Typhoon-Zero Implementation: The Vortex Module

Robert W. Pfile

Technical Report #1290

October 1995

Typhoon-Zero Implementation: The Vortex Module

Robert W. Pfile
Wisconsin Wind Tunnel Project
Computer Sciences Department
University of Wisconsin – Madison

pfile@cs.wisc.edu, wwt@cs.wisc.edu

October 13, 1995

Abstract

This report describes *Vortex*, an Mbus module designed to provide hardware support for the Tempest parallel programming interface on clusters of Sun Sparcstation 10 and 20 workstations. The module consists primarily of two Field Programmable Gate Arrays (FPGAs), and two static RAM chips, both of commodity origin. Vortex is the only custom hardware needed for a distributed shared memory system, called *Typhoon-Zero*, which is currently in production and is intended to be a prototype for the proposed *Typhoon* DSM system. This report first presents background information on the Tempest interface and our decision to prototype Typhoon, then gives a description of the SPARC Mbus. Next the theory of operation for Vortex is given, followed by a comprehensive description of the Vortex architecture. Afterward, FPGA selection and design issues are discussed, as well as our design methodology for producing the Vortex module. Next, the logic and the printed circuit board implementation, verification and testing processes are described. Finally, CAD tool difficulties, timing problems and bugs encountered during the design process are described along with the solutions developed to address them. Verilog source code, printed circuit board schematics and other code are presented in the Appendices.

Contents

List of Figures

List of Tables

1 Background

1.1 Tempest

The Tempest parallel programming interface [RLW94] provides an abstraction upon which shared memory and message passing codes can be built on a variety of parallel computers. Rather than specifying a fixed hardware or software global data coherence mechanism, Tempest instead introduces the concept of *user-level protocols*, which allow the *user* to write custom coherence protocols on a per-application basis. By crafting the protocol to match the data sharing behavior of a parallel program, significant performance gains can be realized. Furthermore, because Tempest is an interface, it has the advantage of portability: Tempest codes can be run on a variety of different hardware platforms spanning the cost-performance spectrum.

The Tempest interface identifies and provides four basic mechanisms necessary for creation of user-level protocols. These are: fine-grain access control, virtual memory management, efficient (low overhead) messaging, and bulk node-to-node data transfers. These mechanisms can be implemented in a variety of ways, ranging from all-software systems (such as Blizzard-S on the Wisconsin COW [SFL+94]), to partially hardware assisted implementations (such as Blizzard-E on the CM-5), to high performance all-hardware implementations such as *Typhoon* [RLW94].

1.2 Fine-Grain Access Control

The key mechanism behind Tempest is fine-grain access control, which allows arbitrary user-level protocol code to be associated with small (on the order of cache-block sized) blocks of memory.

Tempest specifies fine-grain access control by associating tags with aligned, power-of-two-sized blocks of memory. A Tempest block can be tagged with one of the following four states: **ReadWrite**, **ReadOnly**, **Invalid** or **Busy**. Both reads and writes are permitted to ReadWrite blocks. A write to a ReadOnly block or any access to Invalid or Busy blocks is considered a block access fault, which suspends the user's memory access and causes a protocol handler to run to rectify the fault. On handler completion, the faulting access is retried. The distinction between Invalid and Busy is semantic; for instance, a block tagged Busy may be considered by a protocol handler to be in the process of being prefetched into.

1.3 Typhoon

Typhoon is a Tempest implementation that relies on high-performance custom hardware to provide the four Tempest mechanisms on a network of Mbus-based workstations, such as that in Figure 1. The custom hardware, a single ASIC which resides on an Mbus module, is known as the Typhoon NP (Network interface Processor) and is depicted in Figure 2. The Typhoon NP combines a commodity processor core (such as an implementation of the SPARC V8 instruction set), a network interface, and a reverse translation lookaside buffer (RTLB.)

User-level protocol code is executed on the NP processor, which is tightly coupled with the network interface via the NP processor's cache bus. Low overhead message and handler dispatching are provided by the Dispatch Control unit, while the Block Transfer unit and BXB (block transfer buffer) provide fast bulk data transfers together with the NI. The RTLB implements fine-grain access control; it snoops on Mbus Coherent transactions, performs physical to virtual address translations and conditionally stalls the compute thread to invoke a fault handler on the NP processor. The Network Interface is also mapped directly onto the Mbus by the NP Mbus Interface. Typhoon's performance is largely due to the tight integration between fine-grain access control, protocol processing, and the network interface.

Simulations show that user-level global cache coherence protocols running on Typhoon perform comparably (within 30%) to that of an all-hardware scheme [RLW94].

1.4 Typhoon-Zero and Vortex

In order to prove the feasibility of our ideas, we would like to implement a Typhoon system. While the custom hardware for a Typhoon node derives entirely from commodity components, we recognize that a single chip implementation is too complex to quickly implement in an academic setting. With

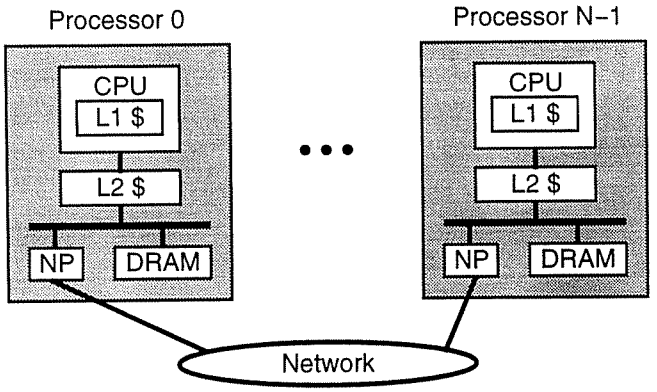


Figure 1: Typhoon System Architecture

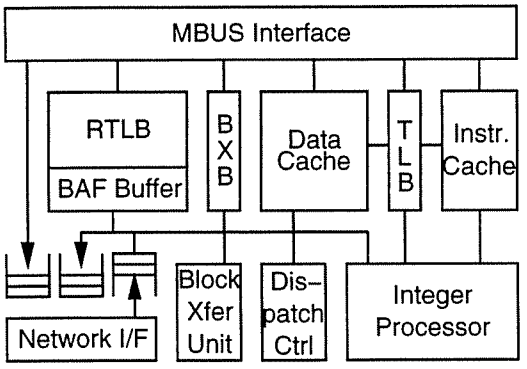


Figure 2: Typhoon NP Block Diagram

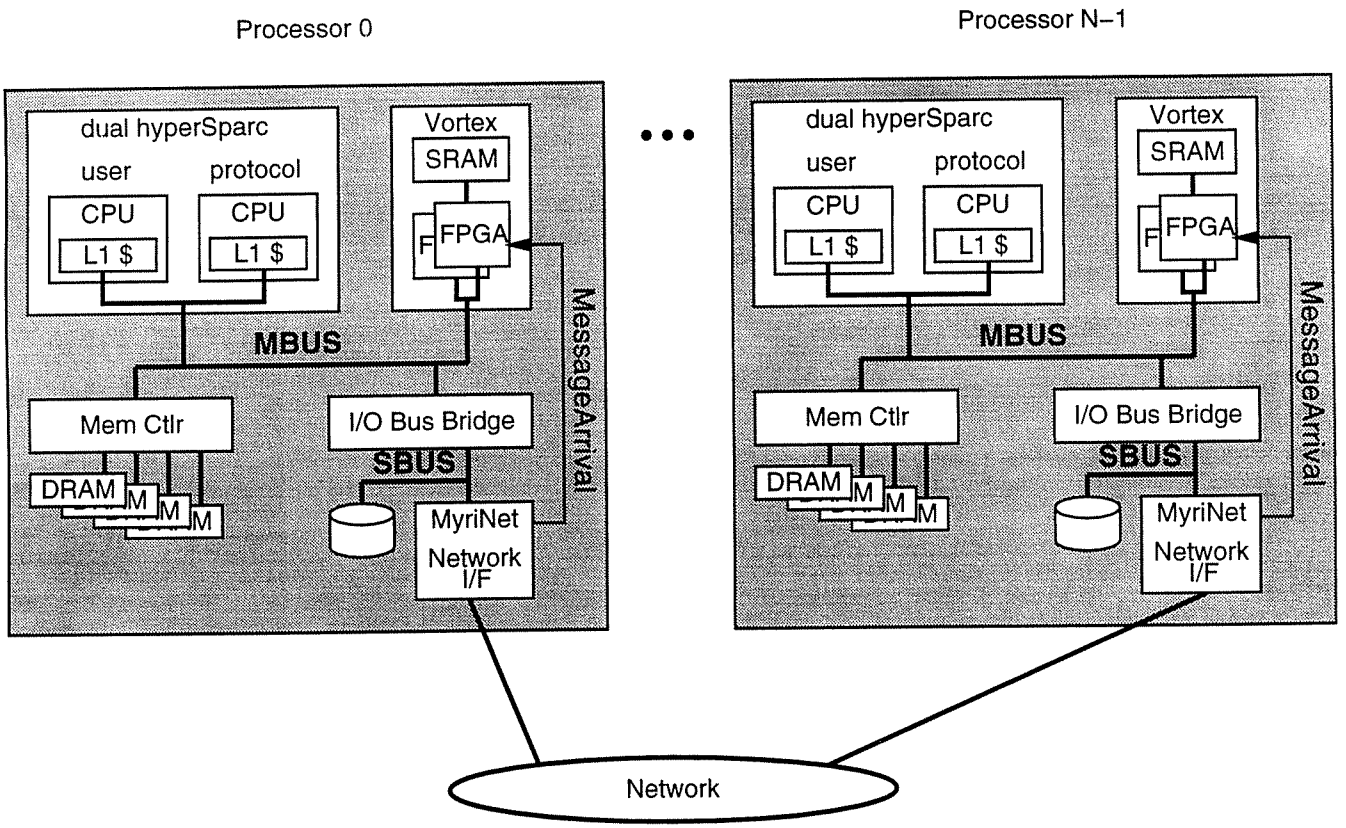


Figure 3: Typhoon-Zero System Architecture

this in mind, we decided to prototype the Typhoon system by building a system called *Typhoon-Zero*. Once again, we would design a single Mbus module (Vortex[1]), but we would trade hardware design complexity for time (and some software complexity) by simplifying the module considerably. By using an off-the-shelf dual ROSS hyperSparc module, we eliminate the need for a module-resident protocol processor – Typhoon-Zero's compute and protocol processors are identical processors residing on the same module. Likewise, by using a CM-5 Network Interface and CM-5 switch, we address both network-related Tempest mechanisms using commodity parts. Reverse translations of bus physical addresses to user virtual addresses are relegated to software, instead of Typhoon's RTLB. Software translation combined with the standard virtual memory support of the user and protocol CPUs provides the necessary Tempest VM support. This leaves fine grain access control, which is the primary responsibility for Vortex. Additionally, Vortex is responsible for "gluing together" the disparate parts of the system (CPUs, network and access control) that were formerly integrated on board the Typhoon NP. This glue logic is important to maintain performance in the face of the physical separation of the components.

In order to have working hardware quickly, we decided to implement all Vortex logic using Field Programmable Gate Arrays (FPGAs.) This choice shifts the design challenges away from ASIC and gate array problems to getting the logic to fit in FPGAs, and to get them to run at Mbus speeds (50MHz).

When Thinking Machines sought Chapter 11 protection in the summer of 1994, we were forced to abandon our plans to include the CM-5 NI aboard the Vortex module, and instead consider SBUS-based solutions, such as ATM or Myricom's Myrinet. This comes with the tremendous disadvantage of much higher network latencies, and possibly reduced bandwidth. In the end, we chose Myrinet. There turned out to be an easy way to efficiently dispatch message handlers with the same mechanism used to dispatch access fault handlers, so the paradigm of Vortex as glue for discrete Typhoon components was left intact.

Figure 4 shows the Vortex Module (Revision A) at full scale. The Mbus connector is at the top of the board and is mounted on the back side. The two FPGAs used to implement all of the logic (Section 9) are below the Mbus connector, and below them are the clock generator PLLs (Section 7.5) and the serial EPROMs used to program the FPGAs when the board is powered up. Below the EPROMs are connectors for the Altera BitBlaster cable, which can be also be used to program the FPGAs. Between the BitBlaster headers are clock configuration jumpers, and below the jumpers are the Tempest fine-grain access control tag SRAMs (Section 5.1). To the left of the SRAMs are a spare connector which can be used to connect currently unused FPGA pins together, and a test header with important signals exposed. To the right of the SRAMs is the messageArrival input (Section 5.2.5) and termination circuitry. Finally, there are 8 LEDs at the bottom edge of the board which are driven by the mode register, the handlerPC status word (Section 6.4.4) and the messageArrival input pin. The board is double sided; the back side is populated with bypass capacitors and miscellaneous resistors and termination diodes.

2 Timeline

This project was completed in one year by two graduate students, Rob Pfile and Steve Reinhardt. Initial design was completed in late May of 1994 by Steve Reinhardt and David Wood. The summer of 1994 was spent writing a C++ simulation of the COW populated by Vortex boards. In late August 1994, Rob Pfile went to Sun Microsystems' s3.mp group under the auspices of Andreas Nowatzyk to leverage off of the work already being done there on the s3.mp machine. By February 1995 the logic was completely debugged (in simulation) and had been demonstrated to be fast enough to properly function at 50MHz (the Sparcstation-20 Mbus clock speed.) The printed circuit board design was done between February and March 1995. After a two-week delay in getting sample quantities of the FPGAs, the first board was completed and tested on 5 April 1995. Over the next month, two bugs that were not exposed in simulation were exposed via random testing and subsequently fixed. There was one board-level bug. By 9 May 1995 the S/N 001 board passed random test, and on 11 May a second Vortex board passed random test. On 3 June 1995, Steve Reinhardt completed the port of Tempest to the Typhoon-Zero system.

As of this writing, we have four populated Vortex revision A boards assembled and tested, and a four-node Typhoon-Zero system is in production. We currently have enough FPGAs to build three more

[1] So named because the "polar vortex" is one of the two forces that contribute to the formation of typhoons, the other being the Coriolis force.

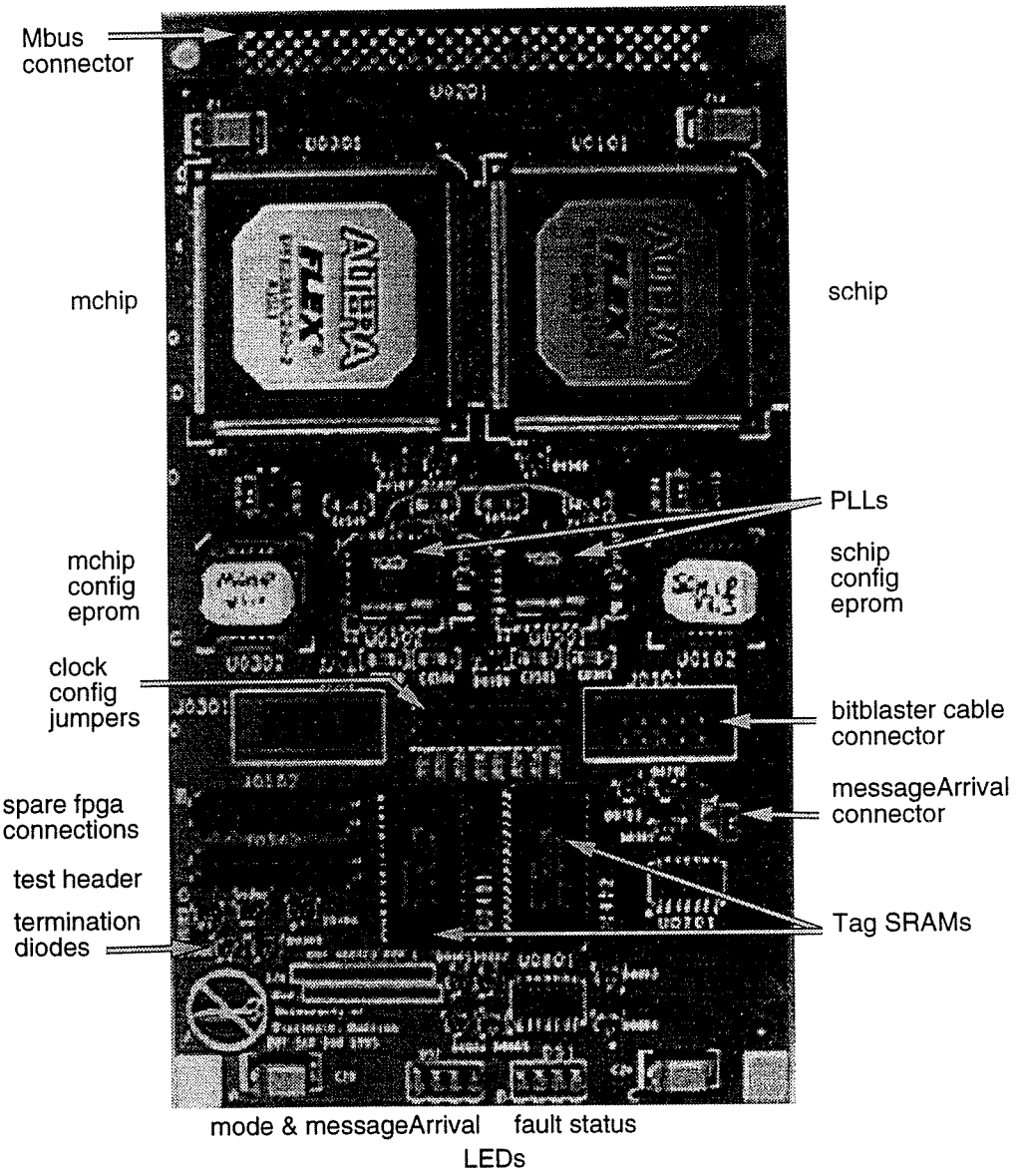


Figure 4: Vortex Module with Annotations

Vortex boards, and are expecting enough FPGAs to build 36 more boards to populate our 40-node Cluster of Workstations.

We have a revised board design ready to be fabricated, and should have them ready when the final shipment of FPGAs arrives in September.

The shortage of the Altera FPGAs we use to implement the logic has caused at least four months of delay in finishing the project.

3 Organization

The organization of the rest of the report is as follows: first an overview of the SPARC Mbus is presented, then a functional description of Vortex is given, consisting of a complete architectural specification and description of services provided by Vortex and the driver software. Next, FPGA selection and design issues are discussed. Afterward, our design methodology, a detailed description of the logic and board implementation and the implementation process are then presented. Finally, we describe our verification process, major problems in the design flow, and the future for this project. Source code and schematics are presented in the appendices.

4 The SPARC Mbus

The SPARC Mbus [Kel91] is a synchronous processor-memory bus which supports shared-memory multiprocessing. It consists of a 64-bit multiplexed address/data bus and 22 control signals, all driven at TTL levels. Control signals are asserted low, but addresses and data are asserted high. Mbus is a single-transaction, big-endian bus, with a 64GB physical address space, which can support up to 16 citizens.

There are two classes of Mbus citizens: *masters* and *slaves*. Masters drive addresses and a transaction type onto the bus and wait for one or more acknowledgments accompanied by data from a slave. Masters are "active" devices, like processors and DMA engines, while slaves are "passive" devices such as memory controllers.

A typical Mbus system is the Sparcstation-20, which has two 50MHz user Mbus slots, an arbiter, a single memory controller supporting up to 512MB of DRAM, and an Mbus to SBUS bridge known as the MSBI. Every Mbus citizen has an identifying number known as an MID, and Sparcstation-20 user Mbus slots are geographically addressed; the motherboard provides the three most significant bits of the MID on the connector. Each citizen on the module must choose it's least significant MID bit, which means that each Mbus slot can support up to two citizens, each of which may contain a master, slave, or both.

4.1 MAD multiplexing

The Mbus address/data bus, known as the MAD bus, is used by masters to signal transaction types and addresses to slave devices as well as for write and read data. When a master becomes the owner of the bus, (after requesting and being granted ownership by the arbiter) it asserts the MAS_ signal and drives the transaction information onto the MAD lines. During the MAS_ phase, the meanings of the MAD pins are given in Table 1, and explained below.

- PA[35:0]

  The Physical Address for the current transaction.

- TYPE[3:0]

  The transaction types (e.g. Read, Write) are shown in Table 3; transaction types 15-6 are reserved.

- SIZE[2:0]

  The transaction size is encoded as log_2[# bytes to transfer]. The encodings are shown in Table 2. Transactions with SIZE larger than 8 bytes are known as Burst transactions; more than one Valid Data acknowledgment is needed to complete the transaction since the Mbus MAD bus is 8 bytes

Signal Name	Physical Signal	Description
PA[35:0]	MAD[35:0]	Physical address of current transaction
TYPE[3:0]	MAD[39:36]	Transaction type
SIZE[2:0]	MAD[42:40]	Transaction data size
C	MAD[43]	Data cacheable (advisory)
LOCK	MAD[44]	Bus lock indicator (advisory)
MBL	MAD[45]	Boot mode / local bus (advisory) (optional)
VA[19:12]	MAD[53:46]	Virtual address (optional) (level-2)
reserved	MAD[58:54]	for future expansion
SUP	MAD[59]	Supervisory Access indicator (advisory) (optional)
MID[3:0]	MAD[63:60]	Module Identifier of master for this transaction

Table 1: MAD signal definitions during address (MAS_) cycle

wide. If a slave generates a non-Valid Data acknowledgment on any of the Burst acknowledgment cycles, that cycle is considered to be the last for the transaction.

- C

  The Cachable indicator may be used to reflect the state of the processor's MMU cacheable bit for the PA of the transaction. As with other advisory signals, it has no effect on the Mbus itself and is only of use to slave devices.

- LOCK

  Advisory signal indicating that the master producing the transaction wishes to lock access to the slave it is attempting to communicate with. If the master loses ownership of the bus on a locked transaction, the targeted slave can reject accesses from any other master until the original master regains ownership of the bus and issues a transaction with the lock bit deasserted.

- MBL

  Optional advisory signal indicating to slaves that the processor issuing the transaction is in local bus mode (SPARC ASI = 0x1) or is in boot mode.

- VA[19:12]

  This field only applies to Level-2 Coherent transactions. It carries bits 19 through 12 of the virtual address (the low byte of the virtual page address) for the block being accessed. This field is provided to support virtually indexed caches on processor modules. The virtual address bits and width used to generate this field assume that the system page size is at least 4KB, and that the maximum cache size is 1MB.

- SUP

  Supervisor access indicator; this signal indicates that the transaction is a processor Supervisor access. It is advisory and optional.

- MID[3:0]

  Module identifier of the master driving the transaction onto the bus.

4.2 Transaction Status Bits

Every transaction produced by an Mbus master is eventually acknowledged by the addressed slave or by a bus monitor which responds with a Timeout error acknowledgement if no slave has responded after an implementation-dependent delay. The Transaction Status control signals, MRDY_, MRTY_ and MERR_ are used to produce this response, and are summarized in Table 4.

Valid Data Transfer is the most common transaction response; it indicates to the master that the slave is returning data on the MAD bus during read transactions, or that write data has been collected

Size[2]	Size[1]	Size[0]	Transaction Size
H	H	H	128-byte Burst
H	H	L	64-byte Burst
H	L	H	32-byte Burst
H	L	L	16-byte Burst
L	H	H	DoubleWord (8 bytes)
L	H	L	Word (4 bytes)
L	L	H	HalfWord (2 bytes)
L	L	L	Byte

Table 2: SIZE[2:0] Encodings

Type[3]	Type[2]	Type[1]	Type[0]	Data Size	Transaction Type
H	H	H	H	-	reserved
H	H	H	L	-	reserved
H	H	L	H	-	reserved
H	H	L	L	-	reserved
H	L	H	H	-	reserved
H	L	H	L	-	reserved
H	L	L	H	-	reserved
H	L	L	L	-	reserved
L	H	H	H	-	reserved
L	H	H	L	-	reserved
L	H	L	H	Burst32	Coherent Read & Invalidate (CRI)
L	H	L	L	Burst32	Coherent Write & Invalidate (CWI)
L	L	H	H	Burst32	Coherent Read (CR)
L	L	H	L	Burst32	Coherent Invalidate (CI)
L	L	L	H	any	Read (RD)
L	L	L	L	any	Write (WR)

Table 3: TYPE[3:0] Encodings

MERR_	MRDY_	MRTY_	Meaning
H	H	H	idle cycle
H	H	L	Relinquish and Retry
H	L	H	Valid Data Transfer
H	L	L	reserved
L	H	H	Error1: Bus Error
L	H	L	Error2: Timeout
L	L	H	Error3: Uncorrectable
L	L	L	Retry

Table 4: Transaction Status Bits Encodings

from the bus by the slave. The Retry response means that the slave would like the master to restart the transaction without releasing the bus; the Relinquish and Retry response tells the master to retry the transaction after relinquishing ownership and re-arbitrating for the bus. The meaning of the three Error responses is implementation-dependent; in the Sparcstation-20, the memory controller returns Bus Error on unaligned memory accesses, and it returns the Uncorrectable error when uncorrectable ECC has been detected. The Timeout error is generated by the bus timeout monitor as described above.

We overload the meaning of these transaction responses as described in Sections 5.1 in order to fault a processor causing a block access violation and indicate malformed transactions to other Vortex control spaces.

4.3 Wrapped Burst Transfers

As indicated by Table 2, there are four "Burst" transactions, where a single address cycle is used to request more than one Doubleword of data from a slave. Burst transactions increase bus bandwidth by amortizing control overhead over several data words. Mbus supports *wrapping* on Burst Reads, which amounts to requiring the transaction PA to be Doubleword aligned, rather than Burst-aligned. The slave is expected to return the addressed Doubleword first, then to increment the Doubleword address modulo the burst size until the full Burst quantity has been returned. This means that after the slave returns the last Doubleword in the Burst, the address is "wrapped around" to the beginning of the Burst and the rest of the Doublewords are returned, up to the one before the originally addressed Doubleword.

Wrapping is useful for processor caches, since they can request and receive the critical Doubleword within a cache block first. This can cut cache miss latency by three Mbus cycles in the worst case, and many more for slow, non-interleaved memories.

4.4 Level-2 Mbus Commands

A key feature of Mbus is the Level-2 command set. In addition to supporting memory read and write transactions, Level-2 systems support shared-memory multiprocessing by providing mechanisms and policies for cache coherence. Cache blocks are 32 bytes in size and are tagged with the following states: **Exclusive Clean**, **Shared Clean**, **Invalid**, **Exclusive Modified** and **Shared Modified** in accordance with the MOESI cache coherence protocol [PP84]. When a processor cache is allowed to share memory with other processor caches, it manipulates memory using one of 4 *Coherent* transactions: Coherent Read (afterward referred to herein as CR), Coherent Invalidate (CI), Coherent Read and Invalidate (CRI) and Coherent Write and Invalidate (CWI). All Coherent transactions except CWI have SIZE = Burst32, and require multiple Valid Data acknowledgments. The exception is the CI transaction which requires only one Valid Data acknowledgment.

Two Mbus control signals are devoted to Level-2 transactions: MSH_ (shared signal), and MIH_ (memory inhibit signal). MSH_ is asserted by caches with a shared copy of a block to indicate that to a cache performing a CR. Since the cache write policy is write-back (with write-allocate), ownership of cache blocks shifts from the memory controller to a processor cache when the processor writes into the cache. When another cache requests a cache-owned block with a CR or CRI transaction, the owning cache uses MIH_ signal to prevent the memory controller from responding and supplies the data itself. All

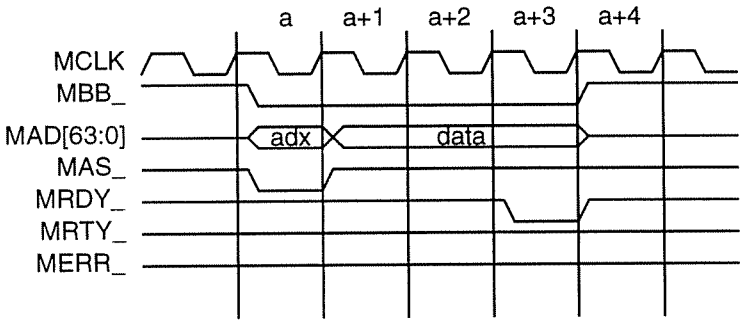


Figure 5: Mbus Timing Diagram for an Uncached Write (WR) Transaction

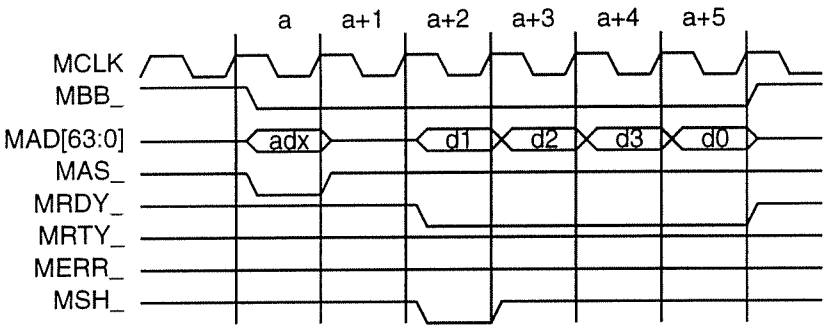


Figure 6: Mbus Timing Diagram for a Coherent Read (CR) Transaction

caches are expected to snoop on the Mbus and change state of blocks they may be cacheing as coherent transactions appear on the bus.

We overload some of the Mbus Level-2 semantics to implement Tempest's fine-grain access control and low-overhead handler dispatching; see Sections 5.1, 5.1.4 and 5.2.1.

4.5 Mbus Timing Diagrams

Figures 5 and 6 depict an Mbus uncached write transaction (WR) and a Coherent Read (CR) transaction respectively. Note that the bus arbitration phases are not shown. MBB_ is asserted by the master that initiates the transaction from the address cycle until the last data acknowledgment. The master may continue to assert MBB_ to retain ownership of the bus after the transaction is finished if the arbiter does not grant the bus to another master, or the transaction was not acknowledged by the Relinquish and Retry acknowledgment.

Note that in Figure 5 the master must produce the first word of the write data in cycle a+1, since the Mbus specifies that the slave may acknowledge the write in a+1. In Figure 5, the slave is wrapping the read data, and one or more caches are asserting the MSH_ line, indicating that the block being read is being cached (but not owned) in another cache. In this case the slave is fast enough to return the burst data in consecutive cycles, with no Idle cycles between Valid Data acknowledgments.

5 Theory of Operation

Vortex must achieve two goals: it has to provide fine-grain access control and related services for Tempest codes, and provide a means of integrating the network interface and protocol processor efficiently. This section first explains the theory behind Vortex's implementation of fine-grain access control, which relies on overloading the Mbus transaction response and cache-coherence semantics. Next the key component of the glue logic is presented: the Cacheable Control Register, which is used for fast fault and message handler dispatching.

Transaction Type	Tag State		
	ReadWrite	ReadOnly	Invalid or Busy
CR (read)	no action	assert MSH_	assert MIH_ & Bus Error ack
CI (write)	acknowledge	Bus Error ack	Bus Error ack
CRI (write miss)	no action	assert MIH_ & Bus Error ack	assert MIH_ & Bus Error ack
CWI	no action	no action	no action

Table 5: Vortex Mbus actions for fine-grain access control

5.1 Fine-Grain Access Control

Vortex implements fine-grain access control by associating two-bit tags with 128MB of Mbus physical memory, starting from physical address (PA) 0x0. To simplify the hardware, the Tempest block size supported directly by Vortex is the same as the Mbus cache block size, 32 bytes. Therefore, the tag store is 8Mbit, organized as 4M x 2bits, and is implemented with two 25nS 4Mbit x 1 Toshiba TC551402J SRAMS [Tos94].

Vortex monitors every transaction on the Mbus. The tag address is determined (see Section 9.4), and driven into the SRAM. If the current transaction is meaningful (it is a coherent operation to an address within Tempest space) and Vortex is enabled (see Section 6.4.4), the tag is compared against the transaction type, and some action is conditionally taken as given in Table 5. The exception to this sequence is when Vortex has generated the transaction as an Mbus master; in this case no action is ever taken (see Sections 5.1.4 and 9.1.2.)

5.1.1 Tempest Tag/Cache State Inclusion

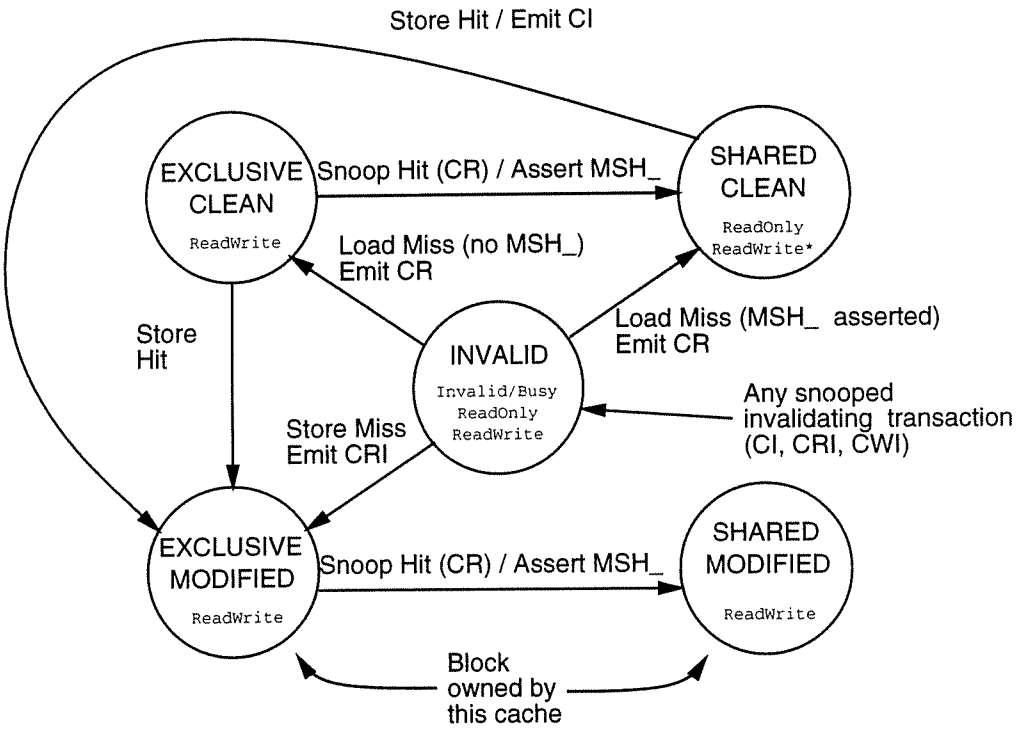
Table 5 describes actions taken on Coherent Mbus transactions, but Tempest specifies that every user program load and store (to Tempest memory) must be checked for access violations. We can only deal with Coherent transactions because Vortex is isolated from individual compute thread load and store instructions by the user processor's cache. It is of course possible to force all Tempest memory references to bypass the cache so that they are exposed to Vortex, but this would come at an incredible performance cost.

There is an alternative to bypassing the cache: by careful use of the Mbus shared signal (MSH_), we can maintain inclusion of the Tempest tag state in the cache state. Thus not every program load or store need appear on the Mbus, yet can still be checked by virtue of how the allowed cache states and the Mbus cache coherence protocol cause activity to appear on the Mbus in response to program loads and stores.

ReadWrite blocks are loaded into processor caches with no intervention by Vortex, and any cache can become the owner of that block – writes happen into the cache, and reads are satisfied by the cache. When a block tagged ReadOnly is first loaded into a processor's cache, Vortex asserts MSH_, which causes the requesting cache to "think" that another cache in the system was cacheing that block, so that the block is loaded into the requesting cache in the Shared Clean state. Thus loads from this block are satisfied by the cache, but a store to this block must be accompanied by an invalidation request on the Mbus (CI), which Vortex can detect and fault. Blocks tagged as Invalid or Busy are not permitted to be read into the cache at all: the first and all accesses (CRs and CRIs) to Invalid blocks are faulted. See Figure 7 for a diagram of Mbus cache block states with included Tempest tag states.

5.1.2 Enforcing Access Control

Vortex is able to intervene and fault offending bus transactions through the Mbus mechanism intended to support write-back caches: the Memory Inhibit (MIH_) signal. MIH_ is normally asserted by a cache when it is holding a modified copy of a block being requested by another processor's cache, and causes the memory controller not to acknowledge the transaction, leaving the bus free for the owning cache to respond with the data. As long as no cache in the system is the owner of a block which could cause a block access fault, Vortex is free to use MIH_ to inhibit memory and respond to the transaction on the



*note: Vortex does not assert MSH_ when a ReadWrite block is loaded, but MSH_ can be asserted by another cache which owns the block as it transitions from Exclusive Modified to Shared Modified, so that a ReadWrite block can be loaded into this cache in the Shared Clean state.

Figure 7: Mbus cache block states with included Tempest states, after [Edm91]

bus with a Bus Error acknowledgement rather than with a Valid Data acknowledgment. We have made sure that ReadOnly, Invalid and Busy blocks are never owned by a processor cache by preventing stores into the cache without some sort of bus activity as described above.

5.1.3 System Constraints on Enforcing Access Control

The fact that both the protocol and the user processors share the Mbus has implications for the use of MIH_ to intervene in bus transactions. The Tempest interface specifies that the protocol processor should be immune from fine-grain access control so that it may freely manipulate memory via the Tempest "`force-read`" and "`force-write`" protocol thread operations. In Typhoon, the RTLB simply ignores all memory references made by the protocol processor to accomplish this. However, we are restricted by the need to keep any cache from becoming the owner of a ReadOnly or Invalid block for our faulting mechanism to work properly. If the protocol processor were to modify a block it is cacheing via a `force-write`, it will become the owner of that block, and when the compute processor makes an reference to that block which must be faulted, Vortex will attempt to drive MIH_ and respond to the transaction while the protocol processor's cache tries to do the same. This condition violates the Mbus electrical specification and would probably lead to a system crash. Furthermore, Solaris threads are not necessarily bound to processors, so what we logically consider to be the protocol processor can be either physical processor. Finally, we need not dedicate one of the processors exclusively to protocol handling, instead using both to run compute threads [FW95]. In this case access control *must* be enforced on both processors.

For these reasons, we enforce access control unconditionally on both processors. Since Vortex snoops only coherent transactions, the protocol thread can manipulate Tempest memory via uncacheable aliases which do not generate coherent transactions. Protocol code accesses memory only through these aliases to perform `force-reads` and `force-writes`.

Note that we are free not to enforce Tempest fine-grain access control semantics on coherent transactions from non-cache Mbus citizens, such as the MSBI, the chip which bridges the SBUS and the Mbus in Sparcstation-20 class machines. The MSBI can never become the owner of a cache block, so there is no danger of violating tag inclusion. Furthermore, these non-processor Mbus transactions to Tempest memory are caused by I/O operations, which are in turn caused by device DMA, or invoked by the user through Tempest's bulk data transfer mechanisms. Message coherence is entirely scheduled by the user under Tempest semantics, and as such is independent of Tempest fine-grain access control. Since the MSBI is not a cache, it will never become owner of any cache block and does not present a problem to our faulting mechanism.

Two more entries in Table 5 deserve further explanation. CI transactions do not require MIH_ because normally the memory controller is the only responder to the CI transaction. Since CIs are conditionally acknowledged, Vortex must take responsibility for CIs, both in and outside of the memory under fine-grain access control (see Section 5.1.7.) Fortunately Sun's Mbus memory controllers can be configured to ignore CIs so that Vortex can acknowledge them. CWI transactions can be unconditionally ignored since they are generated in processors with write-through caches, or when a particular Solaris kernel memory copy routine is invoked, which we disallow. The other possible source of CWIs are block copy devices or bus bridges, like the MSBI. CWIs are generated by the MSBI when I/O is being performed as described in the preceding paragraph, and are ignored as explained there.

5.1.4 Support for Fine-Grain Access Control

The user must be able to manipulate the tag store directly from protocol handlers. At first glance this appears to be easy – simply expose the tag SRAM to the user in some portion of the Mbus address space, then map it into the protocol thread's virtual address space. However, there is some cost to overloading the cache state to include the Tempest tags; when tempest tags are downgraded, (that is, from ReadWrite to any other state, or from ReadOnly to Invalid/Busy) the cache state must be manipulated to reflect the change in the Tempest tags. This could be left to the user, or user-level library code, but leaving this responsibility to the user can have disastrous results; violating Tempest tag/cache state inclusion (such as accidentally allowing any cache to own a non-ReadWrite block) can result in both Vortex and the owning cache responding to a CR or CRI transaction. This will have disastrous results as described in Section 5.1.3.

Therefore, in order to guarantee that the user can not crash a Typhoon-Zero node, we must bind tag downgrades to cache coherence action in hardware. Hardware manipulation of cache states also has the advantage of being much faster than any software method, given the current SPARC instruction set support for cache line invalidation.

5.1.5 User Tag Shadow Space

To expose the tag store to the user, Vortex provides a tag *shadow space* on the Mbus – tags corresponding to Tempest memory blocks are located at a fixed offset from their memory blocks in the Mbus physical address space. Reading from a tag location (a byte read from a cache block-aligned address) returns the current tag in the SRAM. When a tag is written to this shadow space, Vortex first determines if the tag change is a downgrade. If not, the write proceeds normally and the tag SRAM is written when the acknowledgement is given on the bus. If so, Vortex replies to the write transaction with the Relinquish and Retry Mbus acknowledgement, which causes the requesting processor to relinquish ownership of the Mbus. Vortex then becomes bus master, and issues a CRI to the block address corresponding to the tag address. The cache block data is saved in the *Block Buffer*, a cacheable control register in the Vortex control space (see Section 5.2.2). While the CRI is on the bus, the tag SRAM is updated. When the processor writing the tag re-acquires the bus and attempts the write, the tag has been updated and since the write is no longer a downgrade, Vortex acknowledges the write, completing the tag downgrade process.

The downgraded block must be retrieved using a CRI transaction even if the downgrade is from ReadWrite to ReadOnly. This is because we must ensure that memory becomes the owner of the block after the downgrade; if a CR were performed to a block that was owned by a cache (i.e. the block is dirty), it would still be owned by the cache after the CR. This would mean that any further writes to the block (which is now in the ReadOnly state) would happen directly into the cache without appearing on the Mbus, and Vortex would be powerless to intervene. This would cause violation of Tempest tag inclusion, and could lead to electrical failure. Since there is no way to cause the cache state machine to transition from an owned state (Exclusive Modified or Shared Modified) to Shared Clean, our only option is to force the block's cache state to Invalid and incur a miss when the now ReadOnly block is referenced again.

The hyperSparc caches are virtually indexed. The Mbus provides an 8-bit virtual address field that must be driven during Coherent transactions as described in Section 4.1. Since block addresses are implied by tag shadow addresses, there must be some relationship between tag VAs and block VAs so Vortex can properly produce its CRI transaction. Our solution to this is to require User Tag pages to be virtually aligned with their corresponding Tempest memory pages. Since the VA field on the Mbus corresponds to the low 8 bits of the page VA, we require that block page VAs and tag page VAs have the same low 8 bits. The Vortex driver code ensures this relationship between user memory pages and user tag pages when allocating data pages.

Since performing burst writes out of the FPGA proved to be intractable due to timing problems (see Section 12.1, a downgraded block is not automatically written from the Block Buffer back into memory by Vortex. Instead the protocol handler must read the block out of the buffer and write it into memory. In order to avoid unnecessary copies, Vortex watches and records the state of the Mbus MIH_ signal during the CRI it issues to retrieve a block being downgraded. If MIH_ is not asserted, the block was read from memory and thus does not need to be written back to memory. The protocol handler learns of the state of MIH_ by reading back the tag as a halfword after writing it; if the LSB of the most significant byte of the halfword is 1, MIH_ was seen; if it is 0, MIH_ was not seen. This read is needed to flush the hyperSparc write buffer anyway, so it does not present any overhead.

5.1.6 System Tag Shadow Space

An additional tag shadow space, the System Tag space, is provided by Vortex. User Tag space tag downgrade semantics are not enforced in the System Tag space. It is intended for use by the kernel for tag initialization purposes only, and is not mapped into the protocol thread's virtual address space.

5.1.7 Coherent Invalidate Transactions

Vortex must be able to respond to CI transactions, since they indicate writes to ReadOnly blocks in Tempest memory. In ordinary Mbus systems, a memory controller is responsible for unconditionally acknowledging all CIs. There is no need for an acknowledge inhibit protocol (as there is for CRs and CRIs), because the memory controller is always the only responder to CIs. However, since the Mbus specification allows for coherent bus adaptors which may need to assume responsibility for acknowledging CIs, Sun's memory controllers permit their CI acknowledgement feature to be disabled, which is fortuitous for this project.

Vortex can be configured to ignore CIs, to acknowledge CIs unconditionally throughout main memory, or to enforce fine-grain access control and conditionally acknowledge or fault CIs as described in section 5.1. This behavior is determined by the **mode** register as described in Section 6.4.4. At system power-on, custom Forth code in the Sparcstation-20's non-volatile RAM disables the memory controller's CI acknowledgment and enable's Vortex's acknowledge-only mode.

When Vortex is acknowledging CI transactions, it responds to CIs in Vortex control register spaces (Sections 6.4.1–6.4.3) with an Uncorrectable bus error acknowledgement. The exception is when the CI to the UCREG space has been generated by Vortex's master logic in order to invalidate a CCR, (see Section 5.2.1) in which case the CI must be positively acknowledged in order to prevent a bus timeout.

5.2 Glue

Most of the rest of the logic implemented by Vortex is devoted to integration of the protocol processor and network with fine-grain access control. This section motivates and describes the main "glue" mechanism, the Cacheable Control Register (CCR). The CCR is used to quickly dispatch message and block access fault handlers on the protocol processor with low event-to-dispatch latency and minimum impact on bus bandwidth and throughput. Finally, the interface to the Myrinet network adapter, a single-wire "message interrupt" pin, is described.

5.2.1 Dispatching Handlers

When a block access fault occurs (or a message arrives, see Section 5.2.5), a handler must be dispatched on the protocol processor. One way to do this would be for Vortex to generate an interrupt when such an event occurs. This turns out to be undesirable for two reasons. The first is that the overhead of reflecting an interrupt on the Mbus through the Solaris kernel to user level code is quite high, probably on the order of milliseconds; this would severely limit message throughput. The other more fundamental limitation is that the only kind of interrupt that can be generated by an Mbus module is a Level 15 (broadcast) interrupt, which is certainly much more heavyweight than we need. Given that Mbus modules are usually processors, the Mbus designers provided much more support for a module to receive an interrupt than to generate one.

Since interrupting the protocol processor to dispatch handlers is not feasible, we instead require the protocol processor to poll on a Vortex control register for fault and message status. Polling, however, has a significant drawback as well: a processor spinning on the control register would flood the Mbus with useless transactions, which would significantly impact overall system performance.

5.2.2 Cacheable Control Registers

To get around the problem of bus saturation caused by polling, we introduce the concept of a Cacheable Control Register (CCR).[2] When a processor first references a CCR, it causes a cache miss which shows up on the Mbus as a CR transaction. Vortex returns 32 bytes of control data; further polls to the CCR are satisfied by the processor's cache.

When Vortex wants to change the contents of the CCR, it becomes bus master and issues a CI transaction to the CCR address, which causes the cached version to be invalidated. On the next poll the cache issues a CR and reads the updated copy of the block. This method has two advantages – first, bus

[2]The idea for Cacheable Control Registers is due to David Wood

bandwidth is not wasted with useless poll transactions, and second, a whole cache block worth of status can be transferred with one transaction.

Processors need to change the contents of CCRs in order to set and clear status registers or support context-switching, but this presents a consistency problem. We must somehow bind the processor operation that changes the CCR with the CI transaction that invalidates the cached copy. This is necessary to ensure that it is not possible for the processor clearing the CCR to read the cached copy (which is stale after the clear) before the CI has appeared on the Mbus. In Vortex, this is done by providing registers in a non-cacheable register space (the **setStatus** and **clearStatus** registers) which when written update the CCR and cause the CI. The write is acknowledged normally, but further writes will block (return the Mbus Relinquish and Retry transaction status response) until Vortex has issued the CI. Furthermore, other registers are provided which block when read (the uncached **handlerPC**) until the CI has been issued to provide the needed synchronization.

Using writes to clear CCRs is complicated by processors with write buffers. The CCR clear write is retired into the write buffer and does not appear on the bus until the write buffer is flushed. Thus only returning Relinquish and Retry on the initial write does not guarantee synchronization, because we must flush the write onto the bus. To flush the write buffer, a read must immediately follow the write; the hyperSparc does not have an address comparator so any read following the write will cause the flush. The superSparc will not flush its write buffer unless a read is to the same address as the write. So for the hyperSparc, we first write the status clear register, then read the uncached handlerPC, and on the superSparc we write the status clear twice, then read from it.

A better way to support CCR updates is to implement a range of registers, which when read the first time, responds with Relinquish and Retry and causes the CI, then responds with a normal acknowledgment when the CI is retired. The new CCR data is indicated by the address of the register that is read. This will only work well if the part of the CCR which must be written is small (such as some kind of status word), otherwise the registers will consume too much address space. This technique avoids the problems related to write buffers.

CCRs are made to be read-only by driving the MSH_ line while returning the block. If a processor tries to then write the CCR, a CI will be produced on the Mbus which Vortex will respond to with the Uncorrectable bus error acknowledgement. Allowing CCRs to be writable creates a set of problems which are very difficult to manage: loss of ownership of the CCR means Vortex would have to manage its cacheable register space like a real cache. Since there is no real benefit to making CCRs writable through cacheable register space, writing of CCRs is best performed through an uncached alias. See Sections 6.4.3 and 6.4.4 for details.

Since the Mbus supports virtually indexed caches, and hyperSparc processors use them, Coherent transactions such as CI must include the VA[19:12] field (see Section 4.1) to ensure correct cache snooping behavior. Therefore Vortex must somehow learn the low 8 bits of the virtual page address for the protocol thread's mapping of the cacheable register space in order to correctly invalidate CCRs. A programmable register (**regVA**) is provided in Vortex's control space; when the Vortex driver maps the cacheable register page it programs this register.

We separate the control register space of Vortex into two regions, cacheable and non-cacheable. The cacheable register page is same size as an MMU page, (4KB for the SPARC MMU) and the cacheable register page's page table entry is marked cacheable by the Vortex driver.

5.2.3 The handlerPC CCR

The fault dispatch register (known as the **handlerPC** CCR) contains the current fault status bit, message status bit, two software-controllable status bits, handler base PC and fault address (see Section 6.4.4.) It is organized such that the protocol processor can simply jump indirect through the least significant word of the block to arrive in a 16-entry dispatch table with room for 32 instructions in each entry. The 16 entries correspond to all sixteen possible fault conditions as determined by four fault status bits (see Section 6.4.4, item handlerPC). The base virtual address of the dispatch table is registered by the protocol thread via the **handlerPC base** register. Dispatching a handler is extremely fast using this method – 500nS (25 Mbus clock cycles) from the address strobe of a faulting block access to the first Doubleword read (the dispatch table PC) by the protocol processor's cache, and eleven instructions to the first user-level protocol code instruction from the read of the dispatch table PC.

Handler dispatch assembly code is given in Appendix C; note that this code is for the Typhoon-Zero simulator and as such makes references to the CM-5 Network Interface.

5.2.4 handlerPC example

Figure 8 gives a graphical depiction of CCR polling and invalidation for handler dispatching. The two timelines on the left represent the protocol processor's load/store unit and its cache. The middle line is Vortex, and the pair of lines on the right are the user processor and its cache. The sequence of events depicts the protocol processor first loading the CCR into its cache, and polling into the cache while a block access fault occurs. When the fault occurs, Vortex updates the handlerPC and issues a CI to the handlerPC address; the protocol processor then misses in it's cache and loads the new handlerPC, causing it to dispatch a handler. When the protocol code is finished, the relevant handlerPC status bits are cleared, and the new handlerPC is loaded again and polled on. Finally the user processor is given a signal to restart the faulting instruction. Note that real protocol code would read the fault information from Vortex and clear the status bits before dispatching the actual handler code for performance reasons; this allows another block access fault or message to happen while the protocol code is working.

A event by event explanation of Figure 8 is given below.

- Events 1 to 4

  The protocol processor loads the handlerPC CCR for the first time; the load causes a cache miss, which shows up on the Mbus as a CR to the handlerPC CCR. Vortex returns the handlerPC with four Doubleword Valid Data acknowledgments. MSH_ is asserted during the first acknowledgment.

- Event 4 and below

  The protocol processor spins on the cached version of the handlerPC

- Events 5 and 6

  The user processor stores to a block tagged Invalid. This appears on the Mbus as a CRI. The processor is blocked while the cache talks to the Mbus.

- Events 7 and 8

  Vortex translates the block PA to a tag address and looks up the tag, which is found to be Invalid. Vortex captures the fault status information and responds to the user processor with a Bus Error acknowledgement. The user processor then faults, ending up in a signal handler. Vortex produces a CI transaction to the handlerPC CCR address.

- Events 9 and 10

  The protocol processor polls but the CI at event 8 has invalidated the cached copy of the handlerPC, so the cache controller issues a CR to the handlerPC CCR address. The protocol processor is blocked while waiting for the cache to fill.

- Event 11

  Vortex returns the updated handlerPC CCR, which contains the address information for the faulted store.

- Event 12

  The BAF bit is now set in the handlerPC status word, causing the protocol processor to dispatch a handler for the block access fault.

- Event 13

  User protocol activity. This may include tag upgrades, downgrades and message sends. In this case a likely sequence is to retrieve a copy of the faulted block, and upgrade the tag to ReadWrite.

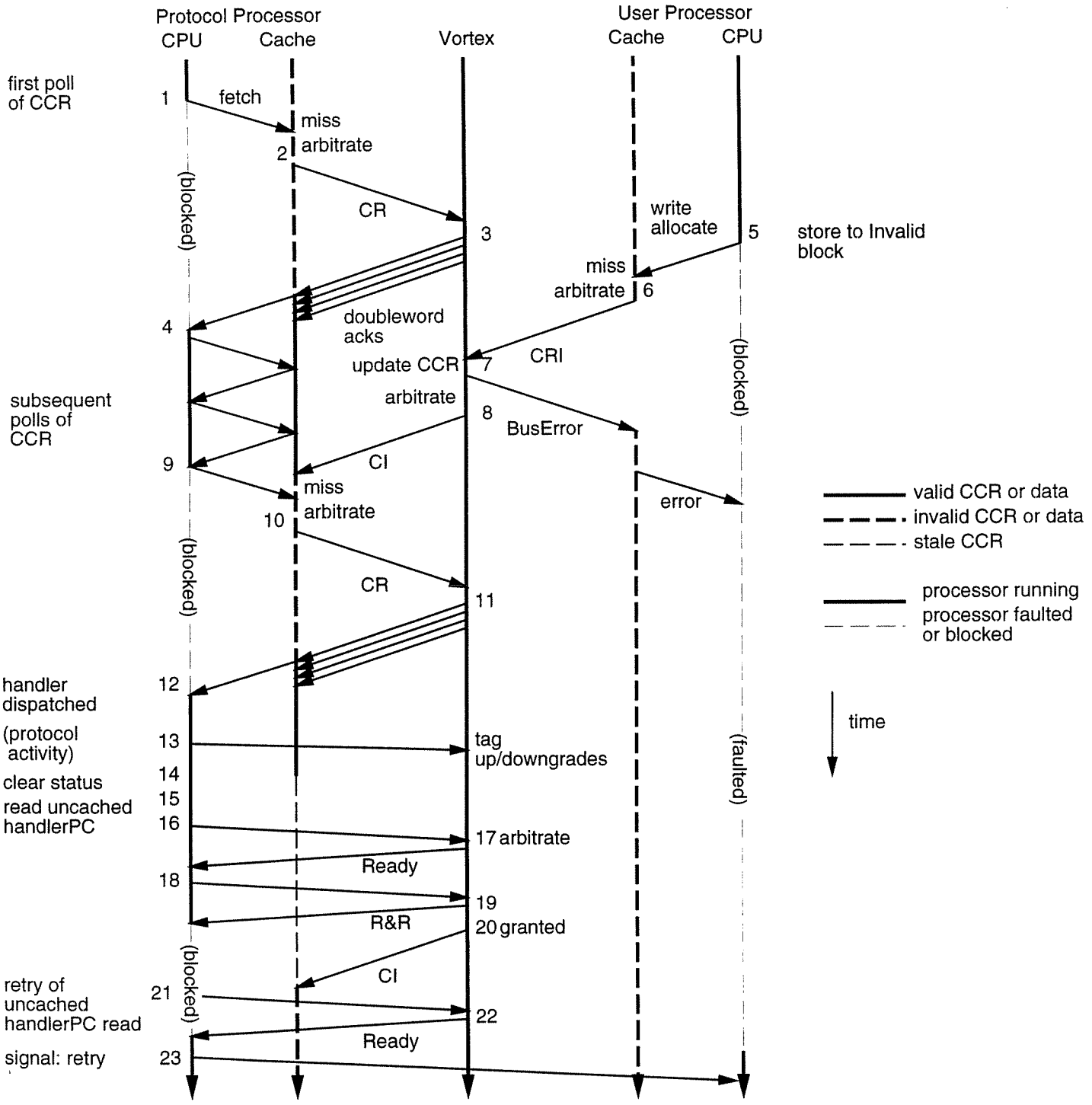


Figure 8: CCR Polling and Update/Invalidate

- Event 14

  The end of the protocol handler. The protocol thread performs a store to the clearStatus register in order to clear the block access fault status bit in the handlerPC CCR. At this point the cached copy of the handlerPC is stale. The clearStatus write is retired into the hyperSparc's write buffer and does not yet appear on the bus.

- Event 15

  The protocol handler now reads the handlerPC status word from UUREG (user uncacheable register) space. This is done to flush the clearStatus write onto the bus. We read the status word because it may indicate that a message was received at our node; we need a read to flush the write buffer, and reading this register is at least useful to us. We can detect another fault or message early by checking the handlerPC status word now.

- Event 16

  The clearStatus write is flushed out of the write buffer onto the bus by the uncached handlerPC read.

- Event 17

  Vortex updates the handlerPC status word, and arbitrates for the bus in order to CI the handlerPC CCR.

- Event 18

  The uncached handlerPC read is placed on the Mbus.

- Event 19

  Vortex responds to the handlerPC read with the Relinquish and Retry acknowledgment, because it has a pending master operation (the handlerPC CCR CI caused at Event 17.) We respond with R&R so that the protocol thread will block, and the protocol processor will relinquish the bus. This permits Vortex to acquire the bus, while guaranteeing that the protocol thread will not be able to read the stale handlerPC CCR in it's cache.

- Event 20

  Vortex is granted the Mbus and it issues the CI to the handlerPC CCR address. When the CI is snooped by the protocol processor's cache, the stale handlerPC is invalidated.

- Events 21 to 23

  The protocol processor eventually regains ownership of the Mbus and retries the uncached handlerPC read. Now that Vortex has retired the CI, it responds with the Valid Data acknowledgment while returning the handlerPC status word. This causes the protocol thread to un-block, and signal the user processor to retry the offending store. When the protocol thread returns to the dispatch loop and reads the handlerPC CCR again, it will cause a cache miss and the updated CCR will be fetched from Vortex.

5.2.5 Network Support

Since the network interface does not reside on the Vortex module, we are not able to provide a high level of integration. For instance, it would be desirable to support invalidate (downgrade) & send as a Vortex primitive mechanism, or to directly support Active Messages [vECGS92] by extracting the message handler PC from the message and including it in the handlerPC CCR. At the very least, however, we would like to integrate the message handler dispatch method with the block fault handler dispatch method in some way.

We accomplish this through the use of a single input pin on the Vortex module, called the MessageArrival pin. The pin is a bistate input; to indicate that a message has arrived, a network interface must simply toggle the state of the pin. This causes Vortex to set the Message Arrived status bit, and

invalidate the **handlerPC** CCR as described above. The protocol processor then dispatches a handler which extracts the message from the network.

Our network interface must be able to accommodate this scheme. The Myrinet LanAI card has a programmable low-level protocol engine, which happens to be connected to two status LEDs on the board. We remove one of these LEDs from the board and attach a wire from the socket to the MessageArrival pin, and reprogram the LanAI to toggle the state of the LED when a new message has arrived.

6 Architectural Overview

This section presents the programmer's architectural view of Vortex. Vortex logic is partitioned into three control units: the slave, the master, and the network interface. The Vortex slave logic (see Section 9.1) is responsible for maintaining the Mbus memory map: snooping on Tempest memory references, servicing the User and System Tag shadow spaces, and responding to three control register spaces. The master logic (see Section 9.2) is invoked by the slave to perform user tag downgrades and invalidations of CCRs. The network interface (Netboy) fields message arrival "interrupts" from the Myrinet LanAI card and communicates with the master to update and invalidate the handlerPC CCR.

6.1 Mbus Memory Map

Vortex provides two functions on the Mbus: snooping on memory references to enforce Tempest tag semantics, and responding to tag and register reads and writes. Figure 9 shows the Tempest snoop space, the two tag spaces, and the three control register pages. There are some complications that are related to snooping and tag SRAM address generation caused by the Sparcstation-20 memory controller which are described in the following sections.

6.2 Tempest Snoop Space

Coherent transactions (CR, CRI and CI) to physical addresses 0x0 to 0x008000000 (128MB), 0x010000000 (256MB) or 0x020000000 (512MB) are checked against the Tempest tag stored in the corresponding tag SRAM location. Note that all transactions created by Vortex are not subject to Tempest access control. Although these address ranges imply up to 512MB of Tempest memory per node, in reality only 128MB is allowed in all three scenarios due to the size of the tag SRAM.

The Sparcstation-20 memory controller does not guarantee that physical memory will be contiguous, unless the machine is populated entirely with 64MB DRAM SIMMs. We support 64, 32 and 16MB SIMMS, but only snoop on a maximum of 128MB of memory. Because there are holes in the memory map when using 32 and 16MB SIMMS, we must be able to adjust the maximum PA which must be snooped on; the **adxConfig** register can be set to one of four modes as shown at the bottom of Figure 9 to accomplish this. There are two 4x32MB SIMM configurations because we misunderstood how the memory controller actually mapped this SIMM configuration onto the Mbus. The physical locations of the SIMMS for the different address configuration modes are shown as gray boxes in the Tempest snoop space.

Tempest fine-grain access control is only enforced in the snoop space when the **mode** register indicates that it should be enforced; see Section 6.4.4 for details.

6.3 Tag Shadow Spaces

There are two tag shadow spaces, as described in Sections 5.1.5 and 5.1.6. The total tag space size is 1024MB, divided into 512MB of *user* tags, where cache-tag/Tempest tag inclusion is maintained during tag downgrades, and 512MB of *system* tags where inclusion is not maintained. Each tag space takes up 512MB of physical memory, though only 128MB worth of tags are present. The inflation is due to the shadow invariant: tags must always be at a constant offset from the memory blocks they belong to. Because the 8x16MB SIMM snoop configuration spans 512MB of physical address space, each tag space must be inflated to the same size as the snoop space. The aforementioned **adxConfig** register also

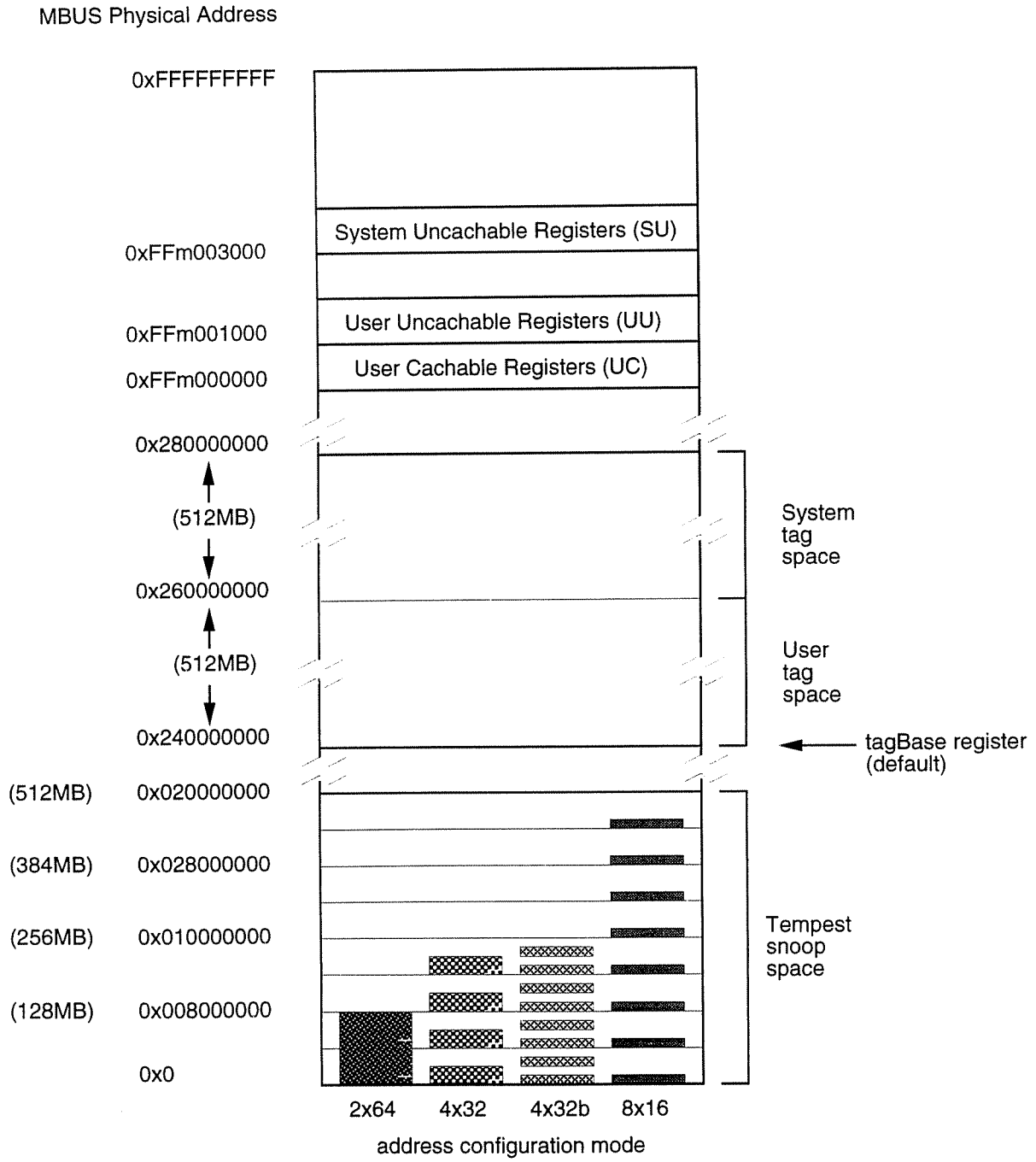


Figure 9: Mbus memory map as provided by Vortex

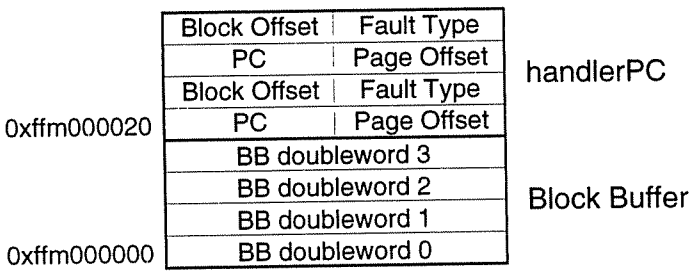


Figure 10: User Cacheable Register Space

controls the translation of block PAs to SRAM addresses based on what SIMM mode is selected. See Section 9.4 for details on the address translation.

6.4 Control Register Spaces

The Mbus specification provides a control register space for each of the 16 possible Mbus citizens, beginning at PAs `0xffm000000` (where `m` is the citizen's MID), and with byte length `0x001000000`. Vortex implements three MMU page-sized register spaces: *User Cacheable* (UCREG), *User Uncacheable* (UUREG) and *System Uncacheable* (SUREG). The layout of all three register spaces is the same, but some types of accesses are not allowed in some spaces but reading registers above the **handlerPC** in the UCREG space will result in undefined data being returned. The distinction between the System and User Uncacheable spaces is made for protection purposes; some registers should not be writable by the user, such as the **mode** register described above. The UCREG space only responds to CR transactions and CIs from Vortex; any other type of access is considered an error.

6.4.1 User Cacheable Registers: base `0xffm000000`

The UCREG space consists of two cacheable control registers, the **handlerPC** and the **Block Buffer**. See Figure 10 for a graphical depiction of this register space.

Offset	Name	Allowed SIZE	Allowed TYPEs	Alignment	Side effects
0x20:	handlerPC	Burst32	CR	DoubleWord	MSH_ driven
0x00:	Block Buffer	Burst32	CR	DoubleWord	MSH_ driven

User Cacheable Registers (UCREG)

Only CR transactions (and CIs from Vortex) are allowed to the UCREG registers. Vortex drives MSH_ during CRs to this register space (see Section 5.2.2.) If either of the registers are modified, (either by Vortex or by writing to particular registers in the Uncacheable register spaces, see below) Vortex will issue a CI to the appropriate address to make sure any cached copies are invalidated. Non CR or CI transactions to this space are acknowledged with the Uncorrectable bus error acknowledgement; CI transactions from masters other than Vortex are also acknowledged with the Uncorrectable error.

Vortex supports wrapping (see Section 4.3) on CR transactions to the UCREG space.

6.4.2 User Uncacheable Registers: base `0xffm001000`

The User Uncacheable Register space contains general Vortex configuration and status registers. Offsets from the base page address are given in the table below.

Offset	Name	Allowed SIZE	Allowed TYPEs	Side effects
0x59	clearStatus mask	Byte	WR	CI handlerPC
0x58	setStatus mask	Byte	WR	CI handlerPC
0x55	regVA	Byte	RD	none
0x54	cidel	Byte	RD	none
0x53	adxConfig	Byte	RD	none
0x52	mode	Byte	RD	none
0x50	error	HalfWord	RD	clear-on-read
0x48	tagBase	DoubleWord	RD	none
0x40	handlerPC base	Word	RD,WR	CI handlerPC (write)
0x30-0x3F	undefined	–	–	–
0x28	handlerPC: Block Offset/Fault Type	DoubleWord	RD	none
0x20	handlerPC: PC/Page Offset	DoubleWord	RD	none
0x18	Block Buffer word 3	DoubleWord	RD	none
0x10	Block Buffer word 2	DoubleWord	RD	none
0x08	Block Buffer word 1	DoubleWord	RD	none
0x00	Block Buffer word 0	DoubleWord	RD	none

User Uncacheable Registers (UCREG)

In general, only WR and RD TYPEd transactions are supported in this space. All remaining transaction types (all the Coherent transactions) are responded to with an Mbus Uncorrectable bus error acknowledgement. Proper alignment of accesses is not checked, and unaligned reads return undefined data while unaligned writes result in undefined data being written. If a write is performed to a read-only register, the write is acknowledged but ignored; therefore misaligned writes to read-only registers have no effect. Misaligned writes are ignored due to problems with the hyperSparc write buffer. Section 12.5 contains details on why malformed writes are not acknowledged with bus errors. Some registers have side effects when read or written as described in Section 6.4.4.

6.4.3 System Uncacheable Registers: base `0xffm003000`

The System Uncacheable Register space contains general Vortex configuration and status registers. Offsets from the base page address are given in the table below.

Offset	Name	Allowed SIZE	Allowed TYPEs	Side effects
0x59	clearStatus mask	Byte	WR	CI handlerPC
0x58	setStatus mask	Byte	WR	CI handlerPC
0x55	regVA	Byte	RD,WR	none
0x54	cidel	Byte	RD,WR	none
0x53	adxConfig	Byte	RD,WR	none
0x52	mode	Byte	RD,WR	none
0x50	error	HalfWord	RD,WR	clear-on-read
0x48	tagBase	DoubleWord	RD,WR	none
0x40	handlerPC base	Word	RD,WR	CI handlerPC (write)
0x30-0x3F	undefined	–	–	–
0x28	handlerPC: Block Offset/Fault Type	DoubleWord	RD,WR	CI handlerPC (write)
0x20	handlerPC: PC/Page Offset	DoubleWord	RD,WR	CI handlerPC (write)
0x18	Block Buffer word 3	DoubleWord	RD,WR	CI Block Buffer (write)
0x10	Block Buffer word 2	DoubleWord	RD,WR	CI Block Buffer (write)
0x08	Block Buffer word 1	DoubleWord	RD,WR	CI Block Buffer (write)
0x00	Block Buffer word 0	DoubleWord	RD,WR	CI Block Buffer (write)

System Uncacheable Registers (SUREG)

As in the UUREG space, only WR and RD TYPEd transactions are supported in SUREG space. All remaining transaction types (all the Coherent transactions) are responded to with an Mbus Uncorrectable bus error acknowledgement. Proper alignment of accesses is not checked, and unaligned reads return undefined data while unaligned writes result in undefined data being written. All registers are writable in this space; writing some registers produce side effects as described in Section 6.4.4.

6.4.4 Register Descriptions

- Block Buffer

  The **Block Buffer** contains the most recently downgraded Tempest memory block. Its reset value is unknown. It appears both as a Burst32-sized CCR in UCREG space, and as individual DoubleWord registers in UUREG and SUREG spaces. The Block Buffer is read-only in UCREG space, and its component Doublewords are read-only in UUREG space, but read-write in SUREG space. Writing a Block Buffer DoubleWord in SUREG space causes Vortex to request the bus and issue a CI to the Block Buffer address in UCREG space. The block buffer is writable in order to support context-switching.

- handlerPC

  The **handlerPC** register contains the current fault status and message arrival status information. It appears as a Burst32-sized CCR in UCREG space and as individual DoubleWord registers in UCREG and SUREG spaces. It is read-only in UCREG space, and its components are read-only in UUREG space, but read-write in SUREG space. Writing a handlerPC DoubleWord in SUREG space causes Vortex to request the bus and issue a CI to the handlerPC block address in UCREG space; this ensures CCR register-cache coherence if the register is written. Writing the handlerPC from SUREG space is provided to support context-switching.

  Reading the handlerPC PC/Page Offset Doubleword in UUREG or SUREG space will be acknowledged by the Relinquish and Retry acknowledgment if there is an outstanding master request. This behavior is used by the protocol thread to cause it to block until the CCR has been updated and invalidated, which ensures that the protocol thread will not read stale CCR data from its cache. See Section 5.2.2 for details.

  The protocol processor should cache this register from UCREG space; during a block fault or a message arrival Vortex updates the register contents and issues a CI to the **handlerPC** UCREG

address in order to cause a cache miss on the protocol processor. The handlerPC register is made up of four word-sized registers concatenated into two Doubleword registers: the **PC word**, the **Page Offset word**, the **Block Offset word**, and the **Fault Type word**. Here is the layout for the DoubleWords that comprise the handlerPC register:

Bit position in DoubleWord

```
6666555555555544444444443333333333  3322222222221111111111000000000
3210987654321098765432109876543210  1098765432109876543210987654321O

Block Offset word:                  Fault Type word:
0000000000000000000BBBBBBB00000     00000000000000000000000000TTW00

PC word:                            Page Offset word:
HHHHHHHHHHHHHHHHHHHHHHSSSS000000    00000000000PPPPPPPPPPPPPPPP0000
```

Key:

Symbol	Bit width	Description
H	22	handler PC base
S	4	fault/message status
P	17	page number of faulted block (PA[28:12]; max PA = 512M-1)
B	7	page offset of faulted block (PA[11:5])
T	2	tag of faulted block
W	1	fault access type: 0 = read, 1 = write

The Block offset word contains the most recently faulted block's physical offset from the beginning of the physical page, which is contained in the Page Offset word. The Fault Type word contains the fault type tag for the faulted block; if the protocol thread changes this tag before the fault is handled, the change will be reflected here by the fault tag comparator (Section 9.3.4.) Finally, the PC word contains the handlerPC base concatenated with the status word to form a PC that the protocol processor can jump indirect through.

The status word consists of two software controlled status bits, the MSG bit which is set when a message arrives at the Myrinet interface (Netboy), and the BAF bit which is set when a block access fault occurs. The layout of the status bits corresponds to the layout of the status set/clear mask register layouts:

Bit position in Status word

```
3210
10MF
```

Key:

Symbol	Description
1	Software-controlled Status bit 1
0	Software-controlled Status bit 0
M	Message Arrived Flag bit
F	Block Access Fault Flag bit

Note that the page number, page offset, tag and fault access type fields are undefined until a block fault has occurred.

- handlerPC base

  This register is used by the protocol thread to register the base virtual address of the dispatch table (as described in Section 5.2.3.) It is readable and writable only as a Word-sized quantity; misaligned

or improperly SIZEd transactions result in undefined behavior. Only the 22 most significant bits are mutable, and after Mbus Reset handlerPC base contains the value 0x0b0bdb00. There is one side effect to writing this register – Vortex will acknowledge the write and generate a CI to the handlerPC block in UCREG space. If the master logic is busy, Vortex will respond to the write with a Relinquish and Retry acknowledgment so the bus will become free.

- tagBase

  The tagBase register is used to configure the tag shadow space mapping logic in Vortex. The register is read-only in UUREG, and is read-write in SUREG. It appears as a DoubleWord, though only bits 35 through 30 are mutable; the rest of the bits are hardwired to 0.

  The six bits comprising the tagBase register represent the high 6 bits of the physical address of tags. The tag base register contains the value 0x0000 0002 4000 0000 after Mbus Reset, so User Tag space begins at PA 0x240000000 and ends at PA 0x25fffffff; System Tag space begins at PA 0x260000000 and ends at PA 0x27fffffff by default.

  The tagBase register can be set to any value, so care must be taken when configuring this register. It is entirely possible to map tags into a memory space being handled by a memory controller, or to move the tags up into the Mbus control register space. Doing so will almost certainly result in Vortex failure at the very best, and system failure at the worst.

- error

  The error register is a halfword-sized register, though only the low byte is used. it is read-only in UUREG space, and is read-write in SUREG. After reset, the error register contains the value 0x00ff, and should be read once to clear it.

  If an error has occurred either due to a transaction in which Vortex was the targeted slave, or due to an error encountered when Vortex was bus master, the error register will be non-zero. When the error register is read in either uncacheable space, it is cleared to 0x0000, indicating no error. If another error happens before any processor has read the error value, the new error syndrome is not recorded, but a bit is set indicating that consecutive errors occurred without an intervening read of the error register. The first error syndrome is preserved in the register.

  When an error is encountered by the slave side, Vortex usually acknowledges the transaction with some sort of bus error; it is assumed that the handler for that bus error will eventually read the error register to find out what happened. In some cases, the error register is set by the slave even when it does not return a bus error; please see Section 12.5 for an explanation of this behavior. When an error is encountered by the master side, there is no way to report the error to a processor. Also see Section 12.5 for details on this problem.

Bit position in HalfWord
1111110000000000
5432109876543210

00000000PTSSSSSS

Key:

Symbol	Bit width	Description
P	1	Error Pending bit
T	1	Error Type bit
S	6	Error Syndrome word

  – Error Pending bit

    If the Error Pending bit is set, this indicates that one or more errors has occurred since the error reported by the Type and Syndrome fields happened.

- Error Type bit
  When this bit is set, the error described by the syndrome code was encountered by the Master state machine, otherwise, the error was encountered by the Slave state machine.
- Error Syndrome word

Error Type	Syndrome	Meaning
T = 0 (Slave Error)	0b000000	No Error
	0b000001	Fault Pending
	0b000010	Tag Space Error
	0b000100	UCREG Error
	0b001000	Bad transaction TYPE in {UU,SU}REG
	0b010000	Write to read-only UUREG register
	0b100000	Illegal CI encountered

Error Type	Syndrome[5:3]	Meaning	Syndrome[2:0]	Meaning
T = 1 (Master Error)	0b001	CWI failed	0b001	handlerPC CI failed
	0b010	CRI failed	0b010	block tag downgrade failed
	0b100	CI failed	0b100	Block Buffer CI failed

Slave Error Syndrome Explanations:

* Fault Pending

  Vortex only has enough state to record one block access fault at a time. Therefore, if another block fault happens before the current fault has been handled by the protocol thread, the Mbus Timeout error is returned to the processor causing the second block access fault, and to any other processor causing consecutive faults. Since the Timeout error acknowledgement is used exclusively by Vortex to indicate Fault Pending, the offending CPU knows immediately to back off and retry the access later. This means that multi-threaded user code or running user code on the protocol processor is supported, although crudely. In the base Typhoon-zero system, where one processor is dedicated to the protocol thread, and the user processor is running single-threaded user code the possibility of back-to-back block access faults does not exist.

* Tag Space Error

  Vortex indicates Tag Space error when:
  A non-RD or non-WR Mbus transaction TYPE occurs to either tag space, or
  A non-byte SIZEd or non-DoubleWord aligned write to either tag space occurs.
  If the transaction type was illegal, or a malformed read is performed to tag space, Vortex will return an Uncorrectable error. If a malformed write is performed to tag space, Vortex acknowledges the write with Ready, ignores the write, though it still sets this register. This somewhat puzzling behavior is explained in Section 12.5.

* UCREG Error

  Vortex returns the Mbus Uncorrectable error and indicates UCREG error when:
  A non-CR, non-CI or non-RD Mbus type transaction is directed at UCREG space, or
  A non-Burst32 sized transaction is directed at UCREG space.

* Bad transaction TYPE in {UU,SU}REG

  Vortex returns the Mbus Uncorrectable error and indicates Bad transaction TYPE when a non-RD or non-WR transaction is directed at UUREG or SUREG spaces.

* Write to read-only UUREG

  Vortex sets this error code when a write is performed to a read-only UUREG-space register, though the write is acknowledged normally with Ready and the write data is discarded.

* Illegal CI encountered

  Vortex returns the Mbus Uncorrectable error and sets this bit when a CI is attempted to:
  Any tag space, or
  Any uncacheable register space, or
  The UCREG space, when the master issuing the CI is not Vortex.

Master Error Syndrome Explanations:

Bits 5-3 of the Syndrome field indicate the request that the Master state machine was handling when it encountered an error. Any non-Ready acknowledgement on the Mbus during Master operation is considered an error condition. Note that although Vortex never generates CWIs, it is able to do so. Bits 2-0 of the Syndrome field indicate the current request that the masterControl state machine (see Section 9.2.1) was handling when the Master state machine returned an error.

Since there is no room to encode what the precise nature of the error was, the master error reporting is of limited use. Furthermore, since Vortex is not a processor it can do very little about a master error. There is currently no way to signal a master error to any processor in the system. Section 12.5 explains why this is so.

- mode

  The mode register is read-only in UUREG space; writes to mode in UUREG are ignored. It is read-write in SUREG space. The mode register is a byte, though only the low two bits are mutable and can take one of four values:

  - `0x0`: mode = standby

    Vortex will not enforce Tempest fine-grain access control, but it still maps tags and registers in this mode. CIs are assumed to be acknowledged by the memory controller.

  - `0x1`: mode = ackcis

    In this mode, Vortex still does not enforce Tempest fine-grain access control, however, it will now acknowledge CIs to legal addresses (any address not in the Tag spaces or not in register spaces) with the Mbus Ready acknowledgment. CIs by Vortex's master to the UCREG space are likewise handled. CIs to illegal addresses (tag spaces, Uncacheable register spaces, and Cacheable register space by a processor) are acknowledged with the Mbus Uncorrectable error.

  - `0x2, 0x3`: mode = run

    In this mode, CIs are handled as in mode = ackcis, but Tempest fine-grain access control is enforced on Mbus physical addresses from `0x0` to `0x007ffffe0`, `0x00ffffe0` or `0x01ffffe0` (depending on the value stored to the Address Configuration Mode register, adxConfig) during Mbus CR, CRI and CI transactions.

  mode is set to `0x0` (standby) on Mbus Reset.

- adxConfig

  adxConfig is the Address Configuration Mode register. It is read-only in UUREG space (writes are ignored) and read-write in the SUREG space.

  This register controls the upper bound on the Tempest physical memory area and how the Tag SRAM address is generated. The existence of this register is due to how Sun's memory controller

works. In systems with SIMMS smaller than 64MB installed, there are holes in the physical memory map. Without this configuration register, a system with, say, 128MB of memory installed as 8 16MB SIMMs could only have 32MB of Tempest memory. This configuration register allows us to skip past the holes in the address map and cover the full 128MB as Tempest memory.

The adxConfig register is one byte wide and can take 3 values:

- 0x0: adxConfig = 2x64MB SIMMs (highest Tempest PA = 0x007ffffe0)
- 0x1: adxConfig = 4x32MB SIMMs (highest Tempest PA = 0x00fffffe0)
- 0x2: adxConfig = 8x16MB SIMMs (highest Tempest PA = 0x01fffffe0)
- 0x3: adxConfig = 4x32MB SIMMs mode B (highest Tempest PA = 0x00fffffe0)

There are two versions of the 4x32MB configuration because the memory controller actually maps 32MB SIMMs differently than we thought; a 32MB SIMM appears as two 16MB sections separated by a 16MB gap within the 64MB allocated to a group of four SIMM slots on the Sparcstation-20 motherboard. See Figure 9 and Sections 6.2 and 6.3 for details on SIMM configurations.

If a system has a hybrid SIMM configuration, it is best to put the larger SIMMs lower in the physical address space, then pick the most appropriate setting for adxConfig.

adxConfig is set to 0x1 (4x32MB) on reset.

- cidel

  cidel is read-only in UUREG space, and read-write in SUREG space.

  This register controls how many cycles from the Mbus address cycle to the acknowledgment cycle of a CI when Vortex is in charge of acknowledging CIs (mode = ackcis or mode = run.)

  cidel is one byte wide but can only take values from 0x0 to 0x0f. CI acknowledgments come at cycle $A + (6 + cidel)$, unless the CI was illegal or caused a block access fault, in which case they always happen during A+6.

  cidel defaults to 0x0 on Mbus reset.

- regVA

  Vortex is designed to work with the Ross hyperSparc CPU. Since the hyperSparc uses a virtually indexed cache, and Mbus supports an 8-bit VA field for use during Mbus coherent transactions, Vortex must be sure to drive the VA field with the correct value when performing a CI to the UCREG space, otherwise the CI would most likely be ignored by caches in the system.

  To this end, the regVA register is programmed by the Vortex driver when the UCREG page Virtual Address is mapped by the user. The low 8 bits of the user virtual address of the UCREG page is recorded. Vortex then drives this register onto the Mbus VA field during address cycles of CI transactions to the Cacheable Control Registers in the UCREG space.

  regVA is one byte wide, and is set to 0x0 on Mbus reset.

- setStatus & clearStatus masks

  The setStatus and clearStatus registers are write-only in both UUREG and SUREG spaces, and are one byte wide, though only the low four bits are sensed. Reading this register in either space will return an undefined result.

  The 4-bit status register, which is part of the handlerPC block's PC word, indicates the current fault status, current message status, and provides two bits of status information which are not used by the hardware, but may be used by software.

  When written with a non-zero value, the set mask sets the status bits corresponding to the bits written to the mask. Likewise the clear mask clears the status bits at those locations. Writing a set or clear mask of 0x0 leaves the status register unchanged. Writing this register has the side effect of causing Vortex to request the bus and issue a CI to the handlerPC CCR; this happens even if 0x0 is written to either mask.

The layout of both masks is as follows:

Bit position in Byte
76543210
iiii10MF

Key:

Symbol	Bit width	Description
i	4	ignored; should be written as '0'
1	1	set/clear Soft Status bit 1
0	1	set/clear Soft Status bit 0
M	1	set/clear Message Pending Flag
F	1	set/clear Block Access Fault Flag

7 FPGAs and FPGA Design Issues

Field Programmable Gate Arrays, or FPGAs, are a form of programmable logic pioneered by Xilinx and/or Altera, depending on whose lawyers you believe. They are characterized by a structure which is reminiscent of gate arrays, that is, a rectangular array of function blocks connected by some kind of routing structure. The difference between gate arrays and FPGAs lies in the routing technique; gate arrays are left unmetalized until a customer specifies the routing for a design in the form of metalization masks, while FPGAs are fabricated in one step and have a programmable routing structure. Having been partially derived from traditional programmable logic such as PALs, FPGAs have programmable function blocks, while traditional gate arrays do not.

There are two major classes of FPGAs available on the market today: antifuse-based and SRAM-based. This classification refers to how the routing between logic elements and the logic element functions are implemented. SRAM-based devices have the advantage of in-circuit reprogrammability, but must be programmed each time they are powered up and so require some external support chip(s) to store the configuration data. Antifuse-based devices, like PROMS, are programmed once and hold their programs across power cycles, but are not mutable once programmed. Theoretically, antifused-based FPGAs are faster than their SRAM-based counterparts since the delay through antifuse router switch elements is much lower than through pass-gate switch elements. In practice this only applies to routing structures which are highly segmented.

FPGAs are very attractive to logic designers since the time from "tapeout" of a design to hardware testing can be measured in hours, rather than weeks for gate arrays and other ASICs. SRAM-based FPGAs also afford a degree of comfort to designers, since minor and even major bugs can usually be immediately fixed without the respin overhead (both cost and time) of ASICs.

Given that we want to implement Vortex, we need to choose some hardware upon which to build it. Our goal of quickly building the Typhoon-Zero system made FPGAs look attractive; using FPGAs would allow us to sidestep all of the difficult and time-consuming issues involved in standard-cell or full custom ASIC design and provide us some degree of flexibility to alter the design as new research ideas arise. On the other hand, the timing requirements of the Mbus are certainly aggressive for FPGAs, and most of the design effort was directed at managing these requirements. From a system design point of view, the trade-off between ASIC problems and FPGA problems was worthwhile, though in the end it took so long to get the quantities of the FPGAs we needed it probably would have been faster to design an ASIC. Nevertheless, by using FPGAs, we can still reprogram the Vortex logic (to a certain degree, see Section 12.)

7.1 FPGA Selection Criteria

There are several requirements imposed by the our design requirements for and electrical characteristics of the Mbus influenced our choice of FPGA to use for Vortex. We must be able to run at at least 40MHz,

which is fairly aggressive for an FPGA design. We must meet Mbus electrical and timing constraints, and deal with ground bounce, a parasitic effect prevalent in high-speed bussed designs. Finally, we need enough logic capacity to fit all the logic in one or two FPGAs.

7.1.1 Clock Speed

Foremost, the FPGA we choose must be clockable at 40MHz at the least and preferably 50MHz; the Sparcstation-20 Mbus runs at 50MHz by default but can be slowed to 40MHz. Most current-generation FPGAs are capable of toggling internal flip-flops in excess of 150MHz, but that rate drops rapidly to the vicinity of 20MHz when routing, logic and pin delays are factored in. We need FPGAs that can support a modest amount of logic between flip-flops while maintaining fast intra-function block routing in order to reach 50MHz with our design.

7.1.2 Clock to Q delay

The Mbus specifications specifies a maximum delay between the positive edge of the Mbus system clock seen at a module driving the bus and the appearance of data or control signals that the module must drive during the following clock cycle. This is to allow for clock skew and signal propagation delay of bus data and control signals while still meeting the setup time for flip-flops listening to bus signals at receiving modules. The so called Clock to Q delay is specified at 1.4nS; meeting this timing constraint requires careful design even when using ASICs. In order to even come close to meeting this constraint, all Mbus signals must be driven out of flip-flops which reside on I/O cell pads, so our FPGAs must have I/O cell registers. Furthermore, the FPGA I/Os must have very fast slew rates to aid in meeting the Clock to Q delay constraint.

7.1.3 Drive Strength

Our FPGAs must be able to sink and source enough current to meet the Mbus AC and DC drive strength requirements. Fortunately, I/O drivers on most FPGAs available today are capable of more than enough sink and source current to satisfy the Mbus electrical requirements.

7.1.4 Ground Bounce

Ground bounce is a transient parasitic phenomenon that occurs in high-speed devices, and is caused in part by device packaging. When several I/O pins are switched simultaneously at high slew rates, (which is of course common when driving a bus) the sum of the driving currents through each I/O pin can be quite substantial. The problem arises because this large current must be returned through the ground pins on the device; when there are many fewer ground pins than driving I/O pins, each ground pin is conducting a large portion of the return current. This current can become large enough to induce a significant voltage across the ground pins' lead inductances. This raises the ground reference voltage, which leads to decreased noise margins at receiving modules. Lowered noise margins can result in logic values being sensed improperly by receivers if the bounce settle time exceeds the clock cycle time. We must pay careful attention to choosing the proper package for the FPGA and to designing the board power distribution system to overcome this problem.

7.1.5 Density

The preceding speed-related requirements would not be hard to meet for small designs running in small FPGAs. Unfortunately, we knew from our C++ simulator that we needed at least 350 bits of register state in the Vortex datapath alone, not including datapath arithmetic logic. The amount of control logic was hard to deduce from the simulator, but it was clear that Mbus control logic and control logic devoted to fine-grain access control would be significant. High logic density is not the only problem; we need a minimum of 100 pins for the Mbus and SRAM interfaces. Not surprisingly, our density requirements coupled with the Mbus timing requirements is what makes implementing this design with FPGAs hard. Indeed, fast and dense FPGAs are the holy grail of the programmable logic industry.

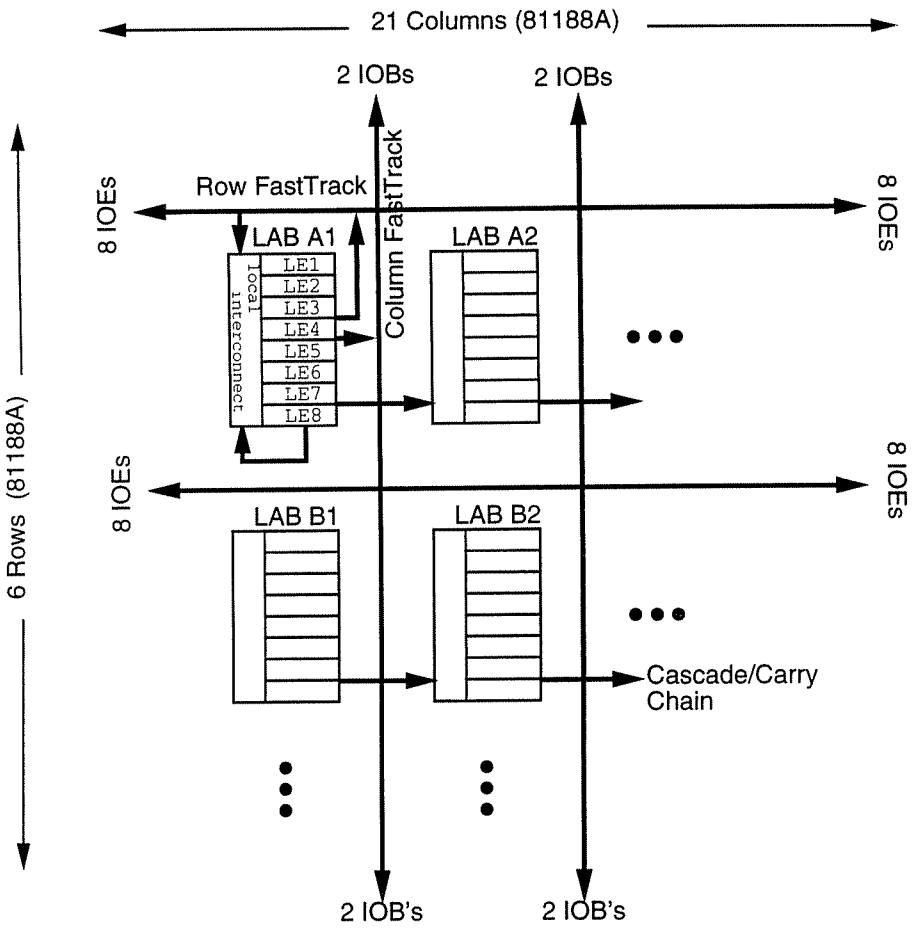


Figure 11: Altera FLEX8000 Architecture

7.2 FPGA Selection

Given the constraints outlined above, we were faced with the difficult task of choosing the appropriate part (or parts) to implement Vortex. We first considered Xilinx parts, which certainly are dense enough for our requirements. Xilinx's highly segmented SRAM-based routing structures, while very tolerant of logic changes after pinouts are fixed, are just not fast enough to support our system clock speed of 50 (or even 40)Mhz. Typical Xilinx 4000 series parts achieve in-system clock speeds in the 20-30MHz range, with very carefully designed small circuits sometimes reaching 50MHz. We next considered the Actel family of parts, the Act-2 and Act-3 lines of antifuse-based FPGAs. While dense enough and probably capable of reaching at least 40MHz, the clock to Q delay of the I/O cell flip-flops in these parts was prohibitively large, about 14nS! There is no feasible clocking strategy for running the Actel parts on a 50MHz Mbus that does not involve multiple clocks and clock resynchronization.

Finally, we looked at Altera's FLEX8000-series FPGAs [Alt95]. This family of parts is SRAM-based, and features high pin counts (up to 208 user I/O pins,) fast clock to Q times for I/O cell flip flops (near 1nS,) high density (up to 1500 flip-flops total,) and can support up to 10 independently-tristatable groups of bidirectional I/O pins. The AC and DC characteristics of the 8000-series I/O cells are compatible with the Mbus. The basic architecture of these FPGAs, as explained in the next section, is amenable to high clock rates. We decided that the Altera FLEX8000 series FPGAs were our best hope for implementing Vortex.

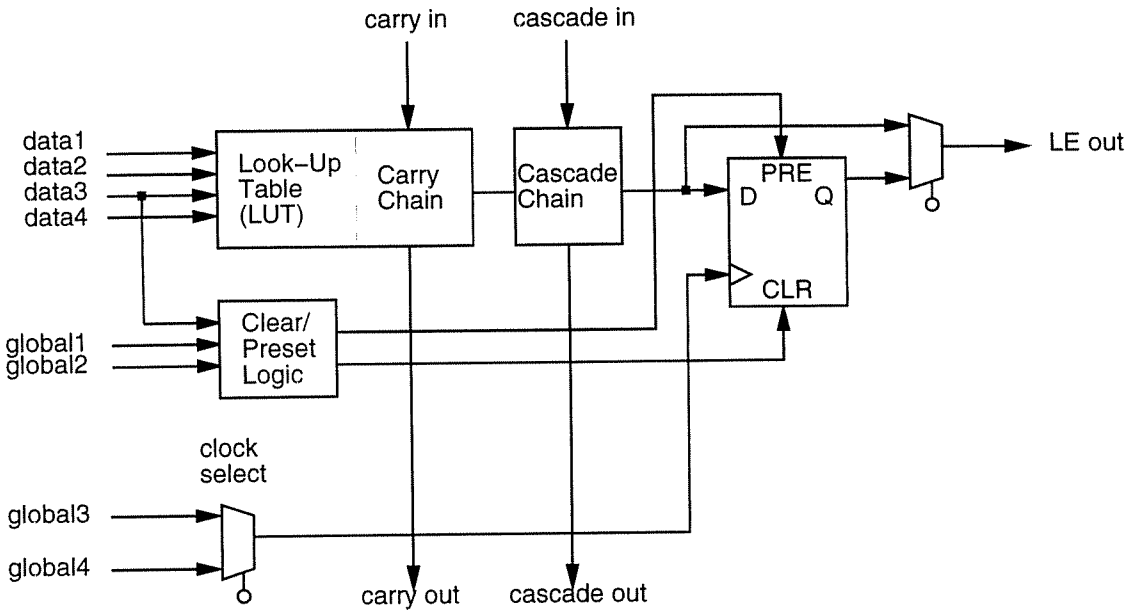


Figure 12: Altera FLEX8000 Logic Element

7.3 Altera FPGA Architecture

Altera FLEX8000 FPGAs, though featuring SRAM-based routing resources, differ from Xilinx FPGAs mainly in that the routing structure is hierarchic rather than segmented. The basic element of the FLEX8000 is the Logic Element (LE, Figure 12), consisting of a 4-input function generator, implemented as an SRAM lookup table, which can optionally feed a D-flip flop. LEs are grouped into Logic Array Blocks (LABs,) which contain 8 LEs and local routing for connecting LAB LE inputs and outputs. In addition to connecting the LE function generators, the LAB connects its LEs with *cascade* and *carry* chains, which aid in constructing fast, wide functions (such as decoders) and arithmetic functions (such as adders,) respectively. LABs are arranged into a rectangular array and connected by row and column routes, dubbed FastTracks^{TM} by Altera. These wires span the entire length and width of the chip and are not segmented. Row and column FastTracks are connected to groups of row and column I/O elements (IOEs) containing a bypassable flip-flop which can be either an input or output register, and a tristate buffer for bidirectional I/O capability (Figure 13). In addition to these I/Os, there are four dedicated input pins, which are routed to each LE and I/O cell in the device; these are used to distribute clocks, resets and other high fan-out input signals. Finally, LABs on the same row have their carry and cascade chains connected by dedicated routing channels. See Figure 11 for a graphical depiction of the FLEX8000 architecture.

The combination of LAB routing and FastTrack routing is what gives the FLEX8000 parts the ability to run at high clock rates across designs – the routing delay between any two LAB LEs is independent of the design complexity, and delay between any two non-LAB LEs is much less variable than that of comparable Xilinx designs. Once a signal is on a FastTrack, it is accessible to every LE along the track at a constant delay, assuming the FastTrack to LAB routing for that LE's LAB is not saturated such that no connection is possible. As this implies, the speed advantage of this routing scheme has been traded off for decreased routing flexibility; while more complex Xilinx designs might run more slowly, more complex FLEX8000 designs are sometimes unroutable. Clearly routing a signal all the way across the chip is sometimes not necessary and consumes a track that can be shared between signals. Since groups of adjacent I/O cells are connected only to particular FastTracks, preassigned pin constraints can affect routeability of the design, especially those designs with high logic and pin utilization [KR95]. Section 12.2.2 deals with our problems with pin assignments in this project.

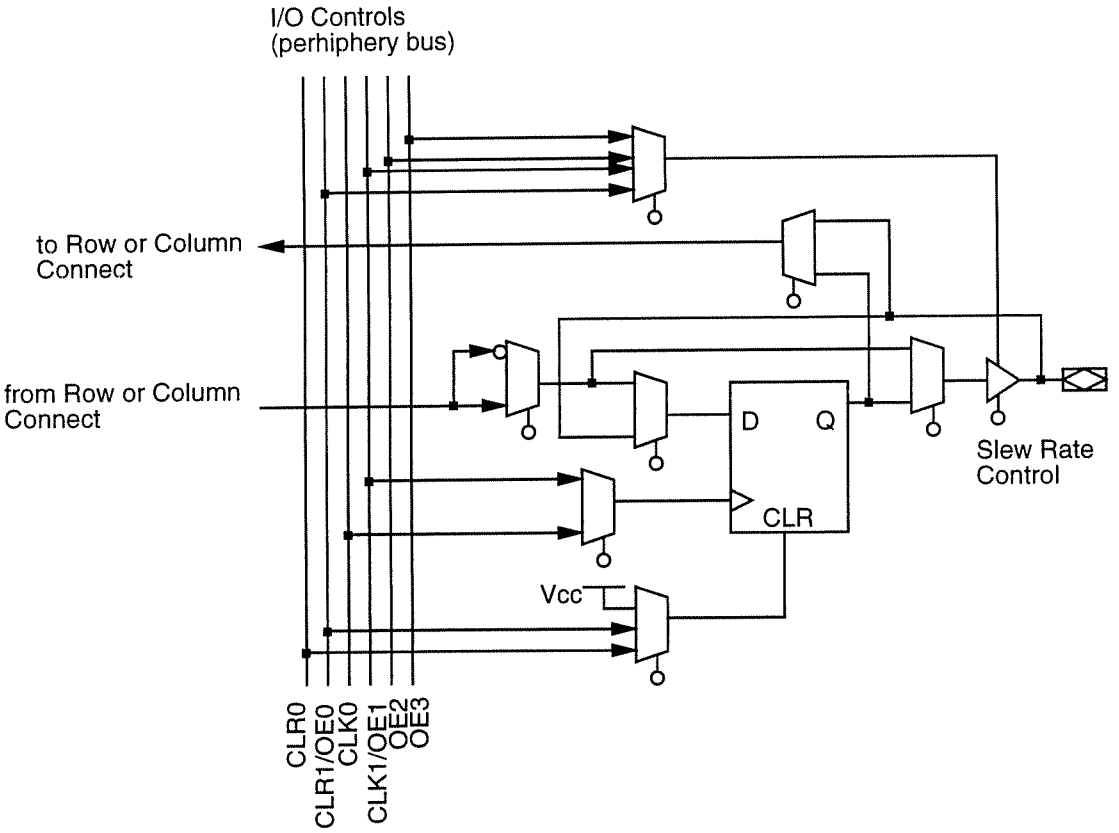


Figure 13: Altera FLEX8000 I/O Element

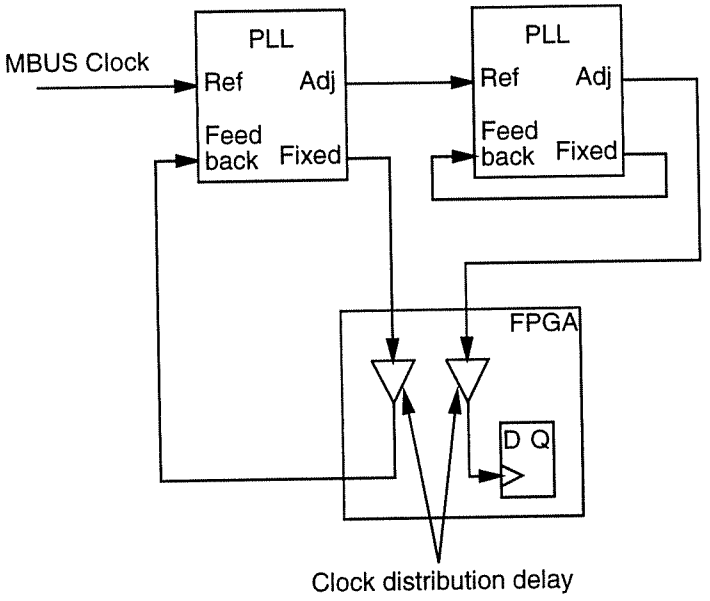


Figure 14: FPGA Clocking Scheme

7.4 The Altera EPF81188ARC240-2

The fastest version of the densest 8000 part, the EPF8115000-3 looked like it could probably support two levels of logic between flip-flops and run at 50MHz, but it would have been quite an effort to shoehorn our design into two-level logic. Fortunately, near the beginning of the project, Altera was making the transition from a 1.2μ two-level metal fabrication process to a 0.65μ three-level metal process. The densest near-term 8000A, as the new family is called, was to be the EPF81188A. The 81188A has a total of 1188 flip-flops (1008 LEs + 180 IOBs.) According to the data book, the 81188A would probably be able to much more easily reach 50MHz than the old 81188, having much better LAB, FastTrack and IOB timing characteristics. Using this part would almost certainly mean that we would need two of them based on our density requirements, however, since the 81188A part can only support four groups of independently-tristatable bidirectional pins, we would need two parts since our design calls for six groups of such signals.

Having addressed clock speed, clock to Q time and density, only drive strength and ground bounce remain to be considered. The I/O drivers on the 8000A family are PCI bus compliant; PCI electrical characteristics are compatible with Mbus. The only device level solutions to the ground bounce problem are to have a sufficient number of VDD and Ground pins on the device, and to minimize lead inductance. We cannot change the number of supply pins, but we can choose the package. Since surface mount Flat Pack technology uses extremely short leads, the leads have lower parasitic inductance than competing packaging technologies such as Pin Grid Array (PGA.) Though it is true that PGAs have shorter bonding wires and better internal power distribution than Flat Packs, the longer (and usually socketed) PGA leads have much higher inductance. Empirical evidence suggests that a PGA-packaged and socketed 81188A does suffer from ground bounce problems. Certainly given the number of I/O pins on this device (184), more than the 41 power and ground pins provided are needed to support the wide buses designers want to implement. Careful attention to board-level power distribution (see Section 10.4) is needed to minimize ground bounce even in Flat Pack packages.

We chose two Altera EPF81188ARC240-2, the 240 pin PQFP (Plastic Quad Flat Pack), -2 speed (fastest) version of the 81188A to implement all of the logic needed for Vortex.

7.5 Clocking Strategy

Even though the clock to Q time of the 81188A IOB flip-flops is extremely low (1nS), this does not in itself meet the Mbus clock to Q timing constraint. All logic devices have input pin delays, and since the

clock is distributed chip-wide there is a 5nS clock distribution delay. Therefore, though an IOB flip-flop drives its output 1nS after a clock edge, it does not see the clock edge for 5nS after it has appeared at the Mbus connector. The total clock to Q time is then 6nS, which violates the Mbus specification of 1.4nS. One solution to this problem is to generate a dedicated clock for the FPGA with clock edges coming early with respect to the Mbus system clock. We can make the clock edge exactly as early as it needs to be by using a PLL (Phase-Locked Loop) and including the FPGA clock routing delay in its feedback path.[3] The Mbus clock is driven into the PLL's reference clock pin. The PLL's output is driven into one of the FPGA's dedicated clock inputs, through the dedicated clock routing, off the FPGA and directly to the PLL's feedback input. The PLL output produces a clock which is exactly early enough so that the positive edges of the returned clock match those of the reference clock. All flip flops in the FPGA are clocked with the early version of the clock.

There is a potential problem with this solution – when the FPGAs are listening to Mbus control and data signals, we may artificially cause setup and hold violations at our receiving flip-flops since the clock is now early with respect to the bus data. Without performing a Spice simulation of the Sparcstation-20 Mbus, all of the motherboard Mbus citizens, the hyperSparc Mbus interfaces and the Altera parts, it is impossible to know exactly how much slack there is to play with. Since it was impossible to collect the characterization data needed to perform the Spice simulation, we decided to make the FPGA clocks adjustable so we could make them only as early as they needed to be and still meet Mbus timing requirements. To this end, the clock circuitry on the Vortex board is overengineered. Figure 14 schematically depicts our final clocking scheme. For the PLLs, we selected the TriQuint GA1088 [Tri94] clock buffer, which includes adjustable outputs. The early clock generated by the first GA1088 is not directly used to clock the FPGA flip-flops; instead another dedicated input is used for the actual clock. An adjustable version of the early clock is first fed to another GA1088, whose PLL is configured to lock to itself. The adjustable versions of this second clock are fed to the FPGAs. Each GA1088 can adjust its clock to be -5, -2.5, 0 or +2.5nS apart from its reference clock.

Having completed the prototype boards, we found that all of the clock adjustment circuitry was not needed; the system works fine when the clocks are adjusted to 0nS early. This is because the Mbus specification was designed prior to the first Mbus-based machine, the Sparcstation-10, and as such is conservative. Given the improved motherboard of the Sparcstation-20, the newer versions of the built-in Mbus citizens and the newness of the Ross hyperSparc module, the system timing requirements are greatly relaxed compared with those given in the Mbus specification.

8 Design Methodology

Our design methodology is directed at rapid prototyping. To this end, we developed a C++ simulation of a cluster of Sparcstation-20 machines populated with Vortex boards to determine the feasibility of the project. When we had accomplished this, we moved on to implementation in Verilog, a hardware description language. While synthesizing from an HDL has performance drawbacks, the ease with which changes to the design can be made to an HDL description is important for a research project. In the end, making changes to the full Vortex design proved very difficult due to the FPGA densities; see Section 12.2.2 for details on re-routability of modified Vortex designs.

8.1 Logic

8.1.1 C++ Simulation

As outlined in Section 2, we first developed a C++ simulation of a Sparcstation-20 Mbus. The Mbus is not modeled on a cycle-by-cycle basis, but at the Mbus transaction level. The simulator was later extended to keep track of the number of bus cycles consumed by each transaction, and to permit simulation of the whole Typhoon-Zero system (up to 40 nodes.) The simulator allowed us to gather system performance data with a variety of hypothetical network hardware and latencies, as well as determining the feasibility of the architectural ideas presented in Section 5. Finally, it allowed us to get an idea of how much state Vortex would need to implement fine-grain access control, and gave some idea of the complexity of

[3]This idea is due to Andreas Nowatzyk of Sun Microsystems

the control logic. The C++ code which simulates the Vortex module comprises the original functional specification for Vortex.

8.1.2 Verilog

In order to complete this project as quickly as possible, we decided to use a hardware description language (HDL) to capture the design rather than using schematics. We felt that the benefits of rapid design entry and ease of making changes to a register-transfer level HDL description far outweighed the main drawbacks of synthesizing the HDL to gates, namely an increase in total gate count and a loss of fine control over synthesized structures.

We chose Verilog as our HDL since Sun uses Verilog.

8.1.3 Synthesis

We used Synopsys versions 3.1a through 3.3a to synthesize the Vortex logic from Verilog. Synopsys is a widely used synthesis package that is best suited for standard cell and custom designs; it is not well-suited to dealing with the discrete routing delays of FPGAs. We encountered problems related to this tool bias which are further described in Section 12.2.

8.1.4 FPGA Place and Route

To place and route the synthesized logic, we used Altera's Max+Plus II tool, versions 5.1 through 5.3; Max+Plus II is the only software available that handles the FLEX8000 family. We encountered additional problems with Max+Plus II, which were related to loss of design structural information at the interface with Synopsys. Version 5.2 represented a major improvement in the place and route algorithms used, and has made possible more post-PCB layout logic design changes than we originally thought possible; see Section 12.2.

Max+Plus II includes static timing analysis (for one FPGA at a time only), and is capable of writing the placed and routed design as gate level Verilog with timing information, which was very useful for timing verification; see Section 11.2.2.

8.2 Printed Circuit Board

The PCB connectivity was captured as a schematic using Cadences Composer tool. The schematic was not automatically verified against the top-level Verilog model due to naming and netlist translation problems, but it was manually checked several times by two people.

The printed circuit board schematics are presented in Appendix B.

9 Implementation

As described in Section 5, Vortex must snoop coherent transactions, map the tag SRAM and control registers, and become bus master to issue coherent transactions. Vortex is partitioned into six logical entities: the slave, (for snooping and tag/register mapping) the master, (for tag downgrades and CCR invalidations) the datapath, (containing the control and fault status registers) the tag address unit, the Mbus interface registers, and Netboy (the Myrinet "network interrupt" pin interface.)

This section describes the implementation of each Vortex component, then describes how these components are partitioned across the two Altera FPGAs. Lastly, timing issues related to the partitioning and intra-FPGA routing are considered. All implementation was done using Verilog [TM91], a hardware description language.

9.1 Slave

The slave handles all incoming Mbus requests to address spaces that Vortex services. As shown in Figure 15, it consists of decoding logic, an interface to the master logic, an FSM and output logic. The FSM is responsible for implementing fine-grain access control (snooping), providing Mbus support for access

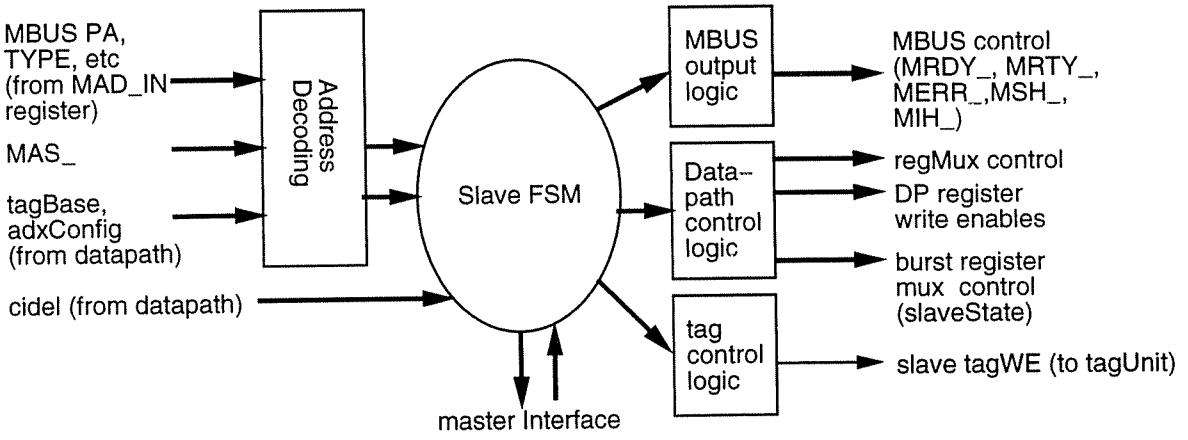


Figure 15: Slave Block Diagram

control (tag shadow spaces), mapping the control registers onto the Mbus, acknowledging CI transactions, and general datapath and Mbus control related to these services. It shares responsibility with the master for the tag SRAM control interface.

9.1.1 Slave Address Decoding

The slave must partition the address space into several regions, as described in Sections 6.1 through 6.4. The slave address decoder examines the current physical address, which is sampled from the bus during each Mbus address phase, and decodes this PA into a one-hot encoding indicating what, if any, address space handled by Vortex has been touched. In some cases the comparison is against a hard-wired constant (Tempest snoop spaces), and in others, against register values (tag shadow spaces.) Additionally, the TYPE and SIZE of the current Mbus transaction are decoded into one-hot encodings for further use.

The slave state machine makes its first level and some of its second level branch decisions based on the outputs of the PA and TYPE decoders, as well as many downstream decisions based on the SIZE, TYPE and data captured from the Mbus data phase.

Originally, the address decoding was implemented using the high-level Verilog '==' equality operator. It soon became clear that this design could never meet timing; the address decoding was not fast enough to happen in a single 20nS cycle. Close examination of the synthesized equality circuits showed that Synopsys was forcing the decoding into too many levels of LEs; it appeared that when the physical address register was compared against another register (instead of against a constant value) it would only utilize two of the four LE inputs in the first-level LEs. To overcome this problem, and to make the decoding as fast as possible, all address-related decoding is done by manually instantiating 4-input XNOR functions and cascade chains (see Section 7.3.) Synopsys was able to map this representation of the decoder into the minimum-depth comparator tree possible, which can decode the address in less than the allotted 20nS.

Some decoded signals, though they evaluate in less than one clock cycle, are delayed one and sometimes even two cycles by registers when they are not needed by the slave FSM in the clock cycle immediately following validity of the data they are decoding. This was done to make the slave logic more routeable; such signals represent false timing goals to the place and route software. By breaking these multicycle paths into several single-cycle paths, the Altera placement algorithms have been 'fooled' into spending effort on the true critical paths in the design. Even if the Altera tools could identify multicycle timing paths, this technique would still be useful in reducing routing pressure – it actually provides a kind of segmented routing structure. The disadvantage of this is of course that extra registers and routes are being used which do not contribute directly to the function of the design.

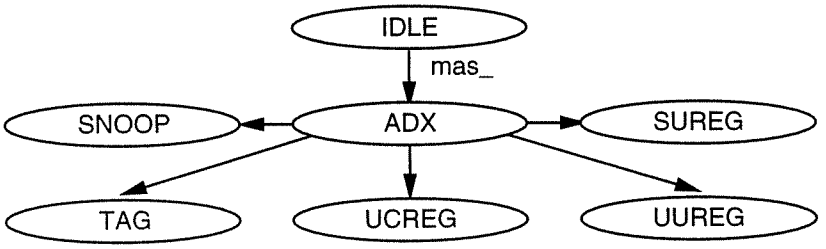


Figure 16: Top Level Slave FSM State Transition Diagram

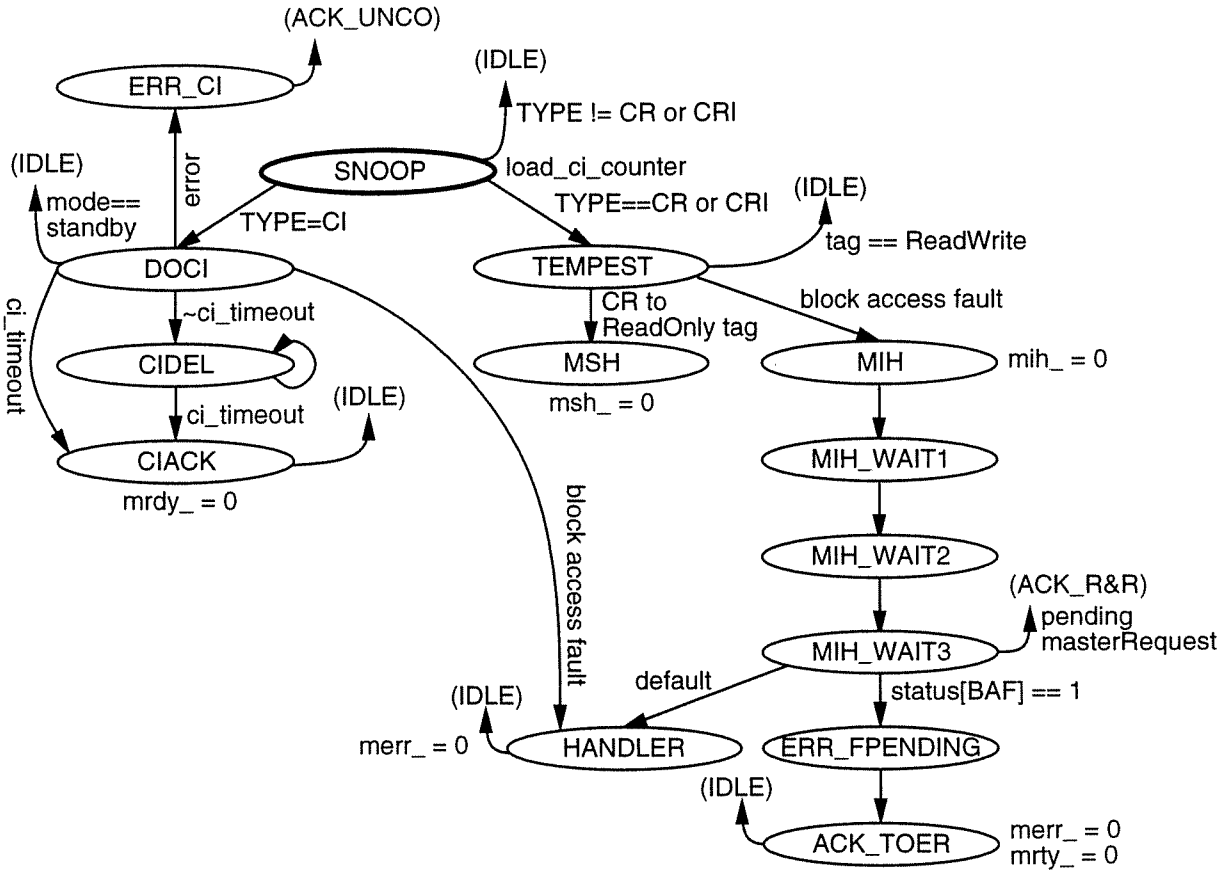


Figure 17: Slave FSM – SNOOP & Related States

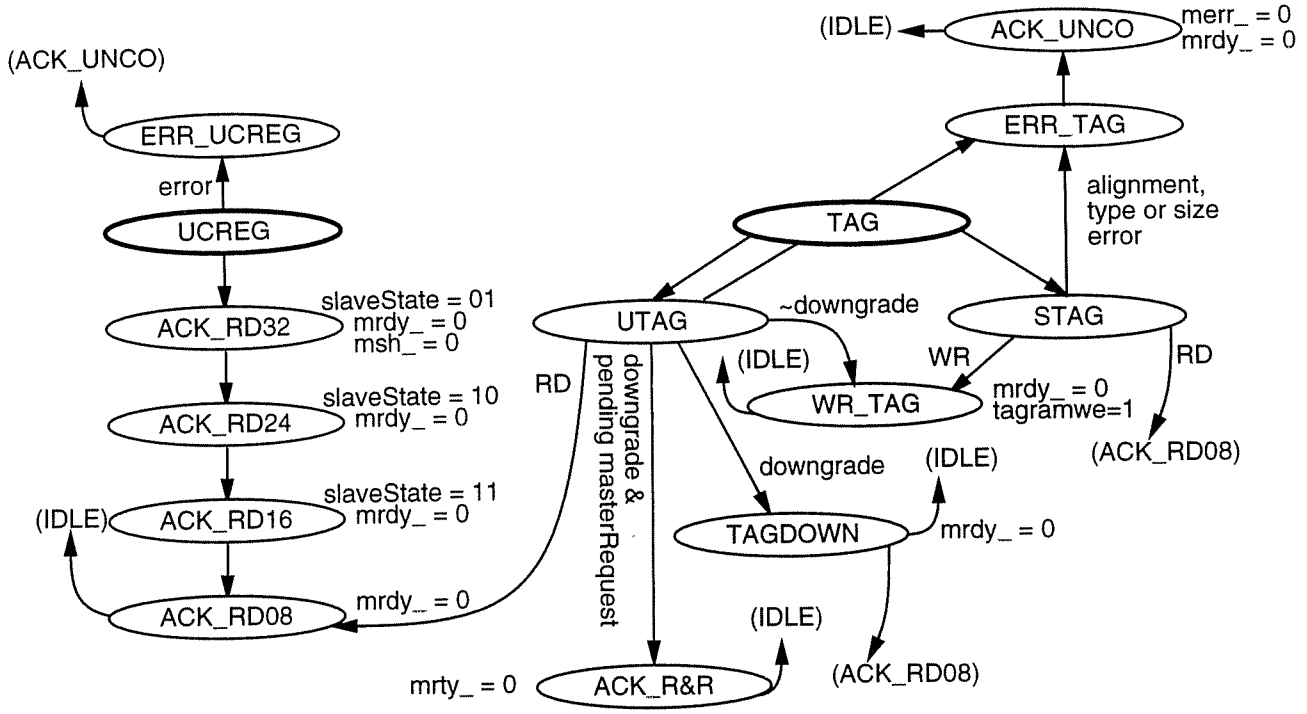


Figure 18: Slave FSM – UCREG, TAG & Related States

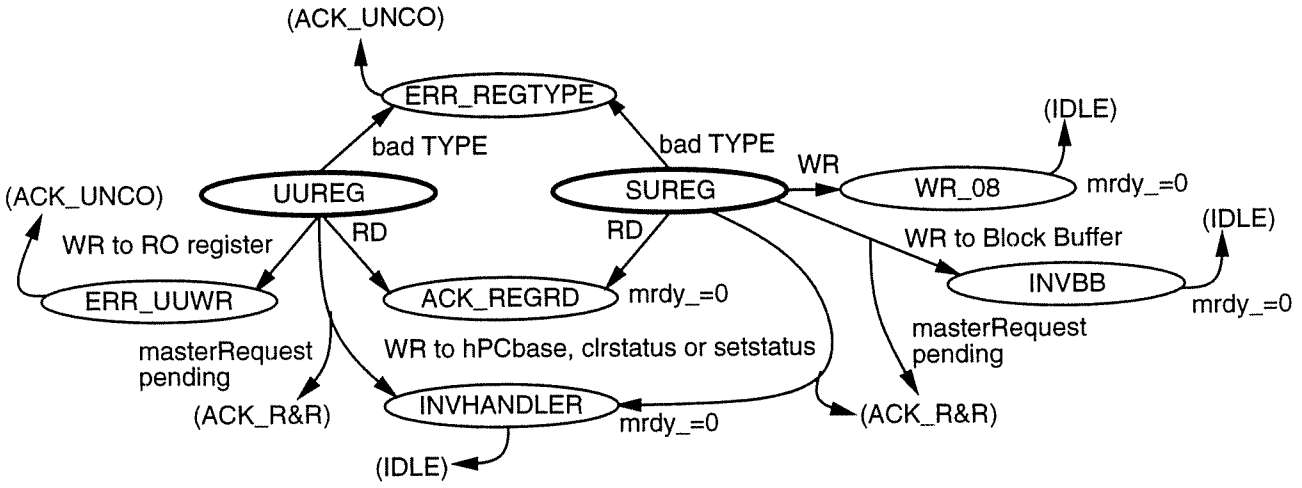


Figure 19: Slave FSM – UUREG, SUREG & Related States

9.1.2 Slave FSM

The slave FSM is implemented using a one-hot state assignment. While this is the least dense encoding for a state machine, it leads to a particularly fast implementation – the next state function for a state only includes those bits representing states from which it is reachable, rather than the entire current state vector as in a binary encoding. Thus the fan-out and fan-in requirements of state register outputs are kept to the absolute minimum in exchange for a longer state vector. Since FPGAs have plenty of registers but limited routing resources, the bit-per-state encoding is optimal.

The slave state machine has 40 states, 15 inputs and 34 outputs. Figures 16 through 19 depict the slave state transition diagram.

The slave FSM leaves the IDLE state when triggered by the Mbus address strobe signal, MAS_. During the next state (ADX), the address decoders generate their outputs as described above. The state machine then branches off to one of the three register spaces, the tag spaces, or to the next level of decoding. From the TAG state, the machine branches to either UTAG (user tag) or STAG (system tag) handling; from the SNOOP state, the machine returns to idle immediately if the transaction is not CR, CI or CRI, or the PA does not fall within the Tempest snoop space, or the transaction was generated by Vortex's master state machine; see Section 9.2.1 If the transaction is a CI, the machine moves into DOCI to evaluate how it should handle the CI, otherwise the machine moves to the TEMPEST state to enforce fine-grain access control.

At this point, if the machine has reached the DOCI state, it handles the three possible CI behaviors based on the **mode** register. The possibility that the CI was issued to an illegal address range is handled (an Mbus Uncorrectable bus error is returned), then the CI is acknowledged if it fell outside of Tempest memory or the tag was ReadWrite in Tempest space. Otherwise the CI is acknowledged with the Mbus BusError, and the fault tag, address and operation are recorded. If there happened to be a pending block access fault, (Status[BAF] == 1, see Section 6.4.4) the fault information is not recorded and the CI is acknowledged with the Mbus Timeout error.

A similar chain of events occurs when the FSM reaches the TEMPEST state; the transaction TYPE can only be CR or CRI (read or write miss) in this state, so the tag is checked, and if the transaction is a CR to a ReadOnly block, the FSM simply asserts MSH_. If the transaction is otherwise illegal first MIH_ is asserted for one cycle, then Vortex pauses for three cycles and finally drives the Mbus BusError acknowledgement onto the bus while capturing the fault information subject to the same exception as described above.

When the address falls within User Tag space, all writes must be checked against the tag already in SRAM to determine if tag/cache state coherence action must be taken as described in Section 5.1.4. When the state machine enters the UTAG state, the tags are compared, and if the write represents a downgrade, the block address and new tag are captured in the datapath and the master is asked to perform the tagDowngrade operation (Section 9.1.3) if there is no pending master request. The write is given the the Relinquish and Retry acknowledgment; the master is responsible for updating the tag such that when the tag write comes back, it is no longer a downgrade and is acknowledged normally.

The rest of the state machine is devoted to servicing the register spaces. In the case of UCREG, control signals are provided to the datapath to sequence the wrapped return of the CCRs; they are generated as early as possible (beginning in ADX) due to heavy pipelining in the datapath; see Sections 9.3.1 and 9.3.3 for details on this.

9.1.3 Slave to Master Interface

Since downgrading tags and writing particular registers require Vortex to become bus master, the slave must be able to invoke the master on its behalf. The master accepts one command from the slave at a time through a register (masterRequest) and strobe interface; when the master finishes the command, it clears the register.

If masterRequest is not clear when the slave wants to queue a master request, the slave will respond to the transaction requiring the master transaction with the Mbus Relinquish and Retry acknowledgment. This causes the processor that issued the transaction to relinquish the bus; Vortex can then acquire the bus and issue the transaction(s) for the pending master request. Eventually the suspended transaction

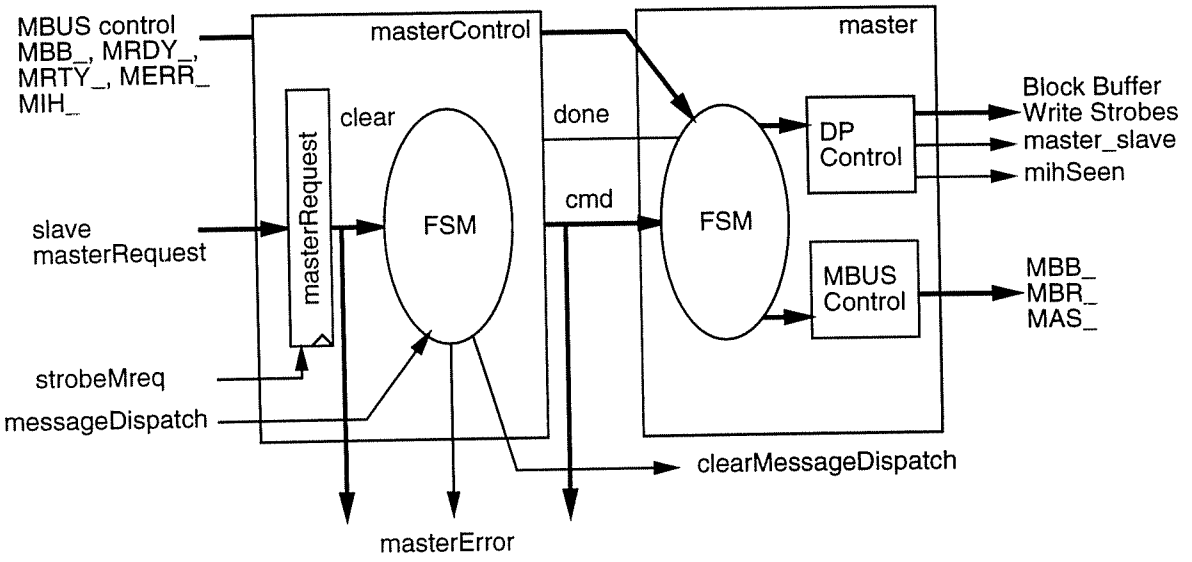


Figure 20: Master Block Diagram

is retried, at which time the slave can queue the master request, so this time the transaction will be acknowledged normally, and the master will complete the request.

The slave can request one of three master operations: **invHandler** (issue a CI to the handlerPC CCR), **invBB** (issue a CI to the Block Buffer CCR), or **tagDowngrade** (issue a CRI to a downgraded block, saving the data in the Block Buffer while writing the new tag into the SRAM, then issue a CI to the Block Buffer)

9.1.4 Slave Output Logic

The slave output logic is used to control the datapath and to return acknowledgments on the Mbus. All control signals are derived from the current state vector; most are registered. Signals that directly control the Mbus must be registered due to Mbus timing constraints, while other control signals would cause timing violations if they were not registered before being distributed to the datapath and other state machines. Adding registers in this manner increases the latency of the control signal, but this technique is absolutely necessary to make sure the design works. See Section 9.8 for details on registers inserted at interface boundaries for timing purposes.

9.2 Master

The master accepts commands from the slave as described in Section 9.1.3 and produces transactions on the Mbus in response to the commands. It also accepts a single command from Netboy (Section 9.6) which is the equivalent of the **invHandler** described above.

The master logic is broken into two parts; one handles high-level requests from the slave and network interface, and one handles the low-level interactions with the Mbus arbitration logic, and is responsible for controlling the datapath during master operation. A block diagram of the master logic is depicted in Figure 20.

9.2.1 masterControl

The masterControl state machine, along with the masterRequest register, implement the interface to the slave state machine and the Netboy state machine. Though this interface consists of a command request register only, more information besides the request type is needed by the master – an address and sometimes a tag value.

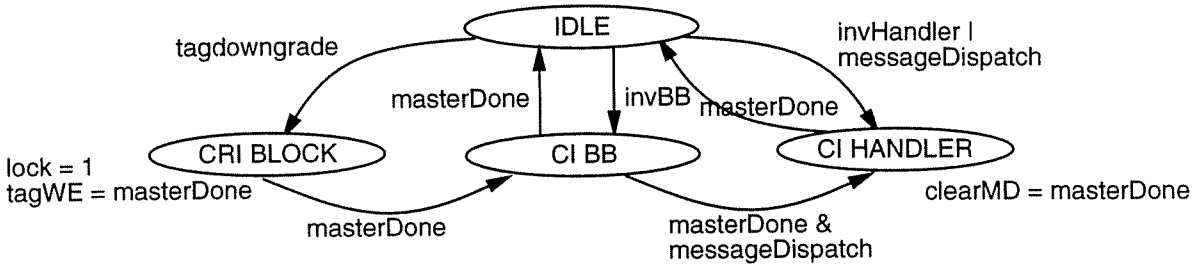


Figure 21: masterControl FSM State Transition Diagram

Two master transactions do not require any explicit address information from the requester; the address is implicit when performing a invHandler or invBB request, since the two CCRs have fixed addresses. tagDowngrade requests require the address of the block, and the new tag, which are captured in the datapath under control of the slave during the initial tag downgrade write. See section 9.3.6 for details on master address generation.

When the masterControl state machine is handling a tagDowngrade request, it drives the new tag into the SRAM and strobes the write enable during the block retrieval CRI, without explicitly generating a tag address. This works because the correct address of the tag is already being presented to the SRAM by the tag address generator as a side effect of the slave snooping the master's CRI. See Section 9.4 for details.

The masterControl state machine is triggered by the presense of a command in the masterRequest register, or by the presense of the messageDispatch signal generated by Netboy (Section 9.6.) masterControl arrives in one of three states (CIHANDLER, CR_IBLOCK or CIBB) depending on the request; the outputs from these states configure the master address multiplexor in the datapath and invoke the master FSM. masterControl waits in the first level state until the master indicates that it is done. The CR_IBLOCK state is followed by CIBB to accomplish the tagDowngrade operation. During CR_IBLOCK, masterControl asserts the *lock* signal, indicating to the Mbus master that it should not relinquish the bus after finishing the transaction. This is done so that the block retrieval and Block Buffer invalidation are performed atomically. When CIHANDLER and CIBB return to the IDLE state, the master FSM *done* signal is used to clear the masterRequest register and the Netboy messageDispatch signal.

masterControl is implemented using a one-hot state assignment. Figure 21 depicts the masterControl state machine.

9.2.2 master

The master state machine fields requests from masterControl and translates them into the appropriate sequence of events on the Mbus. When triggered by masterControl, it attempts to acquire the bus by asserting the Mbus request signal, MBR_. When the Mbus arbiter acknowledges by asserting MBG_, the master drives MBB_ (Mbus busy signal) to grab the bus, then causes the datapath's madMux (see section 9.3) to drive the appropriate address onto the bus and waits one cycle to account for the madMux latency. It then asserts MAS_, and waits for a response. When performing a CI, it waits for a single acknowledgment, which may actually come from Vortex's own slave logic. If the master is doing a CRI, it watches MIH_, and if it is asserted, waits three cycles before listening for more data acknowledgments, in accordance with the Mbus specification. As it receives acknowledgements, it produces write strobes for the Block Buffer. See Figure 22 for the master Mbus state machine. Note that although the RETRY, R&R and SD_ERR states are shown leading from RD_08, they are reachable from any of the RD_* states. Also note that if the *lock* signal was generated by masterControl, the master branches to the LK_OK state rather than the SD_OK state, and returns to ADXWAIT rather than IDLE. This guarantees that Vortex does not release the Mbus after it finishes the locked transaction.

If an Mbus error acknowledgement is encountered by the master FSM, it will arrive in the SD_ERR state, where it strobes the value stored in masterRequest and the current master operation into the **error** register, if there is no pending error.

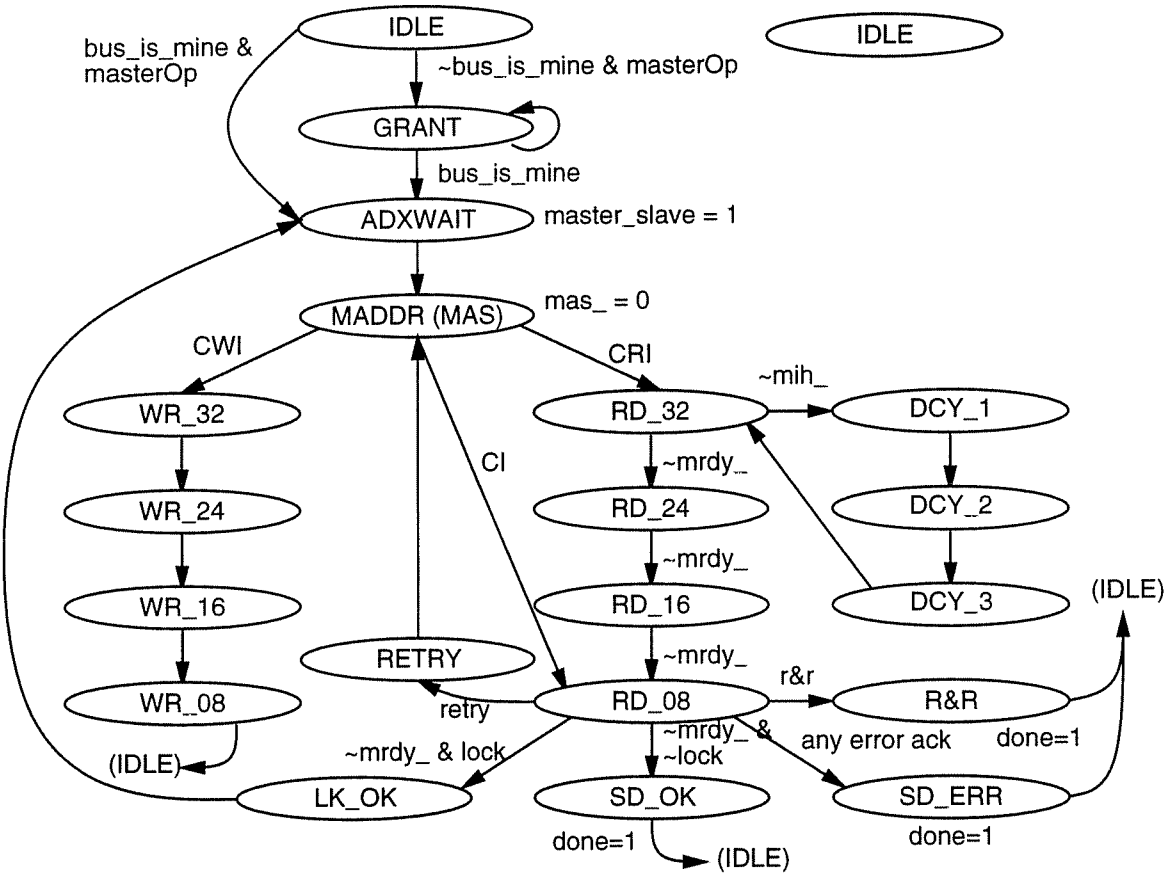


Figure 22: Mbus Master FSM State Transition Diagram

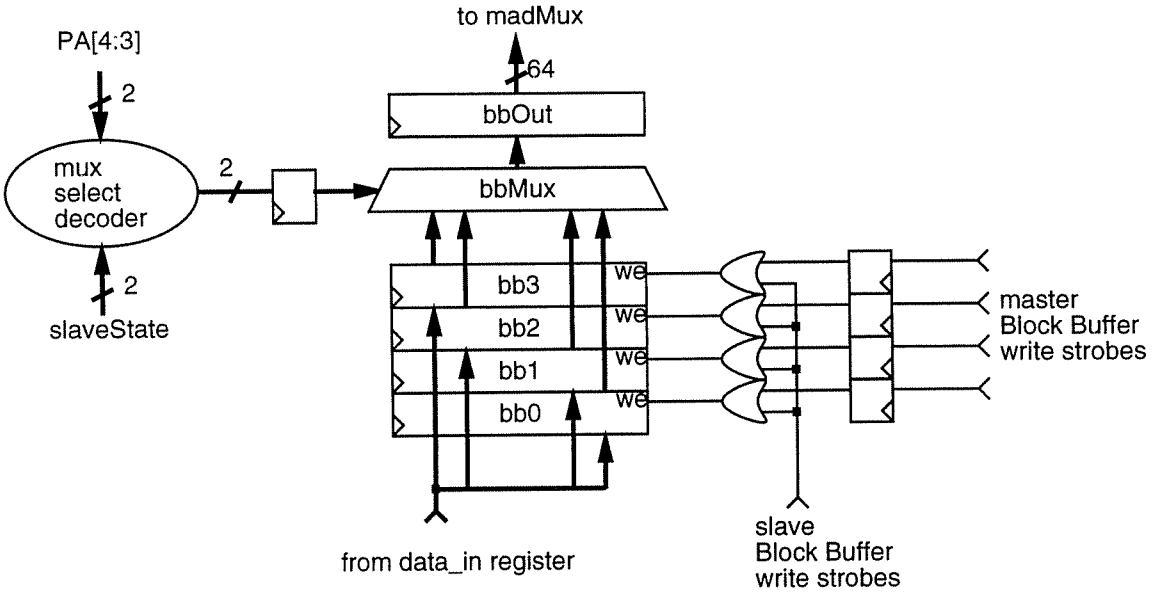


Figure 23: Block Buffer Block Diagram

Like the slave and masterControl, master uses a one-hot state assignment for maximum speed and routability.

9.3 Datapath

The datapath consists of registers used to capture the state of the current Tempest block access fault, the current block downgrade state, and control registers. The rest of the datpath logic is devoted to multiplexing the appropriate registers onto the Mbus at the proper time, as orchestrated by the master and slave datapath control. A substantial number of registers are consumed in the datapath for timing purposes; all multiplexor outputs are registered, leading to high slave read latencies.

9.3.1 Block Buffer

The Block Buffer is a cache-block sized register, organized into four 64-bit (doubleWord) sized registers, with multiplexing and write enable logic as shown in Figure 23.

Since the Mbus supports wrapping on burst transactions, (see Section 4.3) the Block Buffer mux (bbMux) is controlled by bits 4 and 3 of the PA for the current transaction, as well as the slaveState bits produced by the slave FSM as it travels down the states handling the UCREG register space. The two mux control signals fan out to 128 LEs each, so the delay from the output of the decoder to the inputs of the bbOut register are quite substantial. This delay plus the delay back through the decoder to the PA and slave FSM state registers is guaranteed to exceed the clock cycle time of 20nS, so the output of the decoder is registered to break this long timing path.

The master write enable signals are registered to synchronize the write strobes coming from the master FSM with the data becoming valid in the data_in register during master CRI transactions.

Finally, the output of the bbMux is registered; the bbMux data still has to be multiplexed with other data before reaching the register that directly drives the Mbus MAD lines, the mad_out register. Without the bbMux output register the delay from the output of bbMux to the MAD register would exceed the clock cycle time.

9.3.2 Block Downgrade Registers

The Block Downgrade registers (Figure 24) are not visible to the user in Vortex control space. When a tag is downgraded in User Tag space, Vortex must retrieve and invalidate the corresponding block as

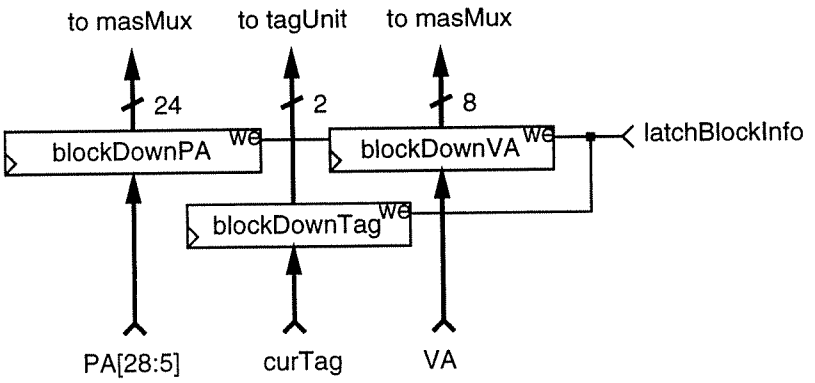


Figure 24: Block Downgrade Registers Block Diagram

described in Section 5.1.4. In order to accomplish this, the block physical address is translated from the tag address by saving PA bits 28 to 5 in the blockDownPA register. Because the hyperSparc caches are virtually indexed, we capture the tag's virtual address from the 8-bit Mbus VA field as well; together with the block PA this address is used during the address cycle when the downgraded block is retrieved. Finally, the new tag value is saved since the master writes the tag into the SRAM during the retrieval.

9.3.3 Fault Status Registers (handlerPC)

The fault status registers comprise the data returned in the handlerPC CCR. When a block access fault is detected by the slave state machine, the fault physical address, type, and tag are latched into registers, and the BAF bit (bit 0) of the Status register is set. These fault specific registers are combined with the handler PC base register by a multiplexor which connects to the main bus multiplexor as shown in Figure 25. Because the handlerPC is a Burst32-sized entity in UCREG space, this multiplexor supports wrapping as also implemented in the Block Buffer, described above. The handlerPC mux drives a 64-bit register which then drives the madMux described below; this is to break the control to data signal paths since the hPCMux itself is very slow. As with the Block Buffer, high fan-out control signals account for most of the latency through the hPCMux.

Since the handlerPC component registers are writable in the Uncachable register spaces, they can be sourced both from the current block fault information and the data_in register, which contains the new data during register writes. In Figure 25, inputs on the left of the smaller multiplexors are used for the fault data and are selected during a block fault, while the inputs on the right are used when the handlerPC registers are written. The status register can also be set and cleared via a bitmask register (see Section 6.4.4, status register), so in that case the status register is set based on the bitmask and its current value; this is the purpose of the AND and OR gates driving the status multiplexor.

9.3.4 Fault Tag Comparator

Tempest specifies no synchronization between user data references and protocol tag changes. Therefore the possibility that a block which has been faulted on may have its tag changed after the fault has been recorded by the FSRs, but before the fault handler has been invoked. This is possible when the protocol processor is handling a message while a block fault occurs; the message handler can change the tag of the faulted block in the tag SRAM. When the protocol processor finally reads the new handlerPC block, the fault tag value is stale.

To maintain tag SRAM/saved fault tag coherence, we include a comparator in the datapath which snoops all tag writes; if the tag PA corresponds to the current faulted block PA, the saved fault tag is updated with the written value. This update takes place immediately and is not deferred even when the update represents a tag downgrade, so the SRAM tag and fault tag are temporarily inconsistent until the master logic does the block retrieval associated with the tag downgrade.

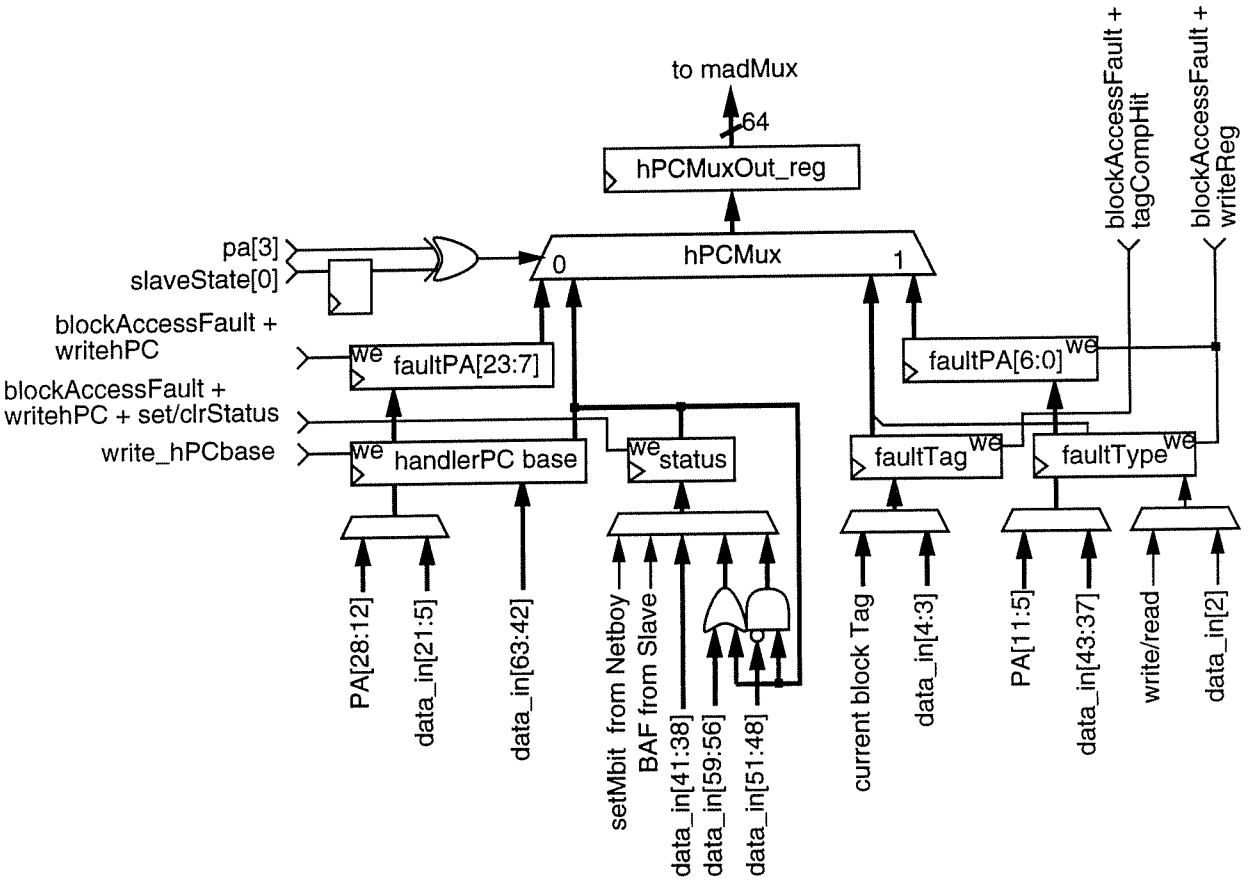


Figure 25: Fault Status Registers and Multiplexor

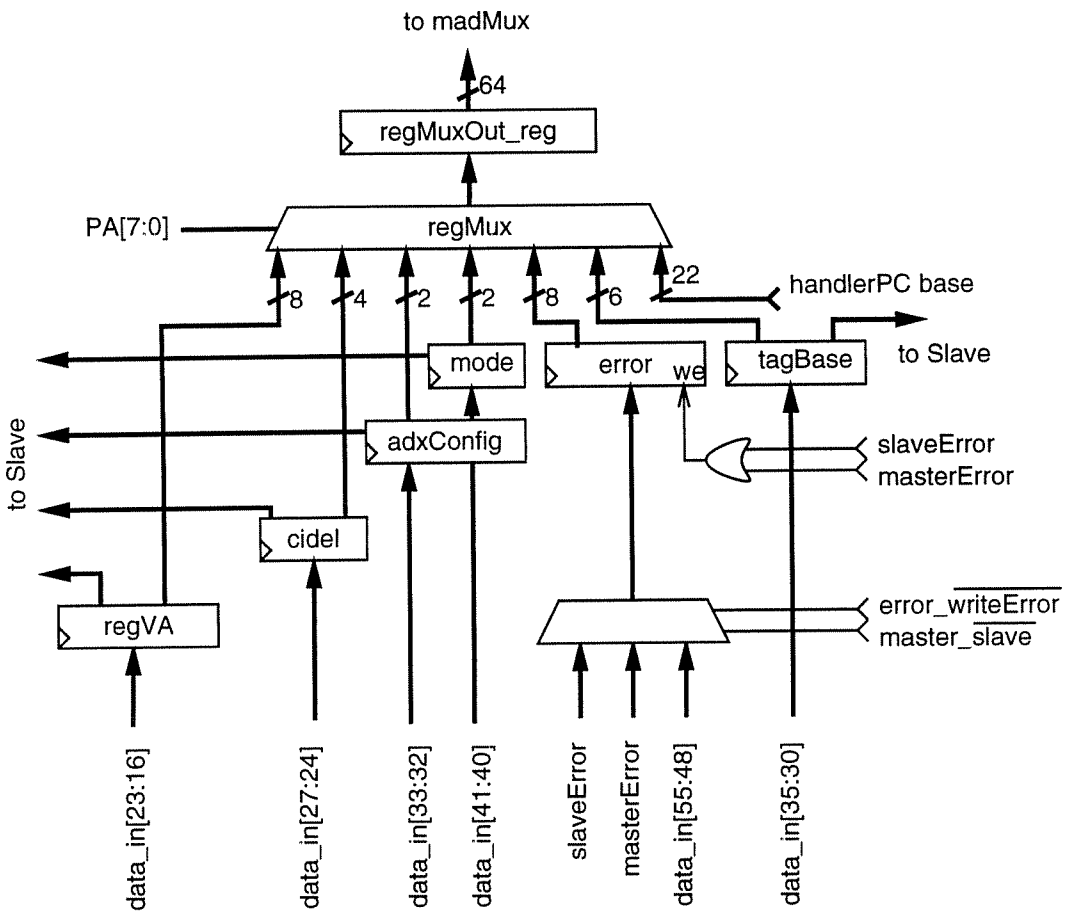


Figure 26: Control Registers and Multiplexor

9.3.5 Control Registers

Figure 26 shows the Vortex control registers and their multiplexor, the regMux. regMux is responsible for selecting and returning the proper register based on the current PA in the mad_in (see Section 9.5) register when there is a read to either uncacheable register space.

Note that the write enables for tagBase, mode, cidel, regVA, and adxConfig are not shown in the figure; each of these signals is generated by the slave FSM when a write occurs to one of the control registers. The handlerPC base register appears in 25, but since it is accessible both through the handlerPC CCR and the uncacheable user spaces, the regMux must provide a port for it.

9.3.6 Master (Address) Multiplexor

The master multiplexor (shown in Figure 27) is responsible for generating the Mbus address data needed during the master's MAS_ cycle. Based on control signals generated by the masterControl state machine, the master mux either generates one of the two CCR addresses using the regVA register for the VA field, a hardcoded address for the PA (using the Vortex's MID bits from the Mbus connector), and TYPE = CI, or it generates the address for block downgrades using the Block Downgrade Registers as described in Section 9.3.2 to populate the PA and VA fields; TYPE always equals CRI for block downgrades as described in Section 5.1.5. As with all the other multiplexors, the master mux has registered outputs to give the multiplexor a whole cycle to produce its output.

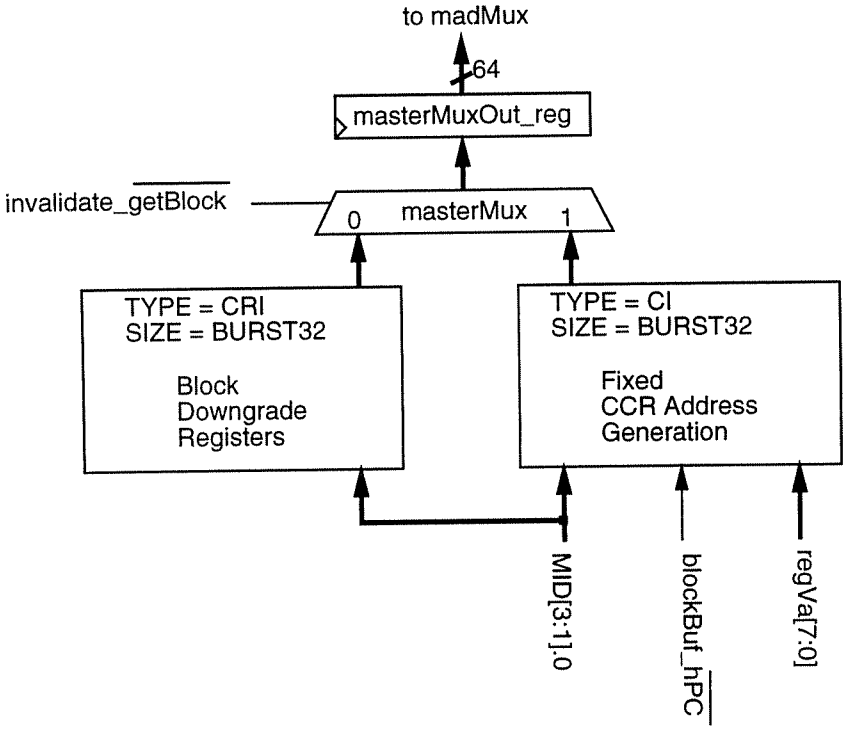


Figure 27: Master Multiplexor

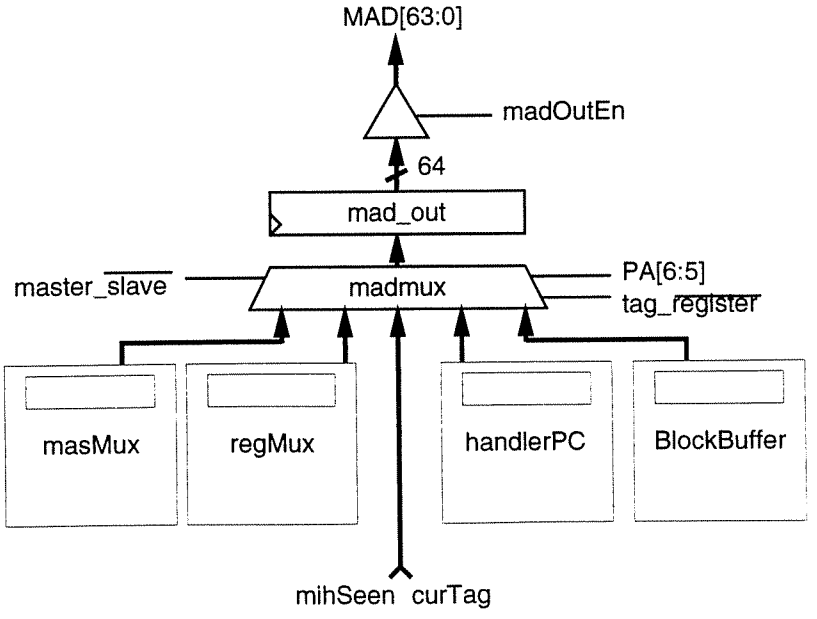
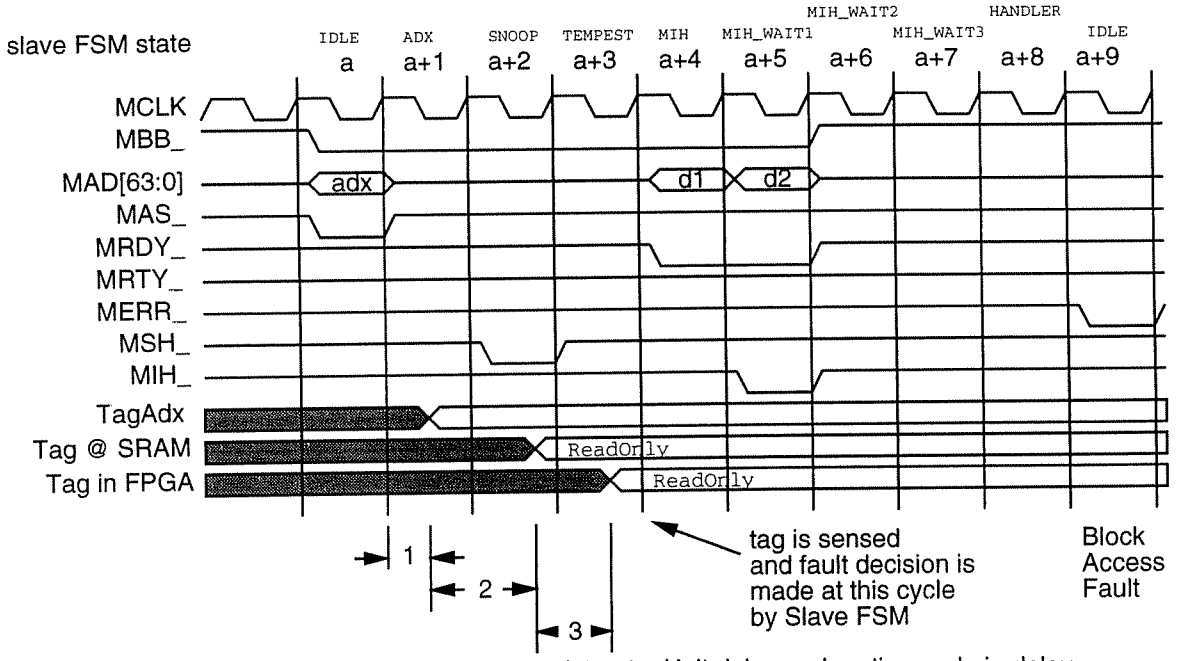


Figure 28: MAD Multiplexor



1. Clock to Q time of adx_in register, tagUnit delay and routing and pin delay
2. Tag SRAM data access time (25nS)
3. pin and routing delay back into FPGA

Bus signals are delayed by synchronization registers by one cycle from their corresponding Slave FSM cycles.

Figure 29: Tempest Snoop Timing Diagram - CRI (Write) to ReadOnly Block

9.3.7 MAD Multiplexor

The MAD multiplexor is responsible for joining together the outputs of all the 'first level' muxes: the Block Buffer mux, the handlerPC mux, the control register mux, and the master mux, as well as the current tag value during reads from the tag shadow spaces. This multiplexor drives the mad_out register which is directly connected to the Mbus's MAD pins through a tristate buffer as shown in Figure 28.

Although the MAD multiplexor is logically one entity, it is duplicated in both FPGAs; see Section 9.7 for details.

9.4 Tag Address Unit (tagUnit)

As described in Sections 6.1 through 6.3, the Mbus physical memory is not contiguous unless the machine is populated with 64MB SIMMS. Therefore, it is necessary to massage the bus physical addresses in order to arrive at a tag address; we must map away the holes in the four supported SIMM configurations to arrive at a contiguous 22-bit (4MB) tag SRAM address.

The tag address unit simply strips off the low five and high seven bits of the 36-bit Mbus PA, then discards some 2-bit set of bits 24, 25, 26 and 27 to form the three most significant bits of the tag address depending on the current value in the adxConfig register. The address mapping logic is sourced by the mad_in register, which is updated with the whole Mbus PA during every Mbus MAS_ cycle. This register is continuously modified by the tag address unit and driven into the tag SRAM. This is why the master does not need to drive an address into the tag SRAM while writing downgraded tags; when the master puts the block address onto the bus to perform the block retrieval CRI, the tag address unit automatically maps this address to the correct SRAM address.

The Toshiba SRAMs we are using take 25nS to return valid data from a valid address; the PA that the tag address decoder sees is valid at the beginning of Mbus cycle a+1 (the cycle following the MAS_ cycle.) The address to tag valid round trip time (from the PA register being clocked to the tag valid

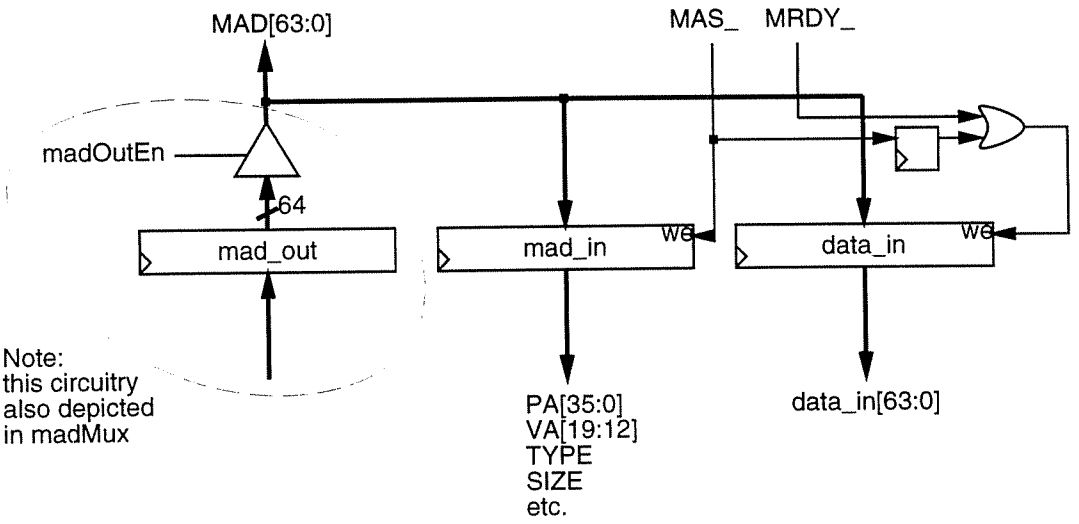


Figure 30: Mbus Interface Registers

at the input of an LE is 40nS (5.5nS output delay, 25nS SRAM delay and 9.5nS input delay.) There is no explicit interlocking for the validity of the tag; the slave FSM is simply required to wait until A+3 to evaluate the tag for the current transaction. This is the critical path for the fine-grain access control logic. A timing diagram depicting a snooped CRI to a ReadOnly block together with tag SRAM timing and Slave FSM state transitions is shown in Figure 29.

9.5 Mbus Interface Registers

The mad_in, mad_out, and data_in registers depicted in Figure 9.5 comprise the Mbus interface registers. mad_in samples the MAD pins on every Mbus address cycle, while data_in samples the MAD pins during the cycle following every address cycle. We can get away with this because we do not support burst write transactions, and the Mbus specification requires write data to immediately follow the address cycle during write transactions. data_in also samples the MAD bus when MRDY_ is asserted; this is to capture data returned as a result of the master's CRI during block downgrades, which is then latched into the Block Buffer. mad_out is driven onto the Mbus by the master or slave FSM when generating addresses or returning data on reads. mad_in contains the PA, VA, TYPE and SIZE of the current transaction and is used both by the slave, (for address decoding) tagUnit, (for tag address generation) and the datapath (for block downgrade and fault status information capture).

9.6 Netboy

Netboy is the network interface to the masterControl state machine and the Status[MSG] bit in the datapath Fault Status Registers, implemented as a small state machine (see Figure 31) When mode = Run and the myrinet message pin changes state, the Status[MSG] register is set via the setMbit signal, and masterControl is asked to perform an invHandler operation via the messageDispatch signal as described in Section 9.1.3. When the master finishes, masterControl strobes the clearMD pin to clear the messageDispatch register.

While Status[MSG] is set, Netboy does not track the state of the myrinet pin. If it toggles before the protocol code has cleared the Status[MSG] bit, the message is ignored and will not be dispatched, thus message queuing must be implemented in the Myrinet driver software.

9.7 Partitioning

As explained in Section 7.4, we use two Altera 81188A FPGAs to implement all of Vortex's logic. We must therefore somehow partition the design between the two devices. The fundamental reason for using

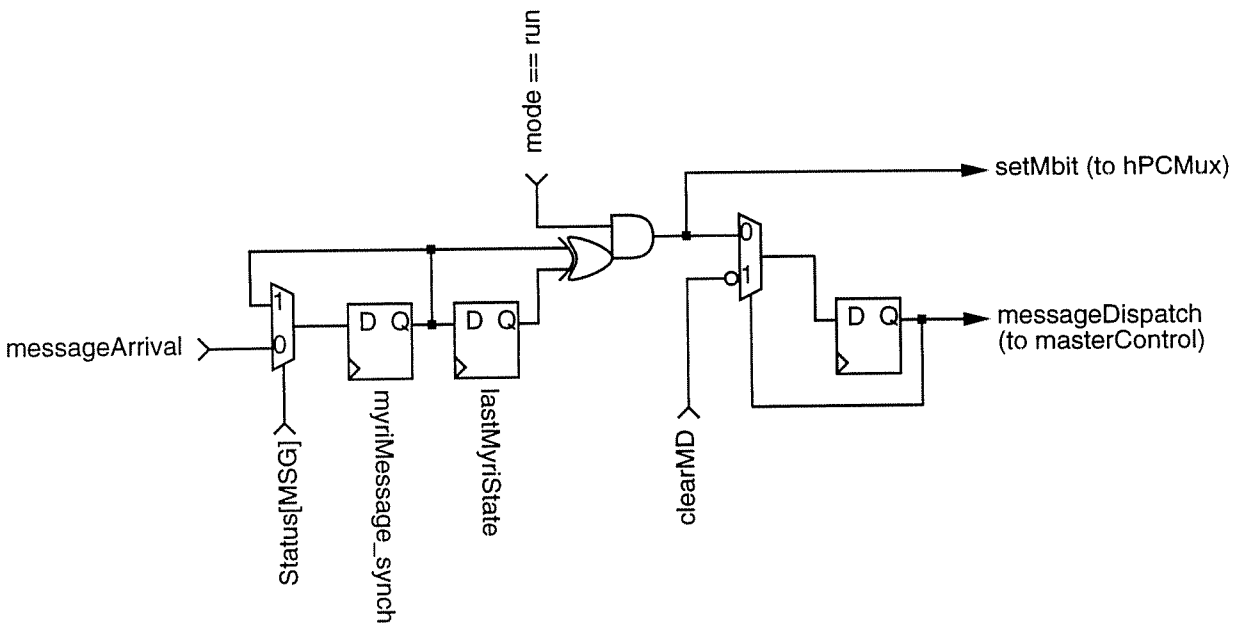


Figure 31: Netboy Schematic

two 81188As is the limited number of independently tristatable bidirectional I/O pins supported; Mbus specifies at least six groups of signals that must be tristated independently: MAS_, MBB_ MAD[63:0], the Mbus transaction status bits, (MRDY_, MRTY_, and MERR_) MSH_ and MIH_; each 81188A can only support four groups of independently tristated signals. Furthermore, even though the number of registers needed to capture fault and block downgrade state and implement control registers and FSMs is less than the number of flip-flops provided in each 81188A, with all the registers inserted for inter- and intra-chip timing and Mbus timing purposes, (see Section 9.8) we need two devices.

Given the tristate signal constraint, we are forced to put the master logic (which drives MAS_ and MBB_) into one 81188A dubbed "mchip", while we put the slave (driving the transaction status bits, MSH_ and MIH_) in the other device, called "schip".

While it would be desirable to put the entire datapath in one or the other of the two devices, this turned out to be impossible because of the size of the datapath, so it is partitioned across schip and mchip; therefore both devices drive the MAD pins and have duplicated bus interface registers, which are described in Section 9.5.

9.7.1 schip

schip contains the Mbus Slave logic, Block Buffer, Tag Unit, (except for the newTag mux) and bus input and output registers. A block diagram is shown in Figure 32. schip is also used in the PLL feedback path to adjust the clock to Q time as described in section 7.5.

9.7.2 mchip

mchip, shown in Figure 33 consists of the Mbus master logic, the datapath (minus the Block Buffer) and the tagUnit newTag mux, since the block downgrade tag is stored in the datapath. mchip also generates the tag SRAM write enable signal.

9.8 Timing Issues

Given the partitioning described above, there are 22 control signals that must cross the boundary between the schip and mchip. There are substantial delays (at least 15nS) associated with this crossing, which pose potential timing problems for some of the signals, depending on how far from the edge of the chip

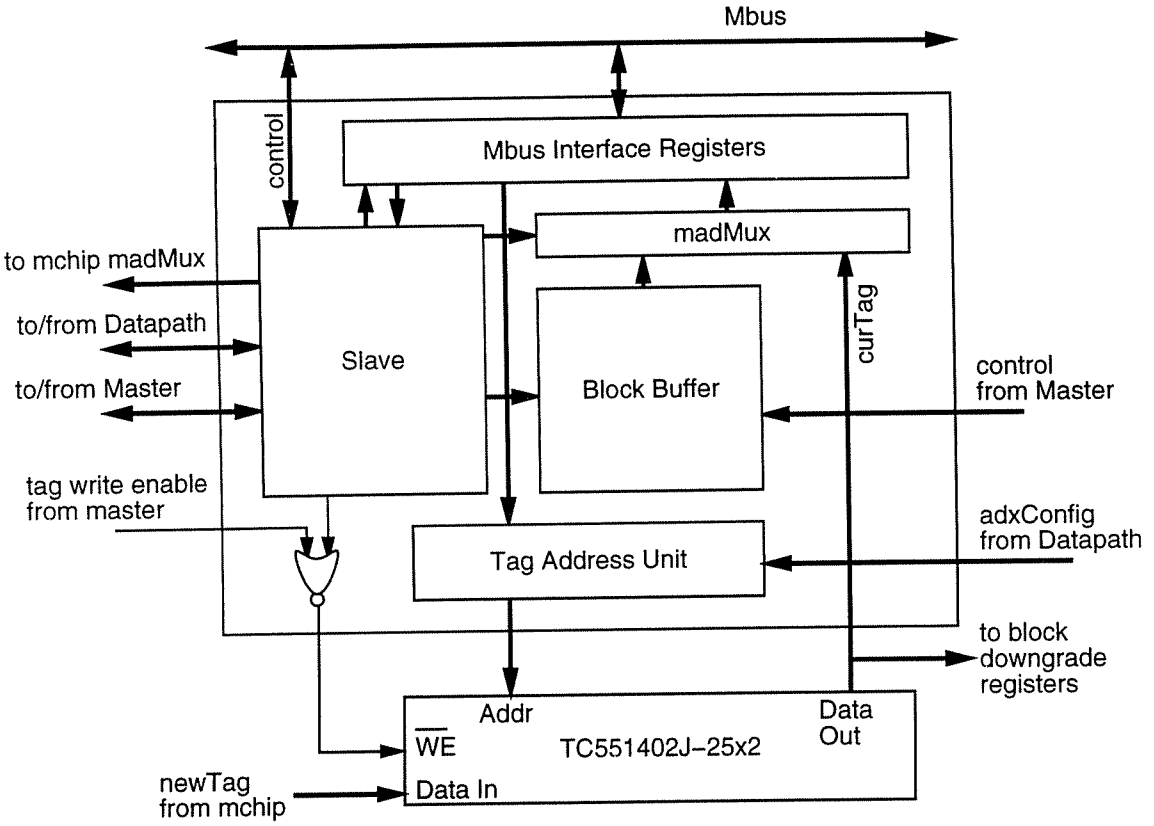


Figure 32: schip Block Diagram, with Tag SRAM

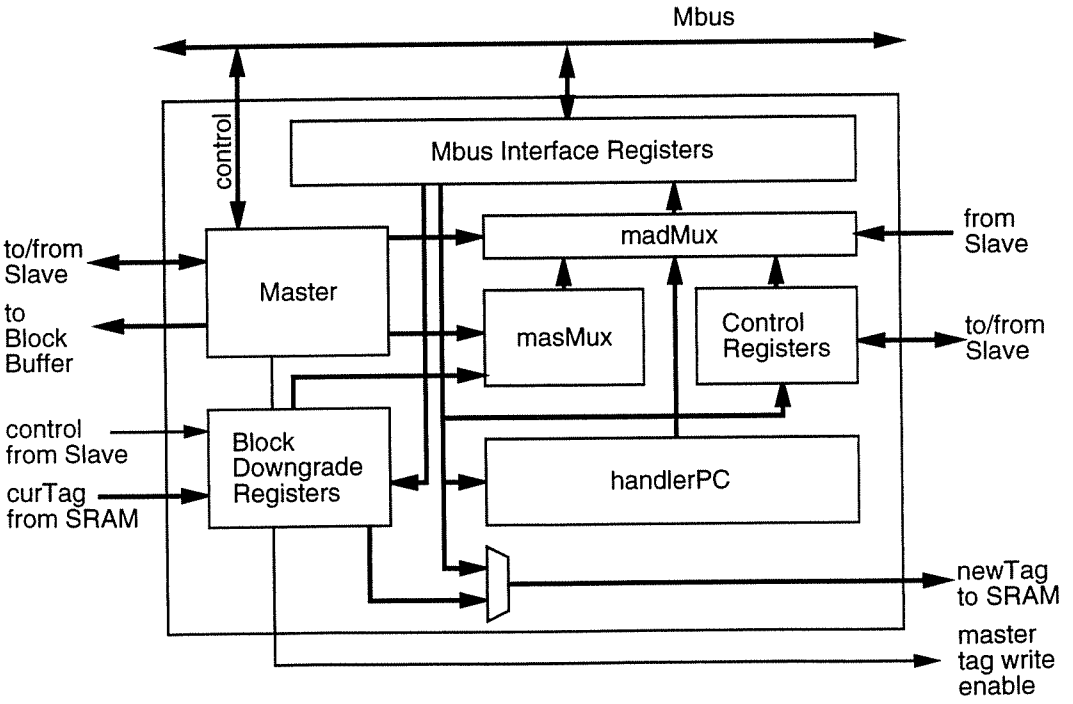


Figure 33: mchip Block Diagram

any registers being driven by those signals have been placed. Since Altera's Max+Plus II software does not do timing-driven placement spanning chips, we must take measures to prevent cross-chip timing violations by hand.

To this end, all datapath control signals generated by the slave are resynchronized at the mchip before being distributed to the datapath to eliminate setup and hold violations in datapath registers, which can lead to unpredictable behavior possibly leading to Vortex failure. Likewise, control signals generated by the master are resynchronized at the schip. The slave-master interface signals are registered both before leaving the schip and when received at the mchip.

9.8.1 Slave-Master Interface

The slave-master interface (described in Section 9.1.3) consists of a command word and a command strobe, which is generated by the slave and is used by the master to latch the command word. Since the command data and its write strobe are generated and travel together, it is important that they be synchronized with respect to one another; differing delays between the command and the strobe could cause setup and hold violations at the masterRequest register, (Section 9.2.1) leading to unpredictable master behavior.

To prevent this problem, the masterRequest and strobeMreq signals are resynchronized before leaving the schip, and again when they are received at the mchip before being passed on to the masterControl state machine. Because of this synchronization, there is a two cycle delay between the generation of a masterRequest by the slave and the masterControl state machine activation. This caused a performance problem in one version of the Vortex logic; see Section 12.1.

9.8.2 Slave-Datapath Interface

As described in Section 9.3.1, the bbMux select logic is resynchronized before being distributed into the multiplexor to alleviate setup and hold violations at the bbOut register. The mux select resynchronization plus the bbOut and mad_out registers combined with the slave FSM latencies make returning Block Buffer data until before cycle A+4 impossible, and the slave FSM is designed to begin driving MRDY_ during that cycle. handlerPC data must be driven out onto the bus starting at A+4 as well, but the slaveState signal must be resynchronized at the mchip also. This is why slaveState[0] is registered before being decoded in Figure 25; it is being resynchronized for chip boundary timing, delayed for functional correctness, and helping to break the timing path from the slaveState pin through hPCMux to the hPCMuxOut_reg register.

9.8.3 Master-Block Buffer Interface

As with the slave-datapath interface, the master block buffer write strobes (generated by the master FSM during a block downgrade CRI) and the data_in register in the schip, which captures the block data from the bus must be synchronized. The write strobe must reach the block buffer register exactly when the block data is valid in the data_in register. Fortunately, the resynchronizing registers for the write strobes create the one cycle delay needed between the strobes (which are generated when the master FSM sees the MRDY_ signal on the bus) and the validity of the data_in register, which is triggered by the MRDY_ signal as well.

10 Printed Circuit Board Design

The Vortex printed circuit board (PCB) is a standard-width, double-sided Mbus module. It has 10 layers, split into four power planes and 6 signal layers and is impedance controlled. The component side of the board is shown in Figure 4 and is explained in Section 1.4. The back side of the board is devoted to bypass capacitors and resistors, as well as the 120 pin Mbus connector.

Designing a small printed circuit board the size of an Mbus module to function correctly at 50MHz is tractable with today's PCB technology, but careful attention must be paid to power distribution, and signal impedance control and termination.

The Vortex PCB was designed by Rob Pfile using Cadence's Composer software and layed out by Leonard Bernal of Yamamoto USA, Inc. with Cadence's Allegro PCB package. The boards were fabricated by Hadco Corporation.

10.1 Impedance Control

Mbus signal impedance is specified at 50 ohms ± 10%. Given 10 layers (6 signal, 4 power), 0.5 ounce outer and 1 ounce inner layer plating thickness, and the dielectric constant of FR-4 fiberglass, core and prepreg thicknesses for 6 mil trace/space width were calculated for nominal 50 ohms impedance. The layer thicknesses were found to be compatible with the Mbus specification of 0.062 inches ± 0.008 inch for overall board thickness.

10.2 Mbus Signal Lengths

The Mbus specification states that control signals between the Mbus connector and any devices talking to the bus must be less than 3.4 inches in total length, with a stub length of no more than 2 inches, while MAD signals can be no longer than 5 inches with a stub length less than 3 inches. Furthermore, Mbus signals fanning out to more than one device must be routed as a star, with the Mbus connector pin at the hub.

Because of the chaotic pin assignments on the FPGAs (described in Section 12.2.2), the manhattan (x-y) lengths of several control and bus signals violate both the overall trace and stub trace length constraints [Sun93]. To minimize the routed length of these signals, the Mbus pins were routed by hand, and although many do violate the specification, no timing problems have been observed. This is probably due to the reasons given at the end of Section 7.5; the Mbus we are running on is superior to the specification.

10.3 Signal Termination

Mbus Clock signals are terminated with HSMS-2822 series schottky barrier diodes per the module design guide [Sun93].

Due to transmission line effects and the fast edge rates (< 1nS) of the Altera FPGAs, signals longer than 3 inches on the Vortex board are susceptible to overshoot, undershoot and ringing. To eliminate any possible latch-up or logic sense errors caused by this, all signals driven by the FPGAs longer than 3 inches are terminated with HSMS-2822 diodes at the receiver. The diode pairs clamp the transmitted signal voltage to approximately 5.3V and -0.3V, and eliminate ringing.

10.4 Power Distribution

Because the inductance of board traces can be quite substantial at the high edge rates produced by the FPGAs, it is important to provide a good power distribution system on the PCB. Altera recommends bypassing each power and ground pin pair with a 0.22μF capacitor placed nearby. In addition, Triquint gives a power distribution scheme for the GA1088 which includes a small ground plane on the component layer of the board and several 0.1μF bypass capacitors. Both of these guidelines were followed, and every other IC on the board was given 0.22μF bypass capacitors as well. Finally, the supply voltage is bypassed with four 47μF capacitors to handle lower frequency power transients.

The PCB has four power planes, two each for VCC and ground.

11 Verification

The verification process for Vortex was part of the design process, though in this report the two steps are split into two sections. Static timing verification had a great deal of influence on how the design was eventually partitioned, and led to a number of functional and implementation changes for individual components (notably the slave FSM) that allow the design to run at 50MHz. Gate simulation with timing exposed the critical chip-crossing signals that needed resynchronization that the static timing tool could not catch.

This section also describes our methods for testing the Vortex boards once they had been fabricated and populated. This testing included basic tests from the Sparcstation monitor and more strenuous random testing by a C program and Solaris driver designed for Vortex.

11.1 Functional Verification

The logic for Vortex was written entirely in Verilog at the register transfer level, (RTL) and so could be fully simulated. Using Cadence's Verilog-XL, the master and slave FSMs were first tested against custom stimulus for functional correctness. Once the FSM logic appeared to be correct, Verilog models for the SRAMs and clock generation circuits were developed, and all Vortex logic, including board-level component logic, was tested against a model of Sun's Viking processor on the Mbus. The Viking models do not execute SPARC code, but rather produce Mbus transactions as specified by a driver file. Once the design was functionally correct, it was synthesized using Synopsys.

11.2 Timing Verification

Functional verification is only half of the picture. Once the design source is correct, the synthesized gates must be checked for functional errors (though they are unlikely), and more importantly checked for timing correctness. Meeting timing goals is an iterative process during which some functional changes are introduced and parts of the design are re-implemented in different styles. In this case the changes included inserting registers into state machine input paths as described in Section 9.1.1; sometimes this required functional changes to the state machines and datapath. The slave state machine was reorganized several times to ensure correct operation at 50MHz, and parts of the design were hand-instantiated to overcome synthesis limitations. Both intra- and inter-FPGA timing was considered using two methods, static timing analysis and gate-level simulation.

11.2.1 Static Timing Analysis

Altera's Max+Plus II tool provides a static timing analyzer. This tool uses a characterization of the FPGA and a delay model to estimate the total delay between flip-flop outputs and inputs within an FPGA after a design has been successfully placed and routed. This tool exposed the critical paths in the schip and mchip, which led to functional and implementation changes as described above. Because the timing analyzer can only consider one device at a time, inter-chip timing problems had to be exposed through simulation.

11.2.2 Gate-Level Simulation with Timing

Once static timing analysis and redesign yielded schip and mchip implementations that were capable of 50MHz internally, the interactions between the two chips had to be tested. The only way to do this was to extract a gate-level Verilog description of each FPGA from Max+Plus II and simulate them together. Max+Plus II annotates these Verilog descriptions with the timing information produced by the static timing analyzer, and inserts check code around each flip-flop in the design to detect setup and hold violations.

The gate-level descriptions of the schip and mchip were dropped into the original RTL framework and re-simulated against the Viking model. This step exposed several intra-chip timing problems which were solved by inserting synchronization registers as described in Section 9.8. All Vortex functions were tested using the gate level Verilog and the Viking model; this simulation completes with no timing violations.

Finally, Mbus transaction traces were extracted from the C++ simulation, and used to produce Mbus transaction commands for a modified version of Sun's Viking/Mbus model. The transaction traces contained both the transaction's MAS_ cycle and slave responses. The Verilog simulation was modified to produce the transactions from the trace file, then check the response to each transaction directed at Vortex to verify that the C++ model of the hardware was functionally the same as the Verilog code. This simulation was not completed due to time constraints; we had working printed circuit boards before the simulation environment was finished.

11.3 Hardware Testing

The Vortex board was tested using three methods. First, basic board-level tests (power and clock generation) were performed, then simple functional tests from the boot monitor were run. Lastly, a driver and a C program were developed to run intensive random tests of the logic.

11.3.1 Initial Board Bringup

When the first Vortex board was finally assembled, we first smoke-tested the board in a Sparcstation-10. The machine did not catch fire, so the clock generation circuitry was tested using an oscilloscope. The clock termination diode and PLL bugs (Section 12.4.1) were discovered and fixed at this point.

11.3.2 Low-level Tests (Forth)

Once the FPGAs were being clocked properly, the next step in testing was to write a series of diagnostics in Forth and execute them from the Sun OpenBoot monitor. The OpenBoot monitor provides a Forth interpreter running in it's own virtual address space, and provides functions for allocating memory, and establishing mappings between Forth virtual addresses and bus physical addresses in addition to the standard Forth commands. It is incredibly useful for low-level hardware testing.

The Forth tests exercized all Vortex functions: Tempest memory snooping, tag space reads and writes, User Tag space tag downgrades, and register reads and writes, both cacheable and non-cacheable. The MSH_ bug described in Section 12.4 was discovered and fixed. Throughout these tests the Mbus was monitored using an Mbus extender card connected to a Hewlett-Packard 16500A Logic Analysis System.

11.3.3 Random Tester

Our final test was to write a C program utilizing the Solaris thread package to simulate the load placed on Vortex by user protocol and compute code. This required first writing a rudimentary driver for Vortex; once it was working the tester itself was written.

The tester allocates two pages of tempest memory through the driver, then maps the corresponding tag pages and uncached aliases of the memory pages. A non-Tempest data "home page" is allocated to mimic another node's shared memory. A simulated user thread is forked, which installs a SIGBUS signal handler to catch block access faults, then reads and writes from addresses in the Tempest cacheable aliases at random. The simulated protocol thread caches the handlerPC CCR and loops, waiting for a block access fault. While it is not handling faults, it randomly upgrades and downgrades tags. When the user thread causes a block access fault, the "real" protocol code is dispatched, which copies the data in from the home page and upgrades the tag, then causes the compute thread to retry the faulting memory access instruction. All data and tag writes are shadowed into separate spaces so that the correctness of block downgrades and tag writes to SRAM can be checked.

The random tester exposed the second bug explained in Section 12.4; the bug was due to a error in the Slave FSM. It manifested itself as mysterious SIGBUS signals in the protocol thread. Vortex was not acknowledging CI transactions when preceded by a particular sequence of transactions on the Mbus. After this bug was fixed, the first Vortex board passed random testing, eventually handling billions of block access faults with no errors.

12 Problems

Several problems were encountered while designing and testing Vortex. There were problems related to meeting the timing goal of 50MHz, which were eventually resolved by changing the logic design. We encountered limitations of the CAD tools, as well as errors in the functional specification. Finally, there were bugs in the logic and board implementations that were not caught in simulation, related to the differences between the processor simulation model (a Viking processor) and the hyperSparc processors we are actually using.

12.1 Timing

Vortex is not capable of issuing write transactions from its master port. The Mbus specification requires write data to appear on the bus immediately following the MAS_ cycle. Because of the latencies in the datapath introduced by the various mux output registers, it would have required extra multiplexor logic to support writes; the mchip was already too full to accommodate this logic. For this reason, downgraded blocks are not directly written back to memory and instead must be read out of the Block Buffer by the protocol thread and written back to memory. The mihSeen bit when reading tags is an attempt to minimize the performance impact of this shortcoming; see Section 5.1.5.

As described in Sections 9.1.1 and 11.2, the slave logic did not originally run at 50MHz. By careful insertion of registers between decoders and state registers, and breaking up decoding into two levels in the slave state machine, the 50MHz goal was met, but in doing so a functional bug was introduced, see Section 12.4.

Inserting registers into control and data signal paths also increased the overall density of the design. At least 286 registers, or 14% of the available 2016 FPGA matrix registers are consumed by registers whose sole function is to break long timing paths. This exacerbates the problems described in Section 12.2.2.

Because of the resynchronization registers inserted at chip boundaries (see Section 9.8, it takes three cycles between the time the slave requests master activity and the cycle that Vortex finally requests the bus. When the protocol thread is requesting a tag downgrade, the original write is given the Relinquish and Retry acknowledgment. The hyperSparc bus controller releases the bus, but before Vortex has requested the bus, it re-acquires the bus and retries the downgrade, causing all tag downgrade writes to take twice as long as is necessary. The solution to this problem was to insert a wait state in the slave to delay the acknowledgement to the write by one cycle. This gives Vortex's master a chance to acquire the bus and complete the tagDowngrade request before the protocol cpu re-acquires the bus.

12.2 CAD Tool problems

12.2.1 Synopsys

Synopsys was initially developed as a ASIC synthesis tool, and is not well-equipped to deal with the discrete routing structure of FPGAs. This is unfortunate, since the lack of a good model of FPGA routing hampers it's ability to apply it's timing-driven synthesis algorithms, which are generally very powerful. As such, Synopsys was only used to translate Verilog to a netlist that Max+Plus II could understand (EDIF).

A further problem with synopsys lies with the FPGA target library. Since the library only supplies single flip-flops, wide datapath registers and structures are reduced to collections of single flip-flops. This loss of high-level design structure made it very hard for Max+Plus II to consider the large-scale structure of the design.

12.2.2 Max+Plus II

Both the schip and mchip designs have very high logic cell utilization (80% for schip, 62% for mchip) and high pin utilization (88% and 82%). It is very difficult for Max+Plus II to successfully route these designs. We found that with *any* pin constraints, the chips were unrouteable. Therefore it was impossible to assign pins intelligently to minimize PCB trace route lengths between the schip, mchip and the Mbus connector; to minimize any problems the Mbus signals were hand-routed as described in Section 10.2. More importantly, not being able to assign pins until the design was correct (both functionally and timing-wise) serialized the logic design and PCB layout, which caused about a month's delay in producing the first working board.

A further complication was that not only was pin preassignment impossible, but assigning the pins as Max+Plus II had chosen them initially caused successive re-routes of the design to fail! This forced us to treat the design as though it were an ASIC, fully simulating it before fixing the pins for printed circuit board design. Fortunately, Max+Plus II can output the design after place and route as an AHDL file. Minor design changes can be made to the AHDL file, and together with the extracted placement information, the new design can be successfully re-routed. This is because the cell names are not modified

by Max+Plus II when the design is input as AHDL; when using EDIF from re-Synopsized Verilog, the LE names change and the placement information is useless for the next place and route run. All bug fixes described below were accomplished using the AHDL editing technique.

12.3 Incorrect Functional Specification

There were some problems with our original functional specification for Vortex. The first is explained in section 5.1.3; originally we had hoped to enforce fine-grain access control only on the user processor, but this proves to be impossible due to how we overload the cache coherence mechanisms.

The second had to do with clearing the Fault Status CCR. Originally, the protocol processor could read stale status information because the CI needed to invalidate the cached copy was not bound in any way to the write operation; the status write would queue a master request and be acknowledged normally, opening a window for the protocol thread to read the old CCR from the cache before the CI happens on the bus. We deal with this as described in Sections 5.2.4 and 6.4.4 under the handlerPC item, by issuing a read to a register which causes the protocol thread to block until the CCR has been invalidated.

Both of these errors were discovered while implementing the slave FSM logic.

12.4 Bugs

There were two logic bugs which went unnoticed during simulation. The first was that the MSH_ pin was inadvertently designed as a bi-state output; the Mbus specifies this pin as open-drain, so that there can be multiple drivers. The Altera 81188A parts do not support open-drain outputs and so MSH_ is implemented as a tristate signal.

The second, and much more insidious bug was a problem with the slave FSM logic. Because of the many slave FSM redesigns for timing purposes, the minimum latency path through the machine (when the snooped transaction was not interesting to Vortex) was inadvertantly increased to 3 cycles. This is a problem when an snooped uncached write appears back-to-back with another transaction on the Mbus; the write is acknowledged by the memory controller in A+2. If the processor that issued the uncached write is executing memory-intensive code, it can issue another transaction in the cycle immediately following the acknowledgment at A+2. The slave FSM would miss this transaction. This bug manifested itself as an occasional bus timeout during random testing; it turned out that Vortex was missing CI transactions it was supposed to acknowledge. Contention between the hyperSparc write buffer and cache controller caused the CI to be immediately preceded by an uncached write. A series of uncached writes was being produced by a protocol processor block copy of Tempest memory through an uncached alias. The solution to this problem was to make sure the slave FSM was capable of returning to IDLE by A+3 by making part of the snooping decision earlier in the state machine. This fix was accomplished using the AHDL editing technique explained in Section 12.2.2; re-routing the bug fix from the Verilog was not successful.

This bug was not caught in simulation because the Viking (superSparc) processor does not issue bus requests in such quick succession. The Viking always inserts a dead cycle after receiving an acknowledgment and before generating the next address cycle, so the extra cycle of slave latency was not exposed as a bug. The hyperSparc processors we are using have Mbus interfaces which are much more aggressive than the superSparc.

12.4.1 PCB Bugs

Finally, there were a couple of PCB schematic errors. In two cases termination diodes were wired backwards, and so had to be mounted on the board upside-down. Also the GA1088 PLLs were miswired, requiring some pins to be cut and two blue wires to be installed. The planned Revision B PCB (see section 13) fixes both of these problems.

12.5 Non-correctable functional bugs

There were two problems with the functional specification which caused problems with the hyperSparc processor and Solaris.

Our error model originally included returning the Mbus Uncorrectable acknowledgement to malformed (unaligned, and incorrect SIZE or TYPE) uncached writes to Vortex registers and tag spaces. It turns out that this is not the best idea in the light of the hyperSparc's write buffer: the `st` instruction that causes the WR transaction on the Mbus is first entered into the write buffer, and the instruction stream moves on. When the WR is finally produced on the bus and the error ack is given, the corresponding `st` instruction is long gone from the pipeline, and the hyperSparc has no choice but to report this error as an Asynchronous error to the Mbus error controller. Solaris deals with the Asynchronous error by hanging the machine so hard that it must be power-cycled. Because the user could cause this simply by writing a tag to an non-cache block aligned address, we were forced to modify the Vortex slave logic to normally acknowledge malformed tag (and register) writes but ignore the write. Unfortunately the error register is still set but there is no way to signal the user that an error has occurred and the error syndrome should be checked. When an error occurs that can be reported to the user, (for instance, a malformed read) the error syndrome reported will be for the long-ago miswritten tag or register rather than for the fault that caused the error response. This makes debugging protocol code potentially very difficult.

A related problem occurred with how master FSM errors were to be reported to the user. We planned to use the Mbus INTOUT_ signal to generate a level-15 (broadcast) interrupt when the master was given an error acknowledgment. Unfortunately, Solaris deals with INTOUT_ the same way it deals with Asynchronous errors. We had to disable this feature, and master errors are not reported to the user.

If we manage to get the design to re-synthesize from Verilog, we can at least report master errors on tag downgrades by returning an indication that the master encountered an error when the tag is read back (which is done to flush the write buffer, see Section 5.1.4 for details.)

13 Future Work

As described in the previous section, we are currently incapable of making changes to the Vortex design at the Verilog level, which hinders our ability to make significant design changes. Minor design changes are still possible via AHDL. An important near-term goal is to reorganize and simplify the Verilog to allow a successful re-synthesis and re-route of the schip and mchip designs. Synopsys is currently at version 3.3a, and Max+Plus II is at version 5.3; we have not yet been able to try these two versions together to attempt a redesign.

Since the inception of this project, new synthesis tools have appeared on the market. Synplicity is an EDA startup company that sells an FPGA-specific synthesis tool. Initial results using beta versions of their Synplify tool indicate that Max+Plus II may be able to successfully place and route a modified schip design with the current pin constraints.

In the coming weeks we will be fabricating a revision of the Vortex printed circuit card which fixes the board-level wiring bugs. This revision has already been designed but has been on hold pending arrival of the balance of the 80 FPGAs we need to complete the 40-node Typhoon-Zero system.

14 Acknowledgments

I would like to thank my advisor, David Wood, for giving me the opportunity to work on the Typhoon-Zero project, and Steve Reinhardt and the rest of WWT for coming up with the ideas for Tempest, Typhoon, and Typhoon-Zero.

I would also like to thank several people at Sun Microsystems; thanks to Andreas Nowatzyk, leader of Sun's s3.mp group, for his guidance, technical help, and for allowing me to go to Sun to work on this project. Thanks also to the other s3.mp team members: Sanjay Vishin for help with Mbus controller design, both sample code and direction, Mike Parkin for Synopsys and Verilog simulator help, Bill Radke for help with Mbus memory controllers, Michael Browne for his help with the Verilog-XL PLI, and Gunes Aybay for help with Verilog design. Thanks to Søren Christensen for his help with Verilog, the Cadence schematic and board tools, insights on digital design, and Verilog help. Thanks are also due to Jeff Rulifson, director of Sun's Technology Development group, for his willingness to support me financially while at Sun. Thanks also to John Crowell and Stephanie Hinman for arranging all the day to day details of life at Sun for me, and for putting up with me bugging them all the time.

I'd like to thank my parents for supporting my education and helping me achieve my goals throughout my lifetime. Finally, I'd thanks to my long-suffering girlfriend, Suruchi Bhatia, for having the patience to endure being separated for so long while I finished my degree.

References

[Alt95] Altera. *Altera 1995 Data Book.* Altera Corporation, March 1995.

[FW95] Babak Falsafi and David A. Wood. When does dedicated protocol processing make sense? Submitted for publication, April 1995.

[Kel91] Edmund G. Kelley, et. al. *SPARC^{TM} MBus Interface Specification.* Sun Microsystems, Inc., March 1991. Revision 1.2.

[KR95] Mohammed A. S. Khalid and Jonathan Rose. The Effect of Fixed I/O Pin Positioning on The Routability and Speed of FPGAs. Technical Report CSRI-325, Computer Systems Research Institute, University of Toronto, Toronto, Canada M5S 1A1, May 1995.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, 1984.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[Sun93] Sun Microsystems, Inc. *MBus Module Design Guide.* Sun Microsystems, Inc., May 1993. Version 2.1.

[TM91] Donald E. Thomas and Philip Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[Tos94] Toshiba. *Toshiba 1994 Static Ram Data Book.* Toshiba America Electronic Components, Inc., 1994.

[Tri94] TriQuint. *TriQuint 1994 Clock Products Data Book.* TriQuint Semiconductor, Inc., 1994.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

A Verilog Source Code for Vortex

Please contact the author for this appendix; it is approximately 60 pages of code.

B Printed Circuit Board Schematics for Vortex

The schematics for the board are not available in this version of the paper.

C Handler Dispatch Code

C.1 proto_asm.s

```
/*
 * Copyright (c) 1995 by Mark Hill, James Larus, and David Wood for the
 * Wisconsin Wind Tunnel Project.
 *
 * ALL RIGHTS RESERVED.
 *
 * This software is furnished under a license and may be used and copied
 * only in accordance with the terms of such license and the inclusion of
 * the above copyright notice.  This software or any other copies thereof
 * or any derivative works may not be provided or otherwise made
 * available to any other persons.  Title to and ownership of the
 * software is retained by Mark Hill, James Larus, and David Wood. Any
 * use of this software must include the above copyright notice.
 *
 * THIS SOFTWARE IS PROVIDED "AS IS".  THE LICENSOR MAKES NO WARRANTIES
 * ABOUT ITS CORRECTNESS OR PERFORMANCE.
 *
 * This source file is part of the Tzero simulator, originally written by
 * Steven K. Reinhardt and subsequently maintained and enhanced by Babak
 * Falsafi, Shubhendu S. Mukherjee, and Steven K. Reinhardt.
 */
#include <machine/asm_linkage.h>
#include "ni_macros.h"
#include "tzero.h"

/*
 * We can use locals and ins since we're never going to return.
 */

#define r_t0_cregs %l7
#define r_t0_uncregs %l4
#define r_ppt_base %l5
#define r_ppt_base_plus8 %l6
#define r_ni_base %i0

.global _t0_dispatch_start, _t0_dispatch_loop

_t0_dispatch_start:

save %sp, -SA(MINFRAME), %sp
set T0_USER_CACHED_REG_VA, r_t0_cregs
set T0_USER_UNCACHED_REG_VA, r_t0_uncregs
set T0_PHYS_PG_TBL_BASE, r_ppt_base
add r_ppt_base, 8, r_ppt_base_plus8
set NI_BASE_VA, r_ni_base

set _t0_dispatch_loop, %l0
st %l0, [r_t0_uncregs + T0_HANDLER_BASE_OFFS]

/* reread base to make sure it's committed */
ba _t0_dispatch_loop
```

```
ld [r_t0_uncregs + T0_HANDLER_BASE_OFFS], %l0


/* this is where we come before re-entering loop: */
/* load status bits to make sure any pending stores are completed */
t0_dispatch_restart:
ldub [r_t0_uncregs + T0_STATUS_CLEAR_OFFS], %l0

/* status = 0000 */

_t0_dispatch_loop:

/* pc in %l0, ppn in %l1 */
ELSIE_SPIN_WHILE_EQ_PC:
ldd [r_t0_cregs + T0_STATUS_BLK1_OFFS], %l0
ldd [r_t0_cregs + T0_STATUS_BLK2_OFFS], %l2 /* pg_offs in %l2, */
/* flt_type in %l3 */
jmp %l0
and %l3, 0x1f, %l3 /* mask out MID  */
/* from VortexPlus */

nop; nop; nop; nop; nop /* 4 + 5 = 9 */
nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 0001 = block access fault */

status0001:
ldd [r_ppt_base_plus8 + %l1], %o2 /* %o2 = home, %o3 = handler_tbl */
mov 1, %l0
ldd [r_ppt_base + %l1], %o0 /* vpn in %o0, usr_ptr in %o1 */
ld [%o3 + %l3], %o3 /* fn ptr in %o3 */
or %o0, %l2, %o0 /* va = (vpn | offs) */
jmpl %o3, %o7
stb %l0, [r_t0_uncregs + T0_STATUS_CLEAR_OFFS] /* clear status bit */

ba _t0_dispatch_loop
/* load status bits to make sure store has completed */
ldub [r_t0_uncregs + T0_STATUS_CLEAR_OFFS], %l0

nop; nop; nop; nop; nop; nop; nop /* 9 + 7 = 16 */

/* status = 0010 = message */

status0010:
/* get size (in words) */
lduh [r_ni_base + NI_STATUS_REG + 2], %o0
/* get first word (source) and second word (handler pc) */
ldd [r_ni_base + NI_INPUT_QUEUE], %o4
/* user size in bytes = (system size in words - 2) * 4 */
sub %o0, 2, %o0
sll %o0, 2, %o1
/* call handler, move source to arg reg in delay slot */
jmpl %o5, %o7
mov %o4, %o0
```

```
/* clear status bit iff ni is empty */
/*  we could loop here until ni is empty, but I don't want */
/*  to risk starving block access faults */
lduh [r_ni_base + NI_STATUS_REG], %o0
tst %o0
bnz _t0_dispatch_loop
mov 2, %l0
stb %l0, [r_t0_uncregs + T0_STATUS_CLEAR_OFFS] /* clear status bit */

/* must re-check NI in case msg arrived in window; */
/* if so, re-set status bit */
lduh [r_ni_base + NI_STATUS_REG], %o0
tst %o0
/* These next 3 insts always end up at t0_dispatch_restart */
/* but the store is only executed if %o0 != 0 */
bnz,a t0_dispatch_restart
stb %l0, [r_t0_uncregs + T0_STATUS_SET_OFFS] /* set status bit */
ba,a t0_dispatch_restart

/* status = 0011 = message & block access fault... do baf */

status0011:
ldd [r_ppt_base_plus8 + %l1], %o2 /* %o2 = home, %o3 = handler_tbl */
mov 1, %l0
ldd [r_ppt_base + %l1], %o0 /* vpn in %o0, usr_ptr in %o1 */
ld [%o3 + %l3], %o3 /* fn ptr in %o3 */
or %o0, %l2, %o0 /* va = (vpn | offs) */
jmpl %o3, %o7
stb %l0, [r_t0_uncregs + T0_STATUS_CLEAR_OFFS] /* clear status bit */

ba _t0_dispatch_loop
/* load status bits to make sure store has completed */
ldub [r_t0_uncregs + T0_STATUS_CLEAR_OFFS], %l0

nop; nop; nop; nop; nop; nop; nop /* 9 + 7 = 16 */

/* status = 0100 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 0101 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 0110 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */
```

```
/* status = 0111 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 0100 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1001 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1010 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1011 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1100 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1101 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1110 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */

/* status = 1111 */

ta 1
nop; nop; nop; nop; nop; nop; nop /* 1 + 7 = 8 */
nop; nop; nop; nop; nop; nop; nop; nop /* 16 */
```

C.2 tzero.h

```
/*
 * Copyright (c) 1995 by Mark Hill, James Larus, and David Wood for the
 * Wisconsin Wind Tunnel Project.
 *
 * ALL RIGHTS RESERVED.
 *
 * This software is furnished under a license and may be used and copied
 * only in accordance with the terms of such license and the inclusion of
 * the above copyright notice.  This software or any other copies thereof
 * or any derivative works may not be provided or otherwise made
 * available to any other persons.  Title to and ownership of the
 * software is retained by Mark Hill, James Larus, and David Wood. Any
 * use of this software must include the above copyright notice.
 *
 * THIS SOFTWARE IS PROVIDED "AS IS".  THE LICENSOR MAKES NO WARRANTIES
 * ABOUT ITS CORRECTNESS OR PERFORMANCE.
 *
 * This source file is part of the Tzero simulator, originally written by
 * Steven K. Reinhardt and subsequently maintained and enhanced by Babak
 * Falsafi, Shubhendu S. Mukherjee, and Steven K. Reinhardt.
 */
#ifndef _tzero_h
#define _tzero_h

#ifndef LOCORE /* C-only stuff */

#include "tppi_types.h"

struct T0Regs
    Uint64 block_buffer[4]; /* 0x000 */
    Uint32 handler_pc; /* 0x020 */
    Uint32 fault_ppn; /* 0x024 */
    Uint64 handler_pc_block_pad1; /* 0x028 */
    Uint32 fault_blk_offs; /* 0x030 */
    Uint32 fault_type; /* 0x034 */
    Uint64 handler_pc_block_pad2; /* 0x038 */
    Uint32 handler_base; /* 0x040 */
    Uint32 handler_base_pad; /* 0x044 */
    Uint64 tag_base; /* 0x048 */
    Uint16 error_status; /* 0x050 */
    Uint8 mode; /* 0x052 */
    Uint8 mih_delay; /* 0x053 */
    Uint8 ci_delay; /* 0x054 */
    Uint8 cached_reg_va; /* 0x055 */
    Uint8 reg_pad[2]; /* 0x056 */
    Uint8 status_set; /* 0x058 */
    Uint8 status_clear; /* 0x059 */
;

#define t0_cached_regs       ((volatile struct T0Regs *)T0_USER_CACHED_REG_VA)
```

```
#define t0_uncached_regs      ((volatile struct T0Regs *)T0_USER_UNCACHED_REG_VA)
#define t0_priv_cached_regs   ((volatile struct T0Regs *)T0_PRIV_CACHED_REG_VA)
#define t0_priv_uncached_regs ((volatile struct T0Regs *)T0_PRIV_UNCACHED_REG_VA)

/* note that 'p' must be dword-aligned!! */
#define t0_tag_ptr(p) ((volatile Uint8 *)((Uint32)(p) + T0_TAG_VIRT_OFFS))
#define t0_tag(p) (*t0_tag_ptr(p))

#define t0_uncached_alias(p) ((Uint32)(p) + T0_UNCACHED_VIRT_OFFS)

#endif LOCORE

#define TAG_BLK_SIZE 32

#define T0_STATUS_BLK1_OFFS 0x020
#define T0_STATUS_BLK2_OFFS 0x030
#define T0_HANDLER_BASE_OFFS 0x040
#define T0_STATUS_SET_OFFS 0x058
#define T0_STATUS_CLEAR_OFFS 0x059

#define T0_USER_REG_BASE_VA 0xffff0000

#define T0_USER_CACHED_REG_VA (T0_USER_REG_BASE_VA + 0x0000)
#define T0_USER_UNCACHED_REG_VA (T0_USER_REG_BASE_VA + 0x1000)
#define T0_PRIV_CACHED_REG_VA (T0_USER_REG_BASE_VA + 0x2000)
#define T0_PRIV_UNCACHED_REG_VA (T0_USER_REG_BASE_VA + 0x3000)

#define T0_TAG_VIRT_OFFS 0x10000000

#define T0_UNCACHED_VIRT_OFFS 0x20000000

#define T0_PHYS_PG_TBL_BASE 0x0f000000

#endif
```