Distributed Computing (1986) 1:187–196

DISTRIBUTED COMPUTING

Original articles

The torus routing chip*

William J. Dally and Charles L. Seitz
Department of Computer Science, California Institute of Technology, Pasadena, CA, USA

Bill Dally *received his B.S. degree in Electrical Engineering from the Virginia Polytechnic Institute in 1980 and his M.S. degree in Electrical Engineering from Stanford University in 1981. From 1980 to 1982 he worked at Bell Telephone Laboratories, where he contributed to the design of the BELLMAC-32 microprocessor. From 1982 to 1983 he worked as a consultant in the area of digital systems design. Since 1983 he has been a graduate student in Computer Science at Caltech, and is expected to complete his Ph.D. studies in the spring 1986. His current research interests include computer architecture, computer aided design, VLSI, design, and concurrent systems.*

Chuck Seitz *earned B.S., M.S., and Ph.D. degrees from M.I.T. Before joining the Computer Science faculty at Caltech in 1977, he worked as a member of the technical staff of the Evans & Sutherland Computer Corporation from 1969 to 1971, as an Assistant Professor of Computer Science at the University of Utah from 1970 to 1972, and as a consultant to Burroughs Corporation from 1971 to 1978. He is currently a Professor of Computer Science at Caltech, where his research and teaching activities are in the areas of VLSI architecture and design, concurrent computation, and self-timed systems.*

* The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597, in part by Intel Corporation, and in part by an AT & T Ph.D. fellowship

Abstract. The torus routing chip (TRC) is a self-timed chip that performs deadlock-free *cut-through* routing in k-ary n-cube multiprocessor interconnection networks using a new method of deadlock avoidance called *virtual channels*. A prototype TRC with byte wide self-timed communication channels achieved on first silicon a throughput of 64 Mbits/s in each dimension, about an order of magnitude better performance than the communication networks used by machines such as the Caltech Cosmic Cube or Intel iPSC. The latency of the cut-through routing of only 150 ns per routing step largely eliminates message locality considerations in the concurrent programs for such machines. The design and testing of the TRC as a self-timed chip was no more difficult than it would have been for a synchronous chip.

Key words: VLSI – Interconnection networks – Communication networks – Concurrent computation – Parallel processing – Deadlock-free routing – Self-timed logic – Asynchronous logic – Message-passing multiprocessers

1 Introduction

Message-passing concurrent computers such as the Caltech Cosmic Cube [13] and Intel iPSC [6] consist of many processing *nodes* that interact by sending messages over communication channels between the nodes. We designed the torus routing chip (TRC) as a building block to construct high-throughput, low-latency, k-ary n-cube interconnection networks for message-passing concurrent computers.

The TRC is a self-timed VLSI circuit that provides deadlock-free packet communications in k-ary n-cube (torus) networks [12] with up to $k=256$

processors in each dimension. While intended primarily for $n=2$-dimensional networks, the chips can be cascaded to handle n-dimensional networks using $\lceil \frac{n}{2} \rceil$ TRC chips at each processing node. A prototype TRC has been laid out, fabricated, and tested.

Even if only two dimensions are used, the TRC can be used to construct concurrent computers with up to 2^{16} nodes. It would be very difficult to distribute a global clock over an array of this size [4]. To avoid this problem, the TRC is entirely self-timed [11], thus permitting each processing node to operate at its own rate with no need for global synchronization. Synchronization, when required, is performed by arbiters in the TRC.

To reduce the latency of communications that traverse more than one channel, the TRC uses *cut-through* [7] routing rather than *store-and-forward* routing. Instead of reading an entire packet into a processing node before starting transmission to the next node, the TRC forwards each byte of the packet to the next node as soon as it arrives. Cut-through routing thus results in a message latency that is the *sum* of two terms, one of which depends on the message length, L, and other of which depends on the number of communication channels traversed, D. Store-and-forward routing gives a latency that depends on the product of L and D. Another advantage of cut-through routing is that communications do not use up the memory bandwidth of intermediate nodes. A packet does not interact with the processor or memory of intermediate nodes along its route. Packets remain strictly within the TRC network until they reach their destination.

The TRC uses *virtual channels* to perform deadlock-free routing in torus networks. By splitting each physical channel into two virtual channels and making routing dependent on the virtual channel on which a message arrives, the TRC converts the cycle of channel dependencies in each dimension into a spiral.

This paper describes the considerations that went into the design of the TRC in a "top-down" order that starts with a formal discussion of the deadlock problem in Sect. 2. We develop a model of communications in multiprocessor interconnection networks and prove a strong theorem about deadlock. Based on this model, the concept of virtual channels is presented in Sect. 3. Sections 4 and 5 present the design of the TRC at the system and logical levels. Experimental results are reviewed in Sect. 6.

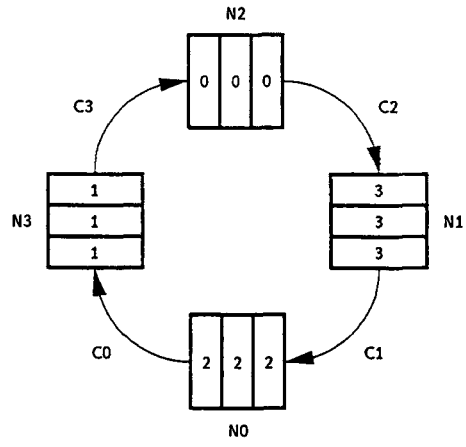

Fig. 1. Deadlock in a 4-cycle

2 Deadlock-free routing

Deadlock in the interconnection network of a concurrent computer occurs when no message can advance toward its destination because the queues of the message system are full [8]. Consider the example shown in Fig. 1. The queues of each node in the 4-cycle are filled with messages destined for the opposite node. No message can advance toward its destination; thus the cycle is deadlocked. In this locked state, no communication can occur over the deadlocked channels until exceptional action is taken to break the deadlock.

The technique of virtual channels allows deadlock-free routing to be performed in any strongly-connected interconnection network [2]. This technique involves splitting physical channels on cycles into multiple virtual channels and then restricting the routing so the dependence between the virtual channels is acyclic.

Definition 1. A flow control digit or *flit* is the smallest unit of information that a queue or channel can accept or refuse. Generally a *packet* consists of many flits. Each packet carries its own routing information.

We have adopted this complication of standard terminology to distinguish between those flow control units that always include routing information – viz. packets – and those lower level flow control units that do not – viz. flits. The literature on computer networks [16] has been able to avoid this distinction between packets and flits because most networks include routing information with every flow control unit; thus the flow control units are packets. That is not the case in the interconnection

networks used by message-passing concurrent computers such as the Caltech Cosmic Cube [13].

We assume the following:

1. Every packet arriving at its destination node is eventually consumed.
2. A node can generate packets destined for any other node.
3. The route taken by a packet is determined only by its destination, and not by other traffic in the network.
4. A node can generate packets of arbitrary length. Packets will generally be longer than a single flit.
5. Once a queue accepts the first flit of a packet, it must accept the remainder of the packet before accepting any flits from another packet.
6. An available queue may arbitrate between packets that request that queue space, but may not choose amongst waiting packets.
7. Nodes can produce packets at any rate subject to the constraint of available queue space (source queued).

The following definitions develop a notation for describing networks, routing functions, and configurations.

Definition 2. An *interconnection network*, I, is a strongly connected *directed* graph, $I = G(N, C)$. The vertices of the graph, N, represent the set of processing nodes. The edges of the graph, C, represent the set of communication channels. Associated with each channel, c_i, is a queue with capacity $\text{cap}(c_i)$. The source node of channel c_i is denoted s_i and the destination node d_i.

Definition 3. A *routing function* $\mathbf{R}$: $C \times N \longrightarrow C$ maps the current channel, c_c, and destination node, n_d, to the next channel, c_n, on the route from c_c to n_d, $\mathbf{R}(c_c, n_d) = c_n$. A channel is not allowed to route to itself, $c_c \neq c_n$. Note that this definition restricts the routing to be memoryless in the sense that a packet arriving on channel c_c destined for n_d has no memory of the route that brought it to c_c. However, this formulation of routing as a function from $C \times N$ to C has more memory than the conventional definition of routing as a function from $N \times N$ to C. Making routing dependent on the current channel rather than the current node allows us to develop the idea of channel dependence. Observe also that the definition of $\mathbf{R}$ precludes the route from being dependent on the presence or absence of other traffic in the network. $\mathbf{R}$ describes strictly deterministic and non-adaptive routing functions.

Definition 4. A *channel dependency graph*, D, for a given interconnection network, I, and routing function, $\mathbf{R}$, is a directed graph $D = G(C, E)$. The vertices of D are the channels of I. The edges of D are the pairs of channels connected by $\mathbf{R}$:

$$E = \{(c_i, c_j) \mid \mathbf{R}(c_i, n) = c_j \text{ for some } n \in N\}. \quad (1)$$

Since channels are not allowed to route to themselves, there are no 1-cycles in D.

Definition 5. A *configuration* is an assignment of a subset of N to each queue. The number of flits in the queue for channel c_i will be denoted $\text{size}(c_i)$. If the queue for channel c_i contains a flit destined for node n_d, then $\text{member}(n_d, c_i)$ is true. A configuration is legal if

$$\forall c_i \in C, \text{size}(c_i) \leq \text{cap}(c_i). \quad (2)$$

Definition 6. A *deadlocked configuration* for a routing function, $\mathbf{R}$, is a non-empty legal configuration of channel queues such that

$$\forall c_i \in C, (\forall n \ni \text{member}(n, c_i), n \neq d_i$$

and

$$c_j = \mathbf{R}(c_i, n) \Rightarrow \text{size}(c_j) = \text{cap}(c_j)) \quad (3)$$

In this configuration no flit is one step from its destination, and no flit can advance because the queue for the next channel is full. A routing function, $\mathbf{R}$, is *deadlock-free* on an interconnection network, I, if no deadlock configuration exists for that function on that network.

Theorem 1. *A routing function, $\mathbf{R}$, for an interconnection network, I, is deadlock-free iff there are no cycles in the channel dependency graph, D.*

Proof.

$\Rightarrow$ Suppose a network has a cycle in D. Since there are no 1-cycles in D, this cycle must be of length two or more. Thus one can construct a deadlocked configuration by filling the queues of each channel in the cycle with flits destined for a node two channels away, where the first channel of the route is along the cycle.

$\Leftarrow$ Suppose a network has no cycles in D. Since D is acyclic one can assign a total order to the channels of C so that if $(c_i, c_j) \in E$ then $c_i > c_j$. Consider the least channel in this order with a full queue, c_l. Every channel, c_n, that c_l feeds is less than c_l, and thus does not have a full queue. Thus, no flit in the queue for c_l is blocked, and one does not have deadlock. □

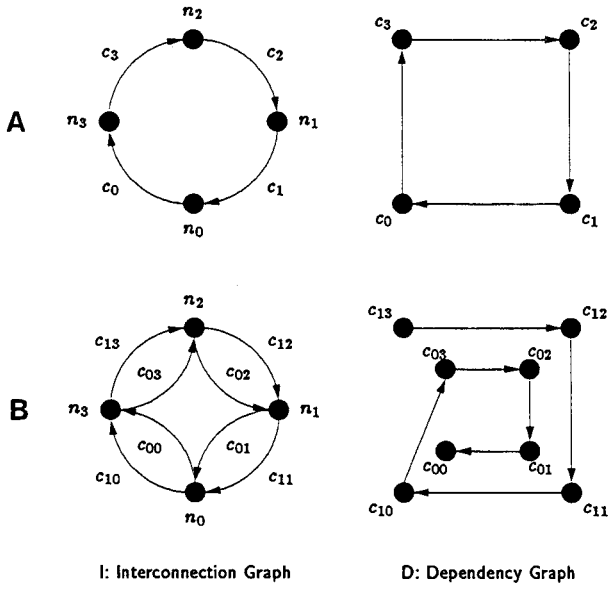

Fig. 2. Breaking deadlock with virtual channels

3 Virtual channels

Now that we have established this if-and-only-if relationship between deadlock and the cycles in the channel dependency graph, we can approach the problem of making a network deadlock-free by breaking the cycles. We can break such cycles by splitting each physical channel along a cycle into a group of *virtual channels*. Each group of virtual channels shares a physical communication channel; however, each virtual channel requires its own queue.

Consider for example the case of a unidirectional four-cycle shown in Fig. 2A, $N=\{n_0, \ldots, n_3\}$, $C=\{c_0, \ldots, c_3\}$. The interconnection graph I is shown on the left and the dependency graph D is shown on the right. We pick channel c_0 to be the dividing channel of the cycle and split each channel into high virtual channels, $c_{10}, \ldots, c_{13}$, and low virtual channels, $c_{00}, \ldots, c_{03}$, as shown in Fig. 2B.

Packets at a node numbered less than their destination node are routed on the high channels, and packets at a node numbered greater than their destination node are routed on the low channels. Channel c_{00} is not used. We now have a total ordering of the virtual channels according to their subscripts:

$$c_{13} > c_{12} > c_{11} > c_{10} > c_{03} > c_{02} > c_{01}.$$

Thus, there is no cycle in D, and the routing function is deadlock-free. In [2] this technique is applied to construct deadlock-free routing functions for k-ary n-cubes, cube-connected cycles, and shuffle-exchange networks. In each case virtual channels are added to the network and the routing is restricted to route packets in order of decreasing channel subscripts. In the next two sections, the routing function for k-ary n-cubes is developed into a chip.

Many deadlock-free routing algorithms have been developed for store-and-forward computer communications networks [5]. These algorithms are all based on the concept of a *structured buffer pool*. The packet buffers in each node of the network are partitioned into classes, and the assignment of buffers to packets is restricted to define a partial order on buffer classes. The structured buffer pool method has in common with the virtual channel method that both prevent deadlock by assigning a partial order to resources. The two methods differ in that the structured buffer pool approach restricts the assignment of buffers to packets while the virtual channel approach restricts the routing of messages. Either method can be applied to store-and-forward networks, but the structured buffer pool approach is not directly applicable to cut-through networks, since the flits of a packet cannot be interleaved.

4 System design

The torus routing chip (TRC) can be used to construct arbitrary k-ary n-cube interconnection networks. Each TRC routes packets in two dimensions, and the chips are cascadable as shown in Fig. 3 to construct networks of dimension greater than two. The first TRC in each node routes packets in the first two dimensions and strips off their address bytes before passing them to the second TRC. This next chip then treats the next two bytes as addresses in the next two dimensions and routes

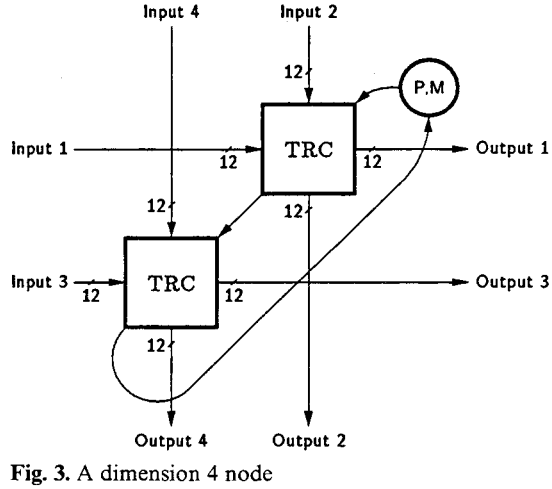

Fig. 3. A dimension 4 node

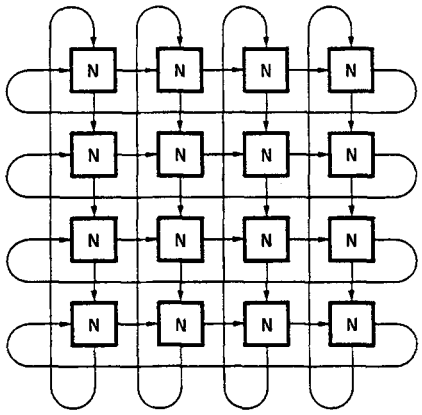

Fig. 4. A torus system

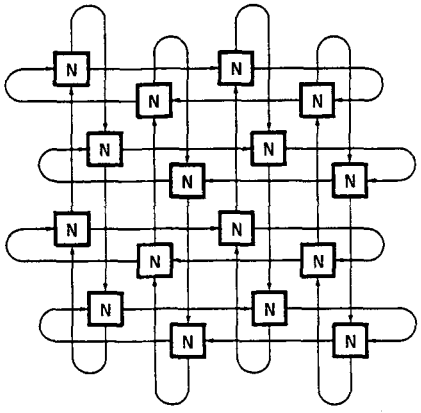

Fig. 5. A folded torus system

packets accordingly. The network can be extended to any number of dimensions.

A block diagram of a two-dimensional message-passing concurrent computer constructed around the TRC is shown in Fig. 4. Each node consists of a processor, its local memory, and a TRC. Each TRC in the torus is connected to its processor by a processor input channel and a processor output channel. Connections on the edges of the torus wrap around to the opposite edge. One can avoid the long end-around connection by folding the torus, as shown in Fig. 5.

A *flit* in the TRC is a byte whose 8 bits are transmitted in parallel. The X and Y channels each consist of 8 data lines and 4 control lines. The 4 control lines are used for separate request/acknowledge signal pairs for each of two virtual channels. The processor channels are also 8 bits wide, but have only two control lines each.

The packet format is shown in Fig. 6. A packet begins with two address bytes. The bytes contain the relative X and Y addresses of the destination

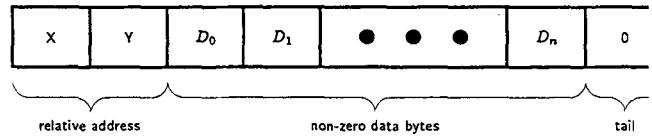

Fig. 6. Packet format

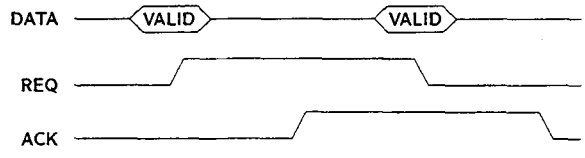

Fig. 7. Virtual channel protocol

node. The relative address in a given direction, say X, is a count of the number of channels that must be traversed in the X direction to reach a node with the same X address as the destination. After the address comes the data field of the packet. This field may contain any number of *non-zero* data bytes. The packet is terminated by a zero tail byte. Later versions of the TRC may use an extra bit to tag the tail of a packet, and might also include error checking.

The TRC network routes packets first in the X direction, then in the Y direction, Packets are routed in the direction of decreasing address, decrementing the relative address at each step. When the relative X address is decremented to zero, the packet has reached the correct X coordinate. The X address is then stripped from the packet, and routing is initiated in the Y dimension. When the Y address is decremented to zero, the packet has reached the destination node. The Y address is then stripped from the packet, and the data and tail bytes are delivered to the node.

Each of the X and Y physical channels is multiplexed into two virtual channels. In each dimension packets begin on virtual channel 1. A packet remains on virtual channel 1 until it reaches its destination or address zero in the direction of routing. After a packet crosses address zero it is routed on virtual channel 0. The address 0 origin of the torus network in X and Y is determined by two input pins on the TRC. The effect of this routing algorithm is to break the channel dependency cycle in each dimension into a two-turn spiral similar to that shown in Fig. 2. Packets enter the spiral on the outside turn and reach the inside turn only after passing through address zero.

Each virtual channel in the TRC uses the 2-cycle signaling convention shown in Fig. 7. Each virtual channel has its own request (R) and ac-

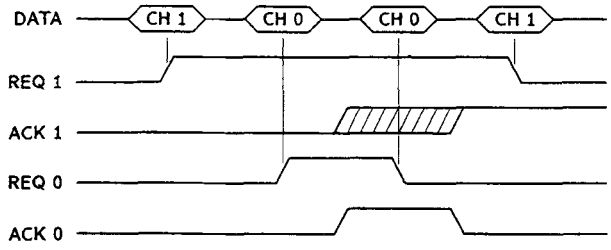

Fig. 8. Channel protocol example

knowledge (A) lines. When $R=A$, the receiver is ready for the next flit (byte). To transfer information, the sender waits for $R=A$, takes control of the data lines, places data on the data lines, toggles the R line, and releases the data lines. The receiver samples data on each transition of R line. When the receiver is ready for the next byte, it toggles the A line.

The protocol allows both virtual channels to have requests pending. The sending end does not wait for any action from the receiver before releasing the channel. Thus, the other virtual channel will never wait longer than the data transmission time to gain access to the channel. Since a virtual channel always releases the physical channel after transmitting each byte, the arbitration is fair. If both channels are always ready, they will alternate bytes on the pyhsical channel.

Consider the example shown in Fig. 8. Virtual channel X1 gains control of the physical channel, transmits one byte of information, and releases the channel. Before this information is acknowledged, channel X0 takes control of the channel and transmits two bytes of information. Then X1, having by then been acknowledged, takes the channel again.

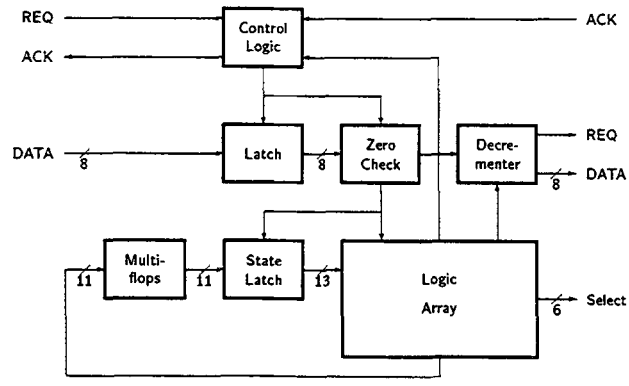

Fig. 10. Input controller block diagram

5 Logic design

As shown in Fig. 9, the TRC consists of five input controllers, a five by five crossbar switch, five output queues, and two output multiplexers. There is one input controller and one output controller for each virtual channel. The output multiplexers serve to multiplex two virtual channels onto a single physical channel.

The input controller is responsible for packet routing. When a packet header arrives, the input controller selects the output channel, adjusts the header by decrementing and sometimes stripping the byte, and then passes all bytes to the crossbar switch until the tail byte is detected.

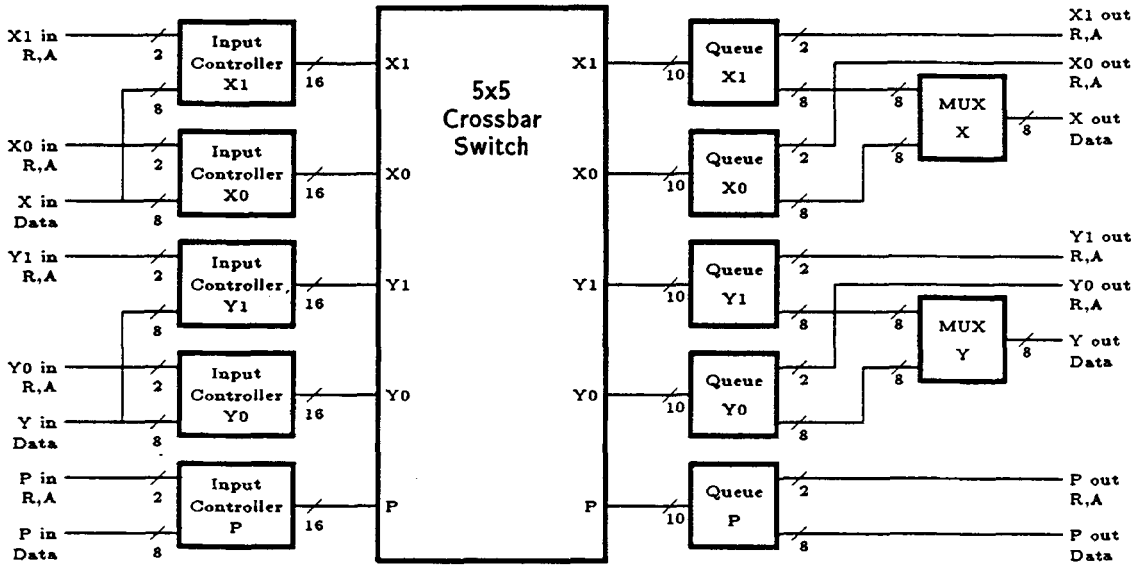

Fig. 9. TRC block diagram

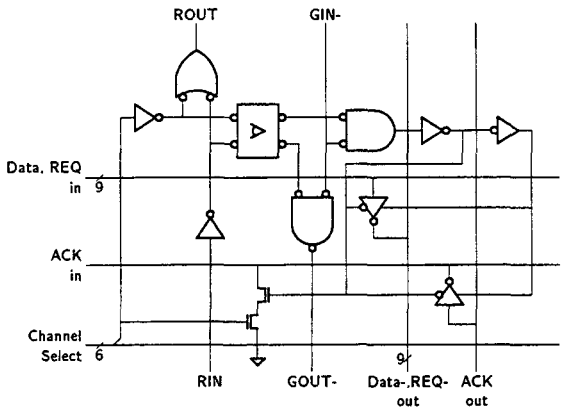

Fig. 11. Crosspoint of the crossbar switch

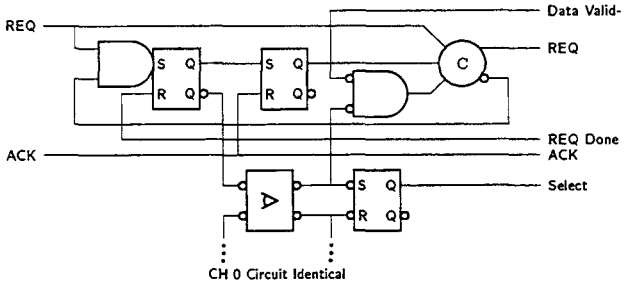

Fig. 12. Output multiplexer control

The input controller, shown in Fig. 10, consists of a datapath and a self-timed state machine. The datapath contains a latch, a zero checker, and a decrementer. A state latch, logic array, and control logic comprise the state machine. When the request line for the channel is toggled, data is latched, and the zero checker is enabled. When the zero checker makes a decision, the logic array is enabled to determine the next state, the selected crossbar channel, and whether to strip, decrement, or pass the current byte. When the required operation has been completed, possibly requiring a round trip through the crossbar, the state and selected channel are saved in cross-coupled multi-flops and the logic array is precharged.

The input controller and all other internal logic operates using a 4-cycle self-timed signaling convention [11]. One function of the state machine control logic is to convert the external 2-cycle signaling convention into the on-chip 4-cycle signaling convention. The signaling convention is converted back to 2-cycle at the output pads.

The crossbar switch performs the switching and arbitration required to connect the five input controllers to the five output queues. A single crosspoint of the switch is shown in Fig. 11. A two-input interlock (mutual-exclusion) element in each crosspoint arbitrates requests from the current input channel (row) with requests from all lower channels (rows). The interlock elements are connected in a priority chain so that an input channel must *win* the arbitration in the current row and all higher rows before gaining access to the output channel (column).

The output queues buffer data from the crossbar switch for output. The queues are each of length four. While a shorter queue would suffice to decouple input and output timing, the longer queue also serves to smooth out the variation in delays due to channel conflicts.

Each output multiplexer performs arbitration and switching for the virtual channels that share a common physical channel. As shown in Fig. 12, a small self-timed state machine sequences the events of placing the data on the output pads, asserting request, and removing the output data. An interlock element is used to resolve conflicts between channels for the data pads.

To interface the on-chip equipotential region to the off-chip equipotential region that connects adjacent chips, self-timed output pads (Fig. 7.22 in [11]) are used. A Schmidt Trigger and exclusive-OR gate in each of these pads signals the state machine when the pad is finished driving the output. These completion signals are used to assure that the data pads are valid before the request is asserted and that the request is valid before the data is removed from the pads and the channel released.

6 Experimental results

The design of the TRC began in August 1985. The chip was completely designed and simulated at the transistor level before any layout was performed. The circuit design was described using CNTK, a language embedded in C [3], and was simulated using MOSSIM [1]. A subtle error in the self-timed controllers was discovered at the circuit level before any time-consuming layout was peformed. Once the circuit design was verified, the TRC was laid out in the new MOSIS scalable CMOS technology [17] using the Magic system [10]. A second circuit description was generated from the artwork and six layout errors were discovered by simulation of the extracted circuit. The verified layout was submitted to MOSIS for fabrication in September 1985.

The first batch of chips was completed the first week of December but failed to function because of fabrication errors. A second run of chips (same

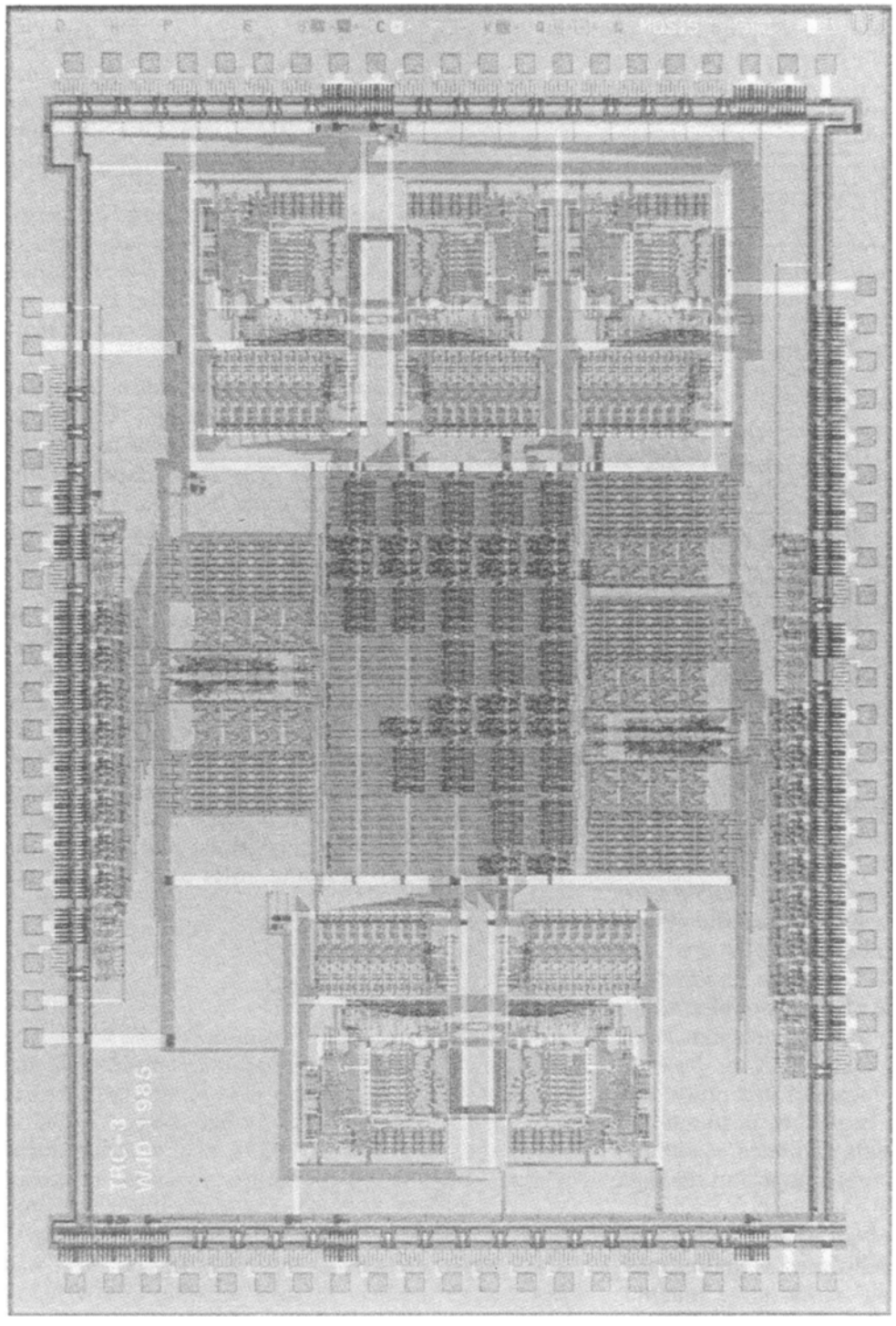

Fig. 13. The torus routing chip

design), returned the second week of December, contained some fully functional chips.

Performance measurements on the chips are shown in Fig. 14. To measure the maximum channel rate, the output request and acknowledge lines were tied together, and the input acknowledge was inverted and fed back into input request. In this configuration the chip runs at a maximum speed, shown in Fig. 14A, of ≈ 4 MHz. This sluggish performance, about one fifth of what we expected, was

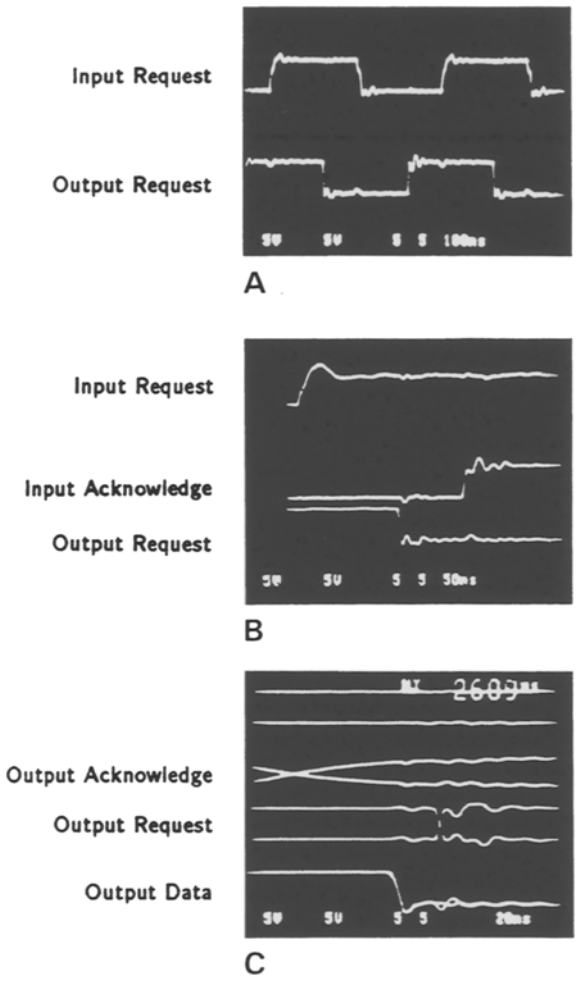

Fig. 14A–C. TRC performance measurements

traced to an overlooked critical path in the input controller. The chip still functioned correctly thanks to the self-timing.

The delays from input request to output request and input acknowledge, shown in Fig. 14B, are 150 ns and 250 ns respectively. Data propagation time from input to output (not shown) was measured to be 60 ns for both rising and falling edges. Thus data is set up 90 ns ahead of the output request. Data hold time, shown in Fig. 14C, is 20 ns.

Tau model calculations suggest that a redesigned TRC should operate at 20 MHz and have an input to output delay of 50 ns. The redesign will involve decoupling the timing of the input controller by placing single stage queues between the input pads and input controller and between the input controller and the crossbar switch. The input controller will be modified to speed up critical paths.

7 Conclusion

This work was motivated by the ongoing design and implementation of experimental concurrent computers at Caltech and the investigation [15] of interconnection networks for these machines. A strong argument for a binary n-cube interconnection was the existence of the e-cube algorithm [9] for deadlock-free packet routing. Until the development of virtual channels, we knew of no comparable algorithm for cubes of higher arity. The TRC demonstrates the use of virtual channels to provide deadlock-free packet routing in k-ary n-cube multiprocessor communication networks.

Communication between nodes of a message-passing concurrent computer need not be slower than the communication between the processor and memory of a conventional sequential computer. By using byte-wide datapaths and cut-through routing, the TRC provides node-to-node communication times that approach main memory access times of sequential computers. Communications across the diameter of a network, however, require substantially longer than a memory access time.

In spite of our past success in building machines using binary n-cube interconnection networks, there are some compelling reasons to experiment with machines using a torus network. First, the torus is easier to wire. Any network topology must be embedded in the plane for implementation. The torus maps naturally into the plane with all wires the same length; the cube maps into the plane in a less uniform way. Second, the torus more evenly distributes load to communication channels. When a cube is embedded in the plane, a saturated communciation channel may run parallel to an idle channel. In the torus, by grouping these channels together to make fewer but higher bandwidth channels, the saturated channel can use all of the idle channel's capacity.

Compare, for example, a 256-node binary 8-cube with a 256-node 16-ary 2-cube (16×16 torus) constructed with the same bisection width. If the 8-cube uses single bit communication channels, 256 wires wil pass through a bisection of the cube, 128 in each direction. Thus, with the same amount of wire we can construct a torus with 8-bit wide communication channels. Assuming the channels operate at the same rate[1], by choosing the torus network we trade a 4-fold increase in diameter (from 8 to 32) for a 8-fold increase in channel throughput. In general, for a $N=2^n$ node computer we trade

1 This assumption favors the cube since some of its channels are quite long while the torus channels are uniformly short

a $\frac{2\sqrt{N}}{n}$ increase in diameter for a $\frac{\sqrt{N}}{2}$ increase in channel throughput.

We plan to use the TRC and its successors in future experimental concurrent computers. Our first machine will use the TRC along with commercial microprocessors and memory parts to construct a 2-dimensional torus of several hundred processors. In 3 μm scalable CMOS technology the TRC measures 4.5 mm × 6.5 mm with pads. After scaling to 1.6 μm technology there will room on a single die to combine both the TRC and a simple processor. With further scaling some of the processor's local memory may be moved on-chip.

The TRC serves as still another counterexample to the myth that self-timed systems are more complex than synchronous systems. The design of the TRC is not significantly more complex than a synchronous design that performs the same function. As for speed, the TRC will certainly be faster than a synchronous chip since each chip can operate at its full speed with no danger of timing errors. A synchronous chip is generally operated at a slower speed that reflects the timing of a worst-case chip and adds a timing margin.

The real challenge in concurrent computing is software. The development of concurrent software is strongly influenced by available concurrent hardware. We hope that by providing machines with higher performance internode communication we will encourage concurrency to be exploited at a finer grain size in both system and application software.

Acknowledgements. The authors thank Craig Steele, Don Speck and Bill Athas for their many helpful suggestions, Fritz Nordby for designing the pads used on the TRC, and Keith Solomon for his work on the layout of the input controller datapath.

References

1. Bryant R, Schuster M, Whiting D (1983) MOSSIM II: A Switch-Level Simulator for MOS LSI User's manual Caltech Tech Rep 5033:TR:82 (January)
2. Dally WJ, Seitz CL (1986) Deadlock-free message routing in multiprocessor interconnection networks. Dept Comput Sci, California Institute of Technology, Tech Rep 5206: TR:86
3. Dally WJ: CNTK: An embedded language for circuit description. Dept Comput Sci, California Institute of Technology, Display File (in preparation)
4. Fisher AL, Kung HT (1985) Synchronizing large VLSI Processor arrays. IEEE Trans Comp, C-34(8) 734–740
5. Gunther KD (1981) Prevention of deadlocks in packet-switched data transport systems. IEEE Trans Commun COM-29, (4) 512–524
6. Intel iPSC User's Guide (1985) Intel Document No. 175455-001 (August)
7. Kermani P, Kleinrock L (1979) Virtual cut-through: a new computer communication switching technique. Comput Networks 3:267–286
8. Kleinrock L (1976) Queuing systems; vol 2. Wiley, New York, pp 438–440
9. Lang CR (1982) The extension of object-oriented languages to a homogeneous concurrent architecture. Dept. of Computer Science, California Institute of Technology, Technical Report, 5014:TR:82 118–124
10. Ousterhout, JK et al. (1985) The magic VLSI layout system. IEEE Design and Test of Computers, 2(1), 19–30
11. Seitz, CL (1980) System Timing. In: Mead CA, Conway LA, Introduction to VLSI Systems, Addison Wesley, London Amsterdam Paris
12. Seitz CL (1984) Concurrent VLSI Architectures. IEEE Trans Comput, C-33(12), 1247–1265
13. Seitz CL (1985) The cosmic cube CACM, 28(1), 22–33
14. Seitz CL et al. (1985) The Hypercube Commun Chip. Dept. of Computer Science, California Institue of Technology, Display File 5182:DF:85
15. Steele CS (1985) Placement of communicating processes on multiprocessor networks. Dept Comput Sci, California Institute of Technology, Tech Rep 5184:TR:85
16. Tanenbaum AS (1981) Comput networks. Prentice Hall, Englwood Cliffs, NJ, pp 15–21
17. Trotter D (1985) Miss MOSIS Scalable CMOS Rules, Version 1.2