Chapter Seven

The DAMQ Buffer Chip

The previous three chapters of this dissertation evaluated the DAMQ buffer architecture in the context of a total communication network. This evaluation involved simulation studies that compared the performance of networks whose configurations are identical except for the type of buffers in the switches. The validity of these simulations relies upon the accuracy of two assumptions. The first is that the DAMQ buffer is capable of transmitting / receiving data at as high a rate as a FIFO buffer. The second assumption is that it is capable of forwarding packets with as low a latency (from when they were received) as a FIFO buffer. This chapter examines an implementation of the DAMQ buffer with two goals in mind. The first is to demonstrate the feasibility of the DAMQ buffer architecture. The second goal is to show that the DAMQ buffer's complexity does not reduce its raw performance relative to the much simpler FIFO buffer architecture.

While the DAMQ buffer was designed to be a functional unit within a single-chip communication switch, this chapter describes a ''stand-alone'' buffer, the *DAMQ Buffer Chip*. Constructing a complete communication switch will require combining multiple DAMQ Buffer Chips together with several other key building blocks, such as a router and arbiter.

When considering a single packet buffer (as opposed to a switching network), there are two key measures of performance. The first is the raw bandwidth of the buffer — the amount of data which can be transmitted and / or received per unit time. The second is the minimum cut-through latency of the buffer — the interval between the time a packet begins to arrive at the input port of an empty, idle buffer and the time said packet begins to depart. This chapter demonstrates that the DAMQ Buffer Chip is the equal of a FIFO buffer in both of these measures.

The key to showing that the DAMQ buffer can sustain as high a raw bandwidth as a FIFO buffer is the fact that the layout of the physical memory used to store the packets is nearly identical for the two buffer architectures. If we assume that the FIFO buffer's control logic is infinitely fast, then it will be its buffer memory which determines the rate at which the FIFO buffer communicates. Thus, asserting that the DAMQ Buffer Chip has as high a raw bandwidth as a comparable FIFO buffer is a two-step operation. First, we design the DAMQ Buffer Chip with a buffer memory whose latency is as low as we are capable of making it. Then, we implement the control logic such that it is the buffer memory which determines the DAMQ Buffer Chip's clock rate.

We take a different tack to demonstrate that the DAMQ Buffer Chip achieves as low a cut-through latency as a FIFO buffer. The approach here is to find a chain of serial dependencies that exist with both FIFO buffers and DAMQ buffers and which combine to determine a theoretical minimum cross-switch latency. If the DAMQ Buffer Chip can be implemented so that it achieves this minimum latency, the implication would be that the cut-through latency with a DAMQ buffer can be as short or shorter than the cut-through latency with a FIFO buffer.

The next section describes the microarchitecture of the DAMQ Buffer Chip. The hardware structures which organize the packets into multiple queues are explained. A minimum virtual cut-through latency of four clock cycles is established in this section. Having set this as a performance goal, the hardware mechanisms capable of meeting this goal are presented. Sec. 7.2 discusses the layout and circuit performance of the individual functional units. Results of SPICE circuit simulations are presented. These results demonstrate that the DAMQ Buffer Chip achieves a raw bandwidth equal to that of a FIFO buffer. Finally, Sec. 7.3 summarizes the contributions of this chapter.

7.1. The Architecture and Microarchitecture of the DAMQ Buffer Chip

The external interfaces and the context in which the DAMQ Buffer Chip is intended to operate will be discussed, and then the internal microarchitecture described. The DAMQ Buffer Chip is a packet buffer with a single byte-wide input port and a single byte-wide output port. It is designed to receive packets from and transmit packets to other DAMQ Buffer Chips. A communication switch can be constructed by connecting the output ports of a number of DAMQ Buffer Chips to the inputs of a crossbar — the input ports of the buffers become the input ports of the switch, and the outputs of the crossbar are the switch's output ports.

The packet format used by the DAMQ Buffer Chip is a single *header byte* followed by one to thirty-two bytes of data. The header byte contains the routing information for the packet plus a bit which indicates whether the packet is thirty-two bytes in length or less. If it is less than thirty-two bytes, then the byte following the header indicates the length of the packet. Packets arriving at an input port have their data stored in the memory array of the buffer at that input port while the header is sent to the router. The router determines the output port to which the packet should be forwarded. With some routing schemes, such as virtual circuits [Bert87], the router also returns a new header byte to be prepended to the packet for its next ‘‘hop’’ through the network. In the DAMQ Buffer Chip, the router returns a four-bit value specifying which of the four queues to append the packet to, an eight-bit header to be associated with the packet, and a *valid queue* signal which causes the routing information to be latched and processed.

In order to forward a packet, a connection must be established between the DAMQ Buffer Chip and the appropriate switch output port. The *crossbar arbiter* establishes connections based upon the state of all the input buffers and the switch output ports. Each buffer has four request lines which indicate to the arbiter which of the buffer’s packet queues are not empty. The arbiter responds to each buffer with a four-bit value specifying which of the four queues is to transmit (i.e. to which output port the buffer has been connected) and a *valid output port* signal which causes the arbitration information to be latched and processed by the buffer. Upon receiving a valid output port signal from the arbiter, the DAMQ Buffer Chip commences transmission of the packet at the head of the queue specified by the arbiter. The packet is transmitted at a rate of one byte per clock cycle, with the first byte being the new header which was supplied by the router.

Since the intended application for the DAMQ Buffer Chip is in the communication switch of a tightly-coupled multicomputer, it is assumed that all switches operate at the same frequency. We avoid the difficulties of distributing a global clock signal by allowing each switch to generate its own clock via a local oscillator. *Synchronizers* are placed between the input port of the switch and the DAMQ Buffer Chip’s input port. These determine for each packet whether to latch incoming data on the rising or falling edge of the local clock [Tami88a]. The probability of synchronization failure due to clock drift over the course of a reception is reduced to insignificance by restricting the length of packets allowed in the system. For this implementation, the maximum packet size is thirty-two bytes [Tami88a].

Having described the external architecture and context of the DAMQ Buffer Chip, the rest of this section will describe the design and implementation of its internal features. Sec. 7.1.1 describes the basic organization of the DAMQ Buffer Chip. Sec. 7.1.2 addresses the specific hardware units which implement this organization. Minimizing the cut-through latency of the buffer is the driving force behind the microarchitecture of the DAMQ Buffer Chip — Sec. 7.1.3 establishes a ‘‘minimum’’ virtual cut-through latency of four clock cycles and describes in detail the hardware features which support this. Finally, Sec. 7.1.4 summarizes the DAMQ Buffer Chip microarchitecture.

7.1.1. The Basic Organization of the DAMQ Buffer Chip

The DAMQ buffer is a single storage array in which memory is dynamically allocated among multiple queues of packets, one queue for each output port. Such dynamic partitioning of the buffer between the queues is significantly more efficient than static partitioning, in terms of both implementation complexity (Ch. 3) and buffer utilization (Chs. 4 and 6). While the order in which queues are serviced is ‘‘random’’ (determined by the arbiter), it is desirable to transmit the packets within each queue in FIFO order (some transmission protocols such as virtual circuits require this).

With dynamic buffer allocation, a packet’s location within the buffer cannot be determined by the location of the packet immediately ahead of it in its queue or the location of the packet whose arrival preceded it. The DAMQ Chip thus requires a mechanism to rapidly locate, access, and transmit the first packet in any one of the queues. Further, in order to handle variable-length packets efficiently, the DAMQ buffer must be able to quickly allocate memory for incoming packets without wasting buffer space due to internal or external fragmentation.

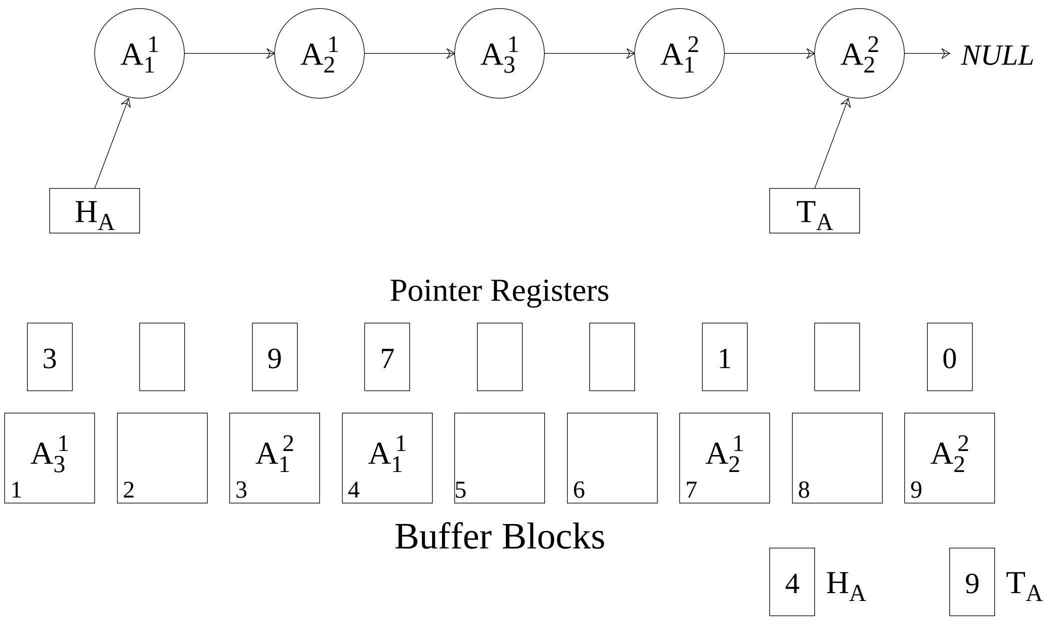

Figure 7.1: *Linked Lists via Buffer Blocks.* Pointer registers link the blocks which comprise a packet (A_1, A_2, etc.) and the packets which comprise a queue (A^1 and A^2 of queue A).

The problem of maintaining the queue organization is resolved via linked lists. The memory array of the DAMQ buffer is partitioned into fixed-size *buffer blocks*. Each queue is a linked list of these blocks (Fig. 7.1). *Head registers* and *tail registers*, which point to the first and last block of each queue, are located outside the array of buffer blocks, allowing access to any one of the queues. To reduce internal fragmentation due to variable size packets, the buffer blocks (eight bytes) are significantly smaller than the maximum packet size (thirty-two bytes in this implementation). Each packet is stored in one or more buffer blocks. For each queue, the linked list of buffer blocks defines the FIFO order between packets as well as the order of blocks that make up each packet. A linked list of ‘‘free buffer blocks’’ (the *free list*) is used to keep track of storage available for incoming packets. For each buffer block there is a *pointer register*, stored outside the packet memory array, which contains the address of the next buffer block of its list or a *NULL* value signifying that it is the last block in the queue.

7.1.2. The DAMQ Buffer Chip Control Logic

The control logic of the DAMQ buffer is significantly more complex than that of a FIFO buffer. The process of receiving a packet involves: (1) locating the block at the head of the free list and directing incoming data to that block, and (2) altering the linked list pointers, logically moving the block to the tail of the appropriate queue. When a packet is transmitted, the block which is at the head of the current queue is moved to the tail of the free list while its data is being read. Whether receiving or transmitting, the primary function of the control logic of the DAMQ buffer is to move a buffer block from the head of one queue to the tail of another.

The control logic of the DAMQ buffer must be able to manipulate the linked list pointers, moving buffer blocks from one linked list to another, in as few clock cycles as possible. The control must also satisfy three additional requirements: (1) minimize the silicon area required for the buffer control hardware, (2) support the simultaneous reception and transmission of packets (both to maximize link utilization and to support virtual cut-through), and (3) ensure that the control logic is not on the DAMQ buffer's critical path. The last requirement is key to allowing the DAMQ buffer to realize its performance advantage [Tami88b, Tami88c] With its simple control logic, other factors (e.g. reading from the memory array or the raw bandwidth of the links) will determine the clock rate of a FIFO buffer.

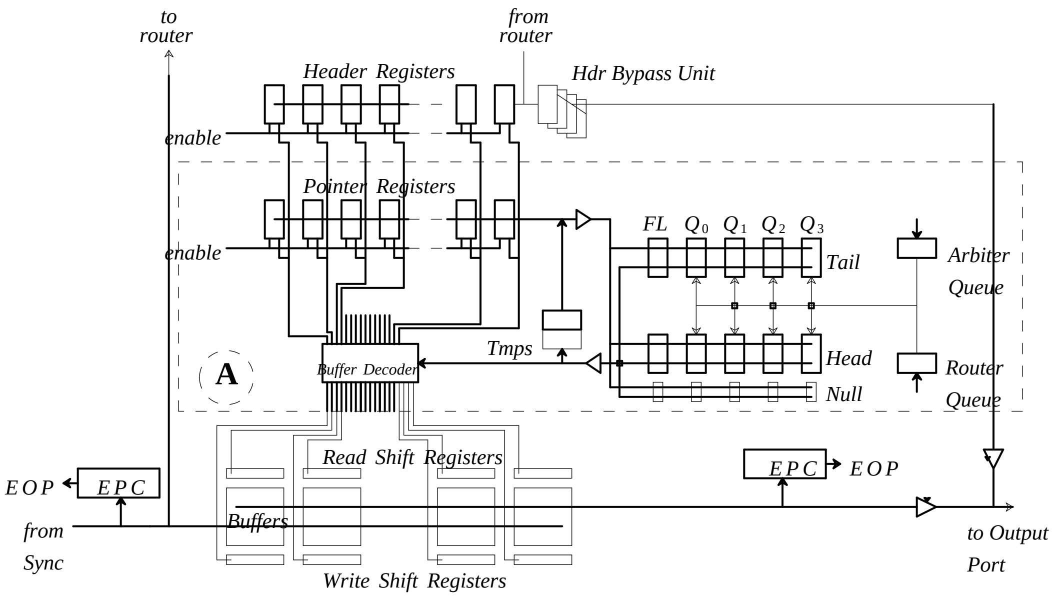

Figure 7.2: *DAMQ Chip Control Logic.* The circuitry in the rectangle marked 'A' implements the five linked lists.

The data path and key control circuitry of the DAMQ buffer are shown in Fig. 7.2. Logically moving a block from one queue to another requires three clock cycles:

cycle 1: The value in the head register of the queue the block is being removed from is copied to one of the two temporary registers. This value is also sent to the decoder, where it is used to select the pointer register containing the address (block number) of the next block in the queue. This pointer register is read and the address is written into the head register, thus removing the block from the queue.

cycle 2: The tail register of the queue to which the block is being appended is read; the value is sent to the decoder to select the pointer register of the block currently at the tail of the queue (if the the queue is empty, the the null register will prevent any pointer register from being addressed). The value in the temporary register, which is the address of the block being ‘‘moved,’’ is copied to the addressed pointer register. The value in the temporary register is also copied to the tail register of the destination queue. The block is now pointed to both by the pointer register of the block ahead of it in the new queue and the tail register of the new queue.

cycle 3: The tail register is read (to address the pointer register of the buffer block), and the *NULL* value is stored in the pointer register, marking this the last block of the queue.

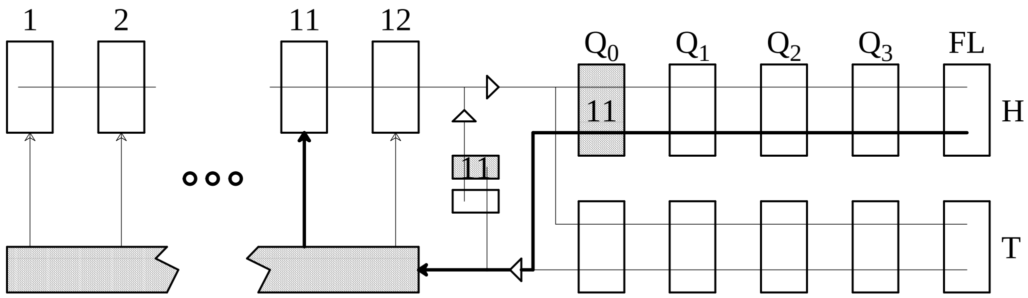

Figure 7.3: *Addressing Phase.*

The operations in each of the three clock cycles are divided into two phases: *addressing* and *data movement*. In the addressing phase, a head or a tail register is read, and its value is driven through the decoder to address a pointer register. This value may also be stored in a temporary register. In the data movement phase, data is either copied from a pointer register to a head or a tail register, or data is copied from a temporary register to both a pointer and a head / tail register. Figs. 7.3 and

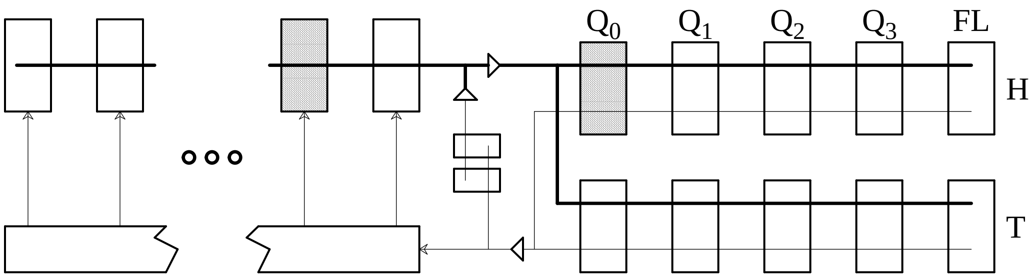

Figure 7.4: *Data Movement Phase.*

7.4 show the actions in each phase of the functional units which implement the linked lists.

Since the buffer must be able to simultaneously receive and transmit packets, the buses and registers are multiplexed. The transmission reception state machines control the data path on alternate clock cycles. Hence, a minimum of five clock cycles are needed to complete the three linked list manipulations involved in receiving or transmitting a packet. However, when a packet destined for an empty queue arrives, there is no need to wait for all of these operations to complete before beginning to forward it (Sec. 7.1.3).

The data path and buffer memory are controlled by three finite state machines (FSMs). FSM_{sto} stores incoming packets into buffer blocks. FSM_{rtr} removes blocks from the free list and appends them to the queue specified by the router. It uses the data path on the same cycles that FSM_{sto} does. FSM_{fwd} forwards the packets. This division of control allows concurrent packet reception, routing and transmission. The queue to which FSM_{rtr} appends packets is specified by the *router queue* register (Fig. 7.2). Similarly, the queue from which FSM_{fwd} transmits packets is specified by the *arbitration queue* register — the finite state machines only distinguish between ‘‘the queue’’ and the free list when specifying head / tail registers for reading or writing.

7.1.3. Minimizing Virtual Cut-Through Latency

Supporting virtual cut-through is key to maintaining low latency communication. With virtual cut-through [Kerm79] the switch can begin to forward a packet before it has completely arrived. A packet can only be cut through a switch if the buffer at which it is arriving is not already transmitting a packet, the output port for which the packet is destined is not connected to another buffer, and there are no packets already in the buffer waiting to be transmitted through that same output port. Optimally, when these conditions exist, the packet will be forwarded as soon as it is routed and the buffer is connected to the appropriate output port. One of the concerns in designed a complex packet buffer such as the DAMQ Buffer Chip is that the buffer will increase the virtual cut-through latency. This section establishes a ‘‘minimum’’ virtual cut-through latency, and then enumerates the hardware requirements for the DAMQ Buffer Chip to achieve this minimum.

Thus, there are two keys to low latency virtual cut-through. First, the routing of the packet and arbitration of the switch must be performed as quickly as possible and in parallel with packet reception. Second, the packet must be available for transmission as soon as the arbitration is complete, no matter how much of the packet has been received at that point. The first condition is satisfied by keeping the routing information at the head of each packet, and by keeping the router and arbiter units independent of the buffer itself. These two measures allow the routing to commence the cycle the packet begins to arrive, and the arbitration to commence as soon as the routing is complete. The second condition is partially met by having separate FSMs for reception and transmission, but some special-purpose hardware is necessary to ensure that the arriving packet is available for transmission when the arbitration is complete.

Cycle	Action	Implication
1	Header is synchronized	
2	Header $\rightarrow$ Router, Router returns valid queue + new header	completion of routing triggers FSM_{rtr}
3	Arbiter receives new connection request, switches the X-bar, returns valid output port. FSM_{rtr} appends the packet to the [empty] queue.	completing of arbitration triggers FSM_{fwd}
4	Packet header traverses the crossbar	FSM_{fwd} must cause header to be read at beginning of this cycle and access buffer memory to send first data byte on the following cycle.
5	First byte of packet traverses the crossbar	To transmit byte at beginning of this cycle, the buffer memory was accessed the previous cycle.

Table 7.1: *Requirements Imposed by Minimizing Cut-Through Latency.* Assuming that routing cannot occur until the packet header is received, arbitration cannot occur until the packet is routed, and transmission cannot begin until the crossbar is arbitrated.

Table 7.1 displays the sequence of events imposed upon the DAMQ buffer by the goal of minimizing virtual cut-through latency and our assumptions with respect to the implementation. We assume that the packet cannot be routed until the packet's header passes through the synchronizer, that the crossbar cannot be arbitrated until the router determines to which output port the packet is destined, that transmission cannot commence until the crossbar arbitration is complete, and, finally, that routing and arbitration will each take at least one clock cycle to perform. Thus, our goal is to support a four-cycle virtual cut-through.

To support a four-cycle virtual cut-through latency, FSM_{rtr} *must* have control of the data path on the cycle following the completion of the routing. Similarly, FSM_{fwd} must control the data path on the cycle following a crossbar arbitration. However, a packet may arrive on any given clock cycle. Thus, the data path must be demand multiplexed between the reception and transmission control logic. Further, the validqueue signal from the router may arrive as late as the end of a clock cycle, but FSM_{rtr} must react to the signal and take control of the data path at the beginning of the next cycle. Sec. 7.2.3 details how the finite state machines support virtual cut-through and demand multiplexing.

As can also be seen from Tab. 7.1, FSM_{fwd} must be able to access both the new packet header and the first byte of data on the clock cycle following the arbitration in order to provide the minimum virtual cut-through latency (the buffer memory read requires a full clock cycle). This presents two problems for the hardware: how to support single-cycle access to a packet header, and how to set the head-of-queue hardware (which previously held nonsense, for an empty queue) to point to the new packet. The remainder of this section details our solution to these problems.

Packet headers are stored in an array of *header registers*. Each header register is associated with a buffer block. Under normal conditions, the DAMQ buffer control addresses the header registers via the same mechanism it addresses the buffer blocks. This addressing mechanism involves reading the block identifier from the head register of the selected queue during the first phase of the clock cycle. The block identifier is sent to the decoder, which asserts the select lines for the header register array, pointer register array and the buffer blocks themselves. However, if this indirect addressing mechanism is used to access the header byte of the packet to be transmitted, the header will not traverse the crossbar until at least two clock cycles after arbitration. What is needed is a mechanism to provide direct access to the header of the packet to be transmitted.

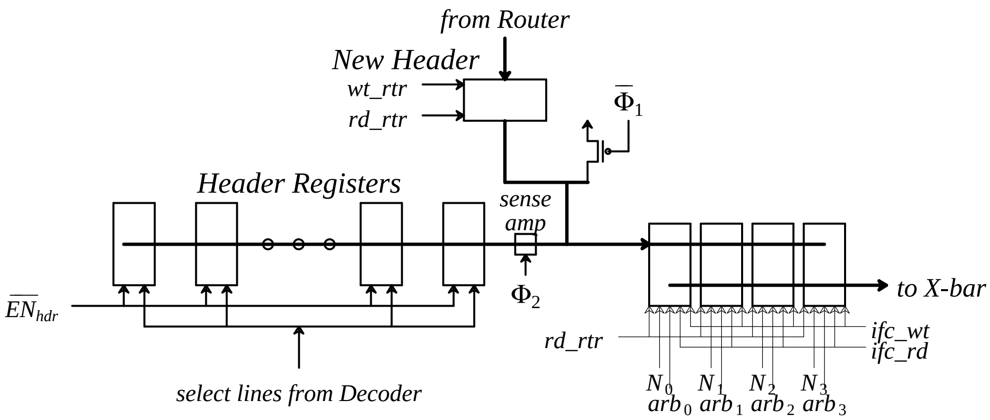

Figure 7.5: *The Header Register Array.* Header bypass registers can be explicitly written ($ifc\_wt \cdot arb_x$) or implicitly written ($rd\_rtr \cdot N_x$) (arb is the four bits from the arbiter which select the queue to be transmitted, N is the values of the four null registers associated with the packet queues).

The *header bypass registers* in the header register array (Fig. 7.5) provide direct access to the header byte of the packet at the front of each of the packet queues. There are four bypass registers — each holds the header of the packet at the head of a queue. When the arbiter returns a *valid output port* signal, FSM_{fwd} transmits the header which is in the bypass register associated with that queue / output port. FSM_{fwd} need only assert *ifc_rd* which, in combination with arb_x (the signals which indicate which queue was selected by the arbiter), cause the header byte to be driven over the crossbar. At the end of each packet transmission, FSM_{fwd} accesses the header register array and loads the bypass register with the header of the new front-of-queue packet.

Virtual cut-through occurs when a queue is empty, however. In the case described in Tab. 7.1, FSM_{fwd} will not have had time to copy the packet's header from the header register to the bypass register before that header is to be transmitted over the crossbar. Since it is FSM_{rtr} which controls moving the packet header from the *New Header* register (Fig. 7.5) to the packet's header register, FSM_{rtr} writes the packet's header into the bypass register when the packet is appended to an empty queue. This is accomplished by writing the header values to the bypass register of every empty queue whenever the *New Header register* is read. Thus, if the router returns a new header at the end of cycle 2 (referring to Tab. 7.1), FSM_{rtr} controls the data path on cycle 3 and reads the New Header register on that cycle, causing the header to be written into the bypass registers of all empty queues.

A similar situation exists with respect to the *head register* of the queue to / from which the cut-through packet is being written / read. As was previously mentioned, the internal organization of the DAMQ buffer is based upon linked lists. Each queue is referenced by two registers; the *head register* points to the first block in the queue, and the *tail register* points to the last. FSM_{rtr} appends a block to a queue by manipulating the tail register of the queue (T_Q). FSM_{fwd} locates the first block of a queue by accessing the head register (H_Q). When the queue is empty, the head and tail registers hold nonsense values.

In clock cycle 4 of Tab. 7.1, FSM_{fwd} reads the head register of the queue, and uses this value to access the buffer block. This implies that, on cycle 3, FSM_{rtr}

must write the identifier of the block which is at the head of the free list to the tail register of the appropriate queue *and* that this value also be written to the the head register. The first issue is making the write to T_Q be the first function performed by $\mathrm{FSM}_{\mathrm{rtr}}$. Reading H_{fl} (the free list's head register) can be performed ahead of time because (a) it is known that any incoming packet will be stored in that buffer block and (b) the lines connecting the decoder to the addressing shift registers of the buffer blocks have latches that allow a buffer block to be selected an indeterminate number of clock cycles ahead of when reception will begin. So, the value in the head register of the free list can be stored in the temporary register ahead of time, and $\mathrm{FSM}_{\mathrm{rtr}}$'s action during cycle 3 (Tab. 7.1) can be $[read(T_Q);\, ptr(T_Q) \leftarrow tmp\,;\, T_Q \leftarrow tmp\,]$ — read and decode the tail register of the queue specified by the router; store the value from the temporary register (which is the identifier of the block at the head of the free list) into the pointer register associated with the last block of the queue; store the value from the temporary register into the tail register of the queue.

In the case of an empty queue (potential for virtual cut-through), the *null register* of the queue (N_Q) will prevent any pointer register from being selected when T_Q is read. It will also cause any value written to T_Q to also be written to H_Q. Thus, the state machine does not differentiate between appending a packet to empty vs. non-empty queues; the null register automatically sets the head register of an empty queue to point to the first block appended to the queue.

7.1.4. Summary — DAMQ Buffer Architecture

The DAMQ buffer design is driven by the goal of minimizing the latency of virtual cut-through. A register latches the packet's header as soon as it passes through the synchronizer, and presents it to the router along with an enable signal, allowing the routing to take place immediately. The same separation control and addressing hardware which allows the DAMQ buffer to simultaneously receive and transmit separate packets also allows it to cut a single packet through. Demand multiplexing allows the finite state machines controlling packet reception and transmission to access the data path on the clock cycles immediately following routing or arbitration. A copy of the header for the packet at the head of each queue is stored in a *bypass register*, allowing FSM_{fwd} to begin a transmission without the latency of decoding a block identifier. Finally, *null registers* associated with each packet queue cause dual tail / head register writes to empty queues. With an on-chip router utilizing simple algorithmic routing or small a routing table, and an on-chip arbiter which operates in a single clock cycle, the minimum latency across a single DAMQ switch is four clock cycles. Each additional cycle required to route a packet or arbitrate the switch adds a cycle to this latency.

7.2. Floorplan and Circuit Performance

In Ch. 4, it was shown that, given equal clock rates, a network made up of DAMQ buffers outperforms a corresponding FIFO network. The microarchitectures of the DAMQ and FIFO buffers must be compared, however, to determine whether and to what degree the additional complexity of the DAMQ buffer affects its performance. This complexity could cause the DAMQ buffer to operate at a slower clock rate than the FIFO buffer. In addition, the control logic could occupy silicon area which the FIFO buffer would use for additional storage. Additional storage for incoming data increases the FIFO buffer's performance.

In order to compare the performance of the DAMQ buffer to that of the FIFO, the DAMQ Buffer Chip has been designed and layed out using MOSIS scalable ($2\,\mu$) CMOS design rules. While the structure of the DAMQ buffer is significantly different from that of a FIFO buffer, the two architectures share a common feature — the buffer memory array. While the DAMQ buffer imposes a logical structure upon the memory array (the buffer blocks), physically, it is identical to the memory array that a FIFO buffer would use. Since the FIFO buffer's control logic is so simple, there is a strong probability that the latency of buffer reads would determine the cycle time of a FIFO buffer (i.e. the memory array accesses would be the critical path).

Thus, in order to assert that the DAMQ buffer can operate at as high a bandwidth as a FIFO buffer, we implemented a high-performance static memory array as the DAMQ Buffer Chip's buffer memory and established that the DAMQ buffer control logic is fast enough to operate within the clock cycle determined by the buffer memory's latency. The next section describes the implementation and SPICE simulation of the buffer memory. Sec. 7.2.2 describes the operations that the control logic must complete in a single clock cycle and demonstrates via SPICE that the buffer memory remains the performance bottleneck. Sec. 7.2.3 presents the circuitry of the finite state machines and a discussion of their operation within the two-phase clock defined by the data path implementation. We conclude the performance analysis by examining the silicon requirements of the DAMQ and FIFO buffer control logic and summarizing the performance and implementation issues addressed by these sections.

7.2.1. The Memory Array

The integrity of this performance analysis depends upon the implementation of the memory array. If the memory array is slow, then the comparison of the speed of the DAMQ buffer vs. the FIFO will not be valid. If the memory array consumes more silicon area than is required, then our evaluation of the silicon requirement of the DAMQ buffer control logic will be optimistic. These goals were pursued without sacrificing the fundamental goal of a multicomputer switch implementation — minimizing communication latency.

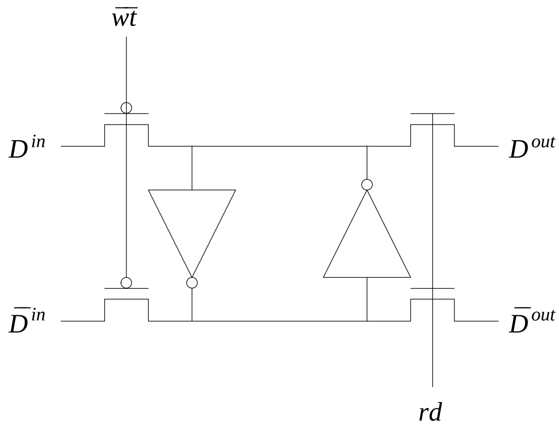

Figure 7.6: *Memory Cell.* Two-ported static memory cell used to construct the buffer memory. Read and write control are independent. Both the read and the write buses are dual-rail.

The memory array is eight bits wide — the same width as the communication links. The array has the capability to read and write a byte of data each clock cycle. It is composed of standard eight-transistor dual-ported static memory cells

(Fig. 7.6). The memory is designed to operate with a two-phase clock. During the first phase, the read bus is precharged and the value to be written is asserted onto the write bus. On the rising edge of the second-phase clock (ϕ_2), read and write select lines (rd and $\overline{wt}$) are asserted on the words of memory to be read and written. A sense amplifier on the read bus accelerates the buffer reads.

In order to increase the speed with which the memory array is accessed, we divided it into two separate arrays (Fig. 7.18), halving the capacitance on the buses. We also took advantage of the sequential nature of the accesses to the memory array by replacing random access address decoding with shift-register addressing. The memory array is divided into eight-byte buffer blocks. Each block has, running along its sides, two eight-bit shift registers: one for reading the block, one for writing. To write to a block of memory, an ''on-bit'' is injected into the write shift register by addressing that block ($\overline{SEL}_{wt}$) while signaling that a write is to begin (EN_{wt}). The bit is shifted during ϕ_1, and the write enable line ($\overline{wt}$) associated with the on-bit is driven low during ϕ_2. When the on-bit reaches the end of the shift register, a signal is generated (EOB_{wt} — end-of-block) indicating that the write to this block is finished.

Figure 7.7 presents the results of a SPICE simulation† of a read of the memory array. At 0 ns, ϕ_2 begins to fall and the precharge of D^{out} and $\bar{D}^{out}$ begins. At ≈ 10 ns, D^{out} and $\bar{D}^{out}$ are equal and ≈ 5.0 v. ϕ_2 rises at 11.0 ns, which causes rd (the read enable signal) to rise. Since rd is in polysilicon (as is $\overline{wt}$), we measured rd^7, which is the voltage of rd as seen by the memory cell on the

† All of the SPICE simulation results presented in this chapter were generated using slow / slow technology files for 2 mu implementations supplied by MOSIS.

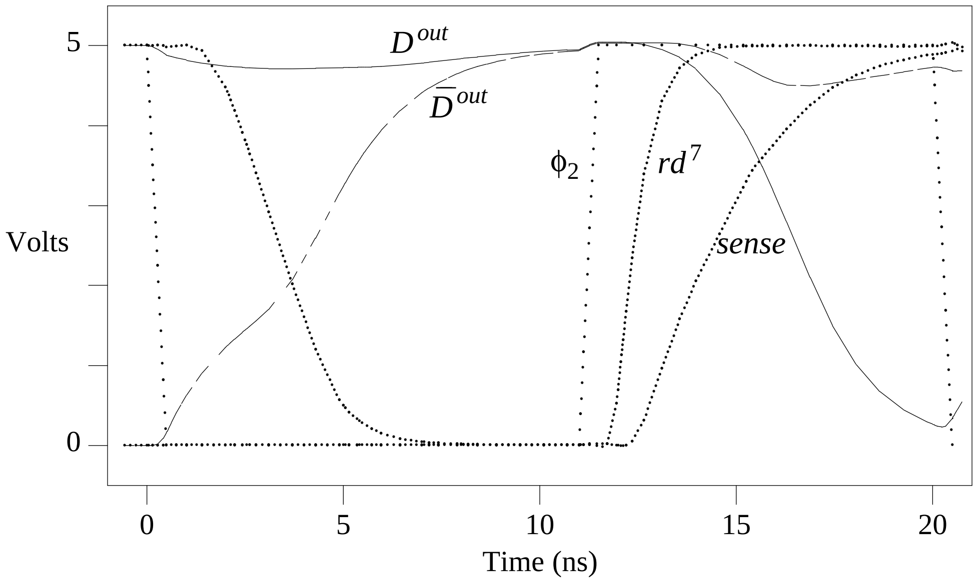

Figure 7.7: *Memory Array Read Latency.* The bus lines (D^{out} and $\bar{D}^{out}$) must be precharged before the rising edge of ϕ_2. The value on the bus is latched on the falling edge of ϕ_2.

opposite side of the memory array from the shift register. The enable signal for the sense amplifier (*sense*) rises soon after rd^7. D^{out} drops below 1.0v at 18 ns — at 20 ns, the value on D^{out} is latched and the precharge for the next read begins. This implementation of the memory array thus defines a 50 MHz two-phase clock in which ϕ_2 has a 45% duty cycle. For comparison, IBM's Vulcan Switch [Stun94], the communication switch of the SP1 multicomputer, is designed to operate at 50 MHz, and also communicates via byte-wide links [Stun94]. Thus, despite using a 2 μ technology, we have successfully implemented a buffer memory which will support the communication throughput demanded by current multicomputers.

7.2.2. The Buffer Control Data Path

The previous subsection established a target clock cycle for the DAMQ Buffer Chip (50 MHz). The goal for the data path implementation is to operate within this clock — i.e. to not be on the critical path. In this subsection, the performance of the functional units which comprise the data path is measured.

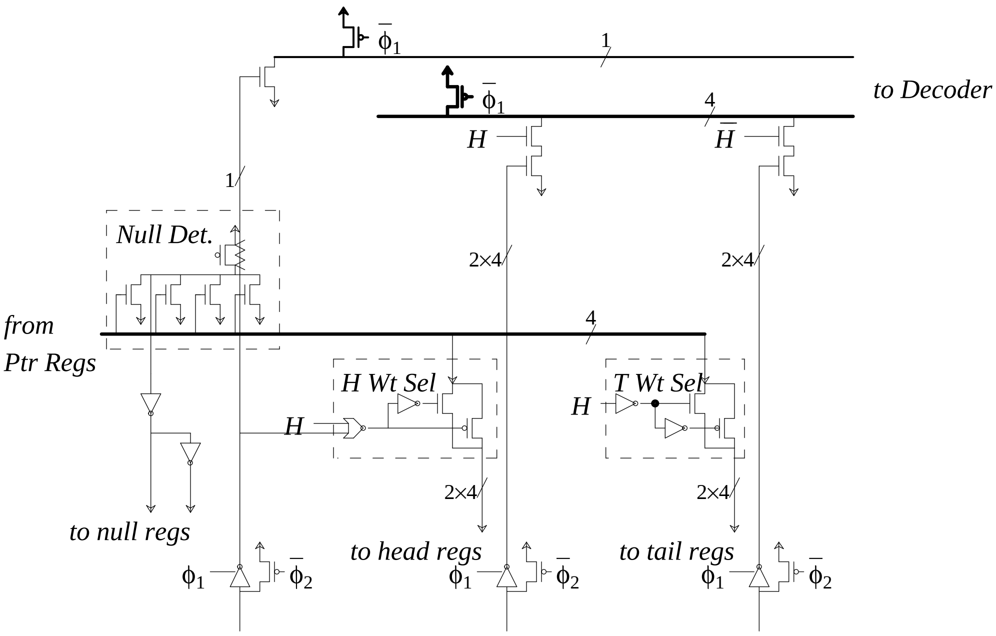

Figure 7.8: *Buses and Associated Logic for the Head/Tail Register Array.*

The actions which occur during a clock cycle begin with the falling edge of ϕ_2. At that point, the finite state machines which control the circuit assert new values on the control lines (Sec. 7.2.3). On the rising edge of ϕ_1, the *head / tail bus* (a dual-rail four-bit bus) is precharged (Figs. 7.8 and 7.10). The buses which are directly driven by the head, tail and null registers are also precharged (Fig. 7.9). It is important that the H and $\overline{H}$ signals be settled before the falling edge of ϕ_1 to prevent charge sharing between the head / tail bus and the transistors gated by H

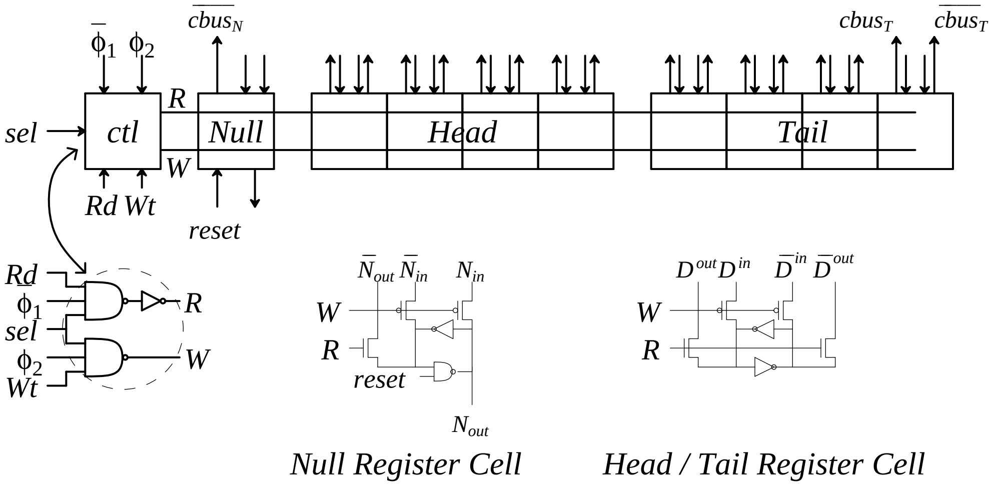

Figure 7.9: *Head / Tail Registers.* The registers which comprise the head and tail register array. A reset line initializes the null registers to '1' (empty queues).

and $\overline{H}$. On the falling edge of ϕ_1, the read signal may be asserted for a head / tail register pair, which begins the *addressing* operations (Sec. 7.1.2). Both the head and the tail registers of the selected queue are read — the $H / \overline{H}$ signal controls pass gates which choose which of their values discharges the bus. The decoder (Fig. 7.10) decodes the block identifier during the underlap between ϕ_1 and ϕ_2, and must complete the decoding of the value (the select lines must have settled to their correct value) by the rising edge of ϕ_2.

The critical path for the addressing occurs when the tail of an empty queue is read. In this situation, there is a '1' stored the null register of the queue, and on the read, the $\overline{N}$ line of the bus will be discharged. In the decoder unit (Fig. 7.10), the decode logic for each of the select signals takes as input $\overline{N}$ and four of the eight lines which comprise the *val* bus. Thus, while no single bit of *val* is connected to more than eight of the twelve select signal decoders (one decoder for each of twelve buffer blocks), $\overline{N}$ gates all twelve. When the tail register of an empty queue

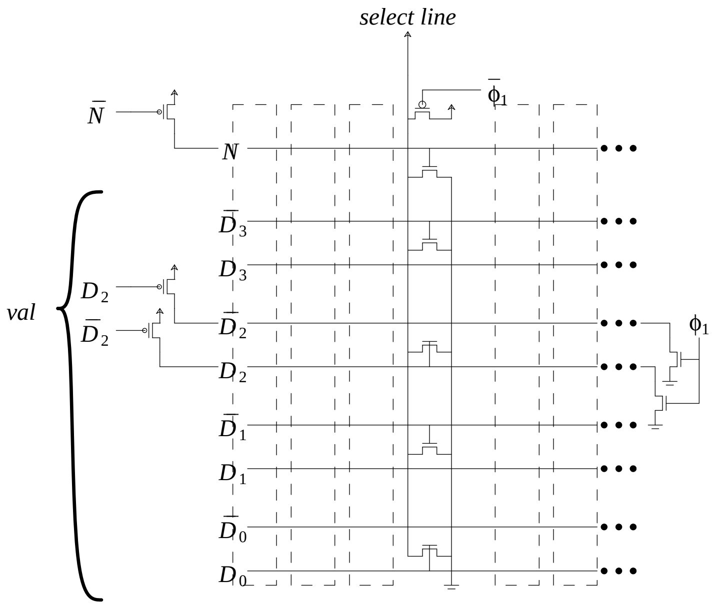

Figure 7.10: *Block decoder.* Decodes four-bit block addresses and asserts a single select line for the addressed buffer block. Address lines are precharged low during ϕ_1, selectively pulled high by *val* on the falling edge of ϕ_1. The select lines are precharged high; the select lines of the unselected blocks must be fully discharged by the rising edge of ϕ_2. The decode cell detailed in the figure is for buffer block 10.

is read, all twelve select lines must be discharged in time to ensure that no pointer register is selected when ϕ_2 rises.

Figure 7.11 shows the results of simulating the reading of a tail register of an empty queue. On the falling edge of ϕ_1, the read enable line is asserted. The selected null register pulls the columnbus ($\overline{cbus}$) down. The inverted value ($cbus$) in turn pulls the $\overline{N}$ line of the bus down. This line, along with the rest of the head / tail register bus, is input for the decoder unit. When the null address line of the decoder is pulled up ($addr_N$), all of the block / pointer register / header register select lines are discharged — no block is selected.

The latency for this operation is just less than 4 ns; for engineering integrity,

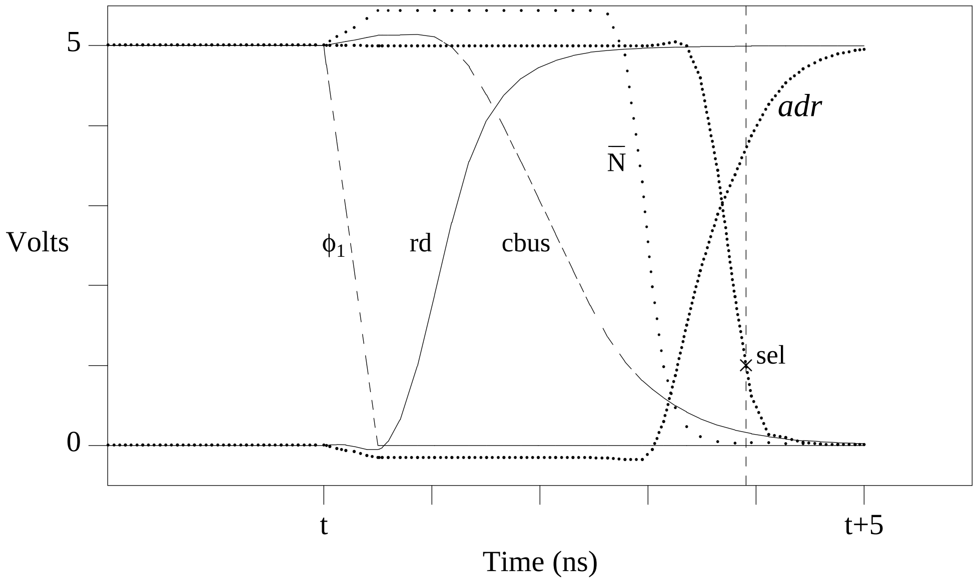

Figure 7.11: *Null Register Read Latency.* The latency from the falling edge of ϕ_1 to when the select lines generated by the decoder unit (Fig. 7.10) drop below 1.0 v.

there must be 5 ns (or more) between the falling edge of ϕ_1 and the rising edge of ϕ_2. If we allow 2 ns for the underlap between ϕ_2 and ϕ_1, and given the 20 ns clock cycle described in Sec. 7.2.1, then the time ϕ_1 is high plus the ϕ_1-ϕ_2 underlap must be 9 ns (or less). If 5 ns are allocated to the underlap between ϕ_1 and ϕ_2, then ϕ_1 is high for 4 ns — which is enough time to settle control signals and precharge the head / tail register buses.

Data movement occurs on the rising edge of ϕ_2. The data movement operations concern the moving of values between the *temporary registers*, the *pointer registers* and the *head / tail registers* (Fig. 7.12). One of three possible operations occurs in the data movement phase. (1) A selected pointer register is read, and its value is written into a head register. The twelve pointer registers (one

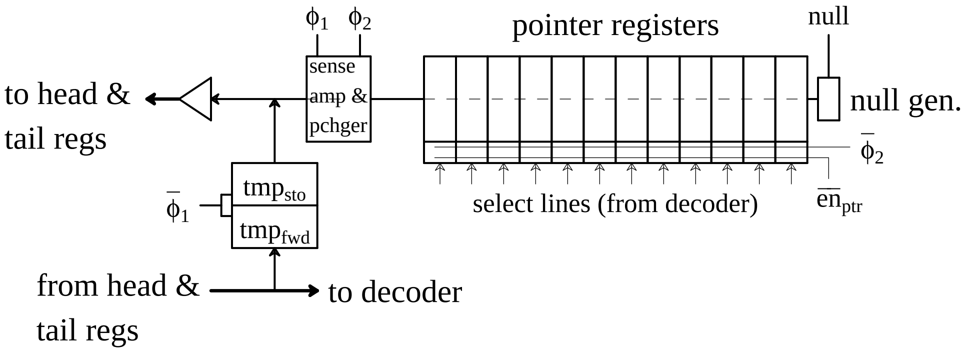

Figure 7.12: *Pointer Register Array.* The pointer registers are accessed via a dual rail bus. The bus is precharged when ϕ_1 is high and discharged during ϕ_2. A sense amplifier accelerates register reads. To move a value from a temporary register to a pointer register, the temporary registers are read on the falling edge of ϕ_1, fully discharging the bus before the rising edge of ϕ_2.

per buffer block) are on a single dual-rail bus — the value is read from a selected pointer register on the rising edge of ϕ_2, and with the help of a sense amplifier and a strong set of buffers is driven into a head or a tail register before the falling edge of ϕ_2. (2) One of the temporary registers is read, and its value is driven into both a pointer register and into a tail register (or both a tail and a head register, if the queue was empty). When a temporary register is read, the read occurs on the falling edge of ϕ_1. When ϕ_2 rises, and a pointer register is selected, the pointer register cannot significantly alter the value on the bus before the sense amplifier is enabled, ensuring that the correct value is driven into the pointer register. (3) The constant NULL (which *is* 0, in this implementation) is written into a selected pointer register.

Figure 7.13 shows the latency of operation (a) — moving a value from a pointer register to a head or a tail register. The rising edge of ϕ_2 causes a pointer register's enable signal to go high. Approximately 2 ns after ϕ_2 rises, the sense

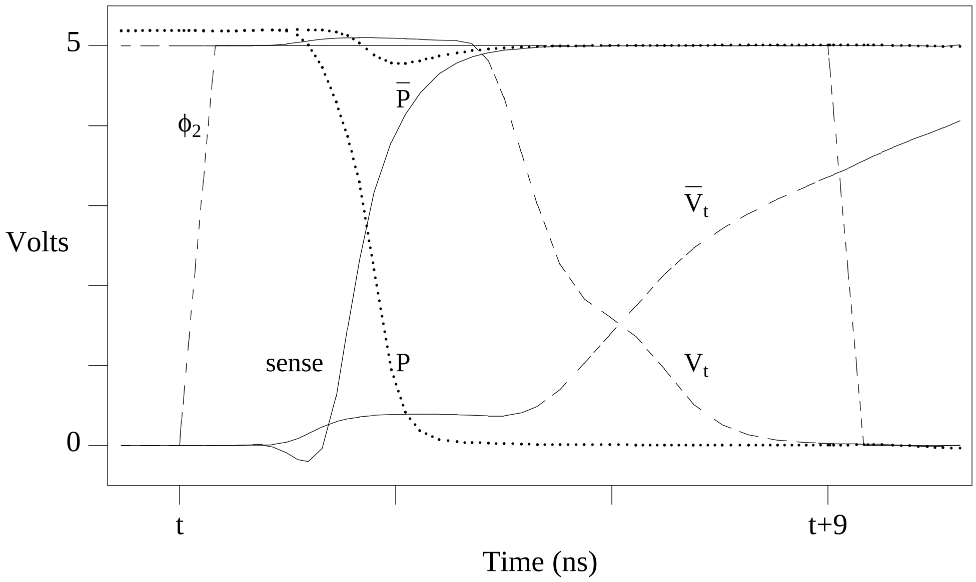

Figure 7.13: *Pointer Register to Tail Register Latency.* The pointer register read begins on the rising edge of ϕ_2. The value read from the selected pointer register is driven into the tail register well before the falling edge of ϕ_2.

amplifier enable (*sense*) rises, forcing the rails of the bus (P and $\overline{P}$) to opposite values. The value on the bus is driven by a buffer through the pass-gates which determine which register set is being written to (head vs. tail), finally flipping the value in the tail register (V_t and $\overline{V_t}$). The values $\mathrm{V_t}$ and $\overline{\mathrm{V}}_\mathrm{t}$ cross 6 ns after the rising edge of ϕ_2.

Figure 7.14 shows the performance of operation (2) (reading a temporary register, driving the value into a pointer, a head and a tail register). The temporary register is read on the falling edge of ϕ_1, giving it an extended period of time to assert its value on the bus. On the rising edge of ϕ_2, a pointer register is enabled, but it has little time to impact the voltage on the bus rails before the sense amplifier is enabled. The sense amplifier cranks the temporary register value into the pointer

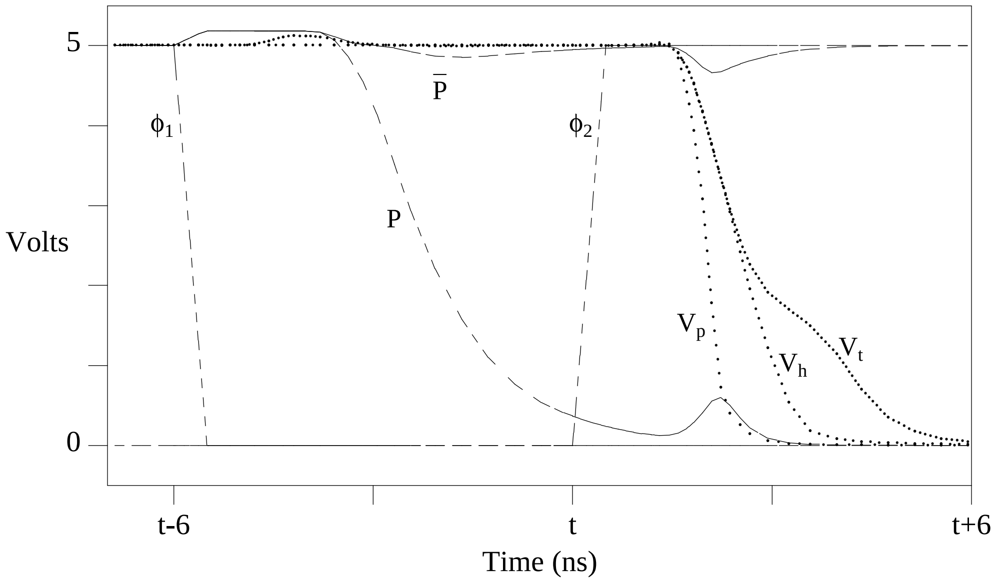

Figure 7.14: *Latency of Temporary Register to Pointer, Head and Tail Register.* The temporary register drives two rails of the bus (P and $\overline{P}$) on the falling edge of ϕ_1. On the rising edge of ϕ_2, a pointer register is selected. Approximately 2 ns later, the sense amplifier is enabled, and the value from the temporary register is cranked into the pointer register (V_p). The value is also written into both the head and the tail register (V_h and V_t). All writes complete well before the falling edge of ϕ_2 ($t+9$).

register. In the mean time, the buffer has been driving the value into the head and tail registers simultaneously; the write to all three registers completes less than 6 ns after the rising edge of ϕ_2 (more than 3 ns before the falling edge of ϕ_2).

Combining the simulation results of the memory array presented in Sec. 7.2.1 with the results presented in this section, we have defined the two-phase non-overlapping clock of the DAMQ Buffer Chip. It is a 50 MHz clock — ϕ_1 has a 15% duty cycle, ϕ_2 has a 45% duty cycle. There is a 5 ns underlap between ϕ_1 and ϕ_2, and a 3 ns underlap between ϕ_2 and ϕ_1. Most significantly, the datapath operations are not on the critical path; the clock frequency is defined solely by the bandwidth of the memory array. Thus, the DAMQ buffer transmits data with the same raw bandwidth that a (much more simple) FIFO buffer would.

7.2.3. The Finite State Machines

The DAMQ Buffer Chip is controlled by three finite state machines: FSM_{sto} (packet reception and storage), FSM_{rtr} (interaction with the router, buffer block enqueueing) and FSM_{fwd} (packet transmission, buffer block dequeueing). The implementation of these FSMs was performed under three constraints. First, the FSM circuits must utilize the two-phase clock defined in the two previous subsections. Second, the moving of a buffer block from the head of one queue to the tail of another must be accomplished in eight clock cycles or less (each block is eight bytes long and is transmitted / received a byte per clock cycle). The third and most significant constraint is that the hardware and firmware must support demand multiplexing.

7.2.3.1. The FSM Circuit

Figure 7.15 is a schematic of the finite state machine circuit. The FSMs are constructed from NOR-NOR PLAs. On the rising edge of ϕ_2, the output latches are opened and the input latches close, causing the state of the FSM to change. On the falling edge of ϕ_2, the input latches open, latching the current values, and the output latches close, driving the control lines with the signals for the next clock cycle. The exception to this timing is the output lines which are the state bits for the FSM. While driving the control lines on the falling edge of ϕ_2 allows control of

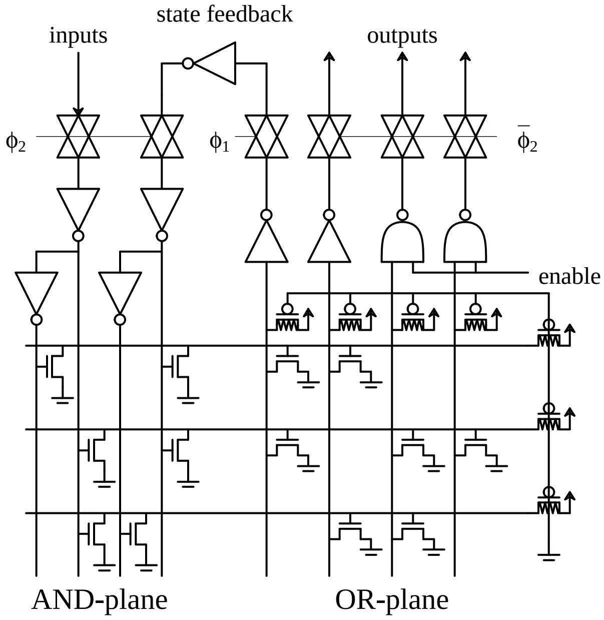

Figure 7.15: *Finite State Machine Circuit.*

actions which are to occur on the rising edge of ϕ_1 (such as driving a header bypass register over the crossbar), passing the state on the falling edge of ϕ_2 would create a race condition. So, the state outputs are passed on the rising edge of ϕ_1.

The FSMs utilize a pseudo-NMOS layout style — both the product and the sum lines are tied to Vcc via weak (resistive) P-transistors. If one or more of the N-transistors in the AND- or OR-plane are closed, the resistance of the P-transistors allows the lines to be quickly pulled down to less than 0.8v. If all of the N-transistors on a line are open, the voltage rises to 5 v. The output signal drivers are located between the OR-plane and the pass gates, rather than beyond the pass gates, to avoid using the non-restored product line voltages (logical zero signals whose level is above 0.5 v) as the capacitive source for the control line drivers when the output pass gates are open. Placing the buffers before the output pass gates necessitates the use of inverters between the state output and input signals.

As Fig. 7.15 indicates, the output drivers can be either inverters or NAND gates. Using NAND gates to drive control signals allows those signals to be masked by an enable signal. The use of an enable signal to mask outputs is discussed in Sec. 7.2.3.2.

The speed at which an FSM operates depends directly upon the size of the FSM, which is a function of the number of inputs and states, the number of product lines, and the number of outputs. FSM_{fwd} is the largest of the FSMs on the DAMQ Buffer Chip — it provides the functionality for transmission which is divided between two FSMs for packet reception. Combining the most heavily populated input, product and sum lines of FSM_{fwd}, the output signal settles ≈ 5 ns after the inputs settle. With every input to the FSMs stabilizing on or before the rising edge of ϕ_2 and the FSMs completing their state transition in 5 ns, the FSM outputs are stable well before the falling edge of ϕ_2.

7.2.3.2. Demand Multiplexing

Under ‘‘normal’’ operation, FSM_{fwd} and FSM_{rtr} use the data path on alternate clock cycles (FSM_{sto} does not share resources with the other FSMs, and thus does not participate in the multiplexing). The signal *SYNC* notifies the FSMs as to which clock cycles they own the data path by alternating high and low each clock cycle. When the SYNC input is low, FSM_{fwd} will control the data path the following clock cycle; when SYNC is high, FSM_{rtr} asserts control the following cycle. Unfortunately, strict time-division multiplexing will not satisfy our performance goals. The need for demand multiplexing of the data path was described in Sec. 7.1.3. Significant resources have been invested to implement demand multiplexing for the DAMQ Buffer Chip. In this subsection, we present the hardware interface between the DAMQ Buffer Chip and the external router and arbiter (it is interactions with these external units which triggers the demand multiplexing), then discuss the implementation of the *SYNC* signal (this is the signal which indicates to the FSMs which of them controls the data path), and finally we discuss how enable signals and control-signal masking were used to implement ‘‘immediate’’ demand multiplexing.

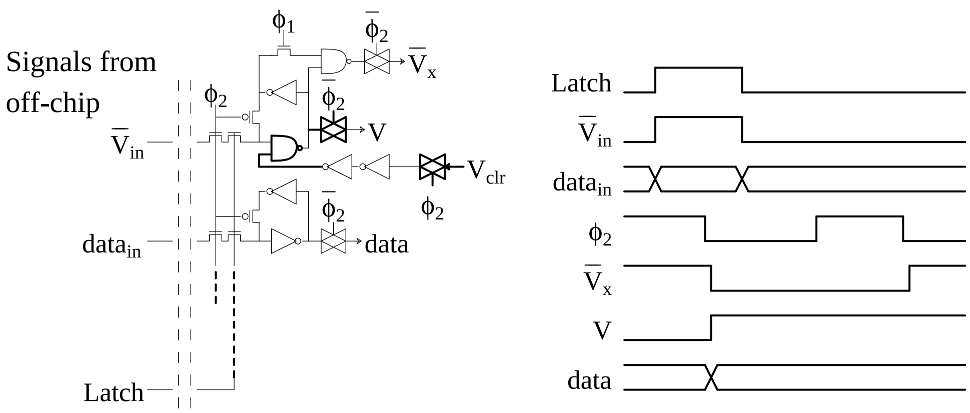

Figure 7.16: *Interface to Router and Arbiter.* Signals from the router or arbiter must be held through the falling edge of ϕ_2. The strobe signal ($\overline{V}_x$) is asserted for a full clock cycle after the data from the router or arbiter is latched (falling edge of ϕ_2 to falling edge of ϕ_2). Once a value is latched, it can only be cleared by the DAMQ buffer control via V_{clr}.

Figure 7.16 is a schematic of the interface register — the router and the arbiter each have an interface register through which they interact with the DAMQ Buffer Chip. The router / arbiter sets the valid signal ($\overline{V}_{in}$) and the input data, and then asserts *Latch*. These signals ($\overline{V}_{in}$, *Latch*, and the input data) must be asserted before the falling edge of ϕ_2 and held until after the falling edge of ϕ_2. Fig. 7.16 shows a timing diagram for input and output signals of the interface register. *Data* is the data being returned by the router or arbiter: a new header and queue identifier from the router, and a queue identifier from the arbiter. For their queue manipulations, $\mathrm{FSM}_{\mathrm{fwd}}$ and $\mathrm{FSM}_{\mathrm{rtr}}$ only differentiate between the free list and ‘‘the queue’’ when accessing head / tail registers — it is the router and arbiter interface registers which specify which queue is being accessed. V is the valid signal — when it is high, a valid queue is being presented by the interface register. $\bar{V}_x$, a ‘‘strobe’’ signal which is asserted for a single clock cycle following the routing or arbitration, is used by the FSMs and the SYNC signal generator to implement the demand multiplexing.

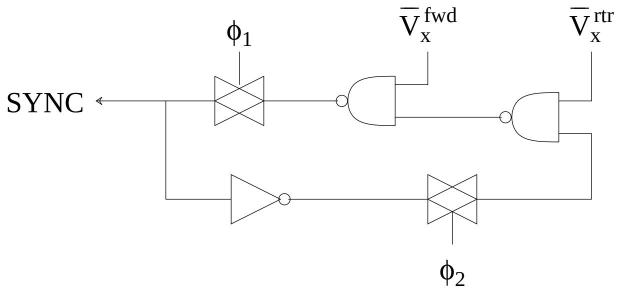

Figure 7.17: SYNC *Signal Generator*. The circuit which generates the signal which specifies which FSMs have control of the data path (*SYNC*).

Figure 7.17 is the schematic for the *SYNC signal generator*. As long as $\bar{V}_x^{rtr}$ and $\bar{V}_x^{arb}$ (the strobe signals from the the router and arbiter) remain high, *SYNC* is alternates high and low every other clock cycle. On any cycle that $\bar{V}_x^{arb}$ is low, *SYNC* will be high. Recall that $\mathrm{FSM}_{\mathrm{rtr}}$ controls the data path on the cycles *following* SYNC being high (i.e. $\mathrm{FSM}_{\mathrm{fwd}}$ controls the data path on the cycle that SYNC is high). Similarly, SYNC is forced low when $\bar{V}_x^{rtr}$ is low, unless $\bar{V}_x^{arb}$ is also low. Since the SYNC signal is taken as input by $\mathrm{FSM}_{\mathrm{fwd}}$ and $\mathrm{FSM}_{\mathrm{rtr}}$, the alteration of SYNC’s value cannot be acted upon by the FSMs until the following clock cycle. It was thus necessary to implement two additional mechanisms to support the ‘‘immediate’’ demand multiplexing which was our goal.

The first problem is how to allow an FSM to commence the queue manipulations associated with packet reception / transmission on the same clock cycle that the valid queue identifier becomes available. This is accomplished by masking the FSM output signals which control the data path. Fig. 7.15 shows output signals being driven either by inverters or by NAND gates. The NAND gates allow control signals to be masked by an enable signal. $\mathrm{FSM}_{\mathrm{rtr}}$ and $\mathrm{FSM}_{\mathrm{fwd}}$ each have an enable signal associated with them (EN^{rtr} and EN^{fwd}). The enable signal is asserted during the cycles that an FSM is controlling the reception / transmission of packets. The V signal from the interface register is asserted for the entire length of a packet reception / transmission. To allow the FSMs to initiate packet reception or transmission on the same cycle that the interface register asserts V (Fig. 7.16) (i.e. the cycle following a routing / arbitration), the FSMs actually assert the control signals which will initiate the queue manipulation while they are in the ‘‘ready’’ state, *before* V is asserted. These signals do not interfere with the other finite state machine, nor do they manipulate the data path prior to there being a valid queue identifier in the interface register, because these signals are masked until V goes high. When V^{rtr} for $\mathrm{FSM}_{\mathrm{rtr}}$ or V^{arb} for $\mathrm{FSM}_{\mathrm{fwd}}$ goes high, the enable signal goes high and the control signals are passed through to the data path. The FSMs also take V as input, notifying them that reception / transmission has begun.

The second problem is how to allow one FSM to ‘‘grab’’ the data path from the other FSM without waiting for the FSMs to change state. The previous paragraph described how the grabbing FSM will assert its control signals on the clock cycle that V goes high; what is needed is the ability to mask the other FSMs control signals on that cycle. Once again, the enable signals and control signal masking are the key. In addition to generating a signal V indicating the presence of a valid queue identifier, the interface registers generate a strobe signal $\bar{V}_x$ which is asserted low for the first clock cycle that V is asserted (Fig. 7.16). Fig. 7.17 showed how these strobes are used to manipulate SYNC, which signals to the FSMs which cycles they control the data path. These strobes are also used to mask the FSM data path control signals for a single clock cycle —

$$EN_{rtr} = V^{rtr} \cdot \bar{V}_x^{arb}.$$

$$EN_{fwd} = V^{arb} \cdot (\bar{V}_x^{rtr} + \overline{\bar{V}_x^{arb}}).$$

If both strobes are asserted on the same cycle, FSM_{fwd}'s control signals are not masked.

Having masked some of the outputs of one FSM so that the other can usurp its control of the data path, we must ensure that the skipped operation will be subsequently executed. This is accomplished by having both FSM_{rtr} and FSM_{fwd} remain in every state for two consecutive clock cycles. The FSMs have control of the data path for the first of these clock cycles, and relinquish control for the second; the SYNC signal indicates when to move on to another state. If the data path is usurped from an FSM, it will occur on the first of the two cycles in a given state. When this happens, the SYNC signal will maintain the same value that it had had the previous clock cycle. Since the FSM has remained in the same state *and* SYNC has its same value, the FSM repeats the actions of the previous clock cycle and ends up spending three consecutive cycles in that state before moving on. This mechanism works because only the signals which control the data path are state dependent; the remaining FSM outputs depend solely upon the inputs, and will thus not be repeated when a state is repeated.

We have thus developed an FSM architecture which supports demand multiplexing. Either FSM_{rtr} or FSM_{fwd} can usurp control of the data path from the other on any given clock cycle, *on the same clock cycle* that a valid output port becomes available. This ‘‘immediate’’ demand multiplexing is possible because (a) the FSM which is grabbing control is idle the cycles prior to gaining control of the data path and (b) the FSM which is losing control is simply delaying its queue manipulations for a single clock cycle, which does not impair its operation.

The mechanism which is instrumental to this implementation of demand multiplexing is the use of NAND-gate-masked outputs for the data path control signals. This allows an idle FSM to be asserting the control signals associated with the first queue manipulation operation on the cycles preceding the arrival of a valid queue signal. These signals are passed through to the data path on the same clock cycle that a new queue identifier becomes available. This also allows the control signals from the other FSM to be masked for a single clock cycle. The FSM whose outputs are thus masked is notified by the SYNC signal to re-assert the data path control outputs the following clock cycle. Thus, FSM_{rtr} and FSM_{fwd} immediately return to their pattern of utilizing the data path on alternating clock cycles.

7.2.4. The Floorplan

The previous sections have established that this implementation of the DAMQ buffer can operate at 50 MHz, and that it is the bandwidth of the memory array that defines this clock frequency. Thus, the DAMQ buffer operates at the same raw bandwidth as a FIFO buffer would (i.e. the complex control logic of the DAMQ buffer does not reduce its performance relative to the more simple FIFO buffer).

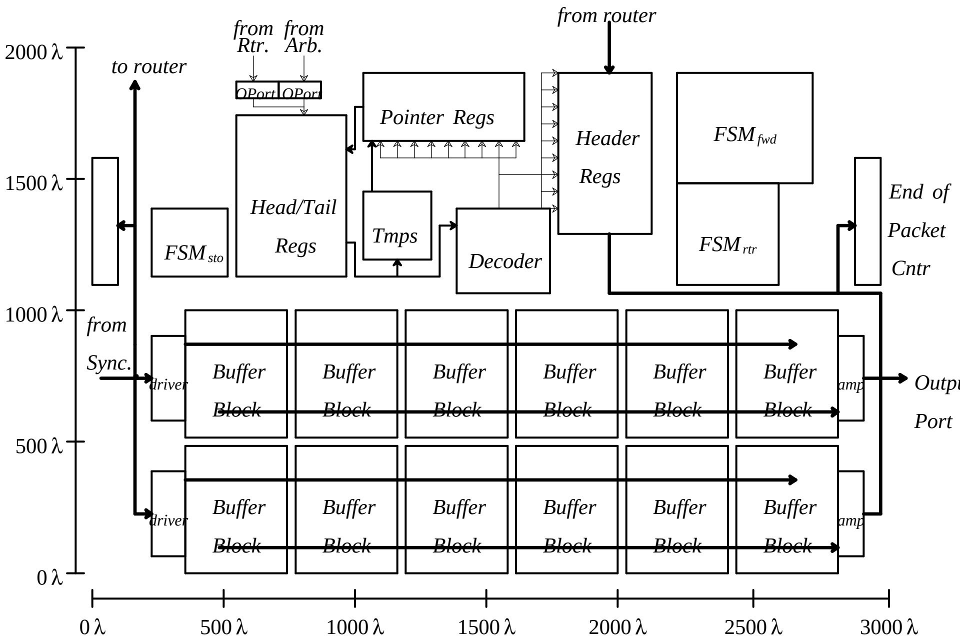

Figure 7.18: The floorplan for the DAMQ buffer.

The remaining issue, then, is a comparison of the DAMQ and FIFO layouts to evaluate their relative silicon usage. The floorplan for the DAMQ buffer is shown in Fig. 7.18. The 96 bytes of storage (slots for three thirty-two byte packets) are broken into two rows with separate read and write buses in order to reduce the latency of reads and writes and to produce a rectangular layout suitable for use as a building block in larger chips. This layout occupies approximately the same area as a FIFO buffer with 128 bytes of storage (slots for four packets). Our simulations indicate that a network of DAMQ buffers with three packet slots outperforms one with FIFO buffers of four packet slots (Ch. 4), achieving throughputs 23% higher than the maximum throughput of the FIFO network. Furthermore, even in comparison with the maximum throughput of a network with eight-slot FIFO buffers, a network with three-slot DAMQ buffers saturates at 10% higher throughput and a network with four-slot DAMQ buffers saturates at 24% higher throughput. We conclude that the DAMQ buffer architecture, as exemplified by the DAMQ Buffer Chip, is a more efficient utilization of silicon than is a FIFO buffer.

7.3. Summary and Conclusions

The DAMQ Buffer Chip is a VLSI implementation of a multi-queue buffer for use in multiprocessor and multicomputer communication networks. As an interface chip in the node of a multicomputer, it asynchronously receives packets from and transmits packets to other DAMQ Buffer Chips. As an example implementation of the DAMQ buffer, it demonstrates that a non-FIFO buffer can be efficiently implemented in VLSI. In this chapter, we have presented the design of the DAMQ Buffer Chip. Details of the implementation are given, including the floorplan and critical path timing.

We have demonstrated that the DAMQ buffer can operate at high clock rates, despite its complex control. Hence, other factors, such as the inter-chip links, are more likely to limit the raw bandwidth. Further, the DAMQ Buffer Chip supports a minimum switching latency of four clock cycles. This latency is determined by the serial dependencies of receiving the packet header, routing the packet, arbitrating the crossbar and then traversing the crossbar — the complexity of the DAMQ buffer does not add to this latency.

For a DAMQ buffer with four queues and a packet size of 32 bytes, it was shown that the DAMQ buffer will have one less packet slot available to it than a FIFO buffer occupying approximately the same chip area. Previous work has indicated that, with as few as two packet slots (64 bytes), a network with DAMQ buffers can outperform a FIFO buffer network which has an additional packet slot (Chs. 4 and 6). With three packet slots, the DAMQ buffer network will saturate at a throughput approximately 23% higher than a FIFO buffer network with four packet slots. The DAMQ buffer is thus shown to be a highly effective building block for packet switches, thus demonstrating that for VLSI switches, increased control complexity may lead to higher performance.