VLSI Design
1995, Vol. 2, No. 4, pp. 287–303
Reprints available directly from the publisher
Photocopying permitted by license only

© 1995 OPA (Overseas Publishers Association)
Amsterdam BV. Published under license by
Gordon and Breach Science Publishers SA
Printed in Malaysia

Design of Components for a Low Cost Combining Switch

SUSAN R. DICKEY* and RICHARD KENNER*
Courant Institute of Mathematical Sciences, New York University

We present the design for the two VLSI components used in a processor-to-memory interconnection network for a shared memory system. These components allow the combining of requests that are destined to the same memory location. The design contains both semi-systolic queues and an associative "wait buffer." Transition equations and schematics of the critical pieces of the design are included.

Key Words: *Interconnection networks, Combining, Shared memory, Fetch-and-ϕ, Systolic queues, Switch architecture*

1 INTRODUCTION

Communication between hundreds or thousands of cooperating processors is the key to massively parallel processing. The NYU Ultracomputer project has studied shared-memory architectures throughout the decade. Successful use of highly-parallel, shared-memory MIMD systems requires avoiding serial bottlenecks at all levels, from algorithm design through hardware components, and thus providing *scalability* as the number of processors grows. The nature of serial bottlenecks in applications, programming environments, operating systems, coordination primitives, system architecture, and within processor-to-memory interconnection networks has been researched extensively by our group. We have developed techniques either to eliminate such bottlenecks or to reduce their impact significantly.

The NYU Ultracomputer network has the topology of an Omega network [13] with a buffered VLSI switch at each node (see Figure 1), N processing elements (PEs) at each input and N memory modules (MMs) at each output. As discussed in [8], such a network has

- Bandwidth linear in N.
- Latency, i.e. memory access time, logarithmic in N, ignoring the effects of contention.
- $O(N \log N)$ identical components.
- Fixed number k of input and output ports for each node.
- Routing decisions local to each switch.

A major problem with such a network may be *tree saturation* due to *hot spots* (locations that receive a disproportionately large number of accesses) at the memory modules [18]. To facilitate synchronization operations and alleviate this problem, the NYU Ultracomputer combines fetch-and-ϕ operations (including loads and stores) at the switches.

Fetch-and-$\phi(X, e)$, where X is an integer variable and e is an integer expression, returns the (old) value of a memory location X and replaces it with $\phi(X, e)$. Concurrent fetch-and-ϕ operations must satisfy the *serialization principle*. Fetch-and-ϕ operations simultaneously directed at X cause the final value of X to be the same as the result of executing the operations in some serial order, and each operation returns a value corresponding to an intermediate value of X in a serialized execution. Fetch-and-ϕ operations can be combined in the network for any associative operator ϕ [9]. Since combined requests can themselves be combined, any number of concurrent memory references to the same location can be satisfied in the time required for one shared memory access from a single PE.

*Supported by the U.S. Department of Energy grant number DE-FG02-88ER25052 and the National Science Foundation grant number MDS-8301768.

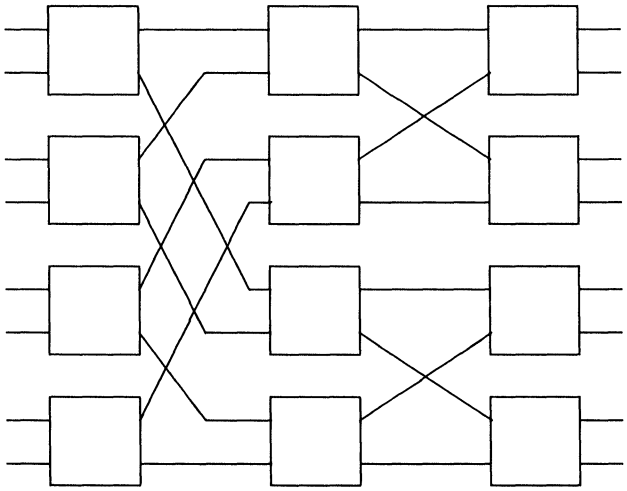


FIGURE 1 An 8 × 8 Omega network.

This paper describes the detailed VLSI design of components for a combining switch. These components have been fabricated, tested, and are being used in the network of a 16 × 16 Ultracomputer prototype. Section 2 describes the operating constraints and CMOS implementation of a semi-systolic queue that is useful as a component in many kinds of switches, particularly if a cut-through switching strategy is employed. Section 3 specifies the combining switch architecture actually constructed, including packaging, packet format, operations supported and flow control logic. Section 4 discusses design choices for the arrangement and arbitration of buffers within a switch. Details about the design of the forward path component are given in section 5, followed by a description of the return path component in section 6.

2 SYSTOLIC QUEUE DESIGNS

Systolic queue designs, as described in [10], have advantages even for non-combining switches. Memory-based FIFO designs require that input and output buses be connected to all storage elements; the capacitance on these buses must be charged and discharged for each insertion and deletion. Systolic designs require external connections only to the first slot in the queue (see Figure 2). Items enter at the edge of the IN row and shift right according to certain rules until all items in the OUT row have been passed, then move down and begin shifting left until they exit from the leftmost slot.

The semi-systolic design we have implemented requires only two global control signals distinguishing the four possible states of the queue: normal (both IN and OUT rows are moving), full, blocked and emptying (see [1]). These global control signals can be precomputed, at the expense of an extra message slot to accept messages after the switch tells the preceding stage that it is full, and can be implemented efficiently as qualified clocks [3]. Furthermore, systolic queue designs have the advantages of regular layout and limited connections per cell characteristic of systolic structures in general [14].

The systolic combining queue design has the further advantage of distributing the comparison logic used to find matching messages in a way that does not add significantly to the cycle time of the switch. Memory-based designs, like those described in [21, 22], require a comparator connecting the input bus to every message slot in the queue. Such a comparison is likely to be quite slow and must be done in series with insertion, since the destination of the message on the input bus will be different depending on the result of the comparison. In a systolic combining queue, matching can be done in parallel with insertion.

Guibas and Liang's design [10] is fully systolic, according to Leiserson's definition [14], requiring no global signals except a clock, but does not allow items to be inserted and deleted on the same cycle. Snir and Solworth [20] developed a design that distributes global information about whether the queue is blocked to each cell in order to be able both to delete and to insert an item every cycle. If the queue is not blocked, a deletion always occurs and all items in the OUT row simultaneously move left. Unlike the Guibas and Liang design, where information can only be moved into empty cells, the global information allows all the items to move in lock-step into the neighboring location, providing an unbroken stream of items to be deleted. To handle the problem of finite queue size, this design also distributes global information about whether the queue is full which determines whether the items in the IN row move right or retain their position, allowing them also to move in lock-step as long as the queue is not full.

2.1 Operating Constraints

Snir and Solworth [20] list the following invariants as necessary for correct operation of the queue:

1. The full slots in the OUT row occupy an initial segment of consecutive locations (i.e., there are no "holes" in the OUT row).

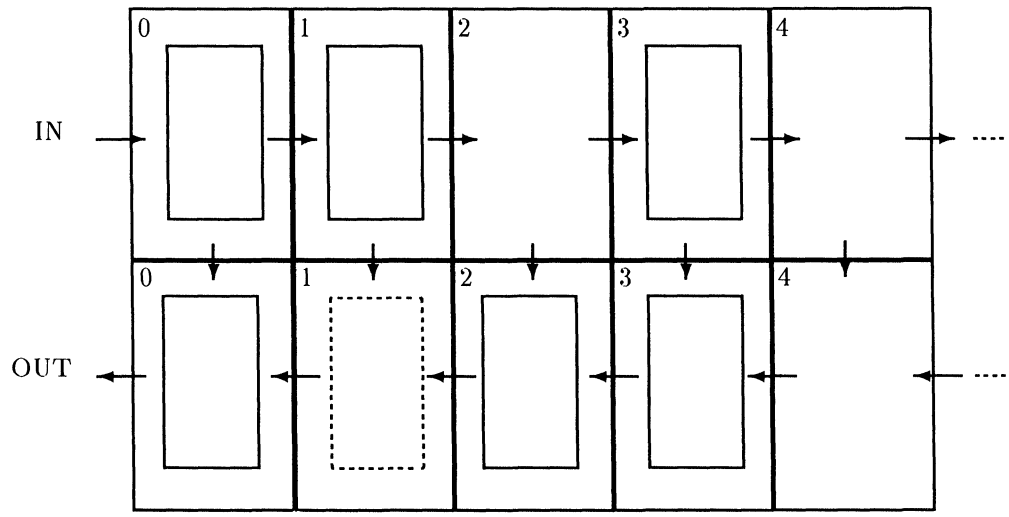


FIGURE 2 A systolic queue design.

2. When the queue is neither blocked nor full, the difference between the largest index of a valid location in the OUT row and the largest index of a valid location in the IN row is non-negative and odd.

They list the following conditions which must be satisfied by the control logic to maintain these invariants:

1. The number of cycles between two consecutive insertions must be even.
2. When the queue is blocked, it must remain so for an even number of cycles.

These constraints are necessary to prevent items from getting out of order. Holes or out of order messages can appear in the OUT row only if at some cycle both rows are "lined up" at the rightmost edge and at the following cycle both rows are moving. For correct operation of the queue, one must ensure that this situation can never arise.

In the switch, each item in the queue is a packet of a message. As shown in [1], a two-row systolic queue design, based on messages with an even number of packets, two global control signals, and simple restrictions on when these global signals can change, satisfies the invariants of Snir and Solworth, ensuring correct operation. The transitions and control signals for this queue are described below.

The two global control signals are called in_moving and out_moving. To describe the queue's operation more precisely, using a standard two-phase clocking scheme like that in Mead and Conway [16], let in1(j) and in2(j) correspond to the values of the item in slot j of the IN row at the end of phase 1 and phase 2, respectively. Similarly, out1(j) and out2(j) are the values for slot j of the OUT row. The Boolean variables valid1(x, j) and valid2(x, j) are true if there is an item in slot j of the x row at the end of the corresponding phase, false if the slot is empty. Using this notation, the phase 1 transitions of the queue are:

$$\begin{aligned}
\text{in1}(j) &:= \text{if in_moving then in2}(j-1),\\
&\quad\ \text{else in2}(j)\\
\text{out1}(j) &:= \text{if out_moving then out2}(j+1),\\
&\quad\ \text{else out2}(j),\\
\text{valid1(IN}, j) &:= \text{if in_moving}\\
&\quad\ \text{then valid2(IN}, j-1)\\
&\quad\ \text{else valid2(IN}, j),\\
\text{valid1(OUT}, j) &:= \text{if out_moving}\\
&\quad\ \text{then valid2(OUT}, j+1),\\
&\quad\ \text{else valid2(OUT}, j),
\end{aligned}$$

where the index $j = -1$ corresponds to the values of the input port. The phase 2 transitions are

$$\begin{aligned}
\text{in2}(j) &:= \text{in1}(j),\\
\text{out2}(j) &:= \text{if valid1(OUT}, j) \text{ then out1}(j)\\
&\quad\ \text{else in1}(j),\\
\text{valid2(IN}, j) &:= \text{valid1(IN}, j) \wedge \text{valid1(OUT}, j)\\
\text{valid2(OUT}, j) &:= \text{valid1(IN}, j) \vee \text{valid1(OUT}, j).
\end{aligned}$$

The following constraints must be enforced:

1. The first packet of a message is accepted only at an even cycle.
2. All messages are of even length.
3. The control signal in_moving can change its value only before the transitions of an even cycle.

4. The control signal out_moving can change its value only before the transitions of an odd cycle.

A cell j in the x row is said to be occupied if valid2(x, j) is true at the end of the cycle.

A queue signals full when it can no longer accept messages. If there are $2n$ packets in a message, then the queue full condition is equivalent to having the nth slot from the right in the OUT row occupied. Data accept (DA) to the sending queue must be lowered when this slot becomes occupied. When this slot first becomes occupied, the parity restrictions ensure that the IN row cannot be receiving the last packet of a message. In the worst case the queue may be receiving the first packet of a $2n$ packet message and the IN row must move $2n - 1$ more times. Even if the output is blocked, there will still be space in the queue for all packets. The sending queue must not initiate a new message after DA has been lowered. When that slot becomes unoccupied, DA may be raised, and there will be room for any new message initiated.

In our queue implementations, layout considerations cause us to use the valid1 signal rather than the valid2 signal to produce the DA signal. Because in the detailed logic of the queue valid2 becomes true while valid1 is still false, we actually take the valid1 signal from the $n + 1$st slot from the right of the queue. If there were never any holes in the IN row and the OUT row were blocked, the valid2 signal of the nth slot would become true one cycle after the valid1 of the $n + 1$st slot; so using this more conservative signal rarely makes a difference. Simulations show no difference between the two schemes in the number of messages contained in the queue when DA is lowered.

2.2 CMOS Implementation of the Semi-Systolic Queue

We control movement in the single-bit data cell with qualified clocks derived from two-phase clock signals, as is done in Mead and Conway's stack design [16], with dynamic latches replacing the use of single transistors as transmission gates. In a straightforward extension of the nMOS design, four basic clocks ($\phi 1$, $\overline{\phi 1}$, $\phi 2$ and $\overline{\phi 2}$) must be provided in order to have non-overlapping clock phases. As in nMOS, the non-overlap time must be guaranteed in the presence of clock skew, so the requirement of non-overlapping clock phases can result in considerable dead time in each clock cycle. In CMOS this problem is compounded by having four clock signals to distribute.

The clocked CMOS (C^2MOS) latch can be used to replace an inverter followed by a pass transistor in an nMOS implementation. The key difference between the C^2MOS latch and the inverter followed by a transmission gate is that with the latch only one clock controls each transition of the output node, i.e., the output can be driven low only if one clock goes high, and it can be driven high only if the other clock goes low. This is true for every latch: the output of a latch can only be driven low when some clock is driven high and vice versa.

The NORA methodology described in [7] uses this latch to construct two-phase pipelined circuits that use only two basic clocks, ϕ and $\overline{\phi}$. In this methodology, *phase 1* corresponds to the time when ϕ is high and $\overline{\phi}$ is low, and *phase 2* corresponds to the time when $\overline{\phi}$ is high and ϕ is low. Therefore, in a standard NORA system, a phase 1 latch has ϕ on its N transistor and $\overline{\phi}$ on its P transistor; a phase 2 latch has the reverse. Restrictions on the parity of inversions between latches prevent data from flowing through a pair of latches during overlap periods when both ϕ and $\overline{\phi}$ have the same value. In [3] we augment the NORA methodology to include qualified clocks and describe how to avoid noise problems with NORA logic when implementing a systolic queue design.

Our basic data cell is shown in Figure 3. Data movement within a row (IN or OUT) occurs on phase 1 (when clock signal $\phi 1$ is high); data transfer from IN to OUT occurs on phase 2 (when clock signal $\phi 2$ is high). Pass transistors control the horizontal and vertical data movement from cell to cell in the data path (Figure 2). The decision as to which direction data will move in a phase is computed in the previous phase; thus data movement and control computations are completely overlapped.

In Figure 3, IN and OUT are input signals from neighboring cells; in2 and out2 are output signals. The qualified clock FI ("Flow In") is high only when $\phi 1$ is high and in_moving is true; HI ("Hold In") is high only when $\phi 1$ is high and in_moving is false. FO and HO are defined similarly for the OUT row. TRV ("Transfer Vertical") and TRH ("Transfer Horizontal") are a pair of $\phi 2$-qualified clocks controlling whether out2 is set from the IN or the OUT row; the pass transistor with gate $\overline{\phi}$ is used to transfer the phase 1 value to in2.

3. COMBINING SWITCH ARCHITECTURE

In implementing our combining switch, we avoided design choices that hurt performance for non-com-

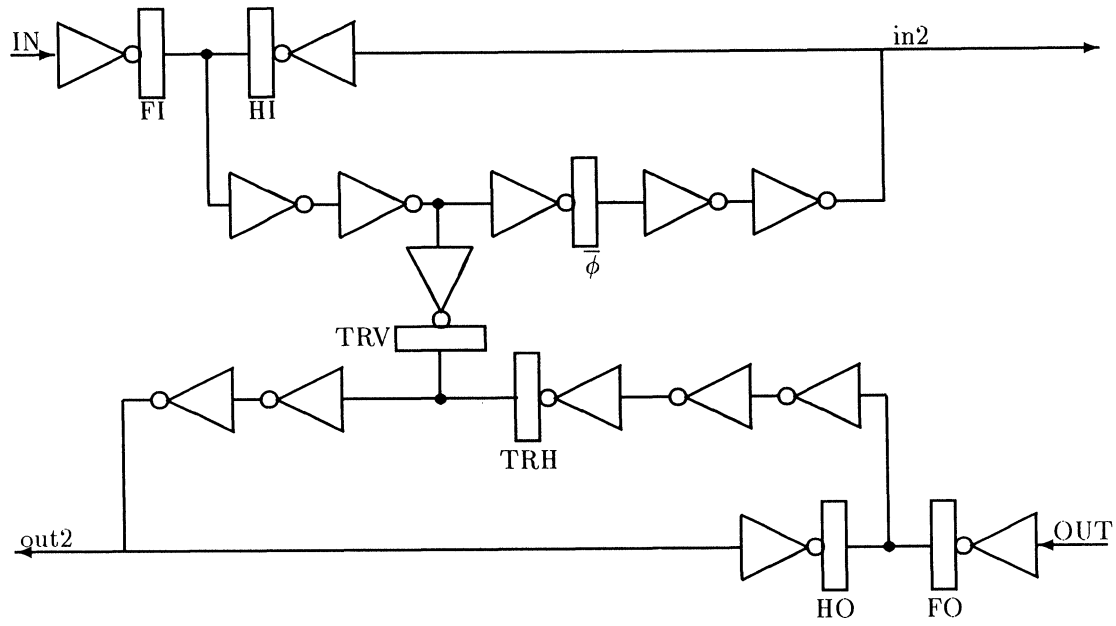


FIGURE 3 NORA CMOS implementation of the basic cell of a non-combining queue

bining traffic. Our goal was to design a network that provided the highest bandwidth and lowest latency possible, given packaging, processing and human resource constraints. In particular, we were careful to avoid design decisions that would increase latency for light traffic.

The combining switch we have designed and fabricated for use in the 16 PE NYU Ultracomputer prototype is a 2 × 2 switch node shown in Figure 4. Each node is composed of four each of two types of custom VLSI design blocks: *forward path components* (FPC) and *return path components* (RPC). Control for the switch is distributed as tri-state selection logic in each chip. Routing information is included in the message. Flow control avoids the necessity of acknowledgment for messages and allows pre-computation of signals that control data movement from stage to stage. The components accept two and four packet messages and allow the first packet of a message arriving when the queue is empty to exit at the next cycle.

3.1 Packet Format

Each message consists of an address packet and from zero to two data packets. Data packets are 32 bits wide, to simplify the interfaces at the processor and memory. Messages with zero or two data packets are padded with a null packet to be of even length. The forward path address packet contains the PE/MM address field, the opcode, the memory address and the outstanding request index (ORI) field. The wait buffer address packet contains two PE/MM address fields and two ORI fields. The return path address packet contains the MM/PE address field, the opcode, the ORI field, and the SWD (store was done) bit.

The 4-bit PE/MM address allows 16 MMs to be addressed by each of the 16 PEs. At each stage in the network the routing decision is made on the basis of the high order bit of the PE/MM address field. Before being sent to the next stage, this bit is replaced with the address of the input port at which the message arrived (0 or 1) and the PE/MM address is rotated one bit left. At the memory this field is reversed and used as the MM/PE address field on the return path.

A 4-bit field is reserved for the op code. In addition to its use at the memory, the op code is used by the switches to control combining and decombining of messages and to signal the length of a message. The operations supported are described in the next section.

The 4-bit ORI field contains a number assigned by the PE when the message is issued. A PE must have only one message with a given ORI in the network at a time; the combination of PE/MM address and ORI is a unique identifier for a message at a switch. One such 8-bit identifier must be sent to the RPC to identify the message waiting to be decombined and another to identify the message that has been sent on to memory. Thus only 16 bits are

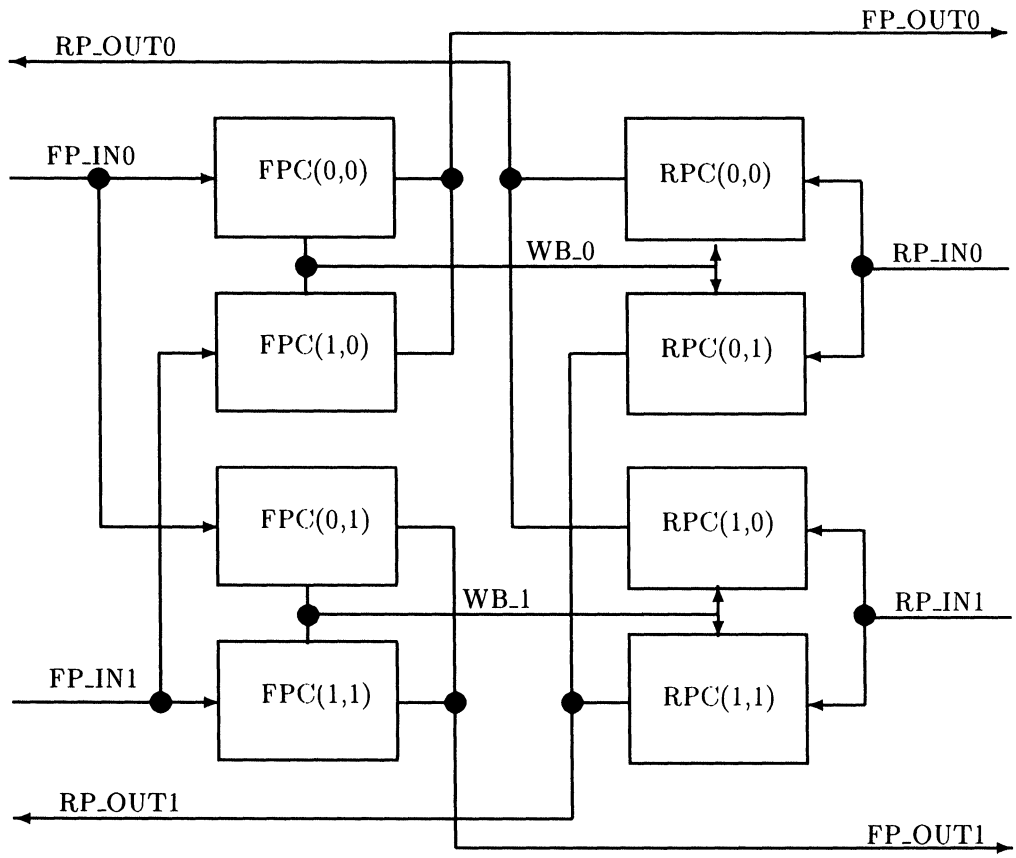


FIGURE 4 Block diagram of combining switch showing connections of forward path components (FPC) and return path components (RPC).

needed in the address packet of a message sent to the wait buffer in the RPC. There are 16 extra bits since the packet size is determined by the 32-bit data width. On the return path as well, the data width determines the packet size, since the memory address is not returned to the PE, and there are 15 unused bits. The SWD bit is returned to indicate whether or not a store was actually done for the conditional Fetch and Store operations.

3.2 Operations Supported

We implement a set of combinable memory requests that have been found useful in the development of parallel algorithms, including fetch-and-ϕ operations as well as loads and stores. If two messages with the same combinable op code meet in a queue, they will be combined, as described in [9]. See Table I for the ALU operations which must be performed on the forward and return paths to implement these requests.

Combinable single word loads are implemented as fetch-and-adds with an operand of 0; combinable single word stores are simply fetch-and-stores with the returned value ignored. In addition, the four-bit op code allows eight non-combinable operations that are forwarded to the MMs without receiving any special processing. The non-combinable operations can include partial word loads and stores, broadcast and reflect (see [6]). Only broadcast requires any special processing in the network.

All messages are two packets in length unless the op code is all ones. This op code is used for Store Double in the forward path and Load Double in the return path; the memory must make the op code change. Both Store Double and Load Double are transmitted as four packet messages, with an empty final packet.

3.3 Flow Control Logic

As described in Section 2.1, the construction of the queues requires that there be an even number of packets per message and that switches distinguish

TABLE I
ALU Operations for the Memory Requests Implemented in the Combining Switch

Memory Operation	Forward Path	Return Path
Fetch and AndNot	OR	OR
Fetch and Or	OR	OR
Store Double	SELECT Chute	Don't Care
Fetch and Store	SELECT Chute	SELECT D
Fetch and Store if $= 0$	SELECT Out	if SWD SELECT D else SELECT RP_IN
Fetch and Store if ≥ 0	SELECT Out	if SWD SELECT D else SELECT RP_IN
Fetch and Add	ADD	ADD
Load Double	Don't Care	SELECT RP_IN

even and odd cycles. Reception of messages starts only at even cycles while transmission of messages starts only at odd cycles. Cycles that are even for one switch are odd for its predecessors and successors. There is no explicit even signal for the queues; end of message (EOM) signals for input and output are generated on-chip to always change at cycles of a given parity and are used to enforce the parity restrictions on operation. An explicit even signal is generated on-chip only for the wait buffer. The wait buffer and the interaction between blocks in the RPC have been designed in such a way that the cycle parity of the FPC and RPC in the same switch are identical (see section 6).

Each port has two protocol bits: a data valid bit (DV) traveling in the same direction as the data and a data accept bit (DA) traveling in the reverse direction. These bits, in conjunction with the routing bit, regulate the transmission of messages through the network. A sender asserts DV when it wishes to initiate a message transmission. Independently, a receiver asserts DA when it is able to accept a new message. A message transfer starts only if both DV and DA are asserted and the cycle parity is correct. Signals controlling the blocking and unblocking of the queue are ignored during cycles when a message transfer cannot be started (that is, they are looked at no more often than every other cycle), so they can be set ahead of time to overlap data transfer and flow control operations.

Note that this is not strictly speaking a handshaking protocol: DA is not an answer to DV, nor an acknowledgement, but is issued independently and simultaneously. The sender is transmitting the data on the data lines whenever DV is asserted. If it receives DA, it assumes the data has been accepted and proceeds with the next packet. No provision for retry is necessary.

Our protocol requires that transmission of a message not be halted once it begins. This requires reserving enough buffer space whenever DA is asserted to accept the rest of the message. In our implementation, this means four packets.

Queues paired at an input keep track of the length of valid messages sent to the other queue in order to know when to look at the input for the start of a new message. The op code bits are decided and either the second or fourth cell of a shift register is set to 1; the output of this shift register is then used to signal input EOM. Whenever input EOM is asserted, and on the next cycle DV is asserted, a new valid message is assumed to begin for the queue with output port number equal to the routing bit.

4 DESIGN CHOICES FOR BUFFER STRUCTURE

Consider a $k \times k$ crossbar switching component in a multistage interconnection network. Its basic function is to forward messages from any of its k inputs to any of its k outputs. It may include buffers to hold messages in case of conflicts for the output ports or blocking from later stages. These buffers may be associated with either input or output ports and may perform extra functions such as combining messages destined for the same memory location or sorting messages according to destination in a later stage. This section discusses the design choices in the arrangement and arbitration of these buffers for the combining switch.

4.1 Arrangement of Buffers

Switches can be classified according to the presence or absence of buffers, and according to the location of the buffers [4]. Figure 5 shows the three types of 2×2 switches that we have considered. The first

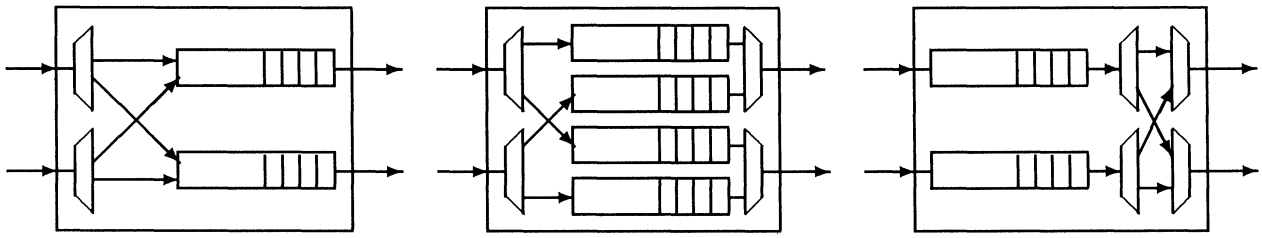


FIGURE 5 Three buffered switch architectures

design, Type A, contains two queues, each queue capable of accepting two messages simultaneously. The queues in this Type A switch can insert two messages into a queue every cycle. Alternative switch architectures use one-input, one-output queues, which are simpler to design and need fewer hardware resources per queue item. Type B has four of these simpler queues, one for each input/output pair. Type C has only two queues, one at each input, and is very attractive to hardware designers because of its simplicity. However, the bandwidth of the Type C switch is limited due to blocking when both queues have a message for the same output; in a 2 $\times$ 2 switch this bandwidth limitation means that a message will be present on an output link no more than 75% of the time, under uniform traffic [17].

We have implemented Type B switches, for the following reasons:

- Output rate and average queue lengths are equivalent to Type A for uniform traffic. [17].
- Implementing the two-input queues for a Type A switch requires either the serialization of the two insertions, or a more difficult two input systolic queue design, which has 2 IN rows and one OUT row and carries out arbitration for each slot of the OUT column rather than just at the output [20].
- Type B switches, which require only one input, one output, and for combining switches, one port to the wait buffer, have greater packaging flexibility than Type A, which require two input ports, one output port and one wait buffer port per switch.

A cost of using Type B switches is fewer combines in the earlier stages of the network, since messages from different PEs that might combine cannot be together in a queue until one stage later than with Type A switches. According to simulation results in [15], combining networks with Type B switches still show significantly better performance than non-combining networks.

4.2 Arbitration of Buffers

The arbitration rules for the queues paired at an output in Type B switches must not reduce bandwidth or starve one of the queues. The analysis described in [17] assumes that if one queue is empty and the other is not, the non-empty queue will drive the output, and that if both queues are not empty, the queue driving the output will be selected at random.

In practice, it is not easy to find a simple and reliable digital CMOS circuit that will select each output randomly but with equal probability. Strict alternation of outputs, ignoring whether or not the selected queue has a message, is easy to implement but increases latency under light load. Our first implementation alternated the selection whenever both queues had messages, chose the non-empty queue if only one was empty, and rememberd which queue was selected last when both queues were empty, in order to give priority to the queue which had not sent the last message. We implemented this in CMOS as a single AOI gate with 4 inputs but no more than two transistors on any path between a power rail and the output, plus the latches necessary to save the old value of *select*. All logic was carried out double-rail, for NORA correctness, since the AOI gate required both the old *select* signal and its inverse.

Unfortunately, as we discovered when simulating the design in a network environment, this logic does not take into account what happens when the queue is blocked. If the next stage blocks for an odd number of arbitrations while neither queue is empty, unblocks for long enough to allow one message to exit and then blocks again for an odd number of arbitrations, the same queue will get repeated service and the queue associated with the other input will be starved. As we discovered while running a version of the switch with this arbitration logic, such starvation is not necessarily an uncommon or self-correcting event; certain memory behavior can cause it to occur consistently. To prevent this starvation due

to arbitration while blocked, the information about whether the output is blocked or not must be included in the arbitration logic. While the output is blocked, the selected queue and the priority must not change. Thus no queue will get repeated service after an interval of blocking if the other queue has a message to send.

This logic can be generalized to arbitrate k queues at an output. In that case the input to the logic is k empty signals. Selection priority is given to queue i if queue $i - 1$ modulo k was the last queue to send a message. If queue i is empty, the next non-empty queue j obtained by incrementing i modulo k is selected to send a message, and priority will be given to queue $j + 1$ at the next arbitration. If all queues are empty, or if the output is blocked, priority stays at queue i. This generalization was implemented for $k = 4$ on the return path, where each RPC has one queue for messages received from the previous stage and one for decombined messages, making a total of four queues at each output.

5 FORWARD PATH COMPONENT DESIGN

The forward path component (FPC) is essentially a single systolic combining queue, with the ALU embedded in the first slot of the queue, rather than on the critical off-chip path. Embedding the ALU in the first slot of the queue allows combining to be done in parallel with data movement off-chip. Loss of combining in the first slice does increase memory latency due to queueing delay in the presence of hot spots, as shown in [5], but the alternative is to increase the latency for all traffic patterns, either by degrading the cycle time or losing the property of having single cycle transmission time when the queue is empty.

Control logic on each component decodes the op code to determine the ALU operation and the length of the message. The output is blocked if a wait buffer data accept signal (WB.DA) or a data accept signal from the next stage (FP_OUT.DA) is low. The QNE (not empty) signal from the paired queue is used to determine which component has priority to drive the output port, as described in section 4.2; out_moving is true if the output is not blocked and the component has priority to drive the port. The DA (data accept) signal sent to the previous stage is derived from the queue full signal and latched on the input end of message (EOM) signal. The EOM signal is derived from the op code, which is used to differentiate two and four packet messages.

5.1 Combining Queue

The semi-systolic combining queue in the forward path component adds another row to the queue design described in section 2.1 (see Figure 6). A non-combining queue has a hardware cost similar to that of a shift register; the extra row costs about 50 percent more, while the comparator to recognize matching messages adds only 8 transistors to each cell (Figure 7). The comparator consists of a dynamic XOR gate followed by an inverter whose output can pull down a pre-charged *match* line which is shared by all the cells used for the match. These gates form a short chain of Domino CMOS logic, as described in [12, 19].

The comparator checks for equality between every slot in the IN row and the corresponding slot in the OUT row. As messages move to the right along the IN row they are compared with the messages in the OUT row below. If a match occurs, data from the IN row moves to the CHUTE. Comparison can be done in parallel with data movement because the message in the IN row is copied to the CHUTE as long as the latter is empty. A valid bit for each row in the slot is set based on the result of the match.

If the IN and the OUT row move simultaneously, an entering packet will be matched against alternate packets. If every message has two packets, an entering address packet will be matched against the address packet of every message that was in the queue when it arrived except the first (since we have placed the adder in the first slice and do not allow combines there). The systolic design we use, which allows a packet to enter one cycle and exit the next, already constrains messages to have an even number of cycles between the start of one message and the start of the next (see [20] and section 2.1), so the addition of combining places no further constraint on the number of packets in a message. Since messages can be either 2 or 4 packets long and the IN and OUT rows need not be moving simultaneously, a match occurs only if both the IN row and the OUT row contain address packets. To ensure that matches take place only between two address packets, an internal start of message (SOM) signal enters the queue with the first packet of a message.

A block diagram of the combining queue is shown in Figure 8. The input to the first valid cell is FP_IN.DV; the outputs are FP_OUT.DV and WB.DV. HI, FI, HO and FO have the same meaning as in section 2.2. OTRH, OTRV and CTRH, CTRV are the analogues of TRH, TRV for OUT and CHUTE, respectively. SOM (start of message) is true if the packet in the slot is an address packet.

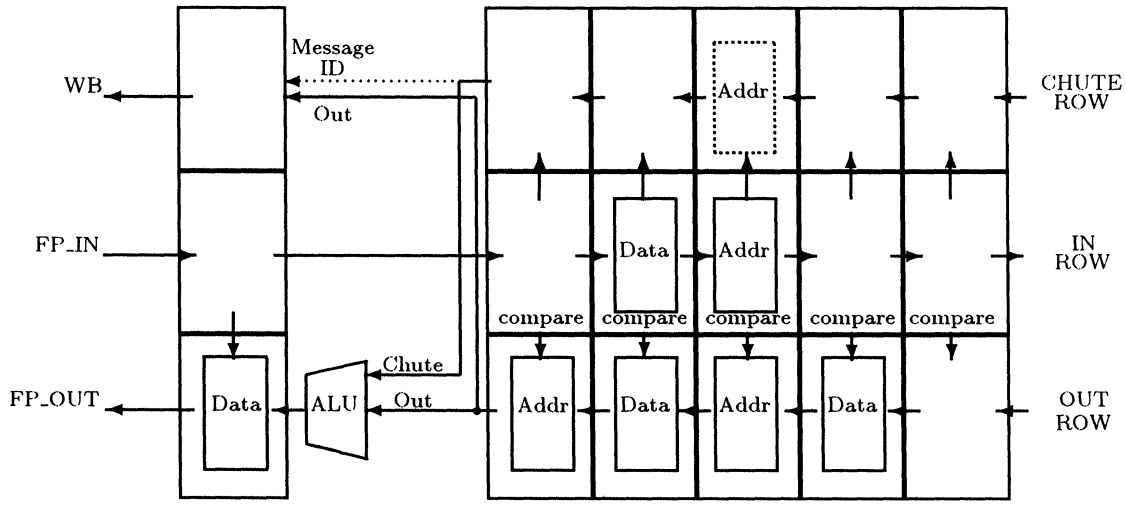


FIGURE 6 Design of systolic combining queue.

The transitions for slice j of a combining queue are shown in Figure 9. The chute_transfer signal flags whether data packets of a message should move into the CHUTE row or not. Chute_transfer is a somewhat complicated function of the match line for that slice and the state of the queue, including in_moving, out_moving and the parity of the cycle. The chute_transfer signal is asserted or deasserted at the start of a valid message, depending on whether or not a match has occurred. The chute_transfer signal must follow the movement of each remaining packet in the message as it proceeds along the IN row, so that the packet is transferred to the CHUTE row if chute_transfer is asserted, and to the OUT row otherwise. If both rows are moving, the chute_transfer signal must stay in place, and the transfer is made at the point where the match was detected. When the IN row is moving and the OUT row is not moving, the chute_transfer signal must move to the right every second cycle, since the packets are piling up behind the first packet. When the IN row is not moving and the OUT row is moving the chute_transfer signal must move to the left every second cycle. If neither row is moving, the chute_transfer signal must

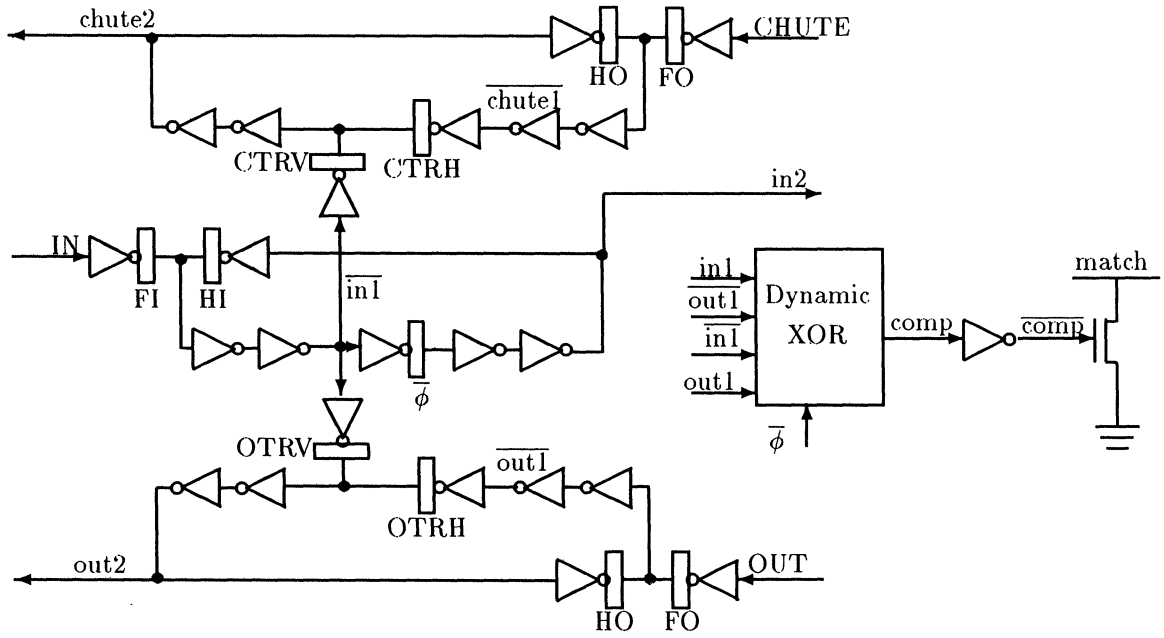


FIGURE 7 Schematic of a single cell of the combining queue in the forward path component.

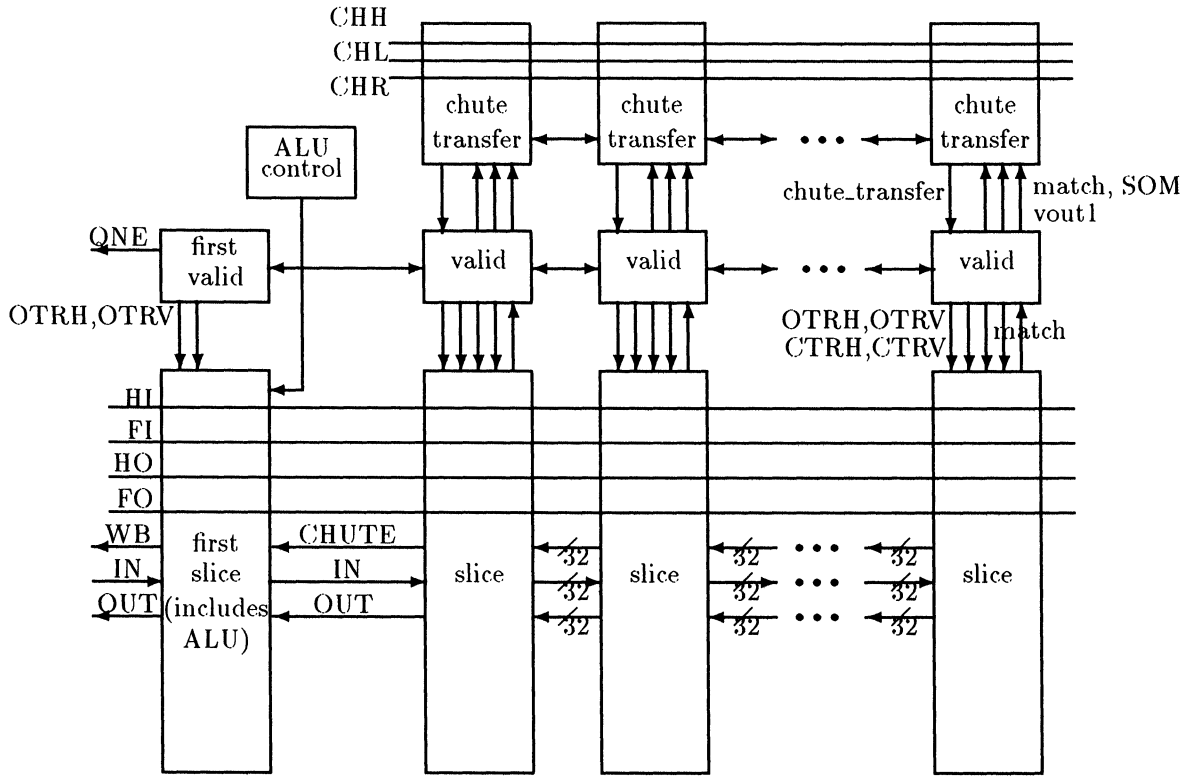


FIGURE 8 Block diagram of a combining queue implementation.

$$
\begin{aligned}
\text{in1}(j) &:= \text{if in\_moving then in2}(j-1) \text{ else in2}(j),\\
\text{out1}(j) &:= \text{if out\_moving then out2}(j+1) \text{ else out2}(j),\\
\text{chute1}(j) &:= \text{if out\_moving then chute2}(j+1)\ \text{chute2}(j),\\
\text{valid1(IN, } j) &:= \text{if in\_moving}\\
&\qquad \text{then valid2(IN, } j-1)\\
&\qquad \text{else valid2(IN, } j),\\
\text{valid1(OUT, } j) &:= \text{if out\_moving}\\
&\qquad \text{then valid2(OUT, } j+1)\\
&\qquad \text{else valid2(OUT, } j),\\
\text{valid1(CHUTE, } j) &:= \text{if out\_moving}\\
&\qquad \text{then valid2(CHUTE, } j+1)\\
&\qquad \text{else valid2(CHUTE, } j),\\
\text{in2}(j) &:= \text{in1}(j),\\
\text{out2}(j) &:= \text{if valid1(OUT, } j) \text{ then out1}(j) \text{ else in1}(j),\\
\text{chute2}(j) &:= \text{if valid1(CHUTE, } j) \text{ then chute1}(j) \text{ else in1}(j),\\
\text{valid2(IN, } j) &:= \text{valid1(IN, } j) \wedge \text{valid1(OUT, } j) \wedge\\
&\qquad (\neg\text{chute\_transfer1}(j) \vee \text{valid1(CHUTE, } j)),\\
\text{valid2(OUT, } j) &:= \text{valid1(IN, } j) \vee \text{valid1(OUT, } j),\\
\text{valid2(CHUTE, } j) &:= (\text{valid1(IN, } j) \wedge \neg\text{chute\_transfer1}(j))\\
&\qquad \vee \text{valid1(CHUTE, } j).
\end{aligned}
$$

FIGURE 9 Combining queue transitions for slot j.

again stay in place until the OUT row starts moving. (See [1] for more details.) The CHH, CHR and CHL signals shown in Figure 8 control the movement of the chute_transfer signal between slices of the queue.

5.2 Combining ALU

Inputs to the ALU come from the second slice of the OUT and CHUTE rows of the combining queue. The next stage output (FP_OUT in Figure 6) gets address packets from the OUT row, passing through the ALU unchanged; data packets on FP_OUT are the output of the ALU, which simply selects the OUT row for messages that did not combine. The wait buffer output (WB) gets message identifiers from both rows for the address packet; data packets on WB come from the OUT row without change.

The fetch-and-store operation can be considered a fetch-and-ϕ operation where ϕ is the projection onto the second operand ($\pi_2(x, y) = y$), since the last store made is the value retained in memory. Thus $\pi_2(data_{OUT}, data_{CHUTE1}) = data_{CHUTE}$ is the correct value to send to memory in order for the operations to be serialized as M_{OUT} before M_{CHUTE}. Although it would require only three functions in the ALU on the forward path if $data_{OUT}$ were sent to memory instead of $data_{CHUTE}$, in that case $data_{CHUTE}$ would need to be sent to the wait buffer. Furthermore, the address packet for the combined message would have to come from M_{CHUTE} in order for the response from memory to pass through the return path stage without modification.

Our four-function ALU has binary carry look-ahead implemented with Domino CMOS precharged logic. Pre-charged gates on phase 1 compute propagate and generate signals. The signals func0 and func1 represent the op codes as shown in Table II:

For each bit, we compute the generate and propagate signals from func0, func1, the CHUTE bit, and the OUT bit using the equations

$$\overline{G} = \text{func0} \wedge \text{func1} \wedge \overline{\text{OUT}} \wedge \overline{\text{CHUTE}},$$

$$P = (\text{CHUTE} \wedge (\text{func0} \vee (\overline{\text{func1}} \wedge \overline{\text{OUT}}))) \vee (\text{OUT} \wedge (\text{func1} \vee (\overline{\text{func0}} \wedge \overline{\text{CHUTE}}))).$$

TABLE II
Function Signals Used to Indicate Op Codes in ALU

func0	func1	Operation
0	0	ADD
0	1	SELECT Out
1	0	SELECT Chute
1	1	OR

During phase 1, the propagate signals, which are inputs to phase 2 dynamic gates at all levels of the carry tree, are set up. At each level, the generate signal to the next level $GG = G_1 \vee (G_0 \wedge P_1)$, and the carry signals $C_0 = CC$ and $C_1 = G_0 \vee (CC \wedge P_0)$ are computed by Domino CMOS gates on phase 2. Multi-output gates are used to shorten the carry chain. The design is similar to that in [11]. All of the logic is done double rail; the final stage is a static exclusive-or of the carry and propagate signals.

6 RETURN PATH COMPONENT DESIGN

The return path component (RPC) in the MOSIS packaging contains a wait buffer (associative memory array that contains the data from a pair of combined messages), an ALU, and two non-combining queues (Figure 10).

Decombining messages requires an associative matching. For good performance, this matching must be done in parallel with insertion of arriving messages into the output queue of the RPC. Data from the OUT row always goes to the wait buffer and the message ID from the OUT row always goes on toward memory. With correct decombining, this has the same effect as if the message from the OUT row arrived at the memory first, with the message from the CHUTE row following immediately. Thus the response from memory does not need to wait to find out whether it had combined before entering the Main Queue in Figure 10, since the value from memory is the correct response to the message originally from the OUT row. A delay register at the RP_IN input to the ALU holds a copy of this message until a match can be detected in the wait buffer, in case the message happens to be a response to a combined message.

A wait buffer is associated with each input/output pair, but the wait buffer input buses associated with a forward path output port can be tied together since they are driven only when the associated output port is driven, and thus will never both be driven at the same time. A message received on RP_IN starting at cycle t is sent to the main queue and also to the wait buffer where its address packet is simultaneously compared with all the messages currently in the wait buffer. If a match is found, the wait buffer asserts its *match* line during cycle $t + 1$. The output of the ALU is sent to the decombined queue at cycle $t + 2$ so that the two queues receive the first packets of their messages at cycles of the same parity.

The WB input to the RPC begins and ends mes-

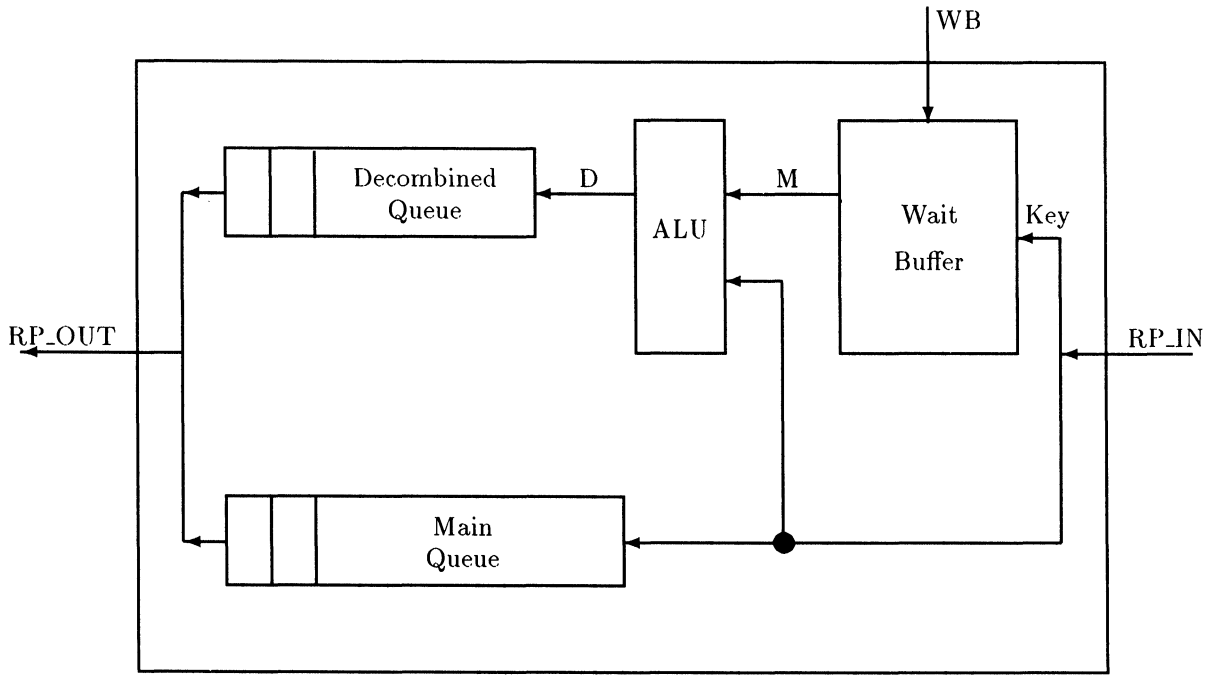


FIGURE 10 Block diagram of a return path component.

sages on a cycle of the same parity as the FP_OUT output of the paired FPC. The input D to the Decombined Queue must have the same cycle parity as the RP_IN input, since the Main and Decombined Queues share an output. As a result, the detailed design of the RPC must agree with the parity of the memory latency. As described in the next section, the internal wait buffer structure determines whether the difference between the cycle a message enters the wait buffer and the cycle it exits is even or odd. In the current implementation, the total number of cycles between the time the first packet of a message begins transmission on WB and the time it appears on D is even. Thus the number of cycles between the time a message begins transmission on FP_OUT and the time it begins transmission on RP_IN must be even as well. The FPCs and RPCs in the same switch must have the same cycle parity for their queues, and the memory must have a delay which is an even number of switch stages.

6.1 Wait Buffer

The wait buffer is an associative memory that stores information sent by the FPC when combining two fetch-and-ϕs into a single request. The wait buffer inspects all responses from MMs and searches for a response to a request previously combined by the FPC. When it finds a response to such a request, it generates a second response and deletes the request from its memory. Packets are stored in return path format, with the PE/MM address field from the WB port reversed to be a MM/PE address. The structure of a wait buffer (WB) is shown in Figure 11.

A typical message slot is shown in Figure 12 and consists of two registers (called Areg and Breg), compare logic, and a controller. Each register contains the data bits and a data valid (DV) bit. The registers are connected in a loop of length two, and shift at each cycle. This structure requires each message sent to the wait buffer to consist of a single address packet followed by a single data packet. Similar structures support messages containing a *fixed* even number of packets. In the combinable operations we support, no message requires more than one data packet to be stored in the wait buffer. For the fetch-and-ϕ operations, the data packet from the OUT row in the forward path is stored. For the store double operation, no value is returned, and the data packet may be set to any function of the ALU that is convenient. For the load double operation, the three data packets from the RP_IN input are selected by the ALU and placed in the decombined message; no additional values need to be stored in the wait buffer.

Each slot connects to the following buses:

- The write bus (Wbus) is used to send data to the wait buffer from the FPC and connects to a wait buffer input port.
- The read bus (Rbus) is used by each slot for

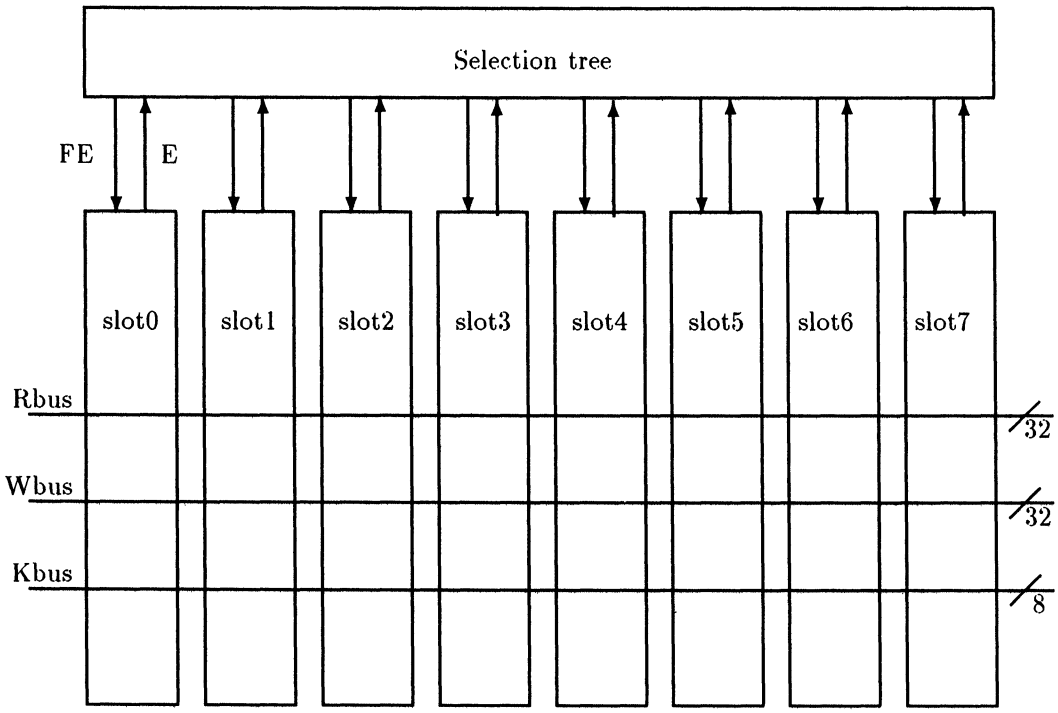


FIGURE 11 Block diagram of a wait buffer.

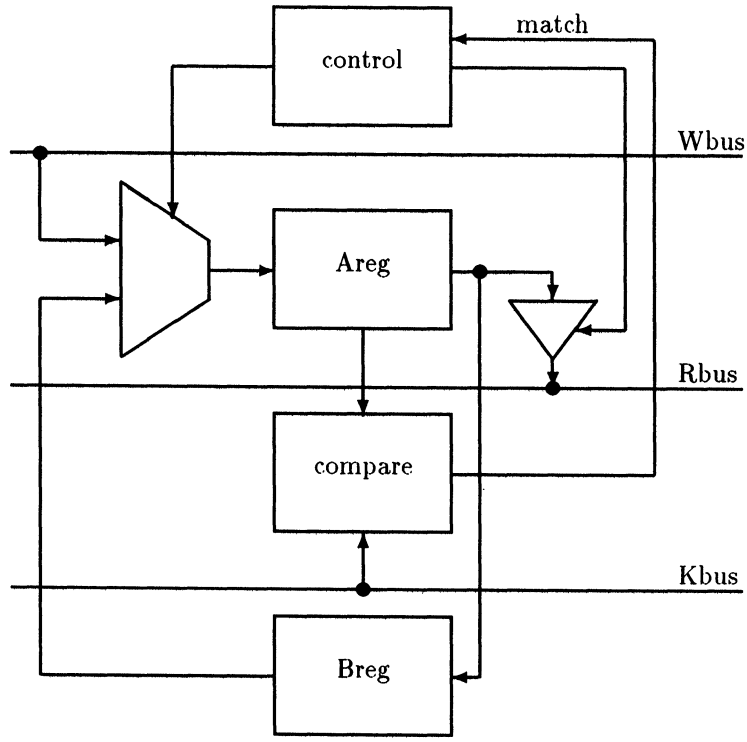


FIGURE 12 Slot of a wait buffer holding a two-packet message.

transmission of its message out of the wait buffer.

- The key bus (Kbus) contains the search key received from RP_IN.

The first-empty (FE) lines are computed in a simple tree structure from the empty (E) lines for each slot. The empty lines are ORed and the result is transmitted up the tree, with an OR performed at every node. The FE value for the parent of each subtree is set to true if there are no empty slots in any subtree to its left (ordering the slots from left to right). FE is true not only for the first empty leaf slot, but for all non-empty slots of lower index. This causes no problem in operation: FE causes the slot to receive a new message when it is asserted, but only if the slot is not already valid.

A schematic of a wait buffer cell is shown in Figure 13. HP ("Hold Packet") is asserted if the cell already has a message, otherwise AP ("Advance Packet") is asserted. The comparison logic to compute the match signal is not shown, but is similar to the short Domino CMOS chain in the combining queue cell, except that a larger dynamic XOR takes the inputs from two bits to produce the comparison signal for the pull-down transistor. CanRead is computed from the match signal at the cycle of appropriate parity, so that a read always begins with the first packet of the message; the precharged $\overline{R}$ signal is shared among all slots of a wait buffer.

The wait buffer is designed under the assumption that only one message present in the wait buffer at a given stage can match a given response from memory. For regular messages, the combination of rotated PE/MM address and ORI number used for the match forms a unique key at a given stage. Since our processor never issues two messages with the same ORI, no false matches can occur. However, if broadcast messages are propagated through the return path by sending the message out on both ports, the PE/MM address and ORI combination can inadvertently match a message in a wait buffer. To avoid this, the valid bit on the key input to the wait buffer is lowered when a broadcast message appears on RP_IN. Thus the wait buffer never sees a broadcast message.

Since the input to the NOR that pulls down $\overline{R}$ (see Figure 13) comes from the "a2" section of the recirculating loop, the head packet of a message will always be read out of the wait buffer an odd number of cycles after it was read in. An additional cycle is required to pass through the ALU and enter the queue, giving the total of an even number of cycles from WB to D in Figure 10, as mentioned previously. The input to the NOR could equally well have come from the "b2", which would have changed the parity of the cycle total and thus changed the parity of the memory latency, measured in switch cycles.

6.2 Decombining ALU

A four-function ALU identical in design to the combining ALU described in section 5.2 is used to gen-

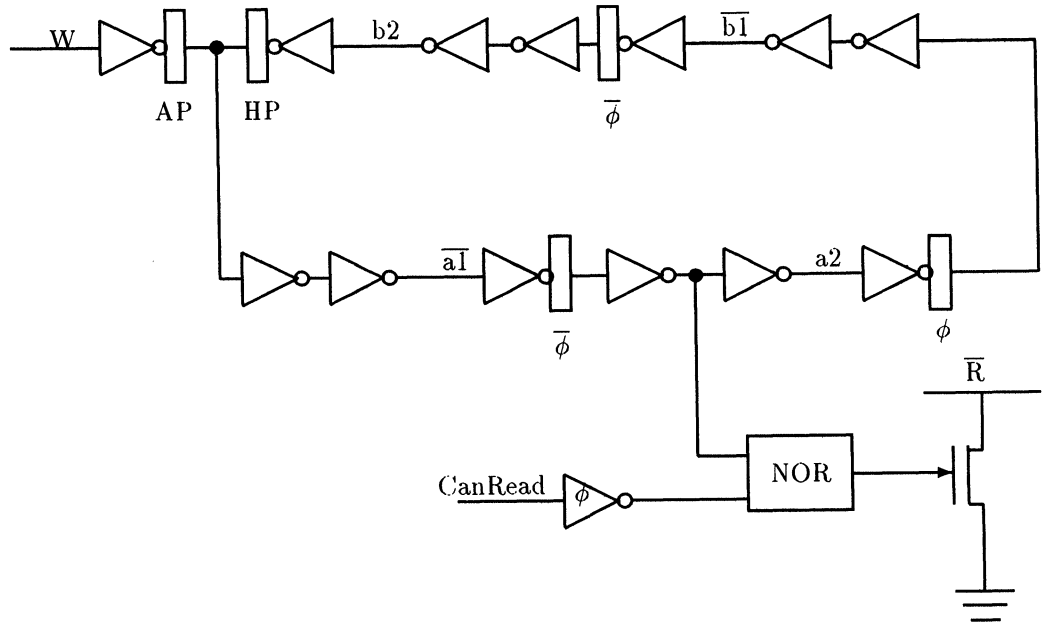


FIGURE 13 Schematic of a wait buffer cell.

erate the second response to a fetch-and-ϕ operation by operating on the data packet received from a wait buffer slot and the data packet received from RP_IN. It passes address packets unchanged, except for replacing the op code with the one from RP_IN. The packet from RP_IN is delayed one cycle at the input to the ALU. The output of the ALU goes to the decombined queue on the cycle after that.

Unlike the forward path, the control signals for the ALU are a function not just of the op codes but of the SWD (store was done) bit as well. For the conditional fetch-and-store operations, if the store was done at memory, then any messages waiting to decombine should return the value stored from the wait buffer, which was stored in the memory by the combined message. If the store was not done, then the value in the message that arrived on RP_IN (representing the value in memory before this conditional fetch-and-store was attempted) should be returned by the decombined message as well. Furthermore, when a decombine occurs, the SWD bit in the message sent to the decombined queue must always be deasserted, since the decombined message always represents an unsuccessful attempt to store its own value.

6.3 Non-Combining Systolic Queues

The decombined queue and the main queue are multiplexed with the queues from the paired RPC for control of the output port. Each queue is a simple non-combining systolic queue, like those described in section 2.

The logic to determine the RP_IN.DA signal is quite complex. It can be asserted only if there is at least one empty slot in each of the two queues. In addition, the decombined queue must leave room for a message coming out of the wait buffer but not yet added to the queue. Simulation of the component in a network environment was required to get the logic correct.

7 CONCLUSION

We have presented the design for two components used in a processor-to-memory interconnection network for a shared memory system. These components allow the combining of requests that are destined to the same memory location. The design contains both semi-systolic queues and an associative "wait buffer."

In our presentation, we have carefully distinguished between those pieces required for combining and those pieces that would be required in any buffered node. By doing so, we have indicated the precise cost of hardware combining, which is much smaller than that previously estimated. Specifically, the major cost of combining in small pin-count packages is the need for extra pins connecting the FPC to the RPC. In a packaging technology allowing the fabrication of an entire switch in a single package, this cost of combining is zero. In [2] we discuss this in greater detail and also discuss the logic costs of combining, which we estimate as about 40% in the FPC and 60% in the RPC.

Both the FPC and the RPC have been fabricated in 2μ CMOS with 132 pin packages using the MOSIS service. These parts were designed using the Magic design tools from UCB. Simulations were performed at multiple levels. We constructed both a behavioral model of the switch and a detailed RTL-level model of our design, both written in C. These were linked to each other and to a switch-level simulator that operated on the circuit extracted from the layout. The verification process used confirmed that all three levels of abstraction agreed. We were able to test the agreement of the behavioral and RTL-level models on simulations of the entire 16-PE system and verified the conformance of the RTL-level model with the layout while simulating smaller systems.

To aid the layout verification process, schematics of each cell were entered into a commercial PC-based schematic capture system. We wrote conversion software to produce the same format transistor list files that are produced from our layout. Thus we were able to validate the layout of a cell against its schematic with a simple ASCII file comparison program.

The FPC has 29,000 transistors, exclusive of pads, while the RPC has 50,000 transistors. Both parts run solidly at 10 MHz, the upper limit of performance that can be measured in the test rig. A 4×4 combining switch board has been in use in a 4 PE prototype since November of 1992, and functions correctly at all speeds at which the memory and processors work reliably (up to 15 MHz). At this writing, we are completing the construction of a 16-PE machine using these components and plan to undertake a 2-year effort to measure the characteristics of the resulting system.

References

[1] Susan R. Dickey. *Systolic Combining Switch Designs*. Ph.D. Dissertation, New York University, Department of Computer Science, 1994.

[2] Susan R. Dickey and Richard Kenner. Hardware combining

and scalability. *4th Annual ACM Symposium on Parallel Algorithms and Architectures,* pages 296–305, June 1992.

[3] Susan R. Dickey and Richard Kenner. Using qualified clocks in the NORA clocking methodology to implement a systolic queue design. *Proceedings of the Brown/MIT Conference on Advanced Research in VLSI,* pages 165–179, March 1992.

[4] Susan R. Dickey and Yue-sheng Liu. Simulation and analysis of enhanced switch architectures for interconnection networks in massively parallel shared memory machines. *Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computation,* pages 487–490, October 1988.

[5] Susan R. Dickey and Ora E. Percus. Performance differences among combining switch architectures. *Proceedings of the 21st International Conference on Parallel Processing,* pages 110–117, August 1992.

[6] Jan Edler et al. Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach. *Proceedings of the 12th Annual International Symposium on Computer Architecture,* pages 126–135, June 1985.

[7] Nelson F. Goncalves and Hugo J. DeMan. NORA: A racefree dynamic CMOS technique for pipelined logic structures. IEEE Journal of Solid-State Circuits, SC-18(3): 261–266, June 1983.

[8] Allan Gottlieb. An overview of the Ultracomputer project. In J. J. Dongarra, editor, *Experimental Parallel Computing Architectures,* pages 25–95, New York, 1987. Halstead Press.

[9] Allan Gottlieb and Clyde P. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News,* pages 16–24, October 1981.

[10] Leo J. Guibas and Frank M. Liang. Systolic stacks, queues and counters. *MIT Conference on Advanced Research in VLSI,* pages 155–164, 1982.

[11] Inseok S. Hwang and Aaron L. Fisher. Ultrafast compact 32-bit CMOS adders in multiple output Domino logic. *IEEE Journal of Solid-State Circuits,* 24(2):358–369, April 1989.

[12] R. H. Krambeck, C. M. Lee, and H. S. Law. High-speed compact circuits with CMOS. *IEEE Journal of Solid-State Circuits,* SC-17(3):614–619, June 1982.

[13] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers,* C-24:1145–1155, December 1975.

[14] Charles E. Leiserson. *Area Efficient VLSI Computation.* MIT Press, Cambridge, Massachusetts, 1983.

[15] Yue-sheng Liu. *Architecture and Performance of Processor-Memory Interconnection Networks for MIMD Shared Memory Parallel Processing Systems. Ph.D. Dissertation, New York University, Department of Computer Science, September 1990.*

[16] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Massachusetts, 1980.

[17] Ora E. Percus and Susan R. Dickey. Performance analysis of clock-regulated queues with output multiplexing in three different 2 by 2 crossbar switch architectures. *Journal of Parallel and Distributed Computing,* 16(1):27–40, September 1992.

[18] Gregory F. Pfister and Allan Norton. 'Hot spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers,* C-34(10):943–948, 1985.

[19] Masakazu Shoji. *CMOS Digital Circuit Technology.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[20] Marc Snir and Jon Solworth. The Ultraswitch—a VLSI network node for parallel processing. Ultracomputer Note #39, New York University, January 1984.

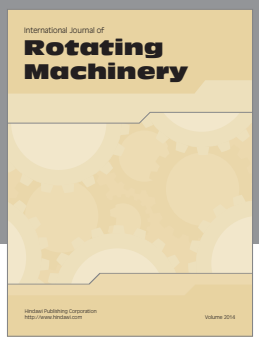
[21] Shiwei Wang, Yarsun Hsu, and C. J. Tan. A novel message switch for highly parallel systems. *International Conference on Computer Design,* pages 150–155, 1989.

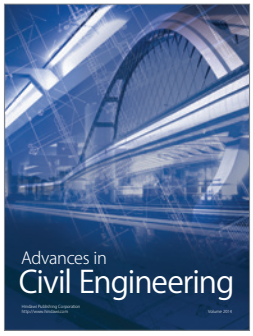
[22] Monica C. Wong. A combining Omega network: Performance vs. implementation. Technical Report RC 11977, IBM Research Report, June 1986.

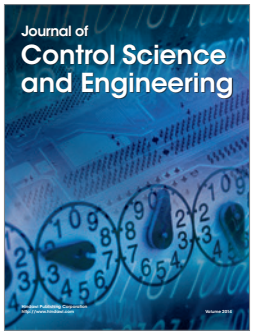
Biographies

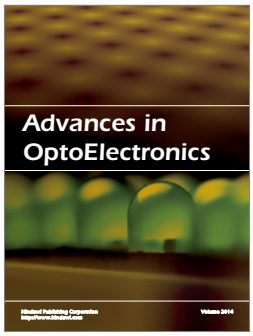
SUSAN DICKEY received a M.S. in Computer Science from New York University in 1984 and received her Ph.D. from NYU in May 1994. She is an Associate Research Scientist with the Ultracomputer Research Laboratory at the Courant Institute of Mathematical Sciences, NYU. Her research interests are VLSI design, interconnection networks and parallel architectures. She is a member of the IEEE Computer Society.

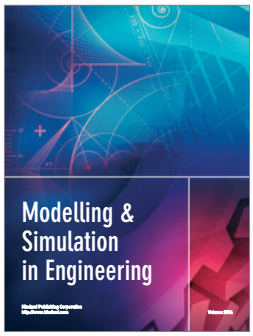
RICHARD KENNER received an M.S. in Computer Science from New York University in 1981. He is an Associate Research Scientist with the Ultracomputer Research Laboratory at the Courant Institute of Mathematical Sciences, NYU. His research interests include architecture, VLSI design, compilers and operating systems. At present, he is responsible for development and maintenance of the GNU C compiler (GCC) and is part of the team developing GNU Ada 9X.

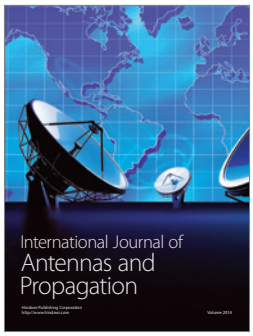


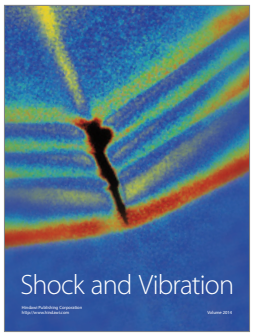


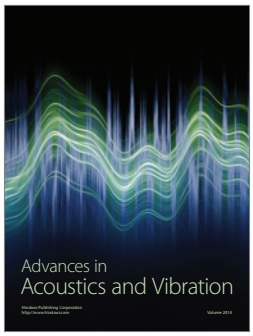












Hindawi

Submit your manuscripts at
http://www.hindawi.com