

HARDWARE IMPLEMENTATION OF A HIGH-SPEED SYMMETRIC CROSSBAR SWITCH

by

Arash Haidari-Khabbaz

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE
in the School
of
Engineering Science
Communication Networks Laboratory

© Arash Haidari-Khabbaz 2000

SIMON FRASER UNIVERSITY

November 27, 2000

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Arash Haidari-Khabbaz
Degree: Bachelor of Applied Science
Title of thesis: Hardware implementation of a high-speed symmetric crossbar switch.

Prof. John Jones
Director
SFU School of Engineering Science

Examining Committee:

Chair and Academic Supervisor:

Prof. Ljiljana Trajkovic
Faculty Member, Associate Professor
SFU School of Engineering Science

Committee member:

Prof. Stephen Hardy
Faculty Member, Professor
SFU School of Engineering Science

Committee member:

Prof. Ash Parameswaran
Faculty Member, Associate Professor
SFU School of Engineering Science

Date Approved:

Abstract

Crossbar architectures for packet network switches have become very popular in the networking industry. Crossbar switches are used to route and switch data through a packet network. Their speed and performance are of significant importance. In this document, I describe the design, the methodology, and the VHDL implementation of three main blocks of an 8x8 input buffered crossbar switch: the ports, the crossbar scheduler, and the crossbar fabric. It utilizes crossbars to schedule and route data at high speed with high quality performance. The switch is being designed in the Communication Networks Laboratory (CNL) at Simon Fraser University (SFU), School of Engineering Science.

The input ports receive Asynchronous Transfer Mode (ATM) packetized data serially, queue the packet in a FIFO, determine the packet's destination output port and new VCI information, send a request to the scheduler and wait for a grant, and finally dequeue the packet towards its destination once a grant is issued. The crossbar scheduler is responsible for finding a path for a packet arriving at an input port to its destination output port. It has a priority algorithm that decides which input should be serviced. Finally, the crossbar fabric is the component that passes data from input port to the output port. At this stage of the project, output ports are simply the pins of the chip. In later stages, we plan to implement buffered output ports with queuing capabilities.

Our design was implemented in VHDL using Altera environment and simulation tools for FELX10KE Altera Field Programmable Gate Arrays (FPGA). Our simulations indicated that our design meets the specifications described in this document.

Acknowledgements

- **Professor Ljiljana Trajkovic**

Special thanks to for her advice, support and supervision throughout this project. She provided me with the opportunity to work on such a interesting and exciting project. I thank her for all her kindness.

- **Maryam Keyvani**

Without her this project would not have reached this stage. She worked on this project as my partner and a good friend. I owe this project to her technical assistance and moral support. Maryam assisted in initial research, design of state machines, coding, simulations and drawing of diagrams, and schematics Thank you very much Maryam for all your help and friendship..

- **Professor Ash Parameswaran and Professor Steve Hardy**

Thank you very much for taking on the task of becoming my thesis committee members and for your kind comments and reviews.

- **Susan Stevenson**

Thank you Susan for all your comments and guidelines for this document.

- **My family members and friends**

Your support, understanding, and friendship gave me the energy to carry on with this project. That means a lot to me and for that I will always be thankful to you.

Table of Contents

APPROVAL	i
Abstract.....	ii
Acknowledgements	iii
List of Figures.....	vii
List of Tables	ix
Chapter 1: Introduction	1
Chapter 2: High-speed symmetric crossbar switch.....	4
Chapter 3: I/O ports	7
3.1 Write sequence state machine (write_seq_sm process)	9
3.2 VCI controller state machine (VCI_controller process)	10
3.3 FIFO dequeue state machine (fifo_dq)	11
3.4 Frame pulse generator state machine (fp_out_generator).....	13
Chapter 4: Crossbar scheduler.....	14
4.1 Two dimensional ripple-carry arbiter	15
4.1.1 Priority issues	18
4.2 Rectilinear propagation arbiter (RPA)	19
4.2.1 Priority issues	20
4.2.2 Propagation Delay.....	22
4.3 Diagonal propagation arbiter (DPA).....	23
4.3.1 Priority issues	24
4.3.2 Propagation delay.....	27
4.4 Comparison between DPA and RPA	27
4.5 Implementation of the 8x8 DPA structure	27
Chapter 5: Crossbar fabric	29
Chapter 6: Integration of the switch elements	32
Concluding remarks	33
References.....	34
Appendix A: High level switch schematic.....	35
Appendix B: High level input port schematic	36

Appendix C: The port state machines.....	37
Appendix C.1: Write sequence state machine	37
Appendix C.2: VCI controller state machine.....	38
Appendix C.3: FIFO dequeue state machine	39
Appendix D: VHDL codes for the switch and its components.....	40
Appendix D.1: VHDL code for the PORT component.....	40
Appendix D.2: VHDL code for DPA structure and Arbiter cell.	52
Appendix D.2.1: VHDL code for Arbiter cell	52
Appendix D.2.2: VHDL code for DPA structure.	53
Appendix D.3: The VHDL code for the Crossbar fabric.....	68
Appendix D.4: The VHDL code for the crossbar switch.....	73
Appendix E: Switch simulation results	81

List of Figures

Figure 3.1: Switch input port. Serial data first enter the Serial shift register and are then stored in the FIFO. The Port processor communicates with Scheduler and arranges the exit of the packets through the Serial shift register.	7
Figure 4.1: Simple 4×4 arbiter.	15
Figure 4.2: Shaded cells are granted cells of Figure 4.1.	15
Figure 4.3: An arbitration cell and a possible implementation.	16
Figure 4.4: A cyclic two-dimensional ripple carry arbiter. Square i,j being marked implies that input port i is requesting to send data to output port j	18
Figure 4.5: Granted requests assuming that (2,3) is the highest priority cell.	19
Figure 4.6: Rectilinear Propagation Arbiter (RPA).	20
Figure 4.7: Modified Arbitration cell for rectilinear propagation arbiter (RPA) architecture.	21
Figure 4.8: Rectilinear propagation arbiter (RPA) architecture. Highest priority cell is (2,3).	22
Figure 4.9: Rectilinear propagation arbiter (RPA) architecture. Highest priority cell is (1,1).	22
Figure 4.10: Fixed priority diagonal propagation arbiter (DPA). Double squares indicate requests.	24
Figure 4.11: Fixed priority diagonal propagation arbiter (DPA). Double shaded squares indicate grants.	24
Figure 4.12: Diagonal Propagation Arbiter (DPA).	25
Figure 4.13: Diagonal propagation arbiter (DPA). Highest priority diagonal row rooted at (1,1).	26
Figure 4.14: Diagonal propagation arbiter (DPA). Highest priority diagonal row rooted at (2,3).	26
Figure 5.1: Crossbar fabric DEMUX for the packet switch.	29

Figure 5.2: The circuit for producing output line 1 in the crossbar fabric. The dots correspond to AND gates between the input line and the *ctrl* line corresponding to that point. 30

List of Tables

Table 3.1: Areas and delays for each arbiter design.....	27
--	----

Chapter 1: Introduction

Amount of traffic on the Internet is continuously growing at a high rate. Everyday we hear of new businesses on the Internet, new services that are offered online, and new ways that the Internet can be utilized. High-speed and high-performance networking is one of the hottest topics in the era of information technology. Anywhere we look, companies, scientists, researchers, and hardware and software engineers are developing new architectures, tools, and services in this area. Although the world is using numerous resources to improve the performance of local and wide area networks, they do not seem to be sufficient. New real-time applications, such as audio and video transmissions, video conferencing, video on demand, distant education, e-commerce, etc., all demand higher rates in information transmissions and in minimization of delay. As a result, there is a tremendous amount of interest in building faster and more efficient communication networks.

Data passes through communication networks, such as the Internet, in the form of packets. These packets of information have a source and a destination, and travel in the network through intermediate nodes. Many devices, such as switches, routers, and bridges, are required to so that the data can pass through a network. Switches are devices with some intelligence that are placed at intermediate nodes of communication networks. They help route the packets through the network and towards their destination node. Small $N \times N$ packet switches are the key components of interconnection networks used in multiprocessors, multicomputers, and integrated communication networks for data, voice, image, and video. Crossbar architectures are frequently used as the internal fabric of high performance network switches and routers. With the rapid growth of the Internet, and the new applications arising, interest has increased in high-speed networks supporting Internet Protocol (IP) or

Asynchronous Transfer Mode (ATM). Consequently, the design of high performance $N \times N$ space division packet switches is of critical importance to the success of multiprocessors and multicomputer systems.

Crossbar switches are of special interest in packet switch designs. A crossbar switch provides fast and efficient data transmission, which makes it a favored industrial choice. The industry's interest in crossbar switches encourages researchers to try to both optimize the performance and economize the design and fabrication of crossbar switches. The many companies working on technologies related to network switches, include Newbridge, Cisco, PMC-Sierra, Agilent, and Marconi. Last year, PMC-Sierra acquired Abrizio, a US based company that created Tiny Tera [1] that is one of the fastest network switches [12].

In an attempt to join this fast growing industry, we at SFU's School of Engineering Science Communication Networks Laboratory (CNL) [10] developed and implemented a high-speed crossbar packet network switch. Our switch has 8 input and 8 output ports, each being able to operate at the 5 Mbps rate. This input buffered switch can service ATM [11] packets. The switch consists of three major blocks: ports, a crossbar scheduler, and a crossbar fabric. The packets first enter the switch through the input ports, where they are queued. The scheduler maps the input ports to the output ports. The data in the mapped input ports are then sent through the crossbar fabric to their destination output ports. Simulation, implementation, and testing were performed using ALTERA MAX+PLUS II simulation software for FLEX10KE devices [7].

In Chapter 2, I discuss the basic theory behind the crossbar switch and fundamentals of its functionality. In Chapter 3, the I/O port architecture including its state machines are discussed in detail. Chapter 4 discusses the two designs of a crossbar scheduler and their priority issues. It also includes the discussion of the scheduler architecture that has been used in our switch. Chapter 5 describes the implementation of the crossbar fabric. Finally, Chapter 6 discusses the integration of the entire switch. A high level schematic of the switch is given in appendix A. Appendix B contains a high level schematic of an input port. The state machine diagram for the input port are drawn in Appendix C. Appendices D and E contain the VHDL code and the simulation results of the switch, respectively.

Chapter 2: High-speed symmetric crossbar switch

We have developed a VHDL implementation of a crossbar switch for use in high-speed packet networks. The switch consists of three major blocks: ports, a crossbar scheduler, and a crossbar fabric. Our switch has 8 input and 8 output ports, each being able to operate at the 5 Mbits/s rate. This input buffered switch can service ATM packets. Simulation, implementation, and testing are performed using ALTERA FPGA (FLEX10KE) evaluation boards, and ALTERA MAX+PLUS II simulation software [7].

The overall functionality of the switch can be described as follows: the packets first enter the input ports of the switch where they are queued based on their order of arrival. Each port has a controller that determines the destination of a packet based on the packet header using a programmable mapper. The port processor then sends a request to the scheduler for the destination output port. The scheduler grants a request based on a priority algorithm that ensures fair service to all the input ports. Once a grant is issued, the crossbar fabric is configured to map the granted input ports to their destination output ports. In the next chapters we discuss the architecture of these three main blocks in more detail.

Our switch design can be scaled down. 8x8 is its largest size. We can easily scale the switch down to any number of input ports (e.g., 2x2, 3x3, 5x5, 4x5, 6x3, ...). Even symmetry is not required. With some additional effort it can also be scaled up.

We have defined the input data format to be the following: there are three input lines to each switch input port. The first one is the input data line that carries the ATM packets serially. The second line is the clock signal. We assumed an external unit before the switch port that extracts or generates the clock for the input data. This clock is global to the switch components; it is used to input data to the switch and to output data from the switch. The input ports sample the data on the falling edge of the global clock. Thus, the data is assumed to be available at the input lines on the rising edge of the global clock. At the output ports, the switch outputs data on the rising edge of the universal clock. The switch has a clock output as well, which can be used as a reference for the neighboring components. The third input line to each switch port is the frame pulse. A frame pulse is a pulse of length 1 bit that indicates the start of each packet. We have assumed that a framer unit that generates a frame pulse at the start of each packet exists before each switch port. While frame pulse is high, the data is ignored. Thus, we expect the packet to begin after the falling edge of the frame pulse.

At the output of each input port, when the data is being sent to its destination output port, the frame pulse is regenerated and it is sent to the output port along with the data. The output ports in our switch do not have any processing capability or any storage capacity. They are simply the pins of the chip. Future plans for this project include the design of buffered output ports with some processing capabilities. However, since we currently lack these features, many lines are carried all the way to the output ports so that the processing units of the neighboring components can process and store the outgoing data. The output lines of our switch are *data_valid* and *input_port_number*.

- *data_valid*: This line indicates that the data present at the output of the switch is valid for sampling. If this line is logic low, the output lines are invalid and should be ignored. This option exists because the grant for an output port may change before a packet is completely dequeued from the input port. As a result the input port will stop transmitting data towards its destination output port. Thus, the value on the output port is no longer valid. The invalidity of data is indicated by asserting *data_valid* low.
- *input_port_number*: This information is available at each output port along with the data. This output line indicates at which input port the data originated. Assume that input ports 1 and 5 are requesting to transmit data to output port 3, and that a grant for output port number 3 is issued to input port 1. Thus, input port 1 starts transmitting packets to output port 3. Now assume that after transmitting 10 bytes, the scheduler changes state (this can happen when the scheduler clock does not have a period of one packet time). The priority sequence is changed and as a result input port 5 now has priority to send data to output port 3. Thus, the grant is now given to input port 5 and it has started to transmit to output port 3. The output port has no way of knowing that it is now receiving data from a different input port. Thus, the outgoing data will be corrupted and useless. Therefore, to avoid this confusion, the originating input port number is sent along with data to the output port. Now, the processing unit at the output port can arrange the data bytes into packets accurately.

Chapter 3: I/O ports

A switch has both input and output ports. In this project, only input ports are implemented with buffers and port controller, and output ports are simply metal pins of the chip. Each input port receives data serially. It queues the received packets into buffers and then, once a grant is issued, transmits them out to their destination output ports. The ports are designed to handle ATM packets. The architecture of an input port is shown in Figure 3.1. A detailed schematic of the port is shown in Appendix B.

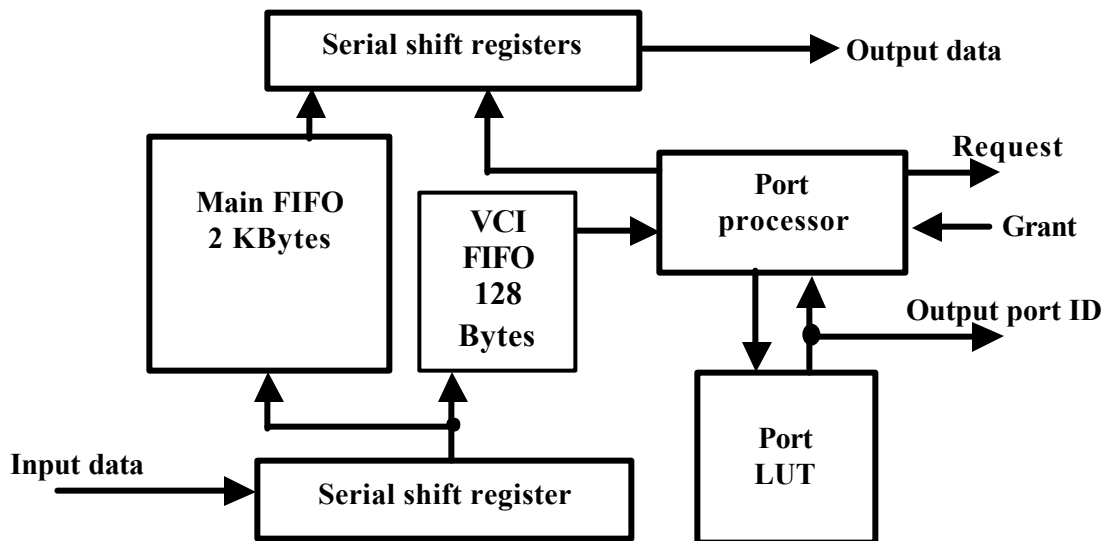


Figure 3.1: Switch input port. Serial data first enter the Serial shift register and are then stored in the FIFO. The Port processor communicates with Scheduler and arranges the exit of the packets through the Serial shift register.

The serial data is first shifted bit by bit into a Serial shift register, shown in Figure 3.1. When the shift register accumulates eight bits (one byte), the byte is sent to the First In First Out (FIFO) queue, and the next byte starts to be shifted into the shift register. At the same time, the processor extracts the address information from the header of the first

packet in the queue and sends it to a Look Up Table (LUT). The LUT updates the VCI bytes of the header. The LUT also returns the output port destination number for that packet. The processor then sends a request for that specific output port to the crossbar scheduler and awaits a grant. Once a grant is issued, the first data byte at the beginning of the queue is dequeued into a shift register and is serially shifted out to the destination output port. The remaining bytes of the packet are sent to the output port in a similar manner. After the entire packet is sent, the same process is repeated for the next packet. Note that as soon as a grant for an output port is issued, the port processor sends the output port ID through the crossbar fabric so that the data leaving the input port can be routed to its destination output port.

While a packet at the input port awaits its grant, all the other packets entering that port remain in the queue. In other words, no packet will be transmitted out of the port until all packets that previously arrived have been transmitted. This problem is known as head of line (HOL) blocking [9]. Virtual Output Queuing (VOQ) has been introduced as a way of eliminating HOL blocking [9]. This feature has been left for future implementation. Various designs of switch ports are discussed in [1] and [8].

Currently, our output ports are simply the metal pins of a chip; however, the next step of the project is to implement output ports with buffers that queue data before exiting the switch.

The input port architecture can be divided into four main sections:

1. write sequence state machine
2. FIFO dequeue state machine
3. VCI controller state machine
4. frame pulse generator state machines.

In the following subsections I discuss the detailed algorithms of each state machine. The VHDL code for the port is attached in Appendix D.1

3.1 Write sequence state machine (write_seq_sm process)

One of the two FIFO queues used in each port is the main FIFO is 2KB deep and it is used to store all packets that arrive at the switch. It will do so until it is full, at which point it will set its full flag, indicating that a FIFO overrun has occurred. In case of a FIFO overflow, all the incoming bytes will simply be discarded until some FIFO space is freed by the FIFO dequeue state machine.

The second FIFO is the VCI FIFO and it is 128 bytes deep. This FIFO is used to store the VCI bytes (bytes 2,3 and 4) of each incoming packet. This FIFO will not run out of space before the main FIFO. The VCI FIFO can contain the VCI information for of 42 packets while the main FIFO can only contain 38 packets before it becomes full. Thus, when the main FIFO becomes full, there will be no more incoming packets so the VCI FIFO is never filled up completely. The write sequence state machine is aware of start of the packet and also keeps count of the number of bytes of the packet that have been

enqueued. As a result, this state machine can locate the VCI bytes of the packet and load them into the VCI FIFO.

This state machine is responsible for the following tasks while not in RESET and while the main FIFO is not full:

1. Look for a frame pulse as a starting point of the packet.
2. Accumulate 8 bits into the serial shift register.
3. Load 8 accumulated bits in step two into the port's main FIFO.
4. Load bytes 2, 3, and 4 of the packet into the VCI FIFO as well.
5. Repeat Steps 2 to 4 until 53 bytes (i.e., an ATM packet) is loaded into port FIFO.
6. Go to Step 1.

A copy of the write sequence state machine is given in Appendix C.1.

3.2 VCI controller state machine (VCI_controller process)

The VCI controller has to communicate with the LUT to obtain the new VCI information that has to be placed in the packet. The LUT also returns the destination output port for the packet that is used to issue a request. The LUT is simply a table that takes a 16 bit input (VCI data extracted from the packet) and maps it into a four bit (packet destination output port number) and a 16 bit (new VCI information) binary numbers. These new data are kept in a few predefined registers and are later used by the FIFO dequeue state machine. The VCI controller monitors the packet counter to ensure that the entire packet is dequeued before it processes the next packet in the queue.

The VCI controller state machine is responsible for carrying out the following tasks:

1. Wait until the FIFO has more than one packet, then go to Step 2. If there are no packets waiting to be sent out, the output port number “0000” will be chosen indicating that no further action will be taken by the scheduler.
2. Dequeue the first three bytes of the VCI FIFO and extract the 16 bit VCI information from them.
3. Send the 16 bit VCI information to the LUT and wait for the new VCI information bits and the destination port number.
4. While the packet is being dequeued towards its destination output port, replace the old VCI information in the packet with the new VCI information obtained from the LUT.
5. Wait until the entire packet is dequeued from the main port FIFO and then go to Step 1.

A copy of the VCI controller state machine is given in Appendix C.2.

3.3 FIFO dequeue state machine (fifo_dq)

The FIFO dequeue state machine has to dequeue a packet once the scheduler issues a grant for the packet’s destination port. As long as the grant vector remains unchanged the bytes of the packet will continue to be dequeued one after the other. If the grant vector is changed before the entire packet is dequeued, the state machine will wait until the grant vector changes back, and will then dequeue the remaining bytes. This feature was added

so that the scheduler's clock, which changes the priority vector and, hence, the grant vector, is independent from the FIFO dequeue clock.

Another feature added to our design is the usage of a dummy byte after a reset or power up. This dummy byte exists because the Altera FIFO component does not function properly when it is empty. When the switch is powered up and is running, all the state machines ensure that at all times there are at least one packet in the FIFO. But, after a reset or a power up the FIFO is completely empty and thus it is filled with a dummy byte. When the first packet is being dequeued, the dummy byte 00h is attached to the beginning of the packet. Thus, the receiving processors can easily detect this dummy byte and discard it. This dummy byte is for internal switch use only. Once output ports are implemented, this byte is discarded before packets exit the switch.

The FIFO dequeue state machine is responsible for carrying out the following tasks:

1. Ensure that there is more than one packet in the main FIFO before going to Step 2.
2. Send a request to the scheduler asking to send data to the port number given by the packet destination port register.
3. Wait until a grant is issued.
4. Start dequeuing packet bytes. While dequeuing, replace the VCI bytes with the new VCI information, prepared by the VCI controller.
5. Dequeue packets for as long as the grant vector and the request vector are identical. If they differ halt the dequeuing process until they become identical.
6. If the entire packet is dequeued go to Step 1.

A copy of the FIFO dequeue state machine is given in Appendix C.3.

3.4 Frame pulse generator state machine (fp_out_generator)

The frame pulse generator has only one task. It generates a frame pulse before the first bit of every packet. Similar to the format of the input data line, the bit that is present on the data output line when the frame pulse is generated is invalid. The frame pulse is routed to the output ports like the data. Again, this frame pulse is for internal use only. Like the dummy byte, once the output ports are implemented, the switch will have no output frame pulses.

Chapter 4: Crossbar scheduler

Various algorithms exist for scheduling packets in a packet switch. Crossbar schedulers, being one of them, have recently become more popular since they are fast, and easy to implement. A scheduler has a very simple function: it accepts one request from each input port and grants some of those requests, during each packet time according to a priority algorithm. Its main goal is to be fair to all the inputs.

Among many different ways of implementing a crossbar scheduler, the Rectilinear Propagation Arbiter (RPA) and the Diagonal Propagation Arbiter (DPA) have been designed and implemented by a group of researchers at UC San Diego [6]. This design has a round robin priority scheme and is easy to implement in hardware. Although our switch utilizes the DPA structure, it is difficult to understand how DPA architecture works without knowing how the RPA structure functions. Thus, in this section I present the design and implementation of RPA and DPA architectures. These designs are intended for use in a high performance input-queued crossbar switch. This centralized scheduler considers requests from all the input queues and finds the best realizable configuration. In RPA and DPA designs, the scheduling decision is implemented entirely in combinational logic using two-dimensional ripple carry arbiters. Although we have designed and implemented an 8x8 switch, for simplicity we describe the architectures of a 4x4 RPA and DPA. In our switch we simply extended the structures to 8x8.

4.1 Two dimensional ripple-carry arbiter

A 4×4 two-dimensional ripple-carry arbiter is shown in Figures 4.1 and 4.2. Here, rows correspond to input ports of the switch, and the columns correspond to output ports of the switch. The arbiter is built from a number of smaller cells. These are called arbiter cells. A sample arbiter cell with its internal combinational logic is shown in figure 4.3. The pairs of numbers written on each cell, specify the requests that are handled by that specific cell i.e., the pair i,j on a cell shows that the cell is responsible for handling packets that are destined to go from input port i to output port j .

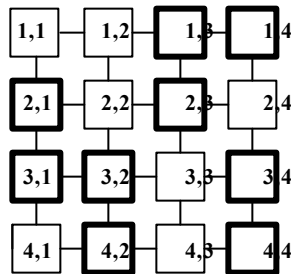


Figure 4.1: Simple 4×4 arbiter.

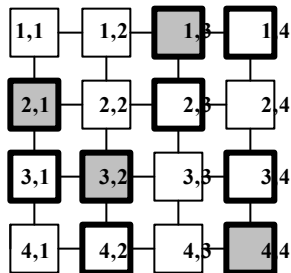


Figure 4.2: Shaded cells are granted cells of Figure 4.1.

Signal R (*Request*) which is an input to every i,j arbiter cell, is active when these two conditions are met:

- 1) There is a packet at the head of input port i 's buffer.

2) This head of line packet is destined for output port j .

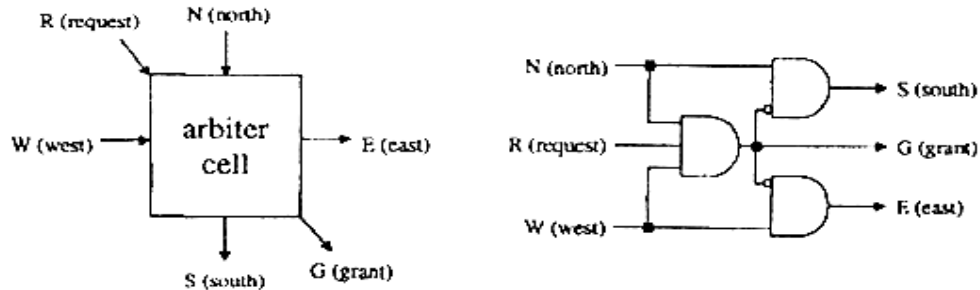


Figure 4.3: An arbitration cell and a possible implementation.

Signal G (*Grant*) which is an output to every i,j arbiter cell is active when the request from input port i to output port j has been granted.

Since each input can be sending (and each output can be receiving) only one packet at a time, there should not be two or more granted requests in each row (and each column). Having, for instance, two requests granted in the same column at a time, means that the output port corresponding to that column is receiving two packets at that time, which is not acceptable. To ensure that this problem never happens, signals N (*North*), S (*South*), W (*West*), and E (*East*) are introduced (Figure 4.3). These signals in each cell have the duty of relaying to the next cell or receiving from the former cell whether a request has been granted. The algorithm is as follows:

1. Start from the top left most cell (i.e., 1,1).

2. Once a cell is reached, move to its right and bottom cells (provided that they exist).
3. For each arbiter cell, the G (*Grant*) signal is activated if and only if the R (*Request*) signal is active and there have not been any requests granted in the cells at the top and to the right of the current cell we are looking at.
4. If a request is granted, activate (active low) the E (*East*) and S (*South*) signals so that E and S signals inform the cells on the right and at the bottom.

This algorithm is implemented in the simple combinational logic of an arbiter cell using simple AND gates (figure 4.3). The reader should note the following:

- In the two dimensional ripple carry arbiter of Figures 4.1 and 4.2, the S signal of each arbiter cell is attached to the N signal of the bottom arbiter cell, and the E signal of each arbiter cell is attached to the W signal of the arbiter cell on its right.
- The W signal of cells in the first column and the N signal of cells in the first row are always set to logic one.
- The S signal of the cells in the last row and the E signal of the cells in the last column are “don’t care”s (i.e., floating).

In Figure 4.1 double squares indicate that the corresponding cell has been requested.

In Figure 4.2 double shaded squares show that the corresponding square has been given a grant. Assuming that each arbiter cell has a delay D , then the time needed for realization of any permutation is $(2n-1)D$ for any $n \times n$ arbiter. It can be seen that when

there exists two or more requests in the same row or column, only one of them (the one higher or on the left) is granted.

4.1.1 Priority issues

The described design gives priority to the cells that are higher and to the left. It specially gives the highest priority to the cell number (1,1). This, of course, is not fair. We would like to be able to rotate the priority, so that every cell has the chance of being the highest priority cell. That is why the cyclic two dimensional ripple carry arbiter of Figures 4.4 and 4.5 is made. Here, we have a two dimensional taurus and because of the symmetry, each cell can be the top left most corner cell, and, hence, can have the highest priority [6]. For example if the current highest priority cell is (2,3) and the requests are the double squares in Figure 4.4 (which are the same as Figure 4.1), then the double shaded squares of figure 4.5 are the selected cells. Comparing Figures 4.2 and 4.5, we can see how the selected cells differ when the highest priority cell changes.

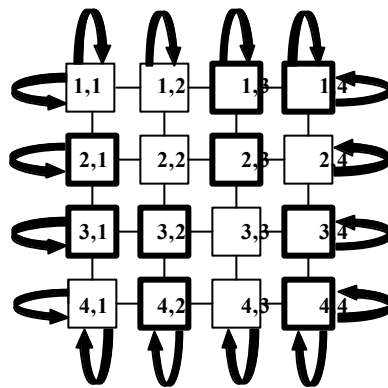


Figure 4.4: A cyclic two-dimensional ripple carry arbiter. Square i,j being marked implies that input port i is requesting to send data to output port j .

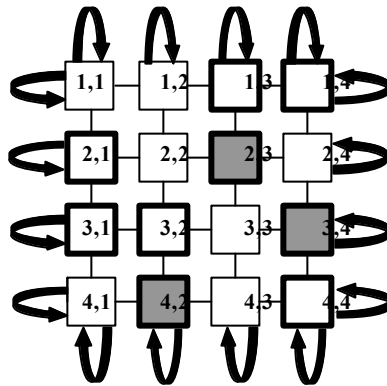


Figure 4.5: Granted requests assuming that (2,3) is the highest priority cell.

This architecture is fair but it has a fundamental problem: It has a “combinational feedback loop”. Such architectures are difficult to design and test, they are not very well supported by the logic synthesis tools and they have to be carefully simulated at the physical layout level. Furthermore, designs with combinational feedback may not be optimized well. In addition, most timing analysis tools cannot work with combinational loops unless loops are manually broken, which may introduce unacceptable inaccuracies in the time reported. These problems convinced us to not use a combinational feedback loop. The design proposed to replace this is the RPA architecture described in section 4.2.

4.2 Rectilinear propagation arbiter (RPA)

To overcome the problems accompanying the cyclic feedback architecture and to be able to, at the same time, rotate the priorities, a solution was proposed in [6]. For an $n \times n$ arbiter we repeat the first $(n-1)$ rows and the first $(n-1)$ columns as is shown in Figure 4.6 for a 4×4 arbiter. This will result in a $(2n-1) \times (2n-1)$ crossbar and again, the W and the N signals are assigned to 1 for the first column and the first row, respectively. This

architecture removes the need for a cyclic feedback. At every time slot only n^2 cells are considered to be active. Active cells are the $n \times n$ bold square shown in Figures 4.8 and 4.9. The general rule is that, the cell that sits at the top left most corner of the bold box has the highest priority. This thick box, therefore, moves one step to the right at every time slot, to rotate this priority. When the top left most corner cell is (n,n) , the bold box has traveled all the way through the arbiter and, therefore, goes back to its starting position shown in Figure 4.8. The box moves to the next row when it reaches the last cell in a row.

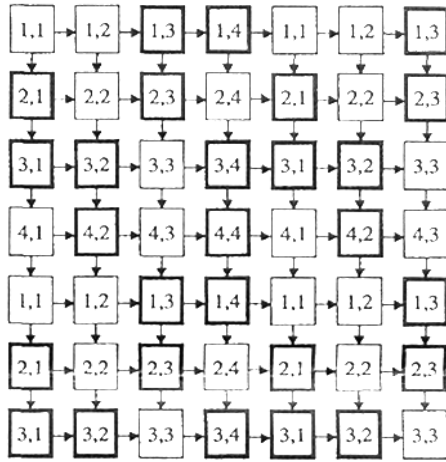


Figure 4.6: Rectilinear Propagation Arbiter (RPA).

4.2.1 Priority issues

To implement priority rotations, two vectors P and Q are introduced. The $(2n-1)$ elements of vector P correspond to the $(2n-1)$ rows of the arbiter, and the $(2n-1)$ elements of vector Q correspond to its columns. When the i^{th} element of this vector is 1, the i^{th} row of the arbiter is active (marked by the bold box) and when the j^{th} element of vector Q is 1, then the

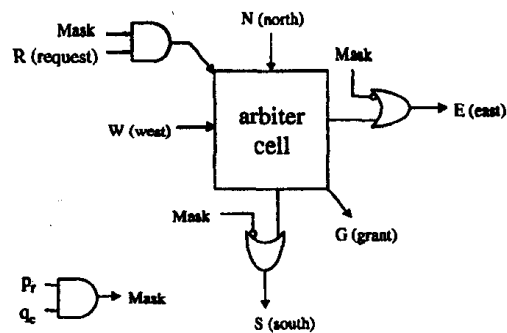


Figure 4.7: Modified Arbitration cell for rectilinear propagation arbiter (RPA) architecture.

j^{th} column of the arbiter is active (marked by the bold box). The algorithm for priority rotations:

- 1) set $P = "1111000"$ and $Q = "1111000"$
- 2) if $P = "0001111"$ then
 - {
 - if $Q = "0001111"$ then go to 1.
 - else set $P = "1111000"$ and rotate Q one position to the right.
 - }
- else rotate P one position to the right.

As it can be seen in Figure 4.7 the arbiter cell of the RPA is somewhat different from that of the basic arbiter introduced earlier. The difference is a signal called "*Mask*" that indicates whether the arbiter cell is in the active zone or not. The additional gates ensure that every request only takes effect. If $Mask = 1$ and if $Mask$ is not 1, then there are no *Grants* given in that cell, and, therefore, E and S signals are forced to 1. The corresponding P and Q elements of the cells in the bold box are 1. Hence, it seems quite

logical to let $Mask$ be the output of an AND gate whose inputs are P and Q elements (p_r, q_r) that correspond to that specific cell.

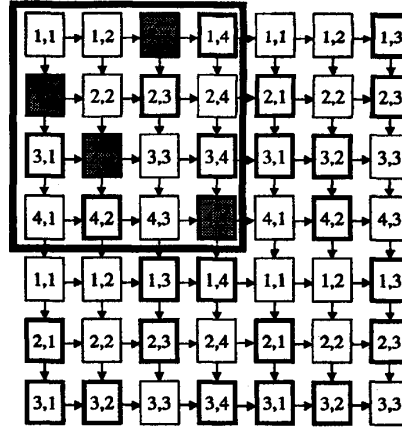


Figure 4.8: Rectilinear propagation arbiter (RPA) architecture. Highest priority cell is (2,3).

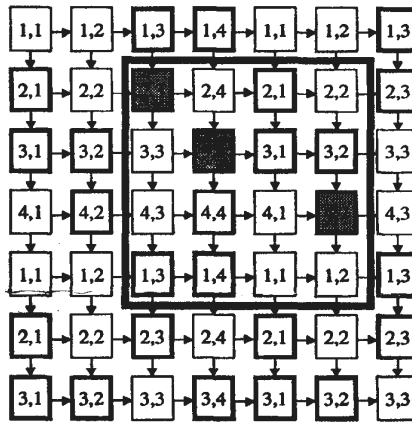


Figure 4.9: Rectilinear propagation arbiter (RPA) architecture. Highest priority cell is (1,1).

4.2.2 Propagation Delay

It can be shown that propagation delay of a $(2n-1) \times (2n-1)$ RPA is $(2n-1)D + 2t$ where D is the delay of one cell, and $2t$ is the delay of two additional gates added for $Mask$ and $R(Request)$ signals. This delay is similar to the delay of the simple propagation arbiter

because the $2t$ term is small and negligible. The reason is that the *Mask* signal forces the *E* and *S* signals to 0 value when the cell is not active. This implies that for any given permutation, *Requests* need not propagate through the whole $(2n-1)(2n-1)$ arbiter cells, but only through the $n \times n$ cells in the active window.

4.3 Diagonal propagation arbiter (DPA)

A modified version of the two dimensional arbiter was introduced in [6]. We have implemented this arbiter for our switch. The key to this design is that there are independent cells and granting one of them does not prevent granting the others. We place the cells that are independent of each other in diagonal rows, as shown in Figure 4.10. It can be seen in this figure that the cells (1,1), (4,2), (3,3), and (2,4) are independent of each other. So are the cells (2,1), (1,2), (4,3), and (3,4). The arbitration process begins with first considering the first diagonal row. If there are requests for any cell in the first diagonal row, they can all be granted. Then, in the next time slot, the arbitration process moves to the next diagonal row. Now, the conditions for granting a request would be the same as the one of the diagonal in the previous timeslot. If the requested cells are the bold squares in Figure 4.10 then the selected cells will be the shaded squares in Figure 4.11. The arbitration delay for such a design will be nD for an $n \times n$ arbiter, assuming that D is the propagation delay of a single cell. This is roughly half of the delay of the RPA architecture described in section 4.2.

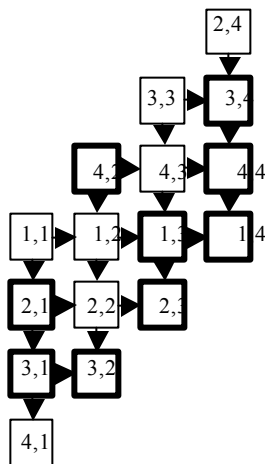


Figure 4.10: Fixed priority diagonal propagation arbiter (DPA). Double squares indicate requests.

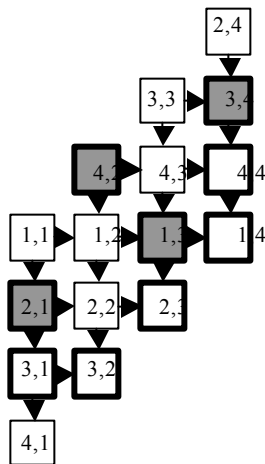


Figure 4.11: Fixed priority diagonal propagation arbiter (DPA). Double shaded squares indicate grants.

4.3.1 Priority issues

As before, we need a way to ensure fairness in our design. It is obvious that this design always gives the highest priority to the cells in the first diagonal row. To overcome this

problem, the first $(n-1)$ diagonal rows of an $n \times n$ arbiter are repeated after the last row. Such architecture is shown in Figure 4.12 for a 4×4 arbiter. The arbitration cell looks like the one in Figure 4.7, except that the intermediate signal *Mask* is simply the priority signal. The priority rotation starts from the first row. In this case $P = "1111000"$. In the next time slot, the highest priority cells are the ones in the second row, with $P = "0111100"$. This rotation of the P vector continues until $P = "0001111"$. It then resets to its starting value $"1111000"$ shown in Figure 4.13. Here, again, the active region of the arbiter is the one inside the thick line. Assuming the double squares are the requested cells, Figures 4.13 and 4.14 show the realized permutation under two distinct priority scenarios.

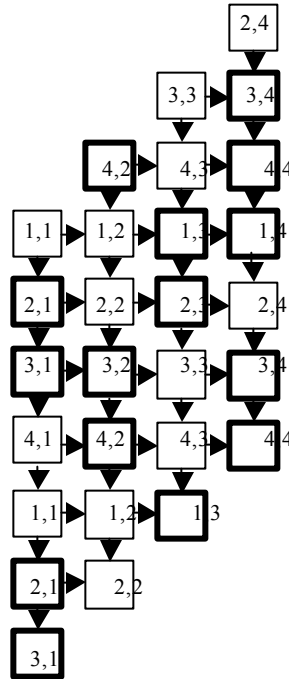


Figure 4.12: Diagonal Propagation Arbiter (DPA).

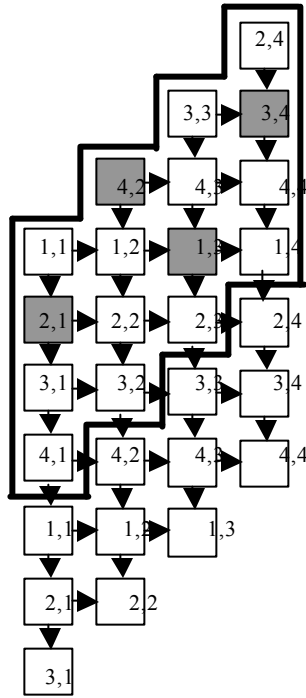


Figure 4.13: Diagonal propagation arbiter (DPA). Highest priority diagonal row rooted at (1,1).

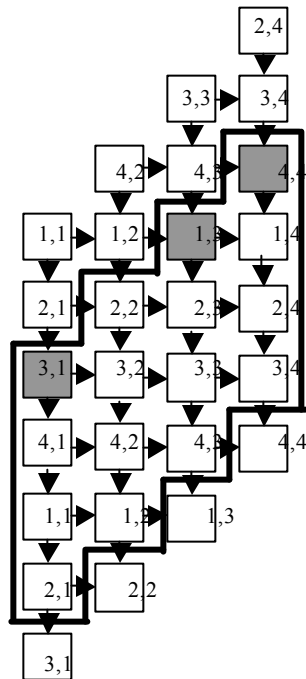


Figure 4.14: Diagonal propagation arbiter (DPA). Highest priority diagonal row rooted at (2,3).

4.3.2 Propagation delay

The critical delay for an $n \times n$ crossbar with DPA architecture is $(nD + t)$, because there are n rows in each active thick box and the propagation delay of each cell is D . The t term is the gate delay of the OR gate with *Mask* signal and *Request* signal as inputs. Note the omitting of $2t$ term present in RPA because a separate gate is not needed to produce the *Mask* signal.

4.4 Comparison between DPA and RPA

Table 4.1 [6] describes the results from timing simulations of DPA and RPA architectures for 16 port and 32 port crossbar switches. The results show that for a DPA architecture the number of cells (and therefore the cost) is almost half of that of RPA architecture. Also, the propagation delay of the DPA design is almost half of the delay in RPA.

	RPA (16)	RPA (32)	DPA (16)	DPA (32)
# Cells	5,766	23,814	2,480	10,080
Area (mm ²)	0.497	2.052	0.184	0.746
Delay (ns)	10.23	21.10	6.15	13.61

Table 4.1. Areas and delays for each arbiter design.

4.5 Implementation of the 8x8 DPA structure

As mentioned earlier, we have chosen to implement an 8x8 DPA structure for our switch. This scheduler is implemented and simulated using VHDL and ALTERA Max Plus II

simulation tool. The VHDL code for DPA and a single Arbiter cell are given in Appendix D.2.

The scheduler requires an input clock signal. The P vector, described in section 4.3, changes state on each rising edge of this clock. The scheduler is also implemented with a asynchronous reset signal. The reset is active low, implying that the scheduler stops working as soon as it goes low. It then disregards all the requests and will set all the grants to logic zero. Following the removal of reset signal, with the first rising edge of the clock, the scheduler will start with the highest priority arbiter cell being (1,1).

Due to memory limitations and the time consuming process of compilation, we were not able to implement the DPA structure with higher input/output ports. For the same reasons we were not able to implement an RPA structure for comparison.

Chapter 5: Crossbar fabric

The crossbar fabric is responsible for physically connecting an input port to its destination output port based on the grant issued by the scheduler. The design of the crossbar fabric is shown in Figure 5.1.

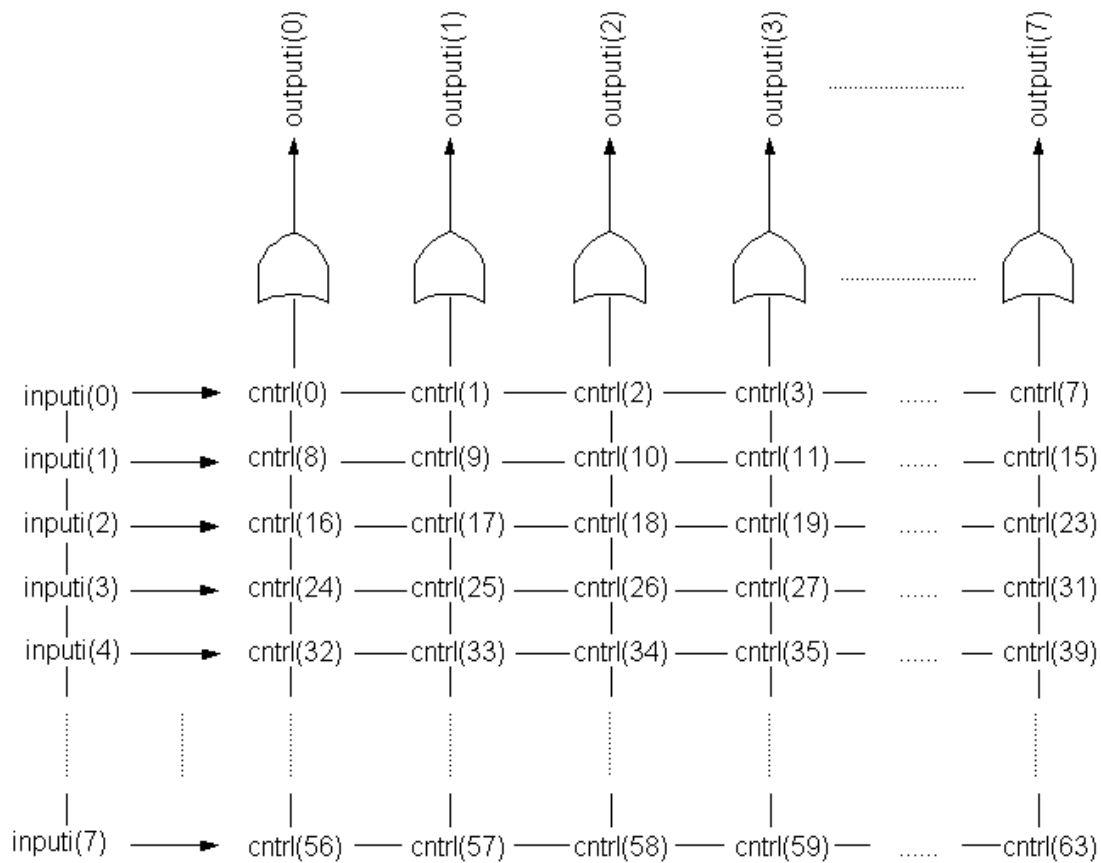


Figure 5.1: Crossbar fabric DEMUX for the packet switch.

Each bit of the output is constructed from inputs 0 to 7 AND'ed with the *cntrl* lines in the column corresponding to the output, and OR'ed at the end. For example:

$$\text{output}_i(2) = [(\text{input}_i(0) \text{ AND } \text{cntrl}(2)) \text{ OR } (\text{input}_i(1) \text{ AND } \text{cntrl}(10)) \text{ OR } (\text{input}_i(2) \text{ AND } \text{cntrl}(18)) \text{ OR } (\text{input}_i(3) \text{ AND } \text{cntrl}(26)) \text{ OR } (\text{input}_i(4) \text{ AND } \text{cntrl}(34)) \text{ OR } (\text{input}_i(5) \text{ AND } \text{cntrl}(42)) \text{ OR } (\text{input}_i(6) \text{ AND } \text{cntrl}(50)) \text{ OR } (\text{input}_i(7) \text{ AND } \text{cntrl}(58))]$$

The grant comes from the crossbar scheduler and it is the same as the one given to the ports. It determines the output port to which input data is routed. A detailed look at the circuit for one output line is shown in Figure 5.2.

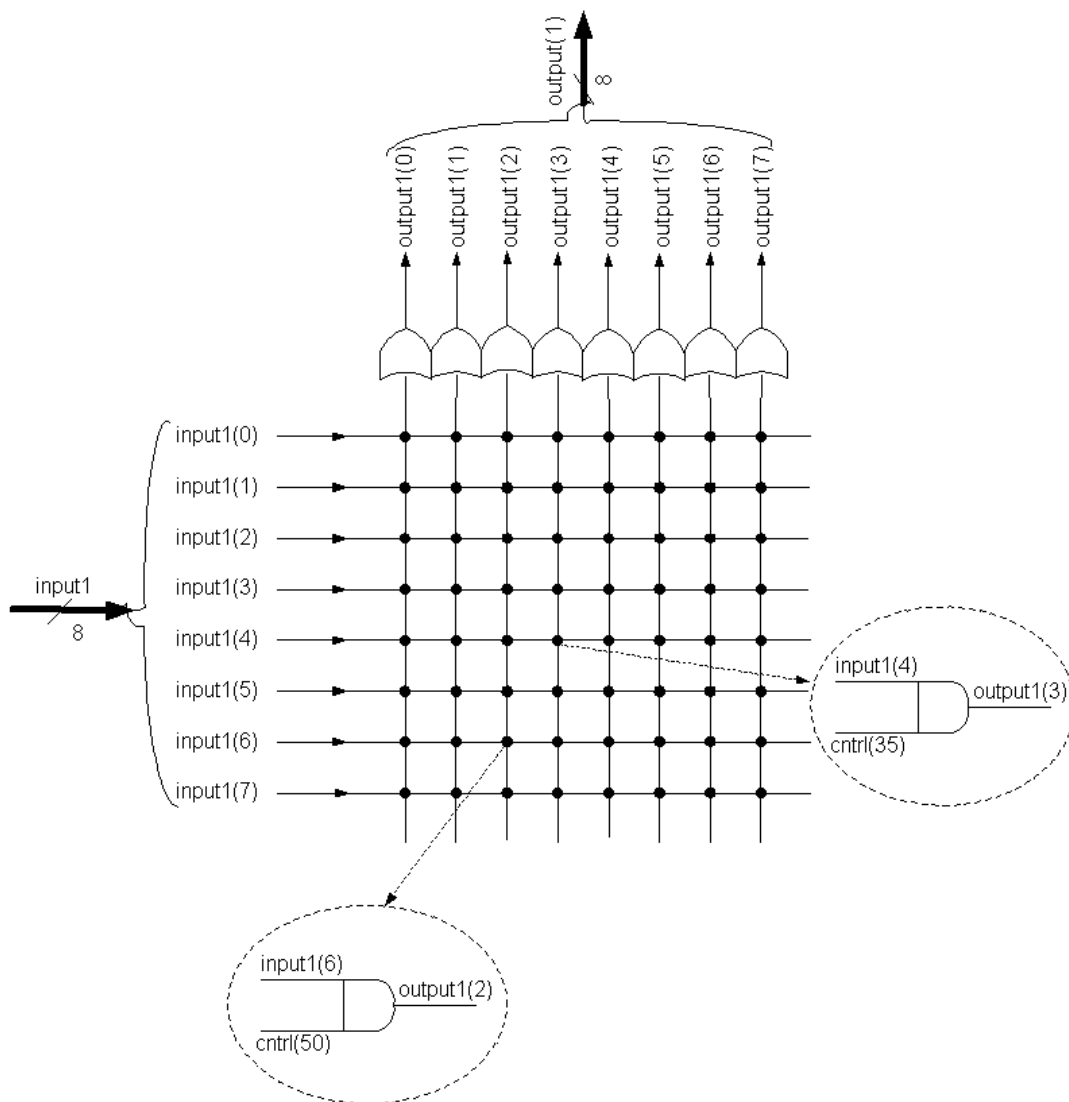


Figure 5.2: The circuit for producing output line 1 in the crossbar fabric. The dots correspond to AND gates between the input line and the *cntrl* line corresponding to that point.

As mentioned in Chapter 2, in addition to data, the four bit input port number and the frame pulse are also routed to the output ports. Thus, when the fabric connects an input port to an output ports, six bits in parallel are transmitted. As a result, we have used six demuxes to route the input ports to the output ports. The structure of the fabric is rather simple. We have used only AND and OR gates to implement the fabric. The fabric passes the data through very quickly. The maximum propagation delay is two gate delays. The VHDL code for the cross bar fabric is given in Appendix D.3.

Chapter 6: Integration of the switch elements

The integration of the switch is carried out in the usual way, in the Altera environment. All the components, i.e., input ports, crossbar scheduler, and crossbar fabric, were built standalone and then hardwired in VHDL to form the actual body of the switch. A high level schematic of the switch is given in Appendix A and its VHDL code is given in Appendix D.4. The integrated switch has been simulated and has proven to be functional. Due to slow the compilation processes of Altera tools, we were unable to simulate the switch prolonged time periods; however, we were able to simulate it long enough (300 μ sec) to verify that it functions according to the specification described in this document. Our original expectations were to achieve switches at higher rates, but we were bounded by the Altera FLEX10KE FPGA timing and speed limitations. The simulation results of the switch are given in Appendix E.

Concluding remarks

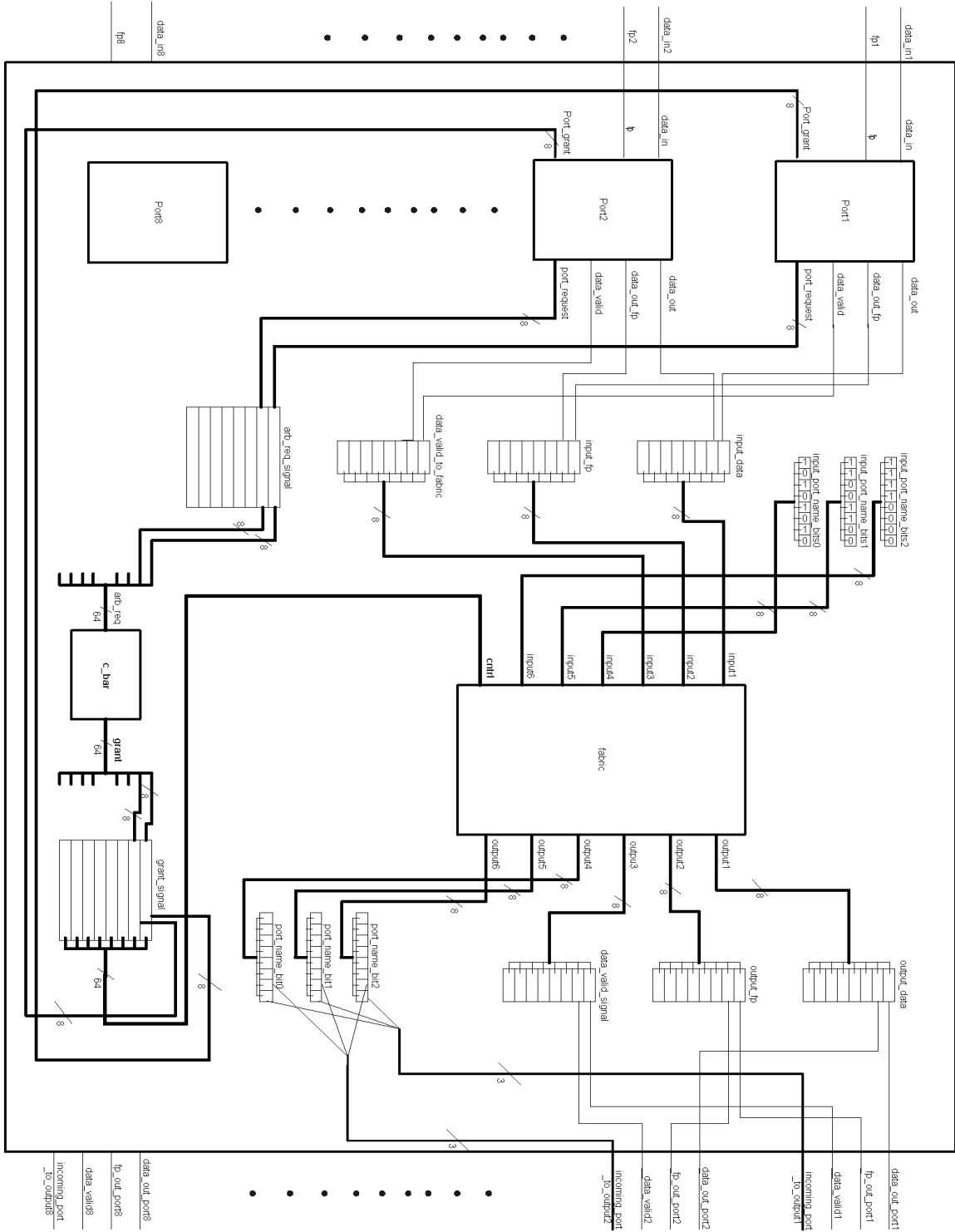
In this thesis, I described the architecture for a basic 8x8 input queued crossbar packet switch. The switch uses a DPA crossbar scheduler, input buffered ports, and a crossbar fabric. It is capable of switching traffic at high rates (5Mbps). The switch simulations indicate that the functional switch is capable of switching 8 ATM input lines into 8 ATM output lines at a rate of 5 Mbps. My original expectations to switch at higher rates cannot be met due to Altera FLEX10KE FPGA timing and speed limitations.

Although I have implemented the basic functions of a switch, my design is far from having performance necessary to be deployed in an actual network. Admission control algorithms, congestion control algorithms, head of line blocking prevention, and multicasting are only a few of many features, functions, and algorithms required for a fully functional network switch. My goal is to design and implement a fully operational switch capable of performing the very basic functions of a network switch. I hope that this design will attract many engineering students to join the group and to help advance the project.

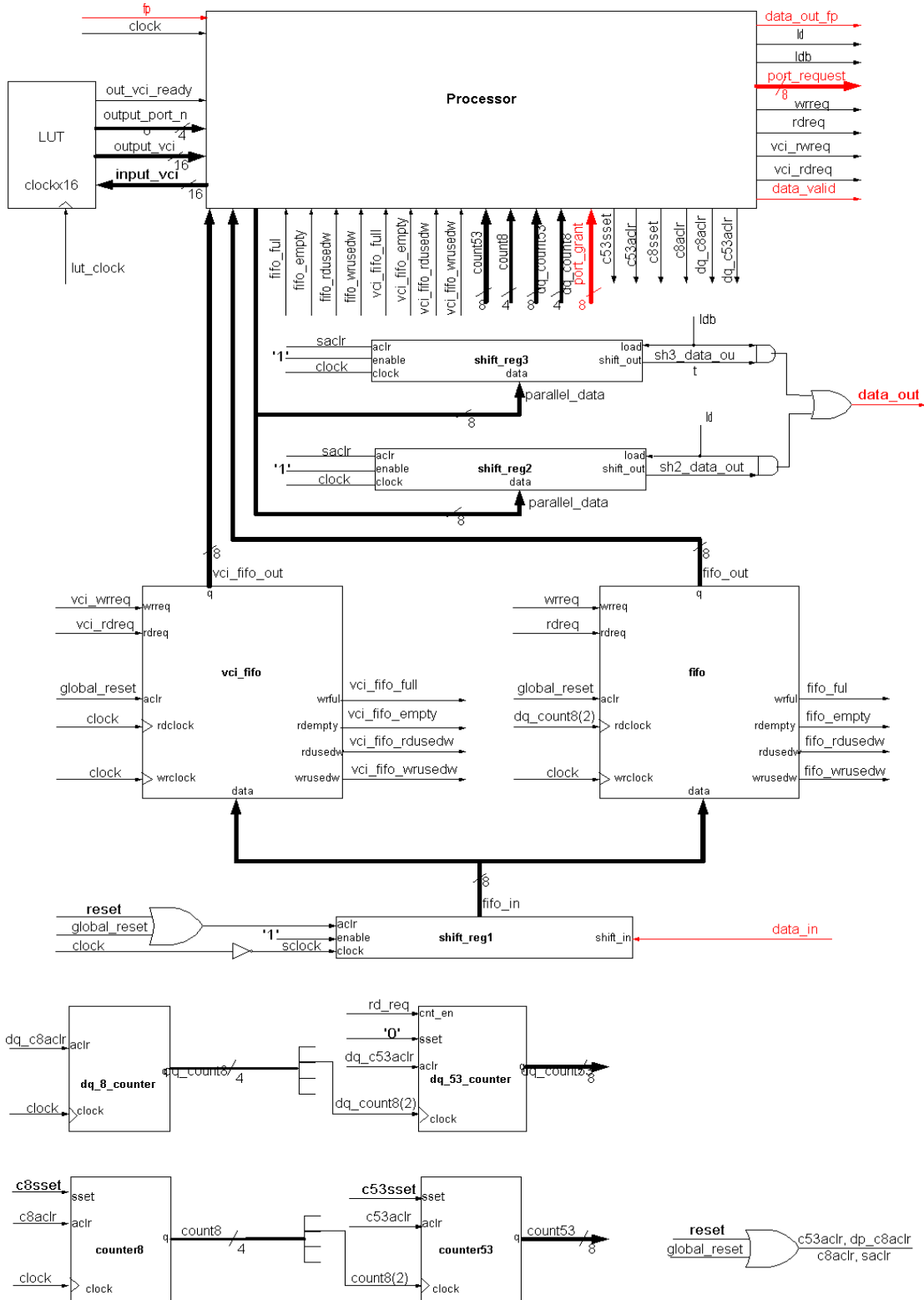
References

- [1] Nick McKeown, Martin Izzard, Adisak Mekkittikul, William Ellersick, and Mark Horowitz, "Tiny Tera: a packet switch core," *IEEE Micro*, Jan/Feb 1997, pp. 26-33.
- [2] Electrical Engineering, Washington University, <http://ee.wustl.edu/ee/research/programs/16.telecom.html>, Jan. 12, 2000.
- [3] Our latest thinking, Telcordia technologies, <http://www.telcordia.com/research/thinking.html>, Mar. 30, 2000.
- [4] The SwitchWare Project, <http://www.cis.upenn.edu/~switchware/home.html>, Jan. 14, 2000.
- [5] Jason J. Hickey, Tony J. Bogovic, Bruce S. Davie, William S. Marcus, Vince F. Massa, Ljiljana Trajkovic, and Daniel V. Wilson, "The architecture of the Sunshine broadband testbed," *Proc. ISS '92, Yokohama, Japan, Oct. 1992*, vol. 1, pp. 204-208.
- [6] James Hurt, Andrew May, Xiaohan Zhu, and Bill Lin, "Design and implementation of high-speed symmetric crossbar schedulers," *Proc. ICC'99, Vancouver, Canada, June 1999*, S37-6.
- [7] ALTERA MAXPLUS II software download, <http://www.altera.com/html/tools/baseline.html>, Oct. 21, 1999.
- [8] Yuval Tamir, and Gregory L. Frazier, "Dynamically-allocated multi-queue buffers for VLSI communication switches," *Proceedings of the 15th Annual Symposium on Computer Architecture*, June 1988.
- [9] Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan, "Input versus output queuing on a space-division packet switch," *IEEE Transactions on Communications*, vol. 35, no. 12, 1987.
- [10] Communication laboratory networks, <http://www.ensc.sfu.ca/research/cnl>, Oct. 5, 1999.
- [11] William Stallings, "Data and computer communications," 6th edition, Prentice Hall, NJ, 2000.
- [12] PMC-Sierra home page, <http://www.pmc-sierra.com>, May 2, 2000.

Appendix A: High level switch schematic

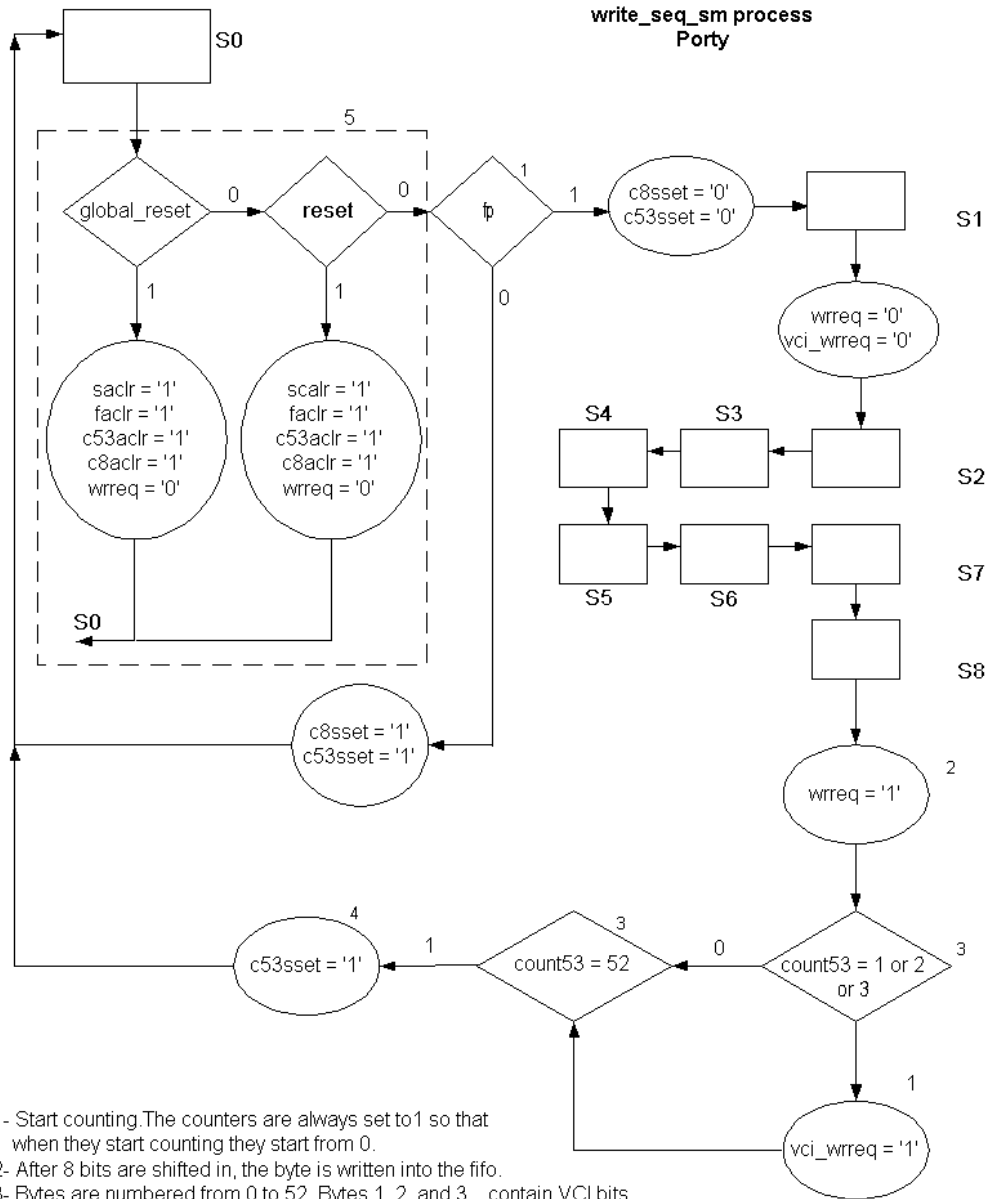


Appendix B: High level input port schematic



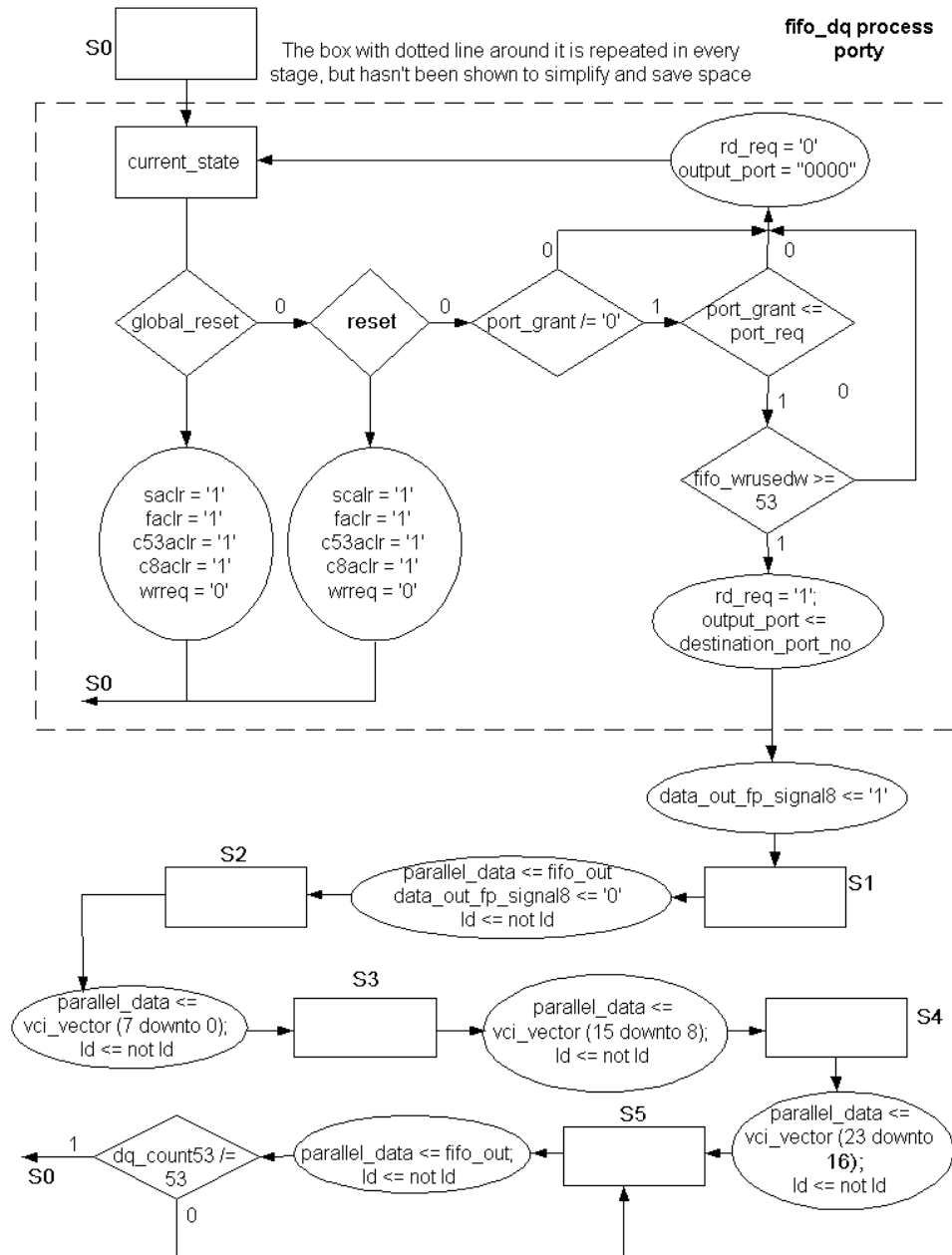
Appendix C: The port state machines

Appendix C.1: Write sequence state machine



- 1- Start counting. The counters are always set to 1 so that when they start counting they start from 0.
- 2- After 8 bits are shifted in, the byte is written into the fifo.
- 3- Bytes are numbered from 0 to 52. Bytes 1, 2, and 3 contain VCI bits.
- 4- One packet was written into the fifo, so now set the counter to 1, so that we can start counting from 0 again for the next packet to come.
- 5- The dotted box is repeated in every state but hasn't been shown for simplification.

Appendix C.3: FIFO dequeue state machine



Appendix D: VHDL codes for the switch and its components

Appendix D.1: VHDL code for the PORT component

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY porty IS
  GENERIC(
    PACKET_SIZE      : INTEGER:= 53; --Port is set to handle packets of size 53 bytes
    COUNTER_53_SIZE  : INTEGER:= 8;  --Counter 53 is an 8 bit counter so it can service packets
    upto 256 bytes long
    COUNTER_8_SIZE   : INTEGER:= 4;  --Counter 8 is a 4 bit counter
    DATA_SIZE       : INTEGER:= 8;  --Each data byte is 8 bits long
    FIFO_SIZE        : INTEGER:= 2048; --Number of words in FIFO
    FIFO_WIDTHTHU    : INTEGER:= 11; --Recommended value is CEIL(LOG2(FIFO_SIZE))
    VCI_FIFO_SIZE    : INTEGER:= 128; --For a full data FIFO at least 78 VCI bytes are required
    VCI_FIFO_WIDTHTHU : INTEGER:= 7;  --Recommended value is
    CEIL(LOG2(VCI_FIFO_SIZE))
    VCI_VECTOR_SIZE  : INTEGER:= 16; --Each VCI is 2 Bytes.
    OUTPUT_PORT_SIZE : INTEGER:= 4;  --4 bits used to address an output port
    TRANSLATION_TABLE : STRING := "lut1.mif" --the lookup table
  );

  PORT(
    --Ports to be replaced later

    Test Signals for simulation purposes
    state : OUT INTEGER RANGE 0 TO 8;
    write_req : OUT STD_LOGIC;
    read_req : OUT STD_LOGIC;
    counter53 : OUT STD_LOGIC_VECTOR(COUNTER_53_SIZE-1 DOWNTO 0);
    fifo_input : OUT STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNTO 0);
    counter8 : OUT STD_LOGIC_VECTOR(COUNTER_8_SIZE-1 DOWNTO 0);
    shrclock : OUT STD_LOGIC;
    c53sset1 : OUT STD_LOGIC;
    vci_write_req : OUT STD_LOGIC;
    vci_read_req : OUT STD_LOGIC;
    lut_input_vci : OUT STD_LOGIC_VECTOR (15 DOWNTO 0); --the signal that goes to LUT to
    be looked up
    fifo_empty : OUT STD_LOGIC; --Indicates that the FIFO is underflowing
    fifo_full : OUT STD_LOGIC; --Indicates that the FIFO is overflowing
    vci_fifo_empty : OUT STD_LOGIC; --Indicates that the VCI FIFO is
    underflowing
    vci_fifo_full : OUT STD_LOGIC; --Indicates that the VCI FIFO is overflowing
    vci_fifo_out : OUT STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNTO 0);
    lut_output_port_no : OUT STD_LOGIC_VECTOR (OUTPUT_PORT_SIZE-1 DOWNTO 0); --The
    destination output port number
    lut_output_vci : OUT STD_LOGIC_VECTOR (VCI_VECTOR_SIZE-1 DOWNTO 0); --The VCI to be
    placed in the outgoing packet
    lut_clock_en : OUT STD_LOGIC; --This is a clock enable for LUT.
    lut_out_vci_ready : OUT STD_LOGIC; --When 1, port processor knows that output VCI and port no. are ready for
    pickup
    vci_vec : OUT STD_LOGIC_VECTOR(23 DOWNTO 0);
    fifo_output : OUT STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNTO 0);
    dq_53_clk : OUT STD_LOGIC;
    dq_count53_out : OUT STD_LOGIC_VECTOR (COUNTER_53_SIZE-1 DOWNTO 0);
    parallel_data_out : OUT STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNTO 0);
    sh2_out : OUT STD_LOGIC;
    sh3_out : OUT STD_LOGIC;
  
```

```

shr2_content : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNT0 0);
shr3_content : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNT0 0);
ld_out      : OUT STD_LOGIC;
ldb_out     : OUT STD_LOGIC;
data_fp_signal8 : OUT STD_LOGIC;

--Actual entity ports
data_in      : IN      STD_LOGIC;      --Input serial data to the port
clock       : IN      STD_LOGIC;      --Input clock to the port
fp         : IN      STD_LOGIC;      --Input frame pulse to the port
global_reset : IN      STD_LOGIC;      --Resets all the counters, registers and the FIFO
reset      : IN      STD_LOGIC;      --Resets everything but the FIFO
port_grant  : IN      STD_LOGIC_VECTOR(7 DOWNT0 0); --The grant vector for the port
data_out   : OUT      STD_LOGIC;      --Serial data output of the port
data_out_fp : OUT      STD_LOGIC;      -- frame pulse showing the beginning of the data being shifted out
data_valid  : OUT      STD_LOGIC;      -- is 1 when FIFO is dequeuing data (i.e. a grant is
issued for the port)
output_port : OUT      STD_LOGIC_VECTOR (OUTPUT_PORT_SIZE-1 DOWNT0 0); --The destination
output port number
port_request : OUT      STD_LOGIC_VECTOR(7 DOWNT0 0) --The request vector for the port

);

END porty;

ARCHITECTURE behav OF porty IS

--Component Declaration

COMPONENT LUT

    GENERIC (VCI_SIZE: INTEGER := 16;          --number of bits in a VCI
             PORT_SIZE: INTEGER := 4;         --number of bits in a port number
             ROM_WIDTH: INTEGER := 36;        --width of the look up table
             ROM_WIDTHHAD : INTEGER := 3;     --address width of the look up table = log2(number of
rows in the table) there are 8 rows in each table
             TRANSLATION_TABLE: STRING       --The file serving as a look up
table
    );

    PORT (input_vci: IN      STD_LOGIC_VECTOR (VCI_SIZE-1 downto 0);
          output_port_no: OUT STD_LOGIC_VECTOR (PORT_SIZE-1 downto 0 );
          output_vci: OUT   STD_LOGIC_VECTOR (VCI_SIZE-1 downto 0);
          clockx16: IN      STD_LOGIC;      --the clock for LUT has to faster than the main
clock so that looking up could be done faster
          renable: OUT      STD_LOGIC
    );

END COMPONENT;

COMPONENT decoder

    PORT(enable : IN      STD_LOGIC;      --If asserted, the decoder is active. If deasserted, the
outputs are tri-stated
          I      : IN      STD_LOGIC_VECTOR(2 DOWNT0 0); --3 bits input of the
decoder
          O      : OUT      STD_LOGIC_vector(7 DOWNT0 0) --8 bits output of the
decoder
    );

END COMPONENT;

SIGNAL HIGH      : STD_LOGIC;
SIGNAL LOW       : STD_LOGIC;

--Shift Register 1 Signals
SIGNAL sclock    : STD_LOGIC;          --Shift register clock
SIGNAL saclr     : STD_LOGIC;          --Asynchronous clear for both shift registers

```

```

--Shift Register 2 Signals
SIGNAL sh2_data_out      : STD_LOGIC;
SIGNAL shreg2_content : STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNT0 0);

--Shift Register 3 Signals
SIGNAL sh3_data_out      : STD_LOGIC;
SIGNAL shreg3_content : STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNT0 0);

--Counter 8 Signals
SIGNAL c8aclr      : STD_LOGIC;           --Asynchronous clear for counter 8
SIGNAL count8      : STD_LOGIC_VECTOR (COUNTER_8_SIZE-1 DOWNT0 0); --4 Bit output of counter 8
SIGNAL c8sset      : STD_LOGIC;           --Synchronous clear

--Counter 53 Signals
SIGNAL c53aclr      : STD_LOGIC;           --Asynchronous clear for counter 53
SIGNAL count53      : STD_LOGIC_VECTOR (COUNTER_53_SIZE-1 DOWNT0 0); --8 Bit output of counter 53
SIGNAL c53sset      : STD_LOGIC;           --Synchronous clear

--Dequeue counter 53 Signals
SIGNAL dq_clock      : STD_LOGIC;           --The clock to the counter
SIGNAL dq_c53aclr    : STD_LOGIC;           ---Asynchronous clear
SIGNAL dq_count53    : STD_LOGIC_VECTOR (COUNTER_53_SIZE-1 DOWNT0 0); --8 Bit output of dq counter 53
SIGNAL dq_c53aset    : STD_LOGIC;

--Dequeue counter 8 Signals
SIGNAL dq_c8aclr     : STD_LOGIC;           --Asynchronous clear for dq counter 8
SIGNAL dq_count8     : STD_LOGIC_VECTOR (COUNTER_8_SIZE-1 DOWNT0 0); --4 Bit output of dq counter 8
--SIGNAL c8sset      : STD_LOGIC;           --Synchronous clear

--FIFO Signals
SIGNAL fifo_in       : STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNT0 0); --The data to be written to
FIFO
SIGNAL faclr         : STD_LOGIC;           --Asynchronous clear for FIFO
SIGNAL wrreq         : STD_LOGIC;           --Write request sent to the FIFO
SIGNAL rdreq         : STD_LOGIC;           --Read request sent to the FIFO
SIGNAL fifo_rducedw : STD_LOGIC_VECTOR(FIFO_WIDTH-1 DOWNT0 0); --Number of bytes in the main FIFO
(not used by the processor)
SIGNAL fifo_wrusedw : STD_LOGIC_VECTOR(FIFO_WIDTH-1 DOWNT0 0); --Number of bytes in the main FIFO
(used by the processor)
SIGNAL fifo_out      : STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNT0 0); --Output of the FIFO

--VCI FIFO Signals
SIGNAL vci_faclr     : STD_LOGIC;           --Asynchronous clear for VCI FIFO
SIGNAL vci_wrreq     : STD_LOGIC;           --Write request sent to the VCI FIFO
SIGNAL vci_rdreq     : STD_LOGIC;           --Read request sent to the VCI FIFO
SIGNAL vci_rducedw   : STD_LOGIC_VECTOR(VCI_FIFO_WIDTH-1 DOWNT0 0); --Number of bytes in the
VCI FIFO (not used by the processor)
SIGNAL vci_wrusedw   : STD_LOGIC_VECTOR(VCI_FIFO_WIDTH-1 DOWNT0 0); --Number of bytes in the
VCI FIFO (used by the processor)
SIGNAL vci_fifo_empty_signal: STD_LOGIC; --This is the signal replacing vci_fifo_empty output port in if statements
SIGNAL vci_fifo_out_signal : STD_LOGIC_VECTOR (7 DOWNT0 0);
SIGNAL input_vci     : STD_LOGIC_VECTOR (15 DOWNT0 0); -- to be connected to the input_vci going into the
look up table

--LUT Signals
SIGNAL output_port_no : STD_LOGIC_VECTOR (OUTPUT_PORT_SIZE-1 DOWNT0 0); --The destination output
port number
SIGNAL output_vci     : STD_LOGIC_VECTOR (VCI_VECTOR_SIZE-1 DOWNT0 0); --The VCI to be placed in the
outgoing packet
SIGNAL out_vci_ready : STD_LOGIC; --When 1, port processor knows that output VCI and port no. are ready for
pickup
SIGNAL lut_clock      : STD_LOGIC;           --The clock that will be connected to LUT through lut_run clock.

--State Machines
SIGNAL current_state : INTEGER RANGE 0 TO 8;
SIGNAL next_state    : INTEGER RANGE 0 TO 8;
SIGNAL cntrl_current_state : INTEGER RANGE 0 TO 8;
SIGNAL cntrl_next_state : INTEGER RANGE 0 TO 8;

```

```

        SIGNAL DQ_state          : INTEGER RANGE 0 TO 8;
        SIGNAL DQ_next_state    : INTEGER RANGE 0 TO 8;
        SIGNAL fp_state         : INTEGER RANGE 0 TO 8;
        SIGNAL fp_next_state    : INTEGER RANGE 0 TO 8;

        --Processor Specific Signals
        SIGNAL vci_vector       : STD_LOGIC_VECTOR(23 DOWNT0 0);    --3 byte long vector to hold the
three bytes coming out of VCI FIFO
        SIGNAL destination_port_no : STD_LOGIC_VECTOR(OUTPUT_PORT_SIZE-1 DOWNT0 0); --The destination port
number.
        SIGNAL port_req        : STD_LOGIC_VECTOR(7 DOWNT0 0); --The request vector to be sent out
        SIGNAL ld              : STD_LOGIC; --signal for loading the output shift registers
        SIGNAL ldb            : STD_LOGIC; --signal for loading the output shift registers
        SIGNAL parallel_data   : STD_LOGIC_VECTOR(DATA_SIZE-1 DOWNT0 0);
        SIGNAL data_out_fp_signal : STD_LOGIC;
        SIGNAL data_out_fp_signal8 : STD_LOGIC;

BEGIN

        --TEST Signals for simulation purposes
        state <= current_state;
        write_req <= wrreq;
        read_req <= rdreq;
        vci_write_req <= vci_wrreq;
        vci_read_req <= vci_rdreq;
        counter53 <= count53;
        fifo_input <= fifo_in;
        counter8 <= count8;
        shrelock <= sclock;
        c53sset1 <= c53sset;
        vci_fifo_empty <= vci_fifo_empty_signal; --vci_fifo_empty_signal is used in the vci fifo portmap so that it can be used in
if statement
        vci_fifo_out <= vci_fifo_out_signal;
        lut_output_port_no <= output_port_no;
        lut_output_vci <= output_vci;
        lut_clock_en <= lut_clock;
        lut_out_vci_ready <= out_vci_ready;
        lut_input_vci <= input_vci;
        vci_vec <= vci_vector;
        fifo_output <= fifo_out;
        dq_53_clk <= dq_clock;
        dq_count53_out <= dq_count53;
        parallel_data_out <= parallel_data;
        sh2_out <= sh2_data_out;
        sh3_out <= sh3_data_out;
        shr2_content <= shreg2_content;
        shr3_content <= shreg3_content;
        ld_out <= ld;
        ldb_out <= ldb;
        data_out_fp <= data_out_fp_signal;
        data_fp_signal8 <= data_out_fp_signal8;

        Write_Seq_SM : PROCESS (clock)

                                BEGIN --Process
                                IF (clock='0' AND clock'event) THEN -- at the falling edge of the
clock next state is calculated
                                        IF ( (global_reset = '0') AND (reset = '0')) THEN

                                                CASE current_state IS

                                                        WHEN 0 =>                IF ( fp = '1' ) THEN -
- frame pulse coming in state zero shows the beginning of a new packet

                                        next_state <= 1;

                                        c8sset <= '0'; --start counting if there was a frame pulse

                                        c53sset <= '0';

```

```

ELSE
c8sset <= '1'; --set the counters to one, and don't count if no frame pulse
c53sset <= '1'; --we set the counters because we want to start counting from 0.
--If the counters are reset, they start counting from 1.
END IF;
wrreq <=
'0';
vci_wrreq
<= '0';
from state 1 to 8 bits are shifted in
WHEN 1 => next_state <= 2; --
wrreq <=
'0';
vci_wrreq
<= '0';
WHEN 2 => next_state <= 3;
WHEN 3 => next_state <= 4;
WHEN 4 => next_state <= 5;
WHEN 5 => next_state <= 6;
WHEN 6 => next_state <= 7;
WHEN 7 => next_state <= 8;
WHEN 8 => IF (count53 /=
"00110100") THEN -- continue reading the bits if 53 bytes are not read yet
next_state <= 1;
ELSE
-- if the whole packet has been received
next_state <= 0;
c53sset <= '1'; --set counter 53 so that when it resumes counting it starts from 0
END IF;
wrreq <=
'1'; -- write the one byte that was shifted in, into the fifo
-- Send
VCI bytes (bytes 2,3 and 4 of the header) to the VCI FIFO
IF
((count53 = "0000001") OR (count53 = "0000010") OR (count53 = "0000011")) THEN
vci_wrreq <= '1';
END IF;
WHEN OTHERS => NULL;
END CASE;
ELSE
IF ( (global_reset = '1') OR (reset = '1')) THEN --In case of reset
next_state <= 0;
wrreq <= '0';
END IF;
END IF;
END PROCESS;

```

```

PROCESS (clock)
BEGIN -- Process
IF ( (global_reset = '1') OR (reset = '1')) THEN --check for reset
current_state <= 0;
cntrl_current_state <= 0;
ELSE
IF (clock = '1' AND clock'event) THEN -- at the rising edge of the clock,
update the states
current_state <= next_state;
cntrl_current_state <= cntrl_next_state;
END IF;
END IF;
END PROCESS;

```

```

VCI_Controller : PROCESS (clock)
BEGIN -- Process
IF (clock = '0' AND clock'event) THEN --
cntrl_next_state is determined on the falling edge
IF ( (global_reset = '0') AND (reset = '0')) THEN
CASE cntrl_current_state IS
WHEN 0 =>
--if the VCI_fifo is
--we want to start
--The reason for four
--that the fifos will
IF
not empty and there are more than or equal to 4 bytes in it
reading those bytes from the VCI_fifo.
is to make sure that the whole packet is in the fifo and
never be empty.
((vci_fifo_empty_signal = '0') AND (vci_wrusedw >= "0000100")) THEN
cntrl_next_state <= 1;
vci_rdreql
<= '1'; --start reading from the VCI(take affect at the next falling edge)
ELSE
cntrl_next_state <= 0;
vci_rdreql
<= '0';
END IF;
destination_port_no
dqc53aclr <= '0'; --
WHEN 1 =>
--first byte is written
cntrl_next_state <= 2;
vci_vector (7 downto
0) <= vci_fifo_out_signal;
WHEN 2 =>
--second byte is
cntrl_next_state <= 3;
vci_vector (15
downto 8) <= vci_fifo_out_signal;
WHEN 3 =>

```

```

--Thirs byte is written
into the MSbyte of the temp signal
stop reading from the VCI(takes affect at the next falling edge)
downto 16) <= vci_fifo_out_signal;
downto 4); --connect the VCI part of the temp signal to the input of LUT
--wait for one clock cycle, because of delay in looking up in LUT
'1') THEN --If the LUT has come up with a value
(19 downto 4) <= output_vci; --write back the outgoing packet's VCI
    cntrl_next_state <= 7;
    destination_port_no <= output_port_no; --retrive the destination port number
    cntrl_next_state <= 6; --stay until LUT has come up with a value
'00110101") THEN --IF 53 packets have been Dequeued go to state 0
    cntrl_next_state <= 0;
    <= '1'; --and clear the counter so that it starts from zero again
    cntrl_next_state <= 7; --if 53 packets haven't been DQed, wait until they are.

    WHEN OTHERS => NULL;
    END CASE;
    END IF;
    END IF;
END PROCESS;

FIFO_DQ : PROCESS (dq_clock)
BEGIN --Process
    --In the DQ process, bytes of data are loaded to the out shift register with the
    speed of dq_clock
    --which is 8 times slower than the main clock. After each loading of the
    output shift register,
    --the data is shifted out with the speed of the main clock.
    determined on the falling edge
    IF (dq_clock = '0' AND dq_clock'event) THEN --cntrl_next_state is
    IF ( (global_reset = '0') AND (reset = '0')) THEN
    --Issue a read request from buffer, if there is a non zero grant produced which
    --there are at least 53 bytes in the buffer.
    IF ((port_grant /= "00000000") AND (port_grant = port_req) AND
    (fifo_wrusedw >= "00000110101")) THEN
        rdreq <= '1';

```

```

output_port <= destination_port_no; --retrieve the destination port
number
CASE DQ_state IS
    WHEN 0 =>
        DQ_next_state <= 1;
        data_out_fp_signal8 <= '1'; --produce a
frame pulse for the outgoing data

    WHEN 1 =>
        DQ_next_state <= 2;
        parallel_data <= fifo_out; --load the shift
register with the data read from the fifo
        ld <= not ld; --shift the data out
        data_out_fp_signal8 <= '0'; --framepulse is
high for only one clock cycle

--In states 2, 3, and 4 the vci vector with the new VCI is
connected to the shift register input
    WHEN 2 =>
        DQ_next_state <= 3;
        parallel_data <= vci_vector (7 downto 0);
        ld <= not ld;

    WHEN 3 =>
        DQ_next_state <= 4;
        parallel_data <= vci_vector (15 downto 8);
        ld <= not ld;

    WHEN 4 =>
        DQ_next_state <= 5;
        parallel_data <= vci_vector (23 downto 16);
        ld <= not ld;

    WHEN 5 =>
        --If haven't reached count 53, do nothing i.e.
        --let the rest of the data be shifted out
        IF (dq_count53 /= "00110101") THEN
            ELSE
                DQ_next_state <= 0;
            END IF;
            parallel_data <= fifo_out;
            ld <= not ld;

        WHEN OTHERS => NULL;

    END CASE;
ELSE --If there is no valid grant, then dont' dequeue and let the destination
port be zero (which
    --non existant.
    rdreq <= '0';
    output_port <= "0000";

    END IF;
    END IF;
    END IF;

END PROCESS;

PROCESS (dq_clock)
BEGIN -- Process
    IF ( (global_reset = '1') OR (reset = '1')) THEN --check for reset
        DQ_state <= 0;
    ELSE
        IF (dq_clock = '1' AND dq_clock'event) THEN -- at the rising edge of the
clock, update the states
            DQ_state <= DQ_next_state;

```

```

                                END IF;
                                END IF;
                                END PROCESS;

data_out_fp_generator: PROCESS (clock)
BEGIN
    IF (clock = '1' AND clock'event) THEN
        IF ( (global_reset = '0') AND (reset = '0')) THEN
            IF data_out_fp_signal8 = '1' THEN

                CASE fp_state is

                    WHEN 0 => fp_next_state <= 1;
                    WHEN 1 => fp_next_state <= 2;
                    WHEN 2 => fp_next_state <= 3;
                    WHEN 3 => fp_next_state <= 4;
                    WHEN 4 => fp_next_state <= 5;
                    WHEN 5 => fp_next_state <= 6;
                    WHEN 6 => fp_next_state <= 7;
                    WHEN 7 => fp_next_state <= 0;
                                data_out_fp_signal <= '1';
                                data_out_fp_signal <= '0';
                    WHEN others => NULL;

                END CASE;

            END IF;
        END IF;
    END PROCESS;

fp_out: PROCESS (clock)
BEGIN -- Process
    IF ( (global_reset = '1') OR (reset = '1')) THEN --check for reset
        fp_state <= 0;
    ELSE
        IF (clock = '0' AND clock'event) THEN -- at the rising edge of the clock,
            fp_state <= fp_next_state;
        END IF;
    END IF;
END PROCESS;

-- Concurrent Procedure Call
-- Concurrent Signal Assignment
-- Conditional Signal Assignment
-- Selected Signal Assignment

HIGH    <= '1';
LOW     <= '0';
sclock  <= NOT clock; --clock for shift register
c8aclr  <= reset OR global_reset;
c53aclr <= reset OR global_reset;
dq_c8aclr <= reset OR global_reset;
saclr   <= reset OR global_reset;
faclr   <= global_reset;
vci_faclr <= global_reset;
lut_clock <= clock;--Faster clocks may replace clock so that LUT can function faster
port_request <= port_req;

```

```

dq_clock    <= dq_count8(2); --a clock 8 times slower than the main clock
dq_c53aset <= '0';
ldb        <= not ld;
data_valid <= rdreq; --whenever the read request goes high, the data on the output of the LUT is valid

-- Component Instantiation Statement
counter_8 : lpm_counter --counts the number of bits
          GENERIC MAP (LPM_WIDTH => COUNTER_8_SIZE)

          PORT MAP (clock => clock,
                   aclr => c8aclr,
                   sset => c8sset,
                   q   => count8
                   );

counter_53 : lpm_counter --counts the number of bytes
          GENERIC MAP (LPM_WIDTH => COUNTER_53_SIZE)

          PORT MAP (clock => count8(2), --this clock makes the counter increment every 8
clock cycles
                   aclr => c53aclr,
                   aset => c53sset,
                   q   => count53
                   );

dq_8_counter : lpm_counter --counts the number of bits being shifted out
          GENERIC MAP (LPM_WIDTH => COUNTER_8_SIZE)

          PORT MAP (clock => clock,
                   aclr => dq_c8aclr,
                   q   => dq_count8
                   );

dq_53_counter : lpm_counter --counts the number of bytes being shifted out
          GENERIC MAP (LPM_WIDTH => COUNTER_53_SIZE)

          PORT MAP (clock => dq_clock,
                   aclr => dq_c53aclr,
                   aset => dq_c53aset,
                   cnt_en => rdreq,
                   q   => dq_count53
                   );

shift_reg1 : lpm_shiftreg --first shift register that takes the incoming serial data in
          GENERIC MAP (LPM_WIDTH => DATA_SIZE)

          PORT MAP (clock => sclock,
                   enable => HIGH,
                   shiftin => data_in,
                   aclr => saclr,
                   q   => fifo_in --output of the shift register going to
the fifo input
                   );

fifox      : lpm_fifo_dc --The input port buffer(FIFO)
          GENERIC MAP (LPM_WIDTH  => DATA_SIZE,
                   LPM_NUMWORDS => FIFO_SIZE,
                   LPM_WIDTHTHU => FIFO_WIDTHTHU
                   )

          PORT MAP (data => fifo_in,
rdclock => dq_count8(2), --read clock is the clock for dequeuing
wrclock => clock,      --writing clock is the main clock
wrreq => wrreq,
rdreq => rdreq,
aclr => faclr,
q   => fifo_out, --output of the fifo
wrfull => fifo_full,
rdempty => fifo_empty,

```

```

        rdusedw => fifo_rdusedw,
        wrusedw => fifo_wrusedw
    );

vci_fifo : lpm_fifo_dc --The FIFO that stores second, third and fourth byte of each packet
    GENERIC MAP (LPM_WIDTH => DATA_SIZE,
        LPM_NUMWORDS => VCI_FIFO_SIZE,
        LPM_WIDTHU => VCI_FIFO_WIDTHU
    )

    PORT MAP (data => fifo_in,
        rdclock => clock,
        wrclock => clock,
        wrreq => vci_wrreq,
        rdreq => vci_rdreq,
        aclr => vci_faclr,
        q => vci_fifo_out_signal,
        wrfull => vci_fifo_full,
        rdempty => vci_fifo_empty_signal,
        rdusedw => vci_rdusedw,
        wrusedw => vci_wrusedw
    );

port_lut : LUT --The look up table where the destination port number and the outgoing VCI are looked up
    GENERIC MAP (VCI_SIZE => VCI_VECTOR_SIZE,
        PORT_SIZE => OUTPUT_PORT_SIZE,
        TRANSLATION_TABLE =>
TRANSLATION_TABLE
    )

    PORT MAP (input_vci => input_vci,
        output_port_no => output_port_no,
        output_vci => output_vci,
        clockx16 => lut_clock,
        renable => out_vci_ready
    );

request_decoder : decoder --Three to eight decoder to change the destination port number
    --The fourth bit of all destination_port_no's are 1. except all zeros when none yet produced.
    --so when bit 3 is zero, decoder disabled
    PORT MAP(enable => destination_port_no(3),
        I => destination_port_no(2 DOWNTO 0),
        O => port_req
    );

--There are two shift registers at the output of the port. When one of them is being loaded with
--data from the FIFO(with an 8 times slower clock), the other one is shifting the data out to the
--output port, and vice versa.

shift_reg2 : lpm_shiftreg --First shift register at the output of the port
    GENERIC MAP (LPM_WIDTH => DATA_SIZE)

    PORT MAP (clock => clock,
        enable => HIGH,
        shiftout => sh2_data_out,
        aclr => saclr,
        q => shreg2_content,
        load => ld, --when
HIGH, a byte is parallel loaded
        data => parallel_data --data from the FFIO
    that is loaded to the shift register
    );

shift_reg3 : lpm_shiftreg --Second shift register at the output of the port
    GENERIC MAP (LPM_WIDTH => DATA_SIZE)

    PORT MAP (clock => clock,
        enable => HIGH,
        shiftout => sh3_data_out,

```

```
aclr => saclr,  
q => shreg3_content,  
load => ldb,  
data => parallel_data  
);
```

--The out going data is the data shifted out from the two shift registers.(Only one is shifting data out at a time)
data_out <= ((ldb AND sh2_data_out) OR (ld AND sh3_data_out));

```
-- Generate Statement  
END behav;
```

Appendix D.2: VHDL code for DPA structure and Arbiter cell.

Appendix D.2.1: VHDL code for Arbiter cell

```
-- Arbiter.vhd

library ieee;
use ieee.std_logic_1164.all;

entity Arbiter is
    port (Req, North, West, Mask: in std_logic;
          South, East, Grant: out std_logic);
end Arbiter;

architecture Behavior of Arbiter is
    signal req1 : std_logic;
    signal east1 : std_logic;
    signal south1 : std_logic;
begin
    req1 <= (Mask and Req);
    south1 <= ((not (North and req1 and West)) and North);
    east1 <= ((not (North and req1 and West)) and West);
    East <= ((not Mask) or east1);
    South <= ((not Mask) or south1);
    Grant <= (North and req1 and West);
end Behavior;
```

Appendix D.2.2: VHDL code for DPA structure.

```
library ieee;
use ieee.std_logic_1164.all;

ENTITY c_bar IS
    PORT(arb_req: IN std_logic_vector(64 DOWNTO 1);
         clk, reset : IN std_logic;
         grant : OUT std_logic_vector(64 DOWNTO 1);
         P: OUT std_logic_vector(15 DOWNTO 1)
        );
END c_bar;

ARCHITECTURE behaviour OF c_bar IS

    COMPONENT Arbiter
        PORT(Req, North, West, Mask: IN std_logic;
             South, East, Grant: OUT std_logic);
    END COMPONENT;

    TYPE c_bar_signal_array IS ARRAY (natural RANGE <>, natural RANGE <>) OF std_logic;

    --Cross Bar Signal Declarations
    SIGNAL south_2_north : c_bar_signal_array(1 TO 15, 1 TO 8);
    SIGNAL east_2_west   : c_bar_signal_array(1 TO 15, 1 TO 8);
    SIGNAL arb_mask      : c_bar_signal_array(1 TO 15, 1 TO 8);
    SIGNAL arb_grant     : c_bar_signal_array(1 TO 15, 1 TO 8);
    SIGNAL c_bar_P       : std_logic_vector(15 DOWNTO 1);
    SIGNAL High          : std_logic;

    BEGIN

grant(1) <= arb_grant(1,1) or arb_grant(9,1);
grant(2) <= arb_grant(1,2) or arb_grant(9,2);
grant(3) <= arb_grant(1,3) or arb_grant(9,3);
grant(4) <= arb_grant(1,4) or arb_grant(9,4);
grant(5) <= arb_grant(1,5) or arb_grant(9,5);
grant(6) <= arb_grant(1,6) or arb_grant(9,6);
grant(7) <= arb_grant(1,7) or arb_grant(9,7);
grant(8) <= arb_grant(1,8);

grant(9) <= arb_grant(2,1) or arb_grant(10,1);
grant(10) <= arb_grant(2,2) or arb_grant(10,2);
grant(11) <= arb_grant(2,3) or arb_grant(10,3);
grant(12) <= arb_grant(2,4) or arb_grant(10,4);
grant(13) <= arb_grant(2,5) or arb_grant(10,5);
grant(14) <= arb_grant(2,6) or arb_grant(10,6);
grant(15) <= arb_grant(2,7);
grant(16) <= arb_grant(2,8) or arb_grant(9,8);

grant(17) <= arb_grant(3,1) or arb_grant(11,1);
grant(18) <= arb_grant(3,2) or arb_grant(11,2);
grant(19) <= arb_grant(3,3) or arb_grant(11,3);
grant(20) <= arb_grant(3,4) or arb_grant(11,4);
grant(21) <= arb_grant(3,5) or arb_grant(11,5);
grant(22) <= arb_grant(3,6);
grant(23) <= arb_grant(3,7) or arb_grant(10,7);
grant(24) <= arb_grant(3,8) or arb_grant(10,8);

grant(25) <= arb_grant(4,1) or arb_grant(12,1);
grant(26) <= arb_grant(4,2) or arb_grant(12,2);
```

```

grant(27) <= arb_grant(4,3) or arb_grant(12,3);
grant(28) <= arb_grant(4,4) or arb_grant(12,4);
grant(29) <= arb_grant(4,5);
grant(30) <= arb_grant(4,6) or arb_grant(11,6);
grant(31) <= arb_grant(4,7) or arb_grant(11,7);
grant(32) <= arb_grant(4,8) or arb_grant(11,8);

grant(33) <= arb_grant(5,1) or arb_grant(13,1);
grant(34) <= arb_grant(5,2) or arb_grant(13,2);
grant(35) <= arb_grant(5,3) or arb_grant(13,3);
grant(36) <= arb_grant(5,4);
grant(37) <= arb_grant(5,5) or arb_grant(12,5);
grant(38) <= arb_grant(5,6) or arb_grant(12,6);
grant(39) <= arb_grant(5,7) or arb_grant(12,7);
grant(40) <= arb_grant(5,8) or arb_grant(12,8);

grant(41) <= arb_grant(6,1) or arb_grant(14,1);
grant(42) <= arb_grant(6,2) or arb_grant(14,2);
grant(43) <= arb_grant(6,3);
grant(44) <= arb_grant(6,4) or arb_grant(13,4);
grant(45) <= arb_grant(6,5) or arb_grant(13,5);
grant(46) <= arb_grant(6,6) or arb_grant(13,6);
grant(47) <= arb_grant(6,7) or arb_grant(13,7);
grant(48) <= arb_grant(6,8) or arb_grant(13,8);

grant(49) <= arb_grant(7,1) or arb_grant(15,1);
grant(50) <= arb_grant(7,2);
grant(51) <= arb_grant(7,3) or arb_grant(14,3);
grant(52) <= arb_grant(7,4) or arb_grant(14,4);
grant(53) <= arb_grant(7,5) or arb_grant(14,5);
grant(54) <= arb_grant(7,6) or arb_grant(14,6);
grant(55) <= arb_grant(7,7) or arb_grant(14,7);
grant(56) <= arb_grant(7,8) or arb_grant(14,8);

grant(57) <= arb_grant(8,1);
grant(58) <= arb_grant(8,2) or arb_grant(15,2);
grant(59) <= arb_grant(8,3) or arb_grant(15,3);
grant(60) <= arb_grant(8,4) or arb_grant(15,4);
grant(61) <= arb_grant(8,5) or arb_grant(15,5);
grant(62) <= arb_grant(8,6) or arb_grant(15,6);
grant(63) <= arb_grant(8,7) or arb_grant(15,7);
grant(64) <= arb_grant(8,8) or arb_grant(15,8);

P <= c_bar_P;

Active_Win : process (clk, reset)
BEGIN
    if reset = '0' then
        c_bar_P <= "0000000000000000";
    elsif (clk = '1' and clk'event) then
        case c_bar_P is
            when "1111111100000000" => c_bar_P <= "0111111110000000";
            when "0111111110000000" => c_bar_P <= "0011111111000000";
            when "0011111111000000" => c_bar_P <= "0001111111100000";
            when "0001111111100000" => c_bar_P <= "0000111111110000";
            when "0000111111110000" => c_bar_P <= "0000011111111000";
            when "0000011111111000" => c_bar_P <= "0000001111111100";
            when "0000001111111100" => c_bar_P <= "0000000111111110";
            when "0000000111111110" => c_bar_P <= "0000000011111111";
            when "0000000011111111" => c_bar_P <= "1111111100000000";
            when others => c_bar_P <= "1111111100000000";
        end case;
    end if;
end process;

High <= '1';

--Arbiter_Row_Col

--First Row

```

```

Arbiter_1_1: Arbiter
    PORT MAP (Req => arb_req(1), North => High, West => High, Mask => c_bar_P(15),
              South => south_2_north(1,1), East => east_2_west(1,1) , Grant => arb_grant(1,1));

Arbiter_1_2: Arbiter
c_bar_P(14),
    PORT MAP (Req => arb_req(2), North => south_2_north(15,2), West => east_2_west(1,1), Mask =>
              South => south_2_north(1,2), East => east_2_west(1,2) , Grant => arb_grant(1,2));

Arbiter_1_3: Arbiter
c_bar_P(13),
    PORT MAP (Req => arb_req(3), North => south_2_north(15,3), West => east_2_west(1,2), Mask =>
              South => south_2_north(1,3), East => east_2_west(1,3) , Grant => arb_grant(1,3));

Arbiter_1_4: Arbiter
c_bar_P(12),
    PORT MAP (Req => arb_req(4), North => south_2_north(15,4), West => east_2_west(1,3), Mask =>
              South => south_2_north(1,4), East => east_2_west(1,4) , Grant => arb_grant(1,4));

Arbiter_1_5: Arbiter
c_bar_P(11),
    PORT MAP (Req => arb_req(5), North => south_2_north(15,5), West => east_2_west(1,4), Mask =>
              South => south_2_north(1,5), East => east_2_west(1,5) , Grant => arb_grant(1,5));

Arbiter_1_6: Arbiter
c_bar_P(10),
    PORT MAP (Req => arb_req(6), North => south_2_north(15,6), West => east_2_west(1,5), Mask =>
              South => south_2_north(1,6), East => east_2_west(1,6) , Grant => arb_grant(1,6));

Arbiter_1_7: Arbiter
c_bar_P(9),
    PORT MAP (Req => arb_req(7), North => south_2_north(15,7), West => east_2_west(1,6), Mask =>
              South => south_2_north(1,7), East => east_2_west(1,7) , Grant => arb_grant(1,7));

Arbiter_1_8: Arbiter
c_bar_P(8),
    PORT MAP (Req => arb_req(8), North => south_2_north(15,8), West => east_2_west(1,7), Mask =>
              South => south_2_north(1,8), East => east_2_west(1,8) , Grant => arb_grant(1,8));

--Second Row

Arbiter_2_1: Arbiter
c_bar_P(14),
    PORT MAP (Req => arb_req(9), North => south_2_north(1,1), West => east_2_west(9,8), Mask =>
              South => south_2_north(2,1), East => east_2_west(2,1) , Grant => arb_grant(2,1));

Arbiter_2_2: Arbiter
c_bar_P(13),
    PORT MAP (Req => arb_req(10), North => south_2_north(1,2), West => east_2_west(2,1), Mask =>

```

```

        South => south_2_north(2,2), East => east_2_west(2,2) , Grant => arb_grant(2,2));

Arbiter_2_3: Arbiter
        PORT MAP (Req => arb_req(11), North => south_2_north(1,3), West => east_2_west(2,2), Mask =>
c_bar_P(12),
                South => south_2_north(2,3), East => east_2_west(2,3) , Grant => arb_grant(2,3));

Arbiter_2_4: Arbiter
        PORT MAP (Req => arb_req(12), North => south_2_north(1,4), West => east_2_west(2,3), Mask =>
c_bar_P(11),
                South => south_2_north(2,4), East => east_2_west(2,4) , Grant => arb_grant(2,4));

Arbiter_2_5: Arbiter
        PORT MAP (Req => arb_req(13), North => south_2_north(1,5), West => east_2_west(2,4), Mask =>
c_bar_P(10),
                South => south_2_north(2,5), East => east_2_west(2,5) , Grant => arb_grant(2,5));

Arbiter_2_6: Arbiter
        PORT MAP (Req => arb_req(14), North => south_2_north(1,6), West => east_2_west(2,5), Mask =>
c_bar_P(9),
                South => south_2_north(2,6), East => east_2_west(2,6) , Grant => arb_grant(2,6));

Arbiter_2_7: Arbiter
        PORT MAP (Req => arb_req(15), North => south_2_north(1,7), West => east_2_west(2,6), Mask =>
c_bar_P(8),
                South => south_2_north(2,7), East => east_2_west(2,7) , Grant => arb_grant(2,7));

Arbiter_2_8: Arbiter
        PORT MAP (Req => arb_req(16), North => south_2_north(1,8), West => east_2_west(2,7), Mask =>
c_bar_P(7),
                South => south_2_north(2,8), East => east_2_west(2,8) , Grant => arb_grant(2,8));

--Third Row
Arbiter_3_1: Arbiter
        PORT MAP (Req => arb_req(17), North => south_2_north(2,1), West => east_2_west(10,8), Mask =>
c_bar_P(13),
                South => south_2_north(3,1), East => east_2_west(3,1) , Grant => arb_grant(3,1));

Arbiter_3_2: Arbiter
        PORT MAP (Req => arb_req(18), North => south_2_north(2,2), West => east_2_west(3,1), Mask =>
c_bar_P(12),
                South => south_2_north(3,2), East => east_2_west(3,2) , Grant => arb_grant(3,2));

Arbiter_3_3: Arbiter
        PORT MAP (Req => arb_req(19), North => south_2_north(2,3), West => east_2_west(3,2), Mask =>
c_bar_P(11),
                South => south_2_north(3,3), East => east_2_west(3,3) , Grant => arb_grant(3,3));

```

```

Arbiter_3_4: Arbiter
    PORT MAP (Req => arb_req(20), North => south_2_north(2,4), West => east_2_west(3,3), Mask =>
c_bar_P(10),      South => south_2_north(3,4), East => east_2_west(3,4) , Grant => arb_grant(3,4));

Arbiter_3_5: Arbiter
    PORT MAP (Req => arb_req(21), North => south_2_north(2,5), West => east_2_west(3,4), Mask =>
c_bar_P(9),      South => south_2_north(3,5), East => east_2_west(3,5) , Grant => arb_grant(3,5));

Arbiter_3_6: Arbiter
    PORT MAP (Req => arb_req(22), North => south_2_north(2,6), West => east_2_west(3,5), Mask =>
c_bar_P(8),      South => south_2_north(3,6), East => east_2_west(3,6) , Grant => arb_grant(3,6));

Arbiter_3_7: Arbiter
    PORT MAP (Req => arb_req(23), North => south_2_north(2,7), West => east_2_west(3,6), Mask =>
c_bar_P(7),      South => south_2_north(3,7), East => east_2_west(3,7) , Grant => arb_grant(3,7));

Arbiter_3_8: Arbiter
    PORT MAP (Req => arb_req(24), North => south_2_north(2,8), West => east_2_west(3,7), Mask =>
c_bar_P(6),      South => south_2_north(3,8), East => east_2_west(3,8) , Grant => arb_grant(3,8));

--Forth Row

Arbiter_4_1: Arbiter
    PORT MAP (Req => arb_req(25), North => south_2_north(3,1), West => east_2_west(11,8), Mask =>
c_bar_P(12),     South => south_2_north(4,1), East => east_2_west(4,1) , Grant => arb_grant(4,1));

Arbiter_4_2: Arbiter
    PORT MAP (Req => arb_req(26), North => south_2_north(3,2), West => east_2_west(4,1), Mask =>
c_bar_P(11),     South => south_2_north(4,2), East => east_2_west(4,2) , Grant => arb_grant(4,2));

Arbiter_4_3: Arbiter
    PORT MAP (Req => arb_req(27), North => south_2_north(3,3), West => east_2_west(4,2), Mask =>
c_bar_P(10),     South => south_2_north(4,3), East => east_2_west(4,3) , Grant => arb_grant(4,3));

Arbiter_4_4: Arbiter
    PORT MAP (Req => arb_req(28), North => south_2_north(3,4), West => east_2_west(4,3), Mask =>
c_bar_P(9),      South => south_2_north(4,4), East => east_2_west(4,4) , Grant => arb_grant(4,4));

Arbiter_4_5: Arbiter

```

```

PORT MAP (Req => arb_req(29), North => south_2_north(3,5), West => east_2_west(4,4), Mask =>
c_bar_P(8),          South => south_2_north(4,5), East => east_2_west(4,5) , Grant => arb_grant(4,5));

Arbiter_4_6: Arbiter
PORT MAP (Req => arb_req(30), North => south_2_north(3,6), West => east_2_west(4,5), Mask =>
c_bar_P(7),          South => south_2_north(4,6), East => east_2_west(4,6) , Grant => arb_grant(4,6));

Arbiter_4_7: Arbiter
PORT MAP (Req => arb_req(31), North => south_2_north(3,7), West => east_2_west(4,6), Mask =>
c_bar_P(6),          South => south_2_north(4,7), East => east_2_west(4,7) , Grant => arb_grant(4,7));

Arbiter_4_8: Arbiter
PORT MAP (Req => arb_req(32), North => south_2_north(3,8), West => east_2_west(4,7), Mask =>
c_bar_P(5),          South => south_2_north(4,8), East => east_2_west(4,8) , Grant => arb_grant(4,8));

--Fifth Row
Arbiter_5_1: Arbiter
PORT MAP (Req => arb_req(33), North => south_2_north(4,1), West => east_2_west(12,8), Mask =>
c_bar_P(11),         South => south_2_north(5,1), East => east_2_west(5,1) , Grant => arb_grant(5,1));

Arbiter_5_2: Arbiter
PORT MAP (Req => arb_req(34), North => south_2_north(4,2), West => east_2_west(5,1), Mask =>
c_bar_P(10),         South => south_2_north(5,2), East => east_2_west(5,2) , Grant => arb_grant(5,2));

Arbiter_5_3: Arbiter
PORT MAP (Req => arb_req(35), North => south_2_north(4,3), West => east_2_west(5,2), Mask =>
c_bar_P(9),          South => south_2_north(5,3), East => east_2_west(5,3) , Grant => arb_grant(5,3));

Arbiter_5_4: Arbiter
PORT MAP (Req => arb_req(36), North => south_2_north(4,4), West => east_2_west(5,3), Mask =>
c_bar_P(8),          South => south_2_north(5,4), East => east_2_west(5,4) , Grant => arb_grant(5,4));

Arbiter_5_5: Arbiter
PORT MAP (Req => arb_req(37), North => south_2_north(4,5), West => east_2_west(5,4), Mask =>
c_bar_P(7),          South => south_2_north(5,5), East => east_2_west(5,5) , Grant => arb_grant(5,5));

Arbiter_5_6: Arbiter
PORT MAP (Req => arb_req(38), North => south_2_north(4,6), West => east_2_west(5,5), Mask
=>c_bar_P(6),

```

```

        South => south_2_north(5,6), East => east_2_west(5,6) , Grant => arb_grant(5,6));

Arbiter_5_7: Arbiter
        PORT MAP (Req => arb_req(39), North => south_2_north(4,7), West => east_2_west(5,6), Mask =>
c_bar_P(5),
        South => south_2_north(5,7), East => east_2_west(5,7) , Grant => arb_grant(5,7));

Arbiter_5_8: Arbiter
        PORT MAP (Req => arb_req(40), North => south_2_north(4,8), West => east_2_west(5,7), Mask =>
c_bar_P(4),
        South => south_2_north(5,8), East => east_2_west(5,8) , Grant => arb_grant(5,8));

--Sixth Row
Arbiter_6_1: Arbiter
        PORT MAP (Req => arb_req(41), North => south_2_north(5,1), West => east_2_west(13,8), Mask =>
c_bar_P(10),
        South => south_2_north(6,1), East => east_2_west(6,1) , Grant => arb_grant(6,1));

Arbiter_6_2: Arbiter
        PORT MAP (Req => arb_req(42), North => south_2_north(5,2), West => east_2_west(6,1), Mask =>
c_bar_P(9),
        South => south_2_north(6,2), East => east_2_west(6,2) , Grant => arb_grant(6,2));

Arbiter_6_3: Arbiter
        PORT MAP (Req => arb_req(43), North => south_2_north(5,3), West => east_2_west(6,2), Mask =>
c_bar_P(8),
        South => south_2_north(6,3), East => east_2_west(6,3) , Grant => arb_grant(6,3));

Arbiter_6_4: Arbiter
        PORT MAP (Req => arb_req(44), North => south_2_north(5,4), West => east_2_west(6,3), Mask =>
c_bar_P(7),
        South => south_2_north(6,4), East => east_2_west(6,4) , Grant => arb_grant(6,4));

Arbiter_6_5: Arbiter
        PORT MAP (Req => arb_req(45), North => south_2_north(5,5), West => east_2_west(6,4), Mask =>
c_bar_P(6),
        South => south_2_north(6,5), East => east_2_west(6,5) , Grant => arb_grant(6,5));

Arbiter_6_6: Arbiter
        PORT MAP (Req => arb_req(46), North => south_2_north(5,6), West => east_2_west(6,5), Mask =>
c_bar_P(5),
        South => south_2_north(6,6), East => east_2_west(6,6) , Grant => arb_grant(6,6));

Arbiter_6_7: Arbiter
        PORT MAP (Req => arb_req(47), North => south_2_north(5,7), West => east_2_west(6,6), Mask =>
c_bar_P(4),
        South => south_2_north(6,7), East => east_2_west(6,7) , Grant => arb_grant(6,7));

```

```

Arbiter_6_8: Arbiter
    PORT MAP (Req => arb_req(48), North => south_2_north(5,8), West => east_2_west(6,7), Mask =>
c_bar_P(3),
    South => south_2_north(6,8), East => east_2_west(6,8) , Grant => arb_grant(6,8));

--Seventh Row
Arbiter_7_1: Arbiter
    PORT MAP (Req => arb_req(49), North => south_2_north(6,1), West => east_2_west(14,8), Mask =>
c_bar_P(9),
    South => south_2_north(7,1), East => east_2_west(7,1) , Grant => arb_grant(7,1));

Arbiter_7_2: Arbiter
    PORT MAP (Req => arb_req(50), North => south_2_north(6,2), West => east_2_west(7,1), Mask =>
c_bar_P(8),
    South => south_2_north(7,2), East => east_2_west(7,2) , Grant => arb_grant(7,2));

Arbiter_7_3: Arbiter
    PORT MAP (Req => arb_req(51), North => south_2_north(6,3), West => east_2_west(7,2), Mask =>
c_bar_P(7),
    South => south_2_north(7,3), East => east_2_west(7,3) , Grant => arb_grant(7,3));

Arbiter_7_4: Arbiter
    PORT MAP (Req => arb_req(52), North => south_2_north(6,4), West => east_2_west(7,3), Mask =>
c_bar_P(6),
    South => south_2_north(7,4), East => east_2_west(7,4) , Grant => arb_grant(7,4));

Arbiter_7_5: Arbiter
    PORT MAP (Req => arb_req(53), North => south_2_north(6,5), West => east_2_west(7,4), Mask =>
c_bar_P(5),
    South => south_2_north(7,5), East => east_2_west(7,5) , Grant => arb_grant(7,5));

Arbiter_7_6: Arbiter
    PORT MAP (Req => arb_req(54), North => south_2_north(6,6), West => east_2_west(7,5), Mask =>
c_bar_P(4),
    South => south_2_north(7,6), East => east_2_west(7,6) , Grant => arb_grant(7,6));

Arbiter_7_7: Arbiter
    PORT MAP (Req => arb_req(55), North => south_2_north(6,7), West => east_2_west(7,6), Mask =>
c_bar_P(3),
    South => south_2_north(7,7), East => east_2_west(7,7) , Grant => arb_grant(7,7));

Arbiter_7_8: Arbiter
    PORT MAP (Req => arb_req(56), North => south_2_north(6,8), West => east_2_west(7,7), Mask =>
c_bar_P(2),
    South => south_2_north(7,8), East => east_2_west(7,8) , Grant => arb_grant(7,8));

```

```

--Eighth Row
Arbiter_8_1: Arbiter
    PORT MAP (Req => arb_req(57), North => south_2_north(7,1), West => east_2_west(15,8), Mask =>
c_bar_P(8),
    South => south_2_north(8,1), East => east_2_west(8,1) , Grant => arb_grant(8,1));

Arbiter_8_2: Arbiter
    PORT MAP (Req => arb_req(58), North => south_2_north(7,2), West => east_2_west(8,1), Mask =>
c_bar_P(7),
    South => south_2_north(8,2), East => east_2_west(8,2) , Grant => arb_grant(8,2));

Arbiter_8_3: Arbiter
    PORT MAP (Req => arb_req(59), North => south_2_north(7,3), West => east_2_west(8,2), Mask =>
c_bar_P(6),
    South => south_2_north(8,3), East => east_2_west(8,3) , Grant => arb_grant(8,3));

Arbiter_8_4: Arbiter
    PORT MAP (Req => arb_req(60), North => south_2_north(7,4), West => east_2_west(8,3), Mask =>
c_bar_P(5),
    South => south_2_north(8,4), East => east_2_west(8,4) , Grant => arb_grant(8,4));

Arbiter_8_5: Arbiter
    PORT MAP (Req => arb_req(61), North => south_2_north(7,5), West => east_2_west(8,4), Mask =>
c_bar_P(4),
    South => south_2_north(8,5), East => east_2_west(8,5) , Grant => arb_grant(8,5));

Arbiter_8_6: Arbiter
    PORT MAP (Req => arb_req(62), North => south_2_north(7,6), West => east_2_west(8,5), Mask =>
c_bar_P(3),
    South => south_2_north(8,6), East => east_2_west(8,6) , Grant => arb_grant(8,6));

Arbiter_8_7: Arbiter
    PORT MAP (Req => arb_req(63), North => south_2_north(7,7), West => east_2_west(8,6), Mask =>
c_bar_P(2),
    South => south_2_north(8,7), East => east_2_west(8,7) , Grant => arb_grant(8,7));

Arbiter_8_8: Arbiter
    PORT MAP (Req => arb_req(64), North => south_2_north(7,8), West => east_2_west(8,7), Mask =>
c_bar_P(1),
    South => south_2_north(8,8), East => east_2_west(8,8) , Grant => arb_grant(8,8));

--Ninth Row
Arbiter_9_1: Arbiter
    PORT MAP (Req => arb_req(1), North => south_2_north(8,1), West => east_2_west(1,8), Mask =>
c_bar_P(7),
    South => south_2_north(9,1), East => east_2_west(9,1) , Grant => arb_grant(9,1));

Arbiter_9_2: Arbiter

```

c_bar_P(6), PORT MAP (Req => arb_req(2), North => south_2_north(8,2), West => east_2_west(9,1), Mask =>
South => south_2_north(9,2), East => east_2_west(9,2) , Grant => arb_grant(9,2));

Arbiter_9_3: Arbiter

c_bar_P(5), PORT MAP (Req => arb_req(3), North => south_2_north(8,3), West => east_2_west(9,2), Mask =>
South => south_2_north(9,3), East => east_2_west(9,3) , Grant => arb_grant(9,3));

Arbiter_9_4: Arbiter

c_bar_P(4), PORT MAP (Req => arb_req(4), North => south_2_north(8,4), West => east_2_west(9,3), Mask =>
South => south_2_north(9,4), East => east_2_west(9,4) , Grant => arb_grant(9,4));

Arbiter_9_5: Arbiter

c_bar_P(3), PORT MAP (Req => arb_req(5), North => south_2_north(8,5), West => east_2_west(9,4), Mask =>
South => south_2_north(9,5), East => east_2_west(9,5) , Grant => arb_grant(9,5));

Arbiter_9_6: Arbiter

c_bar_P(2), PORT MAP (Req => arb_req(6), North => south_2_north(8,6), West => east_2_west(9,5), Mask =>
South => south_2_north(9,6), East => east_2_west(9,6) , Grant => arb_grant(9,6));

Arbiter_9_7: Arbiter

c_bar_P(1), PORT MAP (Req => arb_req(7), North => south_2_north(8,7), West => east_2_west(9,6), Mask =>
South => south_2_north(9,7), East => east_2_west(9,7) , Grant => arb_grant(9,7));

Arbiter_9_8: Arbiter

PORT MAP (Req => arb_req(16), North => High, West => High, Mask => c_bar_P(15),
South => south_2_north(9,8), East => east_2_west(9,8) , Grant => arb_grant(9,8));

--Tenth Row

Arbiter_10_1: Arbiter

c_bar_P(6), PORT MAP (Req => arb_req(9), North => south_2_north(9,1), West => east_2_west(2,8), Mask =>
South => south_2_north(10,1), East => east_2_west(10,1) , Grant => arb_grant(10,1));

Arbiter_10_2: Arbiter

c_bar_P(5), PORT MAP (Req => arb_req(10), North => south_2_north(9,2), West => east_2_west(10,1), Mask =>
South => south_2_north(10,2), East => east_2_west(10,2) , Grant => arb_grant(10,2));

Arbiter_10_3: Arbiter

c_bar_P(4), PORT MAP (Req => arb_req(11), North => south_2_north(9,3), West => east_2_west(10,2), Mask =>
South => south_2_north(10,3), East => east_2_west(10,3) , Grant => arb_grant(10,3));

```

Arbiter_10_4: Arbiter
    PORT MAP (Req => arb_req(12), North => south_2_north(9,4), West => east_2_west(10,3), Mask =>
c_bar_P(3),
              South => south_2_north(10,4), East => east_2_west(10,4) , Grant => arb_grant(10,4));

Arbiter_10_5: Arbiter
    PORT MAP (Req => arb_req(13), North => south_2_north(9,5), West => east_2_west(10,4), Mask =>
c_bar_P(2),
              South => south_2_north(10,5), East => east_2_west(10,5) , Grant => arb_grant(10,5));

Arbiter_10_6: Arbiter
    PORT MAP (Req => arb_req(14), North => south_2_north(9,6), West => east_2_west(10,5), Mask =>
c_bar_P(1),
              South => south_2_north(10,6), East => east_2_west(10,6) , Grant => arb_grant(10,6));

Arbiter_10_7: Arbiter
    PORT MAP (Req => arb_req(23), North => High, West => High, Mask => c_bar_P(15),
              South => south_2_north(10,7), East => east_2_west(10,7) , Grant => arb_grant(10,7));

Arbiter_10_8: Arbiter
    PORT MAP (Req => arb_req(24), North => south_2_north(9,8), West => east_2_west(10,7), Mask =>
c_bar_P(14),
              South => south_2_north(10,8), East => east_2_west(10,8) , Grant => arb_grant(10,8));

--Eleventh Row
Arbiter_11_1: Arbiter
    PORT MAP (Req => arb_req(17), North => south_2_north(10,1), West => east_2_west(3,8), Mask =>
c_bar_P(5),
              South => south_2_north(11,1), East => east_2_west(11,1) , Grant => arb_grant(11,1));

Arbiter_11_2: Arbiter
    PORT MAP (Req => arb_req(18), North => south_2_north(10,2), West => east_2_west(11,1), Mask =>
c_bar_P(4),
              South => south_2_north(11,2), East => east_2_west(11,2) , Grant => arb_grant(11,2));

Arbiter_11_3: Arbiter
    PORT MAP (Req => arb_req(19), North => south_2_north(10,3), West => east_2_west(11,2), Mask =>
c_bar_P(3),
              South => south_2_north(11,3), East => east_2_west(11,3) , Grant => arb_grant(11,3));

Arbiter_11_4: Arbiter
    PORT MAP (Req => arb_req(20), North => south_2_north(10,4), West => east_2_west(11,3), Mask =>
c_bar_P(2),
              South => south_2_north(11,4), East => east_2_west(11,4) , Grant => arb_grant(11,4));

Arbiter_11_5: Arbiter
    PORT MAP (Req => arb_req(21), North => south_2_north(10,5), West => east_2_west(11,4), Mask =>
c_bar_P(1),
              South => south_2_north(11,5), East => east_2_west(11,5) , Grant => arb_grant(11,5));

```

```

Arbiter_11_6: Arbiter
    PORT MAP (Req => arb_req(30), North => High, West => High, Mask => c_bar_P(15),
              South => south_2_north(11,6), East => east_2_west(11,6) , Grant => arb_grant(11,6));

Arbiter_11_7: Arbiter
    PORT MAP (Req => arb_req(31), North => south_2_north(10,7), West => east_2_west(11,6), Mask =>
c_bar_P(14),
              South => south_2_north(11,7), East => east_2_west(11,7) , Grant => arb_grant(11,7));

Arbiter_11_8: Arbiter
    PORT MAP (Req => arb_req(32), North => south_2_north(10,8), West => east_2_west(11,7), Mask =>
c_bar_P(13),
              South => south_2_north(11,8), East => east_2_west(11,8) , Grant => arb_grant(11,8));

--Twelvth Row

Arbiter_12_1: Arbiter
    PORT MAP (Req => arb_req(25), North => south_2_north(11,1), West => east_2_west(4,8), Mask =>
c_bar_P(4),
              South => south_2_north(12,1), East => east_2_west(12,1) , Grant => arb_grant(12,1));

Arbiter_12_2: Arbiter
    PORT MAP (Req => arb_req(26), North => south_2_north(11,2), West => east_2_west(12,1), Mask =>
c_bar_P(3),
              South => south_2_north(12,2), East => east_2_west(12,2) , Grant => arb_grant(12,2));

Arbiter_12_3: Arbiter
    PORT MAP (Req => arb_req(27), North => south_2_north(11,3), West => east_2_west(12,2), Mask =>
c_bar_P(2),
              South => south_2_north(12,3), East => east_2_west(12,3) , Grant => arb_grant(12,3));

Arbiter_12_4: Arbiter
    PORT MAP (Req => arb_req(28), North => south_2_north(11,4), West => east_2_west(12,3), Mask =>
c_bar_P(1),
              South => south_2_north(12,4), East => east_2_west(12,4) , Grant => arb_grant(12,4));

Arbiter_12_5: Arbiter
    PORT MAP (Req => arb_req(37), North => High, West => High, Mask => c_bar_P(15),
              South => south_2_north(12,5), East => east_2_west(12,5) , Grant => arb_grant(12,5));

Arbiter_12_6: Arbiter
    PORT MAP (Req => arb_req(38), North => south_2_north(11,6), West => east_2_west(12,5), Mask =>
c_bar_P(14),
              South => south_2_north(12,6), East => east_2_west(12,6) , Grant => arb_grant(12,6));

Arbiter_12_7: Arbiter
    PORT MAP (Req => arb_req(39), North => south_2_north(11,7), West => east_2_west(12,6), Mask =>
c_bar_P(13),

```

```

        South => south_2_north(12,7), East => east_2_west(12,7) , Grant => arb_grant(12,7));

Arbiter_12_8: Arbiter
        PORT MAP (Req => arb_req(40), North => south_2_north(11,8), West => east_2_west(12,7), Mask =>
c_bar_P(12),
        South => south_2_north(12,8), East => east_2_west(12,8) , Grant => arb_grant(12,8));

--Thirteenth Row

Arbiter_13_1: Arbiter
        PORT MAP (Req => arb_req(33), North => south_2_north(12,1), West => east_2_west(5,8), Mask =>
c_bar_P(3),
        South => south_2_north(13,1), East => east_2_west(13,1) , Grant => arb_grant(13,1));

Arbiter_13_2: Arbiter
        PORT MAP (Req => arb_req(34), North => south_2_north(12,2), West => east_2_west(13,1), Mask =>
c_bar_P(2),
        South => south_2_north(13,2), East => east_2_west(13,2) , Grant => arb_grant(13,2));

Arbiter_13_3: Arbiter
        PORT MAP (Req => arb_req(35), North => south_2_north(12,3), West => east_2_west(13,2), Mask =>
c_bar_P(1),
        South => south_2_north(13,3), East => east_2_west(13,3) , Grant => arb_grant(13,3));

Arbiter_13_4: Arbiter
        PORT MAP (Req => arb_req(44), North => High, West => High, Mask => c_bar_P(15),
        South => south_2_north(13,4), East => east_2_west(13,4) , Grant => arb_grant(13,4));

Arbiter_13_5: Arbiter
        PORT MAP (Req => arb_req(45), North => south_2_north(12,5), West => east_2_west(13,4), Mask =>
c_bar_P(14),
        South => south_2_north(13,5), East => east_2_west(13,5) , Grant => arb_grant(13,5));

Arbiter_13_6: Arbiter
        PORT MAP (Req => arb_req(46), North => south_2_north(12,6), West => east_2_west(13,5), Mask =>
c_bar_P(13),
        South => south_2_north(13,6), East => east_2_west(13,6) , Grant => arb_grant(13,6));

Arbiter_13_7: Arbiter
        PORT MAP (Req => arb_req(47), North => south_2_north(12,7), West => east_2_west(13,6), Mask =>
c_bar_P(12),
        South => south_2_north(13,7), East => east_2_west(13,7) , Grant => arb_grant(13,7));

Arbiter_13_8: Arbiter
        PORT MAP (Req => arb_req(48), North => south_2_north(12,8), West => east_2_west(13,7), Mask =>
c_bar_P(11),
        South => south_2_north(13,8), East => east_2_west(13,8) , Grant => arb_grant(13,8));

```

```

--Fourteenth Row
Arbiter_14_1: Arbiter
    PORT MAP (Req => arb_req(41), North => south_2_north(13,1), West => east_2_west(6,8), Mask =>
c_bar_P(2),
    South => south_2_north(14,1), East => east_2_west(14,1) , Grant => arb_grant(14,1));

Arbiter_14_2: Arbiter
    PORT MAP (Req => arb_req(42), North => south_2_north(13,2), West => east_2_west(14,1), Mask =>
c_bar_P(1),
    South => south_2_north(14,2), East => east_2_west(14,2), Grant => arb_grant(14,2));

Arbiter_14_3: Arbiter
    PORT MAP (Req => arb_req(51), North => High, West => High, Mask => c_bar_P(15),
    South => south_2_north(14,3), East => east_2_west(14,3) , Grant => arb_grant(14,3));

Arbiter_14_4: Arbiter
    PORT MAP (Req => arb_req(52), North => south_2_north(13,4), West => east_2_west(14,3), Mask =>
c_bar_P(14),
    South => south_2_north(14,4), East => east_2_west(14,4) , Grant => arb_grant(14,4));

Arbiter_14_5: Arbiter
    PORT MAP (Req => arb_req(53), North => south_2_north(13,5), West => east_2_west(14,4), Mask =>
c_bar_P(13),
    South => south_2_north(14,5), East => east_2_west(14,5) , Grant => arb_grant(14,5));

Arbiter_14_6: Arbiter
    PORT MAP (Req => arb_req(54), North => south_2_north(13,6), West => east_2_west(14,5), Mask =>
c_bar_P(12),
    South => south_2_north(14,6), East => east_2_west(14,6) , Grant => arb_grant(14,6));

Arbiter_14_7: Arbiter
    PORT MAP (Req => arb_req(55), North => south_2_north(13,7), West => east_2_west(14,6), Mask =>
c_bar_P(11),
    South => south_2_north(14,7), East => east_2_west(14,7) , Grant => arb_grant(14,7));

Arbiter_14_8: Arbiter
    PORT MAP (Req => arb_req(56), North => south_2_north(13,8), West => east_2_west(14,7), Mask =>
c_bar_P(10),
    South => south_2_north(14,8), East => east_2_west(14,8) , Grant => arb_grant(14,8));

--Fifteenth Row
Arbiter_15_1: Arbiter
    PORT MAP (Req => arb_req(49), North => south_2_north(14,1), West => east_2_west(7,8), Mask =>
c_bar_P(1),
    South => south_2_north(15,1), East => east_2_west(15,1) , Grant => arb_grant(15,1));

Arbiter_15_2: Arbiter

```

PORT MAP (Req => arb_req(58), North => High, West => High, Mask => c_bar_P(15),
South => south_2_north(15,2), East => east_2_west(15,2) , Grant => arb_grant(15,2));

Arbiter_15_3: Arbiter

PORT MAP (Req => arb_req(59), North => south_2_north(14,3), West => east_2_west(15,2), Mask =>
c_bar_P(14),
South => south_2_north(15,3), East => east_2_west(15,3) , Grant => arb_grant(15,3));

Arbiter_15_4: Arbiter

PORT MAP (Req => arb_req(60), North => south_2_north(14,4), West => east_2_west(15,3), Mask =>
c_bar_P(13),
South => south_2_north(15,4), East => east_2_west(15,4) , Grant => arb_grant(15,4));

Arbiter_15_5: Arbiter

PORT MAP (Req => arb_req(61), North => south_2_north(14,5), West => east_2_west(15,4), Mask =>
c_bar_P(12),
South => south_2_north(15,5), East => east_2_west(15,5) , Grant => arb_grant(15,5));

Arbiter_15_6: Arbiter

PORT MAP (Req => arb_req(62), North => south_2_north(14,6), West => east_2_west(15,5), Mask =>
c_bar_P(11),
South => south_2_north(15,6), East => east_2_west(15,6) , Grant => arb_grant(15,6));

Arbiter_15_7: Arbiter

PORT MAP (Req => arb_req(63), North => south_2_north(14,7), West => east_2_west(15,6), Mask =>
c_bar_P(10),
South => south_2_north(15,7), East => east_2_west(15,7) , Grant => arb_grant(15,7));

Arbiter_15_8: Arbiter

PORT MAP (Req => arb_req(64), North => south_2_north(14,8), West => east_2_west(15,7), Mask =>
c_bar_P(9),
South => south_2_north(15,8), East => east_2_west(15,8) , Grant => arb_grant(15,8));

END behaviour;

Appendix D.3: The VHDL code for the Crossbar fabric.

-- "fabric.vhd"
-- This is a VHDL behavioral description of a switch fabric. This fabric is like a crossbar.
-- There are 6 inputs and 6 outputs for this fabric. Each input port and each output port are
-- 8 bits long. This is like having 6 instances of an 8x8 crossbar. For each input i, the bit
-- lines are switched according to the fabric configuration, and sent to output i.
-- cntnl input configures the fabric.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY fabric IS
    GENERIC(
        SWITCH_SIZE    : INTEGER:= 8;  --8x8 fabric by default
        GRANT_SIZE     : INTEGER:= 64   --64 lines used to issue grants
    );
    PORT (
        input1 : IN  STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The input lines to
        connected to output lines
        input2 : IN  STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The input lines to
        connected to output lines
        input3 : IN  STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The input lines to
        connected to output lines
        input4 : IN  STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The input lines to
        connected to output lines
        input5 : IN  STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The input lines to
        connected to output lines
        input6 : IN  STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The input lines to
        connected to output lines
        cntnl  : IN  STD_LOGIC_VECTOR(GRANT_SIZE-1 DOWNT0 0);    --The grant lines are used
        to control the fabric
        output1 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The output line
        carrying data out of the fabric
        output2 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The output line
        carrying data out of the fabric
        output3 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The output line
        carrying data out of the fabric
        output4 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The output line
        carrying data out of the fabric
        output5 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The output line
        carrying data out of the fabric
        output6 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0);  --The output line
        carrying data out of the fabric
    );
END fabric;

ARCHITECTURE behav OF fabric IS

    BEGIN --architecture

        output1(0) <= ( (input1(0) AND cntnl(0)) OR (input1(1) AND cntnl(8)) OR (input1(2) AND cntnl(16)) OR
        AND cntnl(40)) OR
        (input1(6) AND cntnl(48)) OR (input1(7) AND cntnl(56)) );

        output1(1) <= ( (input1(0) AND cntnl(1)) OR (input1(1) AND cntnl(9)) OR (input1(2) AND cntnl(17)) OR
```

AND cntrl(41)) OR
 (input1(3) AND cntrl(25)) OR (input1(4) AND cntrl(33)) OR (input1(5)
 (input1(6) AND cntrl(49)) OR (input1(7) AND cntrl(57)));
 output1(2) <= ((input1(0) AND cntrl(2)) OR (input1(1) AND cntrl(10)) OR (input1(2) AND cntrl(18)) OR
 (input1(3) AND cntrl(26)) OR (input1(4) AND cntrl(34)) OR (input1(5)
 AND cntrl(42)) OR
 (input1(6) AND cntrl(50)) OR (input1(7) AND cntrl(58)));
 output1(3) <= ((input1(0) AND cntrl(3)) OR (input1(1) AND cntrl(11)) OR (input1(2) AND cntrl(19)) OR
 (input1(3) AND cntrl(27)) OR (input1(4) AND cntrl(35)) OR (input1(5)
 AND cntrl(43)) OR
 (input1(6) AND cntrl(51)) OR (input1(7) AND cntrl(59)));
 output1(4) <= ((input1(0) AND cntrl(4)) OR (input1(1) AND cntrl(12)) OR (input1(2) AND cntrl(20)) OR
 (input1(3) AND cntrl(28)) OR (input1(4) AND cntrl(36)) OR (input1(5)
 AND cntrl(44)) OR
 (input1(6) AND cntrl(52)) OR (input1(7) AND cntrl(60)));
 output1(5) <= ((input1(0) AND cntrl(5)) OR (input1(1) AND cntrl(13)) OR (input1(2) AND cntrl(21)) OR
 (input1(3) AND cntrl(29)) OR (input1(4) AND cntrl(37)) OR (input1(5)
 AND cntrl(45)) OR
 (input1(6) AND cntrl(53)) OR (input1(7) AND cntrl(61)));
 output1(6) <= ((input1(0) AND cntrl(6)) OR (input1(1) AND cntrl(14)) OR (input1(2) AND cntrl(22)) OR
 (input1(3) AND cntrl(30)) OR (input1(4) AND cntrl(38)) OR (input1(5)
 AND cntrl(46)) OR
 (input1(6) AND cntrl(54)) OR (input1(7) AND cntrl(62)));
 output1(7) <= ((input1(0) AND cntrl(7)) OR (input1(1) AND cntrl(15)) OR (input1(2) AND cntrl(23)) OR
 (input1(3) AND cntrl(31)) OR (input1(4) AND cntrl(39)) OR (input1(5)
 AND cntrl(47)) OR
 (input1(6) AND cntrl(55)) OR (input1(7) AND cntrl(63)));

 output2(0) <= ((input2(0) AND cntrl(0)) OR (input2(1) AND cntrl(8)) OR (input2(2) AND cntrl(16)) OR
 (input2(3) AND cntrl(24)) OR (input2(4) AND cntrl(32)) OR (input2(5)
 AND cntrl(40)) OR
 (input2(6) AND cntrl(48)) OR (input2(7) AND cntrl(56)));
 output2(1) <= ((input2(0) AND cntrl(1)) OR (input2(1) AND cntrl(9)) OR (input2(2) AND cntrl(17)) OR
 (input2(3) AND cntrl(25)) OR (input2(4) AND cntrl(33)) OR (input2(5)
 AND cntrl(41)) OR
 (input2(6) AND cntrl(49)) OR (input2(7) AND cntrl(57)));
 output2(2) <= ((input2(0) AND cntrl(2)) OR (input2(1) AND cntrl(10)) OR (input2(2) AND cntrl(18)) OR
 (input2(3) AND cntrl(26)) OR (input2(4) AND cntrl(34)) OR (input2(5)
 AND cntrl(42)) OR
 (input2(6) AND cntrl(50)) OR (input2(7) AND cntrl(58)));
 output2(3) <= ((input2(0) AND cntrl(3)) OR (input2(1) AND cntrl(11)) OR (input2(2) AND cntrl(19)) OR
 (input2(3) AND cntrl(27)) OR (input2(4) AND cntrl(35)) OR (input2(5)
 AND cntrl(43)) OR
 (input2(6) AND cntrl(51)) OR (input2(7) AND cntrl(59)));
 output2(4) <= ((input2(0) AND cntrl(4)) OR (input2(1) AND cntrl(12)) OR (input2(2) AND cntrl(20)) OR
 (input2(3) AND cntrl(28)) OR (input2(4) AND cntrl(36)) OR (input2(5)
 AND cntrl(44)) OR
 (input2(6) AND cntrl(52)) OR (input2(7) AND cntrl(60)));
 output2(5) <= ((input2(0) AND cntrl(5)) OR (input2(1) AND cntrl(13)) OR (input2(2) AND cntrl(21)) OR
 (input2(3) AND cntrl(29)) OR (input2(4) AND cntrl(37)) OR (input2(5)
 AND cntrl(45)) OR
 (input2(6) AND cntrl(53)) OR (input2(7) AND cntrl(61)));
 output2(6) <= ((input2(0) AND cntrl(6)) OR (input2(1) AND cntrl(14)) OR (input2(2) AND cntrl(22)) OR
 (input2(3) AND cntrl(30)) OR (input2(4) AND cntrl(38)) OR (input2(5)
 AND cntrl(46)) OR
 (input2(6) AND cntrl(54)) OR (input2(7) AND cntrl(62)));

output2(7) <= ((input2(0) AND cntrl(7)) OR (input2(1) AND cntrl(15)) OR (input2(2) AND cntrl(23)) OR
 AND cntrl(47)) OR
 (input2(3) AND cntrl(31)) OR (input2(4) AND cntrl(39)) OR (input2(5)
 (input2(6) AND cntrl(55)) OR (input2(7) AND cntrl(63)));

output3(0) <= ((input3(0) AND cntrl(0)) OR (input3(1) AND cntrl(8)) OR (input3(2) AND cntrl(16)) OR
 AND cntrl(40)) OR
 (input3(3) AND cntrl(24)) OR (input3(4) AND cntrl(32)) OR (input3(5)
 (input3(6) AND cntrl(48)) OR (input3(7) AND cntrl(56)));

output3(1) <= ((input3(0) AND cntrl(1)) OR (input3(1) AND cntrl(9)) OR (input3(2) AND cntrl(17)) OR
 AND cntrl(41)) OR
 (input3(3) AND cntrl(25)) OR (input3(4) AND cntrl(33)) OR (input3(5)
 (input3(6) AND cntrl(49)) OR (input3(7) AND cntrl(57)));

output3(2) <= ((input3(0) AND cntrl(2)) OR (input3(1) AND cntrl(10)) OR (input3(2) AND cntrl(18)) OR
 AND cntrl(42)) OR
 (input3(3) AND cntrl(26)) OR (input3(4) AND cntrl(34)) OR (input3(5)
 (input3(6) AND cntrl(50)) OR (input3(7) AND cntrl(58)));

output3(3) <= ((input3(0) AND cntrl(3)) OR (input3(1) AND cntrl(11)) OR (input3(2) AND cntrl(19)) OR
 AND cntrl(43)) OR
 (input3(3) AND cntrl(27)) OR (input3(4) AND cntrl(35)) OR (input3(5)
 (input3(6) AND cntrl(51)) OR (input3(7) AND cntrl(59)));

output3(4) <= ((input3(0) AND cntrl(4)) OR (input3(1) AND cntrl(12)) OR (input3(2) AND cntrl(20)) OR
 AND cntrl(44)) OR
 (input3(3) AND cntrl(28)) OR (input3(4) AND cntrl(36)) OR (input3(5)
 (input3(6) AND cntrl(52)) OR (input3(7) AND cntrl(60)));

output3(5) <= ((input3(0) AND cntrl(5)) OR (input3(1) AND cntrl(13)) OR (input3(2) AND cntrl(21)) OR
 AND cntrl(45)) OR
 (input3(3) AND cntrl(29)) OR (input3(4) AND cntrl(37)) OR (input3(5)
 (input3(6) AND cntrl(53)) OR (input3(7) AND cntrl(61)));

output3(6) <= ((input3(0) AND cntrl(6)) OR (input3(1) AND cntrl(14)) OR (input3(2) AND cntrl(22)) OR
 AND cntrl(46)) OR
 (input3(3) AND cntrl(30)) OR (input3(4) AND cntrl(38)) OR (input3(5)
 (input3(6) AND cntrl(54)) OR (input3(7) AND cntrl(62)));

output3(7) <= ((input3(0) AND cntrl(7)) OR (input3(1) AND cntrl(15)) OR (input3(2) AND cntrl(23)) OR
 AND cntrl(47)) OR
 (input3(3) AND cntrl(31)) OR (input3(4) AND cntrl(39)) OR (input3(5)
 (input3(6) AND cntrl(55)) OR (input3(7) AND cntrl(63)));

output4(0) <= ((input4(0) AND cntrl(0)) OR (input4(1) AND cntrl(8)) OR (input4(2) AND cntrl(16)) OR
 AND cntrl(40)) OR
 (input4(3) AND cntrl(24)) OR (input4(4) AND cntrl(32)) OR (input4(5)
 (input4(6) AND cntrl(48)) OR (input4(7) AND cntrl(56)));

output4(1) <= ((input4(0) AND cntrl(1)) OR (input4(1) AND cntrl(9)) OR (input4(2) AND cntrl(17)) OR
 AND cntrl(41)) OR
 (input4(3) AND cntrl(25)) OR (input4(4) AND cntrl(33)) OR (input4(5)
 (input4(6) AND cntrl(49)) OR (input4(7) AND cntrl(57)));

output4(2) <= ((input4(0) AND cntrl(2)) OR (input4(1) AND cntrl(10)) OR (input4(2) AND cntrl(18)) OR
 AND cntrl(42)) OR
 (input4(3) AND cntrl(26)) OR (input4(4) AND cntrl(34)) OR (input4(5)
 (input4(6) AND cntrl(50)) OR (input4(7) AND cntrl(58)));

output4(3) <= ((input4(0) AND cntrl(3)) OR (input4(1) AND cntrl(11)) OR (input4(2) AND cntrl(19)) OR
 AND cntrl(43)) OR
 (input4(3) AND cntrl(27)) OR (input4(4) AND cntrl(35)) OR (input4(5)
 (input4(6) AND cntrl(51)) OR (input4(7) AND cntrl(59)));

```

output4(4) <= ( (input4(0) AND cntrl(4)) OR (input4(1) AND cntrl(12)) OR (input4(2) AND cntrl(20)) OR
AND cntrl(44)) OR
(input4(3) AND cntrl(28)) OR (input4(4) AND cntrl(36)) OR (input4(5)
(input4(6) AND cntrl(52)) OR (input4(7) AND cntrl(60)) );

output4(5) <= ( (input4(0) AND cntrl(5)) OR (input4(1) AND cntrl(13)) OR (input4(2) AND cntrl(21)) OR
AND cntrl(45)) OR
(input4(3) AND cntrl(29)) OR (input4(4) AND cntrl(37)) OR (input4(5)
(input4(6) AND cntrl(53)) OR (input4(7) AND cntrl(61)) );

output4(6) <= ( (input4(0) AND cntrl(6)) OR (input4(1) AND cntrl(14)) OR (input4(2) AND cntrl(22)) OR
AND cntrl(46)) OR
(input4(3) AND cntrl(30)) OR (input4(4) AND cntrl(38)) OR (input4(5)
(input4(6) AND cntrl(54)) OR (input4(7) AND cntrl(62)) );

output4(7) <= ( (input4(0) AND cntrl(7)) OR (input4(1) AND cntrl(15)) OR (input4(2) AND cntrl(23)) OR
AND cntrl(47)) OR
(input4(3) AND cntrl(31)) OR (input4(4) AND cntrl(39)) OR (input4(5)
(input4(6) AND cntrl(55)) OR (input4(7) AND cntrl(63)) );

output5(0) <= ( (input5(0) AND cntrl(0)) OR (input5(1) AND cntrl(8)) OR (input5(2) AND cntrl(16)) OR
AND cntrl(40)) OR
(input5(3) AND cntrl(24)) OR (input5(4) AND cntrl(32)) OR (input5(5)
(input5(6) AND cntrl(48)) OR (input5(7) AND cntrl(56)) );

output5(1) <= ( (input5(0) AND cntrl(1)) OR (input5(1) AND cntrl(9)) OR (input5(2) AND cntrl(17)) OR
AND cntrl(41)) OR
(input5(3) AND cntrl(25)) OR (input5(4) AND cntrl(33)) OR (input5(5)
(input5(6) AND cntrl(49)) OR (input5(7) AND cntrl(57)) );

output5(2) <= ( (input5(0) AND cntrl(2)) OR (input5(1) AND cntrl(10)) OR (input5(2) AND cntrl(18)) OR
AND cntrl(42)) OR
(input5(3) AND cntrl(26)) OR (input5(4) AND cntrl(34)) OR (input5(5)
(input5(6) AND cntrl(50)) OR (input5(7) AND cntrl(58)) );

output5(3) <= ( (input5(0) AND cntrl(3)) OR (input5(1) AND cntrl(11)) OR (input5(2) AND cntrl(19)) OR
AND cntrl(43)) OR
(input5(3) AND cntrl(27)) OR (input5(4) AND cntrl(35)) OR (input5(5)
(input5(6) AND cntrl(51)) OR (input5(7) AND cntrl(59)) );

output5(4) <= ( (input5(0) AND cntrl(4)) OR (input5(1) AND cntrl(12)) OR (input5(2) AND cntrl(20)) OR
AND cntrl(44)) OR
(input5(3) AND cntrl(28)) OR (input5(4) AND cntrl(36)) OR (input5(5)
(input5(6) AND cntrl(52)) OR (input5(7) AND cntrl(60)) );

output5(5) <= ( (input5(0) AND cntrl(5)) OR (input5(1) AND cntrl(13)) OR (input5(2) AND cntrl(21)) OR
AND cntrl(45)) OR
(input5(3) AND cntrl(29)) OR (input5(4) AND cntrl(37)) OR (input5(5)
(input5(6) AND cntrl(53)) OR (input5(7) AND cntrl(61)) );

output5(6) <= ( (input5(0) AND cntrl(6)) OR (input5(1) AND cntrl(14)) OR (input5(2) AND cntrl(22)) OR
AND cntrl(46)) OR
(input5(3) AND cntrl(30)) OR (input5(4) AND cntrl(38)) OR (input5(5)
(input5(6) AND cntrl(54)) OR (input5(7) AND cntrl(62)) );

output5(7) <= ( (input5(0) AND cntrl(7)) OR (input5(1) AND cntrl(15)) OR (input5(2) AND cntrl(23)) OR
AND cntrl(47)) OR
(input5(3) AND cntrl(31)) OR (input5(4) AND cntrl(39)) OR (input5(5)
(input5(6) AND cntrl(55)) OR (input5(7) AND cntrl(63)) );

output6(0) <= ( (input6(0) AND cntrl(0)) OR (input6(1) AND cntrl(8)) OR (input6(2) AND cntrl(16)) OR
AND cntrl(40)) OR
(input6(3) AND cntrl(24)) OR (input6(4) AND cntrl(32)) OR (input6(5)
(input6(6) AND cntrl(48)) OR (input6(7) AND cntrl(56)) );

output6(1) <= ( (input6(0) AND cntrl(1)) OR (input6(1) AND cntrl(9)) OR (input6(2) AND cntrl(17)) OR

```

```

AND cntrl(41)) OR
                                (input6(3) AND cntrl(25)) OR (input6(4) AND cntrl(33)) OR (input6(5)
                                (input6(6) AND cntrl(49)) OR (input6(7) AND cntrl(57)) );
output6(2) <= ( (input6(0) AND cntrl(2)) OR (input6(1) AND cntrl(10)) OR (input6(2) AND cntrl(18))OR
                                (input6(3) AND cntrl(26)) OR (input6(4) AND cntrl(34)) OR (input6(5)
AND cntrl(42)) OR
                                (input6(6) AND cntrl(50)) OR (input6(7) AND cntrl(58)) );
output6(3) <= ( (input6(0) AND cntrl(3)) OR (input6(1) AND cntrl(11)) OR (input6(2) AND cntrl(19))OR
                                (input6(3) AND cntrl(27)) OR (input6(4) AND cntrl(35)) OR (input6(5)
AND cntrl(43)) OR
                                (input6(6) AND cntrl(51)) OR (input6(7) AND cntrl(59)) );
output6(4) <= ( (input6(0) AND cntrl(4)) OR (input6(1) AND cntrl(12)) OR (input6(2) AND cntrl(20))OR
                                (input6(3) AND cntrl(28)) OR (input6(4) AND cntrl(36)) OR (input6(5)
AND cntrl(44)) OR
                                (input6(6) AND cntrl(52)) OR (input6(7) AND cntrl(60)) );
output6(5) <= ( (input6(0) AND cntrl(5)) OR (input6(1) AND cntrl(13))OR (input6(2) AND cntrl(21))OR
                                (input6(3) AND cntrl(29)) OR (input6(4) AND cntrl(37)) OR (input6(5)
AND cntrl(45)) OR
                                (input6(6) AND cntrl(53)) OR (input6(7) AND cntrl(61)) );
output6(6) <= ( (input6(0) AND cntrl(6)) OR (input6(1) AND cntrl(14))OR (input6(2) AND cntrl(22))OR
                                (input6(3) AND cntrl(30)) OR (input6(4) AND cntrl(38)) OR (input6(5)
AND cntrl(46)) OR
                                (input6(6) AND cntrl(54)) OR (input6(7) AND cntrl(62)) );
output6(7) <= ( (input6(0) AND cntrl(7)) OR (input6(1) AND cntrl(15))OR (input6(2) AND cntrl(23))OR
                                (input6(3) AND cntrl(31)) OR (input6(4) AND cntrl(39)) OR (input6(5)
AND cntrl(47)) OR
                                (input6(6) AND cntrl(55)) OR (input6(7) AND cntrl(63)) );

END behav;

```

Appendix D.4: The VHDL code for the crossbar switch.

-- This is a VHDL structural description of an 8*8 ATM switch.
 -- Designers: Arash Haidari-Khabbaz and Maryam Keyvani.
 -- Date: July 2000.
 -- The switch is made by connecting 8 instances of "porty" (input port controller),
 -- "fabric" (the switch fabric), and "c_bar" (the crossbar scheduler).

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
```

```
LIBRARY lpm;
USE lpm.lpm_components.ALL;
```

```
ENTITY switchx is
```

```
    GENERIC ( OUTPUT_PORT_SIZE: INTEGER := 4
              );

    PORT ( --Input port data and frame pulse lines. Frame pulse marks the beginning of a data packet
           --at each input port.
           data_in1, fp1           : IN STD_LOGIC;
           data_in2, fp2           : IN STD_LOGIC;
           data_in3, fp3           : IN STD_LOGIC;
           data_in4, fp4           : IN STD_LOGIC;
           data_in5, fp5           : IN STD_LOGIC;
           data_in6, fp6           : IN STD_LOGIC;
           data_in7, fp7           : IN STD_LOGIC;
           data_in8, fp8           : IN STD_LOGIC;

           global_reset            : IN STD_LOGIC;           --Resets all the counters, registers and the FIFO
           reset                   : IN STD_LOGIC;           --Resets everything but the FIFO
           clock                   : IN STD_LOGIC;

           --Output ports for data and frame pulse. Frame pulse marks the beginning of a data packet
           --leaving the switch.
           data_out_port1, fp_out_port1 : OUT STD_LOGIC;
           data_out_port2, fp_out_port2 : OUT STD_LOGIC;
           data_out_port3, fp_out_port3 : OUT STD_LOGIC;
           data_out_port4, fp_out_port4 : OUT STD_LOGIC;
           data_out_port5, fp_out_port5 : OUT STD_LOGIC;
           data_out_port6, fp_out_port6 : OUT STD_LOGIC;
           data_out_port7, fp_out_port7 : OUT STD_LOGIC;
           data_out_port8, fp_out_port8 : OUT STD_LOGIC;

           --data_valid shows whether the data on the output port is valid or not
           data_valid1              : OUT
STD_LOGIC;
           data_valid2              : OUT
STD_LOGIC;
           data_valid3              : OUT
STD_LOGIC;
           data_valid4              : OUT
STD_LOGIC;
           data_valid5              : OUT
STD_LOGIC;
           data_valid6              : OUT
STD_LOGIC;
           data_valid7              : OUT
STD_LOGIC;
           data_valid8              : OUT
STD_LOGIC;
```

```

--Shows the origin(input port) that is sending data packets to each output port
incoming_port_to_output1 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output2 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output3 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output4 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output5 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output6 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output7 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
incoming_port_to_output8 : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);

--Ports for simulation purposes:
P : OUT STD_LOGIC_VECTOR
(15 DOWNTO 1); --Priority vector. rotates with c_bar_clk

--c_bar_clk should be 53*8 times or more slower than main clock, so that the priority vector
--doesn't change while a packet is being switched.
c_bar_clk : IN STD_LOGIC;
request : OUT STD_LOGIC_VECTOR(64
DOWNTO 1);
grant : OUT STD_LOGIC_VECTOR(64
DOWNTO 1)
);

END switchx;

ARCHITECTURE structure OF switchx IS
-- The input port
COMPONENT porty
GENERIC(
PACKET_SIZE : INTEGER:= 53; --Port is set to handle packets of size 53 bytes
(ATM)

--Counter 53 is an 8 bit counter so it can service packets upto 256 bytes long
COUNTER_53_SIZE : INTEGER:= 8;
COUNTER_8_SIZE : INTEGER:= 4; --Counter 8 is a 4 bit counter
DATA_SIZE : INTEGER:= 8; --Each data byte is 8 bits long
FIFO_SIZE : INTEGER:= 2048; --Number of words in FIFO
FIFO_WIDTH : INTEGER:= 11; --Recommended value is
CEIL(LOG2(FIFO_SIZE))
VCI_FIFO_SIZE : INTEGER:= 128; --For a full data FIFO at least 78 VCI bytes are
required
VCI_FIFO_WIDTH : INTEGER:= 7; --Recommended value is
CEIL(LOG2(VCI_FIFO_SIZE))
VCI_VECTOR_SIZE : INTEGER:= 16; --Each VCI is 2 Bytes
OUTPUT_PORT_SIZE : INTEGER:= 4; --4 bits used to address an output port
TRANSLATION_TABLE: STRING --Is the LUT that the VCI's and output port
numbers are looked up at
);

PORT(
data_in : IN STD_LOGIC; --Input serial
data to the port
clock : IN STD_LOGIC; --Input clock to
the port
fp : IN STD_LOGIC;
--Input frame pulse to the port
global_reset : IN STD_LOGIC; --Resets all the counters, registers
and the FIFO
reset : IN STD_LOGIC; --Resets
everything but the FIFO
port_grant : IN STD_LOGIC_VECTOR(7 DOWNTO 0); --The grant vector for the port
data_out : OUT STD_LOGIC; --Serial data output of the port
data_valid : OUT STD_LOGIC;

--The request vector for the port. It is an 8 bit vector. Each bit of the vector specifies one output

```

```

--and is set when there is a request to send to that output
port_request : OUT STD_LOGIC_VECTOR(7 DOWNT0 0);
data_out_fp  : OUT STD_LOGIC --Frame pulse showing the beginning of the data being
shifted out
);
END COMPONENT;

COMPONENT c_bar
PORT(
arb_req      : IN std_logic_vector(64 DOWNT0 1); --For the 8*8 switch,
there are 64 possible requests
clk, reset  : IN std_logic;
grant      : OUT std_logic_vector(64 DOWNT0 1); --For the 8*8 switch,
there are 64 possible grants
P          : OUT std_logic_vector(15 DOWNT0 1) --The
priority vector working within c_bar schedualr
);
END COMPONENT;

COMPONENT fabric
GENERIC(
SWITCH_SIZE : INTEGER:= 8; --8x8 fabric by default
GRANT_SIZE  : INTEGER:= 64 --64 lines used to issue grants
);
PORT(
--inputs 4, 5, and 6 are made by bits from a constant matrix that is formed by input port
numbers
--outputs 4, 5, and 6 help make the incoming_port_to_output(i)s of the switch
input1 : IN STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The 8 input
data lines
input2 : IN STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The 8 input
frame pulse lines
input3 : IN STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The 8
data_valid lines going to the fabric
input4 : IN STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The MSBs of
input_port_name
input5 : IN STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The middle
bit of input_port_name
input6 : IN STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The LSBs of
input_port_name
cntl  : IN STD_LOGIC_VECTOR(GRANT_SIZE-1 DOWNT0 0); --The grant
vector used to control the fabric
output1 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The output
line carrying data out of the fabric
output2 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The output
frame pulse lines out of the fabric
output3 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The output
data_valid lines out of the fabric
output4 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The LSBs
of port_name out of the fabric
output5 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0); --The middle
bits of port_name out of the fabric
output6 : OUT STD_LOGIC_VECTOR(SWITCH_SIZE-1 DOWNT0 0) --The MSBs
of port_name out of the fabric
);
END COMPONENT;

-- Signals needed to connect the components

--arb_req_signal connects the output ports that each input port is requesting to the c_bar request line.It adds
--8 bit port_request outputs of all ports and forms a 64-bit vector that is connected to arb_req of c_bar.

```

```

SIGNAL arb_req_signal: STD_LOGIC_VECTOR (64 DOWNT0 1);

--grant_signal connects the grant output of the c_bar scheduler, which is a 64 bit vector, to the cntrl input
--of the fabric.
SIGNAL grant_signal : STD_LOGIC_VECTOR (64 DOWNT0 1);
SIGNAL input_data : STD_LOGIC_VECTOR (8 DOWNT0 1); --Connects data_out coming out of port to
fabric input

--output_data connects the fabric data output (output(i)) to output data line of the switch ((data_out_port(i))
SIGNAL output_data : STD_LOGIC_VECTOR (8 DOWNT0 1);

--input_fp connects the outgoing data's fp coming from the input port (data_out_fp) to the fabric input
SIGNAL input_fp : STD_LOGIC_VECTOR (8 DOWNT0 1);

--output_fp connects the fabric frame pulse output (output2) to frame pulse output of the switch
((fp_out_port(i))
SIGNAL output_fp : STD_LOGIC_VECTOR (8 DOWNT0 1);

--Reset signal for crossbar scheduler
SIGNAL resetb :STD_LOGIC;

--Input port number signals
SIGNAL input_port_name_bits2 : STD_LOGIC_VECTOR(7 DOWNT0 0);--Will be hard cored to
"11110000"
SIGNAL input_port_name_bits1 : STD_LOGIC_VECTOR(7 DOWNT0 0);--Will be hard cored to
"11001100"
SIGNAL input_port_name_bits0 : STD_LOGIC_VECTOR(7 DOWNT0 0);--Will be hard cored to
"10101010"

--Signals needed to carry control info to the output ports

--data_valid_signal connects output3 of the fabric to data_valid(i), which is the output of the switch
SIGNAL data_valid_signal : STD_LOGIC_VECTOR (8 DOWNT0 1);
SIGNAL data_valid_to_fabric : STD_LOGIC_VECTOR (8 DOWNT0 1);--Connects data_valid coming
from each port to input3 going to the fabric.
SIGNAL port_name_bit0 : STD_LOGIC_VECTOR (8 DOWNT0 1);-
-Connected to the output4 of the fabric
SIGNAL port_name_bit1 : STD_LOGIC_VECTOR (8 DOWNT0 1);-
-Connected to the output5 of the fabric
SIGNAL port_name_bit2 : STD_LOGIC_VECTOR (8 DOWNT0 1);-
-Connected to the output6 of the fabric

BEGIN

--Signal assignments

--Output data lines of the switch are constructed here
--output_data is a vector that connects outgoing data from the fabric to outgoing data of the switch
data_out_port1 <= output_data(1);
data_out_port2 <= output_data(2);
data_out_port3 <= output_data(3);
data_out_port4 <= output_data(4);
data_out_port5 <= output_data(5);
data_out_port6 <= output_data(6);
data_out_port7 <= output_data(7);
data_out_port8 <= output_data(8);

--Outgoing frame pulse lines of the switch are constructed here
--output_fp is a vector that connects the outgoing frame pulse from the fabric to outgoing fp of the switch
fp_out_port1 <= output_fp(1);
fp_out_port2 <= output_fp(2);
fp_out_port3 <= output_fp(3);
fp_out_port4 <= output_fp(4);
fp_out_port5 <= output_fp(5);
fp_out_port6 <= output_fp(6);
fp_out_port7 <= output_fp(7);
fp_out_port8 <= output_fp(8);

--Outgoing data_valid lines of the switch are constructed here

```

```

--data_valid_signal is a vector that connects output3 of the fabric to the data_valid lines of the switch
data_valid1 <= data_valid_signal(1);
data_valid2 <= data_valid_signal(2);
data_valid3 <= data_valid_signal(3);
data_valid4 <= data_valid_signal(4);
data_valid5 <= data_valid_signal(5);
data_valid6 <= data_valid_signal(6);
data_valid7 <= data_valid_signal(7);
data_valid8 <= data_valid_signal(8);

--Origin of each outgoing packet is specified by incoming_port_to_output(i)
incoming_port_to_output1 <= port_name_bit2(1) & port_name_bit1(1) & port_name_bit0(1);
incoming_port_to_output2 <= port_name_bit2(2) & port_name_bit1(2) & port_name_bit0(2);
incoming_port_to_output3 <= port_name_bit2(3) & port_name_bit1(3) & port_name_bit0(3);
incoming_port_to_output4 <= port_name_bit2(4) & port_name_bit1(4) & port_name_bit0(4);
incoming_port_to_output5 <= port_name_bit2(5) & port_name_bit1(5) & port_name_bit0(5);
incoming_port_to_output6 <= port_name_bit2(6) & port_name_bit1(6) & port_name_bit0(6);
incoming_port_to_output7 <= port_name_bit2(7) & port_name_bit1(7) & port_name_bit0(7);
incoming_port_to_output8 <= port_name_bit2(8) & port_name_bit1(8) & port_name_bit0(8);

--These vectors are connected to the fabric and according to the configuration of the fabric and the grants
--that are given, the number of the input port that was granted a request comes to the output of the fabric
input_port_name_bits2 <= "11110000";
input_port_name_bits1 <= "11001100";
input_port_name_bits0 <= "10101010";

--Signals assignments for simulation and testing purposes
request <= arb_req_signal;
grant <= grant_signal;
resetb <= NOT global_reset;

```

--Instances of ports 1 to 8

port1: porty

```

        GENERIC MAP (
                                TRANSLATION_TABLE => "lut1.mif"
        )

        PORT MAP (
                data_in    => data_in1,
                clock      => clock,
                fp          => fp1,
global_reset => global_reset,
                reset      => reset,
                port_grant => grant_signal (8 downto 1),
                data_out   => input_data(1),
                port_request => arb_req_signal (8 downto 1),
                data_out_fp => input_fp(1),
                                data_valid => data_valid_to_fabric(1)
        );

```

port2: porty

```

        GENERIC MAP (
                                TRANSLATION_TABLE => "lut2.mif"
        )

        PORT MAP (
                data_in    => data_in2,
                clock      => clock,
                fp          => fp2,
global_reset => global_reset,
                reset      => reset,
                port_grant => grant_signal (16 downto 9),
                data_out   => input_data(2),
                port_request => arb_req_signal (16 downto 9),
                data_out_fp => input_fp(2),

```

```

data_valid => data_valid_to_fabric(2)
);

port3: porty
    GENERIC MAP (
        TRANSLATION_TABLE => "lut3.mif"
    )

    PORT MAP (
        data_in => data_in3,
        clock => clock,
        fp => fp3,
global_reset => global_reset,
        reset => reset,
        port_grant => grant_signal (24 downto 17),
        data_out => input_data(3),
        port_request => arb_req_signal (24 downto 17),
        data_out_fp => input_fp(3),
        data_valid => data_valid_to_fabric(3)
    );

```

```

port4: porty
    GENERIC MAP (
        TRANSLATION_TABLE => "lut4.mif"
    )

    PORT MAP (
        data_in => data_in4,
        clock => clock,
        fp => fp4,
global_reset => global_reset,
        reset => reset,
        port_grant => grant_signal (32 downto 25),
        data_out => input_data(4),
        port_request => arb_req_signal (32 downto 25),
        data_out_fp => input_fp(4),
        data_valid => data_valid_to_fabric(4)
    );

```

```

port5: porty
    GENERIC MAP (
        TRANSLATION_TABLE => "lut5.mif"
    )

    PORT MAP (
        data_in => data_in5,
        clock => clock,
        fp => fp5,
global_reset => global_reset,
        reset => reset,
        port_grant => grant_signal (40 downto 33),
        data_out => input_data(5),
        port_request => arb_req_signal (40 downto 33),
        data_out_fp => input_fp(5),
        data_valid => data_valid_to_fabric(5)
    );

```

```

port6: porty
    GENERIC MAP (
        TRANSLATION_TABLE => "lut6.mif"
    )

```

```

PORT MAP (
    data_in    => data_in6,
    clock     => clock,
    fp        => fp6,
global_reset => global_reset,
    reset     => reset,
    port_grant => grant_signal (48 downto 41),
    data_out  => input_data(6),
    port_request => arb_req_signal (48 downto 41),
    data_out_fp => input_fp(6),
    data_valid => data_valid_to_fabric(6)
);

```

port7: porty

```

GENERIC MAP (
    TRANSLATION_TABLE => "lut7.mif"
)

```

```

PORT MAP (
    data_in    => data_in7,
    clock     => clock,
    fp        => fp7,
global_reset => global_reset,
    reset     => reset,
    port_grant => grant_signal (56 downto 49),
    data_out  => input_data(7),
    port_request => arb_req_signal (56 downto 49),
    data_out_fp => input_fp(7),
    data_valid => data_valid_to_fabric(7)
);

```

port8: porty

```

GENERIC MAP (
    TRANSLATION_TABLE => "lut8.mif"
)

```

```

PORT MAP (
    data_in    => data_in8,
    clock     => clock,
    fp        => fp8,
global_reset => global_reset,
    reset     => reset,
    port_grant => grant_signal (64 downto 57),
    data_out  => input_data(8),
    port_request => arb_req_signal (64 downto 57),
    data_out_fp => input_fp(8),
    data_valid => data_valid_to_fabric(8)
);

```

--Instance of the c_bar
switch_c_bar: c_bar

```

PORT MAP (
    arb_req => arb_req_signal,
    clk    => c_bar_clk,
    reset  => resetb,
    grant  => grant_signal,
    P      => P
);

```

--Instance of the fabric
data_fabric: fabric

```

PORT MAP (

```

```
input1 => input_data,  
input2 => input_fp,  
input3 => data_valid_to_fabric,  
input4 => input_port_name_bits0,  
input5 => input_port_name_bits1,  
input6 => input_port_name_bits2,  
cntrl => grant_signal,  
output1 => output_data,  
output2 => output_fp,  
output3 => data_valid_signal,  
output4 => port_name_bit0,  
output5 => port_name_bit1,  
output6 => port_name_bit2
```

```
);
```

```
END structure;
```

Appendix E: Switch simulation results