12-2013

An Open Source, Line Rate Datagram Protocol Facilitating Message Resiliency Over an Imperfect Channel

Christina Marie Smith
University of Arkansas, Fayetteville

An Open Source, Line Rate Datagram Protocol Facilitating Message Resiliency Over an Imperfect Channel

An Open Source, Line Rate Datagram Protocol Facilitating
Message Resiliency Over an Imperfect Channel

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Christina Smith
University of Arkansas
Bachelor of Science in Computer Engineering, 2011

December 2013
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____

David Andrews, Ph.D.
Thesis Director

_____

Christophe Bobda, Ph.D.
Committee Member

_____

Dale Thompson, Ph.D.
Committee Member

Abstract

Remote Direct Memory Access (RDMA) is the transfer of data into buffers between two compute nodes that does not require the involvement of a CPU or Operating System (OS). The idea is borrowed from Direct Memory Access (DMA) which allows memory within a compute node to be transferred without transiting through the CPU. RDMA is termed a zero-copy protocol as it eliminates the need to copy data between buffers within the protocol stack. Because of this and other features, RDMA promotes reliable, high throughput and low latency transfer for packet-switched networking. While the benefits of RMDA are well known and available within the general purpose and high performance computing community, only a few open source and portable RDMA capabilities exists for the FPGA community. Within the limited availability of solutions for FPGAs, many rely on standard Internet Protocol. This thesis presents an open source and portable RMDA core that enables line rate scaling for data transfer over packet-switched networks over Ethernet for the FPGA community. An RDMA protocol in which the currency is Datagrams is designed, implemented and tested between two Xilinx FPGA's over a Layer 2 switch. The implementation does not rely on an Internet Protocol and is portable, simple and lightweight. Latency, throughput and area will be reported and discussed. To foster portability, the core was designed and implemented in Bluespec SystemVerilog and does not utilize any vendor specific technologies.

Acknowledgments

I would like to thank my advisor, Dr. David Andrews, for the pleasure of working in his laboratory. His flexibility, guidance and wisdom throughout my higher education have been vital to my success.

I would like to thank my advisory committee, Dr. Dale Thompson and Dr. Christophe Bobda, for their flexibility, knowledge and time given to me throughout my time at Arkansas.

I would like to thank my mentor and friend, Shepard Siegel, for his continued support, encouragement and mentorship throughout the life cycle of the implementation of the DG-RDMA and my career.

I would like to thank my family for their continued support, love and understanding in all circumstances.

I would like to thank my lab mates, Eugene Cartwright, Azad Fakhari, Sen Ma and Abazar Sadeghian, for their encouragement and words of wisdom.

I would like to thank Xilinx for their generous donation of two KC705 Evaluation Kits for my use to complete this thesis.

Contents

List of Figures

List of Tables

Terms and Definitions

10GbE 10 Gigabit Ethernet.

CPU Central Processing Unit. A hardware component, often referred to as processor or core that processes instructions of a program.

DMA Direct Memory Access. Often referring to hardware devices that can perform memory-to-memory operations without processor assistance.

FIFO First-In, First-Out. The notion of processing in a first come first serve basis.

FPGA Field Programmable Gate Array. An integrated circuit that can be configured/re-configured by a designer.

GMAC Gigabit Media Access Controller.

GPU Graphics Processing Unit. A hardware component specialized for graphics processing.

HDL Hardware Description Language (e.g. VHDL or Verilog).

HW Hardware.

I/O Input/Output.

IP Intellectual Property.

LFSR Linear Feedback Shift Register.

LUT Lookup Table. A digital building block used to implement N-bit binary functions via lookup operations.

MII Media Independent Interface.

OS Operating System.

PAR Place and Route.

PCIe Peripheral Component Interconnect Express.

PHY Interfaces GMAC to Ethernet cable.

RDMA Remote Direct Memory Access.

RHEL5 Red Hat Enterprise Linux version 5.

RTL Register Transfer Level.

SRAM Static Random Access Memory.

SW Software.

TCP/IP or TCP Transmission Control Protocol in the Internet Protocol.

UDP/IP or UDP User Datagram Protocol in the Internet Protocol.

Chapter 1

Introduction

Much of the focus within reconfigurable computing has been on designing ever faster accelerators, processors, and custom compute components. The literature contains many examples showing that once these computational components have been implemented, the actual system performance falls well short of expectations. While there is not a single reason for these shortfalls it is common to read in the papers' conclusion section that slow data transfer was the bottleneck. Data transfer inefficiencies, or I/O bottlenecks, within a standalone FPGA accelerator can be caused by various effects, ranging from poor state machine and interface designs, to limited physical resources such as wires and internal buffers. The relatively recent ability of an FPGA to host a complete multiprocessor system on programmable chip (MPSoPCs) utilizing standard operating systems is also bringing the traditional inefficiencies associated with traversing deep software protocol stacks and unnecessary copying of data between intermediate buffers into the world of reconfigurable computing [17].

FPGA's are considered as viable data processing components within emerging distributed networks to bring increased processing power into the emerging "big data" domain. New technologies continue to emerge to increase the collection, monitoring and sensing of environments and people. This is causing even larger amounts of data to be collected and stored. FPGA's can bring real time custom processing capabilities up close to the distributed input sensors. This holds the promise of reducing the volume of raw data that needs to be transferred across the network connecting the sensor nodes and data processing systems, and provide faster response times to real time events being monitored. Allowing the FPGA based compute nodes to exchange and process raw data close to the sensor can enable each node to reason locally with at least a subset of data that may have high geographic semantic meaning. These data intensive applications put more pressure on communication infrastructure to perform well and provide data needed for computation. This raises

the importance of addressing all sources of data bottlenecks within and between FPGA's.

When networking FPGAs together, the latency and bandwidth of the physical network is not the dominating bottleneck. Instead, the real bottleneck resides within the latency of the protocol stacks implemented on the FPGAs at each endpoint. As commodity networks become faster more burden is placed on these protocols. Available bandwidth in commodity networks is continuing to increase leading to the need for endpoint protocols that can scale accordingly. Clearly, to utilize the full performance potential of the interconnect, efficient, flexible and low latency protocol stacks are necessary. There have been attempts to adopt general purpose Internet Protocols such as TCP due to their ubiquity [9][7][16]. While a compliant TCP/IP protocol would certainly be the best answer in terms of compatibility and generality, it comes at the cost of performance. To increase performance TCP Offload Engines (TOE) have been proposed [9]. These TCP offload engines are typically implemented as standalone accelerators to reduce the interactions required by the CPU when processing the protocol. Even though this eases the burden of the CPU these TCP offload engines are complex to design and themselves introduce unwanted processing latency. While certainly beneficial many applications do not require the flexibility of a fully compliant TCP/IP protocol. For these types of applications a new OpenCPI Datagram RDMA (DG-RDMA) protocol specification has been proposed. While this protocol does not provide the ubiquity of a TCP/IP protocol it does establish a standard for systems that need acceptable levels of interoperability but at a very low transfer latency.

This thesis introduces the first hardware implementation of the proposed Datagram RDMA (DG-RDMA) specification for low latency data transfers between FPGAs. A driving requirement for this work is portability. The implementation was developed to serve as an open source core that could be easily adopted by system designers regardless of vendor platforms, physical interconnect medium, or FPGA family.

This is an important piece of infrastructure that can be utilized by almost any application, similarly to how DMA engines and buses are used by a variety of applications. In particular, this infrastructure can be useful to application designers who need to partition their application across

multiple FPGA boards. This is also important within development environments which currently do not provide high speed data logging support between the FPGA and host. There are potentially many use cases for line rate scalable, low latency data transfer that does not require attention from the Operating System or host CPU.

1.1 Thesis Contributions and Organization

In this thesis, we present an *open source, portable, line rate* datagram RDMA capability to the FPGA community. This addresses the current lack of no standard or freely available reliable datagram protocol employing RDMA for imperfect channels that does not rely on Internet Protocol.

To support this statement, the thesis provides the following set of contributions:

- Designed a reliable and portable Ethernet RDMA IP core
- IP core does not require Internet Protocol and can be scaled to different line rates
- Implemented prototype on Xilinx KC705
- Tested two KC705s with RDMA IP core over Layer 2 switch
- Created IP core in Bluespec SystemVerilog

This thesis explores how to bring efficient line rate data transfer capabilities for packet switched networks into the open source FPGA community. Chapter 2 describes approaches to solve data transfer issues through related work. Chapter 3 gives details about system implementation of each of the modules in the DG-RDMA IP core. Chapter 4 describes the development and testing environment along with important metrics and results of the measurements. Chapter 5 provides conclusions drawn from the implementation and potential improvements and applications for DG-RDMA.

Chapter 2

Background

2.1 High Performance Communication

Data transfer among reconfigurable compute nodes has been solved over a wide variety of interconnects using various protocols and standards. Some have become building blocks for new methodologies to tackle high speed transfers. Efficient communication has peaked a lot of interest and research which has lead to many proposed solutions. We focus on wired embedded data transfer, specifically FPGA to FPGA transfers.

When considering paradigms for data transfer, two main classes can be considered - one-sided and two-sided communication. When both endpoints are involved in a request-response manner, such as sending a read request and receiving a response, this is two-sided communication. Both the initiator and receiver share information to agree upon the transfer that will take place. One-sided communication is the direct transmission of data without a request. Only the initiator must have all of the information for a transfer to take place, reducing synchronization overhead [6]. Round trip time, a metric for measuring the efficiency of a transaction, is greater for two-sided communication due to the request-response structure. With an efficient solution in mind, one-sided communication is preferred. Because of lessons learned through general and High Performance Computing solutions, we chose to explore Remote Direct Memory Access (RDMA) protocols. RDMA reduces protocol overhead (zero copy) and allows for local completion of a transfer.

RDMA is not the only solution for efficient data transfer. Research groups have designed approaches for mediums like Ethernet and PCIe, some of which rely on Internet Protocols and some of which do not. Internet Protocol is a ubiquitous technology which is often a common choice for communication solutions. Internet Protocol based data transfer solutions are explored in sections 2.1.1. RDMA is introduced in section 2.1.2. DG-RDMA, the protocol of choice for

this work, is described in section 2.2 as well as design input language (section 2.3) and tool choice in section 2.4.

2.1.1 Internet Protocol Based Related Works

When solving any problem, looking to similar and well known problems and approaches can lead to strong, well guided solutions. In conquering high performance communication it is helpful to consider how data transfer is currently done. The Internet Protocol offers a nice foundation on which to base communication standards on top of. TCP/IP and UDP/IP are ubiquitous, common approaches to transferring data between compute nodes. When faced with the same need in reconfigurable computing, many have turned to solutions based on either TCP/IP or UDP/IP as an underlying transport service. TCP/IP is a ubiquitous protocol in packet switched networks. TCP/IP offers a reliable, error-free and ordered delivery of a stream of bytes. It is popular because of its robust qualities and common use. The protocol is not focused on low latency, but resiliency instead. Because of this, the overhead of the protocol is cumbersome. TCP/IP is typically implemented in software and demands a lot of processing cycles from the CPU and Operating System. The need to service the protocol reduces the available cycles of the CPU to be used for computation and degrades overall system performance [3]. In addition, TCP/IP processing by a singular CPU cannot perform well enough for network hardware capable of throughput greater than 10 Gbps according to [19].

TCP/IP Offload Engines (TOE) are a well accepted approach to reducing the CPU utilization for processing TCP/IP protocol stack. For these reasons, a hardware TOE can bring performance improvements to systems utilizing TCP/IP. Many RDMA implementations utilize this approach to reducing the CPU utilization with a TOE. TOE's are hardware modules that process the TCP/IP protocol to relieve the CPU from having to bother with the cumbersome overhead of TCP/IP. Jang, et al. couples this approach with a software implementation of RDMA that is run on the CPU in an FPGA[12]. The TOE designed here is a hybrid TOE which performs CRC calculations and supports zero-copy data transmission. The CPU is responsible for generating and processing RDMA

protocol headers. This approaches utilizes Ethernet as the communication channel. Another solution from Hashimoto and Moshnyaga that utilizes a TOE makes two contributions which yield an effective solution[9]. Reduction of cost of FIFO memory buffers and the parallelization of speculative processing of TCP/IP headers and data transfers via DMA yield a low power, high throughput implementation. This approach also utilizes Ethernet as the communication channel.

Along with TCP/IP, UDP/IP is a common protocol within the Internet Protocol. Unlike TCP/IP, it offers a simple, unreliable datagram transport service. UDP/IP does not guarantee delivery, order of delivery or protection against duplicates. UDP/IP offers minimal features and is used as the framework of many other protocols such as Transmission Control Protocol(TCP) and Stream Control Transmission Protocol (SCTP). Data transmission solutions based on UDP/IP tend to be lighter weight due to less features to implement. One real-time approach by Khalilzad, et al. utilizes UDP/IP to create a 100Mb/s Ethernet core[14]. The advantage to this approach lies in the use of the Reduced Media-Independent Interface, which requires less hardware to connect physically. Unless fewer pins are required for connection to the Media Access Controller, this approach offers few advantages besides being designed with component based methodologies. A more flexible approach is given by Lofgren, et al. [16]. Three different implementations (minimum, medium and advanced) of a UDP/IP core are created based on different network functionality needs. This flexible approach allows the user to decide if they need full functionality, or if some subset will suit the network needs and save area.

Open source solutions are desired to encourage sharing and wide adoption. An open source full TCP/IP core is implemented in [7]. UDP/IP is also implemented in this fully loaded core that is right at 10,000 slices of a Xilinx Virtex 2 FPGA. Alachiotis, et al. created an open source UDP/IP core specifically for FPGA to PC communication via Gigabig Ethernet [1].n This approach is sensitive to area and performance through optimization of transmission cost and checksum calculations. Lieber and Hutchings present a portable, open source communications framework which is created for use with Xilinx-based FPGAs over Ethernet[15]. The hardware/software co-designed core claims to be extensible and is only 600 slices of a Xilinx Virtex 5. The claim for portability is

supplemented with information about what changes are needed to port from one Xilinx FPGA to another. The required changes involve the Ethernet MAC wrapper, the constraints file and an internal change that is required for hardware/software communication(ICAP). Another open source Ethernet based core is presented by Bertolotti and Hu in [2] which targets real-time applications by coupling a lightweight Internet Protocol core (lwIP) with a real-time operating system for low cost embedded systems.

Although these are viable solutions for data transfer within reconfigurable computing, the overhead of processing Internet Protocols leaves us curious to find another solution that does not require reliance on the Internet Protocol.

2.1.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is an apparatus where data is transferred directly from compute node to compute node over a channel without interrupting the operating system[17]. Data is placed into and taken from predefined application buffers. The idea is borrowed from Direct Memory Access (DMA). DMA is a mechanism that allows data within a compute node to be transferred from one memory location to another without interrupting the CPU. RDMA does not specify specific implementation requirements, but consists of RDMA Verbs, or functionality that must be provided for it to be considered RDMA [10]. RDMA provides performance advantages in a few different ways. True zero-copy data transfer with no intermediate buffering, decoupling of host processor from network yielding low CPU overhead, fixed memory resources (buffers) with no surprise messages, and unstructured, non-blocking data transfer with local completion are the main advantages to using RDMA[5].

RDMA does has a few drawbacks. According to Geoffray[8], due to the one-sided nature of RDMA the ability to agree on where data should be placed between the two communicating nodes is missing. The origin and destination buffers must be previously agreed upon and cannot be altered. Matching may cause unexpected messages if the sender and receiver have not yet agreed, but a message is transmitted. Handling unexpected messages then becomes a necessary function

of the implementation. However, the ability to mutually agree on buffer locations within endpoints allows for greater flexibility with respect to the order that messages are sent and received.

Among a sea of options, we seek a simple and lightweight solution that does not rely on an underlying protocol namely the Internet Protocol. Although widely adopted, underlying protocols can be cumbersome and heavy, even at times unnecessary, leading to higher latency and larger designs. While this work is independent of industry or government, the Department of Defense has interest in a non-bloated communication protocol for embedded systems. TCP does not offer this type of solution. Although this work focuses on the implementation of RDMA on FPGA's, the suggested protocol can be implemented in software or hardware on a CPU or GPU.

2.2 Datagram RDMA (DG-RDMA) Protocol Specification

The protocol of choice is the Datagram RDMA Protocol (DG-RDMA)[13]. This protocol defines transactions between compute nodes or endpoints - one source and one destination endpoint. RDMA Write transactions are used to transport data packaged as datagrams. Datagrams in the DG-RDMA protocol are consumed atomically, either entirely or not at all. RDMA Write transactions are one-sided data transfers that do not require explicit interaction between the nodes[5]. Datagrams are the fundamental unit within a packet-switched network. DG-RDMA requires a transmission layer that provides an unreliable datagram service with three features: no guarantee of delivery, no guarantee of order of delivery but does guarantee error-free delivery when datagrams are delivered. To be clear, the datagram service is not required to be unreliable, however DG-RDMA tolerates the unreliability. The protocol is datagram based but transmission layer agnostic. The transmission layer limits the size of the datagrams.

For each data payload that DG-RDMA transmits, exactly one meta data message describing the payload is transmitted. A message is defined as a unit of data (bytes) written to an endpoint or destination. A frame contains zero or more messages and is encapsulated by a frame header which specifies the source and destination, among other details listed in section 3.2. This specific implementation always has exactly one meta data and exactly one data payload included in a frame.

Both the meta data message and data payload message are described by a message header.

To ensure reliability, frames are acknowledged by an acknowledgment frame that is transmitted to the initiator of the message. After a timeout period, if the transmitted frame is not acknowledged, the initiator must resend the frame. The frame can either be held or recreated in the case of retransmission.

The DG-RDMA protocol is lighter weight and cheaper than implementations built on top of TCP/IP or UPD/IP. DG-RDMA relies on an unreliable transport protocol, but adds reliability without the incurred cost of data copy (zero-copy).

2.3 Design Input Language

Bluespec SystemVerilog (BSV) is a fully synthesizable, high-level language used to describe electronic systems. The combination of fully synthesizable and high level allows complete systems to be modeled on an FPGA instead of simulation only. The ability to design complex concurrent systems at a high level is valuable to a hardware designer, if lower level details are handled properly.

BSV is a formal specification language providing to main ideas for expressing a circuit: behavior and structure. The behavioral model is based on Atomic Rules and Interfaces, which are parallel by nature. BSV consists of modules that are tied together by interfaces. Modules in BSV are similar to modules in Verilog or VHDL. Registers, rules and methods make up a module. A register holds state and can be read and written. Rules are guarded atomic actions that either execute entirely free from interruption or do not execute at all. The guard is a statement that evaluates to a Boolean and if true, the rule fires and executes until completion. Rules can also either fire, or not, based on the implicit conditions of statements inside of the atomic rule. For example, consider the rule in Figure 2.1.

The guard for rule example is x % 2 == 0. The implicit condition is that the FIFO is not full. If the FIFO is full, it cannot be enqueued, so the rule cannot execute at all. Rules are not sequential with one another; they optimistically fire as frequently as they can. Another important aspect of

```
00: rule example(x \% 2 == 0);
01:   fifo.enq(x);
02:   x <= x + 2;
03: endrule;
```

Figure 2.1: BSV Example

rules is that they are private and local to the module. Rules are not synonymous to methods, which can be called from outside the module. Methods are also guarded atomic actions however they can be invoked with arguments and can return a value. Interfaces are composed of methods.

The notion of atomic rules and interfaces is powerful when constructing complex concurrent systems. The designer is freed from the constrictive idea of the clock, although BSV does produce synchronous circuits. Everything is considered in rule steps[11][4]. Rule step order is simpler to reason about than time step order especially when paired with the notion of groups of atomic actions.

2.4 Board and Tool Selection

Xilinx 7 Series FPGAs are currently the latest technology available to the FPGA community. The Kintex 7 (KC705) FPGA was chosen as a development and testing platform.

Vivado, the completely reworked design suite from Xilinx, is chosen for synthesis and place and route (PAR). The Analytical Place and Route technology offers four times faster implementation (PAR) time, twenty percent better device utilization and thirty-five percent lower power over other design suite offerings[18]. Not using Vivado for SAPR would make my design immediately outdated as it would not be able to work with the next generation 20 nm technology.

An interesting fact to note is that while Xilinx parts and tools have been chosen for implementation in this case, DG/RDMA does not require Xilinx parts. The bulk of the implementation is vendor agnostic and can be ported to any FPGA. Figure 2.2 shows the hierarchy of vendor-specific/vendor-agnostic modules. The MAC and interface to the Marvell Alaska PHY must align with Xilinx parts, but the other modules can be used in any design for an FPGA. The top level of

the design for the FPGA specifies the interfaces that will be utilized on the FPGA. Because these are specific to the KC705, this part of the design is also Xilinx specific. However, it is very simple to change the top level module to reflect the FPGA of choice.

There is no reason why this IP core cannot travel to other FPGA boards from other vendors. The IP core is RTL and does not require any Xilinx IP cores or tools. The IP core stands alone. A GMAC core is needed which is board specific by nature for this IP core to be compatible with FPGAs from other vendors.

2.5 Implementation of DG/RDMA

An additive and incremental bottom up approach is taken to implement DG/RDMA in hardware on an FPGA. This approach is taken with great care as to not lose top down clarity and direction for implementation. A long-term schedule was made with many version releases along the way.

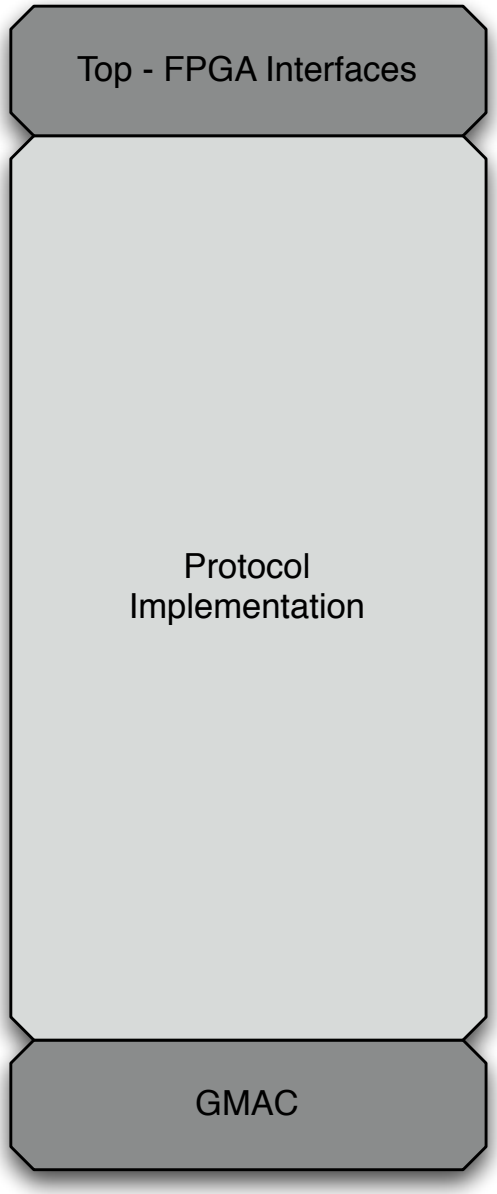

Figure 2.2: RDMA IP Core is Xilinx Agnostic

Chapter 3

System Design and Implementation

3.1 Top Level Design

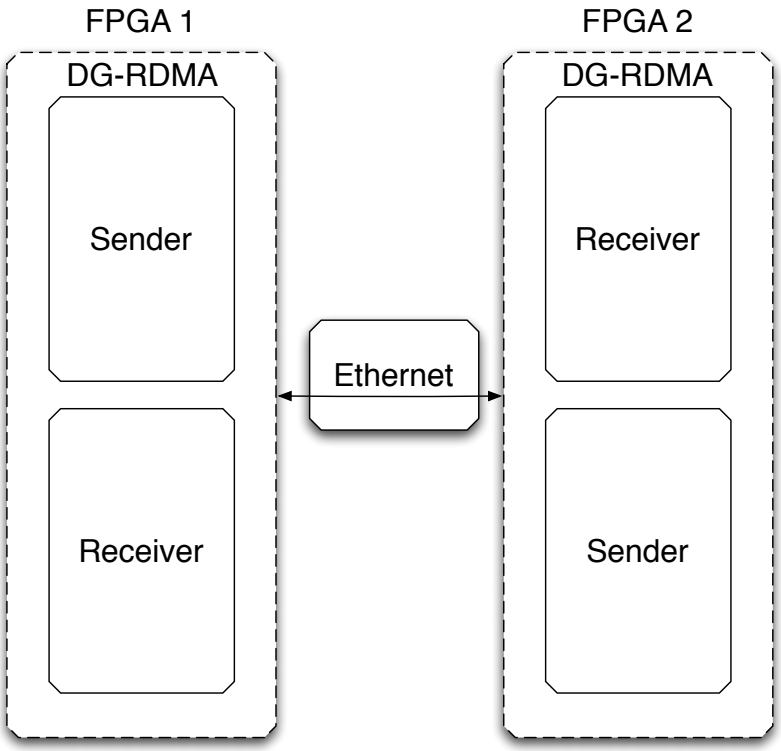

Figure 3.1: Basic System Design

Figure 3.1 shows a very basic description of the implemented system. DG-RDMA has the capability to send and receive messages, making it full duplex. Full duplex is an important feature allowing all compute nodes utilizing the DG-RDMA protocol to both send and receive messages. If the user wishes to only have half duplex transmission, that can be accommodated by making a few changes to the source code in the top level module.

The modules labeled as Sender/Receiver in FPGA 1 and FPGA 2 are functionally identical. The only difference in the Sender and Receiver modules implemented in different compute nodes

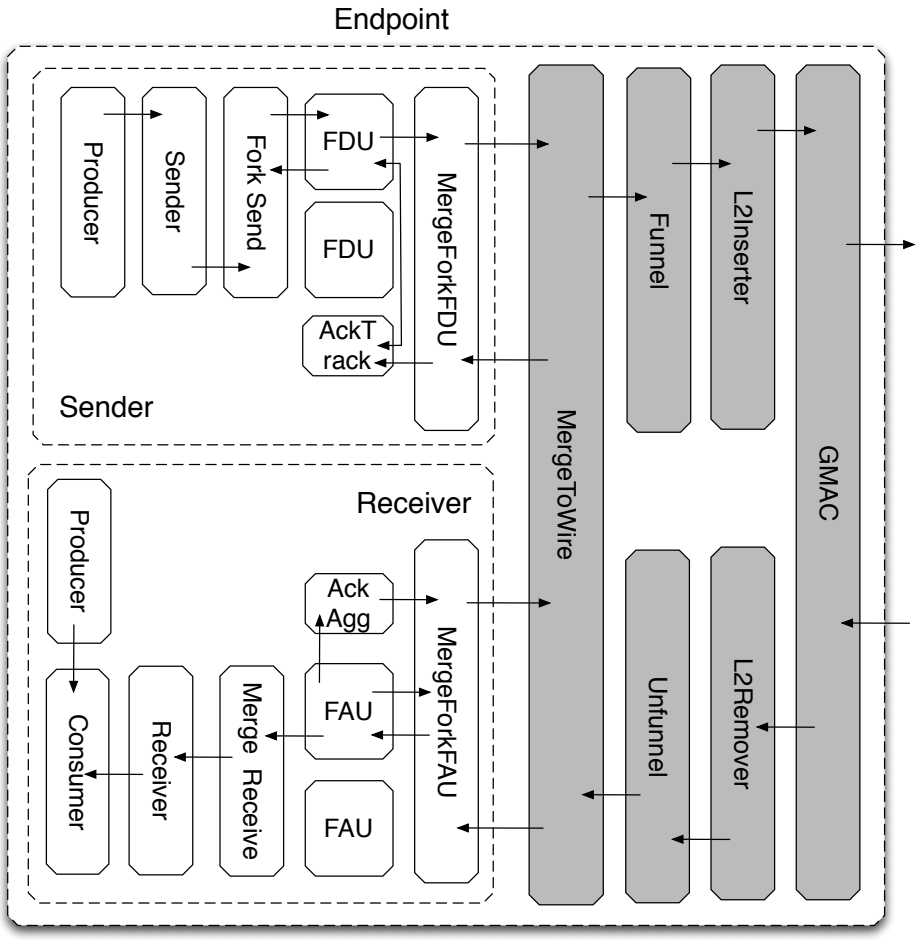

Figure 3.2: Block Diagram of an Endpoint

is the L2 Inserter/Remover. The L2 Inserter prepends the correct source and destination MAC addresses and ether type. The L2 remover checks the L2 header of an incoming packet to make certain that the packet is meant for this particular compute node.

The interfaces between the shown modules is a stream of bytes across an Ethernet link.

The Sending and Receiving modules are further described in Figure 3.2. The components that make up Sending and Receiving modules are outlined. The grayed modules are shared by both Sending and Receiving modules. Table 3.1 describes each of the modules, whether it is a function of the Sending or Receiving module and the basic function of the component. The functionality of each component is described in detail in the following subsections.

The interfaces between most of the modules is a FIFO of user defined type HexBDG. HexBDG

System Modules		
Module	**Function Of**	**Function**
Producer	Send	Generate/Fetch Payload
Consumer	Receive	Check payload for error
Sender	Send	Generate headers, forms frame
Receiver	Receive	Remove headers
Frame Departure Unit	Send	Forwards frame, hold frame until Acknowledgment received
Frame Arrival Unit	Receive	Accepts frame, signals to generate Acknowledgment
Merge Departure	Send	Merge multiple FDU output into one data stream
1Fork Arrival	Receive	Fork input stream to multiple FAU
Acknowledgment Tracker	Send	Receives Acknowledgment frames, informs FDU of frame ackd
Acknowledgment Aggregator	Receive	Generates Acknowledgment frames
Funnel (H2Q)	Both	Funnels 16 byte stream to 4 byte stream
Unfunnel (Q2H)	Both	Converts 4 byte stream to 16 byte stream
Fork Sender	Send	Assign frame to FDU
Merge Receiver	Receive	Multiplex frame from FAU to Receiver
Merge to Wire	Both	Multiplexes traffic to/from MAC
L2 Header Inserter	Both	Inserts the L2 header onto frame
L2 Header Remover	Both	Removes L2 header from frame
Quad Byte GMAC	Both	Initializes MAC and funnels 16 byte stream to 4 byte stream
GMAC	Both	Interfaces with Marvell PHY chip at Ethernet port

Table 3.1: Description of System Modules

```
00: typedef Vector#(16,Bit#(8)) HexByte;
```

Figure 3.3: HexByte Type Definition

```
00: typedef struct {
01:   HexByte data;   // 16B of data, Little Endian packed
02:   UInt#(5) nbVal; // Number of Bytes 0-16 that are valid
03:   Bool isEOP;     // True if this is the end of packet
04: } HexBDG deriving (Bits, Eq);
```

Figure 3.4: HexBDG Type Definition

is a data type that was designed to communicate information about the stream of bytes that is transferred from one module to the next. HexBDG consists of 16 byte data, a 5 bit unsigned integer that holds the number of bytes that are valid in the 16 bytes of data and a Boolean that is true only if it is the final segment of a frame.

HexByte is simply a vector of 16 bytes represented as a Little Endian value. Figure 3.3 shows the definition of a HexByte. HexByte is used in the HexBDG structure as shown in Figure 3.4.

Knowing the number of bytes in a 16 byte segment that are valid allows for frames that are not 16 byte aligned. This is important because any application that may use the DG-RDMA protocol may have an arbitrary number of bytes to transmit, not guaranteeing that a frame will consist of an integer number of full HexBytes.

The Consumer and Producer modules have FIFO interfaces of type MLMesg. MLMesg is a tagged union. A tagged union is a composite type in which something of type MLMesg is either of type MLMeta representing meta data or HexByte representing data as shown in Figure 3.5.

The MLMeta type definition is shown in Figure 3.6. Each data payload is described by meta data. The meta data describes the length of the message payload and an op code. The op code

```
00: typedef union tagged {
01:   MLMeta Meta;
02:   HexByte Data;
03: } MLMesg deriving (Bits, Eq);
```

Figure 3.5: MLMesg Type Definition

```
00: typedef struct {
01:   UInt#(32) length;   // Message Length in Bytes
02:   Bit#(8)   opcode;   // Message Op code
03: } MLMeta deriving (Bits, Eq);
```

Figure 3.6: MLMeta Type Definition

describes the type of operation the payload should be used for (e.g. read, write, ack, etc).

The Sending and Receiving modules are described in sections 3.2 and 3.3.

3.2 Sender

The sending module in an endpoint is responsible for wrapping a payload with DG-RDMA headers and forwarding it to the shared modules (GMAC and supporting modules) for transmission. The initiating endpoint is also responsible for accepting and handling the acknowledgment frames that correspond to the frames in flight. Each of the components/modules shown in the Sender module of Figure 3.2 are described in detail including interfaces, specific function and, in some cases, logical operation. Figure 3.7 shows the flow of an outgoing message followed by a step-by-step description of each modules action.

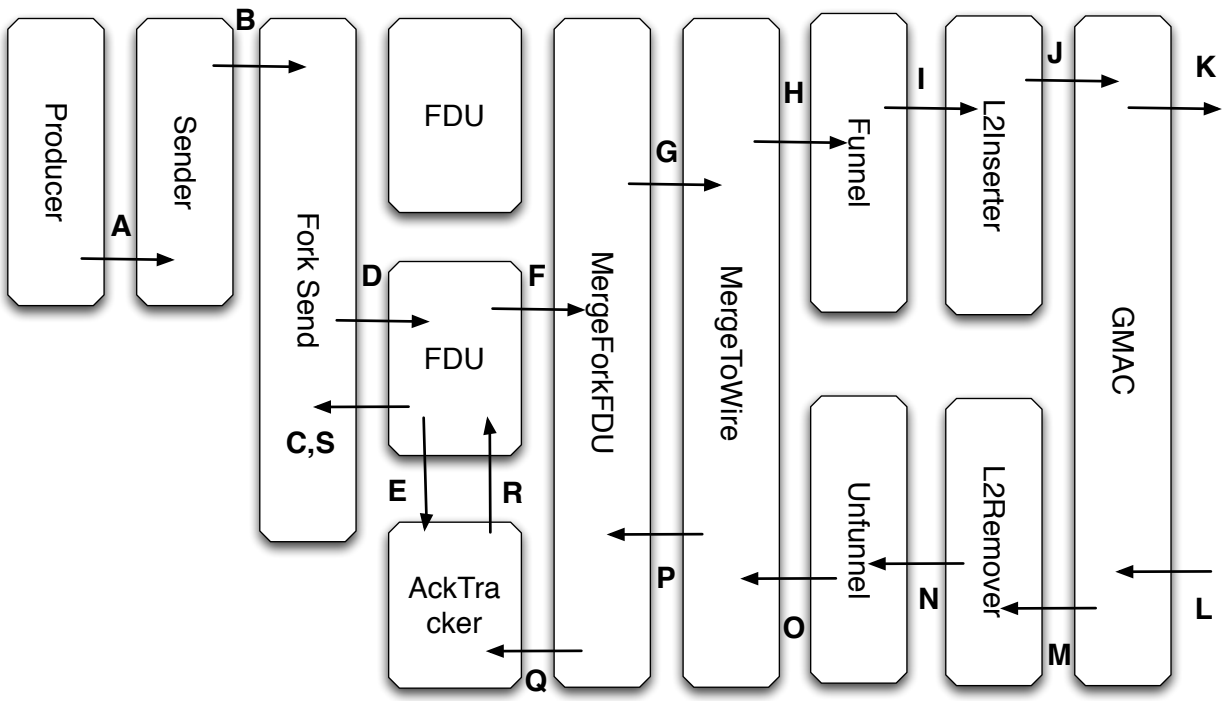

Figure 3.7: A day in the life of an outgoing message

A: Producer generates Payload

B: Payload is encapsulated with message and frame headers forming a frame

C,S: FDU signals that it is free and available to accept a frame

D: Frame is passed into FDU

E: AckTracker is notified of frame ID of Frame in flight

F,G: Frame is forwarded and unaltered

H: Frame is converted from 16 byte wide stream to 4 byte wide stream

I: L2 header is prepended to the frame forming a packet

J: Packet is passed to the GMAC

K: Packet exits through TX resolution layer to PHY

Wait for Ack packet...

L: Ack packet enters through RX resolution layer

M: GMAC gives incoming packet to L2 remover

N: Destination MAC and EtherType of L2 header is checked and removed, yielding a frame

```
00: UInt#(32)   length,
01: LengthMode  lMode,
02: UInt#(32)   minL,
03: UInt#(32)   maxL,
04: DataMode    dMode,
05: Bit#(8)     nukeVal
```

Figure 3.8: Producer Parameters

O: Frame is converted from 4 byte wide stream to 16 byte wide stream

P: Destination ID of Frame Header is checked, frame sent to either sender or receiver (sender in this case because it is an Ack)

Q: ACKCount in Frame Header is checked, if asserted it is sent to AckTracker

R: AckTracker informs FDU that Ack for specified Frame ID has been received

Producer

The function of the producer to create data payload that will be transferred to the Receiver. The typical use case would not utilize the producer, but would read data from memory somewhere in the compute node. In this implementation, it is not a direct goal to interface with the memory subsystem of any particular FPGA, but rather to provide the framework of the DG-RDMA protocol that can be adapted to a particular system. Regardless of where the payload data originates, it is passed on to the Sender for header encapsulation.

The producer has some features that were used to fully test and develop the machinery of DG-RDMA. As parameters passed at instantiation, length, length mode, minimum length, maximum length, data mode and nuke value describe different scenarios which must all be supported by the implementation.

The length mode is used to describe the scheme for determining how long in bytes a payload and each subsequent payload should be. There are three length modes: constant, incremental and random. The constant mode utilizes an additional parameter called length which defines a con-

stant payload length that will be used for each payload. The incremental mode utilizes additional parameters minimum and maximum length. The minimum length defines the starting payload length. Similarly, the maximum length defines the terminating payload length. Each subsequent payload is one byte longer than the last until the maximum length is reached. The random mode utilizes a Linear Feedback Shift Register (LFSR) to generate random values. An LFSR register is a shift register with an input bit that is a linear function of the current state. An LFSR can be used to generate a deterministic and long sequence of seemingly random numbers even though the sequence of generated numbers will recur given the same seed value. The random value can be constrained using the additional parameters of minimum and maximum length in the case that constrained testing is needed.

The data mode parameter is used to describe what type of pattern the payload will consist of. There are three data modes: zero origin, incremental origin and rolling count. Zero origin mode creates each payload beginning with the first byte as zero and increasing each following byte by 1. A zero origin data pattern of length 10 looks like this:

Payload 1: 00010203040506070809

Payload 2: 00010203040506070809

The incremental origin mode creates the first payload beginning with a value of 0 and increases the origin of each subsequent payload by one. Incremental origin payload of length 10 looks like this:

Payload 1: 00010203040506070809

Payload 2: 01020304050607080910

The rolling count mode begins payload creation just as the other two modes do, with the first byte at 0 and increasing by one each time. However, each subsequent payload picks up where the last left off in the incremental sequence. Rolling count payloads of length 10 look like this:

Payload 1: 00010203040506070809

Payload 2: 10111213141516171819

There is one more parameter to discuss. Nuke value is a value provided by the user to define what

numerical value should be used to represent invalid data. This is utilized in the case that number of bytes valid in a HexBDG is less than 16. An easily recognized value is used to distinguish each of the above described payload generation schemes from invalid bytes. Using something easily recognizable helps in debugging and to be certain that no data is being altered from one module to the next as data flows through the Sender and Receiver. As an example, a HexBDG with number of bytes valid set to 5 might look something like this:

0001020304AAAAAAAAAAAAAAAAAAAA

It is easy to see that there is a clear pattern for the first 5 bytes followed by 11 bytes of the same value AA. While the payload is generated, the meta data describing the payload is also created. The meta data precedes the payload when passed on to the Sender for header generation. There is always exactly one meta data for each payload. The meta data is 8 bytes long. Zero length messages are also supported which may be used to signal an event to the receiving endpoint. The interface into the Producer is the arguments discussed above. The interface out of the Producer is a FIFO of type MLMesg.

Sender

The Sender has two main purposes: to generate and prepend message and frame headers and to transform the meta data and payload from type MLMesg to type HexBDG. HexBDG is the main currency used in this implementation and is what is translated into bytes on the Ethernet wire.

The datagram consists of one message. A message is defined as meta data and payload data. Each message is encapsulated as a frame and labeled with a frame header. The meta data and payload data both are also encapsulated with a header, namely, the message header. Both the frame and message headers have fields which describe important details about the payload itself. Table 3.2 shows the frame header fields and a description of each.

The Destination ID is a globally unique identifier associated with an endpoint or compute node. Destination ID's are assigned sequentially beginning with 1 in a cluster of compute nodes and are used as an index value.

Frame Header Fields		
Field Name	**Description**	**Size**
Destination ID	ID of destination endpoint	2 octets
Source ID	ID of source endpoint	2 octets
Frame ID	Rolling sequence of frame number	2 octets
ACKStart	Starting ID of ACK sequence	2 octets
ACKCount	Total number of ACKS	1 octet
Flags		1 octet

Table 3.2: Frame Header Field Descriptions

Frame ID's are increasing values that roll over. The initiating endpoint is responsible for maintaining a unique sequence of frame ID values for each destination.

The upper seven bits of the Flags bytes is reserved and must be set to zero. The remaining, least significant bit, is set if the frame contains at least one message. If the bit is not set, there are no messages in the frame.

The frame header itself may contain acknowledgments of received frames. When the ACKCount field is greater than zero, the ACKStart value is used to determine which frames are being acknowledged.

Example ACK payload for 1 frame:

ACKCount - 1

ACKStart - 6577

ACK 1 frame with ID 6577

Example ACK payload for 3 frames:

ACKCount - 3

ACKStart - oxffffffff

ACK 3 frames with ID 0xffffffff to rolled over IDs 0 and 1

Table 3.3 shows the message header fields with a short description and size of each. The protocol specifies that it is the responsibility of the sending endpoint to pad out the message data in order to ensure that the next header is aligned on an 8 byte boundary. Because the data type used is 16 bytes wide, this implementation pads out the message data to a 16 byte boundary in order to simplify

Message Header Fields		
Field Name	**Description**	**Size**
Transaction ID	Rolling ID that is scoped by source and destination IDs	4 octets
Completion Address	Address in the destination end	4 octets
Completion Data Value	Completion value to be written	4 octets
Number of data messages in this transaction	Can be zero	2 octets
Fields above here are the same in all messages in the same transaction		
Message sequence	TO BE REMOVED	2 octets
Data Address	Endpoint address where message data will be written	4 octets
Data Length	Length of message data following this header in bytes	2 octets
Message Type	TO BE REMOVED	1 octet
Trailing Message	Boolean, non-zero if there is another message after this one	1 octet

Table 3.3: Message Header Field Descriptions

processing logic.

The input FIFO to the module is of type MLMesg which consists of either MLMeta or MLData. An outgoing message will always enter the Sender MLMeta type first. Upon enqueue of the input FIFO, the meta data (MLMeta) triggers creation of the message header for the meta data. Image headers shows the header structure for a datagram.

Figure 3.9 shows the frame composition in detail. As the MLMesg type is enqueued into the input FIFO of the module, creation of the frame header is triggered. A state variable is used to coordinate which rule cloud is fired when, ensuring proper frame composition.

First, Generate Frame Header rule is fired which will use the meta data to populate the frame header fields and enqueue the frame header into the ByteShifter. The ByteShifter is a configurable module which is in the business of converting segments of a byte stream into convenient 16 byte stream. The module can be configured to produce byte segments of almost any length. The ByteShifter was chosen in this case because the frame is created in steps that produce different frame segments of which some are both greater than and less than our 16 byte data type. This allows the headers to be created and shifted out in 16 byte chunks ensuring that the output FIFO

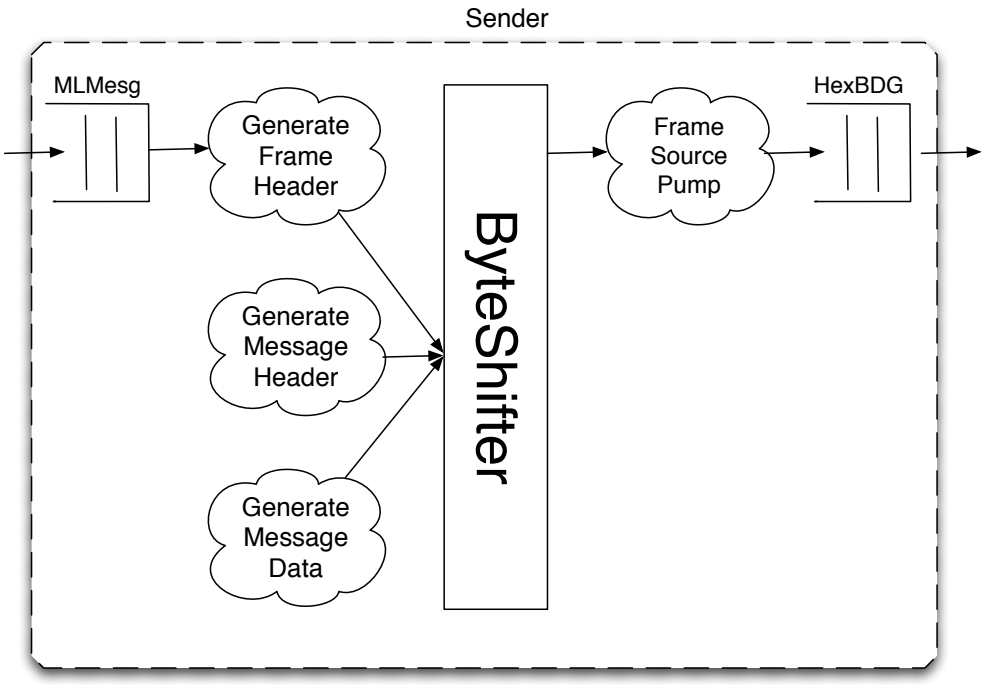

Figure 3.9: Sender Logic

will be enqueued with the proper number of bytes.

Once the frame header is shifted into the ByteShifter, the message header that describes the meta data is created and enqueued into the ByteShifter in the Generate Message Header rule. The fields of the message header are populated according to some details in the meta data. However, some fields are arbitrarily assigned in the current implementation. [[more detail about which fields in header specifically]] In some cases, the user may want to define certain fields of the message header such as sender and receiver ID. Such fields could be passed in as parameters of the module or hard coded into the implementation depending on the needs of the user. Either of these changes can be easily made in the source code.

Next the meta data is forwarded directly to the ByteShifter from the input FIFO. The state variable to set to return to the Generate Message Header rule in order to generate the message header that will describe the payload. As shown in Table 3.3 above, some of the message header fields will be the same for the meta data and payload data.

In the final stage of frame generation, the Generate Message Data rule is fired. This allows

the data payload to be forwarded to the ByteShifter from the input FIFO. To determine the end of the packet, the payload length is collected and stored from the meta data. The number of bytes of the payload are counted as they are enqueued into the ByteShifter. When the end of payload is reached, the state variable is set back to the initial state of the module, Generate Frame Header, which awaits arrival of another message at the input FIFO.

Simultaneously, as 16 bytes of the frame become available at the output of the ByteShifter, they are converted to type HexBDG and sent out of the Sender module through a FIFO of HexBDG.

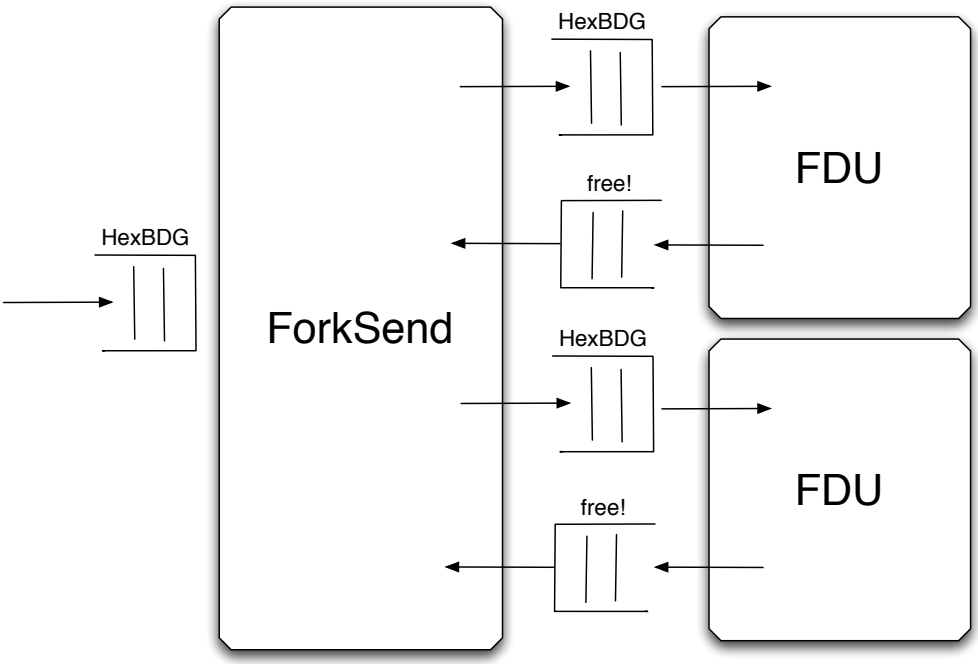

Figure 3.10: Fork Send Module

ForkSend

After leaving the Sender module, the newly created frame consisting of HexBDGs enters the ForkSend module. The main purpose of this module is to assign the frame to a Frame Departure Unit. In this implementation, there are two Frame Departure Units. Each Frame Departure Unit will notify the ForkSend module when it is free and available to accept a new frame. The ForkSend module will pass a frame along to the available Frame Departure Unit and wait for another frame and another free Frame Departure Unit. Figure 3.10 shows a diagram of the interfaces of ForkSend module.

Frame Departure Unit

The Frame Departure Unit (FDU) stages a frame for departure and holds onto the frame until it receives an acknowledgment from the Acknowledgment Tracker. The frame is held in a BRAM so that it can be retransmitted in the case that a cycle counter reaches a defined timeout value before the frame acknowledgment is received.

When a frame enters the FDU, the frame header is examined and the frame ID is saved in a register and forwarded to the Acknowledgment Tracker. The Acknowledgment Tracker holds on to the frame ID until an acknowledgment for that frame is received. The ID of the acknowledged frame is sent to the FDU from the Acknowledgment Tracker and the frame is released from the FDU, allowing the FDU to signal to the ForkSend module that it can accept a new frame.

Once the frame leaves the FDU, it is passed along to the MergeForkFDU module. The FDU will then signal to the ForkSend module that it is available to accept a new frame. In this implementation, there are two FDUs in an endpoint. This allows the bandwidth to be increased so that the acknowledgment of the frame being held by a FDU does not do as much damage to the low latency nature of the design. There will be two frames in flight given that there are frames available to be sent. The number of FDUs could be increased to further diminish the affect of the acknowledgment on latency/bandwidth. Figure 3.11 shows the interfaces between the FDU, ForkSend, AckTracker and MergeForkFDU.

Acknowledgment Tracker

The purpose of the Acknowledgment Tracker is to essentially keep track of frames in flight and acknowledgments. The frame IDs of all of the frames in flight are passed to the Acknowledgment Tracker through a FIFO from each FDU. The Acknowledgment Tracker is a singleton among what could be a variable number of FDUs. In this particular implementation, there are only two FDUs. As mentioned in 3.2, the number of FDUs was an implementation choice and can be changed. Once the acknowledgment frame is received, the frame and message headers are removed, the frame ID is extracted and the proper FDU is notified that the frame it is holding has been acknowledged. Figure 3.11 shows the interfaces of the Acknowledgment Tracker and the surrounding modules.

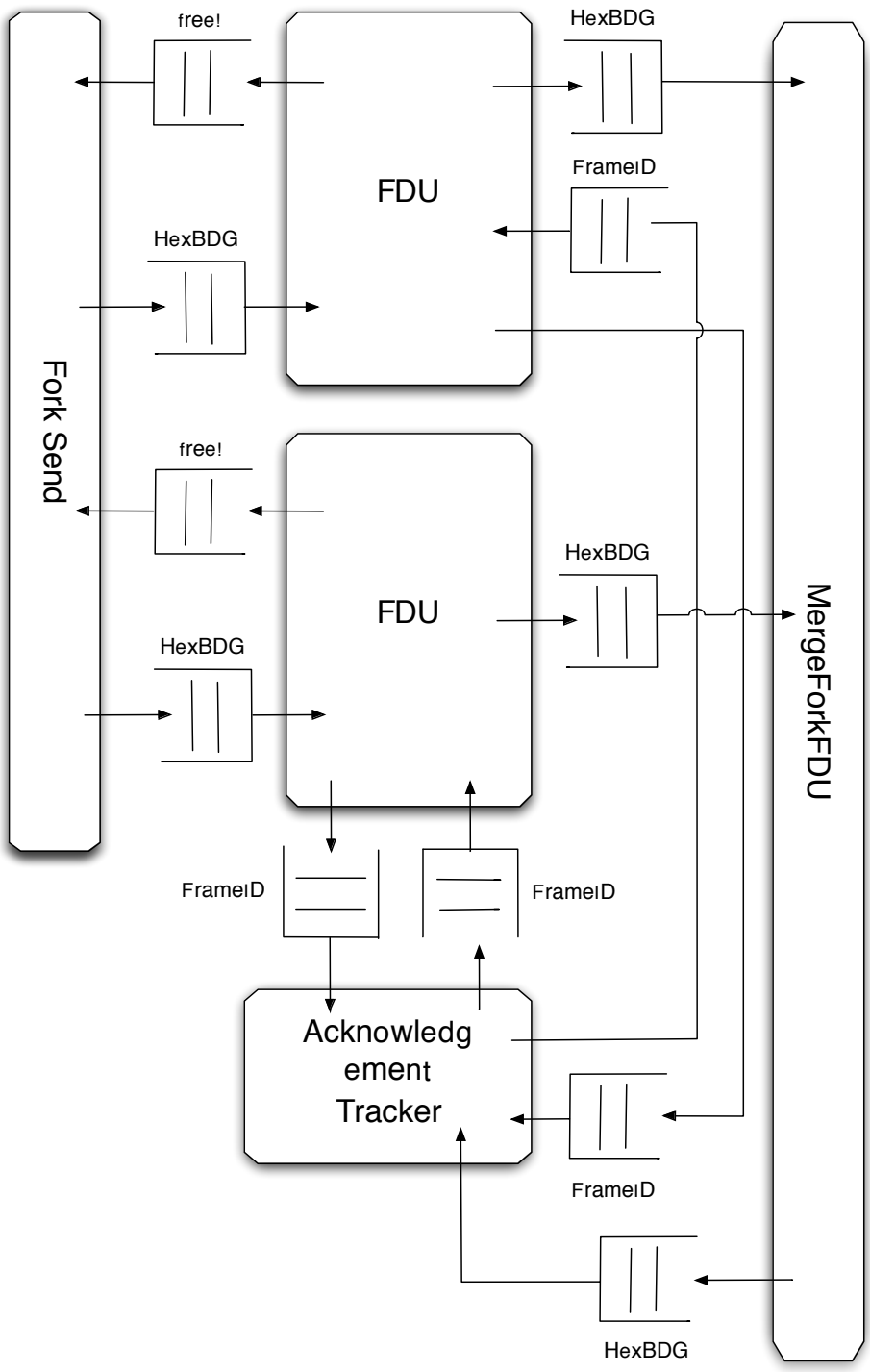

Figure 3.11: Acknowledgment Tracker and surrounding modules

MergeForkFDU

The MergeForkFDU module is responsible for merging the outgoing messages from the FDUs. There cannot be multiple outgoing byte streams from the sending endpoint. There are two FIFOs that interface with the next module in the flow, the Merge To Wire module. They are both of type HexBDG, one for incoming acknowledgment frame and one for outgoing datagram frames. Upon reception of an acknowledgment frame, this module forwards the acknowledgment frame to the Acknowledgment Tracker, which will notify the FDUs that a frame has been received by the other endpoint. The outgoing frames are merged from the two FDUs using a round robin scheme. FDU 1 gets priority at restart because FDU 1 will receive the first frame from the Sender module. One complete frame must leave the module at a time; interrupting any frame will cause the message to not be correct when received by the other endpoint. A block diagram of the module and its interfaces is shown in Figure 3.12.

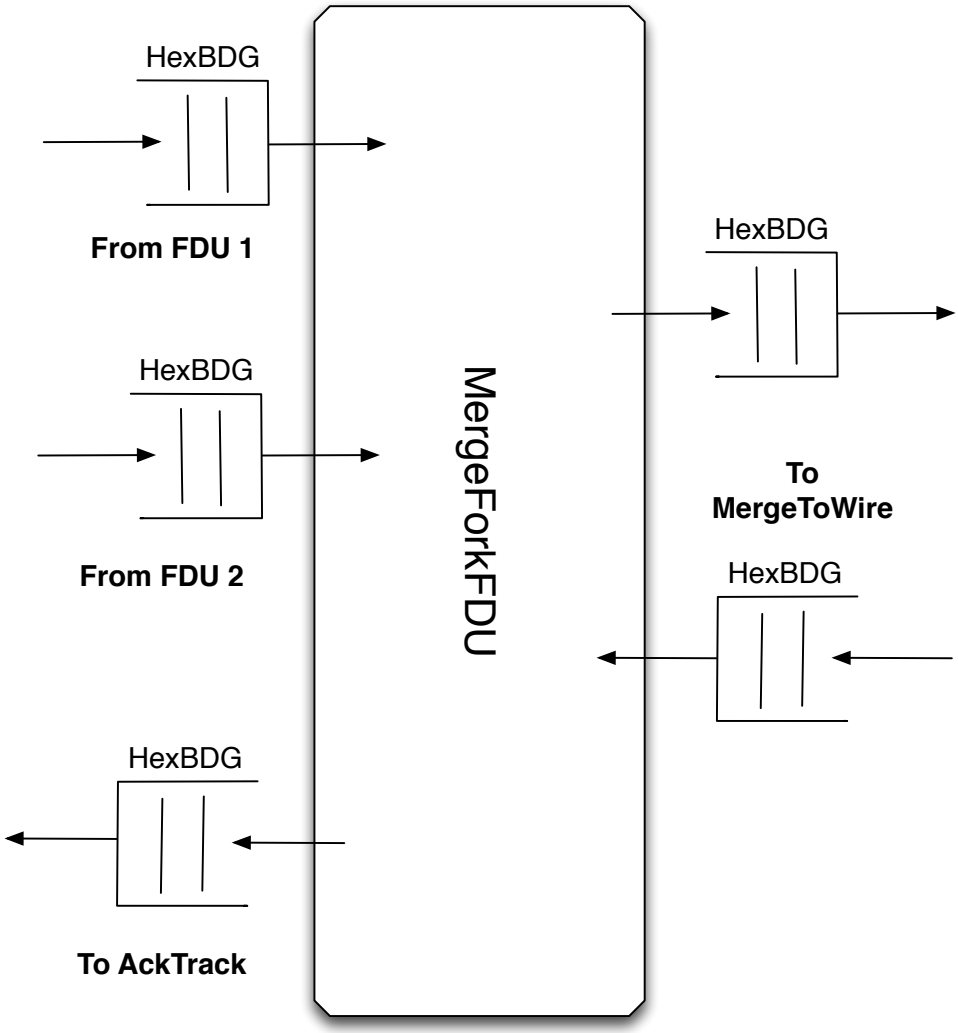

Figure 3.12: Merge Fork Departure Module

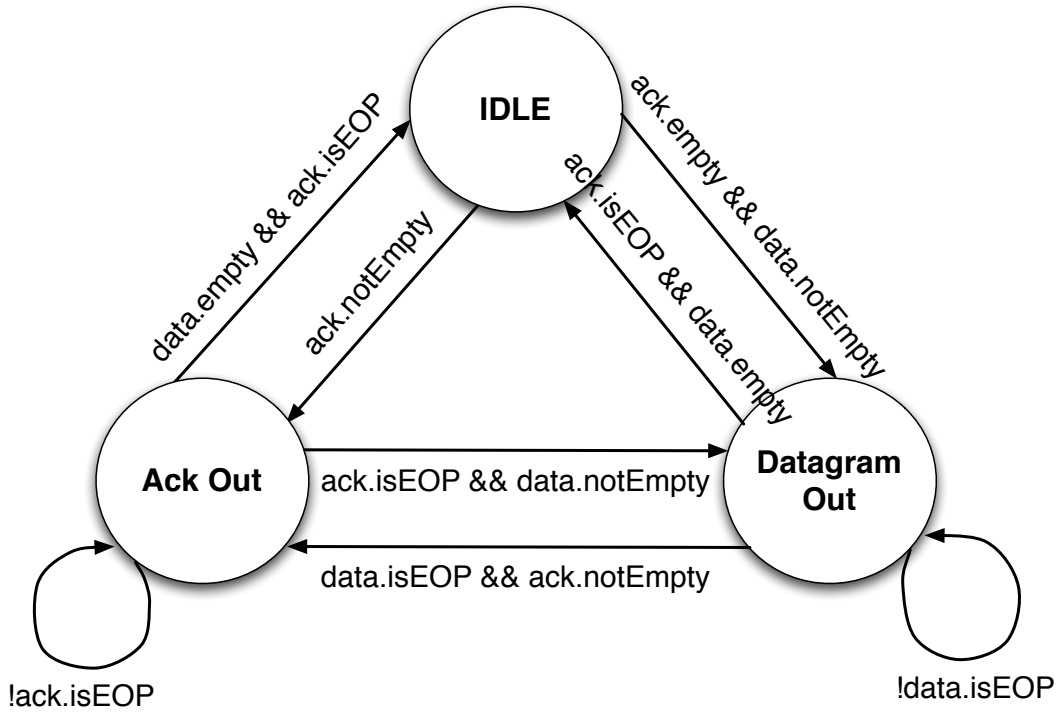

Figure 3.13: Arbitration Scheme for Transmission in Merge To Wire Module

Merge To Wire

MergeToWire module is a singleton in an endpoint. Both the sending and receiving circuitry take advantage of the services that MergeToWire offers. This modules uses an arbitration scheme to merge the outgoing frames from both the sending and receiving components of an endpoint. This was a little tricky, but a solution for all test cases was found. The Figure 3.13 shows the state transition graph of the arbitration scheme for outgoing frames with data payloads and outgoing frames with acknowledgments. A round robin arbitration scheme cannot be used because it is not a guarantee that there will be an outgoing datagram followed by an outgoing acknowledgment followed by another outgoing datagram and so on. It is also important, just as it is in the MergeForkFDU, that frames are sent out atomically. The interruption of stream of bytes of a frame will cause the protocol to break.

The incoming frames are arbitrated using a different scheme. When a frame is received from the GMAC and passed to the MergeToWire module, the frame header is examined to determine the

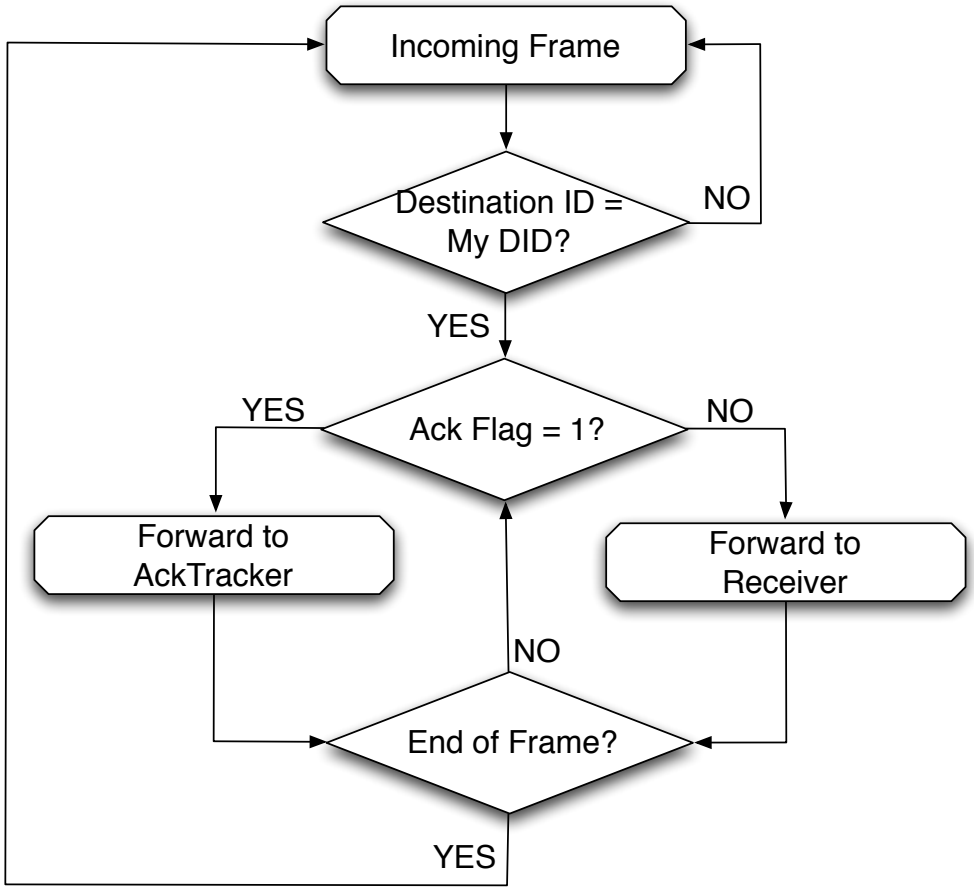

Figure 3.14: Decision Tree for Reception in Merge To Wire Module

destination ID for the frame. The ACKCount flag is also examined to determine if the frame is an acknowledgment or a datagram with a payload. If the frame is for this endpoint, the ACKCount flag is used to pass the byte stream to the either the receiving FAU if it is a datagram or the AckTracker if it is an acknowledgment. Figure 3.14 is a decision diagram for this module.

```
00: typedef union tagged {
01:  Bit#(8) ValidNotEOP;  // Any valid data cell so long as
                           // it is not the last
02:  Bit#(8) ValidEOP;     // A valid final data cell in a
                           // sequence indicates good EOP
03:  void    EmptyEOP;     // The end of a sequence has
                           // occurred, the last data was sent
                           // before indicates good EOP
04:  void    AbortEOP;     // The sequence has ended with an
                           // abort, all data and metadata
                           // from this packet is bad
05: } ABS deriving (Bits, Eq);
```

Figure 3.15: ABS Definition

```
00: typedef Vector#(4,ABS) QABS;
```

Figure 3.16: QABS Definition

Funnel

The funnel module converts the outgoing byte stream from type HexBDG to type QABS. QABS is another user-defined type that is a vector of 4 ABS's. The type definition for ABS is shown below. ABS is a tagged union meaning that each byte of data is tagged with one of the tags described below.

QABS takes 4 of these bytes tagged with a type. The most common type is ValidNotEOP. Exactly one byte in a datagram byte stream will be marked with ValidEOP. Any residual bytes in a QABS that are not valid data will be marked with EmptyEOP. This implementation does not utilize the AbortEOP type.

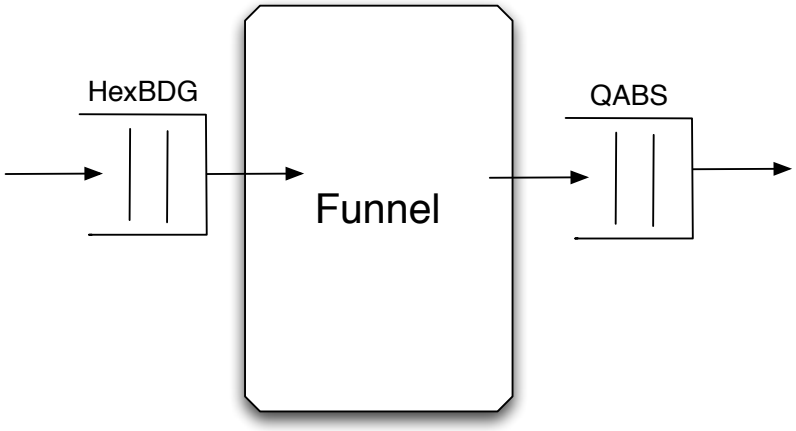

Figure 3.17: Funnel Module

The reason for converting the data from 16 bytes wide down to 4 bytes wide is to interface with the QBGMAC module easily. The QBGMAC module is one that had been previously written, so reuse of the module is ideal. The addition of the funnel component took a comparable amount of work that altering the QBGMAC module to interface with type HexBDG.

A block diagram of the Funnel module is shown in Figure 3.17.

L2Inserter

In order to communicate over a Layer 2 switch, an L2 header must be prepended to the frame on its way to the Gigabit Mac. When the first segment of a frame enters the L2Inserter module, the L2 header is created and sent out to the QBGMAC module. The entire frame is then forwarded to the QBGMAC. The L2 Inserter has two FIFO interfaces of type HexBDG, one for the incoming frame and one for the outgoing packet.

3.3 Receiver

The Receiving module in an endpoint responsible for accepting incoming frames from the shared modules (GMAC and supporting modules). Receiving module must also send acknowledgment frames to the initiating endpoint for received frames. Each of the modules/components in Figure 3.2 are described in detail including interfaces, specific function and, in some cases, logical operation. Figure 3.18 shows the data flow is described step by step in this figure. In general, the operations of the Receiving module in an endpoint are simpler than those of the Sending module in the endpoint.

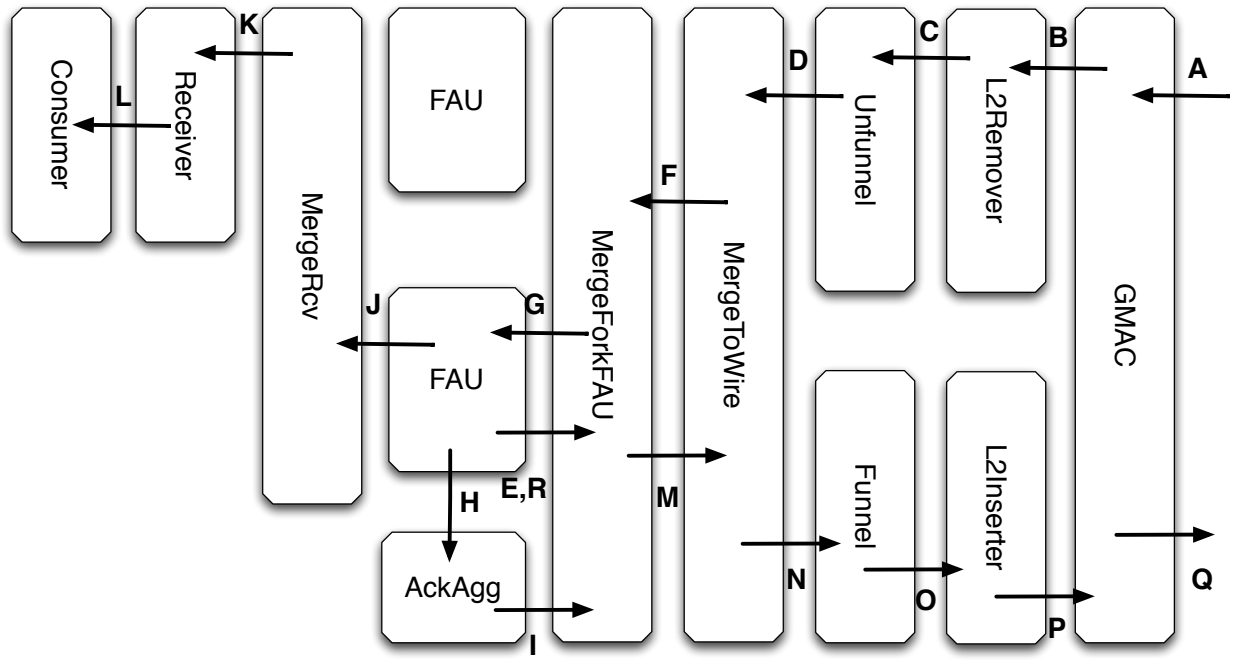

Figure 3.18: Receiver Data Flow Diagram

A: Packet enters through RX resolution layer

B: Packet is passed up through GMAC to L2 remover

C: Destination MAC and EtherType of L2 header is checked and removed, yielding a frame

D: Frame is converted from 4 byte wide stream to 16 byte wide stream

E,R: FAU signals that it is free and available to accept a frame

F: Destination ID of Frame Header is checked, frame sent to either sender or receiver (receiver in this case because it is a frame)

G,J : Frame is forwarded and unaltered

H: AckAggregator is notified of frame ID of received frame

I: Ack frame is generated and passed to MergeForkFAU

J: Frames from both FAU's are merged into one byte stream

K: Frame is passed to Receiver

L: Message and frame headers are stripped from frame yielding a payload

M*: Frame is forwarded and unaltered

N*: Packet is converted from 16 byte wide stream to 4 byte wide stream

O*: L2 header is prepended to the frame forming a packet

P*: Packet is passed to the GMAC

Q*: Packet exits through TX resolution layer to PHY

*Ack frame does not wait for frame payload to reach Consumer in order to be transmitted

L2Remover

When a packet comes in from the GMAC, the L2 header must be removed. The Destination MAC address is checked to see if the packet is for this node. The node is aware of its own MAC address. The EtherType is also checked to make sure that it is the DG-RDMA protocol. If the packet is for the receiving node, the L2 header is removed and the frame is forwarded to the Unfunnel module. If the packet is not for the receiving node, it is not passed further into the protocol stack. The L2 remover has two FIFO interfaces of type HexBDG, one for the incoming packet and one for the outgoing frame.

Unfunnel

The Unfunnel module is responsible for converting the incoming QABS byte stream to a HexBDG byte stream, essentially the opposite of the function of the funnel module. Section 3.2 describes the data types and why the conversion is necessary. Figure 3.19 shows the interfaces of the Unfunnel module.

MergeToWire

Refer to section 3.2. The MergeToWire component is a singleton in an endpoint.

MergeForkFAU

The MergeForkFAU module is responsible for waiting for assigning an incoming frame to an FAU. When an FAU becomes available for a new frame, the FAU signals to the MergeForkFAU that it is

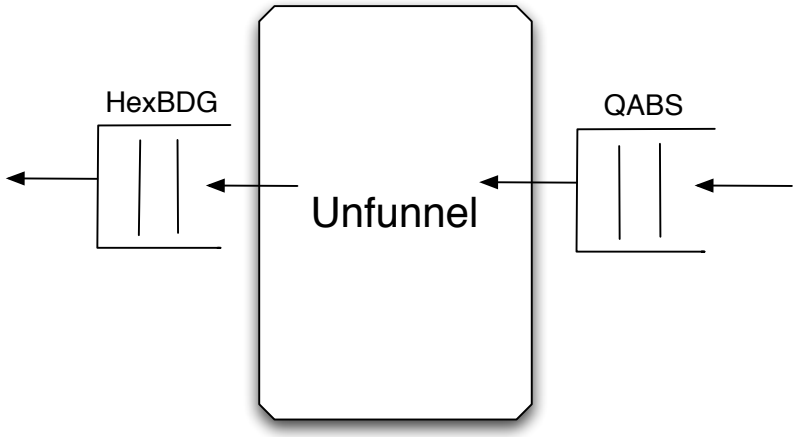

Figure 3.19: Unfunnel Module

free. MergeForkFAU then assigns incoming frame to the available FAU. There are two FAU's in an endpoint. After the frame is passed to the FAU, MergeForkFAU waits for another incoming frame and another signal from either of the FAU's to assign the frame to. The MergeForkFAU module also forwards outgoing Ack frames from the AckAggregator module to the MergeToWire module for transmission. The interfaces for the module are shown in Figure 3.20.

Frame Arrival Unit

The Frame Arrival Unit (FAU) holds the incoming frame and notifies the Acknowledgment Aggregator of the received frame's ID. When an FAU is available to accept a new frame, it signals the MergeForkFAU module. As the frame enters this module from the MergeForkFAU, the Frame ID is saved in a register and enqueued into a FIFO that is sent to the Acknowledgment Aggregator for Ack frame generation and transmission. The Acknowledgment Aggregator is described in section 3.3. BRAM in the FAU holds the frame until the Frame ID of the frame is sent to the Acknowledgment Aggregator and the frame is forwarded to the Merge Receive module. In this implementation, two FAUs are employed to increase overall throughput of the design. The number of FAU's can be increased to further increase the throughput of the design. Figure 3.21 shows the FAU and its surrounding modules.

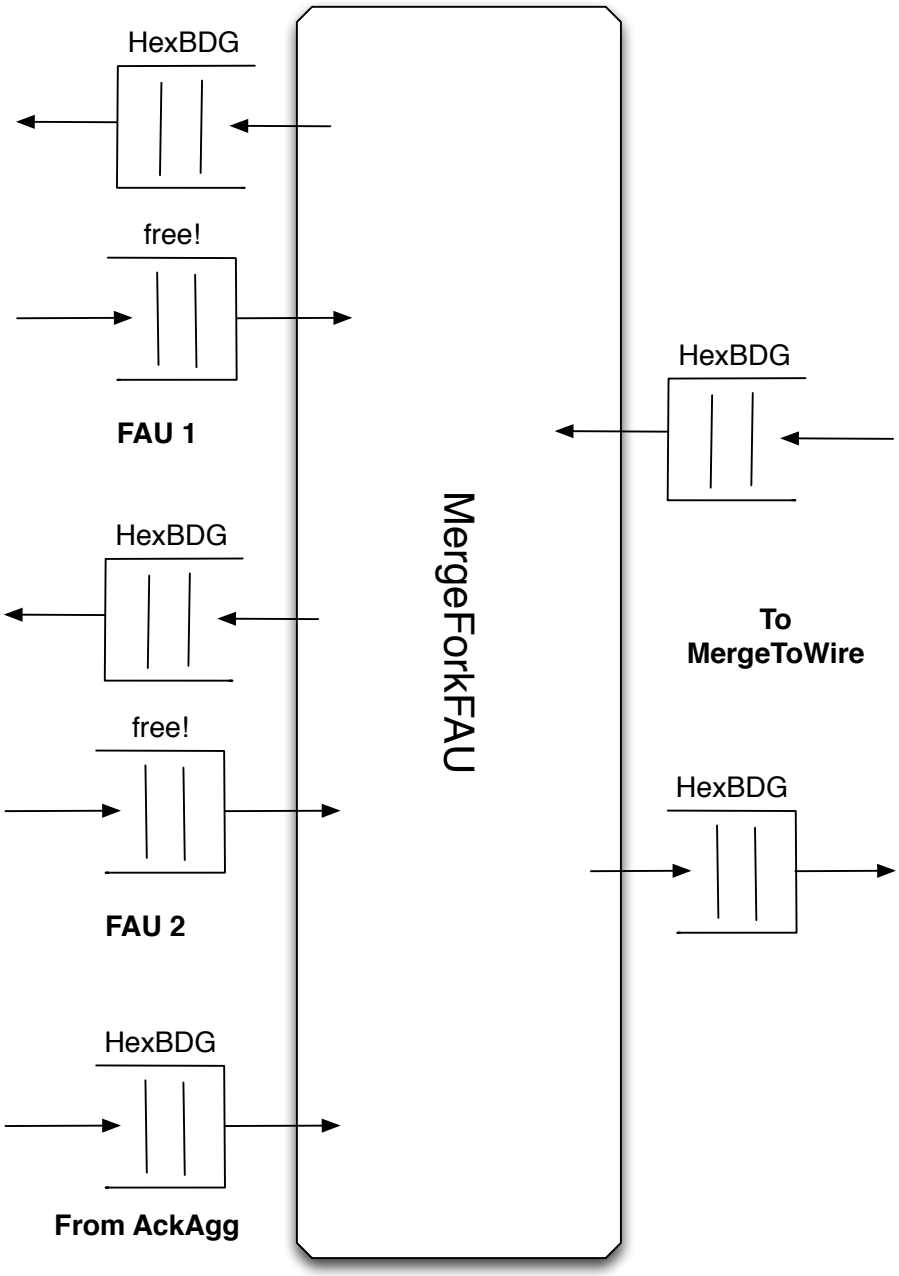

Figure 3.20: MergeForkFAU module

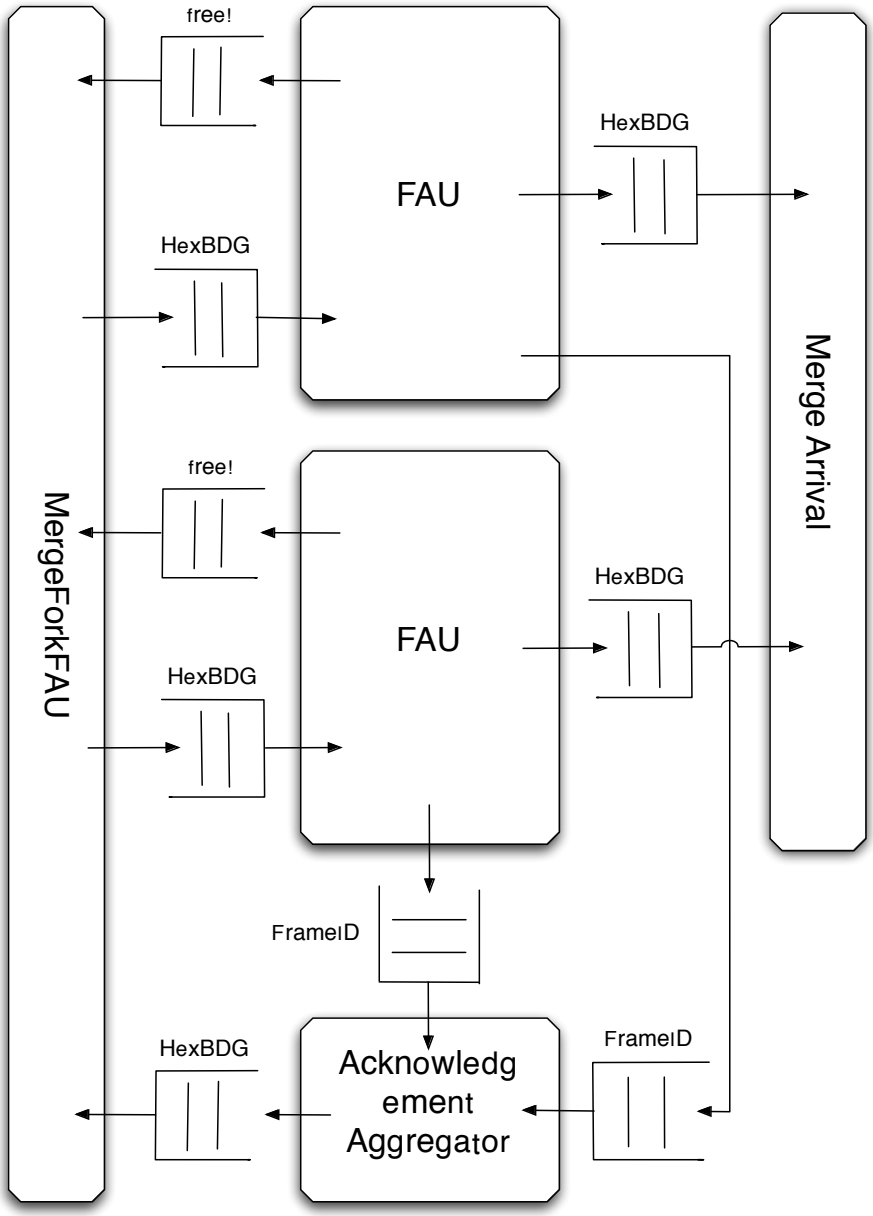

Figure 3.21: Acknowledgment Aggregator and surrounding modules

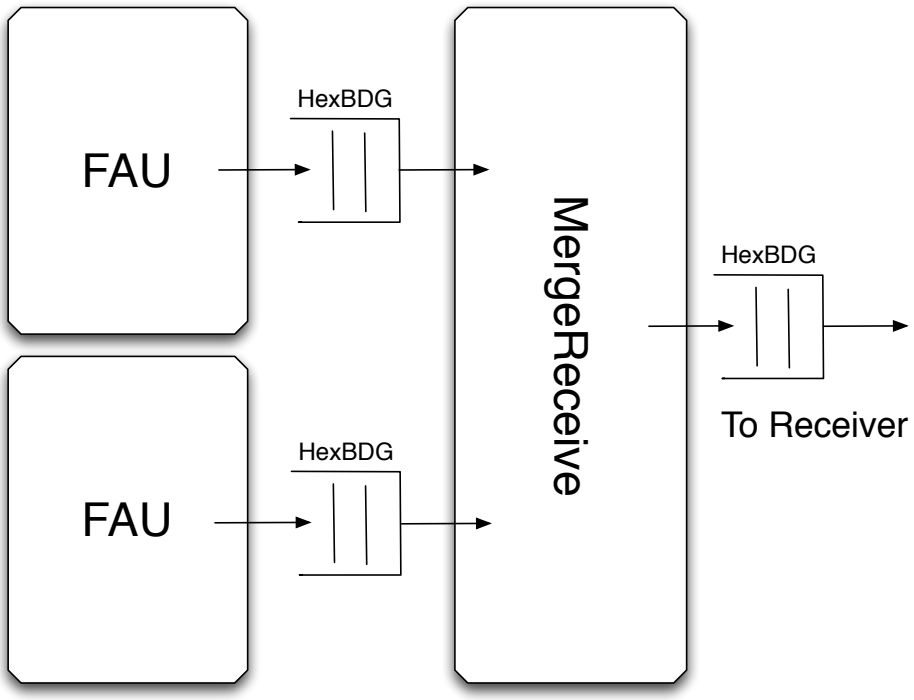

Figure 3.22: Merge Receive module

Acknowledgment Aggregator

The purpose of the Acknowledgment Aggregator is to generate an acknowledgment frame for all incoming datagrams as they arrive in their entirety. The two FAU's forward the frame ID of the frames they have received to this module. Upon reception of a frame ID, the acknowledgment frame is generated and sent to the MergeForkFAU module to be sent back to the initiating endpoint. This module also keeps a count of how many acknowledgments it has generated and transmitted. Figure 3.21 shows the interfaces with key surrounding modules.

MergeReceive

Frames enter the MergeReceive modules from either of the two FAUs and are forwarded to the Receiver unmodified. Frames are passed through atomically into the MergeReceive module and are multiplexed to the Receiver module. The input FIFOs and output FIFO of this module are of type HexBDG. Figure 3.22 shows a diagrams of the module and it's interfaces.

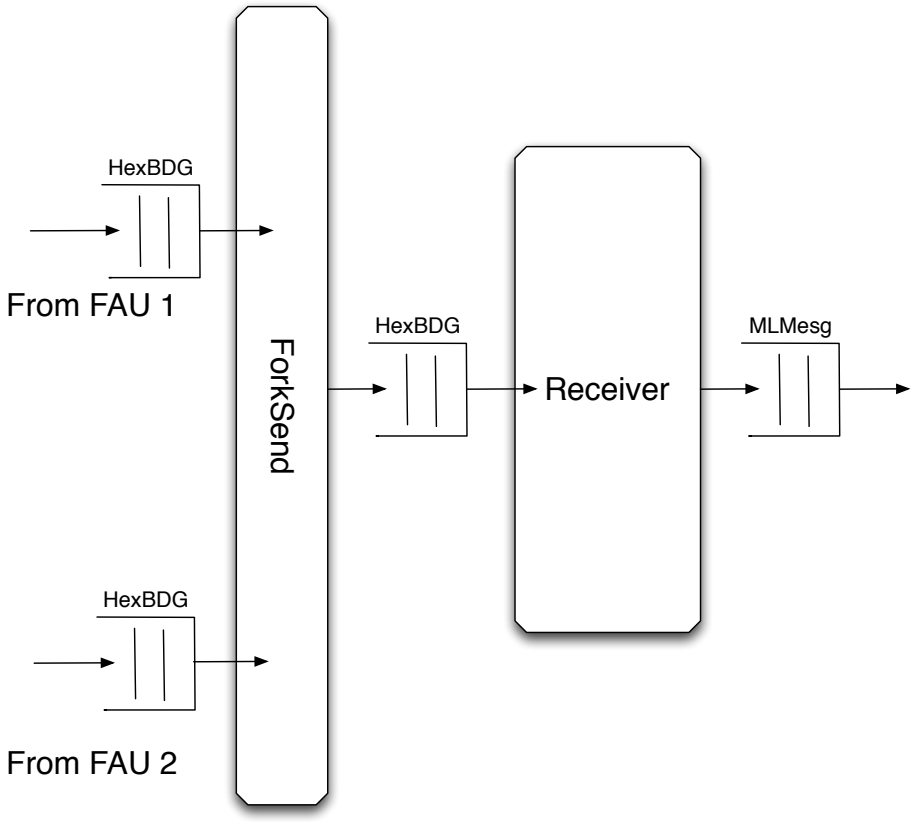

Figure 3.23: Receiver module and surrounding modules

Receiver

The Receiver module is responsible for removing the DG-RDMA protocol headers and forwarding the meta data and payload to the Consumer module. The frame enters the Receiver module through a FIFO of type HexBDG and exits as a message through a FIFO of type MLMesg. Figure 3.23 shows these interfaces. As the frame enters

Figure 3.24 is a logical flow diagram for frame decomposition in the Receiver. A state variable is used to control which logic cloud the frame will go through. The frame enters the module and is loaded into a Byte Shifter, similarly to the Byte Shifter in the section 3.2. The first 10 bytes are removed in the Remove Frame Header logic cloud. The next 24 bytes are removed from the Byte Shifter in the Remove Message Header logic cloud. Every frame contains meta data, which is the next 8 bytes that are removed from the Byte Shifter in the Forward Message Data logic cloud.

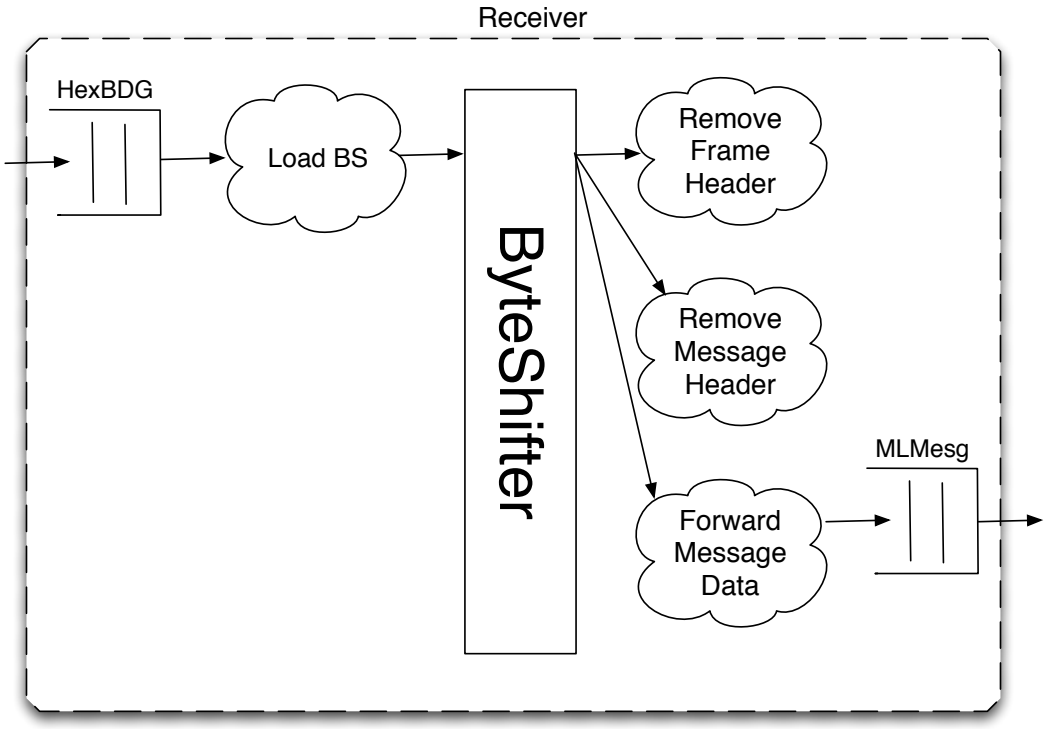

Figure 3.24: Receiver module data flow

Meta Data is forwarded on to the Consumer to be checked against the control for correctness. State then transitions back to the Remove Message Header logic cloud to strip away the message header corresponding to the payload data. To retrieve the payload, the only part of the frame left in the Byte Shifter, the Forward Message Data logic cloud removes the payload from the Byte Shifter end enqueues it in the outgoing MLMesg FIFO. State in the module is reset upon the end of the payload and awaits another frame.

Consumer

The interfaces of the Consumer module are shown in Figure 3.25. Two FIFO inputs of type MLMesg supply the Consumer module with both the expected and received meta data and payload. The expected meta data and payload is sourced from an identical Producer module and is used as a control to compare to the meta data and payload that has been through the complete DG-RDMA protocol stack. For each correct meta data and payload comparison, a count is increased

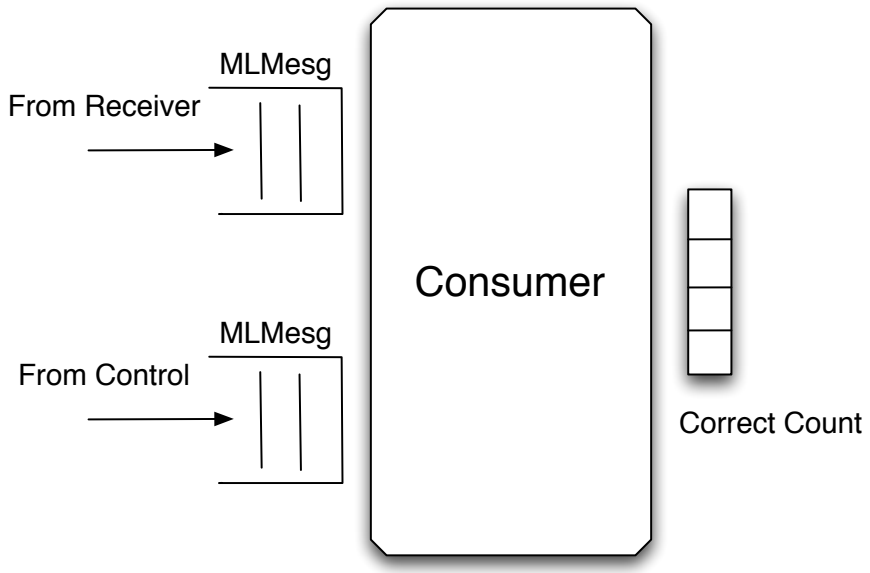

Figure 3.25: Consumer Module

by one. The value of the count is available to be read from outside the module and is displayed on the general purpose LEDs on the KC705. The consumer module is used for debugging during development to ensure that the data payload is unaltered by the protocol stack.

Chapter 4

Results

4.1 Experimental Setup

The experimental set up is shown in Figure 4.1. Two KC705s and a host computer are plugged into a Layer 2 switch. Netgear Prosafe Plus GS105E Switch was chosen because it is inexpensive and has port mirroring capabilities. The Layer 2 switch can be configured with port mirroring easily using a Windows based computer and Netgear's simple interface. Port mirroring allows all traffic that enters on one or more ports to be mirrored to exit on another port than the one it would normally flow to. This will allow traffic to be observed using packet capturing software, Wireshark, on the host computer. Wireshark version 1.10.0 has been used to observe and capture packets as they travel between the two KC705's. The DG-RDMA protocol can be recognized upon packet inspection. A screen capture of DG-RDGMA packets in Wireshark is shown in Figure 4.2. The Source and Destination MAC address shown in the figure correspond to the MAC addresses of the two KC705s.

A variety of data payload sizes have been tested in the range from 0-1K Bytes. Using the data and length modes as described in section 3.2, different data payload values have been tested as well. Messages, synonymous to the payload data, are created opportunistically and propagate through the protocol stack as they are generated.

4.2 Frequency

The system frequency is 125MHz with a 50% duty cycle. All timing requirements at 125MHz were met. Timing closure at 200MHz would be an added bonus, as it was one lofty goal set during implementation. Optimization of the system will likely allow timing closure at a greater frequency, decreasing overall latency.

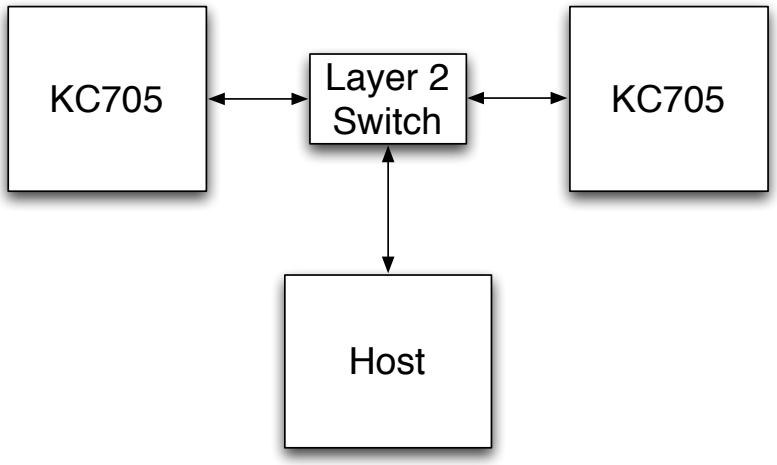

Figure 4.1: Experimental Set up

No.	Time	Source	Destination	Protocol	Length	Info
1055670	4.472980000	Xilinx_02:a2:42	Xilinx_02:76:b3	0x3333	88	Ethernet II
1055671	4.472983000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	60	Ethernet II
1055672	4.472986000	Xilinx_02:a2:42	Xilinx_02:76:b3	0x3333	60	Ethernet II
1055673	4.472990000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	88	Ethernet II
1055674	4.472993000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	60	Ethernet II
1055675	4.472996000	Xilinx_02:a2:42	Xilinx_02:76:b3	0x3333	88	Ethernet II
1055676	4.473000000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	88	Ethernet II
1055677	4.473003000	Xilinx_02:a2:42	Xilinx_02:76:b3	0x3333	88	Ethernet II
1055678	4.473005000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	60	Ethernet II
1055679	4.473008000	Xilinx_02:a2:42	Xilinx_02:76:b3	0x3333	60	Ethernet II
1055680	4.473010000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	88	Ethernet II
1055681	4.473014000	Xilinx_02:a2:42	Xilinx_02:76:b3	0x3333	88	Ethernet II
1055682	4.473018000	Xilinx_02:76:b3	Xilinx_02:a2:42	0x3333	88	Ethernet II

```
▷ Frame 1055675: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface 0
▷ Ethernet II, Src: Xilinx_02:a2:42 (00:0a:35:02:a2:42), Dst: Xilinx_02:76:b3 (00:0a:35:02:76:b3)
▷ Data (74 bytes)

0000  00 0a 35 02 76 b3 00 0a  35 02 a2 42 33 33 20 42   ..5.v... 5..B33 B
0010  10 42 00 02 00 00 00 01  00 00 00 01 fe ed c0 de   .B...... ........
0020  ca fe ba be 00 02 00 01  be ef f0 0d 00 08 01 01   ........ ........
0030  00 00 00 08 00 00 00 01  00 00 00 01 fe ed c0 de   ........ ........
0040  ca fe ba be 00 02 00 00  be ef f0 0d 00 08 00 00   ........ ........
0050  00 01 02 03 04 05 06 07                           ........
```

Figure 4.2: DG-RDMA Packets in Wireshark

Utilization		
Module	**LUTs**	**Flops/Latches**
Producer	478	433
Consumer	173	181
Sender	3036	1385
Receiver	2405	757
FDU	895	886
FAU	704	809
Fork Send	427	812
Merge Receive	554	810
Ack Tracker	146	204
Ack Aggregator	99	150
MergeForkFDU	599	879
MergeForkFAU	706	1251
MergeToWire	873	1411
Funnel	239	359
Unfunnel	409	461
L2 Inserter	495	551
L2 Remover	340	544
GMAC	425	688

Table 4.1: Utilization for each system Module

4.3 Utilization

The device used for implementation and testing is the Xilinx KC705 Evaluation Platform with FPGA part number XC7K325T-2FFG900CES. The Kintex 7 FPGA uses a 6 LUT architecture. Xilinx Vivado v2013.1 64-bit is used as for RTL Synthesis, Place and Route for the design on RHEL5. Total area required for the DG-RDMA IP core is 14,371 Flops/Latches and 14,715 LUTs. Roughly, 7% of the available LUTs and 4% of the available Flops/Latches. Table 4.1 shows a break down of the utilization by module.

The total area of the Sending modules is 6,476 LUTs and 10,970 Flops/Latches. Total area of the Receiving modules is 5,823 LUTs and 10,400 Flops/Latches. The shared modules have a total area of 2,781 LUTs and 4,014 Flops/Latches.

The area of the design could be reduced by logical optimization in many of the modules. Total utilization could also be reduced by combining some of the modules that do not necessarily need

to stand alone. Most of the FIFO's between modules are of type HexBDG which is not a small channel for communication. Employing this reduction would reduce the number of FIFO's needed for communication among the modules.

Figure 4.3 shows device utilization for the KC705. Sending modules are in blue, Receiving modules are in red and shared modules are in yellow.

4.4 Bandwidth

Throughput is the average rate of successful communication transfer over a network. Throughput can be measured by accumulating the number of Bytes sent out of the IP core over a period of time.

Ideal or best case throughput is calculated by multiplying the data width that progresses each cycle by the frequency of the data progressions. The data width is 16 Bytes and the frequency is 125 MHz, as stated in section 4.2. This yields an idealistic throughput of 2 GB/S or 16 Gb/S.

A cycle counter, which measures cycles of a known period, is used to find the length of time the that it takes for bytes to transmitted. Latency for messages of different payload lengths are measured. This latency is reported in section 4.5. The calculated bandwidths are shown in figures 4.4,4.5 and 4.6.

The Round Trip Time bandwidth graph shown in figure 4.6 shows that the implementation comes very close to the ideal bandwidth calculation. With the conversion of MB/S to Mb/S, we see that bandwidth of 976 Mb/S is achieved. This is very close to the 1 Gigabit Ethernet limit of the communication channel.

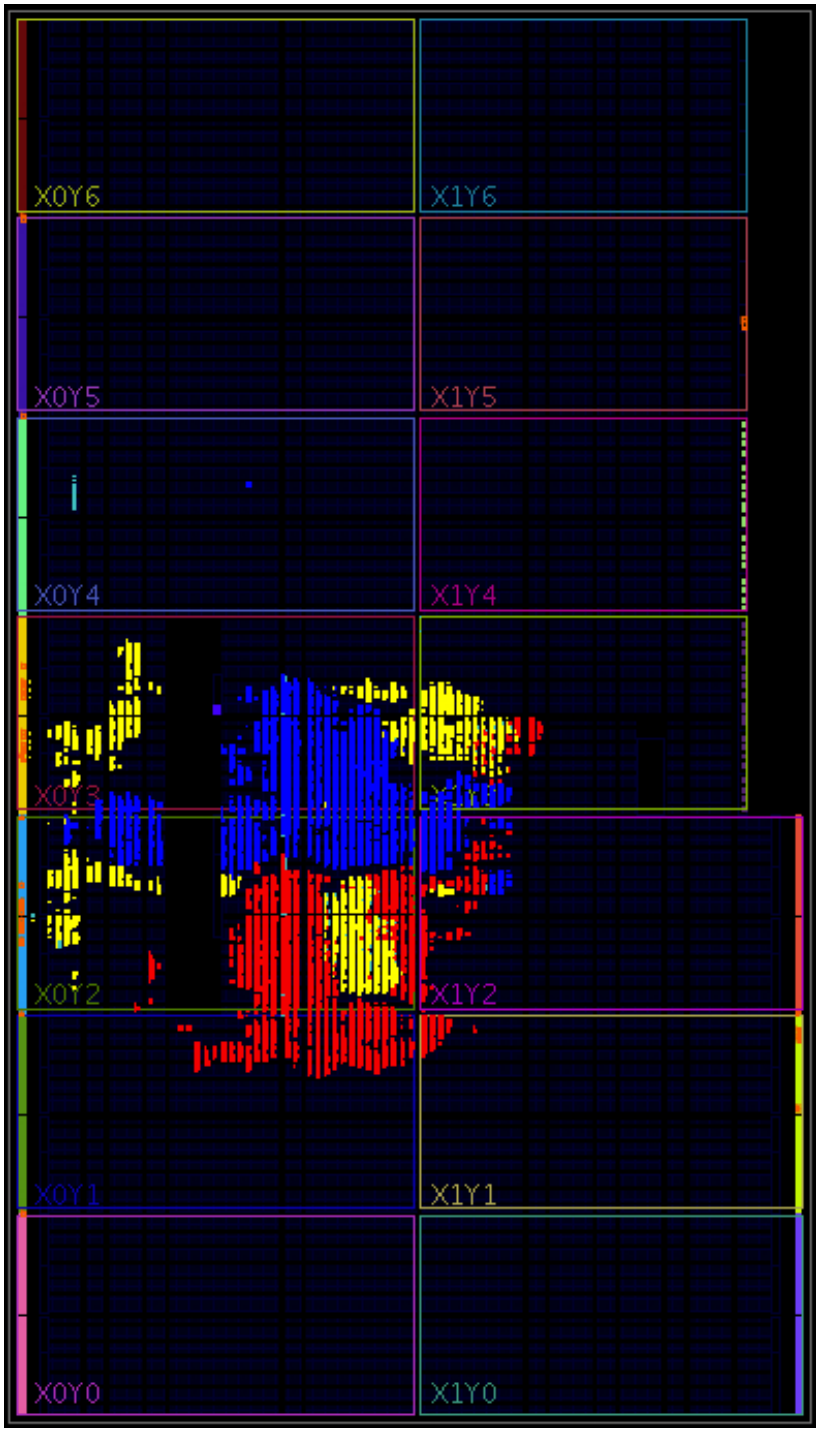

Figure 4.3: KC705 Chip Utilization

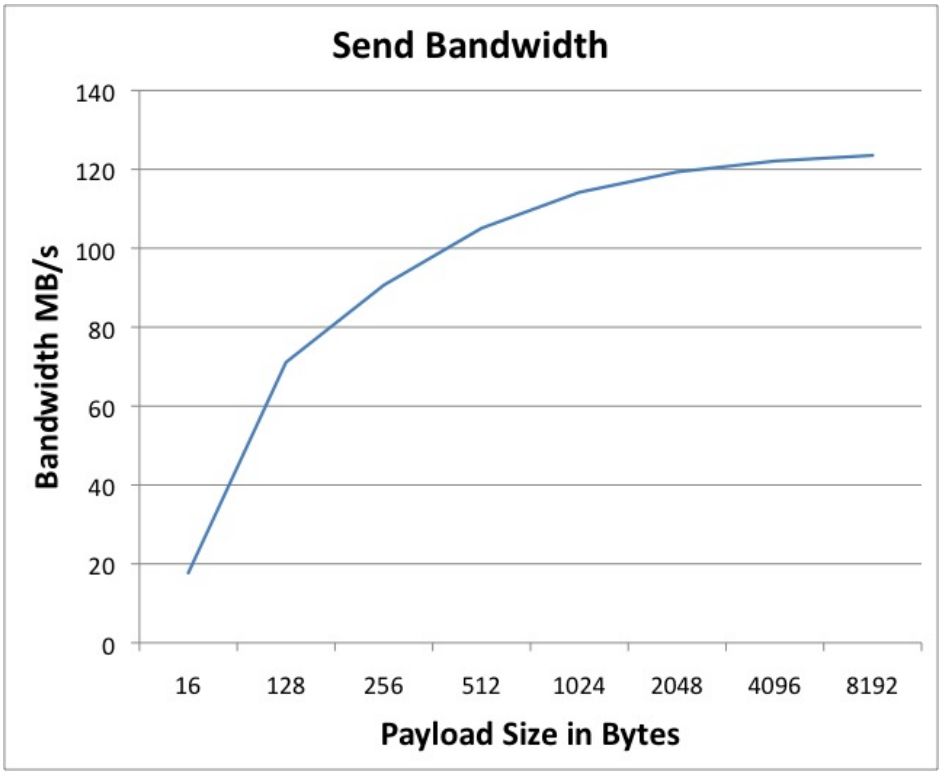

Figure 4.4: Send Bandwidth Graph

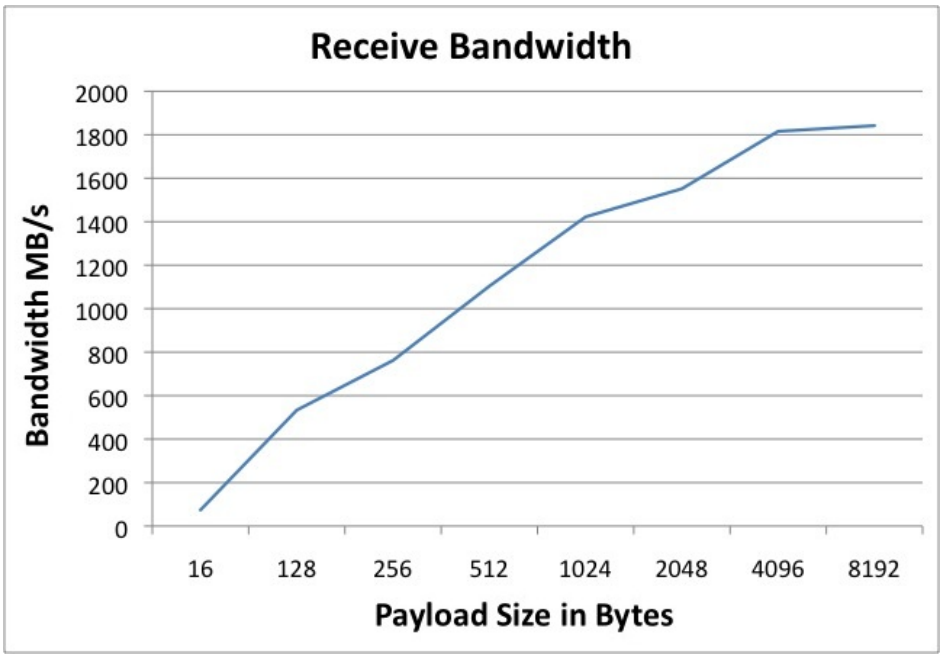

Figure 4.5: Receive Bandwidth Graph

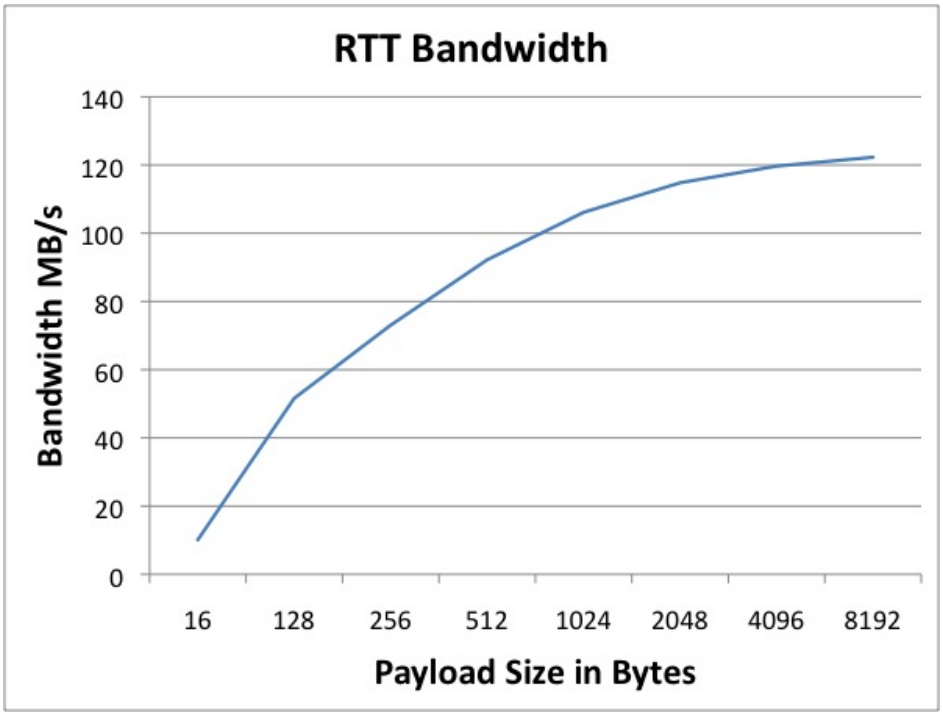

Figure 4.6: Round Trip Time Bandwidth

4.5 Latency

Latency is a measurement of the time for a function to complete. Latency is measured by counted clock cycles at the known frequency. The number of clock cycles is multiplied by the period of the system. Latency is reported in microseconds. Cycle counters are sampled at certain points in the system that indicate complete actions. Overhead latency is measured for a payload size of zero. Latency is sampled for payload sizes 16 bytes, 256 bytes, 512 bytes and 1024 bytes.

In this design, the latency of interest is outlined below.

Send latency is measured by taking a sample of the cycle count when the first payload byte enters the Sender module and again when the last byte of the packet leaves the GMAC and goes to the PHY. This measures the latency between frame creation to packet departure. Figure 4.7 shows the path that is measured for send latency. The green arrow signifies the start of the measurement, the red arrow signifies the end of the measurement and the blue arrows show the measured data path through the protocol stack. This convention is used for each of the described latency measurements in this section.

The measured results for send latency are shown in Figure 4.8. Latency is measured in microseconds. The transmission of messages is the most costly operation within this implementation. Frame generation and acknowledgment are hand

Receive latency is measured by taking a sample of the cycle count when the first byte of the packet enters the GMAC from the PHY and again when the last byte of the payload is passed into the Consumer module. The measured results for receive latency are shown in Figure 4.10. As compared to the latency for transmitting a message, reception latency are much lower. Frames do not have to be generated and acknowledged, rather the headers just have to be checked and removed in the receiving module.

Round Trip latency is measured by sampling the cycle count when the first payload byte enters the Sender module and again when the AckTracker signals to the FDU that it can release the frame and accept a new one. The measured results for round trip latency are shown in Figure 4.11.

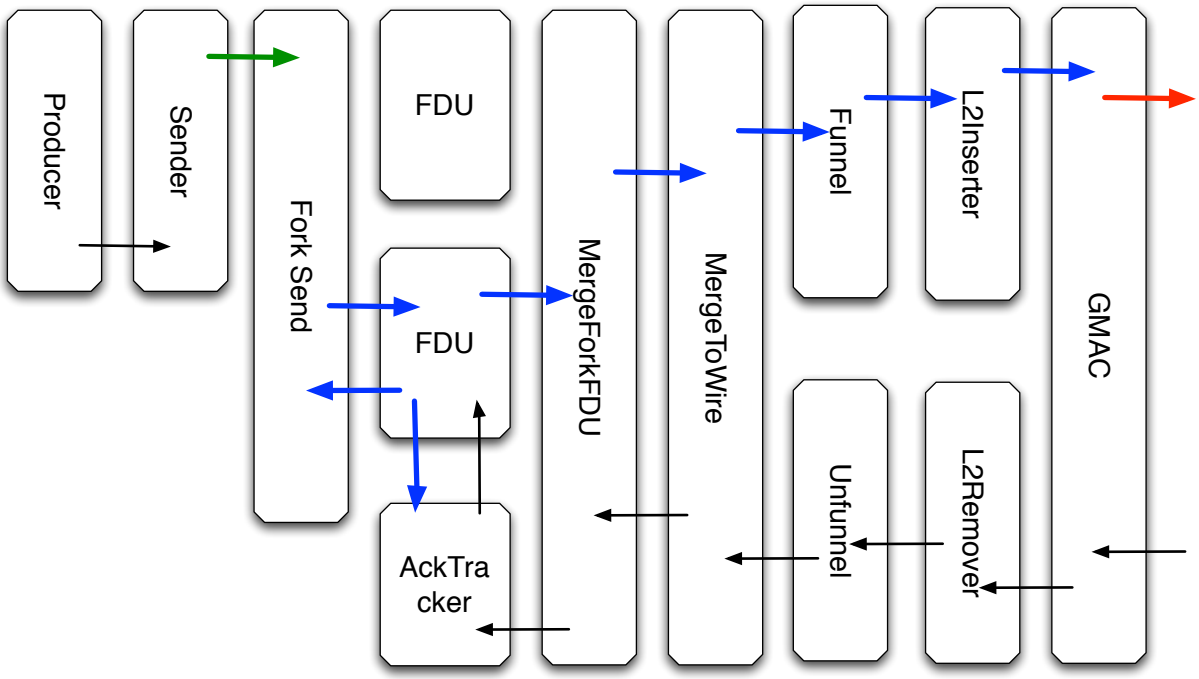

Figure 4.7: Send Latency Measured Path

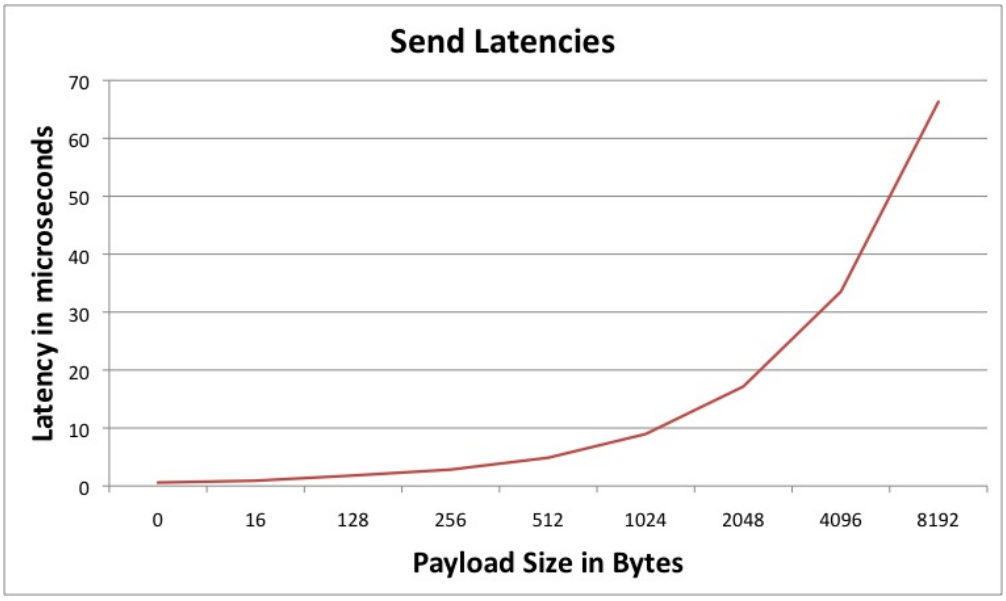

Figure 4.8: Send Latency Graph

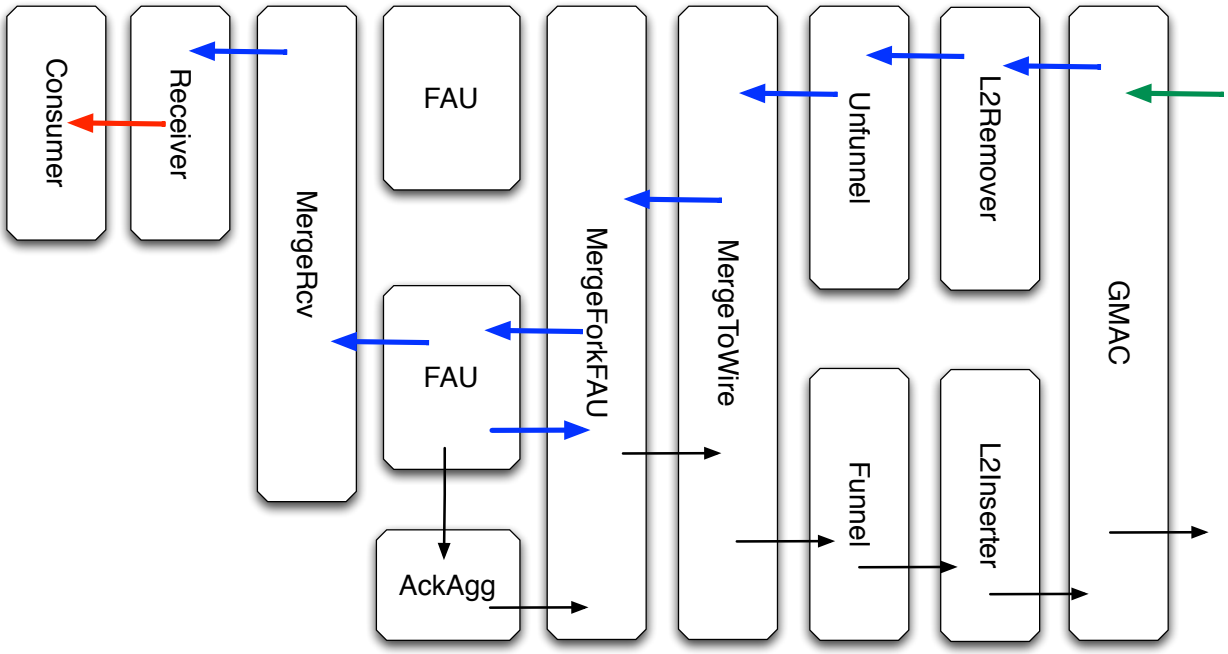

Figure 4.9: Receive Latency Measured Path

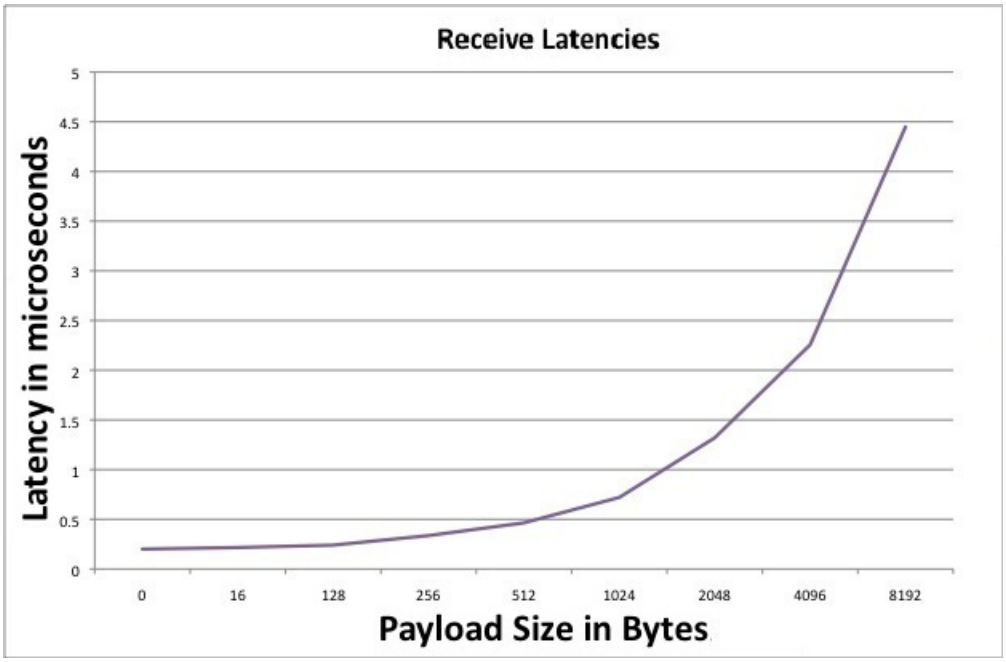

Figure 4.10: Receive Latency Graph

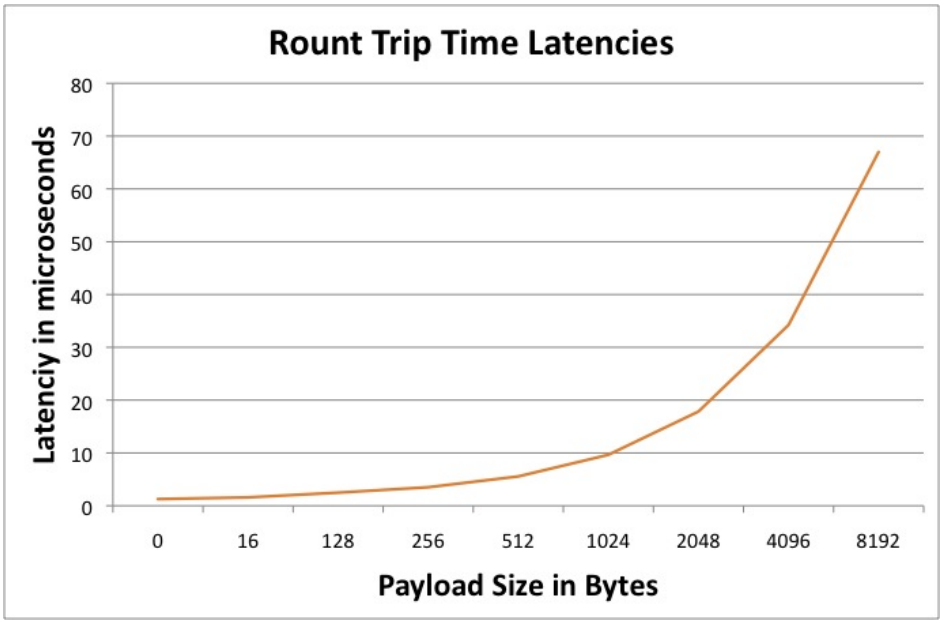

Figure 4.11: Round Trip Latency Graph

Round trip latency measures time from frame creation to frame acknowledgment. The majority of this time is due to the transmission operations.

Retransmission latency is measured by sampling the cycle count when the retransmission time-out counter expires and again when the last byte of the packet leaves the GMAC and goes to the PHY. Figure **??** shows the path that is measured for retransmission latency. The measured results for retransmission latency are shown in Figure 4.13.

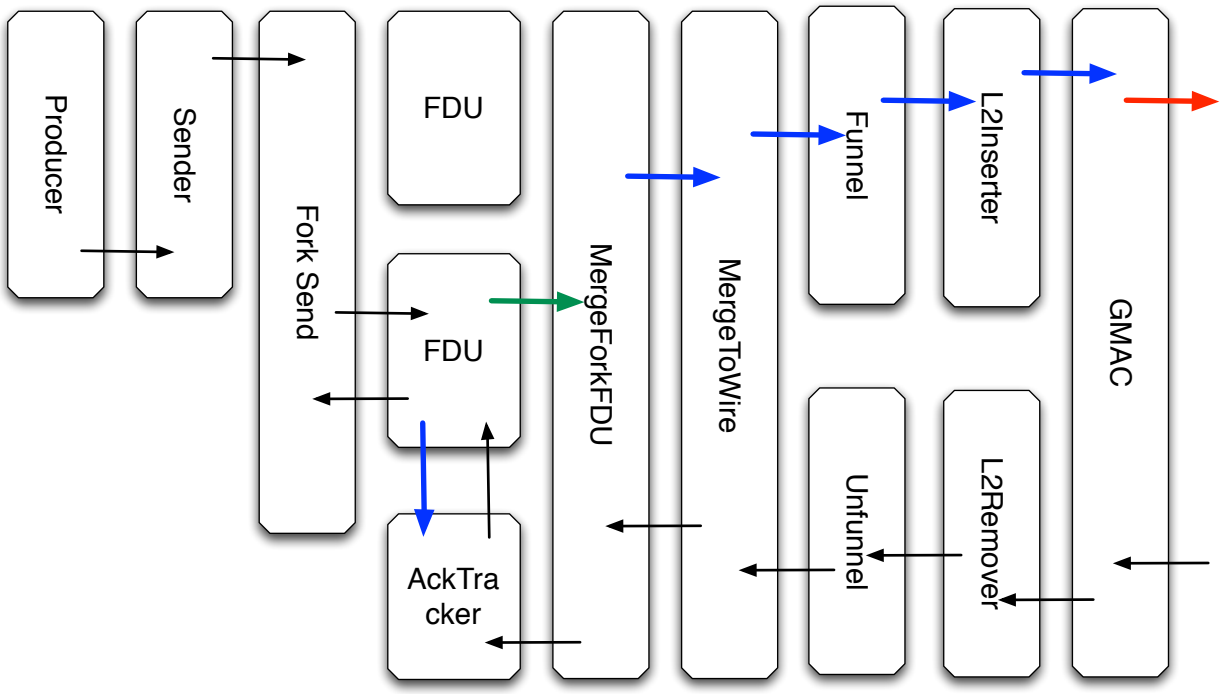

Figure 4.12: Retransmission Latency Measured Path

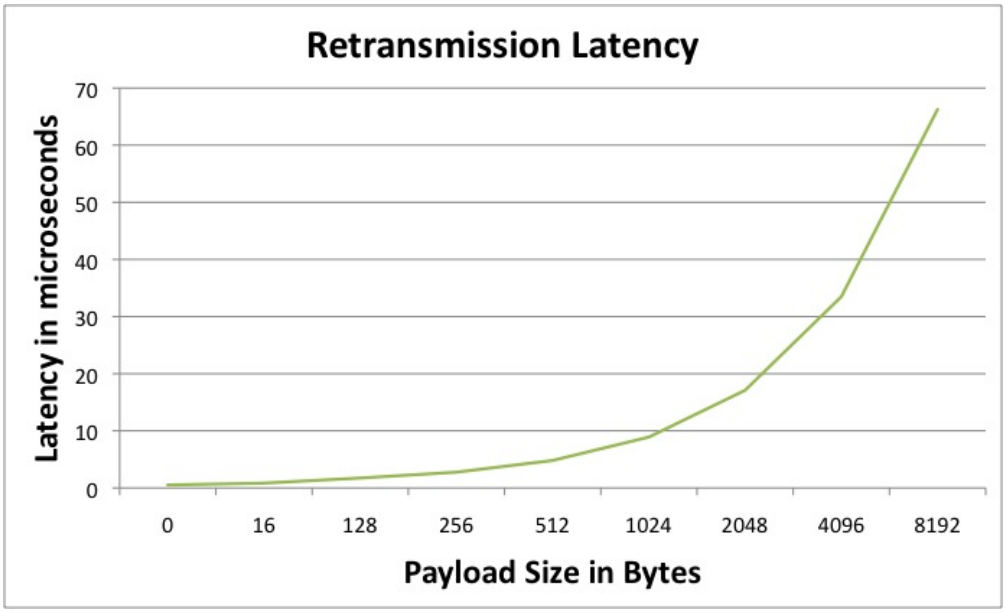

Figure 4.13: Retransmission Latency Graph

4.6 Lines of Source Code

Lines of source code is a metric used to estimate the time to solution and complexity of the design.

This implementation of DG-RDMA is 2,328 lines of Bluespec SystemVerilog code. The total number of lines of code was added together and any empty lines were subtracted from the total number of lines. This calculation will account for only lines in the source that are in fact code.

The Bluespec Compiler is used to generate Verilog RTL. There are 16,939 lines of Verilog that have been generated. This is almost an order of magnitude difference in the amount of source code required to implement DG-RDMA IP core. Without the help of BSV, the effort to implement this design in Verilog would have been much greater and certainly required more time and more engineering effort.

Chapter 5

Conclusion

Communication performance is just as important as computation performance in high speed networks, even in embedded systems. Doing fast and efficient data transfers is a responsibility left to endpoint processing. A myriad of protocols and interconnects exist for high speed transactions. Due to the advantages of a zero copy protocol, Remote Direct Memory Access (RDMA) is chosen as a paradigm. Remote Direct Memory Access is the transfer of data into buffers between two endpoints that does not involve the CPU or Operating System (OS). RDMA promotes reliable, high throughput and low latency transfer for packet-switched networking.

DG-RDMA Protocol Specification is a simple and lightweight protocol that does not rely on an underlying Internet Protocol service. It transports datagrams over a transmission layer that is unreliable, possibly out of order, yet error free when delivered. DG-RDMA has been implemented in Bluespec SystemVerilog on a Xilinx KC705. The protocol is designed for wired, Layer 2 packet-switched networks in which endpoints are addressable by MAC address. An experimental set up with two KC705s connected by an L2 switch is used to perform latency, throughput and area measurements.

This thesis explores how to bring low latency data transfer capabilities for packet-switched networks over Ethernet into the open source FPGA community. An RDMA protocol in which the currency is Datagrams is designed, implemented and tested between two Xilinx FPGA's over a Layer 2 switch. The implementation does not rely on an Internet Protocol and is portable, simple and lightweight.

5.1 Contributions

In this thesis, we present a simple, low-latency, open source, reliable and lightweight RDMA capability to the FPGA community. This addresses the current lack of no standard or freely available

RDMA IP cores that do not rely on Internet Protocol services. The following contributions have been made in support of this statement.

- Designed a reliable and portable Ethernet RDMA IP core
- IP core does not require Internet Protocol and can be scaled to different line rates
- Implemented prototype on Xilinx KC705
- Tested two KC705s with RDMA IP core over Layer 2 switch
- Created IP core in Bluespec SystemVerilog

5.2 Future Work

The implementation of DG-RDMA focuses on functionality first, then on performance. There are many instances in the design that can be optimized to improve latency, throughput and area. Data flow processing can be parallelized and sped up. Increasing the operating frequency from 125 MHz to 200 MHz would also improve the latency and is not an unrealistic goal for the current design. The speed increase to 200 MHz will also allow the protocol to run at a line rate for 10GbE.

The DG-RDMA protocol allows for some features that are useful, but not implemented in this work. Flow control is built in within the endpoint to some degree due to FIFO semantics, however, more explicit flow control could be implemented. A study to define the max number of retransmissions that brings the best performance to the system could be performed. In the current implementation, ack frames are sent separately for each received frame. DG-RDMA allows for multiple acks to be sent in one frame, reducing the amount of bytes on the wire for each transaction, which would improve performance. In order to support variable line rates for Ethernet, the interfaces between modules could be converted to polymorphic types within BSV. This would support further flexibility and has the potential to allow for improved throughput and latency. One of the original goals of this implementation is to move to 10GbE, although this implementation is geared towards 1GbE. With an internal data width of 16 Bytes operating at 200MHz, 10GbE

is over provisioned for. By widening the data path further, even greater Ethernet line rates are probable.

In chapter 3, it is mentioned a few times that the number of FAUs and FDUs in the design is an implementation choice. Conducting a study to measure throughput with varying number of FAUs and FDUs would yield the ideal number for maximum throughput. Another study could be conducted to determine if double buffering the BRAM in each FDU/FAU would increase throughput. Unused header fields that are pointed out in chapter 3 could be removed which would improve the performance as well. The MAC addresses that are used for source and destination addressing in the L2 header are hard coded in this implementation. In order to be more correct, the source MAC address should be read from a local I2C Flash.

The IP core could be packaged and interfaced with Xilinx and Altera tools, making use simple for developers. A sample implementation on both a Xilinx FPGA and Altera FPGA communicating over an L2 switch would be a nice demonstration of the cores potential for portability. Another feature to promote portability and reuse would be to interface the IP core with other interconnection networks such as PCIe. An implementation of DG-RDMA could be developed for use with GPUs allowing wider adoption of the protocol. To further promote reuse, a DG-RDMA packet dissector for WireShark software could be created and made available with the open source IP core.

Creation of a data logging utility in which the protocol is running in HW on the FPGA and in SW on a CPU could be used to send data from the FPGA to the CPU. The data would be written to a text file and available to be read. This is a useful tool for running tests on the FPGA and easily capturing the data the FPGA produces.

References

[1] Nikolaos Alachiotis, Simon A. Berger, and Alexandros Stamatakis. Efficient pc-fpga communication over gigabit ethernet. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1727–1734, Washington, DC, USA, 2010. IEEE Computer Society.

[2] I.C. Bertolotti and Tingting Hu. Real-time performance of an open-source protocol stack for low-cost, embedded systems. In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8, 2011.

[3] Neal Bierbaum. Mpi and embedded tcp/ip gigabit ethernet cluster computing. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, LCN '02, pages 733–734, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Bluespec. Bsv by example. http://wiki.bluespec.com/Home/BSV-Documentation. Last accessed November 22, 2013.

[5] Ron Brigthwell and Matthew Curry. The Case for an RDMA Extension to MPI. http://www.sandia.gov/~rbbrigh/slides/posters/rdma-mpi-sc05-poster.pdf. Last accessed November 22, 2013.

[6] Sun HPC ClusterTools. One-sided communication. http://docs.oracle.com/cd/E19061-01/hpc.cluster6/819-4134-10/1-sided.html. Last accessed November 22, 2013.

[7] A. Dollas, I. Ermis, I. Koidis, I. Zisis, and C. Kachris. An open tcp/ip core for reconfigurable logic. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 297–298, 2005.

[8] P. Geoffray. A critique of rdma. 2006.

[9] K. Hashimoto and V.G. Moshnyaga. A new approach for tcp/ip offload engine implementation in embedded systems. In *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pages 1249–1253, 2010.

[10] Jeff Hilland, Paul Culley, Jim Pinkerton, and Renato Recio. Rdma verbs specification. http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf. Last accessed November 22, 2013.

[11] Amir Hirsh. An interpreted hardware description language. http://fpgacomputing.blogspot.com/2006/12/interpreted-hardware-description.html. Last accessed November 22, 2013.

[12] Hankook Jang, Sang-Hwa Chung, and Dae-Hyun Yoo. Implementation of an efficient rdma mechanism tightly coupled with a tcp/ip offload engine. In *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pages 82–88, 2008.

[13] James Kulp John Miller and Shepard Seigel. Opencpi datagram rdma (dg/rdma) protocol specification. http://opencpi.org/doc/ts.pdf, 2012.

[14] N.M. Khalilzad, F. Yekeh, L. Asplund, and M. Pordel. Fpga implementation of real-time ethernet communication using rmii interface. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 35–39, 2011.

[15] P. Lieber and B. Hutchings. Fpga communication framework. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 69–72, 2011.

[16] A. Lofgren, L. Lodesten, S. Sjoholm, and H. Hansson. An analysis of fpga-based udp/ip stack parallelism for embedded ethernet connectivity. In *NORCHIP Conference, 2005. 23rd*, pages 94–97, 2005.

[17] Allyn Romanow and Stephen Bailey. An overview of rdma over ip. In *In First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet*, 2003.

[18] Xilinx. Xilinx vivado 4x faster implementation. http://www.xilinx.com/products/design-tools/vivado/prod-advantage/faster-implementation/index.htm. Last accessed November 22, 2013.

[19] E. Yeh, H. Chao, V. Mannem, J. Gervais, and B. Booth. Introduction to tcp/ip offload engine (toe). 2002.