Efficient Memory Management for high-speed ATM networks

Panagiotis Karakonstantis{*karakon@csi.forth.gr*}

Revision : 1.8

Contents

1 Introduction **7**

2 ATM system model and functionality **9**
2.1 Operation and applications of the model 10
2.2 ATM system functions 12
2.2.1 Segmentation and Reassembly (SAR) 12
2.2.2 Rate-based flow control 12
2.2.3 Credit-based flow control 13
2.2.4 Selective Discard 14

3 Queue Management Architecture **16**
3.1 Definition of the basic data structures 18
3.2 Design Tradeoffs 19

4 Hardware Implementation **22**
4.1 The MuQPro I implementation 22
4.1.1 Interface 23
4.1.2 Instruction set 25
4.1.3 Internal Architecture 25
4.1.4 SRAM and Master Device requirements 28
4.1.5 Merging two instances of QM together 29
4.1.6 The instruction is enqueue, dequeue, top or init (case A) 31
4.1.7 The instruction is retfree (case B) 31
4.1.8 The instruction is getfree, read or write (case C) 31
4.1.9 Testing 32
4.1.10 Cost and performance 33
4.2 The "fully scalable" implementation 33
4.3 The "fully parallel" implementation 33

5 Software implementation of Queue Management **35**
5.1 Programming environment 35
5.2 The EnQ operation with canonical input 35
5.3 Mixed EnQ/DeQ operations with canonical input 36
5.4 The EnQ operation with random input 36
5.5 The SAR model 37
5.5.1 The packet processing routine 38
5.5.2 The iterative process 40
5.5.3 The parameters 40
5.5.4 Measurements 41
5.5.5 Analysis 45

6 Comparisons **47**

A Functionality and timing diagrams for each instruction **49**
A.1 The enqueue instruction 49
A.2 The dequeue instruction 54
A.3 The getfree instruction 59
A.4 The retfree instruction 60

A.5 Initialization 65
A.6 Read instruction 65
A.7 Write instruction 65
A.8 Top instruction 70
A.9 CreateQ 70
A.10 DeleteQ 70

B Datapath of MUQPRO I QM module **70**

C The "fully scalable" implementation datapath **72**

D The "fully scalable" implementation control path **73**

E The "fully parallel" implementation data and control path **75**

List of Figures

1 OSI layered architecture and AAL-5 and ATM processing . . . 9
2 ATM system model . . . 10
3 Using FIFOs to implement Credit-based flow control . . . 14
4 Using FIFOs to implement Selective Discard . . . 15
5 Queue Manager's generic architecture . . . 16
6 Queue Manager's basic data structures . . . 18
7 MUQPRO I system architecture . . . 23
8 Queue Manager block diagram . . . 24
9 Queue Manager pseudo-pipeline . . . 26
10 Queue Manager block diagram . . . 27
11 Issuing instructions right after an enqueue or dequeue instruction . . . 28
12 Issuing instructions right after a getfree or retfree instruction . . . 29
13 Making 2 sets of queues . . . 30
14 Arbiter finite state machine . . . 32
15 Enqueue serial operations . . . 36
16 Mixed serial operations . . . 37
17 Random Enqueue operations with 4th level optimization . . . 37
18 The SAR model . . . 39
19 Measurements for 1 and 10 cell packets . . . 42
20 Measurements for 50 and 100 cell packets . . . 43
21 Measurements for 200 and 400 cell packets . . . 43
22 Measurements for 800 and 1200 cell packets . . . 43
23 Measurements for 1-1200 cell packets and ULTRA measurements for 1 cell packets 44
24 Issuing instructions right after an enqueue or dequeue instruction . . . 49
25 Issuing instructions right after a getfree or retfree instruction . . . 50
26 Schedule for the EnQ command . . . 51
27 Initial state of the queues, enqueue slot 0 in queue 0, enqueue slot 1 in queue 0 . 52
28 Enqueue instruction on a non-empty queue . . . 52
29 Enqueue instruction on an empty queue . . . 53
30 Initial state of the queues, dequeue from queue 0, dequeue from queue 0 . . . 55
31 Schedule for the DeQ instruction . . . 56
32 Dequeue instruction on an empty queue . . . 57
33 Dequeue instruction on a queue with exactly one element . . . 57
34 Dequeue instruction on a queue with more than one elements . . . 58
35 Initial state of the queues, getfree, getfree . . . 59
36 Schedule for the GetFree instruction . . . 60
37 GetFree instruction on an empty queue . . . 61
38 GetFree instruction on a queue with exactly one element . . . 62
39 GetFree instruction on a queue with more than one elements . . . 63
40 Initial state of the queues, retfree slot , retfree slot . . . 63
41 Schedule for the RetFree command . . . 64
42 Retfree instruction on a non-empty queue . . . 64
43 Retfree instruction on an empty queue . . . 65
44 Initialization example for 4 queues and 8 slots . . . 66
45 Timing diagram I of Init instruction . . . 67
46 Timing diagram II of Init instruction . . . 67
47 Timing diagram III of Init instruction . . . 68

48 Timing diagram of Read instruction 68
49 Timing diagram of Write instruction 69
50 Timing diagram of Top instruction 70
51 The Datapath that implements these operations 71
52 The Datapath of the "fully scalable" implementation 72
53 Schedule for the EnQ command 73
54 Schedule for the GetFree command 73
55 Schedule for the DeQ command 74
56 Schedule for the RetFree command 74
57 The Datapath of the "fully parallel" implementation 75
58 The FSM of the "fully parallel" implementation 76

Abstract

Asynchronous Transfer Mode (ATM) is a connection-oriented networking technology that supports transfer of data, video and voice. The ability to multiplex a large number of connections over a single physical link places high requirements for processing and memory management in ATM systems. The problem becomes more acute as ATM systems scale to both higher speeds and increased number of links. In this work, we analyze the memory management requirements imposed by the main ATM functions (segmentation and reassembly, flow control and selective discard) and identify a core set of operations whose fast execution leads to efficient memory subsystems for generic ATM systems (switches or adapters). We propose hardware architectures that implement this set of operations as well as a software implementation suitable for embedded systems. Finally, we present performance and cost measurements of the various implementations so that one can choose the implementation most suitable for one's target system.

1 Introduction

Asynchronous Transfer Mode (ATM) is a connection-oriented, cell-based networking technology that supports transfer of data, video and voice. Many virtual connections that specify paths between source and destination network nodes may pass over the same physical link. On the other hand ATM scales to very high speeds resulting in strict timing requirements in protocol processing and buffering. The problem becomes more acute as the number of physical links increases. Furthermore, this relatively new technology offers per connection Quality of Service (QoS) which means fixed delay for cell delivery and pre-defined Cell Loss Probability (CLP) as have been agreed between the network manager and the client (user). All these constraints compel ATM systems to incorporate buffers.

Buffering in ATM has a special structure imposed by its operations; Cells are associated in some way forming logical queues. This association results either from structural relations (e.g cells that belong to the same packet) or from similar operational requirements (e.g. cells heading for the same resource) and requires many separate logical queues.

All ATM systems can be modeled in a simple way without loosing any important information. Such a model can depict both adapter and switch nodes and shows the similarities between the seemingly different network components. In this model the signaling and higher-level protocol processing is executed by a Processing Element (PE) while the lower level protocols are executed by a Network I/F component in hardware. Cells are transfered between the PE and the Network I/F(s) and form logical queues inside the buffers. Buffering in such a model can be separated from the protocol processing (hardware or software) and form a distinct unit. This separation is not only for presentation purposes but it also enables parallelism between buffering and processing. Buffering can be further divided into 2 parts; one plain memory (CBM) where all the cells physically reside; and a Memory Management Unit (MMU) that creates, extents and destroys queues by using pointers to the physical locations of the cells. This approach is scalable because operations on queues can occur without actually interfering with the cell bodies. This model though simple depicts most commercial ATM systems with fair accuracy.

In order to show that functionally queues are sufficient for serving ATM functions, we study a core set of ATM functions that exist in any system. These are Segmentation and Reassembly where cells belonging to the same connection form packets; flow control where cells form queues depending on the available buffer space at the downstream switch node and the destination node; and Selective Discard which is responsible for freeing buffer space by dropping low priority queues of cells when buffer space has reached a certain threshold.

Taking into account all the requirements we propose a Queue Management (QM) architecture where the QM is a slave synchronous device that accepts a minimal set of instructions (Enqueue, Dequeue, GetFree, RetFree, Read, Write Top) and handles queues through the use of pointers. The basic data structures required for implementing queues is a Head/Tail table with one Head/Tail entry for each provided queue, Empty Bits for determining if one queue is empty or not and one pointer field for each buffer (e.g. cell buffer). There are many tradeoffs regarding the placement of these data structures into on-chip or off-chip memories, multiple memories or a single one and connection number to queue number translation, that are extensively discussed.

We present 3 different hardware implementations regarding the tradeoffs; one that is incorporated in a prototype ATM system (MUQPRO I) capable of serving 5 155 Mb/s links the processor and the output scheduler; another that fully utilizes its memory ("fully parallel"); and a last one that exhibits the maximum possible parallelism that exists in Queue Management ("fully parallel"). All implementations have as target technology conservative Field Programmable Gate Arrays (FPGAs) and achieve speed between 20 and 30 MHz requiring one FPGA and an external dedicated SRAM chip. Finally we present a software implementation of

Queue Management suitable for embedded systems and measure its performance in conjunction with the hardware implementations considering the scalability in both higher cell rates and number of queues and the relative cost.

In section 2 we present the ATM system's model in some detail together with the functional requirements of ATM. Section 3 defines the architecture of Queue Management together with the exploited tradeoffs, while sections 4 and 5 describe the hardware and software implementations. Finally we compare the different architectures and emphasize on their strong and weak points in section 6 aiding one to select the most suitable solution for his needs. The appendices A and B describe the instruction set and the timing of the MUQPRO I Queue manager together with the datapath in detail. Appendices C, D and E have diagrams regarding the "fully scalable" and "fully parallel" implementations.

2 ATM system model and functionality

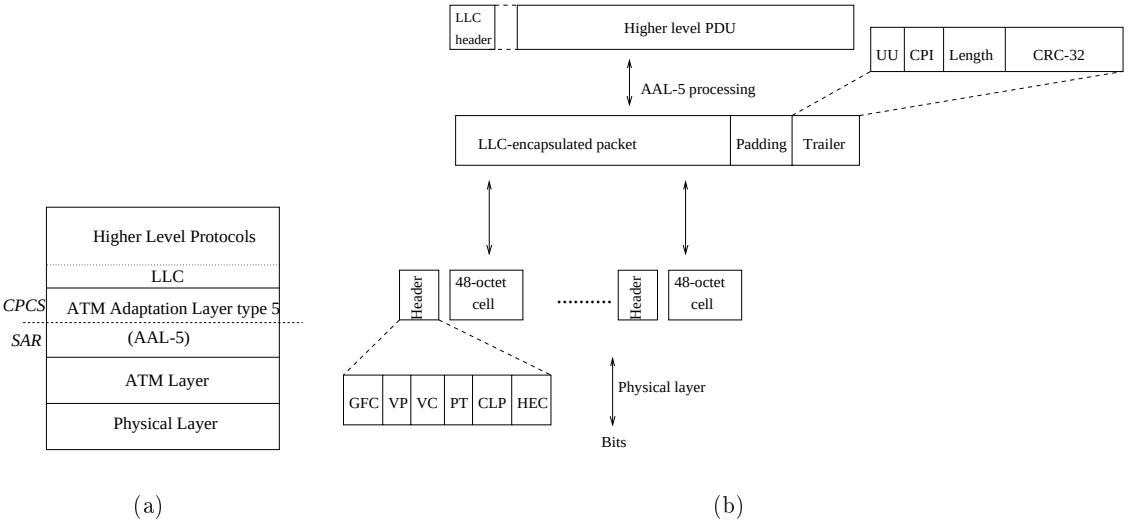

Figure 1: OSI layered architecture and AAL-5 and ATM processing

ATM technology is a cell-based switching technology developed to carry data, video and voice traffic. The basic unit of transfer in ATM is a fixed-size cell and many of them are connected together to make up larger packets as shown in Figure 1a. It covers part of the data link and physical layers in the OSI protocol stack as shown in Figure 1b. The ATM protocol stack is divided into two parts: the Adaptation Layer and the ATM Layer. The adaptation layer is responsible to transform appropriate level PDU's to fixed-size cells and to reassemble packets from received cells. There exist various types (5 overall) of adaptation layers, each appropriate for a different type of traffic (real-time or non-real-time video, audio and data). In this work, we focus on Adaptation Layer 5 (AAL-5), which is the most popular one and which is used not only for data but for delivery of compressed video (MPEG) as well (as defined by ATM Forum standards).

An ATM network is an interconnection of nodes that are either switches or end-system adapters (users). Switches are systems that route ATM cells appropriately in the network. End-system adapters are the network's end-points which collect the cells directed to them, reassemble them to construct the original packets, and deliver the packets to higher layer protocols. Although seemingly different, switches and adapters have many similarities in structure and low level operations. A typical ATM system -switching node or adapter (user node)- consists of a Processing Element (PE) which executes signaling and/or higher layer protocols, memory to store cells (and possibly packets in case of end-system adapters) as well as a number of network and system interfaces, depending on the structure of the system. A typical adapter, which can be used either as an end-system (or communication system) adapter or as a link attachment of a switch is shown in Figure 2.

This adapter architecture that has been introduced in [2], addresses the requirements of high-speed networks with variable-sized packets. The same structure can be used for ATM systems transferring fixed-sized cells as has been presented in [1]. The adapter establishes and manages logical queues of cells -and packets if necessary- where each logical queue represents a collection of logically associated cells. This association can be either structural (e.g. cells of the same packet) or operational (e.g. cells heading for the same resource of the adapter, like the PE or the Network I/F) and differentiates the buffering in such systems from simply placing

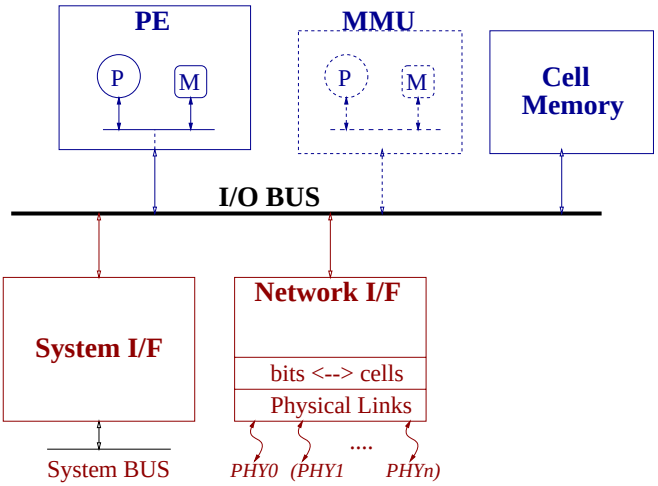

Figure 2: ATM system model

data in a Random Access Memory in contiguous addresses.

One of the most important operations in ATM systems is buffering of cells (and packets, in the case of network endpoints). Buffering in switching is required for temporary storage of cells when conflicts occur for cells that need to exit from the same output link; in adapters, cells of the same connection are assembled together to reconstruct the original packets. Furthermore, buffering is necessary due to the adoption of mechanisms that provide Quality of Service and consequently require smoothing of traffic flows with pre-defined delay and Cell Loss Probability (CLP). The structure and implementation of the buffer in network systems is critical to their performance as has been shown in [1], [16] and [18]. The problem becomes more acute as the SONET hierarchy scales to higher rates for each Network I/F link and the industry tries to fit as many links as possible in one integrated circuit. For example, the cell time - the minimum time between two successive cell arrivals over the same Network I/F - ranges from 2.7 μs for SONET OC-3 (155 Mb/s) links to 0.68 μs for SONET OC-12 (622 Mb/s), 0.34 μs for SONET OC-24 (1.244 Gb/s) and only 0.17 μs for SONET OC-48 (2.48 Gb/s) links. The total number of links in typical commercial switches today is 16 (as shown in [11]) with potential growth as ATM moves to the WANs resulting in system cell time of $\frac{0.17}{16} = 0.1\mu$s which is not implementable today for commercial purposes.

Thus scalability is a key element in current and future ATM buffer implementations and it is the focus of our work that defines the internal structure of buffer subsystems to be incorporated in high-performance ATM systems.

2.1 Operation and applications of the model

The system's model presented in figure 2 depicts in detail the operation of current ATM systems and we will describe how this occurs first for switches and latter for adapters. As mentioned earlier the linking of cells associated in some way is performed through logical queues (or simply queues) implemented in the MMU. The main characteristic that associates cells, is their connection number that describes a virtual path between source and destination. The connection number (VPI/VCI) determines the resource to which cells are heading for and certain VPI/VCIs are reserved for transferring high-priority management information (OAM cells) regarding signaling ATM functions (e.g. connection setup, join of a new node to a multicast group etc) as defined in [21]. All network management protocols are executed by the Processing Element

(PE).

For each incoming (through the Network I/F) cell a decision has to be taken depending on its connection number: If it is a management cell (OAM cell) heading for the PE with high priority then it has to be stored in a queue dedicated to this. Otherwise it has to be stored in a proper queue together with other cells that belong to the same connection waiting for a specific Network I/F to become available. Cut-through can be also considered by directly transferring incoming cells from one Network I/F to another.

At last cells may be placed in queues with different priorities controlled by the PE and specifying the Quality of Service. For example some queues may be dedicated to high priority Constant Bit Rate (CBR) traffic with fixed delay and CLP while some others may be only for Available Bit Rate (ABR) traffic that does not provide any guarantees but it takes up what is left from the other traffic types. Furthermore, the PE generates cells (common and OAM) to depart through the Network I/Fs placing them in proper queues to wait and alters (translates) the connection number of the incoming cell flows as specified by the signaling or routing protocol that the PE executes.

In this model the bodies of the cells physically reside in CBM while the construction of the queues is separately executed by the MMU module that has its own memory and logic. This is accomplished by letting the MMU manipulate pointers to cells instead of the cell bodies themselves. Therefore, multiple copies for transfers between the various system components is avoided and queues can be easily created and destroyed without interfering explicitly with the cells and the packets. The more the Network I/F attached to the system the more critical becomes the speed of the MMU. Other than the MMU approach place buffering in the PE with some elastic buffers at the Network I/F or the opposite. The main disadvantage of this approach is that it is optimized for a specific system and does not allow parallelism between the buffering and the operations executed by the other components - at least at the conceptual level.

The procedure described above applies to adapters as well. If we put away all the Network I/Fs but one our model is transformed into a network adapter model. The routing decisions we had for the switch nodes are replaced by the Segmentation and Reassembly function in the adapter nodes. ATM cells that were grouped in the case of switches on per output Network I/F basis must now be grouped on per connection basis in order for SAR to take place. We can now see the similarity between the two systems: In the switch nodes a cell's connection number (VPI/VCI) determines the destination while in the adapter nodes the connection number determines where to place it. In both cases the VPI/VCI determines where to transfer the cell and this may be an outgoing Network I/F, the PE or a specific buffer in MMU.

The model presented so far though simple represents most of the commercial ATM systems with good accuracy and it has the advantage of depicting possible hardware-software inballances. Commercial systems like [10] and [11] can easily be modeled in this way. The SUN ATM622-s SAR chip implements the physical, ATM and SAR layers in hardware. It incorporates three interfaces to physical layer (Utopia), host processor (SBus), transmit and receive memory. The communication between the host processor and the chip can be in the packet level i.e. exchange of SAR layer PDUs between host processor and SAR chip. Therefore higher that SAR level protocols are executed at the host processor while the lower level ones are carried out by the chip itself. Buffering is completely managed by hardware in order for the chip to be able to keep up with the 622 Mbps incoming and outgoing rate. The FORE ASX-200 ATM switch is among the most popular switches in the ATM market. It has 18 25 Mbps and 4 155 Mbps ports and implements sophisticated buffering per connection in hardware. Other chip sets like the NEC SAR chips implement SAR with a combination of hardware and software: The host processor provides to the chip a portion of its memory which is managed by the chip to store cells and

perform SAR while the extra memory needed for control is on-chip.

2.2 ATM system functions

Through the simplified model described above we have exhibited the need for buffering architectures that in general handle queues. However a more thorough study of the functions offered by ATM is required together with how they can be supported by using queues. The basic functions executed by ATM are Segmentation and Reassembly (SAR) which collects cells that belong to the same connection and forms a higher level PDU, Credit and Rate based flow control which safes the network from data flooding from certain bursty connections and Selective Discard that provides a mechanism for selecting cells to drop when all buffers are full or exceed some threshold. All these functions are implemented inside the AAL-5 layer and are analyzed in subsequent subsections.

2.2.1 Segmentation and Reassembly (SAR)

The AAL-5 layer is divided into two parts: The Convergence Sublayer (CS) and the Segmentation and Reassembly Sublayer (SAR). The first part performs message identification and time/clock recovery and is not discussed in this text, while the second one is responsible for the segmentation of LLC encapsulated PDUs into 48-byte cells at the transmit side and the reconstruction of cells to make up an LLC encapsulated PDU at the receive side. In more detail AAL-5 performs the following on transmit side (the inverse occurs on the receive side) as is displayed in figure 1:

1. Add as much padding as it is needed in order for the incoming PDU to become multiple of 48 bytes.
2. Add an 8-byte trailer to the "packet" emerged from the previous step as defined in [21].
3. Segment the resulting packet into 48-byte cells and forward them to the ATM layer.
4. In each 48-byte cell add a 5 byte header to transform it into a 53-byte ATM cell.

This process is known as Segmentation and Reassembly (SAR) and buffering is required to perform this function on per connection basis. In order to have per connection buffering and considering the connection-oriented nature of AAL-5, one FIFO queue per active connection is required. The fact that we have to maintain queues only for the active connections reduces the total number of queues that must be provided by the buffering subsystem but it still remains unknown and it is a crucial parameter. An implicit requirement n SAR is high utilization of the existing buffers; this can be accomplished by implementing all the queues in shared buffer space. Implementation in shared buffer space has another advantage: Each queue can consume a user-controlled portion of the total memory. The SAR function reconstructs AAL-5 PDUs and the higher level layers (e.g. LLC) are responsible for fetching them for further processing and delivery. However the buffering subsystem can aid the upper layers by constructing, apart from AAL-5 PDUs, higher level PDUs. This can be achieved through the provision of mechanisms for construction of queues of queues i.e. each higher level queue has as elements queues of ATM cells and each one of them constitutes an AAL-5 PDU.

2.2.2 Rate-based flow control

Rate-based flow control is the standard end-to-end congestion control mechanism adopted by the ATM Forum. When a connection's buffer exceeds a certain threshold value a special Resource

Management (RM) cell has to be sent back and notify the source to decrease its Allowed Cell Rate (ACR) according to some function. Another way of implementing this mechanism is to force the source to decrease it's ACR as long as it does not receive a special RM cell that notifies continuation at the same ACR. This mechanism works well under certain conditions but not under any conditions. Rate-based flow control does not impose any extra constraints in comparison with Credit-based flow control and thus we will only refer to Credit-based flow control from now on. More information on Rate-based flow control can be found in [5] and [6].

2.2.3 Credit-based flow control

Credit-based flow control is a mechanism that ensures fairness between different traffic flows and generally applies congestion control in a hop-by-hop fashion in switches. The upstream switch transmits only when it knows that there exists free buffer space at the downstream switch (receiver). For the upstream switch to know when there is free space, a credit counter is required and a coding scheme for the credits. The credit counter of the upstream switch counts free space of the downstream switch. The downstream switch sends tokens or credits whenever buffer space is freed i.e. when a cell leaves this switch. When the upstream switch receives a credit it increments the counter and when it sends a cell it decrements the counter. This simple scheme is a single lane flow control mechanism. It's main problem is in the case that one connection is very congested and consumes all available buffer space leading the other connections even if there is available network bandwidth in starvation. This situation is very similar to head-of-line blocking.

More advanced schemes have been proposed to deal with this issue. One of the most interesting is multi-lane Credit-based flow control where buffer space is partitioned into small segments one for each virtual connection and it is guaranteed that each connection will always have a minimum buffer space at the downstream switch. In this way no connection can be blocked by others. An optimization of this approach exists where the partitions are not as many as the connections but only as many as the different flows i.e. source - destination pairs that follow the same route in the network. In this implementation we have two types of credits: one type that tells us whether there exists free space at the downstream switch and another that tells us whether the flow group has free space at the downstream switch. More detailed description of credit-based flow control can be found in [7] and [8]. Adapters use credit-based flow control as well but only on the side connected to the switch, for flow control termination purposes. Credit-based flow control is a very efficient mechanism for congestion control and is also challenging by means of implementation complexity. It can be implemented at various levels: It can be encoded at the physical layer frames, or carried over ATM cells. In either case buffering plays an important role in distinguishing between different buffer spaces occupied by different connections and thus providing the necessary primitives to assign to each connection a fair amount of bandwidth and buffer space.

For example consider a 2-level credit scheme where the first type of credit is given when there is buffer space available at the destination node (destination credit) and the second type of credit is given when there exists free buffer space at the downstream switch (pool credit). Cells that belong to the same connection in this flow control scheme belong also to one of 4 categories:

1. Cells that have no credit
2. Cells that have destination credit
3. Cells that have pool credit

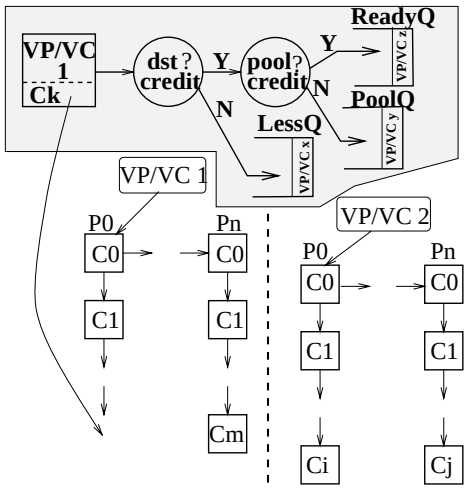

Figure 3: Using FIFOs to implement Credit-based flow control

4. Cells that have both credits

resulting in 4 queues per connection. A more clever scheme depicted in figure 3 can be implemented with only 3 queues hierarchically placed. Initially incoming cells are trying to fetch a destination and a pool credit. If they manage to take both they are enqueued in the Ready Queue and if they manage to take a Destination credit they are enqueued in Pool Queue. Otherwise they are enqueued in the Creditless Queue pending for both credits. Pool credits are given only to cells from the Pool Queue or cells that already have a Destination credit in general.

2.2.4 Selective Discard

This is a secondary mechanism that does not naturally exists in ATM systems, concerns very congested networks and has been proposed by [1] and implemented by companies like FORE and CISCO. However large buffer space we provide there may be times that bursty network traffic overflows the buffers and forces the system to drop cells. Instead of dropping the new incoming cells that may have a higher priority than others or generally belong to CBR or VBR traffic it is more appropriate to drop cells from a buffer assigned to a congested connection that carries lower priority traffic or ABR traffic. In this way we maintain quality of service by cutting down the ABR traffic when it is necessary. Buffering then needs to provide mechanisms for dropping selected uncompleted AAL-5 PDUs. In other words the buffering subsystem must provide a mechanism to free a whole queue when explicitly ordered as shown in figure 4.

Another approach is EPD (Early Packet Discard) where complete AAL-5 PDUs are dropped in case of congestion on the fly. In other words when is being decided that one connection has reached a certain threshold or it is of low priority then all incoming cells are being dropped until a complete AAL-5 frame is thrown away. EPD does not require any special support by the buffering architecture. More information on EPD can be found in [20].

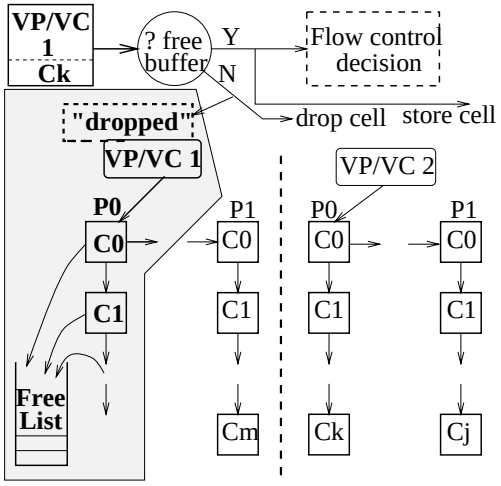

Figure 4: Using FIFOs to implement Selective Discard

3 Queue Management Architecture

So far we have regarded the buffering subsystem as a black box that handles addresses to buffers constructing queues of cells that physically reside in CBM. From now on we will refer to this system as Multi-Queue Manager or simply Queue Manager (QM). The Queue Manager whose functionality has been defined in section 2, can be regarded as a custom processor that inputs instructions specific to queues and replies. This master-slave approach (QM is the slave), as presented in figure 5, has been considered as the most suitable because it simplifies the interaction between the external master device and QM and provides a general-purpose interface that can be incorporated in any ATM system.

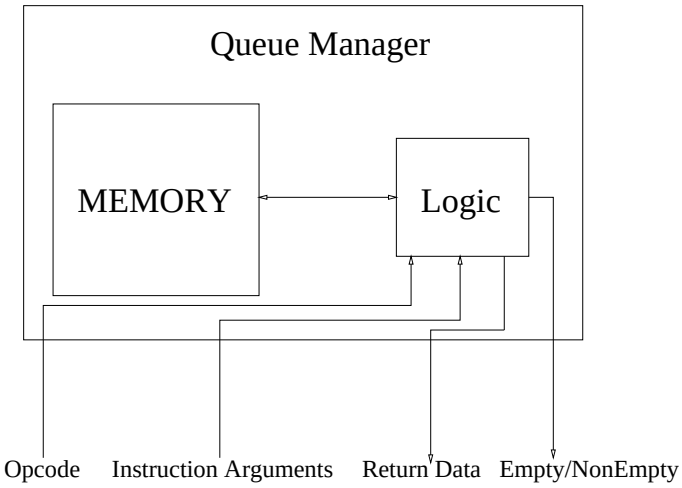

Figure 5: Queue Manager's generic architecture

The core instruction set that covers all the required functionality follows:

Init(arg1,arg2,arg3,arg4): It sets the address space each of the required data structures (Head/Tail table, Pointer Memory) takes up. The Head/Tail table structure starts at address arg1 and uses arg2 words and Pointer Memory structure starts at address arg3 and uses arg4 words. In addition the initialization procedure starts by filling the Free List and setting all queues to empty.

EnQ(arg1,arg2): Enqueue at arg1 queue the address specified by arg2. Answer if the queue was empty before issuing this instruction.

DeQ(arg1): Dequeue an element from queue arg1. Answer with the address of the dequeued element and if queue arg1 has become empty after the completion of this instruction.

Top(arg1): Return the top element of the queue arg1. No changes in arg1 queue occur. In addition reply if the queue is empty or not. This instruction plays the role of both a "Top" instruction and a "IsEmptyQ" instruction.

GetFree(): Returns an address from the List of Free Addresses (Free List) to be used to store a cell. A reply is also returned whether the Free List became empty after this instruction or not.

RetFree(arg1): Returns arg1 address to the Free List. In other words the buffer pointed by arg1 is not used anymore and with this instruction it becomes available for use. A reply is also returned whether the Free List was empty before issuing this instruction or not.

Read(arg1): It simply reads a word from address arg1 of the internal to the QM memory. This instruction is used only for debugging purposes.

Write(arg1,arg2): Writes arg2 at address arg1. This instruction is only used for debugging purposes.

The instruction set described above defines apart from the operational core, instructions that aid in the debugging process which is one of the most time consuming tasks in the digital design flow. These are the Read/Write instructions that can read and write arbitrary addresses in the Queue Manager's memory space and the Top instruction that returns the top queue element of any queue without altering its state. All instructions can be encoded in 3 bits resulting in a small and compact (no bits remain unused) opcode that can be decoded fast and scales easily in high-speed systems.

In many applications it is required to have one queue per VPI/VCI but this is not possible because the number of all possible connections is way too large; the VPI/VCI field is 24 bits for UNI resulting in 2^{24} possible connections. Furthermore, the provision of 2^{24} queues even if it was technologically feasible it would be a waste of space considering that the number of active connections in a typical ATM system is a much smaller number (the network is always designed in a way that it never exhausts all its resources). Consequently some support must be added to map all the possible connections to a smaller set of provided queues. There are many different approaches for mapping or translation from a connection number to a queue number:

1. Use of a Content Addressable Memory (CAM) for full translation of VPI/VCIs into existing queue numbers. This solution provides full mapping of the VPI/VCI to a number of queues determined by the size of the CAM.

2. Use of a DRAM addressed by the VPI/VCI. This solution also provides full mapping of VPI/VCI to a number of queues determined by the size of the DRAM.

3. Explicitly use some of the VPI/VCI bits as the queue number. The use of only a fixed number of bits from the VPI/VCI as the queue number results in many connections mapped to the same queue number. The problem can only be reduced (but not eliminated) through the proper assignment of VPI/VCIs by the network manager. The number of queues is a parameter of the system.

4. Use a fixed number of bits from the VPI and the VCI and let the user dynamically define which ones. This solution is the most common in commercial systems and allows the network manager to use its own policy in VPI/VCI assignment at connection setup. The number of queues is a parameter of the system together with the level of flexibility on which groups of bits to select.

5. Map the VPI/VCIs into queue numbers through some well-defined hash function (e.g. CRC-10). Obviously the number of queues is determined by the properties of the hash function so as to have few collisions during the VPI/VCI mapping to queue numbers.

Each solution has its advantages and disadvantages with the first one being the most expensive fast and flexible; the second one being fair for translating all the VPI/VCI bits with a reasonable delay; the third one being the most simple, suffering from collisions; the fourth one being the one used in most commercial systems with a fair cost/performance ratio; and the fourth one being a low cost but highly depending, on the VPI/VCI assignment and the hash function, solution.

Writing down the main characteristics of the architecture we end up with a structure that has multiple hierarchical queues implemented in a shared buffer space. Two are the main parameters of these queues: The number of queues and the number of buffers that can enter the queues. These two parameters cannot be determined because they depend greatly on the network traffic and therefore a hardware implementation that is scalable by means of these two parameters would provide a cost/performance ratio to be decided by the network administrator. The data structures that implement this architecture follow in section 3.1.

3.1 Definition of the basic data structures

To understand better what stated above lets see the optimal implementation of FIFO queues in shared buffer space as has been originally introduced in [9]. We define as *cell buffer* the memory space required to store one ATM cell i.e. 53 bytes. In order to be able to access one cell buffer(CB) and link many of them in chains one pointer per cell buffer is required, called *cell pointer(CP)*. The memory that holds the CBs is the *Cell Buffer Memory(CBM)* while the one that holds all the CPs is named *Pointer Memory(PM)*. In addition the minimum requirement for implementing one FIFO queue is two pointers: one pointing at the head and one pointing at the tail of each queue called *Head/Tail table*, plus one bit that identifies the state of the queue i.e. whether the queue is empty or not, called *Head/Tail empty bit*. The data structures presented so far are capable of linking and creating FIFO queues of CBs. However, we can't distinguish between empty CBs and CBs attached to a specific queue. For this reason a Free List is required and can be maintained with constant memory: A *Free List head* and a *Free List tail* register plus a *Free List empty bit*. How can this be achieved? If we link all CBs at start up sequentially (CB 0 points at CB 1 which points at CB 2 etc) and make the *Free List head* register point at 0 and the *Free List tail* register point at $N_c - 1$ where N_c is the total number of CBs then we have an optimal Free List implementation. In addition we must be careful to set the *Free List empty bit* to 0. All mentioned data structures are presented in figure 6.

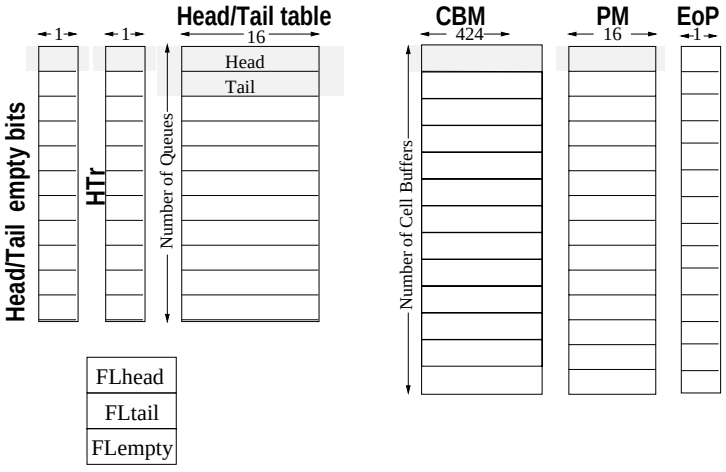

Figure 6: Queue Manager's basic data structures

The equation that gives us the total memory used by all structures in bits as a function of the number of provided queues (N_q) and the number of cells (N_c) follows:

$$S_{all} = S_{CBM} + S_{PM} + S_{HTm} + S_{HTe} + S_{FLm} =$$
$$424N_c + N_c(\log_2 N_c + 1) + 2(\log_2 N_c + 1)N_q$$
$$+N_q + 2(\log_2 N_c + 1) + 1$$

S_{CBM} is the total memory used by the buffers that keep the cells and have size of $424N_c$ in bits. Pointer Memory must have width equal to the bits required for addressing all CBs (i.e. $\log_2 N_c + 1$) and height equal to the number of buffers (N_c). Head/Tail table has height equal to the number of supported queues (N_q) times 2 (head and tail) and width equal to the bits required for addressing all CBs. The Empty Bits are as many as N_q and the Free List Head/Tail registers require $\log_2 N_c + 1$ bits each. In addition the Free List Empty bit is required. By using this formula we can easily compute the amount of memory used for the pointers and compare it with the one used for storing the data (CBM). Some indicative numbers for reasonable memory configurations are in the following table:

N_c	1024	4096	8192	32768	131072
N_q	128	512	1024	4096	16384
$\frac{S_{all}-S_{CBM}}{S_{all}}$	3%	4%	4%	5%	5.1%

3.2 Design Tradeoffs

There are many tradeoffs to be considered regarding placement of the defined data structures in memory:

1. On-chip or off-chip placement of required data structures. The first offers speed but provides no flexibility and has little scalability in size: It is much easier and costs less to take off a memory chip and hook up a larger one in size instead of producing a new ASIC in an improved technology that has more on-chip memory.

2. Many separate memories for the data structures enable parallelism but the cost is too high. For example if all the required data structures are placed in a double width memory where the Head, Tail and Empty Bit fit in one word then the execution time of an Enqueue operation is 2 cycles in comparison with the 4 cycles in a one single-width memory implementation as we will see in section 4. Thus the placement of the Head, Tail and Empty Bits plays an important role in cost and speed even when we choose a single memory. There are two choices regarding the Head/Tail placement,

   (a) The Head/Tail table is placed in the memory with the Head and the Tail being in one memory word.

   (b) The Head/Tail table is placed in the memory as shown in figure 6 with the Head and the Tail in contiguous memory words.

   and two choices regarding Empty Bits placement:

   (a) The Empty Bits are stored in a separate memory space utilizing the existing memory 100%.

   (b) The Empty Bits are encoded inside the Head, the Tail or in both exploiting parallelism and accelerating some operations.

3. The mapping between the Queues and the connection numbers (VPI/VCI) is not an easy issue and some sort of hashing must be used.

4. In order to segment and reassemble packets i.e. queues of cells, we must provide extra mechanisms. There are two solutions: Either implement queues of queues using an extra set of the same data structures (link the cell queues head to head) described above or simply add an extra bit in the Pointer Memory of the previous scheme to notify the end of one packet and the start of another (link the cell queues tail to head). The second

implementation is the most preferable because in ATM the packets arrive and depart in a FIFO manner while the first one offers the ability to dequeue queue i+1 without having accessed the whole queue i.

A first tradeoff is whether we should place the structures in on-chip or off-chip memory or memories. Such an architectural choice is very important because it implicitly defines the technology to be used. On-chip memories are fast, small and require ASIC implementation of the buffering. However there are FPGAs with on-chip memory which is still narrow and slow and does not offer size suitable for our needs. An extensive discussion of an architecture implemented in ASIC with on-chip memories can be found in [4]. On the other hand off-chip memories are widespread and offered in a variety of speeds and sizes. They also have the advantage of scalability in both size and speed: It is always possible to take off one slow and small memory chip and put on a larger and faster one. Our choice is to use off-the-self memory chips which are low cost and scale well in size and speed. Secondary but with equal importance is the choice of a DRAM or an SRAM. It is true that DRAMs have gone a long way in achieving high bandwidth (SDRAM) but latency is still large and the interface much more complex in comparison with SRAMs. Therefore off-the-self SRAM chips is a solution with many advantages suitable for our architecture.

All the required data structures can be placed in separate memories, in one memory or some of them in one memory and some in separate memories. Many different memories add to the cost of the architecture because more chips and more pins (more expensive packaging) are required. Thus one external memory for all structures seems a good choice regarding cost memory utilization and scalability. Another advantage of using a single off-chip memory is flexibility, each data structure can be programmed to utilize a portion of this memory. Of course the great disadvantage is the serialization of every function over the queues. An optimization that one may think is placing the Head/Tail empty bits in a separate on-chip memory that would be very small in size in comparison with the off-chip memory and would allow for some parallelism. However this solution would reduce the scalability of the architecture as the maximum number of queues would be forced not to exceed the provided number of bits the on-chip memory has. For designs with fixed number of queues to support it is a fair solution.

After deciding that the best solution for our purposes is having only one memory for all structures, another consideration is how to place Head and Tail pointers of the Head/tail table for each queue inside the memory. If we put them one beside the other we have the advantage of accessing both the Head and the Tail of a queue in one cycle and the disadvantage of widening our memory. The size of the head pointer should be at least 13 bits to address 8192 CBs which is a reasonable number only for a prototype, a commercial chip should have much more (between 15-20 bits). Memories with width greater than 16 bits tend to be expensive and with few brand choices. By placing the Tail under the Head i.e. in subsequent addresses economy in memory width can be achieved. In addition we provide an easier to scale architecture because we can easier go from 16-bit wide memories to 32-bit wide memories comparing with the passage from 32-bit wide memories to 64-bit wide memories. Of course we must pay the price which is doubling the access time of head and tail for the dequeue instruction where both head and tail of a queue must be read.

Another tradeoff is the placing of the Head/Tail empty bits: Even if we place them in the same memory with the other structures there is still a number of choices. First we can place the empty bits beside the head or tail or beside both in the Head/tail table. In this way we can read the empty bit of each queue together with the head or tail. Placing the empty bit beside both the head and tail pointers we delay the enqueue instruction because we must read and write the head pointer (we would not do this normally) in the case that we perform an enqueue on an empty queue. But if we place it beside the tail pointer only, we gain the acceleration of

the enqueue by one cycle (we read the tail and the empty bit in one cycle) and the same applies to the dequeue instruction where we have to read both the head and the tail anyway. In any of these implementations we gain in speed but we loose in memory utilization. Another alternative is to place the Head/Tail empty bits in a separate memory space in the same memory that all the other data structures reside. In this way we make full utilization of the existing memory with the extra cost of one more cycle in the execution of the Enqueue command. For example if the external memory is 16 bits wide and we want to implement 1024 queues then 2/32 of the total memory used for the Head/Tail table structure is wasted or 2048 bits are not usable if we choose to implement the empty bits in the Head/Tail table. These bits can form 2048/16=128 pointers for pointer memory which means 128 more cell pointers and the ability to use 128 more cell buffers if there is enough memory in CBM.

Another consideration is how to implement the hierarchical queues (queues of queues). There are two possible implementations:

- Either by keep on adding cells at the back of the queue even if the queue has one completed AAL-5 PDU (head to tail linking of queues) and marking the last cell as the end of the PDU,
- or by implementing another set of queues that have as elements the cell queues (head to head linking of queues).

In the first implementation each queue is regarded as a series of cells that form a series of AAL-5 PDUs. In the second implementation each AAL-5 PDU is regarded as one element of a another set of queues. The first implementation has the advantage of not requiring another Free List and the disadvantage of not providing the functionality to remove a whole AAL-5 PDU from the queue before we remove all its cells one by one. The inverse applies to the second implementation. However a careful study of ATM would give us a hint for which one to choose: In AAL-5 for a specific connection no cells can pass forward older in arrival time cells due the connection-oriented nature of the protocol. Therefore the best choice is the simpler one of using one extra bit in each CP for marking the end of a AAL-5 PDU and signaling the start of the next. This bit is called End-of-PDU bit or EoP bit.

4 Hardware Implementation

In this section we provide 3 different hardware implementations of Queue Management:

- The implementation incorporated in the MUQPRO I prototype
- An implementation similar to the one of MUQPRO I, that places the Empty Bits in a separate address space achieving 100% utilization,
- And last but not least an implementation that tries to achieve the maximum possible parallelism without taking into account the cost/performance ratio and without using any sophisticated architectural tricks like VLIW design or pipelining.

The first one which is the most complete as has been used in the MUQPRO I project is a 16-bit architecture that places all data structures in one memory with the Empty Bits incorporated inside the Tail of the Head/Tail table. It runs at 30 MHz and maintains two sets of queues, one used for cells and another for scheduling connections in one of 8 available priorities. MUQPRO I Queue Manager is capable of serving 5 155 Mb/s SONET OC-3 links (4 incoming and 1 outgoing), the attached microprocessor that manages the OAM cells and keeping the 1024 VPI/VCIs in 8 priorities according to the High Priority First algorithm as described in [3]. The target technology was conservative Field Programmable Gate Array (FPGA) from ALTERA corporation, the EPF10K50.

The second implementation - also 16-bit - has one main difference from the MUQPRO I; It requires one extra cycle in executing the Enqueue instruction but it fully utilizes the existing memory. It runs at 20 MHz and fits into an EPF10K40 FPGA. At last the so called "fully parallel" implementation was an initial 5-bit experimental architecture that only demonstrates the available parallelism of the Queue Management operations using the limited bandwidth internal FPGA memory and as many pins required to have full parallelism. In section 6 an extensive discussion on the advantages and disadvantages of all architectures exists.

4.1 The MuQPro I implementation

MUQPRO I is an FPGA-based prototype of a 4 input and 1 output switch. It demonstrates the use of a Queue Management module that fully supports all the necessary operations executed by the switch. The architecture of the system that operates at 30MHz is presented in figure 7 where 4 boards are required to turn the system into a 4x4 switch. In each of the boards there are 4 SONET OC3 modules that carry 155Mb/s traffic to and from the network resulting in a close to 1 Gb/s aggregate throughput which is considered satisfying taking into account the FPGA technology used. The Datapath module is the physical road from the SONET modules to the Cell Bodies SRAM memory where all incoming ATM cells together with their headers reside. Another SRAM (Connection Table) is used to keep per VPI/VCI information useful for separating the connections that are being serviced by the specific board and for VPI/VCI translation (with the aid of the i960 microprocessor). The μP implements Quantum Flow Control (a standard credit-based flow control scheme supported by the industry), extracts/inserts ATM management cells (OAM cells) and is used to initialize and test the board's modules. In order to make the system's architecture more general a Main Controller module is used that provides the interconnection network and the interface functions between the various modules. The Scheduler module implements the High Priority First algorithm to schedule cells for transmission as presented in [3]. At last the Queue Manager module (QM module) implements all the required buffering functions and it is the one that will be presented in this section. In general the data flows that concern the Queue Manager are:

- From the Datapath to the Queue Manager for request and release addresses in the Cell Bodies SRAM in order for the Datapath to transfer cell bodies to/from the SONET I/Fs and to/from the μP through the Main Controller.
- From the μP to the Queue Manager to attach/detach addresses of OAM cells and test the Queue Manager.
- From the scheduler to the Queue Manager in order to attach/detach VPI/VCI connections to specific priorities.

Queue Manager has the ability to service all flows at the required speed without giving away its characteristics that make it general enough to be used in a variety of similar systems. To be more specific, the Queue Manager serves 4 incoming and 1 outgoing 155Mb/s flows, the flow of management cells (OAM cells) to/from the μP and the scheduling of 1024 VPI/VCIs in 8 priorities that can change dynamically as described in [3]. Notice that the Queue Manager is not limited in 8 priorities, 1024 queues and 8192 elements per queue. Its only limit is set by the 16-bit datapath and the size of the attached SRAM.

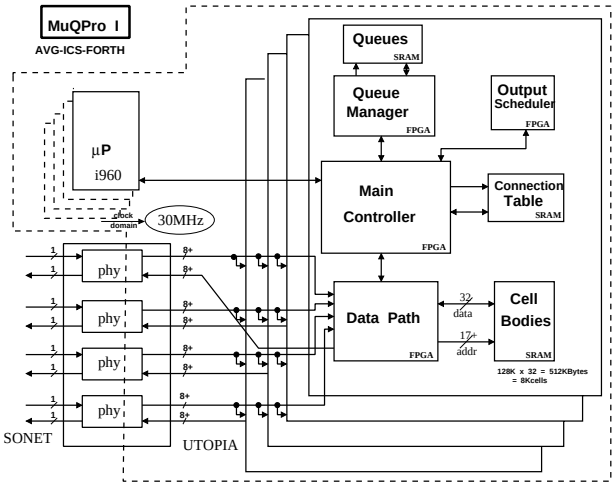

Figure 7: MUQPRO I system architecture

4.1.1 Interface

The interface of the Queue Manager with external devices is presented in figure 8. Generally the Queue Manager is a *slave synchronous* device with multiplexed instruction arguments and data. This means that the instructions and their arguments are latched synchronously to the positive edge of the clock signal and the same applies to the replies. In more detail the I/O and control pins are:

Opcode: The opcode of the instruction to be executed by the Queue Manager. It is 5 bits wide with the most significant bit (bit 5) being the selection of which set of queues will be activated. Bit 4 signals the NOOP instruction and the device remains idle.

Opcode	Instruction
X1XXX	NOOP
Y0$I_2I_1I_0$	instruction $I_2I_1I_0$ on set of queues Y

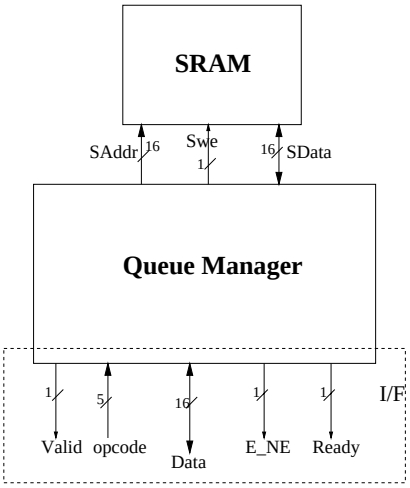

Figure 8: Queue Manager block diagram

In more detail here are the opcodes of all available instructions. A general description can be found in 4.1.2 while a more detailed description in appendix A.

Opcode ($I_2I_1I_0$)	0	1	2	3	4	5	6	7
Instruction	Init	Enqueue	Write	Dequeue	Retfree	Read	Getfree	Top
Arguments	4	2	2	1	1	1	0	1

Data: This is a 16-bit wide bidirectional bus. This bus is used for the arguments of the instructions when a new instruction is issued and the return data of the executed instruction. One turnaround cycle is required between the driving of the bus by the QM and the external master device. The "multiplexing" of the arguments does not add any extra delay to the instructions executed by the Queue Manager because while the first access of the SRAM occurs at the address computed through the first instruction argument, the second argument is fetched. The 16-bit datapath seems to be more than enough for the purposes of the MUQPRO I: It uses 1024 queues (10 address bits) and a maximum of 8192 elements (13 address bits).

E/NE: This signal is used to indicate that the instruction currently executed altered the state of this queue. For an enqueue and a Retfree instruction an E/NE high signal indicates that the queue was empty before the current instruction was issued and for a dequeue and a Getfree instruction indicates that the queue after the completion of this instruction will become empty.

Valid: This signal is implicitly required because the lack of it would force the external master device to have a counter in order to determine the timing of the reply. Therefore when the signal is high the Data is driven by the QM with the answer or the E/NE output is valid.

Ready: This signal is high whenever the QM is ready to accept a new instruction in the *next* positive edge of the clock. Its usefulness comes from the need to feed the Queue Manager with several instructions back-to-back and it informs us when it can accept a new instruction while it executes another. When the Queue Manager is idle the Ready

signal remains high all the time. In the case of the retfree instruction a new instruction can be issued the right next cycle. However the general rule is that no more than 2 instructions can be under execution during each cycle.

4.1.2 Instruction set

Init(arg1,arg2,arg3,arg4): It sets the address space each of the required data structures (Head/Tail table, Pointer Memory) takes up. The Head/Tail table structure starts at address arg1 and uses arg2 words and Pointer Memory structure starts at address arg3 and uses arg4 words. In addition the initialization procedure starts by setting the Free List to non-empty (reset Free List Empty register), initialize the Pointer Memory in order to implement the Free List as described in 3.1, and at last set the Head/Tail Empty bits to 1 to initialize all queues to empty. Recall that the Head/Tail Empty bits are incorporated in the Tail of the Head/Tail table.

EnQ(arg1,arg2): Enqueue at arg1 queue the address specified by arg2. The E/NE pin is high if the queue was empty before this instruction.

DeQ(arg1): Dequeue an element from queue arg1. The E/NE pin is high if the queue became empty after this instruction.

Top(arg1): Return the top element of the queue arg1. No changes in arg1 queue occur. If the queue is empty the E/NE pin becomes high in order for this instruction to be able to answer the question if the queue is empty.

GetFree(): Returns an address from the List of Free Addresses (Free List). If the Free List became empty after this instruction the E/NE pin becomes high in a similar way with DeQ.

RetFree(arg1): Returns arg1 address to the Free List. In other words the buffer pointed by arg1 is not used anymore and with this instruction it becomes available for use. If the Free List was empty then the E/NE pin becomes high in a similar way with EnQ.

Read(arg1): It reads and returns a 16 bit word from address arg1. This instruction is used for debugging purposes.

Write(arg1,arg2): Writes arg2 at address arg1. This instruction is also used for debugging purposes.

4.1.3 Internal Architecture

The Queue Manager maintains two set of queues, one that links ATM cell bodies and another one that maintains queues of VPI/VCIs. We have followed an hierarchical design process, therefore we have designed one module that does all the operations on one set of queues and have used two instances of this module to create the Queue Manager. In this way the design and verification time was reduced significantly. The first set of queues is used for linking cells while the second one depicts priority classes. On each queue all the operations described in subsection 4.1.1 can be conducted. The implementation we found as the most suitable for the purposes of MUQPRO I was the one that places all the required data structures in one external memory (SRAM). The Head and Tail of each queue are one on top of the other i.e. reside in subsequent addresses in the SRAM and the empty bit is encoded inside the Tail as the most significant bit(MS bit). This choice came up after the requirement to reduce the number of

cycles it takes to execute the Enqueue instruction by one in order for the Queue Manager to be able to handle all requests at the specified period of time. The instruction set of each Queue Manager module (QM module) is described in appendix A in full detail.

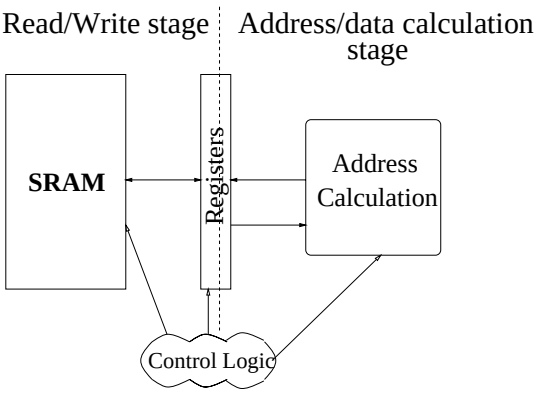

Figure 9: Queue Manager pseudo-pipeline

In a more abstract way each Queue Manager module can be seen as an address and data generator that tries to keep the SRAM occupied as much as possible given that there is a continuous stream of instructions heading for the QM. On the other hand pipelining is a very effective solution for high utilization of the existing hardware and we tried to use it in our architecture although the serialization of the operations for each instruction due to one resource (SRAM) was unavoidable. However, by splitting the operations in two parts one that computes address (and data in the case of write) and one that accesses the memory we can speedup the cycle time. In figure 9 we can see this pseudo-pipeline. It consists of the two stages plus logic that tries to keep it full as much time as possible. By making the argument fetching a separate stage we relax the off-chip communication between the Queue Manager and the Controller while the pseudo-pipeline relaxes the off-chip communication between the Queue Manager and the SRAM chip.

In figure 10 a block diagram of the QM module is presented. There are 4 register file blocks:

- BAR RF : It keeps the Base Addresses and the sizes of all data structures (Head/Tail table, Pointer Memory) that physically reside in the SRAM.
- SRC RF : It latches the instruction arguments.
- FL RF : It maintains the registers required for the Head, Tail and Empty bit of the Free List.
- TMP RF & CNT : The temporary registers needed to store previous head and tail values and the counter for the correct initialization of the module.

Three functions can occur in parallel in this datapath during each cycle: First an access to the external SRAM at address specified by the Saddr register. If it is a write then the data from the Sdata register are used. If it's a read then the return data are stored in the TMP RF. Second, new address and data are computed for store in the Sdata and Saddr registers. At last, a new instruction is fetched and its arguments are stored in the SRC RF. The parallelism enabled by the datapath is required to be used as much as possible and this is achieved by the proper design of the control path. To be more specific for each instruction a break down into SRAM accesses has been performed called micro-operations. An optimized scheduling of the

micro-operations is necessary to get the maximum possible parallelism for each instruction given that all structures reside in the same memory. However smart schedule we perform separately for each instruction, there are always cycles that it's impossible for one instruction to utilize all three functions supplied by the datapath. For example the EnQ instruction cannot start the first access to the SRAM before it has fetched its first argument (Queue number). For this reason more than one instructions can be executed during each cycle. On the other hand the need to keep the design small for cost/speed reasons - given the FPGA technology used - did not allow to have multiple adders and more registers. Therefore, more complex control has been designed in order to utilize the existing datapath in an optimal way by allowing the overlapping of instructions. The control is capable to allow a maximum of 2 instructions to overlap achieving a much better utilization of the existing hardware and without using extra registers and/or adders that have high cost in space and speed in FPGAs.

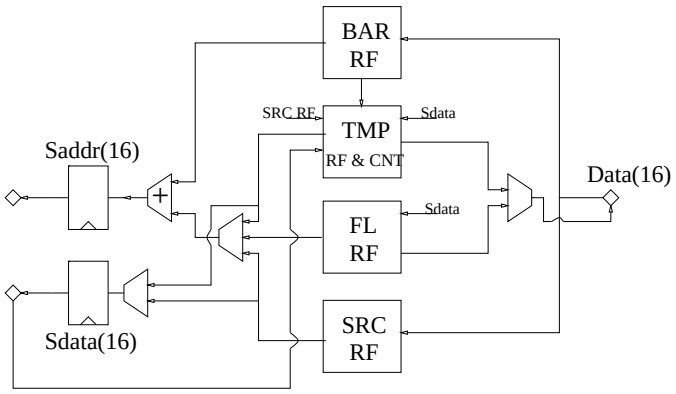

Figure 10: Queue Manager block diagram

In more detail, the EnQ instruction requires 6 cycles to complete its execution (see figure 11a), while the optimal is 4. However by using the overlapping technique a new instruction can start its execution at cycle 5 and consequently 2 EnQ instructions execute in 10 cycles, 3 EnQ instructions execute in 14 cycles etc. The more instructions that are available to feed the Queue Manager the closer the number of cycles each EnQ instruction takes converge to the optimal number 4. The same applies to all instructions and the overlapping occurs between all pairs of instructions with few exceptions mentioned below. Detailed diagrams of all instructions can be found in appendix A.

The DeQ instruction requires 4, 7 or 8 cycles to complete depending if the specific queue is empty, contains only one element, or contains more than one elements as presented in figure 11b. It extracts and returns the top queue element during cycle 5 and replies whether the queue will become empty or not during cycle 6. In case the queue is empty no overlapping with a new instruction can be applied because only at cycle 4 the result (queue is empty and no elements to dequeue exist) is known. In the other two cases it's useless to fetch a new instruction before cycle 7 for three reasons: First, because at cycle 5 the Queue Manager returns the top element over the Data pins and consequently cycle 6 must be used as a turnaround cycle. Second, for symmetry reasons since the user of this module will take into account the worst case, and third because there are no resources available to start a new instruction. One more important point is that a new instruction can be issued only at cycle 7 and not at cycle 8. This requirement does not slow down the instruction and simplifies the control significantly. Exactly the same applies to the EnQ instruction where the only cycle before termination available to issue a new instruction is cycle 5.

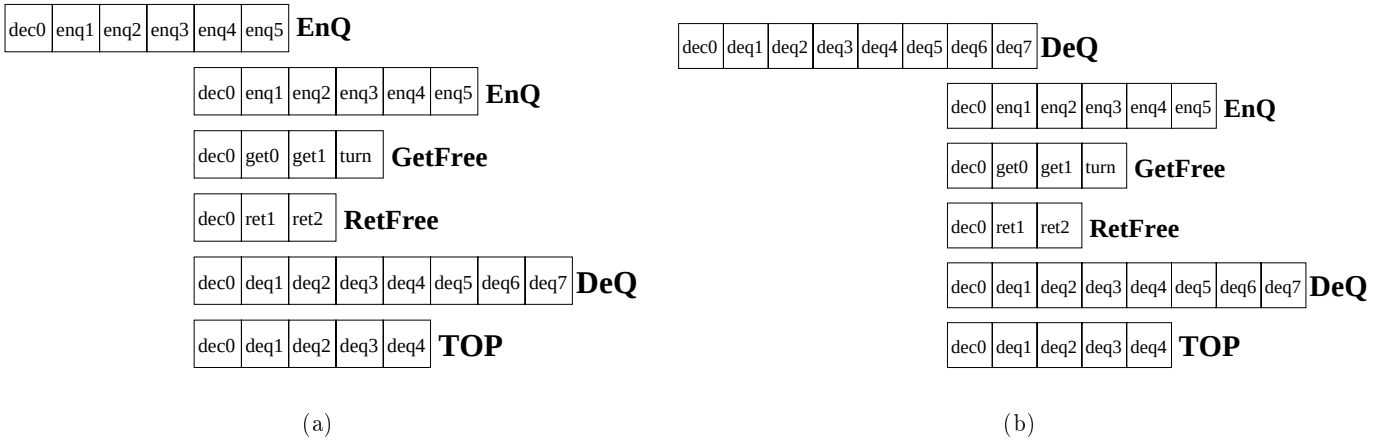

Figure 11: Issuing instructions right after an enqueue or dequeue instruction

The GetFree instruction is completed in 3 cycles. No overlapping can occur with this instruction because of the requirement for a turnaround cycle. The QM answers during cycle 2 while during cycle 3 it remains idle (turnaround cycle). In order to accelerate the real execution of this instruction (i.e. cycles it takes to execute without the turnaround cycle) whenever the Queue Manager is idle it computes the first address required for the first access to the memory. This is possible because this instruction takes no arguments and therefore the first address can explicitly be computed at any time. Although no new instruction can overlap with a GetFree the inverse can occur: During the execution of an EnQ or DeQ instruction a GetFree instruction can start (at cycle 5 and 7 correspondigly) but the real overlapping is only one cycle instead of the potential two. In other words the execution of the GetFree is extended by one cycle but because of the overlapping of two cycles with an EnQ or DeQ instruction it can terminate its execution in 2 cycles instead of three. The extension of the GetFree instruction by one cycle is because it needs to compute an address the right first cycle it is issued and when overlapping occurs the resources required for address calculation are reserved by the previous instruction (EnQ or DeQ) that executes. In figure 12a it is presented when a new instruction can follow a GetFree one.

The RetFree instruction is the faster from all instructions as it can complete its execution in only 3 cycles and a new instruction can be issued the right next cycle the RetFree has been issued as presented in figure 12b. However it has one limitation: Assume that an EnQ instruction has been issued and then overlaps with a newly issued RetFree instruction. At cycle 6 the EnQ instruction is finishing its execution, while the RetFree instruction is working and can accept a new instruction. The execution of more than 2 instructions in parallel *is not valid* for our architecture due to the high implementation complexity.

The Top instruction has the same functionality as the DeQ instruction until cycle 5 when it returns the top element of the specified queue. The Read, Write and Init together with the Top instructions have no optimizations and consequently cannot overlap with some other instruction.

4.1.4 SRAM and Master Device requirements

The external SRAM chip must provide the required data or perform a write 10-15ns before the next positive edge of the clock which practically means that for a 33ns clock cycle we need a 20-25ns access time asynchronous SRAM. In addition the opcode and data must be valid before

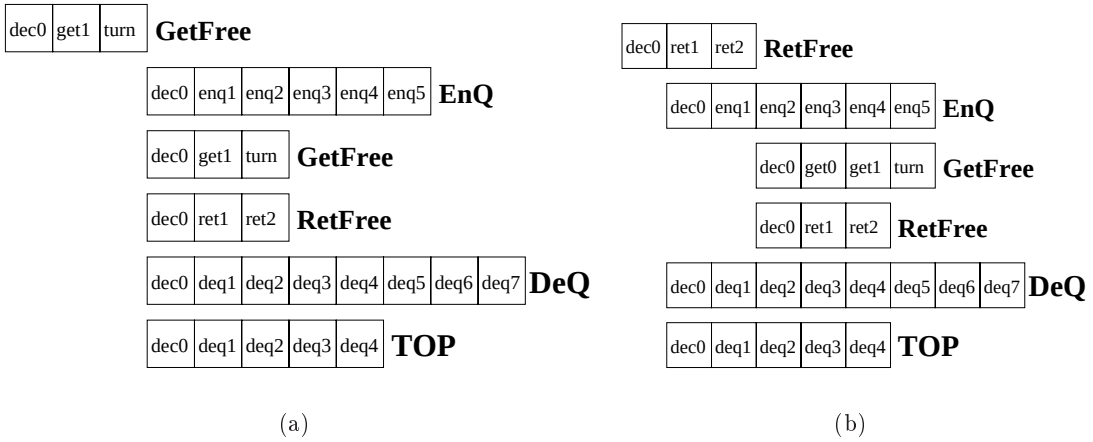

Figure 12: Issuing instructions right after a getfree or retfree instruction

the middle of the cycle in order for the decoder to be able to decode the current instruction and the data to be latched correctly during the same cycle.

4.1.5 Merging two instances of QM together

There are two approaches one could follow to construct two sets of queues:

- Duplicate the multiplexors of the datapath and add another one for each pair to select between the two or use double-sized multiplexors.
- Duplicate the datapath and the FSM of each Queue Manager and use an arbiter for the synchronization between them.

The first solution offers a smaller and more compact solution but in FPGA technology large multiplexors have very large delay For this reason we chose to use duplicate control and datapaths, one for each set of queues and an arbiter for synchronization that leads to a larger and faster design.

A block diagram with the two instances of the QM module is presented in figure 13. The need for the arbiter arises from the fact that two instructions addressing a different QM module may overlap and only one of them must have access in the resources which are the pins and the SRAM. This case happens because the QM module can initiate a new operation (start the execution of a new instruction) 2 cycles before it has finished the execution of a previous instruction. This new instruction may concern the other QM module which can start working. The right thing to do in those cases is for the older instruction to keep driving the I/O pins and after it has finished execution it must pass control to the second one. The QM is designed in such a way that it is not required from the second instruction in the case of overlapping to have control over the I/O pins before the first one finishes its execution. To accomplish this task we do not need an arbiter for all the I/O pins. For example the global Ready signal is the logical AND between the two Ready signals of the QM modules while the E_NE and Valid signals are the logical OR between the corresponding ones of the two instances. As far as it concerns the Data and Sdata bidirectional pins are driven by one module only when required which means that there is no need for arbitration for them. However the SRAM write enable and Saddr signals require arbitration because they are unidirectional signals in the QM modules. As far as it concerns the Data and SData pins they do not need arbitration because they are bidirectional

and whenever one does not use them it leaves them floating. Everything described so far are presented in figure 13.

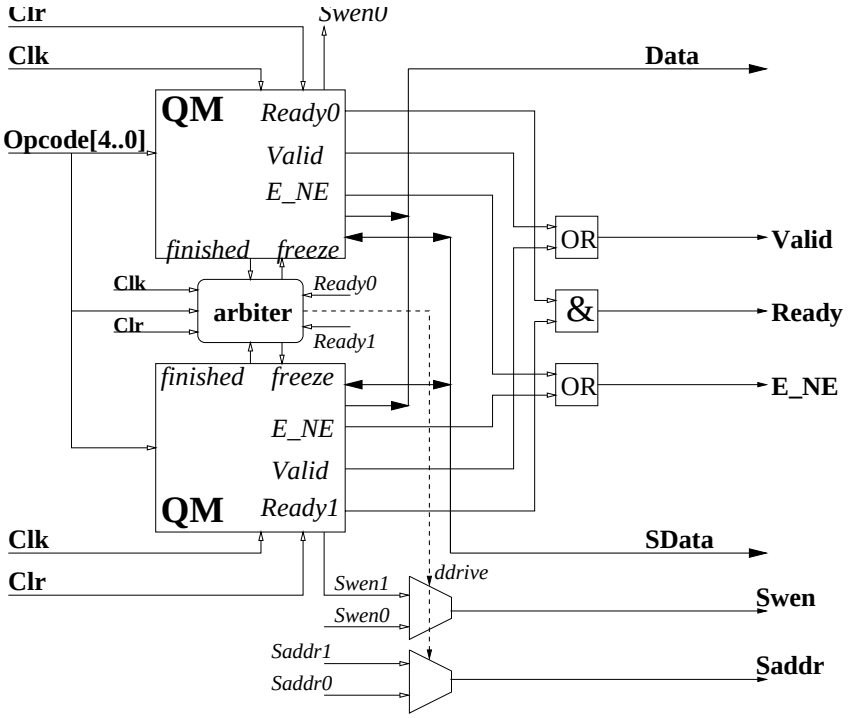

Figure 13: Making 2 sets of queues

The arbiter does the arbitration of the Swen signal. It takes as input the opcode and the finished and Ready signals from the 2 QM modules and returns a ddrive signal which selects who is driving the Swen pin plus the freeze signals that "freeze" the corresponding FSM in case the other one is under execution and no legal instruction can be issued. The finished signals come from the control path and are high when the QM module is executing an instruction for the last cycle i.e. next cycle it is going to finish. The logic of this Finite State machine is presented in figure 14. In this figure we can see what happens during the execution of an instruction concerning QM 0, however the case of QM 1 is symmetrical and is ommited. When in idle state if the opcode[3] bit is low (i.e. the given opcode is not NOOP) then depending on the value of opcode[4] bit that selects a QM module the QM 0 or 1 takes the ownership of the Swen pin (state m0 for the QM 0 module) and starts executing the instruction opcode[3..0]. Lets assume that an instruction concerning QM 0 is issued (the other case is symmetrical). There are a few different cases depending on the type of instruction:

A. if the instruction is enqueue, dequeue, top or init then we move to state edt_m0 where QM 0 becomes the master of the Swen pin and no new instruction can be issued.

B. if the instruction is retfree we move to state r_r0 where QM 0 becomes the master of the Swen pin and a new instruction can be issued. The new instruction can be an enqueue, dequeue or top.

C. if the instruction is getfree we move to state g_m0 where QM 0 becomes the master of the Swen pin and no new instruction can be issued.

The need for this type of grouping comes from the fact that 2 executing instructions may finish at the same cycle and the arbiter must know that both have terminated. For example think of a

retfree instruction in an empty free list issued right after (2 cycles before it finishes) an enqueue instruction. Then at the last cycle of the enqueue instruction both instructions terminate and the arbiter would not be able to distinguish between the two. In other words, the arbiter must have some way to remember what kind of instruction is executing while a new one is being issued. Each instruction may be in one of these states: Master of the I/O pins and no other instruction can start, master of the pins and a new instruction can start or master of the pins a new instruction is executing and no new instruction can be issued. Bare in mind that no more than two instructions may be under execution on the system in any cycle.

4.1.6 The instruction is enqueue, dequeue, top or init (case A)

When in cycle edt_m0 (coming from case A) either of these may occur:

- Either finished=1 and Ready=1 which means that QM 0 will finish next cycle and will be ready to accept a new instruction (this new instruction may be for QM 1 as well) at the same cycle leading to the idle state, or
- finished=0 and Ready=1 which means that next cycle a new instruction can be issued but QM 0 will still be working leading to state edt_r0.

In state edt_r0 a new instruction can be issued. If this new instruction is retfree and the free list happens to be empty then we move to idle state but QM 0 remains the master of the I/O pins for one more cycle. In any other case we move to cycle edt_w0 where QM 0 remains the master and depending on the instruction given on cycle edt_r0 we move to the proper state:

enqueue,dequeue,top: Next state will be edt_m0 (or edt_m1 if the instruction is for QM 1)

retfree and non-empty free list: Next state will be r_m0 (or r_m1 if the instruction is for QM 1). During state r_m0 or r_m1 the QM 0 or 1 is the master and no new instruction can be issued.

getfree and non-empty free list: Next state will be g_m0 (or g_m1 if the instruction is for QM 1) and no new instruction can be issued.

4.1.7 The instruction is retfree (case B)

When in cycle r_r0 (coming from case B) and assuming that QM 0 has issued the current instruction a new instruction can be issued and QM 0 is the master. The new instruction that can be issued cannot be a retfree because the Free List Head and Tail registers have not been updated by the current retfree instruction yet. For the same reason it cannot be a getfree instruction. Next cycle is r_m0 where QM 0 is the master and no new instruction can be issued during this cycle. Depending on the instruction issued in the previous cycle we move to state edt_m0, edt_m1 or idle.

4.1.8 The instruction is getfree, read or write (case C)

When in cycle g_m0 (or g_m1 depending to which module the instruction corresponds) QM 0 (QM 1) is the master and no new instruction can be issued. This is due to the turnaround cycle we discussed about in the subsection that describes the getfree instruction. Next cycle will be the idle state where we move when the finished signal goes high.

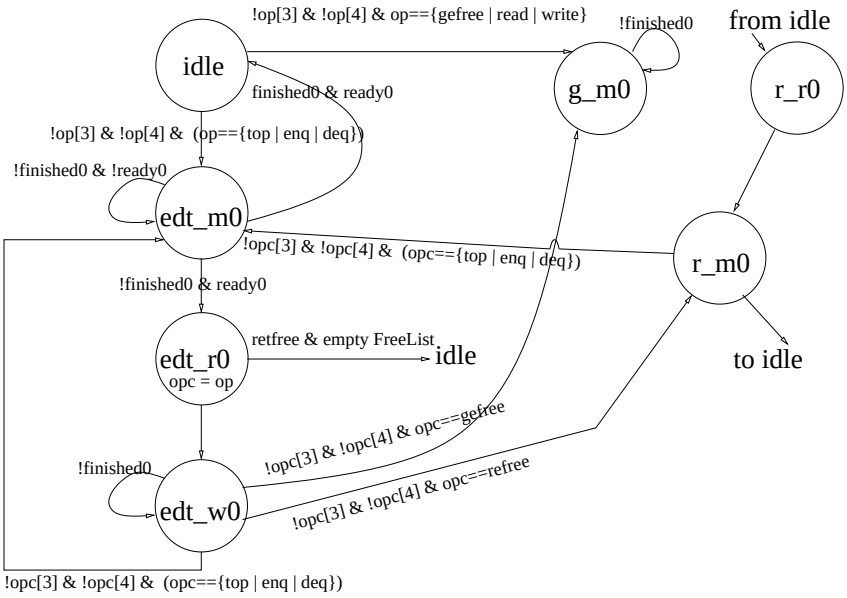

Figure 14: Arbiter finite state machine

4.1.9 Testing

The ALTERA MAX+PLUSII digital design environment has the ability to produce gate-level Verilog code. In order to simulate the external SRAM chip we wrote a simple model of the static memory in Verilog. Whenever a request to the SRAM model is made, after a constant time (a parameter to the model) it responds. Then we had to find a way to generate valid vectors and verify that the output of the simulation is the proper one. For the vector generation we wrote C code that performs the following:

- Queue Management in software
- Random selection of an instruction through a random() routine
- Creates an intermediate queue structure for the free slots: Every time we perform a getfree or a dequeue instruction the returned slot number is stored in this structure and whenever an enqueue or retfree instruction is performed there must be some slot in this structure to take it out.

In every iteration we select randomly a type of instruction, and then check if it is valid. An enqueue or a retfree instruction is valid if the intermediate queue structure is non-empty and we use one of the slots stored in it to enqueue to a random queue or return it to the free list. A dequeue instruction can be performed only if the randomly chosen queue is non-empty and this can be checked through counters that we maintain for each queue. In a similar way a getfree instruction can be performed only if the free list is non-empty which is checked through a global counter. After we have chosen a valid instruction we use the software queue management routines to execute it an we write a vector in Verilog syntax that performs the same instruction in the MUQPRO I implementation. After as many iterations as we have given at the command line we end up with 3 output files: One with the memory image of all structures after the execution of the Queue Management in software, one with a the translation of this file in human readable format and one with a Verilog vector file to be used for the simulation

process. The file which is in human readable format contains for each queue what slots are in or if it's empty. In this way we know how the memory would look like after the specified sequence of instructions.

Then we can start the execution of the simulation with the vector file under the Verilog CADENCE simulation environment and at the end we get a memory image of the MUQPRO I. By using the same routine that turns this image into human readable format we know for each queue what slots are inside. With the unix diff command we can compare the software and hardware results and find possible mistakes. We did this simulation for several hundreds of thousands of instructions to test each QM module and then both together and everything worked fine. The generated vectors are also good examples of how to supply the MUQPRO I with valid instructions.

4.1.10 Cost and performance

The MUQPRO I Queue Manager module has been implemented in an ALTERA FLEX10K50 FPGA in AHDL (Altera Hardware Definition Language) and requires 2198 logic cells and no memory cells. The logic utilization of the device was 76% while the pin utilization was 31%. The clock frequency is 30MHz in worst case analysis as measured with the ALTERA Timing Analyzer tool. The average fan-in is 3.11/4 which means that each cell has 4 inputs on average and 3.11 of them are used (on average). In order to meet the 30MHz timing requirement the logic synthesizer added 30% more logic or 874 extra logic cells. The low pin utilization occurs because we did not find any cheaper packaging for this type of device. The design can fit into a FLEX10K40 device but with clock frequency of 15MHz and 95% logic cell utilization.

4.2 The "fully scalable" implementation

The MUQPRO I implementation has been tuned to serve a specific system with its own characteristics. The main target of this implementation is to sustain the buffering of cells from 5 155Mb/s flows and we would not accomplish that without making useless a portion of the memory space (due to the incorporation of the empty bit into the Tail pointer). However there may be cases that operations are not so time critical and full utilization of the attached memory improves cost/performance ratio. Such a design has few differences regarding the datapath, but the most differences are in the control path. To be more specific the differences in the datapath are:

- One more Base Address and Size register in order to keep the starting address of the memory space that maintains the Empty Bits.
- One extra multiplexor for selecting from the 16-bit word read from the SRAM which is the empty bit that corresponds to the addressed queue.

The extra delay of one cycle in the Enqueue instruction comes from the fact that we cannot read a 16-bit word from the memory and select the appropriate bit through the multiplexor during the same cycle. All the other instructions take the same number of cycles as in the MUQPRO I implementation. The datapath of this implementation is presented in appendix C. As far as it concerns the control path a differences in the scheduling of the memory accesses for each instruction exist, without affecting its complexity. More details can be found in appendix D.

4.3 The "fully parallel" implementation

This implementation was an initial exploratory design to find out possible tradeoffs and the available parallelism. In this implementation each instruction is triggered by one unidirectional

pin and there are two on-chip (inside FPGA) narrow memories, one for the Pointer Memory and one for the Head/Tail table. The Empty Bits are incorporated inside the Head/Tail table.

One word of the Head/Tail table contains the Head, the Tail and the Empty Bit for this queue. This is the reason that 5 bits for the Head and the Tail has been chosen resulting in an 11-bit wide datapath (5 bits for the Head, 5 bits for the Tail and 1 bit for the Empty Bit). This architecture has high-cost and low scalability because it requires one double and another single width memory compared with the MUQPRO I and "fully scalable" implementation that require only one single width memory. The instruction set of this exploratory design is limited to Enqueue, Dequeue, GetFree and RetFree and Init. However this architecture is capable of completing the execution of an Enqueue instruction in 2 cycles, a Dequeue instruction in 3 cycles and the GetFree and the RetFree instructions in 2 cycles without any state-of-the-art pipelining or other smart architectural tricks. The data and control paths are presented in diagrams of appendix E.

5 Software implementation of Queue Management

So far we have designed and evaluated several architectures that perform Queue Management in hardware. However, the rapid development of fast and low-cost μP makes software implementations attractive and scalable solutions for use in embedded systems. Furthermore, we have not seen yet how Queue Management is integrated in real systems. For this reason we first develop software for Queue Management and measure its performance in the - widely used in embedded systems - INTEL i960 RISC μP. To be more specific we measure the time it takes each EnQ/DeQ operation if we only alter the pointers instead of the data with sequential and random input patterns. As an input to an EnQ operation we define the queue ID and a pointer to a buffer while a DeQ operation requires only a queue ID. Then we incorporate these operations in code that performs SAR in the same μP demonstrating its usefulness and showing the percentage of time that Queue Management and data movement takes. Apart from these we compare the various software and hardware implementations and discuss the advantages and disadvantages of each one, resulting in solutions with fair cost/performance ratio as a function of the requirements.

5.1 Programming environment

We used an evaluation board made by CYCLONE with an INTEL i960CA processor attached running at 40MHz. The board was connected to a PC through a PCI interface and a monitor program supplied by INTEL has been used for downloading and executing programs on it. The i960CA processor is capable of executing one assembly instruction every 1 or 2 clock cycles at maximum. All the code has been written in C and compiled with Intel's gcc960 compiler. The executable format was COFF (-Fcoff flag) and the optimization levels exploited were the no-optimization (-O0 flag) and full optimization (-O4 flag). Time has been measured with the benchmarking routines supplied by CYCLONE (bentime()) which provide accuracy of 1 microsecond. The C code implements a Head/Tail table of variable size and a Pointer Table of size 393563 with each entry 32-bits wide. We have chosen the width of 32 bits because it is the basic unit of transfer to and from the i960's memory subsystem. In addition, the table size of 393563 has been chosen because it is close to the maximum available memory of our system configuration (2 MB DRAM). Thus the memory used for the Pointer Table is 1.5MB.

5.2 The EnQ operation with canonical input

In our first experiment we used the minimal code needed to implement an Enqueue operation. For 10^8 iterations we enqueued subsequent cells in subsequent queues. Therefore, in the first iteration we enqueue cell 0 in queue 0, in the second iteration we enqueue cell 1 in queue 1 etc. This is what we call canonical input. When we reach the last queue we start from queue 0 again and we go on and on. The same applies when we reach the last cell. For such a simple pattern the Head/Tail table is referenced sequentially allowing for efficient use of i960's data cache (1KB for the i960CA processor that have been used). We must also mention that the loop code is small enough to fit in the instruction cache: 50 assembly instructions at most, times 32 bits per instruction equals 200 bytes. On-chip instruction cache of the i960CA chip is 1KB organized in two sets of 16 8-word lines. Thus the provided measurements indicate a lower bound for the time that a "real world" EnQ operation would take to run on this processor. The number of assembly instructions executed in each iteration and the number of load and store instructions included in this code segment can be determined by the table presented below:

Optimization level	Number of instructions	Loads	Stores
0	47-50	7-8	4-5
4	20-22	2-3	2-3

In figure 1 we can see the influence of increasing N_{queues} in execution time of each EnQ operation. Clearly, there is a threshold value of about 512 for N_{queues} over which there is no extra time overhead however high is this parameter. This occurs because the Head/Tail table gets large and cannot fit inside the cache, forcing this way the processor to access the Head/Tail table from the much slower DRAM.

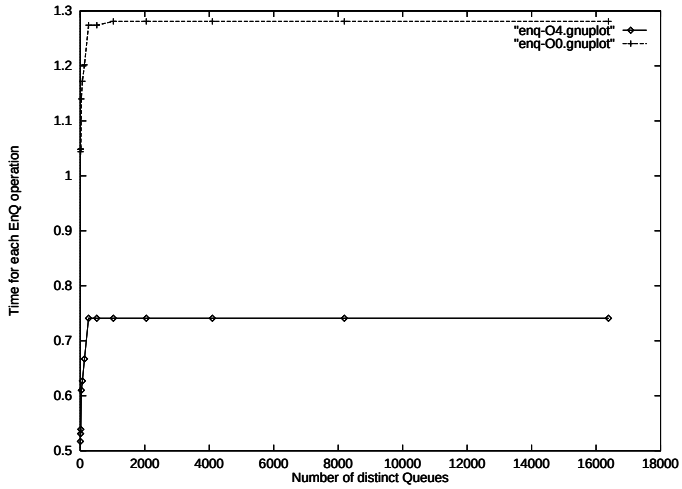

Figure 15: Enqueue serial operations

5.3 Mixed EnQ/DeQ operations with canonical input

The DeQ operation is difficult to be benchmarked because the queues become empty after a few thousands iterations resulting in "dummy" DeQ operations from empty queues. If we let the program run for only a few thousands iterations (to be specific 393563) the accuracy is very low and not satisfying for our purposes. The possibility of using "fake" DeQ operations was considered unrealistic. An implicit way to measure the DeQ operation is through mixed EnQ and DeQ operations. We first fill all the queues with a series of EnQ operations and then empty them by applying an equal number of DeQ operations. By repeating this procedure for a large number of iterations we can safely measure the average time that an EnQ or DeQ operation takes to complete. Knowing the execution time of an EnQ operation with its corresponding parameters from the previous experiment we can approximate the execution time of a DeQ operation (using the same parameters) with fairly high accuracy. The input pattern to this experiment is the same with the one described in the previous subsection.

5.4 The EnQ operation with random input

In real applications Queue Management is executed with random input and this is our next series of measurements. What we expect to see is a slight degradation in speed against canonical

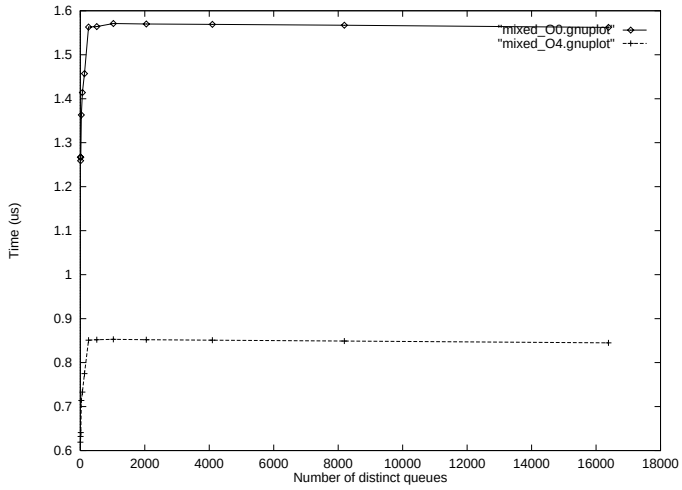

Figure 16: Mixed serial operations

input. In figure 17 the upper curve depicts the time that takes one EnQ operation to complete if we choose randomly a queue and a cell to enqueue through the standard uniform distribution function rand(). A call to rand() has been measured to take 1.7 microseconds and the EnQ operation makes 2 calls to rand(). Therefore the real time a random EnQ operation takes is given by the middle curve of this figure. The lower curve ensures our expectations: A random EnQ operation must be more time consuming than the canonical EnQ but both curves must be close to each other.

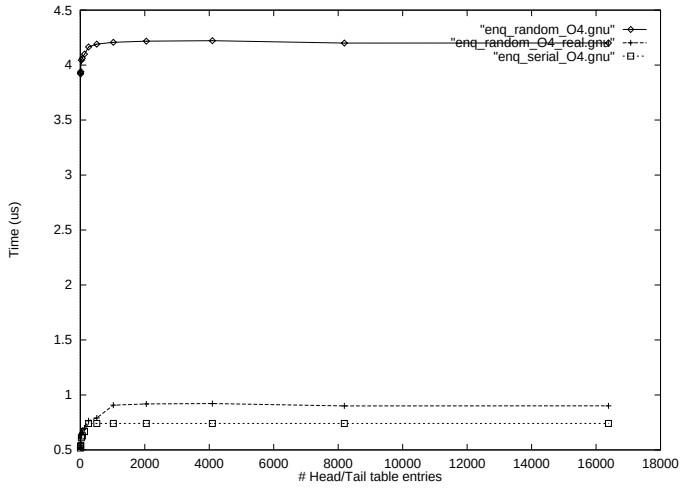

Figure 17: Random Enqueue operations with 4th level optimization

5.5 The SAR model

The basic operation that occurs in an ATM system as we have seen in section 2 is moving cells from one set of queues (the input queues) to another set of queues (the output queues). Segmentation and Reassembly follows this model by adding/removing the proper headers and trailer to/from variable-sized packets as specified by AAL-5 while transferring packets from the

packet (input) queues to the cell (output) queues (by manipulating pointers) and the inverse.

We have implemented the one direction (Segmentation) because the other is equivalent: In Segmentation, in each iteration, one packet is dequeued from a packet queue, padding and trailer are added and the resulting PDU is segmented into 48-byte cells that are enqueued together with the ATM header in the proper cell queue. In Reassembly, in each iteration, one cell is dequeued from a cell queue, ATM header is removed and the cell is enqueued in the proper packet queue. When a cell has a certain bit in the PT field set then it is the last one of the packet and the packet has been fully reconstructed. Then the length of the packet must be determined by reading the corresponding trailer field and padding must be removed leaving the packet body ready for the higher-level (OSI) layer. Differences between the two paths:

- In each iteration in Segmentation the cells of the dequeued packet are moved into the same cell queue while in Reassembly in each iteration each dequeued cell moves in a different packet queue because of multiplexing of cells from different virtual connections over the same physical one (this is not true if the packet size is equal to the cell size).
- In Segmentation the proper padding must be computed based on the size of the dequeued packet while in Reassembly the size of the reconstructed packet must be determined by reading the proper field of the AAL-5 trailer.
- In Segmentation a trailer must be added which is more time consuming than trailer removal (actually to remove a trailer you can simply leave it without reading it).
- In Reassembly the header of each cell must be simply removed and kept once for every connection while adding the header in Segmentation is more time consuming.

We wrote C code that performs segmentation of a packet of variable size (from 1 byte to 64KB) according to the AAL-5 protocol. As we can see in figure 18, we allocate at first two continuous memory blocks, the Packet Bodies and Cell Bodies where the packets and cells physically reside. The Packet Bodies memory block has size equal to MAX_PACKETS*PACKET_SIZE bytes. The Cell Bodies memory block has the same size and both blocks must be a multiple of 4 for alignment in the machine word boundary (a word is 4 bytes in the i960 architecture). The Packet Bodies memory block is divided into smaller blocks - the packets - of size between MIN_PACKET_SIZE and PACKET_SIZE bytes. The PACKET_SIZE is a multiple of the CELL_SIZE which is 48 bytes.

Each of the two memory blocks (Packet Bodies and Cell Bodies) is further divided into smaller blocks of size PACKET_SIZE+8 (8 byte AAL-5 trailer) regarding the Packet Bodies and CELL_SIZE+5 (5 byte ATM cell header) regarding the Cell Bodies. In this way each packet is identified by its base address (lets call it PBP) which is the pointer at the beginning of the Packet Bodies block plus an index number. For example, the packet i resides in the physical location PBP+i*PACKET_SIZE. The same applies for the Cell Bodies memory block. Apart from the physical space that the cell and packet bodies reside we allocate the necessary space for the Queue Management functions EnQ, DeQ, GetFree, RetFree as were presented in section 3 and we keep separate queues for the packets and the cells. We also use a random() function that gives us a uniform distribution of positive integer numbers and the packet processing routine.

5.5.1 The packet processing routine

The packet processing routine performs all the necessary operations over the packet in order to become a series of valid ATM cells. As an input it takes the total number of cells to process. In more detail the following iterative process occur whenever this routine is called:

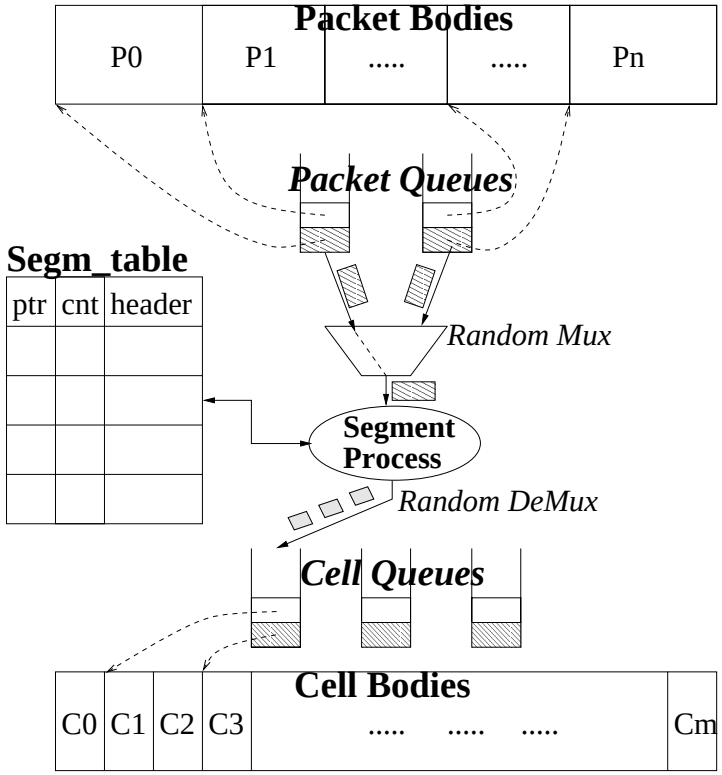

Figure 18: The SAR model

1. A packet queue number is picked up randomly.
2. A packet slot is dequeued from the previously selected queue number.
3. A packet length in bytes is determined (randomly) with lower and upper limits set by MIN_PACKET_SIZE and PACKET_SIZE. A packet size of zero it is not allowed.
4. Depending on the size of the packet the computation of the necessary padding follows. It must be such that the packet's length plus the trailer size (8 bytes long) be a multiple of the cell size i.e. a multiple of 48 bytes. In addition we compute the number of cells contained in this packet.
5. A trailer is generated randomly and copied at the appropriate position of the packet i.e. at the right position of the Packet Bodies memory block.
6. A header is also generated randomly for use when the cells are copied at the proper queues.
7. A cell queue number is picked up randomly.
8. For each 48 byte cell of the packet the following occur:
   (a) A cell slot number is dequeued by using the Queue Manager's GetFree() routine.
   (b) The cell slot is enqueued in the previously selected queue.
   (c) One cell (48 bytes) of the packet is copied into the proper Cell Bodies memory block position determined by the cell slot. For example for the cell slot 4 and packet 2 when only one cell has already been processed, we perform a copy of 48 bytes from Packet_Bodies[2*PACKET_SIZE+CELL_SIZE] to Cell_Bodies[4* (CELL_SIZE+HEADER_SIZE)].

(d) In addition at position Cell_Bodies[4*(CELL_SIZE+HEADER_SIZE)] (continuing the previous example) we copy the pre-computed header of the cell (5 bytes long).

9. The packet slot returns back to its queue with a Queue manager's Enqueue command.

10. A local counter is incremented by the number of cells of the packet that have been processed so far.

This was just one iteration, the loop ends when the total cells processed are equal to the required number of cells that has been given as input. Our intention was not to include any protocol processing or signaling functions in this piece of code but to perform only a primitive set of operations that would be a subset of functions in every ATM system. That's why we did not perform any CRC checking. It would affect execution time greatly by requiring the read of the body (CRC32) and the header (CRC8) of each cell. In addition CRC checking is not something required in all AALs (e.g. AAL-0 does not require any CRC computations).

5.5.2 The iterative process

We call the packet processing routine ITERATIONS times in order to increase accuracy. The argument given as input is PACKET_SIZE*MAX_PACKETS divided by CELL_SIZE. During each iteration the cell queues become full so we must re-initialize them to empty in order for each iteration to finish properly. Of course before we enter the iterative process we must initialize both cell and packet queues, the cell queues to empty and the packet queues filled with all available packet slots.

After this we enqueue the packet back to the queue that was before so that the packet queues never become empty. This is a practical requirement because we repeat the whole process several times and we must have packets to supply it continuously. At the end of this loop we initialize the cell queues back to empty and repeat. The execution time is measured as time per cell. All the cells that contribute to this process are ITERATIONS*PACKET_SIZE_IN_CELLS. By dividing the total execution time with this quantity we get the average time per cell. This measure appears to be the most appropriate because the basic unit of transfer in an ATM system is the cell and we care about the delay of Queue Management for each cell. We expect it to be lower the larger the packet gets.

5.5.3 The parameters

The parameters are the number of packets in the system MAX_PACKETS, the minimum and maximum packet size in bytes MIN_PACKET_SIZE, and PACKET_SIZE+TRAILER_SIZE and the number of packet and cell queues, MAX_CELLQS and MAX_PACKETQS. The parameters that have been chosen as variables are the minimum and maximum packet size and the number of cell queues. The packet size as we will see is a critical parameter together with the number of queues used. We kept the number of packet queues constant in order to be able to measure queue management by varying the number of cell queues. The packet size is between 1 and 1200 cells in size and the number of cell queues is between 2 and 16000. We define the total number of packets in such a way that in each iteration of the iterative process at least 50000 cells are processed. The number of iterations are 400 which means about 5 minutes for every experiment. Keep in mind that all the measures were taken manually because there is no way to store the results and someone must be there to see them on the screen. The total memory used is 4.8MB for the Cell and Packet Bodies memory blocks plus the memory required for the queue management and the rest of the variables. The size of the memory used is considered satisfying because it makes no difference in execution times by accessing 6 or 8MB, provided

that we perform enough iterations of the process. To be certain that the number of iterations is enough we doubled and quadrupled the iterations for a few experiments without noticing any difference in execution times for the first three decimal digits.

5.5.4 Measurements

All measurements present a best-case analysis (a lower limit) of how much time SAR and any other function with the same requirements may take in a typical embedded system with a RISC microprocessor. In the following figures we can see different configurations with variables the cell queues and the packet size. Notice that queue management in the i960 does not change too much with the number of queues, which practically means that 2 queues or 16,000 queues make little or no difference in execution time. This shows the scalability of queue management implemented in software. Enqueue operations in more queues are a bit slower especially when the number of queues gets very large because the processor brings inside the cache blocks of the Head/Tail table that constitute a small portion of this table. On the other hand more queues mean that enqueue in empty queues occurs more often and because of the implementation (the head and the tail are one on the top of the other) writing in subsequent addresses the same data is faster because of the special modes that current DRAM chips offer and the DMA capability of the processor itself. The body of the cell itself (together with the 5-byte header) requires 5.6 microseconds on average for copying from one location to another according to independent measures we did. In addition one call to the random() routine we used takes 0.4 microseconds on average. We also conducted measures on how much time it takes to set the cell queues to empty, depending of course on the number of queues and for 50,000 cell bodies.

Cell Queues	2	16	64	256	1024	2048	4096	8192	16384
Time (us)	49,895	49,895	49,903	49,927	50,024	50,153	50,412	50,928	51,961

On average 1 microsecond per cell is consumed for the body copying. Also remember that each enqueue or dequeue operation takes on average between 0.6 and 0.9 microseconds. For the segmentation of 1 packet that consists of N cells we make the following calls to various routines:

- 16 calls to random() : 2 for the selection of the packet and cell queues, 5 for the header, 8 for the trailer, 1 for the packet length
- N calls to GetFree() (to get N free cell slots)
- 1 call to dequeue() (for dequeuing one packet)
- N+1 calls to enqueue() (to enqueue the N cells of a packet and the packet itself)
- 1 copy of the 8-byte trailer at the end of the packet
- N copies of the 48-byte cells from the Packet to the Cell Bodies memory blocks
- N copies of the header of each cell at the beginning of its body

The results of the measurements for various packet sizes in cells are presented in figures 19, 20, 21, 22 and on the left plot of figure 23. As we can see the per cell cost for this operation drops significantly when we go from 1 cell packets to 10 cell packets or greater. This is because the segmentation process - trailer and header generation and packet length and padding computation - costs more the smaller the packet is and gets its highest value when the packet reaches

the size of one cell. In all measurements when we say packets of 1 cell in size they can vary from 1 byte to 40 bytes so as the packet and the trailer together do not exceed the 1 48-byte cell.

Another observation is that the Queue Management does not affect greatly the execution time of the operation as the number of queues increase. We pay an almost standard cost that changes little with the number of queues. But as the packet size gets larger the time curve becomes more sensitive to changes in the number of queues. This is because the segmentation time per cell drops low and the queue management starts playing a more profound role in the operation. From the 100 cell packets and above there is little change in execution times but there is a small but clear increase tendency with the number of queues. Someone can observe that there are certain points in the curves that spikes occur. This phenomenon can be easily seen in figure 19 and on the left curve where for 11000 queues the execution time jumps 0.5 microsecond up which is a significant variation. These spikes also exist in other figures in smaller scale. The best explanation we found is that the specific input pattern causes much more cache misses leading to that phenomenon. This observation leads us to another conclusion: A processor does not have very accurate guarantees on when the running program will terminate. A question that raises is why we have such a large spike on figure 19 and not in any other figure. The answer is not so easy: We made measurements for packets of 1 cell in size in an ULTRA SparcStation 60 machine and these spikes were disappeared (see on the right plot of figure 23). For the measurements we used the same C code compiled with the GNU gcc compiler and two different optimization levels 0 and 3 (i.e. no optimization and full optimization). Consequently we reached the conclusion that the gcc960 compiler and/or some i960 specific hardware optimization causes this relatively large spike.

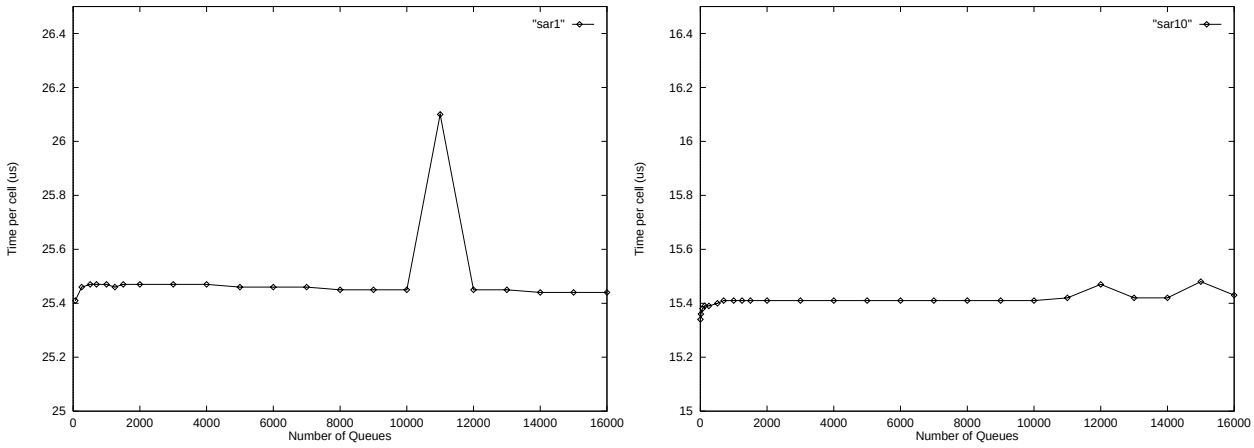

Figure 19: Measurements for 1 and 10 cell packets

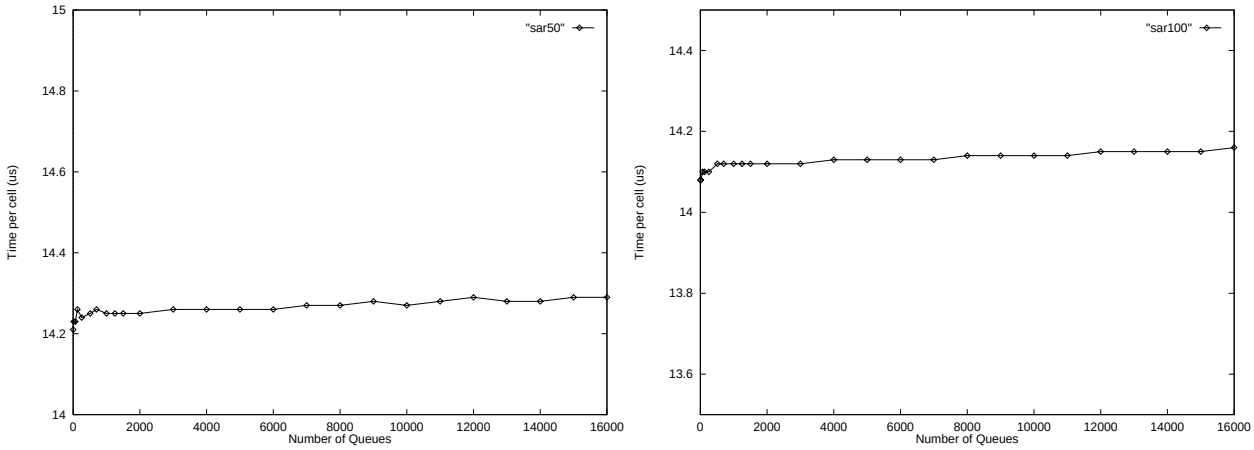

Figure 20: Measurements for 50 and 100 cell packets

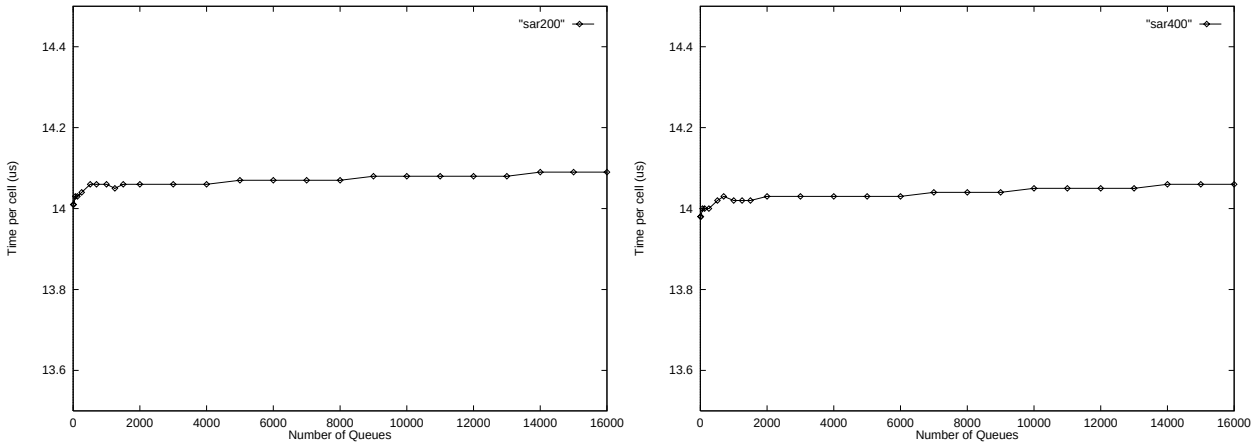

Figure 21: Measurements for 200 and 400 cell packets

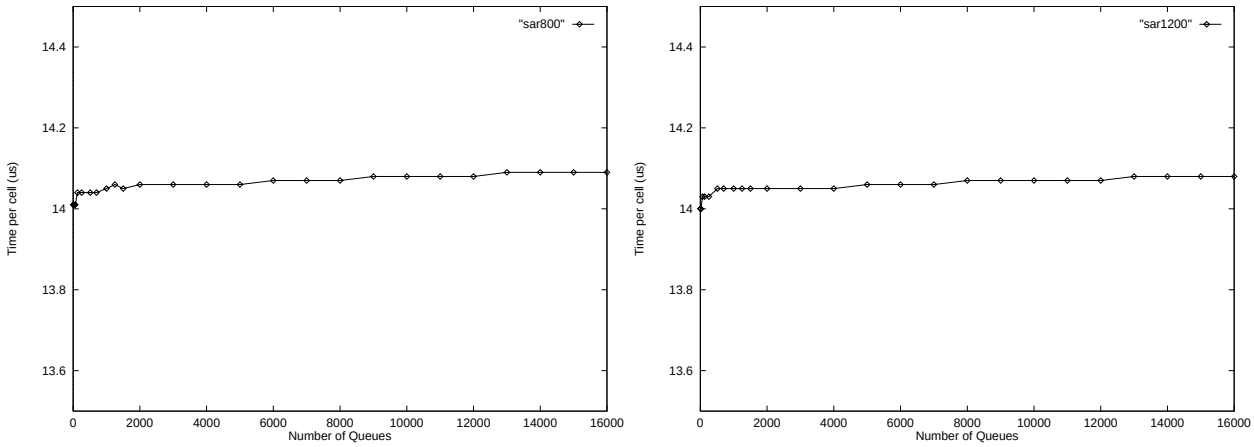

Figure 22: Measurements for 800 and 1200 cell packets

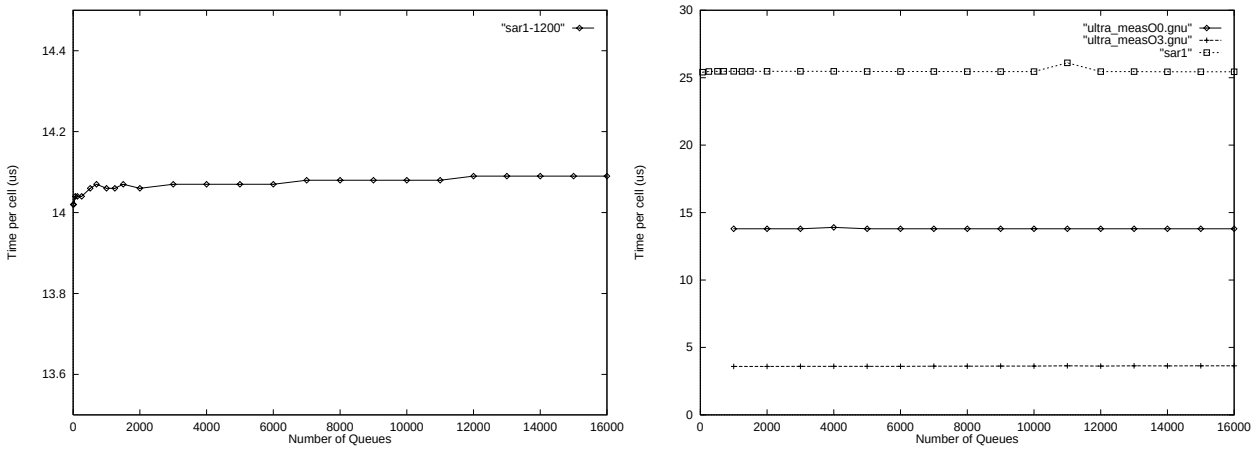

Figure 23: Measurements for 1-1200 cell packets and ULTRA measurements for 1 cell packets

5.5.5 Analysis

In the following tables we analyze the per cell time for Segmentation into the different components. The first table shows the analysis for 1 cell packets while the second table presents what happens for 100-cell packets and above (there are small differences from 100-cell packets and above). For each component (Op. Type) we measure the number of times (Number of Op.) it is executed for processing of 1 cell, and by taking into account a typical execution time for each component (Time/Op.), as measured by independent measurements, we end up with the estimated execution time of it. The various components are the EnQ and DeQ operations (Queue Management) copy of the header, trailer and body of the cell, setting queues to empty after each iteration (in order to be ready for the simulation of the transmission of the next 50000 cells) and the calls to the random() routine. The Total field is the sum of the execution times of all operations according to our computations (done separately for each operation) while the Remaining field is the extra time required to reach the real time measured for Segmentation.

Op. Type	Number of Op.	Time/Op.	Total time(us)
DeQ	1	0.85	0.85 (5.1%)
EnQ	2	0.75	1.5 (9.1%)
Copy	61 bytes	0.11us/byte	6.7 (40.7%)
Empty Queues	1.0	1.0	1.0 (6.1%)
Random()	16	0.4	6.4 (39.0%)
Total	-	-	16.45 (100%)
Remaining	-	-	+8.55 (+52%)

Op. Type	Number of Op.	Time/Op.	Total time(us)
DeQ	$\frac{1}{100}$	0.85	0.0 (0.0%)
EnQ	$1+\frac{1}{100}$	0.75	0.75 (9.9%)
Copy	53 bytes	0.11us/byte	5.83 (76.9%)
Empty Queues	1.0	1.0	1.0 (13.2%)
Random()	$\frac{16}{100}$	0.4	0.0 (0.0%)
Total	-	-	7.58 (100%)
Remaining	-	-	+6.42 (+84%)

Queue Management regards the time it takes to execute 3 operations in the case of 1 cell packets (dequeue a packet enqueue a cell and re-enqueue the packet) and one operation in the case of 100 cell packets (enqueue the cell). The dequeue and re-enqueue of a packet occurs once every 100 cells and represents a small amount of time. The other cases are similar to the one of 100 cell packets and thus there are represented sufficiently.

	Time per cell	Queue Management
1 cell packet	25.0	14.2%
100 cell packet	14.0	9.9%

The general conclusion is that Queue Management takes 10-14% of the CPU time in an optimized ATM system and that number can deviate by 52-84% during dynamic execution in a larger code segment. In other words, Queue Management used as a subroutine of a larger program may take up to 25% of the CPU time which is considered fairly large. The Segmentation routine has a net thoughput on a i960 processor that ranges between $\frac{8*48}{25} = 15.4Mb/s$ to

$\frac{8*48}{14.1} = 27.2Mb/s$ which is considered low and can only serve on the limit very low speed ATM systems with relatively large packets. On the other hand if we use the microprocessor only to execute Queue Management operations and assuming 2 operations per cell we can have a typical net throughput of $\frac{8*48}{0.75+0.85} = 240Mb/s$ or $\frac{8*53}{0.75+0.85} = 265Mb/s$ with the headers. Further discussion and comparisons between different implementations can be found in section 6.

6 Comparisons

So far we have explored two different approaches for Queue Management, one in hardware that has been fully implemented for MUQPRO I and another in software running on an embedded microprocessor (i960). Now it is time to compare the different approaches, balance their advantages and disadvantages and result in conclusions that support each implementation as a function of the requirements. For the hardware implementation we will use the core Queue Management unit of each architecture (MUQPRO I, "fully scalable", "fully parallel"), i.e. the one that implements *one set of queues*. All architectures are targeting the 20MHz frequency in order to keep a size/speed balance and reduce the cost by using smaller devices and their statistics are presented in the following table showing that the required logic is comparable in any case ("Fully scalable" implementation requires some more logic in comparison with MUQPRO I):

Implementation	Pin util.	Logic util.	Avg Fan-in	Extra cells
Fully scalable @ 20MHz	57%	87%	3.15/4	27%
MUQPRO I @ 20MHz	57%	82%	3.21/4	28%
MUQPRO I @ 30MHz	57%	85%	3.14/4	32%

A typical unidirectional system requires one enqueue instruction for every cell arrival or one dequeue if it's a cell departure. We omit the getfree and retfree instructions assuming that we can do something smart and not requiring them very often (e.g. use the dequeued slots to enqueue a new cell that just arrived). The difference between the hardware implementations results from the one cycle difference of the Enqueue operation and thus we will measure the throughput taking into account the receive side of an ATM system (i.e. Enqueuing received cells). However, the common physical layers used for ATM (e.g. SONET) are bidirectional and the presented numbers drop at half. The throughput of the MUQPRO I(20 MHz) is between $\frac{48*8\ bits}{50ns*6cycles} = 1.28Gb/s$ and $\frac{48*8\ bits}{50ns*4cycles} = 1.92Gb/s$ depending on the network traffic and if we can fully utilize the device. The same numbers for the fully scalable architecture are $\frac{48*8\ bits}{50ns*7cycles} = 1.09Gb/s$ and $\frac{48*8\ bits}{50ns*5cycles} = 1.53Gb/s$. Considering the MUQPRO I implementation at 30MHz which also fits in a FLEX10K20 device we can reach a throughput between $\frac{48*8\ bits}{33ns*6cycles} = 1.94Gb/s$ and $\frac{48*8\ bits}{33ns*4cycles} = 2.9Gb/s$ depending on the packet size and the network traffic while for the software the same numbers, considering the typical execution time of an Enqueue operation as 0.75μs, is $\frac{48*8\ bits}{0.75\mu s} = 512Mb/s$. The table bellow summarizes the Queue Management performance of all hardware implementations and the implementation in software. All numbers consider the best and worst case of the enqueue and dequeue operations and we do not consider any getfree or retfree instructions. The throughput is measured without the headers i.e. net throughput.

i960 @ 40MHz	512 Mb/s
fully scalable @ 20MHz	1.09 Gb/s - 1.53 Gb/s
MUQPRO I @ 20MHz	1.28 Gb/s - 1.92 Gb/s
MUQPRO I @ 30MHz	1.94 Gb/s - 2.9 Gb/s

In addition there must be some hardware that transfers the cell bodies and headers at a memory with sufficient throughput to keep up with these speeds. The MUQPRO I implementation offers scalability as we can always move to a larger and more expensive device and reach higher speeds. The highest we reached for a reasonable pin and cell utilization was 32MHz at a EPF10K40 device but as FPGA technology advances higher speeds are not surprising to be

achieved. Another parameter that has not been mentioned is the number of queues, all the hardware discussed so far has 16-bit datapath and can address up to 64K different positions shared between the various data structures. Both hardware implementations are parameterized in such a way that wider datapaths can come up with the change of a few parameters (the datapath's width and the logarithm of it) and with re-compilation it can fit to a newer and faster device. If we implement Queue Management and header and body storing and retrieving in hardware and the rest of the ATM system in software, the throughput of the system assuming zero overlay or zero interface delay between the processor and the hardware is between $\frac{48*8\ bits}{25-5.6-0.75\mu s} = 20.59Mb/s$ and $\frac{48*8\ bits}{25-(5.6+0.75)*1.84\mu s} = 28.84Mb/s$ in comparison with that of a plain software implementation ($\frac{48*8\ bits}{25\mu s} = 15.36Mb/s$) which practically means a speedup of 1.34-1.88. Therefore data transfer and queue management in hardware can provide a significant system performance which can extent further if header generation is also performed in hardware.

In conclusion a microprocessor that executes the basic functions of an ATM system has poor performance (as expected) with throughput that can be as low as 15Mb/s and not higher than 27Mb/s. Hardware add-ons are required in order to achieve higher speeds and several hardware implementations were presented with more preferable that of MUQPRO I at 20MHz and 30MHz that fits into a FLEX10K20 ALTERA FPGA with reasonable cost. On the other hand the fully scalable implementation makes full utilization of the memory but with greater complexity and lower scalability in speed.

A Functionality and timing diagrams for each instruction

In this appendix we will analyze the available commands of the Queue Manager, that cover all the required functionality. In the presented timing diagrams clk is the clock, clr is the reset signal, op[4..0] is the opcode and the valid, ready and e_ne signals follow. The swen signal represents the SRAM write enable signal that performs a write when high and a read when low. The Saddr signal is the address to the SRAM, the sdio signal is the bidirectional data to the SRAM, the dio signal is the Data to the master device, the first qmdpfs is the Finite State Machine (FSM) of the first set of queues and the second one the FSM of the second set of queues. The ddrive signal when high shows that the second QM module drives the I/O pins while the first QM module drives the I/O pins otherwise. The flebit signal is the Free List Empty Bit.

In figures 24 and 25 we can see how many cycles it takes for each instruction to finish and when a new instruction can be issued. Each square represents one cycle and on the right of each sequence of squares we can see which instruction represents. Notice that after a retfree instruction we cannot issue a new retfree or getfree instruction because the Free List Head and Tail registers do not have the right value yet. Missing the right next cycle to issue a getfree or retfree instruction, we are forced to issue a new instruction of this type 2 cycles later (at cycle 3). This is a limitation set up by our architecture so as to accelerate more time consuming instructions like the enqueue and dequeue. Another characteristic of the architecture which is also reflected in figures 24 and 25 is that *no more that 2 instructions can be on the system* working at the same cycle. In other words the scenario of having an enqueue instruction followed by a retfree instruction to an empty free list, followed by an enqueue instruction the right next cycle will not execute correctly.

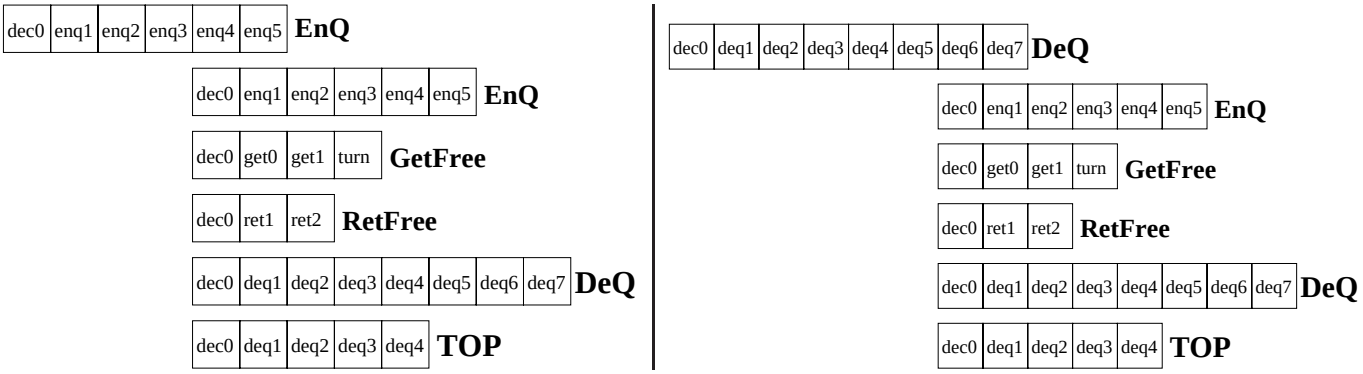

Figure 24: Issuing instructions right after an enqueue or dequeue instruction

A.1 The enqueue instruction

This command takes 2 arguments the queue number and the Slot number and returns whether the queue was empty or not. It enqueues the Slot number on the back of the queue specified by the queue number. Enqueue succeeds even for empty queues for the purposes of MUQPRO I (no translation between VP/VCs and queue numbers exists). The EnQ implementation in C follows:

```
unsigned int
EnQ(unsigned int *HTm,unsigned char *HTe,
        unsigned int *PMm,unsigned int Slotn,unsigned int Qn)
```

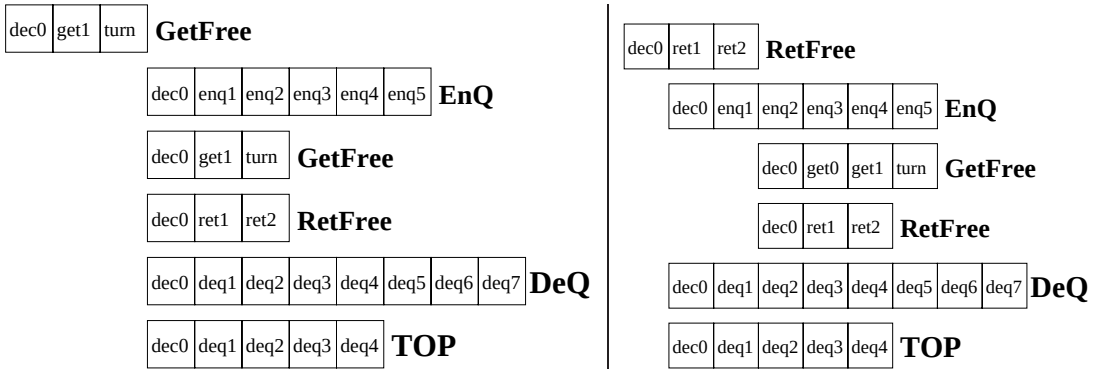

Figure 25: Issuing instructions right after a getfree or retfree instruction

```
{
    unsigned int pTail;

    if (HTe[Qn]==1)
        {
            HTe[Qn]=0;
            HTm[Qn<<1]=Slotn;
            HTm[(Qn<<1)+1]=Slotn;
        }
    else
        {
            pTail=HTm[(Qn<<1)+1];
            PMm[prevTail]=Slotn;
            HTm[(Qn<<1)+1]=Slotn;
        }
}
```

The number of micro-operations that this function includes are 4 as shown by the Code: Read HTempty and check whether it is high or low, and 3 more in each of the two cases. Given the 2-stage pseudo-pipeline these operations can be scheduled in the following way (also presented in figure 26): An example of enqueue in an empty and a non-empty queue for 8 slots and 4 queues is presented in figures 27. During cycle 0 we fetch the queue number (Qn) while the other two stages remain idle.

During cycle 1 we fetch the slot number that will be enqueued (Slotn) and we calculate the address to access the specific queue's empty bit. Since the empty bit is packed together with the tail pointer, in our implementation, we must compute the address of the (2*Qn + 1) element of the Head/Tail table structure. This is the Base Address of the structure plus the effective address (2*Qn + 1). We do not access the external memory (SRAM) during this cycle.

During cycle 2 we access the SRAM to read the empty bit together with the tail of this queue and we can compute another address. However we do not know whether we will read or write the tail pointer at the following cycle because only at the end of the present cycle we will have read the empty bit. Consequently the Address/Data Calculation stage remains idle.

Knowing whether the queue was empty or not, during cycle 3, we compute the address of the tail pointer inside the Head/Tail table. Independently of the previous state of the queue (empty or not) we also compute the data to be written. These are the slot number together

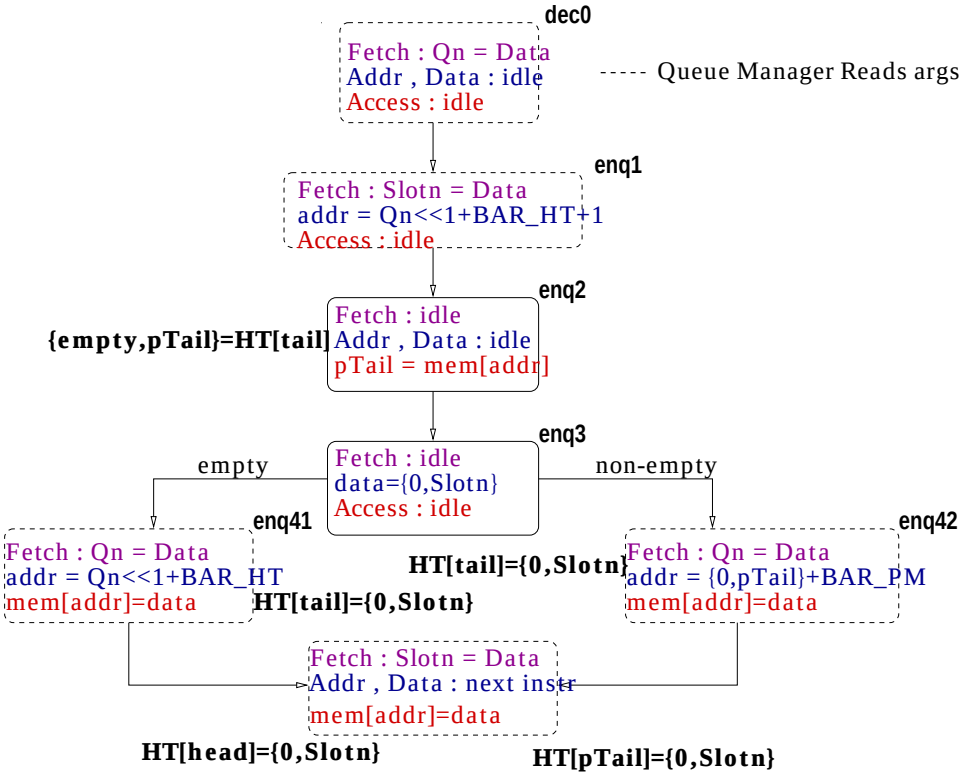

Figure 26: Schedule for the EnQ command

with the empty bit having a 0 value. In this cycle also the Address/Data Calculation stage remains idle. The Ready signal is driven high to inform the master device that next cycle it can provide a new instruction.

During cycle 4 either of two things occur, depending on the previous state of the queue:

The queue was not empty: Having read the previous tail pointer (at cycle 2) we write the new value of the tail pointer. In addition we calculate the address of the previous tail element which is the previous tail plus the Base Address of the Pointer Memory structure (where the pointer field of each cell buffer resides). The data remain the same that were calculated in the previous cycle. We also drive the Valid signal high and the E/NE signal low to inform the master device that the queue was not empty and will remain this way.

The queue was empty: We write the same data at the same address as in the case that the queue was not empty. The difference is that we calculate the head pointer address which is the Base Address of the Head/Tail table structure plus 2*Qn. We also drive the Valid and the E/NE signal high to inform the master device that the queue was empty but after the completion of this instruction will not be any more.

Apart from these during this cycle a new command can be fetched together with its first argument.

During cycle 5 a write occurs at the address computed during cycle 4 and the address and data computed depend on the instruction issued at cycle 4.

Generally one new Enqueue instruction can be issued every 4 cycles which is the optimal given a narrow memory that keeps all the necessary data structures. A stand-alone Enqueue instruction requires 6 cycles to complete but during the last 2 cycles another instruction can

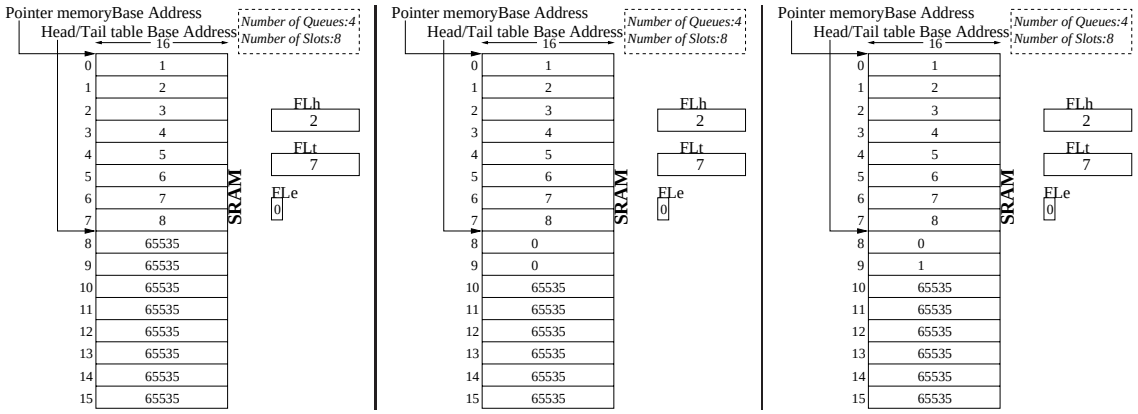

Figure 27: Initial state of the queues, enqueue slot 0 in queue 0, enqueue slot 1 in queue 0

start its execution in parallel thanks to this pseudo-pipeline scheme. The delay can be reduced by widening the interface between the Queue Manager and the master device so as to accept 2 arguments and an instruction in one cycle. But this adds to the cost and it is not required as long as the master can keep the Queue Manager occupied all the time, which is the case for MUQPRO I. Notice also that only from cycle 4 we can issue a new instruction and not from cycle 5. This seems as a restriction at first glance but it does not affect the functionality of the architecture and also simplifies the control path greatly.

The timing diagrams for the two different cases of Enqueue, the first depicting an Enqueue on a non-empty queue (figure 28) and the second one an Enqueue on an empty queue (figure 29), follow:

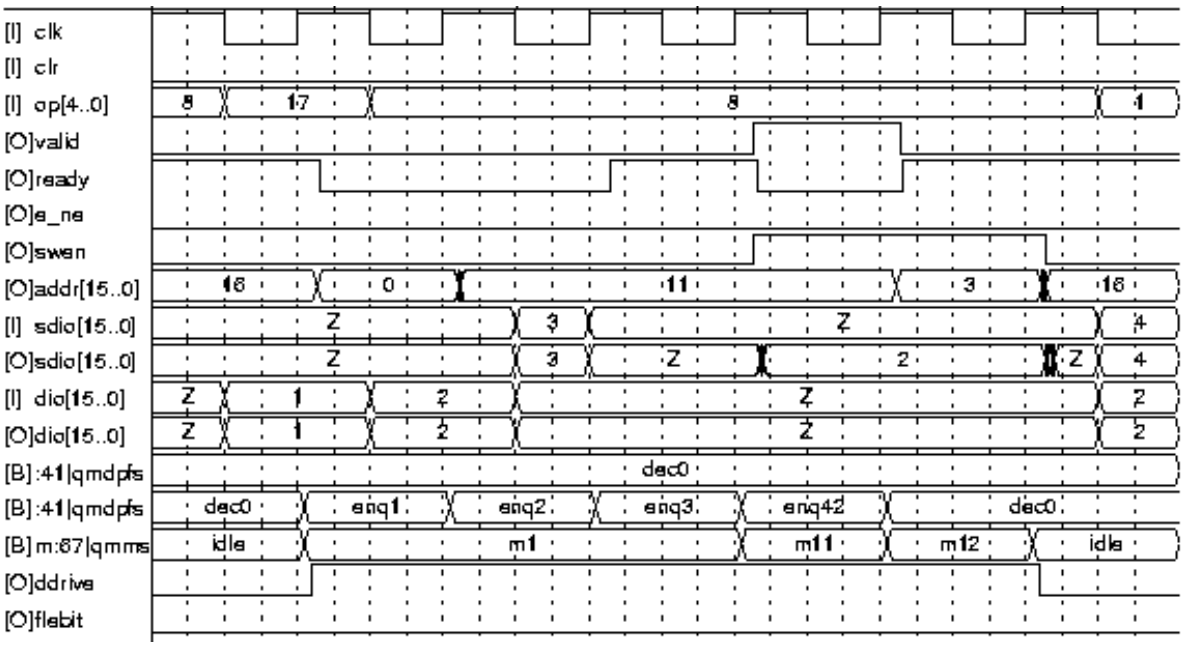

Figure 28: Enqueue instruction on a non-empty queue

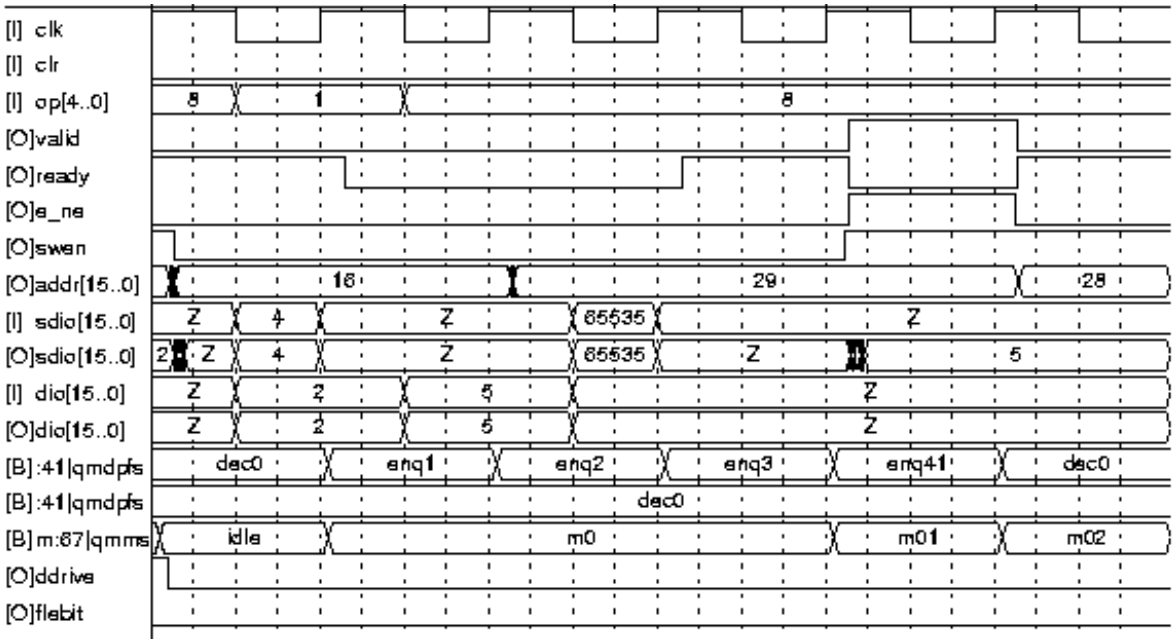

Figure 29: Enqueue instruction on an empty queue

A.2 The dequeue instruction

This instruction takes 1 argument, the queue number, and returns the Slot number i.e. the first entered element of the queue and whether the queue has become empty or not. The DeQ implementation in C follows:

```
unsigned int
DeQ(unsigned int *HTm,unsigned char *HTe,
        unsigned int *PMm,unsigned int Qn,unsigned int *valid)
{
    unsigned int pHead,pTail,pPM,ret_val=0;

    *valid=0;
    if (HTe[Qn]==1)
        {
            *valid=0;
        }
    else
        {
            pHead=HTm[Qn<<1];
            pTail=HTm[(Qn<<1)+1];
            ret_val = pHead;
            *valid = 1;
            if (pHead==pTail)
                HTe[Qn]=1;
            else
                {
                    pPM=PMm[pHead];
                    HTm[Qn<<1]=pPM;
                }
        }
    return ret_val;
}
```

This operation breaks down into 6 micro-operations if the queue is not empty and it has more than one elements, 5 micro-operations if the queue is not empty and has exactly one element and 2 micro-operations if the queue is empty. An example of the two cases of dequeue (dequeue from a queue with more than one elements and dequeue from a queue with one element and dequeue can be seen in figure 30.

During cycle 0 we fetch the queue number and decode the instruction while the other stages remain idle.

During cycle 1 we fetch the slot number that will be enqueued (Slotn) and we calculate the address to access the specific queue's empty bit. This address is computed in the same way as in the EnQ instruction.

During cycle 2 we calculate the address of the head pointer of the queue which is the Base Address of the Head/Tail table plus 2*Qn and we also access the tail pointer of the queue (read from SRAM).

During cycle 3 we compute no address and we read the head pointer of the queue if the queue was not empty or return to idle state otherwise.

During cycle 4 we drive the Data bus with the head pointer we have read in the previous cycle and we drive the Valid signal to 1. In this cycle we also determine if the head and tail

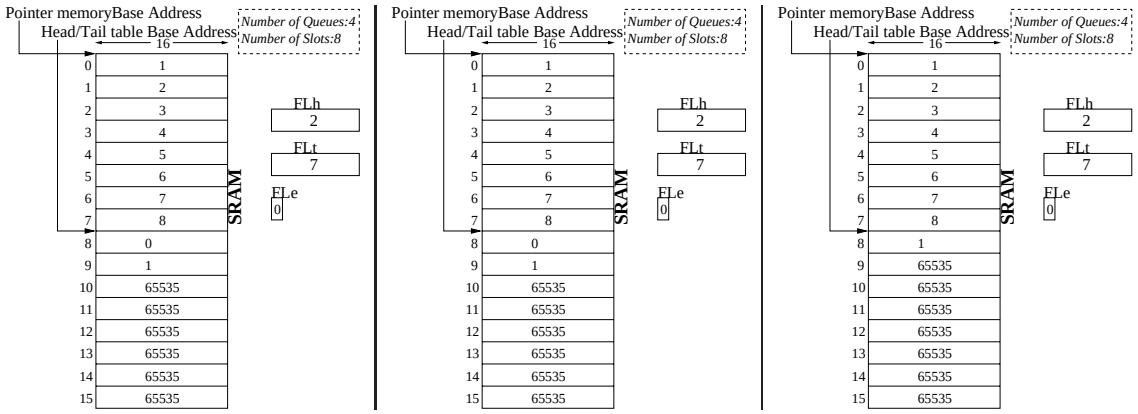

Figure 30: Initial state of the queues, dequeue from queue 0, dequeue from queue 0

pointers we have already read are equal or not. We could compute during this cycle the head element address based on the head pointer we have already read but this would not improve our latency due to the fact that the bus should not be driven during next cycle by the master while the Queue Manager drives it during this cycle and no turnaround cycle exists between. This is a good design methodology rather than a strict requirement.

During cycle 5 either of 2 things may occur depending if the head and tail pointers were found equal:

Head and Tail pointers were found equal: We drive the E/NE signal together with the Valid signals to 1. We also compute the address of the tail pointer (in the same way we mentioned in the Enqueue instruction) at the Head/Tail table and set the most significant bit of the computed data (which is the empty bit) to high. We also drive the Valid and the E/NE signal high to indicate that the queue became empty.

Head and Tail pointers were not found equal: We compute the address of the head element which results from the head pointer already read plus the Base Address of the Pointer Memory structure. We also drive the Valid signal high and the E/NE signal low to indicate that the queue has not became empty.

In both cases above no memory accesses occur. The Ready signal is driven high to indicate that at the next positive edge of the clock a new instruction can be issued.

During cycle 6 one of two things occur:

Head and Tail pointers were found equal: We simply write the pre-computed data at the pre-computed address and fetch a new instruction together with its first argument (if it has one). The data written are actually the empty bit set to high to indicate that the queue became empty.

Head and Tail pointers were not found equal: We compute the address of the head pointer which results from 2*Qn plus the Base Address of the Head/Tail table. In addition we read the head element with the address computed in the previous cycle.

During cycle 7 we write what has been computed in previous cycle in case Head and Tail pointers were not found equal or otherwise we do not access memory. We also compute an address specified by the instruction issued at cycle 6 (if one has been issued) and we fetch a second argument if one exists (e.g. in the case of the Enqueue instruction).

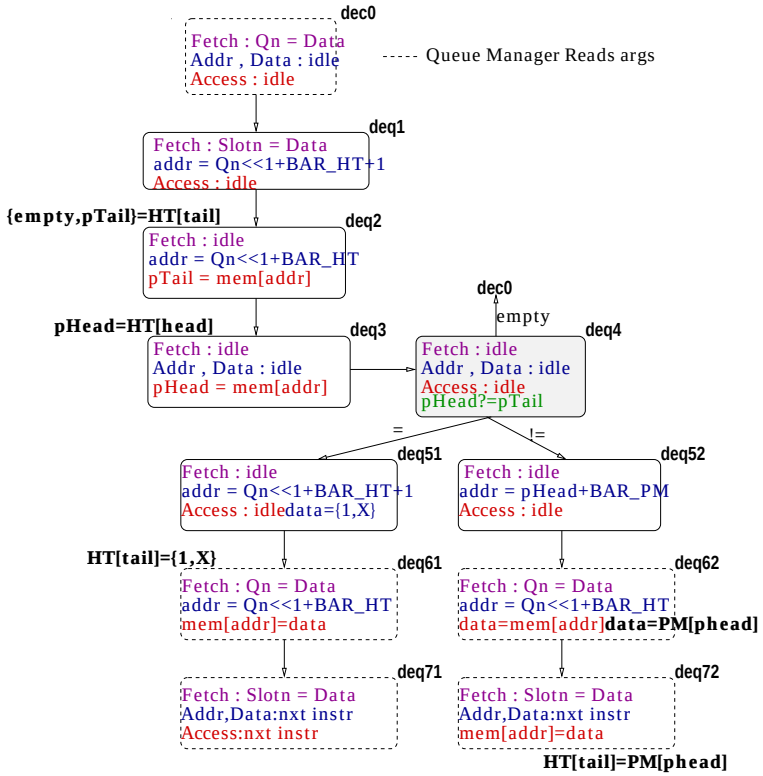

Figure 31: Schedule for the DeQ instruction

A few important hints: First notice that the Dequeue instruction has one case that something read from the memory must be written the right next cycle (cycles 6 and 7). This causes no problem because what is read from memory goes into an edge-triggered register it is available for write back to memory from the beginning of next cycle. What would cause a serious problem is what read from memory in one cycle be used as an address during the right next cycle. This is because address calculation is a stage that requires and addition and a "passing" from a large multiplexor which is a critical path in the design. A requirement we took into account is to avoid the driving of the bus by the Queue Manager and the controller in subsequent cycles. Given these restrictions we ended up with this scheduling that allows the issue of one new Dequeue instruction every 6 cycles and a reply (answer) the fourth cycle after the issue of a Dequeue. Everything discussed above is displayed in figure 31. The timings in the three different cases of a Dequeue instruction (i.e. dequeue from an empty queue, dequeue from a queue with only one element and dequeue from a queue with more than one elements) are presented in figures 32,33,34.

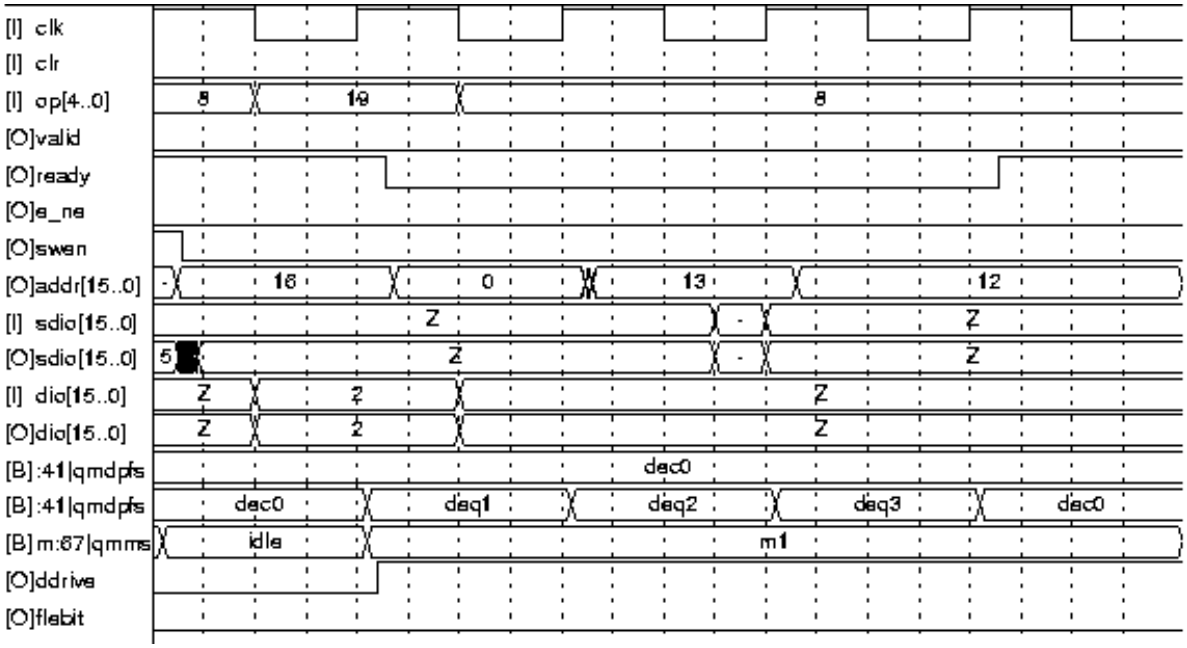

Figure 32: Dequeue instruction on an empty queue

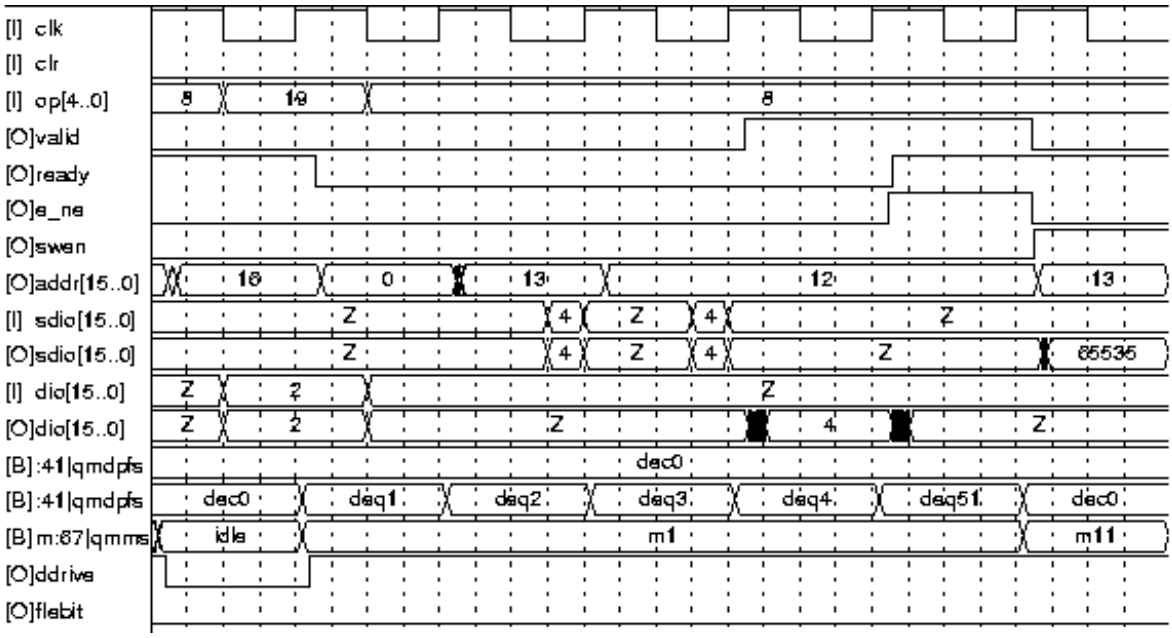

Figure 33: Dequeue instruction on a queue with exactly one element

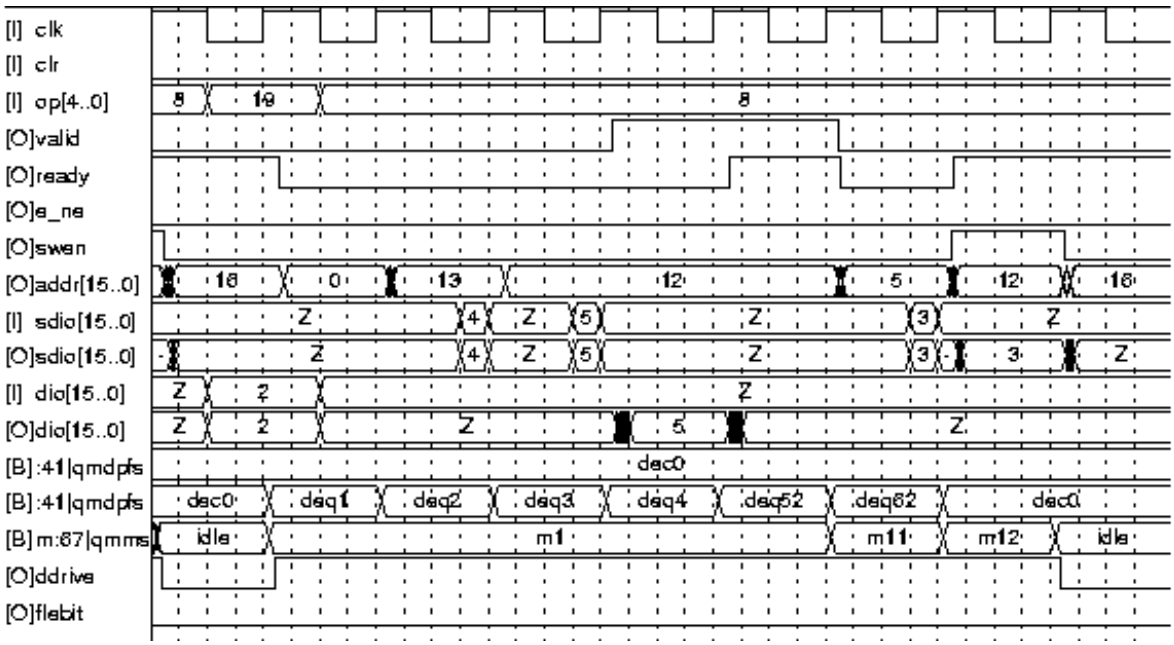

Figure 34: Dequeue instruction on a queue with more than one elements

A.3 The getfree instruction

GetFree takes no arguments and returns a free Slot number i.e. an available address at the cell bodies memory. The C code implementation follows:

```
unsigned int
GetFree(unsigned int *FLh,unsigned int *FLt,unsigned char *FLe,
        unsigned int *PMm,unsigned int *valid)
{
     unsigned int pPM,ret_val;

     *valid = 0;
      if (*FLe==1)
      {
        ret_val=0;
      }
      else
      {
        ret_val=*FLh;
        *valid = 1;
        if (*FLh==*FLt)
          *FLe=1;
        else
        {
          pPM=PMm[*FLhead];
          *FLh=pPM;
        }
      }
      return ret_val;
}
```

In figure 35 we can see an example of getting a free slot from a Free List with more than one elements and then getting a free slot from a Free List with exactly one element.

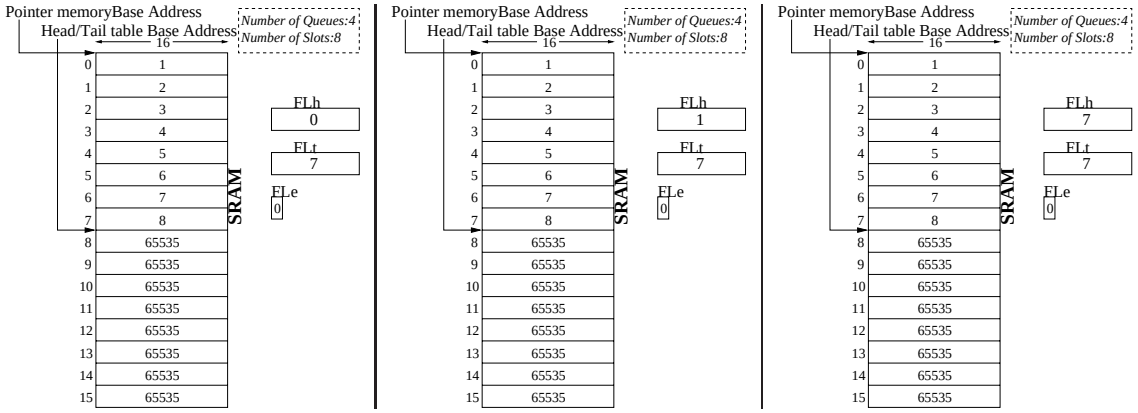

Figure 35: Initial state of the queues, getfree, getfree

During cycle 0 we compare the register that points at the head of the Free List structure (FLh) with the one that points at the tail (FLt) and we compute the address of the head

element. There are three choices: Either Free List is empty, or it is not empty and the head and tail pointers are equal or not equal.

During cycle 1 we drive the E/NE signal with the result of the comparison between FLh and FLt and we execute one of these:

Free List is empty: We return to idle state.

Free List has exactly one element: Flip the Free List empty bit to high and drive the Data bus with the Free List head value.

Free List has more than one elements: Drive the Data bus with the Free List head value and read the head element whose address has been computed in the previous cycle. The data that are being read during this cycle are written at the Free List head register.

During cycle 2 nothing is done, we use this cycle only as a turnaround between subsequent drives of the Data bus by the controller and the Queue Manager.

All the operations are presented in figure 36 that follows:

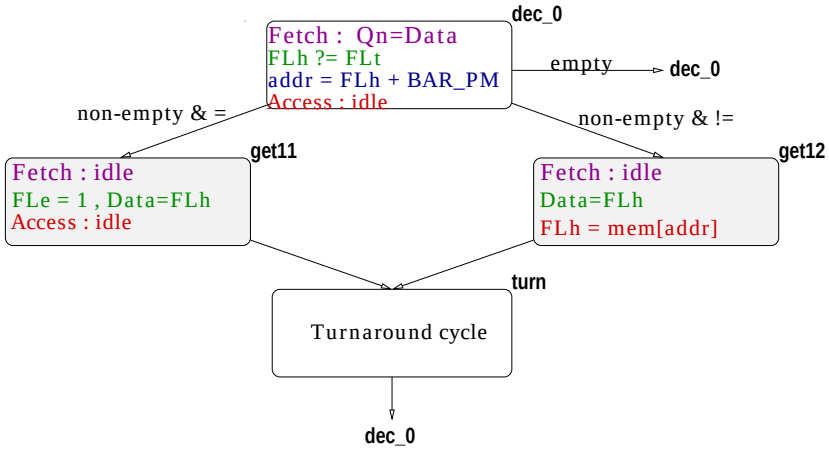

Figure 36: Schedule for the GetFree instruction

The timings in the three different cases of a GetFree instruction (i.e. getfree from an empty queue, getfree from a queue with only one element and getfree from a queue with more than one elements) are presented in figures 37,38,39.

A.4 The retfree instruction

It takes as an argument a Slot number and returns it internally to the Free List structure. The C code that implements it follows:

```
unsigned int
RetFree(unsigned int *FLh,unsigned int *FLt,unsigned char *FLe,
        unsigned int *PMm,unsigned int Slotn)
{
  unsigned int ret_val;

  if (*FLe)
    {
      *FLh=*FLt=Slotn;
```

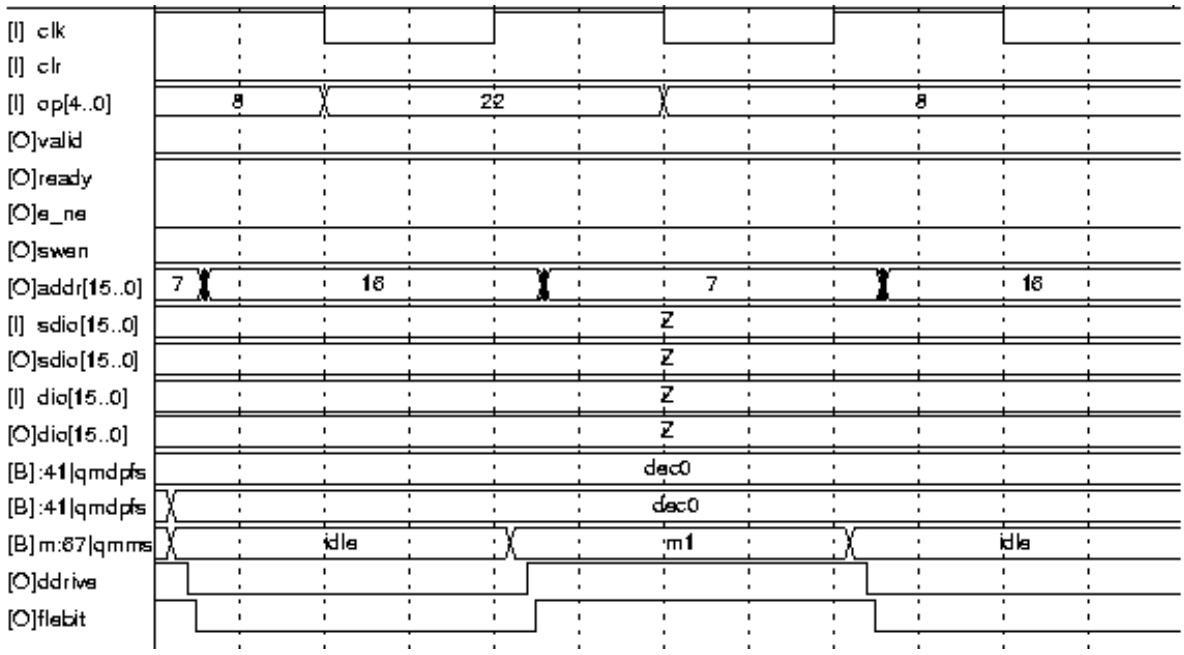

Figure 37: GetFree instruction on an empty queue

```
      *FLe=0;
      ret_val = 0;
    }
  else
    {
      ret_val = 1;
      PMm[*FLtail]=Slotn;
      *FLt=Slotn;
    }
  return ret_val;
}
```

In the following figure 40 we present the various structures in the case we return a free slot to the Free List and the List is empty and in the case we return a free slot to the Free List that is not empty.

During cycle 0 we fetch the slot number and decode the command while the other stages remain idle.

During cycle 1 we can issue a new command together with its first argument and execute one of the following:

Free List was empty: Load Free List head and tail registers with Slotn and compute a new address which is the address of the head element of the Free List (Base Address of Pointer memory plus Slotn).

Free List was not empty: Compute the address of the tail element of the Free List (Base Address of Pointer memory plus Free List tail register) and load Free List tail register with Qn. In addition we load the data register with Slotn.

During cycle 2 the second argument of the new command (issued in cycle 1) can be fetched and we write in the SRAM the data at the address both computed at the previous cycle. This

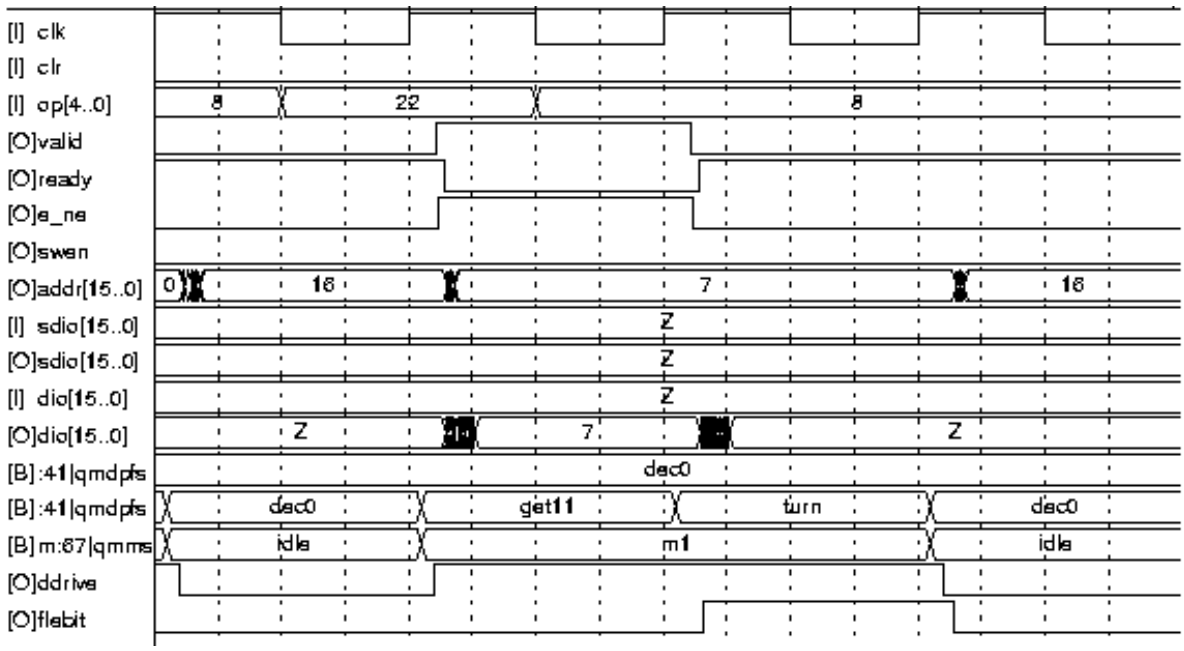

Figure 38: GetFree instruction on a queue with exactly one element

is the case if Free List was not empty. Otherwise no write occurs.

All the operations are presented in figure 41 that follows:

The timings for the two different cases of retfree, the first depicting a retfree on a non-empty queue (figure 42) and the second one a retfree on an empty queue (figure 43), follow:

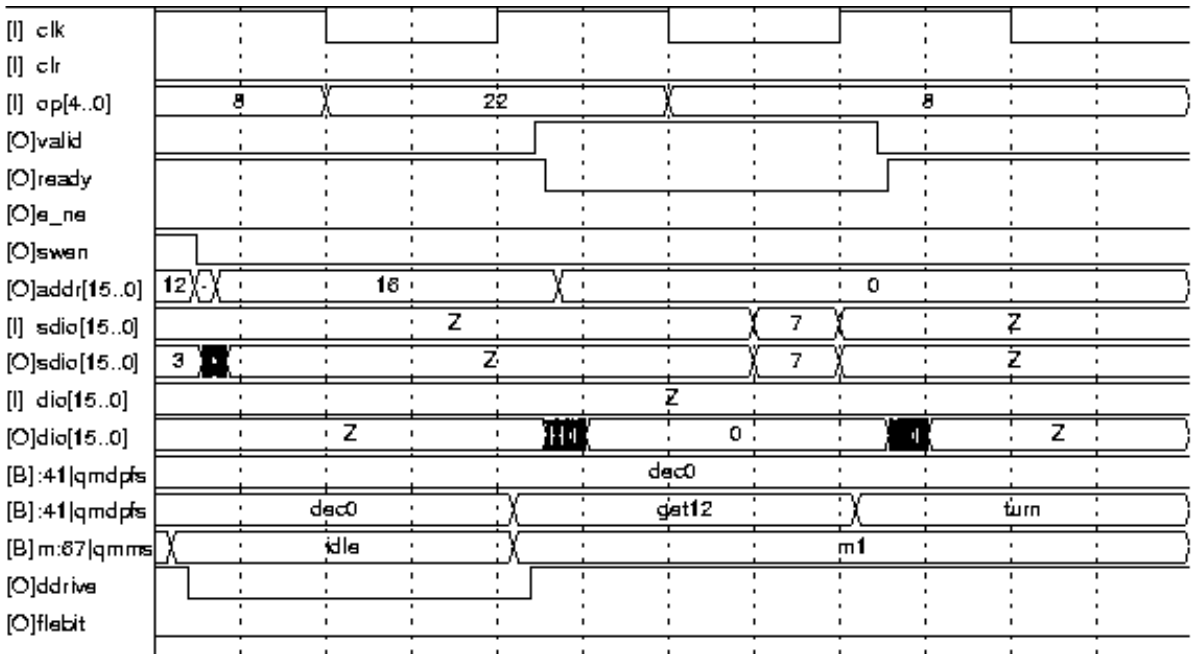

Figure 39: GetFree instruction on a queue with more than one elements

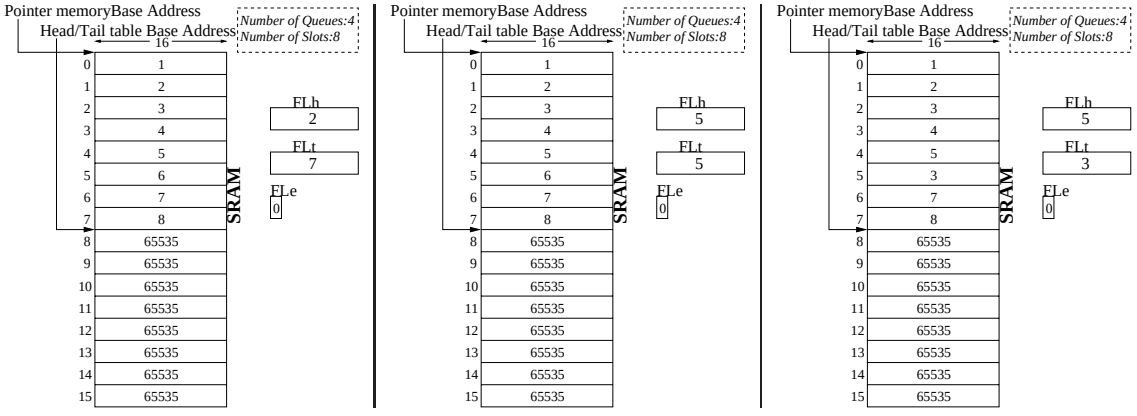

Figure 40: Initial state of the queues, retfree slot , retfree slot

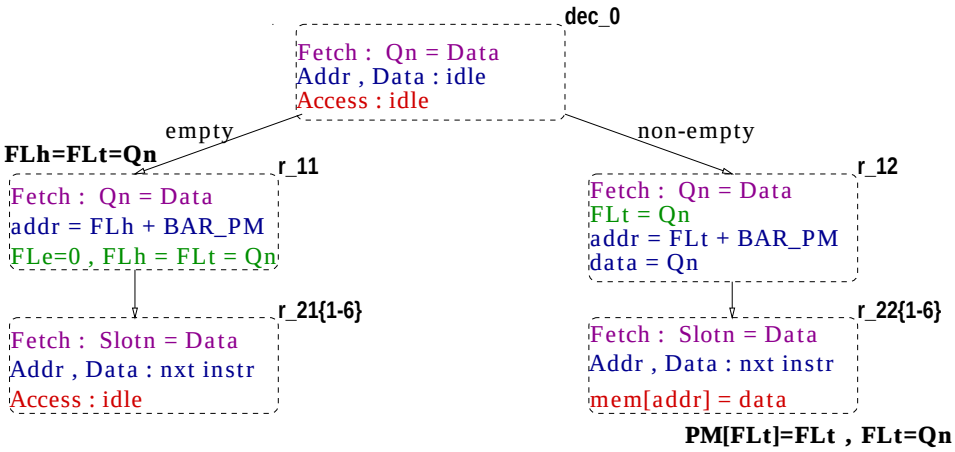

Figure 41: Schedule for the RetFree command

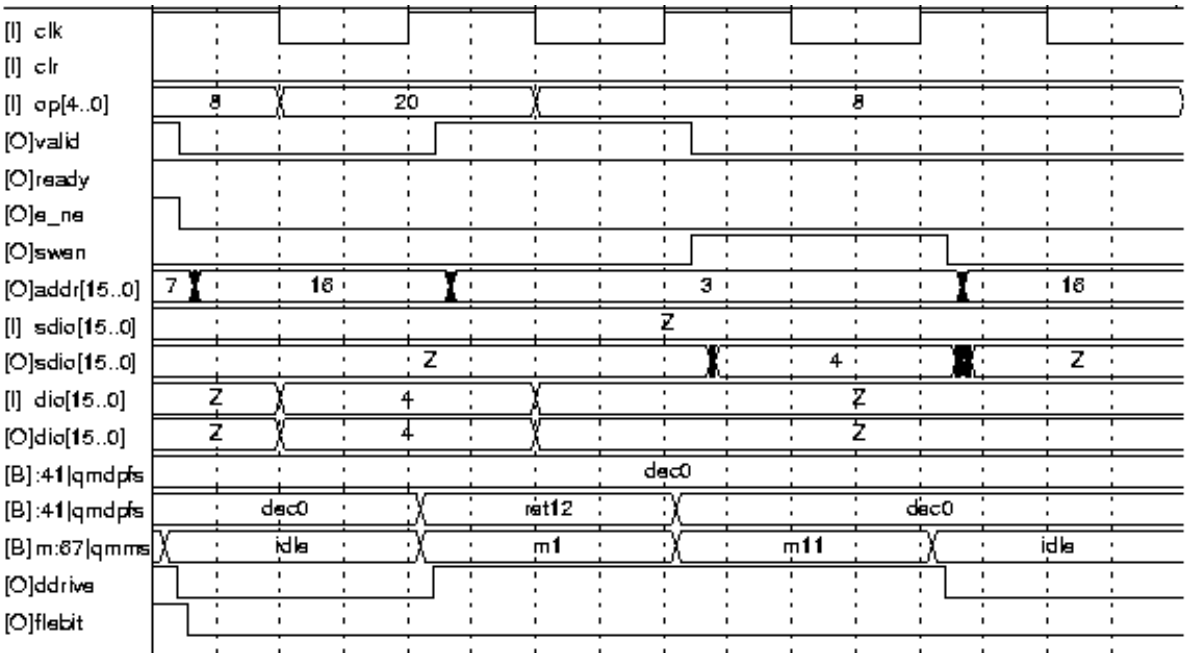

Figure 42: Retfree instruction on a non-empty queue

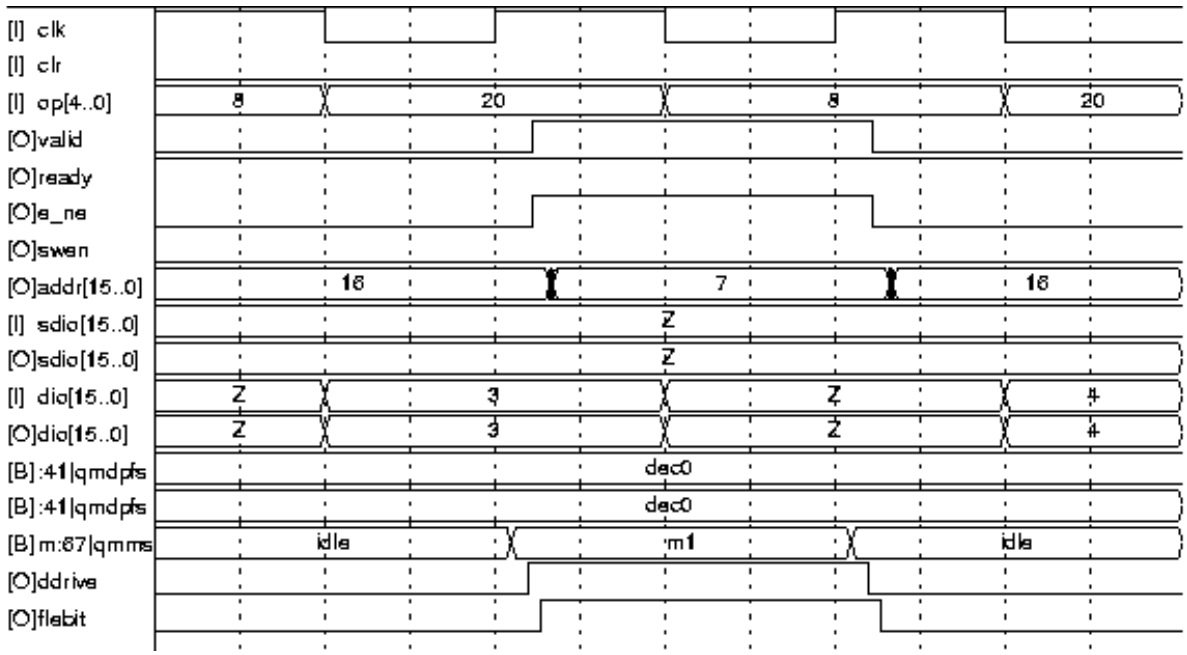

Figure 43: Retfree instruction on an empty queue

A.5 Initialization

The initialization process is required to set all the structures at the proper starting values. In other words all structures must have a dedicated address space inside the SRAM, the Free List structure must be full with all available slots and all the queues must be marked empty. The Init instruction takes 4 arguments starting at cycle 1: The Base Address of the Pointer Memory, the size of the Pointer Memory, the Base Address of the Head/Tail table and the size of the Head/Tail table in this order. No hardware exists for checking that the addresses passed to the Queue Manager are valid. A small C program is provided to do this job and help in the computation of these addresses. The init instruction is very time consuming because it has to write all the Pointer Memory in order to link the slots in a long chain and make Free List head register point at 0, the Free List tail register point at N-1 and drive the Free List empty flip-flop low. In addition this instruction writes all the Head/Tail table with 1's so as all the queues are empty at startup. In figure 44 we can see an initialization example for 4 queues and 8 slots while in figure 45, 46, 47 we can see the timing diagram of initialization process for the same number of queues and slots.

A.6 Read instruction

Read takes 1 argument which is the address of the SRAM to be accessed and returns the content of this address. It is only used for debugging purposes. It has no optimizations to keep the control path simple and avoid aggravating the complexity of the Queue Manager. The timing diagram of this instruction is presented in figure 48 that follows.

A.7 Write instruction

Write takes 2 arguments which are the address of the SRAM to be accessed and the data to be written and returns OK. It is only used for debugging purposes. It has no optimizations to

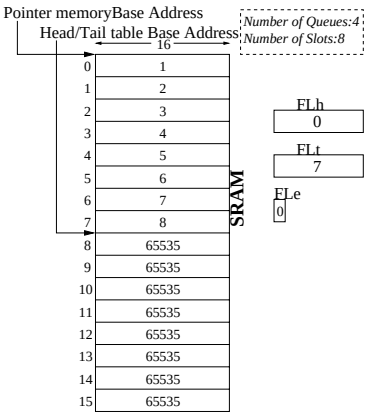

Figure 44: Initialization example for 4 queues and 8 slots

keep the control path simple and avoid aggravating the complexity of the Queue Manager. The timing diagram of this instruction is presented in figure 49 that follows.

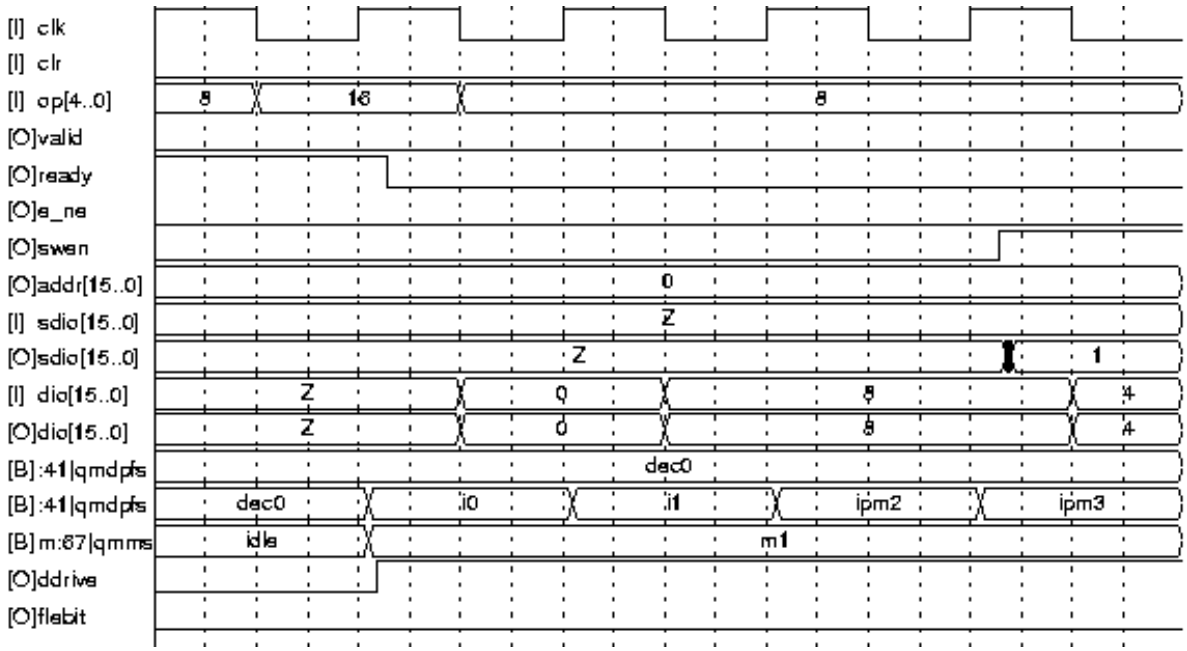

Figure 45: Timing diagram I of Init instruction

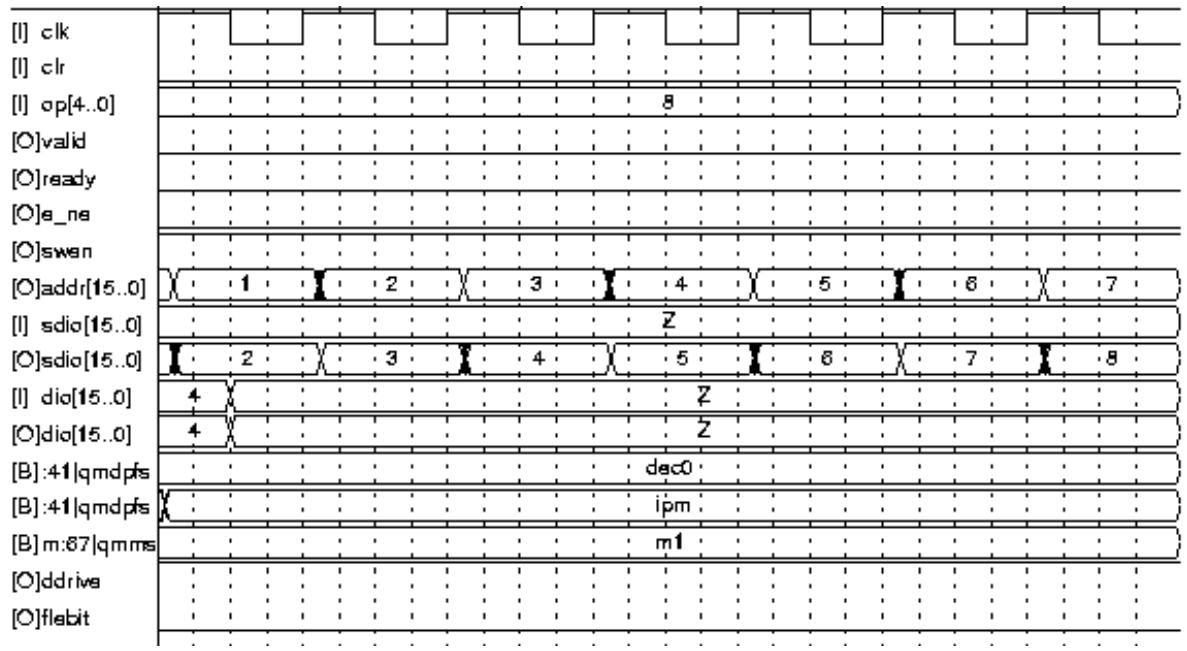

Figure 46: Timing diagram II of Init instruction

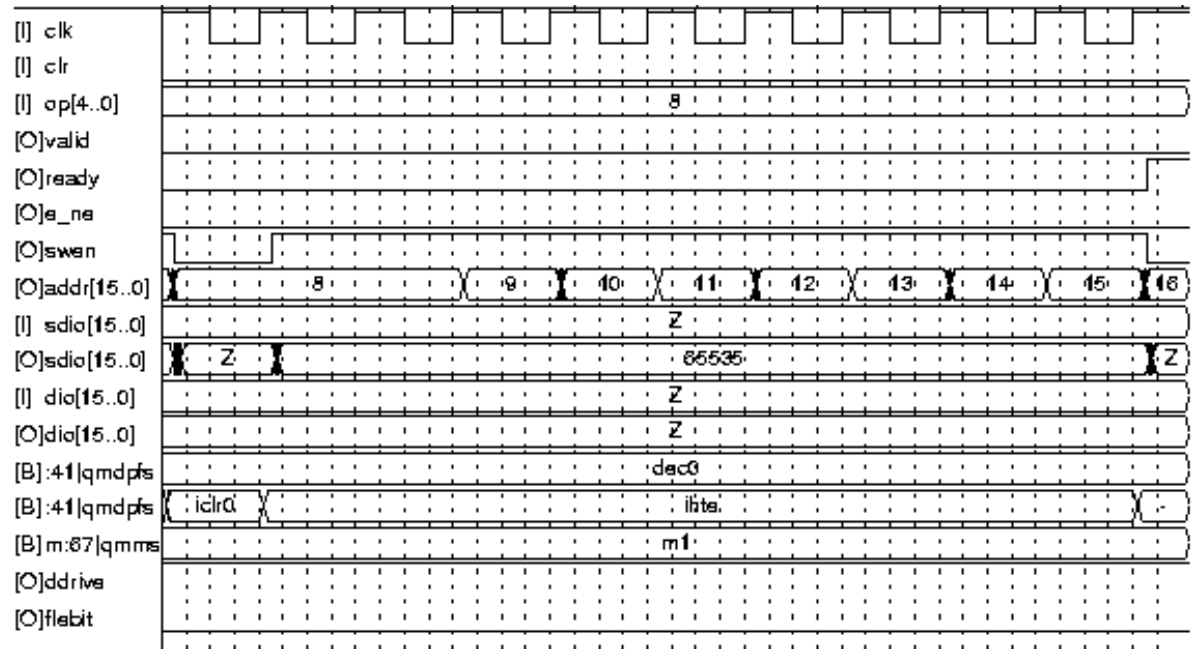

Figure 47: Timing diagram III of Init instruction

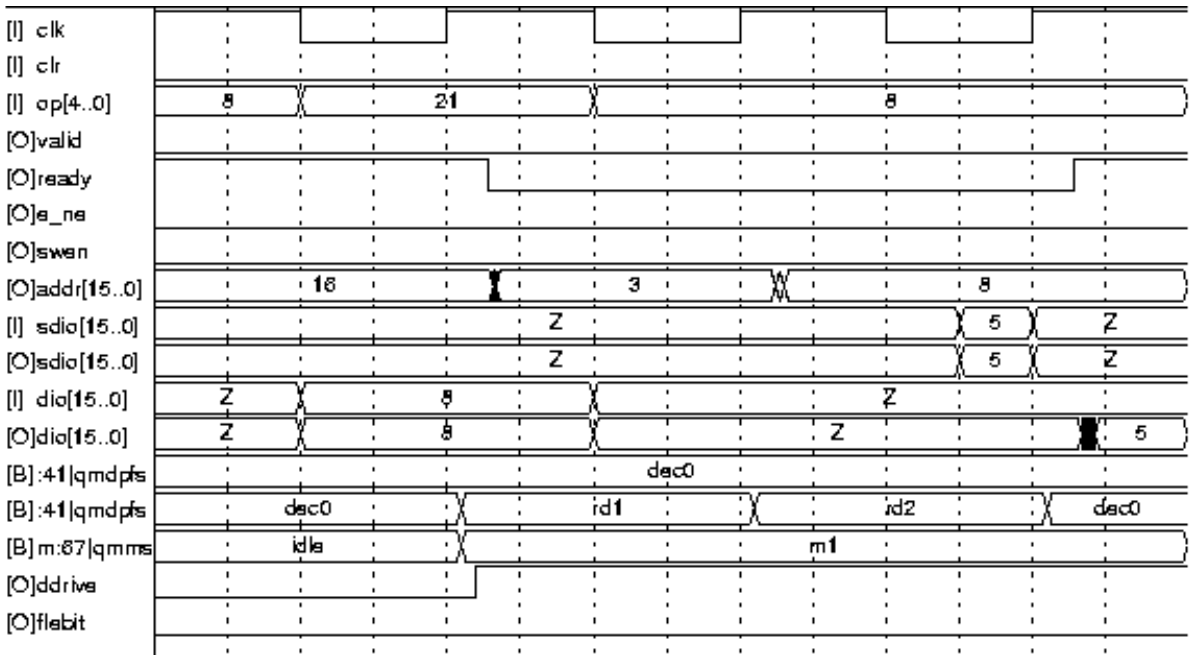

Figure 48: Timing diagram of Read instruction

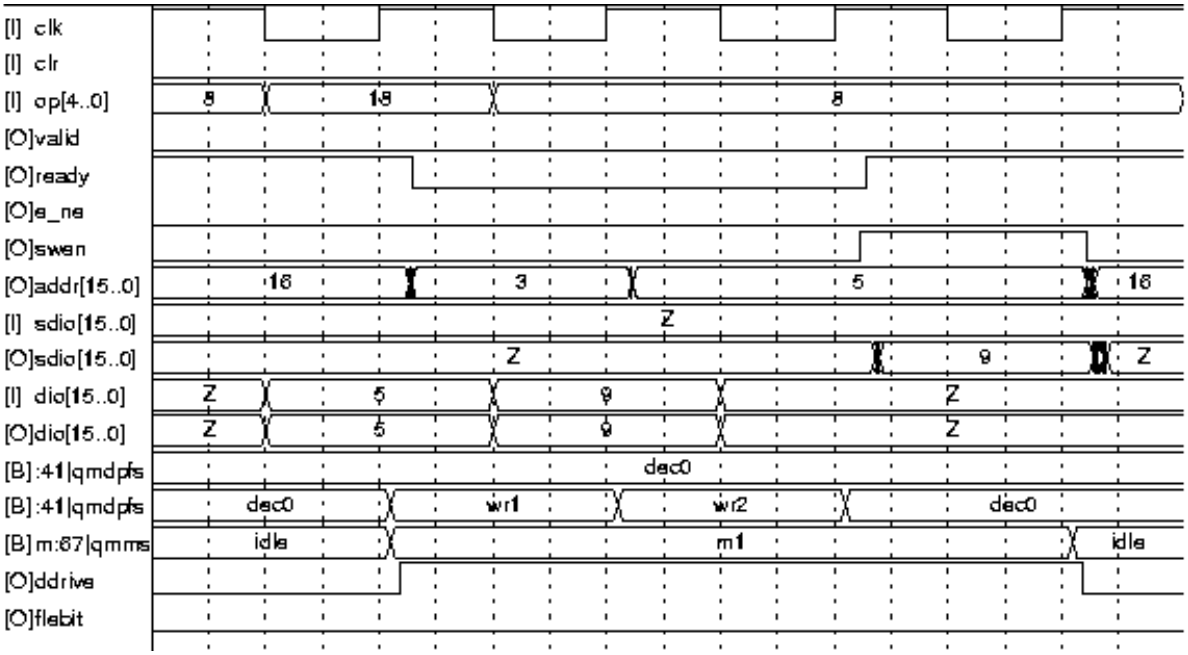

Figure 49: Timing diagram of Write instruction

A.8 Top instruction

Takes as argument the queue number and returns the top element of this queue. It uses the same control logic that is used by the dequeue instruction. The timing diagram of this instruction is presented in figure 50 that follows.

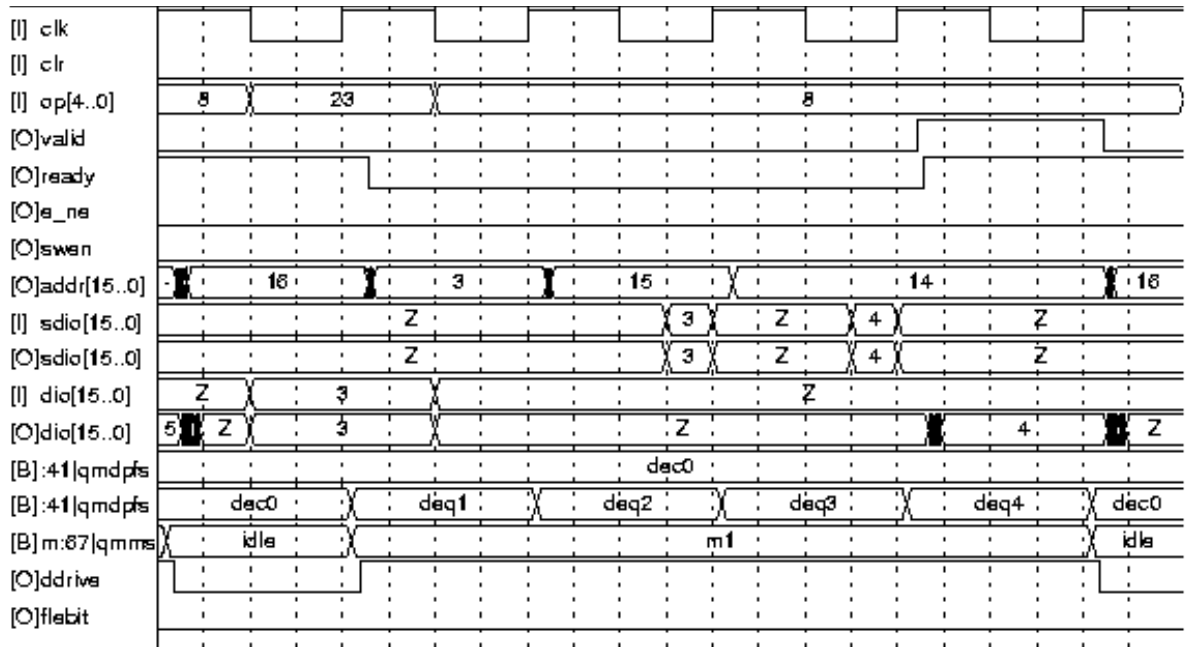

Figure 50: Timing diagram of Top instruction

A.9 CreateQ

CreateQ does not seem to have any use in the MUQPRO I. However, it will be required in future implementations where a VP/VC to queue number translation exists. It takes 1 argument which is the VP/VC, reserves a queue number and returns this "reserved" queue number.

A.10 DeleteQ

DeleteQ does the inverse of what CreateQ does. It releases a queue number and returns OK.

B Datapath of MUQPRO I QM module

The datapath of the QM module is presented in figure 51. Notice that the critical path is the one starting from a register (Src1, Src2 etc), passing from the ADDR_mux and the BAR_add and ending in the Saddr register which provides the address to the external SRAM chip. The path to the SData register which provides the data to the SRAM is more relaxed because it passes from only one medium-sized multiplexor. The CONTROL unit generates all the enable signals for the registers and the select signals of the multiplexors. Thus another not so obvious critical path lays between the CONTROL unit and the datapath. The CONTROL unit is described in appendix A where for each instruction a sequence of operations each taking one cycle to complete is analyzed. The control path is pretty complicated due to the variable size of each instruction. For example a getfree can take either one cycle when the Free List is empty

or 3 cycles in any other case. The path between the generation of the control signals and the datapath is also critical and created problems in achieving the desired clock frequency.

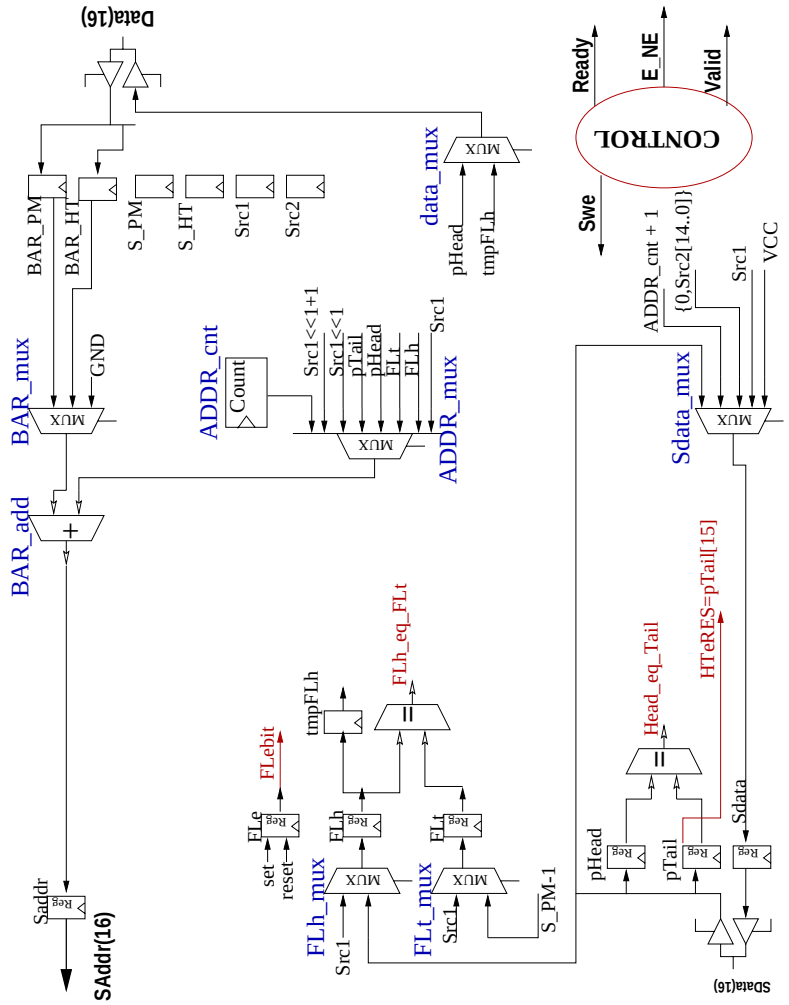

Figure 51: The Datapath that implements these operations

C The "fully scalable" implementation datapath

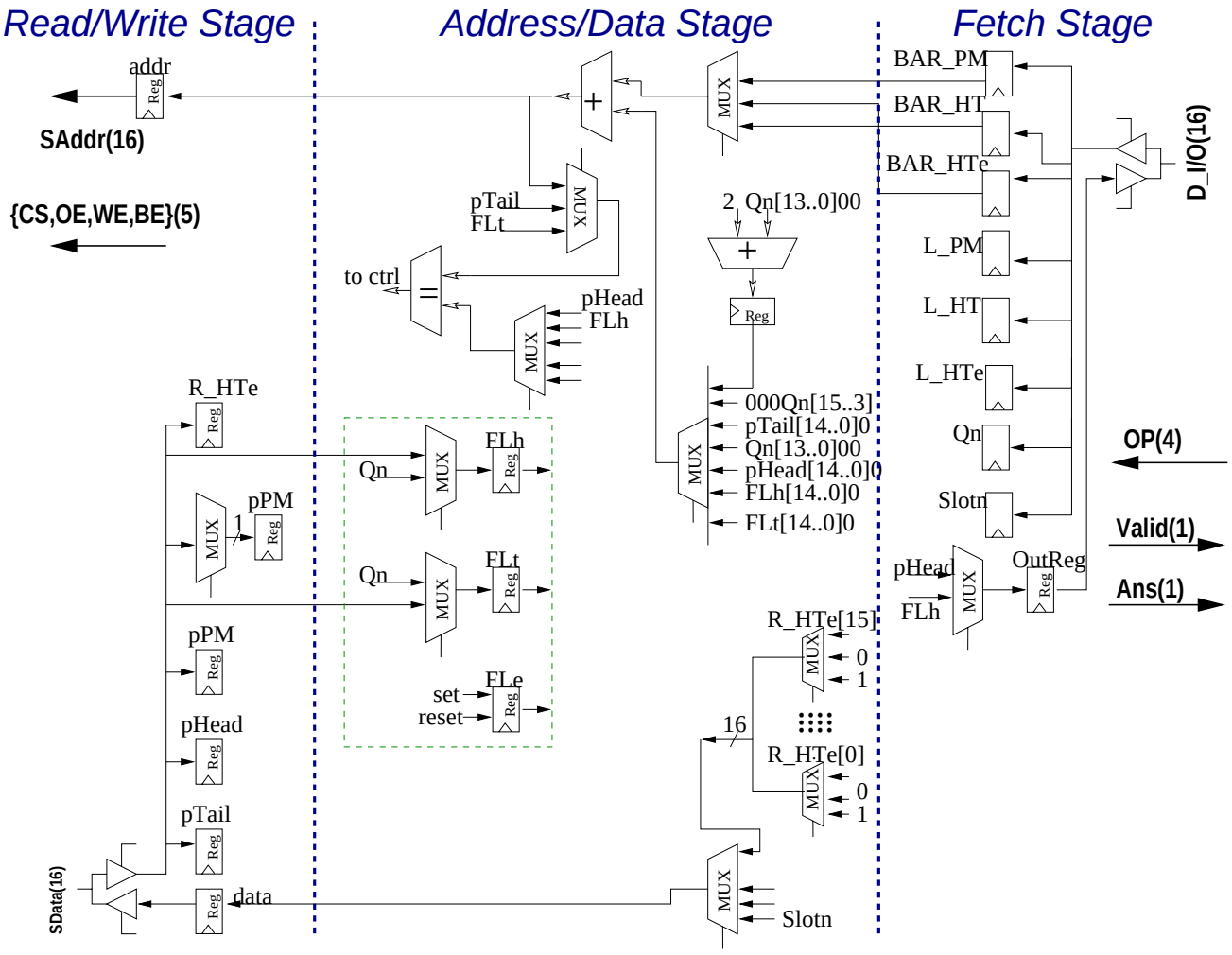

Figure 52: The Datapath of the "fully scalable" implementation

D The "fully scalable" implementation control path

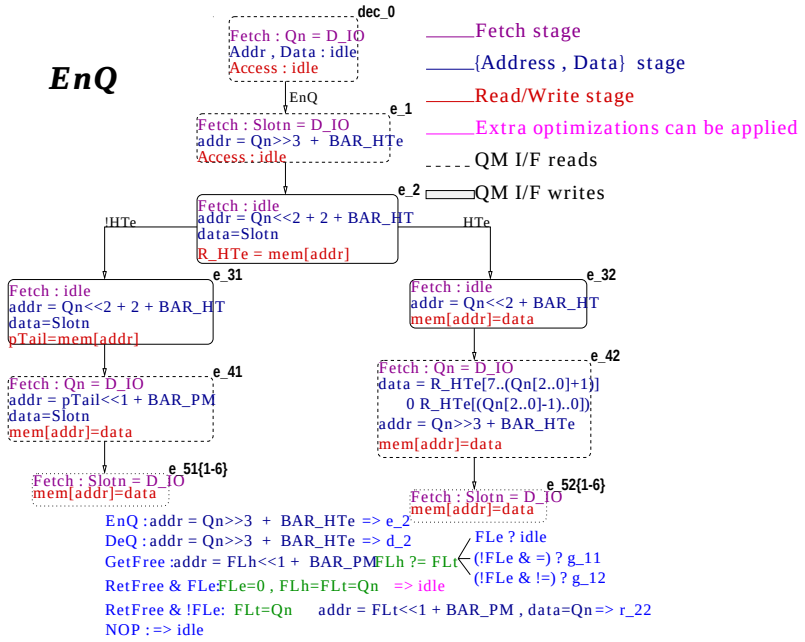

Figure 53: Schedule for the EnQ command

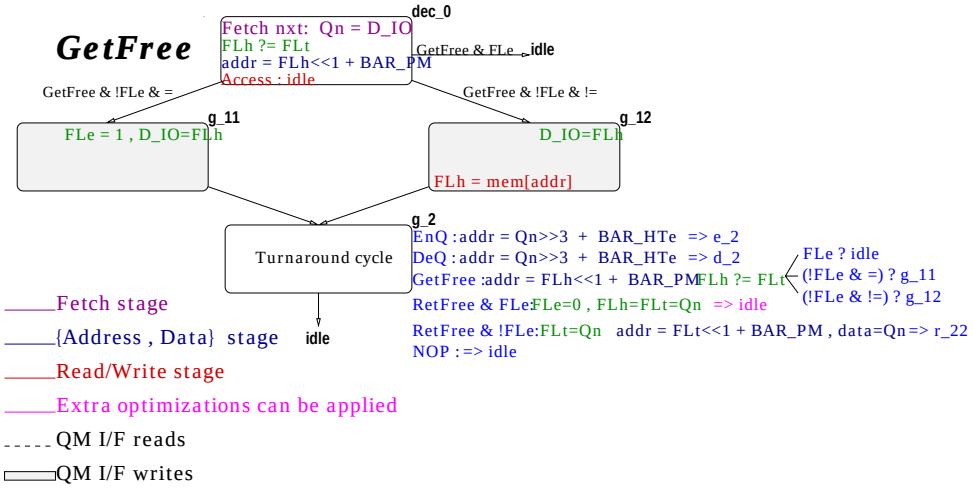

Figure 54: Schedule for the GetFree command

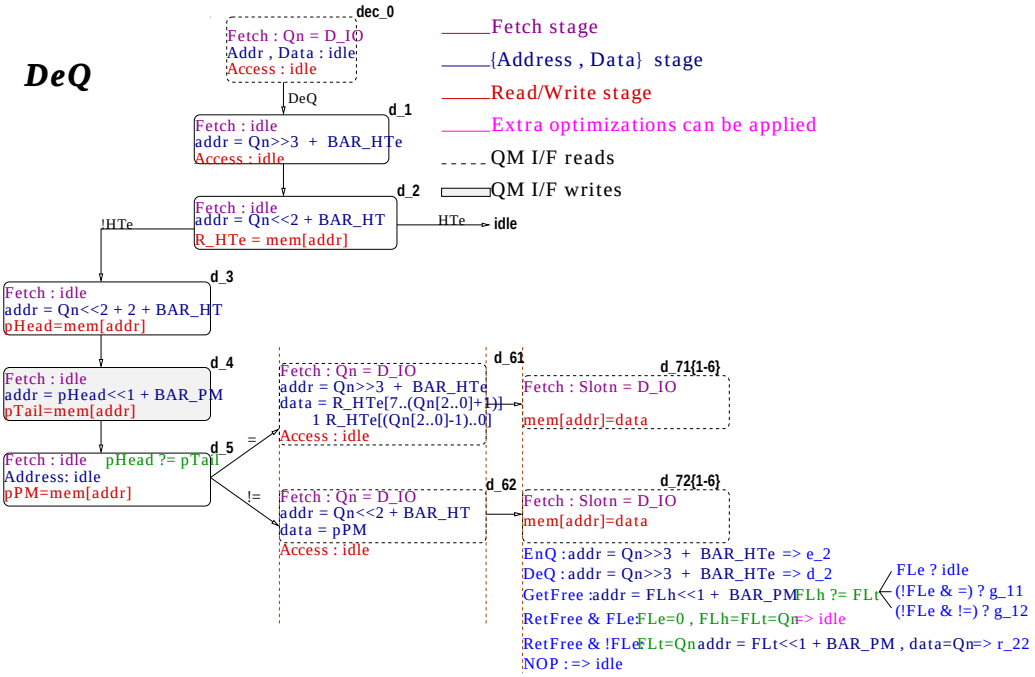

Figure 55: Schedule for the DeQ command

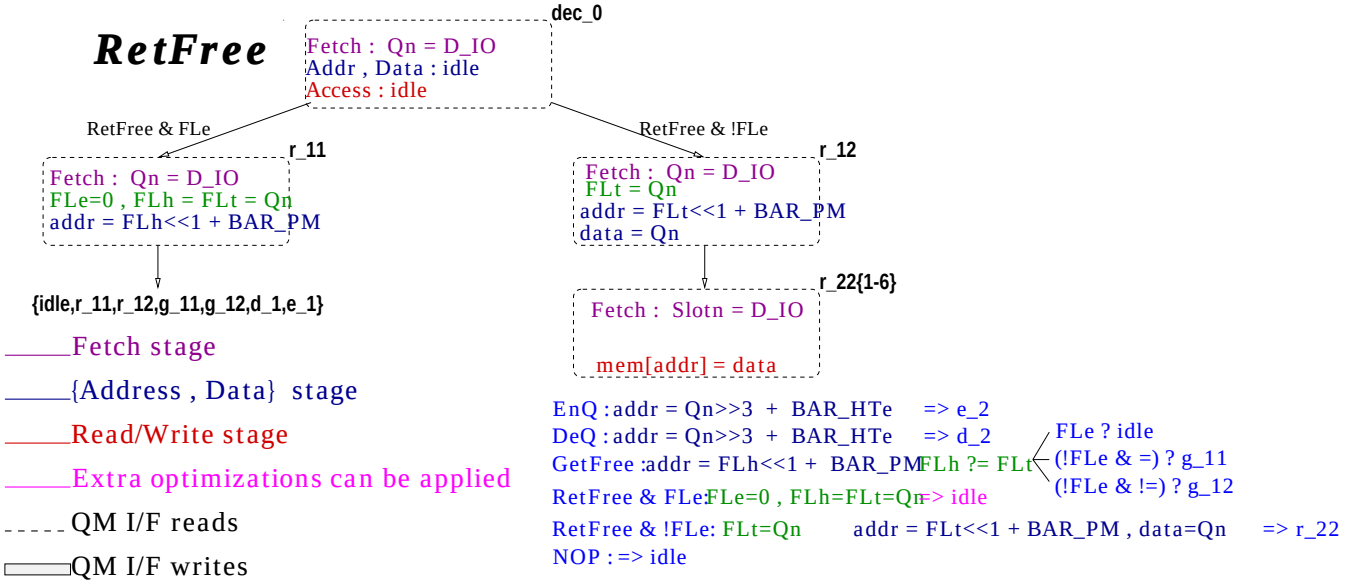

Figure 56: Schedule for the RetFree command

E The "fully parallel" implementation data and control path

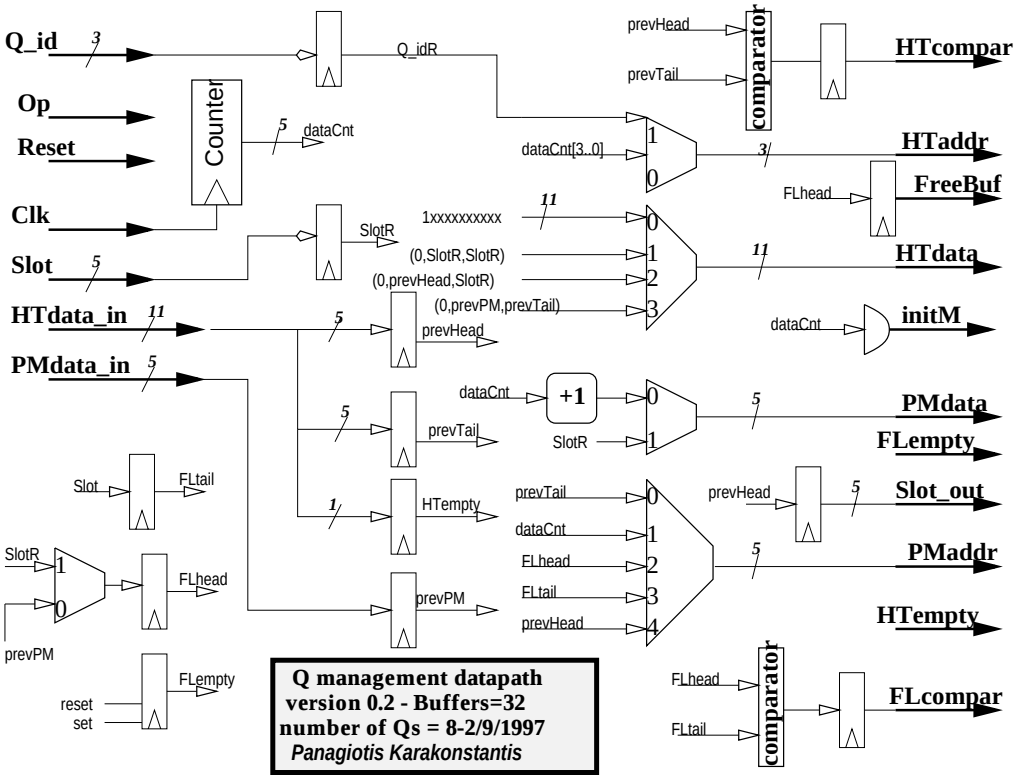

Figure 57: The Datapath of the "fully parallel" implementation

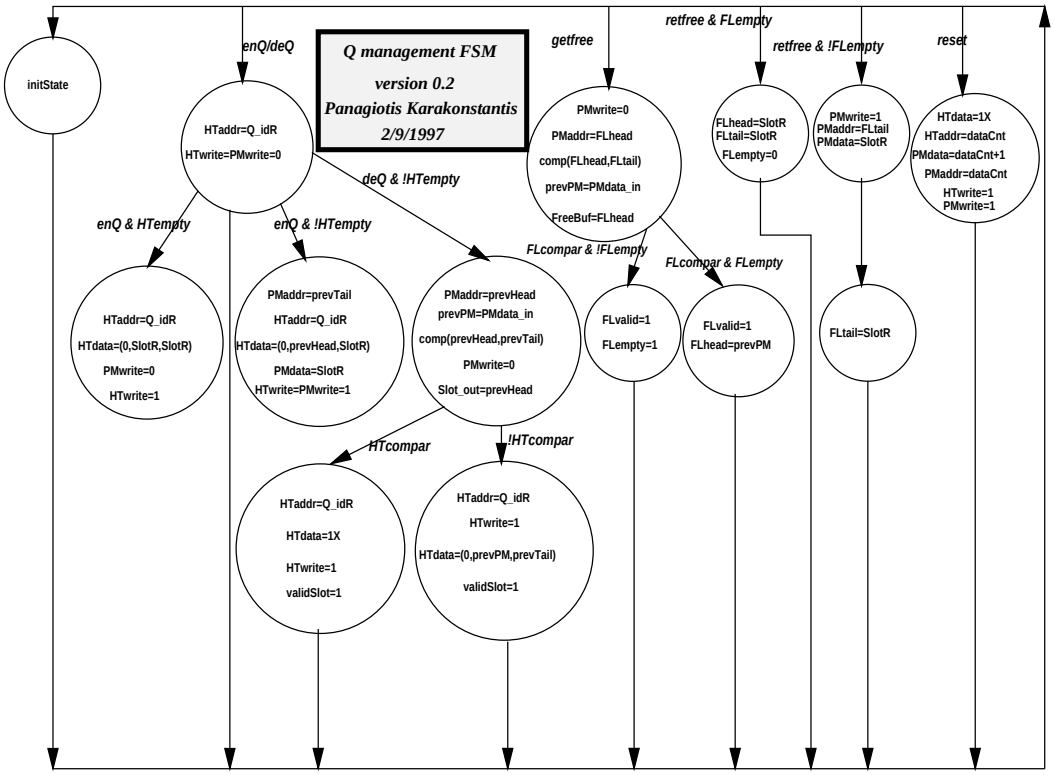

Figure 58: The FSM of the "fully parallel" implementation

References

[1] D. Serpanos, "Communication Subsystems for High-speed networks: ATM requirements", In Asynchronous Transfer Mode, Proceedings of TRICOMM'93, Raleigh, North Carolina, pp. 31-38, April 26-27, 1993.

[2] H.E. Meleis and D.N. Serpanos, "Designing Communication Subsystems for High-Speed Networks".

[3] M. Katevenis, D. Serpanos and E. Markatos, "Multi-Queue Management and Scheduling for improved QoS in Communication Networks", Proceedings of the European Multimedia Microprocessor Systems and Electronic Commerce (EMMSEC'97), Florence, Italy, November 1997.

[4] G. Kornaros, C. Kozyrakis, P. Vatsolaki and M. Katevenis, "Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control", Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI'97), pp. 127-144, Univ. of Michigan, Ann Arbor, USA, Sept. 1997.

[5] Flavio Bonomi and Kerry W. Fendick, " The Rate-Based Flow Control Framework for the Available Bit Rate ATM Service", IEEE Network Magazine, Vol. 9, No. 2, March/April 1995, pp. 25-39.

[6] Raj Jain, "Congestion Control and Traffic Management in ATM Networks: Recent Advances and A Survey", Proceedings of the 4th Int. Symposium on High-Performance Computer Architecture (HPCA-4),

[7] M. Katevenis, D. Serpanos and E. Spyridakis, "Credit-Flow-Controlled ATM for MP Interconnection:the ATLAS I Single-Chip ATM Switch", Proceedings of the 4th Int. Symposium on High-Performance Computer Architecture (HPCA-4), pp. 47-56, Las Vegas, Nevada USA, February 1998.

[8] Kung and R.Morris, "Credit-Based Flow Control for ATM Networks", IEEE Network Magazine, Vol. 9, No. 2, March/April 1995, pp. 40-48.

[9] IBM Corporation, "Algorithm for Managing multiple First-in, First-out Queues from a single shared random-access memory", IBM Technical Disclosure Bulletin: Vol.32, No 3B, August 1989.

[10] SUN Microelectronics, "ATM622-s single-chip ATM SAR", Data Sheet, July 1997.

[11] FORE Systems, "ForeRunner ASX-200BX and ASX-1000", 1998.

[12] ALTERA, "MAX+PLUS II Getting Started".

[13] ALTERA, "AHDL Manual".

[14] John David Carnaugh and Timothy J. Salo, "Internetworking with ATM WANs", Minnesota Supercomputer Center Inc, December 1992.

[15] M. Batubara and A. J.Mc Gregor, "An Introduction to B-ISDN and ATM", MONASH-TR 93/14, September 1993.

[16] C.Kosak, D.Eckhardt, T. Mummert, P.Steenkiste, A. Fisher, "Buffer Management and Flow Control in the Credit Net ATM Host Interface", School of Computer Science - Carnegie Mellon University.

[17] C. Brendan and S. Traw, "Hardware/Software Organization of a High-Performance ATM Host Interface", IEEE Journal on Selected Areas in Communications, Vol 11, No 2, February 1993.

[18] K.K. Ramakrishnan , "Performance Considerations in Designing Network Interfaces", IEEE Journal on Selected Areas in Communications, Vol 11, No 2, February 1993.

[19] Tim Moors and Antonio Cantoni, "ATM Receiver Implementation Issues", IEEE Journal on Selected Areas in Communications, Vol 11, No 2, February 1993.

[20] Casoni, M., and Turner, J. S., "On the Performance of Early Packet Discard", IEEE Journal on Selected Areas in Communications, Vol 15, No 5, June 1997.

[21] ATM Forum, "ATM User-Network Interface Specification V3.1", 1994