An Optimized Hardware Architecture and Communication Protocol for Scheduled Communication

by
David Shoemaker

S.B., S.M Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 1992

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MAY 1997

Signature of Author:____
Department of Electrical Engineering and Computer Science
May 21,1997

Certified by:____
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by:____
Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

An Optimized Hardware Architecture and Communication Protocol for Scheduled Communication

by
David Shoemaker

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 1997 in partial fulfillment of the requirements
for the Degree of Doctor Of Philosophy
in Electrical Engineering and Computer Science

Abstract

Managing communications in parallel processing systems has proven to be one of the most critical problems facing designers. As processor speeds continue to increase, communication latency and bandwidth become more of a bottleneck. Traditional network routers receive messages that are examined and forwarded by a local router. Performance is lost both by the time spent examining a message in order to determine its destination, and by the inability of a processor's router to have a global sense of message traffic.

The NuMesh system defines a high-speed communication substrate optimized for off-line routing. The goal of the project is to explore a new approach to the construction of modular, high-performance digital systems in which the components plug together in a nearest-neighbor three-dimensional mesh. Each component module contains a specialized programmable communications controller that routes message traffic among neighboring modules according to a precompiled pattern. The NuMesh project reserves bandwidth for possible message transfers at compile time. By setting fixed periods in which processors can communicate with each other, no message data need be examined and a compile-time analysis of message traffic can minimize network congestion.

This research will examine the hardware and communication protocols needed to take advantage of scheduled communication, as well as the mechanisms needed to support those cases in which the communication can not be specified at compile-time. In addition, flow-controlled transfers are supported to allow the processors attached to the network to inject or remove messages at undetermined times. A novel architecture is presented that utilizes these ideas. A network chip is implemented that can be connected to a variety of off-the-shelf processors, providing a substrate for a heterogeneous parallel processing system.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis represents an experience that can't really be described in words. MIT Graduate School provides opportunities of learning and interaction that I fear I will never be able to again find. I only hope that I have managed to live up to the standards of those I learned from, and that I can in turn impart a little of the experience to those I meet in the future.

I would like to thank my advisor, Steve Ward, both for his technical expertise and for his personality that made the NuMesh group such an enjoyable place to work. Steve is a rare person that manages to combine an incredible depth and breadth of knowledge with a personality that encourages discussion and debate that is invaluable.

My thesis readers, Chris Terman and Gill Pratt, are great friends as well as mentors. Chris provided incredible help when it came to hardware design, and was a constant source of support when needed. Gill brought out the teacher in me, giving me tremendous support and confidence.

The NuMesh Group has had many members that have made the last five years very enjoyable. In particular, I would like to thank Chris Metcalf for great collaboration and for refusing to allow me to ignore software, Mike Connell whose stay was short but will never be forgotten, and Anne McCarthy who put out many fires over the course of five years. Thanks to Frank Honore, Russ Tessier, Greg Spurrier, Pat LoPresti, and Brad McKesson who made this place that much more interesting.

I'd like to thank God for giving me the strength to finish. In a place where logic rules supreme, keeping one's faith can be difficult and I thank Him for guiding me.

Members of the entire sixth floor provided an incredible amount of entertainment and instruction. I enjoyed many lunch time discussions, even though I was targeted as the lone conservative. I learned a lot from these interactions, and they will be missed.

I want to thank my parents for their support over the years and for believing in me the entire time. My accomplishments are a reflection of their values and beliefs.

Finally, I would like to thank my wife, Judy. It was her love, support, and generosity that made my success possible. It was her patience and understanding during times of frustration as well as her sharing in the joys of accomplishment that made the whole thing have meaning. I look forward to the next challenge our life together will bring.

Table of Contents

List of Figures

Chapter 1

Introduction

Modern multiprocessor systems are limited by their communication networks. As processor speeds and network interfaces improve, communication bandwidth and latency become more of a bottleneck. Although much work has gone into reducing this limitation [Dall87, Bian87, Bork88, Sche91, Lee90, Raja92], the problem persists. This research will present a novel hardware architecture and communication protocol to optimize network performance. By assigning communication bandwidth at compile-time rather than run-time, communications can be more efficiently routed and a host of problems such as deadlock and livelock can be avoided. The thrust of this research is to improve network performance by minimizing network congestion and reducing run-time decisions to improve communication throughput and latency.

1.1 NuMesh Overview

The NuMesh project[Ward93] explores a new approach to the construction of modular, high performance multiprocessor systems in which component modules plug together in a nearest-neighbor three-dimensional mesh. Each component module contains a specialized programmable communications controller that routes message traffic among neighboring modules according to a precompiled pattern. In addition, each node has an independent computation unit that attaches to the communications controller and allows messages to be injected and removed from the system.

In recent history, the backplane bus has been dominant as the primary mechanism for connecting multiple devices together. For small numbers of devices, this model works very well because the designer can simply connect additional processing elements to the bus, without having to consider interactions with other devices or bus arbitration. A protocol has been standardized and a connecting device merely has to follow the protocol set up by the bus designer. As the number of devices grow, the backplane bus quickly runs into trouble, since it serializes all system level communication and impedance matching becomes difficult. High performance designers have developed a variety of special purpose networks, but they require the processor to be modified considerably in order to efficiently communicate with the network. One of the goals of the NuMesh project is to standardize this interface so new nodes can easily be added to a parallel processing system, and any off-the-shelf component can be connected to the network with limited overhead.

This network involves point-to-point communications between NuMesh nodes so the electrical difficulties of traditional busses are avoided. A designer can connect an arbitrary number of processing elements depending on his application needs. The goal is for devices to be added or removed from the network with a “Tinker-Toy” modularity.

A high level view of the NuMesh shows a large number of heterogeneous processors that operate asynchronously. The processors are connected to a communication network which orchestrates communications between the processors on a per cycle basis. At compile time, *communication streams* are extracted from the application and provide the information necessary for the static network during operation. These streams specify a source node, a destination node, and a requested bandwidth of communication. A compiler analyzes these streams and creates a schedule of communication for every communication node. Consecutive clock cycles involve completely different communication patterns through a single node. Some streams may have a very high bandwidth reserved for their communication while others may only be allowed infrequent communication. These bandwidths are determined at compile time, although changes can be made during operation with low overhead. Figure 1.1 shows a global communication pattern for a 2-D network of nodes and how a snapshot of consecutive cycles might appear. Note that although messages A and B both use node (2,0), the communication can be time-sliced so that on consecutive clock cycles, the node is handling completely different communication streams. Message C has been assigned a hundred percent bandwidth so it will be scheduled every cycle. Effectively, this link will be reserved for that message and all other messages that would logically go through this link would be misrouted around the link.

The communication controller in each NuMesh node, known as the communications finite state machine (CFSM), mediates communications between the host node and its immediate neighbors. Off-line analysis of internode communication provides separate communication instructions, loaded during boot-up, for each NuMesh node. Each CFSM provides control signals for its node during each cycle of a globally synchronous clock.

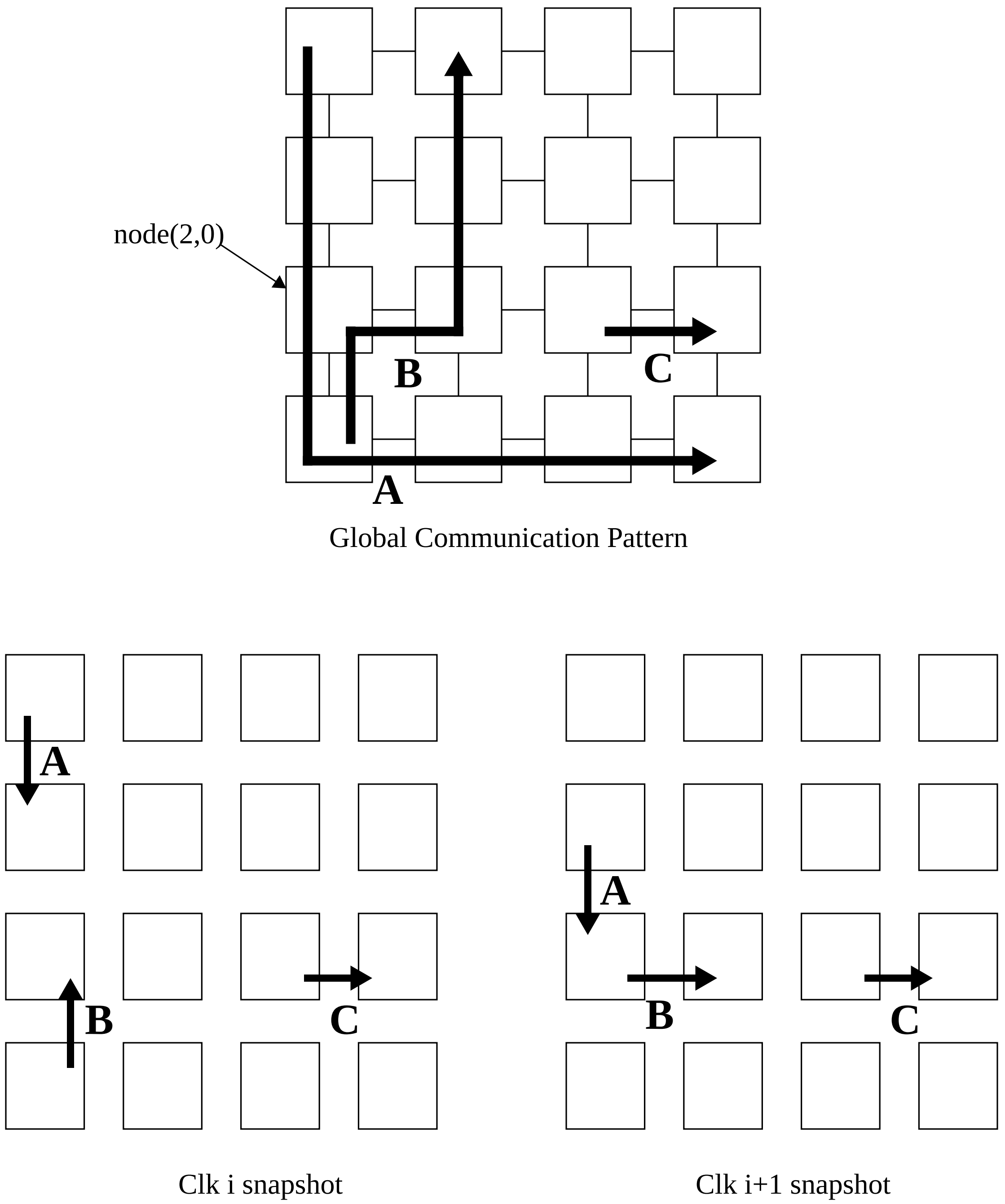


Figure 1.1 High Level View of NuMesh

Individual NuMesh nodes can be connected like Legos, to form the base platform for a scalable, three-dimensional parallel processing machine. Network congestion is minimized through off-line software efforts and the clock cycle delay can be heavily pipelined to provide exceptional performance, especially for those applications that have a limited number of run-time decisions. A picture of a NuMesh node is included in Figure 1.2

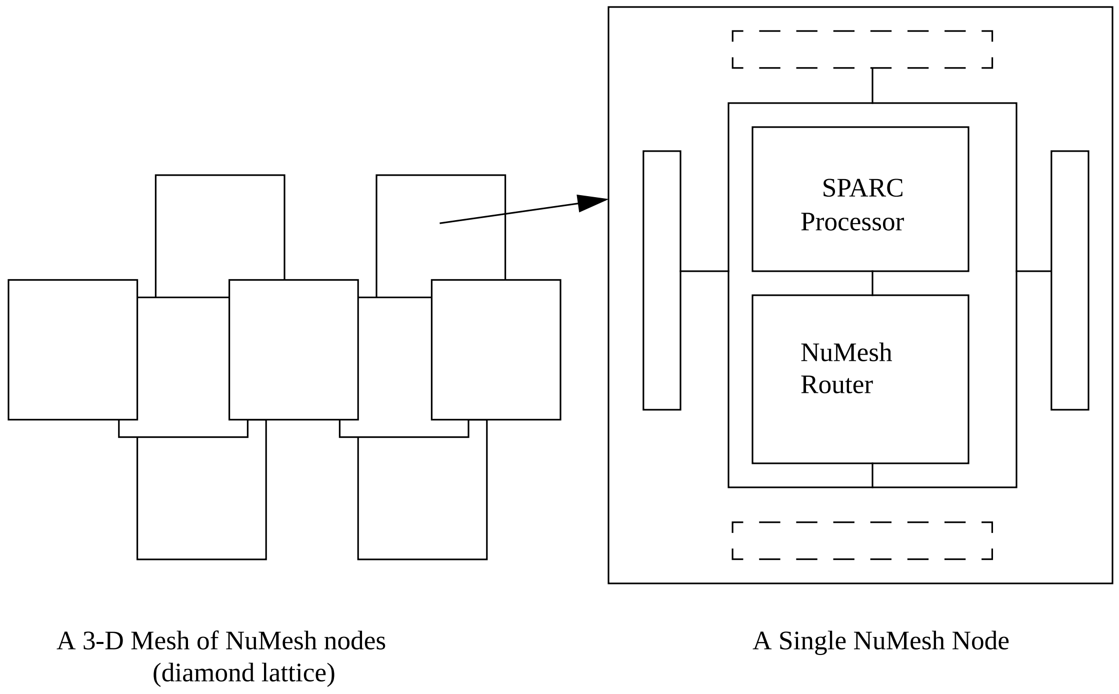


Figure 1.2 NuMesh Overview

1.2 Network Congestion

Typical deterministic routers [Dall87, Leig89] send messages around the network without using any information based on other messages in the system. Encoded destination addresses in the headers of these messages cause the routers to forward the messages down particular paths. If multiple messages need to use the same communication link at the same time, all but one of the messages is delayed. Often there will be unused nearby paths down which a blocked message could be sent, but the deterministic router will be unable to utilize them. Figure 1.3 illustrates a deterministic dimension-ordered, cartesian router suffering needless congestion due to the fact that all messages are routed in the X direction before the Y. A message can only advance one link in the network each clock cycle. In a deterministic cartesian network, the messages always travel in the X direction until the correct X coordinate is matched, and then messages are passed in the Y direction to match that coordinate. In the illustrated example, message A and message B are advanced with no problem for the first two cycles. However, on the third cycle, both messages need to use the same communication link. Since only one message can use the link, the other message must be either buffered or stalled until the link frees up. Buffering the message ties up valuable memory resources while stalling the message ties up communication links in the network.

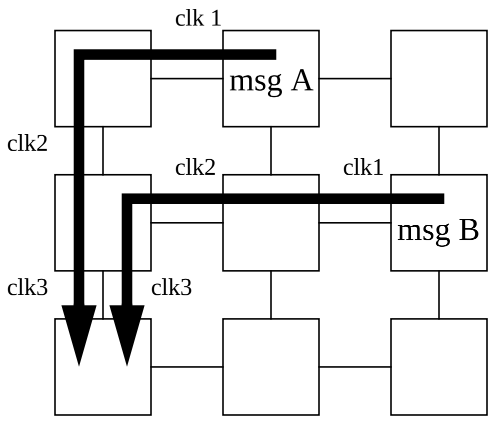


Figure 1.3 Deterministic Router Suffers Congestion

An obvious solution to the problem is for one of the messages to be misrouted. There are several unused links in Figure 1.3, and message B need only be routed in the Y direction before the X direction in order to eliminate the contention. Figure 1.4 illustrates this solution.

Adaptive routers attempt to take advantage of information obtained from recently routed messages in order to utilize links that are not congested. While this misrouting of messages can alleviate congestion, there are associated costs that arise.

The biggest problem that occurs is an increased cost due to hardware complexity. Most adaptive routers today use the notion of virtual channels [Dall90]. Physical ports are multiplexed by virtual threads to keep any one particular communication thread from blocking a resource. In addition, a more complicated communication controller remembers recent network activity in its area and attempts to route subsequent traffic to parts of the network that are less utilized. Much research [Dall91, Aoya93] has gone into trying to evaluate the

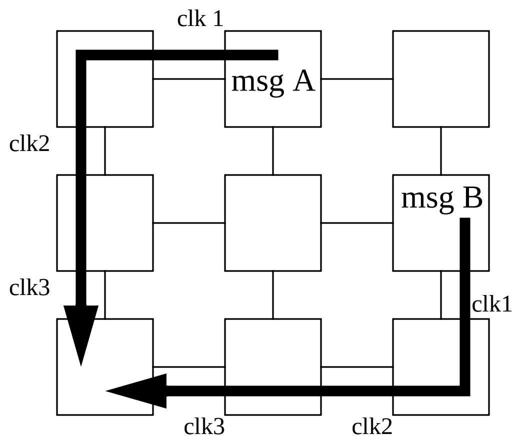


Figure 1.4 Sophisticated Router Eliminate Congestion

trade-off between adaptive router complexity vs network congestion improvement. Even the most complex adaptive routers are often unable to minimize network congestion because they do not have a global view of the network traffic. Since adaptive routers only have information involving messages that pass through its node, they are unable to consider global effects in their decisions. As a result, adaptive routers often make poor decisions resulting in significant network congestion. A trivial example of this is portrayed in Figure 1.5. The router that sends message B does not know anything about message A and sends it on a path that will collide with message A. Likewise, the second node in this path also has no knowledge of message A so it continues sending the message on a collision course. Since no single router has information on the global state of the network, there are communication permutations that can defeat even the more complex adaptive routers. When optimizing from a global point of view, it may turn out that a particular message should be misrouted, even adding to its latency, to minimize overall network congestion. Adaptive routers have a difficult time discovering these cases.

This research proposes an architecture to be used in a system that allocates bandwidth at compile-time. Messages have a reserved time during which they are guaranteed that no other message will be occupying their chosen link. As a result, network congestion can be minimized from a global sense rather than relying on local information. While most static routers provide a fixed communication structure that demands communications at fixed times, this router will allocate bandwidth for possible communications although it will not

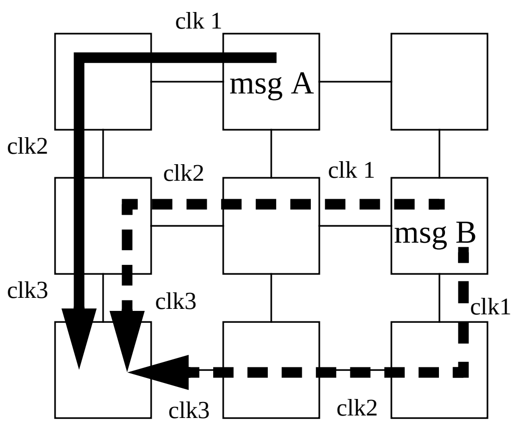


Figure 1.5 Adaptive Routers Still Suffer Congestion

require information to be transferred when a message has a communication link reserved. In addition, mechanisms are present to allow run-time modifications of the scheduled communications to allow for dynamic routing decisions that can not be determined at compile-time. For applications that have communication phases or even evolving communication needs, mechanisms are present to allow the communication patterns to be cached and changed when needed.

1.3 Router Complexity

A basic deterministic router operates by looking at the header of a word and then enabling a crossbar switch to connect the appropriate input to the appropriate output. In order to handle multiple messages at once, multiple crossbars are used. Adaptive routers use a more complex structure since they have to manage both the buffer space associated with virtual channels and the routing decision based on whatever pieces of state they are recording. Typically, none of these operations can be pipelined because they depend on information that can come from a variety of sources at unspecified times.

Additionally, many dynamic routers require a roundtrip transfer of data since the receiving node is required to send an acknowledgment once it accepts data. Since data can be blocked, the sending node has to wait for the transfer to complete before the transfer is ended.

One of the primary goals in this project is to minimize the node latency to either an internode transfer or a small RAM lookup. In addition, the entire architecture is heavily pipelined to allow maximum throughput. Since the communication schedules are determined at compile time, no information need be looked up in the data word to determine a destination. As a result, as soon as a word arrives at a node, its destination is known and it can be immediately forwarded.

Static routing systems often have no need for buffering since communication paths are fixed. While the NuMesh project does reserve bandwidth at compile-time, it does not try to predict the times at which messages can be injected or removed from the network. In order to prevent data from being overwritten or dropped in the communication network due to a processor failing to remove messages at the end of a communication path, the network provides for flow control to allow messages to back up in the system. This requires that buffering be possible for all communication streams. However, the timing for the node need not suffer. While dynamic routing systems often need a round trip communication to ensure a successful transfer, this research utilizes a unique protocol that takes advantage of the scheduled nature of the communication to communicate transfer success with a single internode transfer time.

1.4 CFSM Architecture Overview

A NuMesh node contains a Communication Finite State Machine(CFSM) designed to execute a control instruction for every static communication stream that flows through the node. Attempting to keep track of this state information in a single FSM could prove quite expensive since the number of states of the CFSM would be proportional to the product of the number of states in each of the individual streams. In addition, limited dynamic decisions allowed for a particular static stream would affect the state of the entire CFSM. One solution to this problem is to break up the CFSM into a number of smaller FSMs, each charged with the responsibility of a single communication stream. This insight leads to two possibilities. First, the individual FSMs could be replicated in hardware, allowing true parallelism. Each replicated FSM could issue independent nonconflicting control signals for its particular static stream. The second approach involves using a single FSM time-sliced among a number of independent virtual communication streams. Streams requiring greater bandwidth could be called more frequently by the scheduler. Strict adherence to either of the two approaches would yield unacceptable constraints on the CFSM. The first approach limits the number of communication streams to the number of physical FSMs implemented. The second approach allows an unlimited number of virtual streams, but interleaves them sequentially.

The NuMesh combines both ideas. Two physical and independent pipelines operate concurrently, with each pipeline being allowed thirty-two virtual streams. Each pipeline can choose to have any of the thirty-two streams run on any clock cycle. Each of these streams consists of a single instruction that allows communication between any two sources and destinations on the chip. In the common case, these transfers will involve the four ports on the chip. Since there are two independent pipelines, all four ports on the network can be used on every clock cycle.

The connection between the two pipelines occurs in the schedule RAM. This RAM serves to indicate which stream will operate on every clock. Two stream numbers are encoded in this RAM that serve as indices for the instruction RAM that contains the instruction for each pipeline's thirty-two streams. Another index in the schedule RAM serves to pick out the next entry of the schedule RAM. This allows schedules of an arbitrary length (up to 128) to be created. While the schedule length of each of the pipelines can be different, the schedule in the schedule RAM will be their least common multiple.

Once a communication instruction is selected, the ports are next read and written on subsequent cycles. In between this operation, the data get stored in a register. Since all of these operations are set up at compile time, the architecture can be heavily pipelined to allow the clock cycle to be determined by an internode transfer or a small RAM read. A skeleton picture of the architecture is included in Figure 1.6. Much of the detail for the CFSM is omitted, since this figure is intended only to give a high level view of the chip's operation. Support for flow control, dynamic updates, bootstrapping, etc. are included in a later chapter.

The operation of the CFSM is broken up into four stages. The schedule RAM is first read in order to determine which streams will be read for a particular clock cycle. The second stage involves each pipeline reading the instruction for a particular stream. The third stage causes a source input word to be read and stored in a register. The fourth stage has the source word being written to the appropriate destination.

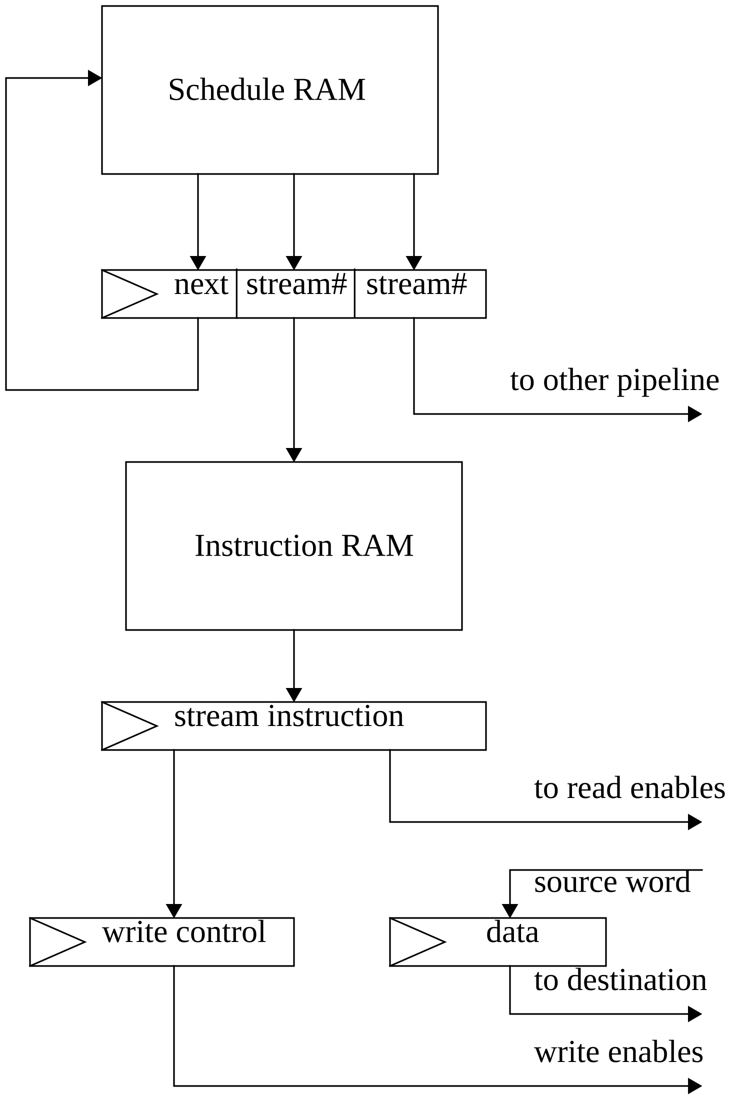


Figure 1.6 Hardware Overview

1.5 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 presents the design space of the NuMesh project and the assumptions made for the rest of the thesis. Chapter 3 presents background and related work. Chapter 4 presents the Instruction Set Architecture (ISA) and discusses the reasoning and trade-offs made in the general architecture design.

Chapter 5 presents a detailed description of the implementation. Chapter 6 presents the result of the chip design and describes the complete hardware and software environment of the NuMesh project. A discussion of future work and an evaluation of the system is also provided.

Chapter 2

Design Space

2.1 NuMesh Design Space Decision

A parallel processing system attempts to take advantage of the fact that applications often have largely independent computational portions that can run concurrently and require some amount of communication between the processes in order to complete. These processes run on different nodes of the parallel processing system, periodically injecting and removing data from the communication network in order to meet the communication demands of the application. Decisions made when implementing the communication network, the processing nodes, and the interface between the two, have marked effects on the number and style of applications that can be supported.

2.1.1 Dynamic Networks

The most general approach to parallel programming is to assume no information about the frequency and patterns of communication that will occur over the course of an application. The need for communication occurs because a node either requests information that exists on another node or it computes some value that needs to be sent to another node. If no information about the frequency or patterns of these communications is known, the sending processor includes a destination address for the message, and the communication network must be able to interpret this data at run-time and eventually route the data to the correct node. The communication routers can keep track of previous messages and can be supplied with various control bits from neighboring nodes to help determine the least congested route for forwarding messages. Routers implemented in this fashion form the broad class of communication networks called dynamic networks [Dall87, Leig89, Lind91, Chie92]. Dynamic routers have the benefits of simplified programming and compilation models. Since all routing decisions are data dependent, the application need only determine where a desired piece of data resides or where a message needs to be sent. The rest of the routing problem is handled in hardware. This dependence on limited hardware to handle the routing decisions suffers a number of disadvantages as well:

• Nodes create messages and inject them into the communication network in an unpredictable pattern causing hot-spots of congestion to be created. Since each node of the communication network has a limited amount of information about which links in the system are currently being used, messages may be stalled due to congested links even though alternative links may be free.

• Every time a message is forwarded a node in the communication network, a data-dependent decision must be made by the router to determine which direction the message should next be sent. For routers attempting to incorporate more information into this decision, the latency of a message through a node is even greater, since the decision process involves some number of gate delays.

Dynamic networks do not attempt to take advantage of communication information present in the application at compile-time. While some applications may yield little such information, many applications provide at least some indication as to the communication that will occur over the course of the application. This information may include the source and destination of upcoming messages, the relative frequency at which nodes can expect to send and receive messages, the exact timing of when processes will send or receive messages, and the evolving communication patterns that might occur under depending on branches taken in the application. While applications may provide all or none of this information at compile-time, dynamic routers can not take advantage of the case even when communication is fully specified.

2.1.2 Systolic Networks

The opposite end of the spectrum yields a system in which the timing of all communications is completely defined at compile time[Beet85]. The processors have a fixed time at which they will create and accept messages, and the communication network knows exactly when and where it will be required to transfer messages through the system. The only variable is the actual data being transferred. The communication nodes consist of simple configurable switches that get set at compile time forming direct physical connections between communicating nodes. Since no run-time data dependent decisions are being made, these switches can operate very quickly. Physical resources are never shared between nodes, so congestion is not an issue. The processing nodes must be able to send and remove messages from the network at predictable times to allow for maximum transfer rates.

While this system works very well for applications involving a small number of regular communications, its uses are limited. As the number of communications in an application grows, the number of available physical links starts to become a scarce resource. Applications that require more complicated communication structures require the communication FSMs to be more intelligent in allocating physical links to the various communications needs.

2.1.3 Static Networks

Similar to systolic networks, static networks demand that all communication patterns and timing be known at compile time. Static networks allow applications to specify more communication paths than systolic networks by allowing physical channels to be shared among a number of communications required by the application[Lee90, Peza93]. The FSMs of the communication routers can be time-multiplexed such that a separate set of communication paths can be supported on every cycle. Since the exact timing of the processors is known at compile time, the communication FSMs can know exactly where each message is to be routed on each clock cycle. The communication network can be viewed as one giant finite state machine that is orchestrating communications between the processors. When this FSM is being created, a clever compiler can utilize all the physical links in the system, thereby maximizing bandwidth for the system. Unavoidable congestion can be handled by time-multiplexing physical channels among the multiple messages requesting use of the channel. The system has the advantages that no run-time data dependent decisions need to be made, and congestion can be minimized at compile time. An arbitrary number of messages can be used by the application since the communication FSMs can allow nodes to share the physical resources by time-multiplexing the channels. However a number of disadvantages remain:

• The application code must be written such that every communication can be extracted at compile time. Not only does the source and destination for each communication message need to be known, but the exact time at which the processing nodes create and accept messages must also be known. Any dynamic behavior in the system can be lethal since the communication network can be viewed as one big FSM that demands precise timing.

• The processing units must be designed such that the time to execute code is completely predictable. Even something as simple as a cache miss can completely throw off the timing of the system. Traditional microprocessors have a number of sources of variable timing. Traps, interrupts, cache misses, and DRAM refreshes are just a few of the sources that prevent application code timing from being easily predictable.

While static networks are more robust than systolic networks, the processor must be carefully engineered and characterized to allow for the precise timing needed in a static system. Static networks are optimized for applications that exhibit an arbitrary number of regular communications. Both the computation and communication behavior of the application must be evident at compile-time. While this model does not hold for all parallel applications, a large body of scientific and graphics code can be made to fit this model.

2.1.4 NuMesh system

The NuMesh system examines a different point in the design space. At compile time, the NuMesh system tries to take advantage of paths of communication in a system, but does not require exact timing information from the processing nodes. *Virtual streams* of communication are extracted from the application that identify a source node, a destination node, and a requested bandwidth that determines how often this communication is likely to occur. A directed graph of communication can be composed from the virtual streams of communication. Similar to the static network, the graph of communication gets mapped to a global FSM of communication, part of which gets stored on each communication node. Each virtual stream gets scheduled for some number of clocks, based on the bandwidth request set up in the program. When a virtual stream is scheduled, data need not be sent, but a communication path will be reserved between the source and destination of the message. Since a message can travel one hop on every clock cycle, scheduling a virtual stream implies assigning consecutive cycles on consecutive nodes of the communication path for the stream. This model of communication is termed *scheduled communication.*

While the communication network reserves bandwidth between communicating nodes, the processors are dynamic in nature, meaning that the exact time at which messages will be injected or removed from the network is variable. This requires that words be allowed to back up in the communication network, without upsetting the timing of the global FSM. Some amount of buffering is assigned to each virtual stream at each node of the communication path.

The decoupling of the communication network from the attached processors yields an interesting combination of static and dynamic behavior. The static nature of the communication network allows the system to take advantage of improved network performance due to the faster cycle times and reduced congestion. At the same time, the restrictive timing constraints of processors found in a traditional static systems are removed, preventing the dynamic behavior of caches and other unpredictable processor effects from crippling the timing of the system.

The disadvantages of the NuMesh approach are that bandwidth in the communication network may go wasted if the processor is unable to fill its reserved communication slots and the processor may be unable to send a message if a virtual stream is not scheduled but the processor is ready to send data. Also, the application must be written such that the sources and destinations of messages can be extracted at compile time. Although the NuMesh system does support limited dynamic routing, there is a cost in efficiency.

2.2 Scheduled Communication

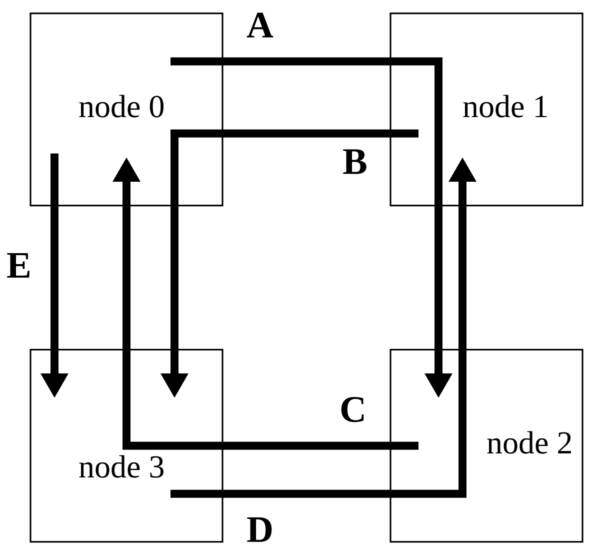
This thesis assumes that communication patterns can be extracted from the parallel language in which the applications are written. Scheduled communication indicates that some determination has been made as to the communication of the processors during compile time. Ideally, all sources and destinations will be specified, and an idea of how often each communication will occur will be provided. Once these virtual streams are specified, a scheduler can load state into each of the NuMesh communication routers that executes a run-time schedule matching the requests of the virtual streams.

However, the determination may be that there are a variety of run-time decisions that will be made. Even for these cases, the NuMesh will load instructions during bootup that allow the nodes to handle more dynamic communication. Off-line analysis of application code may even determine that some routing decisions will be data dependent. In these cases, some mechanism must be incorporated in the scheduled communication to allow for these checks of data. The more regular and predictable the communication patterns, the more benefit can be extracted from scheduled communication.

Scheduled communication does not suffer from the rigidity of static or systolic routing. While a purely systolic system demands that communication be fixed and that data be sent in a regular pattern, a scheduled communication system reserves bandwidth between processors, although this bandwidth need not be used if the source processor has no data to be sent. Dynamic communication has even more flexibility, since all decisions are made during run-time, although this flexibility naturally leads to network congestion and possibly pathological network performance such as livelock or deadlock.

By assuming that the applications can have their scheduled communication extracted at compile-time, this thesis can take advantage of minimized network congestion, a lack of data-dependent routing decisions, and the ability for the hardware to know about possible upcoming communications well ahead of the actual transfer of data.

Figure 2.1 presents a simple example of the kind of information that a scheduled communication analysis might present. In the example, the communication needs of a four node system are examined. In the figure, five communication messages are identified. Each message has a bandwidth assigned to it that represents the percentage of the time that the message is expected to appear in the network. A bandwidth of twenty-five percent would mean that the communication links needed for that message would be utilized around one out of every four clock cycles. These bandwidth numbers need not be exact, but they will be used to reserve communication links for a particular message. If a message gets allocated twenty-five percent bandwidth, but then the sending processor only sends out a message every hundred cycles, then there will be a lot of cycles reserved for that particular communication that are not used. Likewise, if the sending processor has data available to be sent every other cycle, the network will be unable to accommodate all the processor's attempts to inject messages into the network, and data will back up in the sending processor's memory causing the application to stall. While neither of these situations is catastrophic, the closer the assigned bandwidth is to the processor's performance, the better the application will perform.



	BW	Size
Message A	25%	1
Message B	50%	2
Message C	25%	1
Message D	25%	1
Message E	50%	1

Figure 2.1 Scheduled Communication Example

The size metric obtained during off-line analysis serves only to indicate how the bandwidth should be scheduled. When a processor is ready to inject a message into the network, it is usually ready to inject the entire message, regardless of size. If a node has a bandwidth assignment of fifty percent, that means that the data links for that communication will be used every other cycle on average. If the messages are of size two, the processor ideally will have consecutive cycles available to inject both words of the message into the network. The communication thread will not be scheduled again for another two cycles while the processor gets the next message ready. If the message size were one, then the processor would have every other cycle reserved to send its one word message. While, for this example, the difference between size one and size two messages being sent with a fifty percent bandwidth is not that significant, the difference is important for messages of a larger size that might be scheduled infrequently. A message of size thirty communication words might get assigned a bandwidth of one percent. If no size limits are taken into account, the schedule would provide this communication thread one cycle of reserved bandwidth for every hundred cycles. Taking the size metric into account, the thread gets thirty consecutive cycles of reserved bandwidth followed by two hundred and seventy cycles of not getting scheduled. The application performance might be drastically different for the two cases.

Once the scheduled communication information has been extracted, graph analysis can yield a viable network schedule that attempts to meet the required bandwidth and size constraints. The example in Figure 2.1 assumes a fixed path between the nodes, but in practice all that may be specified are the source node and the destination node of the message, allowing more flexibility in the scheduling process. While some constraints may be impossible to satisfy, the communication streams compiler can decide how to come as close as possible to satisfying them. For the simple example in Figure 2.1, one possible schedule is illustrated in Figure 2.2. The schedule that was generated has four cycles in length. For this example, it should be noted that each node can handle two distinct communications on each clock cycle. Each message appears in the schedule based on the bandwidth that was assigned to the node. For example, message A requested a bandwidth of twenty-five percent. Looking at the schedule, we see that node zero (the source node for message A) has message A scheduled in slot zero. This demands that node one, the next node in the communication path, has message A scheduled in clock cycle one. Consequently, node two has message A scheduled in clock cycle two. For messages B and E, which both have a bandwidth of fifty percent, the messages appear in two out of the four cycles for each of the nodes in those paths. Since message B has a size of two, it gets scheduled for two consecutive clock cycles while message E with a size of one gets scheduled every other cycle. This will be the communication pattern for these nodes for all time since after the fourth clock cycle, the schedule starts over again at clock cycle 0. While this simple example is trivial to schedule, more complicated applications provide more complicated communication schedules that often require multiple phases of communication in order to complete.

2.3 Machine Model

The NuMesh node in this thesis consists of a processor, a memory, and a router. The processor is largely irrelevant to this work since one of the goals of this project is to allow for any off-the-shelf processor to be connected to a NuMesh router. The amount of memory on the node is arbitrary, and will logically be determined by the choice of processor.

		nodes			
		0	1	2	3
clock cycle	0	msg A & idle	msg B & idle	msg C & idle	msg D & msg E
	1	msg B & msg E	msg A & msg B	msg D & idle	msg C & idle
	2	msg B & msg C	msg D & idle	msg A & idle	msg B & msg E
	3	idle & msg E	msg 0 & idle	idle & idle	msg B & idle

Figure 2.2 Four Cycle Schedule for Simple Example

The NuMesh communication finite state machine (CFSM) serves as the router, and is the focus of this work. Each of these nodes has four connectors that allow multiple NuMesh nodes to be snapped together to form point-to-point connections. In aggregate, these nodes form a multiprocessor with a user determined total number of nodes. The NuMesh project is careful not to define any sort of memory structure in the global sense, nor does it restrict itself to any particular message passing philosophy. The idea is that these decisions can be made in software and easily laid on top of the NuMesh architecture. Some small efforts have been made in hardware to facilitate different strategies, although NuMesh takes the RISC approach in limiting the amount of hardware that is thrown at problems which can be more easily solved in software.

For the prototype implementation, each NuMesh node is connected to a SPARC processor and eight megabytes of memory. The CFSM is implemented in a custom gate array chip designed by the author for this project.

Chapter 3

Related Work

The NuMesh project defines a communication substrate for scheduled communication. The project attempts to limit network congestion while minimizing both latency and throughput in the network. By taking advantage of scheduled communication, network performance can be more efficiently managed. The logical end of the NuMesh project is a parallel processing system with an arbitrary number of network nodes. Over the past decade, much research has gone into trying to optimize parallel processing networks from both the hardware and software angles. Many previous projects have dealt with some of the myriad ideas that the NuMesh project needs to be successful. This chapter will serve as a literature review of the major areas upon which the NuMesh project is built. Logically, these areas fall into eight categories: network theory, adaptive networks, the iWarp project, scheduled communication, programming environments utilizing scheduled communication, NuMesh prototypes, communication standards, and system architecture issues. The headings in this chapter will address these eight areas.

3.1 Network Theory

Many people have tried to both characterize and create oblivious or deterministic routing schemes[Dall87, Leig89, Raja92]. For a routing scheme to be termed deterministic, the only information allowed when choosing a message route is the message destination, which is usually stored in the first word of the message. This simplest algorithm for deterministic routing is the *basic greedy,* dimension-ordered algorithm. It works for any cartesian network with *d* dimensions. The algorithm is as follows:

- Messages are routed in dimension 1
- If a node has multiple messages that need to use the same link, the message with the farthest distance to go gets preference
- Once a message reaches the correct location for dimension 1, the process is repeated for all remaining dimensions *d*

Much research has gone into analyzing the basic greedy algorithm on the 1-1 packet routing problem[Leig93, Raja92]. The 1-1 packet routing problem consists of a number of single word messages, called packets, being routed at the same time in a network. Each node in the network will receive at most one packet, although some nodes will receive no packets. Figure 3.1 shows an example of greedy routing on a 5 x 5 mesh. In the example, packets are first routed in the X direction until they reach the proper X coordinate, where they are then routed in the Y direction. The packets can advance one node every cycle. In the case of packets A and B, they reach the middle node of the bottom row on the third cycle, and then both want to be routed in the positive Y direction. In the case of the simple greedy algorithm, packet B would be routed first because it has farther to go. Packet A would have to be buffered and sent out on the next cycle.

Ideally, a network designer expects two factors to determine the number of cycles a packet will take when being routed in a network. The first is the network's diameter. The diameter refers to the longest path in the network. In the case of an N x N mesh, this path would be (N-1) x 2 or 2N-2. This number affects the latency in that it describes the total number of hops a packet may have to take. The second factor is the network's bisection bandwidth. The bisection bandwidth refers to the smallest number of wires that must be cut to split the network in two. This metric is important when we consider all the nodes on one side of the network trying to communicate with the nodes on the other side of the network. The bisection bandwidth can give the designer an idea of how much of a bottleneck the middle of the network could become, thus causing packets to be delayed due to congestion over these midsection links. To figure out the performance ramifications of the bisection bandwidth limit, one divides the total number of processors in the network by the bisection bandwidth. The metric assumes that all nodes are communicating to the nodes on the opposite side of the network and therefore have to go through the wires in the bisection bandwidth. Designers often consider a network's diameter and bisection bandwidth an upper bound for the communication times a network can support.

Surprisingly, for the 1-1 packet routing problem in the 2-D case, the basic greedy algorithm can be shown to always run in optimal time: 2N-2 cycles for an N x N mesh. This number is optimal since any packet traveling across the diameter of the network must take at least 2N-2 hops. Unfortunately, the queue size for the worst case of a 1-1 packet routing

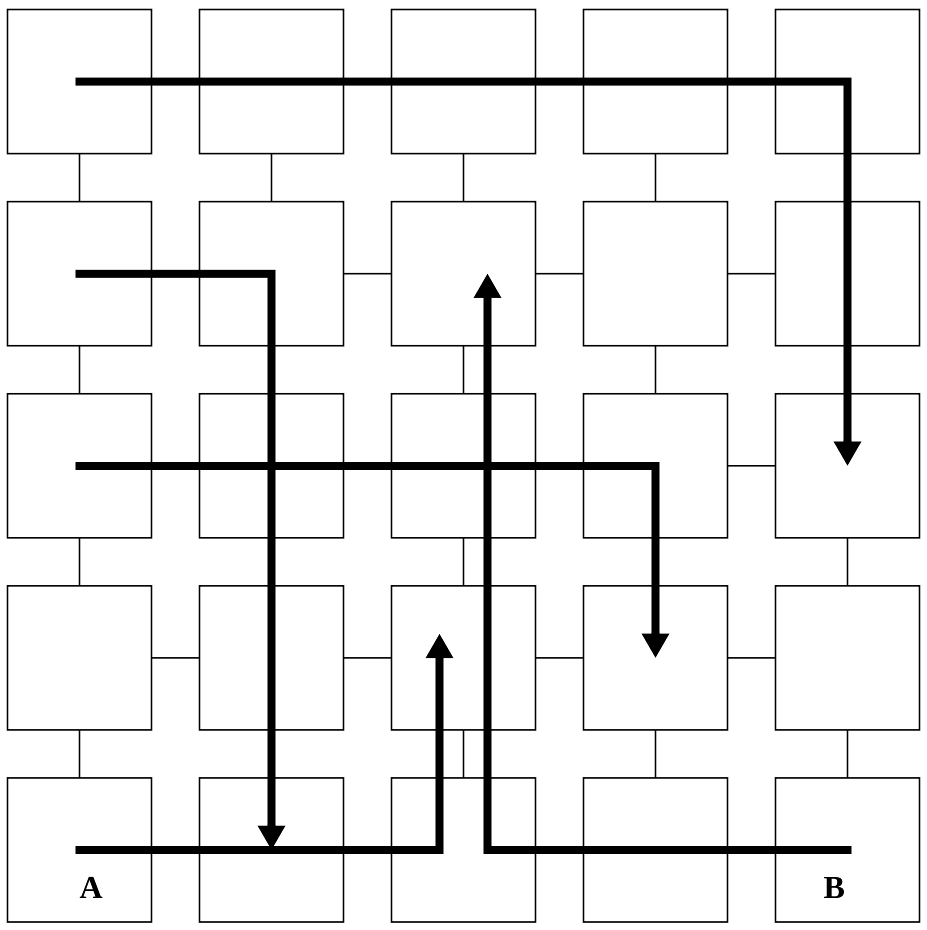


Figure 3.1 Greedy Routing in a 5 x 5 Mesh

problem can be $\Theta(N^2)$ [Leig92]. For all practical purposes, this yields an unacceptable constraint on the network as memory requirements become far too demanding since networks can be composed of many processors. Another alternative involves stalling messages when they are blocked rather than storing them in a buffer queue. While this does allow for constant queue sizes regardless of network size, the run-time performance for the worst case is $\Theta(N^2)$. In [Leig93], the author makes a conjecture that if constant queue sizes are required for any bounded degree network of total size Q, the worst case performance for the network will be $\Theta(Q)$. This observation suggests that the number of cycles required to complete the routing will be proportional to the total number of nodes in the system rather than the diameter or bisection bandwidth of the network. The diameter and bisection bandwidth may turn out to be grossly optimistic when used for estimating network delays. While this remains unproven, the ramifications of the conjecture are quite startling. The rest of this section will show an example of network congestion that is much worse than the diameter and bisection bandwidth of the network predicts.

Consider the case of a three dimensional network of processors arranged N x N x N. The diameter of this network is 3N-3. There can be at most N-1 hops in each dimension and there are three dimensions. To calculate the bisection bandwidth, one need only consider slicing the network down the middle of any of the symmetric planes. Since two of three dimensions will be split, N^2 wires must be cut. Since there are a total of N^3 nodes in the network, one would expect that the bisection bandwidth sets a performance limit of $\Theta(N)$ on the network since N^3 processors might be sending messages over N^2 links. In light of these two constraints, one might expect that for the 1-1 routing problem, the number of routing steps would be proportional to N. Leighton's conjecture, however, suggests that the running time will be much worse for some cases. To illustrate one of these worst case examples, one can consider a variety of very commonly used routing examples. One such example is a bit-reverse operation. In this example, all processors with network addresses (i, j, k) send messages to (i', j', k') where i' is the bit reversal of k, j' is the bit reversal of j, and k' is the bit reversal of i.

Consider routing this series of communications with any dimension-order deterministic routing scheme. Once again we will assume the 1-1 routing problem which only allows one message for each destination. Removing this restriction of 1-1 routing only makes the congestion worse since processors may be receiving multiple messages at the same time. For the sake of simplicity, assume that every node is sending a message. The first step is to route all of the messages in the z dimension to the correct xy plane. A slice of the cube through the x dimension at x=n is provided in Figure 3.2. During the first phase of routing, all of the packets in a particular yz plane are routed to the same column, since the final z coordinate is the reverse of the original x coordinate. A total of N steps is required to complete this first phase, since there will be no contention in the network. The second phase of routing involves sending all packets to the correct y coordinate. At this point, all of the packets in a particular yz plane will be in the same column. The destination of these packets will be the reverse of the source's y coordinate for each packet. Figure 3.3 illus-

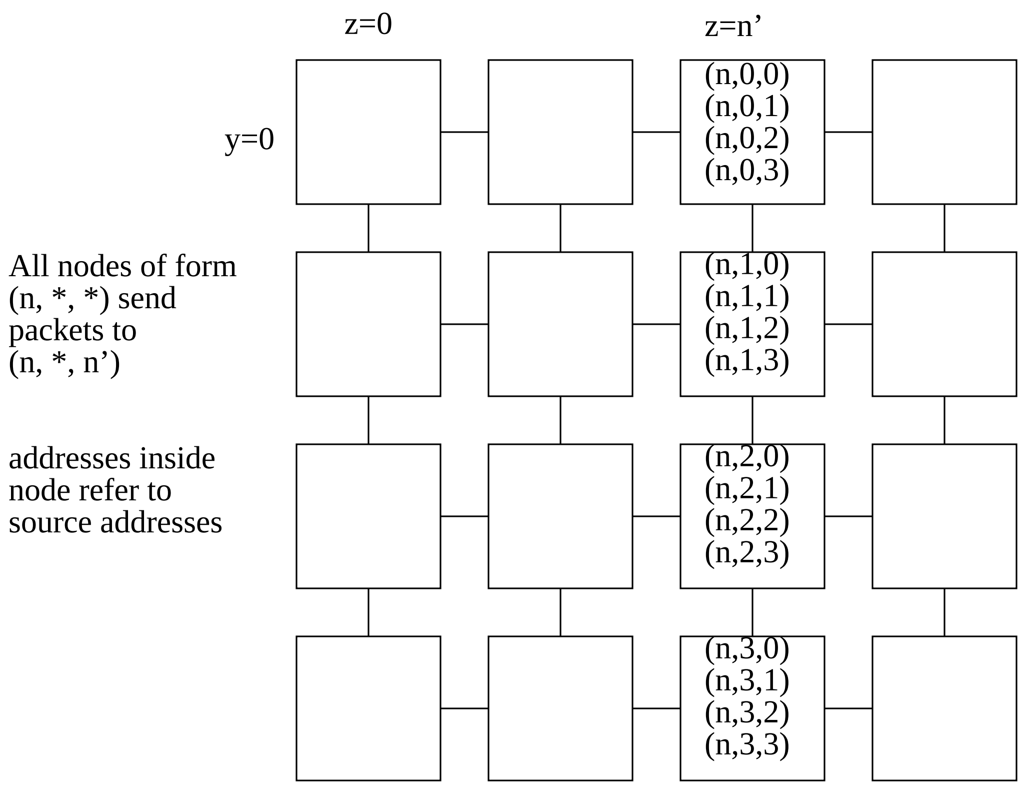


Figure 3.2 YZ Plane Corresponding to x=n, After Phase 1 Routing

trates the phase 2 routing that occurs. Now examine a link at the midpoint of the column. Since all packets are being routed to the y coordinate corresponding to the bit reverse of the source, all packets except the endpoint packets will have to cross this middle link. At the beginning of the stage, there were a total of N^2 packets in this column. Since $\Theta(N^2)$ packets have to cross this single link in a serial fashion, the number of routing steps to complete this phase must be $\Theta(N^2)$. The final phase of the algorithm involves routing messages to the correct x coordinate. The time for this phase must be $\Theta(N)$ due to the fact that there can be at most N packets in any X dimension. This must be true because the problem statement demanded that each node receive only one message, and if all other dimensions have been routed, there can be at most N packets that need to be routed among N processors of any network row. Still, the second phase of the routing process has determined that the total routing time is $\Theta(N^2)$. The network diameter and bisection bandwidth gave over-optimistic predictions of network performance. The reason this example performs so poorly is that during the first phase of routing, all of the packets in a yz plane are routed to

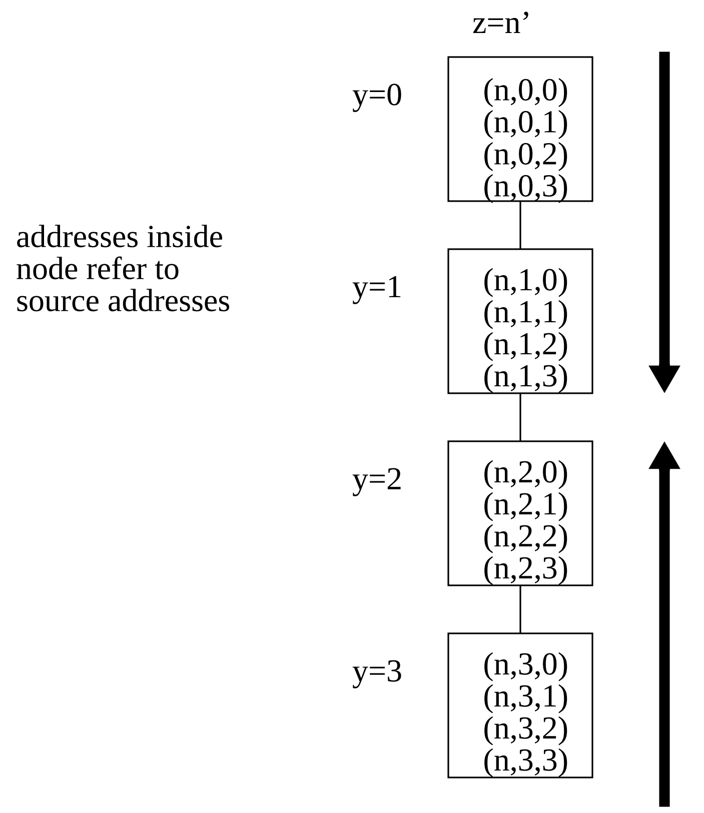


Figure 3.3 Phase 2 Routing Routes Packets in Y Direction

the same column. Then in the second phase of routing this column of nodes has a flurry of activity to route packets in the y direction, while all other nodes in the plane are idle. A simple compile-time scheduling of these communications could have spread the work around to all of the nodes, and a routing time of $\Theta(N)$ could have been achieved.

In [Leig93], the author shows that the odds of randomly choosing a communication pattern that breaks the basic greedy deterministic routing algorithm is quite small. Unfortunately, the author also states that the number of communication patterns that people program happen to intersect these worst case patterns quite regularly for reasons not quite understood.

This example illustrates that deterministic routers can behave quite poorly. The analysis also assumes that communication packets are all being sent at the same time, as is required by the 1-1 packet routing problem. In reality, messages are created and sent randomly and the communication patterns are quite confusing. Messages are routed in all directions at the same time. One very serious problem that can occur is deadlock. Deadlock occurs when messages can block a communication link while waiting for their next communication link to become free. A cycle of dependencies can occur that prevents any message from ever seeing progress. Figure 3.4 provides an example of deadlock. Each

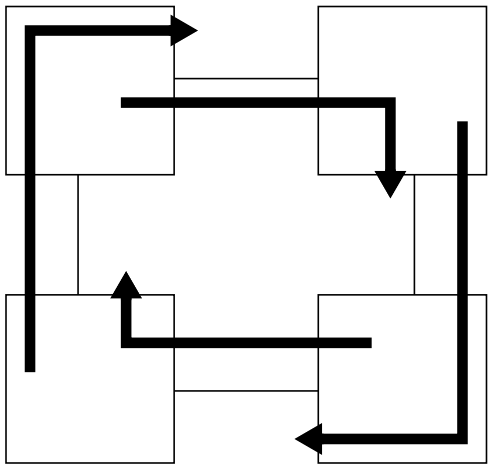


Figure 3.4 Deadlock Example for a Deterministic Router

node in the 2 x 2 array has a message that is attempting to make a right turn. However, each path needed for a right turn is occupied by another packet that in turn is attempting to make a right turn. Since no resource will ever free up, these packets are blocked for all time. Dally proved that dimension-ordered deterministic routing avoids deadlock [Dall91], but most dynamic algorithms need special consideration to avoid deadlock. Deadlock avoidance has been a serious issue in the routing community for some time [Lind91, Bour94]. While many solutions have been proposed, all cost the router in complexity and area. If compile-time scheduling is used, all deadlock cycles can be trivially avoided by breaking any detected cycles of dependencies.

For a two dimensional mesh network, much effort has gone into designing constant queue algorithms that operate in near optimal time [Raja92, Leig89]. Currently, the best bound for queue size that results in the still optimal 2N-2 routing time for the 1-1 routing problem requires a buffer size of 112 per node[Raja92]. Of course, this buffer requirement goes up considerably when the restriction of 1-1 routing goes away. Scheduled communication can be optimized to allow much smaller buffer sizes, or to even remove buffers altogether. The approach of the NuMesh project is to have a limited amount of buffering designated to handle flow control.

The analysis of deterministic routers so far has not attempted to take into account router complexity. As methods for dealing with deadlock, arbitration, or even buffer storage become more complex, router delays become more significant. In [Agar91], the author attempts to include switch delays into the network performance model. Not surprisingly, in most cases the network performance degrades linearly, and degrades far worse than linearly in others as the switching time in the node increases for two and three dimension networks. For lower dimension networks, not only do the node switching times affect performance, but they can be observed to be one of the critical factors in determining network performance. This demonstrates that as routers make more run-time decisions, the effect of router complexity must be considered when determining the benefit to the system. Unfortunately, many such studies completely ignore router complexity.

3.2 Virtual Channels and Adaptive Routers

Two of the bigger potential problems when routing with dynamic routers are the possibility of deadlock and the inability of one message to pass another if the first message is blocked. Figure 3.5 illustrates the second problem. Message A is unable to travel in the negative Y direction because of network congestion. As a result, Message B gets blocked behind message A, even though its path is clear to the following nodes. There is no mechanism for allowing one message to pass another. In [Dall90], the author introduces the idea of virtual channels to alleviate this problem. Virtual channels manage to decouple buffer storage from physical channels to allow for a number of virtual communication streams that are distinct from each other. If one virtual stream is blocked, the packet can be stored in that stream's virtual buffer, while another stream can pass the blocked stream via another virtual channel. Figure 3.6 demonstrates the virtual channel solution. Although message A is still blocked from going in the negative Y direction, the entire packet can be stored in a buffer until its required link frees up, allowing message B to utilize the link it needs.

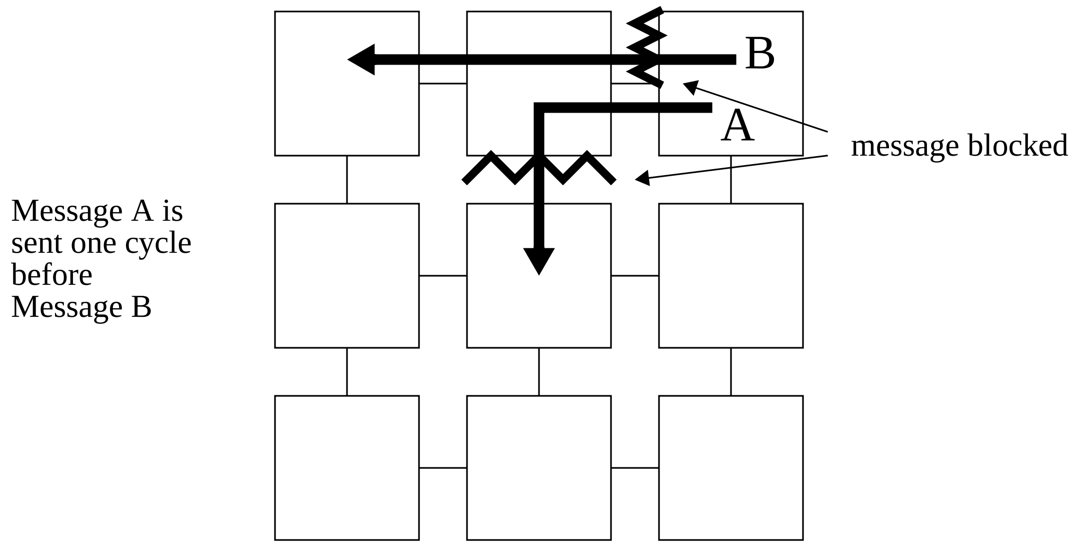


Figure 3.5 Network Contention Caused by a Blocked Message

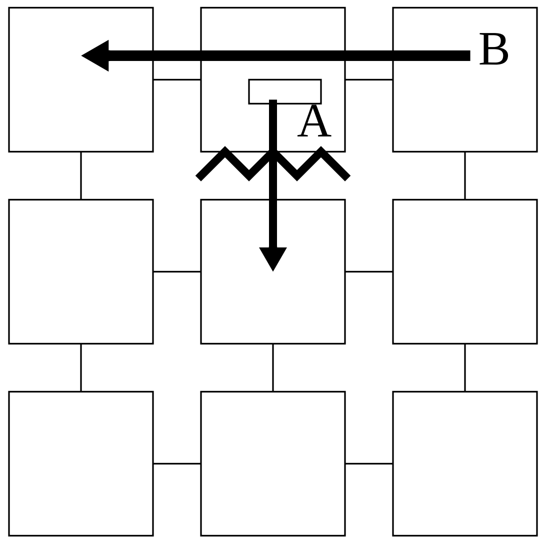


Figure 3.6 Virtual Channels Relieve Tension

Virtual channels can also have the virtue of provably eliminating deadlock. A simple example taken from [Dall90] describes how virtual channels can eliminate deadlock. Deadlock occurs because of a created cycle of dependencies in which each member of the cycle needs a resource that is currently being used by another member of the cycle. One model of virtual channels refuses to allow these dependencies to occur. A normal mesh is connected by bidirectional communication links. Virtual channels will be placed on top of these links to create two distinct networks. One of these networks will only provide links that go in the positive X and Y directions, while the other virtual network will only support links in the negative X and Y directions. There will be separate buffer storage for each of these virtual networks. Cycles could never occur, because buffers in either of the virtual networks could never form a cycle. For a deadlock cycle to occur, there must be a circular chain of resource dependencies in which the first packet of the chain is waiting for a link to free up that is currently being blocked by a series of packets that eventually can be traced back to the first packet of the cycle. With the two virtual networks described above, a packet being blocked in the positive X or Y direction can never be blocked by a link in the negative X or Y direction. This means that a deadlock cycle could never occur.

While virtual channels significantly improve channel utilization for deterministic routers, for a reasonable number of virtual channels (4-8), the throughput of the interconnection networks is still limited to about sixty percent of the network’s capacity [Dall90]. In addition, virtual channels add significantly to the complexity of the chip. Not only does the control logic become more complicated, but the router must be able to handle an additional layer of switching since data has to be switched among the virtual channels as well as the physical ports. Virtual channels also fail to alleviate hot spots in the network, even when alternate paths are available. At the lowest level, the router is still implemented as a deterministic router.

To address “hot spots” in the network, a variety of adaptive routing scheme have been designed. The basic idea behind adaptive routers is to route packets along alternative paths if a requested link is busy. As a result, data can be fanned out to areas of the network that are less congested. A fully adaptive router considers all possible paths between nodes when sending packets in the network. Adaptive routers can choose from among several paths based on network loading, network faults, or any other dynamic data that the switch is recording. Unfortunately, by increasing the number of alternative paths a packet can take, the number of possibilities for deadlock is multiplied. A full adaptive router is described in [Lind91]. In order to prevent deadlock, a total of 2^n virtual channels are

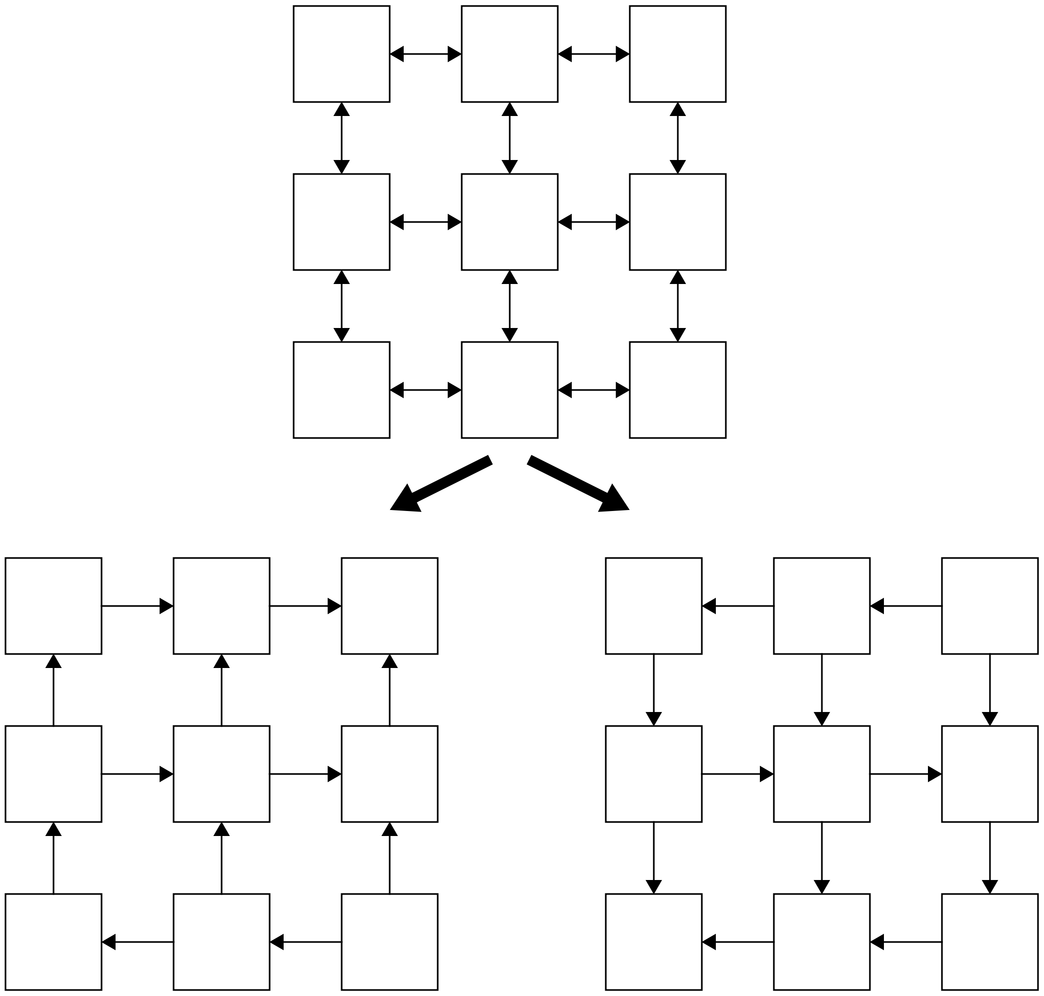


Figure 3.7 Two Distinct Virtual Networks Avoid Deadlock

required where n is the number of nodes along any dimension. The hardware complexity required to implement such a scheme is prohibitive.

As a compromise, several schemes have been developed that allow limited adaptability. By limiting the number of choices a packet can travel, the number of virtual channels needed to avoid deadlock can be reduced.

One such limited adaptive scheme is the *turn model* of adaptive routing [Glas92]. The *turn model* supports a user defined number of virtual channels, but requires none. For every set of channels, every possible cycle must be identified. For every possible cycle, one of the turns must be disallowed in order to break the cycle. The routers must take these absent links into account when determining which direction to next route a packet. One could consider these links to be permanently busy. In order to prevent all of the cycles in a two dimensional mesh, one fourth of the communication links must be eliminated. This scheme allows for limited adaptability while ensuring a deadlock-free network. The complexity of the switches increases since unusual constraints must be checked for every node.

Another popular limited adaptive scheme is the *planar-adaptive* routing scheme [Chie92]. As a compromise between a deterministic routing scheme and a fully adaptive routing scheme, the planar-adaptive scheme tries to route two dimensions at the same time. Subplanes are selected at each point in a packet's transfer and any edge on the plane may be used for a transfer. If some edge of the plane becomes congested, packets will be routed to the less congested side of the plane. To be deadlock free, this scheme requires three virtual channels per physical channel in a two dimensional mesh, and six virtual channels for a three dimensional network. Each subplane is effectively broken up into three virtual subplanes in order to prevent any possible cycles from being created.

While many adaptive schemes purport to create significant network improvement, these studies often neglect the effect of router complexity. If the complexity of the switch causes the latency and the throughput of the network to decrease, this must be overcome by improvements to the network congestion at a commensurate rate. One significant study out of Illinois [Aoya92] studied exactly this trade-off and determined that full adaptability was not worth the added complexity and that even partial adaptability was questionable.

Adaptability negatively effects network performance in two ways. The added virtual channels required to prevent deadlock increases both the arbitration time and control logic of the router. Larger crossbar switches slow down the effective clock cycle of the routing node, since crossbar switch delays grow at a rate $\Theta(N^2)$ where N is the number of virtual channels supported on the node. The authors of [Ayoa92] provided for up to eight degrees of adaptability, meaning that up to eight virtual channels were allowed per physical channel. On average, they discovered that for each added virtual channel, the utilization of the corresponding physical channel. needed to go up thirty percent to justify the cost of the adaptability overhead. While this seems possible for the addition of one or two such channels given the right kind of network traffic, their study showed that many more channels rarely justified their cost.

The Illinois study fully breaks down and compares the cost of the router based on two metrics. Internode delay refers to the amount of time it takes data to get between chips including output buffers, wire delays, and input latches. Intranode delay is determined by the features of the designed router. The Illinois study further breaks down intranode delays into two more metrics. *Path setup* is described as the time it takes a routing node to decode the packet address and choose an appropriate channel destination. *Data through* delays measure the amount of time the data is delayed due to flow control overhead. These two delays can occur in parallel, so only the worst of the two delays determines the routing delay. The planar-adaptive baseline router is implemented in a .8 micron CMOS gate array technology. The internode delay ends up taking 4.9 ns. The intranode delay is much greater. For the baseline case, the *path setup* delay is 10.3 ns while the *data through* overhead is 5.7 ns. Since the *path setup* delay describes the minimum amount of decoding work the node must do to route a packet, a breakdown of this delay is included in Figure 3.8. Since the flow control operations require information from neighbors, it takes two clock cycles for the flow control to complete. This means that the two cycles of *data through* overhead set the clock period to be 11.4 ns. Compared to the 4.9ns of internode transfer time, we see that the overhead for even this limited adaptive routing scheme is quite significant.

By increasing the number of degrees of adaptive freedom, one also increases the number of required virtual channels needed to avoid deadlock. The more adaptability that is added to the network, the more possibility there is to form a deadlock cycle. To ensure deadlock avoidance, the common solution is to create distinct planes of virtual channels. However, as the number of degrees of adaptive channels are increased, the overhead for the adaptive channels becomes significant. Figure 3.9 analyzes the *path setup delay* and the *data through delay* as the number of degrees of freedom are increased. The basic planar-adaptive router corresponds to two degrees of freedom. In all cases, the internode delay stays constant at 4.9 ns. Another interesting data point to consider is the case with one degree of freedom. This corresponds to the most basic deterministic dimension-ordered router. Even for this case, the *path setup delay* is still 5.7 ns, although the *data*

Switch Function	Switch Delay
address decoder	3.3 ns
routing decision block	2.1 ns
flow control handler	2.6 ns
cross bar switch	1.1 ns
virtual channel controller	1.2 ns
total	10.3 ns

Figure 3.8 Breakdown of *Path Setup* Delay

Degree of Freedom	*Path Setup Delay*	*Data Through Delay*
1	5.7 ns	3.0 ns
2	10.3 ns	5.7 ns
3	14.6 ns	8.8 ns
4	17.1 ns	16.5 ns
5	29.8 ns	26.1 ns
6	56.5 ns	51.9 ns
7	114.9 ns	109.5 ns
8	243.8 ns	237.9 ns

Figure 3.9 Delay of Router as Degrees of Freedom are Added

through delay reduces to 3.0 ns. The critical path comes from the fact that the header address decode and the routing decision must occur serially

This analysis of the cost for dynamic routers indicates that a scheduled communication router could greatly improve network performance. Not only could compile-time scheduling eliminate the need for costly deadlock-avoidance hardware, but congestion could be minimized analyzing potential hot-spots in the network when scheduling the actual message routes. In addition, since no header address decode or run time routing decision need occur in a scheduled communication network, the *path setup delay* and the *data through delay* can be significantly decreased. The communication router delay can be reduced to strictly the internode delay plus whatever minimal gate delays can not be determined on previous cycles in a pipelined implementation. Compared to the previous example, this would be a reduction of network delay of sixty-six percent.

3.3 iWarp

The iWarp project attempts to take some advantage of static schedules through a unique hardware architecture [Bork90]. The iWarp project created network nodes that integrated a communication router and a processing unit into a single chip. The iWarp creators targeted their machine to high speed signal, image, and scientific computing. The processing unit consists of parallel computation units designed to carry out operations completely independent of the communication router. The interface between the two parts is a high speed register file. By integrating the processing unit and the communication unit, a very high exchange rate between the two can be achieved at the cost of processing flexibility. All iWarp nodes must have this identical processing capability. Figure 3.10 illustrates a single iWarp node.

The iWarp network combines two very different methods for processor communication. iWarps attempts to combine the efficiency of systolic routing with the generality of dynamic message passing. Each iWarp node has eight unidirectional ports that can be configured in an arbitrary manner. A simple example would be for two of the ports to be connected to each of four neighbors to form a two dimensional mesh.

In order to support systolic routing, the iWarp system allows for channels to be created between computation units for an arbitrary amount of time. A communication *pathway* gets established by a special header word called an *open pathway marker.* These pathway markers have all of the routing information encoded in their data. As this marker passes nodes that are to be included in the systolic path, logical channels and an associated buffer register get reserved. Once the path is completely set up, words can be transferred at very rapid rates since no data dependent decisions need be made. When the source processing

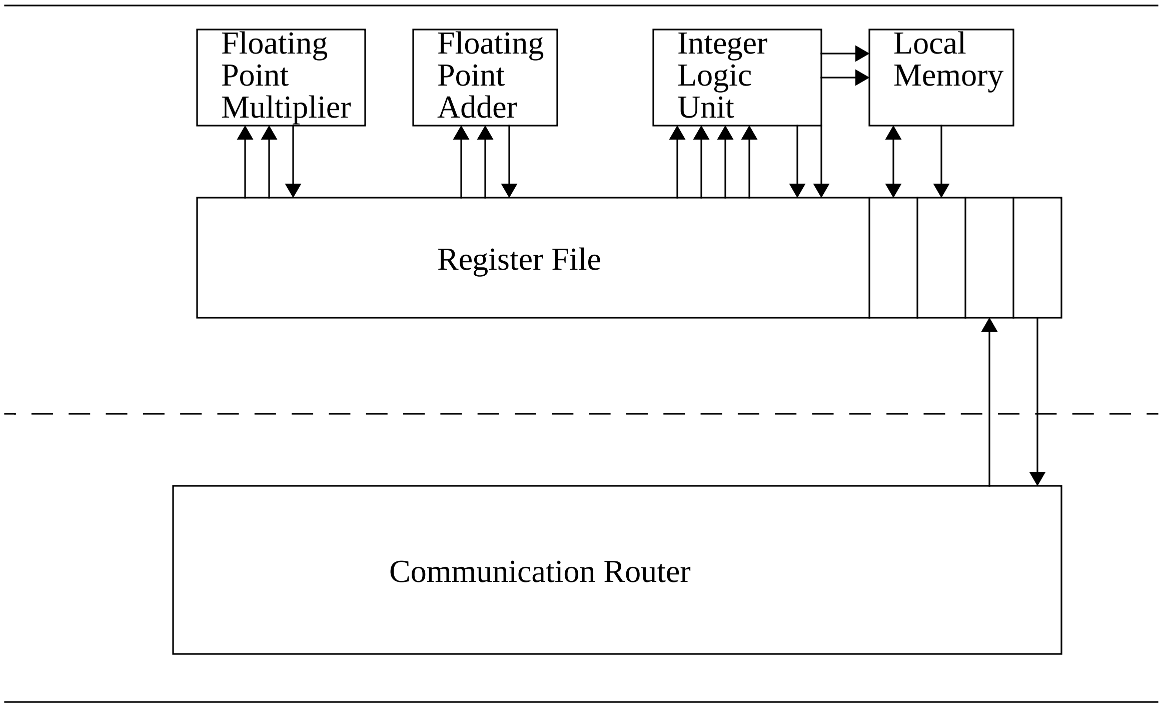


Figure 3.10 Major Units of the iWarp Processor [Borkar]

node wishes to destroy the *pathway*, it sends a *close pathway marker* that frees up each of the reserved resources on each node. Computation units can communicate with the *pathways* through a set of registers located in the register files. These registers can be read and written at arbitrary times by the computation unit. This means that although a systolic path is set up, flow control must still be implemented to keep registers at the end of the path from being overwritten. Flow control in the iWarp system is exercised on a per word basis. In order to reserve a pathway, the *open pathway marker* contains a series of addresses that corresponds to nodes in which the systolic path will need to “turn”. When a node sees an *open pathway marker* it compares the first address against the contents of a small content addressable memory (CAM) that has been written by the computation unit. If a match is found, then the node is either the final destination of the path or the pathway must turn a corner. This information is also encoded in the top word. The node reserves the appropriate logical channel and strips off the top address. The *open pathway marker* than proceeds to the next node in its path. If no match is found in the CAM, then the node assumes that the pathway is supposed to continue in the same direction from which it entered the node and the appropriate logical channel is reserved. This scheme allows a small amount of data to be encoded in the *open pathway marker.*

The iWarp communication routers also support dynamic, wormhole routing. To do this the header of the message acts in a similar fashion to the *open pathway marker.* Links are requested by the header, and when a link is allocated to a message, all subsequent packets of a message will follow that same link on subsequent clock cycles. In the case of message passing, once the message is finished with a link, there is no need to actively close a pathway since the pathway never gets reserved for future communications. Also, in the case of message passing, the destination of the transfer is a message queue. A chief difference in the two communication models in the iWarp system comes from the fact that systolic communications occur directly through program access to a small number of registers and the program is assumed to be able to perform operations for every systolic word that is transferred. Transfer words may build up in a small queue, but the program will consume them one at a time by reading the appropriate register. In the message passing model, it is assumed that the entire message must be received before the program can begin doing useful work on the message. The messages get transferred into the local memory of the destination processor.

As paths get reserved by both systolic and dynamic communications in the iWarp system, the number of physical channels can quickly be used up. To get around this problem, twenty logical channels are defined that can be assigned to any of the physical ports in software. For example, if two systolic pathways and one dynamic message all require the same physical channel, they may be assigned separate logical channels. A round-robin scheduler then alternates between these communication requests giving each path equal bandwidth. A communication queue is assigned along with each logical channel to store the data when a particular channel is not scheduled to run. Each queue has the ability to hold eight thirty-two bit words. If a particular logic channel’s queue become full, the previous link in the path will be required to spin until queue space is emptied. These twenty logical channels can be assigned to the computation units as well as the physical channels. If a logical channel’s destination queue is filled, the scheduler is smart enough to avoid scheduling that channel as long as the queue remains filled. A large reservation table is available for the twenty logical channels that provide a mapping to physical ports.

Dynamic messages look for free spots in this reservation table when attempting to reserve bandwidth.

A variety of systolic vs message passing experiments were run on the iWarp system [Gros92]. Not too surprisingly, the more systolic paths that could be determined by the system, the better the various applications performed. Since the systolic communication paths communicate via the very fast register file while the dynamic message passing communicate through local memory, the application writers discovered they could greatly improve performance by utilizing systolic communication.

The iWarp system suffers from a couple of performance penalties. Once a systolic path is set up, data can be transferred at the maximum network rate of 40MHz. However, the *open* and *close path marker* instructions operated much more slowly. These packets travel through the system at 10MHz when going straight and 6.66 MHz when turning in the network. In addition, the computation unit must create these markers by appending several addresses into a single marker. The computation units can create these addresses at a rate of 50ns per address that causes a route to turn. This overhead can be significant if many systolic paths are being set up over the course of an application. It also affects the latency of the first word in a systolic communication. There is no capability in the iWarp system to allow for communication paths that use the same link to get assigned different bandwidths. Suppose there is a systolic path set up that is utilizing nearly one-hundred percent of the physical channel bandwidth. If a low throughput dynamic message tries to utilize the same physical link by requesting one of the free logical channels, it will immediately be assigned half of the physical channel's bandwidth by the scheduler. There is no mechanism in dynamic messages that tries to avoid hot spots or congestion in the network. Ideally, one would like the dynamic messages to be routed to less congested portions of the network. Deadlock is also possible among the dynamic messages, so virtual channels must be utilized to provide protection against this. Finally, the computation model requires the computation unit to spin when waiting for a message or systolic word to arrive. This prevents the unit from responding to other communication streams that may be routed to the same computation unit and it requires that all data arrive sequentially.

The iWarp system provides an interesting compromise between traditional systolic routing and dynamic message passing. A variety of software compilers for C and FOR-

TRAN were developed for the programming of the iWarp system. Parallel program generators were developed for image processing. The iWarp system demonstrates that compile-time knowledge of systolic paths can be exploited to yield superior performance, especially for scientific and image processing applications. It also demonstrates that dynamic message passing and static routing could exist in the same environment to handle cases that are not predictable at compile time.

3.4 Scheduled Communication

To avoid the potential hazards associated with dynamic routing, an alternative scheme has been developed based on the idea of scheduled communication. Scheduled communication guarantees congestion-free paths between communication nodes based on a pre-compiled communication schedule. Flow control can be implemented to prevent uncertainties at either end of the path, but available throughput is guaranteed to be constant.

3.4.1 GF11

One of the earliest supercomputers to utilize scheduled communication is the GF11 [Beet85] computer designed at IBM in 1985. It was designed for numerical solutions of problems in quantum chromodynamics. This machine has up to 576 floating point processors connected by three stages of a Benes network called the *Memphis switch*. The processors operate on the same instruction with different data in a Single-Instruction Multiple-Data architecture. The three stages of the Benes network consist of 24 24-input muxes connected by a “perfect shuffle” through the middle stage. Figure 3.11 shows a picture of the Memphis switch.

A total of 1024 possible configurations can be loaded into the Memphis switch allowing a wide variety of communication patterns to be supported. For most applications, only a small number of these configurations need to be supported. If a two dimensional mesh of nodes is needed and only nearest neighbor communication is allowed, a very simple connection pattern can be programmed into the crossbars. Simple mechanisms exist to allow the different crossbars to switch between configurations very quickly. While it is possible to dynamically compute and load an entirely new configuration during run time, the overhead makes this action prohibitive. It is expected that configurations will be loaded exclusively at compile-time for efficiency’s sake.

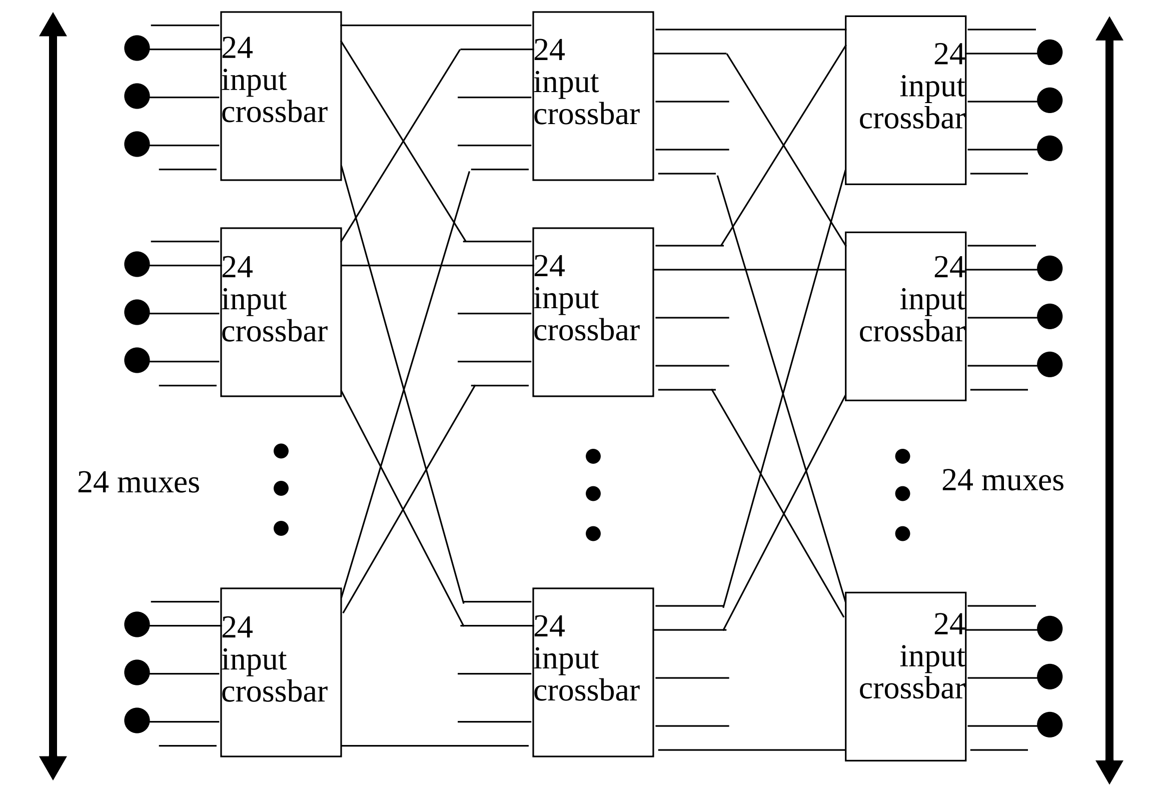


Figure 3.11 GF11 Memphis Switch

The GF11 performs quite well for the types of applications for which it is designed. Compared to the fastest general purpose multiprocessor of its day (Cray I), the GF11 completed its application in one one-hundredth the time.

3.4.2 Bianchini-Shen Algorithm

A number of projects attempt to schedule communication once the communication problem has been represented in a direct graph. The directed graph consists of a representation of all the processing nodes in an application combined with connections between the processing units that represent communication requirements. In many cases these scheduling algorithms work to map the processing units to physical nodes and assign communication bandwidth to the various messages that might be created. Of course, these algorithms require that all communications are specified at compile-time. An example directed graph is shown in Figure 3.12.

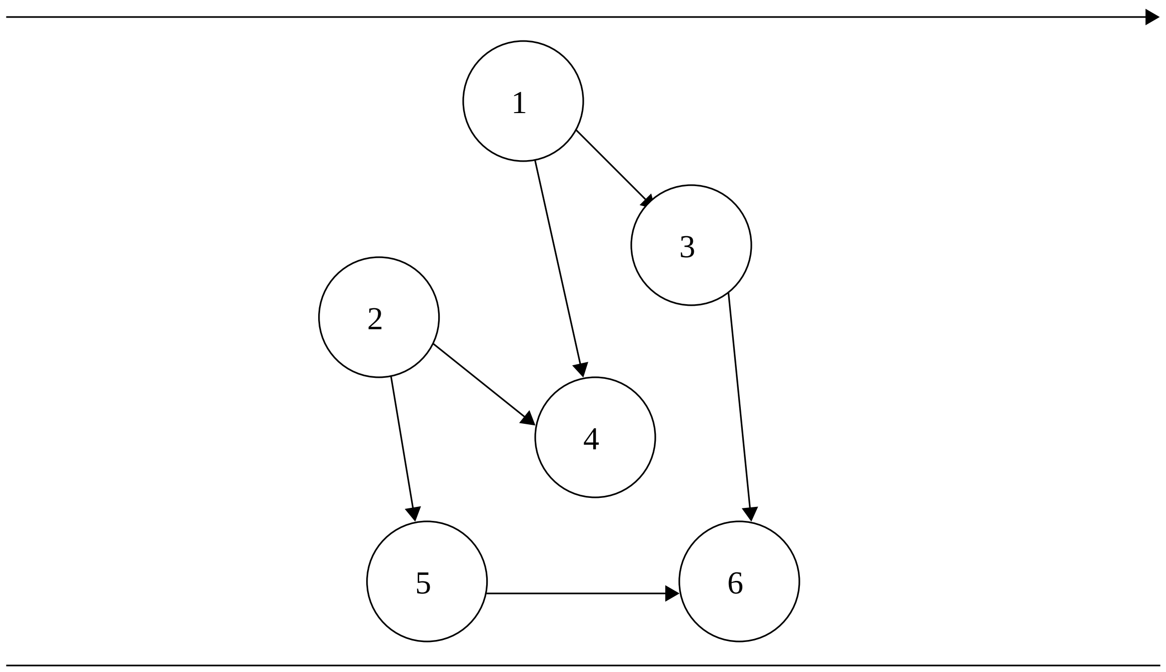


Figure 3.12 An Example Directed Acyclic Graph (DAG)

In [Bian87], the authors developed an efficient polynomial time scheme for reading an arbitrary directed graph, mapping the graph down to a multi-processor system, and obtaining optimal network schedules. An optimal network schedule is defined as a schedule that minimizes the sum of the link costs for all of the links in the system. To explain their model, first there are a series of terms that must be defined:

- I_{ij} - the link connecting nodes n_i and n_j.
- C_{ij} - the capacity of link I_{ij}
- f_{ij} - the volume of traffic on link I_{ij}
- u_{ij} - the fraction of link capacity used
- c_{ij} - the cost of routing traffic onto link I_{ij}
- t_{ij} - the traffic volume specified between nodes n_i and n_j

The scheduler implements an algorithm similar to multi-commodity fluid-flow. The links can be viewed as pipes and specified communication can be viewed as fluid flowing through the pipes. Fluid breaks off at different paths to equalize the pressure in the network. The scheduler works to form this equalization of pressure in the data communication network. A significant deviation from the fluid flow model results because messages flowing in opposite directions over a bidirectional link do not cancel, but must be handled separately. Once this model is accepted, the authors in [Bian87] propose two algorithms to solve the scheduling, loosely based on multi-commodity flow.

The first model is *proportionate traffic build-up*. Some small number of traffic volume is first scheduled. For every link, f_{ij} is calculated based on the messages scheduled during this initial stage. Since C_{ij} is defined by the network, u_{ij} can be determined. After the initial calculation, unscheduled communication gets added to the schedule. Each time a new t_{ij} is added to the network, all possible paths are considered through the calculation of c_{ij}. Each piece of message traffic is routed along the path with the least cost, and the f_{ij} and the u_{ij} are updated. The biggest difficulty in this algorithm is determining the granularity of message traffic added during each iteration. If too much is added, the cost functions become meaningless, but too small a granularity and the process takes too long. The cost function can prevent over-saturation of any link by setting a saturated link's cost to infinity.

The second model for scheduling is termed *successive route improvement*. An initial traffic pattern is laid on the links based on a simple metric such as shortest path. u_{ij} is calculated for all the links. Paths are smoothed by attempting to shift u_{ij} from highly utilized links to undersaturated links. Once no possible reroutes are found, traffic flow in the path of the over-saturated links is proportionally reduced to provide a valid schedule.

Of the two described schemes, the authors determined the *successive route improvement* algorithm to be more valuable. Unfortunately, the authors discovered the algorithm is not polynomial in computation time but is $O(N^2 \times 2^L)$ in time, where N is the total number of nodes in the system and L is the total number of links. A series of modifications to the algorithm can result in polynomial calculation time. The first important observation is that rerouting decisions can be made by only considering the most costly link in the path rather then calculating the cost of the entire path. This must be valid because the link with the highest u_{ij} in the network must be the bottleneck for the path. The second important observation involves the use of minimum spanning trees. A minimum spanning tree is the set of communication links that connect all nodes of the network such that only links with the smallest u_{ij} are used. Once the minimum spanning tree is calculated, whenever a link has u_{ij} greater than one and needs to reroute communication, one only need to consider rerouting traffic to links on the minimum spanning tree since it represents the least used connection paths in the network. The combination of these two changes to the algorithm reduces the running time to $O(L \cdot \log L)$.

3.4.3 Shukla-Agrawal Algorithm

In [Shuk91], the authors propose a scheme for scheduling based on the idea of output consistency. Output consistency demands that the communication time for transmitting messages be equal to the computation time of the node that receives the message. If the communication time is too great, the receiving node will spend a majority of its time idle, while if the processing rate is too large, messages will overflow the receiving node requiring an excessive amount of storage or blocking links on the network. One of the disadvantages of traditional wormhole routing is the first-come first served policy for resolving contention. Without knowledge of the receiving node, wormhole routing will often cause messages that are urgently needed to be delayed, while messages going to a node that has plenty of work will be forwarded. They propose a scheduling scheme that matches assigned communication bandwidth to the ability of the receiving node to process the data.

The authors start by assuming a directed graph of communication requirements exists, although the information in the graph is slightly different than in [Bian87]. Messages are scheduled according to the average processing time of the receiving node. For every communication path, a slack metric is created that attempts to measure the difference between the frequency of message transfer time versus processing time. Traffic can be assigned more or less bandwidth depending on the sign value of this slack metric. Whenever new communication paths are desired, all paths between nodes are considered, and whichever path minimizes slack in the network as a whole is chosen. While the scheduling time for this algorithm is not polynomial, the performance of the algorithm is quite good when compared with traditional wormhole routing. In benchmarks using wormhole routing, nodes are often starved for data because message throughput is not guaranteed. In the scheduled routing examples, messages arrive just as processors can consume them, preventing nodes from being idle and minimizing network congestion. Both latency and throughput are significantly better for the scheduled communication examples.

3.4.4 Dataflow Algorithms

While the previous two subsections described some of the algorithms for scheduling communication, some consideration must be taken into the design of architectures that can handle scheduled communication. Some work has been done in this area, most notably in the iWarp project described in section 2 of this chapter. Another area that has seen a lot of interest is the hardware support of dataflow architectures. Dataflow techniques traditionally map easily to block level languages. The block languages map easily to directed graphs which in turn can be handled by scheduled communication systems.

The communication models for dataflow architectures range from the fully dynamic case in which processes are created and scheduled based on run time decisions, to fully static cases where all communications are precisely determined at compile time. Some work has gone into handling intermediate models of communications. One such model is the self-timed model in which the order of processes is known, but the exact timing is not. In [Lee90], the authors discovered that when the exact order of processes is known, the access pattern for shared memories can also be extracted. In order for the exact order to be defined, the paths for communications must also be present. This information can be used to create schedules between nodes that define an order of communications, although the exact timing is not known. As long as some reasonable bound can be put on the relative timing of events, a compile-time scheduling of precedence graphs can yield fairly effective global schedules. The only additional requirement is that some sort of synchronization stage must be added to allow for the phases of communications to be changed. While the dataflow model does not perfectly fit into the scheduled communication model, there is a subclass of applications that can be handled. The authors of [Lee90] attempted to illustrate that the dataflow model can take advantage of a scheduled communication architecture, provided only limited dataflow models are allowed.

3.5 Scheduled Communication Programming Environments

The previous section shows that when scheduled communication information is extracted from applications in the form of directed graphs, there are several algorithms that have been developed to efficiently compile this information for arbitrary hardware architectures that support scheduled communications. This section shows a large body of work that has gone into creating a variety of parallel languages that allow programmers to design applications in such a way that the extraction of directed graphs from the applications becomes very easy. These source languages support a wide variety of programming styles including graphical interfaces for constructing large DSP systems, semantics to facilitate dataflow architectures, and simple language extensions that allow for general purpose programming to be mapped to a static schedule. A brief listing of some of these efforts follows:

- Programmed Communication Services (PCS) - PCS is a set of programming tools designed to allow a user to define abstract network communications at a high level that is later mapped and routed on an arbitrary physical network. It was developed primarily for the iWarp system. [Hinr95]
- Virtual Connection Facility (VCF) - VCF is a set of library routines that implement static communications, primarily for the Intel Paragon supercomputer.
- Adapt - Adapt is a high level programming language designed for image processing. It utilizes the PCS tools to express static communications. [Webb92]
- Assign - Assign is a parallel program generator that attempts to statically schedule memory accesses from the source application. The memory address space is distributed across all nodes in the system, and communication paths are set up for those nodes whose tasks require memory accesses from neighboring nodes. [OHal91]
- Fx - This is a high level programming language that allows the user to easily express data parallelism and sharing between processing nodes. It relies heavily on distinct tasks that are user-specified with data being scheduled between these tasks at specific times. [Subl93]
- LaRCS - LaRCS stands for a Language for Regular Communication Structures. LaRCS is a description language that allows a programmer to specify both static and dynamic communication information. The goal of the language is to allow for both automatic and guided mapping of parallel languages to parallel architectures. [Lo90]

• CONIC - The CONIC system was developed to allow users to hierarchically develop tasks that could be combined and mapped on to a distributed network. The user can specify communication connections at each level of the hierarchy, and the final system maps the tasks and schedules their communications across the system.[Mage89]

• Graph Description Language (GDL) - GDL is another configuration language that allows users to specify network communication. It is designed for the Prep-P system that targets reconfigurable networks. [Berm87]

• Graph Abstractions for Concurrent Programming (GARP) - GARP is designed as a set of rules to fully specify a graph of network communications in a parallel application. It is used as notation in traditional parallel languages to allow directed graphs to be extracted easily from the parallel language. There is no notion of time in the notation which limits its use for applications that have changing needs. [Kapl88]

• Gabriel - Gabriel is a digital signal processing (DSP) design environment that combines a high level graphical interface with programmer defined communication information to allow arbitrary DSP systems to be easily mapped to scheduled communication graphs.[Bier90]

• Graphic Oriented Signal-Processing Language (GOSPL) - GOSPL is a block-diagram system designed to create single-sample-rate applications based on pre-defined computation blocks. [Karj90]

• Quicksig - This block-diagram system from Lincoln Labs provides the user a LISP-like environment that combines block-level diagrams with user specified signal-objects that can be down-sampled at arbitrary points in the system. [Ziss86]

While the number of distinct programming efforts is too great to individually explain, a few representative efforts will be described to examine the different types of approaches to scheduled communication systems.

3.5.1 Programmed Communication Service (PCS) Tool Chain

PCS was developed to aid programmers in developing applications in the connection-based communication model[Hinrichs]. The primary target architecture is the iWarp system, although the authors contend the tools map easily to any scheduled communication architecture. PCS assigns resources at compile time through the use of user-defined communication connections. Once the actual application is loaded, the communication schedules are fixed across all the nodes, so run-time communication has negligible overhead. The system supports several alternatives for mapping processing units to physical processing nodes including full programmer specification as well as complete automation.

When creating an application with PCS tools, the programmer must create an *array program* as well as a *node program*. Two example programs are copied from [Hinr95] in Figure 3.13. The figure shows an array program specifying input and output ports on two

sample array program

```
outport = create_port(node(0,0),“out”);
inport = create_port(node(0,1), “in”);
nw = create_network(outport, inport);
```

sample node program

```
pcs_init();
if (self == node(0,0)) {
  port = get_port(“out”);
  send_msg(port, data);
}
if (self == node(0,1)) {
  port = get_port(“in”);
  recv_msg(port, data);
```

Figure 3.13 Example Code for the PCS Tools

of the nodes. In the node program, the actual data transfers are specified as the created ports are reserved.

The *array program* and the *node program* are analyzed in a mapping stage that assigns the nodes to physical locations. The user can specify the exact physical location of the nodes if this is desired. Otherwise, a mapping unit will attempt to place the nodes in an optimal configuration based on specified communications. At this point a complete communication graph exists, and a scheduling algorithm can compute the communication schedules for each node. At run time, the network scheduling information gets loaded to each of the nodes after a global synchronization has been established.

Now that the basic tools have been described for building blocks of communication, one can imagine creating a library of such utilities that the user could then hook together in much the same fashion. For instance, a series of DSP blocks such as filters or FFTs could be defined and then linked together in much the same manner as Figure 3.13. The PCS tools have been used for a variety of applications and have been used in the creation of several parallel program generators developed at Carnegie Mellon University. One big limitation to the system is the inability to specify array modules that vary in time. To accomplish changing needs for an application, a series of array modules must be run in sequence.

3.5.2 LaRCS - A Language for Describing Parallel Computations

LaRCS is part of the OREGAMI system that is designed to take advantage of regularity present in parallel applications [Lo90]. The OREGAMI project relies on the user's knowledge of computational power combined with efficient combinatorial algorithms to maximize performance. LaRCS is the description language that allows the programmer to specify static and dynamic behavior in an application.

The LaRCS programming language consists of three major components per application. First a series of nodes are defined that correspond to the different processes in the application. These can be thought of as nodes in a directed graph. Second, a series of communication phases are described. For each communication phase, a series of communication links are set down that define the edges of the directed graph. Finally, an ordering of the communication phases is described that corresponds to the dynamic ordering of operation in the application. The exact timing for each of the phases need not be specified, but all communication links within a phase need to be known.

An example taken from [Lo90] describes how the language works. The application is the n-body problem. Their implementation of the n-body problem assumes an odd number of bodies are located in space in a ring and are under the action of some kind of field from all other bodies. The goal is to discover the equilibrium positioning of each of the n bodies. To accomplish this, the bodies must communicate both with their immediate neighbors as well as the more distant neighbors in the ring. The algorithm works in two phases. First communication occurs with each body communicating its accumulated forces to its neighbor for (n-1)/2 steps. Then for nodes on the other side of the ring, chordal information is passed to each node's opposite in the ring. This process can be repeated a number of times for accuracy.

```
nodetype body
  labels 0..(n-1)
comtype ring_edge(i) body(i) => body((i+1))
comtype chordal_edge(i) body(i) => body((i+(n+1)/2)
comphase ring
  forall i in 0..(n-1) {ring_edge(i)}
comphase chordal
  forall i in 0..(n-1) {chordal_edge(i)};
phase_expr
  ((ring |> compute)**(n-1)/2 |> chordal |>
  compute)**s
```

Figure 3.14 LaRCS Code for N-Body Problem

Figure 3.14 shows the LaRCS code for implementing this problem. First the n bodies are numbered from 0 to (n-1). Then two types of comtypes or graph edges are defined. A ring_edge defines a nearest neighbor connection, while a chordal_edge defines a connection to the node on the opposite side of the ring. Next, two comphases of communication are defined. A comphase corresponds to a static communication graph that does not vary with time. The ring phase describes a static communication graph in which all nodes are connected to their nearest neighbor to form a ring. The chordal phase describes a static communication graph is which each node has a single connection to the node on the opposite side of the ring. The last phase of the program describes the computation. The "ring" phase is to operate for (n-1)/2 steps followed by a single step of the "chordal" phase. These phases correspond to the dynamic portion of the scheduling. For instance, although (n-1)/2 steps of the ring phase are supposed to occur, the number of clock cycles this will takes depends completely on the topology of the network and how long each communication takes. Once the n bodies are mapped to physical nodes, a scheduler will have to determine for how long each phase will have to be scheduled in order to allow the computation and communication to be guaranteed to complete.

LaRCS has a distinct advantage over PCS because it takes into account that communication needs may change over time and that each phase can be unpredictable in length. LaRCS has been shown to handle a variety of applications because it does not require all computation and communication times to be completely static. Its flexibility in scheduling phases allows programmers to see the benefits of scheduled communications while not being held to a rigid static model.

3.5.3 Gabriel

Gabriel is a DSP design environment from Berkeley[Bier90]. Gabriel applications are formed by connecting a series of block-level library models and connecting them together in a dataflow-like graph. The Gabriel system compiles the block information and creates static schedules to implement the models on DSP systems in real-time. Gabriel is driven by a graphical interface in which the user can drag-and-drop application modules to form DSP systems.

While the libraries contain a large number of block functions including filters, FFTs, up and down samplers, and counters, the user can elect to create new library functions (called “stars”) as well. The creation of a new module involves simply defining the ports, providing default parameters if needed, and providing a function call to describe the processing that needs to occur.

An example taken from [Bier90] describes a new filter with the difference equation of $y(n) = x(n) - a_1 y(n-1) - a_2 y(n-2)$. The code required to implement the star library module is shown in Figure 3.15. Library specification is completed in a Lisp-like language. **input** and **output** operators define ports on the block while **param** operators indicate arguments that can be passed to the block. The function is described as all_pole, which is subsequently defined by the **def_function**. The all-pole function assigns to the output the difference of the *in* signal and the two product terms. When calculating the product, *out* means the previous *out* value and *out@1* means the last valid *out* value going back one step. Once this block is defined, it appears in the graphical library as a block with two ports and it can be used in any system.

```
(defstar all-pole
  (input in)
  (output out(max_delay 2))
  (param a1 0.0)
  (param a2 0.9)
  (function all-pole)
def_function all-pole()
  output(difference in (product a1 out)
           (product a2 out@1)) out))
```

Figure 3.15 A Filter Implementation in Gabriel

The Gabriel system is one of many DSP modeling systems that map nicely to a scheduled communication architectures. Once the system is specified, a directed graph can easily be extracted from the block diagram. A sophisticated scheduler allows the user to map the block level diagram to arbitrary architectures with a small amount of guidance. Several real-time systems have been developed using the Gabriel system including sophisticated filters and digital modems.

3.6 Earlier NuMesh Prototypes

The NuMesh prototype FSM was a first cut attempt at defining an architecture that could adequately support scheduled communications [Hono91]. The prototype implementation supports four neighbors in a two dimensional mesh configuration. The boards were created from off-the-shelf TTL parts consisting of PALs, FIFOs, transceivers, small memories and some analog components to support system clocking. Each board was also connected to a TI DSP chip through two unidirectional FIFOs to allow words to be transferred back and forth between the FSM and the DSP. At compile time, each FSM is loaded with state information describing all of the communication that could go through the node. On each clock cycle, a single communication word can be transferred based on run time checks that determine if input data is valid and if the output port is free.

A picture of the prototype datapath is included in Figure 3.16. Each node has two transceivers that correspond to the node’s north and east ports. The south and west transceivers are located on the neighboring nodes. Two 1024 word FIFOS are used for communicating with the DSP processor.

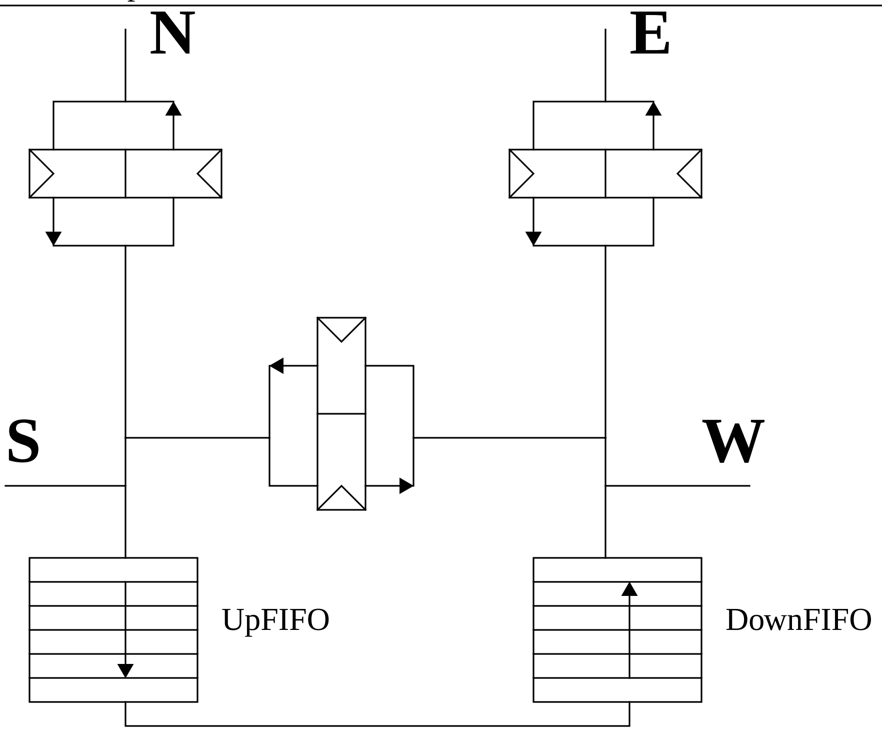


Figure 3.16 Prototype NuMesh Datapath from

A simple FSM model is used to control the datapath. Ten bits of addressing combined with four bits of condition code serve to index a RAM that holds the scheduled communication. Instructions desiring a transfer between ports spin until both the data becomes available and the destination port is free. Words being transferred in the north/south direction or the east/west direction take only a single cycle while words needing to turn a corner suffer an additional cycle of delay due to a transfer between the two busses shown in Figure 3.16. The four condition code inputs to the RAM serve to indicate whether each of the four ports contain data in the transceiver.

Coding applications involves explicitly writing communication code for each FSM. These communications must be ordered such that one communication must be observed passing through a node before the FSM is ready to accept the next communication. The exact timing of the arrival data is unnecessary since an instruction can spin until data arrives and early arriving data can be stored in the transceiver until an instruction finally sees it. There is a limited ability to change the FSM during run time through a single access register that serves to provide the control RAM an updated address. This register can be accessed through a jump instruction that causes a neighboring node to start executing from a different point in the communication schedule. An example of user code is included in Figure 3.17. In the example, data is transferred in either direction between the west and south ports. Flow control is handled explicitly in the program. Input data is checked for through the *iFulls* flag and a free output port is checked for through the *oFulls* flag. Since the data is turning a corner, the data must first be transferred through the transceiver connecting the two busses. In order for the DSP to communicate with the network, the DSP code must include commands that involve reading or writing the FIFOs.

```
Loop:
(case iFulls
   (South ())
   (West (goto FromWest))
   (else (Hold)))
(case oFulls
   (West (goto Loop))
   (else (DriveS LoadXns)))
(DriveXew LoadW)
(goto Loop)

FromWest:
(case oFulls
   (South (goto Loop))
   (else (DriveW LoadXew)))
(DriveXns LoadW)
(goto Loop)
```

Figure 3.17 Sample Code for NuMesh Prototype

While the NuMesh prototype is useful for illustrating proof of concept and for hand-coding small examples, it has several limitations that prevent it from being used in a complex programming system. One limitation comes from the FSM's inability to be broken up into multiple smaller FSMs. The sequential nature of the hardware can cause a single communication thread to block all other communication threads that go through the node. Code can be written that allows the FSM to skip threads that are not ready, but the size of the FSM grows as the product of all the states of the individual communication threads. Even limited dynamic changes to any one communication thread must, by definition, change the state of the entire FSM and upset the entire communication schedule. While this can be programmed around by keeping multiple copies of slightly different FSM structures in memory with a jump instruction switching between them, the cost becomes prohibitive when more than a very small number of dynamic decisions are allowed.

Flow control is handled through the use of *iFulls* and *oFulls* checks that occur each cycle. Since these checks involve data dependencies, a key benefit of scheduled communication is lost. Cycle times must be significantly greater than an internode transfer time because a significant amount of data dependent calculation must occur. No buffering is present to allow data to be temporarily stored in order to free a port, so data must remain blocking a port until a communication thread manages to grab it. Another restriction is that only a single instruction can operate at a time, even though there may be multiple free ports through which an additional transfer can be made.

The early system taught a number of lessons for designing scheduled communication architectures. Data-dependent instructions were shown to be impractical since they greatly increase cycle times. The ability to support multiple FSMs, either by timeslicing or through actual replication, was shown to be vital in order to keep any one communication thread from becoming a bottleneck. Figure 3.18 demonstrates two types of communications that can benefit from timeslicing and physical replication of the FSM.

The FIFO interface to the processor was slow and unwieldy and forced messages to have an ID of some kind since the order of words in the FIFO could be mixed. The ability for flow control was also thought to be useful since the rate at which words could be injected or extracted from the communication network was unpredictable.

In his Master’s thesis, John Pezaris explored a number of more complicated FSM enhancements[Peza94]. Some of the more interesting ideas he explored include multiple single-threaded FSMs that can switch context very quickly, a central arbiter that decides

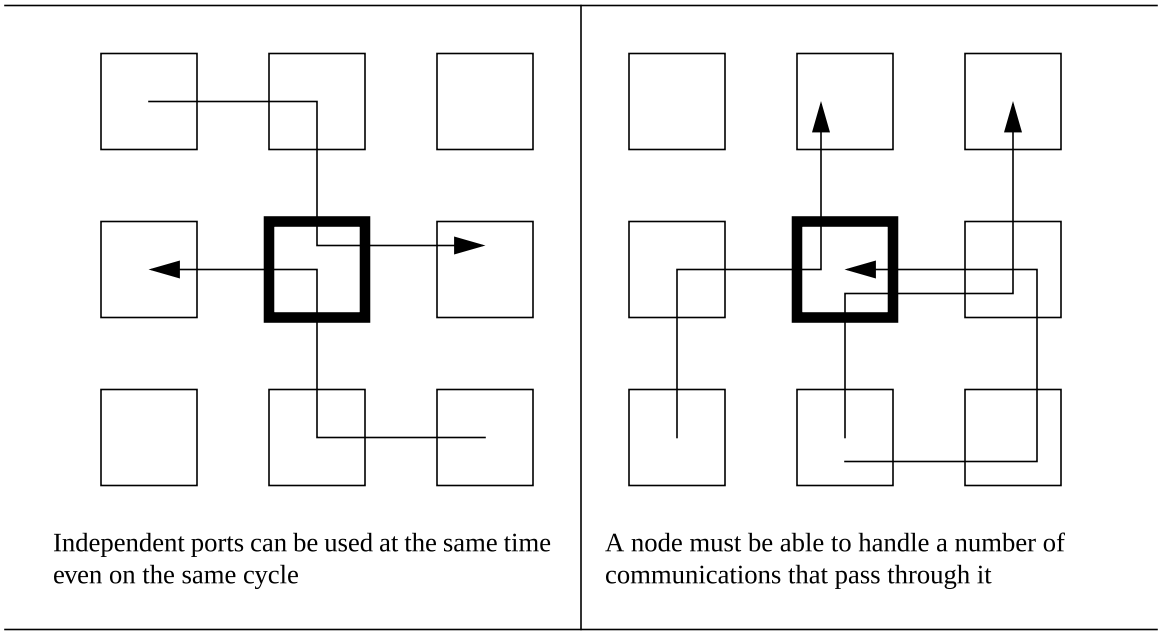

Independent ports can be used at the same time even on the same cycle
A node must be able to handle a number of communications that pass through it

Figure 3.18 Communication Patterns Can Require Virtual and Physical FSMs

which FSM threads can gain control of the ports on every cycle, and multiple processing units on the CFSM that allow computation to occur directly in the CFSM datapath.

3.7 High Speed Communication Efforts

In addition to supporting scheduled communications, another goal of the NuMesh project is to support high-speed communication from a circuit design perspective. Many efforts have been undertaken to move away from backplane busses toward point-to-point communications. Two leading efforts for supporting high speed communication are the Scalable Coherent Interface (SCI) [Gust92] and the Reliable Router Project from MIT [Dall94].

3.7.1 Scalable Coherent Interface (SCI)

The Scalable Coherent Interface [Gust92] was a joint effort among many bus designers and system architects who realized that bus-based communication schemes had fundamental limits that cause them to be bottlenecks as computation power continues to grow. The signaling rate of bus-based backplanes is limited by imperfect transmission lines that result when devices are added or removed from the bus. The serial nature of busses prevent multiple communications from occurring at the same time even though the communications may involve distinct processing nodes. The SCI standard was debated by many researchers and approved as a IEEE standard in 1992.

The original design of SCI touts three high level objectives:

- Scalability - The standard is defined in such a way that a system with thousands of processors can take advantage of it while even a simple desktop system communicating with a small number of local devices does not suffer from excessive overhead.
- Coherence - The developers recognized shared memory as a fundamental problem when connecting multiple devices together. SCI supports the efficient use of cache memories by supporting a general cache coherency protocol. This remains controversial in the community.
- Interface - The system supports an open standardized interface that allows many users to incorporate their products seamlessly into a larger system. The goal of SCI is to support not only multiprocessor communication, but internal local area networks (LANs), desktop workstation busses, shared servers, and even I/O interfaces.

The SCI standard requires two fundamental changes in the way a bus transmits information. The first problem of bus-based systems comes from the signaling rate. As more devices are added to a bus and the distance between communications increases, the signaling time greatly increases due to capacitive load or transmission line effects. In order to get around this, the SCI initiative does not wait for a signal to finish propagating before sending the next signal. When sending information over large distances on fiberoptic cables, the speed of light is the limiting factor in determining delay. In the SCI protocol, multiple signals may be on the line at the same time. An end-to-end acknowledgment tells the sender if the data arrived. The second fundamental change comes from the fact that a bus has a single path for completing all communications. The SCI standard uses multiple signal links and allows transfers between independent units to occur concurrently.

Figure 3.19 illustrates a block diagram of the SCI architecture. Data comes in from the right and the address is stripped from the header word. If the message is using the node only as a passway to get to another node, the word gets sent to the Bypass FIFO to be immediately forwarded. If the message is destined for the node, it gets sent to either the request or receive queue depending on whether it is a response to a message sent or a new message. The response unit reads words from the queue and consumes them at whatever rate it can handle. New messages can be created by the node and sent to the transmit queues. These message can either be requests for new data or responses to nodes indicating that a message has been received. By keeping the request and response queues separate, the chance for deadlock is reduced since excessive requests can never keep responses from being sent. A node must keep data around until it sees the appropriate response. SCI nodes can be connected in a variety of topologies depending on desired network performance. Low cost LANs may connect desktop computers in a ring pattern, although buffering is required since nodes may be handling multiple messages destined for separate nodes at any given time. This scheme should be much cheaper than traditional backplane connections since very few circuit boards are needed, and the same boards can be replicated for all of the devices.

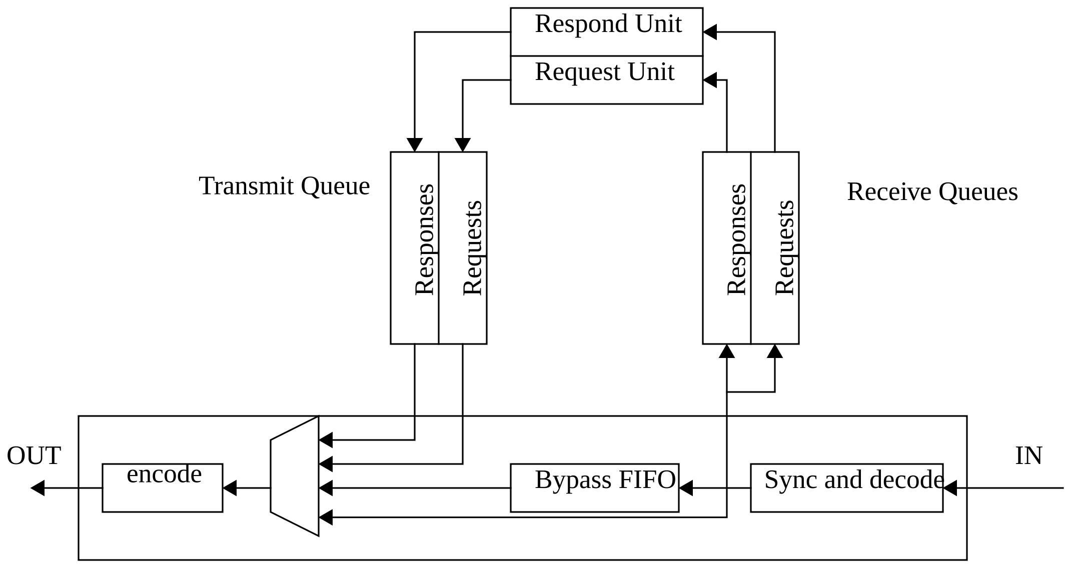


Figure 3.19 Block Diagram of SCI Architecture

The SCI standard provides a cache coherence scheme through the use of a distributed, doubly linked list of caches. The developers believe that directory-based cache coherency is a must for systems of larger sizes since processor snooping quickly becomes prohibitive. The directory contains a list of all processors that share a particular data structure. When a word is written, this list of pointers is traced so that all copies of the data structure can be updated. SCI defines a series of lock primitives that must be supported to allow atomic operations such as compare-and-swap, masked swaps, and fetch-and-adds. These primitives are needed for maintaining the directory list without the list itself suffering from the coherence problem.

The SCI initiative is an important step in attempting to define the next generation of bus design. A general bus interface allows designers with multiple processing objectives to see a standard interface to the network. The overhead of implementing SCI leaves much to be desired. Messages must have headers that are examined by each SCI node, and the designer must take care to avoid deadlocking potential in the network. SCI is a poor choice for implementing designs that exhibit scheduled communications, since it is incapable of making any decisions at compile time. Also, the idea of a forced cache coherency scheme leaves much to be desired because systems can have a wide variety of memory models. Finally, the requirement of end-to-end acknowledgment in the network significantly increases network traffic and forces nodes to needlessly keep data around and possibly stall while waiting for the acknowledgment.

3.7.2 Reliable Router

The Reliable Router is designed for two-dimensional mesh topologies and provides a unique protocol for ensuring fault tolerance in the network while simultaneously offering a unique adaptive routing protocol. Figure 3.20 shows a high level block diagram of the Reliable Router. Basically, a single 6x6 crossbar switch allows messages to be routed from any of the four ports or the processor interface to any of these same ports as an output. An additional diagnostic port can be chosen for debugging purposes. Four virtual channels are overlaid on the physical channels to support the adaptive routing scheme. Two of these channels form minimally adaptive networks, a third virtual network supports deterministic routing to break deadlock cycles, and the fourth virtual network, permits non-minimal adaptive steps for fault tolerant routing. On every cycle, one of these virtual channels is chosen for routing a message depending on the result of the communication controller. While the adaptive routing scheme chosen for the Reliable Router is not very applicable to

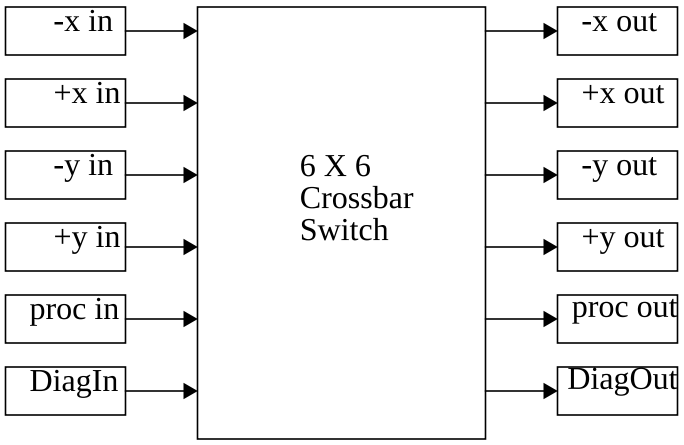


Figure 3.20 High Level Block Diagram of Reliable Router

scheduled communication, there are two ideas present in the work that the NuMesh project can explore.

The Reliable Router takes a new approach to solving the problem of global clock distribution. Each router is clocked using a different local oscillator, at the same nominal frequency. To transfer data, the clock is sent along with the data, and multiple latches on the receiver serve to synchronize the clock on the receiving end. Occasionally, dummy flits must be sent from the source to destination to keep the latches aligned.

The second novel idea in the Reliable Router comes from the use of bidirectional signaling. Data can be sent from both directions at the same time over the same wires. Very small voltage swings (~ 250 mV) are used when signaling. When data is being transferred in both directions simultaneously, the receiver must first subtract off the signal it transmitted in order to see a valid signal. A variety of techniques is used to minimize the noise in this system. The line is modeled as a transmission line, with perfectly tuned termination resistors. Since process variations can significantly alter these resistor values, the resistors can be tuned under programming control. Current is kept roughly equal in the two drivers to keep the power planes from fluctuating. Finally, when the drivers are turned on, they are staggered to prevent current spikes from causing ground bounce.

3.8 System Issues

The NuMesh project involves a fair amount of system design in addition to the design of the communication routing chip. These system decisions have a marked impact on the design of the chip and significantly affect everything from high level instruction set design down to package design. This section will discuss the background that leads to three system decisions that significantly affect the design of the CFSM.

3.8.1 Network Gap

A study out of Carnegie Mellon University [Magg95] showed that a large network *gap* could be a determining factor in network performance. Network gap is defined as the time between successive injections of messages into the network by a single processor. The study argues that while latency can be masked through prefetching or context switching, network gap presents a fundamental limitation on network bandwidth. Bisection bandwidth is usually defined as the minimum number of wires connecting two halves of the network. The author of [Magg95] argues that in practice, network processors may find it impossible to inject messages into the network at a rate that is able to utilize the maximum bisection bandwidth, and the maximum bandwidth might better defined as the maximum rate at which processors can inject messages into the network without the network becoming congested, assuming a uniform distribution of messages. The author provides a rough estimation of network gap by dividing a parallel processing system's peak GFlops by the bisection bandwidth present in the network. The numbers give an idea of the minimum amount of time that must exist between processors injecting messages into the network before the network will be flooded. A large gap number indicates that processors can quite easily flood the network unless messages are sent very infrequently. Figure 3.21 lists the estimated gaps for several popular parallel processing machines.

On the other end of the spectrum, high overhead for processors injecting messages into the network can be debilitating. Many commodity processors suffer from excessive overhead since they are not designed with network communication in mind. Parallel processing machines that utilize custom processors tend to reduce this overhead to a small number of cycles. For a machine that has reduced its gap metric to a couple of cycles, it is vital for the overhead to be low so the processors can take advantage of the network's ability to handle high message injection rates.

Machine	Processor	GFlops	Bisection GWords/s	Gap
CrayC916	16	16	15.4	1.04
CrayT916	16	30	25	1.2
CrayT3D	2048	307.6	9.6	32
IBM SP-2	128	34	.32	106.4
CM-5	1024	131.1	.45	291
Intel Paragon	1024	230.4	.7	329

Figure 3.21 Estimation of Network Gap

The author in [Magg95] goes on to show a high correlation between low network gap numbers and the ability of parallel systems to sustain high computation rates. The lesson to be taken from this is that when designing parallel systems, great care should be taken to reduce the overhead of injecting messages into the system, and the network should be able to handle high injection rates from the processors.

3.8.2 Global Clocking

Global clocking is the most widely used method for timing digital systems. Traditional clocking schemes involve starting with a single clock and fanning it out to all clocked elements through a global clock network. Great care is taken to match the delays in the clock tree. For a multiprocessor system, this scheme is not possible since the delays across nodes are far too significant and matching delays becomes impossible. Even when wave propagation methods are perfectly tuned, the clock skew is still composed of several gate delays, and there exists an added dependency on a single clock source that can be unreliable. Multiprocessor systems often attempt to allow every processing node its own clock. If these clocks need to be synchronous, as in the case for a static scheduled system, special efforts must be made to minimize skew.

A novel global clocking strategy designed for earlier NuMesh prototypes is proposed in [Prat94]. Each node in a network has an independent clock oscillator with the same nominal frequency. Clocks average the phase of all their neighbors and adjust their own phase to match. Naively averaging the phase difference of neighbors results in the possibility of a cycle occurring in which a stable situation is reached with all clocks being offset by an amount such that the cycle of phase errors adds up to 2π. Figure 3.22 shows an example of such a case that the authors call mode-lock. The situation is stable since all of a node's neighbors phase errors average to zero.

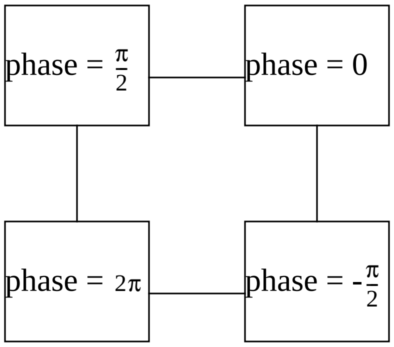


Figure 3.22 Example of Mode-Lock in Clock Distribution From [Prat94]

To remove the possibility of mode-lock, the authors submit a new error function to correct a node's phase. Rather than a linear phase error correction scheme (which corresponds to averaging the phase of all the nodes), a phase error correction of the form in Figure 3.23 is proposed.

In a 2-D mesh, it takes at least four nodes to create a cycle. If a cycle were to exist, at least one of the nodes would have to have a phase between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, since the sum of the four nodes phases is 2π. The node that has a phase in this range will be on the negative slope of the restoring graph, meaning that mode-lock is no longer a stable state. One we agree that the phases can be matched, the only remaining problem is to match frequencies. While each oscillator is of the same nominal frequency, these frequencies will not be identical due to differences in the devices. The authors in [Prat94] have devised a scheme to match frequencies of neighboring nodes through a phase-locked loop that has weak DC feedback.

The authors were able to create networks of four nodes and demonstrate phase errors of less than .1% of a cycle. Much larger networks were simulated and show similar results.

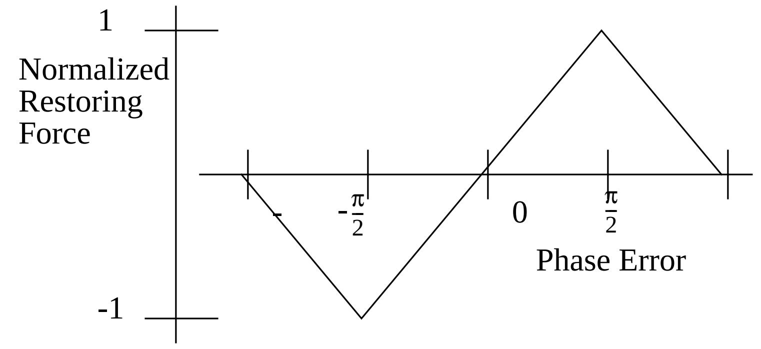


Figure 3.23 Error Correction to Break Mode-Lock

Unfortunately, the authors discovered that this clocking methodology does not work for the diamond topology. The diamond topology creates loops of nodes with up to six members. As a result, the phase error of all six nodes can be under $\frac{\pi}{2}$ and a cycle of phase adding to 2π will still exist. The authors propose another scheme for synchronization for the diamond topology. The system starts with a single node being the "leader". Under software control, other nodes in the system are told to sync up with the original node one at a time. If the order of the nodes is controlled, mode lock can be avoided since whenever a cycle of nodes is closed, all but one of the nodes can already be in sync.

3.8.3 3-D Diamond Topology

The 3-D Diamond topology represents a novel network communication design that grew out of efforts within the NuMesh architecture group [Prat95]. The 3-D diamond topology boasts of several advantages over traditional 3-D cartesian networks. Chief among these is the reduction in switch complexity for each of the nodes. In the 3-D cartesian network, each node can have up to six neighbors. The diamond network can have at most four. One key resource when designing network nodes is the pinout required for communication. This can often be the limiting resource for the network bandwidth. Assuming constant pinout between the design of a diamond node and a cartesian node, the diamond node can have fifty percent more pins per channel. The number of neighbors also affects switch complexity. At some level, a network node can be seen to implement a full crossbar connecting all input ports to output ports. The complexity of a crossbar grows as the square of the number of inputs. This crossbar area and speed can be seen as the primary contributors to the switch's complexity. Since the diamond topology is implemented with two fewer ports than the cartesian node, the complexity is much less. Even taking into account the advantage of wider channels for the diamond node, the complexity is 2/3 that of the cartesian node. Figure 3.24 shows a comparison of the two switches.

P = Total Node Pins (Fixed)
C = Number of Node Channels
W = Width of Channels = P/C
Switch Complexity = C^2W

	Channels	Channel Width	Switch Complexity
Diamond	4	P/4	4P
Cartesian	6	P/6	6P

Figure 3.24 Comparison of Diamond and Cartesian Switches

Another subtle advantage of the 3-D diamond network comes from the way multiple nodes are linked together. To form a diamond network node, four connectors sit on a plane with two connectors facing up and two connectors facing down. Figure 3.25 shows an example of a diamond node. By needing connectors in one dimension only, the insertion forces required to remove and add nodes to the network are much more manageable. Traditional three-dimensional cartesian nodes need connectors in two directions making extraction of a node much more difficult.

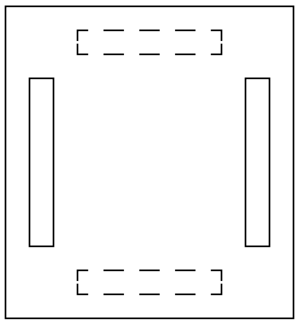


Figure 3.25 Example Diamond Network Node

The addressing for the coordinate system in diamond lattice space actually requires four coordinates, as explained in [Prat93]. However, deterministic algorithms do exist and a simple mapping from a 3-D cartesian mesh to the diamond network can be accomplished. Surprisingly, it can even be shown that in two networks composed of the same number of nodes, the diamond network can simulate the cartesian network with only a factor of three slowdown. Considering the fewer neighbors on each diamond network node, this result speaks well for the diamond network. Research into the properties of the diamond network is ongoing. Initial studies of the network show that the diamond network has many redundant paths between nodes, but that deterministic algorithms tend to create hot spots in the center of the network. A scheduled communication algorithm can take advantage of the redundant paths and significantly boost network performance.

Chapter 4

NuMesh CFSM Architecture

The NuMesh system is geared toward applications in which virtual streams of communication can be extracted at compile time. These virtual streams of communication determine sources and destinations of messages, as well as an indication of the likely frequency of communication. The NuMesh architecture takes advantage of this information and creates a statically scheduled global FSM that manages the communication over the course of the application. The virtual streams do not indicate anything about the timing of the processors, meaning that the exact time that messages are injected or removed from the network is not known.

There are two fundamental ideas in the NuMesh project that guide all decisions in both hardware and software. The first idea is that scheduled communication can be extracted from many applications. High level languages, compilers, schedulers, and the hardware must all be able to support and take advantage of scheduled communication. Software analysis of this communication can minimize network congestion and prevent a host of problems found in traditional dynamic communication systems. The hardware design must include an instruction set and hardware mechanisms that can utilize scheduled communication information coming from the software analysis.

The second fundamental idea is that hardware architecture must be able to run faster than traditional routing systems because the number of run time decisions can be drastically reduced. Also, since most of the routing information is gathered at compile time, the communication hardware can perform most operations far in advance of the actual data transfer, resulting in a cycle time that is determined by a very small number of gate delays and either an internode transfer, or a small RAM read or write. The hardware can be heavily pipelined to meet this goal. Although support for dynamic routing is desirable, the common case that the NuMesh router will handle is for scheduled communications. Any support for dynamic routing must not interfere with the higher level goal of reducing the cycle time.

4.1 Hardware Overview

At compile time, every NuMesh CFSM is loaded with a local copy of an FSM that determines the order and frequency of possible transfers that it will support. In aggregate, these CFSMs will be performing a global schedule of communication based on the data extracted from the application at compile time. Each node exchanges data between its ports and processor interface registers based on this global schedule. Every communication stream going through a node is allocated some number of clock cycles based on its bandwidth request and the schedule compiler. This local schedule forms an outer loop of control that the CFSM will repeatedly execute over the course of the application. A number of mechanism are provided to allow limited dynamic control over this schedule.

There are a number of architectural features designed to take advantage of scheduled communication. This chapter focuses in on some of these ideas and discusses trade-offs that went into their design. A brief description of the key architecture features on the CFSM follows:

• Virtual Streams - Each CFSM node has to support some number of virtual streams over the course of an application. While some of these steams can run concurrently, the rest must be time-multiplexed to give each stream it requested bandwidth. The NuMesh CFSM supports execution of two physical streams at the same time while allowing for up to sixty-four streams to be supported on each node.

• Flow Control - Since the processor timing is not precisely known at compile time, messages may back up in the communication network. To handle this case, flow control is implemented in the CFSM. Because the communication is scheduled and nodes know which virtual stream will be running in advance, the protocol involves only a single inter-node transfer time and requires only two single-direction wires.

• Buffers - In order to support separate virtual streams on every cycle as well as flow control, some amount of buffering is needed in the system. Since messages are associated with particular virtual streams, every virtual stream is provided a single word of buffering.

• Processor Interface - Although the communication schedule is static, the processors can inject and remove messages at arbitrary times. The interface between the processors and the network consists of a number of shared memory locations. Every virtual stream has assigned interface locations for insertion and removal from the network. A number of mechanisms are provided to minimize the amount of time data stays waiting in these shared memory locations.

• Scheduler - The scheduler is organized as a RAM with three fields. Two of the fields indicate which virtual FSM will run on a particular clock. The third field indicates the next scheduler instruction to be executed. At compile time, the scheduler is loaded with instructions that form an outer loop of control for the FSM. The pointer field can be written to allow for easy changes between schedules.

• Instructions - The virtual FSMs consist of a single instruction that specifies the behavior of a virtual stream through a node. The index from the scheduler selects one of thirty-two virtual stream instructions for each of the two pipelines. Each virtual stream instruction has its own buffer register that gets used if data backs up in the network.

The following sections describe in more detail the ideas behind the architectural features in the NuMesh CFSM.

4.2 Multiple Communication Streams

The number of virtual streams is completely dependent on the application being run. A node may need to support only a single communication stream that gets scheduled every clock cycle, or it may need to support many tens of separate communications that go to distinct processors in the network. Each NuMesh communication router gets loaded at compile-time with state information for all communication streams that will run through the node. In aggregate, these routers form a global FSM of communication in which a virtual stream will be guaranteed some amount of bandwidth during which physical links will be free for the stream’s exclusive use. One of the problem in supporting an architecture for scheduled communication is the organization of each node’s communication FSM. The node must be able to support a large number of virtual streams flowing through the node, and must be able to schedule each stream for an independent number of clock cycles. The first requirement allows for flexibility in the types of applications that can be supported, while the second allows each virtual stream to reserve an arbitrary amount of bandwidth.

The NuMesh prototype taught that organizing communication streams using one giant

FSM is impractical. Typically, the state required for each communication stream can be described by a very simple FSM handling a handful of states. The number of states required to track all of these simple FSMs in a single node level FSM grows as the product of the states required to track each communication stream individually. Also, trying to structure the communication schedule as a single FSM seriously constrains the ability of any single communication streams to make dynamic decisions, since any data dependent branch necessarily impacts all communication streams rather than just a single targeted stream.

The limitation in the previous NuMesh design can be addressed by restructuring the CFSM as multiple independent FSMs, each of which is assigned to track a single communication channel. This observation suggests two alternative architectural approaches.

The first approach is to timeslice a single FSM controller among a number of virtual FSMs stored in a small RAM. Although this prohibits the FSM from controlling multiple communication streams on the same cycle, it does have the advantage that the number of supported streams is limited only by the state RAM size rather than by the amount of hardware in the system. Consider the communication scheme illustrated in Figure 4.1. Five separate messages are routed through the middle node of a 3x3 mesh. Rather than organizing the communication in a single FSM, five virtual FSMs could be created with each FSM allowed control of the node for an amount of time determined at compile time. If during operation it was determined that message B needed to be changed to a different route, the virtual FSM corresponding to that node could be updated without affecting the timing relationships required for the other four messages. Since the system relies on scheduled communication, any NOP or bubble inserted into a particular node's timing can be devastating to the global schedule created for the system. By organizing the communication into distinct FSMs that get allocated bandwidth at compile time, the destruction of any one communication stream's route is guaranteed to affect only that particular communication stream's timing.

Time-slicing solves the problems of state explosion and dynamic changes to the communication schedule, but it still suffers from the limitation that only a single communication stream can operate on every cycle. Consider the communication schedule in Figure 4.2. Suppose that messages A and B request a bandwidth of one hundred percent. Assume

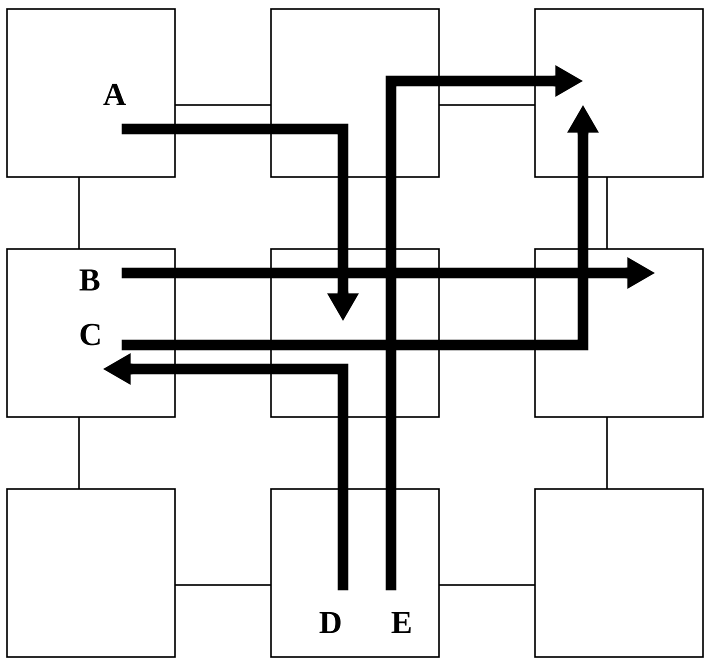


Figure 4.1 Illustration of Need for Virtual FSMs

that they can not be rerouted through other nodes due to congestion not shown in the figure. Ideally, both communications would be allowed full bandwidth since they utilize independent ports. If both communication messages are supported by virtual FSMs that are timesliced, they can at most operate every other clock cycle. One solution to this is to support multiple physical FSMs on each communication node. As long as these communication FSMs are using independent resources, they could operate concurrently to allow different streams full bandwidth. While this allows true N-way parallelism for N physical FSMs, the value of additional FSMs drops considerably as N increases. The number of useful physical FSMs depends on the number of independent resources on the node that each can utilize. Resources include the physical ports, memory locations in the processor interface, and locations within the communication node used for internal calculations. For a topology supporting N ports, one could easily imagine N messages being routed to separate processor interface locations at full bandwidth.

Strict adherence to virtual FSMs or multiple physical FSMs imposes unacceptable constraints on the CFSM capabilities. The former offers the ability for an unbounded num-

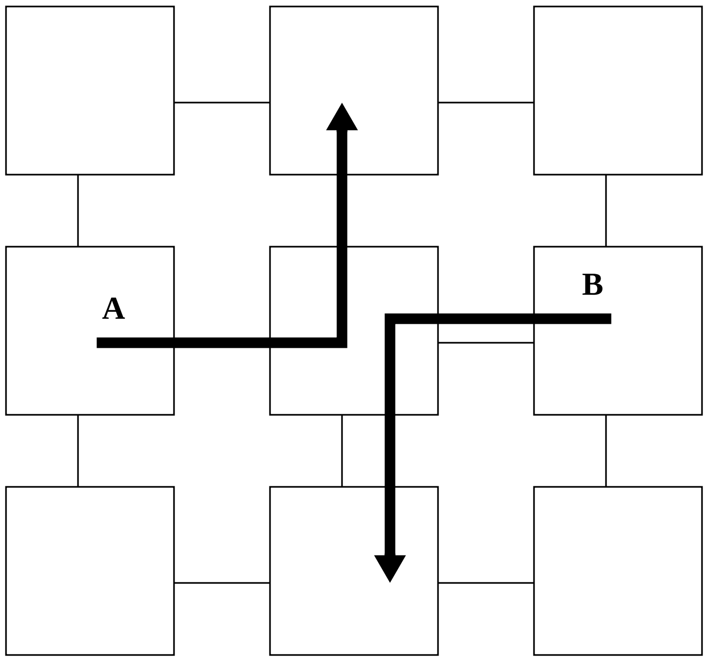


Figure 4.2 Illustration of Need for Physical Replication of FSM

ber of virtual streams but interleaves them sequentially while the latter limits the number of streams while offering real N-way parallelism. Furthermore, the natural bound on the number of communication streams and on the requirement for real parallelism stem from independent sources. The number of streams required to support an application reflects the communication structure and complexity of the node, as well as the partitioning and placement of processing nodes on the mesh. A tight limit on the total number of streams restricts the class of communications that can be supported. The requirement for real parallelism is bounded by the topology of the mesh and the internal makeup of the communication node.

For the NuMesh project, the number of physical FSMs supported is two. This was chosen largely based on the fact that the topology is based on a four-neighbor node. At any time, two independent communication streams can be routed through a node, provided they use independent ports. The added benefit of being able to simultaneously route up to four streams to the processor interface from each of the four ports is not worth the complexity and area required to implement this in hardware. The number of virtual streams in the CFSM is thirty-two per pipeline. This number was determined mainly by the size of the memory required to implement the virtual streams. The memory size must be kept down to meet the higher level goal of allowing the cycle time to be determined by either a internode transfer time or a small RAM lookup. Thirty-two streams per pipeline was also shown to be adequate for a variety of applications written in simulation.

4.3 Flow Control

Even though the communication router sets up a static schedule of communication, the processors attached to the network can remove or send messages depending on the dynamic behavior of an application. If a processor is slow to remove messages from the system, but the sending node continuously sends new communication words, words in a particular virtual stream may start to back up in the network. In order to keep words at the end of the virtual stream from being overwritten, there must be some notion of flow control implemented. Flow control implies that a transfer can be made only if the receiving node can accept new data. If data starts to back up, either a communication word must be stalled on a link, or the word must be temporarily buffered. Stalling a word is unacceptable in the NuMesh system, because on every cycle a physical link may be used by a completely different virtual stream. This means that there must exist some way to buffer words on a NuMesh node. Flow control is particularly difficult in a scheduled communication architecture, because a node can not be stalled if a word needs to be buffered. Any unexpected clock cycle in one of the NuMesh nodes could throw off the global timing of the entire communication network. This section will explore a novel flow control protocol and will discuss a variety of buffering options for the NuMesh architecture.

4.3.1 Conventional Flow Control

Dynamic networks implement flow control in one of two ways. In the first scheme, every virtual channel has some amount of buffer storage. When a node wants to make a transfer to a neighboring node using a particular virtual channel, it sends a request over a set of control lines. The receiving node checks the virtual channel's buffer storage, and sees if it can accept another word. The sending node may send the data along with this request. The receiving node then sends an acknowledgment if it can accept the new data. An important point to realize is that this protocol takes two internode transfers to complete. The transmit and acknowledge must occur in series, because the receiving node has no idea what virtual channel the sending node might want to use.

In the second scheme, the dynamic network is able to reduce the protocol to a single internode transfer. A node transmits the availability of every virtual channel to each of its neighbors. For systems using a small number of virtual channels, this protocol is manageable, although the number of pins dedicated to the protocol is proportional to the number of virtual channels. Another drawback to this second scheme, is that an extra word of buffer storage for each virtual channel must be provided that can not be used for normal buffer storage. This requirement comes about because the virtual channel is effectively sending an acknowledge before it knows the current state of the virtual channel's buffer storage. Consider the case in which the same virtual channel is being used for two consecutive clock cycles. As the first word is being transferred between nodes, the receiving node must indicate whether the virtual channel is available for the second transfer. In order to avoid two internode transfer times, the control line value must not depend on the result of the first cycle's transfer. This means that a virtual channel must indicate that it is full when it has one additional buffer spot empty. That extra buffer slot will only be used if consecutive clock cycle transfers occur over the same virtual channel. If the virtual channel is requested every other clock cycle, the sending node will see a full buffer queue even when there is still an empty slot. Between the wasted buffer word per virtual channel, and the increased decision time to interpret the virtual channel control lines, this scheme too has its drawbacks.

The NuMesh architecture is designed to support a large number of virtual streams. Since these virtual streams must follow a global schedule, if one stream gets blocked it must not affect the timing of any of the other virtual channels. A blocked communication word must get quickly buffered in order to free up the physical link for the next clock cycle when a completely different virtual stream might be scheduled. The next time the blocked stream gets scheduled, the buffered word must be quickly extracted from buffer storage and sent into the communication network. To meet these constraints, it is essential that each virtual stream has its own buffer storage. The amount of this buffer storage will be discussed in the next section. One advantage of a scheduled communication architecture is that a unique flow control protocol can be utilized. While dynamic systems suffer from either two internode transfers per word in one scheme and increased pinout and wasted buffer storage in another, the scheduled communication protocol escapes both penalties. The NuMesh protocol involves a single internode transfer, requires only a single bit line regardless of the number of virtual streams, and does not need to waste any buffer storage. The next subsection will describe the mechanics of this protocol.

4.3.2 Scheduled Communication Flow Control Protocol

In a scheduled communication system, the virtual stream that is to be run on every clock cycle is known at compile time. This information is stored in some format in the communication router. As a result, much of the overhead of an actual transfer can occur well before the data is exchanged between nodes. On every cycle, only one virtual stream is allowed to accept new data, so only a single bit of information needs to be transferred to the source node. Since the receiving node knows ahead of time which virtual stream is going to be used, it can decide whether or not it can accept new data ahead of time. In the case of a dynamic router either the acceptance of the data must occur in serial, or the state of all virtual channels must be transmitted ahead of time. In the scheduled communication architecture, the acceptance for only the scheduled virtual stream can be sent at the same time data is to be transferred.

Figure 4.3 represents the exchange of information that occurs. The receiving node B can decide on the previous clock cycle whether or not it can accept new data from the sending node A. If that stream has been previously blocked and some number of buffered words exist for the stream, the receiving node may decide not to accept new words. Node B would indicate this by sending a low value on the *accept* line when the communication stream is scheduled. During the same clock cycle, the sending node A can send its data (along with a *valid* bit) to the receiving node. No decision needs to be made yet based on the receiving node's ability to accept data. At some point in the future (possibly the same clock cycle or the next), the node can examine the *accept* bit received from the receiving node and can decide whether or not the transfer was successful. If it discovers the receiving node was unable to accept the communication word, it must be stored in a buffer that belongs to the particular communication stream.

On the following clock cycle, the receiving node becomes the sending node for the next hop of the communication stream. If no flow control backup is occurring, the word transferred to node B on the previous cycle gets transmitted to the destination of node B's instruction while at the same time node B receives an *accept* bit from the destination node.

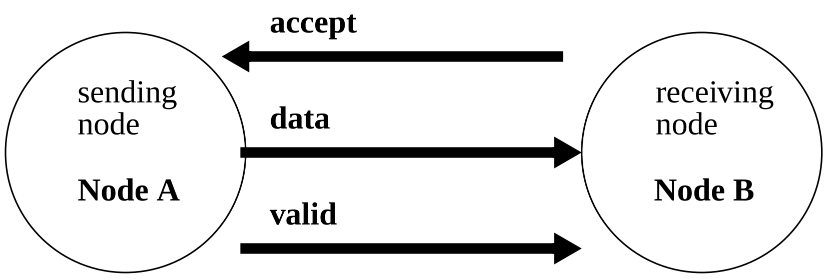


Figure 4.3 Flow Control Protocol for Scheduled Communication

Two important results should be noted from this scheme. First, only a single internode transfer time is required even though flow control is implemented. This allows unrelated virtual streams to operate on consecutive clock cycles. Second, the flow control overhead does not lengthen the cycle time of the node. Since the machine can be pipelined arbitrarily as long as the scheduled times of transfers are met, the handshake overhead can be pushed back to previous stages or future stages in the architecture. The goal of limiting the cycle time to a single internode transfer time or a small RAM lookup has been met.

4.4 Buffering

One consequence of the decision to support flow control is the requirement for storage buffers. In a scheduled communication architecture, if a word can not be transferred, it must be immediately buffered since on the next clock cycle a different communication stream may need to use the same resource as the blocked communication stream. Traditional dynamic routers might provide some amount of global storage in which a word can be temporarily stored. Since the destination address is stored in the header of dynamic packets, this scheme can simply reroute messages it finds in the buffer once the contention frees up. In a scheduled communication system, the problem is more difficult. There is no

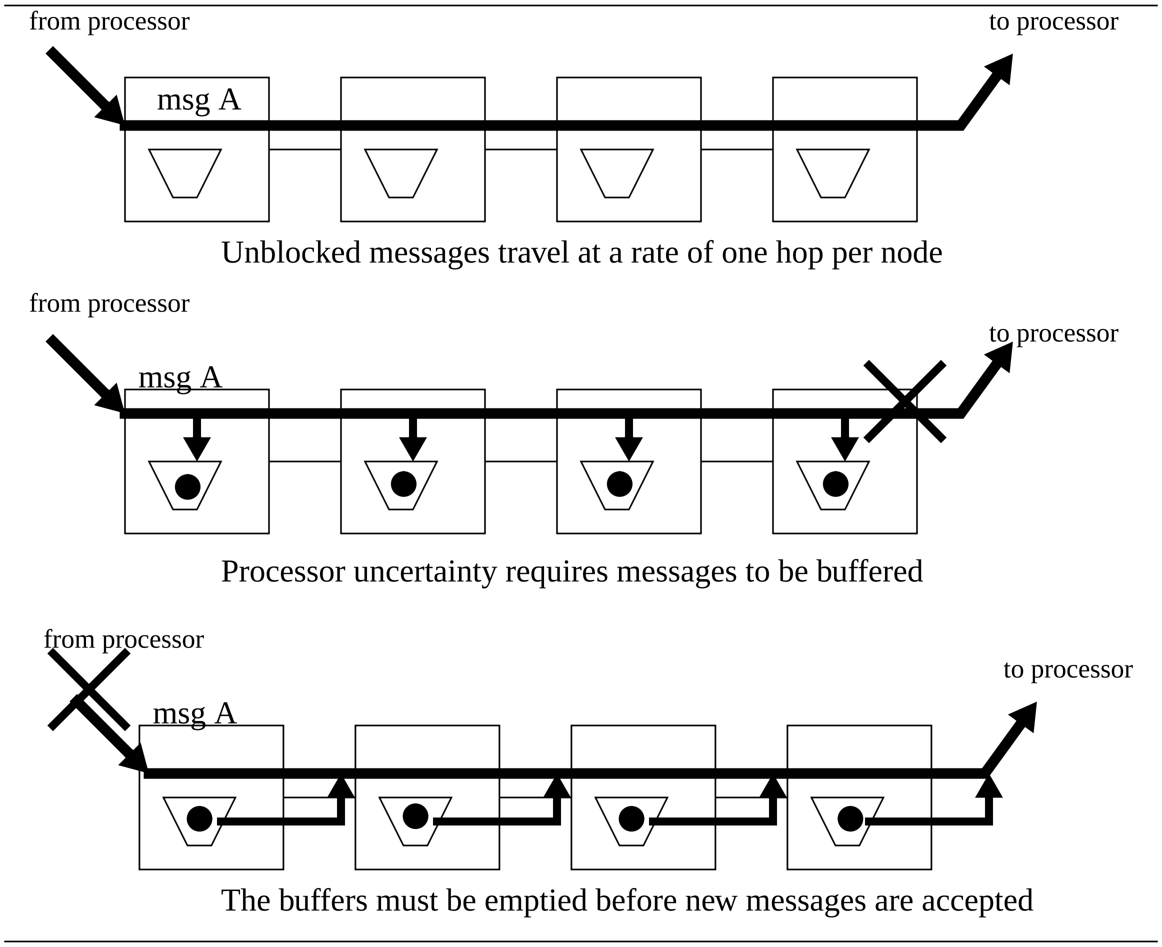

from processor
to processor
msg A
Unblocked messages travel at a rate of one hop per node
from processor
to processor
msg A
Processor uncertainty requires messages to be buffered
from processor
to processor
msg A
The buffers must be emptied before new messages are accepted

Figure 4.4 Illustration of Buffering for Scheduled Communication

destination information stored in the data. If a word in a communication stream gets blocked and stored in a buffer, the word must not be transferred again until the communication stream is again scheduled. Effectively, words must somehow be associated with the communication stream to which they belong. Figure 4.4 represents the need for buffering in a scheduled communication system. In order to operate efficiently, the architecture must allow for the buffers to be checked very quickly and the overhead in assigning and removing words from buffers must not affect the cycle time of the node.

The NuMesh solution is to assign some amount of buffering to every communication stream that goes through the node. Since the communication streams are scheduled at compile time, the storage for a particular stream can be checked before a potential communication occurs. This allows a node to indicate to its predecessor in the communication path that it can not accept new data since it already has valid data to transfer. One effect of the amount of buffering per stream is how much progress a very large message will make before it starts to get blocked. If every communication stream on every node had an infinite amount of storage, then the messages would back up at the node connected to the destination processor, allowing the processor to consume data at its maximum rate. If a single word of storage per stream is allowed, then messages get spread out across the network, until the destination processor is able to accept the single word on its connected network node. Each buffer word of storage costs a single thirty-two bit register of storage. Since a desirable feature of the architecture is support for a large number of virtual streams in order to support a wider variety of applications, there must be some limit placed the amount of buffer storage per virtual stream. The current implementation of the NuMesh supports sixty-four total virtual streams. Since a goal of the project is to keep cycle times down to a small RAM read or write, the number of buffer registers per stream must be held down to keep the register file from getting too large. The next two subsections will compare two of the more cost effective solutions to this problem. First a single word per stream will be examined, followed by two words of storage per stream.

4.4.1 Single Buffer Word Per Virtual Stream

If a single word of buffer storage is allowed for each virtual stream, and there are two physical pipelines, each containing thirty-two virtual streams, this means the node will support a total of sixty-four words of buffer storage. If more than a single word of storage is allowed per node, not only does the area of the buffer storage start to become significant, but the complexity of managing the buffers grows. If multiple words are stored on a node for a particular communication stream, the order the words arrived must be noted because there is no information in the data itself that indicates order. The datapath for the node also becomes more complicated since an incoming communication word can be routed to one of many buffer destinations as well as the four destination ports. Restricting the amount of buffer storage to a single communication word per stream simplifies the datapath and the communication complexity.

A single word of storage for each stream is the minimal solution for handling the flow control problem for scheduled communication. Words must be stored somewhere, and every virtual stream needs the capability to store a word if thing back up. However, there are two significant drawbacks to allowing only one word of buffer storage per virtual stream. The first constraint deals with the manner in which backed up words in the network get cleared. Assume a linear array of nodes as pictured in Figure 4.5. If the commu-

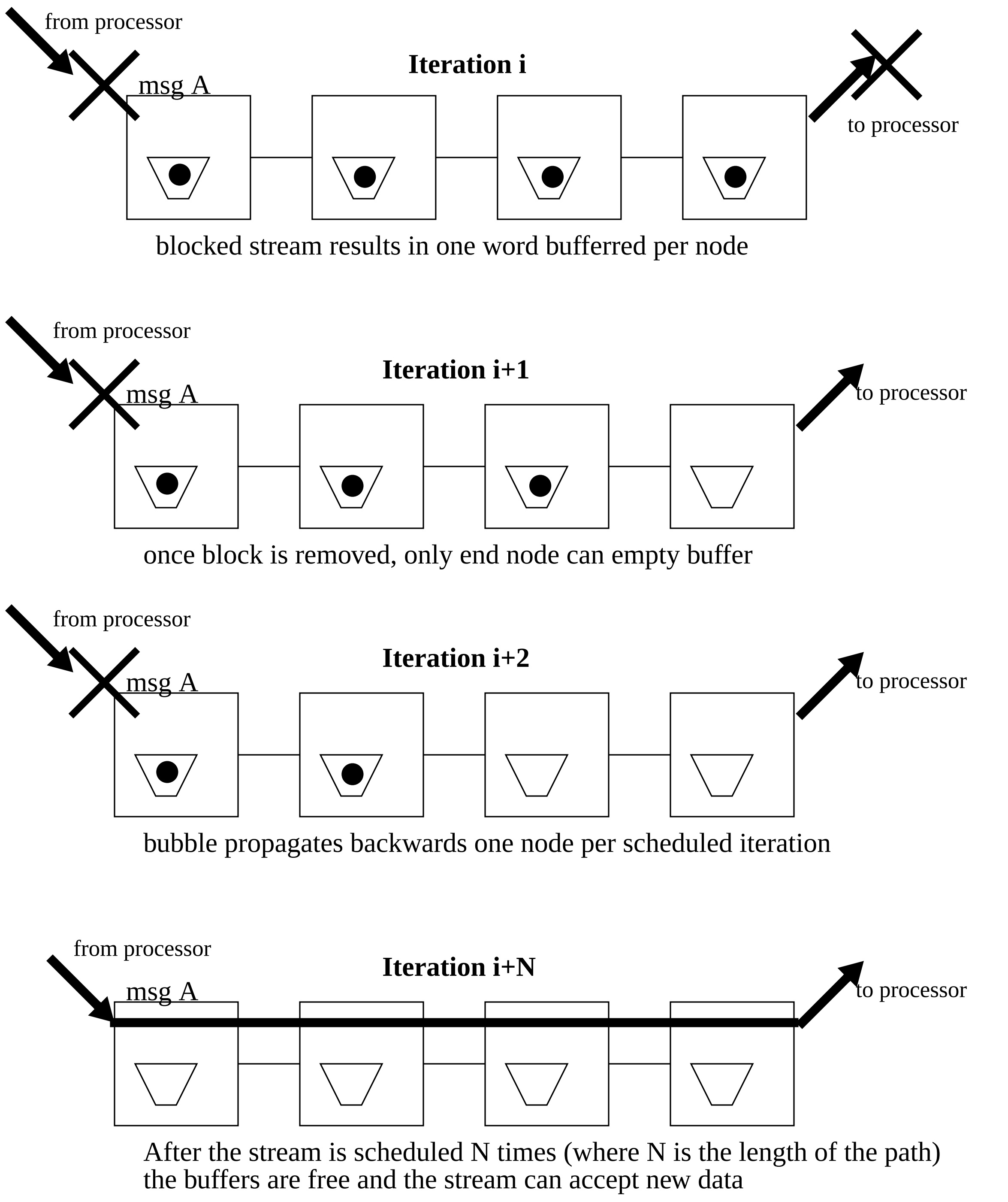

from processor
Iteration i
msg A
to processor
blocked stream results in one word bufferred per node
from processor
Iteration i+1
msg A
to processor
once block is removed, only end node can empty buffer
from processor
Iteration i+2
msg A
to processor
bubble propagates backwards one node per scheduled iteration
from processor
Iteration i+N
msg A
to processor
After the stream is scheduled N times (where N is the length of the path)
the buffers are free and the stream can accept new data

Figure 4.5 Recovering From Network Blockage

nication stream has been blocked for awhile due to the destination processor not removing words from the network, a single word will be stored in each node in the buffer register corresponding to the virtual stream. Once the processor can start accepting new words, ideally one would like a word to be read out of every node's buffer register and transferred along the path. In practice this can not happen. When the stream is scheduled, each node except for the last one will look in the buffer register of the node next in line and see a word stored in the stream's buffer register. This will indicate that a word can not be forwarded and the buffers will stay full. The last node will successfully complete the transfer since the destination processor can read the input word. The next time the stream is scheduled, again, every node but the second to last will see a full buffer on the following node. The second to last node will be able to transfer its word to the last node, but that is all the progress that can occur. In fact, for a chain of N nodes, the stream will have to be scheduled N times before all of the buffers can empty. Unfortunately, there is no faster way that these buffers can be cleared. The fundamental flaw is that a piece of information at the end of a chain needs to be propagated backwards N hops. Since the clock cycle is limited to a single internode transfer, it must take N clock cycles for this piece of information to propagate back to the first node. When a stream is scheduled, it only gets scheduled for one clock cycle, so it must take N iterations of getting scheduled before this information can arrive at the first node in the system.

The second drawback of a single buffer per stream is that it restricts the maximum bandwidth at which a stream can operate. In normal operation a stream will read from its source port on one cycle, and then write to its destination port on the following cycle. A node is allowed to do a read only if its buffer register is empty. If it were full, then the node should take its source word from the buffer register rather than take new data from another node. When a node is writing a destination port, it is discovering whether or not the destination node is able to accept new data at the same time. If it can not accept new data, the node must put the communication word into the stream's buffer storage. Assume that a stream is operating at one hundred percent bandwidth. This means that on every cycle, the node is both reading the source port of the stream and writing a destination port of the stream. In order for the read to occur, the buffer must be guaranteed to be empty. However, at the same time, the word being written may have to be stored into the buffer storage.

This means that the read operation will not know if the buffer storage is empty until after an internode transfer from the destination node. Only at that time can it send an internode transfer to the source node indicating whether or not it can accept new data. In fact, the last word of a transfer in a stream that involves many hops may determine whether or not the first node in the path can make a transfer. Figure 4.6 illustrates this problem. If the processor decides not to remove the last word from the network, then it must be stored in the buffer register of the communication stream. However, this means that the node can not accept a new data word, so the previous node must also store its data. This effect ripples backwards until the initial node is unable to accept a new word because its buffer register is full. This result is catastrophic since it requires the cycle time for a node to involve N internode transfers where N is the number of hops in the longest possible path of the network.

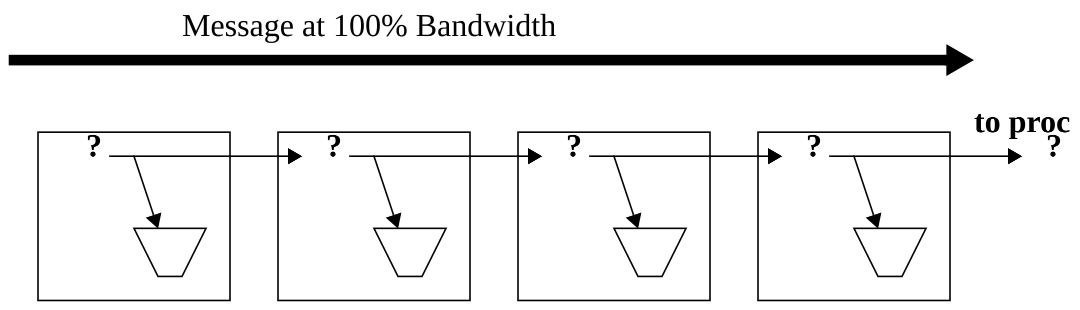


Figure 4.6 Flow Control Ripple Across All Nodes

Fortunately, there is a simple constraint that solves this problem. The same stream must be prohibited from operating on consecutive clock cycles. This ensures that the same stream will not be attempting a read and a write at the same time on any one node. This serves to break the cycle of dependencies that occurs since the result of a failed write no longer affects the acceptance of new data during the same clock cycle. The obvious downside is that streams can not be allocated more than fifty percent bandwidth. Since many applications might require streams of very high bandwidth, this constraint is unacceptable. However, there are ways around this problem. For instance, one communication stream may be broken up into two identical communication streams having a bandwidth of fifty percent. The scheduler could then alternate between the two streams, and the effect would be identical to a single stream with one hundred percent bandwidth. While some care must be taken to prevent out of order communications, this scheme can be made to work and is discussed further in section 4.6 of this chapter.

4.4.2 Two Buffer Words Per Stream

Allowing for two words of buffering per virtual stream has interesting impact on both problems presented by the single buffering scheme. Again assume that a stream has been blocked and the all buffer registers are filled on all the nodes of the communication path, as illustrated in Figure 4.7. When the destination processor unblocks, again this information can only flow back a single cycle, and only the last node of the path can transfer a word. The next time the stream is scheduled, the second to last node can transfer a word, and the last node can forward its second buffered word. The bubble of information still propagates back one node per clock cycle, but the destination processor can continue to receive one communication word per scheduled cycle. At the end of N schedule iterations, the bubble will have propagated back to the source node, and every buffer will contain only a single word. At this point the source processor can begin injecting words into the stream again. As in the single buffer case, it takes N iterations before the congestion is eliminated, but in this example, the destination processor can receive data on every scheduled iteration.

Allowing two buffers per stream also allows for an interesting solution to the problem of scheduling a stream every clock cycle. If the buffers are implemented as described in the previous paragraph, than the problem remains. To show this, imagine every node having one word stored in its buffer while attempting to read and write other words in the same stream. There is still a dependency on the one free buffer slot in each node that is exactly equivalent to the case in which there is one buffer per stream.

However, there is a solution that allows the same stream to operate every clock cycle. Assume that the flow control protocol is modified such that a stream does not assert its *accept* bit unless there is no word stored in the buffer. If a stream is scheduled with one hundred percent bandwidth, it can be trying to write and read a word at the same time. It will only try to read a new word if the buffer is completely empty. Even if the write eventually fails and the node reads a new word, both words can be stored in the buffer registers

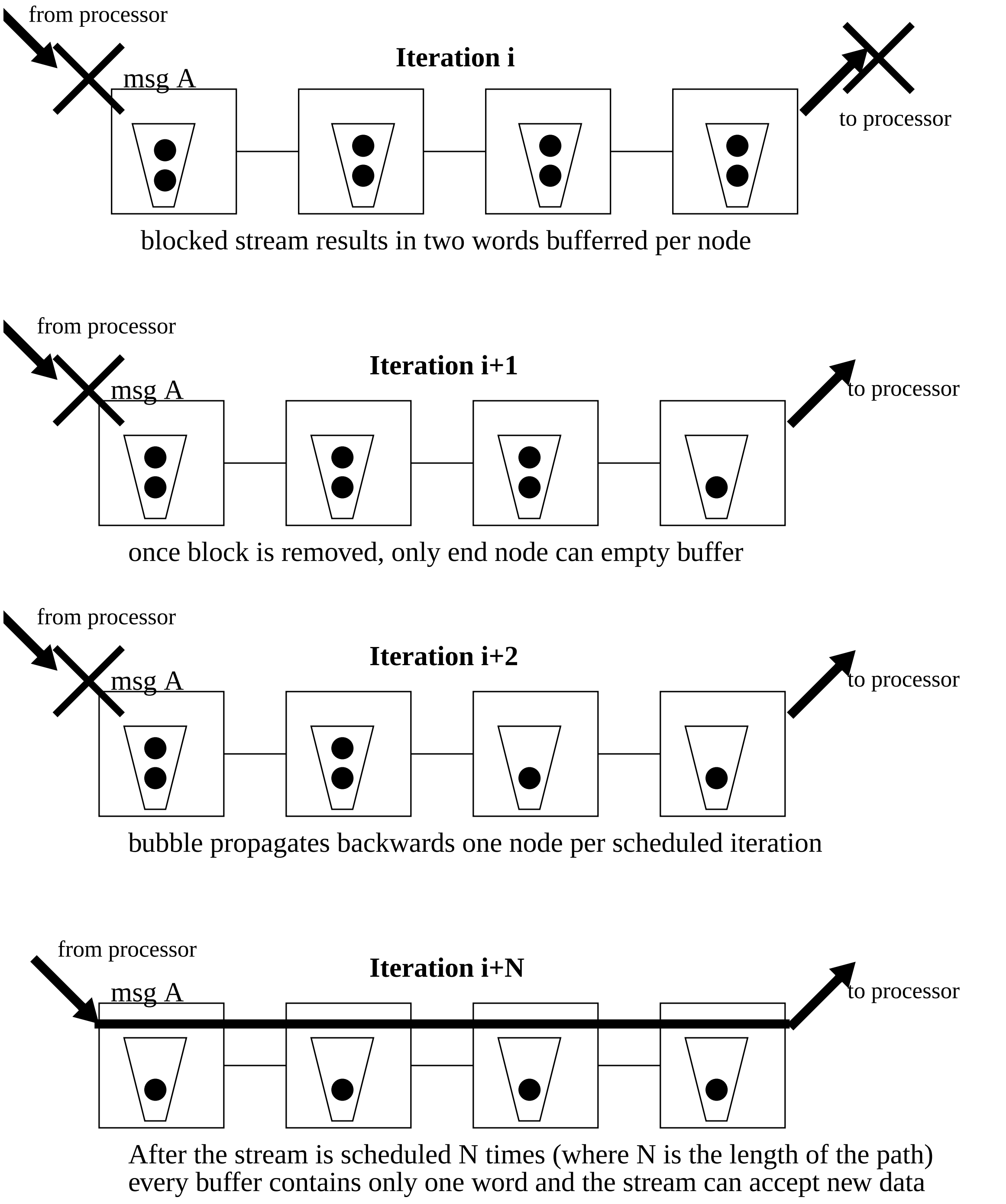

from processor
Iteration i
msg A
to processor
blocked stream results in two words bufferred per node
from processor
Iteration i+1
msg A
to processor
once block is removed, only end node can empty buffer
from processor
Iteration i+2
msg A
to processor
bubble propagates backwards one node per scheduled iteration
from processor
Iteration i+N
msg A
to processor
After the stream is scheduled N times (where N is the length of the path)
every buffer contains only one word and the stream can accept new data

Figure 4.7 Network Blockage Recovery For Two Buffer Case

since there is room for two words. If there is already a word stored in the buffer, than the read will indicate that the node can not accept new data, regardless of the success of the write operation. This scheme prevents the rippling of information across several nodes, because the read operation of the stream does not depend on the success of the write operation. This scheme has the negative effect, that for streams scheduled at less than one hundred percent bandwidth, both buffer locations can never be used. Since a stream is marked full when it contains a single word in its buffer, no new words will be accepted even though there is still a buffer spot free.

4.4.3 NuMesh Buffer Scheme

While the two buffer per virtual stream idea clearly adds utility over the single buffer per stream idea, this implementation of the NuMesh uses only a single buffer. Since the system currently supports sixty-four virtual streams, even the single buffer word scheme requires sixty-four words of storage. Since these buffers have to be accessed every cycle, keeping the size of the buffer register file down seems worth the loss in utility. Also, allowing two buffer words per stream complicates the overhead in managing the streams. Given a larger chip size and faster RAM, the trade-off may be worth it. Ideally, one might even consider allowing the assignment of buffers to streams to be under software control.

The addition of a single buffer per stream allows for a very simple flow control protocol to be implemented. The flow control in no way affects the timing of the scheduled communication and does not add to the clock delay of the node. This protocol is unique to scheduled communication and only works because all possible transfers are known in advance, and work can be done ahead of time to limit the transfer of data to a simple internode transfer. No data dependent decisions need to be made so communications can occur at top speeds.

4.5 Processor-Network Interface

Although the NuMesh architecture will support virtual streams communication by orchestrating a static global schedule, the times at which processors will inject and remove messages from the system is unknown. The communication network serves to reserve clock cycles during which only the scheduled virtual streams will be allowed to communicate. There is an interface between the processors and the communication network at which the two must become synchronized. A processor may be ready to inject a message before a virtual stream is scheduled to communicate. Similarly, a message may reach its destination before the end processor is ready to remove the message from the network.

These transfers take place through memory locations that are shared between the processors and the communication network. Each virtual stream can have its own set of locations on the beginning and end node of the virtual stream where words are transferred between the dynamic processors and the scheduled communication network. Once a communication word gets transferred to the last node of a virtual stream, the word will get written into this shared memory space, and the attached processor can attempt to read the word out of the network whenever it is ready. Similarly, when a processor wants to send a message, it simply writes the shared memory location corresponding to the correct virtual stream. When the virtual stream next gets scheduled, the word will be read out of this memory location and transferred through the network according to the pre-defined schedule. The number of locations reserved for each virtual stream is an architectural parameter that can be adjusted by the designer. One can imagine a FIFO of several words for each virtual stream that would allow the processor to inject several words into the network before the stream finally gets scheduled, and would allow several words to arrive at a destination processor before the processor needs to read the incoming words. On the other hand, multiple words of shared memory per virtual stream can get very expensive, especially if the architecture supports a large number of virtual streams.

The current version of the NuMesh project supports sixteen separate locations of shared memory address space. This means a node can support sixteen virtual streams that either start or finish at any one processor in the system. Since the total number of virtual streams supported by one node was set at two pipelines of thirty-two streams each, the number sixteen was arrived at by assuming that roughly one out of every four virtual streams going through a node could either start or finish at the node. Adding support for more virtual streams is a reasonable thing to do, but for this implementation it made the register file interface larger and more complicated. Since each of the pipelines and the attached processor must use these registers, the shared memory is implemented as a three-ported RAM.

In this implementation, each virtual stream interface register can only support one word of communication. This means if a virtual stream injects a word into the network, but the virtual stream is not scheduled immediately, the processor can not inject another word into the same stream until the previous word is accepted by the communication network. Adding multiple words or small FIFOs for each virtual stream interface would allow for greater decoupling between the processors and the communication network by allowing multiple words of data to back up before needing to be removed. For the current implementation, the cost of this was considered too great since the number and complexity of the memory registers started to become significant.

4.5.1 Synchronizing the Processors and Communication Network

Simply providing shared memory locations between the processors and the scheduled communication network does not mean that the application will behave as desired. For instance, assume that at compile-time a virtual stream is given a bandwidth of one percent. This would mean that one out of every hundred cycles would be reserved for the communication. Since the processor behavior is dynamic, it could inject a word for this stream one cycle after the stream is scheduled. This would mean that the communication word would sit in the interface for ninety-nine more cycles until it is scheduled again. Its effective latency would go significantly up. Unfortunately, there is no easy solution to this dilemma. One solution is to assign the virtual stream a much greater bandwidth than the processor can fill. If the message were thought to be important enough, it could be assigned a bandwidth of ten percent. Nine out of the ten times the reserved bandwidth would go wasted since the processor is only injecting a new message every hundred cycles. However, the one time the message is injected, it will only wait a maximum of nine cycles. The bandwidth metric can reflect not only the rate at which the processor will inject messages into the system, but also the rate at which the system resources will be dedicated to the stream’s transfer.

Synchronizing incoming words occurs a little more naturally. Suppose a processor expects a word to arrive once every hundred cycles. The network can provide the stream a one percent bandwidth. Now assume the computation has a hundred cycle loop and tries to read from the shared memory interface for one out of every hundred cycles. Again, things can line up such that the processor attempts a read a cycle before a word arrives in the interface, causing the word to sit in the interface for the next ninety-nine cycles. However, in this case, the processor can choose to spin until a word arrives. Once the processor gets a valid word, the computation and communication cycles can be in sync, since there will be ninety-nine more cycles of computation before the next read is attempted, and this corresponds exactly to the amount of time it will take until the virtual stream is scheduled to deliver a new word. Once the computation and communication get in sync, they can stay in sync as long as the communication network can deliver words and the computation unit keeps from suffering dynamic delays.

4.5.2 Communication Interface Mechanisms

An application can support a number of different models of communication. Some applications might support very infrequent communication. For these cases, the possibility of the processors and communication network syncing up is slim. Some applications may have several virtual streams all routed to the same destination. For these cases, the possibility of the processor managing to stay in sync will all messages is small, and some messages are bound to spend time sitting in the shared memory locations. One of the goals of this architecture is to provide mechanisms that reduce the number of cycles that messages spend in the shared memory interface.

One mechanism is to allow interrupts to occur whenever one or more messages have arrived. At compile time, a set of input and output conditions can be set. These conditions specify that whenever any combination of shared memory locations are either full or empty, an interrupt occurs. This model works well for applications that communicate infrequently. The computation can occur on the processing node uninterrupted until new messages arrive at the node. A new message causes an interrupt, and a handler can discover which virtual stream has a new message. The interrupt handler can then read the new message and update the computation accordingly. Although interrupts are traditionally slow, this mechanism prevents the code from constantly needing to poll the shared memory registers.

A second mechanism allows the processor to read the state of all of the shared memory registers at the same time. By examining the validity of all of the registers at the same time, the application code can dispatch to various routines based on which messages have arrived. This mechanism handles applications that have several virtual streams routed to the same destination. Since the sending processors are unpredictable in the frequency of injecting messages, the destination node can check on all incoming message locations at the same time and only service those that have delivered new data. The synchronization problem is avoided by allowing the processor to handle only those messages that have arrived. After the messages are consumed, the processor can again check the state of the shared memory interface to determine which virtual streams delivered new messages.

The third mechanism supports the kind of synchronization described at the beginning of this section. A single shared memory location can be read or written, and the processor can choose to stall if no word has arrived or if an outgoing slot is still full. Over time, the processor and communication network might naturally sync up, since the processor can stall until the relative timings are in phase.

These interface mechanisms provide a fair amount of flexibility in the communication model of the application. Applications may choose to use some combination of all three mechanisms over their operation. The NuMesh project goal of decoupling the static timing of the communication and the dynamic timing of the processors makes complete synchrony difficult. The described mechanisms simply attempt to reduce the penalty of what is inherently a difficult timing problem.

4.6 Scheduler Architecture

Since it has been decided that there will exist support for multiple virtual streams on each node, there must be some mechanism for choosing which virtual stream will operate on every clock cycle. Each virtual stream through a node can be reduced to a single instruction that gets executed every time the stream is scheduled. An outer loop of control decides which virtual stream will operate on every clock cycle. One might imagine a large RAM with a pointer that simply increments and reads out an instruction on every cycle. If a stream were to be scheduled with fifty percent bandwidth, the instruction would appear in every other slot of the RAM. However, there must be some way to distinguish between the same virtual stream being scheduled twice and two separate streams that have the same behavior through a node. The difference comes from the associated buffer storage. The first case should use a single buffer word of storage, while the second needs two separate buffer locations. When a word gets buffered, it needs to stay in the buffer until the stream gets scheduled again. Even though two separate streams have the same path through a particular node, they might go down completely different paths at later nodes. If these two streams were to share the same buffer location, words might accidentally be transferred between the streams.

To avoid this confusion, there is a separate piece of hardware called the scheduler. Every virtual stream gets an independent instruction, even if it is a direct copy of another instruction. The job of the scheduler is to choose which virtual stream will be run during each clock cycle. Since there are two separate physical pipelines, the scheduler picks two virtual streams to be run every clock cycle. The scheduler can assign any bandwidth to a communication stream, although the scheduler must follow the pattern of communication set up at compile time. Figure 4.8 illustrates the operation of the scheduler.

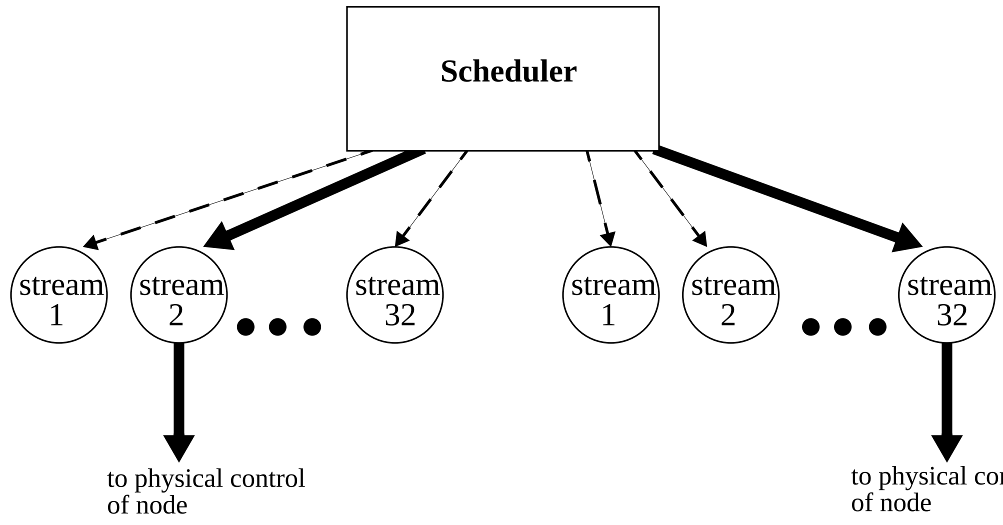


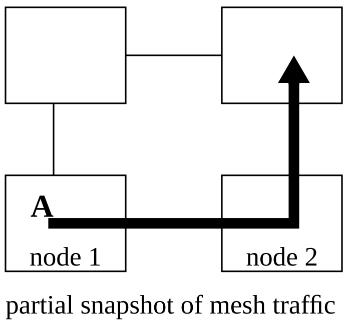
Figure 4.8 Scheduler Operation

At compile time, the scheduler is loaded with the information indicating the pattern of stream assignments that is to be followed. The scheduled information defines an outer control loop of a fixed size, with each stream being scheduled some percentage of the loop. Although variations are possible, typically all nodes in a mesh will have scheduling loops of the same size. A third entry in the scheduler chooses which scheduling instruction will next be executed. Since the scheduler is always executing a loop of a certain size, one might expect a counter to simply increment the scheduler’s instruction and simply loop back to the beginning when a special control bit was present. However, it is possible that multiple schedules could be present in the scheduler, and this third field could be written by the processor to allow an easy mechanism to switch between separate schedules.

A high level compiler will be responsible for generating these schedules and making sure that communication streams involving transfers between nodes will be scheduled in the appropriate slots. Figure 4.9 demonstrates the scheduling constraints across two nodes. If Message A requests a bandwidth of ten percent in a schedule with a loop size of ten, then it will be allocated one of the ten clock cycles. Assume for node 1 the slot chosen is the sixth cycle. This requires that message A be scheduled in the seventh slot for node 2 to ensure that the message can travel through the mesh with a single clock latency per node. If the stream were scheduled on a different clock cycle in node 2, the exchange of data would never occur.

Schedule for Node 1

clk cycle	stream
1	stream C
2	stream D
3	stream C
4	stream F
5	stream C
6	**stream A**
7	stream C
8	stream F
9	stream C
10	stream D



Schedule for Node 2

clk cycle	stream
1	stream E
2	stream G
3	stream B
4	IDLE
5	stream G
6	stream B
7	**stream A**
8	stream G
9	stream B
10	IDLE

Figure 4.9 Effect of Scheduling a Message on Multiple Nodes

For every clock cycle, two independent streams can be scheduled. At compile time, these slot are attempted to be filled as much as possible. Any scheduling slot that does not receive a scheduled stream results in a NOP being scheduled. No communication will occur in the pipeline for that clock cycle. Idle cycles in the communication schedule results in lower communication throughput for the network. One of the hardest jobs of the compiler is to avoid the idle cycles. Fortunately, there are tricks that can be played to reduce them. Assume that the compiler generates the schedules for a node shown in Figure 4.10. The first pipeline requires a schedule length of three while the second pipeline needs

Clk Cycle	stream
1	stream A
2	stream B
3	stream A
4	stream C

Pipeline 1

Clk Cycle	stream
1	stream D
2	stream E
3	stream F

Pipeline 2

Figure 4.10 Example of Mismatched Schedule Lengths

a schedule length of four. Since the scheduler needs the loop size for determining schedules to be the same, a naive solution would be to set the schedule length to four, and cause the second pipeline to idle every fourth clock cycle. A much better solution is illustrated in Figure 4.11. The length of the schedule is dictated by the least common multiple of the natural lengths the compiler found. Effectively, the scheduling loops for each pipeline are unrolled until equal scheduling lengths can be found. This process can be repeated by the compiler across several nodes until a global schedule length is found by the system. If it turns out that the natural schedule lengths found by the compiler for several pipelines involve relative primes, then the schedule size may need to be too large in order to eliminate all NOPs in the schedules. For these cases, idle cycles may still be included. It may also be the case that some particular node has a relatively small amount of traffic going through it when compared to other nodes. Idle stages may be added in this case, simply because there is no work for the node to accomplish.

An application may have changing communication needs over time. A particular schedule of communication may need to be completely changed in order to support new communication demands. The ability to support this is accomplished in two ways. First a variety of different schedules can be stored in the switching node at compile time. When computation nodes determine that a new communication phase is coming, a simple

clk cycle	stream 1	stream 2
1	stream A	stream D
2	stream B	stream E
3	stream A	stream F
4	stream C	stream D
5	stream A	stream E
6	stream B	stream F
7	stream A	stream D
8	stream C	stream E
9	stream A	stream F
10	stream B	stream D
11	stream A	stream E
12	stream C	stream F

Figure 4.11 Avoiding Idle Slots by Lengthening the Schedule

instruction can cause a completely new schedule to start operating. Care must be taken when making this switch. All nodes must make the switch at exactly the same time in order to keep synchronous communication in the mesh. One scheme accomplishes this by providing a very low bandwidth stream that checks the processor interface at specific times in order to notice a request for a new schedule. Every node checks their processor interface at exactly the same time to see if a new communication schedule is requested. Even if some of the processors make the schedule swap and other do not, the rest of the processors will eventually make the change at exactly the same relative time in the loop. This manages to keep the nodes in sync with each other, although some care must be taken to handle straggling communications from old streams. Another scheme for switching schedules has one processing node sending a message to all processors indicating the change. A very low bandwidth stream can be assigned for just this purpose. All the nodes will be guaranteed to change schedules in a particular order at a particular rate since the initiating message will arrive at each node at a predetermined time.

It may be the case that several large communication schedules are needed over the course of an application. It may be impractical to store these separate schedules in the CFSM. For these cases, the processors can serve as a memory storage for additional schedules. Whenever a new schedule is needed, the processor can write the scheduler during operation to change its contents. This process may take many clock cycles to write large schedules, but presumably once a schedule is loaded it operates for a much larger number of clock cycles making the overhead insignificant. Synchrony between the nodes can be maintained as long as there are only fixed times at which the schedules can be updated and if the schedules can all be swapped in an organized manner.

The CFSM scheduler provides all the mechanisms mentioned in this section. The actual details of the implementation will be left for the next chapter.

4.7 Instruction Set Architecture

Now that mechanisms for the scheduler have been described, the virtual FSMs themselves need to be defined. Each virtual FSM provides all the information necessary to handle a communication stream. This includes the management of flow control, the transfer of data from one port to another, interactions with the processor interface, and any run time support for dynamic routing that is needed. One can imagine writing a few lines of code for each virtual FSM. However, the latency goal for transferring a communication word through a node is one clock cycle. In addition, that clock cycle should take no longer than a single internode transfer or a small RAM lookup.

Each virtual FSM consists of a single instruction opcode, a source specification and a destination specification. There is a single word of buffer storage associated with every instruction. The instructions can reference any of the ports, the processor interface, all the buffer storage in a pipeline, the memory address space of the node, and a variety of internal locations used for booting and system management. The following subsections will illustrate the different kinds of instructions supported and why there is a need for each type.

4.7.1 Flow Control Move Instruction

The standard method for transferring messages through the mesh will involve a single flow control move instruction (**fmove).** Consider the communication occurring in node 1 in Figure 4.12. The sole communication operation that will occur in node 1 is a transfer from node 1's -X port to its +Y port. Whenever the stream for message A gets scheduled, node 1 will attempt to read from the negative X direction, and then on the following cycle it will write to the positive Y direction. However, in addition to the simple transfer, the mechanics for flow control must also be handled without adding any length to the node's clock cycle.

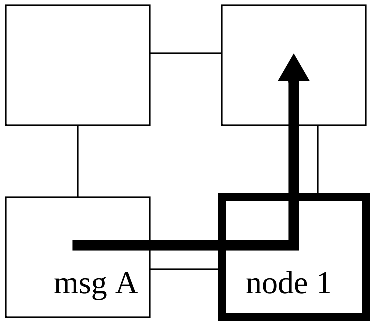


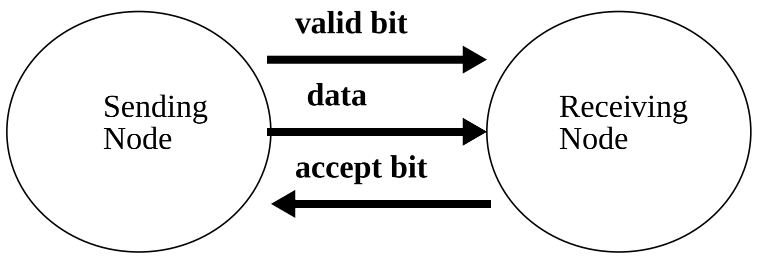
Figure 4.12 Illustration of **fmove** Instruction

A key observation in the handling of the flow control is that the same stream can not operate on two consecutive clock cycles. While this was done for different reasons, it serves to greatly simplify the flow control logic. The buffer word of storage for each stream has a valid bit associated with it that indicates whether valid data has been stored. The buffer register is effectively a single register in a register file that gets addressed by the stream number specified by the schedule. At the beginning of the stream's read cycle, the first piece of information that must be sent is whether or not the node is allowed to receive new data. To determine this, the stream's buffer register must be checked. If a valid word is found, the node must not send an *accept* bit to the previous node. In the case of Figure 4.12, instead of reading the negative X port for data, the stream will take its input word from its buffer register. A potential flaw arises from the fact that the buffer register must be read before the internode *accept* signal can be sent. This seems to violate the goal of a limited cycle time since the buffer registers are effectively a small RAM. This is where the advantage of scheduled communication is exploited. Since it is known at compile time the exact clock cycle that the stream for message A is going to run on node 1, the status of the buffer register can be read on the previous cycle. Since the same stream can not be scheduled two cycles in a row, there is no possible way that the status of the buffer register can change during that clock cycle. This means that the status of the *accept* bit is known at the very beginning of the clock cycle. Once the source word is read from either the port or the buffer, the word is stored into a register at the end of the clock cycle. There is the possibility that the sending node has no data to send. Along with every data transfer is a single *valid* bit that determines whether the data lines hold real values. All thirty-two bits of data and the *valid* bit are stored in the source register. Since there can be only one source word per physical pipeline, the logic to write the source word into a register is very simple.

On the following clock cycle, the source word is written to the appropriate location. If the source *valid* bit indicates that the data is not valid, then no data will be written on this cycle. If the word is valid, the appropriate destination will be written. Once again, flow control overhead must be considered. In Figure 4.12, the word is written in the positive Y direction. At the same time, the other node connected to this port has decided whether or not it can accept new data and is transmitting an *accept* bit. If the node indicates it can not accept new data, node 1 must write the buffer word into its buffer storage. Once again there appears to be a problem in that the buffer register write (really a RAM write) can not start until the node reads the *accept* bit. This problem is solved by always writing the word into the stream's buffer register file. It is only the valid bit of the buffer register that gets written based on the value of the *accept* bit. The valid bits for all the buffer registers are arranged as a single thirty-two bit register. This means that once the *accept* bit arrives on the node, it can almost immediately be stored into the correct location without a full RAM write being needed. This enables the clock cycle time to meet the system goals. The flow control protocol for the sending and receiving nodes is illustrated in Figure 4.13. The entire flow control scheme can basically be implemented without any overhead since much of the work can be done ahead of the actual data transfer.

4.7.2 Conditional Flow Control

While the **fmove** instruction handles most communications that can occur in the mesh, there are two problems it can not handle. For reasons related to flow control, the same communication stream can not be scheduled for two consecutive instructions. As a result,



Protocol:

Sending Node

1. If **accept** bit is set, transfer is completed.
2. If **accept** bit is not set, data word and valid bit are written into stream's buffer register.
3. Always look in buffer register first for a source word.

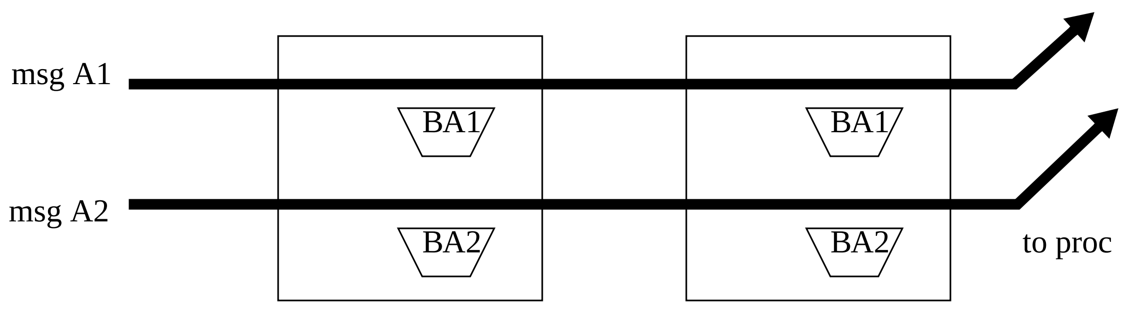
Receiving Node

1. Assert **accept** bit if buffer register is empty.
2. If buffer register is full, use buffer register data as source and deassert **accept** bit.

Figure 4.13 Flow Control Protocol

special efforts must be made to allow a communication path a bandwidth of one hundred percent. One way to handle the problem would be to create two streams that follow an identical path and are assigned fifty percent bandwidth. The scheduler would switch back and forth between the two streams, and the effective bandwidth between the source and destination would be one hundred percent. Figure 4.14 shows message A being split into two identical streams, A1 and A2. The scheduler alternates between the two streams, and the sending processor would inject messages alternately into each of the streams. As long as there is no blocking in the network, the scheme will work fine, and full bandwidth can be achieved for the communication.

Figure 4.15 shows one of the streams being blocked due to the destination processor being unable to accept a communication word for one cycle. The word gets temporarily stored in message A1's buffer register. On the following cycle, the contention clears, but now message A2 is scheduled. It sees no blocking so its word get transferred to the processor. However, originally the communication word for message A1 was injected into the



the scheduler alternates between two identical streams

Figure 4.14 100% Bandwidth Can Be Achieved By Duplicating the stream

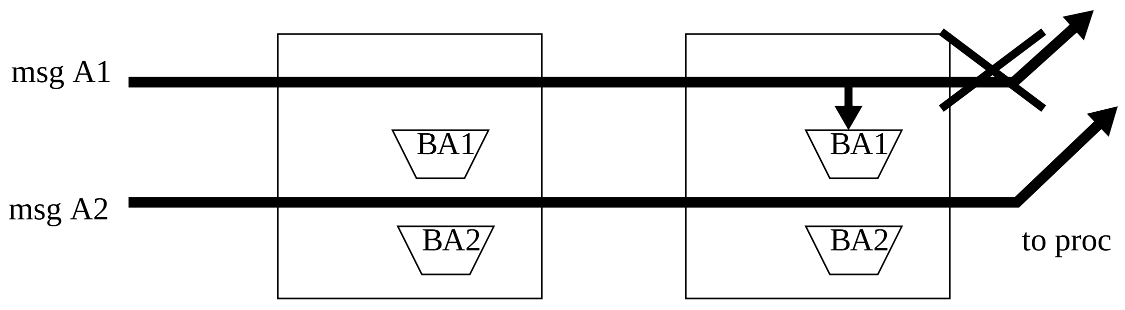


Figure 4.15 A Blocking stream Can Cause Out of Order Data

network a cycle before the communication word for message A2. Due to the fact that message A1 was stalled because of the temporary blocking by the destination processor, the word in message A2 is able to pass message A1. Unless ordering information is encoded in the messages themselves, the data can be transferred out of order without the processor being able to detect the misordering.

The second limitation of the **fmove** instruction comes from its inability to properly handle messages. When processors communicate using messages, the entire message is treated as a logical unit. Either the entire message should be sent, or none of it should be sent. Similarly, if the head of a message gets stored in a buffer register, the rest of a message should be stored in a buffer register. If a message of length ten requests a bandwidth of ten percent in a schedule of length one-hundred, ideally the message would have ten consecutive slots in the schedule. In order for any node in the message's path to be able to store the entire message in case of blocking, a separate virtual stream can be assigned to each word in the message. If each of these streams is defined by an identical instruction, in the best case all ten words of the message will be routed through the network with worm-hole routing. If these instructions were all **fmove** instructions, the previous paragraph illustrated that words could be transferred out of order. Another problem also exists. Suppose the processor injecting the message into the network is not able to do so until the fifth of the ten streams is scheduled. The header would be transferred on the cycle actually reserved for the fifth word of the message. By the time the tenth stream is finished, only five words of the ten word message will have been routed. Since the message's allocated bandwidth is over, ninety more cycles will go by until the rest of the message is routed.

Both of these problems are avoided by the creation of a conditional flow control (**cmove)** instruction. The **cmove** instruction works by remembering the transfer success of the previous instruction. If the previous instruction had a read success but a write failure, this means that a valid source word was read but the destination location refused an additional word. The **cmove** instruction will expect a valid source word to be available and will immediately put the word in the stream's buffer register. Likewise, if the previous instruction failed during a read due to either a valid word not being offered or the stream's buffer register being full, the **cmove** instruction will also fail on the read and indicate that it can not accept new data. A message can be sent by having the head of a message sent with a traditional **fmove** instruction and the rest of the body words sent with **cmove** instructions. This causes all the words of the message to be treated exactly the same as the header of the message, regardless of the state of the network. Consider the example back from Figure 4.15. Two messages were allowed to pass each other due to the pattern of congestion in the network. Using the **cmove** instruction is illustrated in Figure 4.16. Since stream A1 sends its word into its buffer, stream A2 will also send its word into its buffer, regardless of whether the processor can accept new data. Communication words can no longer pass each other in transit. The downside to this scheme is that words must now be transferred in groups of two. However, one hundred percent bandwidth can be achieved.

The second problem of partial messages being sent is also avoided. If the first stream of a series does not see a valid source word, the rest of the streams in the message will not attempt to read any data from the processor, even if the header word is eventually available for transfer.

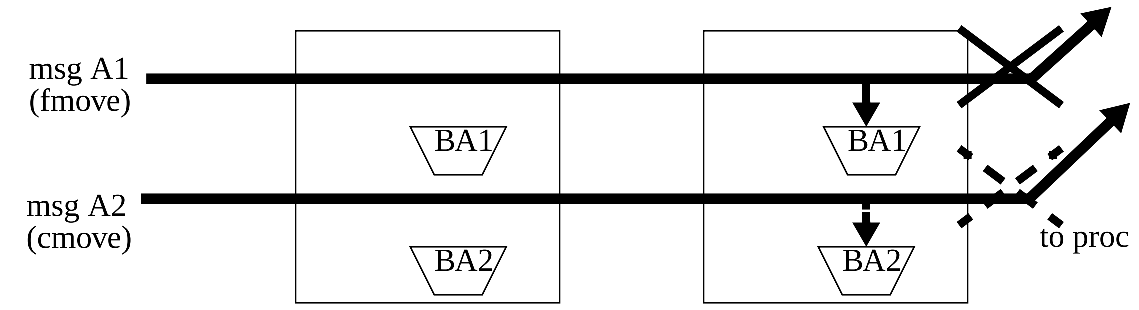


Figure 4.16 The Conditional Flow Control Instruction Preserves Order

4.7.3 Blind Move Instruction

Occasionally, words can be transferred without any regard to the flow control protocol. Systems supporting graphics or digital signal processing might be systolic in nature, guaranteeing that data is to be transferred on every cycle. For these cases, the flow control protocol restriction of a stream not being allowed to be scheduled every clock cycle is pointless, since the only reason the restriction exists is due to the possibility of words being buffered. A systolic system can demand that words are injected and removed from the network on every clock cycle. The blind move instruction (**bmove**) allows transfers to take place regardless of the handshake between the *valid* and *accept* bits.

4.7.4 Forking Communication

A forking or multicast operation involves copying communication words and sending them to multiple destinations. Many applications need a particular node to send identical data to multiple locations. This problem presents a particular difficulty in a scheduled communication system. Consider the example in Figure 4.17. Message A is to be routed to multiple nodes. A simple way for a fork to be implemented would be for the **fmove** instruction to support multiple destinations. The flow control semantics for this become complicated. For a transfer to be considered successful, the *accept* bits from all of the destinations must be asserted. If any one *accept* bit comes back deasserted, not only does the word need to be stored in a buffer, but the information of which transfers succeeded and which failed must also be stored. The next time the stream is scheduled and the source word is taken from the buffer register, the word must only be transferred to those locations that were unable to read the word on the last attempt. Between the complexity of managing this scheme and the overloading of the **fmove** instruction, this scheme would cause more overhead to the clock cycle time than the value of the fork instruction warrants. A much simpler scheme is implemented instead.

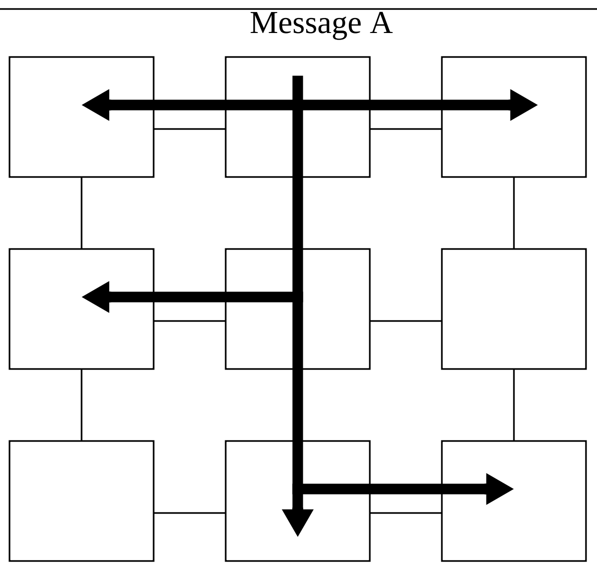
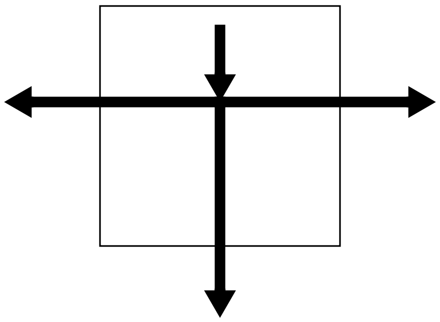


Figure 4.17 Forking Operations Require Data Replication

To accomplish an N-way fork in a node, N consecutive streams must be scheduled. Consider the first node of in the communication stream in Figure 4.17. The first node wishes to transfer data in the positive X direction, the negative Y direction, and the negative X direction. Three consecutive streams are required to accomplish this. The solution is illustrated in Figure 4.18. All three streams use the **fmove** instruction and a single source and destination. The first stream takes a word from the processor interface, and attempts to transfer it to the positive X direction. At the same time the node is writing the X direction, the next stream is attempting to read from the exact same port. For the first communication stream's write to succeed, both the node connected to the positive X port and the second communication stream in the same node must assert an *accept* bit. If either of these *accept* bits is not asserted, the outgoing data must be invalidated and the word must be written in the first stream's buffer storage. This overhead comes for free because the second stream's *accept* bit can be calculated during the previous cycle. If no flow control blocking occurs, at the end of the first stream's operation, the node connected to the



stream 1	**fmove proc5, +x**
stream 2	**fmove +x, -y**
stream 3	**fmove -y, -x**

Figure 4.18 Single Node Fork

positive X port and the second stream will each have a valid copy of the data. If a flow control block occurred, neither of these will have obtained a valid copy of the data, but the first stream will have the communication word stored in its buffer, and the fork will be attempted again the next time the stream is scheduled.

In the case when the transfers succeeds, the second stream would then attempt to write the communication word to the negative Y port on the following clock cycle. At the same time, the third communication stream would be reading the same port. Once again, for the transfer to succeed, the *accept* bits for both the node connected to the negative Y port and the third stream must be asserted. In this manner, an N-way fork can be accomplished with no overhead, although it takes N clock cycles. If data is being forked to many destinations, a tree of forks can be set up among many nodes, and the penalty can be reduced to log N time rather than linear time.

4.7.5 Join Operations

A join operation allows several communication streams to be combined into a single stream. Figure 4.19 shows an example of a join operation. Messages A, B, C, and D all get combined into a single communication stream that gets routed to the final node’s processor. The reason for a join operation is to allow several low bandwidth streams to be combined in any arbitrary order into what might become a fairly high bandwidth stream. By combining the independent streams into a single stream at the end of the communication path, the overhead needed by the final node’s processor is significantly reduced. It only ever needs to check one communication stream for incoming data. In addition, by reducing the number of communication streams in the network, the potential for congestion is also reduced. Join operations may be used when scheduling all the different streams becomes too difficult due to a large number of communication requests in the network.

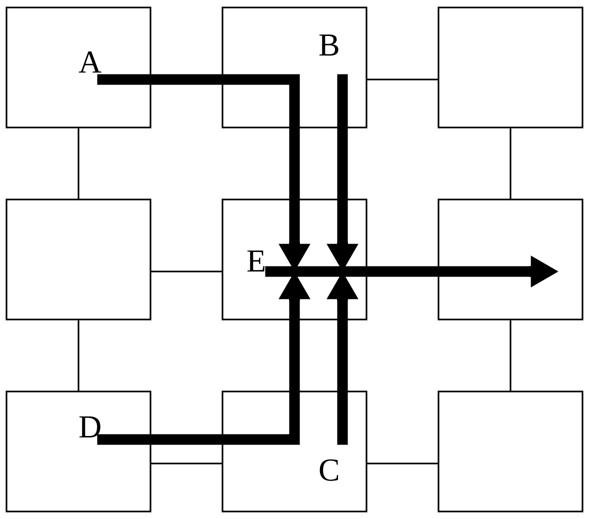


Figure 4.19 Example of a Join Operation

Implementing this scheme in a scheduled architecture is problematic. Since every stream’s communication FSM consists of a single instruction, one scheme might be to specify multiple sources for a stream. However, the problems of arbitration and flow control management quickly rule this scheme out. Instead, an alternate scheme is implemented. For an N-way join, N+1 communication streams must be created at the node where the join occurs. Consider the example in Figure 4.19. In the middle node, four streams are combined into one. For each of the source communication streams, the instruction specifies the appropriate port as a source, but instead of specifying the positive X port as a destination, stream E's buffer register is specified as the destination. If stream E's buffer register is already full, flow control semantics dictate that the word get stored in the stream's own buffer register. The communication instruction for stream E will not specify any source, although it will be implemented with an **fmove** instruction. Since it uses normal flow control protocol, it will always check its own buffer register for a source word. If one of the other four streams inserted a word into stream E's buffer register, it will become stream E's source word. The four input streams can be assigned any bandwidth, and this scheme will correctly work. The only restriction on this scheme, is that out of order transfers be allowed. Of course, for a join operation involving streams with different bandwidth allocations, the proper order of the data is not really defined.

4.7.6 Pipeline Transfers

Many communication streams may attempt to use the same port resources. Ideally, the scheduler can determine a schedule that allows all streams to be timed such that a stream can read a source word on one clock cycle, and write its destination on the following cycle. Occasionally, a resource may already be in use by the other pipeline, and the scheduler may not find any other scheduling slots for the stream. As a result, the word may need to be stalled for a cycle. This can be accomplished by having the stream write its communication word in some unused buffer register. The next stream scheduled could read this buffer register and write the appropriate port as a destination. The net effect is that the communication word still gets transferred over the same ports, but it has one extra cycle latency through the node. Figure 4.20 shows an example of stalling a message.

Stalling a message has the negative effect of requiring an extra stream to be scheduled. Occasionally, there may be idle slots in the opposite pipeline's schedule in which it might make more sense for the extra stream to be scheduled. To accomplish this, the two pipelines in the system must be able to communicate. The natural location for such a communication is the ports themselves. For example, assume the communication stream in Figure 4.20 needs to be stalled a cycle. The normal communication instruction involves an **fmove** instruction from the negative X port to the negative Y port. The stall mechanism can happen over any of the unused ports. The first stream can read an input word from the negative X port, but might write it to the positive X port. At the same time this stream is

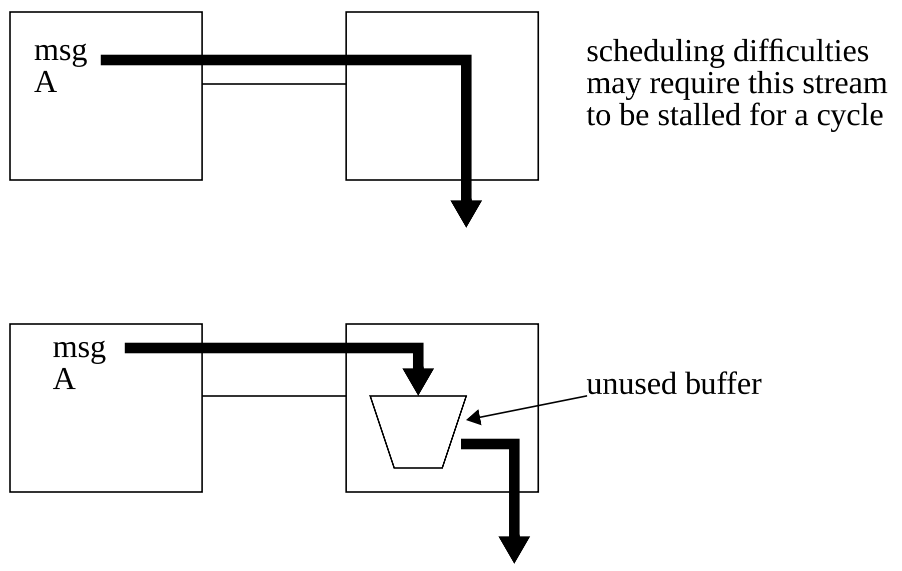

msg
A
scheduling difficulties may require this stream to be stalled for a cycle
msg
A
unused buffer

Figure 4.20 Messages Can be Stalled With an Unused Buffer

writing, a stream in the opposite pipeline could read the positive X port. On the following clock cycle, this second stream would write the negative Y port. Full flow control semantics are supported, and the communication word gets transferred between pipelines. Some care must be taken to cause the original stream's *accept* bit to come from the second pipeline rather than from a node connected to the positive X port. However, since this condition can be detected in an earlier clock cycle, the bypass of the *accept* bit comes for free.

The semantics for a fork operation and a pipeline transfer are similar but have important differences. A fork must happen within a single pipeline, and the logic for the original stream's *accept* bit is the logical AND of the connecting node's *accept* bit and the next stream's *accept* bit. A pipeline transfer occurs between consecutive streams in opposite pipelines. If this case is detected, only the *accept* bit of the opposite pipeline's stream is checked, and the connecting node's *accept* bit is ignored.

4.7.7 Storing Extra Instructions

A particular phase of communication may require a node to only use a small number of communication streams. If this is the case, the streams for several phases of communications can be stored in the CFSM, since each pipeline can hold thirty-two virtual FSMs.

Since the scheduler indicates which streams will run on every clock cycle, streams from future communication phases will never be accidentally run.

The number of streams used over the course of an application may exceed the maximum storage capabilities of the CFSM. In these cases, the processor can act as a main memory for communication instructions that might need to be used the CFSM. During the execution of the application, the CFSM acts as a cache, holding only those instructions that are currently needed. When a communication phase change occurs, the virtual FSMs can be written from the processor interface and updated with the new instructions. The virtual FSMs are part of the memory address space and can be written dynamically. One mechanism for supporting simple dynamic routing would be to allow the processors to set up communication streams on the fly by accessing the CFSM's memory address space based on run time decisions.

4.8 Communication Instruction Reads

While most communication streams will simply transfer data between the node's ports, a node's source word can come from a variety of locations. These include the node's physical ports, processor interface locations, and any stream's communication buffer. In addition a special source can be specified that causes a stream to read its buffer register without clearing its validity on a successful data transfer. This mechanism was added to potentially allow hard constants to exist in the datapath without getting cleared every time they are read.

A read operation is also responsible for recording a *read success* bit that will be saved for the following instruction in the pipeline. For a **bmove** instruction, all reads will be marked as successes since flow control is turned off.

For a **cmove** instruction, the *read success* bit is simply copied from its previous value. A multi-word message will be transferred by having the header issue an **fmove** instruction and the body words issue **cmove** instructions. Since each body word of the message should see the exact same read action occur as for the header, passing the *read success* bit along between successive **cmove** instructions is the correct operation.

For an **fmove** instruction the *read success* bit must be calculated. A read is considered to have succeeded if a valid word is received from the encoded source in the stream's instruction. If a word is grabbed from a stream's buffer register instead, the read is marked as a failure. This informs following **cmove** instructions that the encoded location should not be checked for a source word. A stream will always check its buffer register for valid data, so the correct source will be checked. If a stream tries to read valid data from a port, but no data is available, the *read success* bit must be marked invalid. The following **cmove** instructions should not attempt to read the port, even if valid data is present. This prevents messages from being half sent. If another buffer register or the processor interface is read, the *read success* bit depends on whether a valid word is received.

Flow control semantics are supported for transfers within a node as well as the ports. If an **fmove** instruction wishes to write another stream's buffer register or the processor register interface, these resources must be allowed to accept new data. In the case of writing another stream's buffer register, the **fmove** write must fail if a word already resides in the buffer register. During the read of the source word, the validity of a destination location within a node is also read so it can be determined ahead of time whether a write operation can be allowed. If this operation were to happen at the same time as the actual write itself, then two RAM accesses would be required - the first to check if a valid word already resides in the location, and the second to perform the actual write. This violates the NuMesh goal of only allowing a single RAM access to determine the cycle time of the node. Writing to the processor interface works exactly the same way. During the read operation, the ability for the interface to accept new words is also checked.

4.9 Communication Instruction Writes

The write operation for a stream is fairly simple. The instruction word has an encoded destination that specifies the location to be written. In addition to writing the specified destination, the stream's buffer register also is written. The *valid* bit for the buffer register gets set based on the result of the destination's ability to accept data and the type of instruction opcode. A **bmove** instruction never sets the *valid* bit unless the buffer stream is the actual encoded destination.

A **cmove** instruction mimics the action of the previous stream. If the previous stream wrote a valid word into its buffer register, the **cmove** stream will do the same thing.

The **fmove** opcode causes the flow control protocol to be observed. For the case when a port is a destination, the connecting node's *accept* bit is checked. If the bit is asserted, the *valid* bit of the word in the buffer register is set to zero. If the *accept* bit is deasserted, the *valid* bit gets set since the transferred word was not accepted, and the buffered word must be sent again when the stream gets rescheduled. For the case when either another stream's buffer or the processor interface is chosen, the ability of these locations to accept data will have been determined on the previous cycle. The stream's buffer register *valid* bit will be set based on this information.

The CFSM must also support writing into the memory address space of the node for both bootstrapping and run time dynamic purposes. A specific destination coding corresponds to writing the memory address space. This kind of write can never fail, regardless of the type of instruction opcode selected. Words can be transferred to the memory address space in a similar fashion to all other communication transfers. Data ports, the processor interface, or any of the buffer registers can be sources for writing the memory address space. Message can even be forked with one of the branches going to write the memory address space. The high order bits of the communication word choose which piece of memory on the chip will be written, while the lower order bits serve as the data to be written. Possible locations within the memory address space are included in Figure 4.21. Many of these locations serve system level needs and will be discussed in section 4.8 of this chapter.

4.10 System Issues

The NuMesh project faces a host of system design issues that are difficult to solve. The architecture design of the CFSM is required to aid in the solution of some of those problems. This section will discuss some of the architectural features that are implemented to make the system design more manageable.

4.10.1 Booting

The NuMesh system is designed to be booted by a host processor connected to one of the ports of the mesh. This idea is illustrated in Figure 4.22. On power-up, all nodes are idle, waiting for information to come in on any port. The host computer sends schedule and instruction data to the node to which it is connected. The instructions set up paths to

Location	Function
Scheduler	Scheduler contents can be altered at any time
Schedule Pointer	An index into the scheduler can be written to allow quick changes between communication phases
Communication streams	Communication instructions can be overwritten at any time
Processor JTAG	A JTAG interface to the processor can be written by the CFSM
Reset Bits	A node can reset any of its neighbors and put them in boot mode
Clk System Bits	A node can tell any of its neighbors to perform clock synchronization with it
Diagnostic LEDs	A set of LEDs can be written for diagnostic purposes

Figure 4.21 Components of a Node's Memory Address Space

each of its neighbors so the host computer can send booting information to each of the nodes. The information fans out until the host computer can load application schedules and instructions into each of the nodes. Finally, the host computer sends around a message that takes each node out of boot node at the same time, and the application communication code is run.

Processors also get booted over the communication network. One of the destinations to which the CFSM can transfer data is a JTAG interface to the processor. In the prototype NuMesh modules, this JTAG interface can be used to boot the attached processor - the microSPARC. Through this interface, the processor can be initialized and its memory can be loaded with application code. When each CFSM node gets initialized during bootup, part of the process is to boot the microSPARC as well. The JTAG interface can be read by the CFSM as a source to provide debugging support for the processor.

A CFSM node gets booted one pipeline at a time. Special memory locations get written to switch the loading of boot information between the two pipelines. The pipeline is

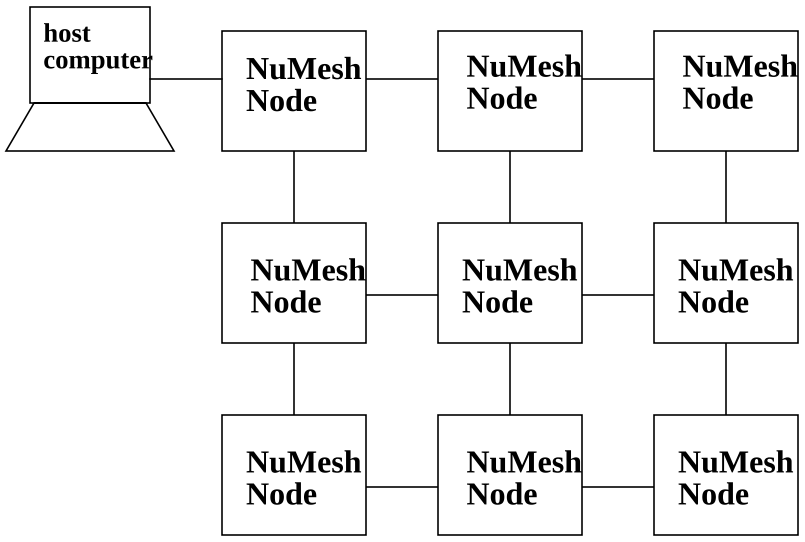


Figure 4.22 Booting the Mesh Occurs From A Host Computer

designed to detect when a node is trying to boot it and will execute a single instruction of reading the specified port and writing the memory address space until the node is taken out of boot mode. Implementation details of the booting process will be included in the next chapter on architecture implementation.

4.10.2 System Clocking

The NuMesh system runs on a synchronous global clock in a scheme described in [Prat95]. All clocks start with the same nominal frequency, but can start out of phase with each other. The node connected to the host computer first synchronizes with each of its neighbors by telling each of the nodes in turn to match its phase. Every CFSM node has a separate clock bit line that is connected to each of its neighbors. These lines can be written as part of the memory address space. When a node sees a clock line from its neighbor pulled high, the node matches phase with the clock line of the neighbor. Successive nodes are added to the collection of phase matched nodes until the fanout includes all nodes in the system. The order in which the nodes are added to the collection of matched phased nodes is important, but is completely under the control of the host computer. The host computer can create a communication path to any node and then transfer a word to the node’s memory address space that asserts the various clock lines.

4.10.3 Reset Logic

When the nodes are powered up, the nodes are in an idle state. Each node has a reset line connected to each of its neighbors. When a node sees a reset line asserted by any of its neighbors, it immediately goes into boot mode and starts trying to read communication words from the port where the reset was initiated. Any valid words are written to the node's memory address port. A node can be programmed to ignore resets from certain ports in case there might be a defective node in the mesh or a node is on the edge of the mesh. Once a reset is initiated, the reset can not be over-ridden by a subsequent reset until the original reset line gets deasserted.

One of the side-effects of the power-up reset is to clear all the *valid* bits in a node for the buffer registers and the processor interface registers. This prevents power-up errors from mistakenly loading these registers with what the node will think is valid data. Resets from a neighbor do not cause all the valid bits to be cleared, although writing a special location in the memory address space can explicitly clear these bits. One way for a node to change communication phases or to load new instruction streams dynamically is for a neighboring node to assert its reset line and put the stream into boot mode in order to load the new data. Since this operation could occur in real time, it may not make sense to automatically clear out all the node's state. On the other hand, if an application decides to go to a new phase of communication, it may make sense to clear out all the data from the previous phase. Since instructions may get overwritten with completely new instructions, it may make no sense to have data that has backed up due to flow control in the previous phase of communication reside in the new streams' buffers. Having a simple mechanism to immediately clear out all the buffer registers can be very useful.

4.10.4 Diagnostic Support

No parallel processing system is complete until lights can be seen flashing. With this goal in mind, four LEDs per node exist that can be written as part of the memory address space. When the nodes are powered up, these bits default to the on position. As each node goes through the booting process, the lights can flash in interesting patterns to give the illusion of computation.

4.11 Communication Models

Several models of communication can be supported on the NuMesh architecture. This section will illustrate a few, though certainly not all, of the ways applications may communicate over the NuMesh network.

4.11.1 Scheduled Streams

The NuMesh architecture is designed to optimally support scheduled communication. In this model, a set of communication streams can be extracted at compile time. A compiler serves to lay down all communication paths to minimize congestion by assigning communication stream instructions to each node. A scheduler than orchestrates the communications assigning a bandwidth to each communication stream and ensuring that neighboring nodes in a communication path schedule their communication instruction in the appropriate slot. More complicated operations such as forks or joins are broken down into combinations of individual communication streams with added constraints. If a set of communication stream requests can not be met, the compiler tries to meet as many constraints as possible, but will reduce the scheduled bandwidth of selected paths until a valid schedule can be achieved.

The input for this model can come from a variety of forms. Graphical block languages or annotated parallel languages are reduced to time-varying directed graphs that can be handled by a robust compiler. Small numbers of dynamic decisions can be handled by assuming scheduling paths for all the possible outcomes of the dynamic decisions. As long as the required number of scheduled communications does not grow too large, this model of scheduled communication runs very fast on the NuMesh architecture. Since no run-time data need be checked, the cycle time of the transfers is much less than that of dynamic routing. In addition, there are no worst case patterns since congestion can be minimized by the compiler.

4.11.2 Nearest Neighbor Communication

Some applications require frequent communications to nearest neighbor nodes. Scientific applications in physics and chemistry as well as graphics based applications often involve nearest neighbor communications that can occur randomly throughout the course of an application. The NuMesh architecture can quite easily support this model. Every node can set up one virtual path to each of its four neighbors, and these streams can be scheduled very tightly allowing each path to get a share of the bandwidth. Each virtual stream will have a single processor interface register assigned to it. Whenever the application decides it needs to communicate with a particular neighbor, the message gets dropped in the processor interface register that belongs to the appropriate stream. Four more processors registers get reserved for receiving messages from each of the four neighbors. The application can either poll these locations periodically, or an interrupt can occur whenever a word arrives at the node.

4.11.3 Dynamic Communication

Dynamic communication presents the biggest problem for the NuMesh architecture, but it can still be handled. For small meshes, a communication path for every pair of nodes can be set up, and the processor can simply drop a message in the appropriate interface register to send a message to any node. As the number of communication streams required for this becomes too large, one can imagine sending messages across the mesh using the nearest neighbor model. At each node, the processor looks at the header and determines the next hop in the message's path. It then alters the header word and drops the message into the interface register corresponding to the appropriate neighbor. The number of hops can be reduced by setting up various express channels that quickly send a message to specific quadrants of the mesh. From there the nearest-neighbor communications can finish the routing. Any sort of virtual network can be laid on top of the NuMesh topology to allow for an arbitrary communication scheme. The overhead comes from having to route the message at each hop to the processor and forcing data dependent computation to occur in order to choose the next hop.

4.11.4 Hybrids

In practice, hybrids of all three of these communication models can be used. Some streams can be scheduled with very high bandwidth if their communication patterns are known, while low bandwidth channels can be set up to handle dynamic routing that goes through the processor. In addition, the node can store multiple communication patterns and switch between them as communication needs in an application change. If a great many different communication patterns are needed, even the processor can store some of these schedules in its memory and download the information to the CFSM when necessary.

4.12 Architecture Overview

The previous sections describe an architecture capable of supporting scheduled communication. A variety of mechanisms are included to facilitate run time dynamics for many different application groups. The following chapter discusses the implementation of the instruction set architecture. This section will serve as an overview of the description of the NuMesh communication FSM

4.12.1 Instruction Set

A single instruction format is supported, with each instruction taking fourteen bits. Each instruction contains three fields: an opcode, a source and a destination. Figure 4.23 defines the three fields of the instruction format.

4.12.2 Pipeline Overview

The NuMesh CFSM can be broken down into a four stage pipeline. During the first stage, the scheduler determines which virtual stream will run for each pipeline. During the second stage the stream's communication instruction is read. The third stage involves the reading of a source word from either the ports, the processor interface registers, or the buffer registers. During the fourth stage, the appropriate destination is written from among the ports, the processor registers, the buffer registers, or the memory address space.The flow control protocol is implemented during the third and fourth stages of the pipelines and may cause reads or writes from locations other than those encoded in the communication instruction. A skeleton picture of the architecture is shown in Figure 4.24.

Much detail, including paths for preloading the RAMs, system support hardware, processor interface support, the second pipeline, and the datapath of the chip, are omitted for clarity. The following chapter will discuss the architecture in much more detail and will provide diagrams of the entire chip architecture.

Opcode	Sources	Destinations
13 12	11 6	5 0

• **bits 13-12** opcode

- 00 **fmove -** flow control move. Standard flow control protocols are followed.
- 01 **cmove -** conditional flow control move. The transfer success of the previous instruction is followed.
- 10 **bmove** - blind move. No flow control semantics are followed.

• **bits 11 - 6** sources

- (0-3) node ports
- (6) processor jtag TDO bit
- (7) bflow read location
- (16-31) processor interface registers
- (32-63) buffer registers for all streams

• **bits 5 - 0** destinations

- (0-3) node ports
- (6) memory address space
- (7) bflow write location
- (16-31) processor interface registers

Figure 4.23 Instruction Definition

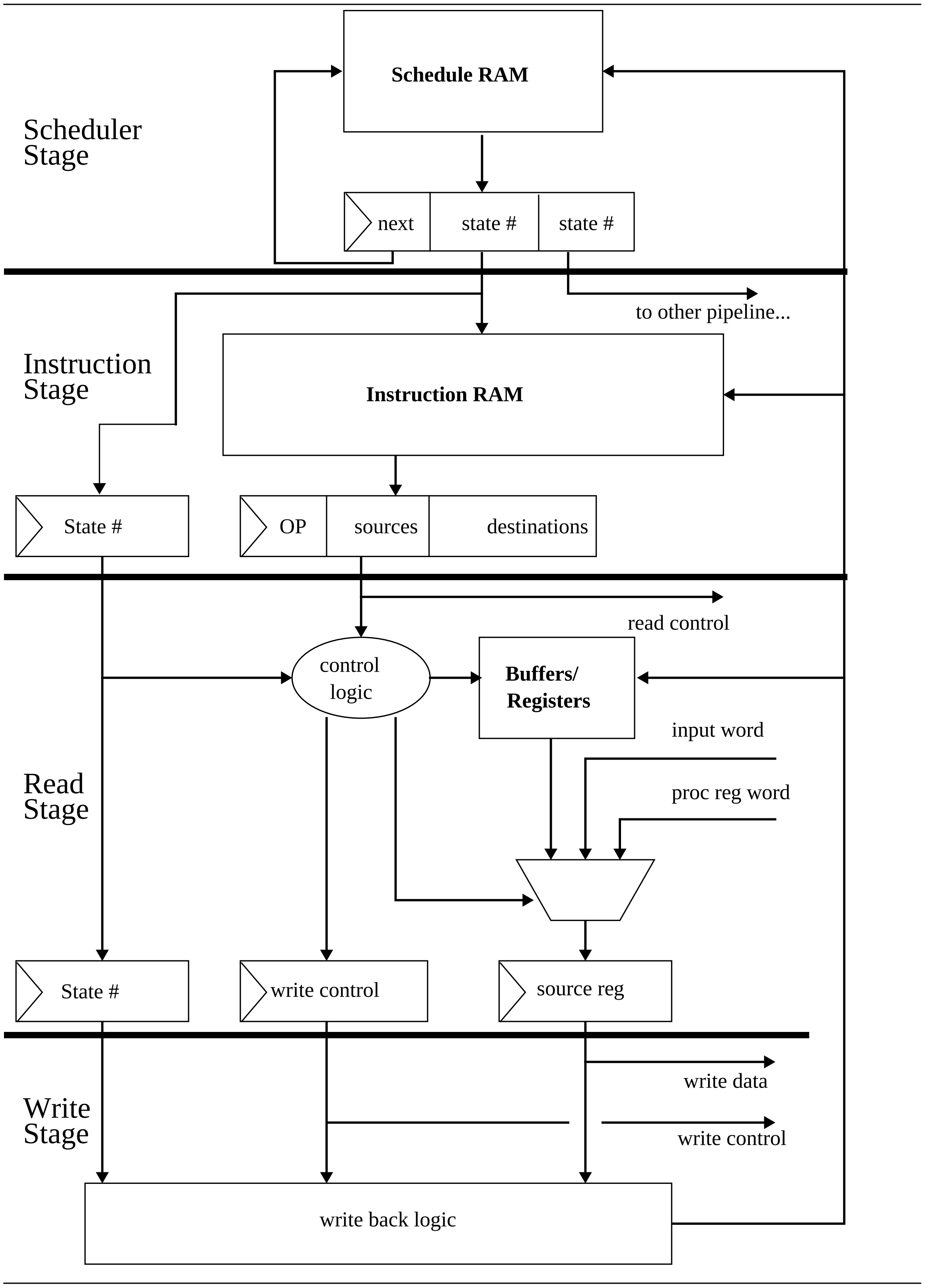


Figure 4.24 Hardware Architecture Design

Chapter 5

CFSM Implementation

The CFSM was implemented using Chip Express, Laser Programmable Gate Array (LPGA) parts. The CX2000 series from Chip Express is fabricated in a .6 micron, 3-layer metal CMOS process. Separate configurable SRAMs exist for implementing higher speed memories. Parts are created by a single mask single-etch process.

The CFSM is designed as a four stage pipeline. The goal of the pipeline is to allow for a cycle time limited by either a single internode transfer or a small RAM operation, plus a minimum number of gates. This chapter will describe the implementation of the design and give diagrams for each of the stages. While the presented design will be organized for clarity, logic delays can often be traded off between stages of the pipeline, allowing certain signals to be precalculated. These decisions can be made based on the delays or signaling characteristics of the memories, as well as the characteristics of internode transfers. Since these decisions are largely determined by the process in which the architecture is implemented, the discussions of such decisions will be held to a minimum. The goal of this chapter is to present the CFSM architecture in manner that allows any designer to understand its operation.

5.1 Stage 1: The Scheduler

The scheduler is responsible for selecting a virtual stream to operate for each pipeline. It also sets the outer loop size of the schedule to be run. Once the end of the schedule is reached, the scheduler must detect this immediately and restart the schedule without losing a clock cycle. Ideally, multiple schedules can be stored in the scheduler, and switching between the different schedules can be done with minimal and predictable overhead.

The scheduler is pictured in Figure 5.1. Its major component is a static RAM whose words contain three fields. Each of the lower two fields contain a five bit number corresponding to the virtual stream that is to be run in each pipeline. The third field contains a seven bit index back into the RAM of the next scheduler instruction to be executed. This makes the total size of the schedule RAM 128x17 bits. Schedule lengths must be kept under 128 instructions in length. While this number is somewhat arbitrary, it allows for complicated schedules to be implemented, while the total size of the schedule RAM is still small enough to allow for quick accesses. Under normal operation, the last instruction of a schedule will point back to the first instruction in the schedule. Instead of a pointer, the

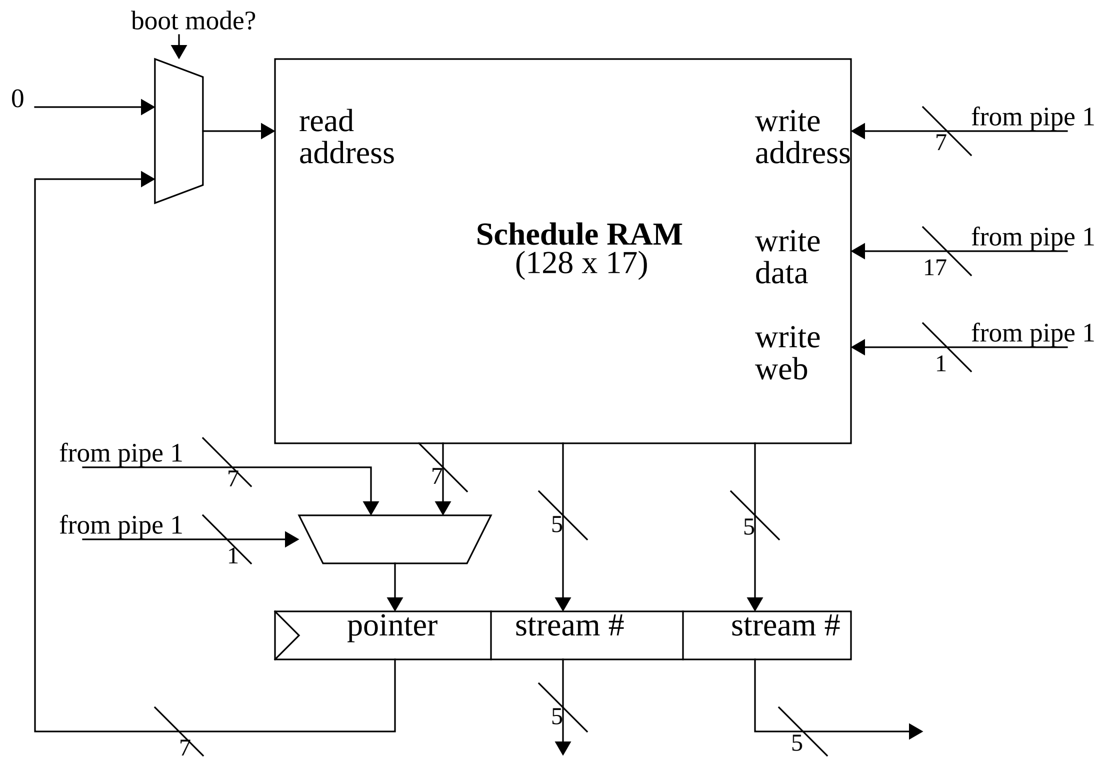


Figure 5.1 Stage 1 - Scheduler

index to the schedule RAM could have been implemented with a counter and a control bit. A pointer was used instead to allow for quick switching between stored schedules in the RAM. While the schedule is executing, this third field of any entry can be updated with the first address of a different loop. When the schedule RAM executes the modified instruction, the pointer causes a whole new schedule to start executing.

Multiple schedules can be stored in separate locations in the schedule RAM. The pointer index into the schedule RAM can be written as part of the memory address space. This ability to write the pointer allows for quick changes of communication phases in the system. A single scheduled instruction can try to transfer a word from the processor interface registers to the pointer. Whenever a processor wants to change phases, it can write this register with the index appropriate to the new phase.

The schedule RAM locations can also be written as part of the memory address space. Each word of the schedule RAM is seventeen bits long. This allows both the address and the data to be encoded in a single thirty-two bit data word. During boot mode, the schedule RAM will always read from address location zero. Once the pipeline leaves boot mode, the first instruction to be read will still be from location zero. This allows for a simple programming model in which the first schedule to be run is loaded in the first addresses of the RAM. A write to the schedule pointer location can get around this constraint.

Routing proved to be the biggest difficulty in the implementation of the chip. The Chip Express technology has a limited number of routing channels available. To make the routing problem easier, only one of the two pipelines is allowed to write the schedule RAM and the schedule pointer. If the other pipeline attempts a write to the scheduler's memory locations, the data is simply dropped. In this implementation, only pipeline zero can write the schedule space.

5.2 Stage 2: Reading Instructions

The second stage of the pipeline serves two main purposes. The instruction of the scheduled stream is read and partially decoded, and the valid bit of the stream's buffer register is checked for valid data. Figure 5.2 shows a diagram of the implementation of stage two for one of the two pipelines. The input stream number serves as an index into the instruction RAM. During boot mode, this index will be forced to zero, providing a consistent behavior when the pipeline is switched out of boot mode. Under normal operation, the stream's fourteen bit instruction will be read out of the RAM and passed on to the next stage of the pipeline. At the same time, the buffer register valid bit of the stream will be read to discover whether or not the stream can accept new data on the next clock cycle. While Figure 5.2 shows only a single port on the valid bit RAM, there are actually several ports, but only a single read port gets accessed by the second stage logic. The rest of the ports will be discussed later.

During boot mode, the instruction RAM contains no valid data. When the chip is powered up, no reset line will be active, so a NOP instruction gets forced into the instruction stream through a mux at the output of the instruction RAM. Once one of the ports asserts a reset, an instruction requesting transfers from the resetting port to the memory address space will be spliced into the instruction stream of the node through the output mux. Every cycle will generate such a transfer until the reset is cleared or boot mode is left. Since a write to a memory address space can not fail, there is no reason to worry about flow control backups on these transfers. Writing the memory address space allows a node to load its schedule RAM, instruction RAM, and even to exit boot mode.

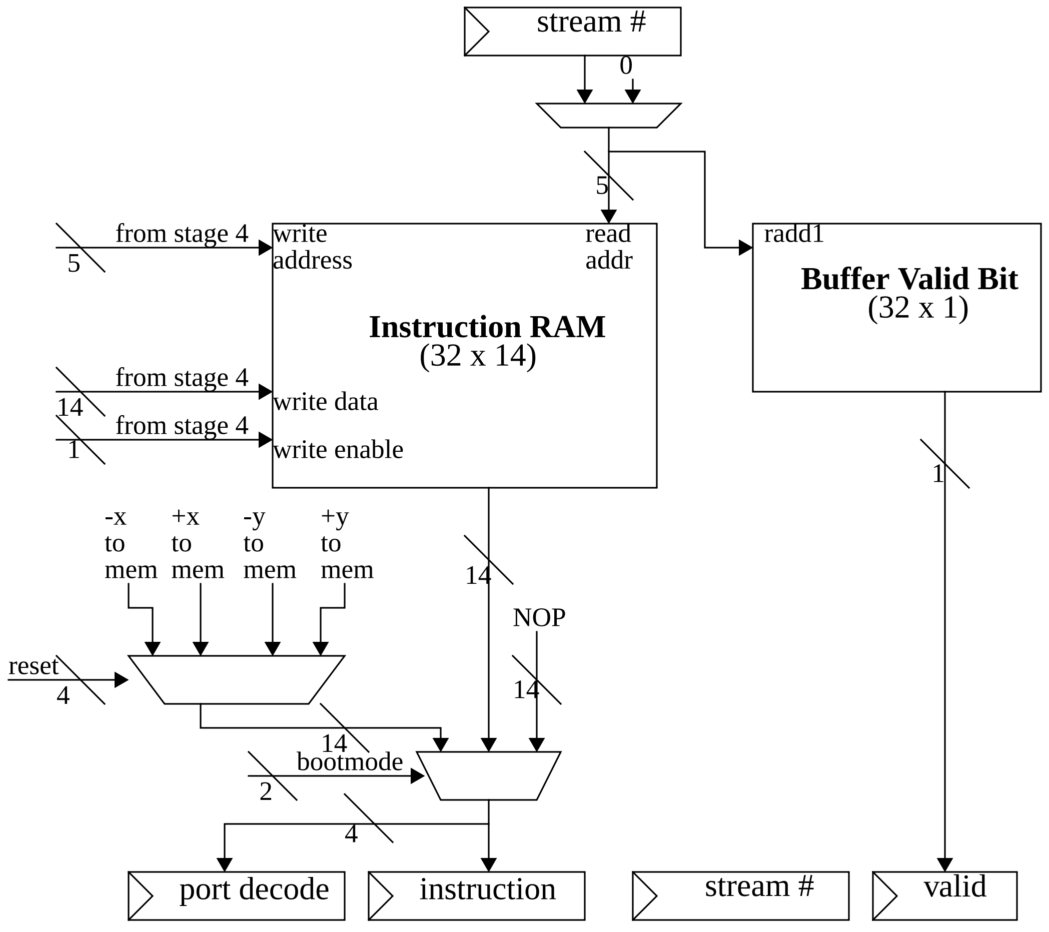


Figure 5.2 Stage 2 - Instruction Read

For purposes of limiting the routing complexity of the data path, a pipeline may write its own instruction RAM, but not the other pipeline's instruction RAM. Separate boot modes indicate which pipeline should be executing transfers. If a pipeline realizes that the other pipeline is booting, it splices NOPs into the instruction stream. Since data words are fourteen bits long and require a five bit address, a single thirty-two bit communication word can be transferred to the memory address space and write an entire instruction.

Finally, it was determined that the internode transfer time was the critical path in the CFSM. To allow port reads in the following stage to occur as fast as possible, the port enables are one-hot encoded at the end of stage two.

5.3 Stage 3: Reading Sources

The third stage of the pipeline requires a source word to be read from the encoded source of the instruction or the stream's buffer register, depending on the buffer valid bit from stage 2 and the opcode of the instruction. Figure 5.3 shows a diagram of most of the functionality required for the source read.

The buffer registers are read on every cycle. If a source encoding translates to another stream's buffer register, the valid bit read the previous stage and the opcode will determine which buffer register is read. If the opcode is **bmove** the encoded buffer stream will always be read. If an **fmove** instruction is used, a valid bit indicates that a word has backed up in the stream's buffer register, and the buffered word will be used. Similarly, if a **cmove** instruction is used and the read success indicates a failure, then the encoded source is discarded and the stream's buffer register is instead read.

The final source word can come from one of four places: the ports, the processor registers, the buffer registers, and the JTAG interface to the processor. If flow control is being used, the source word mux will read the buffer register if a backed up word is detected or if a **cmove** instruction requires a buffer read to mimic the previous instruction.

If the source location of the instruction is a port, the port enable will be asserted at the very beginning of the clock cycle. At the same time, the accept bit can be sent to the port from which the read is occurring, since the valid bit of the stream's buffer register is read on the previous cycle. Once a source word is chosen, it gets stored in a register for use by the next cycle.

The destination field of the instruction can specify a processor register or a buffer register for writing the source word. If flow control instructions are used, these writes must fail if a valid word already exists in the destination processor register or buffer stream. In

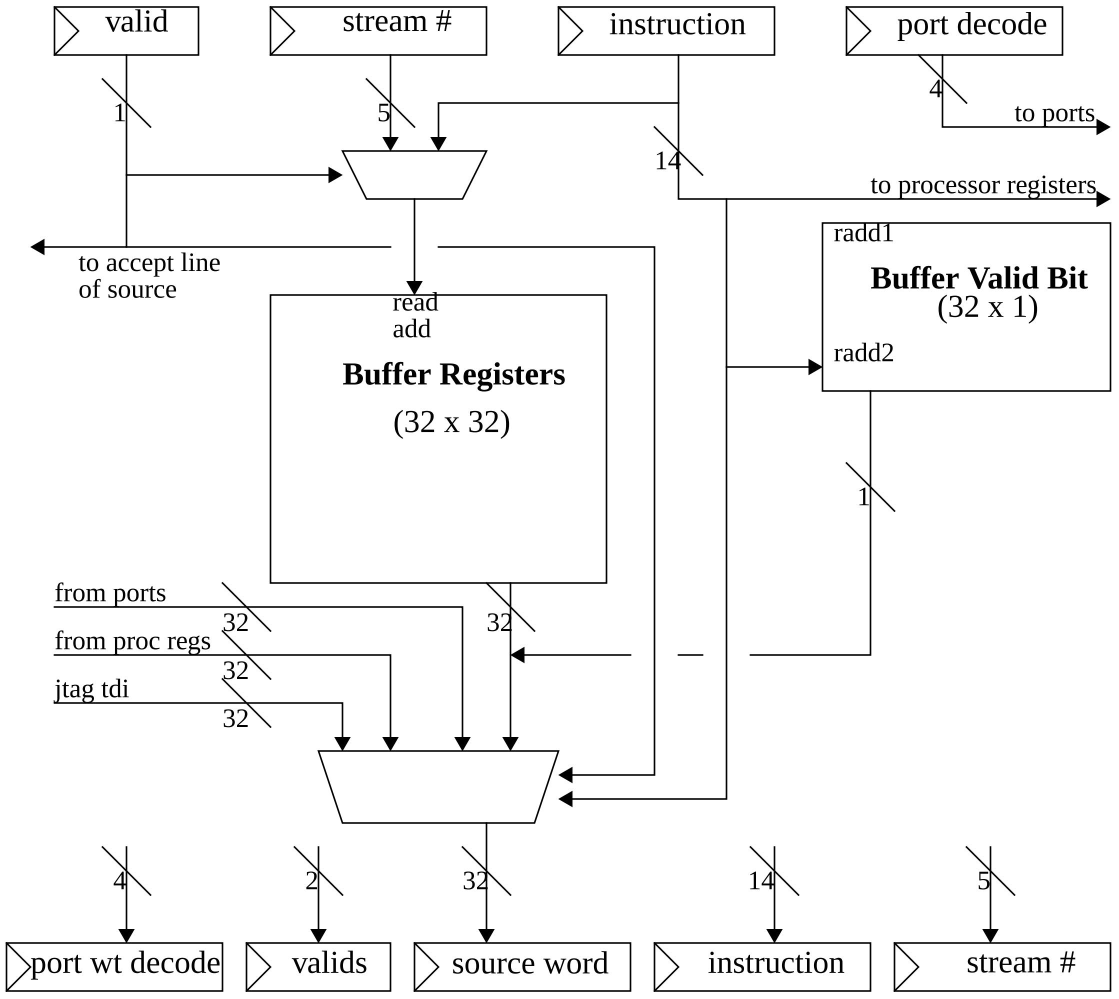


Figure 5.3 Stage 3 - Source Read

order to check this, these valid bits must be read during the read stage of the pipeline. In the case of the buffer registers, an extra read port is added to the valid bit RAM. Since this RAM is really a thirty-two bit register, the logic is fairly simple. The valid bits of the processor registers must also have an extra read port to read this bit a cycle early. These two valid bits get transferred to the fourth stage of the pipeline.

Since the internode transfer time dominates the cycle time of the system, the destination field of the instruction is decoded in the third stage to provide one-hot encoding for the drivers of the ports. These four signals are directly stored in registers that can immediately turn the drivers on at the beginning of the next cycle.

Some logic included in the design of the third stage is not shown for clarity. The buffer registers are bypassed in case a read and write to the same location occurs on the same cycle. Although the same stream can not be scheduled on consecutive clock cycles, the ability to specify other stream's buffers makes this a possibility.

The read success logic is also not shown. The read success bit can only be zero if the *accept* bit is not asserted or if the *valid* bit of the final source word is not set. The read success bit determines the behavior of the read stage if the **cmove** instruction is selected. It's main effect is on the mux that chooses from the various locations for the source word. It can also determine whether or not the stream is allowed to accept a word of input data.

5.4 Stage 4: Writing Destinations

The fourth stage of the pipeline is for writing destinations. Destinations can include the ports, the processor registers, the buffer registers, the JTAG interface, and the memory address space. A diagram of the fourth stage is shown in Figure 5.4.

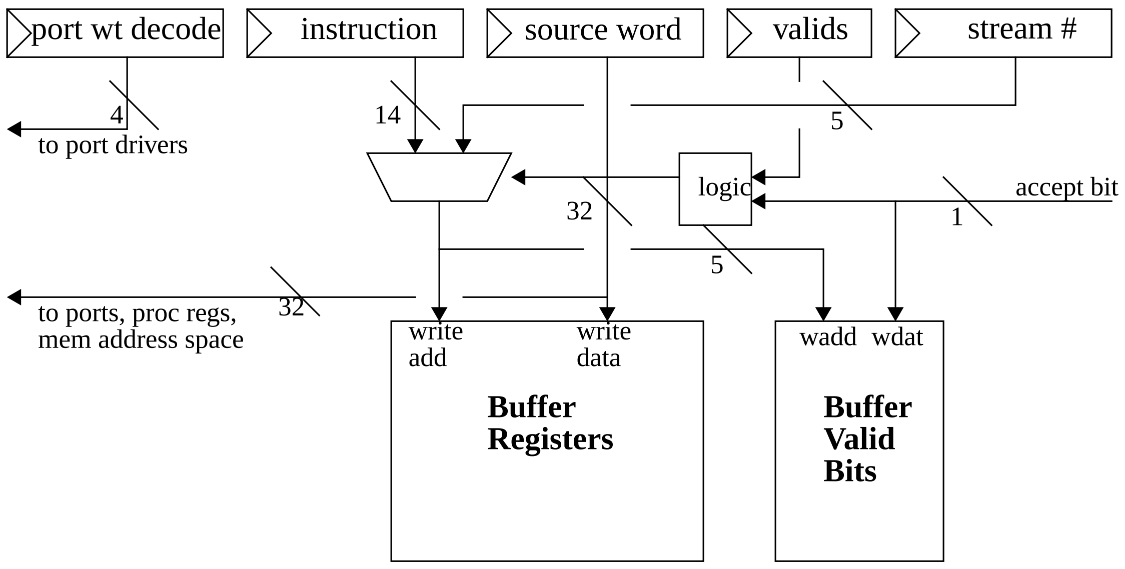


Figure 5.4 Stage 4 - Write Destinations

The *accept* bit is received from one of the ports or from any of the on-chip locations, and determines whether the valid bit of the stream's buffer register will be written. On every cycle, the buffer register is written. Normally the buffer register of the stream will be written, unless an alternate buffer register destination is encoded into the instruction. A mux selects between the stream's address and an encoded buffer address based on flow control semantics and the pre-computed *valid* bits. The incoming *accept* bit is directly connected to the data line of the buffer valid bit register, minimizing the amount of delay that this cross chip signal must undergo. The address can be decoded ahead of time, allowing the *accept* bit to be registered immediately.

The source word is connected to the data ports, the processor registers, and all of the locations in the memory address space. Since the validity of writes to alternate buffer registers or the processor registers is determined on the previous cycle and stored in a register, these bits can now act as write enables for the buffer and processor registers.

The logic for capturing the write success information for **cmove** instructions is not shown. The only way a write can fail is if the *accept* bit, whether from a connecting node or from an internal location such as the buffer or processor registers, is deasserted. The write success bit determines whether or not the *valid* bit of the stream's buffer register will be marked valid.

5.5 Memory Address Space

The memory address space is a single destination to which words can be transferred. Regardless of the flow control implications of the opcode, a write to the memory address space will never fail. The only requirement for the transfer to succeed is that the data word itself be marked valid. The memory address space includes a wide variety of physical locations. The upper three bits of the data word specify the space in which the data is to be written. In some cases, such as the schedule RAM, there may be multiple addresses within the space that are chosen by the remaining high order bits. The data for the actual write always starts from the least significant bit of the data word. Figure 5.5 shows the mapping of communication words to the memory address space.

5.5.1 Clear Valid Bits

When location zero is written in the memory address space, no data actually gets written, but a valid communication word is necessary for the transfer to succeed. All the buffer register valid bits and the processor valid bits get cleared. This function is useful for power up and for switches in communication phases. When a new communication pattern starts

Location	bits [31:29]	Addr Bits	Data Bits
Clear Valids	000	N/A	N/A
Sched RAM	001	[28:22]	[16:0]
Instruction RAM	010	[28:24]	[13:0]
Boot/Reset	011	N/A	[12:0]
NOP	100	N/A	N/A
Jtag/LEDs	101	N/A	[8:0]
Clk System Bits	110	N/A	[3:0]
Schedule Pointer	111	N/A	[6:0]

Figure 5.5 Memory Address Space

executing, communication streams may be completely rewritten. Since any buffered words refer to the old communication streams, a clear valid signal can clear all of the buffers at the same time.

5.5.2 Schedule RAM

When a word gets transferred to the schedule RAM location, the communication word provides both the address and data for the location to be written. Since the top three bits of the word specify the schedule RAM, the next seven bits specify the address within the schedule RAM that gets written. The bottom seventeen bits hold the data to be written into the schedule RAM.

This location is used to write new schedules, one schedule instruction at a time. In operation, one communication stream will periodically be reading a processor register or a port and trying to transfer words to the memory address space. When a new schedule is written, words are provided by the processor or another node and they get loaded into the schedule RAM. Even though a schedule is being created, the final switch does not have to happen until the entire schedule is created. To do this a schedule will be created in a part of the schedule RAM not being used by the current schedule. When the new schedule is finished, a final communication word can write the schedule pointer with the first address of the new schedule.

5.5.3 Instruction RAM

The instruction RAM is written in a similar fashion to the schedule RAM. The top three bits specify the pipeline’s instruction RAM, the next five bits specify the address within the RAM, and the bottom fourteen bits specify the instruction to be written. To save on datapath routing, one pipeline can not write the other pipeline’s instruction RAM. Future designs could easily be allowed to do this by adding a sixth bit of addressing that specifies the RAM to be written.

5.5.4 Boot/Reset

The boot and reset signals are combined into a single location. Three bits serve strictly to put either of the pipelines into boot mode. The high order bits corresponds to an enable that indicates the boot signals are valid. Each of the next two bits can put either of the two pipelines in or out of boot mode. If a pipeline is put into boot mode, it will attempt to read words from whatever port has its reset line asserted, and transfer them into the memory address space. If no reset line is asserted, the pipeline will be idle until a reset line is asserted. If a pipeline detects that it is not in boot mode, but the other pipeline is in boot mode, NOP instructions are spliced into the instruction stream. At power-up, the first pipeline is put into boot mode. After a node receives its communication words and the booting process is finished, a final word must be transferred to take both pipelines out of boot mode.

Bits 4-12 are used for the reset lines of the node. Each node has a bi-directional reset line going to each neighbor, requiring a total of four bits. Four additional bits are used for a reset mask. This mask can cause a node to ignore reset signals from certain ports. The mask is used for nodes at the end of the mesh or for connections to faulty nodes that may cause noise on the reset lines. In addition, since the reset lines are bi-directional, when a node pulls a line high, it must not think the reset is coming from a neighbor. This is prevented by masking out the reset from the port a node is driving. The last bit is used as an enable for writing both the reset lines and the reset line masks. This allows the booting bits to be handled independently from the reset lines. When a node is fully booted, it will usually be responsible for booting a neighbor. It does this by asserting a reset line to the node. This causes the neighbor to constantly try to read words from the booted node and transfer them into the memory address space. streams will have been set up in the booting node to allow for the transfer of communication words to the unbooted node. When the final word is transferred, the node should deassert the reset line, since the neighbor node is pulled out of boot mode.

At power-up, the reset mask is set to allow any resets, and the reset lines are deasserted.

5.5.5 JTAG/LEDs

One memory location space is reserved for writing the connecting processor's JTAG interface as well as writing any of four LEDs. A total of eight bits are needed. The bottom five bits are used by the LEDs. The fifth bit is used to indicate that a valid write to the LEDs is occurring. The next four bits go straight to the LEDs. At power-up, the LEDs are loaded with high values. This can serve as a quick check that the nodes are receiving power and that the booting process has begun.

The next four higher bits are connected to the processor's JTAG interface. The highest bit is an enable on the tri-state output drivers. By successively writing these JTAG bits, the processor can be initialized and even loaded with source code. Although this process is slow due to the serial nature of JTAG transfers, it is a one-time booting charge.

5.5.6 Clock System Bits

In order for the mesh to be synchronized, nodes must be added to a synchronous group in a pattern that prevents cycles from occurring. When a node receives a clock system bit, it tries to synchronize clocks with the node from which the clock bit came. Every node has one clock system bit line per neighbor. One of the first operations at power-up is to create a synchronous clock for the system. A host computer is connected to a node at one corner of the mesh. The node connected to the host becomes the first node in the synchronous group. By transferring the appropriate words to the memory address space, the node can assert the clock lines for any of the neighbors. Then these neighbors can assert clock system bits for their neighbors. The pattern for adding nodes to the synchronous group is under complete control of the software.

5.5.7 Schedule Pointer

A single location space in memory is reserved for the schedule pointer. The bottom seven bits of the communication word is used for this transfer. This location can be written to quickly switch between schedules in the schedule RAM. If the schedule RAM has enough room to store multiple schedules, writing the schedule pointer to switch schedules is far more efficient than creating an entirely new schedule by writing individual schedule RAM locations.

5.6 Datapath

The datapath on the chip can be described as two busses that can read or write almost every location on the chip. Each pipeline has control of one of the busses, and they shares resources such as the ports, scheduler, and the processor interface. At some level, the data-path required for the ports can be thought of as two pipelined crossbar switches, since a new word of data can be read from any of the ports on any clock cycle, and any port can be written by either bus on any clock cycle.

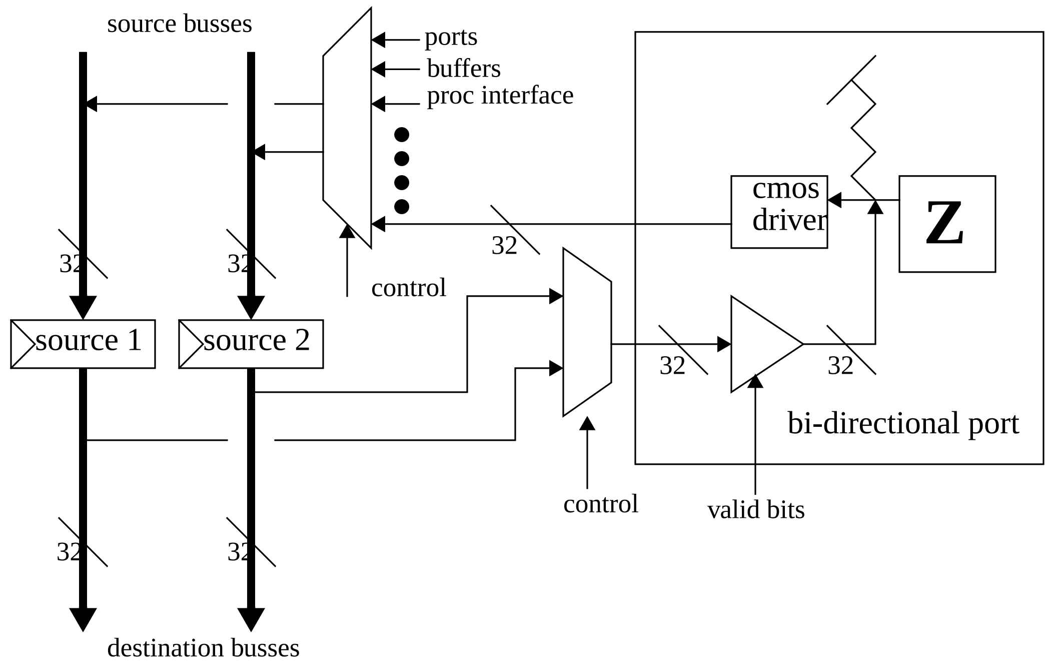


Figure 5.6 Datapath of Port Data

Figure 5.6 shows the datapath for one of the data ports. The ports are bidirectional, and a software violation must occur if two pipelines are attempting to write data to the same port on the same clock cycle. The pads have pull-ups attached to them to keep lines at a stable value when a port is not being used. A port is just one of the many inputs that can drive data on a pipeline's source bus. However, only the two destination busses can drive data onto the port. Control logic picks between the two busses based on whether or not either is attempting to use the port. The enable of the tri-state is connected to the *valid* bit of the source word being written to the port. If a word is not marked valid, then no information will be driven on the ports. Both the source and destination busses go to many locations on the chip since they are the primary vehicle for transferring information. It is the routing of these two busses that leads to the complications in routing the design.

The *valid* and *accept* bit drivers and receivers are implemented as a single driver and a single receiver per port. On any given cycle, at most one pipeline can be either reading or writing any one port under normal operation. Pipeline reads requires an *accept* bit to be sent while pipeline writes requires a *valid* bit to be sent. On any clock, only one of these four signals can be active. This makes the logic for the valid/accept line simply the logical OR of the two *valid* bits and the two *accept* bits. Since the communication is scheduled, the node connected to the other side of the port will be performing the complementary action since both nodes reading or writing the port is an invalid operation. One important result of this implementation is that a a single directional line can be used for transmitting either an *accept* or a *valid* bit. On every clock cycle, the meaning of this line can change. There is one such line going in each direction between nodes. Since single directional lines are faster than bidirectional lines, this allows the flow control protocol to operate more quickly.

Each port has a single valid/accept receiver that can change meaning on every cycle. Although the data ports are bi-directional, data can flow only in one direction on each clock cycle. Since the communication is scheduled, the receiving node knows whether the transmitted bit is a *valid* bit or an *accept* bit. Similar to the single transmitting line, the fact that the receiver can be implemented as a single directional line speeds up the chip's operation.

The datapath diagrams do not show how fork operations or pipeline transfers occur. A fork operation requires that the same data be read and written on the same pipeline on the same clock cycle. In addition, a neighbor node may also be reading the result of the data write. For a transfer to succeed, both the internal reading stream and the external reading stream must assert *accept* bits. This bypass condition is detected a cycle early and the source and destination busses are bypassed accordingly.

Similarly, a pipeline transfer requires one pipeline to be writing a port while another pipeline reads the port. When this operation is detected, a neighbor node may not be part of the transaction. The bypass paths between the source and destination busses for this case also exists. Also, for the pipeline transfer, an external *accept* bit must be ignored, and the *accept* bit must come strictly from the reading pipeline. The valid/accept lines are bypassed to handle this case.

5.7 Processor Interface

The processor interface supports sixteen registers that are mapped into the memory address space of the processor. The CFSM can reference the locations in the processor register file. Sixteen locations are encoded in the instruction word for reading and writing the processor registers. Full flow control semantics are supported on the CFSM side of the interface. For flow control semantics to be followed on the processor side, either the *valid* bits of the registers must be explicitly checked before a transfer is made, or an interrupt to the processor can inform the unit when certain flow control characteristics exist.

Each pipeline has a single port to access the processor register files. This creates the constraint that on any cycle, a pipeline may be reading or writing the processor registers, but not both. Due to the nature of the pipeline, this means that an instruction that reads the processor registers can not follow an instruction that writes the processor registers. This constraint allows the processor register file to have only three ports rather than six, but comes at a cost of increasing the scheduling complexity. The processor registers and valid bits are all bypassed to allow for maximum transfer rates if the same locations are being read and written at the same time.

The processor register valid bits consist of a sixteen bit register that indicates what registers hold valid data. These *valid* bits actually have six ports due to the requirement of

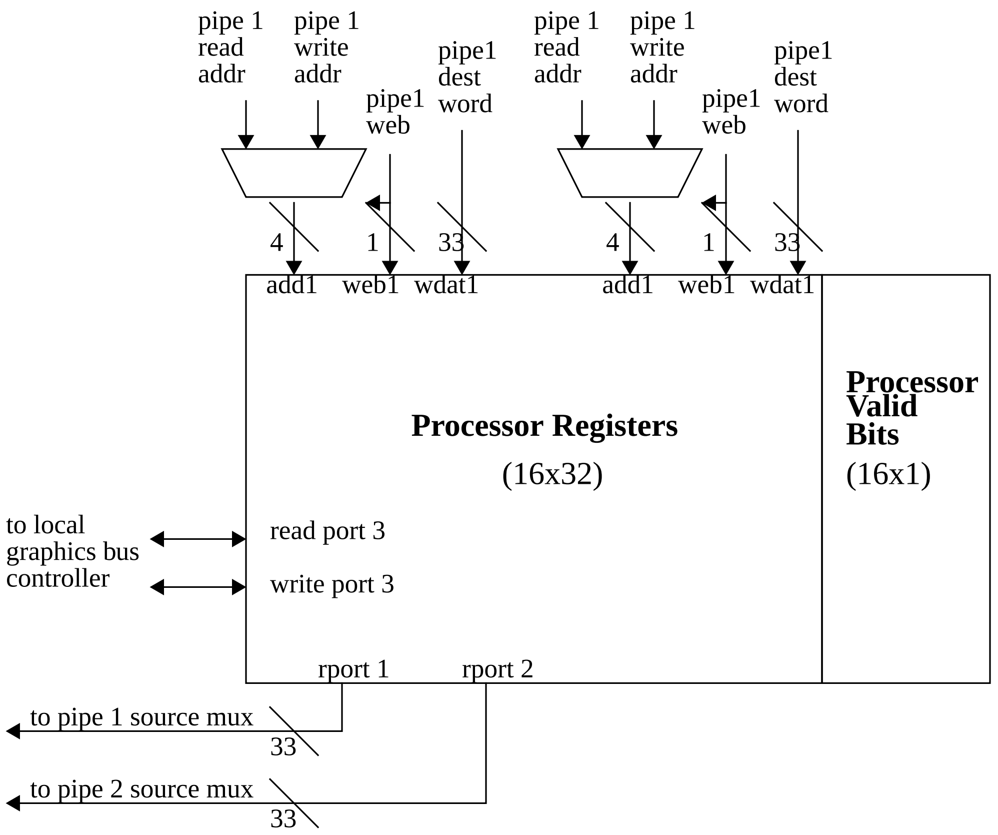


Figure 5.7 Processor Interface Register

flow control. When writing a processor register with an **fmove** instruction, the *valid* bit of the register gets checked during the read stage of the pipeline. Since an actual read or write to the processor registers can be occurring in the pipeline, an added port to the valid bits must be added to allow this flow control check. Since the *valid* bits are made up of simple logic, this requirement does not add to the critical path.

The interface between the register file and the attached processor of the node is a much more difficult decision. Standard bus interfaces for microprocessors were first considered as a solution to the processor interface problem. The S-Bus and M-Bus interfaces are both well defined and supported by a wide variety of processors. However, both protocols have tremendous overhead resulting in a single transfer taking tens of processor cycles to complete. Ideally, the processor interface should be able to transfer words at the same rate the network can transfer words to the processor interface. If it takes tens of clock cycles to transfer words between the network and the processor, the benefits of scheduled communication gets drastically reduced.

Fortunately, there exist busses with much better performance. The bus chosen for the NuMesh prototype nodes is the Local Graphics Bus, defined by Sun Microsystems (also called the AFX bus). The Local Graphics Bus is designed to support high throughput graphics communications for the micro-SPARC family of processors. By deciding to use the Local Graphics Bus, the types of processors that can be connected to the NuMesh network are members of the micro-SPARC family and other computation units that can observe the Local Graphics Bus protocol. While this is not an ideal general solution, it works well to demonstrate the prototype NuMesh system. Active research on a better processor interface is ongoing.

The AFX bus has several desirable characteristics. Write accesses have a latency of two cycles, although multiple writes can be pipelined such that successive writes take a single cycle each. Read accesses take two clock cycles. The AFX bus targets 256 Mbytes of address space in a system memory map. The data lines for the bus are shared with the main memory bus of the processor and up to eight bytes of data can be transferred at a time.

The local graphics bus (AFX bus) is implemented by adding simple logic to the third port of the processor registers. The signals required to implement the AFX bus are included in figure Figure 5.8. The sixteen processor registers are mapped into the memory locations of the processor such that whenever an application tries to access locations in a certain range, a bus access is initiated between the processor and the CFSM.

While the local graphics bus allows for a variable response time when requests for reads or writes are made, the NuMesh system can guarantee the quickest possible access time, since internally, all that is occurring is a simple read or write access to a dedicated access port. This greatly simplifies the logic needed to support the bus, and maps quite easily to a standard address/data protocol that other computation units such as DSPs or graphics chips can take advantage of. By tying certain signals high or low, the bus protocol can be greatly simplified to support more simple protocols.

Figure 5.9 shows the logic necessary for the local graphics bus. To implement the local graphics bus protocol, a pipeline is formed such that the CFSM can support the maximum rate of transfers from the processor. The AB lines specify the address of the register being

Signal Name	Function	Driver
AEN	Address Enable - indicates the address lines contain valid data	Processor
AB[4:0]	Address Bits - Used to select a particular address from the processor interface	Processor
LO_ADDR	Low Address Select - Serves to indicate which address byte is being sent. NuMesh ignores address if high bytes is indicated	Processor
P_REPLY[1:0]	Indicates when Slave is sending valid data	NuMesh
S_REPLY	Indicates when processor is sending valid data	Processor
INT_L	Interrupt line to processor	NuMesh
DB[32:0]	Thirty-three bit data bus (one bit for optional valid)	both

Figure 5.8 Control Lines Between Processor and NuMesh

accessed. If both the AEN and the LO_ADDR are active, then a valid access is beginning. The SREPLY bits indicate whether a read or write access is occurring. Since the SREPLY signal can come at unspecified number of clock cycles after the address, the address must be registered until a new access starts. This is accomplished through the feedback mux of the address register. If the SREPLY bits indicate a write access, the write data will appear on the following clock cycle. This data can be registered and the write will complete on the following clock cycle. A read can also finish on the cycle following the request. As a result, the SREPLY bits can quickly be turned into the PREPLY bits indicating to the processor that the accesses have been accomplished. Using this logic, the CFSM can support a read on every other clock cycle, and a write can be supported on every clock cycle, after the initial two cycle delay of the first write.

Logic is not shown to support the interrupts and the reading of all *valid* bits. When the processor writes location twenty-four, the interrupt mask is written. The bottom sixteen bits of the mask check for empty locations while the top sixteen bits check for full loca-

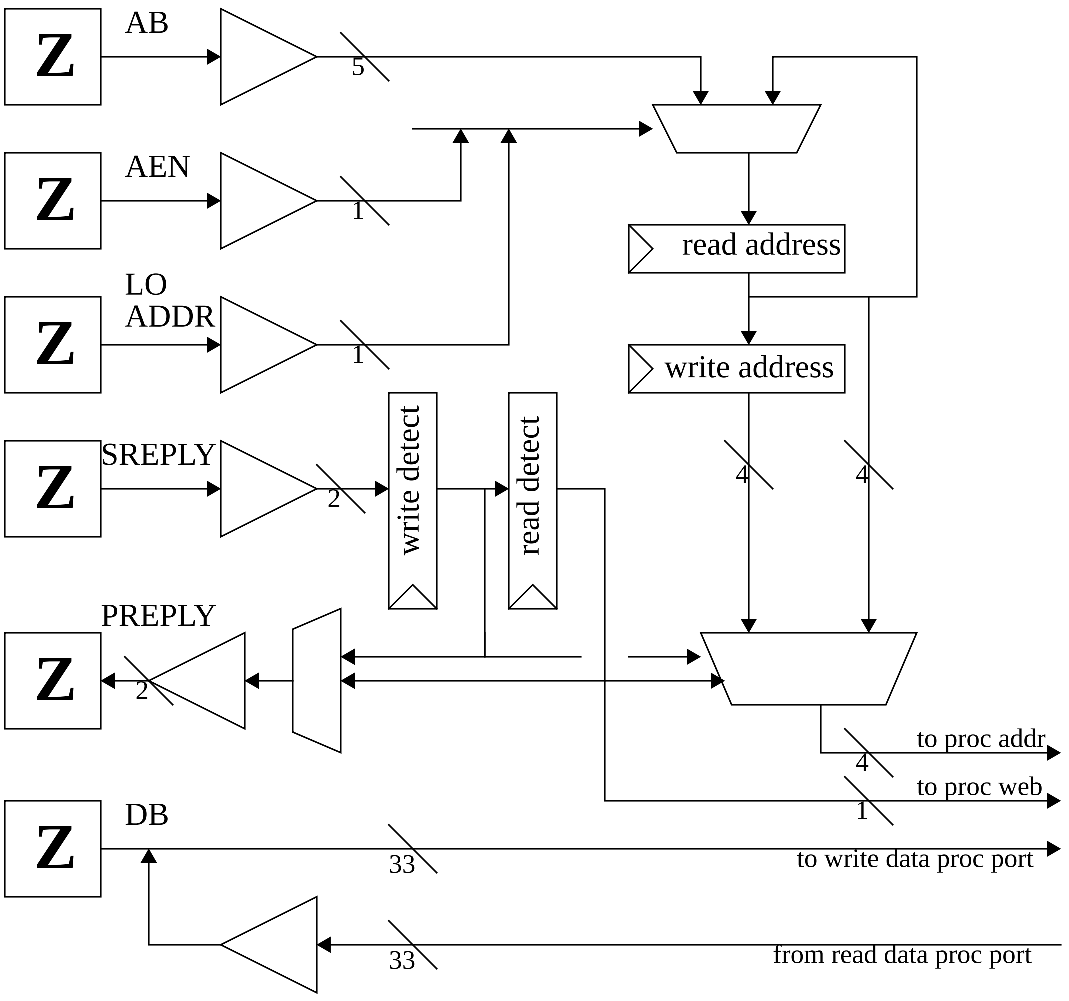


Figure 5.9 Local Graphics Bus Logic

tions. A one in any interrupt mask bit location indicates that the condition is active. If any of the *valid* bits meet the conditions checked for in the mask, the interrupt bit is asserted. Figure 5.10 shows an example of how the interrupt mask works.

The processor can read address sixteen to get a sixteen bit data word consisting of all the *valid* bits of the processor registers. An application might access this word to determine which messages have arrived, and then dispatch to different code based on this result. This mechanism is much faster than the interrupt mechanism, since an interrupt can take many cycles to take effect.

Interrupt Mask

```
0000000010001000 0001000000000001
```

An **interrupt** will be asserted if:
- processor register 0 is not valid
- processor register 12 is not valid
- processor register 3 is valid
- processor register 7 is valid

Figure 5.10 Interrupt Mask Example

Any time the processor reads a processor register word, the *valid* bit gets sent as a thirty-third bit. If the processor reads a double word, this bit will appear in the application as the second word. In this manner, an application can try to read messages and check for their validity based on this extra transferred bit. If a single word access is attempted, the extra bit is simply ignored.

5.8 Booting Process

Mechanisms to facilitate the booting process have been described throughout this chapter. This section will serve to give a step by step description of how a mesh gets booted, from power-up to the running of an application. For this example, the mesh pictured in Figure 5.11 will be used.

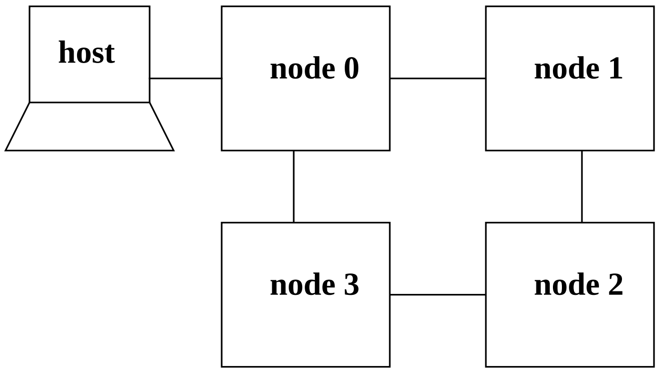


Figure 5.11 Network for Booting Example

The first step of the booting process involves the power-up signal that occurs when the system is turned on. Through an RC circuit, all nodes receive a power-up signal that causes them to go into boot mode. The particular boot mode chosen is for pipeline one. Since no reset signal is asserted, each node will be idle, waiting for a reset line to go active on one of its ports. The power-up signal clears all of the buffer and processor register valid bits that might accidentally get set, sets the LEDs to active, clears out the processor interrupt and reset masks, and deactivates all system bits.

Next the host computer asserts the reset line for node 0. This causes node 0 to constantly attempt to read words from the negative X port and write them to its memory address space. The host can now send words to any locations in node 0. If words need to be sent to the second pipeline, the host can send a control word that switches booting control between the two pipelines. The processor attached to node 0 gets initialized by the host computer writing words to the processor JTAG interface. Rather than setting up the final communication schedule for node 0, a pipe must be set up so that node 1 can receive words from the host computer. Essentially a single stream can be constantly executed on node 0 that transfers words from the negative X port to the negative Y port.

Once the host has control of node 0, it can send a word to node 0 that causes it to assert the clock system bit that is connected to node 1. After a few clock cycles, it can be assumed that node 0 and node 1 are in sync. When this has occurred, the host can write a word to node 0 that causes the reset line between node 0 and node 1 to be asserted. This causes node 1 to transfer words from node 0 into its memory address space. Node 1 can be set up with a path that allows the host to write the memory address space of node 2. Again the clocks are synced, and the reset line is asserted. Eventually, all nodes will be added to the synced clock network, and final communication schedules and instructions can be loaded to the nodes. The paths for loading the more remote nodes can be successively created, until all nodes have their final schedules and instructions loaded. Since the host computer has control of all data transfers, and words are not transferred until the global clock is established, the exact timing that each node is taken out of boot mode can be exactly known. When the last node is taken out of boot mode, all nodes are in sync and the application can begin. Nodes can periodically be rebooted by a processor if a new schedule or new instructions need to be loaded, but it is expected that boot mode will only be used before the application begins, and any subsequent dynamic behavior would happen by a node repeatedly writing its memory address space.

5.9 Packaging and Pinout

The package selected for the CFSM chip was a 299 Pin Grid Array (PGA). One of the bigger constraints when choosing a target technology was finding one that could be fit into a package with enough pins. The Chip Express pad design allows pull-up or pull-down resistors to be attached to every pin, and a variety of slew-rate controlled drivers are supported.

Since the CFSM architecture supports four neighbors as well as the processor interface, many pins are needed for its implementation. On any given clock cycle, many of these pins may not need to be driven. In order to prevent signals from degrading to metastable values, pull-up resistors are attached to the data pins to prevent current spikes. Since it is also possible for many pins to switch at once, all drivers are implemented with a high slew-rate control to prevent current spikes that might cause ground bounce in the chip.

The power-up pin controls many of the initialization functions on the chip, and should only be active when the chip is first turned on. To prevent noise on this pin from accidentally reinitializing the chip, the receiver for this pin has added hysteresis. Since the power-up signal is active for several clock cycles while noise on the pin should be fleeting, this technique serves to isolate the signal from the noise.

A complete listing of all signal pins on the chip is included in Figure 5.12. A total of 200 pins are needed for signaling on the chip. An additional fifty pins are used for power and ground signals on the chip.

5.10 Constraints in the Chip Express Design Flow

Chip Express constraints added two significant changes to the design of the CFSM. The RAMs in the Chip Express design require that the read address be latched in on the clock cycle preceding the cycle in which the data is expected to be offered. The design of the pipeline assumes that the read operation of the RAM is combinational, and that the address can be created at the beginning of the cycle in which data is expected. In most cases, the difference is irrelevant since the read address in the design is coming straight

Signal	Number of Pins	Direction	Attached Resistor?
Data Ports	32 x 4 = 128	Bi-directional	Pull-Up Resistor
Valid/Accept Input	1 x 4 = 4	Input	Pull-Down Resistor
Valid/Accept Output	1 x 4 = 4	Output	None
Clock System Bits	1 x 4 = 4	Output	None
Reset Lines	1 x 5 = 5	4 Bi-directional 1 Output (proc)	Pull-Down Resistor
proc JTAG port	1 x 4	3 Output 1 Input	None
AFX AB(address)	5	Input	None
AFX DB(data)	33	Bi-directional	None
AFX AEN(enable)	1	Input	None
AFX ADDR_LO(enable)	1	Input	None
AFX SREPLY	2	Input	None
AFX PREPLY	2	Output	None
Processor Interrupt	1	Output	None
Power_reset	1	Input	None
LEDs	4	Output	None
System Clk	1	Input	None

Total Pins = 200

Figure 5.12 Pinout Listing of Chip

from a register. For these cases, the register is effectively moved into the RAM module. In some cases in the design, some combinational logic occurs between stages and the address is figured out early in one of the stages. For these cases, the addresses must be precomputed on the previous cycle in order to be stored in the register of the RAM module. Fortunately, only minor modifications must be made to allow this to work, and the critical path is not affected.

The second constraint is much more serious. The system design for clock distribution can minimize skew between the clock signals of any two nodes. On each node, the clock signal is an input to the CFSM chip. Additional and unpredictable skew delay can occur when the clock goes through a pin and a receiver. Traditionally, a phase-locked loop can force the external and the internal clocks to match. Unfortunately, the Chip Express logic family (CX2000) did not support this phase locked loop. As a result, the skew between the clocks became unacceptable. Consider the example in Figure 5.13. Node 0 is attempting to transmit data to node 1. Ideally, an entire clock cycle is allowed for the transfer. Data is latched on rising edges of the clock. In the example, if the rising edge of node 1 occurs much after the rising edge of node 0, the data being transmitted by node 0 may start to change before node 1 has a chance to latch the data.

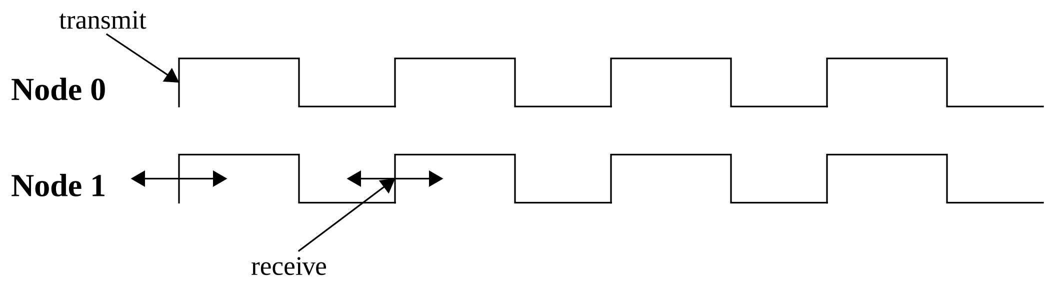


Figure 5.13 Clock Skew Can Cause Transmit Errors

Some amount of skew can be tolerated due to the hold time characteristics of the logic in the transmit path. The absence of a phase-locked loop causes enough skew potential to keep the hold time delays from solving the problem. As a result, a new clocking scheme was implemented. All nodes transmit on the rising edge of the clock, but receive data on the falling edge of the clock. Consider the same example as before, in Figure 5.14.

In this example, since the transmit and receive edges do not line up, clock skew can not cause the wrong data to be latched, although it can reduce the effective amount of time allowed for the transfer. The biggest drawback to this system is that only half a clock cycle is allowed for a data transfer. This causes the internode transfer time to be the critical path in the CFSM.

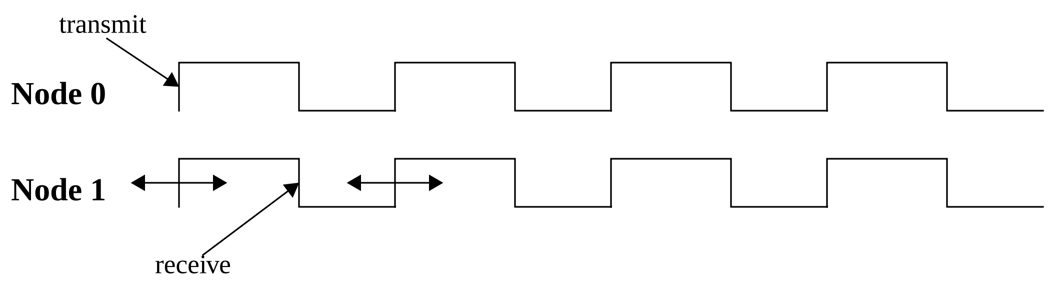


Figure 5.14 Overcoming Clock Skew

Chapter 6

Conclusion and Results

The goal of this thesis is to show the design and implementation of an architecture capable of efficiently supporting scheduled communication. Virtual streams of communication are extracted from an application at compile time. These virtual streams indicate a source processor, a destination processor, and a requested bandwidth of communication. A directed graph of communication can be extracted from these virtual streams, and a global schedule of communication can be created that choreographs communication in the network. By taking advantage of scheduled communication information already present in many applications, the NuMesh system can minimize congestion by intelligently scheduling the virtual streams. In addition, the clock cycle of the network can be reduced due to the absence of data-dependent decisions and a simple flow control protocol. Although the communication network is static, the processors attached to each node are unpredictable in nature and can inject and remove messages at arbitrary times. A flow control protocol is supported to allow messages to back up in the network without affecting the choreographed schedule of the network.

The rest of this chapter is devoted to summarizing the thesis. First the major architectural ideas are discussed. Then the design flow for the actual chip is described. Once the chips were received a variety of testing environments were created. Finally, some effort is made to evaluate the architecture based on the two fundamental goals of reducing the network router's cycle time and minimizing congestion. The chapter ends with some ideas of future work.

6.1 Architectural Features

This thesis discusses several architectural ideas unique to a scheduled communication system. A goal of this thesis was to walk through a series of architecture decisions that were made when designing the system. It is hoped that future designers of scheduled communication architectures can take advantage of some of these features. This section will recap some of the more important features of the NuMesh architecture.

6.1.1 Virtual FSMs for Virtual Streams

Applications can support a variety of communication models that result in very different numbers of virtual streams. A goal of the NuMesh architecture is to support an arbitrary number of streams without performance being lost for applications that require fewer or greater numbers of streams. The NuMesh CFSM supports two physical pipelines of communication that allow two different virtual streams to operate on every clock cycle. Since the NuMesh system supports four ports, this allows all four port to be used on every clock cycle. In addition, each of the pipelines can be time-multiplexed to support up to thirty-two virtual streams each. The architecture does not put a limit on the number of virtual streams, but the implemented chip uses a total of sixty-four due to memory constraints.

By arranging the NuMesh CFSM as a combination of many small virtual FSMs, each supporting a single virtual stream, the number of state bits required to keep track of the global communication FSM is reduced. In addition, a scheduler can assign each virtual stream an arbitrary amount of bandwidth without having to design a complicated global FSM.

6.1.2 Flow Control Protocol

Once it was discovered that words could back up in the communication network, flow control needed to be implemented. A novel protocol is introduced that allows flow control to occur on a single communication word. The protocol only requires a single internode transfer to complete and only requires two bits of information to be exchanged between the nodes involved in the transfer.

The flow control scheme requires some amount of buffering for each virtual stream, since each virtual stream has the potential to be blocked. A single word of buffering is provided for each virtual stream. The flow control protocol dictates that a virtual stream is not allowed to accept new data if its buffer register is full. Since the virtual streams are scheduled at compile time, the CFSM can decide a cycle earlier whether a virtual stream will accept a new word of data. At the same time the transmitting node is sending a data word along with a single *valid* bit, the receiving node can transmit a single *accept* bit that indicates whether the stream will accept new data. Instead of two cycles or multiple buffers being required as in traditional dynamic routing systems, the NuMesh architecture can accomplish the handshake in a single clock cycle and with one buffer per virtual stream.

6.1.3 Scheduler

One might have assumed that the CFSM state memory could have consisted of a single RAM containing a program of instruction to be executed during run-time. Each instruction could correspond to the actions of a single virtual stream, and the streams could be scheduled for greater bandwidth by being simply replicated in the loop as needed.

This thesis showed the benefits of decoupling the scheduler state from the virtual stream instructions. Since each virtual stream must have its own buffer storage, there must be a distinction between two streams that travel the same path through a node and a single stream that is scheduled twice. The CFSM architecture defines a single instruction for up to sixty-four virtual streams. The scheduler simply decides which of these sixty-four streams gets scheduled on every cycle. The scheduler forms an outer loop of control across all the CFSMs and forms a global schedule. Another advantage of decoupling the scheduler from the instructions is that either can be changed without a complicated program code needing to be updated.

6.1.4 Processor Interface

Since the static communication network has to interact with the dynamic timing of the processors, there must be an interface to synchronize the two. A register file of shared memory locations is described, which allows each virtual stream some number of locations to store words at each end of the communication path. A variety of techniques are described to reduce the number of cycles that words spend waiting in these shared memory locations. Mechanisms for interrupts, polling of all the streams, and single stream flow control are described, and the benefits of each are discussed.

6.1.5 Dynamics in a Scheduled Architecture

Static routers require very precise timing between all processors and the communication network. This thesis shows how dynamic behavior can be incorporated into a static communication network. Virtual streams are assigned certain clock cycles for operation at compile time. If the dynamic timing behavior of the destination processor of a virtual stream prevents the node from removing messages from the communication network, messages back up along the communication path according to the flow control protocol described in 6.1.2. An important result of this protocol is that when words are backed up and stored in buffers, the entire operation occurs within the scheduled clock cycle of the virtual stream that is operating. On the following clock cycle, a completely different set of virtual streams may be operating and the blockage from the previous virtual stream has no lingering effect on the timing of the network. This allows schedules to be created at compile time without any regard to the fact that virtual streams can behave dynamically in that they may be backed up in the network depending on the processors' behavior.

Several mechanisms are described to allow for run-time changes to the static schedule. Both virtual stream instructions and CFSM schedules can be written during operation of an application. Since these writes can only occur at precise times that are set up at compile time, it is possible for the entire system to change its communication patterns at the same time. Multiple schedules and extra sets of virtual stream instructions can be stored in the CFSM at compile time, allowing a single write to completely change the static schedule being run on a node. For more subtle changes, individual schedule slots and virtual stream instructions can also be written by the processor as they are needed during an application.

6.2 Design Flow

The process of implementing the chip involved several stages of design. High level C simulations of a variety of architectures were implemented and refined. Separate hardware modules were created as modules for the *nsim* [Metc95] simulator environment. Once the communication needs for a wide variety of communication strategies were identified, the architecture could be tweaked to optimize for the common case.

Once the basic architecture was identified, behavioral verilog code was developed in order to provide a detailed picture of the CFSM. While the verilog code was too complicated to simulate many nodes concurrently, a small number of nodes could be booted and provided with snippets of communication traffic obtained from the high level simulator.

Next, the target architecture needed to be selected. The biggest constraint on the design turned out to be the required pin-out of the design. Originally, a six-neighbor 3-D cartesian network was planned. The 3-D diamond network was selected as an interesting compromise that provided fewer ports as well as a chance to explore a novel topology. Traditional university chip designs use the MOSIS design process. The MOSIS process did not provide a package that could support the pin-out of the design. FPGAs were explored as an alternative to custom chips due to their ability to allow quick experimentation on the hardware. However, the inefficient routing of FPGAs and their slow cycle times made them a poor choice. Even a scaled down version of the CFSM consisting of a single pipeline and limited stream support could not easily fit on an FPGA. As a compromise, the Chip Express laser-cut technology was chosen. They were able to support the pin-out demands of the design and provide performance significantly better than FPGAs, although not quite as good as custom ASICs. Once the target technology was chosen, the behavioral verilog was changed into structural verilog that met the requirements for the Chip Express technology.

Once the structural verilog was simulated with a set of benchmarks, the verilog design was fed into Synopsys for synthesis. While Synopsys was able to turn the verilog description into a Chip Express netlist, critical paths needed to be hand-tuned to reduce delays. The critical path at this point was identified as time required to get data from one chip's drivers through another chip's receivers and latched into a register. Only estimated delays were used at this point, since the design had yet to be physically routed. Once hand-tuning possibilities were exhausted, the design was shipped to Chip Express for routing. Eventually, the routing succeeded and an actual delay file was returned. This data file determined the clock cycle of the chip at about 37 MHz under worst case conditions.

6.3 Testing

Testing for the chip design occurred at three separate levels. Chip Express was provided with over five thousand test vectors that tested the basic functionality of the chip. Before chips were returned, these test vectors were run on the chip at 1 MHz. These vectors were comprised of the same tests that were run on both the behavioral and structural verilog. Each benchmark involved a reboot of the chip and a full loading of chip state from the datapath. These benchmarks are included in Figure 6.1.

Once the initial chips were returned, two testing environments were created in the MIT lab. The first involves a wire wrapped FPGA-based board. A small Xilinx 4005-PC84 FPGA is connected to a host computer and its pins are connected to one of the ports of the CFSM chip. The FPGA can control the CFSM clock, reset, and power-up pins. Words can be downloaded into the FPGA which acts as a synchronous FIFO and supplies words to the pins of the CFSM. In this manner, arbitrary benchmarks can be downloaded into the FPGA and subsequently used to boot and run sample communications patterns on the CFSM. The LEDs can be used as a final destinations for words that get transferred to various parts of the chip to allow for a quick indication if a benchmark succeeded.

The second testing environments takes advantage of JTAG boundary scan. An additional testing feature is added to the design of the chip. All of the pads around the edge of the chip are linked together in a serial chain as indicated in Figure 6.2. A JTAG controller is included to provide control over this scan chain. The JTAG controller can override the value currently on each pin with a user-controlled value that gets serially shifted. In this manner, every input pin can be supplied with a user defined value without any additional external connection to the pin. Similarly, the output value of each pin can be latched and read out through the same serial shift chain, allowing the user to see the state of all the pins without needing any external connections to the pins.

Five external pins on the chip are used to control this interface. By asserting the appropriate JTAG opcode, the chip will use either the serially shifted data or the current values on its pins for an upcoming clock cycle. At the end of a clock cycle, all of the values of the pins can be shifted back out and examined. Special software allows the mechanics of the interface to be abstracted away by a series of high level commands. The user can simply write benchmarks and execute them, examining the state of the pins at any point in the code. Using this technique, any of the original benchmarks can be run and all ports can easily be tested. Multiple chips can be connected, and the JTAG boundary chain of both chips can be connected to form a single boundary scan chain. This allows more complicated benchmarks to be executed with extensive debugging possible at any stage of the benchmark.

Benchmark	Description
Buffer_Write	Words transferred from port to buffer register
PReg_Write	Words transferred from port to processor register
Buffer_Read-Write	One thread writes buffer reg while another reads
Buffer_1pipe-2read	Consecutive threads try to read a buffer register after words are transferred there from port
PReg_1pipe-2read	Consecutive threads try to read a processor register after words are transferred from their from port
Buffer_1pipe-2write	Consecutive threads try to write same buffer register using flow control
PReg_1pipe-2write	Consecutive threads try to write same processor register using flow control
Preg_2pipe	Multiple threads in each pipeline access the same processor register
Bmove	Combinations of blind moves occur between ports
CMove	Messages of various sizes are transferred under various flow control conditions
Fork	Words are forked among various ports under various conditions
Pipeline Transfer	Words are transferred between pipelines
Memory	Various memory locations are written including scheduler, instructions, LED, clk sys, and JTAG
Proc_test	Words are exchanged between the ports and processor through the interface registers
Proc_Interrupt	Words are transferred through interface registers such that mask conditions are met and interrupt fires
Proc_Valid	Processor reads all valid bits over Local Graphics Bus
2node	Traffic is routed between two connected nodes
4node	Traffic is routed between four connected nodes

Figure 6.1 Description of testing Benchmarks

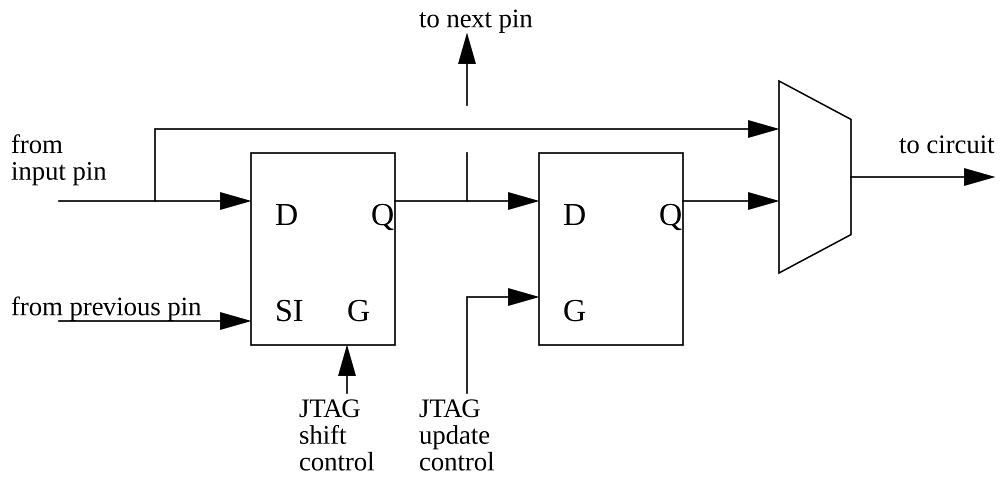


Figure 6.2 JTAG Boundary Scan (Input Pin)

6.4 Cycle Time

One of the goals of this thesis is to produce a communication architecture that allows for a very fast cycle time. Special care was spent to reduce all critical paths to either an internode transfer or a small RAM access. The architecture presented meets this goal. The implemented chip achieved a maximum frequency of just under 37 MHz. However, this chip is designed as a prototype for a scheduled communication system. Since a target architecture of a laser-cut gate array was chosen, performance was compromised. A more aggressive full-custom implementation could have drastically increased the frequency of the design.

Comparing the maximum clock frequency of the NuMesh CFSM to other dynamic routers is a difficult proposition since a wide variety of technologies and implementations are possible and can confuse the comparison. The main advantage of the NuMesh CFSM is the lack of data dependent run time decisions and the ability to perform flow control operations with a single internode transfer. A paper out of Illinois [Aoya93] made a comparison to dynamic routers, all implemented in the same technology. In the paper, each contributor to the clock period was identified. Since the contributors to the delay for the

NuMesh CFSM have been fully specified, it is possible to compare the performance of the Numesh CFSM to a variety of dynamic routers.

The paper from Illinois describes a base-line adaptive router that allows two degrees of freedom. While a traditional deterministic router routes messages along dimensions in a particular order, their base-line adaptive router allows the router to choose from between two dimensions to advance the message. Messages are routed among planes in a particular order, but not dimensions.

In the design of adaptive and deterministic routers, the delay can be broken down to two components. The first is the intranode, or internal delay a word has to undergo before being sent. For a deterministic router, this delay might simply be an examination of the header address and the overhead of choosing the appropriate channel for transfer. An adaptive router might have a few more decisions to make depending on how many degrees of freedom the adaptive router can choose between. This intranode delay can further be broken down into two components. The first is termed the *path setup* while the second is called the *data through* delay. The *path setup* delay refers to the time the router takes to set up the crossbar transfer based on the decoding of the header address. For the Illinois base-line adaptive router, the *setup path* delays are included in Figure 6.3.

Delay Component	Description	Delay
Address Decoder	Chooses destination ports based on header address	3.3 ns
Routing Decision	Arbitrates among requests of address decoder result	2.1 ns
Packet Update	Controls switch operation and updates header packet based on decision	2.6 ns
Crossbar Switch	Time for data to go through crossbar	1.1 ns
Virtual Channel	Multiplexes virtual channels among physical channel	1.2 ns
Total		10.3 ns

Figure 6.3 Components of Path Setup Delay

The second component of the intranode delay is termed the *data through* delay. The *data through* delay comes from the overhead of managing flow control on a transfer. For the Illinois baseline router, the delay time consists of the components illustrated in Figure 6.4. To allow for data synchronization and flow control management between nodes, it

Delay Component	Description	Delay
Internal Flow Control	Time for internal flow control decisions	2.2 ns
Crossbar Switch	Time for flow control data to propagate through node	1.0 ns
Virtual Channel	Time for negotiation of virtual channel	2.5 ns
Total		5.7 ns

Figure 6.4 Components of Data Through Delay

takes two *data through* delays for a message to be transferred. For the baseline Illinois router, two *data through* delays is greater than the *setup path* delay so the intranode delay is 5.7*2 = 11.4 ns.

The components of the internode delay are more simple. They are described in Figure 6.5. Basically, data has to get through an output driver, and input receiver, and then is latched. Although this delay is a non-trivial 4.9 ns, the delay is far less than the 11.4 ns required by the intranode delay. The Illinois router cycle is dominated by the intranode delay, even though it is the most simple of the adaptive routers they examined. If more degrees of freedom are allowed in the adaptive router, the intranode delay becomes much worse.

The Illinois paper goes on to examine a deterministic router based on the same assumptions as their adaptive router. For the deterministic router, the *path setup* delay becomes 5.7 ns while the *data through* delay becomes 3.0 ns. The internode delay remains fixed at 4.9 ns. The savings in intranode delay come from the absence of any virtual channel controller and the more simple routing decision that is made. However, two internode

Delay Component	Description	Delay
Output Buffer	Time for data to get off chip	2.5 ns
Input Buffer	Time for data to get on chip	.6 ns
Input Latch	Setup time to latch data	.8 ns
Synchronizer	Time charged for differences between system clocks	1.0 ns
Total		4.9 ns

Figure 6.5 Components of Internode Delay

transfer times are required per message word, so the cycle time is 4.9*2 = 9.8, only a small savings over the simple adaptive routing case.

The NuMesh CFSM can also be modeled based on these assumptions. By far, the biggest savings comes from the fact that only a single internode transfer is required per word. The intranode delay for the NuMesh router consists of only the *path setup* delay, since most of *data through* delay occurs in parallel with the *path setup* delay, and the rest occurs on a later stage of the pipeline. Compared to the *path setup* delay of the adaptive router, only the crossbar switch delay and some amount of the routing decision delay is charged to the NuMesh CFSM. Even if the routing decision delay is assumed to be equal to that of the adaptive case (a doubtful proposition, since no data dependent decisions need be made), the total intranode delay would be 1.1 + 2.1 = 3.2ns. Since the *path setup* delay in the NuMesh design occurs in a different pipeline stage than the internode transfer, the NuMesh cycle time will be the worse of the *path setup* delay and the internode delay. That gives the CFSM a clock period of 4.9ns. This is 50% of the clock period of the deterministic router, and 43% of the simple adaptive router. The Illinois router was implemented in a .8 micron CMOS gate array library from Mitsubishi Corporation.

Since the NuMesh cycle time is determined by an internode transfer, one could imagine using fancier signaling techniques or even pipelining the internode transfer itself to allow for an even faster clock cycle. Pipelining the internode transfer would add an additional constraint involving the number of required bubbles between operation of the same stream, but would allow for very high data throughput. Since no data dependent decisions need ever be made in the NuMesh design, the cycle time can be continually decreased as technology improves. A direct comparison of the three routers is included in Figure 6.6.

	Intranode Delay	Internode Delay	Total Cycle Time
Deterministic Router	5.7 ns	4.9 ns	9.8 ns
Adaptive Router	11.4 ns	4.9 ns	11.4 ns
NuMesh Router	<=3.1 ns	4.9 ns	4.9 ns

Figure 6.6 Cycle Time Comparison of Routers

6.5 Network Congestion

A second goal of this thesis was to provide a communication routing chip that can efficiently support scheduled communication. In order for the architecture to be considered useful, it must be shown that scheduled communication operating on the NuMesh system can outperform dynamic communication. The previous section showed that the cycle time of the network can be improved by fifty percent or more. This section will show that scheduled communication can reduce congestion in the network enough to provide significantly improved application performance.

Chris Metcalf[Metc97], will compare scheduled communication to a variety of dynamic routing schemes. In some of his early analysis, he takes several well-known application kernels and compares their performance for both off-line scheduled routing on the Numesh, and on-line dynamic communication methods. For the rest of this section, it should be noted that the cycle time for all routers is assumed to be the same. The added benefit of the increased potential frequency of the NuMesh CFSM is not taken into account. Three application kernels that Metcalf studied are parallel prefix, transpose, and prefix.

A parallel prefix algorithm works as follows. Assume there exists some operation $\otimes$. A group of nodes is labeled from n_0 - n_N. Each node n_i holds the value $n_0 \otimes \ldots \otimes n_i$. The operator can be any associative function such as ADD, MIN, or OR. Parallel machines use parallel prefix for a variety of algorithms such as graph traversal, traveling salesman, and various min/max problems. Since the communications for a parallel prefix are known at compile time, they can be completely scheduled. The algorithm does require several phases of communications that must be loaded in during operation. Metcalf's comparison of different communication schemes for the parallel prefix algorithm is included in Figure 6.7.

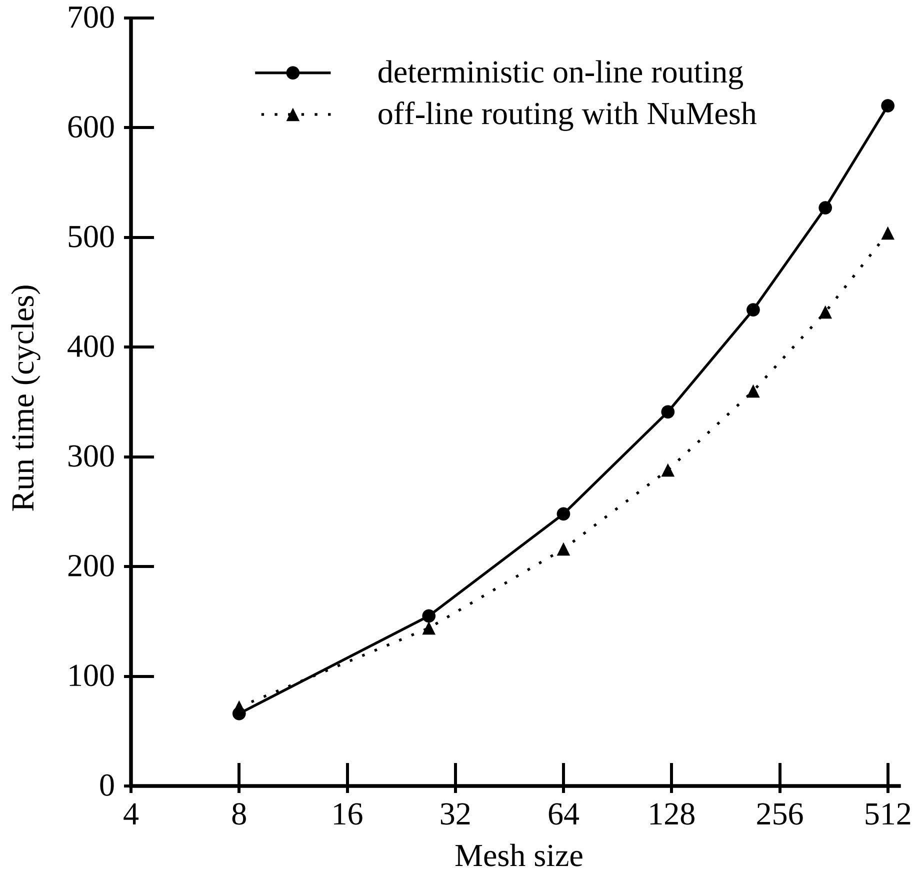


Figure 6.7 Comparison of Routing Schemes for Parallel Prefix

The second application Metcalf looked at was the transpose benchmark. In a three-dimensional array, every node sends data to another node based on its address. Messages are transferred from (x,y,z) to (z,y,x). Two Kilobytes of data are transferred from each node. For this application, both deterministic and adaptive schemes are compared to the NuMesh scheduled routing. The results are shown in Figure 6.8.

The final application kernel is a bit reverse. Again, all nodes are sending data to other nodes based on their address. In this case, every node is assigned a binary address of $n_o n_1 ... n_N$. A two Kilobyte message is sent to the node with address $n_N n_{k-1} ... n_0$. Again, both deterministic and dynamic routing schemes are compared to NuMesh scheduled communication. Results are shown in Figure 6.9.

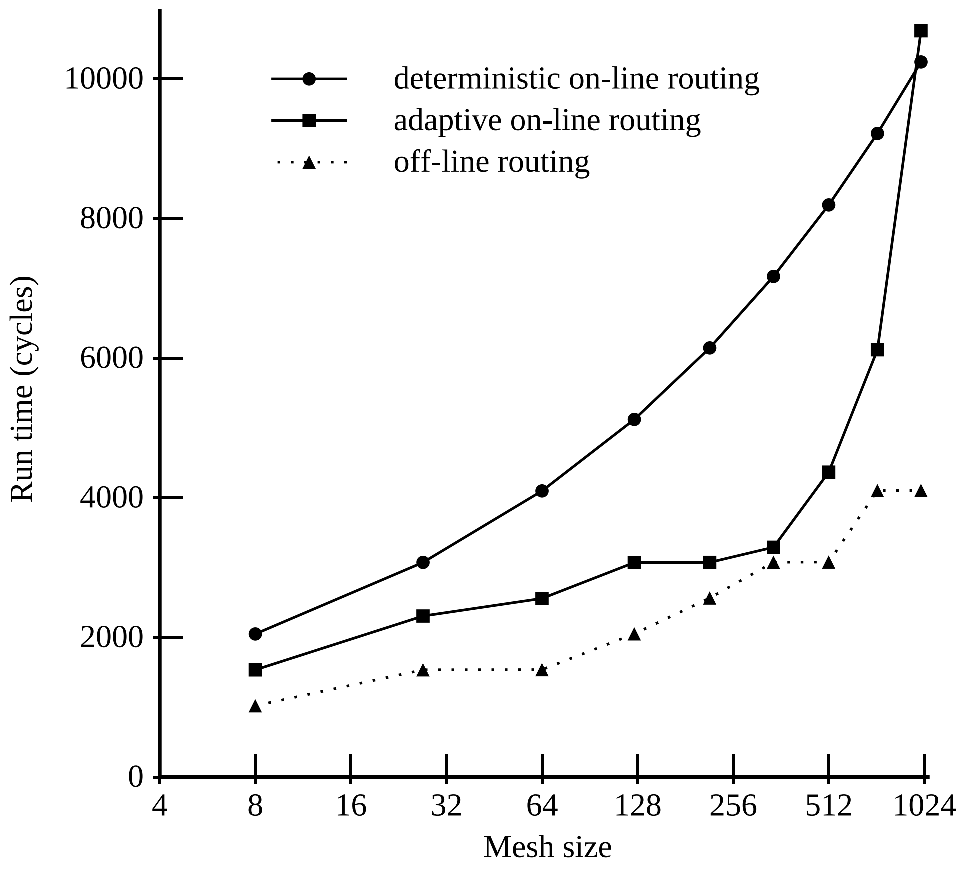


Figure 6.8 Comparison of Routing Schemes for Transpose

For all three applications, the scheduled communication model outperforms both the deterministic and adaptive schemes. The deterministic schemes suffer because nodes attempt to force too much data over selected links while free nearby links remain unused. The adaptive scheme suffers because bottlenecks are generated by some of the misroutes resulting in even longer delays than the simple deterministic router. If the frequency of the communication nodes was taken into account, the NuMesh scheduled communication results would fare significantly better, and the adaptive scheme would fall even further behind the deterministic scheme.

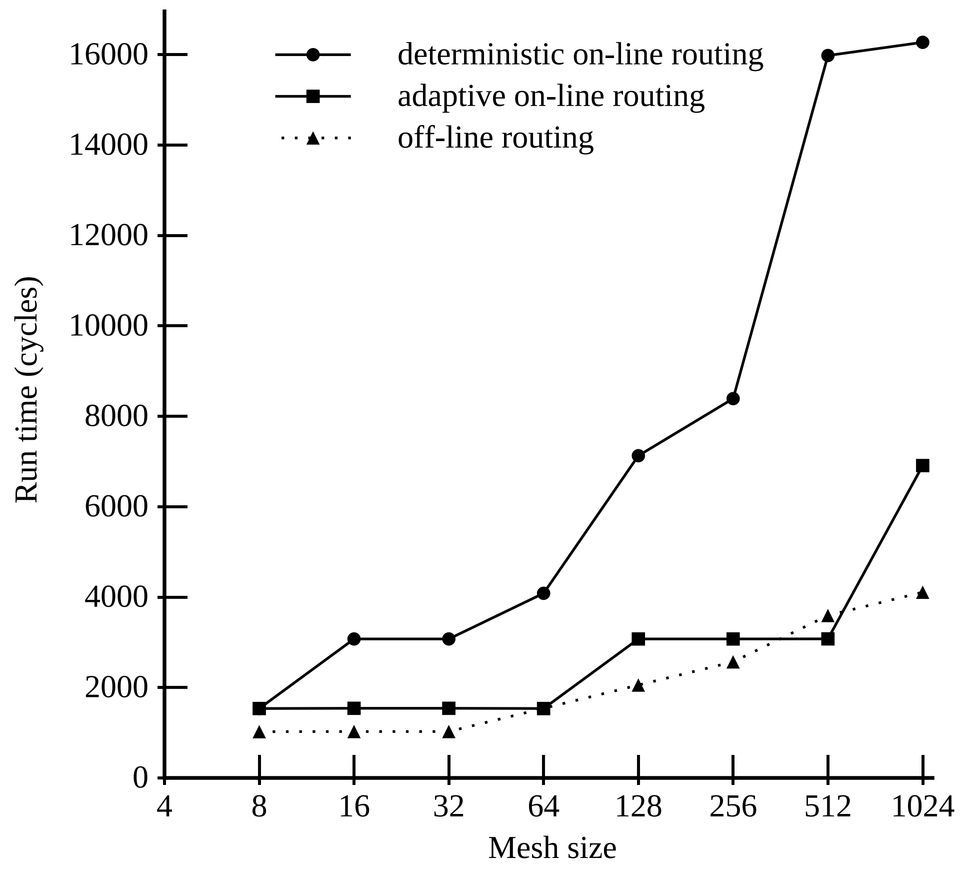


Figure 6.9 Comparison of Routing Schemes for Bit Reverse

6.6 NuMesh Environment

The CFSM forms just one part of the NuMesh system. The goal of the NuMesh project is to provide a hardware and a software solution for scheduled communication. This thesis has discussed a hardware architecture that is optimized for scheduled communication.

These chips were designed and received and will be incorporated into a multi-node system. This section will discuss the software environment that can take advantage of the implemented hardware and provide a framework in which programmers can write and execute parallel applications.

6.6.1 COP

The Communication Operators Language (COP) is proposed as part of Chris Metcalf's PhD thesis as a language for implementing scheduled communication. COP allows communication to be expressed at a high level and can include complex router-level semantics that might be difficult to specify on a per node basis. Operators for broadcasting messages, setting up communication streams between nodes, or even specifying various permutation patterns are included in a library. A programmer can simply use high level abstractions when writing applications, and the COP compiler can capture the parallelism needed at the node level.

The COP compiler can even decide when some decisions need to be handled dynamically. If multiple phases of communication are needed, the COP compiler can set up correct barriers to allow phases of scheduled communication on the NuMesh. If true dynamic routing is needed, the COP compiler might lay down many paths connecting all nodes at some low bandwidth and allow the computation units to insert messages as they see fit. Ideally, scheduled communication streams will be generated from the high level language, since the Numesh system is geared toward handling these cases, but the goal of the COP compiler is to be able to handle any parallel application.

6.6.2 Bullfrog

The Bullfrog environment was created as part of Mike Connel's Master's thesis [Conn96]. It describes a high level approach to developing NuMesh applications. A programmer can link together function blocks with data streams using either text or a graphical interface. A library of function blocks are described that provide basic signal processing and computation operations. The user can either link these blocks together, or create new application blocks that can be added to the library. An example of a trivial system is shown in Figure 6.10. A library module for calculating the square of a number is defined. A user can connect two of these blocks to form an application or to form another library module that raises a number to the fourth power.

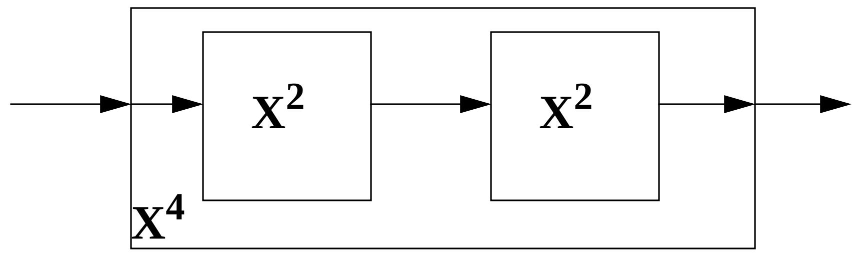


Figure 6.10 A Simple Bullfrog Example

Complicated DSP, graphics, or scientific applications can be built out of the various library modules allowing a user to ignore the details of the scheduled communication. The output of the Bullfrog system is a set of primitive function block instances along with any run-time parameters that they require, and a set of stream connections that can later be mapped to the communication hardware.

6.6.3 Lilypad

The Lilypad system was developed by Greg Spurrier for his Master's thesis. While the Bullfrog Block Diagram approach to programming is very attractive to the user for its ease of use and flexibility, it suffers from two limitations. First, it assumes a one-to-one mapping of a computation module to a processing node. This provides a very serious constraint in that a complex system might quite easily consist of more primitive blocks than there are nodes in the system. Secondly, primitive blocks may have a set of constraints that must be followed when they are connected. For instance, dataflow-type blocks might require all inputs to be valid before firing, or might be able to begin computation when it receives any of its multiple inputs. The Bullfrog system had no special mechanisms present for combining blocks. While this was attractive for simple applications, more complicated programs require a higher level of sophistication.

The Lilypad system was developed to automatically generate control procedures that allow an arbitrary number of primitive blocks to reside on a single NuMesh node, thereby eliminating the one-to-one constraint imposed by Bullfrog. In addition a set of conventions were defined that specify the state of input and output streams required before a block may be invoked. These conventions turn into synchronous barriers in the scheduled communication streams that are eventually extracted from the user block diagram.

6.6.4 Tadpole

The Tadpole compiler handles the job of turning a graph of communication streams into scheduled communication that can run on the NuMesh hardware [Lopr97]. Both the COP language and the Bullfrog/Lilypad software environments cause a directed graph of virtual streams to be extracted from the user application. This graph contains communication requirements between all nodes as well as bandwidth requests for each communication. The communication needs may change over time.

The Tadpole streams compiler works by first laying down all requested communication paths according to the shortest path of communication. For the initial attempt, many links will go unused while some links will be assigned a bandwidth greater than that which they are capable of carrying. Through a simulated annealing approach, the traffic gets thinned out across available links until all communication bandwidth requests are eventually met. Some messages may get misrouted and shortest path conventions may no longer be followed, but every communication path will get some bandwidth assigned during which its communication links are guaranteed to be free. If the scheduler decides that the bandwidth constraints can not be met, allocated bandwidth among all messages gets equally reduced until a valid schedule is produced.

Schedules are generated for each of the two pipelines on every NuMesh node, and complex operation such as forks and joins are also supported. Multiple communication phases are supported by scheduling each phase of communication separately, and then providing glue to allow the phases to switch during operation. The COP compiler can analyze changing communication phases and provide Tadpole with directives for managing multiple generated schedules. Once Tadpole generates its schedule, the code can be downloaded to the NuMesh CFSMs for operation.

Tadpole provides the final pieced of the software hierarchy that allows a user to design applications in a very high-level language or graphical environment that avoids the details of the NuMesh architecture.

6.7 Future Work

Now that hardware and software environments exist for creating parallel applications, the first order of business is to implement a wide range of parallel applications. In order to run larger applications, a multi-node system of thirty-two nodes will be implemented. This will require strict attention to a variety of system issues including clock synchrony, power distribution, cooling, physical support, and debugging support.

Ideally, the NuMesh system could be connected to a host that resides on the internet and allows programmers to remotely execute applications. A variety of user support programs will have to be written to support this idea.

The number of transistors allowed on a chip is growing at phenomenal rates, with a billion transistor chip being predicted in a few years. One could imagine a large number of small microprocessors or DSP cores being implemented on a single chip. The communication between these processors could occur through special registers set up at compile time in a similar fashion to the NuMesh system. Bandwidth could be assigned between registers of these processors in exactly the same fashion as the communication is currently scheduled in the NuMesh system. By allowing communication between integrated processors through high speed register files, very fine grain parallelism could be exploited, even on a per instruction basis. Since the pin limitations of the current NuMesh system would be eliminated in favor of virtually an unlimited number of short metal lines between processors, the system could handle large amounts of high speed communication. This idea could prove to be one of the few manageable ways to organize large numbers of processors on a single chip that provides truly fine-grained parallelism.

The NuMesh is a logical system to implement compute intensive multi-media applications. One proposed idea is for a graphics processing unit to be designed that could take the place of the SPARC processor in the current design. Each graphics unit could perform one part of complex multi-media operations such as MPEG encoding or 3-D graphics manipulation. Since most multi-media applications involve very regular communications, the NuMesh is a natural choice for implementation.

To improve the CFSM, a more aggressive circuit implementation could drastically reduce the cycle time of the network. Although the local graphics bus speed currently sets an upper bound on network performance, an alternative or even custom bus could be implemented to improve system performance. Much work has gone into fast signaling between chips, and this technique would serve to greatly improve the critical path of the NuMesh CFSM.

References

[Agar91] A. Agarwal. “Limits on Interconnection Network Performance”, *IEEE Transactions on Parallel and Distributed Systems*. October 1991.

[Aoya93] K. Aoyama and A. Chien. “The Cost of Adaptivity and Virtual Lanes in a Wormhole Router”, *Journal of VLSI Design.* Vol. 2, No. 4, pages 315-333, May 1993.

[Beet85] J. Beetem, M. Denneau, and D. Weingarten. “The GF11 Supercomputer”, *Proceedings of the 12th International Symposium on Computer Architecture*. June 1985.

[Berm87] F. Berman and L. Snyder. “On Mapping Parallel Algorithms Into Parallel Architectures”, *Journal of Parallel and Distributed Computing.* Vol. 4, No. 5, pages 439-458, October 1987.

[Bian87] R. Bianchini and J.P. Shen. “Interprocessor Traffic Scheduling Algorithm for Multiple-processor Networks”, *IEEE Transactions on Computers*, C-36 (4): 396-409, April 1987.

[Bier90] J. Bier et al. “Gabriel: A Design Environment for DSP”, *IEEE Micro.* Vol. 10, No. 5, pages 28-45, October 1990.

[Bour94] Y. Boura and C. Das. “Efficient Fully Adaptive Wormhole Routing in N-Dimensional Meshes”, 1*4th International Conference on Distributed Computing Systems.* June, 1992.

[Bork88] S. Borkar et al. “iWarp: An Integrated Solution to High-speed Parallel Computing”, In Proceedings of *Supercomputing ‘88*, pp. 330-339, November 1988.

[Bork90] S. Borkar et al. “Supporting Systolic and Memory Communication in iWarp”, *Proceedings of the 17th Annual International Symposium on Computer Architecture.* pages 70-81, June 1990.

[Chie92] A. Chien and J. Kim. “Planar-Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors”, *Journal of the Association fpr Computing Machinery*, March 1992.

[Conn96] M. Connell. “Bullfrog: An Extensible, Modular Toolkit for the Construction of NuMesh Applications”, *Master’s Thesis.* MIT, February 1996.

[Dall91] W. Dally and H. Aoki. “Deadlock-free adaptive routing in multicomputer networks using virtual channels”, *IEEE Transactions on Parallel and Distributed Systems, 2 (3): 176-183,* November, 1991.

[Dall87] W. Dally and C.L. Seitz. “Deadlock-free message routing in multiprocessors”, *IEEE Transactions on Computers*, C-36 (5) :547-53., May 1987.

[Dall90] W. Dally. “Virtual-Channel Flow Control”, *IEEE Transactions on Parallel and Distributed Systems.* Vol. 3, No. 2, pages 194-205, December 1990.

[Dall94] W. Dally et al. “The Reliable Router: A Reliable and High-Performance Communication Substrate for Parallel Computers”, *Proceedings of the First International Parallel Computer Routing and Communication Workshop*, May 1994.

[Glas92] C. Glass and L. Ni. “The Turn Model for Adaptive Routing”, *International Symposium on Computer Architecture*, May, 1992.

[Gros92] T. Gross et al. “The impact of communication style on machine resource usage for the iWarp parallel processor”, *Carnegie-Mellon Report CMU-CS=92-215*, November 1992.

[Gust92] D. Gustavson. “The Scalable Coherent Interface and Related Standards Projects”, *IEEE Micro.* June 1992.

[Hinr95] S. Hinrichs. “A Compile Time Model for Composing Parallel Programs”, *IEEE Parallel and Distributed Technology.* May 1995.

[Hono91] F. Honore. “Redesign of a Prototype NuMesh Module”, Master’s Thesis, Massachusetts Institute of Technology, June 1991.

[Kapl88] S. Kaplan and G. Kaiser. “Garp: Graph Abstractions for Concurrent Programming”, *European Symposium on Programming.* Vol. 300, pages 191-205, March 1988.

[Karj90] M. Karjalainen. “DSP Software Integration by Object-Oriented Programming: A Case Study of QuickSig”, *IEEE ASSP Magazine.* Vol. 7, No. 2, pages 21-31, April 1990.

[Lee90] E. Lee and J. Bier. “Architectures for Statically Scheduled Dataflow”, J*ournal of Parallel and Distributed Computing*. Vol. 10, pages 333-348, June 1990.

[Leig89] F.T. Leighton, F. Makedon, and I. Tollis. “A 2n-2 Step Algorithm for Routing in an NxN Array With Constant Size Queues”, *Proceedings 1989 ACM Symposium on Parallel Algorithms and Architectures.* pages 328-335, June 1989.

[Leig92] F.T. Leighton. “Introduction to Parallel Algorithms and Architectures - Arrays, Trees, Hypercubes,” *Morgan Kaufman Publishers*. 1992.

[Leig93] F.T. Leighton. “Theory of Parallel and VLSI Computation”, *Class Notes for 6.848.* September 1993.

[Lind91] D. Linder and J. Harden. “An Adaptive and Fault-Tolerant Wormhole Routing Strategy for K-ary n-cubes”,*IEEE Transactions on Computers*, Vol. C-40, pp. 178-186, Jan 1991.

[Lo90] V. Lo et al. “LaRCS: A Language for Describing Parallel Computations”, *IEEE Transactions on Parallel and Distributed System,* June 1990.

[Lopr97] P. LoPresti. “Tadpole: An Off-line Router for the NuMesh System”, *Master’s Thesis.* MIT, Jan 1997.

[Mage89] J. Magee, J. Kramer, and M. Sloman. “Constructing Distributed Systems in CONIC”, *IEEE Transactions on Software Engineering,* Vol. 15, No. 6, pages 663-675, June 1989.

[Magg95] B. Maggs. “A Critical Look at Three of Parallel Computing’s Maxims”, *CMU Technical Memo.* June 1995.

[Metc95] C. Metcalf. “The Numesh simulator, nsim”, NuMesh Group, MIT LCS January 1994.

[Metc96] C Metcalf. “A Comparative Evaluation of Online and Offline Routing”, *PhD Thesis in Progress*. MIT, 1997.

[OHal91] D. O’Hallaron. “The Assign Parallel Program Generator”, *Proceedings of the 5th Distributed Memory Conference,* 1991.

[Peza93] J Pezaris. “NuMesh CFSM Revision 2: A comparative Analysis of Three Candidate Architectures”, *Master’s Thesis*. MIT, February 1993.

[Prat93] G.Pratt, S Ward, J. Nguyen, and J Pezaris. “The Diamond Interconnect”, *MIT Technical Report.* December 1993.

[Prat94] G. Pratt, Gill and J. Nguyen. “Distributed Synchronous Clocking”, *Proceedings of the 16th Conference on Advanced Research in VLSI.* March, 1995.

[Prat95] G. Pratt, Gill and J. Nguyen. “Distributed Synchronous Clocking”, *IEEE Trabsactions on Parallel and Distributed Systems.* Vol. 6, No. 3, pages 314-328, March 1995.

[Raja92] S. Rajasekaran and R. Overholt. “Constant Queue Routing on a Mesh”, *Journal of Parallel and Distributed Computing*, Vol. 15:160-166, 1992.

[Sche91] I. Scherson and P. Corbett. “Communications Overhead and the Expected Speedup of Multidimensional Mesh-Connected Parallel Processors”, *Journal of Parallel and Distributed Processing*, Vol. 11:86-96, 1991.

[Shoe95] D. Shoemaker. “Hardware architecture and communication protocols optimized for static routing”, *MIT Workshop on Scalable Computing*, August 1995.

[Shoe96] D. Shoemaker, C. Metcalf and S. Ward. “NuMesh: An architecture for static routing”, In Proceedings of *International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 425-434, November 1995.

[Shoe96] D. Shoemaker et al. “NuMesh: An Architecture Optimized for Scheduled Communication”, *Journal of Supercomputing.* Vol. 10, pages 285-302, August 1996.

[Shuk91] S. Shukla and D. Agrawal. “Scheduling pipelined communications in distributed memory multiprocessors for real-time applications”, *18th Annual ACM SIGARCH Computer Architecture News*, 19 (3): 222-231, 1991.

[Sing93] J.P. Singh, W.D. Weber, and A. Gupta. “SPLASH: Stanford parallel applications for shared memory”, *Computer Architecture News*, 20 (1): 5-44, 1993.

[Spur96] G. Spurrier. “Supporting Multiple Function Blocks on a Single NuMesh Node”, *Master’s Thesis.* MIT, May 1996.

[Subl93] J. Subhlok et al. “Exploiting Task and Data Parallelism on a Multicomputer”, *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pages 13-22, May 1993.

[Ward93] S. Ward, et al. “The NuMesh: A modular, scalable, communications substrate”, In *International Conference on Supercomputing*, pp. 123-132, July 1993.

[Webb92] J. Webb. “Steps Toward Architecture-Independent Image Processing”. *Computer,* Vol. 25, No. 2, pages 21-31, February 1992.

[Ziss86] M. Zizzman, G. O’Leqry, D. Johnson. “A Block Diagram Compiler for a Digital Signal Processing MIMD Computer”, *Proceedings IEEE ASSP Digital Signal Processing Workshop.* October 1986.

Appendix A

Verilog Code

This appendix includes the verilog code used to synthesize the initial design of the chip. The synthesized netlist was later hand-tuned to improve the timing of critical paths. The verilog code is broken up into separate files.

• sched.v - This files describes the logic required by the scheduler.

• pipeline.v - This file includes the logic for an entire pipeline, except for the scheduler. The current implementation uses two such pipeline modules.

• datapath.v - Thhis file included drivers, receivers, and processor interface logic used in the CFSM.

• CFSM.v - This file connects the previous file together. In additional, system level logic bits are implemented here.

• modules.v - This file contains all of the base level modules used in the system. They form the building blocks from which more complicated designs are created.

“sched.v”

```
/* Stage one of the CFSM pipeline */

module schedule_ram(stage4_sched_addr, stage4_sched_web,
          stage4_sched_data,
          stage4_sched_ptr_web,
          ram_out_index1, ram_out_index2, clk, bootmode);

input [6:0] stage4_sched_addr;
input stage4_sched_web, clk, bootmode;
input stage4_sched_ptr_web;
input [16:0] stage4_sched_data;
output [4:0] ram_out_index1, ram_out_index2;

wire [6:0] ramadd, finramadd;
wire [6:0] ram_out_pointer, finram_out_ptr;
wire [16:0] stage4_sched_data;
wire [6:0] stage4_sched_addr;
wire stage4_sched_web;
wire hibit0, hibit1, lobit0, lobit1, lobit2, lobit3, lobit4, lobit5,
   lobit6, lobit7, lobit8, lobit9, lobit10, lobit11, lobit12, lobit13,
   lobit14, lobit15, lobit16, lobit17;

/* Schedule RAM is 128 x 17.  Only one pipeline is allowed to write the
RAM at any given time.  This is a software convention. */

assign finramadd = (bootmode == 0) ? 7’b0000000:ramadd;

hilo lobt0 (.LO(lobit0), .HI(hibit0));
hilo lobt1 (.LO(lobit1), .HI(hibit1));
hilo lobt2 (.LO(lobit2));
hilo lobt3 (.LO(lobit3));
hilo lobt4 (.LO(lobit4));
hilo lobt5 (.LO(lobit5));
hilo lobt6 (.LO(lobit6));
hilo lobt7 (.LO(lobit7));
hilo lobt8 (.LO(lobit8));
hilo lobt9 (.LO(lobit9));
hilo lobt10 (.LO(lobit10));
hilo lobt11 (.LO(lobit11));
hilo lobt12 (.LO(lobit12));
hilo lobt13 (.LO(lobit13));
hilo lobt14 (.LO(lobit14));
hilo lobt15 (.LO(lobit15));
hilo lobt16 (.LO(lobit16));
hilo lobt17 (.LO(lobit17));
/*
sim_ram128x17 ram_dual_128_17_2c2_inst (stage4_sched_addr,
finram_out_ptr, stage4_sched_data[16:0], {ram_out_pointer,
ram_out_index1, ram_out_index2}, stage4_sched_web, clk);
*/
ram_dual_128_17_2c2 ram_dual_128_17_2c2_inst
    (.AA0(stage4_sched_addr[0]), .AA1(stage4_sched_addr[1]),
     .AA2(stage4_sched_addr[2]), .AA3(stage4_sched_addr[3]),
     .AA4(stage4_sched_addr[4]), .AA5(stage4_sched_addr[5]),
      .AA6(stage4_sched_addr[6]),
      .AB0(finram_out_ptr[0]), .AB1(finram_out_ptr[1]),
      .AB2(finram_out_ptr[2]), .AB3(finram_out_ptr[3]),
      .AB4(finram_out_ptr[4]), .AB5(finram_out_ptr[5]),
      .AB6(finram_out_ptr[6]),
      .DA0(stage4_sched_data[0]), .DA1(stage4_sched_data[1]),
      .DA2(stage4_sched_data[2]), .DA3(stage4_sched_data[3]),
      .DA4(stage4_sched_data[4]), .DA5(stage4_sched_data[5]),
      .DA6(stage4_sched_data[6]), .DA7(stage4_sched_data[7]),
      .DA8(stage4_sched_data[8]), .DA9(stage4_sched_data[9]),
      .DA10(stage4_sched_data[10]), .DA11(stage4_sched_data[11]),
      .DA12(stage4_sched_data[12]), .DA13(stage4_sched_data[13]),
      .DA14(stage4_sched_data[14]), .DA15(stage4_sched_data[15]),
      .DA16(stage4_sched_data[16]),
      .DB0(lobit0), .DB1(lobit1),
      .DB2(lobit2), .DB3(lobit3),
      .DB4(lobit4), .DB5(lobit5),
      .DB6(lobit6), .DB7(lobit7),
      .DB8(lobit8), .DB9(lobit9),
      .DB10(lobit10), .DB11(lobit11),
      .DB12(lobit12), .DB13(lobit13),
      .DB14(lobit14), .DB15(lobit15),
      .DB16(lobit16),
     .QB0(ram_out_index2[0]), .QB1(ram_out_index2[1]),
      .QB2(ram_out_index2[2]), .QB3(ram_out_index2[3]),
      .QB4(ram_out_index2[4]),
      .QB5(ram_out_index1[0]), .QB6(ram_out_index1[1]),
      .QB7(ram_out_index1[2]), .QB8(ram_out_index1[3]),
      .QB9(ram_out_index1[4]),
      .QB10(ram_out_pointer[0]), .QB11(ram_out_pointer[1]),
      .QB12(ram_out_pointer[2]), .QB13(ram_out_pointer[3]),
      .QB14(ram_out_pointer[4]), .QB15(ram_out_pointer[5]),
      .QB16(ram_out_pointer[6]),
      .CKA(clk), .WEA(stage4_sched_web), .SELA(hibit0),
     .CKB(clk), .WEB(lobit17), .SELB(hibit1));

/* pipeline 1 gets the bottom five bits, and pipeline 2 gets the next
five bits */

/* the address field will get the high seven bits of the
schedule ram */

assign finram_out_ptr = (bootmode == 0) ? 7'b0000000:
             (stage4_sched_ptr_web == 0) ?
     stage4_sched_data[6:0]:ram_out_pointer;

/* myreg #(7) nxt(finram_out_ptr, ramadd, clk); */
endmodule
```

“pipeline.v”

```
module pipeline0(stage1_index, dp_src, dp_preg, dp_accept,
        out1_sched_web,
        outdp_src_ports,
        outdp_src_preg_addr, outdp_accept, outdp_write_ports,
        outdp_write_data, outdp_preg_waddr, outdp_preg_web,
        outdp_preg_read_clr, out1_sched_ptr_web,
        clk, my_boot, other_boot, clear_val, stage4_toggle,
        stage4_clr_val, stage4_jtag_web,
        stage4_clk_sys_web, reset, jtag_in, stage4_source_reg);

input clk, dp_accept, my_boot, other_boot, clear_val, jtag_in;
input [3:0] reset;
input [4:0] stage1_index;
input [32:0] dp_src, dp_preg;

output [3:0] outdp_src_ports, outdp_write_ports;
output [3:0] outdp_src_preg_addr, outdp_preg_waddr;
output [32:0] outdp_write_data, stage4_source_reg;
output out1_sched_web, out1_sched_ptr_web, outdp_accept;
output outdp_preg_web, outdp_preg_read_clr, stage4_clr_val;
output stage4_toggle;
output stage4_clk_sys_web, stage4_jtag_web;

wire [13:0] newiw, stage3_iw;
wire [4:0] stage2_index,stage3_index, stage4_breg_waddr;
wire [4:0] finstage3_index, stage2_breg_read_addr;
wire [4:0] stage2_breg_valid_read, stage3_breg_valid_read;
wire [4:0] stage3_breg_read_addr, stage3_breg_waddr;
wire outdp_preg_read_en;
wire rsuccess, wsuccess, stage4_wsuccess;
wire  stage2_valid, finstage2_valid;
wire stage4_breg_web, stage2_valid_pre;
wire [32:0] stage3_source_reg;
wire [32:0] stage3_breg_out, stage3_breg_out_pre;
wire [3:0] stage2_src_ports, stage2_src_preg_addr;
wire [3:0] stage2_write_ports, stage3_write_ports;
wire [5:0] stage4_write_addr;
wire stage3_preg_write_en;
wire stage4_iw_web, stage4_valid_bit;
wire stage3_other_valid, stage3_other_valid_pre, stage3_valid;
wire pipe_read_write, pre_outdp_accept, pre_stage3_source_val;
wire stage3_mem_addr, stage4_mem_addr;
wire stage3_source_reg_pre;
wire dp_accept_final, stage3_buf_sel, bootmode, jtag_source;
wire [13:0] reset_word, finaliw;
wire stage2_preg_read_en, stage2_preg_read_clr, stage2_preg_write_en;
wire pre_stage2_accept, stage2_pipe_read_write;
wire stage2_bmove, stage2_cmove, stage2_fmove;
wire stage3_bmove, stage3_cmove, stage3_fmove;
wire stage4_cmove, stage4_fmove, port_write_detect;
wire stage2_accept, stage3_write_detect, stage2_buf_sel, stage2_mem_addr;
wire stage3_outdp_preg_web, stage4_preg_write_en;
wire [4:0] stage1_index_fin;
wire wsuca, wsucb, stage_wsuca, stage4_wsucb, stage2_breg_read_addr_pre;

wire hibit0, hibit1, lobit0, lobit1, lobit2, lobit3, lobit4, lobit5,
    lobit6, lobit7, lobit8, lobit9, lobit10, lobit11, lobit12, lobit13,
    lobit14;

wire hibitb0, hibitb1, lobitb0, lobitb1, lobitb2, lobitb3, lobitb4, lobitb5,
    lobitb6, lobitb7, lobitb8, lobitb9, lobitb10, lobitb11, lobitb12,
    lobitb13, lobitb14, lobitb15, lobitb16, lobitb17, lobitb18,
    lobitb19, lobitb20, lobitb21, lobitb22, lobitb23, lobitb24,
    lobitb25, lobitb26, lobitb27, lobitb28, lobitb29, lobitb30,
    lobitb31, lobitb32;

/* start of stage 2 */
/* instruction RAM is 32 x 16 */
/* input order: address a and b interleaved low to high, data, read
output, write enable, clk */

assign stage1_index_fin = (bootmode == 0) ? 5’b00000:stage1_index;

myreg #(5) index1(stage1_index_fin, stage2_index, clk);

assign bootmode = my_boot & other_boot;

/* boot word is one of four ports or proc reg 0 */
assign reset_word = (reset[0] == 1) ? 16’h0006:
         (reset[1] == 1) ? 16’h0046:
         (reset[2] == 1) ? 16’h0086:
         (reset[3] == 1) ? 16’h00C6:16’h034D;

/* mux to select which instruction will be used, boot or regular */
assign finaliw = (my_boot == 0) ? reset_word:
       (other_boot == 0) ? 14’h034D:newiw;

my_ram32x14 ram_dual_32_14_1c2_inst (stage4_source_reg[28:24],
stage2_index, stage4_source_reg[13:0], newiw, stage4_iw_web, clk);
/*
hilo lobt0 (.LO(lobit0), .HI(hibit0));
hilo lobt1 (.LO(lobit1), .HI(hibit1));
hilo lobt2 (.LO(lobit2));
hilo lobt3 (.LO(lobit3));
hilo lobt4 (.LO(lobit4));
hilo lobt5 (.LO(lobit5));
hilo lobt6 (.LO(lobit6));
hilo lobt7 (.LO(lobit7));
hilo lobt8 (.LO(lobit8));
hilo lobt9 (.LO(lobit9));
hilo lobt10 (.LO(lobit10));
hilo lobt11 (.LO(lobit11));
hilo lobt12 (.LO(lobit12));
hilo lobt13 (.LO(lobit13));
hilo lobt14 (.LO(lobit14));
```

```
ram_dual_32_14_1c2 ram_dual_32_14_1c2_inst
                (.AA0(stage4_source_reg[24]), .AA1(stage4_source_reg[25]),
                 .AA2(stage4_source_reg[26]), .AA3(stage4_source_reg[27]),
                 .AA4(stage4_source_reg[28]),
                 .AB0(stage1_index[0]), .AB1(stage1_index[1]),
               .AB2(stage1_index[2]), .AB3(stage1_index[3]),
               .AB4(stage1_index[4]),
               .DA0(stage4_source_reg[0]), .DA1(stage4_source_reg[1]),
                 .DA2(stage4_source_reg[2]), .DA3(stage4_source_reg[3]),
                 .DA4(stage4_source_reg[4]), .DA5(stage4_source_reg[5]),
                 .DA6(stage4_source_reg[6]), .DA7(stage4_source_reg[7]),
                 .DA8(stage4_source_reg[8]), .DA9(stage4_source_reg[9]),
                 .DA10(stage4_source_reg[10]), .DA11(stage4_source_reg[11]),
                 .DA12(stage4_source_reg[12]), .DA13(stage4_source_reg[13]),
                 .DB0(lobit0), .DB1(lobit1),
               .DB2(lobit2), .DB3(lobit3),
               .DB4(lobit4), .DB5(lobit5),
               .DB6(lobit6), .DB7(lobit7),
               .DB8(lobit8), .DB9(lobit9),
               .DB10(lobit10), .DB11(lobit11),
               .DB12(lobit12), .DB13(lobit13),
               .QB0(newiw[0]), .QB1(newiw[1]),
                 .QB2(newiw[2]), .QB3(newiw[3]),
                 .QB4(newiw[4]), .QB5(newiw[5]),
                 .QB6(newiw[6]), .QB7(newiw[7]),
                 .QB8(newiw[8]), .QB9(newiw[9]),
                 .QB10(newiw[10]), .QB11(newiw[11]),
                 .QB12(newiw[12]), .QB13(newiw[13]),
                 .CKA(clk), .SELA(hibit0), .WEA(stage4_iw_web),
               .CKB(clk), .SELB(hibit1), .WEB(lobit14));
*/
/* in stage 2 the RAM is read and the instruction is registered along
with the thread number.  Also, the valid bit of the b register is read. */
/* input order: radd1, radd2, wadd1, wdat1, wadd2, web1, web2, rout1,
rout2, clk, reset (second write always writes a zero) */

/* the buffer register that will be read will either be the register
number specified in the instruction or the thread number.  The buffer
will be the thread number unless the alternate buffer source is set,
and either: 1.  It is a bmove, 2. It is flow control and the thread
valid is not set, 3. It is a cmove and the rsuccess is set. */

assign stage2_breg_read_addr_pre =  ((finaliw[11] == 1’b1) &&
       ((finaliw[13:12] == 2’b00) ||
       ((finaliw[13:12] == 2’b01) &&
       (finstage2_valid == 1’b0)))) ? 1:0;

assign stage2_breg_read_addr = ((stage2_breg_read_addr_pre == 1) ||
       ((finaliw[11] == 1) &&
        (stage2_cmove == 1) && (rsuccess == 1)))
       ? finaliw[10:6]:stage2_index;
```

```
myreg #(5) bregr (stage2_breg_read_addr, stage3_breg_read_addr, clk);

RAM_4port_breg Breg_valid_ram(stage2_index,stage3_breg_valid_read,
            stage4_breg_waddr, stage4_valid_bit,
            stage3_breg_read_addr,
            stage4_breg_web,
            stage2_valid, stage3_other_valid,
            clk, clear_val);

/* pass the valid bit of the thread number to stage 3. */

assign finstage2_valid = (bootmode == 0) ? 0:stage2_valid;
myreg valid_reg(.D(finstage2_valid), .Q(stage3_valid), .CLK(clk));

/* pass instruction word to the third stage */
myreg #(14) outreg(finaliw, stage3_iw, clk);

/* pass the thread number to the third stage */
myreg #(5) out2reg(stage2_index, stage3_index, clk);

/* start of stage 3 */

/* decode of sources */
/* First the read port signals are decoded and sent to the ports */

assign stage2_src_ports[0] = (finaliw[11:6] == 6'b000000) ? 1:0;
assign stage2_src_ports[1] = (finaliw[11:6] == 6'b000001) ? 1:0;
assign stage2_src_ports[2] = (finaliw[11:6] == 6'b000010) ? 1:0;
assign stage2_src_ports[3] = (finaliw[11:6] == 6'b000011) ? 1:0;

assign outdp_src_ports = stage2_src_ports;

/* The processor registers can only be accessed for a read on a single
port.  Because of this, an instruction is not allowed to read and
write a processor register.  If a flow control write is needed, the
valid bit is read in during the third pipeline stage.  Two signals are
created to indicate preg read and write enables */
assign jtag_source = (stage3_iw[11:6] == 6'b000110) ? 1:0;

assign stage2_preg_read_en = (finaliw[11:10] == 2'b01) ? 0:1;
myreg prgread (stage2_preg_read_en, outdp_preg_read_en, clk);

assign stage2_preg_read_clr = ((stage2_preg_read_en == 0) &&
             (pre_stage2_accept == 1)) ? 0:1;
myreg prgclr (stage2_preg_read_clr, outdp_preg_read_clr, clk);

assign stage2_preg_write_en = (finaliw[5:4] == 2'b01) ? 0:1;
myreg prgwt (stage2_preg_write_en, stage3_preg_write_en, clk);
myreg prgwt2 (stage3_preg_write_en, stage4_preg_write_en, clk);

/* the preg address will be either the read address or the write
address.  Software conventions will not allow both. If doing a flow
control write, I need to read the valid bit now. */
```

```
assign stage2_src_preg_addr = (stage2_preg_read_en == 0) ?
          finaliw[9:6]:finaliw[3:0];
myreg #(4) pregaddr (stage2_src_preg_addr, outdp_src_preg_addr, clk);

/* the alternate register to be read for the valid bit of the buffer
register will be the buffer register of another register encoded in
the instruction.  I will read a destination for flow control purposes.
*/

assign stage2_breg_valid_read = (finaliw[5] == 1) ?
      finaliw[4:0]:finaliw[10:6];
myreg #(5) bvalrd (stage2_breg_valid_read, stage3_breg_valid_read, clk);

/* The valid bit of the buffer register will be either the thread's
valid bit or alternate buffer register's valid bit. */

assign stage3_breg_out[32] = (finstage3_index == stage3_breg_read_addr) ?
         stage3_valid:stage3_other_valid;

/* input order: write address interleaved with read address, data, ,
read output, web, clk */
/*
sim_ram32x32 ram_dual_32_32_1a2_inst (stage4_breg_waddr,
stage2_breg_read_addr, stage4_source_reg[31:0],
stage3_breg_out_pre[31:0], stage4_breg_web, clk);
*/

hilo lobtb0 (.LO(lobitb0), .HI(hibitb0));
hilo lobtb1 (.LO(lobitb1), .HI(hibitb1));
hilo lobtb2 (.LO(lobitb2));
hilo lobtb3 (.LO(lobitb3));
hilo lobtb4 (.LO(lobitb4));
hilo lobtb5 (.LO(lobitb5));
hilo lobtb6 (.LO(lobitb6));
hilo lobtb7 (.LO(lobitb7));
hilo lobtb8 (.LO(lobitb8));
hilo lobtb9 (.LO(lobitb9));
hilo lobtb10 (.LO(lobitb10));
hilo lobtb11 (.LO(lobitb11));
hilo lobtb12 (.LO(lobitb12));
hilo lobtb13 (.LO(lobitb13));
hilo lobtb14 (.LO(lobitb14));
hilo lobtb15 (.LO(lobitb15));
hilo lobtb16 (.LO(lobitb16));
hilo lobtb17 (.LO(lobitb17));
hilo lobtb18 (.LO(lobitb18));
hilo lobtb19 (.LO(lobitb19));
hilo lobtb20 (.LO(lobitb20));
hilo lobtb21 (.LO(lobitb21));
hilo lobtb22 (.LO(lobitb22));
hilo lobtb23 (.LO(lobitb23));
hilo lobtb24 (.LO(lobitb24));
hilo lobtb25 (.LO(lobitb25));
hilo lobtb26 (.LO(lobitb26));
hilo lobtb27 (.LO(lobitb27));
hilo lobtb28 (.LO(lobitb28));
hilo lobtb29 (.LO(lobitb29));
hilo lobtb30 (.LO(lobitb30));
hilo lobtb31 (.LO(lobitb31));
hilo lobtb32 (.LO(lobitb32));


ram_dual_32_32_1a2 ram_dual_32_32_1a2_inst
  (.AA0(stage4_breg_waddr[0]), .AA1(stage4_breg_waddr[1]),
   .AA2(stage4_breg_waddr[2]), .AA3(stage4_breg_waddr[3]),
   .AA4(stage4_breg_waddr[4]),
   .AB0(stage2_breg_read_addr[0]),
   .AB1(stage2_breg_read_addr[1]), .AB2(stage2_breg_read_addr[2]),
   .AB3(stage2_breg_read_addr[3]), .AB4(stage2_breg_read_addr[4]),
   .DA0(stage4_source_reg[0]), .DA1(stage4_source_reg[1]),
   .DA2(stage4_source_reg[2]), .DA3(stage4_source_reg[3]),
   .DA4(stage4_source_reg[4]), .DA5(stage4_source_reg[5]),
   .DA6(stage4_source_reg[6]), .DA7(stage4_source_reg[7]),
   .DA8(stage4_source_reg[8]), .DA9(stage4_source_reg[9]),
   .DA10(stage4_source_reg[10]), .DA11(stage4_source_reg[11]),
   .DA12(stage4_source_reg[12]), .DA13(stage4_source_reg[13]),
   .DA14(stage4_source_reg[14]), .DA15(stage4_source_reg[15]),
   .DA16(stage4_source_reg[16]), .DA17(stage4_source_reg[17]),
   .DA18(stage4_source_reg[18]), .DA19(stage4_source_reg[19]),
   .DA20(stage4_source_reg[20]), .DA21(stage4_source_reg[21]),
   .DA22(stage4_source_reg[22]), .DA23(stage4_source_reg[23]),
   .DA24(stage4_source_reg[24]), .DA25(stage4_source_reg[25]),
   .DA26(stage4_source_reg[26]), .DA27(stage4_source_reg[27]),
   .DA28(stage4_source_reg[28]), .DA29(stage4_source_reg[29]),
   .DA30(stage4_source_reg[30]), .DA31(stage4_source_reg[31]),
   .DB0(lobitb0), .DB1(lobitb1),
   .DB2(lobitb2), .DB3(lobitb3),
   .DB4(lobitb4), .DB5(lobitb5),
   .DB6(lobitb6), .DB7(lobitb7),
   .DB8(lobitb8), .DB9(lobitb9),
   .DB10(lobitb10), .DB11(lobitb11),
   .DB12(lobitb12), .DB13(lobitb13),
   .DB14(lobitb14), .DB15(lobitb15),
   .DB16(lobitb16), .DB17(lobitb17),
   .DB18(lobitb18), .DB19(lobitb19),
   .DB20(lobitb20), .DB21(lobitb21),
   .DB22(lobitb22), .DB23(lobitb23),
   .DB24(lobitb24), .DB25(lobitb25),
   .DB26(lobitb26), .DB27(lobitb27),
   .DB28(lobitb28), .DB29(lobitb29),
   .DB30(lobitb30), .DB31(lobitb31),
   .QB0(stage3_breg_out_pre[0]), .QB1(stage3_breg_out_pre[1]),
   .QB2(stage3_breg_out_pre[2]), .QB3(stage3_breg_out_pre[3]),
   .QB4(stage3_breg_out_pre[4]), .QB5(stage3_breg_out_pre[5]),
   .QB6(stage3_breg_out_pre[6]), .QB7(stage3_breg_out_pre[7]),
   .QB8(stage3_breg_out_pre[8]), .QB9(stage3_breg_out_pre[9]),
   .QB10(stage3_breg_out_pre[10]), .QB11(stage3_breg_out_pre[11]),
   .QB12(stage3_breg_out_pre[12]), .QB13(stage3_breg_out_pre[13]),
    .QB14(stage3_breg_out_pre[14]), .QB15(stage3_breg_out_pre[15]),
    .QB16(stage3_breg_out_pre[16]), .QB17(stage3_breg_out_pre[17]),
    .QB18(stage3_breg_out_pre[18]), .QB19(stage3_breg_out_pre[19]),
    .QB20(stage3_breg_out_pre[20]), .QB21(stage3_breg_out_pre[21]),
    .QB22(stage3_breg_out_pre[22]), .QB23(stage3_breg_out_pre[23]),
    .QB24(stage3_breg_out_pre[24]), .QB25(stage3_breg_out_pre[25]),
    .QB26(stage3_breg_out_pre[26]), .QB27(stage3_breg_out_pre[27]),
    .QB28(stage3_breg_out_pre[28]), .QB29(stage3_breg_out_pre[29]),
    .QB30(stage3_breg_out_pre[30]), .QB31(stage3_breg_out_pre[31]),
    .CKA(clk), .SELA(hibitb0), .WEA(stage4_breg_web),
    .CKB(clk), .SELB(hibitb1), .WEB(lobitb32));


/* Here is the bypass for the buffer registers.  Software conventions
must not allow the same thread to operate back to back or esle a race
condition will occur. */

assign stage3_breg_out[31:0] = ((stage4_breg_waddr == stage3_breg_read_addr)
        && (stage4_breg_web == 1'b1))
        ? stage4_source_reg[31:0]:stage3_breg_out_pre[31:0];

/* detect case where pipeline is reading and writing the same port
high active */

assign stage2_pipe_read_write = ((stage3_write_ports == stage2_src_ports)
       && (stage3_write_detect == 1)) ? 1:0;
myreg piperw (stage2_pipe_read_write, pipe_read_write, clk);

/* The accept bit is generated when a move doesn't use flow control,
when a move uses conditional flow control and the rsuccess bit is set,
or when regular flow control is used and the thread's buffer register
is empty opcode map: blind = 00, flow = 01, cmove = 10, cnmove = 11*/

/* I use a pre_outdp_accept in order to handle the case when I have a
same pipe read/write case */

assign stage2_bmove = (finaliw[13:12] == 2'b00) ? 1:0;
assign stage2_cmove = (finaliw[13:12] == 2'b10) ? 1:0;
assign stage2_fmove = (finaliw[13:12] == 2'b01) ? 1:0;

myreg bm (stage2_bmove, stage3_bmove, clk);
myreg cm (stage2_cmove, stage3_cmove, clk);
myreg fm (stage2_fmove, stage3_fmove, clk);

myreg cm2 (stage3_cmove, stage4_cmove, clk);
/*myreg fm2 (stage3_fmove, stage4_fmove, clk);*/

assign pre_stage2_accept = ((stage2_bmove == 1) ||
         ((stage2_cmove == 1) && (rsuccess == 1)) ||
         ((stage2_fmove == 1) && (finstage2_valid == 0)))
         ? 1:0;
myreg pacc (pre_stage2_accept, pre_outdp_accept, clk);

/* If I am reading and writing the same pipe, I will zero out the
```

```
accept bit because a fork operation is occurring.  Since I will not be
reading another port, I don't want to send out an accept to another
port by accident */

assign stage2_accept = (stage2_pipe_read_write == 1) ?
       0:pre_stage2_accept;
assign outdp_accept = stage2_accept;

/* the source register value will come from either the p registers,
the b registers, or the datapath.  A=00 or 01, B=10, C=11.  The valid
bit will not yet be finalized to handle the read/write same pipeline
case.  */

assign stage2_buf_sel = (finaliw[11] == 1) ? 0:pre_stage2_accept;
myreg bufreg (stage2_buf_sel, stage3_buf_sel, clk);

assign {stage3_source_reg_pre, stage3_source_reg[31:0]} = (jtag_source == 1) ?
           {32'h80000000, jtag_in}:
         (stage3_buf_sel == 0) ? stage3_breg_out:
         (outdp_preg_read_en == 0) ? dp_preg:dp_src;

/* A read fails if the accept bit is not set or if a flow control read
yields no valid word.  The pre_outdp_accept is used to ignore the
zeroing out affect of the pipe read/write case. */

assign rsuccess = ((pre_outdp_accept == 1'b0) || ((stage3_fmove == 1) &&
         (stage3_source_reg[32] == 0))) ? 0:1;

/* decode of destinations */
/* The decode of destinations is one hot encoded for prompt execution
in stage 4.  The four special locations are also decoded */

assign stage2_write_ports[0] = (finaliw[5:0] == 6'b000000) ? 1:0;
assign stage2_write_ports[1] = (finaliw[5:0] == 6'b000001) ? 1:0;
assign stage2_write_ports[2] = (finaliw[5:0] == 6'b000010) ? 1:0;
assign stage2_write_ports[3] = (finaliw[5:0] == 6'b000011) ? 1:0;

myreg #(4) wtpts (stage2_write_ports, stage3_write_ports, clk);
/* The mem address location corresponds to writing any of the various
memories. */

assign stage2_mem_addr = (finaliw[5:0] == 6'b000110) ? 0:1;
myreg memadr (stage2_mem_addr, stage3_mem_addr, clk);

/* Thread number is passed on to the next stage */
assign finstage3_index = (bootmode == 0) ? 5'b00000:stage3_index;

/* the undecoded destination field is passed to stage 4 */
myreg #(6) waddr(stage3_iw[5:0], stage4_write_addr, clk);

/* the one-hot encoded port control signals are passed to stage 4, and
immediately sent to the ports */

assign outdp_write_ports = stage3_write_ports;
```

```
/* The source word is passed to stage 4 */
myreg #(33) source_val_reg(stage3_source_reg, stage4_source_reg, clk);

/* The decoded memory location destination bit is passed to stage 4 */
myreg mem_add_reg(.D(stage3_mem_addr), .Q(stage4_mem_addr), .CLK(clk));

/* The valid bit from the processor register source word is passed to
stage 4.  If a processor register is a destination, the destination
location's valid bit will appear here. */

/* stage 4 */
/* invalidate accept bit for same pipe read/write if the read stage is
unable to accept a word. */
assign dp_accept_final = ((pipe_read_write == 1) &&
        (pre_outdp_accept == 0)) ? 0:dp_accept;

/* The various write enables of memory locations depend on both the
mem_addr bit being set and the top two bits in the source word. */
assign stage4_clr_val = ((stage4_source_reg[32:29] == 4'b1000) &&
       (stage4_mem_addr == 1'b0)) ? 0:1;/* clear vals */
assign out1_sched_web = ((stage4_source_reg[32:29] == 4'b1001) &&
       (stage4_mem_addr == 1'b0)) ? 1:0; /*SchRAM */
assign stage4_iw_web = ((stage4_source_reg[32:29] == 4'b1010) &&
           (stage4_mem_addr == 1'b0)) ? 1:0; /*iwRAM */
assign stage4_toggle = ((stage4_source_reg[32:29] == 4'b1011) &&
           (stage4_mem_addr == 1'b0)) ?
           0:1; /* boot*/
assign stage4_jtag_web = ((stage4_source_reg[32:29] == 4'b1101) &&
           (stage4_mem_addr == 1'b0)) ?
           0:1; /*jtag*/
assign stage4_clk_sys_web = ((stage4_source_reg[32:29] == 4'b1110) &&
           (stage4_mem_addr == 1'b0)) ?
           0:1; /* out clocks */
assign out1_sched_ptr_web = ((stage4_source_reg[32:29] == 4'b1111 ) &&
       (stage4_mem_addr == 1'b0)) ? 0:1; /*SchRAM ptr*/

assign stage3_write_detect = stage3_write_ports[3] | stage3_write_ports[2] |
         stage3_write_ports[1] | stage3_write_ports[0];

/*myreg wtdt (stage3_write_detect, port_write_detect, clk);*/

/* This line merely renames the source word.  I have it only to make
warning messages disappear in the compiler */

assign outdp_write_data[32] = (((stage2_pipe_read_write == 1) &&
             (pre_stage2_accept == 0)) ||
            ((stage3_cmove == 1) && (wsuccess == 0)) ||
            ((pipe_read_write == 1) &&
      (dp_accept_final == 0))) ?
            0:stage3_source_reg_pre;
assign outdp_write_data[31:0] = stage3_source_reg[31:0];
assign stage3_source_reg[32] = ((pipe_read_write == 1) &&
      (dp_accept_final == 0)) ?
      0:stage3_source_reg_pre;

/* this is the address of the p-register being written.  The initial
enable was calculated in the previous stage and comes straight from a
register.  The final write enable depends on the initial write enable
being set plus one of the following: 1. a blind move 2. A flow control
move and the destination valid bit being zero 3. a cmove plus the
wsuccess being set. */

assign outdp_preg_waddr = stage4_write_addr[3:0];
assign stage3_outdp_preg_web = ((stage3_preg_write_en == 0) &&
        ((stage3_bmove == 1) ||
        ((stage3_fmove == 1) && (dp_preg[32] == 0)) ||
        ((stage3_cmove == 1) && (wsuccess == 1)))
                && (stage3_source_reg[32] == 1)) ? 0:1;
myreg pweb (stage3_outdp_preg_web, outdp_preg_web, clk);

/* the bit that gets written into the B register will be invalid if
the write succeeds */
assign stage4_valid_bit = ((wsuccess == 0) ||
        ((stage4_breg_waddr == stage4_write_addr[4:0]) &&
         (stage4_write_addr[5] == 1'b1)))  ? stage4_source_reg[32]:0;

/* The b register write address is either the thread number or a
buffer that is encoded as a destination.  It is a buffer encoded as a
destination if the address is specified in the instruction and one of
the following: 1. it is a bmove 2. it is a flow control move and the
destination valid bit is clear 3. it is a cmove and the previous
wsucess is set. */

assign stage3_breg_waddr = ((stage3_iw[5] == 1) &&
      (((stage3_fmove == 1) && (stage3_other_valid == 0)) ||
       ((stage3_cmove == 1) && (wsuccess == 1)) ||
       (stage3_bmove == 1))) ?
         stage3_iw[4:0]:finstage3_index;
myreg #(5) bwt (stage3_breg_waddr, stage4_breg_waddr, clk);

assign stage4_breg_web = (stage4_source_reg[32] == 1'b0) ? 0:1;

/* a write success fails if a conditional flow control statement sees
the wsuccess bit not set or if a flow control statement tries to write
a port but doesn't receive the accept bit, or if a write to a
processor register or buffer register fails */

/*assign wsuccess = (((stage4_fmove == 1) && (port_write_detect ==
               1'b1) && (dp_accept_final == 1'b0))
              || ((stage4_cmove == 1) &&
               (stage4_wsuccess == 1'b0))
              || (stage4_preg_write_en !=
               outdp_preg_web)
              || ((stage4_breg_waddr != stage4_write_addr[4:0]) &&
               (stage4_write_addr[5] == 1'b1)))  ? 0:1;

myreg wsuccess_reg(.D(wsuccess), .Q(stage4_wsuccess), .CLK(clk));
```

```
*/

assign wsuca = ((stage3_fmove == 1) && (stage3_write_detect == 1))
            ? 1:0;
assign wsucb =       ((stage3_breg_waddr != stage3_iw[4:0]) &&
             (stage3_iw[5] == 1)) ? 0:1;

myreg w1 (wsuca, stage4_wsuca, clk);
myreg w2 (wsucb, stage4_wsucb, clk);

/* the wsuccess bit gets registered */

assign wsuccess = (((stage4_wsuca == 1) && (dp_accept_final == 0)) ||
               (stage4_wsucb == 0) ||
               (stage4_preg_write_en != outdp_preg_web) ||
               ((stage4_cmove == 1) && (stage4_wsuccess == 0))) ? 0:1;

myreg wsuccess_reg(.D(wsuccess), .Q(stage4_wsuccess), .CLK(clk));
endmodule
```

“datapath.v”

```
module datapath(outdp_src_preg_addr1, outdp_accept1, outdp_write_ports1,
        outdp_write_data1, outdp_preg_waddr1, outdp_preg_web1,
        outdp_preg_read_clr1,
        outdp_src_preg_addr2, outdp_accept2, outdp_write_ports2,
        outdp_write_data2, outdp_preg_waddr2, outdp_preg_web2,
        outdp_preg_read_clr2,
        outdp_src_ports1, outdp_src_ports2, proc_AB,
        proc_DB, proc_AEN, proc_LO_ADDR,proc_INT_L,
        proc_P_REPLY, proc_S_REPLY, dp_src1, dp_preg1,
        dp_accept1, dp_src2, dp_preg2, dp_accept2,
        valacc_in, valacc_out, minx_data, posx_data, miny_data,
        posy_data, clk, clear_val, bootmode);

input outdp_accept1, outdp_accept2;
input clk, clear_val, bootmode;
input [3:0] outdp_src_preg_addr1, outdp_src_preg_addr2;
input [4:0] proc_AB;
input [3:0] outdp_src_ports1, outdp_src_ports2;
input [3:0] outdp_write_ports1, outdp_write_ports2;
input [32:0] outdp_write_data1, outdp_write_data2;
input [3:0] outdp_preg_waddr1, outdp_preg_waddr2;
input outdp_preg_web1, outdp_preg_web2;
input outdp_preg_read_clr1, outdp_preg_read_clr2;
input proc_AEN, proc_LO_ADDR;
input [1:0] proc_S_REPLY;

input [3:0] valacc_in;
output [3:0] valacc_out;
inout [31:0] posx_data, minx_data, posy_data, miny_data;

inout [32:0] proc_DB;
output dp_accept1, dp_accept2, proc_INT_L;
output [1:0] proc_P_REPLY;
output [32:0] dp_src1, dp_src2, dp_preg1, dp_preg2;

wire [3:0] write_webs1, write_webs2;
wire proc_web, proc_read;
wire [3:0] procw_addr, rwr_port, rw_port1, rw_port2;
wire [3:0] proc_addr, val1, val2, acc1, acc2, valacc_outfin;
wire [3:0] read_ens1_fin, read_ens2_fin, write_webs1_fin, write_webs2_fin;
wire [32:0] proc_out;
wire [31:0] proc_data;
wire [3:0] temp_write_ports1, temp_write_ports2, read_ens1, read_ens2;
wire [15:0] preg_valid_bits;
wire [31:0] dp_src1_pre0, dp_src1_pre1, dp_src1_pre2, dp_src1_pre3;
wire [31:0] dp_src2_pre0, dp_src2_pre1, dp_src2_pre2, dp_src2_pre3;
wire [32:0] dp_src1_pre, dp_src2_pre;
wire [32:0] outdp_write_data1_fin, outdp_write_data2_fin;
wire [3:0] outdp_src_ports1_fin, outdp_src_ports2_fin;
```

```
/* Since our implementation in chip express logic can have only two
ports, we will only allow one pipeline to read or write a processor
register on any given clock cycle */

/* the read_write address from the CFSM can only be one of four
possibilities.  */
assign rw_port1 = (outdp_preg_web1 == 0) ?
      outdp_preg_waddr1:outdp_src_preg_addr1;
assign rw_port2 = (outdp_preg_web2 == 0) ?
      outdp_preg_waddr2:outdp_src_preg_addr2;

/* If either pipeline is trying to write, the write enable is active
(low) */

/* The data port is used for both reading and writing.  If reading,
the port must not be driven by someone else */

/* These correspond to a set of pins, some of which are bidirectional.
The addr, web, and en are all dedicated input pins from the processor.
If the processor is enabled and the web is high (inactive), the
proc_out gets written to the processor.  If the enable is active and
the web is active, the write data appears on the proc_data lines.  The
proc_data_port corresponds to the physical bidirectional port of the
processor.  Proc_port_web and proc_port_en also correspond to the port
signals.  The proc_biport also handles making the actually processor
access every other CFSM cycle (processor cycle is twice as slow as
CFSM).  This is a little shaky, the processor clock is put in a
logical or for both the processor enable and the processor write
enable.  My first analysis says that even if skew messes me up, I will
write an incorrect value followed by writing the correct value.  The
only possible flaw is that the address is not set up in time, and I
end up writing a random location.  */
proc_biport pads (proc_AB, proc_AEN,
        proc_out, proc_DB, proc_S_REPLY, proc_P_REPLY,
        proc_LO_ADDR, proc_INT_L, proc_addr, proc_web, proc_read,
        proc_data, clk, preg_valid_bits, bootmode);

/* This is the set of 16 processor registers.  The inputs are ordered
as follows: CFSM address, CFSM write port, CFSM read port, CFSM web,
proc read/write address, proc write port, proc read port, proc web,
clk */

my_ram16x32 proc_ram (rw_port1, outdp_write_data1_fin[31:0], dp_preg1[31:0],
           rw_port2, outdp_write_data2_fin[31:0], dp_preg2[31:0],
           proc_addr[3:0], proc_data, proc_out[31:0],
           outdp_preg_web1, outdp_preg_web2, proc_web, clk);

/* There are four locations being used on any clock cycle: The first
two ports are read ports read the two CFSM read addresses into the
valid bits of the two processor register busses.  The proc addr causes
a valid bit to be read into proc_out[32].   */
RAM_3portclr preg_valid(outdp_src_preg_addr1, outdp_src_preg_addr2,
             proc_addr[3:0], outdp_preg_waddr1,
      outdp_preg_waddr2, outdp_write_data1_fin[32],
       outdp_write_data2_fin[32],
       outdp_preg_web1, outdp_preg_web2, proc_web,
         dp_preg1[32], dp_preg2[32], proc_out[32],
               outdp_preg_read_clr1, outdp_preg_read_clr2, proc_read,
               clk, clear_val, preg_valid_bits);

/* dealing with multiple writes */
/* if a port is being written by both pipelines, only one will succeed
and this will be determined by a bit that will flip every time a
decision is made */

/* if a write data word is invalid, I will never to a write so I
disable all the port write signals */
assign write_webs1 = (outdp_write_data1[32] == 0) ?
          4'b0000:outdp_write_ports1;

assign write_webs2 = (outdp_write_data2[32] == 0) ?
          4'b0000:outdp_write_ports2;

/* logic for handling two reads to the same port.  If an accept is not
going to be generated, then I don't want to read the port.  Pipeline 1
will always win if both pipelines are trying to read the same port.
The valid bits are handled elsewhere. */

assign read_ens1 = (outdp_accept1 == 0) ? 4'b0000:outdp_src_ports1;
assign read_ens2 = (outdp_accept2 == 0) ? 4'b0000:outdp_src_ports2;

assign valacc_outfin[0] = read_ens1_fin[0] | read_ens2_fin[0] |
              write_webs1_fin[0] | write_webs2_fin[0];
assign valacc_outfin[1] = read_ens1_fin[1] | read_ens2_fin[1] |
              write_webs1_fin[1] | write_webs2_fin[1];
assign valacc_outfin[2] = read_ens1_fin[2] | read_ens2_fin[2] |
              write_webs1_fin[2] | write_webs2_fin[2];
assign valacc_outfin[3] = read_ens1_fin[3] | read_ens2_fin[3] |
              write_webs1_fin[3] | write_webs2_fin[3];

myreg #(4) rd1 (read_ens1, read_ens1_fin, clk);
myreg #(4) rd2 (read_ens2, read_ens2_fin, clk);

myreg #(4) rd1b (outdp_src_ports1, outdp_src_ports1_fin, clk);
myreg #(4) rd2b (outdp_src_ports2, outdp_src_ports2_fin, clk);

myreg #(4) wt1 (write_webs1, write_webs1_fin, clk);
myreg #(4) wt2 (write_webs2, write_webs2_fin, clk);

myreg #(33) wdat1 (outdp_write_data1, outdp_write_data1_fin, clk);
myreg #(33) wdat2 (outdp_write_data2, outdp_write_data2_fin, clk);

/* must zero out incoming accept if not writing a port */
assign dp_accept1 = acc1[0] | acc1[1] | acc1[2] | acc1[3];
assign dp_accept2 = acc2[0] | acc2[1] | acc2[2] | acc2[3];

/* send valid and accept bits */
/* the valacc driver is simply the logical or of the four signals that
```

```
all can drive the line high.  */

ostc06hh padout0 (.D(valacc_outfin[0]), .Z(valacc_out[0]));
ostc06hh padout1 (.D(valacc_outfin[1]), .Z(valacc_out[1]));
ostc06hh padout2 (.D(valacc_outfin[2]), .Z(valacc_out[2]));
ostc06hh padout3 (.D(valacc_outfin[3]), .Z(valacc_out[3]));

/* receive valid and accept bits */
/* the valid receive consists of the four signals trying to drive the
external val/acc line onto the appropriate signal: read_ens->dp_src1,
read_ens2->dp_src2, write_webs1->dp_accept1, write_webs2->dp_accept2.
It is here that the cross transfer valid/accept change occurs as well.
If a pipeline is trying to read and the opposite pipeline is trying to
write, the accept is the logical or of the external accept and the
reading pipeline's accept. */
valacc_receive minxr(val1[0], val2[0], acc1[0], acc2[0],
        read_ens1_fin[0], read_ens2_fin[0],
        write_webs1_fin[0], write_webs2_fin[0], valacc_in[0], clk);

valacc_receive posxr(val1[1], val2[1], acc1[1], acc2[1],
        read_ens1_fin[1], read_ens2_fin[1],
        write_webs1_fin[1], write_webs2_fin[1], valacc_in[1], clk);

valacc_receive minyr(val1[2], val2[2], acc1[2], acc2[2],
        read_ens1_fin[2], read_ens2_fin[2],
        write_webs1_fin[2], write_webs2_fin[2], valacc_in[2], clk);

valacc_receive posyr(val1[3], val2[3], acc1[3], acc2[3],
        read_ens1_fin[3], read_ens2_fin[3],
        write_webs1_fin[3], write_webs2_fin[3], valacc_in[3], clk);

/* must zero out incoming valids if not writing a port */

myand4 dp1 (dp_src1_pre0, dp_src1_pre1, dp_src1_pre2,
        dp_src1_pre3, dp_src1_pre[31:0]);
assign dp_src1_pre[32] = val1[0] | val1[1] | val1[2] | val1[3];

myand4 dp2 (dp_src2_pre0, dp_src2_pre1, dp_src2_pre2,
        dp_src2_pre3, dp_src2_pre[31:0]);
assign dp_src2_pre[32] = val2[0] | val2[1] | val2[2] | val2[3];


assign dp_src1 = ((outdp_src_ports1_fin == write_webs1_fin) &&
         (outdp_src_ports1_fin != 4'h0)) ? outdp_write_data1_fin:
        ((outdp_src_ports1_fin == write_webs2_fin) &&
         (outdp_src_ports1_fin != 4'h0)) ? outdp_write_data2_fin:
        dp_src1_pre;

assign dp_src2 = ((outdp_src_ports2_fin == write_webs2_fin) &&
         (outdp_src_ports2_fin != 4'h0)) ? outdp_write_data2_fin:
        ((outdp_src_ports2_fin == write_webs1_fin) &&
         (outdp_src_ports2_fin != 4'h0)) ? outdp_write_data1_fin:
        dp_src2_pre;
```

```
/* These are the thirty two bit datapaths.  The ports are ordered as
follows: wdat1, wen1, ren1, wdat2, wen2, ren2, rdat1, rdat2, external
port.  At this point, only one of the pipelines will be reading or
writing the port.  It can be the case that I am reading and writing
the ports, it's sort of a cheap way to do pipeline swapping.  */

biport_32 minx1(outdp_write_data1_fin[31:0], write_webs1_fin[0],
      read_ens1_fin[0], outdp_write_data2_fin[31:0],
      write_webs2_fin[0], read_ens2_fin[0], dp_src1_pre0,
      dp_src2_pre0, minx_data, clk);

biport_32 posx1(outdp_write_data1_fin[31:0], write_webs1_fin[1],
      read_ens1_fin[1], outdp_write_data2_fin[31:0],
      write_webs2_fin[1], read_ens2_fin[1], dp_src1_pre1,
      dp_src2_pre1, posx_data, clk);

biport_32 miny1(outdp_write_data1_fin[31:0], write_webs1_fin[2],
      read_ens1_fin[2], outdp_write_data2_fin[31:0],
      write_webs2_fin[2], read_ens2_fin[2], dp_src1_pre2,
      dp_src2_pre2, miny_data, clk);

biport_32 posy1(outdp_write_data1_fin[31:0], write_webs1_fin[3],
      read_ens1_fin[3], outdp_write_data2_fin[31:0],
      write_webs2_fin[3], read_ens2_fin[3], dp_src1_pre3,
      dp_src2_pre3, posy_data, clk);

endmodule
```

“CFSM.v”

```
`timescale 1ns / 10ps
`include “modules.v”
`include “sched.v”
`include “pipeline0.v”
`include “pipeline1.v”
`include “datapath.v”

module CFSM(valacc_in, valacc_out, minx_data,  posx_data,
        miny_data, posy_data, pjtag_in, pjtag_pin, clk_sys_pin,
        proc_AB, proc_DB,
        proc_AEN, proc_LO_ADDR, proc_INT_L, proc_P_REPLY,
        proc_S_REPLY, LED_pin, clk, reset, proc_reset_port,
        power_reset);

input clk, power_reset, proc_AEN, proc_LO_ADDR, pjtag_in;
input [1:0] proc_S_REPLY;
inout [3:0] reset;

inout [31:0] minx_data, posx_data, miny_data, posy_data;
inout [32:0] proc_DB;
input [3:0] valacc_in;
input [4:0] proc_AB;

output [3:0] valacc_out, LED_pin;
output [1:0] proc_P_REPLY;
output proc_INT_L, proc_reset_port;
output [2:0] pjtag_pin;
output [3:0] clk_sys_pin;

wire clk, stage4_sched_web1;
wire [4:0] stage2_index1, stage2_index2;
wire [32:0] dp_preg1, dp_preg2, dp_src1, dp_src2;
wire dp_accept1, dp_accept2;
wire outdp_accept1, outdp_accept2;
wire [3:0] outdp_src_preg_addr1, outdp_src_preg_addr2;
wire [3:0] outdp_src_ports1, outdp_src_ports2;
wire [3:0] outdp_write_ports1, outdp_write_ports2;
wire [32:0] outdp_write_data1, outdp_write_data2;
wire [32:0] stage4_source_reg1, stage4_source_reg2;
wire [3:0] outdp_preg_waddr1, outdp_preg_waddr2;
wire outdp_preg_web1, outdp_preg_web2;
wire outdp_preg_read_clr1, outdp_preg_read_clr2;
wire bootmode1, next_bootmode1, bootmode2, next_bootmode2;
wire [12:0] toggle;
wire stage4_toggle1, stage4_toggle2;
wire bootmode;
wire [3:0] next_clk_sys, clk_sys;
wire clk_sys1, clk_sys2, jtag1, jtag2;
wire [3:0] next_jtag, jtag;
wire reset_mode, stage4_clr_val1;
wire jtag_in_pre, jtag_in_fin, stage4_clr_val2, clear_val, power_reset_final;
wire [3:0] reset_in, led, next_led;
```

```
/* reset, boot toogle, and bootmode are low active */

/* boot mode is entered anytime the global reset is deasserted.  Also
it can be toggled by writing the appropriate memory location */

/* clk_driver ckd (clk_in, clk); */

assign next_bootmode1 = ((reset_mode == 1) |
        (((toggle[2] == 0) && (toggle[0] == 0)) ||
         ((toggle[2] == 1) && (bootmode1 == 0)))) ? 0:1;

assign next_bootmode2 = ((reset_mode == 0) &&
        (((toggle[2] == 0) && (toggle[1] == 0)) ||
         ((toggle[2] == 1) && (bootmode2 == 0)))) ? 0:1;

myreg  bootreg1 (next_bootmode1, bootmode1, clk);
myreg  bootreg2 (next_bootmode2, bootmode2, clk);

assign bootmode = bootmode1 & bootmode2;

assign toggle = (stage4_toggle1 == 0) ? stage4_source_reg1[12:0]:
       (stage4_toggle2 == 0) ? stage4_source_reg2[12:0]:
              13'b1111111111111;

reset_port rp (toggle[7:3], toggle[11:8], reset, proc_reset_port,
              reset_in, power_reset, power_reset_final, bootmode,
              reset_mode, toggle[12], clk);

assign next_jtag = (power_reset_final == 1) ? 4'b0000:
          (jtag1 == 0) ? stage4_source_reg1[3:0]:
          (jtag2 == 0) ? stage4_source_reg2[3:0]:
                 jtag;
myreg #(4) jtag_reg (next_jtag, jtag, clk);

ossc06hh padj0 (.D(jtag[0]), .OE(jtag[3]), .Z(pjtag_pin[0]));
ossc06hh padj1 (.D(jtag[1]), .OE(jtag[3]), .Z(pjtag_pin[1]));
ossc06hh padj2 (.D(jtag[2]), .OE(jtag[3]), .Z(pjtag_pin[2]));

assign next_led = (power_reset_final == 1) ? 4'b1111:
          (jtag1 == 0) ? stage4_source_reg1[7:4]:
          (jtag2 == 0) ? stage4_source_reg2[7:4]:
                 led;
myreg #(4) led_reg (next_led, led, clk);

ostt12hh padl0 (.D(led[0]), .Z(LED_pin[0]));
ostt12hh padl1 (.D(led[1]), .Z(LED_pin[1]));
ostt12hh padl2 (.D(led[2]), .Z(LED_pin[2]));
ostt12hh padl3 (.D(led[3]), .Z(LED_pin[3]));

iptndh padjin1 (.Z(pjtag_in), .ZI(jtag_in_pre));
dfnnn ltjtag (.D(jtag_in_pre), .Q(jtag_in_fin), .CPN(clk));

assign next_clk_sys =  (power_reset_final == 1) ? 4'b0000:
```

```
                (clk_sys1 == 0) ? stage4_source_reg1[3:0]:
                (clk_sys2 == 0) ? stage4_source_reg2[3:0]:
                        clk_sys;
myreg #(4) clk_sys_reg (next_clk_sys, clk_sys, clk);

ostc06hh pads0 (.D(clk_sys[0]), .Z(clk_sys_pin[0]));
ostc06hh pads1 (.D(clk_sys[1]), .Z(clk_sys_pin[1]));
ostc06hh pads2 (.D(clk_sys[2]), .Z(clk_sys_pin[2]));
ostc06hh pads3 (.D(clk_sys[3]), .Z(clk_sys_pin[3]));

assign clear_val = stage4_clr_val1 & stage4_clr_val2 & bootmode;

schedule_ram t1(stage4_source_reg1[28:22], stage4_sched_web1,
        stage4_source_reg1[16:0],
        stage4_sched_ptr_web1,
        stage2_index1, stage2_index2,
        clk, bootmode);

pipeline0 pipeA(stage2_index1, dp_src1, dp_preg1, dp_accept1,
               stage4_sched_web1,
               outdp_src_ports1,
               outdp_src_preg_addr1, outdp_accept1, outdp_write_ports1,
               outdp_write_data1, outdp_preg_waddr1, outdp_preg_web1,
              outdp_preg_read_clr1,
              stage4_sched_ptr_web1,
              clk, bootmode1, bootmode2, clear_val,
              stage4_toggle1, stage4_clr_val1, jtag1,
              clk_sys1, reset_in, jtag_in_fin, stage4_source_reg1);

pipeline1 pipeB(stage2_index2, dp_src2, dp_preg2, dp_accept2,
               outdp_src_ports2,
               outdp_src_preg_addr2, outdp_accept2, outdp_write_ports2,
               outdp_write_data2, outdp_preg_waddr2, outdp_preg_web2,
              outdp_preg_read_clr2,
              clk, bootmode2, bootmode1, clear_val,
              stage4_toggle2, stage4_clr_val2, jtag2,
              clk_sys2, reset_in, jtag_in_fin, stage4_source_reg2);

datapath dp(outdp_src_preg_addr1, outdp_accept1, outdp_write_ports1,
             outdp_write_data1, outdp_preg_waddr1, outdp_preg_web1,
            outdp_preg_read_clr1,
            outdp_src_preg_addr2, outdp_accept2,
            outdp_write_ports2, outdp_write_data2, outdp_preg_waddr2,
            outdp_preg_web2, outdp_preg_read_clr2,
            outdp_src_ports1,
            outdp_src_ports2, proc_AB, proc_DB,
            proc_AEN, proc_LO_ADDR, proc_INT_L, proc_P_REPLY, proc_S_REPLY,
            dp_src1, dp_preg1, dp_accept1, dp_src2, dp_preg2,
            dp_accept2, valacc_in, valacc_out, minx_data,
            posx_data, miny_data, posy_data, clk, clear_val, bootmode);

endmodule
                /*Beware of the man-eating cow! */
```

“modules.v”

```
/* This is a listing of basic modules */

module clk_driver(clk_in, clk);
input clk_in;
output clk;
wire clk_pre;

iptnnh padclk1 (.Z(clk_in), .ZI(clk_pre));
cdqn6 clktree (.A(clk_pre), .Z(clk));

endmodule

module mymux_2(A,B,SEL,Z);
// synopsys template
parameter w = 1;
input [w-1:0] A,B;
input SEL;
output [w-1:0] Z;

assign Z = (SEL == 0) ? A:B;
endmodule

module mymux_3(A,B,C,SEL,Z);
/* three way mux A=00 or 01, B=10, C=11 */
// synopsys template
parameter w = 1;
input [w-1:0] A,B,C;
input [1:0] SEL;
output [w-1:0] Z;
assign Z = (SEL[0] == 0) ? B:
                    (SEL == 2'b01) ? A:C;
endmodule

module mymux_4(A,B,C,D,SEL,Z);
/* three way mux A=00 or B=01, C=10, D=11 */
// synopsys template
parameter w = 1;
input [w-1:0] A,B,C,D;
input [1:0] SEL;
output [w-1:0] Z;

assign Z = (SEL == 2'b00) ? A:
                    (SEL == 2'b01) ? B:
                    (SEL == 2'b10) ? C:D;
endmodule

module myreg(D,Q,CLK);
  // synopsys template
  parameter w = 1;
  input [w-1:0] D;
  output [w-1:0] Q;
  reg [w-1:0] Q;
  input CLK;

  always @(posedge CLK)
    Q = D;

endmodule

module mylatch(D, G, Q);
// synopsys template
  parameter width=1;
  input G;
  input [width-1:0] D;
  output [width-1:0] Q;

  reg [width-1:0] Q;

  always @(G or D)
    begin
      if( G)
        Q = D;
    end

endmodule


module myreg_low(D,Q,CLK);
  // synopsys template
  parameter w = 1;
  input [w-1:0] D;
  output [w-1:0] Q;
  reg [w-1:0] Q;
  input CLK;

  always @(negedge CLK)
    Q = D;

endmodule

module mycomparator (A,B,Z);
// synopsys template
parameter w = 1;
/* Z is 0 if A = B */
input [w-1:0] A, B;
output Z;

assign Z = (A==B) ? 0:1;

endmodule

module mytristate( A,E,Z);
// synopsys template
```

```
parameter width = 1;

input E;
input [width-1:0] A;
output [width-1:0] Z;

wire [width-1:0] Z = E ? A : 'bz;

endmodule

module myand (A, B, C);
// synopsys template

input [15:0] A,B;
output [15:0] C;

assign C[0] = A[0] & B[0];
assign C[1] = A[1] & B[1];
assign C[2] = A[2] & B[2];
assign C[3] = A[3] & B[3];
assign C[4] = A[4] & B[4];
assign C[5] = A[5] & B[5];
assign C[6] = A[6] & B[6];
assign C[7] = A[7] & B[7];
assign C[8] = A[8] & B[8];
assign C[9] = A[9] & B[9];
assign C[10] = A[10] & B[10];
assign C[11] = A[11] & B[11];
assign C[12] = A[12] & B[12];
assign C[13] = A[13] & B[13];
assign C[14] = A[14] & B[14];
assign C[15] = A[15] & B[15];

endmodule

module myand4 (A, B, C, D, E);
// synopsys template

input [31:0] A,B,C,D;
output [31:0] E;

assign E[0] = A[0] & B[0] & C[0] & D[0];
assign E[1] = A[1] & B[1] & C[1] & D[1];
assign E[2] = A[2] & B[2] & C[2] & D[2];
assign E[3] = A[3] & B[3] & C[3] & D[3];
assign E[4] = A[4] & B[4] & C[4] & D[4];
assign E[5] = A[5] & B[5] & C[5] & D[5];
assign E[6] = A[6] & B[6] & C[6] & D[6];
assign E[7] = A[7] & B[7] & C[7] & D[7];
assign E[8] = A[8] & B[8] & C[8] & D[8];
assign E[9] = A[9] & B[9] & C[9] & D[9];
assign E[10] = A[10] & B[10] & C[10] & D[10];
assign E[11] = A[11] & B[11] & C[11] & D[11];
assign E[12] = A[12] & B[12] & C[12] & D[12];
assign E[13] = A[13] & B[13] & C[13] & D[13];
assign E[14] = A[14] & B[14] & C[14] & D[14];
assign E[15] = A[15] & B[15] & C[15] & D[15];
assign E[16] = A[16] & B[16] & C[16] & D[16];
assign E[17] = A[17] & B[17] & C[17] & D[17];
assign E[18] = A[18] & B[18] & C[18] & D[18];
assign E[19] = A[19] & B[19] & C[19] & D[19];
assign E[20] = A[20] & B[20] & C[20] & D[20];
assign E[21] = A[21] & B[21] & C[21] & D[21];
assign E[22] = A[22] & B[22] & C[22] & D[22];
assign E[23] = A[23] & B[23] & C[23] & D[23];
assign E[24] = A[24] & B[24] & C[24] & D[24];
assign E[25] = A[25] & B[25] & C[25] & D[25];
assign E[26] = A[26] & B[26] & C[26] & D[26];
assign E[27] = A[27] & B[27] & C[27] & D[27];
assign E[28] = A[28] & B[28] & C[28] & D[28];
assign E[29] = A[29] & B[29] & C[29] & D[29];
assign E[30] = A[30] & B[30] & C[30] & D[30];
assign E[31] = A[31] & B[31] & C[31] & D[31];

endmodule

module reset_port(reset_out_write, reset_mask_write,  reset_port,
                                    proc_reset_port, final_reset_in, power_reset,
                                    power_reset_final, bootmode, reset_mode,
                                    toggle_web, clk);

input [4:0] reset_out_write;
input [3:0] reset_mask_write;
input power_reset, clk, bootmode, toggle_web;
inout [3:0] reset_port;
output proc_reset_port;
output reset_mode, power_reset_final;
output [3:0] final_reset_in;

wire next_proc_reset, proc_reset;
wire [3:0] next_reset_out, next_reset_mask, reset_in;
wire [3:0] reset_out, reset_mask, pre_reset_in, next_reset_in;



reset_pads rset_pads(reset_out, reset_port, reset_in,
                                     proc_reset, proc_reset_port, clk );

assign next_proc_reset = (power_reset_final == 1) ? 0:
                                                 (toggle_web == 0) ? reset_out_write[4]:proc_reset;

myreg proc_reset_reg (next_proc_reset, proc_reset, clk);

assign next_reset_out = (power_reset_final == 1) ? 4'h0:
                                                (toggle_web == 0) ?
                                                 reset_out_write[3:0]:reset_out;

assign next_reset_mask = (power_reset_final == 1) ? 4'hF:
```

```
                                                    (toggle_web == 0) ?
                                                    reset_mask_write:reset_mask;

myreg #(4) reset_out_reg (next_reset_out, reset_out, clk);
myreg #(4) reset_mask_reg (next_reset_mask, reset_mask, clk);


assign next_reset_in = (power_reset_final == 1) ? 4'h0:
                                          ((bootmode == 0) && (final_reset_in != 4'h0))  ?
                                          final_reset_in:reset_in;
myreg #(4) reset_in_reg (next_reset_in, pre_reset_in, clk);


assign final_reset_in[0] = pre_reset_in[0] & reset_mask[0];
assign final_reset_in[1] = pre_reset_in[1] & reset_mask[1];
assign final_reset_in[2] = pre_reset_in[2] & reset_mask[2];
assign final_reset_in[3] = pre_reset_in[3] & reset_mask[3];

assign reset_mode = ((power_reset_final == 1) || ((bootmode == 1) &&
                                         (next_reset_in != 4'h0))) ? 1:0;

ipcnnh padpr1 (.ZI(power_reset_pre), .Z(power_reset));
dfnnn ltres (.D(power_reset_pre), .Q(power_reset_final), .CPN(clk));

endmodule

module reset_pads (reset_out, reset_port, reset_in, proc_reset,
                                        proc_reset_port, clk);

input [3:0] reset_out;
input proc_reset, clk;
inout [3:0] reset_port;
output [3:0] reset_in;
output proc_reset_port;
wire [3:0] reset_in_pre;

bstcnd06hh padbit0 (.D(reset_out[0]), .OE(reset_out[0]),
                              .Z(reset_port[0]), .ZI(reset_in_pre[0]));
bstcnd06hh padbit1 (.D(reset_out[1]), .OE(reset_out[1]),
                             .Z(reset_port[1]), .ZI(reset_in_pre[1]));
bstcnd06hh padbit2 (.D(reset_out[2]), .OE(reset_out[2]),
                              .Z(reset_port[2]), .ZI(reset_in_pre[2]));
bstcnd06hh padbit3 (.D(reset_out[3]), .OE(reset_out[3]),
                              .Z(reset_port[3]), .ZI(reset_in_pre[3]));

dfnnn ltres0 (.D(reset_in_pre[0]), .Q(reset_in[0]), .CPN(clk));
dfnnn ltres1 (.D(reset_in_pre[1]), .Q(reset_in[1]), .CPN(clk));
dfnnn ltres2 (.D(reset_in_pre[2]), .Q(reset_in[2]), .CPN(clk));
dfnnn ltres3 (.D(reset_in_pre[3]), .Q(reset_in[3]), .CPN(clk));

ostc06hh padpr (.D(proc_reset), .Z(proc_reset_port));

endmodule
```

```
module biport_32(wdat1, wen1, ren1, wdat2, wen2, ren2, rdat1, rdat2,
                                        port_data, clk);
/* enable is high */
input [31:0] wdat1, wdat2;
input  wen1, ren1, wen2, ren2, clk;
inout [31:0] port_data;
output [31:0] rdat1, rdat2;

wire [31:0] r_port, wdat, r_port_pre;
wire wen;

assign wen = wen1 | wen2;

bstcnu06hh padb0 (.D(wdat[0]), .OE(wen), .Z(port_data[0]), .ZI(r_port_pre[0]));
bstcnu06hh padb1 (.D(wdat[1]), .OE(wen), .Z(port_data[1]), .ZI(r_port_pre[1]));
bstcnu06hh padb2 (.D(wdat[2]), .OE(wen), .Z(port_data[2]), .ZI(r_port_pre[2]));
bstcnu06hh padb3 (.D(wdat[3]), .OE(wen), .Z(port_data[3]), .ZI(r_port_pre[3]));
bstcnu06hh padb4 (.D(wdat[4]), .OE(wen), .Z(port_data[4]), .ZI(r_port_pre[4]));
bstcnu06hh padb5 (.D(wdat[5]), .OE(wen), .Z(port_data[5]), .ZI(r_port_pre[5]));
bstcnu06hh padb6 (.D(wdat[6]), .OE(wen), .Z(port_data[6]), .ZI(r_port_pre[6]));
bstcnu06hh padb7 (.D(wdat[7]), .OE(wen), .Z(port_data[7]), .ZI(r_port_pre[7]));
bstcnu06hh padb8 (.D(wdat[8]), .OE(wen), .Z(port_data[8]), .ZI(r_port_pre[8]));
bstcnu06hh padb9 (.D(wdat[9]), .OE(wen), .Z(port_data[9]), .ZI(r_port_pre[9]));
bstcnu06hh padb10 (.D(wdat[10]), .OE(wen), .Z(port_data[10]), .ZI(r_port_pre[10]));
bstcnu06hh padb11 (.D(wdat[11]), .OE(wen), .Z(port_data[11]), .ZI(r_port_pre[11]));
bstcnu06hh padb12 (.D(wdat[12]), .OE(wen), .Z(port_data[12]), .ZI(r_port_pre[12]));
bstcnu06hh padb13 (.D(wdat[13]), .OE(wen), .Z(port_data[13]), .ZI(r_port_pre[13]));
bstcnu06hh padb14 (.D(wdat[14]), .OE(wen), .Z(port_data[14]), .ZI(r_port_pre[14]));
bstcnu06hh padb15 (.D(wdat[15]), .OE(wen), .Z(port_data[15]), .ZI(r_port_pre[15]));
bstcnu06hh padb16 (.D(wdat[16]), .OE(wen), .Z(port_data[16]), .ZI(r_port_pre[16]));
bstcnu06hh padb17 (.D(wdat[17]), .OE(wen), .Z(port_data[17]), .ZI(r_port_pre[17]));
bstcnu06hh padb18 (.D(wdat[18]), .OE(wen), .Z(port_data[18]), .ZI(r_port_pre[18]));
bstcnu06hh padb19 (.D(wdat[19]), .OE(wen), .Z(port_data[19]), .ZI(r_port_pre[19]));
bstcnu06hh padb20 (.D(wdat[20]), .OE(wen), .Z(port_data[20]), .ZI(r_port_pre[20]));
bstcnu06hh padb21 (.D(wdat[21]), .OE(wen), .Z(port_data[21]), .ZI(r_port_pre[21]));
bstcnu06hh padb22 (.D(wdat[22]), .OE(wen), .Z(port_data[22]), .ZI(r_port_pre[22]));
bstcnu06hh padb23 (.D(wdat[23]), .OE(wen), .Z(port_data[23]), .ZI(r_port_pre[23]));
bstcnu06hh padb24 (.D(wdat[24]), .OE(wen), .Z(port_data[24]), .ZI(r_port_pre[24]));
bstcnu06hh padb25 (.D(wdat[25]), .OE(wen), .Z(port_data[25]), .ZI(r_port_pre[25]));
bstcnu06hh padb26 (.D(wdat[26]), .OE(wen), .Z(port_data[26]), .ZI(r_port_pre[26]));
bstcnu06hh padb27 (.D(wdat[27]), .OE(wen), .Z(port_data[27]), .ZI(r_port_pre[27]));
bstcnu06hh padb28 (.D(wdat[28]), .OE(wen), .Z(port_data[28]), .ZI(r_port_pre[28]));
bstcnu06hh padb29 (.D(wdat[29]), .OE(wen), .Z(port_data[29]), .ZI(r_port_pre[29]));
bstcnu06hh padb30 (.D(wdat[30]), .OE(wen), .Z(port_data[30]), .ZI(r_port_pre[30]));
bstcnu06hh padb31 (.D(wdat[31]), .OE(wen), .Z(port_data[31]), .ZI(r_port_pre[31]));

dfnnn ltb0 (.D(r_port_pre[0]), .Q(r_port[0]), .CPN(clk));
dfnnn ltb1 (.D(r_port_pre[1]), .Q(r_port[1]), .CPN(clk));
dfnnn ltb2 (.D(r_port_pre[2]), .Q(r_port[2]), .CPN(clk));
dfnnn ltb3 (.D(r_port_pre[3]), .Q(r_port[3]), .CPN(clk));
dfnnn ltb4 (.D(r_port_pre[4]), .Q(r_port[4]), .CPN(clk));
dfnnn ltb5 (.D(r_port_pre[5]), .Q(r_port[5]), .CPN(clk));
dfnnn ltb6 (.D(r_port_pre[6]), .Q(r_port[6]), .CPN(clk));
dfnnn ltb7 (.D(r_port_pre[7]), .Q(r_port[7]), .CPN(clk));
dfnnn ltb8 (.D(r_port_pre[8]), .Q(r_port[8]), .CPN(clk));
dfnnn ltb9 (.D(r_port_pre[9]), .Q(r_port[9]), .CPN(clk));
dfnnn ltb10 (.D(r_port_pre[10]), .Q(r_port[10]), .CPN(clk));
dfnnn ltb11 (.D(r_port_pre[11]), .Q(r_port[11]), .CPN(clk));
dfnnn ltb12 (.D(r_port_pre[12]), .Q(r_port[12]), .CPN(clk));
dfnnn ltb13 (.D(r_port_pre[13]), .Q(r_port[13]), .CPN(clk));
dfnnn ltb14 (.D(r_port_pre[14]), .Q(r_port[14]), .CPN(clk));
dfnnn ltb15 (.D(r_port_pre[15]), .Q(r_port[15]), .CPN(clk));
dfnnn ltb16 (.D(r_port_pre[16]), .Q(r_port[16]), .CPN(clk));
dfnnn ltb17 (.D(r_port_pre[17]), .Q(r_port[17]), .CPN(clk));
dfnnn ltb18 (.D(r_port_pre[18]), .Q(r_port[18]), .CPN(clk));
dfnnn ltb19 (.D(r_port_pre[19]), .Q(r_port[19]), .CPN(clk));
dfnnn ltb20 (.D(r_port_pre[20]), .Q(r_port[20]), .CPN(clk));
dfnnn ltb21 (.D(r_port_pre[21]), .Q(r_port[21]), .CPN(clk));
dfnnn ltb22 (.D(r_port_pre[22]), .Q(r_port[22]), .CPN(clk));
dfnnn ltb23 (.D(r_port_pre[23]), .Q(r_port[23]), .CPN(clk));
dfnnn ltb24 (.D(r_port_pre[24]), .Q(r_port[24]), .CPN(clk));
dfnnn ltb25 (.D(r_port_pre[25]), .Q(r_port[25]), .CPN(clk));
dfnnn ltb26 (.D(r_port_pre[26]), .Q(r_port[26]), .CPN(clk));
dfnnn ltb27 (.D(r_port_pre[27]), .Q(r_port[27]), .CPN(clk));
dfnnn ltb28 (.D(r_port_pre[28]), .Q(r_port[28]), .CPN(clk));
dfnnn ltb29 (.D(r_port_pre[29]), .Q(r_port[29]), .CPN(clk));
dfnnn ltb30 (.D(r_port_pre[30]), .Q(r_port[30]), .CPN(clk));
dfnnn ltb31 (.D(r_port_pre[31]), .Q(r_port[31]), .CPN(clk));

assign rdat1 = (ren1 == 1) ? r_port:32'hFFFFFFFF;
assign rdat2 = (ren2 == 1) ? r_port:32'hFFFFFFFF;

assign wdat[0] = (wdat1[0] & wen1) | (wdat2[0] & wen2);
assign wdat[1] = (wdat1[1] & wen1) | (wdat2[1] & wen2);
assign wdat[2] = (wdat1[2] & wen1) | (wdat2[2] & wen2);
assign wdat[3] = (wdat1[3] & wen1) | (wdat2[3] & wen2);
assign wdat[4] = (wdat1[4] & wen1) | (wdat2[4] & wen2);
assign wdat[5] = (wdat1[5] & wen1) | (wdat2[5] & wen2);
assign wdat[6] = (wdat1[6] & wen1) | (wdat2[6] & wen2);
assign wdat[7] = (wdat1[7] & wen1) | (wdat2[7] & wen2);
assign wdat[8] = (wdat1[8] & wen1) | (wdat2[8] & wen2);
assign wdat[9] = (wdat1[9] & wen1) | (wdat2[9] & wen2);
assign wdat[10] = (wdat1[10] & wen1) | (wdat2[10] & wen2);
assign wdat[11] = (wdat1[11] & wen1) | (wdat2[11] & wen2);
assign wdat[12] = (wdat1[12] & wen1) | (wdat2[12] & wen2);
assign wdat[13] = (wdat1[13] & wen1) | (wdat2[13] & wen2);
assign wdat[14] = (wdat1[14] & wen1) | (wdat2[14] & wen2);
assign wdat[15] = (wdat1[15] & wen1) | (wdat2[15] & wen2);
assign wdat[16] = (wdat1[16] & wen1) | (wdat2[16] & wen2);
assign wdat[17] = (wdat1[17] & wen1) | (wdat2[17] & wen2);
assign wdat[18] = (wdat1[18] & wen1) | (wdat2[18] & wen2);
assign wdat[19] = (wdat1[19] & wen1) | (wdat2[19] & wen2);
assign wdat[20] = (wdat1[20] & wen1) | (wdat2[20] & wen2);
assign wdat[21] = (wdat1[21] & wen1) | (wdat2[21] & wen2);
assign wdat[22] = (wdat1[22] & wen1) | (wdat2[22] & wen2);
assign wdat[23] = (wdat1[23] & wen1) | (wdat2[23] & wen2);
assign wdat[24] = (wdat1[24] & wen1) | (wdat2[24] & wen2);
assign wdat[25] = (wdat1[25] & wen1) | (wdat2[25] & wen2);
assign wdat[26] = (wdat1[26] & wen1) | (wdat2[26] & wen2);
assign wdat[27] = (wdat1[27] & wen1) | (wdat2[27] & wen2);
assign wdat[28] = (wdat1[28] & wen1) | (wdat2[28] & wen2);
assign wdat[29] = (wdat1[29] & wen1) | (wdat2[29] & wen2);
assign wdat[30] = (wdat1[30] & wen1) | (wdat2[30] & wen2);
assign wdat[31] = (wdat1[31] & wen1) | (wdat2[31] & wen2);

endmodule
/*

module valacc_driver(val1_en, val2_en, acc1_en, acc2_en, port_valacc);

input val1_en, val2_en, acc1_en, acc2_en;

output port_valacc;
wire drive_sig;
assign drive_sig = val1_en | val2_en | acc1_en | acc2_en;



endmodule
*/
module valacc_receive(valid1, valid2, accept1, accept2, val1_en,
                                        val2_en, acc1_en, acc2_en, port_valacc, clk);

input val1_en, val2_en, acc1_en, acc2_en, port_valacc, clk;
output valid1, valid2, accept1, accept2;
wire data, valid1_pre, valid2_pre, acc1_pre, acc2_pre, data_pre;

ipcndh padval (.Z(port_valacc), .ZI(data_pre));
dfnnn ltaden (.D(data_pre), .Q(data), .CPN(clk));

assign valid1 = val1_en & (acc2_en | data);
assign valid2 = val2_en & (acc1_en | data);
assign accept1 = acc1_en & (val2_en | data);
assign accept2 = acc2_en & (val1_en | data);

endmodule

module mydecode_4 (address, lines);
/* outputs are low active */
input [3:0] address;
output [15:0] lines;

assign lines = ~(1'b1 << address);

endmodule

module mydecode_5 (address, lines);
/* outputs are low active */
input [4:0] address;
output [31:0] lines;

assign lines = ~(1'b1 << address);
```

```
endmodule

module mymux32_1 (A, sel, out);

input [31:0] A;
input [4:0] sel;
output out;

assign out = A[sel];
endmodule


module mymux16_1 (A, sel, out);

input [15:0] A;
input [3:0] sel;
output out;

assign out = A[sel];
endmodule

module mymux8_1 (A, sel, out);

input [7:0] A;
input [2:0] sel;
output out;

assign out = A[sel];
endmodule

module RAM_4port_breg(r_add1,r_add2, w_add1, wdat1, w_add2, web1,
                                        r_out1, r_out2, clk, bootmode);

/* this RAM is implemented with a gated 32 bit register.  For this
specialized RAM, there are two read addresses and two write addresses.
The second write address always writes a zero when its write enable is
set.  */

input [4:0] r_add1, r_add2, w_add1, w_add2;
input web1, clk, bootmode;
input wdat1;
output r_out1, r_out2;

wire [31:0] wadd1_dec, wadd2_dec, current_val, next_value, finnext_val;
wire r_out1_pre, r_out2_pre;

assign finnext_val = (bootmode == 0) ? 32'h00000000:next_value;
myreg #(32) bregv(finnext_val, current_val, clk);

assign r_out1_pre = current_val[r_add1];
assign r_out2_pre = current_val[r_add2];

assign r_out1 = ((r_add1 == w_add1) && (web1 == 1)) ? wdat1:
```

```
                                        (r_add1 == w_add2) ? 0:r_out1_pre;

assign r_out2 = ((r_add2 == w_add1) && (web1 == 1)) ? wdat1:r_out2_pre;

mydecode_5 w1decode (w_add1, wadd1_dec);
mydecode_5 w2decode (w_add2, wadd2_dec);

assign next_value[0] = (wadd2_dec[0] == 0) ? 0:
          ((wadd1_dec[0] == 0) && (web1 == 1))
                    ? wdat1:current_val[0];

assign next_value[1] = (wadd2_dec[1] == 0) ? 0:
          ((wadd1_dec[1] == 0) && (web1 == 1))
                    ? wdat1:current_val[1];

assign next_value[2] = (wadd2_dec[2] == 0) ? 0:
          ((wadd1_dec[2] == 0) && (web1 == 1))
                    ? wdat1:current_val[2];

assign next_value[3] = (wadd2_dec[3] == 0) ? 0:
          ((wadd1_dec[3] == 0) && (web1 == 1))
                    ? wdat1:current_val[3];

assign next_value[4] = (wadd2_dec[4] == 0) ? 0:
          ((wadd1_dec[4] == 0) && (web1 == 1))
                    ? wdat1:current_val[4];

assign next_value[5] = (wadd2_dec[5] == 0) ? 0:
          ((wadd1_dec[5] == 0) && (web1 == 1))
                    ? wdat1:current_val[5];

assign next_value[6] = (wadd2_dec[6] == 0) ? 0:
          ((wadd1_dec[6] == 0) && (web1 == 1))
                    ? wdat1:current_val[6];

assign next_value[7] = (wadd2_dec[7] == 0) ? 0:
          ((wadd1_dec[7] == 0) && (web1 == 1))
                    ? wdat1:current_val[7];

assign next_value[8] = (wadd2_dec[8] == 0) ? 0:
          ((wadd1_dec[8] == 0) && (web1 == 1))
                    ? wdat1:current_val[8];

assign next_value[9] = (wadd2_dec[9] == 0) ? 0:
          ((wadd1_dec[9] == 0) && (web1 == 1))
                    ? wdat1:current_val[9];

assign next_value[10] = (wadd2_dec[10] == 0) ? 0:
          ((wadd1_dec[10] == 0) && (web1 == 1))
                    ? wdat1:current_val[10];

assign next_value[11] = (wadd2_dec[11] == 0) ? 0:
          ((wadd1_dec[11] == 0) && (web1 == 1))
                    ? wdat1:current_val[11];
```

```
assign next_value[12] = (wadd2_dec[12] == 0) ? 0:
                ((wadd1_dec[12] == 0) && (web1 == 1))
                                ? wdat1:current_val[12];

assign next_value[13] = (wadd2_dec[13] == 0) ? 0:
                ((wadd1_dec[13] == 0) && (web1 == 1))
                                ? wdat1:current_val[13];

assign next_value[14] = (wadd2_dec[14] == 0) ? 0:
                ((wadd1_dec[14] == 0) && (web1 == 1))
                                ? wdat1:current_val[14];

assign next_value[15] = (wadd2_dec[15] == 0) ? 0:
                ((wadd1_dec[15] == 0) && (web1 == 1))
                                ? wdat1:current_val[15];

assign next_value[16] = (wadd2_dec[16] == 0) ? 0:
                ((wadd1_dec[16] == 0) && (web1 == 1))
                                ? wdat1:current_val[16];

assign next_value[17] = (wadd2_dec[17] == 0) ? 0:
                ((wadd1_dec[17] == 0) && (web1 == 1))
                                ? wdat1:current_val[17];

assign next_value[18] = (wadd2_dec[18] == 0) ? 0:
                ((wadd1_dec[18] == 0) && (web1 == 1))
                                ? wdat1:current_val[18];

assign next_value[19] = (wadd2_dec[19] == 0) ? 0:
                ((wadd1_dec[19] == 0) && (web1 == 1))
                                ? wdat1:current_val[19];

assign next_value[20] = (wadd2_dec[20] == 0) ? 0:
                ((wadd1_dec[20] == 0) && (web1 == 1))
                                ? wdat1:current_val[20];

assign next_value[21] = (wadd2_dec[21] == 0) ? 0:
                ((wadd1_dec[21] == 0) && (web1 == 1))
                                ? wdat1:current_val[21];

assign next_value[22] = (wadd2_dec[22] == 0) ? 0:
                ((wadd1_dec[22] == 0) && (web1 == 1))
                                ? wdat1:current_val[22];

assign next_value[23] = (wadd2_dec[23] == 0) ? 0:
                ((wadd1_dec[23] == 0) && (web1 == 1))
                                ? wdat1:current_val[23];

assign next_value[24] = (wadd2_dec[24] == 0) ? 0:
                ((wadd1_dec[24] == 0) && (web1 == 1))
                                ? wdat1:current_val[24];

assign next_value[25] = (wadd2_dec[25] == 0) ? 0:
```

```
                    ((wadd1_dec[25] == 0) && (web1 == 1))
                                        ? wdat1:current_val[25];

assign next_value[26] = (wadd2_dec[26] == 0) ? 0:
                    ((wadd1_dec[26] == 0) && (web1 == 1))
                                        ? wdat1:current_val[26];

assign next_value[27] = (wadd2_dec[27] == 0) ? 0:
                    ((wadd1_dec[27] == 0) && (web1 == 1))
                                        ? wdat1:current_val[27];

assign next_value[28] = (wadd2_dec[28] == 0) ? 0:
                    ((wadd1_dec[28] == 0) && (web1 == 1))
                                        ? wdat1:current_val[28];

assign next_value[29] = (wadd2_dec[29] == 0) ? 0:
                    ((wadd1_dec[29] == 0) && (web1 == 1))
                                        ? wdat1:current_val[29];

assign next_value[30] = (wadd2_dec[30] == 0) ? 0:
                    ((wadd1_dec[30] == 0) && (web1 == 1))
                                        ? wdat1:current_val[30];

assign next_value[31] = (wadd2_dec[31] == 0) ? 0:
                    ((wadd1_dec[31] == 0) && (web1 == 1))
                                        ? wdat1:current_val[31];

endmodule

module RAM_3portclr(rport1, rport2, port3, wport1, wport2,
                                        wdat1, wdat2,
                                        web1, web2, web3, dp_preg1, dp_preg2, proc_out,
                                        clr1, clr2, clr3, clk, bootmode, current_value);

/* This RAM has 3 ports.  In addition, there are three clear signals */

input [3:0] rport1, rport2, port3, wport1, wport2;
input wdat1, wdat2, web1, web2, web3, clr1, clr2, clr3;
input clk, bootmode;

output dp_preg1, dp_preg2, proc_out;
output [15:0] current_value;

wire [15:0] radd1, radd2, add3, wadd1, wadd2, next_value;
wire [15:0] finnext_value;
wire pre_dp_preg1, pre_dp_preg2, pre_proc_out;

mydecode_4 dec1 (wport1, wadd1);
mydecode_4 dec2 (wport2, wadd2);
mydecode_4 dec3 (port3, add3);
mydecode_4 dec4 (rport1, radd1);
mydecode_4 dec5 (rport2, radd2);
```

```
assign finnext_value = (bootmode == 0) ? 16'h0000:next_value;
myreg #(16) pregv(finnext_value, current_value, clk);

assign pre_dp_preg1 = current_value[rport1];
assign pre_dp_preg2 = current_value[rport2];
assign pre_proc_out = current_value[port3];

assign dp_preg1 = ((rport2 == rport1) && (clr2 == 0)) ? 0:
                                    ((port3 == rport1) && (clr3 == 0)) ? 0:
                                    ((wport1 == rport1) && (web1 == 0)) ? wdat1:
                                    ((wport2 == rport1) && (web2 == 0)) ? wdat2:
                                    ((port3 == rport1) && (web3 == 0)) ? 1'b1:pre_dp_preg1;

assign dp_preg2 = ((rport1 == rport2) && (clr1 == 0)) ? 0:
                                    ((port3 == rport2) && (clr3 == 0)) ? 0:
                                    ((wport2 == rport2) && (web2 == 0)) ? wdat2:
                                    ((wport1 == rport2) && (web1 == 0)) ? wdat1:
                                    ((port3 == rport2) && (web3 == 0)) ? 1'b1:pre_dp_preg2;

assign proc_out = ((rport1 == port3) && (clr1 == 0)) ? 0:
                                    ((rport2 == port3) && (clr2 == 0)) ? 0:
                                    ((wport1 == port3) && (web1 == 0)) ? wdat1:
                                    ((wport2 == port3) && (web2 == 0)) ? wdat2:pre_proc_out;

assign next_value[0] = (((radd1[0] == 0) && (clr1 == 0)) ||
               ((radd2[0] == 0) && (clr2 == 0)) ||
               ((add3[0] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[0] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[0] == 0) && (web2 == 0)) ? wdat2:
              ((add3[0] == 0) && (web3 == 0)) ? 1'b1:
              current_value[0];

assign next_value[1] = (((radd1[1] == 0) && (clr1 == 0)) ||
               ((radd2[1] == 0) && (clr2 == 0)) ||
               ((add3[1] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[1] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[1] == 0) && (web2 == 0)) ? wdat2:
              ((add3[1] == 0) && (web3 == 0)) ? 1'b1:
              current_value[1];

assign next_value[2] = (((radd1[2] == 0) && (clr1 == 0)) ||
               ((radd2[2] == 0) && (clr2 == 0)) ||
               ((add3[2] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[2] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[2] == 0) && (web2 == 0)) ? wdat2:
              ((add3[2] == 0) && (web3 == 0)) ? 1'b1:
              current_value[2];

assign next_value[3] = (((radd1[3] == 0) && (clr1 == 0)) ||
               ((radd2[3] == 0) && (clr2 == 0)) ||
               ((add3[3] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[3] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[3] == 0) && (web2 == 0)) ? wdat2:
              ((add3[3] == 0) && (web3 == 0)) ? 1'b1:
              current_value[3];

assign next_value[4] = (((radd1[4] == 0) && (clr1 == 0)) ||
               ((radd2[4] == 0) && (clr2 == 0)) ||
               ((add3[4] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[4] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[4] == 0) && (web2 == 0)) ? wdat2:
              ((add3[4] == 0) && (web3 == 0)) ? 1'b1:
              current_value[4];

assign next_value[5] = (((radd1[5] == 0) && (clr1 == 0)) ||
               ((radd2[5] == 0) && (clr2 == 0)) ||
               ((add3[5] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[5] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[5] == 0) && (web2 == 0)) ? wdat2:
              ((add3[5] == 0) && (web3 == 0)) ? 1'b1:
              current_value[5];

assign next_value[6] = (((radd1[6] == 0) && (clr1 == 0)) ||
               ((radd2[6] == 0) && (clr2 == 0)) ||
               ((add3[6] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[6] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[6] == 0) && (web2 == 0)) ? wdat2:
              ((add3[6] == 0) && (web3 == 0)) ? 1'b1:
              current_value[6];

assign next_value[7] = (((radd1[7] == 0) && (clr1 == 0)) ||
               ((radd2[7] == 0) && (clr2 == 0)) ||
               ((add3[7] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[7] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[7] == 0) && (web2 == 0)) ? wdat2:
              ((add3[7] == 0) && (web3 == 0)) ? 1'b1:
              current_value[7];

assign next_value[8] = (((radd1[8] == 0) && (clr1 == 0)) ||
               ((radd2[8] == 0) && (clr2 == 0)) ||
               ((add3[8] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[8] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[8] == 0) && (web2 == 0)) ? wdat2:
              ((add3[8] == 0) && (web3 == 0)) ? 1'b1:
              current_value[8];

assign next_value[9] = (((radd1[9] == 0) && (clr1 == 0)) ||
               ((radd2[9] == 0) && (clr2 == 0)) ||
               ((add3[9] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[9] == 0) && (web1 == 0)) ? wdat1:
              ((wadd2[9] == 0) && (web2 == 0)) ? wdat2:
              ((add3[9] == 0) && (web3 == 0)) ? 1'b1:
              current_value[9];

assign next_value[10] = (((radd1[10] == 0) && (clr1 == 0)) ||
               ((radd2[10] == 0) && (clr2 == 0)) ||
               ((add3[10] == 0) && (clr3 == 0))) ? 0:
              ((wadd1[10] == 0) && (web1 == 0)) ? wdat1:
```

```
                    ((wadd2[10] == 0) && (web2 == 0)) ? wdat2:
                    ((add3[10] == 0) && (web3 == 0)) ? 1'b1:
                    current_value[10];

assign next_value[11] = (((radd1[11] == 0) && (clr1 == 0)) ||
                     ((radd2[11] == 0) && (clr2 == 0)) ||
                     ((add3[11] == 0) && (clr3 == 0))) ? 0:
                    ((wadd1[11] == 0) && (web1 == 0)) ? wdat1:
                    ((wadd2[11] == 0) && (web2 == 0)) ? wdat2:
                    ((add3[11] == 0) && (web3 == 0)) ? 1'b1:
                    current_value[11];

assign next_value[12] = (((radd1[12] == 0) && (clr1 == 0)) ||
                     ((radd2[12] == 0) && (clr2 == 0)) ||
                     ((add3[12] == 0) && (clr3 == 0))) ? 0:
                    ((wadd1[12] == 0) && (web1 == 0)) ? wdat1:
                    ((wadd2[12] == 0) && (web2 == 0)) ? wdat2:
                    ((add3[12] == 0) && (web3 == 0)) ? 1'b1:
                    current_value[12];

assign next_value[13] = (((radd1[13] == 0) && (clr1 == 0)) ||
                     ((radd2[13] == 0) && (clr2 == 0)) ||
                     ((add3[13] == 0) && (clr3 == 0))) ? 0:
                    ((wadd1[13] == 0) && (web1 == 0)) ? wdat1:
                    ((wadd2[13] == 0) && (web2 == 0)) ? wdat2:
                    ((add3[13] == 0) && (web3 == 0)) ? 1'b1:
                    current_value[13];

assign next_value[14] = (((radd1[14] == 0) && (clr1 == 0)) ||
                     ((radd2[14] == 0) && (clr2 == 0)) ||
                     ((add3[14] == 0) && (clr3 == 0))) ? 0:
                    ((wadd1[14] == 0) && (web1 == 0)) ? wdat1:
                    ((wadd2[14] == 0) && (web2 == 0)) ? wdat2:
                    ((add3[14] == 0) && (web3 == 0)) ? 1'b1:
                    current_value[14];

assign next_value[15] = (((radd1[15] == 0) && (clr1 == 0)) ||
                     ((radd2[15] == 0) && (clr2 == 0)) ||
                     ((add3[15] == 0) && (clr3 == 0))) ? 0:
                    ((wadd1[15] == 0) && (web1 == 0)) ? wdat1:
                    ((wadd2[15] == 0) && (web2 == 0)) ? wdat2:
                    ((add3[15] == 0) && (web3 == 0)) ? 1'b1:
                    current_value[15];

endmodule

module proc_biport(proc_AB, proc_AEN, proc_out, proc_DB,
                                   proc_S_REPLY, proc_P_REPLY, proc_LO_ADDR,
                                   proc_INT_L, proc_addr_out, proc_web_ram, proc_en_ram,
                                   proc_data_ram, clk, preg_valid_bits, bootmode);

input [4:0] proc_AB;
input proc_LO_ADDR, clk;
input proc_AEN, bootmode;
```

```verilog
input [32:0] proc_out;
input [1:0] proc_S_REPLY;
input [15:0] preg_valid_bits;

output [3:0] proc_addr_out;
output proc_web_ram, proc_en_ram, proc_INT_L;
output [31:0] proc_data_ram;
output [1:0] proc_P_REPLY;

inout [32:0] proc_DB;

wire [4:0] proc_addr_pre, proc_addr, next_proc_addr;
wire [4:0] reg_proc_addr, proc_addr_ram;
wire proc_read, proc_web, proc_addr_en_pre, proc_addr_en;
wire lo_addr_pre, lo_addr, val_read;
wire [31:0] proc_data_pre, proc_data, next_int_bits, int_bits;
wire [1:0] sreply_pre, sreply, preply, sreply2, preply_out;
wire [32:0] fin_proc_out_pre, fin_proc_out;
wire [15:0] input_int, output_int;
wire interrupt, proc_out_en_pre, proc_out_en;


assign val_read = (reg_proc_addr[4] == 1'b1) ? 1:0;

assign fin_proc_out_pre = (val_read == 1) ?
                                          {17'b00000000000000000, preg_valid_bits}:proc_out;
myreg #(33) prreg (fin_proc_out_pre, fin_proc_out, clk);

ipcndh padaddr0 (.Z(proc_AB[0]), .ZI(proc_addr_pre[0]));
ipcndh padaddr1 (.Z(proc_AB[1]), .ZI(proc_addr_pre[1]));
ipcndh padaddr2 (.Z(proc_AB[2]), .ZI(proc_addr_pre[2]));
ipcndh padaddr3 (.Z(proc_AB[3]), .ZI(proc_addr_pre[3]));
ipcndh padaddr4 (.Z(proc_AB[4]), .ZI(proc_addr_pre[4]));
ipcndh pad2 (.Z(proc_AEN), .ZI(proc_addr_en_pre));
ipcndh padsrp0 (.Z(proc_S_REPLY[0]), .ZI(sreply_pre[0]));
ipcndh padsrp1 (.Z(proc_S_REPLY[1]), .ZI(sreply_pre[1]));
ipcndh padlo (.Z(proc_LO_ADDR), .ZI(lo_addr_pre));

dfnnn ltad0 (.D(proc_addr_pre[0]), .Q(proc_addr[0]), .CPN(clk));
dfnnn ltad1 (.D(proc_addr_pre[1]), .Q(proc_addr[1]), .CPN(clk));
dfnnn ltad2 (.D(proc_addr_pre[2]), .Q(proc_addr[2]), .CPN(clk));
dfnnn ltad3 (.D(proc_addr_pre[3]), .Q(proc_addr[3]), .CPN(clk));
dfnnn ltad4 (.D(proc_addr_pre[4]), .Q(proc_addr[4]), .CPN(clk));

dfnnn ltaden (.D(proc_addr_en_pre), .Q(proc_addr_en), .CPN(clk));
dfnnn ltspl1 (.D(sreply_pre[1]), .Q(sreply[1]), .CPN(clk));
dfnnn ltspl0 (.D(sreply_pre[0]), .Q(sreply[0]), .CPN(clk));
dfnnn ltlo (.D(lo_addr_pre), .Q(lo_addr), .CPN(clk));

assign next_proc_addr = ((proc_addr_en == 1) && (lo_addr == 1))
                                                   ? proc_addr:reg_proc_addr;
myreg #(5) proc1 (next_proc_addr, reg_proc_addr, clk);
myreg #(5) proc2 (reg_proc_addr, proc_addr_ram, clk);
assign proc_addr_out = (sreply2 == 2'b11) ?
```

```
                                        reg_proc_addr[3:0]:proc_addr_ram[3:0];

assign proc_en_ram = ((sreply2 == 2'b11) && (val_read == 0) &&
                                    (bootmode == 1)) ? 0:1;

assign proc_web = ((sreply2 == 2'b10) && (val_read == 0) &&
                                    (bootmode == 1)) ? 0:1;

myreg web1 (proc_web, proc_web_ram, clk);

assign next_int_bits = (bootmode == 0) ? 32'h00000000:
                                                   ((preply == 2'b10) &&
                                                   (proc_addr_ram == 5'b11000)) ?
                                                     proc_data_ram:int_bits;
myreg #(32) int_reg (next_int_bits, int_bits, clk);

myand input_empty (~preg_valid_bits, int_bits[15:0], input_int);
myand output_full (preg_valid_bits, int_bits[31:16], output_int);

assign interrupt = (((input_int != 16'h0000) ||
                                     (output_int != 16'h0000)) && (bootmode == 1)) ? 0:1;

ostc06hh padint0 (.D(interrupt), .Z(proc_INT_L));

myreg #(2) sbitbuf (sreply, sreply2, clk);
myreg #(2) pbitbuf (sreply2, preply, clk);

assign preply_out = (preply == 2'b11) ? 2'b11:
                                    (preply == 2'b10) ? 2'b10:2'b00;


ostc06hh padp0 (.D(preply_out[0]), .Z(proc_P_REPLY[0]));
ostc06hh padp1 (.D(preply_out[1]), .Z(proc_P_REPLY[1]));

myreg #(32) pdata (proc_data, proc_data_ram, clk);
assign proc_out_en_pre = ((sreply2 == 2'b11) && (bootmode == 1'b1)) ? 1:0;
myreg outen (proc_out_en_pre, proc_out_en, clk);

bstcnd12hh padbit0 (.D(fin_proc_out[0]), .OE(proc_out_en), .Z(proc_DB[0]),
.ZI(proc_data_pre[0]));
bstcnd12hh padbit1 (.D(fin_proc_out[1]), .OE(proc_out_en), .Z(proc_DB[1]),
.ZI(proc_data_pre[1]));
bstcnd12hh padbit2 (.D(fin_proc_out[2]), .OE(proc_out_en), .Z(proc_DB[2]),
.ZI(proc_data_pre[2]));
bstcnd12hh padbit3 (.D(fin_proc_out[3]), .OE(proc_out_en), .Z(proc_DB[3]),
.ZI(proc_data_pre[3]));
bstcnd12hh padbit4 (.D(fin_proc_out[4]), .OE(proc_out_en), .Z(proc_DB[4]),
.ZI(proc_data_pre[4]));
bstcnd12hh padbit5 (.D(fin_proc_out[5]), .OE(proc_out_en), .Z(proc_DB[5]),
.ZI(proc_data_pre[5]));
bstcnd12hh padbit6 (.D(fin_proc_out[6]), .OE(proc_out_en), .Z(proc_DB[6]),
.ZI(proc_data_pre[6]));
bstcnd12hh padbit7 (.D(fin_proc_out[7]), .OE(proc_out_en), .Z(proc_DB[7]),
.ZI(proc_data_pre[7]));
```

```
bstcnd12hh padbit8 (.D(fin_proc_out[8]), .OE(proc_out_en), .Z(proc_DB[8]),
.ZI(proc_data_pre[8]));
bstcnd12hh padbit9 (.D(fin_proc_out[9]), .OE(proc_out_en), .Z(proc_DB[9]),
.ZI(proc_data_pre[9]));
bstcnd12hh padbit10 (.D(fin_proc_out[10]), .OE(proc_out_en), .Z(proc_DB[10]),
.ZI(proc_data_pre[10]));
bstcnd12hh padbit11 (.D(fin_proc_out[11]), .OE(proc_out_en), .Z(proc_DB[11]),
.ZI(proc_data_pre[11]));
bstcnd12hh padbit12 (.D(fin_proc_out[12]), .OE(proc_out_en), .Z(proc_DB[12]),
.ZI(proc_data_pre[12]));
bstcnd12hh padbit13 (.D(fin_proc_out[13]), .OE(proc_out_en), .Z(proc_DB[13]),
.ZI(proc_data_pre[13]));
bstcnd12hh padbit14 (.D(fin_proc_out[14]), .OE(proc_out_en), .Z(proc_DB[14]),
.ZI(proc_data_pre[14]));
bstcnd12hh padbit15 (.D(fin_proc_out[15]), .OE(proc_out_en), .Z(proc_DB[15]),
.ZI(proc_data_pre[15]));
bstcnd12hh padbit16 (.D(fin_proc_out[16]), .OE(proc_out_en), .Z(proc_DB[16]),
.ZI(proc_data_pre[16]));
bstcnd12hh padbit17 (.D(fin_proc_out[17]), .OE(proc_out_en), .Z(proc_DB[17]),
.ZI(proc_data_pre[17]));
bstcnd12hh padbit18 (.D(fin_proc_out[18]), .OE(proc_out_en), .Z(proc_DB[18]),
.ZI(proc_data_pre[18]));
bstcnd12hh padbit19 (.D(fin_proc_out[19]), .OE(proc_out_en), .Z(proc_DB[19]),
.ZI(proc_data_pre[19]));
bstcnd12hh padbit20 (.D(fin_proc_out[20]), .OE(proc_out_en), .Z(proc_DB[20]),
.ZI(proc_data_pre[20]));
bstcnd12hh padbit21 (.D(fin_proc_out[21]), .OE(proc_out_en), .Z(proc_DB[21]),
.ZI(proc_data_pre[21]));
bstcnd12hh padbit22 (.D(fin_proc_out[22]), .OE(proc_out_en), .Z(proc_DB[22]),
.ZI(proc_data_pre[22]));
bstcnd12hh padbit23 (.D(fin_proc_out[23]), .OE(proc_out_en), .Z(proc_DB[23]),
.ZI(proc_data_pre[23]));
bstcnd12hh padbit24 (.D(fin_proc_out[24]), .OE(proc_out_en), .Z(proc_DB[24]),
.ZI(proc_data_pre[24]));
bstcnd12hh padbit25 (.D(fin_proc_out[25]), .OE(proc_out_en), .Z(proc_DB[25]),
.ZI(proc_data_pre[25]));
bstcnd12hh padbit26 (.D(fin_proc_out[26]), .OE(proc_out_en), .Z(proc_DB[26]),
.ZI(proc_data_pre[26]));
bstcnd12hh padbit27 (.D(fin_proc_out[27]), .OE(proc_out_en), .Z(proc_DB[27]),
.ZI(proc_data_pre[27]));
bstcnd12hh padbit28 (.D(fin_proc_out[28]), .OE(proc_out_en), .Z(proc_DB[28]),
.ZI(proc_data_pre[28]));
bstcnd12hh padbit29 (.D(fin_proc_out[29]), .OE(proc_out_en), .Z(proc_DB[29]),
.ZI(proc_data_pre[29]));
bstcnd12hh padbit30 (.D(fin_proc_out[30]), .OE(proc_out_en), .Z(proc_DB[30]),
.ZI(proc_data_pre[30]));
bstcnd12hh padbit31 (.D(fin_proc_out[31]), .OE(proc_out_en), .Z(proc_DB[31]),
.ZI(proc_data_pre[31]));
ossc12hh padbit32 (.D(fin_proc_out[32]), .OE(proc_out_en), .Z(proc_DB[32]));


dfnnn ltb0 (.D(proc_data_pre[0]), .Q(proc_data[0]), .CPN(clk));
dfnnn ltb1 (.D(proc_data_pre[1]), .Q(proc_data[1]), .CPN(clk));
dfnnn ltb2 (.D(proc_data_pre[2]), .Q(proc_data[2]), .CPN(clk));
dfnnn ltb3 (.D(proc_data_pre[3]), .Q(proc_data[3]), .CPN(clk));
dfnnn ltb4 (.D(proc_data_pre[4]), .Q(proc_data[4]), .CPN(clk));
dfnnn ltb5 (.D(proc_data_pre[5]), .Q(proc_data[5]), .CPN(clk));
dfnnn ltb6 (.D(proc_data_pre[6]), .Q(proc_data[6]), .CPN(clk));
dfnnn ltb7 (.D(proc_data_pre[7]), .Q(proc_data[7]), .CPN(clk));
dfnnn ltb8 (.D(proc_data_pre[8]), .Q(proc_data[8]), .CPN(clk));
dfnnn ltb9 (.D(proc_data_pre[9]), .Q(proc_data[9]), .CPN(clk));
dfnnn ltb10 (.D(proc_data_pre[10]), .Q(proc_data[10]), .CPN(clk));
dfnnn ltb11 (.D(proc_data_pre[11]), .Q(proc_data[11]), .CPN(clk));
dfnnn ltb12 (.D(proc_data_pre[12]), .Q(proc_data[12]), .CPN(clk));
dfnnn ltb13 (.D(proc_data_pre[13]), .Q(proc_data[13]), .CPN(clk));
dfnnn ltb14 (.D(proc_data_pre[14]), .Q(proc_data[14]), .CPN(clk));
dfnnn ltb15 (.D(proc_data_pre[15]), .Q(proc_data[15]), .CPN(clk));
dfnnn ltb16 (.D(proc_data_pre[16]), .Q(proc_data[16]), .CPN(clk));
dfnnn ltb17 (.D(proc_data_pre[17]), .Q(proc_data[17]), .CPN(clk));
dfnnn ltb18 (.D(proc_data_pre[18]), .Q(proc_data[18]), .CPN(clk));
dfnnn ltb19 (.D(proc_data_pre[19]), .Q(proc_data[19]), .CPN(clk));
dfnnn ltb20 (.D(proc_data_pre[20]), .Q(proc_data[20]), .CPN(clk));
dfnnn ltb21 (.D(proc_data_pre[21]), .Q(proc_data[21]), .CPN(clk));
dfnnn ltb22 (.D(proc_data_pre[22]), .Q(proc_data[22]), .CPN(clk));
dfnnn ltb23 (.D(proc_data_pre[23]), .Q(proc_data[23]), .CPN(clk));
dfnnn ltb24 (.D(proc_data_pre[24]), .Q(proc_data[24]), .CPN(clk));
dfnnn ltb25 (.D(proc_data_pre[25]), .Q(proc_data[25]), .CPN(clk));
dfnnn ltb26 (.D(proc_data_pre[26]), .Q(proc_data[26]), .CPN(clk));
dfnnn ltb27 (.D(proc_data_pre[27]), .Q(proc_data[27]), .CPN(clk));
dfnnn ltb28 (.D(proc_data_pre[28]), .Q(proc_data[28]), .CPN(clk));
dfnnn ltb29 (.D(proc_data_pre[29]), .Q(proc_data[29]), .CPN(clk));
dfnnn ltb30 (.D(proc_data_pre[30]), .Q(proc_data[30]), .CPN(clk));
dfnnn ltb31 (.D(proc_data_pre[31]), .Q(proc_data[31]), .CPN(clk));

endmodule

/*
module sim_ram128x17(write_addr, read_addr, write_data, read_port, web, clk);

input [6:0] write_addr, read_addr;
input [16:0] write_data;
input clk, web;
output [16:0] read_port;
reg [16:0] mem[127:0];
reg [6:0] read_addr_fin;

assign read_port = mem[read_addr_fin];

always @(posedge clk)
                    read_addr_fin = read_addr;

always @(posedge clk)
    if( web === 1)
      mem[write_addr] = write_data;

endmodule

module sim_ram64x16(write_addr, read_addr, write_data, read_port, web, clk);
```

```
input [5:0] write_addr, read_addr;
input [15:0] write_data;
input clk, web;
output [15:0] read_port;
reg [15:0] mem[63:0];
reg [5:0] read_addr_fin;

assign read_port = mem[read_addr_fin];

always @(posedge clk)
                    read_addr_fin = read_addr;

always @(posedge clk)
    if( web === 1)
      mem[write_addr] = write_data;

endmodule

module sim_ram32x14(write_addr, read_addr, write_data, read_port, web, clk);

input [4:0] write_addr, read_addr;
input [13:0] write_data;
input clk,web;
output [13:0] read_port;
reg [13:0] mem[31:0];
reg [5:0] read_addr_fin;

assign read_port = mem[read_addr_fin];

always @(posedge clk)
                    read_addr_fin = read_addr;

always @(posedge clk)
    if( web === 1)
      mem[write_addr] = write_data;


endmodule

module sim_ram32x32(write_addr, read_addr, write_data, read_port, web, clk);

input [4:0] write_addr, read_addr;
input [31:0] write_data;
input clk, web;
output [31:0] read_port;
reg [31:0] mem[31:0];
reg [5:0] read_addr_fin;

wire [31:0] write_data_fin;
wire [4:0] read_addr_fin_pre, write_addr_fin;
wire web_fin;
```

```
assign #10 write_addr_fin = write_addr;
assign #10 read_addr_fin_pre =  read_addr;
assign #10 write_data_fin =  write_data;
assign #10 web_fin =  web;


assign read_port = mem[read_addr_fin];

always @(posedge clk)
                        read_addr_fin = read_addr_fin_pre;

always @(posedge clk)
     if( web_fin === 1)
        mem[write_addr_fin] = write_data_fin;

endmodule
*/

module my_ram16x32(rwaddr1, wdat1, rport1, rwaddr2, wdat2, rport2,
              rwaddr3, wdat3, rport3,  web1, web2, web3, clk);

input [3:0] rwaddr1, rwaddr2, rwaddr3;
input [31:0] wdat1, wdat2, wdat3;
input clk, web1, web2, web3;
output [31:0] rport1, rport2, rport3;

wire [31:0] pre_rport1, pre_rport2, pre_rport3;
wire [31:0] reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7, reg8,
                           reg9, reg10, reg11, reg12, reg13, reg14, reg15;

wire [31:0] nextreg0, nextreg1, nextreg2, nextreg3, nextreg4,
                           nextreg5, nextreg6, nextreg7, nextreg8, nextreg9,
                           nextreg10, nextreg11, nextreg12, nextreg13, nextreg14,
                           nextreg15;

wire [15:0] add1, add2, add3;

assign rport1 = ((rwaddr2 == rwaddr1) && (web2 == 0)) ? wdat2:
                                                ((rwaddr3 == rwaddr1) && (web3 == 0)) ? wdat3:pre_rport1;

assign rport2 = ((rwaddr1 == rwaddr2) && (web1 == 0)) ? wdat1:
                                                ((rwaddr3 == rwaddr2) && (web3 == 0)) ? wdat3:pre_rport2;

assign rport3 = ((rwaddr1 == rwaddr3) && (web1 == 0)) ? wdat1:
                                                ((rwaddr2 == rwaddr3) && (web2 == 0)) ? wdat2:pre_rport3;

assign nextreg0 = ((add1[0] == 0) && (web1 == 0)) ? wdat1:
                                                 ((add2[0] == 0) && (web2 == 0)) ? wdat2:
                                                 ((add3[0] == 0) && (web3 == 0)) ? wdat3:reg0;

assign nextreg1 = ((add1[1] == 0) && (web1 == 0)) ? wdat1:
                ((add2[1] == 0) && (web2 == 0)) ? wdat2:
                ((add3[1] == 0) && (web3 == 0)) ? wdat3:reg1;
```

```
assign nextreg2 = ((add1[2] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[2] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[2] == 0) && (web3 == 0)) ? wdat3:reg2;

assign nextreg3 = ((add1[3] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[3] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[3] == 0) && (web3 == 0)) ? wdat3:reg3;

assign nextreg4 = ((add1[4] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[4] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[4] == 0) && (web3 == 0)) ? wdat3:reg4;

assign nextreg5 = ((add1[5] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[5] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[5] == 0) && (web3 == 0)) ? wdat3:reg5;

assign nextreg6 = ((add1[6] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[6] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[6] == 0) && (web3 == 0)) ? wdat3:reg6;

assign nextreg7 = ((add1[7] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[7] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[7] == 0) && (web3 == 0)) ? wdat3:reg7;

assign nextreg8 = ((add1[8] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[8] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[8] == 0) && (web3 == 0)) ? wdat3:reg8;

assign nextreg9 = ((add1[9] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[9] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[9] == 0) && (web3 == 0)) ? wdat3:reg9;

assign nextreg10 = ((add1[10] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[10] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[10] == 0) && (web3 == 0)) ? wdat3:reg10;

assign nextreg11 = ((add1[11] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[11] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[11] == 0) && (web3 == 0)) ? wdat3:reg11;

assign nextreg12 = ((add1[12] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[12] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[12] == 0) && (web3 == 0)) ? wdat3:reg12;

assign nextreg13 = ((add1[13] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[13] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[13] == 0) && (web3 == 0)) ? wdat3:reg13;

assign nextreg14 = ((add1[14] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[14] == 0) && (web2 == 0)) ? wdat2:
                  ((add3[14] == 0) && (web3 == 0)) ? wdat3:reg14;

assign nextreg15 = ((add1[15] == 0) && (web1 == 0)) ? wdat1:
                  ((add2[15] == 0) && (web2 == 0)) ? wdat2:
```

```
           ((add3[15] == 0) && (web3 == 0)) ? wdat3:reg15;


mydecode_4 dec1 (rwaddr1, add1);
mydecode_4 dec2 (rwaddr2, add2);
mydecode_4 dec3 (rwaddr3, add3);

myreg #(32) p0(nextreg0, reg0, clk);
myreg #(32) p1(nextreg1, reg1, clk);
myreg #(32) p2(nextreg2, reg2, clk);
myreg #(32) p3(nextreg3, reg3, clk);
myreg #(32) p4(nextreg4, reg4, clk);
myreg #(32) p5(nextreg5, reg5, clk);
myreg #(32) p6(nextreg6, reg6, clk);
myreg #(32) p7(nextreg7, reg7, clk);
myreg #(32) p8(nextreg8, reg8, clk);
myreg #(32) p9(nextreg9, reg9, clk);
myreg #(32) p10(nextreg10, reg10, clk);
myreg #(32) p11(nextreg11, reg11, clk);
myreg #(32) p12(nextreg12, reg12, clk);
myreg #(32) p13(nextreg13, reg13, clk);
myreg #(32) p14(nextreg14, reg14, clk);
myreg #(32) p15(nextreg15, reg15, clk);

wire [15:0] temp0 = {reg15[0], reg14[0], reg13[0], reg12[0], reg11[0],
          reg10[0], reg9[0], reg8[0], reg7[0], reg6[0], reg5[0],
          reg4[0], reg3[0], reg2[0], reg1[0], reg0[0]};

wire [15:0] temp1 = {reg15[1], reg14[1], reg13[1], reg12[1], reg11[1],
          reg10[1], reg9[1], reg8[1], reg7[1], reg6[1], reg5[1],
          reg4[1], reg3[1], reg2[1], reg1[1], reg0[1]};

wire [15:0] temp2 = {reg15[2], reg14[2], reg13[2], reg12[2], reg11[2],
          reg10[2], reg9[2], reg8[2], reg7[2], reg6[2], reg5[2],
          reg4[2], reg3[2], reg2[2], reg1[2], reg0[2]};

wire [15:0] temp3 = {reg15[3], reg14[3], reg13[3], reg12[3], reg11[3],
          reg10[3], reg9[3], reg8[3], reg7[3], reg6[3], reg5[3],
          reg4[3], reg3[3], reg2[3], reg1[3], reg0[3]};

wire [15:0] temp4 = {reg15[4], reg14[4], reg13[4], reg12[4], reg11[4],
          reg10[4], reg9[4], reg8[4], reg7[4], reg6[4], reg5[4],
          reg4[4], reg3[4], reg2[4], reg1[4], reg0[4]};

wire [15:0] temp5 = {reg15[5], reg14[5], reg13[5], reg12[5], reg11[5],
          reg10[5], reg9[5], reg8[5], reg7[5], reg6[5], reg5[5],
          reg4[5], reg3[5], reg2[5], reg1[5], reg0[5]};

wire [15:0] temp6 = {reg15[6], reg14[6], reg13[6], reg12[6], reg11[6],
          reg10[6], reg9[6], reg8[6], reg7[6], reg6[6], reg5[6],
          reg4[6], reg3[6], reg2[6], reg1[6], reg0[6]};

wire [15:0] temp7 = {reg15[7], reg14[7], reg13[7], reg12[7], reg11[7],
          reg10[7], reg9[7], reg8[7], reg7[7], reg6[7], reg5[7],
                reg4[7], reg3[7], reg2[7], reg1[7], reg0[7]};

wire [15:0] temp8 = {reg15[8], reg14[8], reg13[8], reg12[8], reg11[8],
                reg10[8], reg9[8], reg8[8], reg7[8], reg6[8], reg5[8],
                reg4[8], reg3[8], reg2[8], reg1[8], reg0[8]};

wire [15:0] temp9 = {reg15[9], reg14[9], reg13[9], reg12[9], reg11[9],
                reg10[9], reg9[9], reg8[9], reg7[9], reg6[9], reg5[9],
                reg4[9], reg3[9], reg2[9], reg1[9], reg0[9]};

wire [15:0] temp10 ={reg15[10], reg14[10], reg13[10], reg12[10],
                              reg11[10],reg10[10], reg9[10], reg8[10], reg7[10],
                              reg6[10], reg5[10],reg4[10], reg3[10], reg2[10],
                              reg1[10],reg0[10]};

wire [15:0] temp11 = {reg15[11], reg14[11], reg13[11], reg12[11],
                            reg11[11], reg10[11], reg9[11], reg8[11], reg7[11],
                            reg6[11], reg5[11], reg4[11], reg3[11], reg2[11],
                            reg1[11], reg0[11]};

wire [15:0] temp12 = {reg15[12], reg14[12], reg13[12], reg12[12],
                            reg11[12], reg10[12], reg9[12], reg8[12], reg7[12],
                            reg6[12], reg5[12], reg4[12], reg3[12], reg2[12],
                            reg1[12], reg0[12]};

wire [15:0] temp13 = {reg15[13], reg14[13], reg13[13], reg12[13],
                            reg11[13], reg10[13], reg9[13], reg8[13], reg7[13],
                            reg6[13], reg5[13], reg4[13], reg3[13], reg2[13],
                            reg1[13], reg0[13]};

wire [15:0] temp14 = {reg15[14], reg14[14], reg13[14], reg12[14],
                            reg11[14], reg10[14], reg9[14], reg8[14], reg7[14],
                            reg6[14], reg5[14], reg4[14], reg3[14], reg2[14],
                            reg1[14], reg0[14]};

wire [15:0] temp15 = {reg15[15], reg14[15], reg13[15], reg12[15],
                            reg11[15], reg10[15], reg9[15], reg8[15], reg7[15],
                            reg6[15], reg5[15], reg4[15], reg3[15], reg2[15],
                            reg1[15], reg0[15]};

wire [15:0] temp16 = {reg15[16], reg14[16], reg13[16], reg12[16],
                            reg11[16], reg10[16], reg9[16], reg8[16], reg7[16],
                            reg6[16], reg5[16], reg4[16], reg3[16], reg2[16],
                            reg1[16], reg0[16]};

wire [15:0] temp17 = {reg15[17], reg14[17], reg13[17], reg12[17],
                            reg11[17], reg10[17], reg9[17], reg8[17], reg7[17],
                            reg6[17], reg5[17], reg4[17], reg3[17], reg2[17],
                            reg1[17], reg0[17]};

wire [15:0] temp18 = {reg15[18], reg14[18], reg13[18], reg12[18],
                            reg11[18], reg10[18], reg9[18], reg8[18], reg7[18],
                            reg6[18], reg5[18], reg4[18], reg3[18], reg2[18],
                            reg1[18], reg0[18]};
```

```
wire [15:0] temp19 = {reg15[19], reg14[19], reg13[19], reg12[19],
                      reg11[19], reg10[19], reg9[19], reg8[19], reg7[19],
                      reg6[19], reg5[19], reg4[19], reg3[19], reg2[19],
                      reg1[19], reg0[19]};

wire [15:0] temp20 = {reg15[20], reg14[20], reg13[20], reg12[20],
                      reg11[20], reg10[20], reg9[20], reg8[20], reg7[20],
                      reg6[20], reg5[20], reg4[20], reg3[20], reg2[20],
                      reg1[20], reg0[20]};

wire [15:0] temp21 = {reg15[21], reg14[21], reg13[21], reg12[21],
                      reg11[21], reg10[21], reg9[21], reg8[21], reg7[21],
                      reg6[21], reg5[21], reg4[21], reg3[21], reg2[21],
                      reg1[21], reg0[21]};

wire [15:0] temp22 = {reg15[22], reg14[22], reg13[22], reg12[22],
                      reg11[22], reg10[22], reg9[22], reg8[22], reg7[22],
                      reg6[22], reg5[22], reg4[22], reg3[22], reg2[22],
                      reg1[22], reg0[22]};

wire [15:0] temp23 = {reg15[23], reg14[23], reg13[23], reg12[23],
                      reg11[23], reg10[23], reg9[23], reg8[23], reg7[23],
                      reg6[23], reg5[23], reg4[23], reg3[23], reg2[23],
                      reg1[23], reg0[23]};

wire [15:0] temp24 = {reg15[24], reg14[24], reg13[24], reg12[24],
                      reg11[24], reg10[24], reg9[24], reg8[24], reg7[24],
                      reg6[24], reg5[24], reg4[24], reg3[24], reg2[24],
                      reg1[24], reg0[24]};

wire [15:0] temp25 = {reg15[25], reg14[25], reg13[25], reg12[25],
                      reg11[25], reg10[25], reg9[25], reg8[25], reg7[25],
                      reg6[25], reg5[25], reg4[25], reg3[25], reg2[25],
                      reg1[25], reg0[25]};

wire [15:0] temp26 = {reg15[26], reg14[26], reg13[26], reg12[26],
                      reg11[26], reg10[26], reg9[26], reg8[26], reg7[26],
                      reg6[26], reg5[26], reg4[26], reg3[26], reg2[26],
                      reg1[26], reg0[26]};

wire [15:0] temp27 = {reg15[27], reg14[27], reg13[27], reg12[27],
                      reg11[27], reg10[27], reg9[27], reg8[27], reg7[27],
                      reg6[27], reg5[27], reg4[27], reg3[27], reg2[27],
                      reg1[27], reg0[27]};

wire [15:0] temp28 = {reg15[28], reg14[28], reg13[28], reg12[28],
                      reg11[28], reg10[28], reg9[28], reg8[28], reg7[28],
                      reg6[28], reg5[28], reg4[28], reg3[28], reg2[28],
                      reg1[28], reg0[28]};

wire [15:0] temp29 = {reg15[29], reg14[29], reg13[29], reg12[29],
                      reg11[29], reg10[29], reg9[29], reg8[29], reg7[29],
                      reg6[29], reg5[29], reg4[29], reg3[29], reg2[29],
                        reg1[29], reg0[29]};

wire [15:0] temp30 = {reg15[30], reg14[30], reg13[30], reg12[30],
                        reg11[30], reg10[30], reg9[30], reg8[30], reg7[30],
                        reg6[30], reg5[30], reg4[30], reg3[30], reg2[30],
                        reg1[30], reg0[30]};

wire [15:0] temp31 = {reg15[31], reg14[31], reg13[31], reg12[31],
                        reg11[31], reg10[31], reg9[31], reg8[31], reg7[31],
                        reg6[31], reg5[31], reg4[31], reg3[31], reg2[31],
                        reg1[31], reg0[31]};


assign pre_rport1[0] = temp0[rwaddr1];
assign pre_rport1[1] = temp1[rwaddr1];
assign pre_rport1[2] = temp2[rwaddr1];
assign pre_rport1[3] = temp3[rwaddr1];
assign pre_rport1[4] = temp4[rwaddr1];
assign pre_rport1[5] = temp5[rwaddr1];
assign pre_rport1[6] = temp6[rwaddr1];
assign pre_rport1[7] = temp7[rwaddr1];
assign pre_rport1[8] = temp8[rwaddr1];
assign pre_rport1[9] = temp9[rwaddr1];
assign pre_rport1[10] = temp10[rwaddr1];
assign pre_rport1[11] = temp11[rwaddr1];
assign pre_rport1[12] = temp12[rwaddr1];
assign pre_rport1[13] = temp13[rwaddr1];
assign pre_rport1[14] = temp14[rwaddr1];
assign pre_rport1[15] = temp15[rwaddr1];
assign pre_rport1[16] = temp16[rwaddr1];
assign pre_rport1[17] = temp17[rwaddr1];
assign pre_rport1[18] = temp18[rwaddr1];
assign pre_rport1[19] = temp19[rwaddr1];
assign pre_rport1[20] = temp20[rwaddr1];
assign pre_rport1[21] = temp21[rwaddr1];
assign pre_rport1[22] = temp22[rwaddr1];
assign pre_rport1[23] = temp23[rwaddr1];
assign pre_rport1[24] = temp24[rwaddr1];
assign pre_rport1[25] = temp25[rwaddr1];
assign pre_rport1[26] = temp26[rwaddr1];
assign pre_rport1[27] = temp27[rwaddr1];
assign pre_rport1[28] = temp28[rwaddr1];
assign pre_rport1[29] = temp29[rwaddr1];
assign pre_rport1[30] = temp30[rwaddr1];
assign pre_rport1[31] = temp31[rwaddr1];

assign pre_rport2[0] = temp0[rwaddr2];
assign pre_rport2[1] = temp1[rwaddr2];
assign pre_rport2[2] = temp2[rwaddr2];
assign pre_rport2[3] = temp3[rwaddr2];
assign pre_rport2[4] = temp4[rwaddr2];
assign pre_rport2[5] = temp5[rwaddr2];
assign pre_rport2[6] = temp6[rwaddr2];
assign pre_rport2[7] = temp7[rwaddr2];
assign pre_rport2[8] = temp8[rwaddr2];
assign pre_rport2[9] = temp9[rwaddr2];
assign pre_rport2[10] = temp10[rwaddr2];
assign pre_rport2[11] = temp11[rwaddr2];
assign pre_rport2[12] = temp12[rwaddr2];
assign pre_rport2[13] = temp13[rwaddr2];
assign pre_rport2[14] = temp14[rwaddr2];
assign pre_rport2[15] = temp15[rwaddr2];
assign pre_rport2[16] = temp16[rwaddr2];
assign pre_rport2[17] = temp17[rwaddr2];
assign pre_rport2[18] = temp18[rwaddr2];
assign pre_rport2[19] = temp19[rwaddr2];
assign pre_rport2[20] = temp20[rwaddr2];
assign pre_rport2[21] = temp21[rwaddr2];
assign pre_rport2[22] = temp22[rwaddr2];
assign pre_rport2[23] = temp23[rwaddr2];
assign pre_rport2[24] = temp24[rwaddr2];
assign pre_rport2[25] = temp25[rwaddr2];
assign pre_rport2[26] = temp26[rwaddr2];
assign pre_rport2[27] = temp27[rwaddr2];
assign pre_rport2[28] = temp28[rwaddr2];
assign pre_rport2[29] = temp29[rwaddr2];
assign pre_rport2[30] = temp30[rwaddr2];
assign pre_rport2[31] = temp31[rwaddr2];

assign pre_rport3[0] = temp0[rwaddr3];
assign pre_rport3[1] = temp1[rwaddr3];
assign pre_rport3[2] = temp2[rwaddr3];
assign pre_rport3[3] = temp3[rwaddr3];
assign pre_rport3[4] = temp4[rwaddr3];
assign pre_rport3[5] = temp5[rwaddr3];
assign pre_rport3[6] = temp6[rwaddr3];
assign pre_rport3[7] = temp7[rwaddr3];
assign pre_rport3[8] = temp8[rwaddr3];
assign pre_rport3[9] = temp9[rwaddr3];
assign pre_rport3[10] = temp10[rwaddr3];
assign pre_rport3[11] = temp11[rwaddr3];
assign pre_rport3[12] = temp12[rwaddr3];
assign pre_rport3[13] = temp13[rwaddr3];
assign pre_rport3[14] = temp14[rwaddr3];
assign pre_rport3[15] = temp15[rwaddr3];
assign pre_rport3[16] = temp16[rwaddr3];
assign pre_rport3[17] = temp17[rwaddr3];
assign pre_rport3[18] = temp18[rwaddr3];
assign pre_rport3[19] = temp19[rwaddr3];
assign pre_rport3[20] = temp20[rwaddr3];
assign pre_rport3[21] = temp21[rwaddr3];
assign pre_rport3[22] = temp22[rwaddr3];
assign pre_rport3[23] = temp23[rwaddr3];
assign pre_rport3[24] = temp24[rwaddr3];
assign pre_rport3[25] = temp25[rwaddr3];
assign pre_rport3[26] = temp26[rwaddr3];
assign pre_rport3[27] = temp27[rwaddr3];
assign pre_rport3[28] = temp28[rwaddr3];
assign pre_rport3[29] = temp29[rwaddr3];
assign pre_rport3[30] = temp30[rwaddr3];
assign pre_rport3[31] = temp31[rwaddr3];


endmodule
```