A Combining Switch for the NYU Ultracomputer

Susan R. Dickey
Richard Kenner [1]
Courant Institute of Mathematical Sciences
New York University

June 25, 1992

[1]Both authors are supported by U.S. Department of Energy grant number DE-FG02-88ER25052.

Abstract

This report describes the combining switch that we have implemented for use in the 16×16 processor/memory interconnection network of the NYU Ultracomputer prototype. Packaging, message types and message formats are described. Details are given about the internal logic of the two component types used in the network, including the systolic combining queue, the wait buffer and the ALU. Systolic combining queue designs with greater combining capability are sketched.

1 Combining switch architecture

A combining switch has been fabricated for use in the Ω network of the NYU Ultracomputer prototype [3]. A 2×2 switch node is composed of four each of two types of custom VLSI chips: *forward path* and *return path* components (Figure 1). Control for the switch is distributed as tri-state selection logic in each chip. The routing information is included in the message. Flow control avoids the necessity of acknowledgement 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.

1.1 Packet format

Each message consists of an address packet and from zero to two data packets. Messages with zero or two data packets are padded with a null packet to be of even length. Packet formats are shown in Figure 2. The numbers in parentheses after the field names are the number of bits in the field.

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, the high order bit is replaced with the address of the input port where the message arrived (0 or 1) and the PE/MM address is rotated left. At the memory this field is reversed and used as the MM/PE address field on the return path.

The ORI (outstanding request index) 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. Thus only 16 bits need to be sent to the RPC to identify the message waiting to be decombined and the message that has been sent on to memory.

The first and second fields in the address packet sent to the wait buffer identify the message that is waiting to be decombined; the third and fourth fields are to be matched with the messages returning from memory. There are 16 extra bits, since the packet size is limited by the 32-bit data width. On the return path as well, the packet size is limited by the data width, since the memory address is not returned to the PE.

The SWD bit is returned to indicate whether or not a store was actually

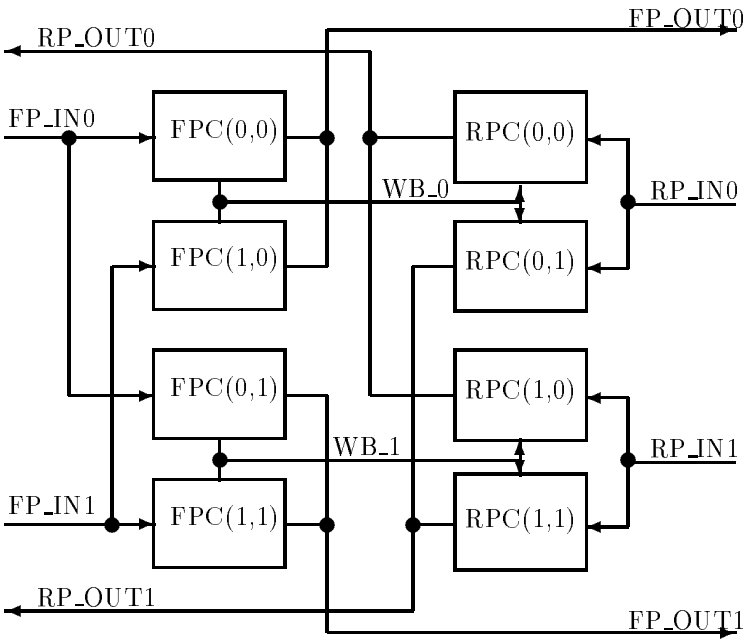

Figure 1: Block diagram of combining switch showing connections of forward path components (FPC) and return path components (RPC).

Forward path address packet

PE/MM address (4)	op code (4)	memory address (23)	ORI (4)

Forward path data packet

Extra (3)	Data (32)

Wait buffer address packet

PE/MM address0 (4)	ORI0 (4)	PE/MM address1 (4)	ORI1 (4)

Return path address packet

MM/PE address (4)	op code (4)	ORI (4)	SWD (1)

Return path and wait buffer data packet

Data (32)

Figure 2: Packet formats for the interconnection network in a 16×16 NYU Ultracomputer prototype.

done for the conditional Fetch and Store operations.

1.2 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 [4]. See Table 1 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-add of 0;

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

Table 1: ALU operations for the memory requests implemented in the combining switch

combinable single word stores are simply fetch-and-store ignoring the returned value. 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 [2] for the use of reflect in the Ultracomputer architecture). Only broadcast requires any special processing in the network. Broadcast is done from a memory location to all processors, after a broadcast request has been received from a processor. A message can be inserted in parallel into both queues connected to an input when the broadcast op code is decoded.

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.

1.3 Flow control logic

As described in Section 4, the construction of the queues requires that there be an even number of packets per message and that switches distinguish even and odd cycles. A cycle is defined to be "even" for a given switch if the parity of the cycle is the same as the parity of the stage to which the switch belongs,

so that cycles which are even for a switch are odd for its predecessors and successors. At initialization, the parity of the cycle is the same as the parity of the stage to which the switch belongs, so that cycles that are even for a switch are odd for its predecessors and successors while the cycle parity of the FPC and RPC in the same switch are identical. Reception of messages starts only at even cycles while transmission of messages starts only at odd cycles.

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. The two protocol 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. Since each sender sends to two receiving queues or waitbuffers, DA must be asserted only when both can accept data. In the partitioning of Figure 1, this requires an external AND gate for each paired input. More complicated protocols could consider DA signals from both receivers; see [9].

Since these control signals are ignored during cycles when a message transfer cannot be started, they can be set ahead of time to overlap data transfer and flow control operations. These signals are precomputed and extra space is left in the queue to keep communication with the next stage off the critical path.

Note that this is not strictly speaking a handshaking protocol: DA is not an answer to DV, nor an acknowledgment, 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.

1.4 Packaging

Both the FPC and the RPC have been fabricated in 132 pin packages using the MOSIS service. Both parts run solidly at 10 MHz, the upper limit of performance which can be measured in the test rig. A single-board, 2-input/2-output non-combining network composed of eight of the forward path components with their "don't combine" signal held asserted has been used in a 2-PE/2-MM system for over a year, functioning correctly at all speeds at which the memory and processors work reliably (up to 15MHz). A 4×4 combining switch board is under test at 12.5 MHz.

	Transistors 10^3	Area $10^6\lambda^2$	Pins				
			Input	Output	Tri-state	Clocks	Total
8 chips per switch							
FPC	29	18	43	2	74	2	121
RPC	50	32	70	4	35	2	111
4 chips per switch							
FPC	58	36	82	76	0	2	160
RPC	100	64	134	38	0	2	174
2 chips per switch							
FPC	116	72	84	150	0	2	236
RPC	200	128	136	74	0	2	210
1 chip per switch	316	200	152	150	0	2	304

Table 2: Signal pin count, area and transistors per chip for 2×2 combining switches to be used in a 256-PE system with a 4 gigabyte address space

With higher pin count packages, a switch can be implemented with fewer chips; the next level of integration is achieved by packaging components which share output ports together. Pin counts, area estimates and transistor counts for these packaging alternatives are shown in Table 2. We plan to fabricate the 4 chip per switch alternative in 208 pin packages from NCR this year.

The area estimates include routing for internal buses, but do not include pads. The transistor count is slightly overestimated for the higher levels of integration because some of the control logic is duplicated in each part. The switch design is pin-limited rather than area-limited. Even with all the logic on a single chip, the area for internal logic would only be approximately 6 mm $\times$ 6 mm with 0.8 micron feature size.

2 Systolic queue design of Guibas and Liang

The systolic queue design we have implemented is essentially the folded shift register shown in Figure 2. Items enter the IN row at the left and move right until the slot underneath, in the OUT row, is empty. As items are deleted from the front of the queue, items in the OUT row move to the left.

Guibas and Liang's design [5] is fully systolic, according to Leiserson's definition [8], requiring no global signals except a clock. In their design, as items are inserted into the IN row, they move right whenever the slot to

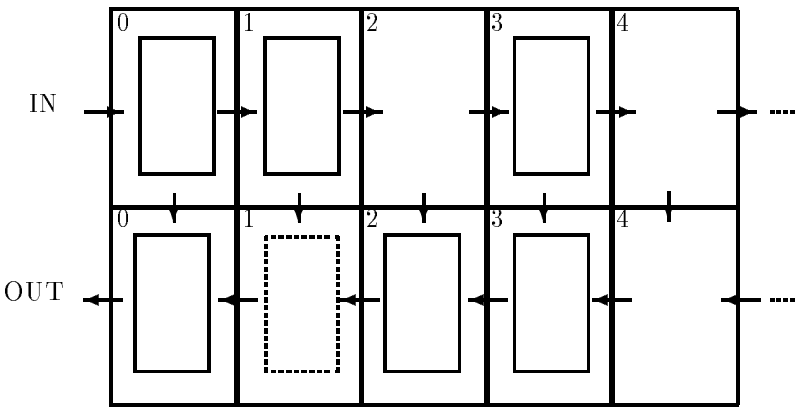

Figure 3: A systolic queue design.

the right is empty and the slot below in the OUT row is occupied. As an item is deleted from the front of the OUT row, the cell the item has vacated is left "indisposed" instead of empty. Any item immediately to the right of an indisposed cell in the OUT row senses that it may move left on the next cycle, but an item in the IN row directly above will not move down. In general the OUT row will consist of a string of occupied and indisposed cells, terminated by a string of empty cells to the right. An item can be deleted or inserted every second cycle.

There are practical difficulties with using this scheme for the queues in an interconnection network switch.

1. To keep bandwidth up and latency down, the insertion rate should be limited by the off-chip delays, not by the characteristics of the internal switch logic. Not allowing insertions and deletions on consecutive clock cycles is a serious practical 3defect of this design.

2. The rules governing the movement of items, which are based on the assumption of unlimited size, are different for the rightmost cells in the IN and OUT rows.

3. Since the OUT row has three states, the representations of its state transitions can be somewhat complicated.

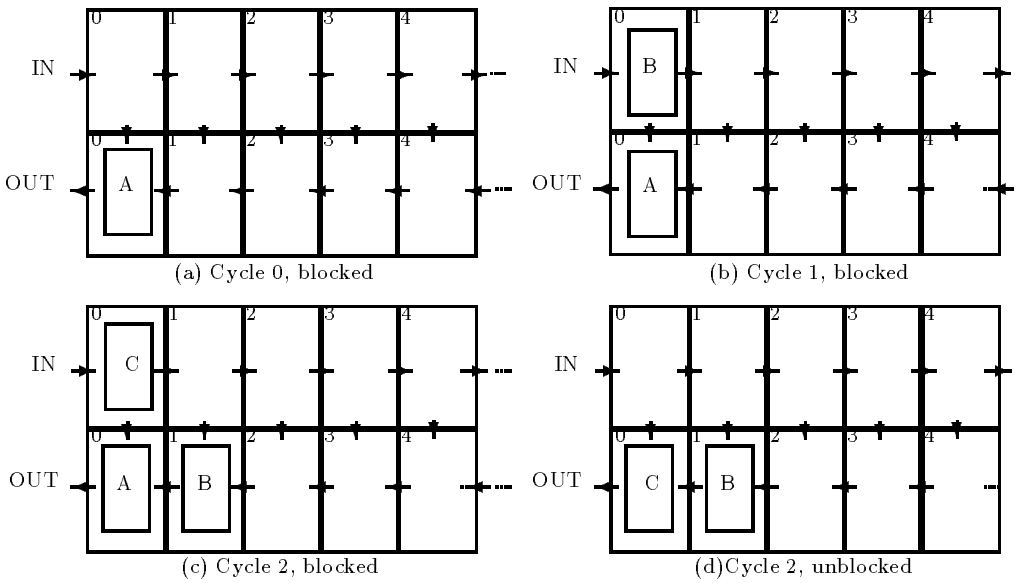

Figure 4: A queue must unblock an odd number of cycles after the first item was inserted.

3 Snir and Solworth's basic queue design

Snir and Solworth [11] developed a design that distributes global information about whether the queue is blocked to each cell in order to be able to delete an item every cycle. If the queue is not blocked, a deletion will always occur and all items in the OUT row simultaneously move left. The global information allows all the items to move in lock-step into the neighboring location, even though it was not empty on the preceding cycle, thus 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. This information 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.

To describe the queue's operation more precisely, using a standard two-phase clocking scheme like that in Mead and Conway[10], 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 at the end of phase 2. Similarly, out1(j) and out2(j)

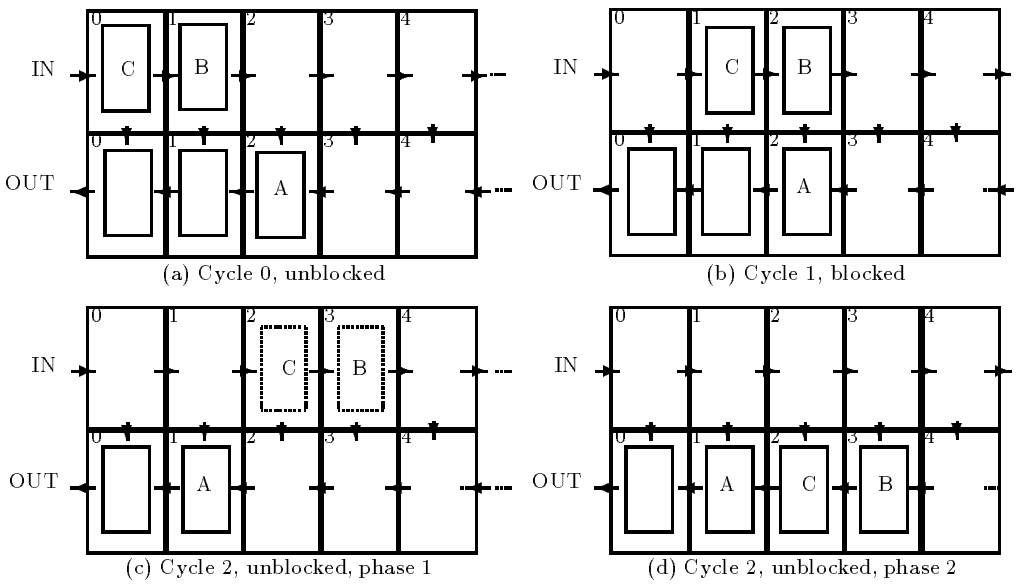

Figure 5: Out of order items due to blocking a non-empty queue for an odd number of cycles.

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 Snir and Solworth design when the queue is neither blocked nor full are

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

and the phase 2 transitions are

$$\begin{aligned} \text{in2}(j) &:= \text{in1}(j), \\ \text{out2}(j) &:= \text{ if valid1}(\text{OUT}, j) \end{aligned}$$

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

Snir and Solworth claim that the following invariants ensure 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).

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. [1].

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

1. The number of cycles with no insertions between two consecutive insertions must be even.

2. When the queue is blocked, it must be for an even number of cycles.

These constraints are necessary to prevent items from getting out of order. Consider what happens when the queue is blocked for the first time. As long as the queue is not blocked, at the end of phase 2 a single item will be available for output as out2[0]. When the queue is blocked, this item will remain in place and the OUT row will keep moving. Let cycle 0 be the cycle that the item at the front of a queue that has blocked for the first time was inserted. As illustrated in Figure 3, if a stream of consecutive insertions is fed to the queue while it remains blocked, the rightmost items will be lined up vertically at the end of each odd cycle. If the queue is then unblocked during the next even cycle, the items in the OUT row will move left during phase 1 while the items in the IN row move right, and during phase 2 two items will slide down into the OUT row in the wrong order. Thus correct operation requires that a queue stay blocked for an odd number of cycles after the first insertion.

Figure 3 shows the effect of blocking a non-empty queue. At cycle 0, the queue is in a legal state, with both rows moving and an odd difference

[1] The numbering convention here is different than that in [11], so this difference is odd, rather than even

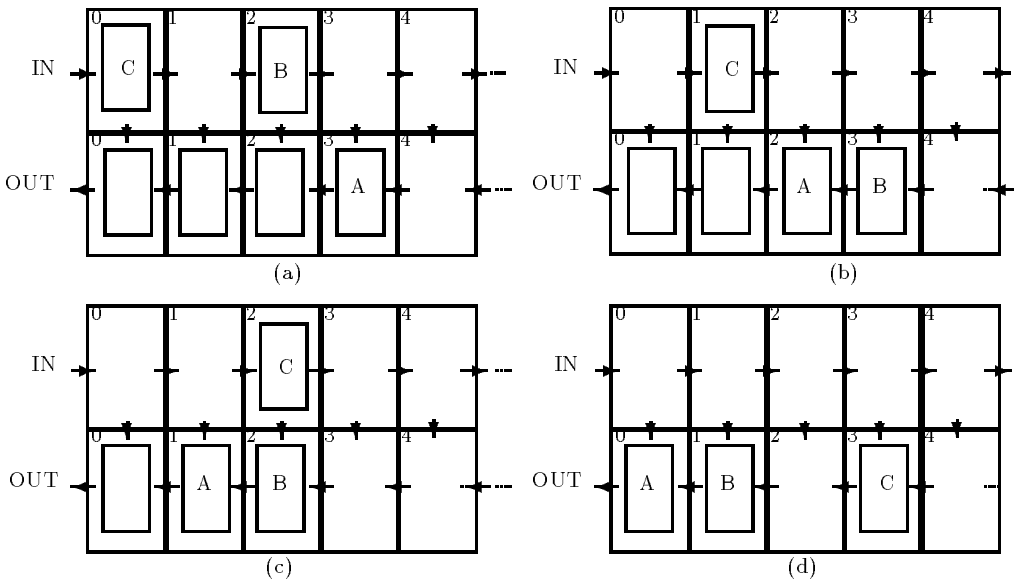

Figure 6: A hole in the OUT row due to an odd number of cycles between insertions.

between the highest indices of the IN and OUT rows. At cycle 1, the queue is blocked for one cycle; the IN row keeps moving but the OUT row is stationary. At cycle 2, both columns start moving, resulting in items B and C being placed out of order in the OUT row. Note that if C had not been inserted at cycle 0 there would have been a hole in the OUT row at cycle 2 instead, which might have taken some later item out of order.

The input stream need not be a continuous stream of consecutive insertions for correct operation, but, as illustrated in Figure 3, there cannot be an odd number of cycles between insertions. In the figure, there is only one cycle between B and C; this results in a hole between B and C in the OUT row.

In the Snir and Solworth description of their design, a queue is considered to be in one of three states: normal, blocked and full. They assume that a full queue is blocked as well. In implementing such a queue, the full signal can be derived from valid bit of the rightmost slot in the OUT row which can be occupied and still leave room for the rest of the message. There is a fourth state, which we will call emptying, after the queue has unblocked but before the IN row can begin moving again. The IN row can begin moving only when there is enough space to allow a complete message to be accepted, even if the queue becomes blocked again.

4 A systolic queue with two global control signals

As discussed in the previous sections, holes or out of order messages can appear in the OUT row only if at some cycle n both rows are "lined up" at the rightmost edge and at the following cycle $n+1$ 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 will be a packet of a message. In this section we show that 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 the previous section, ensuring correct operation.

The two global control signals are in_moving and out_moving. The phase 1 transitions of the queue are:

$$\begin{aligned} \mathrm{in1}(j) \quad := \quad & \text{if in_moving} \\ & \text{then } \mathrm{in2}(j-1), \end{aligned}$$

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

and the phase 2 transitions are

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

Consider that the index $j = -1$ corresponds to the values of the input port.

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 an even cycle.
4. The control signal out_moving can change its value only before 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. Under the above constraints, the following can be proved:

Theorem 4.1 *If the queue is not empty and, at the end of cycle n, the highest index of an occupied cell in the IN row is equal to the highest index of an occupied cell in the OUT row, then in_moving and out_moving cannot both be true during cycle $n+1$.*

Proof:

If the queue is empty, the theorem is trivially satisfied. Let cycle 0 be the first cycle that a message enters an empty queue; in_moving must be true to allow a message to enter, and the first packet may leave the queue at cycle 1 if out_moving is true. At the end of the first cycle the theorem is satisfied, since the premise is false, as OUT(0) is the only occupied cell. Assume inductively that the theorem is true for every cycle through $n-1$, and thus at cycle n there are no holes in the OUT row. Assume without loss of generality that the queue is not empty at any cycle from 0 to $n-1$.

At cycle n, for n odd, consider separately the cases when out_moving is true and when it is false.

1. out_moving is false: Since out_moving can only change at the beginning of odd cycles, it will still be false the next cycle, which is even.

2. out_moving is true: Some even number $2j$ of packets must have entered the queue since cycle 0, because there have been an even number of cycles and messages have an even number of packets and can be initiated only at even cycles. Some odd number $2k+1$ of packets must have exited, because out_moving, since it is true now, was false for an even number of cycles from the time packets could have begun to exit at cycle 1. So $2j-2k-1$ packets (an odd number) must remain in the queue. Let m is the highest index of an occupied cell in the OUT row; there are m packets in the OUT row, since it has no holes. Suppose m were also the highest index of an occupied cell in the IN row. The number of unoccupied cells to the left of cell m must be $2l$ for some l, because messages are of even length and initiated only at even cycles. So there would be $m+m-2l$ or an even number of packets in the queue. But the number of packets must be odd, so m is not the highest index of an occupied cell in the IN row.

At cycle n for n even, consider separately the cases when in_moving is true and when it is false.

1. in_moving is false. Since in_moving can only change at the beginning of even cycles, it will still be false the next cycle, which is odd.

2. in_moving is true. An even number $2k$ of packets has exited the queue since the the first packet could exit on cycle 1, as the queue has been blocked for an even number of cycles in that time. At the end of cycle n, in_moving will have been true for an odd number of cycles, so either an even number $2j$ of packets has entered and cell 0 of the IN row is not occupied, or an odd number $2j + 1$ has entered and cell 0 of the IN row is occupied (because the queue is in the middle of receiving a message). This creates two subcases :

   (a) There are an even number of packets in the queue, since an even number have entered and an even number have exited, and cell 0 of the IN row is not occupied. Let m be the number of packets in the OUT row. There must be some even number $2l$, $l < m$ of unoccupied cells in the IN row with index greater than 0 and less than the highest occupied index in the IN row, so there would be $2m - 2l - 1$ packets in the queue if the highest occupied index were m, which contradicts the assumption that the number of packets in the queue is even.

   (b) There are an odd number of packets in the queue, since an odd number have entered and an even number have exited, and cell 0 in the queue is occupied. Again, there must be some even number $2l$, $l < m$ of unoccupied cells in the IN row with index greater than 0 and less than the highest occupied index in the IN row, so there would be $2m - 2l$ packets in the queue if the highest occupied index were m, which contradicts the assumption that the number of packets in the queue is odd.

Corollary 4.2 *The OUT row in a systolic queue operating according to the constraints given on cycle parity of message initiation, message length, in_moving and out_moving will not have holes or out of order packets.*

We implement the parity constraints by tying changes in in_moving and out_moving and the related input and output DA (data accept) signals (see section **??**) to the end of a message on the input and output ports. The input DA must be lowered in time to prevent the queue from filling, but will not be noticed by the preceding stage until the end of the message being sent. This means that in the worst case there must be room for the maximum number of packets in a message in the OUT row when the input DA is lowered.

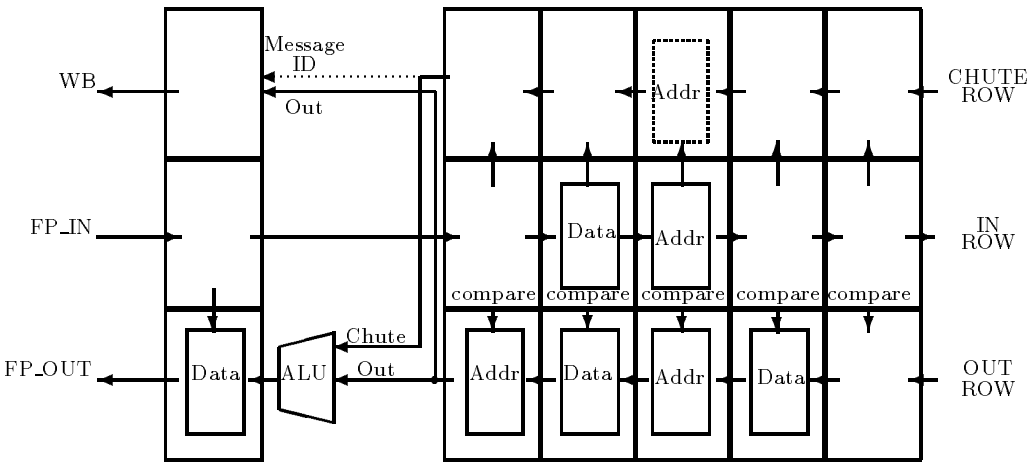

Figure 7: Design of systolic combining queue.

5 Forward path component design

The forward path component is essentially a single systolic combining queue, with the ALU embedded in the first packet location of the queue, rather than on the critical off-chip path, so that message transmission is not delayed by the combining logic. Separate control logic decodes the op code to determine the ALU operation and the length of the message, forms the AND of the WB.DA and FP_OUT.DA and checks the NE signal from the paired queue to determine if the output port is available, and regulates the timing of changes to FP_IN.DA.

5.1 Combining queue

The semi-systolic combining queue in the forward path component compares and enqueues messages with a hardware cost similar to that of a shift register; the comparator to recognize matching messages is distributed within each cell (Figure 8).

The queue of Figure 7 has a comparator between every slot in the IN row and the corresponding slot in the OUT row; this requires only local connections and only a small amount of logic. As messages move 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

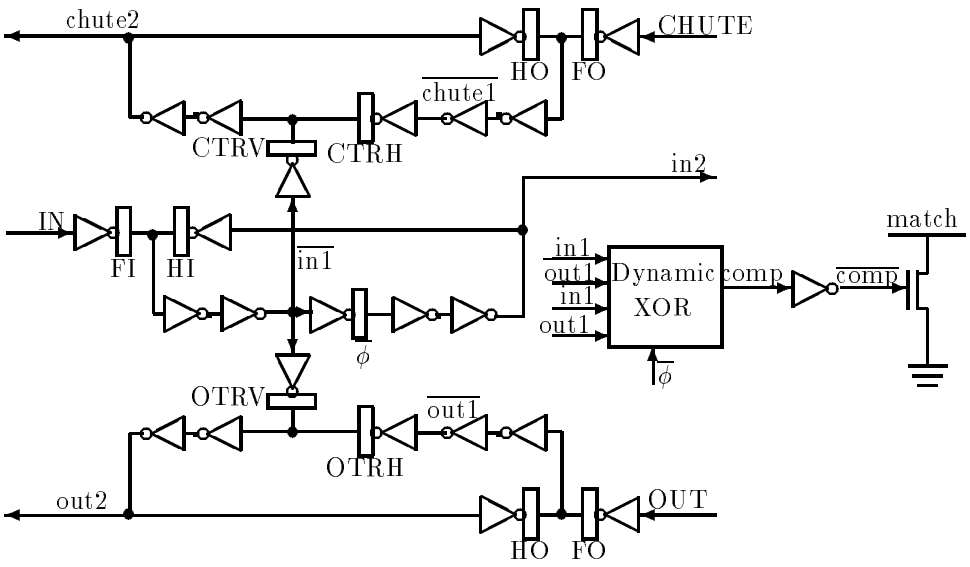

Figure 8: Schematic of a single cell of the combining queue in the forward path component.

can be done in parallel with data movement because the message in the IN row is copied to the the CHUTE as long as it is empty. A valid bit for each row in the slot is set based on the result of the match. The comparator to recognize matching messages is a few transistors distributed in each cell (Figure 8).

If the IN and the OUT row move simultaneously, an entering packet will be matched against every other message. If every message has two packets, an entering address packet will be matched against every message in the queue when it arrives except the first.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 [1,11]), so the addition of combining places no further constraint on the number of packets in a message.

A block diagram of the combining queue is shown in Figure 9. The input to the first valid cell is FP_IN.DV; the outputs are FP_OUT.DV and WB.DV.

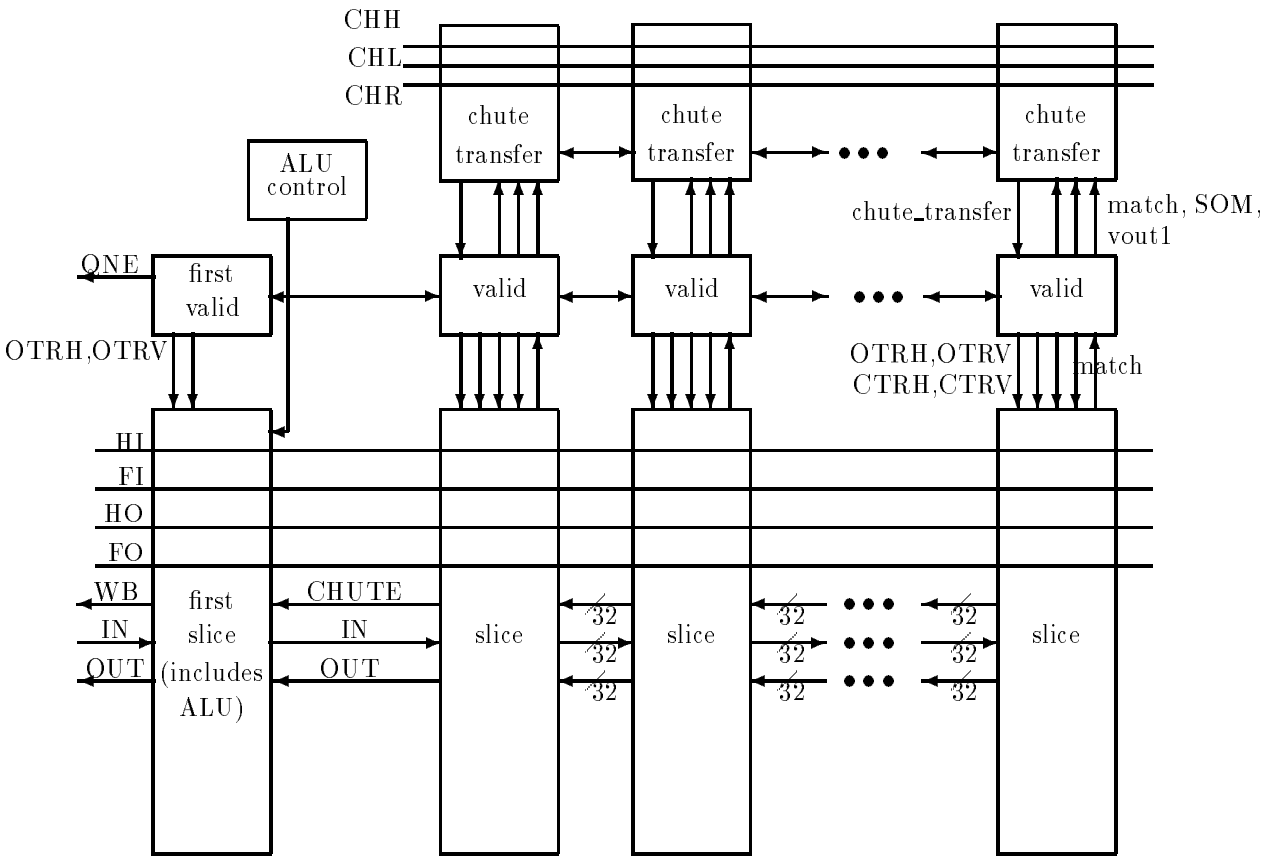

Figure 9: Block diagram of a combining queue implementation.

$$
\begin{array}{rcll}
\text{in1}(j) & := & \text{if} & \text{in_moving} \\
 & & & \quad \text{then in2}(j-1) \\
 & & & \quad \text{else in2}(j), \\
\text{out1}(j) & := & \text{if} & \text{out_moving} \\
 & & & \quad \text{then out2}(j+1) \\
 & & & \quad \text{else out2}(j), \\
\text{chute1}(j) & := & \text{if} & \text{out_moving} \\
 & & & \quad \text{then chute2}(j+1) \\
 & & & \quad \text{else chute2}(j), \\
\text{valid1(IN}, j) & := & \text{if} & \text{in_moving} \\
 & & & \quad \text{then valid2(IN}, j-1) \\
 & & & \quad \text{else valid2(IN}, j) \text{ and} \\
\text{valid1(OUT}, j) & := & \text{if} & \text{out_moving} \\
 & & & \quad \text{then valid2(OUT}, j+1) \\
 & & & \quad \text{else valid2(OUT}, j), \\
\text{valid1(CHUTE}, j) & := & \text{if} & \text{out_moving} \\
 & & & \quad \text{then valid2(OUT}, j+1) \\
 & & & \quad \text{else valid2(OUT}, j), \\
\text{in2}(j) & := & \text{in1}(j), & \\
\text{out2}(j) & := & \text{if} & \text{valid1(OUT}, j) \\
 & & & \quad \text{then out1}(j) \\
 & & & \quad \text{else in1}(j), \\
\text{chute2}(j) & := & \text{if} & \text{valid1(CHUTE}, j) \\
 & & & \quad \text{then chute1}(j) \\
 & & & \quad \text{else in1}(j), \\
\text{valid2(IN}, j) & := & \multicolumn{2}{l}{\text{valid1(IN}, j) \wedge \text{valid1(OUT}, j) \wedge} \\
 & & & \quad (\neg\text{chute_transfer1}(j) \vee \text{valid1(CHUTE}, j)), \\
\text{valid2(OUT}, j) & := & \multicolumn{2}{l}{\text{valid1(IN}, j) \vee \text{valid1(OUT}, j), } \\
\text{valid2(CHUTE}, j) & := & \multicolumn{2}{l}{(\text{valid1(IN}, j) \wedge \neg\text{chute_transfer1}(j))} \\
 & & & \quad \vee \text{valid1(CHUTE}, j).
\end{array}
$$

Figure 10: Combining queue transitions for slot j.

The transitions for slice j of a combining queue are shown in Figure 10. 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. When the IN row is moving and the OUT row is not moving, the chute_transfer signal must move right every other cycle (see Figure 11). When the IN row is not moving and the OUT row is moving the chute_transfer signal must move left every other cycle (see Figure 12). If both rows are moving or neither row is moving, it stays in place (see Figure 13). The CHH, CHR and CHL signals shown in Figure 9 control the movement of the chute_transfer signal between slices of the queue. The chute_transfer signal is reset from the match signal at the start of a valid message.

5.2 ALU

The 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 using the logic shown in Figure 14. The signals func0 and func1 represent the op codes according to the following table:

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; see Figure 15. 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 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 6). Each of these components was

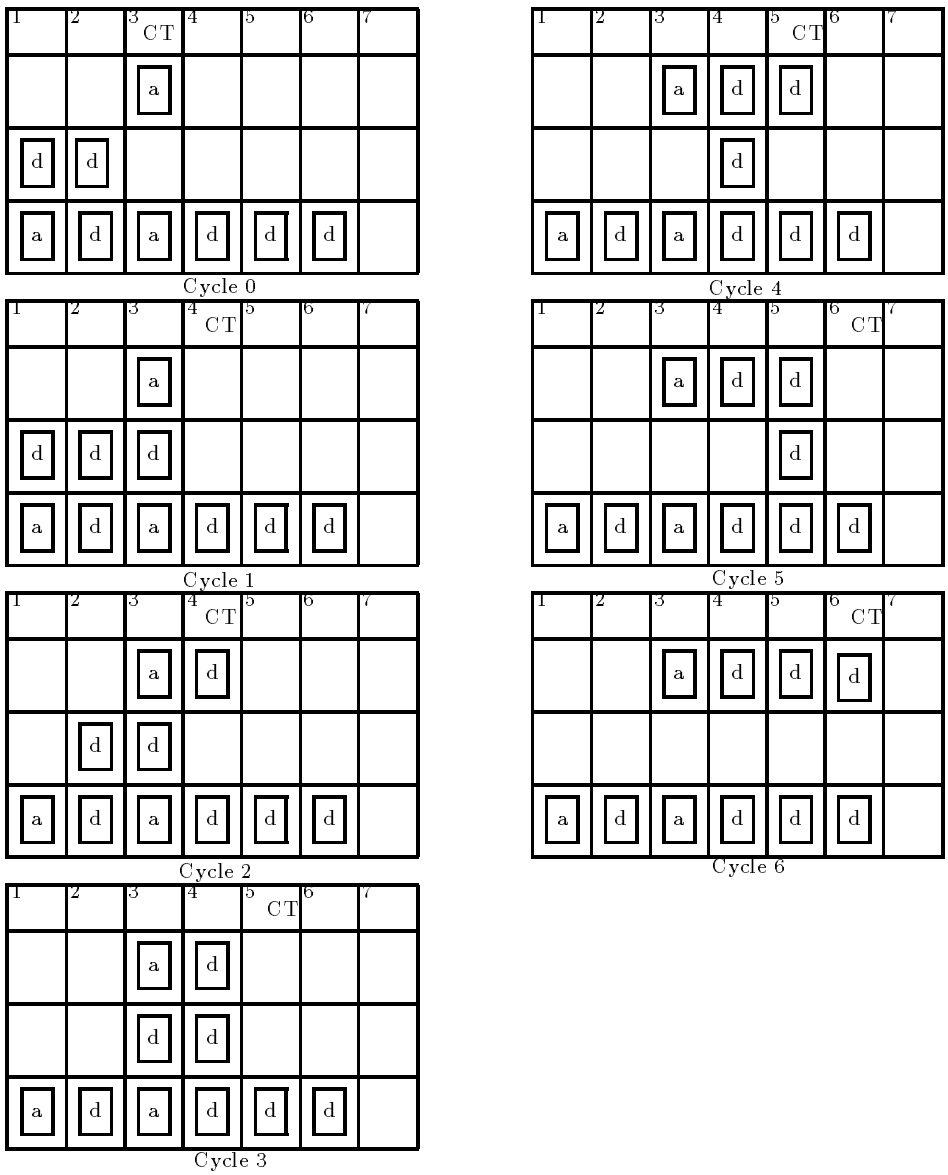

Figure 11: Behavior of chute transfer signal with IN moving and OUT not moving.

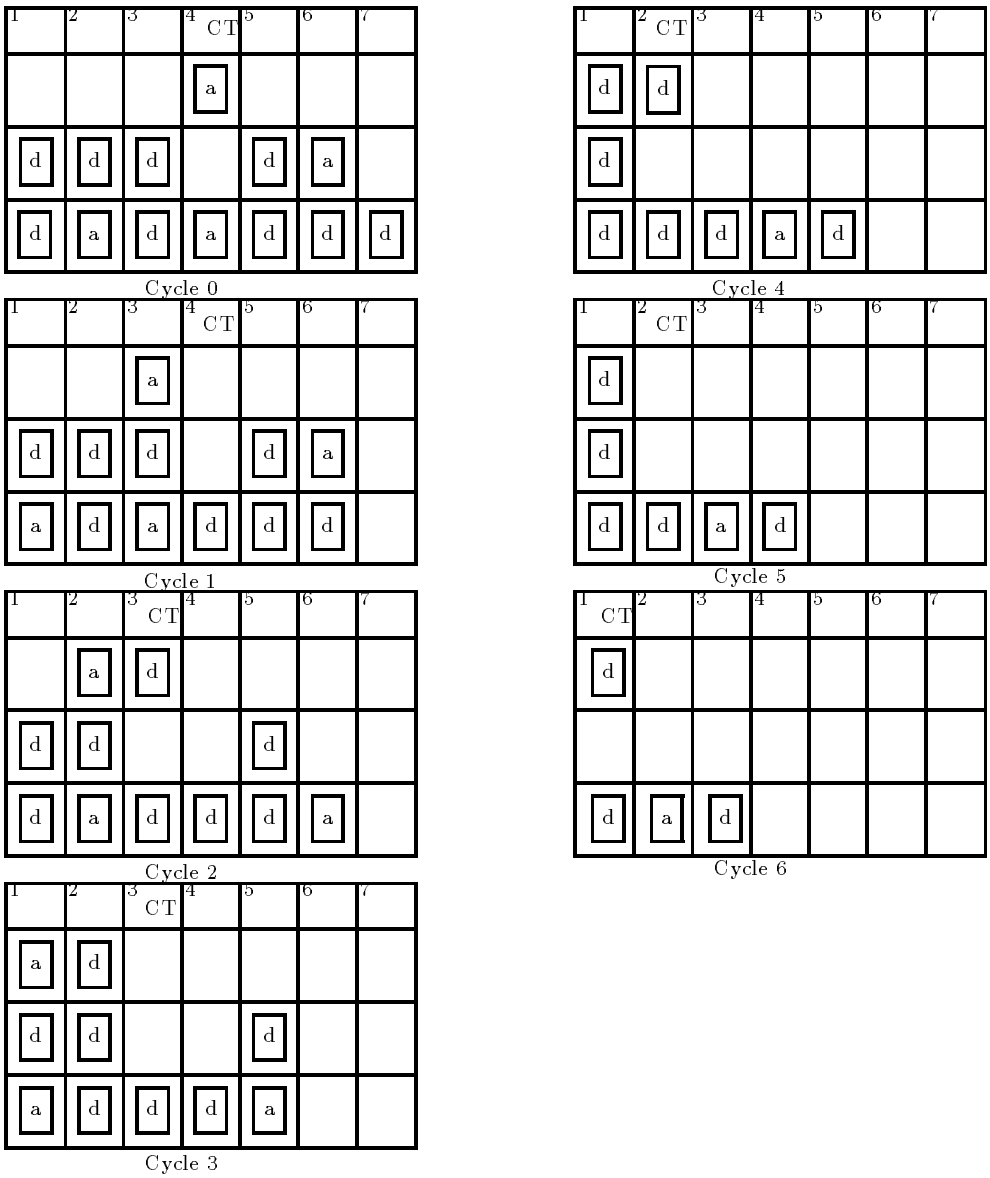

Figure 12: Behavior of chute transfer signal with OUT moving and IN not moving.

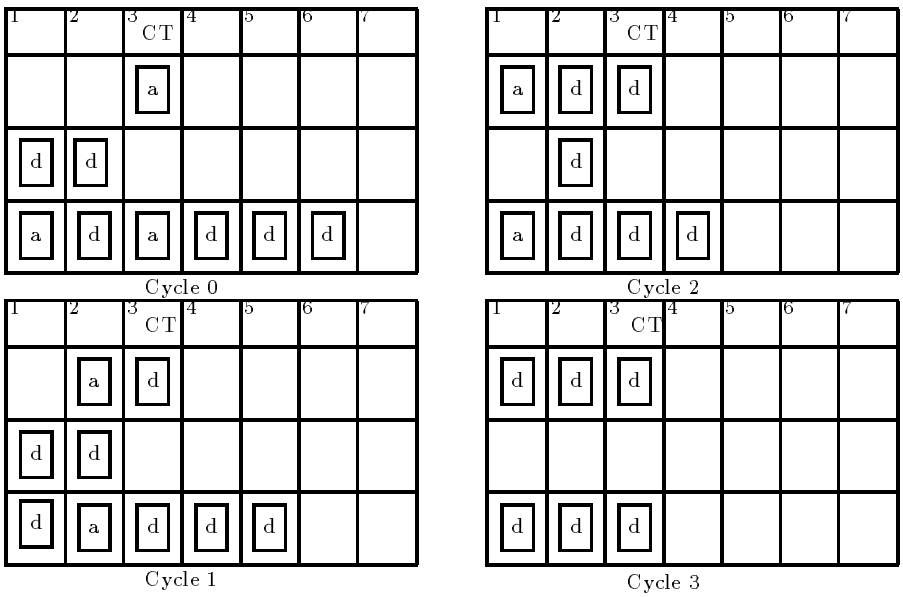

Figure 13: Behavior of chute transfer signal with IN and OUT both moving or both stopped.

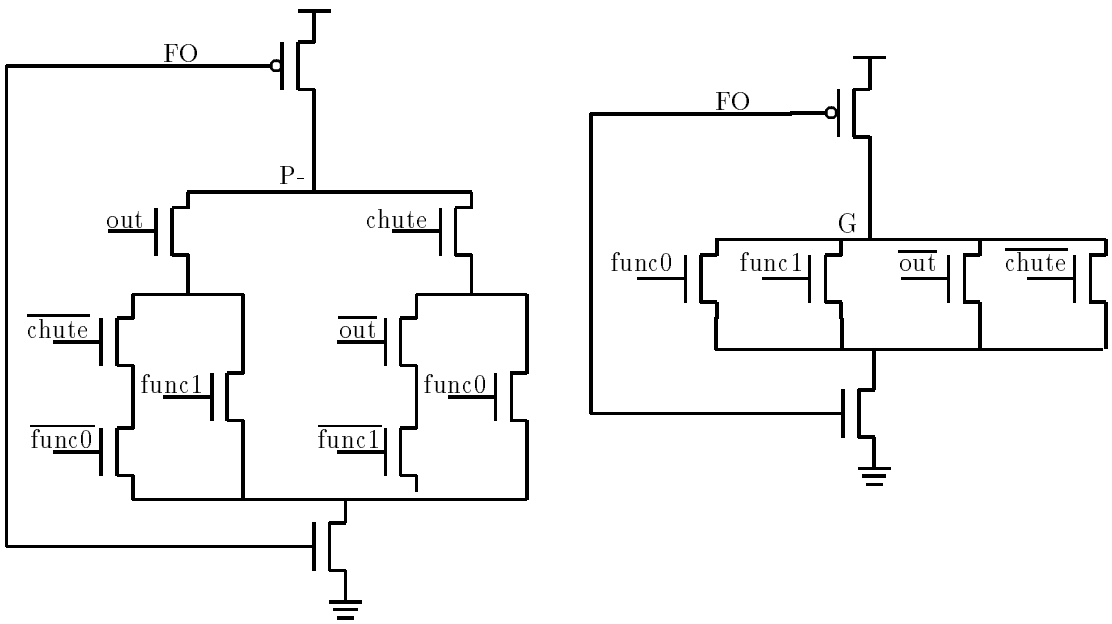

Figure 14: Logic to produce propagate and generate signals.

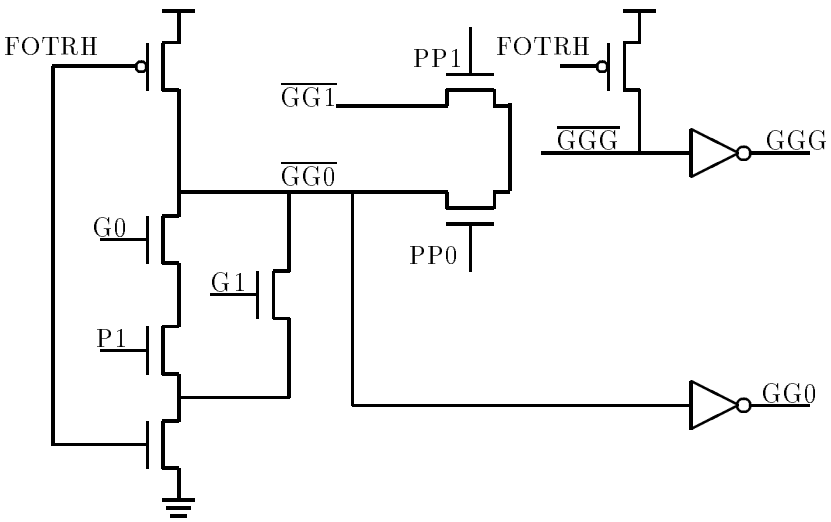

Figure 15: Multiple output Domino CMOS gate in carry chain.

fabricated and tested individually before completing the single-chip return path component design.

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 will never both be driven at the same time. A message received on RP_IN starting at cycle $2t$ is sent to the main queue and also to the wait buffer where its address packet is compared with the messages currently in the wait buffer. If a match is found, the wait buffer asserts its *match* line during cycle $2t + 1$. The output of the ALU is sent to the decombined queue at cycle $2t + 2$ so that the two queues receive the first packets of their messages at cycles of the same parity.

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 was required to get it right.

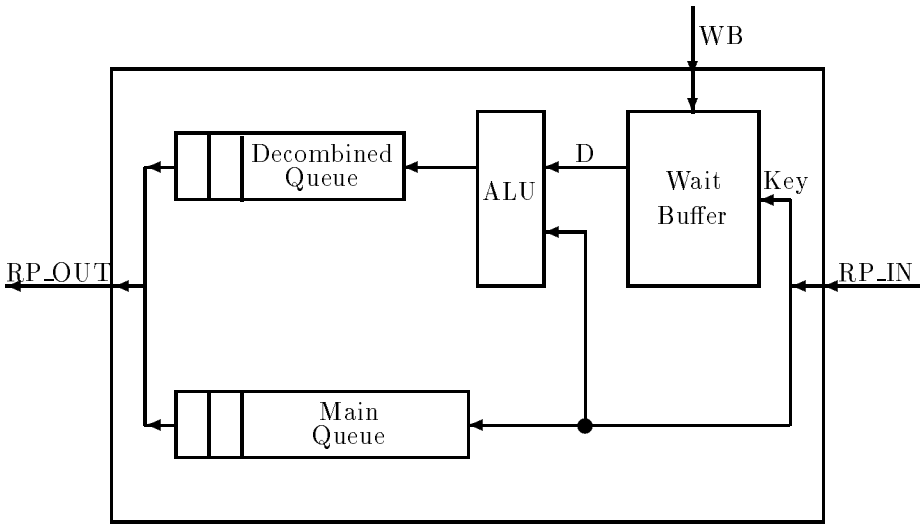

Figure 16: Block diagram of a return path component.

6.1 Wait buffer

The wait buffer is an associative memory that stores information sent by the FPC when combining two fetch-and-add's 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. The structure of a wait buffer (WB) is shown in Figure 17. 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 17.

A typical message slot is shown in Figure 18 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 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 operations we support, no message requires more than one data packet to be stored in the wait buffer.

Each slot connects to the following buses:

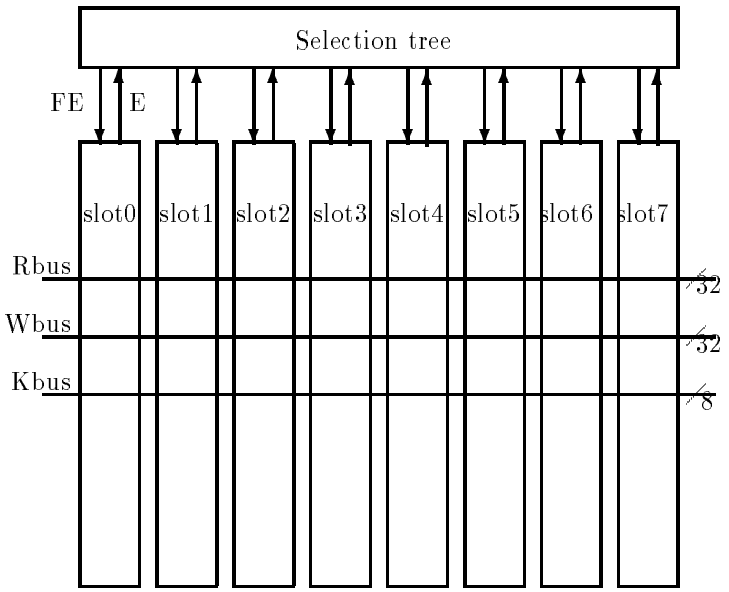

Figure 17: Block diagram of a wait buffer.

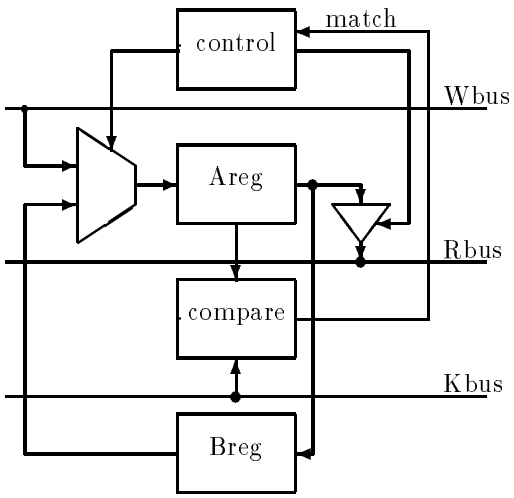

Figure 18: Slot of a wait buffer holding a two-packet message.

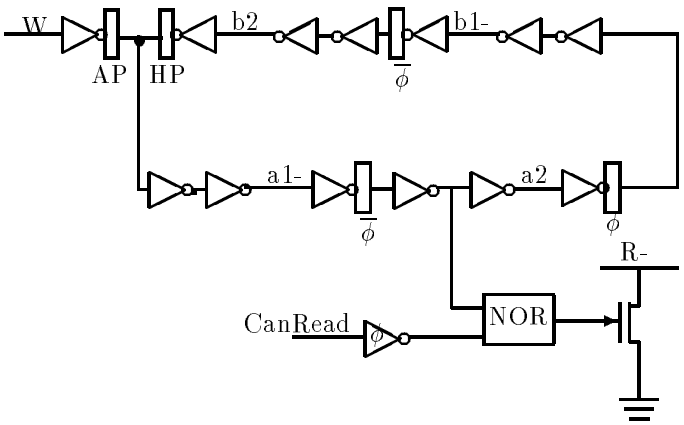

Figure 19: Schematic of a wait buffer cell.

- 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 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 lines for each slot. As an input to the logic that sets the valid bit for a slot, FE selects the slot to receive a new message when it is asserted.

A schematic of a wait buffer cell is shown in Figure 6.1. HP ("Hold Packet") is asserted if the cell already has a message; otherwise AP ("Advance Packet") is asserted. CanRead is computed from the match signal at the cycle of appropriate parity. The match signal is computed in the same way as in the combining queue cell, for two bits at a time.

6.2 ALU and non-combining systolic queues

An ALU is used to generate the second response to a fetch-and-ϕ operation by operating on the data packet received from a 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. Since the wait buffer is not

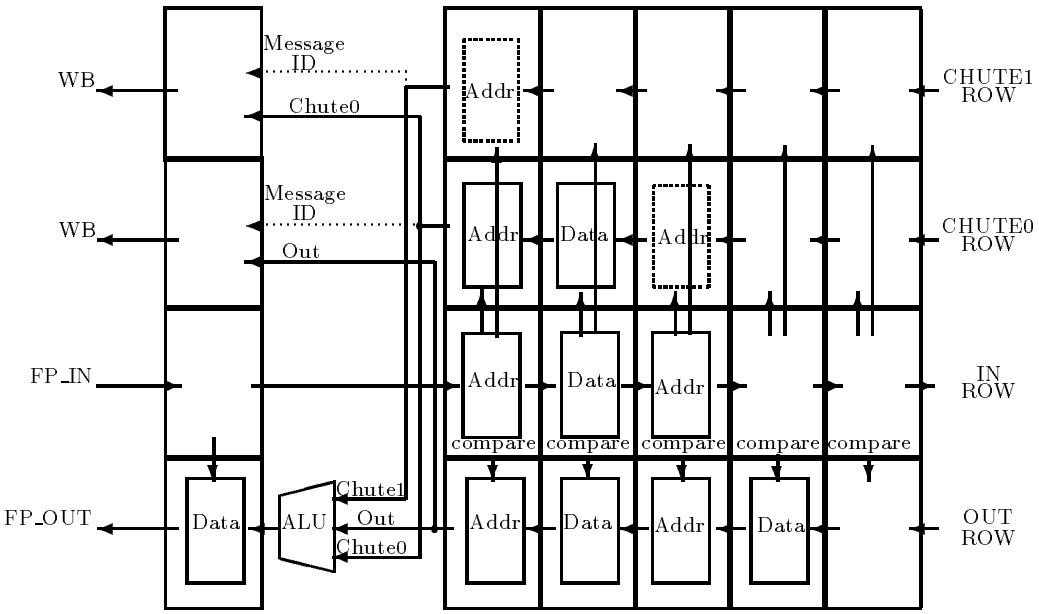

Figure 20: A queue capable of combining three requests.

required to respond until two cycles after it receives a matching memory response, the adder can take a full cycle.

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. The decombined queue and the main queue are multiplexed with the queues from the paired RPC for control of the output port.

7 Queues with greater combining capability

In our design, a combining queue with 2 inputs and 1 output is implemented by multiplexing the outputs of two combining queues each having 1 input and 1 output. Let the abbreviation a, b queue stand for a combining queue having a-inputs and b-outputs. Using this terminology, we would say that our current 2,1 queue is implemented as two 1,1 queues and a multiplexor. A disadvantage of our 2,1 queue implementation is that requests in separate 1,1 queues cannot be combined whereas in an ideal 2,1 queue they would be. A further limitation of the current design is that it combines only

pairs of requests so that if N requests to a single combining queue are all directed at the same address, at least $\lceil N/2 \rceil$ will emerge. Requests to the same location that enter via different ports will not be combined but will exit via the same output port and hence will be eligible for combination at the next stage of the network. In particular, if we ignore the (somewhat anomalous) situation in which a *single* processor has multiple outstanding requests for the same memory location, the first stage of the network will perform no combining.

7.1 Extension to support three-way combining

The design of Figure 7 can be extended to support three-way combining by adding a second CHUTE as shown in Figure 20.

When an entry in the IN row matches one in OUT, the first entry moves to a CHUTE unless the corresponding entries in both CHUTEs are full, which means that three items have already combined. This design presents several difficulties.

- Logic is needed to decide which CHUTE is to obtain a matched entry.
- Two messages must be sent to the wait buffer. If they are sent on the same cycle, more pins will be needed. If they are sent serially, there is a problem with flow control with consecutive triple combines.
- The ALU in the FPC must now accept three inputs. For correct decombining, the ALU in the RPC must either accept three inputs, to combine the Out data packet and the Chute0 data packet with the the data packet from memory, or the Out and Chute0 data packet must be added together before being stored in the wait buffer.
- The wait buffer must be able to match two messages at once.
- Bursts of three messages may increase delays on the return path [7].

One can use still more CHUTEs to support higher levels of combining but the VLSI layout seems likely to be poor, the multi-input adder is likely to be slow, and results in [6,9] indicate that 3-way combining suffices for two-way switches.

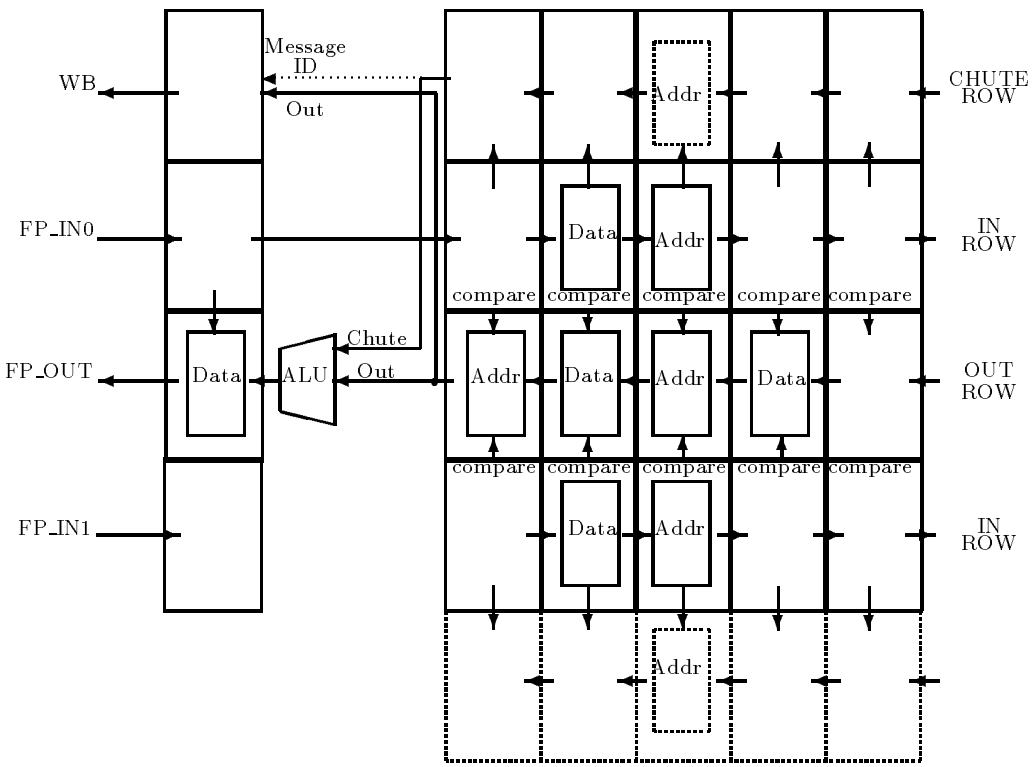

Figure 21: A 2-input combining queue with one CHUTE

7.2 Implementation of 2,1 queues

These designs implement 2,1 queues directly without using 1,1 queues. Again starting with Figure 7, we add a second IN row and a second row of comparators to the bottom of the OUT row. The result is shown in Figure 21.

There is actually only one CHUTE; the dashed row on the bottom is the same as the top row. Since the vertical lines from IN1 to CHUTE are actually wires passing through several rows, the VLSI layout would be poor. The arbitration that is performed just on the output in the switch of Figure 1 now must be performed twice for every column, once for the OUT row and once for the CHUTE row.

With only a single CHUTE, three-way combining is impossible (the CHUTE entry fills when a pair combines) but the restriction that both partners come from the same input port has been eliminated.

These ideas can be combined to obtain a 2-input queue capable of combining three requests to the same memory location. With two IN rows each able to shift to either of two CHUTEs, the queue can combine three requests regardless of their input ports. This implementation has all difficulties of the 1,1 queue with three-way combining. Furthermore, the VLSI layout will be more costly per cell due to the multiple connections, and arbitration must take place for both CHUTE rows as well as the OUT row.

8 Two-and-a-half-way combining

We propose "two-and-a-half-way combining" as an alternative with better layout and decombining properties than three-way combining that will still show good performance for very large networks. As shown in Figure 22, two-and-a-half-way combining limits the connections of the IN rows to the neighboring CHUTE rows. Three messages can combine as long as the second and third messages come from different inputs. Comparison of three-way with two-and-a-half-way in [9] show no significant performance difference between the two for systems with 1024 PEs and MMs.

The logic to do two-and-a-half way combining requires

- No additional storage or data paths in the main part of the two-input queue over that for the current implementation of two one-input queues.

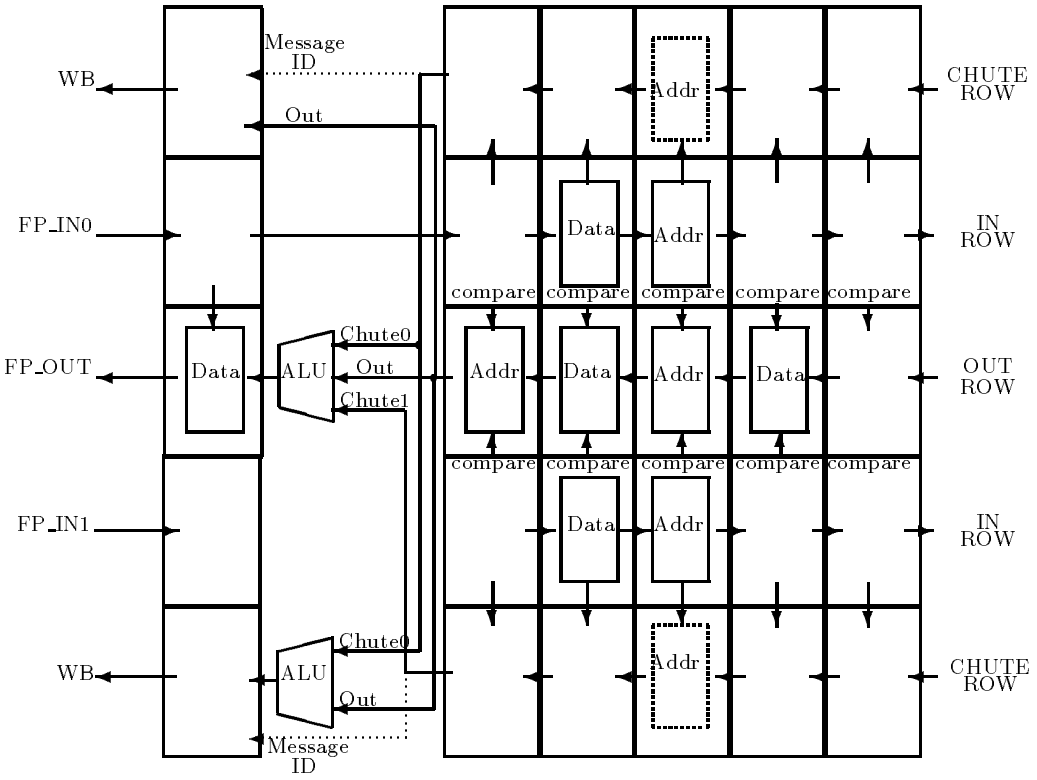

Figure 22: Two-and-a-half-way combining queue

- The same number of wait buffer input ports as in eight chip implementation with 1,1 queues, though the buses cannot be tied together and thus more pins would be required for the 4 chip and 2 chip designs in Table 2.
- Additional ALU operations on the Out and two Chute outputs, either with a tree of 2-way ALUs or with special 3-way carry logic, as for the 1,1 queue with three-way combining.

Since the combined messages came in on different input ports on the forward path, they are guaranteed to exit from different wait buffers on the return path. Therefore, decombining is unlikely to create a problem with bursty traffic. Furthermore, if the combination of the data packets from the OUT row and one of the CHUTE rows is done on the forward path, the wait buffer design can remain simple. With one wait buffer for each input-output pair, the two combined messages will be in different wait buffers, and can be matched and queued in parallel.

References

[1] Susan R. Dickey and Richard Kenner. A combining switch for the NYU Ultracomputer. Ultracomputer Note No. 178, New York University, February 1992.

[2] 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.

[3] 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.

[4] Allan Gottlieb and Clyde P. Kruskal. Coordinating parallel processors: A partial unification. *SIGARCH*, October 1981.

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

[6] Gyung Ho Lee, Clyde P. Kruskal, and David J. Kuck. The effectiveness of combining in shared memory parallel computers in the presence of 'hot spots'. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 35–41, August 1986.

[7] Gyungho Lee. Another combining scheme to reduce hot spot contention in large-scale shared memory parallel computers. In *Lecture Notes in Computer Science, Vol 297: Supercomputing*, pages 68–79. Springer-Verlag, 1988.

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

[9] 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.

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

[11] Marc Snir and Jon Solworth. The Ultraswitch – a VLSI network node for parallel processing. Ultracomputer Note No. 39, New York University, January 1984.