Exploring Ethernet Switching Architectures for Area-Efficient Low-End Switches

JON SWEDBERG AND FELIX GHOSH

MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Exploring Ethernet Switching Architectures for Area-Efficient Low-End Switches

Jon Swedberg and Felix Ghosh

Department of Electrical and Information Technology
Lund University
Packet Architects AB

Supervisor: Liang Lui

Examiner: Erik Larsson

June 3, 2023

© 2023
Printed in Sweden
Tryckeriet i E-huset, Lund
All figures were made by the authors

Abstract

The aim of this thesis project has been to develop an architecture for L2 ethernet switches that would be optimized for silicon area, targeting smaller *low-end* switches. A selection was made of three different switching architectures, which were compared and analyzed to explore the benefits and drawbacks of different approaches. From these, one architecture called *Shared Memory Linked-List* was selected that served as a base to develop a new area-efficient architecture. This architecture was implemented in the form of two different port configurations using MyHDL to generate Verilog code, which was used for behavioral simulation. The RTL code was synthesized into both an FPGA and ASIC implementation which was compared to a contemporary alternative in the form of an equivalent ethernet switch generated by the FlexSwitch tool suite developed by Packet Architects AB. The four-port configuration of the thesis implementation showed significant area reductions in the buffer management subsystems for both the FPGA and ASIC versions, while the ten-port configuration showed a similar reduction in the ASIC version, while the FPGA implementation decreased the usage of certain hardware components while others increased. An analysis of the architecture, its benefits, and drawbacks was performed and potential future improvements were suggested.

Keywords: Ethernet Switch, Architecture, Silicon Area, Area Optimization, ASIC, FPGA

Popular Science Summary

Ethernet switches are electronic devices that serve a very important role in the modern internet. These devices could be seen as some sort of traffic controller that takes data that is being sent over the internet and ensures that it continues to travel toward its correct destination. Ethernet switches can be found in large data centers, office buildings, homes, and even inside other machines. When you are sending data over the internet in any form, be it sending an e-mail, or uploading a photo to a website, that information will travel from your computer through a long series of switches that forward it before it finally reaches its destination.

Another thing that is interesting about ethernet switches is the fact that they come in all shapes and sizes. The switches that you might have seen in a data center are very large and contain hundreds or thousands of ports connected to ethernet cables, while the ones you see in an office or home are much smaller and could contain less than ten ports. The fact that these devices vary so greatly in size, means that the underlying design and architecture of how the switch is constructed becomes quite important. The way in which you build a data center switch that is the size of a large closet, might not be an ideal way to build one that is the size of a DVD case. This is the case since even though both of these machines are performing the same tasks, the way in which they do it might be completely different. A very large switch must be built to be as fast as possible, while a smaller one might have to consume less power or be as small as possible. There exists plenty of research regarding how one should build very large switches effectively, but there seems to be an apparent knowledge gap regarding architectures that are well suited for smaller switches. This is the gap that this thesis work attempted to start filling.

The thesis goal was to produce an efficient architecture for constructing smaller "low-end" ethernet switches. This architecture was constructed to minimize the total area of the switch, since for a low-end switch, making it smaller can be deemed more important than making it faster. This was done by selecting and comparing different existing architectures so that their respective benefits and drawbacks could be studied. These architectures were in some ways combined with each other as well as with some ideas of our own in order to produce the final architecture. Finally, the architecture was actually built. In order to measure its "area-efficiency", its size was compared against a contemporary alternative in the form of ethernet switches produced by the company Packet Architects AB. The results showed that the thesis architecture enabled significant area reductions for certain parts of the switch, but also that this architecture does not seem to be well suited for larger switches.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Ethernet switches are an integral part of networking infrastructure that serve a variety of roles in a network. Most commonly they are found at the edges of the network, connecting devices to an access point. Alternatively, they can be found deeper in a network, connecting other switches that reside at the edges. The importance of their role in network communication can be seen reflected in their market demand, which has been observed to be increasing [1].

Different applications for switches also produce different requirements for the devices themselves. Large-scale switches with Tbit speeds and hundreds or thousands of ports are typically found in large data centers, while smaller switches in the ranges of tens or fewer ports, with Gbit/Mbit speeds, can be found in small offices, homes, or embedded constructions such as cars. This means that there exists a very wide variety of ethernet switches that all come with different requirements for their production.

One way of meeting the demand for such a broad spectrum of product specifications is to use *IP-core generators*, systems that are capable of generating IC designs for a customized ethernet switch based on user-specified requirements. This approach allows one to construct a design in a relatively short time compared to designing the switch from scratch every time. However, constructing such a system is not a trivial task.

Packet Architects AB is a company based in Lund, Sweden, which has developed such a tool suite, called FlexSwitch. This tool allows them to write packet processing functionality in a high-level language which is synthesized down to RTL code, which can then be used to implement packet processing hardware such as switches, and routers for both FPGA and ASIC technologies. In implementing this system, they constructed an architecture for their switches that would be well suited for covering a broad range of specifications, and able to deliver high-performance switches with up to Tbit/s bandwidth.

However, one architecture will never be able to cover the demands of every specification. An architecture that is built to facilitate high performance, will do so at the expense of other factors such as area and power consumption. These factors can be tremendously important for switches at the lower end of the spectrum, where performance requirements can be relatively lax. Thus they identified a problem in that the FlexSwitch tool suite currently does not generate designs that are area-efficient for smaller low-end switches.

To try and rectify this problem, they reviewed the current literature regarding ethernet switch architectures. They found that there exists research in the areas of ethernet switch architectures as well as optimizations for area efficiency and power consumption, focusing on larger-scale systems. An example of this is [2], where a switch architecture is proposed for use in data centers that yielded reductions in cost, power consumption, and area. Another example can be found in [3]. Here the authors present an architecture that is scalable for larger systems with a design based on using a network on chip (NoC).

However, there exists an apparent gap in the scientific literature regarding research on ethernet switch architectures for smaller, "low-end" switches. Questions regarding which architectures are well suited for smaller switches, how these architectures affect area and performance, as well as how these architectures scale, are integral when designing an IP-core generator such as FlexSwitch. This master thesis aims to begin answering some of these questions by exploring different ethernet switch architectures to produce an area-efficient architecture that is well-suited for low-end switches.

1.2 Project scope and methodology

All ethernet switches manage data in the form of so-called packets, which are chunks of serial data that travel into the switch through one of its ports and exists through another port. On a conceptual level, an ethernet switch can be divided into two parts. The first part is responsible for managing the processing of packets, which includes modification of packet data as well as deciding which ports packets should be sent to. The other part consists of modules that are responsible for intermediate storage of the packet data, finding a suitable data path for the packet through the switch, as well as deciding when to drop packets in case of overload. These two parts can be named *Packet Processing* and *Buffer Management* respectively and each comes with its own architectural challenges.

Presently, the FlexSwitch tool suite is developed to facilitate high performance and throughput of packets, and its architecture reflects this. However, given the relaxed performance requirements for a low-end switch, certain architectural changes may yield a more area-efficient design. The FlexSwitch architecture has a highly advanced packet processing unit, which consists of two modules IPP (Ingress Packet Processing) and EPP (Egress Packet Processing) which have the capabilities to perform the address lookups as well as to modify the packet for more advanced forwarding using technologies such as VLAN. This system is highly complex as it is developed to be configurable to be able to meet high performance requirements. Given this high complexity, this thesis limited its scope to developing architectural improvements for the second part of the switch, namely the buffer management. The storage and movement of data inside the switch contains many different architectural decisions that must be made with due consideration paid to the performance requirements of the device.

Thusly, in testing and simulation, the real Packet Processing modules were substituted for dummy modules that provide the basic address lookup functionality that is required for the switch to operate, but will not have any of the more advanced features that the FlexSwitch architecture implements. In order to ensure the accuracy of the final measurements, all the modules that facilitate the Packet Processing capabilities of both the thesis switch as well as FlexSwitch were subtracted from the final values, so that solely the Buffer Manager of each respective switch was compared.

The thesis work was conducted by first reviewing existing switching architectures. Three different architectures were selected for further study and comparison, which were used to determine how an area-efficient architecture could be constructed. One of these was selected as a base and in some ways combined with aspects of the other architectures as well as other design ideas, in order to produce a final area-efficient architecture. This architecture was implemented using the Hardware Descriptive Language (HDL) MyHDL, which was used to generate RTL (Register Transfer Level) Verilog code that could be used to simulate the behavior of the architecture as well as synthesize FPGA and ASIC implementations of the switch. To measure the area efficiency of the final architecture, the implementation was compared to a design corresponding to the same switch specifications, that was generated by FlexSwitch. This allowed for observations regarding how the architecture holds up against a contemporary alternative for both FPGA and ASIC technology using several different measurements.

1.3 Project target

This thesis produced an ethernet switch architecture for low-end switches. The target of the project was to ensure that this architecture was optimized for area-efficiency. A switch using this architecture was implemented in RTL code that met the following requirements:

- The switch will operate solely on Layer 2.
- The port configuration will be 4 and 10 ports respectively.
- The data bus width will be 8 bits.
- The clock frequency will be 12.5 MHz.
- Scheduling will use a strict priority scheduler with 4 queues.

This RTL code was used to synthesize an implementation for a XILINX Field Programmable Gate Array (FPGA) board as well as a synthesis for a 0.18 μm Application Specific Integrated Circuit (ASIC) technology.

A "Mesh Test" was employed in order to verify the behavior switch. This tested the switch during a "full overlap" scenario that consists of every port continuously sending simultaneous packets to a single destination port which rotates with every packet. During this test, the switch had to be able to correctly identify the destination port of every packet and forward it correctly without dropping a single packet.

Chapter 2

Theory

This chapter presents the prerequisite theory regarding ethernet switches, their function, as well as their composition. After this, a brief introduction is given regarding different IC technologies. Finally, the tools that were utilized during the thesis are described.

2.1 Layer 2 ethernet switches

Traditionally, ethernet switches operate on layer 2 in the OSI-model, the so called link layer. This means that the switch uses the protocol data unit of the second layer, namely the Ethernet frame, to make decisions about where an incoming packet should be forwarded. As seen in Figure 2.1, an ethernet frame contains the MAC address of both the sender of the packet as well as the destination machine. An ethernet switch uses these two addresses to deduce where the packet should be sent.

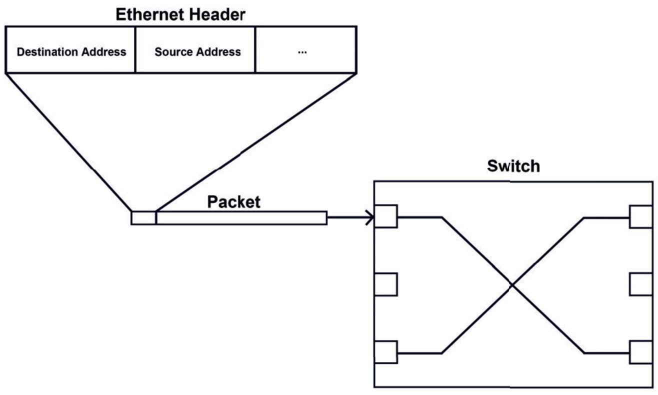


Figure 2.1: A look at the inside of an ethernet packet.

2.1.1 Algorithmic overview

A switch contains a lookup table that matches MAC addresses to ports on the switch. This lookup table uses Content-Addressable Memory (CAM)[6] which can be viewed as a hardware equivalent of a software key-value map data structure. That is, the memory can be searched using a key, which will return the data corresponding to the key if it exists in the table. This table is filled by looking at the source address of a packet and creating an entry mapping it to the port that received it. This way the switch will gradually learn which ports lead to which MAC addresses. This means that when a packet arrives, the destination address can be used as a key to lookup which destination port the packet should be sent to. If the destination address is not currently contained inside the table, the switch has no way of knowing which port leads to the destination machine. To ensure that the packet will arrive at its destination, the switch will then *flood* the packet, meaning that it is sent to every port except the port that received it. When a packet is received on a port, it will be placed in some kind of buffer while its destination port is being determined. It will also need to be placed in a buffer while it is waiting for its destination port to become available since a port can only send out one packet at a time. The switch will also implement some kind of prioritization regarding the order in which it sends out its packets. The method by which this prioritization is determined varies and certain switches will even look at the contents of the L3 protocol data unit inside the payload of the packet to make this decision.

This process of learning which MAC addresses are associated with which ports, and using the lookup table to forward packets to the correct ports constitute the basic algorithm that L2 switches implement. Of course, there exist other more advanced features that a switch may implement such as packet modification, Virtual Local Area Networks (VLAN), and mirroring. However, these are of no relevance to this thesis, since these features fall outside of the project specification.

2.1.2 Architectural introduction

Here follows a brief introduction to the different conceptual modules that make up the inner workings of an ethernet switch. These modules, their implementation, as well as their interconnections and layout, constitute the architecture of a switch. Thus, understanding the basic function that each module serves is vital to understanding the switch as a whole.

The switch itself can be divided into an outer layer that handles receiving and transmitting data using standard modules for ethernet communication, as well as an interior layer called the *Switching Core*. When one refers to different ethernet switch architectures, one is usually really referring to different architectures of the switching core, since this is what performs the actual switching. This core can be divided into five different conceptual modules as can be seen in Figure 2.2.

The switching core takes its input data from each ethernet port in the form of a stream of serial bytes. For each byte in this stream there also exists one-bit of data indicating if this byte is the first byte of the packet, one bit indicating if it is the last byte of the packet, as well as one bit indicating if the byte is valid or not. This data is sent into the first of the five modules named **Serial to Parallel**

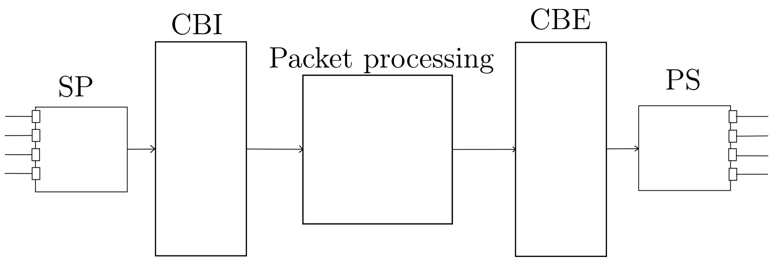


Figure 2.2: A conceptual overview of an ethernet switch.

or simply **SP**. In general, it can be quite cumbersome to handle information as a stream of serial data in hardware. It tends to be easier to handle larger chunks of parallel data given the inherently parallel nature of integrated circuits. Thus, the job of the first module is to convert the stream of serial data into parallel chunks of a fixed size that are referred to as cells. Throughout the rest of the switch, these cells will be used as the smallest primitive of packet data that is sent around, until the packet is exiting the switching core on its way to its output port, when the data will need to be converted back to a serial stream. The SP module usually performs this conversion from serial to parallel data by using a shift register that can output cells at regular timed intervals. These cells will then be sent as input to the next module (along with data marking the first and last cells as well as how many bytes are valid in each cell). Since data can arrive at many ports simultaneously, one SP module is needed for each port.

Once a packet has been divided into cells, the next step is to perform the address lookup in the Content Addressable Memory to determine which port the packet should be sent to. However, since many packets of data can arrive at the same time on different ports, and only one lookup can be performed at a time, these cells must be placed into some kind of intermediate storage while they await their destination. The next module in the switch is where this buffer is located, and it is called **Cell Buffer Ingress** or **CBI**. This buffer memory can either be one large memory that all ports share or each port can have its own dedicated memory. If the memory is shared between the ports, then the SP modules will need to use some sort of buffer for their cells, since only one cell can be written to the memory at any given time. In this case, some sort of basic scheduler will also be needed to ensure that all ports have equal access to the buffer memory.

The next module in the chain is the one called **Packet Processing** or **PP**. This module can be split into two parts called **Ingress Packet Processing** (**IPP**) and **Egress Packet Processing** (**EPP**) respectively. The IPP module is responsible for performing the address lookup in the content addressable memory to determine the destination port of the packet. IPP also determines which priority level the packet has. This priority level is used by a scheduler to determine in which order packets should be sent out of the switch. The EPP module is responsible for modifying the packet in any way deemed necessary by the switch before it is ready to be sent out to its destination port. Since the scope of this thesis is limited to the buffer management of the switch, the details of exactly how or why a packet should be modified, are not relevant to the reader. It is sufficient to say that IPP determines the destination port and prioritization of the packet, and EPP may or may not modify the packet itself before it is sent out. IPP and EPP can be two distinctly separate modules in the switch that are placed in different areas, or they can be directly connected and viewed as a single unit. In the first case, there needs to exist some kind of intermediate buffer for the cells that are output by IPP, where they can reside before being sent to EPP.

The cells output by EPP are then ready to be sent to their destination port. However, since the destination port may already be busy outputting the data of another packet, the cells must yet again be placed into a buffer. This is the role of the fourth module, **Cell Buffer Egress** (**CBE**). This module holds packets and forwards them to their destination port according to their prioritization level. Just like the CBI module, CBE can exist as one singular buffer that is shared by all the ports, or each port can have its own dedicated buffer.

The final module in the chain is **Parallel to Serial** (**PS**). This module takes the cells from CBE and converts them back into a serial stream of bytes that are sent out to the ethernet port. Consequently, the switch needs one of these modules per port, just like the SP module.

This overview of the switch and how packet data flows through it represents only a conceptual abstraction of an ethernet switch, not an actual implementation. There exist many different architectures that vary from the system described here, with different modules, layouts, and terminology.

One thing of particular import to understanding an ethernet switch architecture is the fact that packets that enter the switch for the first time do not have a corresponding entry in the CAM, which again means that they will be flooded to all ports except the one through which they entered. This means that the output bandwidth of the switch suddenly becomes larger than the input bandwidth since more packets will leave the switch than enter it. The consequence of this is that the memory buffers inside the switch will start to fill up with cells more quickly than they can be sent out. Thus, the switch will potentially use all of its available memory and become full. When this happens, incoming packets must be dropped. Given this, it becomes impossible to design a switch that will always be able to process every single packet sent to it, no matter the size of its internal memory. The actual architecture of the switch will determine when, how, and why packets must be dropped.

2.2 IC technology

When creating integrated circuits and systems that use them, there exist different ways and methods that can be used to produce chips. These methods differ from each other in significant ways and have a significant impact on factors such as production cost, performance, and size of the actual IC. Understanding the differences between these technologies is essential to understanding the importance of the area of an IC-design. Therefore, the following sections describe two of the most common technologies called ASIC and FPGA.

2.2.1 ASIC

Application Specific Integrated Circuits are chips that are custom made for a specific application. The actual method of production for asics is highly complex and comprised of hundreds of steps, but the general idea of the process can be summarised as follows [4]:

- Silicon is extracted from sand and quartz and is purified into electrical grade silicon.
- The silicon ingot is cut into thin disks called wafers.
- These wafers are placed under beams of ultraviolet light that travel through several photolithographic masks.
- When this light hits the wafer it undergoes a chemical reaction that "etches" the design onto the wafer.
- The design is printed onto the wafer this way as many times as it will fit.
- Finally the wafer is cut into several chips (or dies).

Modern chip production technologies allow for production of integrated circuits where a single transistor has a width of several micrometers down to as little as five nanometers [5].

Since ASIC chips are printed onto circular wafers, the actual size of the design that is being manufactured is of crucial import, for several reasons. Firstly, given a silicon wafer of a fixed size, the smaller the chip that is being produced is, the more chips will be able to fit onto a single wafer. This means that the price per chip will shrink if the total area of each die is lowered. Secondly, given the rectangular shape of each die, there will always be a certain area of the circular wafer along its edges where no chip will fit. This area will go to waste during production. However, a smaller chip area means that the effective silicon area that can be utilized on the wafer will increase since the smaller die resolution leaves a shape that is closer to approximating a circle, as can be seen in Figure 2.3. Lastly, the photolithographic process that etches designs onto the wafer will leave imperfections that can manifest themselves as faults in the dies. Given that a certain number of points on the wafer will statistically be incorrectly produced, the area of the chip itself will determine how many chips are likely to be faulty on a given wafer. The ratio of functional chips to faulty ones is referred to as the die yield, and a smaller design will lead to a higher yield as can again be seen in Figure 2.3. Since the production methods for ASICs are so complex, the costs are very high. The photolithographic masks need only be produced once for the fabrication of a single design, but each mask may cost as much as over one million dollars to produce, and a different number of masks will be required depending on which ASIC technology is used. In addition to this, each wafer of chips also costs large amounts, which means the price per chip will be large unless the chips are produced in large quantities. Thusly, reducing the area of the IC design can drastically affect the production cost for each chip.

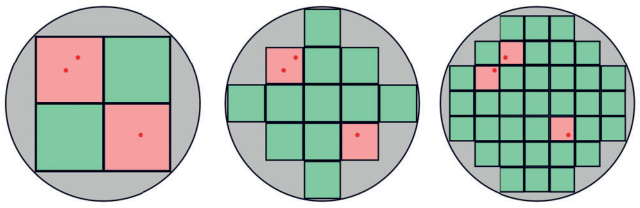


Figure 2.3: An illustration of how the silicon area of the design affects the die yield, given the same three defective spots on the wafer.

2.2.2 FPGA

An alternative to using fully custom chips is to use available off-the-shelf chips. A **Field Programmable Gate Array** is a chip that is programmable and can load an IC design onto it. Given that ASIC fabrication will only yield a relatively low price per chip for large production volumes, FPGAs stand as an alternative and are often used for prototyping designs. An FPGA is typically composed of several elements such as lookup tables, flip-flops, I/O-ports, as well as random access memory units [14]. Each FPGA is often more expensive than a mass-produced ASIC, however, if the planned production volume is low, using FPGAs could potentially be a more economically sound choice. These FPGA chips exist in various sizes and the larger the FPGA is, the more it will cost. Given this, the size of the design that will be uploaded to the FPGA will decide how large the FPGA needs to be. Given the fact that several different IC designs can be placed together on a single FPGA (commonly referred to as a System on Chip), the area of any particular IC such as a switching core, must often be constrained to a particular subsection of the actual FPGA.

In summary, optimizing the area of the switch design can be of crucial importance, leading to lower production costs, less silicon waste, higher yields, and better use of available silicon area. As such, the area of the final switch design was measured using several different tools that accommodate both ASIC and FPGA technology.

2.3 Design tools and measurements

Throughout this thesis work, several different tools and technologies were utilized to implement and measure different aspects of the switch. The following section gives a brief introduction to these tools, what roles they fill, as well as which measurements these tools yielded.

2.3.1 MyHDL

When designing integrated circuits, one often makes use of a Hardware Descriptive Language (HDL) that, just as the name suggests, describes the hardware of the IC and its behavior. These languages allow one to write Register Transfer Level (RTL) code describing signals and behaviors which can then be synthesized into actual logic gates. One of the most common HDLs that is used is called Verilog which is described in the IEEE standard 1364 [11]. This language is integrated into the workflow at Packet Architects, however, it is not the actual language that is used for hardware implementation. One key feature that Verilog lacks that is necessary to build a tool like FlexSwitch, which is capable of generating designs based on a specification, is the capability to parameterize aspects of the design. Since this is the case, another language is used that allows for parameterization, which can then be used to generate Verilog code that is then used for simulation and synthesis. The tool of choice is called MyHDL [12], which is a package that allows one to use Python as a Hardware Descriptive language. MyHDL lets one define signals and bit vectors whose behavior can be described as both combinational or sequential logic using traditional Python function definitions that are annotated to describe things such as clocking and reset behavior. A design written in MyHDL can be simulated directly in Python using built-in MyHDL features, or it can be converted into Verilog (or VHDL). In the case of this thesis, all the designs were implemented in MyHDL, which was converted to Verilog and then simulated using a more sophisticated simulator called Verilator. Packet Architects provided pre-written tests that the designs were run through to verify the switch's behavior as well as all the necessary tools and scripts for performing the conversion from MyHDL to Verilog code. These scripts also provide measurements of the total memory instances of the design in bits. The generated Verilog code was finally used for synthesis using the tools described in the following sections.

2.3.2 Vivado

The first synthesis tool used was the Vivado tool suite developed by Xilinx, Inc. [13]. The tool suite includes an integrated development environment, a high-level synthesis compiler, and tools for simulation. In this thesis Vivado was used to take the generated Verilog design and target it to a Xilinx FPGA board, translating and mapping the RTL code to components on the board in the form of lookup tables (LUTs), flip-flops (FFs), as well as block RAMs. A Xilinx Virtex UltraScale+

XCVU13P FPGA [20][21] was used for the final measurements. This FPGA features three different types of RAM units: 18kb block RAM (RAMB18), 36kb block RAM (RAMB36), as well as larger 288 kb UltraRAM (URAM) [15][16]. Vivado was utilized to take measurements of the total component utilization on the FPGA board as well as a timing analysis that was used to measure the critical path and total slack of the design. These measurements were used to evaluate the potential usefulness of the final thesis architecture for the use with FPGA technology.

2.3.3 Yosys

The second synthesis tool that was utilized was an open-source project by the name of Yosys [17]. The Yosys framework can be used for Verilog RTL synthesis into an ASIC standard cell library. The standard cell library that was used in the thesis was the Oklahoma State University Library for 0.18μm TSMC technology[18] [19]. This tool was used to obtain measurements for the total gate equivalent area of the design for ASIC technology. However, since the thesis design instantiates specific memory instances, Yosys is not able to translate these memories into standard cells and will therefore put these modules into a "black box" which is not accounted for in the final measurements. This effectively means that the Yosys measurements are covering everything in the design except for the memories, which means that it can be utilized as an effective way to observe how the combinational logic as well as sequential logic in the form of regular flip-flops scale with the architecture for ASIC technology. This however means that the memories must be measured using a different tool. An overview of the design flow using the different tools is presented in Figure 2.4

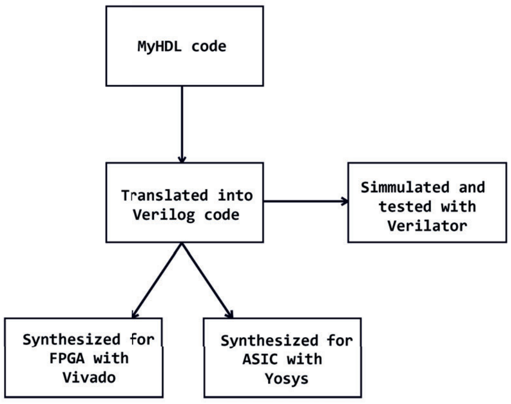

MyHDL code
Translated into Verilog code
Simmulated and tested with Verilator
Synthesized for FPGA with Vivado
Synthesized for ASIC with Yosys

Figure 2.4: An overview of the IC design flow.

2.3.4 Arm Artisan Embedded Memory IP

The final tool that was utilized was an embedded memory compiler developed by ARM Physical IP, Inc. by the name of Ultra High Density 2P SRAM SVT MVT Compiler. This compiler is part of the Arm Artisan physical IP product range and is capable of compiling SRAM memories for 16 nm TSMC ASIC technology [22]. This tool gave measurements on the total silicon area of the memory instances (in μm^2), which combined with the Yosys measurements gives comprehensive coverage of the total area design for ASIC technologies.

Chapter 3

Design and implementation

3.1 Design

3.1.1 IC design

Designing an IC can be a difficult process for many reasons. There exist plenty of pitfalls, and one must consider how every architectural decision will affect the design as a whole. One of the foundational principles of IC design that one must take into consideration is the fact that (almost) nothing comes for free. Every architectural decision is a trade-off, meaning that improvements in one area of the design will almost certainly come at the cost of another one. Therefore, one of the keys to a successful IC design lies in evaluating the resources at one's disposal and realizing which ones should be spared at the expense of others.

In the case of this thesis work, the key resource is quite obvious since the aim is to optimize the design for silicon area. In practice this means that since the performance requirements for the switch are quite lax, there exist situations where optimizations can be made that sacrifice performance in terms of throughput and latency, to gain a smaller area. Any decision that leads to area being gained by expending more clock cycles, should be considered optimal for the design. This principle served as the guiding motivation behind the selection and comparison of the different architectures, as well as most of the final architecture itself. Given this, the first optimization that was explored was the idea of shared-memory buffers.

3.1.2 Shared-Memory switches

In Section 2.1.2, the switch architecture overview, two different ways of structuring the buffers CBI and CBE are proposed, either with one buffer per port or as one shared memory. Even though the shared memory approach requires some extra external buffers and a scheduler to ensure fairness between ports, this approach has some major benefits when it comes to potential utilization. In the case where all input ports send packets to the same output port, the queue of outgoing packets will start to grow, since the output bandwidth on one port is lower than the bandwidth on all input ports combined. With separate buffers for each port, only the buffer of the port connected to the destination of the packets will be utilized, while all other buffers will sit empty. However, with a shared buffer the free space in the empty buffers could be utilized by the congested port. This does not eliminate the risk of packet drops, since the output buffer can grow indefinitely. It does, however, lower the memory requirements given a worst case it should be able to handle without packet drops. This is made clear in Figure 3.1, where less memory is needed to handle the data from a congested port. The figure illustrates that shared memories give better utilization of the memory area, meaning that the total area can possibly be reduced.

Switching architectures that implement a shared memory suffer from one main drawback that congestion on one output port could potentially occupy memory space needed for a different port, which in turn means that a packet that should have a ready path from its input to its output, instead gets dropped due to the high load on the first output port. This situation is not acceptable for a switch and must be dealt with in some fashion. To solve this issue, memory can be guaranteed for each port. The idea is to not let a single port use all the memory so that packets with a congested destination port will be dropped before a packet with an uncongested destination port.

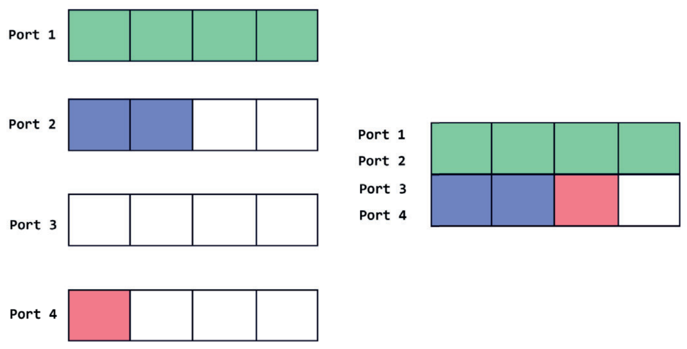


Figure 3.1: Dedicated vs shared memories.

In the case of CBI, using one shared buffer instead of separate ones does not yield as clear of a benefit. The system needs to be designed in a way so that the IPP module can handle packets at least as fast as packets can arrive on the input ports, which means that CBI can not grow indefinitely, and has a maximum amount of data it needs to be able to buffer, with or without shared memory. Packets are not allowed to be dropped before they have received their destination port and priority. This is because packets with a higher priority are never allowed to be dropped over packets with a lower priority.

There are however benefits to letting CBI and CBE share a physical memory. CBI needs to be conceptually separate from CBE since it needs to be guaranteed that none of its packets get dropped. This means that the two buffers can not use any space allocated to the other. However, in the case of shared memory between CBI and CBE, only one instance of the buffer manager is needed. Both of the buffers can use the same logic, to perform tasks such as selecting the next packet to be sent to a port or the packet processing module. Other parts of the manager, such read and write logic can also be shared between the two buffers. This effectively halves the bandwidth available on both the input and the output of the buffer. However, this is not a problem because of how the buffers handle data. The input on the ports is at most one byte per clock cycle, but the buffers handle data in cells, which consist of several bytes. Therefore, as long as the cell size is sufficiently large, the available bandwidth on the input and the output of the buffers are still enough to handle the total input bandwidth of the input ports. The idea of spending time to gain area is also perfectly in line with the aim of this project.

3.2 Selected architectures

Given the principle of shared-memory switches, three different architectures compatible with this idea were selected for evaluation and comparison. These architectures were analyzed with the goal of selecting the one offering the best potential gains in terms of area optimizations. The selected architecture was then used as a base to construct the final architecture that was implemented. The three selected architectures were a *Cut-Through* switch, an *Allocation/Defragmentation* switch, and a *Shared Memory Linked List* switch.

3.2.1 Cut-Through

When a packet enters a switch and is divided into cells, there exist different ways of transporting the data that comprises the packet through the switch. One of the most common methods is known as *Store and Forward* switching. In this method, the entire packet is treated as one unit, and every single byte of data must enter the switch before its cells can begin traveling through it. As an alternative to this, *Cut-Through* switching is a method where the packet can start traveling through the switch before the entire packet has finished entering it [7]. Figure 3.2 shows a conceptual representation of this architecture, with the same packet on the input port being sent out on an output port, and one cell of the packet in memory.

Implementing a Cut-Through switch would potentially offer several benefits, some of which could lead to silicon area optimizations. The most immediate benefit that is often associated with Cut-Through switches is the fact that they reduce packet latency. Since the switch starts to forward the packet data before it has fully entered the switch, the first cell will leave the switch through its destination port sooner than its Store-and-Forward counterpart. This benefit does not directly align with the design philosophy guiding the thesis architecture, which states that extra clock cycles should be spent in order to decrease the total area of the design. However, a side effect of cut-trough switching is that each packet being sent through the switch will not need to take up the size equivalent to the entire packet in buffers and memories that it travels through. If the packet is forwarded as soon as each cell is constructed, then every buffer would only need to be able to hold one cell, and likewise, only one cell would need to exist in the main memory at a time. This means that the size of the main memory as well as the various intermediate buffers could potentially be reduced.

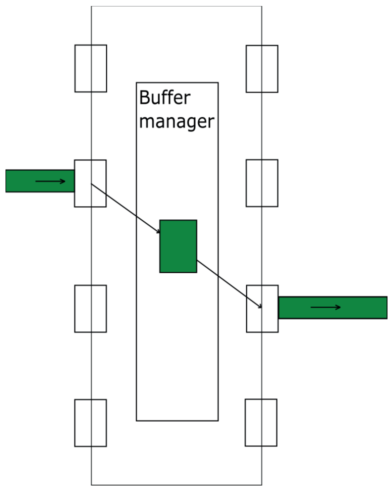


Figure 3.2: Overview of a cut-through switch, where a packet starts exiting the switch before it has fully entered it.

These benefits made this architecture a suitable candidate for further study. This analysis revealed several drawbacks that were taken into consideration. Firstly, a basic requirement of a Cut-Through switch is the fact that the bandwidth of the outgoing port of a packet must be equal to or lower than that of the incoming port. If the outgoing port sends out an entire cell as a stream of bytes before the next cell is ready to be sent to the port, then the outgoing packet will be left with a hole inside its data, which will lead to corruption. This is not a problem for the target switch of the thesis, since all ports are specified to have a constant uniform bandwidth, however, it would mean that this architecture would not be able to be scaled to another switch configuration with non-uniform bandwidths across the ports. Another potential issue is the fact that a cut-through architecture imposes stricter timing and scheduling requirements for every part of the switch. Given the fact that a single packet can simultaneously be entering and exiting the switch on different ports, there can be no gaps in the flow of data from the input port to the output port, and every single cell must be delivered to the destination at strictly timed intervals to ensure the continuity of the data flow. This means that more sequential and combinational logic in the form of cycle counters and schedulers would be needed, which servers to make the architecture more complex. If the total area of silicon required to implement these complexities is lower than the total area reduction of the buffers and memory, then this architecture would still serve as a good candidate to be implemented.

However, upon further analysis, it was also discovered that there exists a scenario that negates most of the main benefit that a cut-thorough switch offers. In an ideal scenario, each packet traveling through the switch only needs to store a single cell in the switch's main memory at any given time. However, this only applies if the packets are traveling to mutually exclusive destination ports. Since a single port can only output the data of a single packet at any given time, if two or more ports are simultaneously sending packets to the same destination port, only one of them would be able to exit the switch, and the rest would need to be stored in memory until the port is free again. In the worst case, every single port except for one is sending packets to the same port, meaning that only one packet receives the benefit of only storing one cell in memory at a time, and all the others must store full-size packets. Since the switch needs to be dimensioned for this worst case, it would not be possible to reduce the size of the main memory to any significant extent. This revelation made this architecture an unlikely candidate for implementation.

Nevertheless, even though the worst case scenario makes significantly reducing the size of the main memory impossible, the idea of reducing the size of intermediate buffers was still deemed a good idea. For instance, since a shared memory between ports was planned to be used, there needed to exist a buffer between the **SP** and **CBI** modules so that cells could be buffered in case they arrive simultaneously to the ingress buffer. Using the idea of a cut-trough switch, one could dimension this intermediate buffer to only be one cell large, and construct the packets cell by cell inside the CBI memory, rather than waiting for the entire packet to enter the switch, and then writing it into memory all at once.

3.2.2 Contiguous allocation/Defragmentation

The next architecture that was selected deals with a specific method for how packets and cells should be efficiently stored in memory. The main memory of the switch can be viewed as a long array of cells. When a packet is written to memory, it must be allocated a number of these memory cells, which will store the packet data. There exist several different methods for allocating and deallocating memory to packets that offer different kinds of benefits and drawbacks. The main idea of this architecture is that a packet will always be allocated a list of cells that are contiguous in memory. In essence, this means that starting from the address of the first cell of the packet, the rest of the packet can be found by simply iterating over the following addresses until the last cell is reached. If the cells were not contiguously allocated but rather spread out in a fragmented pattern all across the memory at different addresses, one would need a second link memory that for every cell in the packet keeps track of at which address the next cell can be found. This means that every packet would require memory equivalent to one address pointer per cell in the packet. If the cells are instead contiguously allocated, these links are unnecessary, and instead, only one pointer to the first cell and one to the last would be needed for every packet. Alternatively, one could use one pointer to the first cell, and store a value indicating how many cells long the packet is.

Given that this benefit directly decreases the amount of memory required for keeping track of packets in memory, and thus reduces area, this allocation method was deemed a suitable candidate for evaluation. Upon analyzing this architecture, two different methods were discovered for handling contiguous allocation.

Method one - Traditional allocation algorithms

The first method is to use one of several traditional memory allocation algorithms that are commonly found in software. Two of the most common of these would be a *linked-list allocation* algorithm or a *buddy system allocation* algorithm [8] [9].

The main idea of these algorithms is that they keep track of all the memory regions that are free in some kind of data structure which is iterated over to find suitable regions to be allocated. In the case of a linked-list allocator, the data structure takes to form of a linked list just as the name implies, while a buddy system allocator splits the memory into different tiers and groups together regions of the same size. Since these memory regions consist of a contiguous series of cells, each region only needs one pointer to the starting cell and one to the final cell (alternatively a size variable). When a packet is to be allocated a memory region, this data structure of free memory which henceforth will be referred to as the *free list*, is iterated over and a suitable region is returned. This iteration will either need a very large combinational net of logic gates so that the entire data structure can be searched in one clock cycle or a single or small subset of regions can be processed in one cycle. This approach would yield a smaller net of logic gates at the cost of taking more cycles to find a region for allocation. Both of these algorithms are quite complex, and must in addition to this manage the free list by splitting regions into smaller pieces and mergeing together different regions into larger blocks based on different circumstances. This complexity is not an issue when implementing such an algorithm in software, but given that hardware implementations of algorithms cannot rely on the principle of sequential execution, it would inevitably take longer to implement such an algorithm compared to other simpler alternatives discussed below.

Another prerequisite of this type of allocation algorithm is the fact that the entire packet size must be known at the start of the algorithm, since a contiguous memory region must be found that will fit the entire packet. This means that the approach discussed in Section 3.2.1 of building packets in memory one cell at a time and thus minimizing the size of the input buffers, would not be compatible with this approach. The final drawback of this approach that was discovered, was the fact that in a worst case scenario where the entire memory is free and has been split up into the smallest possible size of one cell per region, then the size of the free list data structure would grow to one entry per cell in the memory. Given the fact that one of the main benefits of this method was the fact that the data structure would only need a single pair of pointers per memory region, this worst case unfortunately invalidates that benefit since the switch would need to be able to handle this worst case scenario. This combined with the complex nature of the allocation algorithms meant that this approach was not deemed a good candidate to reduce the area of the switch.

Method two - Defragmentation

The second method to handle contiguous allocation would be a much simpler approach. The idea is to keep a single pointer to an address in memory beyond which every cell is free. Again the size of the packet would need to be known when allocation happens, but the actual allocation algorithm is as simple as allocating this address as the first cell, and then moving the free-pointer forward as many steps as the packet contains cells. In essence, this pointer would then mark the region of allocated memory, and everything below it must be considered allocated.

In theory, this seems like a great idea since it not only greatly simplifies the allocation algorithm, but it also practically eliminates the free list and replaces it with a single pointer, and thus reduces the silicon area. However, this approach introduces a new problem. Whenever a packet is sent out from memory, its cells need to be deallocated. The problem lies in the fact that the only way to reclaim a region and add it back to the memory that can be allocated, is to move the free-pointer backward, which means that this can only happen if the packet which left the memory was the one that is located last in memory. There is however no guarantee that the packet that is situated last in memory will be sent out first, which means that every time another packet is sent out, that memory region will become lost and will not be able to be utilized until the pointer is moved back to that region again. This means that the allocated region of the memory will become fragmented and full of holes that cannot be utilized.

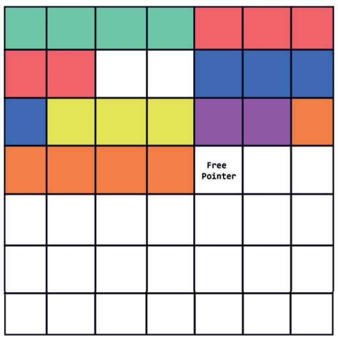


Figure 3.3: A gap in the data in the main memory.

A potential solution to this problem which was considered was to employ some sort of defragmentation algorithm that could move packets around in memory and push packets together in such a way that the gaps would be filled. Consider the following example as illustrated in Figure 3.3 where the colored cells represent different packets in memory and the white cells are free. There exist two different alternative algorithms that can be employed to fill the gap between packets. The first involves iterating over all the remaining packets in the memory after the gap and searching for one that fits the dimensions of the hole. Given that this iteration happens in hardware, either a very large combinational net of logic gates would be required to find a packet of the correct size (also consider that this size will vary depending on the size of the gap) in one cycle, or a subset of potential packets would need to be searched through every cycle, which means that the process takes longer but requires a smaller combinational net. When a suitable cell is found, it needs to be read and written into the cells that constitute the gap. The cells that this packet previously occupied in memory must simultaneously be deallocated. If the packet which was transferred was not the last packet in memory, then this means a new gap has now formed and the process would need to be repeated recursively, filling each hole with a new packet. Unfortunately, there is no guarantee that a packet of the correct size will exist in memory to fill the gap, which means that the second algorithm would need to be utilized. This method involves shifting all data after the gap backward to fill the empty cells. This means that each cell that resides at a higher address than the gap would need to be read and written back to another position. This process therefore takes as many clock cycles as there are occupied cells after the gap.

Whenever these kinds of data transfers occur inside the memory, both the read and write facilities would be occupied, essentially blocking both the input and output to and from the memory, meaning that this would potentially lower the bandwidth of the main memory significantly. Both of these methods take up tens if not potentially hundreds of clock cycles each, at which point it would perhaps have become more efficient to simply send the packets out of memory instead, thus shrinking the block of allocated memory. This combined with the fact that method one was deemed infeasible, meant that this architecture was also not considered an optimal choice for implementation.

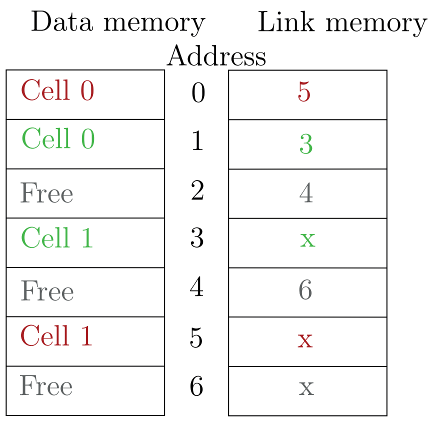
3.2.3 Shared memory linked-list

Since the method of allocating cells contiguously in memory was deemed an unfavorable approach, the alternative of allocating cells in memory and linking them together was instead explored.

Chao, H. J., & Liu, B.[10] present several implementations of switches utilizing shared memory buffers, without contiguous allocation. One of these architectures achieves this by using linked lists to link the addresses of cells belonging to the same packet. This linked list approach consists of using lists to link both cells in a packet as well as packets in a queue. In essence, memories are constructed that contains pointers to memory addresses, that can point to the other cells and serve as links between different memory elements. The main memory of the switch is split into two segments, one where the actual data cells are stored, and one where links between the cells of a packet are stored. This second link memory needs to have as many entries as there are cells in the main memory. An additional memory could be used to keep track of a linked list of all the cells that are free to be allocated in the memory. Keeping one pointer to the start of this list, and one to the end means that allocation becomes quite simple. Each cell that is to be written to memory is simply allocated the first cell of the free list, and the pointer to this free list head is then updated by reading which address that cell points to in the free list memory. This means that the cells of one packet could potentially be spread out across different addresses that are not contiguous. Removing the requirement of contiguous allocation means that fragmentation of data in the memory no longer becomes an issue at all since all free cells will be accessible through the free list, which gives perfect utilization of the memory on the cell level.

The size of these memories needing one pointer per cell initially seemed like it would be a problem until it was discovered that the equivalent data structure discussed in Section 3.2.2 would be as large, needing two pointers per cell (or one pointer and a size value). Another exciting possibility that was discovered which makes this even less of an issue is the fact that these two memories that link packets and the free cells, can be combined into a single memory since the subset of cells that are allocated will never overlap with the subset of cells that are free. Thus one unified link memory can be kept meaning that in total only one pointer per cell is needed. There is one problem with this approach, however. When a cell is written to memory, both the link that represents the packet and the link representing the free list need to be updated simultaneously. Likewise, when cells are read from memory and deallocated, both the link from the packet and the free list must be read. In essence, this means that these link memories would need to be able to perform two reads and two writes simultaneously if reads and writes to the memory are to be able to occur in parallel. One solution to this would be to only perform reads or writes exclusively, but this would halve the bandwidth of the switch. Another solution would be to overclock this memory at double the clock frequency which means that, essentially, a memory with double the amount of ports is created. Given the clock frequency requirement of 12.5 MHz in the project specification, this means that this overclocked memory would run at 25 MHz instead giving a maximum critical path of $\frac{1}{25*10^6} = 4*10^8$s or 40 nanoseconds. So long as the design can meet this timing requirement then that means that this one unified memory could be used to keep track of all links and still be able to handle parallel reads and writes.

In Figure 3.4 an overview of the system is shown, with two packets in memory, one colored green, and one colored red, with the cells of each packet linked through entries in the link memory. The data in the link memory is the address of the next cell of the same packet. The figure also shows that the list of free entries is part of the same memory structure. Since this approach only allocates one cell at a time instead of a contiguous region, the size of the packet does not need to be known when allocation occurs, which means that the method of constructing packets in memory one cell at a time mentioned in Section 3.2.1, would be compatible with this architecture.



Data memory	Address	Link memory
Cell 0	0	5
Cell 0	1	3
Free	2	4
Cell 1	3	x
Free	4	6
Cell 1	5	x
Free	6	x

Figure 3.4: Memory structure, with one data memory and one link memory.

The unified link memory and the data memory will have the same number of entries and therefore share the same address space, however, the size of the data blocks will vary. In the part of the memory that stores the actual cells, the block size needs to be the size of the cells, while the link memory block size will need to be large enough to fit a pointer to an address in this address space. This means the cell size of packets will impact the size of the link memory since given a fixed data memory size, the cell size will determine how many entries the link memory has as well as how large each entry will be. A smaller cell size yields more addresses in the data memory, and therefore more link entries, and since the address space is larger, the size of each pointer will grow as well. However, a smaller cell size means there will be less memory wasted in the case where the last bytes of data don't utilize an entire cell, meaning that the memory utilization will increase as the cell size decreases. Finding the optimal cell size will therefore be of great import to optimizing the size of the memories.

In addition, each packet needs a data structure to store information about where in the memory its cells are located. This data structure will henceforth be referred to as a *header*. This header can be constructed in several ways, for instance by storing just one pointer to the first cell of the packet. Any cell could still be accessed since all cells have a link to the next in the packet. To find the last cell, which is needed to append cells, the whole packet would need to be iterated through. Another way to structure the header is to store both a pointer to the first and last cell, which would eliminate the need to iterate through the whole packet each time a cell is appended. Choosing between the two approaches takes careful consideration between minimizing the memory required and the amount of clock cycles required for appending one cell. Important to note is also that the maximum number of cells a packet could consist of determines the maximum number of clock cycles needed to append a cell to a packet.

Given the fact that this architecture gives perfect utilization of every cell in the memory without the need for complex defragmentation algorithms, combined with the simplicity of allocation and deallocation, the fact that it is compatible with the method of building packets in memory one cell at a time, and the fact the the link memory can combine the free list as well as the packet links, the shared memory linked-list architecture was finally chosen as the base upon which to build the final thesis architecture.

3.3 Final architecture

Given the benefits and drawbacks discussed in Section 3.2, the architecture actually implemented in this project is mainly a shared memory, linked list architecture, with ideas taken from the cut-through approach. Mainly the idea of letting just a small part of a packet stay in a buffer, and sending the first part of the packet to the next module before the last part of the packet has reached the buffer.

The architectural overview of a switch discussed in Section 2.1.2, is a good starting point when discussing the actual implementation in this project. In the overview data only flows one way, from input to output via both buffers and packet processing modules. On a conceptual level the implementation works in the same way, however, the CBI and CBE buffers share one physical memory contained in a module that was named *Buffer Manager*, much like the concepts discussed in

Section 3.1.2. This large shared buffer will be further explained in the coming section.

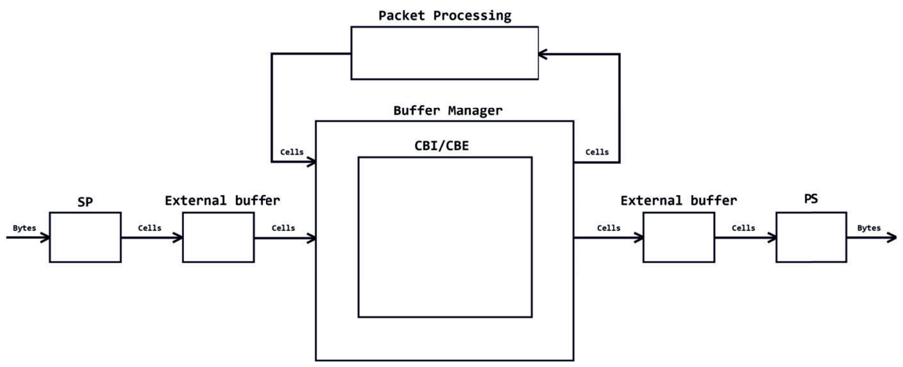


Figure 3.5: An overview of the final thesis architecture.

3.3.1 Buffer manager

As discussed in Section 2.1.2 packets move through the switch from input ports to output ports via intermediate buffer storage and packet processing modules. Because of the fact that the buffer manager contains both the CBI and CBE, packets need to be buffered twice, both on the way in, when it is waiting to be processed by IPP, and on the way out from IPP to the actual output port. As seen in Figure 3.5, the output to and input from IPP can be seen as any other input or output port. This effectively adds another port on each end of the buffer manager. This is in line with the goal of the project, where the same hardware can be used for several purposes. However, since each packet has to go into the buffer two times, the bandwidth is halved. In this architecture, the buffer manager module can handle one cell per clock cycle on its input, which would set the limit on the total bandwidth on all input ports combined to not be more than one cell per two clock cycles, because of the fact that each packet goes through twice. The worst case is when every input port has incoming data on a given clock cycle. Therefore, for the switch to function, Section 3.1 needs to hold. Where N is the number of ports on the switch, and C is the cell size in bytes.

$$N \leq \frac{C}{2} \tag{3.1}$$

The switches implemented in this project will have four and ten ports respectively, which means that a hard lower limit on the size of cells exists. For the switch with four ports the limit is 8 bytes, and 20 bytes for the ten-port configuration. However, this calculation does not take into account the fact that the buffer manager needs clock cycles to handle the packets inside the switch when no input is permitted, meaning that in practice the cell size will need to be at least slightly larger than this.

The main part of the buffer manager is the cell memory. This is the memory where the actual data passing through the switch is stored. This is, however, not the only memory needed to build a buffer utilizing shared memory and linked lists. As discussed in Section 3.2.3, a memory that stores the links between cells is also needed. The link memory has as many entries as the actual cell memory and has the purpose of representing the links between cells of the same packet. The content of a memory entry is an address to another cell in the same address space and represents the address to the next cell of the packet. This can again be illustrated in Figure 3.4, where, for example, cell 1 is followed by cell 3, which means that at address 1 in the link memory exists the value 3. Every cell in the main memory will either belong to a packet, or it will be free and not have any valid data stored in it. As discussed above, due to the fact that this memory is overclocked at twice the speed, this link memory can keep both the links between the cells in a packet, and the free list. The free list needs additional external information in the form of the addresses of its head and tail, to allow appending elements to the list at the same time as reading the next free element. Another benefit of having a link memory with both packets and the free cells in it is the fact that deallocation of a packet becomes quite simple. Because of how the cells are linked in the memory, deallocating an entire packet is as easy as deallocating a single cell. Since the cells of the packet are already linked, only one link needs to be set, the link between the tail of the free list, and the first cell of the packet that is being deallocated.

The idea of having a link memory separate from the data memory is also used in other parts of the buffer manager, namely in situations where an ordered list of elements is needed. In the case of this architecture, it is needed to keep track of the packets. Both in keeping information about each specific packet in their headers and in maintaining queues, to make sure that the packets are processed in the correct order.

Packet headers

In this implementation of the architecture, a data structure is used to keep track of information about packets, called headers. It consists of the following 7 fields:

- **Destination port**, a bitmask indicating which ports the packet has as its destination.
- **Source port**, the port the packet arrived on.
- **Valid bytes**, how many bytes are valid in the last cell of the packet.
- **Last pointer**, a pointer to where the last cell is stored in the cell memory.
- **First pointer**, a pointer to the first cell.
- **Ready to send flag**, one bit indicating whether or not the packer has all its cells in memory and is ready to be sent.

Upon entry of the first cell of a packet into the buffer manager, it is assigned a header. The last cell field of the header is updated each time a new cell arrives from the same source port until a last flag is set on the input port. Then one bit is toggled in the header, which signals that this packet is ready to be sent to the packet processing module or output port. Packets can come interleaved with one another, one cell at a time. Therefore, each source port gets its own header allocated, so there exist as many allocated but unfinished headers, as the switch has input ports. The header memory is also organized in a similar way as the data memory, with one part consisting of the actual headers, and another part being the links between them. In the memory, there exists both a list of free elements as well as links between headers that represent an ordered list of packets. This memory is also used to keep track of the links between the queue of packets that are awaiting to be sent to the packet processing module, in exactly the same fashion as the cell link memory. However, once a packet returns from the packet processing module and has received its destination port, the possibility exists that the packet will be sent out on multiple ports. This means that the packet must then be placed in several queues simultaneously. Since the header link memory can only link a packet to one other packet, that means that this memory cannot also be used to maintain the queues for the output ports. This is instead handled in another memory, the packet queue memory. This is also split into two parts, one containing the address of the packet header, and one linking these entries together to form the queues. This means that a single header address can be written to this packet queue memory more than once, and the links will be written to link it to the next packet in each respective queue. An overview of the different memories contained inside the buffer manager is presented in Figure 3.6.

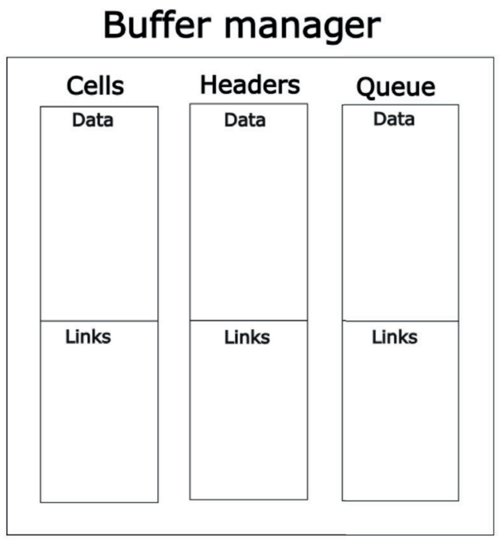


Figure 3.6: An overview of the different memories that reside in the buffer-manager module.

Initially, the headers were built using flip-flops, however, early results showed that this approach did not scale well when the number of headers increased. The number of headers needed in this switch is discussed later in Section 3.3.3. Instead, the headers were remade using memory modules, since large memory modules should be constructed out of memory blocks. The behavior of the original switch that was constructed using flip-flops was tested and verified. However, modifying the new version to produce identical behavior to the first unfortunately fell outside of the time frame for the thesis. But in order to ensure a fair comparison of the thesis architecture to its FlexSwitch counterparts which are constructed using memory instances instead of flip-flops, the measurements were taken on a switch that uses memory instances that equal the size of the original flip-flop counts.

The results therefore come from a switch that was not tested, but given minor adjustments it would work the same way as the fully tested version, with flip flops as header memory. This choice was made since the goal of this project is to minimize and compare area, which means that fair area comparisons take precedence over a fully tested design.

Packet priority

One of the requirements from Section 1.3 is that the switch should be able to handle priority between packets on the same port. For this functionality to be implemented each port needs to have several queues, both to separate packets with a different priority, and to keep the ordering between packets with the same priority, since it is important that packets that arrive first are also sent out first. As mentioned above, queues are structured in a similar fashion to how the data and link memory are structured for cells, and the header data and header link memory are structured for headers. The queue memory is also split into two parts, one memory that holds the actual data, in this case, the entries are headers, and one that holds the links between the packets. In the same way as with the data link memory, the queue link memory also contains a list of all free entries. The same principle of allowing multiple ports to utilize a shared memory was employed here, rather than giving each port its own dedicated instance of these queue memories. This means that the total area of this shared memory could be reduced. When this packet queue memory is full, packets will need to be dropped.

Packet drop

On some occasions when a cell arrives on an input port to the buffer manager, it needs to be dropped. This means all cells currently stored in the buffer that belong to this packet need to be deallocated, and all future cells of this packet ignored. This can happen in a few different circumstances. The first and most obvious one is that the memory is full, there are simply no free cells available in the buffer memory. Another reason to drop a packet is due to guarantees. Every destination port has a part of the memory guaranteed, in order to avoid starvation of ports when other ports are congested. There are therefore situations where a packet is dropped even though there exist free cells in the buffer memory, however, they are guaranteed to another destination port. A third reason to drop a packet is when all the entries in the queue memory are depleted.

3.3.2 External buffers

Most of the storage capacity of the switch lies within the buffer manager, but external buffers are also required for one main reason. The buffer manager can only handle one input at a time, while all ports could have data coming in simultaneously. Some sort of intermediate buffer is required between the SP module and the buffer manager. In this architecture, a one cell large ingress buffer is used. No more than one cell is needed since the packets are allowed to come interleaved into the buffer manager. The same case of needing extra buffer capacity is true for the output as well, because of the same reasons as on input. However, the external egress buffer is larger, it has capacity for one whole packet. This is due to the fact that the PS module needs to have a continuous stream of cells so that there are no gaps between bytes sent. Because of this, there exists a strict deadline for the buffer manager or an external buffer to send the next cell to the PS module. This would not be a problem if it were not for the way in which the IPP module receives data. It needs all cells of a packet sent continuously, which means that a packet sent out to IPP could potentially make a packet going out on an actual port miss its deadline. Therefore, in this architecture, the egress buffer has enough capacity to hold a whole packet.

A similar situation as what can happen on output, with the IPP port breaking a strict deadline of another port, can also happen on input. However, there are no actual deadlines for the packet to meet on input in the same way as on output, where a missed deadline means an incomplete packet is sent out on the network. The buffer manager can handle arbitrary amounts of clock cycles between cells of the same packet. Input from IPP, on the other hand, comes one cell per clock cycle until the whole packet has been written into the buffer manager. Cells from this port are also prioritized since they can not be buffered between IPP and the buffer manager. The issue with this approach is that a situation can arise where cells on an input port are lost, which would mean that a packet would need to be dropped even though there is free capacity in the buffer manager. This situation happens when an ingress buffer has a cell waiting to enter the buffer manager, and if this cell is not read before the next cell arrives from the SP module, the first cell is simply overwritten. If this were to happen, the packet would no longer be complete and would need to be dropped. However, there are criteria for when this situation can occur. A packet coming from IPP is never allowed to take more clock cycles to read than it takes clock cycles for SP to construct one whole cell. In other words, the longest packet can never contain more cells than one cell contains bytes. This is however not enough for all deadlines to be met, since the buffer manager only handles input from one port at a time, the time taken to read all ports needs to be taken into consideration when setting a lower limit on the cell size. The condition in Equation 3.2 needs to be true for the timing to work, where N is the number of ports, C is the size of a cell in bytes, and 1522 is the largest possible packet in bytes.

$$N + \left\lceil \frac{1522}{C} \right\rceil \leq C \tag{3.2}$$

For four ports the lower limit on the cell size is then 41 bytes, and the same number for ten ports is 45 bytes.

3.3.3 Cell size optimization

An important decision when designing the specifics of the implementation is choosing an appropriate cell size because it affects the area of the switch in several ways. Firstly the SP and PS modules need to be able to store at least one cell, which means that a larger cell size scales the size of those modules linearly. However, a larger cell size would decrease the number of cells needed to have the same amount of data in memory. This means that the link memory would not need as many data entries, and as a result, the pointer to the next cell address does not need to be as large since the address space has decreased. In summary, both the number of pointers as well as the size of those pointers decrease with a larger cell size. However, with a larger cell size the utilization of the memory may decrease. Since the memory can not be split up further than into the size of a cell, each packet will in the worst case use $cellsize - 1$ bytes more memory than it actually needs.

The total size of the parts of the memory affected by the cell size can be calculated by summing the memory of each module. Both the SP and PS modules need to be able to store one cell each. In this architecture, the first ingress buffer also needs to buffer one cell per port. The egress buffer is not affected by the cell size since it is always able to buffer one whole packet, no matter the size of the cells. The total size of the buffers outside of the buffer manager in bytes can be expressed as Equation 3.3, where x is the size of one cell, and N is the number of ports.

$$M_{buf}(x) = 3Nx \tag{3.3}$$

The total required size of the cell memory for the switch to be able to pass the full overlap test described in Section 1.3, as a function of the cell size is described in Equation 3.4. This is known since given a fixed number of ports, N, a set maximum number of packets are in the switch simultaneously during the full overlap test.

$$M_c(x) = \left\lceil \frac{1522}{x} \right\rceil \left(\left(\sum_{i=1}^{N} i\right) + N\right)x \tag{3.4}$$

If the size of the memory in the buffer manager module is constant, then the cell size mainly affects the size of the cell link memory, since the address space grows with a smaller cell size. With a total memory size M, the number of cells is $\frac{M_c}{x}$ and the number of bytes required to represent the whole address space is $\frac{\lceil \log_2 \frac{M_c}{x} \rceil}{8}$. The size of the cell link memory in bytes can then be derived as Equation 3.5.

$$M_{c\_link}(x) = \frac{M_c}{x} \frac{\lceil \log_2 \frac{M_c}{x} \rceil}{8} \tag{3.5}$$

Before the optimal cell size can be found by solving Equation 3.11, an appropriate number of headers needs to be chosen. The upper bound for the amount of headers needed to never cause a packet drop, with a minimum packet size of 64 bytes, depends on the cell size. If the cell size is larger than the minimum packet size, the number of headers is the same as the number of cells in memory, while if the cell size is smaller than 64, the number of headers is the number of cells in memory divided by how many cells a packet of 64 bytes needs allocated. The number of required entries for the headers memory is then calculated as 3.6

$$H = \frac{\frac{M_c(x)}{x}}{\lceil \frac{64}{x} \rceil} \tag{3.6}$$

The number of headers is therefore dependent on the cell size, which in turn, means that all memories dependent on the number of headers are also dependent on the cell size. Much like the cell memory structure, the headers also have a link memory. The address space of the header link memory is the same as the header memory, and each entry in the header link memory is an address in the same address space. Hence, the total size of the header link memory is described by Equation 3.7

$$M_{h\_link}(x) = \frac{H\lceil \log_2 H \rceil}{8} \tag{3.7}$$

All headers store 2 pointers each, one to the first cell of the packet and one to the last. With H headers, the size of the header memory can be then obtained from Equation 3.8

$$M_h(x) = H\frac{\lceil \log_2 \frac{M_c}{x} \rceil}{4} \tag{3.8}$$

The queues are also a memory that is dependent on the number of headers, which means that its size can be described as a function of the cell size. Much like the headers and cells, the queue memory is also split into two parts, with one data memory and one link memory. The size of the data memory can be found in Equation 3.9, and the size of the queue link memory can be found in Equation 3.10. Since a header can exist in several queues, the queue memory needs to be larger than the header memory. In this architecture, the queue memory was scaled up by a factor of 2 compared to the header memory.

$$M_q = 2H\frac{\lceil \log_2 H \rceil}{8} \tag{3.9}$$

$$M_{q\_link} = 2H(\frac{\lceil \log_2 H \rceil}{8} + 1) \tag{3.10}$$

Given this, the total combined size of every memory instance S expressed as a function of the cell size x can then be derived as Equation 3.11.

$$S(x) = M_{buf} + M_c + M_{c\_link} + M_h + M_{h\_link} + M_q + M_{q\_link} \tag{3.11}$$

According to Equation 3.11, with the number of ports set to 4 and 10, the two configurations have an optimal cell size of 61 bytes and 153 bytes respectively.

However, these calculations only take into consideration the memory instances of the switch, and leave out all logic and some registers. These numbers were used as starting points in experiments that were used to optimize the total area of the implementations.

Memory area

It is important to note that not all memory bits are equal when it comes to silicon area. The physical area required to instantiate a memory is dependent both upon the technology used as well as the structure of the memory. There are several ways to create a single memory cell, with different numbers of required transistors, as well as different properties [23]. However, in the general case, a larger memory is usually more area-efficient per bit stored since there exists some overhead no matter the size, which becomes a smaller portion of the total area with a large memory. Furthermore, in this architecture, the link memories need to be clocked on a faster clock, in order to be able to read and write on several addresses each clock cycle. With a faster clock comes stricter demands on the critical path, and a large memory can based on the structure, violate the timing requirements, since a large memory uses more area which makes the signals travel longer distances. As with many other aspects of hardware design, this takes careful consideration, where a benefit almost always comes with a drawback. With this in mind, further optimizations can potentially be made on the memory level when the exact technology used for the specific hardware implementation of the switch is known. In conclusion, it is important to note that not all memories are created equal, and the exact configuration can have an impact on the total area of the switch, especially the part of the switch this thesis is focused on, the buffering and forwarding of packets.

Chapter 4

Result and analysis

4.1 Total size of buffer management modules

In order to ascertain how large the buffer management modules are in relation to the total switch, measurements were taken of the two generated FlexSwitch switches which are presented in Table 4.1, 4.2, and 4.3. These measurements demonstrate that the buffer management modules constitute a significant part of the total switching area. The switches generated by FlexSwitch included the packet processing module, while the thesis implementations did not contain such a module. However, in the results presented below, the packet processing module has been subtracted from all the measurements of the FlexSwitch switches, in order to ensure that only the buffer management modules were compared.

Config.	LUT	FF	RAMB36	RAMB18
FlexSwitch 4 ports	20308/62144(33%)	18813/59162(33%)	11/18(61%)	2/2(100%)
FlexSwitch 10 ports	32719/90693(36%)	34032/89500(38%)	15/22(68%)	2/6(37%)

Table 4.1: Vivado measurements of the size of the buffer management modules of the FPGA implementations in relation to the total size of the FlexSwitch switches.

Configuration	Gates	Flip-Flops
FlexSwitch 4 ports	228121/781193(29%)	18459/61183(30%)
FlexSwitch 10 ports	419950/1201549(35%)	33714/90050(37%)

Table 4.2: Yosys measurements of the size of the buffer management modules of the ASIC implementation in relation to the total size of the FlexSwitch switches.

Configuration	Memory bits	Memory area (μm)
FlexSwitch 4 ports	242804/421844(58%)	23591/45149(52%)
FlexSwitch 10 ports	1162544/1389458(84%)	60893/77456(79%)

Table 4.3: Memory measurements of the size of the buffer management modules in relation to the total size of the FlexSwitch switches. The memory bits are the same for both the FPGA and ASIC implementations, however, the memory area only represents the ASIC implementation.

4.2 Implementation results

The final architecture was implemented and synthesized into both an FPGA and ASIC implementation for a four and ten-port configuration respectively. The tools mentioned in Section 2.3 were utilized to measure the area for these implementations as well as the equivalent switches produced by the FlexSwitch tool suite. Only the modules relating to the buffer management of the switches were used when comparing the designs.

4.2.1 Vivado

The measurements obtain through Vivado for the FPGA implementation are presented in Table 4.4 and Figures 4.1,4.2, and 4.3.

Configuration	LUT	FF	RAMB36	RAMB18	URAM
Thesis 4 ports	11312	9198	7	0	0
FlexSwitch 4 ports	20308	18813	11	2	0
Thesis 10 ports	35499	27992	3	1	7
FlexSwitch 10 ports	32719	34032	15	2	0

Table 4.4: Measurements obtained from Vivado for four and ten ports respectively.

The results of the timing analysis obtained from Vivado are presented in Table 4.5. The table presents the maximum delay path between the two different clocks that the system uses, the core clock which is clocked at 12.5 MHz and the overclocked fast clock which is clocked at 25 MHz.

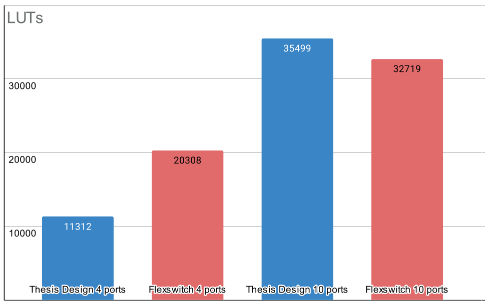


Figure 4.1: Number of FPGA LUTs used vs. configuration

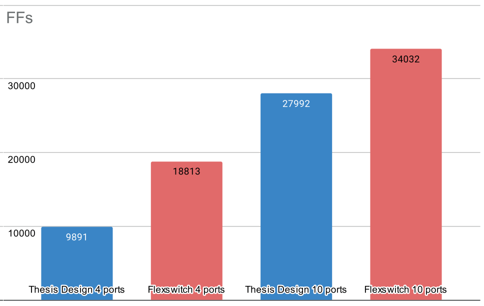


Figure 4.2: Number of FPGA flip flops used vs. configuration

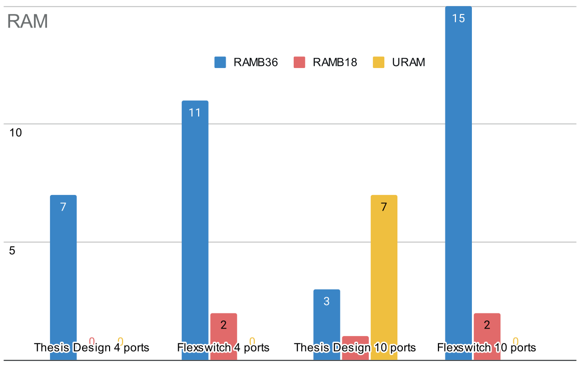


Figure 4.3: Ram blocks for each configuration

Configuration	core-core	core-fast	fast-core	fast-fast
Thesis 4 ports	8.67 ns (71.34 ns slack)	7.448 ns (32.504 ns slack)	4.21 ns (35.8 ns slack)	4.082 ns (35.928 ns slack)
Thesis 10 ports	18.984 ns (61.025 ns slack)	14.111 ns (25.610 ns slack)	7.336 ns (32.675 ns slack)	5.369 ns (34.293 ns slack)

Table 4.5: Measurements obtained from Vivado for the different maximum delay in nanoseconds of the critical paths for the different clock combinations of the design.

4.2.2 Yosys

The measurements obtain through Yosys for the ASIC implementation are presented in Table 4.6 as well as Figures 4.4, and 4.5.

Configuration	Total gate count	Total flip-flops
Thesis 4 ports	111771	8699
FlexSwitch 4 ports	228121	18459
Thesis 10 ports	333637	24733
FlexSwitch 10 ports	419950	33714

Table 4.6: Measurements obtained from Yosys for four and ten-ports respectively. (Flip-flops are included in the total gate count).

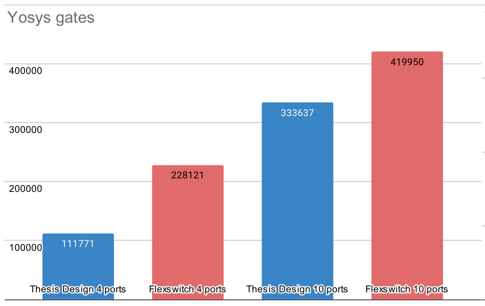


Figure 4.4: Gate equivalents generated by Yosys for each configuration.

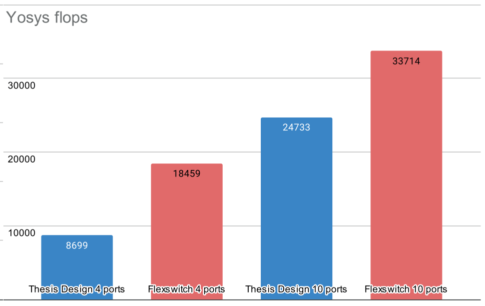


Figure 4.5: Flip flops generated by Yosys for each configuration.

4.2.3 Memory instances

The number of memory bits as well as the memory area for each configuration utilize obtained through the Arm Artisan Embedded Memory compiler are presented in Table 4.7 and Figures 4.6, and 4.7.

Configuration	Memory bits	Memory area (μm^2)
Thesis 4 ports	236575	16286.67432
FlexSwitch 4 ports	242804	23591.07288
Thesis 10 ports	1013263	34413.82632
FlexSwitch 10 ports	1162544	60893.1324

Table 4.7: Memory measurements in bits and in silicon area.

4.3 Result analysis

The results presented in chapter four demonstrate that the thesis architecture is successful in its aim of area optimization for the four-port configuration, but also that the ten-port configuration seems to scale relatively poorly. The FPGA implementation of the four-port configuration shows a decrease in lookup table utilization, needing only 55% of the amount that FlexSwitch uses. Similarly the

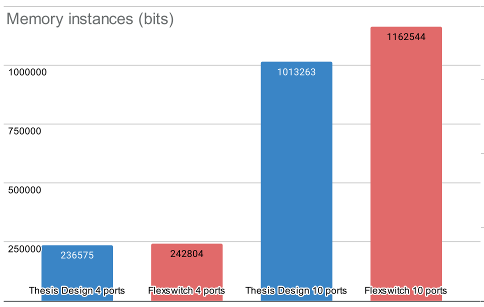


Figure 4.6: Number of bits of memory each configuration use.

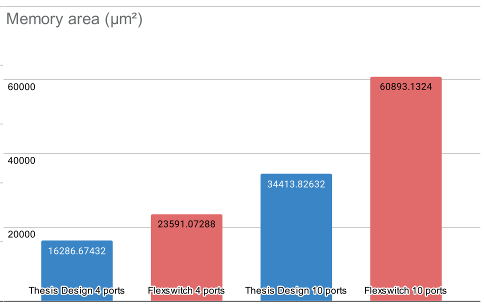


Figure 4.7: The total area of the memories in μm^2 for each configuration.

flip-flop count stands at 53% of the FlexSwitch counterpart, and RAM usage is down needing only 7 36kb blocks instead of 11 and 0 18kb blocks instead of the FlexSwitch's 2. Comparing the utilization of different blocks of RAM is not entirely straightforward, and given the complexity of Vivado, one cannot make many assumptions regarding the inner workings of its compilers and how the RTL code is mapped to the FPGA hardware, meaning that it is relatively difficult to draw conclusions regarding how one should compare the utilization of the different types of RAMs. However, since the four-port configuration uses fewer blocks in both the 36 and 18 kb categories, it is safe to say that the total memory area has been reduced. This conclusion seems to be corroborated by the fact that as can be seen in Table 4.7, the memory instances of the four-port configuration uses 6229 fewer bits than its FlexSwitch counterpart.

The FPGA results for the ten-port configuration, however, are less straightforward. Even though the design uses only 82% of the flip-flop count of the ten-port FlexSwitch, lookup table usage has actually increased by 8%. Similarly, the RAM usage looks to have worsened since the thesis implementation uses a whole 7 units of the larger 288 kb URAM blocks, whereas the FlexSwith uses only 36 and 18 kb blocks. Granted, the FlexSwitch uses 15 blocks of 36 kb blocks and 2 18kb blocks, compared to the thesis implementation using 3 and 1 blocks respectively, however, given the much larger size of the URAM blocks, the total memory usage of the thesis implementation seems much higher. This is puzzling since Table 4.7 again shows that the memory instances of the ten-port configuration uses a whole 149281 fewer bits than FlexSwitch.

Looking instead at the results for the ASIC version in Table 4.6 and 4.7, then it can again be observed that the thesis architecture demonstrates significant reductions in area. The memory instances of the four-port configuration take up 69% of the equivalent FlexSwitch area, and uses 49% of the FlexSwitch logic gates. Meanwhile, the ten-port configuration also came out smaller using 57% of the memory area and 79% of the logic gates compared to FlexSwitch.

These results suggest that the thesis architecture enables significant area reductions for both ASIC and FPGA implementations for a small switch such as the four-port configuration. Scaling up the switch configuration to a larger number of ports still gives beneficial results for ASIC implementations, but the results seem to indicate that the architecture has trouble scaling up for FPGAs.

The results of the timing analysis presented in Table 4.5 demonstrate that the timing demands are met with plenty of slack for the specified 12.5 MHz clock frequency. This frequency gives a bandwidth of $12.5 * 10^6 * 8\text{bits} = 100\text{Mbits/s}$ per port. The core-core clock domain allows for a theoretical maximum delay of 80 ns while the other three involving the fast clock allow for 40 ns since the fast clock is clocked at twice the frequency. Given this result, dividing the maximum measured delay path by the theoretical maximum allows one to calculate the factor by which the frequency could be multiplied while still meeting the timing requirements. The most constrained clock domain of the four is the core-fast domain which for the four-port configuration allows for a speedup factor of $\frac{40}{7.448} = 5.37$. This means that the switch could be scaled up to handle port bandwidths of over 500 Mbits/s using this FPGA technology. The ten-port configuration yields a maximum speedup of $\frac{40}{25.610} = 1.56$, meaning that this switch could be scaled up to handle around 150 Mbits/second.

4.4 Architecture analysis

The architecture itself has several aspects that directly serve to benefit the thesis goal of area reduction, as well as certain limitations that future work could potentially reduce. For starters, one of the main design principles that was heavily utilized in the architecture is the method of combining different memories into a large shared unit. If many ports can share a single memory buffer, that allows one to avoid duplicating logic for managing reads and writes to different memories. This also allows for better utilization of the existing memories, since one heavily congested port can utilize memory space that would otherwise be exclusively reserved for another port, even though that memory could potentially be unused at that moment.

This principle of shared memories is also observed to be applicable when two different components of the switch can share the same memory if they will exclusively occupy distinct memory regions that are guaranteed not to overlap. Given the fact that this is the case for the memory regions that the free list and the packet data will occupy, having these two share the same link memory, eliminates an entire memory of equivalent size that would otherwise have been needed. However, this would not be possible for certain higher-end switch specifications, since the memory needs to be overclocked to twice the regular clock frequency to enable this kind of memory sharing. This is the case, since both the free list and the read/write processes that handle the links between cells in a packet, must be able to independently access and modify the content of the memory concurrently.

Another decision that benefits the overall area is the decision to design the architecture around a method of allocation that does not explicitly need to know the size of the entire packet at the time of allocation, but rather allocates space for the individual cells as they come. This allows for the switch to utilize buffers that are only one cell in size rather than buffering the entire packet after the SP module, before they can be written to memory. Combining this principle from the

Cut-Through architecture with the rest of the thesis design seems to have directly reduced the overall area of these modules.

Constructing the data path through the packet processing modules IPP and EPP as a loop with the buffer manager also allowed for certain area optimizations. Structuring the architecture in this way comes at the cost of halving the output bandwidth of the BM module, but this is completely fine for a lower-end switch such as the ones implemented here, given the fact that the bandwidth requirements for these switches can still be achieved. This allowed the flow of packets to and from the packet processing modules to be treated almost exactly the same as the flow of packets to and from the real switch ports, which in turn allows for a lot of the internal logic of the buffer manager to be reused. A slight drawback to this approach is the fact that since both the CBI and CBE modules now reside in the same memory, certain data values will need to be saved unnecessarily. A packet that conceptually resides in the CBI module, will not need to store a bitmask representing its destination ports, since these values are not yet known, but it will need to store its source port. Likewise, a packet that resides in the CBE module will not need to know its source port, but it does need to know its destination ports. In the case of the thesis switch, both packets will still need hardware capable of storing both of these values even though only one will be needed for any given packet, which would not be the case if CBE and CBI were completely separate modules. It would be interesting to implement another version of the architecture that separates these two modules to compare the total sizes so that the question of whether the area saved by the reuse of logic of the thesis design outweighs the unnecessarily stored data.

Finally, the decision to use the shared-memory linked list architecture as a base, not only greatly simplifies the allocation algorithm that can be utilized, and thus reduces area in the form of the gates required to implement more complex algorithms, but it also grants the switch perfect utilization of the memory at the level of cells. This completely eliminates the issue of fragmentation, since cells can be allocated in any order so long as the links between them are set. This does however come at the cost of needing more pointers in the form of the link memory, which means that these results are insufficient grounds to draw the direct conclusion that this architecture is definitively smaller than an architecture that implements a more complex algorithm that utilizes some form of contiguous allocation. This could serve as a starting point for future work, where such an implementation could be compared to the thesis architecture so that more definitive conclusions could be drawn regarding how the different modules of the architecture impact the total size of the switch.

The architecture implemented in this project had the aim of being suitable for switches with low bandwidth and few ports. For a switch with more ports the SP module and its interface with CBI would need to be altered. In the case with many ports, a lot of demand would be put on writing to CBI, since there are a lot of parallel data paths into the buffer. There would be an inevitable bottleneck, either at the writing to CBI, if it would be a shared buffer, or to IPP if the switch would have one buffer per input port. FlexSwitch is designed to be able to generate switches with different amounts of input ports. Therefore the architecture generated by the tool has a comparatively large module for parallelizing data and writing it to the ingress buffer. However, for small switches with few ports, there is no problem in letting each input port get more direct access to the CBI. This saves on both logic and buffer area, compared to the design generated by FlexSwitch, and other architectures capable of handling many parallel input ports.

In Section 3.3 the relation between the number of ports and cell size was discussed, specifically regarding the bandwidth and timing when writing to the CBI. The highest bandwidth used in this project is 10 ports with capacity for one byte each clock cycle, which gives a total input bandwidth of 10 bytes per clock cycle. The buffer manager module can handle one cell per cycle but needs to process each packet twice, which gives it a maximum capacity of half a cell per clock cycle. This means that there is a discrepancy between the input and output bandwidth of the buffer manager. However, this surplus of bandwidth on the input of the buffer manager results in the module being idle. The main idea of this project is to trade clock cycles to save area, and with the final implementation, there are still clock cycles to spare. One way of utilizing these extra clock cycles was discussed in Section 3.2.2, however, those methods were deemed infeasible or unnecessary. In future works, different ways of utilizing these extra clock cycles could be explored, either in ways to further reduce the silicon area of the switch or perhaps in ways to increase performance. For example, one approach could be to shorten the critical paths in the implementation, in order for the switch to be able to handle higher bandwidths. Although this is entirely dependent on the technology used, it always has the potential to increase the maximum bandwidth the switch can handle.

Chapter 5

Future work and conclusions

5.1 Future work and improvements

5.1.1 External buffers

The thesis work has yielded several ideas regarding potential improvements to the architecture. Currently, there are two major drawbacks to the design that could in theory be eliminated if the architecture was developed further. The first of these is the fact that currently, each packet that is to be sent out to its destination port is first sent to an external egress buffer that sends one cell at a time to the PS module. This decision was taken since it greatly simplifies the logic of outputting packets from the buffer manager and allows packets being sent to the PP module to be handled in the same way as packets being sent to output ports, which allows for the reuse of existing logic. However, this means that these external egress buffers must be exclusively dedicated to a single port and this memory will be unavailable to other ports that could potentially utilize it if the memory were shared. If logic was implemented that handled the sending of cells directly from the buffer manager to the PS modules, then the need for these external buffers would be eliminated. This is not to say that they could be completely removed from the design, as their memory capacity would still be needed in the calculation of the minimum required memory of the buffer manager. Essentially these buffers would instead become a part of the main buffer memory which would increase its potential utilization and possibly enable it to be slightly reduced in size. However, this would require much more complex logic with counters and a sophisticated scheduler ensuring that each PS module receives its cell at precisely the correct clock cycle, so that no gaps appear in the output data, corrupting the packet.

5.1.2 Headers

Another area of improvement in the design can be found within the headers memory. Currently, this memory stores lots of information regarding each packet in the switch, and given the fact that packets can be as small as two cells in the current design, that means that this memory must have the capacity to store as many entries as there are cells in the memory divided by two. Given the large depth of this memory, reducing the size of each entry would yield large reductions in its area. Currently this memory stores pointers to the first and last cell of the packet, however upon further analyzing this design in conjunction with the engineers at Packet Architects, it was revealed that these pointers could removed from this memory if certain alterations were made to the architecture. The first pointer is needed in order to be able to start reading a packet from memory so that it can be sent either to the PP module or out of the buffer manager to its destination port. Currently, the logical queues that are implemented in the thesis architecture store pointers to the address of the header of the packet that lies in the queue, which is then used to index the header memory and retrieve the address of the first cell of the packet. However, instead of storing a pointer to the address of the header, these queue memories could instead directly store the address of the first cell, which removes one level of indirection and allows for the first pointers to be removed from the header entries. The pointer to the last cell of the packet is needed for two different reasons. The first is that when a packet is being written to the buffer manager, each cell arriving needs to be linked to the previous cell of the packet. That means that a pointer is required to keep track of which cell is the current last cell of the packet so that this link can be written to the link memory. However, this pointer does not necessarily need to reside in an entry in the header memory. Instead, there could exist one such pointer per port that keeps track of the address of the last cell that was written. Having one pointer per port instead of one per header entry would greatly reduce the number of pointers that are stored. The second reason the pointer is needed is so that the buffer manager can determine when all cells of a packet have been read. An alternate solution to this problem is to replace the last pointer in the header entry with a much smaller memory containing a one bit flag for each cell in the memory which could be read to determine if the cell is the last in its package or not. Using one bit per cell instead of one address per header would also reduce the total number of memory bits utilized so long as the header pointer is larger than two bits. If these two alternatives were implemented, then the last cell pointer could be completely removed from the header entries which would greatly reduce its size.

5.1.3 Overclocking

Another area that could be explored further is the ways in which overclocking could be utilized. The thesis architecture utilizes overclocking to double the ports of certain memories in order to ensure that the free list and cell links can share the same memory. Another way that overclocking can be utilized is not to expand the number of ports, but rather the shorten the data width of the memory. As has been previously discussed in Section 3.3.3, a single bit of data in a memory will not always occupy the same amount of silicon area, but instead, be dependent on how that memory is constructed. Given that the data width of the memory will in some way affect the total area that the memory will need to occupy, overclocking the memory and writing a subsection of the data comprising the total data width on each overclocked cycle, could potentially alter the size of the memory. Given the low clocking requirements of the switch, this is another avenue that could be pursued to potentially reduce the switch area even further.

5.1.4 CBI and CBE separation

Future work studying an alternate version of this architecture where CBI and CBE are separated into two modules would be able to more conclusively answer questions regarding the design decisions made during this thesis. Additionally, implementing an architecture that utilizes contiguous allocation and comparing it to the thesis architecture would also be of great benefit, as suggested above.

5.2 Conclusion

This thesis project aimed to compare different architectures for L2 ethernet switches in order to produce an architecture that was optimized for silicon area, that could be utilized for smaller low-end switches. The resulting measurements of the switch implementation allow one to conclude that the architecture that was produced during this thesis, succeeds in terms of optimizing the switch's silicon area for smaller port configurations, and could be utilized to produce a smaller switch compared to contemporary alternatives such as the FlexSwitch architecture. However, even though the ASIC implementation of the architecture performed well for both the four and ten-port configurations, the observation that the architecture currently seems to not scale well for FPGA implementations when the number of ports increases, suggests that further work is necessary, exploring some of the above suggested improvements, before the architecture could be deemed generally useful for low-end switches.

References

[1] *IDC's Worldwide Quarterly Ethernet Switch and Router Trackers Show Continued Growth in Third Quarter of 2022*. International Data Corporation. `https://www.idc.com/getdoc.jsp?containerId=prUS49948322`. (Accessed May 8, 2023)

[2] N. Farrington et al. *Data Center Switch Architecture in the Age of Merchant Silicon*. 17th IEEE Symposium on High Performance Interconnects. New York. NY. USA. 2009. pp. 93-102

[3] E. Bastos et al. *MOTIM - A Scalable Architecture for Ethernet Switches*. IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07), Porto Alegre, Brazil, 2007, pp. 451-452, doi: 10.1109/ISVLSI.2007.70.

[4] *From Sand to Silicon "Making of a Chip" Illustrations*. Intel. (2011). `https://download.intel.com/newsroom/kits/chipmaking/pdfs/Sand-to-Silicon_32nm-Version.pdf`. (Accessed May 11, 2023)

[5] *Logic Technology*, Taiwan Semiconductor Manufacturing Company, `https://www.tsmc.com/english/dedicatedFoundry/technology/logic`, (Accessed May 11, 2023)

[6] K. Pagiamtzis, A. Sheikholeslami. (2006). *Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey*. IEEE Journal Of Solid-State Circuits. vol. 41, no. 3, p. 712,

[7] *Switching modes: Store-and-Forward vs Cut-Through*. Network Academy. `https://www.networkacademy.io/ccna/ethernet/store-and-forward-vs-cut-through-switching`. (Accessed May 31, 2023)

[8] B. Kernighan, D. Ritchie. (1988). *The C Programming Language* (Second ed.). Prentice Hall P T R. p. 185-189

[9] D. Knuth. (1997). *The Art of Computer Programming. Vol. 1* (Third ed.). Addison-Wesley. p. 435–455.

[10] H. J. Chao, B. Liu (2007). *High Performance Switches and Routers*. John Wiley Sons Inc. p. 207-213.

[11] *IEEE Standard for Verilog Hardware Description Language*. IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001). pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.

[12] *Overview*. MyHDL Community. https://www.myhdl.org/start/overview.html. (Accessed May 16, 2023)

[13] *Vivado Overview*. Xilinx, Inc. https://www.xilinx.com/products/design-tools/vivado.html. (Accessed May 16, 2023)

[14] *Understanding FPGA Architecture*. Xilinx, Inc. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/odz1504034293215.html (Accessed May 16, 2023)

[15] *BRAM and Other Memories*. Xilinx, Inc. https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/jbt1504034294480.html. (Accessed May 19, 2023)

[16] *Xilinx Power Estimator User Guide (UG440)*. Xilinx, Inc. https://docs.xilinx.com/r/en-US/ug440-xilinx-power-estimator/UltraRAM-support-for-UltraScale-Devices. (Accessed May 19, 2023)

[17] *Yosys Open Synthesis Suite*. YosysHQ. https://yosyshq.net/yosys/about.html. (Accessed May 16, 2023)

[18] *Flows/FreePDK45*. Oklahoma State University. https://vlsiarch.ecen.okstate.edu/flow/. (Accessed May 16, 2023)

[19] G. Petley. *Illinois Institute of Technology, Oklahoma State University Cell Libraries*. https://www.vlsitechnology.org/html/libraries04.html. (Accessed May 16, 2023)

[20] *proFPGA Xilinx Virtex UltraScale+ XCVU13P FPGA*. Xilinx, Inc. https://www.xilinx.com/products/boards-and-kits/1-18q1pyl.html. (Accessed May 16, 2023)

[21] *Virtex UltraScale+*. Xilinx, Inc. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html. (Accessed May 16, 2023)

[22] *Artisan Embedded Memory IP*. arm. https://www.arm.com/zh-TW/products/silicon-ip-physical/embedded-memory. (Accessed May 16, 2023)

[23] K. Eshraghian, N. H. E. Weste (1993). *Principles of CMOS VLSI Design* (Second ed.) Addison-Wesley. p. 563-579

Printed by Tryckeriet i E-huset, Lund 2023

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-920
http://www.eit.lth.se