A High-Speed Channel Controller for the Chaos Router

An Independent Master's Project by Robert Wille
December 8, 1992

1 Introduction

The chaos network router is a high-performance chip used for interprocessor communication. Every processor has a router that is used for communicating with other processors. When a processor needs to send data to another, it breaks it up into messages and injects them into its router. The router will then send the messages to other routers. The messages flow through various routers until they reach their destination, where they are ejected to the target processor. It is the routers' responsibility to ensure that messages are delivered to their target processors. The chaos router creates a network with a mesh topology. The edges of the mesh are wrapped around to form a torus. Figure 1 shows part of a network with a sample message route.

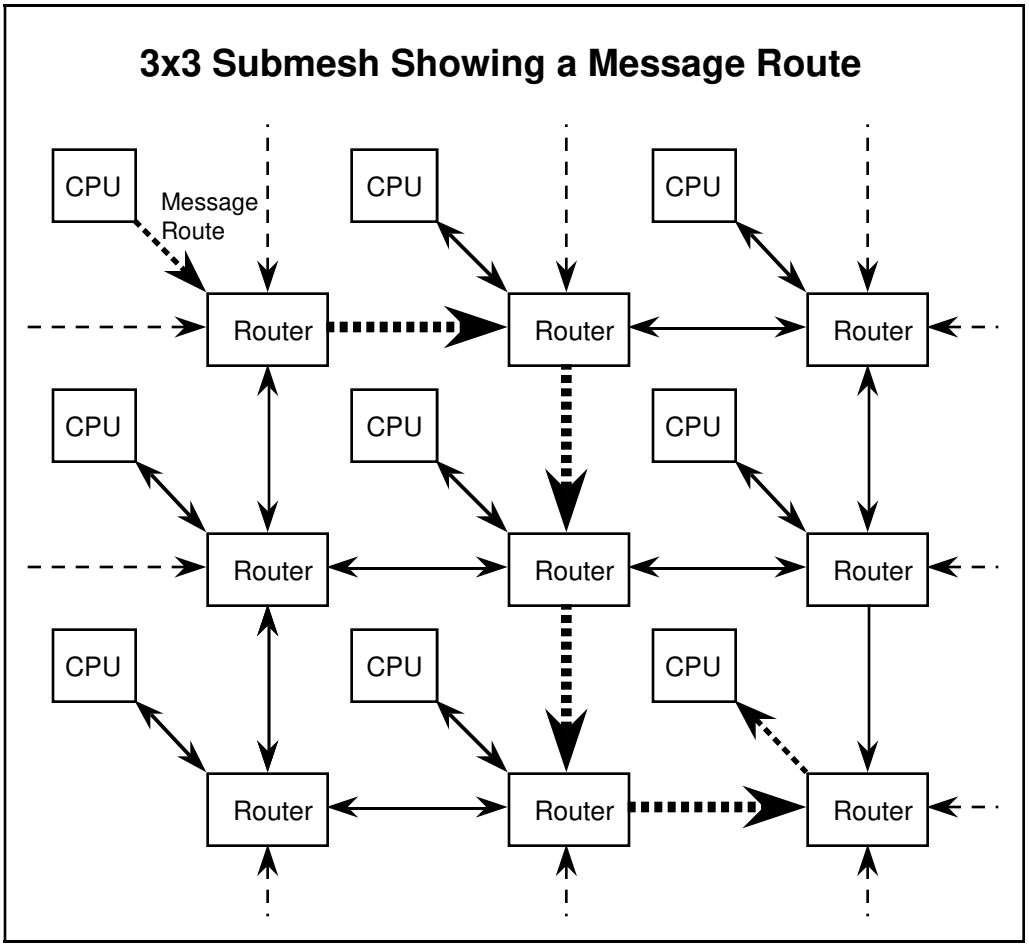

Figure 1

1.1 The Chaos Router

The chaos router is composed of numerous functional blocks. Among these are the channel controller, input frame, output frame, crossbar and multiqueue. Figure 2 shows the relationship of these functional blocks.

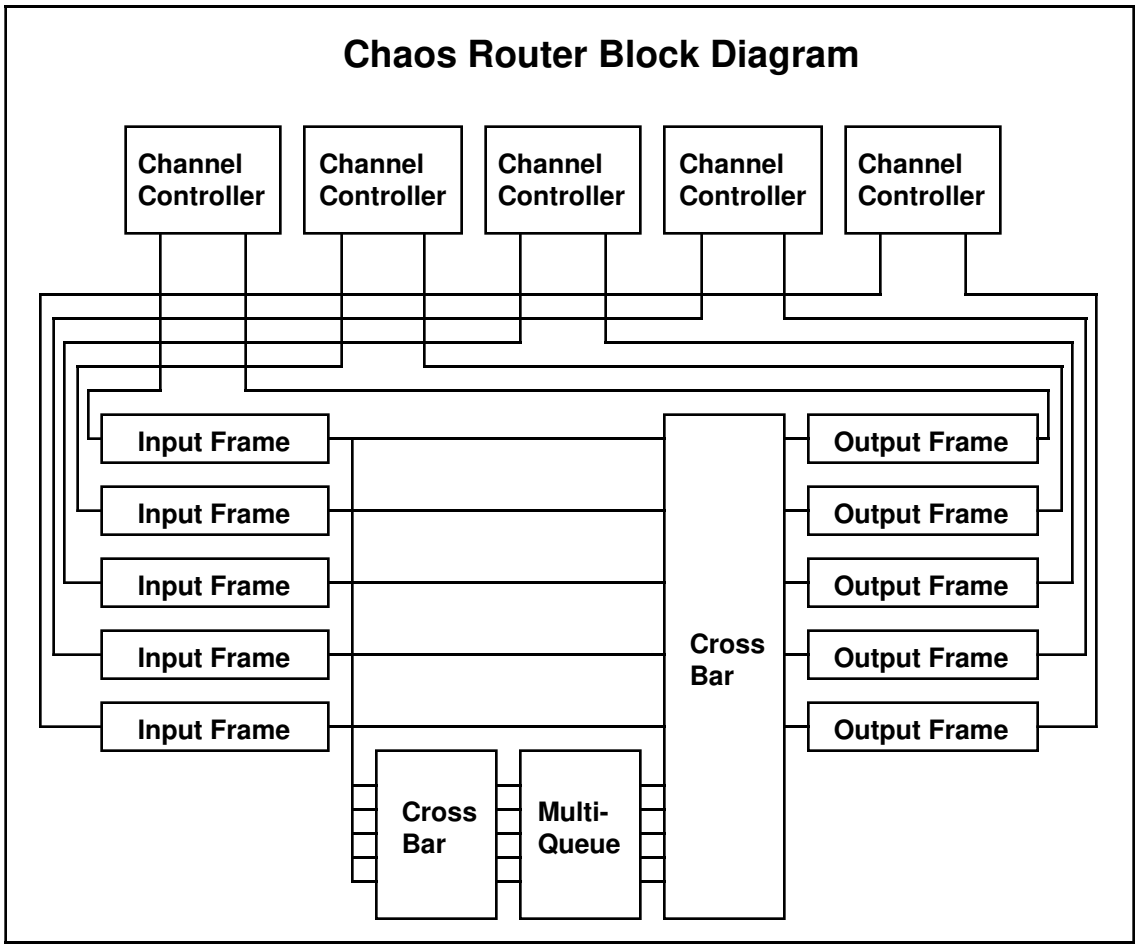

Figure 2

Data enters a router via the channel. (In this document, the term *channel* refers to the physical interconnect between two routers. The terms *channel controller* and *controller* refer to the functional block within the router that sends and receives data across the *channel*.) As data is received, it is placed in the input frame, where it waits until it can be moved to either an output frame or the multiqueue. The input frame contains a FIFO that can hold an entire message. When a message in the multiqueue or an input frame is available to be routed to another router, it is placed in an output frame. The output frame, like the input frame, contains a FIFO that can hold an entire message. The data waits in the output frame until the channel controller can transmit it across the channel. The channel is bi-directional, so the controller must arbitrate for its ownership. The multiqueue serves as a holding place for data that cannot be quickly moved to an output frame. It is important to note that messages can cut through the input and output frames as well as the multiqueue. They do not have to wait until the FIFOs get full.

1.2 Speed Limitation of the Synchronous Controller

This report makes frequent reference to two channel controllers, referred to as the *phase-adjusting* controller and the *synchronous* controller. The phase-adjusting controller is the object of this report. The synchronous controller is the one currently in use by the chaos router.

The synchronous controller assumes that a flit (messages are composed of a sequence of words which are called *flits*) of data can be transferred from one router to another in a single clock cycle. This assumption limits the speed at which the router can operate and requires that the clocks of all the routers be in phase with each other to a high degree of tolerance. The minimum transmission time of a single flit is the sum of the delay through two latches, the pad delay and the delay across the interconnect. Data begins to go across the channel at the beginning of $\varphi 1$ and is latched at the destination at the end of $\varphi 2$. Therefore, the minimum clock cycle is the sum of the transmission time, the clock underlap (the time from the end of $\varphi 2$ to the beginning of $\varphi 1$) and the desired skew tolerance. The minimum clock cycle is limited almost entirely by factors that cannot be altered.

The motivation for designing the phase-adjusting controller is to overcome the interconnect delay (including the pad delays) as the speed-limiting factor. If flits are pipelined across the interconnect, the limiting factor becomes the speed of the logic within the router. The cycle time of the router can then be shortened by optimization of critical paths, retiming, addition of pipeline stages and various other well-known techniques. The goal of this project was to design a channel controller that can operate at speeds independent of the pad and interconnect delays, while increasing bandwidth and decreasing router-to-router latency.

2 Design

2.1 Overview

The key to overcoming the interconnect delay as a bottleneck is to pipeline the data. If the delay is assumed to be arbitrary, the data is no longer guaranteed to be synchronous with the clock. The phase difference between the data and the local clock is determined by the delay of the interconnect and the pads and the skew between the clocks.

An integral component to the phase-adjusting router is the phase adjuster. The phase adjuster takes as inputs the data to be synchronized, a forwarded clock that is synchronous with the data and the local clock. The output of the phase adjuster is data that is synchronous with the local clock. The phase adjuster is a modification of a circuit provided by Mark Greenstreet [1], [2]. Figure 3 contains a block diagram of the channel with the channel controllers, showing where the phase adjuster is in the data path.

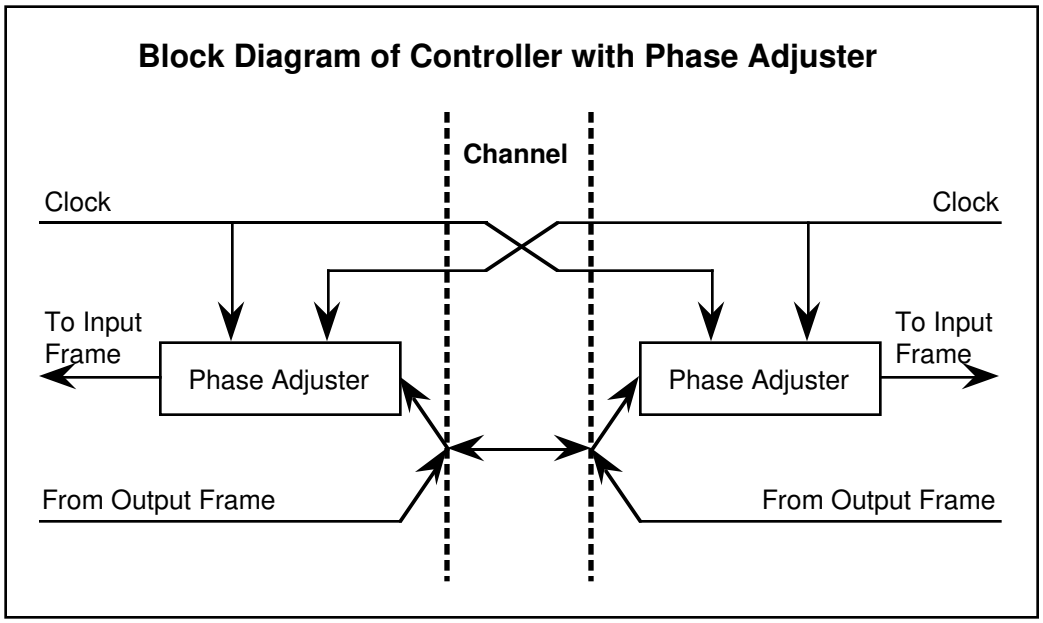

Figure 3

It was desirable that the phase-adjusting controller be compatible with the synchronous controller. The controller was designed in such a way that it may be substituted for the synchronous controller without any modification to the router's logic.

2.2 Communication Protocols

2.2.1 Input/Output Frame Communication

The input and output frames use a simple three-signal protocol. The protocol is designed in such a way that an input frame can be connected to an output frame and form a larger FIFO. The three signals are named *DV* (Data-Valid), *TD* (Taking-Data) and *EOM* (End of Message). The output frame asserts *DV* when it has data to transmit. The input frame asserts *TD* if it can take data. When *DV* and *TD* are asserted in the same cycle, data is transmitted. *EOM* is asserted by the output frame on the last flit of the message. Once the first flit of a message has been transmitted, the remainder of the flits are guaranteed to be transmitted without any delays. Although the output frame and input frame can communicate directly in this fashion, they are never connected directly together. The channel controller provides two virtual unidirectional connections across the bi-directional channel. Each controller communicates with one output frame, one input frame and one other controller.

In addition to *DV*, *TD* and *EOM*, there are two other signals used to communicate between the controller and the input/output frames, called *leavingS1* and *reqChanS1*. The input frame asserts *leavingS1* when the current message is leaving its FIFO. It is used much like an early *TD* signal. If *leavingS1* is low, the input frame may become full upon receipt of the *EOM* flit. The output frame asserts *reqChanS1* to inform the controller that data will be available soon. *reqChanS1* is used much like an early *DV* signal. The controller will not yield ownership of the channel if it receives it (ownership) after receiving a channel request, even if it has not received a *DV*. This can help reduce the latency of a message, particularly when the channel is standing idle. Figure 4 shows a graphical representation of this protocol.

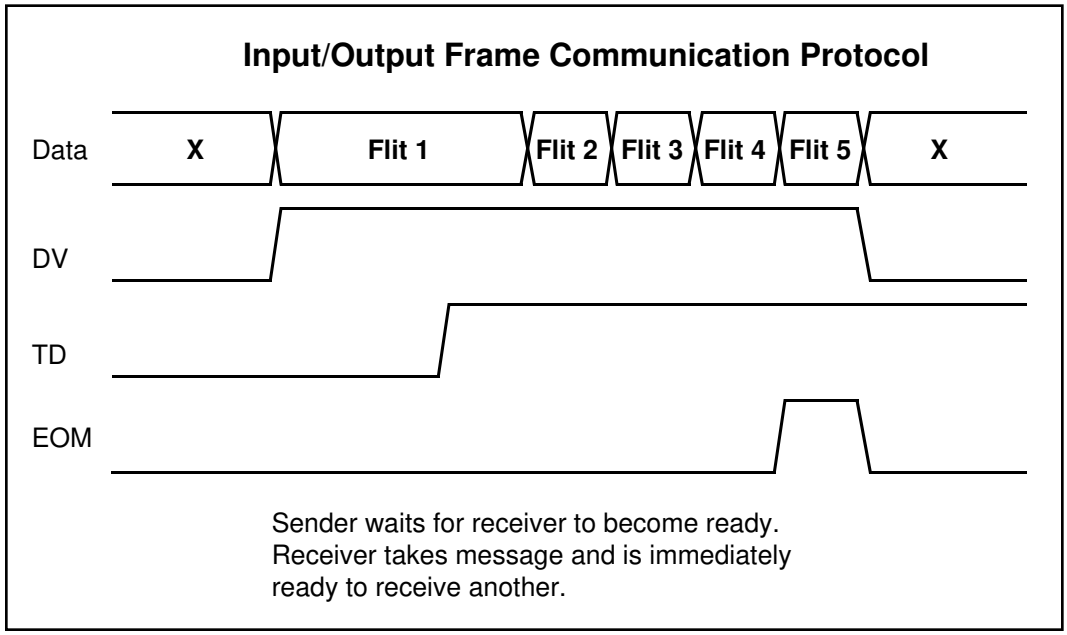

Figure 4

2.2.2 Controller to Controller Communication

The controllers communicate with each other via the channel. The channel is bi-directional, so they must arbitrate for ownership. The controllers alternate ownership of the channel. When a controller gains ownership of the channel, it may send at most one message.

The channel consists of some data bits, a parity bit and three control bits. The controllers use the control bits to communicate with each other to arbitrate for the channel. The three control bits are called *EOM*, *yield* and *nop*. *EOM* has the same function as the *EOM* in the input/output frames. *yield* is asserted when the owner wishes to yield the channel. *nop* is asserted when the data bits are invalid (i.e. the data should not be sent to the output frame). A flit with *nop* asserted is called a *nop* flit and a flit without *nop* asserted is called a *data* flit. A nop flit will always be sent following a flit with *yield* asserted. Figure 5 shows a graphical representation of this protocol.

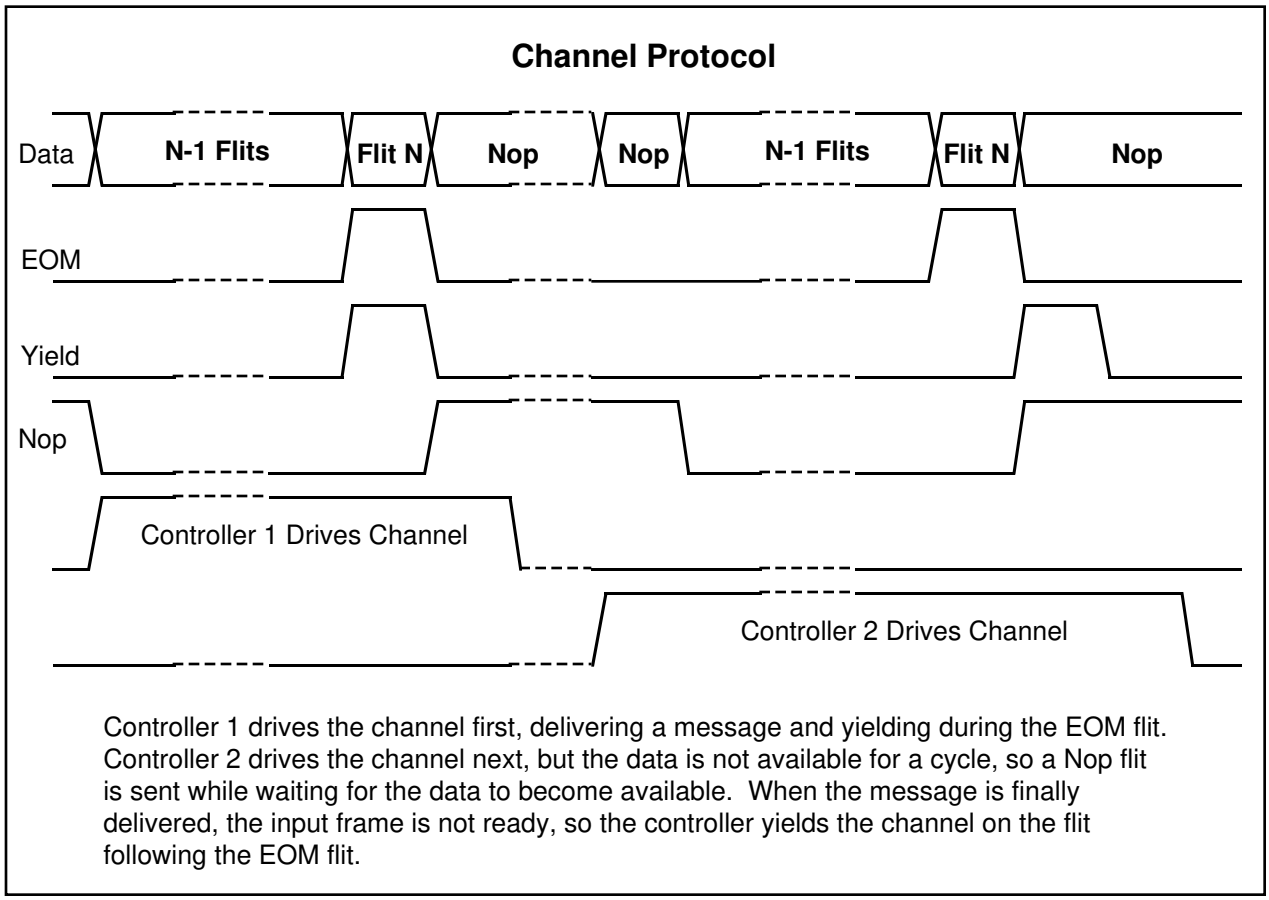

Figure 5

2.3 Functional Blocks

The control logic has been partitioned into two blocks, an input block and an output block. The input block communicates with an input frame and examines the data coming from the interconnect. The output block communicates with an output frame and drives the interconnect.

The data path has also been partitioned into an input block and an output block. The output block puts data onto the interconnect and controls the pads. The input block adjusts the phase of the incoming data stream.

Figure 6 contains a block diagram showing the controller's major functional blocks.

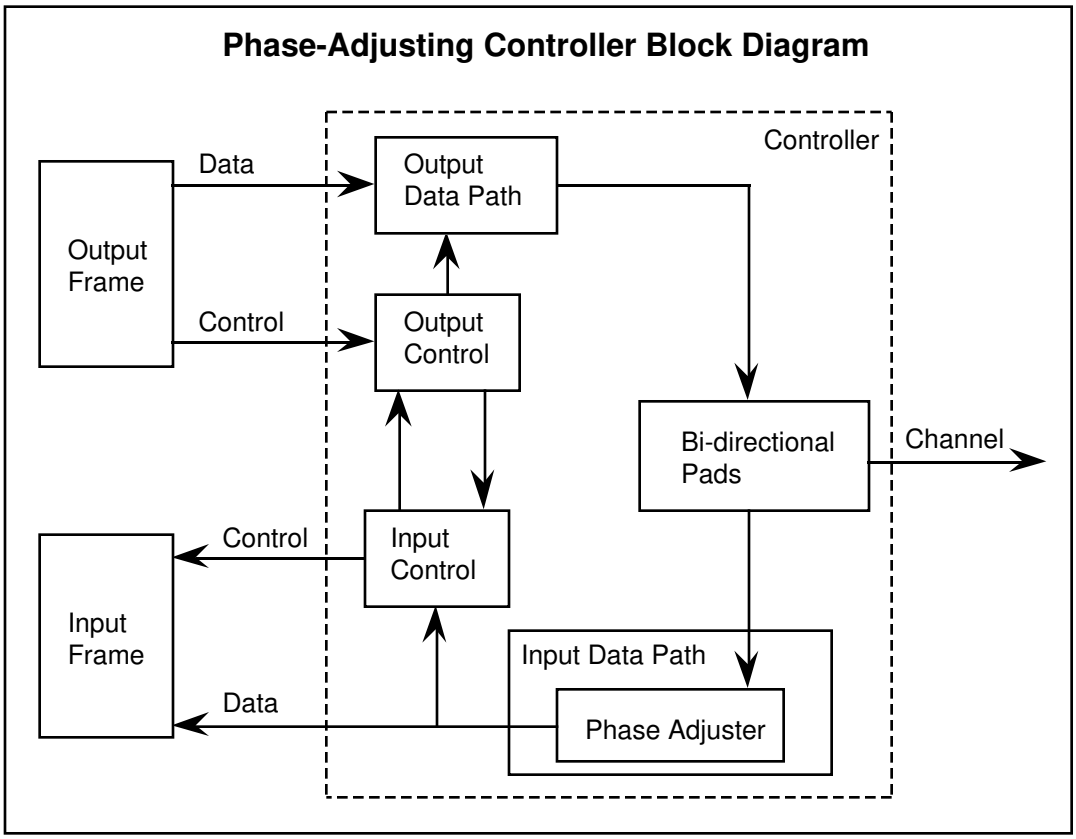

Figure 6

2.3.1 Input Control

The input control logic receives data from the channel and pipes it to the input frame. It also performs a simple mapping from the channel control signals to the input frame control signals. The data lines and parity bit are passed directly to the input frame. The complement of *nop* is sent to the input frame as *DV*. When *yield* is asserted, the input control signals the output control logic that it can become active.

2.3.2 Output Control

The output control logic is a finite-state machine (FSM) that communicates with an output frame and generates the control signals for the interconnect. Figure 7 shows a block diagram of the output control logic.

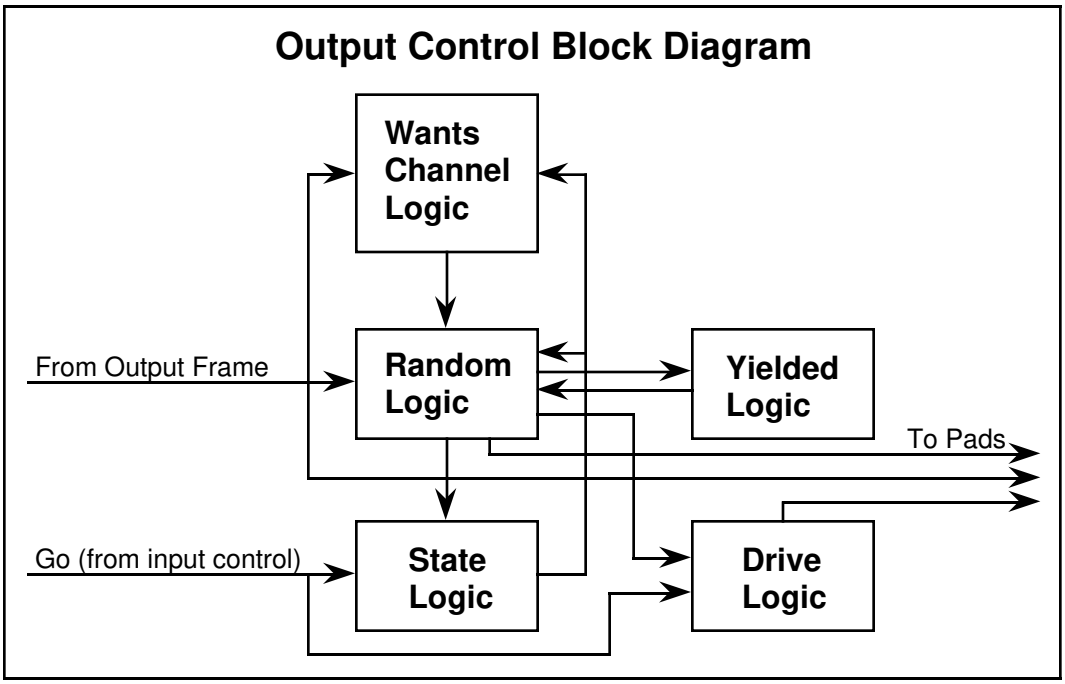

Figure 7

All of the functional blocks shown above are finite-state machines, with exception of the random logic. The random logic generates the *yield* and *nop* signals for the channel.

The *wants channel* FSM indicates whether or not the controller would like to take ownership of the channel. When ownership of the channel is received, this FSM will decide whether it is kept or given back. The *yielded* FSM remembers if the channel was yielded during the current message. Because the channel can be yielded at different times during the transmission of a message, this FSM is needed to keep the controller from yielding the channel more than once. The *drive* FSM controls the output enable of the bi-directional pads. The *state* FSM keeps track of the overall state the controller is in. When the controller is active, it will cycle through two or three states, called *PreSend*, *Sending* and *Sent*. Figure 8 shows the output control's state diagram.

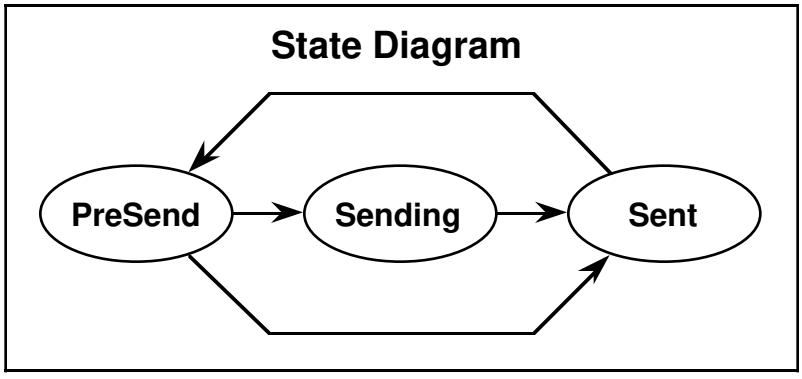

Figure 8

The output control logic becomes active when *goS1* is asserted. Upon being activated, the controller always enters the *PreSend* state. The *sent* state is the inactive state.

During the *PreSend* state, the channel controller will either yield the channel, if no data is available and *reqChanS1* has not been asserted, or will begin sending data. If *reqChanS1* has been asserted, but data is not yet available, nop flits will be sent until data is available. If the channel is to be yielded during the *PreSend* state, the FSM will proceed directly to the *Sent* state, otherwise it will proceed to the *Sending* state as soon as data is available.

During the *Sending* state, data is sent from the output frame to the channel. The FSM will remain in this state until an *EOM* is seen, at which time it will proceed to the *Sent* state.

When the channel controller reaches the *Sent* state, it becomes inactive. It will remain in this state until *goS1* is asserted, at which time it will proceed to the *PreSend* state to start the cycle over again.

The controller asserts *yield* when it wishes to yield ownership of the channel to the other controller. This can occur during the *PreSend* state if there is no data available for transmission and *reqChanS1* has not been asserted. It can occur during the *Sending* state when the last flit of the message is sent. It will occur during the *Sent* state if it has not been asserted previously. The controller will never yield the channel unless the input frame is guaranteed to be able to receive a message. If the input frame is full, the controller will send nop flits until there is room in it.

A flit with the *yield* signal asserted is always followed by a single nop flit. This puts the interconnect in a state in which the input logic can safely wait for a transmission from the other controller.

The *EOM* signal is a faithful copy of the output frame's *EOM*.

2.3.3 Input Data Path

The input data path logic consists of a multiplexor, a clock doubler and the phase adjuster. Figure 9 shows a block diagram of the input data path logic.

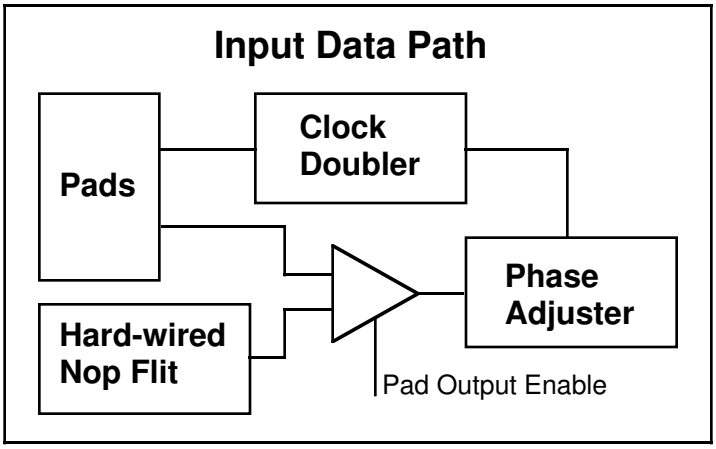

Figure 9

The multiplexor selects between a static nop flit and the data from the pads. When the controller is driving the pads, the phase adjuster receives nop flits as input. When the controller is not driving the pads, the phase adjuster receives actual data from the pads.

The phase adjuster requires a forwarded clock from the sender. This clock is sent as a half-frequency clock. If a normal clock were forwarded, the interconnect for the clock would require twice the bandwidth as that for the data. The clock doubler reconstructs the original clock of the sending controller.

2.3.4 Output Data Path

The output data path logic consists of a multiplexor, a register bank and a clock divider. Figure 10 shows a block diagram of the output data path logic.

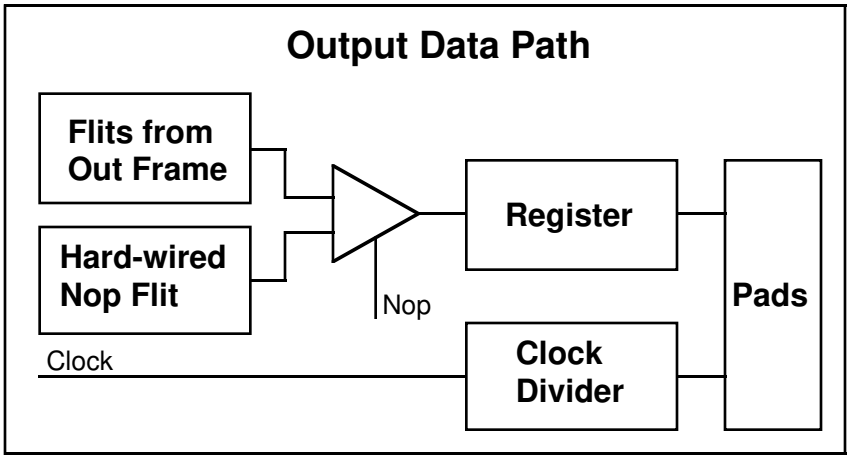

Figure 10

The multiplexor sets the data lines to those of a nop flit whenever *nop* is asserted. The register bank holds the data and control signals steady for an entire cycle. Because the control signals arrive late during $\varphi 1$, the register is used to hold these signals steady during $\varphi 2$ and $\varphi 1$ of the following cycle. The clock divider divides the clock frequency in half. This half frequency clock is sent across the interconnect as the forwarded clock.

2.4 Verification

2.4.1 Phase Adjuster

It is fairly easy to prove the correctness of the phase adjuster. To be correct, the phase adjuster needs to be free of metastability and to be able to compensate for clock skew adequately.

2.4.1.1 Metastability

The phase adjuster is composed of a series of cascaded asynchronous registers that form an asynchronous FIFO. Each asynchronous register is composed of two latches and has a *request* and an *acknowledge* signal. Each register also generates an *acknowledge out* and a *request out* signal. Asserting *request* puts data into the register and asserting *acknowledge* takes it out. *acknowledge out* is asserted when the register is taking the data and *request out* is asserted when the register is ready to deliver data. Figure 11 shows the schematics for an asynchronous register.

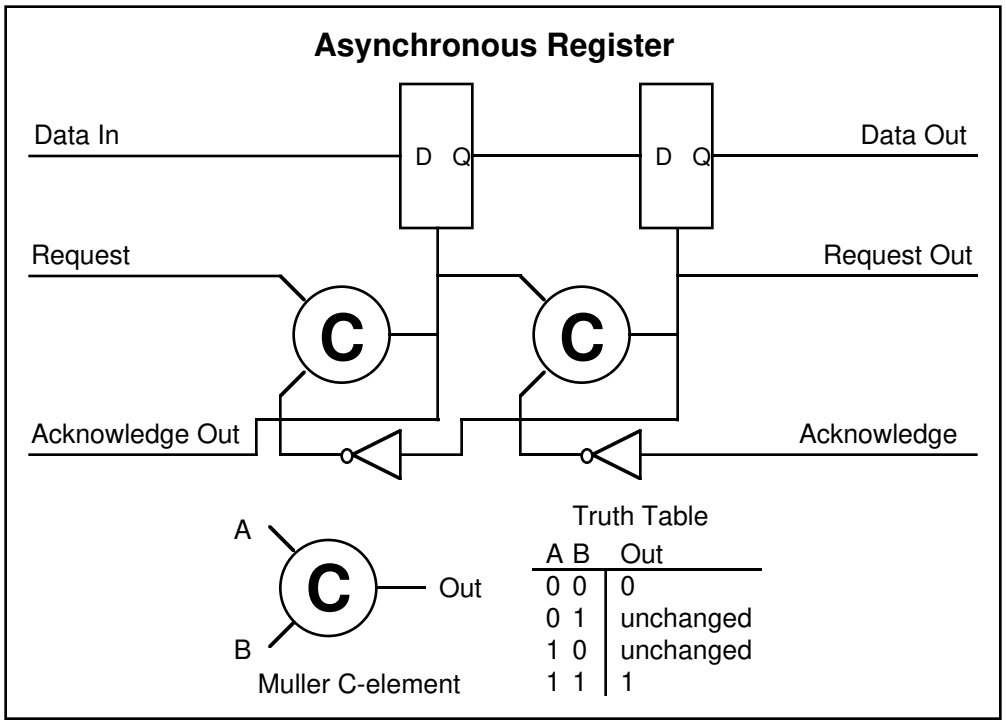

Figure 11

The registers can be cascaded by making the following connections. *acknowledge out* is connect to *acknowledge* of the previous register. *request out* is connected to *request* of the following register. *data out* is connected to *data in* of the following register.

request for the first register is obtained by forwarding a clock that is synchronous with the data. *acknowledge* for the last register is obtained from the local clock. Data enters the FIFO via *data in* of the first register and exits via *data out* of the last. The entire FIFO is preceded and followed by latches that are clocked by the forwarded clock and the local clock. Figure 12 shows a two-stage FIFO composed of asynchronous registers.

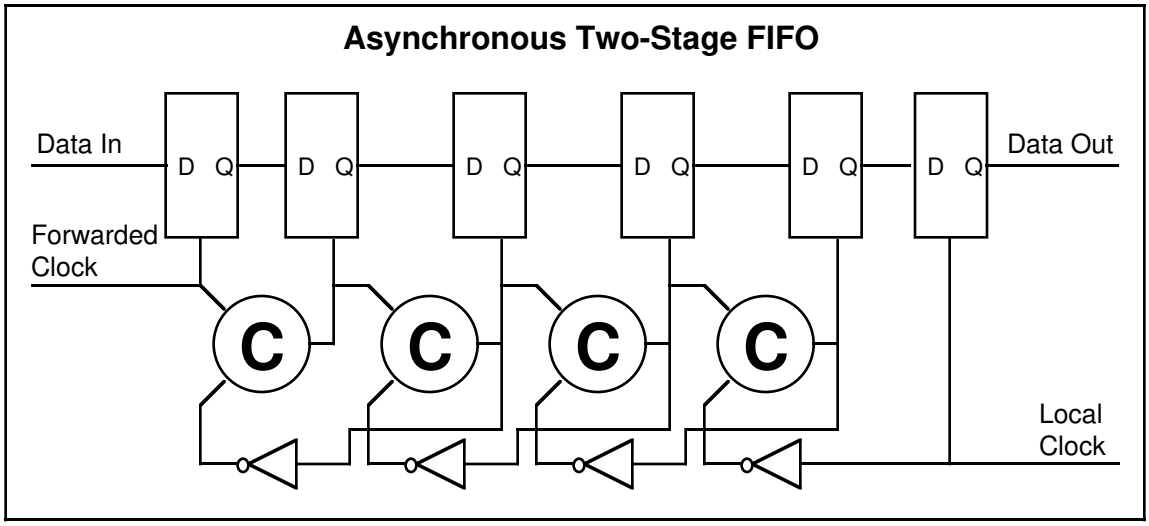

Figure 12

There are two criterion that must be satisfied for the FIFO to be free of metastability: no latch must close while the data on its input is changing and the C-elements must not generate a metastable clock. The following proof of correctness assumes that the minimum time between rising transitions on either clock does not exceed the propagation delay through a register.

Once a given latch is open, both inputs to its C-element must be low for it to close. This means that the previous latch must be closed and the following latch must be open. If a latch must be closed for its successor to close, no metastability can be introduced by a latch closing on changing data (assuming that the propagation delay through the latch is less than that through a C-element).

This argument applies to all the latches except for the first, since the first latch does not receive its *request* from a C-element. The *request* signal for the first register is obtained from the forwarded clock. The first latch can only open when the forwarded clock is high. Because the forwarded clock is synchronous with the data, the data is guaranteed to be stable as long as the clock is high. The entire FIFO is prefixed by a latch which holds the input to the FIFO steady while the clock is low. Because the clock may not be faster than the propagation delay of a register, the data is guaranteed to have entered the FIFO by the time the forwarded clock rises again. The FIFO therefore guarantees that no metastability will be introduced as long as the time between rising transitions on the forwarded clock does not exceed the propagation delay through a register.

The data leaving the FIFO is synchronous with the FIFO's *acknowledge*. *acknowledge* is obtained from the local clock, so the data leaving the FIFO is guaranteed to be synchronous with the rest of the controller.

The second thing that needs to be shown is that the C-elements can never output a metastable clock. The only way for this to happen is for a C-element's inputs to change simultaneously while one is high and the other is low. The inputs to each C-element are obtained from the output of the previous C-element and the complement of the output of the subsequent C-element. If the inputs to a C-element are not equal, the outputs of the previous and subsequent C-elements must be the same. If these outputs are changing simultaneously, two adjacent registers must be in the same state and changing to the next state simultaneously. For this to happen, the clock cycle must be at least as fast as the propagation delay through one register. This violates the assumption that the minimum time between clock transitions was not less than the propagation delay through an asynchronous register.

We can therefore conclude that the phase adjuster is free of metastability as long as the clock cycle time is not shorter than the propagation time of an asynchronous register. However, noise on the forwarded or local clocks can shorten the cycle time for one cycle. Rapid changes in skew between the two clocks has the same result. Therefore, the cycle time must be at least as long as the delay through an asynchronous register plus the maximum change in skew per cycle.

There is one additional issue that needs to be addressed: the pads are bi-directional. It does no good to assure that the phase adjuster is free of metastability if the data is not synchronous with the forwarded clock (as is the case when a controller is driving the channel). Therefore, the input to the phase adjuster is multiplexed. When the controller is receiving data, the phase adjuster receives its input from the pads. When the controller is driving the channel, the phase adjuster receives a nop as input. Since every transmission, whether it be a message or just a yield, is terminated by a nop flit, the select line of the

multiplexor can be guaranteed to change only when the data and control lines have a nop on them. Therefore, the data and control lines are guaranteed to never change asynchronously with the forwarded clock.

2.4.1.2 Skew Tolerance

In the previous section it was shown that the phase adjuster can compensate for some amount of skew. An important consideration is the amount of skew it can compensate for, and how much that skew can vary following initialization. Figure 13 shows the model used for analyzing how the phase adjuster behaves in the presence of skew.

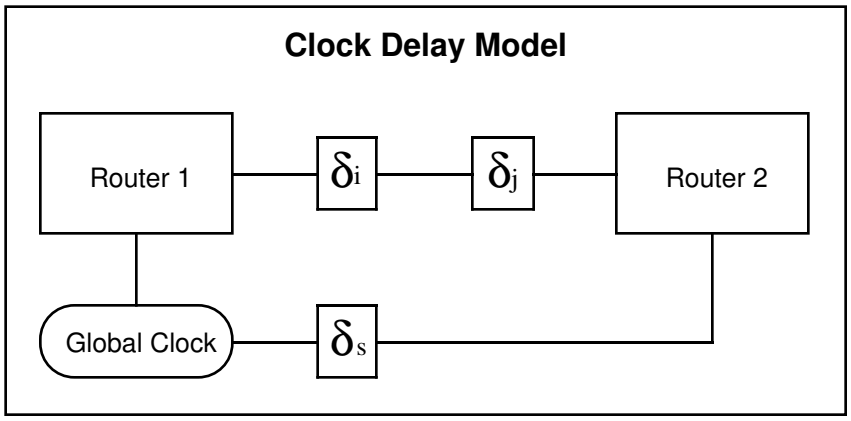

Figure 13

δ_i represents the delay through the pads plus the delay across the interconnect and is strictly positive. δ_s represents the average skew between the clocks and can be either positive or negative. δ_j represents the jitter in a forwarded clock with respect to a local clock. δ_i and δ_s are both constants, determined by the fabrication process, the interconnect, etc. δ_j, on the other hand, is a time varying delay that is assumed to change slowly (less than half a cycle per cycle). To simplify things, let $\delta_d = \delta_i+\delta_j$. To router 1, router 2's clock is delayed by $\delta_d+\delta_s$. To router 2, router 1's clock is delayed by $\delta_d-\delta_s$.

The simplest case is to assume that δ_d and δ_s, are both zero, that the FIFO is two deep and is infinitely fast. Upon initialization, the outputs of the C-elements are set such that the FIFO is exactly half full. When a flit is inserted into the FIFO, another taken out. The newly inserted flit will immediately fall to the bottom of the FIFO where it can be taken out on the next cycle. In this situation, the FIFO is always exactly half full.

Now assume that δ_s is not zero. Because a clock that is delayed an integral number of cycles is indistinguishable from itself, δ_s will be considered to be the actual skew modulo one cycle, ranging from -180 to 180 degrees. If δ_s is positive, router 2 will be late in inserting data into router 1's FIFO, and router 1 will be early. Therefore, Router 1's FIFO, on average, will be less than half full and router 2's FIFO will be more than half full. If δ_s is negative, the opposite condition will exist. Depending on the value of δ_s and where in the cycle the reset occurred, the FIFOs could be, on average, nearly full or nearly empty.

Now assume that δ_j is allowed to be some non-zero value. If $|\delta_j+\delta_s|$ exceeds a cycle, then the phase adjusters will fail, just as before. If δ_s is allowed to be any value, then no guarantee can be made about how much δ_j can be tolerated. This necessitates a longer FIFO.

Now assume that the FIFO is four deep and δ_i and δ_j are again constrained to be zero. Upon leaving initialization (again, the FIFO is initialized to be half full), the FIFO will be, on average, between one quarter and three quarters full. If δ_j is no longer constrained to be zero, the phase adjuster will operate correctly as long as δ_j is less than one cycle. If the FIFO were made even longer, it would be able to compensate for multiple cycles of jitter.

Now assume that δ_i is no longer constrained to be zero. It has already been shown that the phase adjuster will compensate for any value of δ_s even if it is greater than 180 degrees. Since δ_i simply adds to δ_s, δ_i cannot affect the amount of skew and jitter that can be tolerated. δ_i will, however, affect the latency.

It has been shown that with infinitely fast logic, a four deep FIFO will be able to compensate for any values of δ_s and δ_i, as long as δ_j is restricted to be less than one cycle. However, the FIFO is made of real logic and therefore is less than ideal. The amount of jitter that can actually be tolerated is one cycle minus the delay through one asynchronous register.

2.4.2 Protocol Correctness

There are several factors that need to be considered to show that the controller-to-controller protocol is correct. The data in the input frame should never be overrun, both controllers should never output to the channel simultaneously and it should be deadlock free.

The data in the input frame can never be overrun. The controller will never yield the channel unless the input frame is ready to receive data. Therefore, a controller, upon receiving ownership of the channel, is guaranteed that it can safely send at most one message. The output control FSM, upon leaving the *Sending* state, always enters the *Sent* state. From the *Sent* state, the FSM will never send any data until it has yielded ownership of the channel and received it back again. When the controller receives ownership again, it is again guaranteed that it can safely send a message. Therefore, data will never be sent to a full input frame.

Both controllers never output to the channel simultaneously. With exception of the nop flit that always follows the flit with *yield* asserted, a controller will only drive the channel when it has ownership. Upon initialization, one controller has ownership and the other does not. Ownership is always explicitly given, never taken. Therefore, the two controllers will never have ownership at the same time. In order to tolerate varying amounts of skew, there must be at least one cycle where neither controller drives the channel. The latency through the phase adjuster guarantees that at least two "dead" cycles will always exist.

The controller is deadlock free. The only possibility for deadlock is if an input frame never became empty because its corresponding output frame was full. Suppose controller 1 and controller 2 both have full output frames and controller 1 sends a message to controller 2 and yields the channel at the end of the message. Controller 2 now has full input and output frames. Controller 2 proceeds to send a message and receives another message into its output frame. When the message is delivered, the controller is in the only potential deadlock situation. Controller 2 has a full input frame and a full output frame. It cannot send a message and it cannot yield the channel. However, the router guarantees that when it puts a message into an output frame, it will remove a message from its corresponding input frame, if there is one. Therefore, in the above scenario the router guarantees that it will eventually remove the message from the input frame when it puts the second message into the output frame. Therefore, it can never get into a deadlock situation.

2.4.3 Functional Correctness

Verification that the logic actually implements the protocols was done by simulation. Verification of the simulation results was done by hand using ad-hoc methods. Extensive simulations were performed to verify its correctness.

2.4.4 Speed Verification

The cycle time of the design was determined using irsim. Irsim uses a simple RC model to estimate delays. Gate and drain capacitance is considered, but not wiring capacitance. Resistance through the transistor is considered, but not wiring resistance. The model used by irsim is fairly accurate as long as wires are kept short.

2.5 Alternate Design Considerations

The phase-adjusting controller evolved over a number of months and various tradeoffs were considered. The design presented in this document represents the results of this evolution. The major factors considered for each tradeoff were cycle time, latency, and pin count. In general, cycle time was the primary consideration, followed by latency and last of all pin count.

2.5.1 Latency vs. Pin Count

It is possible to reduce the pin count by eliminating the *EOM* pin. This would require the input control logic to delay the data by a cycle in order to reconstruct the *EOM* signal. While removal of the *EOM* pin would increase the latency of a message by a cycle, it would not waste a cycle on the interconnect. The output frame could be delivering the first flit of a message while the input frame is receiving the last flit of another message. When the network is lightly loaded, the latency of a message through the router (not counting the channel latency) is very low, nominally three cycles. An additional cycle of latency would significantly impact a message's latency under these conditions. Due to the big impact on the minimum latency, this alternative was not taken.

It is also possible to reduce the pin count by eliminating one of the two control signals and multiplexing one of the data pins. The controller asserts *yield* during the last data flit of a message whenever possible. Elimination of one of the control pins would require an additional flit to send the yield signal and hence would often tie up the channel for an additional cycle. The logic that generates the yield signal is the critical path in the controller. Multiplexing the pins would lengthen that critical path, in addition to increasing the complexity of the controller. Due to the increased latency and the longer cycle time, this pin count reduction alternative was not taken.

A cycle of latency could have been eliminated by the addition of an ID pin. The controller depends on there being at least one nop flit between each message. This allows the input control logic to distinguish between an outgoing *yield* and an incoming one. An ID pin would allow the input control logic to distinguish between the two without requiring a trailing nop flit. However, this nop flit is important in avoiding metastability in the synchronizer. At one time the controller did use an ID pin, but it was removed for this reason.

2.5.2 Latency vs. Cycle Time

The logic in the controller is fairly heavily pipelined. By putting more logic in the critical path, a cycle of latency could be eliminated. When the input control logic asserts *goS1*, the output control logic does not respond until the next cycle. By allowing *goS1* to directly affect the state of the output control FSM (as opposed to waiting a cycle for it to affect the state), the FSM could respond one cycle more quickly. This was clearly the most difficult tradeoff to decide upon. The cycle time would have been impacted by approximately 10-15 percent while reducing the latency by a cycle. Reducing the latency at the expense of the cycle time makes the controller perform better for lightly loaded networks and perform worse for heavily loaded networks. Shortening the cycle time at the expense of an extra cycle of latency has the opposite effect. It is definitely not clear which alternative is the best.

2.5.3 Intelligence vs. Cycle Time

It is possible for the controller to take advantage of cycles when the channel is standing idle and there is work that could be done. For example, the controller might have delivered a message, but cannot yield the channel until the input frame becomes ready. It is quite possible that the receiving input frame could receive another message. In this situation, a second message could be delivered while the controller waits for the input frame to become ready.

A fairly elaborate scheme was developed in which the sending controller could query the receiving controller to determine if the input frame could take a second message or not. The receiving controller would respond and the sender might be able to send a second message. The major drawback to such a design is that the cycle time would be impacted and very little would be gained due to the long latency involved in turning the channel around.

2.5.4 Receiver Feedback

In the synchronous router, the sending controller receives one bit of information from the receiving controller. This allows the sending controller to make intelligent decisions concerning whether to yield the channel or not. The controller will not yield the channel unless it needs to or the other controller wants it. This allows the synchronous router to send back-to-back messages in one direction without any wasted cycles between them. It is much like the query-response flits described above, only much more efficient. It can also greatly reduce the number of times the channel is turned around in a lightly loaded network.

It would be possible for the phase-adjusting controller to do the same. However, there are a number of reasons for not doing this:

1. The additional logic would impact the cycle time.
2. It would require an additional pin.
3. It would place an upper bound on the latency of the interconnect.

Receiver feedback was not included primarily because of reason number one. Having no upper bound on the latency of the interconnect is a nice property, but practically, it is not a particular advantage.

3 Results

Because the phase-adjusting controller was designed as a faster alternative to the synchronous controller, the following results discussion consists of a comparison between the two designs.

3.1 Quantitative Comparison with the Synchronous Controller

There are a number of quantitative measurements that can be made of this design. Among these are transistor count, cycle time, latency, hop time, bandwidth and channel utilization.

3.1.1 Transistor Count

Router	Total	Excluding Pads
Synchronous	2384	1304
Phase-Adjusting	4116	2929
Increase	1.73	2.25

3.1.2 Cycle Time

The cycle time for the phase-adjusting controller has been determined through simulation using irsim. The fabrication parameters used were those for a 1.2 micron n-well process.

The cycle time for the synchronous controller is not the cycle time for any actual router. It is the cycle time for a theoretical router representing the fastest possible router based on the limitations induced by the synchronous controller. The cycle time is based primarily on pad, latch and interconnect delays. The latch delays are approximate, the pad delays come from the Seattle Silicon Databook [3]. The following table shows how the cycle time of this theoretical router has been determined:

Minimum Cycle Time for the Synchronous Controller

Pad Delay (Input)	3.7 ns
Pad Delay (Output)	9.6 ns
Latch Delay (X2)	3.0 ns
Underlap	1.0 ns
Total (Minimum)	17.3 ns
Allowance for Interconnect Latency	1.0 ns
Allowance for 5% Skew	0.9 ns
Total	19.2 ns

The figures presented in this report assume the same clock underlap and the same interconnect latency for the both routers. With a 1 nanosecond underlap and a 1 nanosecond interconnect latency, the phase-adjusting controller will function correctly with a 12 nanosecond cycle.

3.1.3 Latency

The following table shows the channel latency, which is the time required for one flit to cross the channel, once the controller has ownership of the channel. It does not include any latency induced by the rest of the router or by channel contention.

Channel Latency

Router	Cycles	nsec
Synchronous	1	19.2
Phase-Adjusting	4	48.0
Speed Up	0.25	0.4

The total latency of a message will be affected by the message length and the channel's traffic pattern. The message length is always 20. Three different traffic patterns are considered: heavy bi-directional, heavy unidirectional and sporadic. Under all traffic patterns, the input frames are always considered to be empty. Non-empty input frames will increase latency for both controllers. Heavy bi-directional traffic assumes that all output frames are always full. Heavy unidirectional traffic assumes that one controller's output frame is always full and the other's is always empty. Sporadic traffic assumes that whenever a message arrives at an output frame, the channel is idle. Traffic is bi-directional, with no bias towards one direction. There is no correlation between the direction of a message and the direction of previous messages. The arrival time of a message is independent of the state that the controllers are in.

The assumption that the input frames are always empty is reasonable for the bi-directional and sporadic case, because under these conditions the input frame would have ample time to begin to clear. This assumption is not as valid for the unidirectional case, because the input frame would be required to always clear in less than a message time.

Given these traffic patterns, we can now compute the latency involved in waiting for the channel. These figures represent the latency from when the message is available to when it begins to be delivered. It does not count the channel crossing time.

Arbitration Latency for
Heavy Bi-directional and Heavy Unidirectional Traffic

	Bi-directional		Unidirectional	
Router	Cycles	nsec	Cycles	nsec
Synchronous	22	422.4	0	0.0
Phase-Adjusting	28	336.0	9	108.0
Speed Up	0.78	1.26	0	0.0

Arbitration Latency for Sporadic Traffic

	Minimum		Maximum		Average	
Router	Cycles	nsec	Cycles	nsec	Cycles	nsec
Synchronous	0	0	2	38.4	1.0	19.2
Phase-Adjusting	0	0	7	84.0	2.8	33.6
Speed Up	---	---	0.29	0.46	0.36	0.57

3.1.4 Hop Time

One of the most important latency-related figure is hop time, which is a composite of the latencies described above. The hop time is the total time required for a message to "hop" from one router to an adjacent one.

For the congested cases (bi-directional and unidirectional), the hop time cannot actually be determined because it is affected by the traffic patterns of the other four channels. The hop time can be considered to be the sum of two latencies, which we will call *input latency* and *output latency*. Input latency is the time between the arrival of a message to an input frame and its arrival to an output frame. Output latency is the time between the arrival of a message to an output frame and its arrival to an input frame on another router. The input latency cannot be determined from the traffic pattern on a single channel, but the output latency can. Even though the input latency cannot be determined, intuitively it will decrease if the output latency is decreased. The reasoning behind this is that the input latency exists because of contention for the channels. If the output latency is decreased, the incoming messages will have to wait less time, thereby reducing the input latency. The output latency is summarized in the following table.

Partial Hop Time (Output Latency) for
Heavy Bi-directional and Heavy Unidirectional Traffic

Router	Bi-directional		Unidirectional	
	Cycles	nsec	Cycles	nsec
Synchronous	42	806.4	20	384.0
Phase-Adjusting	51	612.0	32	384.0
Speed Up	0.82	1.32	0.63	1.0

The total hop time for the sporadic case can be easily determined if the network is assumed to be empty. In this situation, the router will add three cycles to the channel's latency. This is the time required for a message to pass through the input frame, cross the crossbar and enter the output frame. The following table shows the hop time for sporadic traffic, assuming the router adds three cycles of latency and the network is empty.

Hop Time for Sporadic Traffic

Router	Minimum		Maximum		Average	
	Cycles	nsec	Cycles	nsec	Cycles	nsec
Synchronous	4	76.8	6	115.2	5.0	96.0
Phase-Adjusting	7	84.0	14	168.0	9.8	117.6
Speed Up	0.57	0.91	0.42	0.69	0.36	0.82

3.1.5 Bandwidth and Channel Utilization

Bandwidth and channel utilization are presented together because they are direct functions of one another. Bandwidth is simply the channel utilization divided by the cycle time. The following comparisons use the same traffic patterns as before, except for the sporadic case. This case omitted because the bandwidth and utilization change depending on the time between messages. In the following table, bandwidth is measured in millions of flits per second.

Bandwidth and Channel Utilization for
Heavy Bi-directional and Heavy Unidirectional Traffic

Router	Bi-directional		Unidirectional	
	Bandwidth	Utilization	Bandwidth	Utilization
Synchronous	49.4	95%	52.0	100%
Phase-Adjusting	69.1	83%	57.4	69%
Speed Up	1.4	.87	1.1	.69

3.2 Qualitative Comparisons

There are a number of advantages and disadvantages that the phase-adjusting controller has with respect to the synchronous controller that cannot be measured qualitatively.

1. The phase-adjusting controller will operate at clock speeds independent of the interconnect speed. Additional cycles of latency will be introduced as the interconnect becomes slower, but the cycle time need not change. The synchronous controller, on the other hand, must have a slower clock to accommodate a slower interconnect.

2. The phase-adjusting controller can have a very simple clock distribution mechanism. The synchronous controller requires a very tight tolerance on its clock to get its cycle time low. This will require phase-locked loops or some other complex circuitry to keep the clocks in phase.

3. The phase-adjusting controller is more tolerant of noisy interconnects. Because the synchronous controller requires that timing be very tight, it is fairly intolerant of noise. The phase-adjusting controller, on the other hand, does not require tight timing, and hence is much more tolerant of noise.

4. The high-level description of the synchronous controller is more readable and more maintainable. The logic that implements the controller is described in a hardware description language. The phase-adjusting controller, on the other hand, was designed using logic gates. All the random logic was designed by hand. Future modifications of the synchronous controller will require far less work than modifications to the phase-adjusting controller.

5. The synchronous controller is more intelligent than the phase-adjusting controller. This advantage is somewhat reflected in the quantitative comparisons made above, but the figures do not tell the entire story.

4 Future Work

4.1 Alternate Phase Adjuster

The phase adjuster used in this design is large and slow. There are other phase adjusters that would perform this function better. One of these is a variable delay line that compensates for skew by dynamically changing the delay along the data path.

4.2 Message Length and Buffering Considerations

While the phase-adjusting controller is quite fast, it has a very large latency associated with turning the channel around. Much of the advantage gained by the faster clock cycle is wasted turning the channel around. The phase-adjusting controller would perform much better if it did not have to turn the channel around so frequently. One way of accomplishing this would be to lengthen the messages. An alternative to this would be to put a FIFO in the controller. This would allow two messages to be sent without turning the channel around, thus eliminating half the arbitration latency.

4.3 Retiming Test Case

A significant amount of retiming was done to shorten critical paths. This design would provide a nice test case for retiming research being done at the University of Washington. In addition, a description in some type of hardware description language would be helpful for other research being done in synthesizing circuits.

5 Summary

The phase-adjusting controller is a design for a replacement for the synchronous controller used in the chaos router. The goal was to design a controller that would eliminate the pad and interconnect delays as the fundamental impediment to reducing the cycle time, while increasing bandwidth and reducing latency.

This controller has some nice properties that make it an attractive alternative to the synchronous controller. Among these are clock skew compensation and a cycle time that is independent of the pad and interconnect delays.

The controller has a higher bandwidth at the cost of higher latency. Three traffic patterns were examined to compare the latency, hop time and bandwidth of the two controllers. The phase-adjusting controller improved the bandwidth over the synchronous controller by 10 to 40 percent. It increased the latency due to channel arbitration by 0 to 57 percent, while reducing the total hop time by approximately -23 to 24 percent. In general, the phase-adjusting controller outperforms the synchronous controller under heavy loads, while the synchronous controller outperforms the phase-adjusting controller under light loads.

6 Acknowledgments

I would like to thank my advisors, Carl Ebeling and Gaetano Borriello. I would especially like to thank Carl for all the help and guidance he has given me through this project. I would also like to thank Mark Greenstreet for his help with the phase adjuster and for inspiring me to think of this project in the first place. Also, thanks to the entire Chaos team, and in particular to Kevin Bolding. And finally, I would like to thank Larry McMurchie for many hours of technical support.

References

[1] Mark R. Greenstreet and Kai Li. Simple Hardware for Fast Interprocessor Communication. Technical report, Princeton University, Jan 1990.

[2] Mark R. Greenstreet. STARI: A Technique for High-Bandwidth Communication. PhD Thesis, Stanford University, Jan 1993.

[3] Seattle Silicon Corporation. *Seattle Silicon Databook*, 1991. p. 8-23.