Design and Implementation of a Per-Flow Queue Manager for an ATM Switch using FPGA technology

Dimitrios S. Kapsalis
Institute of Computer Science (ICS)
Foundation of Research & Technology – Hellas (FORTH)
P.O. Box 1385
Heraklio, Crete, GR-711-10 GREECE
Tel: +30 81 0391 655, fax: +30 81 0391 661
kapsalis@ics.forth.gr, kapsalis@isd.gr

Technical Report 302

Abstract

Advanced Switches and routers rely mostly on Dynamic RAM technology for providing large, low-cost buffer space needed due to the burstiness of Internet traffic. Quality of Service is also desirable, therefore, per flow queueing of traffic is often implemented.

We designed and implemented a queue manager that supports per flow queueing of thousands of flows of ABR traffic for an ATM Switch. A large FPGA chip was used for fast development and extensive on-board testing. To avoid SRAM chip usage and lower the pin and trace count, a single SDRAM DIMM is used for storing both cells and pointers. We implemented dynamic memory allocation.

Buffer preallocation and free list bypassing were used to reduce memory accesses and increase buffer bandwidth. These techniques proved essential for satisfying the switch buffer requirements. ATM Flow Control features such as EFCI and RM Relative Rate Marking has been provided for each supported flow. We used synthesizable Verilog for simulation and of the architecture instead of ALTERA AHDL, so as to achieve cross-platform compatibility.

The ALTERA MaxPlus II tool has been used for synthesis and FPGA programming. We achieved a clock frequency of 35 MHz; this translates to a peak of 800 Mbps of combined incoming and outgoing throughput for the Queue Manager; the queue manager occupies 2500 FPGA Logic Elements and 2000 SRAM bits for 64K flows.

Table of contents

ABSTRACT 1

TABLE OF CONTENTS 3

LIST OF FIGURES 5

LIST OF TABLES 6

1 INTRODUCTION 7

1.1 Motivation 7

1.2 This thesis and the DIPOLO Switch 8

1.3 Switch/Router Generations and Queueing Architectures 9
1.3.1 First Generation Switches/Routers 9
1.3.2 Second Generation Switches/Routers 10
1.3.3 Third Generation of Switches/Routers 11

1.4 Queueing Architectures in general 12
1.4.1 Output Queueing 12
1.4.2 Input queueing 13
1.4.3 Variations 14
1.4.4 Per-Flow Queueing Vs Single FIFO Queueing in Output Queueing 15

2 THE DIPOLO ATM SWITCH 17

2.1 The DIPOLO Architecture 17

2.2 The ATM 155Mbps Card 18
2.2.1 The switching device (Transwitch Cubit Pro) [18] 18
2.2.2 The Cell Processing device (Motorola MC92501 Cell Processor) [17] 19
2.2.3 The local CPU (MPC860 by Motorola) [17] 19
2.2.4 The Physical level device (PM5350 S/UNI-155-ULTRA) [20] 19

2.3 The CPU Card 19
2.3.1 Motorola MPC869SAR (PowerQUICC) 20
2.3.2 The CubitPro Device 21
2.3.3 Memories 21

2.4 The ATM Line Card 21
2.4.1 Framer part - PMC-Sierra PM7344 [20] 22
2.4.2 Line interface circuit part - PMC-Sierra PM4314 [20] 22

2.5 The ABR Server Card 22
2.5.1 The Switching device (Transwitch Cubit Pro) 23
2.5.2 The ABR Server Unit (ABRSU) (EPF10K200EBC600-1 FPGA by Altera) [16] 24
2.5.3 The Memory module (256 MB of SDRAM) [19] 24
2.5.4 The local processor (MPC860 by Motorola) 24

3 THE ABR SERVER UNIT (ABRSU) 25

3.1 The ABR Server Architecture 25

3.2 The CPU Interface 26
3.2.1 MPC 860 Interface Block - Queue Manager Block Interface 27
3.2.2 MPC 860 Interface Block - Cell Scheduler Block interconnection 28

3.3 The Cell Demultiplexor 28

3.4 The UTOPIA Interfaces 29
3.4.1 UTOPIA Input Interface 30
3.4.2 UTOPIA Output Interface 31
3.4.3 The Pulse Synchronizer 32

3.5 The Queue Manager 32

3.6 The Cell Scheduler 32

4 THE QUEUE MANAGER IP 34

4.1 The Queue Manager IP Architecture 35

4.2 Functional Implementation 36
4.2.1 Interfaces 36
4.2.1.1 The Interface with the CPU 37
4.2.1.2 The Cell Demux Interface (Incoming cells) 37
4.2.1.3 The Cell Scheduler – Queue Manager Interface 38
4.2.1.4 The SDRAM – Queue Manager Interface 39
4.2.2 Flows 40
4.2.3 Flow Groups 42
4.2.4 Queue Manager Commands 42
4.2.5 EFCI and RM marking [21] 44

4.3 Design Implementation 45
4.3.1 Flow Record Format 45
4.3.2 Flow Group Record Format 46
4.3.3 Cell Format and Alignment 47
4.3.4 SDRAM Memory Organization 48
4.3.5 State Machines 49
4.3.6 The SDRAM Controller 51
4.3.7 Free List Bypassing [13] 52
4.3.8 Cell Buffer Pre-allocation [10] 54

4.4 Timing Issues 54
4.4.1 UTOPIA clock Vs Queue Manager (ABRSU) clock. 54
4.4.2 Worst case of Enqueue and Dequeue 55
4.4.3 Normal case of Enqueue and Dequeue 56
4.4.4 Synthesis Results, Free-list bypassing and Cell Buffer Pre-allocation contribution 58

5 CONCLUSIONS AND FUTURE WORK 59

6 REFERENCES 60

List of Figures

Figure 1-1: First Generation Switch Routers 10
Figure 1-2: Second Generation Switch/Router 11
Figure 1-3: Left: Third generation Switch/Router, Top-Right: A crossbar, Bottom-Right: An 8x8 Banyan Fabric made of small 2x2 Switch blocks. 12
Figure 1-4: Left: Output Queueing with a Switching Fabric and multiple buffers Right: Input Queueing with a Switching Fabric. 13
Figure 1-5: Left : Head of Line Blocking Right: Advanced Input Queueing 14
Figure 1-6: Left: Internal Speed Up Switch, Right: Crosspoint Queueing Switch 14
Figure 1-7: Single FIFO queueing and two threshold congestion detection approach 15
Figure 1-8: Per-Flow queueing and two-threshold detection approach 16
Figure 2-1: General Architecture of DIPOLO ATM Switch 17
Figure 2-2: The ATM 155 Card block diagram 18
Figure 2-3: The CPU Card block diagram 20
Figure 2-4: The VDSL Line Card Block Diagram 21
Figure 2-5: ABR Server Card block diagram 23
Figure 3-1: The ABRSU internal block diagram 26
Figure 3-2: The CPU Interface sub-block diagram 27
Figure 3-3: The Cell Demultiplexor sub-block diagram 29
Figure 3-4: UTOPIA Input Interface block diagram 30
Figure 3-5: UTOPIA Output Interface sub-block diagram 31
Figure 3-6: The Pulse Synchronizer 32
Figure 4-1: The Queue Manager IP sub-block diagram 35
Figure 4-2: The Flow Structure and changes on it after an enqueue and a dequeue operation. 41
Figure 4-3: The 64 Flow Group cyclic lists 42
Figure 4-4: Flow Record Fields and alignment 45
Figure 4-5: Flow Group memory organization and Flow Group records. 47
Figure 4-6: Cell Format and alignment 48
Figure 4-7: SDRAM Memory-space division and organization 48
Figure 4-8: State Machine Top Level Diagram 49
Figure 4-9: Enqueue command FS M bubble diagram 51
Figure 4-10 : State Machine Diagram of the SDRAM controller 52
Figure 4-11: Free List Bypassing implementation in the Queue Manager IP 53
Figure 4-12: Worst case Enqueue-Dequeue timing Diagram 56
Figure 4-13: Normal case Enqueue-Dequeue timing diagram 57

List of Tables

Table 4-1: The Interface with the CPU.....37
Table 4-2 : The Cell Demux Interface (Incoming cells).....37
Table 4-3: The Cell Scheduler – Queue Manager Interface 38
Table 4-4: The SDRAM – Queue Manager Interface.....40
Table 4-5: Table of all the Queue Manager commands 43
Table 4-6 : Flow Record Field bits and description.....45
Table 4-7: Flow Record Field description.....47
Table 4-8: Queue Manager Command Priorities 50

1 Introduction

1.1 Motivation

The introduction of bandwidth hungry applications both in corporate and client communications is one of the most consistent trends in the networking world. Multimedia applications that ride the Moore's Law on one hand, produce excessive amounts of voice and video data to be relayed through the network, while booming business to business networking contributes small but frequent chunks of data between corporate headquarters. These network requirements are heavy burden to the current network infrastructure that relies mostly on IP protocol and wire cables. While the second is the subject of the last mile problem and requires gradual but enormous investment of funds in a global scale with the introduction of fiber optics, the first will still pose problems in efficient networking because IP is incapable of differentiating among diverse services required by network users. ATM protocol should gradually replace or merge with IP applications along the way to high speed, fiber optic networking.

ATM offers generic Quality of Service (QoS) guaranties to existing networking by differentiating network traffic into several types depending on the requirements of the application that produces it and provides special handling and billing for each. These requirements are:

- Bandwidth – The rate at which, the network must carry an application's traffic.
- Latency – The delay that the application can withstand in the delivery of its data
- Jitter – The variation in latency
- Loss – The percentage of acceptable loss of data.

Each of the family of traffic types supported by ATM networks places most of its interest over some of these requirements and less to the remaining. These types are:

- Constant bit rate (CBR) - This type is used for emulating circuit switching. The bandwidth is constant with time. CBR applications are quite sensitive to jitter but not no much to data loss. Examples of applications that can use CBR are telephone traffic, videoconferencing, and television.
- Variable bit rate–non-real time (VBR–NRT) - This type allows users to send traffic at a rate that varies with time depending on the availability of user information. Statistical multiplexing is provided to make optimum use of network resources. Multimedia e-mail is an example of VBR–NRT.
- Variable bit rate–real time (VBR–RT) - This type is similar to VBR–NRT but is designed for applications that are sensitive to cell-delay variation. Examples for real-time VBR are voice with speech activity detection (SAD) and interactive compressed video.
- Available bit rate (ABR) - This type of ATM services provides rate-based flow control and is aimed at data traffic such as file transfer and e-mail. Although the standard does not require the cell transfer delay and cell-loss ratio to be guaranteed or minimized, it is desirable for switches to minimize delay and loss as much as possible. Depending upon the state of congestion in the network, the source is required to control its rate. The users are allowed to declare a minimum cell rate, which is guaranteed to the connection by the network.

- Unspecified bit rate (UBR) - This type is the catch-all-other class and is widely used today for TCP/IP.

The last two types of traffic are the most opportunistic of them all since they tend to use the leftovers of the network resources reserved by the others whenever available, thus increasing network efficiency. Still they carry the burden of the most traditional of network applications such as E-mail, FTP, HTML that is the base of contemporary networking. Although they pose some bandwidth requirements, they step aside in times of network congestion. This is translated to extensive buffering (or queueing) of these types of traffic at the network nodes they reside when the latter suffers from congestion.

There are a number of architectures proposed throughout the evolution of switches and routers that address the problem of effective traffic queueing. Large buffers alone can not support the variety of traffic types and the problems of delays and HoL (head of line) blocking that arise. It seems that providing separate queueing for each flow of traffic (per flow queueing) gives the ability to the switch/router to service more accurately the requirements of each one **[5], [10], [11], [14]**, while the provision of resizable buffer space (dynamic queueing) for each flow takes advantage of unused, by dormant flows, buffer space **[5]**. Switch/router architectures with these characteristics can accommodate high speed network traffic while using off-the-shelf, inexpensive memory modules of Dynamic RAM dropping the memory cost to the minimum of a standard PC.

Dynamic RAM memory modules in all their forms have a wide range of usability in contemporary computing and especially SDRAM which is used almost in any computing device that needs large cheap buffer space, with moderate to high throughput. Best performance is achieved when large chunks of data are moved, making these memories suitable for networking that is now well in its packet (~64 bytes) switching era. Their wide applicability, simple interface, and industry-wide standardization drops their cost substantially, fuelling the addition of more ports to the networking device **[1,ch9], [12]**.

1.2 This thesis and the DIPOLO Switch

In this thesis, we describe the architecture of a Queue Manager IP that supports the features of per flow, dynamic queueing of ABR traffic type of ATM networks. This IP was designed for the purposes of a 1Gps ATM Switch called Dipolo.

It is used to accommodate the queueing of a maximum of 64K flows for a centralized ABR Server Card. The Card uses a large FPGA to host the IP and a single SDRAM DIMM module to store cells and queue pointers. There is also a CPU interface that programs the Queue Manager with parameters of each ABR flow so that the IP can support ABR flow control features such as RM marking and EFCI. We also discuss the general architecture of the DIPOLO Switch and the features of the IP architecture that were used to increase Memory utilization, such as free buffer preallocation **[10]** and free list bypassing **[13]**. The IP can also support traffic differentiation, by organizing flows into Flow groups according to service needs and output port, making it flexible even to support CBR or VBR traffic when special care is taken by the scheduling hardware. Special care was also given to the interfaces of the IP with other IPs such as the Scheduler and the CPU so that they are simple and

effective. The single SDRAM interface did not allow for any parallel accesses. Instead Enqueue and Dequeue commands where implemented as a whole instead of smaller atomic commands to increase Memory utilization. A clock speed of 35 MHz was achieved for the FPGA implementation that is translated to a maximum 800 Mbps of combined incoming and outgoing throughput. In ASIC implementations where clock speeds of 133 MHz are feasible this throughput could rise substantially making this architecture suitable for use as part of a Networking Chip **[7], [10]**. A total of 2,5K of FPGA Logic Elements was used as well as only 2K bits of on chip (FPGA) SRAM.

DIPOLO Switch design was a joined project by the University of Crete, Greece, the National Technical University of Athens, the Foundation of Research and Technology of Crete and Intracom Company. The purpose of this project is the design and manufacturing of ATM switch of 1 Gbps of throughput, for the provision of broadband networking to domestic VDSL users. The main system requirements were, the provision of CBR, VBR, ABR types of ATM traffic. Other factors that were also taken into account were low system cost, use of commercial chips and the optimum division of work between the conglomerates.

Under the scope of DIPOLO design, the writer had the chance to work on the physical design of the ABR Server Card. This card is described in section 2.5. More precicely, he worked on the definition of its organization, its description in concept level. He also defined the internal organization of the ABRSU unit (FPGA) of the card. This unit is decribed in chapter 3. He designed the Cell Demux (see section 3.3), the Queue Manager, the main issue of this thesis that is described in chapter 4, the interface of the ABRSU with the card's cpu (see section 3.2) and the interface with the SDRAM memory unit. He was also in charge of top level synthesis and verification. He designed tests and demos that proved the feasibility of the design.

1.3 Switch/Router Generations and Queueing Architectures

In this section, a brief description of the switch/routers generations that evolved over the years and the queueing architectures that were introduced with each one of them is given.

1.3.1 First Generation Switches/Routers

The evolution of networking devices can be roughly separated into three generations mainly by the hardware used and the level of integration. The first generation of switches were devices that resembled to a general-purpose computer and consisted of line cards interconnected by a I/O bus. There was also a CPU that hosted all the routing software that decided were to forward the packet, a Main Memory, and an optional DMA module to release the CPU of the burden of moving data packets between line cards and Main Memory. A data packet would enter the device through the ingress side of the Line Card by use of an Analog to Digital Converter. Then the CPU would extract the header of the packet in order to route the packet over the Bus to the appropriate queue in the Main Memory. Later the CPU would schedule the packet to be routed over the BUS again, to the Line Card, through a Digital to Analog converter and off to its destination. Figure 1-1 depicts the configuration of first generation devices.

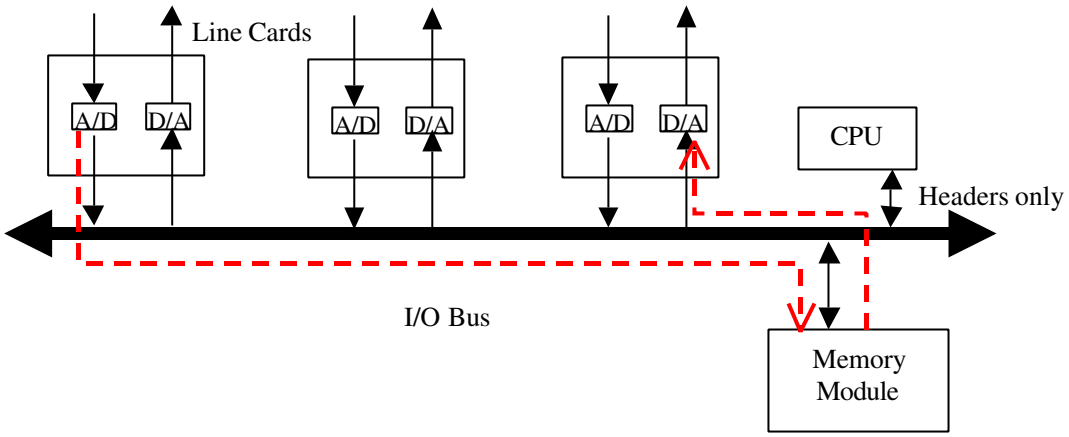

Figure 1-1: First Generation Switch Routers

There are three bottlenecks in this architecture:

- CPU Power
- Memory Throughput
- I/O Bus bandwidth

As the line speed rates increased the CPU would perform poorly, since it implemented both routing and scheduling of packets for all line cards. As for Main Memory and I/O Bus, both would scale badly with the introduction of additional cards especially for the bus that relayed each packet twice through its course through the device. Still, these devices were sufficient for low speed rates and the poor data production of applications, during the first years of the Internet era.

1.3.2 Second Generation Switches/Routers

The second generation of switches eliminated the CPU and Memory bottlenecks of the first by introducing redundancy to both of them. As seen in Figure 1-2, each line card now owns a separate memory module and a small CPU. The main memory is, now, not necessary. The local CPU implements routing and scheduling of packets, while storing takes place in the local memory. Input Queueing or Output Queueing or both can be implemented. The sole purpose of the central CPU is to arbitrate the usage of the Bus, the exchange of routing information between the local cards and the programming and maintenance of the whole system. Due to the redundancy, the only bottleneck of these devices was I/O Bus bandwidth that failed to scale along with the high speed line cards and the port count.

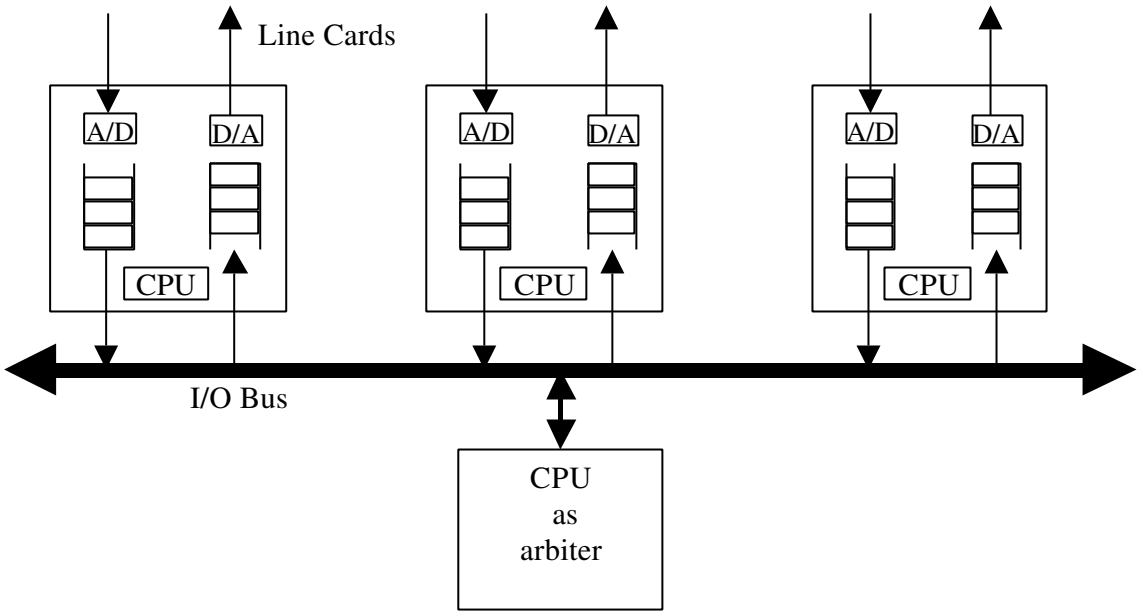

Figure 1-2: Second Generation Switch/Router

1.3.3 Third Generation of Switches/Routers

This generation introduced Switching fabrics to replace the I/O bus as the medium to relay packets between cards. Buffering and routing of data packets is performed inside the line cards while specialized hardware is provided to give to the line cards access to the fabric. Switching fabric can accept multiple simultaneous transfers of packet with a maximum of N transactions when N Line cards are connected to the fabric. The current trend for network devices, that rides the ASIC large-scale integration, is SoC (System on Chip) architectures. Except analog parts, all the line hardware (buffers, routing) for all ports are stored inside the same chip along with a crossbar (the most effective but less scalable of switching fabric), a scheduling unit and a CPU. Such chips can accommodate up to 32 input/output ports and are sufficient for a low-end switch/router. They can also be used as a building block of a large high-end switch/router. It the latter case, they are organized in Switching Fabric topologies such as Banyan, Benes, and Batcher-Banyan networks **[1,chapter 8]**. All are depicted in figure 1-3.

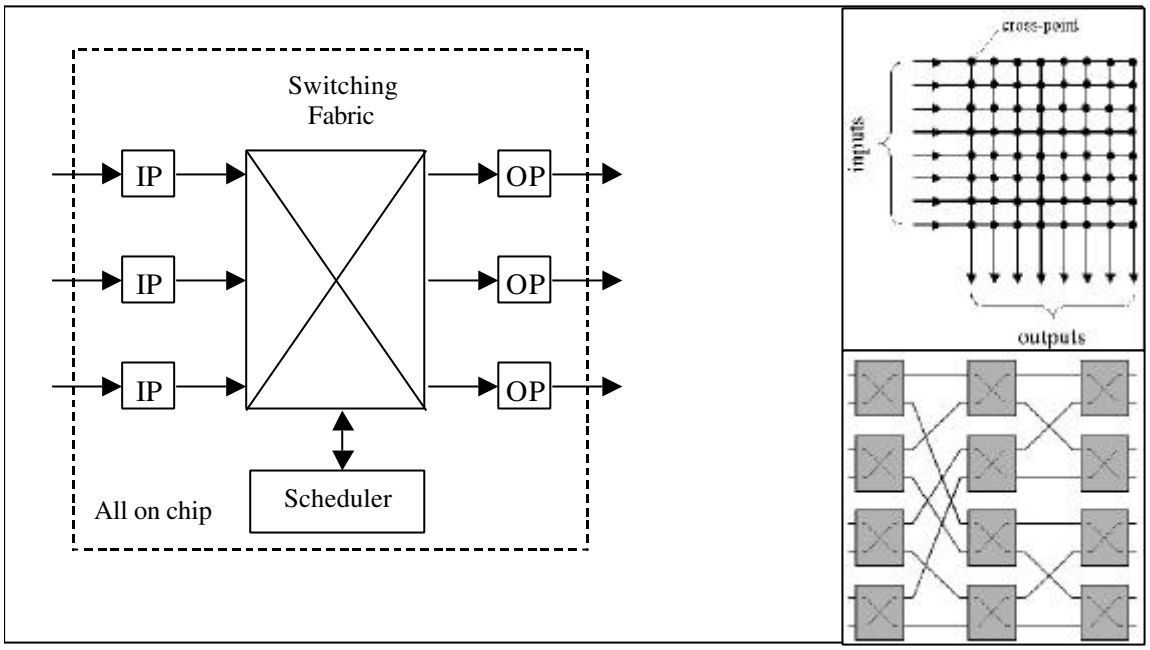

Figure 1-3: Left: Third generation Switch/Router, Top-Right: A crossbar, Bottom-Right: An 8x8 Banyan Fabric made of small 2x2 Switch blocks.

1.4 Queueing Architectures in general

In this section, we describe the main queueing architectures that are implemented in the spectrum of networking device generations described in the previous section. Advantages and disadvantages are also given. Finally per flow queueing is described.

1.4.1 Output Queueing

There are two queueing architecture families: Input queueing and Output queueing architectures. The output queueing families place the buffering memory that stores the queues near the outputs of the device as shown in figure. The interconnecting medium can be either a shared medium (like an I/O Bus) or a switching fabric (like a cross bar). One single module (a shared buffer architecture) or separate ones, one for each output port can serve all the link output queues. The first case with a shared medium is the first generation of switches/routers. Both of them can accept all the available throughput but place heavy requirements on the rate of interconnecting medium and the memories. In the case of N input ports and N output ports with a dedicated memory module for each output port the module must have a throughput of N+1, when input throughput for each port is 1. In the case of a single memory module for all output ports, the module must provide a throughput of Nx(N+1). Both configurations, shown in figure 1-4, are not scalable so output queueing is generally not used.

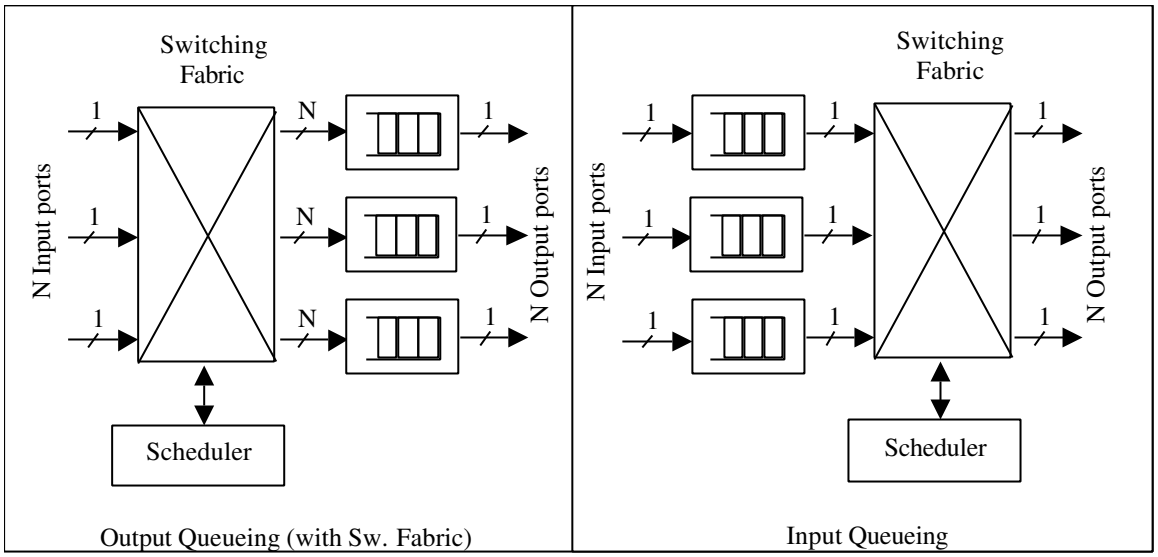

Figure 1-4: Left: Output Queueing with a Switching Fabric and multiple buffers Right: Input Queueing with a Switching Fabric.

1.4.2 Input queueing

When the buffering memory is placed at the inlet side of each port of the switch we describe this scheme as input queueing. Each time a packet arrives it is placed in a queue at this memory and when it gets to the head of the queue it waits for the scheduler to forward it to the output port of destination. The architectures that fall into input queueing are more scalable than the ones that fall into output queueing since their memories must provide twice the input port throughput. Still, they fail to accept all the input rate, due to Head of Line blocking at each input port that decreases by one third of the optimum the switch throughput. Head of Line blocking can be seen in Figure 1-5. When two inlet queues have at their head position a packet destined to the same output port, the fabric can accept only one of them. The other queue remains idle, although there are other packets behind the head that are destined to other output ports and could be served at the same time. To overcome HoL blocking an alternate scheme is often used depicted in figure 1-5. Each packet that arrives at an input port is stored in a separate queue, according to the output port of destination. This scheme is called Advanced Input Queueing or Virtual Output Queueing and theoretically can achieve 100% utilization but requires fast scheduling hardware to find the optimum schedule from inputs to outputs **[2]**.

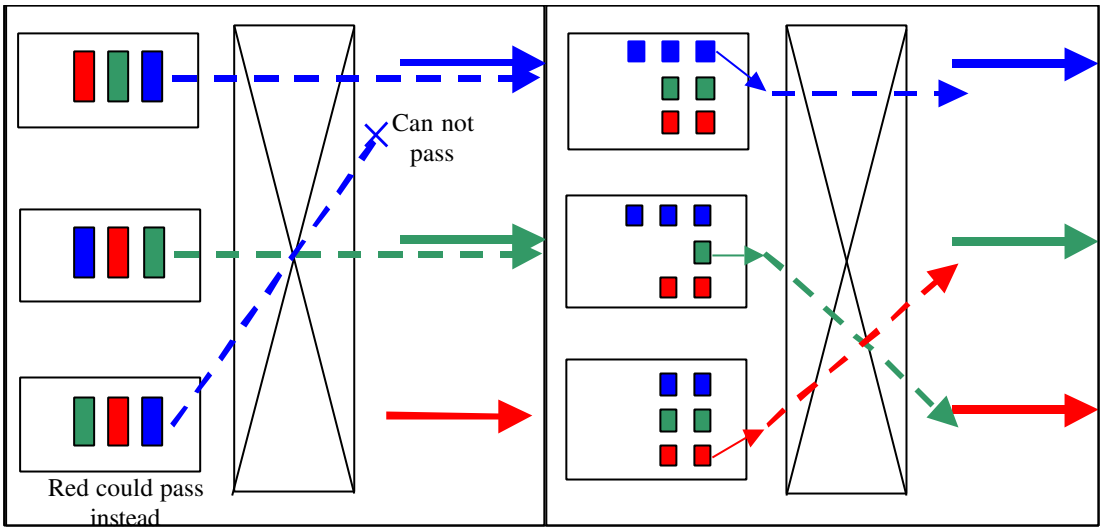

Figure 1-5: Left : Head of Line Blocking Right: Advanced Input Queueing

1.4.3 Variations

Input queueing and Output Queueing can also be used in combination for better performance. In that scheme usually some internal speed-up is used in the switching fabric **[3], [4], [5], [6]**. The fabric inputs are able to receive packets at a higher rate than the port rate. This resembles to normal Input queueing working at relatively low rates. Additionally, the fabric transmits to the output queues at a higher rate than the output line rate in order to accommodate for the accumulation of packets when most of the traffic at a given time is destined to the same output. Figure 1-6 depicts Internal Speed-Up. Another variation, shown in the figure, is Cross-point or Distributed queueing. It achieves top performance like output queueing by storing packets at the cross-points of a crossbar. This scheme though, requires extensive (NxN) usage of buffers that is costly either on chip or off-chip and is not scalable for large number of ports. Recent advances in embedded DRAM technology (DRAM memory on the same silicon chip as common logic gates) could promote such schemes.

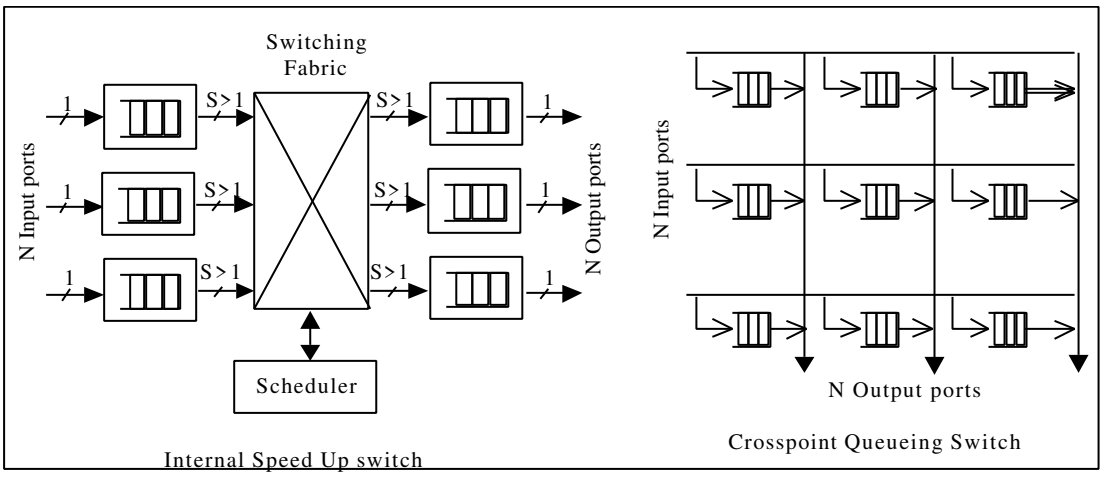

Figure 1-6: Left: Internal Speed Up Switch, Right: Crosspoint Queueing Switch

1.4.4 Per-Flow Queueing Vs Single FIFO Queueing in Output Queueing

In the theory of Queueing there is also the matter of buffer management to be addressed. The buffer size is a limited resource and is heavily contended. The heavy contention over a period of time may lead to congestion, possible loss and extensive delay of packets. Therefore in order to provide Quality of Service, a switch must take care of the buffering and scheduling techniques it uses.
In Output Queueing with a dedicated memory module at each Output, or a shared buffer, the arriving packets could be stored in a single FIFO, or in dedicated queue for each flow of Traffic. In ATM networks where a flow is a VP/VC the latter scheme is called Per-VC queueing.

In the Single queueing approach, a centralized memory is completely shared by a single queue, where all the packets from different sources or input ports enter. Then they are scheduled in a FIFO manner. This discipline is the simplest, most economical and commonly implemented queueing discipline. If the queue length exceeds the available buffer space, the incoming cells are discarded. To minimize cell loss, congestion must be detected by use of two queue thresholds. When the queue length is above the high-threshold level, a congestion indication flag is set initiating marking of packets that will eventually drop the incoming packet rate. It remains set until the queue length drops below the low-threshold level. Single queueing with two thresholds is illustrated in Figure 1-7. Single FIFO queueing, because of its simple nature is easily implemented, particularly at very high-speed ports. It does not, however, provide any local mechanisms to enforce a fair access to buffers and bandwidth, and it leaves such resources open to abuses by malicious flows. In that case, the marking mechanism described suffers in terms of fairness. The problem can be overcome by using processing power to calculate the fair share of bandwidth for each flow and communicate it to its source as in Explicit Rate technique of ABR traffic of ATM. The same external to queueing mechanism must also impose a policing function to punish the malicious sources.

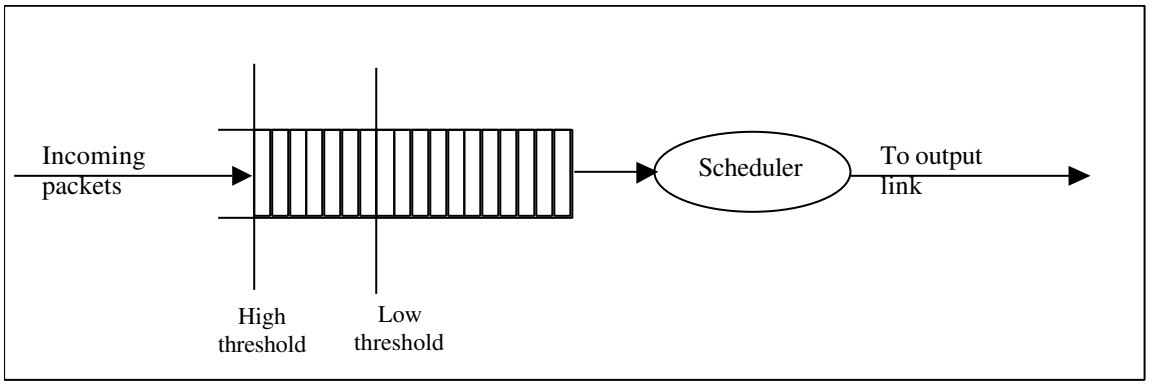

Figure 1-7: Single FIFO queueing and two threshold congestion detection approach

Unlike the first approach, where all flows are queued in one single queue, in the per-Flow approach, cells from different flows (or VCs in ATM networks) are queued in

separate queues and buffer space is allocated in a per-flow basis. Multiple classes of traffic, with different priorities and QoS requirements can be fairly served, with the per-Flow implementation in conjunction with a proper scheduling policy. The output –port buffer space could be divided among all flows in a fixed manner or dynamically shared among them. In the fixed buffer allocation method (static), each VC is only allowed to occupy its own VC buffer share, but in dynamic buffer management, VCs can take more than their share. The state of congestion is determined similarly using the two-threshold approach described earlier. Still two thresholds can be maintained for each of the flows, as seen in Figure 1-8, so that ill-behaved flows can be identified. The isolation provided by the separate queues ensures fair access to buffer space and bandwidth. This also allows the delay and loss behavior of individual flows to be isolated from each other. This per-flow information can be used for congestion control, either by marking of packets (EFCI marking of ATM cells for example), or Explicit rate mechanisms. It can also help identify and police misbehaving sources effectively. In the static buffer management variation, where flows are given a fixed buffer share, the policing can be completely eliminated. This happens because in the static buffer scheme, if a flow is misbehaving, only its queue will grow and overflow. The dynamic buffer scheme, on the other hand, must utilize some intelligent mechanism to achieve policing.

Generally per-VC queueing offers some advantages over single FIFO queueing. These become critical when QoS must be implemented, since the latter when used broadly achieves the most efficient usage of network capacity. Still, per-flow implementation suffers considerably in terms of implementation and scheduling complexity. Since per VC queueing is proportional to the number of maximum VCs that can be served, it doesn't scale well for million of VCs. In addition complex scheduling policies must be implemented on a per-flow basis. When the number of VCs is relatively low (like 64K flows in our IP) buffering complexity is low but the scheduling complexity is still high **[1, chapter 9], [8], [9], [10,] [11]**.

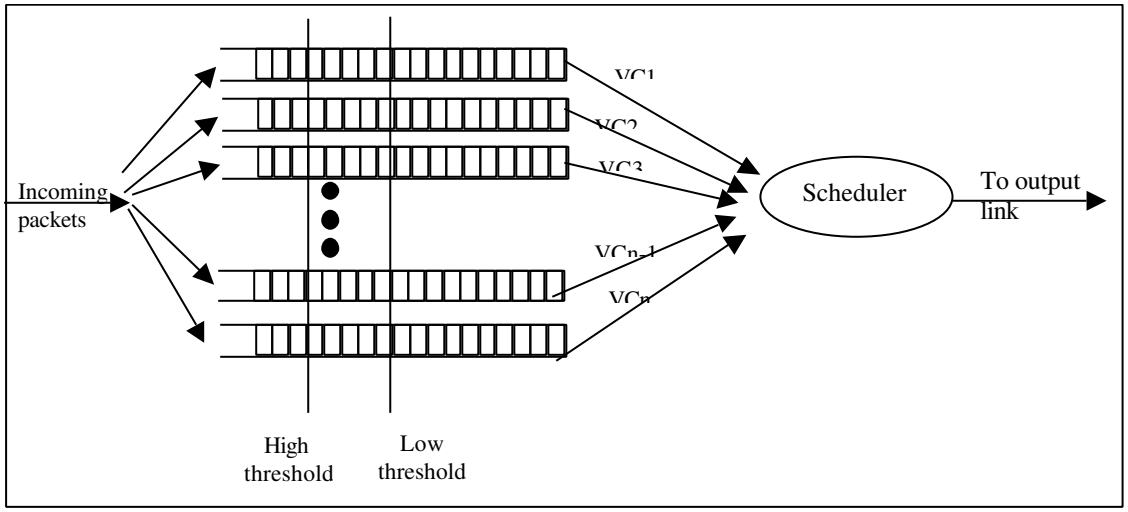

Figure 1-8: Per-Flow queueing and two-threshold detection approach

2 The DIPOLO ATM Switch

2.1 The DIPOLO Architecture

The general architecture of the DIPOLO ATM Switch is depicted in figure 2-1. Based on a Second Generation architecture, this system is made of a number of VDSL Line Cards that connect the End Users with the Switch, one or more ATM 155 cards for the interconnection of the Switch with the central ATM Network through OC-3/STM-1 lines, a CPU Card, an ABR Server Card and a shared Cell Bus.
The exact functionality of each card is given below:

- Shared Cell Bus: At system level, the exchange of data and management and signaling information is done by ATM connections. The transmission of such data is done through the use of a shared Cell Bus (CellBus). For redundancy the system uses two such busses. One main and one redundant that it is used in case of failure by the main one
- ATM 155 Card: It executes all the functionality of the physical level of ATM, like physical level ending, ATM routing, signaling Q.2931, OAM function etch. The connection of many cards over a backplane incorporates the switch.
- ABR Server Card: It executes all the necessary functionality for the acceptance, storage and forwarding of cells that carry ABR traffic. The ATM 155 cards can not handle such traffic because it requires extensive buffering. The core of the functionality is the maintenance of a large number of multiple level queues.
- CPU Card: It provides top system maintenance and performs functions like Call Admission Control (CAC).
- VDSL Line Card: It implements the interconnection of end users with VDSL modems through wire cables.

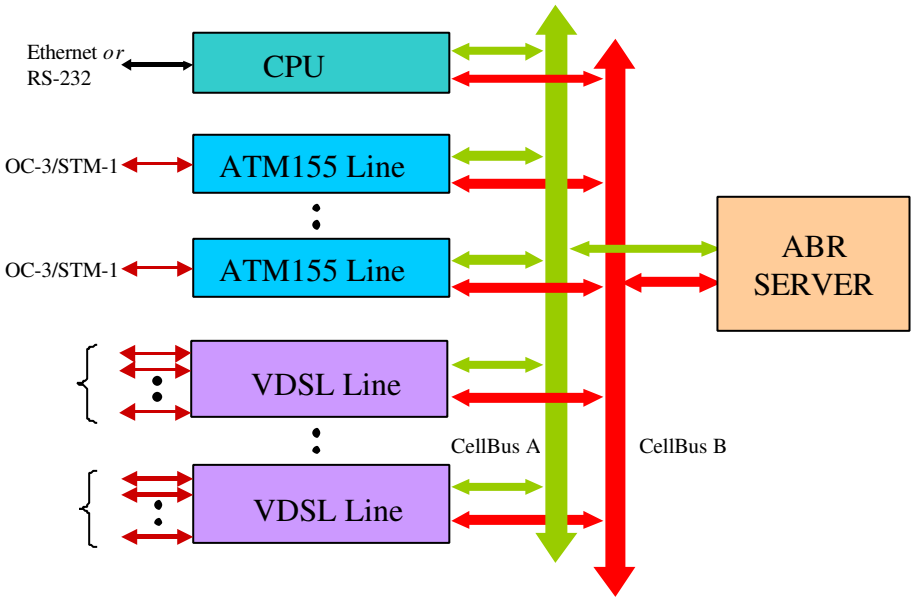

Figure 2-1: General Architecture of DIPOLO ATM Switch

With this organization, the centralized approach of handling the ABR Traffic (by the centralized ABR Server Card), allows the concentration of temporary memory in one card, increases memory utilization and drops the system cost. It also allows the creation of systems with or without ABR support and increases the flexibility of the system in terms of needs and development by the contractors.

2.2 The ATM 155Mbps Card

The ATM 155 Card's architecture is composed by 4 subsystems shown in figure 2-2

- The switching device
- The Cell processing device
- The local control processor
- The local physical level device

In the next subsections we give a description of these subsystems and the commercial products that were used for their implementation.

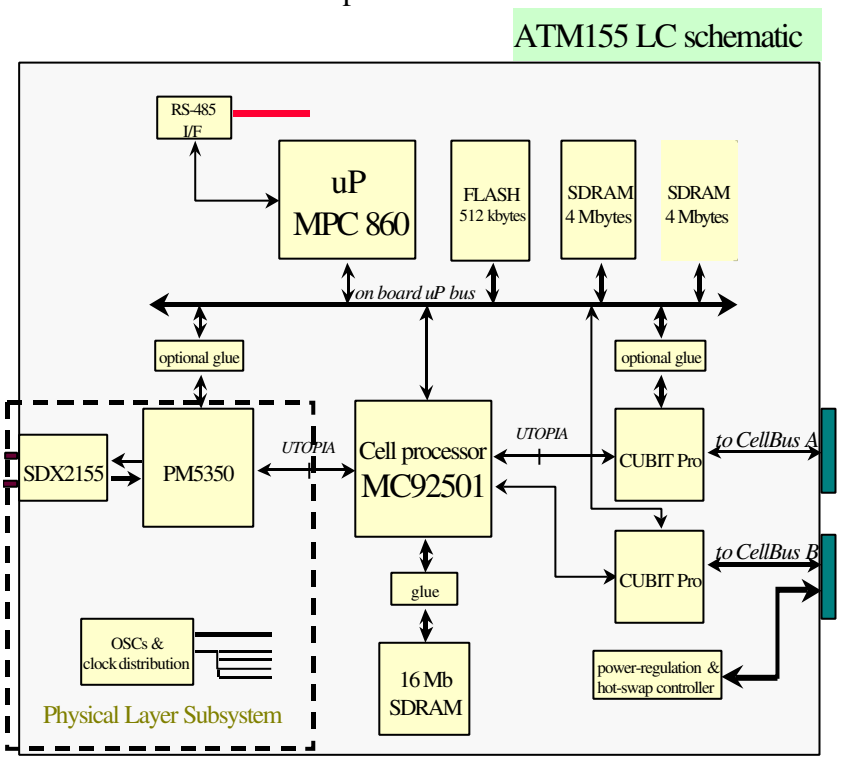

Figure 2-2: The ATM 155 Card block diagram

2.2.1 The switching device (Transwitch Cubit Pro) [18]

The switching device is connected with the cell-processing device through a UTOPIA interface and it comprises the interconnection of the card with the Cell Bus. The latter is located on a separate backplane box that holds together all the cards.

2.2.2 The Cell Processing device (Motorola MC92501 Cell Processor) [17]

For the implementation of the cell-processing device we use the MC92501 Cell Processor by Motorola, that is a well-known and standard processor. It is connected with the physical device and the CubitPro device through a UTOPIA interface. It also connects with the local processor by use of a special processor interface.
The Cell Processor executes all the OAM functions in hardware, and inserts in the system cells that carry signaling and managing information through the local CPU interface. It also executes address translation (VPI/VCI) translation and inserts in the incoming cells routing headers that notify the CubitPro with the card of destination of the cell. The address translation table is located in an external dedicated SRAM that is updated by the local CPU. The MC92501 also performs per flow policing (UPC/NPC) by using a leaky bucket algorithm.

2.2.3 The local CPU (MPC860 by Motorola) [17]

For the local control of the ATM 155 Card, the MPC860 processor by Motorola was used. This processor is responsible for the initialization and normal function of the card. It interfaces with all the devices on the card. The function of the CPU is supported by a DRAM that is used for storing data and an EPROM for storing code.
By interfacing with the other devices the MPC860 can extract information relative to their functionality. It also polls them to get a view of the ATM traffic that passes through the card and collects statistic for system management. It performs signaling while detecting and correcting faults. Special ATM cells that are transmitted over the CellBus through dedicated VP/VC connections do the communication between this CPU and the central CPU card of the switch. Finally the MPC860 uses an RS485 interface for external communication and supervision.

2.2.4 The Physical level device (PM5350 S/UNI-155-ULTRA) [20]

For the STM-1 physical connection the PM5350 S/UNI-155-ULTRA device by PMC Sierra was used. This is a device that interfaces with the MC92501 through a UTOPIA interface and implements the ATM transmission convergence for the SONET/SDH 155.52 Mbit/sec. The PM5350 is used for the implementation of basic functions of the physical level for an ATM UNI (User Network Interface) interface. In a typical STM-1 application, the device performs clock and data recovery at the reception side and clock generation at the transmission side. For the optical transmission of data an SDX2155 simple rate optical transceiver is used for SDH STM-1. It can support a connection length longer that 15 km and is manufactured with 155 Mbit/s standards set by the ATM Forum. Finally the SDX2155 has as PECL interface for communication with the PM5350 device.

2.3 The CPU Card

The basic functions performed by the CPU Card are:

- Central administration and monitoring of the DIPOLO Switch
- Call Admission Control (CAC)
- Interface with external management systems (support for SNMP and/or CMIP)

In figure 2-3, the block diagram of the CPU Card is given.
The basic parts of the Card are described in the following subsections.

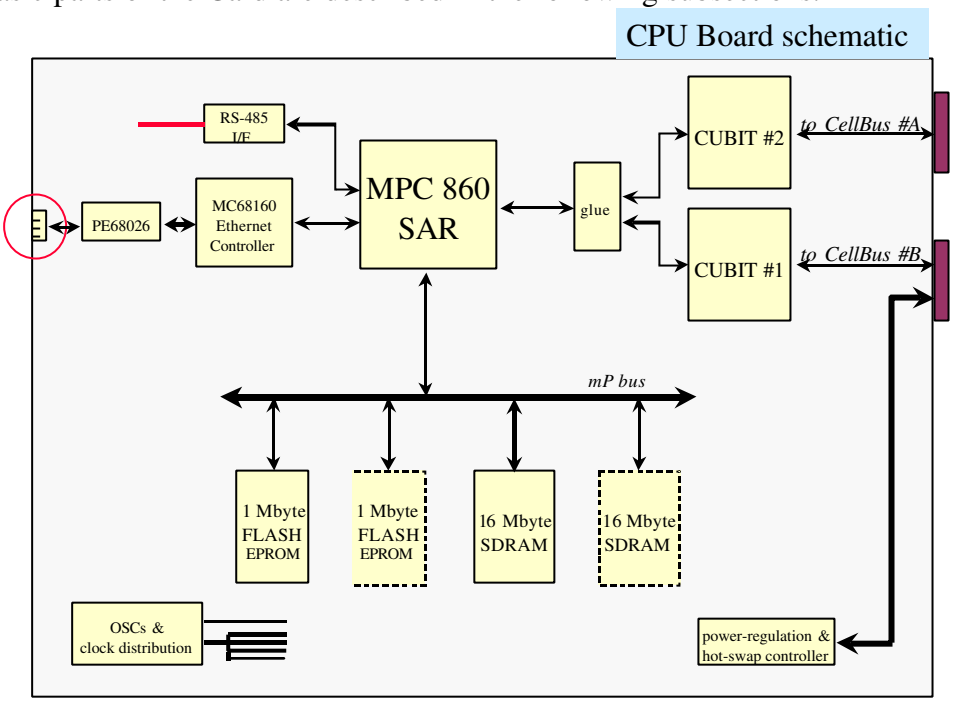

Figure 2-3: The CPU Card block diagram

2.3.1 Motorola MPC869SAR (PowerQUICC)

The MPC860SAR is advanced version of the Motorola's MPC860 processor, based on the PowerPC architecture. It executes additional ATM and SAR (Segmentation and Re-assembly) functions. One of its basic advances, in terms of system design, is that it provides interfaces to memories, serial transceivers and bus buffers. The MPC860SAR provides the following interfaces:

- Interface with SRAM, DRAM, EPROM, Flash and other peripherals.
- Seven serial interfaces controlled by an integrated communication subsystem processor that can support a number of different communication protocols.
- Half-duplex UTOPIA interface at 155 Mbps that can support multi-PHY function and extended ATM cell size.

The processor performance reaches 52 MIPS at 40 MHz, with cell processing (incoming and outgoing) rate of over 60Mbps.

2.3.2 The CubitPro Device

Like on the ATM 155 Card, the Cubit Pros provide the interface of the CPU Card with the double Cellbus, allowing the communication of the CPU (MPC860SAR) of the CPU Card with the local CPUs of the other cards, with the use of Control Cells.

2.3.3 Memories

Memories used are:

- SDRAM: 16Mbytes for the storage of code and data
- ROM: 2MB
- Flash EPROM for the storage of code for the Central and the local CPUs.

2.4 The ATM Line Card

The E1 Physical Layer part terminates 16 E1 cell-based lines. All terminated data are interchanged with the MC92501 cell-processor over a UTOPIA II bus. As shown in figure 2-4, the E1 physical layer sub-system consists of the following components:

- 4 PM7344 QUAD E1/T1 Framers
- 4 PM4314 QUAD E1/T1 Line interface components

In figure 2-4, the block diagram of the VDSL Line Card is given.

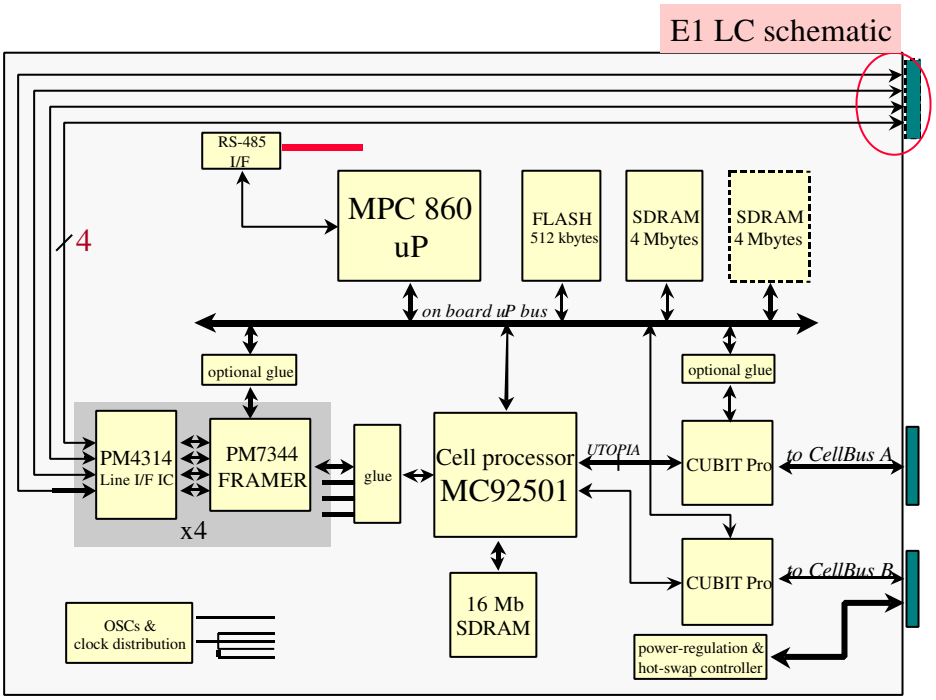

Figure 2-4: The VDSL Line Card Block Diagram

The incorporated parts are described below.

2.4.1 Framer part - PMC-Sierra PM7344 [20]

The PMC-Sierra PM7344 is a 4 port (input and output) Framer device. It accepts raw line data (RZ encoded) from the Line interface circuit (PMC-Sierra PM4314), and provides cells to the ATM Board through a UTOPIA 2 like interfacing (SCI-PHY). Some glue logic (FPGA) is required to convert the SCI-PHY signals to pure UTOPIA 2 signals. The PM7344 provides cell scrambling (from UTOPIA to PM4314), de-scrambling (from PM4314 to UTOPIA), and cell delineation (according to G.804). It inserts/extracts idle-unassigned cells to/from the data stream. Rate de-coupling of the UTOPIA interface is achieved by a 4-cell input FIFO and a 4-cell output FIFO. The PM7344 is connected to the Motorola MPC860 with an 8-bit CPU interface and the appropriate glue logic. The MPC860 assign values to the internal registers of the PM7344 in order to program it, and gets information about the line status by reading the appropriate registers. The PM7344 can also generate interrupts to the MPC860 under certain line conditions.

2.4.2 Line interface circuit part - PMC-Sierra PM4314 [20]

The PMC-Sierra PM4314 integrates 4 duplex E1 (G.703) compatible line interface circuits. It accepts E1 (G.703) bipolar line signal from the electrical components, and provides digital (RZ encoded) pulses to the Framer part (PM7344). The PM4314 provides clock recovery and performance monitoring in the receiver. It is also equipped with a generic microprocessor interface (connected to the MPC860 with the appropriate glue logic) for initial configuration, ongoing control and status monitoring. The microprocessor interface utilizes an 8-bit data, and a separate 8-bit address bus, and is able to generate interrupts upon detection of various alarms, events, or changes in status.

2.5 The ABR Server Card

The ABR ATM Traffic covers the needs of traditional LAN networks. Computational systems on a LAN want to send data, the minute these are generated, at the highest possible rate, but without congestion that causes cell loss. The reason behind this is that computer data are sensitive to loss (as opposed to multimedia data like voice and video where loss can be tolerated to a point), and retransmissions can seriously degrade the network performance.
Instead of reserving network resources for the bursty traffic of LANs, it can be served by the ABR service that uses the available bandwidth, left unused by other sensitive to delay and jitter, traffic services of ATM networks, like CBR and VBR. In order though to allow for end-users to send whatever they want whenever they want without loss, switches with extensive buffering capabilities should do the service of ABR traffic. In that way, cells that can not be transmitted during time of congestion can be stored for some time.
For this reasoning, the DIPOLO Switch utilizes a centralized ABR Server card, in order to efficiently utilize the memory resources available at the lowest of costs. The

ABR Server Card can provide for the buffering of all the ABR traffic of the Switch. In that way, supplying memories to all the card thus increasing complexity and cost, is avoided. Memory utilization is also the highest. The downside is that some CellBus bandwidth is lost since each ABR traffic cell traverses the bus twice (once from card of entry to ABR Server card and once from ABR Server card to card of exit) and that one ABR Server card is a single point of entry. Since though the switch is designed for low-end home application, where multimedia data are the core rather than computer data, and cost matters for a the highest possible market penetration, centralized buffering of ABR Traffic is justified.
The architecture of the ABR Server Card is given in figure 2-5. The card is made out of the following devices:

- The Switching device (Transwitch Cubit Pro)
- The ABR Server Unit (ABRSU) (EPF10K200EBC600-1 by Altera)
- The Memory module (256 MB of SDRAM)
- The local processor (MPC860 by Motorola)

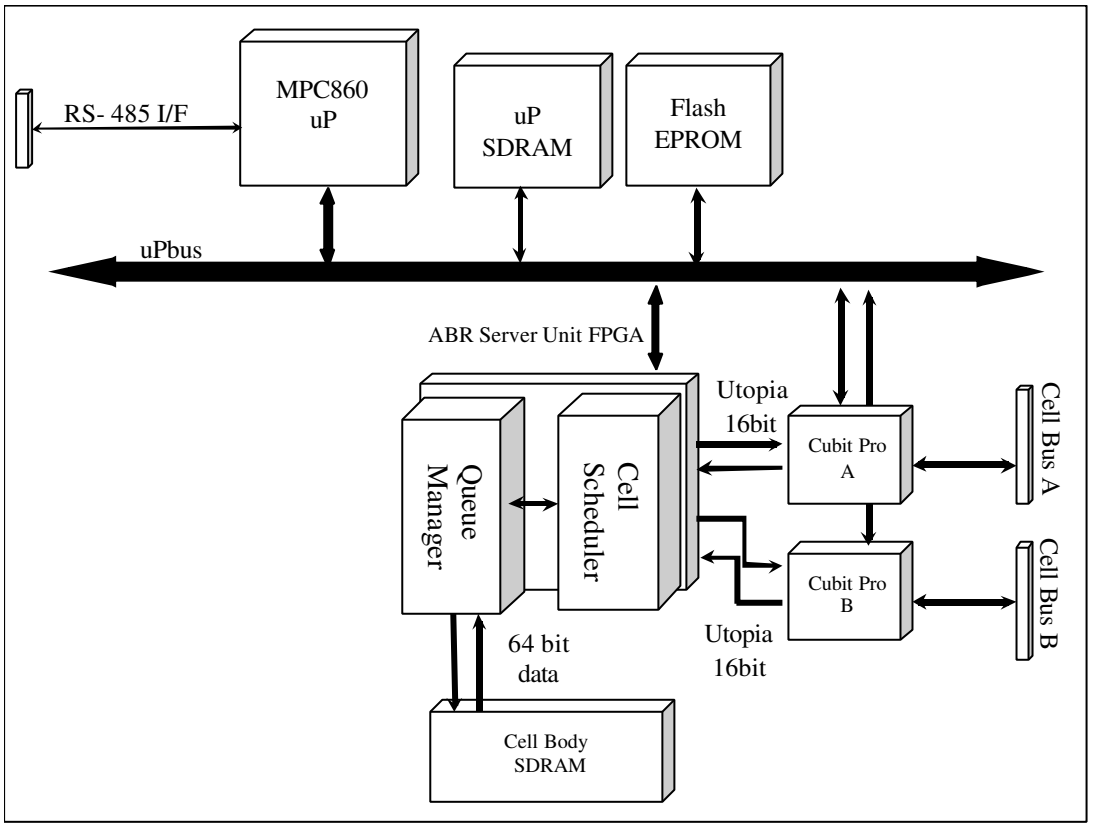

Figure 2-5: ABR Server Card block diagram

2.5.1 The Switching device (Transwitch Cubit Pro)

See Section 2.2.1.

2.5.2 The ABR Server Unit (ABRSU) (EPF10K200EBC600-1 FPGA by Altera) [16]

The ABR Server unit (ABRSU) is composed by two main blocks, one for buffering incoming cells, the Queue Manager and one for scheduling their departure, the Cell Scheduler. The ABRSU interconnects with the CubitPro Switching devices through a two way UTOPIA interface 16 bit wide. It also has a CPU Interface to communicate with the MPC860 local processor. Finally it has an SDRAM controller to access the buffer space of the SDRAM memory module that stores the Cell Queues and the queuing data (head tail pointers for the queues and other).

The purpose of the ABRSU is to store ABR cells according to the 16bit field that is in the header of every cell and defines uniquely the Virtual flow (VP/VC) that it belongs. The ABRSU must also schedule the transmission of each cell. The whole logic of the ABRSU is implemented by an EPF10K200EBC600-1 FPGA by Altera. The ABRSU can support up to 64K flows of ABR traffic, since the routing ID is 16 bit wide.

2.5.3 The Memory module (256 MB of SDRAM) [19]

A 256MB DIMM of SDRAM was used as the dedicated buffer of the ABRSU, where cells where stored in queues as well as other queue data (head and tail pointers for the queues) and per flow information. Although less than 256 bytes could suffice for the application, 256Mbytes of SDRAM are common DIMMs on the market.

2.5.4 The local processor (MPC860 by Motorola)

See Section 2.2.3

3 The ABR Server Unit (ABRSU)

In this chapter we give a presentation of the ABRSU that includes the subject of this thesis, the Queue Manager. It also includes additional blocks that interface with the Queue Manager in order for the latter to implement the Queueing function for the ABR Server card of the DIPOLO Switch. The most important is scheduling block, the Cell Scheduler that is subject of this thesis.

3.1 The ABR Server Architecture

The ABR Server Unit is implemented on the ABR Server Card by Altera's EPF 10K200EBC600-1 FPGA. It provides the following functions to the Card:

- Acceptance of Incoming ABR traffic cells through a dedicated UTOPIA input interface with the CubitPro devices of the card.
- Recognition of RM cells so that RM marking can be performed on them in case they do not conform to their qyality of service.
- Storage of ABR,RM cells in the dediceted cell memory (SDRAM DIMM) based on the flow they belong. (per flow queeueing). Tail pointers of the queues are also enumerated with the new arrivals.
- Grouping of the flows into group of flows (Flow Groups). The grouping is free of any constraint and can be used to group flows either by the Dipolo Switch port/card of destination or the Quality of Service they have assigned with or both.
- Scheduling of the transmitions of stored cells that lay on the head of the flow queues from the ABR Server Card. The transmition is based on the available bandwidth of the CellBus. For this function, a Scheduling block has been implemented that uses the information of the quality of service parameters for each flow group to request the dequeing of one cell from a group of flows to be transmited. That cell will then be dequeued from the SDRAM by the Queue Manager block that will enumerate the head pointer of that flows queue and send the cell for transmition. When the correct transmition to the card of destination is acknoeledged then the memory block that the cell was kept will enter a free list and will eventually be used by another incoming cell **[23]**.
- Transmition of the dequeued cells to the CubitPro devices of the card through a dedicated UTOPIA output interface. There is also the capability of retransmition of the cell in case the first transmition over the CellBus by the Cubit fails.
- Provision of a CPU Interface block that is used for interconnection of the ABRSU logic with the MPC860 local processor. Through this interface the MPC860 can initialize or terminate the ABR flows that are served by the card. It can also have access to the data structures of queues and the queues themselves that are stored in the SDRAM. The MPC 860 can also access the Flow Control parameters of each Queue dynamicaly by providing the thresholds that are used for marking of flows individually. It also can access the service times used by the Scheduling device to request the service of Flow Groups.

In figure 3-1 the block diagram of the ABRSU is given:

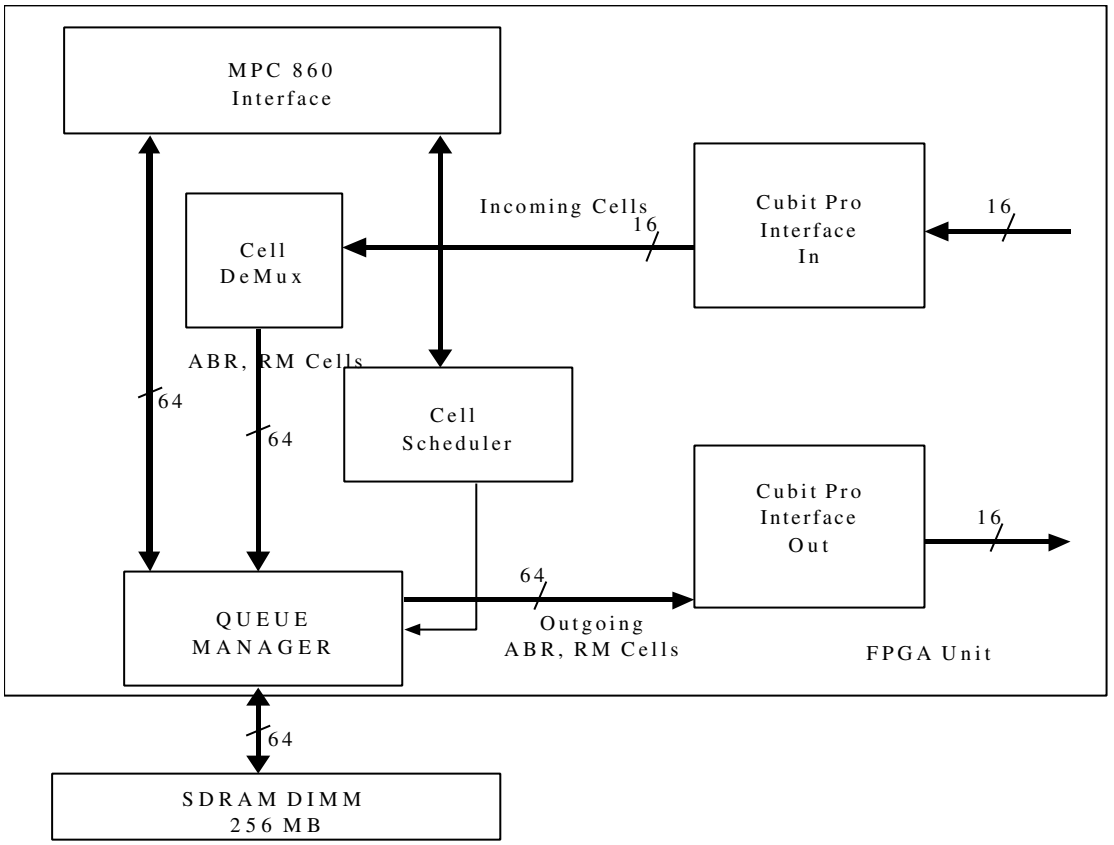

Figure 3-1: The ABRSU internal block diagram

The internal sub-blocks of the ABRSU are presented in the following chapters, while a separate chapter is dedicated for the Queue Manager IP that is the subject of this thesis (See Chapter 4).

3.2 The CPU Interface

The CPU Interface subblock, as mentioned previously, is responsible for the communication, initialization, and dynamic configuration of the rest subblocks of the ABRSU. Through this subblock the CPU initializes the Flow Data Structures stored in SDRAM by issuing special commands executed by the Queue Manager sublock. If needed the Queue Manager will reteurn data to the CPU through this sub-block.. The CPU can also dynamically configure QoS service parameters at the Cell Scheduler **[22],[23]**.

The internal organization of MPC 860 Interface Block is given at Figure 3-2. Since the MPC860 bus on the ABR Server Card works at clock speeds ranging from 10 to 25 MHz while the FPGA can achieve a higher rate internal clock, the MPC860 Interface subblock must resign on both these clock domains and use some kind of synchronization.

From the CPU side the MPC Inreface Subblock uses the CPU bus pins as input/output. These bits is a 32 bit wide data bus, plus some control signals (chip select,Read/Write etc).Through these pins the CPU writes at special configuration registers and configures the Queue Manager and the Cell Scheduler. ? ?s? a?t?? t?? pins ? ? ????e?e??t?? µp??e? ?a ???fe? ?a? se s???e???µ????? ?ata????t?? ?a?

?ts? ?a d?aμ??f??e? d??aμ??? t?? ?e?t?????a ?a? t?? d?μ?? t?? ?p?μ???d?? Queue Manager Block ?a?Cell Scheduler Block. It can also receive data by reading daticated for this purpose registers.

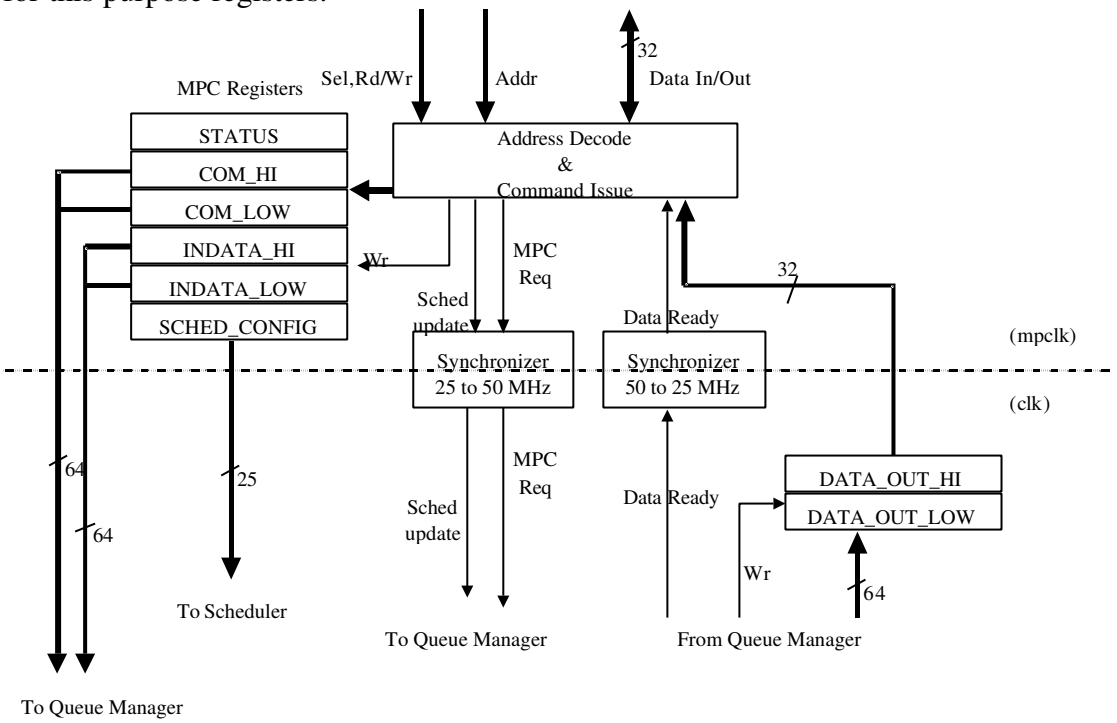

Figure 3-2: The CPU Interface sub-block diagram

The subblock seen in figure 3-2 that is responsible for the access of the CPU dedicated registers is Address Decode block and Command Issue. Its purpose is to decode the data of the CPU address bus and execute a Read or Write Operation according to the address. When special registers used for command request are writen by the CPU, a signal named Command Request is generated by the subblock to notify (after synchronization) the Queue Manager that a command has been issued by the CPU , that must be executed by the Queue Manager.The same function has the signal Sched Update that is generated each time the CPU writes the dedicated QoS register. This signal after synchronization notifies the Cell Scheduller of the change.

3.2.1 MPC 860 Interface Block - Queue Manager Block Interface

The MPC 860 Interface Block can configure the Cell and Queue data Memory unit that lies in the SDRAM. The SDRAM fully controlled by the Queue Manager and CPU can access it by placing certain commands to the latter. Some command include data to enumarate the Queue and flow structures, others aren't, while others return data to the CPU.
The commands that the CPU can place at the MPC860 Interface block concerning the Queue Manager and the registers used for each (Refer to figure 3-2) are the following:

1) **Open Flow**: Initilizes flow, requires write access of registers COM_HI, COM_LOW.
2) **Close Flow**: Terminates a flow, , requires write access of registers COM_HI, COM_LOW.

3) **Write**: Writes a 64-bit word to SDRAM. It requires write access of registers COM_HI, COM_LOW, INDATA_HI and INDATA_LOW.
4) **Read**: Reads a 64-bit word to SDRAM. It requires write access of registers COM_HI, COM_LOW.
5) **Read Counter**: Reads the counter of cells queued for a certain flow. It requires write access of registers COM_HI, COM_LOW
6) **Change Parameters** : Change the High and low thresholds used for cell marking of misbehaving flows. It requires write access of registers COM_HI, COM_LOW.

For each of the commands to be issued, at least two registers must be written. It is also necessary, that the last register to be written should be COM_LOW, since it sets signal Mpc_Req. The latter after passing synchronization from 25MHz of MPC 860 Interface Block to 50 MHz of FPGA logic, notifies the Queue Manager sub-block, that there is a new Microprocessor command to be executed.
Read and Read Counter Commands request data to be returned to the CPU by the Queue Manager IP. When the latter creates these data, stores them in registers DATA_OUT_HI and DATA_OUT_LOW. The two 32 bit are necessary since the data are 64 bits long and the MPC860 Interface is only 32 bits long. DATA_OUT_HI stores the 32 most significant bits while DATA_OUT_LOW the least significant. Writing of these two registers sets the signal Data_Ready that after passing syncronization from 50 MHz to 25 MHz it set the corresponding bit in the STATUS register. Thr CPU polls the status register and learns that the data are ready for accessing. The STATUS register also informs the CPU that it is able to issue a new command by another bit.
The command arguments and their execution by the Queue Manager is presented in detail in subsection 4.2.4.

3.2.2 MPC 860 Interface Block - Cell Scheduler Block interconnection

The MPC860 Interface Sub-block allows the MPC860 to shape the Quality of Service given to each one of the 64 Flow Groups that can exist at any time at the ABRSU, by programming the Cell Scheduler. Writing QoS parameters to the SCHED_CONFIG register does this. In this register the CPU writes the minimum service interval which is the minimum time between two successive dequeues from flows in that Flow Group- and the id of the respective Flow Group. When this register is written by the CPU, the pulse signal, Sched_update, is set and after passing from the synchronization logic it is used as a write enable for the Service Interval Memory that exist in the Cell Scheduler.

3.3 The Cell Demultiplexor

The Cell De-multiplexor block is responsible for the accumulation of a hole cell in 7 words of 64 bits each, in order to be sent to the Queue Manager sub-block for enqueueing in the appropriate queue. This block is necessary because first enters in the Cubit Interface block and stored in a 16-bit fifo and must be restored in 64-bity fifo so that when enqueue operation is performed succesive 64-bit word should be provided to the SDRAM Interface on eack clock cycle.

The Cell Demux FSM polls the avail signal coming from the Cubit Interface sub-block to see if there is a cell for enqueue. If this signal is one then a whole cell is waiting in the Cubit Interface blocks FIFO. The FSM then proceeds in filling a 64-bit register with 16 words read with the appropriate signal from the Cubit Interface block. Each time the register fills a write operation is done by the FSM to the Input cell buffer with the data of the 64-bit register. The Input Cell Buffer is a 7x64 show ahead FIFO (show ahead FIFO is the one that outputs immediately the first datum after a write). When this FIFO is full a cell is ready to be enqueued and actually the fifo_full signal of this FIFO is used as an enqueue request to the Queue Manager. No new cell import can begin by the FSM until the FIFO becomes empty again. This is done by the successive reads by the Queue Manager from this FIFO. The first datum of each cell contains the Flow ID that the Queue Manager uses for enqueueing the cell to the respective queue (This is the reason that the FIFO is show-ahead). After the Input cell buffer empties, a new cell can be loaded into it and wait for its enqueue operation.
The Cell Demultiplexor sub-block can be seen in Figure 3-3.

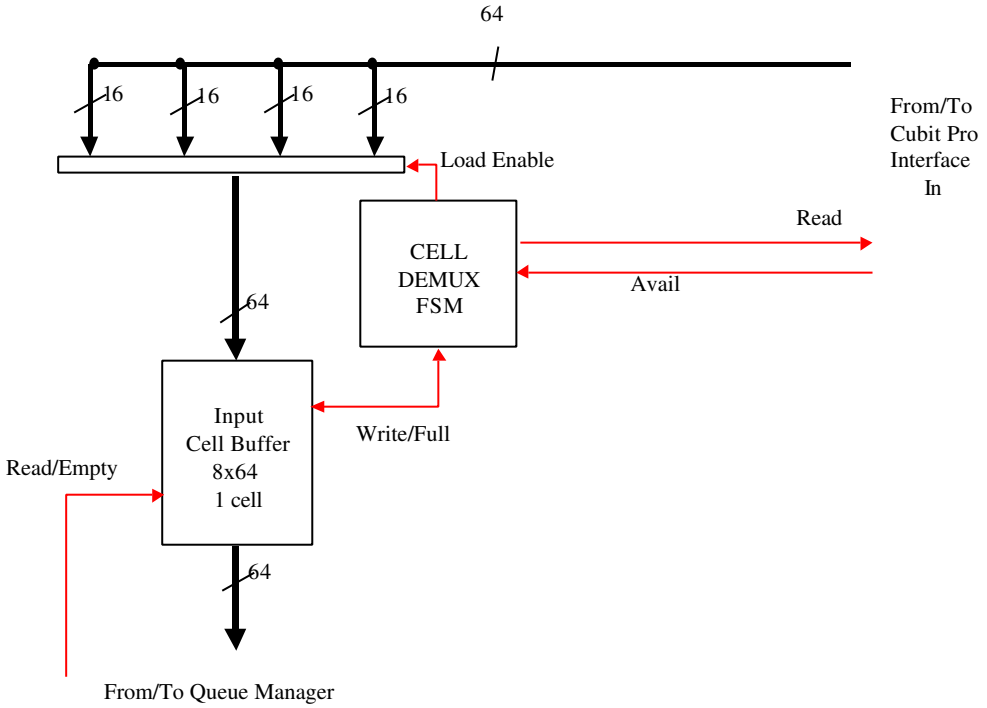

Figure 3-3: The Cell Demultiplexor sub-block diagram

3.4 The UTOPIA Interfaces

There are two UTOPIA interfaces, both of 16 bits, incorporated in the ABRSU. The first is called UTOPIA Input Interface and is used to import the cells from the Cubits of the ABR Server Card into the ABRSU in order to be enqueued. The second is called UTOPIA Output Interface and is used to sent the dequeued cells from the ABRSU to the Cubit devices of the ABR Server card for transmission over the CellBus to the card of destination. A brief description of these two interfaces is given in the following two subsections.

3.4.1 UTOPIA Input Interface

A simple block diagram of this 16-bit interface is given in figure 3-4. The Cubit Devices send data to the FPGA (ABRSU) through a 16 bit Utopia interface. This block has a controller that recognizes and responds to Utopia signals. It can therefore be notified of the beginning of a cell and store its 16-bit words in an elastic FIFO, with the use of the utopia clock provided by the Cubit. This FIFO is 8 cells long. A separate counter exists to count the numbers of cells that exist in the FIFO. Note that a cell is considered to be inside the FIFO (Counter ++) only when the whole cell has entered the FIFO. This is done, in order to allow the Cubits to send a cell inconsecutively and the rest of the ABRSU to receive it consecutively. If the FIFO is full then the controller notifies with the avail signal to send no other cell. On the other side the Cell Demultiplexor see an avail signal that notifies it (After passing synchronization from the Cubit Clock to the FPGA Internal Clock) that there is a cell inside the FIFO. The Cell Demultiplexor then proceeds in reading the cell out of the FIFO and into the Input Buffer FIFO, by use of the read port of the FIFO that has different clocking than the write port. The port clock is FPGA internal clock. Cell Counter is decreased when the whole cell has been read out of the FIFO.

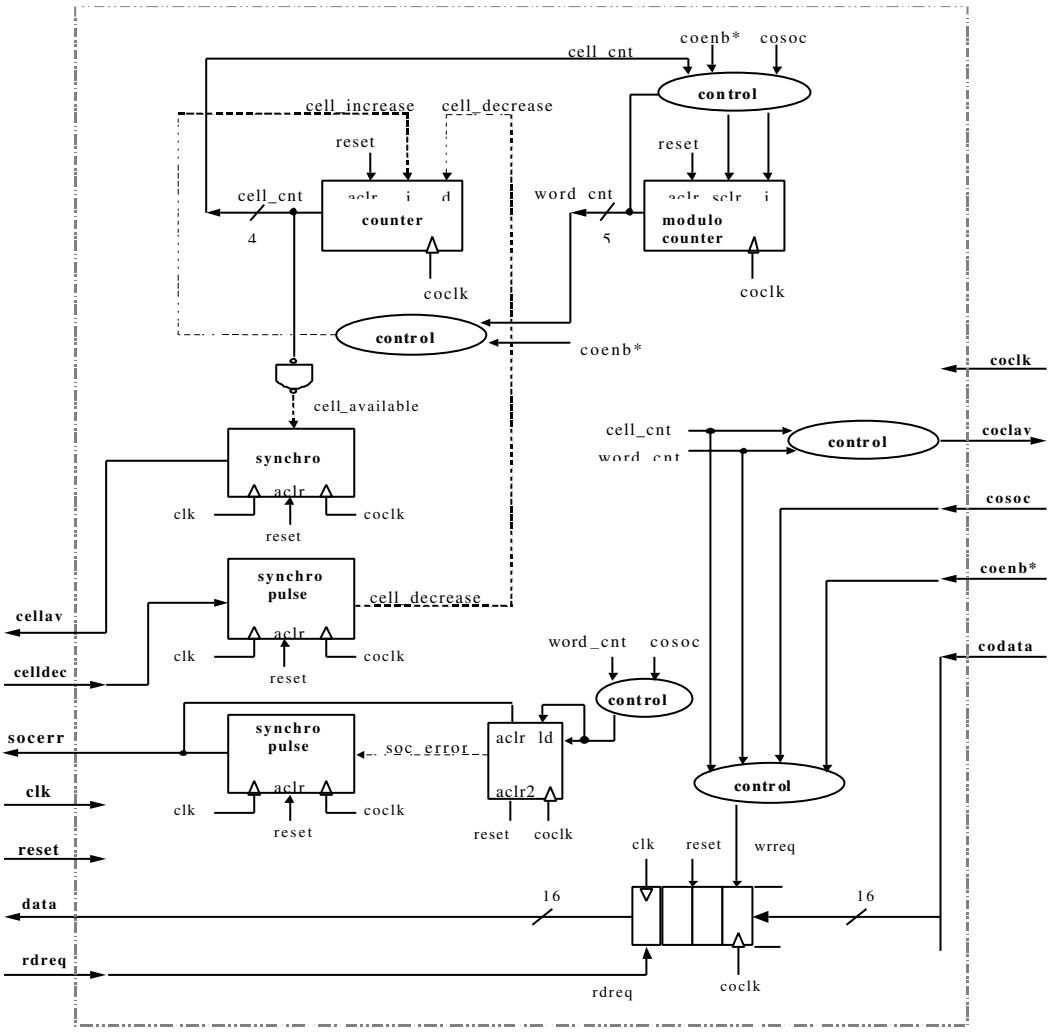

Figure 3-4: UTOPIA Input Interface block diagram

3.4.2 UTOPIA Output Interface

The UTOPIA Output Interface is responsible for forwarding the dequeued cells from the ABRSU to the Cubits for transmission over the Cellbus. From the side of the Cubits it operates on the UTOPIA clock, produced by the Cubit Devices. A Controller sub-block uses this clock along with the UTOPIA handshake signals to transmit 16-bit word of a cell stored inside an elastic FIFO. This FIFO 8 cells long. If the Cubit internal FIFOs are full then the controller stops transmitting new cells. On the Side of the ABRSU, the Cell Scheduler that receives a cell after dequeueing uses the write port of the FIFO that operates on the FPGA clock to fill it. If the cubit accept no cells then the FIFO becomes full and the Cell Scheduler receives an no apace available signal. The scheduler then stops scheduling any new transmissions. An external to the FIFO counter is used to count the number of cells inside the FIFO. Signals that increase this counter, as well as avail signals, pass from synchronizing logic that lies between the clock domains.
A block diagram of the UTOPIA Output Interface is given in figure 3-5.

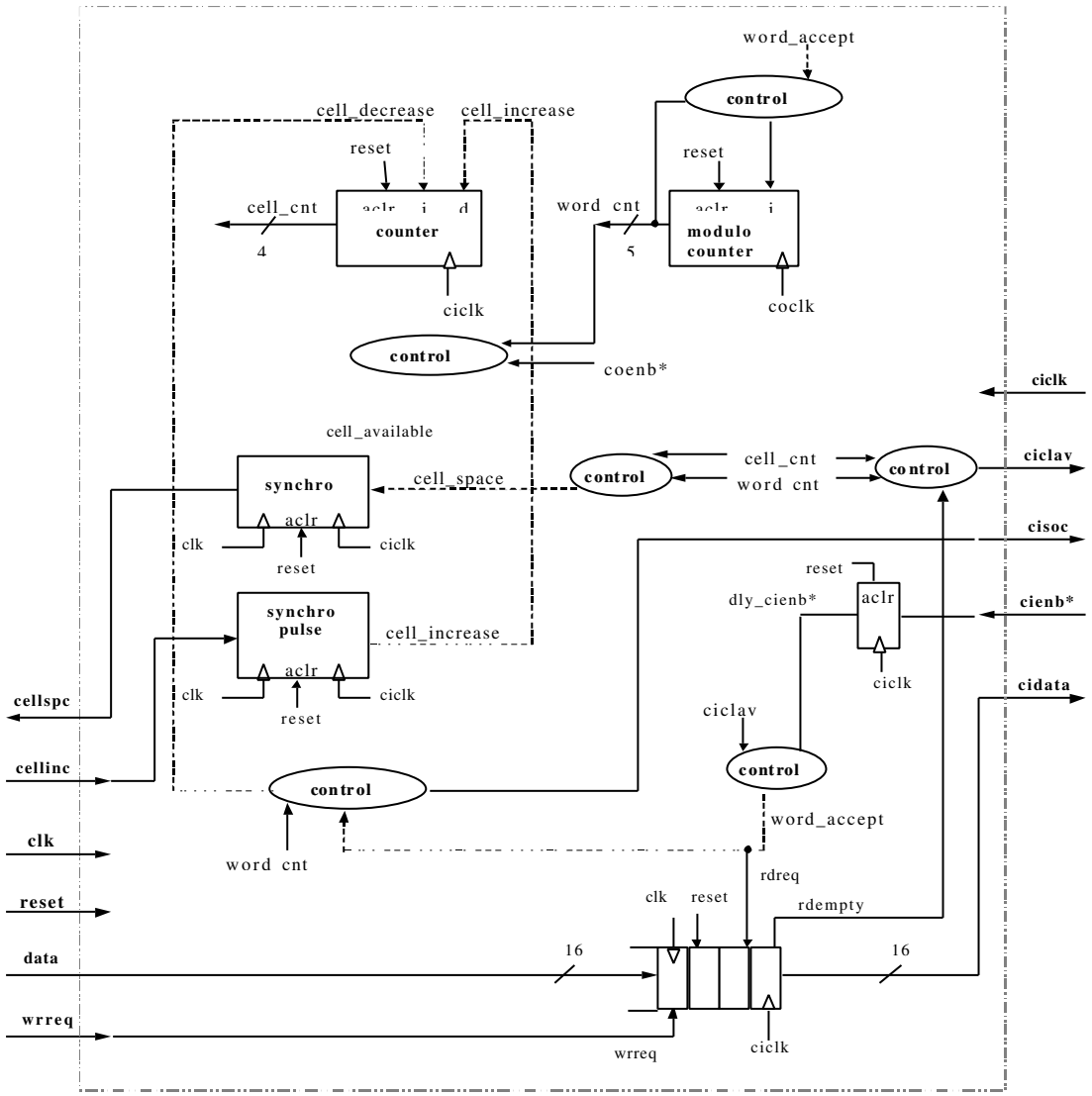

Figure 3-5: UTOPIA Output Interface sub-block diagram

3.4.3 The Pulse Synchronizer

Figure 3-6 gives the internal organization of Synchronizing block that is used to pass signals between the UTOPIA interface clock domain and the rest of the ABRSU clock domain.

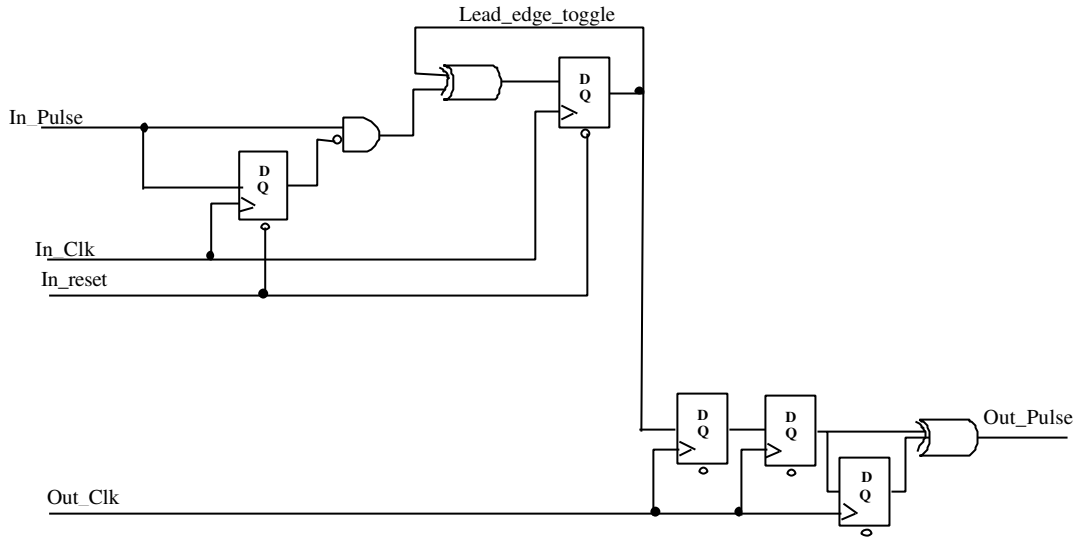

Figure 3-6: The Pulse Synchronizer

The main features of this synchronizer are:

- **It requires the following to be true:**
  Cycles between in pulses > 2 * Tout / Tin
- **it guaranties that there is more than two output clocks between changes of lead_edge_toggle**
- **no conditions on clock relation, frequencies duty cycles and pulse widths**

3.5 The Queue Manager

This block is the main subject of this work and is described in detail in Chapter 4.

3.6 The Cell Scheduler

The Cell Scheduler is, along with the Queue Manager, one the two most important sub-blocks of the ABRSU. While the Queue Manager implements the queueing architecture of the ABRSU and the ABR Server Card in general, the Cell Scheduler implements the scheduling policy. This block can distinguish 64 separate flow groups each with different QoS characteristics and services them by requesting by the Queue Manager to dequeue a cell from a flow belonging to the group for transmission over the CellBus. Flows in each of the Flow groups are serviced equally with a Round Robin mechanism. Maintaining the structure of each Flow Group is a responsibility of the Queue Manager. The main functionality of the block is:

- Maintain a memory with Service Intervals for each of the 64 Flow Groups. These service intervals give the minimum time between subsequent dequeue operations on a Flow Group. The CPU, through the MPC860 Interface register SCHED_CONFIG writes these service intervals to the Service Interval Memory.
- Schedule the next request for a dequeue operation from a specific Flow Group and send it to the Queue Manager
- Receive the cell from the Queue Manager along with the address of the block inside the SDRAM where the cell is stored.
- Forward the cell to the CUBIT Output Interface for transmission over the Cellbus. Wait for an acknowledgement from the Cubits that the cell was transmitted successfully.
- If a negative acknowledgement arrives (the cell was transmitted with an error) log the reason in a special memory that notes which Flow Groups have congestion on their destination Cards and perform exponential back off on the retransmission of the failing cell. When the time of retransmission arrives, request from the Queue manager to read again the cell from the SDRAM, by providing the first with the address of the cell inside SDRAM memory, kept from the first dequeue.
- If a positive acknowledgement arrives then schedule the next request from the Queue Manager and place the address of the cell just transmitted to a FIFO of free cells that can be reused. This FIFO is read by the Queue Manager when is not empty, to get a pointer to a free block that can be used by the first to enqueue a new arriving cell. This saves the Queue Manager from an access to its list of free pointers and thus an access to the SDRAM. The whole procedure is called free list bypassing.

The cell scheduler is not discussed further since it is not the subject of this work but more can be found on **[23]**

4 The Queue Manager IP

In this chapter, the Queue Manager IP is presented. The IP was implemented with the use of Verilog Hardware Description language and synthesized by MaxPlus II synthesis tool, to fit in an Altera FPGA. This IP uses only one SDRAM DIMM as the only memory module to store the incoming ABR cells, as well as metadata for the logical queues of cells, thus dropping the pin count and costs of the design. Per-flow queueing is implemented as the queueing architecture. This means that each separate flow of traffic served by the switch reserves a special FIFO queue for storing its cells. The number of maximum flows of traffic that can be served is 64K at any time that is more than enough for the needs of an edge switch. A list of free buffers is also maintained, to keep all the free buffers that are not used and provide them for storing incoming cells. Thus dynamic memory allocation is implemented allowing specific flows to have queues of arbitrary size and for better memory utilization. Flows of traffic are also organized in Flow Groups by using cyclic lists. Metadata for these flow groups lists is kept in SRAM memories internal to the FPGA. Initialization of flows and definition of the flow group that they belong is being done by the card micro-controller at connection setup, with set up commands issued through the CPU interface of the IP. Flow Control parameters can also be set during flow initialization by the CPU. It does this, by setting information on the maximum queue size allowed to the respective flow. Beyond this size, EFCI and RM marking as described by the ATM forum is performed.

Since only one memory module is used for the needs of the IP, no parallel accesses can be made. Only sequential accesses are possible during the enqueueing or dequeueing of a cell. This fact makes the memory the bottleneck of the queueing bandwidth. In order to reduce memory accesses buffer pre-allocation to each of the 64K queues is implemented, as well as free-list bypassing with the help of logic external to the IP (the Cell Scheduler in the ABRSU environment). The SDRAM controller that is incorporated in the IP to handle the accesses with the SDRAM module is also carefully designed to handle consecutive accesses without loss of clock cycles. The enqueue and dequeue operations are designed in a way that avoids data dependencies (the result of a read operation is used as the address of the next read or write access) that can induce idle clock cycles during the execution.

Except the interface with the CPU and the SDRAM module, the IP keeps an interface for receipt of cell to be enqueued and an interface with the scheduling module (the Cell Scheduler in the DIPOLO environment) that makes requests for dequeueing cells from a specific flow group. The request is served by the IP dequeueing a cell from that flow group and sending it to the interface in 64 bit data words.

4.1 The Queue Manager IP Architecture

In Figure 4-1 a sub-block diagram of the Queue Manager IP is given.

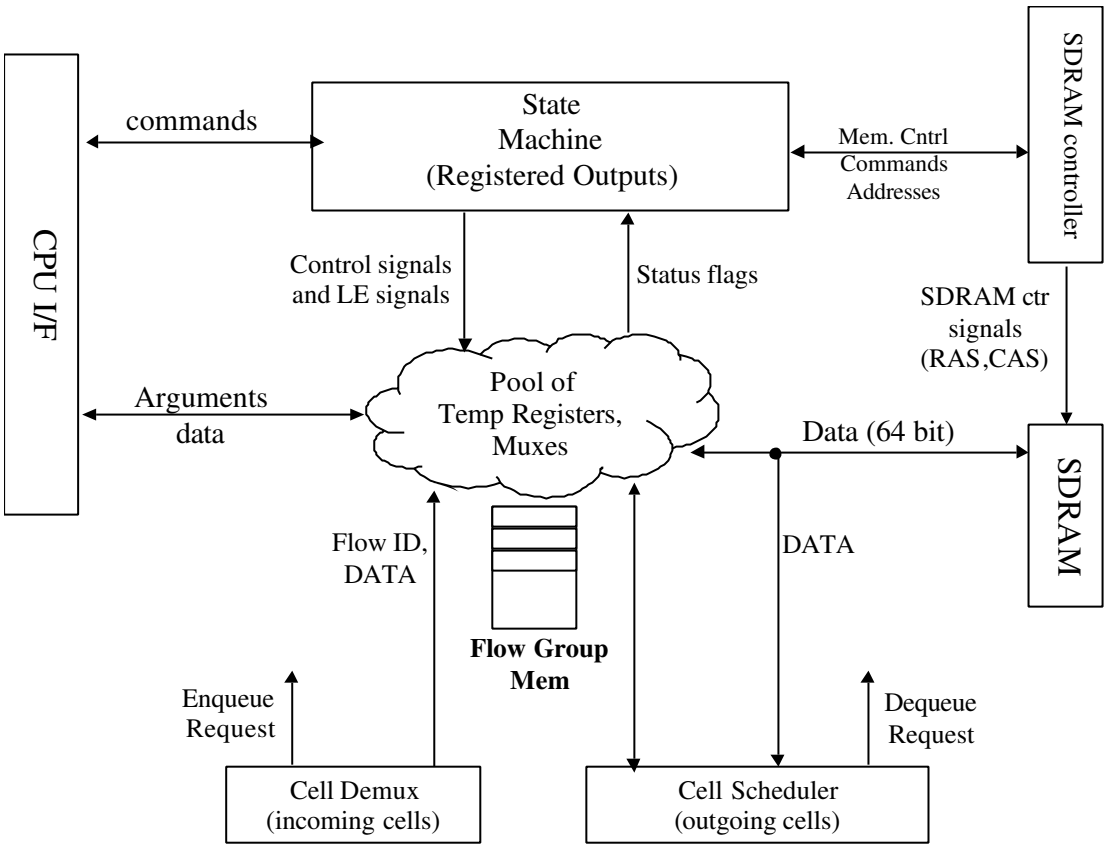

Figure 4-1: The Queue Manager IP sub-block diagram

As seen on the figure the Queue Manager IP is composed by:

- **A Large State Machine**: This state machine is actually composed of many state machines each dedicated to controlling the rest of the logic and memory of the IP in order to implement a specific command. All of these State machines are initiated by a TOP state machine that arbitrates which command should be executed at a given time.
- **A Pool Of Temp Registers and Muxes (Datapath)**: These temporary registers are loaded with data coming from all the blocks that interface with the IP. They are loaded with load enable signals produced by the State Machines and they are preceded by muxes that select the origin of the data stored. Again, the State Machines sets the select signals of these muxes. Names of these registers and their respective muxes, to name a few, are: Flow ID, Head Pointer, Tail Pointer, Cell Counter, Hi Watermark, Flow Group ID (of a certain queue accessed at a given command), Free List Head, Free List Tail, Free List Counter (Queue information for the list of Free Buffers of the Cell Memory (SDRAM)) etc.
- **Flow Group Memory**: This Memory keeps data for the 64 Flow Groups supported by the Queue Managers. It is a 64 word memory that stores the Head of

the list of active flows that belong in the Group, the tail of the list, and the status of the Flow Group (if the Flow Group has any active Flows or not). A Flow is considered active when a cell of this flow exists in the Cell Memory (SDRAM) and must be dequeued in the future. The State Machines are doing control of this memory.

- **The SDRAM controller**: This block implements the accesses of the SDRAM DIMM on behalf of the State Machines. It sets the SDRAM DIMM control pins and programs it to read/write bursts of 1,2,8 words of 64 bits each. Data are given directly by the rest of the IP in synchronization with the control signals of the sub-block. No synchronization is implemented since the FPGA that implements the IP and the SDRAM DIMM use the same carefully distributed clock.

The Queue Manager IP interfaces with the following blocks:

- **The CPU Interface**: This interface allows a given CPU the ability to configure the IP, initialize flows, configure the flow control attributes of flows and the QoS parameters of Flow Groups and extract debug information.
- **The Cell Demultiplexor Interface**: By this Interface the IP accesses a 64-bit FIFO that contains a cell to be enqueued. If the FIFO is full a cell exists and the full signal is used as an enqueue request to the State Machines.
- **The Cell Scheduler**: This is the interface with the scheduling block that implements the scheduling architecture. A dequeue request can be given to the IP through this interface along with the ID of the Flow Group that the dequeued cell must come from. The IP responds by sending the cell in 64-bit words, along with the ID of the Flow it belonged to and the address of its respective buffer in the SDRAM DIMM. When free-list bypassing is implemented, such addresses are given back to the Queue Manager for restoring newly arrived cells.
- **The SDRAM DIMM interface**: This Interface is controlled as described above by the SDRAM Controller and only data are given by the rest of the logic.

A detailed description of the Interfaces of the IP is given in Subsection 4.2.1.

4.2 Functional Implementation

In this section, the functionality of the Queue Manager is described thoroughly. First, the Interfaces of the IP are given in detail, and a description is provided for all the buses and signals. Then, the main data structures of the IP are presented. The commands that are accepted by the Queue Manager are given afterwards as well as a description of their arguments. Finally the Flow Control provisions of the IP are examined.

4.2.1 Interfaces

In the following sections a detailed description of the interfaces of the Queue Manager IP is given.

4.2.1.1 The Interface with the CPU

In table 4-1 the buses and signals of the CPU – Queue Manager Interface are given. Through this interface the CPU can place commands to the Queue Manager and receive the results when necessary.

Table 4-1: The Interface with the CPU

Command [63:0]	Input	This Input bus is used by the CPU to give a command to the Queue Manager for execution.
In_Data [63:0]	Input	If the command requested by the CPU is a Write operation to a 64-bit word in SDRAM, then this bus is used to give the data that are written.
Out_Data[63:0]	Output	If the command requested by the CPU is a Read operation from a 64 bit word in SDRAM, or any other command that requires data to be returned to the CPU, then this bus is used to give to the CPU the data.
CPU_req	Input	When the CPU requests a command from the Queue Manager this signal is used as a flag of the request. Command[63:0] and other buses must be valid.
Com_Acc	Output	When the Queue Manager starts the processing of a CPU command this flag signal is set to notify the CPU that it can prepare a new command since the previous command attributes have been latched in the IP.
Com_ready	Output	When a CPU command has been executed and resulting data are valid on Out_Data[63:0] then this signal is set to notify the CPU.
CPU_lock	Input	For debugging reasons, the CPU can set this signal that orders the Queue Manager to process only CPU commands and stop any enqueueing or dequeueing of cells.

4.2.1.2 The Cell Demux Interface (Incoming cells)

In table 4-2 the buses and signals of the Cell Demux – Queue Manager Interface are given. This interface can be used to give to the Queue Manager the incoming cells that must be enqueued. The Queue Manager is the master of the Interface and decides when the cells can be read in.

Table 4-2 : The Cell Demux Interface (Incoming cells)

Enq_Req	Input	When a cell is available in the 7x64 Input buffer of Cell Demultiplexor then this signal is set to notify the Queue Manager that a cell

		needs enqueueing. This signal is the fifo full signal of the buffer.
Cell_DataIn[63:0]	Input	When Enq_Req = 1 then this bus gives the 7 64-bit words of the cell. The first word of the cell is valid on the first cycle the Enq_Req = 1 and contains the Flow ID that the cell belongs to.
Read_En	Output	The Read_En that notifies the Buffer that acell word has been latched and it must produce the next in the next cycle.
Cell_Read	Output	When a whole cell has been read, this signal is set by the Queue Manager to notify the buffer that it can be loaded with a new cell.

4.2.1.3 The Cell Scheduler – Queue Manager Interface

In table 4-3 the buses and signals of the Cell Demux – Queue Manager Interface are given. This interface can be used by the Scheduling logic to request cells of specific Flow Groups to be dequeued and given to it. If subsequent transmission of the cells fails re-read of the cells can be requested also. A FIFO of free buffers from successfully transmitted cells can be kept in the Cell Scheduler, and its elements can be given to the Queue Manager through this interface, to assist the latter in free-list bypassing.

Table 4-3: The Cell Scheduler – Queue Manager Interface

Deq_Req	Input	When the Cell Scheduler wants to make a request to the Queue Manager, to dequeue a cell from a specific Flow Group, it sets this signal.
Flow_Group[5:0]	Input	When the Cell Scheduler wants to make a request to the Queue Manager, to dequeue a cell from a specific Flow Group, it puts the Flow Group ID on this bus (from 0 to 63)
Cell_DataOut[63:0]	Output	The Queue Manager uses this bus to give the dequeued cell to the Cell Scheduler in 7 64-bit words.
Write_En	Output	The Queue Manager uses this signal to write the in 7 64-bit words into the Cell Scheduler. This signal is used as a write enable in an internal cell buffer kept in the Cell Scheduler.
Flow_Id_Out[15:0]	Output	During the cycle of first word of the outgoing cell, this bus contains the ID of the Flow that the cell belongs to.
Cell_Addr_Out[21:0]	Output	During the transmission of the outgoing cell, this bus contains its buffer address inside the SDRAM memory module. The Cell Scheduler will store this address in case the transmission

		of the dequeued cell fails. In that case it will request from the Queue Manager to read again the cell from the SDRAM, by providing this address.
Read_Cell_Req	Input	If the transmission of a previously cell fails, then the Cell Scheduler uses this signal to request from the Queue Manager to read again the cell from the SDRAM
Cell_Addr_In[21:0]	Input	If the transmission of a previously cell fails, then the Cell Scheduler uses this bus to give the address of the cell that must be read again from the SDRAM The address was given to Cell Scheduler during the first dequeue, through Cell_Addr_Out[21:0] bus.
Free_Cell_Empty	Input	In case a cell is transmitted without error its address in the SDRAM memory is kept in a FIFO that is read by the Queue Manager. This FIFO provides free buffers to the latter for free list bypassing (See Subsection 4.3.7). This signal when set notifies the Queue Manager that this FIFO is empty and no free list by-passing can be done.
Free_Req	Output	If the FIFO of freed buffers inside the Cell Scheduler is almost full, the latter requests with this signal from the Queue Manager to dequeue one free buffer.
Free_Cell_Ptr[21:0]	Input	This bus carries to the Queue Manager the head of the list of free buffers inside the Cell Scheduler (when not empty). The Queue Manager uses this buffer for free list bypassing or to add it to the free list inside the IP, in case the FIFO is almost full (Free_req = 1).
Free_Cell_Ld	Output	After reading the Free_Cell_Ptr[21:0], the Queue Manager uses this signal as a read enable of the FIFO to dequeue the buffer and output the next.
Deq_Ack	Output	When a request by the Cell Scheduler (Deq_Req = 1, Read_Cell_Req = 1, or Free_Req = 1) has been accepted by the Queue Manager, the latter sets this signal to 1 for 1 cycle.
Cell_Ready	Output	When a whole cell has been written into the Cell Scheduler (after a Dequeue or a Read Cell request) this signal is set to 1 for 1 cycle.

4.2.1.4 The SDRAM – Queue Manager Interface

The last interface of the Queue Manager presented here, is the only one that is external to the FPGA that hosts it. It is the pin interface of the Queue Manager with

the SDRAM DIMM memory module. Through this interface, the IP makes its memory accesses, in order to enqueue or dequeued cells, or to enumerate the data structures kept there. The control signals and address of this Memory interface is set by the SDRAM Controller sub-block of the IP while the data are given or taken directly by the Queue Manager Datapath. The interface is the SDARM DIMM standard and any commercial SDRAM can be used. Table 4-4 gives the control and data signals-busses of this interface.

Table 4-4: The SDRAM – Queue Manager Interface

MemData[63:0]	InOut	The bi-directional Data Bus of the Interface, used to read/write 64-bit words from/to the Memory
SDRAMaddr[11:0]	Output	The address bus, used by the Queue Manager to give the row address and the column address of an access
BnkEn[1:0]	Output	The signals that select the internal banks of the SDRAM chips on the DIMM
ClkEn[1:0]	Output	These Clock Enable signals activate or deactivate the DIMM
Cs[3:0]	Output	These Chip Select signals select or deselect half of the SDRAM chips on the DIMM.
We_	Output	Write Enable signal. When low, a write access is performed.
RAS_	Output	Row Access Strobe, low when row address is given.
CAS_	Output	Column Access Strobe, low when column address is given.
DQM[7:0]	Output	Mask bits for each of the 8 bytes of the 64 bits words. Not used by Queue Manager.

4.2.2 Flows

The Flow is the main data structure that is supported by the Queue Manager. Its meaning is that of a flow of connection traffic that is served (its cells are enqueued or dequeued) by the Queue Manager. The Queue Manager can support up to 64K flows of traffic. To distinguish between them its Flow has a specific ID (Flow ID) number that is reserved by the CPU during connection-flow set up. Since the maximum number of supported flows is 64K the Flow ID is a 16-bit binary number. Information for each of the flow is kept in Flow Records. These Records have always reserved space inside the SDRAM memory. The Data Structure that implements the Flow is that of the unidirectional list seen in figure. The Record of each flow maintains information for that list like, the head pointer (address of the first element's buffer), tail pointer (address of the last element's buffer), counter (number of cells in the list), status bits (if the flow ID is used by a connection, or if the Flow ID is active) and Quality of Service parameters. Note that a flow is considered used when the CPU has reserved it for a connection during connection setup and active when it has a cell

stored inside the SDRAM. A detailed description of the Flow Record is given in Subsection 4.3.1.
When a cell that belongs to a certain flow (this is determined by the Flow ID that it carries in its header) arrives at the Queue Manager, it is enqueued in the list. The Queue Manager does this, by writing the cell into the free buffer that is always at the end of the list, and setting the next pointer of this buffer to point to a new empty buffer (There must always be an empty pointer at the end of the list. This is due to reasons explained in Subsection 4.3.8). Empty buffers for this purpose are taken from a list (Free List) that is kept by the Queue Manager. The Tail Pointer of the Flow Record is set to point to that empty buffer also and the counter is increased.
When a cell from that flow must be dequeued, and send to the Cell Scheduler for transmission over the Cell Bus, then the Queue Manager reads the cell that is the head of the list and sets the head pointer to point at the next buffer. If the transmission of the cell succeeds then the buffer that was read can be put to the Free list or reused as the empty tail buffer of an enqueue operation (Free List bypassing. See Section 4.3.7).
Figure 4-2 depicts the Flow structure, the enqueue operation and the dequeue operation.

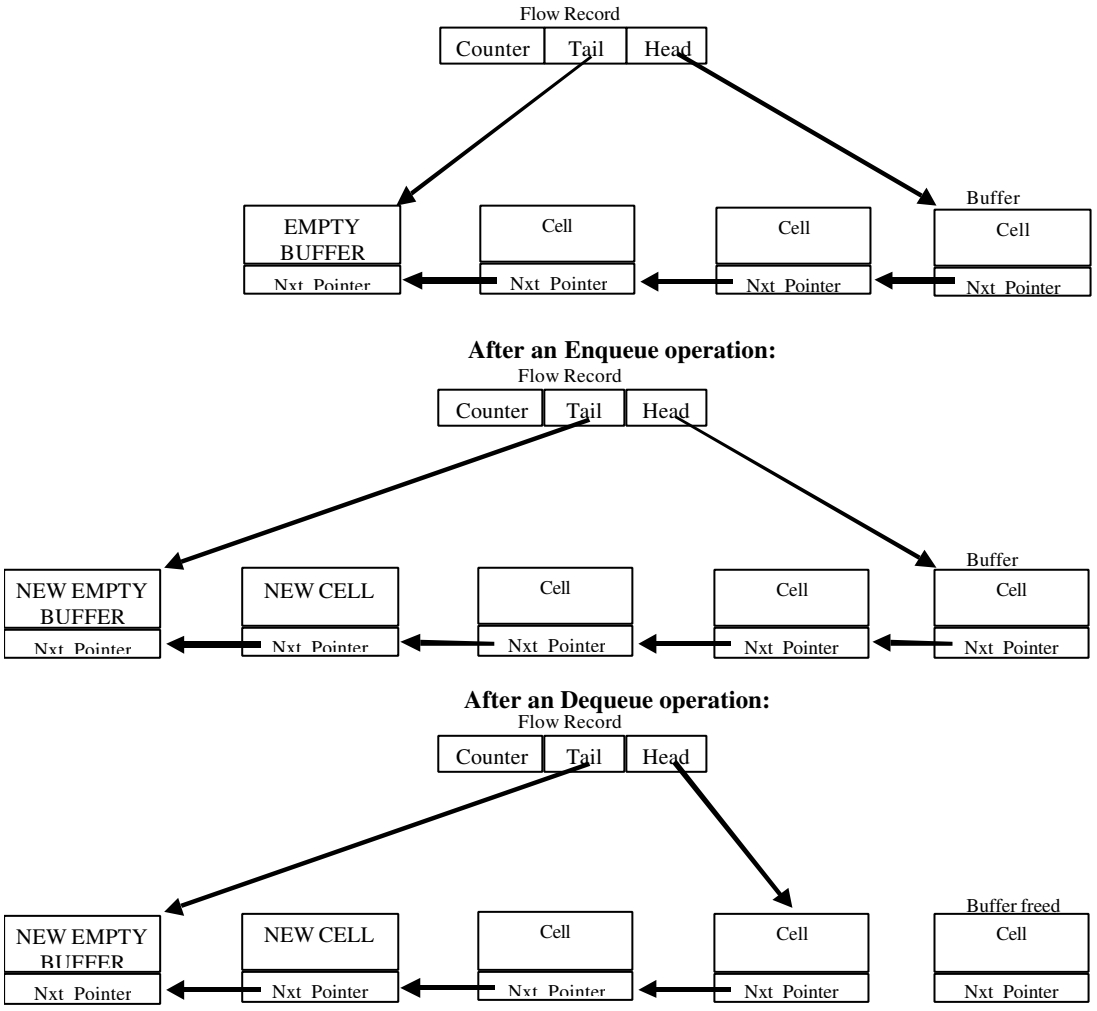

Figure 4-2: The Flow Structure and changes on it after an enqueue and a dequeue operation.

4.2.3 Flow Groups

The 64K flows supported by the Queue Manager are organized in a higher level to Flow Groups. All the flows that are used by connections must belong to a flow group. The number of Flow Groups supported by the Queue Manager is 64. The reason of this grouping is to assist the Scheduling block in the schedule of outgoing ABR traffic. Per flow scheduling according to QoS parameters is impossible for a very large number of flows. For that reason flows are organized in flow groups by the Queue Manager, each Flow Group has its own QoS parameters kept in the Cell Scheduler and the latter has to schedule traffic by providing bandwidth to these 64 Flow Groups. It does this by requesting from the Queue Manager to dequeue cells from specific Flow Groups. The Flows in each Flow Group each get equal amount of bandwidth, since the Queue Manager serves them in a round robin manner.
The data structure that implements a Flow Group is that of a cyclic list depicted in Figure 4-3. Heads and tails of this list (there are 64 of them) are stored in the Flow Group Memory. Only active flows exist in these lists. If a flow after dequeueing its last cell in SDRAM memory becomes empty then it is removed from the cyclic list. When a new cell arrives for that flow, it is then reintroduced in the list by being placed as the tail of the list.

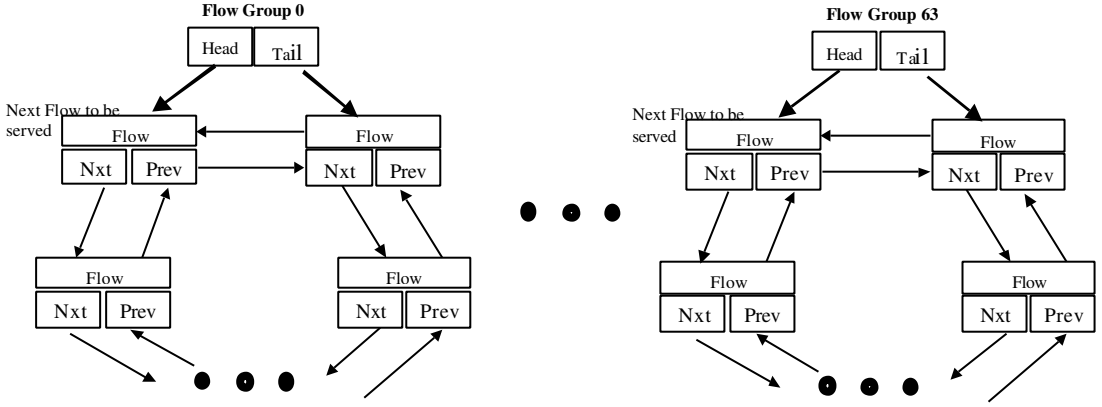

Figure 4-3: The 64 Flow Group cyclic lists

If a the Cell Scheduler requests a dequeue operation for a specific Flow Group, the Flow that is head of the cyclic list will be selected to dequeue a cell and the Tail Pointer will be set to point to that Flow. Then the Head pointer of the cyclic list will point to the next Flow.

The cyclic list is unidirectional since the connection represented by a Flow may close. In that case the flow must be removed from the list while the list must remain connected in a O(1) operation. Pointers to the next and the previous flow for a specific flow are kept in that flow's Record.

4.2.4 Queue Manager Commands

The Queue Manager can accept a range of commands and execute them using one State Machine for each one of them, to control data structures, registers and memory

for each one of them. The most important commands are the Enqueue and Dequeue Commands, issued by the Cell Demultiplexor and Cell Scheduler respectively. Their importance lies in that the number of cycles that spent for their execution defines the Queueing bandwidth of the Queue Manager. Table 4-5 lists the available commands of the Queue Manager, their arguments, their return data and the number of cycles needed for their execution.

Table 4-5: Table of all the Queue Manager commands

NAME	Issued by	Args	Return Data	Clks	Description
Read	CPU	Address	Mem Data	5	Read a 64-bit word from the SDRAM Memory.
Write	CPU	Address, Data		5	Write a 64-bit word to the SDRAM Memory.
OpenFl	CPU	FlowID, FGID, Hwmark, LWmark		10	Initialize a Flow at connection set-up. Reserves a Flow ID, and assigns it to a Flow Group. Sets the Flow Control parameters(Hwmark, Lwmark)
CloseFl	CPU	FlowID		20	Closes a flow at connection shutdown. Releases the Flow ID removes it from Flow Group cyclic list and adds all used buffers to free list.
ReadCnt	CPU	FlowID	Counter	5	Read the counter of buffered cells of a Flow from its record in the SDRAM Memory.
Enqueue	Cell Demux	FlowID, Cell		20, 40	Enqueue of an incoming cell to its respective Flow ID Queue.
Dequeue	Cell Sched	FGID	Address, Cell	20, 40	Dequeue a cell from the Flow that is Head to the cyclic list of the given Flow Group ID. Send it to Cell Scheduler alng with its buffer address.
RdCell	Cell Sched	Address		12	Read the contents of the buffer with the given address.
Free	Cell Sched	Address		10	Add the buffer with the given address to the Free List.
ChParam	CPU	FlowID, Hwmark, LWmark		10	Change the Flow Control parameters of the Flow ID to the given.

The Enqueue requires 20 clock cycles to be executed, except in the special case that the flow was inactive. In that case the Flow must enter into the cyclic list of its respective Flow Group. The records of its previous and next flows in that list must be enumerated adding another 20 cycles in the execution cycles. Thus the number increases to 40.
The Dequeue also requires 20 clock cycles to be executed, except in the special case that the flow becomes inactive (the cell dequeued was the last in SDRAM memory). In that case the Flow must be removed from the cyclic list of its respective Flow Group. The records of its previous and next flows in that list must be enumerated adding another 20 cycles in the execution cycles. Thus the number increases to 40.

4.2.5 EFCI and RM marking [21]

One of the features of the Queue Manager IP is the provision of generic ATM FORUM Flow Control mechanisms. There are two ways that misbehaved flow can be ordered to drop its traffic flow, according to the ATM FORUM specifications.
The first is EFCI (Explicit Forward Congestion Indication). EFCI is a bit included in the header of each ATM cell. When congestion exists or is about to exist in a network node, the latter can set this bit to one, ordering thus the traffic destination to decrease its request of data.
The second is RM (Resource Management) cell marking. RM cells are regularly sent by the source of ABR traffic to the destination node. The destination U turns these cells back to the destination. These cells contain information about traffic resources of the intermediate nodes. Two bits inside this cell, CI (Congestion Indication) and NI (No Increase) can be set to drop or stop the increase of the traffic flow respectively.
The Queue Manager uses both of these mechanisms to instruct a misbehaved flow to decrease its cell rate. A misbehaved rate is determined by a Hi Watermark – Low Watermark mechanism. At the time of initialization of a Flow Record during connection set-up, the CPU set the highest number of cells (Hi Watermark) that the Flow is allowed to use in the SDRAM memory. This Hi Watermark is stored inside the Flow Record. When the Flow Counter during an enqueue operation on that Flow surpasses this Hi Watermark, a bit inside the Flow Record (Mark bit) is set, and the cells that are dequeued from that time and on are EFCI marked. If the cells are RM their CI, NI bits are also marked. Cells are marked at the output in order for the congestion information to reach its target faster. This saves us the time of waiting the number of unmarked cells to leave the Flow Queue.
The cells will stop being marked when the Flow Counter becomes less that Low Water Mark that is also set during the time of Flow initialization. Resetting the Mark bit in the Flow Record does this.
The ChParam command issued by the CPU and described in subsection 4.2.4 changes the Hi Watermark – Low Watermark fields in the Flow Record, thus dynamically affecting the Flow Control mechanism. This means that a well-behaved flow, after the execution of this command can become misbehaved and the marking of its dequeued cells will commence. A detailed description of the fields in the Flow Record, relative to the flow control mechanism of the Queue Manager is given in subsection 4.3.1.

4.3 Design Implementation

4.3.1 Flow Record Format

In figure 4-4 the detailed description of the Flow Record is given. There are 64K records of like this, one for each of the 64K Flows. Each record is 2 words of 64 bits long. This is 2 SDRAM memory words. They are stored inside the SDRAM memory in 2 word alignment.

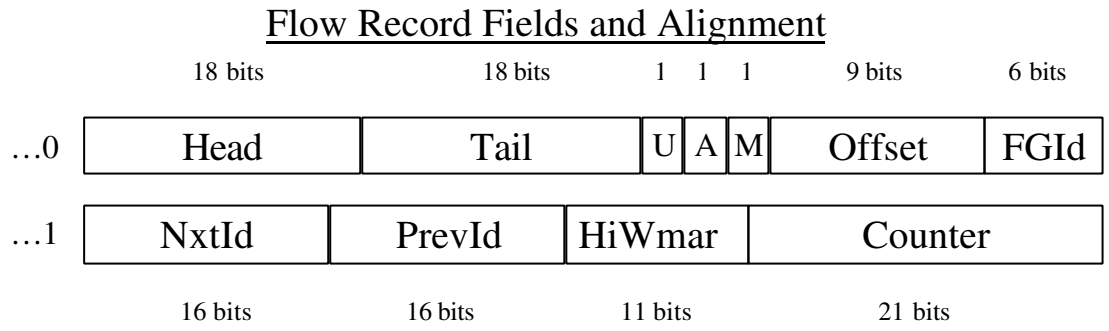

Figure 4-4: Flow Record Fields and alignment

Table 4-6 gives a description of each field in the Flow Record.

Table 4-6 : Flow Record Field bits and description

Field	Bits	Description
Head	22	Head Pointer: It contains the address of the head of the queue of the respective flow.
Tail	22	Tail Pointer: It contains the address of the tail of the queue of the respective flow.
Used	1	Used Flow: When 1 this Flow ID is used by a connection. All the record fields are valid.
Active	1	Active Flow: When 1 this Flow ID has not empty queue. Some cells belonging to the flow are stored in the SDRAM memory.
Mark	1	Mark Bit: The cell count for the respective queue has surpassed the Hi Watermark. Dequeued cells are RM, EFCI marked. Will reset to 0 when Cell Counter drops below Hi Watermark - Off
HiWmH	6	Hi Watermark Most Significant bits: The Most significant bits of the Hi Watermark.
HiWmL	12	Hi Watermark Least Significant bits: The Least significant bits of the Hi Watermark.

FGId	6	Flow Group ID: The Flow Group that this Flow is assigned.
Off	5	Offset: Hi Watermark – Off is equal to the Low Watermark
NextID	16	Next Flow ID: The next flow in the cyclic list of this flow's Flow Group. If Active = 0 this field is invalid.
PrevID	16	Previous Flow ID: The previous flow in the cyclic list of this flow's Flow Group. If Active = 0 this field is invalid.
Counter	20	The number of cells of the flow inside the SDRAM memory.

An effort was made during the design so that the Flow record would be 2 words long. If it was 3 words long it would mess the 2 word alignment of the SDRAM memory. If 4 words it would require 2 extra cycles to access it, dropping the queueing bandwidth.
Buffer pointers such as Head and Tail are buffer aligned in a 256 MByte Memory organized in 2^25 words of 64 bits. A buffer is 8x64bit words, thus a buffer aligned pointer is 22 bits long.
Notice that there is no field for the Flow ID of the Flow that the Record belongs to. This is done because each Flow Records is stored in 2 word whose address is that of the Flow ID. An additional least significant bit is used in the addressing of the record to distinguish between the 2 words of the record. In that organization, we put the records at the beginning of the SDRAM memory (See Sub-Section 4.3.4) and save field space in the record.

4.3.2 Flow Group Record Format

All the Flow Group information is stored in a 64x33 memory, the Flow Group memory. Each of the 64 words of this memory stores the Flow Group record with ID that of its address and keeps all the data necessary for the maintenance of the Flow Group cyclic list. Figure 4-5 depicts the Flow Group memory and the Flow Group records.

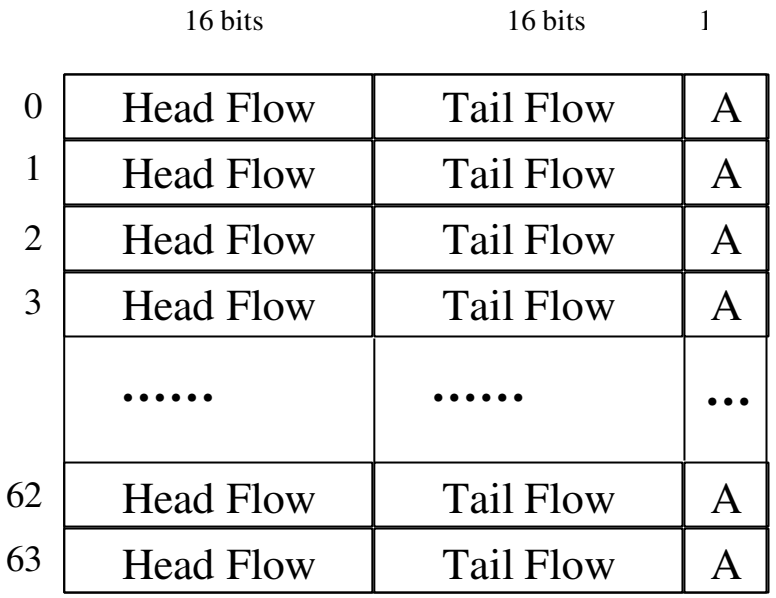

Figure 4-5: Flow Group memory organization and Flow Group records.

Table 4-7 gives the description of the Flow Group Record fields.

Table 4-7: Flow Record Field description

Field	**Bits**	**Description**
Head Flow	16	Head Flow ID: It contains the Flow ID of the Flow that will give the next dequeued cell, when a dequeueing operation is requested for the respective Flow Group.
Tail	16	Tail Flow ID: It contains the Flow ID of the Flow that was served in the previous dequeueing operation for that flow group.
Active	1	Active Flow: When 1 this Flow Group has Active Flows, else no Flow assigned to this Flow Group has any cells inside the SDRAM memory

The contents of the Flow Group memory are visible to the Cell Scheduler. The latter needs to know which of the Flow Groups are active so that it will keep them in the scheduling loop. Request for dequeueing of an inactive Flow Group would cause an error.

4.3.3 Cell Format and Alignment

Each cell is stored inside the SDRAM memory in 8x64 buffers. The 7 first words (7x8=56) store the ABR cell plus the internal Switch header (Cell Bus, Tandem Routing headers in the DIPOLO environment. The last word of the buffer stores the next pointer. This points to the buffer that stores the next cell of the flow queue. Each

buffer is 8 word aligned. Figure 4-6 depicts the Cell Format and Alignment in the SDRAM Memory.

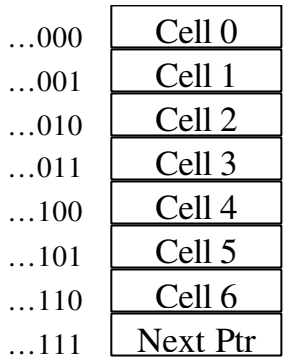

Figure 4-6: Cell Format and alignment

4.3.4 SDRAM Memory Organization

The SDRAM DIMM module that supports the memory needs of the Queue Manager has a capacity of 256 Mbytes. Inside this memory space the Flow Records for each of the 64K flows is stored. The rest is divided in cell buffers that are dynamically allocated to the incoming traffic. The Flow Records are statically allocated. This means that all of the 64K Records are present, even if they are not used.
The CPU is capable of initializing the SDRAM memory by using the Write command. Still, there is an FSM inside the State Machine that after reset can initialize the SDRAM memory. The contents of the SDRAM memory after proper initialization are shown in Figure 4-7.

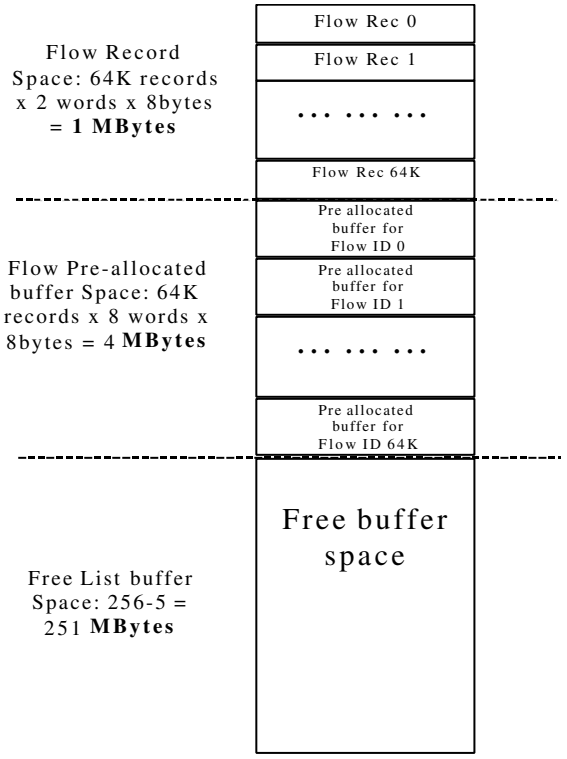

Figure 4-7: SDRAM Memory-space division and organization

Since the memory is 256 Mbytes = 2^28 bytes, organized in 64-bit (8 bytes) words, the number of word addresses in the memory is 2^25. Pointers to a buffer that is 8 words and aligned need to be 22 bits.

As described in subsection 4.3.1, in order not to have a field for the Flow ID of each Record, the ID is used as a pointer to the position of its Flow Record in the Memory and the Records are placed in the beginning of the Memory. Thus, the address of the first word of the Flow Record of Flow ID 0b1111000111110001 (Flow Id is 16 bits) is 0b{00000000,1111000111110001,0}, while the address of the second word is 0b{00000000,1111000111110001,1}. Since the Queue manager supports 2^16 Flows (64K) then the Flow Record use the first 2^16 Flows * 2 words/Flow = 2^17 words = 2^20 bytes = 1 MByte of Memory.

Since each Flow Queue must always have an empty buffer (even if not used, see sub-section 4.3.8) due to buffer pre-allocation, one is given to each during initialization. These buffers are positioned in the SDRAM memory, after the Flow Records. After some time of system operation these buffers are used for cell storing, but others take their place as empty buffers. Thus the number of empty pre-allocated buffers is constant and equal to 2^16 (one for each Flow). So buffer pre-allocation uses up another 2^16 buffers * 8 words/buffer = 2^19 words = 2^22 bytes = 4 MBytes.

The remaining 256 – (4+1) = 251 MBytes can be used for cell buffering. During Memory Initialization, this free space is organized in a large FIFO unidirectional list of free buffers, the Free List. This Free List is used to provide the enqueue operation with free buffers for storing and accept the freed buffers after the dequeue operations.

4.3.5 State Machines

As described in section 4.1, the State Machine sub-block of the Queue Manager contains the State Machines that execute the commands requested by the other blocks. Figure 4-8 shows the internal hierarchy of the FSMs inside this sub-block.

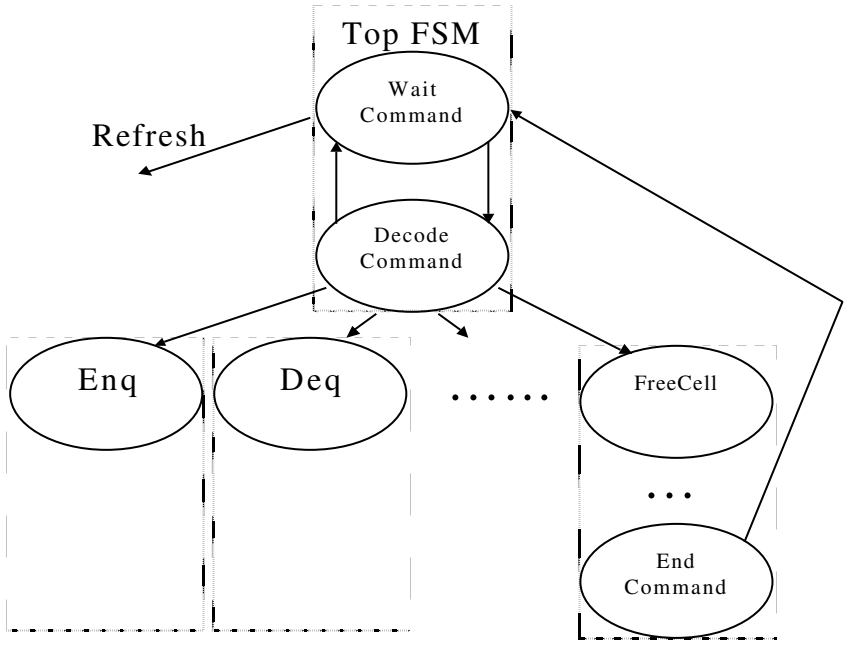

Figure 4-8: State Machine Top Level Diagram

These FSMs set the control signals of the Datapath of the Queue Manager, they also set and accept the control signals of its Interfaces and the make requests for SDRAM memory accesses to the SDRAM controller block. There is one FSM for each one of the commands of the Queue Manager IP. At the top of the hierarchy of these FSMs lies the TOP FSM. The purposes of this FSM are:

- **Initialize the Datapath Registers after System Reset**: Some registers like (Free List Head) need to be initialized in a specific value after reset. The TOP Fsm sets the control signals to do that.
- **Accept the Command Requests by other blocks**: The TOP FSM remains idle, and pools the request signals from the command issuing blocks.
- **Arbitrate Queue Manager Command issue**: There is the case that more than several blocks can request one command, at the same time. The TOP Fsm arbitrates which one will use the Queue Manager, according to some priorities. Initiate a command FSM: When a command is to be executed, the Top Fsm sets the control signal that initiates its respective State Machine. It waits for the control signal from that State Machine that notifies that the execution has finished. It also acknowledges the command acceptance to the issuing block.
- **Request for a SDRAM Memory refresh**: SDRAM Memories must be refreshed periodically. The TOP Fsm has an internal counter that when it reaches zero, issues SDRAM Memory refresh command to the SDRAM controller.

Table 4-8: Queue Manager Command Priorities

PRIORITY	NAME	Issued by	Comments
1 (highest)	Refresh	State Machine	Needs to be executed periodically or else data in SDRAM may be lost.
2	Write, Read, OpenFl, CloseFl, ReadCnt, ChParam	CPU	CPU Commands have the same priority for the reason, that only one can be requested at any time. They have higher priority than the ones issued by Cell Demux, Cell Scheduler because they are sort, they configure and control the Queue Manager operation and are necessary for debugging purposes.
3	Enqueue,	Cell Demux	Enqueue command (issued by Cell Demux) has equal priority with the commands issued by Cell Scheduler. When there is a contention between Enqueue and one Cell Scheduler command, an alternative arbitration is made by TOP FSM. If most recent. command execution was Enqueue, the Sched command will be selected, else if most recent was Sched Command then Enqueue executes
	Dequeue, RdCell, Free	Cell Scheduler	

Table 4-8 gives the prioritization of Commands that is taken into account when there is a contention that must be resolved by the TOP Fsm arbitration. The command priority is given from highest to lowest. Commands in the same row have the same priority.
Figure 4-9 shows a simplified state diagram of the FSM that executes the Enqueue Command. This FSM along with the Dequeue FSM are the most complex in the State Machine Sub-block. The states where an access to the SDRAM is requested to the SDRAM controller are shown with big bubbles that contain the type of access. The maximum number of states/cycles needed is 40. This is the case when the flow was previously inactive. In that case the Flow must enter into the cyclic list of its respective Flow Group. The records of its previous and next flows in that list must be enumerated adding another 20 cycles in the execution cycles. Thus the number increases to 40. Some 5 cycles can be saved when Free-List bypassing is being implemented (See Sub-section 4.3.7). In that case an access to get a new free buffer pointer is avoided.
The SDRAM memory accesses are being made in such an order that data dependencies are avoided (the result of a read operation is used as the address of the next read or write access immediately), while the SDRAM is used continuously.

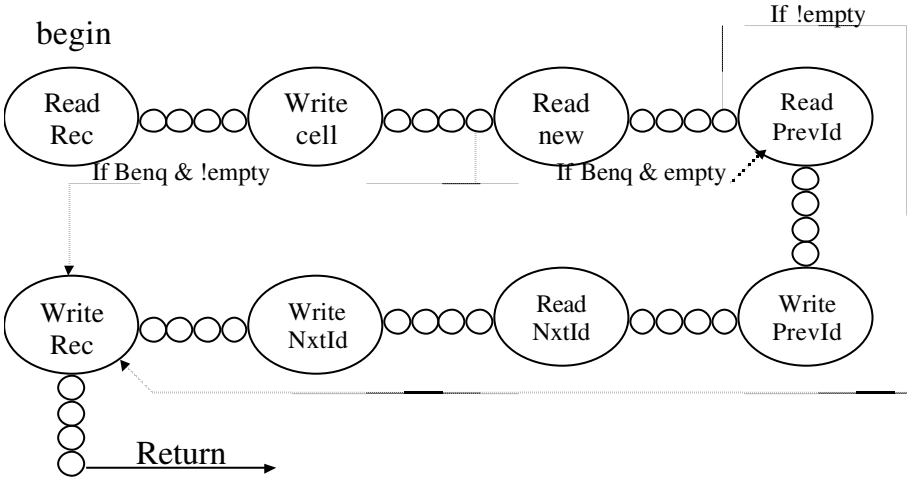

Figure 4-9: Enqueue command FSM bubble diagram

4.3.6 The SDRAM Controller

The SDRAM controller is the sub-block of the Queue Manager that is responsible for controlling the SDRAM memory. It accepts simple burst commands from the State Machines of the Queue Manager and issues these to the memory. The addresses needed are received by the Queue Manager datapath and are used to set Chip Select Signals, Bank Enable signals, as well as to provide the Row and Column addresses for the burst access. The mode register that configures the operation of the SDRAM DIMM is also written during initialization by the sub-block.
The Queue Manager uses the SDRAM Controller for the following operations:

- **Load the Mode Register**: This is done after Reset of the system
- **Auto Refresh**: The TOP Fsm periodically requests the refreshing of one SDRAM row.
- **Read burst of 1 64-bit word**: Read access to half of a Flow Record.

- **Read burst of 2 64-bit words** : Read access to a Flow Record.
- **Read burst of 8 64-bit words** : Read access to Cell Buffer plus its next pointer.
- **Write burst of 1 64-bit word**: Write access to half of a Flow Record.
- **Write burst of 2 64-bit words** : Write access to a Flow Record.
- **Write burst of 8 64-bit words** : Write access to Cell Buffer plus its next pointer.

Figure 4-10 gives the State machine diagram of the SDRAM Controller block:

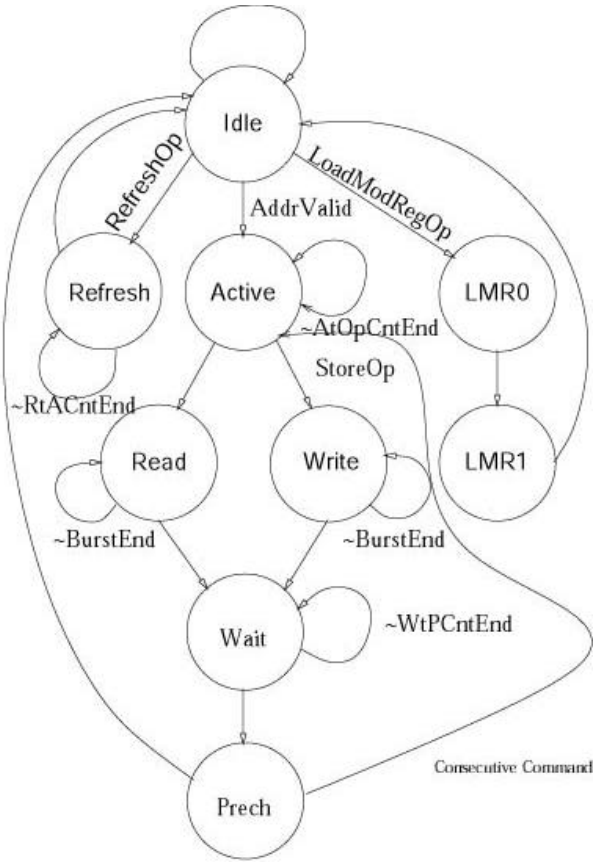

Figure 4-10 : State Machine Diagram of the SDRAM controller

The Controller is in idle state until, a read or a write command is requested by the State Machines. Then it activate the accessed row to be accessed and makes the burst access (stays in Read, Write states for as many cycles as the burst size), waits one cycle and pre-charges the used raw. If a consecutive command has already been requested, the state machine enters to the Active state again, or else in moves to the idle state.
The SDRAM controller doesn't handle the data to/from the SDRAM DIMM. This is done by the Queue Manager datapath, which places them to, or reads them from, the SDRAM 64-bit data bus, in sync with the SDARM Controller State.

4.3.7 Free List Bypassing [13]

The Free-List Bypassing is a Queueing technique implemented by the Queue Manager IP that avoids an additional access to SDRAM memory during an Enqueue or a Dequeue Operation. This decreases the number of cycles needed for both to finish execution and thus increases the Enqueueing/Dequeueing Bandwidth.

When an Enqueue operation is being done a new cell Buffer must be given to the Queue that receives the cell. The Cell Buffer that is pointed by the Free List Head pointer is selected for this purpose but now the Free List Head pointer must be set to point to the next free cell buffer of the Free List. Reading the Next Pointer of the cell buffer taken from the Free List does this. This Next Pointer lies in the External SDRAM Memory and that read access will cost an additional 5 cycles to the Enqueue process.
Moreover, when a Dequeue operation is executed and the dequeued cell is transmitted correctly, its cell buffer is no longer used and must be added to the Free List. This is done in the Queue Manager with the execution of the Free Command that enumerates the Free List Head pointer with the address of the newly added cell and write in the Next Pointer of the latter the address of the previous Free List Head cell. This Write access is done to the SDRAM Memory and cost 5 cycles.
Free list bypassing avoids the cost of the extra 5 cycles spent on Enqueueing and on Dequeueing, as described above. Instead of placing the newly freed cell buffer in the Free List (5 cycles of Free Command), the pointer is placed in a SRAM FIFO inside the Cell Scheduler. When a subsequent Enqueue operation begins, the cell new cell buffer requested, is taken from that SRAM FIFO, thus the extra access to the Free List is also avoided.
In total the Free List Bypassing decreases the number of cycles for enqueueing and dequeueing of a cell by 10 cycles. Since Enqueueing as dequeueing takes 20 or 40 cycles (See subsection 4.3.5), the technique improves the Queue Manager Performance by a factor of 10/(20+20) = ¼ = 25% or 10/(40+40) = 1/8 = 12,5%.
Figure 4-11 depicts the Free-List Bypassing technique

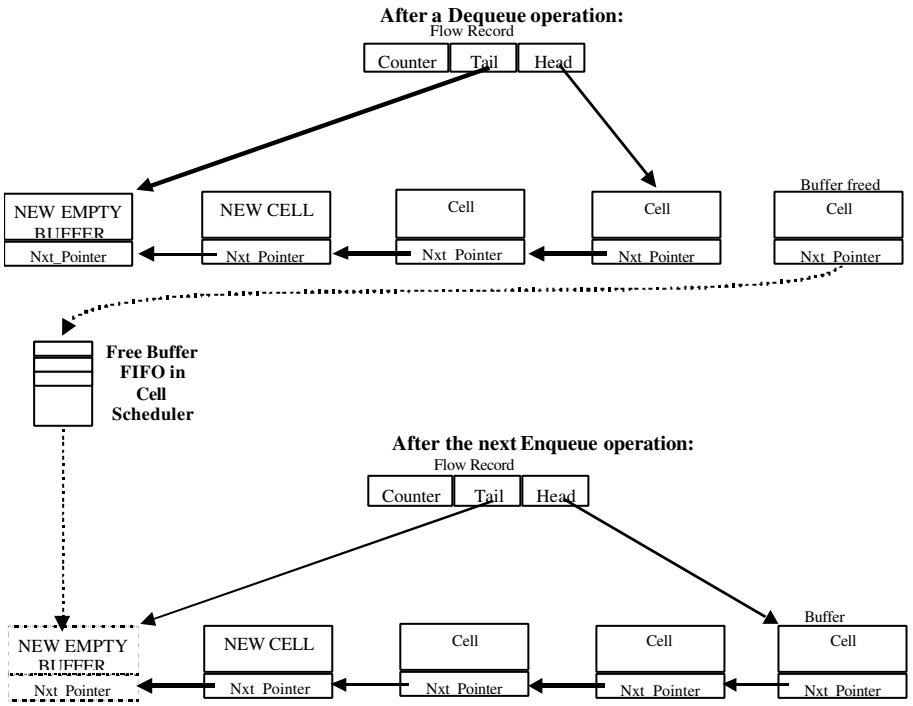

Figure 4-11: Free List Bypassing implementation in the Queue Manager IP

4.3.8 Cell Buffer Pre-allocation [10]

The Cell Buffer Pre-allocation is a technique implemented in the Queue Manager that with the cost of one unused cell buffer per supported Flow, decreases the number of cycles for an enqueue operation by 5.
As described in previous sections, each one of the 64K queues of the Queue Manager has an empty cell buffer as its tail element. This is true even for queues that are empty (inactive). In this case both the Head and the Tail pointer field of the respective Flow Record point to that empty cell buffer (this buffer is called the pre-allocated buffer). If this buffer wasn't pre-allocated and the tail pointer of the Flow Record pointed to the last used cell buffer, then during enqueueing, the newly arrived would have to be written on a new empty cell, taken from the free-list (or the Free-List Bypassing mechanism), and the next pointer of the previously last cell should be set to point to the newly arrived cell. These are two memory accesses in different buffers in memory. Since these two buffers can be in different SDRAM lines, two separate bursts would be necessary.
Instead, with Cell Buffer Pre-allocation, the cell is written to the pre-allocated empty cell buffer that is tail element and the next pointer of the same buffer is set to point to the new pre-allocated buffer taken from the free-list (or the Free-List By-passing mechanism). Only one buffer is accessed in this way, and only one write burst is needed. The extra pointers write access in the non pre-allocating implementation that is avoided costs 5 cycles.
In total, the Free List Bypassing decreases the number of cycles of enqueueing and dequeueing of a cell by 5 cycles. Since Enqueueing as well as dequeueing take 20 or 40 cycles (See sub-section 4.3.5), the technique improves the Queue Manager Performance by a factor of 5/(20+20) = 1/8 = 12,5% or 5/(40+40) = 1/16 = 6,25%. Memory cost of this technique is 4 Mbytes which is 4/256 = 1/64 =1,56% percent of the total memory.

4.4 Timing Issues

In this subsection the bandwidth capabilities of the Queue Manager IP are examined in conjunction with the input/output bandwidth requirements of the ABR Server Cards. The timing parameters assumed and the timing examples that are given prove that the implementation of Free-List Bypassing, along with Cell Buffer pre-allocation are essential if the Queue Manager is to succeed in complying with the Switch requirements, while using only one SDRAM memory module for all necessary memory accesses.

4.4.1 UTOPIA clock Vs Queue Manager (ABRSU) clock.

In order for the Queuing Architecture implemented to be able to balance the incoming and outgoing capabilities of the Switch (DIPOLO), it must be able both to enqueue and dequeue a cell during a cell time. A cell time is the time needed for a cell to arrive or to depart in full in or out of the UTOPIA interfaces. These interfaces are slaves to the switching hardware. Since enqueue and dequeue operations can't be performed in any parallel way and they both need the same number of cycles, the Queue Manager

(and the ABRSU) must be use internal speed up in relation to the UTOPIA Interfaces clock.
A cell needs 28 UTOPIA clock cycles to enter the ABRSU (16-bit interface). On the other hand, both one enqueue and one dequeue operation need:
80 (Worst case) or 40 (Normal Case) Queue Manager clock cycles.
This means that the clock speed up needed for both cases is:

Worst case:
1 Cell arrival time = 1 Enq time + 1 Deq time=>
Tclk_utopia * 28 utopia_cycles = Tclk_qm * (40+40) qm_cycles =>
Tclk_utopia / Tclk_qm = 80 / 28 = 2,8 =>
Speed_up_worst = 2,8

Normal case:
1 Cell arrival time = 1 Enq time + 1 Deq time=>
Tclk_utopia * 28 utopia_cycles = Tclk_qm * (20+20) qm_cycles =>
Tclk_utopia / Tclk_qm = 40 / 28 = 1,4 =>
Speed_up_normal = 1,4

The FPGA selected during the design face to host the Queue Manager can reach clock frequencies of up to 50 MHz. Since the Cubit Clocks are used in the DIPOLO Switch reach 25 MHz of Frequency a maximum speed up value of 2 can be selected. In that case
QM_Clk = 2 * UTOPIA_Clk.
Since:
Speed_up_worst > Speed_up_sel = 2 > Speed_up_normal

The next subsections will show that the Queue Manager is the bottleneck of the system when both enqueue and dequeue take the worst case number of cycles and that the UTOPIA interfaces are the bottleneck system when both enqueue and dequeue take the normal number of cycles.

4.4.2 Worst case of Enqueue and Dequeue

Figure 4-12 gives the timing diagram cells entering and leaving the ABRSU when Enqueue and dequeue commands need 40 cycles each to execute.

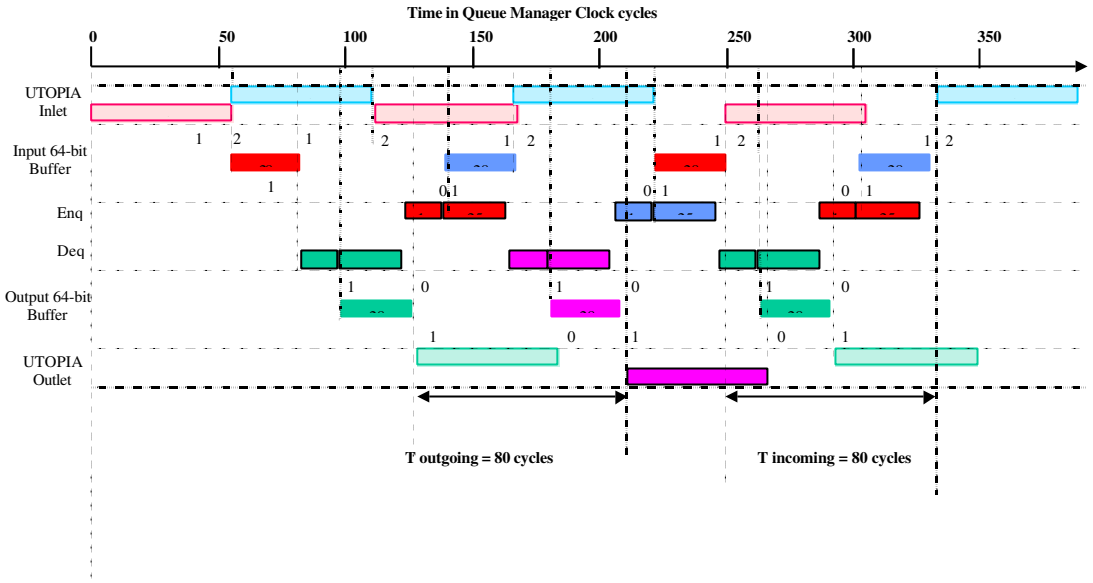

Figure 4-12: Worst case Enqueue-Dequeue timing Diagram

The following parameters have been assumed on this diagram:

- **2 Cell FIFOs in the UTOPIA interfaces. (inlet & outlet) :** The FIFOs inside the UTOPIA interfaces are assumed to be 2 cells long for simplicity. They are 8 cells long **- Single buffering on each Cell Buffer**
- **Speed-up = 2:** The ABRSU speed up is 2. This means that **QM_Clk = 2 * UTOPIA_Clk**
- **Full throttle Traffic**: Incoming cells arrive consecutively, Cell Scheduler requests outgoing cells consecutively.
- **Mixed Enq & Deq Commands**: Enqueue and dequeue commands are executed alternately.
- **40 cycle Enqueue and Dequeue Commands**: Each Enqueue and Dequeue Command takes 40 cycles to complete.
- **15 Cycles access of Input/Output 64-bit Buffers**: The Enqueue command uses its first 15 cycles to empty the Input 64-bit Buffer. The buffer can begin to fill immediately. The Dequeue command uses its first 15 cycles to fill the Output 64-bit Buffer. The buffer can begin to empty immediately.
- **56 QM_clk cycles for UTOPIA I/Fs**: A cell takes 56 QM clock cycles to enter since speed up is 2.

In the diagram, we can see that after 2,3 cell times the UTOPIA inlet is unable to receive the cells consecutively and the UTOPIA outlet to send the cells consecutively. Instead of 56 QM_clk cycles, cells need 80 cycles to enter or leave the ABRSU. This means that the 80 cycles needed by the Queue Manager to Enqueue and dequeue a cell is the bottleneck of the system and sets the cell time to 80 QM_clk cycles.

4.4.3 Normal case of Enqueue and Dequeue

Figure 4-13 gives the timing diagram cells entering and leaving the ABRSU when Enqueue and dequeue commands need 20 cycles each to execute.

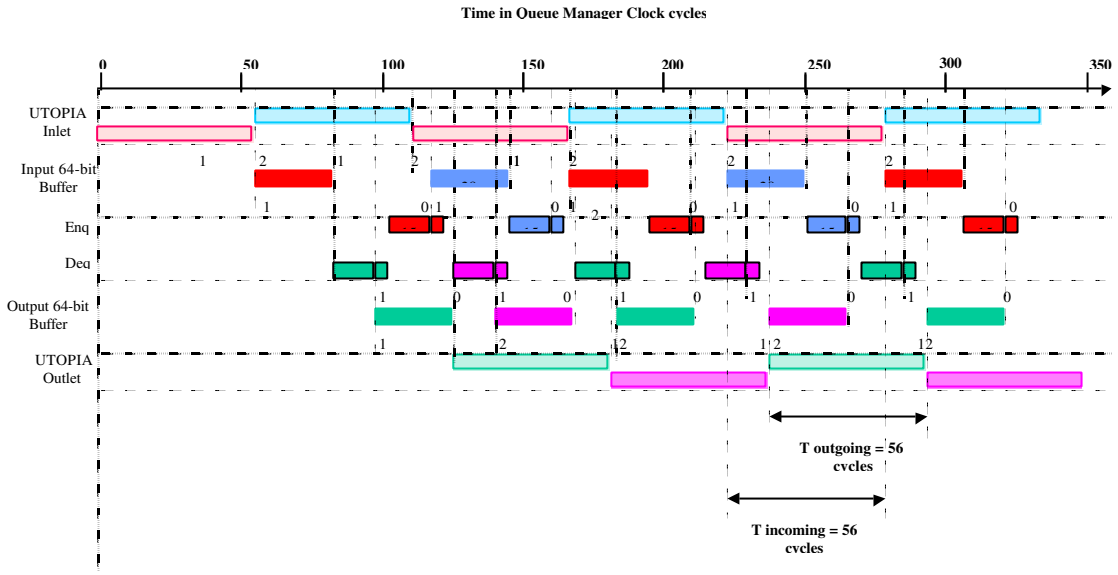

Figure 4-13: Normal case Enqueue-Dequeue timing diagram

The following parameters have been assumed on this diagram:

- **2 Cell FIFOs in the UTOPIA interfaces. (Inlet & outlet) :** The FIFOs inside the UTOPIA interfaces are assumed to be 2 cells long for simplicity. They are 8 cells long **- Single buffering on each Cell Buffer**
- **Speed-up = 2:** The ABRSU speed up is 2. This means that **QM_Clk = 2 * UTOPIA_Clk**
- **Full throttle Traffic**: Incoming cells arrive consecutively, Cell Scheduler requests outgoing cells consecutively.
- **Mixed Enq & Deq Commands**: Enqueue and dequeue commands are executed alternately.
- **20 cycle Enqueue and Dequeue Commands**: Each Enqueue and Dequeue Command takes 40 cycles to complete.
- **15 Cycles access of Input/Output 64-bit Buffers**: The Enqueue command uses its first 15 cycles to empty the Input 64-bit Buffer. The buffer can begin to fill immediately. The Dequeue command uses its first 15 cycles to fill the Output 64-bit Buffer. The buffer can begin to empty immediately.
- **56 QM_clk cycles for UTOPIA I/Fs**: A cell takes 56 QM clock cycles to enter since speed up is 2.

In the diagram, we can see that after 5 cell times the UTOPIA inlet is able to receive the cells consecutively and the UTOPIA outlet to send the cells consecutively. Cells need 56 cycles to enter or leave the ABRSU. This means that the 40 cycles needed by the Queue Manager to Enqueue and dequeue a cell is not the bottleneck of the system rather than the 56 cycles cell time of the UTOPIA interfaces.

4.4.4 Synthesis Results, Free-list bypassing and Cell Buffer Pre-allocation contribution

After synthesis of the Verilog HDL files, that described the ABRSU blocks with the MaxPlusII synthesis tool, the following results where received.

- **ABRSU (Queue Manager) Clock at 35 MHz.** This clock speed yields a combine incoming and outgoing throughput of 400Mbps for 40cc Enq, Deq commands and 800 Mbps for 20cc Enq, Deq commands.
- **FPGA SRAM Utilization at 95%**
- **FPGA Logic gate Utilization 55%**

Assuming 35MHz as the Queue Manager clock the ABRSU Speed up is 1,4 instead of 2, that was assumed in subsections 4.4.1 - 4.4.3.
In order for the Queue Manager not to be the bottleneck of the system it must be that:

1 Cell arrival time >= 1 Enq time + 1 Deq time =>
Tclk_utopia * 28 utopia_cycles >= Tclk_qm * (20+20) qm_cycles =>
Tclk_utopia / Tclk_qm >= 50 / 28 = 1,8 =>
40 ns/ 28ns >= 1,4 =>
1,4 >= 1,4

This means that the succeeded speed up barely satisfies the Bandwidth needs of the UTOPIA interfaces. If Free-list bypassing and Cell Buffer Pre-allocation where not implemented then the cycles needed for the normal case of enqueue and dequeue would be 20+20 + 10 cycles avoided by Free-list bypassing + 5 cycles avoided by Cell Buffer Pre-allocation = 55 cycles. In that case an ABRSU clock of ~50 MHz of frequency would be necessary.
The combined system performance improvement due to these 2 techniques is 15 cycles / 55 cycles = 27%.

5 Conclusions and Future Work

In this thesis, we studied the architecture of a Per-Flow Queue Manager for the purpose of queueing the ABR traffic of an ATM switch in times of congestion. The Queue Manager was implemented in a large FPGA that was placed on one of the Cards of the Switch. The use of FPGA allowed extensive on board testing of the design during its development and gave us the ability to confirm speed and feasibility assumptions early in the design face.
We used a single SDRAM DIMM memory module for storing of cells and cell pointers. This significantly reduced the pin and trace count of the physical design, yielding a low cost system. The careful scheduling of memory accesses to the SDRAM module by the Queue Manager proved that this single-buffer approach is feasible.
Although dynamic memory allocation increased the number of accesses for each enqueue or dequeue operation and lowered the buffering bandwidth, it allowed us, during testing, to use the Queue Manager for queueing thousands of cells of one flow and maintain the ability to handle the other 64K flows.
The interfacing of the Queue Manager with the external CPU allowed us to both debug the design effectively, insert test traffic that confirmed the correctness of the physical interface of the FPGA with the SDRAM DIMM and the Cubit Pros.
The use of the Free-List bypassing and Cell-Buffer preallocation techniques was proven essential in reaching the goal of near Gbps queueing bandwidth of our design. The 26% improvement of performance they induced with their use compensated for the loss of our clock speed goal (35 MHz instead of the 50 MHz clock that we targeted at the beginning of the design phase). Thus a maximum of 800 Mbps of combined incoming and outgoing ABR throughput was achieved that is sufficient for the ABR traffic needs of a Gbps Switch.

Adding various other switch features to our design is an interesting issue for future work. For instance, the enlargement of the Flow Record from 2 64-bit words to 4 words, could allow the addition of extra fields for each of the 64K supported flows. A New ID field could be added that would be accessible by the external CPU. This field would substitute the Header ID of incoming cells of a Flow. Thus the Queue Manager would offer VP/VC translation along with per-flow queueing.
Another field that could be added, is an Explicit Rate Field. The CPU would compute this appropriate explicit rate and use it to set the Explicit Rate Field inside the RM cells of a given flow, if it is smaller than the RM explicit rate. In that way, the RM explicit rate flow control could be supported, with the software taking care of the rate calculation and the hardware making the RM field enumeration.
Other interesting features that could be supported with extra fields are cell dropping mechanisms for ill-behaved flows.

6 References

[1] S. Keshav: "An Engineering Approach to Computer Networking", Addison-Wesley, 1997, ISBN 0-201-63442-2

[2] R. Lamaire, D. Serpanos: A 2-Dimensional Round-Robin Scheduling Mechanism for Switches with Multiple Input Queues , In IEEE/ACM Transactions on Networking, pages 471-482, 2(5), Oct 1994.

[3] Y. Oie, M. Murata, K. Kubota and H. Miyahara: «Effect of speedup in non-blocking packet switch,» Proc. ICC `89, Boston, MA, June 1989, pp. 410-414.

[4] J. G. Jim: «The throughput of data switches with and without speedup», Dai Schools of Industrial and Systems Engineering, and Mathematics Georgia Institute of Technology Balaji Prabhakar.

[5] A. Charny: «Providing QoS Guarantees in Input Buffered Crossbar Switches with Speedup», PhD Thesis, MIT, 1998.

[6] P. Prabhakar and N. McKeown: «On the speedup required for combined input- and output-queued switching», to appear in Automatica.

[7] G. Kornaros, C. Kozyrakis, P. Vatsolaki, M. Katevenis: "Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control", in Proc. ARVLSI'97 (17 th Conference on Advanced Research in VLSI), Univ. of Michigan at Ann Arbor, MI USA, Sept. 1997, IEEE Computer Soc. Press, ISBN 0-8186-7913-1, pp. 127-144.

[8] Ioannis Mavroidis: "Heap Management in Hardware", Technical Report 222, ICS-FORTH,July 1998

[9] A. Ioannou, M. Katevenis: "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High Speed Networks", Proc. IEEE Int. Conf. on Communications (ICC'2001), Helsinki, Finland, June 2001, pp. 2043-2047; http://archvlsi.ics.forth.gr/muqpro/heapMgt.html

[10] A. Nikologiannis, M. Katevenis: "Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques", Proc. IEEE Int. Conf. on Communications (ICC'2001), Helsinki, Finland, June 2001, pp. 2048-2052; http://archvlsi.ics.forth.gr/muqpro/queueMgt.html

[11] M. Katevenis, D. Serpanos, E. Markatos: "Multi-queue management and scheduling for improved QoS in communication networks", *Proceedings of EMMSEC'97* (European Multimedia Microprocessor Systems and Electronic Commerce Conference), Florence, Italy, Nov. 1997,pp. 906-913; http://archvlsi.ics.forth.gr/html papers/ EMM-SEC97/paper.html

[12] Tzi-cker Chiueh, Varadarajan, S.: "Design and evaluation of a DRAM-based shared memory ATM", 1997 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 97) Seattle, WA, USA 15-18, June 1997

[13] P. Andersson, C. Svensson (Lund Univ., Sweden):"A VLSI Architecture for an 80 Gb/s ATM Switch Core", IEEE Innovative Systems in Silicon Conference, Oct. 1996.

[14] V. Kumar, T. Lakshman, D. Stiliadis: "Beyond Best Effort: Router Architectures for the Differentiated Services of Tomorrow's Internet", IEEE Communications Magazine, May 1998, pp152-164.

[15] B. Suter, T.V. Lakshman, D. Stiliadis, A.K. Choudhury: "Buffer Management Schemes for Supporting TCP in Gigabit Routers with Per-Flow Queueing", IEEE Journal in Selected Areas in Communications, August 1999.

[16] http://www.altera.com
[17] http://www.motorola.com
[18] http://www.transwitch.com
[19] http://www.micron.com
[20] http://www.pmc-sierra.com
[21] ATM Forum: "Traffic Management Specification, Version 4.1", AF-TM-0121.000, March 1999.
[22] Ch. Lolas: "Design and Implementation of Low-Level software for high-speed packet switches", Master of Science Thesis, Computer Science Department, University of Crete, Greece, November 2001.
[23] G. Papadakis: "Design and Implementation in FPGA of an ABR Traffic Scheduler for an ATM Switch", Master of Science Thesis, Computer Science Department, University of Crete, Greece, November 2001.