THE DESIGN OF A MULTIPROCESSOR DEVELOPMENT SYSTEM

Thomas Lee Anderson

September 1, 1982

This research was supported by the Advanced Research Projects Agency of the Department fo Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge **Massachuetts 02139**

This page intentionally left blank!!

The Design of a Multiprocessor Development System

by

Thomas Lee Anderson

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 1982 in partial fulfillment of the
requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

A multiprocessor development system has been designed and a prototype system is being constructed. The system, known as Concert, is intended to support multiprocessor research efforts at M.I.T. The motivation for Concert and the project history are summarized briefly. Some intended applications are also identified.

The system incorporates the RingBus architecture, a novel scheme for interconnecting processors and memory in a tightly-coupled multiprocessor system. The architecture is described both in its general form and in the particular implementation used in the system. The results of some analysis and synthesis of the architecture are summarized.

The design of the Concert multiprocessor development system is described, with particular emphasis on the tradeoffs considered in the design process. The design of two particular hardware modules is discussed in considerable detail. Finally, some suggestions are offered for future use of the system and further investigation into the RingBus architecture.

Name and Title of Thesis Supervisor:

Robert H. Halstead, Jr.,
Assistant Professor of Computer Science and Engineering

Key Words and Phrases:

multiprocessor systems, segmented computer buses, parallel processing, computer architecture

Acknowledgements

There are numerous people whose assistance on the Concert project was instrumental in the completion of this thesis. The project leaders, Richard Zippel and Robert Halstead, provided constant guidance throughout all phases of the project. The software members of the team — Dave Alpern and John Morrison — have contributed in numerous ways to the project. Tom Sterling, who intends to use the system for his own thesis work, assisted with suggestions on several aspects of the hardware design, particularly hardware monitoring. His interest as a potential user has helped to insure that we stayed on the track of making the system a useful tool for the M.I.T. computing community.

The Real Time Systems Group, under the leadership of Steve Ward, has provided both a special place to work and a great deal of assistance. The hardware expertise of numerous members of the group has undoubtedly helped me to avert some major problems. The group also provided the computing facilities and text processing capabilities to produce this thesis. The support of Michael Dertouzos, Director of the Laboratory for Computer Science, has also been crucial. Without his assistance in obtaining funds for both students and hardware, it would have been impossible to construct a prototype system.

My officemate Jim Troisi deserves thanks for his assistance in spicing up my arbiter simulator and for his interest in the project and willingness to hear my complaints and problems. Finally, I owe a special debt to my thesis advisor, Bert Halstead. Bert's RingBus Architecture provided the basis for most of the interesting aspects of the system design, and his suggestions on the design itself were frequent and helpful. It is largely because of his encouragement and drive that I was able to complete this thesis.

TABLE OF CONTENTS

Chapter 1: Introduction

Several months ago, a research group was formed under the auspices of the Real Time Systems Group in the Laboratory for Computer Science at M.I.T. to develop a multiprocessor development system. The system, now known as "Concert," is intended as a tool to allow researchers at M.I.T. to experiment with multiprocessing ideas and concepts on a working multiprocessor system. A prototype version of this system has been designed and is now under construction.

This thesis documents the hardware design of the prototype system. It starts by providing background on the project and the motivation for Concert in Chapter 2. There has been considerable interest within the M.I.T. computing community in the use of the system for a variety of applications. Chapter 2 briefly discusses many of these applications, some of which will be undertaken within the next few months.

The design of the Concert multiprocessor development system is outlined from two angles. First, Chapter 3 describes the RingBus architecture, a scheme for interconnecting processors and global memory in a tightly-coupled multiprocessor system. Some effort was spent in analyzing and simulating this architecture, and the results of this work are summarized.

The remainder of the thesis describes the system that is being constructed. The system is interesting both in its own right — as a multiprocessing research vehicle for the M.I.T. computing community — and as the first hardware implementation of the RingBus architecture. Chapter 4 describes the implementation at the block-diagram

level and identifies the major components of the design effort.

The thesis project primarily involved the design of two hardware modules, and these are discussed in considerable detail in Chapters 5 and 6. Particular attention is paid to the tradeoffs considered in the design process and the reasons particular implementation decisions were made. The thesis is intended to document the system as it currently exists, but also to be an interesting case study in hardware design.

Finally, Chapter 7 concludes the thesis by evaluating the current state of the Concert project and the future usefulness of the system. Some suggestions are offered for future applications work on the multiprocessor development system as well as for more investigation into the RingBus architecture.

Chapter 2: Background on the Concert Project

This chapter describes the effort to build a multiprocessor development system at M.I.T. After establishing the research framework for the use of such a system, it gives a brief history of the project and lists some potential applications which guided the system specification.

2.1: Multiprocessor Research at M.I.T.

It is a truism of computer science that single processors are reaching the limits of their performance. Such fundamental physical constants as the speed of light place hard limits on the speed which a single computer can ever attain, and industry is fast approaching these limits on several fronts. Given this, a considerable portion of current research in computer science and engineering is devoted to the design and analysis of multiprocessor systems. There are several foci for this research, including multiprocessor programming models, distributed processing, and multiprocessor architectures.

Like many other research institutions, M.I.T. is actively engaged in a wide range of multiprocessing research. Several groups are studying architectures for multiprocessor systems; others are working on the software aspects of multiprocessing, including operating systems, parallel algorithms and fault-tolerance. Many more people are doing research not in multiprocessing *per se*, but rather in applications which are particularly

well-suited for implementation in a multiprocessing environment.

At the present time, most research of this nature is carried out by software simulation. The task of simulating parallel execution of multiple processors is not difficult in theory, except for a few thorny timing and synchronization issues. The problem is that software simulation is a painfully slow method for testing parallel programs. Even on fairly fast machines, the time necessary for simulation of realistic programs is tremendous.

The reason for this is simple. If a single processor computer is used to simulate a multiprocessor of many nodes, the simulator must sequentially execute tasks intended to be performed in parallel. The problem is compounded if the computer must be time-shared with other users, as is generally the case at M.I.T. However, if a working multiprocessor is available to these researchers, at least some of the parallelism in the application programs can be exploited in hardware rather than merely simulated. The multiprocessor development system was conceived to satisfy exactly this need.

The goal of the project has been to provide researchers with a readily-available multiprocessor system on which to work. The system will be particularly useful for groups who want to investigate the use of multiprocessors, but don't wish to spend a large amount of time on the construction of a system. For people investigating parallel algorithms or distributed software, the system will provide an actual multiprocessor on which to try out their ideas.

Concert also has some potential uses for research groups interested in multiprocessor architectures. They can use software to simulate their architecture or processor interconnection strategy while still exploiting the inherent parallelism of the multiproces-

sor. The system can effectively serve as a stepping stone for those groups who are interested in ultimately building their own hardware.

The motivation for the project is the dream of an easily-configured off-the-shelf multiprocessor system available to the M.I.T. computing community. The ultimate scenario is fairly simple. A standard processing node will be designed which can be connected with other nodes of like kind in an arbitrarily large multiprocessor system. A large supply of these nodes will be available from the supply room, already built, tested, and ready to be plugged together. Although the final design of the system will be reached by compromise among interested parties, it will have enough features to make it useful for a large number of engineers and computer scientists at M.I.T.

This brief description captures the essence of a multiprocessor development system — availability and ease of use. The intent is that people engaged in research involving multiprocessors will not have to spend time building and debugging their own systems. They can just grab fifty or a hundred processing nodes, connect them together in a configuration suitable for their particular needs, and begin playing. This need not be the ultimate system for them, but rather a "quick and dirty" way to check out theories, run benchmarks and experiment in general.

2.2: Project History

Discussion on the construction of a multiprocessor development system began in mid-1981 in the Real Time Systems Group. Rich Zippel organized a group to investigate the project, and by the end of the year the specification of the system was well under way. The term "multiprocessor development system" was coined as an analog

to so-called microprocessor development systems. After several earlier choices the name "Concert" was selected. It is intended to invoke an image of multiple processors performing independent tasks, but with all tasks aimed at solving a common problem.

The original proposal called for a network of independent processing nodes interconnected by dedicated serial lines. Two major problems with this approach — limited interprocessor communication speed and difficulty in down-line loading code — led to the choice of a shared-memory system instead. The first such system proposed was based on a hierarchical bus structure. The amount of hardware necessary to support this scheme was excessive, and the concern was voiced that bus contention would severely limit performance.

The crucial juncture in the project occurred early in 1982, when Bert Halstead suggested a circular segmented bus as the top-level interconnection scheme for the system. His approach was dubbed "the RingBus architecture," and has been incorporated into the Concert prototype. The next chapter describes the architecture in detail, but it is worth noting here that its attractiveness lies in its ability to support simultaneous accesses to global memory.

2.3: Suggested Applications

From its inception, Concert was intended to be a computing resource available to a wide range of people at M.I.T. Throughout the project, the interest and encouragement of researchers within both Real Time Systems and other groups has been a strong motivation for its completion. These researchers have proposed a wide variety of applications for Concert, some of which will be started in the near future. The ma-

jor applications which have been suggested include the following.

1) The MuNet - The MuNet is a proposed "myriaprocessor" system developed primarily by Bert Halstead [25-28]. He would like to use Concert as a testbed to try out various ideas about message-passing and communication among MuNet nodes.

2) Communications - During the early phases of the project, the Laboratory for Information and Decision Systems (LIDS) expressed some interest in using Concert to simulate communications networks. They would like to test out routing strategies and communications protocols, and to simulate both existing and proposed network configurations. This would allow them to vary parameters they could not touch on an actual operating network.

3) VLSI - Current VLSI circuit simulation and layout programs take a great deal of time even on large computers. A multiprocessor system which could take advantage of parallelism could theoretically speed up such programs by orders of magnitude. Rich Zippel would like to use Concert to bring up a circuit simulation program, perhaps a parallel version of SPICE.

4) Data Flow - Although Concert is not a data flow machine, it could serve as a testbed for programming and architectural concepts during the design of a true data flow multiprocessor [5-7,16,24]. Arvind believes that Concert would be useful in this respect, and has indicated some interest in using the system.

5) Parallel Control Flow - The control flow approach to multiprocessor programming has some similarities to data flow [19,54]. Tom Sterling is working on the specification of a dispatcher for a parallel control flow multiprocessor system. He will use his dispatcher to control a Concert system and operate it as a parallel control flow machine.

6) High-Performance Graphics - Bert Halstead will use Concert to implement a high-performance graphics system. He envisions a number of Concert nodes processing graphics commands and filling up a common bitmap in global memory. A custom graphics processor will be built to display the contents of the bitmap memory on a high-resolution monitor.

7) Music Synthesis - Bert Halstead also has some interest in the use of Concert for high-performance music synthesis. For example, if each Concert node is assigned a voice, the resulting music will be much more complex than a single processor could generate.

8) Multiprocessor LISP - Bert Halstead and Rich Zippel are planning to bring up a multiprocessor version of LISP to run on Concert. This will be a particularly interesting test of the system in a non-traditional programming environment.

9) Electrical Demand Simulation - Fred Schweppe and Jim Kirtley of the Electrical Power Systems Engineering Laboratory (EPSEL) are interested in using Concert to simulate electrical energy demand in transmission and distribution systems [34,48-49]. This particular application could prove a best-case test for Concert, since the inherent locality of the simulation algorithms produces low bus contention.

2.4: Design Goals

If Concert is to be useful for all the applications listed in the previous section, it must be flexible as well as easy to use. Specifically, it must support most applications without the need for hardware redesign. Exceptions will be made primarily for reasons of efficiency. For example, a specialized dispatcher will be designed for the parallel control flow machine because a standard Concert node would be too slow. Regardless, it would be possible for a standard node to simulate the control flow dispatcher in software if required.

If Concert is to be a useful tool for the M.I.T. computing community, it must be a reliable, robust system. There are two ways to approach the issue of reliability. The first is that the hardware be physically reliable. Experience with previous projects in the Real Time Systems Group has shown that it is difficult to produce reliable hardware in a university environment. For this reason, the decision was made that Concert

would use off-the-shelf boards and other technology whenever possible. If it achieves its goal of providing off-the-shelf multiprocessing, Concert hardware will be used repeatedly by different groups. It is important that it be able to survive multiple applications without becoming unreliable.

The other main aspect of reliability is software reliability. For the most part this lies outside the scope of this thesis. However, there are some aspects of the design which have been included to help support software robustness, and these are identified as they are encountered.

The basic Concert architecture is designed to support several dozen processing nodes, but the ultimate system will support hundreds. This implies that the hardware must be relatively inexpensive. For example, using fancy floating-point processors for each node would certainly produce a very powerful machine. However, it would be impractical for the scale of research envisioned for the system.

These considerations led to four major goals for the Concert multiprocessor development system: low cost, flexibility, reliability, and ease of use. These goals have permeated every aspect of the hardware (and software) design. The remainder of this thesis describes the attempt to meet these goals. It outlines the Concert system architecture and describes in considerable detail the specific design responsibilities for the thesis project.

Chapter 3: The RingBus Architecture

This chapter outlines the RingBus architecture, a scheme for interconnecting processors in a shared-memory multiprocessor system. It presents some background terminology and then describes the architecture in its general form. The specific implementation used in Concert is discussed in Chapter 4.

3.1: Dimensions of Multiprocessing

There is a wide variety of ways to look at "multiprocessing." In 1966, Flynn published a classification of computers [20] which is still in use today. He termed the common garden-variety computer a *Single Instruction stream, Single Data stream* (SISD) machine, because it consists of one control unit whose instructions control a single data path. The majority of computers today are still essentially SISD machines, although a slight degree of parallel processing is accomplished by separate I/O processors and the like.

The only class of computers other than SISD to be implemented widely has been *SIMD* (Single Instruction, Multiple Data), often called *parallel processors*. These machines contain a single instruction stream and control unit which manipulates multiple data paths. Some computers in this class are called *array processors*, [8,11,36,53] since the elements of an array may be fed in parallel into the data units. *Associative processors* [10,22,59] are also generally considered SIMD machines.

Flynn's *MISD* category, in which multiple control units manipulate a single data

path, has not turned out to be very useful. Pipelined processors [41,42] are sometimes considered MISD machines, although this is a controversial categorization. Generally, neither SIMD nor MISD machines are considered "true" multiprocessors.

The final, and most complex, class of machines is *MIMD*, in which multiple intelligent control units manipulate multiple data paths. The most common MIMD configuration is a collection of independent processors, each with its own instruction and data streams. These processors are interconnected in some fashion, so that they can cooperate in the solution of a single problem.

There are two primary subdivisions of MIMD machines. The processors in a *loosely-coupled* multiprocessor system communicate by sending messages over dedicated links or via a communications network. Such a configuration encourages a *message-passing* style of programming, in which subtasks execute in parallel on different virtual (and perhaps physical) machines and send commands back and forth. Loosely-coupled systems are sometimes called "multiple computer systems" rather than "multiprocessors" [17-18].

Tightly-coupled systems contains processors which communicate via shared memory, and are generally considered to be "true" multiprocessors. They encourage a programming style in which shared memory locations, often protected by semaphores, are used to pass information between processors and to control program flow.

There are a variety of techniques for interconnecting processors and memory in a tightly-coupled multiprocessor system. The simplest technology is a single shared bus. Tightly-coupled multiprocessors in which accesses to shared memory are made by means of a central system bus are often called *common-bus* systems. This term em-

phasizes the fact that the processors must contend for the bus if they wish to access shared memory. This contention places some inherent limitations on performance.

A variety of approaches has been tried to ease the contention problem. One partial solution is to give each processor its own *local* memory for instructions and private data. Shared, or *global*, memory is used only for shared data structures and for passing information between nodes. If the majority of a processor's accesses are to its own local memory, and this memory is available without contention, the speedup over a system with only global memory can be quite dramatic.

Even with generous local memories, the number of processors which may be placed on a single global bus is small, on the order of a dozen or so. Beyond this point, adding additional nodes does not increase the performance; bus contention negates the parallelism gained by the extra nodes. The simple local/global division is sometimes extended to a hierarchy of buses to ameliorate this problem. In such a system, an attempt is made to place shared data in a location easily accessible to all nodes which need access to it. If this is done properly, the so-called "principle of locality" tends to keep references to global memory as local as possible.

Figure 1 shows a typical multiprocessor system with a hierarchical bus structure. A group of processors on a common bus form a *cluster*. The clusters are themselves connected through address maps along a common system bus. If two tasks frequently access a particular piece of data, an intelligent task and memory allocation scheme might assign the tasks to the nodes labelled $B1$ and $B2$ and place the data in the block of global memory labeled B. Since both processors are in the same cluster as the data, they can access it without tying up the system bus. However, if the node la-

belled *A3* wishes to access that same data, it must use the system bus to get from its cluster to the desired location in the destination cluster.

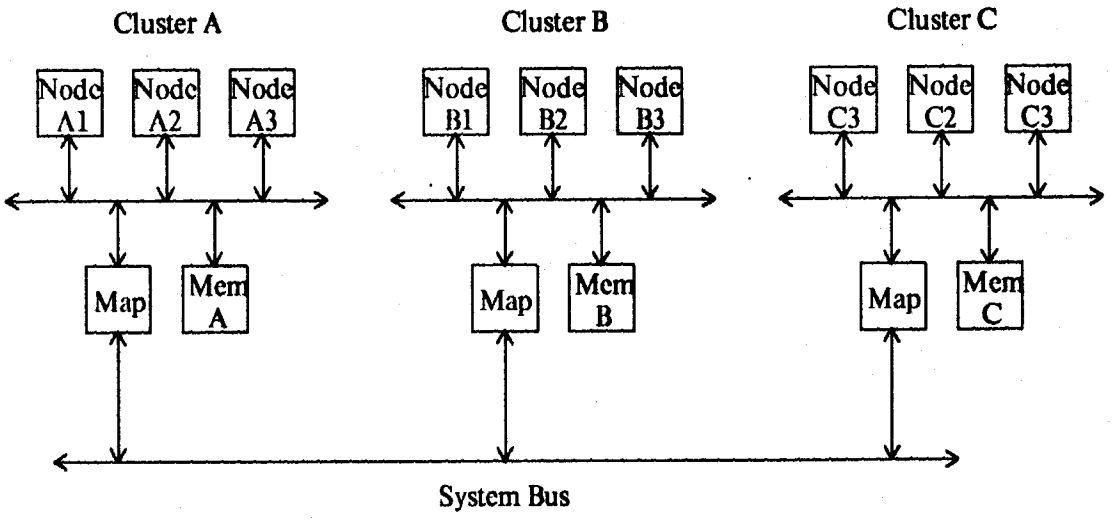


Figure 1: Hierarchical Bus Architecture

Even in a hierarchical bus system, contention places limits on the performance. The same constraints that limit the number of processors that may be placed on a single bus apply even more severely to the number of clusters on the system bus. Replicating buses, thereby providing multiple paths between processors and memory modules, is another approach to reducing memory contention [51-52]. The extreme — providing as many paths as processors — is usually implemented by using a crossbar switch or similar technology [39,57,58]. If the switching logic is migrated to the memory modules, the result is a system of processors and multi-port memories, each with as many ports as processors.

The basic Concert system is a tightly-coupled multiprocessor, since it is composed of independent microprocessor nodes which communicate via shared memory. The ul-

timate Concert system will be a loosely-coupled network of tightly-coupled multiprocessors. The system architecture for a basic Concert system uses a hierarchy of buses, but there's an important difference from the organization of Figure 1. The top-level "system" bus is actually formed from a series of bus segments, which may carry out independent accesses to blocks of global memory or may be connected to carry out longer accesses. The remainder of this chapter describes this interconnection architecture.

3.2: The General RingBus Architecture

As discussed in the previous section, bus contention is the major limiting factor in the performance of most common-bus multiprocessor systems. Bert Halstead has proposed the RingBus architecture, a processor and memory interconnection scheme which holds the promise of expanding these limits. More importantly, the hardware required to support this architecture is less than that required in many previous tightly-coupled multiprocessor systems. The RingBus architecture was chosen for Concert precisely because of these characteristics.

Figure 2 shows a simple picture of a RingBus-based multiprocessor system. The configuration is a *ring* of processing *slices*, interconnected by RingBus *segments*. Each slice contains a block of global memory and, generally, one or more processing *nodes*. A node contains at least a processor, and may also include local memory or other private resources. The RingBus is a single-transaction (read and write) bus which is under the control of a central arbiter.

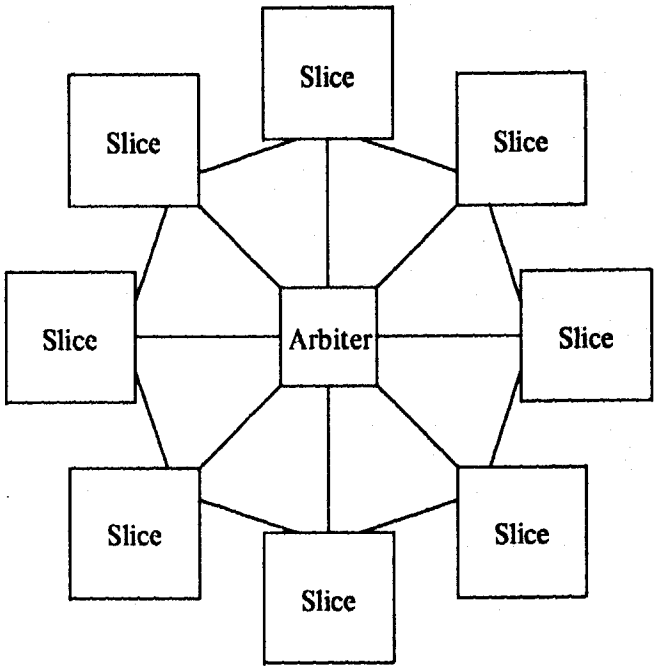


Figure 2: RingBus Multiprocessor System

The key attribute of the RingBus architecture is that different transactions may be carried out simultaneously on different RingBus segments. Several processors may be carrying out independent accesses to different blocks of global memory in a single ring. All accesses to global memory are under the control of the arbiter, which periodically examines and grants requests from the slices. In general, requests may be granted (and therefore carried out) simultaneously if they do not require any common segments on the RingBus. The arbiter can cause multiple segments to be connected together to perform a single memory access.

The ability to carry out multiple memory accesses simultaneously is what makes the RingBus architecture promising as a multiprocessor interconnection scheme. Most

methods for allowing parallel access to global memory in tightly-coupled multiprocessors require expensive hardware like crossbar switches. The RingBus architecture is a lower-cost alternative which should still yield performance superior to that of common-bus multiprocessors. The idea of using a segmented bus is not original, but it has only rarely been investigated before [4,21].

The next chapter describes the specific implementation of the RingBus architecture which was used as the top-level interconnection scheme for the Concert multiprocessor development system. The design of the hardware to support this architecture comprised the bulk of the thesis project. However, some effort was devoted to investigating the architecture itself, both through analysis and simulation.

3.3: Analysis of the RingBus Architecture

Formal analysis of most aspects of computer architecture is notoriously difficult. Most believable results require some knowledge of the programs which would be run on the machine being simulated. For this reason, software simulation is generally considered more useful than formal models. In the case of the RingBus architecture, both simulation and the design of a working system have been employed. However, some simple analysis was undertaken to investigate useful properties of the architecture. This section outlines some of these properties and describes how they influenced the design.

3.3.1: Major Implementation Options

There are a number of implementation options which affect how much parallelism the RingBus architecture can provide. The most fundamental is whether the RingBus is unidirectional or bidirectional. If it is unidirectional, then address and control lines need to be propagated in only one direction (clockwise or counterclockwise) around the ring. A bidirectional bus allows these lines to propagate in either direction. In either case, the data lines must be bidirectional to support both reads and writes or two sets of unidirectional data lines must be provided.

The choice of directionality on the RingBus is not trivial. A unidirectional bus requires a minimum of hardware, but places some inherent limitations on performance. For example, the worst-case access of a processor to global memory in the slice "behind" it requires the entire RingBus for completion. The worst-case access on a bidirectional RingBus takes only about half of the segments, but more hardware is required to support the bidirectionality. In addition, the arbiter has to be more clever if it is to take advantage of the bidirectional capability.

Another factor which affects the performance of the architecture is the degree of parallelism in each slice. For example, a slice may allow an incoming request on a RingBus segment to access its global memory at the same time a request from a processor within the slice is being propagated out along the other RingBus segment. Again, the arbiter needs to know if such parallelism exists and how best to exploit it.

The design options for the RingBus directionality and the parallelism in each slice are discussed in detail in Chapter 5. The two primary issues involved are the amount of hardware required in the slice and the time required by the arbiter to grant a re-

quest. Determining the tradeoffs between these costs and the parallelism on the Ringbus was a major part of the design effort.

3.3.2: Requesting and Granting

The problem of how to identify simultaneously grantable requests is a fundamental one in a RingBus-based system. The first part of this task is to determine the resources required to carry out a particular access. These requirements may be simply expressed in the *Segment Needed List* (SNL), which identifies the RingBus segments needed to complete an access. Consider the eight-slice ring shown in Figure 3, in which slices and segments are numbered in order starting at an arbitrary point. Suppose that the following requests are made to the arbiter with no accesses in progress and no requests pending:

$$1 \rightarrow 3;\ 2 \rightarrow 4;\ 4 \rightarrow 7;\ 5 \rightarrow 5;\ 6 \rightarrow 4$$

The format for the requests is simple; $S \rightarrow D$ is a request to the arbiter from Slice S for access to global resources in Slice D. Given this set of requests, the arbiter must then decide which requests to grant and which to defer. Table 1 expresses the same set of requests in a slightly different format which anticipates the design of the interface between the arbiter and the slices. The *REQ* line from a slice is asserted if it is making a request, and the *DST* lines identify the destination slice for a request.

In general, the segments needed for a request depend upon the direction of the access on the RingBus. Table 2 shows the SNLs for both clockwise and counterclockwise accesses for the requests of Table 1. If the RingBus is unidirectional, then all requests propagate in one direction around the ring. If the RingBus is bidirectional, then

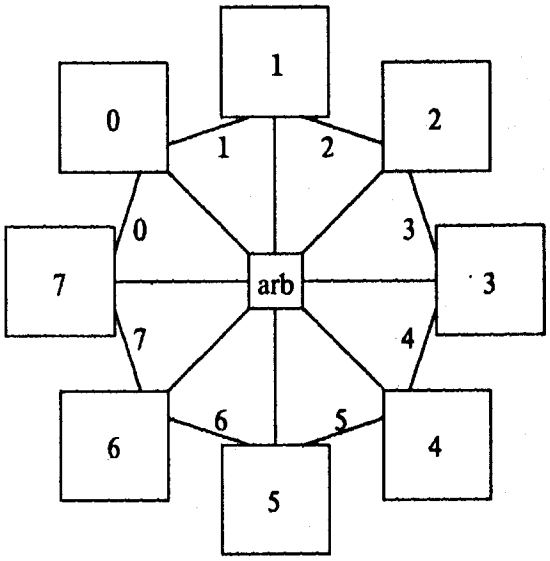


Figure 3: Ring with Slice and Segment Numbers

Source	*REQ*	*DST*
0	0	x
1	1	3
2	1	4
3	0	x
4	1	7
5	1	5
6	1	3
7	0	x

Table 1: Sample Round of Requests

a request may propagate in either direction. The directionality of the ring in Figure 3 is intentionally left unspecified.

The SNL may be translated into a binary vector simply by placing a "1" in a bit if the segment is required, or a "0" if not. For example, the SNL for a clockwise access

segment	Clockwise Access								Counterclockwise Access							
request	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1→3		x	x	x					x	x		x	x	x	x	x
2→4				x	x	x			x	x	x		x	x	x	x
4→7					x	x	x	x	x	x	x	x	x			x
5→5						x								x		
6→4	x	x	x	x	x		x	x					x	x	x	

Table 2: Segment Needed Lists for Requests of Table 1

for the request 1→3 is "01110000." Given the SNLs for two requests, it is a simple matter to determine if they are simultaneously grantable. Two global accesses may be carried out simultaneously if they do not require any common RingBus segments. Thus, two requests may be granted concurrently if the AND of their two SNLs has no bits set.

In general, when the arbiter is deciding whether to grant a request there are some number of accesses already in progress. Therefore, it is necessary to be able to efficiently determine if a request conflicts with any of a number of requests already granted or in progress. This also turns out to be simple. A composite list of segments in use thus far may be maintained, and the request under consideration is ANDed with this list. Each time a new request is granted, its SNL is ORed with the list to produce a new composite list.

3.3.3: The Role of the Arbiter

The arbiter is responsible for controlling all accesses to global memory on the RingBus. It takes as input request lines from the slices, and sends back grant signals which control the flow of information on the RingBus segments. The first cut at the arbiter is a block of combinational logic mapping request inputs to grant outputs. The arbiter's inputs must remain stable long enough for it to make a decision; likewise its outputs to the slices must remain stable while it is making decisions. Thus, both the request inputs and grant outputs must be latched. Figure 4 shows a simple model for such an arbiter in ring with *n* slices.

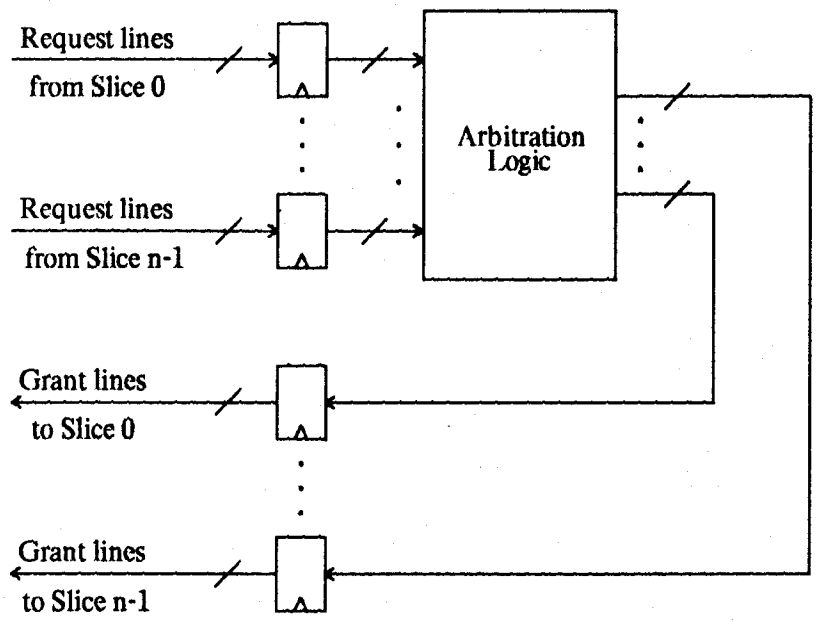


Figure 4: Combinational Model of the Arbiter

The arbiter latches its inputs and outputs on a synchronous clock. The time between clock pulses is the arbiter *cycle*, which is determined by the latency of the arbitration logic. Each time a cycle begins, a *round* of requests from the slices is latched in. At the end of the cycle, the lines indicating which of the requests have been granted are latched and sent back to the slices.

Unfortunately, the simple model of Figure 4 is insufficient on two counts. First, the arbiter must have some sort of state to record which requests are currently in progress. Since a RingBus access may take an arbitrary number of arbiter cycles to complete, the arbiter must insure that once a request is granted it remains granted until completion. Since the grant signals sent to the slices must be latched anyway, the obvious way to insure this is to feed them back into the arbiter. Figure 5 shows the resulting finite-state machine implementation of the arbiter.

The arbiter implementation of Figure 5, while more realistic than a totally combinational version, still lacks one important property. For the arbiter to be useful in a real system, it must eventually grant all requests. Some of the requests in a given round may require common RingBus segments. Thus, in general, not all requests can be granted in a single arbiter cycle. Any ungranted requests will still be pending on subsequent arbiter cycles. It is possible that whatever method the arbiter uses to select among conflicting requests will result in some requests never being granted.

This problem may be rectified by maintaining some state in the arbiter to insure that pending requests are eventually granted. This is the simplest form of *fairness* for an arbitration scheme; a request can never be locked out forever. A more strict definition of fairness in the RingBus architecture requires that all nodes have an equal

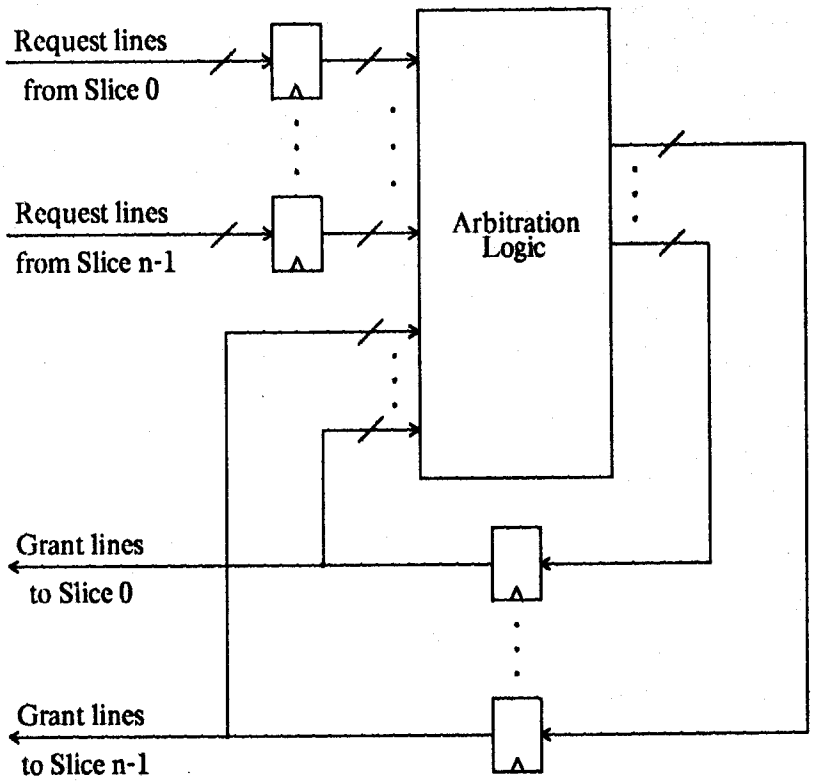


Figure 5: Finite-State Machine Model of the Arbiter

chance for global memory access. If all slices contain an equal number of nodes, this is equivalent to the requirement that all slices have an equal chance at global resources.

If the slices have different numbers of nodes, the problem of guaranteeing fairness becomes more difficult. In this case, the arbiter needs to know the exact configuration of the system, including the number of nodes in each slice. All arbitration schemes discussed in this chapter consider all *slices* equivalent. Thus, they do not guarantee

that all *nodes* have an equal chance for access to global resources.

The next section discusses some schemes to enforce a priority order on the requests. Such an order may be used to insure that a pending request is eventually granted by assigning it a higher priority than incoming new requests. The finite-state machine model of the arbiter is still valid, but it must incorporate some additional state to insure fairness. Figure 6 shows a modified model which includes some priority state.

3.3.4: Arbiter Priority Schemes

A priority ordering on the requests to the RingBus arbiter is required to insure that all requests will eventually be granted. For a ring of n slices, no more than n requests may be made to the arbiter at any one time. Thus, only n priority levels are required. If each priority level can contain only one request, then there are $n!$ possible priority orderings. If more than one request can have the same priority, there are n^n combinations.

There are two fundamentally different ways for the arbiter to order the requests. The first approach is a *history priority* scheme, in which requests are ordered by age. There are several possible implementations to support such a scheme. For example, the arbiter might include a "history counter" for each source. The priority ordering may be determined from the values of these counters by a comparison tree or similar hardware. A new request starts at the lowest priority, and its priority is increased by incrementing its counter as long as it is pending. Requests of the same age may have

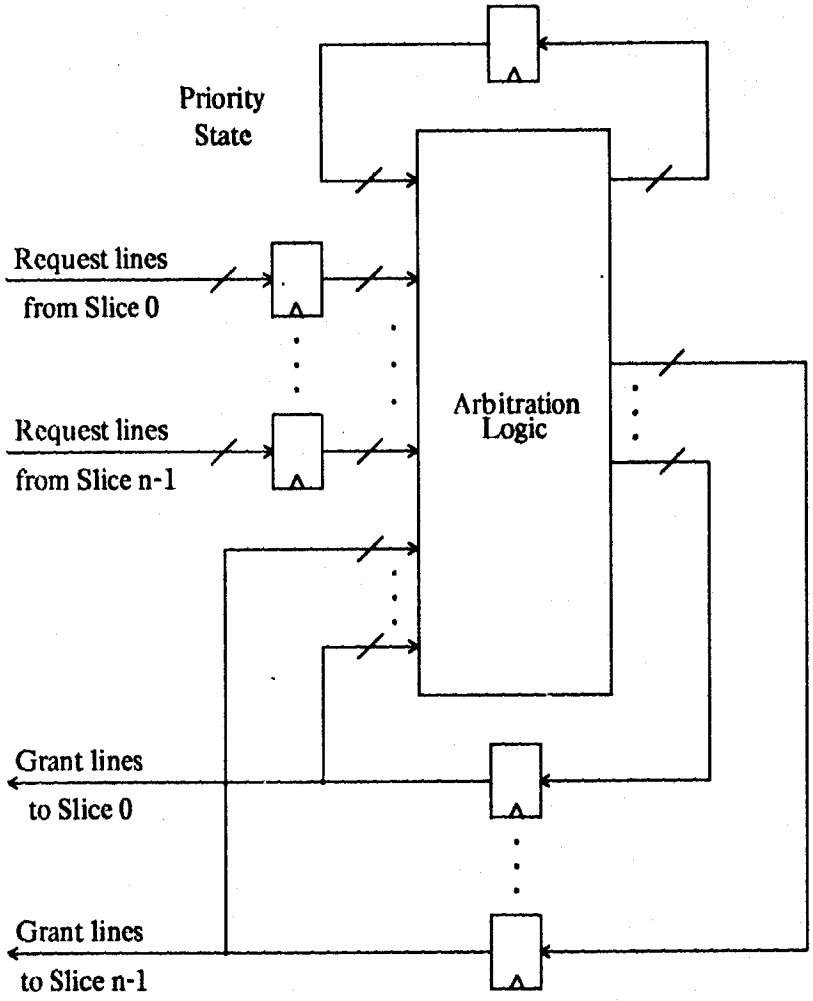


Figure 6: Model of Arbiter with Priority

the same priority; in such a case, the arbiter must decide among them arbitrarily.

Since the history counters need to hold one of *n* possible values, the number of bits required is the logarithm of the number of slices. The issue of when to increment the counters is not as simple as it might seem. Since each request may take an arbitrary number of arbiter cycles to complete, the arbiter may have periods of arbitrarily

many cycles in which it can grant no new requests. If the history counters are incremented each arbiter cycle, they can eventually overflow and older requests can end up with lower priority than newer requests.

A better, but more difficult, approach is to increment the history counters up to the maximum value to fill in "holes" in the priority ordering. Once a request has reached the top priority, it remains there until granted. If only one request is desired at each priority level, then the ordering of requests which arrive in the same cycle may be chosen arbitrarily. If multiple requests may have the same priority, a curious result ensues. As long as two requests from different cycles always have different priorities, then the number of requests possible at priority level p in a ring of n slices is $n-p$. This assumes that the priority p ranges from 0 to $n-1$, with $n-1$ being the top priority.

The "hole-filling" scheme suggests an alternative implementation for history priority — a queue for the requests. The queue needs one stage for each slice in the ring. Presumably the stages would be completely ordered, so that each stage represents a fixed priority level and contains at most one request. Each time a new request arrives, the earlier requests are pushed up and the new request is added to the bottom.

There are several serious problems with this approach. As discussed in the next section, there are circumstances in which it is valid to grant a request even if requests of higher priority remain ungranted. Once such requests complete, "holes" are left in the request queue. If the queue only has as many elements as slices, it is necessary to fill the holes in order to fit all the requests in. This requires the ability to selectively shift elements in the queue, which is not provided by many possible implementations.

If the elements of the queue are available in parallel, it is possible to simultaneous-

ly examine requests of different priority levels. If not, then a shift must be performed to present each request in turn in priority order. In a ring with *n* slices, as many as *n* requests may be granted in a single arbiter cycle. Thus, it may be necessary to shift *n* times. Similarly, as many as *n* new requests may occur in a single cycle. Thus, it is necessary either to shift as many as *n* times each cycle or to have some way of loading the queue in parallel.

The amount of hardware necessary to implement the history counter scheme is far from trivial, since it involves comparison hardware to determine the priority ordering. On the other hand, the queue implementation requires some clever hardware and may exact a time penalty for the shifting around of requests in the queue.

A history priority scheme has the advantage that requests are granted by age, a reasonable criterion for deciding conflicts. The other major class of methods to order the requests to the arbiter involves priority schemes which do not rely on the nature of the requests themselves. A *fixed priority* ordering is one such method, but it does not give all slices an equal access to global resources.

One scheme commonly used in bus arbitration is *rotating priority*. This is basically a fixed priority ordering which rotates among the slices. Such an approach works with the RingBus as well. The simplest form just assigns the top priority slot to a different slice each cycle. For example, consider again an eight-segment RingBus with the slices numbered from 0 to 7. Table 3 shows the priority orderings for nine consecutive arbiter cycles. From this representation, it is clear how the term *rotating* arose. Unlike some of the history schemes, each priority level has exactly one request at a time.

Priority Level	Slice Priorities for each Cycle								
	n	n+1	n+2	n+3	n+4	n+5	n+6	n+7	n+8
0 (low)	3	4	5	6	7	0	1	2	3
1	4	5	6	7	0	1	2	3	4
2	5	6	7	0	1	2	3	4	5
3	6	7	0	1	2	3	4	5	6
4	7	0	1	2	3	4	5	6	7
5	0	1	2	3	4	5	6	7	0
6	1	2	3	4	5	6	7	0	1
7 (high)	2	3	4	5	6	7	0	1	2

Table 3: Nine Consecutive Cycles of Rotating Priority

The implementation for rotating priority is extremely simple — a counter which identifies the current top priority slice and which is incremented each cycle. In an n-slice ring there are only n possible priority orderings, so that the number of bits in the counter is $\log_2 n$. Unfortunately, this simple scheme will not work for the RingBus. Unless a request is held at the top priority until it is granted, it may be superseded by a new request. It is possible that a request is not granted (because of conflicts with accesses already in progress) every time it reaches top priority, and is therefore locked out forever.

A more sophisticated version of rotating priority solves this problem. A request is held at the top priority until it is granted. Once the request is granted, the priority is rotated so that the next slice which has a pending ungranted request gets top priority. The hardware to implement such a scheme is a little more substantial. Instead of a simple "top priority" counter which gets incremented each cycle, more complicated logic is needed to determine the next value of the counter based on the its current value and the current ungranted requests.

3.3.5: Arbitration Algorithms

Before attempting to design the arbiter, some effort was spent in examining some possible arbitration algorithms to better understand the function which has to be performed. Assuming some sort of priority ordering on the requests, the basic algorithm is quite simple — try to grant the requests in priority order. However, there are several factors to consider.

The first point is that requests may take an arbitrary number of arbiter cycles to complete. This requires that any request, once granted, *must* be allowed to complete without interruption. In terms of the arbiter, this implies that all requests which were granted on a previous cycle and which are still active must be granted again. Algorithm 1 expresses this procedure in a simple Pascal-like language. All algorithms in this section use this same language.

```
procedure UPDATE_GRANT;          ! update the grant list

integer         S;               ! current slice being considered
constant        NSL;             ! number of slices, numbered 0 ... NSL-1
boolean array REQ[NSL];          ! true if the slice has a request pending
boolean array GRANT[NSL];        ! true if slice's request has been granted

{
    ! update the grant list with those requests granted
    ! but not yet complete

    for S := 0 to (NSL-1) step 1 do
        GRANT[S] = GRANT[S] and REQ[S];
}
```

Algorithm 1: Updating the Grant List

The *GRANT* array is the structure that is modified by the arbitration algorithms. In hardware terms, the array can be thought of as the *GRANT* lines which feed into the output latches. At the end of the arbiter cycle, the *GRANT* values are latched into the register and sent to the slices. Likewise the value of the *REQ* array may be thought of as the latched values of the request lines from the slices. An access is assumed complete once its request line has dropped. Thus, at the start of an arbiter cycle, a request is still in progress if its bits in the *GRANT* and *REQ* arrays are both set.

Once the *GRANT* array has been updated, the actual arbitration process can occur. The core of the decision whether or not to grant a request is the SNL abstraction discussed earlier in this chapter. Two requests may be granted simultaneously if they require no common RingBus segments. Algorithm 2 is a simple function for identifying conflicts among requests. It takes as inputs two slice numbers and returns *true* if the two requests may be granted simultaneously and *false* if they may not. It determines this by comparing the SNLs of the two requests. It uses an unspecified function *SNL* which takes a slice number and a segment number and returns *true* just in case the specified request needs the specified segment for completion.

There is one other auxiliary routine which is used by the arbitration algorithms. This procedure, *UPDATE_PRI*, updates the priority ordering of the requests at the end of an arbiter cycle. A detailed priority update algorithm is not given because it is so hardware-dependent and because there are so many possible schemes. The exact action taken by *UPDATE_PRI* depends upon the particular priority scheme and hardware implementation selected. For example, the rotating priority update algorithm keeps the priority order the same if the top priority request is ungranted. If it is granted, then the

```
function CONFLICT(S1, S2);          ! returns true if requests of slices
                                    ! S1 and S2 require any common segments
integer         S1, S2;
boolean         CONFLICT;

extern function    SNL(SL,SG);      ! returns true if request of slice SL needs
                                    ! segment SG, else false
constant        NSL;                ! number of slices, numbered 0 ... NSL-1
integer         SEG;                ! current segment being considered
boolean         CONFL;              ! set true once a conflict is found

{
    CONFL := false;
    for SEG := 0 to (NSL-1) step 1 do
        CONFL := CONFL or (SNL(S1,SEG) and SNL(S2,SEG));
    return (CONFL);
}
```

Algorithm 2: Identifying Conflicting Requests

priority is rotated to give the highest-priority ungranted request top priority for the next arbiter cycle. If no requests remain ungranted, the priority might be rotated by one.

The arbitration algorithms assume only that some sort of priority order has been established. This order is represented by the array *PRI*, which holds the priority value for each of the *NSL* slices in the ring. Each priority value is between 0 and *NSL*–1, with *NSL*–1 being the highest priority. The algorithms allow for arbitrary priority orderings and permit multiple requests at each priority level. They support any of the priority schemes discussed in the last section, ranging from rotating priority to history counters.

There are two fundamental goals for the arbitration algorithm, and for the arbiter itself. It should be *fair*, giving all slices an equal shot at the RingBus and the global resources. Of course, this also means that all requests should be eventually granted.

It should also try to exploit the maximum parallelism on the RingBus by granting as many requests as possible simultaneously. There are several important variations in the arbitration algorithm which affects its compliance with these design goals. Algorithm 3 is the most straightforward arbitration scheme. It simply iterates down through the priority levels, attempting to grant all requests at each level. In this manner, requests of higher priority get a chance to be granted before requests of lower priority.

There are several important points to note about Algorithm 3. The first is that it is general enough to work with any of the three priority schemes presented in the last section. The appropriate version of *UPDATE_PRI* is chosen to reflect the priority scheme. If a rotating priority scheme is used, the algorithm may be optimized to recognize that there is exactly one request at each priority level. The algorithm also contains calls to *UPDATE_GRANT* and *CONFLICT*, as defined earlier.

Although Algorithm 3 appears to be a perfectly reasonable arbitration method, it does not guarantee that all requests are eventually granted. The problem arises when a high priority request does not get granted because of a conflict with a request already in progress. If this happens, lower priority requests may be granted in that same cycle. However, it is possible for the high priority request to get locked out forever if lower priority requests which conflict with it are continually granted. The obvious solution to the problem is shown in Algorithm 4, which stops granting requests as soon as a conflict is found.

Algorithm 4 is termed the *limited arbitration scheme* because it grants fewer requests than does Algorithm 3. This means that it allows fewer simultaneous accesses on the RingBus, and some of the advantage of the architecture is lost. Fortunately,

there is a third choice which exploits more parallelism than Algorithm 4 yet insures that all requests *will* be eventually granted. This approach, the *full arbitration scheme*, is shown in Algorithm 5.

The key is that the problem with Algorithm 3 arises only when it grants lower priority requests which conflict with ungranted higher priority requests. Algorithm 5 solves this problem by granting a request only if it does not conflict with either a request already granted *or* an ungranted higher priority request. As usual, it checks for conflicts by comparing the SNLs of the requests.

All three arbitration algorithms share a fundamental common trait — they make only one pass through the requests. It is possible to imagine algorithms which make multiple passes through the requests, trying to achieve the maximum parallelism on the RingBus. Such schemes require some sort of figure of merit to compare different sets of grants for the same round of requests. Among the metrics which might be considered by a multiple-pass arbiter are:

1) Granting each request in as short a time as possible.

2) Granting as many requests as possible simultaneously.

3) Using as many RingBus segments as possible simultaneously.

4) Giving preference to "short" requests, where the length of a request is defined as the number of RingBus segments in its SNL.

5) Giving preference to "long" requests.

6) Considering both directions on a bidirectional RingBus for each request.

```
procedure ARBITRATE;                ! perform the arbitration

integer         S;                  ! current slice being considered
integer         C;                  ! competing slice being considered
integer         P;                  ! current priority being considered
boolean         CONFOUND;           ! true if conflict has been found
boolean array   REQ[NSL];           ! true if the slice has a request pending
boolean array   GRANT[NSL];         ! true if slice's request has been granted
integer array   PRI[NSL];           ! priority of slice's request (ranges
                                    ! between 0 and NSL-1)
{
   UPDATE_GRANT;

   ! work down through priority levels

   for P := (NSL-1) to 0 step -1 do

      ! iterate over the slices

      for S := 0 to (NSL-1) step 1 do
         if (PRI[S] = P) and REQ[S] and not(GRANT[S])
         {
            ! if the request is the right priority
            ! and not yet granted, try to grant it

            CONFOUND := false;
            for C := 0 to (NSL-1) step 1 do

               ! check for conflicts with requests
               ! already in progress

               if GRANT[C] then
                  if CONFLICT[S,C] then
                  {
                     CONFOUND := true;
                     exitloop(C);
                  }
            ! if no conflict was found, grant request

            if not(CONFOUND) then GRANT[S] := true;
         }
   UPDATE_PRI;
}
```

Algorithm 3: Initial Arbitration Scheme

```
procedure ARBITRATE;                ! perform the arbitration

integer         S;                  ! current slice being considered
integer         C;                  ! competing slice being considered
integer         P;                  ! current priority being considered
boolean array   REQ[NSL];           ! true if the slice has a request pending
boolean array   GRANT[NSL];         ! true if slice's request has been granted
integer array   PRI[NSL];           ! priority of slice's request (ranges
                                    ! between 0 and NSL-1)

{
    UPDATE_GRANT;

    ! work down through priority levels

    for P := (NSL-1) to 0 step -1 do

        ! iterate over the slices

        for S := 0 to (NSL-1) step 1 do
            if (PRI[S] = P) and REQ[S] and not(GRANT[S]) then
            {
                ! if the request is the right priority
                ! not yet granted, try to grant it

                for C := 0 to (NSL-1) step 1 do

                    ! check for conflicts with requests
                    ! already in progress

                    if GRANT[C] then
                        if CONFLICT[S,C] then
                            exitloop(P);

                ! if no conflict was found, grant request

                GRANT[S] := true;
                }
    UPDATE_PRI;
}
```

Algorithm 4: Limited Arbitration Scheme

```
procedure ARBITRATE;                ! perform the arbitration

integer         S;                  ! current slice being considered
integer         C;                  ! competing slice being considered
integer         P;                  ! current priority being considered
boolean         CONFOUND;           ! true if conflict has been found
boolean array   REQ[NSL];           ! true if the slice has a request pending
boolean array   GRANT[NSL];         ! true if slice's request has been granted
integer array   PRI[NSL];           ! priority of slice's request (ranges
                                    ! between 0 and NSL-1)
{
   UPDATE_GRANT;

   ! work down through priority levels

   for P := (NSL-1) to 0 step -1 do

      ! iterate over the slices

      for S := 0 to (NSL-1) step 1 do
         if (PRI[S] = P) and REQ[S] and not(GRANT[S]) then
         {
            ! if the request is the right priority
            ! and not yet granted, try to grant it

            CONFOUND := false;
            for C := 0 to (NSL-1) step 1 do

               ! check for conflicts with requests in progress
               ! or ungranted requests of higher priority

               if GRANT[C] or (REQ[C] and (PRI[C]>PRI[S])) then
                  if CONFLICT[S,C] then
                  {
                     CONFOUND := true;
                     exitloop(C);
                  }
            ! if no conflict was found, grant request

            if not(CONFOUND) then GRANT[S] := true;
         }
   UPDATE_PRI;
}
```

Algorithm 5: Full Arbitration Scheme

Multiple-pass arbitration algorithms are not considered in this thesis. The primary reason for this is the practical difficulty in implementing such an approach in hardware. Fundamentally, considering different possibilities for the same round of requests requires either parallel hardware or serial passes with the same hardware. Since the RingBus was investigated in terms of its usefulness for Concert, both implementations were judged unacceptably costly.

3.4: Simulation of the RingBus Architecture

Simulation has long been a useful tool in the design of multi-level memory systems, protocols, and other aspects of computer architecture. Likewise, it can have considerable impact on the detailed definition and implementation of a RingBus-based multiprocessor system. Simulation can help to choose the arbitration and priority schemes and to assess the impact on performance of some possible variations in the RingBus architecture. The thesis project included some simulation to help understand these issues for the Concert RingBus implementation.

An arbiter simulator was written in the language C [35] under the UNIX [55] operating system to test the effects of some different implementations of the RingBus architecture. Three main parameters were varied — the directionality of the RingBus, the scheme used to enforce priority, and the actual arbitration algorithm. The program was designed in a modular fashion to support these variations.

The simulator supports either a unidirectional or bidirectional RingBus. The unidirectional case allows some parallelism within the slices; an access to the resources in a slice can be in progress at the same time that a request from a node within the slice

is being sent out to another slice. The bidirectional RingBus supports this same type of parallelism in one direction, but not in the other. Complete details of the two options are given in Chapter 5; a model for the unidirectional case is shown in Figure 19b and a model for the bidirectional bus in Figure 19f. From the arbiter standpoint, the difference between the two is reflected purely in the SNLs generated for the requests.

The simulator supports two priority schemes. The first is a history priority scheme which allows only one request per priority level and shifts requests to fill up holes. The other is a rotating priority scheme which makes the highest priority ungranted request the top priority request for the next cycle. In addition, two different arbitration algorithms are also allowed. The *limited* arbitration scheme is modelled after Algorithm 4, since it stops granting requests once it encounters one it can't grant. The *full* arbitration scheme grants any requests which don't conflict with higher priority requests or requests already granted, and thus follows Algorithm 5. The simple rotating priority scheme and the arbitration scheme of Algorithm 3 were not supported because, as explained earlier, they can result in a request being locked out forever.

A simple random number generator program was used to generate lists of requests to send to the arbiter. It uses a distribution in which accesses from a given slice to other slices vary as an inverse exponential based upon the distance between the slices. This distribution was chosen in an attempt to model the pattern of accesses which might be found in an actual RingBus system if locality was considered in the allocation of global data to global memory blocks in the slices. The distribution was also chosen to reflect the Concert RingBus implementation. The majority of the requests

are assumed to be to global memory within the same slice; in Concert such an access does not require a request to the arbiter or the use of any RingBus segments. The least likely access is thus to the same slice, which happens only on an access to global control registers.

The request sequences generated by this method are probably not a realistic model of the memory accesses in the operating Concert system. Regardless, it is interesting to compare the results of variations in the simulator options for a single sequence of requests. Table 4 summarizes two such experiments. The first set of figures was generated by a sequence of 500 requests from each slice. About half of these requests are *null*, corresponding to memory accesses within the slice. The remainder are requests to another slice. The distribution for the requests favors the bidirectional RingBus, since it makes accesses to slices at the same distance in either direction around the bus equally likely. The same sequence was run through the arbiter simulator eight times, over all variations in the options.

The second set of figures in Table 4 was generated by a different sequence of 500 requests from each slice. The distribution for this sequence favored the unidirectional RingBus, since it considers only one direction. In other words, an access from a slice to the one "behind" it on the RingBus is very unlikely. The same eight combinations of options were tried.

Table 4 gives four statistics for each of the simulator runs. t_T is the total number of arbiter cycles required to grant all the requests. The average number of cycles a request had to wait before being granted is shown as t_{WM}. r_M is the average number of requests made to the arbiter each cycle. The parallelism on the RingBus is

RingBus Direct.	Arbiter Scheme	Priority Scheme	Bidirectional Bias					Unidirectional Bias				
			t_T	t_{WM}	r_M	a_M	s_M	t_T	t_{WM}	r_M	a_M	s_M
Unidir.	Limited	History	2568	5.29	6.05	1.94	4.50	2303	4.21	5.81	2.17	4.25
Unidir.	Limited	Rotating	2765	6.11	6.21	1.80	4.18	2584	5.47	6.15	1.93	3.79
Unidir.	Full	History	2344	4.46	5.92	2.13	4.93	2013	3.16	5.61	2.48	4.87
Unidir.	Full	Rotating	2415	4.81	6.04	2.07	4.78	2149	3.82	5.86	2.32	4.56
Bidir.	Limited	History	2205	3.86	5.75	2.26	4.10	2265	4.08	5.80	2.20	4.09
Bidir.	Limited	Rotating	2351	4.66	6.07	2.12	3.85	2461	5.08	6.15	2.03	3.77
Bidir.	Full	History	1874	2.52	5.34	2.66	4.82	1927	2.75	5.44	2.59	4.81
Bidir.	Full	Rotating	1879	2.72	5.54	2.66	4.81	2020	3.24	5.67	2.47	4.59

t_T = Total cycles to grant all requests
t_{WM} = Average cycles waited between request and grant
r_M = Average requests made each cycle
a_M = Average accesses in progress each cycle
s_M = Average number of segments in use each cycle

Table 4: RingBus Arbiter Simulation Results
for Two Sequences of 500 Requests per Slice

represented by a_M and s_M, which list the average number of accesses taking place and the average number of segments in use on the RingBus each cycle. a_M is equivalent to the average number of requests granted each cycle.

Each sequence has 1994 *active* (non-null) requests. Once granted, each request takes one or two cycles to complete. The simulator inserts a null request between any two consecutive active requests from a single slice, since reading a null request for a cycle is the only way the arbiter can tell that the previous request is done.

For comparison purposes, a program was also written to simulate the arbiter for a multiprocessor system in which all the slices are connected along a single common bus. An idle cycle is required between any two accesses to signal the end of a re-

quest and to simulate the bus exchange time. The same two priority schemes were used to perform the arbitration, and simulation runs were made with both priority schemes for the same two request sequences which produced Table 4. The result for the common bus simulator is shown in Table 5.

Priority Scheme	Bidirectional Bias					Unidirectional Bias				
	t_T	t_{WM}	r_M	a_M	s_M	t_T	t_{WM}	r_M	a_M	s_M
History	5018	15.0	6.96	0.994	0.994	5018	15.0	6.96	0.994	0.994
Rotating	5020	15.0	6.95	0.994	0.994	5020	15.0	6.95	0.994	0.994

t_T = Total cycles to grant all requests
t_{WM} = Average cycles waited by request before grant
r_M = Average requests made each cycle
a_M = Average accesses in progress each cycle
s_M = Average number of segments in use each cycle

Table 5: Common Bus Arbiter Simulation Results
for Two Sequences of 500 Requests per Slice

In a common-bus system, there is only one bus segment and only one request can be granted at a time, so a_M and s_M have a maximum value of one. Idle bus cycles reduce these numbers, although they remain identical. As it turned out, the request sequences produced a few idle cycles, and so the values of a_M and s_M are slightly less than unity. The statistics for both the unidirectional and bidirectional biased sequences are identical; the only difference between them is the destinations, which don't matter to the arbiter of a common bus.

In addition to computing statistics based on rounds of arbitration, the simulator also displays a crude representation of the RingBus, showing the accesses currently in

progress and the segments currently in use. This display capability is due to the efforts of Jim Troisi, and provides an easy way to observe the activity in the ring.

It is unwise to draw any quantitative conclusions on the basis of the experiments with the arbiter simulator. However, the results do give a feel for the advantage gained by a bidirectional RingBus over a unidirectional one, and the variations caused by the different arbitration algorithms and priority schemes. More credible results would require more extensive simulations, preferably based on sequences of memory accesses culled from programs written for a shared-memory multiprocessor system. Such results could undoubtedly be used to help make design decisions for future RingBus implementations.

Chapter 4: The Concert System Architecture

This chapter outlines the implementation of the RingBus architecture which was incorporated in the Concert multiprocessor development system. It includes a discussion of the Concert address space and the terminology which is used in the remainder of the thesis. The detailed design of two custom hardware modules is discussed in the following chapters.

4.1: The Concert RingBus Implementation

A complete Concert system consists of some number of RingBus rings, interconnected by dedicated serial or parallel lines, as shown in Figure 7. The result is a loosely-coupled network of tightly-coupled multiprocessors. The initial hardware and software design has concentrated on the construction of a single ring, while providing the hooks for an eventual multi-ring system. Most of the discussion in the remainder of the thesis concerns a Concert system of only one ring, although mention is made of some important points about larger systems.

The Concert architecture can best be understood in terms of its hierarchical structure. The computing power of the system lies in processing *nodes*, each containing a processor and, possibly, local memory. Several of these nodes, global memory boards, and a RingBus interface board comprise a RingBus *slice*, which is physically housed in a card cage. Multiple cages connected together by RingBus *segments* and a RingBus arbiter form a *ring*. Finally, one or more rings may be interconnected to form a Con-

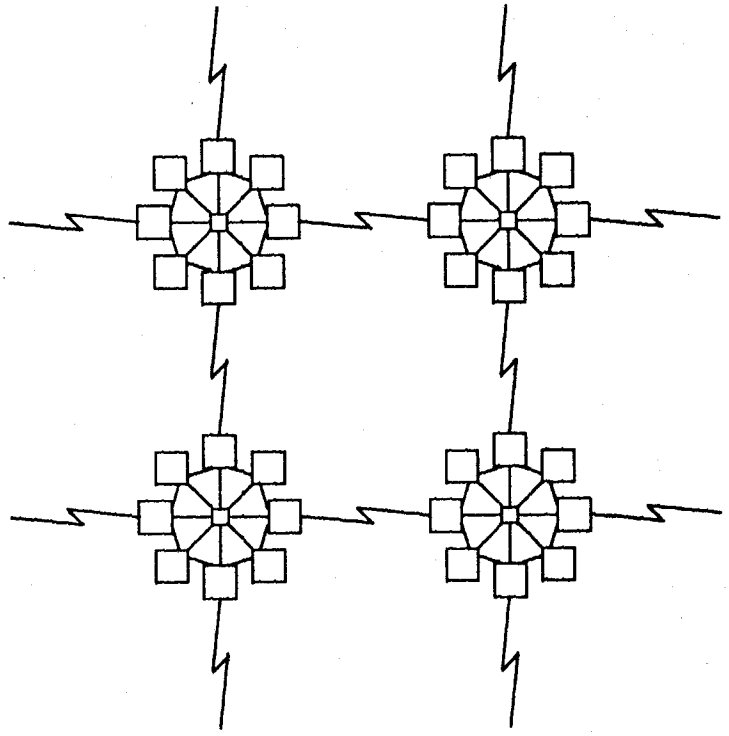


Figure 7: Concert System

cert *system*. Each of the levels in this hierarchy is outlined briefly.

4.1.1: The Node

A Concert processing node contains at least a processor, and generally some memory and I/O ports as well. The Concert prototype uses a Microbar DBC68K processor board [13] and a Microbar DBR50 memory board [15] for each node. Figure 8 shows the contents of a DBC68K board. It is centered around the Motorola MC68000 [37], a 16-bit microprocessor with a 24-bit address bus. The DBC68K also contains 24

bits of parallel I/O, two RS-232C serial I/O ports, two programmable timers, and a programmable interrupt controller. Sockets are included for an optional Motorola MMU68451 memory management unit, 4K bytes of RAM, and 32K bytes of ROM. The DBR50 memory board contains up to 512K bytes of dynamic RAM, organized as 16-bit words. It includes refresh circuitry and parity generation and check.

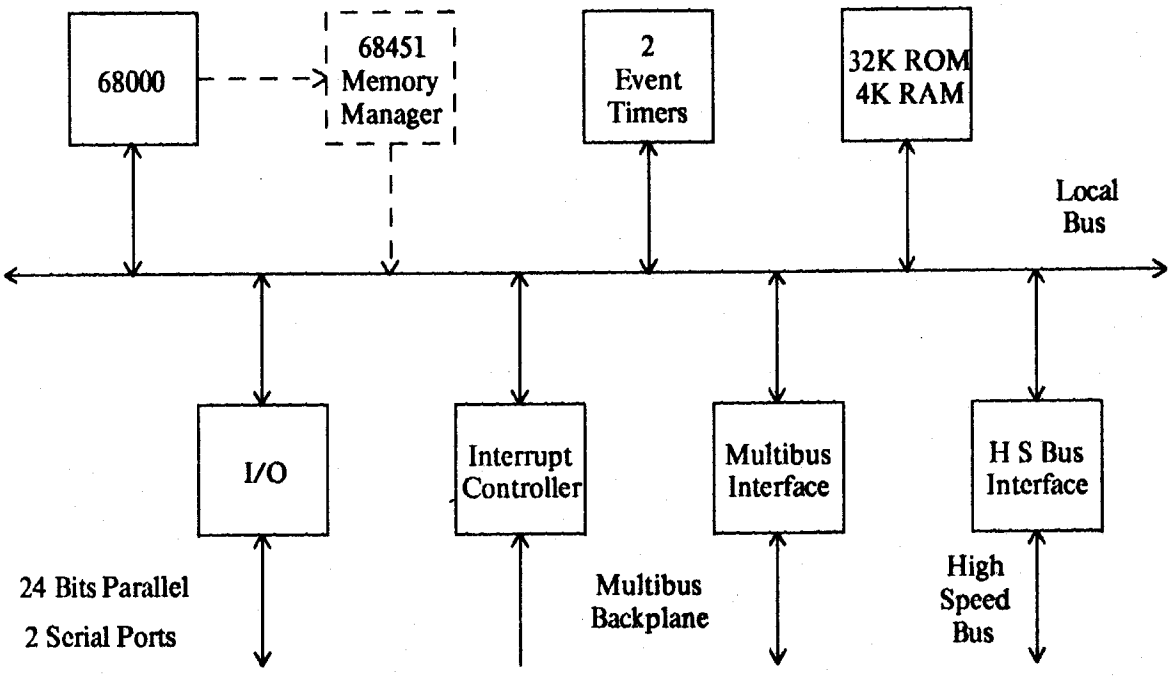


Figure 8: Concert Node — DBC68K Board

Both boards provide two buses, which serve different purposes in the Concert system. They both provide a full IEEE 796 bus [40], a standard bus closely related to Intel's Multibus [9]. The card cage which houses the nodes has a Multibus backplane, and both the processor and memory boards plug into this cage. Both boards also have an interface for a Microbar custom High Speed Bus. Each processor board in a

Concert system uses its High Speed Bus exclusively to access its node memory board. The Multibus backplane allows access to global memory, as described in the next section. Both the Multibus and High Speed Bus interfaces on the processor board are "one-way;" the resources on the DBC68K are not available from off the board.

The basic Concert node consists of one DBC68K board and one DBR50 card, and the remainder of the thesis assumes this configuration. However, a number of variations are possible. Multiple memory cards may be connected along the High Speed Bus of a single processor. Other processor boards, such as the Intel iSBC 86/12A [33], can be used instead of the Microbar DBC68K. In fact, any processor with a Multibus interface to plug into the backplane can be used as a Concert node. Some future applications will use floating-point processors or other special nodes to perform specific functions.

A Concert node performs all communication with other nodes in the ring by accessing shared memory. The next section describes in detail how nodes communicate with each other within a single Multibus card cage. Later sections describe communication between nodes in different card cages and between nodes in different rings.

4.1.2: The Slice

The basic physical Concert building block is the slice, a Multibus card cage containing one or more nodes, some slice memory, and a *RingBus Interface Board* (RIB). Figure 9 shows a slice containing four processing nodes, each composed of a processor board and a memory board. The number of nodes in a slice is flexible, since card cages are available with different numbers of slots. The main limitation on the number

of nodes in a slice is the contention for the Multibus backplane. Concert allows up to eight nodes per slice, although contention may set lower practical limits for some applications.

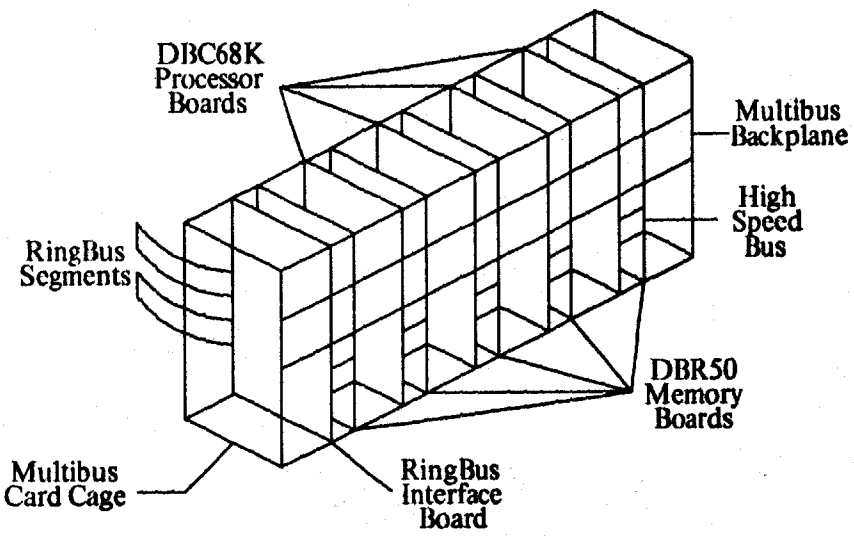


Figure 9: Concert Slice

Since the memory board of each node is connected to the Multibus backplane as well as to the processor's High Speed Bus, nodes within a single slice can access each other's memory directly on the Multibus. However, the node memories are not accessible from outside the slice. It is possible to plug other cards into the Multibus backplane, such as network ports, memory boards without High Speed Bus interfaces, or other special-purpose hardware. These boards would also be accessible via the Multibus from any node in the slice, but not from outside the slice.

Since node memories are not accessible from outside the slice, global memory must be provided to allow communication between nodes in different slices. Global

memory is supported by the RIB, which plugs into the Multibus backplane along with the nodes. Figure 10 shows a block diagram of the RIB.

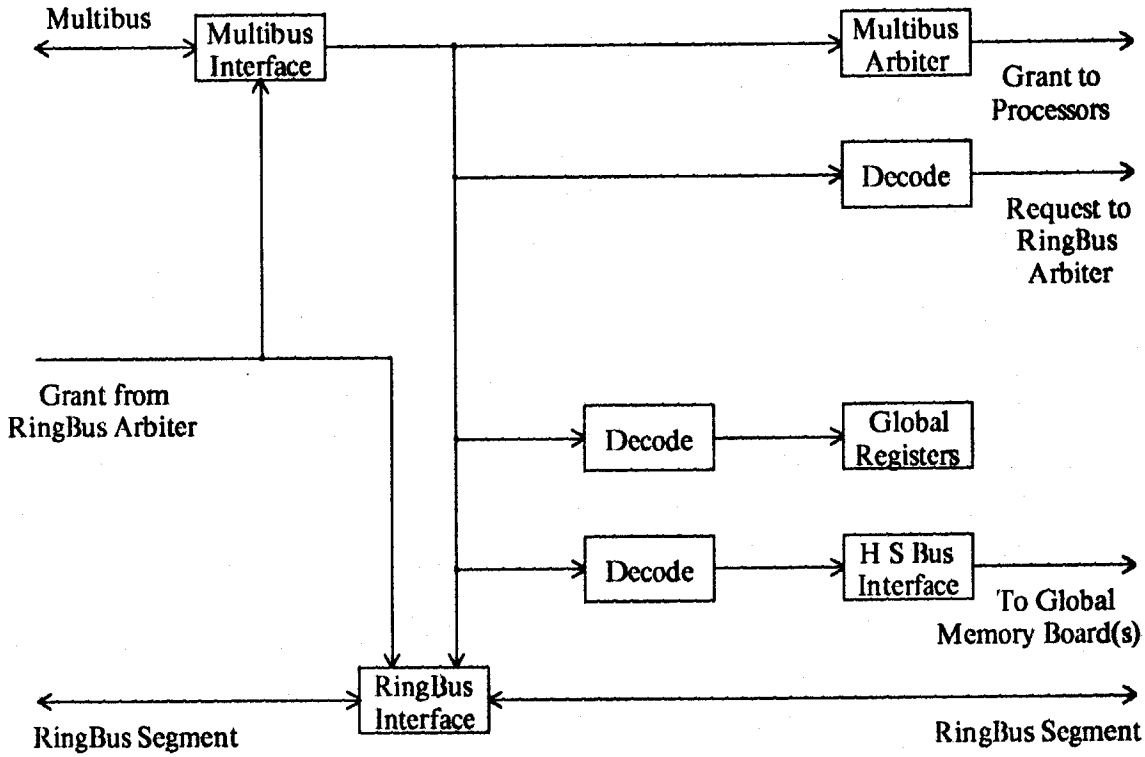


Figure 10: RingBus Interface Board

The RIB includes hardware to perform a variety of important functions. It contains a High Speed Bus interface to allow other slices to access global memory boards plugged into the backplane. Global memory consists of one or more DBR50 boards, the same as those used for the node memories. The difference is that global memory is accessible via the Multibus from nodes within the slice *and*, via the RingBus, RIB and High Speed Bus, from nodes in other slices in the ring.

The RIB contains several global registers which are accessible from any node in

the ring. It plugs into the Multibus backplane to provide an access path from nodes within the slice to the global registers or to other slices. It provides interfaces to two RingBus segments. If the RingBus is unidirectional, each segment propagates requests in only one direction. The interface to the incoming RingBus segment carries requests for global resources either in the RIB's slice or in another slice farther along the RingBus. If a request is bound for global resources in another slice, it is propagated through the outgoing RingBus interface. Such a request can originate either from a node within the slice or from the incoming RingBus segment. If the RingBus is bidirectional, either segment may propagate requests to or from the slice.

All accesses to global resources which are handled by the RIB are under the control of a central RingBus arbiter. The RIB is responsible for sending requests to the arbiter. The arbiter in turn sends back grant lines which tell the RIB how to connect the buses to carry out the accesses. Finally, the RIB also contains the hardware to perform the arbitration for the Multibus backplane.

4.1.3: The Ring

A collection of up to eight slices and an arbiter comprise a Concert ring. The slices are interconnected by RingBus segments, with the RIBs providing the interface to the slice Multibus. Request and grant lines run between the arbiter and the slices to control the operations on the RingBus. A Concert ring forms a complete tightly-coupled multiprocessor system, since the global memory in each slice is accessible from any node in the ring. The hardware allows up to eight nodes in each of eight slices, yielding a maximum ring of sixty-four nodes.

4.1.4: Multi-Ring Systems

If a Concert system of more than sixty-four nodes is desired, multiple rings must be interconnected. The result is a loosely-coupled system of tightly-coupled rings; a node in one ring is not able to directly access memory in another ring. Communication between two rings in a multi-ring system is effected by dedicating a node in each ring as a server and using one of its I/O ports to provide a link.

Since two rings can communicate only by sending messages, any reference to a node in another ring is a reasonably complicated process. The source node has to inform the node handling the ring interface via shared memory. The interface node must then send a message over the port to the interface node on the other ring. Finally, that node must contact the destination node, again by shared memory. Although such a scheme is somewhat unwieldy, most of the details are hidden from the typical Concert user. The drawback, of course, is that a different mechanism is used to communicate between nodes in different rings and nodes in the same ring.

4.2: Concert Terminology

Before describing the Concert design, it is necessary to define more completely the hierarchy of resources in the system. In so doing, a terminology is established which is used in the remainder of this thesis. The terms *ring*, *slice* and *node* are used in a manner consistent with their earlier definitions. Within a Concert ring, there are several classes of resources, each available to some subset of the nodes in the ring.

As discussed in the previous section, the DBC68K processor board includes a

variety of resources which are not accessible from off the board. These include small amounts of RAM and ROM, I/O ports, and various peripherals. Since the resources on a processor board are available only within the node, they are referred to as *private resources*.

The DBR50 memory board of each processor is not a private resource, since it is accessible via the Multibus from any node within the same slice. However, the RIB does not provide a Multibus master and there is no way for nodes in other slices of the ring to access this memory. Any other boards plugged into the Multibus backplane except the global memory boards are also accessible from any node within the slice but not to any nodes in other slices. The local memory boards and other resources available on the Multibus of a slice are called *Multibus resources*.

Private resources and Multibus resources together comprise *local resources*, so termed because they are available only locally to the nodes within a slice. Those resources which are accessible from any node in a ring are called *global resources*, and they fall into two classes.

Each slice contains one or more DBR50 boards of global memory, which are connected to the RIB by a High Speed Bus. The RIB provides an access path from the RingBus to the High Speed Bus, allowing any node in any slice of the ring to access global memory. The global memory boards are dual-ported and are also accessible from the Multibus. They are the only boards plugged into the Multibus, other than the RIB itself, which are global rather than local. The dual-port feature decreases the loading on the RingBus, since a node may access global memory within its own slice

directly on the Multibus.

The global memory comprises the bulk of the global resources, but the RIB also provides a set of global registers. As the name implies, these registers are accessible from any node in the ring via the RingBus. The global registers are not dual-ported; all accesses, even from a node within the same slice, must pass through the RIB. Like all accesses requiring the RIB and one or more RingBus segments, they occur under the control of the RingBus arbiter.

The tree of Figure 11 summarizes the classes of resources available in a Concert ring. All these resources are collectively referred to as *ring resources*. Since multi-ring systems are loosely-coupled, resources in other rings are available only indirectly, by means of message-passing.

4.3: The Concert Address Space

Prior to discussing the individual modules of the Concert design, it is necessary to define the address space in which they operate. The DBC68K processing node has a 24-bit address space. The address lines from the processor are actually virtual addresses, and the memory management unit may translate these into physical addresses. The present discussion refers to the physical address space, i.e. the actual input addresses which cause resources to respond.

As discussed in the previous section, there are four classes of resources which must be mapped in the Concert address space — private resources, Multibus resources, global memory and global registers. Table 6 shows the address space of a Concert node, and the assignments which accommodate these classes of resources.

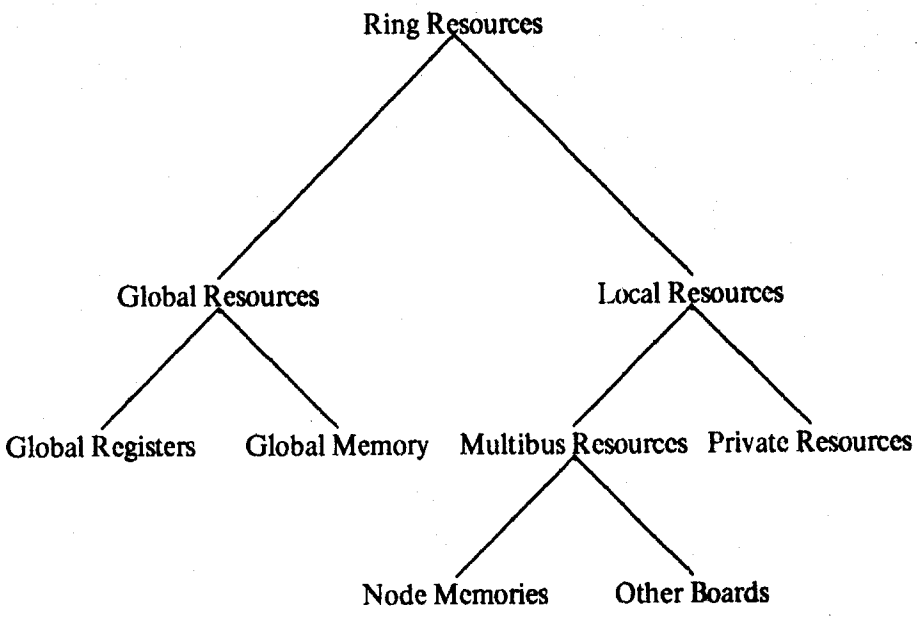


Figure 11: Hierarchy of Concert Resources

The 24-bit physical address from the processor board is the same as that which appears on the Multibus backplane when the processor is master. This address is called the *Multibus address*.

If the top two bits of the Multibus address are "01" or "10," the reference is to global memory. The first, third and fourth highest-order bits define the slice containing the global memory, and the remaining twenty bits address the byte within the memory. This allows eight slices per ring and a maximum of one megabyte (two DBR50 boards) of global memory per slice, for a total of 8M bytes of global memory in a ring. If a node is accessing global memory within its own slice, the access is carried out entirely on the Multibus. Otherwise, the RIB sends a request to the arbiter which must be

Locations (Hex)	Allocated For
000000 - 000FFF	4K On-Board RAM
001000 - 00FFFF	Available for Multibus Resources
010000 - 01FFFF	Reserved for Global Registers
020000 - 3FFFFF	Available for Multibus Resources
400000 - 4FFFFF	Global Memory for Slice 0
500000 - 5FFFFF	Global Memory for Slice 1
600000 - 6FFFFF	Global Memory for Slice 2
700000 - 7FFFFF	Global Memory for Slice 3
800000 - 8FFFFF	Global Memory for Slice 4
900000 - 9FFFFF	Global Memory for Slice 5
A00000 - AFFFFF	Global Memory for Slice 6
B00000 - BFFFFF	Global Memory for Slice 7
C00000 - EE7FFF	Available for Multibus Resources
EE8000 - EEFFFF	32K On-Board PROM
EF0000 - EFFEFF	Multibus I/O Space
EFFF00 - EFFFFF	On-Board I/O Space
F00000 - FFFFFF	Available for Multibus Resources

Table 6: Multibus Address Space

granted before the access can proceed.

The remainder of the Multibus address space is divided between global registers and local resources. The global registers on the RIB are mapped in the 64K block of addresses from 010000 to 01FFFF. The exact locations of these registers as well as details of the registers themselves are given in Chapter 5. The RIB must send a request to the arbiter for any access to these registers, even if the request is from within the same slice.

One component of the local resources is the private resources on a processor board. The DBC68K maps such resources into several different parts of the address space. The 32K PROM is mapped into low memory (starting at 000000) at startup, and is then mapped between EE8000 and EEFFFF. The 4K RAM is mapped into low

memory after startup. Interrupt vectors are located in the range from 000000 to 0003FF, and thus fall in the on-board RAM.

The range from EF0000 to EFFFFF is designated as *I/O space*. All on-board control registers and I/O devices fall in the range from EFFF00 to EFFFFF. The addresses between EF0000 and EFFEFF are passed out as I/O operations on the Multibus. Thus, any I/O boards on the slice Multibus are mapped in this range.

The final resources included in the address space are the Multibus resources. These include the DBR50 node memory boards and any other boards plugged into the Multibus card cage which respond to memory reads or writes *except* the global memory boards, which are mapped in global address space. The assignment of Multibus resources is almost totally unrestricted by hardware. Any address with its two highest bits both "0" or both "1" is allowable, so long as it does not conflict with either the global registers or the private resources.

Much as the global memory and global registers have the same addresses throughout the ring, all nodes within a slice access the Multibus resources at the same addresses. Specifically, this means that a node memory card must have its jumpers for both the Multibus and High Speed Bus interfaces set to the same address. If not, a node could access its own memory at two different addresses, only one of which would be valid for other nodes in the slice.

It is possible by convention to further restrict the Concert address space. One scheme which may be used in the future is to re-map the 32K PROM and I/O space of each node somewhere in the range 000000 - 07FFFF. This would leave all global registers and private resources in the bottom half megabyte of the address space. Since

eight megabytes is used by global memory, seven and a half megabytes (fifteen DBR50 boards) worth of node memory could be accommodated. This method requires the replacement of the mapping PROMs supplied with the DBC68K boards.

The address space division of the RingBus is shown in Table 7. It differs only slightly from the Multibus address space. The RingBus has twenty-four address bits, which address all global resources in the ring. If the high-order bit is "1," the reference is to global memory. The next three bits identify the destination slice and the remaining twenty bits address the byte within that slice's global memory. These bits are supplied by the RIB from the Multibus address. The second most significant bit of the RingBus address is taken from the highest-order address bit of the Multibus address, and the remaining twenty-two low-order bits are copied directly.

Locations			Allocated For
000000	–	00FFFF	Unallocated
010000	–	010FFF	Reserved for Global Registers
011000	–	7FFFFF	Unallocated
800000	–	8FFFFF	Global Memory for Slice 0
900000	–	9FFFFF	Global Memory for Slice 1
A00000	–	AFFFFF	Global Memory for Slice 2
B00000	–	BFFFFF	Global Memory for Slice 3
C00000	–	CFFFFF	Global Memory for Slice 4
D00000	–	DFFFFF	Global Memory for Slice 5
E00000	–	EFFFFF	Global Memory for Slice 6
F00000	–	FFFFFF	Global Memory for Slice 7

Table 7: RingBus Address Space

If the high-order RingBus address bit is "0," the reference is to a global register location. In this case, the RIB supplies the remaining twenty-three bits directly from the Multibus address. Currently only the 16K block of addresses from 010000 to

01FFFF is passed directly from the Multibus to the RingBus. Thus, any other RingBus address in this range is illegal and causes an *ABORT* operation.

4.4: Definition of the RingBus

The RingBus is the sole medium for communication among the slices of a Concert ring. It is a synchronous single-transaction bus, supporting memory read and write cycles. Each slice in a ring contains a RingBus segment. When all of the slices are connected together, a complete RingBus is formed. The addition of a central arbiter produces a Concert ring. Accesses to global resources in the ring occur on the RingBus, and are controlled by the arbiter. The arbiter orders the segments of the RingBus to be isolated or connected together to carry out these accesses. Since a Concert ring may have up to eight slices, it is possible that eight accesses to global resources may be occurring simultaneously.

Table 8 shows the signals on the Concert RingBus. The address space is 24 bits, partitioned as described in the previous section. The RingBus also has sixteen bidirectional data lines and eleven control lines. Nine of the control lines are driven by the *master*, the node which originated the request to the arbiter and which wishes to perform the operation. The other two are reply lines from the *slave*, the global memory or register upon which the operation is performed.

The control lines asserted by the master identify the nature of the operation. $\overline{R}$ and $\overline{W}$ indicate if a read or write operation is to be performed; only one may be asserted on a segment at one time. The size of the data to be operated upon is identified

Name	Description	Name	Description
A0	Address bit 0 (lsb)	D0	Data bit 0 (lsb)
A1	Address bit 1	D1	Data bit 1
A2	Address bit 2	D2	Data bit 2
A3	Address bit 3	D3	Data bit 3
A4	Address bit 4	D4	Data bit 4
A5	Address bit 5	D5	Data bit 5
A6	Address bit 6	D6	Data bit 6
A7	Address bit 7	D7	Data bit 7
A8	Address bit 8	D8	Data bit 8
A9	Address bit 9	D9	Data bit 9
A10	Address bit 10	D10	Data bit 10
A11	Address bit 11	D11	Data bit 11
A12	Address bit 12	D12	Data bit 12
A13	Address bit 13	D13	Data bit 13
A14	Address bit 14	D14	Data bit 14
A15	Address bit 15	D15	Data bit 15 (msb)
A16	Address bit 16	$\overline{R}$	Read operation
A17	Address bit 17	$\overline{W}$	Write operation
A18	Address bit 18	$\overline{GO}$	Begin operation
A19	Address bit 19	BYTE/WORD	Data type
A20	Address bit 20	$\overline{RMW}$	Read-Modify-Write
A21	Address bit 21	$\overline{ACK}$	Acknowledge
A22	Address bit 22	$\overline{ABORT}$	Abort operation
A23	Address bit 23 (msb)	S0	Source bit 0 (lsb)
		S1	Source bit 1
		S2	Source bit 2 (msb)

Table 8: RingBus Signals

by BYTE/$\overline{WORD}$. If the signal is not asserted, a byte operation is performed, with the byte selected by A0. Unlike the Multibus, the RingBus passes the data for an odd byte operation on the upper data lines. If BYTE/$\overline{WORD}$ is asserted, and A0 is low, a word operation is performed. BYTE/$\overline{WORD}$ and A0 both asserted is illegal, and causes an

ABORT operation on the RingBus.

If the memory operation is to be a read-modify write cycle, the $\overline{RMW}$ line is asserted by the master. This insures that no access occurs on the Multibus side of a global memory card while the RingBus is performing an atomic operation via the High Speed Bus. Three bits — $S2$, $S1$, and $S0$ — carry the number of the slice from which the request originated. These bits are used in conjunction with a protection register on the RIB to prevent unauthorized accesses to global resources.

A high-to-low transition on $\overline{GO}$ actually begins the operation. Thus, all address, data and control lines from the master must be stable before $\overline{GO}$ is asserted. The slave signals completion of the indicated operation by asserting $\overline{ACK}$. The result is the standard handshaking protocol used by countless asynchronous buses. Figure 12 summarizes this simple protocol.

In the event of a read-modify-write signal, the master asserts the $\overline{RMW}$ line in the read cycle before asserting $\overline{GO}$, and drops it at an appropriate point in the write cycle. The timing of all signals is intentionally left unspecified. The RingBus is primarily just a set of wires linking a master and slave; its characteristics depend upon the characteristics of the buses or modules to which it is attached.

If an attempt is made to address a "hole" in the global address space, the RIB in the destination slice asserts the $\overline{ABORT}$ line to force termination of the operation. When the source RIB sees this line asserted, it drops its request to the arbiter so that the RingBus segments are freed for use by other slices. A number of other memory errors also cause an *ABORT* cycle; Chapter 5 describes the *ABORT* function in more de-

Read Operation	Write Operation
Master drives address lines	Master drives address lines
Master drives source lines	Master drives sources lines
Master drives $\overline{WORD}$	Master drives $\overline{WORD}$
Master asserts $\overline{R}$	Master drives data line
Master asserts $\overline{GO}$	Master asserts $\overline{W}$
	Master asserts $\overline{GO}$
Slave performs read	
Slave drives data lines	Slave performs write
Slave asserts $\overline{ACK}$	Slave asserts $\overline{ACK}$
Master gets data	Master releases $\overline{GO}$
Master releases $\overline{GO}$	
	Slave releases $\overline{ACK}$
Slave releases data lines	
Slave releases $\overline{ACK}$	Master releases address lines
	Master releases source lines
Master releases address lines	Master releases data lines
Master releases source lines	Master release $\overline{WORD}$
Master release $\overline{WORD}$	Master releases $\overline{W}$
Master releases $\overline{R}$	

Figure 12: RingBus Operation Protocol

tail.

Although most lines on the RingBus change asynchronously, there are two important exceptions. Since RingBus signals propagate through buffers in each slice, it is difficult to ascertain the delays incurred. In particular, if the bus were purely asynchronous it would be difficult to insure that the $\overline{GO}$ and $\overline{ACK}$ signals arrive after the lines whose stability they are supposed to affirm. The solution chosen for Concert is to make $\overline{GO}$ and $\overline{ACK}$ synchronous. At the end of an arbiter cycle, data, address and control lines start propagating for a new request. After a sufficient period to account for any delays in these lines, the arbiter sends out a signal which latches $\overline{GO}$. This

same signal is used to synchronize $\overline{ACK}$ to signify the completion of a bus operation.

There are many features of standard buses which the RingBus does not provide. Since all arbitration is handled by a central arbiter, no arbitration lines are required. The RingBus does not support separate memory and I/O operations. It also does not directly support interrupts; these are generated by writing to memory-mapped interrupt registers.

Chapter 5 discusses the Ringbus interfaces provided on the RIB in considerable detail. However, several important features are essential. When a slice tries to access global resources, the RIB decodes the Multibus address, recognizes it as global, and sends a request to the arbiter. When the arbiter grants the request, it sends back signals which connect the slice Multibus to the RingBus segment. This allows the read or write transaction to complete. Once the Multibus lines drop, this in turn drops the request line to the arbiter. The next arbiter cycle lowers the grant lines, thus separating the RingBus from the Multibus and allowing the RingBus segment to be used by other slices.

Chapter 5: The Design of the RingBus Interface Board

The RIB was the most difficult piece of the Concert design, due to the number of functions it must perform. It provides global protection, monitor, and control registers and controls access to both these registers and to global memory in the slice. It also performs the arbitration on the slice Multibus and generates interrupts and resets. Figure 13 shows a fairly detailed block diagram of the RIB.

This chapter discusses each of the major blocks of the RIB in detail. No attempt is made to discuss the design at the gate level. Instead, the basic concepts behind the different function are discussed, along with rough sketches of the hardware used. An attempt is also made to point out any particularly difficult or novel aspects of the design.

5.1: Global Registers

The global memory boards account for the bulk of the global resources available in a slice. However, the RIB provides twelve control registers which are also globally accessible. All accesses to the global registers from any node must pass through the RIB and be cleared by the arbiter. The reason is simple; unlike the global memory, the global registers are not dual-ported and cannot be accessed directly on the Multibus by nodes within the same slice. This section describes the function and implementation of each of the global registers.

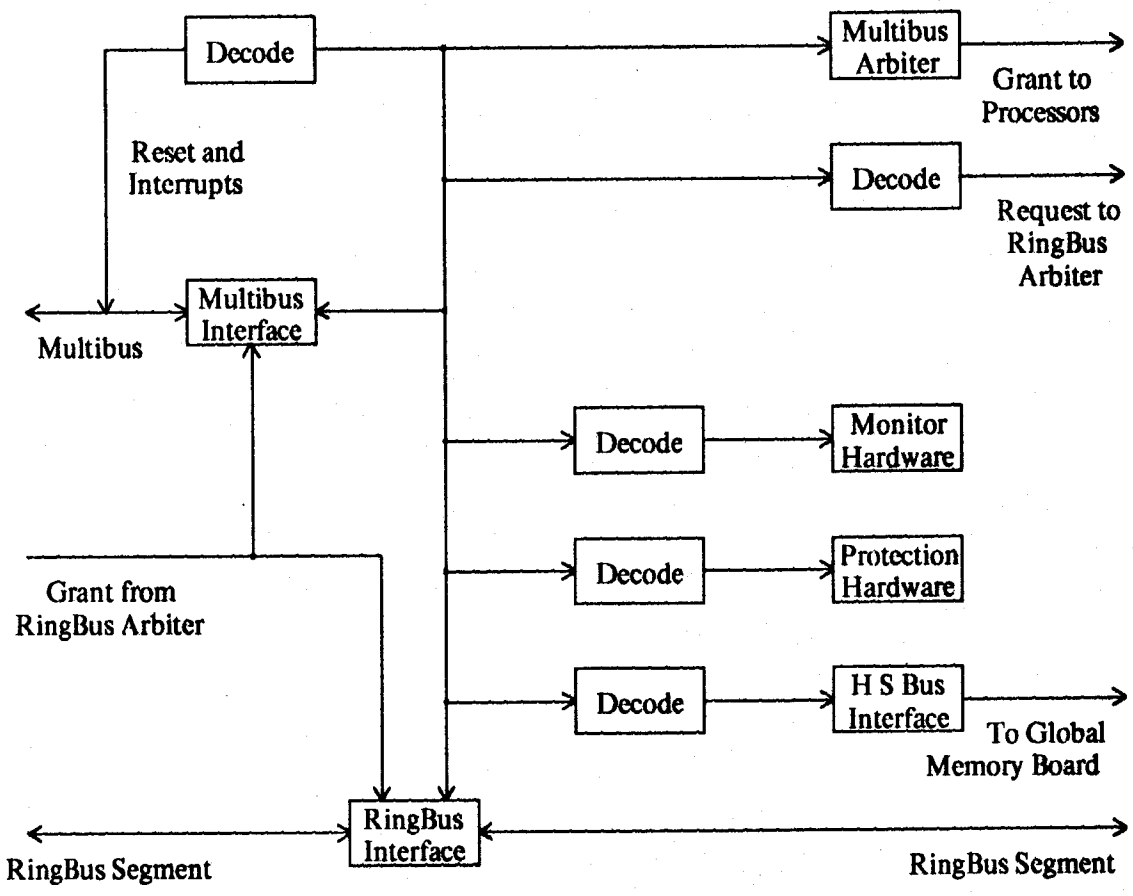


Figure 13: RingBus Interface Board Block Diagram

5.1.1: The Slice Reset Register

The Slice Reset Register (SRR) provides a means to reset all the nodes in the slice. The SRR resets the nodes by strobing the *INIT** line on the slice Multibus. The definition of the bus makes it impossible to individually reset nodes. The main purpose of the reset capability is to put all nodes in a slice in a known state at system initializa-

tion time.

Writing *any* value to the SRR causes the reset function to occur. Thus, it does not have to be implemented as a true register. Instead, it is a simple flip-flop which gets set by a write to the appropriate location. The SRR is automatically cleared at the end of its reset cycle. The SPR is *not* protected by the Slice Protection Register, so that a reset may be performed regardless of the state of the slice.

5.1.2: The Node Interrupt Registers

The RIB contains a Node Interrupt Register (NIR) for each of the eight nodes possible in the slice. Although the DBC68K processor board contains internal interrupts for the serial port, timers, and other peripherals, the NIR provides the only way to interrupt a processor from outside the node. Each interrupt level on a slice Multibus is associated directly with one of the nodes in the slice, and writing any value to the appropriate NIR generates an interrupt by asserting the Multibus interrupt line. Each NIR, like the SRR, is implemented as a single flip-flop rather than as a true register.

The Multibus interrupts are vectored, so the RIB must place an interrupt vector on the bus in response to the proper interrupt acknowledge. This vector is the same for all NIRs except for the lowest three bits, which identify the node for which the interrupt is destined. All information needed by the interrupt handler, such as interrupt type and interrupt source, is passed in a control block located in global memory. An NIR is reset when the proper interrupt acknowledge is sent on the Multibus. All NIRs are also cleared when a slice is initialized by writing to the SRR.

There are only eight interrupt lines on the Multibus, and Concert allows as many

as eight nodes in a slice. If a Multibus board which generates interrupts is included in a slice, it is probably desirable to disable the NIR which drives the interrupt request line it uses. A set of switches is provided on the RIB to allow selective enabling or disabling of the eight NIRs.

5.1.3: The Slice Protection Register

The Slice Protection Register (SPR) protects the global resources in the slice from unauthorized access. Its format is shown in Figure 14. Setting the appropriate bit disables read or write access by a particular slice to the global registers and the global memory of the RIB's slice. The SPR may also be read, which allows selective setting of its bits by ORing with the current value. However, the SPR is generally set at system initialization and not altered during the normal course of operation.

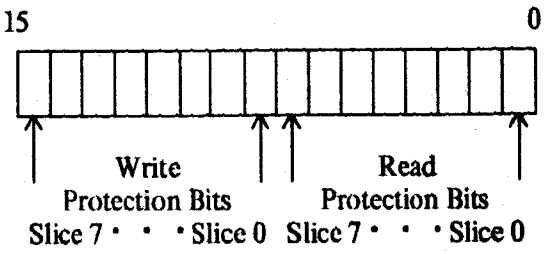


Figure 14: Slice Protection Register

The SPR is provided for two main reasons. Like any protection scheme, it supports a measure of software robustness by not allowing erroneous programs to access areas of memory they shouldn't. It also allows the the global memory space to be partitioned into separate spaces for multiple processes running in a single ring. In fact, it

is possible to have two independent applications running on the same ring, each having its own dedicated set of processors and blocks of global memory.

The only global resource not protected by the SPR is the Slice Reset Register. Although this leaves a hole in the protection scheme, it is necessary to insure that a reset operation may be performed regardless of the state of the slice. The SPR is designed to protect against faulty programs more than maliciousness, and the chance of an accidental access to the SRR is probably quite slim.

The protection specified by the contents of the SPR is enforced rather easily. When the RIB receives a read or write request over the RingBus, the signals *S2*, *S1* and *S0* contain the source of the request. It examines the read or write protection bit for this source. If that bit is clear, the access proceeds normally. If that bit is set, it refuses to continue the access and causes an *ABORT* cycle on the RingBus. Figure 15 gives an overview of the implementation of the protection function. The SPR is reset by the slice reset function, thereby enabling access from all slices.

5.1.4: Support for Hardware Monitoring

Early in the design of Concert Tom Sterling suggested that the system include hardware to monitor and gather statistics about its operation. This allows real-time measurements, a feature rarely found in computer systems of any kind. Two approaches were taken to facilitate this function. The RIB includes two global registers to support monitoring — the Slice Monitor Register (SMR) and the Slice Monitor Counter (SMC). In addition, interesting control signals on both the RIB and the arbiter

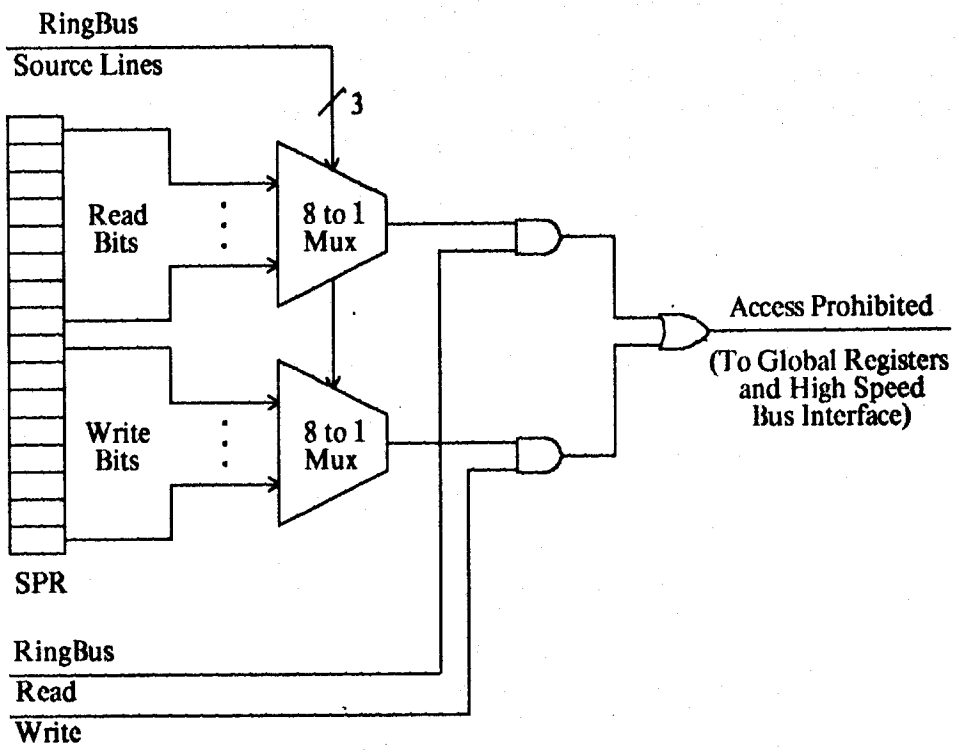


Figure 15: Hardware Protection Support

are brought out to edge connectors for external monitoring and logging.

The SMC is a 16-bit counter which logs events on the slice Multibus. It does so under the control of the SMR, a 16-bit register whose format is shown in in Figure 16. The SMR can monitor the slice operation by controlling the SMC. Both the SMC and the SMR may be read or written from the RingBus. Since these registers are global, it is possible for a single node to monitor and gather statistics for all slices in the system.

The primary function of the SMR is to select one of eight *functions* of various control lines, as shown in Figure 17. The functions available are determined by an EPROM, which allows the user to monitor the combinations of signals most interesting

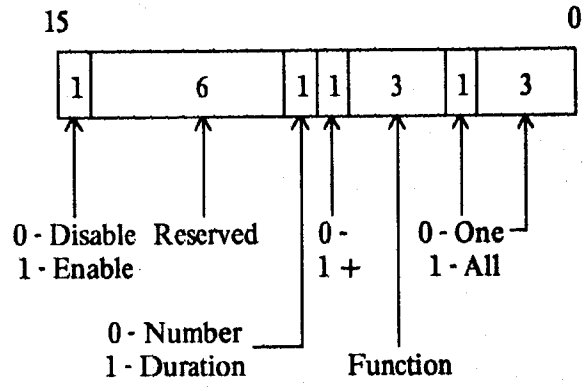


Figure 16: Monitor Control Register Format

for the particular application. The signals which are fed into the EPROM include the Multibus request, read, write and lock lines. By using appropriate functions of these lines, it is possible to determine the state of the Multibus. The remaining inputs to the EPROM are the request line from the slice to the arbiter, designated *REQ*, and the grant line from the arbiter back to the slice, designated *GRT*. These may be used to differentiate global from local accesses.

The SMR causes the SMC to operate in one of two modes, as determined by the *Number/Duration* bit. The SMC may be incremented whenever the selected function makes a particular transition, as selected by the +/– bit. The SMC may also be incremented on *BCLK**, the Multibus arbitration clock, whenever the function value is high or low. The +/– bit selects between the high and low values. Thus, the SMC can either count the number of transitions a function value makes or the duration of a particular value.

The SMR includes a bit which specifies whether the SMC is incremented when the selected function is encountered while *any* node is the Multibus master, or only when a

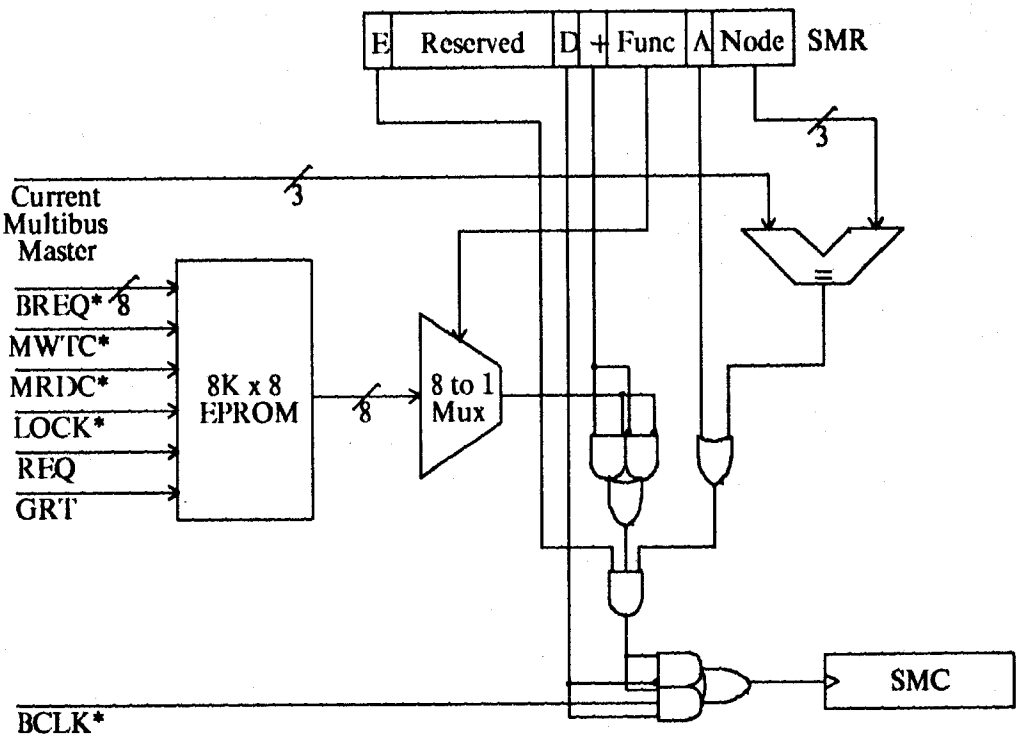


Figure 17: Hardware Monitor Support

specific node is master. In the latter case, a three-bit field identifies the node. This allows the monitoring of the activity of a specific node only. The SMR field is compared against the current Multibus master, which is supplied by the Multibus arbitration logic on the RIB. Finally, the SMR contains a bit to enable or disable the monitor function; this is used to stop and start counting.

In addition to the hardware on the RIB board, a variety of signals is brought out to edge connectors on both the RIB and the arbiter. The purpose of these signals is twofold. First, it is possible to dedicate a node to the task of gathering statistics for a RingBus or for a slice Multibus, simply by feeding these signals into the parallel ports

on the processor board. If the processor makes all its references to a local memory board, and no other nodes in the slice use that memory, then it can monitor the system without perturbing it in any way. The edge connector signals are also useful for debugging boards; they provide handy places to tap key signals.

5.1.5: Global Register Addresses

The decoding of global register RingBus addresses is a bit more complicated than for global memory. In the case of the memory, address bits 23, 21 and 20 of the Multibus address are mapped into bits 22, 21 and 20 of the RingBus address. These bits specify the destination slice, and the remaining twenty bits address the global memory within the slice. The RingBus global register addresses are taken directly from the Multibus addresses, but they comprise a much smaller block of space — only 64K. Of this, only a very small amount is presently used.

From the address space viewpoint, the global registers are divided into two classes, differentiated by address bit 15. If this bit is low, the register is a *per slice* global register, meaning that there is one such register for each slice. The next three address bits identify the slice, and the remaining 12 address the register. If address bit 15 is high, the register is *per node*. The same three bits identify the slice, but the next three bits after that identify the node within the slice.

Table 9 shows the locations of the global registers, with all addresses given in binary. The letter *s* is used to signify the three-bit field identifying the slice. Likewise, *n* is used to identify the node field. Word (two-byte) registers may be addressed as words or as bytes, depending upon the values of *A0* and $\overline{WORD}$. Byte registers are

addressed only if $\overline{WORD}$ is high and A0 is low. All "holes" in the address space are illegal and attempts to address them generate *ABORT* cycles on the RingBus.

Locations (Binary)						Allocated For
0000	0001	0sss	0000	0000	0000	Slice Reset Register (byte)
0000	0001	0sss	0001	0000	0000	Slice Protection Register (word)
0000	0001	0sss	0010	0000	0000	Slice Monitor Register (word)
0000	0001	0sss	0011	0000	0000	Slice Monitor Counter (word)
0000	0001	1sss	nnn0	0000	0000	Node Interrupt Register (byte)

Table 9: Global Register Address Space

It may seem unusual to use high address bits to differentiate between the registers, but there is a good reason for this decision. The 68451 memory management unit has a 256-byte protection granularity. Since all global registers currently implemented can be differentiated by address bits outside this range, it is possible to protect them separately if desired. In particular, it may be desirable to allow anyone access to the monitor registers but to guard the interrupt and protection registers more closely.

5.2: Access Control

The primary function of the RIB is to provide access to global memory and registers under control of the RingBus arbiter. This involves, among other things, interfacing with the Multibus, the RingBus and the High Speed Bus. These interfaces allow the RIB to handle three distinct kinds of accesses. It must allow a node within its slice to make a global access. Such an access may be either to global registers within the slice or to global resources in another slice. It must also handle requests for global resources within the slice which originated in other slices in the ring. Any of these

three situations entails interaction with the RingBus arbiter.

5.2.1: Basic Requirements

If the Multibus is asserting an address which corresponds to the global registers within the same slice, or to global registers or memory in another slice, a request must be sent to the arbiter. Four lines are required to do this. $\overline{REQ}$ informs the arbiter that the slice wishes to access global resources. Three additional bits, *DST*2, *DST*1 and *DST*0, are passed to the arbiter to indicate the destination slice for the request. If the request is for global memory, address bits 23, 21, and 20 identify the slice. If the reference is to global registers, bits 14, 13, and 12 are sent to indicate the destination. Note that the source slice is a valid destination if the request is for a global register.

The arbiter eventually grants the request by sending back lines to control the flow of information on the RingBus segments. Although these lines represent the requests which have been granted, the term "grant" is actually somewhat of a misnomer. The lines are actually enable signals which tell the RIB how to connect the resource access paths. Some combinations of signals connect the Multibus to the RingBus and allow an access to occur. Other combinations simply cause the RIB to propagate requests from other slices in the ring. For this reason, the control lines from the arbiter to the RIB are termed *enable lines*.

When the arbiter grants the slice's request, it sends back enable signals to connect the Multibus to the RingBus. It must also send out the appropriate enable signals to connect together the RingBus segments needed to complete the access. It must

also send enable signals to the destination slice to allow the request to reach the global memory or register for which it is intended.

5.2.2: Access Path Options

The most interesting aspect of the RIB design was determining the access paths to provide. Figure 18 shows a simple model of the input and outputs of the RIB with all access paths which might be reasonable to include. The paths are drawn with arrows pointing from the source of the access request (master) to the destination resource (slave). Since the local resources on the Multibus are not accessible from outside the slice, the paths from the RingBus segments to the Multibus are not required. Another way of phrasing this is that the RIB is never a Multibus master.

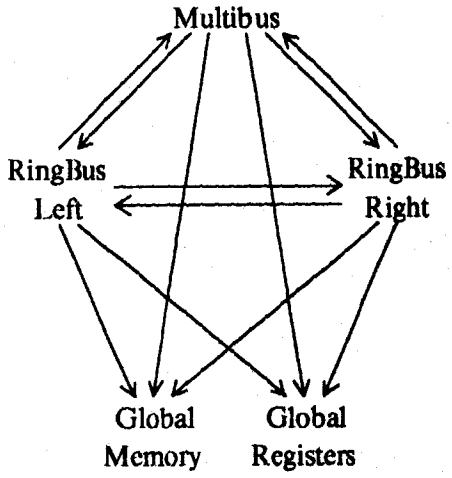


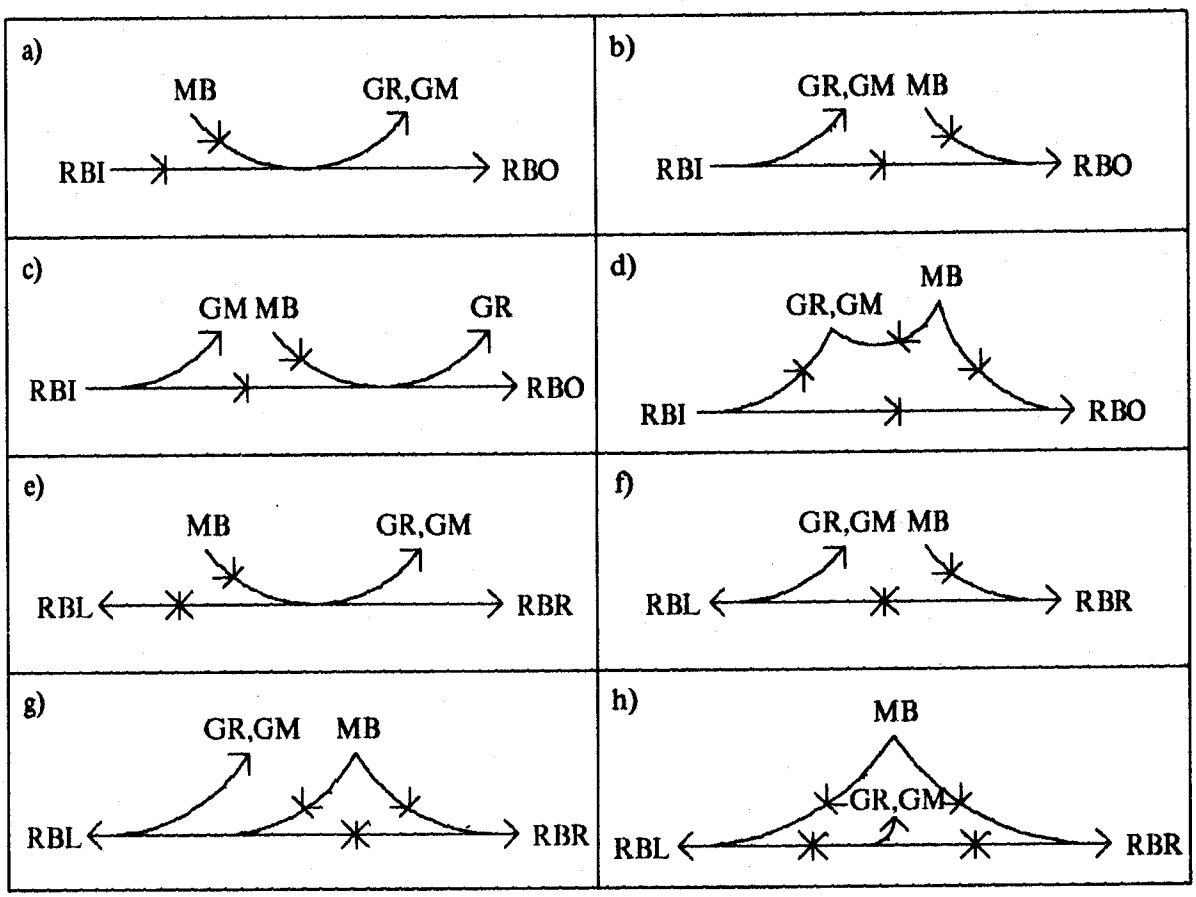
Figure 18: Possible Access Paths in the RIB

The remaining paths all support various kinds of legal accesses under the Concert RingBus architecture. Depending upon the degree of flexibility desired, many of these

paths may not be provided. Flexibility is traded off against the extra hardware required on the RIB to support the paths and the extra logic required in the arbiter to take advantage of the flexibility. One path is eliminated by the use of DBR50 boards for global memory. Since the Multibus can access the global memory within the slice directly, the path from the Multibus to the global memory is not required. In fact, this path *cannot* be provided or both the High Speed Bus and Multibus would try to make accesses to global memory for a single request.

The crucial issue for deciding which of the remaining paths to support is the directionality of the RingBus. If the RingBus is unidirectional, then one RingBus segment is used only for incoming requests and the other only for those outgoing. Thus, the Multibus has a path to only one RingBus segment, and only one RingBus segment has access to the global memory and registers. Likewise, the path connecting the two segments is unidirectional. Although a unidirectional RingBus requires fewer access paths than a bidirectional one, there is less parallelism to be exploited.

A wide variety of options for the resource access paths were considered during the design of the RIB. Figure 19 shows eight of the most reasonable in schematic form. The "diode" symbols represent tri-state drivers, with the arrows indicating the direction of address and control flow. "MB" indicates the slice Multibus, "GM" the global memory within the slice and "GR" the global registers on the RIB. For a unidirectional RingBus, "RBI" means the "RingBus In" segment and "RBO" the "RingBus Out" segment. If the RingBus is bidirectional, both segments can carry either inbound or outbound requests, and "RBL" for "RingBus Left" and "RBR" for "RingBus Right" are used to differentiate them.



MB = Multibus; GM = Global Memory; GR = Global Register

RBI = RingBus In; RBO = RingBus Out

RBL = RingBus Left; RBR = RingBus Right

Figure 19: Possible RIB Access Path Designs

Figure 19a is the simplest case to consider. It uses a unidirectional RingBus, and essentially functions as a multiplexor. At any point in time, either the RingBus In or the Multibus may be connected to the resource access paths. These paths provide

connections to the global registers on the RIB and, via the High Speed Bus, to the global memory in the slice. A request from either the RingBus In or the Multibus may be connected to the RingBus Out and thus propagated to another slice further along the ring.

A single enable line from the arbiter is required to select which of the two sources is connected to the access paths. It is not necessary to provide for the case when both sets of drivers are off. No harm results if a request from either the RingBus In or the Multibus reaches a resource for which it is not destined. A single global address space means that only one resource ever responds to a given RingBus address.

The scheme shown in Figure 19b adds a level of parallelism while requiring no additional hardware. It is possible for the RingBus In to be accessing global resources in the slice at the same time that a request from the Multibus is being sent along the RingBus Out. This requires a slightly different arbiter design than the method of Figure 19a, but a single enable line still suffices.

The inherent parallelism of this scheme gives it some advantage over the first design. However, there is one drawback which offsets some of this gain. The Multibus can only access the global registers through the RIB; the cost of dual-porting the registers is too great to justify. However, in the scheme of Figure 19b, a node requires the entire RingBus to access the global registers in its own slice. Presumably such accesses are very infrequent compared to global memory accesses, but the cost of tying up all the RingBus segments is considerable and should be avoided if possible.

Figure 19c shows one possible way to remedy this situation. The global registers and global memory are separated, so that the Multibus can access the global registers

in its slice with a single RingBus segment. It is necessary to differentiate between requests for global memory and requests for global registers, although this may be handled easily. The RIB can simply increment the destination value it sends to the arbiter if the request is for global registers to reflect the fact that the registers are located one segment beyond the memory in each slice.

The method of Figure 19d exploits the maximum parallelism for a unidirectional bus. As with the the second scheme, the RingBus In may be accessing global resources at the same time that the Multibus is connected to the RingBus Out. In addition, the Multibus can access global registers while the two RingBus segments are connected. Unfortunately, it takes double the amount of drivers to provide this extra parallelism. With the assumption that accesses to global registers are infrequent, it doesn't seem worth the extra hardware.

The simplest access path design which supports a bidirectional RingBus is shown in Figure 19e. It is very similar to the first scheme, except that the drivers linking the two RingBus segments are bidirectional. Thus, requests from the Multibus can propagate in either direction, and requests can arrive from either direction. As long as the arbiter is clever enough to take advantage of this feature, the throughput on the RingBus will be increased considerably. However, no parallelism within the RIB is supported.

The same transformation that produced the second scheme from the first is used to turn the design of Figure 19e into Figure 19f. In terms of hardware, it is only a minor variation on the second proposal, with the drivers between the RingBus segments made bidirectional. This solves the problem of global register access fairly well;

a Multibus access to the registers in the same slice requires only two RingBus segments.

Figure 19g shows a minor variation on this design, in which the Multibus can directly access the RingBus Left without tying up the RingBus Right. This has the effect of reducing by one the number of segments needed for many accesses. The cost is an additional set of drivers to connect the Multibus to the RingBus Left.

Both of these two schemes fully support a bidirectional RingBus. They also allow some parallel accesses, although they are asymmetric. The RingBus Left can access the global resources at the same time that the Multibus accesses the RingBus Right, as was the case in the second scheme. However, Multibus access to the RingBus Left and RingBus Right access to the global resources are mutually exclusive. Because of this asymmetry, there is a "preferred direction" (clockwise or counterclockwise) for access on the RingBus. If desired, the arbiter may take this into account when granting requests.

Figure 19h shows a way to rectify this asymmetry and gain more parallelism on the RingBus. Unfortunately, it requires still more drivers and thus greater hardware cost. More importantly, it also requires a more complicated arbiter design. It is no longer sufficient for the arbiter to keep track only of the RingBus segments needed for an access; it must also worry about the access path to the global resources between the segments as well.

After considering all these options, the access path design of Figure 19f was chosen for the Concert RIB. It supports a bidirectional RingBus and some parallelism in the RIB with a minimum of hardware. It also does not require the arbiter or the RIB

to differentiate between accesses to global memory and accesses to global registers, or to keep track of more than the RingBus segments in use.

5.2.3: The Arbiter Interface

Given the access paths as shown in Figure 19f, it is relatively easy to define the exact interface between the arbiter and the RIB. A single request line and the three bits indicating the destination slice are sufficient information for the arbiter to make its decision. It sends back three lines to the RIB to enable the drivers. The request and enable lines are all active-low and pulled up, so that unconnected lines do not cause spurious bus cycles.

The three lines from the arbiter to the RIB are $\overline{ENM}$, $\overline{ENL}$, and $\overline{ENR}$. $\overline{ENM}$ indicates if an access on the Multibus is allowed to propagate, and is used to enable the drivers which connect the Multibus to the RingBus Right. $\overline{ENL}$, when asserted, enables the drivers which connect the RingBus Left segment to the RingBus Right segment. This allows accesses to propagate from left to right. Likewise, $\overline{ENR}$ enables the drivers in the opposite direction to permit right-to-left accesses. Table 10 summarizes the interpretation of these lines, showing their active-high values for clarity.

Different interpretations of the enable lines are possible. However, as seen in Table 10, there are five different states which must be encoded and therefore any scheme would require three lines. The interpretation chosen is straightforward and requires a minimum of logic in the RIB. None of the enable lines tells the RIB for certain if the global resources in its slice are being accessed. In fact, such access can occur

ENM	*ENL*	*ENR*	Meaning
0	0	0	No global access in this slice.
0	0	1	Right-to-left access on RingBus; no access from Multibus.
0	1	0	Left-to-right access on RingBus; no access from Multibus.
0	1	1	ILLEGAL - Arbiter will never output.
1	0	0	Multibus access to RingBus Right.
1	0	1	Multibus access to RingBus Left.
1	1	0	ILLEGAL - Arbiter will never output.
1	1	1	ILLEGAL - Arbiter will never output.

Table 10: Enable Signals from the Arbiter to the RIB

in any but the "illegal" states. This is consistent with any standard bus, since it is the bus address which tells a resource when it is being accessed.

5.3: Access Support

The access path model presented in the previous section glosses over the implementation details of the drivers which link together different buses. There are four such interfaces which are provided. The two RingBus segments are connected together, and Multibus cycles are mapped into RingBus Right cycles. In turn, interfaces are provided from the RingBus Left to the High Speed Bus for global memory access and to the RIB registers for global register access. In addition, support is provided for atomic and *ABORT* operations on the RingBus.

5.3.1: Bus Interfaces

The simplest interface to design is that connecting the left and right segments of the RingBus. The address and control lines are simply propagated in the proper direction from segment to segment, as determined by $\overline{ENR}$ and $\overline{ENL}$. The data buffers are only slightly more complicated. Data is sent from the right RingBus segment to the left segment in the event of a right-to-left write or a left-to-right read. Similarly, data flows from left to right on a left write or a right read.

The Multibus-to-RingBus interface is a bit more complicated. All buffers are enabled by the $\overline{ENM}$ signal from the arbiter, as described in the previous section. Since the address lines on the Multibus are active-low, they must be inverted to produce the active-high RingBus address lines. The 22 low-order address bits are taken directly from the inverted Multibus lines. As described in Chapter 4, the mapping from the Multibus to the RingBus address space requires that the second-highest-order RingBus address bit be generated from the inverse of the highest-order Multibus address bit. Finally, the two highest-order Multibus address bits are XORed to produce the highest RingBus address bit.

The data lines must also be inverted, and the direction of the buffers is determined by whether the Multibus is requesting a memory read or write operation. The "byte-swapping" function on the Multibus mandates that odd byte transfers use the lower eight data lines, and so an extra set of drivers is required to accommodate this function. The RingBus $\overline{GO}$ signal is asserted when either the read or write command is asserted on the Multibus. The RingBus source bits are generated from the appropriate

Multibus address lines as described in Chapter 4. The remaining RingBus control lines — $\overline{R}$, $\overline{W}$, $\overline{WORD}$, $\overline{RMW}$, and $\overline{ACK}$ — are all generated directly from the corresponding Multibus signals. The $\overline{ABORT}$ line is discussed in a later section.

The Microbar High Speed Bus is a fast bus designed primarily for use with their DBC68K [13] and DBC86 [14] processor boards. However, the RIB accesses global memory boards by means of this bus, and thus contains a RingBus-to-High Speed Bus interface. The High Speed Bus uses multiplexed address and data lines, and most of the hardware in the interface is used to perform the multiplexing and demultiplexing functions. The High Speed Bus address and data lines are all active-low, so the RingBus lines are inverted during the multiplexing process. The RingBus control signals are mapped into read/write, lock, byte enable and strobe lines to produce the proper results.

The most unusual aspect of the High Speed Bus is that within a fixed period after a memory access it sends back a signal indicating if the address was found on the memory board or not. If this line, *-MYOK*, is asserted, the operation completes and the acknowledge is sent back along the RingBus to the master. If this line is not asserted within the specified time, an *ABORT* operation is begun on the RingBus.

Support for the global registers is not much more complicated, although the address decoding requires more hardware. In the event of a read operation, the RingBus data is latched into the addressed register. On a read, the register contents are driven onto the RingBus. The acknowledge signal is generated by the RIB after a sufficient delay to allow the operation to complete. Operations to the global registers happen synchronously with respect to *LCLK*, the signal which the arbiter sends to all RIBs to

latch $\overline{GO}$ and $\overline{ACK}$.

5.3.2: Support for Atomic Operations

Since semaphores are used extensively to control access to data structures in global memory, all buses in Concert must support an atomic *test-and-set* (or *read-modify-write*) operation. The 68000 has a special TAS instruction which asserts a particular combination of output pins to indicate that an atomic read and write is needed. The DBC68K board decodes these lines to identify this state, and then asserts the *LOCK** signal on the slice Multibus. By monitoring this line, the RIB can determine when an atomic operation is being requested.

The RIB insures an atomic access by not lowering its request line to the arbiter as long as the *LOCK** line on the Multibus remains asserted. The arbiter does not know that an access has been completed until $\overline{REQ}$ is released, so it sees the test and set as a long single memory operation. This trick allows read-modify-write cycles without passing extra lines to the arbiter. However, since the arbiter sees the atomic operation as a single cycle, it cannot allocate any new RingBus segments in the middle. A node can only access locations within the same slice for the duration of the atomic operation; a generalized lock function such as that allowed on the Multibus is not supported. Fortunately, the DBC68K board asserts the *LOCK* line only during the TAS instruction, which operates on a single memory location.

The Multibus *LOCK** line is propagated along the RingBus as $\overline{RMW}$ to the RIB in the destination slice. The RIB passes $\overline{RMW}$ to the global memory as the High Speed

Bus signal *-HSLOCK*. This signal remains asserted for the duration of the atomic operation, insuring that no access occurs from the Multibus interface. Likewise, a read-modify-write access to global memory from within the same slice asserts the Multibus *LOCK** line, which forestalls High Speed Bus accesses until the cycle is complete.

5.3.3: Abort Operations on the RingBus

Unlike most buses, the RingBus provides a means to abort cycles in the event of memory errors. The $\overline{ABORT}$ signal is driven by the RIB in the slave's slice and sent back to the RIB in the slice of the master. When $\overline{ABORT}$ is asserted, the master RIB stops driving the request line to the arbiter. Thus, the RingBus segments are freed for use by other slices. There is no way to abort cycles on the Multibus, and so the source node must detect a Multibus timeout before it knows that an error has occurred. However, the timeout is restricted to one slice, and the rest of the ring is unaffected.

There are a number of events which cause an RIB to abort the RingBus cycle. Any reference to an unallocated portion of global register space, an illegal word address ($\overline{WORD}$ asserted and *A0* high) or a protection violation (as signalled by the SPR) cause an *ABORT* operation. An abort is also sent if the global memory in the destination slice signals a parity error by asserting *-HSBERR* on the High Speed Bus or does not respond within the specified time by asserting *-MYOK*.

5.4: Multibus Arbitration

The final function performed by the RIB is the arbitration of the Multibus backplane. There are two different ways to provide Multibus arbitration [9]. *Serial arbitration* is the simplest, but it is limited to a three-master bus. Since a Concert slice may have up to eight nodes, the RIB uses the other scheme, *parallel arbitration*.

The discussion in Chapter 3 on RingBus arbitration schemes also has considerable relevance for the Multibus. The two most common schemes for Multibus arbitration are fixed priority and some sort of rotating priority. The RIB uses a variation of the rotating priority scheme discussed in Chapter 3. As shown in Figure 20, the only hardware necessary to perform this arbitration is a ROM, a decoder, and a three-bit register.

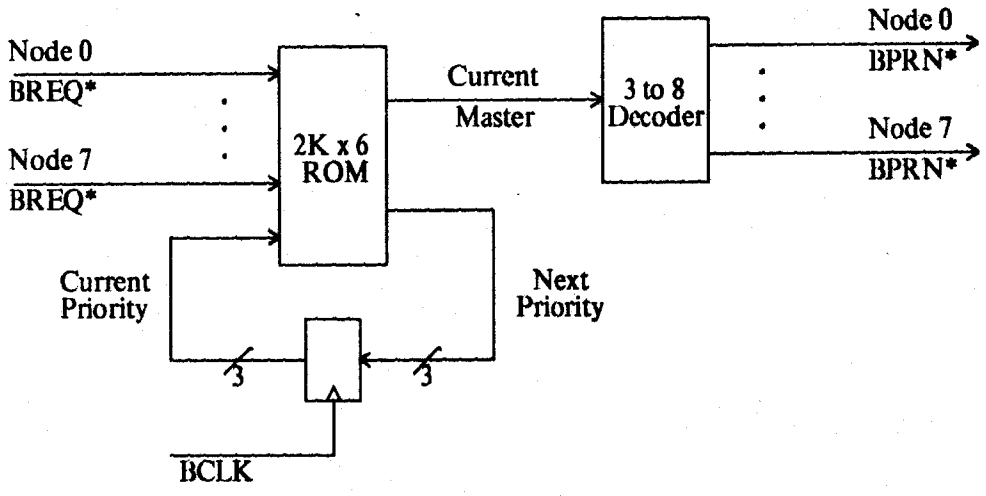


Figure 20: Multibus Parallel Arbitration Scheme

The register holds the number of the current node having top priority. The ROM takes as inputs the value of the counter and the nodes currently requesting the Mul-

tibus. It outputs two three-bit values — the current master and the next top priority value. The three bits identifying the current bus master are decoded and sent to the processor boards as bus grant signals. Only one of these is asserted at a time.

The ROM determines the current bus master and the next priority value by counting down (modulo eight) from the top priority node. It grants the Multibus to the first node it finds which is making a request, if any, and sets the next top priority to be the number of the second such node it finds. If no nodes are requesting the Multibus, the current master retains the bus and the priority is incremented by one modulo eight.

Chapter 6: The Design of the RingBus Arbiter

The RIB was the most difficult portion of the Concert design, because of the number of different functions it has to serve. The other major portion of the design was the RingBus arbiter. Although its complexity does not approach that of the RIB, the range of arbitration schemes possible made its design an interesting task. This chapter outlines the design and indicates some options which were investigated during the design process.

6.1: Overview of the Arbiter

The arbiter is responsible for controlling all transactions on the RingBus and for explicitly granting all accesses to global resources *except* accesses to global memory within a slice. The arbiter accomplishes this task by interacting with the RIB of each slice in the ring. When the RIB identifies a request for global resources by decoding the address lines on the Multibus, it must pass this request to the arbiter. The arbiter synchronously samples the requests from all the slices and, at the end of its cycle, sends back enable lines to the RIBs.

As described in Chapter 4, Concert allows up to eight slices per ring, and eight nodes per slice. The number of nodes in a slice has no effect on the arbiter, since it only "sees" one request per slice at a time. Each slice RIB provides local Multibus arbitration to determine which node is master, and thus which node's request for global

resources is passed to the arbiter.

The number of slices in a ring does have considerable impact upon the arbiter design. At a minimum, it defines the number of arbiter inputs and outputs. In fact, the number of slices whose requests must be examined and granted also affects the internal arbiter logic as well. This chapter discusses the design of an arbiter to handle up to eight slices. However, all the ideas discussed can be conceptually (if not always practically) extended for a ring with an arbitrary number of slices.

The RIB design chosen, as described in the last chapter, requires seven lines between each RIB and the arbiter. Each RIB sends a request line and three destination bits to the arbiter, and the arbiter in turn sends back three enable lines to each RIB. In fact, the arbiter also sends the *LCLK* signal to each RIB to latch the RingBus $\overline{GO}$ and $\overline{ACK}$ lines. However, this signal is simply derived from the arbiter clock and has no relation to the rest of the arbiter.

The analysis of the RingBus architecture in Chapter 3 showed that a combinational arbiter which maps the request lines to the enable lines is not sufficient. The arbiter is better modeled as a finite-state machine, with state representing both requests in progress and the priority of pending requests. Figure 21 shows a top-level view of the Concert arbiter, incorporating both kinds of state information. The request and enable lines are shown as active-high, although they are actually inverted between the RIB and the arbiter.

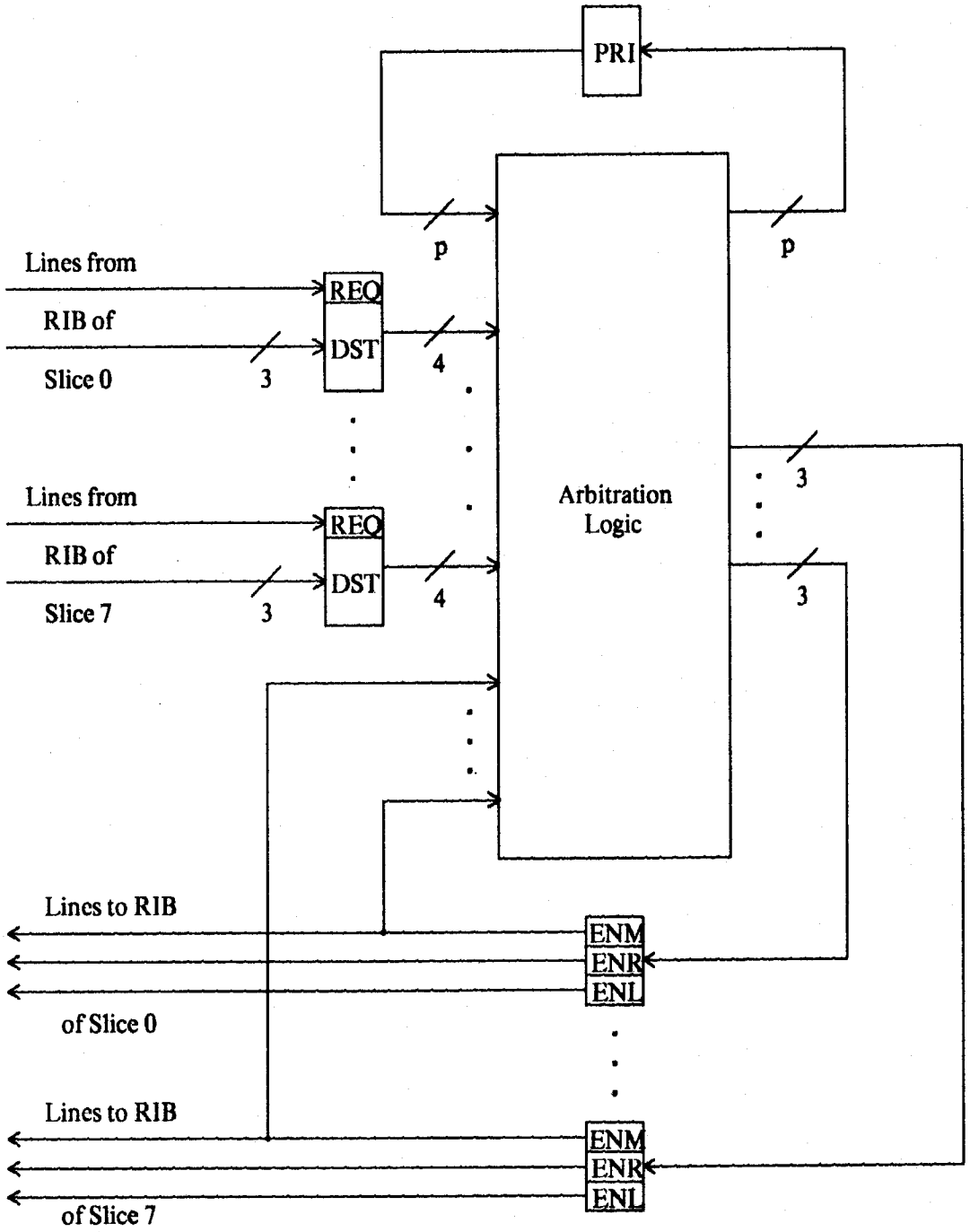


Figure 21: Top-Level View of the Concert Arbiter

Only one of the three enable lines — *ENM* — needs to be fed back into the arbiter. Since *ENM* is asserted for a slice only if its request is granted, the values of *ENM* are all the arbiter needs to determine which requests have been granted in a previous cycle. The number of lines necessary to encode the priority has been designated p to encompass a variety of possible schemes. The remainder of this chapter describes the decisions made on the priority scheme and other aspects of the arbiter, and the design process which led to these decisions.

Three major goals guided the arbiter design. The arbiter must grant all requests and allow accesses to complete without interruption. It should be fair, giving all slices an equal shot at the global resources. Finally, the arbiter should allow as many parallel operations on the RingBus as possible. After all, the whole purpose of using the RingBus architecture instead of a more traditional shared-bus scheme is to allow simultaneous accesses to different pieces of global memory.

6.2: Examining the Requests

Before deciding which requests to grant, there are several functions the arbiter must perform. The first is to determine whether two requests can be simultaneously granted. Unless it is possible to economically determine simultaneously grantable requests, then only one request at a time can be granted and the advantage of the RingBus structure is lost.

Chapter 3 described the "Segment Needed List" (SNL), which identifies the RingBus segments needed to grant a particular request. Since requests may be simultaneously granted if and only if they require no common RingBus segments, these lists

are all that the arbiter requires to determine simultaneously grantable requests. Table 11 shows the round of requests used in the example of Chapter 3. The RIB design chosen differs significantly from the simple model presented then. Thus, the SNLs for the round of requests differs as well.

Source	*REQ*	*DST*
0	0	x
1	1	3
2	1	4
3	0	x
4	1	7
5	1	5
6	1	3
7	0	x

Table 11: Round of Requests from Chapter 3

Figure 22 shows a Concert ring with RingBus segments numbered. Since the Concert RingBus is bidirectional, any request may be granted either clockwise or counterclockwise. Table 12 shows the SNLs for each direction for the round of requests of Table 11. Because of the RIB access path design chosen, there are some curious anomalies. The segment associated with a slice is the segment on which the global resources reside. Since Multibus requests connect directly to the next segment, the source segment is not needed to carry out a clockwise access.

If the source and the destination are different, the SNL for a clockwise access is nothing more than the set of integers from the source slice plus one to the destination slice, counting up modulo eight. A counterclockwise access requires the segments from the source plus one to the destination, counting down modulo eight. If the source and the destination slice are identical, the request is for access to global regis-

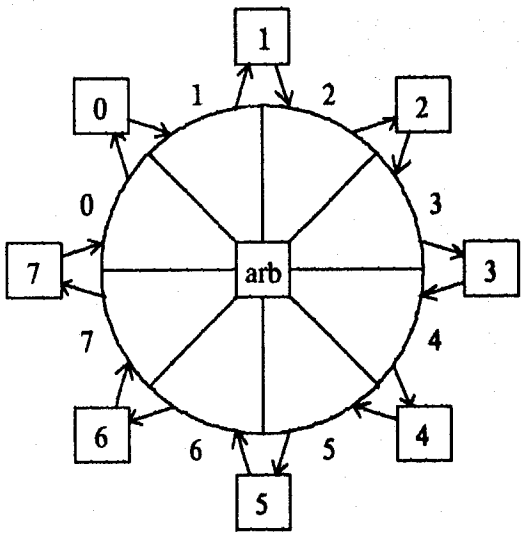


Figure 22: Concert Ring with Slice and Segment Numbers

segment	Clockwise Access								Counterclockwise Access							
request	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1→3			x	x					x	x	x	x	x	x	x	x
2→4				x	x				x	x	x	x	x	x	x	x
4→7						x	x	x	x	x	x	x	x	x		x
5→5	x	x	x	x	x	x	x	x						x	x	
6→4	x	x	x	x	x			x					x	x	x	x

Table 12: Segment Needed Lists for Requests of Table 11

ters in the source slice. This requires the destination slice and its successor (modulo eight) for a counterclockwise access. A clockwise access to the same slice requires all the RingBus segments, and is never used.

It is clear that short accesses in one direction are long accesses in the other direction. All arbiter designs considered in this chapter always grant the shorter of the two paths. In general, such schemes do not perform optimally, since there are some

cases (e.g. 2→5 and 3→4) when choosing the longer path for an access would actually result in more parallelism on the RingBus. As mentioned in Chapter 3, the investigation of the RingBus for Concert has concentrated exclusively on arbitration schemes which make only one pass through the requests. Specifically, this means that the arbiter considers only one path for each request.

As described in Chapter 3, two requests can be granted in parallel if they do not require any common RingBus segments. This may be determined by ANDing the SNLs for the two requests. It is easy to see how to do this in hardware, or how to incorporate it as part of a larger function in a ROM.

6.3: Granting the Requests

Given the SNLs for a round of requests, the requests currently in progress, and some priority ordering, the arbiter must determine which (if any) new requests to grant in a given cycle. The basic approach is to try to grant requests in priority order, where the order is established by one of the priority schemes outlined in Chapter 3.

There are many possible implementations of the arbiter function, some of which are rather impractical. Since the arbitration algorithms discussed in Chapter 3 all iterate over the slices in priority order, the most obvious hardware implementation of these algorithms is to somehow perform just such an iteration. The solution which immediately springs to mind is a microprocessor; it would easily handle the task of examining requests, executing some decision algorithm, and sending out grants. However, the speed would be orders of magnitude slower than what is needed. Since the arbiter is controlling *single* memory cycles of the nodes, its total time to make a decision must

be of the same order — a few hundred nanoseconds.

Since the arbiter may be modelled as a finite-state machine, another obvious implementation is a large ROM to replace the block labelled "Arbitration Logic" in Figure 21. Unfortunately, such a ROM is far beyond the capabilities of current technology. It would have to map $40+p$ inputs (4 request lines and one feedback line for each of eight sources, plus p priority bits) to $24+p$ outputs (3 enable lines for each of eight sources plus p priority bits). Clearly, a more clever approach is required.

Two fundamentally different classes of feasible arbiter implementations were examined in the course of the design. The first implementation is shown in Figure 23. The requests are sorted based upon whatever priority scheme is being used. The sorted requests are then sent to a *chain* of ROMs (or other logic) which decide whether or not to grant the requests. They do this by examining a "Segments in Use" (SIU) list, which is nothing more than the accumulated SNLs for all requests granted thus far. Each ROM compares its input SIU with the SNL for its request, and grants the request if there are no conflicting segments. Since the ROMs are connected in priority order, a higher priority request always gets a chance to be granted before a lower priority request.

The chain method may be used to implement any of the three arbitration algorithms discussed in Chapter 3. The difference lies in how each ROM updates the SIU. If it ORs in the SNL only when it grants its request, the net result is the arbitration scheme given as Algorithm 3. If it is unable to grant the request, it could set all the SIU bits and effectively prevent any more requests from being granted that cycle. This corresponds with the limited arbitration algorithm of Algorithm 4. Finally, it may OR in

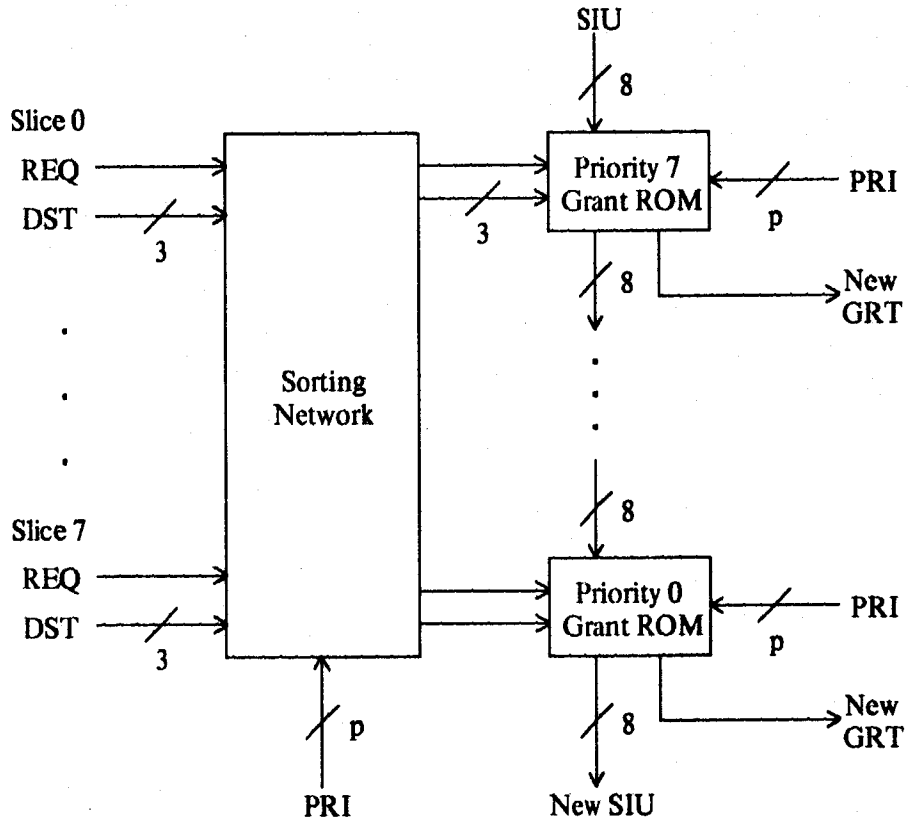


Figure 23: Chain Arbiter Implementation

the SNL for its request regardless of whether or not it grants the request, which implements full arbitration as shown in Algorithm 5.

Despite its flexibility, there are a number of problems with the chain arbitration method. The SIU must be initialized at the beginning of each cycle with the segments used by the requests in progress. Another problem arises in trying to define the arbiter's interface to the RIBs. Requests are actually carried out by sending appropriate enable signals. The implementation of Figure 23 yields only grant lines based on

priority; these must be "unsorted" to derive the enable lines for the slices.

In addition to these shortcomings, the chain arbiter scheme is both slow and costly in hardware. Eight ROM delays plus random logic is a rather heavy penalty in terms of latency. Although the actual decision hardware is not very extensive, the priority sorter and the logic necessary to generate the enable signals is quite considerable.

Several variations on the chain arbitration method were also investigated. A good deal of hardware is saved by replacing the sorting network with a queue of requests which can be read in parallel. Once the queue is set up, its elements are simply fed to the appropriate ROMs in the chain. Chapter 3 discussed at some length the problems associated with such a queue, and the concerns still apply.

It is also possible to reduce the hardware cost considerably by folding the chain arbiter into an *iterative* implementation. Only one grant ROM is required, and an arbiter cycle requires eight passes with the SIU being saved as state each time. The priority sorter can also be eliminated; a queue which presents the next highest priority request on each pass suffices. Unfortunately, the time to perform an arbitration cycle depends on not only the logic delays but also on the settling time of the latches..

One possibility for increasing speed is to reduce the number of levels or iterations, and only grant the requests at high priority levels each cycle. However, the loss of parallelism on the RingBus is quite substantial. At a minimum, new requests should not have to climb several priority levels before even being considered for granting.

There are also several compromises between the chain and iterative implementations. For example, the arbiter might make four iterations through a chain of only two ROMs. Unfortunately, the dependencies enforced by the priority order make it impossi-

ble to perform the arbitration of the requests in parallel. This makes the speed essentially constant, and unacceptable, for any of the variations of the chain method.

A considerably different approach to the problem yields a scheme which is much faster than any of the chain implementations. This implementation, termed *criss-cross*, is shown in Figure 24. It adopts an approach orthogonal to that of the chain implementation. Instead of arbitrating all the segments for each slice in parallel, it arbitrates all the slices for each segment in parallel. In other words, each segment of the RingBus is provisionally granted to a single slice. This decision is made on the basis of the priority ordering. A request is granted if and only if it has been granted all of the segments it needs.

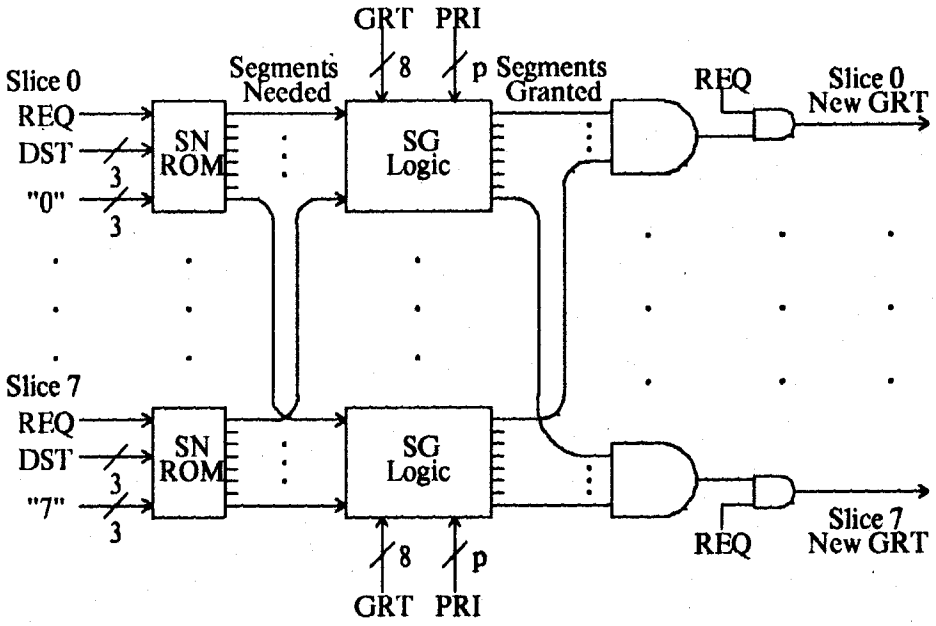


Figure 24: Criss-Cross Arbiter Implementation

The criss-cross arbiter implementation works in a reasonably straightforward manner. A *SN* ROM is used to generate the SNL for each of the slices, with no sorting by priority order. The values coming out of the ROMs are the segments needed by each slice. These are *crossed* and collected to form the slices needing each segment. The *segment grant* logic takes these lines as inputs, and outputs a line for each slice indicating if the slice's request has been granted for the segment.

In order to make this decision, the SG logic needs to know two additional pieces of information. It needs to know which requests have previously been granted, so that the accesses may continue without interruption. This information is represented by the values of the grant signals generated in the previous arbiter cycle. If a request which uses the segment in question is in progress, the SG logic simply continues to grant the segment to that slice. Since only one slice can be using a segment at a time, there is at most one such slice.

If the segment is not currently tied up by an ongoing access, then the logic grants it to the highest priority slice which requires it by asserting the appropriate output line. The priority order is encoded by lines which feed into the SG logic. If a general priority scheme is used, in which each of the eight slices may have any of eight priority levels, there are 8^8 possible combinations. Twenty-four lines are required to encode these priorities.

Even if only one slice is allowed at each priority level, there are still 8! orderings. In this case, sixteen lines are required. However, if rotating priority is used, then there are only eight possible priority orderings and these may be encoded in three lines.

If a request does not need a particular segment, the SG logic asserts the output

line to signify that the request is grantable in terms of that segment. The outputs of the blocks of SG logic are the slices granted for each segment. These lines are "crossed" and collected to form the segments granted for each slice, known as the *Segments Granted List* (SGL). If all the segment grant lines for a particular slice are asserted, i.e. the AND is true, then that slice's request may be granted.

There is one minor complication with this scheme. A slice that is not making a request outputs all 0s from its SN ROM, since it requires no segments. Since the SG logic grants segments that are not needed, the request is granted all slices, and its $\overline{ENM}$ would be erroneously asserted. To avoid this, the grant line is ANDed with the slice request line to insure that a request is really active.

The attractions of this scheme are obvious; the basic hardware is purely combinational with fewer ROM delays than the chain scheme. Depending upon how many bits are used, it can handle any of the priority schemes discussed in Chapter 3. It does not handle as wide a range of arbitration algorithms as the first two implementations. Since the SG logic locally grants any requests which do not need the segment or the highest priority request which does need the segment, only the full arbitration algorithm from Chapter 3 may be implemented. Fortunately, this is the preferred algorithm since it tries for maximum parallelism without sacrificing fairness.

The criss-cross method uses a fair amount of hardware, but some steps can be taken to reduce it. The bulk of the hardware lies in the SG logic, and the amount required is largely dependent upon the number of priority and grant lines. Section 6.5 outlines some ways to reduce the number of these lines.

6.4: Generating the Enable Signals

All arbitration schemes discussed in the previous section generate grant signals which must be translated into enable signals for the RIBs. One advantage of the criss-cross arbiter implementation is that it makes it easy to generate the enable signals. Since the grant lines are ordered by slice number rather than by priority it is easy to recombine them with the destination bits to calculate the proper values of the enable lines. As described in the previous chapter, there are three such signals for each RIB in the ring. The $\overline{ENM}$ signal connects the slice Multibus to the RingBus segment of the next slice. The $\overline{ENR}$ signal allows an access to flow from the slice's RingBus segment to the next segment. The $\overline{ENL}$ allows an access to propagate in the opposite direction.

The process of generating the enable signals is actually rather simple. The arbiter asserts the $\overline{ENM}$ signal for each RIB whose request has been granted. This connects the slice Multibus to the RingBus segment. It then asserts the $\overline{ENL}$ and $\overline{ENR}$ signals necessary to connect together the segments needed for that request. If two adjacent segments are needed, then the buffers connecting them are enabled with the direction determined by the direction of the access. The arbiter asserts the $\overline{ENR}$ signal for a slice to propagate a request from its left RingBus segment to its right RingBus segment. The $\overline{ENL}$ signal allows a request to propagate from either the Multibus or the right RingBus segment to the left RingBus segment.

As shown in Figure 25, the enable signals may be generated in a manner similar to the SNLs. Two more sets of ROMs take as inputs the *DST* lines for each slice, plus

the signal indicating if the request has been granted. The $\overline{ENM}$ signal is just the negation of the grant signal; the ROMs output the $\overline{ENR}$ and $\overline{ENL}$ settings required for each slice in the ring. Since only one access can take place on a RingBus segment at a time, at most one granted request causes a particular enable signal to be asserted. Thus, it is safe to OR together the ROM outputs for each request to produce the final $\overline{ENR}$ and $\overline{ENL}$ values to send to RIBs.

The Concert arbiter follows the convention that the RingBus Left is connected to the next lowest numbered slice and the RingBus Right to the next highest (both modulo eight). Assuming this convention, Table 13 shows the SNLs and the enable signals which are generated for the request 6→4. To make the relationship clearer, the positive-true enable signals are listed.

6.5: The Final Design

This section outlines the final arbiter design chosen for the Concert system. It gives an overview of the implementation as well as some particularly crucial practical details.

6.5.1: The Arbitration Scheme

Figure 26 summarizes the Concert arbiter implementation. It is a variation of the criss-cross technique. After being latched, the *REQ* and *DST* lines from each RIB are sent through ROMS to generate the Segment Needed Lists. The segments required by each slice are crossed and collected to form the slices requiring each segment. These

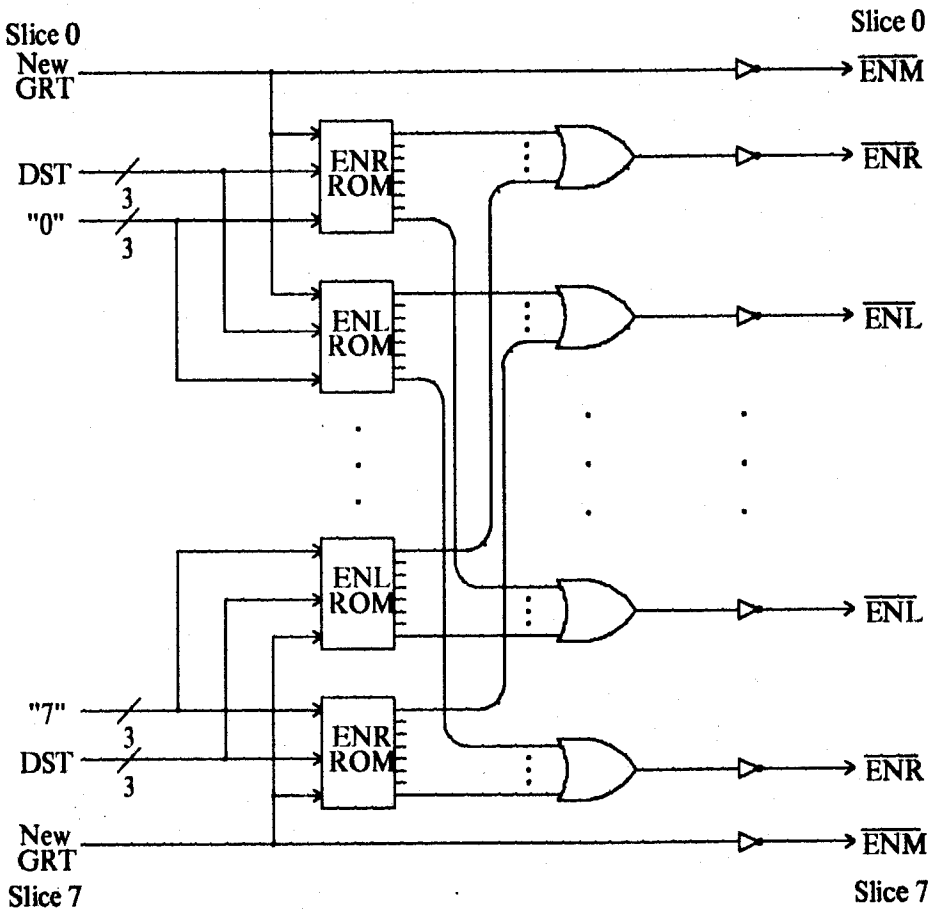


Figure 25: Arbiter Enable Signal Generation

signals are fed into another set of ROMS, which generate the list of slices whose requests may be granted locally, i.e. in terms of that particular segment.

As discussed earlier, the ROMS which generate the Segment Grant Lists also need inputs identifying the current priority order and the requests previously granted. Two tricks are used to reduce the number of lines necessary to carry this information, and

segment	Clockwise Access								Counterclockwise Access							
request	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
SNL	x	x	x	x	x			x					x	x	x	x

Table 13a: Segment Needed Lists for Request 6→4

slice	Clockwise Access								Counterclockwise Access							
request	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
ENM	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
ENL	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0
ENR	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0

Table 13b: Enable Signals for Request 6→4

hence the size of the SG ROMs. The arbiter incorporates a simple rotating priority scheme, which requires a minimum of hardware. A three-bit register identifies the current top priority slice, and its output bits are fed into each SG ROM. From these, the ROM determines the top priority request and orders the other requests by counting down modulo eight.

Each arbiter cycle, the priority is rotated to the next slice which has a pending ungranted request. The scheme is very similar to that used on the RIB to perform the Multibus arbitration. As shown in Figure 27, the only difference is that the SG ROMs automatically determine the top priority active request, and thus the priority update ROM does not have to output this information. As on the RIB, the ROM does output the number of the top priority slice for the next cycle, which is loaded into the register. The decision to use rotating priority instead of more complicated history schemes reduces the number of lines required to identify the priority ordering from a possible twenty-four down to three.

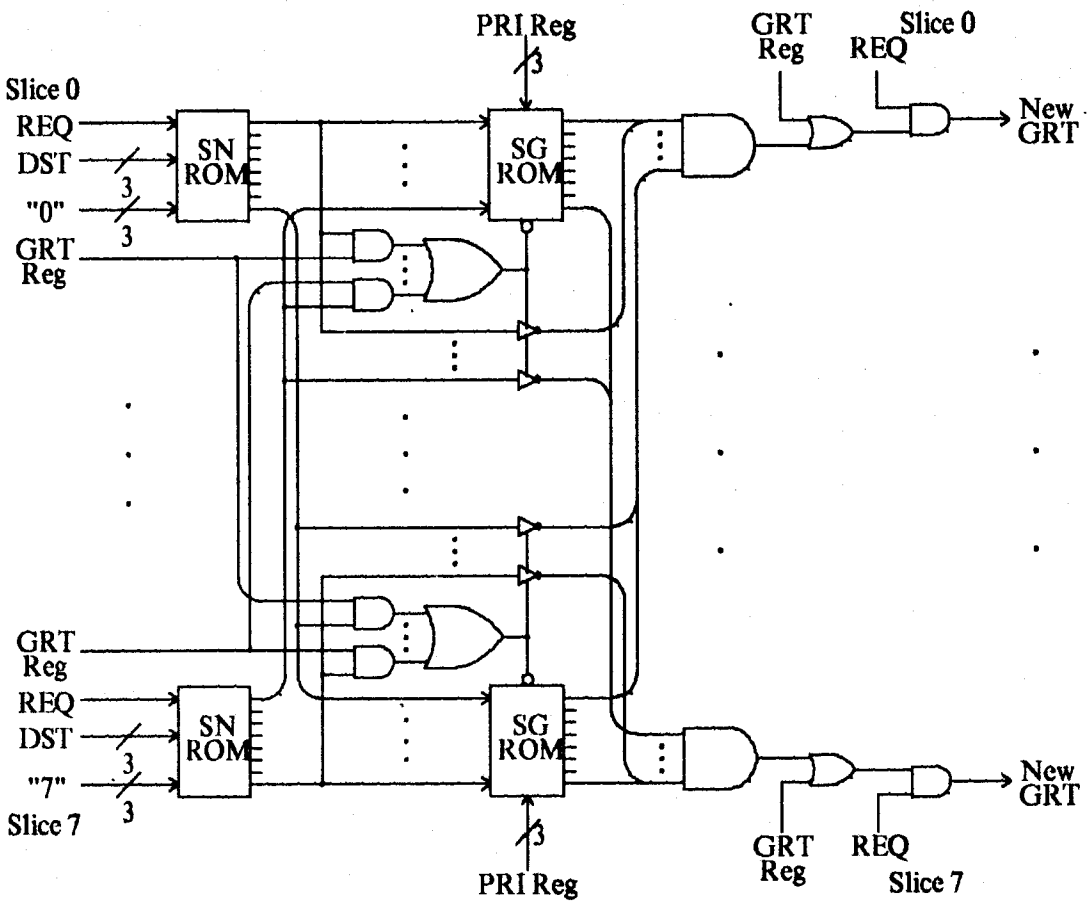


Figure 26: Concert Arbiter Design Scheme

The most obvious way to tell the SG ROMs what requests are currently in progress is to input the current grant lines. Unfortunately, this requires eight lines, and thus increases the ROM size by a factor of 256. Instead, a single line indicating if the segment is currently in use is sent to each SG ROM. If the in-use line is not asserted, the ROM uses the normal criss-cross scheme of locally granting all requests which do not need the segment plus the highest priority request which does require it. If the in-use line is asserted, the ROM outputs are disabled and instead separate tri-state drivers

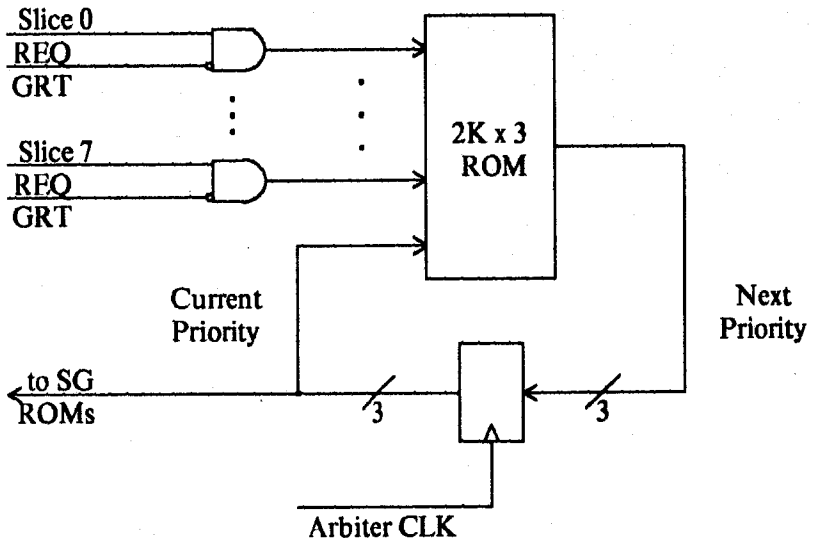


Figure 27: Concert Arbiter Priority Implementation

output the negated segments needed. This results in the local grant of only those requests which don't require the segment.

The in-use line for each segment is generated by an AND-OR tree. A segment is currently in use if any request that is currently granted has the bit for that segment bit asserted in its SNL.

The slices granted each segment are criss-crossed and collected to yield the segments granted for each slice. A new request is globally granted if it has been locally granted for each segment. Since null requests require no segments, they also show up as granted. For this reason, the request line for a slice is ANDed with the Segments Granted List to determine if a new request may be granted. If the request has been previously granted and is currently in progress, the "segment in use" line to the ROMs insures that the request is not granted through the normal path. Instead, all previously

granted requests that are still active are automatically granted on subsequent arbiter cycles.

The generation of the enable signals for the RIBs is done in the exact manner described in the previous section and shown in Figure 25. The $\overline{ENM}$ signal is driven directly from the inverted grant line, and the $\overline{ENR}$ and $\overline{ENL}$ signals are generated by ROMs.

6.5.2: Flexibility

All the arbiter schemes discussed, including the final design, have made extensive use of ROMs both to reduce package count and to provide a measure of flexibility. In fact, the different ROM technologies available allow a wide variety of options. For example, using EPROMs allows the arbitration scheme to be modified easily for experimentation. The Concert prototype uses PROMs for the generation of the Segment Needed Lists, Segment Grant Lists and enable signals. Although they do not provide the same degree of flexibility as EPROMs, they are much faster and therefore allow a much shorter arbiter cycle.

Although the PROMs themselves cannot be altered, different PROMs may still be inserted if desired. Thus, it is possible to make slight modifications in the arbitration scheme by programming new PROMs. More importantly, the PROMs allow the same arbiter board to work equally well over a wide variety of variations in the RIB access paths. The SN PROMs can be altered to reflect different segment requirements for the requests, and the ER and EL PROMs can be changed to reflect a different interpreta-

tion of the lines to the RIBs.

Such changes might be desired even if the RIBs themselves are not altered. For example, it would be interesting to run the same application on both a unidirectional and bidirectional RingBus and compare the performance differences. All that is required for such an experiment is to replace the SN and enable PROMs for each slice. If the SNLs accurately reflect the segments required for unidirectional accesses and the enable lines are only asserted in one direction, the result is a undirectional RingBus produced by underutilizing the RIB access paths.

Some aspects of the arbiter cannot be easily changed. Only three lines of priority information run to the SG PROMs, which means that only eight different orderings are possible. Although the SG PROMs can interpret these lines differently, it's hard to imagine any useful orderings other than rotating priority. Since a PROM is used to update the priority register, some flexibility is available in terms of the rotation scheme used. Any other priority schemes, including history methods, would require extensive redesign of the arbiter board. Redesign would also be required to extend the number of slices in a ring or the number of control lines running from the arbiter to the RIBs.

6.5.3: Practical Issues

There are a number of practical timing issues which had to be faced in the course of the arbiter design. Both the input and output lines of the arbiter are latched to insure glitch-free operation. A round of requests begins when the current values of $\overline{REQ}$ and *DST* from the slices are latched in. At the end of the cycle, the values of $\overline{ENM}$,

$\overline{ENR}$, and $\overline{ENL}$ determined by the arbitration are latched and sent back to the slices.

This scheme, by itself, is not quite sufficient for reliable performance. Since the requests from the RIBs happen asynchronously with respect to the arbiter cycles, it is possible for the input registers to experience metastable states. The standard solution — two levels of registers — was employed to solve this problem. Shortly before the beginning of an arbitration cycle, a sample clock pulse latches the input lines. These values are then latched into the second set of registers by the arbiter clock. The time between these two pulses is sufficient to allow any metastable states in the first set of latches to settle. Using a second set of registers was judged preferable to the alternative of extending the arbiter cycle to allow for the settling.

The lack of coordination between the slices and the arbiter caused another sticky problem. The arbiter samples the request lines from the slices at the beginning of each cycle. In order to know that a request is complete, it must read an unasserted request line for at least one cycle. However, the length of the arbiter cycle when compared with the memory access time of the 68000 makes it conceivable that a node might finish one memory access and begin another within a single arbiter cycle. The consequences of the arbiter missing a dropped request are considerable — timeout on the RingBus with no way to detect the problem and abort the cycle. The solution chosen was to put a latch on each request line which insures that a deasserted request line is held for at least one arbiter cycle.

Another practical issue was mentioned in Chapter 4 during the definition of the RingBus. All RingBus transactions begin on the leading edge of $\overline{GO}$, which signifies that all address, data and control lines are stable and ready for the operation. Since

RingBus lines may propagate through several sets of buffers between the source and destination slice, care must be taken to insure that the $\overline{GO}$ pulse doesn't begin a transaction before the other signals are stable. Likewise, the leading edge of $\overline{ACK}$ indicates that the transaction is complete and, in the event of a read, that data is available. This line also must not arrive before the data lines are stable.

The solution chosen, as described in Chapter 4, is to make the RingBus a "semi-synchronous" bus. All address, data and most control lines propagate freely along RingBus segments from the source to the destination. However, both $\overline{GO}$ and $\overline{ACK}$ do not pass to their intended destination until a pulse from the arbiter latches them. This pulse, *LCLK*, is sent out by the arbiter and passed to all the slices as a signal on the RingBus. *LCLK* is generated after the end of an arbiter cycle, once sufficient time has elapsed for all RingBus signals to propagate and settle at their respective destinations.

Chapter 7: Conclusions

This final chapter summarizes the results of the thesis project. It attempts to evaluate the current state of the Concert project, and makes some suggestions for continuing research. It also discusses the possibilities for future research into the RingBus architecture.

7.1: Summary

It is difficult to evaluate the usefulness of Concert since, at the time of this writing, the prototype system is still being constructed and debugged. However, the hardware design effort has revealed a number of encouraging facts. The basic architecture has turned out to be fairly easy to implement, except for the all-too-common problems of poor documentation and unexpected delays in various places.

The initial estimates for the amount of hardware necessary to implement the Concert system turned out to be a little low. Both the arbiter and the RIB use more chips than originally envisioned, but they each still fit on a single Multibus card. The increase in hardware was due to part to an underestimation of the difficulty of certain parts of the design, and in part to the scourge of "creeping featurism." Regardless, most of the design goals have been met thus far.

The promise of Concert as a research vehicle in M.I.T. is also difficult to ascertain. Several people, notably Bert Halstead and Tom Sterling, are committed to the project and are actively planning applications work on the system. At a minimum, it would be

interesting to get a system of a a dozen or two dozen nodes running and measure the performance.

7.2: Suggestions for Future Research

There are a number of possibilities for future research into the RingBus architecture in general and the Concert implementation in particular. Future redesign of the hardware is probably inevitable. Some aspects of the design may be suitable for implementation in VLSI; this would be an interesting route to pursue. The large number of input and output lines to the RIB make it an unlikely candidate. However, a VLSI implementation of the arbiter is feasible. If a modular arbiter design can be identified, it would be possible to integrate the arbitration logic for a fixed number of slices on a chip, and then use multiple chips for larger rings.

There are numerous opportunities for comparing the RingBus architecture to previous tightly-coupled multiprocessor projects. The real-time measurement capabilities of Concert provide one such means. Further simulation, possibly with new arbitration or priority schemes, is also feasible. The arbiter simulator described in Chapter 3 is fairly simple. The use of data from actual multiprocessor programs and a more sophisticated simulator would provide more believable results.

Very little formal analysis of the architecture has been attempted. A model which would allow analytic comparisons of segmented and non-segmented bus architectures would aid greatly in the understanding of the RingBus architecture. It would also help to assess the value of segmented buses in general; very little previous work has been

done in this area.

Finally, of course, there is a wide range of applications which could be run on Concert. Those listed in Chapter 2 would literally take years to complete, and it is likely there are plenty of others which have not yet been identified. The process of writing applications programs will help to understand more about programming Concert, and multiprocessors in general. For example, some applications will use functional languages like LISP and others will use more traditional approaches like communicating sequential processes [32]. Comparison of performance data from programs using different multiprocessor programming schemes could produce some interesting results.

Whatever the future usefulness of the RingBus architecture or of Concert-like systems, it is clear that there are many avenues to explore in the immediate future. The applications listed in Chapter 2 and the suggestions for future work made in this section provide a number of topics suitable for research and student projects in the next few years.

Bibliography

[1] Alpern, D., "Some Notes on Concert Environment and Communication," Concert working paper, June 1982.

[2] Anderson, G.A. and E.D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics and Examples," *Computing Surveys 7:4*, December 1975, pp. 197-213.

[3] Anderson, T.L., "Design Specification for a Multiprocessor Development System," Concert working paper, June 1982.

[4] Arden, B.W. and R. Ginosar, "MP/C: A Multiprocessor / Multicomputer Architecture," *Proc. Compcon Spring 81*, February 1981, pp. 290-293.

[5] Arvind, K.P. Gostelow, and W.E. Plouffe, "An Asynchronous Programming Language and Computing Machine," TR114A, Dept. of Information and Computer Science, University of California at Irvine, December 1978.

[6] Arvind, V. Kathail and K. Pingali, "A Dataflow Architecture with Tagged Tokens," MIT/LCS/TM-174, Laboratory for Computer Science, M.I.T., September 1980.

[7] Arvind, V. Kathail, and K. Pingali, "A Processing Element for a Large Multiple Processor Dataflow Machine," VLSI Memo No. 80-26, M.I.T., August 1980.

[8] Barnes, G.H., et al., "The Illiac IV Computer," *IEEE Transactions on Computers C-17:8*, August 1968, pp. 746-757.

[9] Barthmaier, J., "Intel Multibus Interfacing," in "iAPX 86, 88 User's Manual," Intel Corporation, August 1981, pp. A175-A208.

[10] Batcher, K.B., "STARAN/RADCAP Hardware Architecture," *Proc. 1973 Sagamore Conference on Parallel Processing*, August 1973, pp. 147-152.

[11] Bouknight, W.J., et al., "The Illiac IV System," *Proc. IEEE 60:4*, April 1972, pp. 369-388.

[12] Bowen, B.A. and R.J.A. Buhn, *The Logical Design of Multiple-Microprocessor Systems*, Prentice-Hall, Inc., 1980.

[13] "DBC68K Hardware Reference Manual," Microbar Systems, Inc., May 1982.

[14] "DBC86 Hardware Reference Manual," Microbar Systems, Inc., January 1982.

[15] "DBR50 Hardware Reference Manual," Microbar Systems, Inc., December 1981.

[16] Dennis, Jack, "Data Flow Supercomputers," *Computer 13:11*, November 1980, pp. 48-56.

[17] Enslow, P.H., Jr., "Multiprocessor Organization — A Survey," *Computing Surveys 9:1* , March 1977, pp. 103-129.

[18] Enslow, P.H., Jr., ed., *Multiprocessors and Parallel Processing*, John Wiley and Sons, 1974.

[19] Farrell, E.P., N. Ghani and P.C. Treleaven, "A Concurrent Computer Architecture and a Ring Based Implementation," *Proc. Sixth Annual Symposium on Computer Architecture*, April 1979, pp. 1-7.

[20] Flynn, M. J., "Very High Speed Computing Systems," *Proc. IEEE 54:12*, December 1966, pp. 1901-1909.

[21] Fong, J. and C. Pottle, "Parallel Processing of Power System Analysis Problems via Simple Parallel Microcomputer Structures," *IEEE Transactions on Power Apparatus and Systems PAS-97:5*, September/October 1978, pp. 1834-1841.

[22] Foster, C.C., *Content Addressable Parallel Processors*, Van Nostrand Reinhold, 1976.

[23] Fuller, et al., "Multi-Microprocessors: An Overview and Working Example," *Proc. IEEE 66:2*, February 1978, pp.216-228.

[24] Gostelow, K.P. and R.E. Thomas, "Performance of a Simulated Data Flow Computer," *IEEE Transactions on Computers C-29:10*, October 1980, pp. 905-919.

[25] Halstead, R., "Architecture of a Myriaprocessor," *Proc. COMPCON Spring 81*, February 1981, pp. 299-302.

[26] Halstead, R., "Architecture of a Myriaprocessor," in *Advanced Computer Concepts*, J. Solinsky, ed., La Jolla Institute, 1981.

[27] Halstead, R., "Reference Tree Networks: Virtual Machine and Implementation," MIT/LCS/TR-222, Laboratory for Computer Science, M.I.T., July 1979.

[28] Halstead, R.H. and S.A. Ward, "The MuNet: A Scalable Decentralized Architecture for Parallel Computation," *Proc. Seventh Annual Symposium on Computer Architecture*, May 1980, pp. 139-145.

[29] Hansen, P.B., "Multiprocessor Architectures for Concurrent Programs," *Proc. ACM '78 Vol. 1*, December 1978, pp. 317-323.

[30] Harris, J.A. and D.R. Smith, "Hierarchical Multiprocessor Organizations," *Proc. Fourth Annual Symposium on Computer Architecture*, March 1977, pp. 41-48.

[31] Haynes, L.S., et al., "A Survey of Highly Parallel Computing," *Computer 15:1*, January 1982, pp. 9-24.

[32] Hoare, C.A.R., "Communicating Sequential Processes," *CACM 21:8*, August 1978, pp. 666-677.

[33] "iSBC 86/12A Single Board Computer Hardware Reference Manual," Intel Corporation, 1979.

[34] Kassakian, J.G., "Simulating Power Electronic Systems — A New Approach," *Proc. IEEE 67:10*, October 1979, pp. 1428-1439.

[35] Kernighan, B.W. and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[36] Kuck, D.J., "A Survey of Parallel Machine Organization and Programming," *Computing Surveys 9:1*, March 1977, pp. 29-59.

[37] "MC68000 16-Bit Microprocessor User's Manual," Second Ed., Motorola, Inc., January 1980.

[38] *Multiprocessor Systems*, Infotech, 1976.

[39] Patel, J.H., "Processor-Memory Interconnections for Multiprocessors," *Proc. Sixth Annual Symposium on Computer Architecture*, April 1979, 168-177.

[40] "Proposed Microcomputer System 796 Bus Standard," *Computer 13:10*, October 1980, pp. 89-105.

[41] Ramamoorthy, C.V. and H.F. Li, "Pipeline Architecture," *Computing Surveys 9:1*, March 1977, pp. 61-102.

[42] Russell, R.M., "The Cray-1 Computer System," *CACM 21:1*, Jan. 1978, pp. 63-72.

[43] Satyanarayanan, M., "Multiprocessing: An Annotated Bibliography," *Computer 13:5*, May 1980, pp. 101-116.

[44] Satyanarayanan, M., *Multiprocessors: A Comparative Study*, Prentice-Hall, Inc., 1980.

[45] Siegel, H.J., "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," *IEEE Transactions on Computers C-28:12*, December 1979, pp. 907-917.

[46] Siewiorek, D.P., "Modularity and Multi-Microprocessor Structures," *Proc. Seventh Annual Workshop on Microprogramming*, October 1974, pp. 186-193.

[47] Smith, A.J., "Multiprocessor Memory Organization and Memory Interference," *CACM 20:10*, October 1977, pp. 754-761.

[48] Sterling, T.L., J.G. Kassakian and E.Y. Chan, "A Multiprocessor for Power Electronic Circuit Simulation," *1981 IEEE Power Electronics Specialists Conference*, 1981.

[49] Sterling, T.L., "Parallel Computer Processing for Power Electronic Network Simulation," M.S. Thesis, Electrical Eng. and Computer Sci. Dept., M.I.T., May 1981.

[50] Stone, H.S., "Parallel Computers," in *Introduction to Computer Architecture*, H. S. Stone, ed., Science Research Associates, 1975, pp. 318-374.

[51] Swan, R.J., A. Bechtolsheim, K. Lai, and J.K. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor," *Proc. AFIPS 1977 National Computer Conference Volume 46*, 1977, pp. 645-655.

[52] Swan, R.J., S.H. Fuller and D.P. Siewiorek, "Cm*: A Modular Multi-Microprocessor," *Proc. AFIPS 1977 National Computer Conference Volume 46*, 1977, pp. 637-644.

[53] Thurber, K.J. and L.D. Wald, "Associative and Parallel Processors," *Computing Surveys 7:4*, December 1975, pp. 215-255.

[54] Treleaven, P.C. and R.P. Hopkins, "Decentralized Computation," *Proc. Eighth Symposium on Computer Architecture*, May 1981, pp. 279-290.

[55] "Unix Programmer's Manual: Virtual VAX-11 Version," Seventh Ed., University of California at Berkeley, November 1980.

[56] Weitzman, C., *Distributed Micro/Minicomputer Systems*, Prentice-Hall, Inc., 1980.

[57] Widdoes, L.C., "The S-1 Project: Developing High-Performance Digital Computers," *Proc. Spring COMPCON 1980*, February 1980, pp. 282-291.

[58] Wulf, W. and C.G. Bell, "C.mmp - A Multi-mini-processor," *Proc. AFIPS 1972 Fall Joint Computer Conference, Volume 41*, December 1972, pp. 765-777.

[59] Yau, S.S. and H.S. Fung, "Associative Processor Architecture — A Survey," *Computing Surveys 9:1*, March 1977, pp. 3-27.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-279	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Design of a Multiprocessor Development System	5. TYPE OF REPORT & PERIOD COVERED: Technical Report Sep. '82
6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-279	
7. AUTHOR(s): Thomas Lee Anderson	8. CONTRACT OR GRANT NUMBER(s): DARPA N00014-75-C-0661
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, Ma. 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA, 1400 Wilson Blvd., Arlington, Va. 22217	12. REPORT DATE: September 1982
	13. NUMBER OF PAGES: 122
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research, Department of the Navy, Information Systems Program, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public sale and release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

see back

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see back

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The Design of a Multiprocessor Development System

by

Thomas Lee Anderson

Submitted to the Department of Electrical Engineering and Computer Science
on September 1, 1982 in partial fulfillment of the
requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

A multiprocessor development system has been designed and a prototype system is being constructed. The system, known as Concert, is intended to support multiprocessor research efforts at M.I.T. The motivation for Concert and the project history are summarized briefly. Some intended applications are also identified.

The system incorporates the RingBus architecture, a novel scheme for interconnecting processors and memory in a tightly-coupled multiprocessor system. The architecture is described both in its general form and in the particular implementation used in the system. The results of some analysis and synthesis of the architecture are summarized.

The design of the Concert multiprocessor development system is described, with particular emphasis on the tradeoffs considered in the design process. The design of two particular hardware modules is discussed in considerable detail. Finally, some suggestions are offered for future use of the system and further investigation into the RingBus architecture.

Name and Title of Thesis Supervisor:

Robert H. Halstead, Jr.,
Assistant Professor of Computer Science and Engineering

Key Words and Phrases:

multiprocessor systems, segmented computer buses, parallel processing, computer architecture