LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility

Computation Structures Group Memo 220
October 1982

Robert A. Iannucci

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0061. The author is supported by the International Business Machines Corporation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Table of Contents

List of Figures

Abstract

This paper presents the goals of, and the design for, a high-speed packet communication switch to be used to interconnect a number of microprogrammable processors for the purpose of emulating multiprocessor computer architectures. A primary application for such a facility is the emulation of the tagged-token data flow architecture.

While the principles of design for such a device are quite general, we examine a particular implementation wherein the switch is physically partitioned across n identical logic modules, with each module physically integrated with one of the microprogrammable processors. Such a strategy is shown to be economical in terms of replication cost and design effort. Further, reliability is enhanced over a non-integrated design in that there are fewer un-checked interfaces. More importantly, however, the tight coupling of a processor and a switch module is shown to be mutually beneficial; the processor is able to unload significant amounts of low-level work on the switch, while the switch is able to rely on the computing power of the processor for powerful and flexible fault recovery.

Key words and phrases: computer architecture, fault tolerance, multiprocessors, packet communication, store-and-forward networks

Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility

1. Introduction

1.1. A Multiprocessor Emulation Facility

In recent months, considerable effort has been expended in laying out a multiprocessor computer architecture emulation facility for use within the M.I.T. Laboratory for Computer Science. This was motivated primarily by the desire to construct an emulator for the tagged-token data flow machine being studied at M.I.T. by the Functional Languages and Architectures Group [ArvindK81 81, Iannucci82-1 82]. By microcoding the instruction set for the machine [ArvindI81 81] on some suitable processor, and by interconnecting a number of these processors with a packet network, it would be possible to run large data flow application programs and, hopefully, to gain some insight into the issues involved with data flow machine design.

It was recognized that, given a sufficiently powerful and flexible base, more than just this single need could be satisfied. Several multiprocessor projects are currently under consideration, and, although the ultimate goals of each are somewhat different, they all share the need to be able to *prototype* ideas quickly and inexpensively prior to committing to hardware.

It was decided to organize the facility around a number of high-speed user-microprogrammable processors and to interconnect these with a high-speed packet switching network optimized for throughput. The resulting design also happens to have very good latency characteristics for those projects in which this is a concern.

Just as the processors in the proposed machine would be used to emulate a target architecture, so the communication network would be used to emulate a target interconnection[1]. We have selected an approach which is oriented toward flexibility and reliability of the emulated interconnect; users of the facility should have the ability to postulate and to experiment with arbitrary interconnections without undue concern about the vagaries of the underlying hardware.

Several other schemes were considered, but were found to be lacking in one or more basic areas. One proposal was to construct a two-dimensional grid of Ethernets.[Metcalfe76 76]; n^2 processors would be connected using $2n$ Ethernets. Each processor in the grid would connect to exactly two Ethers and would be responsible for generating, accepting, and forwarding network traffic. The scheme, while simple to implement, suffers from high per-message overhead and low bandwidth relative to the goals we had in mind. We wanted to provide a network which could easily extend to a very large number (say, 128) of processors and to guarantee each one a conflict free bandwidth of

---

[1] It is important to separate issues of emulator interconnect design from those of the *emulated* interconnect design.

4 megabytes per second.

Other high-speed network strategies, such as the Ring Century Bus [Okuda ??], were considered but rejected on architectural grounds (too reliant on technology to make extension practical).

Our approach to the design of the proposed emulator network is based on an 8 × 8 packet communication switch module, one of which is physically and logically integrated with each of the processors. On first analysis this may seem more expensive an approach than connecting processors via some external network. In fact, externalizing the switch would require the construction of an *interface* module (providing electrical isolation, DMA interface, and error detection) as well as a *switch* module. The proposed approach, however, requires that only one logic module be designed and fabricated rather than two.

The switch is constructed to allow processors to be interconnected in a binary n-cube topology [Burton81 81, Thomas 81, Wittie81 81]. This provides a level of redundancy which can be effectively used in tolerating hard failures. The network, however, can be physically reconfigured to any topology by re-plugging of the interconnection link cables. The links themselves are designed to have the property that the major failure modes are minimized, and that the remaining modes are easily detected.

Dynamic routing of traffic is done in the switch hardware via table lookup. This simple and inexpensive mechanism allows for tremendous flexibility of the emulated network as well as for reconfiguration. The hardware itself performs the low level arbitration functions while allowing for redundant paths. It should be readily apparent that this design will allow for virtually any *emulated* topology.

Failure detection is done at several levels[2]; electrical and logical checks are performed on all inter-switch links to catch hard and transient failures. Further, all network traffic carries a hardware generated and checked error detecting code. Detected errors are most conveniently and economically handled by a microcode task on the associated processor. Clearly, designing the error-handling logic in microcode and invoking it only when necessary will result in a much simpler yet highly robust packet switch.

1.2. Goals

As stated above, the desire is to create a large, robust network by constructing a packet switching module, integrating it with exactly one microprogrammable processor, and interconnecting these processor/switch pairs. The switch must have the following properties:

1. **Reliability:** The switches, and hence the network, should be designed to detect failures

---

[2] The terms *failure*, *fault*, and *error* have very specific meanings here. The definitions used are from Leung [Leung 80]: "... A *failure* is an unexpected out-of-specification physical change in component parameters ... A *fault* is an unspecified and disruptive change of logic values caused by a failure ... An *error* is a deviation of the logic machine from its program-specified behavior into a sequence of error states due to faults ..."

(directly or indirectly), and to recover from their effects. Simple transient failures should be handled on the fly; hard failures can be handled through simple, software controlled network reconfiguration. The network should be designed with the understanding that the best way to tolerate faults is by minimizing the probability of their occurrence.

2. **Bandwidth:** The network should have enough bandwidth to handle the *peak* load that the attached processors are able to generate. Thus, the network will not become an artificial constraint to the types of experiments which are possible using the emulation facility.

3. **Total Capacity:** The network should be easily extended. Artificial constraints should be avoided.

4. **Configurability:** In addition to the ability to handle static faults, the network should be readily reconfigurable. This includes the ability to change the routing algorithm *and* the ability to partition the network into multiple, isolated sub-networks. Moreover, the sub-networks should be truly independent, implying the ability to use different routing algorithms on different sub-networks.

5. **Ease of Implementation:** The design should be of a simple, regular structure.

The remainder of this paper discusses the high-level structure of the packet switching module. First, the design approach is outlined, and some of the characteristics are reviewed. Second, the basic design is presented along with a first-level decomposition to more primitive structures. Finally, this design is compared to the Block Switch implemented at the IBM Research Center in Zurich.

2. Approach

2.1. Topology and Routing

The network, as envisioned, will allow the attachment of a large number of microprogrammable processors. The network will be made up of individual packet switching modules, one of which will be physically integrated with each processor. The attachment will provide a direct memory access path to and from the processor's main storage as well as a status and control interface to the processor proper.

The design is optimized for, but not limited to, the binary n-cube topology discussed earlier. Referring to Figure 2-1, it can be seen that for a cube of dimensionality n, a node must act like the logical equivalent of an $(n+1) \times (n+1)$ crosspoint switch. One of the input/output pairs is connected to the processor associated with the node while the other n input/output pairs connect to the nodes immediately adjacent in the cube. The Figure illustrates this with a 3-dimensional cube (8 processor nodes).

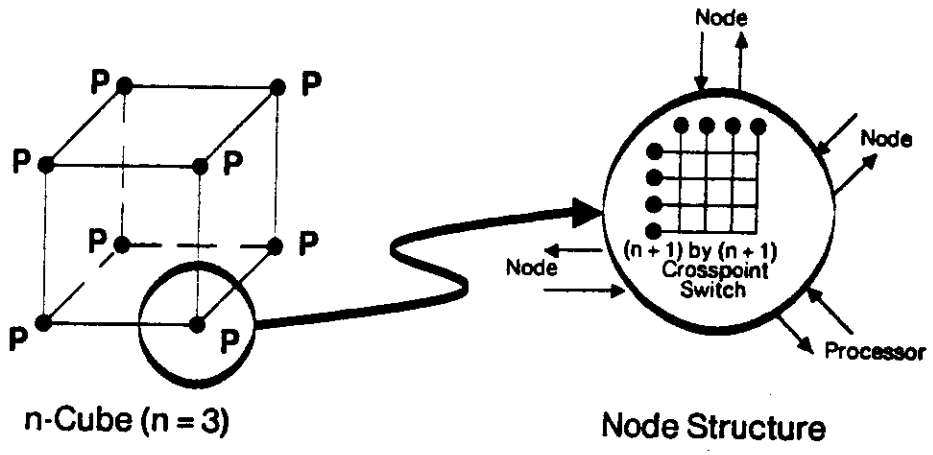

Figure 2-1: The Binary n-Cube and Node Structure

Routing of messages in the n-cube is straightforward. First, assign to each processor/switch pair (a node) a unique n-bit identifier, called its network *address*. Physically connect *adjacent* nodes. Nodes are adjacent if and only if their addresses have a Hamming distance of 1 (*i.e.*, they differ by exactly one bit). Tag each message with the address of the target node, forming a network *packet*[3].

As a packet arrives at a node, exclusive-OR the node's address with the target address carried in the packet. If the result is a vector of all ZEROs, the packet has arrived at its destination. Extract the message and pass it to the processor. If the result is not all ZEROs, then the positions in which the result of the exclusive-OR contains ONEs represent dimensions of the cube along which the packet will move closer to its destination. Select one of these and pass the packet to the next node.

Note the ability to make use of redundant paths in the network. When more than one path out of a node exists (*i.e.*, more than one ONE in the result of the exclusive-OR), the switch is free to select among these based on local traffic information, busy paths, failed paths, etc.

While this algorithm is simple, it can be made more general by replacing the exclusive-OR operation with a simple table lookup. Exclusive-OR can obviously be implemented by filling the table in properly; however, by using a table, static reconfiguration and partitioning of the network can be facilitated. Moreover, the concept of address can be generalized to allow for source-based routing. A node may have many *alias* addresses, each denoting a unique path through the network. The message source may then pre-determine the path a packet takes by suitable selection of an address.

[3] In this paper, the term *message* is used to refer to the logical object which is being transported while the term *packet* refers to the physical vehicle used. Packets may contain addresses, length fields, check bytes, and other *out-of-band* information.

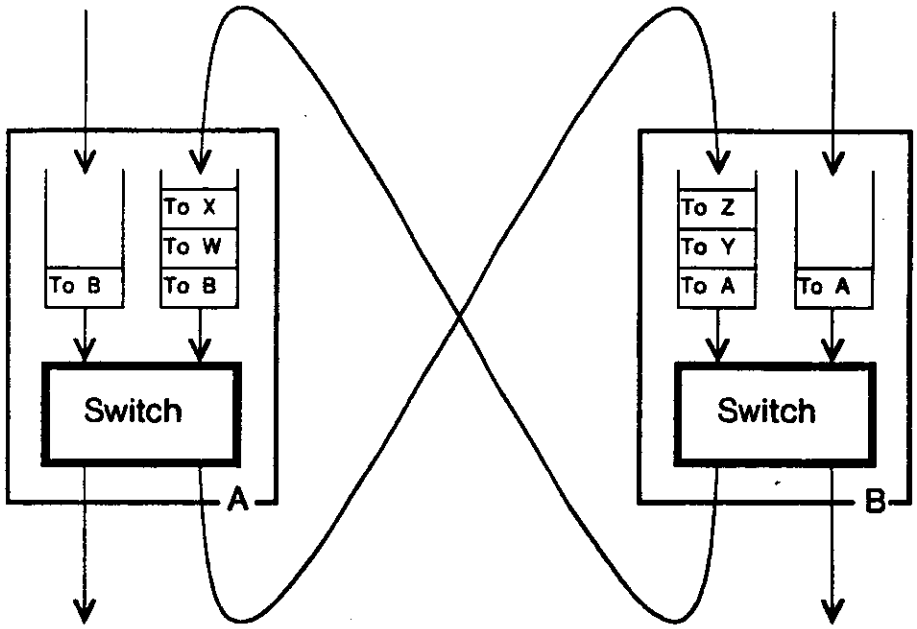

Figure 2-2: Link Deadlock Caused by a Routing Loop

While useful, the generality provided by this routing table scheme must be used with care. Without imposing some minimal constraints on routing algorithms, the proposed network can deadlock. A simple example illustrates this (refer to Figure 2-2). Imagine a subset of a large network containing two nodes, called *A* and *B*. Further, suppose that all of the routing tables in all of the switches *permit*, as one of several alternatives, A-bound traffic to pass through B, and B-bound traffic to pass through A. Clearly, then, there is a potential routing loop between switches A and B.

Routing loops can cause unnecessary network traffic, and will generally reduce the network's effective bandwidth. More troublesome, however, is the possibility that the finite-sized buffers in each switch may fill to the point that, for some pair of nodes, both wish to send to the other, but cannot themselves accept data until the other does.

This deadlock situation can be avoided by proper selection of the routing algorithm. It is easily shown that this aberrant behavior will not occur with the n-cube algorithm (or any subset of it). This is simply due to the monotonicity property - in the n-cube, a message always moves exactly one step closer to its destination on each link traversal. A routing loop would contradict this assertion.

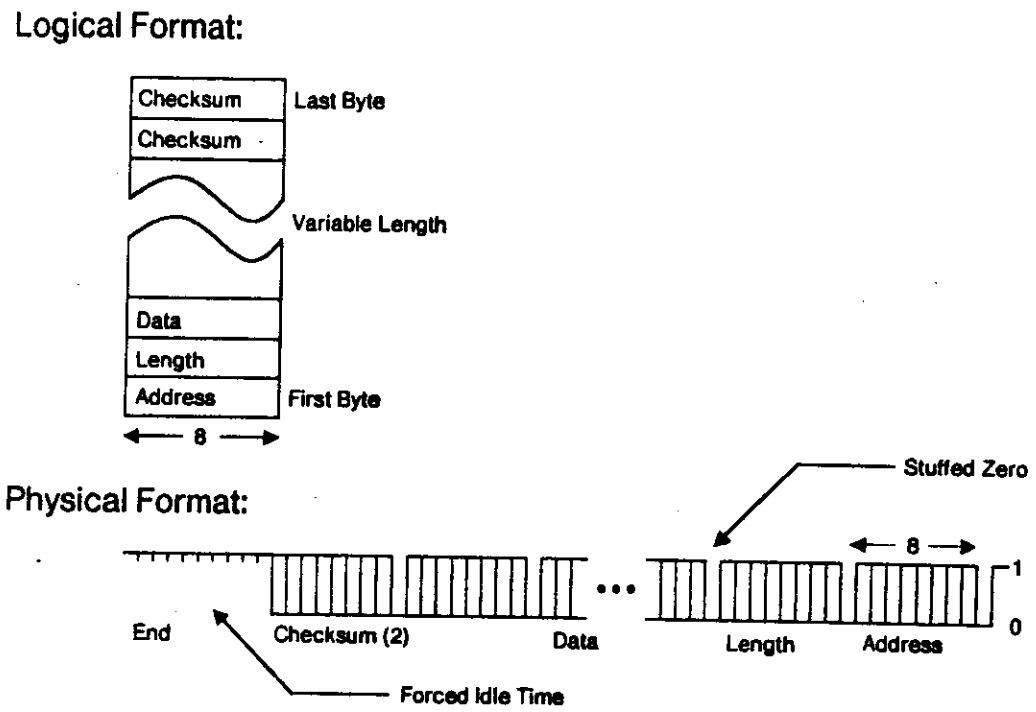

Figure 2-3: Packet Format Using Fixed Bit Stuffing

2.2. Packet Format

Packets transported by this network may vary in length. The hardware constrains the first byte of each packet to contain the address of the destination node, and the second byte to contain the message (not packet) length. The last two bytes contain a hardware generated and checked error detecting code (diagonal checksum, to be discussed later). Refer to Figure 2-3.

The design makes substantial use of overlapped operations but does not require the packets to be longer than the minimum of 5 bytes (Address + Length + Data + 2 byte diagonal checksum) to take advantage of it. Moreover, the switch contains sufficient buffering to handle packets of the maximum size with equal ease.

The links are designed to support bit- and message-level resynchronization. To this end, a fixed bit-stuffing technique is employed at each link traversal. Fixed bit-stuffing simply inserts a ZERO at regular intervals, independent of the contents of the data stream. As in normal bit-stuffing, this assures the uniqueness of the distinguished packet delimiter value. Fixed bit-stuffing suffers a statistical loss in bandwidth over normal bit-stuffing, but is somewhat easier to implement.

2.3. Flow Control

To simplify the design, it will be assumed that transmission of a packet occurs as an uninterruptible stream. Thus, it will be necessary to provide flow control information at *packet boundaries* to assure that finite-sized buffers are not overrun. This implies that the hardware must exercise flow control when it cannot be guaranteed that enough space exists to hold an entire packet at the receiving end.

Further, if a centralized clock is not used, clock skew from a fast sender to a slow receiver may introduce further complications once packet transmission begins due to the streaming assumption. At least three solutions to this problem exist: use a common clock between the sender and the receiver, artificially slow the sender relative to the receiver, or provide enough buffer space at the receiver to accommodate the maximum packet size at the worst-case skew. The latter option is preferred.

2.4. Fault Tolerance

As mentioned, the first line of defense against failures is to minimize their probability. Experience in designing hardware systems suggests a primary source of failures is in physical interconnections. By minimizing their number, the inherent reliability is increased. Further, simple hardware checks can be used to detect failures when they do occur.

The strategy here is to rely on the physical integration of the processor and the switch to insure proper delivery of data to and from the packet network. This assures the interconnection is the same order of reliability as the connection between the processor's ALU and the main memory. Once a message leaves the protective environment of the processor, extra care must be exercised.

Of concern are I/O pins, external wires, and power supply boundary crossings. *I/O pins* are inherently noisy and unreliable. Any data passing through a physical connector should have some provision for this unreliability. *External wires* bring with them the possibilities for coupled noise which must be minimized. *Power boundaries* represent a third danger. As signals cross from the domain of one power supply to another, care must be taken that the signal is still recoverable. This implies the ability to ignore signal and ground level shifts, etc.

I/O pins can be minimized by using serial, rather than parallel, data transmission. The obvious tradeoff here is that the serial signalling rate must track the product of the parallel signalling rate and the parallel path width. Checking of serial interconnections is, however, much simpler than checking parallel interconnections.

Given that serial communication is being used, it is reasonable to take great care with the physical interconnect. For example, fiber optics, shielded twisted pair, or twin-axial cable can be used to minimize coupled noise despite the additional expense over simpler schemes.

The power boundary issue can be addressed by several methods. Direct current paths between systems can be blocked by using pulse transformers or optical couplers. Alternatively, dual differential signal transmission can be used to separate out common-mode noise. The latter two

schemes can also be combined to yield excellent isolation and noise immunity characteristics.

All of the above factors were weighed in the selection of a serial link methodology. The approach used here represents a good balance between performance and cost. The salient characteristics of the chosen approach are

- **Encoding:** Manchester biphase. The idle state of a link is defined to be a steady stream of ONEs rather than a no-signal condition. Effectively, then, there is no startup delay time at the beginning of a packet necessitated by the energy storage characteristics of the cable.

- **Impedances:** All properly matched. The source, at 50Ω, drives a pulse transformer with a 1:2 impedance ratio. The cable and termination are both 100Ω.

- **Interconnect:** Shielded twisted pair.

- **Isolation:** Dual-differential transmission plus pulse transformer. Fiber optics appeared to be a good alternative as well due to the inherent noise immunity, but the state of the art in optical fiber technology cannot provide the needed bandwidth at an acceptable cost.

- **Technology:** Emitter-coupled logic line drivers and receivers.

To further harden the system against faults, the following steps will be taken:

- **Analog Domain Check:** Using receivers with a nonzero offset characteristic (see Figure 2-4), a simple continuity check can be performed at almost no expense or loss of performance. This allows for the detection of open-circuit and short-circuit conditions.

  The circuit operates as follows: for a dual-differential transmission line terminated at the receiving end, an open-circuited or short-circuited interconnecting cable will cause the voltage seen at the input to the line receivers to be very nearly zero. If the receiver has the characteristic that its output is indeterminate for some finite range of voltages around zero (shaded area in the graphs), nothing can be determined by looking at the receiver's output. If, however, a receiver with an offset characteristic is used (see Figure), this region of uncertainty does not include the range of voltages that will appear during an open/short failure. Manufacturers of line drivers/receivers recognize the desirability of this and typically make provisions for offsetting the receivers either internally or through an external high-valued resistor.

  By connecting two such receivers to the line, one in-phase and one out-of-phase, it can be seen that when the line is open or shorted, *both* receivers will produce the same valid logic level (in the case of the configuration shown, this is a logic "1"). During normal operation, this situation will only occur during periods of signal transition (input going negative to positive, or conversely). By ANDing the output of the two receivers and sampling the result *between* line transitions, it can be immediately determined when the

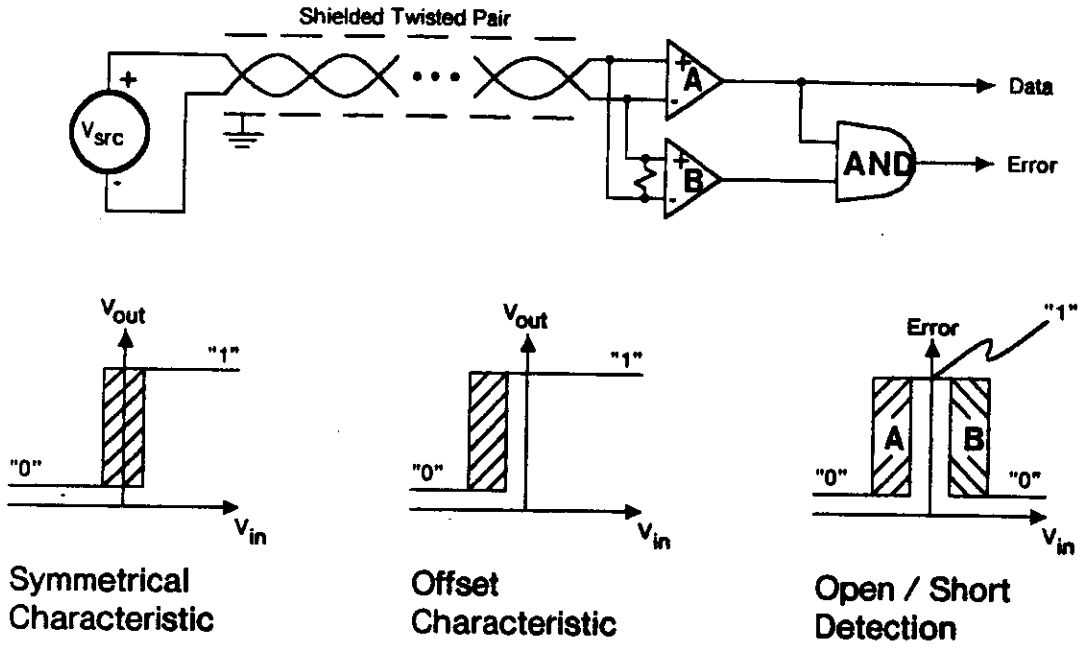

Figure 2-4: Analog Domain Check using a Nonzero Offest Characteristic

link has failed due to an open or short circuit. What is lost in doing this sort of check is that the signal voltage swings at the receiver must be *greater* than the receivers themselves would normally accept by more than a factor of two (see Figure).

- **Echo Check:** Each inter-switch path must be implemented as a bidirectional channel - it is necessary to pass flow control information from receiver to sender. Reverse-direction paths suffer the same unreliabilities as forward-direction paths, so it makes sense to use identical links in both directions. Given that such a path exists, it costs little for the receiver to echo bits as they are received back to the sender where a comparison with the transmitted data can be made.

- **Timeout/Overrun Check:** By properly delimiting packet boundaries *and* by designing the circuitry such that, once begun, a packet will be transmitted in its entirety, a simple timeout check can be performed on the data stream emerging from each serial receiver. Further, by redundantly coding packet length, it is possible to detect overrun conditions as they occur.

- **End-to-End Checksum:** A diagonal checksum is generated and checked by the processor interface logic on the switch as an ultimate end-to-end test of the proper delivery of a message.

- **Microcoded Recovery:** By designing the switch with a rich family of self-checks, the

possibility exists to recover from errors rather than just giving up the computation. This usually incurs significant expense; the control logic for recovery is usually quite complex. However, the presence of an integrated microprogrammable processor makes the task economically viable. By dedicating a microcode task to switch monitoring and error recovery, and by only invoking this task when a failure is detected, no main-line performance is degraded nor is the switch design made significantly more complex.

Error recovery procedures are beyond the scope and intent of this paper. Typically, however, the recovery algorithms involve enabling of the error-handling microtask, interrogation of the switch state vector, logging of the event, case analysis to determine the cause of the failure and the appropriate continuation, coordination with other nodes in the network, restoration of a consistent state, and resumption of normal network operation.

The next section discusses the design of a packet switch which embodies these principles.

3. Design

3.1. Basic Structure

The overall structure of the switch is shown in Figure 2-5. It is modular in character, and is organized around eight major control and data buses. Connected to the buses are eight input FIFOs, eight output buffers, and a Sequencer / Scheduler. Seven of the input FIFOs and seven of the output buffers connect to serial ↔ parallel conversion logic. The eighth input / output pair interfaces to the data bus of the attached microprogrammable processor. Internally, the switch is totally synchronous. Resynchronization of incoming data with the local clock is done by the serial-to-parallel conversion logic.

Control and status lines (not shown) also interface to the processor. These serve the purposes of initialization, error reporting, and debugging. *Initialization* involves the establishment of the starting state of the switch. Loading of the routing table is under the control of the processor. *Error reporting* represents the maintaining of a state vector, containing error-capture latches. A summary error signal is generated and used to request processor service when necessary. *Debugging* is aided by making much of the switch's state is both visible and modifiable from the processor.

The eight physical buses partition into two logical buses - one associated with the allocation and deallocation of output ports (*i.e.*, the routing and arbitration functions), and one associated with the transfer of data from input to output. The former is under control of a Scheduler and the latter under control of a simple Sequencer (both described in section 3.6).

Bus timing derives directly from the Sequencer and the Scheduler. The Sequencer defines the shortest time quantum, called a *minor cycle*. Eight minor cycles make up a *major cycle*[4]. During a

[4] At a per-link bandwidth of 4 megabytes per second, a minor cycle will be 31.25 nsec., and a major cycle will be 250 nsec.

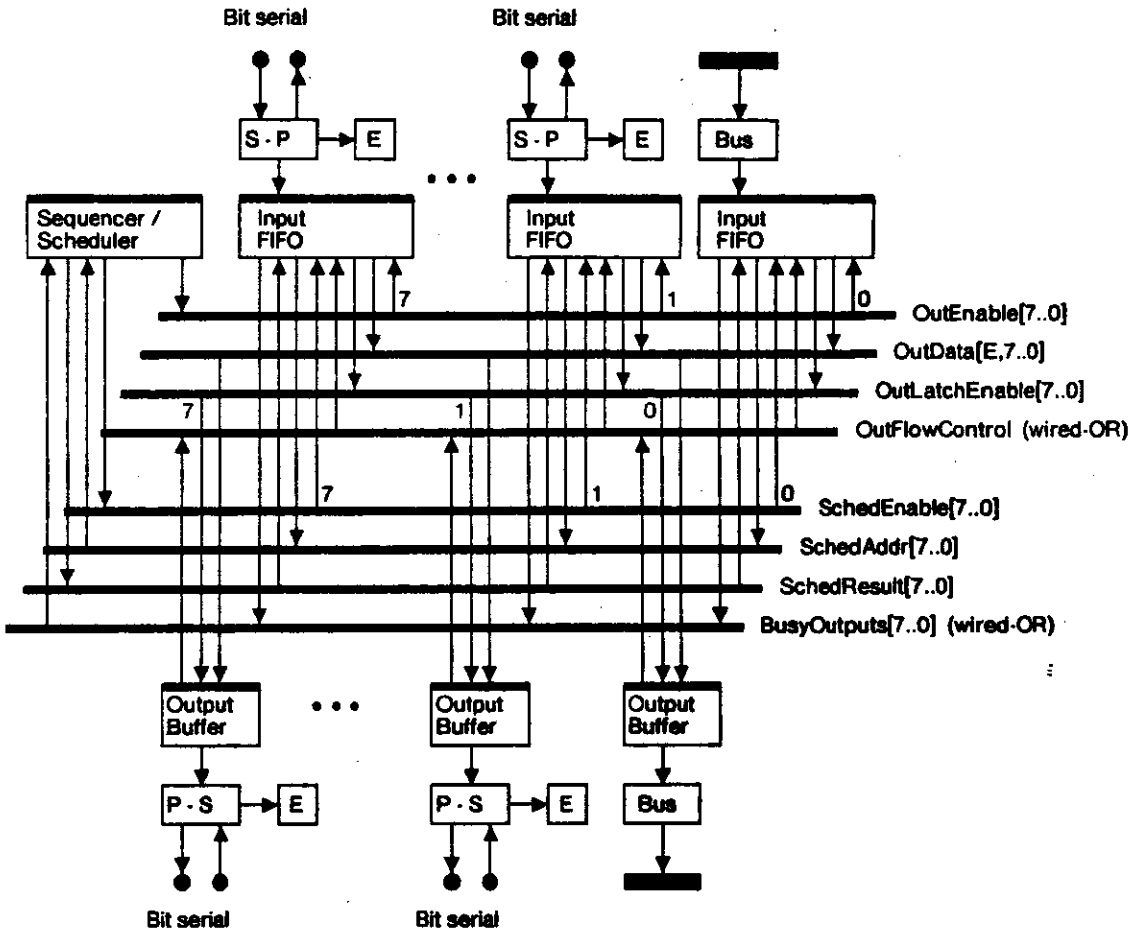

Figure 2-5: Basic Structure of the Packet Switch

minor cycle, one byte can be transferred between one input FIFO and one output buffer.

The Sequencer does nothing but issue permission for each input FIFO to transmit a byte on the bus in simple round-robin fashion. Round robin service supports mixed-size packets more fairly than would a scheme based on block transmission of an entire packet. Moreover, the interleaving of transfers allows the use of multiple, slow RAMs for buffering without unnecessarily slowing down the bus.

The Scheduler, being more complex, has a period which is three minor cycles in length. It likewise polls the input FIFOs in round-robin fashion, looking for one which is requesting allocation of an output port for a given packet.

Briefly, the eight buses shown in the figure are defined as follows:

- **OutEnable[7..0]:** A non-shared bus made up of eight discrete enable lines, exactly one

of which is active at any given time. These signals are used to give each input FIFO access to the output data bus in round-robin fashion.

- **OutData[E,7..0]:** A shared bus used to transmit data from input FIFOs to output buffers. Access is controlled by OutEnable[7..0]. Bits 7..0 contain data; E is an out-of-band bit which is used to mark the last byte in a packet. The E bit is regenerated at every link traversal and is used in conjunction with the timeout/overrun check.

- **OutLatchEnable[7..0]:** A non-shared bus wherein *at most* one signal is active. This bus is used to signal an output buffer to accept the data on the OutData[E,7..0] bus.

- **OutFlowControl:** A wired-OR signal driven by the selected output buffer. Assertion of this signal indicates that the output buffer is full and that the word being presented on the OutData[E,7..0] bus *has not* been accepted.

- **SchedEnable[7..0]:** A non-shared bus of eight lines, not unlike OutEnable[7..0] in purpose. Exactly one of the eight lines is asserted at any one time, and indicates which of the eight input FIFOs is being polled for a scheduler request.

- **SchedAddress[7..0]:** A shared bus, access to which is controlled by SchedEnable[7..0]. The selected input FIFO will present to this bus the waiting packet's target address.

- **SchedResult[7..0]:** A single-source bus carrying the result of a scheduler cycle. At most one bit of this bus will be at a logic ONE. In the event of an unsuccessful scheduler cycle (*i.e.*, unable to allocate an output), the bus will contain all ZEROs.

- **BusyOutputs[7..0]:** A wired-OR bus which always contains the vector of output buffers which have been allocated.

3.2. Inter-Switch Link

Delivery of error-free bytes from the output buffer of one switch to the input FIFO of another switch is the responsibility to the inter-switch link. It implements the parallel ↔ serial conversion, electrical checking, timeout/overrun checking, and echo checking discussed previously. Its interface is a byte-wide data path and a full-handshake protocol. Refer to Figure 3-1.

Outgoing data are latched in a holding register and transferred to two parallel to serial shift registers. One of the registers delivers bits to an encoding network (Manchester Biphase), and is clocked at a rate of one bit every minor cycle. An ECL dual-differential driver transmits the data over a shielded twisted pair to the receiving switch (*i.e.*, the actual inter-processor path). A logic ZERO precedes every byte as it leaves the switch (fixed bit-stuffing).

At the receiving end, the clock and data are separated. Data are shifted into a serial to parallel register and are simultaneously echoed back to the sender. The echo path is identical in every respect (save the direction of transmission) to the forward path. Stuffed ZEROs are checked and deleted from the recovered stream but are left in the echoed stream.

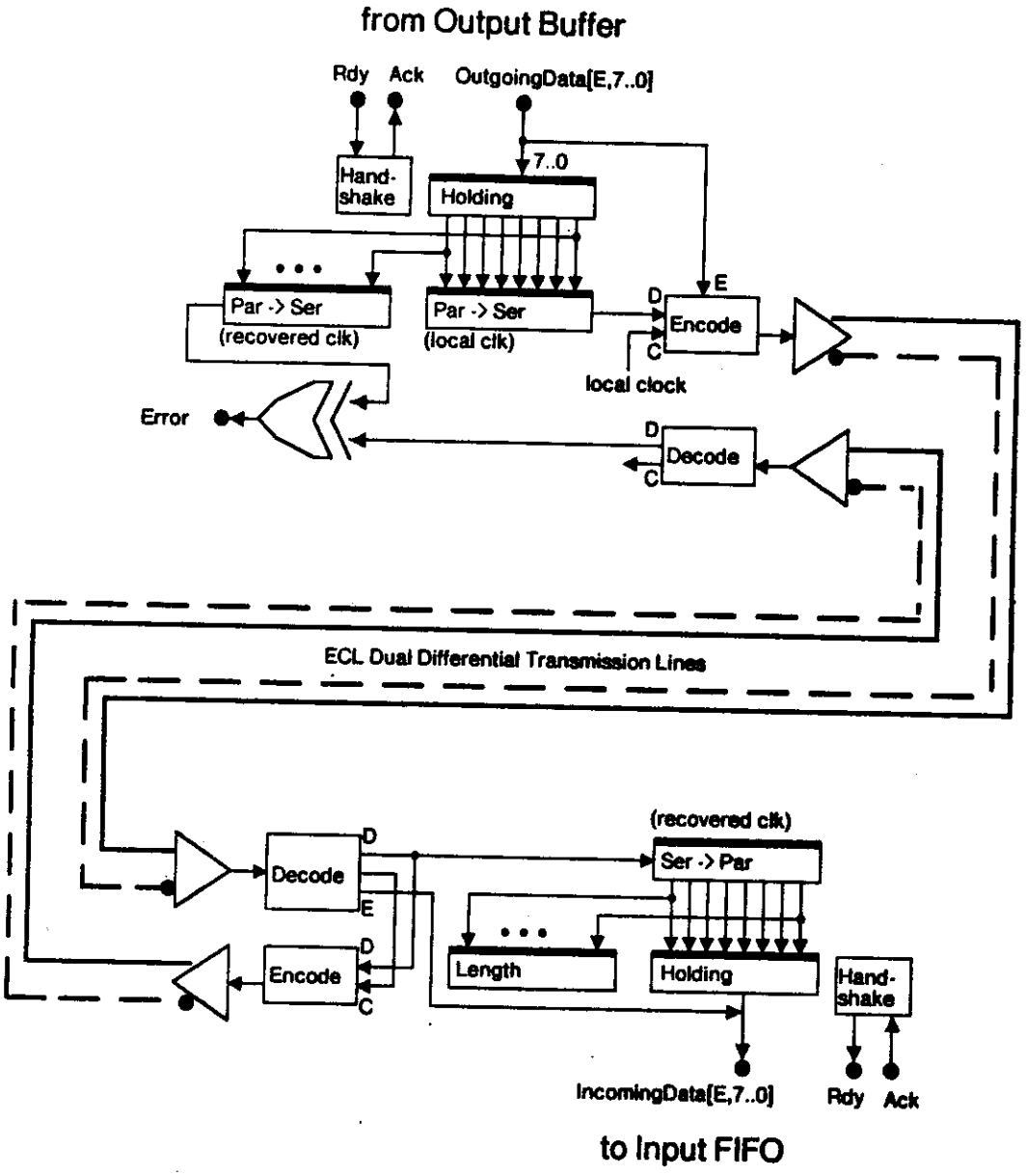

Figure 3-1: Organization of an Inter-Switch Link

Back at the sending end, the echoed data are recovered and are bit-for-bit compared with what was originally transmitted. Any miscompare is reported to the processor (which is responsible for recovering from the failure). The holding registers at either end of the connection are used as a staging area for the parallel ↔ serial conversions.

Flow control is implemented at the receiving end by *not* echoing data at the beginning of a

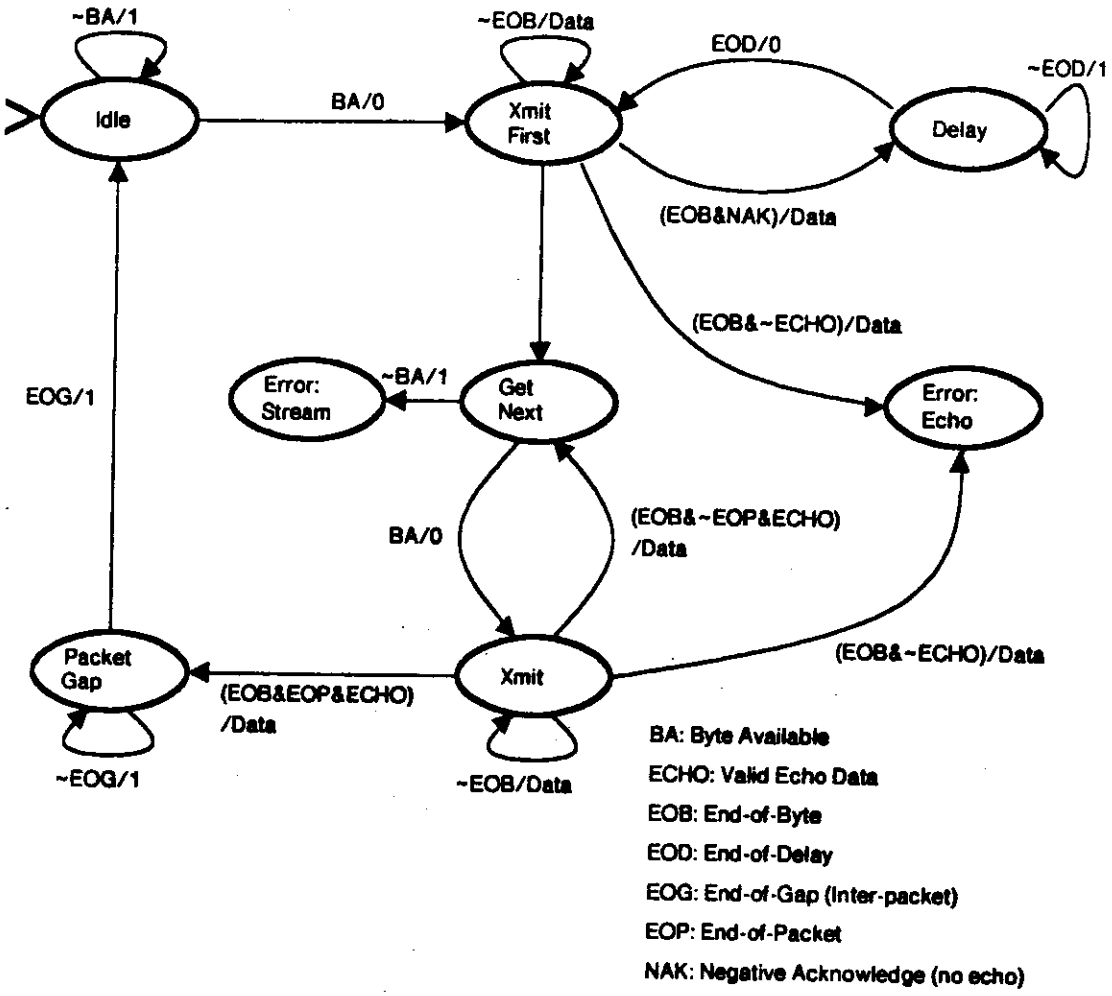

Figure 3-2: Link Transmitter State Diagram

packet when there is insufficient space in the FIFO to contain the largest possible packet. The sending end is then responsible for idling the link and then re-attempting transmission. The sending end's state diagram is shown in Figure 3-2. No flow control information is transmitted once packet streaming has begun successfully.

As the receiver accepts packet bytes, it extracts the length field and uses it to check the actual packet size. End-of-packet is signalled on the link as shown in the state diagram (*Packet Gap*).

Several issues arise in the design of the link hardware. One very important consideration is the effect of *round-trip delay*. It is not unreasonable to expect echoed data to lag transmission by six bit times (minor cycles) with a typical 10 meter interconnecting cable. Also, in developing state transition signals (*e.g.*, NAK), it is important to check that an analog domain failure *has not* occurred. Furthermore, a clear tradeoff exists in the selection of a maximum practical packet size. As the size increases, so does the chance of a transient fault. As the size decreases, so does the

transmission efficiency.

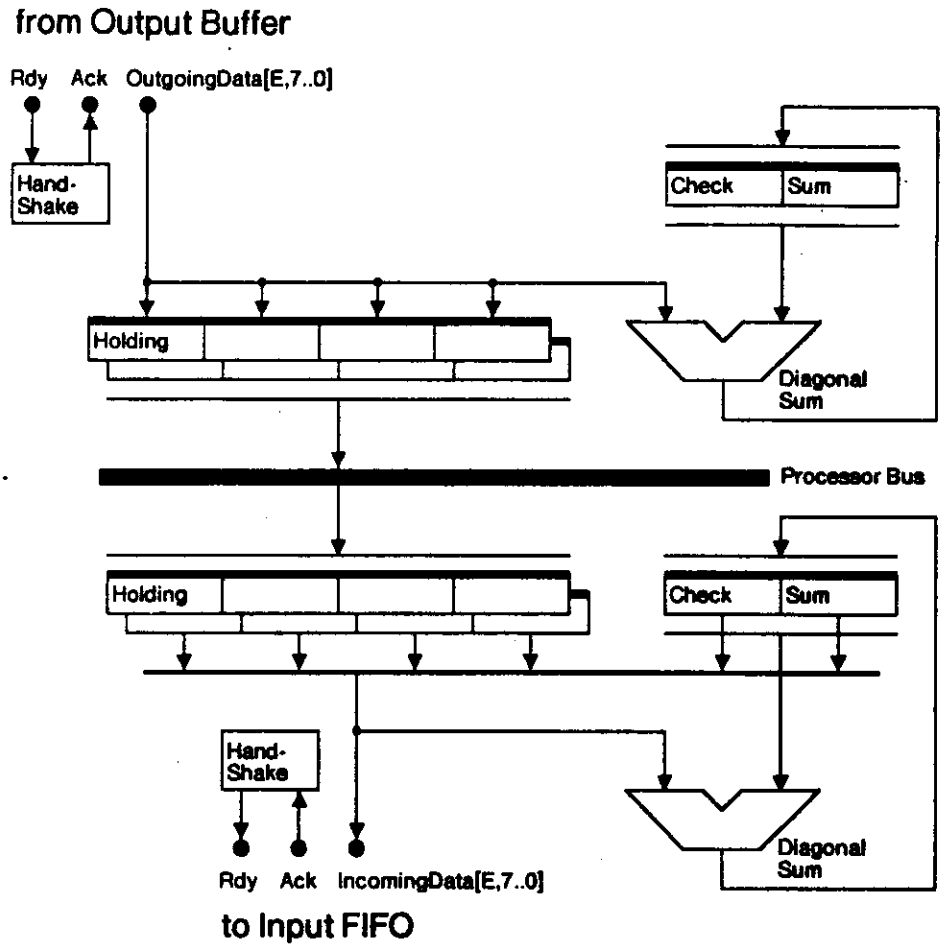

Figure 3-3: Local Processor Bus Interface

3.3. Processor Bus Interface

At the point of origin of a message, circuitry must be provided to reformat the data from the local processor's bus to the format used internally in the switch. Two basic functions are involved: converting bus words into an 8 bit wide stream, and computing a diagonal checksum. Referring to the bottom half of Figure 3-3, it can be seen that these operations are performed by a double-buffered holding register (to allow one processor word to be picked apart while another is being prefetched), a six input multiplexer, and a simple adder for computing the checksum.

At the message's destination, the data must be re-converted to the width of the processor bus, and the diagonal checksum must be verified. The top half of Figure 3-3 shows the circuitry. Again, a double-buffered bus interface is used.

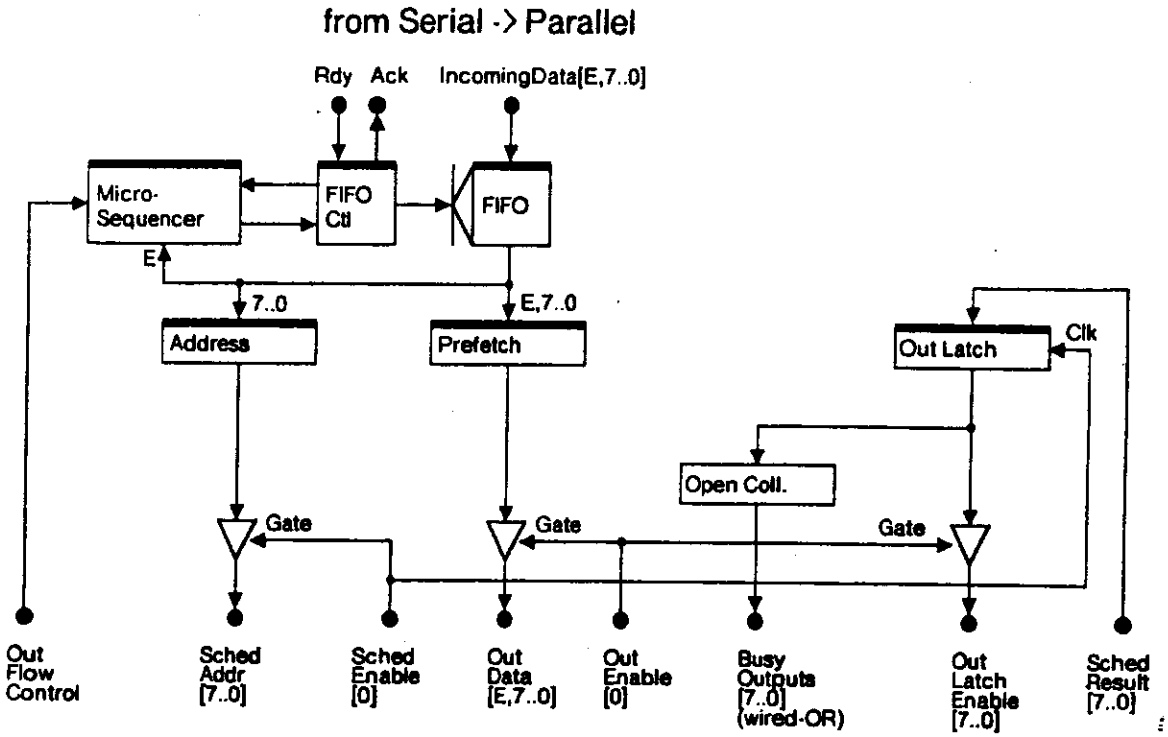

Figure 3-4: Input FIFO and Control Logic

3.4. Input FIFO and Control

Referring to Figure 3-4, it is seen that the input FIFO is responsible for the following:

- Queueing of input bytes from the link's serial to parallel converter.
- Interpreting packet contents to extract Address and Length information.
- Prefetching packet bytes prior to transmission over the internal data bus.
- Communicating with the Scheduler subsystem.

All of the sequencing of the input FIFO is handled by a simple ROM-based state machine [Iannucci 81]. The FIFO is managed by a single-chip FIFO controller (*e.g.*, the Signetics 8X60). The hardware need only respond to incoming data (at most one byte every major cycle), output requests (one every major cycle), and scheduler polls (one every three major cycles).

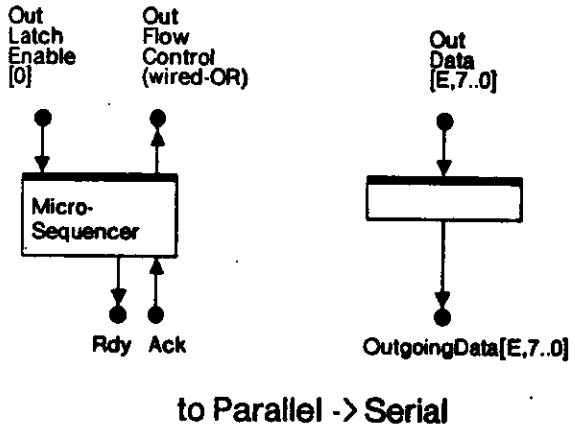

Figure 3-5: Output Buffer and Control Logic

3.5. Output Buffer and Control

The functions performed by the output buffer (Figure 3-5) are a simple subset of those performed by the input FIFO. Data are delivered by the OutData[E,7..0] bus, and triggering of an output cycle is via the OutLatchEnable bus. Another ROM-based sequencer controls the buffer and provides the backward flow control information. Despite the shortness of the output cycle, recall that no output port will ever be expected to accept bytes at a rate faster than one every major cycle.

3.6. Sequencer / Scheduler

At the heart of the packet communication switch are two simple but essential subsystems: an output sequencer and a scheduler (arbiter). As described, each has associated with it parts of the central bus complex.

The sequencer is responsible for polling input FIFOs in strict round-robin order at a rate of one every minor cycle. This is done (see Figure 3-6) by a simple modulo-8 counter and a 3 to 8 decoder which drives the OutEnable[7..0] bus. This results in an input polling order of $(01234567)^*$.

The scheduler is the more complex of the two subsystems (refer to Figure 3-7). It operates by polling input FIFOs for scheduling requests at a rate of one every three minor cycles. This is done by latching the value of the output sequencer's counter on every third increment, resulting on a polling order of $(03614725)^*$. During a scheduler cycle, the following actions occur:

1. The three bit identifier of the selected input FIFO is latched.

2. The identifier is decoded (3 to 8), and drives the SchedEnable[7..0] bus.

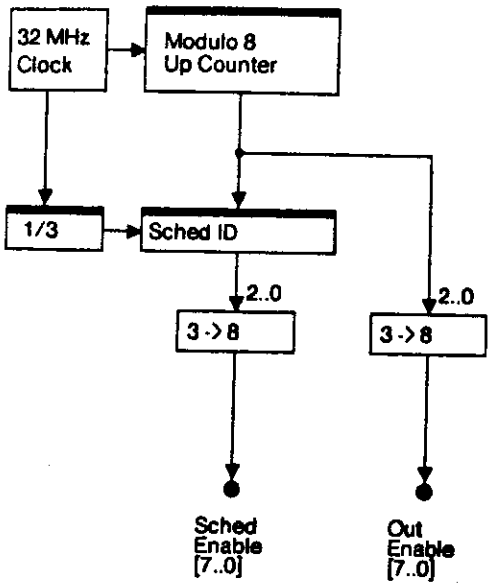

Figure 3-6: Sequencer

3. The selected input FIFO responds with the target address for the waiting packet, if there is one. The address is combined with the three bit identifier of the selected FIFO to index into the Routing Table. The result is the *vector* of legal output candidates.

4. At all times, input FIFOs assert their claim to previously allocated output buffers by signalling on the appropriate wire of the BusyOutputs[7..0] bus.

5. By computing the bit-for-bit AND of the candidates vector and the complement of BusyOutputs[7..0], an eight bit *requested but not allocated* vector is generated.

6. If the vector contains only ZEROs, the scheduler cycle has failed. The SchedResult[7..0] bus reflects this fact by having no bits at a logic ONE.

7. If the vector is not all ZEROs, one of the ports corresponding to a ONE in the vector may be allocated to the input FIFO. To assure an unbiased distribution of traffic when multiple paths are available, some sort of fair-scheduler must be used. Three approaches were considered:

   a. For each input FIFO, maintain a modulo-8 counter (implemented with MSI or RAM). Each time a successful schedule cycle occurs, use the value of the counter to *rotate* the eight bit result vector. From the rotated result, select the left-most bit. Compute the index of this bit, and subtract from this the amount by which the vector was shifted. Decode this into a one-of-eight form.

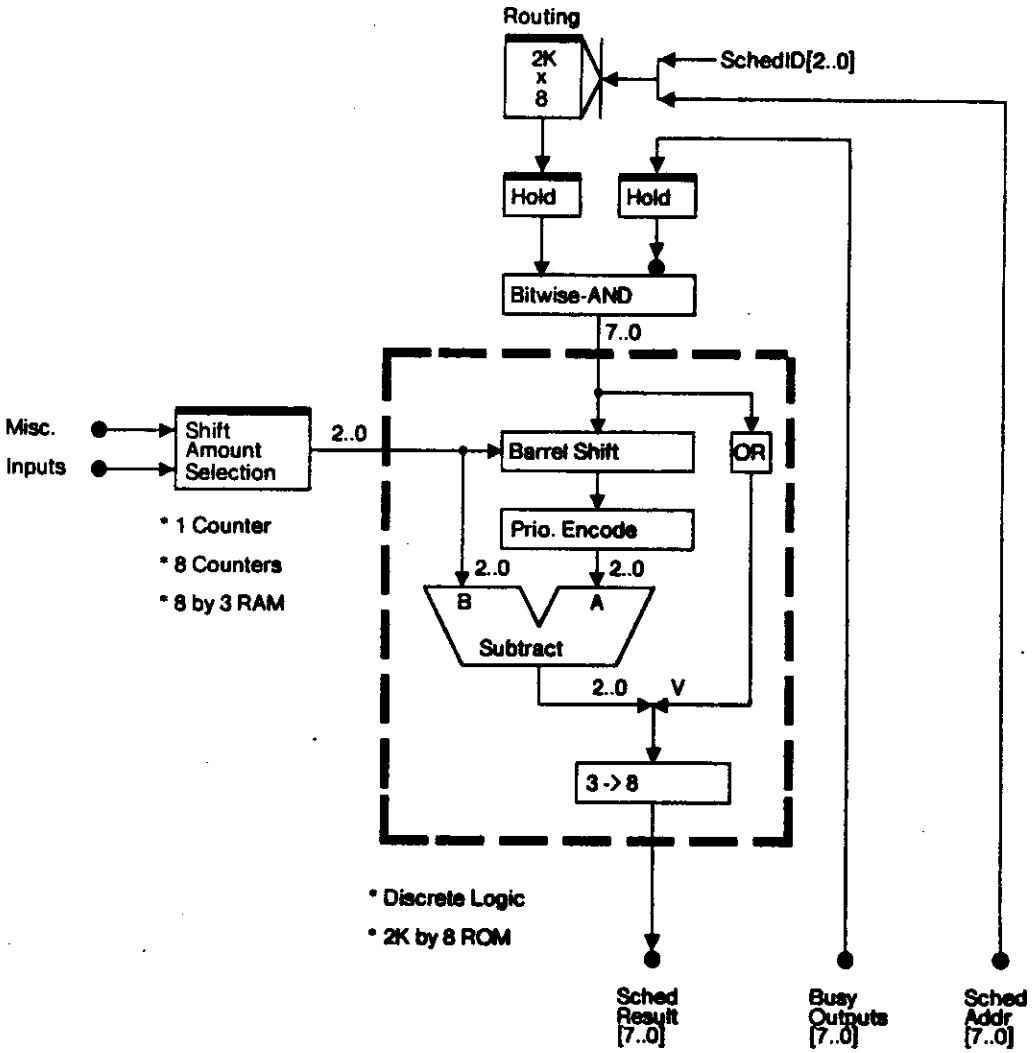

Figure 3-7: Scheduler

b. Use only one counter for all eight FIFOs. Given a random distribution of packet arrivals, this should tend to randomize the selections across multiple outputs. Rotate, encode, shift and decode as above.

c. Recognize that the previous scheme can be implemented with shifters, encoders, decoders, and subtracters *or* with a simple 2K × 8 ROM (the area enclosed by the dashed box in the Figure).

The result will be a very fast and simple hardware structure with good, though not ideal, packet distribution characteristics. The careful reader will have observed a subtle flaw in all of the above encoding algorithms. In any scheme that uses a priority encoder "randomized" by a simple rotation

of the input (as do *all* of the above, MSI or ROM based), the result *will* show a bias for certain result values in certain cases. An example illustrates this:

> Assume that the bitwise-AND has produced the vector 00000011 (*i.e.*, ports 0 and 1 are both acceptable). Assume also that the hardware works as described above - the vector is rotated by some random amount and then left-to-right priority encoded. For rotation amounts 0,1,2,....6, the result of the priority encode will be the selection of port 1. Only in the case of a rotation amount of 7 will the encoder choose port 0. Thus, if the rotation amounts are uniformly distributed, the probability of choosing port 1 will be 7/8 while the probability of choosing port 0 will be 1/8.

A similar problem was described in [Iannucci82-2 82]. The solution is to *assure* that the rotation amounts are correlated with the previous scheduling decision. Specifically, the shift amount should be chosen so as to give *least* preference to the port chosen during the last cycle. This can be done by using a RAM to store the value of the last output buffer selected.

4. State of the Art

After the completion of the design as presented in this paper, the author learned of a strikingly similar design done at the IBM Zurich Research Laboratory and reported in [Bux82 82, Rothauser81 81]. Their design of a *block switch* (a shared-bus, high-speed, packet switch) is being used to interface local communication networks with each other. It differs from the proposed design in several ways:

- **Routing:** The IBM design relies on end-to-end (source based) routing decisions. No routing based on local traffic congestion can be made. Moreover, the application domain is much more static in nature - a local network is only likely to be reconfigured at long intervals. Hence, no effort was expended to provide generality of reconfiguration.

- **Fault Tolerance:** The IBM design does little to assure the proper delivery of messages and relies, again, on end-to-end methods.

- **Application Assumptions:** Where the proposed system is designed to handle sustained traffic, the IBM block switch is optimized for *bursty* (low duty cycle) traffic. Further, the design presented here makes synergistic use of the associated processors in eliminating much of the cost associated with some fairly elaborate features (*viz.*, debug aids, fault recovery, initialization) where the IBM switch is designed to stand alone.

5. Conclusion

We have examined the principles behind, and the design of a high-speed, robust packet switch which is capable of interconnecting a large number of microprogrammed processors. The design has several interesting properties which derive from the close relationship that the switch shares

with its attached processor. We briefly review the goals:

1. **Reliability:** This has been addressed on several levels. First and foremost, the predominant failures have been minimized. I/O related failures have been reduced by using a serial, rather than parallel, data transmission scheme. Coupled noise and common-mode noise problems have been alleviated by the use of shielded twisted pair cable and dual-differential signalling. Transient and hard failures are detected on the wire interfaces, and rely on a microcode-based recovery mechanism. An end-to-end checksum is provided to cover the unchecked data paths internal to each switch.

2. **Bandwidth:** The scheme presented delivers a raw bandwidth on the order of 4 megabytes per second per link. This is more than adequate considering the processors to which the network will be attached.

3. **Total Capacity:** The design of the switch can accommodate 128 processors interconnected in a binary n-cube. Other topologies will allow more processors (*e.g.*, cube-connected cycles [Preparata81 81]). The addressing scheme (part of the packet format) allows for up to 256 unique target addresses.

4. **Configurability:** The topology is physically alterable (through re-plugging of link cables); more importantly, the routing is completely flexible due to a writable routing table per input port per switch.

5. **Ease of Implementation:** The switch is highly regular in structure and relies on standard techniques for sequencing and data path control.

6. Acknowledgements

The author gratefully acknowledges the suggestions of Dr. Robert E. Thomas. Significant among his contributions was the recommendation to design for the n-cube topology. In addition, Dr. Thomas did an extensive survey of manufacturer's literature which led, ultimately, to the selection of single-chip FIFO controllers as a fundamental system building block, and to the rejection of fiber optics as a practical and economically viable communication link technology.

I would also like to thank Dr. David D. Clark, Prof. David K. Gifford, and Prof. Jerome H. Saltzer of the Computer Systems and Communications group for valuable discussions related to link technologies and for bringing the IBM block switch design to my attention. Further, I wish to acknowledge the advice and sound judgement of Prof. Robert H. Halstead, Prof. Richard E. Zippel, William B. Ackerman, and G. Andrew Boughton, who all participated in the design of and planning for the emulation facility.

References

[Arvindl81 81]
Arvind, and R. A. Iannucci.
Instruction Set Definition for a Tagged-Token Data Flow Machine.
Memo 212, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., December, 1981.
revised February, 1983

[ArvindK81 81]
Arvind, and V. Kathail.
A Multiple Processor Dataflow Machine That Supports Generalized Procedures.
In *The 8th Annual Symposium on Computer Architecture*, pages 291-302. May, 1981.

[Burton81 81]
Burton, F. W., and M. R. Sleep.
Executing Functional Programs on a Virtual Tree of Processors.
In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187-194. October, 1981.

[Bux82 82]
Bux, W., *et. al.*.
A Local-Area Communication Network Based on a Reliable Token-Ring System.
In *Local Computer Networks*, pages 69-82. North-Holland Publishing Company, 1982.

[Iannucci 81]
Iannucci, R. A.
Attached Data Flow Vector Processor.
Master's thesis, Department of Electrical and Computer Engineering, Syracuse University, Syracuse, NY, August, 1981.

[Iannucci82-1 82]
Iannucci, R. A.
Implemetation Strategies for a Tagged-Token Data Flow Machine.
Memo 218, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1982.

[Iannucci82-2 82]
Iannucci, R. A.
Single-Chip Microtask Scheduler for a Data Flow Micromachine.
Design Note 1, Functional Languages and Architectures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1982.

[Leung 80]
Leung, C. K. C.
Fault Tolerance in Packet Communication Computer Architectures.
Technical Report TR-250, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

[Metcalfe76 76]
Metcalfe, R. M., and D. R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM 19(7):pp. 395-404, July, 1976.

[Okuda ??]
Okuda, N., T. Kunikyo, and T. Kaji.
Ring Century Bus - an Experimental High Speed Channel for Computer Communications.
Toshiba Research and Development Center, Tokyo Shibaura Electric Co., Ltd., Tokyo, Japan.

[Preparata81 81]
Preparata, F. P., and J. Vuillemin.
The Cube-Connected Cycles: a Versatile Network for Parallel Computation.
Communications of the ACM 24(5):pp. 300-309, May, 1981.

[Rothauser81 81]
Rothauser, E. H., P. A. Janson, and H. R. Mueller.
Meshed-Star Networks for Local Communication Systems.
In *Local Networks for Computer Communications*, pages 25-41. North-Holland Publishing Company, 1981.

[Thomas 81]
Thomas, R. E.
A Dataflow Architecture with Improved Asymptotic Performance.
PhD thesis, Department of Information and Computer Science, University of California, Irvine, California, April, 1981.

[Wittie81 81]
Wittie, L. D.
Communication Structures for Large Networks of Microcomputers.
IEEE Transactions on Computers C-30(4):pp. 264-273, April, 1981.