A Logic-level Simulation of the ATMSWITCH

A thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at Massey University

Bo Yan

1998

Abstract

ATM networks are intended to provide a "one-size-fits-all" solution to a variety of data communication needs, from low speed, delay-insensitive to high-speed, delay-intolerant. The basic ATM protocol certainly delivers traffic within this broad range, but it does not address the quality of service requirements associated with the various type of traffic.

The ATMSWITCH is designed to use two different mechanisms to provide the quality of service for the various type of traffic. It treats the cells according to their connected virtual channel type and services them as predefined scheme.

The ATMSWITCH architecture is a shared-memory and output buffer strategy switch. The switch has been designed much of buffer location and identification can occur in parallel with the 12ns read/write cycle time required to buffer the cell data. The problem is essentially one of design circuitry so that buffer location and identification are as short as possible.

The present project has therefore been intended to measure the number of clock cycles required to perform the buffer maintenance activities, and to determine whether the logic speed required to fit this number of clock cycles into the 12ns window is feasible using current technology. The simulated result and timing analysis shows that 10 clock cycles are required during 12ns buffer read and write time, and a reasonable clock speed is 1.2ns per clock cycle.

Acknowledgements

I would like to thank my supervisor Mr. Paul Lyons for his guidance, encouragement and great help during my study.

Thanks also extend to all staff and postgraduate students in the Department of Computer Science for their assistance and friendship.

On the personal side, I would especially like to thank my wife, Wei Ma, and my daughter, Angela for their continued support and encouragement.

Table of Contents

Chapter 1

Introduction

In the past, separate telecommunications networks have been specially designed for different services. For instance, the public switched telephone network has been developed for conversational speech; data networks for computer communications; and broadcast networks for television. While these networks are very capable of supporting their intended services, they are generally not well suited for other services that are inherently different in such network requirements as bandwidth, holding times, end-to-end delays, and error rates.

The idea of a single ubiquitous network providing all services in an integrated manner has existed for some time. In a sense, this notion was suggested by AT&T's first president, Theodore Vail, in his vision of "one policy, one system, universal service" [Boettinger, 1983], and it has been recently restated [Mayo, 1985]. Implementation has been hindered by the lack of the necessary technology and public demand from multimedia applications, which involve the processing and exchange of information in the various media of text, audio, and images. In addition, research in high-speed switching, fibre optics, and new protocols have demonstrated the technical feasibility of integrated services networks.

Research in the past few years has led to the concept of ATM as a possible means to realising an integrated services network. The ATM will represent a dramatic change in the evolution of the public switched telephone network. It combines characteristics of both conventional packet switching and circuit switching. Not only does ATM imply a major change in the network facilities, but for the first time, the network will be designed to provide much more than POTS (Plain Old Telephone Services).

In order to explain how the ATM approach combines both circuit and packet switching techniques to provide all services in an integrated manner. I will first introduce public telephone network and packet-switched data networks. The second part of this chapter will introduces ISDN (Integrated Services Digital Network) and ATM based public switched network that derived from ISDN and promote to B-ISDN. The later part of this chapter will introduce ATMSWITCH that was simulated in this research project.

1.1. Circuit Switching

In traditional analogue circuit switching, a call is set up on the basis that it receives a path (from source to destination) that is its "property" for the duration of the call. I.e. the whole of the bandwidth of the circuit is available to the calling parties for the whole of the call. In a digital circuit-switched system, the whole bit-rate of the line is assigned to a call for only a single time slot per frame. This is called time division multiplexing.

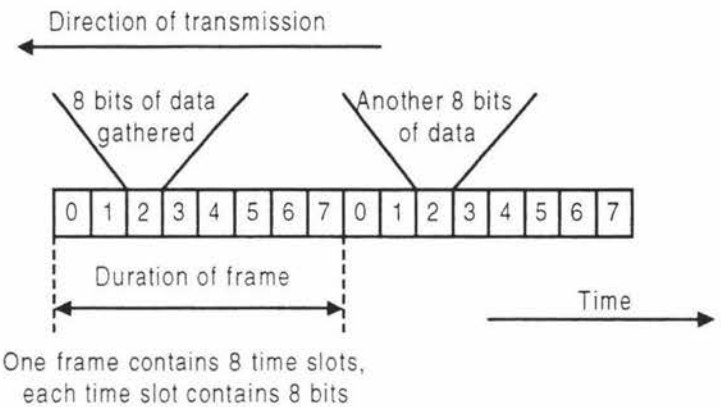


Figure 1.1: An example of time division, position, multiplexing

During the time period of a frame, the transmitting party will generate a fixed number of bits of digital data (for example, eight bits to represent the level of an analogue telephony signal) and these bits will be grouped together in the time slot assigned in every frame for the duration of the call, to that call (Figure 1.1). So the time slot is identified by its position in the frame, hence use of the name "position multiplexing", although this term is not used as much as time division multiplexing.

When a connection is set up, a route is found through the network and that route remains fixed for the duration of the connection. The route will probably traverse a number of switching nodes and require the use of a similar number of transmission links to provide a circuit from source to destination. The time slot position used by a call is likely to be different on each link. The switches which interconnect the transmission links perform the time slot interchange (as well as the space switching) necessary to provide the

"through-connection" (e.g. link M, time slot 2 switches to link N, time slot 6 in Figure 1.2).

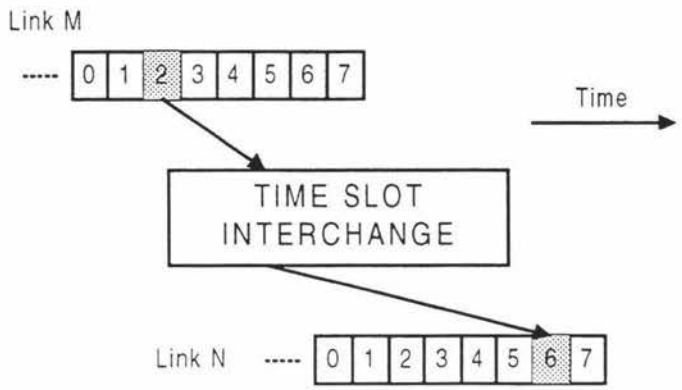


Figure 1.2 Time slot interchange

In digital circuit switched telephone networks, frames have a repetition rate of 8000 frames per second (and so a duration of 125 μs), and as there are always eight bits (one byte) per time slot, each channel has a bit-rate of 64 Kbit/s. With N time slots in each frame, the bit-rate of the line is N×64 Kbit/s. In practice, extra time slots or bits are added for control and synchronisation functions. So for example, the widely used 30 channel system has two extra time slots, giving a total of 32 time slots, and thus a bit-rate of (30+2)×64 = 2048 Kbit/s.

The time division multiplexing concept can be applied recursively by considering a 24 or 30 channel system as a single "channel", each frame of which occupies one time slot per frame of a higher order multiplexing system. This is the underlying principle in the SDH (Synchronous Digital Hierarchy), and an introduction to SDH can be found in [Griffiths, 1992].

The main performance issue for the user of a circuit switched network is whether, when a call is requested, there is a circuit available to the required destination. Once a circuit is established, the user has available a constant bit-rate with a fixed end-to-end delay. There is no error detection or correction provided by the network on the circuit – that is the responsibility of the terminals at either end, if it is required. Nor is there any per circuit overhead -- the whole bit-rate of the circuit is available for user information.

1.2. Packet switching

To see how ATM has evolved from both circuit switched and packet switched networks, it is helpful to consider a "generic" packet switching network. I.e. one intended to represent the main characteristics of packet switching, rather than any particular packet switching system.

Instead of being organized into eight-bit time slots which repeat at regular intervals, data in a packet switched network is organized into packets comprising many bytes of user data (bytes are also known as octets, in order to divorce them from any association with an eight-bit coding scheme). Packets can vary in size depending on how much data there is to send, usually up to some predetermined limit (for example, 4096 octets). Each packet is then sent from switching node to switching node as group of contiguous bits fully occupying the link bit-rate for the duration of the packet. If there is no packet to send, then nothing is sent on the link. When a packet is ready, and the link is idle, then the packet can be sent immediately. If the link is busy (another packet is currently being transmitted), then the packet must wait in a buffer until the previous one has completed transmission (Figure 1.3).

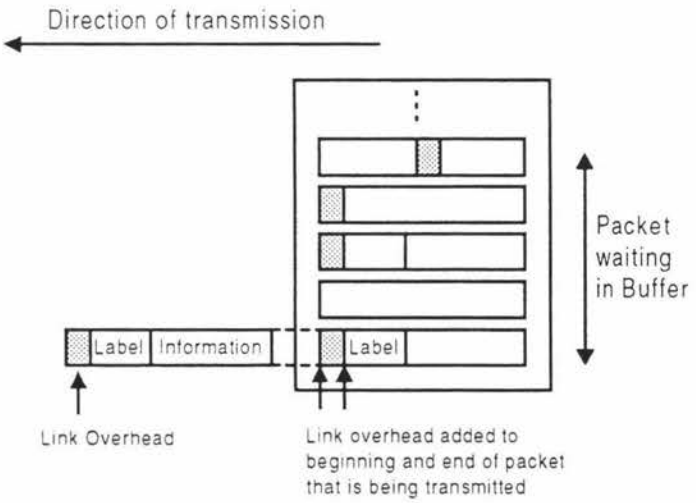


Figure 1.3: An example of Packet Switching

Each packet has a label to identify it as belonging to a particular communication (The word "communication" has the same notion as the word "call" in the circuit switching. The "communication" is usually used in the data communication term.). Thus packets from different sources and to different destinations can be multiplexed over the same link by being transmitted one after the other. This is also called label multiplexing [Pitts,

1996]. The label is used at each node to select an outgoing link, routing the packet across the network. The outgoing link selected may be predetermined at the set-up of the connection, or it may be varied according to traffic conditions (e.g. take the least busy route). The former method ensures that packets arrive in the order in which they were sent, whereas the latter method requires the destination to be able to resequence out-of-order packets (in the event that the delays on alternative routes are different).

Whichever routing method is used, the packets destined for a particular link must be queued in the node prior to transmission. It is this queuing which introduces variable delay to the packets. A system of acknowledgment ensures that corrupted packets are not lost but are retransmitted. This is done on a link-by-link basis, rather than end-to-end, and contributes further to the variation in delay. There is quite a significant per packet overhead required for the error control and acknowledgment mechanisms, in addition to the label. This overhead reduces the effective bit-rate available for the transfer of user information. The packet plus link overhead is often (confusingly) called a "frame". Note that it is not the same as a frame in circuit switching.

A simple packet-switched network may continue to accept packets without assessing whether it can cope with the extra traffic or not. Thus it appears to be non-blocking, in contrast to a circuit switched network which rejects (blocks) a connection request if there is no circuit available. The effect of this non-blocking operation is that packets experience greater and greater delay across the network, as the load on the network increases. As the load approaches the network capacity, the node buffers become full, and further incoming packets cannot be stored. This triggers retransmission of those packets which only worsens the situation by increasing the load; the successful throughput of packets decreases significantly.

In order to maintain throughput, congestion control techniques, particularly flow control, are used. Their aim is to limit the rate at which sources offer packets to the network. The flow control can be exercised on a link-by-link, or end-to-end basis. Thus a connection cannot be guaranteed any particular bit-rate: it is allowed to send packets to the network as and when it needs to, but if the network is congested then the network exerts control by restricting this rate of flow.

The main performance issues for a user of a packet switched network are the delay experienced on any connection and the throughput. The network operator aims to maximize throughput and limit the delay, even in the presence of congestion. Once a connection is established, the user is able to send information on demand. The network

provides error control through re-transmission of packets on a link-by-link basis. Capacity is not dedicated to the connection, but shared on a dynamic basis with other connections. The capacity available to the user is reduced by the per packet overheads required for label multiplexing, flow and error control.

1.3. Statistical Multiplexing and The Jitter Problem

Packet switching is also called statistical multiplexing. ATM is actually based upon a statistical multiplexing technique called cell relay switching [Clark, 1996]. Statistical multiplexing is a means of multiplying the effective capacity of a transmission line or network, by taking advantage of variations in traffic intensity associated with different calls.

For example, when speech connections are statistically multiplexed, the silent periods can be suppressed and not sent over the line. Meanwhile the words from other conversations can be carried in the gaps.

The major benefit of statistical multiplexing is that the useful carrying capacity of the line is maximized by avoiding the unnecessary transmission of redundant information (i.e. pauses). The first practical realizations of statistical multiplexing were data networking protocols. In particular, statistical multiplexing forms the basis of data packet switching. It is the principle upon which IBM's SNA (systems network architecture) and ITU-T's X.25 recommendation are based. As such, statistical multiplexing is widely in use within computer data networks.

Today's public voice networks, in contrast to data networks, have not used statistical multiplexing. Instead, voice and telephone networks have historically been based upon circuit switching, the allocation of a path across the network on a fully dedicated point-to-point basis for the duration of the call. The strength of circuit switching is the guaranteed throughput and delay performance of the resulting connection. This is critical in order that acceptable voice quality can be achieved (in the subjective opinion of telephone users). A telephone call connected in a circuit-switched manner is like an empty pipe between two telephone users. Whatever one speaker says into the pipe comes out at the other end in an identical format -- but only one pair of callers can use the pipe during any particular call.

Historically, telephone networks have not used statistical multiplexing techniques because of the problem of achieving acceptable speech quality. There were attempts to 'packet switch' voice across data networks, but the problem was that individual words or parts of words take different times to propagate through a packet network, so that the listener hears a rather broken form of the original signal. The effect is caused by a phenomenon called **jitter**. The more jitter (variable propagation delay) experienced by a telephone connection, the worse the perceived quality of the connection.

Jitter in data networks is relatively unimportant. So long as the average delay is not great, computer users do not notice whether some typed characters appear imperceptibly faster or slower than others. As a result, telephone systems have remained circuit switch based, because of the quality problems. The consequence has been the evolution of two entirely separate networking worlds -- voice and data. Transmission lines cannot easily and efficiently be shared between voice and data, and dynamic allocation of bandwidth -- one instant to voice, the next moment to data -- has not been possible.

1.4. ISDN and ATM

A number of attempts have been made to develop technologies capable of handling equally well both voice and data over the same network. The two most noteworthy technologies in this category are ISDN (integrated services digital network) and, now, ATM (Asynchronous Transfer Mode).

ISDN is a technology based upon circuit switching within digital telephone networks. The digital nature of the network is exploited for the use of data transmission. And to make integrated data and voice carriage possible, the signaling within the network (between the exchanges in the network and from the calling handset to the first exchange) is very advanced -- far superior to the simple pulsing technique used in order analogue telephone networks. Unfortunately, ISDN as a data transport medium is limited in its efficiency and flexibility due to its circuit switched nature.

While ISDN will revamp customer expectations of telephone services (with new features like caller identification before answer and ring back when free), ISDN in its basic form (narrowband ISDN) is unlikely to form the basis of advanced integrated voice and data networks.

In contrast, because ATM (which is a form of so-called broadband ISDN or B-ISDN) has evolved from the statistical multiplexing technique inherent in packet switched data networks, it is likely to have more success as an integrated voice, data and video transport medium. The developers of ATM have simply concentrated on improving the packet switching technique to limit the jitter on speech, video and other delay sensitive applications. The resulting technique is called cell relay switching, or simply cell relay [Clark, 1996].

1.5. The Problems to be Solved by Cell Relay

The normal statistical multiplexing of data connections is carried out by packet switching. Packets of data are created by each of the sources, and interleaved as appropriate by the statistical multiplexor, as illustrated in Figure 1.4:

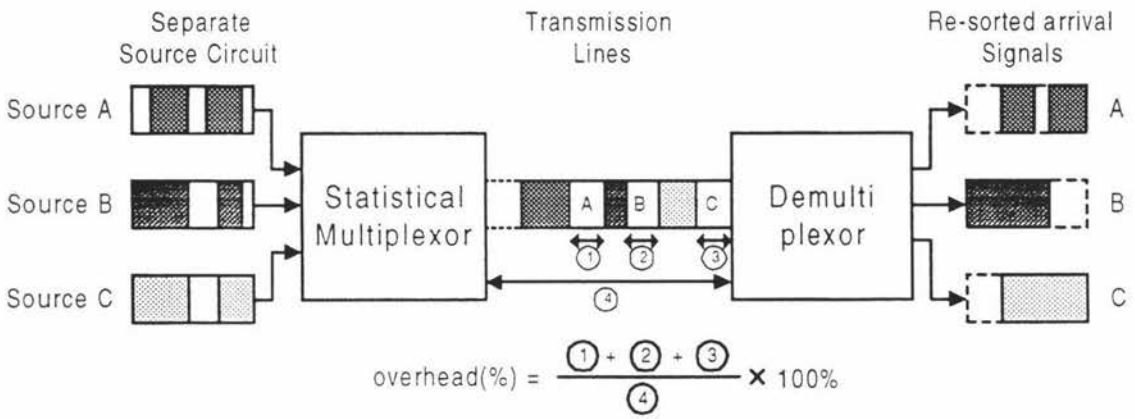


Figure 1.4: Statistical multiplexing headers and overhead

The interleaving is usually carried out on a simple FIFO queue basis (first in -- first out). Packets received from the sending sources are stored at the back of the queue. Meanwhile, packets at the front of the queue are being transmitted along the link.

A typical data packet contains between 1 and 256 characters (between 8 and 2048 bits), and the line speed is typically 9600 bit/s. The propagation delay at a time when two sources try to send simultaneously (due to the extra waiting time) may therefore be up to 200 ms (2048/9600 s) longer than when there is no simultaneous transmission. In other words, there may be up to 200 ms of jitter. This is unacceptable for speech transmission. But before discussing how cell relay circumvents this problem, we should cover one

other important aspect of statistical multiplexing -- an aspect which constrains the maximum achievable line usage efficiency.

In order to allow the demultiplexor to sort out the various packets belonging to the different logical connections, and forward them to the correct destinations (A to A, B to B, C to C etc.) there needs to be a label attached to each packet to say to which logical connection (i.e. telephone conversation or data communications session) it belongs. This label is contained in the header, which is an addition to the front of the packet and has a function like the envelope of a letter. The header (Figure 1.4) is crucial to the technique of statistical multiplexing, but has the disadvantage that it adds to the information which must be carried by the transmission line between multiplexor and demultiplexor. At the demultiplexor, the header is removed so it does not disturb the receiver, but meanwhile it has generated an overhead load for the transmission line. It is thus impossible using statistical multiplexing techniques to archive 100 per cent loading of a transmission line with raw user information. Some of the capacity has to be given up to carry the overhead.

The major challenges for ATM developers have therefore been to minimize the jitter experienced by speech, video, and other delay-sensitive applications while simultaneously optimizing line efficiency by minimizing network overhead. As we shall see, these demands contend with one another.

1.6. The Technique of Cell Relay

Cell relay is a form of statistical multiplexing similar in many ways to packet switching, except that the packets are instead called cells. Each of the cells is of a fixed rather than a variable size.

The fixed cell size defined by ATM standards is 48 octets (bytes) plus a 5 octets header (i.e. 53 octets in all -- see Figure 1.5). The transmission line speeds currently foreseen to be used are either 155, 622 or 1200 Mbit/s. We can therefore immediately draw certain conclusions about ATM performance:

- the overhead is at least 5 bytes in 53 bytes, i.e. > 9 per cent;
- the duration of a cell is at most 53 × 8 bits/155 Mbit/s = 2.74μs. (0.360μs at 1200 Mbit/s).

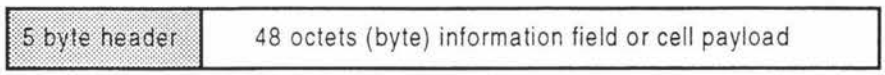


Figure 1.5: ATM 53-byte cell format

Since the cell duration is relatively short, provided a priority scheme is applied to allow cells from delay-sensitive signal sources (e.g. speech, video, etc.) to have access to the next cell slot, then the **jitter** (variation in signal propagation delay) can be kept very low -- not zero as is possible with circuit switching, but at least low enough to give a subjectively acceptable quality for telephone listeners or video watchers. Jitter-insensitive traffic sources (e.g. data communication channels) can be made to wait for the allocation of the next free or low priority slot.

Although the cell size and priority scheme can solve the jitter problem in principle, the jitter still could occur when the cell pass the not well-designed switch. So the architecture of an ATM switch is also a key to ensure the quality of time sensitive channel transmission.

1.7. ATM Switch

The cell header carries information sufficient to allow the ATM network to determine to which connection (and thus to which destination port and end-user) each cell should be delivered. We could draw a comparison with a postal service and imagine each of the cells to be a letter with 48 characters of information contained in an envelope on which a 5-digit postcode appears. You simply drop your letters (cells) in the right order and they come out in the same order at the other end, though maybe slightly jittered in time. Just as a postal service has numerous vans, lorries and personnel to carry different letters over different stretches, and sorting offices to direct the letters along their individual paths, so an ATM network can comprise a mesh of transmission links and switches to direct individual cells by inspecting the address contained in the header (Figure 1.6).

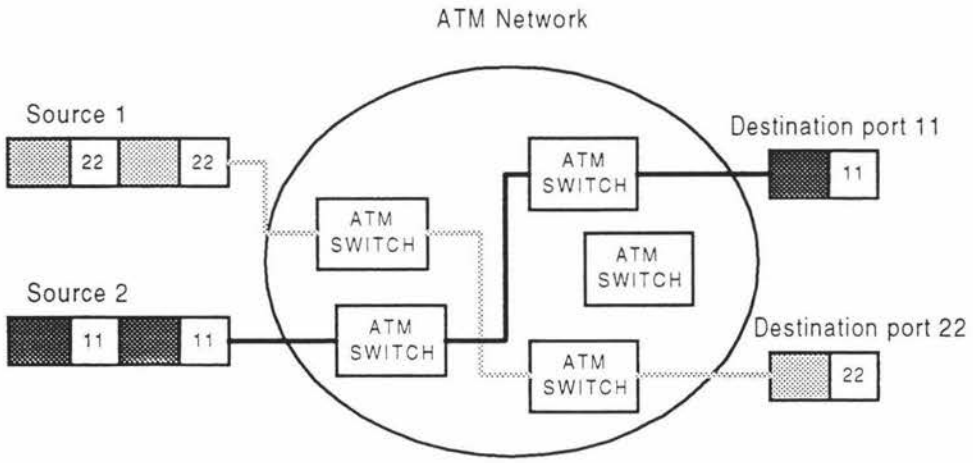


Figure 1.6: Switching in an ATM network

The ATM "switch" acts in much the same way as a postal sorter. On its incoming side is a FIFO buffer, like a pile of letters. At the front of the buffer (like the top letter in the pile) is the cell which has been waiting longest to be switched. New cells arriving are added to the back of the buffer. The switching process involves looking at each cell in turn, and determining from the address held in the header which outgoing line should be taken. The cell is then added to the FIFO output buffer which is queuing cells waiting to be transmitted on this line. The cell then proceeds to the next exchange.

ATM will provide flexibility in bandwidth allocation and will allow a network to carry heterogeneous services ranging from narrowband to wideband services requiring real time. However, the challenge is to build fast ATM switches able to match the high speeds of the input links and the high performance requirements imposed.

In 1990, Tobagi described a large number of switching architectures [Tobagi, 1990]. All the approaches point to the need of a very high speed hardware switch because of the high transfer rates involved; on the other hand, due to the statistical multiplexing, buffering is also required in order to avoid cell loss whenever there are multiple input cells arriving simultaneously on different input ports and destined for the same output. Only one cell at a time can be transmitted over an output link; the rest must be temporarily stored in a buffer for later transmission.

An attractive ATM architecture is using shared memory [Hluchyj, 1988] to implement output buffering and therefore to attain the best throughput /delay/cell loss performance.

In shared-memory type switches that operate without blocking, all input and output ports have access to a shared-memory module able to write up to N incoming cells and to read out N outgoing cells in a switching time cycle, so that, as in output-buffered switches, throughput is not reduced by output port contention, and an optimal throughput/delay performance is achieved. But in this architecture, we still need to consider how to design the buffer scheme, how to make jitter low enough when the voice or video cells pass through the switch.

1.8. ATMSWITCH Architecture

ATM networks are intended to provide a "one-size-fits-all" solution to a variety of data communication needs, from low speed, delay-insensitive to high-speed, delay-intolerant. The basic ATM protocol certainly delivers traffic within this broad range, but it does not address the quality of service requirements associated with the various type of traffic. In particular, its statistical multiplexing nature is inherently antagonistic towards constant-delay traffic such as real-time video.

The ATMSWITCH is proposed by Lyons et al in (1996, 1997). The ATMSWITCH uses two different mechanisms to provide the quality of service for the various type of traffic. It treats the cells according to their connected virtual channel type and service them as predefined scheme. The ATMSWITCH ensures that the jitter can be minimized when the CBR/VBR channel's cell transmission in the ATM network.

The ATMSWITCH architecture is a shared-memory and output buffer strategy switch. On the size the limitations of this type of switch come from the memory control logic (which must be able to handle N incoming and N outgoing cells in each time slot), and the memory bandwidth that must be at least the sum of the bandwidths of the incoming and the outgoing lines. The memory bandwidth depends on the word length, which in turn is limited to the cell size (53 octets = 424b). Therefore, for a given memory cycle time (or memory access time) the number of links N is defined by the following relation (for single ported memories):

$$N = \frac{celllength(b)}{2 \times cycletime \times linkspeed(b/s)}$$

Thus a switch incorporating 12ns memory can theoretically support up to 14 1.2 Gbps links. However, this theoretical result does not allow for the other operations which the

switch must perform each time a cell is served - circuit identification, insertion into the buffer, and subsequent extraction from the buffer and assembly into a new cell. Because the buffer structures are comparatively complex in the ATMSWITCH architecture, these operations may consume a significant amount of time. It is therefore important to pack them into the minimum amount of time so that the maximum possible throughput can be achieved. The architecture of the ATMSWITCH has been designed so that much of this administrative business can occur in parallel with the 12ns read/write cycle time required to buffer the cell data. The problem is essentially one of design circuitry so that buffer location and identification and cell assembly are as short as possible and so that the time required for channel identification + buffer insertion is no greater than 12ns, and so that the time required for buffer extraction and cell assembly is no greater than 12ns. This is illustrated in Figure 1.7.

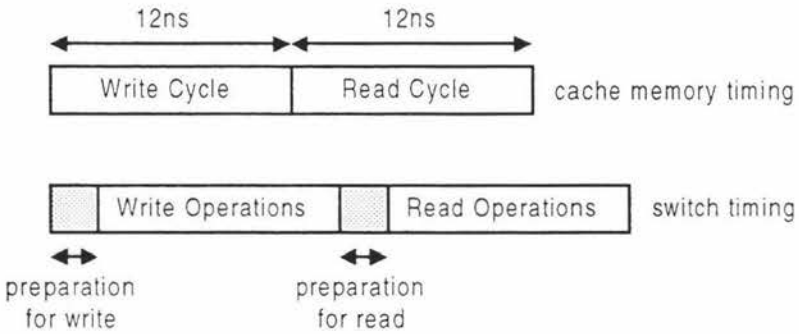


Figure 1.7: Cache Memory and Switch Timing

The buffer location and identification, and cell assembly operations are minimal complexity - they can be completed in a single clock cycle. Maintenance of the buffers, on the other hand, is quite complex and it was therefore important to minimize the amount of time involved.

The present project has therefore been intended to measure the number of clock cycles required to perform the buffer maintenance activities, and to determine whether the logic speed required to fit this number of clock cycles into the 12ns window is feasible using current technology.

Previous work on the ATMSWITCH has focused on a high-level aspects of the architecture. A detailed logic-level design was not available at the start of this project, and consequently, a large part of the project was the construction of a VHDL (Very High

Speed Integrated Circuit Hardware Description Language) definition of the circuit, so that the number of clock cycles involved could be accurately determined.

An interesting spin-off from the project has been the design of an improved architecture using interleaved memory, which has the potential to improve the speed of the switch still further.

Chapter 2

ATM Protocol Stack

2.1. ATM Concept [Pitts, 1996]

In ATM, asynchronous does not refer to physical transmission, which will in fact be synchronous in B-ISDN (e.g., SONET/SDH). Asynchronous refers to the manner in which bandwidth is allocated among connections and users. Bandwidth is divided into time slots of fixed length. These time slots are allocated for user information as needed and therefore do not have predetermined temporal positions (within a periodic frame, for example). Instead of identified with explicit prefix labels. Transfer mode is a term intended to signify that it is a multiplexing and switching technique.

The ATM concept is defined by a number of principles:

- All information is carried in the form of fixed-length data units called cells, which consists of a herder and an information field (sometimes called payload).
- ATM is connection-oriented, and cells in the same virtual connection maintain their sequential order.
- Traffic sources may generate cells as needed-i.e., without predetermined temporal positions-and therefore cells have explicit labels (a field in the header) for connection identification.
- The main function of the cell header is identification of cells belonging to the same virtual connection.
- The identifier labels have only local significance (they are not explicit address) and are translated at each switch.
- The information fields are carried transparently; for instance, no error control is performed on the information fields.
- Cell streams are asynchronous time-division multiplexed.

As a multiplexing technique, ATM is potentially capable of more efficient utilization of transmission facilities compared to synchronous TDM. In synchronous TDM, a periodic frame consisting of short time slots (usually byte size) is defined on a transmission link, and connections sharing the link are reserved fixed time slot positions in each frame. It is apparent that bandwidth will be wasted if the traffic is burst and contains idle periods.

This inefficiency can be prevented in asynchronous TDM where time slots are allotted to connections as needed. However, a prefix label is needed for each time slot to identify the connection, As a consequence, the time slots are larger than bytes so that the labels consume a smaller fraction of the total bandwidth. Also, processing is required for each time slot, and buffering is required to resolve contention. ATM is an example of this labeled asynchronous TDM technique in that identifier labels in the cell header have only local, and not end-to-end, significance. As a multiplexing technique, ATM is motivated by the potential for efficient utilization.

As a switching technique compared to STM (or multirate circuit switching), the main advantage of ATM is that it avoids the necessity of peak rate allocation as STM. Another difference is that ATM involves network processing of cells, whereas STM carries traffic transparently through the network. This is a disadvantage in terms of processing burden but allows greater network control over routing, error control, flow control, copying, and priorities. Considering priorities, for example, each cell can be assigned delay and loss priorities. By means of priorities, the network can exercise control over the preferential treatment of one traffic class relative to another class at the levels of virtual connections or individual cells. Thus, as a switching technique, ATM may be motivated by the capability for granular and flexible control of network traffic.

There are basically two undesirable consequences of the asynchronous nature of ATM. Since network resources are not reserved, congestion is possible when an excessive amount of traffic contends for limited buffer resources, making it necessary to lose cells. Another consequence is variable cell delays through the network caused mainly by random queuing delays at each switch and multiplexer. Hence the concept of QoS (Quality of Service), means that B-ISDN will provide multiple QoS classes of virtual connections to support a range of traffic types with different QoS requirements. Support of multiple traffic classes and protection of their QoS, which simultaneously maximising statistical sharing and utilization of network resources, is the most difficult challenge in the implementation of ATM.

2.2. The Protocol Reference Model

Just as the OSI layered protocol model describes communication between two computers over a network, according to the B-ISDN reference model, [Prycker, 1993], the ATM protocol model describes how two end systems communicate via ATM switches. As

shown in Figure 2.1 the key layers that need explanation are the ATM Adaptation, ATM, and Physical layers.

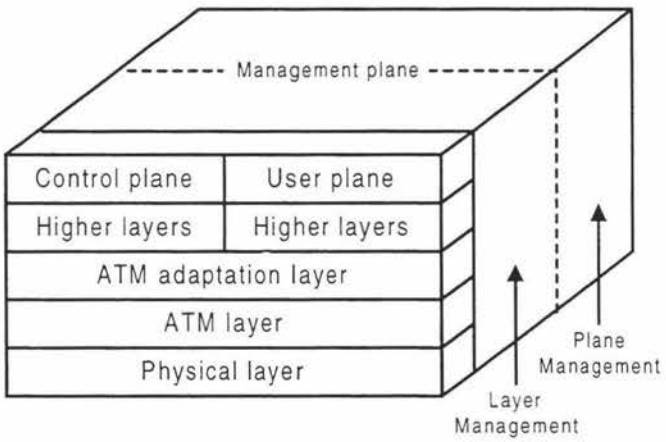


Figure 2.1 The ATM protocol reference model

The ATM Adaptation layer (AAL). A user transmits a data packet designed for another user. The ATM adaptation layer provides services to the higher layers that support classes of service for transported data. It major concern, through, is the segmentation and reassembly of data. It takes this data and splits it up to multiple 48-byte cells.

The ATM layer. The ATM layer is responsible for providing the appropriate routing information for cells in the form of VPI/VCI values, which are part of the control information found in a cell's five-byte header. The VPI/VCI values (local to a specific switch) ensure that the cell will exit the correct switch output port. The ATM layer is also responsible for ensuring that cells are ordered (stay in the correct order). This layer passes control over the cells to the physical layer at a switch's output port.

The physical layer. The physical layer is primarily responsible for transmitting and receiving data. Then it groups these cells in payload envelopes, adds routing information, and transmits them. The process is reversed at the other end of the network when the destination node begins the process of translating the cells back into data.

The part of the layered architecture used for end-to-end or user-to-user data transfer is known as U plane (User plane). The control plane defines higher-level protocols used to support ATM signaling, and the M plane (Management plane) provides control of an ATM node and consists of two parts: plane management and layer management. The

plane management function manages all other planes and the layer management function is responsible for managing each of the ATM layers.

2.2.1. The physical Layer

The physical layer defines the physical interfaces and framing protocols associated with ATM. This layer is segmented into two sublayers: the TC (Transmission Convergence) sublayer and the PMD (Physical Medium Dependent) sublayer. The reason for sublayers in this ATM architecture is to decouple the transmission from the physical medium to permit a variety of physical media.

The TC concerns itself with adaptation to the transmission system, defined as the reception of cells from the ATM layer and their packaging into appropriate format for transmission over the PMD. The TC also handles cell delineation is the extraction of cells from the bit stream received from the PMD. The function of cell-rate decoupling is to insert/suppress idle cells to or from the payload in order to provide a continuous flow of cells. Finally, the TC generate and verifies the HEC (Header Error Check). It calculates the HEC from the bits received and checks it against the HEC value of the received cell. If there's a match on consecutive cells, then the TC assumes correct cell boundaries. If there's no match for many successive cells, the TC knows that the correct cell delineation isn't yet found.

2.2.2. The ATM Layer and ATM cells

The ATM layer performs four basic functions. It multiplexes and demultiplexes cells of different connections. Multiplexing refers to the process of talking several different data streams and consolidating them into a fast-flow data stream. At the other end of the communication path, demultiplexing reverses the process and directs the data back to its appropriate data stream and towards its ultimate destination. These connections are identified by their VCI (Virtual Channel Identifier) and VPI (Virtual Path Identifier) values. The ATM layer also translates VCI and /or VPI values at the switches or cross connections, if required. It's responsible for extracting/inserting the header before or after the cell is delivered to or from the ATM adaptation layer. Finally, this layer implements a flow control mechanism at the UNI (User-Network Interface) by using the GFC (General Flow Control) bits of the header.

2.2.3. The ATM Cell Header

According to the paper [Schatt, 1993], the ATM cell header fields is shown in Figure 2.2 The four-bit GFC fiels is used only across the UNI to control traffic flow and prevent overload conditions. This fields isn't defined across the NNI (Network-Network Interface), and the corresponding bits are used for an expanded VPI field.

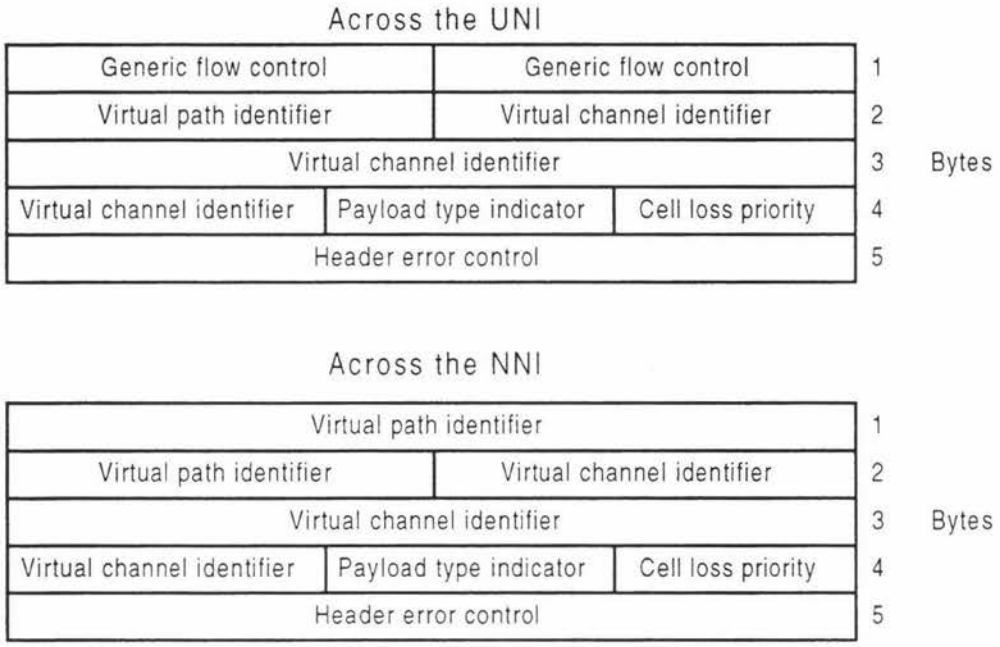


Figure 2.2: ATM cell header formats across the UNI and NNI

Two of the fields in the cell header will be discussed here. The virtual path identifier field is used to identify virtual paths. Consisting of eight bits across the UNI and 12 bits across the NNI, the field isn't defined by either the CCITT or ATM Forums. The virtual channel identifier field is 16 bits long. End devices assign a value to the VPI and VCI fields when requesting a connection to an end system.

The PTI (Payload Type Identification) field consists of three bits and is used to identify the payload type carried in the cell, as well as to identify control procedures. The ATM Forum designates the setting of one bit to indicate congestion, a second bit for network management, and a third bit to indicate an error condition.

The CLP (Cell Loss Priority) field is a single bit that indicates a cell's loss of priority. This bit is set to 1 when a cell can be discarded due to congestion; if a switch experiences congestion, it will drop cells with this bit set. This results in giving priority to certain types of cells carrying certain types of traffic, such as video in congested networks.

The HEC (Header Error Check) is an eight-bit cyclic redundancy code that's calculated over all fields in the ATM header. This type of error checking can is very important in ATM operations because an error in the VPI/VCI could corrupt the data flow of their circuits.

2.2.4. General ATM operation

ATM requires that a connection be made between two end points before information can be exchanged. An end point on a network sends a signal across the UNI to the network requesting a connection to another point. The network sends this request to its destination point, where it's interpreted. If this node accepts the request for a connection, a virtual channel is established across the network. Two end or switching points can be lined via a virtual channel link. The VPI and VCI fields of the ATM cell header contain the routing information that's required.

Figure 2.3 illustrates how this process works. End point A requests connection to point B. The ATM network assigns value P to the VCI of A, and value Q to point B. Node A will use P for outgoing information, and B will use Q for incoming information. Lookup tables are set up throughout the network. The same process is followed for the reverse direction. The first switching node that receives a cell from A will consult its lookup table to find out where the cell should be switched to and what value the outgoing VCI should be assigned to. This process is repeated until it arrives at B.

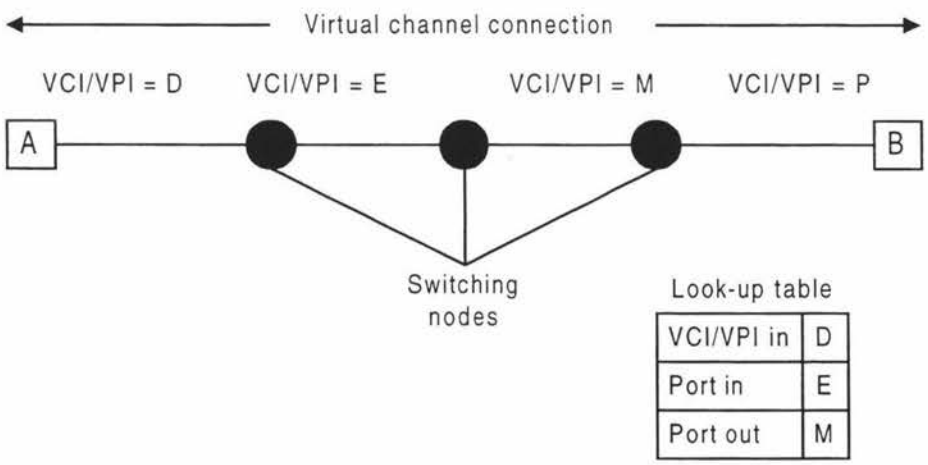


Figure 2.3: An ATM virtual channel connection.

How does the ATM layer function when the ATM node is an end system? The AAL layer provides it with information. When the ATM layer exchanges a cell stream with the physical layer, it inserts this information as well as the required parameters in the header fields, including the crucial VPI/VCI values. If it has no information to transmit, it fills the information field with idle cells. The ATM layer is also responsible for controlling the quality of service for each circuit, a value that's negotiated when circuits are established. Among the parameters that are negotiated are the peak and average data rates, the acceptable delay, and the loss rate.

The ATM layer's operation is even less complicated for a switch. The ATM layer under this circumstance receives an ATM cell on one port and uses the VPI/VCI value to determine to which port to forward the cell. It then forwards the cell to the appropriate port, changes the VPI/VCI to reflect the cell's routing, and transmits the cell to the physical layer of that port.

2.2.5. The Four Classes of ATM Traffic

A major advantage of ATM network is its ability to handle a variety of different types of traffic. According to the paper [Schatt, 1993], current specifications define four different classes of traffic listed in Table 2.1.

Table 2.1 Classes of ATM Traffic

Class of service	Old class of Service	Description
CBR	Class A	Constant bit rate (CBR), connection-oriented, synchronous traffic(uncompressed voice or video); supports peak cell rate traffic
VBR-RT	Class B	Variable bit rate (VBR) real-time, connection-oriented, synchronous traffic(live video transmissions); supports peak cell-rate traffic, sustained cell-rate traffic, and maximum burst-size traffic
VBR-NRT	Class B	Variable bit rate non-real-time traffic (video playback, multimedia); supports peak cell-rate traffic, sustained cell-rate traffic, and maximum burst-size traffic

ABR	C lass C	Variable bit rate, connection-oriented, asynchronous traffic (wide area X.25, frame relay over ATM); supports peak cell-rate traffic and maximum burst-size traffic
UBR	Class D	Connectionless packet data (Lan traffic, wide area SMDS traffic, etc.)

Under CBR, a network application establishes a CBR connection and negotiates what's known as a PCR (Peak Cell Rate), the maximum data rate the connection will support without losing data. Traffic is then transmitted at that rate. The ideal type of traffic for CBR service is real-time voice and video traffic since both require constant data streams and cannot tolerate lost data.

VBR (Variable Bit Rate) service requires negotiation not only for the PCR but also the SCR (Sustained Cell Rate), which refers to the average throughput rate the application is permitted. Part of the VBR negotiation actually consists of determining how long transmission will stay at the PCR rate (known as the burst tolerance). In other words, under VBR the traffic can soar as the average rate by dropping traffic flow to a lower rate for the time needed to achieve the SCR. VBR users have a guaranteed quality of service regarding cell loss and bandwidth availability as long as the traffic meets the criteria negotiated.

ABR (Available Bit Rate) addresses many of the concerns that have troubled network managers about VBR and provides reliable delivery of burst traffic. It uses excess bandwidth and network management algorithms to evaluate network congestion and eliminate cell loss. ABR provides a guaranteed quality of service concerning bandwidth availability and cell loss. It does not guarantee against cell delay, so non-real time LAN applications are ideal for ABR.

UBR (Unspecified Bit Rate) lends new meaning to the term service. It has no specified bit rate and no quality-of-service guarantees. In fact, the only assurance is that UBR will make its best effort to deliver cells. There's no flow control, so if traffic becomes very heavy then cells will be lost once buffers are full. Think of this service as what you'd expect when you send a letter to someone in China. It might get there, but then again it might not.

2.3. ATM Networking

According to the paper [Pitts, 1996], an ATM network may be considered a collection of ATM network elements (switches, crossconnects, multiplexers, concentrators) interconnected by transmission facilities which together provide the information transport functions of the ATM layer and physical layer. In addition, the ATM network supports the relevant call control and network management functions.

2.3.1. ATM-Layer Cell Transport

The primary responsibility of an ATM network is the cell transport functions of the ATM layer. The ATM layer receives 48-bytes data units from the AAL or ATM-Layer management. These are encapsulated with header information into cells. The ATM layer provides end-to-end sequential connection-oriented transport of the cells according to the protocol information in their headers. The transport of user cells with an agreed quality of service and throughput is often called the ATM bearer service. The handling of cells containing network information depends on the particular cell type and its function.

The transport of cells in the ATM layer involves:

- Cell encapsulation and decapsulation;
- Insertion and extraction of idle (unassigned) cells;
- Cell header processing (including VPI/VCI translation);
- Cell forwarding (routing and buffering);
- Multiplexing and concentration;
- Generic flow control at the UNI.

2.3.2. Virtual Channels and Virtual Paths

Let's take a more detailed look at the cell header. The label consists of two components: the virtual channel identifier and the virtual path identifier. These identifiers do not have end-to-end (user-to-user) significance; they identify a particular virtual channel or virtual path on the link over which the cell is being transmitted. When the cell arrives at the next node, the VCI and the VPI are used to look up in the routing table to which outgoing port the cell should be switched and what new VCI and VPI values the cell should have. The routing table values are established at the set-up of a connection, and remain constant for the duration of the connection, so the cells always take the same route through the network, and the "cell sequence integrity" of the connection is maintained.

But surely only one label is needed to achieve this cell routing mechanism, and that would also make the routing tables simpler: so why have two types of identifier? The reason is for the flexibility gained in handling connections. The basic equivalent to a circuit switched or packet switched connection in ATM is the virtual channel links, i.e. it groups a number of VC links in parallel. This idea enables direct "logical" routes to be established between two switching nodes that are not connected by a direct physical link.

In setting up a VCC (Virtual Channel Connection), the cross-connect is effectively invisible; it does not need to know about VCIs and is therefore not involved in the process. If there was only one type of identifier in the ATM cell header, then either direct physical links would be needed between each pair of switching nodes to create a mesh network, or another switching node would be required at the hub of the star network. This hub switching node would then have to be involved in every connection set-up on the network.

Thus the VP concept brings significant benefits by enabling flexible logical network structures to be created to suit the needs of the expected traffic flows. It is also much simpler to change the logical network structure than the physical structure. This can be done to reflect, for example, time-of-day changes in demand to different destinations.

In some respects the VP/VC concepts are rather similar to having a two level time division multiplexing hierarchy in a circuit switched network. It has extra advantages in that it is not bound by any particular framing structure, and so the capacity used by the VPs and VCs can be allocated in a very flexible manner.

2.3.3. Multiplexors, Switches and Cross-Connects

Nodes in a network handling ATM cells have to merge traffic streams from different sources and route them to different destinations via switch paths and transmission links. This process involves the temporary storage of cells in finite sized buffers, the actual pattern of cell arrivals causing queues to grow and diminish in size. This is similar in principle to queuing in packet switched networks, although the speed of operation is significantly higher (typically 155.52, 622 Mbit/s in ATM compared with 64kbits/s for packet-switched networks).

A major difference, however, is the fact that there is no link-by-link acknowledgment and error control on the whole cell. The cell header is protected by an eight bit HEC field. This is used to correct single-bit errors, and detect and discard cells with multiple errors.

But there is no protection for the information field and no provision for cell retransmission over the link -- if this is required it must be performed end-to-end by the terminals.

Thus the main job of multiplexors, switches and cross-connects is provide temporary storage for cells in transit, and routing to the correct output port.

2.3.4. Controlling the Connections

I have mentioned that one of the main problems in packet switching is the large variation in delay suffered by packets. ATM also queues cells in nodes across the network, introducing variable delays. How can this be tolerated if an ATM network is supposed to cater for all types of telecommunication – both interactive and non-interactive? Voice services, for example, cannot tolerate large and variable delays, along with rate reduction if the network is congested.

To establish connections (whether virtual paths or virtual channels), ATM operates in a similar way to circuit switching. Upon receiving a connection request, the network has to access whether or not it can handle the connection, in addition to what has already been accepted on the network. This process is rather more complicated than for circuit switching, because some of the connection requests will be from variable bit-rate services. Consequently, the instantaneous bit-rate required by such services will be varying in a random manner over time, as indeed will be the capacity available because some of the existing connections will also be VBR! So if a request arrives for a time varying amount of capacity, and the capacity available is also varying with time, it is no longer a trivial problem to determine whether the connection should be accepted.

In practice such a system will work in the following way: the user declares values for some parameters which describe the traffic behaviour of the requested connection, as well as the loss and delay performance required; the network will then use these traffic and performance values to come to an accept/reject decision and inform the user. If the connection is accepted, the network has to ensure that the connection corresponds to the declared traffic values. This whole process is aimed at preventing congestion in the network and ensuring that performance requirements of each connection are met.

The traffic and performance values agreed by the user and the network form a traffic contract. The mechanism which makes the accept/reject decision is called CAC (Connection Admission Control), and this resides in the switching nodes in an ATM

network. A mechanism is also necessary to ensure subscriber compliance with the traffic contract, i.e. the user should not exceed the peak (or mean, or whatever) cell rate that was agreed for the connection. This mechanism is called UPC (Usage Parameter Control) and is situated on entry to the network. If the user does exceed the traffic contract, then the UPC mechanism takes action to protect the network from the effects of this excess, e.g. discarding some of the cells from the non-compliant connection.

CAC and UPC are the two most important traffic control mechanisms in ATM. In order to design algorithms for these mechanisms, we need to understand the characteristics of ATM traffic sources, and the effects these sources have when they are multiplexed through buffers in the network, in terms of the delay and loss suffered by cells. How we design the algorithm is very closely related to how large we make the buffers, and whether and priority mechanisms are proposed. Buffer dimensioning and priority mechanisms depend on how we intend to handle the different services and their performance requirements.

2.3.5. ATM Switching

In general ATM switching systems are network elements that support the cell transport, connection control, and management functions of ATM networks as above. While standards specify the ATM-layer and physical layer functions of ATM networks, they do not specify their implementation at the switch level. Therefore, the ATM switch architecture is a challenge for switch researcher to integrate the complexity ATM network functions and broadband traffic services on an ATM switch. In next chapter, several typical types of ATM switch architecture will be discussed in detail.

Chapter 3

ATM Switch Architecture Overview

A large number of switching architectures has been proposed, [Tobagi, 1990], [Ahmadi, 1989], [Awdeh, 1993], [Jacob, 1993], [Listanti, 1989], [Newman, 1990]. All the approaches point to the need of a very high speed hardware switch because of the high transfer rates involved; on the other hand, due to the statistical multiplexing, buffering is also required in order to avoid cell loss whenever there are multiple input cells arriving simultaneously on different input ports and destined for the same output. Only one cell at a time can be transmitted over an output link; the rest must be temporarily stored in a buffer for later transmission.

In this chapter, the architecture of various approaches to ATM switch design will be introduced, including ATMSWITCH (ATM Switch With Integrated Trigger and Chandelier Hardware) architecture [Lyons, 1997], which is the subject of this thesis. The chandelier is a dynamic data structure implemented in hardware, which buffers cells for UBR and ABR channels. The Trigger is a scheduling mechanism which shares some of the chandelier data structures, and provides for the regular forwarding of cells belonging to CBR and VBR channels.

3.1. ATM Switch Definition and Functionality

According to the paper [Tobagi, 1990], An ATM switch is a box with N inputs and N outputs which routes the cells arriving on its inputs to their requested outputs. For simplicity in presentation, we begin with the assumption that all lines have the same transmission capacity, all cells are of the same size, and that the arrival times of cell at the various input lines are time-synchronized. We thus consider the time axis to be slotted with the slot size equal to the transmission time of a cell on a line, and consider the operation of the switch to be synchronous. We also assume for now that each cell is destined for a single output port. However, there is no coordination among arriving cells as far as their destination requests are concerned, and thus more than one cell arriving in the same slot may be destined to the same output. Such an event is referred to as output conflict. Due to output conflicts, buffering of cells within the switch must be provided.

Thus an ATM switch is a box which provides two functions: routing (or, equivalently, switching) and buffering.

Tobagi describes an ideal switch as one which is work-conserving, "An ideal switch is one of that can route all cells from their input lines to their requested output lines without loss and with minimum transit delay possible, while preserving the order in which they have arrived to the switch". This description seems sufficient at first glance, but it is a little naive. In particular, it fails to take into account the particular requirements of CBR channels, which should not be given automatic access to any available time-slot if it becomes available when there are no cells queued for other channels. If this were to occur, the ultimate destination of the CBR channel could be overrun with cells. Accordingly, the idea switch should be work-conserving (i.e., at any slot t, if there is at least one cell which had arrived at the switch and which is destined to output port j, then such a cell must be transmitted out on output line j). The switch should also have sufficient buffering capacity that the buffer size is sufficiently large to reduce the probability of cell loss to less than some acceptable ratio, i.e. 10^{-8}. (Note that, for a work-conserving switch, if cells arriving in each slot present no output conflicts, then virtually no buffering would be required.)

"While the functionality required of an ATM switch is practically the same as that required of packet switches used traditionally in computer networks, the challenge here is to design switches that meet the speeds required." The definition about functionality of switch is given by Chen [Chen, 1995]. The functionality required is practically the same as for computer is to miss an important point about the constant time-relationship which must be maintained between the devices at the ends of the network connection in many data transfer situations. The required switch function must provide various transmission scheme regarding the time required for different type of cells.

Several architectural designs have emerged in the recent years. These may be classified into three categories; namely: the shared-memory type, the shared-medium type, and the space-division type. Each of these categories presents features and attributes of its own. In all three cases, however, the technology used in implementing the switch places certain limitations on the size of the switch and line speeds; thus to build large switches, many modules are interconnected in a multistage configuration, which provides multiple paths from the inputs to the outputs, thus offering the concurrency required to handle the large size.

An important factor affecting the performance of an ATM switch is the traffic pattern according to which cells arrive at its inputs. The traffic pattern is determined by (1) the

process which describes the arrival of cells at the inputs of the switch, and (2) the destination request distribution for arriving cells. The simplest traffic pattern of interest is one whereby the process describing the arrival of cells at an input lines is a Bernouilli process with parameter p, independent from all other input lines, and whereby the requested output port for a cell is uniformly chosen among all output ports, independently for all arriving cells. Such a traffic pattern is referred to as the independent uniform traffic pattern. Other traffic patterns may actually arise which exhibit dependencies in the cell arrival processes as well as in the distribution of output ports requested. For example, cells may arrival at an input line in the form of bursts of random lengths, with all cells in a burst destined to the same output port. The traffic pattern in this case is defined in terms of the distributions of burst lengths, of the gap between consecutive bursts, and of the requested output port for each burst traffic pattern. Yet another example of a dependent traffic pattern may be found in applications that produce cells at regular intervals.

Besides the basic switching operation that is performed by a switch, two other functions are required. The first is multicast operation. Depending on the application being serviced, it may be necessary for a cell originating at a source node in the network to be destined to more than one destination. This could be accomplished by creating multiple copies of the cell at the source node, each destined to one of the desired destinations, and routing the copies independently. Alternatively, multicast routing may be achieved by requiring the switches in the network to have the capability to replicate a packet at several of their output ports, according to information provided for that purpose, thus reaching the multiple destination from a single cell originating at the source. This mode of operation results in lower traffic throughput the network but at the expense of higher complexity in switch design. The second function that may be required of a switch is the priority function. It consists of the ability to differentiate among cells according to priority information provided in them, and to give preferential treatment to higher priority cells. Multicast and priority functions are achieved in the various architectures by various means.

In order to describe more detailed ATM switch architecture, I give following descriptions that will be used in rest of the thesis.

3.2. Performance Measure

The performance of an ATM switch is usually evaluated based on three measures: throughput, delay, and cell loss probability. Throughput (TP) is defined as the average

number of cells which are successfully delivered by the switch per time slot per input line. Maximum throughput (TP_{max}) is the value of TP under maximum-load conditions. While TP_{max} is an important performance measure, it will not be directly felt by network users. On the other hand, the end-to-end delay, which includes the delay of individual switching nodes, will be experienced by network users. The reason is that users are only sensitive with time related signals, such as voice and video signal. Switch delay (D) is defined as the average time (in time slots) a cell spends from the time it arrives at an input port, till the time it is successfully delivered on its requested output line. D includes the time spent in any input, internal, and/or output buffers. Cell loss probability (P_{loss}) is defined as the fraction of cells lost within the switch. Cell loss might occur as a result of blocking and/or buffer overflows. Because cell re-transmission takes place on an end-to-end basis in ATM networks, and because of the high speeds involved in these networks, P_{loss} is considered as very important performance measure.

3.3. Traffic Model

Here, we refer to the traffic as seen by the input ports of the switch. According to the paper [Awdeh, 1994], the traffic model is described by two random process. The first is the process that governs the arrival of cells in each time slot. The second process describes the distribution ports. It is clear that input traffic can follow an infinite number of models. In the following, we describe three of the most frequently used traffic models for the performance evaluation of ATM switches.

Uniform traffic

In this model, cells arrive at the input ports of the switch according to independent and identically distributed Bernoulli processes, each with parameter p $(0<p\leq 1)$. In other words, at an input port in a given time slot, a cell arrives with probability p, and there is no arriving cell with probability $1-p$. Thus, p represents the input load or the arrival rate to each input port of the switch. An incoming cell chooses its destination uniformly among all N output ports, and independently from all other requests; i.e., it chooses a particular output port with probability $1/N$. This traffic model is sometimes referred to as the independent uniform traffic model [Tobagi, 1990], or simply the random traffic model. Asserting the assumption of uniformity for real-life situations may lead to optimistic evaluation of performance measures. However, a large number of studies on the performance evaluation of ATM switches assume uniform traffic. The main reasons behind this trend are as follows:

- This assumption makes the analytical evaluation of the switch more tractable, specially if the switch is of the blocking type and/or employs a complex buffering strategy.
- A distribution/randomization network can be used at the front end of the switch to randomize incoming traffic.
- It was observed that the traffic arriving at the switching nodes is less burst than the traffic arriving at user access nodes, due to the inherent smoothing that takes place when burst cell streams are queued and then released at a given rate (the link services rate) to the network [Friesen, 1993]. Furthermore, it was shown that subsequent stages of switching cause the traffic to become even less burst [Descloux, 1991], making the uniform traffic assumption closer to reality.

Burst traffic

Future B-ISDN is expected to support virtually all existing and emerging services including voice, video and data. Strong correlation may exist among cells originating from the same source, giving rise to burst traffic. A burst source generates cells at a peak or a near-peak rate for very short duration, and remains almost inactive in between. Several models have been proposed to describe such burst sources [Bae, 1991]. One popular and simple model is the On/Off model, where the source alternates between a busy (also called on, active, or silent) period. The probability that the active period lasts for a duration of i time slots is given by $B(i) = a(1-a)^{i-1}$, $i \geq 1$; it is assumed that a burst contains at least one time slot.

Hot-spot traffic

Hot-spot traffic refers to a situation where many input ports prefer to communicate with one output port (the hot-spot). This kind of traffic may arise in many real-life application [Yoon, 1988]. It occurs, for example, when many callers compete to call a popular location in a telephone network. Another example is a local area network consisting of many diskless computer systems and a single file server. In the model introduced in [Pfister, 1985], a single hot-spot of higher access rate is superimposed on a background of uniform traffic. The cell arrival process is the same as for the uniform traffic case. If we let h be the fraction of cells directed at the hot-spot, then p, the arrival rate to an input port, can be expressed as $p = ph+p(1-h)$. In other words, hp cells are directed at the hot-spot, and $p(1-h)$ cells are uniformly distributed over all output ports. Also, the average number of cells which request the hot-spot per time slot is $phN + p(1-h)$.

3.4. Buffering Strategies

Due to the statistical nature of the traffic, buffering in any packet switch is unavoidable. This is true because even with an output-non-blocking switch (which can clear all incoming cells to the output side of the switch before a new time slot begins), two or more cells may address the same output port within the same time slot. In such a situation, given the assumption that input and output lines operate at the same speed, each output line can serve only one cell per time slot; other cells must be buffered. This is called output buffering since the buffers will be physically located at the output side of the switch.

With a switch that is able to deliver a maximum of one cell to each output port in any given time slot, output buffering is not needed. In such switches, buffers can be placed at the input ports (called input buffering), within the switching fabric at possible points of contention (called input-internal buffering). On the other hand, if the switch can deliver more than one cell (but not all possible cells) to each output port simultaneously, then it also needs output buffering. In this section, we review and compare two extreme strategies of external buffering, namely, input buffering and output buffering.

3.4.1. Output Buffering

Here, we assume an output-nonblocking switch (Fig. 3.1) where all arriving cells in a given time slot are cleared to the output side (i.e., are switched) before the beginning of the next time slot, even if all N inputs have cells destined to the same output port. This can be achieved by, for example, speeding up any switching fabric by a factor of N. However, only one cell can be served by an output line in each time slot and other cells with the same output request have to buffered, if space is available. The system is stable, since the average utilization of an output line is the same as that of an input line.

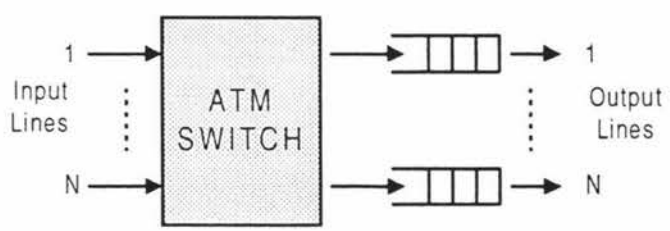


Figure 3.1: A generic nonblocking output buffering switch

The average queueing delay Q when $N = \infty$ and $B_{out} = \infty$ is the same as that of an M/D/1 queue [Eckberg, 1988]:

$$Q = \frac{\rho}{2(1-\rho)}; \quad 0 \leq \rho < 1$$

3.4.2. Input buffering

In an input buffering switch (Fig. 3.2), an arriving cell enters a first-in first-out (FIFO) buffer located at its port of entry, if space is available. In each time slot, the switch resolves contentions prior to switching. If all head-of-line (HOL) cells are destined to distinct output ports, then all of them are admitted and switched to their desired output lines. However, if K HOL cells ($1 < K \leq N$) are destined to a particular output port, only one cell is chosen, according to some selection policy, to be switched and other cells wait to participate in the next time slot selection process.

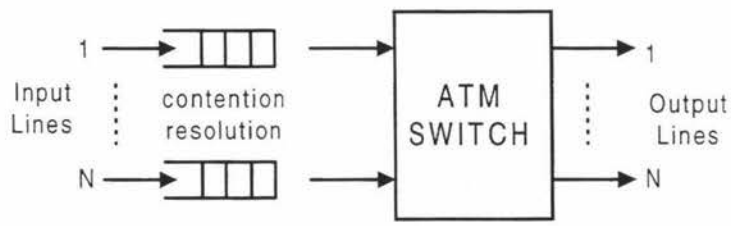


Figure 3.2: A generic input buffering switch

Several selection policies (also called contention resolution mechanisms or HOL arbitration resolution mechanisms or HOL arbitration schemes) have been proposed. It should be emphasized here that while the selection policy among contending HOL cells can be non-FIFO, the service discipline within each input buffer is still assumed to be FIFO.

The following equation [Hui, 1987] describes the relationship between the average queuing delay Q and the arrival rate p (for $N = \infty$, $B_{in} = \infty$, and random selection):

$$Q = \frac{(2-\rho)(1-\rho)}{(2-\sqrt{2}-\rho)(2+\sqrt{2}-\rho)} - 1; \quad 0 \leq \rho < 2-\sqrt{2};$$

The throughput is given by the following equation:

$$TP = 1 - \left(1 - \frac{\rho}{N}\right)^{N}; \quad 0 \leq \rho \leq 1;$$

3.5. ATM Switch Architecture

Tobagi has described a number of architectures which have been proposed for ATM switches [Tobagi, 1990]. All the approaches point to the need for very high speed hardware in the switch because of the high transfer rates involved. The architectures may be classified into three categories: the shared-memory type, the shared-medium type and the space-division type. In the following section I will introduce each of them.

3.5.1. Shared-Memory ATM Switches

Shared-memory ATM switch is actually a great conceptual resemblance that they bear to traditional packet switches used in conventional wide-area computer networks, and to synchronous time division switches based on the time-slot interchange mechanism used in circuit switched networks. The switch consists of a single dual-ported memory shared by all input and output lines. Cells arriving on all input lines are multiplexed into a single stream which is fed to the common memory for storage; internally to the memory, cells are organized into separate output queues, one for each output line. Simultaneously, an output stream of cells is formed by retrieving cells from the output queues sequentially, one per queue; the output stream is then demultiplexed, and cells are transmitted on the output lines.

In this type of architecture, two main design constraints must be satisfied. First, the processing time required to determine where to enqueue the cells and issue the proper control signals for that purpose should be sufficiently small to keep up with the flow of incoming cells. That is, there must be a central controller capable of processing sequentially N incoming cells and selecting N outgoing cells in each slot. The second and most important design constraint pertains to the shared memory. I focus here on memory access time and bandwidth requirements. Clearly, the memory bandwidth should be sufficiently large to accommodate simultaneously all input and output traffic. If N is number of ports and V is the port speed, then the memory bandwidth must be 2NV. For example, for a 16-line switch with line speeds of 1.2Gb/s, the memory bandwidth should

be at least 38.4Gb/s. Given the limitation on memory access speeds, the required memory bandwidth is achieved by the means of a parallel memory organization.

On the other hand, given that the size of cells is limited, there is a limit on how many memory banks may be used in parallel, and there is a point at which the memory access time becomes the bottleneck. In the limit, the memory must be accessed N times per slot for enqueuing cells, and N times per slot for dequeuing cells. Thus, the size of a switching module (number of lines and line speeds) is determined by available memory speeds and achievable processing speeds. Given the line speeds and switch sizes being contemplated, it is clear that the implementation of such switches must be hardware-based using LSI (Large-Scale Integrated) circuits, and thus the design problem consists of identifying and building the various LSI circuits needed. Furthermore, given the high-speed nature of the required circuitry, it is important to limit the number of chips involved (and this includes memory chips), so as to keep the board-level design manageable within the imposed constraints.

3.5.2. Shared-Medium Switches

In shared-medium type switches, all cells arriving on the input lines are synchronously multiplexed onto a common high-speed medium, typically a parallel bus, of bandwidth equal to N times the rate of a single input lines (see Figure 3.3).

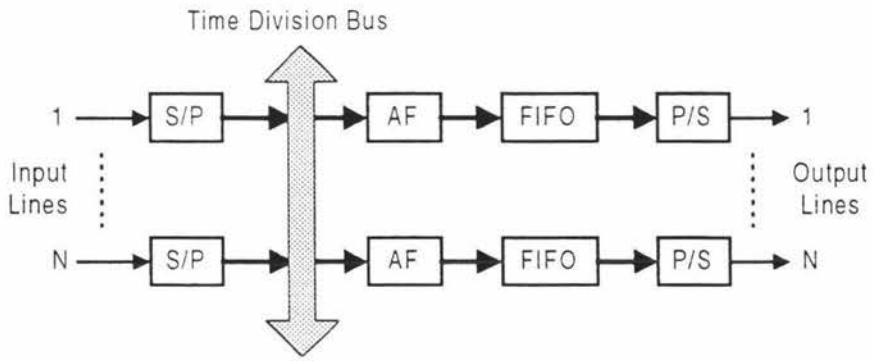


Figure 3.3: Basic structure of a shared-bus type architecture

Each output line is connected to the bus via an interface consisting of an address filter and an output FIFO buffer. Such an interface is capable of receiving all cells transmitted on the bus. Depending on the cell's virtual circuit number (or its output address), the address filter in each interface determines whether or not the cell observed on the bus is

to be written into the FIFO buffer. Thus, similarly to the shared-memory type, the shared-medium type switch is based on multiplexing all incoming cells into a single stream, and then demultiplexing the single stream into individual streams, one for each output line. The pathes through which all cells flow here is the basically done by the address filters in the output interfaces. Conceptually, this approach is also similar to the architecture used in circuit switches based on a TDM bus, with the exception again that here each cell must processed on the fly to determine where it must be routed, rather than based on prerecorded information established during the circuit setup phase, and each output interface must be capable of receiving cells at the aggregate rate of N cells per slot.

The distinction between this type and the shared-memory type is that in this architecture there is complete partitioning of the memory among the output queues, and thus the latter can be organized as FIFOs. An example of such a switch architecture can be found in the Atom (ATM output buffer modular) switch proposed and being designed by NEC [Suzuki, 1989].

Similarly to the shared-memory architecture, an essential issue in realizing the shared-bus architecture is how to implement the high-speed bus and buffer memories, all of which must operate at the aggregate speed of NV, where V is the port speed. The solution to this problem is also very similar to that introduced for the shared-memory architecture. Given typical limitations on memory access speeds and LSI chip sizes (gate count and pin count), the throughput of designed ATM switch is thus determined.

3.5.3. Space-Division Switches

By contrast with the shared-memory and shared-bus architectures, where traffic from all input lines is multiplexed into a single stream of bandwidth equal to N times the bandwidth of a single line, in a space-division switch, multiple concurrent paths are established from the inputs to the outputs, each with the same data rate as an individual line. As a result, no memory component in the switching fabric has to run at a speed higher than 2V, the line speed. Another distinctive feature is that the control of the switch need not be centralized, but may be distributed throughout the switching fabric.

This type of architecture, however, does present problems of its own. Depending on the particular internal fabric used, and the resources available therein to establish the paths, it may be impossible for all required paths to be set simultaneously. This characteristic, commonly referred to as internal blocking, limits the throughput of the switch, and thus becomes a central issue underlying space-division switches.

A related issue is buffering. In fabrics exhibiting internal blocking, it is not possible to buffer cells at the outputs, as is possible in switches of the shared-memory and shared-bus type. Instead, buffers must be located at the places where potential conflicts among paths may occur, or upstream of them. Ultimately, buffers may be placed at the input of the switch. The placement of buffers has an important effect on the performance of a space-division switch, as well as on its hardware implementation.

According to the paper [Tobagi, 1990], space-division switches have taken many forms, The typical space-division switch can be represented by crossbar fabrics, shown Figure 3.4. To best present and discuss the fabrics, it is useful to consider the following simple abstract model of a switch. The model prescribes that for each input line i there is a router (that is, demultiplexor) which routes its cells to N separate bins, numerbeded (i, 1) to (i, N), one bin for each output port. At the output side, we consider that for each line j there is a concentrator (multiplexor) which connects all bins (i, j), i=1, 2, ..., N, to output line j and which, in each time slot, selects one cell, if any, from these output bins for transmission on output line j. Thus the model switch consists of N routers, N concentrators, and N^2 buffers. The various fabrics proposed differ by the ways the routers and concentrators are implemented, and by the locations of the buffers.

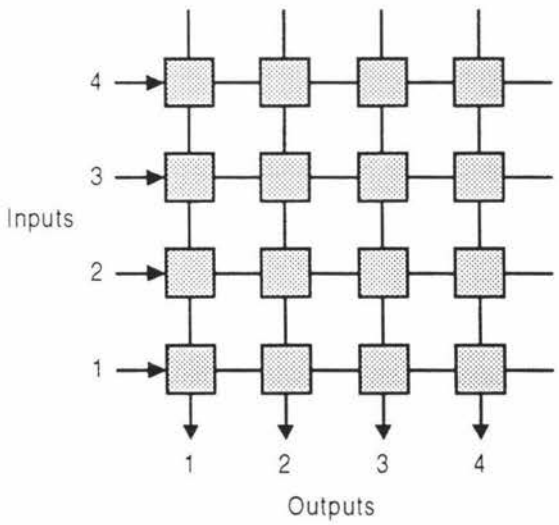


Figure 3.4: Crossbar switching fabric

The ATM switch architectures described above, all focused on internal structure of switch for routing a cell from input port to the desired output port. None of them consider any approach to the QoS (Quality of Service) required by different types of data traffic.

The QoS guarantees should be provided in ATM networks by the use of the proper traffic scheduling algorithms in the ATM switches, [Verma, 1997]. Implementation of the algorithm must usually be implemented in hardware to satisfy the speed requirements. This part of design should be considered as part of ATM switch architecture. But, none of above proposed architecture addressed on this problem.

In recently years, several papers have discussed how to provide QoS guarantees in ATM network by using a scheduling algorithm in the switches, [Bennett, 1996], [Zhang, 1995], [Stiliadis, 1996], [Georgiadis, 1996], [Fenwick, 1997]. The main idea of proposed algorithm is using the traffic schedulers to provide network-level QoS to application by isolating their traffic from other. A scheduler defined in these papers is regarded as a queuing system consisting of a server providing service to a set of customers. The customers queue packets for service, which are chosen by the scheduler for transmission based on a service discipline defined by the scheduling algorithm. The service discipline must be designed to meet the desired QoS requirements of individual customers.

Several service disciplines are introduced in these literature for bandwidth allocation and scheduling in output-buffered switches. These algorithms can be classified based on several criteria. One classification of a scheduler is based on the type of guarantees to individual sessions, but do not provide explicit delay guarantees. In most of these schedulers, however, the bandwidth guarantee results in an implicit end-to-end delay guarantee, [Stiliadis, 1996], but the later cannot be controlled independent of the former. Other schedulers provide independent control of bandwidth and delay bounds, but at the expense of increased complexity, [Georgiadis, 1996]. In the paper, [Verma, 1997], FFQ (Frame-Based Fair Queuing) and SPFQ (Starting Potential-Based Fair Queuing) is introduced, but the article just describe a general methodology for the design of scheduling algorithm.

In next section, the ATMSWITCH architecture will be introduced, [Lyons et. al 1996, 1997]. It addressed on basic ATM service and provided a solution to the problem of conflicting requirements of different sorts of data traffic.

3.6. Trigger Mechanism and Chandelier Structure based ATM Switch Architecture

In this section, a **trigger mechanism** and **chandelier structure** based switch architecture will be introduced. The architecture was proposed by Lyons in 1996. ATM networks are intended to provide a "one-size-fits-all" solution to a variety of data communication needs, from low speed, delay-insensitive to high-speed, delay-intolerant. The basic ATM protocol certainly delivers traffic within this broad range, but it does not address the quality of service requirements associated with the various type of traffic. In particular, its statistical multiplexing nature is inherently antagonistic towards constant-delay traffic

such as real-time video. The ATMSWITCH is a shared-buffer type of architecture which uses two different mechanisms for selecting cells to output according to their type. The trigger mechanism solves the jitter problem of CBR/VBR channels, by forwarding cells from these channels at a constant intervals. The chandelier mechanism allocates bandwidth to ABR/UBR channels in a more opportunistic manner, consistent with the conventional ATM approach, but uses a separate queue for each channel, to minimize interference between high-throughput and short response-time channels. The trigger mechanism has higher-priority access to the output port than the chandelier, but if a channel which uses the trigger mechanism has no cell queued for output when it is due to be triggered, access to the output port reverts to the chandelier. Hence the bursty traffic characteristic of VBR channels does not result in wasted bandwidth. The architecture can be classified as a shared-buffer type of switch architecture in which I already described on last section. The used buffer strategy is output buffer.

3.6.1. Architecture Description

In 1996, Lyons, McGregor and Moretti proposed an Associative Chandelier structure [Lyons, 1996] based ATM switch architecture. It can perform almost all functions of ATM layer, such as routing a cell from input port onto assigned output port., buffering a cell, and outputting cells in correct serial order. It also allow priority services among different cell queues. Lyons et al. desciibed a detail circuit level structure that can achieve 1.2 Gb ATM transmission speed. The architecture uses a round robin as a multiplexor to get a cell from selected input port at each time slot. A CAM (Content Addressable Memory) known as Map Memory was used as a look up table to handle all information about virtual path/channel, cell input port and output port, channel type, cell queue information. The cell buffer is implemented by a fast cache memory which can read or write a cell within a 12ns time cycle. The cell buffer memory was shared by all output ports. The design used another component that was called cell queue link to maintain the cell queue pointer. The novel feature of this structure is chandelier structure. It put outgoing cell onto a queue belonging to the channel and connected the queue header as a round robin. During each output time cycle, the output port will remove a cell from the first item of the selected queue and put it onto output buffer. This architecture is very suitable for ABR, UBR classes of service, but it have jitter problem with CBR and VBR channels.

In 1997, Lyons proposed adding a new Trigger mechanism [Lyons, 1997] into the associative chandelier to provide the regular cell-forwarding behaviour necessary for CBR and VBR channels. The trigger mechanism adds a counter into ATM switch and

stores an ITV (initial trigger value) and a mask value (the relationship between the mask value and the bandwidth requirement of the channel is described later) for each channel. On each output cycle, the counter value is increased and if, masked, it matches a channel's ITV, then that channel is due for output. If no channel is selected for output by the trigger, access to the output port defaults to the round robin chandelier.

The trigger mechanism schedules regular output from CBR/VBR channels. So it can solve the problem of time sensitive channel CBR and VBR transmission within ATM network.

3.6.2. Associative Chandelier Structure

Lyons, McGregor and Moretti's (1996) hardware Associative Chandelier is a development of an earlier, software buffering system used in MasseyNet, a packet-switching LAN (Lyons and McGregor, 1986, 1990). In 1996, Lyons, McGregor and Moretti proposed the associate chandelier, an updated version for use in ATM systems. The new version of chandelier was designed for implementation in hardware to achieve the speeds characteristic of ATM networks. One of the main features of the architecture was an associative memory or CAM (Content-Addressable Memory) which was used for matching incoming cells' channel identification information so that the cell could be buffered in the correct queue. To support this, the associative memory stored the channel's input port number and channel number. The channel's output port number and channel number are identical for all cells in a particular queue, and the storage requirement for the system can therefore be minimized by storing them also in the channel header in the associative memory. This memory therefore contained all the route-mapping information for all channels with links through the switch, and is consequently called Map Memory.

The chandelier comprises two data structures. The first data structure is implemented in Map Memory and a circular list of channel identifiers. Only channels which currently have data buffered in the switch have an entry in the circular list. When a channel's buffer becomes empty, its entry is removed from the circular list, but not disposed of completely. The consequence of this is that traversal times for the circular list are minimized - there is not need to deal with channels which do not have data buffered in the switch. The associative addressing on the other hand allows rapid identification of the entry corresponding to an incoming cell's channel. Thus when new data arrives for that channel, the entry can quickly be linked into the circular list again.

The second data structure is a set of FIFOs, implemented in two interrelated hardware modules; 12ns cache memory chips contains cell data fields. The links between the FIFO elements are maintained in a separate, higher speed memory, variously referred to as link memory and CQLinks (for cell queue links). The link memory must run faster than the data buffer because, although map memory contains pointers to both the head and the tail of the FIFO it is still necessary to access this memory more than once when updating the links.

Each entry in the first data structure, the circular list, points at the FIFO containing cells buffered for its channel. A round robin server travels around the circular list, visiting each channel in turn, and outputting one cell from its FIFO during the visit. If the FIFO becomes empty, the channel's header entry is removed from the circular list. Thus only channels which currently have cells buffered in the switch contribute to the service time for the circular queue. Figure 3.5 shows these two data structures and empty channel header.

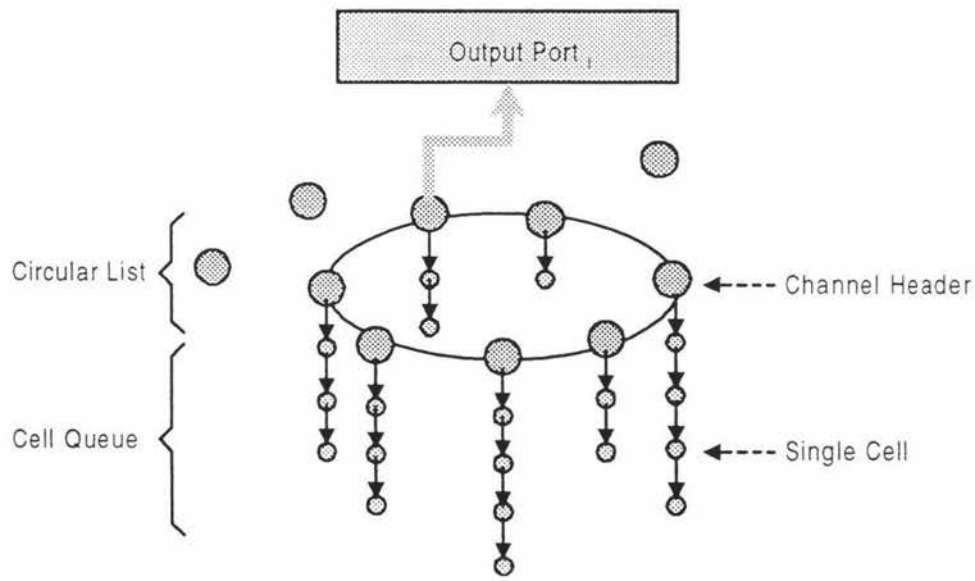


Figure 3.5: Two Data Structures of Chandelier

In above picture, a cell arriving from an input port will be appended at the end of its channel's cell queue. At each outgoing slot, the round robin server moves to the next channel header in the circular list and the cell pointed to by this channel header will be sent to the output port.

Since the number of active channels varies, the size of chandelier and hence the service rate for channels with cells queued in the chandelier, will vary. This will be experienced as jitter at the destination. So the chandelier structure will suit ABR and UBR channels, but have problems with CBR/VBR channels, which are not tolerant of jitter. In order to overcome the jitter problem, Lyons proposed a Trigger Mechanism [Lyons, 1997] that combines with the associative chandelier structure to support all class of services of ATM switch.

3.6.3. Trigger Mechanism

The general idea of the trigger mechanism is to schedule regular output slots for a CBR/VBR channel. Thus a burst of cells arriving at the switch from an outside source will be forwarded at regular intervals - which reduces the probability of buffer overrun in subsequent nodes - and some of the cells in the burst will be buffered in the first node while they wait for their scheduled service. Due to all cells of one CBR/VBR channel pass the switch at a constant time, the delay within these cells almost same. The jitter thus can be ignored.

The ATM switch should therefore regularly trigger output for a given CBR/VBR channel, and the trigger frequency must be able to be set according to the agreed bitrate(the maximum bitrate, in the case of VBR channels) of the channel. So we need a special device that is able to trigger each channel in terms of their speed.

As described in Lyons (1997), a regularly incremented counter was proposed to provide a time base to support this mechanism. Each CBR/VBR channel is associated with a unique set of equally-spaced values in the counter's range and will be triggered whenever the counter value equals one of these trigger values. The spacing between triggers will be inversely proportional to the channel's speed, so that high-speed channels will be triggered frequently, low-speed channels less frequently. Allocating suitable trigger values in a dynamic environment, where channels of different speeds are being set up and destroyed, is not completely trivial, and will be discussed later.

Figure 3.6 shows the counter as a pointer that rotates through a set of values, 0 to 2^n-1 (n is 4 in this case), shown on the inner ring. Some of the values are trigger values for particular channels. These are indicated by arrows to the channel header from the sectors on the outer ring adjacent to the trigger values. Thus the left channel is due to be triggered, because the counter value is 1. It will also be triggered when the counter value is 9, and the right channel would be triggered when the counter value is 2, 6, 10, or 14. It

should be emphasized that this is a simplified model showing the functionality of the trigger mechanism; it is not an accurate representation of its implementation.

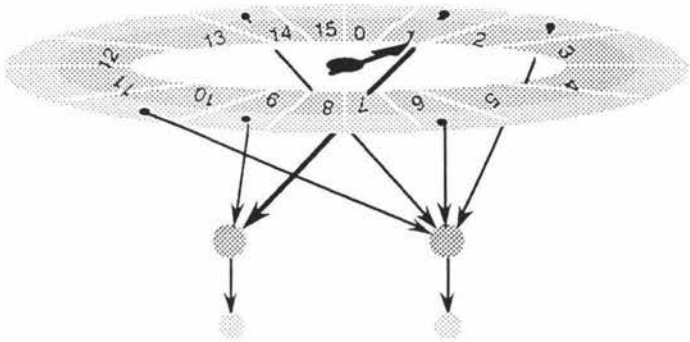


Figure 3.6: Counter value and trigger value

If a channel is triggered, but its buffer queue is currently empty (as would often be the case for VBR channels), then access to the output port will default to the ABR/UBR channels with data buffered in the chandelier. Similarly, if the current counter value is not a trigger for a CBR/VBR channel, then output will also be able to come from the chandelier.

Figure 3.7 shows the trigger mechanism and chandelier structure selecting a cell onto output port. In this diagram, the counter number is three and it equal to a trigger value of a CBR/VBR channel. So at this particular time, the outgoing cell will be selected from this channel. In the next chapter, I will detailed describe how the trigger mechanism can make CBR/VBR cell stay at switch as constant time and then is triggered onto output port.

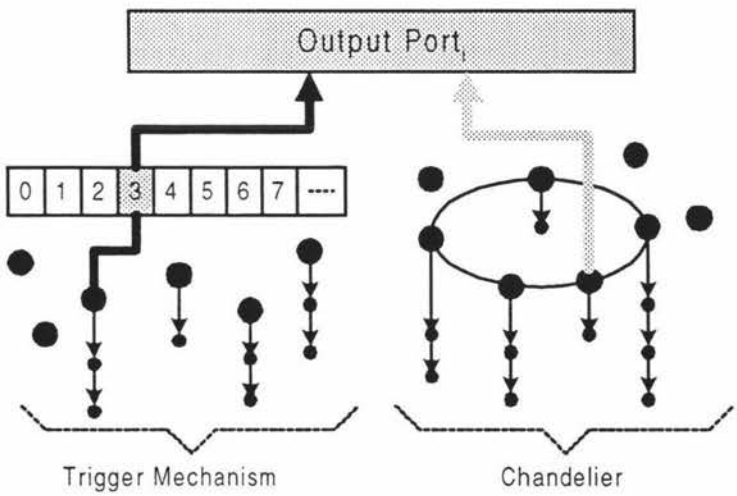


Figure 3.7: Trigger Mechanism and Chandelier

3.6.4. Speed Achievable of Trigger Mechanism and Chandelier Based Switch

In above sections, I introduced some basic ATM switch concepts and reviewed three typical switch architectures. Lyons ATMSWITCH architecture, incorporating the trigger and the chandelier was described later in this chapter. According to classification of ATM switch, the trigger and chandelier structure based switch is shared-memory and output buffer strategy switch. The limitations of this type of switch on the size come from the memory control logic (which must be able to handle N incoming and N outgoing cells in each time slot), and the memory bandwidth that must be at least the sum of the bandwidths of the incoming and the outgoing lines. The memory bandwidth depends on the word length, which in turn is limited to the cell size (48 octets = 384b), therefore for a given memory cycle time (or memory access time) the number of links N must satisfy the following relation (for single ported memories):

$$cycletime = \frac{celllength(b)}{2 \times N \times linkspeed(b/s)} \qquad (3.1)$$

Above equation shows a clear limitation in the maximum achievable size of a shared-memory type switch.

In ATMSWITCH architecture the cell queues are stored in a separate memory implemented using 12ns cache memory. We can use the above equation to estimate the maximum of throughput of switch. For example, if the read and write times for the cell buffer are 12ns and the speed of the input and output lines is 1.2 Gbps. The memory could therefore achieve:

Total Throughput = $424/(24*10^{-9})$ bps = 17.6667Gbps;
Switch Size = 17.6667Gbps/1.2Gbps = 14 Lines;

The above value is only an expected result. When the switch is working, it's not only writing cell data into and reading cell data from the buffer memory, but also involved in many other related circuit operations. If the switch is to achieve the maximum possible throughput, as calculated above, then all of the other circuitry for maintaining the chandelier and the trigger must operate within the 12ns memory cycle time. Lyons (1996 and 1997) provided a broad-bruch outline of circuit designs which are capable of achieving these tasks.

The task undertaken in this project, and described in this thesis was the development of a full logic design for the chandelier and trigger components of the ATMSWITCH, and the determination of the speed at which this logic would have to run if full use is to be made of the memory bandwidth.

The next chapter I will describe a basic switch structure which was based on trigger mechanism and chandelier structure. The related circuit module was also designed and implemented by using VHDL language.

Chapter 4

ATM Switch Structure and Implementation

This chapter will describe the basic structure of ATMSWITCH. The switch is designed at circuit component level and simulated by using the VHDL language.

4.1 Introduction ATMSWITCH Structure and its Circuit Modules

According to the architecture that was proposed by Lyons [1996, 1997], we described a switch structure and designed each circuit module of switch. The following diagram shows the basic circuit modules of the ATMSWITCH.

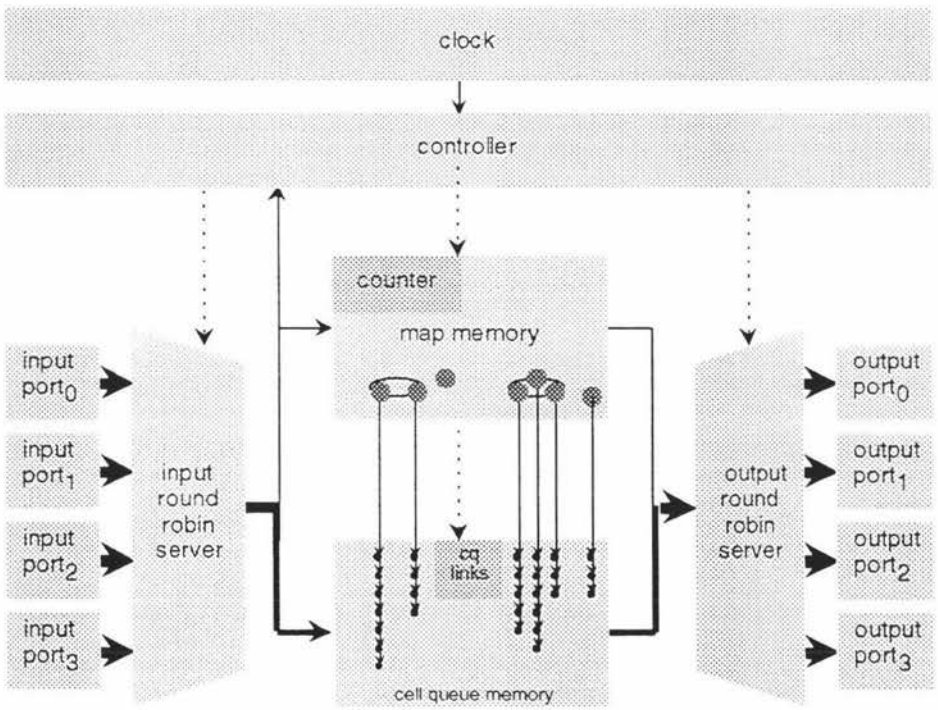


Figure 4.1: ATMSWITCH Structure

Figure 4.1 shows the main circuit modules of ATMSWITCH. They construct the switch and each of modules performs a different function and is controlled by the Controller. The basic functions are:

- **Input Port:** receives cells from input line of switch, the incoming cell wait at the input port and then is sent to cell memory (cell buffer);
- **Clock** module: used to maintain the internal time control for the switch;

- **Controller** module: outputs control signal to different circuit module according to its input signal events.
- **Map Memory:** stores each established channel information and routing information. It is used to map incoming cell onto predefined cell queue when a cell arrives, and points to the address of an output cell when cell output is due. The map memory is actually an associative memory and the chandelier structure was realized in map memory;
- **Cell Memory** (Cell Buffer): used to store the data fields of all incoming cells. The cell memory is a shared buffer structure. It's shared by all output ports;
- **CQlinks** (Cell Queue Link) memory: used to maintain the cell queue link pointer and manage cell memory address;
- **Input Port Round Robin Server** (simply called Input Server): indicates the current input port number at each incoming cell slot;
- **Output Port Robin Server** (simply called Output Server): indicate the current output port number at each *outgoing* cell slot;
- **Output Port**: receives cells from cell buffer, the cell at the output port is sent to switch output line;
- **Counter** module: outputs a regularly increased integer, the value will trigger a special CBR/VBR channel that matches this value. It's a part of trigger mechanism in ATMSWITCH.
- **D_Bus & Head_bus**: the *d_bus* is the cell payload data bus and it transfers cell payload data within the switch; the *head_bus* is cell head data bus and it transfers cell head data within the switch. Both the *d_bus* and *head_bus* are shared with switch modules.
- **A_Bus**: is cell memory address bus. It provides the cell memory address for cell reading and writing. The *a_bus* is shared with the switch modules.

In later subsections, I will describe each of the above modules in detail. Before this, however, I will describe the switch's operation at a high level.

4.2 Operation of the ATMSWITCH

ATM switches are in general used for routing a cell from an input port to a desired output port. In Lyons *et al*'s ATMSWITCH, the working procedure can be briefly divided into two main stages. The first stage is accepting a cell from an input port into the cell buffer; the second one is outputting a cell from the buffer onto the desired output port.

The Clock module outputs a clock signal to other modules of switch. The high phase of the clock signal delimits the accepting cell stage. When the clock signal changes from low to high, the controller initiates cell input. First, it checks the input port *Full* signal for the current port to see whether there are cells at input ports. If its single-cell buffer is full, the controller begins transferring the cell payload to the next available location in cell memory. During the 12ns necessary for this to occur,

- the channel identification information in the cell header is matched associatively with the map memory. The matching entry in map memory contains pointers to the head and tail of the buffer queue for the channel to which the incoming cell belongs, so that it is also possible to ...
- update the CQlinks memory to link the location in cell memory into its channel's queue.

In order for this second action to occur to happen while the (comparatively slow) cell memory is being loaded, CQlinks (which is logically parallel to cell memory) must run considerable faster than cell memory. It has been the object of this research project to determine just how fast CQlinks, map memory, and the controller logic must run in order for all the administration to occur while the cell buffer is loading or unloading a payload.

Cell output occurs while the clock signal is low. First, the trigger mechanism checks to see whether there are any CBR/VBR channels waiting to output a cell. If the trigger value is true, the cell will be chosen from the trigger mechanism rather than chandelier structure. If the trigger value is false, the cell will be sent from chandelier structure to the output port. After the cell transfer is initiated, CQlinks and the channel's map memory entry are updated to reflect the new structure of the buffer queue, and the channel header is removed from a chandelier if its buffer queue has become empty, the relative value will be updated.

In addition to these two main phases, there is also a Setup phase during which the switch sets up or clears a map memory entry at the establishment or disestablishment of a virtual channel.

4.3 Clock Module

The clock provides a synchronous timing control for all circuit components of ATMSWITCH. In designed switch, the clock outputs three signals for other circuit modules. The first one is the *reset* signal. When the switch is turned on or in other special cases in which the switch needs be reset, the signal will be set to high, logic '1'. If the *reset* signal was changed to high, all components of the switch will be reset.

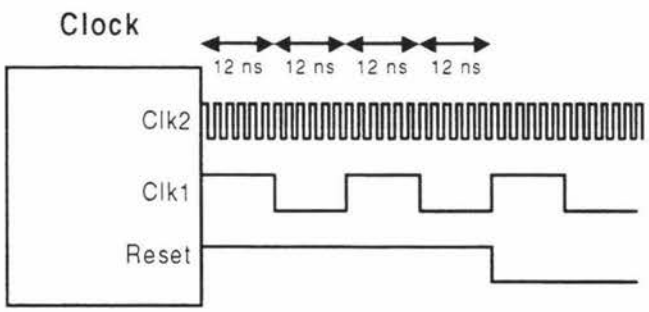


Figure 4.2: Signals of Clock Module

The others output signal of clock is *clk1* and *clk2*. Clk1 and Clk2, which continuously output a sequence of alternating logic "1"s and "0"s (see Figure 4.3). In ATMSWITCH, the *clk1* cycle was defined as 12ns for high phase and 12 ns for low phase. The *clk2* cycle was defined as 500 ps for high phase and 500 ps for low phase. When the *clk1* signal is high, the controller will change the switch state into the *Acceptcell*. When the *clk1* signal change to low, the controller will change the switch state to the *outgoing* cell. The *clk2* signal is used to give the timing control for the circuit modules of ATMSWITCH.

4.4 Input Port Module

The input port captures cells arriving *via* the ATMSWITCH's serial input line and stores them in a single-cell input buffer. The buffer automatically asserts the *full* signal when a cell is loaded into it. The *full* signal will be cleared by the controller after the cell is moved into the cell memory. In simulated ATMSWITCH module, four input ports are defined to receive cells from input line of switch. They are *Input Port 0#, Input Port 1#, Input Port 2#, Input Port 3#*. All input ports are designed with same circuitry function. The name Input Port is thus used to represent each input ports.

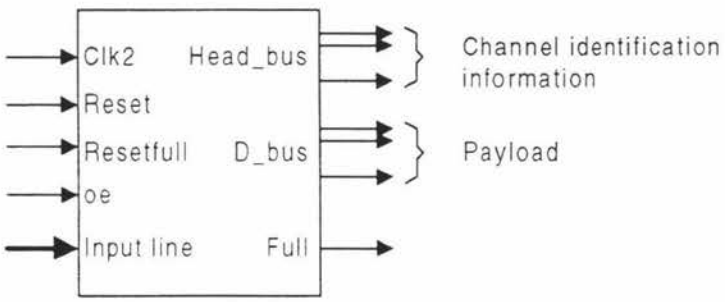


Figure 4.3: External Port Diagram of Input Port

The external ports diagram of the simulated input port is shown in Figure 4.3. There are several ports in input port. The function of each port can be described below:

- **Reset Port**: receives *reset* signal from the Clock module. When the *reset* signal is high, the Input Port will clear the current cell in the Input Port and reset the *full* signal as low.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the input port operations.
- **Oe Port**: is connected with the *en_port_no* signal of input round robin server. At *acceptcell* state, if the input port is selected by input server to send its cell to cell memory, the oe port will receive an identical number which is correspond to its port number. For example, signal value "0001" is corresponding to input port 1#. If the input port is enabled, it will put the cell onto *d_bus* and *head_bus* .

- **D_Bus Port & Head_bus Port**: the *d_bus* is cell payload data bus and its width is equal to cell payload data length. The *head_bus* is cell head data bus and its width is equal to cell head length. In ATM switch, the cell head is defined as 5 bytes and cell payload data is 48 bytes. In VHDL simulated ATMSWITCH, the *d_bus* is defined as 8 bits width and *head_bus* is 28 bits. Both *d_bus* and *head_bus* are used to transfer cells within ATMSWITCH. The bus is shared with switch modules. For simple reason, the cell payload data bus was called **cell data bus** and cell head data bus was called **cell head bus**. Sometime the **cell bus** was used to represent both cell data and cell head bus. The *d_bus* port of input port is used to send cell data to the cell memory and the *head_bus* port is used to send cell head.
- **Full Port**: outputs *full* signal. When a cell is load into input port, the *full* signal is automatically set to high. After the cell is transferred to cell memory, the *full* signal is reset to low.
- **Resetfull Port**: is connected with *resetfull* signal of controller. After the cell is transferred from input port to cell memory, the *resetfull* signal is set to high. The *full* signal is reset to low when the high *resetfull* signal reaches input port.

4.5 Input Round Robin Server Module

The input ports have a round robin server to indicate current input port number and update current port number on each accepting state. The input round robin server indicates which input port is going to send its cell to cell memory at current accepting cell state. It also outputs a signal to enable selected input port to put its cell onto the cell bus.

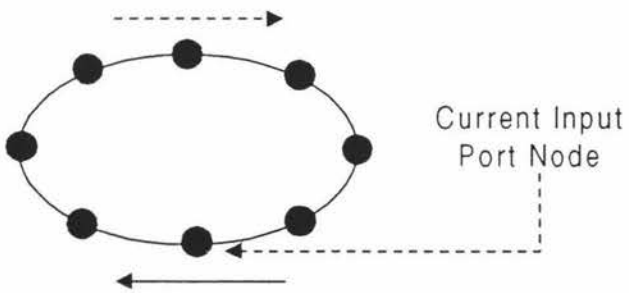


Figure 4.4: Structure of Input Round Robin Server

When an input port is loaded a cell, the flag signal *full* is set as high. The flag will be reset by the switch controller after the cell has been sent to the cell memory. The *full* signal for each input port is represented by *full00*, *full01*, *full02* and *full03* in simulated ATMSWITCH. They represent the *full* signals of input port from 0# to input port 3#. All of the full signals are checked by the input round robin server, which select one of the input port to send its cell to the cell memory. As Figure 4.4 shown, the Input Round Robin Server actually maintains a circular list structure for input ports. Each node of list is identified with a unique input port number. At each accepting cell state, the server check the *full* signal of current input port. If the *full* signal of current input port is high,

the server will select this port and output the *en_port_no* signal to the port so the cell of this port can be transferred to the cell memory.

If the *full* signal of the current input port is low, the server moves forward to next node and checks the *full* signal value. If the *full* signal of checked input port is still low, the server continue to visit the rest of the nodes in the circular list until either one of the *full* signals is high or all of nodes have been visited. So the input server make switch work efficiency at each *acceptcell* state. This is why the input round robin server is said to perform a supersaturated operation mode.

Therefore the operation of input round robin server is not only used to encode the current input port, but also ensure that the input ports are served efficiently and fairly. That is, the switch is able to accept everything produced by an input port which is dumping data into it at full speed, if that is the only loaded port, but if other ports are loaded, their data will be accepted with a priority equal to that of the "dumping" port.

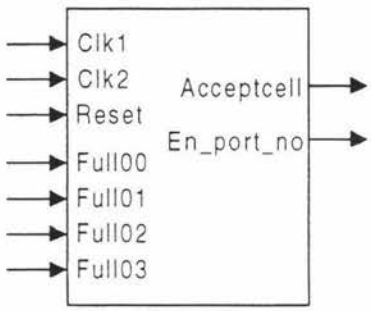


Figure 4.5: Input Round Robin Server Port Diagram

The external ports of the simulated input round robin server are shown in Figure 4.5. There are several defined ports in input round robin server module. Their functions are described below:

- **Full Port**: there are four full ports are in input server module. They are from *full00* to *full03*. Each full port is connected with the *full* signal of input port. If the input port is loaded a cell, the *full* signal is automatically set to high. Input server use these full ports to check the *full* signals of input port.
- **Clk1 Port**: receives *clk1* signal from Clock module. When the clk1 signal is changed from the low to high, the input server starts its encoding operation. When the *clk1* signal is changed from high to low, the input server stops its operation and wait next high phase signal is coming.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the input server operations.
- **Reset Port**: receives *reset* signal from Clock module. When the *reset* signal is high, the input server will back to initialized state.
- **En_port_no Port**: outputs the enable signal to the selected input port to send its cell to cell memory. For example, if the *en_port_no* output the value of "0001", the input

port 1# is enabled; if the *en_port_no* value is "0010", the input port 2# is enabled. Each *en_port_no* value is corresponding to an identical input port number.

- **Acceptcell Port**: notifies the controller of switch or other related circuit modules that the cell is ready to transfer from input port to cell memory. The *acceptcell* signal is set to high after the input server examined the *full* signals of input port, in which the input port is not empty.

4.6 Counter Module

The counter outputs a series increased integer number at each counter cycle. The number compares with the ITV value and *mask* value at each *outgoing* cell slot. If the value is matched, one of CBR/VBR channel is triggered and the matching CBR/VBR channel is due for output, so one of its cells should be sent to the output port. The counter is a part of our designed trigger mechanism.

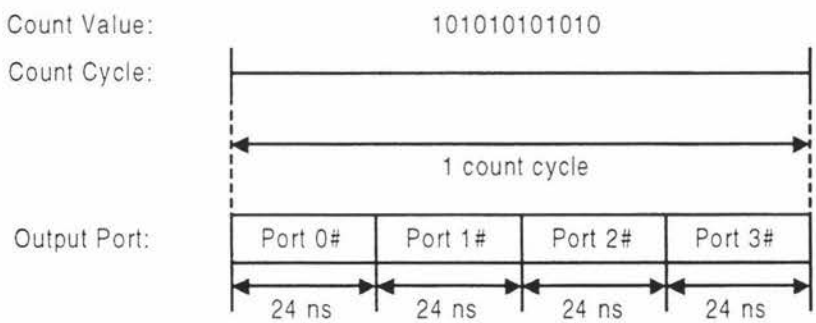


Figure 4.6: One counter cycle equal to four clock cycle

In simulated ATMSWITCH, the counter is triggered at each four *clk1* cycle. As Figure 4.6 shows, the counter number "1010101010" is checked by all output ports from 0# to 3# in their turn to *outgoing* cell. That means the same counter number can be used to trigger CBR/VBR channels that belong to different output port. It's shared with the output ports. Two CBR/VBR channels might have same ITV, *Mask* and counter number range if they are not routed at same output port. The reason of the counter cycle was designed as four *clk1* cycle is due to four output to be used in simulated ATMSWITCH module. The detail discussion is given at later sections of this chapter.

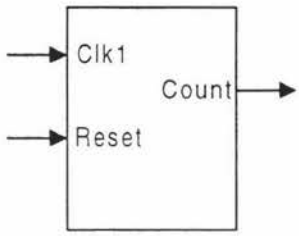


Figure 4.7: Counter Port Diagram

The external ports diagram of simulated counter is shown in Figure 4.7. It's quite simple circuit module in ATMSWITCH. Each port functions is described at below:

- **Reset Port**: receives *reset* signal from Clock module. When the *reset* signal is high, the counter backs initial state and begin to count from the zero.
- **Clk1 Port**: receives *clk1* signal from the Clock module. It triggers counter to increase its current output number. The counter is triggered at each four times *clk1* cycle.
- **Count Port**: outputs current counting number to map memory. The number is regularly increased at each counter cycle.

4.7 Controller Module

The controller consists of a control circuit and a set of control signal to control and adjust each circuit operations. The switch can thus transfer a cell from the input port into cell buffer at *acceptcell* state and send a cell from the cell buffer to the output port at the *outgoing* state. The operation of controller can be described as these two states.

Acceptcell State

When the controller receives high phase of *clk1* signal, the controller begins to work at *acceptcell* state. First controller waits for *acceptcell* signal event that is from input round robin server. If the *acceptcell* signal is high, it means the input port is ready to transfer cell onto data bus, the controller will thus prepare to receive the cell head information from *head_bus*. After the input port put the cell head information onto the *head_bus*, the controller fetch the data from the *head_bus* and check the cell head information. If the checked cell head is a normal cell type, the controller set the *write* signal as high so that the cell memory is enabled to write the cell data from the cell data bus into the cell memory.

If the incoming cell head was examined as Setup cell type, it means the incoming cell is used to establish the new virtual channel. The controller will thus perform a series of operations for building the new virtual channel. There are two different operations involved in creating the new channels. One is for building CBR/VBR channels, another is for building the ABR/UBR channels. In both cases, the controller set the *setupcell* signal as high so the switch start to create a new channel at this *acceptcell* state.

If the incoming cell head was examined as Release cell type, it means the incoming cell is to release a no longer used channel. The controller will perform related operations to clear the given virtual channel. In this case, the controller set the *releasecell* signal as high so the switch start to remove the virtual channel from the channel information list.

In order to assign a bandwidth to the new requested CBR/VBR virtual channel, the controller maintains several ITV trees for managing bandwidth of output ports. Each output port corresponding to an ITV tree. If a setup cell is coming to request a CBR/VBR channel with a predefined speed and routing destination, the controller check the ITV tree of referred output port. If requested bandwidth is available in the ITV tree, the controller builds the new channel and updates the ITV tree. The ITV tree is part of trigger mechanism. The details about ITV tree and trigger mechanism is described at the rest of this chapter. Here we just use the concept of ITV tree to explain the functions of the controller.

The ITV tree is also updated when a CBR/VBR channel is released from the switch. When controller has examined the type of incoming cell to be a Release cell type, it will find the released channel in the ITV tree and removed it from the tree.

The ITV tree is a special binary tree structure. Each CBR/VBR channel has a node in the tree. The operation of setup a new channel in the controller is actually finding the most appropriate position in the tree and adding the new channel node into the position. The operation of releasing a channel is actually removing a channel node from the tree.

Outgoing Cell State

When the controller receives a low phase *clk1* signal from Clock module, it begins to have *outgoing* cell operations. First controller check *outgoing* and *match* signal event. The *outgoing* signal is from output round robin server. It's set to high means that the current chandelier buffer of output port is not empty. The *match* signal is from map memory. It's set to high means that a CBR/VBR channel is triggered at this *outgoing* state. Whatever the *outgoing* or *match* signal is high, the controller will set the *read* signal to high. So the cell memory is enabled to read cell from memory onto the *d_bus*. Meanwhile, the map memory output the re-assembly cell head onto the *head_bus*. The currently output port fetches the cell from both *d_bus* and *head_bus* and then send the cell to the output line of switch.

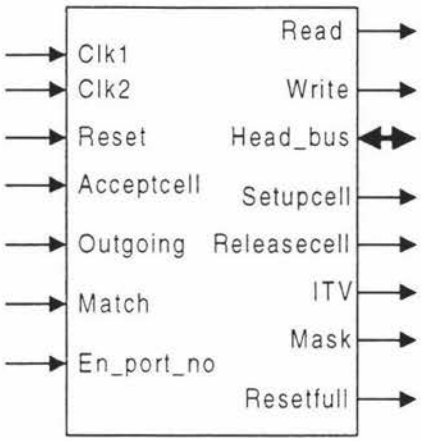


Figure 4.8 Controller Ports Diagram

The external ports diagram of simulated controller is shown in Figure 4.8. Each functions of controller port is described below:

- **Clk1 Port**: receives *clk1* signal from Clock module. When the *clk1* signal is changed from the low into high, the controller controls the switch to start working at *acceptcell* state. When the *clk1* signal is changed from high to low, the controller control the switch begin to work at *outgoing* cell state.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the controller operations.
- **Reset Port**: receives *reset* signal from Clock module. When the *reset* signal is high, the controller sets each signal as initial state and clear all ITV trees.
- **Acceptcell Port**: the *acceptcell* port is connected with the *acceptcell* signal of input server. At *acceptcell* state, if the controller get the high phase of *acceptcell* signal, it will fetch the cell head value from the *head_bus* and check the type of cell head. According to the checked cell type, the controller either set the *write* signal as high for transferring cell from input port to cell memory or set the *setupcell* signal as high for establishing a new channel or releasing an old channel.
- **Outgoing Port**: is connected with the *outgoing* signal of the output server. At *outgoing* cell state, if the controller receives a high phase of *outgoing* signal, the controller will set the *read* signal as high. The cell memory is thus enabled to read the cell.
- **Match Port**: is connected with the *match* signal of the map memory. At *outgoing* cell state, if the controller receives a high phase of *match* signal, the controller will set the *read* signal as high. The cell memory is thus enabled to read the cell.
- **Head_bus Port**: transfers cell head information within switch. The controller uses the *head_bus* port to fetch cell head after it receives the high phase *acceptcell* signal. The value of the cell head determines what kind operations the switch will do. If the cell head is a setup cell, the controller will establish a new channel at switch and the re-assembly cell head will be sent to the map memory through the *head_bus* port.

- **Setupcell Port**: outputs the *setupcell* signal to other modules of switch. If the controller identified the incoming cell is setup cell, the *setupcell* signal is set high. The high phase of *setupcell* signal disable the cell memory writing cell operation and notify map memory to create a new channel item in the channel information list for the new channel.
- **Releasecell Port**: the function of the releasecell port is same as setupcell port. The distinction is that the high *releasecell* signal releases an old channel item from channel information list.
- **Write Port**: outputs the *write* signal to other modules of switch. If the controller identified the incoming cell is normal cell, the *write* signal is set to high so that the cell memory is enabled to write a cell data into memory.
- **Read Port**: outputs the *read* signal to other modules of switch. If the controller receive either the high phase *outgoing* or *match* signal, the *read* signal is set to high so the cell memory is enabled to read the cell.
- **Resetfull Port**: outputs the *resetfull* signal to input port. After the cell is transferred from input port to cell memory, the *resetfull* signal is set to high so the full signal of input port is reset to low.
- **ITV Port**: outputs ITV value to map memory. If the incoming cell is identified as setup cell, the controller will travel ITV tree to find an available position for request CBR/VBR channel bandwidth. After the controller insert the node in the tree, the controller outputs the ITV value to map memory through this port.
- **Mask Port**: outputs *mask* value to map memory. Same as ITV port, if the incoming cell is identified as setup cell, the controller insert a channel node in the ITV tree. According to the ITV value and given channel bandwidth, the *mask* value can be calculated by using predefined algorithm. The *mask* value is sent to the map memory through this port.

4.8 Map Memory

Map memory maps incoming cells onto a predefined cell queue in the accepting cell state. In the *outgoing* state, the map memory use two special mechanisms; trigger mechanism and chandelier round robin server, to determine which channel is selected to send its cell onto output port.

Map memory maintains a list of routing information for each established channel. Each item of the list consists of a channel number, the input port number, output port number and other related fields. When a cell is coming from one of input port, its channel number and input port number will be treated as a key and compared with the routing information list. When the key is identified, the cell will be appended at the pre-assigned chandelier cell queue. If incoming cell is a Setup cell, the map memory will create a newly channel item in the information list for the new established virtual channel.

Map memory has a well-defined data structure. The most part of trigger mechanism and chandelier is implemented by map memory. In the *outgoing* state, the trigger mechanism will check whether there are any CBR/VBR cell waiting for output. If no CBR/VBR cells

are due to be sent, the chandelier will be used to choose a cell for the output port. The item of map memory is inserted and removed in terms of creating the new channel and delete the old channel operations.

In remaining sections, the details about map memory and chandelier will be described.

4.8.1 Structure of a Word in Map Memory

In Figure 4.9, the data structure shows each data fields in map memory. The definition about each data field is described below.

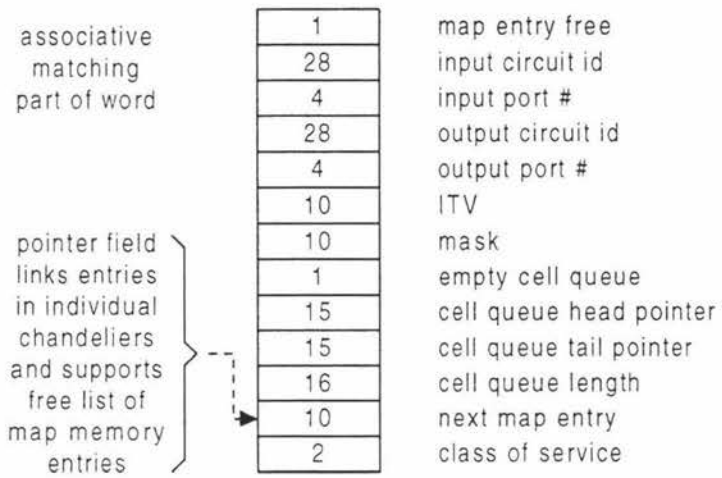


Figure 4.9: Data storage fields in map memory

- **Map Entry Free**: specifies whether the current address in map memory is used or not. If the field is set to '1', this address of map memory is in use. The '0' means the address of memory is free.
- **Input Circuit id**: represents the Virtual Path Identifier and Virtual Channel Identifier of the cell header. When a new virtual channel is established, the map memory will store the VPI/VCI in the input circuit id field of a free word in map memory.
- **Input Port Number**: contains the port number of the port through which the cell arrived at the switch. Channels associated with different ports are not required to have unique circuit ids (VPI/VCI information), so the channel's port number must be stored as well as its circuit id if incoming cells are to be unambiguously assigned to their channels.
- **Output Circuit Id**: contains the VPI/VCI information used on the channel's next hop. When an *outgoing* cell is assembled, the values in this field are placed in the VPI/VCI fields of the cell header.

- **Output Port Number**: the port through which cells belonging to this channel should be output.
- **ITV** (Initial Trigger Value): A number allocated to the channel by the switch controller to CBR and VBR channels when they are established. Output from the channel is scheduled whenever as *masked* match (see next field) occurs between the trigger counter and the channel's ITV.
- **Mask Field**: A number defining the relationship between the channel's bandwidth requirement and the bandwidth of the data communication link. In the extreme case, if the channel requires 100% of the link bandwidth, the *mask* value is all 1's, and any counter matches the ITV (irrespective of its value). If the bandwidth requirement is 50% or 25%, then the *mask* has all 1's except for the bottom 1 or 2 bits respectively, and thus the counter value will match the ITV once every two or four cycles.
- **Empty Cell Queue**: specifies whether the cell queue is empty or not. Each cell queue belongs to one cannel.
- **Cell Queue Head**: points the first cell address of cell queue in cell memory. During the *outgoing* cell state, whatever the channel is selected by either the trigger mechanism or chandelier output server, the cell queue head pointer will give the address of the first cell in the channel's queue so that the cell will be send to the output port.
- **Cell Queue Tail**: points the address of the last cell of cell queue in cell memory. During accepting cell state, the cell is appended at the end of cell queue. The cell queue tail pointer is used to give the last cell address of cell queue. This avoids the need to traverse the entire list when inserting a new cell, so that it is easy to determine when to add a cell header into, or remove it from, a chandelier.
- **Cell Queue Length**: records the length of cell queue.
- **Next Map Entry**: links cell queue headers together into the circular list that gives the chandelier its characteristic shape in map memory. All channels that appear on the same chandelier are routed through the same output port. Each output port maintains a chandelier that selects a channel and outputs its cell via this output port. The chandelier structure is designed as a dynamic data structure. If no cell is queued in the channel's cell queue, the channel header will be removed from chandelier. If a cell arrives at this channel, the channel header will be added into the chandelier again.
- **Class of service**: indicates the type of the channel, such as CBR, ABR, VBR or UBR.

4.8.2 Chandelier Structure

From described above, we know that the next map entry field of data structure is used to link channel entries and form circular lists for each output port in map memory. The chandelier is dynamic data structure. Figure 4.10 shows the structure of chandelier. In this diagram, the channel headers which have cells buffered in the cell memory appear in the circular list. Those channel headers which aren't linked at the circular list are no any cell in the cell memory. If the empty queue field of channel header is changed from false to true, it means the last cell has been removed from the cell queue of the channel, the

channel header will be removed from the circular list. If a cell arrives to the channel that wasn't linked at the circular list, the channel header will be inserted at the circular list. So the circular list is dynamic formed by channel headers.

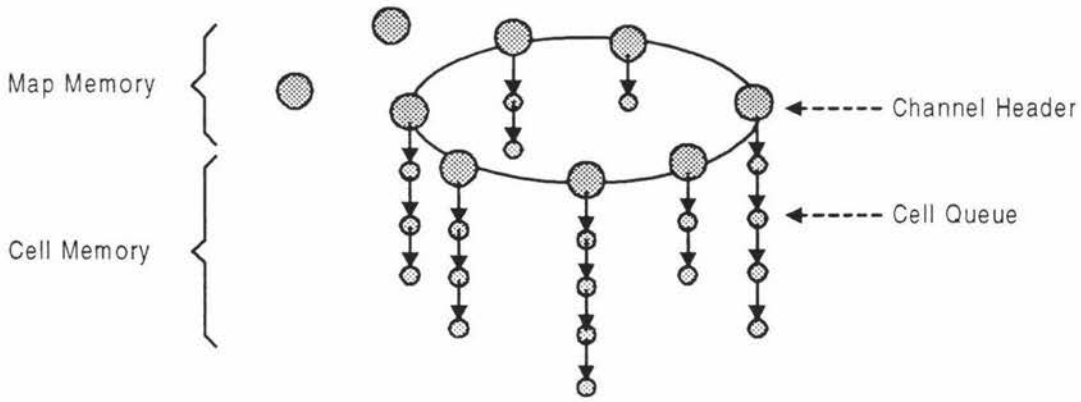


Figure 4.10: Chandelier Structure: Channel Header Circular List and Cell Queue

The diagram also shows the cell queue structure of chandelier. The first cell of the cell queue is pointed by the cell queue head field of channel header. The last cell is pointed by the cell queue tail field of channel header. Each time the switch enters the *outgoing* state, the chandelier server will select one of channel from circular list node. At the next *outgoing* state, it will advance a pointer round the circular list to choose a cell from the queue belonging to the next channel in the sequence. The structure of circular list is like a linked list. It needs a pointer that always points current chandelier node. We thus employ another circuit component, **Chandelier Register** to manage current chandelier entry for each output port. Figure 4.11 shows the chandelier register and the chandelier structure work together for accessing the current channel in map memory.

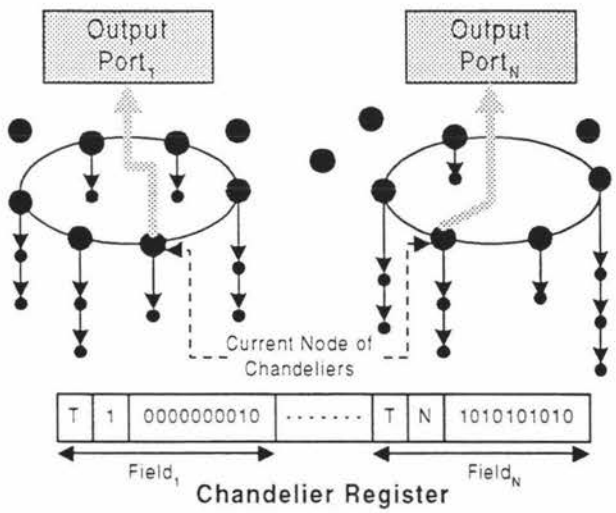


Figure 4.11: Chandelier and Chandelier Register

In above diagram, the chandelier register have N (where N is the number of output ports associated with the switch) pieces of field and each of them has three subfields. The *Field i* in the diagram indicate the map memory entry of current channel node in *output port i's* chandelier. The definition of each subfield is described below:

- The first subfield is a Boolean. It is set to true if the chandelier has any cells queued for output; it will be set to false if the chandelier buffer is empty. Actually the chandelier is a dynamic linked system. If there are any cell in the buffer, the chandelier should be established except CBR/VBR cell.
- The second subfield of chandelier register represents each output port number. This means the chandelier that belongs to which output port.
- The third subfield is used to store the address in map memory of the current channel header for that chandelier. This entry will be used for finding current channel information in map memory. Through accessing the channel header information, the switch will find the first item of the cell queue by using the cell queue head pointer field value of this channel. After the first cell is output, the channel header information will be updated, such as queue length, cell queue head pointer and so on. At the same time the chandelier register will be updated. Normally the **current map entry** field will be overwritten with the address in the current map entry node's **next map entry** field.

So the chandelier register always keeps the current chandelier entry for each output port. It is used to access the current *outgoing* cell.

4.8.3 Implementation of Map Memory

From the above description, map memory maintains a routing information record for each established channel. Each field in the record consists of a channel number, input port number, output port number and other related fields. When a cell is coming from one of input port, its channel number and input port number will be treated as a key and compared with routing information record. When the key is identified, the cell will be mapped onto a cell queue that is pointed to by a channel header node of the chandelier structure. If incoming cell is a Setup cell, the map memory will create a record for the newly established virtual channel. At the *outgoing* state, the ATMSWITCH uses trigger mechanism and chandelier to determine which cell should be sent onto output port. Figure 4.12 shows the three major circuit components module that make up the Map Memory.

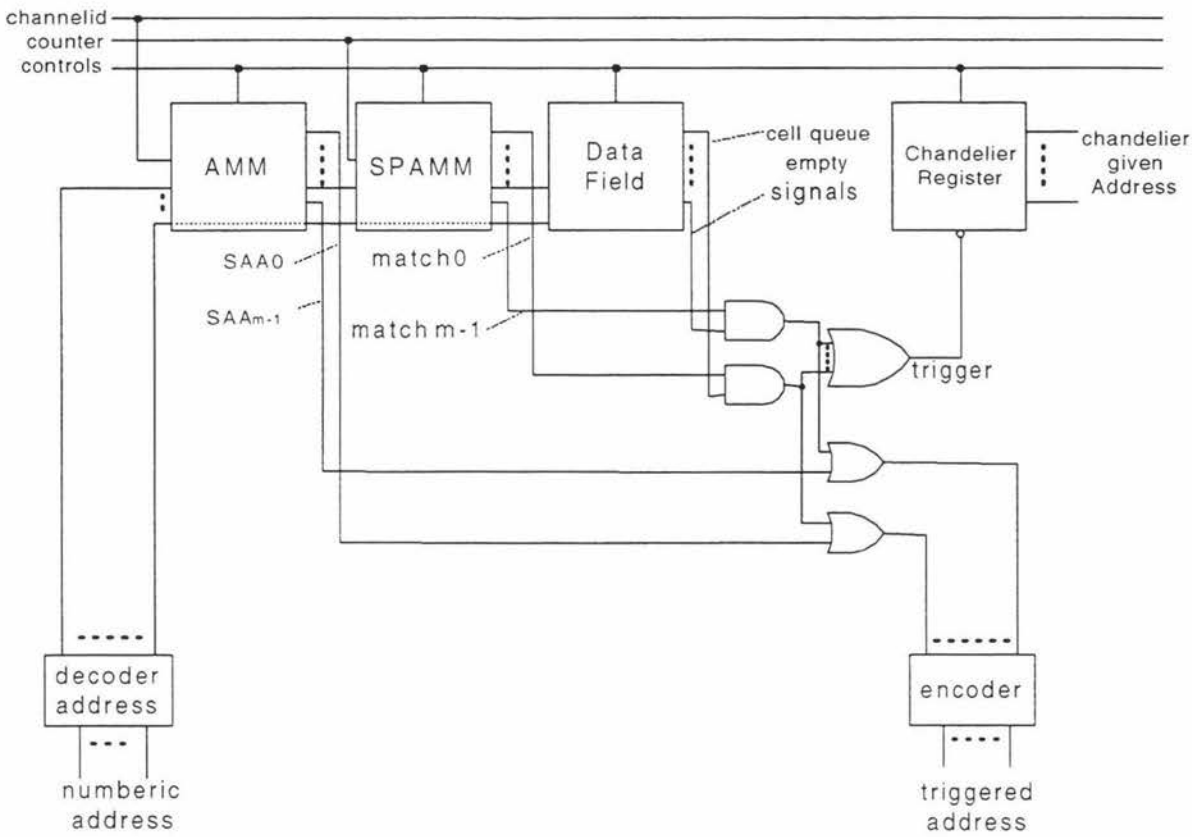


Figure 4.12: Map Memory circuit components module

In above diagram the map memory is divided into three modules. When *clk1* signal is high (Accepting cell state), the input circuit id of the incoming cell will be treated as a key and compared with the channel numbers and input port numbers recorded in all the associative memory locations of map memory. The module that performs this function is AMM (Associative Memory Module) module in the diagram. The numerical address of the location in the memory that matches the incoming cell's identification is used subsequently to identify its map memory entry. At *outgoing* state, the *clk1* signal goes low; the counter number is compared with the ITV and *mask* value in SPAMM (Special Purpose Associative Memory Module) module. If a channel matches and the cell queue of the channel isn't empty, its numerical address is returned, and used to access the location while the output cell is assembled. If no channel matches the counter value, the trigger signal should be zero. The chandelier register will thus be enabled to output its current map memory entry and the decoder will select the channel by using given map memory entry. In the case of diagram, the selected channel comes from the chandelier.

In the map memory diagram, the three major circuit modules consist of map memory. The AMM module maps channel id of incoming cell at *acceptcell* state and the SPAMM module matches counter number with ITV and *mask* value at *outgoing* cell state. The Data Field module stores remaining information of channel header except AMM and

SPAMM. The chandelier register keeps current map memory entry for chandelier. The Data Field and chandelier register components in map memory are simple circuitry modules. Below I thus introduce the AMM and SPAMM circuit modules only.

AMM (Associative Memory Module)

According to paper [Lyons, 1996], the AMW module has been designed and is represented by circuit notation. It's shown in Figure 4.13.

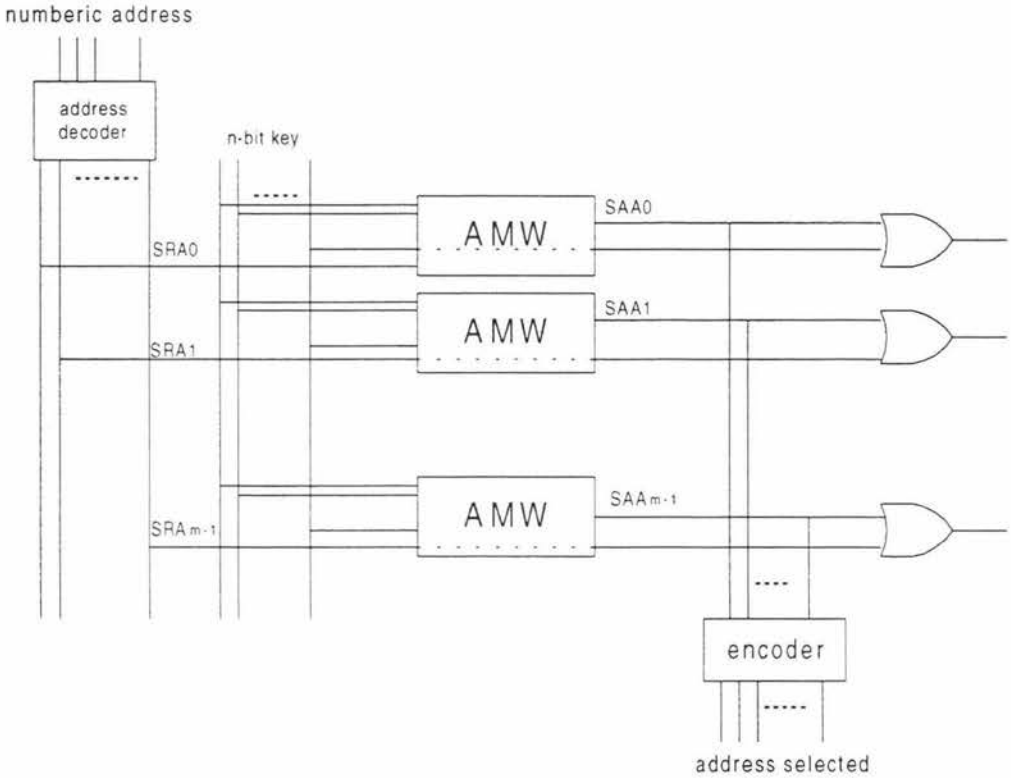


Figure 4.13: m-word associative memory

In above picture, the n-bit key is channel circuit id and input port number, it compares with all contents of AMW (Associative Memory Word) simultaneously. The AMW is contents matched memory word and it's a basic unit to make up of AMM. An AMM is stored the part of a channel information record. It contains the channel id and input port number. Each AMW compares with the key at the same time. If one of AMW content is matched with the key value, the SAA_i signal for this AMW is changed to high. The incoming cell is thus mapped to this channel.

SPAMM Circuit Module

The SPAMM matches the counter number with ITV and *mask* value of the map memory. Like AMM module, the SPAMM is also designed as an associative memory module. The ITV and *mask* value of a CBR/VBR channel is stored at an SPAMM-Word in SPAMM.

At each *outgoing* state, the counter number compares with all SPAMM-Words at the same time. If one of match signal of SPAMM-Word is changed to high, the channel is selected. The basic unit of the SPAMM is shown in Figure 4.13a. It's described by using circuit notation.

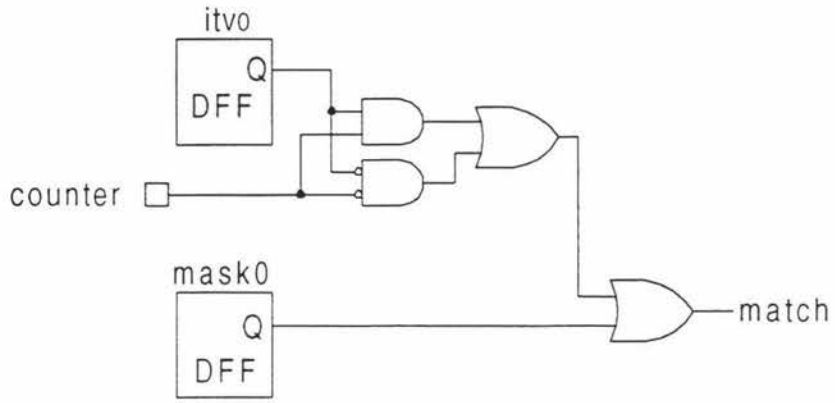


Figure 4.14: Single-bit of SPAMM (SPAMM-Unit)

In Figure 4.14, the signal bit value of ITV, *mask* and counter are compared. Either *mask* value is "1" or ITV value equal to counter value, the output signal match will be "1". N (ITV width plus *mask* width) piece of SPAMM-Units consists of a SPAMM-Word. A SPAMM-Word is described in the following diagram.

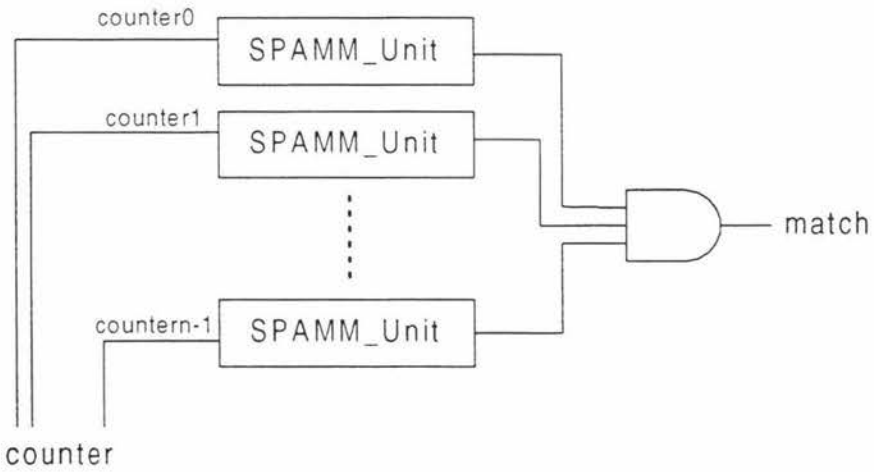


Figure 4.15: n-bit SPAMM-Word

The diagram shows that only all output results of SPAMM-Units is "1", the *match* signal is high. The high value of *match* signal means this CBR/VBR channel is matched. The complete SPAMM module is shown in Figure 4.16. This diagram shows the SPAMM is designed as content memory module. At each *outgoing* cell state, all words of SPAMM compare with the counter number. If one of *spamm* signal is high, the *match* signal is high.

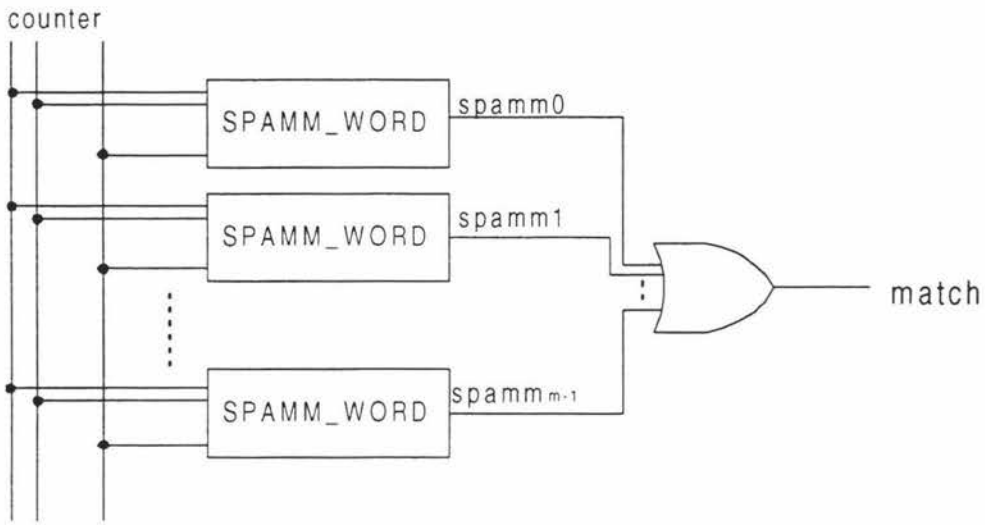


Figure 4.16: Counter, ITV and *Mask* matching module: SPAMM

Simulated Module of Map Memory

The external port diagram of simulated map memory is shown in Figure 4.17. There are several defined ports in map memory. The definition of each port is described below:

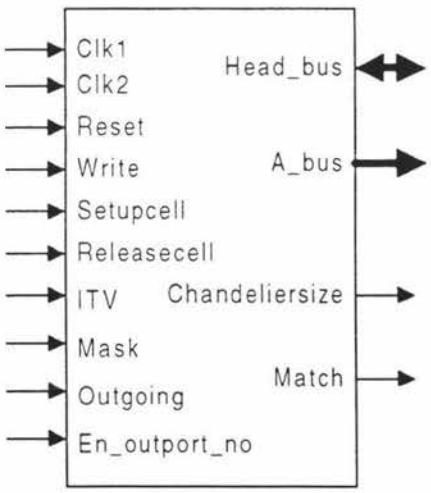


Figure 4.17: Map Memory Ports Diagram

- **Clk1 Port:** is connected with the *clk1* signal of clock module. When the *clk1* signal is changed from the low to high, the map memory starts to accept cell operations. According to the value of *write*, *setupcell* and *releasecell* signals, the map memory either maps the incoming cell channel or creates a new channel or releases an old

channel in the map memory. If the *clk1* signal is changed from the high to low, the map memory matches the counter number with each ITV value and *mask* value of CBR/VBR channel. If the value is matched, the *outgoing* cell is from the matched channel. If the value is not matched, the *outgoing* cell is from the chandelier buffer.

- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the map memory operations.
- **Reset Port**: is connected with the *reset* signal of controller module. When the *reset* signal is changed to high, the map memory clears all channel information records in map memory. The map memory is reset to initial state.
- **Write Port**: is connected with the *write* signal of controller module. At *acceptcell* state, if the *write* signal is changed from the low to high, the map memory read the cell head from the *head_bus* and map the channel number of cell head with the input circuit id field of the map memory. If the channel is identified, the several fields of the mapped channel information record could be updated (Note: the detail description of updating map memory algorithm is introduced at last section of this chapter). If the channel is not matched, the switch will discard the incoming cell.
- **Setupcell Port**: is connected with the *setupcell* signal of controller. At *acceptcell* state, if the *setupcell* signal is changed to high, the map memory will create a new channel record in the map memory. All parameters for creating the new channel is created by controller. It puts these values onto the *head_bus*. The map memory read those data from the *head_bus* after it receives the high phase *setupcell* signal.
- **Releasecell Port**: is connected with the *releasecell* signal of controller. At *acceptcell* state, if the *releasecell* signal is changed to high, the map memory will release an old channel record from the map memory.
- **ITV Port**: is connected with the *ITV* signal of the controller. At *acceptcell* state, if the new established channel is CBR/VBR channel, the map memory will receive the *ITV* value from the ITV port and then store it at the ITV field of new channel header.
- ***Mask* Port**: is connected with the *mask* signal of controller. Same as ITV port, the map memory will get the *mask* value from the *mask* port and store it into the *mask* field of the newly CBR/VBR channel header.
- ***Outgoing* Port**: is connected with the *outgoing* signal of the output server. When the map memory receives the high phase *outgoing* signal, it selects a channel from the chandelier circular list and outputs the cell queue head pointer onto *a_bus*. The cell memory uses this address to read the first cell of cell queue.
- **En_outport_no Port**: is connected with the *en_outport_no* signal of output server. When the map memory detects the low phase *clk1* signal, it gets the current output port number from the en_output_no port and matches the counter number with ITV and *mask* value. If the value is matched and the matched CBR/VBR channel is located at current output port. The channel is triggered and its cell is sent to output port. The channel is triggered only the channel is routed to the current output.
- **Head_bus Port**: is connected with the cell head bus. At *acceptcell* state, the map memory read cell head from the *head_bus*. At *outgoing* cell state, the map memory outputs the re-assembly cell head through this port.
- **A_bus Port**: is connected with the cell address bus. At *acceptcell* state, the map memory read the incoming cell address from the *a_bus* to update the cell queue head

and cell queue tail field of channel header. At *outgoing* cell state, the map memory outputs the cell queue head pointer of selected channel onto *a_bus* for the cell memory reading cell.

- **Match Port**: indicates that a CBR/VBR channel is triggered. When the *match* signal is high, the *outgoing* cell is taken from the triggered CBR/VBR channel rather than chandelier buffer.
- **Chandeliersize Port**: outputs the first subfield value of chandelier register. The output server will use this value and *free* signal of output port to select an available chandelier.

4.9 Trigger Mechanism

The trigger mechanism is designed to provide CBR/VBR channel's cells with the regular bit rate to pass the ATMSWITCH. The general idea of the trigger mechanism is to schedule cell output for a particular channel at regular intervals, to maintain a constant end-to-end delay within the network. Due to all cells of each CBR/VBR channel pass switch with a regular intervals, the delay for these cells are almost same. The jitter can thus be ignored.

In last section, the SPAMM circuit module of map memory was described. The SPAMM lets the counter number compare with ITV and *mask* value of CBR/VBR channel. If the value is matched, a CBR/VBR channel is triggered and the *outgoing* cell is taken from the matched channel rather than chandelier. If the compared value is not matched, the *outgoing* cell is taken from chandelier buffer. Figure 4.18 shows the trigger mechanism and chandelier selecting a channel for *outgoing* a cell.

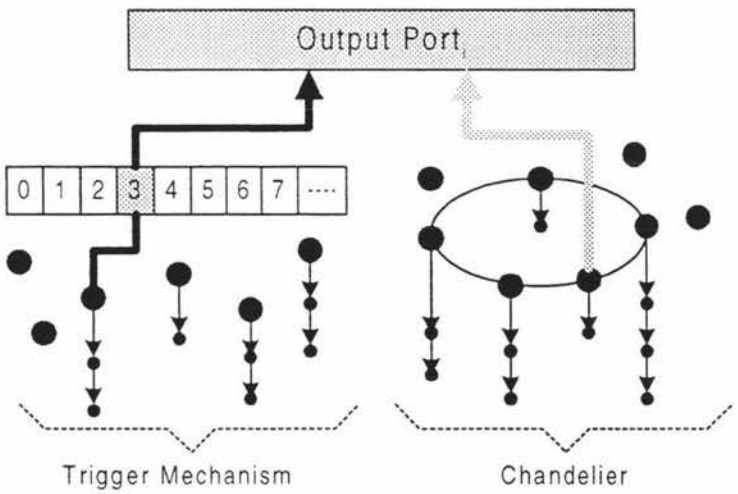


Figure 4.18: Trigger Mechanism and Chandelier Structure

For each newly established CBR/VBR channel, the controller will assign an ITV value and *mask* value. The ITV and *mask* value are stored in the channel's record in map memory. The value of ITV and *mask* are generated by a special control algorithm based on the proportion of total link bandwidth required by the virtual channel. The counter regularly increments at each counter working cycle. So an output slot for the channel occurs regularly, and it is triggered and a cell sent output the output port. Thus the trigger mechanism provides the real-time quality of service required for voice.

4.9.1 Principle of Trigger Mechanism

As described before, the trigger mechanism compares the counter number with the ITV and *mask* value of each channel. If match between the two values occurs, the matched channel will send its cell onto the output port. The circuit module which implements the trigger mechanism was introduced earlier in the description of the SPAMM module of the map memory. However, we have not discussed the reasons why the ITV and a *mask* value contain sufficient information to determine whether any arbitrary counter value should trigger a CBR/VBR channel; what the relationship is between these three values, how to assign ITV and *mask* value for a CBR/VBR channel. In this section the theory behind the trigger mechanism will be introduced.

The principle of the trigger mechanism is to schedule cell output for a particular channel at regular intervals, to maintain a constant end-to-end delay within the network. The trigger mechanism uses a regularly incremented counter as the core of its scheduling algorithm [Lyons, 1997]. Each CBR/VBR channel is associated with a unique set of equally-spaced values in the counter's range and will be triggered whenever the counter value equals one of these trigger values. The spacing between triggers will be inversely proportional to the channel's speed, so the high-speed channels will be triggered frequently, low-speed channels less frequently. If the channel is triggered, but its buffer queue is currently empty (as would often be case for VBR channels), then access to the output port will default to the ABR/UBR channel, then output will also be able to come from the chandelier. Spacing triggers appropriately for each channel's bandwidth, and ensuring that triggers for two or more channels do not occur simultaneously is a function of the switch control software which is executed at channel establishment time.

The following example shows how a range of counter values can be assigned to trigger CBR/VBR channels with particular (and differing) speed requirements. Suppose a switch currently supports 4 channels, A to D, all destined for output via the same port, where:

- A is a CBR channel which requires ¼ of link bandwidth of the output port (Note the ¼ link bandwidth means the required bandwidth take ¼ of the total bandwidth. For example, if the total bandwidth is 1.2Gbps, the ¼ bandwidth will be 300Mbps. In this example, the channel A will be assigned 300Mbps bandwidth)
- B is a VBR channel which requires a maximum of $^1/_8$ of link bandwidth (Note the channel B will be assigned 150Mbps bandwidth, if the total bandwidth is 1.2Gbps.)
- C is an ABR channel
- D is a UBR channel

If we use a four-bit counter, the total range of the counter is from 0 to 15. Because the A takes ¼ of the output port bandwidth, A could be triggered by counter values 0, 4, 8, 12, 0, 4, 8, 12, ..., "using up" four output slots at each counter cycle. And B could be triggered by counter values 1, 9, 1, 9, "using up" two output slots, (or 2, 10, 2, 10,..., or 3, 11, 3, 11..., or 5, 13, 5, 13,... etc). The rest of the output slots will be used to serve channels C and D, for which the cell queues will be in chandeliers.

Whenever the counter generates a trigger value for channel A or B, these channels will be given unconditional access to the output port. If the channel has no cell buffered when this occurs, or if the counter output is not equal to a trigger value for channel A or B, then the channels buffered in the chandelier will be given access to the output port. Table 4.1 is a time-sequence showing what happens at successive output slots.

Table 4.1: Time-sequence showing output from a triggered node

Count	channel to output	Reason
0	A	count = 0 triggers output from channel A unconditionally
1	B	count = 1 triggers output from channel B unconditionally
2	C	Neither A nor B is scheduled for output, and the next entry in the chandelier's circuit list which the round robin output server will reach contains the channel header for channel C.
3	D	Neither A nor B is scheduled for output and channel D is the next entry in the chandelier's circular list
4	A	count = 4 triggers output from channel A unconditionally
5	C	Neither A nor B is scheduled for output and channel C is the next entry in the chandelier's circular list

6	D	Neither A nor B is scheduled for output and channel D is the next entry in the chandelier's circular list
7	C	Neither A nor B is scheduled for output and channel C is the next entry in the chandelier's circular list
8	A	Count = 8 triggers output from channel A unconditionally
9	D	count = 9 triggers output from channel B unconditionally, but B is a VBR channel with an empty buffer, so service defaults to D, the next channel to be reached by the chandelier's round robin output server
10	C	Neither A nor B is scheduled for output and channel C is the next entry in the chandelier's circular list. Because the cell queue of channel C becomes empty after the last cell has been output, the header of channel C is removed from the chandelier's circular list
11	D	Neither A nor B is scheduled for output and channel D is the next entry in the chandelier's circular list
12	A	Count = 12 triggers output from channel A unconditionally
13	D	Neither A nor B is scheduled for output and the header of channel C has been removed from the chandelier's circular list, so channel D is the next entry in the chandelier's circular list
14	D	Neither A nor B is scheduled for output and channel D is the next entry in the chandelier's circular list. Because the cell queue of channel D becomes empty after the last cell has been output, the header of channel C is removed from the chandelier's circular list
15	Idle	Neither A nor B is scheduled for output and the cell queues of channel C and D are empty, so no cell is output at this slot.

And begin to next counter cycle.

From the above description, we know the channel will be triggered by a set of counter values determined by the relationship between the channel speed and the link's total bandwidth. Below, I describe this relationship in detail. According to the paper [Lyons, 1997], we define two data fields (ITV, *Mask*) for each channel in the map memory. These will match a regularly spaced set of values in the counter range, which are to be

used as triggers for the channel. In the following section I will describe how ITV and *mask* values are selected for a CBR/VBR channel.

4.9.2 ITV Space and Mask

When a CBR/VBR channel is first established, the controller assigns it an ITV, its Initial Trigger Value – the lowest counter value which will trigger output from a particular channel – and a TS, its Trigger Separation, the channel will be allocated 1/TS of the output bandwidth and it will subsequently be triggered when the counter has values ITV, ITV + TS, ITV + 2 TS, and so on. This way of characterizing the set of trigger values for a channel - as an initial value and a separation is of more than academic interest; with the addition of a suitable *mask* value, it provides for a fast, low-storage way of determining whether the current counter value is the trigger for a channel or not.

For the example of last section, channel A has been allocated ¼ of the link bandwidth (ITV = 0, and TS = 4). Therefore the channel will be triggered by counter values of 0, 4, 8, 12,... from the counter range, 0..16. If the ITV is 0 and the TS is 1, then output from the channel will be triggered by counter values of 0, 1, 2, 3, etc. That is, it will be allocated 100% of the channel bandwidth. Table 4.2 shows the channels A and B from the example above could be therefore allocated ITVs and TSs as follows:

Table 4.2 ITV and TS value of A and B channels

channel	bandwidth requirement	ITV	TS
A	1/4	0	4
B	1/8	1	8

Many other allocations of ITV and TS are possible. How do we find an ITV for the channel being established? It is not sufficient for this ITV simply to differ from any other ITVs that have previously been allocated, because although the *initial* trigger values for two channels running at different speeds may differ, later trigger values could coincide, which would cause the two channels to be triggered simultaneously. The ITV allocation

procedure must prevent this from occurring. The requirement can be expressed by the following inequality, where ITV_n is the ITV to be allocated to a channel (n):

$$\text{i.e. } ITV_n + pTS_n + ITV_m + qTS_m \quad \forall\ m, p, q$$

where $p = 0 - (\text{max counter value} / TS_n) + 1$

$q = 0 - (\text{max counter value} / TS_m) + 1$

$m = 0$ - no of established channels

Finding ITV values that satisfy this inequality is not completely trivial, given that the Trigger Separations for different channels are not equal, and that channels are being established and disestablished dynamically.

However, upon examination, it transpires that a simple relationship exists between the set of existing ITV/TS pairs, and the ITV allocated to a new channel with a particular TS. A channel with a TS of 1 requires 100% of the link bandwidth and has an ITV of 0. If such a channel exists, then no other channels can be allocated bandwidth. However, if such a channel does not exist, it would be possible to allocate two TS = 2 channels with ITVs of 0 and 1, or four TS = 4 channels with ITVs of 0, 2, 1, and 3. According to paper [Lyons, 1997], if we use a binary tree to represent the ITV space, the solution to this problem is easy to find. As Figure 4.19 shows, each level in the tree corresponds to a particular TS and each node corresponds to an ITV value. A channel which requires a new ITV can be established if, at the level in the tree with the desired TS, there is an unallocated node with no allocated children or ancestors.

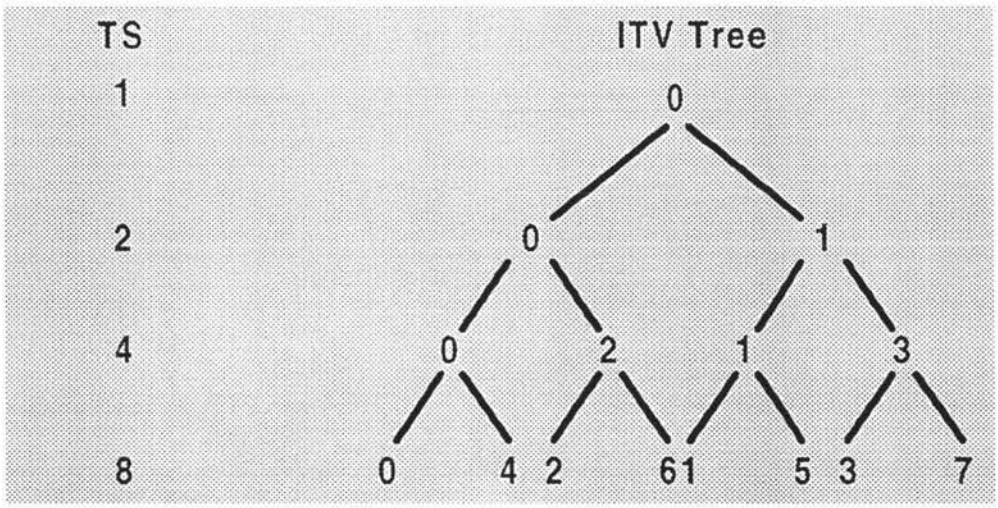


Figure 4.19: ITV space represented as a binary tree

The pattern in Figure 4.19 is fairly easy to understand. A channel associated with the children of a particular node will have a TS which is twice that of the parent. Thus only half of the trigger values which would be used by a channel associated with the parent node will be used by either of the children. Hence the root node, which has an Initial Trigger Value of 0, and a Trigger Separation of 1, to a channel which uses all counter values, thus using 100% of the link bandwidth. If such a channel has not been established, two channels, each using half the link bandwidth could be established. They would both have a Trigger Separation of 2. Thus one could have an Initial Trigger Value of 0, and the other could have an Initial Trigger Value of 1, without a trigger concidence conflict. In general:

$$
\begin{aligned}
ITV_{root} &= 0 \\
ITV_{leftchildnode} &= ITV_{node} \\
ITV_{rightchildnode} &= ITV_{node} + TS_{node}
\end{aligned}
$$

It is not permissible for a channel to be associated with a node which has ancestors or descendants which are associated with a channel, as the intersection between the complete set of trigger values generated by a node's (ITV, TS) combination and the complete set of trigger values generated by the (ITV, TS) combination of any of its offspring is non-empty.

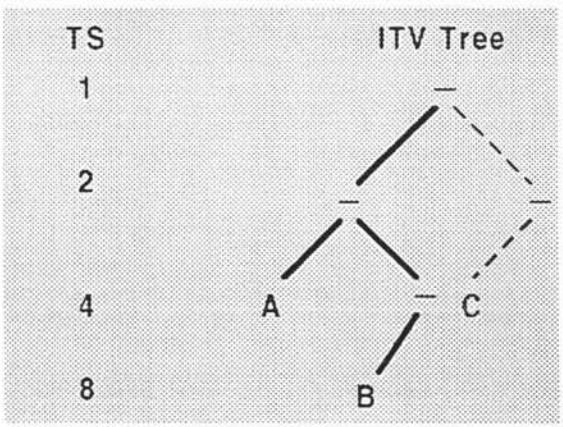


Figure 4.20: Finding an ITV for C after A and B have been established

At any moment during the operation of the system, only part of the binary tree will actually exist. After the establishment of channels A and B from the example above, only the part of tree shown using solid lines in Figure 4.20, below, would exist, and the nodes allocated to channels A and B would contain the names of those channels. Further allocations would involve traversing (and adding to) the tree to find an unallocated node

at the desired level with no allocated ancestors or children. Thus a node C which requires $^1/_4$ of the output bandwidth (a TS of 4) would be allocated by adding the dotted-line part of the tree.

It has previously been explained that a channel, n say, will be triggered by any of the counter values ITV_n, $ITV_n + TS_n$, $ITV_n + 2TS_n$, and so on. There is little time during the output phase of the switch's clock cycle to determine whether the current counter value matches any of these trigger values, so it would be infeasible to store them all and search the stored values sequentially for one that matched the current counter output. Searching them all in parallel, using constant-addressable memory would be technically feasible, but expensive, because of the size of the resulting CAM.

However, the all the trigger values for a particular channel share a bit-pattern in the low-order bits, and this bit-pattern occurs in the trigger values for no other channel. Therefore, if the other, high-order, bits of the counter value are *mask*ed, a comparison between the current counter value and the ITV will suffice to detect any trigger value for the channel.

We have previously seen that the TSs is powers of 2. Thus a channel with a TS of 64 and an ITV of 12_{10}, (001100_2) will be triggered when the counter is

$$
\begin{aligned}
12_{10} &= (0 \times 2^4 + 12)_{10} = 00\ 1100_2 \\
28_{10} &= (1 \times 2^4 + 12)_{10} = 01\ 1100_2 \\
44_{10} &= (2 \times 2^4 + 12)_{10} = 10\ 1100_2 \\
60_{10} &= (3 \times 2^4 + 12)_{10} = 11\ 1100_2
\end{aligned}
$$

In other words, the bottom four bits of *all* the trigger values for the channel are identical to the bottom four bits of the ITV, and furthermore, there are no numbers with the 1100 bit combination in the bottom four bits that are not valid trigger values for the channel. In general, the trigger values for an arbitrary channel with a TS of 2^n will differ in only their bottom n bits. Thus, if we give the *mask* value as 11 0000, the result of *mask* value OR (ITV value EQ[1] counter value) should be equal to 11 1111, such as:

mask		counter		ITV		result
11 0000	OR	(00 1100	EQ	00 1100)	=	11 1111
11 0000	OR	(01 1100	EQ	00 1100)	=	11 1111
11 0000	OR	(10 1100	EQ	00 1100)	=	11 1111
11 0000	OR	(11 1100	EQ	00 1100)	=	11 1111

4.9.3 Implementation of the Trigger Mechanism

In the above sections, the principle of the trigger mechanism and an algorithm for working out the ITV and *mask* values are introduced. As we described before, the trigger operation is implemented by using the SPAMM module of the map memory. The controller maintains an ITV tree for each output port. So an ITV tree represents the full bandwidth of an output port and the numbers of ITV trees is same as the output port numbers. If the switch receives a Setup cell for building a CBR/VBR channel, the controller checks the ITV tree of the indicated output port. If the required bandwidth is available in the ITV tree, the controller establishes this new channel and updates the ITV tree.

Because each output port has a unique ITV tree, the same ITV and *mask* values might be assigned to the different CBR/VBR channels which are not routed to the same output port.

This might seem likely to cause conflict at the output ports, if two channels were triggered by the same counter value. Such "conflicts" would not, in fact, be a problem, because output ports supply cells to separate communication lines, and can therefore operate completely independently. However, output ports are supplied from a single source, cell memory. In order to give equal service to all the output ports, they are served by a round robin server (i.e., time-division-multiplexing). A second counter controls this. Thus for a particular channel to gain access to its output port during a particular *outgoing* cell state, the output port counter must match the channel's output port number, and the trigger mechanism must detect a match between the trigger counter and the channel's ITV.

It is apparent that, for an n-port switch the output port counter must increment at n times the rate of the trigger counter. This is why the counter cycle is defined as four times the *clk1* cycle in Section 4.6, as shown in figure 4.6.

Data Structure of ITV Tree Node

In the above sections, the ITV tree and its structure were introduced. Figure 4.21 shows the structure of individual nodes in the ITV tree.

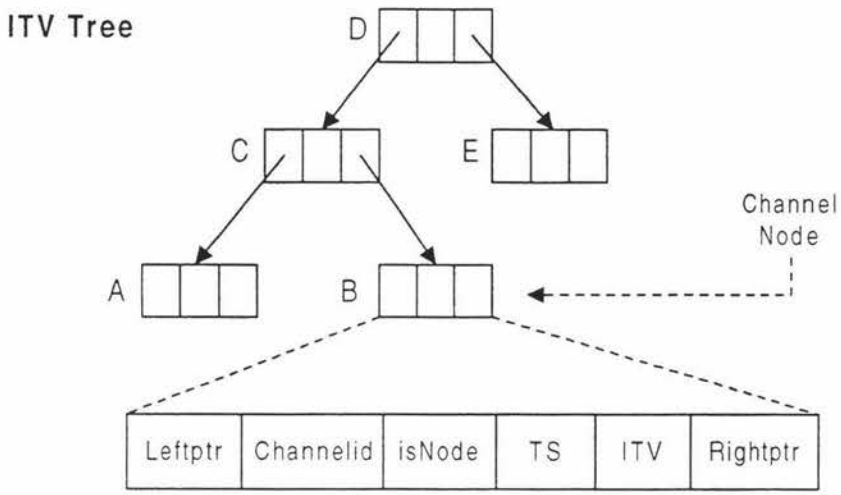


Figure 4.21: ITV Tree and ITV Tree Node Structure

Figure 4.21 shows the ITV tree node consists of five fields. They are two pointer fields and three data fields:

- **Leftptr**: is a pointer that points to the left subtree of the current node.
- **Rightptr**: is a pointer that points to the right subtree of the current node.
- **Channelid**: stores the channel id for an established CBR/VBR channel. The same channel id is also stored in the map memory.
- **isNode**: flags the node which is either leaf or node type in the ITV tree. The leaf represents an assigned channel and it won't have a child. The node doesn't represent a channel and it's just a node in the tree. If a node is treated as a leaf, the value of **isNode** is false. Otherwise, it's set to true.
- **TS**: stores the TS value of the node.
- **ITV**: stores the ITV value of the node.

Node B in Figure 4.18 can thus be represented by the following values (Note: in the simulated ATMSWITCH module, the ITV, *mask* and channelid fields are defined to be 10 bits wide.).

$$
\begin{aligned}
&NodeB.leftptr &&= NULL\\
&NodeB.rightptr &&= NULL\\
&NodeB.channelid &&= 0001000010_2\\
&NodeB.isNode &&= false\\
&NodeB.TS &&= 4\\
&NodeB.ITV &&= 0000000010_2
\end{aligned}
$$

In the above example, *leftptr* and *rightptr* are of type pointer. They have the value *NULL* because the Node B is a leaf in the tree. *Channelid* and *ITV* are of type binary and they contain the channel number 0001000010_2 and the *ITV* value 0000000010_2 respectively.

The *TS* is of type integer and *isNode* is of type Boolean. The Boolean value *FALSE* is assigned to *isNode* because the node is a leaf.

General Algorithm for the Controller

As we described above, the controller maintains one ITV tree for each output port. An ITV tree represents full bandwidth of an output port. That is, the initial state of the ITV tree is empty, and the whole bandwidth is available. When a new CBR/VBR channel is created, the controller searches the ITV tree for a location which has a Trigger Separation corresponding to the desired bandwidth and does not already contain a node that has been allocated to a channel and has no allocated ancestors or children (either of which would cause simultaneous output conflicts). If such a location exists, a node is created there, and its fields are loaded with the information about the channel. As previously explained, the ITV allocated to a CBR/VBR channel is determined by the position of its node in the ITV tree, and the depth of the node is in turn determined by the channel's bandwidth requirement.

When an old CBR/VBR channel is released, the node for this channel is removed from the tree. Figure 4.22 shows the ITV tree before, during, and after the life of a ¼ bandwidth CBR/VBR channel C. The diagram also shows that the intermediate node is created for building a new CBR/VBR channel node. After the channel is released, the intermediate node is removed from the tree as well.

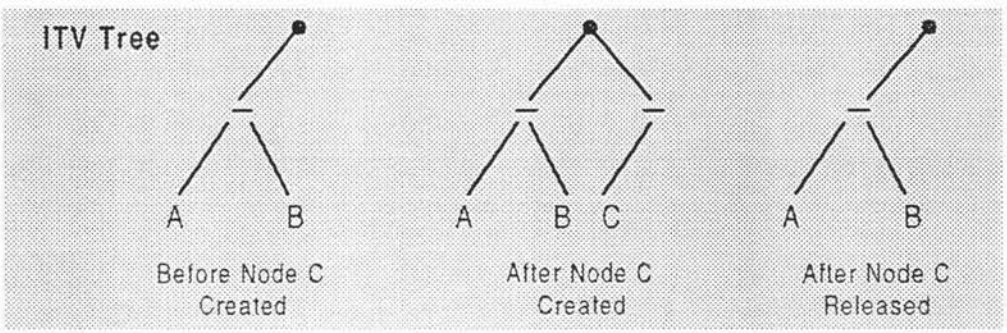


Figure 4.22: Procedure of Channel C Created and Released

When a Setup cell arrives for a new CBR/VBR channel, the controller traverses the ITV tree and finds a location for adding the new channel node. As Figure 4.23 shows a new CBR/VBR channel, D needs $^{1}/_{8}$ of the total link bandwidth, the controller travels the tree and finds a location for the channel. The controller first walks the tree in an ordered manner which is defined in the simulated module.

In the example, the controller accesses nodes in the order: A-N1-R-B-N2-N4-N3. It does not traverse to a location below nodes A and B, because these nodes have been allocated previously, so their descendants are unavailable. The node C hasn't be accessed because the requested bandwidth ($^1/_8$ of total bandwidth) is greater than can be supplied by a node at the level of C. The node N1 is the first place that is available for inserting channel D. When the controller accesses the node N3, it found that the N3 is also an available place and N3 is more suitable than node N1, because the available bandwidth in the node N3 is more appropriate the required bandwidth. The controller thus add a node as a child of node N3 for channel D. The dotted-line shows the channel D is created in the tree. To find an appropriate place, the controller traverses the tree and records the current most suitable node until travel is finished. The recorded node is the place to insert the new channel node.

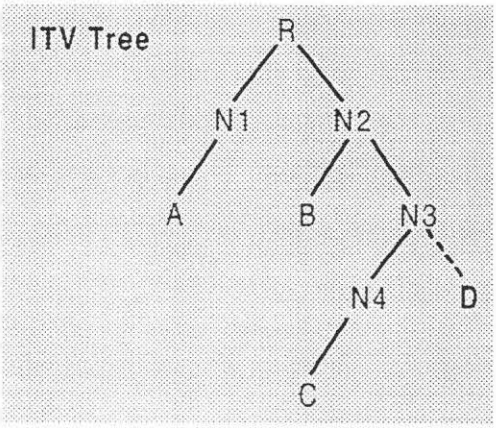


Figure 4.23: Insert Channel D in the ITV Tree

The algorithm of controller managing the tree is described below in detail.

Initialise ITV Tree

Create the four root nodes for each output port and each node is initialised as:

root.leftptr := null;
root.rightptr := null;
root.isNode := true;
root.ITV := 0000000000;
root.TS := 1;

Insert New Channel in the ITV Tree

{

Get the new channel bandwidth and channel id;
Assign root pointer to rootptr;
In_order_travel the tree from the root pointer of tree;
if (current pointer is not null and the tree level is great than request bandwidth)

```
{
        recursively in_order_travel the tree from currentNode-> leftptr;
        if (current node is not a leaf node and the node level in the tree is great
        than
                request bandwidth level)
        {
                compare current node with the recorded node
                record the node whose level is less than another node;
                return to last caller;
        }
        recursively in_order_travel the tree from currentNode-> rightptr;
        if (current node is not a leaf node and the node level in the tree is great
        than
                request bandwidth level)
        {
                compare current node with the recorded node
                record the node whose level is less than another node;
                return to caller;
        }
}
Get the most suitable node N;
if (the level of node N is just last high level than the request bandwidth level)
        create the node for this request channel;
else
        create the intermediate node recusively until the node is just high than the
        bandwidth level;
Assign the channelid, ITV, TS, leftptr and rightptr to the new channel node;
}
```

The ITV value for the node can be worked out by using following algorithm:

ITV_{root} = *0000000000;*
$ITV_{leftchild(parent)}$ = *parent ITV value;*
$ITV_{rightchild(parent)}$ = *parent ITV value OR parent TS value;*

For example, if parent ITV value is 3, it's on the level 2 and the TS value for this level is 4, so

$ITV_{leftchild} = 3 = 0000000011_2$
$ITV_{rightchild} = 3\ OR\ 4 = 0000000011_2 + 0000000100_2 = 0000000111_2$

The *mask* value for the node can be worked out by using following algorithm:

Find the TS value that equal to bandwidth;
In terms of TS value, find the level value l;
left shift "1111111111" value l times and the result is *mask* value.

For example, if bandwidth is 1/8, the TS value is 8. The related level value is 3, so

"1111111000" is *mask* value for this channel.

Release the Old Channel from the Tree

```
Get the channel id N;
Assign root pointer to rootptr;
In_order_travel the tree from the root pointer of tree;
if (current pointer is not null)
{
        if (the channelid of current node is equal to N)
                exit to in_order_travel;
        else
        {
                recursively in_order_travel the tree from currentNode-> leftptr;
                recursively in_order_travel the tree from currentNode-> rightptr;
        }
}
```

Release channel N node from the tree;
Remove the parent node of channel N if the channel N is only its child;
Remove the parent node if current removed node is only its child until the condition is not satisfied.

4.10 Cell Memory Module

Cells are buffered in cell memory in the ATMSWITCH. When a cell arrives from the switch's input port, it will be stored in cell memory until it is selected for output and then transferred to the output port. The cell buffer is constructed from memory chips, so it has distinct read and write cycles. Each operation cycle of the ATMSWITCH is thus divided into two states. One involves writing a cell into the buffer and it is called the ***acceptcell*** state. The other involves reading a cell from buffer; it is called ***outgoing*** state. The switch continually alternates between these two states. This is a characteristic of buffer-based ATM switch architectures.

In the ATMSWITCH architecture, a complex data structure is used to organize cells so that the servicing behaviour they experience suits their required Class of Service. Three modules are used to implement the data structure. They are **Map Memory**, **Cell Memory** and **CQlinks** (Cell Queue links memory) modules. Earlier, map memory, which provides the links necessary for implementing the chandelier's circular list, was described in detail. In this section we deal only with the cell memory module. The CQlinks module which supports the cell queue structure of chandelier will be introduced in next section.

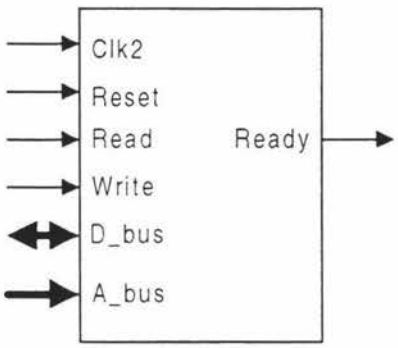


Figure 4.24 : Cell Memory Port Diagram

The external ports of the simulated cell memory module are shown in Figure 4.24. The cell memory has several ports, like normal memory; their functions are described below:

- **Reset** Port: connected to the Clock module. When the *reset* signal is changed from low to high, the cell memory is reset and all data stored in it is cleared.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the cell memory operations.
- **Write** Port: connected to the controller. During the *acceptcell* state, if the *write* signal is set to high by controller, the cell data is written into the cell memory.
- **Read** Port: connected to the controller. During the *outgoing* cell state, if the *read* signal is set to high, the cell memory reads cell data from the memory.
- **A_bus** Port: connected to the address bus in the switch. During the *acceptcell* state, the address bus provides an available cell memory address for the incoming cell to store. During the *outgoing* cell state, the address bus is provided with a cell memory address from which the cell will be read.
- **D_bus** Port: connected to the cell data bus in the switch. During the *acceptcell* state, the input port outputs its cell onto the *D_bus* from which it can be written into the cell memory through the *D_bus* port. During the *outgoing* cell state, data is transferred from cell memory's *D_bus* port via the *D_bus* to the output port.
- **Ready Port**: outputs the *ready* signal when the cell memory has finished writing or reading operations. The *ready* signal notifies the other related modules that the cell memory has completed its job.

4.11 Cell Queue Link Module

In last section, the *CQlinks* module, which is part of the cell queue structure of chandelier, was mentioned. As described in previous sections, the cells of each channel are organized into a cell queue which is maintained separately from the queues of cells associated with other channels. Figure 4.22 shows the channel header which is organized as a circular list implemented in map memory, and queues of cells, pointed to by the cell headers. Because the circular list of headers has been described before, here I just describe how the cell queue structure is realized by using the CQlinks module.

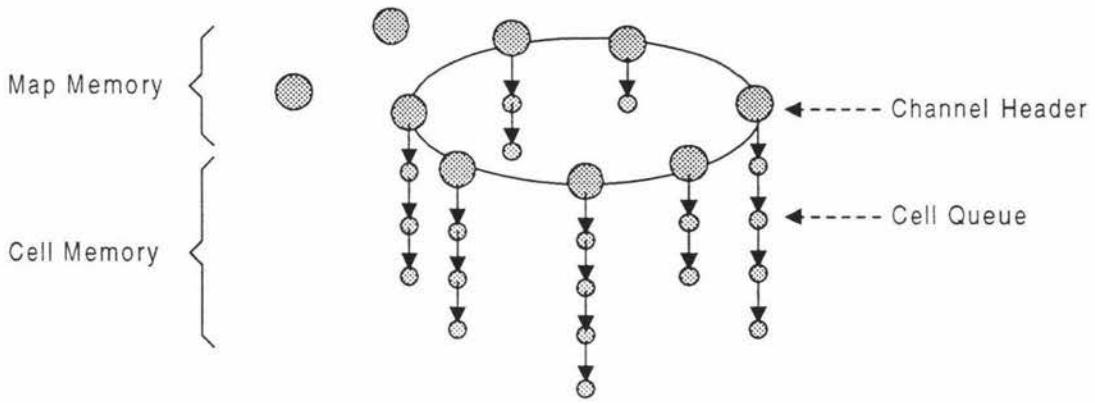


Figure 4.25: Channel Header Circular List and Cell Queue Structure

Figure 4.25 shows that the cell queue is implemented as a linked list; the first item of the queue is pointed to by the channel header. In previous section, cell memory was described as storing cell data only and no special structure was defined for linking the cells together into a queue. The purpose of the CQlinks is to hold the queue pointers. CQlinks is actually a separate high-speed memory, running in parallel to cell memory. If cell memory has a pair of adjacent cells in a cell queue at addresses m and n, then location m in CQlinks will contain the value m. The reason that the link addresses are not stored in cell memory is that in order to insert an item into the list, it is necessary to write that entry, and also the pointer field of another entry. Doing this using two separate accesses to call memory would reduce the speed of insertion operations by 50%. Deletion operations would also involve a similar delay, as data would have to be read from the cell which was being output, and then its pointer field would have to be updated to insert it into the list of free cells.

The following description concerns the details of the CQlinks module.

When it is first initialized, CQlinks is a list of free cells. This is shown in Figure 4.26; each location$_i$ of CQlinks stores the next address i + 1. So location$_0$ of CQlinks contains 1, location$_1$ contains 2, and so on. CQlinks is the same size as cell memory. If a cell is stored at the address$_n$ in cell memory, then the link pointer for this cell is stored at the address$_n$ in CQlinks. The address of the head of the list of free cells is stored in a register called *Freecell*. When the switch is about to write a cell into the cell memory, the address where the write operation will occur is given by the *Freecell* register. *Freecell* is then updated to point to the next cell in the list of free cells. For example, the value of current *Freecell* register is 0 in the Figure 4.26. At this time, if the switch accepts an incoming cell, the cell will be stored at the address 0 of cell memory because the cell memory address is given by *Freecell*. After the cell memory has written the cell, the value of *Freecell* is updated to point at the address contained in the pointer field of location 0 of CQlinks, that is, 1.

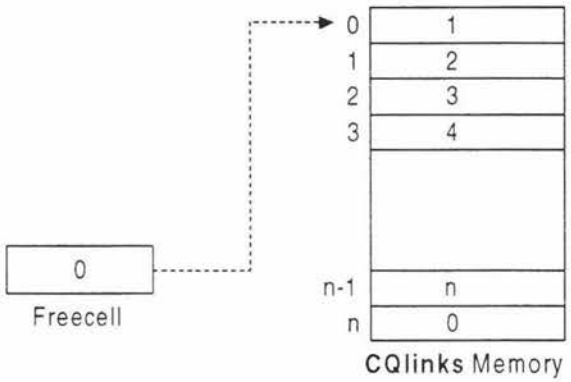


Figure 4.26: Initialized State of Cqlinks Memory

After the cell has been written into cell memory, the cell queue head and cell queue tail pointer (and the header, if the cell is the only one in the queue) fields of the channel header belonging to that queue are updated.

Let's consider an example to show how the CQlinks manages the cell queue and handles the available address for cell memory. Suppose three channels, A, B, and C have been established through the switch. Channel A has 3 cells, A_1, A_2, and A_3 to be stored in the cell memory. Channel B has 4 cells, B_1, B_2, B_3, and B_4. In this example, only the process of writing cells into memory is considered and CQlinks is assumed to start in the initialised state. We thus get the following results after the switch writes a cell at each *acceptcell* state.

- Write Cell State 0: the *Freecell* value is 0, cell memory is empty and CQlinks is in the initialised state;
- Write Cell State 1: cell A_1 is stored at address 0 of cell memory. The *Freecell* value is updated to 1 and address 0 of CQlinks is set to 0. Both cell queue head and cell queue tail pointers in the channel header for channel A are are pointed at the address of the new cell in cell memory, that is 0;
- Write Cell State 2: the cell B_1 is stored at address 1 of cell memory. The *Freecell* value is updated to 2 and address 1 of CQlinks is set to 0. Both cell queue head and cell queue tail pointers in the channel header for channel B are pointed at the address of the new cell in cell memory, that is 1;
- Write Cell State 3: the cell A_2 is stored at address 2 of cell memory. The *Freecell* value is updated to 3 and address 2 of CQlinks is set to 0. Channel A already has a cell A_1 stored at the address 0 of cell memory, so the contents of address 0 of CQlinks is updated to 2 which is the address of cell A_2 in the cell memory. So address 0 in cell memory contains the data for cell A_1, and address 0 in CQlinks contains 2, a pointer to the address of the next location containing a cell for circuit a cell memory. At this time the cell queue tail pointer of channel A is updated to 2, because it always points to the address of the last cell in the queue.

- Write Cell State 4: Same as Write Cell State 3, the cell *B2* is stored at address 3 of cell memory. The *Freecell* value is updated as to and address 3 of CQlinks is set to 0. The address 1 of CQlinks is updated to 3 and the cell queue tail of channel B is updated to 3.

After several write states, the cells that were coming through channel A and B have been stored in the cell memory and the cell link pointer is stored in the CQlinks memory. Figure 4.27 shows the result of the CQlinks and cell memory value after Write Cell State 7.

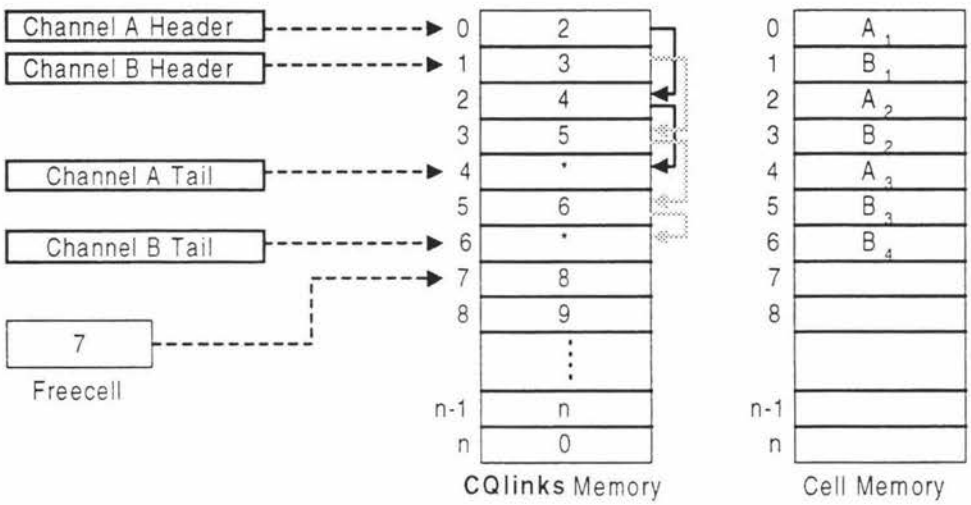
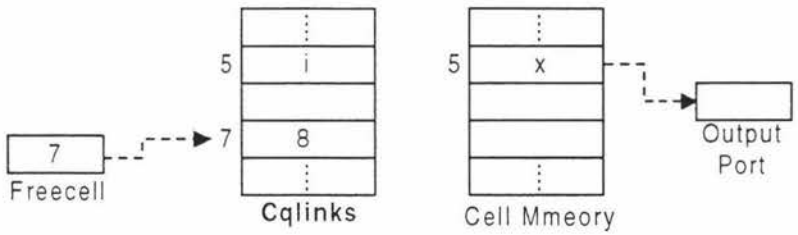


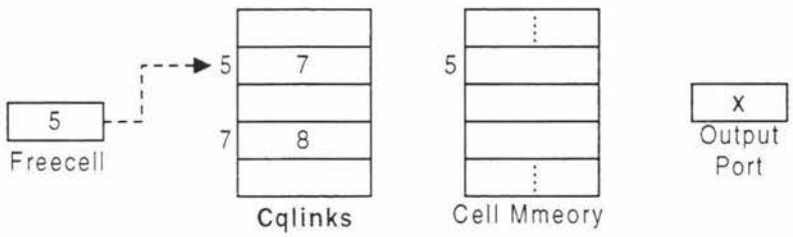
Figure 4.27: CQlinks and Cell Memory Value After Write Cell State 7

The above example only considered the writing of cells into cell memory. When switch is in the *outgoing* cell state, different operations are performed on CQlinks. At each *outgoing* state, the cell memory address from which the *outgoing* cell is to be taken will be given by the cell queue head pointer field of the channel header in map memory. After the first cell is removed from the cell queue, the cell queue head is updated by the value of the first cell link pointer which is stored in the CQlinks.

When a cell has been removed from the cell memory, this cell must be released and CQlinks also needs to be updated. The strategy used here for updating CQlinks is quite simple. The successor pointer field for the cell which has just been output are stored at the same address in CQlinks as the cell data which has just been output, so it is straightforward to insert the freed cell into the free list by copying the value of *Freecell* into the cell' CQlinks, and copying the value of the cell's address into *Freecell*. The removed cell address is therefore reused by the next incoming cell. As Figure 4.28 shows, after the cell *x* has been output from cell memory address 5, the CQlinks address 5 is updated by the value 7 which is the *Freecell* value. And the *Freecell* content will be replaced by the cell *x* address 5. The next incoming cell will thus be stored at cell memory address 5.



Contents of CQlinks and cell memory Before Cell X be sent to Output Port



Contents of CQlinks and cell memory After Cell X be sent to Output Port

Figure 4.28: Updated CQlinks after cell x be transferred from Cell Memory onto Output Port

Let us consider another example to show how these operations are done during the *outgoing* cell state. Let's start at the end of write cell state 7 in the last example. We assume that three cells for channel C, C_1, C_2, C_3 will arrive at the switch during write cell state 7; the channel B still has cells that need to put into the cell memory and they are B_5, B_6, B_7, B_8. At each *outgoing* state, a cell will be sent to output port from the cell memory. We also assume that the channel A, B and C are of equal priority and their cells will be transferred from cell memory onto output port in order. The example follows the progress of the system starting at write state 8.

- Write cell state 8: the cell C_1 is stored at address 7 of cell memory. The *Freecell* value is updated to 8 and address 7 of CQlinks is set to 0. Both cell queue head and cell queue tail of channel C is assigned the address of cell C_1 address in cell memory, that is 7;
- Read cell State 1: It's time for channel A to output a cell. The address of the cell is given by cell queue head pointer of channel A, it's 0. The content of address 0 in cell memory, A_1, is thus sent to the cell data bus. The cell queue head is assigned the value in CQlinks address 0, it's 2. The content of CQlinks address 0 is updated with the current *Freecell* value 8 and the value in *Freecell* is replaced by the address of cell A_1, 0;
- Write Cell State 9: cell B_5 is stored at address 0 of cell memory. The *Freecell* value is updated to 8 and location 0 of CQlinks is set to 0. The last cell address for channel B is changed from 6 to 0 and the cell queue tail of channel B is set to 0.

- Read cell State 2: It's time for channel B to output a cell. The address of the cell is given by the cell queue head of channel B, it's 1. The content of address 1 in cell memory, B_1 is thus sent to the cell data bus. The cell queue head is assigned the value at CQlinks address 1, it's 3. The content of CQlinks address 1 is updated with the current *Freecell* value 8 and the value in *Freecell* is replaced by the address of cell B_1, 1;
- Write Cell State 10: cell C_2 is stored at address 1 of cell memory. The *Freecell* value is updated as 8 and location 1 of CQlinks is set to 0. The last cell address for channel C of CQlinks is changed from 7 to 1 and the cell queue tail of channel C is set to 1.
- Read cell State 3: It's time for channel C to outputs a cell. The address of the cell is given by the cell queue head pointer of channel C, it's 7. The content of address 7 in cell memory C_1 is thus sent to cell data bus. The cell queue head is assigned the content of CQlinks address 7, it's 1. The content of CQlinks address 7 is updated with the current *Freecell* value 8 and the *Freecell* is replaced by the address of cell C_1 7;

Figure 4.29 shows the values in CQlinks and cell memory locations 0 to 8 after Read Cell State 3. The diagram also show the headers and tails location of channel A, B and C.

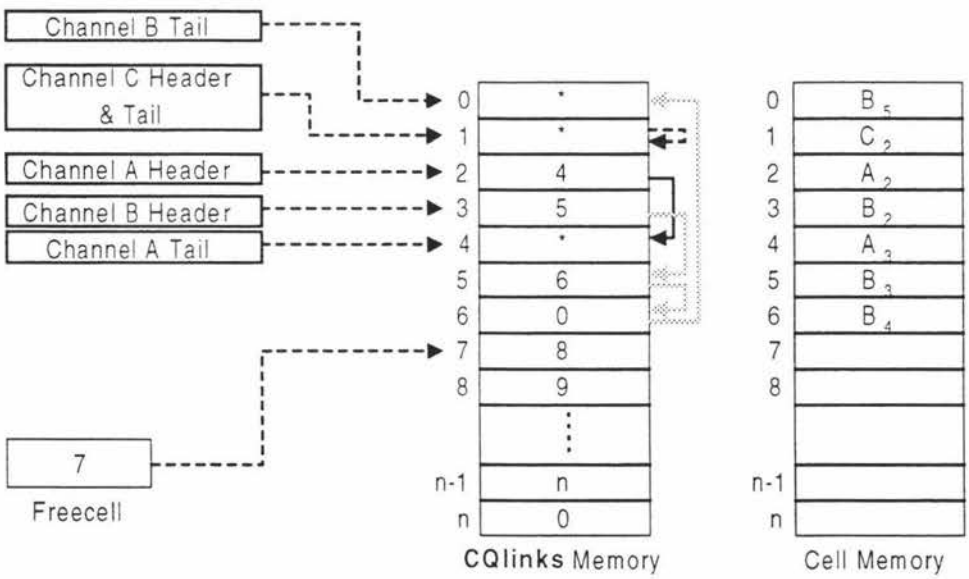


Figure 4.29: CQlinks and Cell Memory Value After Read Cell State 3

The cell queues of channel A, B and C after the Read Cell State 3 are shown in Figure 4.30.

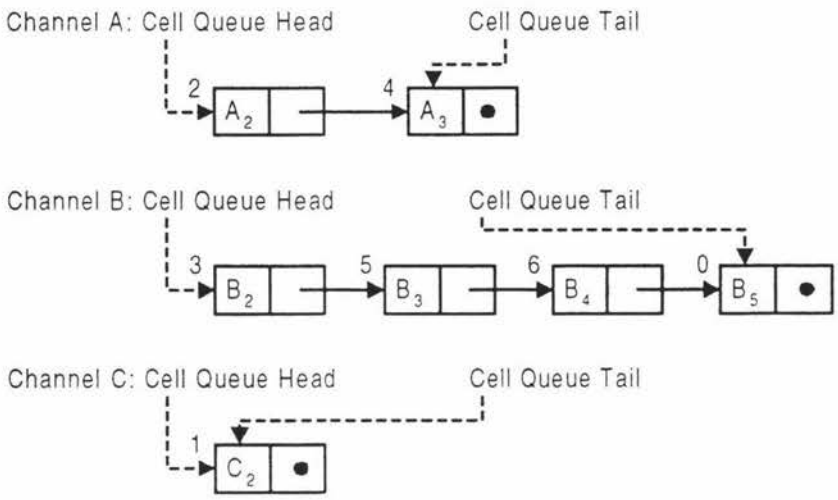


Figure 4.30: Cell Queues of Channel A, B and C after Read Cell State 3

The above description illustrates how CQlinks is used to manage and update cell memory addresses, to support the operation of the cell queues used in the chandelier and trigger mechanisms of the ATMSWITCH.

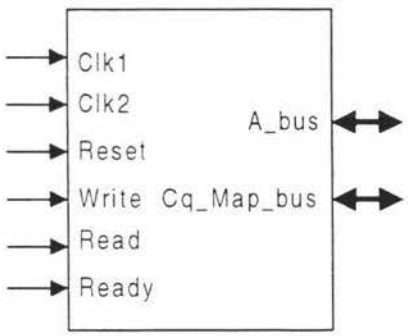


Figure 4.31: **CQlinks** Port diagram

The external ports of the simulated CQlinks module are shown in Figure 4.31. CQlinks has ports like those in a normal memory that are used to input and output signals for CQlinks module.

- **Reset Port**: is driven by the Clock module. When the signal on the **Reset** port input changes from low to high, CQlinks is reset to initialised state.
- **Clk1 Port**: is driven by the Clock module. When the *clk1* signal is changed from low to high, CQlinks outputs an available address for writing an incoming cell into cell memory.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the CQlinks operations.
- **Ready Port**: is driven by cell memory. During the *Acceptcell* state, the high phase of the ready signal notifies CQlinks that the cell has been written into cell memory.

During the *Outgoing* cell state, a high value on the ready signal notifies the CQlinks that the cell has been read from the cell memory.

- **A_bus Port**: outputs an available cell memory address for writing coming cell into cell memory at *acceptcell* state.

4.12 Output Round Robin Server

Like the Input Round Robin Server (simply called the input server), the Output Round Robin Server is used to indicate the currently available output port and cooperate with the map memory module to determine which output port is going to output the cell. It uses a circular list to manage each output port. As Figure 4.32 shows, the nodes on the circular list represent the output ports. At each *outgoing* state, the output server start to check whether the current output port is free or not.

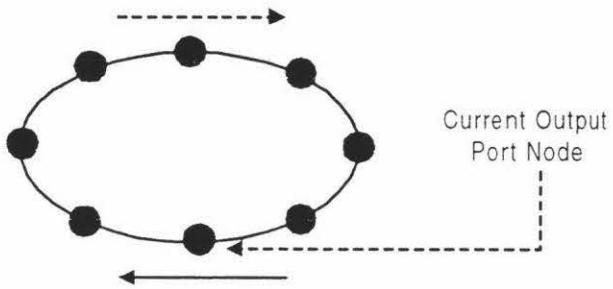


Figure 4.32: Structure of Output Round Robin Server

After a cell is sent out from the output port, the flag signal *free* is set to high. If a cell has been transferred from the cell memory to the output port, the *free* signal is set to low. At each *outgoing* state, the output server checks the current output number with the free signal of this port. If the free signal is high, the output server outputs the current port number to map memory. The current output number is used by the map memory to examine whether there are any CBR/VBR channels to be triggered for this port.

During the map memory is doing trigger operation, the output server receives the chandelier information signal which is from the *chandelier register* of map memory, and checks the chandelier information associated with the current output port. If the chandelier buffer of the current output port is not empty and the free signal of this port is high, the output server outputs the high phase *outgoing* signal to the controller. The high phase *outgoing* signal means the chandelier buffer of the current output port is not empty and the current output port is free.

The external ports of the simulated output round robin server is shown in Figure 4.33. Their functions can be described below:

- **Free Port**: there are four free ports are used in output server module, they are from free00 to free03. Each free port is connected with the *free* signal of an output port. If the output port is empty, the free signal is set to high. The output server will use those free ports to check whether the output port is available to receive the cell.
- **Clk1 Port**: the **clk1** port receives *clk1* signal from the Clock module. When the *clk1* signal changes from high to low, the output server starts its operation. When the *clk1* signal changes from low to high, the output server stop its operation and waits for the next low phase signal.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the output server operations.
- **Reset Port**: receives *reset* signal from the Clock module. When the *reset* signal is high, the output server will back to initialised state.
- **En_outport_no Port**: outputs the port number of the current output port. The output server also uses this port to enable the current output port to receive the cell. For example, the *en_outport_no* "0000" enables the output port 0# to receive the cell; the "0001" enables output port 1# to receive the cell.
- ***Outgoing* Port**: notifies the controller that the buffer and the current output port are ready to transfer the cell.
- **Chandeliersize Port**: is connected with the *chandeliersize* signal of map memory. During the *outgoing* state, the output server receives the chandelier register information from this port. The output server uses this value and *free* signal of the current output port to check whether it's available for transferring a cell from the chandelier to output port.

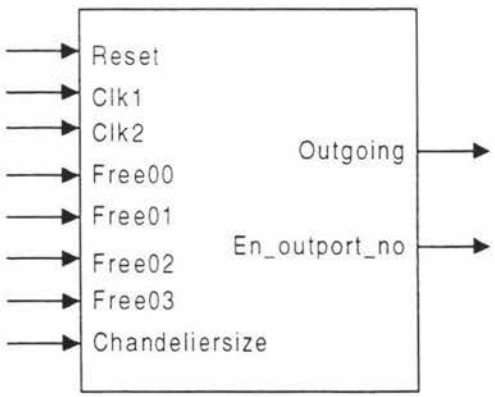


Figure 4.33: Output Round Robin Server Port Diagram

4.13 Output Port

The output port gets the cell from cell memory and then sends it out of switch. When the cell has been emptied out of the output buffer, the signal *free* is automatically set to true (high). After a cell is loaded into the output port from cell memory, the signal *free* will

be set to false (low). In the simulated ATMSWITCH module, four output ports are defined to receive cells from cell memory. They are Output Port #0, Output Port #1, Output Port #2, Output Port #3. All Output ports have the same circuit. We thus use the name of the output port to represent all the output ports.

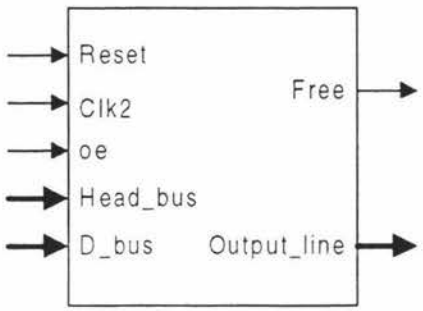


Figure 4.34: Output Port Module

The external ports of the simulated output port are shown in Figure 4.34. Their functions are described below:

- **Reset Port**: receives the *reset* signal from the Clock module. When the *reset* signal is high, the output port will clear the current cell in the output port and reset the *free* signal.
- **Clk2 Port**: receives *clk2* signal from the Clock module. The *clk2* signal gives the timing control for the output port operations.
- **OE Port**: is driven by the *en_outport_no* signal of output round robin server. During the *outgoing* cell state, if a cell is going to be sent to this output port, the **OE** port will receive an identical number which corresponds to its port number from the output server. For example, signal value "0001" corresponds to output port #1. The identified output port is thus enabled by the *en_outport_no* signal. If the output port is enabled, it gets the cell from the cell bus.
- **D_Bus Port & Head_bus Port**: The **d_bus** port receives cell data from cell memory and the *head_bus* port receives a cell header for re-assembly from map memory.
- **Ready Port**: notifies the output port that the cell has been sent onto the cell bus so the output port can download the cell from the cell data bus and cell head bus.
- **Free Port**: outputs the *free* signal. When a cell is loaded into the output port, the freeport signal is set to low. After the cell has been sent out of output port, the *free* signal is automatically reset to high.

4.14 Control Algorithm

In the above sections, the structure of the ATMSWITCH and the simulated circuit modules were introduced. While the switch is working, the operation of each module is controlled by controller. In order to give a complete impression of the switch's working

procedure, we have used a pseudo-code to describe the control algorithm of the four major components of ATMSWITCH.

According to the paper [Lyons, 1996], the algorithm is described as four parts. The first part is **Initialize** state, which initializes the **Map Memory** and **Cqlinks** when the switch is turned on or reset. The second part is **Update routing information** state, which establishes the new channel if a new channel connection is required. The third part is **Accept incoming cell** state, which maps the incoming cell onto the desired cell queue. The forth part is **Send cell out port,** which selects a channel from either trigger mechanism or chandelier to send its cell to the output port. The pseudo-code for control algorithm is described below in details.

Pseudo-code for Control algorithm

Initialise memory

```
free map entry := 0;          {variable show current map memory entry. 10-bit width}
free map entries := 1024;     {variable show available map memory quantity, 10-bit
width}
free cells := 1024;           {variable show available cell memory quantity, 10-bit width}
free cell := 0;               {variable show current cell memory address, 10-bit width}

for index := 0 to 1023 do
        {start with map memory}
        map mem[index].map entry free := T;
        map mem[index].next map entry :=index + 1;
        map mem[index].empty cell queue := T;
        {then set links in the free queue for the  cell queue memory}
        cq links[index] := index +1;

{now set up the I/O port}
for index := 0 to 3 do
        chandelier size[index] := 0;
```

Update routing information (Establish the new channel)

```
if  newcct
        if free map entries > 0 then
                new map entry :=free map entry;
                free map entry :=map mem[free map entry].next map entry;
                --free map entries;
                map mem[new map entry].map entry free       := F
                map mem[new map entry].incoming channel id := incoming channel id;
                map mem[new map entry].incoming port #      := incoming port #;
                map mem[new map entry].outgoing channel id := outgoing channel id;
```

```
                map mem[new map entry].outgoing port #      := outgoing port #;
                map mem[new map entry].empty cell queue     := T;
                map mem[new map entry].cct priority         := cct priority;
                map mem[new map entry].channel type         := channel type;
                if the newcct is CBR/UBR channel then
                        map mem[new map entry].mask : = mask;
                        map mem[new map entry].ITV := ITV;

        else
                full map memory := T;

else {clear channel associated with route mapping supplied by node administrator}
        if map mem<<route mapping>>.map entry free then
                redundant channel clear := T;
        else {first return any queued cells to the free cell list}
                queue length := map mem<<route mapping>>.cell queue length;
                while queue length > 0 do
                        freed cell := map mem<<route mapping >>.cq head;
                        map mem<<route mapping>>.cq head := cq links[freed cell];
                        cq links[freed cell] := free cell;
                        free cell := freed cell;
                        -- queue length;
        map mem<<route mapping>>.map entry free := T;
        map mem<<route mapping>>.next map entry := free map entry;
        free map entry := encoded map address;
        ++ free map entries;
```

Accept incoming cell

```
cq mem[free cell] := input buffer[current I/P port];
output := map mem<<I/P buffer[current I/P port]>>.outgoing port #;

if map mem<<I/P buffer[current I/P port]>>.channel class := channel type;
        map mem<<I/P buffer[current I/P port]>>.empty cell queue := F;
        map mem<<I/P buffer[current I/P port]>>.cq head := free cell;
        map mem<<I/P buffer[current I/P port]>>.cq tail := free cell;
        map mem<<I/P buffer[current I/P port]>>.cq length := 1;
        map mem<<I/P buffer[current I/P port]>>.next map entry := encoded map address;

        next free cell := cq links [free cell];
        cq links[free cell] := free cell;
        free cell := next free cell;
        -- free cells;
else
        if map mem<<I/P buffer[current I/P port]>>.empty cell queue
                then{we need to initialise a queue for the virtual channel in cell memory}
```

```
            if chandelier size[outport] = 0 then
                map mem<<I/P buffer[current I/P port]>>.empty cell queue := F;
                map mem<<I/P buffer[current I/P port]>>.cq head := free cell;
                map mem<<I/P buffer[current I/P port]>>.cq tail := free cell;
                map mem<<I/P buffer[current I/P port]>>.cq length := 1;
                map mem<<I/P buffer[current I/P port]>>.next map entry
                    := encoded map address;
                last chandelier entry[outport] := encoded map address;

            else {link map entry into existing chandelier}
                map mem<<I/P buffer[current I/P port]>>.empty cell queue := F;
                map mem<<I/P buffer[current I/P port]>>.cq head := free cell;
                map mem<<I/P buffer[current I/P port]>>.cq tail := free cell;
                map mem<<I/P buffer[current I/P port]>>.cq length := 1;
                map mem<<I/P buffer[current I/P port]>>.next map entry
                    := map mem[last chandelier entry[output].next map entry;
                map mem[last chandelier entry [output]].next map entry := encoded map
address;
            cell quota[outport] := map mem [encoded map address].cct priority;
            next free cell := cq links [free cell];
            cq links[ free cell] := 0;
            free cell := next free cell;
            -- free cells;
        else    {there is already at least one cell queue entry to link into}
            if free cells > 0 then
                next free cell := cq links[free cell];
                cq links[map mem<<I/P buffer[current I/P port]>>.cq tail] := free cell;
                cq links[free cell] := 0;
                current cq length := map mem<<I/P buffer[current I/P port]>>.cq length;
                map mem<<I/P buffer[current I/P port]>>.cq tail := free cell;
                map mem<<I/P buffer[current I/P port]>>.cq length := current cq length+1;
                map mem<<I/P buffer[current I/P port]>>.next map entry
                    := last chandelier entry[outport];
                free cell := next free cell;
                -- free cells;

reset full [current I/P port] {do this even if the buffer was full(free cells = 0)}
wait for 12 ns timer;
```

Send cell out port

```
current outgoing  port := O/P counter value;
if  Channel counter value OR map mem<<current outgoing port>>.mask =
        map mem<<current outgoing port>>.ITV OR map mem<<current outgoing port>>.mask then
        current O/P port := map mem[encoded map address].output port;
        if map mem[encoded map address].empty cell queue := F then
```

```
                cq source := map mem[encoded map address].cq head;
                output buffer[current O/P port] := cq mem[cq source];
                map mem[encoded map address].empty cell queue := T;
                if free cells = 0 then
                        free cell := cq source;
                        free cells := 1;
                else
                        cq links[cq source] := free cell;
                        free cell := cq source;
else
        {current O/P port is from Output Port Selector }
        current chandelier entry := map mem[last chandelier entry[current O/P port]].nextmapentry;
        cq source := map mem[current chandelier entry].cq head;
        output buffer[current O/P port] := cq mem[cq source];
        start 12ns timer;

        {delink cell entry from cell queue}
        if free cells = 0 then
                free cell := cq source;
                free cells := 1;
        else
                cq links[cq source] := free cell;
                free cell := cq source;

        new cq size := map mem[current chandelier entry].cq size-1
        if new cq size := 0 then
                map mem[current chandelier entry].empty cell queue := T;
                map mem[last chandelier entry[current O/P port]].next map entry
                        := map mem[current chandelier entry].next map entry;
                -- chandelier size[current O/P port];
                cell quota[current O/P port]
                        := map mem[current chandelier entry].next map entry].cct priority;
        else {new cq size > 0; update pointers to cq instead of shrinking chandelier}
                map mem[current chandelier entry].cq head := cq links[cq source];
                map mem[current chandelier entry].cq size := new cq size;
        if cell quota[current O/P port] =1 then
                cell quota[current O/P port]
                        := map mem[map mem[last chandelier entry].next map entry].cct priority;
        else
                -- cell quota[current O/P port];
        wait for 12 ns timer;
        reset empty[current O/P port]
```

4.15 Summary

The ATMSWITCH architecture consists of several circuit modules. They are controlled by the **controller** to accept a cell from **input port** to the **cell memory** at *acceptcell* state and send a cell from the cell memory to the **output port** at *outgoing* cell state. When a cell is coming from the input port, the **map memory** maps the cell onto the cell queue. At each *outgoing* cell state, the **trigger mechanism** and **chandelier** select a channel to send its cell to the output port. By using the trigger mechanism, the ATMSWITCH is capable to solve the **jitter** problem which is described in former chapters. The complete circuit modules of ATMSWITCH has been logical designed. The real time running of ATMSWITCH has been simulated by the constructed VHDL modules.

Chapter 5

Simulating Result and Performance Analysis

In the last chapter, the ATMSWITCH architecture and its major circuit modules were described. The ATMSWITCH architecture is an integrated design which incorporates triggering hardware for CBR/VBR channels and chandelier hardware for ABR/UBR channel (Note that we use the "ABR/UBR" to represent the ABR and UBR channels because both of them is related with the chandelier hardware in ATMSWITCH, althrough the priority of these two kind of channels is different.). In the ATMSWITCH, the associative chandelier provides round robin services to ABR/UBR channels with cells buffered in the switch. It maintains a circular list of channel headers and visits them in succession, removing one cell from a channel header's buffer queue at each visit. The trigger mechanism runs in parallel with the associative chandelier, but at a higher priority, to handle CBR/VBR channels. It uses a regularly incremented counter to generate a sequence of integers which act as the raw material of a scheduling service. Each CBR/VBR channel is allocated a set of equally-separated trigger values in the counter's range, the separation being inversely proportional to the channel's bandwidth. When the counter's value equals a channel's trigger value, output from the channel is triggered, overriding output from the chandelier. When a CBR/VBR channel is not due for triggering (or has no cells buffered), cells will be output from any ABR and UBR channels with cells buffered in the associative chandelier. Thus ABR and UBR channels are serviced regularly, and there are no "dead" cycles in which buffered data cannot be transmitted.

The general idea of the trigger mechanism is to schedule cell output for a particular channel at regular intervals, to maintain a constant end-to-end delay within the network. Because each switch involved in an end-to-end connection provides a regularly scheduled forwarding service, which is sufficiently frequent to cope with the maximum expected data rate for the channel, the total end-to-end delay experienced by individual cells will be constant. Jitter, which is unacceptable in real-time applications such as video transfer and telephony, is thus eliminated.

A complete ATMSWITCH simulated module has been constructed in VHDL, based on the circuit modules described in the last chapter. The result of running the simulated ATMSWITCH will be described in this chapter. An ideal traffic pattern is considered as

the resource to send signals to the switch so the ATMSWITCH can be tested. The timing relationship among switch modules are thus be measured. The performance analysis will also be described in the rest of this chapter.

5.1 Simulating Result

An ideal traffic pattern is considered to test the designed ATMSWITCH. It is assumed that cells are arriving at the switch at a constant rate and the service rate is the same as the line speed. It is also assumed that the load of input lines is same as the load of output line. That is, there is no bursty data, and no output port overloading. Under these conditions, it is simple to measure the time required for each circuit module to route a cell within the switch. The result presents the timing for all circuit modules of ATMSWITCH. In order to describe a complete cycle for all the circuit modules, I will describe their behaviour through two states. One is the accepting cell state when a cell is transferred from the input port to the cell memory; another is the *outgoing* cell state when a cell is sent from the cell memory to the output port.

5.1.1 Accepting Cell

In the ATMSWITCH, the clock circuit module provides a synchronous timing control for all other circuit components. When the phase of clock output signal *clk1* goes high, the controller starts to execute the operations belonging to the accept cell state, and other related circuit modules also start operations involved in transferring a cell from an input port to the cell memory. The description below shows the time sequence and operation time for these module.

Input Round Robin Server

The input round robin server indicates which input port is going to send its cell to the cell memory. After it selects an input port, it outputs a signal to enable this input port to put its cell onto the cell bus. While the switch is working, the input server reads *full* signals from all input ports (Note: a high voltage on the *full* signal indicates that a cell is buffered at the input port). Figure 5.1 shows the simulated result of the input server running. In the diagram, the *Full00* signal represents the full signal from the Input Port #0, *Full01* is from Port#1, *Full02* is from Port#2, and *Full03* is from Input Port #3. At each *acceptcell* state, the input server chooses the current input ports to let its cell transfer to the cell memory in case the input port is full.

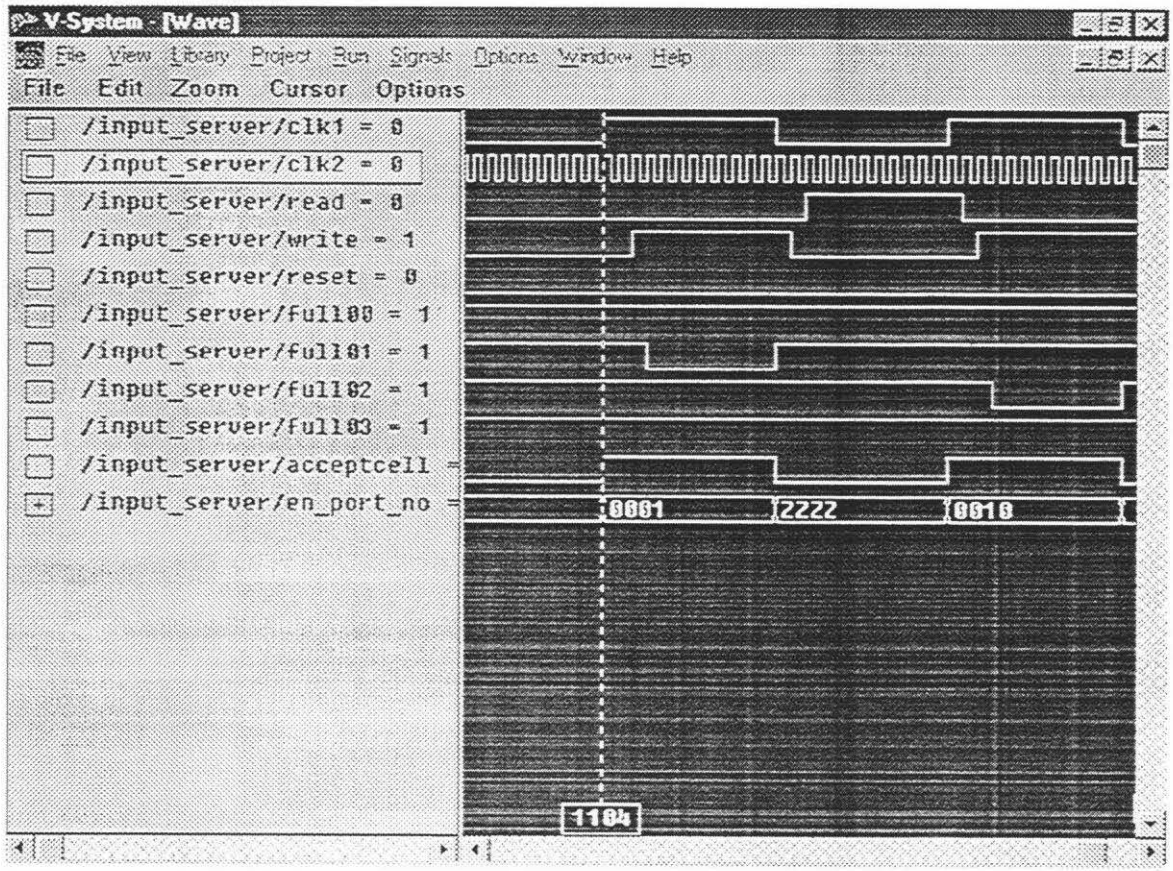


Figure 5.1: Timing of Input Server Operation

This diagram shows the timing for the selection of an input port by the input server. At the point = 1104 (indicated by the vertical line in the middle of the diagram) the leading edge of the *clk1* signal changes to high. The input server outputs a high *acceptcell* signal and *en_port_no* signal. The *acceptcell* signal notifies the controller that a cell is waiting in the input port to be sent out. The *en_port_no* signal enables the current input port to send its cell. The values of the two signals were determined by the input server before the *clk1* edge so that they can be output by the input server when the *clk1* signal goes high.

At the point = 1104, the *en_port_no* signal outputs the value "0001" to enable input port #1. The diagram also shows that *en_port_no* is "0010" after 24 *clk2* cycles, so then input port #2 is selected to supply a cell output. From the diagram, we can see that the input ports are fairly selected and repeats from #0 to #3 (Four input ports is designed in the simulated ATMSWITCH module). The diagram also shows that the *acceptcell* signal and *en_port_no* signal is reset at the low-going edge of *clk1*.

The results of the simulation show the input server operating simultaneously with other circuits. The operation which selects the input port occurs before the high-going edge of *clk1*. Once the input server receives the *clk1* event, it can output its signal immediately. So the input server doesn't consume any of the time nominally available for writing a cell to the cell buffer.

Input Port

The input port captures cells from the ATMSWITCH's input line and stores them in the cell input buffer. The input port automatically asserts the *full* signal after a cell is loaded. During each *acceptcell* state, the Input Round Robin Server selects the current input port by sending its port number to *oe* port. The current input port is thus enabled to put its cell onto the cell bus. After this has occurred, the *full* signal of the input port will be reset by the controller.

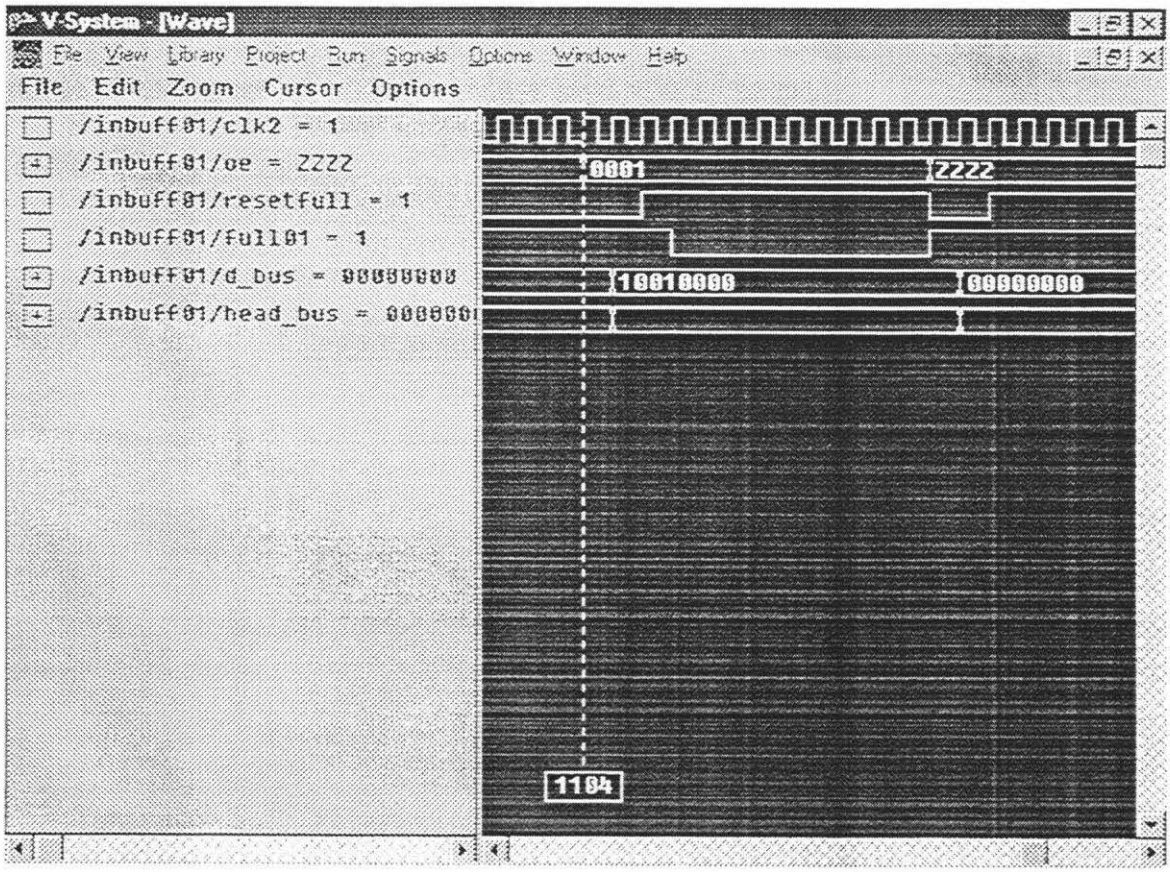


Figure 5.2 : Timing of Input Port Operation

Figure 5.2 is a simulated voltage/time trace showing the signals involved in transferring a cell from input port #1 onto the cell bus. Before point = 1104, the port's *full* signal is high, indicating that it contains a cell. At point = 1104, the *oe* port which is connected with the *en_port_no* port of the input server receives the *en_port_no* signal "0001", and input port #1 is output-enabled. After 1 *clk1* cycle from the point = 1104, input port #1 outputs the cell data onto the *d_bus* port and the cell head onto the *head_bus* port. After 2 *clk2* cycles, the controller resets input port #1 by setting the *resetfull* signal to high. After input port #1 detects the *resetfull* event, the *full01* signal is reset. When a new cell arrives from the input line at input port #1, causing the *full01* signal to be set. After 12 *clk2* cycles, the *en_port_no* signal is disabled, so the input port stops outputting its cell data onto the bus.

The results of the simulation shows that the input port #1 took 1 *clk2* cycle to output its cell onto the bus after it's enabled by the *en_port_no* signal. So the input port spent 1 *clk2* cycle for outputting its data.

Controller

The controller consists of a control circuit and a set of control signals to control and adjust the operation of each circuit module, so the switch can transfer cells from the input port to cell memory and send cells from the cell memory to the output port.

The *acceptcell* state occurs while the *clk1* signal is high. First controller waits for an *acceptcell* signal from the input server. When this goes high, the controller fetches the cell header from *head_bus*. According to the type of the incoming cell, the controller either sets the *write* signal to high or *setupcell* signal or *releasecell* signal. A high *write* signal enables the cell memory to write a cell from the cell *d_bus* into the memory. The high *setupcell* signal control the switch to establish a new channel and the *releasecell* signal lets the switch release an old channel.

Figure 5.3 shows the result of simulating the behaviour of the controller during the *acceptcell* state, after *acceptcell* and *write* have both been set to high. The *clk1* signal goes high at point = 1104ns, after that the controller waits for an *acceptcell* signal event. At the same time, a high *acceptcell* signal comes from the input server, indicating that the input port has been triggered to send the cell. The controller thus goes to next stage and after 1 *clk2* cycle from the point = 1104, it fetches the cell header data from the *head_bus* port and starts to check its type. The checking operation has taken 1 *clk2* cycle time. In the case shown in the diagram, the incoming cell is a normal data cell (neither *Setup* cell

nor *Release* cell). The controller thus sets the *write* signal to high at the beginning of third *clk2* cycle. The high phase of *write* signal enables the cell memory to write the cell data from *d_bus* into the memory. At same *clk2* cycle, the controller outputs a *resetfull* signal to the input port so the input port can reset its *full* signal.

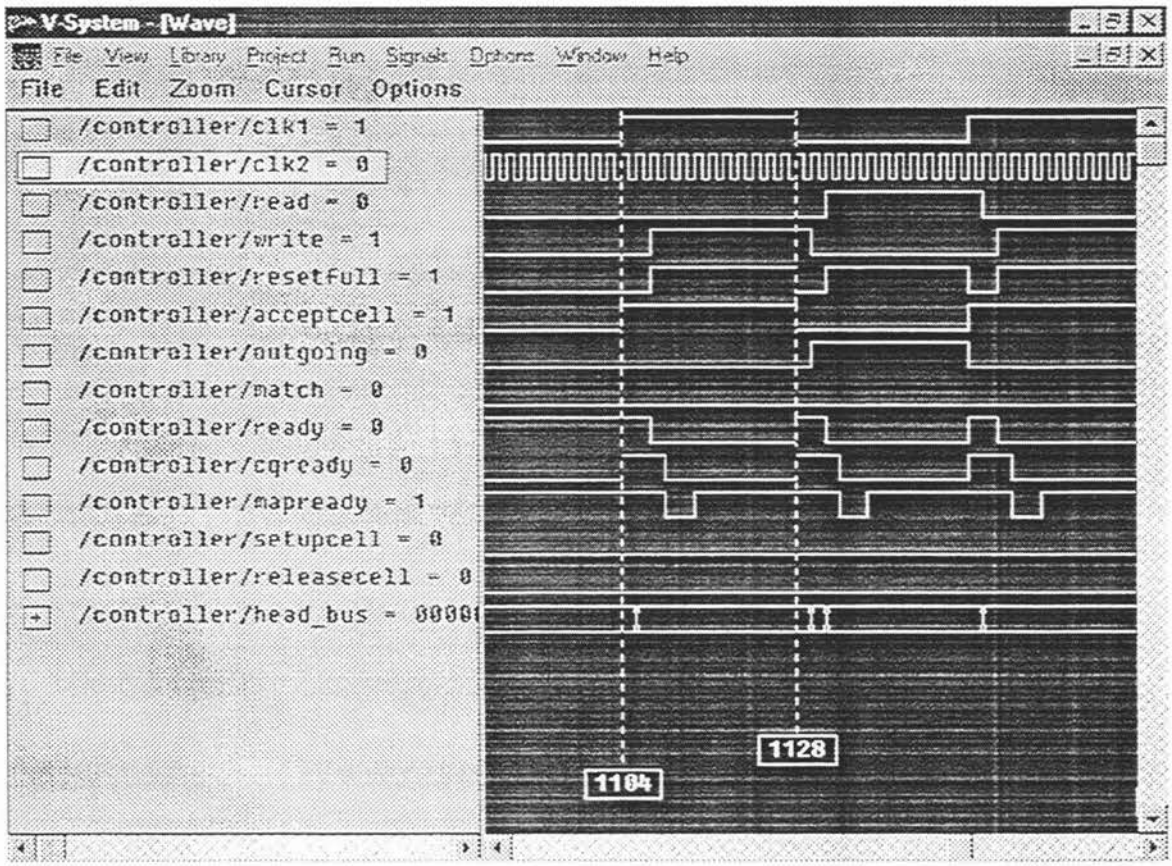


Figure 5.3: Timing of Controller Operation

The signal *setupcell* and *releasecell* is low because the incoming cell is a normal data cell. After 1 *clk2* cycle from the point = 1128, the *write* signal is reset because the *ready* signal of the cell memory is changed to high, indicating that cell memory has finished storing the cell data. The waveform in the diagram shows that the controller took 1 *clk2* cycle to check the cell head type. The elapsed time between the controller receiving the *acceptcell* signal and outputting the *write* signal, is 2 *clk2* cycle.

Map Memory

The map memory maps the incoming cell into the channel header that belongs to this cell in the map memory. According to the channel type, the cell data will be appended either onto the cell queue of a chandelier buffer (for ABR/UBR channels) or onto an independent queue (for CBR/VBR channels). During the *outgoing* cell state, map memory uses two special devices, the trigger mechanism and the chandelier, to select a channel from which a cell will be transferred to the output port.

Map memory maintains routing information for each established channel, comprising the channel number, the input and output port numbers, and other related fields. When a cell arrives from one of the input ports, its channel number and input port number will be treated as a key and compared with the routing information in all the occupied locations in map memory. The cell data is appended to the cell queue associated with the header that matches the key.

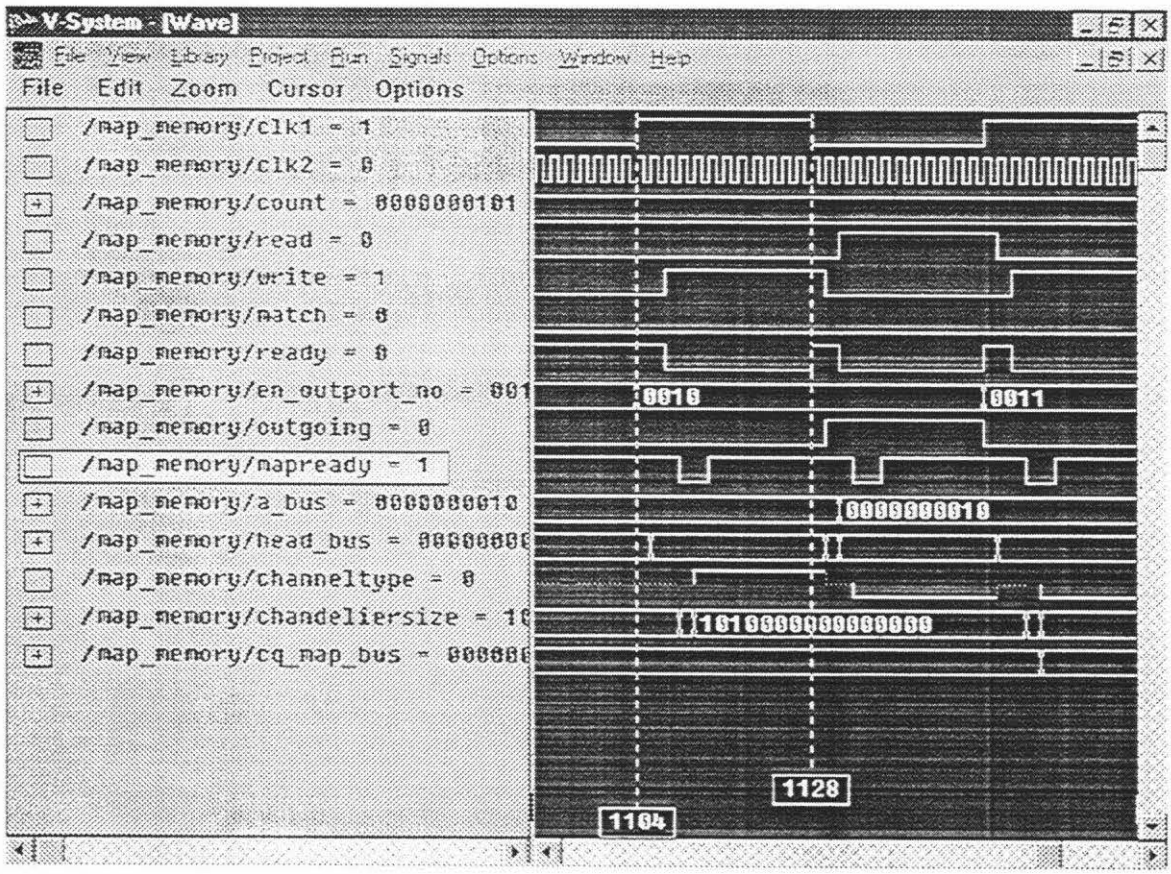


Figure 5.4: Timing of Map Memory Operation

Figure 5.4 shows the output that results from simulating the behaviour of map memory. At point = 1104, *clk1* goes high and the map memory starts to execute the operations for

which it is responsible during the *acceptcell* state. After 2 *clk2* cycle, the controller asserts the *write* signal. Map memory gets a cell header from *head_bus* and compares the incoming cell's channel id with channel id field of the channel information record. If the incoming cell matches, map memory will output the *channeltype* signal and *cell queue tail pointer* of the mapped channel to the CQlinks. The *cell queue tail pointer* always points to the last cell address of cell queue. It's used by the CQlinks to update the link pointer in the last cell in the queue for this channel to point to the newly included node. Meanwhile, the map memory replaces the *cell queue tail pointer* with the address of the location in cell memory in which the data for the incoming cell is stored. The *cell queue length* and *cell queue empty* fields of the map memory channel information record for the mapped channel also need to be updated. When that has been done, map memory outputs the refreshed *chandeliersize* value (chandelier register information) to the output server after 4 *clk2* cycles from the point = 1104. It also outputs the *channeltype* signal, *cell queue tail pointer* value. The map memory require 2 *clk2* cycle to map the incoming cell and update the mapped channel record. After that, it outputs the *mapready* signal to indicate that the map memory has completed its operations. The next *clk1* signal event occurs at point = 1128 when the *clk1* signal changes from high to low. Map memory thus stops outputting the *channeltype* signal, *cell queue tail pointer* and *chandeliersize* value at that point.

The waveform of map memory module shows that the map memory took 2 *clk2* cycle to map incoming cell and update its channel record. It works concurrently with other circuit modules during the *acceptcell* state, so it doesn't take extra switch clock cycle to perform its operations.

CQlinks

CQlinks manages the cell memory space and the queues of cells for the currently established channels. It's a special purpose memory and makes up one part of the chandelier structure. When the switch writes a cell into the cell memory, the address is given by the *Freecell* register of CQlinks. After the cell is stored at the given address, the value in the *Freecell* register will be replaced by the address of the next free location - i.e., the value in the CQlinks word currently addressed by *freecell.*

When the CQlinks module detects that the *clk1* signal has changed from low to high, it outputs an available address onto the *a_bus* to tell cell memory where to store the incoming cell. After that, CQlinks updates the *Freecell* register value and *cell queue link pointer* for this channel.

The timing for the CQlinks simulation during the *acceptcell* state is shown in Figure 5.5. After CQlinks detects a high-going edge on the *clk1* signal, at point = 1104, it puts the cell memory address "0000000000" onto *a_bus* after 1 *clk2* cycle from the point = 1104. After 2 *clk2* cycles, the controller sets the *write* signal to high and the cell memory is thus enabled to write the incoming cell into address "0000000000" of cell memory. At the point = 1112, the CQlinks module receives the *channeltype* signal and the *cell queue tail pointer* from the map memory. The CQlinks module uses the *cell queue tail pointer* to update the cell queue link pointer and assigns a new value to the *Freecell* register. At point = 1128, the CQlinks module complete the operation about updating the cell queue pointer and assigning the new value to the *Freecell*. It thus set the *cqready* signal to high. At the same point, it receives a high phase *ready* signal from cell memory. It stops outputting the cell address to the *address bus*.

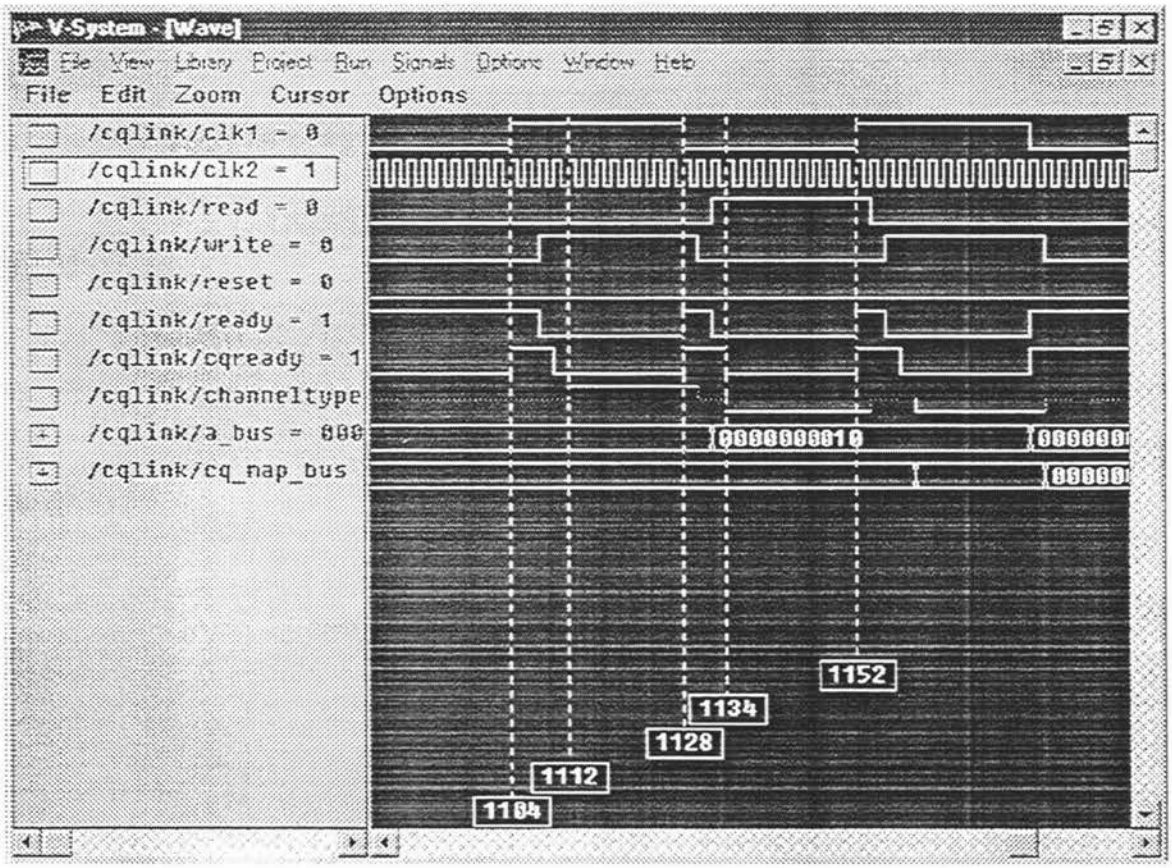


Figure 5.5: Timing of CQlinks Operation

The waveform in the CQlinks timing diagram shows that CQlinks spent 1 *clk2* cycle to output the address to the *address bus*. From the time it receives the cell tail pointer value

till the time *cqready* signal was set to high, it took 8 *clk2* cycles to have its updating operation. Due to the CQlinks works concurrently with other circuits modules, the queue maintenance operations which it performs do not add to the overall cycle time for the ATMSWITCH.

Cell Memory

The cell memory module operates as the cell buffer in the ATMSWITCH architecture. When a cell arrives from the switch's input port, it's stored in the buffer for a while and then transferred to the output port. Figure 5.6 shows the output that results from simulating the behaviour of cell memory. The diagram shows CQlinks asserting the address of to be used in the cell memory write operation after 1 clk2 cycle from the point = 1104. After another 1 *clk2* cycle time, the cell memory receives a *write* signal from the controller and the 12ns write cycle begins. It writes the data "10010000" from *d_bus* into the memory. After it finished the writing operation, the ready signal was set to high.

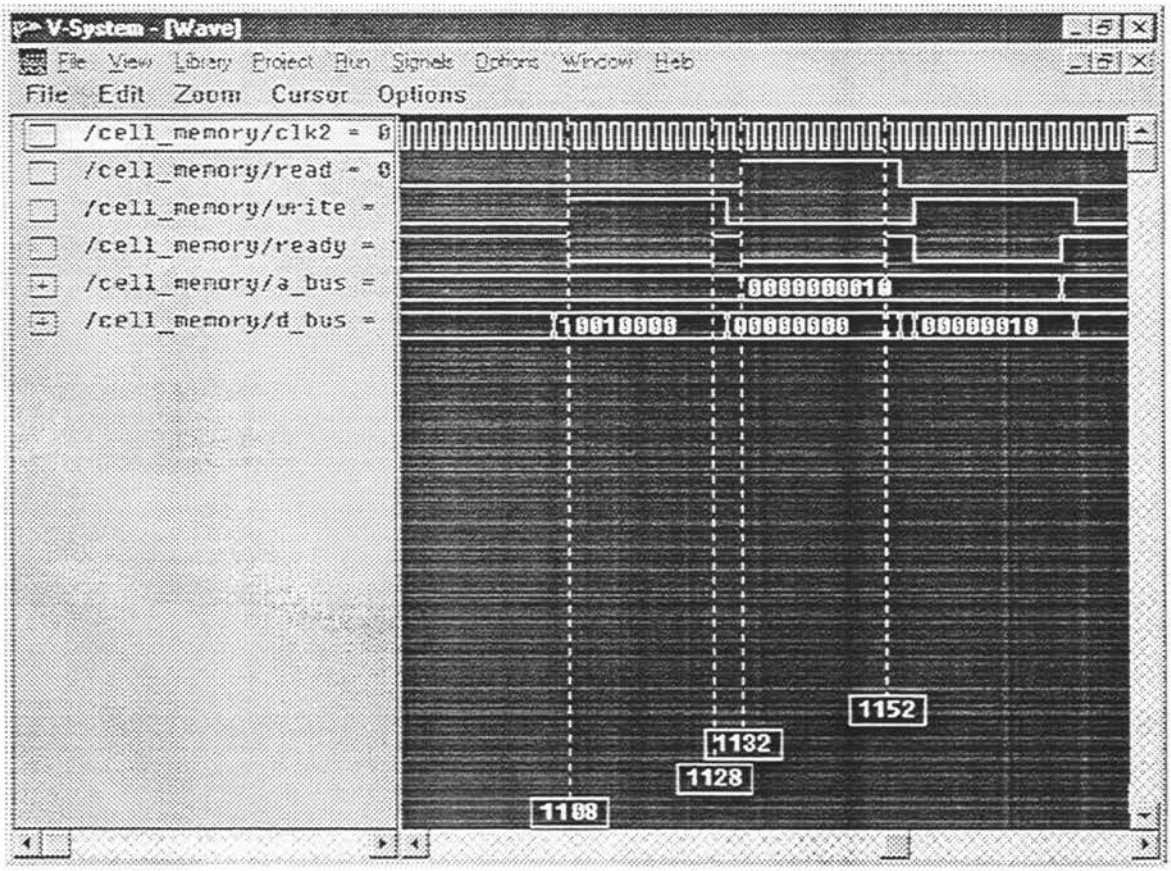


Figure 5.6: Timing of Cell Memory

Timing Relationship of ATMSWITCH

From above description, it can be seen that the ATMSWITCH takes 12 *clk2* cycle to transfer a cell from the input port to the cell memory. 2 *clk2* are used to check and locate the destination address within the buffer and 10 *clk2* cycles are used to write the cell into the buffer. The timing relationship among all the signals of the switch circuits is shown in Figure 5.7.

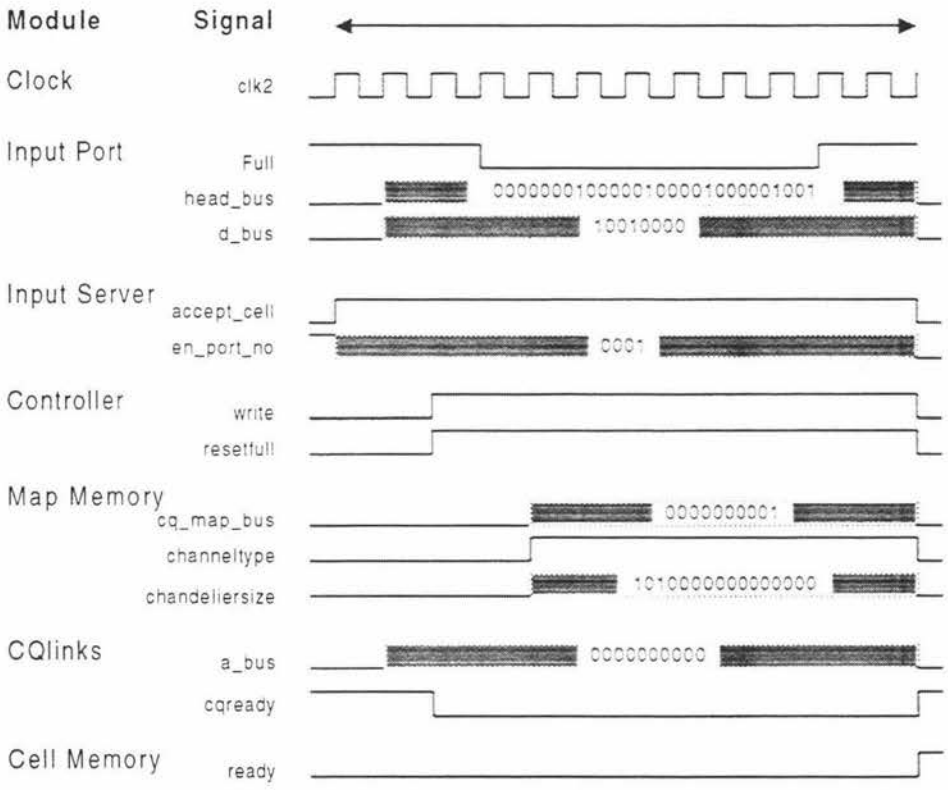


Figure 5.7: Timing Relationship of ATMSWITCH

In the above diagram, *clk*, *acceptcell* and *en_port_no* change from '0' to '1' at beginning of *clk2* cycle. It is assumed that the input server has collected a cell from the input line before the *clk2* event. The *full* signal is already TRUE before the first *clk2* cycle point. Thus the input server is available to check the value of the *full* signal from the input port before first *clk2* cycle.

At 2nd *clk2* cycle, the input port is enabled by the *en_port_no* signal and outputs the cell's payload onto *d_bus* and the cell's routing information onto *head_bus*. At the same time, the *cqlinks* module is triggered by the *clk1* signal and outputs an available cell memory address. The controller reads the cell head from the *head_bus* and begins to check the type of the incoming cell.

At 3rd *clk2* cycle, the controller has finished its checking operation and asserts the *write* signal. The cell memory is thus enabled by the *write* signal to store the cell data into the memory. At the same time, the controller asserts the *resetfull* signal which is delivered to the selected input port.

While the cell memory module is copying a cell into its storage, the other circuit modules continue to operate. At 3rd *clk2* cycle, the map memory module receives the *write* signal from the controller. It starts to compare the incoming cell channel identification information with the channel information list in map memory. After it has finished the mapping operations, at the 5th *clk2* cycle, it outputs the *channeltype* signal and *cell queue tail pointer* to the CQlinks module for updating pointers in CQlinks. Meanwhile, the map memory outputs the updated *chandeliersize* value to the output server.

At 11th *clk2* cycle, the input port receives a new incoming cell from the switch's input line and thus the *full01* signal of input port #1 is automatically set to high.

At 12th clk2 cycle, the cell memory completes its cell writing operation and outputs a *ready* signal to the switch. The *ready* signal notifies the related circuit modules that the cell memory has finished its operation. The input port thus stops outputting the cell onto the bus and the *cqlinks* module stops outputting its address. At same time, the *acceptcell* stage is complete. The *acceptcell* signal, *en_port_no* signal, *write* signal, and *resetfull* signals are reset to initialise the state of the switch for the *outgoing* cell state.

5.1.2 Outgoing Cell

When the low-going edge of the *clk1* signal occurs, the switch enters the *outgoing* state. Each module of the switch performs some part of the work involved in sending a cell from the cell memory to the output port. As in the *outgoing* state, the timing for each circuit module is analyzed below.

Output Round Robin Server

The Output Round Robin Server indicates the current output port. It incorporates with the map memory module to determine whether the current output port can get a cell from the chandelier. The output line operates at a fixed speed, so each output port is assigned an *outgoing* cell slot for transferring cell from the cell memory to its port. During the assigned *outgoing* slot, the output port will not receive a cell if neither a CBR/VBR channel is triggered nor a cell buffered in chandelier cell queue for the current output.

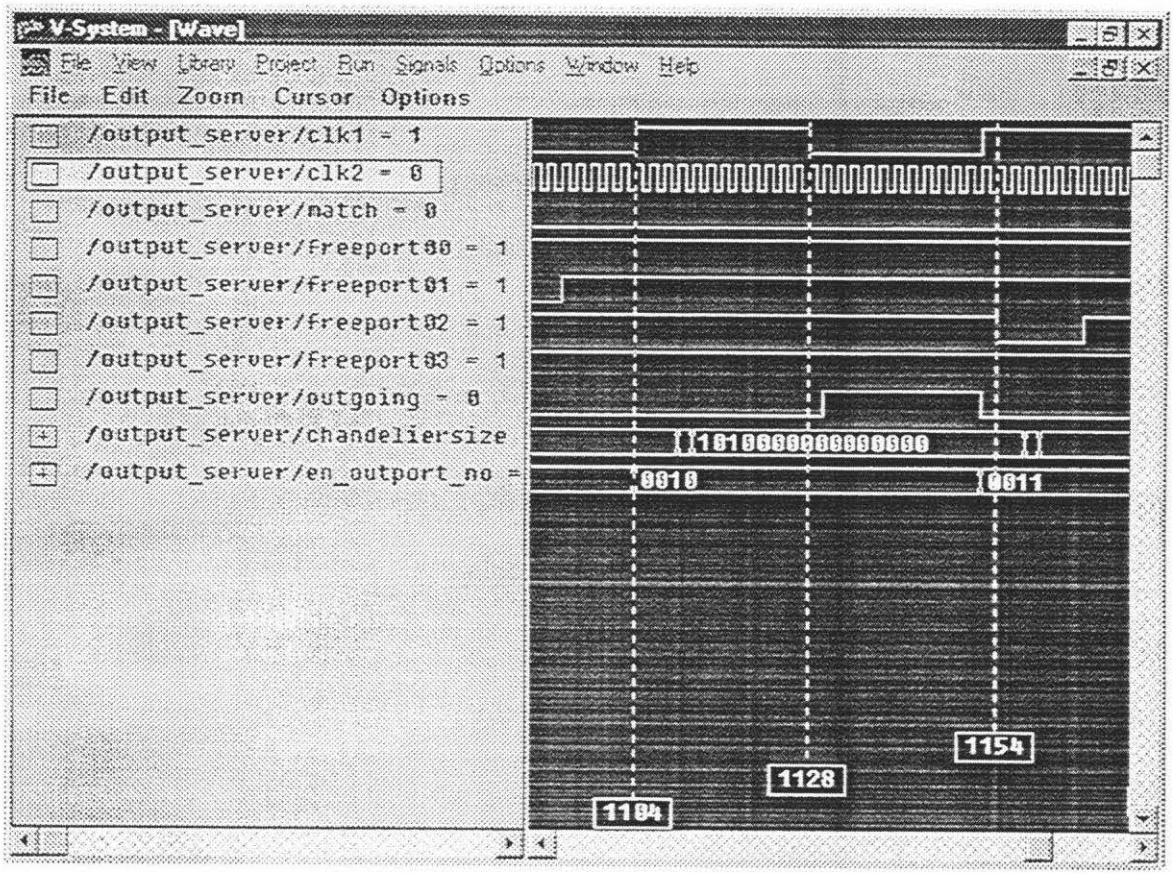


Figure 5.8: Timing of Output Server

The output server works before *outgoing* cell state. It outputs the current output port when the *clk1* signal goes high. During the *acceptcell* state, the map memory outputs the updated chandeliersize value to the output server at 5th *clk2* cycle during *acceptcell* state, as shown in Figure 5.7. The output server uses this value to check the chandelier buffer of the current output port at the *acceptcell* state. The result will be output when the *clk1* signal changes from high to low. All checking of the output server is completed before the *outgoing* cell state. Thus the current output port signal, *en_outport_no*, is available to map memory when the switch state changes from *acceptcell* to *outgoing*. So map memory

starts its trigger operation at the beginning of the *outgoing* state. If the counter number matches the ITV and *mask* value, map memory asserts the *match* signal. The *outgoing* cell comes from the triggered channel rather than chandelier circular list. If the *match* signal is FALSE and the *outgoing* signal is high, a cell is taken from the chandelier buffer rather than the trigger mechanism.

Figure 5.8 shows the signal waveform generated by the output server. It shows the output server outputting the current port number at beginning of the *clk1* cycle. At point = 1104, the *clk1* signal goes high, and the current output port signal *en_outport_no* value is thus changed from 0001 (1) to 0010 (2). At beginning of next *clk1* cycle, the *en_outport_no* is changed from 0010 (2) to 0011(3). After 4 *clk2* cycles from the point = 1104, the map memory outputs the updated chandeliersize value to the output server, and the output server begins to check the *free* signal of output port #2 and the information in the chandelier buffer. If the chandelier buffer of port #2 is not empty and the output port #2 is free, the output server will assert the *outgoing* signal at the *outgoing* state. Figure 5.8 shows that the *outgoing* signal is asserted after 1 *clk2* cycle from the point = 1128. The *outgoing* signal is reset when the *clk1* signal goes high.

The timing diagram for the output server shows that it operates concurrently with other circuit modules. It takes 1 *clk2* cycle to output the *outgoing* signal, but otherwise, the selection and checking operations which it performs do not add to the overall cycle time for the ATMSWITCH.

Output Port

The output port receives a cell from the cell memory and transfers it to the output line of the switch. After it has been output, the output port's *free* signal is set to indicate that the output port is free to receive a cell again. After a cell is loaded into the output port, the free signal is reset to indicate that the output port can't receive another cell until it has output the current cell. So the *free* signal is used by the output server to check whether the current output is available or not.

Figure 5.9 shows the timing while the output port is running. At point = 1104, the output server sends the value 0010 to the output ports using the signal *en_outport_no*. Output port #2 is selected and the *outgoing* cell slot is assigned to output port #2 at the current switch working cycle. From the point = 1130, output port wait for the *ready* signal which will be from the cell memory. The high phase *ready* signal means the cell memory has

finished its reading operation and the cell data is available on the *d_bus*. Output Port #2 is thus able to receive the cell data from the *d_bus*.

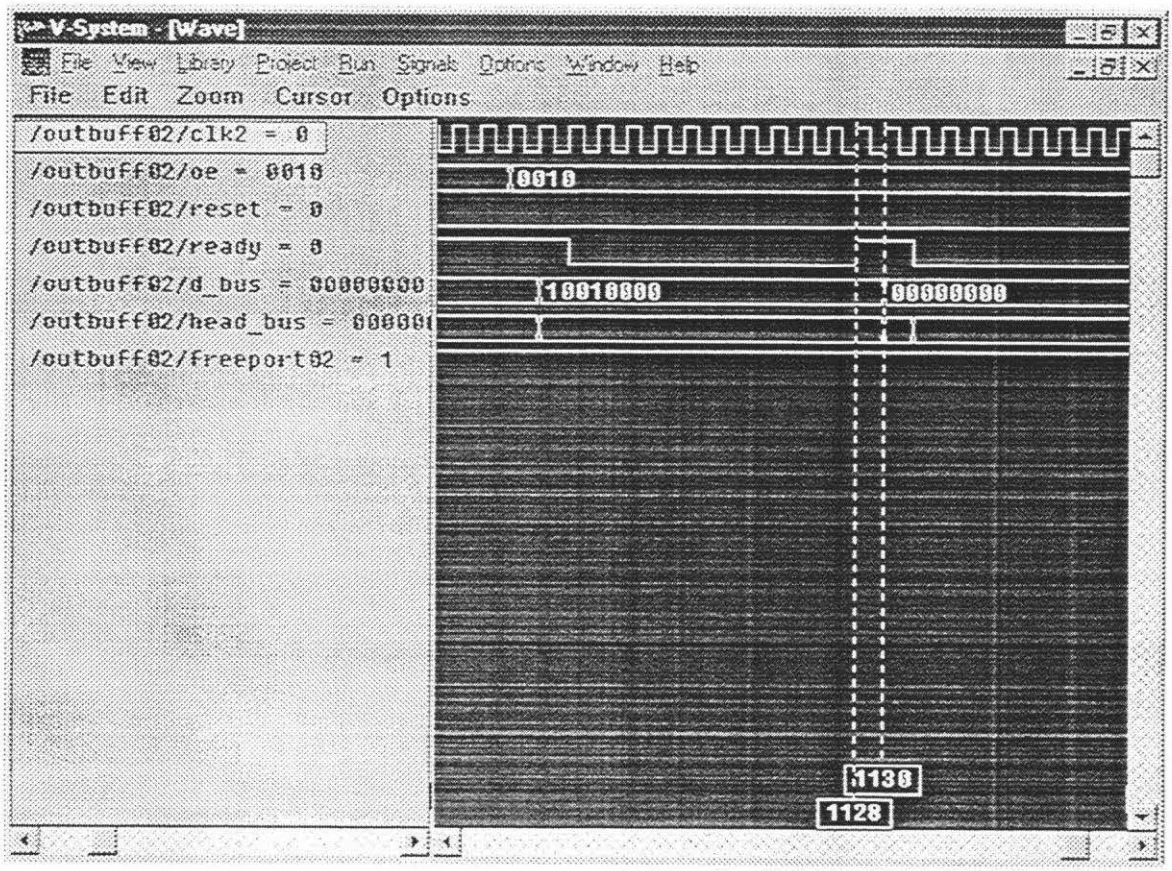


Figure 5.9: Timing for the Output Port

The simulation shows that the output port #2 doesn't take extra time to get the cell during the switch's read and write cycle.

Controller

During the *acceptcell* state, signals from the controller are used to cause the switch modules to transfer a cell from the input port to the cell memory. During the *outgoing* cell state, signals from the controller are used to cause the switch modules to send a cell from cell memory to the output port. When the *clk1* signal changes from high to low, the controller initiates the operations belonging to the *outgoing* cell state. It waits for the *outgoing* and *match* signals. If either of these signals becomes TRUE, the controller asserts the *read* signal so that the cell memory is enabled to read a cell from given address.

Figure 5.10 shows a timing diagram for the controller during the *outgoing* state. *Clk1* goes low at point = 1128. When the controller detects the low *clk1* signal, it checks the *match* and *outgoing* signals. A TRUE *match* signal means that one of CBR/VBR channel is due to be triggered and a TRUE *outgoing* signal means that the chandelier buffer of the current output port is not empty. After 1 *clk2* cycle from the point = 1128, the controller receives a TRUE *outgoing* signal. It thus asserts the *read* signal after 2 *clk2* cycle from the point = 1128. Cell memory begins to read cell and set the *ready* signal high after it finished reading. The *read* signal is reset to low after the controller receives the high phase *ready* signal from cell memory.

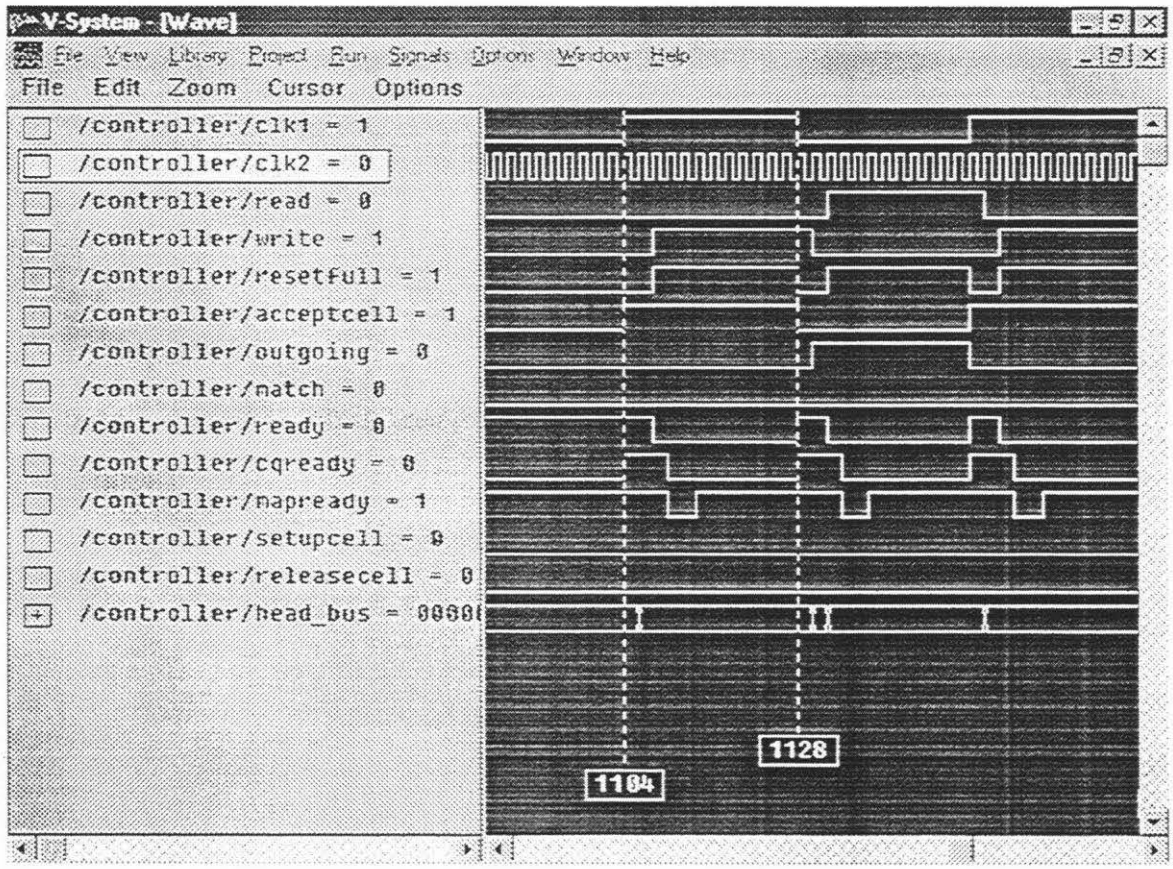


Figure 5.10: Timing for the Controller

Simulation the controller shows that it spends 1 *clk2* cycle outputting the *read* signal when it receives the *outgoing* signal. From the time between the *clk1* signal going low and the *read* signal being set, the switch spends 2 *clk2* cycles on the addressing and re-assembly cell operations.

Map Memory

The map memory maps the incoming cell to the cell buffer during the *acceptcell* state. During the *outgoing* cell state, map memory uses two special devices, the trigger mechanism and the chandelier to select an output channel.

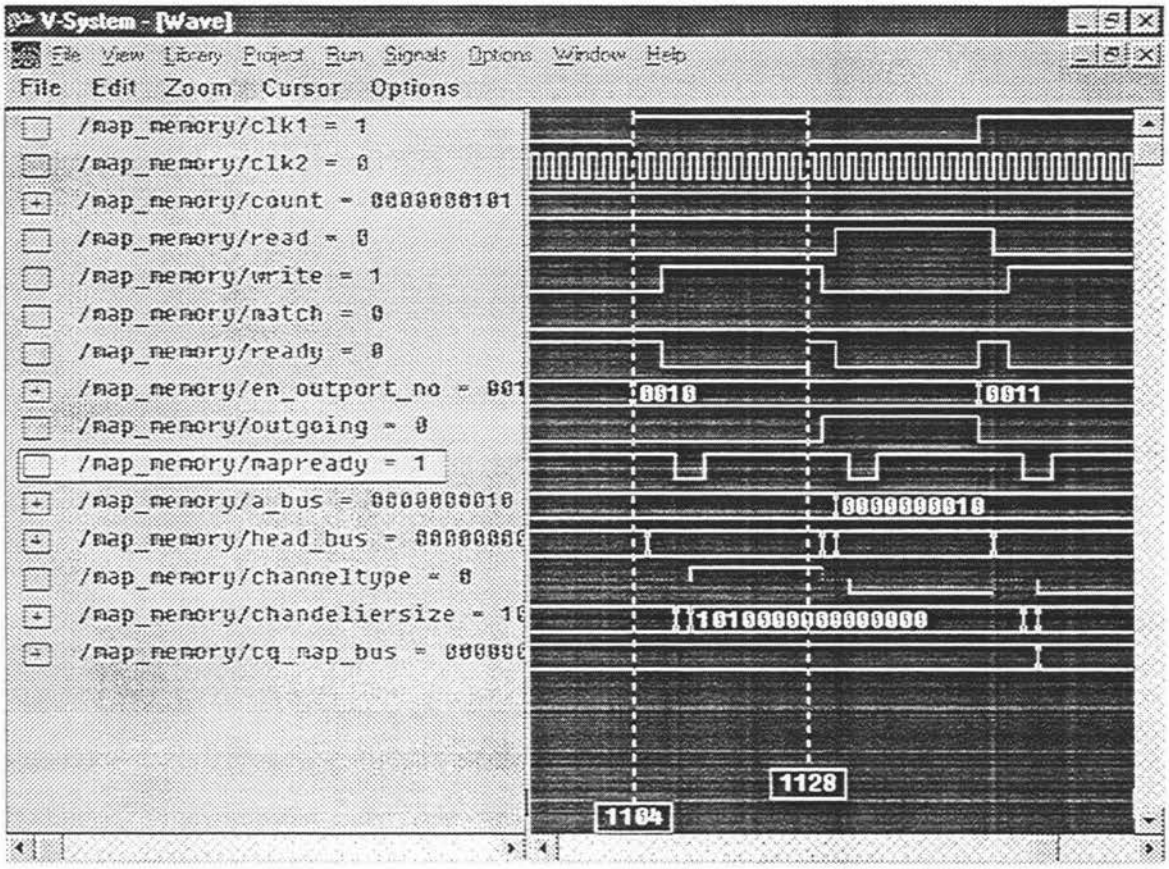


Figure 5.11: Timing for the Controller

Figure 5.11 shows the timing of map memory while it was running. Because the timing for the map memory during the *acceptcell* state has been described before, here we only introduce the timing for the map memory during the *outgoing* cell state. At point = 1128, the *clk1* signal goes low and map memory thus start to perform the operations associated with the *outgoing* cell state. Map memory first reads the current counter value and current output port number. It performs a masked comparison between the counter number with ITV and *mask* value of each channel. If the matched channel holds a cell and the port

through which its cells are output is the current output port, the matched channel is triggered. The map memory outputs the high phase *match* signal to the controller. Than the map memory transfers the cell queue head pointer of the triggered channel via *a_bus* to the cell memory so that it can read the cell from the triggered channel. If *match* is false, or if it is true, but the queue length for the triggered channel is 0, no CBR/VBR channel is triggered, and instead, the map memory checks the *outgoing* signal. If the *outgoing* signal is high, it means the chandelier buffer is occupied. Map memory selects the current channel from the circular list of the chandelier and outputs the cell queue head pointer of the selected channel to the *a_bus*. In this case the cell is transferred from chandelier buffer rather than trigger mechanism.

In Figure 5.11, the map memory starts the trigger operation at point = 1128. It determines that no CBR/VBR channel is triggered. The map memory then checks the *outgoing* signal. The high *outgoing* signal reaches at the map memory after 1 *clk2* cycle from the point = 1128. It means the chandelier buffer of current output port #2 is not empty. The map memory thus outputs an address by using the cell queue head pointer of the current channel header in the circular list. After 2 *clk2* cycles from the point = 1128, the cell memory address "0000000010" is put on *a_bus*. After the cell queue head pointer is put onto the address bus, the map memory needs to update the channel header and chandelier register. First the map memory waits for the new cell head queue pointer which will be sent by *cqlinks*. After 4 *clk2* cycle from the point = 1128, the map memory gets the new head pointer through the *cq_map_bus* port. It replaces the old cell queue head pointer with the new one and updates the channel header and the chandelier register. When the *ready* signal of cell memory is set to high. The map memory stops outputting the address.

The timing simulation for map memory shows that it runs simultaneously with other circuit modules. It takes 1 *clk2* cycle to perform the CBR/VBR matching operation and another 1 *clk2* cycle to output the address to the *a_bus*, so the cell at this address is output from the cell memory. But these 2 *clk2* cycle time do not add to the switch read and write time because they occur while other circuits are active.

Cqlinks

When CQlinks detects a low-going edge on the *clk1* signal, it begins the *outgoing* cell operation. During the *outgoing* cell state, CQlinks releases the word in cell memory which was occupied by the newly output cell and updates the *Freecell* register value and cell queue link pointer.

Figure 5.12 is a timing diagram for the operations performed by CQlinks during the *outgoing* state. It shows, a low-going edge for the *clk1* signal at point = 1128. 2 *clk2* cycle later, CQlinks receives a *read* signal. It fetches the current cell memory address from *a_bus*. The cell which is stored at this address will be removed from the cell memory. CQlinks thus releases this space for the cell memory and gets the content of the released cell memory address in the CQlinks. The content of released address in the CQlinks is the link pointer of cell queue. It points to the next cell address of cell queue. Because the first cell of the cell queue has been removed, the link pointer is thus the new cell queue head address. After 4 *clk2* cycles from the point = 1128, a low voltage on the *channeltype* signal triggers CQlinks to send the new cell queue head pointer to the map memory through the *cq_map_bus* port. Meanwhile CQlinks updates the current *Freecell* register value and link pointer for the newly released address. At point = 1152, Cqlinks completed its maintain operations and thus set the *cqready* signal to high.

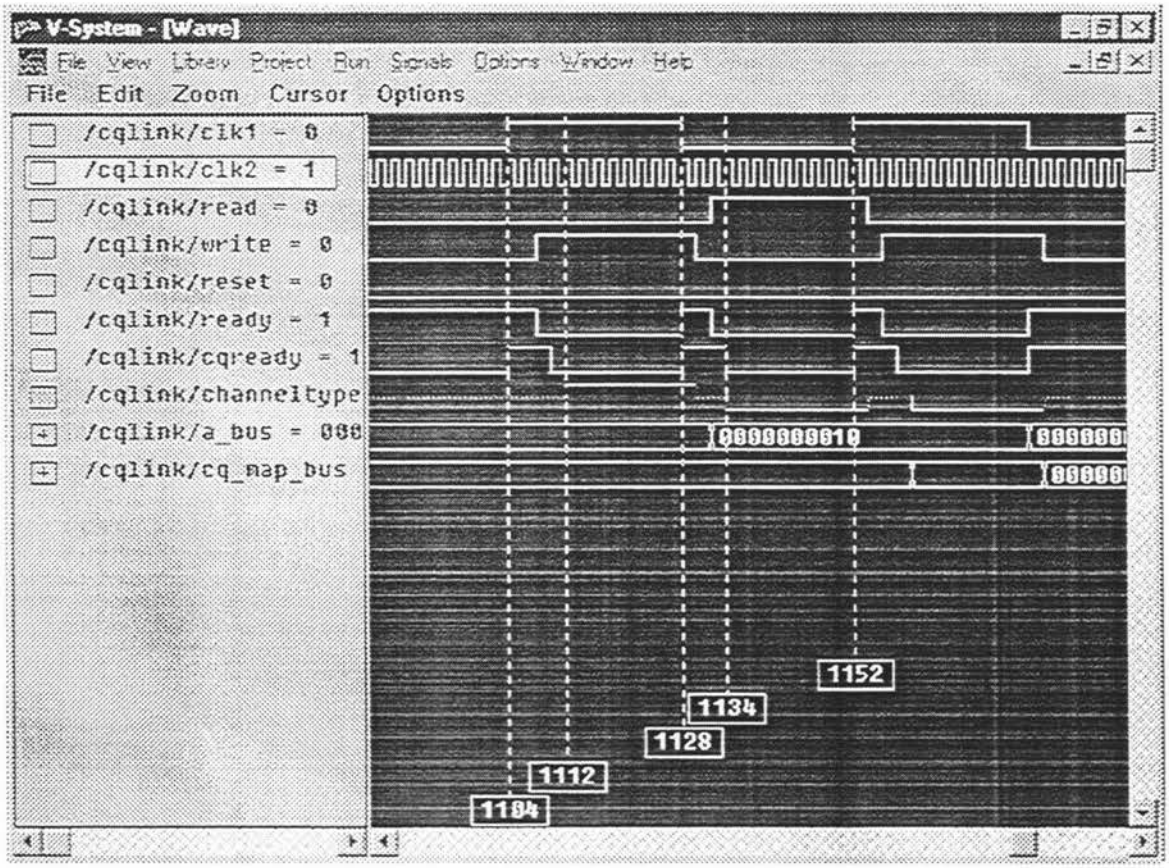


Figure 5.12: Timing for the CQlinks

The timing simulation for CQlinks shows that CQlinks took 8 *clk2* cycles to update its link point *Freecell* value. It works simultaneously with other circuit modules at *outgoing* cell state. It doesn't lengthen the switch cycle in order to update its values.

Cell Memory

In the ATMSWITCH, It takes 2 *clk2* cycles from the low-going edge of the *clk1* signal to the time when the controller outputs a TRUE *read* signal. That means the switch need 2 *clk2* cycles to address the cell memory for reading a cell. The 12ns time for cell memory reading operation also required.

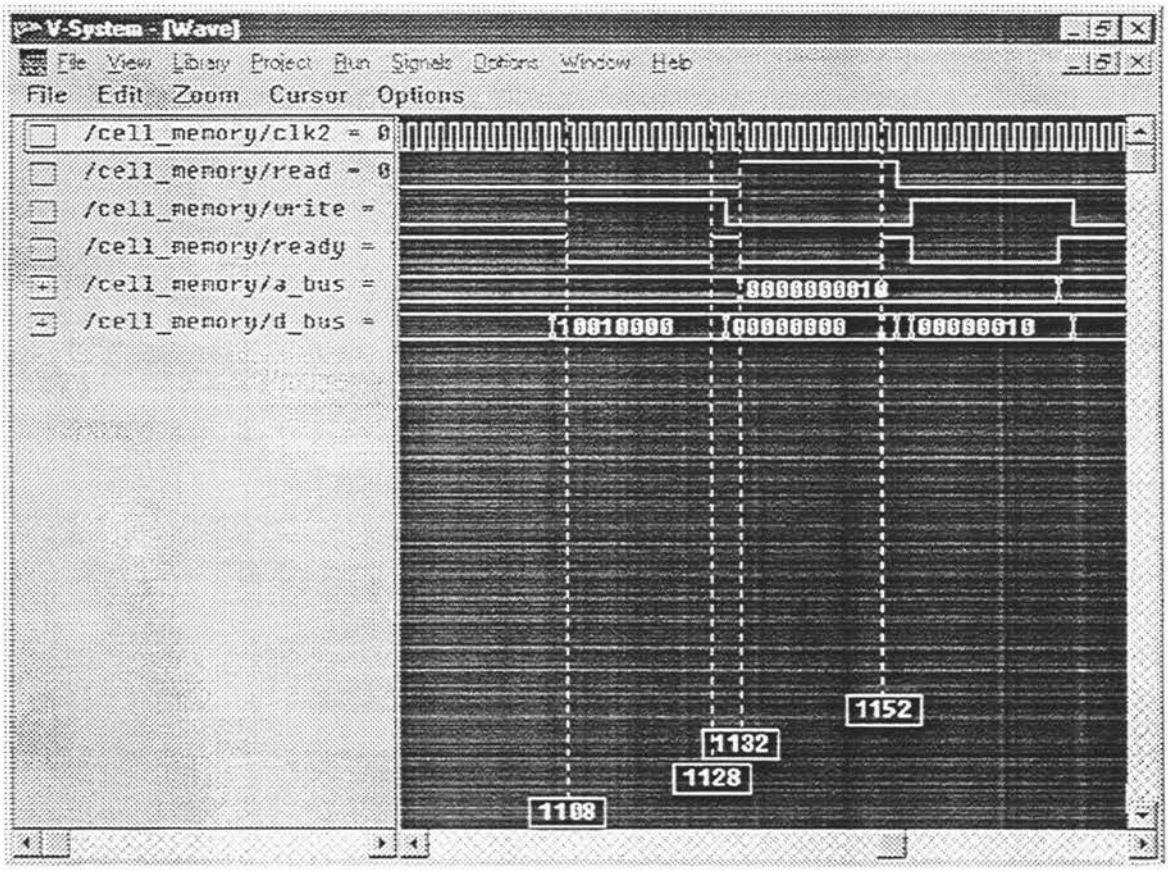


Figure 5.13: Timing for the CQlinks

Figure 5.13 shows a timing diagram for a simulation of cell memory. It shows a cell memory address, "0000000010", being generated by map memory at point = 1132. At the same time the *read* signal is set. Cell memory is thus enabled to read cell from the given

address and put the cell data to the *d_bus*. At point = 1152, the cell memory has finished the reading operation and sets the *ready* signal to high.

Timing Relationship of ATMSWITCH

As the above descriptions illustrate, the ATMSWITCH takes 10 *clk2* cycle to read a cell from cell memory. 2 *clk2* cycles is used to address cell memory. The relative timing for all the switch circuit signals is shown in Figure 5.14. The signal events that happen during the *outgoing* state are copied from the simulation runs for the individual modules which were described above.

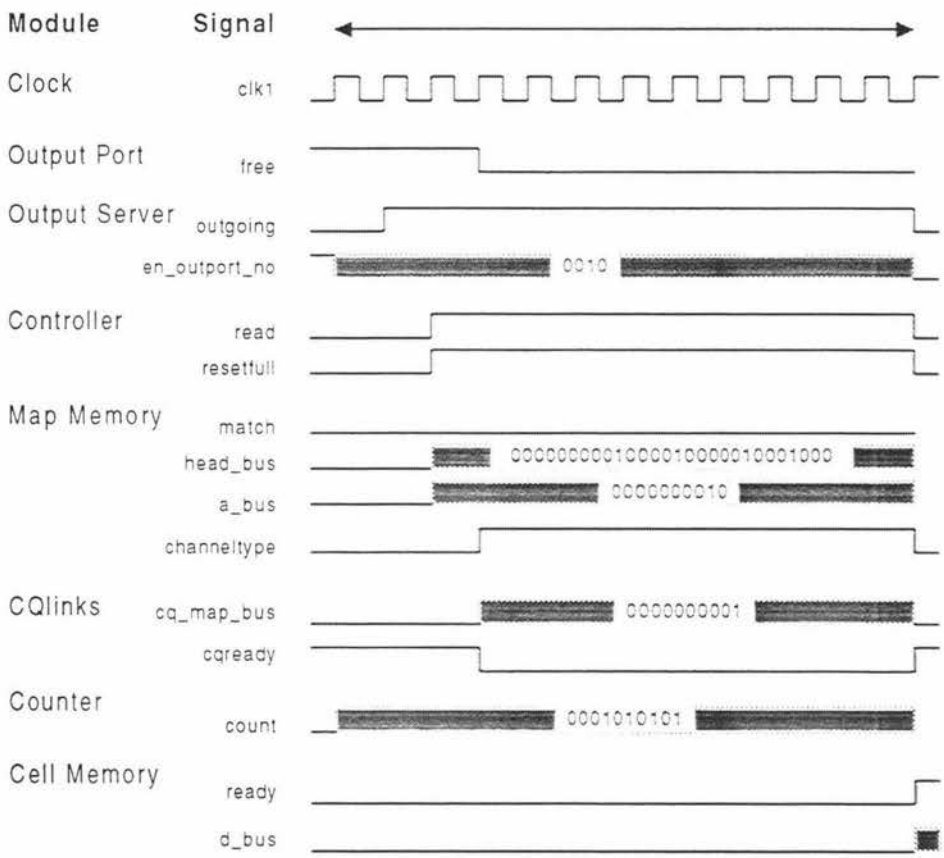


Figure 5.14: Timing Relationship of ATMSWITCH

The diagram shows that the *clk2* signal at whole *outgoing* state. Before 1st *clk2* cycle, the *free* signal, *en_outport_no* signal and *counter* value have already been set by their modules. As described before, the current output port number, *en_outport_no* signal is set

by the output server at beginning of each switch working cycle. At the *outgoing* cell time, it's certainly available for other modules to read. The *free* signal represents currently available output ports. It is set to high at any time if the output port is empty. The counter number is set at beginning of switch working cycle as well. Therefore, the map memory can read the *counter* signal and *en_outport_no* signal at the beginning of the *outgoing* cell state.

At beginning of 2nd *clk2* cycle, the output port outputs the *outgoing* signal to the controller and map memory modules. Meanwhile, the map memory outputs the *match* signal to the controller. Whatever the *match* signal or *outgoing* signal, if one of them is set, the controller will set the *read* signal. In this case, only the *outgoing* signal is set, so the channel is selected from the chandelier rather than trigger mechanism.

At beginning of 3rd *clk2* cycle, the controller has finished the checking operation and sets the *read* signal. At the same point, the map memory finishes the channel selection operation and outputs the *cell queue head pointer* from the header for the selected channel to the *a_bus*. Cell memory is thus enabled by the *read* signal to read the cell from the given address. At the same point, the controller also asserts the *resetfull* signal and the map memory outputs the re-assembled cell head onto *head_bus*.

Other circuit modules are operating while the cell memory is reading the cell. At 3rd *clk2* cycle, CQlinks receives the *read* signal and thus begins to update the value in the *Freecell* register and release the word in cell memory which contained the cell. At beginning of 4th *clk2* cycle, CQlinks transfers the new *cell queue head pointer* to map memory so map memory can replace the old cell queue head pointer with the new one.

At 12th *clk2* cycle, the cell memory has completed reading operation and puts the data onto *d_bus*. At the same time, it outputs the high phase *ready* signal to notify the related circuit modules that the cell is ready to pick up. The output port thus fetches the cell from the *d_bus* and map memory stops outputting the address. At same point, the switch *outgoing* cell is complete and the *clk1* signal is changed from low to high. Other control signals, such as *outgoing*, *en_outport_no*, *resetfull* and *match* are thus reset to their initial states. The *outgoing* state ends and the switch enters the *acceptcell* state.

5.1.3 Clock Speed and Switch Working Cycle

As described in last two sections, the cell buffer is built out of commercially available cache memory chips with a 12ns reading and writing time. According to the simulated

result of ATMSWITCH timing, the time between the high-going edge of the *clk1* signal and the *write* signal generated by the controller is **2** *cll2* cycles. The time between the low-going edge of the *clk1* signal and the *read* signal generated by the controller is **2** *clk2* cycles as well. This means that the switch needs 2 clock cycle to prepare for writing a cell into the buffer and another 2 clock cycle for reading a cell from the buffer. According to the simulation result, the **clock cycles** which is required if the control operations are to fit into the 12ns memory write or read time has been counted as **10** *clk2* cycles. So the **clock speed** for the controller will be:

$$\textit{clock speed} = \textit{12ns/clock\_cycles\_no} = \textit{12ns/10} = \boldsymbol{1.2ns};$$

Therefore, the total time of switch for either accepting or outgoing a cell is:

$$\begin{aligned} \textit{acceptcell\_outgoing} &= \textit{buffer write\_read} + \textit{preparation} \\ &= \textit{12ns} + \textit{2 clcok cycles} \times \textit{1.2ns} = \boldsymbol{14.4ns}; \end{aligned}$$

So the total period of the **write cycle** or **read cycle** will be:

$$\textit{write cycle} = \textit{read cycle} = \textit{14.4ns};$$

Consequently, the total write and read cycle (switch **working cycle**) is *14.4ns* + *14.4ns* = ***28.8ns***, and the **clock speed** is ***1.2ns***.

5.2 Cell Memory Size Requirement

Owing to the finite memory size and the lack of co-ordination among arriving cells as far as their destination requests are concerned, some cells may not be accepted by the switch and be lost. The memory size should then be determined so as not to exceed a certain maximum cell loss rate. The size of the memory required is a function not only of the size of the switch N, the offered load p, and the traffic pattern, but also of the way the memory is shared among the various output queues. In this section, we use our simulated result to analysis how much size of the memory should be designed in ATMSWITCH. Two different traffic situations are considered to analysis required memory size.

5.2.1 Cell Memory Size for Simple Traffic Situation

It is assumed that the cells from input line of switch are arriving at a constant rate equal to the maximum bit rate supported by the line. It also assumed that the load on the input lines

is equal to the capacity of the output lines. In this special case, in which there is no burst data, and no output port overloading the analysis is quite simple. We simulate the behaviour of the switch t = 0ns to t = 9783290ns. Because we are interested in the behaviour of cell memory, and cell memory is controlled by CQlinks, we record the values of the CQlinks variables during this time. This allows us to determine the used and free memory space of the cell memory at t = 9783290ns. From the time when the switch is turned on till t = 9783290ns, a total 407619 cells has passed the switch. Figure 5.15 shows the values of the CQlinks variables during this time.

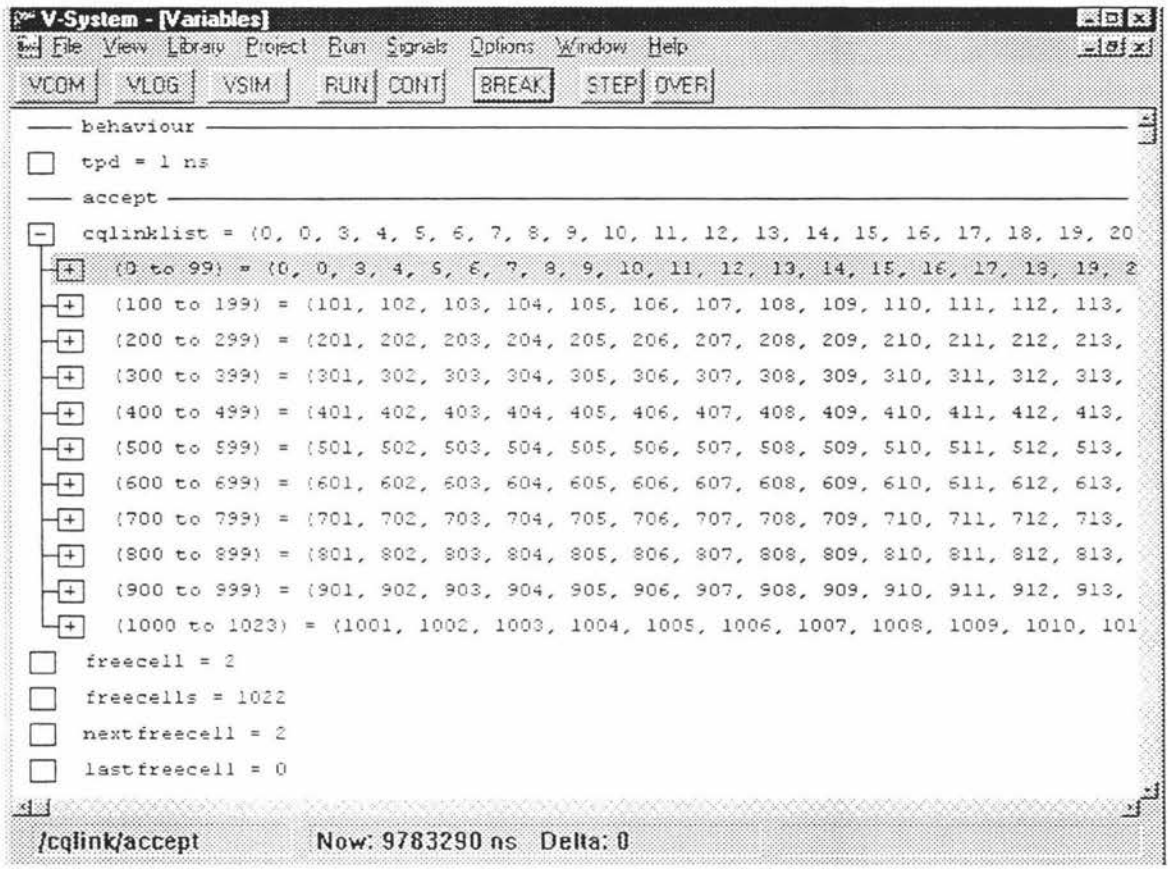


Figure 5.15: Simulated Result of CQlinks Variables

In the diagram, each address of the CQlinks value is listed. It shows the memory space from 0 to 1023. The highlight line in the diagram shows the first several values of CQlinks memory address. We know that the CQlinks is initialised at switch turned on so that the the value i+1 is stored at CQlinks[i] for all values of i in the range 0 to 1023.

The diagram shows that only the first 2 words of CQlinks are used. The value of the variable *Freecells* is, correspondingly, equal to 1022. This result is to be expected because the switch loading was assumed to be ideal, in which a cell will be transferred from the input port to the cell memory during the *acceptcell* state and sent out of the switch at *outgoing* cell state. So one cell memory address is used to store incoming cell during the *acceptcell* state and another cell memory address is released during the *outgoing* state. At the end of each switch working cycle, the cell memory size is still same as the beginning of switch working cycle. This is why only two spaces are being used at the time 9783290ns. In this special case, the number of words in cell memory size could be twice the number of ports on the switch, and no cells would be lost.

5.2.2 Cell Memory Size for Output Overload

In last section, the required size for cell memory under ideal (though heavy) traffic conditions is measured. In the case, the ATMSWITCH requests only a few words of memory. Real network traffic patterns are, however, more complex.

In the ATMSWITCH architecture, the input line and output line are designed as fixed speed. The input server module accesses each input port fairly and the output server assigns each output port a time slot periodically as well. If the speed of the source which send cells to the switch is less than the input line speed, the cells it sends will have no problem being accepted by the switch. But if many cells from different input ports are routed to the same output port, the incoming cells must be buffered in cell memory while they wait for an output slot. But this bursty situation should not last long because the channel CAC (Connection Admission Control) has ensure that the load of each output port is not over a load of 100% and the length of bursty data is restricted. Here we give an example to show what size of cell memory is required for output port overloading situation.

Suppose cells that are from two fully loaded input ports are routed to output port #0. So the output port #0 has a load of 200%. Again, we use the results obtained from simulating the behaviour of the CQlinks module to examine the required memory size.

Figure 5.16 shows the value of the CQlinks variables at the t = 600ns. The values of *Freecells* and *cqlinklist* indicate that only two memory address are used at that moment. At t = 28500ns, we again print (Figure 5.17) the values of the CQlinks variables. *Freecells* is 877, which means that 1024-877=147 memory spaces have been used at the time 28500ns.

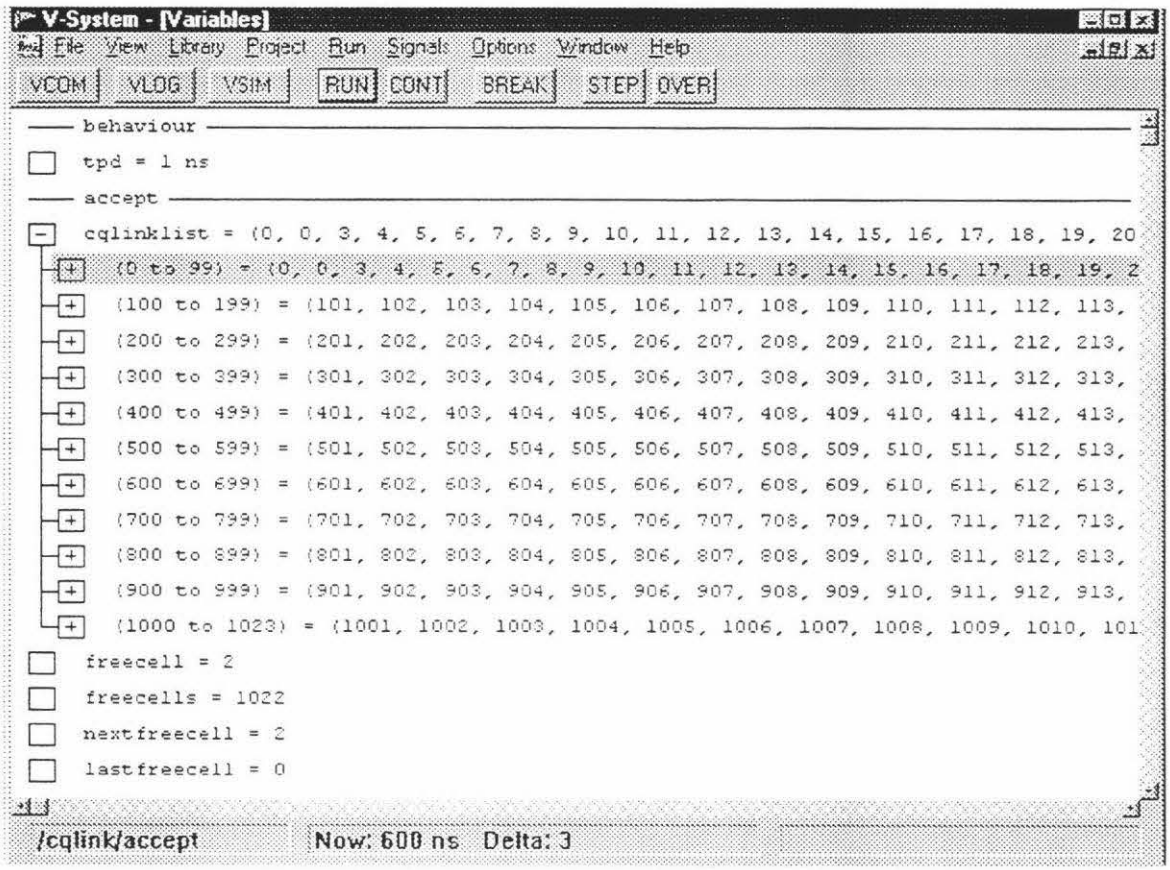


Figure 5.16: Simulated Result of CQlinks Variables at the time 600ns

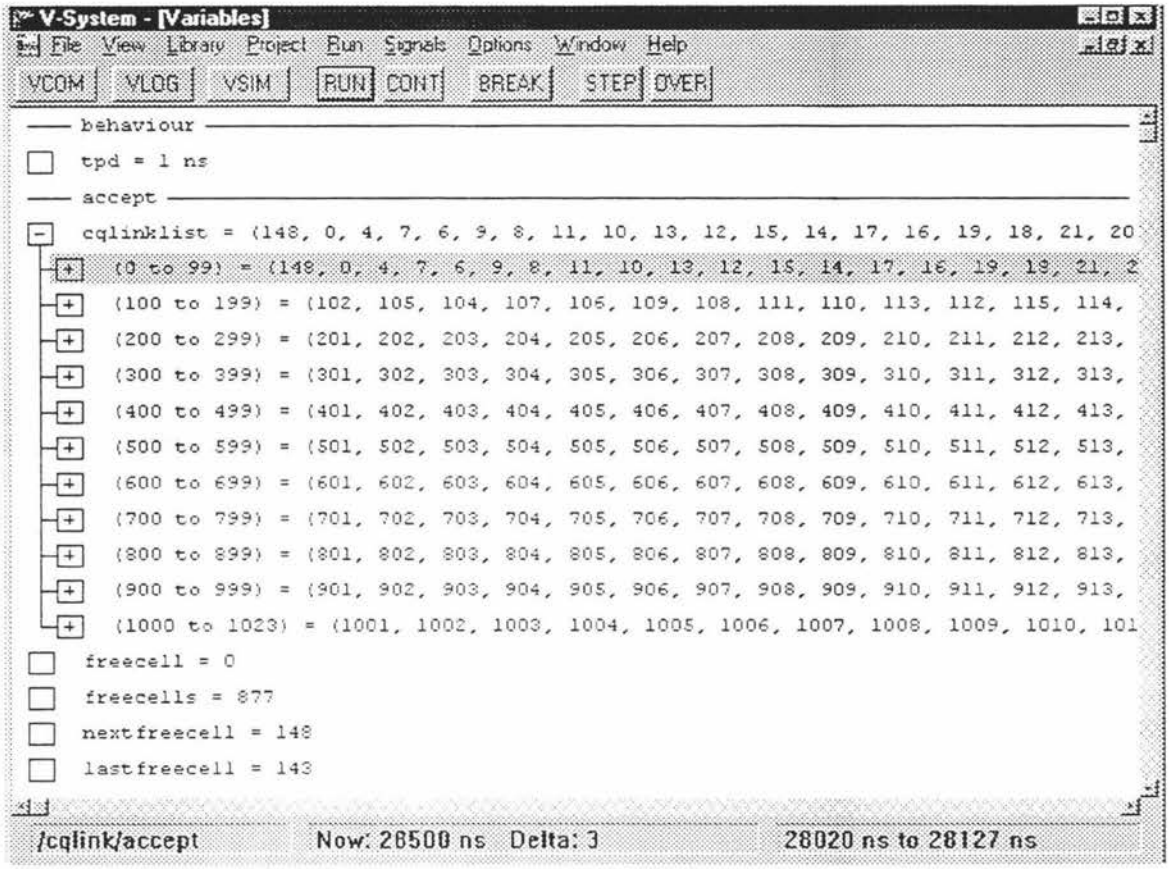


Figure 5.17: Simulated Result of CQlinks Variables at the time 28500ns

In this example, the cell memory needs at least 147 cells to store cells so that no incoming cells will be lost. It seems that the size of the cell memory depend on given traffic situation. Therefore, the cell memory size should be defined by using real traffic pattern to test it and can thus get an appropriate value.

5.3 Throughput

The simulated shows that the ATMSWITCH takes 14.4ns to transfer a cell from input port to cell memory. 2.4ns time is used to check and locate the address of buffer and 12ns time is used to write cell into buffer. At *outgoing* state, ATMSWITCH takes 2.4ns to address cell memory and 12ns to read a cell from memory. During each 28.8ns switch working cycle, the switch can receive a cell from the input line and send a cell out of the switch. So the throughput for ATMSWITCH can be worked out by using following equation:

$$cycle\ time = \frac{celllength}{2 \times N \times linkspeed(b/s)}$$

where throughput $= N \times linkspeed(b/s)$;

So the total throughput for the switch is:

$$\textbf{\textit{throughput}} = 424/(28.8 \times 10^{-9}\ bps) = 14.7222Gbps$$

The maximum port number for ATMSWITCH is:

$$\textbf{\textit{port number}}\ (N) = throughput / linkspeed = 14.7222Gbps / 1.2Gbps = 12$$

5.4 Map Memory Size

The map memory size will depend on the number of virtual channels that the switch is required to handle. The ATMSWITCH uses 12 1.2Gbps input ports and output ports. If all the established channel were 64kbps limited voice and data channels (e.g. ISDN B channels) and the switch were able to create all those channels, then the maximum map memory size required would be (1.2Gbps/64kpbs) × 12 = 225000 channel unit.

5.5 Summary

In this chapter, the simulated result for ATMSWITCH timing is described. The total write and read cycle is 28.8ns. 24ns time is used for cell buffer writing and reading. 4.8ns time is used to prepare buffer writing and reading operations. The clock speed for the controller is worked out as 1.2ns. The performance analysis for the buffer size, map memory size and throughput of ATMSWITCH has also been described.

Chapter 6

Future Work

In shared-buffer switch architectures, the cell memory is shared by all output ports. This approach requires the minimum possible amount of buffering and has the most flexibility to accommodate traffic dynamics, in the sense that the shared memory can absorb large bursts directed to any output. Unfortunately, the approach has its disadvantages. As the cells must be written into and read out from the memory one at a time, the shared memory must operate at the total throughput rate. According to the paper [Tobagi, 1990], it must be capable of reading and writing a cell in every 1/NVs, where N is the number of ports and Vs is port speed, that is, N times faster than the port speed Vs. As the factor N limits the ability of this approach to scale up to large sizes and fast imposes a limit on the product NV, which is the total throughput.

In the ATMSWITCH architecture, very fast cache memory with a read and write cycle time of 12ns is employed for storing cells. A total of 28.8ns time is worked out in last chapter for each switch working cycle. If the speed of switch port line is designed as 1.2Gbps, the maximum port number for this port speed is thus be determined. In this case, the number (N) is equal to 12. It's not possible to add more ports in ATMSWITCH because the memory speed has restricted the total throughput. From other perspective of view, the ATMSWITCH need very fast cache memory to build the cell buffer. The speed of memory chips is thus a bottleneck in ATMSWITCH implementation.

To overcome the restriction caused by the memory speed, an alternative multi-buffer architecture is introduced. The new switch architecture is considered to improve the performance of ATMSWITCH and it's not necessary to require very fast memory to build its cell buffer.

6.1 Multi-Buffer Architecture

In the multi-buffer based switch, each output port has a separate buffer block and those buffer blocks receive cells in parallel. During each switch working cycle, as many as N cells (N is the number of input and output ports) can be written into the buffer blocks during the *acceptcell* state and as many as N cells can be output from the buffer blocks

during the *outgoing* cell state. Because each buffer block is working separately, it is possible to build large scale switch without memory speed restrictions. In addition, if multicasting function is designed into a shared buffer switch architecture, it would increase the complexity of the controller because it need more control signals and algorithm to achieve it. But in a multi-buffer architecture, it will be easily realised. Therefore, the multi-buffer switch architecture can improve the total throughput of switch and reduce the required cell memory speed. Figure 6.1 shows the multi-buffer switch architecture.

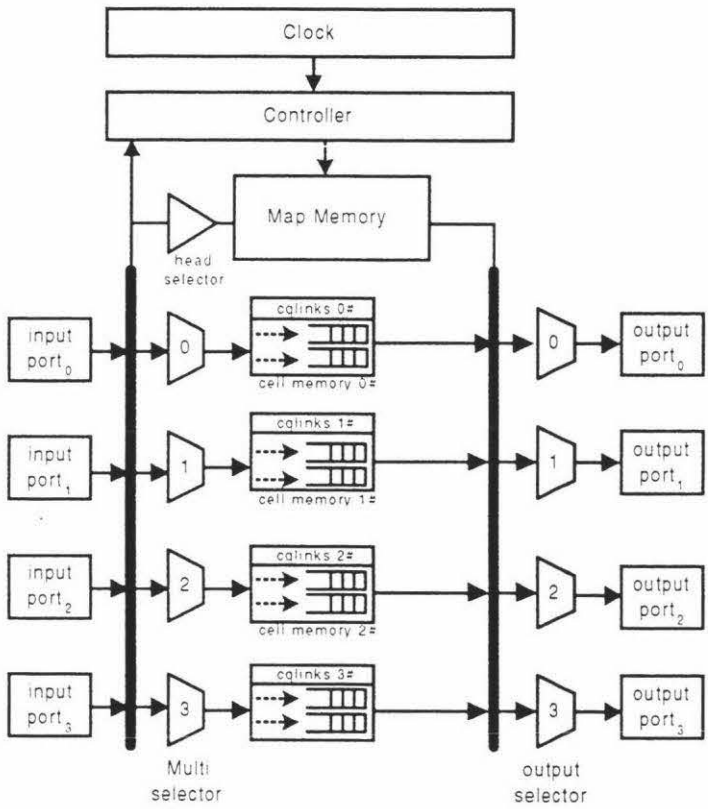


Figure 6.1: Multi-Buffer ATM architecture

Figure 6.1 shows the architecture of a multi-buffer switch. It is based on the shared-buffer architecture for the ATMSWICH which was shown in Figure 4.1 in chapter 4, except that it contains N buffer modules (where N is the number of output ports) instead of 1. Most of the component modules have been inherited from the shared-buffer architecture to the new one.

The major distinction between shared-buffer architecture and the multi-buffer architecture is that the first only uses one buffer module which is shared by all output ports. During each working cycle, cell memory reads one cell from the input port and

sends one cell to the output port. The multi-buffer switch gives each output port a buffer block and all of the buffers are able to work in parallel. Therefore, the multi-buffer switch can receive as many as N cells from input port and send as many as N cells to output port simultaneously.

6.2 General Operation of Multi-Buffer Architecture

In the multi-buffer architecture, the switch still has two states, the *acceptcell* state and the *outgoing* cell state. During the *acceptcell* state, the controller causes the switch to transfer cells from input ports to cell memory blocks. During the *outgoing* cell state, the controller causes it to transfer cells from the cell memory blocks to the output ports with which they are associated.

At beginning of each *acceptcell* state, the controller checks the *full* signals of each of the input ports. If current input port is not empty, it signals the head-selector module to enable the current input port to send its cell header to map memory. Map memory maps the incoming cell channel id and input port number onto a channel information record in the map memory, and then map memory outputs the *input_port_number* and *output_port_number* values for the mapped channel to the multi-selector. Each multi-selector corresponds to a buffer block and an output port, which all share the same unique id number After map memory outputs the *output_port_number* signal, the corresponding multi-selector is triggered. The multi-selector thus enables the indicated input port (by using the *input_port_number* signal) to send its cell data directly to the buffer block. The controller checks each *full* signal of all the input ports at each pre-defined time slot. Map memory works with the controller to map each cell header and output the *input_port_number* and *output_port_number* signal. It only takes a short time to achieve this job during the *acceptcell* state; the controller and map memory are designed to be very fast. As in the earlier single-buffer version of the ATMSWITCH, map memory and CQlinks updates their values simultaneously while the memory block is writing cells into the buffer.

Within one *acceptcell* state, the maximum number of cells which switch get in is equal to switch size, N. It means that all cells in the input port are accepted by the memory blocks. And it also means that no more than two cells are routed to the same buffer blocks in this case. At each *acceptcell* state, the minimum number of cells can be counted as one, suppose all cells of input ports are going to the same output buffer.

At each *outgoing* cell state, the controller checks the *free* signals of each output port. If the output port is empty, it signals to output selector. At the same time, map memory gives the current *outgoing* cell address to each output port and outputs the re-assembled cell head onto the *head_bus*. The current *outgoing* cell is selected either from the CBR/VBR cell queue or chandelier buffer. All memory blocks are able to send the cells to the output ports at the same time because they all work concurrently.

6.3 Summary

In this chapter, a multi-buffer switch architecture has been brief introduced. Actually, the new proposed architecture is based on ATMSWITCH architecture. The most modules of ATMSWITCH switch have been inherited to the multi-buffer switch. The major distinction between ATMSWITCH architecture and multi-buffer architecture is that the ATMSWITCH is a shared buffer structure, but the multi-buffer switch assigns a buffer to each output port. At a switch working cycle, only one cell can be transferred from the input port to the output port in ATMSWITCH. But as many as N (input and output ports size) cells can be transferred from the input ports to the output port in multi-buffer switch. Therefore, the multi-buffer switch architecture can improve the total throughput of switch and reduce the required cell memory speed.

The future work for the multi-buffer switch is to design each module in detail. Although the main structure is proposed, but the functions of each module, and details of circuit design need to be completed.

Conclusion

The task undertaken in this project, and described in this thesis was the development of a full logic design for the chandelier and trigger components of the ATMSWITCH, and the determination of the speed at which this logic would have to run if full use is to be made of the memory bandwidth.

The full logic design of ATMSWITCH in this thesis has been completely achieved. The number of clock cycles required to perform the buffer maintenance activities, and the logic speed required to fit this number of clock cycles into the 12ns memory read and write cycle has been measured.

The designed ATMSWITCH consists of several circuit modules. Each module performs well-defined functions during the switch working cycle. Modules are controlled by the **controller** to accept an incoming cell from **input port** to the **cell memory** and send a cell from the cell memory to the **output port** at each switch working cycle. For a particular CBR/VBR channel, the **trigger mechanism** schedules cell output at regular intervals. Because all cells are forwarded through the network with a constant delay at each node, the end-to-end delay which they experience is constant. The **jitter** problem can thus be ignored. Consequently, the ATMSWITCH provides a solution to the problem of conflicting requirements of different sorts of data traffic.

The **chandelier** cooperates with the trigger mechanism and sends the ABR/UBR cells during no CBR/VBR cells are scheduled output, so the switch works efficiently.

All circuit modules designed in ATMSWITCH were constructed by VHDL definition of the circuit. And the simulated result of ATMSWITCH timing is shown clearly in the circuit signal waveform. Therefore, the number of clock cycles required to perform the buffer maintenance activities has been determined, and clock period required to fit this number of clock cycles into the 12ns write and read cycle is 1.2ns time.

A multi-buffer switch architecture has been introduced. The most modules of ATMSWITCH switch have been inherited to the multi-buffer switch. The multi-buffer switch assigns each output port a buffer. During a switch working cycle, as many as N (

number of input and output ports) cells can be transferred from the input ports to the output port. Therefore, the multi-buffer switch architecture can improve the total throughput of switch. Although the main structure of multi-buffer switch is proposed, but the functions of each module, and details of circuit design need to be completed.

References

Aanen, E. et al., "Cell loss performance of the GAUSS ATM switch", Proc. IEEE INFOCOM'92, Florence, Italy, May 1992, pp. 717-726.

Ahmadi, H., and Denzel, W. E. "A survey of modern high-performance switching techniques", IEEE J. Selected Areas Commun., Vol. 7, No, Sep. 1989, pp. 1091-1103.

Awdeh, R. Y. and Mouftah, H. T. "Broadband packet switch architectures", Proc. Photonics'93: 3rd IEEE Int. Workshop on Photonic Networks, Components, and Applications, Atlanta, GA, Sep. 1993, pp. 183-188.

Awdeh, R. Y., Mouftah, H. T., "Survey of ATM switch, architectures", Computer Networks and ISDN Systems, Vol. 27, 1995, pp. 1567-1613.

Bae, J. J., Suda, T. "Survey of traffic control schemes and protocols in ATM networks", Proc. IEEE, Vol. 79, No.2, Feb. 1991, pp. 170-189.

Bennett, J. C. R., Zhang, H. "Hierarchical Packet Fair Queuing Algorithm", Proc, ACM SIGCOMM'96, Sept. 1996, pp. 143-156.

Boettinger, H., "The Telephone Book", New York, Stearn Publishers Ltd., 1983.

Chen, T. M., Liu, S. S. "ATM Switching Systems", Artech House, Boston, ISDN 0890066825, 1995.

Chen, X. "A survey of multistage interconnection networks for fast packet switches", Int. J. Digital and Analog Commun. Systems, Vol. 4, pp. 33-59, 1991.

Cheng, T. H. "A multichannel ATM switch with output buffering", Computer Networks and ISDN Systems, Vol. 29, Nov. 1997, pp. 195-208.

Clark, M. P. "ATM Networks Principles and Use", Wiley & Teubner, ISBN 0 471-96701-7, 1996

Descloux, A. "Stochastic models for ATM switching networks", IEEE L. Selected Areas Commun., Vol. 9, No. 3, Apr. 1991, pp. 450-457.

Eckberg, A., Hou, T. C., "Effects of output buffer sharing on buffer requirements in an ATDM packet switch", Proc. IEEE INFOCOM'88, New Orleans, LA, Mar. 1988, pp. 459-566.

Eng, K. Y., Hluchyi, M. J., Yeh, Y. S. "Multicast and Broadcast Services in a Knockout Packet Switch," Proc. IEEE INFOCOM, 1988, pp. 29 - 34.

Fenwick, P. "Queue Prediction: an efficient scheduler for fast ATM cell transmission", Proc. ATMWORKS' 97 (NZ), Feb. 1997, pp. 83-97.

Friesen, V. J., Wong, J. W. " The effect of multiplexing, switching and other factors on the performance of broadband networks", Proc. IEEE Network, Vol. 7, No. 1, Jan. 1993, pp. 12-26.

Georgiadis, L. et al., "Efficient Support of Delay and Rate Guarantees in an Internet", Proc. ACM SIGCOMM '96, Oct. 1996, pp. 106-116.

Griffiths, J. M. (ed.), ISDN Explained: Worldwide Network and Applications Technology, John Wiley & Sons Ltd., 1992.

Hluchyj, M. G. and Karol, M. J., "Queueing in High-Performance Packet Switching," IEEE J. Select. Areas Commun, vol. 6, Dec. 1988, pp. 1587-1597.

Hui, J. Y., Arthurs, E. "A broadband packet switch for integrated transport", IEEE J. Selected Areas Commun., Vol. 5, No. 8, Oct. 1987, pp. 264-273.

Jacob, A. R., "A survey of fast packet switches", Computer Commun. Review, Vol.20, No. 1, Jan. 1990, pp. 54-64.

Kumar, S. "On Multicast Support for Shared-Memory Based ATM Switch Architecture", IEEE Network, Vol. 10. No.1, Jan./Feb., 1996

Kumar, S., Agrawal, D. P. "A Shared-Buffer Direct-Access (SBDA) ATM Switch Architecture for Broadband Networks," Proc. IEEE Int'l. Conf. Commun., 1994, pp. 101 - 105.

Lee, T. T. "Nonblocking Copy Networks for Multicast Packet Switching," IEEE JSAC, vol. 6, 1988, pp. 1455-1467.

Listanti, M. and Roveri, A. "Switching structures for ATM", Computer Commun., Vol. 12, No. 6, Dec. 1989, pp. 349-358.

Lyons, P. J., McGregor, A. J. and Moretti, G. S., "The Associative Chandelier -- Fair, Efficient, Prioritised Buffering in ATM Switches", Proc. of the First New Zealand ATM and Broadband Workshop, February, 1996, pp. 155 - 167.

Lyons, P. J. "Output Triggering Adds Classes of Service to the Associative Chandelier," ATMWorks '97, February 1997, pp. 99-116.

Lyons, P. J., McGregor, A. J., MasseyNet: "A University-oriented Local Area Network", MICROS PLUS: Education Peripherals; Proc. of the IFIP Working Conference on the Educational Implications of Connecting Tools and Devices to Micro-computers, August 1986, pp. 155-167.

Mayo, J., "The Evolution Toward Universal Information Services," Telephony, Vol. 208, March 4, 1985, pp. 40-50.

Newman, P. "ATM technology for corporate networks", IEEE Commun. Mag., Apr. 1992, pp 90-101.

Oie, Y., Suda, T., Murata, M. and Hiyashara, H. "Survey of switching techniques in high-speed networks and their performance", Int. J. Satellite Commun., Vol. 9, 1991, pp. 285-303.

Oie, Y. et al., "Effect of Speedup in Nonblocking Packet Switch," ICC'89 Conf. Rec., Boston, MA, June 1989, pp. 410-414.

Pattavina, A. "Nonblocking architectures for ATM switching", IEEE Commun. Mag., No.2, Feb. 1993, pp. 38-48.

Pfister, G. F., Norton, V. A. 'Hot spot' contention and combining in multistage interconnection Networks, IEEE Trans. Computers, Vol. 34, No. 10, Oct. 1985, pp. 943-948.

Pitts, J. M. "Introduction to ATM Design and Performance", John Wiley & Sons, ISBN 0-471-96340-2, 1996.

Prycher, M. D. "Asynchronous Transfer Mode: Solution for Broadband ISDN", Second Edition (Chichester, England: Ellis Horwood, 1993).

Prycker, M. D., Peschi, R., Landegem, T. V. "B-ISDN and the OSI Protocol Reference Model", IEEE Network, March, 1993, pp. 10-18.

Schatt, S. "Understanding ATM," McGraw-Hill, ISBN 0-07-057679-3, 1996.

Stephens, W. E., Deprycker, M., Tobagi, F. A. and Yamaguchi, T. "Guest editorial: large-scale ATM switching systems for B-ISDN", Ieee J. Selected Areas Commun, Vol. 9, No. 8, Oct. 1991, pp. 1157-1158.

Stiliadis, D., Varma, A. "Latency-Rate Service: A General Model for Analysis of Traffic Scheduling Algorithms", Proc. IEEE INFOCOM'96, Apr. 1996, pp. 111-119.

Suzuki, H. et al., "Output-buffer switch architecture for asynchronous transfer mode" in Proc. Int. Conf. On Communications, Boston, MA, June 1989, pp. 4.4.1—4.1.5.

Tobagi, F. A. "Fast Packet Switch Architectures for Broadband Integrated Services Digital Networks," Proc. of the IEEE, vol. 78, No. 1, Jam. 1990, pp.133-167.

Varma, A. "Hardware Implementation of fair Queuing Algorithms for Asynchronous Transfer Mode Networks", IEEE Communication Magazine, Dec. 1997, pp. 54-68.

Yoon, H., Liu, M. T., Lee, K. Y. "The Knockout switch under nonuniform traffic", Proc. IEEE GLOBECOM'88, Hollywood, FL, Nov. 1988, pp. 1628-1634.

Zhang, H. "Service Disciplines for guaranteed Performance Service in Packet-Switching Networks", Proc. IEEE, vol. 83, Oct. 1995, pp. 1374-1396.

CCITT Recommendation I.121. "Broadband Aspects of ISDN," blue Book, Fascicle III.7, Geneva 1989.

Appendix

VHDL Program Listing

```
--------------------------------------------------------------------------------
-- Copyright 1997, Massey University, All Rights Reserved
--   File name :  atmpack.vhd
--   Title     :  [ATM Package]
--   Module    :  [The ATM Package]
--   Purpose   :  [Declare the global types and procedures for the ATM switch
--                 modules]
--------------------------------------------------------------------------------
--   Modification History :
--         Date              Author          Revision        Comments
--   Tur July  10  1997      Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;

PACKAGE atm_types is

  CONSTANT unit_delay   : Time    := 1 ns;
  CONSTANT MapSize      : integer := 1023;
  CONSTANT MapWidth     : integer := 10;
  CONSTANT CellMemSize  : integer := 1023;
  CONSTANT CellMemWidth : integer := 10;
  CONSTANT BufferSize   : integer := 15;


  ------------------------------------------------------
  --Declare type that used for ATM stitch
  ------------------------------------------------------

  subtype  bit_15 is std_logic_vector(14 downto 0);
  type     bit_15_array is array(integer range<>) of bit_15;
  function resolve_bit_15(driver : in bit_15_array) return bit_15;
  subtype  bus_bit_15 is resolve_bit_15 bit_15;

  subtype  bit_8 is std_logic_vector(7 downto 0);

  type     boolean is(false, true);


  ------------------------------------------------------
  -- Input Port Declaration
  ------------------------------------------------------

  SUBTYPE  cell_head_type is std_ulogic_vector(27 DOWNTO 0);
  TYPE     cell_head_type_array is array (integer range <>) of cell_head_type;
  FUNCTION resolve_cell_head_type(driver : in cell_head_type_array)
           RETURN cell_head_type;
  SUBTYPE  bus_cell_head_type is resolve_cell_head_type cell_head_type;

  SUBTYPE  oe_type is std_ulogic_vector(3 DOWNTO 0);
  TYPE     buf_full_type is array(0 to BufferSize) of std_ulogic;
  SUBTYPE  portenable_type is std_ulogic_vector(3 DOWNTO 0);


  ------------------------------------------------------
  -- Chandelier Declaration
  ------------------------------------------------------
```



```
  TYPE     chandelier_element_rec IS
             RECORD
               size      : integer;
               lastentry : integer range 0 to MapSize;
             END RECORD;
  TYPE     chandelier_type is array(0 to BufferSize) of chandelier_element_rec;
  TYPE     chandelier_size is array(0 to BufferSize) of std_ulogic;

  subtype  map_entry is integer range 0 to MapSize;
  subtype  port_type is integer range 0 to BufferSize;

  TYPE     chandelier_head_rec;
  TYPE     chandelier_head_ptr IS ACCESS chandelier_head_rec;
  TYPE     chandelier_head_rec IS
             RECORD
               entry    : integer range 0 to MapSize;
               nexthead : chandelier_head_ptr;
             END RECORD ;

  TYPE     chandelier_cycle IS array(Natural Range <>) of chandelier_head_rec;
  TYPE     chandelier_blk_ptr IS array(0 to 15) of chandelier_head_ptr;


  ------------------------------------------------------------------
  -- Memory Block Declaration
  ------------------------------------------------------------------
  TYPE     mem_element_rec;
  TYPE     mem_element_ptr IS ACCESS mem_element_rec;
  TYPE     mem_element_rec IS
             RECORD
               addr: bit_vector(15 DOWNTO 0);
               data: bit_vector(7 DOWNTO 0);
               nxt : mem_element_ptr;
             END RECORD ;

  TYPE     mem_blks IS array(Natural Range <>) of mem_element_ptr;

  TYPE     init_file_rec IS
             RECORD
               addr: STRING(1 TO 6);
               data: INTEGER;
             END RECORD;

  ------------------------------------------------------------------
  -- Declare global procedures for the ATM switch modules
  ------------------------------------------------------------------

  PROCEDURE read_byte    (VARIABLE   blk_addr     : IN  integer;
                          VARIABLE   link_addr     : IN  bit_vector(15 DOWNTO 0);
                          VARIABLE   link_data     : OUT bit_vector(7 DOWNTO 0);
                          VARIABLE   memory_block  : IN  mem_blks);

  PROCEDURE program_byte (VARIABLE   blk_addr     : IN integer;
                          VARIABLE   link_addr     : IN bit_vector(15 DOWNTO 0);
                          VARIABLE   link_data     : IN bit_vector(7 DOWNTO 0);
                          VARIABLE   memory_block  : INOUT mem_blks);

  PROCEDURE erase_block  (VARIABLE   blk_addr     : IN integer;
                          VARIABLE   memory_block  : INOUT mem_blks);
```

```
  PROCEDURE check_addr   (VARIABLE   mem_addr     : IN  std_ulogic_vector
                                                       (19 DOWNTO 0);
                          VARIABLE   addr_valid   : OUT BOOLEAN);

  PROCEDURE check_data   (VARIABLE   mem_data     : IN  std_ulogic_vector
                                                       (7 DOWNTO 0);
                          VARIABLE   data_valid   : OUT BOOLEAN);

  PROCEDURE Int_to_Bits  (VARIABLE   int          : IN  integer;
                          VARIABLE   bits         : OUT std_logic_vector);

  ------------------------------------------------------------------
  -- Declare global functions for the ATM switch modules
  ------------------------------------------------------------------

  FUNCTION Int_to_BitVector    (CONSTANT    int_data  : IN integer;
                                CONSTANT    size      : IN natural)
                                RETURN bit_vector;

  FUNCTION BitVector_to_Int    (CONSTANT    bv_data   : IN bit_vector)
                                RETURN integer;

  FUNCTION Bits_to_Int         (CONSTANT    bits      : IN std_logic_vector)
                                RETURN integer;
  FUNCTION Bits_to_Natural     (CONSTANT    bits      : IN std_logic_vector)
                                RETURN natural;
END atm_types;



--------------------------------------------------------------------------------
--   File name :  atmbody.vhd
--   Title     :  [ATM Package Body]
--   Module    :  [The ATM Package Body]
--   Purpose   :  [Implement the functions and procedures of the ATM switch
--                 package]
--------------------------------------------------------------------------------
--   Modification History :
--          Date              Author          Revision        Comments
--   Tur July  10  1997       Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

PACKAGE BODY atm_types IS

  ------------------------------------------------------
  -- Function resolve_cell_head_type :
  ------------------------------------------------------
  FUNCTION resolve_cell_head_type(driver : in cell_head_type_array)
           RETURN cell_head_type IS
  CONSTANT head   : cell_head_type := "00000000000000000000000000000000";
  VARIABLE result : cell_head_type := head;
  BEGIN
    for i in driver'range loop
        result := result or driver(i);
    end loop;
    return result;
  END resolve_cell_head_type;
```



```
------------------------------------------------------------------------------
--  Procedure read_byte: A search is made through the linked list to find the
--  link that contains the data for the given address. If found, then that
--  data is sent out. If not then it is assumed that the location is erased
--  with data = "11111111".
------------------------------------------------------------------------------
PROCEDURE read_byte (VARIABLE      blk_addr      : IN  integer;
                     VARIABLE      link_addr     : IN  bit_vector
                                                       (15 DOWNTO 0);
                     VARIABLE      link_data     : OUT bit_vector(7 DOWNTO 0);
                     VARIABLE      memory_block  : IN  mem_blks)
IS
VARIABLE temp_mem_ptr : mem_element_ptr := memory_block(blk_addr);
BEGIN

 IF (memory_block(blk_addr) = NULL) THEN
     link_data := "11111111";
 ELSE
     read_loop : WHILE (temp_mem_ptr /= NULL) LOOP
                       IF (temp_mem_ptr.addr = link_addr) THEN
                             link_data := temp_mem_ptr.data;
                       EXIT   read_loop;

                       ELSIF (temp_mem_ptr.nxt /= NULL) THEN
                             temp_mem_ptr := temp_mem_ptr.nxt;
                       ELSE
                             link_data := "11111111";
                             EXIT read_loop;
                       END IF;
     END LOOP read_loop;
 END IF;

END read_byte;


------------------------------------------------------------------------------
--  Procedure program_byte: If the data being programmed is "11111111" then
--  this procedure does nothing. Else,it reads the array to check if that
--  location has previously been written to. If the existing data read back
--  is "11111111" then a new link is created with the current data provided.
--  Else,an AND operation between the existing data and the current data is
--  done and the value stored in the location.
------------------------------------------------------------------------------
PROCEDURE program_byte (VARIABLE blk_addr      : IN    integer;
                        VARIABLE link_addr     : IN    bit_vector(15 DOWNTO 0);
                        VARIABLE link_data     : IN    bit_vector(7 DOWNTO 0);
                        VARIABLE memory_block  : INOUT mem_blks)
IS
VARIABLE temp_mem_ptr : mem_element_ptr := memory_block(blk_addr);
VARIABLE exist_data   : bit_vector(7 DOWNTO 0);

BEGIN
  IF (link_data /= "11111111") THEN
      read_byte(blk_addr,link_addr,exist_data,memory_block);
      IF (exist_data = "11111111") THEN
          temp_mem_ptr           := NEW mem_element_rec;
          temp_mem_ptr.addr      := link_addr;
          temp_mem_ptr.data      := link_data;
          temp_mem_ptr.nxt       := memory_block(blk_addr);
          memory_block(blk_addr) := temp_mem_ptr;
```

```
      ELSE
          program_loop : WHILE (temp_mem_ptr /= NULL) LOOP
                         IF (temp_mem_ptr.addr = link_addr) THEN
                             temp_mem_ptr.data := temp_mem_ptr.data
                             AND link_data;
                             EXIT program_loop;
                         END IF;
                         temp_mem_ptr := temp_mem_ptr.nxt;
          END LOOP program_loop;
      END IF;
  END IF;

END program_byte;


-------------------------------------------------------------------------------
--  Procedure erase_block:The entire linked list associated with the given
--  block address is deallocated one at a time starting from the head of the
--  list.
-------------------------------------------------------------------------------
PROCEDURE erase_block (VARIABLE blk_addr      : IN    integer;
                       VARIABLE memory_block  : INOUT mem_blks)
IS
VARIABLE temp_mem_ptr : mem_element_ptr := memory_block(blk_addr);
BEGIN
  erase_loop : WHILE (memory_block(blk_addr) /= NULL) LOOP
                      temp_mem_ptr           := memory_block(blk_addr);
                      memory_block(blk_addr) := temp_mem_ptr.nxt;
                      DEALLOCATE(temp_mem_ptr);
  END LOOP erase_loop;
END erase_block;


-------------------------------------------------------------------------------
--  Procedure check_addr:The address is checked to see if it is invalid. The
--  boolean addr_valid returns the result. If each of the address bits is
--  anything but a 0 or 1 then addr_valid is set to FALSE.
-------------------------------------------------------------------------------
PROCEDURE check_addr (VARIABLE  mem_addr  : IN std_ulogic_vector(19 DOWNTO 0);
                      VARIABLE  addr_valid: OUT BOOLEAN)
IS
BEGIN
      check_loop : FOR i IN mem_addr'LEFT DOWNTO mem_addr'RIGHT LOOP

                IF (mem_addr(i) /= '0' AND mem_addr(i) /= '1') THEN
                    addr_valid := FALSE;
                    RETURN;
                END IF;
      END LOOP check_loop;
      addr_valid := TRUE;
      RETURN;

END check_addr;


-------------------------------------------------------------------------------
--  Procedure check_data: The data is checked to see if it is valid. If it is
--  anything but a 0 or 1 then the boolean data_valid is set to FALSE.
-------------------------------------------------------------------------------
```



```
PROCEDURE check_data (VARIABLE  mem_data  : IN std_ulogic_vector(7 DOWNTO 0);
                      VARIABLE  data_valid : OUT BOOLEAN)
IS
BEGIN
      check_loop : FOR i IN mem_data'LEFT DOWNTO mem_data'RIGHT LOOP

                IF (mem_data(i) /= '0' AND mem_data(i) /= '1') THEN
                    data_valid := FALSE;
                    RETURN;
                END IF;
      END LOOP check_loop;
      data_valid := TRUE;
      RETURN;

END check_data;


-------------------------------------------------------------------------------
--  Function Int_to_BitVector: Convert an integer to bit_vector.
-------------------------------------------------------------------------------
CONSTANT        MaxIntBitLength          : integer := 32;
FUNCTION Int_to_BitVector (CONSTANT   int_data   : IN integer;
                           CONSTANT   size       : IN natural)
                           RETURN bit_vector
IS
VARIABLE      bit_value : bit_vector(size - 1 DOWNTO 0);
VARIABLE      temp      : integer;
VARIABLE      temp_data : integer := int_data;
VARIABLE      neg_flag  : BOOLEAN := FALSE;
VARIABLE      rems      : bit_vector((MaxIntBitLength - 1) DOWNTO 0):=
                          (OTHERS=>'0');
BEGIN
  IF (size = 0) THEN
       assert FALSE
         report "Int_to_BitVector: vector size specified as zero."
         severity error;
  END IF;


  FOR i IN 0 TO (MaxIntBitLength - 1) LOOP
       EXIT WHEN temp_data <= 0;
       temp := temp_data/2;
       rems(i) := bit'VAL(temp_data - (temp*2));
       temp_data := temp;
  END LOOP;

  assert (temp_data = 0)
       report "Int_to_BitVector: integer too large."
       severity error;

  bit_value := rems(size - 1 DOWNTO 0);
  RETURN (bit_value);

END Int_to_BitVector;


-------------------------------------------------------------------------------
--  Function BitVector_to_Int: Convert a bit_vector to integer.
--
-------------------------------------------------------------------------------
```

```
FUNCTION BitVector_to_Int (CONSTANT bv_data : IN bit_vector)
                                        RETURN integer
IS
VARIABLE    size      : integer := bv_data'LENGTH;
VARIABLE    data_shift: bit_vector(bv_data'LENGTH - 1 DOWNTO 0) := bv_data;
VARIABLE    int_value : integer := 0;
VARIABLE    success   : BOOLEAN := false;
BEGIN
  IF (size = 0) THEN
       assert false
         report "BitVector_to_Int: input data vector has size zero."
         severity error;
  END IF;

  convert_loop : LOOP
  IF (size >= MaxIntBitLength) THEN
       assert false
         report "BitVector_to_Int: bitvector is greater than 31 bits."
         severity error;
       EXIT convert_loop;
  END IF;

  int_value := bit'POS(data_shift(size - 1));

  FOR i IN (size - 2) DOWNTO 0 LOOP
       int_value := (int_value * 2) + bit'POS(data_shift(i));
  END LOOP;

  success := TRUE;
  EXIT convert_loop;
  END LOOP convert_loop;

  IF (success=TRUE) THEN
       RETURN (int_value);
  ELSE
       RETURN (0);
  END IF;

END BitVector_to_Int;



FUNCTION resolve_bit_15(driver : in bit_15_array) return bit_15 is
  constant float_value : bit_15 :="000000000000000";
  variable result : bit_15 :=float_value;

begin
  for i in driver'range loop
      result :=result or driver(i);
  end loop;
  return result;
end resolve_bit_15;


-------------------------------------------------------
--Function that convert std_logic into integer
-------------------------------------------------------
function bits_to_int(bits : in std_logic_vector) return integer is
  variable temp : std_logic_vector(bits'range);
  variable result : integer := 0;
```



```
begin
  if bits(bits'left) = '1' then
       temp := not bits;
  else
       temp := bits;
  end if;
  for index in bits'range loop
      result := result*2 + std_logic'pos(temp(index));
  end loop;
  if bits(bits'left) = '1' then
       result := (-result) -1;
  end if;
  return result;
end bits_to_int;


-------------------------------------------------------
--Function that convert std_logic into natural type
-------------------------------------------------------
function bits_to_natural(bits : in std_logic_vector) return natural is
  variable result : natural :=0;
begin
  for index in bits'range loop
      result :=result*2 + std_logic'pos(bits(index));
  end loop;
  return result;
end bits_to_natural;


-------------------------------------------------------
--Procedure that convert integer into std_logic type
-------------------------------------------------------
procedure int_to_bits(int : in integer; bits : out std_logic_vector) is
  variable temp : integer;
  variable result : std_logic_vector(bits'range);

begin
  if int < 0 then
       temp :=-(int+1);
  else
       temp :=int;
  end if;
  for index in bits'reverse_range loop
      result(index) :=std_logic'val(temp rem 2);
      temp :=temp/2;
  end loop;
  if int <0 then
       result :=not result;
       result(bits'left) :='1';
  end if;
  bits := result;
end int_to_bits;

END atm_types;



-------------------------------------------------------------------------------
--   File name :  atmtest.vhd
--   Title     :  [The Test Bench of the ATM Switch Modules ]
```

```
--   Module    :  [The Test Bench Modules]
--   Purpose   :  [Declare the global types and procedures for the ATM switch
--                 modules]
-------------------------------------------------------------------------------
--   Modification History :
--         Date                Author          Revision        Comments
--   Sat March 28  1998        Bo Yan          Rev1.0.0        Creation
-------------------------------------------------------------------------------

--
--Test bench circuit of atm controller
--

USE work.atm_types.all;
USE work.map_pkg01.all;
USE work.mem_pkg.all;
USE work.cq_link_pkg.all;

Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.numeric_std.all;

ENTITY atm_controller_test is
end atm_controller_test;

ARCHITECTURE structure of atm_controller_test is

  component clock_gen
    port ( clk1   : out std_ulogic;
           clk2   : out std_ulogic;
           reset  : out std_ulogic);
  end component;

  component regular_counter
    port ( clk1   : in  std_ulogic;
           clk2   : in  std_ulogic;
           reset  : in  std_ulogic;
           count  : out std_ulogic_vector(9 DOWNTO 0));
  end component;

  component atm_controller
    port ( read   : out std_ulogic;
           write  : out std_ulogic;
           clk1   : in std_ulogic;
           clk2   : in std_ulogic;
           reset  : in std_ulogic;
           acceptcell : in  std_ulogic;
           outgoing   : in  std_ulogic;
           resetfull  : out std_ulogic);
    end component;

  component channel_handler
  port (clk1        : in    std_ulogic;
        clk2        : in    std_ulogic;
        read        : out   std_ulogic;
        write       : out   std_ulogic;
        reset       : in    std_ulogic;
        match       : in    std_ulogic;
        resetfull   : out   std_ulogic;
        acceptcell  : in    std_ulogic;
```



```
        outgoing    : in    std_ulogic;
        setupcell   : out   std_ulogic;
        releasecell : out   std_ulogic;
        itv         : out   std_ulogic_vector(9 DOWNTO 0);
        mask        : out   std_ulogic_vector(9 DOWNTO 0);
        en_port_no  : in    oe_type;
        head_bus    : inout bus_cell_head_type bus);
  end component;


  component MapMemory
  port( read        : in    std_ulogic;
        write       : in    std_ulogic;
        clk1        : in    std_ulogic;
        match       : out   std_ulogic;
        reset       : in    std_ulogic;
        ready       : in    std_ulogic;
        a_bus       : inout bus_cell_mem_addr bus;
        head_bus    : inout bus_cell_head_type bus;
        outgoing    : in    std_ulogic;
        count       : in    std_ulogic_vector(9 DOWNTO 0);
        itv         : in    std_ulogic_vector(9 DOWNTO 0);
        mask        : in    std_ulogic_vector(9 DOWNTO 0);
        setupcell   : in    std_ulogic;
        releasecell: in     std_ulogic;
        channeltype: out    std_ulogic;
        cq_map_bus : inout bus_cq_map_bus bus;
        en_port_no      : in  oe_type;
        en_outport_no   : in  oe_type;
        chandeliersize : out chandelier_size);
  end component;

  component cellmem
    port ( clk1         : in     std_ulogic;
           d_bus        : inout  bus_cell_mem_data bus;
           a_bus        : in     bus_cell_mem_addr bus;
           read         : in     std_ulogic;
           write        : in     std_ulogic;
           reset        : in     std_ulogic;
           ready        : out    std_ulogic);
  end component;

  component inputserver
  port ( clk1         : in  std_ulogic;
         clk2         : in  std_ulogic;
         read         : in  std_ulogic;
         write        : in  std_ulogic;
         reset        : in  std_ulogic;
         acceptcell   : out std_ulogic;
         en_port_no   : out oe_type;
         full00       : in  std_ulogic;
         full01       : in  std_ulogic;
         full02       : in  std_ulogic;
         full03       : in  std_ulogic;
         full04       : in  std_ulogic;
         full05       : in  std_ulogic;
         full06       : in  std_ulogic;
         full07       : in  std_ulogic;
         full08       : in  std_ulogic;
         full09       : in  std_ulogic;
```

```
        full10      : in  std_ulogic;
        full11      : in  std_ulogic;
        full12      : in  std_ulogic;
        full13      : in  std_ulogic;
        full14      : in  std_ulogic;
        full15      : in  std_ulogic);
end component;

component input_buffer01
  port ( d_bus      : out bus_cell_mem_data bus;
         reset      : in  std_ulogic;
         oe         : in  oe_type;
         ready      : in  std_ulogic;
         full01     : out std_ulogic;
         head_bus   : out bus_cell_head_type bus;
         resetfull  : in  std_ulogic;
         setupcell  : in  std_ulogic);
end component;

component input_buffer00
  port ( d_bus      : out bus_cell_mem_data bus;
         reset      : in  std_ulogic;
         oe         : in  oe_type;
         ready      : in  std_ulogic;
         full00     : out std_ulogic;
         head_bus   : out bus_cell_head_type bus;
         resetfull  : in  std_ulogic;
         setupcell  : in  std_ulogic);
end component;


component input_buffer02
  port ( d_bus      : out bus_cell_mem_data bus;
         reset      : in  std_ulogic;
         oe         : in  oe_type;
         ready      : in  std_ulogic;
         full02     : out std_ulogic;
         head_bus   : out bus_cell_head_type bus;
         resetfull  : in  std_ulogic;
         setupcell  : in  std_ulogic);
end component;

component input_buffer03
  port ( d_bus      : out bus_cell_mem_data bus;
         reset      : in  std_ulogic;
         oe         : in  oe_type;
         ready      : in  std_ulogic;
         full03     : out std_ulogic;
         head_bus   : out bus_cell_head_type bus;
         resetfull  : in  std_ulogic;
         setupcell  : in  std_ulogic);
end component;


component output_buffer01
port    ( oe         : in  oe_type;
         reset      : in  std_ulogic;
         ready      : in  std_ulogic;
```



```
         d_bus      : in  bus_cell_mem_data bus;
         head_bus   : in  cell_head_type;
         freeport01 : out std_ulogic );
end component;

component output_buffer00
port    ( oe         : in  oe_type;
         reset      : in  std_ulogic;
         ready      : in  std_ulogic;
         d_bus      : in  bus_cell_mem_data bus;
         head_bus   : in  cell_head_type;
         freeport00 : out std_ulogic );
end component;


component output_buffer02
port    ( oe         : in  oe_type;
         reset      : in  std_ulogic;
         ready      : in  std_ulogic;
         d_bus      : in  bus_cell_mem_data bus;
         head_bus   : in  cell_head_type;
         freeport02 : out std_ulogic );
end component;



component output_buffer03
port    ( oe         : in  oe_type;
         reset      : in  std_ulogic;
         ready      : in  std_ulogic;
         d_bus      : in  bus_cell_mem_data bus;
         head_bus   : in  cell_head_type;
         freeport03 : out std_ulogic );
end component;


component output_encoder
port ( clk1         : in  std_ulogic;
       clk2         : in  std_ulogic;
       read         : in  std_ulogic;
       write        : in  std_ulogic;
       reset        : in  std_ulogic;
       match        : in  std_ulogic;
       outgoing     : out std_ulogic;
       chandeliersize : in  chandelier_size;
       en_outport_no  : out oe_type;
       freeport00 : in  std_ulogic;
       freeport01 : in  std_ulogic;
       freeport02 : in  std_ulogic;
       freeport03 : in  std_ulogic;
       freeport04 : in  std_ulogic;
       freeport05 : in  std_ulogic;
       freeport06 : in  std_ulogic;
       freeport07 : in  std_ulogic;
       freeport08 : in  std_ulogic;
       freeport09 : in  std_ulogic;
       freeport10 : in  std_ulogic;
       freeport11 : in  std_ulogic;
       freeport12 : in  std_ulogic;
       freeport13 : in  std_ulogic;
```

```
                    reset => reset, resetfull => resetfull,
                    full01 => full01, oe => en_port_no,
                    setupcell => setupcell, ready => ready);

inbuff00      : input_buffer00
                port map (d_bus  => d_bus, head_bus => head_bus,
                    reset => reset, resetfull => resetfull,
                    full00 => full00, oe => en_port_no,
                    setupcell => setupcell, ready => ready);

inbuff02      : input_buffer02
                port map (d_bus  => d_bus, head_bus => head_bus,
                    reset => reset, resetfull => resetfull,
                    full02 => full02, oe => en_port_no,
                    setupcell => setupcell, ready => ready);

inbuff03      : input_buffer03
                port map (d_bus  => d_bus, head_bus => head_bus,
                    reset => reset, resetfull => resetfull,
                    full03 => full03, oe => en_port_no,
                    setupcell => setupcell, ready => ready);

outbuff01     : output_buffer01
                port map (oe => en_outport_no, reset => reset, ready => ready,
                    d_bus => d_bus,
                    head_bus => head_bus, freeport01 => freeport01);

outbuff00     : output_buffer00
                port map (oe => en_outport_no, reset => reset, ready => ready,
                    d_bus => d_bus,
                    head_bus => head_bus, freeport00 => freeport00);

outbuff02     : output_buffer02
                port map (oe => en_outport_no, reset => reset, ready => ready,
                    d_bus => d_bus,
                    head_bus => head_bus, freeport02 => freeport02);

outbuff03     : output_buffer03
                port map (oe => en_outport_no, reset => reset, ready => ready,
                    d_bus => d_bus,
                    head_bus => head_bus, freeport03 => freeport03);

output_server : output_encoder
                port map (clk1 => clk1, clk2 => clk2, read => read,
                    write => write, reset => reset,
                    outgoing  => outgoing, match => match,
                    chandeliersize => chandeliersize,
                    en_outport_no => en_outport_no,
                    freeport00 => freeport00, freeport01 => freeport01,
                    freeport02 => freeport02, freeport08 => freeport08,
                    freeport03 => freeport03, freeport04 => freeport04,
                    freeport05 => freeport05, freeport11 => freeport11,
                    freeport06 => freeport06, freeport07 => freeport07,
                    freeport09 => freeport09, freeport10 => freeport10,
                    freeport12 => freeport12, freeport13 => freeport13,
                    freeport15 => freeport15, freeport14 => freeport14);

cqlink        : cq_link
                port map (a_bus => a_bus, read => read, ready => ready,
                    write => write, reset => reset, clk1 => clk1,
```



```
                    cq_map_bus => cq_map_bus, channeltype=>channeltype);
END structure;


CONFIGURATION atm_controller_behaviour_test of atm_controller_test is
  for structure

      for clock         : clock_gen
          use entity work.clock_gen(behaviour)
              generic map(Tpw => 12 ns, Tps => 12 ns);
      end for;
      for counter       : regular_counter
          use entity work.regular_counter(behaviour);
      end for;
      for map_memory    : MapMemory
          use entity work.MapMemory(behaviour);
      end for;
      for cell_memory   : cellmem
          use entity work.cellmem(behaviour);
      end for;
      for controller    : channel_handler
          use entity work.channel_handler(behaviour);
      end for;
      for input_server  : inputserver
          use entity work.inputserver(behaviour);
      end for;
      for inbuff01      : input_buffer01
          use entity work.input_buffer01(behaviour);
      end for;

      for inbuff00      : input_buffer00
          use entity work.input_buffer00(behaviour);
      end for;


      for inbuff02      : input_buffer02
          use entity work.input_buffer02(behaviour);
      end for;


      for inbuff03      : input_buffer03
          use entity work.input_buffer03(behaviour);
      end for;


      for outbuff01     : output_buffer01
          use entity work.output_buffer01(behaviour);
      end for;

      for outbuff00     : output_buffer00
          use entity work.output_buffer00(behaviour);
      end for;


      for outbuff02     : output_buffer02
          use entity work.output_buffer02(behaviour);
      end for;
```

```
      for outbuff03     : output_buffer03
          use entity work.output_buffer03(behaviour);
      end for;


      for output_server : output_encoder
          use entity work.output_encoder(behaviour);
      end for;
      for cqlink        : cq_link
          use entity work.cq_link(behaviour);
      end for;

  end for;

END atm_controller_behaviour_test;



--------------------------------------------------------------------------------
--   File name :  mempkg.vhd
--   Title     :  [ATM Cell Memory: 1024 x 8-bits Cache Static Random Access
--                 Memory]
--   Module    :  [Package Declaration]
--   Purpose   :  [Declare the types and procedures for the Cell Memory module]
--------------------------------------------------------------------------------
--   Modification History :
--         Date            Author          Revision        Comments
--   Tur August 21  1997   Bo Yan          Rev1.0.0        Creation
--   Mon Jan    12  1998   Bo Yan          Rev1.0.0        Modified
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;
USE std.textio.all;

USE work.atm_types.all;

PACKAGE mem_pkg IS

  TYPE     cell_mem_blks is array(0 to CellMemSize) of std_ulogic_vector
           (7 DOWNTO 0);

  SUBTYPE  cell_mem_data is std_ulogic_vector(7 DOWNTO 0);
  TYPE     cell_mem_data_array is array(integer range <>) of cell_mem_data;
  FUNCTION resolve_cell_mem_data(driver : in cell_mem_data_array)
           RETURN cell_mem_data;
  SUBTYPE  bus_cell_mem_data is resolve_cell_mem_data cell_mem_data;

  SUBTYPE  cell_mem_addr is std_ulogic_vector(9 DOWNTO 0);
  TYPE     cell_mem_addr_array is array(integer range <>) of cell_mem_addr;
  FUNCTION resolve_cell_mem_addr(driver : in cell_mem_addr_array)
           RETURN cell_mem_addr;
  SUBTYPE  bus_cell_mem_addr is resolve_cell_mem_addr cell_mem_addr;

END mem_pkg;
```



```
--------------------------------------------------------------------------------
--   File name :  membdy.vhd
--   Title     :  [ATM Cell Memory: 1024 x 8-bits Cache Memory]
--   Module    :  [Package Body of the Cell Memory]
--   Purpose   :  [Implement the procedures which are declared in the Cell
--                 Memory Package]

PACKAGE BODY mem_pkg IS

--------------------------------------------------------------------------
-- The function resolves the cell data from the d_bus (data bus)
--------------------------------------------------------------------------

FUNCTION resolve_cell_mem_data(driver : in cell_mem_data_array)
         RETURN cell_mem_data IS
  CONSTANT data   : cell_mem_data := "00000000";
  VARIABLE result : cell_mem_data := data;
BEGIN
  for i in driver'range loop
      result := result or driver(i);
  end loop;
  return result;
END resolve_cell_mem_data;


--------------------------------------------------------------------------
-- The function resolves the cell memory address from the a_bus (address bus)
--------------------------------------------------------------------------

FUNCTION resolve_cell_mem_addr(driver : in cell_mem_addr_array)
         RETURN cell_mem_addr IS
  CONSTANT addr   : cell_mem_addr := "0000000000";
  VARIABLE result : cell_mem_addr := addr;
BEGIN
  for i in driver'range loop
      result := result or driver(i);
  end loop;
  return result;
END resolve_cell_mem_addr;

END mem_pkg;


--------------------------------------------------------------------------------
--   File name :  ement.vhd
--   Title     :  [ATM Cell Memory: 1024 x 8-bits Cache Memory ]
--   Module    :  [The Cell Memory Entity]
--   Purpose   :  [Define the entity of the Cell Memory]

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.all;

USE work.atm_types.all;
USE work.mem_pkg.all;

ENTITY cellmem is
```

```
  GENERIC ( Tpd   : Time  := unit_delay);
  PORT    ( clk1  : in    std_ulogic;
            a_bus : in    bus_cell_mem_addr bus;
            d_bus : inout bus_cell_mem_data bus;
            read  : in    std_ulogic;
            write : in    std_ulogic;
            reset : in    std_ulogic;
            ready : out   std_ulogic);
END cellmem;


--------------------------------------------------------------------------------
--   File name :  memarc.vhd
--   Title     :  [ATM Cell Memory: 1024 x 8-bits Cache Memory]
--   Module    :  [Behaviour Description of the Cell Memory]
--   Purpose   :  [Implement the functions of the cell memory]

USE work.mem_pkg.all;

ARCHITECTURE behaviour OF cellmem IS

BEGIN

PROCESS

  VARIABLE cellmem  : cell_mem_blks;
  VARIABLE celladdr : cell_mem_addr;
  VARIABLE celldata : cell_mem_data;
  VARIABLE memaddr  : integer range 0 to CellMemSize;

  ------------------------------------------------------------------
  --This procedure is used to initialise the cell memory
  ------------------------------------------------------------------
  PROCEDURE initialise(VARIABLE memory : INOUT cell_mem_blks) IS
    VARIABLE i : integer := 0;
  BEGIN
    for i in 0 to CellMemSize loop
        memory(i) := "00000000";
    end loop;
  END initialise;


  ------------------------------------------------------------------
  --This procedure is used to write a cell data into cell memory
  ------------------------------------------------------------------
  PROCEDURE add(VARIABLE memory : INOUT cell_mem_blks;
                CONSTANT data   : IN    cell_mem_data;
                CONSTANT addr   : IN    cell_mem_addr) IS
    VARIABLE addr_int : integer;
  BEGIN

     addr_int := BitVector_To_Int(To_BitVector(addr));
                                  -- convert the address into integer

     if addr_int<0 OR addr_int>CellMemSize then
                                  -- check whether the address is over scope
        assert false
        report "The given address is over scorp of Cell Memory !"
        severity error;
     end if;
```



```
     memory(addr_int) := data;       -- write the cell data into cell memory
  END add;


  ------------------------------------------------------------------
  --This procedure is used to read a cell data from cell memory
  ------------------------------------------------------------------
  PROCEDURE remove(VARIABLE memory : INOUT cell_mem_blks;
                   VARIABLE data   : OUT   cell_mem_data;
                   CONSTANT addr   : IN    cell_mem_addr) IS
    VARIABLE addr_int : integer;
  BEGIN

     addr_int := BitVector_To_Int(To_BitVector(addr));
                                  -- convert the address into integer

     if addr_int<0 OR addr_int>CellMemSize then
                   -- check whether the address is over scope of cell memory
        assert false
        report "The given address is over scorp of Cell Memory !"
        severity error;
     end if;
     data := memory(addr_int);
                                  -- read the cell data from cell memory
  END remove;


BEGIN                             -- begin the process

  --
  --check for reset active
  --
  wait until reset='0' or reset='1';
  if reset='1' then
     initialise(cellmem);
     ready <= 'Z' after Tpd;
     wait until reset='0';
  end if;


  loop
  if reset='0' then
     d_bus <= null after Tpd;
     wait until write='1' or read='1'; -- Wait for write or read signal
     if write='1' then                 -- write signal is high
        ready <= '0';                  -- reset ready signal
        celladdr := a_bus;          -- get cell memory address from address bus
        celldata := d_bus;                    -- get cell data from data bus
        add(cellmem, celldata, celladdr);     -- write cell data to cell memory
        wait until clk1='0'; -- after 9 times Tpd, the write operation  finished
        ready <= '1';                         -- set ready signal high
     end if;                                  -- end write='1'
     if read='1' then                         -- read signal is high
        ready <= '0';                         -- reset ready signal
        celladdr := a_bus;                    -- get address from address bus
        remove(cellmem, celldata, celladdr);  -- read the data from cell memory
        wait until clk1='1';
        ready <= '1';
        d_bus <= celldata;                    -- output data onto data bus
```

```
        wait for 1*Tpd;
      end if;                                   -- end read='1'
    else                                        -- reset /='0'
      exit;
    end if;                                     -- end reset='0'
    end loop;

END process;
END behaviour;
```

```
--------------------------------------------------------------------------------
--    File name :  clockent.vhd
--    Title     :  [ATM Clock Module]
--    Module    :  [ATM Clock Entity]
--    Purpose   :  [Define the entity of the Clock module]
--------------------------------------------------------------------------------
--    Modification History :
--          Date          Author          Revision        Comments
--    Tur July 10  1997     Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;

ENTITY clock_gen is
  GENERIC(Tpw    : Time :=12 ns;
          Tps    : Time :=12 ns;
          Tunit  : Time :=2  ns);
  PORT   (clk1   : out  std_ulogic;
          clk2   : out  std_ulogic;
          reset  : out  std_ulogic );
END clock_gen;
```

```
--------------------------------------------------------------------------------
--    File name :  clockarc.vhd
--    Title     :  [ATM Clock]
--    Module    :  [Behaviour Description of the Clock]
--    Purpose   :  [Implement the functions of the clock module]

ARCHITECTURE behaviour of clock_gen is
  constant clock_period : Time := Tpw+Tps;
  constant clock_unit   : Time := 3*Tunit;

BEGIN
  reset_driver:
    reset <='1', '0' after 2*clock_period;

  clock_driver1: process
  begin
    clk1 <='1', '0' after Tpw;
```



```
    wait for clock_period;
  end process clock_driver1;

  clock_driver2: process
  begin
    clk2 <='1', '0' after clock_unit;
    wait for 2*clock_unit;
  end process clock_driver2;
END behaviour;
```

```
--------------------------------------------------------------------------------
--    File name :  contrpkg.vhd
--    Title     :  [ATM Controller Module]
--    Module    :  [Package Declaration and body defination]
--    Purpose   :  [Declare the types and procedures for the Controller]
--------------------------------------------------------------------------------
--    Modification History :
--          Date          Author          Revision        Comments
--    Mon July 12  1997     Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;
USE std.textio.all;

USE work.atm_types.all;

PACKAGE controller_pkg IS

  TYPE boolean IS (false, true);
  TYPE itv_tree_node;
  TYPE itv_tree_ptr  IS ACCESS itv_tree_node;

  TYPE itv_tree_node IS
    RECORD
      channelid : std_ulogic_vector(9 DOWNTO 0);
      isnode    : boolean;
      ts        : integer;
      itv       : std_ulogic_vector(9 DOWNTO 0);
      leftptr   : itv_tree_ptr;
      rightptr  : itv_tree_ptr;
    END RECORD;

  TYPE itv_tree IS
    RECORD
      rootptr   : itv_tree_ptr;
    END RECORD;

  TYPE itv_stack IS
    RECORD
      bottom : integer;
      topptr : itv_tree_ptr;
    END RECORD;

  TYPE itv_tree_blk is array(0 to 40000) of itv_tree_ptr;
```

```
END controller_pkg;


-------------------------------------------------------------------
-- Define the package body of the controller
-------------------------------------------------------------------

PACKAGE BODY controller_pkg IS
END controller_pkg;


-------------------------------------------------------------------------------
--    File name :  contrent.vhd
--    Title     :  [ATM Controller]
--    Module    :  [The Controller Entity]
--    Purpose   :  [Define the entity of the controller]

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;
USE work.controller_pkg.all;

ENTITY channel_handler IS
  GENERIC ( Tpd         : Time  := unit_delay);
  PORT    ( clk1        : in    std_ulogic;
            clk2        : in    std_ulogic;
            read        : out   std_ulogic;
            write       : out   std_ulogic;
            reset       : in    std_ulogic;
            resetfull   : out   std_ulogic;
            acceptcell  : in    std_ulogic;
            outgoing    : in    std_ulogic;
            match       : in    std_ulogic;
            setupcell   : out   std_ulogic;
            releasecell: out    std_ulogic;
            itv         : out   std_ulogic_vector(9 DOWNTO 0);
            mask        : out   std_ulogic_vector(9 DOWNTO 0);
            en_port_no  : in    oe_type;
            head_bus    : inout bus_cell_head_type bus);
END channel_handler;


-------------------------------------------------------------------------------
--    File name :  contrarc.vhd
--    Title     :  [ATM Controller]
--    Module    :  [Behaviour Description of the Controller]
--    Purpose   :  [Implement the functions of the controller]
--    Note      :  [The program is designed to handle four itv trees of outputs]

ARCHITECTURE behaviour of Channel_handler IS

   TYPE boolean IS (false, true);
BEGIN                                          -- behaviour of Channel Handler
```



```
PROCESS

  ------------------------------------------------------------------------
  -- declare constants and variables
  ------------------------------------------------------------------------

  CONSTANT itvconst     : std_ulogic_vector(9 DOWNTO 0) := "0000000000";
  CONSTANT itvweak      : std_ulogic_vector(9 DOWNTO 0) := "ZZZZZZZZZZ";
  CONSTANT channelconst : cell_head_type := "00000000000000000001111111111";

  VARIABLE nodeptr0     : itv_tree_ptr;          -- declare first itv tree root
  VARIABLE nodeptr1     : itv_tree_ptr;          -- declare second itv tree root
  VARIABLE nodeptr2     : itv_tree_ptr;          -- declare third itv tree root
  VARIABLE nodeptr3     : itv_tree_ptr;          -- declare forth itv tree root

  VARIABLE itvtree0     : itv_tree;              -- declare first itv tree
  VARIABLE itvtree1     : itv_tree;              -- declare second itv tree
  VARIABLE itvtree2     : itv_tree;              -- declare third itv tree
  VARIABLE itvtree3     : itv_tree;              -- declare forth itv tree

  VARIABLE cellhead     : cell_head_type;
  VARIABLE enportno     : oe_type;
  VARIABLE levelvalue   : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE channelid    : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE channelspeed : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE channeltype  : std_ulogic_vector(1 DOWNTO 0);
  VARIABLE channelflag  : std_ulogic_vector(5 DOWNTO 0);
  VARIABLE channelitv   : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE channelmask  : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE channelkey   : cell_head_type;
  VARIABLE channeloutid : cell_head_type;
  VARIABLE speedint     : integer;
  VARIABLE isDone       : boolean;
  VARIABLE isValue      : boolean;
  VARIABLE isChannel    : boolean;
  VARIABLE isRelease    : boolean;
  VARIABLE isReleased   : boolean;
  VARIABLE portnoint    : integer := 0;          -- used by getchanneloutid
  VARIABLE index        : integer := 0;

  ------------------------------------------------------------------
  -- this procedure is used to create the node of itv tree
  ------------------------------------------------------------------

  PROCEDURE createnode(VARIABLE ptr     : INOUT itv_tree_ptr;
                       CONSTANT tempts  : IN    integer) IS
    VARIABLE temp_ptr : itv_tree_ptr;
  BEGIN
    temp_ptr := NEW itv_tree_node;
    temp_ptr.leftptr  := NULL;
    temp_ptr.rightptr := NULL;
    temp_ptr.isnode   := true;
    temp_ptr.ts       := 2*tempts;
    ptr := temp_ptr;
  END createnode;

  PROCEDURE initialisetree IS
  BEGIN
    createnode(itvtree0.rootptr, 0);             -- initialise first itv tree
    itvtree0.rootptr.itv := itvconst;
```

```
   itvtree0.rootptr.ts  := 1;

   createnode(itvtree1.rootptr, 0);            -- initialise second itv tree
   itvtree1.rootptr.itv := itvconst;
   itvtree1.rootptr.ts  := 1;

   createnode(itvtree2.rootptr, 0);            -- initialise third itv tree
   itvtree2.rootptr.itv := itvconst;
   itvtree2.rootptr.ts  := 1;

   createnode(itvtree3.rootptr, 0);            -- initialise forth itv tree
   itvtree3.rootptr.itv := itvconst;
   itvtree3.rootptr.ts  := 1;

END initialisetree;


-------------------------------------------------------------------
-- this program is used to travel the tree
-------------------------------------------------------------------

PROCEDURE inOrderTravel(VARIABLE ptr      : INOUT itv_tree_ptr) IS
  VARIABLE temp_ptr : itv_tree_ptr;
BEGIN
  if ptr/=NULL then
     inOrderTravel(ptr.leftptr);
     if ptr.ts>tempnodeptr.ts AND ptr.ts<=speedint AND ptr.isnode=true then
        tempnodeptr := ptr;
     end if;
     inOrderTravel(ptr.rightptr);
  end if;
END inOrderTravel;


-------------------------------------------------------------------
-- this program is used to build a tree
-------------------------------------------------------------------

PROCEDURE reBuildTree(VARIABLE temptree : INOUT itv_tree;
                      CONSTANT channel  : IN    std_ulogic_vector(9 DOWNTO 0);
                      VARIABLE tempitv  : OUT   std_ulogic_vector(9 DOWNTO 0);
                      VARIABLE isbuild  : OUT   boolean) IS
  VARIABLE temp_ptr  : itv_tree_ptr;
  VARIABLE temp_ptr1 : itv_tree_ptr;
  VARIABLE temp_itv  : std_ulogic_vector(9 DOWNTO 0);
BEGIN
  tempnodeptr := temptree.rootptr; -- initialise tempnodeptr for inOrderTravel
  temp_ptr1 := temptree.rootptr;
  inOrderTravel(temp_ptr1);
  temp_ptr := tempnodeptr;

  if not (temp_ptr.isnode=true and temp_ptr.ts<speedint) then
     assert false
     report "The itv tree is not return a correct pointer"
     severity error;
  end if;

  if temp_ptr.isnode=true and temp_ptr.ts<speedint then
     createNode(temp_ptr1, temp_ptr.ts);
     levelvalue := To_StduLogicVector(Int_To_BitVector(temp_ptr.ts, 10));
```



```
     if temp_ptr.leftptr=NULL then
        temp_ptr1.itv    := temp_ptr.itv;       -- parent itv and "0"
        temp_ptr.leftptr := temp_ptr1;
     else
        temp_ptr1.itv    := temp_ptr.itv OR levelvalue;
                                                -- parent itv and ts value
        temp_ptr.rightptr:= temp_ptr1;
     end if;
     temp_ptr := temp_ptr1;


     while temp_ptr.ts<speedint loop
        createNode(temp_ptr1, temp_ptr.ts);
        temp_ptr1.itv    := temp_ptr.itv;       -- parent itv and "0"
        temp_ptr.leftptr := temp_ptr1;
        temp_ptr := temp_ptr1;
     end loop;
     temp_ptr.channelid  := channel;
     temp_ptr.isnode := false;
     tempitv := temp_ptr.itv;                   -- assign current itv
     isbuild := true;

  else
     isbuild := false;
  end if;

END reBuildTree;


-------------------------------------------------------------------
-- this program is used to in order search a tree
-------------------------------------------------------------------

PROCEDURE inOrdersearch(VARIABLE ptr      : INOUT itv_tree_ptr;
                        VARIABLE tempptr  : OUT   itv_tree_ptr;
                        CONSTANT speed    : IN    integer) IS
  VARIABLE temp_ptr : itv_tree_ptr;
BEGIN

  testptr1(j) := ptr;
  j := j+1;
  if ptr/=NULL and ptr.ts<speed then
     inOrdersearch(ptr.leftptr, tempptr, speed);
     if ptr.isnode=true and ptr.ts<speed then
        if ptr.rightptr=NULL then
           i := i+1;
           if isValue=true then
              tempptr  := ptr;
              isValue  := false;
           end if;
           return ;
        else
           inOrdersearch(ptr.rightptr, tempptr, speed);
           if ptr.leftptr=NULL then             -- new line need to test
              if isValue=true then
                 tempptr := ptr;
                 isValue := false;
              end if;
              return;
           end if;
```

```
        end if;
      else
        return;
      end if;
    end if;
END inOrdersearch;


-------------------------------------------------------------------------
-- this program is used to establish a tree
-------------------------------------------------------------------------

PROCEDURE establishTree(VARIABLE temptree: INOUT itv_tree;
                        CONSTANT channel : IN  std_ulogic_vector(9 DOWNTO 0);
                        CONSTANT speed   : IN  integer;
                        VARIABLE tempitv : OUT std_ulogic_vector(9 DOWNTO 0);
                        VARIABLE isbuild : OUT boolean) IS
  VARIABLE temp_ptr  : itv_tree_ptr;
  VARIABLE temp_ptr1 : itv_tree_ptr;
  VARIABLE temp_itv  : std_ulogic_vector(9 DOWNTO 0);

BEGIN
  isValue := true;                    -- initialise isValue for inOrdersearch
  temp_ptr1 := temptree.rootptr;
  inOrdersearch(temp_ptr1, temp_ptr, speed);

  if not (temp_ptr.isnode=true and temp_ptr.ts<speed) then
     assert false
     report "The itv tree is not return a correct pointer"
     severity error;
  end if;

  if temp_ptr.isnode=true and temp_ptr.ts<speed then
     createNode(temp_ptr1, temp_ptr.ts);
     levelvalue := To_StduLogicVector(Int_To_BitVector(temp_ptr.ts, 10));
     if temp_ptr.leftptr=NULL then
        temp_ptr1.itv    := temp_ptr.itv;       -- parent itv and "0"
        temp_ptr.leftptr := temp_ptr1;
     else
        temp_ptr1.itv    := temp_ptr.itv OR levelvalue;
                                                -- parent itv and ts value
        temp_ptr.rightptr:= temp_ptr1;
     end if;

     temp_ptr := temp_ptr1;

     k := k+1;
     while temp_ptr.ts<speed loop
        createNode(temp_ptr1, temp_ptr.ts);
        temp_ptr1.itv    := temp_ptr.itv;       -- parent itv and "0"
        temp_ptr.leftptr := temp_ptr1;
        temp_ptr := temp_ptr1;

     end loop;
     temp_ptr.channelid  := channel;
     temp_ptr.isnode := false;
     tempitv := temp_ptr.itv;                   -- assign current itv
     isbuild := true;
  else
     isbuild := false;
```



```
  end if;

END establishTree;


-------------------------------------------------------------------------
-- this program is used to set channel mask value
-------------------------------------------------------------------------
PROCEDURE setchannelmask(CONSTANT speed : IN  std_ulogic_vector(9 DOWNTO 0);
                         VARIABLE mask  : OUT std_ulogic_vector(9 DOWNTO 0))
IS
  VARIABLE i : integer := 0;
BEGIN
  while i <= 9 loop                             -- get mask value
    if speed(i)='1' then
       exit;
    else
       mask(i) := '0';
    end if;
    i := i + 1;
  end loop;
  while i <= 9 loop
    mask(i) := '1';
    i := i + 1;
  end loop;

END setchannelmask;



-------------------------------------------------------------------------
-- this procedure is used to get the re-assembly channel id and header
-------------------------------------------------------------------------

PROCEDURE getchanneloutid(VARIABLE channelin  : IN  cell_head_type;
                          VARIABLE channelout : OUT cell_head_type) IS
  VARIABLE channelvar : cell_head_type := "00000011000000000001111111111";
  VARIABLE portno     : oe_type;
  VARIABLE i          : integer;
  VARIABLE j          : integer;
BEGIN
  j := 0;
  channelout := channelvar AND channelin;
  portno := To_StdULogicVector(Int_To_BitVector(portnoint,4));
                                                -- create output port number
  if portnoint=3 then
     channelout(10) := '0';
     channelout(11) := '0';
     channelout(12) := '0';
     channelout(13) := '0';
  else
  for i in 10 to 13 loop
      channelout(i) := portno(j);               -- assign output port number
      j := j+1;
  end loop;
  end if;

  j := 0;
  for i in 14 to 17 loop
      channelout(i) := enportno(j);             -- assign input port number
```

```
      j := j+1;
   end loop;

   if portnoint < 3 then                          -- output port from 0# to 3#
      portnoint := portnoint + 1;
   else
      portnoint := 0;
   end if;

END getchanneloutid;



-------------------------------------------------------------------
-- this procedure is used to seek a tree
-------------------------------------------------------------------

PROCEDURE inOrderSeek(VARIABLE ptr     : INOUT itv_tree_ptr;
                      CONSTANT channel : IN    std_ulogic_vector(9 DOWNTO 0))
IS
  VARIABLE temp_ptr : itv_tree_ptr;
BEGIN

  if ptr/=NULL then
     inOrderSeek(ptr.leftptr, channel);
     if isChannel=true then                -- channel is found
        if isReleased=true then
           return;
        else
           deallocate(ptr.leftptr);
           ptr.leftptr := NULL;
           isReleased  := true;
        end if;
     end if;

     if ptr.isnode=false then              -- current node is leaf
        if ptr.channelid=channel then      -- check whether is released channel
           isChannel := true;              -- find this channel
           return;                         -- return to last call
        else                               -- channel is not given channel
           return;                         -- continue to seek
        end if;
     end if;

     inOrderSeek(ptr.rightptr, channel);
     if isChannel=true then                -- channel is found
        if isReleased=true then
           return;
        else
           deallocate(ptr.rightptr);
           ptr.rightptr := NULL;
           isReleased   := true;
        end if;
     end if;

  end if;                                  -- end ptr/=NULL
END inOrderSeek;


-------------------------------------------------------------------
```



```
-- this procedure is used to release a tree
-------------------------------------------------------------------

PROCEDURE releasechannel(VARIABLE temptree : INOUT itv_tree;
                         CONSTANT channel  : IN std_ulogic_vector(9 DOWNTO 0);
                         VARIABLE isRelease: OUT boolean)
IS
  VARIABLE temp_ptr  : itv_tree_ptr;
BEGIN
  isChannel  := false;                 -- initialise isChannel for inOrderSeek
  isReleased := false;                 -- initialise isReleased for inOrderSeek

  temp_ptr   := temptree.rootptr;
  inOrderSeek(temp_ptr, channel);
  if isChannel=true and isReleased=true then
                                       -- channel has been found and released
     isRelease := true;
  else
     isRelease := false;
  end if;
END releasechannel;


BEGIN                                              -- process is begin
  --
  --check for reset active
  --
  wait until reset='1' or reset='0';
  if reset ='1' then
     initialiseTree;
     isDone      := false;
     isRelease   := false;
     isReleased  := false;
     setupcell   <= '0' after Tpd;
     releasecell <= '0' after Tpd;           -- initialise Itv Tree
     head_bus    <= null after Tpd;
     itv         <= itvweak after Tpd;
     mask        <= itvweak after Tpd;
     wait until reset='0';
  end if;

  loop
  if reset='0' then
     read  <= '0' after Tpd;
     write <= '0' after Tpd;
     if clk1 ='1' then                             -- accept cell state
        wait until acceptcell='1' or clk1='0';
        if acceptcell='1' then
           wait for Tpd;                           -- wait for Tpd time
           cellhead := head_bus;
           enportno := en_port_no;
        if cellhead(27 DOWNTO 22)="111111" then  -- incoming cell is setup cell
           resetfull    <= '1' after Tpd;
           setupcell    <= '1' after Tpd;
           channelid    := cellhead(9  DOWNTO 0);
           channelspeed := cellhead(19 DOWNTO 10);
           channeltype  := cellhead(21 DOWNTO 20);

           getchanneloutid(cellhead, channeloutid);
                                       -- get the output port number
```

```
    if channeltype="00" OR channeltype="01" then
                                -- new channnel is CBR/VBR channel
       speedint := BitVector_To_Int(To_BitVector(channelspeed));
       if speedint=0 then
          speedint := 1024;
       end if;

       if enportno="0000" then
            establishTree(itvtree0, channelid, speedint,
                          channelitv, isDone);
                          -- find and assign a bandwidth for the new channel
       elsif enportno="0001" then
            establishTree(itvtree1, channelid, speedint,
                          channelitv, isDone);
                          -- find and assign a bandwidth for the new channel
       elsif enportno="0010" then
            establishTree(itvtree2, channelid, speedint,
                          channelitv, isDone);
                          -- find and assign a bandwidth for the new channel
       else
            establishTree(itvtree3, channelid, speedint,
                          channelitv, isDone);
                          -- find and assign a bandwidth for the new channel
       end if;

       if isDone=true then      -- new channel connection is built
          setchannelmask(channelspeed, channelmask);
          itv  <= channelitv  after Tpd;
          mask <= channelmask after Tpd;
          head_bus <= channeloutid after Tpd;
          wait until clk1='0';
       else                     -- new channel connection is not built
          assert false
          report "No enough space for the new channel, connection fail !"
          severity NOTE;
          wait until clk1='0';
       end if;
    else                -- new channel is ABR/UBR channel
       itv  <= itvweak after Tpd;
                        -- itv is not used by ABR/UBR channel header
       mask <= itvweak after Tpd;
                        -- mask is not used by ABR/UBR channel header
       head_bus <= channeloutid after Tpd;
                        -- put the new channel information on the head bus
       wait until clk1='0';
    end if;             -- end channel type check

    resetfull <= '0' after Tpd;
    setupcell <= '0' after Tpd;
    head_bus  <= null after Tpd;
    itv       <= itvweak after Tpd;
    mask      <= itvweak after Tpd;

elsif cellhead(27 DOWNTO 22)="111000" then
                        -- incoming cell is relese channel type
       resetfull    <= '1' after Tpd;
       releasecell  <= '1' after Tpd;
       setupcell    <= '1' after Tpd;
       channelid    := cellhead(9  DOWNTO 0);
```



```
       releasechannel(itvtree0, channelid, isRelease);
       if isRelease=false then
          assert false
          report "Given channel is not found !"
          severity NOTE;
       else
          channeloutid(9 DOWNTO 0) := channelid;
          channeloutid := channeloutid AND channelconst;
          head_bus <= channeloutid after Tpd;
       end if;
       wait until clk1='0';
       resetfull   <= '0' after Tpd;
       setupcell   <= '0' after Tpd;
       releasecell <= '0' after Tpd;
       head_bus    <= null after Tpd;
elsif cellhead(27 DOWNTO 22)="111100" then
                        -- incoming cell is setup cell
       resetfull    <= '1' after Tpd;
       setupcell    <= '1' after Tpd;
       channelid    := cellhead(9  DOWNTO 0);
       channelspeed := cellhead(19 DOWNTO 10);
       channeltype  := cellhead(21 DOWNTO 20);

       getchanneloutid(cellhead, channeloutid);
                        -- work out the new cell header

       speedint := BitVector_To_Int(To_BitVector(channelspeed));
       if speedint=0 then
          speedint := 1024;
       end if;
       reBuildTree(itvtree0, channelid, channelitv, isDone);
                        -- insert the new node in the tree
    if isDone=true then
       setchannelmask(channelspeed, channelmask);
                        -- calculate the mask value of CBR/VBR channel
       itv  <= channelitv  after Tpd;
       mask <= channelmask after Tpd;
       head_bus <= channeloutid after Tpd;
       wait until clk1='0';
    else
       assert false
       report "No enough space for the new channel, connection fail !"
       severity NOTE;
       wait until clk1='0';
    end if;
    resetfull <= '0' ;
    setupcell <= '0' ;
    head_bus  <= null after Tpd;
    itv       <= itvweak after Tpd;
    mask      <= itvweak after Tpd;

else                                    -- incoming cell is normal cell
       write     <= '1' after Tpd;
       read      <= '0' after Tpd;
       resetfull <= '1' after Tpd;
       wait until clk1='0';
       resetfull <= '0' ;
       write     <= '0' after Tpd;
end if;                                 -- end cellhead/="111111"
end if;                                 -- end acceptcell='1'
```

```
      end if;                                        -- end clk1='0'

      if clk1='0' then                               -- outgoing cell state
         wait until match='1' or outgoing='1' or clk1='1';
         if outgoing='1' or match='1' then
            read  <= '1' after Tpd;
            write <= '0' after Tpd;
            resetfull <= '1' after Tpd;
            wait until clk1='1';
            read  <= '0' after Tpd;
            resetfull <= '0';
         end if;
      end if;                                        -- end clk1='0'

    else                                             -- reset/='0'
      exit;
    end if;                                          -- end reset='0'
    end loop;                                        -- end loop

END process;
END behaviour;



--------------------------------------------------------------------------------
--    File name :  countent.vhd
--    Title     :  [ATM Counter]
--    Module    :  [The Counter Entity]
--    Purpose   :  [Define the entity of the counter]
--------------------------------------------------------------------------------
--    Modification History :
--          Date                  Author          Revision        Comments
--    Tur August 20 1997          Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;

ENTITY regular_counter is
  GENERIC(Tpw    : Time :=12 ns);
  PORT    (clk1   : in   std_ulogic;
           clk2   : in   std_ulogic;
           reset  : in   std_ulogic;
           count  : out  std_ulogic_vector(9 DOWNTO 0));
END regular_counter;


--    File name :  countarc.vhd
--    Title     :  [ATM Counter]
--    Module    :  [Behaviour Description of the counter]
--    Purpose   :  [Implement the functions of the counter]

ARCHITECTURE behaviour of regular_counter is
```



```
  CONSTANT counter_period : Time :=Tpw;
BEGIN

PROCESS
  VARIABLE counter_value  : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE counter_temp   : integer := 0;
  VARIABLE i              : integer := 0;
BEGIN
  --
  --check for reset active
  --
  wait until reset='1' or reset='0';
  if reset ='1' then
     counter_value := "0000000000";     -- initialise counter value
     wait until reset='0';
  end if;

  loop                          -- execute following statement until reset='1'
  if reset='0' then
     if clk1='1' then
        wait until clk1='0';
     end if;

     if clk1='0' AND i=0 then -- change count value every four clock cycle
        count <= counter_value;
        wait until clk1='1';
        if counter_value="1111111111" then
           counter_value := "0000000000";
        else
           counter_temp  := BitVector_To_Int(To_BitVector(counter_value)) + 1;
           counter_value := To_StdULogicVector
                            (Int_To_BitVector(counter_temp, 10));
        end if;
     else
        wait until clk1='1';
     end if;

     i := i+1;
     if i>3 then
        i := 0;
     end if;

  else
     exit;
  end if;
  end loop;
END process;
END behaviour;



--------------------------------------------------------------------------------
--    File name :  cqlinpkg.vhd
--    Title     :  [ATM Cell Queue Link Memory]
--    Module    :  [Package Declaration]
--    Purpose   :  [Declare the types and procedures for the CQlink Memory]
--------------------------------------------------------------------------------
--    Modification History :
--          Date            Author          Revision        Comments
--    Tur August 21  1997   Bo Yan          Rev1.0.0        Creation
```

```
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;
USE std.textio.all;

USE work.atm_types.all;

PACKAGE cq_link_pkg IS


  TYPE cqlink_blks IS array(0 to CellMemSize) of integer;

  SUBTYPE  cq_map_bus is std_ulogic_vector(9 DOWNTO 0);
  TYPE     cq_map_bus_array is array(integer range <>) of cq_map_bus;
  FUNCTION resolve_cq_map_bus(driver : in cq_map_bus_array)
           RETURN cq_map_bus;
  SUBTYPE  bus_cq_map_bus is resolve_cq_map_bus cq_map_bus;


--Ddclare a procedure which initialises the cell memory pointer
PROCEDURE initialise(VARIABLE cqlinkblk : INOUT cqlink_blks;
                     VARIABLE freeaddr  : INOUT integer);

--Delare a procedure which adds a new pointer into the cell memory
PROCEDURE add (VARIABLE cqlinkblk      : INOUT cqlink_blks;
               VARIABLE freeaddr       : INOUT integer);


--Declare a procedure which removes a pointer from the cell memory
PROCEDURE remove (VARIABLE cqlinkblk   : INOUT cqlink_blks;
                  VARIABLE freeaddr    : INOUT integer);


END cq_link_pkg;



--   File name :  cqlinbdy.vhd
--   Title     :  [ATM Cell Queue Link Memory]
--   Module    :  [Package Body of Cqlink Memory]
--   Purpose   :  [Implement the procedure which are declared in cqlink package]

PACKAGE BODY cq_link_pkg IS

-------------------------------------------------------------------
-- Define a resoluation function for cq_map_bus
-------------------------------------------------------------------
FUNCTION resolve_cq_map_bus(driver : in cq_map_bus_array)
         RETURN cq_map_bus IS
  CONSTANT addr   : cq_map_bus := "0000000000";
  VARIABLE result : cq_map_bus := addr;
BEGIN
  for i in driver'range loop
      result := result or driver(i);
  end loop;
  return result;
```



```
END resolve_cq_map_bus;


-------------------------------------------------------------------
--Define a procedure that initialise Cq Link Memory Block
-------------------------------------------------------------------
PROCEDURE initialise(VARIABLE cqlinkblk : INOUT cqlink_blks;
                     VARIABLE freeaddr  : INOUT integer) IS
  VARIABLE index : integer := 0;
  VARIABLE nextaddr : integer := 0;
BEGIN
  for index in 0 to CellMemSize loop
      if (index/=CellMemSize) then
         nextaddr := index+1;
         cqlinkblk(index) := nextaddr;
      else
         cqlinkblk(index) := 0;
      end if;
  end loop;
  freeaddr := 0;

END initialise;


-------------------------------------------------------------------
--Define a procedure that add new pointer into the cqlink memory
-------------------------------------------------------------------
PROCEDURE add (VARIABLE cqlinkblk      : INOUT cqlink_blks;
               VARIABLE freeaddr       : INOUT integer) IS

  VARIABLE index       : integer := 0;

  BEGIN
    if (freeaddr < 0) or (freeaddr > CellMemSize) then
       assert false
       report "The given address is over scope of Cell Memory !"
       severity error;
    end if;
    freeaddr := cqlinkblk(index);
END add;



-------------------------------------------------------------------
--Declare a procedure that remove a pointer from the cqlink memory
-------------------------------------------------------------------
PROCEDURE remove (VARIABLE cqlinkblk   : INOUT cqlink_blks;
                  VARIABLE freeaddr    : INOUT integer) IS

  VARIABLE index       : integer := 0;

BEGIN
  if (freeaddr < 0) or (freeaddr > CellMemSize) then
     assert false
     report "The given address is over scope of Cell Memory !"
     severity error;
  end if;
  cqlinkblk(freeaddr) := freeaddr;      -- not correct
END remove;
```

```
END cq_link_pkg;


-------------------------------------------------------------------------------
--   File name :  cqlinent.vhd
--   Title     :  [ATM Cell Queue Link Memory]
--   Module    :  [The Cell Queue Link Entity]
--   Purpose   :  [Define the entity of the cqlink]

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;
USE work.cq_link_pkg.all;
USE work.mem_pkg.all;

ENTITY cq_link is
  GENERIC( Tpd         : Time  := unit_delay);
  PORT   ( read        : in    std_ulogic;
           write       : in    std_ulogic;
           reset       : in    std_ulogic;
           clk1        : in    std_ulogic;
           ready       : in    std_ulogic;
           channeltype: in    std_ulogic;
           a_bus       : inout bus_cell_mem_addr bus;
           cq_map_bus  : inout bus_cq_map_bus bus);
END cq_link;


-------------------------------------------------------------------------------
--   File name :  cqlinarc.vhd
--   Title     :  [ATM Cell Queue Link Memory]
--   Module    :  [Behaviour Description of the Cell Queue Link Memory]
--   Purpose   :  [Implement the functions of the cqlink]

ARCHITECTURE behaviour of cq_link IS

BEGIN

  accept : process
  variable cqlinklist   : cqlink_blks;
  variable freecell     : integer range 0 to CellMemSize;
  variable freecells    : integer range 0 to CellMemSize+1 := CellMemSize+1;
  variable nextfreecell : integer range 0 to CellMemSize := 0;
  variable lastfreecell : integer range 0 to CellMemSize := 0;
  variable cqmemaddr    : cell_mem_addr := "0000000000";
  variable celltail     : cell_mem_addr := "0000000000";

  --
  --declare variables that for outgoing stage
  --
  variable headentry    : integer range 0 to CellMemSize;
  variable newheadaddr  : integer range 0 to CellMemSize;
  variable cellhead     : cell_mem_addr := "0000000000";
  variable addr1        : cell_mem_addr := "0000001111";
```



```
variable addr2        : cell_mem_addr := "0000001110";
begin

  --
  --check for reset active
  --
  wait until reset='0' or reset='1';
  if reset='1' then
     initialise(cqlinklist, freecell);
     a_bus <= null after Tpd;
     wait until reset='0';
  end if;

  --
  --output cell memory address
  --

  loop
  if reset='0' then
     if clk1='1' then                -- begin accept cell state
        cq_map_bus <= null after Tpd;
        cqmemaddr := To_StdULogicVector
                     (Int_To_BitVector(freecell, CellMemWidth));
        a_bus <= cqmemaddr after Tpd;
                  -- output address of cell memory that will store cell data
        wait until write='1' or clk1='0';
                  -- wait for write or read siginal coming

        if write='1' then
           wait until channeltype='0' OR channeltype='1';
           celltail := cq_map_bus;
                        -- read the cell queue tail from map memory
           nextfreecell := cqlinklist(freecell);
                        -- get the next free address from the cqlink

           if celltail/=cqmemaddr AND channeltype='0' then
                        -- cell queue of ABR/UBR channel is already existed
              lastfreecell := BitVector_To_Int(To_BitVector(celltail));
                        -- convert the cell queue tail into integer
              cqlinklist(lastfreecell) := freecell;
                        -- append a new cell into cell queue
           end if;

           cqlinklist(freecell) := 0;      -- assign zero for new cell pointer
           freecell := nextfreecell;       -- update freecell value

           if freecells=1 then
              assert false
              report "The Cell Memory is full! It had to discard current cell"
              severity WARNING;
           else
              freecells := freecells - 1; -- cell memory decrease 1
           end if;

           wait until ready='1'; -- wait until cell memory write data is ready
           a_bus <= null after Tpd;
           wait until clk1 ='0';

        end if;                                  -- end write='1'
     end if;                                     -- end clk1='1'
```

```
      if clk1='0' then                          -- begin to outgoing cell state
         a_bus <= null;
         wait until read='1' or clk1='1';
         if read='1' then
            wait until channeltype='0' OR channeltype='1';
            cellhead    := a_bus;
            headentry   := BitVector_To_Int(To_BitVector(cellhead));
            newheadaddr := cqlinklist(headentry);
            cellhead    := To_StdULogicVector(Int_To_BitVector(newheadaddr,
                                              CellMemWidth));

            cq_map_bus  <= cellhead after Tpd;
                        -- output new cell queue head pointer to map memory

            --
            --update cqlink
            --
            cqlinklist(headentry) := freecell;
                     -- give new free link for just released memory
            freecell  := headentry;             -- update freecell value.
            freecells := freecells + 1;         -- cell memory space increase 1

            wait until clk1 ='1';
         end if;                                -- end read='1'
      end if;                                   -- end clk1='0'
   else                                         -- reset/='0'
      exit;
   end if;                                      -- end reset='0'

   end loop;
END process accept;
END behaviour;


--------------------------------------------------------------------------------
--   File name :  inporent.vhd
--   Title     :  [ATM Input Port 0#: ]
--   Module    :  [ATM Input Port 0# Entity]
--   Purpose   :  [Define the entity of the Input Port 0#]
--------------------------------------------------------------------------------
--   Modification History :
--         Date            Author          Revision        Comments
--   Tur August 10  1997     Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;
USE work.mem_pkg.all;

ENTITY input_buffer00 is
  GENERIC( Tpd        : Time := unit_delay);
  PORT   ( oe         : in  oe_type;
           reset      : in  std_ulogic;
```



```
           resetfull  : in  std_ulogic;
           setupcell  : in  std_ulogic;
           ready      : in  std_ulogic;
           d_bus      : out bus_cell_mem_data bus;
           head_bus   : out bus_cell_head_type bus;
           full00     : out std_ulogic );
END input_buffer00;


--------------------------------------------------------------------------------
--   File name :  inporarc.vhd
--   Title     :  [ATM Input Port 0# ]
--   Module    :  [Behaviour Description of the input port 0#]
--   Purpose   :  [Implement the functions of the input port 0#]

ARCHITECTURE behaviour of input_buffer00 is

BEGIN

  PROCESS
  constant cbrchannel12    : cell_head_type := "00000000000000000100000000100";
  constant cbrchannel14    : cell_head_type := "00000000000000001000000001000";
  constant cbrchannel18    : cell_head_type := "00000000000000010000000001100";
  constant cbrchannel28    : cell_head_type := "00000000000000010000000010000";
  constant setupcellhead   : cell_head_type := "11111100000000000000000000000";
  constant channelhead     : cell_head_type := "00000000000000000000000011111";
  variable channeldata     : cell_head_type;
  variable setupcbrhead    : cell_head_type;

  variable full_signal     : std_ulogic :='1';
  variable oe_value        : oe_type := "0000";
  variable setupdata       : cell_mem_data  :="11111111";
  variable data            : cell_mem_data  :="00001111";

  variable index           : integer := 0;
  variable i               : integer := 0;
  variable casevalue       : integer := 0;

  begin

    --
    --check for reset active
    --
    wait until reset='0' or reset='1';
    if reset='1' then
       wait until reset='0';
       full00 <= '1';
    end if;

    --
    --set full signal and output cell from input buffer until reset='1'
    --
    loop
    if  reset='0' then
        d_bus    <= null after Tpd;
        head_bus <= null after Tpd;
        wait until oe=oe_value or (resetfull='1' and oe=oe_value) ;
                                  -- wait for priority circuit check full signal
        if oe=oe_value then
           if i<4 then            -- setup CBR channel
```

```
        casevalue := i;
        if casevalue=0 then-- setup a CBR channel with 1/2 bandwidth speed
                setupcbrhead := setupcellhead OR cbrchannel12;
                d_bus     <= setupdata after Tpd;
                head_bus <= setupcbrhead after Tpd;
        elsif casevalue=1 then
                        -- setup a CBR channel with 1/4 bandwidth speed
                setupcbrhead := setupcellhead OR cbrchannel14;
                d_bus     <= setupdata after Tpd;
                head_bus <= setupcbrhead after Tpd;
        elsif casevalue=2 then
                        -- setup a CBR channel with 1/8 bandwidth speed
                setupcbrhead := setupcellhead OR cbrchannel18;
                d_bus     <= setupdata after Tpd;
                head_bus <= setupcbrhead after Tpd;
        elsif casevalue=3 then
                        -- setup a CBR channel with 1/8 bandwidth speed
                setupcbrhead := setupcellhead OR cbrchannel28;
                d_bus     <= setupdata after Tpd;
                head_bus <= setupcbrhead after Tpd;
        end if;

    else                                    -- input normal CBR cell
        casevalue := i mod 8;

        if casevalue=0 OR casevalue=2 OR casevalue=4 OR casevalue=6 then
            channeldata := channelhead AND cbrchannel12;
                        -- set channel id equal to 1/2 bandwidth CBR channel

            d_bus     <= data after Tpd;          -- input cell data
            head_bus <= channeldata after Tpd;  -- input cell head

        elsif casevalue=1 OR casevalue=5 then
            channeldata := channelhead AND cbrchannel14;
                        -- set channel id equal to 1/4 bandwidth CBR channel
            d_bus     <= data after Tpd;          -- input cell data
            head_bus <= channeldata after Tpd;  -- input cell head

        elsif casevalue=3 then
            channeldata := channelhead AND cbrchannel18;
                        -- set channel id equal to 1/8 bandwidth CBR channel
            d_bus     <= data after Tpd;          -- input cell data
            head_bus <= channeldata after Tpd;  -- input cell head

        elsif casevalue=7 then
            channeldata := channelhead AND cbrchannel28;
                        -- set channel id equal to 1/8 bandwidth CBR channel
            d_bus     <= data after Tpd;          -- input cell data
            head_bus <= channeldata after Tpd;  -- input cell head
        end if;                                 -- end casevalue check
        if data="11111111" then
            data := "00000000";
        end if;
        data := std_ulogic_vector(unsigned(data) + "00000001");
    end if;                                     -- end i<4 check
    i := i + 1;

    wait until resetfull='1' or setupcell='1';
    full00 <= '0' after Tpd;
    if setupcell='1' then
```



```
            d_bus     <= null after Tpd;
            head_bus <= null after Tpd;
            wait for 8*Tpd;
        else
            wait until ready='1';               -- wait cell memory is ready
        end if;
        full00 <= full_signal;
      end if;                                   -- end oe=oe_value
    else                                        -- reset/='0'
        exit;
    end if;                                     -- end reset='0'

    end loop;
  end process;
END behaviour;



--------------------------------------------------------------------------------
--    File name :  inserent.vhd
--    Title     :  [ATM Input Round Robin Server]
--    Module    :  [The Entity of the input server]
--    Purpose   :  [Define the entity of the input server]
--------------------------------------------------------------------------------
--    Modification History :
--          Date              Author          Revision        Comments
--    Tur August 20  1997     Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;

ENTITY inputserver is
  GENERIC (Tpd        : Time := unit_delay);
  PORT (   clk1       : in   std_ulogic;
           clk2       : in   std_ulogic;
           read       : in   std_ulogic;
           write      : in   std_ulogic;
           reset      : in   std_ulogic;
           full00     : in   std_ulogic;
           full01     : in   std_ulogic;
           full02     : in   std_ulogic;
           full03     : in   std_ulogic;
           full04     : in   std_ulogic;
           full05     : in   std_ulogic;
           full06     : in   std_ulogic;
           full07     : in   std_ulogic;
           full08     : in   std_ulogic;
           full09     : in   std_ulogic;
           full10     : in   std_ulogic;
           full11     : in   std_ulogic;
           full12     : in   std_ulogic;
           full13     : in   std_ulogic;
           full14     : in   std_ulogic;
```

```
          full15     : in   std_ulogic;
          acceptcell : out  std_ulogic;
          en_port_no : out  oe_type);
END inputserver;


--------------------------------------------------------------------------------
--    File name :  inserarc.vhd
--    Title     :  [ATM Input Round Robin Server]
--    Module    :  [Behaviour Description of the input server]
--    Purpose   :  [Implement the functions of input server module]

Architecture behaviour of inputserver is
  TYPE     boolean is(false, true);
BEGIN
  PROCESS
  VARIABLE currentport  : oe_type;
  VARIABLE encoderport  : oe_type;
  VARIABLE index        : integer := 0;
  VARIABLE fullvalue    : buf_full_type;
  VARIABLE inputcell    : boolean;


  ------------------------------------------------------------------------
  -- this program is used to initialise the input port round robin server
  ------------------------------------------------------------------------
  PROCEDURE initialise(fullblks    : INOUT buf_full_type;
                       portnumber  : INOUT oe_type;
                       bufferempty : INOUT boolean) IS
  BEGIN
    for index in 0 to BufferSize loop
        fullblks(index) := '0';
    end loop;
    portnumber  := "0000";
    bufferempty := true;
  END initialise;


  ------------------------------------------------------------------------
  -- this program is used to get signal from input port
  ------------------------------------------------------------------------
  PROCEDURE getfullsignal(fullblks : INOUT buf_full_type) IS
  BEGIN
    fullblks(0)  := full00;
    fullblks(1)  := full01;
    fullblks(2)  := full02;
    fullblks(3)  := full03;
    fullblks(4)  := full04;
    fullblks(5)  := full05;
    fullblks(6)  := full06;
    fullblks(7)  := full07;
    fullblks(8)  := full08;
    fullblks(9)  := full09;
    fullblks(10) := full10;
    fullblks(11) := full11;
    fullblks(12) := full12;
    fullblks(13) := full13;
    fullblks(14) := full14;
    fullblks(15) := full15;
  END getfullsignal;
```



```
------------------------------------------------------------------------
-- this program is used to check full signal of input port
------------------------------------------------------------------------
PROCEDURE checkfull(fullblks    : INOUT buf_full_type;
                    currport    : IN    oe_type;
                    portnumber  : OUT   oe_type;
                    bufferempty : OUT   boolean) IS
VARIABLE temp_index : integer;
BEGIN
  index := BitVector_to_Int(To_BitVector(currport));
  if (index<0) or (index>15) then              -- error checking
     assert false
     report "The input buffer number is out of scope !"
     severity error;
  end if;

  if (fullblks(index)='1') then      -- check current input port full signal
     portnumber  := currport;
     bufferempty := false;           -- current input buffer not empty
     return;
  end if;
  temp_index := index;
  index := index +1;
        if (index>15) then
           index := 0;
        end if;

  while (index /= temp_index) loop
                                     -- check rest of input port full signal
        if (fullblks(index)='1') then-- check given input port full signal
           portnumber  := To_StdULogicVector(Int_To_BitVector(index, 4));
           bufferempty := false;     -- given input port not empty
           return;
        end if;
        index := index +1;
        if (index>15) then
           index := 0;
        end if;

  end loop;
  bufferempty := true;                          -- all of input buffer empty
  return;
END checkfull;

begin                                           -- Process begin

  --
  --check for reset active
  --
  wait until reset='0' or reset='1';
  if reset='1' then                             -- reset state
     initialise(fullvalue, currentport, inputcell);
     acceptcell <= '0' after Tpd;
     wait until reset='0';
  end if;

  loop
  if reset='0' then
```

```
      if clk1='1' then
         getfullsignal(fullvalue);        -- read full signal from input buffer
         checkfull(fullvalue, currentport, encoderport, inputcell);
         if inputcell=false then          -- Input Buffer not empty
            acceptcell <= '1'         ;   -- output accept cell signal as true
            en_port_no <= encoderport ;   -- output encoded input buffer number

            wait until clk1='0';
            currentport := encoderport;
            if currentport /="1111" then
               currentport :=std_ulogic_vector(unsigned(currentport) + "0001");
            else
               currentport :="0000";
            end if;
         else
            acceptcell <= '0' after Tpd;  -- output accept cell signal as false

         end if;                          -- end inputcell check
      end if;                             -- end write='1' check
      if clk1='0' then                    -- outgoing cell stage
         en_port_no <= "ZZZZ";
         acceptcell <= '0';

         wait until clk1='1';

      end if;
   else                                   -- reset /='0'
      exit;
   end if;                                -- end reset='0'
   end loop;
END process;
END behaviour;
```

```
--------------------------------------------------------------------------------
--   File name :  mappkg.vhd
--   Title     :  [ATM Map Memory]
--   Module    :  [Package Declaration]
--   Purpose   :  [Declare the types and procedures for the Map Memory module]
--------------------------------------------------------------------------------
--   Modification History :
--         Date           Author          Revision        Comments
--   Tur August 21  1997    Bo Yan          Rev1.0_00       Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;
USE std.textio.all;

USE work.atm_types.all;

PACKAGE map_pkg01 IS

  TYPE Bit_4 is array(0 to 3) of bit;
  TYPE mem_type is array(0 to MapSize) of Cell_Head_Type;
  TYPE boolean is (false, true);
```



```
  CONSTANT MapWidth   : integer :=10;

  TYPE map_element_rec IS
                -- define the data structure of the channel header
       RECORD
         entryfree     : boolean;
         channelidin   : cell_head_type;
         inputport     : integer range 0 to BufferSize;
         channelidout  : cell_head_type;
         outputport    : integer range 0 to BufferSize;
         emptyqueue    : boolean;
         cellqueuehead : std_ulogic_vector(9 DOWNTO 0);
         cellqueuetail : std_ulogic_vector(9 DOWNTO 0);
         queuelength   : integer range 0 to CellMemSize;
         nextentry     : integer range 0 to MapSize;
         priority      : std_ulogic_vector(1 DOWNTO 0);
         channeltype   : std_ulogic_vector(1 DOWNTO 0);
         itv           : std_ulogic_vector(9 DOWNTO 0);
         mask          : std_ulogic_vector(9 DOWNTO 0);
       END RECORD;

  TYPE map_mem_blks IS array(0 to MapSize) of map_element_rec;

--Ddclare a procedure that initialise Map Memory pointer block
PROCEDURE initialise(VARIABLE mapmemblk : INOUT map_mem_blks);


--Declare a procedure that create a new element of map memory
PROCEDURE create (VARIABLE cell_head : IN cell_head_type);

--Delare a procedure that add new channel into map memory
PROCEDURE add (VARIABLE mapmemblk   : INOUT map_mem_blks;
               VARIABLE cellHead    : IN cell_head_type;
               VARIABLE inputPort   : IN integer;
               VARIABLE priority    : IN std_ulogic_vector(1 DOWNTO 0);
               VARIABLE channeltype : IN std_ulogic_vector(1 DOWNTO 0));



--Declare a procedure that remove a channel from map memory
PROCEDURE remove (VARIABLE mapmemblk   : INOUT map_mem_blks;
                  VARIABLE cellHead    : IN    cell_head_type;
                  VARIABLE width       : IN    integer);

--Declare a procedure that seek given channel in map_memory
FUNCTION seek (CONSTANT mapmemblk : IN map_mem_blks;
               CONSTANT cell_head : IN cell_head_type)
               RETURN integer;


--Declare a procedure that check whether map memory is empty
FUNCTION isMapEmpty RETURN boolean;

--Declare a procedure that check whether map memory is full
FUNCTION isMapFull RETURN boolean;


END map_pkg01;
```

```
--------------------------------------------------------------------------------
--   File name :  mapbdy01.vhd
--   Title     :  [ATM Map Memory]
--   Module    :  [Package Body]
--   Purpose   :  [Implement the procedures which are declared in the map memory
--                 package]

PACKAGE BODY map_pkg01 IS

--------------------------------------------------------------------
--Declare a procedure that initialise Map Memory pointer block
--------------------------------------------------------------------
PROCEDURE initialise(VARIABLE mapmemblk : INOUT map_mem_blks) IS
  VARIABLE nextentry : integer := 0;
  VARIABLE index     : integer :=0;
BEGIN
   for index in 0 to MapSize loop
       mapmemblk(index).entryfree   := true;
       mapmemblk(index).emptyqueue  := true;
       mapmemblk(index).queuelength := 0;
       if (index/=MapSize) then
           nextentry := index + 1;
       else
           nextentry := 0;
       end if;
       mapmemblk(index).nextentry   := nextentry;
   end loop;

END initialise;


--------------------------------------------------------------------
--Declare a procedure that create a new element of map memory
--------------------------------------------------------------------
PROCEDURE create (VARIABLE cell_head : IN cell_head_type) IS
BEGIN

END create;


--------------------------------------------------------------------
--Delare a procedure that add new channel into map memory
--------------------------------------------------------------------
PROCEDURE add (VARIABLE mapmemblk    : INOUT map_mem_blks;
               VARIABLE cellHead     : IN Cell_Head_Type;
               VARIABLE inputPort    : IN integer;
               VARIABLE priority     : IN std_ulogic_vector(1 DOWNTO 0);
               VARIABLE channeltype  : IN std_ulogic_vector(1 DOWNTO 0)) IS

  VARIABLE index       : integer := 0;
  VARIABLE full        : boolean;
  VARIABLE size        : integer :=MapWidth;
  VARIABLE inport      : integer := 0;
  VARIABLE outport     : integer := 0;

  BEGIN
    full := isMapFull;
    if (full=true) then
       assert false
```



```
       report "Map Memory is Full !"
       severity error;
    end if;
    for index in 0 to MapSize loop
        if (mapmemblk(index).entryfree = true) then
           mapmemblk(index).channelidin   := cellHead;
           mapmemblk(index).inputport     := inport;
           mapmemblk(index).channelidout  := cellHead;
           mapmemblk(index).outputport    := outport;
           mapmemblk(index).emptyqueue    := true;
           mapmemblk(index).cellqueuehead := "1111111111";
           mapmemblk(index).cellqueuetail := "1111111111";
           mapmemblk(index).queuelength   := 0;
           mapmemblk(index).nextentry     := 0;
           mapmemblk(index).priority      := priority;
           mapmemblk(index).channeltype   := channeltype;
        end if;
    end loop;

END add;


--------------------------------------------------------------------
--Declare a procedure that remove a channel from map memory
--------------------------------------------------------------------
PROCEDURE remove (VARIABLE mapmemblk    : INOUT map_mem_blks;
                  VARIABLE cellHead     : IN    cell_head_type;
                  VARIABLE width        : IN    integer) IS
  VARIABLE tempElement : map_element_rec;
  VARIABLE index       : integer := 0;

BEGIN
  index := seek(mapmemblk, cellHead);
  if (index=-1) then
     assert false
     report "Cell channel can't match !"
     severity error;
  end if;
  mapmemblk(index).entryfree := true;
END remove;


--------------------------------------------------------------------
--Declare a procedure that seek given channel in map_memory
--------------------------------------------------------------------
FUNCTION seek (CONSTANT mapmemblk : IN map_mem_blks;
               CONSTANT cell_head : IN cell_head_type)
               RETURN integer
IS
  VARIABLE temp_result1 : cell_head_type;
  VARIABLE temp_result2 : cell_head_type;
  VARIABLE temp_match   : cell_head_type;
  VARIABLE addr_index   : integer :=0;
BEGIN

  for addr_index in 0 to MapSize loop
      temp_result1 := mapmemblk(addr_index).channelidin AND cell_head;
      temp_result2 := (NOT mapmemblk(addr_index).channelidin)
                       AND (NOT cell_head);
      temp_match   := temp_result1 OR temp_result2;
```

```
      if temp_match="11111111111111111111111111111111" then
         return addr_index;
      end if;
  end loop;
  return -1;
END seek;



--------------------------------------------------------------------------
--Function that check whether map memory is empty
--------------------------------------------------------------------------
FUNCTION isMapEmpty RETURN boolean
IS
  VARIABLE bool : boolean := true;
BEGIN
  return (bool);
END isMapEmpty;



--------------------------------------------------------------------------
--Function that check whether map memory is full
--------------------------------------------------------------------------
FUNCTION isMapFull RETURN boolean
IS
  VARIABLE bool : boolean := false;
BEGIN
  return (bool);
END isMapFull;


END map_pkg01;



--------------------------------------------------------------------------------
--   File name :  mapent.vhd
--   Title     :  [ATM Map Memory]
--   Module    :  [The Map Memory Entity]
--   Purpose   :  [Define the entity of the map memory]

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;
USE work.map_pkg01.all;
USE work.mem_pkg.all;
USE work.cq_link_pkg.all;

ENTITY MapMemory IS
  GENERIC (Tpd           : Time  := unit_delay);
  PORT( read             : in    std_ulogic;
        write            : in    std_ulogic;
        head_bus         : inout bus_cell_head_type bus;
        clk1             : in    std_ulogic;
```



```
        match            : out   std_ulogic;
        reset            : in    std_ulogic;
        ready            : in    std_ulogic;
        a_bus            : inout bus_cell_mem_addr bus;
        setupcell        : in    std_ulogic;
        releasecell      : in    std_ulogic;
        itv              : in    std_ulogic_vector(9 DOWNTO 0);
        mask             : in    std_ulogic_vector(9 DOWNTO 0);
        count            : in    std_ulogic_vector(9 DOWNTO 0);
        en_port_no       : in    oe_type;
        en_outport_no    : in    oe_type;
        outgoing         : in    std_ulogic;
        channeltype      : out   std_ulogic;
        chandeliersize   : out   chandelier_size;
        cq_map_bus       : inout bus_cq_map_bus bus);
END MapMemory;


--------------------------------------------------------------------------------
--   File name :  maparc.vhd
--   Title     :  [ATM Map Memory]
--   Module    :  [Behaviour Description]
--   Purpose   :  [Implement the functions of the map memory module]
--------------------------------------------------------------------------------
--   Modification History :
--          Date           Author          Revision        Comments
-- Sun July  4   1997      Bo Yan          Rev1.0.1        Creation
-- Tue Jan   6   1998      Bo Yan          Rev1.0.4        process added
-- Sat Jan   24  1998      Bo Yan          Rev1.0.5        Modified
--------------------------------------------------------------------------------

USE work.map_pkg01.all;

ARCHITECTURE behaviour of MapMemory is


BEGIN                                   -- begin the behaviour of Map_SPAMM

PROCESS

  ------------------------------------------------------------
  -- declare variables for map memory module
  ------------------------------------------------------------

  VARIABLE dvalue      : cell_head_type :="00000000000000000000000000000000";
                                        --channel number
  VARIABLE maplist     : map_mem_blks;
  VARIABLE outport     : integer range 0 to BufferSize;
  VARIABLE key         : cell_head_type;
  VARIABLE mapaddr     : integer;
  VARIABLE chandelier  : chandelier_type;
  VARIABLE freecell    : integer range 0 to CellMemSize;
  VARIABLE celladdr    : cell_mem_addr;
  VARIABLE celltail    : cell_mem_addr;
  VARIABLE index       : integer := 0;
  VARIABLE nextentry   : integer := 0;

  VARIABLE curroutport     : oe_type;
  VARIABLE currmapentry    : integer range 0 to MapSize;
  VARIABLE cellheadaddr    : cell_mem_addr;
```

```
VARIABLE newheadaddr     : cell_mem_addr;
VARIABLE chandeliersizevalue : chandelier_size;
VARIABLE trigger  : std_ulogic := '0';


-------------------------------------------------------------
-- initialise chandelier
-------------------------------------------------------------
PROCEDURE initchandelier IS
BEGIN
  for index in 0 to BufferSize loop
      chandelier(index).size := 0;
      chandelier(index).lastentry := 0;
      chandeliersizevalue(index) := '0';
  end loop;

END initchandelier;


-------------------------------------------------------------------------
-- matching the counter number with itv and mask value. If the number is
-- matched, the trigger signal was set to '1'
-------------------------------------------------------------------------

PROCEDURE triggering(CONSTANT maplist1: IN  map_mem_blks;
                     CONSTANT port1   : IN  integer;
                     VARIABLE entry   : OUT integer;
                     VARIABLE trigger1: OUT std_ulogic) IS
  VARIABLE counter : std_ulogic_vector(9 DOWNTO 0);
  VARIABLE temp    : std_ulogic_vector(9 DOWNTO 0);

BEGIN

  counter := count;                 -- get counter value from count port
  for index in 0 to MapSize loop
      if (maplist1(index).channeltype="00" or
          maplist1(index).channeltype="01") AND
         (maplist1(index).outputport=port1) then
         temp := NOT (maplist1(index).itv  XOR counter);
         temp := maplist1(index).mask OR  temp;
         if temp="1111111111" then  -- ITV, Mask and counter value was matched

            if maplist1(index).emptyqueue=false then
                                --current cell queue not empty
               trigger1 := '1';
               entry := index;
            else                --current cell queue is empty
               trigger1 := '0';
            end if;
            return;
         end if;
      end if;
  end loop;
  trigger1 := '0';
  return;
END triggering;



-------------------------------------------------------------
```



```
-- map a coming cell with channel header in map memory
-------------------------------------------------------------

PROCEDURE acceptcell IS
BEGIN
   match <= '0' after Tpd;-- output match signal
   celladdr := a_bus;   -- get cell memory address from address bus
   freecell := BitVector_To_Int(To_BitVector(celladdr));
   key      := head_bus;
            -- get coming cell head(channel number) from input buffer
   mapaddr  := seek(maplist, key);
            -- map coming cell head(channel number) with map list

   if mapaddr<0 or mapaddr>MapSize then
            -- check whether address of map memory is over scope
      assert false
      report "Map Memory is Full !"
      severity error;
   end if;

   if maplist(mapaddr).channeltype="00" OR maplist(mapaddr).channeltype="01"
      then
                           -- check whether coming cell is CBR/VBR channel
      channeltype <= '1'  after Tpd;
                           -- channel type is CBR/VBR channel
      maplist(mapaddr).emptyqueue    := false;
                           -- assign cell empty queue value
      maplist(mapaddr).cellqueuehead := celladdr;
                           -- assign cell queue head pointer
      maplist(mapaddr).cellqueuetail := celladdr;
                           -- assign cell queue tail pointer
      maplist(mapaddr).queuelength   := 1;
                           -- assign cell queue length
      maplist(mapaddr).nextentry := mapaddr;
                           -- assign next map entry
      celltail := celladdr;
      cq_map_bus <= celltail;
           -- output current cell queue tail address to cqlink
      chandeliersize <= chandeliersizevalue after Tpd;
                           -- output chandelier value to output server
      wait until clk1='0';
      cq_map_bus <= null after Tpd;
      a_bus      <= null after Tpd;
      head_bus   <= null after Tpd;
      return;
   else
                           -- coming cell is ABR/UBR channel type
      channeltype <= '0' after Tpd;
                           -- channel type is ABR/UBR
      outport := maplist(mapaddr).outputport;
                           -- get coming cell output port number

      if maplist(mapaddr).emptyqueue=true then
                           -- check whether cell queue is empty
         if chandelier(outport).size=0 then
            -- check whether chandelier of current output port exist
            maplist(mapaddr).emptyqueue    := false;
            maplist(mapaddr).cellqueuehead := celladdr;
            maplist(mapaddr).cellqueuetail := celladdr;
            maplist(mapaddr).queuelength   := 1;
```

```
            maplist(mapaddr).nextentry     := mapaddr;
            chandelier(outport).lastentry  := mapaddr;
              -- assign current map memory entry as last chandelier

            chandelier(outport).size       := chandelier(outport).size+1;
                           -- increase chandelier size
            chandeliersizevalue(outport)   := '1';
         else                -- chandelier of current output port exist
            maplist(mapaddr).emptyqueue    := false;
            maplist(mapaddr).cellqueuehead := celladdr;
            maplist(mapaddr).cellqueuetail := celladdr;
            maplist(mapaddr).queuelength   := 1;

            --
            -- insert a cell queue into channdelier of indecated output port
            --
            maplist(mapaddr).nextentry     := maplist(chandelier
                                              (outport).lastentry).nextentry;
            maplist(chandelier(outport).lastentry).nextentry := mapaddr;
            chandelier(outport).lastentry  := mapaddr;

            chandelier(outport).size       := chandelier(outport).size+1;
                                           -- increase chandelier size
            chandeliersizevalue(outport)   := '1';
         end if;                           -- end check chandelier

         celltail := celladdr;
      else                                 -- current cell queue isn't empty
         celltail := maplist(mapaddr).cellqueuetail;
         maplist(mapaddr).cellqueuetail := celladdr;
         maplist(mapaddr).queuelength   := maplist(mapaddr).queuelength + 1;

      end if;                              -- end check cell queue

      cq_map_bus <= celltail after Tpd;
           -- output the cell queue tail pointer to cqlink
      chandeliersize <= chandeliersizevalue after Tpd;
           -- output chandelier value to output server

      wait until clk1='0';
      cq_map_bus <= null after Tpd;
      a_bus      <= null after Tpd;
      head_bus   <= null after Tpd;
   end if;                               -- end CBR/VBR channel check
END acceptcell;



-------------------------------------------------------------
-- set the new channel information into the map memory
-------------------------------------------------------------

PROCEDURE setupchannel IS
  CONSTANT channelconst : cell_head_type := "00000000000000000001111111111";
  VARIABLE inportno     : oe_type;
  VARIABLE inportint    : integer;
  VARIABLE outportno    : oe_type;
  VARIABLE outportint   : integer;
  VARIABLE channeltype  : std_ulogic_vector(1 DOWNTO 0);
  VARIABLE newchannel   : cell_head_type;
```



```
  VARIABLE channelin    : cell_head_type;
  VARIABLE channelout   : cell_head_type;
BEGIN
  wait for Tpd;
  newchannel  := head_bus;
  channeltype := newchannel(21 DOWNTO 20);          -- get channel type
  inportno    := newchannel(17 DOWNTO 14);          -- get input port number
  inportint   := BitVector_To_Int(To_BitVector(inportno));
  outportno   := newchannel(13 DOWNTO 10);          -- get output port number
  outportint  := BitVector_To_Int(To_BitVector(outportno));
  channelin   := newchannel AND channelconst;       -- get input channel id
  channelout  := newchannel AND channelconst;       -- get output channel id
  for index in 0 to MapSize loop
      if maplist(index).entryfree=true then
         maplist(index).entryfree     := false;     -- set current map entry
         maplist(index).channelidin   := channelin; -- assign input channelid
         maplist(index).inputport     := inportint; -- assign input port no.
         maplist(index).channelidout  := channelout;-- assign output channelid
         maplist(index).outputport    := outportint;-- assign output port no.
         maplist(index).itv           := itv;       -- assign itv value
         maplist(index).mask          := mask;      -- assign mask value
         maplist(index).queuelength   := 0;
         maplist(index).nextentry     := 0;
         maplist(index).channeltype   := channeltype;
         return;
      end if;
  end loop;
END setupchannel;



-------------------------------------------------------------
--release a no longer used  channel header at map memory
-------------------------------------------------------------

PROCEDURE releasechannel IS
  CONSTANT channelconst : cell_head_type := "00000000000000000000000000000";
  VARIABLE inportno     : oe_type;
  VARIABLE inportint    : integer;
  VARIABLE outportno    : oe_type;
  VARIABLE outportint   : integer;
  VARIABLE channeltype  : std_ulogic_vector(1 DOWNTO 0);
  VARIABLE newchannel   : cell_head_type;
  VARIABLE channelin    : cell_head_type;
  VARIABLE channelout   : cell_head_type;
BEGIN
  wait for Tpd;
  newchannel := head_bus;
  mapaddr    := seek(maplist, newchannel);-- seek released channel in map list

  if mapaddr<0 or mapaddr>MapSize then    -- error checking
     assert false
     report "No Released channel in Map List !"
     severity NOTE;
  else
     maplist(mapaddr).entryfree   := true;-- set current map entry empty
     maplist(index).channelidin   := channelconst AND channelin;
                                        -- assign input channel id
     maplist(index).channelidout  := channelconst AND channelout;
                                        -- assign output channel id
```

```
      maplist(index).queuelength    := 0;
      maplist(index).nextentry      := 0;
   end if;
END releasechannel;



------------------------------------------------------------
--output a cell eithehr from trigger mechanism or chandelier
------------------------------------------------------------

PROCEDURE outgoingcell IS
BEGIN

   wait for Tpd;
   newheadaddr := cq_map_bus;          -- get new cell queue head from cqlink

   --
   --update map memory
   --

   if currmapentry<0 or currmapentry>MapSize then     -- error checking
      assert false
      report "Current map entry is over scope!"
      severity NOTE;
   end if;

   if maplist(currmapentry).queuelength <=0 or
      maplist(currmapentry).queuelength>MapSize then
                                                      -- error checking
      assert false
      report "current queuelength is over scope !"
      severity NOTE;
   end if;

   if maplist(currmapentry).outputport =0 then        -- error checking
      assert false
      report "current output port is = 0 "
      severity NOTE;
   end if;


   maplist(currmapentry).queuelength := maplist(currmapentry).queuelength-1;
                                       -- decrease cell queue length
   if maplist(currmapentry).queuelength<=0 then-- cell queue is empty
      maplist(currmapentry).emptyqueue := true;
                           -- assign true to empty queue field
      maplist(chandelier(outport).lastentry).nextentry
      := maplist(currmapentry).nextentry;
                           -- remove the channel header from chandelier
      chandelier(outport).size := chandelier(outport).size -1;
                           -- decrease the chandelier size
      if chandelier(outport).size=0 then
                           -- check whether chandelier is empty
         chandeliersizevalue(outport) := '0';
                           -- assign '0' to current chandelier
      end if;
   else
      maplist(currmapentry).cellqueuehead := newheadaddr;
                           -- update new cell head address
```



```
         chandelier(outport).lastentry := currmapentry;
                              -- update chandelier address
      end if;

      wait until clk1='1';
      a_bus    <= null after Tpd;
      head_bus <= null after Tpd;

   END outgoingcell;          -- end outgoingcell procedure


begin                         -- process begin
   --
   --check for reset active
   --
   wait until reset='1' or reset='0';
   if reset ='1' then
      initialise(maplist);    -- initialise map memory
      initchandelier;         -- initialise chandelier
      a_bus       <= null after Tpd;
      head_bus    <= null after Tpd;
      cq_map_bus <= null after Tpd;
      chandeliersize <= "ZZZZZZZZZZZZZZZZ" after Tpd;
      wait until reset='0';
   end if;

   loop                       -- execute following statement until reset='1'
   if reset='0' then
      wait until write='1' or setupcell='1' or releasecell='1' or clk1='0';
                              -- Wait for write or read sinal coming
      chandeliersize <= "ZZZZZZZZZZZZZZZZ" after Tpd;
      if write='1' then
         acceptcell;          -- coming cell is normal cell
      end if;
      if setupcell='1' and releasecell='0' then
         setupchannel;        -- coming cell is setup cell
         wait until clk1='0';
      end if;
      if setupcell='1' and releasecell='1' then
         releasechannel;      -- coming cell is release cell
         wait until clk1='0';
      end if;
      channeltype <= 'Z' after Tpd;    -- initialise channel type

      if clk1='0' then
         trigger := '0';      -- initialise trigger value
         chandeliersize <= chandeliersizevalue after Tpd;
         cq_map_bus <= null after Tpd;

         curroutport := en_outport_no; -- get current outport number

         if curroutport/="ZZZZ" then
            outport := BitVector_To_Int(To_BitVector(curroutport));
                              -- convert current outport number into integer
            triggering(maplist, outport, currmapentry, trigger);
                              -- check whether CBR/VBR channel is triggered
         end if;

         if trigger='1' then  -- the one of CBR/VBR channels was triggered
```

```
            match <= '1' after Tpd;
            channeltype <= '1'after Tpd;
                              -- output channel type is CBR/VBR channel
            a_bus <= maplist(currmapentry).cellqueuehead after Tpd;
                              -- output cell memory address onto a_bus
            head_bus <= maplist(currmapentry).channelidout;
                              -- output re-assembly header of cell
            wait until clk1='1';
            maplist(currmapentry).emptyqueue  := true;
            maplist(currmapentry).queuelength := 0;
            match    <= '0';
            a_bus    <= null after Tpd;
            head_bus <= null after Tpd;
            trigger  := '0';
         else                         -- the channel was selected by chandelier
            match <= '0';
            wait until outgoing='1' OR clk1='1';

            if outgoing='1' then -- outgoing channel was choosed by chandelier
               curroutport := en_outport_no; -- get new output port number
               outport := BitVector_To_Int(To_BitVector(curroutport));
               currmapentry := maplist(chandelier(outport).lastentry).nextentry;
                              -- get current map memory entry from chandelier
               cellheadaddr := maplist(currmapentry).cellqueuehead;
                              -- get cell address from current channel header
               a_bus    <= cellheadaddr after Tpd;
                                          -- output current cell address
               head_bus <= maplist(currmapentry).channelidout after Tpd;
                                          -- output cell header
               wait until read='1' or clk1='1';

               if read='1' then           -- cell memory is ready to read cell
                  channeltype <= '0' after Tpd;-- output channel type as ABR/UBR
                  outgoingcell;           -- read a cell from cell memory
               else                       -- cell memory is not ready
                  a_bus    <= null after Tpd;
                  head_bus <= null after Tpd;
               end if;                    -- end read='1' check
            else
               a_bus    <= null after Tpd;
               head_bus <= null after Tpd;
            end if;                       -- end outgoing='1'
         end if;                          -- end trigger='1'
      channeltype <= 'Z' after Tpd;       -- initialise channel type
      end if;                             -- end clk1='0'
   else
      exit;
   end if;                                -- end reset='0'
   end loop;                              -- end loop

END process;

END behaviour;                        -- end map memory behaviour description


--------------------------------------------------------------------------------
--    File name :  ouporent.vhd
--    Title     :  [ATM Output Port 0# ]
--    Module    :  [The Output Port 0# Entity]
--    Purpose   :  [Define the entity of the output port 0#]
```



```
--------------------------------------------------------------------------------
--    Modification History :
--          Date              Author          Revision        Comments
--    Tur Oct. 10  1997       Bo Yan          Rev1.0.0        Creation
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;
USE work.mem_pkg.all;

ENTITY output_buffer00 is
  GENERIC( Tpd        : Time := unit_delay);
  PORT   ( oe         : in  oe_type;
           reset      : in  std_ulogic;
           ready      : in  std_ulogic;
           d_bus      : in  bus_cell_mem_data bus;
           head_bus   : in  cell_head_type;
           freeport00 : out std_ulogic );
END output_buffer00;



--------------------------------------------------------------------------------
--    File name :  ouporarc.vhd
--    Title     :  [ATM Output Port 0#]
--    Module    :  [Behaviour Description of the output port 0#]
--    Purpose   :  [Implement the functions of the output port 0# module]

ARCHITECTURE behaviour of output_buffer00 is

BEGIN

  PROCESS
  variable portfree_signal : std_ulogic :='1';
  variable oe_value    : oe_type := "0000";
  variable data        : cell_mem_data;
  variable head        : cell_head_type;
  begin

    --
    --check for reset active
    --
    wait until reset='0' or reset='1';
    if reset='1' then
       freeport00  <= portfree_signal;
       wait until reset='0';

    end if;

    --
    --set port free signal and input cell from cell memory buffer
    --
    loop
    if (reset='0') then
```

```
        freeport00 <= portfree_signal after Tpd;
        wait until oe=oe_value;     -- wait for priority circuit check full signa
l
        wait until ready='1';
        data := d_bus;
        head := head_bus;
        freeport00 <= '0' after Tpd;
        wait for 6*Tpd;
    else                            -- reset/='0'
        exit;
    end if;                         -- end reset='0'
    end loop;
  end process;
END behaviour;


-------------------------------------------------------------------------------
--    File name :  ouserent.vhd
--    Title     :  [ATM Output Round Robin Server]
--    Module    :  [The Output Server Entity]
--    Purpose   :  [Define the entity of the output server]
-------------------------------------------------------------------------------
--    Modification History :
--          Date              Author          Revision        Comments
--    Tur Oct. 10 1997        Bo Yan          Rev1.0.0        Creation
-------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

LIBRARY std;
USE std.textio.ALL;

USE work.atm_types.all;

ENTITY output_encoder is
  GENERIC( Tpd         : Time := unit_delay);
  PORT   ( clk1        : in   std_ulogic;
           clk2        : in   std_ulogic;
           read        : in   std_ulogic;
           write       : in   std_ulogic;
           reset       : in   std_ulogic;
           match       : in   std_ulogic;
           freeport00  : in   std_ulogic;
           freeport01  : in   std_ulogic;
           freeport02  : in   std_ulogic;
           freeport03  : in   std_ulogic;
           freeport04  : in   std_ulogic;
           freeport05  : in   std_ulogic;
           freeport06  : in   std_ulogic;
           freeport07  : in   std_ulogic;
           freeport08  : in   std_ulogic;
           freeport09  : in   std_ulogic;
           freeport10  : in   std_ulogic;
           freeport11  : in   std_ulogic;
           freeport12  : in   std_ulogic;
           freeport13  : in   std_ulogic;
           freeport14  : in   std_ulogic;
           freeport15  : in   std_ulogic;
```



```
           outgoing    : out  std_ulogic;
           chandeliersize    : in  chandelier_size;
           en_outport_no     : out oe_type);
END output_encoder;


-------------------------------------------------------------------------------
--    File name :  ouserarc.vhd
--    Title     :  [ATM Output Round Robin Server]
--    Module    :  [Behaviour Description of the Output Server]
--    Purpose   :  [Implement the functions of the output server]

Architecture behaviour of output_encoder is
  TYPE boolean is(false, true);

BEGIN
  PROCESS

  -----------------------------------------------------------------------------
  -- declare variables
  -----------------------------------------------------------------------------
  VARIABLE currentport    : oe_type;
  VARIABLE encoderport    : oe_type;
  VARIABLE index          : integer := 0;
  VARIABLE freeportvalue  : buf_full_type;
  VARIABLE chandelier     : chandelier_size;
  VARIABLE outportfull    : boolean;
  VARIABLE outgoingcell   : boolean;


  -----------------------------------------------------------------------------
  -- This procedure is used to initialise the output round robin server
  -----------------------------------------------------------------------------
  PROCEDURE initialise(VARIABLE freeportblks : INOUT buf_full_type;
                       VARIABLE portnumber   : INOUT oe_type;
                       VARIABLE bufferfull   : INOUT boolean) IS
  BEGIN
    for index in 0 to BufferSize loop
        freeportblks(index) := '0';
    end loop;
    portnumber := "0000";
    bufferfull := true;
  END initialise;


  -----------------------------------------------------------------------------
  -- This procedure is used to receive the free signal from each output port
  -----------------------------------------------------------------------------
  PROCEDURE getsignal(VARIABLE freeportblks : INOUT buf_full_type) IS
  BEGIN
    freeportblks(0)   := freeport00;
    freeportblks(1)   := freeport01;
    freeportblks(2)   := freeport02;
    freeportblks(3)   := freeport03;
    freeportblks(4)   := freeport04;
    freeportblks(5)   := freeport05;
    freeportblks(6)   := freeport06;
    freeportblks(7)   := freeport07;
    freeportblks(8)   := freeport08;
    freeportblks(9)   := freeport09;
```

```
   freeportblks(10)  := freeport10;
   freeportblks(11)  := freeport11;
   freeportblks(12)  := freeport12;
   freeportblks(13)  := freeport13;
   freeportblks(14)  := freeport14;
   freeportblks(15)  := freeport15;
END getsignal;


-------------------------------------------------------------------------
-- This procedure is used to count the current output port number
-------------------------------------------------------------------------
PROCEDURE getcurrentport(VARIABLE freeportblks : IN  buf_full_type;
                         VARIABLE currport     : IN  oe_type;
                         VARIABLE outputfull   : OUT boolean) IS
BEGIN

  index := BitVector_to_Int(To_BitVector(currport));
                                        -- convert current output port number
  if index<0 or index>15 then           -- error checking
     assert false
     report "The input buffer number is out of scope !"
     severity error;
  end if;

  if freeportblks(index)='1' then       -- check current output port
     outputfull := false;               -- current output port is free
  else
     outputfull := true;                -- current output port is not free
  end if;

END getcurrentport;


-------------------------------------------------------------------------
-- This procedure is used to check whether the current output port is free
-- The procedure is used here for super mode
-------------------------------------------------------------------------
PROCEDURE checksignal (VARIABLE freeportblks : IN  buf_full_type;
                       VARIABLE chandeliers  : IN  chandelier_size;
                       VARIABLE currport     : IN  oe_type;
                       VARIABLE portnumber   : OUT oe_type;
                       VARIABLE cellready    : OUT boolean) IS
VARIABLE temp_index : integer;
BEGIN
  index := BitVector_to_Int(To_BitVector(currport));
                                        -- convert current output port number
  if index<0 or index>15 then           -- error checking
     assert false
     report "The input buffer number is out of scope !"
     severity error;
  end if;

  --
  -- check both current free port signal and chandelier size
  --
  if freeportblks(index)='1' and chandeliers(index)='1' then
     portnumber  := currport;
     cellready   := true;               -- cell is waitting for outgoing
     return;
```



```
  end if;

  temp_index := index;
  index := index + 1;
  if index>15 then
     index := 0;
  end if;

  while index /= temp_index loop        -- check reset of outport free signal
        if freeportblks(index)='1' and chandeliers(index)='1' then
                                        -- check given input port full signal
           portnumber := To_StdULogicVector(Int_To_BitVector(index, 4));
           cellready  := true;          -- cell is waitting for outgoing
           return;
        end if;
        index := index +1;
        if index>15 then
           index := 0;
        end if;
  end loop;
  cellready := false;  -- all of output buffer full or no cell for this port
  return;
END checksignal;


-------------------------------------------------------------------------
-- This procedure is used to check whether the current output port is free
-- The procedure is used here for normal mode
-------------------------------------------------------------------------
PROCEDURE checkchandelier (VARIABLE freeportblks : IN  buf_full_type;
                           VARIABLE chandeliers  : IN  chandelier_size;
                           VARIABLE currport     : IN  oe_type;
                           VARIABLE portnumber   : OUT oe_type;
                           VARIABLE cellready    : OUT boolean) IS
VARIABLE temp_index : integer;
BEGIN
  index := BitVector_to_Int(To_BitVector(currport));
                                        -- convert current output port number
  if index<0 or index>15 then           -- error checking
     assert false
     report "The input buffer number is out of scope !"
     severity error;
  end if;


  --
  -- check both current free port signal and chandelier size
  --
  if freeportblks(index)='1' AND chandeliers(index)='1' then
     portnumber  := currport;
     cellready   := true;               -- chandelier have cell for outgoing

  else
     cellready   := false;              -- chandelier is no cell for outgoing
     portnumber  := currport;
  end if;

END checkchandelier;
```

```
BEGIN                                   -- Process begin

  --
  --check for reset active
  --
  wait until reset='0' or reset='1';
  if reset='1' then                     -- reset state
     initialise(freeportvalue, currentport, outgoingcell);
     outgoing <= 'Z' after Tpd;
     wait until reset='0';
  end if;


  loop
  if reset='0' then
     if clk1='1' then                   -- accept cell stage
        en_outport_no     <= "ZZZZ";

        getsignal(freeportvalue);       -- read free signal of output port
        getcurrentport(freeportvalue, currentport, outportfull);
                                -- get currentport number and free signal
        if outportfull=false then
           en_outport_no <= currentport; -- output current port number
        else
           en_outport_no <= "ZZZZ";
        end if;
        wait until clk1='0';
     end if;

     if clk1='0' then                   -- outgoing cell stage
        chandelier := chandeliersize;
                                -- get chandelier information from port

        checkchandelier(freeportvalue, chandelier, currentport, encoderport,
                        outgoingcell);
                                -- check chandelier of current output port
        if outgoingcell=true then
           outgoing <= '1'  after Tpd;   -- output accept cell signal as true
           wait until clk1='1';
           outgoing <= '0'  ;
        else
           outgoing      <= '0'  after Tpd;
                                -- output outgoing cell signal as false
           wait until clk1='1';
        end if;                 -- end match='1' check

        if currentport /="0011" then
           currentport :=std_ulogic_vector(unsigned(currentport) + "0001");
        else
           currentport :="0000";
        end if;
     end if;                                        -- end clk1='0' check
  else                                              -- reset /='0'
     exit;
  end if;                                           -- end reset='0'
  end loop;
END process;
END behaviour;
```



```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--                                                                            --
--                          The Program END                                   --
--                                                                            --
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```