4.9 gives the amount of PCIe bandwidth that will be utilized to actually transfer the data packets to and from system memory to the flexswitch.

To maintain a system which is functional despite the overhead introduced by fetching flow descriptors from memory, one needs to ensure that

$$O_s < B_T - B_a \tag{4.10}$$

where B_T is the total bandwidth supported by the PCIe link.

Using the above equations, one can find an expression for the target bandwidth of the flows that can be supported even with the overhead that is generated.

$$B_t < \frac{B_T - ((l_f + PCIe_b) * f_{\text{fetch}} * n_{\text{fetch}})}{1024} \tag{4.11}$$

One of the reasons why the PCIe protocol is chosen for this project, as described in section 1.2, is its high B_T which can be maximized by choosing a x16 lane (widest configuration) and using the hardware platform which supports the latest version of the protocol (up to 31.5 GB/s). Considering the availability of this hardware, one can assume B_T = 31.5 GB/s. Furthermore, from section 4.4.3 it is known that n_{fetch} = 42 * 1024 = 43k, and from section 4.4.4 the value of f_{fetch} is found to be 1/T. l_f is 383 bits from the Packet Arc specifications and the value of $PCIe_b$ can be determined from Figure 3.5 as 96 bits. Using these values in equation 4.11 the calculations as shown in Table 4.4 are obtained.

Table 4.4: Effective bandwidth as a function of frequency of flow fetch

B_t(Mbps)	B_T(GB/s)	n_f	l_f(bits)	$PCIe\_b$(bits)	f(Hz)
222	31.5	42	383	96	1190.4
243	31.5	42	383	96	119.04
245	31.5	42	383	96	11.9
246	31.5	42	383	96	1.19
246	31.5	42	383	96	0.119

From Table 4.4 it can be interpreted that if there were 1024 VFs and all were transmitting flows at the same time, and all of these flows had the same B_t, then target rates up to 240 Mbps can be supported. But one must remember that in real life cases, not all flows need to be configured the same. Some applications (and their corresponding flows like for example a media transfer) can be assigned a higher (1Gbps) bandwidth and others can do well with a lower (kbps) bandwidth (for example text messaging). The actual permutation will definitely be dependent on the client's specification, and this thesis has calculated the overhead due to flow fetches which will have an impact on the available target bandwidth for the flows in question. The total bandwidth available for all VFs to share is in the range of (245 Mbps *1024) = 250 Gbps. The value of B_t from Table 4.4 seems to have saturated because of the nature of equation 4.11 where B_T is a significantly large value.

4.5 Summary and results

This chapter has presented a theoretical exploration of the viability of QCN support for the proposed technical solution. Through calculations, discussions and results at every step of the project, it has been shown that QCN can be implemented, using the PCIe support from the IP developed in the previous chapter. So, it can be argued that the primary goal of this section of the project has been

met. At this point, the other goals of the project as discussed in section 4.1 are revisited.

In section 4.3, the organization of the data, and the descriptors required to handle the same has been presented. Specifically, it has been shown that the full hardware solution in section 4.3.1 has a limited feasibility and section 4.3.2 has discussed an alternative solution for descriptor management. This fulfills the second goal from section 4.1.

Section 4.4 has presented a multitude of design considerations that must be considered in order to deem the proposed solution to be technically sound. In particular, section 4.4.3 has suggested that the optimum size of the on-chip memory is 2MB. So, the third objective of the project has been satisfied.

Utilizing the calculations from sections 4.4.4, 4.4.5 and 4.4.6, it can be said that the PCIe standard has the capacity to handle the overhead and latency requirements to enable QCN support in the proposed solution, and hence the fourth goal of the project has been achieved.

Finally, as reflected in the discussion presented throughout this chapter, the number of client machines that are to be supported by the proposed solution, can vary as per the specifications as well as the client requirements. However, for a clear depiction of the results from section 4.4.6, a case study of a typical client is presented below.

It is assumed that the client is a mid sized growing business, who are running roughly 500 machines in their office. Since its mid sized and is a growing enterprise, the client is likely to plan ahead for the future when ordering networking infrastructure. In the typical user-case, the client will require 100*1Gb ports, 250*10Mb ports and 150*1Mb ports. The 1Gbps ports are standard to maintain connections to other offices and the external world, servers etc. The 10Mbps ports form the majority of work stations in offices around the world. And for typical office tasks such as email or surfing, 1Mbps ports are sufficient. The effective (useful) bandwidth that needs to be supported for this client is

$$B_{\text{eff}} = 100 * 1G + 250 * 10M + 150 * 1M = 102.65Gbps \tag{4.12}$$

which is less than the maximum that can be supported (250Gbps). Hence, this project has been successful in suggesting a practical estimation of client machines that can be supported by the proposed solution. In conclusion, it can be stated that all the goals from section 4.4.1 have been met by this part of the thesis.

Chapter 5

Conclusion

This thesis is a compilation of three parts which is the background work for the implementation of the proposed solution from section 1.2. These three parts are

1. Designing and implementing a traditional simple NIC in verilog.
2. Studying the PCIe interface, implementing an IP and interfacing it to a physical host. Controlling the on-chip hardware over PCIe to prove that it can be used in conjunction with the solution.
3. Investigating a novel descriptor management architecture which allows the solution to support QCN. This involves identification of trade-offs and calculations to determine the feasibility of a feature set.

The goals and results of each part of the project has been discussed in the preceding chapters. So, in this conclusion, I will point out the limitations and scope of future work.

In the first part of the project, I have worked with a setup of networking hardware and a NIC architecture has been designed in Verilog. The traditional approach outlined in section 1.1 has been implemented, and through this part, I have shown that the major issue of this approach is the (finite) amount of on-chip buffer memory. An operating frequency of 282 MHz has been achieved for a Virtex 5 platform. The limitation of the NIC is that it does not include a PHY for the MAC and it lacks an industry standard for the memory interface (like SATA). Due to these two issues, this NIC cannot be used to interface a client machine to the network.

Further work on this part of the project does not need to be pursued as far as the goals in section 2.1 are concerned. However, if this NIC is supplemented with a MAC IP (with support for the PHY and link logic layer) and support for the SATA interface is built in as well, then it can be functional for a computer.

In the next part of the project, which is discussed in the third chapter of this thesis, I have explored the PCIe standard and developed a customized IP which can be used to interface the technical solution to a (client machine's) motherboard.

The reader has been provided with an understanding of the protocol architecture and how transactions and packets are significant for using PCIe. At the end of this step, I am now capable of interfacing any device using this protocol, and also have gained an insight into how to design driver software to control and utilize the advantages offered by PCIe. The IP has achieved an operating frequency of 63.5 MHz, and the total hardware solution (along with the supplementary blocks on the development board) consumes 2346mW of power.

The limitation of this part of the project is the constraint of the target platform. The board used has support for the PCIe 2.2 standard and the physical PCIe port is a 1x type. Furthermore, since it has been built on top of a Xilinx PCIe core, the IP will not work on other platforms and the driver includes support for the Linux platform. Further work on this has already been started. Currently I am working with a 16x PCIe 3.0 board, and the IP is undergoing several modifications such as the addition of a AXI-4 streaming interface. In the long run, it is suggested that the IP is supplemented with cross-platform support so that it can be reused for most designs.

The final part of the project is to investigate the feasibility of implementing QCN on the proposed solution, in conjunction with the available target platform. This step, as shown in chapter 4 of this thesis, also involves the systematization of the *Hydra* architecture and the novel descriptor management system and through calculations which involved real world network traffic configurations, I have explored variables such as size of the on-chip memory, latency on PCIe and overhead on bandwidth due to the fetching of descriptors.

At the end of this part of the project, I have shown that it is feasible to implement QCN, but there are some restrictions on the number of flows and VFs that need to be kept in mind. The limitation of this part is the shortage of test data. Though all the assumptions made during the calculations are sound from a theoretical perspective, from my limited experience, I have found that once a solution moves from napkin to silicon, a multitude of issues are bound to creep up. So, the future work for this part is to implement the solution, and conduct more tests with the specific hardware requested by the client.

The major experience which I gathered from this project is an effective transition from the theory learned in class to the industry. In the first two steps of the project where I was working in Verilog and building hardware I applied my knowledge from the 'Introduction to structured VLSI' course at Lund University. I also took other courses as electives which helped me to work in the field of computers and network design, particularly 'Computer Architecture' which enabled me to understand the concepts of how system memory is shared, where the operating system lies, and the function of driver software. My course in 'Embedded systems' enabled me to get a clearer understanding of how hardware and software co-exist and interact with each other (for example how registers in the hardware world and variables in the lower level software domain work in tandem), which helped me in my work with the driver and the board. I also took the course 'Computer Communications' which first introduced me to the idea of data packets and the basics of Ethernet and networking which I utilized extensively during the first step of the project

when I was building the NIC.

When I started my thesis at PacketArc AB, I was introduced to new ideas and novel design architectures which were the result of years of experience in the switching silicon industry. It was my function to investigate these ideas in depth and determine the possibility of them being implemented in hardware. At the end of the project, I have been able to show that it is feasible to convert these novel ideas into a finished product and so the future work after this thesis is to start the implementation of the Hydra which is now scheduled for completion in Q3 2014.

Bibliography

[1] 42U. Industry leader of server racks. 42U, 2014. URL `http://www.42u.com/`. [Online; last accessed 21-04-2014].

[2] Douglas Comer and Larry Peterson. *Network Systems Design Using Network Processors*. Prentice-Hall, Inc., 2003.

[3] Jon Postel. Rfc 793: Transmission control protocol, september 1981. *Status: Standard*, 88, 2003.

[4] Karl-Johan Grinnemo, Johan Garcia, and Anna Brunstrom. Taxonomy and survey of retransmission-based partially reliable transport protocols. *Computer Communications*, 27(15):1441–1452, 2004.

[5] Alexander Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in qos switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.

[6] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[7] Bob Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, et al. Recommendations on queue management and congestion avoidance in the internet. 1998.

[8] International Committee for Information Technology Standards (formerly NCITS). Fibre channel âĂŤ fibre channel backbone - 5 (fc-bb-5). 2010.

[9] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[10] Cisco. Quantized congestion notification and todays fibre channel over ethernet networks (white paper). Cisco Public Information, 2011. URL `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-630674.html`. [Online; last accessed 29-10-2013].

[11] Jeffrey Shafer and Scott Rixner. A reconfigurable and programmable gigabit ethernet network interface card. *Network*, 1(2):3, 2006.

[12] Consumer nic products. Amazon. URL `http://www.amazon.com/b/ref=dp_brw_link?ie=UTF8&node=13983711`. [Online; last accessed 04-04-2014].

[13] Xilinx. 10gemac ip product information. Xilinx, 2013. URL `http://www.xilinx.com/products/intellectual-property/DO-DI-10GEMAC.htm`. [Online; last accessed 09-04-2014].

[14] Arasan. Xgmac ip product information. Arasan, 2013. URL `http://arasan.com/products/ethernet/10ge-mac-2/`. [Online; last accessed 09-04-2014].

[15] Altera. Xaui ip product information. Altera, 2013. URL `http://www.altera.com/technology/high_speed/protocols/10gb_ethernet/pro-10gb_ethernet.html`. [Online; last accessed 09-04-2014].

[16] Don Anderson, Tom Shanley, and Ravi Budruk. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[17] PCI Local Bus Specification. Revision 2.3. *MOTOROLA Ltd., European Literature Center*, 2008.

[18] David Mayhew and Venkata Krishnan. Pci express and advanced switching: evolutionary path to building next generation interconnects. In *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 21–29. IEEE, 2003.

[19] Weikuan Yu, Ranjit Noronha, Shuang Liang, and Dhabaleswar K Panda. Benefits of high speed interconnects to cluster file systems: a case study with lustre. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[20] Faraj Nassar, Jan Haase, Christoph Grimm, Herbert Nachtnebel, and Majid Ghameshlu. Design and simulation of a pci express based embedded system. In *IEEE Austrian workshop on microelectronics (AUSTROCHIP08), Linz, Austria, October*, 2008.

[21] Xillybus. Xillybus. URL `http://xillybus.com/`. [Online; last accessed 14-04-2014].

[22] Xilinx. Virtex 5 pcie ip product information. Xilinx, 2013. URL `http://www.xilinx.com/products/intellectual-property/V5_PCI_Express_Block_Plus.htm`. [Online; last accessed 14-04-2014].

[23] Xilinx. Ml507 development platform information. Xilinx, 2013. URL `http://www.xilinx.com/products/boards-and-kits/HW-V5-ML507-UNI-G.htm`. [Online; last accessed 14-04-2014].

[24] Mike Rose. Fpga pcie bandwidth. *University of California San Diego*, 7, 2010.

[25] Ray Bittner. Speedy bus mastering pci express. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 523–526. IEEE, 2012.

[26] Anandhavel Sakthivel. Implementing a pci-express amba interface controller on a spartan6 fpga. Master's thesis, Chalmers University of Technology, February 2013.

[27] Peng Yu, Li Bo, Liu Datong, and Peng Xiyuan. A high speed dma transaction method for pci express devices. In *Testing and Diagnosis, 2009. ICTD 2009. IEEE Circuits and Systems International Conference on*, pages 1–4. IEEE, 2009.

[28] Qiang Wu, Jiamou Xu, Xuwen Li, and Kebin Jia. The research and implementation of interfacing based on pci express. In *Electronic Measurement & Instruments, 2009. ICEMI'09. 9th International Conference on*, pages 3–116. IEEE, 2009.

[29] W Gao, A Kugel, R Männer, and G Marcus. Pci express dma engine design. *CBM Progress Report*, page 54, 2007.

[30] Lattice Semiconductors. Power considerations in fpga design (white paper). Lattice Semiconductors, 2009. URL `http://www.latticesemi.com/~/media/Documents/WhitePapers/NZ/PowerConsiderationsinFPGADesignLatticeECP3.PDF`. [Online; last accessed 14-04-2014].

[31] National Instruments. Fpga fundamentals (white paper). National Instruments, 20. URL `http://www.ni.com/white-paper/6983/en/`. [Online; last accessed 18-04-2014].

[32] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):203–215, 2007.

[33] Aaron Balchunas. Qos and queing(white paper). Router Alley, 2010. URL `http://www.routeralley.com/ra/docs/qos_queuing.pdf`. [Online; last accessed 29-10-2013].

[34] Ieee standards for local and metropolitan area networks: Supplements to carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications - specification for 802.3 full duplex operation and physical layer specification for 100 mb/s operation on two pairs of category 3 or better balanced twisted pair cable (100base-t2). *IEEE Std 802.3x-1997 and IEEE Std 802.3y-1997 (Supplement to ISO/IEC 8802-3: 1996; ANSI/IEEE Std 802.3, 1996 Edition)*, pages 0–324, 1997.

[35] Packet Arc Inc. Dragon chipset specification. PacketArc AB, 2012. [Protected Documentation].

[36] Masato Yasuda & Balaji Prabhakar. 10g qcn implementation on hardware. NEC corporation, Japan, 2009. URL http://www.stanford.edu/~balaji/presentations/au-yasuda-10G-QCN-Implementation-1109.pdf. [Online; last accessed 29-10-2013].

[37] Brian Holden. Latency comparison between hypertransport and pci-express in communications systems. HyperTransport Technology Consortium, 2006. URL http://www.hypertransport.org/docs/wp/Low_Latency_Final.pdf. [Online; last accessed 29-10-2013].

LUND
UNIVERSITY

http://www.eit.lth.se