DESIGN, IMPLEMENTATION AND VERIFICATION OF A HIGH-SPEED ON-CHIP PACKET SWITCH

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AYHAN SEFA YILDIZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2022

Approval of the thesis:

DESIGN, IMPLEMENTATION AND VERIFICATION OF A HIGH-SPEED ON-CHIP PACKET SWITCH

submitted by **AYHAN SEFA YILDIZ** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** _____

Prof. Dr. Şenan Ece Güran Schmidt
Supervisor, **Electrical and Electronics Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Cüneyt F. Bazlamaçcı
Computer Engineering, İzmir Institute of Technology _____

Prof. Dr. Şenan Ece Güran Schmidt
Electrical and Electronics Engineering, METU _____

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering, METU _____

Prof. Dr. Ali Ziya Alkar
Electrical and Electronics Engineering, Hacettepe University _____

Assist. Prof. Dr. Serkan Sarıtaş
Electrical and Electronics Engineering, METU _____

Date: 09.02.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Ayhan Sefa Yıldız

Signature :

ABSTRACT

DESIGN, IMPLEMENTATION AND VERIFICATION OF A HIGH-SPEED ON-CHIP PACKET SWITCH

Yıldız, Ayhan Sefa
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Şenan Ece Güran Schmidt

February 2022, 77 pages

In this thesis, an on-chip packet switch architecture to interconnect modules on System on Chip (SoC) platforms at high line speeds is proposed. The particular target application for the proposed on-chip switch is hardware accelerated cloud computing systems. To this end, FPGA Accelerator Cards (FAC) are employed in heterogeneous cloud data centers which implement hardware accelerators on the FPGA. The data from the cloud user is brought on the accelerators and delivered after processing through high-speed Ethernet Interfaces on the FAC. The FPGA has other modules such as memory modules and SoC processor for supporting the cloud services. To this end, a high-throughput on-chip packet switch is required to interconnect heterogeneous interfaces. Furthermore, the switch design should be scalable and configurable to meet the dynamically changing demands of the cloud data center.

The contributions of this thesis are the design, verification and evaluation of an on-chip packet switch that addresses these requirements. The switch is an input-queued switch that operates at line rate to support scalability. The number of

ports, the data width and buffer sizes are parametrized and configurable. To the best of our knowledge, there is no on-chip switch implementation presented together with its systematic verification.

The on-chip switch design is implemented on the XC7Z100FFG1156-2 SoC of the Xilinx Zynq-7000 family. The pipelined hardware architecture and the memory organization are described in detail. The systematic verification is carried out using the SystemVerilog infrastructure. We demonstrate that the switch supports 100% throughput at 40 Gbps line speed and a maximum latency around 1250 nsec by making use of the statistics collected by SystemVerilog in Modelsim tool.

Keywords: On-chip switch, switch fabric arbitration, cloud computing, verification, coverage

ÖZ

YUKSEK HIZLI YONGA ÜSTÜ PAKET ANAHTARI TASARIMI, GERÇEKLEŞTİRİMİ VE DOĞRULAMASI

Yıldız, Ayhan Sefa
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Şenan Ece Güran Schmidt

Şubat 2022 , 77 sayfa

Bu tezde, yüksek hat hızlarında System on Chip (SoC) platformlarındaki modülleri birbirine bağlamak için bir yonga üstü paket anahtar mimarisi önerilmiştir. Önerilen yonga üstü anahtar için özel hedef uygulama, donanım hızlandırmalı bulut bilişim sistemleridir. Bu amaçla, FPGA üzerinde donanım hızlandırıcıları uygulayan heterojen bulut veri merkezlerinde FPGA Hızlandırıcı Kartları (FAC) kullanılmaktadır. Bulut kullanıcısından gelen veriler hızlandırıcılara getirilir ve FAC üzerindeki yüksek hızlı Ethernet Arayüzleri aracılığıyla işlendikten sonra teslim edilir. FPGA, bulut hizmetlerini desteklemek için bellek modülleri ve SoC işlemcisi gibi başka modüllere sahiptir. Bu amaçla, heterojen arayüzleri birbirine bağlamak için yüksek verimli bir yonga üstü paket anahtarı gereklidir. Ayrıca, anahtar tasarımı, bulut veri merkezinin dinamik olarak değişen taleplerini karşılamak için ölçeklenebilir ve yapılandırılabilir olmalıdır.

Bu tezin katkıları, bu gereksinimleri karşılayan bir yonga üstü paket anahtarının tasarımı, doğrulanması ve değerlendirilmesidir. Anahtar, ölçeklenebilirliği desteklemek için hat hızında çalışan giriş tamponlu bir anahtardır. Bağlantı noktası

sayısı, veri genişliği ve arabellek boyutları parametrelendirilir ve yapılandırılabilir. Bildiğimiz kadarıyla, sistematik doğrulamasıyla birlikte sunulan bir yonga üstü anahtar uygulaması yoktur.

Yonga-üstü anahtar tasarımı, Xilinx Zynq-7000 ailesinin XC7Z100FFG1156-2 SoC ürününde uygulanmaktadır. Boru hattı donanım mimarisi ve bellek organizasyonu ayrıntılı olarak açıklanmıştır. Sistematik doğrulama, SystemVerilog altyapısı kullanılarak gerçekleştirilir. Modelsim aracında SystemVerilog tarafından toplanan istatistiklerden yararlanarak anahtarın 40 Gbps hat hızında 100% verimi ve 1250 ns civarında maksimum gecikmeyi desteklediğini gösteriyoruz.

Anahtar Kelimeler: Yonga-üstü anahtar, anahtar örgüsü çekişmesi, bulut bilişim, doğrulama, kapsam

To my family

ACKNOWLEDGMENTS

Before anything else, I would like to express my deepest thankfulness to my advisor Prof. Dr. Ece Güran Schmidt who has always supported me with her vast knowledge and motivation.

I would like to thank ASELSAN for its technical material support and for its permission to attend the classes.

I also thank TÜBİTAK for their support with TÜBİTAK-BİDEB M.Sc. scholarship.

Also, I would like to thank my mother Aysel, father Şeref, and brother Onur Yıldız who have always been by my side and supported me. I would like to thank my wife Seda Nur Yıldız who is with me during my intense thesis process and give peerless support.

I am thankful to my friends, Alper Yazar, Murat Akpınar, Emre Şahin, Ahmet Anıl Dursun, and Ahmet Faruk Akyüz for their support throughout the development and the improvement of this thesis.

This thesis was supported by the Scientific and Research Council of Turkey (TUBİTAK) [Project Code 117E667-117E668].

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ALGORITHMS xvi

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND AND PREVIOUS WORK 5

2.1 Packet Switching 5

2.1.1 Packet Switching Basics 6

2.1.2 Buffer Architecture 7

2.1.3 Fabric Arbiters 9

2.2 Network On-chip Packet Switches 11

2.3 SystemVerilog Verification 14

2.3.1 Testbench 15
2.3.2 Coverage 17
2.3.3 Assertions 19
2.3.4 DPI 19
2.4 Placement of the Thesis Work in the Literature 19
3 ON-CHIP SWITCH DESIGN 23
3.1 Hardware Architecture of Switch 23
3.1.1 VOQ Controller 27
3.1.2 Virtual Output Queue 28
3.1.3 Crossbar Fabric 30
3.1.4 Arbiter 31
3.1.5 Reassembly Controller 36
3.1.6 Reassembly Buffer 38
3.1.7 Reassembly Scheduler 39
3.1.8 Pipelined Switching Cycles 43
4 EVALUATION 49
4.1 Performance Evaluation 49
4.1.1 Verification of the On-chip Switch 49
4.1.2 Performance of the On-chip Switch 65
4.2 FPGA Hardware Implementation Evaluation 68
5 CONCLUSION AND FUTURE WORK 71
REFERENCES 73

LIST OF TABLES

TABLES

Table 2.1 Summary of Related Previous Work 21

Table 3.1 Bit Field of VOQ Flits 28

Table 3.2 Bit Field of RAB Flits 31

Table 3.3 On-chip Switch Pipeline Stages - 1 46

Table 3.4 On-chip Switch Pipeline Stages - 2 47

Table 3.5 On-chip Switch Pipeline Stages - 3 48

Table 4.1 Generated Test Input Flit Numbers for Input Ports 58

Table 4.2 FPGA Implementation Results of On-chip Switch 69

LIST OF FIGURES

FIGURES

Figure 2.1 Crossbar Fabric 7

Figure 2.2 An Example of HoL Blocking Problem 8

Figure 2.3 Switch Buffer Organization 9

Figure 2.4 Components of SystemVerilog Language 15

Figure 2.5 SystemVerilog Testbench Components 15

Figure 3.1 Switch Architecture 26

Figure 3.2 VOQ Controller Block Diagram 27

Figure 3.3 Virtual Output Queue Architecture 29

Figure 3.4 Crossbar Switch Block Diagram 30

Figure 3.5 Arbiter Block Diagram 32

Figure 3.6 Scheduler Cycles 33

Figure 3.7 RAB Controller Block Diagram 36

Figure 3.8 Reassembly Buffer Architecture 38

Figure 3.9 Reassembly Scheduler Block Diagram 40

Figure 3.10 Reassembly Scheduler Arbitration Example 42

Figure 4.1 Switch Testbench Architecture 50

Figure 4.2 Instant Console Display Messages of Verification Test 53

Figure 4.3 Verification Test Report of the First Experiment 57

Figure 4.4 Coverage Report for *source_coverage* 60

Figure 4.5 Coverage Report for *destination_coverage* 61

Figure 4.6 Verification Test Report for RAB with 113 flits 62

Figure 4.7 Verification Test Report for RAB with 227 flits 64

Figure 4.8 Average Flit Latency of On-chip Switch under Uniform Traffic . 66

Figure 4.9 Throughput of On-chip Switch under Uniform Traffic 67

Figure 4.10 Packet Drop vs RAB Depth 68

LIST OF ALGORITHMS

ALGORITHMS

Algorithm 1 Coverage Class Definition Example Code 18

LIST OF ABBREVIATIONS

ABBREVIATIONS	
3D	Three Dimensional
40G	40 Gbps
ACCLOUD	Accelerated Cloud
AXI	Advanced Extensible Interface
BRAM	Block RAM
BPS	Bit per Second
CVP	Coverpoint
DDR	Double Data Rate
DRAM	Distributed RAM / Dynamic RAM
DRR	Dual Round Robin
DPI	Direct Programming Interface
DUV	Design Under Verification
DUT	Design Under Test
FAC	FPGA Accelerator Card
FF	Flip Flop
FIFO	First In First Out
FMAX	Maximum Frequency
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GbE	Gigabit Ethernet
GPU	Graphics Processing Unit
HA	Hardware Accelerator

HACDC	Hardware Accelerated Cloud Data Center
HDL	Hardware Description Language
HDVL	Hardware Description and Verification Language
HOL	Head-of-Line
IEEE	The Institude of Electrical and Elecktronics Engineers
IP	Intellectual Property
iSLIP	Iterative Round Robin Matching with Slip
LAN	Local Area Network
LUT	Look Up Table
MHZ	Mega Hertz
NOC	Network-on-Chip
PCIE	Peripheral Component Interconnect Express
RAM	Random Access Memory
RA	Reassembly
RAB	Reassembly Buffer
RR	Reconfigurable Region
SOC	System-on-Chip
vFPGA	Virtual FPGA
VHSIC	Very High Speed Integrated Circuit Program
VHDL	VHSIC Hardware Description Language
UVM	Universal Verification Method
WNS	Worst Negative Slack
VCT	Virtual Cut-Through
VOQ	Virtual Output Queue
XBAR	Crossbar

CHAPTER 1

INTRODUCTION

Recently, in cloud computing services, hardware accelerators (HA) are also provided as computing resources alongside conventional cloud server resources such as memory, processor and disk [1, 2]. For this purpose, FPGA Accelerator Cards (FAC) cards containing pure FPGA or FPGA with a processor (SoC) are directly connected to the data center network without connecting to servers in the cloud data center [3]. Thanks to the partial reconfiguration feature of FPGAs, it allows for instantiation of HAs on demand. Thus, it is possible to present multiple HAs performing different tasks on the same FPGA.

In cloud computing systems, the FACs are sophisticated system on chip (SoC) platforms. These platforms incorporate processors, hardware accelerators, memory modules, and high speed Ethernet interfaces to enable receiving and delivering data from the cloud users. To this end, on-chip switching is necessary to facilitate data exchange among all these components. On the one hand, the heterogeneity of these components and the application characteristics indicate that rather than the switches for classical on-chip mesh networks that interconnect identical Processing Elements, a switch architecture that is similar to computer network packet switches in terms of buffer organization and fabric arbitration is more suitable. On the other hand, the on-chip communication based on the amount of data that is transmitted within one clock cycle, namely flit, should be maintained. Furthermore, the resource constraints of the on-chip implementation together with the advantages of high-speed data exchange via shared memory, registers and on-chip interconnects should be taken into consideration.

Today, with increasing computing capacity applications, the size and complexity of designs are increasing. This makes functional verification one of the most important parts of the design development process. The most time-consuming process in the design development process is design verification [4]. For design verification, the use of universal verification methods is increasing to speed up the functional verification and debugging process. SystemVerilog and Universal Verification Methodology (UVM) are widely used tools developed to enable the integration of designers' work and system-level design verification [5]. An on-chip switch runs under different types of workloads, with different arrival patterns and packet sizes. To this end, it is important to have a systematic verification procedure that ensures the functional correctness of the implementation [6]. Systematic verification refers to testing predefined features and comparing them to a gold model or gold results. During the testing phase, there should be functional coverage constraints that show that the device under test is tested in all desired conditions. A result or report is then obtained showing that all of these constraints are hit during the testing phase. In addition, the results of the tested design and the golden results must be the same. In this way, the design requirements are systematically verified.

In the work described throughout this thesis, we propose the design, implementation, verification and performance evaluation of an on-chip packet switch architecture that operates at the line rate of 40 Gbps and provides 100% throughput. The design is configurable in terms of the number of ports, the data width, and the amount of buffer memory. The on-chip packet switch design, implementation, verification and performance evaluation are presented for FACs to be used in cloud computing systems, which will enable communication between components implemented on FPGA.

In our proposed architecture, the FPGA Accelerator Card (FAC) has two 40 Gbps Ethernet interfaces that are implemented as IP Cores [7]. The first interface is to the connected cloud server and the second interface is to the data center network. There are four Reconfigurable Regions (RR) to implement hardware accelerators. There is an SoC processor, a DDR interface and a PCIe interface. Accordingly, the switch is designed with 9 input/output ports. All lines and the fabric work at the rate of 40 Gbps.

We implement an on-chip switch design on XC7Z100FFG1156-2 SoC of the Xilinx Zynq-7000 family as the target device and present the hardware resource and operating frequency results. The performance of our switch architecture is evaluated with simulations under different load scenarios. Furthermore, the design is verified using the SystemVerilog verification environment. Scoreboard values showing verification test results and coverage results with the ability to observe generated test data are also displayed.

To the best of our knowledge, the work in this thesis is a first work that addresses the design and systematic verification of an on-chip switch for SoC systems.

The rest of the thesis is organized as follows:

In Chapter 2, firstly, basic information about packet switching and on-chip packet switches is given. The reason for the head of line blocking problem and the buffer structure developed to solve this problem are explained. Then, basic information about fabric arbiter methods, which are widely used in the literature, is given. Next, the motivation for design verification to have a vital place in the design development processes is mentioned. Past studies on validation in the literature are described. Finally, the SystemVerilog verification process and the basics of the sub-elements used in the verification process are explained in detail.

The proposed on-chip switch design details are described in Chapter 3. First, the on-chip switch architecture is mentioned. Then, all sub-blocks that make up the design and the signals that provide communication between sub-blocks are explained in detail. It is also mentioned that the packet transmitted over the switch is separated into flits. Finally, the pipeline structure of the blocks on the switch is shown.

Chapter 4 describes the evaluation of the on-chip switch. While evaluating, the latency and throughput of the on-chip switch performance evaluation are made. In addition, in this section, the verification test method of the switch is explained in detail. Then, the verification test result and test coverage information are displayed.

Chapter 5 presents the summary of the work in the thesis and the work planned for the future.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

In this part, we give an overview and fundamentals of the on-chip packet switch and design verification process. To this end, we present the basic and important past work for on-chip switch arbiters in network systems. Then, we focus on SystemVerilog verification fundamentals and relevant previous studies. Finally, we explain the contribution of the studies within the scope of the thesis.

2.1 Packet Switching

Packet transfers have gained an important place in developing network structures recently. Many different systems transfer data within themselves or with other systems. In cloud networks, embedded systems, the internet, and real-time computer systems, packet exchanges with various features, large sizes, and high speeds are needed. In these cases, $packet\ switches$ that enable all end-points to communicate with each other are needed. Packet switches receive the packet from an input line and send it to an output packet [8]. For the packets to be successfully transmitted over the packet switches, there is some control information required in the packets because packet switches can only learn to which destination the incoming packets want to go [9]. Packet switches decide the priority or order of the packets is sent during data exchange with their arbitrary structure, which is a predetermined decision-making method. Ethernet switches, also known as LAN switches, and routers are the most well-known examples of packet switches.

2.1.1 Packet Switching Basics

The traditional packet switch structure has multiple input lines and multiple output lines. In a NxN packet switch architecture, the packet on each input line $i \in (0 \ldots N-1)$ can send packets with any output line $j \in (0 \ldots N-1)$. The output line to be sent is determined by the destination information in the package. Thus, a one-to-one connection is established between the input line and the target output lines. The dimensions of the packages to be sent may vary. For this reason, packets are divided into smaller pieces of fixed size throughout the packet switch to ensure a deterministic operation. Small pieces whose transmission is completed are reassembled before being sent over the output lines.

It can be seen that more than one input line wants to send packets to the same output line at the same time. Because packet switches can provide only a one-to-one match between input and output ports at a time, a $contention$ occurs in these cases. This problem is solved by two different methods. The first method is to exchange packets on the switch fabric faster than line speed C. In this case, the packet switch bandwidth should be equal to NxC value to prevent packet dropping. The fact that the packet switch connection speed depends on the number of ports N limits the scalability of the switch. In addition, in cases where the connection speed is higher than the line speed, buffers are needed to store the excess packets transmitted on the output lines. NxN packet switches operating at NxC speed are called $pure\ output\ queuing\ switches$. The second method for solving the contention occurring on the packet switch is to store these packets in the buffers created on the input lines. In this case, the switch operating speed can be equal to the line speed. In this case, there must be a crossbar structure that can connect all input lines to all output lines. This crossbar connection structure has a feature that can change dynamically. The current connection type of the crossbar is determined by the arbiters. Due to its low complexity, crossbar fabrics consist of $Nx1$ multiplexers as shown in Figure 2.1. Such switches are called $pure\ input\ queuing\ switches$.

There are also NxN packet switches with fabric speed speed-up of $1 < S < N$. These switches are called $combined\ input - output\ switches$ because they need buffers on both the input and output ports. Since they operate at SxC speed, S unit

chips can be transferred on the switch at one unit line rate.

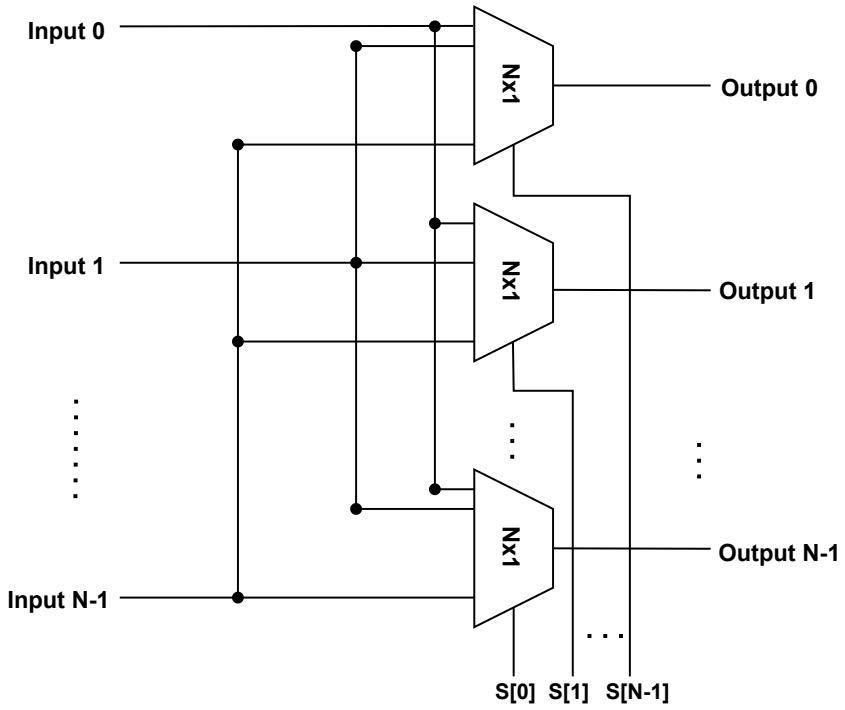


Figure 2.1: Crossbar Fabric

2.1.2 Buffer Architecture

The packet switch structure described in this thesis operates at line speed C and includes crossbar fabric. For this reason, we focus on such packet switch structures from now on.

The size of the packets passing through the packet switch may not be fixed. For this reason, it is common practice to fragment packets into small units before switching in the packet switch [9, 10]. Fixed-size units sent to the output lines need to be reassembled on the output lines.

Packets arriving at the input lines of packet switches are stored in buffers after they are fragmented into fixed-size units. In these buffers, they wait to be transmitted to the output lines with the crossbar fabric. Basically, each input port can store buffers input cells with a simple FIFO structure. However, a problem may arise with this

simple FIFO structure. This performance-limiting problem is called $head - of - line\ blocking\ (HOL)$. The head-of-line blocking problem occurs when the cell waiting to be sent at the head of the FIFO is not sent due to congestion, even if the other packets waiting in the back can be sent. An example of this problem can be seen in Figure 2.2. The box in the middle of the figure shows the crossbar fabric. The boxes to the left of the crossbar fabric indicate the buffers of the input ports, and the numbers inside the boxes indicate the output lines that the fixed-size cells want to go to. Head cell in buffer at input port 1 wants to go to output port 2, while head cells of buffers of input port 0 and input port 2 want to go to output port 0. However, the crossbar fabric is configured to establish a connection between output port 0 and input port 0 according to the arbiter decision. Output port 1 could not establish a connection with any input port. Although there is a packet on input port 1 to be sent to output port 1, it cannot send this packet to output port 1 because the cell located at the beginning of the buffer of input port 1 blocks these next cells. $Head - of - line\ blocking$ is a problem that seriously reduces switch efficiency.

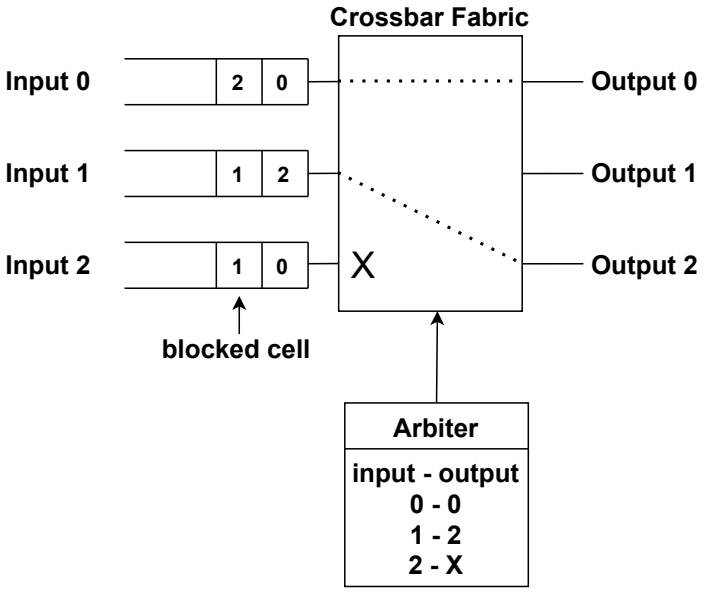


Figure 2.2: An Example of HoL Blocking Problem

Special buffers are organized at the input ports to avoid the head-of-line problem. These buffer structures are called $Virtual\ Output\ Queues\ (VOQs)$. Thus, each input port i has a different $VOQ_{i,j}$ buffer for cells that want to go to each output port j. Thus, even if the buffer sizes increase, the head cell of the buffer does not block

the cells that follow it.

To summarize, the reasons for the need for packet switches, the problems encountered in packet switch structures, and their solutions have been explained. A switch structure designed considering the mentioned critical buffer structure is shown in Figure 2.3. This packet switch performs the switching operation periodically. First, the arbitrator decides which input and output ports to match in this periodic cycle. Then, according to this decision, the crossbar fabric establishes the physical connection between the input and output lines. Finally, the cells stored in the VOQs on the input lines are sent to the output lines.

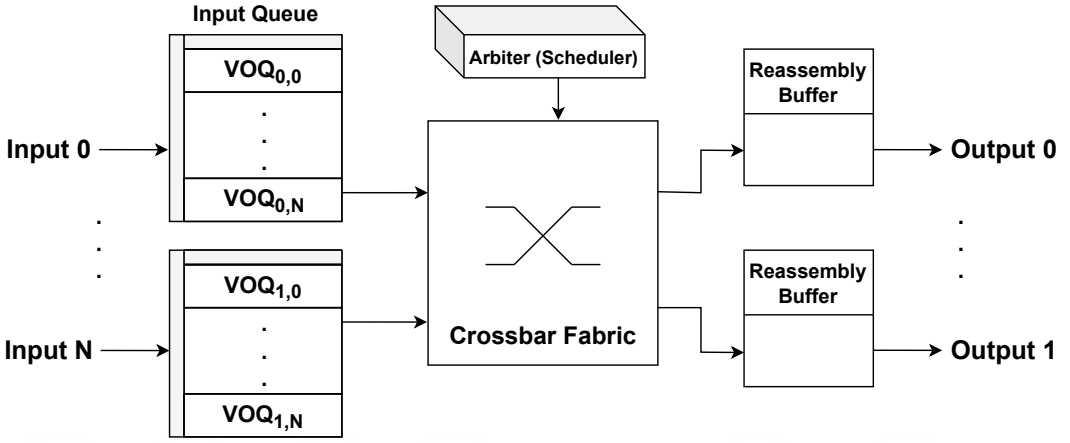


Figure 2.3: Switch Buffer Organization

2.1.3 Fabric Arbiters

Nowadays, there has been a dramatic increase in the communication needs of advanced systems. In particular, smart devices connected to each other over the Internet and cloud computing are used a lot in daily life. These applications need throughput and low packet delays. In this case, fabric arbiter designs have become very important because fabric arbiter performance is a factor that directly determines all on-chip switch performance. All of the fabric arbiters basically try to increase efficiency by ensuring that the largest number of input-output lines are matched. Fabric arbiters are fast and of low complexity so that packets can be sent over the packet switch with low latency. They also ensure fairness between switching input

lines and prevent starvation of any input lines.

For input queued switches, an arbitration method that gives the highest priority to the input line with the most packet in its VOQ is proposed in [11]. Also, this method decides input-output match in a single iteration, reducing the packet delay on the switch. It has less delay than other methods compared according to test results. Another method providing a deadline guarantee for input queued switches is proposed in [12]. However, it is very difficult to give $Quality-of-Service$ guarantee because the packet traffic is unknown in input queued switches [13].

$Work-conserving\ arbiters$ are the decision mechanisms where the maximum number of input and output ports are matched. Maximum input and output line matching can only be given as a result of iterative decisions. Iterative steps continue until no more input and output lines can be matched [8]. A fastest method for maximum matching is suggested in [14]. Even if this algorithm gives confidence that it will provide high efficiency, it is insufficient to prevent starvation on the switch.

Another method that offers maximum matching guarantee is $Parallel\ Iterative\ Matching$ [15]. In this method, the input and output ports are matched iteratively with random priorities. For this reason, it cannot provide a fair Quality-of-Service between the input and output ports. $Iterative\ Round\ Robin\ Matching\ with\ Slip\ (iSLIP)$ algorithms use a round robin sequence instead of [16, 17] random selection. Thus, fairness is achieved between the input lines of the switch. But in this case, it causes an increase in complexity. Additionally, another method commonly used in network switches is $Dual\ Round-Robin\ (DRR)$ [18]. [18] iteratively ensures that the maximum input and output lines are matched. Also, the accept state in the $iSLIP$ method is not needed in DRR. Hence, DRR has low complexity and makes matching decisions in a short time.

2.2 Network On-chip Packet Switches

Thanks to the sharp progress in silicon technology, it has become possible for technology users to perform many calculations on a single chip. Today broadcasts over the Internet, cloud computing, and artificial intelligence applications are the most popular applications that users need to use. In this period, which is called the information age, people need high throughput in such applications. Systems where the required application can be realized on a single silicon chip are known as $System-on-Chip\ (SoC)$. Advanced System-on-Chips have multiple processors and DSPs, re-programmable logic elements, high-speed communication blocks, memory sticks, and multi-purpose I/O pins. These blocks communicate with each other for high-capacity computational operations. This network structure on a single chip is called $Network-on-Chip\ (NoC)$. It can also exist on multiple networks on a single chip [19]. In Network-on-Chips, a fixed size cell sent per unit time is known as a $flit$.

Network-on-Chip (NoC) switches have several advantages and disadvantages over computer switches. Since all units communicating in NoCs are on the same silicon chip, networks with low packet delay and high communication speed can be established. On the other hand, NoCs may face resource problems because they have limited resources and also want to run other applications on the chip. There are two basic types of on-chip switches. These are $homogeneous$ and $heterogeneous\ switches$. Homogeneous switches are regular network structures that enable communication between similar computing cores. An example of these structures is artificial neural network applications [20]. In heterogeneous switches, communication takes place between blocks with different characteristics. For example, in a heterogeneous network, a link is established between hardware accelerators, RAM, and other high-speed communication interfaces.

The on-chip switch designed within the scope of this thesis has a $heterogeneous$ structure to ensure the communication between the reconfigurable regions in the FPGA and between the processor, Ethernet, PCIe, and DRAM blocks. It is foreseen to be used as $a\ hardware\ accelerator$ in cloud platforms of reconfigurable regions created in FPGA. Other blocks are used to enable these hardware accelerators to

perform their tasks efficiently.

These days, the need for faster computational calculations has increased in computer architectures. For this reason, there are differences in the structures of traditional computer architectures. Now, computer architectures that contain pure processors are replaced by processors that also use auxiliary elements to speed up a certain process. This creates highly efficient architectures that can successfully perform a specific operation [21]. Components that can perform a specific task by helping processors are generally called accelerators. Accelerators generally include $graphics\ processing\ units\ (GPUs)$, $field\ programmable\ gate\ arrays\ (FPGAs)$, or many processors with smaller capacities. The most suitable type of accelerator to be used varies according to the intended application. Two decades ago, GPUs were predominantly used for real-time 3D rendering. Today, it is widely preferred in artificial intelligence applications, especially thanks to its parallel processing capability [22]. On the other hand, FPGAs, which have the ability to perform parallel processing like GPUs, are also used as accelerators. It offers high-performance gains to processors with its parallel processing and superior combinational design capabilities [23]. FPGAs have very low energy and resource consumption compared to GPUs and CPUs in image processing, vector operations, and integer convolution operations [24]. In addition, FPGAs and processors can easily work together. Processors can use FPGAs as accelerators in a task or the whole service while performing a service. After the processors write data to the RAMs of the FPGAs that have been previously configured for a specific application, they trigger the FPGAs to start performing their tasks. Then, when the application is terminated on the FPGA side, the processed data in the FPGA memory is read by the processor [25, 26].

In cloud systems, which are increasingly used in computational operations, $hardware\ accelerators\ (HA)$ are used as much as processors and memories [2]. $Hardware\ Accelerated\ Cloud\ Data\ Centers\ (HACDC)$ uses $FPGA\ Accelerator\ Cards\ (FACs)$ with a processor as a hardware accelerator.Another reason why $FACs$ are preferred is that it is possible to implement multiple HAs on a single FAC with SoC thanks to its Reconfigurable Regions (RR) [3, 27]. Such a cloud system needs a structure that provides high-throughput communication between $FACs$. There should also be an on-chip switch specially developed for these high-speed data

transfers [28, 29, 30].

A hardware accelerator design with multiple reconfigurable regions called $vFPGAs$ is proposed in [27]. In this design, all vFPGAs have external memory, PCIe interface, and AXI-Stream interface with two neighboring vFPGAs. Data transfer between these regions is realized with a switch applied on the FPGA. The switch arbiter algorithm is Round Robin. In addition, the arbiter is customized to give high bandwidth to the required vFPGA in the desired situation. These designs are developed on the Xilinx VC709 FPGA board and its static resource consumption is stated to be approximately equal to 7% of the FPGA.

The on-chip packet switch we designed for use in a hardware-accelerated cloud service architecture, described in [3], is proposed in our previous work [29, 30]. This on-chip switch is called *ACCLOUD-SWITCH (Accelerated Cloud Switch)*. The switch has 8 input and 8 output ports and operates at a line speed of 156.25 MHz. Switch input and output lines are compatible with the AXI-Stream interface. The switch provides data exchange between RRs, ARM processor, external RAM, PCIe, and 40G Ethernet interfaces in the FAC. The switch has VOQs that are used to prevent head-of-line (HoL) blocking. The flit size on the switch is 256 bits. Flits are stored in reassembly buffers (RAB) for reassembly before being sent to the output lines of the switches. *ACCLOUD-SWITCH* design is implemented in the Xilinx XC7Z100 SoC programmable logic side on Vivado Design Suite 2016.4. The utilization rates of FPGA LUT, FF, BRAM resources are approximately 7%, 8% and 30%, respectively. The estimated power consumption is around 0.7 Watts. [31] proposes an on-chip packet switch architecture with capabilities of offering different bandwidth allocations to input-output port pairs. The focus of this work is the fabric scheduler that provides the service differentiation and its dynamic buffer memory allocation. The architecture is evaluated with a C++ performance simulator without a systematic verification approach.

2.3 SystemVerilog Verification

With the increasing computational capacity applications, the size and complexity of the designs have been increasing recently. Functional verification is a vital part of systems as well as the design process. Research in [4] indicates that the most time-consuming process in a project life-cycle is design verification. Complex designs can be developed in a short time thanks to the IP cores that are reused during the design phase. However, even if previously designed cores are used, that design needs to be functionally verified again. After the design processes, there should be a universal verification process to reduce the time spent on functional verification and debugging [32, 33]. Thus, as a result of increasing system requirements, the need for fast and accurate designs is also increasing. $SystemVerilog$ and $Universal\ Verification\ Methodology\ (UVM)$ have become invaluable tools for the integration of the work of collaborating designers and to overcome a system-level design verification [5]. SystemVerilog is a combined $Hardware\ Description\ and\ Verification\ Language\ (HDVL)$ developed for system-level verification and design purposes. SystemVerilog standardized as IEEE 1800 in 2005 [34]. It is a language based on Verilog extensions. SystemVerilog is used by many verification designers to functionally test complex systems [35].

[36] proposes a weighted Round-robin arbiter design and verification. There are only arbiter and crossbar fabric designs in the design. However, there is no solution to the head-of-line blocking problem. In the verification part, the accuracy of the contention arbiter decisions is tested using the SystemVerilog language. An architectural solution is not proposed for application to complex systems such as Network-on-Chip, including the arbiter described.

SystemVerilog language consists of 5 basic components as shown in Figure 2.4. These components are testbench, assertion, coverage, Verilog, and DPI (Direct Programming Interface). $Verilog$ is a well-known Hardware Description Language (HDL). Components other than Verilog will be explained in more detail throughout this section.

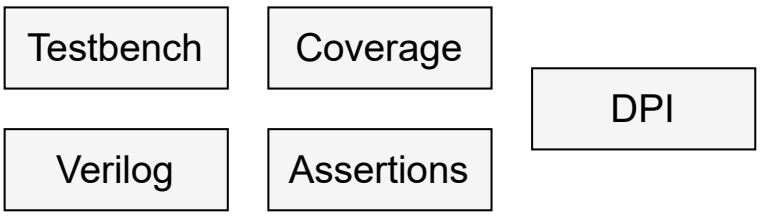


Figure 2.4: Components of SystemVerilog Language

2.3.1 Testbench

$Testbench$ or verification environment provides functional verification of designs in a simulation environment. Testbench consists of classes with different tasks for design and verification of the design. Testbench generates predefined input signals for designs and drives the design with these input signals. Then, it captures the design results and compares the results with the expected values. A standard SystemVerilog testbench consists of different sub-components, shown in Figure 2.5.

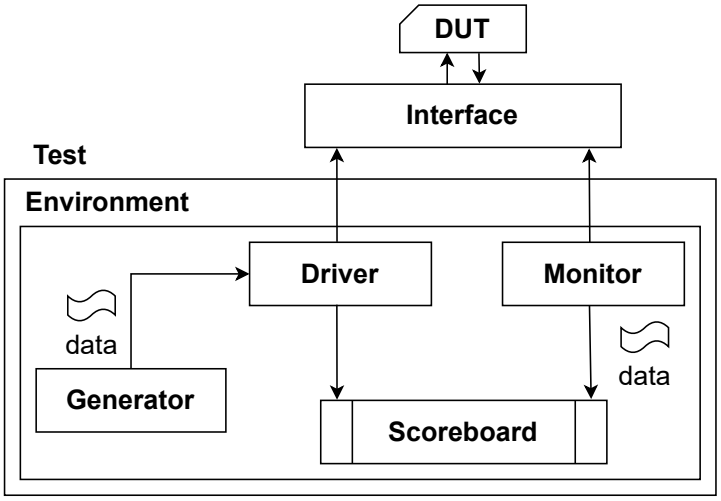


Figure 2.5: SystemVerilog Testbench Components

$Device\ Under\ Test\ (DUT)$ represents the design to be verified, developed with one of the hardware design languages (Verilog or VHDL). It can also be called $Design\,Under\,Verification\ (DUV)$.

$Interface$ is an important structure that facilitates the reusability of designs. It is

created to encapsulate the communication between blocks. Interface structures have features such as parameters, variables, assignments, functions, and tasks. Thus, it is suitable for testing system-level applications. Other modules in testbench can access the information in the interface. They are especially helpful for applications with functional coverage and assertions. The most important difference that distinguishes the interface from other classes in testbench is that the interface is allowed to be connected as a port. Interfaces can also be seen as a container in which all input and output signals in the testbench are placed. Hence, DUTs are driven and monitored through an interface.

$Generator$ is one of the main classes of the testbench. Generator is responsible for generating data transactions to be sent to the design to be verified. The data to be sent to the DUT can be constrained or randomized by this class. $Driver$ class receives the transactions generated by $Generator$ and drives the packet-level data to the DUT through the interface.

The main purpose of verification is to obtain and evaluate the results of the DUT driven by the input data. $Monitor$ class is responsible for monitoring the input data set to the DUT and the output data processed in the DUT to capture the design behavior. Monitor achieves this by observing the activity of the DUT's inputs and outputs at pin level. It then sends these observed signals to the scoreboard component to be checked.

$Scoreboard$ is a class that checks whether the output data processed in the DUT is in the expected behavior. It performs this control process in 2 different ways. Scoreboard can compare the design output data with the expected output data thanks to the golden data-set, which is the expected output data in return for the input data. In another method, the scoreboard has a reference model with the expected behavior of the DUT. Thus, the input data sent to the DUT is also sent to the scoreboard. Then, it is checked whether the DUT outputs match the reference model outputs on the scoreboard. If the design has a functional problem, it is determined in this way. To summarize, a scoreboard is a class that compares the design results and expected results and maintains a score based on these match results.

$Environment$ is a higher-level container that contains verification components Generator, Driver, Monitor, and Scoreboard classes. It also enables easy adaptation of a previously used verification infrastructure to advanced versions of the project or a different project. In short, the environment makes verification more scalable and flexible.

$Test$ component contains the environment that can be set with different configuration settings. Test is a program that creates an object of environment. Thus, it starts the process of configuring testbench and creating other components. It then triggers the drive of the input data-set to be verified for the testbench. In a design that needs to be tested many times, it is not feasible to change the environment for each test. Instead, there are some parameters tweaked for each test the environment has. So, it will be much easier to be able to test a design.

The testbench components described so far are commonly used components to verify a design. As the complexity of the design to be verified increases, new custom auxiliary components like classes may be defined that facilitate the testing process.

2.3.2 Coverage

$Coverage$ is a feature that is used to measure the tested functional parts or features of the design. Thus, it can be observed under which conditions the design can be tested from the features desired to be verified. After specifying the constraints to be observed, it can be concluded how many of these constraints have been achieved as a result of the test. Coverage is divided into two parts. The first of these is $code\ coverage$. Measures how much of the code coverage design code is hit. For a designed Finite State Machine (FSM), the measurement of whether all states are entered can be given as an example of code coverage. $Functional\ coverage$ measures the realization of a user-specified design metric.

The features to be measured in the design can be specified with coverage models. Coverage models can be implemented with the help of $Covergroups$. There may be more than one different situation to be measured within the covergroups defined by the users. In this case, a separate $coverpoint$ variable is defined for each feature.

$Bins$ that can be defined within the coverpoints represent the results of the coverpoint states. With bins, hitting only one result can be defined, or more than one result can be defined with a single bin. An example of coverage class code is shown in Algorithm-1 to better understand the concepts of $covergroup$, $coverpoint$, and bin.

```
class myCoverage ;
    rand bit [2:0] CovPoint ;
    rand bit  CovPoint1 ;
    covergroup CovGroup ;
        coverpoint CovPoint {
            bins b1 = {0};
            bins b2 = {1};
            bins b3 = {2 : $};
        }
        coverpoint CovPoint1 {
            bins d1 = {0};
            bins d2 = {1};
        }
    endgroup
endclass
```

Algorithm 1: Coverage Class Definition Example Code

With the covergroup shown in Algorithm 1, two different coverpoints named $CovPoint$ and $CovPoint1$ are defined to observe whether the desired situations occur during design verification. $CovPoint$ is assumed to be a 3-bit wide variable. Thus, the $CovPoint$ variable can take 8 different values. If these 8 different values want to be observed, an example coverage group can be as in the Algorithm-1. The bin variables defined under $CovPoint'$s coverpoint show the values that CovPoint takes. If the $CovPoint$ variable takes the value 0, it means that bin $b1$ hit. Similarly, bin $b2$ reports whether a value of 1 is observed. On the other hand, $b3$ is defined to check whether it hits more than one result under a single bin. If values greater than 2 and 2 are observed, it means that bin $b3$ has occurred. With $CovPoint1$, it can be checked whether another situation has occurred.

Another feature of covergroups is that it can be cross-checked whether the conditions specified in the bin definitions of other coverpoints are met at the same time in a different coverpoint. In addition, it is provided by the functions defined in SystemVerilog with different features that can be controlled such as the minimum number of hits to a coverpoint bin metric.

2.3.3 Assertions

$Assertions$ are used to control the desired behavior of a system. In this way, a desired feature of the design can be verified in the simulation process. Additionally, checking whether a condition has occurred can also be checked using assertions. Assertions generate errors or warnings when an undesirable situation occurs. In other words, assertions are representations that provide functional control of a feature of the design.

2.3.4 DPI

$Direct\ Programming\ Interface\ (DPI)$ is the interface between one programming language and another programming language. SystemVerilog Direct Programming Interface refers to the interface between SystemVerilog and foreign programming languages. Thanks to this interface, SystemVerilog codes can be called by C, and C functions can be easily called by SystemVerilog. This allows users to reuse previously produced codes. Hence, a faster code development process is possible.

2.4 Placement of the Thesis Work in the Literature

This thesis fills in the gap in the literature by addressing on-chip packet design and verification together. The on-chip switch proposed in the thesis has generic parameters for flexible implementation with different configurations on FPGA. Hence, the desired number of input and output ports, buffer memory, and data width can be easily selected with the help of defined parameters. In addition, the on-chip switch's input and output ports support the AXI4-Stream protocol, which is a standard interface used in FPGAs to connect components that want to exchange data. Hence, general

IP cores and designs with AXI4-Stream communication standards can also be flexibly connected to the on-chip switch. Furthermore, a verification design using SystemVerilog infrastructure with cycle accuracy is developed to verify the on-chip switch within the scope of the thesis. This verification design tests the on-chip switch extensively with random input data. In addition, a detailed performance evaluation of the design is obtained on the RTL simulator.

The works that cover verification are [36] and [37]. [36] proposes a Round Robin arbiter design and verification of the arbiter in the SystemVerilog environment. However, this study only verifies the contention arbiter method, a packet switch architecture is not proposed. [37] focus on a verification methodology in the SystemVerilog environment. However, it does not include detailed verification design descriptions, test coverage, and scoreboard reports.

To the best of our knowledge, the design and the systematic verification using the SystemVerilog environment of an on-chip switch with AXI interfaces is not covered in the literature. We provide all design details together with the detailed coverage and scoreboard reports.

Mentioned previous studies are summarized in Tables 2.1.

Table 2.1: Summary of Related Previous Work

Source, Year	Design	Implementation Platform	Verification Platform	Coverage	Scoreboard
[36], 2018	Contention Arbiter	ASICs	SystemVerilog	Included	Included
[37], 2010	Ethernet Switch	NetFPGA	SystemVerilog	Included	Included
[29], 2020	On-Chip Packet Switch	Xilinx Zynq-7000 Family XC7Z100FFG1156-2 SoC	C++	Not Applicable	Not Applicable
[*], 2022	On-Chip Packet Switch	Xilinx Zynq-7000 Family XC7Z100FFG1156-2 SoC	SystemVerilog	Detailed Coverage Constraints and Reports	Included
(*) High Speed On-Chip Packet Switch developed in the scope of this thesis.					

CHAPTER 3

ON-CHIP SWITCH DESIGN

In this chapter we present the design of the on-chip switch in detail. The on-chip switch is designed with $N = 9$ lines and all lines are designed to operate at 40 Gbps. The switch interconnects 4 Reconfigurable Regions (RR) to implement the hardware accelerators, ARM SoC processor, DDR, PCIe, and 2x 40 Gbps Ethernet network interfaces. The switch is designed similar to a computer network switch to interconnect such heterogeneous interfaces. The scalability is supported by the line-speed operation of the switch without any internal speed-up. To this end, all the buffers are at the input ports and organized as Virtual Output Queues (VOQs). We implement Dual Roun Robin (DRR) [18] method for the fabric arbiter to achieve full throughput. The switching of data is carried out in flits compatible with the on-chip interconnection infrastructure. To this end, the variable size packets that are received on the input interfaces are carried through the fabric interleaved by the fabric arbiter and reassembled at the switch outputs in Reassembly Buffers (RABs).

3.1 Hardware Architecture of Switch

High Speed On-chip Switch block architecture is shown in Figure 3.1 for $N = 9$ ports. The design is detailed for the input line i and the output line j. The switch fabric is designed to operate at line speed $(c = 1)$ in a scalable way. The size $flit$, which is the number of bits switched in one clock cycle, is selected as 256 bits. The reason for this is that it is desired to develop a structure compatible with the 40 Gbps IP Core [7] interface which is 256 bit wide.

Here, we first provide an overview of the blocks, then explain the on-chip switch

design in detail in the rest of this chapter.

In order to prevent the head of line blocking, virtual output queues are created at each input $i \in (0 \ldots 8)$ to increase data output and prevent data loss. The virtual output queue block at input port i is named as $\texttt{VOQ}_i$. $\texttt{VOQ}_i$ is partitioned into 9 $\texttt{VOQ}_{i,j}$ regions for output port $j \in (0 \ldots 8)$ to stores the flits that are destined to output port j.

$\texttt{VOQ Controller}_i$ module controls packets stored in $\texttt{VOQ}_i$. To this end, $\texttt{VOQ Controller}_i$ writes the flits arriving at input i, destined to output j to $\texttt{VOQ}_{i,j}$. In addition, $\texttt{VOQ}_i$ sends the information to `Arbiter` module that the input line i has data ready to be sent to which output lines by $empty_status_i$ signal. `Crossbar Fabric` block is a basic crossbar structure that connects each input line to all output lines with a multiplexer.

`Arbiter` module decides the one-to-one connection configuration for `Crossbar Fabric` and sets the selection inputs of the multiplexers. $\texttt{VOQ Controller}_i$ block ensures that the data stored in $\texttt{VOQ}_{i,j}$ is transmitted to the `Crossbar Fabric` input ports according to the contention arbiter decision. The arbiter algorithm used in the design is a well-known algorithm as the Dual Round Robin (DRR) approach in the literature [18]. `Arbiter` uses the DRR approach with a maximum of 3 iterations to increase switch efficiency in a decision-making phase.

Decision-making within the switch and data transmission over the crossbar fabric is implemented with a pipeline method. Switching is accomplished with fixed size *cells* to increase pipeline throughput and to make configuration changes often enough to determine the appropriate configuration for packet arrival. The time spent calculating the connection configuration and switching cells is what we call $operation\ cycle$ (op_cycle). We select an $operation\ cycle$ of 14 clock cycles for both achieving a high pipeline efficiency and frequent enough fabric arbitration to closely track the incoming data traffic. Hence, each cell is 14 flits, 14 × 256/8 = 448 Bytes. To this end, the configuration of the fabric during op_cycle$_n$ stays constant and is updated in op_cycle$_{n-1}$. During an op_cycle up to 448 Bytes are switched between the connected input/output port pairs. We provide detailed operation of the pipeline design in Section 3.1.8.

In some rare cases, there may be no flit left to send from VOQs. However, `Crossbar Fabric`'s input and output ports are still connected. In order to control this situation, with the help of a bit in the VOQ flit structure, the information of whether that packet is valid is transmitted across the switch. When a valid flit is not observed at `Crossbar Fabric` input, this flit is then ignored by `Crossbar Fabric`.

Flits delivered from input line i to output line j are stored at the reassembly buffers as named $\texttt{RAB}_j$ for output port j which is partitioned into regions $\texttt{RAB}_{j,i}$ where $i \in (0 \ldots 8)$. To this end, the flits arriving from input i are stored in $\texttt{RAB}_{j,i}$ to be reassembled into the original packet.

$\texttt{RAB Controller}_j$ block controls $\texttt{RAB}_j$ and directs the flits from input port i of the `Crossbar Fabric` module to be stored in $\texttt{RAB}_{j,i}$ region. The module that reassembles the flits in the $\texttt{RAB}_j$ is $\texttt{RA Scheduler}_j$ block. $\texttt{RA Scheduler}_j$ module receives the information that the last flit of a packet has reached $\texttt{RAB}_j$ by $last_flit_info_j$ signal from $\texttt{RAB Controller}_j$. It is possible that more than one $\texttt{RAB}_{j,i}$ have complete packets and are ready to go out of port j. For these reasons, $\texttt{RA Scheduler}_j$ selects the next packet to be transmitted with a Round Robin approach and transmits the reassembled packet to output line j with appropriate packet format and timing.

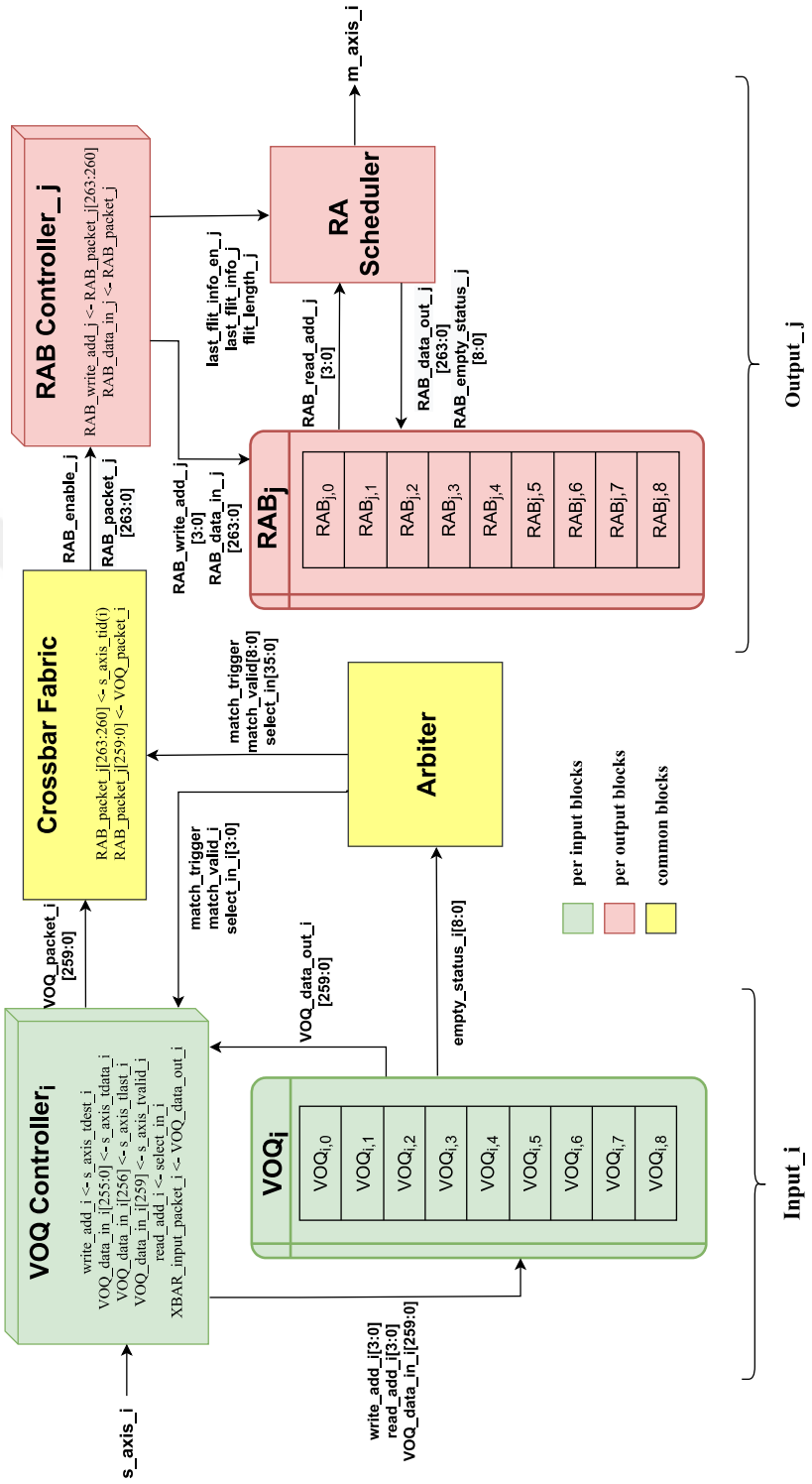


Figure 3.1: Switch Architecture

3.1.1 VOQ Controller

VOQ Controller$_i$ receives the packets arriving at input i, converts them to the flit format that we show in Table 3.1, writes the flits in VOQ$_{i,j}$ and sends them to Crossbar Fabric according to the decision of Arbiter. In Figure 3.2, the interface signals of VOQ Controller$_i$ for input port $i \in (0 \ldots 8)$ are shown in detail.

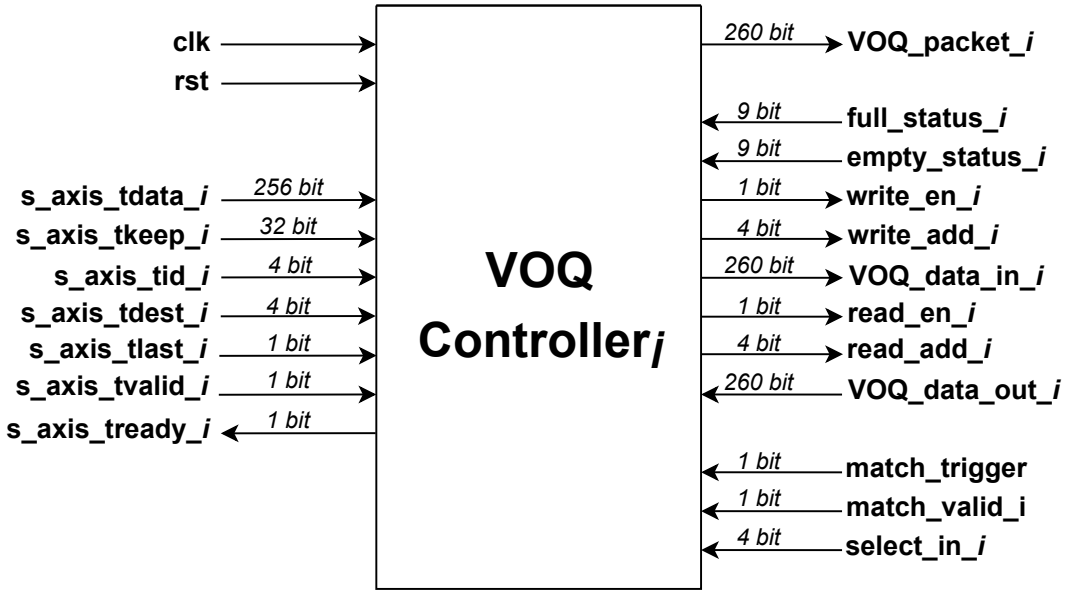


Figure 3.2: VOQ Controller Block Diagram

Input data of the on-chip switch is in AXI4-Stream standard. VOQ Controller also supports AXI4-Stream standard communication. Thus, the source and destination port information for the flits is obtained with a successful AXI4-Stream communication.

The successfully received packets from input line i destined to output j are written to the region VOQ$_{i,j}$ as a VOQ flit thanks to VOQ Controller$_i$. The destination region VOQ$_{i,j}$ is determined by $s_axis_dest_i$ signal value. VOQ flits have 260-bit length data. It contains AXI4-Stream data, last and valid signal information as shown in Table 3.1. Moreover, if there is not enough space in the destination region VOQ$_{i,j}$, this situation is indicated by pulling $s_axis_ready_i$ signal into logic low signal level according to AXI4-Stream protocol.

Table 3.1: Bit Field of VOQ Flits

259	**258:257**	**256**	**255:0**
valid	reserved	last	data

The rising edge of the $match_trigger$ signal indicates the start of a new match period. At the start of a match period, the $select_in_i$ signal is updated and displays the current match until the next match period. When $match_valid_i$ signal is logic high, it means that there is a valid match for input port i. In this case, `VOQ Controller`$_i$ reads the flits stored in the corresponding region of the `VOQ`$_i$ based on the destination id info in $select_in_i$ signal. Then, the flits are sent to the `Crossbar Fabric`. We note that the fabric ports stay connected and switching goes on during the op_cycle. Hence, if there are no more real flits to send, a fixed flit is sent to `Crossbar Fabric` with the valid bit set to 0. Hence, `Crossbar Fabric` can ignore this flit.

3.1.2 Virtual Output Queue

In a traditional Xilinx Block RAM (BRAM) resource, there are two signals that indicate a BRAM is full and empty. Additionally, the depth of BRAMs is interpreted as 2's power. If we use a BRAM for each virtual output queue, it will consume BRAM resources that are the power of 2, greater than the packet depth to be specified for the VOQ. This limits the depth of each virtual output queue to 2's power and leads to an inefficient memory organization.

A special VOQ architecture is developed to increase resource efficiency and have flexible virtual output queue depth. The VOQ architecture designed is shown for the input port $i \in (0 \ldots 8)$ in Figure 3.3 with separate `VOQ`$_{i,j}$ regions allocated for each output port $j \in (0 \ldots 8)$. `VOQ`$_{i,j}$ depths can be easily changed with the help of a parameter located in the top module of the switch design. Furthermore, the information that each `VOQ`$_{i,j}$ region is full and empty can be sent to other blocks

in switch module by 9-bit $empty_status_i$ and $full_status_i$ signals. In short, all $\texttt{VOQ}_{i,j}$s can be implemented in a single Block RAM for an input port i which provides more efficient resource consumption thanks to the VOQ architecture in Figure 3.3.

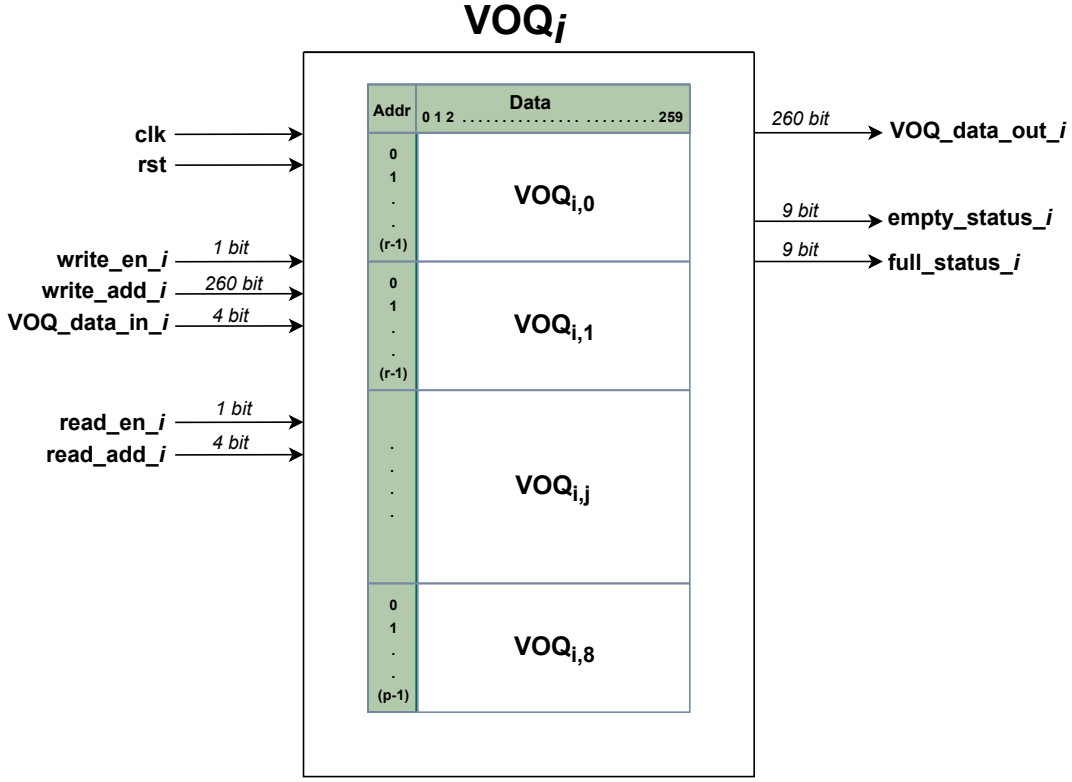


Figure 3.3: Virtual Output Queue Architecture

Although the application running and the data produced on Reconfigurable Regions are under the designer's control, it is assumed that there is no such control in data communication made from other interfaces. In traffic under 95% maximum load, according to the simulator results [30], the VOQ size is 16 flits on lines connected to RRs, and 512 flits on other lines. Hence, $\texttt{VOQ}_{i,j}$ region size is reserved 16 flit for $i \in (0 \ldots 3)$ and 512 flit $i \in (4 \ldots 8)$. Moreover, these sizes for VOQs are in a generic structure. They can be adjusted via a parameter easily.

3.1.3 Crossbar Fabric

`Crossbar Fabric` has NxN port switch structure. In this thesis, we implement a crossbar fabric with $N = 9$. Block diagram of `Crossbar Fabric` can be seen in Figure 3.4. The fabric is capable of connecting all input ports $i \in (0 \ldots 8)$ to all output ports $j \in (0 \ldots 8)$ with multiplexers. The multiplexer select inputs of `Crossbar Fabric` are determined by `Arbiter` module matching decision. Input and output ports of `Crossbar Fabric` operate at line speed 148.5MHz.

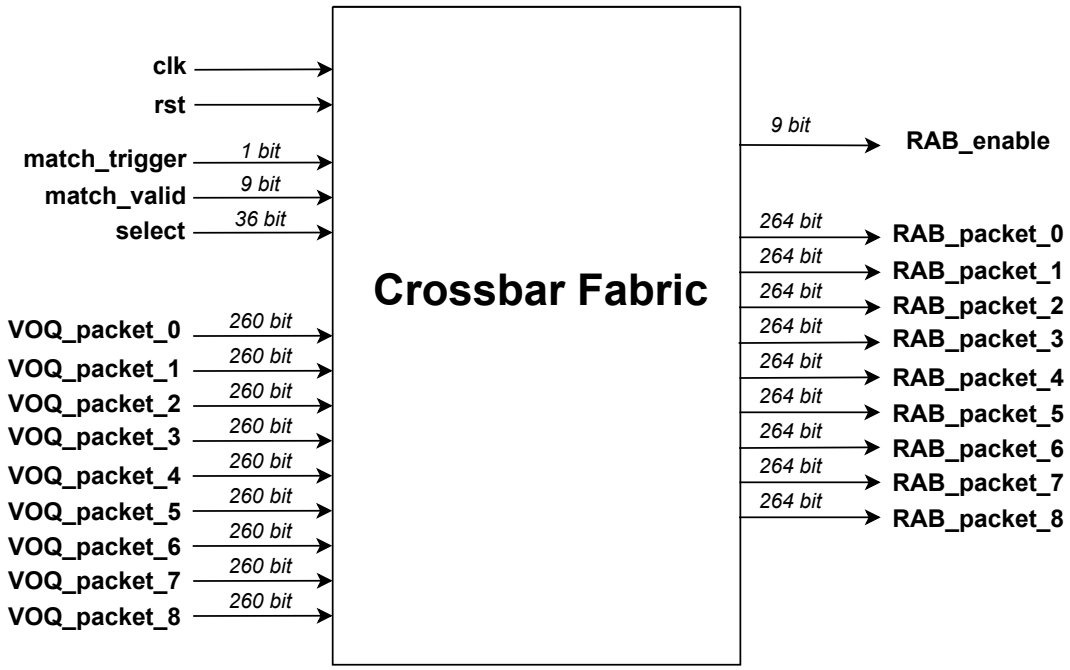


Figure 3.4: Crossbar Switch Block Diagram

`Crossbar Fabric` ensures that the flits stored in the virtual output queues are transmitted to the target reassembly buffers according to arbiter decision. `Crossbar Fabric` input and output ports connection are configured according to the $select$ signals that have matching information as a result of `Arbiter` decision. `Crossbar Fabric` configuration continues until the rising edge of the $match_trigger$ signal indicates the next match arrives. `Crossbar Fabric` input ports support data format in VOQ flit standard. The bit field of VOQ flits can be seen in Table 3.1. Output ports are connected to RABs. RAB flit width is 264 bits. The reason why the RAB flit width differs from the VOQ flit width is due to the fact that the flits going to the reassembly buffers also have input port information. The bit field of RAB Flits

is shown in Table 3.2. `Crossbar Fabric` also adds the input port id information to the 260-bit input data before transferring input flits to output ports. Hence, the `Crossbar Fabric` output port width is 264 bits.

Table 3.2: Bit Field of RAB Flits

263:260	**259**	**258:257**	**256**	**255:0**
source id	valid	reserved	last	data

All input and output ports of `Crossbar Fabric` may not be matched at each match. To this end, the RAB_enable signal has been added to indicate the matched ports. Each bit of the 9-bit RAB_enable signal indicates whether the data on the corresponding output line is valid or not. In addition, the RAB_enable signal indicates that there are no more valid flits at the output ports when the flits to be sent from the VOQs are finished. In short, it is used to notify `RAB Controller` when there are no valid flits on the output ports.

3.1.4 Arbiter

It is mentioned in the previous sections that packets from the on-chip switch input ports are stored in VOQs. `Arbiter` decides an one-to-one input/output port connection configuration. `Crossbar Fabric` configuration is updated after `Arbiter` decides the matching status of the input and output ports. The input and output ports of `Arbiter` module used in the design are shown in Figure 3.5.

The signal $request_in_i$ indicates whether there are packets on the $\texttt{VOQ}_i$ of input port i to be sent to the output ports. $request_in_i$ is a 9-bit signal and each bit corresponds to the regions of the $\texttt{VOQ}_i$. For this reason, $empty_status_i$ signal, which shows the emptiness information of $\texttt{VOQ}_i$ regions, is used as the $request_in_i$ signal of `Arbiter`. If there is no flit stored in the j region of the $\texttt{VOQ}_i$ ($\texttt{VOQ}_{i,j}$), the $request_in_i[j]$ bit is equal to the logic low level. If there is at least one flit to be sent in $\texttt{VOQ}_{i,j}$, $request_in_i[j]$ signal is at logic high level. Thus, `Arbiter`

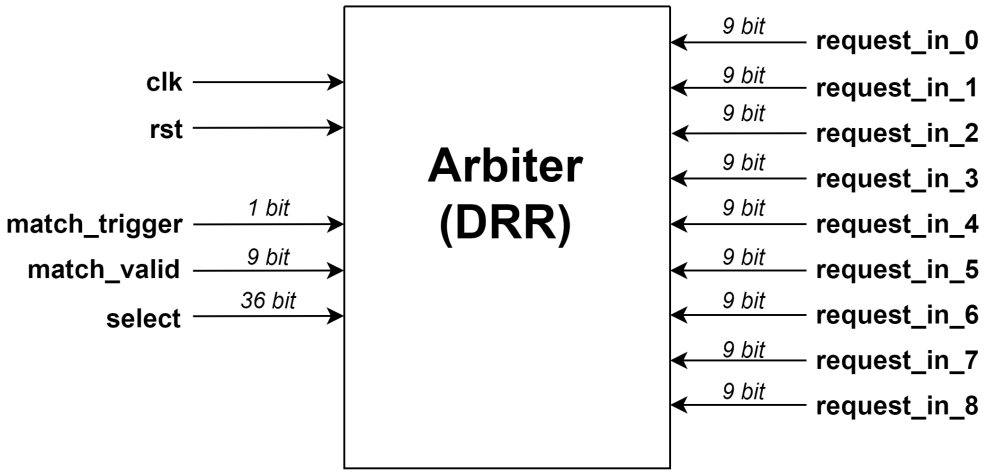


Figure 3.5: Arbiter Block Diagram

obtains the information to which output ports the input ports want to send flits. After `Arbiter` has made a matching decision, matching information must be sent to the relevant blocks in the on-chip switch to make other blocks ready for new flit transfer. Matching information is transmitted with a 36-bit $select$ signal. For each input port, the information to which output port it can send packets includes 4-bit output port id information. Since there are 9 input ports in total, the select signal is 36-bit. Each bit of the 9-bit $match_valid$ signal indicates whether there is a valid match for the input lines. If the i^{th} bit of the $match_valid$ signal is at the logic high level, it means that the input line i is deserved to send packets as a result of `Arbiter` decision. After `Arbiter` block has decided which input and output lines will transfer data, `Crossbar Fabric` is reconfigured to allow flit transfer. This configuration remains unchanged until `Arbiter` block makes its next decision. Each new decision result of `Arbiter` block is indicated by $match_trigger$ signal to other blocks. Hence, other blocks in the switch can prepare themselves for the new match.

`Arbiter` used in this study is Dual Round Robin (DRR) [18] with a maximum of 3 iterations. `Arbiter` notifies other blocks for new match information in every op_cycle (14 clock cycles). The algorithm can be seen in detail in Figure 3.6.

`Arbiter` starts from $Request$ state first after initial reset. In the $Request$ state, the information that the input ports have flits to send to which output ports is instantly

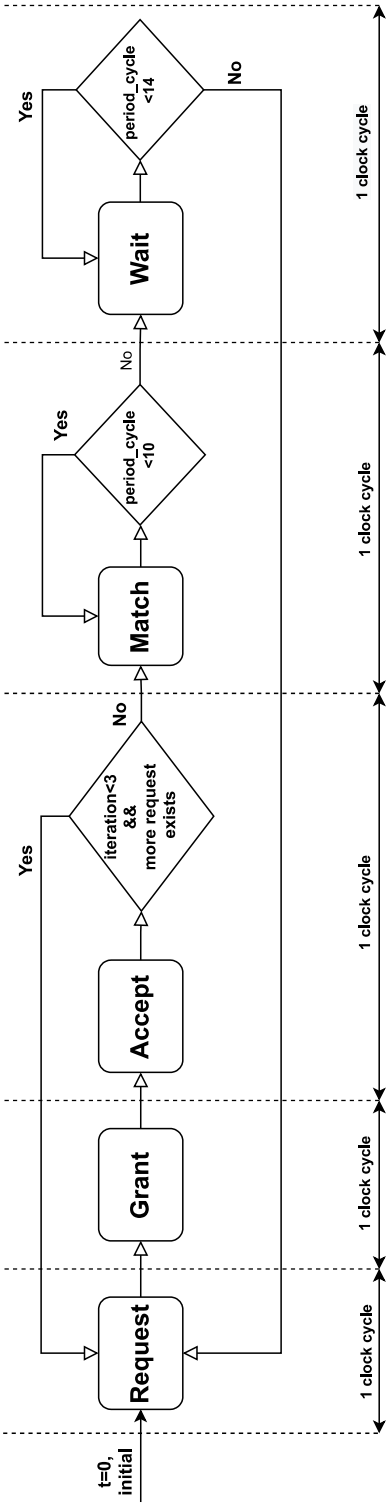


Figure 3.6: Scheduler Cycles

received and saved for use in other iterations. This request information is obtained with $request_in_i$ signals. $request_in_i$ signals are not sampled in each iteration. Requests are sampled only at the start of each new round of decision-making. Also, the next thing to do in the $Request$ state is to specify the output ports that the input ports want to send flits to. Input ports may want to send flits to more than one output port at the same time. However, according to the Dual Round Robin algorithm, one output port is selected among all the output ports to be sent. In general, all output ports are equal. There is a request pointer to decide between the output ports in a fair way. The pointer initially gives priority to one output port. Then, the pointer value is updated so that the output line that the flit is sent is the least prioritized for the next matching decision. Since there are 9 output ports in the design, the pointer value for the input port i is updated as $(j + 1) mod\ 9$ after sending a flit to the output port j. Within 1 clock cycle, it is decided which output port all input ports want to send flits to.

The state passed after the request information collected in $Request$ state is $Grant$ state. In the $Grant$ state, the output ports evaluate requests from input ports that want to send flits to them. If there is only one input line that wants to send flits to an output line, the output line accepts this request. However, if more than one input line wants to send flits to the same output line at the same time, this causes a problem since an output line can only be connected with a single input port at a time. This problem is solved with the help of a pointer similar to the one used in the $Request$ state. There is a special grant pointer for each output port. If there is more than one input line request for the same output line, this pointer gives priority to one line specific to that match round. Then, in order to ensure fairness between the input ports, the grant pointer is updated so that the input line that is decided as a result of the match has the least priority. Since there are 9 output ports in the design, the grant pointer value is updated as $(i + 1) mod\ 9$ after the input port i sends a flit to the output port j. Within 1 clock cycle, it is decided that all output ports agree to send flits from which input port.

There are basically 3 steps in the $Accept$ state. In the first step, match information is collected based on the approvals given in the $Grant$ state. Then, the requests of the matching input ports are deleted from the request information collected in the

$Request$ state. In addition, requests to the matched output ports from input ports that fail to match in the previous iteration are also discarded. The final step is to decide whether a new iteration is needed. If there are unpaired input ports that want to send flits to unpaired output ports, the efficiency of the switch can be increased with a new iteration. Therefore, to start a new iteration, the $Accept$ state is passed to the $Request$ state. Thus, an output port is determined for the input ports to which they want to send flits again. If there is no need for a new iteration, it is passed from the $Accept$ state to the $Match$ state. The design has been developed to make a maximum of 3 iterations. After 3 iterations, it will switch from $Accept$ state to $Match$ state regardless of whether there are still unmatched ports.

The $match_trigger$ signal that notifies other blocks that there is a new match is driven in $Match$ state. On the rising edge of the $match_trigger$ signal, the information contained in the $match_valid$ and $select$ signals are sampled by other blocks. Another task of the $Match$ state is to ensure that the match rounds are completed at the same time. As a result of one or two iterations, the lines that want to send flits can be determined. In these cases, the timing of the decision rounds after three iterations differs. However, such a situation is not desired for the switch design to work in a deterministic way. In addition, if the matching round is completed as a result of one or two iterations and the information that there is a new match is reported to the other blocks, less time will be allocated for the flit transfer that took place as a result of the previous match. This causes an unfairness between the input lines. The value of $period_counter$, which is reset at the beginning of the match round and increases with each clock cycle that passes in the match, is checked to ensure that each match round is completed in an equal amount of time. When $period_counter$ is 10, the $Wait$ state is passed from the $Match$ state.

As a result of the decisions made by `Arbiter`, it takes time for the flits stored in the VOQs to arrive at the `Crossbar Fabric`. At the same time, the fact that the read requests sent in the previous matching round are not terminated also causes errors in the value of the request signals collected from the VOQs. That's why `Arbiter` waits for packet reading from VOQs to start based on the current decision result before starting a new round of decision making. Since these processes take 4 clock cycles, the $Wait$ state controls the $period_cycle$ value to reach 14. After the value of

$period_cycle$ reaches 14, the $Request$ state is returned to start the new match round. In this section, `Arbiter` working cycle is explained. More detailed information can be found in Section 3.1.8 where the pipeline working cycle of the switch is explained.

3.1.5 Reassembly Controller

Reassembly controller for each output port $j \in (0 \ldots 8)$ named `RAB Controller`$_j$ basically performs two tasks. The first of these tasks is to send flits from `Crossbar Fabric` with source port i to reassembly buffer `RAB`$_{j,i}$.

The other main task is to inform `RA Scheduler`$_j$ how many flits are in the packets coming from `Crossbar Fabric` and the last flit of the current packet has arrived. In Figure 3.7, the input and output signal interfaces of `RAB Controller`$_j$ are shown in detail.

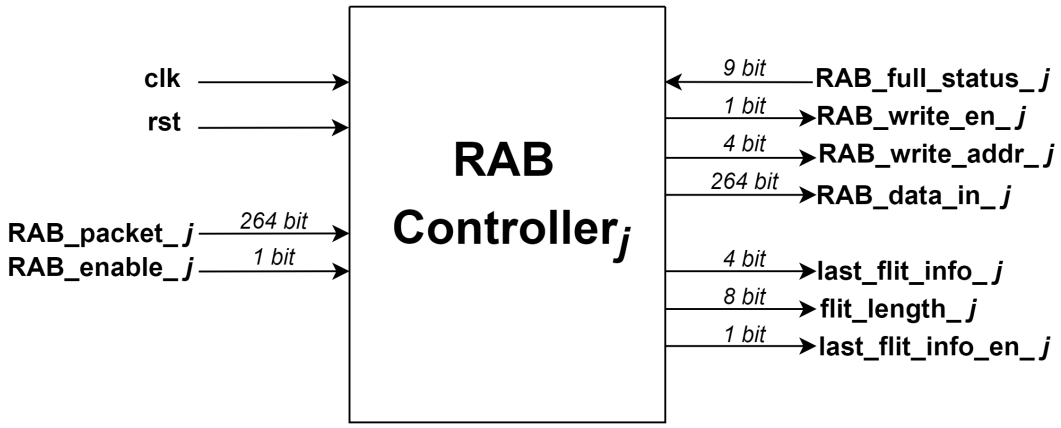


Figure 3.7: RAB Controller Block Diagram

`RAB Controller` runs at line speed. Thus, each flit coming from `Crossbar Fabric` is sent to the reassembly buffers with the same speed. RAB_packet_j signal represents flit data from `Crossbar Fabric`. As shown in Table 3.2, bits between 260 and 263 of the RAB_packet_j flit indicate which input port the flit comes from. The id information of the input port in these bits is transferred to the $RAB_write_add_j$ signal in order to determine the destination region in `RAB`$_j$. Although the reassembly buffers are implemented as Block RAM, it is sufficient to

send only the input port id information as a write address. The reason for this is that the read and write pointers are stored in specially designed reassembly buffers. This structure is explained in more detail in the `Reassembly Buffer (RAB)` section.

RAB_packet_j is sent to be stored in reassembly buffers by signal $RAB_data_in_j$. The $RAB_full_status_j$ signal indicates the fullness status of the regions in the output port j reserved for all input ports. Bit i of $RAB_full_status_j$ signal indicates fullness status for ith region in RAB$_j$. Thanks to a write error counter kept in the RAB controller, the number of unsuccessful write requests to the reassembly buffers of the flit coming from the crossbar switch is calculated in case of no space in reassembly buffers.

256th bits of the VOQ and RAB flits show whether that flit is the last flit of the communication packet to be sent, as can be seen in Table 3.1 and Table 3.2. The number of flits coming from each input port i is kept in the corresponding part of the $packet_length$ vector in `RAB Controller`. The number of flits is increased until the last flit arrives. When the last flit arrives, the last flit information is reported to `RA Scheduler`$_j$ with the rising edge of the $last_flit_info_en_j$ signal. `RA Scheduler`$_j$ obtains all flit information about the packet sent to the `RAB`$_j$ by sampling the signals $last_flit_info_j$ and $flit_length_j$. The $last_flit_info_j$ signal indicates which input port it came from, while the $flit_length_j$ signal shows the number of flits the completed packet consists of.

3.1.6 Reassembly Buffer

A packet that is passing over the switch may consist of one or more flits. However, all flits in a packet may not pass through the switch one after the other. This is due to changes in the connection of `Crossbar Fabric` input and output ports. All flits in a packet may not exit `Crossbar Fabric` in the same match cycle, or a packet may contain more flits than the clock iteration in a matching cycle. In this case, the integrity of the packets leaving `Crossbar Fabric` must be preserved. The common solution to this problem is to store the flits passing through the switch in one area and then assemble them to form a packet. The common name given to these storage areas is the reassembly buffer. In summary, reassembly buffers are regions where the flits are stored after leaving `Crossbar Fabric`.

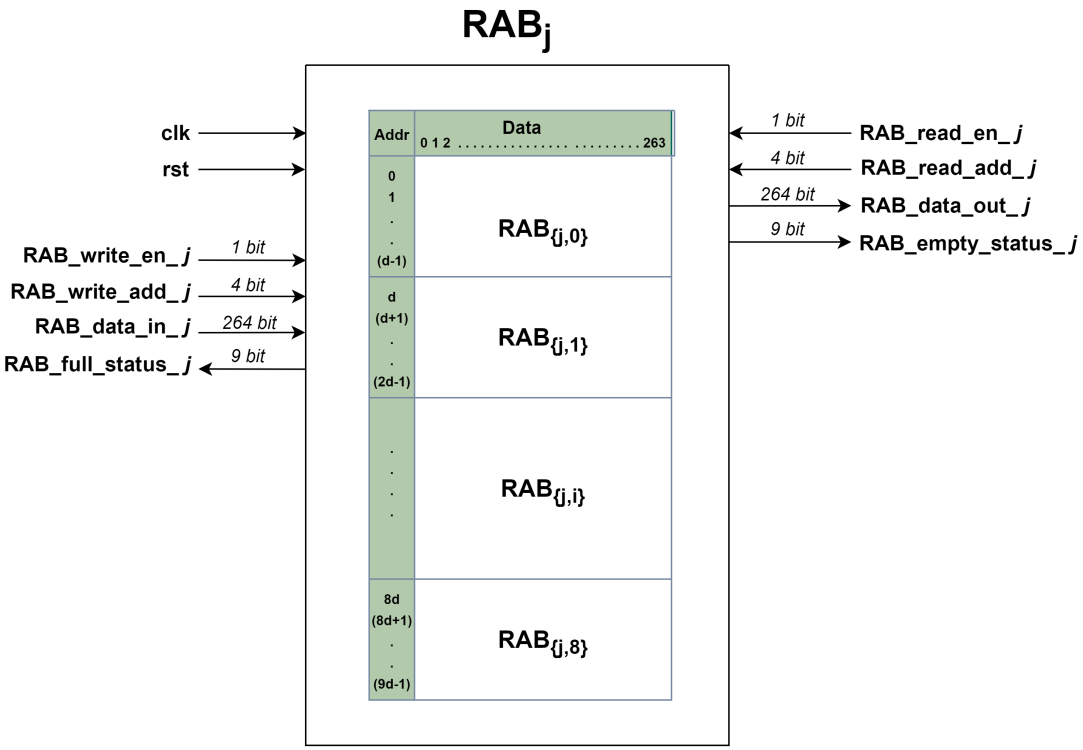


Figure 3.8: Reassembly Buffer Architecture

In this study, reassembly buffers ($\texttt{RAB}_j$) store the flits sent by `RAB Controller`. The stored flits are sent from $\texttt{RAB}_j$s with the read request of the $\texttt{RA Scheduler}_j$. It supports 264 bits of input data and output data and operates at a line rate. There is a separate $\texttt{RAB}_{j,i}$ in output port j for each input port $i \in (0 \ldots 8)$. A detailed

architectural representation showing the input and output ports and the regions where the data is stored for output port $j \in (0 \ldots 8)$ is shown in Figure 3.8.

$\texttt{RAB}_j$s are also specially designed to reduce resource consumption such as virtual output queues. Xilinx FPGA Block RAM modules have 1-bit standard logic signals for showing the full and empty status of RAMs. For this reason, if $\texttt{RAB}_j$s are tried to be implemented with a standard Block RAM, different storage resources must be used for each $\texttt{RAB}_{j,i}$. In the $\texttt{RAB}_j$ architecture shown in Figure 3.8, there is a BRAM resource divided into regions. It is named $\texttt{RAB}_j$ for each output port $j \in (0 \ldots 8)$. In $\texttt{RAB}_j$ structure, the regions specially reserved for the input port $i \in (0 \ldots 8)$ are called $\text{RAB}_{j,i}$. Furthermore, there are special write ($head_i$) and read ($tail_i$) pointers for regions in $\texttt{RAB}_j$ in order to implement them for all input ports in a single Block RAM. In addition, the relevant empty and full flags are used for the fullness or emptiness status of the packets in the regions. The empty ($RAB_empty_status_j$) and full ($RAB_full_status_j$) signals have a 9-bit width to indicate the status in all regions reserved for input ports. Moreover, reassembly buffer sizes can be changed generically, thanks to the RAM_depth parameter in the project top module. The RAM_depth parameter is 2 or a power of 2 allows for more efficient resource consumption. Otherwise, a minimum power of 2, which is larger than the number in the parameter size, is already reserved for reassembly buffers.

3.1.7 Reassembly Scheduler

Packets passing through the switch may consist of one or more flits. There may also be cases where the flits that make up the package do not pass through the switch one after the other. In these cases, other flits are stored in reassembly buffers until the last flit of the package arrives. After the last flit arrives, all flits must be sent to the output ports. The structure that understands the last flit is coming and sends the flits to the output ports is called the Reassembly Scheduler named `RA Scheduler` in this design. It is explained in the Reassembly Buffer section that there are special regions for all input ports in reassembly buffers. In the process of transmitting all the flits in a packet sent from an input port to output ports, the last flits of the packets sent from other input ports may also have arrived in the reassembly buffers. In short, packets

belonging to multiple input ports waiting to be sent to output ports may be ready in reassembly buffers. In such a case, since only one packet can be sent at the same time on the output port, there must be a decision mechanism for the packets to be sent.

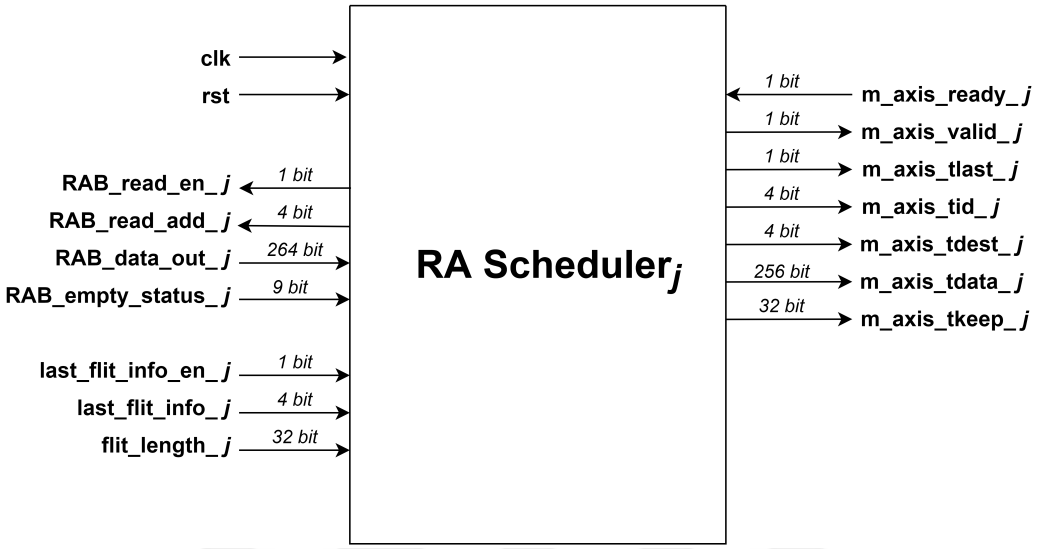


Figure 3.9: Reassembly Scheduler Block Diagram

Reassembly Scheduler is the block that sends the flits stored in the reassembly buffers to the output ports. While providing this transmission, it should have a fair decision mechanism for the input ports. For this decision mechanism, Round Robin arbitration (RR) is used to prioritize all options sequentially in a loop. In addition, since the output ports of the on-chip switch design have communication in the AXI4-Stream standard, data should be sent to the output ports in this standard.

There are 9 Reassembly Schedulers in total, one for each output port. The detailed block diagram of `RA Scheduler`$_j$ for output port j where $j \in (0 \dots 8)$ used in the design is shown in Figure 3.9. `RA Scheduler`$_j$ can read the packets in the desired region of RAB$_j$ with the signals $RAB_read_en_j$ and $RAB_read_add_j$. The first 256-bit data piece of the 264-bit data in $RAB_data_out_j$ signal is the data information to be used in AXI4-Stream communication. The $RAB_empty_status_j$ signal indicates whether there are flits stored in the `RAB`$_j$ regions. These signals are between `RA Scheduler`$_j$ and `RAB`$_j$ blocks of output port j. `RA Scheduler`$_j$ block needs the information that the last flit of a communication packet is also stored in `RAB`$_j$ to start sending data to the output port. This information comes from `RAB`

`Controller` of output port j. The signal $last_flit_info_en_j$ indicates that the last flit has arrived. While $last_flit_info_j$ signal indicates the input port id of the last flit incoming packet, $flit_length_j$ gives information about how many flits the last flit incoming packet consists of. The bit field diagram of a flit read from RAB_j is shown in Table 3.2. Bits 260 and 263 of the $RAB_data_out_j$ signal carry the input port id of the flit and are assigned to the output port $m_axis_tid_j$. Previously assigned id information for output port j determines the value of $m_axis_tdest_j$ signal. $m_axis_tlast_j$ signal points to the last flit of a communication packet and takes value according to the 256^{th} bit information, which is the last flit information in the flit from $\texttt{RAB}_j$. $m_axis_tready_j$ and $m_axis_tvalid_j$ signals are the handshaking mechanism signals in the AXI4-Stream protocol. The $m_axis_tready_j$ signal indicates that the slave side to which the switch wants to send flits is ready to accept new data. The $m_axis_tvalid_j$ signal means that new valid flit data is sent by the master side, switch here. Valid flit exchange occurs when both $m_axis_tready_j$ and $m_axis_tvalid_j$ signals are at a logic high level at the same time.

A total of 9 different reassembly buffer regions $\texttt{RAB}_{j,i}$ where $i \in (0 \dots 8)$, which are special for each input, want to send packets to output port j. Here, the Round Robin algorithm has to be a contention arbiter used to determine 1 port out of 9 input reassembly buffer regions. For `RA Scheduler`, 9 reassembly buffer regions have equal priority. For this reason, a region is prioritized for that decision cycle with the help of the decision pointer, which is updated after each decision. Decision Pointer has a circular structure that will follow the port numbers sequentially. If there is no packet to be sent in the region that the pointer prioritizes, the pointer checks whether there is a packet to be sent for the next region to complete the circle. In this way, the next reassembly buffer region that will send packets to the output port in one clock cycle time is selected. Then, the decision pointer is updated according to the selected reassembly buffer region.

A working example of $\texttt{RA Scheduler}_j$ for output port j can be seen in the Figure 3.10. $last_flit_info_i$ shows how many packets whose last flit arrived in $\text{RAB}_{j,i}$. The packets that are ready for each reassembly $\text{RAB}_{j,i}$ region are colored differently. In addition, the numbers written in the packets indicate how many flits the packets consist of. In the trapezoidal shape, the arrows indicate the $\text{RAB}_{j,i}$ regions to be

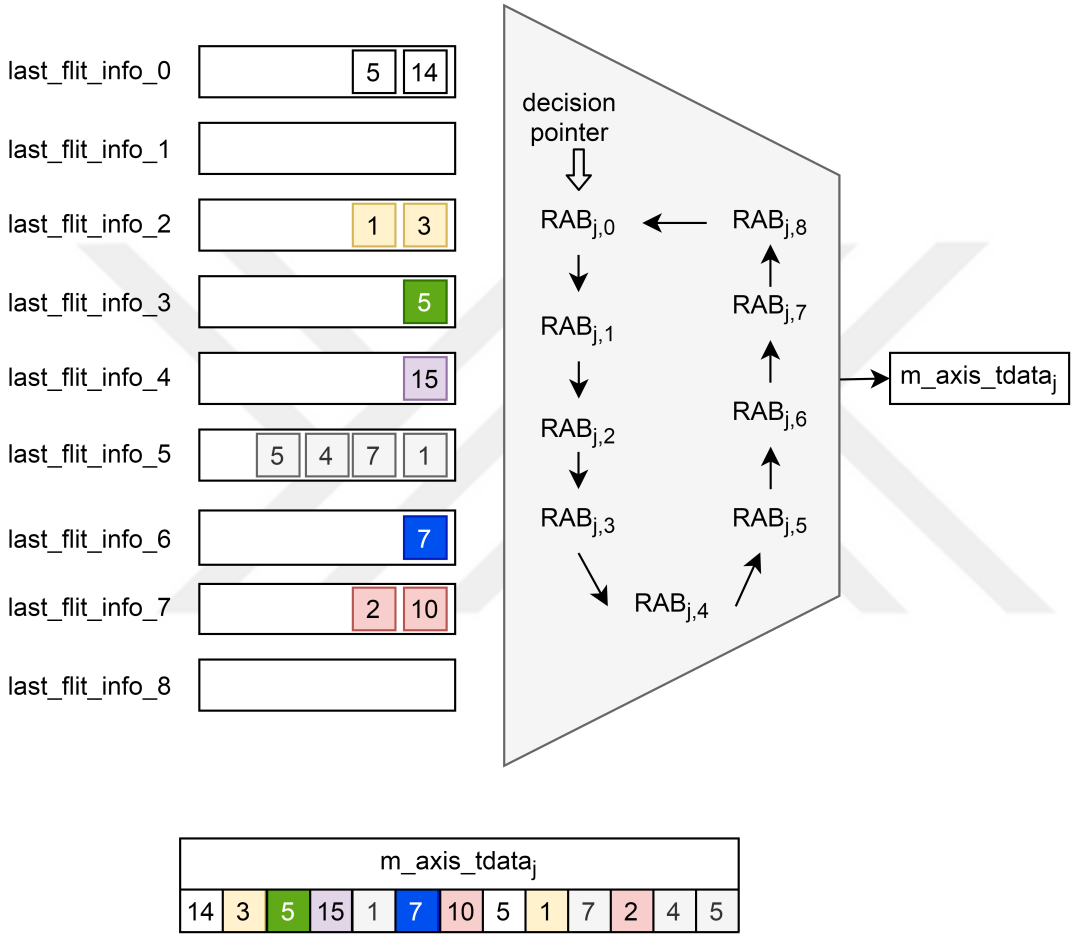


Figure 3.10: Reassembly Scheduler Arbitration Example

prioritized for the decision pointer. The initial position of the decision pointer is such that it prioritizes the $\text{RAB}_{j,0}$ region. This means that if there is a packet to be transmitted to the output port j in $\text{RAB}_{j,0}$ region during the decision-making process, $\text{RAB}_{j,0}$ region will win this decision cycle. When $\texttt{RA Scheduler}_{j,i}$ block starts working with the traffic shown, the packet containing 14 flits in the $\text{RAB}_{j,0}$ region is first sent to the output port j. Then, the decision pointer is updated to point to the $\text{RAB}_{j,1}$ region. In the next decision cycle, since there is no packet in the $\text{RAB}_{j,1}$ region, $\text{RAB}_{j,2}$ region is entitled to send a packet to the output port. Next, the decision pointer is updated to point to the $\text{RAB}_{j,3}$ region. In this way, the decision-making mechanism continues to work until the completed packets in all of the $\text{RAB}_{j,i}$ regions are sent. The order of sending the packets to the output $m_axis_tdata_j$ port is shown time sequentially in the table in Figure 3.10. The packet on the most left is the packet sent first.

3.1.8 Pipelined Switching Cycles

We implement the on-chip switch in a pipeline structure. It is indicated that 14 clock cycles for both achieving a high pipeline efficiency and frequent enough fabric arbitration to closely track the incoming data traffic is selected in Chapter 3.1. On the other hand, the speed-up of the on-chip switch is 1. It means the operating rate of the switch is equal to the line rate. Also, the 40 Gbps Ethernet IP operates at 156.25MHz with 256 bit flit length. For all these reasons, the pipeline structure is developed to run only at the rising edge of the 156.25 MHz main clock. Table 3.3, Table 3.4, and Table 3.5 show the execution of the pipeline structure. The process of a packet arriving at the input port i on the on-chip switch until it goes to the output port j is explained in detail.

In the example given, the packet arriving at the on-chip switch consists of 6 flits. Flits are labeled $\texttt{f}_n^{i,j}$, where $n \in (1 \ldots 6)$. Packet's last flit is specified as $\star\texttt{f}_n^{i,j}$. The numbers in $Cycle$ row refer to each clock cycle in the design. $\texttt{VOQ}_{i,j}$ shows the region reserved for output port j of input port i, while $\texttt{RAB}_{j,i}$, indicates the region reserved for the input port i of output port j. An e flag indicates whether the $\texttt{VOQ}_{i,j}$ regions are empty, and an f flag indicates whether the $\texttt{RAB}_{j,i}$ regions are full. A

indicates the stages of the arbiter block. In the $\mathtt{A}_{k,l}$ block, each k where $k \in (1 \ldots 3)$ value represents k^{th} iterations. $\mathtt{A}_4$ represents the matching state of `Arbiter` block whereas $\mathtt{A}_5$ is represents the waiting state.

In Cycle 1, the first flit of the 6-flit packet reaches the input port i of the on-chip switch. At this time, $\mathtt{VOQ}_{i,j}$ and $\mathtt{RAB}_{j,i}$ are empty. Therefore, $\mathtt{e}_{i,j}$ is 1 for $\mathtt{VOQ}_{i,j}$ and $\mathtt{f}_{j,i}$ is 0 for $\mathtt{RAB}_{j,i}$. In Cycle 2, the $\mathtt{VOQ\ Controller}_i$ block writes $\mathtt{f}_1^{i,j}$ flit to the corresponding $\mathtt{VOQ}_{i,j}$ region. In Cycle 3, since $\mathtt{f}_1^{i,j}$ is written to $\mathtt{VOQ}_{i,j}$, it is now $\mathtt{VOQ}_{i,j}$ region is not empty. Thus, the value of $\mathtt{e}_{i,j}$ becomes 0. In Cycle 4, $\mathtt{A}_{1,1}$, the first stage of `Arbiter` block, collects the output port requests to which the flits on the input ports want to go. In Cycle 5, output ports approve the input ports' requests according to grant pointers' values in $\mathtt{A}_{1,2}$ stage. In Cycle 6, it is first checked to see if a new iteration is needed in $\mathtt{A}_{1,3}$ stage. If a new iteration is required, the pipeline proceeds to $\mathtt{A}_{2,1}$ stage. If not, the pipeline proceeds to the $\mathtt{A}_4$ stage and waits here such that the arbitration cycle is completed at the same time for each arbiter decisions. Also, in the $\mathtt{A}_{1,3}$ stage, signal values that inform the arbiter decision to other blocks are set. In case of a new iteration is required, the requests of the matching input ports in the current iteration are discarded. The incoming flits are simultaneously written to the switch to the relevant $\mathtt{VOQ}_{i,j}$ regions by $\mathtt{VOQ\ Controller}_i$ thanks to the pipeline structure.

In Cycle 7, if `Arbiter` is in $\mathtt{A}_4$ stage, it waits here until the end of Cycle 13. If `Arbiter` is in $\mathtt{A}_{2,1}$ stage, the operations performed in $\mathtt{A}_{1,1}$, $\mathtt{A}_{1,2}$ and $\mathtt{A}_{1,3}$ stages are repeated sequentially in $\mathtt{A}_{2,1}$, $\mathtt{A}_{2,2}$ and $\mathtt{A}_{2,3}$ stages for unconnected ports. At the end of the $\mathtt{A}_{2,3}$ stage, it is checked again whether a new iteration is needed. If a new iteration is required, $\mathtt{A}_{3,1}$, $\mathtt{A}_{3,2}$ and $\mathtt{A}_{3,3}$ stages are performed for the last iteration cycle. In Cycle 13, $\mathtt{A}_4$ stage executes and the input and output port pairs information is forwarded to other blocks in the on-chip switch design, according to the arbiter decision. Before starting a new arbiter matching round, it waits for 4 cycles at the $\mathtt{A}_5$ stage during Cycle 14 to Cycle 17. This time is for the current match information to be transmitted to the other blocks in the on-chip switch and for them to prepare themselves according to the new match configuration. Because VOQ regions are implemented in BRAM structures, flits in VOQ regions are obtained 1 clock cycle after read requests are sent by `Arbiter`. Thus, `Crossbar Fabric` is not updated

until the last flit for the previous match decision read from the VOQ reaches the `Crossbar Fabric`. Also, according to the current match information, a new flit read request is sent without updating `Crossbar Fabric` so that there is no interruption between incoming flits to `Crossbar Fabric`. In other words, after the matching information is sent to the blocks in the on-chip switch, 4 clock cycles must pass before `Crossbar Fabric` structure is updated. VOQ empty signals are also updated when flits are read from VOQ regions. Requests should not be collected for the next match decision until the last flit for the previous match decision has been removed from the VOQ region. For the reasons explained, the arbiter decision is sent to all blocks in the on-chip switch in Cycle 13. At the end of Cycle 17, `Crossbar Fabric` configuration is updated. Then, a new flit transfer through the on-chip switch can start In Cycle 18.

In Cycle 18, $\mathtt{f}_1^{i,j}$ reaches the updated crossbar fabric's input port. During Cycle 19 to Cycle 24, $\mathtt{RAB\ Controller}_j$ sends incoming flits to the corresponding $\mathtt{RAB}_{j,i}$ regions. In Cycle 24, $\mathtt{Reassembly\ Scheduler}_j$ receives the information that the last flit of the relevant packet has also reached the $\mathtt{RAB}_{j,i}$ region. Since the output ports are in the AXI4-Stream standard, it is assumed that the communication auxiliary signals are valid for data transfer in order to send flit. In this case, $\mathtt{Reassembly\ Scheduler}_j$ sends a read request to $\mathtt{RAB}_{j,i}$ to transfer all the flits that make up the packet to the output port. Because RABs are also implemented as BRAMs, flits can arrive at $\mathtt{Reassembly\ Scheduler}_j$ block in Cycle 26, 1 clock cycle after read requests are sent. During Cycle 27 to Cycle 32, flits read from $\mathtt{RAB}_{j,i}$ are sent from the output ports to the destinations in accordance with the AXI4-Stream standard.

To summarize, this section describes the movement of a packet containing 6 flits through the switch from the packet that arrives at the input ports of the switch until the packet is sent to the output ports of the on-chip switch.

Table 3.3: On-chip Switch Pipeline Stages - 1

Cycle	1	2	3	4	5	6	7	8	9	10	11
Input	$f_1^{i,j}$	$f_2^{i,j}$	$f_3^{i,j}$	$f_4^{i,j}$	$f_5^{i,j}$	$*f_6^{i,j}$					
VOQ Controller		$VOQ_{i,j} \leftarrow f_1^{i,j}$	$VOQ_{i,j} \leftarrow f_2^{i,j}$	$VOQ_{i,j} \leftarrow f_3^{i,j}$	$VOQ_{i,j} \leftarrow f_4^{i,j}$	$VOQ_{i,j} \leftarrow f_5^{i,j}$	$VOQ_{i,j} \leftarrow *f_6^{i,j}$				
VOQ	$e_{i,j} \leftarrow 1$		$e_{i,j} \leftarrow 0$								
Arbiter				$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{2,1}/A_4$	$A_{2,2}/A_4$	$A_{2,3}/A_4$	$A_{3,1}/A_4$	$A_{3,2}/A_4$
XBAR											
RAB Controller											
RAB	$f_{j,i} \leftarrow 0$										
RA Scheduler											
Output											

Table 3.4: On-chip Switch Pipeline Stages - 2

Cycle	12	13	14	15	16	17	18	19	20	21	22
Input											
VOQ Controller											
VOQ											
Arbiter	$A_{3,3}/A_4$	A_4	A_5	A_5	A_5	A_5					
XBAR						config	$f_1^{i,j}$	$f_2^{i,j}$	$f_3^{i,j}$	$f_4^{i,j}$	$f_5^{i,j}$
RAB Controller								$RAB_{j,i} \leftarrow f_1^{i,j}$	$RAB_{j,i} \leftarrow f_2^{i,j}$	$RAB_{j,i} \leftarrow f_3^{i,j}$	$RAB_{j,i} \leftarrow f_4^{i,j}$
RAB											
RA Scheduler											
Output											

Table 3.5: On-chip Switch Pipeline Stages - 3

Cycle	23	24	25	26	27	28	29	30	31	32	33
Input											
VOQ Controller											
VOQ	$e_{i,j} \leftarrow 1$										
Arbiter											
XBAR	$\leftarrow * f_6^{i,j}$										
RAB Controller	$RAB_{j,i}$ $\leftarrow f_5^{i,j}$	$RAB_{j,i}$ $\leftarrow * f_6^{i,j}$									
RAB											
RA Scheduler		last flit		$RAB_{j,i}$ $\rightarrow f_1^{i,j}$	$RAB_{j,i}$ $\rightarrow f_2^{i,j}$	$RAB_{j,i}$ $\rightarrow f_3^{i,j}$	$RAB_{j,i}$ $\rightarrow f_4^{i,j}$	$RAB_{j,i}$ $\rightarrow f_5^{i,j}$	$RAB_{j,i}$ $\rightarrow * f_6^{i,j}$		
Output					$f_1^{i,j}$	$f_2^{i,j}$	$f_3^{i,j}$	$f_4^{i,j}$	$f_5^{i,j}$	$* f_6^{i,j}$	

CHAPTER 4

EVALUATION

4.1 Performance Evaluation

The on-chip switch design in this thesis has 9x9 input and output ports. In addition, the arbiter method is the basic Dual Round Robin (DRR) [18] with 3 iterations. All ports are running at the line speed of 40 Gbps. We evaluate the proposed and implemented on-chip switch for its functional correctness and performance. To this end, we perform a systematic verification procedure. These design source codes are generated and synthesized using the Vivado 2020.2 tool, and resource consumption is obtained through this tool. Every sub-block is tested for functional correctness by its own testbench in Vivado. Then, in Modelsim SE-64 10.1d, we verify the overall on-chip switch design by a testbench using the SystemVerilog verification architecture. Scoreboard and coverage reports are obtained from Modelsim SE-64 tool. We then evaluate the throughput and latency of the switch under different traffic loads.

4.1.1 Verification of the On-chip Switch

Designs should be tested throughout the development process to verify that they have the desired functionality. The on-chip switch design, detailed in Chapter 3, is functionally verified. During the functional test of the switch, the sub-blocks that compose the switch are tested one by one with their own testbenches and it is checked whether they meet the desired function or not. However, the fact that all blocks can achieve the desired task on their own does not mean that the overall design will work as desired functionally. At the same time, there is a need to know whether the design can be tested under the desired conditions. For these reasons, a verification design

should be developed for the overall design of the switch.

The on-chip switch design is developed with Vivado 2020.2 tool. Behavioral simulations of the sub-blocks that compose on-chip switch are performed with testbenches created in the internal simulator in the Vivado 2020.2 tool. The tests of the sub-blocks are achieved thanks to the signal wave-forms that are output from the simulator. Overall switch verification test bench is developed using the SystemVerilog verification architecture. The testbench structure developed for the switch verification can be seen in Figure 4.1. Modelsim SE-64 10.1d is used to verify the overall on-chip switch design and obtain coverage results.

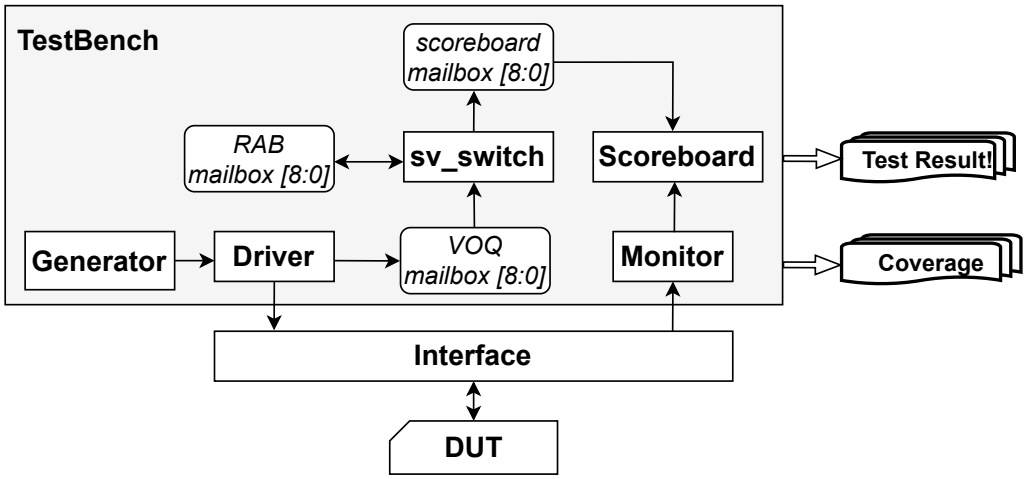


Figure 4.1: Switch Testbench Architecture

Thanks to `Generator`, the input data to be sent to the on-chip switch design (DUT) is generated. `Generator` is a class of various functions in it to generate input data. Since the on-chip switch input ports accept data by acting as AXI4-Stream slaves, the necessary signals for this interface are generated through the functions in `Generator`. s_tid, s_tdest, s_tdata, s_tkeep, s_tlast and s_tvalid are required to provide the data generated for testing purposes in an appropriate AXI4-Stream format. To achieve this, there are 4 functions in `Generator` class. get_s_tid function prepares the previously defined identification information for each input port. get_s_tdest is a function that generates random identification numbers that indicate which output block the incoming data wants to go to. Since the switch has 9 input and output ports, it randomly generates a value within 9 different port identification

numbers. The get_s_tdata is a function that randomly generates 256 bits of data. The task of the get_s_tkeep function is to generate random data for the keep signal, which specifies the information of the current bytes in the data. get_packet_length function randomly generates the total number of flit numbers in a packet to be sent to the switch for testing purposes. The level of the s_tlast and s_tvalid signals should be set by `Driver` when sending the generated flits to the DUT.

The randomly generated input data in the `Generator` is transmitted to the DUT by `Driver`. `Driver` class is responsible for sending data to the on-chip switch at the appropriate timing in a loop. A `Generator` class instance is created in `Driver` class. s_tid, s_tdest, s_tdata and s_tkeep signals and flit length information are generated randomly from the functions in `Generator` using this instance. After s_tdest and flit length information are generated randomly once, the value of s_tdest remains constant until all flits of that packet are sent. Flit length information is reduced at each transmitted clock cycle. Data and keep values are generated randomly in each clock cycle. When the flit length number to be sent is 0, s_tdest and flit length information are generated randomly again. After the desired signals are produced, s_tready signal at the input ports of the DUT is sampled in accordance with the AXI4-Stream standard. If there is a valid s_tready signal, that is, if the value of s_tready is logic high and the flit length number to be sent is greater than 0, the data is transmitted to the DUT. In addition, the value of the s_tlast and s_tvalid signals in each clock cycle is determined by `Driver`. With the help of the flit information to be sent, appropriate values for these signals are determined. If the remaining flit length information to be sent is greater than 0, s_tvalid signal is set as logic high, otherwise, it is set as logic low. s_tlast signal is set to logic high while the last remaining flit is sent. In other cases, s_tlast signal has a logic low level.

In addition, `Driver` sends the same data sent to the DUT to the `VOQ mailbox` in the testbench. Thus, the same test data can be sent to the golden model and device under test in the testbench for verification. It is the task of `Driver` to send the incoming flits to the relevant `VOQ mailbox` if there is free space. Thus, the task of the *VOQ Controller* block on the on-chip switch is modeled in this way.

Mailboxes are containers used in SystemVerilog to exchange data between processes. Mailboxes work with the first in first out method. While the process that wants to send data puts the desired data in a mailbox, the process that wants to receive the data can get this data from the mailbox. Mailboxes are used to model block RAMs in the on-chip switch design. `VOQ mailbox` acts as virtual output queue structures used to avoid the head of line blocking problem in the on-chip switch. `Driver` puts test data generated in `Generator` into a container named `VOQ mailbox` to be sent to `sv_switch`, the testbench switch model. Each input port on the on-chip switch has a dedicated `VOQ mailbox`. The depth of `VOQ mailboxes` can be changed with a generic parameter. In addition, the ability to check the number of flits in the mailboxes at any time allows the differences that occur during the verification process to be found quickly.

Packets split into flits at inputs of the on-chip switch are stored to reassembly at the output ports. `RAB mailbox`, on the other hand, is used to model the structures of reassembly buffers. `RAB mailbox` have the same features as `VOQ mailbox`. Since the on-chip switch has 9 output ports, the verification testbench of the on-chip switch has a total of 9 `RAB mailboxes`.

After the input data is processed in the DUT, the testbench needs golden results or another verification model with the same characteristics as the on-chip switch to check the correctness of the output data of the DUT. For this reason, `sv_switch` class has been developed. The `sv_switch` works in coordination with the DUT in each clock cycle. *Crossbar Fabric*, *Arbiter (DRR with 3 iterations)*, *RAB Controller* and *RA Scheduler* sub-blocks of the on-chip switch in Chapter 3 are performed in parallel in `sv_switch`. Input flits received from `VOQ mailbox` are prioritized with the DRR method, passed through the crossbar fabric, and stored in `RAB mailbox`. Packets with all flits passed through the crossbar fabric are reassembled and sent to the output port. On the other hand, information of dropped flits is also obtained in `sv_switch` since there is no free space in the `RAB mailboxes`.

`scoreboard mailbox` is the mailbox where the expected data from output ports of DUT sent from `sv_switch` are stored. `Scoreboard` compares the experimental

DUT results obtained by `Monitor` and the theoretically expected results stored in the `scoreboard mailbox`. As a result of the comparison, the verification score of the DUT performance is calculated. `Scoreboard` also makes it possible to observe the number of flits compared, the number of successful/failed results, the number of flits dropped on the switch, and the contents of all flits. It then prints a summary report containing this information and the test result to a console screen. If the verification test result is unsuccessful, some guiding information about why the flits could not pass through the switch may also be included in the result report.

While the verification test is in progress, the instantaneous comparison result of `sv_switch` and DUT results may also be observed with the outputs printed on the console screen. Figure 4.2 shows a sample instant console output. The message with the "SUCCESS" tag contains information that the `sv_switch` and DUT outputs are consistent. The instant flit value is also displayed. The message with the "ERROR" tag indicates a difference between the `sv_switch` and DUT output. Also, thanks to this message, the output flit values of the testbench model and DUT can be seen. Another instantaneous message is the information of the dropped flits on the switch. It is also reported that the dropped flit goes from which input port to which output port. In addition, the data value of this flit is also printed on the console screen.

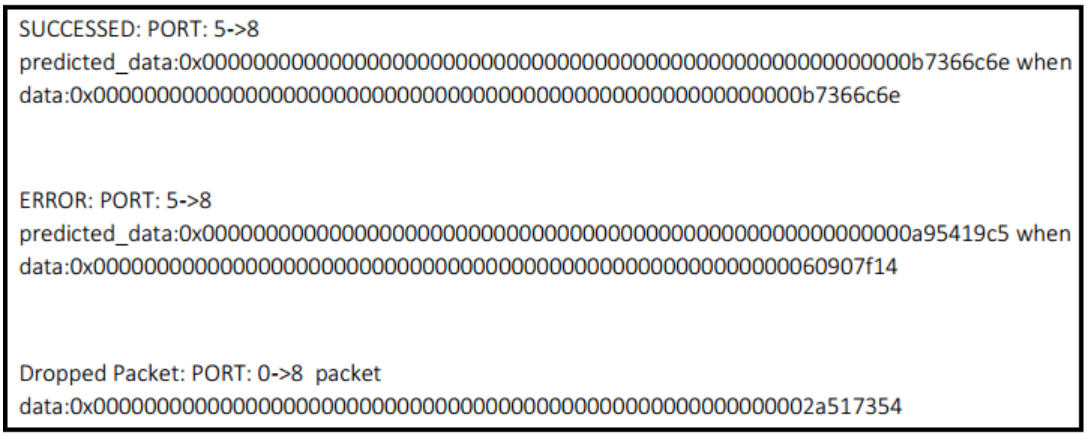


Figure 4.2: Instant Console Display Messages of Verification Test

In addition, it is important to monitor whether the design can be tested with the desired input conditions. For this reason, the desired input conditions are determined as a coverage rule in `Coverage` class before the test. Also, at the end of the test, the test

coverage results can be analyzed in detail as a report of whether each of the specified conditions was hit.

The tested on-chip switch (DUT) and switch model of the testbench should have exactly the same functional characteristics. However, after the model is developed in SystemVerilog, there may be cases where there is a difference between the DUT and `sv_switch` results at first. In this case, one or both of the results may be incorrect. At this point, first of all, output values for test input data should be checked manually. Thus, it is determined which result is wrong. `sv_switch` has been developed in an object-oriented way. In addition, thanks to the SystemVerilog console outputs, messages can be printed visually at any time. For these reasons, in case of inconsistency between theoretical and experimental results, it is easier to first check `sv_switch` model in SystemVerilog. If there is an error with the theoretical output values in `sv_switch`, the verification test should be repeated after `sv_switch` has been modified. On the other hand, if the output values in `sv_switch` are correct, the on-chip switch design needs to be analyzed. The source of the error in the on-chip switch may be found with the help of a simulation waveform. After the error in the on-chip switch design is corrected, the test should be repeated. If there is still a difference between the theoretical and experimental results, the steps described above are repeated until the testbench model and DUT results are consistent. After modifying the design with error, the test should be repeated with the same test input data where the difference is found. Although `Generator` generates the input data randomly, a specific seed value is used for randomness. Therefore, it is possible to repeat a verification test with the same test input data, thanks to its own seed value.

The verification report first shows the percentage of completion of the test. A test completion percentage of 100% means that all test data created in `Generator` is processed in both the DUT and `sv_switch`. In the test report, $Total\ Number\ of\ Generated\ Input\ Flits$ represents the total number of input test flits generated in `Generator`. $Total\ Number\ of\ Tested\ Flits$ indicates the number of experimental (DUT) and theoretical (sv_switch) output flits compared in `Scoreboard`. The value of $Total\ Number\ of\ Tested\ Flits$ is calculated by summing the values $Number\ of\ Successful\ Flits$, $Number\ of\ Failed\ Flits$, $Number\ of\ Dropped\ Flits$, and $Number\ of\ Flits\ on\ switch$. If the output value of experimental and theoretical

results have the same data for a flit, this is considered successful. The total number of successful flits according to the test result is indicated by the $Number\ of\ Successful\ Flits$. If there is a difference between the instantaneous theoretical and experimental output results, it is evaluated as failed. $Number\ of\ Failed\ Flits$ gives the total number of different output flits between results. The value of $Number\ of\ Failed\ Flits$ must be "0". If this value is different from "0", it means that the on-chip switch used as DUT and its verification model `sv_switch` designs have functionally different characteristics. If this value is different than "0", the test fails. The $Number\ of\ Dropped\ Flits$ indicates the number of dropped flits as they pass through the switch. Dropping of flits on the switch usually occurs when *VOQ* or *RAB* block RAMs are full. In these cases, if a new flit is desired to be written, the desired flit is dropped because there is no empty space. In Chapter 3.1.7, it is explained that *Reassembly Scheduler* block reassembles the packet divided into flits after the last flit of the respective packet arrives in *RAB RAM*. If the last flit of a packet is dropped before it is written to *RAB RAM*, other flits in the packet wait for the last flit of the following packet to be reassembled. $Number\ of\ Flits\ on\ switch$ displays the number of flits remaining in *RAB RAM* due to the last flit dropped. There may be some flits remaining in *RAB RAMs* in the switch when the test is finished before the last flit of the following packet arrives at the same output port. Test results are determined according to the numbers explained in detail so far. There are two different reasons for the test result to be determined as "FAILED". The first reason is that $Number\ of\ Failed\ Flits$ is greater than "0". The other reason is that $Total\ Number\ of\ Generated\ Input\ Flits$ and $Total\ Number\ of\ Tested\ Flits$ are not equal even if $Number\ of\ Failed\ Flits$ is equal to 0. This means that more flits than expected are in the switch or dropped across the switch. When $Total\ Number\ of\ Generated\ Input\ Flits$ and $Total\ Number\ of\ Tested\ Flits$ are equal to each other and $Number\ of\ Failed\ Flits$ value equal to "0", the test result is determined as "SUCCESSFUL".

A test result of $PASSED$ means that the on-chip switch and its SystemVerilog model, `sv_switch`, are functionally identical. However, this result does not mean that all flits are successfully passed through the on-chip switch. Because FPGA has a limited hardware resource, there is no unlimited block RAM for *VOQ* and *RAB*. For this reason, there may be dropped flits while transmitting packets through the

switch. For such a situation, some feedback is given about the flits dropping in the switch or the flits remaining in the switch as a result of the verification test. Thus, the shortcomings of the on-chip switch design are shown as a result of the test. Also, the switch verification test report includes information about the remaining flit numbers in *RAB* RAMs.

We perform the following experiment for the verification of a DUT, whose $\texttt{VOQ}_{i,j}$ depth is 112 flits for $j \in (0 \ldots 3)$ and 320 flits for $j \in (4 \ldots 8)$ for all input ports is evaluated. The DUT also has $\texttt{RAB}_{j,i}$ regions with 113 flits depth for each input port $i \in (0 \ldots 8)$ in all output ports. While selecting the depths of VOQ_i for $i \in (0 \ldots 8)$ and RAB_j for $j \in (0 \ldots 8)$ block RAMs, care is taken to select areas that are close to the powers of 2 to ensure efficient FPGA resource consumption. In the verification process, the test lasts a total of 14070 clock cycles. The experiment report printed on the Modelsim SE-64 console as a result of the verification test is shown in Figure 4.3.

Firstly, we perform the following experiment for the verification of a DUT, whose $\texttt{VOQ}_{i,j}$ depth is 112 flits for $j \in (0 \ldots 3)$ and 320 flits for $j \in (4 \ldots 8)$ for all input ports is evaluated. The DUT also has $\texttt{RAB}_{j,i}$ regions with 113 flits depth for each input port $i \in (0 \ldots 8)$ in all output ports. While selecting the depths of VOQ_i for $i \in (0 \ldots 8)$ and RAB_j for $j \in (0 \ldots 8)$ block RAMs, care is taken to select areas that are close to the powers of 2 to ensure efficient FPGA resource consumption. Although the application running on the reconfigurable regions (RR) and the data generated in the RR are under the designer's control, we assume that this is not the case in data communication made from other interfaces. According to the simulator results we presented in [30], an average of 16.21 cell queues is observed under 95% maximum load. For these reasons, block RAM with 2048 depth is used for each input port $i \in (0 \ldots 8)$ VOQ_i. 112 of 2048 depth are allocated for the designer-controlled RR inputs whereas 320 of 2048 depth for the other input ports. For the output ports used in this experiment, each RAB_j size is chosen to be 1024. 1024 depth is allocated to equal regions with 113 depth for 9 output ports. In the verification process, the test lasts a total of 14070 clock cycles. The experiment report printed on the Modelsim SE-64 console as a result of the verification test is shown in Figure 4.3.

```
---------- SWITCH TEST IS FINISHED! ----------
 100% completed!

---------- Test Report ----------
____Total Number of Generated Input Flits = 90075
____Total Number of Tested Flits          = 90075
__Number of Successful Flits   = 90075
__Number of Failed Flits       =     0
__Number of Dropped Flits      =     0
__Number of Flits on switch    =     0

-----> Test result : PASSED!
-All input flits are transferred to output ports succesfully.
-------------------------------------------------

RAB RAM Flits details:
Number of Flits in RAB RAM[ 0] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 1] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 2] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 3] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 4] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 5] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 6] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 7] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 8] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.3: Verification Test Report of the First Experiment

In this experiment, random 90075 flits are generated to create uniform traffic and the DUT is tested with these flits. The number of flits tested is 90075. Also, all `sv_switch` and DUT output flits are exactly the same. On the other hand, no flits passing through the switch are dropped or remain in the storage areas on the switch. Thus, when these results were evaluated, the verification test result is determined as $PASSED$. In addition, the numbers of flits produced for the test are given in Table 4.2. It is observed that approximately 10000 flits are produced for each port. Thus, each port is faced a similar traffic load.

Table 4.1: Generated Test Input Flit Numbers for Input Ports

Input Port	Number of Tested Flits
0	9864
1	9844
2	10034
3	10023
4	10242
5	10020
6	9989
7	9937
8	10122

According to the verification test result, the on-chip switch successfully forwards all the flits from the input ports to the output ports. However, this does not give any information about the test input data diversity. For this reason, while performing the verification test, the `Coverage` test is also performed. The test data produced in `Generator` is sent to the DUT over the interface. `Coverage` observes the test data by sampling the input lines of the DUT in the interface. It can be observed whether the previously defined conditions as coverage are met, or if so, how many times the specified condition has been hit. Therefore, the coverage test result is as important as the verification test result for the design verification process.

For this experiment, two different covergroups are defined in `Coverage`. These covergroups are $source_coverage$ and $destination_coverage$. There are nine coverpoints in the $source_coverage$ covergroup. Each coverpoint samples the

$s_axis_tid_i$ signal where $i \in (0 \dots 8)$ when the $s_axis_tvalid_i$ signal is logic high. Thus, it is monitored whether any flit is sent to the input ports and if so how many flits are sent. In $destination_coverage$ there is a separate coverpoint for each input line $s_axis_tdest_i$ where $i \in (0 \dots 8)$. There are 9 bins in each coverpoint. Each bin value represents the output port number j where $j \in (0 \dots 8)$. Thus, the condition of sending flits to 9 different outputs for the input port i can be observed thanks to the bins. Additionally, another coverpoint named $cross_s_axis_tdest$ is defined in $destination_coverage$. $cross_s_axis_tdest$ cross-samples $s_axis_valid_i$ signals. Thus, information on how many of the 9 input ports have flit at the same time can be found. The $cross_s_axis_tdest$ coverpoint automatically contains as much as 2 to the power 9 bins for different conditions.

The coverage report for the covergroup $source_coverage$ is shown in Figure 4.4. CVP stands for coverpoint. 100% coverage is achieved for each CVP. This means that data has been successfully driven from all input ports. The coverage report for $destination_coverage$' is shown in Figure 4.5. It is observed from the 100% coverage result that all of the $s_axis_tdest_i$ CVPs are hit. Also, $cross_s_axis_tdest$ coverage is 100%. This means that the design is driven with the test data coming in 2 to the power 9 different input conditions. More detailed test results showing bin conditions can also be generated as a coverage report. However, coverage information can be seen in summary in Figure 4.4 and Figure 4.5.

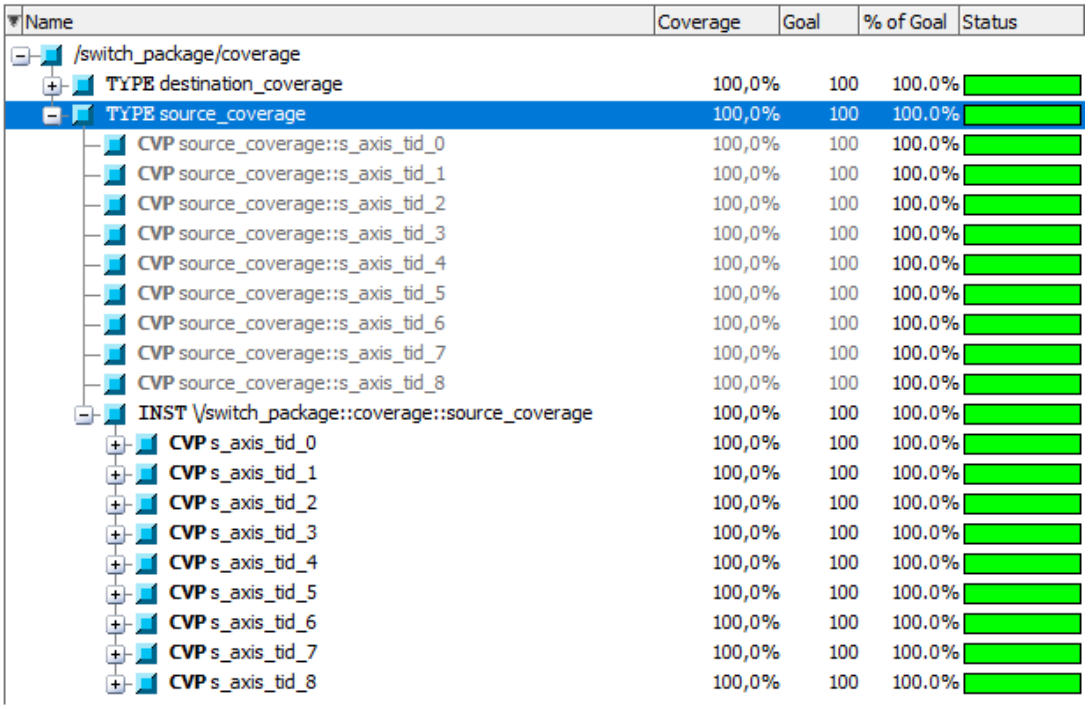
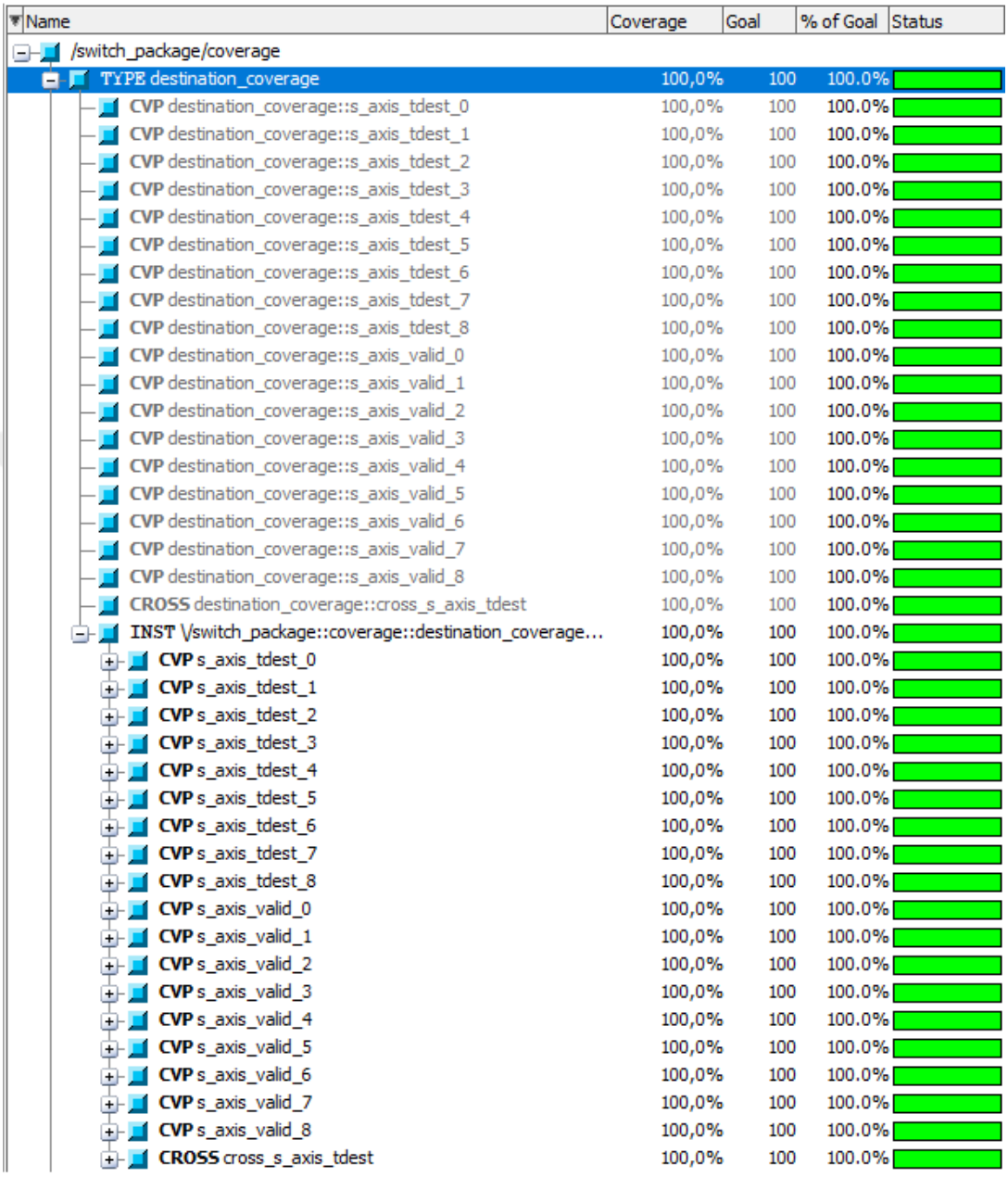


Figure 4.4: Coverage Report for *source_coverage*



Name	Coverage	Goal	% of Goal	Status
/switch_package/coverage				
TYPE destination_coverage	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_0	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_1	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_2	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_3	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_4	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_5	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_6	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_7	100,0%	100	100.0%	
CVP destination_coverage::s_axis_tdest_8	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_0	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_1	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_2	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_3	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_4	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_5	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_6	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_7	100,0%	100	100.0%	
CVP destination_coverage::s_axis_valid_8	100,0%	100	100.0%	
CROSS destination_coverage::cross_s_axis_tdest	100,0%	100	100.0%	
INST \/switch_package::coverage::destination_coverage...	100,0%	100	100.0%	
CVP s_axis_tdest_0	100,0%	100	100.0%	
CVP s_axis_tdest_1	100,0%	100	100.0%	
CVP s_axis_tdest_2	100,0%	100	100.0%	
CVP s_axis_tdest_3	100,0%	100	100.0%	
CVP s_axis_tdest_4	100,0%	100	100.0%	
CVP s_axis_tdest_5	100,0%	100	100.0%	
CVP s_axis_tdest_6	100,0%	100	100.0%	
CVP s_axis_tdest_7	100,0%	100	100.0%	
CVP s_axis_tdest_8	100,0%	100	100.0%	
CVP s_axis_valid_0	100,0%	100	100.0%	
CVP s_axis_valid_1	100,0%	100	100.0%	
CVP s_axis_valid_2	100,0%	100	100.0%	
CVP s_axis_valid_3	100,0%	100	100.0%	
CVP s_axis_valid_4	100,0%	100	100.0%	
CVP s_axis_valid_5	100,0%	100	100.0%	
CVP s_axis_valid_6	100,0%	100	100.0%	
CVP s_axis_valid_7	100,0%	100	100.0%	
CVP s_axis_valid_8	100,0%	100	100.0%	
CROSS cross_s_axis_tdest	100,0%	100	100.0%	

Figure 4.5: Coverage Report for *destination_coverage*

The second experiment performs another verification test of the on-chip switch. In some situations, all flits are not successfully passed through on-chip switch since FPGA has a limited hardware resource, there is no unlimited block RAM for *VOQ* and *RAB*. For this reason, there may be dropped flits while transmitting packets through the switch. In this experiment, the verification test result is shown when the Reassembly Buffer (RAB) depth is insufficient and therefore some flits are dropped. To see the verification test result for the case of flit drops, the switch is tested with an offered load close to 100%. As explained reasons in the previous experiment, the depth of $\texttt{VOQ}_{i,j}$ on all input ports is 112 flits for output ports $j \in (0 \ldots 3)$ and 320 flits for output ports $j \in (4 \ldots 8)$. DUT also has $\texttt{RAB}_{j,i}$ regions with a depth of 113 flits for input ports $i \in (0 \ldots 8)$ on all output ports. The verification test result is shown in Figure 4.6.

```
----------------- SWITCH TEST IS FINISHED! -----------------
 100% completed!

---------- Test Report ----------
____Total Number of Generated Input Flits = 14761
____Total Number of Tested Flits          = 14761
__Number of Successful Flits   = 14752
__Number of Failed Flits       =     0
__Number of Dropped Flits      =     7
__Number of Flits on switch    =     2

-----> Test result : PASSED!
- Some input flits are dropped on switch. Please increase RAB RAM size.
- Each flit's test history on switch can be seen on console prints.

RAB RAM Flits details:
Number of Flits in RAB RAM[ 0] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 1] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 2] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 3] ->  0, 0, 0, 0, 0, 0, 0, 2, 0
Number of Flits in RAB RAM[ 4] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 5] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 6] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 7] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 8] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.6: Verification Test Report for RAB with 113 flits

During the test, 14761 flits arrive at the switch. 14752 flits are successfully transmitted across the switch and sent to the output lines. On the other hand, 7 flits are dropped in the switch. When more detailed information with dropped flits is examined from the instant test console outputs, it is seen that there are flits that want to go from input port 7 to output port 3. Also, $Number\ of\ Flits\ on\ Switch$ is 2. It shows the remaining flits in the switch as a result of the test. There is also warning information about these flits in the test report. In addition, it is seen in detail in which RAB regions these flits are located. The remaining flits in the switch are the flits that cannot be sent since the last flit of the relevant packet has been dropped. Then, these flits will be sent to the output line when the successful last flit in the next packet, which wants to go from the same input port to the same output port as them, arrives in RAB. The test result is $PASSED$ because the experimental results and theoretical results are consistent with each other.

Since the depth of the RAB regions is limited, the dropped flit information on the switch can be seen in Figure 4.6. To prove the accuracy of this inference, a test is performed with different RAB depths. In the third example, the second experiment test conditions are repeated with the same test data for RAB regions with greater depth. This time, the switch has $\texttt{RAB}_j$ regions with a depth of 2048 flits, twice the depth in the second experiment in all output ports. 2048 depth is allocated to equal regions with 227 depth for 9 input ports. Hence, the switch has $\texttt{RAB}_{j,i}$ regions with a depth of 227 flits for $i \in (0 \dots 8)$ input ports in all output ports.

This verification test result is shown in Figure 4.7. When the test is repeated with the same traffic and input data, it is seen that all 14761 flits are successful. $Number\ of\ Failed\ Flits$, $Number\ of\ Dropped\ Flits$, and $Number\ of\ Flits\ on\ switch$ values are all 0. Therefore, as a result of all these values, the verification test result is determined as $PASSED$. Thus, when the switch is faced with an offered load close to 100%, it is seen that the packages drop at the switch after a certain time due to the limited depth of RAB.

```
---------------- SWITCH TEST IS FINISHED! -----------------
 100% completed!

---------- Test Report ----------
____Total Number of Generated Input Flits = 14761
____Total Number of Tested Flits          = 14761
__Number of Successful Flits   = 14761
__Number of Failed Flits       =     0
__Number of Dropped Flits      =     0
__Number of Flits on switch    =     0

-----> Test result : PASSED!
-All input flits are transferred to output ports succesfully.
-------------------------------------------------

RAB RAM Flits details:
Number of Flits in RAB RAM[ 0] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 1] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 2] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 3] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 4] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 5] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 6] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 7] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
Number of Flits in RAB RAM[ 8] ->  0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 4.7: Verification Test Report for RAB with 227 flits

4.1.2 Performance of the On-chip Switch

The performance of the high speed on-chip switch is evaluated by keeping statistical information in the verification infrastructure developed with SystemVerilog in Modelsim tool. We perform the next experiments to evaluate the $flit\ latency$ and $throughput$ performance of the on-chip switch. In this experiment, as explained in Chapter 4.1.1, the depth of $\texttt{VOQ}_{i,j}$ is 112 flits for each output port $j \in (0 \ldots 3)$ and 320 flits for for each output port $j \in (4 \ldots 8)$ for all input ports is evaluated. The switch has $\texttt{RAB}_{j,i}$ regions with 113 flits depth for each input port $i \in (0 \ldots 8)$ in all output ports.

The rate of the traffic in bps arriving on the 40 Gbps input lines is called $offered\ load$. In other words, the rate of incoming flits divided by the line rate is calculated as offered load. Flit latency is calculated for a flit as the difference between the time the flit is fed from the switch input port and the time it is observed at the switch output port. $Average\ flit\ latency$ value is obtained by calculating the average of the flit latency values for each flit. Average flit latency values measured under different offered load is shown in Figure 4.8. As seen in Figure 4.8, similar average flit delay measurements are obtained with respect to the line load. As the offered load increases, the value of average flit latency increases as expected. In the case of offered load above 80%, the value of average flit latency increases more. In the on-chip switch, a pipeline structure is used for the transmission of flits over the switch. In other words, the transmission of flits and the next matching decision are two different processes that occur simultaneously. Dual Round Robin with 3 iterations is used as the arbiter method of the switch. Thanks to the pipeline structure, until the decision-making stages are completed, the transfer of the flits is made according to the matching result of the previous decision. Therefore, a matching decision is made and the current input-output matching continues until the next decision. For this reason, flit latency does not increase much as the offered load increases until a certain input traffic load. But after a certain traffic load, 80% in Figure 4.8, a dramatic increase in average flit latency is observed as the offered load increases. The reason for this delay is the increase in the offered load from all input ports to all output ports. Thanks to the DRR method, when an input port is matched with an output port, the pointers are

mutually updated with the least priority for the most recently matched port. In cases where the input traffic load is high after an input port matches with a specific output port, the time taken for new matching of the same ports increases. In other words, the average flit latency of the on-chip switch increases dramatically to a large value because of the saturation as a result of decreased matching efficiency for DRR. In summary, as expected, the increase in average flit latency with respect to the traffic load to the on-chip switch is seen in Figure 4.8.

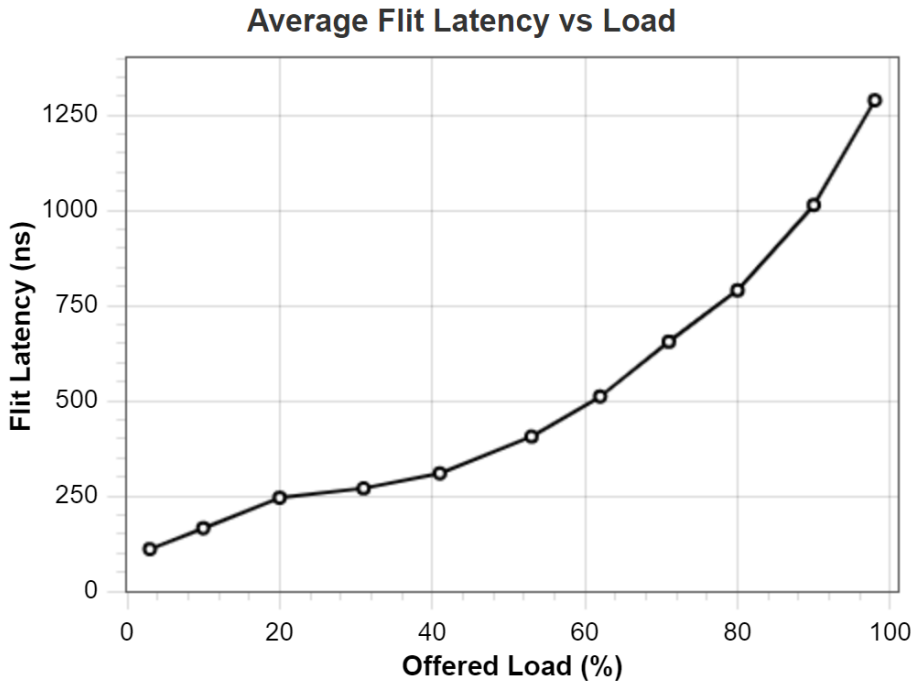


Figure 4.8: Average Flit Latency of On-chip Switch under Uniform Traffic

In the next experiment, $throughput$ values under different load values are evaluated. Figure 4.9 shows the throughput of an output port j with DRR under uniformly distributed input traffic load. DRR arbiter is used with 3 iterations. Throughput is calculated as the rate of coming out bits per second from the output port j. Offered load is measured as the rate of incoming bits to input ports are divided by the rate of the full capacity of input ports, then multiplied by 100 to obtain a percentage. Under uniform input traffic, as offered load increases the value of throughput also increases as expected. When the offered load is about 100%, the throughput is also very close to 40 Gbps output port j. This result shows DRR method used in the on-chip switch is work conserving.

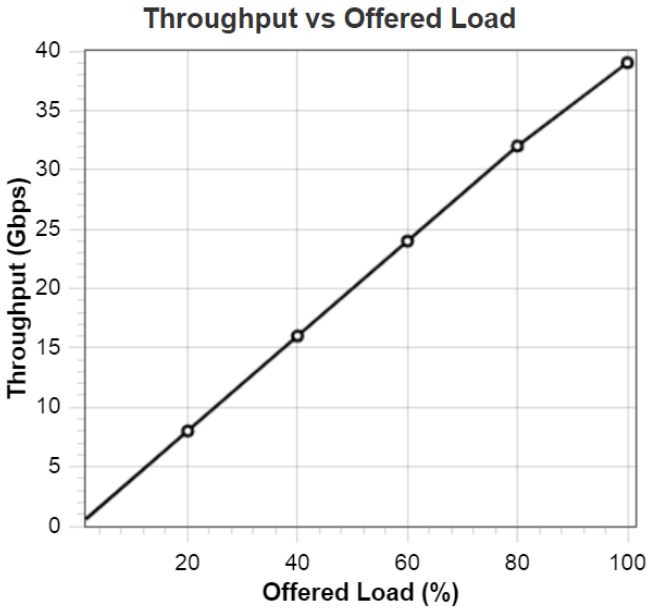


Figure 4.9: Throughput of On-chip Switch under Uniform Traffic

Finally, an experiment is conducted with different numbers of flits to observe the number of dropped flits according to the depth of RAB. During the test, the input lines are driven at 40 Gbps line speed under 100% offered load. All tests are repeated with the same seed value for RAB BRAMs with depths of 512, 1024, and 2048. Thus, designs with 3 different RAB depths are tested with the same input test loads. All variables are the same in the designs, except for the RAB depths. For BRAM = 512 case, the switch has $\texttt{RAB}_{j,i}$ regions with 56 flits depth for each input port $i \in (0 \ldots 8)$ in all output ports. For BRAM = 1024 case, the switch has $\texttt{RAB}_{j,i}$ regions with 113 flits depth for each input port $i \in (0 \ldots 8)$ in all output ports. For BRAM = 2048 case, the switch has $\texttt{RAB}_{j,i}$ regions with 227 flits depth for each input port $i \in (0 \ldots 8)$ in all output ports.

In Figure 4.10, dropped flit numbers are shown according to different RAB BRAM depths. As expected, as RAB depth increases, the number of dropped flits at the same input load decreases. Due to the limited BRAM resource, flit drops occur after a while under a continuous 100% offered load. According to the predicted offered load, the appropriate RAB depth can be determined by simulations.

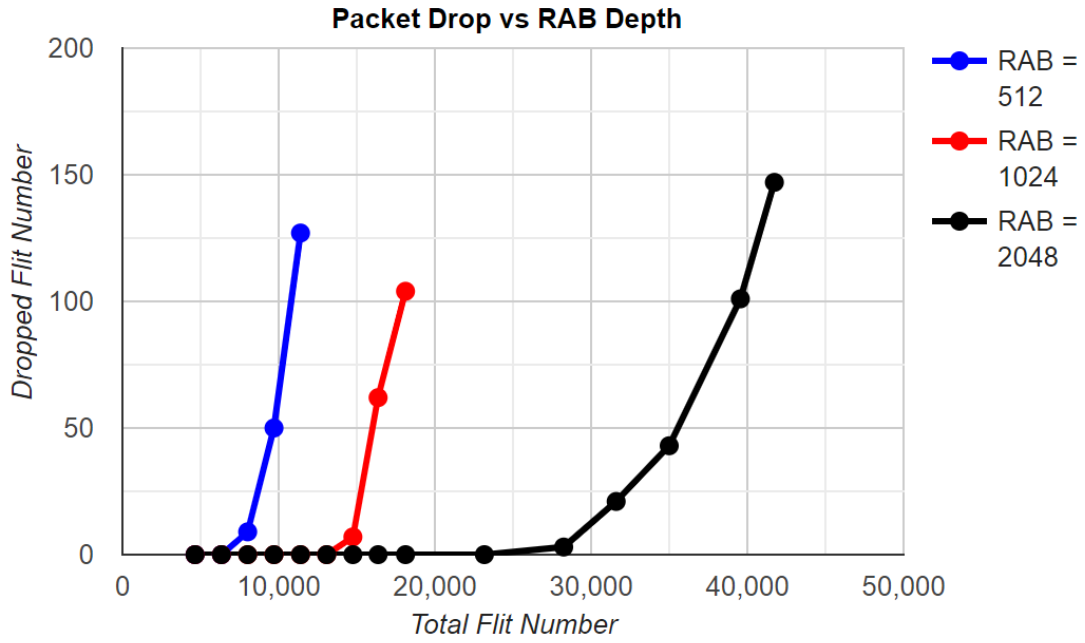


Figure 4.10: Packet Drop vs RAB Depth

4.2 FPGA Hardware Implementation Evaluation

The on-chip switch design is developed in Vivado 2020.2 tool using Verilog language. After the design verification test is successful, the codes are synthesized in the Vivado tool by selecting the XC7Z100FFG1156-2 SoC of the Xilinx Zynq-7000 family as the target. The design operating clock frequency is 156.25 MHz. There are available 277400 LUTs, 554800 FFs, and 755 BRAMs (36 Kb for each) in selected SoC.

As explained in Chapter 3, the switch can be easily converted to different configurations with the help of generic parameters. The synthesis result for the switch configuration with a VOQ depth of 2048 in each input port and a RAB depth of 1024 in each output port is shown in Table 4.2. BRAM modules support two independent 18Kb blocks or a single 36Kb block of RAM [38]. 36Kb BRAMs are 72 bits wide and 512 elements deep. 18Kb BRAMs are 36 bits wide and 512 elements deep. RAMs with 36Kb are specified as 1 BRAM block, while BRAMs with 18Kb refer to 0.5 BRAM block. As detailed in Chapter 3.1.2 and Chapter 3.1.7, respectively, VOQs are 260 bits wide and RABs are 264 bits wide. According to Vivado tool synthesis reports, 1 VOQ block consumes 14.5 BRAM resources, while 1 RAB block consumes

7.5 BRAM resources. In total, $9 \times (14.5 + 7.5) = 198$ BRAMs are used for 9 input and output ports.

Table 4.2: FPGA Implementation Results of On-chip Switch

Resource	**Utilization**	**Available**
LUT	47207 (17.02%)	277400
FF	27060 (4.88%)	554800
BRAM	198 (26.23%)	755

The target operating frequency of the on-chip switch design is 156.25 MHz. For the synthesis of switch design, 156.25MHz is specified as the time constraint. The synthesizer tool tries to synthesize the design so that it can operate at this frequency. After the synthesis is completed, the design is examined by creating a timing report. However, Vivado tool does not automatically calculate a maximum operating frequency. In fact, it would be misleading to specify a term as maximum frequency. Because the synthesizer tools take into account the specified timing constraints and accordingly try to achieve the fastest design that consumes the least resources. Then, in the timing reports produced as a result of the synthesis, it is indicated whether there is a violation according to the entered time constraints. If it is desired to calculate the maximum frequency value, it can be done by considering the slowest path in the design. However, the maximum frequency value to be obtained here shows the maximum operating frequency at which the synthesized design can be run. However, this does not mean that the design cannot operate at higher frequencies. If the synthesis is repeated by specifying the time constraint for a new frequency higher than the calculated maximum frequency value, a new maximum frequency value for the design can be obtained. This is because the synthesis tool is trying to meet the new time constraint.

If it is desired to calculate the maximum operating frequency in the synthesized design, Xilinx recommends a method [39] on how to calculate the maximum operating frequency. According to this method, the maximum operating frequency is $Fmax = 1/(T - WNS)$. T in the equation is the target operating frequency, while WNS

is the worst negative slack. As a result of the calculation made according to the mentioned formula, the maximum operating frequency is calculated as $1/(6.4 - 1.95)ns = 224.71MHz$. The design is re-synthesized, specifying 224.71 MHz as the time constraint, to support the correctness of the case described above with the maximum frequency. The calculated operating frequency for the design synthesized according to the method described in [39] is $1/(4.450 - 0.090)ns = 229.35MHz$. As can be seen from the different values calculated above, the synthesis tool tries to synthesize the design according to the time constraint rule. Thus, different maximum operating frequencies are obtained for different frequency values specified. In short, a maximum frequency for designs simply indicates the maximum operating frequency of the hardware synthesized according to the specified timing constraints. Since the clock frequency is 156.25 MHz for the synthesized on-chip switch design, it is appropriate to specify the maximum operating frequency value as 224.71 MHz.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This thesis proposes a complete design verification and evaluation workflow of an on-chip packet switch to interconnect heterogeneous high-speed interfaces on a System on Chip (SoC) platform. The design particularly addresses the requirements of hardware accelerators implemented on an FPGA and served as a cloud computing service by receiving and transmitting data over high-speed Ethernet interfaces. These requirements include scalable, high-throughput interconnection, support of heterogeneous interfaces, and low latency. Furthermore, the design should be configurable in terms of the number of ports, buffer sizes, and data width to meet the dynamic demands of the cloud applications and evolving hardware platforms. The connected modules to the on-chip switch generate different types of workloads, with different arrival patterns and packet sizes. To this end, another significant requirement is a systematic verification procedure that ensures the functional correctness of the implementation.

The proposed switch in this thesis addresses these requirements by a pipelined packet switch architecture that runs at a line rate of 40 Gbps. The line rate operation provides scalability without any internal speed-up and is enabled by implementing a fabric arbiter that achieves 100% throughput. The on-chip switch has a Virtual Output Queue (VOQ) organization to prevent head of line blocking problems for network switches. Furthermore, the operation is in fixed size cycles to support the pipelined operation. To this end segmentation and reassembly of the variable sized packets are implemented with reassembly buffers at the switch outputs. The switch design is parametrized and reconfigurable.

The on-chip switch design is implemented on the XC7Z100FFG1156-2 SoC of the Xilinx Zynq-7000 family. Pipelining is utilized to increase the efficiency of the implementation. All design details and operation of the hardware components are provided in the thesis. Different than previous work in the literature we perform a systematic verification of the switch design using the SystemVerilog infrastructure. We demonstrate that the switch functions correctly, supports 100% throughput at 40 Gbps line speed and a maximum latency around 1250 nsec by making use of the statistics collected by SystemVerilog in Modelsim tool. Furthermore, the design is verified using the SystemVerilog verification environment. The scoreboard and coverage results of the verification test are displayed.

In the future work, we have both theoretical and practical studies. In the scope of the theoretical contribution, a traffic monitoring module is planned to add the on-chip switch that can dynamically adjust the buffer regions for virtual output queues and reassembly buffers. One application to benefit from this contribution would be to manage the memory of the on-chip switch efficiently. Hence, packet drops on-chip switch due to lack of enough memory can be prevented by monitoring the traffic. The second future theoretical contribution is adding a communication protocol to switch. By this protocol, the information of dropped packets on the switch is collected. Then, the block whose packet is dropped on the switch is informed about this situation. The practical extension of this work includes testing the on-chip switch implementation on FPGA in experimental hardware-accelerated cloud servers with real network traffic and accelerator implementations.

REFERENCES

[1] "Amazon ec2 instance types – amazon web services (aws)." https://aws.amazon.com/ec2/instance-types/, Amazon, Accessed: 2020-08-20.

[2] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7, IEEE Press, 2016.

[3] A. Yazar, A. Erol, and E. G. Schmidt, "Accloud (accelerated cloud): A novel fpga-accelerated cloud archictecture," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, IEEE, 2018.

[4] D. Rich, "The unique challenges of debugging design and verification code jointly in systemverilog," in *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pp. 1–7, 2013.

[5] S. Marconi, E. Conti, J. Christiansen, and P. Placidi, "Reusable systemverilog-uvm design framework with constrained stimuli modeling for high energy physics applications," in *2015 IEEE International Symposium on Systems Engineering (ISSE)*, pp. 391–397, 2015.

[6] I. Assayad, L. Eljadiri, and A. Zakari, "Systematic verification of embedded components with re-usable properties," in *2017 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pp. 1–7, 2017.

[7] H. Systems, "40Gbps Ethernet Solution." `https://hiteksys.com/pdf/40G-Ethernet-Verification-Report.pdf`. Accessed: 2021-12-29.

[8] H. J. Chao and B. Liu, *High performance switches and routers*. John Wiley & Sons, 2007.

[9] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*. Pearson, 8 ed., 2021.

[10] M. Ajmone Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, "Packetmode scheduling in input-queued cell-based switches," *IEEE/ACM Transactions*, vol. 10, no. 5, pp. 666–678, 2002.

[11] B. Hu, F. Fan, K. L. Yeung, and S. Jamin, "Highest rank first: A new class of single-iteration scheduling algorithms for input-queued switches," *IEEE Access*, vol. 6, pp. 11046–11062, 2018.

[12] Y. Lee, J. Lou, J. Luo, and X. Shen, "An efficient packet scheduling algorithm with deadline guarantees for input-queued switches," *IEEE/ACM Transactions on Networking*, vol. 15, no. 1, pp. 212–225, 2007.

[13] M. Akpinar, "Switch fabric schedulers with intelligent multi-class support: Design, implementation and evaluation on fpga," Master's thesis, Middle East Technical University, Turkey, 9 2014.

[14] J. E. Hopcroft and R. M. Karp, "An algorithm for maximum matchings in bipartite graphs," in *Soc. Ind. Appl. Math J. Computation*, vol. 2, pp. 225–231, 1973.

[15] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High-speed switch scheduling for local-area networks," *ACM Transactions on Computer Systems (TOCS)*, vol. 11, no. 4, pp. 319–352, 1993.

[16] N. McKeown, "The islip scheduling algorithm for input-queued switches," *IEEE/ACM transactions on networking*, no. 2, pp. 188–201, 1999.

[17] N. McKeown, "The islip scheduling algorithm for input-queued switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[18] J. Chao, "Saturn: a terabit packet switch using dual round robin," *IEEE Communications Magazine*, vol. 38, no. 12, pp. 78–84, 2000.

[19] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004.

[20] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[21] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[22] Intel, "What Is a GPU?." `https://www.intel.com.tr/content/www/tr/tr/products/docs/processors/what-is-a-gpu.html`. Accessed: 2021-12-10.

[23] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, IEEE, 2014.

[24] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 269–278, 2018.

[25] A. Tırlıoğlu, O. B. Demir, A. Yazar, and E. G. Schmidt, "Hardware accelerators for cloud computing: Features and implementation," in *2021 29th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2021.

[26] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet, "Lowering the latency of data processing pipelines through fpga based hardware acceleration," *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 71–85, 2019.

[27] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 430–435, IEEE, 2015.

[28] T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "Interconnection network for tightly coupled accelerators architecture," in *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pp. 79–82, IEEE, 2013.

[29] F. Yazıcı, A. S. Yıldız, A. Yazar, and E. G. Schmidt, "A novel scalable on-chip switch architecture with quality of service support for hardware accelerated

cloud data centers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pp. 1–4, IEEE, 2020.

[30] F. Yazıcı, A. S. Yıldız, A. Yazar, and E. G. Schmidt, "An on-chip switch architecture for hardware accelerated cloud computing systems," in *2020 28th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2020.

[31] F. Yazıcı, "A novel flexible on-chip switch architecture for reconfigurable hardware accelerators," Master's thesis, Middle East Technical University, 2021.

[32] M. Rashid, M. W. Anwar, and F. Azam, "Expressing embedded systems verification aspects at higher abstraction level — systemverilog in object constraint language (svocl)," in *2016 Annual IEEE Systems Conference (SysCon)*, pp. 1–7, 2016.

[33] M. W. Anwar, M. Rashid, F. Azam, and M. Kashif, "Model-based design verification for embedded systems through svocl: an ocl extension for systemverilog," *Design Automation for Embedded Systems*, vol. 21, pp. 1–36, 2017.

[34] "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018.

[35] S. R, J. S, R. A. Rahiman, R. Karthik, A. M. S, and S. S. S, "Verification of a risc processor ip core using systemverilog," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 1490–1493, 2016.

[36] A. Toe, "Design and verification of a round-robin arbiter," Master's thesis, Rochester Institute of Technology, New York, 8 2018.

[37] J. Tonfat and R. Reis, "Design and verification of a layer-2 ethernet mac classification engine for a gigabit ethernet switch," in *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pp. 146–149, 2010.

[38] Xilinx, “57304 - Vivado Timing - Where can I find the Fmax in the timing report?.” https://support.xilinx.com/s/article/57304?language=en_US. Accessed: 2021-12-28.

[39] Xilinx, “7 Series FPGAs Memory Resources.” https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf. Accessed: 2021-12-28.