DESIGN AND PERFORMANCE ANALYSIS OF A NEW ATM SWITCH

by

SANDEEP AGARWAL
B.E., Awadesh Pratap Singh University, India, 1993

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this dissertation as conforming
to the required standard

[redacted]

Dr. F. ElGuibaly, Supervisor, Dept. of Elect. & Comp. Eng.

[redacted]

Dr. K. Li, Member, Dept. of Elect. & Comp. Eng.

[redacted]

Dr. N. Djilali, Outside Member, Dept. of Mechanical Eng.

[redacted]

Dr. J. L. Wegner, External Examiner, Dept. of Mechanical Eng.

Supervisor: Dr. F. ElGuibaly

ABSTRACT

Asynchronous Transfer Mode (ATM) has been accepted as a high speed networking solution for the Broadband Integrated Services Digital Network (BISDN) to meet the accelerating bandwidth principle and to integrate data, voice and video on one universal network. However, crucial to efficient routing of data, voice and video through the network requires development of fast and reliable ATM switches.

This thesis presents the design and performance analysis of a new ATM switch. Many different ATM switch architectures have been proposed in the literature. This thesis examines these architectures in detail to study their pros and cons. It aims to eliminate the problems faced by earlier ATM switch architectures by presenting a novel design of an ATM switch.

The general requirements of an ATM switch are presented based on ATM standards documents, and these requirements are used to design a new ATM switch, the VR switch. The name VR reflects the routing principle of the design which involves Virtual Routing prior to actual cell routing. The VR switch supports the different cell flows found in ATM networks by providing necessary functionality to route cells and manage the network resources. It is fast, supports Quality of Service (QoS) and multicast and broadcast functions. The VR switch shows good scaling properties and has the potential to be included in very high data rate networks (giga and tera bits). In addition, this is the first instance of a switch that is capable of using photonic switching technology in its switching fabric.

The VR switch is modeled to evaluate its performance in terms of cell loss probability, throughput and delay. It is shown that the VR switch has lower cell loss probability, optimal throughput/delay performance and satisfies all the general requirements of an ideal ATM switch. The logic of the switch design has been verified through simulation using VHDL. The gate level implementation of the VR switch is done using SYNOPSYS.

Examiners:

[redacted]

Dr. F. ElGuibaly, Supervisor, Dept. of Elect. & Comp. Eng.

[redacted]

Dr. K. Li, Member, Dept. of Elect. & Comp. Eng.

[redacted]

Dr. N. Diilali, Outside Member, Dept. of Mechanical Eng.

[redacted]

Dr. J. L. Wegner, External Examiner, Dept. of Mechanical Eng.

Table of Contents

List of Figures

List of Tables

Acknowledgement

This is the right time to take the opportunity to express my feelings and to thank everyone who helped me a lot and showed me the right path to complete my thesis. First and foremost, I would like to thank my supervisor, Dr. Fayez ElGuibaly for seeing potential in me and giving me an opportunity to work with him. Despite his busy schedule he always managed to get some time to guide me in my studies and research work. He created an atmosphere that was well suited to innovation and the pursuit of excellence. My time at the University of Victoria was a great success because of his personal, technical and financial support.

Next, I would like to express my gratitude to my cousin Dr. A.K. Agarwal and his family for their moral and financial support. They created an atmosphere that made me feel like home away from home.

I would also like to thank Dr. Ganesh Murthy, Inderpreet Singh, Mohamed Watheq El-Kharashi and Jasjeet Singh for their help. They used to spend hours with me discussing things. In addition, I would like to mention some very important friends who became an important part of my student life at the University of Victoria. These friends include Ranganathan, Venkat, Aziz, Rahyan, Siva, Amr Saaba, Saami Saab, Jeremy, Xiofang Wang and many others that I am sure I will later regret not having them included in this section.

Finally, I would like to thank my parents who listened to my fears and worries, encouraged me and gave me the will to continue. At this stage of their life when they needed me the most, they sent me abroad to pursue higher studies. I would also like to thank my sisters Ritu and Richa who supported me by taking care of my parents back in India.

Dedication

To

My Beloved Parents

Notation

AAI	ATM-user-to-ATM-user Indication
AAL	ATM Adaptation Layer
ATM	Asynchronous Transfer Mode
BISDN	Broadband Integrated Services Digital Network
BSS	Bus Structure Switching
CAC	Connection Admission Control
CBR	Constant Bit Rate
CLP	Cell Loss Priority
FCFS	First Come First Served
FIFO	First In First Out
HEC	Header Error Control
HOL	Head Of Line
IAR	Idle Address Register
ILMI	Interim Local Management Interface
ITU	International Telecommunications Union
NNI	Network Network Interface
OAM	Operation Administration and Maintenance
PDU	Protocol Data Unit
PT	Payload Type
QOS	Quality Of Service
SAR	Segmentation And Reassembly
SDM	Space Division Multiplexing
STM	Synchronous Transfer Mode
TDM	Time Division Multiplexing
UNI	User Network Interface
UPC	Usage Parameter Control
VBR	Variable Bit Rate

VC	Virtual Channel
VCI	Virtual Channel Identifier
VP	Virtual Path
VPI	Virtual Path Identifier

Chapter 1

Introduction

1.1 Motivation

During $70's$ and $80's$ systems that provided transfer rates of 1.5 to 2.0 Mb/s were sufficient to meet the requirements of most user applications which needed transfer rates of just a few kb/s. But as we moved further better means of communicating become increasingly important. Along with telephone and fax machines, users moved toward video conferencing and other computer-based methods of communication like e-mail. Hypertext, bit-mapped images, audio, video, animation and JAVA applications are now the norm and the expectations of users continue to rise. Thus, it became necessary to have systems that operate at much higher speeds. An increase in the methods of communication and in the number of people attempting to use them, made room for a technology capable of handling higher bandwidth requirements. This new technology must be capable of integrating data, voice and video on one universal network. Thus users will need only one interface to support all their communication requirements and service providers will no longer need to support several network infrastructures.

Inorder to satisfy the above requirements Asynchronous Transfer Mode (ATM) has emerged as the most promising technology for Broadband Integrated Services Digital Network (BISDN). It supports services which may be distributive or interactive, constant bit-rate (CBR) or variable bit-rate (VBR) and connection-oriented or connectionless. It also supports connections that may be point-to-point or multipoint, symmetric or asymmetric, unidirectional or bidirectional, and switched or (semi) permanent. It defines a user plane for the transfer of user information, a control plane

for connection control functions and a management plane for network supervision functions.

For efficient routing of information through the network it is necessary to build fast and reliable ATM switches. A large number of switching architectures has been proposed in the literature. A review of these switching architectures is found in [1] [2] [3] [4]. Each of these switching architecture has its own desirable features but has certain limitations. Some of these limitations make the switch not practical for implementation while others make them not scalable in terms of size and speed. The detailed study of these switching architectures revealed the problems with earlier designs and gave us the motivation to design a new ATM switch.

1.2 Research Goals

The main research goals of this thesis are summarized as follows:

- Design a new ATM switch that overcomes the drawbacks and limitations of earlier ATM switches.
- Analyze the new ATM switch by computer simulation and modeling to evaluate its performance in terms of cell loss probability, throughput and delay.
- Compare the performance of the new ATM switch with other switches to show that this switch performs better than other switches designed earlier.
- Study the complexity of the new ATM switch by considering the hardware details of each module of the switch and thus show that it is simple and can be easily implemented.
- Verify the design through simulation and implement the switch at the gate level.

1.3 Thesis Contributions

The main contributions of the thesis are summarized as follows:

- Designed a new ATM switch which overcomes the drawbacks of earlier ATM switches, supports Quality of Service (QoS), multicast and broadcast functions.

- Simplified the design of the switching fabric which enabled the use of photonic switching technology in the new ATM switch.
- Evaluated the performance of the new ATM switch by computer simulation and modeling to show that it has lower cell loss probability and optimal throughput/delay performance.
- Verified the design through simulation using VHDL.
- Implemented the switch at the gate level using SYNOPSYS.

1.4 Thesis Overview

Following this introduction, Chapter 2 gives an introduction of the ATM technology and presents a detailed survey of the existing ATM switch architectures. It discusses the pros and cons of each switch architecture and finally summarizes, in general, the limitations with earlier switch designs.

Chapter 3 discusses the design of the proposed ATM switch. It also discusses the switching operation of the proposed ATM switch. In addition, it also explains how the new ATM switch overcomes the drawbacks and limitations of earlier ATM switches.

Chapter 4 deals with the performance analysis of the proposed ATM switch. The analysis is done to evaluate its performance in terms of cell loss probability, throughput and delay by computer simulation and modeling. The performance is then compared with earlier ATM switches.

Chapter 5 discusses the complexity of the proposed ATM switch. It gives the hardware details of each module of the proposed ATM switch explaining very clearly the function carried by each of them. It also presents the behavioral simulation of the proposed switch and it's gate level implementation.

Chapter 6 concludes the thesis by providing a summary of the work, as well as some future directions for study.

Chapter 2

ATM Technology and Switch Architectures

2.1 Introduction

This chapter begins with an introduction of the ATM technology, the ATM cell and the ATM protocol model. Then it discusses the ATM switch and it's general requirements. Later, it gives a detailed survey of different ATM switch architectures. Finally, the chapter concludes with a summary of the pros and cons of different ATM switch architectures.

The main concept of BISDN is the support of a wide range of narrowband and broadband voice, data, video and multimedia services within the same network. The first switching and multiplexing technique to be considered for BISDN was the Synchronous Transfer Mode (STM) [5] [6]. Time slots within a periodic structure called frame, are allocated to a service for the duration of a connection. Each time slot represents a reserved piece of bandwidth dedicated to a single channel. STM channel is identified by the position of its time slot in a frame. STM switching is known as *circuit switching*. It provides fixed throughput and constant delay, and thus it is suitable for fixed-rate services but for different and variable-bit rates required by the diversity of services supported by BISDN, this approach grows cumbersome. This is because if a channel is not transmitting data, the time slot remains reserved and is still transmitted without any useful payload. If the other channels have more data to transmit, they have to wait until their reserved, assigned time slot occurs again.

Another approach which was first proposed by Baran [7] in 1964 is known as *packet switching*. Source information is segmented and each segment is encapsulated with a header to form a packet. The packet header contains information used for routing, error control, and flow control. At each switch, incoming packets are buffered, their headers are processed, and the packets are transmitted to the next appropriate switch. Packet switching results in large delay because of the requirements of complex protocols and complex buffer management inside the network and hence was found unsuitable for high speed integrated services network.

The problems of STM and conventional packet switching contributed later to the development of the Asynchronous Transfer Mode (ATM) concept.

2.1.1 The ATM concept

ATM is a high-speed connection-oriented switching technology which uses short fixed length packets called cells. Hence ATM is also known as *cell switching*. Each ATM cell consists of 53 octects. ATM combines the concepts of circuit switching and packet switching. It maintains the time slotted nature of transmission in circuit switching and these time slots are available to any user who has data ready to transmit. If no users are ready to transmit, then an empty or idle, cell is sent. It also maintains the concept of a packet but restricts it to a fixed size of 53 octects. Thus in ATM longer packets can not delay shorter packets because long packets are chopped up into many cells. This enables ATM to carry Constant Bit Rate traffic such as voice and video in conjunction with Variable Bit-Rate data traffic, potentially having very long packets within the same network. The term "Asynchronous" in ATM does not necessarily imply asynchronous transmission. Rather, it implies aperiodicity that is, no source shall own a time slot on a periodic basis.

2.1.2 The ATM Cell

The basic unit of ATM is the cell since all information is switched and multiplexed in an ATM network in these cells. Each cell consists of 53 octets (or bytes) comprised of a 5-octet header and a 48-octet payload. The ATM cell at the User-Network Interface

(UNI) consists of six fields whereas at the Network-Network Interface (NNI) it has five fields. The format of the ATM cell at the UNI and NNI is shown in Figure 2.1 [8]. The Generic Flow Control field allows a multiplexer to control the rate of an

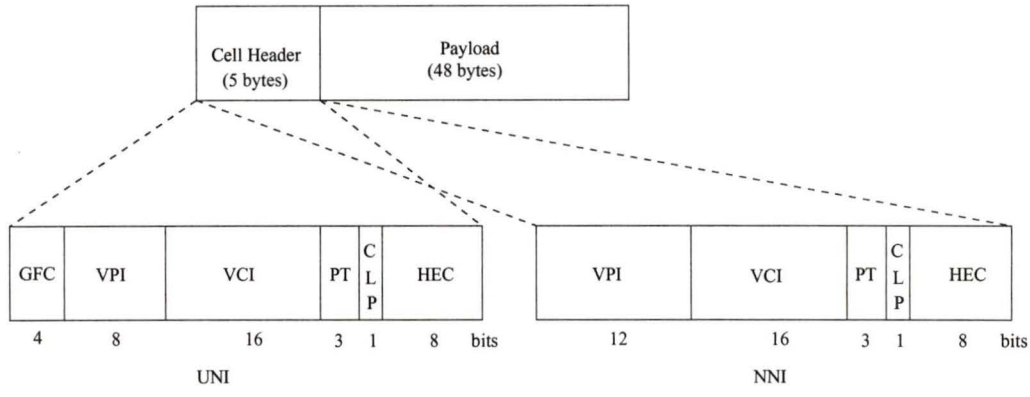


Figure 2.1. *Format of ATM cell at UNI and NNI.*

ATM terminal. The Virtual Path Identifier and Virtual Channel Identifier hold local significance only and identify the destination. The Payload Type indicates whether the cell contains user data, signaling data or maintenance information. The cell loss priority bit indicates whether the cell is high priority (0) or low priority (1). Lower priority cells are discarded before higher priority cells during congested intervals. The Header Error Check field detects and corrects error in the header. The payload field is passed through the network intact, with no error checking or correction. ATM relies on higher layer protocols to perform error checking and correction on the payload. These functions are performed on an end-to-end and not link-by-link basis.

2.1.3 The ATM Protocol Model

As shown in Figure 2.2, the ATM protocol model consists of Layers and Planes. The Layers are the Physical Layer, the ATM Layer, the ATM Adaptation Layer (AAL) and Higher Layers. The Planes are the User Plane, the Control Plane and the Management Plane.

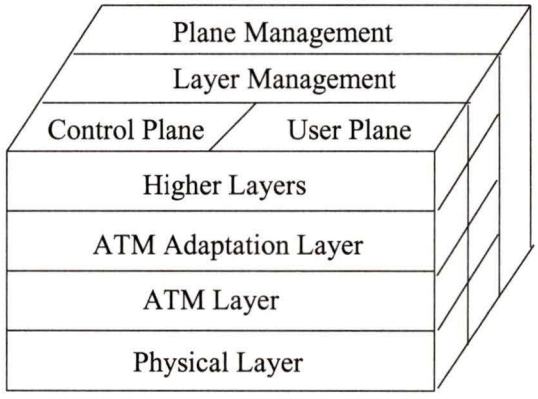


Figure 2.2. *The ATM Protocol Model.*

The User Plane is responsible for the transfer of user information across the network. The Control Plane is responsible for connection admission control functions. It handles the acceptance/rejection of a new connection and release of established connections. The Management Plane is responsible for management functions and includes operations like failure notification and connectivity verification. Management Plane is further divided into Layer Management and Plane Management. Layer Management includes fault management and performance management whereas Plane Management provides functions that span all three layers of the stack.

The Physical Layer is further divided into Physical Medium sub-layer and Transmission Convergence sub-layer. The Physical Medium sub-layer is responsible for sending and receiving a continuous flow of bits with associated timing information to synchronize transmission and reception. The Transmission Convergence sub-layer is responsible for cell delineation, header error control sequence generation and verification, cell rate decoupling, transmission frame adaptation and transmission frame generation and recovery.

The ATM layer establishes virtual connections and passes ATM cells through the ATM network. To do this, it uses the information contained in the header of each ATM cell. It is responsible for cell multiplexing/demultiplexing, header generation

and VPI/VCI translation. The ATM Adaptation Layer (AAL) is divided into two sub-layers, the Segmentation and Reassembly (SAR) sub-layer and the Convergence sub-layer. The SAR sub-layer performs segmentation of the higher layer information into a size suitable for the payload of the ATM cells of a virtual connection and at the receive side, it reassembles the contents of the cells of a virtual connection into data units to be delivered to the higher layers. The Convergence sub-layer performs functions like message identification and time/clock recovery.

2.2 ATM Switch Architectures

An ATM switch has a set of input ports and output ports through which the switch is interconnected to users, other switches, and other network elements. The requirements to identify the major functional blocks of ATM switch architecture are user plane considerations, control plane considerations and management plane considerations. The input modules, output modules and cell switch fabric together perform the basic cell routing and buffering functions required in ATM switching. The input modules receive incoming cells and prepare them for routing through the switch fabric. Output modules prepare outgoing cells for transmission. The switch fabric routes user data cells from input to output ports and possibly routes signaling and management cells between the other functional blocks in the switch.

An ATM switch can be either a Virtual Path (VP) or a Virtual Channel (VC) switch for a connection. A VP switch routes the cell to the correct output port based on the VPI of the cell, translating only the VPI. A VC switch routes the cell based on VPI and VCI, translating both the VPI and VCI.

2.2.1 ATM Switch Requirements

The general requirements of an ATM switch are

- ***Speed***. An ATM switch must have a wide range of operating speeds. ATM switches with operating speeds of 25, 51, 100, 155, 622 Mbps and 2.5 Gbps are already available. However, the challenge is to build high speed ATM switches so that it can support very high rate data (giga and tera bits).

- ***Cell loss***. The unscheduled nature of cell arrivals to an ATM switch results in two or more cells arriving at different inputs destined for the same output. The switching fabric allows only one cell to pass through while others need to be buffered. Hence buffering is essential in an ATM switch to store cells. Since it is not possible to have infinite buffers, cell loss is unavoidable in an ATM switch. However, an ATM switch should be designed to have minimum cell loss.
- ***Delay***. The delay in an ATM switch is defined as the average time a cell spends from the time it arrives at an input port, until the time it is successfully delivered on its requested output port. Thus it includes transit delay and buffer delay. Transit delay depends on the type of switching fabric and buffer delay depends on the buffering strategy. However, an ATM switch should be designed to have minimum delay.
- ***Small area and low power consumption***. An ATM switch must take small area and consume little power to facilitate higher integration levels.
- ***Reliability***. It is necessary for an ATM switch to be reliable and offer a certain level of redundancy. Also, running costs such as maintenance and upgrades should be as low as possible.
- ***Scalability***. It is necessary for an ATM switch to be scalable both in terms of size and speed. Low speed switches are required in less demanding applications whereas high speed switches are required in bandwidth hungry environments. Hence scalability in operating speed makes the switch possible to be used in any environment. Similarly, if the switch is scalable in size then it can be deployed in networks of any desired size, from a handful of nodes to thousands of nodes.
- ***Quality of Service***. Quality of Service is the mechanism by which applications and the network interact to ensure that applications enjoy the bandwidth and latency characteristics they require. QoS is negotiated at the time a connection is set up and is guaranteed by the network to remain at that level. Hence, it is very important for an ATM switch to support a number of QoS classes.
- ***Multicast and Broadcast functions***. An ATM switch must support multicast and broadcast functions.

Based on the general requirements discussed above, a number of ATM switch architectures have been proposed in the literature. These switch architectures are discussed

in detail in the next section.

2.3 Classification of ATM Switches

ATM switches are classified into two main categories.

- Time-division switches
- Space-division switches

In time-division switches all cells flow through a single resource that is shared by all input and output ports. This resource may be either a memory or a medium such as a ring or a bus. Hence time-division switches can be classified as

- Shared-memory switches
- Shared-medium switches

However, in time-division switches the shared resource must be capable of handling all the incoming and outgoing cells in one switching cycle. This requirement limits both the capacity and the size of the switch since the shared resource has to run at a speed higher than the line speed.

In space-division switches, multiple paths are established from the inputs to the outputs which run concurrently as opposed to the sequential operation of the time-division switches. Hence, the switching fabric does not need to run at a speed higher than the line speed. Thus the capacity of the switch is only limited by physical implementation restrictions like device pinout, connector restrictions and synchronization [9]. However, depending on the switching fabric used it may not be possible for all the required paths to be set simultaneously. This causes internal blocking and reduces the throughput of the switch. Space-division switches can be

- Crossbar based switches
- Disjoint-path based switches
- Banyan based switches

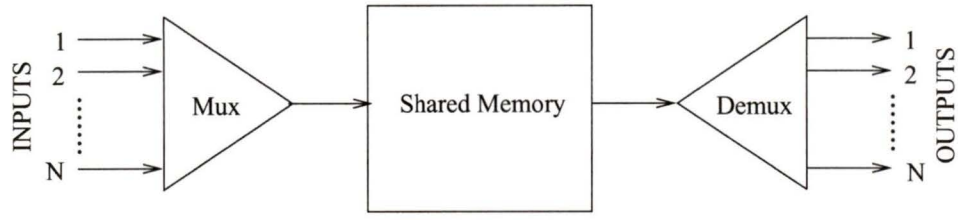


Figure 2.3. *Shared-memory switch.*

2.3.1 Shared-Memory Switches

This type of switch, shown in Figure 2.3, consists of a single dual-ported memory shared by all input and output lines. Cells arriving on all input lines are multiplexed into a single stream which is fed to the common memory for storage. Simultaneously, an output stream of cells is formed by retrieving them sequentially, one for each output port, from the shared memory. The main design constraints in shared-memory switches are

- ***Processing speed***. For a switch of size N, the time required by the controller to process sequentially N incoming cells and select N outgoing cells must be sufficiently small to keep up with the flow of incoming cells.
- ***Memory speed***. In order to accommodate all input and output traffic in a single time slot, the memory bandwidth must be at least the sum of the bandwidth of the incoming and outgoing lines. If N is the number of ports and V is the port speed, then the memory bandwidth must be atleast $2NV$ and the memory access speed is given by

$$\text{Memory access speed} = \frac{\text{cell length}}{2 \cdot N \cdot V}$$

Given a limit on memory access speed, required memory bandwidth can be achieved by means of parallel memory organization. However, the fixed cell length puts a limit on the number of memory banks that can be used in parallel and hence memory access speed becomes the bottleneck. Thus, the available memory speed and achievable processing speed puts a limit on the switch size N. However, to alleviate port restrictions due to memory speeds, multistage

shared-memory switches can be used.

- ***Memory size***. The memory size depends on the way it is shared. There are two ways in which memory can be shared. They are
  - ***Complete partitioning***. Memory is divided into N separate sections, each one allocated to a particular output queue. A cell destined to output j is lost if the section allocated to output queue j is full.
  - ***Complete sharing***. All queues share the entire memory and a cell is lost only if the entire memory is full.

  In complete sharing, less memory is required to achieve a certain cell loss probability but it may be unfair to other ports when traffic is bursty, causing degradation in switch performance.

Many shared-memory switches have been proposed in the literature. The Prelude switch [10] was the first shared-memory switch developed for ATM. The drawback of this switch is that it limits the number of input/output ports to the number of bytes per cell. Also, it employs complete partitioning of memory. Hence, the total memory size is much larger to guarantee acceptable loss rates at relatively high load. Also, no considerations have been made for multicasting or priority service. Another type of shared-memory switches discussed in literature are linked-list based switches [11], where the buffer memory is logically organized as N linked lists, one for each output. These switches employ complete sharing of memory and hence lesser memory is required to achieve a given cell loss probability. QoS can be easily supported in these switches. However, to support multicast functions, a multicast circuit is required. Another shared-memory switch discussed in literature is Hybrid shared and dedicated output buffer switch [12]. In this switch the control mechanism to route the cell in the shared memory to its output port uses N dedicated FIFO buffers, one for each output port. The advantage of this switch is that it is easy to switch from complete sharing to complete partitioning of shared memory. This is because the sizes of the address FIFO buffers determine the number of cells which can be stored into the shared buffer memory with the same destination output port. Thus, by controlling the size of address FIFO buffers, overloading of the shared buffer by some output ports during bursts can be easily controlled [13]. The disadvantage of this switch is that the amount of buffering required is slightly higher due to the

buffering required for the FIFO queues. Also, an additional broadcasting service control circuit is required to support multicast and broadcast functions. There are other improved shared-memory architectures found in literature. Architectures that replace the usual multiplexing and demultiplexing stages by crosspoint space switches to relax the memory access speed are described in [14], [15]. Architectures that parallelise the memory management and memory access operations to increase the processing speed are described in [16].

2.3.2 Shared-Medium Switches

Figure 2.4 shows the basic structure of a shared-medium switch. It consists of a

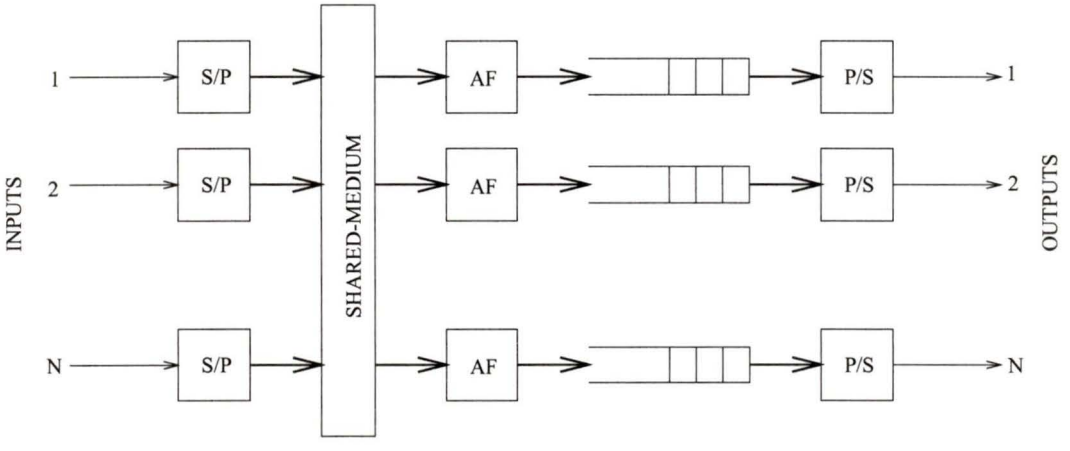


Figure 2.4. *Shared-medium switch.*

high-speed medium, typically a bus, shared by all input and output lines. Each output line is connected to the bus via an interface consisting of an address filter and a FIFO buffer. Thus, after serial to parallel conversion, cells arriving on the input lines are synchronously multiplexed on to the shared bus. The address filter selects the FIFO buffers based on the destination output ports. Thus cells are stored in the respective FIFO buffers from where they are demultiplexed on to the output lines. One big advantage of shared-medium switches is that multicast and broadcast functions and QoS can be easily supported. However, the disadvantage is that the

speed of the medium puts a limit on the size of the switch. Thus, similar to shared-memory switch, the main design constraints are the processing speed and the bus speed. However, in this case the bandwidth of the bus must be equal to N times the line speed. A shared-medium switch found in literature is the ATOM (ATM Output Buffer Modular) switch [17]. It uses a bit-slice organization to alleviate the bottleneck of the medium speed. Another shared-medium switch found in literature is the PARIS (Packetized Automated Routing Integrated System) switch [18]. However, this switch was designed with an intention to accomodate variable size packets. One recently presented shared-medium switch is the BSS (Bus Structure Switching) element [19]. It uses a full ATM cell bus width to transfer a cell within one clock cycle.

2.3.3 Crossbar-based Switches

A crossbar switch is shown in Figure 2.5, for switch size N=4. Basically, an $(N \times N)$

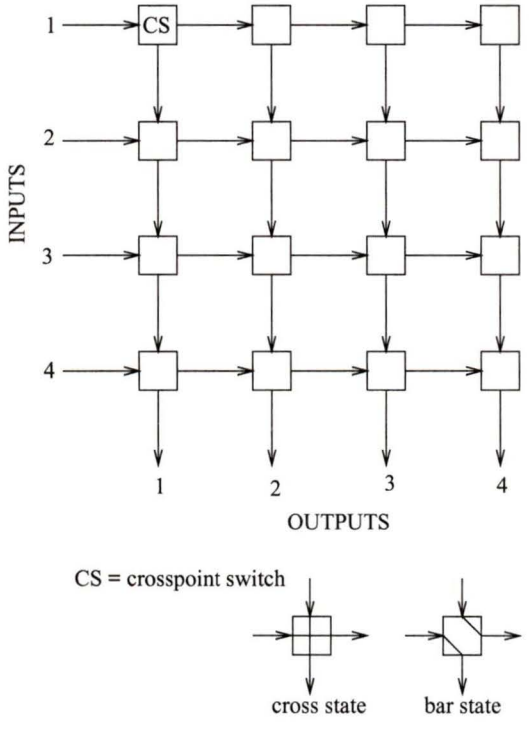


Figure 2.5. *The crossbar switch.*

crossbar switch consists of a square array of N^2 individually operated crosspoints, one for each input-output pair. Each crosspoint has two possible states: cross (default) and bar. A connection between input port i and output port j is established by setting the $(i,j)th$ crosspoint switch to the bar state. Crossbar switches are nonblocking, simple in architecture, and modular. However, they have the following main drawbacks.

- Square growth of complexity that is, it requires N^2 crosspoints and therefore the size of realizable switches tend to be limited.
- When self-routing is used, the processing performed at each crosspoint requires knowledge of the complete output port address.
- The transit time is not constant over all input/output pairs unless artificial delays are introduced at the inputs and outputs of the switch.
- Despite being nonblocking, crossbar switches suffer from output blocking.

Crossbar switches discussed in literature are the Bus Matrix switch [20], and the Butterfly switch [21], in which each crosspoint is replaced by an address filter and a FIFO buffer. Limited Intermediate Buffer switch [22] is another crossbar switch in which a single buffer called an intermediate buffer is placed at each crosspoint in addition to the FIFO buffers at the input ports. The Knockout switch [23] is another crossbar switch in which every input port has a contention free path to every output port.

2.3.4 Disjoint-path-based switches

These switches have the capability of disjointly establishing all possible N^2 paths. Thus, no blocking occurs within the switching fabric, even among cells destined to the same output port. However, buffering at the output ports is needed for multiple cell arrivals destined to the same output port. In these switches paths can be disjoint either in time or in space. The Bus Matrix switch [20], the Butterfly switch [21] and the Knockout switch [23] mentioned earlier also lie under this category.

2.3.5 Banyan-based switches

Banyan networks are a broad class of multistage interconnection networks (MINs) with exactly one path from any input to any output. Besides the self-routing property, banyan networks are modular, have the same latency for all input-output pairs and have a complexity of O (N log_2N) compared to O (N^2) for the crossbar switch. However, while path-uniqueness results in the ease of preserving cell-sequencing, it also results in the network being internally blocking. Internal blocking arises when two cells destined to different output ports request the same outlet, in such a situation, one cell is selected, and the other cell is dropped. Thus, crossbar switches suffer only from output blocking, while banyan-based switches suffer from both output and internal blocking and the later increases with the number of stages of a banyan network. The solutions to overcome the problem of internal blocking are

- Place buffers at the points of conflicts, leading to what is known as buffered-banyan switches. Performance analysis of these switches [24] [25] [26] show that these switches have a limited throughput which decreases rapidly as N, the switch size, increases.
- Sort the input cells in order to remove output conflicts by means of a batcher sorter, leading to what is known as batcher-banyan switches. Some batcher-banyan switches found in literature are the Starlite switch [27] and the Sunshine switch [28].
- Replace each internal link by d links, $d \geq 2$, leading to what is known as dilated-banyan switches [29] [30]. Dilation improves the throughput of banyan switches.
- Use multiple copies of the banyan interconnection network in series, leading to what is known as tandem-banyan switches [31]. The number of possible paths between inputs and outputs is increased and output queueing is achieved.

2.4 Buffering Design Options

In an ATM switch, it is possible that two or more cells may arrive at different input ports destined for the same output port. The switching fabric allows one cell

to pass through while others need to be buffered. Hence, buffering is essential in ATM switches. ATM switches are also classified according to the buffering strategy employed in it. The different buffering design options are

- Input buffering
- Output buffering
- Completely shared buffering

2.4.1 Input Buffering

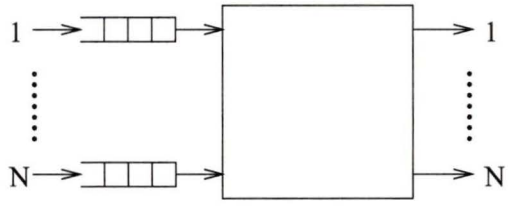


Figure 2.6. *Input Buffering.*

With input buffering, shown in Figure 2.6, a separate buffer is placed on each input to the switch to store the incoming cell. The cell at the head of each buffer is served. If the cells at the head of each buffer are destined to different outputs, the nonblocking switch fabric allows each to pass. However, if more than one cell is destined to the same output, only one cell is allowed to pass through the switch fabric, while the others must wait until the next time slot, when a new selection is made among the cells that are then waiting. It is to be noted here that while a cell at the head of the buffer is waiting for its turn to get access to the switching fabric, other cells queued behind it are blocked from reaching possibly idle outputs on the switch. This is called head of line (HOL) blocking which reduces the throughput of an input buffered switch.

Many different switch architectures have been proposed to improve the throughput of input-queued switches [32] [33]. In these switch architectures, switch performance is improved by providing multiple logical queues (one per output) in each input buffer

as shown in Figure 2.7. The switching fabric for this architecture must solve a match-

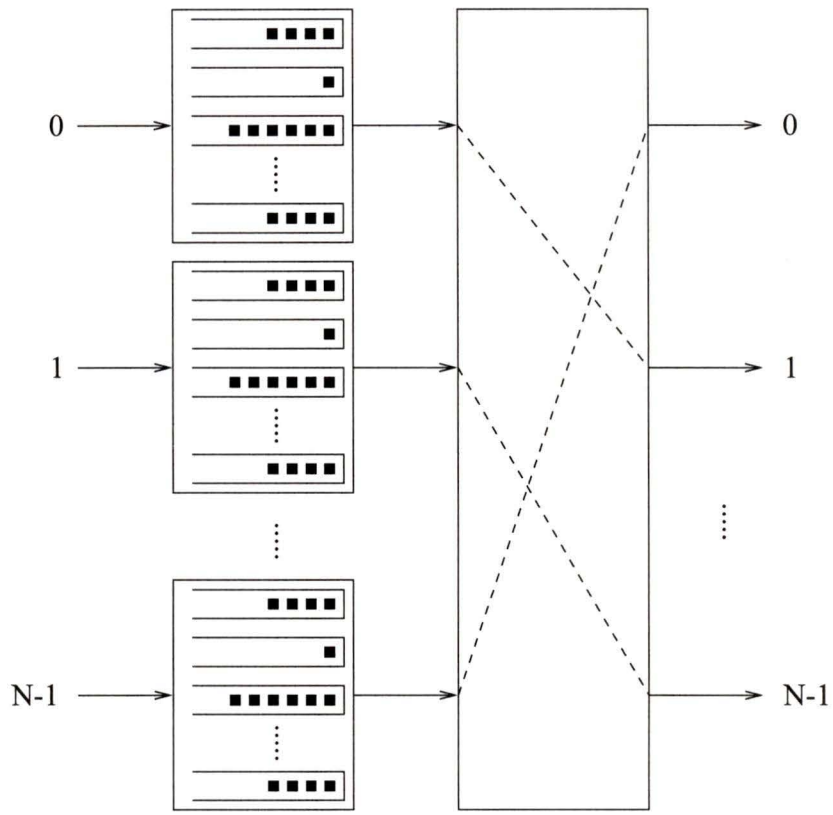


Figure 2.7. *Design of input-queued switch to improve throughput.*

ing problem during each cell time, for each input buffer, such that no two input buffers choose cells destined for the same output port and also the number of cells routed is maximized. This matching problem is extremely hard to solve quickly and with good performance.

Another arrangement in which the arriving cells are not so much queued at each input but smoothed is shown in Figure 2.8 and is known as input smoothing [34]. In this arrangement, a separate buffer is placed at each input to the switch where incoming cells are stored but are served every b time slots. Thus, atmost Nb cells enter the fabric simultaneously, of which b can be simultaneously received at each output where the cells are multiplexed onto the output line. Any more than b cells destined

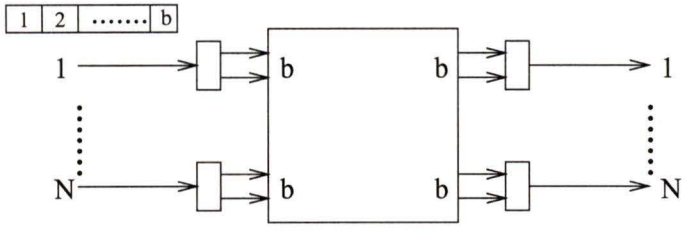


Figure 2.8. *Input Smoothing.*

for an output are lost. As shown in Figure 2.9, input smoothing with buffering is another technique for queueing in ATM switches proposed in [35]. This technique is

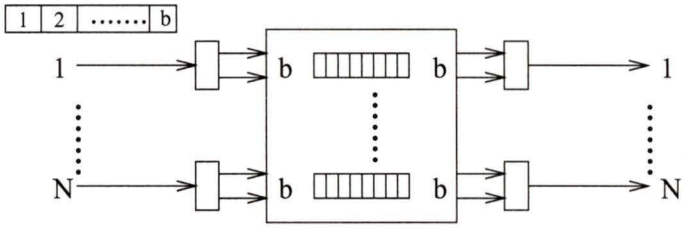


Figure 2.9. *Input Smoothing with Buffering.*

the same as that of input smoothing but here cell loss is reduced by adding a pool of buffers assigned per output, inside the switching fabric.

2.4.2 Output Buffering

In output buffering, as shown in Figure 2.10, a separate buffer is placed on each output to the switch. All buffering is done at the outputs of the switch. It is to be noted that output buffering, unlike input buffering does not suffer from head of line blocking. Cells going to different outputs are also not delayed. However, there is unavoidable congestion caused by multiple cells simultaneously arriving on different inputs addressed to the same output. Hence, an output buffer must be able to receive upto N cells at a time, where N is the switch size. This increases the implementation

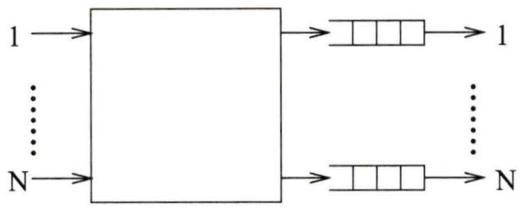


Figure 2.10. *Output Buffering.*

complexity because an output buffered switch needs to operate N times faster than the input/output line speed.

2.4.3 Completely Shared Buffering

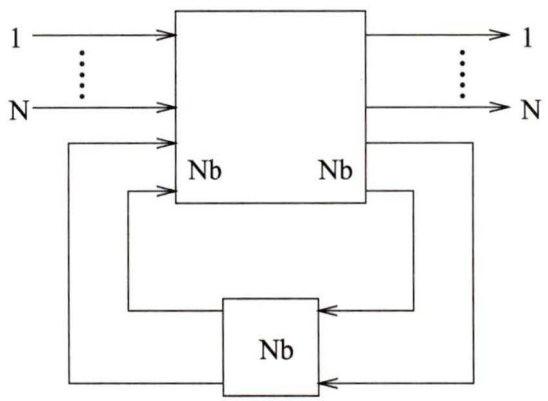


Figure 2.11. *Completely Shared Buffering.*

In completely shared buffering, as shown in Figure 2.11, rather than having a separate buffer for each output, there is one completely shared buffer. Upto Nb cells can enter/leave the shared buffer. Thus switch size is increased from $N \times N$ to $N(b+1) \times N(b+1)$. In each time slot upto N new cell arrivals to the switch and upto Nb buffered cells enter the switching fabric whereas only upto N cells (with distinct destination ports) are routed to the output and remaining cells are routed again to the shared buffer. These cells are then recirculated in the next time slot. Increase

in switch size and recirculation of buffered cells in each time slot are the drawbacks with this technique.

2.5 Other Classifications

In this section we discuss some other classifications of ATM switches. These are

- Single-stage or multi-stage switches
- Single-path or multi-path switches
- Blocking or non-blocking switches

In single-stage switches, inputs and outputs are connected through one stage only whereas in multi-stage switches, they are connected through multiple stages. Similarly, in single-path switches, there is only one path between any input-output pair whereas in multi-path switches, more than one path exists for each input-output pair. Switches which do not suffer from internal blocking are called non-blocking switches. However non-blocking switches may be output-blocking or output-nonblocking. Switches described earlier in this chapter may fall in any of these categories. However, it is to be noted here that a blocking switch is not desirable.

2.6 Summary

The ATM technology, the ATM switch and the general requirements of an ATM switch has been presented. The different ATM switch architectures were studied based on the general classification of ATM switches. It has been concluded that time-division switches do not scale well in terms of size and speed since the shared resource has to run at a speed higher than the line speed. In space-division switches, the switching fabric does not need to run at a speed higher than the line speed but they may suffer from internal blocking. Also, it has been concluded that the implementation of input buffered switch is simple in the sense that the switch has to operate at the same speed as the input/output lines and not many times faster. However, input buffered switch has low throughput/delay performance because of head of line blocking. Output buffered switch has optimal throughput/delay performance but the switch has to run at a speed higher than line speed. Increase in switch size and recirculation of buffered

cells in each time slot are the drawbacks in switches with completely shared buffering.

Finally, it is hard to single out any one among the architectures as the best because each architecture has its own desirable features but they have some drawbacks also. The desirable features of different switch architectures are combined to design a new ATM switch, the VR switch, discussed in next chapter.

Chapter 3

Design of The VR Switch

3.1 Introduction

In an ATM switch, an incoming cell is first stored at the input to process the cell header. Cell header processing is done to identify the type of the cell, update VPI/VCI and determine the destination port. Cells can be identified as user, signaling (CAC) or management (OAM) based on the payload type field in cell header. User cells are then routed to the destined output ports whereas signaling and OAM cells are first processed in connection admission control (CAC) and system management (SM) modules. An ATM switch should support multicast and broadcast functions and maintain Quality of Service (QoS).

Detailed study of different ATM switch architectures revealed that each switch architecture has some desirable features but suffers from several drawbacks. In this chapter a new ATM switch, the VR switch, is proposed which combines the desirable features of earlier ATM switches and eliminates the problems present in earlier ATM switches [36].

3.2 Design Goals and main design decisions

The VR switch is designed to achieve the following goals

- ***Operation at line rate***. This reduces delays and ensures low speed os silicon. This requirement implies that the switch must be an input buffered switch and

must have localized control. Thus we have localized Connection Admission Control (CAC) and Operation and Maintenance (OAM) functions.
The switching fabric can not be time-division based and hence we decided to have space-division switching fabric in the VR switch.
Processing at each switching node should be simple and the switching fabric should be output driven to overcome output blocking and this implies there should be output queues and priority scheduling is done locally at each output port.

- ***Reduced power and silicon area***. This goal implies the need of bit-serial operation. Typically bit-serial operation is slow and serial-to-parallel conversion of the incoming cells is required. However, in the case of the VR switch bit-serial operation can still be performed because cell processing is distributed between input and output ports.
- ***Scalability***. In terms of size and speed, this implies no communication among the different input ports and similarly for the output ports.
- ***Support of photonic switching***. This implies that no processing should be done in the switching fabric.
- ***Multicast and Broadcast Functions***. The VR switch aims at providing multicast and broadcast functions easily without making multiple copies of the cell.
- ***Quality of Service***. Another goal in the design of the VR switch is that it should support different service classes inorder to maintain the required QoS.

3.3 Main Features of the VR Switch

The main features of the VR switch are

- Cells are stored in shift register buffers located in the input module. Each input port has its own bank of input buffers. This feature makes the switch an input-buffered switch.
- All shift registers are directly connected to the switching module. This feature eliminates HOL blocking and low throughput problems faced by input-buffered switches.

- Cells are routed between input and output ports in bit serial format. Hence no serial-to-parallel conversion is required. This feature reduces communication area and power requirements.
- Connection Admission Control (CAC) and Operation and Maintenance (OAM) modules are local to each input. This feature matches the processing speed to line speed and makes it independent of switch size.
- All cells are routed simultaneously and internal data rate matches the line rate.
- Each output port has virtual output queues that store only the addresses of the cells destined to that port. This feature allows the switching fabric to be output driven thereby making it non-blocking.
- This switch has localized concentrators which do the switching. This feature simplifies the design of switching fabric which now requires no processing but simple movement of data in bit serial fashion. Thus the switch has the ability to drive fiber cable straight from input module.
- QOS and multicast functions are easily supported by the switch.

3.4 Design of the VR Switch

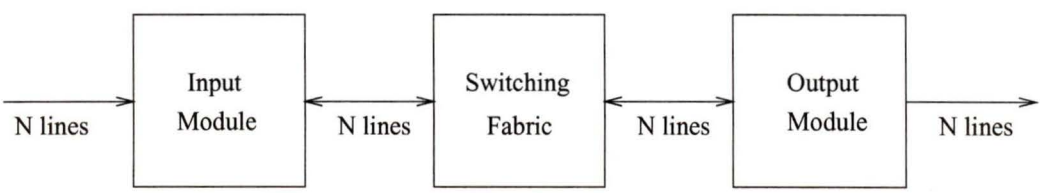


Figure 3.1. *Block diagram of the proposed ATM switch.*

Figure 3.1, shows the block diagram of the VR switch. Like the contemporary ATM switches [1], [4], the proposed switch contains input module, switching module and the output module.

3.4.1 Input Module

For a switch of size N, the input module consists of N input ports. Figure 3.2, shows the block diagram of an input port. As shown in Figure 3.2, each input port consists

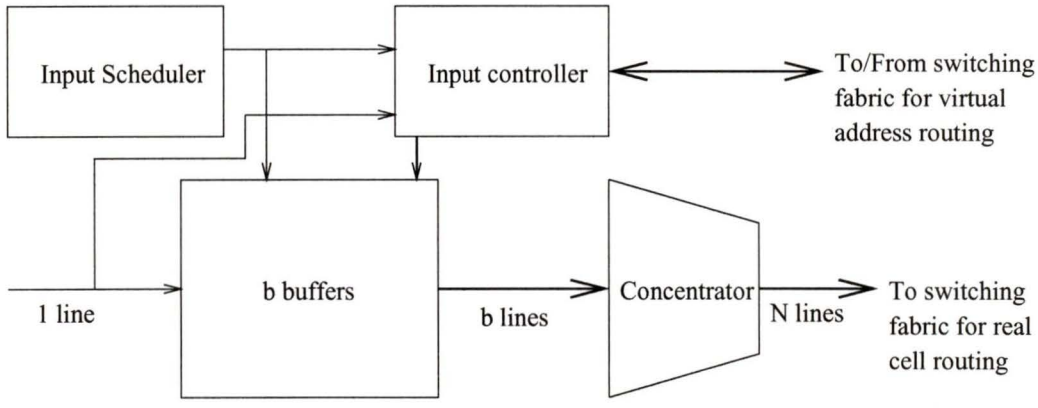


Figure 3.2. *Block diagram of an input port.*

of buffers, input controller, input scheduler and a concentrator.

- ***Buffers***. Each input port consists of a bank of parallel shift register buffers to store only the payload of the incoming cells. Buffer access speed match the port speed and no serial-to-parallel conversion is required since the buffers accept, store and dispatch the data in bit-serial fashion.
- ***Input controller***. The input controller receives only the header of the incoming cell. It is responsible for processing the cell header to determine the type of the cell and to determine the call number of the cell. The call number is used by output module to determine the destination port. There is seperate controller for each input port rather than a single shared controller and this avoids access conflict. Also, the input controller consists of cac_Processor and sm_Processor to perform connection admission control (CAC) and management (OAM) functions. Thus, CAC and OAM are local to each input port. This makes the switch more scalable to large sizes because the amount of processing is not increased with the volume of signaling/OAM traffic which is proportional to the switch size. Usage parameter control (UPC) is also located in the in-

put controller to do policing. It ensures that user traffic complies to the traffic contract. Refining actions are taken to violating cells.

- ***Input scheduler***. The input scheduler keeps updated information of the empty shift register buffer. It consists of decoder, idle address registers (IAR), priority generator and encoder. Each IAR corresponds to a particular shift register buffer and hence the number of idle address registers is equal to the number of shift register buffers. Each IAR stores a '1' or '0' in it. A '1' indicates that the corresponding shift register buffer is empty whereas a '0' indicates that the corresponding shift register buffer is full. Initially, all the idle address registers have stored '1'. This is due to the fact that initially, all the shift register buffers are empty.
- ***Concentrator***. Each input port consists of a b to N concentrator, where b is the number of shift register buffers in each port and N is the switch size. Thus the concentrator connects each shift register buffer to the switching module. This feature eliminates head-of-line (HOL) blocking and low throughput problems. Also, the switching fabric design is simplified since the processing to establish the connections between shift register buffers and their destination output ports is done by the localized concentrators.

3.4.2 Switching Fabric

The VR switch uses two different switching fabrics.

- ***Switching fabric for virtual address routing***. This switching fabric is used to send the address of the input buffer where the cell is stored, the call number of the cell and 12 bits of the cell header which remains unchanged throughout the connection, to the output port. It is also used to send a cell's address from output port to the input port to establish connections for real cell routing. Thus this switch is input driven as well as output driven but involves small protocol data units. The simplest way is to use three buses, selection bus ($\log_2 N$ bits wide), address bus ($\log_2 N + \log_2 b$ bits wide) and header bus (12 bits wide). Thus, in effect the VR switch has time division virtual routing.

- ***Switching fabric for real cell routing***. This switching fabric is used to route the cell from input port to output port. It is a bus, N bits wide, which connects the input concentrators to the outputs. Thus it is possible to drive fiber optics directly from the input module.

3.4.3 Output Module

The output module consists of N output ports for a switch of size N. Each output port consists of output controller, virtual FIFO queues, output buffer and output scheduler. The input module sends the addresses of the cells stored in input shift register buffers to the output module. At the same time it also sends the call number of the cell and 12 bits of cell header to the output module. The output controller receives the call number of the cell, determines the destination port and helps in storing the address of the cell, the call number and 12 bits of cell header in the corresponding virtual queues of the destination port. The output controller also does the VPI/VCI translation and generates 28 bits of new cell header. The output scheduler controls the virtual FIFO queues and output buffer by sending them read signals. Furthermore, it drives a priority control scheme to maintain the QoS of different services. The priority scheduler for output queueing strategy is simpler and more efficient than for input queueing or shared memory. The VR switch, though an input buffered switch, uses virtual output queues which makes it easy to support any priority scheduler designed for output buffered switch.

3.5 Switching Operation

Each input port of the VR switch receives cells from the physical layer. As a synchronous physical medium is assumed, if there are no valid cells at the physical layer then an unassigned cell is generated by the physical layer and sent to the ATM switch. In the switch cells are processed and routed to the destined output port. Each output port then transfers the cell to the physical layer. If an output port has no cell to transfer to the physical layer, then a signal is sent instructing the physical layer to generate an unassigned cell.

The switching operation of the switch is synchronized with two clocks. A high level clock, denoted hclk, that controls the flow of cells across the switch and a primitive clock, denoted pclk, used to transfer the bits of a cell. The clock hclk corresponds to the arrival/departure of ATM cells; during each rising edge of hclk an ATM cell arrives at and an ATM cell leaves from each port of the VR switch. The period of hclk is the time between the arrival of two successive ATM cells. According to Figure 3.3, this is equivalent to the time required to transfer one 424-bit ATM cell (x) plus the time (y) after which next ATM cell arrives. The clock hclk is maintained

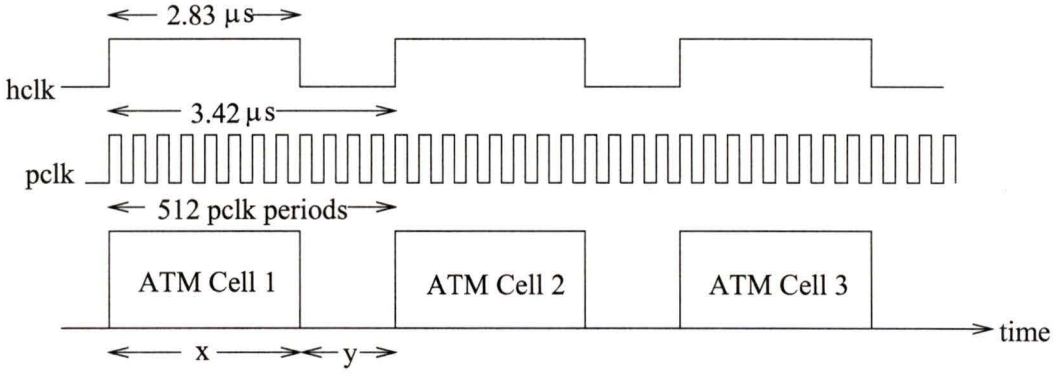


Figure 3.3. *Timing diagram of ATM cell arrivals, line rate 155 Mbps.*

high for the time needed to transfer one 424-bit ATM cell. Assume that the ports operate at the SONET rate of 155.52 Mbps. For this physical layer, the cell transfer capacity is 149.76 Mbps [8]. To transfer one 424-bit ATM cell at this rate requires approximately 2.83 μs. Also, from ATM forum specifications for 155.52 Mbps SONET STS-3c physical layer it can be assumed that the time between arrival of two successive ATM cells is equivalent to the time required for transferring 512 bits; approximately 3.42 μs. Thus the period of hclk is assumed 3.42 μs out of which it remains high for 2.83 μs. The whole hclk period is divided into 512 pclk periods. Hence a 6.68 ns pclk period is necessary. The rising edge of each pclk is used to transfer one bit. The switching operation of the VR switch is shown in Figure 3.4.

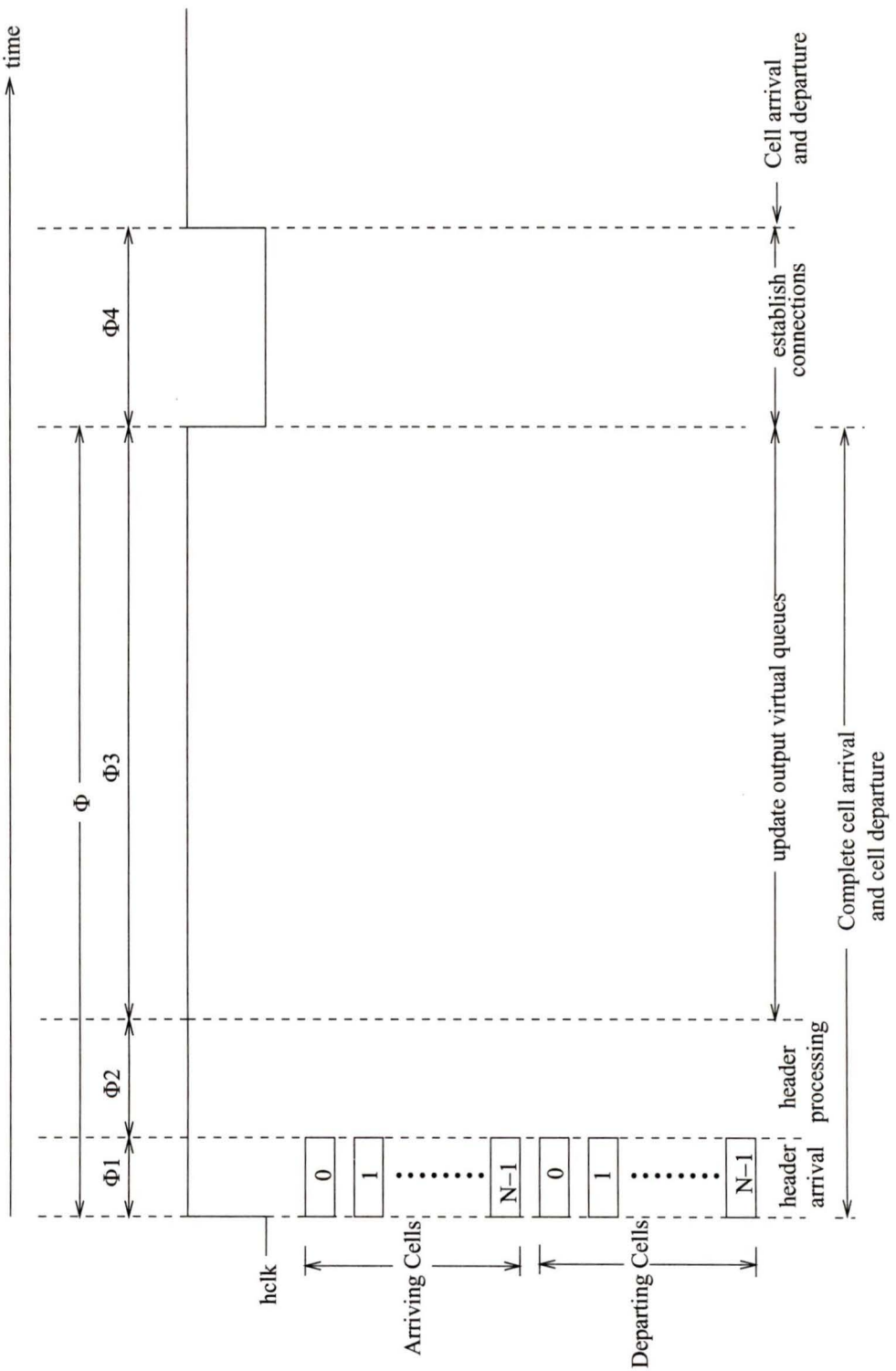


Figure 3.4. *Switching operation of the VR switch.*

As shown in Figure 3.5, a cell arrives in each input port simultaneously, at the rising edge of hclk. As soon as the header of the cell is received i.e. after 40 pclk periods (phase $\phi 1$), cell header processing is done simultaneously in all input ports (phase $\phi 2$). Cell header processing at the input consists of the following tasks.

- ***Determine the type of the cell.***
- ***Determine the call number of the cell.***
- ***Determine the cell loss priority.***

The task of translating the VPI/VCI values is done at the output ports. Now, data routing is done in three parts.

- ***Virtual address routing to update the output virtual queue.*** The address of the buffer where the cell is stored, the call number of the cell and 12 bits of cell header are sent to the output port. The call number determines the destination port and is stored in the virtual queue of that port. Also, the address of the buffer where cell is stored and the 12 bits of cell header are stored in the corresponding virtual queues of the destination port. This involves $N \times N$ switching problem which is input driven and uses small protocol data units (PDU), the cell address $\log_2 N + \log_2 b$ bits, the call number $\log_2 N$ bits and 12 bits of cell header. We can use space division multiplexing (SDM), time division multiplexing (TDM), contention or a combination of all three to ensure full routing within one cell frame. This is shown by phase $\phi 3$ in Figure 3.5. By the time the payload of the cell is still arriving and is stored in the payload buffer. Complete cell arrival is shown by phase ϕ in Figure 3.5.
- ***Virtual address routing to establish connections.*** A cell's address is selected from each output port and connection is established between the input shift register buffer (in which that cell is stored), cell switch fabric and output module, to route the cell. Though it is output driven, it also uses small protocol data units (PDU) and the same $N \times N$ switching problem. This is shown by phase $\phi 4$ in Figure 3.5.
- **Real cell routing:** Once the connections are established, real cell routing is done simultaneously from each input port at the rising edge of hclk. Real cell routing uses relatively longer protocol data units (PDU), 53 bytes of the cell, but it is output driven which simplifies the switching fabric design.

3.6 Summary

The design of the VR switch is proposed. It is an input buffered switch in which cells are received and routed in bit serial format. No serial-to-parallel conversion is required. No part of the VR switch needs to run at a speed higher than the line speed. It is seen that in the VR switch cell header processing is done in two parts, first at the input ports and then at the output ports. Cell header processing at the inputs is done simultaneously at each input port as soon as the header is received and payload of the cell is still arriving. It determines the type of the cell and the call number of the cell. Cell header processing at the outputs is done to translate the VPI/VCI values of the cell header. It is seen that in the VR switch cell routing is done in three parts. Prior to real cell routing, virtual cell routing is done twice, first to update the output virtual queues and then to establish connections. Hence it needs two switching fabrics, one for virtual cell routing and one for real cell routing. Virtual cell routing, to update the output virtual queues is done during the time the payload of the cell is still arriving thus utilizing the time in a good way. The use of concentrators in the input module not only simplifies the design of switching fabric for real cell routing but also makes the switch capable of driving fiber cable straight from input module. The VR switch solved the head-of-line problem and hence is expected to give a good performance. The performance analysis of the VR switch is done in next chapter.

Chapter 4

Performance Analysis of The VR Switch

4.1 Introduction

The performance of an ATM switch is usually evaluated based on three measures: cell loss probability, throughput and delay. Cell loss probability is defined as the fraction of cells lost within the switch. Cell loss might occur as a result of blocking and/or buffer overflows. Because cell re-transmission takes place on an end-to-end basis in ATM networks, and because of the high speeds involved in these networks, cell loss probability is considered as a very important performance measure. Throughput is defined as the average number of cells which are successfully delivered by the switch per time slot per input line. Switch delay is defined as the average time (in time slots) a cell spends from the time it arrives at an input port, till the time it is successfully delivered on its requested output line. It includes the time spent in any input, internal, and/or output buffers. Typically, an ATM switch is required to support a high throughput, a small delay and an extremely low cell loss probability.

In this Chapter the performance analysis of the VR switch is done to evaluate cell loss probability, throughput and delay. The performance of the VR switch is compared with other switch architectures based on input queueing, input smoothing, output queueing and completely shared buffering.

4.2 Performance Evaluation Methods

There are three methods for evaluating the performance of an ATM switch. They are

- Computer Simulation
- Theoretical Analysis
- Actual Hardware Evaluation

In computer simulation, performance analysis is done by writing programs in languages like C or Matlab based on the behavior of the switch. Hence it is easy to change the model and evaluate the performance under various conditions. However, the computing power is insufficient to evaluate cell loss ratios less than 10^{-6} [37]. Theoretical analysis requires analytical modeling by developing closed-form formulas and solving them to evaluate the performance, thus allowing simulation of very low cell loss ratios under some simplifying assumptions. Hardware evaluation needs a complete experimental set up with ATM switch prototype, cell generation unit, measurement unit and a controller. The controller is a workstation that sets routing and input traffic distribution data for the experimental system, manages the sequence of events, collects the data from the experimental system and thus evaluates the performance. Performance evaluation of the VR switch is done by computer simulation and theoretical analysis.

4.3 Computer Simulation of the VR Switch

Computer simulation is done to evaluate cell loss probability, throughput and delay in the VR switch. The assumptions made for simulation are

- Cell arrivals on the N inputs is modeled by independent and identical Bernoulli processes.
- In any given time slot, the probability that a cell will arrive on a particular input is p.
- Each cell has equal probability $1/N$ of being addressed to any given output port and their address is written in the output virtual queue of that port.
- The number of buffers per input port is b and the number of buffers per output port is d.

- Cell loss due to timeout is not considered i.e. a cell is not lost even if it is delayed by more than a certain given time.

The computer programme for simulation is written in C language and the simulation is run for a million time slots. The main parameters for simulation are the traffic load p, number of buffers per input port b, number of buffers per output port d and the switch size N. The uniform traffic used in simulation is generated by finding a random number between zero and one. If the number is less than p, the traffic load, then it is assumed that a cell has arrived.

4.3.1 Cell Loss Probability

Cell loss in the VR switch takes place through the following two mechanisms.

- An input buffer is full.
- An output virtual queue is full.

The cell loss probability for the VR switch as a function of output buffer size d for $N = 8$ and different values of b is shown in Figure 4.1. for traffic load $p = 0.70$ and $p = 0.90$ [36]. It is obvious that cell loss probability increases with traffic load. It is

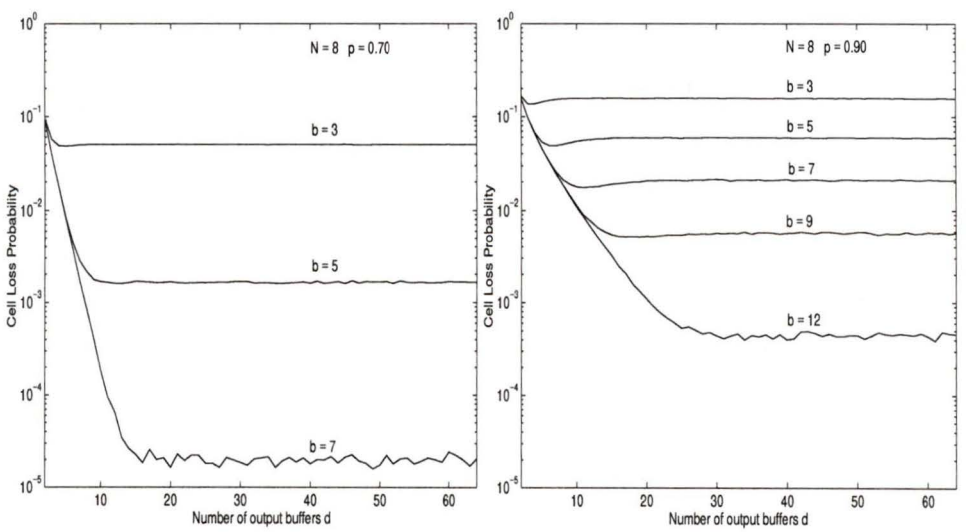


Figure 4.1. *Cell loss probability for the VR switch.*

seen that for a fixed number of input shift register buffers b, the cell loss probability decreases as the number of output buffers d is increased upto a limit (Figure 4.1). This is due to the fact that an increase in d reduces the cell loss due to output buffers by accommodating more addresses in their virtual queue. Cell loss due to the output buffers can be completely eliminated by having a sufficient number of output buffers. The maximal upper bound on the number of output buffers is easily seen to be Nb, and increasing d beyond Nb has no effect on cell loss probability.

Cell loss probability for the VR switch as a function of input buffer size b for different traffic loads p and switch sizes $N = 8$ and $N = 16$ is shown in Figure 4.2. The number of output buffers d is large enough so as not to cause any cell loss due

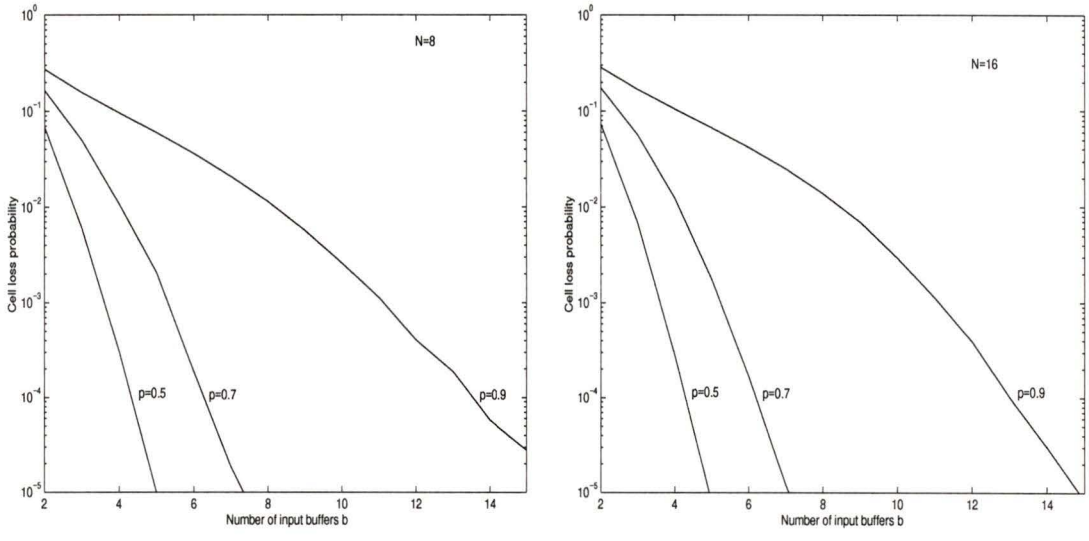


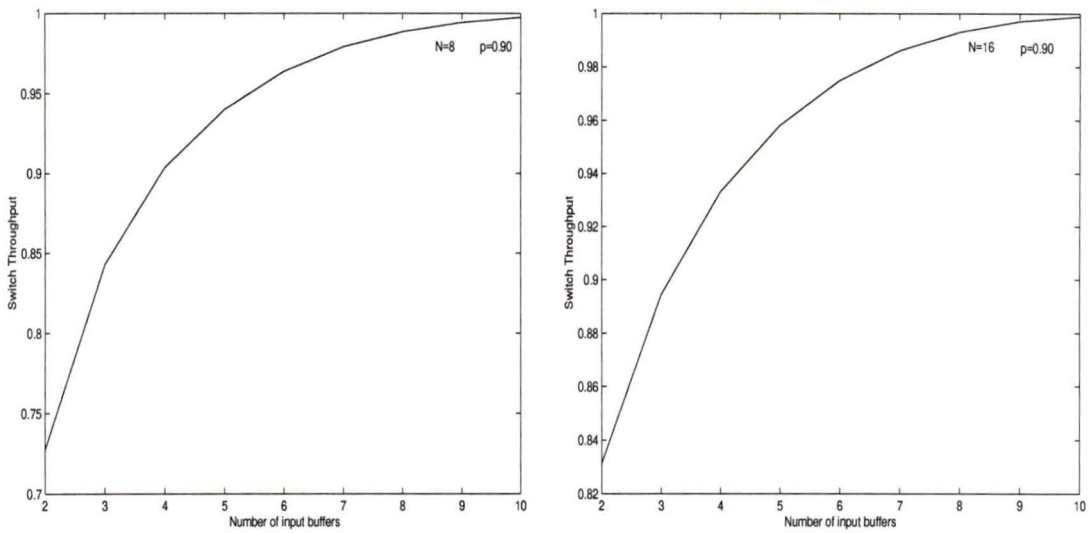
Figure 4.2. *Cell loss probability for the VR switch, by computer simulation. (a) $N = 8$ (b) $N = 16$.*

to output buffers being full. Since the simulation was run for one million time slots, cell loss probability only upto the order of 10^{-6} can be reached but results upto the order of 10^{-5} can be considered accurate and reliable. Hence, Figure. 4.2 only shows cell loss probability upto the order of 10^{-5}. It is seen from Figure 4.2 that cell loss probability decreases as the number of input buffers increases. Since an ATM switch is supposed to have very low cell loss probability, of the order of 10^{-12}, it is necessary

to find the number of input buffers required in the VR switch to obtain cell loss probability of 10^{-12}. Hence it is necessary to find cell loss probability in the VR switch by theoretical analysis.

4.3.2 Throughput

The throughput for the VR switch as a function of input buffer size b for traffic load $p = 90$ and switch sizes $N = 8$ and $N = 16$ is shown in Figure 4.3. It is seen from



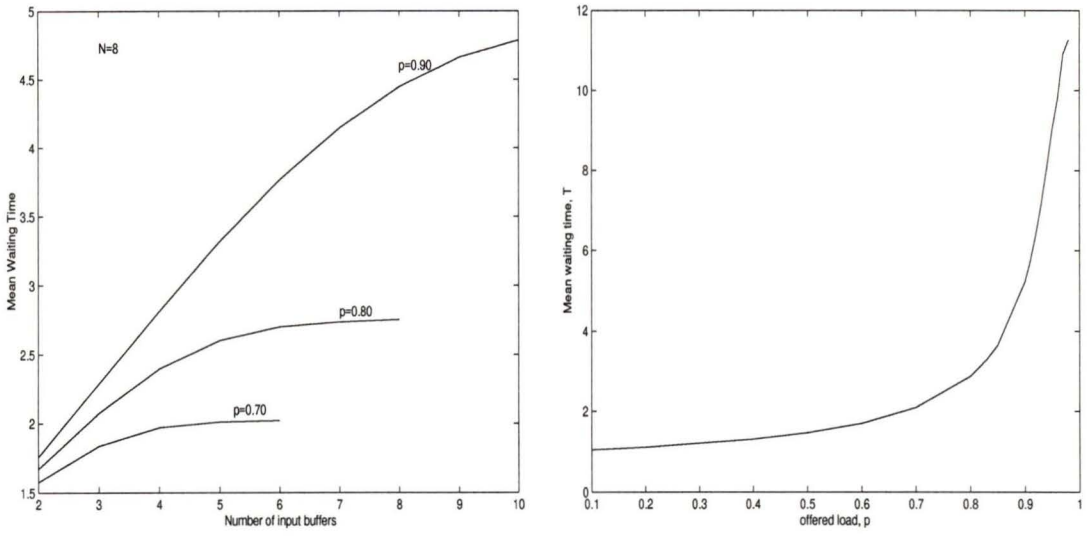
(a) (b)

Figure 4.3. *Throughput for the VR switch, by computer simulation. (a)* $N = 8, p = 0.90$ *(b)* $N = 16, p = 0.90$.

Figure 4.3 that as the number of input buffers is increased, the throughput of the VR switch approaches unity. Thus, if the number of input buffers in the VR switch is increased to the value which gives cell loss probability of 10^{-12} then 100% throughput can be achieved in the VR switch.

4.3.3 Delay

The delay or mean waiting time for the VR switch as a function of input buffer size b for switch size $N = 8$ and different traffic loads is shown in Figure 4.4 (a). It is seen that the mean waiting time increases as the number of input buffers is increased



(a) (b)

Figure 4.4. *Mean waiting time for the VR switch, by computer simulation. (a)* $N = 8$ *(b)* $N = 16$.

up to a limit and then it becomes constant. Thus, if the number of input buffers is large enough to have very low cell loss probability then for each traffic load there will be a constant mean waiting time. Figure 4.4 (b) shows the mean waiting time as a function of traffic load for switch size $N = 16$.

4.4 Theoretical Analysis of the VR Switch

In this section, the theoretical analysis of the VR switch is discussed to evaluate its performance in terms of cell loss probability, throughput and delay. The assumptions made in the theoretical analysis are the same as that of computer simulation.

4.4.1 Cell Loss Probability

In this analysis, we fix our attention on a particular (i.e. tagged) input port. In a given time slot, probability that a cell arrives in this input port is p. Let $q_k(k = 0, 1, 2, \ldots, b)$ be the probability that a particular input port has k cells in its shift register buffers. Thus, in a particular time slot, if a cell arrives in the input port which has b cells in its shift register buffers then the arriving cell will be lost because all the b shift

register buffers for that port are full. Thus a cell is lost in an input port if and only if

- A cell arrives in the input port (probability = p).
- Shift register buffers of that port are full (probability = q_b).

Thus

$$\text{Probability that a cell is lost in an input port} \;=\; p \cdot q_b$$

There are N input ports. Hence

$$\begin{aligned} \text{Average number of cells lost} &= N \cdot p \cdot q_b \\ \text{Average number of cells arrived} &= N \cdot p \end{aligned}$$

Therefore cell loss probability is given by

$$CLP \;=\; \frac{N \cdot p \cdot q_b}{N \cdot p} = q_b \tag{4.1}$$

In a particular time slot if the tagged input port has s cells ($s = 0, 1, 2, \ldots, b$), then the probability that out of these s cells, l cells ($l = 0, 1, 2, \ldots, s$) are transmitted out from that port is denoted by $a_{s,l}$. If in $(m-1)th$ time slot, an input port has i cells, then the probability that in mth time slot, j cells are left is denoted by state transition probability P_{ij}. Thus,

$$\begin{aligned} P_{0,0} &= 1-p \\ P_{0,1} &= p \\ P_{i,0} &= (1-p)a_{i,i} && 0 < i < b \\ P_{b,j} &= a_{b,b-j} && 0 \le j \le b \\ P_{i,j} &= pa_{i,i+1-j} + (1-p)a_{i,i-j} && 1 \le i \le b-1, 1 \le j \le i \\ P_{i,i+1} &= pa_{i,0} && 1 \le i < b \\ P_{i,j} &= 0 && 0 \le i \le b-2, i+2 \le j \le b \end{aligned}$$

4.4.1.1 Illustrative Example

Let us consider a 2×2 switch with two shift register buffers in each input port i.e. $b = 2$. Then

$$P_{00} \;=\; 1-p \tag{4.2}$$

$$P_{01} = p \tag{4.3}$$
$$P_{02} = 0 \tag{4.4}$$
$$P_{10} = (1-p)a_{1,1} \tag{4.5}$$
$$P_{11} = pa_{1,1} + (1-p)a_{1,0} \tag{4.6}$$
$$P_{12} = pa_{1,0} \tag{4.7}$$
$$P_{20} = a_{2,2} \tag{4.8}$$
$$P_{21} = a_{2,1} \tag{4.9}$$
$$P_{22} = a_{2,0} \tag{4.10}$$

Steady state probability will be given by

$$q_0 = P_{00}q_0 + P_{10}q_1 + P_{20}q_2 \tag{4.11}$$
$$q_1 = P_{01}q_0 + P_{11}q_1 + P_{21}q_2 \tag{4.12}$$
$$q_2 = P_{02}q_0 + P_{12}q_1 + P_{22}q_2 \tag{4.13}$$

Also

$$q_0 + q_1 + q_2 = 1 \tag{4.14}$$

Solving these equations iteratively we can calculate q_0, q_1 and q_2 provided $a_{1,0}$, $a_{1,1}$, $a_{2,0}$, $a_{2,1}$ and $a_{2,2}$ (the probabilities by which cells are transmitted out from the tagged input port) are known.

Now, it is necessary to calculate the probabilities $a_{i,j}$ by which cells are transmitted out from the tagged input port. For the 2×2 switch, let us call the two input ports as port A and port B and the two output ports as port C and port D. If the tagged input port (port A) has i cells then we need to find $a_{i,k}$, the probability that k cells are transmitted out from this port. Let

q_k = Probability that port B has k cells ($k = 0, 1, 2$).

A_{ikl} = Probability that l cells are transmitted out from port A through any output port, if port B has k cells in it, given port A has i cells in it.

CA_{ik} = Probability that address of k cells from port A are stored in port C

given port A has i cells.

CB_{ik} = Probability that address of k cells from port B are stored in port C given port B has i cells.

DA_{ik} = Probability that address of k cells from port A are stored in port D given port A has i cells.

DB_{ik} = Probability that address of k cells from port B are stored in port D given port B has i cells.

The probabilities $CA_{ik}, CB_{ik}, DA_{ik}$ and DB_{ik} are

$$= \begin{pmatrix} i \\ k \end{pmatrix} (\frac{1}{N})^k (1 - \frac{1}{N})^{i-k} \tag{4.15}$$

$$\tag{4.16}$$

Thus,

$$a_{1,0} = q_0 \cdot A_{100} + q_1 \cdot A_{110} + q_2 \cdot A_{120} \tag{4.17}$$

$$a_{1,1} = q_0 \cdot A_{101} + q_1 \cdot A_{111} + q_2 \cdot A_{121} \tag{4.18}$$

$$a_{2,0} = q_0 \cdot A_{200} + q_1 \cdot A_{210} + q_2 \cdot A_{220} \tag{4.19}$$

$$a_{2,1} = q_0 \cdot A_{201} + q_1 \cdot A_{211} + q_2 \cdot A_{221} \tag{4.20}$$

$$a_{2,2} = q_0 \cdot A_{202} + q_1 \cdot A_{212} + q_2 \cdot A_{222} \tag{4.21}$$

where

$$\begin{aligned} A_{100} &= 0 \\ A_{110} &= CA_{11} \cdot CB_{11} \cdot DA_{10} \cdot DB_{10} \cdot P_{BC} + CA_{10} \cdot CB_{10} \cdot DA_{11} \cdot DB_{11} \cdot P_{BD} \\ &= \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \\ &= \frac{1}{4} \\ A_{120} &= CA_{11} \cdot CB_{22} \cdot DA_{10} \cdot DB_{20} \cdot P_{BC} + CA_{11} \cdot CB_{21} \cdot DA_{10} \cdot DB_{21} \cdot P_{BC} + \\ &\quad CA_{10} \cdot CB_{21} \cdot DA_{11} \cdot DB_{21} \cdot P_{BD} + CA_{10} \cdot CB_{20} \cdot DA_{11} \cdot DB_{22} \cdot P_{BD} \\ &= \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{3} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} + \frac{1}{2} + \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{3} \end{aligned}$$

$$
\begin{aligned}
&= \frac{5}{12} \\
A_{101} &= CA_{11} \cdot CB_{00} \cdot DA_{10} \cdot DB_{00} \cdot P_{AC} + CA_{10} \cdot CB_{00} \cdot DA_{11} \cdot DB_{00} \cdot P_{AD} \\
&= \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 \cdot 1 + \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\
&= 1 \\
A_{111} &= CA_{11} \cdot CB_{11} \cdot DA_{10} \cdot DB_{10} \cdot P_{AC} + CA_{11} \cdot CB_{10} \cdot DA_{10} \cdot DB_{11} \cdot P_{AC} + \\
&\quad CA_{10} \cdot CB_{11} \cdot DA_{11} \cdot DB_{10} \cdot P_{AD} + CA_{10} \cdot CB_{10} \cdot DA_{11} \cdot DB_{11} \cdot P_{AD} \\
&= \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \\
&= \frac{3}{4} \\
A_{121} &= CA_{11} \cdot CB_{22} \cdot DA_{10} \cdot DB_{20} \cdot P_{AC} + CA_{11} \cdot CB_{21} \cdot DA_{10} \cdot DB_{21} \cdot P_{AC} + \\
&\quad CA_{11} \cdot CB_{20} \cdot DA_{10} \cdot DB_{22} \cdot P_{AC} + CA_{10} \cdot CB_{22} \cdot DA_{11} \cdot DB_{20} \cdot P_{AD} + \\
&\quad CA_{10} \cdot CB_{21} \cdot DA_{11} \cdot DB_{21} \cdot P_{AD} + CA_{10} \cdot CB_{20} \cdot DA_{11} \cdot DB_{22} \cdot P_{AD} \\
&= \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 + \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 + \\
&\quad \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{1}{3} \\
&= \frac{7}{12} \\
A_{200} &= 0 \\
A_{210} &= CA_{22} \cdot CB_{11} \cdot DA_{20} \cdot DB_{10} \cdot P_{BC} + CA_{20} \cdot CB_{10} \cdot DA_{22} \cdot DB_{11} \cdot P_{BD} \\
&= \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{3} + \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{3} \\
&= \frac{1}{12} \\
A_{220} &= CA_{22} \cdot CB_{22} \cdot DA_{20} \cdot DB_{20} \cdot P_{BC} + CA_{22} \cdot CB_{21} \cdot DA_{20} \cdot DB_{21} \cdot P_{BC} + \\
&\quad CA_{21} \cdot CB_{21} \cdot DA_{21} \cdot DB_{21} \cdot P_{BC} \cdot P_{BD} + CA_{20} \cdot CB_{21} \cdot DA_{22} \cdot DB_{21} \cdot P_{BD} + \\
&\quad CA_{20} \cdot CB_{20} \cdot DA_{22} \cdot DB_{22} \cdot P_{BD} \\
&= \frac{1}{4} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{4} + \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{3} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{3} + \\
&\quad \frac{1}{4} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{4} \\
&= \frac{10}{48} \\
A_{201} &= CA_{22} \cdot CB_{00} \cdot DA_{20} \cdot DB_{00} \cdot P_{AC} + CA_{20} \cdot CB_{00} \cdot DA_{22} \cdot DB_{00} \cdot P_{AD} \\
&= \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 \cdot 1 + \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 \cdot 1
\end{aligned}
$$

$$
\begin{aligned}
&= \frac{1}{2} \\
A_{211} &= CA_{22} \cdot CB_{11} \cdot DA_{20} \cdot DB_{10} \cdot P_{AC} + CA_{22} \cdot CB_{10} \cdot DA_{20} \cdot DB_{11} \cdot P_{AC} + \\
&\quad CA_{21} \cdot CB_{11} \cdot DA_{21} \cdot DB_{10} \cdot P_{BC} \cdot P_{AD} + CA_{21} \cdot CB_{10} \cdot DA_{21} \cdot DB_{11} \cdot P_{AC} \cdot \\
&\quad P_{BD} + CA_{20} \cdot CB_{11} \cdot DA_{22} \cdot DB_{10} \cdot P_{AD} + CA_{20} \cdot CB_{10} \cdot DA_{22} \cdot DB_{11} \cdot P_{AD} \\
&= \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{2}{3} + \frac{1}{4} \cdot \frac{1}{2} \cdot \cdot 1 \cdot 1 \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 \cdot \frac{1}{2} + \\
&\quad \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 + \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{2}{3} \\
&= \frac{2}{3} \\
A_{221} &= CA_{22} \cdot CB_{22} \cdot DA_{20} \cdot DB_{20} \cdot P_{AC} + CA_{22} \cdot CB_{21} \cdot DA_{20} \cdot DB_{21} \cdot P_{AC} + \\
&\quad CA_{22} \cdot CB_{20} \cdot DA_{20} \cdot DB_{22} \cdot P_{AC} + CA_{21} \cdot CB_{22} \cdot DA_{21} \cdot DB_{20} \cdot P_{BC} \cdot P_{AD} + \\
&\quad CA_{21} \cdot CB_{21} \cdot DA_{21} \cdot DB_{21} \cdot P_{AC} \cdot P_{BD} + CA_{21} \cdot CB_{21} \cdot DA_{21} \cdot DB_{21} \cdot P_{BC} \cdot \\
&\quad P_{AD} + CA_{21} \cdot CB_{20} \cdot DA_{21} \cdot DB_{22} \cdot P_{AC} \cdot P_{BD} + CA_{20} \cdot CB_{22} \cdot DA_{22} \cdot DB_{20} \cdot \\
&\quad P_{AD} + CA_{20} \cdot CB_{21} \cdot DA_{22} \cdot DB_{21} \cdot P_{AD} + CA_{20} \cdot CB_{20} \cdot DA_{22} \cdot DB_{22} \cdot P_{AD} \\
&= \frac{1}{4} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{4} + \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{2}{3} + \frac{1}{4} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 + \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{3} \cdot 1 + \\
&\quad \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 \cdot \frac{2}{3} + \frac{1}{4} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 + \\
&\quad \frac{1}{4} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{2}{3} + \frac{1}{4} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{2}{4} \\
&= \frac{31}{48} \\
A_{202} &= CA_{21} \cdot CB_{00} \cdot DA_{21} \cdot DB_{00} \cdot P_{AC} \cdot P_{AD} \\
&= \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \\
&= \frac{1}{2} \\
A_{212} &= CA_{21} \cdot CB_{11} \cdot DA_{21} \cdot DB_{10} \cdot P_{AC} \cdot P_{AD} + CA_{21} \cdot CB_{10} \cdot DA_{21} \cdot DB_{11} \cdot P_{AC} \cdot \\
&\quad P_{AD} \\
&= \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot 1 \cdot \frac{1}{2} \\
&= \frac{1}{4} \\
A_{222} &= CA_{21} \cdot CB_{22} \cdot DA_{21} \cdot DB_{20} \cdot P_{AC} \cdot P_{AD} + CA_{21} \cdot CB_{21} \cdot DA_{21} \cdot DB_{21} \cdot P_{AC} \cdot \\
&\quad P_{AD} + CA_{21} \cdot CB_{20} \cdot DA_{21} \cdot DB_{22} \cdot P_{AC} \cdot P_{AD} \\
&= \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot \frac{1}{3} \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{4} \cdot 1 \cdot 1 \cdot 1 \cdot \frac{1}{3}
\end{aligned}
$$

$$= \frac{7}{48}$$

Thus we have

$$a_{1,0} = \frac{1}{4}q_1 + \frac{5}{12}q_2 \tag{4.22}$$

$$a_{1,1} = q_0 + \frac{3}{4}q_1 + \frac{7}{12}q_2 \tag{4.23}$$

$$a_{2,0} = \frac{1}{12}q_1 + \frac{10}{48}q_2 \tag{4.24}$$

$$a_{2,1} = \frac{1}{2}q_0 + \frac{2}{3}q_1 + \frac{31}{48}q_2 \tag{4.25}$$

$$a_{2,2} = \frac{1}{2}q_0 + \frac{1}{4}q_1 + \frac{7}{48}q_2 \tag{4.26}$$

Substituting equations (4.22)-(4.26) in equations (4.2)-(4.10) and then solving equations (4.2)-(4.14) iteratively we can calculate q_0, q_1 and q_2 for different values of traffic load p. Cell loss probability is given by q_2. Table. 4.1. compares the calculated cell loss probability with the cell loss probability obtained by simulation for $N = 2$ and different traffic load p.

Traffic Load p	CLP by simulation	CLP calculated
0.3	0.00785	0.007828
0.5	0.03789	0.03746
0.7	0.09950	0.09720
0.9	0.183910	0.178561

Table 4.1. *Comparision of CLP*

4.4.1.2 General Approach

In the previous section recursive equations are developed to find cell loss probability for a 2×2 switch with 2 shift register buffers in each input port. With this approach, the calculated cell loss probability is approximately the same as the one obtained by

simulation. But this approach is too complicated to find the cell loss probability in general, for any number of shift register buffers in each input port and for any size of the switch. Hence it is necessary to follow a different approach [38].

Let us define the random variable A as the number of cells arriving at the switch inputs and destined for a particular output in a given time slot. Then A^i denotes the number of cell arrivals destined for output port i. For finite N, A^i depends on $A^j (j \neq i)$. This is due to the fact that at most N cells arrive to the switch, a large number of cells arriving for one output implies a small number for the remaining outputs. As N increases, A^i becomes an independent Poisson random variable (with mean value p). We will use the Poisson and independence assumptions even for finite N [34]. Thus the probability that k cells are destined to an output port is given by the Poisson distribution

$$a_k = Pr[A = k] = \frac{p^k e^{-p}}{k!} \quad k = 0, 1, 2, \ldots \tag{4.27}$$

Let us define another random variable Q as the steady-state number of cell addresses stored in the virtual queue of a particular output port at the end of a given time slot. Then Q^i denotes the steady-state number of cell addresses stored in the virtual queue of output port i. The virtual queue of each output port is modeled as an $M/D/1$ queue for which the steady-state queue size probabilities are obtained directly from the balance equations for the Markov chain shown in Figure 4.5. Equations (4.28)-

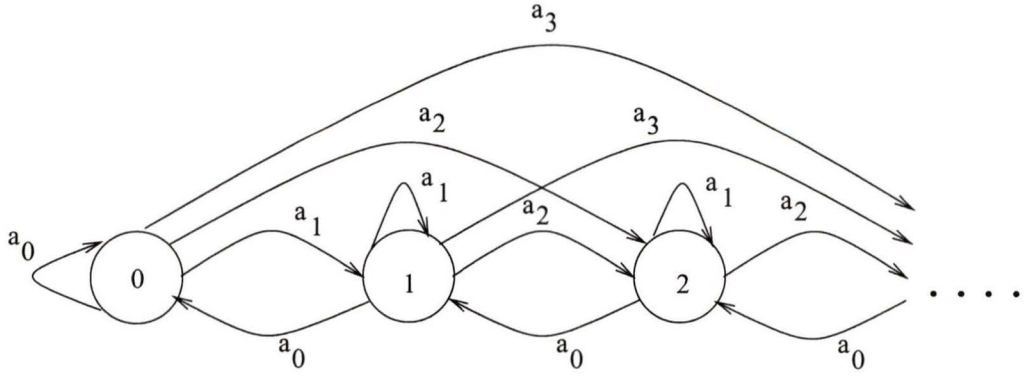


Figure 4.5. *The discrete-time Markov chain state transition diagram for the output queue size.*

(4.31) numerically provide the steady-state queue size probabilities [1].

$$q_0 = Pr(Q=0) = 1-p \tag{4.28}$$

$$q_1 = Pr(Q=1) = \frac{(1-a_0)}{a_0} \cdot q_0 \tag{4.29}$$

$$q_2 = Pr(Q=2) = \frac{(1-a_1)}{a_0} \cdot q_1 - \frac{a_1}{a_0} \cdot q_0 \tag{4.30}$$

$$\vdots$$

$$q_n = Pr(Q=n) = \frac{(1-a_1)}{a_0} \cdot q_{n-1} - \sum_{i=2}^{N} \frac{a_i}{a_0} \cdot q_{n-i} - \frac{a_{n-1}}{a_0} \cdot q_0 \quad n > 2 \tag{4.31}$$

Let us define the random variable S as the steady-state total number of cell addresses stored in output. Then

$$S = \sum_{i=1}^{N} Q^i \tag{4.32}$$

The steady-state probability that the total number of cell addresses stored in output is k is given by

$$s_k = Pr(S=k) = Pr(\sum_{i=1}^{N} Q^i = k) \tag{4.33}$$

The steady-state probability s_k can be calculated by the N-fold convolution of $NM/D/1$ queues [34].

Let us assume that each input port has infinite buffers. However, if an input port has more than b cells, it is evident that cell loss will take place because each input port actually has b buffers only. Let P_0 denote the probability that no cell is lost and P_{0k} denotes the probability that no cell is lost when k cell addresses are stored in the outputs. Then probability that no cell is lost is given by

$$P_0 = \sum_{k=0}^{\infty} \text{Prob.[no cell is lost} \mid k \text{ cell addresses stored in outputs]} \cdot \text{Prob.[}k \text{ cell addresses stored in outputs]}$$

$$P_0 = \sum_{k=0}^{\infty} P_{0k} \cdot s_k \tag{4.34}$$

[1] The steady-state queue size probabilities in Eqns. (13)-(15) of reference [39] are for the case when cell flows through the switch without suffering any delay i.e. a cell arriving in a given time slot may be transmitted out in the same time slot. In our case there is a minimum delay of one time slot. An arriving cell is not transmitted out in the same time slot and hence the modifications to (4.28)-(4.31).

If CLP denotes the cell loss probability in the switch then

$$CLP = 1 - P_0 \tag{4.35}$$

If the total number of cell addresses in the output is less than or equal to b then there will be no cell loss. This is because, in the worst case, all the addresses stored in the outputs may be due to cells stored in one input port. Since each input port has b shift register buffers to store cells, there will be no cell loss. If the total number of cell addresses in the output is k, where k is greater than b but less than or equal to Nb, then cells may be lost due to one or more input ports being full. In this case, if P_{Fl} denote the probability that l input ports are full then

$$\begin{aligned} P_{Fl} &= (\text{Prob.[more than } b \text{ cells are stored in a par ticular input port]})^l \cdot \text{Prob.[choose} \\ &\quad l \text{ ports from } N \text{ ports]} \\ P_{Fl} &= (r_k)^l \cdot I_l \end{aligned} \tag{4.36}$$

But probability that more than b cells are stored in a particular input port is given by

$$\begin{aligned} r_k &= 1 - \sum_{l=0}^{b} \text{Prob.}[l \text{ cells are stored in a particular input port}] \\ &= 1 - \sum_{l=0}^{b} a_{kl} \qquad\qquad b+1 \le k \le Nb \end{aligned} \tag{4.37}$$

where a_{kl} denotes the probability that the input port contains l cells out of a total of k cells. But probability that l cells are stored in a particular input port is given by

$$a_{kl} = \binom{k}{l} (\frac{1}{N})^l (1 - \frac{1}{N})^{k-l} \quad \begin{cases} b < k \le Nb \\ 0 \le l \le b \end{cases} \tag{4.38}$$

Also, the probability of selecting l input ports out of N ports is given by

$$I_l = \binom{N}{l} (\frac{1}{N})^l (1 - \frac{1}{N})^{N-l} \quad l = 0, 1, \ldots, N \tag{4.39}$$

Thus, probability that none of the input ports are full if k cell addresses are stored in the output is given by

$$\begin{aligned} P_{0k} &= 1 - \sum_{l=1}^{v} \text{Prob.}[l \text{ ports are full}] \\ P_{0k} &= 1 - \sum_{l=1}^{v} I_l \cdot (r_k)^l \quad \begin{cases} v = k \bmod b \\ b < k \le Nb \end{cases} \end{aligned} \tag{4.40}$$

Also, if the total number of cell addresses in the output is greater than Nb then it is sure that there is cell loss. This is because, in the best case, all the addresses stored in the output may be of the cells which are equally divided in all input ports. Even in this case, it is sure that shift register buffers of at least one port is full. Thus

$$P_{0k} = \begin{cases} 1 & 0 \leq k \leq b \\ 1 - \sum_{l=1}^{v} I_l \cdot (r_k)^l & \begin{cases} v = k \bmod b \\ b < k \leq Nb \end{cases} \\ 0 & k > Nb \end{cases} \tag{4.41}$$

By using equations (4.27)-(4.41), cell loss probability in the VR switch is given by

$$CLP = 1 - \sum_{k=0}^{Nb} P_{0k} \cdot s_k \tag{4.42}$$

Thus, this approach gives cell loss probability in general, for any number of shift register buffers in each input port and for any size of the switch. This approach does not give accurate results for smaller number of shift register buffers but as the number of buffers increases the result becomes accurate. Thus, this approach gives good results in the region of interest: the low cell loss probability region (e.g., less than 10^{-6} cell loss probability).

4.4.2 Throughput

In output buffered switches a cell is always transmitted out if the output queue is not empty. Similarly, in the VR switch, one address from the virtual queue of each output port is selected and the corresponding cell from input buffer is transmitted out through cell switch fabric. Thus a cell will always be transmitted from each output port if the output virtual queue is not empty. Thus similar to output buffered switches, an optimal throughput is achieved in this ATM switch.

If ρ_0 denote the throughput of the VR switch then it is given by [34], [40].

$$\rho_0 = p(1 - CLP) \tag{4.43}$$

The normalized throughput R is given by

$$R = \frac{\rho_0}{p} \tag{4.44}$$

$$= (1 - CLP) \tag{4.45}$$

Thus, the switch throughput can be studied easily once the switch CLP is determined.

4.4.3 Delay

In the VR switch, one address from the virtual queue of each output port is selected and the corresponding cell from input buffer is transmitted out. This cell address is selected from the virtual queue of each output port on a First-Come-First-Serve (FCFS) basis in order to preserve the cell sequence. Thus, the virtual queue of each output port is a FIFO buffer. Hence, the mean waiting time (T) for a cell in an output FIFO is obtained from Little's result and is given by

$$T = \frac{C_{avg}}{\rho_0} \tag{4.46}$$

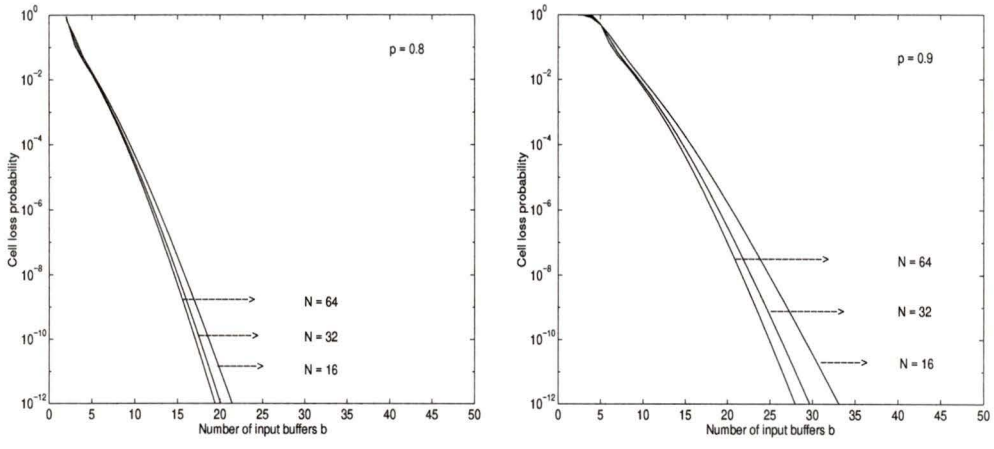
where C_{avg} is the average number of cell addresses present in the output FIFO virtual queue and is given by

$$C_{avg} = \sum_{n=1}^{Nb} n \cdot q_n \tag{4.47}$$

Thus, the delay is studied in the VR switch.

4.5 Results

The recursive formulas developed in the previous section are programmed in MAPLE, which has arbitrary precision, to obtain the final results for cell loss probability, throughput and delay. The results are discussed in this section. Figure 4.6 shows the cell loss probability for different values of N and traffic load of 0.8 and 0.9.



(a) (b)

Figure 4.6. *Cell loss probability for the VR switch. (a) Traffic load* = 0.80. *(b) Traffic load* = 0.90.

The throughput of the VR switch is shown in Figure 4.7 for $N = 16$ and traffic load $= 0.90$. Figure 4.8 shows the mean waiting time for different traffic loads. The

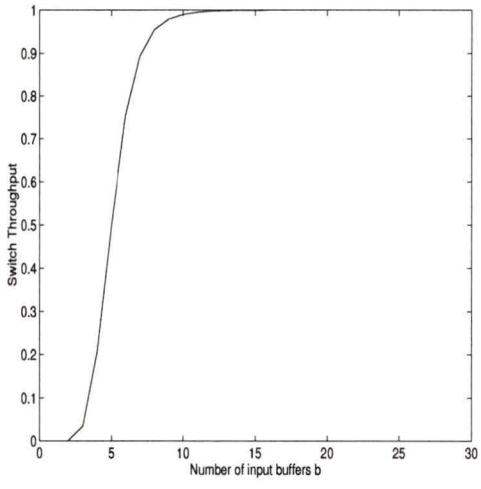


Figure 4.7. *Throughput for the VR switch,* $N = 16$, $p = 0.90$.

number of input buffers b is 35, large enough to have very small cell loss probability. It can be seen that there is a minimum delay of one time slot in the VR switch.

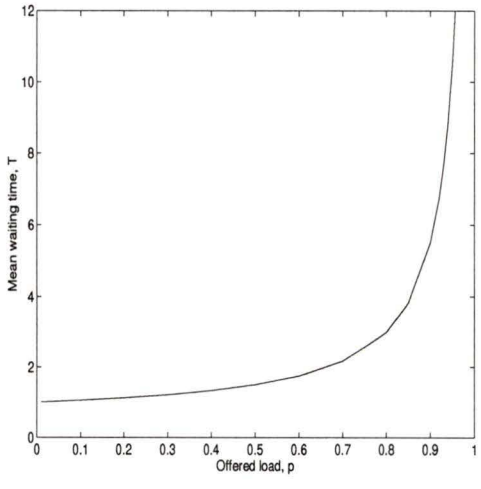


Figure 4.8. *Mean waiting time for the VR switch,* $N = 16$, $b = 35$.

4.6 Comparision with other Switch Architectures

The performance of the VR switch is compared with the performance of switches based on input queueing, input smoothing, output queueing and completely shared buffering. Figure 4.9 shows a comparison of cell loss probabilities for switch size $N = 16$ and a uniform traffic load of 0.90. It can be seen that for same number of

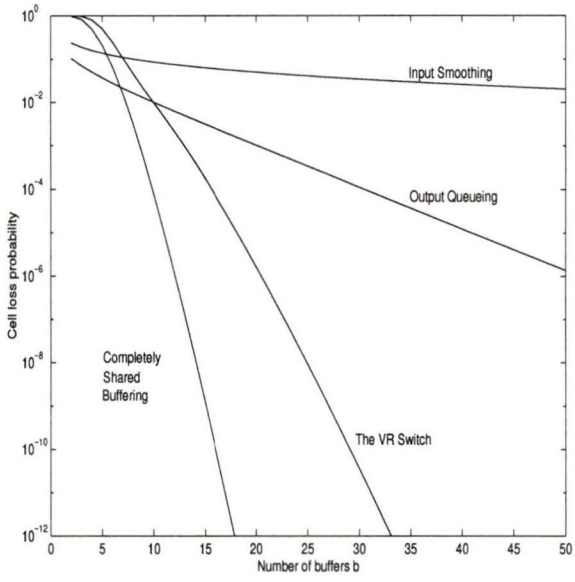


Figure 4.9. *A comparison of cell loss probabilities,* $N = 16$, $p = 0.90$.

buffers, the VR switch has lower cell loss probability as compared to input smoothing and output queueing switches. One reason for lower cell loss probability in this switch as compared to output queueing is that in output queueing at most N cells can enter the buffer in a particular time slot and only one cell can go out in that time slot. However, in the VR switch only one cell enter the buffer in a particular time slot and at most N cells can go out in that time slot. The cell loss probability is much lower in case of completely shared buffering switches. A comparison of mean waiting times for the case $N = \infty$ is shown in Figure 4.10 for input smoothing, input queueing, output queueing and completely shared buffering [34]. From Figure 4.10, it is seen that input queueing does not have an optimal throughput/delay performance. Its throughput is limited to 0.586. Though the VR switch is also an input queued switch it does not suffer from low throughput/delay problems because it overcomes

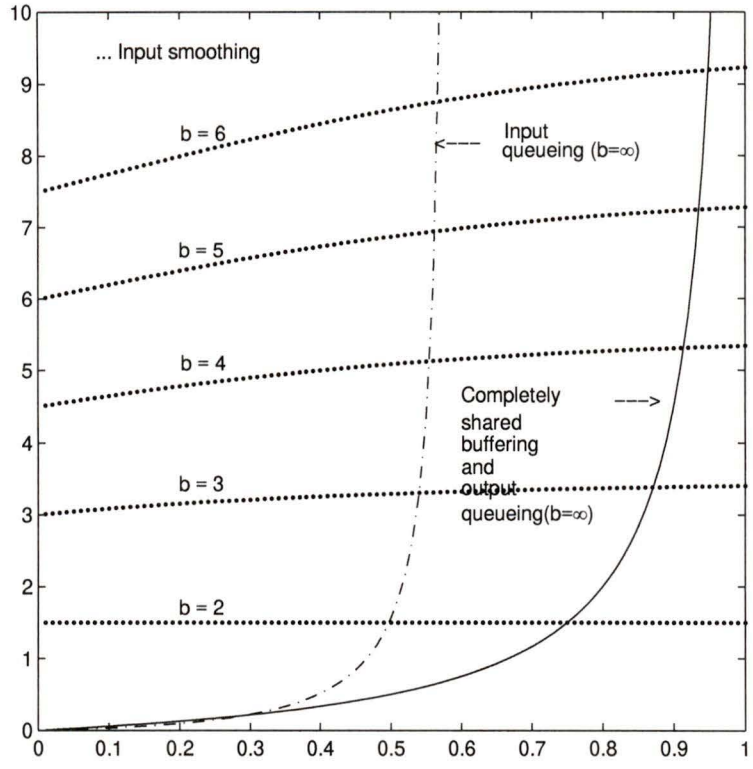


Figure 4.10. *A comparison of mean waiting times for the case* $N = \infty$

head-of-line blocking. In the VR switch, no output port remains idle. If there is a cell destined for a particular output port then it is always routed. Thus an optimal throughput performance is achieved in the VR switch (as shown in Figure 4.7). The mean waiting time for the VR switch is very close to the mean waiting time of switches based on completely shared buffering and output queueing, however, there is a minimum delay of one time slot (as shown in Figure 4.8).

It is to be noted that the number of inputs/outputs grow as Nb in input smoothing switches and as $N(b+1)$ in completely shared buffering switches [34]. However, in the VR switch only the number of inputs grow as Nb.

4.7 Summary

The performance analysis of the VR switch is done in terms of cell loss probability, throughput and delay. Lower cell loss probabilities of the order of 10^{-12} can not be determined by computer simulation. Hence, analytical modeling is done to evaluate cell loss probability, throughput and delay. The recursive formulas developed are then programmed in MAPLE, which has arbitrary precision, to get the final values. The performance is compared with other traditional queueing approaches in ATM switch. It is observed that to achieve desired cell loss probability, less number of input shift register buffers is required in the VR switch. Also, an optimal throughput-delay performance is achieved in this switch.

Chapter 5

Hardware Details of The VR Switch

5.1 Introduction

The design and operation of the VR switch is discussed in Chapter 3. It is seen that the main modules of the VR switch are the input module, switching fabric and the output module.

This Chapter presents the hardware details of the input module, switching fabric and the output module. It discusses how Quality of Service (QoS) and multicast and broadcast functions are easily supported by the VR switch. The scalability of the VR switch is also discussed.

The logic of the design is verified by behavioral/structural modeling in VHDL. The VHDL code for the VR switch is imported in SYNOPSYS to get the gate-level implementation.

5.2 Input Module

The input module consists of N input ports for a switch of size N. Figure 5.1 shows block diagram of an input port.

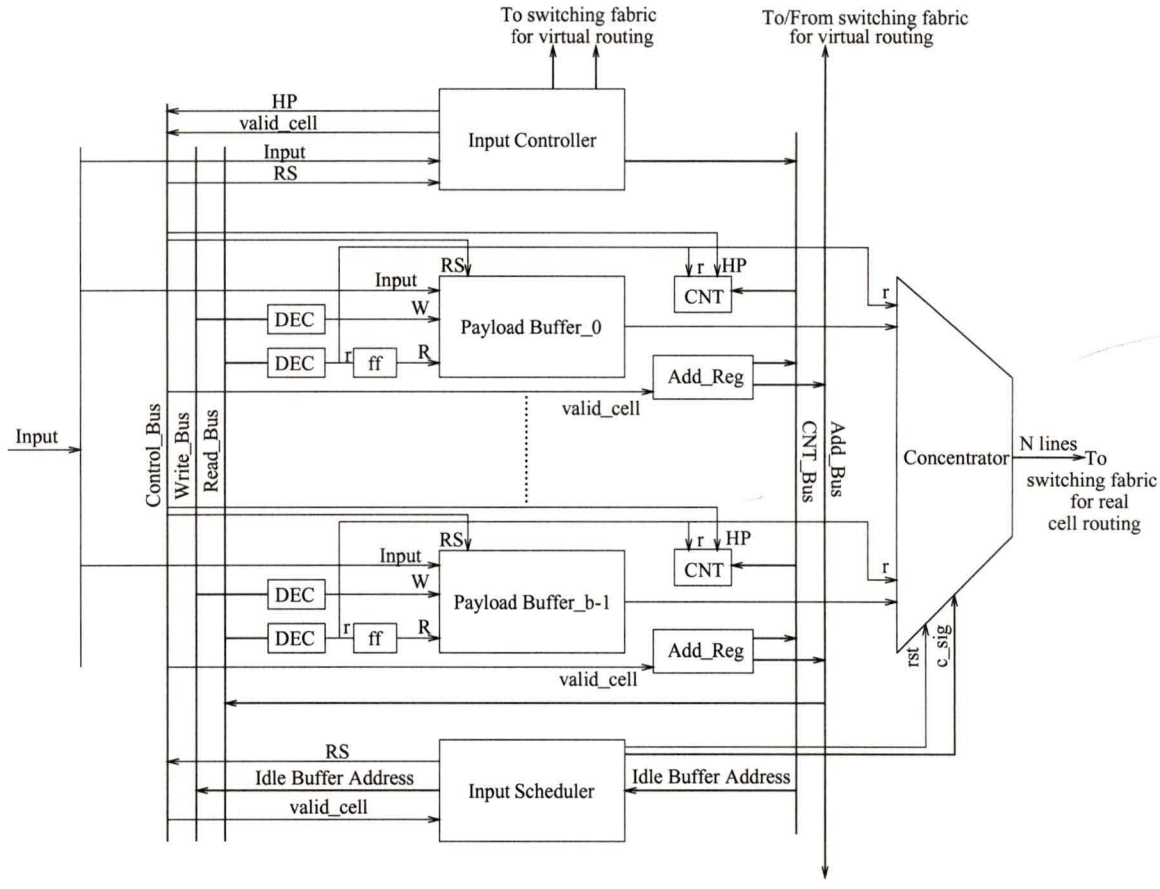


Figure 5.1. *Block diagram of an input port.*

The input port consists of b buffers, input controller, input scheduler and b to N concentrator. The input scheduler keeps updated information of empty buffers. Thus an arriving cell is stored in the empty buffer. Cell header processing is done in input controller. The concentrator connects each buffer to the switching module.

As shown in Figure 5.1 the input scheduler places the idle buffer address on write_bus. The decoders decode this address and thus write signal (W) is asserted at one of the payload buffers. A cell arrives and is stored in the corresponding payload buffer. The payload buffer is a shift register buffer and stores only the payload of the cell. The cell is simultaneously received by the input controller and is stored in a buffer until the input scheduler sends a signal RS to it. Thus only the header of the cell is received by the input controller. The input controller processes the cell header

and determines the type of the cell, it's call number and the number of destination ports. After cell header processing indicated by signal HP, the input controller places the number of destination ports on the CNT_bus which is received and stored in the counter CNT of the buffer which is still receiving the payload of the cell. Now, the call number of the cell and 12 bits of cell header (received from input controller) and the address of shift register buffer where the cell is stored (received from corresponding register Add_Reg after it gets signal valid_cell from input controller) is sent to the output module through *switching fabric for virtual address routing* to update the output virtual queues. The input controller also sends the signal valid_cell to input scheduler to inform that a valid cell has been received and stored in the idle buffer. After receiving signal valid_cell, the input scheduler updates its information of empty shift register buffers.

Virtual address routing to establish connections is done in the following way. Each output port sends the address at the head of its queue through the *switching fabric for virtual address routing.* Thus the addresses of the cells which are to be routed are received by input port, one by one, in succession, through Add_bus. These addresses are decoded by decoders and a read signal (R) is asserted at the corresponding shift register buffers. At the same time signal r is sent to the corresponding counter CNT. On receiving this signal the counter CNT decrements its reading by one. It is to be noted here that the read signal received from the decoder toggles between zero and one. Hence a flip-flop is used so that the shift register remains in the read mode, once the flip-flop receives the read signal. When the flip-flop is reset the shift register buffer no longer remains in the read mode. Signal r is also sent to the concentrator to establish the necessary connections. Thus, cell is routed out from all those shift register buffers which are in the read mode through *switching fabric for real cell routing.* Also, the data is re-circulated in the buffer to support multicast and broadcast functions. However, if the reading on the counter becomes zero then it sends a signal to the corresponding register Add_Reg which in turn places its address on the CNT_bus. This address is then received by the input scheduler to keep updated information of idle shift register buffers. Thus, real cell routing is achieved.

5.2.1 Input Scheduler

The block diagram of input scheduler is shown in Figure 5.2. The input scheduler

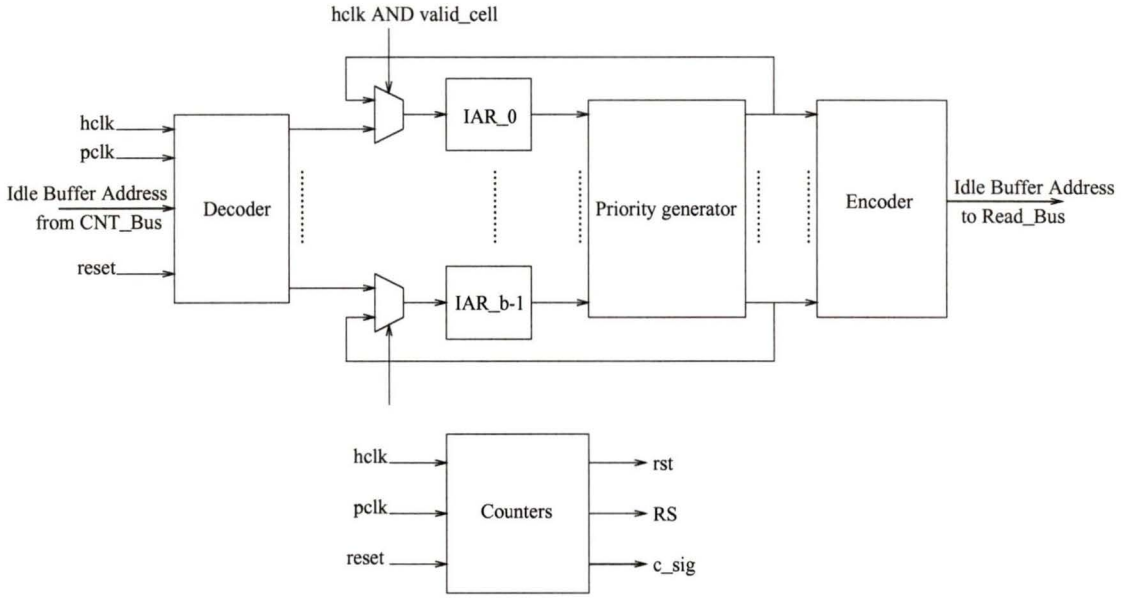


Figure 5.2. *Block diagram of an input scheduler.*

consists of consists of decoder, idle address registers (IAR), priority generator and encoder. Each IAR corresponds to a particular shift register buffer and hence the number of idle address registers is equal to the number of shift register buffers. Each IAR stores a '1' or '0' in it. A '1' indicates that the corresponding shift register buffer is empty whereas a '0' indicates that the corresponding shift register buffer is full. The decoder sends a '1' or '0' to each IAR based on the idle buffer address received from CNT_Bus. At reset, the decoder sends a '1' to all the IARs. Thus, all the idle address registers have stored '1' initially. This is due to the fact that initially, all the shift register buffers are empty. The priority generator looks for a '1' in the IARs and gives priority to the IARs in increasing order. Thus the output of the priority generator corresponding to the first IAR which has a '1' in it, is '1'. All other outputs of the priority generator are '0'. The encoder receives the output of the priority generator and sends idle buffer address to the Read_Bus. Thus, the input scheduler keeps updated information of empty shift register buffers. The input scheduler also consists of counters which generate signals rst, RS and c_sig used in the input port.

5.2.2 Input Controller

The block diagram of the input controller is shown in Figure 5.3. The input controller

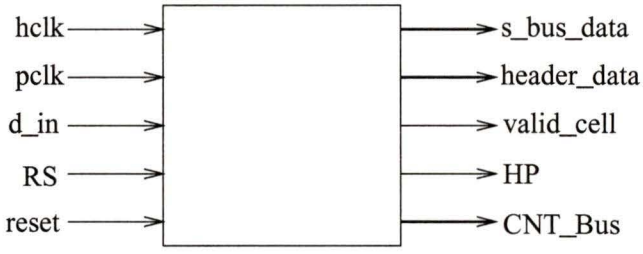


Figure 5.3. *Block diagram of input controller.*

receives only the header of the cell. Cell header processing is done in input controller. After cell header processing the input controller sends HP, valid_cell and cnt_bus to other modules of input port. It also sends s_bus_data and header_data to the output module through switching fabric for virtual address routing. Signal HP indicates that cell header processing is done whereas signal valid_cell indicates that a valid_cell has arrived. The data on cnt_bus is the number of destination ports of the cell arrived. The s_bus_data contains the call number of the cell whereas the header_data consists of 12 bits of cell header. The input controller consists of header buffer, cell sort module, route_table module and traffic module.

The block diagram of header buffer is shown in Figure 5.4. The header buffer

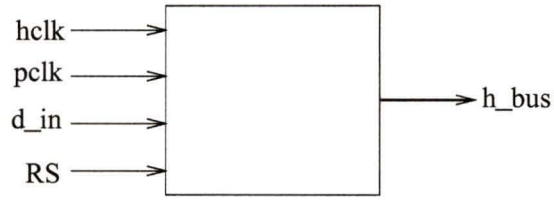


Figure 5.4. *Block diagram of header buffer.*

receives the cell until it gets signal RS from input scheduler. Thus only the header of the cell is stored in the header buffer.

The block diagram of cell sort module is shown in Figure 5.5. The cell sort module

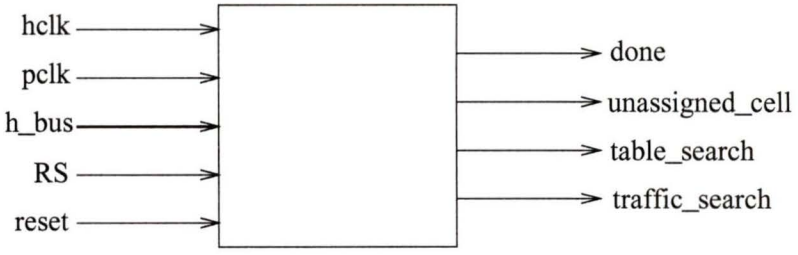


Figure 5.5. *Block diagram of cell sort module.*

loads the first 32 bits of cell header from header buffer after receiving signal RS from input scheduler. The cell sort module makes decision based on the four octects of the header. The last octect is neglected since it is used for header error control (HEC) in the physical layer. The four octects of header contain preassigned cell header values at UNI shown in Table. 5.1. and defined payload types shown in Table 5.2.

Type	Octect 1	Octect 2	Octect 3	Octect 4
Unassigned	00000000	00000000	00000000	0000xxx0
Signalling	00000000	00000000	00000000	01010aac
Segment VP OAM	0000aaaa	aaaa0000	00000000	00110a0a
End-to-End VP OAM	0000aaaa	aaaa0000	00000000	01000a0a
ILMI	00000000	00000000	00000001	0000aaa0

Table 5.1 ATM pre-defined headers

"a" indicates the bit is available for ATM functions

"x" is don't care bit

"c" means sender sets CLP = 0, but may be changed in network

PT code	Meaning
000	User data cell, congestion not experienced, AAI = 0*
001	User data cell, congestion not experienced, AAI = 0*
010	User data cell, congestion not experienced, AAI = 0*
011	User data cell, congestion not experienced, AAI = 0*
100	Segment VC OAM
101	End-to-End VC OAM

Table 5.2 Defined payload types

* ATM-user-to-ATM-user indication (AAI) used only by
ATM layer users to distinguish between two types of cells.

Thus, the cell sort module identifies the cell as unassigned, signalling, OAM and user cells. An unassigned cell receives no further processing and the cell stored in the shift register buffer is dropped. This is done by input scheduler, since it will not receive the signal valid_cell. User cells and OAM cells require table lookup and hence a signal table_search is sent to the route_table module to load the data from header buffer. Also, if the cell is a user cell, a signal traffic_search is sent to traffic module to load the data from header buffer for UPC processing.

The block diagram of the route_table module is shown in Figure 5.6. The route_table

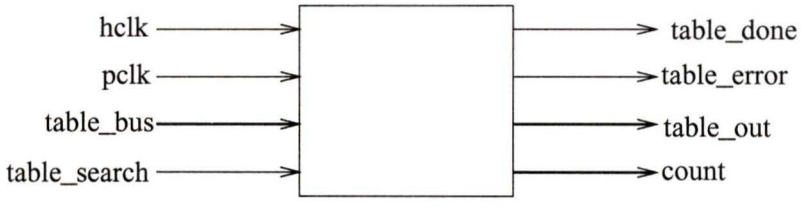


Figure 5.6. *Block diagram of route_table module.*

module after receiving signal table_search from cell sort module reads the first 28 bits

of cell header, which it received from header buffer through table_bus, and searches its CAM to find a match. The form of a table entry is shown in Figure 5.7. Each

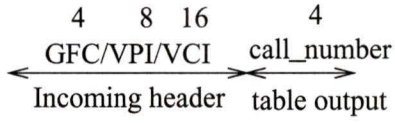


Figure 5.7. *Table entry.*

entry is 32-bits and contains the incoming header and table output. If no match is found then a signal table_error is sent to indicate that the cell will receive no further processing and the cell stored in the shift register buffer is dropped. This is done by input scheduler, since it will not receive the signal valid_cell. One match indicates a unicast cell whereas more than one match indicates a multicast cell. The count of the total number of matches is placed on the CNT_Bus to update the counter of the shift register buffer in which payload of the cell is still arriving. If a match is found then table output is the call number of the cell which is sent to the output module through switching fabric for virtual address routing.

The block diagram of the traffic module is shown in Figure 5.8. The traffic module

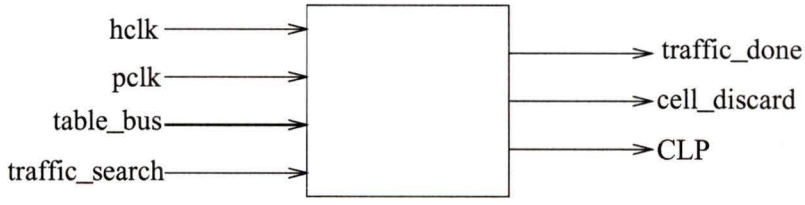


Figure 5.8. *Block diagram of traffic module.*

after receiving signal traffic_search from cell sort module reads the first 28 bits of cell header, which it received from header buffer through table_bus. Within this module lies the software based UPC that enforces the traffic contract. The possible results indicate cell_discard or a two-level cell loss priority (CLP = 0 for high, CLP = 1 for low). These results are sent back to the cell sort module. If cell_discard signal is sent

to cell sort module then it will not send signal valid_cell to input scheduler so that the cell stored in shift register buffer will be dropped.

5.2.3 Input Concentrator

The hardware design of the concentrator is shown in Figure 5.9. As shown in Figure

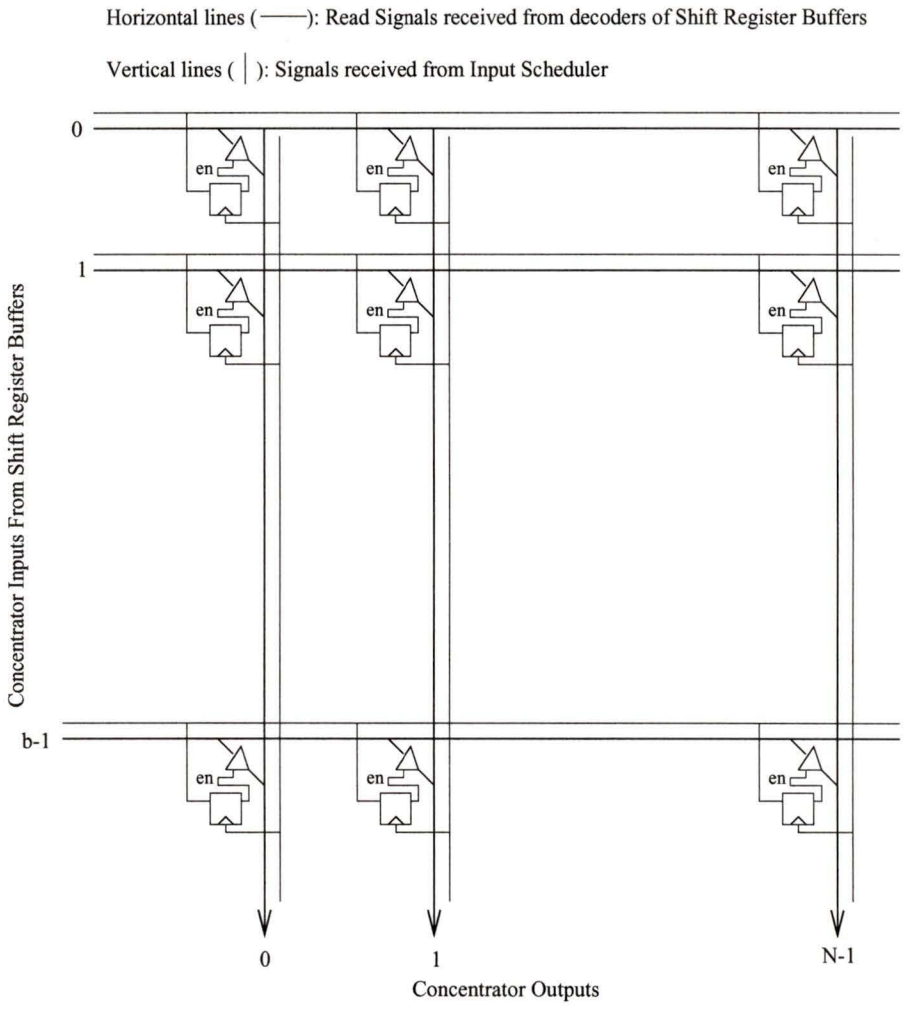


Figure 5.9. *Design of the concentrator.*

5.8. each input port consists of a b to N concentrator, where b is the number of shift register buffers in each input port and N is the switch size. The concentrator directly connects each shift register buffer to the switching module. It is a $(b \times N)$ crossbar

switch. Each crosspoint consists of a flip-flop and a tristate buffer. The flip-flop is enabled by the signals received from the input scheduler. The input to the flip-flop is the read signals received from decoders of shift register buffers and its output enables the tristate buffer thereby establishing connections between concentrator inputs and outputs along columns. It should be noted here that a centralized switching fabric of size $(Nb \times N)$ would be too large, complex and not practical. This is because it can not be implemented on a single chip and if multiple chips are used then they need to communicate between each other. However, partitioning it into N localized concentrators simplifies the design. All the processing to establish the connections between shift register buffers and their destination output ports is done by localized concentrators. The concentrators do not need to communicate between each other. Another big advantage of using localized concentrators is that delay is reduced. Maximum delay is $N + b$ as compared to $Nb + N$ of centralized switching fabric.

5.3 Switching Fabric

The VR switch uses two different switching fabrics.

- ***Switching fabric for virtual address routing***. This switching fabric is used to send the address of the input buffer where the cell is stored, the call number of the cell and 12 bits of the cell header which remains unchanged throughout the connection, to the output port. It is also used to send a cell's address from output port to the input port to establish connections for real cell routing. Thus this switch is input driven as well as output driven but involves small protocol data units. The simplest way is to use three buses, selection bus ($\log_2 N$ bits wide), address bus ($\log_2 N + \log_2 b$ bits wide) and header bus (12 bits wide). Thus, in effect the VR switch has time division virtual routing.
- ***Switching fabric for real cell routing***. This switching fabric is used to route the cell from input port to output port. The use of concentrators at each input port simplified the design of this switching fabric. This is because the switching fabric needs only to connect the concentrator outputs to the output ports as shown in Figure 5.10. Since the switching fabric is a bus, N bits wide, which connects the input concentrators to the outputs, it is possible to drive

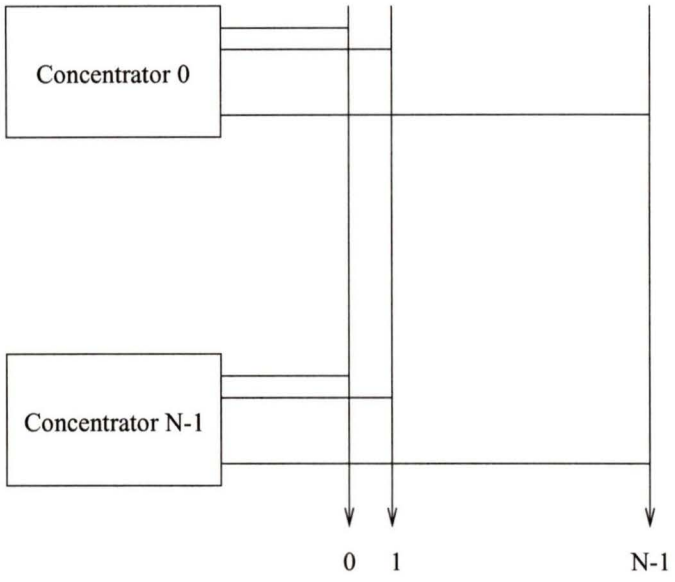


Figure 5.10. *Switching fabric for real cell routing.*

fiber optics directly from the input module.

5.4 Output Module

The output module consists of N output ports for a switch of size N. Each output port consists of output controller, virtual FIFO queues, output buffer and output scheduler. The output controller does the VPI/VCI translation. The block diagram of output controller is shown in Figure 5.11. The input to the output controller, data_in,

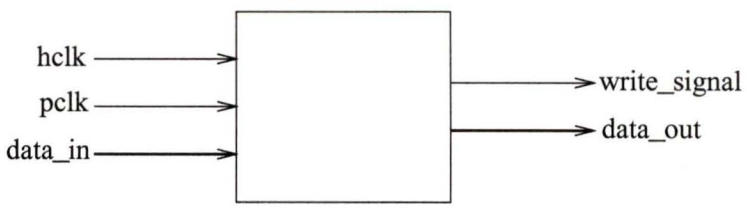


Figure 5.11. *Block diagram of output controller.*

is the call number of the cell and input port address. It is received from selection bus and address bus respectively during the phase of updating virtual FIFO queues.

It is also received from the virtual FIFO queues during the phase of establishing connections for real cell routing. A lookup table in the output controller reads this data and searches its CAM to find a match. If a match is found then the output controller sends a write signal to the virtual FIFO queues. There are three virtual FIFO queues.

- ***Add_FIFO***. It keeps information of the input port address and address of input shift register buffer where the cell is stored received from input module through address bus.
- ***Call_no_FIFO***. It stores the call number of the cell received from input module through selection bus.
- ***Header_FIFO***. It stores the 12 bits of cell header received from input module through header_data_bus.

The block diagram of an FIFO queue is shown in Figure 5.12. The FIFO queue stores

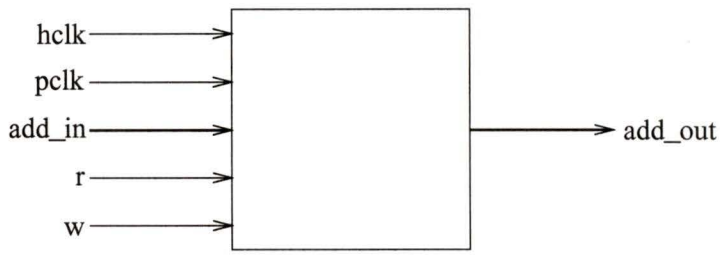


Figure 5.12. *Block diagram of FIFO queue.*

the input data when it is in write mode. This write signal is sent to the virtual FIFO queues by output controller. The data at the head of the FIFO queue is sent out when it is in read mode. This read signal is sent to the virtual FIFO queues by output scheduler. The size of each FIFO queue is Nb where N is the size of the switch and b is the number of shift register buffers in each input port. This is due to the fact that the total number of cells that can be stored in the inputs is Nb. This requirement is not expensive because only the cell's addresses ($\log_2 N + \log_2 b$ bits), call number ($\log_2 N$ bits) and 12 bits of cell header are stored in the respective virtual queues of the output port.

The output controller receives the input port address and call number of the cell from the respective virtual queues during the phase of establishing connections for real cell routing. A lookup table reads this data and searches its CAM and gives 28 bits of new header with translated VPI/VCI values. Thus VPI/VCI translation is done in the output controller. The form of a table entry in lookup table of output controller is shown in Figure 5.13. The 12 bits of cell header received from the header_fifo and

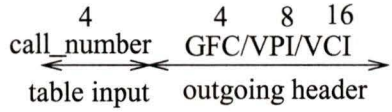


Figure 5.13. *Table entry in lookup table.*

28 bits of cell header received from output controller are stored in output buffer from where they are routed out followed by the payload received from input module. The output scheduler controls the virtual FIFO queues and output buffer by sending them read signals. Furthermore, it drives a priority control scheme to maintain the QoS of different services. The priority scheduler for output queueing strategy is simpler and more efficient than for input queueing or shared memory. The VR switch, though an input buffered switch, uses virtual output queues which makes it easy to support any priority scheduler designed for output buffered switch.

5.5 Plane Function Requirements

The VR switch is able to support three types of information flows: user, control(signalling) and management(OAM). Each flow has different functional requirements within the switch.

5.5.1 User Plane Requirements

The user plane represents the protocols for carrying information between users through the network. Hence the main function of an ATM switch is to transfer the user information contained in the payload of user cells from its input port to its output port.

User cells arriving at the input module are stored in the input shift register buffers. Their headers are processed and the number of destination ports and call number of the cell is determined. For user cells the address of shift register buffer, where cell is stored, the call number of the cell and 12 bits of cell header are written in output virtual FIFO queues. When a user cell is to be sent out, the output module loads the address of input shift register buffer(from which cell is driven out) on the address bus. Required connections are established and the cell is routed from input module to output module.

5.5.2 Control Plane Requirements

The control plane consists of AAL signalling above the ATM layer. Connection request and connection admission are carried by signalling cells which are identified by their VPI/VCI fields. Signalling information must be separated from user cells and processed in the switch.

Signalling cells are identified by special VPI and VCI values. Unlike user cells these cells are not routed directly from input module to output module through cell switch fabric. They are first sent to local cac_Processor in the input controller. The cac_Processor performs connection admission control functions and generate new signalling cells. Now these cells present in cac_Processor are routed to the destined output port. However, it is to be noted that the output port will have a separate high priority queue for signalling cells. When this address is selected for routing the signalling cell is routed to the output module.

5.5.3 Management Plane Requirements

The management plane monitors and controls the ATM network to ensure an efficient operation. The main functions are fault management and performance management. Management cells(OAM cells) are identified by their VCI and PT values. They are separated from user cells and sent to local sm_Processor for management functions.

Like signalling cells, management cells are first routed to local sm_Processor in the

input controller. The sm_Processor performs management functions and generate new management cells which are sent to the destined output port. For this, the address of OAM cells is written in output virtual queue which is a separate high priority queue for OAM cells. The procedure for routing the management cell is the same as that of signalling cells.

5.6 Support for QoS, Multicast and Broadcast Functions

The VR switch easily supports QoS, multicast and broadcast functions without any additional hardware requirements. In this section these issues are discussed in detail. The scalability of the VR switch is also discussed.

5.6.1 Priority Control for Various Service Classes

Four different service classes have been defined by the International Telecommunications Union (ITU). Class A includes time-sensitive, constant-bit-rate, connection-oriented data (CBR video). Class B contains time-sensitive, variable-bit-rate, connection-oriented data (VBR audio and video). Class C consists of connection-oriented, variable-bit-rate data (interactive communication). Class D includes time-insensitive, variable-bit-rate, connectionless data. An important feature of this switch is that it can easily provide support for different service classes thus maintaining the required quality of service (QOS). This is done by adding a virtual FIFO queue for each service class corresponding to each output port.

The priority control for various service classes can be done by employing a priority scheduler. The scheduling algorithm used by the priority scheduler can be weighted round-robin algorithm [41] or window-based algorithm [42]. The VR switch uses window-based algorithm due to the following reason. In weighted round-robin algorithm, if the number of cells available for queue i is less than its allowed value, then the server will move to next priority queue $(i+1)$ and will return to queue i only after serving all other priority queues. This causes partial loss of bandwidth allocated to

queue i and delay for cells from high priority queues. However, for the same case, in window-based algorithm the server will move to the next queue but it will serve only one cell from that queue. It will then return back to queue i. If there is a cell it will be served, otherwise the server will move again to the next queue.

5.6.2 Multicast and Broadcast Functions

One of the basic requirements of an ATM switch is support for multicast and broadcast functions. Multicast operations are very important in supporting point-to-multipoint communications. Many schemes have been developed for multicast communications [3] [43]. However, various ways of supporting multicast operation depends on the queueing strategy involved in the switch.
In an output-buffered ATM switch, multicast cell is copied to all the destination output buffers. This requires additional memory as the multicast cell is copied many times in the copy network. For a shared memory switch, the ways of supporting multicast functions can be categorized under three different classes [44]:

- Multiple write, multiple read (MWMR) of a multicast cell: The number of replicated cells input to the switch can be $O(N^2)$, for a finite memory space it would result in considerable cell loss. Also, during its output, multiple-read operation is limited by the memory speed.
- Single write, multiple read (SWMR) of a multicast cell: Multicast cell is not replicated and stored directly in shared memory space (single write) in a separate logical queue. However, multiple-read operation is limited by the memory speed.
- Single write, single read (SWSR) of a multicast cell: Multicast cells form a separate logical queue within the shared memory space. During its output, it is routed to high-speed bus for its replication and transmission to its destinations. Use of additional hardware is the disadvantage in this scheme.

In the proposed ATM switch multicast and broadcast functions can be easily supported without the use of any additional hardware. No movement or copying of the cell is required. Also, it is not required to increase the size or speed of the memory. A multicast cell is received and its payload is stored in the input shift register buffer

in the usual way. The VPI/VCI values of the cell header are translated at the output ports in the usual way. Thus all the destination output ports will have the new header of the multicast cell which can be transmitted out from the switch. To route the common payload of the cell simultaneously, to more than one output, connections are established between the input shift register buffer (in which the cell is stored) and all the destination ports. Also, the shift register buffer is used as a ring shift register. Each time payload of the cell is shifted out, a copy of the cell is shifted in. At the same time the counter of that shift register decrements by one. When the counter reads zero, the shift register is made free to store another incoming cell.

5.6.3 Scalability

One of the basic requirements for an ATM switch is scalability. The switch architecture must permit size growth from a small number of ports to large numbers. The modular design of the proposed switch makes it easy to support a flexible number of ports. However, in our design we have time-division multiplexing for virtual address routing to update the output virtual queues and to establish the connections. Thus, if T_1 is the total time available to update the output virtual queues and t_1 is the time taken to update virtual queue for one port then $Nt_1 \leq T_1$, where N is the size of the switch. Similarly, if T_2 is the total time available to establish the connections and t_2 is the time taken to establish connection for one port then $Nt_2 \leq T_2$. This puts an upper limit on the number of ports of the switch given by

$$N \leq min\left\{\frac{T_1}{t_1}, \frac{T_2}{t_2}\right\}$$

This upper limit on the number of ports of the switch can be removed by using space-division multiplexing instead of time-division multiplexing for updating the output virtual queues and establishing the connections. Instead of using bus we can use multistage interconnection networks (MINs) such as buffered banyan network so that virtual address routing can be done simultaneously for each port.

5.7 Implementation

The VR switch is implemented using high-level VHDL modeling. This implementation provides many of the ATM switch functions but supports only the user plane requirements. The logic of this VHDL model is verified through simulation. The behavioral/structural code in VHDL for the VR switch can be found in Appendix A.

The VHDL model of the VR switch is of size $N = 4$ with $b = 7$ buffers in each input port. Figure 5.14 shows the simulation results to verify the logic of VHDL model of the VR switch. As shown in Figure 5.14, cells are arriving at all the four input ports at the rising edge of hclk. As soon as cell header is received, it is processed and the call number of the cell, the address of input port and shift register buffer where cell is stored and 12 bits of cell header are sent to the output module one by one, in succession, through s0, s2 and s3 respectively.

At the falling edge of hclk connections are established to route the cell. Each output port sends the address of the cell, which is to be routed, to the input module one by one, in succession, through s2.

Cells are routed from the shift register buffers to the destined output ports and transmitted out from the switch at the next rising edge of hclk (as shown in figure).

The VHDL code of the VR switch is imported in SYNOPSYS to get the gate-level implementation. Figure 5.15 show the block diagram of the VR switch obtained from SYNOPSYS. However this implementation supports only user plane requirements i.e. it can route only user cells from input ports to destined output ports.

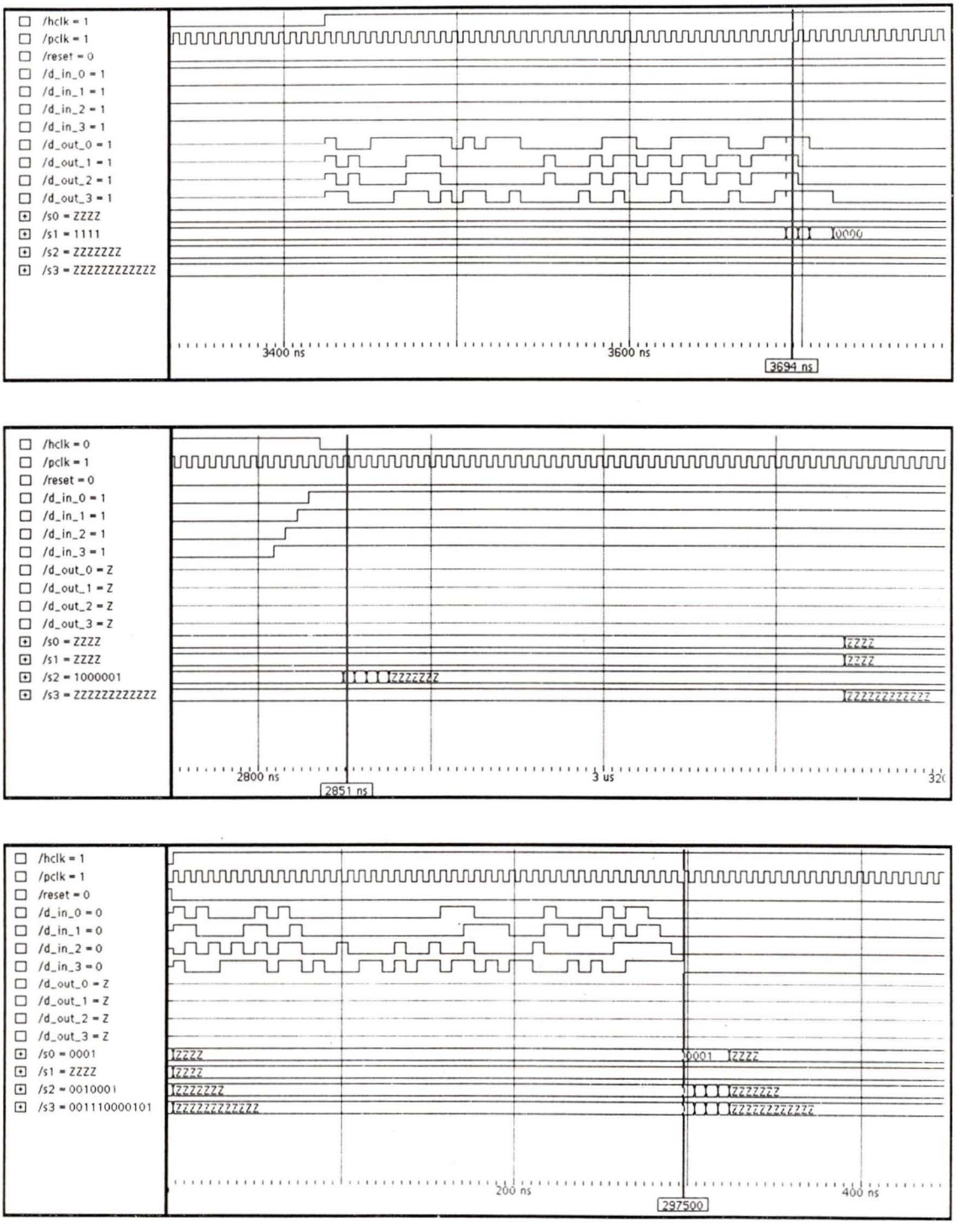
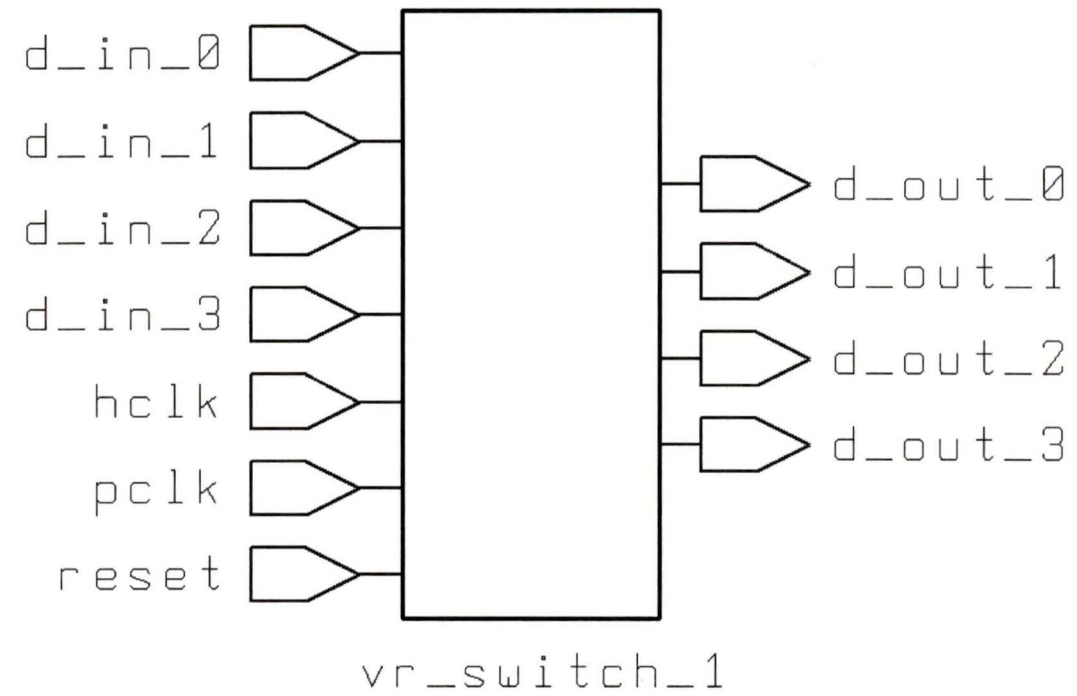


Figure 5.14. *Simulation results for the VHDL model of VR switch.*



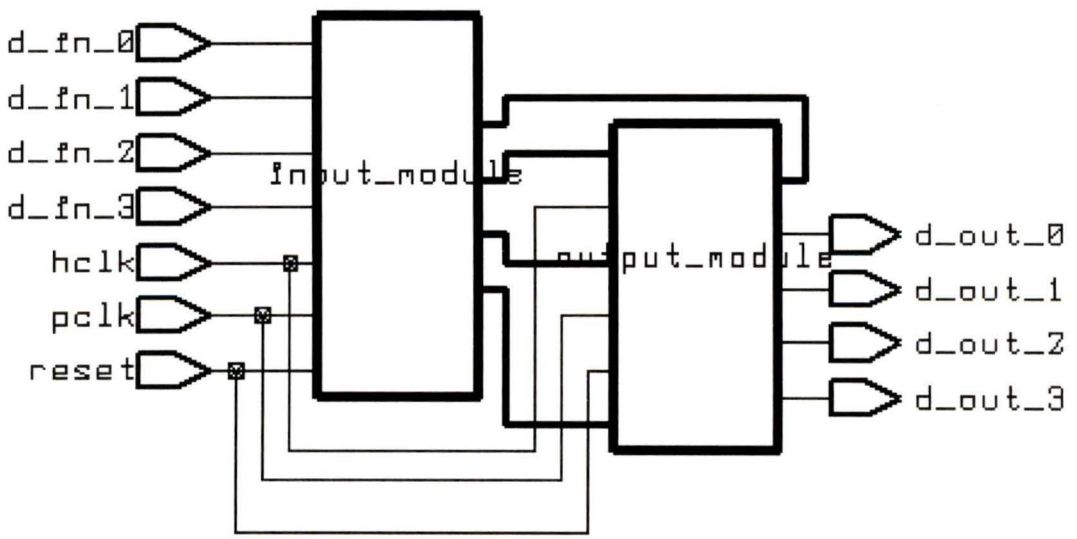


Figure 5.15. *Block diagram of the VR switch in decreasing order of hierarchy.*

5.8 Summary

The hardware deatails of the VR switch are discussed. It is seen that the design of the VR switch is simple and modular and hence easy for implementation. Quality of Service (QoS), multicast and broadcast functions are easily supported in this switch. The use of localized CAC and OAM makes the switch more scalable to large sizes. However, the use of selection bus, address bus and header bus in the switching fabric for vitrual address routing puts a limit on the size of the switch because in effect, there is time division virtual routing. The use of localized concentrators simplifies the design of switching fabric for real cell routing and makes it possible to drive fiber cable directly from the input port.

The VR switch is implemented using high-level VHDL modeling. The logic of this VHDL model is verified through simulation. The VHDL code is imported in SYNOPSYS to get the gate-level implementation of the VR switch.

Chapter 6

Conclusions and Future Work

6.1 Thesis Contributions

- Designed a new ATM switch which overcomes the drawbacks of earlier ATM switches, satisfies the general requirements of an ATM switch, supports Quality of Service (QoS), multicast and broadcast functions. It is scalable in terms of size and speed and has the potential to be included in very high data rate networks.
- Simplified the design of the switching fabric by using localized concentrators, which enabled the use of photonic switching technology in the new ATM switch.
- Evaluated the performance of the new ATM switch by computer simulation and modeling to show that it has lower cell loss probability and optimal throughput/delay performance.
- Implemented the VR switch using high-level VHDL modeling. The logic of this VHDL model is verified by simulation.
- Implemented the VR switch at the gate level using SYNOPSYS.

6.2 Suggested Future Work

- Investigate the possibility of the use of multistage interconnection networks for virtual address routing and localized concentrators.
- Investigate the possibility of eliminating localized concentrators by making use of several wavelengths of an optical fiber for routing.

- Modify the VHDL code to support control plane and management plane functions.
- Get the VHDL code converted to verilog in SYNOPSYS and import it in CADENCE to do VLSI implementation and get the ATM chipset.

Bibliography

[1] F. Tobagi, "Fast packet switch architectures for broadband integrated services digital networks", *Proceedings of the IEEE*, Vol. 38, January 1990, pp. 133-167.

[2] E. Zegura, "Architecture for ATM switching systems", *IEEE Communication Magazine*, Vol. 31, No. 2, 1993, pp. 28-37.

[3] A. Pattavina, "Non-blocking architectures for ATM switching", *IEEE Communication Magazine*, Vol. 31, No. 2, 1993, pp. 38-48.

[4] R. Awdeh, and H. Mouftah, "Survey of ATM switch architectures", *Computer Networks and ISDN systems*, Vol. 27, 1995, pp. 1567-1613.

[5] J. Turner, "Design of an integrated services packet network", *IEEE J. Selected Areas Commun.*, Vol. 4, N0. 8, November 1986, pp. 1373-1380.

[6] J. Berthold, "High speed integrated electronics for communication systems", *Proceedings of the IEEE*, Vol. 78, No. 3, March 1990, pp. 486-511.

[7] P. Baran, "On distributed communication networks", *IEEE Trans. Commun*, Vol. 12, March 1964, pp. 1-9.

[8] *ATM User-Network Specification*, Version 3.1, The ATM Forum, 1994.

[9] P. Newman, "ATM technology for corporate networks", *IEEE Communication Magazine*, Vol. 30, April 1992, pp. 90-101.

[10] M. Devault, J. Cochennec, and M. Servel, "The "prelude" ATD experiment: Assessments and future prospects", *IEEE J. Selected Areas Commun.*, Vol. 6, No. 9, December 1988, pp. 1528-1537.

[11] N. Endo, T. Kozaki, T. Ohuchi, H. Kuwahara, and S. Gohara, "Shared buffer memory switch for an ATM exchange", *IEEE Trans. Commun*, Vol. 41, N0. 1, January 1993, pp. 237-245.

[12] H. Lee, et al., "A shared output buffer switch for ATM", *Fourth International Conf. on Data Communication Systems and their Performance*, Barcelona, June 1990, pp. 163-179.

[13] J. Causey, and H. Kim, "Comparision on buffer allocation schemes in ATM switches: Complete sharing, Partial sharing and Dedicated allocation", *ICC '94 Conf. Rec.*, New Orleans, LA, May 1994, pp. 1164-1168.

[14] K. Oshima, et al., "A new ATM switch architecture based on STS-type shared

buffering and its LSI implementation", *Proc. XIX ISS '92*, Yokohama, Japan, October 1992, pp. 359-363.

[15] H. Yamanaka, et al., "622 Mb/s 8 × 8 shared multibuffer ATM switch with hierarchical queueing and multicast functions", *GLOBECOM '93 Conf. Rec.*, Houston, TX, November/December 1993, pp. 1488-1495.

[16] S. Kumar, and D. Agrawal, "A shared buffer direct access (SBDA) switch architecture for ATM-based networks", *ICC '94 Conf. Rec.*, New Orleans, LA, May 1994, pp. 101-105.

[17] H. Suzuki, et al., "Output-buffer switch architecture for asynchronous transfer mode", *Proc. Int. Conf. on Communications*, Boston, MA, June 1989, pp. 4.1.1-4.1.5.

[18] I. Cidon, et al., "Real-time packet switching: a performance analysis", *IEEE J. Selected Areas in Communications*, Vol. 6, No. 9, December 1988, pp.1576-1586.

[19] C. Fayet, A. Jacques, and G. Pujolle, "High speed switching for ATM: the BSS", *Computer Networks and ISDN Systems*, Vol. 26, May 1994, pp. 1225-1233.

[20] S. Nojima, E. Tsutsui, H. Fukuda, and M. Hashimoto, "Integrated services packet network using bus-matrix switch", *IEEE J. Selected Areas in Communications*, Vol. 5, No. 10, October 1987, pp. 1284-1292.

[21] E. Rathgeb, T. Theimer, and M. Huber, "Buffering concepts for ATM switching networks", *Proc. IEEE GLOBECOM'88*, Hollywood, FL, November 1988, pp.1277-1281.

[22] A. Gupta, L. Barbosa, and N. Georganas, "A 16 × 16 limited intermediate buffer switch module for ATM networks", *Proc. IEEE GLOBECOM'91*, Phoenix, AZ, December 1991, pp. 939-943.

[23] Y. Yeh, M. Hluchyj, and A. Acampora, "The Knockout switch: a simple, modular architecture for high-performance packet switching", *IEEE J. Selected Areas in Communications*, Vol. 5, No. 8, October 1987, pp. 1274-1283.

[24] Y. Jenq, "Performance analysis of a packet switch based on single-buffered banyan networks", *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1, December 1983, pp. 1014-1021.

[25] H. Kim, and A. Leon-Garcia, "Performance of buffered banyan networks under nonuniform traffic patterns", *IEEE Trans. Commun.*, Vol. 38, May 1990, pp. 648-658.

[26] Y. Mun, and H. Youn, "Performance analysis of finite buffered multistage interconnection networks", *IEEE Trans. Computers*, Vol. 43, February 1994, pp. 153-162.

[27] A. Huang, and S. Knauer, "Starlite: a wideband digital switch", *Proc. IEEE GLOBECOM'84*, Atlanta, GA, November 1984, pp. 121-125.

[28] J. Giacopelli, J. Hickey, W. Marcus, W. Sincoskie, and M. Littlewood, "Sunshine: a high-performance self-routing broadband packet switch architecture", *IEEE J. Selected Areas Commun.*, Vol. 9, No. 8, October 1991, pp. 1289-1298.

[29] E. Bushnell, and J. Meditch, "Dilated multistage interconnection networks for fast packet switching", *IEEE INFOCOM*, 1991, pp. 1264-1273.

[30] T. Lee, and S. Liew, "Broadband packet switches based on dilated interconnection networks", *IEEE Trans. Commun.*, Vol. 42, February 1994, pp. 732-744.

[31] F. Tobagi, T. Kwok, and F. Chiussi, "Architecture, performance and implementation of the tandem banyan fast packet switch", *IEEE J. Selected Areas Commun.*, Vol. 9, October 1991, pp. 1173-1193.

[32] M. Karol, K. Eng, H. Obara, "Improving the performance of input-queued ATM packet switches," *INFOCOM '92,* pp. 110-115.

[33] H. Obara, Y. Hamazumi, "Parallel contention resolution control for input queueing ATM switches," *IEE Electronics Letters,* Vol. 28, No. 9, April 1992, pp. 838-839.

[34] M.G. Hluchyj and M.J. Karol, "Queueing in high performance packet switching", *IEEE J. Select. Areas Commun.,* vol. 6, No. 9, December 1988, pp. 1587-1597.

[35] H. ElGebaly, J. Muzio and F. ElGuibaly, "Input smoothing with buffering: A new technique for queueing in fast packet switching," *IEEE*, 1995.

[36] F. El-Guibaly and S. Agarwal, "Design and performance analysis of shift register-based ATM switch," *IEEE Proc. Pacific Rim Conference on Communications, Computers and Signal Processing,* Victoria, August 1997, pp. 70-73.

[37] T.E. Eliazov, V. Ramaswamy, W. Willinger, and G. Latouche, "Performance of an ATM switch: Simulation study," *Proc. INFOCOM*, June 1990, pp. 644-659.

[38] S. Agarwal and F. El-Guibaly, "Modelling of shift register-based ATM switch," *IEEE Proc. Eighth Great Lakes Symposium on VLSI,* Lafayette, Louisiana, February 19-21, 1998, pp. 146-151.

[39] M.J. Karol, M.G. Hluchyj, and S.P. Morgan, "Input versus output queueing on a space-division packet switch", *IEEE Trans. Commun.* vol. COM-35, No. 12, December 1987, pp. 1347-1356.

[40] F. El-Guibaly, "Design and analysis of arbitration protocols," *IEEE Trans. Commun.*, Vol. 38, No. 2, February 1989, pp. 161-171.

[41] M. Katevenis, S. Sidiropoulos and C. Courcoubetis, "Weighted round -robin cell multiplexing in a general-purpose ATM switch chip," *IEEE J. Selected Areas Commun.,* Vol. 9, October 1991, pp. 1265-1279.

[42] A. Sabaa, F. El-Guibaly, J. Muzio and D. Shpak, "Implementation of a window-based scheduler in an ATM switch," *Canadian Conference on Electrical and Computer Engineering*, September 1995, pp. 32-35.

[43] H. Chao and B. Choe, "Design and analysis of a large-scale multicast output buffered ATM switch," *IEEE/ACM Trans. Networking*, Vol. 3, no. 2, April 1995, pp. 126-138.

[44] S. Kumar and D.P. Agrawal, "On multicast support for shared-memory based ATM switch architecture," *IEEE Network*, January/February 1996, pp. 34-39.

Appendix A

VHDL Code For The VR Switch

The VR switch is implemented using high-level VHDL modeling. In this Appendix, the VHDL code for each module of the VR switch is provided.

A.1 Input Module

A.1.1 atm_pkg.vhd

```
--  File atm_pkg.vhd
--
-- This package contains the constants and types used in the ATM switch, as
-- well as the initial function that performs header discrimination in the
-- cell_sort module. This package must be compiled before any core ATM
-- components are compiled, but may be compiled after the library components.

-- Programmed by Sandeep Agarwal

library ieee, std;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-- pragma synthesis_off
use std.textio.all;
-- pragma synthesis_on


PACKAGE atm_pkg IS


function conv1_StdlogicVector (arg1 : integer; size : natural) RETURN std_logic_vector;
--pragma synthesis_off
procedure read(L: inout line; Value: out std_logic_vector);
type std_logic_chars is array(character) of std_logic;
constant to_stdlogic: std_logic_chars :=
       ('U' => 'U', 'X' => 'X', '0' => '0', '1' => '1', 'Z' => 'Z',
 'W' => 'W', 'L' => 'L', 'H' => 'H', '-' => '-', others => 'X');
--pragma synthesis_on
END atm_pkg;

PACKAGE BODY atm_pkg IS
```

```
      FUNCTION conv1_StdlogicVector (arg1 : integer; size : natural)
               RETURN std_logic_vector IS
-- pragma synthesis_off
         VARIABLE vector : std_logic_vector(0 TO size-1);
         VARIABLE tmp_int : integer := arg1;
         VARIABLE carry   : std_logic := '1';   -- setup to add 1 if needed
         VARIABLE carry2  : std_logic;
         -- Integer to signed vector conversion
         -- ATTRIBUTE synthesis_return OF vector:VARIABLE IS "FEED_THROUGH" ;
      BEGIN
         FOR i IN size-1 DOWNTO 0 LOOP
             IF tmp_int MOD 2 = 1 THEN
                vector(i) := '1';
             ELSE
                vector(i) := '0';
             END IF;
             tmp_int := tmp_int / 2;
         END LOOP;

         IF arg1 < 0 THEN
            FOR i IN size-1 DOWNTO 0 LOOP
                carry2    := (NOT vector(i)) AND carry;
                vector(i) := (NOT vector(i)) XOR carry;
                carry     := carry2;
           END LOOP;
         END IF;
         RETURN vector;
-- pragma synthesis_on
      END conv1_StdlogicVector;

-- pragma synthesis_off
procedure read(L: inout line; Value: out std_logic_vector) is
 variable temp: string(value'range);
 variable good_string: boolean;
begin
  read(L, temp, good_string);
  if good_string = true then
   for i in temp'range loop
      value(i) := to_Stdlogic(temp(i));
   end loop;
  end if;
end read;
-- pragma synthesis_on

END atm_pkg;
```

A.1.2 p_buffer.vhd

```
-- This is the code for the payload buffer which stores 384 bits of ATM cell
-- payload. At the rising edge of hclk, at each falling edge of pclk one bit
-- data is stored and the others are shifted to the right, if it is in write
-- mode. If the payload buffer is in read mode then At the rising edge of hclk,
-- at each rising edge of pclk one bit of data is sent out as well as
-- re-circulated. Thus the payload shift register buffer acts as a ring
-- register.

-- Programmed by Sandeep Agarwal
```

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

entity p_buffer is
          generic (p_size : integer := 384);
          port(hclk,pclk,R,W,d_in,rs1 : in std_logic;
            d_out : out std_logic);
end p_buffer;

architecture behav of p_buffer is
signal pb : std_logic_vector(p_size-1 downto 0);
begin

P_write:
 process(pclk)
 begin
if(hclk='1') then
             if(pclk'event and pclk='0') then
                 if(W='1') then
                   for i in 0 to p_size-2 loop
                       pb(i) <= pb(i+1);
                   end loop;
                   pb(p_size-1) <= d_in;
   d_out <= 'Z';
                elsif(R='1' and rs1='1') then
                   for i in 0 to p_size-2 loop
                     pb(i) <= pb(i+1);
                   end loop;
                     pb(p_size-1) <= pb(0);
                     d_out <= pb(0);
 else
     d_out <= 'Z';
                end if;
             end if;
        end if;
 end process;

end behav;
```

A.1.3 Add_Reg.vhd

```
-- This module stores the address of the payload buffer which is the address of
-- the input port and the address of the buffer. In this code the address is of
-- 7 bits, 3 for the input port address and 4 for the buffer address. This
-- module sends this address to output module for updating their virtual queues
-- through add_bus if it is in write mode and a valid_cell has arrived. It also
-- sends the address to input scheduler through cnt_bus if it receives signals
-- cnt_sig and R_d.

-- Programmed by Sandeep Agarwal

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

```
entity add_reg_ip is
generic (add_int : integer:=17);
port(cnt_sig,pclk,reset : in std_logic;
     valid_cell,w,r1 : in std_logic;
     cnt_bus,add_bus : out std_logic_vector(6 downto 0));
--constant address : std_logic_vector(6 downto 0) := "0010001";
end add_reg_ip;

architecture behav of add_reg_ip is
signal R_d : std_logic;
signal address : std_logic_vector(6 downto 0) ;
begin


 P_add_bus: process(valid_cell)
   begin
   -- pragma synthesis_off
address<=conv1_StdlogicVector(add_int,7);
   -- pragma synthesis_on
if(valid_cell='0') then
    add_bus<="ZZZZZZZ";
elsif(w='1') then
    add_bus<=address;
else
    add_bus<="ZZZZZZZ";
end if;
   end process;

 P_cnt_bus : process(pclk)
     begin
if(pclk'event and pclk='1') then
   if(R_d='1' and cnt_sig='1') then
cnt_bus<=address;
   else
cnt_bus<="ZZZZZZZ";
   end if;
end if;
     end process;

 P_R_d : process(pclk,reset)
begin
if(reset/='1') then
  if(pclk'event and pclk='0') then
R_d<=r1;
  end if;
else
R_d<='0';
end if;
  end process;
end behav;
```

A.1.4 counter_ip.vhd

```
-- This module keeps the count of the number of destination ports of the cell
-- which is stored in its payload buffer. Each time the cell is routed out
-- it decrements its reading by one and when the reading becomes zero it sends a
-- signal cnt_sig to the add_reg module to indicate that the buffer is now
-- free to store a new cell.
```

```
-- When it is in write mode it receives the count from cnt_bus and stores it.
-- Each time it receives signal r1 it decrements the count by one. When the
-- count becomes zero it sends signal cnt_sig.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.all;

entity counter_ip is
port(pclk,HP,W,r1,reset : in std_logic;
     cnt_sig : out std_logic;
     cnt_bus : in std_logic_vector(6 downto 0));
end counter_ip;

architecture struc of counter_ip is
component updn_cntr
generic(width : POSITIVE);
  port(data     : in STD_LOGIC_VECTOR(width-1 downto 0);
       up_dn, load, cen, clk, reset : in STD_LOGIC;
       count    : out STD_LOGIC_VECTOR(width-1 downto 0);
       tercnt   : out STD_LOGIC);
end component;
signal data,count : std_logic_vector(2 downto 0);
signal up_dn,cen,clk,tercnt,reset1,load : std_logic;
begin
c0: updn_cntr
   generic map(3)
   port map(data,up_dn,load,cen,clk,reset1,count,tercnt);

data<=cnt_bus(2 downto 0);
up_dn <= '0';

load <= not(HP and W);
clk <= not pclk;
cen<=(W and HP) or r1;
cnt_sig<=tercnt;
reset1<= not reset;
end struc;
```

A.1.5 ip_buffer_single.vhd

```
-- This module combines the payload buffer, add_reg and counter. It also
-- adds two decoders to decode the addresses on write_bus and read_bus
-- and send write and read signals respectively to the payload buffer.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;

entity ip_buffer_single is
generic(p_size : integer :=384;
        p_address : integer := 17);
```

```
port(din,hclk,pclk,reset: in std_logic;
     c_bus:in std_logic_vector(4 downto 0);
     r_bus,w_bus : in std_logic_vector(6 downto 0);
     dout,r2:out std_logic;
     cnt_bus : inout std_logic_vector(6 downto 0);
     add_bus : out std_logic_vector(6 downto 0));
end ip_buffer_single;

architecture struc of ip_buffer_single is
component  p_buffer
           generic (p_size : integer := 384);
           port(hclk,pclk,R,W,d_in,rs1 : in std_logic;
             d_out : out std_logic);
end component;

component counter_ip
        port(pclk,HP,W,r1,reset: in std_logic;
             cnt_sig : out std_logic;
             cnt_bus : in std_logic_vector(6 downto 0));
end component;

component add_reg_ip
        generic (add_int : integer := 17);
        port(cnt_sig,pclk,reset : in std_logic;
             valid_cell,w,r1 : in std_logic;
             cnt_bus,add_bus : out std_logic_vector(6 downto 0));
        --constant address : std_logic_vector(6 downto 0) := "0010001";
end component;

signal r1,r,w,cnt_sig,valid_cell,hp,rst,rs,rs1 : std_logic;
signal port_address : std_logic_vector(6 downto 0);

begin

-- pragma synthesis_off
P_p_address:
process
begin
port_address<=conv1_StdlogicVector(p_address,7);
wait ;
end process;
-- pragma synthesis_on

P_decoder :
process(r_bus,w_bus)
begin
if(r_bus=port_address) then
r1<='1';
else
r1<='0';
end if;
if(w_bus=port_address) then
w<='1';
else
w<='0';
end if;
end process;

P_R:
process(pclk,rst,r1,reset)
```

```
begin
     if(rst/='1' and reset/='1') then
if(pclk'event and pclk='0') then
    if(r1='1') then
r<=r1;
    end if;
end if;
     else
r<='0';
     end if;
end process;

u0: p_buffer
generic map(p_size)
port map(hclk,pclk,r,w,din,rs1,dout);

u1: counter_ip
port map(pclk,hp,w,r1,reset,cnt_sig,cnt_bus);

u2: add_reg_ip
generic map(p_address)
port map(cnt_sig,pclk,reset,valid_cell,w,r1,cnt_bus,add_bus);

valid_cell<=c_bus(1);
hp<=c_bus(0);
rst<=c_bus(2);
rs<=c_bus(3);
rs1<=c_bus(4);
     r2<=r1;

end struc;
```

A.1.6 ip_buffers.vhd

```
-- This is the code for input buffers for one port. Each port consists of
-- seven payload buffers to store the payload of the incoming cell.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ip_buffers is
generic(port_number : integer := 16);
port(din,hclk,pclk,reset: in std_logic;
             c_bus:in std_logic_vector(4 downto 0);
             r_bus,w_bus : in std_logic_vector(6 downto 0);
     r2 : out std_logic_vector(6 downto 0);
             dout:out std_logic_vector(6 downto 0);
             cnt_bus : inout std_logic_vector(6 downto 0);
             add_bus : out std_logic_vector(6 downto 0));
end ip_buffers;

architecture struc of ip_buffers is
component ip_buffer_single
        generic(p_size : integer :=384;
                p_address : integer := 17);
```

```
        port(din,hclk,pclk,reset: in std_logic;
             c_bus:in std_logic_vector(4 downto 0);
             r_bus,w_bus : in std_logic_vector(6 downto 0);
             dout,r2:out std_logic;
             cnt_bus : inout std_logic_vector(6 downto 0);
             add_bus : out std_logic_vector(6 downto 0));
end component;
begin

b0: ip_buffer_single
generic map(384,port_number+1)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(0),r2(0),cnt_bus,add_bus);

b1: ip_buffer_single
generic map(384,port_number+2)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(1),r2(1),cnt_bus,add_bus);

b2: ip_buffer_single
generic map(384,port_number+3)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(2),r2(2),cnt_bus,add_bus);

b3: ip_buffer_single
generic map(384,port_number+4)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(3),r2(3),cnt_bus,add_bus);

b4: ip_buffer_single
generic map(384,port_number+5)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(4),r2(4),cnt_bus,add_bus);

b5: ip_buffer_single
generic map(384,port_number+6)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(5),r2(5),cnt_bus,add_bus);

b6: ip_buffer_single
generic map(384,port_number+7)
port map(din,hclk,pclk,reset,c_bus,r_bus,w_bus,dout(6),r2(6),cnt_bus,add_bus);

end struc;
```

A.1.7 ip_scheduler.vhd

```
-- This is the code for input scheduler which keeps updated information
-- of empty shift register buffers. It receives the idle buffer address
-- from cnt_bus and thus keeps information of all idle buffers. At the
-- falling edge of hclk it sends an idle address to input buffers
-- through write_bus. This address is decoded in input buffers and thus
-- write signal is sent to one payload buffer to store the incoming cell.
-- The input scheduler also sends signals rst, rs, rs1, new_clk and c_sig
-- to different modules of the input port.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;

entity ip_scheduler is
```

```
generic(p_size : integer := 40;
h_size : integer := 40;
port_id : integer := 1);
port(hclk,pclk,reset,v_cell:in std_logic;
     cnt_bus : in std_logic_vector(6 downto 0);
     write_bus : out std_logic_vector(6 downto 0);
     rst,rs,rs1,new_clk : out std_logic;
     c_sig : out std_logic_vector(3 downto 0));
end ip_scheduler;

architecture behav of ip_scheduler is
signal hclk_d,clk_new : std_logic;
signal pg_in,pg_out : std_logic_vector(7 downto 0);
signal max_count : integer:= 5;          -- Total number of ports(N=4)+1
signal count1: integer :=0;
signal port_address : std_logic_vector(2 downto 0);
signal reset1 : std_logic;
signal c_sig1, c_sig2 : std_logic_vector(3 downto 0);
begin

-- pragma synthesis_off
P_p_address:
        process
        begin
                port_address<=conv1_StdlogicVector(port_id,3);
                wait ;
        end process;
-- pragma synthesis_on

P_pg_in :
process(pclk,hclk,reset)
begin
    if(reset='1') then
pg_in<="11111111";
    else
    if(pclk'event and pclk='0') then
   if(hclk='1') then
      if(v_cell='1') then
pg_in<=pg_in and (not pg_out);
      end if;
   elsif(hclk='0') then
      if(cnt_bus(6 downto 4)=port_address) then
 if(cnt_bus(3 downto 0)="0001") then
pg_in(1)<='1';
 elsif(cnt_bus(3 downto 0)="0010") then
pg_in(2)<='1';
 elsif(cnt_bus(3 downto 0)="0011") then
pg_in(3)<='1';
 elsif(cnt_bus(3 downto 0)="0100") then
pg_in(4)<='1';
 elsif(cnt_bus(3 downto 0)="0101") then
pg_in(5)<='1';
 elsif(cnt_bus(3 downto 0)="0110") then
pg_in(6)<='1';
 elsif(cnt_bus(3 downto 0)="0111") then
pg_in(7)<='1';
 end if;
      end if;
   end if;
end if;
```

```
   end if;
end process;

P_pg_out:
process(hclk,reset)
begin
   if(reset='1') then
pg_out<="10000000";
   else
if(hclk'event and hclk='0') then
if(pg_in(7)='1') then
pg_out<="10000000";
elsif(pg_in(6)='1') then
pg_out<="01000000";
elsif(pg_in(5)='1') then
pg_out<="00100000";
elsif(pg_in(4)='1') then
pg_out<="00010000";
elsif(pg_in(3)='1') then
pg_out<="00001000";
elsif(pg_in(2)='1') then
pg_out<="00000100";
elsif(pg_in(1)='1') then
pg_out<="00000010";
elsif(pg_in(0)='1') then
pg_out<="00000001";
end if;
end if;
    end if;
end process;

P_encoder:
process(reset,pg_out)
begin
 if(reset='1') then
   write_bus<=port_address & "0001";
 else
             if(pg_out="10000000") then
                   write_bus<=port_address & "0001";
             elsif(pg_out="01000000") then
                   write_bus<=port_address & "0010";
             elsif(pg_out="00100000") then
                   write_bus<=port_address & "0011";
             elsif(pg_out="00010000") then
                   write_bus<=port_address & "0100";
             elsif(pg_out="00001000") then
                   write_bus<=port_address & "0101";
             elsif(pg_out="00000100") then
                   write_bus<=port_address & "0110";
             elsif(pg_out="00000010") then
                   write_bus<=port_address & "0111";
             elsif(pg_out="00000001") then
                   write_bus<=port_address & "0000";
             end if;
          end if;
       end process;


P_rst:
process(hclk_d,hclk,reset)
```

```
begin
    if(reset='1') then
rst<='1';
    else
if(hclk_d='0') then
    rst<='0';
elsif(hclk='0') then
    rst<='1';
end if;
    end if;
end process;

P_hclk_d:
process(pclk)
begin
if(pclk'event and pclk='0') then
    hclk_d<=hclk;
end if;
end process;

P_clk_new:
process(pclk)
begin
if(hclk='1') then
clk_new<=pclk;
else
clk_new<='0';
end if;
end process;

P_c_sig1:
        process(pclk,hclk)
 variable count: integer :=0;
         begin
           if(pclk'event and pclk='0') then
             -- if(hclk='1') then
              --   reset:='0';
              if(hclk='0' and reset1='0') then
                 count := count+1;
                 count1<=count;
              end if;
              --if(count=max_count) then
      if(reset1='1') then
              --   reset:='1';
                 count:=0;
 count1<=count;
              end if;
           end if;
         end process;

        process(pclk)
         begin
           if(pclk'event and pclk='1') then
     if(hclk='1') then
reset1<='0';
     elsif(count1=max_count) then
reset1<='1';
     end if;
              if(count1=1) then
                c_sig1 <= "1000";
```

```
              elsif(count1=2) then
c_sig1 <= "0100";
              elsif(count1=3) then
c_sig1 <= "0010";
              elsif(count1=4) then
c_sig1 <= "0001";
              else
                c_sig1 <= "0000";
              end if;
            end if;
          end process;


P_rs:
process(clk_new,hclk,reset)
variable count : integer;
begin
    if(reset='1') then
count:=0;
rs <='0';
    else
if(hclk='1') then
    if(clk_new'event and clk_new='1') then
count:=count+1;
                    if(count=h_size) then
       rs1 <= '1';
    elsif(count=h_size+1) then
rs<='1';
    end if;
    end if;
else
    count:=0;
    rs1<='0';
    rs<='0';
end if;
   end if;
end process;
new_clk <= clk_new;
process(pclk)
 begin
  if(pclk'event and pclk='0') then
   c_sig2 <= c_sig1;
  end if;
end process;
process(pclk)
 begin
  if(pclk'event and pclk='1') then
   c_sig <= c_sig2;
  end if;
end process;

end behav;
```

A.1.8 head_buffer.vhd

```
-- This is the code for header buffer. It stores the header of the cell and
-- sends it to cell sort module, route table module and traffic module.
```

```
-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity head_buffer is
generic (h_size : integer := 40);
port(hclk,pclk,d_in,rs : in std_logic;
     h_bus : out std_logic_vector(h_size-1 downto 0));
end head_buffer;

architecture behav of head_buffer is
signal hb : std_logic_vector(h_size-1 downto 0);
begin

process(pclk)
begin
if(hclk='1') then
 if(rs='0') then
    if(pclk'event and pclk='0') then
       for i in 0 to h_size-2 loop
         hb(i)<=hb(i+1);
               end loop;
         hb(h_size-1)<=d_in;
            end if;
         end if;
        else
 hb <= "0000000000000000000000000000000000000000";
        end if;
end process;
 h_bus <= hb;
end behav;
```

A.1.9 cell_sort.vhd

```
-- This is the code for cell sort module. It identifies the type of
-- the cell based on header information. The possible results are
-- coded as follows.
--        empty    000
--        sig      001
--        vpoams   010
--        vpoame   011
--        vcoams   100
--        vcoame   101
--        ilmi     110
--        user     111
-- If the result of cell sort is a user cell or vpoams cell then signal
-- table_search is sent to route table module to find a match and determine
-- the call number of the cell. Also for user cell a signal traffic_search
-- is sent to traffic module to do UPC processing.

-- Programmed by Sandeep Agarwal


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

```
entity cell_sort is
  port(loaded, reset,pclk,hclk : in std_logic;
       header : in std_logic_vector(31 downto 0);
       done : out std_logic;
       unassigned_cell, table_search, traffic_search : out std_logic);
end cell_sort;

architecture behav of cell_sort is
signal result : std_logic_vector(2 downto 0);
begin
process(loaded,pclk)
begin
 if(reset='1') then
  unassigned_cell <= '0';
  table_search <= '0';
  traffic_search <= '0';
  done <= '0';
  result <= "000";
 elsif(reset='0') then
  if(hclk='0') then
    unassigned_cell <= '0';
    table_search <= '0';
    traffic_search <= '0';
    done <= '0';
    result <= "000";
  elsif(hclk='1') then
  if(pclk'event and pclk='0') then
   if(loaded='1') then

  -- point to point signalling  (VCI = 5)

    IF header(19 DOWNTO 4) = "0000000000000101" THEN
        result <= "001";

  -- OAM F4 flow based on VCI  (VCI = 3 or 4)

    ELSIF header(19 DOWNTO 4) = "0000000000000011" THEN
        result <= "010";

     ELSIF header(19 DOWNTO 4) = "0000000000000100" THEN
        result <= "011";

  -- OAM F5 flow based on PT

     ELSIF header(3 DOWNTO 1) = "100" THEN
        result <= "100";

     ELSIF header(3 DOWNTO 1) = "101" THEN
        result <= "101";

  -- User cell based on PT and not F4 OAM cell

     ELSIF (header(3 DOWNTO 1) = "000" OR header(3 DOWNTO 1) = "001" OR
            header(3 DOWNTO 1) = "010" OR header(3 DOWNTO 1) = "011") AND
            header(31 DOWNTO 4) /= "00000000" & "00000000" & "00000000" & "0000"
            THEN
         result <= "111";

  -- ILMI default cell
```

```
     ELSIF header(31 DOWNTO 4) = "00000000000000000000000010000" THEN
        result <= "110";

  -- unnassigned cell

     ELSE result <= "000";
     end if;
    done <= '1';
 end if;

  if(result /= "001") then
     if(result /= "110") then
       if(result= "000") then
          unassigned_cell <= '1';
       elsif(result="111") then
         table_search <= '1';
         traffic_search <= '1';
         unassigned_cell <= '0';
       else
         table_search <= '1';
         traffic_search <= '0';
         unassigned_cell <= '0';
       end if;
     end if;
  end if;
end if;
end if;
end if;
end process;
end behav;
```

A.1.10 route_table.vhd

```
-- This is the code for route table module. It receives 28 bits of cell
-- header through table_bus. When it receives signal table_search it
-- searches its CAM to find a match. If no match is found it sends signal
-- table_error. If one or more matches are found it sends the count of
-- the number of matches to the counter in ip_buffers through cnt_bus.
-- The result table_out is the call number of the cell.

-- Programmed by Sandeep Agarwal

library ieee,std;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;
use std.textio.all;

entity route_table is
--pragma synthesis_off
generic(rt_file_input : string :="file0");
--pragma synthesis_on
port(pclk, hclk, signal_address,table_search: in std_logic;
     table_bus : in std_logic_vector(27 downto 0);
     signal_control : in std_logic_vector(3 downto 0);
     count : out std_logic_vector(2 downto 0);
     table_done, table_error : out std_logic;
```

```
     table_out : out std_logic_vector(3 downto 0));
end route_table;

architecture behav of route_table is
begin

--pragma synthesis_off
read_in :
process(pclk,hclk)
file table_file : text is in rt_file_input;
variable rt_entry_line : line;
variable rt_entry : std_logic_vector(31 downto 0);
variable count1 : integer;
begin
  if(hclk'event and hclk='1') then
   count1 :=0;
   table_error <= '0';
   table_done <= '0';
          end if;
  if(hclk='1' and table_search='1') then
    while not endfile(table_file) loop
            if(pclk'event and pclk='1') then
    readline(table_file,rt_entry_line);
    read(rt_entry_line, rt_entry);
    if(rt_entry(31 downto 4)=table_bus) then
table_out<=rt_entry(3 downto 0);
count1:=count1+1;
    end if;
end if;
   end loop;
   count<=conv1_stdlogicvector(count1,3);
   table_done<='1';
   if(count1=0) then
table_error<='1';
   else
table_error<='0';
   end if;
 else
   table_done<='0';
 end if;
 if(hclk='0') then
   count <= "ZZZ";
   table_out <= "ZZZZ";
         end if;
end process;

--pragma synthesis_on
end behav;
```

A.1.11 traffic_module.vhd

```
-- This is the code for traffic module. It receives the 28 bits of cell header
-- through table_bus. When signal traffic_search is received it does UPC
-- processing. The possible results are cell_discard and two level cell
-- loss priority.

-- Programmed by Sandeep Agarwal
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;
use std.textio.all;

entity traffic_table is
--pragma synthesis_off
 generic(tt_file_input : string :="trafficfile0");
--pragma synthesis_on
        port(pclk, hclk, signal_address,traffic_search: in std_logic;
             table_bus : in std_logic_vector(27 downto 0);
             signal_control : in std_logic_vector(3 downto 0);
             traffic_done : out std_logic;
             cell_discard, CLP : out std_logic);
end traffic_table;

architecture behav of traffic_table is
begin
--pragma synthesis_off
read_in :
        process(pclk,hclk)
        file traffic_file : text is in tt_file_input;
        variable entry_line : line;
        variable entry : std_logic_vector(1 downto 0);
        begin
  if(hclk'event and hclk='1') then
   cell_discard <= '0';
   CLP <= 'Z';
          end if;
          if(hclk='1' and traffic_search='1') then
            while not endfile(traffic_file) loop
                if(pclk'event and pclk='1') then
                    readline(traffic_file,entry_line);
                    read(entry_line, entry);
                    if(entry(1)='1') then
                      cell_discard<='1';
                      CLP <= 'Z';
      traffic_done<='1';
                    elsif(entry(0)='1') then
                      CLP <= '1';
                      cell_discard<='0';
      traffic_done<='1';
                    elsif(entry(0)='0') then
                      CLP <= '0';
                      cell_discard<='0';
      traffic_done<='1';
                    end if;
                end if;
             end loop;
            end if;
       end process;
-- pragma synthesis_on
 end behav;
```

A.1.12 input_controller.vhd

```
-- This is the code for the input controller. It consists of header buffer,
```

```
-- cell sort module, traffic module and route table. It sends the call
-- number of the cell and 12 bits of cell header to the output module. It
-- also sends the count of the number of destination ports to the input
-- shift register buffers.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use work.atm_pkg.all;

entity input_controller is
--pragma synthesis_off
 generic(port_id : integer := 1;
 h_size : integer := 40;
 rt_file_input : string :="file0";
 tt_file_input : string :="trafficfile0");
--pragma synthesis_on
 port(hclk, pclk, d_in, rs,reset : in std_logic;
      s_bus_data : out std_logic_vector(3 downto 0);
      header_data : out std_logic_vector(11 downto 0);
      valid_cell : out std_logic;
      hp : out std_logic;
      loading_sig : out std_logic;
      cnt_bus : out std_logic_vector(2 downto 0));
end input_controller;

architecture struc of input_controller is

component head_buffer
        generic (h_size : integer := 40);
        port(hclk,pclk,d_in,rs : in std_logic;
             h_bus : out std_logic_vector(h_size-1 downto 0));
end component;

component cell_sort
  port(loaded, reset,pclk,hclk : in std_logic;
       header : in std_logic_vector(31 downto 0);
       done : out std_logic;
       unassigned_cell, table_search, traffic_search : out std_logic);
end component;

component route_table
       -- pragma synthesis_off
        generic(rt_file_input : string :="file0");
       -- pragma synthesis_on
        port(pclk, hclk, signal_address,table_search: in std_logic;
             table_bus : in std_logic_vector(27 downto 0);
             signal_control : in std_logic_vector(3 downto 0);
             count : out std_logic_vector(2 downto 0);
             table_done, table_error : out std_logic;
             table_out : out std_logic_vector(3 downto 0));
end component;

component traffic_table
        -- pragma synthesis_off
        generic(tt_file_input : string :="trafficfile0");
        -- pragma synthesis_on
        port(pclk, hclk, signal_address,traffic_search: in std_logic;
             table_bus : in std_logic_vector(27 downto 0);
```

```
            signal_control : in std_logic_vector(3 downto 0);
            traffic_done : out std_logic;
            cell_discard, CLP : out std_logic);
end component;

signal s1, s2, s3, s4, s5,s7,s8,s9,s10,s11 : std_logic;
signal port_address : std_logic_vector(2 downto 0);
signal s0 : std_logic_vector(39 downto 0);
--signal s0 : std_logic_vector(h_size-1 downto 0);
signal s6 : std_logic_vector(3 downto 0);
signal max_count : integer:= port_id+1;
signal count1 : integer :=0;
begin
-- pragma synthesis_off
P_p_address:
        process
        begin
                port_address<=conv1_StdlogicVector(port_id,3);
                wait ;
        end process;
-- pragma synthesis_on

u0:     head_buffer
-- pragma synthesis_off
        generic map(h_size)
-- pragma synthesis_on
        port map(hclk,pclk,d_in,rs,s0);

u1:     cell_sort
        port map(rs,reset,pclk,hclk,s0(39 downto 8),s1,s2,s3,s4);

u2:    route_table
-- pragma synthesis_off
       generic map(rt_file_input)
-- pragma synthesis_on
       port map(pclk,hclk,s5,s3,s0(39 downto 12),s6,cnt_bus,s7,s8,
                s_bus_data(3 downto 0));

u3:   traffic_table
-- pragma synthesis_off
      generic map(tt_file_input)
-- pragma synthesis_on
      port map(pclk,hclk,s5,s4,s0(39 downto 12),s6,s9,s10,s11);

sig_gen: process(pclk)
      begin
        if(hclk='0') then
          valid_cell <= '0';
        elsif(s7='1' and s9='1' and hclk='1') then
          if(s2='1' or s8='1' or s10='1') then
            valid_cell <= '0';
          else
           valid_cell <= '1';
          end if;
        end if;
   end process;

loading_sig_gen:
 process(pclk,hclk,s7)
 variable count: integer :=0;
```

```
 variable reset: std_logic;
   begin
       if(pclk'event and pclk='0') then
        if(hclk='0') then
          reset:='0';
        elsif(hclk='1' and reset='0' and s7='1') then
          count := count+1;
          count1<=count;
        end if;
        if(count=max_count) then
         reset:='1';
         count:=0;
        end if;
       end if;
 end process;
--pragma synthesis_off
 process(pclk)
  begin
         if(pclk'event and pclk='1') then
           if(count1=port_id) then
              loading_sig <= '1';
           else
              loading_sig <= '0';
           end if;
         end if;
 end process;
--pragma synthesis_on


     header_data(7 downto 0) <= s0(7 downto 0);
     header_data(8) <= s11;
     header_data(11 downto 9) <= s0(11 downto 9);
     hp <= s7;
end struc;
```

A.1.13 crosspoint.vhd

```
-- This is the code of one cross-point of the crossbar switch used in
-- localized concentrators. On receiving signals c_sig and r1 the connections
-- are established whereas on receiving signal rst the connections are
-- broken.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;

entity crosspoint is
port(c_sig,rst : in std_logic;
     r1 : in std_logic;
     pclk : in std_logic;
     cin : in std_logic;
     cout : out std_logic);
end crosspoint;

architecture behav of crosspoint is
signal enable : std_logic;
begin
```

```
P_enable :
process(cin,pclk)
begin
   if(rst='1') then
enable<='0';
   elsif(r1='1' and c_sig='1') then
if(pclk'event and pclk='0') then
    enable<='1';
end if;
   end if;
end process;

P_cout:
process(pclk)
begin
 if(pclk'event and pclk='1') then
   if(enable='1') then
cout<=cin;
   else
cout<='Z';
   end if;
          end if;
end process;

end behav;
```

A.1.14 ip_concentrator.vhd

```
-- This is the code for the concentrator of an input port. Since the
-- number of input and output ports in the VR switch is assumed to be 4 and
-- each port has 7 buffers the concentrator has 28 crosspoints.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;

entity ip_concentrator is
port(cin : in std_logic_vector(6 downto 0);
     cout : out std_logic_vector(3 downto 0);
     c_sig : in std_logic_vector(3 downto 0);
     r1 : in std_logic_vector(6 downto 0);
     pclk,rst : in std_logic);
end ip_concentrator;

architecture struc of ip_concentrator is
component crosspoint
        port(c_sig,rst : in std_logic;
             r1 : in std_logic;
             pclk: in std_logic;
             cin : in std_logic;
             cout : out std_logic);
end component;
begin
U00: crosspoint port map(c_sig(0),rst,r1(0),pclk,cin(0),cout(0));
U10: crosspoint port map(c_sig(0),rst,r1(1),pclk,cin(1),cout(0));
U20: crosspoint port map(c_sig(0),rst,r1(2),pclk,cin(2),cout(0));
```

```
U30: crosspoint port map(c_sig(0),rst,r1(3),pclk,cin(3),cout(0));
U40: crosspoint port map(c_sig(0),rst,r1(4),pclk,cin(4),cout(0));
U50: crosspoint port map(c_sig(0),rst,r1(5),pclk,cin(5),cout(0));
U60: crosspoint port map(c_sig(0),rst,r1(6),pclk,cin(6),cout(0));

U01: crosspoint port map(c_sig(1),rst,r1(0),pclk,cin(0),cout(1));
U11: crosspoint port map(c_sig(1),rst,r1(1),pclk,cin(1),cout(1));
U21: crosspoint port map(c_sig(1),rst,r1(2),pclk,cin(2),cout(1));
U31: crosspoint port map(c_sig(1),rst,r1(3),pclk,cin(3),cout(1));
U41: crosspoint port map(c_sig(1),rst,r1(4),pclk,cin(4),cout(1));
U51: crosspoint port map(c_sig(1),rst,r1(5),pclk,cin(5),cout(1));
U61: crosspoint port map(c_sig(1),rst,r1(6),pclk,cin(6),cout(1));

U02: crosspoint port map(c_sig(2),rst,r1(0),pclk,cin(0),cout(2));
U12: crosspoint port map(c_sig(2),rst,r1(1),pclk,cin(1),cout(2));
U22: crosspoint port map(c_sig(2),rst,r1(2),pclk,cin(2),cout(2));
U32: crosspoint port map(c_sig(2),rst,r1(3),pclk,cin(3),cout(2));
U42: crosspoint port map(c_sig(2),rst,r1(4),pclk,cin(4),cout(2));
U52: crosspoint port map(c_sig(2),rst,r1(5),pclk,cin(5),cout(2));
U62: crosspoint port map(c_sig(2),rst,r1(6),pclk,cin(6),cout(2));

U03: crosspoint port map(c_sig(3),rst,r1(0),pclk,cin(0),cout(3));
U13: crosspoint port map(c_sig(3),rst,r1(1),pclk,cin(1),cout(3));
U23: crosspoint port map(c_sig(3),rst,r1(2),pclk,cin(2),cout(3));
U33: crosspoint port map(c_sig(3),rst,r1(3),pclk,cin(3),cout(3));
U43: crosspoint port map(c_sig(3),rst,r1(4),pclk,cin(4),cout(3));
U53: crosspoint port map(c_sig(3),rst,r1(5),pclk,cin(5),cout(3));
U63: crosspoint port map(c_sig(3),rst,r1(6),pclk,cin(6),cout(3));
end struc;
```

A.1.15 ip_port.vhd

```
-- This is the code for input port. It consists of input shift register
-- buffers, input controller, input scheduler and input concentrator.
-- It sends the call number of the cell to the output module through
-- s_bus_data, 12 bits of cell header through header_data and the input
-- address of the cell through add_bus.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;

entity ip_port is
  generic(port_id : integer :=1;
          port_number : integer :=16;
--pragma synthesis_off
  rt_file_input : string :="file0";
  tt_file_input : string :="trafficfile0");
--pragma synthesis_on
  port(hclk, pclk, d_in, reset : in std_logic;
       r_bus : in std_logic_vector(6 downto 0);
       loading_sig : out std_logic;
       s_bus_data : out std_logic_vector(3 downto 0);
       header_data : out std_logic_vector(11 downto 0);
       add_bus : out std_logic_vector(6 downto 0);
```

```
       d_out : out std_logic_vector(3 downto 0));
end ip_port;

architecture struc of ip_port is

component input_controller
--pragma synthesis_off
 generic(port_id : integer :=1;
         h_size : integer := 40;
 rt_file_input : string :="file0";
 tt_file_input : string :="trafficfile0");
--pragma synthesis_on
 port(hclk, pclk, d_in, rs,reset : in std_logic;
      s_bus_data : out std_logic_vector(3 downto 0);
      header_data : out std_logic_vector(11 downto 0);
      valid_cell : out std_logic;
      hp : out std_logic;
      loading_sig : out std_logic;
      cnt_bus : out std_logic_vector(2 downto 0));
end component;

component ip_buffers
        generic(port_number : integer := 16);
        port(din,hclk,pclk,reset: in std_logic;
             c_bus:in std_logic_vector(4 downto 0);
             r_bus,w_bus : in std_logic_vector(6 downto 0);
             r2 : out std_logic_vector(6 downto 0);
             dout:out std_logic_vector(6 downto 0);
             cnt_bus : inout std_logic_vector(6 downto 0);
             add_bus : out std_logic_vector(6 downto 0));
end component;

component ip_scheduler
        generic(p_size : integer := 40;
                h_size : integer := 40;
port_id : integer := 1);
        port(hclk,pclk,reset,v_cell:in std_logic;
             cnt_bus : in std_logic_vector(6 downto 0);
             write_bus : out std_logic_vector(6 downto 0);
             rst,rs,rs1,new_clk : out std_logic;
     c_sig : out std_logic_vector(3 downto 0));
end component;

component ip_concentrator
        port(cin : in std_logic_vector(6 downto 0);
             cout : out std_logic_vector(3 downto 0);
             c_sig : in std_logic_vector(3 downto 0);
             r1 : in std_logic_vector(6 downto 0);
             pclk,rst : in std_logic);
end component;

signal c_bus : std_logic_vector(4 downto 0);
signal s0,s1,s2,s3 : std_logic_vector(6 downto 0);
signal s4 : std_logic;
signal s6 : std_logic_vector(3 downto 0);

begin

c0: input_controller
```

```
--pragma synthesis_off
 generic map(port_id,40,rt_file_input,tt_file_input)
--pragma synthesis_on
 port map(hclk,pclk,d_in,c_bus(3),reset,s_bus_data,header_data,c_bus(1),
         c_bus(0),loading_sig,s3(2 downto 0));

c1: ip_buffers

 generic map(port_number)
port map(d_in,hclk,pclk,reset,c_bus,r_bus,s0,s1,s2,s3,add_bus);

c2: ip_scheduler

 generic map(40,40,port_id)
 port map(hclk,pclk,reset,c_bus(1),s3,s0,c_bus(2),c_bus(3),c_bus(4),s4,s6);

c3: ip_concentrator

 port map(s2,d_out,s6,s1,pclk,c_bus(2));

end struc;
```

A.1.16 input_port.vhd

```
-- This is the code for one input port of the VR switch. It receives
-- the incoming cell through d_in, sends the call number of the cell, 12 bits
-- of cell header and input address of the cell to the output module through
-- sel_bus, header_bus and add_bus respectively. It sends data to the output
-- module through d_out.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;

entity input_port is
--pragma synthesis_off
  generic(port_id : integer :=1;
          port_number : integer :=16;
          rt_file_input : string :="file0";
          tt_file_input : string :="trafficfile0");
--pragma synthesis_on
  port(hclk, pclk, d_in, reset : in std_logic;
       add_bus : inout std_logic_vector(6 downto 0);
       sel_bus : out std_logic_vector(3 downto 0);
       header_bus : out std_logic_vector(11 downto 0);
       d_out : out std_logic_vector(3 downto 0));
end input_port;

architecture struc of input_port is

component ip_port
--pragma synthesis_off
  generic(port_id : integer :=1;
          port_number : integer :=16;
          rt_file_input : string :="file0";
```

```
          tt_file_input : string :="trafficfile0");
--pragma synthesis_on
  port(hclk, pclk, d_in, reset : in std_logic;
       r_bus : in std_logic_vector(6 downto 0);
       loading_sig : out std_logic;
       s_bus_data : out std_logic_vector(3 downto 0);
       header_data : out std_logic_vector(11 downto 0);
       add_bus : out std_logic_vector(6 downto 0);
       d_out : out std_logic_vector(3 downto 0));
end component;

signal s0 : std_logic_vector(6 downto 0);
signal s1 : std_logic_vector(6 downto 0);
signal s2 : std_logic_vector(3 downto 0);
signal s3 : std_logic_vector(11 downto 0);
signal s4 : std_logic;

begin

 uo: ip_port
--pragma synthesis_off
   generic map(port_id,port_number,rt_file_input,tt_file_input)
--pragma synthesis_on
   port map(hclk, pclk, d_in, reset,s0,s4,s2,s3,s1,d_out);

P_tristate:
process(add_bus)
  begin
  if(hclk='0') then
    s0 <= add_bus;
    add_bus <= "ZZZZZZZ";
    sel_bus <= "ZZZZ";
    header_bus <= "ZZZZZZZZZZZZ";
 else
     s0 <= "ZZZZZZZ";
     add_bus <= "ZZZZZZZ";
     sel_bus <= "ZZZZ";
     header_bus <= "ZZZZZZZZZZZZ";
 end if;
end process;
process(pclk,add_bus)
    begin
      if(pclk'event and pclk='1') then
         if(hclk='1' and s4='1') then
           add_bus <= s1;
           s0 <= "ZZZZZZZ";
           sel_bus <= s2;
           header_bus <= s3;
        else
           s0 <= "ZZZZZZZ";
           add_bus <= "ZZZZZZZ";
           sel_bus <= "ZZZZ";
           header_bus <= "ZZZZZZZZZZZZ";
        end if;
      end if;
  end process;

end struc;
```

A.1.17 input_module.vhd

```
-- This is the code for input module of the VR switch. It has 4 input ports
-- which receive data through d_in_0, d_in_1, d_in_2, d_in_3 respectively.
-- Virtual address routing is done through sel_bus, header_bus and add_bus.
-- Data is sent to output module through d_out.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;

entity input_module is
 port(hclk, pclk,reset : in std_logic;
      add_bus : inout std_logic_vector(6 downto 0);
      sel_bus : out std_logic_vector(3 downto 0);
      header_bus : out std_logic_vector(11 downto 0);
      d_in_0, d_in_1, d_in_2, d_in_3 : in std_logic;
      d_out : out std_logic_vector(3 downto 0));
end input_module;

architecture struc of input_module is

component input_port
--pragma synthesis_off
  generic(port_id : integer :=1;
          port_number : integer :=16;
          rt_file_input : string :="file0";
          tt_file_input : string :="trafficfile0");
--pragma synthesis_on
  port(hclk, pclk, d_in, reset : in std_logic;
       add_bus : inout std_logic_vector(6 downto 0);
       sel_bus : out std_logic_vector(3 downto 0);
       header_bus : out std_logic_vector(11 downto 0);
       d_out : out std_logic_vector(3 downto 0));
end component;

begin

 u0: input_port
--pragma synthesis_off
   generic map(1,16,"file0","trafficfile0")
--pragma synthesis_on
   port map(hclk, pclk, d_in_0, reset, add_bus, sel_bus, header_bus, d_out);

 u1: input_port
--pragma synthesis_off
   generic map(2,32,"file1","trafficfile1")
--pragma synthesis_on
   port map(hclk, pclk, d_in_1, reset, add_bus, sel_bus, header_bus, d_out);

 u2: input_port
--pragma synthesis_off
   generic map(3,48,"file2","trafficfile2")
--pragma synthesis_on
   port map(hclk, pclk, d_in_2, reset, add_bus, sel_bus, header_bus, d_out);

 u3: input_port
```

```
--pragma synthesis_off
   generic map(4,64,"file3","trafficfile3")
--pragma synthesis_on
   port map(hclk, pclk, d_in_3, reset, add_bus, sel_bus, header_bus, d_out);

end struc;
```

A.2 Output Module

A.2.1 add_fifo.vhd

```
-- This is the code for add_fifo. It receives write address from lookup
-- table and stores the input buffer address of the cell in a FIFO, when hclk
-- is high. When hclk is low and it is in read mode then it sends the address
-- at the head of the FIFO to the input module and to the lookup table.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity add_fifo is
 port(r,w,hclk,pclk : in std_logic;
      add_in : in std_logic_vector(6 downto 0);
      add_out : out std_logic_vector(6 downto 0));
end add_fifo;

architecture behav of add_fifo is
type fifo is array(natural range <>) of std_logic_vector(6 downto 0);
 signal temp : fifo(3 downto 0);
begin
 process(add_in,hclk,pclk)
 variable r_p,w_p : integer :=0;
  begin
   if(pclk'event and pclk='1') then
     if(hclk='1' and w='1') then
        temp(w_p) <= add_in;
        w_p := w_p+1;
      elsif(hclk='0' and r='1') then
        add_out <= temp(0);
        w_p := w_p-1;
        temp(2 downto 0) <= temp(3 downto 1);
        temp(3) <="0000000";
      else
       add_out <= "ZZZZZZZ";
     end if;
   end if;
 end process;
end behav;
```

A.2.2 call_no_fifo.vhd

```
-- This is the code for call_no_fifo. When hclk is high and write signal is
-- received from lookup table then it stores the call number of the cell in
```

```
-- its FIFO. When hclk is low and read signal is received from output
-- scheduler it sends the call number at the head of the FIFO to the lookup
-- table.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity call_no_fifo is
 port(r,w,hclk,pclk : in std_logic;
      add_in : in std_logic_vector(3 downto 0);
      add_out : out std_logic_vector(3 downto 0));
end call_no_fifo;

architecture behav of call_no_fifo is
type fifo is array(natural range <>) of std_logic_vector(3 downto 0);
 signal temp : fifo(3 downto 0);
begin
 process(add_in,hclk,pclk)
 variable r_p,w_p : integer :=0;
  begin
   if(pclk'event and pclk='0') then
     if(hclk='1' and w='1') then
        temp(w_p) <= add_in;
        w_p := w_p+1;
      elsif(hclk='0' and r='1') then
        add_out <= temp(0);
        w_p := w_p-1;
        temp(2 downto 0) <= temp(3 downto 1);
        temp(3) <="0000";
      else
       add_out <= "ZZZZ";
     end if;
   end if;
 end process;
end behav;
```

A.2.3 header_fifo.vhd

```
-- This is the code for the header_fifo. When hclk is high and write signal
-- is received from lookup table then it stores the 12 bits of cell header
-- in its FIFO. When hclk is low and read signal is received from output
-- scheduler then it sends the data at the head of its FIFO to op_buffer.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity header_fifo is
 port(r,w,hclk,pclk : in std_logic;
      add_in : in std_logic_vector(11 downto 0);
      add_out : out std_logic_vector(11 downto 0));
end header_fifo;
```

```
architecture behav of header_fifo is
type fifo is array(natural range <>) of std_logic_vector(11 downto 0);
 signal temp : fifo(3 downto 0);
begin
 process(add_in,hclk,pclk)
 variable r_p,w_p : integer :=0;
  begin
   if(pclk'event and pclk='0') then
     if(hclk='1' and w='1') then
        temp(w_p) <= add_in;
        w_p := w_p+1;
      elsif(hclk='0' and r='1') then
        add_out <= temp(0);
        w_p := w_p-1;
        temp(2 downto 0) <= temp(3 downto 1);
        temp(3) <="000000000000";
     end if;
   end if;
 end process;
end behav;
```

A.2.4 lookup_table.vhd

```
-- This is the code for lookup table at the output port. When hclk is high
-- it reads the data table_in and searches its CAM to find a match. If a
-- match is found it sends write signal to add_fifo, call_no_fifo and
-- header_fifo. When hclk is low it reads the data table_in, does VPI/VCI
-- translation and sends updated header to table_out.

-- Programmed by Sandeep Agarwal

library ieee,std;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.atm_pkg.all;
use std.textio.all;

entity lookup_table is
--pragma synthesis_off
generic(op_file_input : string :="op_file0";
num_entry : integer := 13);
--pragma synthesis_on
        port(pclk, hclk : in std_logic;
             table_in : in std_logic_vector(6 downto 0);
             write_sig : out std_logic;
             table_out : out std_logic_vector(27 downto 0));
end lookup_table;

architecture behav of lookup_table is
begin

--pragma synthesis_off
read_in :
        process(table_in,hclk)
        file lookup_file : text is in op_file_input;
        variable lookup_entry_line : line;
        variable lookup_entry,table_temp : std_logic_vector(34 downto 0);
        variable count1,i : integer :=0;
```

```
variable flag : boolean := true;
type type_table is array(natural range <>) of std_logic_vector(34 downto 0);
variable l_table : type_table(0 to num_entry-1);
        begin
if(flag=true) then
    i:=0;
            while not endfile(lookup_file) loop
                    readline(lookup_file,lookup_entry_line);
                    read(lookup_entry_line, lookup_entry);
    l_table(i):=lookup_entry;
    i:=i+1;
    end loop;
end if;
flag:=false;

        count1 :=0;
i:=0;
        if(hclk='1') then
    while i<num_entry loop
    table_temp:=l_table(i);
                    if(table_temp(34 downto 28)=table_in) then
                      count1:=1;
                    end if;
    i:=i+1;
           end loop;
           if(count1=1) then
              write_sig <='1';
           else
             write_sig<='0';
           end if;
          elsif(hclk='0') then
    i:=0;
    while i<num_entry loop
       table_temp:=l_table(i);
               if(table_temp(34 downto 28)=table_in) then
                  table_out<=table_temp(27 downto 0);
                end if;
i:=i+1;
           end loop;
         end if;
 end process;
--pragma synthesis_on
end behav;
```

A.2.5 op_buffer.vhd

```
-- This is the code for op_buffer. It stores the complete cell header when hclk
-- is low and transmits it out of the switch when hclk is high. It receives
-- 12 bits of cell header from header_fifo and 28 bits of cell header from
-- lookup table.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;

entity op_buffer is
 port(add_in : in std_logic_vector(39 downto 0);
```

```
      hclk,pclk,r,rs : in std_logic;
      add_out : out std_logic);
end op_buffer;

architecture behav of op_buffer is
signal temp : std_logic_vector(39 downto 0);
begin
 process(pclk)
 begin
  if(pclk'event and pclk='1') then
    if(r='1') then
      if(hclk='0') then
        temp(11 downto 0) <= add_in(11 downto 0);
        temp(39 downto 12) <= add_in(39 downto 12);
       elsif(hclk='1' and rs='0') then
        add_out <= temp(0);
temp(38 downto 0) <= temp(39 downto 1);
temp(39) <= '0';
      else
       add_out <= 'Z';
      end if;
    else
       add_out <= 'Z';
    end if;
   end if;
 end process;
end behav;
```

A.2.6 op_scheduler.vhd

```
-- This is the code for output scheduler. It sends read signals to add_fifo,
-- call_no_fifo and header_fifo. It also sends read signal to op_buffer.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity op_scheduler is
 generic(port_id : integer :=1);
 port(hclk,pclk,reset : in std_logic;
      sig_out1, sig_out2, sig_out3 : out std_logic);
end op_scheduler;

architecture behav of op_scheduler is
signal d_out : std_logic_vector(2 downto 0);
signal port_no : std_logic_vector(2 downto 0);
signal max_count : integer:=0;
signal rst : std_logic;
signal hclk_d, sig1 : std_logic;
begin
--pragma_synthesis_off
 port_no <= std_logic_vector(conv_unsigned(port_id,3));
 max_count <= port_id+1;
--pragma_synthesis_on

process(pclk,hclk)
```

```
 variable count: integer :=0;
 variable r_set: std_logic;
   begin
       if(pclk'event and pclk='0') then
if(hclk='1') then
  r_set:='0';
        elsif(hclk='0' and r_set='0') then
          count := count+1;
          d_out <= std_logic_vector(conv_unsigned(count,3));
        end if;
        if(count=max_count) then
 r_set:='1';
 count:=0;
        end if;
       end if;
 end process;

process(pclk,reset)
 begin
       if(pclk'event and pclk='1') then
if(reset='1') then
 sig1 <='0';
        elsif(reset='0' and d_out=port_no) then
          sig1 <= '1';
        else
          sig1 <= '0';
        end if;
       end if;
 end process;

process(pclk,hclk,reset)
variable count: integer :=0;
begin
   if(reset='1') then
       sig_out2 <= '0';
   elsif(reset='0' and hclk='1') then
     if(pclk'event and pclk='0') then
        count := count+1;
      if(count=40) then
        sig_out2 <= '1';
      end if;
     end if;
   else
    count := 0;
    sig_out2 <= '0';
   end if;
end process;

P_rst:
        process(hclk_d,hclk,reset)
        begin
            if(reset='1') then
                rst<='1';
            else
                if(hclk_d='0') then
                    rst<='0';
                elsif(hclk='0') then
                    rst<='1';
                end if;
            end if;
```

```
        end process;

P_hclk_d:
        process(pclk)
        begin
                if(pclk'event and pclk='0') then
                    hclk_d<=hclk;
                end if;
        end process;

P_sig_out3:
       process(pclk,rst,sig1,reset)
        begin
             if(rst/='1' and reset/='1') then
                if(pclk'event and pclk='0') then
                    if(sig1='1') then
                        sig_out3<=sig1;
                    end if;
                end if;
             else
                        sig_out3<='0';
             end if;
        end process;
sig_out1 <= sig1;
end behav;
```

A.2.7 op_port.vhd

```
-- This is the code for op_port. It consists of lookup table, add_fifo,
-- call_no_fifo, header_fifo, op_buffer and output scheduler.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;

entity op_port is
--pragma synthesis_off
 generic(port_id : integer :=1;
 op_file_input : string :="op_file0");
--pragma synthesis_on
 port(hclk,pclk,reset : in std_logic;
      s_bus : in std_logic_vector(3 downto 0);
      a1_bus : in std_logic_vector(6 downto 0);
      a2_bus : out std_logic_vector(6 downto 0);
      h_bus : in std_logic_vector(11 downto 0);
      d_in : in std_logic;
      d_out,r : out std_logic);
end op_port;

architecture struc of op_port is

component lookup_table
--pragma synthesis_off
        generic(op_file_input : string;
                num_entry : integer := 13);
--pragma synthesis_on
        port(pclk, hclk : in std_logic;
```

```
            table_in : in std_logic_vector(6 downto 0);
            write_sig : out std_logic;
            table_out : out std_logic_vector(27 downto 0));
end component;

component add_fifo
        port(r,w,hclk,pclk : in std_logic;
            add_in : in std_logic_vector(6 downto 0);
            add_out : out std_logic_vector(6 downto 0));
end component;

component call_no_fifo
        port(r,w,hclk,pclk : in std_logic;
            add_in : in std_logic_vector(3 downto 0);
            add_out : out std_logic_vector(3 downto 0));
end component;

component header_fifo
        port(r,w,hclk,pclk : in std_logic;
            add_in : in std_logic_vector(11 downto 0);
            add_out : out std_logic_vector(11 downto 0));
end component;

component op_buffer
        port(add_in : in std_logic_vector(39 downto 0);
            hclk,pclk,r,rs : in std_logic;
            add_out : out std_logic);
end component;

component op_scheduler
--pragma synthesis_off
generic(port_id : integer :=1);
--pragma synthesis_on
port(hclk,pclk,reset : in std_logic;
     sig_out1, sig_out2, sig_out3 : out std_logic);
end component;

signal s0,s6 : std_logic_vector(6 downto 0);
signal s1,s7,s8, s9 : std_logic;
signal s2 : std_logic_vector(27 downto 0);
signal s3 : std_logic_vector(3 downto 0);
signal s4 : std_logic_vector(11 downto 0);
signal s5 : std_logic_vector(39 downto 0);
begin

  u0: lookup_table
--pragma synthesis_off
    generic map(op_file_input,13)
--pragma synthesis_on
    port map(pclk, hclk,s0,s1,s2);

  u1: add_fifo
   port map(s7,s1,hclk,pclk,a1_bus,s6);

  u2: call_no_fifo
   port map(s7,s1,hclk,pclk,s_bus,s3);

  u3: header_fifo
   port map(s7,s1,hclk,pclk,h_bus,s4);
```

```
  u4: op_buffer
   port map(s5,hclk,pclk,s9,s8,d_out);

  u5: op_scheduler
--pragma synthesis_off
   generic map(port_id)
--pragma synthesis_on
   port map(hclk,pclk,reset,s7,s8,s9);

 process(hclk,s_bus,a1_bus,s6,s3)
   begin
    if(hclk='1') then
      s0(6 downto 4) <= a1_bus(6 downto 4);
      s0(3 downto 0) <= s_bus(3 downto 0);
    elsif(hclk='0') then
      s0(6 downto 4) <= s6(6 downto 4);
      s0(3 downto 0) <= s3;
    end if;
 end process;
   s5(11 downto 0) <= s4;
   s5(39 downto 12) <= s2;
   r <= s7;
   a2_bus <= s6;
   d_out<=d_in;

end struc;
```

A.2.8 output_port.vhd

```
-- This is the code for output port of the VR switch. It receives data
-- for virtual address routing through sel_bus, add_bus and header_bus.
-- It receives the data for real cell routing through d_in which is only
-- the payload of the cell. The header of the cell is in the op_buffer
-- and is transmitted out of the switch from there.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;

entity output_port is
--pragma synthesis_off
   generic(port_id : integer :=1;
   op_file_input : string :="op_file0");
--pragma synthesis_on
   port(hclk,pclk,reset : in std_logic;
        sel_bus : in std_logic_vector(3 downto 0);
        add_bus : inout std_logic_vector(6 downto 0);
        header_bus : in std_logic_vector(11 downto 0);
        d_in : in std_logic;
        d_out : out std_logic);
end output_port;

architecture struc of output_port is

component op_port
--pragma synthesis_off
 generic(port_id : integer :=1;
```

```
 op_file_input : string :="op_file0");
--pragma synthesis_on
 port(hclk,pclk,reset : in std_logic;
      s_bus : in std_logic_vector(3 downto 0);
      a1_bus : in std_logic_vector(6 downto 0);
      a2_bus : out std_logic_vector(6 downto 0);
      h_bus : in std_logic_vector(11 downto 0);
      d_in : in std_logic;
      d_out,r : out std_logic);
end component;

signal s1: std_logic_vector(3 downto 0);
signal s2,s3: std_logic_vector(6 downto 0);
signal s4: std_logic_vector(11 downto 0);
signal s0: std_logic;
begin

  u0: op_port
--pragma synthesis_off
    generic map(port_id,op_file_input)
--pragma synthesis_on
    port map(hclk,pclk,reset,s1,s2,s3,s4,d_in,d_out,s0);

 process(hclk,sel_bus,add_bus,header_bus,s3,pclk)
  begin
   if(hclk='1') then
    s1 <= sel_bus;
    s2 <= add_bus;
    s4 <= header_bus;
   else
     s1 <= "ZZZZ";
     s2 <= "ZZZZZZZ";
    s4 <= "ZZZZZZZZZZZZ";
   end if;
   if(hclk='0' and s0='1') then
    add_bus <= s3;
   elsif(pclk'event and pclk='1') then
    add_bus <= "ZZZZZZZ";
   end if;
 end process;

end struc;
```

A.2.9 output_module.vhd

```
-- This is the code for output module of the VR switch. It consists of
-- four output ports.

-- Programmed by Sandeep Agarwal

library ieee;
use ieee.std_logic_1164.all;

entity output_module is
   port(hclk,pclk,reset : in std_logic;
        sel_bus : in std_logic_vector(3 downto 0);
        add_bus : inout std_logic_vector(6 downto 0);
        header_bus : in std_logic_vector(11 downto 0);
```

```
        d_in : in std_logic_vector(3 downto 0);
        d_out_0, d_out_1, d_out_2, d_out_3 : out std_logic);
end output_module;

architecture struc of output_module is

component output_port
--pragma synthesis_off
   generic(port_id : integer :=1;
   op_file_input : string :="op_file0");
--pragma synthesis_on
   port(hclk,pclk,reset : in std_logic;
        sel_bus : in std_logic_vector(3 downto 0);
        add_bus : inout std_logic_vector(6 downto 0);
        header_bus : in std_logic_vector(11 downto 0);
        d_in : in std_logic;
        d_out : out std_logic);
end component;

begin

 u0: output_port
--pragma synthesis_off
    generic map(1,"op_file0")
--pragma synthesis_on
    port map(hclk,pclk,reset,sel_bus,add_bus,header_bus,d_in(0),d_out_0);

 u1: output_port
--pragma synthesis_off
    generic map(2,"op_file1")
--pragma synthesis_on
    port map(hclk,pclk,reset,sel_bus,add_bus,header_bus,d_in(1),d_out_1);

 u2: output_port
--pragma synthesis_off
    generic map(3,"op_file2")
--pragma synthesis_on
    port map(hclk,pclk,reset,sel_bus,add_bus,header_bus,d_in(2),d_out_2);

 u3: output_port
--pragma synthesis_off
    generic map(4,"op_file3")
--pragma synthesis_on
    port map(hclk,pclk,reset,sel_bus,add_bus,header_bus,d_in(3),d_out_3);

end struc;
```

VITA

Surname: Agarwal ***Given Names:*** Sandeep

Place of Birth: Lucknow, India ***Date of Birth:*** Jan. 18, 1971

Educational Institutions Attended

University of Victoria, Victoria, Canada	1996 to 1998
Government Engineering College, Rewa, India	1988 to 1993

Degrees Awarded

Bachelor of Engineering	Awadesh Pratap Singh University	1993

Honors and Awards

University of Victoria Research Assistantship	1996 to 1998
University of Victoria Graduate Teaching Fellowship	1996 to 1998

Publications

1. F. El-Guibaly and S. Agarwal, "Design and performance analysis of shift register-based ATM switch," *IEEE Proc. Pacific Rim Conference on Communications, Computers and Signal Processing,* Victoria, August 1997, pp. 70-73.
2. S. Agarwal and F. El-Guibaly, "Modelling of shift register-based ATM switch," *IEEE Proc. Eighth Great Lakes Symposium on VLSI,* Lafayette, Louisiana, February 19-21, 1998, pp. 146-151.

Work Submitted for Publication

1. A. Rahyan, A. Almulhem, F. El-Guibaly and S. Agarwal, "Fault tolerant ATM switch using logical neighbourhood network"

PARTIAL COPYRIGHT LICENSE

Title of Dissertation: DESIGN AND PERFORMANCE ANALYSIS OF A NEW ATM SWITCH.

Author: 

SANDEEP AGARWAL
14 December 1998