The *Viking* Microprocessor (T.I. TMS390Z50) User Documentation

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-4510-02
Sun Microsystems Proprietary Revision 2.00 of November 1, 1990

Preface

This manual specifies the *user's* view of the *Viking* microprocessor. It is intended to provide all necessary information for the use of *Viking*, both hardware and software. Detailed pin timing and other physical information may be found in the TMS390Z50 Data Sheet from Texas Instruments.

The document is intended to fully specify the operation of *Viking* and includes:

An introduction to *Viking*
A simplified pipeline description
Guidelines for code generation
The programmer's model
A description of JTAG-based In-Circuit Emulation facilities
A description of internal software debugging facilities (breakpoints)
A detailed system interface definition

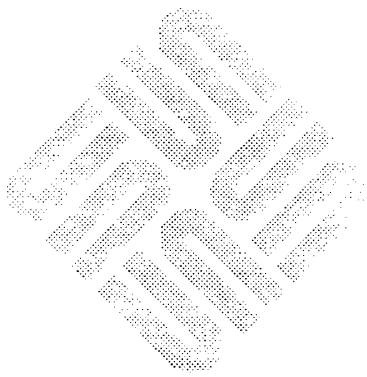
For background and supporting information, the following documents are useful:

The Version 8 SPARC Architecture Manual, including
Reference MMU, Memory Model, and Suggested ASI appendices
The *Viking* Cache Controller (*MXCC*) Specification (External Cache Controller)
The SPARC MBUS Specification
The DynaBus and *XBus* Specifications
The IEEE P1149.1 JTAG Specification
The Sun-4M System Architecture Specification
The SunDragon Architecture Manual
The *Viking* SRAM Test Documentation
T.I. TMS390Z50 (*Viking*) Data Sheet
T.I. TMS390Z55 (*MXCC*) Data Sheet

Effort has been made to make this an easy-to-use, accurate, and complete document. It may contain errors and/or omissions. Please report any erratum or requests for additional information to Greg Blanck (gblanck@Eng.Sun.COM).

Revision History

Revision	Date	Comment
01	9-25-89	First Release
02	11-1-90	Second Release



Contents

Tables

Figures

[Blank Page]

1

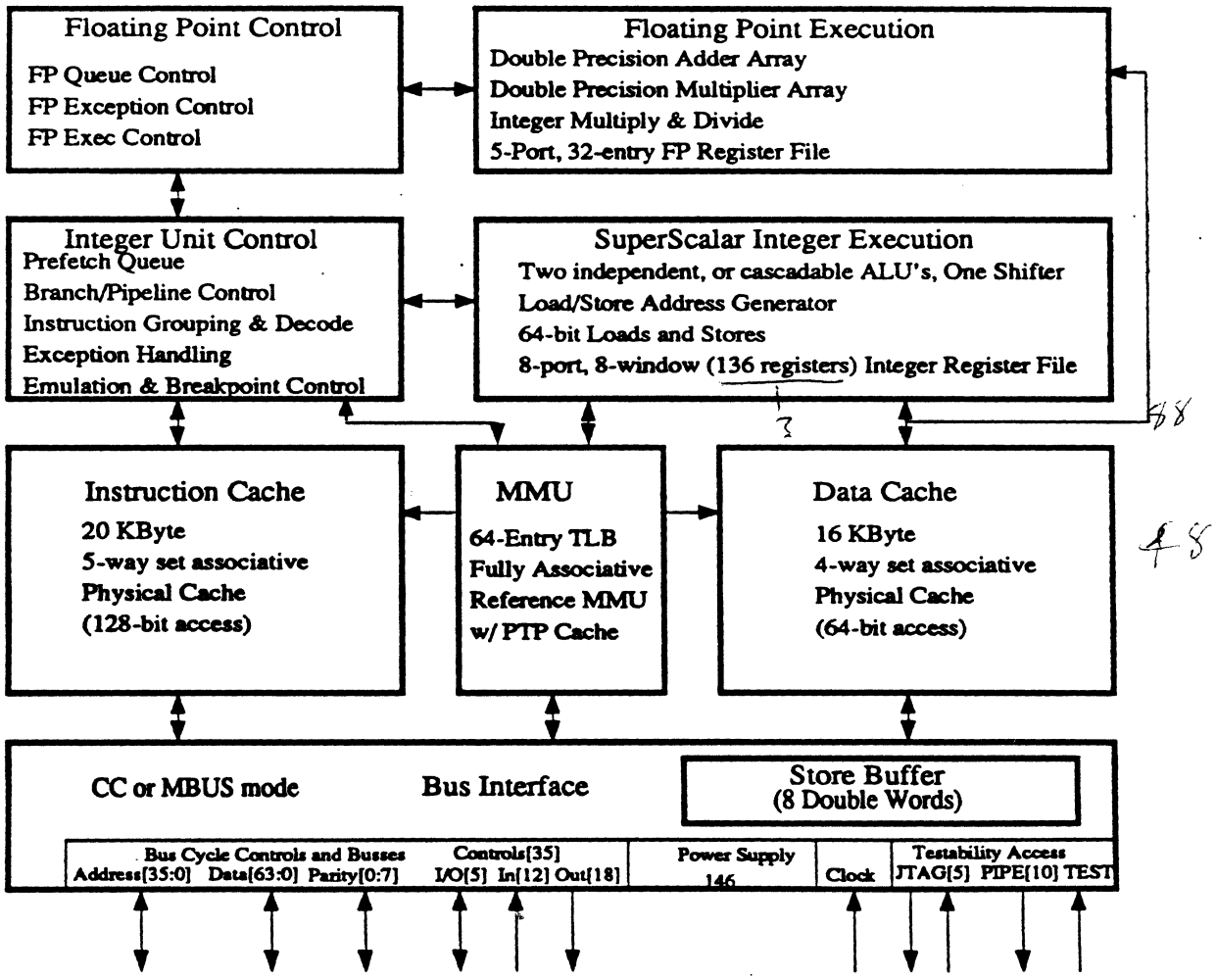
Introduction to *Viking*

[Blank Page]

1

Introduction to *Viking*

Viking is a highly integrated, high performance implementation of the SPARC RISC architecture. It is a single chip processor implemented in full custom BiCMOS technology by Texas Instruments. It is intended for use in a broad spectrum of system environments: from large scale multiprocessor systems, to low cost single user workstations and high performance embedded control applications. A simplified functional block diagram of *Viking* is shown below:



1.1. High Integration

Integrated within the processor are most of the support functions normally required to build a SPARC based system. These features total approximately three million transistors, and include:

1.1.1. Integer Unit

A fully SPARC compatible integer unit is provided on chip. This integer unit is a high performance *superscalar* (multiple instructions per cycle) design. Eight register windows are provided. Integer multiply (IMUL) and integer divide (IDIV) instructions are implemented in hardware.

1.1.2. Memory Management Unit

An implementation of the SPARC Reference Memory Management Unit (MMU) is included within *Viking*. The MMU provides a 64-entry TLB (Translation Lookaside Buffer) to translate virtual to physical addresses. A second-level Page Table Pointer (PTP) cache and a root pointer cache are included to reduce TLB miss penalties.

1.1.3. Floating Point Unit

A standard SPARC floating point unit and controller are included on chip. This FPU provides high performance single and double precision floating point arithmetic functions. Integer multiply and divide instructions are also performed by the FPU.

1.1.4. Instruction Cache

In order to increase performance and reduce the demands on an external memory system, a large instruction cache is included. This instruction cache is a 5-way set associative cache with 20KB total storage. The sets are independent; instructions at any physical address can be stored in any of the 5 set. The cache is a *physical address* cache. The instruction cache is **non-writable**, but is kept consistent with the data cache, and external memory, through extensive cache coherence support.

1.1.5. Data Cache

Fast execution of LOAD and STORE instructions is critical to high performance RISC processors. To achieve single-cycle execution of these instructions, a data cache is included on chip. This data cache is a 4-way set associative cache with 16KB total storage. These sets are independent; data at any physical address can be stored in any of the 4 sets. This cache enforces *cache coherence* with other caches in a system. The coherence mechanism is described in the System Interface chapter. The data cache is a *physical address* cache. Depending on the system environment, the data cache works in either write-through or copy-back mode. The behavior of each mode is explained in the Programmer's Model and the System Interface chapters.

1.1.6. Store Buffer

An 8-doubleword entry store buffer is provided to reduce the latency on ST. This is a FIFO queue which holds the data until resources allow data to be written out to the external cache and/or memory. This buffering allows the pipeline to continue execution, thereby increasing performance.

1.1.7. External Cache Support

Viking provides a flexible external cache interface. An external cache controller chip, such as the *MXCC*, can provide a complete implementation of a large, direct mapped, physically addressed external cache. This interface is described in detail in the System Interface chapter. The *MXCC* provides a single chip interface to the SPARC MBUS standard (level-2). It also provides a general purpose packet switched interface (the XBus) that can be used to interface to a variety of bus standards.

Both *Viking* and the *MXCC* are independently optimized to work with fully pipelined cache RAMs, and they both support SPARC's TSO (Total Store Ordering) and PSO (Partial Store Ordering) memory models. See SPARC Architecture Manual for more details.

1.1.8. Multi-Processor Cache Coherence support

Viking provides built-in cache coherence. The protocols are described in detail in the System Interface chapter. The protocol supports multiple cached copies of shared data for reading and in some systems for writing.

Physical address bus snooping is used to implement the coherence algorithms.

1.1.9. Hardware Breakpoints

To simplify software debugging, and reduce system development time, *Viking* provides on-chip hardware breakpoints. Code and data access breakpoints are provided. Virtual and Physical addresses can be selected.

A 16-bit instruction counter and a 16-bit cycle counter are provided for debug and performance analysis.

These breakpoints all have programmable *actions* when they occur. They may generate exceptions, interrupts, or toggle an external pin to help trigger external analysis equipment.

1.1.10. JTAG Emulation

Viking provides the equivalent of traditional ICE (In-Circuit Emulation) support internally in hardware. Through the JTAG (IEEE P1149.1) asynchronous scan interface, the state of the processor may be viewed or modified without changing other processor state. The processor may be single stepped through a program, with all processor states being observed after each cycle. This interface may be used to view or modify system memory as well.

Emulation operates at full processor speed, without affecting any pin timings, or loading. No test pod, nor other specialized hardware is required (except a JTAG control device with appropriate software).

1.2. Full Testability

Viking is a 100% testable device. All internal datapath and control logic can be tested using JTAG scan. Large internal arrays are not directly in the scan chain, but may still be tested through the serial JTAG interface.

An automatic power-up selftest (software) can be initiated with or without any external scan hardware. This, along with functional test of the arrays, gives high confidence that the device is correct, with minimal effort.

Boundary scan is implemented to perform system level testing.

All tests can be performed *in-circuit*; only a JTAG control device is required.

1.3. High Performance

Viking achieves high performance in many ways. These are broken down into two categories: High clock rate due to technology, and low sustained cycles-per-instruction (CPI) due to μarchitecture.

1.3.1. Clock rate/technology

A full custom BiCMOS implementation allows for a target frequency of 50MHz or greater. Advanced process technology makes possible many of the μarchitectural features described below. The system interface is designed for operation at these high speeds. An internal Phase Locked Loop is included to reduce system clock skew.

1.3.2. μArchitecture

In order to push beyond the improvement from clock rate, the μarchitecture has been optimized to execute multiple instructions simultaneously and critical instructions quickly. These μarchitectural features increase the average number of instructions executed per cycle by a factor of two for integer programs. A much greater improvement is found for floating point bound programs.

Viking typically executes programs from its cache at about 1.35 instructions per cycle (IPC), or about 0.74 clocks per instruction (CPI). This figure derates to about 1.1 IPC for large programs not fully contained in the cache. Floating point performance is generally much higher.

The optimizations are outlined below:

1.3.2.1 Multiple Instructions Per Cycle Execution

Viking can issue up to three instructions simultaneously. Certain rules are followed to determine how many of the available instructions may be executed in any particular cycle. These are fully described in section 3.4 — *Instruction Grouping Rules*.

1.3.2.2 Fast LOAD and STORE instructions

All LOAD and STORE instructions operate in a single clock cycle when the referenced data is present in the on-chip data cache. This includes 64-bit transfers and floating point transfers. When the data is not present in the on-chip data cache, a 5 cycle penalty is imposed to access the external cache. Each cache miss reads a block (32-bytes) of data from the external cache. These bus transactions are fully pipelined. The processor can use this data "on the fly", as the data arrives from the bus.

When in CC mode, no miss penalty is incurred for normal store misses. An internal store buffer holds the store transaction and allows the pipeline to continue.

No load-use interlocks are imposed on LD instructions. The instruction immediately following may use the data without incurring any delay. There are some cases of interlocks between the LD instruction and the following address calculations. (Described in section 2.3.1.1 — *LD (Load operation)*). The external bus is capable of nearly one transfer per cycle in the steady state (A mix of data and instruction fetches).

Because *Viking* uses fully physical caches with cache coherence, no *flushing* of cached entries must be done, and no virtual address aliasing conditions exist. Eliminating flushing overhead can boost performance significantly.

1.3.2.3 Floating Point Implementation

The *Viking* Floating Point Unit is tightly coupled to its integer execution pipeline, and allows one floating point operation *and* one memory reference to be issued in *every* clock cycle. *Viking* supports single and double precision operations, but not extended nor quad precision. The FPU maintains a 4 entry FIFO queue from which FPOPs are executed. Some operations require more execution cycles than others, for example FDIV (floating point divide) and FSQRT (floating point square root) take many more cycles than FADD. *Viking* FPU also handles the integer multiply and divide. FPOPs and FPEVs are executed in the order they are issued by the processor, allowing no out of order completion. Register dependencies can delay execution stream, and exceptions can interrupt the pipeline, sometimes requiring instruction aborts. *Viking* handles all cases of normalization, and register alignments for double precision arithmetic, directly in hardware. No *unfinished* exceptions (which consume extra cycles) are required.

More details of the FPU can be found in section 2.3.2 — *Floating Point Pipeline.*

[Blank Page]

2

Processor Pipeline Overview

2

Processor Pipeline Overview

The *Viking* µprocessor pipeline is presented in this chapter. This information is used throughout the document to describe *Viking*'s operation. The next chapter, Code Generation, suggests code generation strategies to attain maximum performance from *Viking*'s SuperScalar pipeline.

2.1. Pipeline Fundamentals

Viking's pipeline comprises eight stages, which execute in four clock cycles. Each stage has a unique function, which varies depending on the group of instructions being executed. In general, they follow the standard Fetch, Decode, Execute, Write Back model. The *Viking* pipeline stages are:

F0, F1, D0, D1, D2, E0, E1, WB

and each stage is defined in detail below.

2.1.1. F0/F1 (Fetch)

All instructions must be fetched before they are executed. However, not all instructions are fetched in the cycle immediately preceding their execution. They may be prefetched, and placed in the instruction queue. The Fetch stages (F0/F1) of the pipeline manage the instruction queue, including fetching and prefetching required instructions from memory. Not all instructions fetched are executed. Some may be discarded if a control transfer instruction (branch) changes the flow of execution. Up to 128 bits (four instructions) may be read from the instruction cache in every cycle. These instructions are entered into the instruction queue, and can be removed at a maximum rate of three instructions per cycle.

2.1.2. D0 (Grouping)

The D0 stage selects from 0 to 3 instructions from the instruction queue to form an in-order instruction group. This selection depends on the set of instruction *candidates* that are available at the head of the instruction queue prefetch buffer, as well as the current state of the processor pipeline. The grouping rules used to form this selection are described in section 3.4 These instructions must be taken in order from the queue, *Viking* does not execute instructions out of order.

Once a group of instructions is selected, D0 identifies the first memory reference instruction in the group, and latches the corresponding register index. D0 forms extension words based on the immediate values for memory reference and control transfer instructions' displacements. D0 identifies *cascade* conditions (integer instruction data dependencies within and between instruction groups), and inserts pipeline bubbles when necessary. A *bubble* or *pipeline stall* of *dead*

cycle is a cycle where no instruction is executed. This cycle is necessary when there is a pipeline hazard, or if required data is not available.

2.1.3. D1 (Resource Allocation)

D1 assigns available resources within the integer unit to individual instructions in the group selected during D0. All cases of data forwarding (or bypass) are resolved in this stage. All operand register index are selected and assigned to individual register file *ports*. These resources stay constant throughout the execution of the instructions.

The two address registers selected during D0 are read via two dedicated register file ports during D1. This data is used in D2 to compute a LD/ST virtual address. The data for these may also be *forwarded* from currently executing instruction groups.

Branch target addresses are generated in D1, taken from the extension words selected in D0 and the Program Counter (PC) value of the branch instruction within the group. Next PC values are also generated.

2.1.4. D2 (Read Operands)

Stage D2's primary function is to read the operand registers selected in the preceding D1 stage. In addition, the address operands read during D1 will be combined in the virtual address adder. The result is a 32-bit virtual address which will be used to reference the MMU and data cache in subsequent stages. During D2, any bypass paths required for execution will be set up to transfer data in cycles that follow.

2.1.5. E0 (Execute first stage)

Viking has two execution stages. E0 is the primary execution stage. Most arithmetic operations complete in E0. During E0, the data operands read from the register file during D2 are passed through one of two ALUs, or the shifter. Up to two integer results can be generated in E0. Only one may be generated by the shifter. These results are then presented as input to the E1 cascaded ALU, and sent into many forwarding paths.

For memory references, the virtual address generated in D2 is used in E0 to begin accessing the TLB and the data cache. Only the low-order 12 bits of the virtual address are needed to begin cache lookup. The high order bits are supplied by the MMU in the E1 phase for tag comparison with the physically cached data. The MMU must inform the data cache unit in E0 if there is an access exception in the current group of instructions. The IU is also informed in E1 stage about the access exception. If it is not yet known whether an exception must be reported to the current group (due to TLB or cache misses), the pipeline is stalled at this stage until all exception sources have been resolved.

Floating point operations are dispatched to the FPU during E0.

2.1.6. E1 (Execute second stage)

The second stage of execution can generate at most one additional integer ALU result. This result is generated in the *cascaded* ALU. The computed results from the E0 ALU or shifter are used as inputs to this ALU. All execution results from the current instruction group are available by the end of the E1 stage. This includes data returned from the data cache.

Results generated in E1 are delayed a cycle before they can be used as address operands. Address dependencies for a load from memory result in one cycle of pipeline bubble. Condition codes generated in E1 are delayed a cycle before they can be used in resolving conditional branches.

2.1.7. WB (Write Back Results)

When stage E1 has completed, all results are guaranteed to be available. The primary action in the WB pipeline stage is to 'Write Back' these results into the register files based on write enable signals generated earlier in the pipeline and potentially modified due to exceptions. The WB stage executes at the same time as the E0 stage of the next instruction group. Forwarding paths are used to transmit data between successive groups. The integer unit and FPU update the register files during WB, and normally the data cache updates its contents when a ST instruction has appeared in E0-E1.

Viking can operate in either CC mode or MBUS mode. The choice of mode has a major impact on the behavior of ST instructions. In CC mode, *Viking* assumes the existence of an external cache. In this mode, the *Viking* data cache behaves as a 'Write Through' cache, which means that all ST instructions that modify the internal cache also write their data through to the external cache.

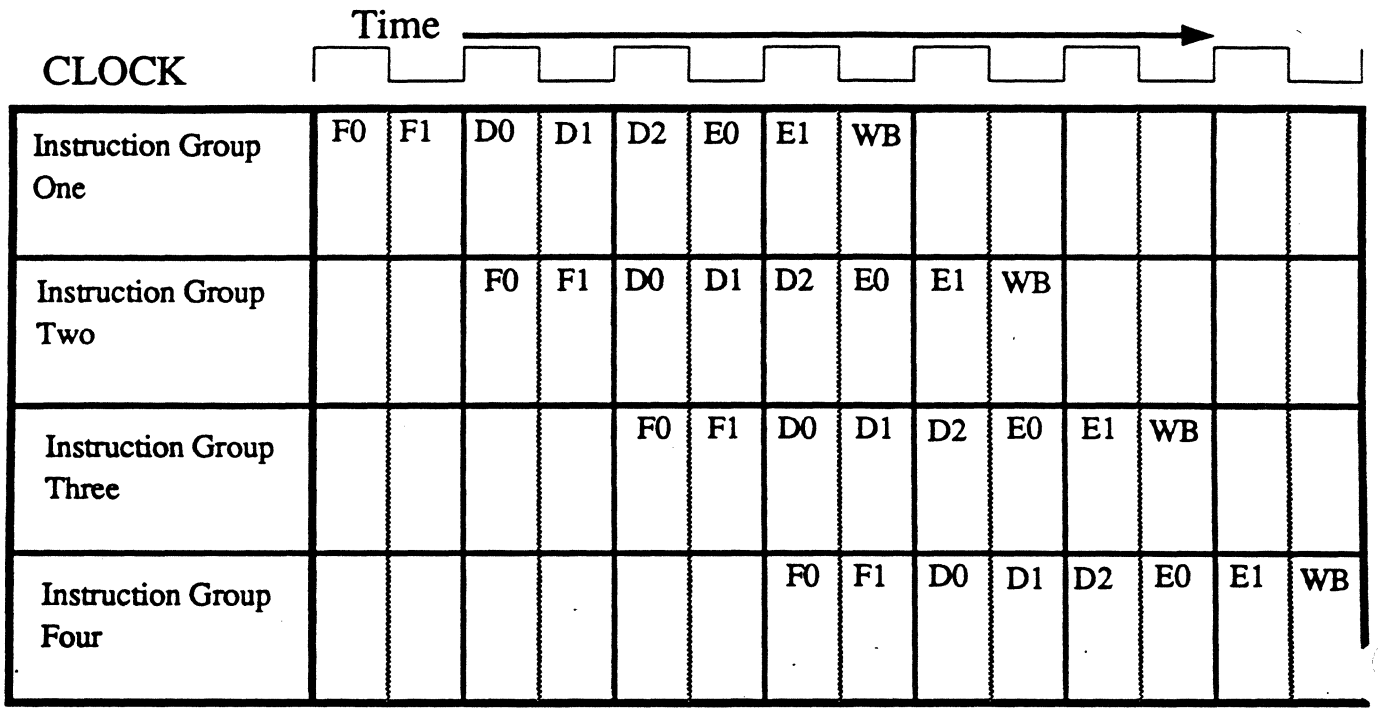
In MBUS mode, the *Viking* data cache operates as a 'Copy Back' cache, which means that ST data is not written out to the external system until the line containing the data is replaced in the cache, or a snoop on the bus forces a copy back. Also, in MBUS mode the cache implements a 'Write Allocate' policy, which means that if a ST misses in the cache, the line containing that data is brought in from memory, and then the ST is performed locally (i.e. memory does not get updated, consistent with the 'Copy Back' strategy). MBUS mode does not assume the presence of an external cache.

As in CC mode, there are conditions that will force a ST in MBUS mode to be treated as a synchronous ST.

2.2. Basic Pipeline Diagram

Throughout the document, *pipeline diagrams* are used to represent the flow of instructions and data through the processor. The most basic pipeline diagram is shown in the figure below. This diagram is generic, and is intended to show the relation of groups in the pipeline.

Figure 2-1 *Basic Pipeline Description*



All pipeline stages are identified. The bold vertical lines indicate major (rising) clock edges. The shaded vertical lines are minor (falling) clock edges. In general, the contents of the instruction group will be indicated in the left-side heading. Significant operations and interactions are included in the boxes for individual stages.

2.3. Pipeline Examples

Viking's pipeline is straightforward for simple instruction sequences (e.g., integer arithmetic). The complexity rises quickly for memory reference and control transfer instructions. This section describes these cases in detail. Standard LD and ST sequences are presented first, followed by floating point operations (fpops). Then SAVE, RESTORE, and all forms of control transfers are described. The final section describes how the pipeline deals with exceptions.

This section describes only the *simple* cases of these sequences. In particular, pipeline stalls caused by a variety of sources are not considered here. In general, pipeline bubbles or idle cycles may be injected into the pipeline at any point for a variety of reasons.

2.3.1. Memory References

LDs and STOREs are very frequent operations in SPARC code. In a typical program, as many as 30% of the instructions are loads or stores. Since *Viking* executes up to 3 instructions per cycle, it may be required to execute a memory reference nearly *every* cycle. This presents significant challenges to the processor design.

To maximize performance, *Viking* has removed restrictions that have existed in prior RISC designs. In particular, many sources of interlocks on load instructions have been removed. This allows *Viking* to execute a LD instruction, immediately followed by a *dependent* ALU instruction in the next instruction group (an ALUOP with a register dependency with the LD).

All LD and ST instructions that *hit* in the internal data cache execute in a single cycle. This includes all byte, half-word, word, and double-word references. Up to two other instructions may be included in the instruction group with the memory reference. Stores are generally buffered. In CC mode, they take a single cycle to execute whether or not they hit in the cache.

2.3.1.1 LD (Load operation)

The diagram below (load after ALUOP example) shows a LD instruction executing, surrounded by arithmetic operations. For simplicity, the sequence uses single instruction groups, forced by the dependencies in the code. The code sequence being executed demonstrates the use of many data forwarding paths for reference. The sequence is:

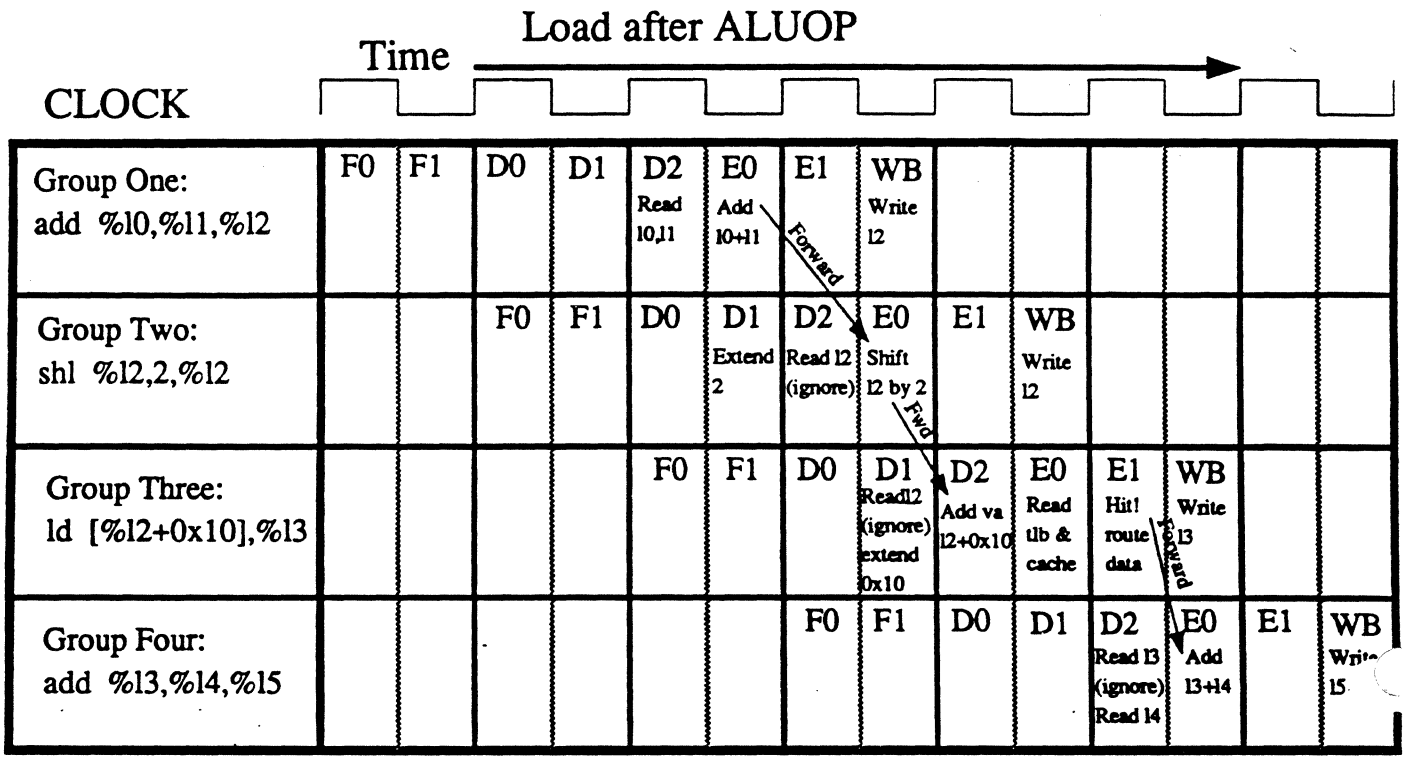
```
add %10,%11,%12
!---Break (Can't cascade into shifter)
sll %12,2,%12
!---Break (address dependency)
ld  [%12+0x10],%13
!---Break (Load data dependency)
add %13,%14,%15
```

Note the use of a dependent shift instruction to force a break between the add and shift. If the shift were replaced by an add, it would be considered a cascaded instruction and the two would be grouped together. This would have resulted in a pipeline bubble between the second add and the load, as shown in the example below. The total execution time would have been *identical*.

```
add %10,%11,%12
add %12,2,%12
!---Break (address dependency)
bubble
!---Break (address dependency)
ld  [%12+0x10],%13
!---Break (Load data dependency)
add %13,%14,%15
```

The execution of this code sequence through the pipeline is shown below:

Figure 2-2 *Basic load pipeline sequence*



The add and shift instructions execute, pass data through forwarding paths from the add result, into the shifter; then from the shifter result into the virtual address adder for the load. The "0x10" offset is extended into a 32-bit value in the D1 stage. The offset extension word, and the forwarded version of register %l2 are added, and the result passed to both the data cache and the MMU, which are accessed in parallel. When a hit is identified in the TLB, the physical page number is extracted and passed on to the data cache. In the meant time, the cache has completed reading all data and tags for the four possible sources of the memory location (4-way set associative cache). The tags are compared with the physical page number from the MMU. When the proper set is identified, the data is routed back and forwarded into the next E0 execute stage for the last add instruction, it is also written into the register file.

Had a cache or MMU miss occurred, pipeline bubbles would have been inserted. The E0 and E1 stages of the load would be repeated until all the misses had been satisfied. If any errors occurred, they would be reported to the E0 stage (which is being repeatedly executed).

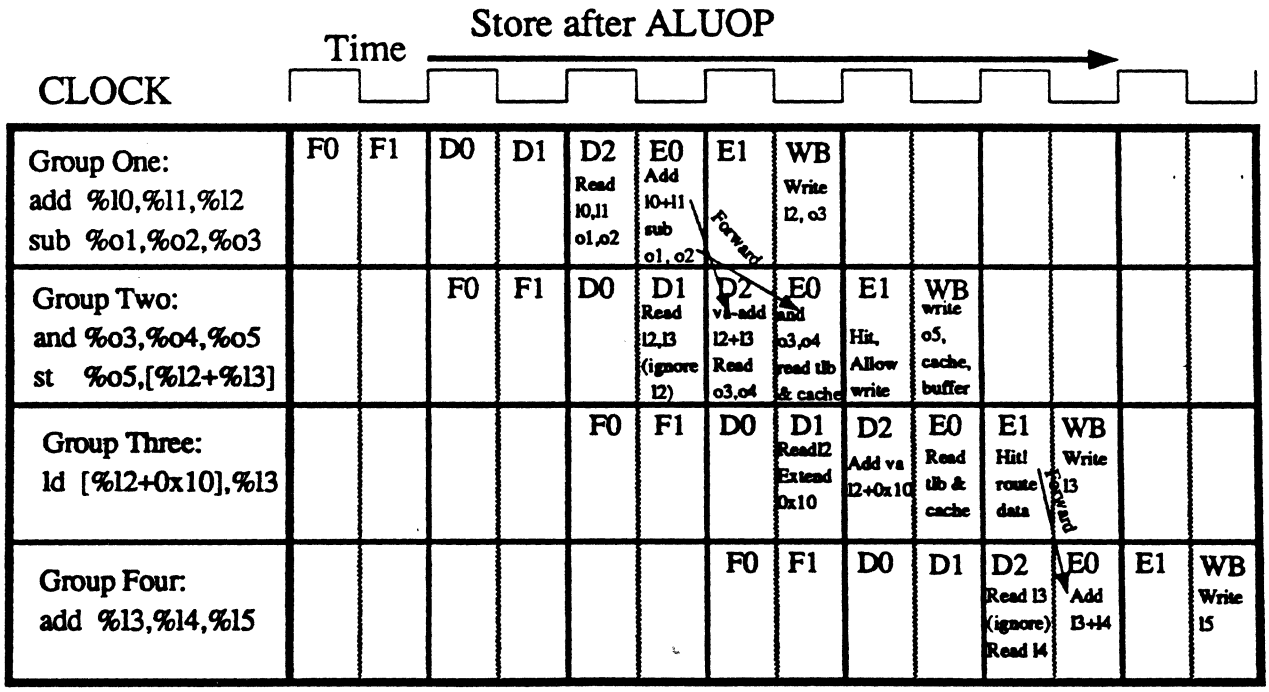
2.3.1.2 ST (Store operation)

Store operations are similar to loads in many ways. The address computation, cache lookup and MMU access are done in exactly the same manner for loads. The primary difference is the handling of the data. The sequence below demonstrates a store operation. The instruction sequence is more complex than the previous example to illustrate multiple instruction handling.

```
add %l0,%l1,%l2
sub %o1,%o2,%o3
!---Break (two write ports)
and %o3,%o4,%o5
st  %o5,[%l2+%l3]
!---Break (only one memory reference)
ld  [%l2+0x10],%l3
!---Break (load data dependency)
add %l3,%l4,%l5
```

The first and last two instructions of the sequence are identical to those in the last example (load after ALUOP). The shift from that example is replaced by an AND and ST instruction. A SUB instruction has also been added. Note that the store requires data from the and. This is executed with no delay, as the data to be stored is not required until the stage when it is *actually* written out.

Figure 2-3 *Store Pipeline Operation*



This diagram is very similar to the previous example, with the addition of another memory reference instruction. Many of the forwarding paths used in the previous diagram are no longer used, while others are illustrated. The store address is computed in the D2 stage, then used to check the MMU and cache tags in E0/E1. Assuming all protection checks pass, the write is actually performed during WB. The physical write to the data cache is delayed another phase *past* WB, but this is not visible to the program execution.

Operation of the data cache on a ST miss depends on whether *Viking* is operating in CC or MBUS mode. In CC mode, if the store operation had missed the data cache, the timing would be identical. The store data would be written *only* to the store buffer, and not to the data cache. In this case *Viking*'s data cache does not write allocate on misses. In MBUS mode, however, if the store operation had missed the data cache, *Viking*'s data cache would write allocate, by bringing the data from memory, retrying the ST (which would now hit), then writing the new data onto the cache line. This new data would be copied back to memory when the cache line is to be replaced.

In some cases, a load needs to read data which has been written recently. If this data is in the cache, it is returned immediately. In CC mode, this data may not be in the cache, since it may still be in transit to the external cache, or to main memory. The drain rate of the store buffer depends on the external system. In such cases, the store buffer is checked for a copy of the needed data (store buffer snooping). If it is found to be present, the processor requests that the store buffer be drained (store buffer copy out). The processor then waits until the requested data is no longer in the store buffer, then continue and read the data *back* in from memory, as for a normal load. *Viking* does not forward data from the store buffer to satisfy read requests. In MBUS mode, a ST guarantees that the data cache has the new data. See section 8 — *MBUS Interface* for more details.

2.3.2. Floating Point Pipeline

Viking's on chip floating point is tightly coupled to the integer pipeline. A floating point operation may be started every cycle. The latency of most floating point instructions is three cycles. The FPU pipeline has the following stages:

FD, FRD, FE, FL, FWB
(Decode, Read, Execution, Last, WriteBack)

Floating point instructions are dispatched *late* in the processor pipeline. They are not issued to the FPU until the E0 stage of the integer pipeline. Once issued, the floating point instruction proceed through the FPU's pipeline. The stages in the FPU pipeline are also fairly standard: decode, read, execute, and writeback. Forwarding paths are provided to chain the result of one floating point operation to a source of a subsequent operation.

The floating point pipeline becomes visible to the integer pipeline in several cases. When an FBFCC (Branch on floating point condition codes) instruction is issued, the processor may need to wait until a preceding FCMP instruction has completed. When a floating point store instruction is executed, the pipeline also becomes visible. The integer pipeline waits in the E0/E1 stage until the requested data is available from the FPU. In some cases, the floating point queue may become full, typically when many long latency floating point instructions

(divide, square root, or highly dependent operations) are issued.

Since floating point instructions are issued late in the pipeline (E0), and the actual arithmetic is not begun until one cycle later (WB), *Viking* may issue a load and a dependent FPOP simultaneously. This is demonstrated in the following code fragment. A pipeline diagram of a simpler case is shown below.

```
ldd     [%l0],%f2
faddd   %f2,%f0,%f6
add     %l0,0x8,%l0
!--- Break (Three instruction max)
ldd     [%l0],%f4
add     %l0,0x4,%l0
fmuld   %f4,%f0,%f8
!--- Break (Three instruction max)
ldd     [%l0+4],%f10
cmp     %l0,0x100
be      Loop
!--- Break (Branch, Three instructions)
faddd   %f6,%f8,%f0
.
.
```

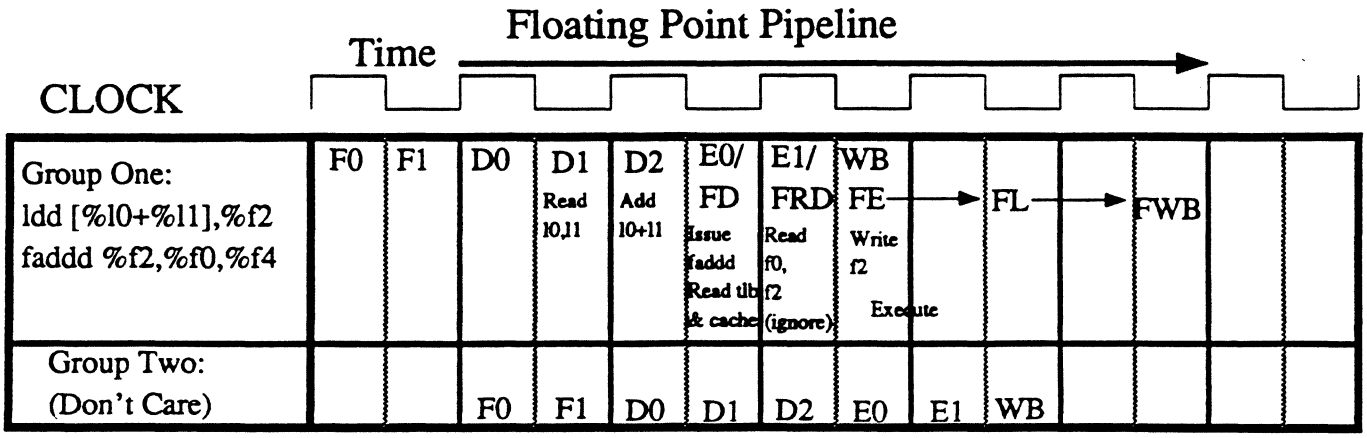
The example above may not contain a particularly interesting sequence, but it shows many of *Viking*'s strengths. All but the last group contain three instructions. The floating point operations are grouped with load instructions on which they depend. The data returns from *Viking*'s data cache at the end of the E1 stage, and is immediately used by the Floating Point Unit's FRD stage, and then by its FE stage.

The pipeline diagram below shows the following code fragment being executed:

```
ldd     [%l0+%l1],%f2
faddd   %f2,%f0,%f4
!--- Break
```

by its For simplicity, other instructions in the pipeline are not shown.

Figure 2-4 *Floating Point Pipeline*



2.3.3. Floating Point Instructions

There are two types of floating point instructions:

FPOPs (floating point operations) and
FPEVs (floating point events).

The FPEVs (floating point events) are executed by the FPU but they do not enter the FPU queue, and these include:

LDF/STF (load and store to floating point registers)
LDFSR/STFSR (load and store to floating point status register)
STDFQ
Integer Multiply and Integer Divide operations

The FPOPs include all *falus*, *fbfcc* etc., and they specifically enter the FPU queue.

2.3.4. Floating Point Queue

The FPU queue is a FIFO, holds up to 4 FPOPs, stalls the execution stream when the it is full, and signals FSR.QNE to indicate whether/not it is empty. FPEVs do not enter the FPU queue, these operations use the FPEV logical port. Recall that IMUL IDIV are included in FPEV. The FPOPs use the FPOP port.

IMUL and IDIV can start execution only when the FPU queue is empty, unless the FPU is in exception mode (See section 4.4.1 — *Integer Multiply (IMUL)* and 4.4.2 — *Integer Divide (IDIV)* for more details).

2.3.5. Floating Point Execution Times

Different FP operations require different number of cycles to complete. The following table summarizes the performance of *Viking* FPU for its FP operations. *falu* includes fadd, fsub, fmov, fneg, fabs, fcmp, and all of the conversions between integer, single and double precision numbers. Single is denoted "s", it means single precision operation carried on 32 bits. Double is denoted "d", it means double precision operation carried on 64 bits. Viking supports neither

extended "x" precision (80 bits) nor quad "q" precision (128 bits). fmul is f-multiply, fdiv is f-divide, fsqrt is f-square-root, each of them allowing either single or double precision operation. fsmuld is multiply a single precision number to a double precision number. imul is integer multiply and idiv is integer divide. imulcc is integer multiply that affects the condition codes *icc*, and idivcc is integer divide that affects the condition codes *icc*. These operations as stated earlier are performed by the FPU. Each of these FPOPs is a SPARC instruction, only the implementation is Viking specific.

The following table lists FPOPs with a 3 cycle latency.

Table 2-1 *Floating Point Operation Execution Time - 3 cycle latency*

Instruction	cycles
fadds	3
faddd	3
fsubs	3
fsubd	3
fcmps	3
fcmpd	3
fcmpes	3
fcmped	3
fstoi	3
fstod	3
fdtoi	3
fdtos	3
fitos	3
fitod	3
fmovs	3
fabss	3
fnegs	3

The following table lists other FPOPs and their latencies. The execution time of each of these FPOP depends on the source(s) and destination data formats. The notation used in this table designates:

n: floating point normal number.

s: floating point subnormal number.

ns=n Identifies source 1, source 2, and destination data formats respectively.

Table 2-2 *Floating Point Operation Execution Time - latency*

Instruction	nn=n	nn=s	sn=n	sn=s	ns=n	ns=s	ss=n	ss=s
fdivd	7	8	10	11	11	-	11	-
fdivs	6	7	9	10	10	-	10	-
fmuld	3	4	6	7	7	8	8	-
fmuls	3	4	6	7	7	8	8	-
fsqrtd	10	-	13	-	-	-	-	-
fsqrts	8	-	11	-	-	-	-	-
fsmuld	3	-	6	-	7	-	8	-
idiv	15	-	-	-	-	-	-	-
idivcc	15	-	-	-	-	-	-	-
imul	4	-	-	-	-	-	-	-
imulcc	4	-	-	-	-	-	-	-

2.3.6. Conditional Branch Pipeline

Branches are a fundamental performance limiter in most processors. This is particularly true in RISC machines, where basic block sizes are usually small. It is a serious problem in superscalar machines, which reduce the number of cycles taken to execute the *already* small number of instructions between branches. In normal operation, *Viking* could see a branch every three to five cycles.

Viking's branch implementation can execute *untaken* branches somewhat more efficiently than *taken* branches. The peak performance through an untaken branch is 3 instructions per cycle. Peak performance through a taken branch is 2.3 instructions per cycle. The difference comes from the delayed branch instruction which must be executed as a single instruction group in the taken case. Due to usual code scheduling restrictions, this peak performance will not generally be attained, and the difference between taken and untaken branches will be less visible.

Viking implements branch prediction in the prefetch logic. It always attempts to fetch the target of the branch. However, the pipeline assumes that the branch is untaken. This relies on the instruction queue having several *sequential* instructions available. The target instructions are fetched into the target queue. Once the direction of the branch has been resolved, the appropriate instructions are routed into the instruction queue.

When a branch occurs, the pipeline continues to execute normally for one cycle. In this additional cycle, the *delay* instruction is executed, possibly along with other instructions in the untaken stream. This implies that, if the branch is taken, some instructions have begun execution that *should not have*. *Viking* terminates the execution of these instructions for a taken branch as soon as the branch sequence has been resolved. This is called *squash*. The instruction must be canceled before any machine state has been modified.

Viking executes the typical branch sequence by grouping together the branch instruction, and the *previous* instruction(s) that normally set the condition codes. This is a form of *branch folding*. Up to two other instructions may be executed in the group with the branch. The delay instruction is executed in the next group. A typical instruction sequence might be:

```
            subcc   %l0,0x100,%g0
            bz      TakenTarg
            add     %l1,%l2,%l3
            st      %l3,[%g1+0x100]
            and     %o0,%o1,%o2
            .

TakenTarg:  st      %g0,[%g1+0x100]
            .
```

This same basic code sequence is used to demonstrate both taken and untaken branches. The same basic control transfer mechanism is used for all forms of branches, jumps, and calls.

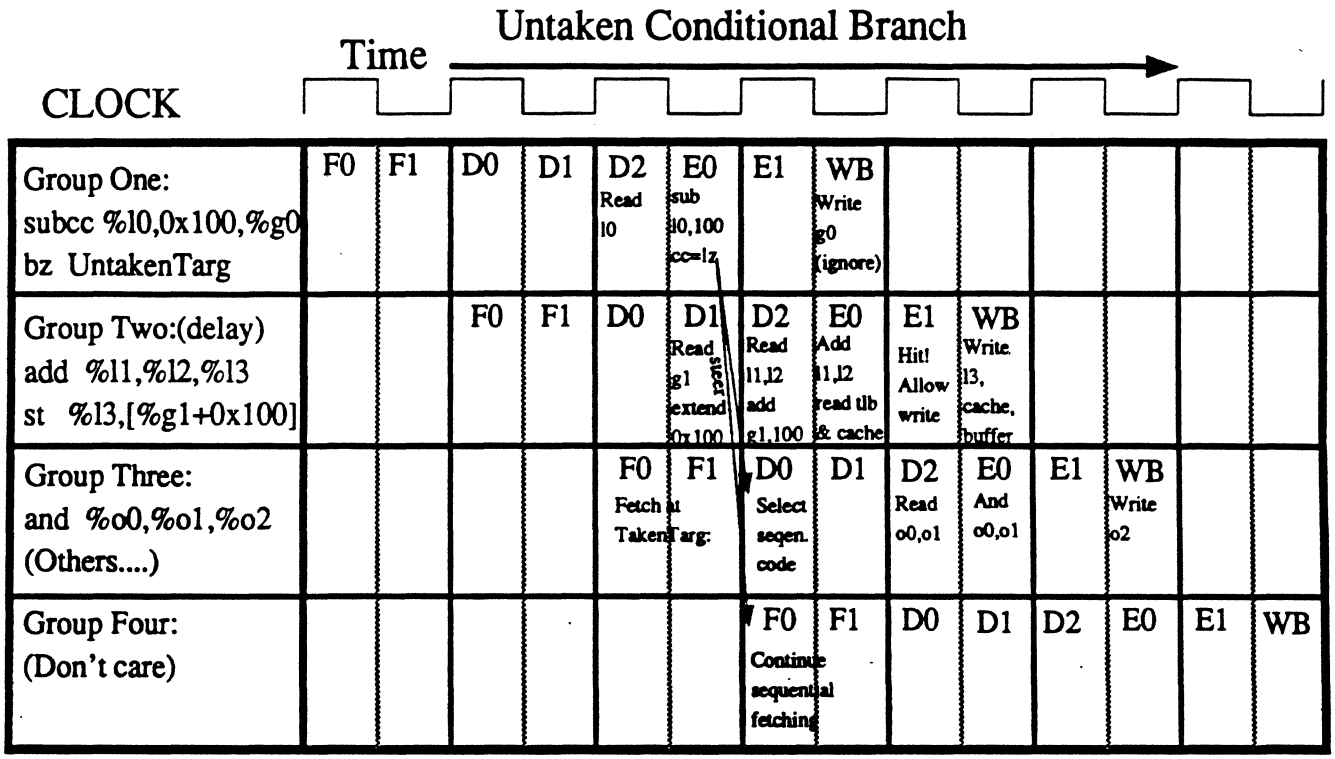
2.3.6.1 Untaken Branch

The untaken branch is the simplest case. The example sequence above is grouped as follows in the untaken case:

```
            subcc   %l0,0x100,%g0
            bz      UntakenTarg
            !---Break (Break after CTI)
            add     %l1,%l2,%l3
            st      %l3,[%g1+0x100]
            !---Break (No more write ports)
            and     %o0,%o1,%o2
            (others)
            .
```

The instructions at *UntakenTarg* will not be executed, since the branch is not taken. Notice that two instructions (add, store) are executed in the *delayed instruction group*. The instructions beyond the third group are not significant. The pipeline behavior is shown below:

Figure 2-5 *Untaken Branch Pipeline*



The diagram shows the fetch that is being done because the branch is assumed to be taken. The condition codes (from the subcc instruction) are resolved in the branch groups E0 pipeline stage. This condition code is used immediately to determine the branch direction, in the delay groups D2 stage, the *target* group's D0 stage (by selecting the correct queue), and the *following* group's F0 stage (by selecting the correct prefetch program counter). Thus, after one cycle of uncertainty, the branch is resolved. Fetches once again follow the correct execution stream.

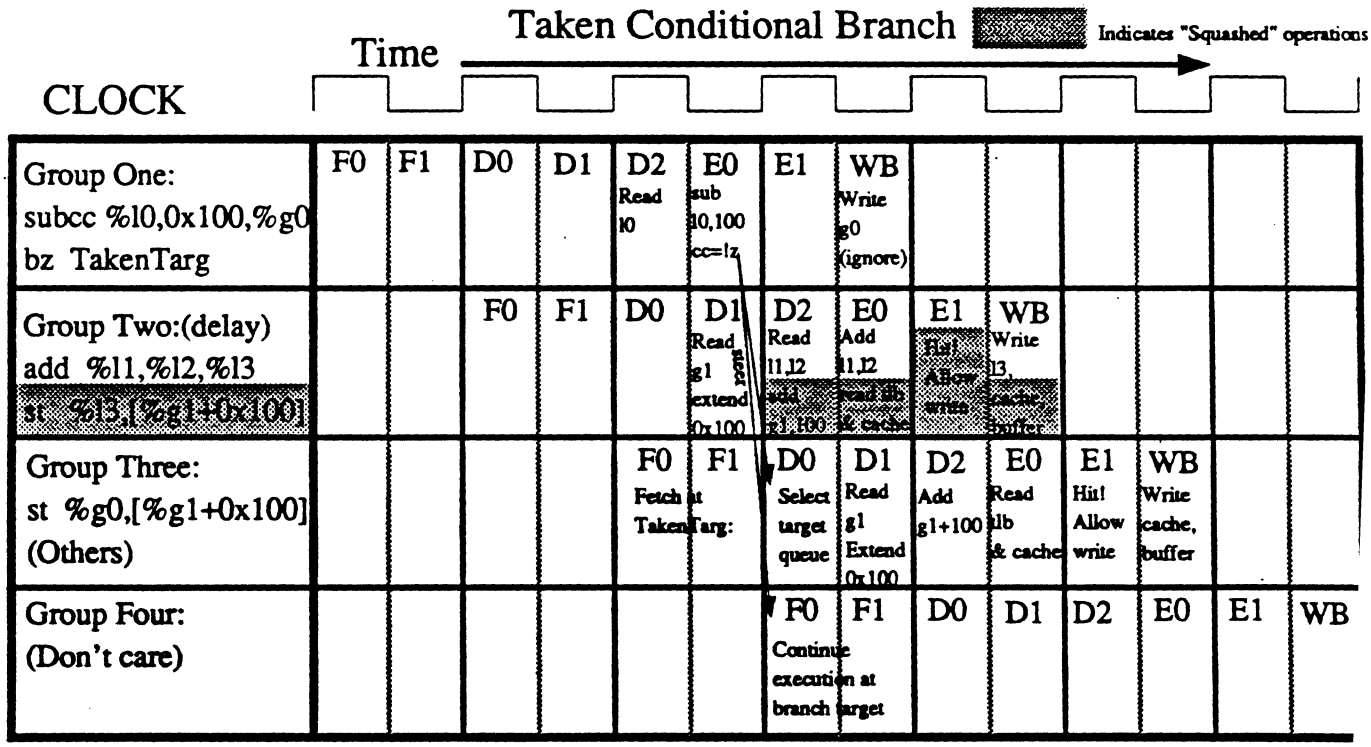
2.3.6.2 Taken Branch

The taken branch case is somewhat more complicated. The most significant change is that instructions which *begin* execution in the delayed instruction group are "squashed", and never *complete* execution. Squashed instructions are indicated with gray shading in the pipeline diagrams. The apparent grouping of instructions in the taken case is:

```
            subcc   %l0,0x100,%g0
            bz      TakenTarg
            !---Break (Break after CTI)
            add     %l1,%l2,%l3
            !---Break (Squash: st %l3,[%g1+0x100])
TakenTarg:  st      %g0,[%g1+0x100]
            .
            !---
            .
```

The instruction grouping is the same as for the untaken branch case, up to the delay group. In the delay group, the grouping begins in the same as for an untaken branch, but the grouping is modified (squashed) when the branch is resolved as taken.
The taken branch pipeline is:

Figure 2-6 *Taken Branch Pipeline*



2.3.6.3 Branch Couple

SPARC allows for a limited set of CTI (Control Transfer Instruction) *couples*. These cases are quite complex to understand and implement. They execute in the same *general* manner as normal branches. *Viking* executes all the legal CTI couples correctly and, conforming to SPARC V.8, choose not to support the implementation dependent case of B*CC and DCTI couple (Case 6 in the SPARC V.8

Manual). The B*CC instructions are Bicc and FBfcc (including BN, FBN, but excluding B*A and FBA). DCTI instructions are CALL, JMPL, RETT, B*CC taken, and B*A (with a=0).

2.3.6.4 JMPL

The JMPL instruction is an unconditional branch which uses a register as the source for the branch target. Other branch instructions compute the target address from an immediate operand and the current program counter value. JMPL's suffer a delay since they must read the register file before issuing a branch target fetch. The extra cycle is injected into the pipeline *after* the delay instruction of the JMPL, that is, before execution of the target instruction.

2.3.7. Procedure Call and Return

SPARC defines a set of instructions for procedure entry and exit. CALL is used to branch to a subroutine. It saves the PC value in register %o7 which is used to compute the return address. JMPL is used to do a *return*, using a register to supply the target of the branch. By convention, for a return, JMPL uses the register written by the last previous CALL.

SAVE and RESTORE instructions work as a pair to allocate and deallocate a register window and a stack frame for a procedure. SAVE decrements the CWP, while RESTORE increments it. Both instructions do an implicit ADD operation, which is used to allocate/deallocate the stack frame.

2.3.7.1 CALL

CALL executes in exactly the same manner as a taken branch, including execution of the delay instruction, except that the current PC value is written into register r15 (%o7). The value is written in the E0 stage of the group containing the CALL instruction.

CALL is grouped with previous instructions in the same manner as branches. It does, however, require a register file write port, so the grouping may be somewhat constrained by available resources.

2.3.7.2 CWP Pipeline

SAVE and RESTORE instructions modify the processor's CWP. This causes a register window to be allocated, or deallocated. These instructions also perform an ADD operation. The sources for this add are from the *old* register window, while the destination is in the *new* window. For simplicity, *Viking* executes both these instructions as single instruction groups. To make instructions before and after SAVE and RESTORE operations reference the correct windows, multiple CWP's are maintained in the pipeline. Depending on an instruction's position relative to the CWP change, the appropriate CWP value is chosen.

2.3.7.3 SAVE

SAVE is always a single instruction group. Except for changing of CWP during the decode phases, it is identical to an ADD instruction. SAVE takes a single cycle to execute.

2.3.7.4 RESTORE

RESTORE is also a single instruction group. It also behaves just as an *add* instruction, except for the CWP update. The RESTORE takes a single cycle to execute.

2.3.8. Exceptions

Exception handling is an extremely complicated part of *Viking*. The precise exception architecture of SPARC must be maintained. The most difficult exceptions to handle are those which cause *partial execution* of a group of instructions, e.g.,

```
!--- Break (Start of group)
add     %l1,%l2,%l3
ldd     [%o0+0x100],%f2
and     %l3,0x20,%l3
!--- Break (3 instructions)
```

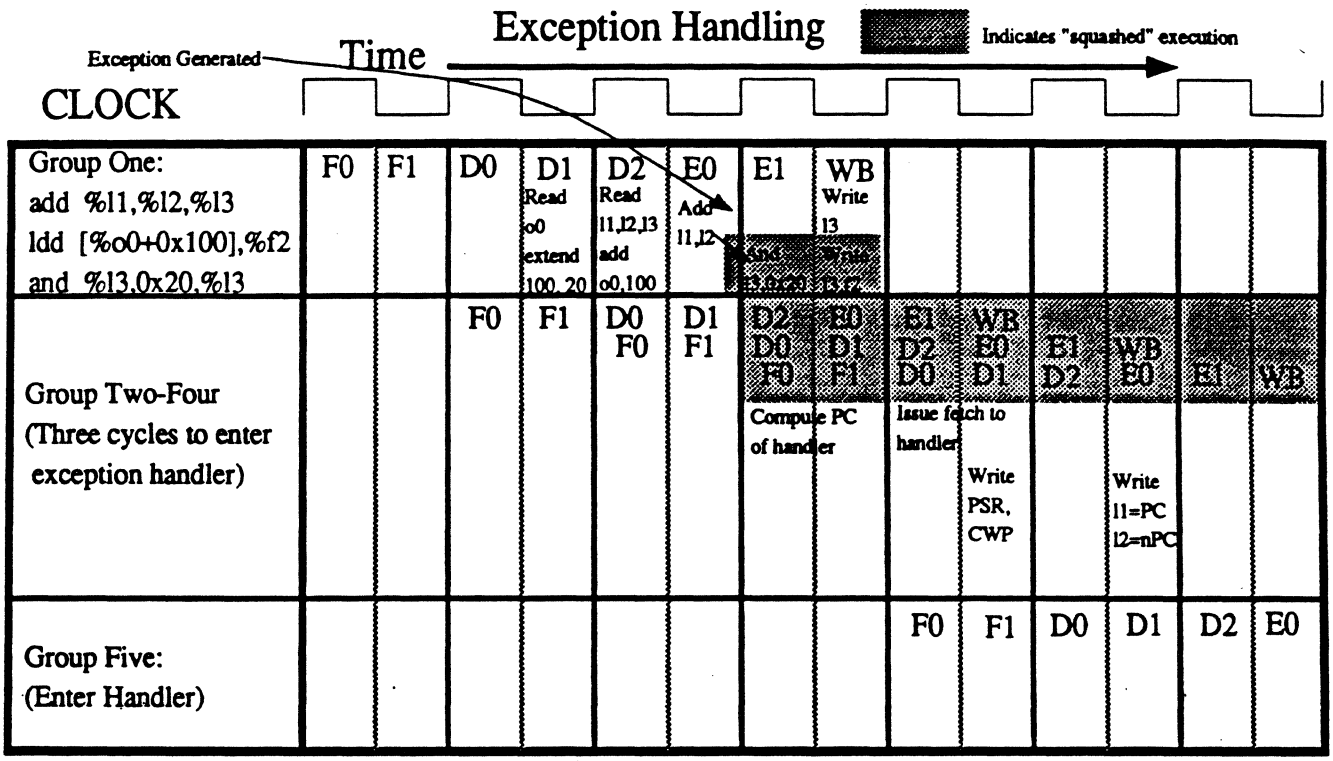
The two arithmetic instructions form a *cascade*. The load instruction (which happens to be to floating point registers) is *between* the two arithmetic instructions. Also, the destination registers of the two arithmetic instructions are identical. The problem arises in this case if the load instruction induces an exception (say, due to a page protection violation, or a misaligned memory address). In this case, the *first* instruction (ADD) must complete and write back its result. The *second* instruction (LDD), however, must not complete.

If the instruction were scheduled to complete without incurring an exception, the result of the *add* would not have been written to the register file - it would simply be used as a temporary value passed on to the *and*. When the exception occurs, this is no longer true and the result must be written to the register file. *Viking* correctly handles complex cases such as this and many others.

2.3.8.1 Exception Pipeline

The case above, with some surrounding instructions, is used below to illustrate the exception handling pipeline. The diagram assumes that an exception is reported for the load double instruction. The exception forces the write enable of the current group to be modified, allowing only the first add instruction to write its result. The exception handling logic then takes control of the pipeline. All subsequent instructions in the pipeline are *squashed*. The store buffer is flushed, and the pipeline stalls until the flush is complete. In the next cycles, the PSR and CWP are modified to reflect the exception state of the processor. An instruction fetch to the proper interrupt handler in the trap table is requested. The exception PC and nPC are written into r17 and r18 (%l1,%l2). Finally, the instructions in the handler begin execution.

Figure 2-7 *Exception Handling Pipeline*



2.3.8.2 Interrupts

Interrupts are handled in the same manner as exceptions. The interrupt request pins are sampled on the rising edge of the clock. In the next cycle, the highest priority interrupt is selected. This may be from the pins, or from an internally generated interrupt (breakpoints). The request level of the selected interrupt is compared against the enable and processor interrupt level fields in the PSR (PSR_ET and PSR_PIL). If the interrupt request level (IRL) is higher than PSR_PIL, an exception is generated in the next cycle, to the instruction group at the E0 pipeline stage.

In order for the interrupt to be accepted, there *must* be a valid instruction in the E0 stage. If there is no valid instruction, the interrupt is not taken until an instruction arrives at E0. This ensures that there is a valid PC to report as the interrupted program counter.

The *last valid* instruction in the E0 stage is normally squashed. If there is only a single instruction in E0, it is squashed, and the saved PC value points to that instruction. All instructions in the pipeline after this squashed instruction are also squashed. Breakpoints are reported to the instruction that caused the breakpoint detection, rather than the last valid instruction.

The interrupt pipeline is identical to the exception pipeline (see above). It behaves just as if an exception were reported to the last valid instruction in the E0 group. See section 7.4 — *Interrupts* for more details.

2.3.8.3 RETT (Return From Trap) Pipeline

The return from trap instruction executes in cooperation with a JMPL that must immediately precede it. (See the SPARC Architecture Manual). Execution of a RETT is very similar to a JMPL. It is required to restart instruction sequences which were trapped on the delay instruction of a branch. In this case, the JMPL returns to the delay instruction and the RETT returns to the target of the branch. If there was no previous branch, NPC is set to PC+4.

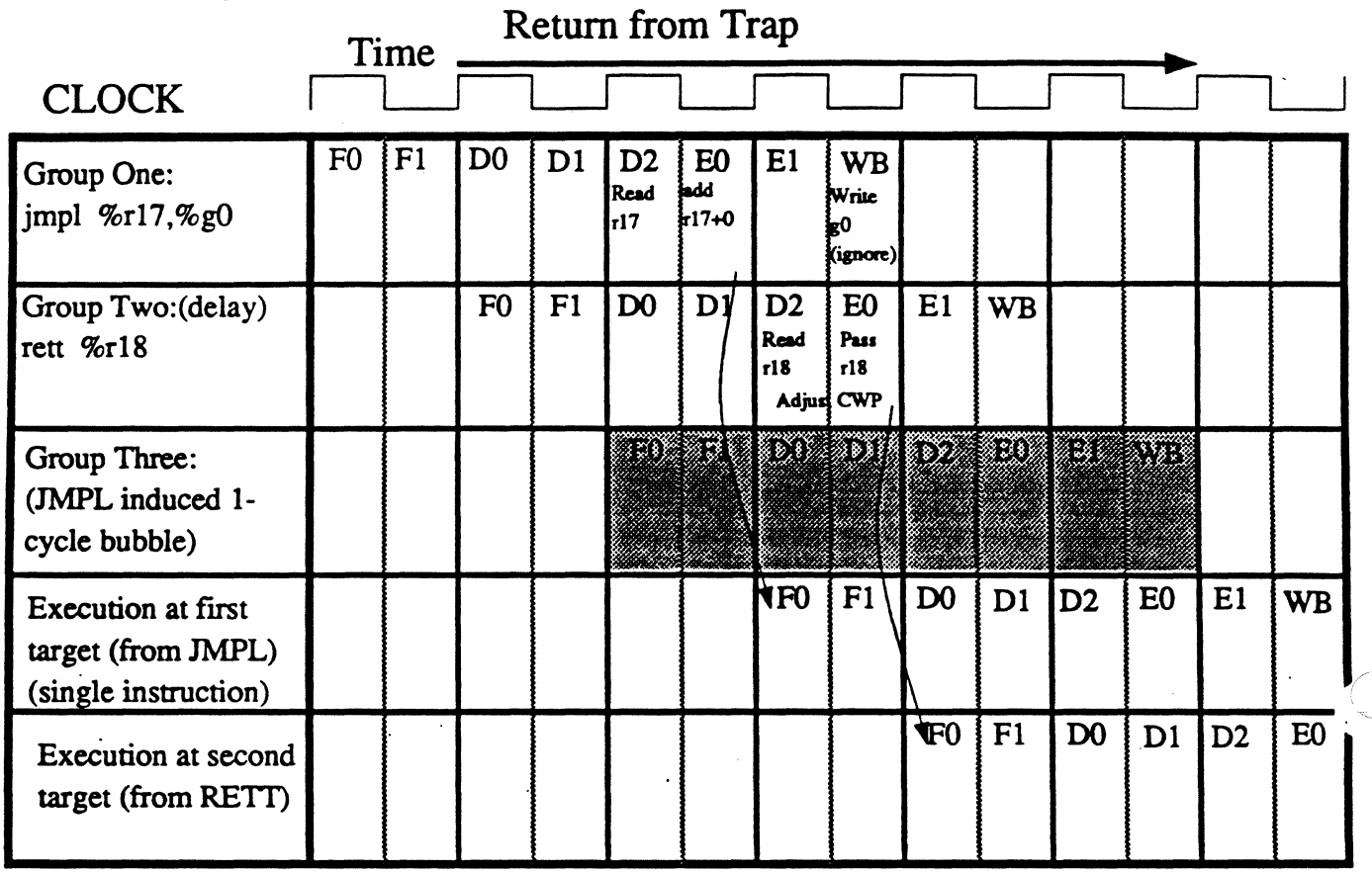
A RETT executes as a single instruction group. It incurs no additional pipeline cycles. The preceding JMPL provides a *free* pipeline hold cycle for the RETT to change CWP and PSR, and to issue a fetch to the saved NPC.

The typical code sequence used to return from a trap is as follows:

```
jmpl    %r17,%g0
rett    %r18
```

The instructions at the target of this return are *don't care*'s. The instruction at the target of the JMPL *must* be executed as a single instruction group, since it may not be contiguous with the return address in the RETT. The pipeline operation for the sequence is shown below:

Figure 2-8 *Return from Trap Pipeline*



3

Code Generation Principles

3

Code Generation Principles

All μprocessors benefit from code sequences generated with intimate knowledge of the processor's operation.

This section describes some guidelines that can be used to construct optimal, or nearly optimal code sequences for *Viking*. They can be applied to compiler code generation, as well as manually generated code for highly performance sensitive routines.

3.1. Performance of Existing Code

Viking is designed to execute code from existing SPARC compilers efficiently. In general, *Viking* executes this code at about 1.35 *instructions per cycle* (IPC), or 0.74 cycles per instruction (CPI). This number is derated to about 1.1 IPC for *average* large programs when cache effects are considered. Floating point intensive programs typically execute at a higher IPC.

This performance level can be increased significantly with code generated explicitly for *Viking*. In addition, compiler improvements not targeted explicitly at *Viking* also offer substantial improvements. This text considers only *low-level* code scheduling issues.

3.2. Areas that Hurt Performance

Viking performance is sensitive to many factors. The significant performance limiters vary greatly between programs. Classes of limiters include: branch frequency and direction; memory reference patterns; floating point operation scheduling; instruction ordering and data (register) dependencies.

In floating point programs, branch performance is rarely a limiter. This is mostly because loops are often *unrolled*. In most cases, floating point programs are *memory reference* limited, rather than arithmetic limited. Since only a single memory reference can be executed per instruction group, interleaving memory references with nearly anything else improves performance.

Most programs are limited by either branch performance or load/store bandwidth. In particular, taken branches can only execute a *single* instruction in the branch delay group. Load and store operations are often "bunched" together. This prevents *other* instructions from being executed in parallel with the memory references.

Cache performance is also critical to achieving maximum performance. Many routines are small enough to have their entire code and data set contained in *Viking*'s on-chip caches. Only (usually) insignificant cold-start penalties will

degrade the performance of such routines. Most larger programs, however, do not fit entirely in the on-chip caches. These programs typically lose 20% to 30% of their performance from cache miss penalties. Prefetching logic serves to limit the degradation in some cases. Localizing code and data references within a program can lead to substantial performance improvements.

3.3. General Guidelines

A *general* rule for *Viking* optimization is to interleave as many different classes of operations as possible. A good simple measure of the minimum execution time of any routine is:

$$MIN\ Cycles = MAX\begin{cases} MemoryReferences \\ FloatingPointOperations \\ \dfrac{IntegerOperations}{2} \\ BranchOperations \times 2 \end{cases}$$

The rule holds when one, and possibly two of the terms are close to the maximum. As more terms get larger, the likelihood of approaching the minimum number of cycles decreases.

In order to approach the minimum above, all of the classes of operations must be interleaved as much as possible. In the worst case, the equation can become:

$$Cycles = Memory\ References + FloatingPointOperations + (\frac{IntegerOperations}{2}) + (BranchOperations \times 2)$$

Viking hardware tends to locally compensate for minor scheduling variations. As an example, the sequence

```
ldd [%l0],%g1
!--- Break (only one memory reference)
ldd [%l1],%g4
add %g1,0x100,%g3
!--- Break
```

executes in the same number of cycles as the following sequence:

```
ldd [%l0],%g1
!--- Break (dependent Ld-Use)
add %g1,0x100,%g3
ldd [%l1],%g4
```

This is a trivial example, but it demonstrates the *Self-Aligning* nature of execution on *Viking*. The pipeline tends to align itself based on the positions of the critical operations in the code (the memory references, FPOP's, and branches). Local variations rarely affect performance.

Again, the important consideration is that the critical operations do not become additive, but rather are interleaved to increase parallel operation. Along these lines, most codes have *many* optimal schedules.

3.3.1. Reduce Branches

In the equation above, each branch increases the execution time of a sequence by about two cycles. Thus, it is far more significant to remove a branch than to remove a memory reference.

Code should be *unrolled* wherever possible. This is a performance boost on all machines, and especially on *Viking*. The addition of as many as four other instructions is generally better than a single branch. Where possible, arithmetic logic, rather than sequences of branches will offer improved performance.

Viking will benefit from unrolling all types of code, not just floating point.

3.3.2. Allocate delay instructions carefully

Since the delay instruction of a taken branch is forced to be a single instruction group, it is very important to make the best use of that instruction. For example, in the unrolled LINPACK case, optimal performance cannot be achieved unless a memory reference is placed in the delay instruction.

The guideline, is to make certain that the delay instruction is used to execute one of the critical performance limiting operations in the code.

Reorganizing code to properly fill the delay instruction may require *adding instructions* to the code. This is a tradeoff that must be made carefully. Even though adding instructions can increase *Viking*'s performance, it is guaranteed to *decrease* performance on any non-superscalar machine. In addition, it does expand the code space. One alternative is to use *Software Pipelining* which often allows the same performance levels without increasing the number of instructions. This is accomplished by spreading the execution of each loop iteration over several actual trips through the code. The resulting code in the loop would execute the final operations of the *previous* loop, the core of the *current* loop, and the prologue of the *next* loop.

Annulled Branches should not be used unless there is no alternative. An annulled branch saves only the code space used for the delay instruction. The cycle in which the annulled branch would have been executed is still required.

3.3.3. Reduce floating point register dependencies

This can be a very significant performance limiter in some benchmarks. No floating point register can be modified until all pending operations in the floating point queue which either *produce* or simply *use* that register have completed. This is often violated in an attempt to reduce temporary FP register usage. It may, in fact, increase *Viking* performance to use more loads and stores to move floating point data back and forth between memory and registers, than to overuse a single register and introduce dependencies.

This is true of arithmetic operations, as well as loads and stores. The Floating Point Unit has a fairly long *execution* pipeline. Performance suffers when there are frequent FP data dependencies.

These guidelines do not apply to integer operations. Integer register dependencies are handled much more effectively. The following two code fragments show good and bad examples of FP register dependencies:

Bad Example

```
ld      [%i0],%f1
!--- Break (Only one memory reference)
ld      [%i0 + 4],%f2
fmuls   %f0,%f1,%f0
!--- Break (Only one memory reference)
ld      [%i0 + 8],%f1
fadds   %f0,%f1,%f2
!--- Break
!--- PIPELINE STALL FOR 4 CYCLES!
```

Note that the last load operation reuses register %f1. This prevents the load operation from being executed until *after* the FMULS has completed. Also the FADDS uses %f0 as source operand, and can only start execution when FMULS has produced a result for %f0. A much better schedule is:

Good Example

```
ld      [%i0],%f1
!--- Break (Only one memory reference)
ld      [%i0 + 4],%f2
fmuls   %f0,%f1,%f0
!--- Break (Only one memory reference)
ld      [%i0 + 8],%f3
fadds   %f3,%f1,%f2
!--- Break
```

By changing the register used in the last load and add operations, the four pipeline bubbles are removed completely. Of course, there are other optimizations that *might* be applied to this code, namely, using a load double to replace the first two memory references. Such optimizations can sometimes violate high-level language execution rules if not done carefully. In the example above, a load double would only be valid when targeted to an even-odd register pair, not odd-even pair.

3.3.4. More floating point code examples

The following section provide examples of floating point codes that illustrate the many *Viking* FPU operation details. Each example is preceded by a brief explanation about the significant details.

3.3.4.1 FP code example: Throughput

The following example shows how to take full advantage of FPOP latency (for example 3 cycle latency of fadds), and execute each FPOP without any pipeline stall. This arrangement achieves the highest throughput.

```
fadds   %f0,%f1,%f20
!--- Break ---------
fadds   %f2,%f3,%f21
!--- Break ---------
fadds   %f4,%f5,%f22
!--- Break ---------
fadds   %f6,%f7,%f23
st  %f20,[%l1]
!--- Break ---------
fadds   %f8,%f9,%f24
...
```

3.3.4.2 FP code example: One Stall

The following example is similar to the previous code (section 3.3.4.1 — *FP code example: Throughput*), except that the ST is attempted a cycle earlier. Because of the 3 cycle required latency of fadds, a bubble will be inserted to stall the pipeline before the ST can be completed. *Viking*'s grouping logic places "fadds %f4,%f5,%f22" together with ST in the same group, (this is done prior to the FPU detecting any dependency), hence when ST is stalled, the group is as well. Since the pipeline will stall in either case, there is no performance difference.

```
fadds   f0,%f1,%f21
!--- Break ---------
fadds   %f2,%f3,%f22
!--- Break ---------
fadds   %f4,%f5,%f23    (issued)
st      %f21,[%l1]      (issued)
(bubble)
!--- Break ---------
fadds   %f4,%f5,%f23    (completed)
st      %f21,[%l1]      (completed)
!--- Break ---------
...
```

3.3.4.3 FP code example: fcmp/fbfcc latency

The following example demonstrates the latency of an fcmp-fbfcc pair. An fcmp requires 3 cycles before the fcc (condition codes) are resolved for an fbfcc use. Arranging the code as given in this example allows each cycle to complete without any pipeline stall.

```
!--- Break ---------
fcmps    %f6,%f7
!--- Break ---------
fadds    %f0,%f1,%f21
!--- Break ---------
fadds    %f2,%f3,%f22
!--- Break ---------
fadds    %f4,%f5,%f23
!--- Break ---------
fbne     t1
!--- Break ---------
```

3.3.4.4 FP code example: fcmp/fbfcc Stall

The FPU detects if there is a pending fcmp, and stalls the pipeline if it encounters an FPOP that may need fcmp's resolution. In the following example, an fbfcc is issued immediately after an fcmp. *Viking* issues the fcmps, but then stalls the pipeline for 3 cycles until the fcmp has completed, before executing the fbne.

```
!--- Break ---------
fcmps    %f6,%f7      (issued)
(bubble)
!--- Break ---------
(bubble)
!--- Break ---------
(bubble)
!--- Break ---------
fcmps    %f6,%f7      (completed)
!--- Break ---------
fbne     t1
!--- Break ---------
```

3.3.4.5 FP code example: freg dependency +out -> ST

The following example shows how a floating point register dependency with a ST may stall the pipeline. The code issues fadds and ST, *Viking* groups them together and tries to execute. Because of the 3 cycle latency of fadds to compute results, the pipeline is stalled for 3 cycles before those instructions can complete.

```
!--- Break ---------
fadds    %f0,%f1,%f21     (issued)
st       %f21,[%l1]       (issued)
cmp      %i7,2            (issued)
(bubble)
!--- Break ---------
(bubble)
!--- Break ---------
(bubble)
!--- Break ---------
fadds    %f0,%f1,%f21     (completed)
st       %f21,[%l1]       (completed)
cmp      %i7,2            (completed)
!--- Break ---------
```

Note that a floating point exception from fadds will be reported to the ST in the above example. fp_exception is a deferred trap and is always reported to the next FPOP or FPEV (ldf or stf).

3.3.4.6 FP code example: freg dependency +in -> LDin

The following example shows how a floating point register dependency before a LD may stall the pipe. The code issues fadds and LD, *Viking* groups them together and tries to execute. The LD can not complete before the fadds completes, and the fadds takes 3 cycles to complete, hence the pipeline is stalled for that period. This is required to avoid destroying the source registers for current FPOPs.

```
!--- Break ---------
fadds    %f0,%f1,%f21     (issued)
ld       [%l1],%f21       (issued)
cmp      %i7,2            (issued)
(bubble)
!--- Break ---------
(bubble)
!--- Break ---------
(bubble)
!--- Break ---------
fadds    %f0,%f1,%f21     (completed)
ld       [%l1],%f21       (completed)
cmp      %i7,2            (completed)
!--- Break ---------
```

Note that a floating point exception from fadds will be reported to the LD in the above example.

3.3.4.7 FP code example: freg dependency +out -> +in

The following example demonstrates how a floating point register dependency between 2 FPOPs activates the data forwarding, and does not cause the pipeline to stall. The destination register of FPOP1 is a source register for FPOP2, both FPOP are able to enter the FPQ immediately, but FPOP2 will not start until FPOP1 reaches FWB stage. However, FPOP2 does not wait until FPOP1 has written the data into the register, because FPOP2 receives the data from the forwarding path.

```
!--- Break ---------
fadds   %f0,%f1,%f21
!--- Break ---------
fadds   %f21,%f2,%f22
!--- Break ---------
```

3.3.4.8 FP code example: freg dependency - +in -> +out

The following example demonstrates that a floating point register dependency between 2 FPOPs does not cause the pipeline to stall. Both instructions are able to complete immediately.

```
!--- Break ---------
fadds   %f0,%f1,%f21
!--- Break ---------
fadds   %f2,%f3,%f0
...
!--- Break ---------
```

3.3.5. Spread address calculation and Memory reference

In order for *Viking* to implement single cycle memory references, it is necessary for the address registers to be stable by the D2 pipeline stage of the memory reference. This implies that address computation must be completed in the E0 stage of the previous instruction group. A typical example is:

```
sll %l0,0x3,%l0
!--- Break (ALUOP into Memory reference Address)
ld  [%o0+%l0],%l1
```

Since the result of the shift is needed for the load address calculation, they must be executed in separate groups. If the shift were not producing data for *that* load, they could have been grouped together. This example is not particularly bad. In general, the shift will be grouped with previous instructions, and the load will be grouped with subsequent operations.

The next case is less common, but can be significant in some codes. This arises when the address is calculated in a *cascaded* instruction group. The results of a cascade are not available until the end of the E1 execution stage. Since the memory reference requires data at the end of E0, a wasted pipeline cycle must be inserted. Example:

```
sll %l0,0x3,%l0
add %l0,%o0,%l0
!--- Break (Cascade into memory reference)
!--- PIPELINE STALL for one cycle
ld  [%l0],%l1
```

Note that this example is functionally *identical* to the previous example, (except for the actual content of %l0). However, it requires three cycles to execute rather than two. In the more general case, a sequence such as this will be used only when more complex address arithmetic is required.

A third, and quite common case occurs during (for example) linked list traversal. This happens when the results of one load are immediately used as part of the address of the *next* load.

```
ld  [%l0],%l1
!--- Break (load into memory reference)
!--- PIPELINE STALL for one cycle
ld  [%l1],%l2
```

The pipeline stall is required for the same reason as the previous example- the results of a load instruction are not available until the end of the E1 pipeline stage.

If any of the above cases can be *spread* apart, by moving other instructions which can be executed between these dependencies, performance will be increased.

3.3.6. Arithmetic dependencies

Viking must group instructions according to available hardware resources. Some of these restrictions are more severe than others. The effect of each of these restrictions varies greatly depending on the code being run.

3.3.6.1 Ports to memory

The most basic hardware resource limitations is a single port to memory. This is what restricts *Viking* to one load or store per cycle. Similar to this, a single port to the instruction cache restricts branch performance.

3.3.6.2 Ports to the FPU

Although *Viking* has independent floating point adder and multiplier, only one floating point operation per cycle can be dispatched. A load or store between a floating point register and memory can be done in the same cycle as a floating point operation, however.

3.3.6.3 Integer register write ports

The most restrictive of these resources are the write ports into the *integer* register file; 2 are available. All arithmetic instructions use *one* of these ports. Load instructions use *one* write port, and load double instructions use *both* write ports. Note that floating point load operations do not use *any* integer register write ports. So, for example, the following code segment (which loads double floating point registers) executes in two cycles:

```
add %10,%11,%12
and %12,%0xff,%13
ldd [%o0+0x100],%f2
!--- Break (Three instructions)
add %14,%15,%16
and %16,0xff,%17
ldd [%o0+0x108],%f4
!--- Break (Three instructions)
```

While the next example (which is similar, but loads into integer registers), executes in four cycles:

```
add %10,%11,%12
and %12,%0xff,%13
!--- Break (No more write ports)
ldd [%o0+0x100],%i2
!--- Break (No more write ports)
add %14,%15,%16
and %16,0xff,%17
!--- Break (No more write ports)
ldd [%o0+0x108],%i4
```

3.3.6.4 Integer Arithmetic units

Viking has two logical integer ALU's. This is implemented internally with three separate ALU's and a shifter, for speed reasons. Each of these can produce one result each cycle. One of the ALU's can use the output of either the other ALU *or* the shifter as its input.

The number of ALU's cannot limit the machines performance, since no more than two results can be stored by the register file. However, only a single shifter exists. This means that a result cannot be cascaded *into* a shift operation.

3.4. Instruction Grouping Rules

Viking forms groups of instructions by examining the available, or *candidate* instructions from its instruction queue. A set of rules are applied to decide which of the instructions will be selected for inclusion in the next group to be executed.

The group size can be limited by the number of instructions available in the queue. The number of available instructions depends mostly on the number of branches executed recently, and the instruction cache performance. When a program is executed for the first time, most of the instructions will not be present in the cache, and performance will be dominated by the time taken to fetch instructions from external memory.

There are several classes of rules. The most basic classes are *break after*, and *break before*. This determines whether the instruction group will be terminated before a particular instruction, or after it. These two classes are further broken down into rules based on the available instructions, rules based on the *previous* instruction group, and rules based on exceptions.

The following sections will provide a description of all these rules.

3.4.1. Break After Rules

The following rules "Break After" an instruction based on relations among the first three instructions in the queue. These rules will prevent any further instructions from being included in the current group. The best example of this is a branch instruction. Instruction groups are *always* terminated when a branch is included.

Table 3-1 *Break After Rules*

Break After First Valid Exception
Break After Any Control Transfer Instruction
Break After Condition Codes set in Cascade
Break After MULSCC destination not equal to source of next MULSCC
Break After first instruction after Annulled Branch
Break After first instruction midway through a branch couple

3.4.1.1 Break After First Valid Exception

This rule prevents instructions from entering the pipeline after an exception (*instruction access exception*) has been signaled. The exception will travel through the pipeline, and only actually occur when it reaches farther into the pipeline.

3.4.1.2 Break After Any Control Transfer Instruction

This rule breaks the current group between any branch, and the delay instruction which follows the branch. Any instructions which are to be grouped along with a branch must appear before it in the code.

3.4.1.3 Break After Condition Codes set in Cascade

This rule prevents any additional instructions from being accepted after an instruction cascade in which the second ALU operation sets condition codes. It is used primarily to simplify implementation. This rule also terminates groups when two properly formed MULSCC instructions are grouped.

3.4.1.4 Break After MULSCC destination not equal to source of next MULSCC

This rule detects poorly formed MULSCC cascades. It prevents multiple MULSCC instructions from being executed in parallel unless the second uses the result of the first. This condition will never happen in normal multiply sequences, but is architecturally legal.

3.4.1.5 Break After first instruction after Annulled Branch

This instruction prevents multiple instructions from executing in the delay group of an annulled branch. The instruction in the delay group of an annulled branch will normally be executed only if the branch is taken. When this occurs, only the *first* instruction after the branch is expected to be executed. If the previous branch is untaken, the instruction in the delay position is not executed at all. *Viking* will begin executing this delay instruction, assuming the branch will be taken, and squash it later if need be. Restricting the grouping to contain only a single instruction in this position simplifies the implementation.

3.4.1.6 Break After first instruction midway through a branch couple

This rule prevents multiple instructions from being executed when a branch couple is being processed. SPARC allows for a restricted number of branch couple conditions. To simplify implementation, *Viking* will only execute single instructions during branch couples.

The following sequence demonstrates this rule:

```
        ba  dest1
        !--- Break after CTI
        be  dest2
        !--- Break after CTI
        (Never Executed)
        .
dest1:  add %l0,%l1,%l2
        !--- Break midway through branch couple
        add %l3,%l4,%l5
        .
dest2:  add %l4,%l5,%l6
        .
        .
```

3.4.2. Break Before

Most rules are "Break Before" rules. They are typically used to break a group when one of *Viking*'s limited resources has been completely used. For example all instruction groups are terminated (a break is generated) *before* a second memory reference.

There are many grouping rules which are required to handle pipeline hold conditions. These rules will not be listed here.

All the "Break Before" rules are listed below:

Table 3-2 *Break Before Rules*

Break Before Invalid Instruction
Break Before Out of Integer Register Read Ports
Break Before Second Memory Reference
Break Before Second Shift
Break Before Second FPOP
Break Before Second Cascade
Break Before Cascade into Shift
Break Before Cascade into JMPL
Break Before Cascade into Memory Reference Address
Break Before Load Data Cascade Use
Break Before Previous Group Cascade into Memory Reference Address
Break Before Sequential Instruction
Break Before Control Register Read after Previous SetCC
Break Before MULSCC unless first one or two instructions
Break Before Extended Arithmetic from CC set in Current Group
Break Before Delay Group CTI unless first
Break Before CTI in JMPL delay unless RETT

3.4.2.1 Break Before Invalid Instruction

Viking uses this rule to wait for instructions from the instruction queue and instruction cache. Even instruction access exceptions are considered valid instructions. It is possible for fewer than three instructions to be valid from the queue. This limits the maximum number that can be executed at a given time.

3.4.2.2 Break Before Out of Integer Register Ports

This rule prevents a group from using too many register file ports. Four operand read ports and two operand write ports are available. Memory address register ports are independent, and do not affect this rule.

3.4.2.3 Break Before Second Memory Reference

Viking has only a single port to memory. This rule prevents a group from attempting to use it twice. This rule does not discriminate between floating point or integer LD/ST.

3.4.2.4 Break Before Second Shift

Though *Viking* has several ALUs, only a single shifter is provided. It must be used in the first execute stage, so it may not be the second operation in a cascade. The result is produced early enough that it may be the source of a cascaded operation. This allows for common operations like shift&add, and shift&compare.

3.4.2.5 Break Before Second Cascade

Only one operand may be cascaded into the E1 of the integer pipeline. This rule prevents using two E0 results to form a cascaded operation.

3.4.2.6 Break Before Cascade into Shift

All shift operations must execute in the E0 stage. This rule enforces that restriction.

3.4.2.7 Break Before Cascade into JMPL

JMPL reads from the register file to calculate the target of its branch. The value is required before the E0 stage. This requires that the register not be changed in the group with the JMPL. This rule detects when such a condition exists, and forces a break before the JMPL.

3.4.2.8 Break Before Cascade into Memory Reference Address

In order to perform single cycle memory references, the registers forming the address for a load or store must be available before the D2 stage of the memory reference. This requires that they be changed no later than the E0 stage of the previous instruction group. This rule, and the *Break Before Previous Cascade into Memory Reference Address* rule enforce this restriction.

3.4.2.9 Break Before Load Data Cascade Use

A full cycle is required to access the on-chip data cache. The data from a load is available after the E1 stage of the load. Therefore, it may not be used before the E0 stage of the next instruction group. This rule prevents an instruction from using that data in the group with the load instruction (during E0 or E1).

3.4.2.10 Break Before Previous Group Cascade into Memory Reference Address

This rule enforces an extension of the *Break Before Cascade into Memory Reference Address* rule. It requires that the address registers stabilize by the end of the previous instruction groups E0 stage.

3.4.2.11 Break Before Sequential Instruction

Viking defines a set of instructions that can only be executed as single instruction groups. In addition, *Viking* can be forced to execute *all* instructions as single instruction groups through an ASI visible control bit,

The set of instructions which are always single instruction groups is:

SAVE and RESTORE
LDD and STD operations to integer registers
All *Alternate Space* stores (STA STDA STBA STHA)
Atomic operations: SWAP and LDSTUB
All *control register* accesses: Read and Write PSR ASR WIM Y
RETT
FLUSH
All software traps (Ticc)
Integer Multiply: UMUL UMULCC SMUL SMULCC
Integer Divide: UDIV UDIVCC SDIV SDIVCC
Tagged Operations that can trap (TADDCCTV TSUBCCTV)
Load or Store FP status registers (LDFSR STFSR STDFQ)
FBFCC (Branch on floating point condition code)

3.4.2.12 Break Before Control Register Read after Previous SetCC

Integer condition codes are part of the processor status register (PSR). To prevent them from being read while they are being modified, the processor forces an extra cycle between RDPSR instructions and a previous arithmetic operation that may have changed the condition codes.

3.4.2.13 Break Before MULSCC unless first one or two instructions

Viking executes MULSCC either as a single instruction group, or it executes *two* MULSCCs as a group. MULSCC is never grouped with any other instructions.

3.4.2.14 Break Before Extended Arithmetic from CC set in Current Group

Viking cannot use condition codes calculated in E0 as input to extended precision arithmetic in the same group. This rule inserts a break between the condition code computation, and its use in extended arithmetic (ADDX ADDXCC SUBX SUBXCC).

3.4.2.15 Break Before Delay Group CTI unless first

This rule prevents additional branches, other than true branch couples, from being executed in the delay group of a branch. This simplifies implementation by never having to squash the later branch when the previous branch is taken.

3.4.2.16 Break Before CTI in JMPL delay unless RETT

This rule allows an optimization of JMPL/RETT pairs, and simplifies the general case of JMPL couples. A full cycle is required in the pipeline between a JMPL and any other CTI, except RETT.

[Blank Page]

4

Viking Programmer's Model

[Blank Page]

4

Viking Programmer's Model

This chapter describes the *Viking* programming model. It explains the *Viking* processor from a programmer's point of view by discussing the processor's components, reset modes and how it affects *Viking*, JTAG, In-Circuit Emulation, Diagnostics and BIST (Built In Self Test). The processor components included in the descriptions are the two caches (instruction and data), the memory management unit (MMU), the store buffer, and the Floating Point Unit. Every control register involved in the operation of *Viking* is explained in the respective sections, along with comprehensive ASI descriptions which detail register bits and fields. The same ASI Map also formally defines all of the ASIs that *Viking* supports.

4.1. *Viking* Processor

Viking is a highly-integrated, super-scalar (multiple instructions per cycle) SPARC processor for use in single- and multi- processor systems. It features physical instruction and data caches, a Memory Management Unit (MMU), a Store Buffer, and a Floating Point Unit (FPU), all on chip. Accesses to these and other components within the chip are available through the use of ASI references, generally used for control and diagnostic purposes. See the SPARC Architecture Manual for more on ASIs.

4.2. CC or MBUS mode

Viking can operate in either of the two modes, CC or MBUS. In CC mode, *Viking* assumes the existence of an external cache, and a cache controller (such as the *MXCC*, for example). In MBUS mode, *Viking* makes no such assumption, and conforms to level-2 MBUS protocol. The differences of the two modes in terms of *Viking* operation will be compared and explained throughout this document.

4.3. Reset Operation

Viking implements three forms of reset. Hardware reset, sometimes called *power-up reset* is initiated *external* to the processor by asserting the RESET_ signal. There is no direct mechanism within *Viking* to assert hardware reset. This function is typically implemented within the system logic. It is also possible to cause a hardware reset in emulation mode.

Built in self test (BIST) generates a second type of reset, which is nearly identical to the hardware reset. BIST operations can be requested either by software (with a STA), or via the JTAG interface. When BIST is initiated through software using STA, an internal reset is automatically generated. When BIST is initiated by JTAG. however, the user needs to generate the reset. This can be done by entering the

TAP reset state by either assertion of TMS for 5 consecutive TCK cycles or asserting TRST_.

In addition to the hardware reset there is an internally-generated reset referred to as a *Watchdog Reset.* This reset is caused by entry into error mode as described in the SPARC Architecture Manual.

4.3.1. Hardware Reset

Several actions are taken following a hardware reset. A detailed description of timing requirements on the reset signal is presented in section 10.2.2.6, in the System Interface chapter. Once the reset has been requested, *Viking* spends several hundred cycles initializing internal logic. In particular, during this time the cache column redundancy repair circuits are configured. All chip outputs are held inactive, or tri-stated during this time.

The following table show what actions are taken by hardware reset.

Table 4-1 *State after hardware reset*

Register/Bit Affected	States After Reset
Floating Point Queue	Invalidated
Boot Mode (MCNTL.BT)	1 (indicating boot mode)
MMU Enable (MCNTL.EN)	0 (MMU disabled)
No Fault (MCNTL.NF)	0 (Faults enabled)
Data Cache Enable (MCNTL.DE)	0 (Data cache disabled)
Instruction Cache Enable (MCNTL.IE)	0 (Instruction Cache disabled)
Store Buffer Enable (MCNTL.SB)	0 (Buffer Disabled)
MBUS/CC Mode (MCNTL.MB)	1/0, Depends on CCRDY_ pin
Parity Enable (MCNTL.PE)	0 (Even Parity Disabled)
Snoop Enable (MCNTL.SE)	0 (Snooping Disabled)
Partial Store Ordering (MCNTL.PSO)	0 (TSO/Strong Ordering)
Data Prefetcher (MCNTL.PF)	0 (Prefetcher Disabled)
Alternate Cacheable (MCNTL.AC)	0 (non-cacheable)
Table Walk Cacheable (MCNTL.TC)	0 (Non-cacheable)
Error Mode (MFSR.EM)	0 (Not an error mode, or watchdog reset)
TLB Lock Bits	0 (all TLB lock bits cleared)
Multiple Instruction Mode ACTION.MIX	0 (Single Instruction Execution)
Breakpoints (MDIAG)	0 (All breakpoints disabled)
Program Counter	0 (PC = 0x0, NPC = 0x4)
BIST Status (BIST.STATUS-LONG)	0 (No BIST since reset)
Store Buffer Tags	Valid bits cleared
Store Buffer Control	Pointers set to zero
Store Buffer Contents	Uninitialized
Data Cache	Contents Uninitialized
Instruction Cache	Contents Uninitialized
Register File	Contents Uninitialized
Processor Status Register (PSR)	S=1, ET=0, EC=0, Ver=4, Impl=0 PSR.CWP Uninitialized
WIM (Window Invalid Mask)	Uninitialized
MFSR	Uninitialized (except for MFSR.EM bit)
MSFSR (Shadow FSR)	Uninitialized
Emulation Facilities	Disabled

All values shown as *uninitialized* are just that, not set to any guaranteed state after hardware reset. They must be initialized before use.

Important Note:
In order for *Viking* to properly reset, care must be taken in the system implementation. In particular, JTAG operation may affect *Viking*'s ability to reset (the JTAG TAP controller should be in the reset state when hardware reset is asserted).

The internal phase locked loop (PLL) may take a long period of time to stabilize at power on (approximately 100 miliseconds). This time must be accounted for in the assertion of an external reset. Unpredictable operation occurs if reset is deasserted before the PLL has locked.

After the above state has been set, *Viking* initializes all internal state, and take a reset trap. This forces execution to begin at virtual address 0x0. Since boot mode is set, physical address 0xff000000 is used to fetch instructions from memory. System software may distinguish hardware reset from watch dog reset by the MFSR.EM (Error Mode) bit being cleared. Since the ACTION.MIX (multiple instruction execution) bit is cleared, *Viking*'s superscalar execution is disabled, and a maximum of one instruction may be executed in each cycle.

None of the entries in the Store buffer, data cache, Instruction cache, or TLB (except lock bits) change. Software is responsible for initializing each resource before enabling them.

Important Note:
System software is expected to examine the state of the BIST_STATUS register after any reset to determine whether the reset trap occurred as a result of a built in self test operation.

4.3.2. Watchdog Reset

In addition to the hardware reset there is an internally-generated reset referred to as a watchdog reset. This reset is caused by entry into error mode as described in the SPARC Architecture Manual.

To allow recovery from many error mode conditions, as little state as possible is affected by watchdog reset. The only *memory* bit affected by a watchdog reset is the MCNTL.BT (boot mode) bit. The MFSR.EM (Error Mode) bit is set to indicate that this is a watchdog reset, as opposed to a hardware reset. Since the cache redundancy logic has already been programmed (during hardware reset), it is not done again. Breakpoints are cleared at watchdog reset.

In CC mode, *Viking* issues an error mode bus cycle causing the cache controller (or external system logic) to record the occurence of error mode. The completion of this bus cycle causes watchdog reset.

Once the above actions have been completed, a reset trap will be generated.

4.3.3. Single Instruction Execution

At power-on reset, *Viking* will execute in *single instruction execution* mode. Multiple instruction per cycle execution is enabled by setting the ACTION.MIX bit. (See section 4.14.4.5 — *Breakpoint ACTION Register*).

4.3.4. Boot Mode

For a detailed description of boot mode, see section 4.11.11.1.

Boot mode is a special MMU bypass mode, where all *instruction* accesses (and alternate spaces references through ASIs 0x08 and 0x09) pass their *virtual* address in physical address bits 27 through 0, and the upper eight physical address bits (35 through 28) are set to 0xff.

Boot mode is entered after any reset (either hardware or watchdog). It may be disabled by clearing the MCNTL.BT bit explicitly. Note that boot mode *overrides* the MMU enable for instruction accesses. Boot mode does not affect data references.

During boot mode, the MCNTL.AC bit is ignored for instruction references and alternate space transactions through instruction space ASIs (0x08,0x09), making these accesses to instruction space non-cacheable when BT=1. However, the AC bit is still used for *data* references in this mode.

4.3.5. Built-In Self Test (BIST)

Viking has included Built-In Self Test (BIST) logic on chip. There are two types of BIST, short and long versions. BIST is a quick check for device integrity, it is not an exhaustive proof that the device is 100% correct. Many types of device faults will be detected by an incorrect signature value after a BIST. The long BIST operation is a more exhaustive check of the logic than the short BIST. This section describes how BIST operates, how to initiate BIST, how to use the results, and warnings regarding BIST operation.

4.3.5.1 Internal BIST operation

BIST uses internal logic scan paths to write in pseudo-random test patterns into the chip logic (internal states). One cycle of execution is then run, to let the states assume their next states, then state of the logic is captured through the scan path into a signature analyzer. The signature analyzer creates a signature value based on the results from the logic and stores this value in the BIST.SIGNATURE register.

4.3.5.2 BIST coverage

The BIST sequence checks all normally scannable logic, but does not check most internal memories. The TLB, store buffer, prefetch buffers, cache arrays, and register files are not checked by BIST.

4.3.5.3 Signature

The correct signature value is known but is device stepping dependent. Correct signature values for the device will be published. (Not yet known for this revision of the documentation).

Different signature values are generated for the long and short BIST operations.

4.3.5.4 Initiating BIST

To initiate BIST a STA to ASI 0x39 is issued. A "long BIST" is selected by writing to virtual address 0x100, while a store to address 0 selects a "short BIST". An ASI 0x39 access to any address other than 0x100 or 0x0 will generate a data_access_exception.

Internal logic controls the BIST operation once requested. An external reset aborts the BIST operation. When the sequence completes, an internal reset is generated (see the hardware reset description above).

BIST may also be initiated through the JTAG interface. For details on JTAG initiated BIST, see chapter 5 — *JTAG Serial Scan Interface*

4.3.5.5 BIST ASI operation

ASI=0x39 - BIST Diagnostics.

There are 2 memory mapped, *Viking* -specific, MMU resident diagnostic registers used to support built-in self test (BIST). Any Byte, Halfword and Doubleword or Swap access into any of these diagnostic registers are explicitly illegal and will generate a data_access_exception. LDA/STA single only is allowed.

Table 4-2 *BIST Diagnostic Registers within ASI 0x39*

Address	Load/Store	Data Format	Description
0x00000000	Store	Don't Care (should be zero)	Start Short BIST
0x00000100	Store	Don't Care (should be zero)	Start Long BIST
0x00000000	Load	Signature[31:0]	Read 31-bit signature
0x00000100	Load	Status[31:0] (see value below)	Read 2-bit BIST Status

The possible values of the status register are:

Table 4-3 *BIST status register values*

Value	Meaning
0x00000000	No BIST run
0x00000001	Short BIST run
0x00000002	Long BIST run

Hardware reset PIN will clear BIST.STATUS. These bits are not affected by watchdog reset.

4.3.5.6 Warnings regarding BIST operation

After BIST finishes, *Viking* generates an internal reset. This internal reset behaves similar to the hardware reset. One of the consequences of this reset is that MCNTL gets initialized - in particular the MCNTL.PE bit gets reset so that Viking starts generating inverted parity on the pins. This internally generated reset is *not* seen by a cache controller such as the *MXCC*, so if the parity was enabled in the *MXCC* before starting BIST - it is still enabled in the *MXCC* after BIST. This results in a mismatch between what *Viking* and *MXCC* do and expect on writes. Software should check the MXCC.PE bit after BIST has completed and adjust the *Viking*'s MCNTL.PE bit before attempting any writes. To be safe - the following algorithm

could be used:

(Before starting BIST)
Flush the store buffer (even though STA 0x39 does it too)
Check the *MXCC* status register and wait until nothing is pending
Start BIST

(After BIST finishes)
Read BIST status
Read BIST signature
Read *MXCC* control register
Set *Viking*'s MCNTL.PE bit if MXCC.PE bit is set
Continue

Even though the *MXCC* was mentioned as an example, any other system component could have a similar problem since they do not see *Viking*'s internally generated reset. Thus care has to be taken in recovering the system after BIST finishes.

4.4. Instructions

This section describes *Viking* instruction operation for those instructions which require a more detailed description than is already provided in the SPARC Architecture Manual. The instructions described here include:

Instructions
IMUL
IDIV
WRPSR
FLUSH
STBAR
SIGM

4.4.1. Integer Multiply (IMUL)

Viking implements integer multiply in all its forms conforming to the SPARC architecture specification. It is mentioned here only because the instruction is new to the *Version 8* SPARC architecture.

Integer multiply is implemented in the *floating point* unit of the processor. Normally, these operations will wait until the completion of any pending floating point operations before execution (indicated by FP Queue empty). If the FPU is in *exception mode*, integer multiply operations will proceed without waiting for the floating point queue to empty. Integer multiply will not cause any deferred floating point exceptions to be signalled.

4.4.2. Integer Divide (IDIV)

For most numeric conditions, integer divide works exactly to the SPARC architecture specification. Due to limitations in the hardware, there are certain numeric cases that cannot be completed. In these cases, *Viking* will signal an *illegal_instruction* trap (trap type 0x02). The case where this occurs is when the 64-bit value comprised of {Y,rs1} has numerically significant bits beyond bit 51.

In effect, *Viking* only implements a 52-bit by 32-bit integer divide, compared to the 64-bit by 32-bit specification. This holds true for both positive and negative numbers when the operations is a signed divide.

System software is expected to emulate these integer divide operations when required.

Integer divide is implemented in the *floating point* unit of the processor. Normally, these operations will wait until the completion of any pending floating point operations before execution. If the FPU is in *exception mode*, integer divide operations will proceed without waiting for the floating point queue to empty. Integer divide will not cause any deferred floating point exceptions to be signalled.

4.4.3. Write PSR (WRPSR)

Viking implements the write PSR instruction according to the SPARC architecture manual, with one qualification. Since no coprocessor port is provided on *Viking*, system software is prevented from trying to enable coprocessor operations.

If a WRPSR instruction attempts to set the PSR.EC bit, an *illegal_instruction* trap will be generated immediately. Since the EC bit can never be set, all coprocessor instructions will generate *cp_disabled* traps (trap type 0x24).

The timing requirements of PSR write operations match the SPARC Architecture Manual exactly. A three *instruction* (not cycle) delay is required between changing any PSR fields, and using the contents. Note also that when a JMPL/RETT instruction pair is seen, *Viking* will use the PSR.PS (previous supervisor) bit to check protections for the instruction fetch of the JMPL's target.

The PSR.IMPL (implementation number) field of the PSR is always set to 0x4. The PSR.VER (version number) field is set to 0x0.

4.4.4. Flush (IFLUSH)

Viking implements FLUSH slightly differently than the SPARC Architecture Manual suggests. No cached information is explicitly flushed by the instruction. The cache consistency mechanisms are used to ensure that caches always have correct data (both instruction and data caches). The FLUSH operations simply causes an exact synchronization of all pending activity.

When a FLUSH instruction is executed, it will cause *Viking*'s store buffer to be drained. This causes all pending bus activity to complete. Any cache coherency transactions (for instance invalidation of instruction cache entries) will occur as the store buffer clears. The internal processor pipeline and instruction buffer will also be cleared. When the pipeline resumes execution after the FLUSH, it will fetch data from the instruction cache which is guaranteed to be up to date with respect to all prior processor activity.

See also section 4.7.5 — *Instruction Cache Consistency* for further discussion of FLUSH instruction implementation.

Important Note:

The FLUSH instruction affects only a *single* processor. No other processors in a system will do a flush operation unless explicitly requested to do so (by their own FLUSH).

This is not of concern to most applications, but is significant to certain operations, such as dynamic linkers. These programs must be written carefully to ensure that all processors see a consistent view of program memory under all circumstances. This may require modifying memory in a certain order, and/or writing intermediate values to guard against temporary inconsistencies. Note that cache coherence is maintained on *double words*.

4.4.5. Store Barrier (STBAR)

The STBAR instruction forces store operations to be performed in order while in PSO (partial store ordering) mode. *Viking* implements this functionality as described in the SPARC Architecture Manual.

The instruction is implemented in the "RDASR" *reserved* instruction space. The opcode is equivalent to RDASR 0x0f, %g0, the exact encoding is 0x8143c000.

The STBAR instruction sets the SBTAGS.SP in the last allocated (valid) entry. If no entry is allocated, the STBAR instruction is remembered in the bus unit arbitration logic. When the bit is set, the bus unit waits for the PEND_ input to go inactive before issuing any stores after the one with the bit set. Without the STBAR, the SBTAGS.SP bit will not get set, and the b_unit continues issuing stores as fast as the external cache controller can buffer them. This would allow stores to be performed out of order in the sytem.

All this happens only in PSO mode. In TSO mode, the bus unit waits for PEND_ at all times.

See section 4.5.1.3 — *Partial Store Ordering (PSO)* for more details.

4.4.6. Signal User Emulation Request (SIGM)

Viking implements a special, *Viking-specific* instruction called SIGM (Signal Emulation), that is used in conjunction with the on chip JTAG based emulation facilities provided by *Viking*.

The instruction is implemented in the "RDASR" implementation dependent extended opcode space defined by the SPARC architecture. The opcode is equivalent to RDASR 0x1f, %g0, which encodes to 0x8147c000.

The operation of this instruction is dependent on the state of the JTAG controlled MCMD register. If the MCMD.INITM bit is cleared, the SIGM instruction will be executed as a NOP. If the MCMD.INITM bit is set, execution of SIGM will cause immediate entry into emulation mode (See chapter 6 — *Remote Emulation Support*) for details).

The MCMD.INITM is always initialized to zero at JTAG Tap controller reset. In order for the SIGM instruction to cause entry into emulation mode, the bit must be *explicitly* set by a remote emulation processor using the JTAG. It is not possible to set the MCMD.INITM bit using the processor alone.

4.5. Memory Model

This section will describe the programmers view of memory in a *Viking* based system. The exact view of memory is highly dependent on system implementation, some of the details below may not apply to certain system environments.

4.5.1. Three models of memory: Strong Ordering, TSO, PSO

The SPARC Architecture defines three views of memory; the differences between these models are described in detail in the SPARC Architecture Manual.

Viking allows the use of any of the three models at any time. Each of these models places different requirements on the system as well. Certain systems may be capable of implementing some or all of the models.

4.5.1.1 Strong Ordering

Strong ordering allows for maximum software compatibility, but can decrease performance and increase system complexity. In this model of memory, all transactions are seen in the exact order that they were issued by all processors, caches, and memories in the system. It allows for a very simple programmers model.

If the system allows, *Viking* can implement strong ordering by disabling the internal store buffer (MCNTL.SB bit). This will cause decreased performance in most system environments, particularly when using CC mode, since all store operations write through to external caches.

4.5.1.2 Total Store Ordering (TSO)

Total store ordering is similar to strong ordering, except that only the order of store operations is guaranteed to be exact across the system. This memory model allows *most* multi-threaded applications to operate consistently with good performance.

TSO mode is the *nominal* memory model of *Viking* based systems. For TSO mode, the store buffer must be enabled (MCNTL.SB=1), and the MCNTL.PSO (partial store ordering) bit must be zero.

4.5.1.3 Partial Store Ordering (PSO)

Partial store ordering is the highest performing of the memory models. It eliminates most implicit ordering requirements and expects software to *explicitly* enforce ordering when needed. This model requires the most careful use of memory by applications.

This explicit ordering may be requested by use of the STBAR instruction. The STBAR instruction will guarantee the order of store operations before and after the STBAR instruction only. To achieve the equivalent of the TSO model, an STBAR might need to be inserted prior to *every* store operation.

PSO is enabled by setting both the MCNTL.PSO and MCNTL.SB bits to one. *Viking* implements PSO mode in cooperation with external cache and memory controllers. The PEND_ signal is sampled in order to determine the completion of store transactions.

4.5.2. Atomic Operations- SWAPs and LDSTUB

SWAP and LDSTUB instructions are used to implement semaphores and other atomic operations in memory. In both uniprocessor and multiprocessor (MP) systems, it is important to maintain synchronization between processes which share common memory.

Atomic operations force the store buffer to copy out before starting.

If an exception occurs during the store buffer copy-out caused by an atomic operation, the operation is *not* completed. A data_store_error is taken, which disables the store buffer at once. The atomic operation may be restarted when the CPU returns from the store buffer exception handler (see section 4.12.5 — *Store Buffer (Data Store) Exceptions*).

If the atomic operation itself encounters an exception on either the write or read access, a data_access_exception will be reported. *Viking* guarantees that the destination register will *not* be updated. The system is responsible for insuring that the destination memory location is not modified (as in any store exception).

4.5.2.1 Atomic Operations in CC mode

Atomic operations have traditionally been implemented as a *locked* sequence of loads and stores (*Read-Modify-Write*). In order to support higher performance *packet-switched* buses, *Viking* implements a true swap operation, whereby it supplies the new data to be written along with the request for the current memory data. This is done to ensure that *Viking* receives the current value of the "old" data. The system or external cache logic must be capable of accepting this transaction. This is made possible by the definition of SPARCs atomic operators- the data to be written is independent of what is read.

4.5.2.2 Atomic Operations in MBUS mode

Since MBUS mode operates with a copy-back, write allocate cache protocol, the operation of atomic transactions is simpler than in CC mode.

For cacheable references in MBUS mode, no special bus operations are done for atomic transactions. They are implemented as a simple sequence of reads and writes. *Viking* will read, and acquire ownership of the data being referenced, and all operations will occur within the internal cache. (If the data is shared, acquiring ownership implies issuing a CI as necessary).

Non-cacheable references in MBUS mode operate more traditionally. They will appear on the bus as a locked read-write sequence. The LOCK bit within the MBUS address field will be set, and bus arbitration will not be released between the read and write.

4.5.2.3 Alternate Space Atomics

The SWAPA/LDSTUBA are similar to the SWAP/LDSTUB operations, with some restrictions. Swap Alternates to ASI locations other than 0x08-0x0b, and 0x20-0x2f will cause data_access_exceptions. Alternate atomic operations to ASIs 0x08-0x0b, and 0x20-0x2f are handled like ordinary atomic operations.

4.5.3. Load and Store Alternates

Loads and stores to alternate address spaces are generally performed for low level control of *Viking*, and the external system. A summary list of valid ASIs can be found in section 4.16 — *ASI Map*

All ASI operations are performed *synchronously*. Before starting execution of any ASI operation, the store buffer is flushed (except for ASI 0x20-0x2F, 0xa-0xb, which are asynchronous and do not cause a store buffer copyout). This action ensures that the processor state is consistent before the access begins. As an example, context register writes cause the store buffer to copyout preventing the store buffer from containing operations that "belong" to contexts other than the current one. This allows data_store_exceptions to be associated with the faulting process more easily.

If an exception occurs during the store buffer copy-out caused by an LDA/STA, the operation is *not* completed. A data_store_exception is taken, which disable the store buffer at once. The STA operation may be restarted when the CPU returns from the store buffer trap handler.

Alternate space transactions through ASIs 0x8-0xb are treated as normal load and store operations. They are translated by the MMU according to the definition of the ASIs in the SPARC architecture manual.

Transactions through the *pass-through* ASI space (0x20-0x2f) are treated as normal loads and stores, except that they are not translated by the MMU. For these operations, cacheability is determined by the MCNTL.AC (alternate cacheable) bit.

4.5.4. Non-Cacheable Loads and Stores

The cacheability of memory references is determined as described in the instruction cache, data cache, and MMU sections of this manual.

Non-cacheable loads cause the contents of the Store Buffer to be copied out to memory before proceeding. This is done to ensure proper memory ordering of accesses to I/O space.

If an exception occurs on the store buffer copy-out caused by a non-cacheable load, the load operation is *not* completed. A data_store_exception is taken, which disables the store buffer at once. The load operation may be restarted when the CPU returns from the store buffer exception handler.

Unlike loads, noncacheable stores do not normally force the Store Buffer to copyout. They are simply placed in the store buffer, like cacheable stores.

4.5.5. Page Table Memory Operations

As described above, normal system software access to the in-memory MMU page tables must be done carefully. Conflicts may exist between software and hardware use of these tables.

4.5.5.1 Hardware use of page tables

The MMU hardware must autonomously access the page tables to perform translations, and modify the page tables to keep referenced and modified statistic bits up to date.

When the *Viking* MMU table walk hardware accesses these page tables it will do so in a way that guarantees consistency between multiple processors. This is done primarily through the use of locked or *atomic* memory transactions

whenever modifying page table R&M bits. As long as certain rules are followed in system software, the entire table walk need not be done with a locked transaction, only the updates.

4.5.5.2 System software use of page tables

System software must access page table to check statistics, and remap physical memory as required.

When only system software accesses page tables in memory, consistency is guaranteed by the normal cache consistency mechanisms, as well as by standard critical section mutual exclusion semaphores. Unfortunately, *Vikings* table walk hardware has no way to recognize these software locking conventions. The following section describes how to deal with the consistency issues.

4.5.5.3 Hardware/Software page table consistency

Since both hardware and software generated references to the page tables may be in progress simultaneously, some algorithm must be used to guarantee that these two sources (as well as multiple instances of both of them) will not interfere with each other.

Inconsistency can occur in several ways. The most general is when software changes (e.g. invalidates, etc.) a page mapping and the table walk hardware has already read the table. In this situation, it is system softwares responsibility to do a MMU flush (or DeMap) operation to force the page table to be re-read by the MMU. In a uniprocessor environment this is sufficient. In a multiprocessor environment, the flush operation must be done on *every* processor in the system.

> Important Note:
> Software must guarantee that only a single DeMap operation is in progress at any one time across the entire system. Inconsistent operation will result if two DeMaps are received by a processor at any one time (including internal DeMap requests).

In CC mode, and only in systems that implement broadcast DeMap operations, the local flush operations is *automatically* broadcast to all processors. There is no need to interrupt remote processors to issue local flush transactions. In MBUS mode, and in CC mode systems which do not broadcast DeMap, *all processors* must be interrupted and told to do their own local flush operation. This can take considerable time to complete, but is generally not a performance bottleneck in smaller systems. Broadcast DeMap capability is recommended for higher performance, large multiprocessor systems.

A more difficult problem arises when the MMU hardware must *re-write* a page table entry to set or clear the referenced of modified bits. The hardware must be prevented from *overwriting* a modification that system software has just completed.

A general algorithm which prevents inconsistency in both situations is presented in figure 4-1 below. The exact implementation of this code is system dependent. The implementation of lock and unlock operations is memory model dependent, see the SPARC Architecture Manual for proper sequences.

Figure 4-1 *Generalized safe page table update algorithm*

```
        Lock                                /* Acquire exclusive page table access */
        RM_Accum = 0                        /* Any R+M updates will accumulate here */
Loop:   Reg = 0                             /* Set PTE to zero temporarily */
        Swap(TargetPTE,Reg)                 /* Write 0, read back current PTE */
        FlushAllMMUs(TargetPTE)             /* Flush ALL reference to this PTE in system */
        RM_Accum = RM_Accum OR Reg          /* Catch any late R+M Changes */
        if (TargetPTE ≠ 0) goto Loop        /* Continue until it's really zero */
        TargetPTE = NewPTE                  /* Safe to write new value */
        Unlock                              /* Release lock on page table access */
```

Operationally, the algorithm may take several iterations to complete. When used with non-broadcast system wide DeMap, a single iteration should be sufficient. The *FlushAllMMUs* operator is a system dependent mechanism to execute a flush operation on all MMUs in the system. System software should use the *RM_Accum* value as the *final* value that was in the PTE entry before modification. This will guarantee that no page table status information was lost.

4.5.6. Memory, Exceptions and No-Fault operation

All exceptions, including taken interrupts, invoke the SPARC trap handling mechanism. This mechanism includes a store buffer copy-out.

When a SPARC CPU takes a trap, it disables further traps by setting PSR.ET=0. This makes the CPU vulnerable; if a synchronous (i.e. non-interrupt) exception occurs here, the CPU enters an undesirable "Error Mode". Error mode will initiate a watchdog reset.

Several steps are taken to avoid this second exception and error mode. Before traps are disabled (PSR.ET is still 1) and before the exception handler is fetched, the store buffer copies out all pending writes to memory. This prevents any user-level faults caused by these writes from occurring at unexpected or unsafe points within the trap handler. If an exception does occur on this copyout, a data_store_exception is taken instead of the pending trap. The pending trap is ignored, since its restartable. If the copy-out successfully completes, the normal trap handling sequence continues, and PSR.ET is cleared.

Supervisor exceptions should be controlled by software. Once invoked, the trap handler can immediately set the MCNTL.NF bit to 1. The NF (No Fault) bit disables reporting of data_access_exceptions to the CPU. Since the trap handler is kernel code, most exceptions are fatal. Other errors, notably bus errors, are always possible, and so are disabled by the NF bit.

The MCNTL.NF bit disables exception reporting for all operations except those to ASI 0x09 (Supervisor Instruction Fetch), and actual instruction fetches to ASI 0x08 (User instruction fetch). Errors from LDA and STA transactions to ASI 0x08 *will* be masked by NF.

It is the responsibility of system software to ensure that the NF bit is never set upon return to user code. Any load operation which receives an exception masked by the NF bit will load *indeterminate* data to the destination register. Any

store which receives an exception masked by NF will have no effect on registers (guaranteed) or memory (system dependent).

An undesirable side effect of flushing the store buffer policy is a higher maximum interrupt latency. This latency is system dependent, based on the maximum time required to empty the store buffer. Every interrupt requires a copy-out, which in the worst case involves 8 unbuffered writes to memory. This delay may be unacceptable for real time applications. However, the copy-out requirement can be avoided by running with the store buffer disabled; in this case, the store buffer is always empty. Fast interrupt response is guaranteed, although overall performance is reduced by the resulting synchronous external stores.

4.5.7. Prefetch Exception Handling

For most cache misses on load or instruction fetch operations, *Viking* performs a *block read*. This will load four double-words into the cache in a burst transaction from memory. *Viking* will always request the word that it needs as the first word of the burst, then read the rest of the four double words addressed modulo four. A bus error may be encountered on any one of the double-word transfers.

Demand fetches are those required by the processor immediately. If a bus error occurs on a demand fetch, an exception is generally reported to the pipeline, causing an instruction_access_exception to occur.

Non-demand fetches are the remaining words in the burst, also called *prefetches*. If a bus error occurs on prefetch data, it will *not* be reported to the pipeline. In this case, the *entire* cache line referenced by this transaction will be invalidated. The demand fetch will have been satisfied, but none of the additional prefetch data is in the processor. If that data is required by the processor in the future (very likely), it will be fetched again, as a demand fetch. If the error at that location persists, it will then be reported to the pipeline as an instruction_access_exception.

Information about errors of this sort may be accumulated outside the processor for repair, or gathering statistics. If desired, external controllers may raise an interrupt to inform system software of the existence of these errors. System software may initiate attempts to eliminate the error at this point before the data is truly required (demapping pages, etc.).

4.6. Floating Point Unit

The *Viking* Floating Point Unit operates in accordance with the SPARC Architecture Manual, and this section describes in detail some of the operations.

All floating point operations are completed in their natural program order. No *out of order* execution is done. All register dependencies are resolved in hardware, causing pipeline delays wherever required. Floating point branch operations will hold the processor pipeline until all pending floating point compare operations have completed. No instruction delay is required between floating point compare and floating point branch instructions. All floating point load double and store double operations *ignore* the least significant register index bits, and will use the naturally aligned register pair.

This section will concentrate on the floating point queue interface, special numeric cases, and floating point exceptions.

4.6.1. FSR Version and Implementation Fields

Viking always sets both FSR.VER and FSR.IMPL fields to zero.

4.6.2. Special Numeric Cases

Viking handles all non-exceptional numeric cases directly in hardware. No *unfinished* exceptions are generated. Completing execution of many of these categories requires additional cycles.

4.6.2.1 NaN → Integer Conversion

When converting a number considered to be a NaN (Not A Number), *Viking* will always produce the fixed representation of 0 in the destination register. This is contrary to other implementations which may produce either 0x80000000 (-NaN), or 0x7fffffff (+NaN).

4.6.2.2 NaN Output Representation

When *Viking* produces a NaN result, it is stored in one of the following fixed formats:

Table 4-4 *NaN Output Representation Values*

Single Precision:	0x7fc0_0000
Double Precision:	0x7ff8_0000_0000_0000

This differs from the IEEE standard, and other SPARC implementations. These values will only be created in NaN reporting masked.

4.6.2.3 Rounding Operations and Underflow Detection

Viking detects underflow *after* the rounding operation. The IEEE Specification allows either method, *Viking*'s implementation may differ from other SPARC implementation.

4.6.2.4 FxTOiR Not supported

Old descriptions of the SPARC Architecture described floating point conversion with rounding to the rounding mode bits. These instructions have been removed from the architecture, and are not implemented on *Viking*.

At *Viking*, FxTOi is always rounded to ZERO, while other convertions adheres to FSR.RD mode bit.

4.6.2.5 Quad Precision

Viking does not support quad precision operations.

4.6.3. Floating Point Queue

Viking's floating point queue is 4 entries deep. All floating point operations are written to the queue. Floating point memory references, FSR operations, and queue operations are *not* written to the queue.

Storing the contents of the floating point queue should only be done when the floating point unit is in *exception mode*. Store Floating Point Queue operations when not in exception mode is *implementation dependent*. In these cases, *Viking* will hold the processor pipeline until all floating point operations have completed, and store information the *last completed* floating point operation. The floating point queue is *not* initialized at reset, storing queue contents before executing floating point operations will store undefined values.

The user visible values in the floating point queue contain the virtual address of the executing instructions, as well as the opcode being executed. A STD operation on the floating point queue register will store these values for the operation at the *front* of the queue to memory. The memory format is:

Table 4-5 *Floating Point Queue Format*

Address+0x0:	32-bit virtual address program counter
Address+0x4:	32-bit opcode

Each time the floating point queue is read while in exception mode, the *next* pending floating point operation in the queue will be stored. The FSR.QNE bit should be checked before each store to identify the last valid queue entry.

4.6.4. Floating Point Exception Details

Viking implements *deferred* floating point exceptions. Any floating point exception will remain pending until another floating point operation is requested, at which point the pending exception will be reported.

If an exception occurs in *exactly* the same cycle in which a new floating point instruction is being issued, the exception will remain pending until the next floating point operations is issued. If the new floating point instruction created a dependency with other instructions in the floating point queue, the exception will be reported immediately. This situation will also add one additional cycle to the execution time of the previous operation.

The presence of processor pipeline hold conditions (particularly from data cache misses) can cause the acceptance of a floating point exception to be delayed.

Viking will report exceptions immediately to instructions which are dependant on the result of the instruction which created the exception. Otherwise, the exception will be reported to a later floating point operation.

A *Sequence_Error* will be reported, and recorded in the FSR when a new floating point request is issued while the FPU is in exception mode. The exception wil be reported to the instruction causing the sequence error.

4.6.5. No FAST (non-standard) Mode

Viking ignores the *non-standard* mode bit in the FSR. The FSR.NS bit can be read or written, but has no effect on floating point execution.

4.6.6. FsMULd

The *FsMULd* (Floating point multiply single, produce double result) operation is fully supported by *Viking*.

4.6.7. Integer Multiply

Integer multiply operation is more completely defined in section 4.4.1 — *Integer Multiply (IMUL)*

Since integer multiply uses floating point logic, execution of integer multiply operations affects the timing of normal floating point operations. Integer multiply operations will only start if the floating point queue is empty, or if the FPU is in exception mode. Normal floating point operations will not resume until the

integer multiply has completed. Functionally, integer multiply operations do not affect floating point execution in any way.

Integer multiply operations cannot cause any exceptions.

4.6.8. Integer Divide

Integer divide operation is more completely defined in section 4.4.2 — *Integer Divide (IDIV).*

Since integer divide uses floating point logic, execution of integer divide operations affects the timing of normal floating point operations. Integer divide operations will only start if the floating point queue is empty, or if the FPU is in exception mode. Normal floating point operations will not resume until the integer divide has completed. Functionally, integer divide operations do not affect floating point execution in any way.

Integer divide operations can cause *divide_by_zero* and *illegal_instruction* exceptions, but do not cause any floating point exceptions.

4.7. Instruction Cache

The *Viking* internal instruction cache is a 20 K-byte physical address cache. It is organized as a 5-way set-associative cache of 64 sets. The line size is 64 bytes, divided in two half-lines of 32 bytes.

The Instruction Cache is physically addressed. Virtual addresses are translated by the MMU before accessing the Instruction Cache. The requirements for a cache hit are: bits [35:12] of the physical address must match the physical tag and the Valid bit of the referenced half-line must be set.

The cache is enabled by the IE bit of the MMU control register (see section 4.11.11.1). The cache is disabled at power-on reset (see section 4.3.1) and remains disabled until the IE bit is set.

At power-on reset the contents of the instruction cache are undefined. It is the responsibility of the software to initialize the Instruction Cache by resetting the Valid bits (See section 4.7.6.1 — *Instruction Cache Flash Clear*).

4.7.1. Cacheability

Most references are *cacheable*. This implies that they will be stored in the on-chip caches after they are read in from external memory. The instruction cache cacheability is determined as follows:

Table 4-6 *Instruction Cache Cacheability*

Translation Mode	IE=0, AC=X	IE=1, AC=0	IE=1, AC=1
Boot Mode BT=1, EN=X	Not Cached	Not Cached	Not Cached
MMU Disabled BT=0, EN=0	Not Cached	Not Cached	Cached
MMU Enabled BT=0, EN=1	Not Cached	C=1:Cached	C=1:Cached
		C=0:Not Cached	C=0:Not Cached

BT is the Boot Mode bit of the MMU control register.
EN is the MMU Enable bit of the MMU control register.
IE is the Instruction Cache enable bit of the MMU control register.
AC is the Alternate Cacheable bit of the MMU control register.
C is the Cacheable bit kept in each Page Table Entry.

4.7.2. Instruction Cache Replacement Policy

A limited history LRU (least recently used) algorithm with minimal complexity and fair accuracy is used to determine which lines in the cache will be replaced when necessary. The instruction cache maintains a history for each line in the cache, and a lock bit for all but line0. These bits reside in the set tag. Whenever a memory reference hits in the cache, the history bit is written to one. At the time this bit is being written, all the other history bits are checked. If all the *other* history bits in the set, logically ORed with their respective lock bits are already asserted, then they are all *cleared*. The result is that all five bits are never set at the same time, and that the last used entry will *always* be set. When this transition occurs, the accumulation of history begins again. In this way, a limited record of the most recently used members of a set can be kept.

When an instruction cache miss occurs, the required data is brought in from external memory and forwarded to the instruction queue and pipeline. Simultaneously, the new instructions are placed in the cache. This requires that some old information be displaced from the cache. Since the data cache is 5-way set associative, there are five lines which may be chosen for replacement.

The set tag is examined to evaluate the history and lock bits. The replacement logic first examines the lock bits.

A line that is locked into the cache is never selected for replacement. Since line 0 cannot be locked, if all other entries are locked it will always be selected for replacement, regardless of the state of the history bits.

If there is more than one unlocked line, the replacement logic uses the history bits to determine which line should be replaced. If there is a line with a history bit that is set to 0, it will be selected as the victim. The ordering of *Viking* cache lines starts with the big-end, hence for the instruction cache it fills starting line 4, 3, 2, 1, then 0.

Figure 4-2 provides two examples of the replacement scheme.

Figure 4-2 *Example of Instruction Cache Replacement Policy*

MRU Mask :	1	1	0	0	1
LCK Mask :	0	0	0	1	
Composite Mask :	1	1	0	1	1
Line :	4	3	2	1	0

MRU Mask :	1	0	0	1	1
LCK Mask :	0	0	0	0	
Composite Mask :	1	0	0	1	1
Line :	4	3	2	1	0

MRU = Most Recently Used - provides a limited history used bits. LCK = Lock - provides locked lines information.

In the first example, based on the composite mask, line 2 is chosen for replacement. In the second case, line 3 is chosen, since it is the rightmost available line. Note again that line 0 can never be locked. This is to ensure that cacheable data may always be stored in the cache (when the cache is enabled).

4.7.3. Snoop Hits and Lock bits

This section describes how snoop hits and lock bits may create an unrecommended scenario. It also explains what happens to a locked line in the cache that gets snoop hit.

> A *snoop hit* to a locked line in *Viking* bus mode will cause the line to be invalidated, without clearing the lock bits. A CRI, CI, or CWI in MBUS mode will also cause the line to be invalidated, without clearing the lock bits. This will prevent the addressed cache line from being reused. If that line happens to be locked, the data will not be removed from the cache.

If a locked line gets invalidated by a snoop hit, that line will never be replaced (because it is locked), and it will never be a cache hit (because it has become invalid), essentially reducing the number of active cache lines by one.

If a line is to be locked, the programmer must make sure that it will not get snooped out.

4.7.4. Instruction Prefetching

Instruction prefetching supplies instructions to the Instruction Queue (IQ). The IQ provides instructions for the pipeline to execute. The IQ comprises an 8 word FIFO *sequential instruction queue*, and a 4-word *target queue*. The pipeline can consume up to 3 instructions per cycle from the IQ. Instruction prefetching continuously tries to fill the IQ with the instructions in the execution stream, either

from the instruction cache (on a hit) or from memory (on a miss). There is no separate instruction prefetch buffer. When a CTI is encountered, prefetching immediately retrieves the target instructions and places them in the target queue.

A *demand fetch* occurs when the processor needs instructions that are not currently available in the IQ. This can occur during exceptions, traps, CTIs, their combinations that may alter the execution stream, or straight line accross previously unreferenced code. Only a *demand fetch* can initiate an MMU table walk, or cause an exception. Instruction prefetching can not initiate an MMU table walk, nor can it cause an exception. If a required translation is not present in the TLB, the MMU will wait for a demand fetch before starting a table walk.

Instruction prefetching is always enabled when the instruction cache is on. There is no explicit control bit. Instruction prefetching works the same way in both CC and MBUS modes. Errors may occur during prefetching, see section 4.5.7 for details on prefetch exceptions.

4.7.5. Instruction Cache Consistency

Instruction cache consistency is maintained in hardware. This means that modifications to instructions are reflected by invalidations in the instruction cache. The invalidations are executed in a finite, deterministic (but system and state dependent) amount of time, as long as MCNTL.SE (snoop enable) bit is asserted.

To make sure that a modification to an instruction has been effectively performed and is observable by the issuing processor, a FLUSH instruction must be executed (See SPARC Architecture Manual). The FLUSH instruction forces the execution of all the pending writes and will also flush the instruction prefetch buffer and pipeline. The FLUSH instruction executes synchronously, implying that the *Viking* processor is stalled until all previous memory operations are completed. The instruction which was modified prior to the FLUSH instruction is guaranteed to execute properly after the FLUSH has been completed.

Cache consistency transactions use physical addresses, so the *Ptags* are consulted for address comparison. The instruction cache does not allow writes, so it never becomes an owner of data. Since the instruction cache never becomes an owner, it never needs to transmit its contents back to the system bus. All instruction cache snoop hits are handled by invalidating the appropriate cache entries. This includes snoop hits generated by transactions generated by load and store operations on the *same* processor.

In MBUS mode, snoop hits caused by CR operations respond by asserting the MSH_ signal, and do not invalidate.

4.7.6. Instruction Cache Diagnostics & Controls

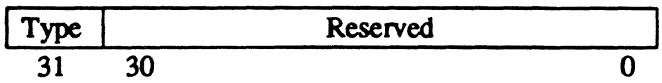
This section describes low level diagnostic and control interfaces to the instruction cache.

4.7.6.1 Instruction Cache Flash Clear

ASI=0x36 - Instruction Cache Flash Clear.

The entire instruction cache can be invalidated, or all the Lock bits can be cleared, by issuing a store alternate with the ASI value 0x36. The store must be a word operation, all other data sizes provoke a data_access_exception. The data issued by the store operation is ignored. The most significant bit of the address determines the type of operation.

The Address format is:



MRU = Most Recently Used bit, indicates history. If Type=0, all Valid and MRUs bits are cleared in the Ptags and Stags respectively. If Type=1 all Lock bits in the Instruction Cache Stags are cleared. All reserved bits are ignored, but should be set to zero. *Flash clear* operations should always be used before enabling the instruction cache.

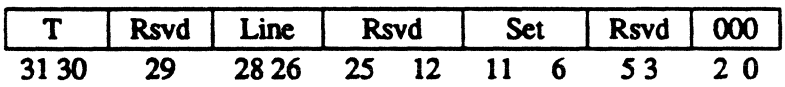
4.7.6.2 Instruction Cache Tags

ASI=0x0c - Instruction Cache Tags

Instruction cache tags are accessible to read and write using ASI value 0x0c. This direct access capability is provided mainly for diagnostic purposes.

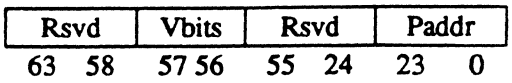
A physical tag (Ptag) is associated with each cache line and the *set tag*, Stag, is associated with each set. The Ptags and the Stags are accessed as double-words. All other data sizes provoke a data_access_exception. The state of the cache tags is not affected by watchdog or hardware reset.

The tags are addressed as pairs, since each instruction cache line has both a Ptag and Set tag. The Address format is:



Rsvd	Reserved. These bits are ignored.
Set	Selects which of the instruction cache's 64 sets is referenced.
Line	Selects one of the 5 lines in a Set (0-4). Line 5-7 generate data_access_exception.
T	Type of tag: Stag (T=1) and Ptag (T=2), while T=0 or T=3 generate data_access_ exception. Tags may only be accessed as a double word operation - all other data sizes will result in data_access_exception.

The Ptag format is:



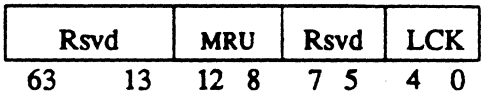
The various bit fields have the following meanings:

Paddr: Physical Address bits [35:12].

Rsvd: Reserved. Read as zero and ignored on a write.

Vbits: Valid bits for the two half-lines. Bit 56 is the valid bit for the low-order (A[4]=0) 32-byte half-line and bit 57 is the valid bit for the high-order 32 byte half-line. These bits indicate if the corresponding half-line is valid. When these bits are cleared, the contents of the corresponding sub-block have no meaning. If at least one of the two bits is set, the selected Ptag is valid. When they are both cleared, the Ptag has no meaning. At power-on reset (see section 4.3.1) both valid bits are undefined.

The Stag format is:



The various bit fields have the following meanings:

Lock: Lock bits. There is one Lock bit per line which can be used to "pin" a block inside the cache. The position of the bit determines to which line it is associated with. Bit[0] is fixed to zero, and ignores write. Bit[1-4] lock Line[1-4] respectively. When a Lock bit is set to one, the corresponding line will not be displaced by the replacement algorithm.

MRU: Most Recently Used. This bit field indicates which line of the set has been used the most recently. The position of the bit in the field determines to which line it is associated with. Bit[8-12] correspond to Line[0-4] respectively. This bit field is updated by the hardware replacement algorithm and is used to select a victim when necessary.

Rsvd: Reserved. Read as zero and ignored on a write.

4.7.6.3 Instruction Cache Data

ASI=0x0d - Instruction Cache Data.

Instruction cache data is accessible in read and write mode with the ASI value 0x0d. This direct access capability is provided mainly for diagnostic purposes. The lines are only accessed as double-words. Other data sizes provoke a data_access_exception. Instruction cache data is not affected by watchdog or hardware reset. Flash clear does not affect instruction cache data. The Address format is:

Rsvd	Line	Rsvd	Set	DblWrd	000
31 29	28 26	25 12	11 6	5 3	2 0

Rsvd Reserved. These bits are ignored.

Line Designates which of the instruction cache's 5 lines (0-4) is referenced. If line 5-7 is requested, a data_access_exception is generated.

Set Selects one of 64 sets of the cache.

DblWrd Selects the double word within the cache line.

4.8. Data Cache

The *Viking* internal data cache is a 16 K-byte, physical address cache. It is organized as a 4-way set-associative cache of 128 sets. The line size is 32 bytes. There is no sub-blocking.

The data cache is physically addressed. Virtual addresses are translated by the MMU before accessing the data cache. The requirements for a cache hit are: bits [35:12] of the physical address must match the physical tag bits [23:0] and the valid bit must be set.

The cache is enabled by the MCNTL.DE bit (See section 4.11.11.1). The cache is disabled at power-on reset, and remains disabled until the DE bit is set.

At reset the contents of the data cache are undefined. It is the responsibility of the software to initialize the data cache by resetting the valid bits (using a flash clear). After a Watchdog Reset the contents of the data cache are unmodified.

4.8.1. CC and MBUS Modes (Write-Through and Copy-Back)

The MB bit in MMU Control register (MCNTL.MB) is a read-only indicator of whether *Viking* is operating in CC or MBUS mode. (See section 4.11.11.1). This bit is determined by the state of the CCRDY_ pin at reset. When *Viking* is in CC mode, the data cache operates in a write-through fashion. In MBUS mode, the data cache operates as a copy-back cache with write allocation.

In CC mode, all stores are immediately written to the store buffer, and the store buffer will send them out to memory (written through) as soon as resources are available, and the data cache does not write allocate. (A write miss does not cause a read to occur before the write is completed.)

In MBUS mode, cache lines that have been modified by the processor are written back to memory only when necessary (due to replacement or snoop reads). In this mode, the data cache operates with write-allocation, when a store miss occurs, the data cache will allocate a line, bring in the missing data from memory, and write that data into the cache line.

4.8.2. Cacheability

Data cacheability is determined according to table 4-7 below. Data returned from MMU table walk references will never be cached internally, although they may appear in the internal cache after having been referenced by page table manipulation software. In this case, the coherence protocol will invalidate the internal copy. In MBUS mode, however, reads and writes initiated by the table walk hardware do not snoop the data cache, hence software must never allow page tables to be cacheable, to ensure consistency. See section 4.11.11.1 for description on page table cacheability, table walk access, etc.

> Important Note:
>
> In MBUS mode, page tables must be accessed through *non-cacheable* memory space to ensure consistency. In CC mode, page tables should be accessed using *cacheable* space, to improve performance.
>
> In MBUS mode, the MCNTL.TC bit should be set to zero. In CC mode, the MCNTL.TC bit should be set to one.

Table 4-7 *Data Cache Cacheability*

Translation Mode	DE=0, AC=X	DE=1, AC=0	DE=1, AC=1
MMU Disabled BT=X, EN=0	Not Cached	Not Cached	Cached
MMU Enabled BT=X, EN=1	Not Cached	C=1:Cached	C=1:Cached
		C=0:Not Cached	C=0:Not Cached
MMU Transparent	Not Cached	Not Cached	Cached

BT is the Boot Mode bit of the MMU control register.
EN is the MMU Enable bit of the MMU control register.
DE is the data cache enable bit of the MMU control register.
AC is the Alternate Cacheable bit of the MMU control register.
C is the Cacheable bit kept in each Page Table Entry. X is *don't care*

Generally, references are non cacheable when the data cache is disabled. When the cache is enabled, the C bit from the MMU controls cacheability. For "special" accesses (like MMU passthrough/bypass transactions), which do not have a corresponding C bit, the AC (alternate cacheable) bit is used to determine cacheability. For MMU table walk references, the TC (table walk cacheable) bit will indicate *external* cacheability, since table walk data is never cached internally.

4.8.3. Data Cache Replacement Policy

A limited history LRU algorithm is used to determine which lines in the data cache will be replaced when necessary. The data cache maintains a history and lock bit for each line in the cache. These bits reside in the set tag. Whenever a memory reference hits in the cache, the history bit is written to one. At the time this bit is being written, all the other history bits are checked. If all the *other*

history bits, logically ORed with their respective lock bits are already asserted, then they are all *cleared*. The result is that all four bits are never set at the same time, and that the last used entry will *always* be set. When this transition occurs, the accumulation of history begins again. In this way, a limited record of the most recently used members of a set can be kept.

When a LD instruction demands data that is not in the data cache, that data is brought in from external memory and forwarded to the instruction pipe. Simultaneously, the new data is placed in the cache. This requires that any old data be displaced from the cache. Since the data cache is 4-way set associative, there will be four choices for which line to replace.

The set tag is examined to evaluate the history and lock bits. The replacement logic first examines the lock bits. Since line 0 cannot be locked, if all other entries are locked it will always be selected for replacement, regardless of the state of the history bits.

If there is more than one unlocked line, the replacement logic uses the history bits to determine which line should be replaced. If there is a line with a history bit that is set to 0, it will be selected as the victim. The ordering of *Viking* cache lines starts with the big-end, hence for the data cache it fills starting line 3, 2, 1, then 0.

4.8.4. Snoop Hits and Lock bits

See section 4.7.3 — *Snoop Hits and Lock bits*

4.8.5. Data Cache Consistency

In a multiple processor system a mechanism must exist to keep local caches consistent with each other and with main memory. The *Viking* processor uses a protocol that is implemented largely in hardware to achieve this. Part of this protocol involves snooping the *Viking* address bus. All addresses that are snooped are compared with the cache tags in the instruction cache and the data cache. A *snoop hit* occurs if the address presented on the bus matches the physical address tag in the appropriate cache entries (and MCNTL.SE is enabled).

The action taken on a snoop hit depends on whether the data cache is operating in MBUS or CC mode. The purpose of snooping is to make sure that the contents of the data cache are consistent with external caches and main memory. In CC mode, preserving this consistency is fairly simple. All that has to be done is to invalidate an entry in the cache if some other processor writes to this location. In MBUS mode, the actions taken on a snoop hit are more complicated, including responding to the bus transaction with the current cache contents. Both the data cache and the store buffer snoop for incoming transactions that request data invalidation. The data cache responses on external transactions are listed in the following table:

Table 4-8 *Data Cache Snoop Mechanism (MBUS mode)*

External Transaction Being Snooped	Response in Cache	Resulting Action
R, W	Don't Care	No action
CR,CRI,CI,CWI	Miss	No action
CR	Hit, not owned, not shared	Set shared bit
CR	Hit, not owned, shared	No action
CR	Hit, owned, not shared	Copy out data, set shared bit
CR	Hit, owned, shared	Copy out data
CRI,CI,CWI	Hit, not owned, not shared	Invalidate entry
CRI,CI,CWI	Hit, not owned, shared	Invalidate entry
CRI	Hit, owned, not shared	Copy out data, invalidate entry
CWI,CI	Hit, owned, not shared	invalidate entry
CRI	Hit, owned, shared	Copy out data, invalidate entry
CI,CWI	Hit, owned, shared	invalidate entry

CR=Coherent Read; W=Write (non-coherent);
CRI=Coherent Read and Invalidate; R= Read(non coherent)
CWI=Coherent Write and Invalidate; CI=Coherent Invalidate
(Refer to the SPARC MBUS Specification for definitions of these terms).

Table 4-8 shows the responses of the data cache to various MBUS transactions depending on the state of the line in the cache that is accessed. This status is held in the cache tag in the *shared* and *dirty* bits (Note: the dirty bit being set indicates ownership, *dirty* bit is another name for *owned* bit). The *shared* bit is set in the cache tag if the data is also present in another processor's cache. The *dirty* bit is set if the data has been modified by the processor, and the changes have not yet been written out to memory. The *dirty* bit will only be set if the cache is in MBUS mode.

4.8.6. Data Cache Diagnostics and Control

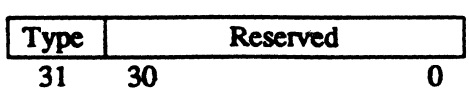
This section describes low-level diagnostic and control operations for the data cache.

4.8.6.1 Data Cache Flash Clear

ASI=0x37 - Data Cache Flash Clear.

The entire data cache can be invalidated (valid bits cleared), or all the Lock bits can be cleared, by issuing a store alternate with the ASI value 0x37. The store must be a word operation, all other data sizes cause a data_access_exception. The data issued by the store operation is ignored. The most significant bit of the address determines the type of operation.

The Address format is:



If Type=0, all valid and MRU bits are cleared in the Ptags and Stags. If Type=1 all Lock bits in the data cache Stags are cleared. All reserved bits are ignored, but should be zero.

4.8.6.2 Data Cache Tags

ASI=0x0e - Data Cache Tags.

The data cache tags are accessible in read and write modes via the ASI value 0x0e. This direct access capability is provided for diagnostic purposes.

The Ptags and Stags are accessed as double-words. Other data sizes generate a data_access_exception. The state of the cache tags is not affected by watchdog or hardware reset. *Flash clear* operations should always be used before enabling the data cache.

A physical tag (Ptag) is associated with each line, and a set tag (Stag) is associated with each set. The Address format is:

T	Rsvd	Line	Rsvd	Set	Rsvd	000
31 30	29 28	27 26	25 12	11 5	4 3	2 0

T: Type of tag: Stag=1, Ptag=2. Type 0 and 3 generate data_access_exception. Tags may only be accessed as a double word operation - all other data sizes will result in data_access_exception.

Rsvd: Reserved. These bits are ignored.

Line: Selects which of the data cache's four lines (0-3) is referenced.

Set: Indexes into the tag array of the cache to select one of the 128 sets.

The Ptag format is:

Rsvd	Valid	Rsvd	Dirty	Rsvd	Shared	Rsvd	Paddr
63 57	56	55 49	48	47 41	40	39 24	23 0

The various bit fields have the following meanings:

Rsvd: Reserved. Read as zero and ignored on a write.

Valid: Valid bit. This bit indicates that the line and its associated tag are valid. When this bit is cleared, the tag and the data contained in the line have no meaning. At reset valid bits are undefined.

Dirty: Dirty. This bit indicates that the line has been modified by one or more writes. If the cache is in write-back mode and this bit is set, the line is copied back into main memory (or a second-level cache) when it is displaced.

Shared: Shared. This bit indicates that the line is "shared" by the cache. Thus, writes to this line must be propagated to memory.

Paddr: Physical Address bits [35:12].

The Stag format is:

Rsvd	MRU	Rsvd	LCK
63 12	11 8	7 4	3 0

The various bit fields have the following meanings:

Lock: Lock bits. There is one Lock bit per line which can be used to "pin" a block inside the cache. The position of the bit determines the block it is associated with. Bit[0] is fixed to zero, and ignores write. Bit[1-3] lock Line[1-3] respectively. When a Lock bit is set to one the corresponding line will not be displaced by the replacement algorithm.

MRU: Most Recently Used. This bit field indicates which line of the set has been used the most recently. The position of the bit in the field determines the block it is associated with. Bit[8-11] correspond to Line[0-3] respectively. This bit field is updated by the replacement algorithm and is used to select a victim when necessary. In the usual mode, there is only one bit set to one.

Rsvd: Reserved. Read as zero and ignored on a write.

4.8.6.3 Data Cache Data

ASI=0x0f - Data Cache Data.

Data in the data cache are accessible in read and write modes via the ASI value 0x0f. This direct access capability is provided to assist diagnostics. Flash clear does not affect cached data. The lines are accessed as double-words only. Other data sizes provoke a data access exception.

The Address format is:

Rsvd	Line	Rsvd	Set	DblWrd	000
31 28	27 26	25 12	11 5	4 3	2 0

Rsvd Reserved. These bits are ignored.

Line Designates which of the data cache's four lines (0-3) is referenced.

Set Indexes into the data array of the cache to select a set.

DblWrd Selects the double word within the cache line.

4.9. Data Prefetching

Viking supports data prefetching primarily to increase the performance of numerical floating point intensive applications. Usually, these operate on large sequential data arrays, which easily overflow *Viking*'s 16 KByte on-chip data cache. Prefetching from these arrays greatly reduces on-chip cache misses in these iterative programs. This effect, when combined with store buffer block collection can increase performance of certain applications significantly.

Prefetching is enabled in software, by setting the MCNTL.PF bit. The prefetcher can be used *only* in CC mode. While in MBUS mode, the MCNTL.PF bit is ignored.

Once enabled, the *Viking* prefetcher monitors data cache load misses. Consecutive load misses to two consecutive cache blocks will generate an additional read access to prefetch the next block. The prefetch will not be issued until the store buffer is empty. Prefetched data is stored in the 32-byte prefetch buffer. If subsequent loads hit the prefetch buffer, data is forwarded from the buffer with no delay, and another prefetch is issued. The next prefetch is only issued when all doublewords in the current buffer have been referenced (the order is not significant). All prefetching is based on *physical addressing*.

In general, the prefetcher is effective when data is being referenced by a series of LD or LDD instructions (nominally 4 LDDs), followed by a series of ST operations. The series of store operations allows the prefetch to complete. The ST operations accumulate in the buffer and become write-burst transactions on the bus, which provides additional performance improvement.

Although all prefetched data is cacheable, it is *not* cached in *Viking*'s data cache. This can avoid cache thrashing and also simplifies operation. As a result, the prefetch buffer maintains coherence of the fetched block. To accomplish this, the prefetch buffer snoops *Viking*'s write misses as well as off-chip system writes. If either hits the prefetched line, the line is invalidated. As a result, the prefetcher cannot contain any modified information. In addition, prefetch addresses are matched with the store buffer before being issued, as for all read accesses. The data cache is also checked, to avoid prefetching data that's already present on *Viking*.

Prefetches cannot cross page boundaries. Although large data arrays will probably appear as large contiguous memory blocks, this cannot be guaranteed in physical memory. Inadvertent block prefetches from non-cacheable I/O space must be prevented. However, the prefetcher will usually encounter only one cache miss before resuming prefetching in this case, since it will have correctly guessed the miss address when the pages are contiguous.

The prefetch buffer is *not* directly accessible through diagnostic ASI space.

4.10. Control Space Access

ASI 0x02 allows access to the information in an external device, nominally an external cache controller. The information may include cache controller (CC) registers, the external cache and its directories. These accesses are non-cacheable, regardless of MCNTL.AC indication. Examples of address generation and data accessed can be found in the *Viking* Cache Controller Specification document.

Data references to this control space access may be any size. External system hardware is responsible for proper data alignment. Any faults will be reported as data_access_exception.

4.11. Memory Management Unit (MMU)

The *Viking* processor implements a 64-entry fully-associative MMU compatible with the SPARC Reference MMU (SRMMU or MMU) Specification. The set of 64 entries is called the Page Descriptor Cache (PDC) or Translation Lookaside Buffer (TLB).

The MMU translates 32-bit virtual addresses into 36-bit physical addresses. The mapping is done in units of 4 K-byte page, 256 K-byte segment, 16 Mbyte region or 4 G-byte context.

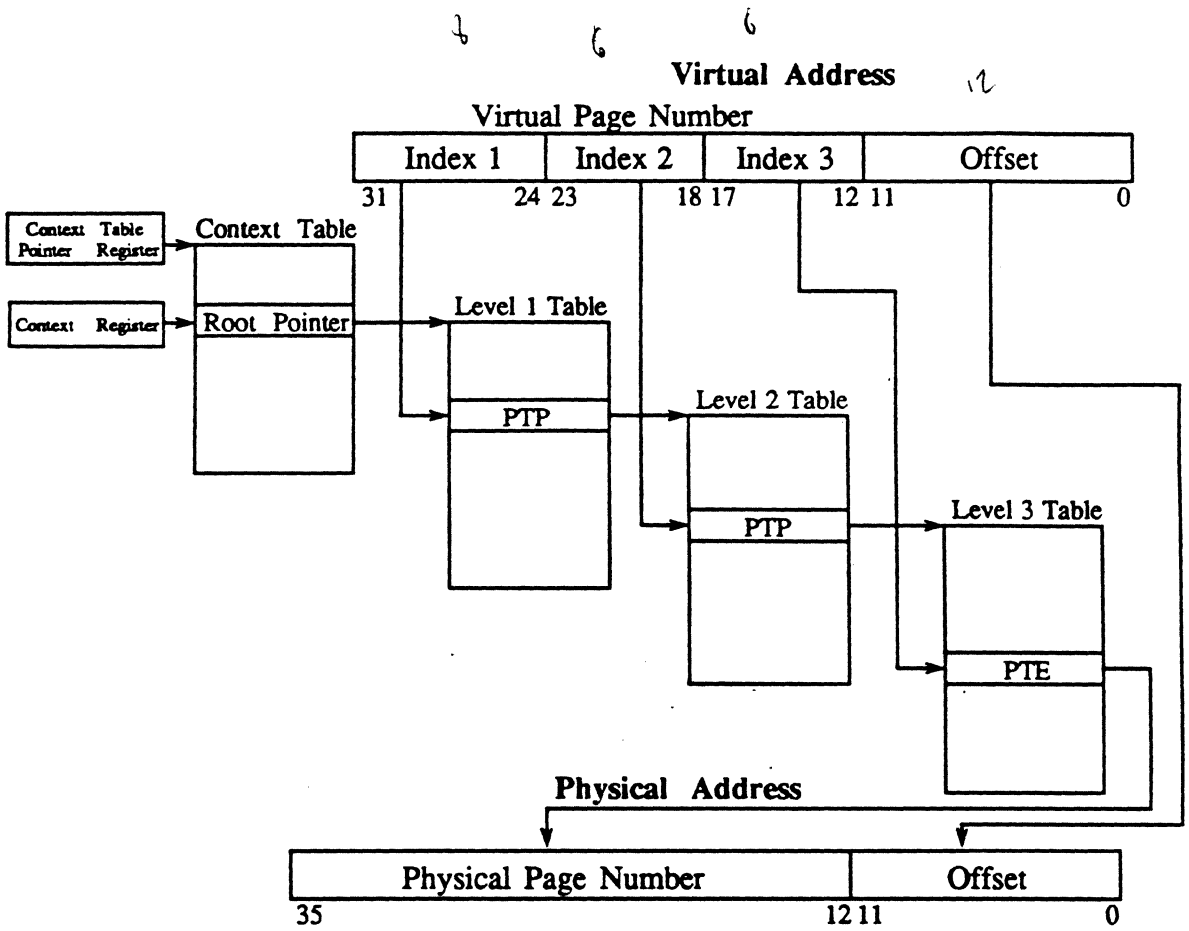
4.11.1. Address translation

This section briefly describes the software view of memory mapping using the *Viking* MMU. For background, refer to the SPARC Reference MMU Specification in the SPARC Architecture Manual.

Virtual and physical addresses are composed of an offset within the page and respectively a Virtual Page Number (VPN) or a Physical Page Number (PPN).

The MMU translates 32-bit virtual addresses and 16-bit context numbers into 36-bit physical addresses by accessing up to four levels of page tables in memory. Normally, this translation is cached in the on-chip 64-entry TLB. When the translation entry is missing from the TLB, the MMU table walk hardware automatically retrieves the translation from the page tables in memory. Figure 4-3 describes the full structure of these page tables.

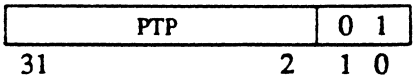
Figure 4-3 *Address Translation Utilizing Four Levels of Page Tables*



Each virtual address space is identified by a context number which is kept in the Context register. Virtual addresses kept in the TLB are "tagged" with a 16-bit Context Number. The effective size of the context register is variable between 10 and 16-bits. This allows the size of the page tables to be reduced. See section 4.11.11.2 — *Context Table Pointer Register (MCTP) and Context Register (MC)* for further details.

The page tables can contain Page Table Pointers (PTP) or Page Table Entries (PTE). A PTE is distinguished from a PTP by the two low order bits of the table entry (see the ET field description below). A PTP contains the physical address of the next page table level while a PTE contains the physical address of the page with its access rights.

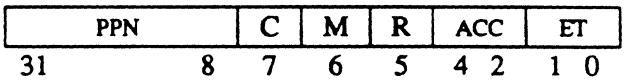
The Page Table Pointer format is:



The value of 1 in the least significant two bits indicates that the entry type (ET) is a page table pointer.

> Important Note:
> The page tables must be aligned on boundaries equal to their sizes. Low order bits of the PTP field must be zero. PTPs must point to tables aligned to their natural size.

The Page Table Entry format is:



The various bit fields have the following meanings:

PPN: Physical Page Number. High order 24 bits of the 36-bit physical address. If the PTE maps a 256 K-byte segment, 16 M-byte region, or 4 G-byte context, the lower 6, 12 or 20 bits respectively of the PPN are ignored.

C: Cacheable. If this bit is set to one, the page is cacheable in the *Viking* internal (and external) caches. If it is zero, the page is not cacheable. *Viking* asserts the CCHBL_ pin for transactions involving virtual addresses with the C bit set.

M: Modified. When a page is accessed for writing, and the modified bit is not set, the MMU sets the modified bit in both the TLB, and the in-memory page table entry.

R: Referenced. This bit is set to one by the hardware when the page is accessed (on a read or a write) and the PTE is missing from the Page Descriptor Cache. Both copies are set.

> Important Note:
> See section 4.11.3.1 — *MMU R&M Updates* and Section 4.11.3 — *Referenced and Modified bits* for a description of the interaction between hardware and software access to and modification of the reference and modified bits.

ACC: Access Permissions. This bit field is encoded as follow:

ACC	Permissions	
	User	Supervisor
0	Read Only	Read Only
1	Read/Write	Read/Write
2	Read/Execute	Read/Execute
3	Read/Write/Execute	Read/Write/Execute
4	Execute Only	Execute Only
5	Read Only	Read/Write
6	No Access	Read/Execute
7	No Access	Read/Write/Execute

The MMU checks if an access is authorized according to the mode in which the instruction is executed (i.e. user or supervisor) and the type of access (instruction or data reference). If there is a violation of the permissions a data or instruction access exception is incurred. For more details on ACC vs AT (Access Permission and Access Type) refer to 4-12 — *Access Permission vs Access Type*.

ET: Entry Type. This field is used to distinguish a PTE from a PTP and to indicate if a table entry is valid. The encoding is:

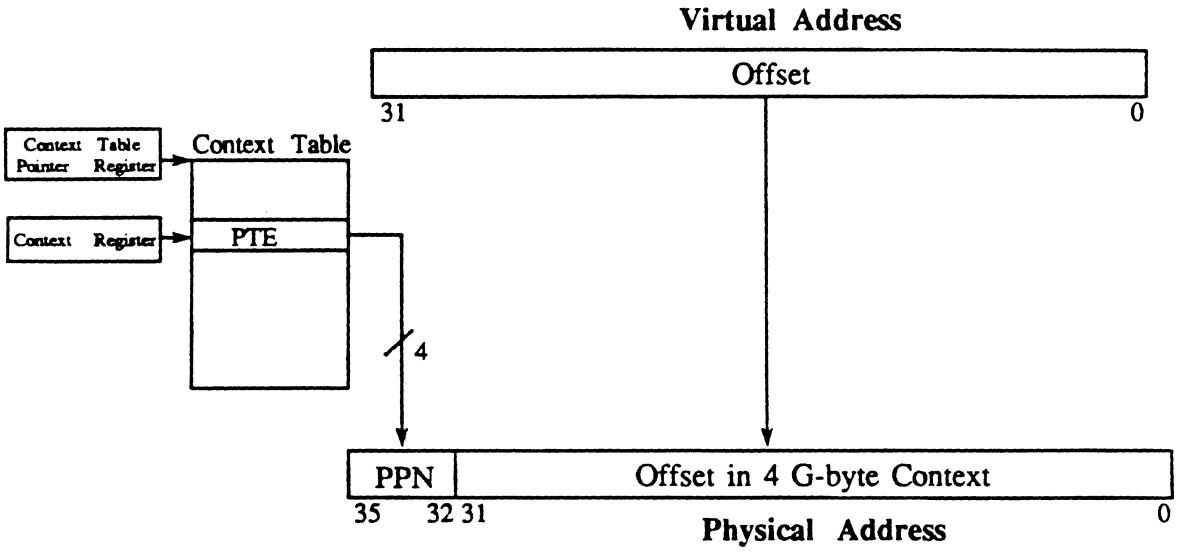
ET	Entry Type
0	Invalid
1	Page Table Pointer
2	Page Table Entry
3	Reserved

4.11.2. Linear Mapping

The MMU supports mapping sizes larger than the page size. This is done by configuring an entry in the Context Table, Level 1 Table or Level 2 Table as a PTE (ET=2).

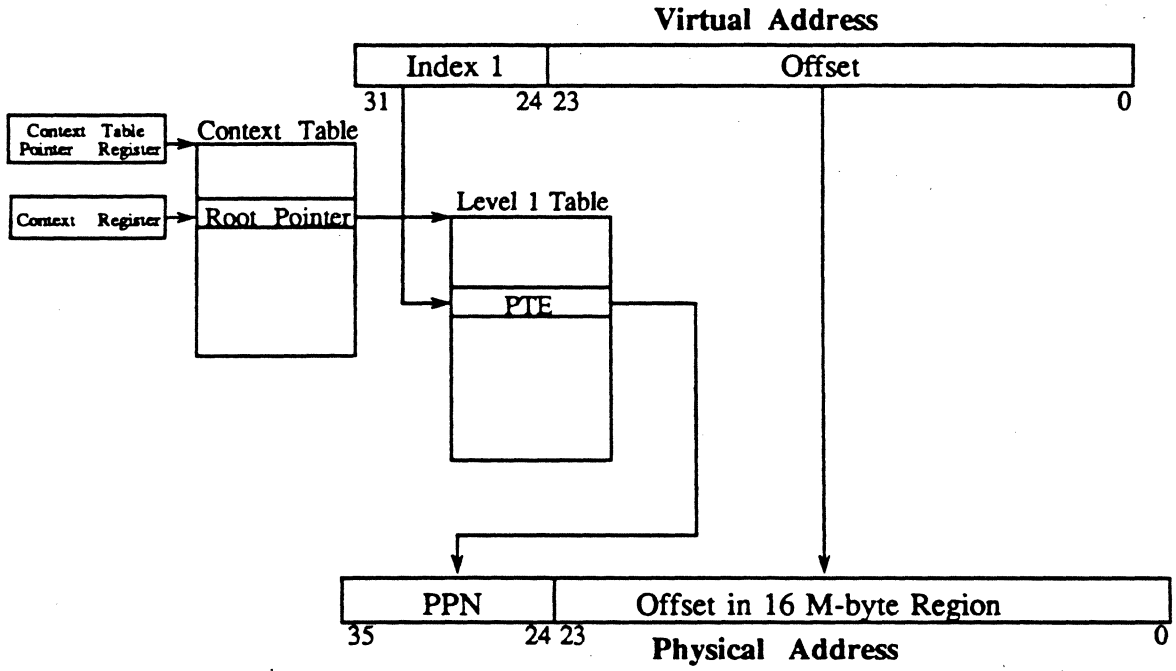
If a PTE is found in the Context Table, the virtual to physical mapping is done as indicated below for a 4 Gigabyte context.

Figure 4-4 *Address Translation With Maximum Page Size*



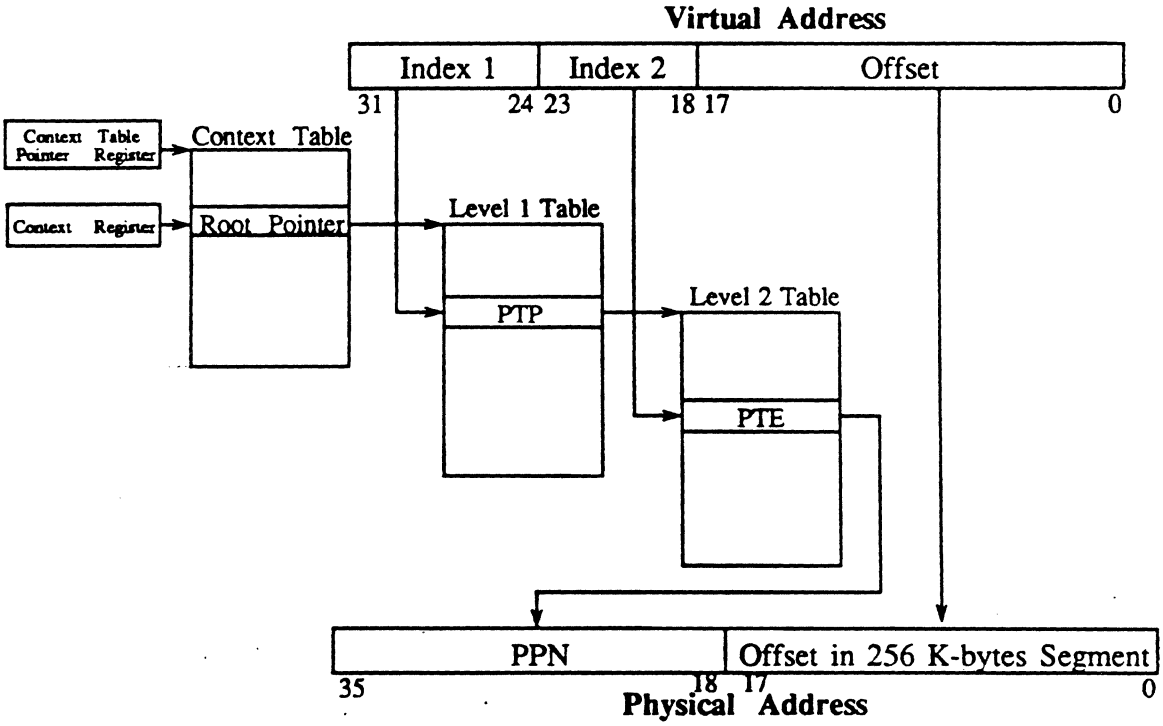
If a PTE is found in a Level 1 Table, the virtual to physical mapping is done for a 16 Megabyte region.

Figure 4-5 *Address Translation With 16 MB Page*



If a PTE is found in a Level 2 Table, the virtual to physical mapping is done for a 256 Kilobyte segment.

Figure 4-6 *Address Translation With 256 KB Page*



Whenever the MMU finds a PTE in an entry, it stops the table walk and stores the translation in the TLB. The *Viking* MMU Page Descriptor Cache is implemented by a CAM (content addressable memory) array which can simultaneously match the address tag fields corresponding to the four mapping sizes defined by the SPARC Reference MMU.

4.11.3. Referenced and Modified bits

The referenced bit (R) of a PTE is set to one whenever its page is accessed by the MMU during miss handling or when an entire probe is initiated. If the referenced bit is already set, it is not set again.

The Modified bit is checked when the PTE is accessed as a result of the execution of a store instruction. If the Modified bit is clear in the MMU entry, it is set to one and the copy of the PTE in main memory is also set to one.

The Referenced and Modified bit must be updated in main memory (e.g. shared memory image) in a manner which guarantees their consistency in a multiprocessor environment. For a discussion of possible algorithms see section 4.5.5 —

Page Table Memory Operations.

4.11.3.1 MMU R&M Updates

Occasionally, the MMU must modify a Page Table Entry (PTE) in memory, as memory pages are referenced or modified by *Viking*. These writes are required by the reference MMU architecture to be synchronous; thus, they block the execution pipeline. Also, they also force a Store Buffer copy-out, to preserve the sequence of writes. This copy-out is initiated when the memory reference is present at the E0 stage of the execution pipeline.

If an exception occurs on the Store Buffer copy-out caused by an R&M update, the R&M update operation is *not* completed. A data store exception is taken, which disables the Store Buffer at once. The Load, Store, or fetch which caused the R&M update, will be restarted when the CPU returns from the Store Buffer trap handler; this in turn will eventually restart the R&M update.

The table walk hardware within *Viking* will use a standard write operation when setting *both* referenced and modified bits in memory. If *only* the referenced bit must be set, atomic memory transactions will be used to ensure that another processor is not simultaneously attempting to set *both* bits. If the processor did not use this protection, it would be possible to overwrite an PTE with both R and M bits set, with another updated PTE with only the R bit set. This is clearly illegal and is prevented by using SWAP transactions to do R-bit only updates. Note also that the only combinations that *Viking* will ever write back to the PTE are (R=1,M=0), and (R=1,M=1).

Using SWAP for R-bit updates is essentially the only page table consistency algorithm implemented in hardware. All other cases of page table consistency must be implemented in software as described in section 4.5.5 — *Page Table Memory Operations*.

4.11.4. TLB Replacement Policy

The Memory Management Unit has 64 TLB entries. The replacement logic is used to select a TLB entry to replace when a new PTE is brought in to the MMU for a TLB miss. If one or more of the 64 entries is invalid, then the replacement logic selects this entry, starting its selection from entry '0' and progressing to entry '63' in case of multiple invalid entries. When all the entries are valid the replacement logic selects the entry to be replaced using the information available in the *used* bits. The algorithm used is a limited history LRU policy.

Each TLB entry has a used bit. Initially when a demap-all operation is done to invalidate all TLB entries all used bits get cleared. The used bit is then set for any valid TLB entry which has a TLB hit. When all entries have their used bits set, then all used bits, except the last one to be set and those which are locked, are cleared. This is the case where all past history is lost. To select a entry to be replaced (when all TLB entries are valid) the first entry starting from entry '0' for which the used bit is not set will be chosen. This represents the least recently used entry (based on the limited history available, since a TLB hit causes the used bit to be set).

When a entry is invalidated or flushed from the TLB its corresponding used bit is cleared. In addition to the used bits, the replacement policy also checks the

corresponding lock bit (one per TLB entry). If a lock bit for an entry is set, then regardless of the used bits that entry will never be replaced. An invalid entry with its lock bit set can still be replaced, and the newly written entry becomes locked, since the lock bit remains set. If all entries have their lock bits set no replacement takes place and the newly brought in PTE is not stored in the TLB. Since a translation finishes only after a table-walk operation has completed, this causes an infinite table-walk loop. Thus one should **never** lock all the entries in the TLB.

Important Notes:

Setting all lock bits in the TLB can lead to *deadlock* and is not recommended.

Allowing the lock bit to be set for invalid entries is also not recommended. It can lead to inconsistent operation. Since these entries remain locked after a new entry is written into them by a table walk, that entry will remain in the TLB, even after a DeMap operation which should invalidate it.

It is recommended that lock bits be set only in conjunction with explicit writes to that TLB entry by supervisor software. Note also that the lock bits are cleared by hardware reset.

4.11.5. Other Cached Entries (Root and Level2 PTP2 cache)

To reduce the time required for each table-walk to access the PTE's, the *Viking* MMU caches two special pointers, PTP0 (PTP level 0) and PTP2 (PTP level 2).

PTP0 is the root-pointer (level-0 pointer) for the process in execution. On every context switch this cached entry is invalidated, and the first table-walk for the new process will be used to cache in the new root pointer. Cacheing the root pointer saves the MMU from performing a level of table walk for each TLB miss for that particular context. If a context were to execute for a long time this could mean a considerable saving of cycles. The root-pointer entry is qualified by a valid bit and implicitly corresponds to the context in the Context register. The valid bit for this entry is cleared on context register write, context table pointer write, demap-all and demap for a context which matches the context register value.

To optimize MMU table-walk, the *Viking* MMU also caches one level-2 PTP. A TLB miss with this level-2 PTP would require just the new level3 PTE to be fetched rather than the entire three level table-walk fetch operation. A second level PTP requires a virtual tag and a valid bit for this cached entry. The context is implicitly assumed to be the value of the context register. This second level cached PTP entry is used only for table-walks, R & M bit updates and probe-entire operations. Other operations still go through the three level table-walk mechanism. The valid bit for this entry is cleared on context register write, context table pointer write, on a table-walk operation not using the cached PTP (in this case a new entry will be written), demap-all, demap for a context which matches the context register value, and level-1 and level-2 demaps for which this PTP has a match. The PTP entry is cached on a new table-walk operation and is not cached if the level-2 entry is a PTE.

4.11.6. Hit Criteria

The criteria for a hit in the MMU are defined as follows:

Mapping Size	Matching Criteria
4 KB	V=1 and VAddr[31:12]_equal and (ACC=6-7 or Context_equal)
256 KB	V=1 and VAddr[31:18]_equal and (ACC=6-7 or Context_equal)
16 MB	V=1 and VAddr[31:24]_equal and (ACC=6-7 or Context_equal)
4 GB	V=1 and ACC=6-7 or Context_equal

VAddr[31:xx]_equal indicates that the corresponding bit fields of the virtual address issued by the processor and the virtual address contained in an MMU Vaddr tag entry match.

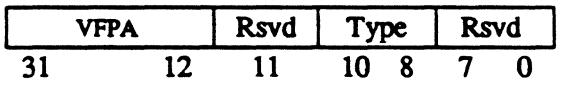
Context_equal means that the contents of the Context register and the Context field in the MMU entry match. Note that context is not compared for any page classified as a *supervisor* page, by the ACC field being either 6 or 7. In this manner, supervisor pages are present in *all* contexts simultaneously.

In all cases, the entry must be valid (V=1).

4.11.7. MMU Probe and Demap/Flush

The ASI value 0x03 is used to invalidate or probe entries in the MMU. An invalidation of an MMU entry is also commonly called a flush or a demap. These three terms are used indistinctly in this document.

A probe is done with a load alternate while a demap/flush is done with a store alternate instruction. The data of the load or store alternate is ignored. The address format for both cases is:



The various bit fields have the following meanings:

VFPA: Virtual Flush or Probe Address. According to the type of flush or probe not all 20 bits are significant.

Type: This field specifies the extent of a demap (mapping size) or the level of the entry probed.

Rsvd: Reserved. These bits are ignored. They should be zeros.

4.11.7.1 MMU Probe

The different types of MMU probes which can be performed and the corresponding returned data are:

Type	Probe Object	Returned Data
0 (Page 4 KB)	Level 3 Entry	Level 3 PTE or 0
1 (Segment 256 KB)	Level 2 Entry	Level 2 PTE/PTP or 0
2 (Region 16 MB)	Level 1 Entry	Level 1 PTE/PTP or 0
3 (Context 4 GB)	Level 0 Entry	Level 0 PTE/PTP or 0
4 (Entire)	Level n Entry	Level n PTE or 0
5-7 (Reserved)		

For all the probe operations the MMU (TLB) is accessed first and if a translation which complies with the matching criteria is found, the cached PTE is returned. The PTE is returned in the memory format with R=1 and ET=2 since only PTEs are cached in the *Viking* MMU. The matching criteria for a successful probe in the MMU are:

Type	Matching Criterion
0 (Page 4 KB)	V=1 & VAddr[31:12]_equal & Context_equal & LVL=3
1 (Segment 256 KB)	V=1 & VAddr[31:18]_equal & Context_equal & LVL=2
2 (Region 16 MB)	V=1 & VAddr[31:24]_equal & Context_equal & LVL=1
3 (Context 4 GB)	V=1 & Context_equal & LVL=0
4 (Entire)	V=1 & Context_equal & ((VAddr[31:12]_equal & LVL=3) | (VAddr[31:18]_equal & LVL=2) | (VAddr[31:24]_equal & LVL=1) | (LVL=0))
5-7 (Reserved)	

VAddr[31:xx] _equal means that the corresponding bit fields of the virtual address issued by the processor and the virtual address contained in an MMU entry match.

Context_equal means that the contents of the Context register and the Context field in the MMU entry match.

If the PTE is not found in the MMU, a table walk is initiated. In this case, the data returned may not be a PTE. The returned data can be a PTP or the value 0 if an error occurs. The error cases are slightly different from the ones which may occur during a regular table walk when the MMU is processing a miss. They are detailed in the following paragraphs. If an error occurs during a probe table walk, no exception is taken, but the AT field of the MMU status register is set to 1, the FT field to 1 (Invalid Address) or 4 (Translation error) and the L field is set to the table level where the error was detected.

For a page probe, the hardware does a table walk and returns the PTE found in the level 3 table even if this level 3 entry is invalid (ET=0). If the table walk does not complete correctly because the level 3 entry accessed is not a PTE (ET=1 or 3), an intermediate level entry is not a PTP (ET = 1) or a hardware error occurs the value 0 is returned, and the FT is set to 4.

For a segment or region probe, a PTE (ET=2) or a PTP (ET=1) is returned from respectively the level 2 or level 1 entry even if it is invalid (ET=0). If the table walk does not complete because the entry accessed is reserved (ET=3), an intermediate level is not a PTP (ET = 1) or an hardware error occurs the value 0 is returned, and the FT is set to 4.

For a context probe the level 0 entry accessed is returned if it is a PTE (ET=2) a PTP (ET=1) or is invalid (ET=0). The value 0 is returned, and the FT is set to 4 if the entry is reserved (ET=3) unless a hardware error occurs.

For an entire probe, the hardware does a regular table walk and returns the valid PTE found in whichever level table. If no PTE is found because an invalid (ET=0) or reserved entry (ET=3) is accessed in an intermediate level, the level 3 entry accessed is a PTP (ET=1) or an hardware error occurs, a 0 is returned. If an invalid entry is accessed, the FT field is set to 1 in the MMU status register, in the other cases the FT field is set to 4.

When an entire probe completes successfully, the PTE accessed is loaded in the TLB and the R bit is updated if necessary. For all the other probe operations, the TLB is left unchanged and the R bit is not updated.

> Important
> *Viking* generates data_access_exception on probe types 0x5-0x7, and treats probe types 0x8-0xf identical as types 0x0-0x7 respectively.

4.11.7.2 Demap/Flush

The different types of demaps and the objects flushed from the MMU are:

Type	Flush Object
0 (Page 4 KB)	Level 3 PTE
1 (Segment 256 KB)	Level 2 and 3 PTEs
2 (Region 16 MB)	Level 1, 2, and 3 PTEs
3 (Context 4 GB)	Level 0, 1, 2, and 3 PTEs
4 (Entire)	All PTEs
5-7 (Reserved)	

The hit criteria for an MMU flush are:

Type	Matching Criteria
0 (Page 4 KB)	LVL=3 & VAddr[31:12]_equal & (ACC=6-7 or Context_equal)
1 (Segment 256 KB)	LVL=3|2 & VAddr[31:18]_equal & (ACC=6-7 or Context_equal)
2 (Region 16 MB)	LVL=3|2|1 & VAddr[31:24]_equal & (ACC=6-7 or Context_equal)
3 (Context 4 GB)	ACC=5-0 & Context_equal
4 (Entire)	None

In a multiprocessor system, distinct processors can hold copies of the same PTE in their MMUs. Therefore, when a portion of a virtual space is demapped, the flush operation must be applied to all MMUs in the system.

Depending on the system implementation, demap operations may be broadcast automatically to all processors, or may require explicit software intervention on all processors to demap the required pages. *Viking* allows for automatic demapping only when used in CC mode, whether or not this is implemented is system dependent. For example, the *Viking* cache controller chip (*MXCC*) implements system demaps in XBUS mode, but does not in MBUS mode. In direct MBUS mode, all processors must be interrupted and requested to flush their own TLBs whenever the page table is modified.

> Important Note:
>
> Software must guarantee that only a single DeMap operation is in progress at any one time across the entire system. Inconsistent operation will result if two DeMaps are received by a processor at any one time (including internal DeMap requests).

When an MMU flush has been completed by all processors (either through hardware or software), the following should be true:

All memory references concerning the virtual space(s) mapped by the flushed PTE(s), that have been issued before the demap must have been completed.

No MMU has a valid copy of the PTE(s).

All memory references to the PTE(s) itself (themselves) that were issued before the demap have been completed.

Once the system has reached this state, page tables may safely be modified and applications allowed to continue.

4.11.8. MMU Transparent Mode

The ASI values 0x20-0x2f are used to bypass the MMU for data accesses. The MMU does not translate the physical address through the Page Descriptor Cache. The virtual address is translated as follow:

ASI[3:0] → Paddr[35:32], Vaddr[31:0] → Paddr[31:0].

Cacheability of these accesses is determined by MCNTL.AC bit. Since these transactions are completely based on *physical* memory addresses they can have no effect on virtual memory components, such as the MMU R&M bits.

4.11.9. Address Translation Modes

The following table summarizes the different translation modes used by the *Viking* processor.

Translation Mode	Instruction Fetch (ASI=0x08 or 0x09)	Data Access (ASI=0x0a or 0x0b)
Boot Mode BT=1, EN=X	PA[35:28]=0xff, PA[27:0]=VA[27:0]	EN=0 -> Disabled Mode EN=1 -> Enabled Mode
MMU Disabled BT=0, EN=0	PA[35:32]=0x0, PA[31:0]=VA[31:0]	PA[35:32]=0x0, PA[31:0]=VA[31:0]
MMU Enabled BT=0, EN=1	PA[35:12] from PTE PA[11:0]=VA[11:0]	PA[35:12] from PTE PA[11:0]=VA[11:0]
MMU Transparent	Not Applicable	LDA and STA with ASI=0x20-0x2f PA[35:32]=ASI[3:0], PA[31:0]=VA[31:0]

BT is the Boot Mode bit of the MMU control register.
EN is the MMU Enable bit of the MMU control register.
PA[bit range] are the bits of the physical address.
VA[bit range] are the bits of the virtual address.

4.11.10. No-Fault Operation

The MCNTL.NF bit, when enabled, turns on *no-fault* operation. In this mode, most exceptions generally reported to the pipeline are disabled. This mode is intended for use by system software during the processing of exceptions, and during system diagnostic functions. The NF bit should *never* be set during user code execution.

Any transaction which has an error blocked by NF will complete. In the case of load transactions, the destination register will be updated with indeterminate information. In the case of stores, no registers will be updated, and the system is responsible for not modifying memory.

When operating with NF set, the success or failure of *every* memory transaction should be verified by explicitly reading the MFSR fault status register.

In general, all *normal* memory exceptions are disabled by NF. There are several types of exceptions which are not disabled by NF. The exceptions types which are not disabled are all considered *fatal* errors which are generally not recoverable and should induce error mode. The following exceptions are not disabled by NF:

Internal Error

Errors such as multiple tag matches from the caches are considered fatal internal errors. See section 4.11.11.3.5 — *Error Mode and Internal Errors*.

Control Space Error

Errors reading or writing *Viking* ASI registers are considered fatal. See section 4.11.11.3.4 — *Control Space Errors*

ASI 0x09

Supervisor instruction fetch errors cannot be disabled by NF, since the processor effectively has received an error in the instructions it needs to execute. There is no other source for instructions, an exception must be generated.

ASI 0x08

True instruction fetches (explicitly *not* alternate space read and writes) to ASI 0x08 (User Instruction space) will cause exceptions to be reported. As above, without the exception no instructions could be executed.

Important Note:

System software is responsible for properly setting and clearing the NF bit. Improper use of no fault mode can lead to undefined processor operation.

In particular, the NF bit must never be set when returning to user code execution.

4.11.11. MMU Registers

ASI=0x04 - MMU Registers.

Accesses to this ASI read and write SPARC Reference MMU control registers. These registers are all 32-bits wide, and are shown in the following table. Attempts to access them with byte, Halfword, or Doubleword operations will result in data_access_exception. Virtual address bits [12:8] are used to select individual registers, all other bits are ignored and should be 0. Any access to addresses other than defined in the table below causes a data_access_exception.

Table 4-9 *MMU Registers*

VA[12:8]	Description
0x00	Control Register (MCNTL)
0x01	Context Table Pointer Register (MCTP)
0x02	Context Register (CONTEXT)
0x03	Fault Status Register (MFSR)
0x04	Fault Address Register (MFAR)
0x13	read/write Fault Status Register
0x14	read/write Fault Address Register
0x15	Shadow Fault Status Register (MSFSR)

4.11.11.1 MMU Control Register (MCNTL)

The control register contains the general MMU control and status flags. In addition, it also contains the control flags for the instruction and data caches. The MMU control register is defined as follows:

Impl	Ver	Rsvd	PF	Rsvd	TC	AC	SE	BT
31 28	27 24	23 19	18	17	16	15	14	13

PE	MB	SB	IE	DE	PSO	Rsvd	NF	EN
12	11	10	9	8	7	6 2	1	0

Impl — Implementation number of the *Viking* chip, it is hardwired and is read-only. *Viking* fixes MCNTL.Impl = 0x0.

Ver — Version number of the *Viking* chip, typically a mask number. This field is hardwired and is read-only. *Viking* fixes MCNTL.Ver = 0x0. Mask revs are reflected only in PSR.

Rsvd — Reserved. These bits are ignored and read-only.

PF — Data Prefetcher Enable. This bit when asserted indicates that the Data Prefetcher is enabled. This is only meaningful in CC mode. This bit is ignored in MBUS mode (Data Prefetcher is disabled in MBUS) mode).

Rsvd — Reserved. This bit is ignored and read-only.

TC — Table walk Cacheable bit. When this bit is asserted, references from MMU table walks are cached in the *Viking external* cache. When this bit is deasserted, all table walk accesses are not cached in the *Viking* external cache. Table walk references are *never* cached internally. For MBUS mode, TC must be deasserted.

AC — Alternate Cacheable bit. This bit indicates whether an access is cacheable or not in the absence of the C bit in the PTE due to the MMU being disabled or in a mode where the PTE are not needed for translation. The only exception to this case is the instruction fetches in boot mode which are always non-cacheable. When this bit is clear, all memory

accesses, except table walking accesses, for which the physical address is not obtained through a PTE are not cached in the *Viking* internal caches and the External cache. If this bit is set to one, those accesses are cached.

SE — Snoop Enable. This bit enables cache snooping on the *Viking* bus when set to one. This bit must be set to enable the cache consistency mechanisms. Assertion of this bit does not affect Store Buffer snooping nor Data Prefetcher snooping, which are always enabled.

In MBUS mode, both the instruction and data caches will snoop regardless of whether they are enabled. This is necessary for maintaining consistency should any of the caches be disabled after they contain real data. Therefore, initialization code should contain a flash clear for these caches before enabling snoops at power-on reset, so that no garbage data may reside in the snooping caches. Snooping is controlled by the MCNTL.SE (snoop enable) bit.

BT — Boot Mode. When not asserted indicates normal SPARC reference MMU operation. When asserted, indicates boot mode, and the physical address generated (for instruction fetches) is formed by adding the address 0xff0000000 to the lower 28 bits of the virtual address. BT is asserted high (BT=1 is boot mode). Instruction accesses in boot mode bypass the *Viking* internal cache. Data accesses are unaffected by the boot mode.

PE — Parity Enable. If set, this bit enables parity checking (Even Parity) for each byte of data brought into the chip.

MB — MBUS Mode. This bit if set indicates that the data cache is in Copy Back Mode, else the data cache is in write-through mode. This bit is read-only, and it reflects the CCRDY_ pin.

SB — Store Buffer Enable. This bit if set enables the store buffer operations.

IE — Instruction Cache Enable. This bit if set enables the instruction cache.

DE — Data Cache Enable. This bit if set enables the data cache.

PSO — Partial Store Ordering. When asserted, the memory model is in PSO (Partial Store Ordering) mode. When deasserted, it is in TSO (Total Store Ordering) mode.

Rsvd — Reserved. These bits are ignored.

NF — No Fault Bit. When this bit is asserted, faults that occur are ignored (not reported to the processor) for ASIs 0x08, 0x0a, 0x0b, 0x20-0x2f, while the remaining ASIs including ASI 0x09, *Viking* internals and control space (ASI 0x02) will take the fault (These latter ASIs are said to 'ignore' the NF bit). Note that regardless of whether/not the fault is taken, the FSR is always updated.

EN — MMU enable. This bit enables or disables the operations of the MMU. The BT bit must be deasserted for the EN bit to indicate a MMU enable

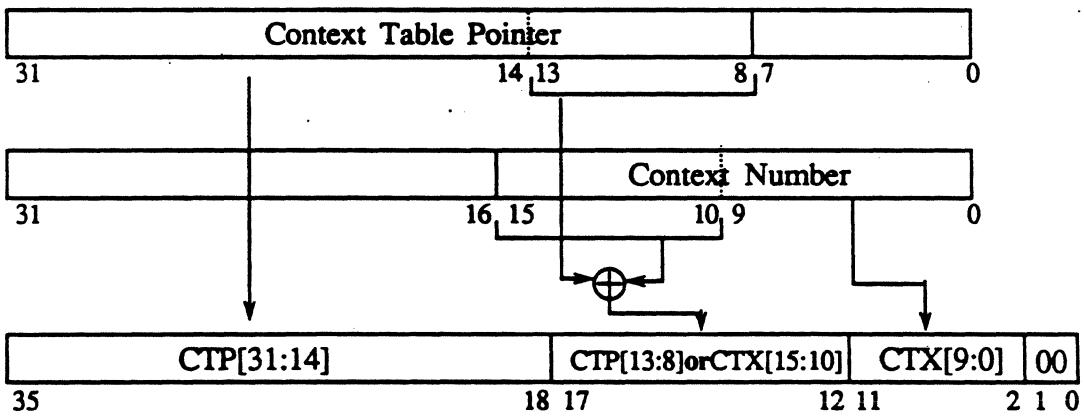
or disable, thus the BT bit asserted over-rides the EN bit for instruction fetches only. MMU Enable. When this bit is set to one the MMU is enabled. When the EN=0 and BT indicates normal operation the 4 most significant bits of the physical address are forced to zero and the 32 least significant bits are the 32 bits of the virtual address. When EN=0 and BT indicates Boot Mode, instruction fetch physical addresses are formed according to the rules given above for the BT bit, data fetch physical addresses are formed as described before.

On power-on reset, all the control bits mentioned above (except BT) are cleared.

4.11.11.2 Context Table Pointer Register (MCTP) and Context Register (MC)

The Context Table Pointer (CTP) is a pointer to the Context Table in physical memory. The Context Table contains the root page table pointers for all the contexts. The Context Register provides the offset into the Context Table to retrieve the root page table pointer for that context. In *Viking* the Context Table Pointer is a 24 bit register and the Context Register is a 16 bit register. The Physical address used to retrieve the root page table pointer is formed as shown below :

Figure 4-7 *Root Pointer Physical Address Generation*

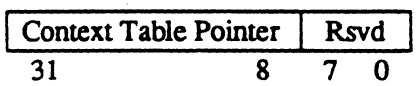


This address formation gives the kernel the freedom to trade-off number of context bits against alignment restrictions on the context table. Note, if a context of 16 bits is desired the context table must be aligned to a 256K byte boundary.

The table shown below gives the alignment requirements for the context table for different widths for the context register.

Context bits	Alignment
≤10	4K
11	8K
12	16K
13	32K
14	64K
15	128K
16	256K

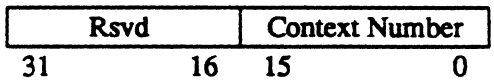
The Context Table Pointer register has the following structure:



The reserved field is ignored on a write and read as zero.

The Context register contains the displacement in the context table to access the root pointer. It defines the current virtual address space.

The Context Register has the following structure:



The reserved field is ignored on a write and read as zero.

4.11.11.3 MMU Fault Status Register (MFSR)

The Fault Status register provides information for *Viking* faults which associated with the memory system, and other internal error sources. The MFSR, along with the reported trap type are used to distinguish between the various types of errors and faults that can occur.

There are several general types of exceptions: instruction access faults, data access faults, store buffer exceptions, and internal errors.

4.11.11.3 Instruction access errors

Instruction access faults are signalled through the instruction_access_exception trap. There are several possible sources for the exception. It may be created by normal *page faults* reported by the MMU. MMU generated errors are distinguished by the encoding of the MFSR.FT field, indicating the MMU fault type. The fault may be externally generated, for bus timeouts, parity errors, etc. Externally generated faults are indicated by various bits in the MFSR that are tied to equivalent error responses on the external busses. A final source of errors is from internally detected inconsistencies. For example, a multiple tag match in the instruction cache. In this case, an error mode trap (watchdog reset) is generated, the MFSR.EM bit will be set.

4.11.11.3 Data access errors

Data access faults are signalled through the data_access_exception trap. As with instructions, there are several source for this exception. As above, page faults, external bus errors, and internal errors may all cause exceptions. Additional errors can be caused by erroneous accesses to internal ASI control spaces. These are indicated by the MFSR.CS status bit.

4.11.11.3 Store buffer errors

Store buffer errors are signalled as data_store_error traps. These errors are *deferred exceptions*. They are reported *after* the apparent completion of the instruction which caused them. When this type of error occurs, the MFSR.SB bit will be asserted. The source of the error is generally a parity, timeout, or similar error on the bus. These stores have already been successfully translated by the MMU, or a data_access_exception would have been reported. The operation can be recovered by reading the contents of the store buffer and explicitly completing the transaction using transparent MMU physical address memory references or other means. See section 4.12 — *Store Buffer* for a detailed description of store buffer errors and operation.

4.11.11.3 Control Space Errors

Any ASI operations which are known to be illegal by *Viking* will be reported as control space errors. These errors will cause data_access_exceptions, and set the MFSR.CS bit.

Any of three conditions can cause these errors:

References to an invalid ASI space.
References to a legal ASI space, but with an invalid data size.
An invalid virtual address field within a valid ASI space.

The MFSR.CS bit is not asserted for the following conditions: Bus error on ASI 0x08, 0x09, 0x0a, 0x0b, 0x20-0x2f (the *standard* and *passthrough/bypass* ASIs), or errors on MMU probes. The contents of the Fault Address register is valid for control space access errors.

4.11.11.3 Error Mode and Internal Errors

If the processor enters error mode for any reason (such as an exception while PSR.ET is deasserted), the MFSR.EM bit will be set. This bit should be examined by the reset handler to distinguish software induced error conditions from hardware reset.

Viking will also enter error mode if a detected internal error occurs. An example of this is detection of multiple tag matches within the instruction and data caches. In these cases, the MFSR.FT field will be set to 6, indicating an internal error.

> Important
> When internal error occurs and watchdog reset is taken, only MFSR.EM and MFSR.FT are meaningful. The state of the other MFSR bits are not guaranteed.

4.11.11.3 MFSR timing and operation

The contents of the Fault Status register can be misleading if examined at an arbitrary time. For example, an instruction fetch which is not a "demand" fetch (not needed immediately for execution) may cause the Fault Status register to indicate a fault which is never signalled as an actual exception. The MFSR is guaranteed to be valid only after instruction, data, and store buffer (data_store_error) exceptions.

The Fault Status register is read-only and is cleared on a read. Writes are ignored. The standard virtual address is 0x00000300 using ASI 0x04. *Viking* provides a specific address (ASI=0x04, VA=0x00001300) to allow read/write access to the MFSR. For more on the standard features of MFSR, consult the SPARC Architecture Manual.

The MFSR.SB (Store Buffer) error bit is *sticky*. In other words, once it is set, it will not be overwritten by the occurence of any other exceptions. This is to ensure that store buffer error events can never be lost. The SB bit will be cleared on any read of the MFSR, and can also be cleared by writing explicitly to the read/write version of the MFSR.

> Important Note:
>
> Translation errors are considered to be high priority errors. The occurence of a translation error can not be overwritten by any other errors. Even if the exception associated with a translation error is not taken (due to other exceptions, prefetches, branches, etc.), the MFSR.FT bit will continue to indicate the occurence of a translation error.
>
> This implies that under certain circumstances, a trap may incorrectly stated to be a translation error, when in fact it may have been caused by other events. System software should be able to recover from this situation by using *probe* operations to test the validity of translations, and if correct retrying the instruction which reported the exception. If the true source of the exception continues to exist, it will be reported again.
>
> Translation errors may be overwritten by *other* translation errors, in which case the MFSR.OW (overwrite) bit will be set.

General rules for *overwriting* the MFSR (and MFAR) are presented in table 4-10. The MFSR.OW bit will always be set if an overwrite condition occurs.

Table 4-10 *MFSR Overwrite Operations*

Pending Error	New Error	OW Status	Action Signalled
Translation Error	Translation Error	Set	Translation Error
Translation Error	Data Access Exception	Unchanged	Data access Exception
Translation Error	Instruction Access Exception	Unchanged	Instruction Access Exception
Data Access Exception	Translation Error	Clear	Translation Error
Data Access Exception	Data Access Exception	Set	Data access Exception
Data Access Exception	Instruction Access Exception	Unchanged	Instruction access Exception
Instruction Access Exception	Translation Error	Clear	Translation Error
Instruction Access Exception	Data Access Exception	Clear	Data Access Exception
Instruction Access Exception	Instruction Access Exception	Set	Instruction Access Exception

In all cases where *simultaneous* errors occur, *Viking* will choose the highest priority error and update the status accordingly. The positional priority described above must also be taken into account. The priority order is listed in the following table, where priority 1 is the highest priority:

Table 4-11 *MFSR Error Priority*

Error	Priority
Translation Error	1
Data Access Exceptions	2
Instruction Access Exceptions	3

4.11.11.3 MFSR Register

Description

Rsvd	EM	CS	SB	P	UD	UC	TO	BE	L	AT	FT	FAV	OW
31 18	17	16	15	14	13	12	11	10	9 8	7 5	4 2	1	0

Rsvd Reserved. These bits are read-only zeroes.

EM Error Mode Reset Taken. This bit when asserted indicates that an Error Mode Reset has been taken.

CS Control Space Access Error. This bit is asserted on the following conditions: [1] invalid ASI space, [2] invalid ASI size, [3] invalid VA field in valid ASI space including external control space (0x02) ASIs, or [4] bus errors on ASI 0x02. This bit will not be asserted, however, for the following conditions: [1] bus error on ASI 0x08, 0x09, 0x0a, 0x0b, 0x20-

0x2f and [2] bus error on MMU Probe. Note that MFAR holds a valid address on control space access (CS) errors.

SB — Store Buffer Error. This bit when asserted indicates that a Store Buffer error (data_store_error) has occurred.

P — Parity Error. This bit when asserted indicates that a Parity error has occurred. Parity errors also set MFSR.UC bit.

UD — Undefined Error. This bit is set for external bus errors when the external system signals a retry operation and asserts data ready. UD/UC/TO/BE errors are mutually exclusive.

UC — Uncorrectable Error. This bit is set for external bus errors for which an uncorrectable error occurred. Parity errors and ECC errors are also reported as uncorrectable errors. UD/UC/TO/BE errors are mutually exclusive.

TO — Time-Out. When this bit is set, it indicates that a time-out error occurred for a external bus transaction. UD/UC/TO/BE errors are mutually exclusive.

BE — Bus Error. When this bit is set, it indicates that a bus error occurred on a faulting access. This includes invalid bus transactions and errors for which system registers need to be probed. UD/UC/TO/BE errors are mutually exclusive.

L — The Level field is set to the page table level of the entry which caused the fault. If an external bus error is encountered while fetching a page table entry (either a PTE or PTP) the Level field records the page table level for the entry. The field is defined as follows:

L	Level
0	Root pointer
1	Level 1 entry
2	Level 2 entry
3	Level 3 entry

AT — The Access Type field define the type of access which caused the fault. It is defined as follows:

AT	Access Type
0	Load from User Data Space
1	Load from Supervisor Data Space
2	Load/Execute from User Instruction Space
3	Load/Execute from Supervisor Instruction Space
4	Store to User Data Space
5	Store to Supervisor Data Space
6	Store to User Instruction Space
7	Store to Supervisor Instruction Space

FT The Fault Type field defines the type of the current fault. It is defined as follows:

FT	Fault Type
0	None
1	Invalid address error
2	Protection error
3	Privilege violation
4	Translation error
5	Access bus error
6	Internal error
7	Reserved

Invalid address errors, protection errors and privilege violations are a function of the Access Type and the ACC field of the corresponding PTE. The errors are set as follows:

Table 4-12 *Access Permission vs Access Type*

AT	FT Code								
	PTE[V]=0	PTE[V]=1, PTE[ACC]=							
		0	1	2	3	4	5	6	7
0	1	2	2	-	-	-	2	3	3
1	1	2	2	-	-	-	2	-	-
2	1	-	-	-	-	2	-	3	3
3	1	-	-	-	-	2	-	-	-
4	1	2	-	2	-	2	2	3	3
5	1	2	-	2	-	2	-	2	-

The invalid address error code is set when an invalid PTE or PTP is found while fetching an entry from the page table for a regular table-walk or a probe operation. A translation error code is set when an external bus error, reserved PTE or a level-3 PTP is found while fetching an entry from a page table for a regular table-walk or a probe operation. The L field records the page table level at which the error occurred for the above two error codes. The UD, TO, BE, and UC fields record the type of bus error, if any. The protection error code is set if an access is attempted that is inconsistent with the protection attributes of the corresponding page table entry. The privilege error code is set when a user program attempts to access a supervisor only page. A bus error code is set when an external bus error occurs during memory access. The internal error code is set when either cache detects an internal inconsistency, like multiple matches for a particular request. When this happens error mode is entered, requiring a system reset using watchdog reset. See section 4.11.11.3.5 — *Error Mode and Internal Errors*

FAV Fault Address Valid bit is asserted if the contents of the Fault Address Register are valid. The Fault Address Register is not valid for instruction access faults.

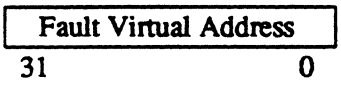
OW The Overwrite bit is asserted if the Fault Status Register has been written more than once by faults of the same class since the last time it was read. Programmer's Note: A trap handler should consult this OW bit to find out if the information in FSR reflects the current fault or not. OW asserted indicates that the FSR has been updated since the last fault that was/is currently being taken. Multiple faults can occur before the trap is taken by the processor and the fault status is read by the processor. The Fault Status register records the cause of the latest and sets the OW bit to indicate that a previous status has been lost.

4.11.11.4 Fault Address Register (MFAR)

The Fault Address register records the virtual address of the fault reported in the Fault Status register. This register is overwritten according to the policy defined for the MFSR. The MFAR is *read-only* according the the reference MMU specification. For diagnostic purposed, *Viking* has added additional read/write access to the MFAR using virtual address 0x00001400 in ASI 0x04. The *normal* read-only MFAR is at virtual address 0x00000400 in the same space.

> Important Note:
>
> *Viking* will *never* place instruction fault addresses in the FAR. The information is not needed, since it is saved as the faulting PC/NPC when the trap occurs.

This register must be accessed as a word. Other data references provoke a data_access_exception. The structure of the Fault Address register is:



Important Note:

A rare scenario can occur when *Viking* is operating in MBUS mode with the store buffer off (not normal operating conditions). If a memory reference takes place, that causes a copy-back, and the copy-back suffers a fault (memory fault), *Viking* responds by sending data_access_exception to the processor pipeline. When this occurs, the MFAR holds the virtual address that caused the copy-out. Note that this is not the address that caused the fault to occur. In this situation, *Viking* does not set MFAV because it is misleading.

If such an error occurs (MBUS, data_access_exception, FAV deasserted, store buffer disabled), system software can recover by manually forcing any modified data in the four possible cache lines (based on the virtual address) out to memory using transparent MMU references. Once this data has been flushed, the appropriate valid bits should be cleared, and the original operation may be retried.

This situation is not expected to occur during normal operation.

4.11.11.5 MMU Shadow FSR Register (MSFSR)

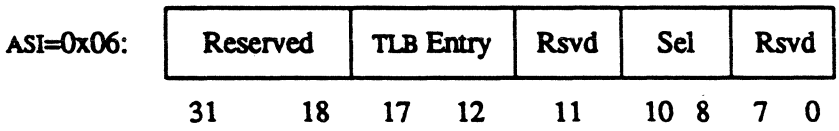
This is identical to the FSR but is used to record memory system errors while in Emulation mode. This is done to avoid destroying the regular MFSR because of emulation instructions. This register is not used for normal operation. Also see section 4.15).

4.11.12. MMU TLB (Page Descriptor Cache) direct access

The MMU entries are accessible directly with the ASI value 0x06. This direct access capability is provided for diagnostic purposes and also to lock TLB entries.

MMU entries are accessed as a 32-bit word. Other data sizes provoke a data access exception.

The Address format is:



The various bit fields have the following meanings:

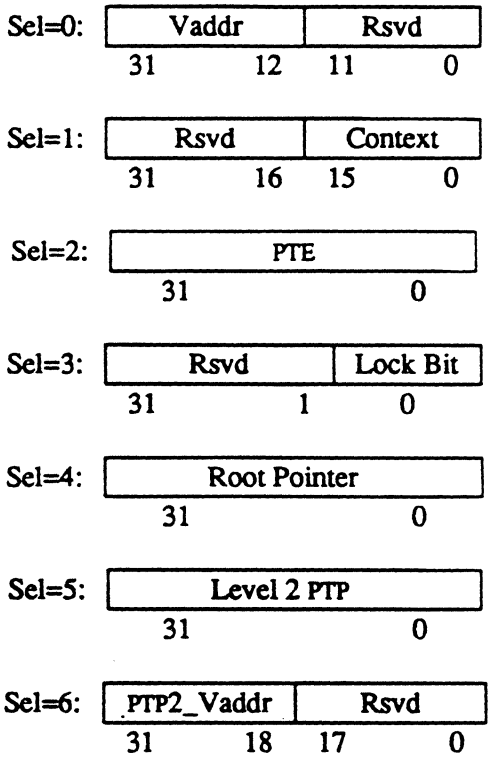
Rsvd: Reserved. All reserved bits are ignored (But should be zero).

Entry: Selects which of the 64 entries is referenced.

Sel: Select. Determines what part of the referenced entry is accessed. The Sel field allows selection of both TLB fields, as well as the values cached in the root pointer and PTP2 caches. Depending on the Sel encoding, various registers are selected.

Sel values 0 through 3 are used to access TLB entries, value 4 is used to access the level-0 pointer and values 5 and 6 are used to access the level-2 pointer entry.

For Sel values 4, 5 and 6 the TLB entry field is not used. Data will be accepted or returned, in the following formats:

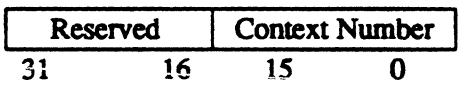


Vaddr: Virtual Page Number. When the entry maps a 4 K-byte page all bits are defined. When the entry maps a 256 K-byte segment only the 14 most significant bits [31:18] are significant. When an entry maps a 16 M-byte region only the 8 most significant bits [31:24] are significant. When an entry maps 4 G-byte region, the entire VAddr field is not significant.

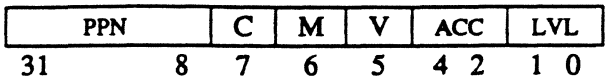
Rsvd: Reserved

PTP2_Vaddr:
Level 2 PTP virtual address.

The Context Number tag is accessed (Sel=1) with the following data format:



The cached PTE is accessed (Sel=2) with the following data format: (Note that this format is slightly different than the in-memory PTE format.)



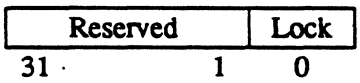
The meaning of the bit fields is identical to the definition of a PTE except for the V and LVL fields.

V: Valid. The Referenced bit location in the PTE is used to hold the Valid bit for the entry.

LVL: Level. When a PTE is cached the ET field is used to specify the mapping size. The encoding is the following:

LVL	Mapping Size
0	4 Gigabytes
1	16 Megabytes
2	256 Kilobytes
3	4 Kilobytes

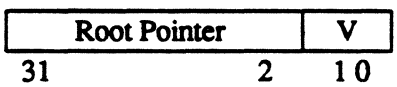
The Lock bit is accessed (Sel=3) with the following data format:



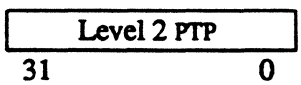
The Lock bit is set to 0 by the table walking hardware upon power-on reset. If set to one, the entry will not be displaced by the replacement algorithm. It is the responsibility of the software not to lock all entries. When all entries are locked, new PTEs can not be brought into the TLB and hence the translation can not occur.

> Important Note:
> If all TLB entries are locked, no replacement can take place. This will result in the processor entering an *infinite table walk* sequence. The processor will continue to read from the page tables forever and never return. No error will be reported, and the only way to exit is with reset.

The cached Root Pointer (Sel=4) with the following data format:

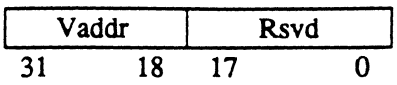


The cached level 2 PTP is accessed (Sel=5) with the following data format:



The virtual address of the level 2 PTP is accessed (Sel=6) with the following data

format:



Important Note:

It is possible to have multiple matches in the MMU if two or more entries mapping the same virtual space area are simultaneously present in the TLB. This cannot happen during the regular operating modes where entries are loaded by the table walking hardware. However, with the direct access capability it is possible to erroneously load distinct entries mapping the same portion of a virtual space. When a virtual address from this area is translated by the MMU, the result is undefined. This condition is not reported to the software. Though operation is undefined, the hardware is internally protected and will not be damaged should this condition occur.

This condition may also occur under non-diagnostic situations if MMU FLUSH transactions are not issued where required. As an example, if a normal level3 PTE is present in the TLB, the page table is modified to include a level2 or higher PTE mapping the same space, and a reference to a a different location within the level2 mapping. Under these conditions, the TLB will end up with two entries mapping the original page. A FLUSH transaction is required after changing the page table mapping, before any user instructions are generated. FLUSH operations are required by the reference MMU specification for these cases.

It is the software's responsibility to maintain the consistency between the MMU and the page tables when entries are explicitly modified.

4.12. Store Buffer

The *Viking* store buffer is a fully-associative cache of 8 double-word entries. This buffer functions to eliminate most of the performance penalties associated with write-through cache operation, and copy-back (cache flush) operations. This depth is sufficient to hold 16 SPARC registers, the number required to flush a register window.

4.12.1. General Operation

The store buffer is a *flushing* type buffer. If the (doubleword) address of a read transaction matches the (doubleword) address of any write transaction currently in the buffer, the read will wait until that write has completed before continuing. No data is returned from the store buffer to the instruction pipeline. This type of match will force the buffer to flush to memory as quickly as possible.

In addition, the buffer does not do any *byte collection*. It does, however, turn sequences of memory references within a cache block into *burst write* operations on the bus (CC mode only). Burst writes will continue for an arbitrary number of cycles, as long as they are within a cache line.

The store buffer components (tags, data, and control) are accessible via ASI spaces 0x30-0x32 for diagnostic purposes.

4.12.2. Operation in CC Mode

The buffer is primarily used in CC mode, where all store operations are completed immediately into the store buffer. In CC mode, a ST operation will never wait for completion, unless the buffer is full, disabled, or a TLB miss operation occurs (see section 4.12.4 for a more complete description). The state of the data cache (hit, miss, or disabled) does not affect store buffer operation in CC mode.

4.12.3. Operation in MBUS Mode

In MBUS mode only non-cacheable stores and copy back data goes into the store buffer. This is primarily due to the copy-back, write allocate caching policy used on the MBUS.

4.12.4. Non-buffered (Synchronous) Operations

In CC mode, there are several cases when stores cannot or should not be buffered. These cases include atomic operations, Store Alternates (STA), MMU R&M Updates, and operations when the store buffer is disabled. All of these actions block the execution pipeline until they have completed. Since external stores must be performed in order, each of these conditions forces all entries in the store buffer to be written out to memory before the synchronous operation begins.

When the buffer is full, new store operations will cause the pipeline to stall until a single entry in the buffer is available. The buffer is not forced to flush in this situation.

4.12.5. Store Buffer (Data Store) Exceptions

Data store exceptions take priority over all other exceptions, except reset. When an exception occurs on a buffered write, the resulting trap handler must be able to inspect and attempt to restart the write which failed. In order for a data store error to be reported, the PSR.ET bit must be asserted, and the MFSR.NF bit should be deasserted.

When an error occurs, the store buffer is automatically disabled. A Store Buffer copyout is *not* initiated: and no other writes will be issued from the Store Buffer. All buffer entries, including the faulting one, are retained in the buffer. All subsequent writes (nominally in the trap handler) are synchronous, bypassing the buffer. This will continue until the store buffer is re-enabled.

Once in the trap handler, the faulting write can be retried. This is accomplished loading the faulty address and data directly from the store buffer into registers (using the diagnostic ASI access to the buffer's address and data information directly). A store alternate through the MMU *pass through* ASIs can then be used to perform an untranslated store to this physical address. In this case, the MCNTL.AC (Alternate Cacheable) bit will be used to determine cacheability. The AC bit should be set to the same value as the C bit field of the store buffer tag register (since this is was the state of the C bit in the original transaction).

The data store error handler code should set the MCNTL.NF (no fault) bit before attempting to retry the operation. After the STA to retry the operation, the MFSR error bits should be checked explicitly to determine if the operation was successful or not. If the MCNTL.NF bit is not set, an error on the retry of these transactions can cause entry into error mode.

If this retry fails system software must decide how to continue recovery efforts. Generally, the error is always guaranteed to be from a store operation in the current context (since context changes always force a store buffer copy-out, pending stores from another context cannot be present.) Once the faulting process is identified, the process can be interrupted or killed, rather than stop the entire system.

> Important Note:
>
> In MBUS mode, a data store error resulting from a copy-back operation is not guaranteed to be from the current context. In this case, isolating faulty accesses to the process which caused them is more difficult. This situation may still be recoverable, but is very complex.

Note that standard ordering of memory transactions can be maintained even in the presence of store buffer errors and recovery operations. As long as the recovery routines retry store buffer operations in the order they were requested, no ordering is changed. Any memory operations within the trap handler can be considered to have executed "before" these buffered writes are performed.

When a store buffer exception (data_store_error) occurs, *Viking* retains information for all pending stores, including the store which encountered the exception, in the store buffer. These can be accessed in several asi control spaces: store buffer control (ASI 0x32), store buffer tag (ASI 0x30), store buffer data (ASI 0x31) and MMU.SB (ASI 0x04). These store buffer entries assist recovery from a store buffer exception.

The SBCNTL.Dptr is left intact after an exception to allow software recovery. In order to re-enable the store buffer after a data_store_error was taken, the SBCNTL.Dptr must be initialized to allow proper execution.

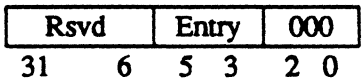
4.12.6. Store Buffer Disabled Operation- strong ordering

The Store Buffer is disabled when *Viking* is first powered up. As a result, all stores are synchronous, and block the execution pipeline until they complete. During normal operation, disabling the store buffer allows applications to be run with ***strong sequential ordering*** of memory references.

4.12.7. Store Buffer Tags

ASI=0x30 - Store Buffer Tags.

This ASI is used to perform reads and writes to the store buffer's physical tags, mainly for diagnostic purposes. The Address format is:



The bit fields are:

Rsvd: Reserved. These bits are ignored.

Entry: Entry Number. The eight entries of the store buffer are accessed using the entry field.

Accessing the store buffer tags with a quantity other than a double-word will result in a data_access_exception.

Tags are returned (or written to the appropriate entry).

The Tag format is:

Rsvd		SP	Burst	V	S	C	Size		Address	
63	43	42	41	40	39	38	37	36	35	0

Rsvd — These bits are read as zeros and ignored for writes.

SP — Store Barrier Pointer bit. This bit gets asserted by the STBAR instruction. See section 4.4.5 for more details.

Burst — Burst Mode access. This bit if set indicates that the next entry in the store buffer corresponds to the next consecutive address and can thus be issued in burst mode.

V — Valid bit. If set it indicates that the entry is valid.

S — Supervisor Bit. If set it indicates that the entry corresponds to a supervisor process.

C — Cacheable bit. If set it indicates that the entry is a cacheable access.

Size — Size of transaction according to the table below:

Size	Data Quantity
00	Byte
01	Half-Word
10	Word
11	Double-Word

Address — Address for the store buffer entry. This address must be correct depending on the size of the transaction. For instance a double-word store buffer entry must have the lower 3 bits of its address to be zero, if not an error is generated.

4.12.8. Store Buffer Data

ASI=0x31 - Store Buffer Data.

This ASI is used to perform reads and writes to 64-bit data stored in store buffer entries. Data entries are addressed in the same way as store buffer tags. The Address format is:

Reserved		Entry		000	
31	6	5	3	2	0

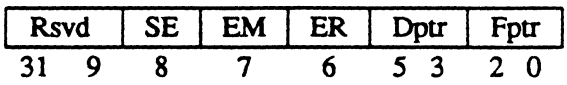
The store buffer data must be accessed as double-words. Other data quantities provoke a data_access_exception.

4.12.9. Store Buffer Control

ASI=0x32 - Store Buffer Control.

The store buffer control register containing the fill and drain pointers is accessible with the ASI value 0x32. This access is a single word access. All other size accesses will generate a data_access_exception. The address field is not used for this access.

The number of pending stores is determined by subtracting the Drain pointer from the Fill Pointer (modulo 8). When the two pointers are equal and there are valid entries, the store buffer is full and cannot accept any more entries. The buffer is empty when the two pointers are equal but there are no valid entries. The Data format is:



Rsvd These bits are read as zeros and ignored for writes.

SE: Store buffer enabled. This bit is Read-only, and indicates whether the store buffer is enabled (1) or disabled (0). This is a shadow copy of the MCNTL.SB bit, and is provided for convenience.

EM: Store buffer empty. This bit is read-only, is set at reset, and indicates whether the store buffer is empty (1) or non-empty (0).

ER: Store buffer error pending. This bit is Read-only, and indicates whether an untaken store buffer error is pending. This bit is set when a store buffer exception occurs while traps are disabled (PSR.ET=0). This bit is cleared by taking the store buffer exception trap, which occurs automatically when traps are re-enabled. The bit is also cleared on Reset.

Dptr. Drain Pointer. This value indicates the first entry of the store buffer which would perform a bus transaction. See section 4.12.5 on cases when this bit must be cleared after a data_store_error.

Fptr. Fill Pointer. This value indicates the first entry of the store buffer where a new store request can be written. If it is equal to the drain pointer then the store buffer is full.

4.13. Traps

This section gives an overall view of the types of traps which may occur during normal or exceptional operation of *Viking*. In particular, this section describes the set of standard SPARC traps, and the exact definition of *Viking*'s implementation of them. In many cases, the trap is implemented exactly to the definition in the SPARC Architecture Manual. Standard architectural operations, like decrementing CWP, and copying the PSR.S bit to PSR.PS will not be discussed in detail.

Some general properties are presented first, followed by the details of various traps. Note that *Viking* is fully conformant to the SPARC architecture, and implements *exact* traps. In some cases (floating point and store buffer), *Viking* uses the *deferred* trap model of the SPARC architecture.

All exception requests propagate through the instruction pipeline and are resolved only in the last pipe stage (WB). *Viking* can executes more than one instruction in a given cycle, and each of those instructions can suffer one or more exceptions. The following rule resolves exception priorities among faulting instructions in the same group:

> Important Note:
> A *low priority exception* at an *earlier instruction* in a given instruction group will have a *higher priority* than a *high priority exception* at a *later instruction* in the same group.

Note that *interrupts* are effectively positioned at the *last* valid instruction in a particular instruction group. In addition, a *valid instruction* must be present in order for the interrupt to be registered. If the execution pipeline is not progressing (waiting on cache misses, for example) an interrupt will not be taken. Due to this, maximum interrupt latency for *Viking* is highly system dependent.

4.13.1. Exceptions and Program Counters

Three program counters are involved when an exception occurs.

XPC, the exception program counter
XNPC, the exception next program counter
XHPC, the beginning of the exception handler code

Viking recovers the XPC and XNPC values from one of the many stored program counters, depending on where the exception occurs. The values are then stored in registers *l1* and *l2* in the new register window upon entry into the trap handler. The new program counter (exception handler target) is computed according to the standard SPARC model, shown below:

Table 4-13 *Exception Handler PC formation*

Exception Handler PC:

TBR[31:12]	Trap Type[7:0]	0000
31 12	11 4	3 0

4.13.2. Error Mode

Error Mode can be entered when any exception occurs while the ET bit is cleared. In general, these are considered fatal errors, though system software may be able to recover in certain cases.

Error mode generates a watchdog reset trap, which acts like any other trap, with a few differences. For most error mode traps, the TT (trap type) field is not set. This is to retain the *prior* value of TT, to help recovery. There is one case where this rule is *not* applied: If the error mode trap occurred during the execution of an RETT instruction, the TT field *will be set* based on the exact cause of the exception.

In all cases, the PSR.PS bit will *not* be affected by a watchdog reset.

In all cases, the MCNTL.BT (Boot Mode) bit *will be set* by a watchdog reset.

4.13.3. Fault Status Updates

There are many details associated with updates to the fault status register. These are discussed in detail in section 4.11.11.3.6 — *MFSR timing and operation*

4.13.4. Trap Details

The following lists *Viking* Trap Table, which for the most part looks similar to the SPARC Manual Trap Table (refer to the SPARC Architecture Manual) but provides a formal definition of which traps *Viking* does or does not take, how *Viking* handles its traps, and any reasoning behind those operations.

Table 4-14 *Table of Traps supported by Viking*

Exception or Interrupt Request	Priority	Trap Type
reset	1	0x00
data_store_error	2	0x2b
instruction_access_exception	5	0x01
privileged_instruction	6	0x03
illegal_instruction	7	0x02
fp_disabled	8	0x04
cp_disabled	8	0x24
window_overflow	9	0x05
window_underflow	9	0x06
mem_address_not_aligned	10	0x07
fp_exception	11	0x08
data_access_exception	13	0x09
tag_overflow	14	0x0a
division_by_zero	15	0x2a
trap_instruction	16	0x80-0xff
interrupt_level_15	17	0x1f
interrupt_level_14	18	0x1e
interrupt_level_13	19	0x1d
interrupt_level_12	20	0x1c
interrupt_level_11	21	0x1b
interrupt_level_10	22	0x1a
interrupt_level_9	23	0x19
interrupt_level_8	24	0x18
interrupt_level_7	25	0x17
interrupt_level_6	26	0x16
interrupt_level_5	27	0x15
interrupt_level_4	28	0x14
interrupt_level_3	29	0x13
interrupt_level_2	30	0x12
interrupt_level_1	31	0x11

Note: The second column describes the exception priority in the event of more than one exception is detected at a given instruction. The third column is the value that TBR.TT will be set to and also serves as an offset to a quad-word block fetched to get at the exception handler when the exception event occurs (see table 4-13).

4.13.4.1 Reset Trap

Trap type (TT not set), priority 1.

Viking recognizes two types of reset: *Viking* Reset and Watchdog Reset. For a detailed description of reset operation, see section 4.3 — *Reset Operation.*

4.13.4.2 Data Store Error Trap

Trap type 0x2b, priority 2.

Data_store_error is caused when a store buffer copyout encounters an external bus error (e.g. parity or ECC problem). For a detailed description see section 4.12.5 — *Store Buffer (Data Store) Exceptions*.

4.13.4.3 Instruction Access Exception Trap

Trap type 0x01, priority 5.

Instruction_access_exception may be caused by many events. The *normal* source of this error is MMU protection errors for pages which have been swapped out of virtual memory space. In addition, bus errors and code breakpoints can cause this exception. See the MMU for a detailed description of error reporting for these traps. Note that the fault address register (FAR) is never updated for these errors.

4.13.4.4 Privileged Instruction Trap

Trap type 0x03, priority 6.

Privileged_instruction trap is caused when PSR.S is cleared and a privileged instruction is issued. There are many classes of privileged instructions, see the SPARC Architecture Manual for a complete list.

There are many details related to generating traps for the RETT instruction. *Viking* implements the rules described in the SPARC Architecture Manual.

4.13.4.5 Illegal Instruction Trap

Trap type 0x02, priority 7.

Illegal_instruction (sometimes also called unimplemented_instruction) trap is caused when an instruction with an unassigned opcode or illegal opcode is executed. For example, *Viking* issues an illegal instruction trap when a RETT instruction is executed with the PSR.ET bit set, and in several data dependent cases of the WRPSR instruction (see section 4.4.3 — *Write PSR (WRPSR)* Integer divide instruction also generate data dependent illegal instruction traps (see section 4.4.2 — *Integer Divide (IDIV)*).

4.13.4.6 Floating Point Disabled Trap

Trap type 0x04, priority 8.

Fp_disabled trap is caused when PSR.EF is cleared and any floating point instruction is issued. This trap is never issued for integer multiply or integer divide.

4.13.4.7 Coprocessor Disabled Trap

Trap type 0x24, priority 8.

Viking does not provide a coprocessor interface. PSR.EC (enable coprocessor) bit is zero, and non-writable. Any attempt to set the PSR.EC bit will generate an illegal_instruction trap. Any attempt to execute a coprocessor instruction will generate a cp_disabled trap. *Viking* never generates a cp_exception trap.

4.13.4.8 Window Overflow Trap

Trap type 0x05, priority 9.

Window_overflow trap is caused by a SAVE instruction being executed when the next available register window (CWP-1) is marked invalid in the WIM register. Note that this trap is generated *only* in response to a SAVE instruction, no other operations, including other traps, can cause this exception.

4.13.4.9 Window Underflow Trap

Trap type 0x06, priority 9.

Window_underflow trap is caused by either a RESTORE or RETT instruction which would cause the *incremented* value of CWP to point to an invalid register window. In the case of RETT, this trap will immediately lead to an error mode condition, and watchdog reset.

4.13.4.10 Memory Address Not Aligned Trap

Trap type 0x07, priority 10.

Mem_address_not_aligned trap is caused by a load, store or control transfer operation which violate the correct SPARC alignment. The alignment of an instruction is a *word*. Halfword references require the low bit to be zero. Word references require the low 2 bits to be zero. Double word references require the low 3 bits to be zero.

4.13.4.11 Floating Point Exception Trap

Trap type 0x08, priority 11.

Fp_exception traps are caused by certain floating point arithmetic conditions being detected. These exceptions are *deferred* traps, and will only be reported upon execution of another floating point operation (or floating point event) at some later time. The time taken to cause the exception is variable, depending on the pipeline state, numeric conditions, and the exact sequence of instructions. Improper floating point error recovery can also cause these exceptions, particularly sequence errors.

The exact instruction which caused the exception can be determined by reading the state of the floating point queue, described in section 4.6.3 — *Floating Point Queue*

4.13.4.12 Data Access Exception Trap

Trap type 0x09, priority 13.

Data_access_exception traps are generally caused by MMU protection errors for load and store operations. They may also be caused by bus errors, and data breakpoints. See the MMU section for a detailed description of error reporting for these exceptions.

4.13.4.13 Tagged Operation Overflow Trap

Trap type 0x0a, priority 14.

tag_overflow traps are caused by execution of *tagged add or subtract and trap on overflow* (TADDCCTV and TSUBCCTV) instructions. The trap will be signalled when the least significant 2 bits of either source operand is *non-zero*, or when the operation produces a result which causes the overflow flag to be set.

4.13.4.14 Integer Divide by Zero Trap

Trap type 0x2a, priority 15.

Division_by_zero trap is caused when an integer divide instruction is issued with the value of the second operand (denominator) being zero.

Note also that an illegal instruction trap will be generated by *Viking* if the numerator of a divide operation has significant digits beyond 52 bits. If both divide by zero, and the illegal instruction conditions exist, the illegal instruction trap will be signalled.

4.13.4.15 Trap Instructions (TICC)

Trap type 0x80-0xff (instruction dependent), priority 16.

Trap_instruction traps are software initiated traps, caused by the execution of TICC instructions. *Viking* implements these instructions according to the SPARC Architecture Manual.

4.13.4.16 Interrupts

Trap type 0x11-0x1f (IRL pin dependent), priorities 17-31.

Interrupt_level_n traps are by generation of hardware interrupt requests. The normal source of these requests is from external pin assertions on the IRL[3-0] pins. In addition to these external requests, *Viking* can generate *internal* interrupt requests from several sources. Generation and definition of these interrupt requests is system dependent.

Interrupt requests are compared to the current *processor interrupt level* (PSR.PIL). Requests at levels *higher* than the current PIL, or equal to 15 (*non-maskable*) will be taken, assuming there are no higher priority exceptions pending. The PSR.ET (enable traps) bit must be set for *any* interrupt to be accepted (including non-maskable interrupts).

The external interrupt requests are sampled in successive cycles to debounce transient requests. The request must be asserted for a minimum of three cycles before it is presented to the pipeline. A valid instruction must exist at that stage of the pipeline before the interrupt will be taken. Interrupt requests are *level sensitive* at the pins of *Viking*. System hardware can control these pins in a implementation dependent manner to implement different system interrupt architectures.

Once an interrupt is presented to the pipeline, it will cause the store buffer to be flushed, will update register state according to the SPARC Architecture Manual, and will initiate a control transfer to the beginning of the trap handler. The duration of this activity is processor state, and is system dependent. Maximum interrupt latency is determined by internal and external factors, such as the maximum length of certain floating point operations, system memory latency times, and the maximum store buffer depth.

Internal sources for interrupt requests are from breakpoints. The breakpoint logic can be programmed to cause interrupts at a software controlled level upon certain events. These may be code, data, or counter generated breakpoints. Several registers, including MDIAG.BKC (breakpoint control), ACTION (action on breakpoint event), and the counter registers determine when these interrupts are generated. The level generated for these interrupts is defined in the ACTION.BCIPL

(breakpoint and counter interrupt level).

4.13.4.17 Unsupported Trap Types

Viking does not implement the following optional SPARC trap types:

Table 4-15 *Unimplemented Trap Types*

Instruction_access_error
R_register_access_error
Cp_exception
Data_access_error
Unimplemented_FLUSH
Unimplemented_MUL
Unimplemented_DIV
Implementation-Dependent-Exception

4.14. Software Debugging Facilities

Viking provides debugging capabilities to facilitate debugging software and hardware prototypes. Four types of breakpoint mechanisms are provided:

Code address
Data address
Instruction count underflow
Cycle count underflow

These mechanisms can be selectively enabled to generate precise exceptions (data_access_exception or instruction_access_exception). They may be programmed to generate a selectable interrupt. In addition, they may be set to activate an external pin to easily trigger external analysis equipment. They are also a fundamental part of *Viking*'s *emulation* features, described in chapter 6 — *Remote Emulation Support*

In addition to address breakpoint facilities, programmable timers are provided for debug, code profiling and performance analysis. They can provide a high precision, low overhead timer for small application benchmarking. Their major application is to assist development or diagnostic teams in early hardware and software debug.

4.14.1. Priorities of Debug Interrupt and Exception

All address breakpoint interrupts and counter underflow interrupts use a common user programmable interrupt priority level (ACTION.BCIPL). Status bits are provided to differentiate between these interrupt sources. The following describes the priority order that these interrupts and exceptions adhere to.

If multiple breakpoint fault events (code, data) are signalled simultaneously, both *code* and *data* breakpoint status registers will set their *fault status bits*.

If multiple breakpoint interrupt events are signalled simultaneously, each activity will set its *interrupt status bits*.

Exception sources from multiple instructions are prioritized based on instruction order. An exception reported to an instruction will have higher priority than a simultaneous exception to the next instruction.

Asynchronous data_access_exceptions are honored before instruction_access_exceptions.

Instruction_access_exceptions are honored before synchronous data_access_ exceptions, and all data_access_exceptions are honored before interrupts.

4.14.2. Address Breakpoints - Code (Instruction) or Data

A single code (instruction) or data breakpoint register is available. This breakpoint can match on either 32-bit virtual or 36-bit physical addresses for code or data. When a breakpoint is set on an instruction, the instruction will not be executed. This holds true for fault, interrupt, and emulation.

> Important Note:
> A maximum of one *code space breakpoint* or one *data space breakpoint* can be active at a given time, not both simultaneously.

Each bit in the bitwise address comparison can separately be masked off to force equality on that bitwise comparison. The address equality bitmasks can be used to find references within a particular segment, page, cache line or word independent of the size of the access. Data space breakpoints can be qualified with access type (reads-only, writes-only, read-or-write). Atomic references (e.g. SWAP/LDSTUB) are considered both read and writes.

The action upon event control register (ACTION) specifies whether the address breakpoint should generate an exception, interrupt, or activate the external strobe (ESB) pin. The action on event register is defined in 4.14.4 — *Access to Debug Features*

4.14.3. Counter Breakpoints - Code (Instruction) or Cycle

> Important Note:
> A maximum of one *instruction count breakpoint* and one *cycle count breakpoint* can be active at a given time.

A single 32-bit wide control register programs the two 16-bit counters for instruction and cycle counts at the same time. It is not possible to modify one counter without the other.

The 16-bit cycle counter will count up to about 1.3 milliseconds at 50MHZ. Longer duration counters must be simulated in software by accumulating underflows of the 16-bit counter into a larger counter in memory.

The instruction counter can count either faster or slower than the cycle counter, depending on execution characteristics of the processor. It could theoretically count three times faster, if *Viking* was continuously executing three instruction groups. In general, it will count slightly faster than the cycle counter.

The combination of these two counter interrupts can be used to calculate the dynamic performance (in *million instructions per cycle*, or MIPS) of the executing program.

Cycle counter underflow events are always reported to the last valid instruction in the current instruction group. Instruction counter events occur as interrupts or emulation requests to the instruction after the instruction which causes the counter event. Instruction counter expiration events will be reported *after* the first instruction which causes the underflow.

Once set, the counter expiration event is persistent until served. The instruction which caused the counter event will ultimately be restarted.

When a cycle counter expires, action for that event may be deferred until valid instructions are available, and the pipeline is able to progress. If the action is deferred, the request (interrupt or emulation) will persist until the pipeline is able to continue. Once the initial event is signalled, the cycle counter continues counting down through the most positive number. In this way the total number of elapsed cycles from a given point may be calculated.

The instruction counter decrements the existing instruction count by the number of instructions which complete execution in a given cycle. The number can vary anywhere from 0 to 3 instructions per cycle. i The cycle counter always decrements by 1.

The following table describes how to set up breakpoints using the proper control registers, and where to find information that the actual action(s) has/have been triggered after the breakpoint.

Table 4-16 *Breakpoints - Control and Status*

		Transaction	*Breakpoint Response*	CONTROL: BKC CSPACE	PAMD	CBFEN	CBKEN	DBFEN	DBREN	DBWEN	ACTION MIX	BCIPL	STEN_CBK	STEN_ZIC	STEN_DBK	STEN_ZCC	IEN_CBK	IEN_ZIC	IEN_DBK	IEN_ZCC	CTRC ICNTEN	CCNTEN	STATUS: CTRS ZICIS	ZCCIS	BKS CBKIS	CBKFS	DBKIS	DBKFS	MSTAT CBKM	DBKM	ZICM	ZCCM
Address Breakpoints	*Data*	RD	data_acc_exc	0	x	0	0	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	u	u	1	u	u	u	u
		WR	data_acc_exc	0	x	0	0	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	u	u	1	u	u	u	u
		RD	interrupt	0	x	0	0	0	1	0	x	S	x	x	x	x	x	x	1	x	x	x	u	u	u	u	1	u	u	u	u	u
		WR	interrupt	0	x	0	0	0	0	1	x	S	x	x	x	x	x	x	1	x	x	x	u	u	u	u	1	u	u	u	u	u
		RD	emulation req	0	x	0	0	0	1	0	x	x	x	x	x	x	x	x	0	x	x	x	u	u	u	u	u	u	u	1	u	u
		WR	emulation req	0	x	0	0	0	0	1	x	x	x	x	x	x	x	x	0	x	x	x	u	u	u	u	u	u	u	1	u	u
		R/W	ESTROBE	0	x	0	0	x	1	1	x	x	x	x	1	x	x	x	x	x	x	x	u	u	u	u	E	E	u	E	u	u
	Code		instr_acc_exc	1	x	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	u	u	u	1	u	u	u	u	u	u
			interrupt	1	x	0	1	x	x	x	x	S	x	x	x	x	1	x	x	x	x	x	u	u	1	u	u	u	u	u	u	u
			emulation req	1	x	0	1	x	x	x	x	x	x	x	x	x	0	x	x	x	x	x	u	u	u	u	u	u	1	u	u	u
			ESTROBE	1	x	x	1	x	x	x	x	x	1	x	x	x	x	x	x	x	x	x	x	u	u	E	E	u	u	u	E	u	u
Counter Breakpoints	*Cycle*		interrupt	x	x	x	x	x	x	x	x	S	x	x	x	x	x	x	x	1	x	1	u	1	u	u	u	u	u	u	u	u
			emulation req	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	x	1	u	u	u	u	u	u	u	u	u	1
			ESTROBE	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	1	u	E	u	u	u	u	u	u	u	E
	Code		interrupt	x	x	x	x	x	x	x	x	S	x	x	x	x	x	1	x	x	1	x	1	u	u	u	u	u	u	u	u	u
			emulation req	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	x	x	1	x	u	u	u	u	u	u	u	u	1	u
			ESTROBE	x	x	x	x	x	x	x	x	x	x	1	x	x	x	x	x	x	1	x	E	u	u	u	u	u	u	u	E	u

S = Set BCIPL to select IRL; x = don't care; E = Event dependent; u = unchanged

In table 4-16 above, the notations S, x, and u are self-explanatory. The notation E (which is given to an affected status bit indicates that the bit is *Event dependent*, which means that the bit status after a breakpoint depends if a particular event has occurred. For example, the BKS.DBKIS bit in row *data address breakpoint* ESB, which is designated the notation E, will be set if ACTION.IEN_DBK and ACTION.BCIPL were set when the breakpoint occurred. (The latter 2 bits are designated don't cares).

For all the emulation request cases that are initiated by breakpoints, the MCMD.INITM bit *must be set.*

When setting a breakpoint, the programmer must allow for the latency until the breakpoint can be guaranteed active. To achieve this, the last STA 0x38 instruction that sets the breakpoint registers must be followed by an IFLUSH or a BA instruction.

Also, after the breakpoint has been taken, the programmer must explicitly disable (by disabling the appropriate control bits) that particular breakpoint, so that when normal execution is resumed, the same breakpoint will not be taken again.

4.14.4. *Access* to Debug Features

The MDIAG alternate address space contains code and data space breakpoint register facilities (CBK/DBK). The EDIAG (Emulation Diagnostics) address spaces contains the ACTION , instruction and cycle counter facilities (CTR). ASI addressing for emulation registers has been distributed into multiple ASI spaces to simplify the emulation interface. These registers are all shared with emulation resources.

> Important Note:
> In systems using scan-based emulation, there may be contention for use of the breakpoint and counter facilities between the remote emulator and the kernel. There is no contention avoidance mechanism provided to lockout debug software from disturbing the emulator context.

Care must be taken when both emulation and software debug features are being used at the same time.

4.14.4.1 MMU Breakpoint control registers

ASI=0x38 - MMU Diagnostics and Breakpoints.

ASI	Function	Access	Access
0x38	MMU Breakpoint Diagnostics	LD/ST	double

There are 4 memory mapped, *Viking*-specific, MMU breakpoint diagnostic registers (see table 4-17). These registers are double-word access only, any other size will cause a data_access_exception. All breakpoint enable and status bits are cleared at reset. All values and masks are unchanged through reset.

A single *address* breakpoint is controlled by these 4 registers. The address breakpoint may be set in code space or data space- but not both simultaneously.
An *address breakpoint* event can generate:

An instruction or data access breakpoint *exception*, or
An instruction or data address breakpoint *interrupt*, or
An instruction or data address breakpoint *emulation request*, or
Enable the ESB pin, or
none of the above

The response selected is determined by MDIAG_BKC ASI register, the ACTION register, and JTAG MCMD scan register. This JTAG MCMD scan register is initialized on powerup to inhibit IU initiated emulation requests. A remote emulation service processor can use *Viking*'s JTAG interface to:

Force *Viking* into emulation mode,
Setup an instruction or data address breakpoints,
Setup the breakpoint event to generate emulation mode requests,
Enable the pipeline to honor processor emulation entry requests,
Exit emulation mode

Instruction and data address breakpoint shares the same breakpoint register set. The address map for these MMU diagnostic registers is:

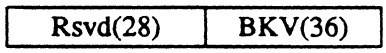
Table 4-17 *MMU diagnostic (breakpoint) registers*

Addr<9:8>	Reg Name	Description
0	MDIAG_BKV	Breakpoint Value (Addr)
1	MDIAG_BKM	Breakpoint Mask
2	MDIAG_BKC	Breakpoint Control
3	MDIAG_BKS	Breakpoint Status

These registers are defined as follows:

4.14.4.1 Breakpoint Value Reg

MDIAG_BKV:



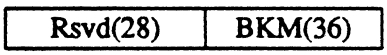
This register defines the code (or data) breakpoint address value.

Rsvd(28)
Reserved bits, read-only, should be zeroes.

BKV(36) Contains the 36 bit value with which either physical or virtual address of the code (or data) being accessed will be compared (as determined by MDIAG_BKC.CSPACE). When a match occurs, an event is generated depending on the state of the MDIAG_BKC and ACTION ASI registers. These actions are also affected by the JTAG MCMD.INITM bits (not programmer visible).

4.14.4.1 Breakpoint Mask Reg

MDIAG_BKM:



This register defines a mask-off value. Useful for address matching across a range.

Rsvd(28)
Reserved bits, read-only, should be zeroes.

BKM(36) BKM defines a per-bit comparison mask for BKV. For any bit which is *set* in BKM, the equivalent bit in BKV is *ignored* in the address comparison. This can be used to match on ranges of addresses.

4.14.4.1 Breakpoint Control Register

This register controls whether the breakpoint is to be set for code or data space address, whether to compare a physical or virtual address, and it also controls the different types of breakpoint enable signals. All bits are cleared on both hardware and watchdog reset.

MDIAG_BKC:

Rsvd(57)	CSPACE	PAMD	CBFEN
63 7	6	5	4

CBKEN	DBFEN	DBREN	DBWEN
3	2	1	0

CSPACE If CSPACE=1, the address in BKV is compared to *code* space address. DBREN, DBWEN and DBFEN are ignored. If CSPACE=0, the address in BKV is compared to *data* space address. CBKEN and CBFEN are ignored.

PAMD If PAMD=1, *physical* address is compared (bits 0-35 of BKV). If PAMD=0, *virtual* address is compared (bits 0-31 of BKV, ignore bits 32-35).

CBFEN Enables code breakpoint fault generation. When this bit is set, a code breakpoint match will cause an instruction_access_exception. When this bit is cleared, an interrupt as defined in ACTION will be reported.

CBKEN Enables code breakpoints. If disabled, no code breakpoint can occur.

DBFEN Enables data breakpoint fault generation. When this bit is set, a data breakpoint match will cause a data_access_exception. When this bit is cleared, (and either DBREN or DBWEN is set), an interrupt as defined in ACTION register will be reported.

DBREN Enables data breakpoints for *read* transactions (LD). This bit must be set, along with CSPACE being *cleared* for data read breakpoints to occur.

DBWEN Enables data breakpoints for *write* transactions (ST).

4.14.4.1 Breakpoint Status Reg

BKS:

Rsvd(60)	CBKIS	CBKFS	DBKIS	DBKFS
63 4	3	2	1	0

This register reports on the status of either code or data breakpoints.

CBKIS Indicates that an interrupt was generated as a result of a code breakpoint.

CBKFS Indicates that a code access exception was generated as a result of a code breakpoint.

DBKIS Indicates that an interrupt was generated as a result of a data breakpoint.

DBKFS Indicates that a data_access_exception was generated as a result of a data breakpoint.

Any type of reset sequence or any load ASI from this register will clear all the bits in this BKS register.

4.14.4.2 Counter Breakpoint Value (CTRV)

ASI=0x49 - Counter Breakpoint Values.

CTRV:

ICNT	CCNT
31 16	15 0

CTRV is a register that holds the counter breakpoint values, and is composed of two 16-bit fields: ICNT and CCNT.

ICNT This field specifies the number of *instructions* left before the next counter event. (See Note below).

CCNT This field specifies the number of *cycles* left before the next counter event. (See Note below).

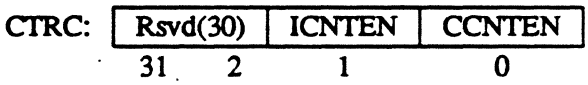
Neither counter is decremented when *Viking* is in emulation mode. ICNT and CCNT are read and written as a pair, and are unchanged at reset.

Important Note:

The total number of *instructions* remaining to execute before the next counter event is given by the value held in CTRV.ICNT (readable by LDA 0x49) *plus* the number of instructions in the group prior to the group containing the LDA 0x49 instruction. If the group prior to the group containing the LDA 0x49 instruction has zero instructions, then the value held by CTRV.ICNT reflects the *exact* number of instructions remaining to execute before the next counter event. An example is to use *jmpl* and *ba* branching to the LDA 0x49, which will force a bubble (hence zero instruction group) between the *ba* and the LDA. Another example might be using a sequential instruction before the LDA 0x49 or other methods that guarantee a group with a known number of instructions, allowing the programmer to get an exact instruction count until the counter event. Similarly, the number of *cycles* remaining before the next counter event is always *one* more than the value held in CTRV.CCNT (readable by LDA 0x49).

4.14.4.3 Counter Breakpoint Control (CTRC)

ASI=0x4a - Counter Breakpoint Controls.

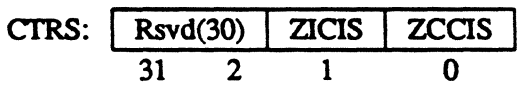


ICNTEN Enables the *instruction* counter. This bit is cleared on either a power-on reset or watchdog reset.

CCNTEN Enable the *cycle* counter. This bit is cleared on either a power-on reset or watchdog reset.

4.14.4.4 Counter Breakpoint Status (CTRS)

ASI=0x4b - Counter Breakpoint Status.



ZICIS Records the status of whether an *instruction counter interrupt* was generated. This bit is not set when an *emulation request* is induced by this event, and is cleared on either a power-on reset or watchdog reset.

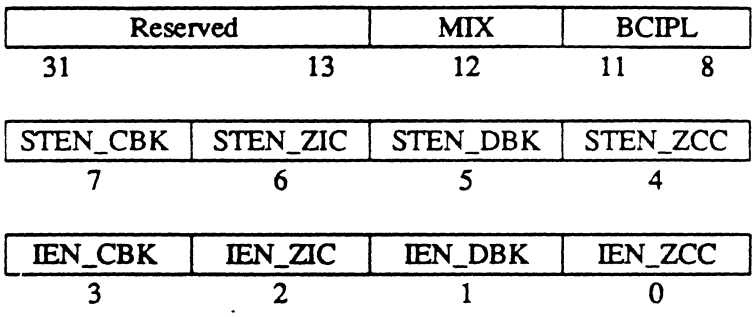
ZCCIS Records the status of whether a *cycle counter interrupt* was generated. This bit is not set when an *emulation request* is induced by this event, and is cleared on either a power-on reset or watchdog reset.

Viking can read and write the ASI memory mapped emulation counter status register. It will implicitly be written (set) by action register enabled counter

events.

4.14.4.5 Breakpoint ACTION Register

ASI=0x4c - ACTION (on event) Register.



MIX Enables multiple instruction per cycle execution mode. When cleared, *Viking* will issue no more than one instruction every cycle. This bit is cleared at reset.

BCIPL Determines the IRL to be used for all breakpoint or counter interrupts. When generated, these interrupts behave just as external interrupts- they must meet SPARC criteria for PIL and IRL to be seen in the execution stream. This field is cleared at reset.

STEN_CBK
Enables the ESB pin action on a code address breakpoint.

STEN_ZIC
Enables the ESB pin action on a zero instruction count expiration.

STEN_DBK
Enables the ESB pin action on a data address breakpoint.

STEN_ZCC
Enables the ESB pin action on a zero cycle count expiration. These bits are cleared at reset.

IEN_CBK Generates an interrupt in response to code breakpoint.

IEN_ZIC Generates an interrupt in response to zero instruction count.

IEN_DBK Generates an interrupt in response to data breakpoint.

IEN_ZCC Generates an interrupt in response to zero cycle count event.

If the selected event is to generate an emulation request, the associated IEN bit must be cleared. When an interrupt on these events is desired, the associated IEN bit must be set. These bits are all cleared at reset.

4.14.5. External Monitors

Viking is a highly integrated processor. It is possible for *Viking* to execute code continuously from its internal cache with no external indications *forever*. This can make system debug very difficult.

Several features are provided to reduce these problems. Scan based emulation provides access to the internal state of the machine. Additional pins have been defined to provide cycle by cycle observation of key internal states (the PIPE[0-9] pins). These pins are defined in detail in the System Interface chapter, and are mentioned here completeness.

These pins provide information on activity within a clock cycle, including: The number of instructions which complete execution. When branches (including a *taken* indication), and memory operations occur. When floating point instructions are issued. When the pipeline is being held by either floating point operations or memory operations. Interrupts and exceptions are also indicated.

In addition, the breakpoint registers may be programmed to activate the ESB pin when a breakpoint is detected. This allows an external device to be triggered (oscilloscope, logic analyzer, etc.).

If no interrupt or exception actions are enable for the breakpoint requests, these breakpoints will not alter the flow of execution in any way. They simply provide some clues as to the internal state of the machine.

4.14.6. PIPE[9:0] definition

Ten signals are provided to monitor the internal state of the processor. The definition of these signals is provided below:

Table 4-18 *PIPE[9:0] definitions*

Signal	Definition
PIPE[9]	Active when any valid memory reference occurred in the E0 stage of the previous clock cycle.
PIPE[8]	Active when any valid floating point operation occurred in the E0 stage of the previous clock cycle.
PIPE[7]	Active when any valid control transfer instruction was executed in the E0 stage of the previous clock cycle.
PIPE[6]	Indicates that no instructions were available when the group currently at the WB stage was decode (D0).
PIPE[5]	Active when the pipeline is being held by the Data Cache. (Generally processing a cache miss)
PIPE[4]	Active when the pipeline is being held by the Floating Point Unit. (Either queue is full, or dependencies)
PIPE[3]	Indicates that the branch in E0 stage of the previous cycle was taken (1) or not (0).
PIPE[2:1]	Indicates the number of instructions in the E0 stage of the current cycle: 00=None, 01=1, 10=2, 11=3.
PIPE[0]	Indicates that there is an exception or interrupt being signalled in the current cycle

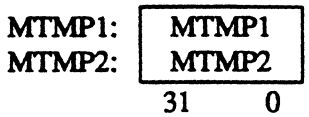
4.15. JTAG and Emulation

A brief description on these features is presented in this section for completeness. See chapter 5 (JTAG Serial Scan Interface) and chapter 6 (Remote Emulation) for more details.

4.15.1. Emulation Temporary Registers (MTMP[1-2])

ASI=0x40-0x41 - Emulation Temporaries (MTMP[1-2])

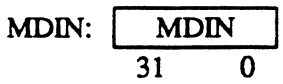
MTMP[1-2] are useful for temporarily keeping non-emulation context while *Viking* enters emulation. These registers are not intended for normal (non-emulation) use. MTMP1 is accessed by ASI 0x40, and MTMP2 is accessed by ASI 0x41. The address is ignored, the size is word, any other size will generate a data_access_exception. These registers are R/W, and are unchanged at reset.



4.15.2. Emulation JTAG Data Input Register (MDIN)

ASI=0x44 - Emulation Data In (MDIN).

A LDA from this ASI space will access the specified MDIN register. Address is ignored. Size is word, any other will generate a data_access_exception. MDIN is read-only, a STA has no effect and will be ignored. The register is unchanged at reset.

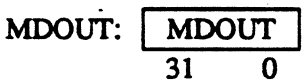


MDIN is a uni-directional port from the emulator to *Viking*. It is used to transfer data from the emulator (via JTAG) into processor visible ASI space. JTAG is the only available mechanism to modify this register.

4.15.3. Emulation JTAG Data Output Register (MDOUT)

ASI=0x46 - Emulation Data Out (MDOUT).

MDOUT is allocated ASI encoding 0x46. The address is ignored. The size is word, any other will generate a data_access_exception. MDOUT is a unidirectional port from *Viking* back to the emulator. *Viking* can read and write this memory mapped resource. The register is unchanged at reset.



4.15.4. Emulation Program Counters

ASI=0x47,0x48 - Emulation Exit PC and NPC.

MPC and MNPC reside in different ASI encodings. Any address within their respective ASI will access that register. Byte, Halfword and Double word accesses to any MPC/MNPC register is explicitly illegal and will generate a data_access_exception.

MPC and MNPC are written by *Viking* upon successful entry to *Viking* emulation mode. They define the PC pair needed upon exit from *Viking* emulation mode to resume execution of the normal instruction stream. Any data written into these register outside of emulation mode is not guaranteed in the presence of emulation activity. They are unchanged at reset.

4.16. ASI Map

This section provides an overview of the *Viking* ASI assignments, which are consistent with the SPARC Reference MMU (SRMMU), and the "Suggested ASI assignments" from the SPARC v8 architecture manual.

Since *Viking* does not generally transmit ASI accesses external to the chip, system visible ASI accesses are limited to: transparent MMU mode, normal data reference, and control space accesses (ASIs 0x20-0x2f, 0x08-0x0a, and 0x02 respectively).

All 8 bits of the ASI are decoded, an error occurs on all access to reserved values. Each supported ASI specifies the types (LD/ST) and size of accesses which are allowed, violations cause errors.

The following table lists all of the ASI values supported by *Viking*. Each has been explained in detail throughout this chapter. A reference to the manual section which describes the ASI is also provide.

Table 4-19 *ASIs supported by Viking*

ASI	Function	Access	Size	Section
0x00-0x01	Reserved	-	-	(None)
0x02	Control Space Access	LD/ST	all	4.10
0x03	RefMMU Flush/Probe	LD/ST	single	4.11.7
0x04	RefMMU Registers	LD/ST	single	4.11.11
0x05	Reserved	-	-	(None)
0x06	RefMMU TLB Diagnostics	LD/ST	single	4.11.12
0x07	Reserved	-	-	(None)
0x08	User Instruction	LD/ST	all	4.5.3
0x09	Supervisor Instruction	LD/ST	all	4.5.3
0x0a	User Data	LD/ST	all	4.5.3
0x0b	Supervisor Data	LD/ST	all	4.5.3
0x0c	Instruction Cache Tags	LD/ST	double	4.7.6
0x0d	Instruction Cache Data	LD/ST	double	4.7.6
0x0e	Data Cache Tags	LD/ST	double	4.8.6
0x0f	Data Cache Data	LD/ST	double	4.8.6
0x10-0x1f	Reserved	-	-	(None)
0x20-0x2f	RefMMU Bypass	LD/ST	all	4.5.3
0x30	Store Buffer Tags	LD/ST	double	4.12
0x31	Store Buffer Data	LD/ST	double	4.12
0x32	Store Buffer Control	LD/ST	single	4.12
0x33-0x35	Reserved	-	-	(None)
0x36	Instruction Cache Flash Clear	ST	single	4.7.6
0x37	Data Cache Flash Clear	ST	single	4.8.6
0x38	MMU Breakpoint Diagnostics	LD/ST	double	4.14.4
0x39	BIST Diagnostics	LD/ST	single	4.3.5
0x3a-0x3f	Reserved	-	-	(None)
0x40-0x41	Emulation Temps[1-2]	LD/ST	single	4.15
0x42-0x43	Reserved	-	-	(None)
0x44	Emulation Data In1	LD	single	4.15
0x45	Reserved	-	-	(None)
0x46	Emulation Data Out	LD/ST	single	4.15
0x47	Emulation Exit PC	LD/ST	single	4.15
0x48	Emulation Exit NPC	LD/ST	single	4.15
0x49	Emulation Counter Value	LD/ST	single	4.14
0x4a	Emulation Counter Mode	LD/ST	single	4.14
0x4b	Emulation Counter Status	LD/ST	single	4.14
0x4c	ACTION register	LD/ST	single	4.14
0x4d-0xff	Unassigned	-	-	(None)

Notes:

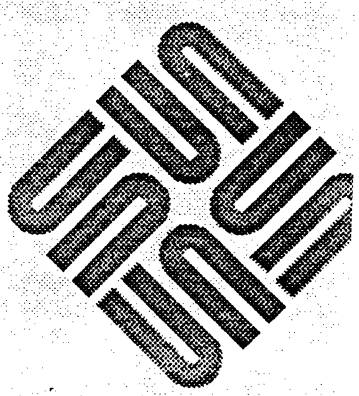
The instructions LDSTUBA and SWAPA generate data_access_exception on ASI values other than 0x08-0x0b or 0x20-0x2f.

Any ASI access with a size other than indicated on table 4-19 will generate data_access_exception.

[Blank Page]

5

JTAG Serial Scan Interface



5

JTAG Serial Scan Interface

5.1. Overview

Viking provides the IEEE P1149.1 JTAG serial scan interface mechanism to allow observation and control for different applications:

5.1.1. To implement manufacturing fault coverage

JTAG allows test software access to internal scan logic to determine device manufacturing correctness.

5.1.2. To permit periphery and interconnect testing

JTAG allows software controlled boundary scan, to test the periphery and interconnect between chips on boards which use *Viking*. Boundary scan testing requires software that uses JTAG to scan-in, apply, scan-out and compare vectors.

5.1.3. To permit built-in self test

Using internal scan, *Viking* provides BIST. In addition to software initiation using ASI 0x39, *Viking* BIST can be initiated by software that uses the JTAG mechanism. (See section 4.3.5, — *Built-In Self Test (BIST)* for more details on BIST).

5.1.4. To support a remote debugging environment

Remote emulation debug software can use JTAG to halt an application program, examine or alter register and memory state (including the program counters), set breakpoints or counters (to specify conditions where control should leave the application code and return back to the emulator), and to resume control within the application code. Such software can download SPARC assembly language for the intended emulation function, inspect device specific JTAG status and provide a recovery mechanism when emulation instructions fault. See section 6, — *Remote Emulation Support* for more details on emulation.

5.2. JTAG Requirements

Viking requires that the internal JTAG TAP (Test Access Port) controller be *reset* before normal system operation begins. Also, systems which do *not* provide a TCK input to *Viking* during normal system operation must assert TRST_ during power-on reset. *Viking* requires that the external JTAG busmaster TAP controller remain *in synchronization* with *Viking's* internal JTAG TAP controller. See section 4-1, — *State after hardware reset* for more details on *Viking* hardware reset.

> Important Note:
> All systems (with TCK active) which uses the TRST_ assertion to reset *Viking*'s internal JTAG TAP controller *must*
>
> Keep TMS asserted during TRST_ assertion.
> Hold TMS asserted for a minimum of 3 TCK cycles (as seen by *Viking*) after negating TRST_.

After the three TCK cycles have elapsed, the external JTAG busmaster is allowed to negate TMS.

5.3. JTAG Interface

The IEEE P1149.1 JTAG serial scan interface mechanism is composed of a set of pins and TAP controller state machine that responds to those pins. The design is partitioned into several serially scanned 'rings' which are independently accessed. Ring access selection is determined by the IR (Instruction Register). Access into the IR scan chain is in accordance with the JTAG protocol.

The *Viking* JTAG interface is composed of 5 wires:

Test Clock (TCK)
Test Mode Select (TMS)
Test Logic Reset (TRST_)
Test Data In (TDI)
Test Data Out (TDO)

5.4. JTAG Operations

JTAG defines three basic scan chain operations:

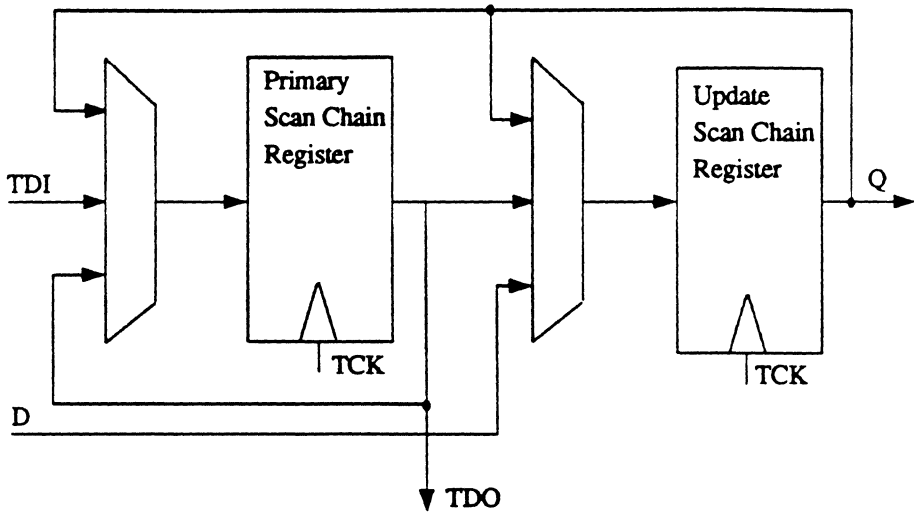
CAPTURE
SHIFT
UPDATE

Each of the serially scanned rings internal to *Viking* is composed of a reconfigurable shift register chain. Each stage of the scan chain has two register chains of equal length:

The *primary* scan chain register
The *update* scan chain register

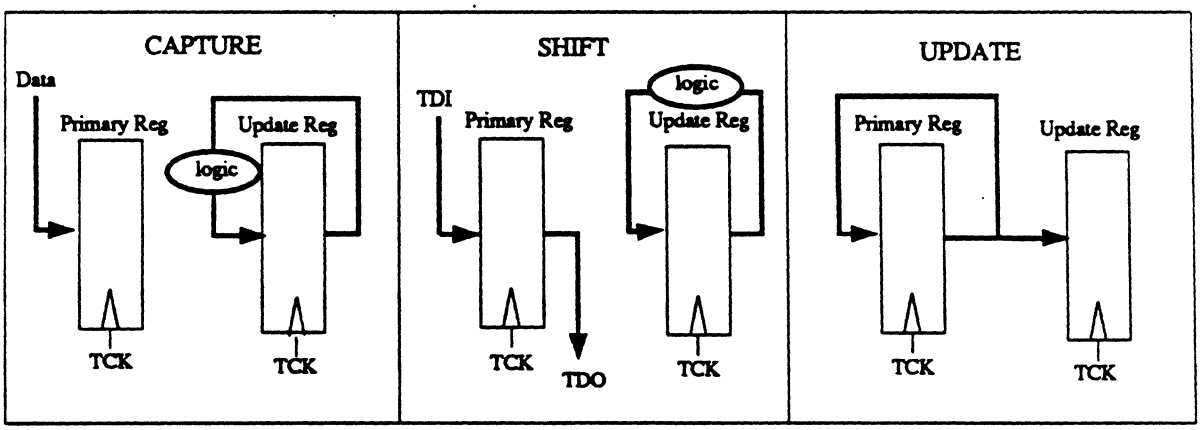
The *primary* scan chain register is configurable as a shift register, while the *update* register is not. (See the IEEE P1149.1 JTAG Specification for more details). The functionality of a single bit JTAG scan chain register is:

Figure 5-1 *One Bit JTAG Scan Chain Datapath Element*



The diagram below illustrates the three operations in a simplified functional view.

Figure 5-2 *JTAG operations - CAPTURE, SHIFT, and UPDATE*



5.4.1. JTAG CAPTURE operation

During a CAPTURE operation, the JTAG Scan Chain element captures selected data into the primary register. Different data is captured for different instructions.

BYPASS	- captures 1-bit zero
CID	- captures component ID 0x0000402f
BSCAN_INTEST	- captures values at output buffers
BSCAN_EXTEST	- captures values at output pins
BSCAN_SAMPLE	- captures values at all pins
Signature	- captures the signature register
MSTAT	- captures MSTAT register
MDOUT	- captures MDOUT register
others	- either captures the update register or has no effect

After one TCK, those captured values are ready to be shifted out (using SHIFT operation) to the Viking level TDO at the end of the scan chain ring. In some specific cases, DRCAPTURE captures the value of a particular register into the primary register. This capability allows capturing of internal logic states, and the information is then shifted out to be read. The MDOUT register provides this capability.

5.4.2. JTAG SHIFT operation

During a SHIFT operation, the JTAG Scan Chain element operates as a shift register. Each primary register stores its TDI value and shifts forward its TDO in the next TCK cycle, as input to the next primary register in the scan chain. For a ring of size N, the TAP controller must shift N times to completely fill the scan chain. During this operation, the update register is unused, and retains its value.

5.4.3. JTAG UPDATE operation

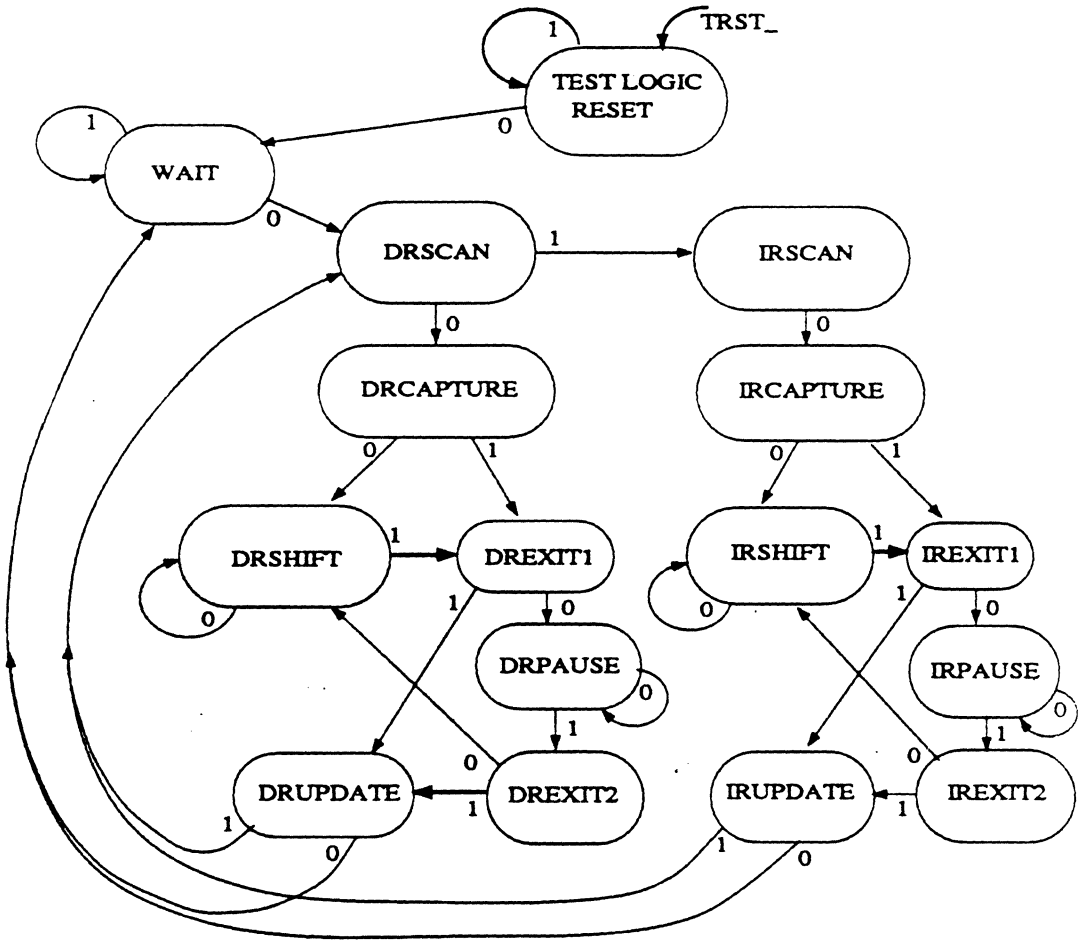
During an UPDATE operation, the JTAG Scan Chain element delivers the value contained in the primary register into the update register. Except for this overwrite period, the update register retains its previous value. *Viking* internal logic only sees the the value contained in the update register. A single UPDATE cycle will deliver the intended values to the update register scan chain. The UPDATE is performed after all the values have been shifted into the primary scan chain registers. During this operation, the primary register retains its value. UPDATE is ignored by *non-writable* registers.

5.5. TAP Controller

The Test Access Point (TAP) controller is a *Viking* internal sequencer which manages access to all JTAG scan register rings. It ignores TDI/TDO. The TAP controller examines TRST_ and TMS sequences each cycle for JTAG state transitions. The state of the TAP controls assertion of CAPTURE, SHIFT, and UPDATE operations. See the IEEE P1149.1 JTAG Specification for more details.

The TAP controller state transition diagram is:

Figure 5-3 *JTAG TAP Controller State Transition Diagram*



Two things guarantee that the TAP controller enters the TEST LOGIC RESET state, assertion of TMS for five consecutive TCK cycles, or a single assertion of TRST_. Both TMS and TRST_ must be negated to exit the TEST LOGIC RESET state, into the WAIT state. There are 2 basic TAP Controller state groups: those related to the IR (Instruction Register) scan chain related to a TDR (Test Data Register) scan chain. Each state group is composed of 7 member states (see diagram):

Three states implement CAPTURE, SHIFT, UPDATE operations within the selected scan chain.
One state is used to stop midway through a SHIFT sequence into the selected scan chain.
Three states make transitions into other states.

Reading JTAG registers requires a CAPTURE followed by SHIFT. *Writing* JTAG registers requires SHIFT followed by UPDATE. After a complete (write in) SHIFT

sequence, an UPDATE is caused to occur by two consecutive TCK cycles with TMS = 1. Before a readout SHIFT sequence, a CAPTURE should take place. Recall that UPDATE will be ignored for any *non-writable* registers.

5.5.1. JTAG Reset Requirements

The JTAG TAP controller must be reset at power-up to guarantee correct *Viking* operation. If TCK is not present, TRST_ must be asserted at power-up, to properly reset the TAP controller. Otherwise, the JTAG IR will be in an *indeterminant* state which could result in undefined operations.

> Recommendation:
> If JTAG is not used in a system, TRST_ should be asserted to avoid unintended JTAG operation.

5.5.2. Effects Of TAP Reset

The following table show what internal states will be selected when TAP reset (TEST LOGIC RESET) state is entered

Table 5-1 *State after TAP reset*

Register/Internal state Affected	State After TAP Reset
The IR register	0x10 (CID TDR selected)
SHIFT, UPDATE, CAPTURE	negated
select_ir signal	negated
enable_tdo signal	negated
JTAG UPDATE register output_enable bit	negated
MCMD register	cleared
MSTAT register	cleared

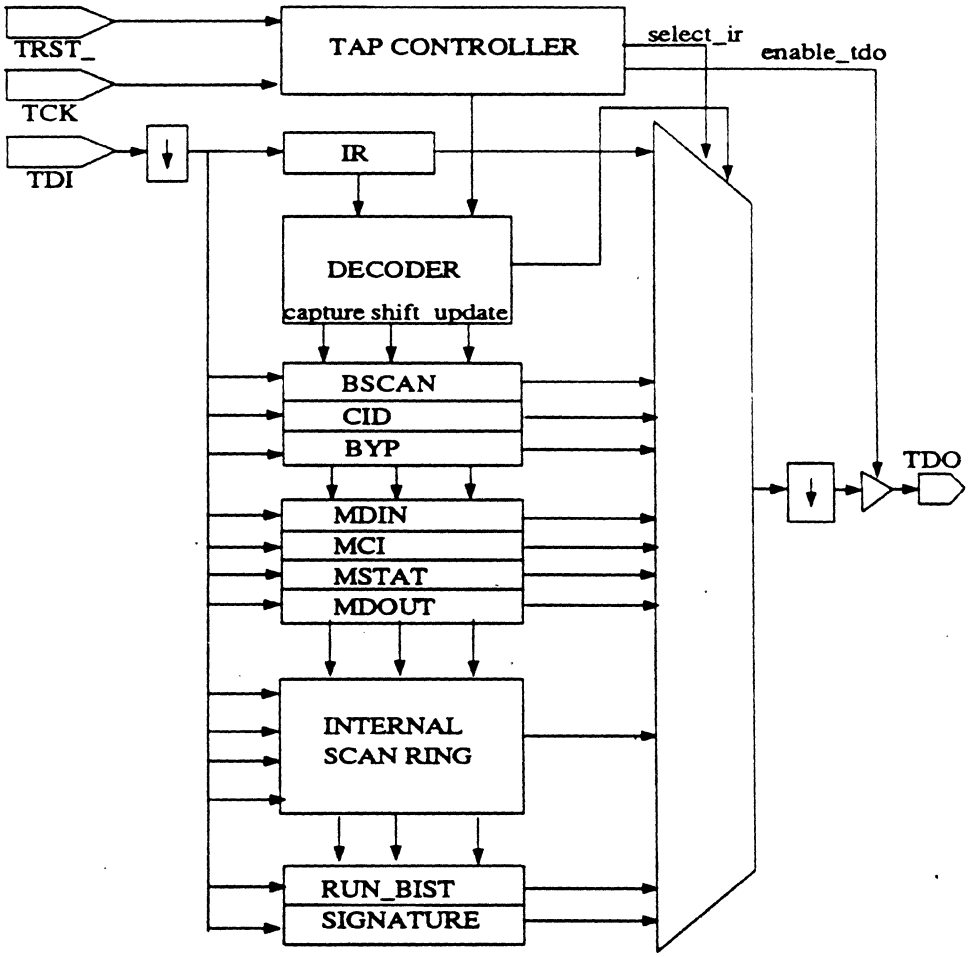
5.6. Accessible Scan Chains inside *Viking*

There are 5 scan chains inside *Viking* that are accessible through JTAG, they are:

The IR register domain
The standard JTAG register domains (BYPASS, CID, BSCAN)
An internal hardware test domain (internal use only)
The BIST register domains (SHORT_BIST, LONG_BIST, SIGNATURE)
The emulation register domains (MDIN, MCI, MSTAT, MDOUT)

The block diagram of all JTAG accessible serial scan chains is:

Figure 5-4 *Block Diagram of JTAG Scan Chains inside Viking*



The IR register consists of one 5-bit scan chain, and its value selects the TDR scan chain for subsequent SHIFT, UPDATE, and CAPTURE operations.

The emulation register domain consists of 4 scan chains: MDIN, MCI, MSTAT and MDOUT. Two are input, and the other two output. MDIN and MCI provide information from the remote emulation processor to *Viking* internals, while MSTAT and MDOUT brings information from *Viking* to the remote emulation processor. See chapter 6, — *Remote Emulation Support* for more details.

Hardware test domains are for manufacturing use only, other usage will provide undefined results.

The BIST register domain permits a BIST sequencer to internally generate, scan-in apply, scan-out and obtain a signature for state vectors.

5.7. IR Format and Encodings

The *Viking* 5-bit IR (Instruction Register) selects TDR scan chains for the SHIFT, UPDATE, and CAPTURE operations. The IR scan ring is selected when the JTAG TAP controller is in the IRCAPTURE, IRSHIFT and IRUPDATE states. An IRCAPTURE does not capture the current value of the IR update register. Instead it returns a fixed binary encoding of 00001. The low 2 bits of this encoding is required by the IEEE P1149.1 JTAG Specification (under IR rule D).

IEEE 1149.1 identifies public and private instructions for IR encodings. *Viking* offers 18 IR *instructions*, selected by a 5-bit IR. Most *instructions* select which serial TDR ring to operate on during a JTAG "test data register" access, and two *instructions* initiate BIST (long and short).
The 18 IR *instructions* are categorized into 7 categories:

Table 5-2 *Categories of Viking IR instructions*

Category	Number of *instructions*
Basic scan rings	2 instructions: CID, BYPASS
Boundary scan rings	3 instructions: Extest, Sample, Intest
Examine PLL clock	1 instruction: SeePLL
Internal manufacturing scan rings	5 instructions
Initiate BIST	2 instructions: start long/short BIST
BIST signature ring	1 instruction: Signature
Emulation scan rings	4 instructions: MCI, MSTAT, MDIN, MDOUT

The IR encoding to select access to a particular *Viking* TDR scan chain is:

Table 5-3 *TDR Scan Chain selection by IR Encoding*

TDR Ring Selected	IR Value	# bits
BSCAN_EXTEST	0x00	290
BSCAN_SAMPLE	0x01	290
BSCAN_INTEST	0x02	290
Internal_Scan_Capture_Clock_Mode	0x03	n/a
Internal Scan Domain 0	0x04	n/a
Internal Scan Domain 1	0x05	n/a
Internal Scan Domain 2	0x06	n/a
Internal Scan Domain 3	0x07	n/a
MCI	0x08	37
MDIN	0x09	32
MDOUT	0x0a	32
MSTAT	0x0b	13
SIGNATURE	0x0c	32
RUN_SHORT_BIST	0x0d	n/a
RUN_LONG_BIST	0x0e	n/a
SeePLL	0x0f	n/a
CID	0x10-0x1e	32
BYPASS	0x1f	1

Three are *standard* JTAG TDR scan chains (BYPASS,CID,BSCAN). The rest are *Viking* specific scan chains.

5.7.1. Public JTAG instructions and scan rings

Viking provides 12 *public* JTAG *instructions*, which belong to the following categories:

Public JTAG instruction
BYPASS
CID
BSCAN
BIST
Signature
Emulation

A brief description of actions that occur during CAPTURE, SHIFT, and UPDATE operations at the different scan chains are given below. For JTAG more details, see the IEEE P1149.1 JTAG Specification.

5.7.1.1 BYPASS

The BYPASS scan chain comprises a 1-bit primary scan register, with no update register. When IR selects BYPASS, the chip's TDI and TDO are essentially connected to this primary register, and DRSHIFT only requires one TCK cycle to forward data. And DRCAPTURE loads a 1-bit zero into the primary register. DRUPDATE has no effect.

5.7.1.2 CID

The CID scan chain comprises a 32-bit ring of primary registers, where bit[0] is always 1, bit[11-1] the manufacturer ID, bit[27-12] the part number, and bit[31-28] the version number. DRCAPTURE loads the value 0x0000402f into the CID primary registers scan chain. Subsequent DRSHIFT cycles shift (bit[0]) out and scan new TDI data in (bit[31]). There is no update register, and DRUPDATE has no effect.

> Note:
> *Viking* CID value is 0x0000402f.

5.7.1.3 BSCAN

Viking BSCAN is a 290-bit ring of JTAG scan chain register elements. DRCAPTURE reads data from the chip pins into the primary register. DRSHIFT forwards data through the scan chain, where bit[0] is output to TDO. For the entire chain to be completely written in or read out, 290 TCK cycles are needed. DRUPDATE copies data from the primary register into the update register, taking only 1 TCK cycle. The following is *Viking* boundary scan map:

Table 5-4 *Viking Boundary Scan bit definition*

Bit	Signal	Bit	Signal	Bit	Signal	Bit	Signal
0	reset_in	1	test_in	2	srmtst_in	3	ccrdy_in
4	data63_in	5	data63_out	6	data62_in	7	data62_out
8	data61_in	9	data61_out	10	data60_in	11	data60_out
12	data59_in	13	data59_out	14	data58_in	15	data58_out
16	data57_in	17	data57_out	18	data56_in	19	data56_out
20	data55_in	21	data55_out	22	data54_in	23	data54_out
24	data53_in	25	data53_out	26	data52_in	27	data52_out
28	data51_in	29	data51_out	30	data50_in	31	data50_out
32	data49_in	33	data49_out	34	data48_in	35	data48_out
36	data47_in	37	data47_out	38	data46_in	39	data46_out
40	data45_in	41	data45_out	42	data44_in	43	data44_out
44	vck_in	45	pllbyp_in	46	data43_in	47	data43_out
48	data42_in	49	data42_out	50	data41_in	51	data41_out
52	data40_in	53	data40_out	54	data39_in	55	data39_out
56	data38_in	57	data38_out	58	data37_in	59	data37_out
60	data36_in	61	data36_out	62	data35_in	63	data35_out
64	data34_in	65	data34_out	66	data33_in	67	data33_out
68	data32_in	69	data32_out	70	we3_out	71	we2_out
72	we1_out	73	we0_out	74	dpar3_in	75	dpar3_out
76	dpar2_in	77	dpar2_out	78	dpar1_in	79	dpar1_out
80	dpar0_in	81	dpar0_out	82	pend_in	83	size1_out
84	size0_out	85	owner_in	86	owner_out	87	shared_in
88	shared_out	89	error_out	90	cchbl_out	91	su_out
92	addr23_in	93	addr23_out	94	addr22_in	95	addr22_out
96	addr21_in	97	addr21_out	98	addr20_in	99	addr20_out
100	addr19_in	101	addr19_out	102	addr18_in	103	addr18_out
104	addr17_in	105	addr17_out	106	addr16_in	107	addr16_out
108	addr15_in	109	addr15_out	110	addr14_in	111	addr14_out
112	addr13_in	113	addr13_out	114	addr12_in	115	addr12_out
116	daboe_out	117	addr11_in	118	addr11_out	119	addr10_in
120	addr10_out	121	adboe_out	122	addr09_in	123	addr09_out
124	addr08_in	125	addr08_out	126	dpboe_out	127	addr07_in
128	addr07_out	129	srboe_out	130	addr06_in	131	addr06_out
132	allboe_out	133	erboe_out	134	addr05_in	135	addr05_out
136	addr04_in	137	addr04_out	138	snboe_out	139	addr03_in
140	addr03_out	141	mbboe_out	142	addr02_in	143	addr02_out
144	mrboe_out	145	stboe_out	146	addr01_in	147	addr01_out
148	addr00_in	149	addr00_out	150	oe_in	151	oe_out
152	wr_in	153	wr_out	154	oeboe_out	155	rd_in
156	rd_out	157	burst_out	158	retry_in	159	wee_in
160	wee_out	161	mexc_in	162	ardy_in	163	ardy_out
164	busreq_out	165	wrdy_in	166	wrdy_out	167	rrdy_in
168	wgrt_in	169	rgrt_in	170	cmds_in	171	cmds_out
172	demap_in	173	demap_out	174	csa_out	175	ldst_out

(continued)

Bit	Signal	Bit	Signal	Bit	Signal	Bit	Signal
176	dpar4_in	177	dpar4_out	178	dpar5_in	179	dpar5_out
180	dpar6_in	181	dpar6_out	182	dpar7_in	183	dpar7_out
184	we4_out	185	we5_out	186	we6_out	187	we7_out
188	data0_in	189	data0_out	190	data1_in	191	data1_out
192	data2_in	193	data2_out	194	data3_in	195	data3_out
196	data4_in	197	data4_out	198	data5_in	199	data5_out
200	data6_in	201	data6_out	202	data7_in	203	data7_out
204	data8_in	205	data8_out	206	data9_in	207	data9_out
208	data10_in	209	data10_out	210	data11_in	211	data11_out
212	data12_in	213	data12_out	214	data13_in	215	data13_out
216	data14_in	217	data14_out	218	data15_in	219	data15_out
220	data16_in	221	data16_out	222	data17_in	223	data17_out
224	data18_in	225	data18_out	226	data19_in	227	data19_out
228	data20_in	229	data20_out	230	data21_in	231	data21_out
232	data22_in	233	data22_out	234	data23_in	235	data23_out
236	data24_in	237	data24_out	238	data25_in	239	data25_out
240	data26_in	241	data26_out	242	data27_in	243	data27_out
244	data28_in	245	data28_out	246	data29_in	247	data29_out
248	data30_in	249	data30_out	250	data31_in	251	data31_out
252	pipe00_out	253	pipe01_out	254	pipe02_out	255	pipe03_out
256	pipe04_out	257	pipe05_out	258	pipe06_out	259	pipe07_out
260	pipe08_out	261	pipe09_out	262	irl0_in	263	irl1_in
264	irl2_in	265	irl3_in	266	addr24_in	267	addr24_out
268	addr25_in	269	addr25_out	270	addr26_in	271	addr26_out
272	addr27_in	273	addr27_out	274	addr28_in	275	addr28_out
276	addr29_in	277	addr29_out	278	addr30_in	279	addr30_out
280	addr31_in	281	addr31_out	282	addr32_in	283	addr32_out
284	addr33_in	285	addr33_out	286	addr34_in	287	addr34_out
288	addr35_in	289	addr35_out				

5.7.1.4 SHORT_BIST, LONG_BIST

The *Viking* BIST mechanism is initiated by or examined through either JTAG or ASI memory references. When BIST is initiated, any pre-BIST *Viking* state will be destroyed. At the completion of a JTAG initiated BIST, the user needs to generate the reset. This can be done by entering the TAP reset state by either assertion of TMS for 5 consecutive TCK cycles or asserting TRST_. Internal timing sequencing will guarantee PLL restabilization. Once initiated, BIST will be under the control of *Viking*, and the JTAG TAP controller need not remain in WAIT/RUN_BIST state. See section 4.3.5, — *Built-In Self Test (BIST)* for more details. When IR selects BIST, CAPTURE will return BIST_DONE status. UPDATE has no effect. At the completion of BIST, a CAPTURE of the SIGNATURE TDR should be done.

5.7.1.5 Signature

Signature scan chain is a 32-bit ring. Both long and short BIST operations cause the BIST sequencer to generate, scan-in, apply, and scan-out a signature for one of two pre-defined pseudo-random test vectors. The *Viking* response to these vectors is collected and compressed into the signature register. The correct value of the signature register will be different for these two cases. See section 4.3.5, — *Built-In Self Test (BIST)* for more details.

5.7.1.6 Emulation

There are four independent scan rings associated with remote emulation: MDIN, MCI, MSTAT, and MDOUT. See section 6.3.1.3, — *MCI (Emulation Command and Instruction)* for more details.

The MDIN scan chain is a 32-bit ring. DRSHIFT shifts data from a primary register into the next primary register. Bit[0] goes out to TDO, while bit[31] reads in TDI. DRUPDATE copies data from the primary register into the update register. DRCAPTURE has no effect. See section 6.3.1.2, — *MDIN (Emulation Data In)* for more details.

The MCI scan chain is a 37-bit ring, covering both subfields MCMD and MINST. DRSHIFT shifts the primary register to the next primary register in the chain, where bit[36] reads in TDI, and bit[0] outputs TDO. DRUPDATE copies data from the primary register into the update register. DRCAPTURE has no effect.

The MSTAT scan chain is a 13-bit ring. DRCAPTURE captures data from the MSTAT register into the primary register. DRSHIFT shifts the primary register to the next primary register in the chain, where bit[12] reads in TDI, and bit[0] outputs TDO. DRUPDATE copies data from the primary register into the update register.

The MDOUT scan chain is a 32-bit ring. DRCAPTURE captures data from the MDOUT register into the primary register. DRSHIFT shifts the primary register to the next primary register in the chain, where bit[31] reads in TDI, and bit[0] outputs TDO. DRUPDATE copies data from the primary register into the update register.

5.7.2. Private JTAG instructions and scan rings

Viking provides 6 *private* JTAG *instructions*, which belong to the following categories:

Private JTAG instruction
SEE_PLL
INTERNAL_SCAN_CAPTURE_CLOCK_MODE
INTERNAL_SCAN_DOMAIN[0-3]

5.7.2.1 SEE_PLL

This scan chain is used to verify the integrity of the on-chip PLL with respect to clock jitter and VCO behavior. When selected (refer to table 5-3, — *TDR Scan Chain selection by IR Encoding*), the scan chain will output PLL clock on TDO. This scan should *not* be used when *Viking* is plugged into a board.

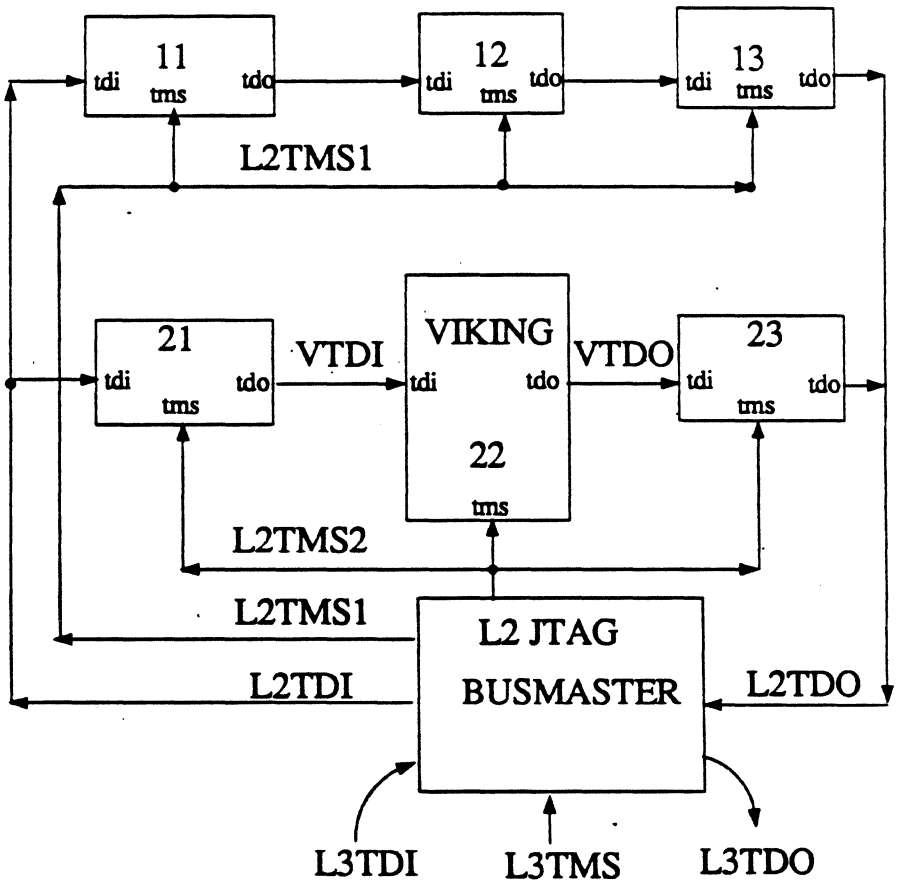
5.7.2.2 INTERNAL_SCAN

This scan chain is used for hardware manufacturing tests, details are not provided in this manual.

5.8. System Level Test

The IEEE P1149.1 JTAG Specification (chapter *Legal Interconnections Of Components Compatible With P1149.1*) provides guidelines on how to connect TCK, TMS, TDI and TDO for making serial, parallel and hierarchical configurations for JTAG board level sub-systems. This allows a hierarchical JTAG test technique, and *Viking* supports it. Similar in ways with how *Viking* selects TDR scan chains, a second level JTAG controller (e.g. board level JTAG busmaster) may connect multiple chips on a board to form parallel and serial paths. Furthermore, a third level JTAG controller (e.g. backplane level JTAG busmaster) may connect multiple second level JTAG controllers to form a chain.

Figure 5-5 *Example of System Level JTAG Test Hierarchy*



The example shows one board level JTAG busmaster controlling 6 JTAG components configured into 2 parallel daisy chains. Each component in the chain connects its TDI to the preceding chip's TDO. The first chip in each of the parallel daisy chains is tied to a common level-2 TDI, and the last chip in each chain is *wire-ORed* to a common level-2 TDO. Each of the parallel chains receives a unique TMS from the second level TAP controller.

6

Remote Emulation Support

6

Remote Emulation Support

Viking provides facilities to observe and control processor execution from a remote device using the IEEE P1149.1 JTAG serial scan interface, called *in circuit emulation*. Their use is described in this chapter, and the JTAG interface is described in chapter 5.

6.1. Overview

Traditionally, systems debugging methods required very expensive dedicated add-on hardware, connected using ribbon cables and fragile connectors, and are *not* generally available when the first processor prototypes are delivered (which is when they would have been most useful). Furthermore, the length of the cable was limiting the processor system clock rate when the emulator is being used, and also introduced extra electrical loading which affected pin timings.

As a solution to that problem, *Viking* provides *in circuit emulation* to support a remote environment for debugging systems, done entirely over the serial JTAG bus. The features are useful for both *hardware and software development*. It is completely non-intrusive into the system design. Neither the pin timings nor the processor speed are affected. Nearly all features of traditional emulators are provided, except for real-time trace and memory emulation.

All programmer visible state is accessible, and changeable using the JTAG interface. Software must be provided to control the emulation from a remote computer with a JTAG interface. During emulation, the caches and store buffer continue to operate; these resources will snoop incoming system bus requests. Many emulation resources are shared with standard software debugging features.

6.2. Emulation Strategy

Viking provides *virtual in-circuit emulation* (VICE), or also known as *remote emulation*. The remote emulator's interface to *Viking* uses the JTAG IR (instruction register) to select one of four emulation JTAG TDRs. The remote emulator provides *Viking* with emulation protocol commands, emulation instructions, addresses and data for updating *Viking*'s state through two JTAG TDR registers: MCI (eMulation Command and Instruction), and MDIN (eMulation Data In). *Viking* returns existing *Viking* system state data and emulation protocol status through two other JTAG TDR registers: MDOUT (eMulation Data Out) and MSTAT (eMulation STATus).

Through these registers, the remote emulator can command the processor to temporarily halt execution of the normal SPARC instruction stream. Once halted, the

processor can be directed to execute any normal SPARC instruction. No processor state information which is not explicitly modified by this emulation instruction will be altered.

An example of the simplest emulation sequence is to allow the remote emulator to observe an internal *Viking* register. To achieve this, the emulator halts the processor, scans in a SPARC instruction to write the desired register into the MDOUT register, then scans out the MDOUT JTAG TDR. Other sequences can be significantly more complex.

6.3. Emulation Register Set

The control registers for emulation exist in two register sets:

Type	Emulation registers
JTAG TDR	IR, MDIN, MCI, MDOUT, MSTAT
ASI space	PC/NPC, MTMP[1-2], MDIN (ASI 0x44), MDOUT (ASI 0x46)

Note that the JTAG TDR MDIN and MDOUT are accessible through ASI space as well. These registers are briefly described below, but for more details on the JTAG TDR operations, see section 5. For more details on the ASI visible emulation registers, see section 4.16.

6.3.1. JTAG TDRs emulation registers

There are five JTAG TDRs that are important for emulation, they are listed on the table below:

JTAG TDR	Name
IR	(Instruction Register)
MCI	(eMulation Command and Instruction)
MDIN	(eMulation Data In)
MDOUT	(eMulation Data OUT)
MSTAT	(eMulation STATus)

See chapter 5 for details on JTAG TDR scan operation.

6.3.1.1 IR (Instruction Register)

The IR register selects which JTAG TDR scan chain to access. The following table only lists IR encoding that select emulation register JTAG TDR scan chains. For a complete listing see table 5-3 and section 5.7.

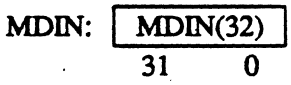
Table 6-1 *Emulation register TDR Scan Chain selection by IR Encoding*

TDR Ring Selected	IR Value	# bits
MCI	0x08	37
MDIN	0x09	32
MDOUT	0x0a	32
MSTAT	0x0b	13

6.3.1.2 MDIN (Emulation Data In)

This 32-bit register allows information passing from the remote emulator to *Viking*. For example, the remote emulator scans in data to the MDIN register, and *Viking* retrieve the data using LDA ASI 0x44. It is particularly useful to set pointers to memory references, to update register or memory values. Refer to sections 5.7.1.6 — *Emulation* and 4.15.2 — *Emulation JTAG Data Input Register (MDIN)* for more details.

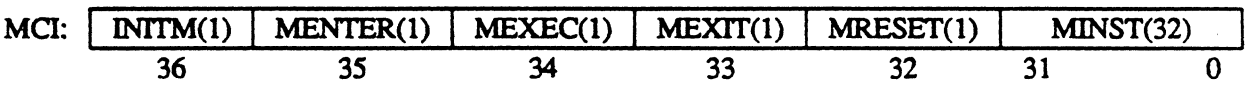
Table 6-2 *MDIN Scan Register Format*



6.3.1.3 MCI (Emulation Command and Instruction)

The MCI scan chain is 37 bits and comprises 2 subfields: a 5-bit MCMD register and a 32-bit MINST register. MCMD qualifies MINST and sends configuration and command information to *Viking*. All bits in MCMD are cleared upon JTAG TAP controller reset. (Refer to table 5-1 — *State after TAP reset*). Also see section 5.7.1.6 for details on JTAG operation. The format of the JTAG MCI scan register {MCMD,MINST} is shown below:

Figure 6-1 *MCI (Emulation Command and Instruction) Register Format*



MCMD.INITM

The primary function of this bit is to enable *Viking* to enter emulation mode as a result of a breakpoint, assuming ACTION register is properly programmed. In addition, this bit affects the operation of SIGM (Signal Emulation, a *Viking-specific* instruction). If INITM=0, SIGM will execute as a NOP. If INITM=1, SIGM initiates a user level emulation entry.

(Refer to section 4.4.6 — *Signal User Emulation Request (SIGM)* and 4.14.4.5 — *Breakpoint ACTION Register*). This bit is cleared on JTAG TAP controller reset.

MCMD.MRESET
When set, this bit forces a full *Viking* hardware reset (rather than a watch dog reset). This bit is cleared on JTAG TAP controller reset.

MCMD.MENTER
When set, *Viking* is forced to enter emulation mode. The program counter (PC/NPC pair) is captured to resume post-emulation execution. In emulation mode, the *Viking* prefetch controller stops accessing the instruction cache, starts passing NOPs into the IU pipeline and examines the state of MEXEC to wait for an instruction to execute. This bit is cleared on JTAG TAP controller reset.

MCMD.MEXEC
When set, a single instance of the MINST instruction will be forced into the processor pipeline. This will cause it to be executed as a normal SPARC instruction. Once launched, the prefetch controller will clear MEXEC, resume passing NOPs into the IU pipeline and monitor valid bits at the last stage of the IU pipeline. Once the prefetch controller determines that no remaining emulation instructions are in the processor pipeline, it examines MEXIT to determine whether to remain in emulation mode or to resume normal execution. A floating point related emulation instruction (LDF, STF, LDFSR, STFSR) may only be issued when the FP queue is empty (signalled by the MSTAT.FQE bit). This MEXEC bit is cleared on JTAG TAP controller reset.

MCMD.MEXIT
When set, *Viking* will exit emulation mode (to resume normal execution) as soon as all execution in the pipeline is complete. The execution stream branches to the PC/NPC values stored upon entry to emulation (and possibly intentionally modified during emulation). The prefetch controller continues to pass NOPs into the pipeline until either an MEXEC or MEXIT is asserted. This bit is cleared on JTAG TAP controller reset.

MINST This register contains a single SPARC instruction which is emitted as the emulation instruction (qualified by MEXEC). Several emulation instructions are typically required to completely execute the semantics of an emulator primitive.

6.3.1.4 MDOUT (Emulation Data Out)

This register stores data to be passed back to the remote emulator. It can be used to pass *Viking* state information to the remote emulator. This register is accessible through ASI space 0x46 (refer to section 4.15.3). The format of the MDOUT register is:

Table 6-3 *MDOUT (Emulation Data Out) Register Format*

MDOUT:	MDOUT(32)
	31 0

6.3.1.5 MSTAT (Emulation Status)

The MSTAT scan chain is 13 bits long, and it contains information about *Viking* emulation status. The only way to retrieve this information is through JTAG scan operation. The MSTAT register is cleared upon TAP Controller reset. The MSTAT format is:

Table 6-4 *MSTAT (Emulation Status) Register Format*

MSTAT:	ECHOTMR	MACK	TMRM	CBKM	ZICM	DBKM
	12	11	10	9	8	7

ZCCM	IPND	ERRMODE	MIFLTD	PFPX
6	5	4	3	2

MIDONE	FQE
1	0

MSTAT.ECHOTMR

ECHOTMR is an echoed version of MCMD.MENTER after it has passed through TCK-VCK-TCK synchronization. (TCK is test clock, VCK is *Viking* clock). ECHOTMR is asserted asynchronously to emulation instruction execution. The purpose of this signal is to signal that *Viking* has *seen* the request to enter emulation mode. Assertion of this signal does *not* indicate that *Viking* has actually entered emulation mode. This bit is cleared upon the JTAG TAP Controller reset.

MSTAT.MACK

MACK is an indication that *Viking is in emulation mode.* It is asserted as soon as *Viking* enters emulation mode and stays active until *Viking* leaves emulation mode. It is synchronized to TCK (refer to chapter 5. This bit is cleared upon the TAP Controller reset.

> Note:
> TMRM, CBKM, ZICM, DBKM and ZCCM qualify MACK to identify to the remote emulation service processor software the cause for *Viking*'s entry into emulation mode. It is possible that more than one is asserted, indicating that there were more than one cause.

MSTAT.TMRM

TMRM indicates that *Viking* entered emulation due to an MCMD.MENTER

request from the emulator. This bit is cleared on JTAG TAP Controller reset, *Viking* hardware reset, or by updating the MCI register.

MSTAT.CBKM

CBKM indicates that *Viking* entered emulation *due to a code address breakpoint*. Cleared on JTAG TAP Controller reset, *Viking* hardware reset, or by updating the MCI register.

MSTAT.ZICM

ZICM indicates that *Viking* entered emulation *due to a zero instruction count breakpoint*. This bit is cleared on JTAG TAP Controller reset, *Viking* reset, or by updating the MCI register.

MSTAT.DBKM

DBKM indicates that *Viking* entered emulation *due to a data address breakpoint*. This bit is cleared on JTAG TAP Controller reset, *Viking* reset, or by updating the MCI register.

MSTAT.ZCCM

ZCCM indicates that *Viking* entered emulation *due to a zero cycle count breakpoint*. This bit is cleared upon a JTAG TAP Controller reset, *Viking* reset, or by updating the MCI register.

MSTAT.IPND

IPND assertion indicates the processor has a *pending interrupt request* that is higher than the current *Viking* PSR.IPL or at level 15. IPND is used to inform the remote emulation processor that an interrupt is pending and that *Viking* should be released to service it. This bit is cleared upon the JTAG TAP Controller reset.

MSTAT.ERRMODE

ERRMODE indicates *Viking* has *entered error mode* as a result of a fault generated while in emulation mode. Any exception occurring during an emulation instruction sequence will force *Viking* into error mode. Entering error mode will induce the watch dog reset sequence, set MSTAT.ERRMODE, and exit emulation mode. The ERRMODE bit will stay asserted until the next MSTAT update operation (or cleared by the JTAG TAP Controller reset).

> Important Note:
> When error mode occurs while in emulation mode, no assertion of MEXIT or MRESET is needed for *Viking* to leave emulation and restart execution at the reset vector.

MSTAT.MIFLTD

MIFLTD indicates that an emulation instruction with a data memory reference created a data access fault. It does not indicate whether other sources of exceptions (e.g. pending FP exceptions) occurred during emulation instruction execution. MIFLTD is only meaningful when qualified by the assertion of MIDONE status bit. No emulation instruction exception update MFSR or MFAR. Only MSFSR (the *shadow* FSR)

will be updated. MSFSR is cleared upon entrance to emulation mode. The emulation service processor must check the MIFLTD status bit after every emulation instruction which references memory, and if that bit is set, it is the responsibility of the remote emulator software to clear it with an explicit STA emulation instruction to the MSFSR. This MIFLTD bit is cleared by JTAG TAP Controller reset, updating MCI, or *Viking* reset.

MSTAT.PFPX

PFPX indicates that a pending floating point exception exists. This exception can be caused by two ways:

Prior non-emulation FPOPs (which were already in the FQ and continued to execute while the processor has entered emulation mode) generate an FP exception.

Emulation FPOPs can also generate an FP exception.

> Note:
> Before issuing an FP related emulation instruction, the remote emulator must make sure that: All FPOPs have cleared from the FQ (MSTAT.FQE = 1). There is no pending FP exception generated (PFPX = 0).

Any taken FP exception in emulation mode will cause error mode. This PFPX bit is cleared by clearing out the FQ, by a *Viking* hardware reset, or a JTAG TAP Controller reset.

MSTAT.MIDONE

MIDONE is asserted when an emulation instruction completes execution. No additional emulation instructions should be issued until the current emulation instruction has completed, as indicated by MIDONE. If an emulation instruction is an FPOP, MIDONE assertion only means that the FPOP was successfully issued to the FPU. Therefore, for floating point related emulation instructions, the requirements to satisfy before issuing another FP related emulation instruction is error-free execution of the previous FPOP which is indicated when FQE = 1 and PFPX = 0. This MIDONE bit is cleared by JTAG TAP Controller reset, an MSTAT register read, subsequent entry into emulation mode, or *Viking* hardware reset.

MSTAT.FQE

FQE indicates that the FQ is empty, and is asserted when all FPOPs have finished error-free execution.

6.3.2. Emulation Registers in ASI Space

There following 4 ASI mapped registers directly support emulation: MPC/MNPC, MTMP[1-2], MDIN, and MDOUT. The following sections briefly describe each of those registers. Other ASI mapped registers are also accessible through SPARC ASI instructions.

6.3.2.1 Emulation exit PC/NPC Registers

ASI	Function	Access	Size	Section
0x47	Emulation Exit PC	LD/ST	single	4.15
0x48	Emulation Exit NPC	LD/ST	single	4.15

The program counter of the instruction executing just prior to emulation entry is written to PC/NPC registers. These values are the targets of the branch when *Viking* resumes normal execution upon exit from emulation. These registers can be examined or altered by the emulator, and are accessible through LDA/STA 0x47-0x48 (refer to section 4.15.4 — *Emulation Program Counters*).

6.3.2.2 MTMP[1-2] Registers

ASI	Function	Access	Size	Section
0x40-0x41	Emulation Temps[1-2]	LD/ST	single	4.15

These two registers are useful for temporarily storing information while in emulation. If more than two words of information are to be temporarily stored, the remote emulator must use its JTAG MDOUT scan capability, and later restore it through JTAG MDIN. These MTMP[1-2] registers are accessible through ASI 0x40-0x41 (refer to 4.15.1, — *Emulation Temporary Registers (MTMP[1-2])*).

6.3.2.3 MDIN Register

ASI	Function	Access	Size	Section
0x44	Emulation Data In1	LD	single	4.15

An emulation instruction sequence can move incoming emulation data (address pointers and data for *Viking* system state updates) from the JTAG MDIN register into the IU register file using LDA instructions. Subsequent emulation instructions can use this data to update memory or *Viking* state. This MDIN is a *read only* register through an ASI 0x44 access.

6.3.2.4 MDOUT Register

ASI	Function	Access	Size	Section
0x46	Emulation Data Out	LD/ST	single	4.15

An emulation instruction sequence can move outgoing *Viking* state data from the IU register file to the emulation MDOUT port using a STA. Emulation control software can then use this information to display processor state or check status. This register is accessible through ASI 0x46.

6.4. Supported Emulation Primitives

The table below identifies the emulation primitives or functions that *Viking* supports.

Emulation primitive	Description/Method
Force Emulation Mode	Assert MCMD.MENTER
R/W integer register	PSR, WIM, TBR, Y
R/W integer register file	SPARC LD/ST instructions
R/W floating point register	FSR, FQ
R/W floating point register file	SPARC LD/ST %f0-%f31
R/W memory	SPARC LD/ST, LDA/STA to memory
R/W emulation exit PC/NPC	LDA/STA ASI 0x47-0x48
Observe emulation status	Scan out MSTAT
Resume normal execution	Assert MCMD.MEXIT
Hardware Reset *Viking*	Assert MCMD.MRESET
Watchdog Reset *Viking*	In emulation, issue TA
Allow user emulation request	Assert MCMD.INITM, user exec SIGM

Each of the primitive is briefly described:

6.4.1. Force Emulation Mode

The remote emulator can force *Viking* to enter emulation by asserting the MCMD.MENTER bit. Since this bit is on the MCI scan chain, the only way to access it is by JTAG scan methodology.

6.4.2. R/W Integer Registers

While in emulation mode, the following IU registers are completely accessible through the use of normal SPARC instructions (loaded onto MINST). They are: PSR, WIM, TBR, Y.

6.4.3. R/W Integer Register File

While in emulation mode, the IU register file is completely accessible through the use of normal SPARC instructions (loaded onto MINST). It covers all implemented windows.

6.4.4. R/W FP Registers

While in emulation mode, the following FP registers are completely accessible through the use of normal SPARC instructions (loaded onto MINST). They are: FSR, FQ. See "Note on FP related emulation instructions" below.

6.4.5. R/W FP Register File

While in emulation mode, the FP register file is completely accessible through the use of normal SPARC instructions (loaded onto MINST). They are: %f0-%f31. See "Note on FP related emulation instructions" below.

> Note on FP related emulation instructions:
> Before issuing any FP-related emulation instructions, the remote emulator must examine MSTAT.FQE to make sure that all prior FPOPS in the FQ have completed and no pending floating point exception was generated.

6.4.6. Read/Update Memory

While in emulation mode, memory is completely accessible through the use of normal SPARC instructions (loaded onto MINST). All ASI spaces are also accessible.

6.4.7. R/W Emulation Exit PC and NPC

While in emulation mode, allow the remote emulator to examine the emulation exit PC/NPC pair. There is no simple architectural means available to observe the current program counter values, other than CALL and traps. This emulation primitive allows read and write, using LDA/STA 0x47-0x48. Changes in the processors execution stream are accomplished by modifying these PC/NPC register pair.

6.4.8. Observe Emulation Status

While in emulation mode, MSTAT records if the attempted emulation instruction has completed, faulted, caused error mode or has a pending interrupt request. MSTAT also specifies if the FQ is empty or has generated a pending FP exception. The pending interrupt request can be used to suggest to the emulator that normal program flow should be continued, to minimize interrupt latency.

6.4.9. Resume Normal Operation

While in emulation mode, force *Viking* to resume non-emulation execution from the saved (or emulation modified) PC/NPC pair. This is accomplished by asserting MCMD.MEXIT

6.4.10. Hardware Reset *Viking*

Emulation can initiate both a full reset or a watchdog reset. To initiate a full reset, assert MCMD.MRESET.

6.4.11. Watchdog Reset *Viking*

To initiate a watchdog reset, while in emulation, issue a trap always (TA) emulation instruction.

6.4.12. Allow user emulation request

By asserting MCMD.INITM, the remote emulator is allowing a user request to enter emulation, which the user accomplishes by executing the SIGM (a *Viking* specific instruction). SIGM executes as a NOP when MCMD.INITM is not asserted.

6.5. Emulation Sequences

Issuing each emulation instruction requires the emulator to send multiple JTAG scan sequences. Since the MCI TDR contains both the MINST and MCMD registers, only one register needs to be loaded to issue a single emulation instruction. Depending on the emulation instruction to execute, MDIN may need to be set before the instruction is executed. For more complex emulation sequences, processor state will need to be preserved before any state is modified. This involves

many emulation instructions.

Unless *Viking* is in error mode, the remote emulator can force *Viking* into emulation mode by scanning in an asserted MCMD.MENTER value. The emulator polls MSTAT for an indication that the instruction is complete. The scan sequence for emulating an individual instruction is at least a portion of the following:

Scan-In any required pointers and data into MDIN.
Scan-In the emulation instruction (MCI_MINST) and
emulation protocol command (MCMD), including the
MEXEC and optionally the MEXIT bits.
Scan-Out MSTAT emulation status register
to determine when the emulation instruction has completed, faulted, or induced error mode. This poll also indicates whether any prioritized interrupt is currently at *Viking*'s pins.
Scan-Out of any requested *Viking* system state data from MDOUT.

6.6. Emulation Execution Details

This section provides some additional information on how *Viking* operates in emulation mode.

6.6.1. State during Emulation mode

Viking assumes the following state when in emulation mode:

Table 6-5 *Viking State upon entry into Emulation mode*

Register/Bit Affected	State
PSR.S	asserted
PSR.EF	asserted
PSR.ET	negated
ACTION.MIX	negated
MCNTL.NF	asserted
	Synchronous ST

As reflected in the table above, when *Viking* enters emulation mode, all emulation instructions execute in supervisor mode, multiple instruction execution mode is disabled, the NF bit is asserted, and emulation ST instructions are synchronous and bypass the store buffer. A store buffer copy-out was done before entry into emulation. The FPU is still enabled, it is allowed to continue execution, without being aware that the processor has entered emulation.

6.6.2. Faults during Emulation Execution

If an emulation data memory reference instruction faults, it sets the .MSTAT.MIFLTD bit (but does not cause error mode). The remote emulator must force an emulation instruction to clear out the MSFSR register through the MMU ASI space. Failing to do this will leave MSTAT.MIFLTD asserted, and can give subsequent emulation instructions a faulty emulation status indication.

The processor executes with PSR.ET disabled. Any synchronous exceptions that are reported will cause *Viking* to enter error mode (and set MSTAT.ERRMODE) which induces a watchdog reset. The fact that PSR.ET was disabled upon emulation entry allows asynchronous exceptions (such as priority interrupts, data_store_exception, floating_point_exception) to be ignored.

6.6.3. Legal and Illegal Emulation Instructions

The emulation instruction in MINST must be a legal SPARC instruction. However, only a subset of the SPARC instruction set is supported during emulation mode. In general, instructions which affect the flow of execution when not in emulation mode are not supported. A simple example is a branch instruction. There is no reason to support control transfer instructions in emulation mode. Operation of the processor in emulation mode on these illegal instructions is undefined.

Legal emulation instructions:

All legal memory reference instructions (including alternates, atomics).
All arithmetic and logical instructions, except trapping tagged arithmetic.
SETHI.
All integer state register accesses.

Illegal emulation instructions (but legal SPARC instructions):

All control transfer instructions (CALL,BICC,FBFCC,JMPL,RETT).
All software traps (TICC).
The FLUSH instruction.
All trapping tagged arithmetic.
SAVE and RESTORE (manipulate CWP directly instead).
All illegal instructions.

Many of these illegal operations will cause entry into error mode, force *Viking* to leave emulation mode and induce a non-emulation watch dog reset.

6.6.4. Compound Emulation Protocol Commands

Separate control bits are used to enter emulation, execute an emulation instruction, and then exit. These separate bits may be used together to optimize emulation sequences. The simplest emulation sequence writes the MCI.MINST register with a single emulation instruction, and set the MENTER, MEXEC, and MEXIT bits. This will cause *Viking* to enter emulation, execute the emulation instruction, then resume execution. This entire sequence will require only a few cycles to execute (after MCI is scanned in through JTAG).

Table 6-6 *Valid compound emulation sequences*

MENTER	MEXEC	MEXIT	Action
1	1	0	Enter emulation mode, issue a command, then pause (awaiting MEXEC or MEXIT).
x	1	0	Once in emulation mode, issue a new emulation instruction, then pause (awaiting MEXEC or MEXIT).
x	1	1	Once in emulation mode, issue a new emulation instruction, exit emulation mode upon completion, then resume execution at the captured (or altered) non-emulation PC pair.
x	0	1	Once in emulation mode, immediately resume execution at the captured (or altered) non-emulation PC pair.
x	0	0	Once in emulation mode, is a NOP.
1	0	0	Will cause entry into emulation mode, and wait.
1	0	1	Will start entry into emulation, but then immediately exits. No emulation instruction is executed. MSTAT.MACK is not updated. To the programmer it might appear that emulation was never entered. (This sequence is not very useful)

Simultaneous assertion of MEXEC and MEXIT can dangerously lead to timing races in sampling MSTAT.MIDONE that cause bad inferences by the remote emulation service processor. Upon completion of the last emulation instruction in the current emulation session, this compound MCMD mode will exit emulation mode. MSTAT.MIDONE will be set and MSTAT.MACK will be negated.

The remote emulator must check MSTAT to make sure the previous emulation instruction has completed and was error-free. A re-entry into emulation clears out MSTAT.MIDONE and MSTAT.MIFLTD, and it could do so before the remote emulator can check MSTAT. If this corrupted case, the remote emulator would mistakenly assume that the last emulation instruction had not completed.

It is recommended that an MEXEC be issued without MEXIT. The MSTAT can determine the completion status of the last emulation instruction. When MSTAT reports error-free, MEXIT without MEXEC can be issued to exit from emulation. An immediate re-entry into emulation mode will allow the emulation program to differentiate between the end of the first emulation session and the start of the second emulation session.

6.7. Emulation Instruction Sequences for Common Emulator Functions

This section will describe some possible emulation instruction sequences for some common emulator primitives. Many other implementation are possible. The primitives may be combined to create other functions as needed.

The primitives to be described are:

Read/Write Integer Registers
Read/Write Integer Control Registers (PSR WIM TBR Y)
Read/Write Floating Point Registers
Read/Write Floating Point Control Registers
Read/Write Memory (byte,half,word,double,etc.)
Read/Write Memory (Normal and ASI)
Set Code and Data Address Breakpoints
Single Step
Run for N Cycles
Run Until Breakpoint reached

The next sections provide detailed sequences for each of the above operations. Throughout these sequences, numerous symbolic constants will be used.

The *symbolic constants* are described below:

Table 6-7 *Symbolic Constants for Emulation Sequences*

mdiag	0x38
bkv	0x000
bkm	0x100
bkc	0x200
bks	0x300
mtmp1	0x40
mtmp2	0x41
mdin	0x44
mdout	0x46
mpc	0x47
mnpc	0x48
ctrv	0x49
ctrc	0x4a
ctrs	0x4b
action	0x4c
XREG	emulation register
XADDR	emulation memory addr

6.7.1. Integer Register File Read

The most basic emulator operation is to display an integer register. The following emulation instruction sequence will transfer the contents of the integer register XREG within the current CWP to MDOUT register. Once in MDOUT, the data can be scanned out to the emulator.

```
// copy XREG to mdout; scan it out.
scan_in("sta %XREG, [%g0] mdout", mci, poll)
scan_out(mdout, remote_emulator)
```

6.7.2. Integer Register File Write

To store a new value (or restore an old value) into an integer register at the current CWP, the following sequence can be used. The new value for integer register XREG is assumed to have been previously scanned-in to register MDIN.

```
// scan in new value; install it into iu rfile.
scan_in(new_iu_rfile_entry_value,mdin)
scan_in("lda       [%g0] mdin, %XREG", mci, poll)
```

6.7.3. Integer State Register Read

The following sequence will transfer any of the integer state registers (PSR WIM TBR Y) registers into the MDOUT register. The sequence below assumes access to the PSR is desired. Note the use of the MTMP1 register to preserve and restore integer register state.

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)

// copy iu_creg to mdout through g1 temp. scan it out.
scan_in("rd  %psr, %g1", mci, poll)
scan_in("sta %g1, [%g0] mdout", mci, poll)
scan_out(mdout, remote_emulator)

// epilogue
scan_in("lda       [%g0] mtmp1, %g1", mci, poll)
```

6.7.4. Integer State Register Write

The following sequence will modify any integer state register. The new value for the given IU state register is assumed to have previously been scanned-in to MDIN. The sequence below modifies the PSR.

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)

// scan in new value to g1 and install into psr.
scan_in(new_iucreg_value,mdin)
scan_in("lda      [%g0] mdin, %g1", mci, poll)
scan_in("wr %g1, %psr", mci, poll)

//epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll
```

6.7.5. Memory Read

The sequence below demonstrates reading a signed byte value at a given memory address *XADDR* within an implicit alternate address space (supervisor data space where ASI=0xb). This ASI is used since the processor is *effectively* executing in supervisor mode. The value of *XADDR* is assumed to reside in the MDIN I/O register before any emulation instruction is emitted.

At the conclusion of the sequence, the value of the signed byte residing in memory address XADDR within the implicit supervisor data space ASI (0xb) will be placed in the MDOUT register. When MIDONE is asserted, the data can be scanned-out to the emulator.

The sequences for signed and unsigned half-word, word and double word reads are very similar. Reads from alternate address spaces are also similar, the LDSB instruction is replaced with a LDSBA.

```
//prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll);

// scan in xaddr. copy to g1
scan_in(new_xaddr_value,mdin)
scan_in("lda      [%g0] mdin, %g1", mci, poll)

// load g2 w/ *xaddr; copy to mdout. scan it out.
scan_in("ldsb     [%g1], %g2", mci, poll)
scan_in("sta %g2, [%g0] mdout", mci, poll)
scan_out(mdout, remote_emulator)

//epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll)
scan_in("lda      [%g0] mtmp2, %g2", mci, poll);
```

6.7.6. Write Memory

The following sequence modifies memory by writing a byte value at a given memory address XADDR within an implicit alternate address space (supervisor data space where ASI=0xb).

During a memory write, MDIN will be used twice by the emulation software. Initially the remote emulation software will scan into MDIN the value of the memory address XADDR to be written. Once this address is transferred to the IU register file, the remote emulation software will then scan in the new value (NEW_DATA_VALUE) to be written at the specified memory location (XADDR).

The sequences for half-word, word and double word writes to are very similar. Writes to alternate spaces are also similar, with the STB instruction replaced by a STBA.

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll);

// scan in XADDR value. copy XADDR to g1.
scan_in(XADDR_VALUE, mdin)
scan_in("lda      [%g0] mdin, %g1",mci, poll)

// scan in NEW_DATA value.  copy NEW_DATA to g2.
scan_in(NEW_DATA_VALUE, mdin)
scan_in("lda      [%g1] mdin, %g2",mci, poll)

// install NEW_DATA at XADDR.
scan_in("stb %g2, [%g1]",mci, poll)

// epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll)
scan_in("lda      [%g0] mtmp2, %g2", mci, poll)
```

6.7.7. Floating Point Register Read

For simplicity, this example provides the save address in register MDIN, its selection is assumed to be safe. The following sequence will transfer a floating point register XREG into the MDOUT register for the emulator to scan out. The sequence for reading floating point control registers is similar.

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)

// scan address of background memory word (XADDR).
// copy XADDR into %g1; load *xaddr into g2.
scan_in(xaddr_value, mdin);
scan_in("lda      [%g0] mdin, %g1",mci, poll)
scan_in("ld [%g1], %g2", mci, poll)

// this emulation primitive MUST not change non-emulation
// memory state. FP reads require we alter and then
// restore a background memory state. No sufficient number
// of mtmp to do this so we augment the storage by scanning
// out to external remote emulation processor memory.
scan_in("sta %g2, mdout",mci, poll)
scan_out(mdout, remote_emulator_temp1)

// before issuing next emulation instruction,
// make sure mstat.fqe=1 and mstat.pfpx=0
// copy fp entry to bgnd word; copy into iu temp.
scan_in("st  %fXREG, [%g1]",mci, poll)
scan_in("ld [%g1], %g2",mci, poll)

// copy FP entry to mdout; scan out fp entry.
scan_in("sta %g2, [%g0] mdout",mci, poll)
scan_out(mdout, remote_emulator)

// restore original background memory word
scan_in(remote_emulator_temp1,mdin)
scan_in("lda      [%g0] mdin, %g2", mci, poll)
scan_in("st  %g2, [%g1]",mci, poll)

// epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll)
scan_in("lda      [%g0] mtmp2, %g2", mci, poll)
```

6.7.8. Floating Point Register Write

The sequence for writing floating point registers is similar to the reading sequence, except a floating point register is loaded from memory, rather than written. The following is a possible code sequence:

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)

// scan address of background memory word (XADDR).
// copy XADDR into %g1; load *xaddr into g2.
scan_in(XADDR_VALUE, mdin)
scan_in("lda      [%g0] mdin, %g1",mci, poll)
scan_in("ld [%g1], %g2", mci, poll)

// preserve original background word in remote_emulator_temp1.
// this emulation primitive must not change non-emulation
// memory state. FP writes require we alter and then restore
// a background memory state. No sufficient number of mtmp
// to do this so we augment the storage by scanning out to
// external remote emulation processor memory.
scan_in("sta %g2, mdout",mci, poll)
scan_out(mdout, remote_emulator_temp1)

// scan in new new fp_data_value
// load it into iu rfile. write it to background word.
scan_in(NEW_FP_DATA_VALUE,mdin)
scan_in("lda      [%g0] mdin, %g2",mci, poll)
scan_in("st  %g2, [%g1]",mci, poll)

// before issuing next emulation instruction,
// make sure mstat.fqe=1 and mstat.pfpx=0
// install new value into FP register.
scan_in("ld [%g1], %fXREG",mci, poll)

// restore original background memory word.
scan_in(remote_emulation_temp1,mdin)
scan_in("lda      [%g0] mdin, %g2", mci, poll)
scan_in("st  %g2, [%g1]",mci, poll)

// epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll)
scan_in("lda      [%g0] mtmp2, %g2", mci, poll)
```

6.7.9. Floating Point State Register Read

Reading values from floating point state registers, such as the FSR is similar to the previous two examples. The floating point memory references are replaced by store FSR operations. Extracting entries from the floating point queue is more difficult. While in emulation mode, FQ entries should *never* be extracted unless MSTAT.PFPX indicates an FP exception. Otherwise the remaining FPOPS in the FQ should be allowed to execute. If PFPX is asserted and the emulator wants to recover, the FQ can be read with a double word size. Since MDOUT is only a

single word wide, two scan passes are required to transfer the full queue entry to the emulator.

The following sequence will read the floating point queue.

There is an additional *architectural* side affect of reading the floating point queue. When an entry is read, it is effectively removed from the queue. Since emulation is required to be non-intrusive to program execution, the old state of the queue must be restored. If the queue state is to remain unchanged after being observed, then all entries in the queue must be extracted using STDFQ until it is empty. This must be followed by re-inserting (re-writing) all of these removed entries back into the queue. However, there exist no simple instruction to allow writing to the FP queue. It may however be restored by executing a floating point instruction while in emulation mode. The instruction and PC value are both stored in the queue. By writing the PC into the MDIN register, and then issuing the floating point instruction as an emulation instruction, the queue will be restored.

6.7.10. Floating Point State Register Write

Writing to the floating point state registers is similar to reading them. The same restrictions apply.

6.7.11. Setting Code and Data Address Breakpoints

The sequence below will set up a code or data address breakpoint and resume normal execution. When (if) the breakpoint occurs, the processor will re-enter emulation mode.

A string of emulation instruction sequences will be required to write the code address breakpoint register set.

Write the desired code or data space breakpoint address value (virtual or physical) into MDIAG_BKV.
Write the code or data space breakpoint address compare mask into MDIAG_BKM.
Write the breakpoint control register, MDIAG_BKC, to clear {C,D}BKFEN, and select the desired value for CSPACE, PAMD, CBKEN, DBREN and DBWEN,
Write the breakpoint status register, EDIAG_{C,D}BKS to clear any prior code breakpoint status.
Write the action on event control register, EDIAG_ACTION, so that the desired breakpoint event will generate an emulation request and (optionally) assert an emulation strobe (ESB pin).

An example for such an emulation sequence code is given below:

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)
scan_in("sta %g3, [%g0] mdout", mci, poll)
scan_out(mdout, remote_emulator_temp1)

// set g1 to ASI cbkv addr offset w/in MDIAG ASI;
scan_in("or %g0, bkv, %g1",mci, poll)

// scan-in lower 32-bit (of 36 bits) for next cbkv value
// install it in g2 of register pair g[23].
scan_in(NEW_CBKV_VALUE_LO32,mdin)
scan_in("lda      [%g0] mdin, %g2",mci, poll)

// scan-in upper 4-bit portion (of 36-bits) for next cbkv value
// install it in g3 of register pair g[23].
scan_in(NEW_CBKV_VALUE_HI4,mdin)
scan_in("lda      [%g0] mdin, %g3",mci, poll)

// install register pair g[23] into mdiag cbkv.
scan_in("stda     %g2, [%g1] mdiag",mci, poll)

// set g1 to ASI cbkm addr offset w/in MDIAG ASI;
scan_in("or %g0, bkm, %g1",mci, poll)

// scan-in lower 32-bit portion (of 36-bits) for next cbkm value
// install it in g2 of register pair g[23].
scan_in(NEW_CBKM_VALUE_LO32,mdin)
scan_in("lda      [%g0] mdin, %g2",mci, poll)

// scan-in upper 4-bit portion (of 36-bits) for next cbkm value
// install it in g3 of register pair g[23].
scan_in(NEW_CBKM_VALUE_HI4,mdin)
scan_in("lda      [%g0] mdin, %g3",mci, poll)

// install g[23] to cbkm
scan_in("stda     %g2, [%g1] mdiag",mci, poll)

// set g1 to addr cbkc;
scan_in("or %g0, bkc, %g1",mci, poll)
```

(Code sequence is continued on next page)

```
// scan-in NEW_CBKC_VALUE.
// read it into the iu rfile; install it into CBKC.
scan_in(NEW_CBKC_VALUE, mdin)
scan_in("lda      [%g0] mdin, %g2",mci, poll)
scan_in("sta%g2, [%g1] mdiag",mci, poll)

// set g1 to addr cbks in mdiag; clear cbks.
scan_in("or %g0, bks, %g1",mci, poll)
scan_in("sta%g0, [%g1] mdiag",mci, poll)

// scan-in NEW_ACTION_VALUE.
scan_in(NEW_ACTION_VALUE, mdin);
// read it into the iu rfile; install it into ACTION.
scan_in("lda      [%g0] mdin, %g2",mci, poll)
scan_in("sta%g2, [%g0] action",mci, poll)

// epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll)
scan_in("lda      [%g0] mtmp2, %g2", mci, poll)
scan_in(remote_emulator_temp1, mdin)
scan_in("lda      [%g0] mdin, %g3", mci, poll)
```

Setting a breakpoint on a specific data memory address reference is very similar to the above sequence. The only difference is the value written into memory mapped BKC and ACTION asi registers. In the above example, CSPACE, CBKEN and IEN_CBK are set. A write-only data address breakpoint would clear CSPACE, DBFEN, DBREN and IEN_DBK while DBWEN would be set.

6.7.12. Run for "N" Instructions/Cycles

Sequences for programming "Run-for-N" instructions and cycles are similar to setting code/data breakpoints. The cycle counter breakpoint is most useful for statistically profiling execution of a program. The instruction counter breakpoint is useful for single stepping, or block stepping through a program execution.

```
// prologue
scan_in("sta %g1, [%g0] mtmp1", mci, poll)
scan_in("sta %g2, [%g0] mtmp2", mci, poll)
scan_in("sta %g3, [%g0] mdout", mci, poll)
scan_out(mdout, remote_emulator_temp1)

// set g1 to ASI address for CNTV w/in EDIAG.
scan_in("or %g0, 0x000, %g1",mci, poll)

// scan-in NEW_CNTV_VALUE for ICNT/CCNT.
// read it into %g2; install into CNTV.
scan_in(NEW_CNTV_VALUE, mdin);
scan_in("lda      [%g0] mdin, %g2",mci, poll)
scan_in("sta %g2, [%g1] cntv",mci, poll)

// clear cnts
scan_in("sta %g0, [%g0] cnts",mci, poll)

// scan-in NEW_ACTION_VALUE for ACTION.
// read it into %g2; install it into ACTION.
scan_in(NEW_ACTION_VALUE, mdin);
scan_in("lda      [%g0] mdin, %g2",mci, poll)
scan_in("sta %g2, [%g1] action",mci, poll)

// scan-in NEW_CNTC_VALUE for ICNTEN/CCNTEN.
// read it into %g2; install it into CNTC.
scan_in(NEW_CNTC_VALUE, mdin)
scan_in("lda      [%g0] mdin, %g2",mci, poll)
scan_in("sta %g2, [%g1] cntc",mci, poll)

// epilogue
scan_in("lda      [%g0] mtmp1, %g1", mci, poll)
scan_in("lda      [%g0] mtmp2, %g2", mci, poll)
scan_in(remote_emulator_temp1, mdin)
scan_in("lda      [%g0] mdin, %g3", mci, poll)
```

6.8. Approximate Latencies for Each Emulator Primitive

Each emulation instruction execution requires several multi-bit JTAG TDR scan operations.

The number of emulation instructions per emulator primitive is:

1 to access an IU register file entry
4 to access an IU control register
7 to access a memory location
13 to access a floating point register
13 to set up an instruction(cycle) counter expiration
21 to set up an instruction (data) address breakpoint

The number of MDIN (or MDOUT) scan operations per emulator primitive is:

1 scan operation per IU register file accesses
1 MDIN scan operation per FP register file write
1 MDIN and 1 MDOUT scan operation per FP register file read
2 (MDIN or MDOUT) scan operations per memory access
3 MDIN scan operations per setting of a breakpoint or counter event

Each emulation instruction scan in and emulation status scan out takes about 50 TCK cycles (about 500 native VCK cycles). Each MDIN scan in takes about 40 TCK cycles. Each MDOUT scan out takes about 40 TCK cycles.

6.9. Details about Entering Emulation

Emulation mode can be entered through any one of six different ways:

1. Set MCMD.MENTER to force emulation.
2. Execute SIGM, with MCMD.INTM set.
3. Through Code Address Breakpoint.
4. Through Data Address Breakpoint.
5. Through Instruction Counter Breakpoint.
6. Through Cycle Counter Breakpoint.

The first is used by the remote emulation processor to unconditionally present an emulation mode request by using the JTAG tap controller to assert MCMD.MENTER. The other ways require ASI registers to be configured so they conditionally generate an emulation request. SIGM (a special *Viking* instruction) also allows emulation entry, see 4.4.6 for more details. For further description on how to set up the breakpoints, see table 4-16.

6.10. Emulation Exception Issues

During emulation, PSR.ET is negated and MMU.NF is asserted. Negating PSR.ET disables any asynchronous exception from being honored by the *Viking* IU while in emulation mode. This includes priority interrupts, store buffer exceptions, and floating point exceptions. Any synchronous non data_access_exception will cause the *Viking* IU processor to enter watch dog reset.

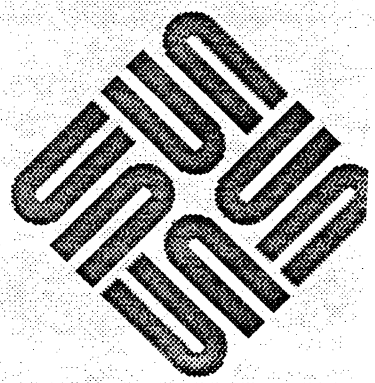
IPND informs the emulator that a priority interrupt is being deferred while in emulation mode. MIFLTD informs the emulator that a store buffer (or emulation memory reference) exception is being deferred while in emulation mode and the MFSR register will describe the nature of the fault in more detail. MSTAT.PFPX informs the emulator that a floating point exception is being deferred while in emulation mode. When any of these events occurs, the action to be taken is emulator dependent. The remote emulation processor should avoid issuing an emulation instruction that might cause control to be transferred to an exception handler.

Floating point exceptions during emulation mode are of particular concern, requiring elaborate trap handlers. Care must be taken to examine all the signals indicating error-free condition before issuing an FP related emulation instruction (refer to section 6.3.1.5 for more details.

[Blank Page]

7

System Interface



[Blank Page]

7

System Interface

Viking supports a variety of different system environments. This flexibility allows system designers to make proper cost and performance tradeoffs.

This chapter will briefly introduce *Viking's* two bus interfaces. Further details on each interface will be presented in later chapters. In addition, functions which are *common* to both interfaces (clock, reset, interrupts, test) will be described in this chapter.

7.1. Multiple System Interfaces

Viking supports two primary bus interfaces. The SPARC MBUS standard is provided for direct connection to MBUS systems. A more optimized interface, the *Viking Bus* is provided for higher performance systems and other more dedicated applications, including the use of an external second level cache.

7.1.1. Lower Cost Systems → MBUS

For *lower cost* systems, *Viking* provides a standard SPARC MBUS interface. The MBUS is described in detail in the SPARC MBUS specification. *Viking's* implementation of the MBUS will be described in chapter 8. MBUS is a 64-bit multiplexed cache consistent bus. Using the MBUS interface, *Viking's* data cache operates in a buffered *copy-back* mode, with *write miss allocation*.

7.1.2. Higher Performance Systems → Viking Bus

For higher performance, or other non-MBUS systems, *Viking* provides its own bus interface, called the *Viking bus*. The *Viking bus* is a non-multiplexed, highly pipelined bus which has been optimized to interface with external second level cache implementations. The bus is very flexible, allowing *Viking* to be used in a variety of system environments. Using the *Viking bus*, the data cache operates as a *write-through* cache. Cache consistency is also maintained. *Viking Bus* operation is fully described in chapter 9. Use of this bus interface places the processor into *CC mode*, as described in earlier chapters.

7.1.3. Selecting Bus Modes

The choice of bus interface must be made statically in the system. *Viking* configures itself for operation in a particular mode based on the CCRDY_ pin. This pin must be valid when RESET_ is deasserted, and must *never* be changed during normal system operation. System software may view the state of this pin in the MCNTL.MB status bit. If the bit is a one, *Viking* is in MBUS mode, otherwise *Viking* bus is selected.

7.2. Clock Operation

This section will describe *Viking's* clock requirements. Proper clocking is essential at high operating frequencies. In order to reduce system clock skew, a phase locked loop (PLL) is implemented on chip. For testing and other purposes, a PLL bypass mechanism is provided. When the PLLBYP_ signal is active (low), the PLL circuitry will be bypassed completely.

7.2.1. Phase Locked Loop Operation

The PLL operates by constantly measuring internal clock routing delay and internally generating a clock which is effectively *ahead* of the external clock by an amount equal to that internal routing delay. This ensures that all internal logic sees a clock signal nearly equivalent to the external clock pin. All system logic using the same clock as the processor is expected to provide acceptable setup and hold times relative to the processor clock input pin.

> Important Note:
>
> Prior to normal operation, the PLL must be allowed time to stabilize. During this time, RESET_ should be active. The time required is 100ms (milliseconds).
>
> The input clock to *Viking* must never be stopped or changed from its normal periodic operation while the PLL is enabled. Doing so will cause PLL instability and unpredictable operation.

While the PLL does improve system performance, it must also be used carefully. As noted above, the external clock input must be stable at all times. The PLL may take long periods of time to stabilize initially. During this initialization time, the RESET_ input must be asserted, as the internal clock may be unstable.

> Important Note:
>
> It is essential that the JTAG TAP controller be reset prior to, or at the same time as RESET_ in order for the PLL to begin initialization. The TAP controller may be initialized either by asserting the TRST_ pin, or by asserting the TMS pin for five consecutive cycles of TCK (Test Clock). If this reset does not occur, the PLL clock feedback loop may not be established, and unpredictable operation may result.
>
> Whenever the JTAG interface is not in use by a particular system, asserting the TRST_ signal statically is strongly recommended.

7.2.2. Input Clock Requirements

Other than providing a clean, stable clock, *Viking* can tolerate most clock sources when the PLL is enabled.

With the PLL enabled, *Viking* uses only the *rising edge* of the incoming clock. Internally, *Viking* multiplies, then divides the clock to provide a stable 50% duty cycle clock. Input duty cycle must be at least 25% (either high or low).

When the PLL is bypassed, care must be taken to provide a 50% duty cycle clock. Pin timings for operation with the PLL bypassed are not fully defined.

> Important Note:
> Operation in a system with the PLL bypassed is not recommended or fully specified. Use in this manner will generally require reduced operating frequencies and very careful system design.

7.3. Reset Operation

Proper reset of *Viking* is critical. Improper use of reset will result in unpredictable operation. From a software perspective, two reset operations are defined: *Hardware Reset* and *Watchdog Reset*. Since watchdog reset is strictly a function of software errors, it will not be discussed here. This section will describe hardware reset requirements.

7.3.1. JTAG and Hardware Reset

In addition to hardware reset, another reset must be considered for proper system operation. *JTAG TAP Reset* must be asserted during initial power-on reset. This may be accomplished either by asserting the TRST_ signal, or holding TMS active for five cycles of TCK.

7.3.2. Pin States During Reset

As soon as RESET_ is asserted, *Viking* will tri-state all signals immediately (except for TDO and ESB). All external logic should monitor RESET_ to ensure the validity of control signals.

7.3.3. Reset Timing

Once the PLL has initialized, RESET_ must be asserted for an additional sixteen cycles. If further reset operations are required beyond the initial power-up reset, RESET_ need only be asserted for a total of eight cycles. Since PLL synchronization time is not exact, eight cycles beyond that time is not a critical specification.

7.3.4. Response to Reset (RAM Redundancy)

Viking implements internal RAM redundancy to increase component yield. A portion of this redundancy must be initialized each time the component is reset (hardware reset only). This initialization takes approximately 340 cycles, and is internally timed. These cycles begin once the RESET_ signal is *deasserted*, and before the processor tries to fetch its first instruction. During this time the bus will be inactive, *Viking* will tri-state all I/O signals. All bus requests will be inactive. *Viking* will execute its first bus cycle 345 cycles after RESET_ is deasserted.

7.3.5. Execution Following Reset

Immediately after hardware reset and redundancy repair are complete, *Viking* will execute a *reset trap*. This trap will cause the processor to enter boot mode (MCNTL.BT is set) and begin execution at virtual address 0x00000000. This will force a *READ-SINGLE* bus operation at physical address 0xff0000000. The upper eight bits are set as a result of boot mode (See section 4.3 — Reset Operation for a detailed description of boot mode and processor state at reset.

In response to the read single operation, system logic should supply two valid SPARC instructions on the 64-bit data bus (in accordance with either MBUS,or *Viking* bus protocols). Once these instructions have entered the pipeline, another read single request for the next two instructions will appear on the bus. The physical addresses requested by these reads will be contiguous until a control transfer instruction is executed (normally within two or three instructions).

7.4. Interrupts

Viking provides four external interrupt request signals, IRL[3:0]. External interrupt requests are given to the processor by setting the IRL[3:0] pins to the proper interrupt level.

The externally applied level corresponds to the same SPARC interrupt levels. A value of zero (IRL[3:0]=0000) on the pins indicates no interrupt is present. A value of 0xf (IRL[3:0]=1111) is considered a ***non-maskable*** interrupt. All interrupts are disabled (even non-maskable interrupt) if the PSR.ET (enable traps) bit is not set.

Interrupt requests are *level sensitive*. System logic is expected to maintain all interrupt status, prioritize all external requests, and apply proper interrupt levels. Once an interrupt occurs, system software should manipulate any system control registers necessary to satisfy the interrupt request.

Interrupt requests are synchronized for three cycles after being applied to the IRL[3:0] pins. A debouncing circuit compares the values of the interrupt requests after two and three cycles. If the values are the same, that level interrupt will be presented for comparison with PSR.PIL. If the incoming value is greater, or equal to 0xf, the interrupt request will be presented to the pipeline (provided PSR.ET is asserted). If the values in these two stages of synchronization are different, no interrupt request will be made.

The synchronization mechanism above allows interrupt requests to be asynchronous. Due to the synchronization time, pending interrupts will remain active for several cycles internally after they are removed externally. System software should ensure that enough time is allowed between clearing external interrupt requests and re-enabling interrupts to the processor.

7.5. Memory Model Support (PEND_)

The two standard memory models, TSO, and PSO (see section 4.5 — Memory Model), are supported under system control using the PEND_ input pin. This pin may be used in both MBUS and CC modes. It's use is generally not required, or recommended in MBUS mode however. Selection of TSO or PSO mode affects MBUS operation *only* if PEND_ is not always asserted. (In normal operation it should be tied to VCC).

Viking will sample this signal prior to the start of all bus transactions which require permission to perform a store transaction on the bus. If the PEND_ signal is asserted, the access will not begin. PEND_ must be asserted along with the ready response of the previous store to guarantee that it is seen before the next store. The transactions which require this check depend on the memory model selected. The table below defines the significance of PEND_.

Table 7-1 *PEND_ operation*

	Viking Bus		MBUS	
Transaction Type	PSO	TSO	PSO	TSO
Write-through Store	Only if prior STBAR	Yes	N/A	N/A
Non-cacheable Store	Only if prior STBAR	Yes	Only if prior STBAR	Yes
Swaps	Only if prior STBAR	Yes	N/A	N/A
Non-cacheable Swaps	Only if prior STBAR	Yes	Only if prior STBAR	Yes
Control Space (CSA_	Yes	Yes	N/A	N/A
Demaps	Yes	Yes	N/A	N/A
Internal ASI's	Yes	Yes	Yes	Yes
Copyback writes	N/A	N/A	Only if prior STBAR	Yes

Note that internal ASI operations (both loads and stores) will wait for deassertion of the PEND_ signal. There is one exception to this, ASI 0x4c, the "Action on event" register, which does not depend on PEND_.

7.6. Test Support

Viking provides considerable support for system test. This support includes JTAG, and several dedicated test pins.

7.6.1. JTAG

JTAG boundary scan is provided to allow for system continuity testing. See section 5 — JTAG Serial Scan Interface, and the IEEE P1149.1 specification for a full description of this interface.

The JTAG interface is also used to gain access to additional functional test facilities. A BIST (built in self test) mechanism is available that can be used either from the JTAG interface, or from software. Extensive debugging and emulation facilities are also available over JTAG. (See sections 6 — Remote Emulation Support, and 4.14 — Software Debugging Facilities).

7.6.2. Dedicated Test Functions

Several device pins are provided to improve external visibility of processor operation. The PIPE[9:0] pins provide information about the current state of the processor pipeline. These signals are defined as an aid to system hardware and software debug.

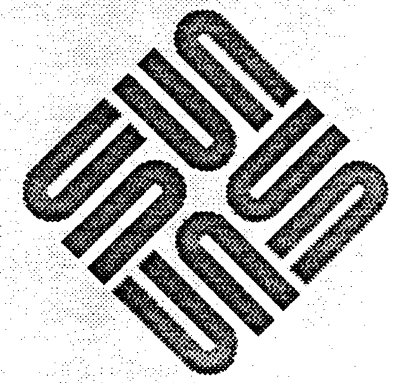
A single *strobe* pin is available, called ESB. This pin operates under software control to provide a programmable external synchronization pulse. It may be triggered by code, data, or cycle count breakpoint detection (see section 4.14 — Software Debugging Facilities). A typical application of this signal is to trigger logic analysis equipment.

Viking provides a simple mechanism to disable all outputs. The TEST_ pin will immediately tri-state all output drivers when asserted. This allows external test equipment to control *Viking* signals. Normally, JTAG boundary scan will be used to accomplish this function. The signal is provided primarily for non-JTAG environments. Processor state is not guaranteed after assertion of TEST_. Note that the TDO and ESB pins are not tristated in response to TEST_ assertion.

[Blank Page]

8

MBUS Interface



8

MBUS Interface

This section describes *Viking's* implementation of the SPARC MBUS standard. *Viking* is fully compliant with the SPARC MBUS standard.

Viking implements *level-2* MBUS. This allows operation in a multiple processor environment. Cache operation and consistency policies are defined in the MBUS specification.

This section will not describe basic MBUS operations. For this information, refer to the MBUS specification. Since the MBUS allows for many variances in detailed processor operation, details of *Viking's* MBUS operation are described.

8.1. Compatibility

Viking may be used in conjunction with some other MBUS processor modules, but not all. In particular, modules which require the *virtual address superset* bits will not be compatible.

Viking responds properly to all standard MBUS transactions, though it does not produce all of them. Only 32-byte burst transactions are supported. All cache consistent transactions operate on 32-byte blocks.

All responses on MSH_ and MIH_ are provided at time "A+3" (the third cycle after assertion of address on the bus). *Viking* can operate in a system with variable response times, up to "A+7". MBUS systems should be designed to accept these timing variations.

Even though *Viking* may be protocol compatible with some modules, electrical and signal timing differences may make certain modules incompatible. See 11.2.3 — MBUS pin timing, or the *Viking* data sheet for pin timing details.

8.2. Selecting MBUS Mode

MBUS operation is selected by driving the CCRDY_ pin inactive. This signal should be driven statically by the system. Any change in the state of this signal after the deassertion of RESET_ will result in unpredictable operation. Operation in MBUS mode may be determined by software reading the MCNTL.MB bit.

8.3. Module ID

The current module number is determined at reset by the processor. The MID[3:0] signals are connected to *Viking's* address bus, pins ADDR[3:0]. The module number should be asserted statically by system hardware.

The number may be any value except 0000, which references the reserved boot mode address space.

8.4. Cache Policy

The level-2 MBUS requires *copy-back* cache operation. Caches must also *write-allocate* data for write misses. This is due to a lack of any *write broadcast* (cached data update) operation on the bus. All cacheable writes are done to the local cache, so the line must be allocated (read) first for write misses.

Cacheability for instruction and data references is described in tables 4-6 and 4-7 respectively.

8.5. Level-2 Consistency Operation

The level-2 MBUS specifies a single ownership, write invalidate cache consistency policy. Only a single cache may *own* a modified copy of a cache line at any one time. Multiple *shared* copies of modified or clean data may exist within the system at any time. These multiple copies are *invalidated* any time the cache line is modified (the modified data may then be re-read from the owner and cached as modified & shared information). This protocol is described int he MBUS specification.

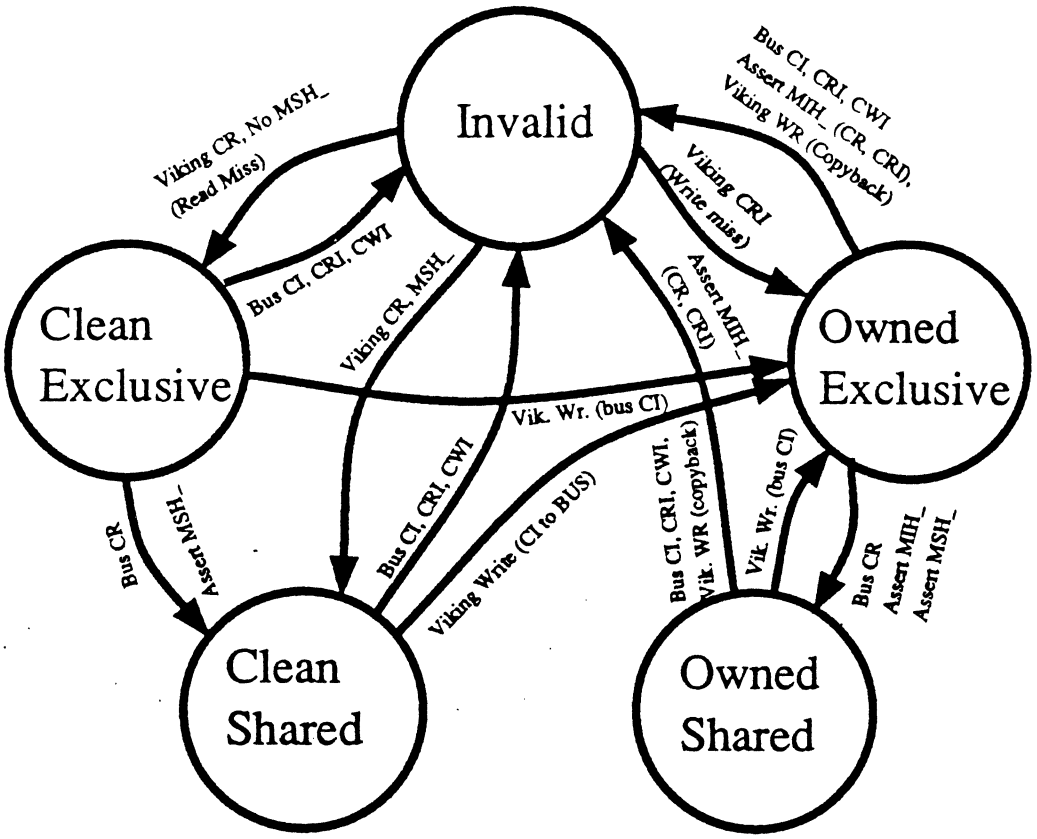
During any coherent read transaction, all processors in the system will assert the MSH_ (Shared) signal if they currently have a cached copy of that data. Since several caches may be sharing the information, the MSH_ signal is open collector, and may be asserted by all caches simultaneously. MSH_ must be pulled inactive external to the processor (typically a resistive pull-up). MSH_ may be asserted at any time prior to the first data acknowledgment for the transaction. MSH_ must be asserted for at least one cycle.

Any cache may *intervene* in a read transaction by asserting the MIH_ (memory inhibit) signal. Since there may be only one owner at a time for each cache line, only a single cache may assert MIH_. *Viking* asserts MSH_ and MIH_ signals in cycle "A+3", three cycles after the address phase of each MBUS transaction.

Once *Viking* asserts MIH_ in response to a read (CR or CRI) transaction, it will complete the bus cycle by supplying data and a ready reply starting four cycles later. During these four cycles, the memory controller can drive other data and ready signals in response to the transaction for two cycles, followed by a third cycle where the bus is driven high. Any replies received during this time are ignored by *Viking*. No exceptions may be reported after asserting MIH_.

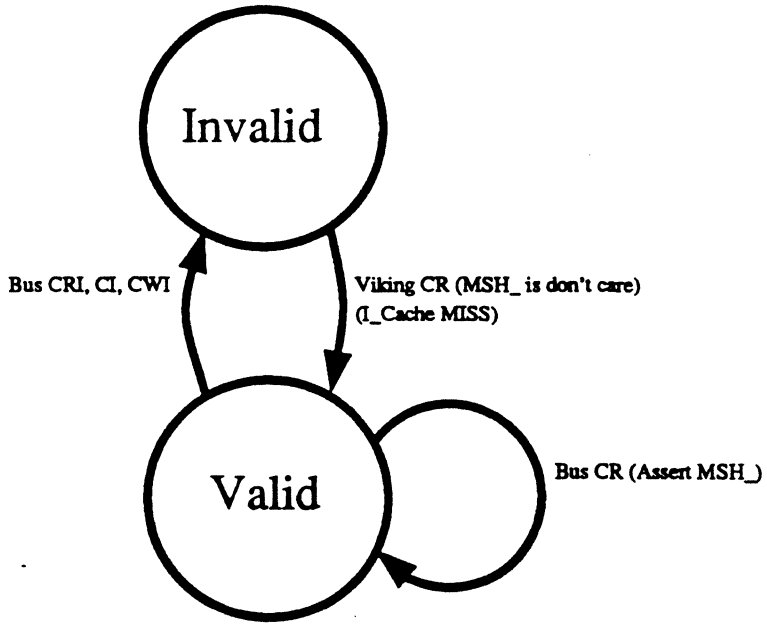
The diagram below represents the cache consistency algorithm used by *Viking's* data cache on the MBUS.

Figure 8-1 *Cache consistency algorithm: Data cache on the MBUS*



The diagram below represents a simpler consistency algorithm used by *Viking's* instruction cache on the MBUS.

Figure 8-2 *Cache consistency algorithm: Instruction cache on the MBUS*



8.6. MBUS Transactions

Viking will produce all valid MBUS transactions, except for CWI (Coherent Write and Invalidate). *Viking* accept all transactions (including CWI) from other system components.

The transaction type used by *Viking* depends on the processor state (including the instruction, and MMU state), as well as the current state of the cache line within the processor.

8.6.1. Non-Cacheable Operations

For I/O transactions, or during boot up, *non-cacheable* transactions will be used. No snooping is done for these references. The standard MBUS level-1 transactions are used for reading and writing. When the store buffer is enabled (MCNTL.SB) non-cacheable stores will be queued into *Viking's* store buffer, the processor will not wait for completion. All non-cacheable reads will cause the buffered stores to copyout before the read is begun.

Cacheability of transactions is computed according to tables 4-6 and 4-7

8.6.1.1 Reads

Non-cacheable read operations (for code or data) use the READ transaction. Only single cycle (64-bits maximum) transfers will be used, *Viking* cannot burst transfer non-cacheable data. The transaction may be byte, half-word, word, or double word in length. *big-endian* word ordering is used (the least significant bytes in a word appear on the *high* bits of the bus, according to SPARC standards.)

8.6.1.2 Writes

Non-cacheable write operations are queued in *Viking's* store buffer. As soon as *Viking* receives a bus grant, the transactions will be issued on the bus. The processor will not wait during this time, unless the buffer fills (see section 4.12 — Store Buffer for a detailed description of store buffer operation.)

Burst writes will not be used for non-cacheable writes. Bytes, half-words, words, and double words may all be stored, with *big-endian* ordering. Any errors are reported as *deferred* data store errors. (Only after MMU protection has succeeded).

For synchronization reasons, all *cacheable* stores will wait until the store buffer has completed all non-cacheable stores before completing.

8.6.1.3 Atomic Operations

Non-cacheable atomic transactions are caused by execution of either SWAP or LDSTUB instructions. These operations are performed as a *locked* sequence of READ and WRITE transactions.

The *lock* bit field in the address phase of the read and write transactions will be set. *Viking* will not release the bus between the two transactions, unless explicitly requested by a relinquish and retry (R&R) reply. R&R replies are accepted for both read and write portions of atomic transactions.

8.6.1.4 Port Register Reads

MBUS port registers may be read using non-cacheable accesses to physical addresses in the range 0xff100000→0xfff00000, depending on module number being addressed. It is legal for a processor to address *it's own* port register. This is the only case of snooping on non-coherent read transactions.

See section 8.10, for full details of the port register.

8.6.2. Cacheable Operations

During normal operation, most transactions are *cacheable*. Cacheable transactions will use the level-2 consistency mechanism. The exact transactions requested by *Viking* depend on the instruction being executed, the cacheability of the reference, and the current state of the cache line.

8.6.2.1 Reads

There are three types of read transactions on the MBUS. Standard non-coherent READ transactions are used for non-cacheable operations. *Coherent Read* (CR) and *Coherent Read with Invalidate* (CRI) transactions are used for cacheable references.

CR transactions are used to read data from the *current owner*. The owner may be memory, or another cache. CR will be used for all data cache load misses, and all instruction cache misses. If another cache owns the data, it will respond by asserting the MIH_ signal, and providing the data.

CRI transactions are used to read data from the current owner for *write allocate* operations. Use of this transaction implies that the data will be modified upon

arrival at the processor. The current owner should relinquish ownership, all copies of the cache line should be invalidated. After the transaction completes, *Viking* will be the *exclusive owner* of the cache line.

All CR and CRI transactions use *critical word first* ordering. The doubleword which is needed first will be the starting address of the transaction. Doublewords from memory must be returned in *modulo 32-byte* address order. Once the needed data arrives, the processor will use it immediately.

Any processor which has a valid cached copy of data referenced by CR transactions must assert the MSH_ signal to indicate that the information is shared. *Viking* can accept the assertion of MSH_ at any time until receipt of the first data word.

If the data is owned by another cache, *Viking* will *ignore* any data ready responses until four cycles beyond the assertion of MIH_. This allows memory controllers to begin transmitting data sooner. Memory controllers must not respond with data until a time equal to the maximum MIH_ assertion delay for any cache in the system.

8.6.2.2 Writes

MBUS does not provide any true coherent write transactions. A processor must *own* data before it may write to it. As a result, all cacheable write operations are done locally to a processor cache. The WRITE transaction is used for all non-cacheable stores, and for cache *copyback* operations. WRITE operations to any shared data will cause CI transactions to be issued to all other shared copies.

Viking will relinquish ownership of a cache line under several circumstances: As a result of a copyback operation (using a WRITE transaction). In response to a CRI transaction for that line. In response to either a CI, or CWI transaction.

8.6.2.3 Atomic Operations

Since all cacheable data is modified locally, no special handling is needed for cacheable atomic operations. A normal CRI transaction is used to fetch read data, and an internal write operation will store the modified data locally.

8.6.2.4 Pure Consistency Operations

Viking will request only a single type of pure consistency operation (operations which transfer no data, but affect the state of a cached line). This is the CI, *coherent invalidate*. The CI will invalidate all cached copies of a line in the system. This is used when the processor attempts a store to a *shared* line, regardless of ownership. After the CI, the processor will become the *exclusive owner* of the cache line. The processor will wait for proper completion of the CI transaction before allowing the internal write to occur.

8.6.3. Table Walk Operations

All page tables should be treated as *non cacheable* in *Viking* direct MBUS systems. As a result, only level-1 transactions are used to reference page tables. During the duration of the table walk, *Viking* will maintain bus ownership, unless explicitly told to release by an R&R response. The lock bit in the MAD field will be set.

All levels of the page table will be read with standard single word READ transactions. Updates to R and M bits, or just M bits will be done with standard WRITE operations. Updates the the R bit only will be done with non-cacheable atomic swaps (See non-cacheable atomic references above).

This mechanism eliminates potential status bit inconsistencies when multiple MMUs are attempting to update page tables simultaneously.

8.6.4. Bus-initiated (consistency) Operations

Viking maintains cache consistency on the level-2 MBUS by snooping other processors transactions. The sections below describe three different classes of consistency operations that *Viking* must participate in.

8.6.4.1 Reads

Viking will respond with the proper MSH_ and MIH_ signals in response to a coherent read (CR) transaction on the bus. These signals are driven in the "A+3" cycle. If *Viking* owns the cache line, after asserting MIH_, *Viking* will wait an additional four cycles and respond with data. For CRI transactions, the MSH_ signal will never be asserted, MIH_ will be asserted if the processor owns the data.

The MSH_ pin is driven as an open collector style signal. It will only be driven low, it must be pulled inactive externally. Anytime MIH_ must be driven, it will be driven low for a cycle, then driven high again for a cycle, then tri-stated.

8.6.4.2 Write

Since write transactions are non-coherent, *Viking* does not snoop these operations. CWI transactions are treated as simple invalidates.

8.6.4.3 Pure Consistency Operations

Viking will treat all pure consistency operations as simple invalidations. This includes CI and CWI.

Since no reply from the processor is required, *Viking* can accept CI and CWI transactions at maximum bus rate, every other cycle. The rate of these transactions is controlled by system logic, and will generally be slower than two cycles per transaction.

8.7. Store Buffer Operation in MBUS Mode

Viking's store buffer will be used to buffer all copy back data, as well as non-cacheable stores in MBUS mode. Any errors on these transactions will be reported as deferred *data_store_errors*. See section 4.5.6 — Memory, Exceptions and No-Fault operation for details of these exceptions.

Viking's store buffer maintains consistency. During copy back operations, the store buffer becomes the *owner* of a cache line. The store buffer will snoop all coherent read operations and respond appropriately with the status of the line. By definition, the contents of the copyback buffer are *owned*. Data is always returned in *critical word first* order.

The following table describes the actions taken if a coherent transaction hits copy back data in *Viking's* store buffer:

Table 8-1 *Store buffer copyback snoop hit actions*

MBUS transaction	Action
CR	MIH_, MSH_, supply data, continue copyback
CRI	MIH_, supply data, cancel copyback
CI	cancel copyback
CWI	cancel copyback

8.8. Bus Arbitration

Viking requests the use of the MBUS with the MBR_ signal. This signal will be asserted when there is any internal bus cycle pending. Since *Viking* issues some transactions speculatively, the request for a particular transaction may disappear after the bus has been requested. The MBR_ signal will be deasserted immediately should this occur. If the internal request is still valid when the MBUS is granted, the transaction will occur. *Viking* will attempt to overlap arbitration with current bus cycles (including its own bus cycles).

A processor is granted future use of the bus by receiving the MBG_ signal. When MBG_ is received, the bus may still be busy servicing the previous owner. This will be indicated by the MBB_ (MBUS busy) signal. The MBR_ signal will be deasserted as soon as MBG_ is asserted. In order to gain ownership of the bus, the processor waits for its MBG_ signal to be active, and the MBB_ to be deasserted. Once this occurs, it will assert MBB_. After arbitration is complete, the processor will assert MAS_ to begin the transaction.

Bus busy will be maintained during locked and atomic transactions (like table walks). If *Viking* already owns the bus there will be no arbitration delay to begin subsequent transactions.

8.9. Error and Retry Handling

Several different transaction responses are defined by the MBUS. Successful bus cycles are terminated with the *Valid Data Transfer* acknowledgment (Ready).

Unsuccessful transactions may be terminated with several different error responses, or retries. *Viking* supports all error responses, but only one of the retry responses.

All error responses are applied to the current data transfer only. Any data received with a *Valid Data Transfer* response will be assumed correct and used internally. If any errors occur during the transfer of a cache line, the internal cache will not be validated. All data returned prior to the error may be used internally.

8.9.1. Errors

There are three error responses defined. These are ERROR1 (Bus Error), ERROR2 (Timeout), and ERROR3 (Uncorrectable). *Viking* treats all these errors in the same manner, and sets the MFSR to indicate the exact error response, as shown in the table:

Response	MFSR Setting
ERROR1 (Bus Error)	MFSR.BE
ERROR2 (Timeout)	MFSR.TO
ERROR3 (Uncorrectable)	MFSR.UC

All RETRY responses will be treated as ERROR3 replies.

8.9.2. Asynchronous Errors

Viking will assert the AERR_ signal whenever the processor enters error mode, or an exception occurs during a store buffer copyout. AERR_ will remain asserted until the MFSR register is cleared of the exception (it is clear on read).

8.9.3. Relinquish and Retry (R&R)

Viking supports *relinquish and retry* (R&R) replies only on the *first* response to any transaction, *including copyback operations*. After receiving an R&R, *Viking* will release bus ownership, and attempt to retry the transaction.

8.9.3.1 Retry

Viking does not support the retry reply. All retries will be turned in the equivalent of ERROR3 replies. *Viking* expects that all data returned to the processor with a READY response is truly correct. This data is used immediately by the pipeline. *Viking* can not tolerate late error responses under any circumstances.

8.10. Port Register

The MBUS port register contains vendor identification information for the component. Processors respond to READ transactions to an address which is based on the current module ID value (See MID above). See the MBUS specification for further details.

Viking will return the value 0x00000004 on MAD[31:0] in response to any read of its port register. This indicates device 0, revision 0 for the vendor (Texas Instruments). The version number may change in future releases of the component.

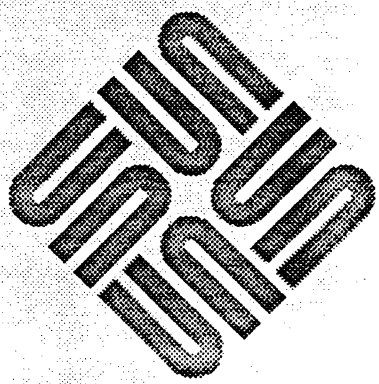
8.11. MBUS Pin Connections

Since *Viking's* bus interface is shared between MBUS and *Viking Bus*, unique pin connections must be made for each interface. The table below shows pin connectivity for *Viking* on the MBUS. Only connections unique to the MBUS are shown here.

Pin Type	Pin Name	Direction	MBUS Connection
Busses [108]	ADDR[35:0]	I/O	ADDR[3:0]=MID[3:0] ADDR[35:4]=n/c
	DATA[63:0]	I/O	MAD[63:0]
	DPAR[0:7]	I/O	n/c
Data Size [3]	BURST	O	n/c
	SIZE[1:0]	O	n/c
Request Strobes [3]	RGRT_	I	VCC
	WGRT_	I	MBG_
	BUSREQ_	O	MBR_
Access Strobes (13)	ARDY_	I/O	MAS_
	CCRDY_	I	VCC
	RRDY_	I	VCC
	WRDY_	I/O	MRDY_
	CCHBL_	O	n/c
	CMDS_	I/O	n/c
	CSA_	O	n/c
	LDST_	O	n/c
	SU_	O	n/c
	DEMAP_	I/O	n/c
	RD_	I/O	n/c
	WEE_	I/O	MBB_
	WR_	I/O	n/c
Exception Signals [9]	IRL[3:0]	I	IRL[3:0]
	MEXC_	I	MERR_
	PEND_	I	VCC
	RESET_	I	RSTIN_
	RETRY_	I	MRTY_
	ERROR_	O	AERR_
Clock	VCK	I	CLK
	VPLLRC	I	VPLLRC
	PLLBYP_	I	VCC
Cache [9]	OE_	I/O	n/c
	WE_[0:7]	O	n/c
Copyback [2]	OWNER_	I/O	MIH_
	SHARED_	I/O	MSH_

9

Viking Bus Interface



9

Viking Bus Interface

The *Viking Bus* interface provides a non-multiplexed highly pipelined bus interface for a variety of system applications. The interface is very flexible, and can be used to connect to a range of systems from low cost DRAM optimized designs, to high performance second level cache based machines.

9.1. Overview

The following sections introduce the basic transactions produced by the *Viking Bus*. The interaction of these transactions with the cache consistency protocol will be described as well.

9.1.1. Bus Transaction Overview

Read and write single transactions are the most basic bus operations. They will appear very similar in most designs. Block read and burst write transactions are used to improve *Viking*-bus bandwidth. Most transactions may be pipelined to provide additional performance improvements. A swap (atomic load/store) operation is defined to provide locked bus transactions.

As examples, the operation of these bus cycles in several different system configurations will be provided.

9.1.2. Cache Consistency

When using the *Viking Bus*, the processor is operating in *CC mode*. This forces the internal data cache to operate as a *write through* cache. Since the cache writes through, all modifications that the processor makes to cached data will be reflected externally. These transactions are used to keep internal caches, external caches, and other external processors up to date with any write operations.

Viking's internal caches are kept up to date by *invalidation*. Whenever an external bus master asserts the CMDS_ and WR_ signals (with DEMAP_ deasserted), any copies of data cached internally that match the address on the bus will be invalidated.

In systems with external caches, *inclusion* is normally maintained with the external cache. This allows the external cache controller to filter many system bus transactions, and pass on only those which require action within the first level caches to *Viking*.

9.1.3. MMU Consistency

"Demap" transactions are used by *Viking* to flush one or more pages from its MMU's TLB. Internally, these are generated by reference MMU flush operations (see section 4.11.7 — MMU Probe and Demap/Flush). The information transmitted for a DeMap includes virtual address and type, which the MMU uses as criteria to match pages in the TLB for removal. The context to be used for the demap is broadcast in bits 47 through 32. The lower 32 bits are equivalent to the data format of the flush operation. The exact format is:

Table 9-1 *Broadcast DeMap Data Format*

Rsvd		Context		VFPA		Rsvd	Type		Rsvd	
63	48	47	32	31	12	11	10	8	7	1

In addition to broadcasting this demap transaction externally, *Viking* can *receive* external demaps. When a demap is received, the processor executes the demap as if it had been generated internally, only using the provided context rather than the current internal context.

Viking requires a single ready reply for the demap operation. It is system hardwares' responsibility to ensure that the demap is broadcast to, and completed by all MMUs in the system. Incoming demaps use a *two phase* request/reply protocol.

> Important Note:
>
> If broadcast DeMaps are used, only a single DeMap transaction may be pending in the system at any one time. System software is responsible for maintaining this. Indeterminate operation may result if multiple DeMaps are in progress at the same time by any processor.

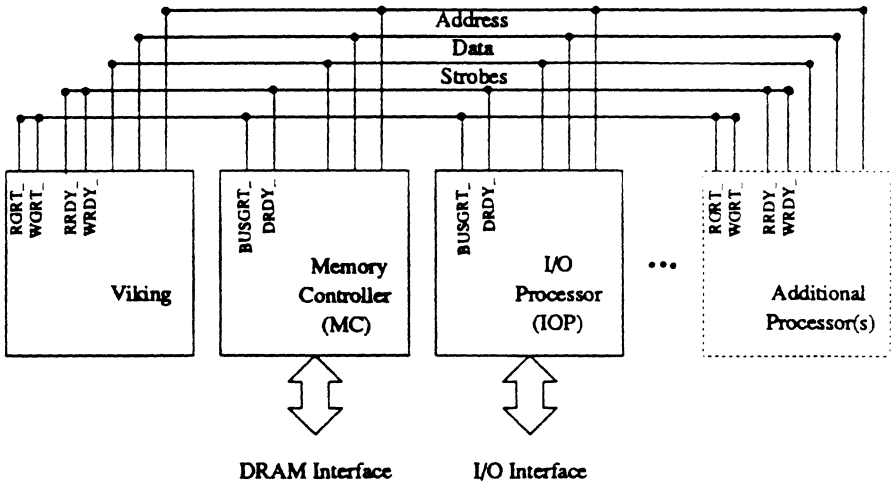
9.2. Systems Without External Cache

Viking systems built without an external cache are targeted towards lower cost, and generally lower performance applications. Good performance can be achieved in this environment by taking full advantage of *Viking's* on-chip caches, and providing low latency access to system memory. The *Viking bus* allows for efficient implementation of a variety of systems in this class.

The sections below will describe operation of such a system as an example. The system described assumes the use of *static column* mode dynamic RAM. Similar configurations could be built with *nibble*, or *fast page mode* DRAM.

An example of this interface is shown below:

Figure 9-1 *Viking Non-Cached System*



This diagram shows the principle components of such a system: *Viking*, a memory controller device, and an I/O processor. The diagram is a logical diagram only. Additional devices, and possibly buffering may be required depending on the exact implementation. The protocol allows for additional processors, although this may contribute unacceptable bus traffic and electrical loading.

Two simplifications are made to the protocol and interconnect for this environment. One involves *Viking's* bus grant signals, the other deals with its "data ready" signals. As discussed in section 9.3, *Viking* has separate bus grant signals for read and write: RGRT_ and WGRT_, which are separate to allow overlapped read and write cache misses. RRDY_ and WRDY_ are also separate, to clearly qualify data for these accesses. Since this overlap is not needed in a DRAM based system, RGRT_ and WGRT_ may be tied together, to form a single bus grant signal, shown above as BUSGRT_. Similarly, RRDY_ and WRDY_ are wired together, defined above as DRDY_.

9.2.1. Cache Consistency

In this configuration, cache consistency is managed with *invalidation*. Since *Viking* is operating in *CC mode*, all internal store operations will *write through* to external memory. All other caches in the system should see these write transactions and cause any internal copies of that cache line to be invalidated.

Viking's internal caches snoop on each others transactions as they appear on the bus. Only write transactions will cause invalidation to occur. All read responses will come from *main memory*, which is always correct due to the write through operation of the cache. Self modifying code is implicitly supported by this

protocol, FLUSH instructions are still required to guarantee proper operation (see section 4.4.4 — Flush (IFLUSH)).

9.2.2. Arbitration

When *Viking* needs to perform an external bus access, it must have access to the system bus. Access is granted to *Viking* by asserting the bus grant signals RGRT_ and WGRT_. In systems with external caches, these signals will normally be controlled independently. In simpler DRAM based systems, a single composite BUSGRT_ signal may be used, by connecting WGRT_ and RGRT_ together.

Viking supports lazy arbitration; if the bus grant is already asserted when *Viking* wants to begin an access, the access will begin immediately. If a grant is not present, the BUSREQ_ signal will be asserted until the bus is granted. Once the bus is granted, *Viking* may issue an access, as described in the following sections. The bus grant must remain active for the duration of the bus cycle. If the grant signal is deasserted, *Viking* will tri-state it's busses on the following cycle.

Important Note:

Viking will begin a transaction on the bus immediately if the bus grant signals are active. Since it is possible for *Viking* to begin a bus cycle at the same time external arbitration is removing these bus grants, system logic must monitor the CMDS_ signal to ensure that a transaction has not started as the bus grant was removed. Since external arbiters should ensure that an *empty* cycle exists between bus owners, this cycle may be used to detect the arbitration collision.

Once the bus grant has been deasserted, *Viking* will tri-state it's I/O signals on the next cycle. This will generally interfere with the transaction that has been started, although some transaction may still be completed.

When this situation occurs, it should be resolved by retrying *Viking's* transaction. When the memory or external cache controller detects the arbitration conflict (grant is deasserted and CMDS_ is asserted), it should immediately assert the RETRY_ signal. This will force *Viking* to cancel the pending transaction and re-arbitrate for use of the bus.

Viking also samples the OE_ pin to prevent collisions with returning read data from reads generated by the external cache controller. Although it is generally legal for *Viking* to overlap write cycles with outstanding read miss requests, this can cause a data drive conflict when the external cache controller is reading from the SRAM, and *Viking* tries to begin a store transaction. *Viking* will not initiate driving the data bus if the OE_ signal was asserted externally on the previous cycle. This case should generally be covered by arbitration, but is included for extra protection.

9.2.3. Error Reporting

All *Viking* bus transactions may have error responses. There are several different types of error responses which are encoded on the RRDY_, WRDY_, RETRY_ and MEXC_ signals. The encoding is similar to the encoding defined for MBUS error responses. RRDY_ and WRDY_ are interpreted the same way, qualified by read and write transactions. The table below defines the legal replies:

Table 9-2 *Reply Codes*

MEXC_	WRDY_/RRDY_	RETRY_	Reply Definition
1	1	1	No reply (idle)
1	1	0	Retry
1	0	1	Data transfer complete
1	0	0	Error: UD (Undefined)
0	1	1	Error: BE (Bus Error)
0	1	0	Error: TO (Timeout)
0	0	1	Error: UC (Uncorrectable)
0	0	0	Illegal Response (Reserved)

Throughout the bus cycle examples in this chapter, a standard error response with MEXC_ and either (or both) RRDY_ or WRDY_ asserted is used. This error will be interpreted as an uncorrectable error.

Other than retry responses, the error replies are all treated in exactly the same manner. The particular error type is decoded and use to set the appropriate bits in the MFSR. See section 4.11.11 for a full description of the MFSR register.

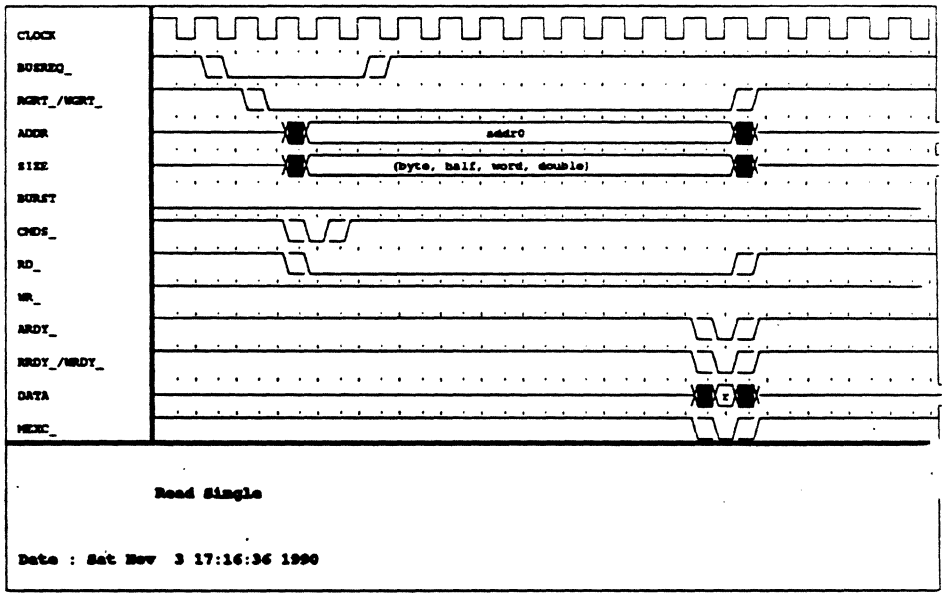
9.2.4. Memory and I/O Transactions

Memory transactions are generally cacheable, while I/O transactions are generally not. Cacheable read operations, with the exception of MMU table walk operations, will use read block transactions. Non-Cacheable transactions will always use single cycle transfers. The CCHBL_ signal will be asserted during all cacheable transactions.

9.2.4.1 Read Single

The following timing diagram shows a *Viking* read single operation. The read single is indicated by the BURST signal being inactive. Read singles may be one, two, four, or eight bytes; specified by SIZE[1:0]. An example of a read single is shown below. Timing will vary depending on the devices used.

Figure 9-2 *Read Single Protocol*



This diagram shows *Viking* arbitrating for the bus, using BUSREQ_ and BUSGRT_ (RGRT_/WGRT_). The *command strobe* signal, CMDS_, is then asserted, identifying the beginning of the access, and qualifying the initial address and strobes, such as RD_, WR_, BURST, and SIZE. The RD_, WR_ and SIZE[1:0] signals will remain active for the duration of the transfer. The BURST signal will be inactive for all read single operations.

The memory controller responds with ARDY_, to show the current address has been accepted, and that another may be sent. DRDY_ (RRDY_/WRDY_) is also asserted by the controller to qualify returned data. If an error is encountered on the read, MEXC_ may be asserted to terminate the access, as shown in the example above..

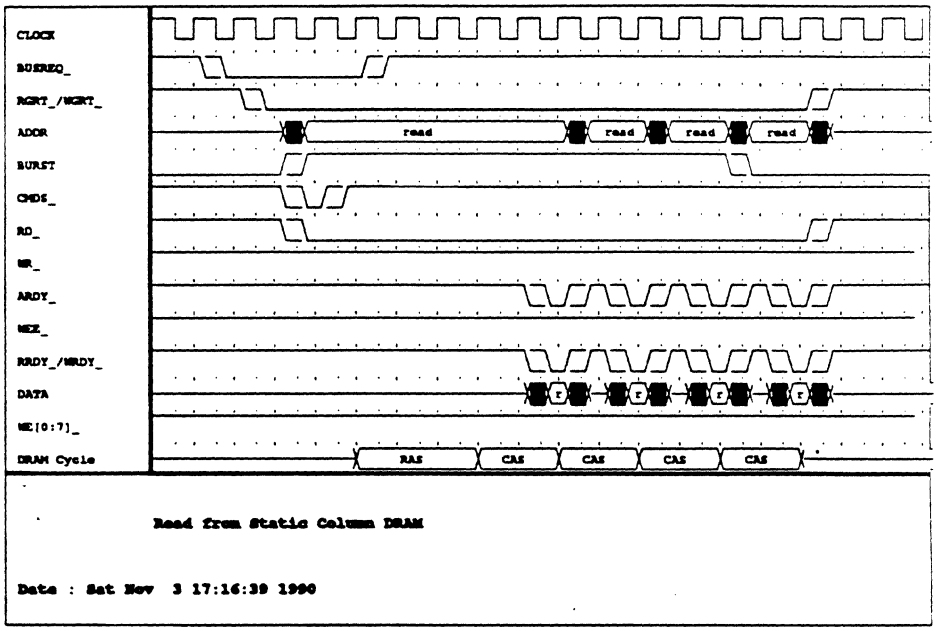
9.2.4.2 Read Block

Read block timing is similar in some respects to the read single. However, its target is always memory, since read blocks are cacheable accesses. 32-bytes of data are always returned. Read block latency is system dependent, and to a large degree determines system performance. A read block operation to static column DRAM is shown in the figure below. This figure assumes that the memory system is capable of providing one data word *every other* cycle.

The transaction begins in the same manner as the read single. The BURST signal will be active along with the command strobe assertion. BURST will remain active

for the first three transfers and deassert for the fourth, indicating the end of the transaction.

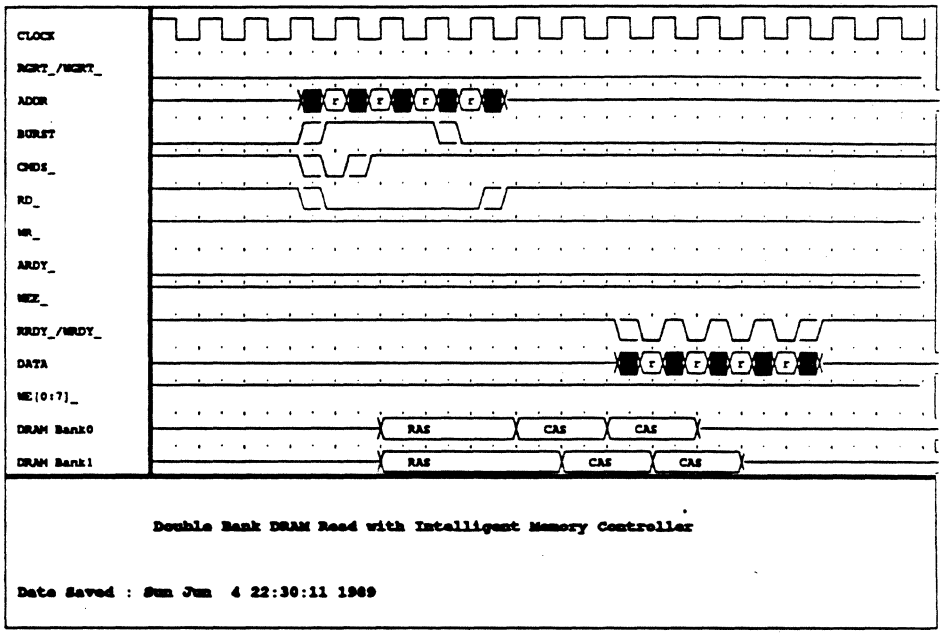
Figure 9-3 *SCRAM Read Block - Alternate Cycle Data*



The memory controller responds with ARDY_, to show the current address has been accepted, and that another may be sent. DRDY_ (RRDY_/WRDY_) is also asserted to qualify returned data. The last line, "DRAM Cycle", is given as an example of possible SCRAM timing.

In order to improve performance, it is possible to use *two banks* of memory to *interleave* data transfer. This can achieve twice the memory bandwidth of the previous example. Operation in a system such as this is described to show the flexibility of the *Viking bus* in supporting different memory configurations. After an initial access time, this model can return subsequent data doublewords every cycle. This example also shows no BUSREQ_; when *Viking* already has the bus (RGRT_/WGRT_ is asserted), it may begin an access immediately.

Figure 9-4 *Read Block - Data on Consecutive Cycles*

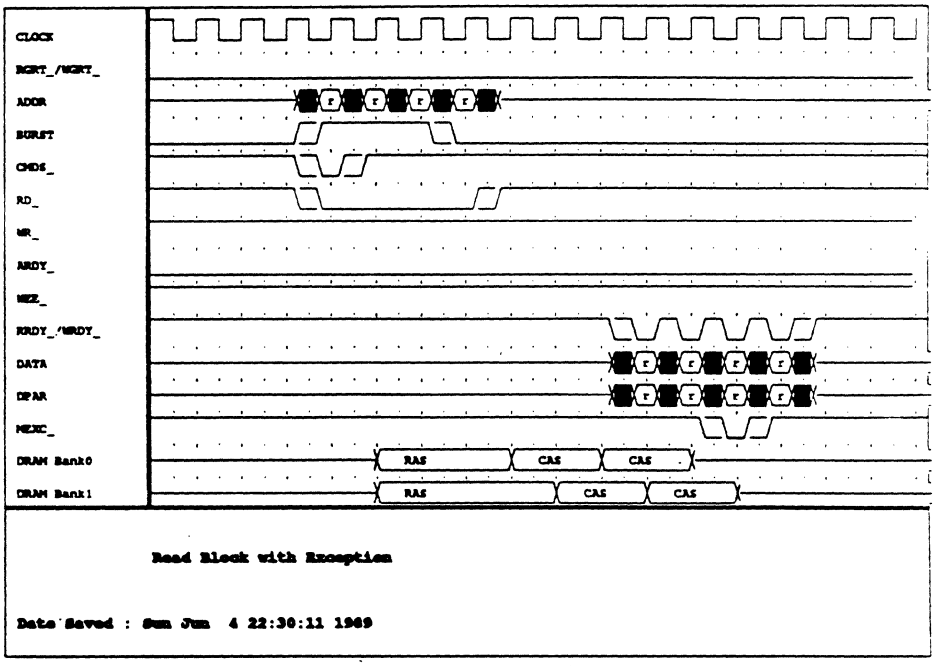


This configuration has ARDY_ constantly asserted, allowing *Viking* to send out all four block addresses immediately in consecutive cycles. This allows the memory controller to look ahead into the transaction and start memory cycles as early as possible.

The memory controller may return exceptions along with any of the four doublewords by asserting MEXC_, as shown below. *Viking* also checks parity on each of the incoming doublewords (if MCNTL.PE is enabled). Memory exceptions are signaled for both parity errors and MEXC_. The handling of memory exceptions is described in further detail in section 4.5.6.

Memory exceptions always invalidate the *Viking* cache block into which they were being placed. However, the execution stream actually responds to the exception only if the specific exception data was being demanded by the pipeline. In all other cases, the access is treated as a prefetch, and the errors are ignored. An example of memory exception signalling is shown below.

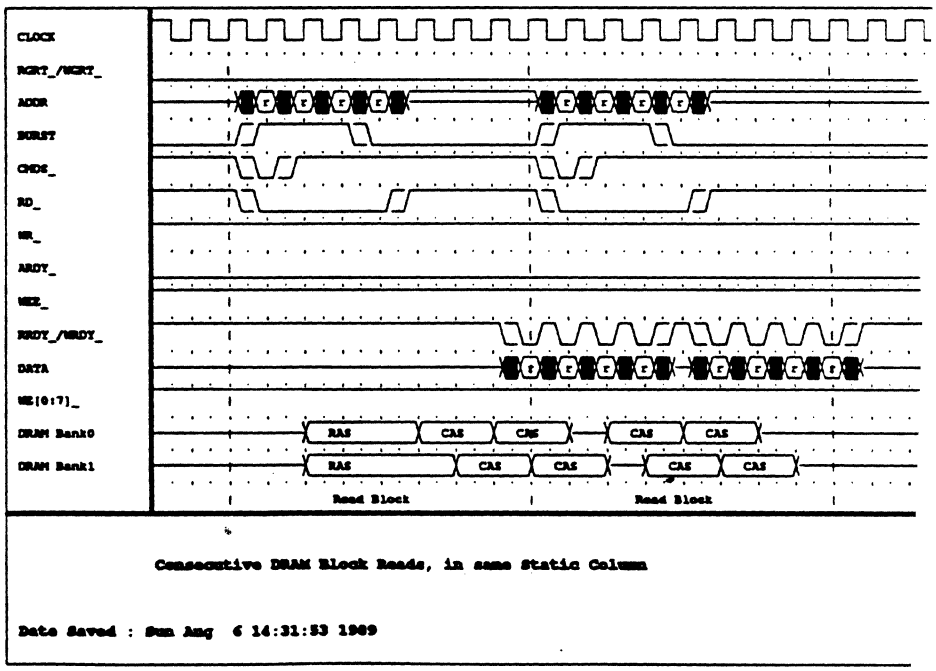
Figure 9-5 *Read from SCRAM with Exception*



9.2.4.3 Overlapped Read Blocks

To further improve memory usage, *Viking* can overlap memory reads to a large extent. The example below illustrates this overlap.

Figure 9-6 *Overlapped Read Blocks*



In this example, as in figure 9-4, ARDY_ is grounded, allowing subblock addresses to be issued in consecutive cycles. DRDY_ is asserted for each returned doubleword. However, the second read begins before the first has completed. Specifically, once *Viking* has seen the first DRDY_ of the previous access, the next may begin. When the read blocks are to the same SCRAM page, the effect is to combine them, into one nearly continuous stream of reads. Even without consecutive SCRAM hits, memory latency is reduced significantly. Typically, the memory controller will check if the two bus transactions are to the same RAM page, which would allow use of static column or page mode DRAM optimizations.

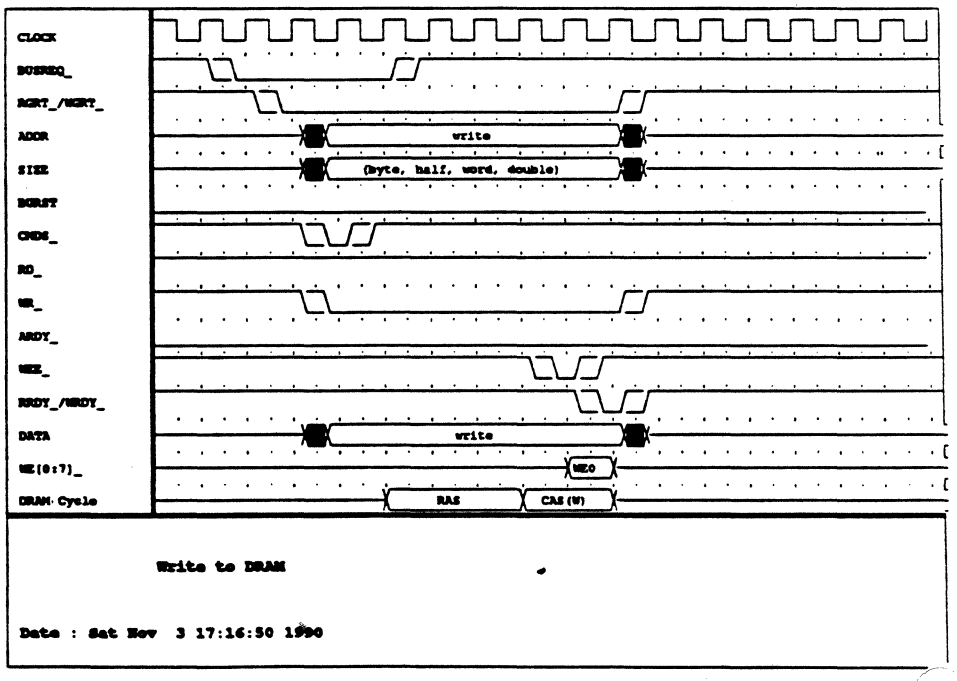
Note that overlapped read singles are precluded. A read cannot start until one DRDY_ is seen from the previous read; however, this single DRDY_ completes the previous transaction. This also prevents the potential complexity of overlapped I/O reads.

Overlapping may be disabled by negating the bus grant signals (RGRT_/WGRT_) after each read block has started. As an alternative, not asserting ARDY_ constantly will eliminate the overlap.

9.2.4.4 Write Singles

A write single is shown below. In this particular example, *Viking's* bus grant signals are not asserted, so *Viking* must request the bus first before proceeding. As with reads, CMDS_ is then asserted for one cycle, to begin the access. ADDR[2:0] indicates the starting position of the transfer on the data bus, and SIZE[1:0] indicates the number of bytes (1,2,4,8). WR_ identifies this transaction as a write, it will remain asserted for the duration of the write.

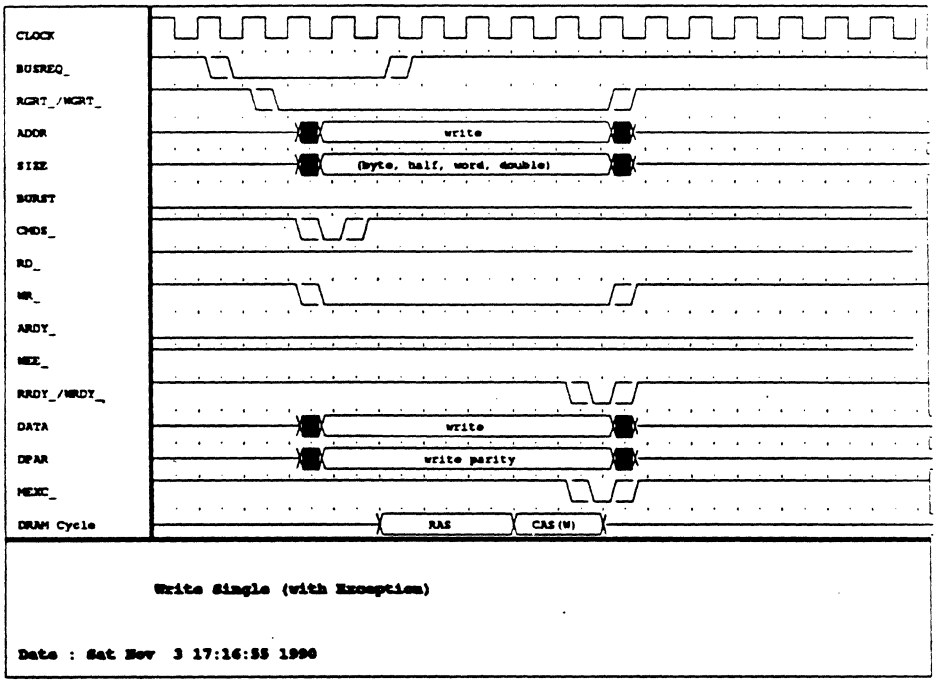
Figure 9-7 *Write Single to DRAM*



Note that Address and Data are driven throughout the write cycle, until a WRDY_ is received. Parity is generated and driven with data. Although ARDY_ is

grounded, as in the earlier read block cases, the address does not advance until data is also accepted, with DRDY_. This avoids confusion between consecutive write singles. Memory errors may also be reported by the memory controller with MEXC_ in this cycle, as shown below:

Figure 9-8 *Write Single to DRAM with Exception*



The WEE_ signal controls assertion of the WE[7:0] pins. If WEE_ is not asserted prior to the ready response, the WE[7:0] pins will remain in tri-state.

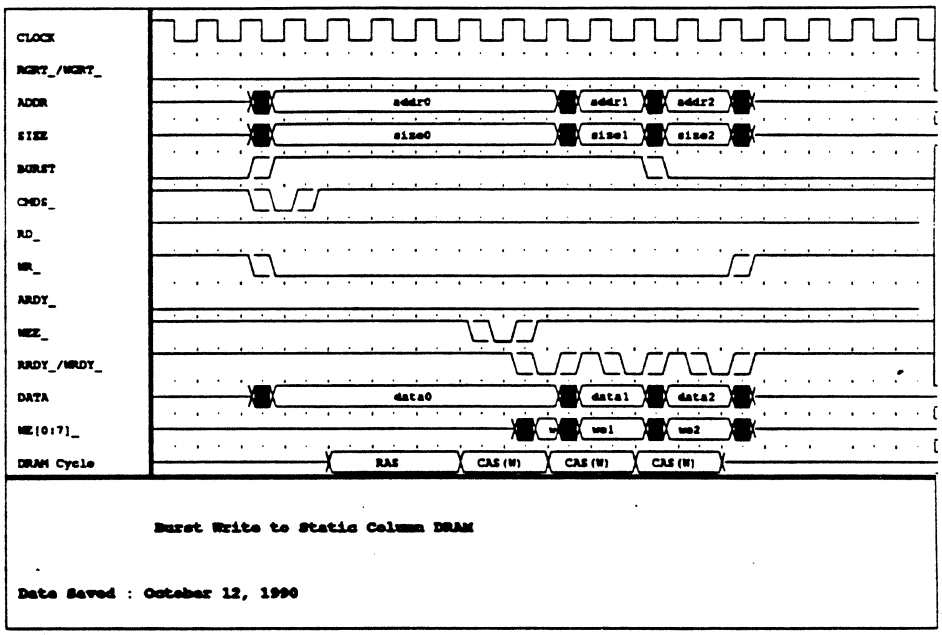
9.2.4.5 Burst Writes

Viking may issue an arbitrary length sequence of write single transactions as a *write burst*. This burst can continue as long as the store transactions in the store buffer are within the same 32-byte cache line. Any transaction crossing this boundary will cause the write burst to terminate. The individual transfers within the burst may be of any size. They need not be to consecutive bytes or words, only within the cache line.

This optimization is typically used to eliminate cache tag checks in configurations with an external cache, but it can also be used in DRAM based designs as described in this section. In particular, since the burst does not cross 32-byte boundaries, it is guaranteed not to cross the page address of most dynamic rams (static column, nibble mode, page mode, etc.). This allows optimized DRAM write cycles to be used, typically avoiding RAS time.

An example burst write is shown below. The figure displays a burst of three writes.

Figure 9-9 *Burst Write to SCRAM*



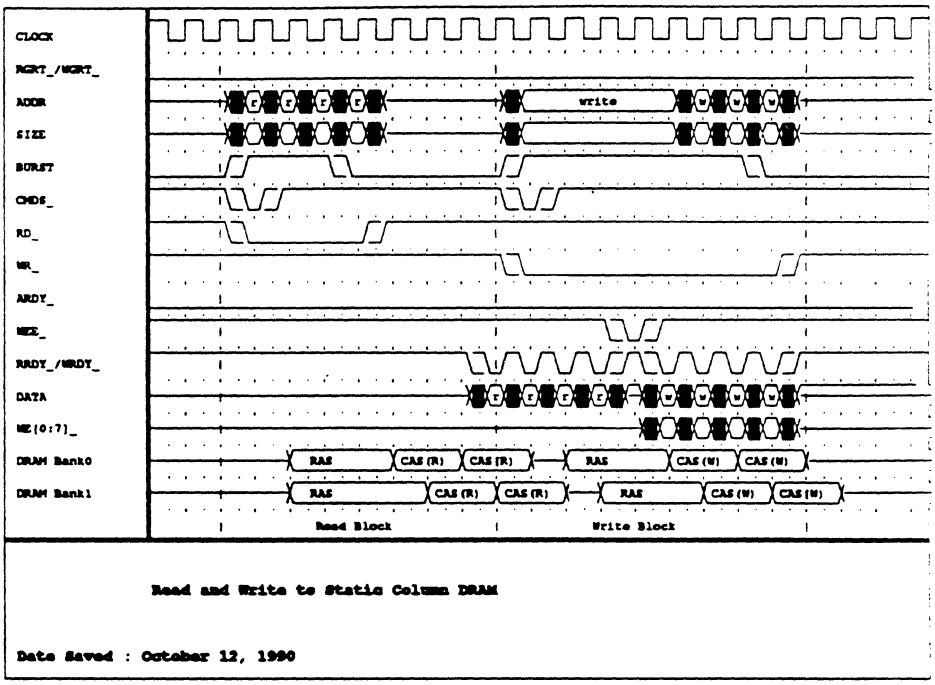
The burst is identified by two signals, CMDS_ and BURST. Command strobe is asserted once at the beginning of the burst. BURST will remain asserted until the beginning of the last transaction in the burst. The next data and address in the burst will appear in the cycle immediately following assertion of the WRDY_ signal. CMDS_ *will not* be asserted for each transfer. Assertion of the BURST signal indicates that there *will be* at least one more store in the burst.

The WEE_ signal is used to enable subsequent transfers in a burst write. If the WEE_ signal is not asserted prior to the first ready response, subsequent transfers will be forced to reassert CMDS_. This essentially disables all burst write transactions. The WEE_ signal also controls assertion of the WE[7:0] signals. If WEE_ is not asserted any time before the first ready, the WE[7:0] signals will remain in tri-state.

9.2.4.6 Overlapped Read/Writes

Read and write cycles may be overlapped in the same manner as overlapped reads. The initial write latency may be interleaved with a previous read. *Viking* can overlap any write with a preceding read block, as shown below. Store data is not valid until all data from the preceding read block has been returned. System logic is responsible for determining when the data is valid on the bus, there is no explicit signal to indicate this. *Viking* automatically inserts a single buffer cycle between incoming read data, and outgoing write data, to avoid bus collisions. Once store data is driven on the bus, it will remain until WRDY_ is asserted. Write bursts may overlap read blocks as well as write singles, since the protocol is identical. Only the first address will overlap with data being returned on the bus.

Figure 9-10 *Overlapped Read/Write Block*

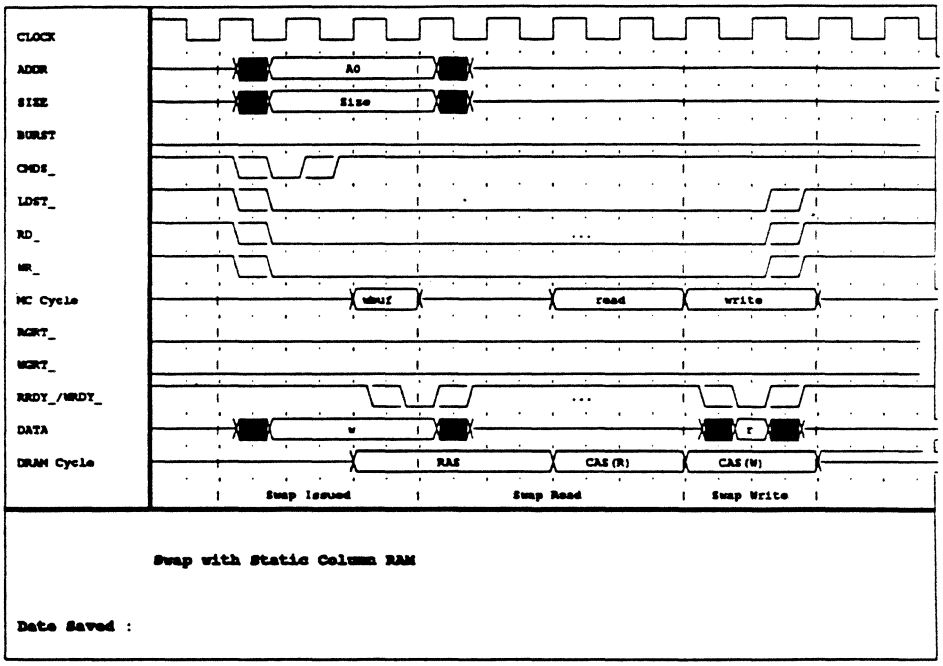


This overlap can be disabled by negating the bus grant signal (WGRT_) during the read block access.

9.2.4.7 Swap

Viking performs swaps by issuing writes before reads. This allows system logic to takes advantage of the read-modify-write cycle of most DRAMs, allowing the Swap to be performed in one memory cycle. The timing for this is shown below. Swap bus cycles are generated for the SWAP and LDSTUB transactions, as well as some page table status updates from the MMU. Two ready responses are required for this transaction: one for the write portion, another for the read. The memory controller must store the write data during the write phase, and only modify memory after the read phase has completed. System logic should not return data for the read phase until the RD_ signal has been asserted. Returning data sooner may cause bus drive conflicts.

Figure 9-11 *Swap with SCRAM*

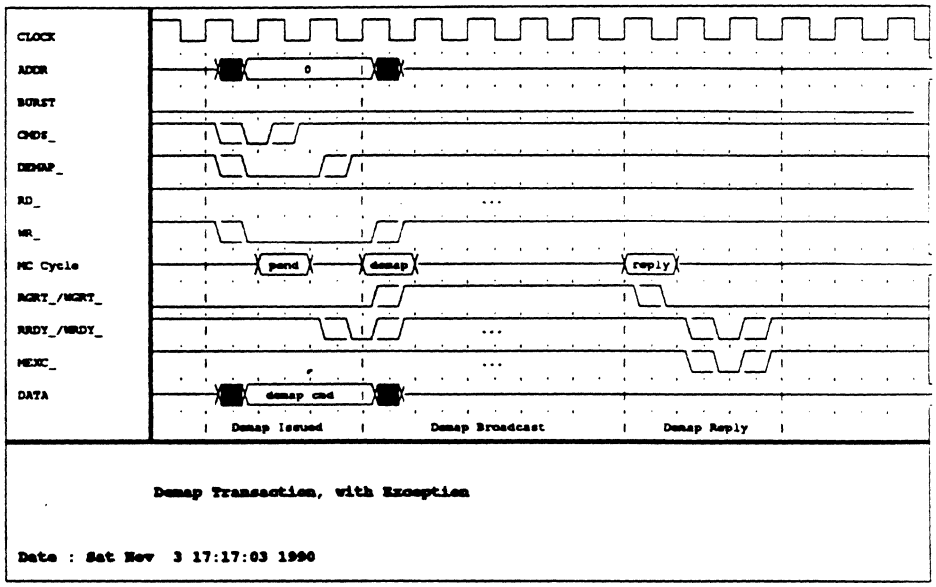


LDST_ identifies an access as an atomic Swap. SIZE[1:0] indicates whether one or four bytes will be swapped (these are the only legal sizes for swaps). LDST_, RD_, and WR_ are all asserted continuously for the duration of the swap operation. After the first DRDY_ assertion, *Viking* will wait for another DRDY_, indicating that read data is ready. To return this old data, the memory controller must buffer the store data, then read memory first. After data is returned to *Viking*, it can then write the contents of the buffer into the RAM, using the second portion of a *read-modify-write* cycle.

9.2.4.8 Processor Initiated Demap

Viking uses the Demap transaction to remove a PTE from all system TLBs. In this type of system, a demap may affect an I/O TLB, other processor MMUs, or simply be reflected back by the memory controller with no action taken in the system. The timing for the demap transaction is similar to a swap, two replies are required. The first reply (using WRDY_ acknowledges receipt of the demap request. The second reply informs the processor that the demap has successfully completed across the system, and is signalled with the RRDY_ signal. Both ready signals may be asserted at the same time, as shown in the diagram below, but two separate ready responses must be given. Exceptions may be reported to the demap, by asserting the exception along with the RRDY_ response.

Figure 9-12 *Processor Initiated Demap*



The demap is signaled by assertion of DEMAP_ and CMDS_. All information for the demap, including the virtual address, context, and command information, is passed on DATA[63:0]. The address is a "don't care" for demaps, and will be all zeroes. The format is defined in table 9-1.

Many systems may chose not to take any action in response to the Demap. They must respond to the demap with two RRDY_/WRDY_ assertions. In this case, the memory controller may hold RRDY_/WRDY_ active for two consecutive cycles.

> Important Note:
>
> If broadcast DeMaps are used, only a single DeMap transaction may be pending in the system at any one time. System software is responsible for maintaining this. Indeterminate operation may result if multiple DeMaps are in progress at the same time by any processor.

9.2.4.9 External Demap Requests

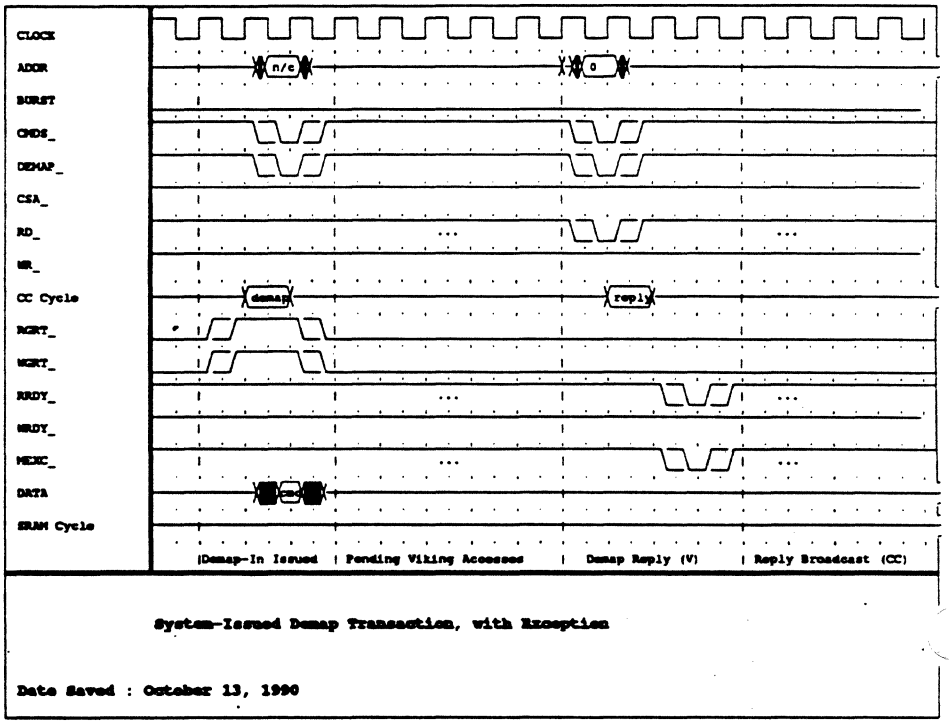
Incoming demaps use a *two phase* protocol. The first phase of the protocol is an *external demap request*. The second phase is a reply to that request. There may be other bus activity between the request and reply.

The request portion of the demap is issued by an external bus master activating CMDS_, and DEMAP_. DATA[63:0] should contain the demap command in the format described in table 9-1. This command should be issued on the bus for a single cycle. *Viking* will not respond to this request immediately.

Once the external request has been serviced internally, *Viking* will begin a demap reply transaction. This reply appears on the bus similar to an internally generated demap. The major difference being that it is signalled as a *read*, rather than write. The reply is signalled by *Viking* asserting CMDS_, RD_, and DEMAP_. System

logic should respond with a ready (RRDY_) to complete the transaction. An example external demap transaction is shown below.

Figure 9-13 *Externally Generated Demap and Reply*



9.2.5. Bus Monitoring

In *CC mode*, all consistency is done through invalidation. Whenever the bus grant (RGRT_/WGRT_) signals are deasserted, an external bus master is free to initiate an invalidation on the *Viking bus*. The external master must assert CMDS_, WR_, and ADDR[35:0].

9.3. External Cache Based Machines

Viking's Bus Unit supports an external second level cache, which significantly enhances performance. This E-cache may be built primarily from discrete SRAM chips, allowing it to easily track advances in commercial RAM technology. The on-chip control logic supports a simple, flexible cache design, for the different sizes, types and speeds of SRAMs which may be available upon *Viking's* completion.

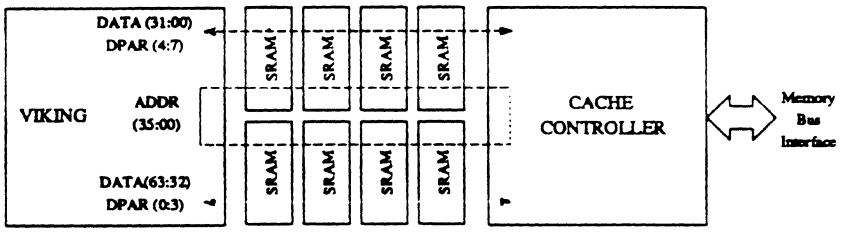
An external cache controller (CC) controls the cache and provides an interface to memory and I/O. The external cache controller also implements system cache consistency functions, in conjunction with *Viking* cache invalidations.

The key features and operations described in this section are pipelined memory transfers, overlapped cache accesses, and cache consistency operations.

9.3.1. Basic System Configuration

This section will describe general processor operation with the *MXCC* cache controller chip, along with pipelined cache RAMs. Detailed operation of the *MXCC* is described in a separate document — the *MXCC Specification*. The nominal *processor module* configuration is shown below:

Figure 9-14 *Viking with External Cache*



The external cache normally consists of eight SRAM chips. These RAM chips are organized as either 128Kx9 or 128Kx8. The "by nine" configuration provides for external cache parity.

The CC is responsible for controlling *Viking's* access to the cache, including bus grant and data ready signals. The CC must also process cache misses, and cache consistency operations (snoops). This requires an interface to a system memory bus, as shown to the right of the figure.

To optimize cache miss handling, *Viking* has separate bus grant and data ready strobes for reads and writes. The use of these strobes will be shown below in detail, as cache misses are discussed.

9.3.2. External Cache Organization

Many E-cache characteristics arise from *Viking*, the *MXCC* chip, system requirements, and the use of discrete RAM components. This section generally describes one external cache implementation, many alternatives are possible. Some of the characteristics of this environment are described below.

The E-cache is always a *physical address* cache. Typically, the cache is implemented as a direct mapped cache, but it is possible to construct set associative caches. The E-cache must have a line size of at least 32-bytes (the processor internal line size). Larger line sizes can be accommodated, with appropriate logic in the cache controller. Sub-blocking may be implemented, as long as the sub block at least 32-bytes. Processor data access is done on a 64-bit data bus. The system bus interface can be 64-bits, or possibly larger.

The cache controller is free to implement whatever cache consistency policy desired. The cache controller interacts with *Viking* using invalidation requests to propagate the consistency policy into the processor. Typically, the cache controller contains an integrated set of tags for the E-Cache. These tags are kept up to date as a *superset* of the information contained in *Viking's* caches. This allows the cache controller to snoop bus activity and only pass required invalidation

requests in to the processor. There is some loss of internal cache occupancy due to this policy of inclusion, which is minimized by the large size of the external cache.

The E-cache is a *single processor* cache. Generally, it should not be shared with other processors. This is largely due to the high bandwidth requirements imposed by a single *Viking* processor. Although it is not recommended, sharing can be accomplished by controlling bus arbitration as described above.

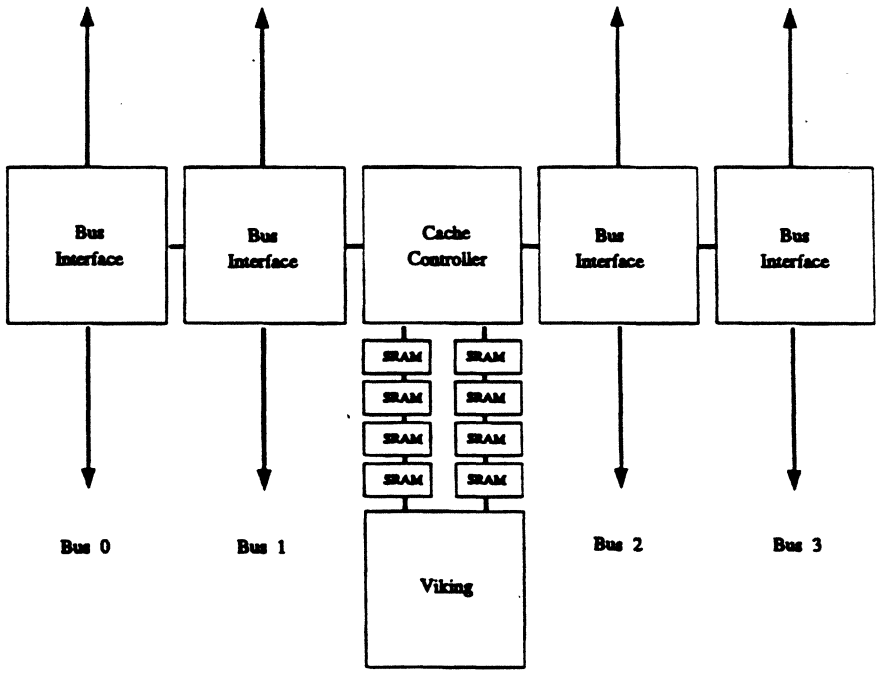
The E-cache protocol is optimized for single cycle pipelined transactions. This assumes registered input and output SRAM devices. An E-cache read takes three cycles to complete, throughput is one read per cycle (pipelined). Non-pipelined RAMs can be used, with reduced performance. A variety of access times can be supported (all timing is determined by the cache controller).

9.3.3. Consistency Requirements

The two major properties supporting E-cache consistency are *inclusion*, which ensures that all information stored in *Viking's* Instruction and Data Caches is also present in the external cache. A second property is for *Viking's* D-cache to write-through all stores to the E-cache.

An example of a processor with external cache is shown below:

Figure 9-15 *E-Cache Processor Configuration (four system busses)*



9.3.3.1 Write Through, Cache Inclusion

Cache inclusion is assumed in any *Viking* system with an external cache. This allows the cache controller to *snoop* system bus transactions on behalf of the processor. *Viking's* address bus is usually occupied, snooping all bus traffic would reduce performance. When a snoop write-hit occurs, the E-cache forces *Viking* to release it's bus (by deasserting RGRT_ and WGRT_ at the appropriate time, see section 9.2.2 — Arbitration). Once the bus is available, it can update the E-cache. At the same time, *Viking* snoops its address bus to determine if it must invalidate its caches also. This combined snoop mechanism provides a simple system interface, where consistency information comes from one source, the E-cache tags.

Cache inclusion may restrict what can be cached within *Viking*. *Viking's* 5-way associative Instruction cache and 4-way associative Data cache may not contain any data that is not also present in the E-cache. Whenever the E-cache replaces a line externally it must force *Viking* to invalidate any internal copies of the *victimized* line. This can lower the performance of the *internal* caches by forcing certain data to be invalidated where it would otherwise remain in the cache. This suggests that the E-cache should be several times the size of *Viking's* combined caches, a 256K-byte external cache is a good minimum size.

Invalidation occurs when the cache controller asserts CMDS_, WR_ and the snoop address on ADDR[35:0], enabling *Viking's* snoop logic. *Viking* can accept one snoop transaction every other cycle.

The impact of writing through all stores to the external cache is minimized by the *Viking's* store buffer, which allows the processor to continue even though some stores haven't yet reached the E-cache. The store buffer has lower priority than read transactions, so stores usually do not interfere with cache miss traffic. This interference is further reduced by burst writes and overlapped accesses.

9.3.4. Cache Hit Transactions

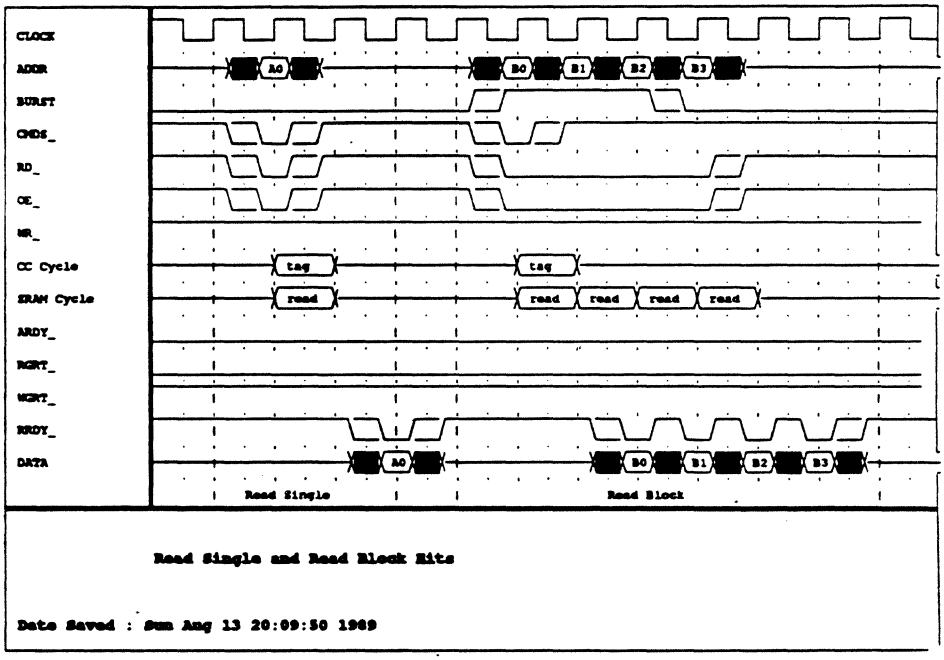
This section describes accesses which hit in an external cache. The protocol is very similar to *Viking's* access to main memory in DRAM based systems described in section 9.2. Several important differences exist, and will be described in this section. Bus grant signals for read and write accesses have been separated, as have read and write data-ready strobes. Another difference is that two strobes (OE_ and WE_[7:0]), are provided to control the cache SRAMs directly. The WEE_ (write enable enable) signal is used to control *Vikings* use of the WE_[7:0] signals.

Most of the timing diagrams that follow illustrate *Viking* bus timing with a cache controller chip like the *MXCC*, and pipelined single-cycle SRAMs. Many variations are possible, such as different memory timing. At the end of this section, several examples with different memory timing are provided.

9.3.4.1 Read Single and Read Block

The following timing diagram shows two cache read hits: a read single and a read block. RGRT_ is asserted throughout, allowing *Viking* to perform both reads without arbitration delay. For all read transactions, WGRT_ is not required, only RGRT_ is needed.

Figure 9-16 *E-Cache Read Hits*



To begin the read single, *Viking* asserts CMDS_, ADDR[35:0], SIZE[1:0], RD_, OE_. ADDR and OE_ are directly connected to the SRAMs, which latch the signals on the next clock rising edge.

After latching address and strobes, the cache controller performs a tag lookup and compare to determine if the access hits in the E-cache. The SRAM read takes place in parallel with this, in preparation for returning the data to *Viking*. This parallel access is possible since the E-cache is *direct mapped*. The data is latched in the SRAM data output latch on the next clock rising edge. In the following cycle, this data is driven on the DATA[63:0] bus to *Viking*. As the data is transmitted, the cache controller asserts the RRDY_ signal to indicate a cache hit. It the access were a cache miss, no ready would have been generated and *Viking* would have ignored any data driven on the bus.

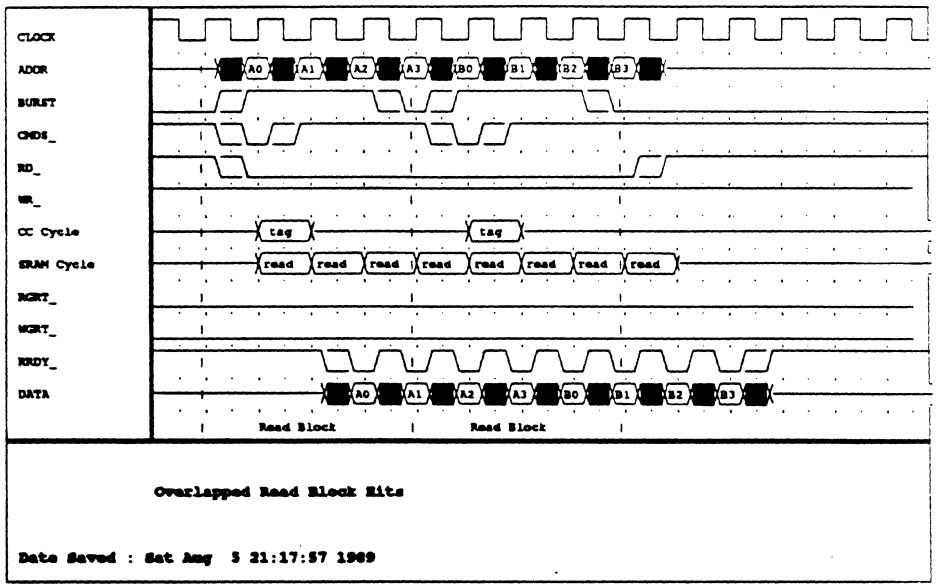
Since both outgoing address and received data are latched on rising edges, these accesses can be pipelined to form a read block. In these examples, ARDY_ is always asserted, allowing *Viking* to issue new addresses every cycle. Four addresses are issued, each requesting consecutive doublewords from the same 32-byte block (with wraparound). To identify the access as a block, *Viking* asserts BURST along with the first three addresses, then negates it for the last, marking the last address of the block. All block reads transfer exactly four double words. The RD_ and OE_ signals are asserted as long as addresses are driven.

The SRAMs latch all four addresses and OE_, and return data for each, delayed by two cycles. The cache controller also returns four RRDY_'s, it only needs to actually check tags once, since all four are always from the same cache block.

9.3.4.2 Overlapped Read Hits

The pipelined read mechanism employed for normal block reads also allows multiple cache blocks to be read consecutively, without buffer cycles. Maximum data read bandwidth is achieved using this overlap. The timing for two overlapped read blocks is shown below:

Figure 9-17 *Overlapped Read Hits*



Each block consists of four addresses, which read four doublewords from their respective 32-byte cache block. There is no relation between the blocks. The cache controller performs tag compares for both blocks, allowing them to be from arbitrarily different addresses.

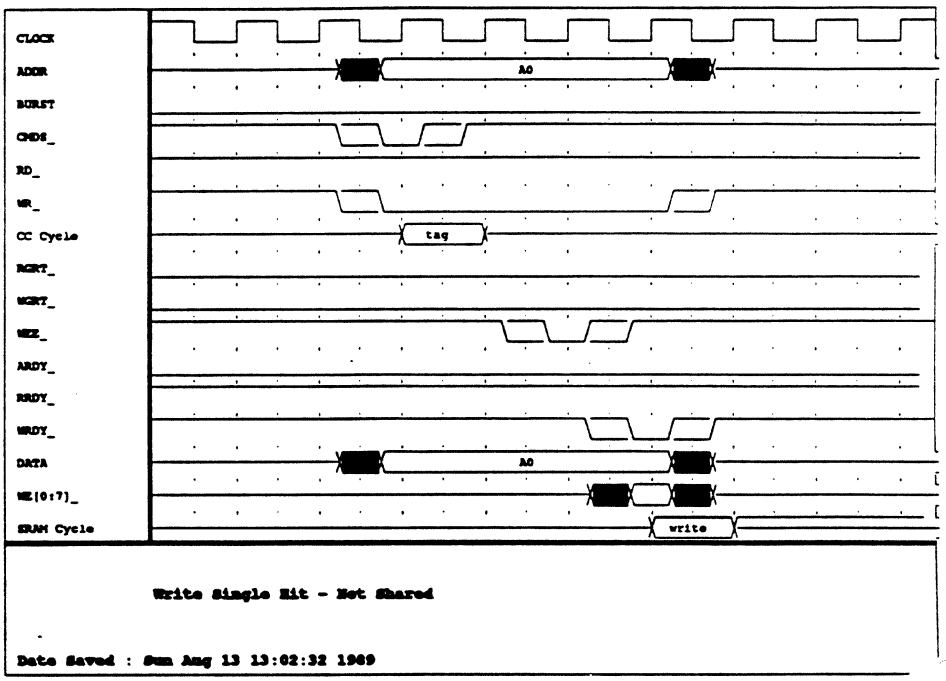
9.3.4.3 Write Singles

Cache writes require more time than cache reads. As shown above, read accesses always read the cache, whether or not there is hit. Writes cannot begin until the cache tags have been checked. Otherwise copy back cache data could be corrupted.

Since the E-cache tags are external to *Viking* several cycles are required to perform the tag compare before *Viking* can actually write to the cache. *Viking* issues CMDS_ and WR_. *Viking* drives address and data continuously, while the cache controller performs the tag check. The controller may respond by asserting either WRDY_ (for an E-cache miss), or WEE followed by WRDY_ (for a cache hit). In this example, ARDY_ is always asserted, but does not affect the assertion of new write addresses. Write address generation is controlled by WRDY_, not ARDY_.

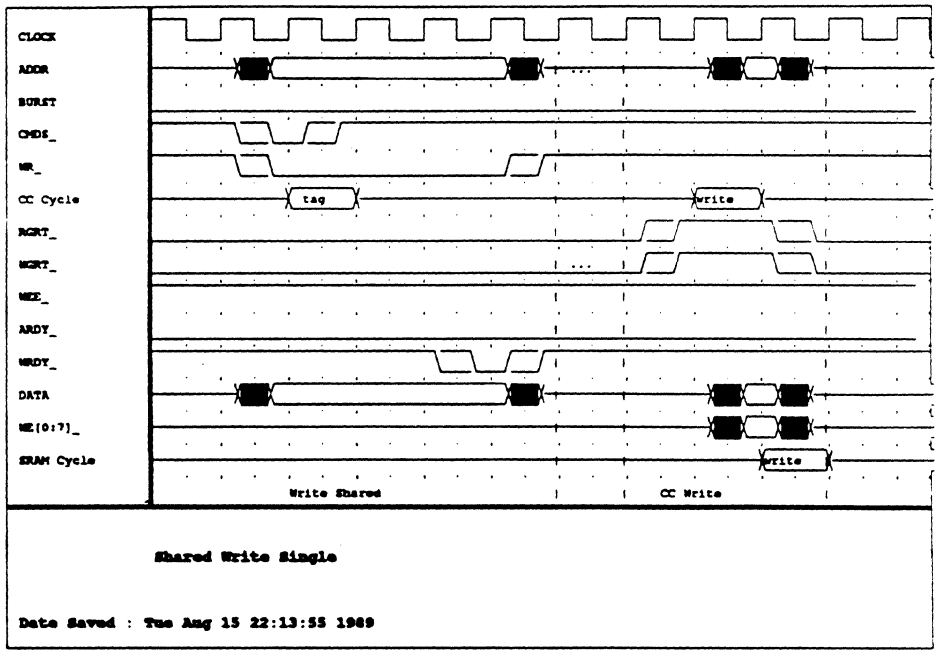
The cache controller asserts WEE_, (write enable enable), in response to the tag check. This allows *Viking* to assert the appropriate WE_[7:0] signals to begin a store to the cache RAMs. There are eight separate WE_ strobes, one for each byte, so that individual partial words can be stored without read-modify-write cycles. The write enables are asserted until WRDY_ is received from the cache controller, which terminates the write.

Figure 9-18 *Write Single Hit*



Depending on the cache consistency algorithm used, certain writes (for example to shared data) must be broadcast to the system before they are written into the external cache. In these cases, cache controller may respond with WRDY_, without asserting WEE_. This terminates the write as far as the processor is concerned, just as if a miss had occurred. Once the transaction has been completed in the system, the cache controller may gain bus ownership and actually perform the write to the cache RAM. This operation is shown below, it is a combination of a write miss followed by a snoop write-update. The write update portion will cause *Viking* to invalidate any internal copies of the data.

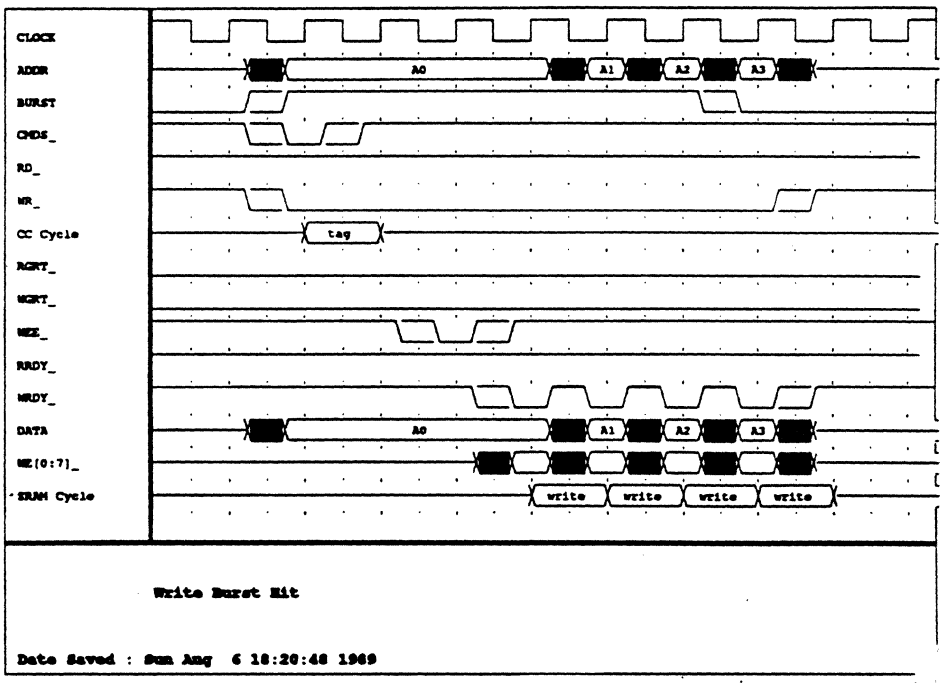
Figure 9-19 *Shared Write Single*



9.3.4.4 Write Bursts

The penalty for the write tag checks may be amortized among several consecutive writes, if they are to the same cache block. Although each store is generated by a separate Store instruction, *Viking's* store buffer can recognize these nearby stores, and group them. The length of the write burst is arbitrary. It will continue as long as store transactions to the same cache block continue. The example below illustrates four consecutive transfers in a write burst. Four double word (64-bit) stores, four transfers is the most likely duration, since that covers the entire 32-byte block. Each transaction in the burst may have a different size, and does not need to be in any order.

Figure 9-20 *Write Burst Hit*



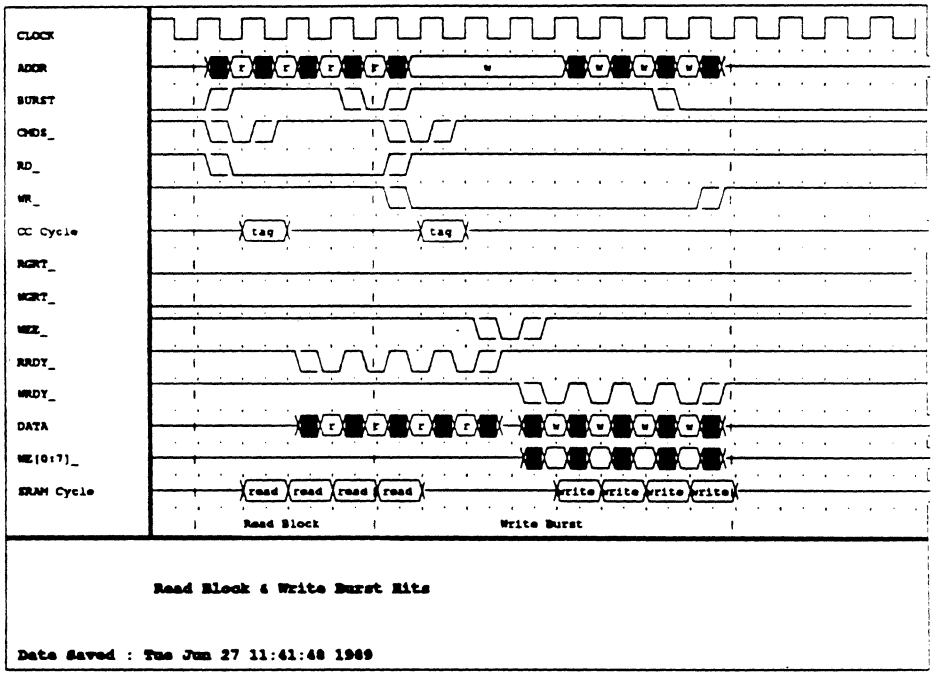
Once the cache controller has determined that the access has hit it issues WEE_, as for normal write hits. Since this qualifies the whole 32-byte cache block for writing, *Viking* may perform multiple writes in consecutive cycles. This will continue until either the writes are exhausted, or the controller requires the bus and negates WGRT_. See section 9.2.2 — Arbitration for more details on arbitration.

9.3.4.5 Overlapped Read/Write Hits

The penalty for the store tag check can be completely eliminated in many cases. In the same way that reads are overlapped, *Viking* allows a preceding cache read block to overlap the tag-check for a write block. This takes advantage of the fact that, for read blocks, the address bus is free before the data bus. Since only the address is required for the write's tag check, it may be driven out immediately, even though data is still returning from the previous read block. The three-cycle pipelined read timing is well suited to this mechanism: the read block and write tag compare both finish at exactly the same time, allowing the SRAMs to be written with no additional delay. *Viking* will drive valid data after a one cycle bus turnaround delay after the last RRDY_ for the read burst. The cache controller should assert WEE_ no sooner than one cycle after the last RRDY_. The first WRDY_ assertion should be in the cycle following WEE_.

In this example, a write burst is shown following a read block.

Figure 9-21 *Overlapped Read/Write Hits*



9.3.4.6 Swap

To support packet-switched system busses, *Viking* performs Swaps (atomic read/writes) by issuing the write before the read. Otherwise, an atomic swap would require two bus transactions and a locking mechanism on the packet switched bus.

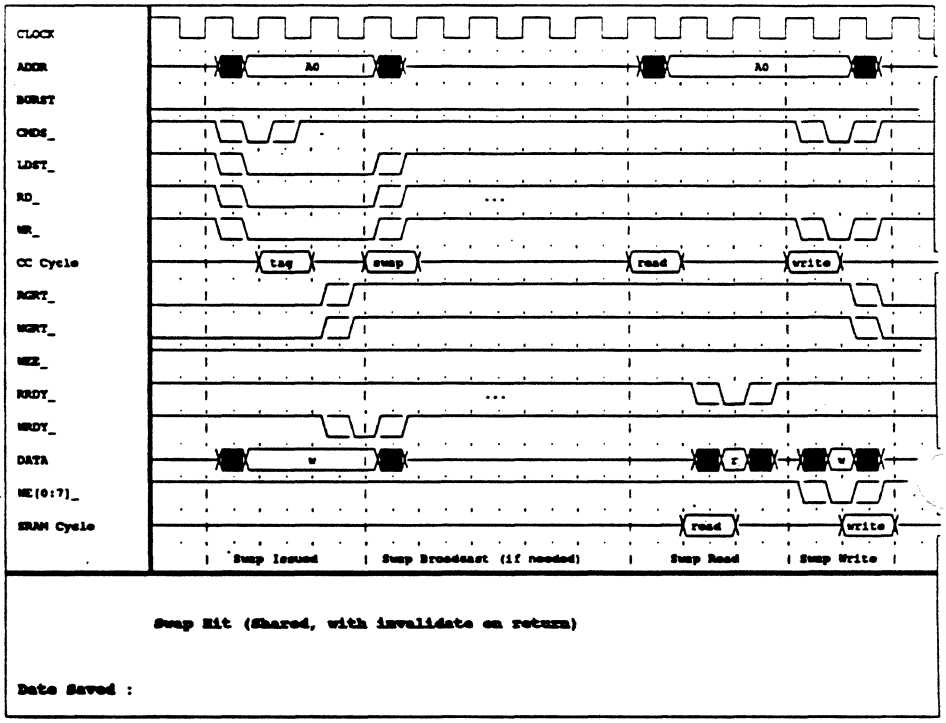
The swap begins much like a write single. *Viking* drives out CMDS_, along with ADDR, DATA, WR_, LDST_ and RD_. The LDST_ signal is equivalent to the logical AND of RD_ and WR_. The OE_ is not driven by *Viking* at any point during the swap. The cache controller latches the data and returns WRDY_, ending the write portion of the Swap. The cache controller should check the cache tags to determine how the remainder of the swap must proceed. In the simplest case, the external cache is the exclusive owner of the referenced data, allowing the swap to complete locally. If the data is shared, the swap must be broadcast to the system prior to local completion, which complicates the bus cycle.

Once the write portion is complete, the cache controller should negate the bus grant signals, RGRT_ and WGRT_. This will force *Viking* to relinquish the bus, allowing the cache controller to control the SRAM and complete the reference as required.

If the access can be completed locally, the cache controller reads the SRAMs, by asserting addresses and OE_. When the data is valid from the SRAM, the cache controller should signal the processor to accept this data by asserting the RRDY_ signal. Once this read is complete, the cache controller can retain ownership of the bus, and perform the modify portion of the swap to the SRAMs, by asserting the stored data, address, and the appropriate WE[7:0]_ signals.

If the cache block is shared with other processors the cache controller must broadcast the store across the system bus before supplying *Viking* with the previous data. This is done to ensure proper system synchronization. Once the broadcast has completed, *Viking* can return the correct data to the processor by driving the data bus and asserting RRDY_. The write update portion of the swap can then proceed as above. An example of this case is shown in the figure below.

Figure 9-22 *Swap Hit (Shared, with invalidate)*



Note that WEE_ must never be asserted, even if the access is not shared. WEE_ would allow *Viking* to write directly into the SRAM, overwriting the old data, which must be returned to *Viking* during the read phase of the swap.

9.3.4.7 Demap

Demap transactions, both processor and system initiated, operate in the same manner as described for non-external cache systems. They will not be described further here.

9.3.5. External Cache Misses (and Non-Cacheables)

In most applications, the external cache will have an outstanding hit rate, approaching or even exceeding 99%. Still, long miss penalties can have substantial effects on performance. Some of this performance loss can be reclaimed by efficiently using the *Viking* bus in the presence of cache misses. The effect is to reduce and sometimes to even eliminate the penalty of cache misses.

Basic read and write miss protocols are described below. Overlapping bus accesses to increase performance during cache misses will be presented in detail. These overlapped accesses may result in *Viking* generating two concurrent cache misses, one from a read miss, the other from a write miss. This can makes good

use of packet switched buses, which can support multiple outstanding requests. Non-cacheable reads and writes are commonly used to access I/O devices, and are described to conclude this section.

9.3.5.1 Read Misses

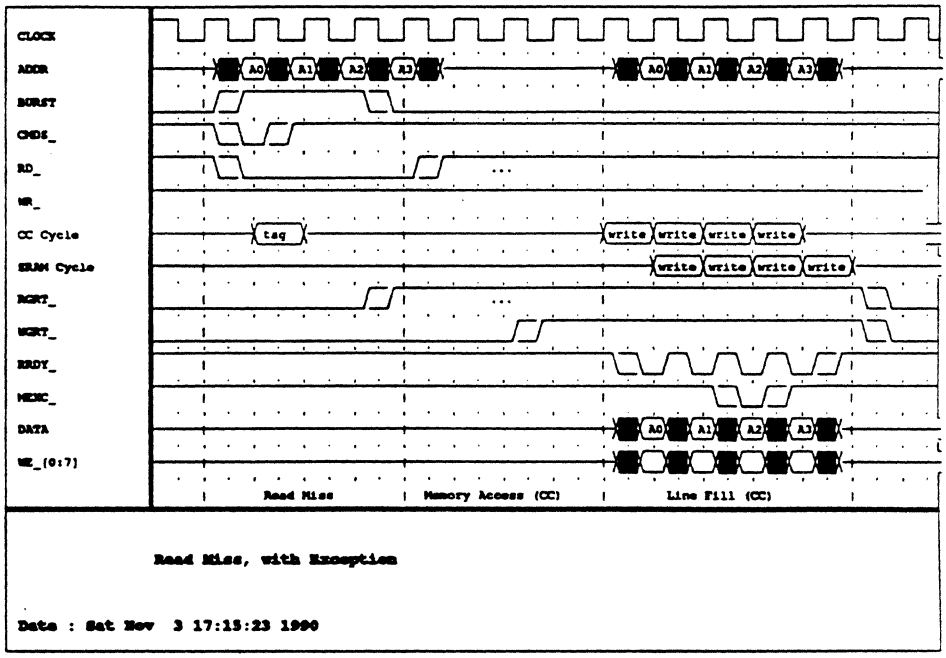
A read miss begins as a normal read access. Because the E-cache tag compare indicates a miss, RRDY_ is not asserted. Any data that may be returned by the E-cache RAM during this time is ignored. *Viking* must wait while the cache controller accesses main memory to obtain the required cache block. During this time, the controller negates RGRT_, preventing *Viking* from issuing any additional reads.

While waiting for a response from memory, the WGRT_ signal may remain asserted, allowing write operations to overlap the read miss. When data is about to return from memory, the cache controller should deassert WGRT_ (taking into account any pending write operations). This action allows the cache controller to control the SRAM, and initiate a write operation when the read data arrives from memory. The sequence for this is shown below. Examples of overlapped transactions are presented later.

As data returns from main memory, the cache controller writes it into the external cache. Note that the "CC Cycle" and "SRAM Cycle" lines are skewed by one cycle, because of the one cycle delay imposed by the pipelined SRAM. To perform the writes, the cache controller drives ADDR, DATA, and WE_[0:7], one cycle for each doubleword.

At the same time that data is written to the cache, the RRDY_ signal is asserted, which will allow *Viking* to sample the data at the same time it is written in to memory.

Figure 9-23 *Read Miss*

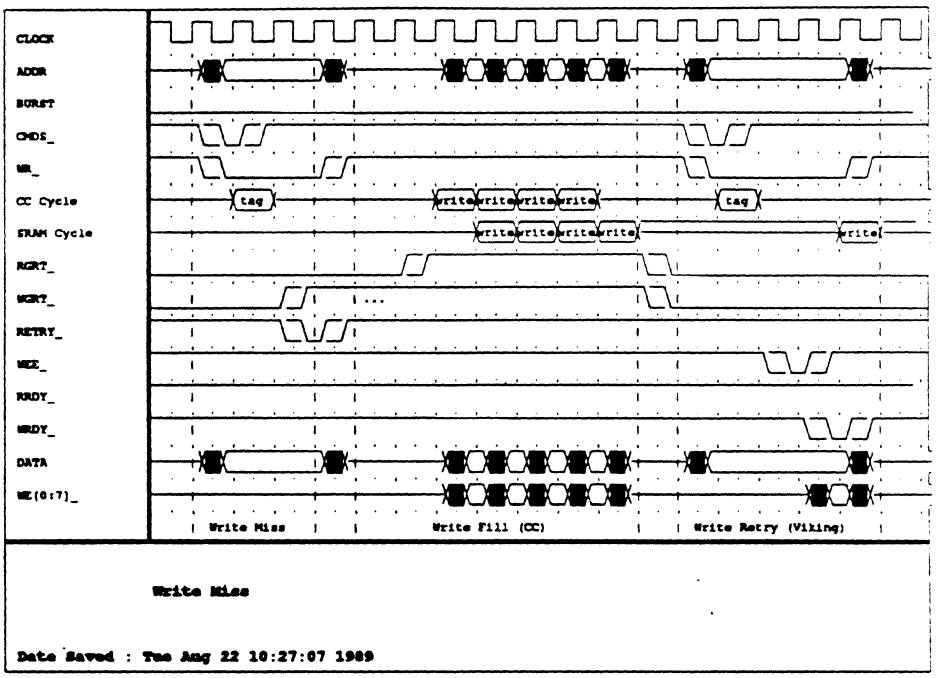


Exceptions can be reported on any of the four incoming doublewords. The timing diagram above shows an error on the third doubleword. All such exceptions cause both the CC and *Viking's* on-chip caches to invalidate their respective cache line. A trap is only reported to the pipeline if the exception occurs on the first word, and is a demand access. Exception handling is described in more detail in section 4.5.6.

9.3.5.2 Write Misses

Write cache misses are handled quite differently from read misses. When a miss is detected, the write operation is *aborted* by asserting the RETRY_ signal. By deasserting the write grant signal (WGRT_), the processor is prevented from attempting this retry until the bus is once again granted for writes. Once the *write allocate* to the E-cache has been completed by the cache controller, the write is retried, and now hits in the external cache. This protocol is required to prevent *Viking* from driving ADDR and DATA until it received a WRDY_, which would prevent the CC from being able to write the required cache line. The timing for this is shown below.

Figure 9-24 *Write Miss*



The write is started by driving ADDR, DATA, CMDS_, and WR_. The cache controller performs a tag compare. After determining the write is a miss, the cache controller asserts RETRY_ and negates WGRT_, and begins to handle the miss. The RETRY_ assertion tells *Viking* to end the transaction, and retry it later. The WGRT_ negation tells *Viking* that it cannot attempt to retry the write yet. WGRT_ stays negated until the CC has finished fetching the missed cache block.

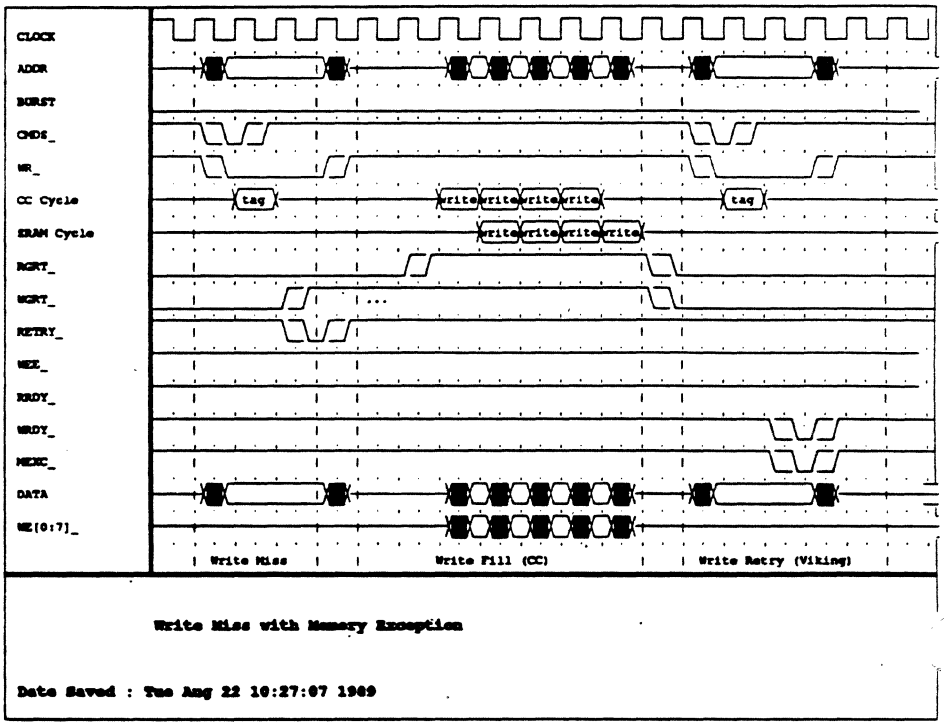
As the missed block arrives from memory, the cache controller writes it into the cache, shown above as "Write Fill". The is similar to read fill operations for read misses. One major difference is that RRDY_ is not asserted as the doublewords come in. The write fill operation is invisible to *Viking*.

When the write fill is done, the cache controller asserts WGRT_, allowing *Viking* to retry the write. The retried write is shown in the timing diagram as "Write Retry (Viking)". This time it hits in the cache, and completes normally.

The memory controller may encounter a memory exception on any of the four incoming doublewords. Since the write fill is invisible to *Viking*, the error cannot be reported immediately to the processor. The preferred action for the cache controller to take is to delay reporting the error, and allow *Viking* to retry the write. The cache controller can then report the error synchronously to the write, by asserting MEXC_ with WRDY_. The write is guaranteed to be associated with the error, since *Viking* always performs writes in order. This matches *Viking's* store exception handling policies, for both buffered and synchronous writes. Synchronous write exceptions are reported directly to the processor pipeline, while errors on buffered stores cause a store buffer exception to occur. See section 4.12.5 — Store Buffer (Data Store) Exceptions for a more detailed description. An

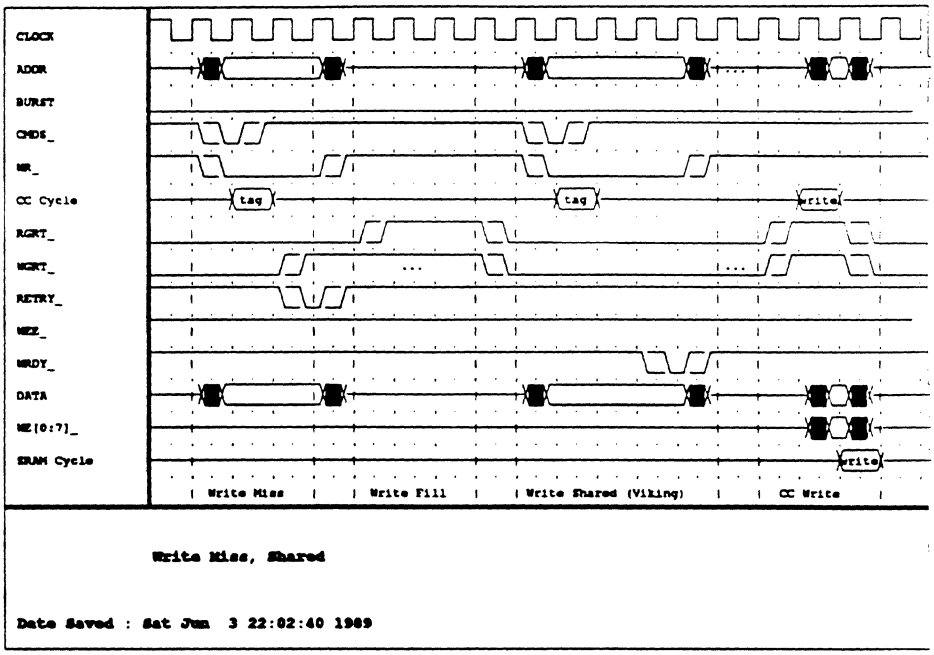
example of this is shown below. The cache controller must invalidate the tags for the data being fetched in this case.

Figure 9-25 *Write Miss with Exception*



In some systems, depending on the state the fetched data, additional system transactions may be required before the write miss is completed. An example below illustrates broadcasting a shared write to the system immediately following the write miss. This transaction combines a write miss with a shared write hit, as described earlier. Notice that the WEE_ signal is not asserted. This allows the write to complete from the processors perspective, while it has still not propagated through the system. When the system operations are complete, the cache controller will update the external cache by performing it's own write operation. The exact operation of this type of sequence is system dependent.

Figure 9-26 *Write Miss, Shared*

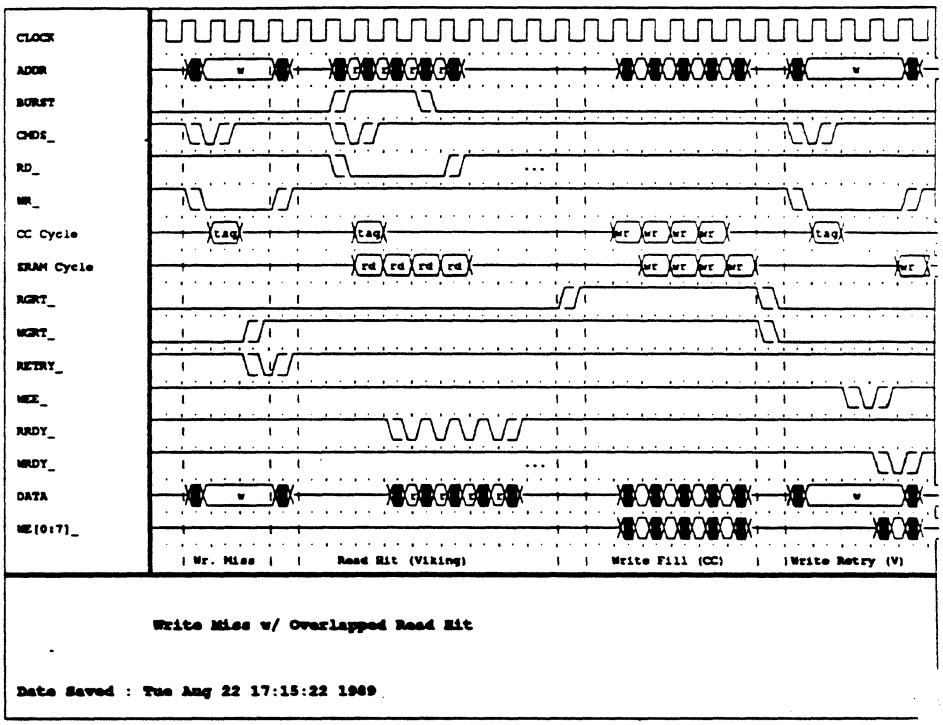


The shared write broadcast may also encounter an error; it would be reported synchronously, by asserting MEXC_ with WRDY_.

9.3.5.3 Overlapped Hits and Misses

Cache misses may require many cycles in some systems. Previous examples have shown the *Viking* bus to be idle while misses are being processed. In many cases, *Viking* can make use of this idle cache bandwidth. Except for certain synchronization points, *Viking's* reads are issued independently of writes, and vice versa. By separating RGRT_ from WGRT_, and RRDY_ from WRDY_, the *Viking* bus can also treat these independently. As a result, when writes are blocked by an external cache miss, *Viking* can still issue reads to the cache. This prevents a store, which is usually buffered in *Viking's* store buffer, from potentially blocking the IU-pipe by interfering with external cache reads. An example is shown below.

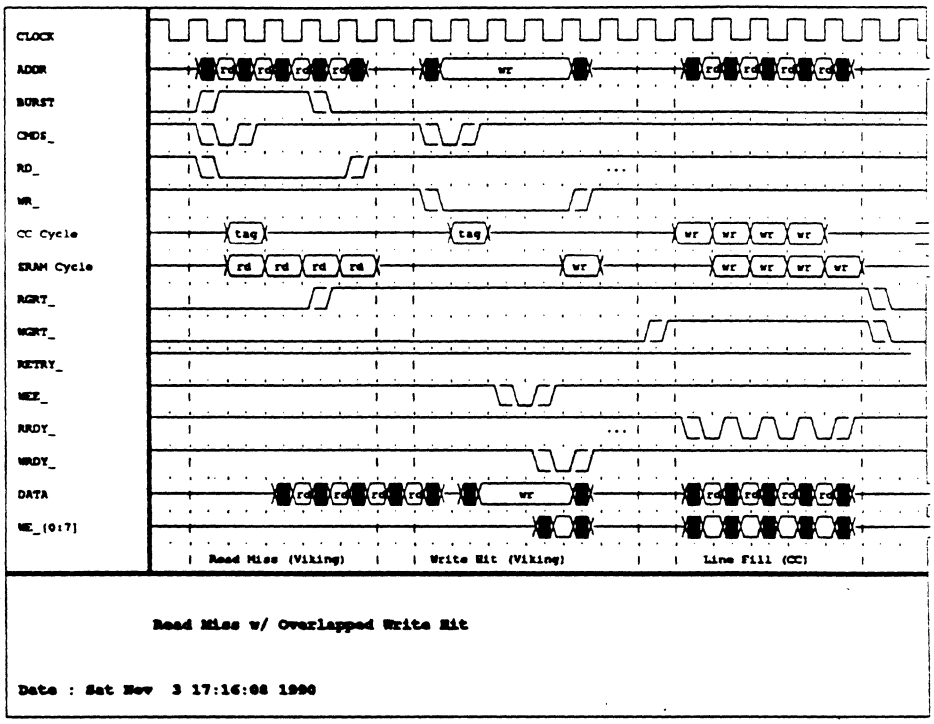
Figure 9-27 *Overlapped Write Miss and Read Hits*



After a write miss occurs, the cache controller negates only the WGRT_ signal. RGRT_ remains asserted, allowing *Viking* to issue read transactions. When the cache controller has received the write miss data from memory, it disallows further reads by finally negating RGRT_. While both WGRT_ and RGRT_ are negated, the cache controller writes the incoming line into the cache SRAMs. Then, *Viking* can retry the write at its convenience (other intervening transactions can occur).

Conversely, while *Viking* is waiting for a read miss to be processed, it can issue stores to copy its store buffer to the external cache. This allows the store buffer to drain to the external cache while a read miss is being processed, using otherwise idle bus time. An example of this is show below.

Figure 9-28 *Overlapped Read Miss and Write Hits*

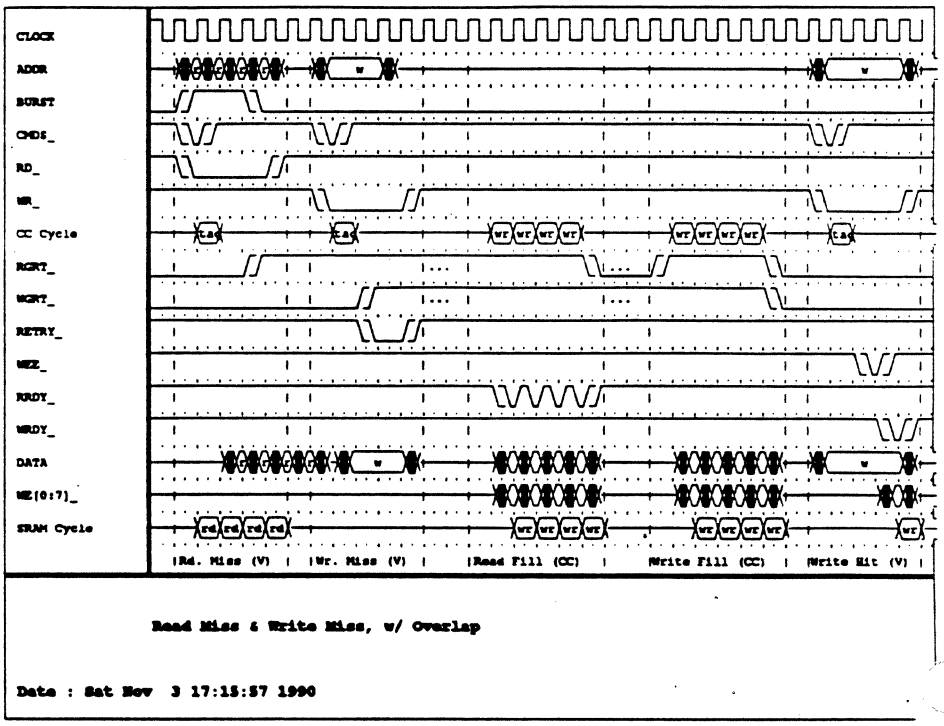


After a read miss, the cache controller negates RGRT_ to indicate the miss. WGRT_ remains asserted, allowing *Viking* to initiate write transactions. Although read misses generally stall the processor pipeline, *Viking's* store buffer is unaffected, and may issue stores. This example shows one such store, which hits in the cache. Once the cache controller receives the read miss data, it disallows further writes by finally negating WGRT_. With both RGRT_ and WGRT_ negated, the cache controller completes the read miss, writing the data into the cache SRAMs and passing it onto *Viking* by asserting RRDY_ for each doubleword. The controller then asserts both RGRT_ and WGRT_, allowing *Viking* to proceed normally.

9.3.5.4 Overlapped Read/Write Misses

The previous examples described cache hits that can occur while the external cache controller is processing a cache miss; specifically, read hits during a write miss, and write hits during a read miss. These overlapped "hits" could also result in cache misses, which must be properly serviced. Since reads and writes are handled independently, the protocol is not altered if both accesses generate cache misses. The following two examples show a read miss followed by a write miss, then the reverse.

Figure 9-29 *Overlapped Read/Write Miss*

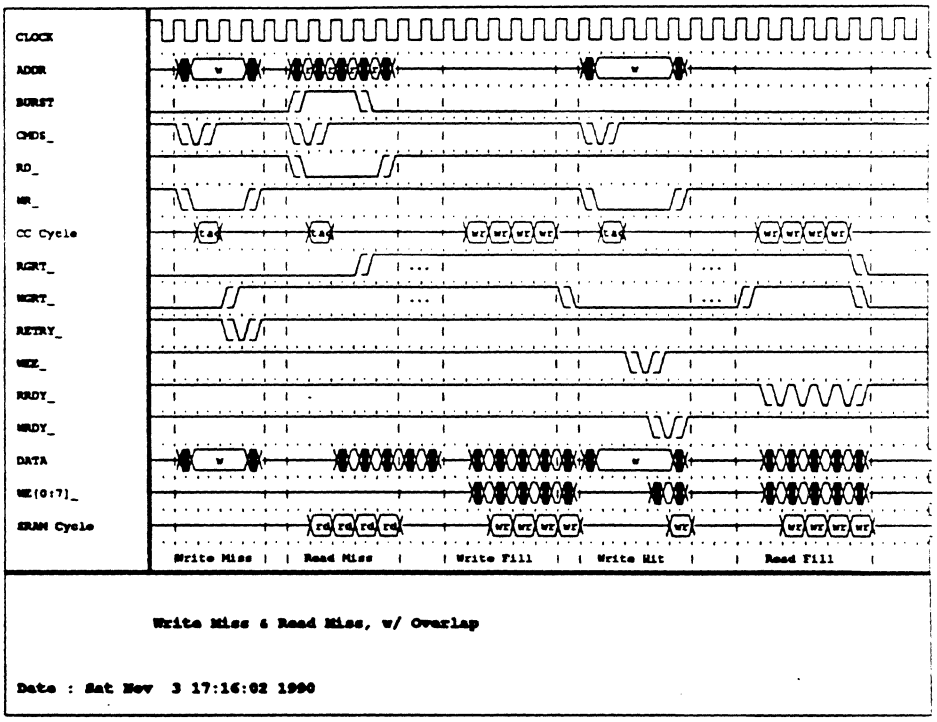


This example shows *Viking* issuing a read miss, then attempting to copy out a store buffer entry while the read was being processed, as before. After checking tags for the read, the controller negates RGRT_ but leaves WGRT_ asserted. This allows writes to be issued while the read miss is taking place. The write also misses, forcing the cache controller to negate WGRT_, as for a normal write miss. All further *Viking* accesses are blocked at this point.

Eventually read miss data is returned from memory to the cache controller. As the controller writes data to the cache SRAMs, it passes it onto *Viking* by asserting RRDY_. At this point RGRT_ is reasserted, unblocking further *Viking* reads. Later, when the write miss data arrives from memory, the cache controller retakes the bus by negating both RGRT_ and WGRT_, and writes that line into the cache. Finally, the cache controller asserts both RGRT_ and WGRT_, allowing *Viking* to perform reads and writes as it desires. In this example, *Viking* chose to retry the write immediately, which completed normally.

The example can be reversed. *Viking* can generate a read miss while a write miss is already in progress. This case is handled the same way, and is shown in detail below.

Figure 9-30 *Overlapped Write/Read Miss*



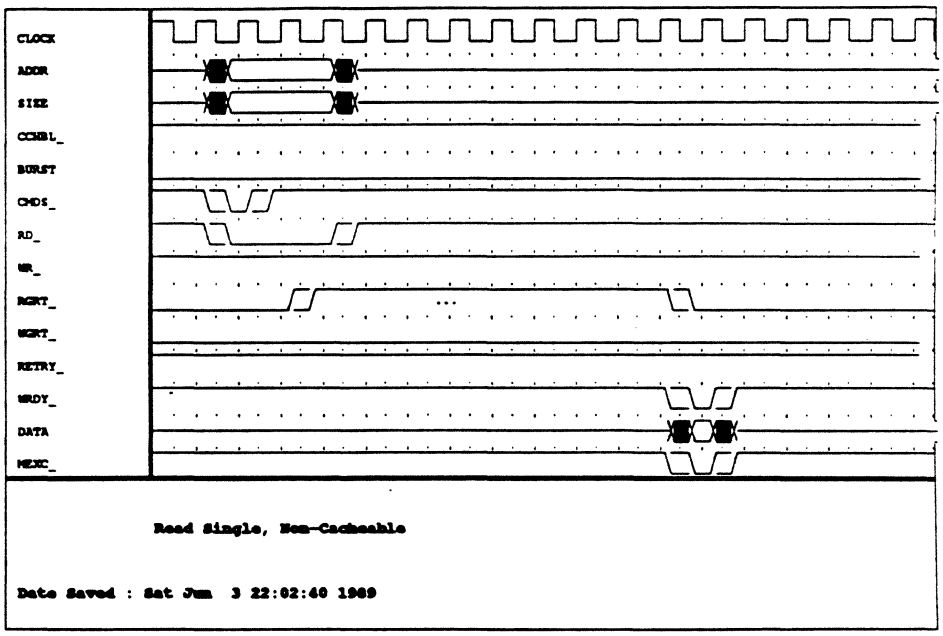
Note that the read and write misses can be serviced out of order. Even if the read miss happened first, it is possible that the write miss line could have returned before the read does. This ordering is completely legal. *Viking* simply proceeds with whatever access the cache controller allows, as signaled by the RGRT_ and WGRT_ lines.

9.3.5.5 I/O and Reads and Writes

Non-cacheable and I/O reads are similar to read misses for both *Viking* and the cache controller. They always initiate system bus transactions to transfer the needed data. The cache controller can differentiate between cacheable and non-cacheable reads by sampling the CCHBL_ pin for any transaction. The CCHBL_ signal is determined by the state of the C (cacheable) bit in the corresponding TLB entry, or by the MCNTL.AC (alternate cacheable) or MCNTL.TC (tablewalk cacheable) bit if no TLB entry applies. If CCHBL_ is asserted, the access is cacheable; otherwise, it is considered non-cacheable. During system reads the cache controller can allow *Viking* to perform cache writes by negating RGRT_ while keeping WGRT_ asserted. When the incoming data arrives, both RGRT_ and WGRT_ are negated, and the incoming data is driven onto the *Viking* bus, qualified with RRDY_. Only one "packet" of data is transferred, comprising one, two, four, or eight bytes. I/O errors are reported synchronously with RRDY_.

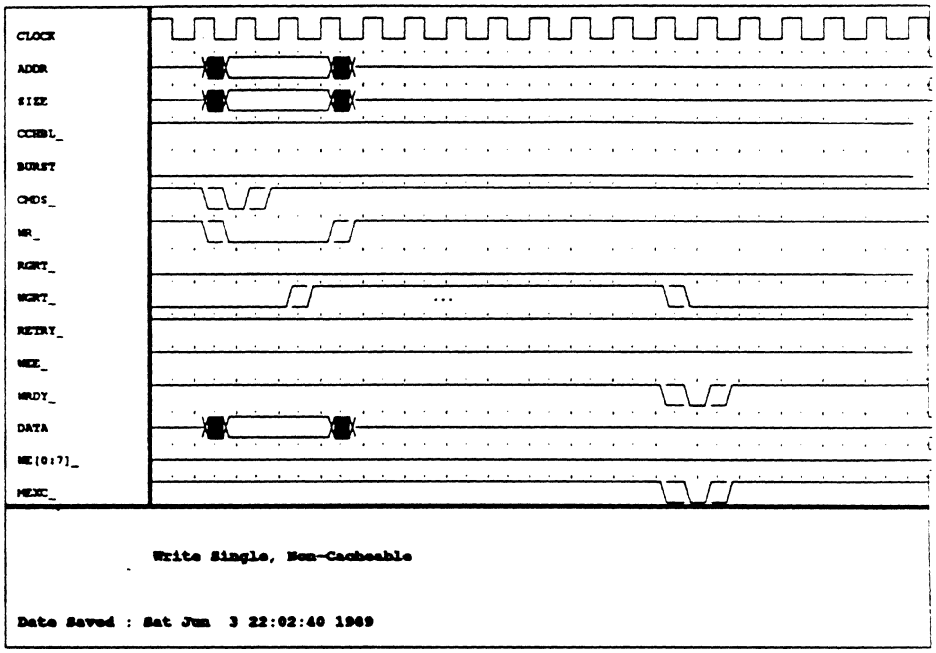
All non-cacheable references are single word transfers. Cache controllers may implement other caching policies than those indicated by the CCHBL_ pin.

Figure 9-31 *Noncacheable Read*



I/O and non-cacheable writes are handled differently than other write misses. When the cache controller sees a non-cacheable write (identified by CCHBL_ negated), it negates WGRT_, as for a write miss. This causes *Viking* to stop driving ADDR and DATA. The controller should not assert RETRY_, so *Viking* continues to wait for the write to be acknowledged, even though it is not driving the busses. This condition exists until the needed data is received, and acknowledged (although the I/O write may simply be placed the targets store buffer).

Figure 9-32 *Noncacheable Write*



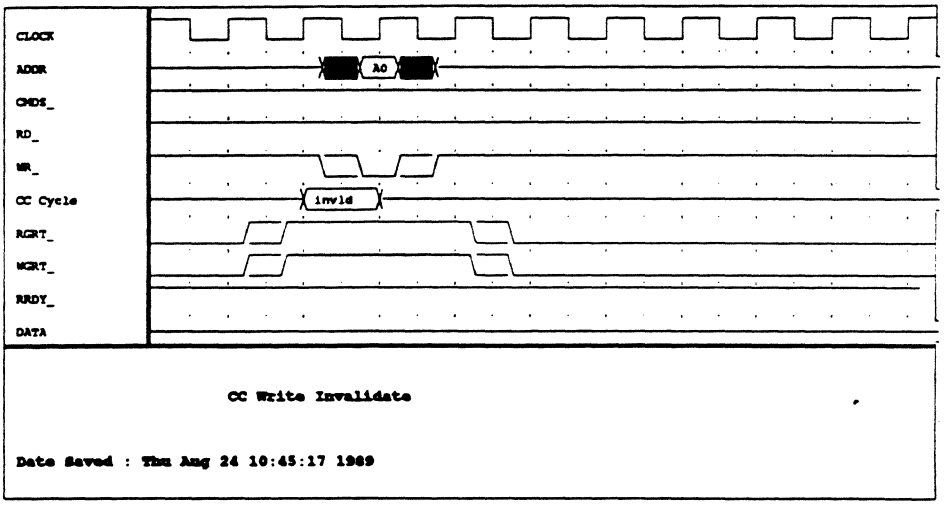
The cache controller negates WGRT_ during this time to allow it access to the cache, to satisfy system requests.

9.3.5.6 Cache Invalidations

Frequently, the cache controller must invalidate a block of data that is cached in *Viking's* internal caches. This can occur when the cache controller replaces a line in the external cache. To maintain cache inclusion, *Viking* must also remove this line. Invalidation is also required when the the cache line is updated due to another processor's write.

The invalidation protocol is very simple. When the cache controller is bus master, *Viking* snoops external bus traffic, using the same signals that it normally uses to perform accesses. The controller becomes bus master by negating RGRT_ and WGRT_. It then asserts CMDS_, and WR_, telling *Viking* to invalidate the cache line, if any match the current address on ADDR.

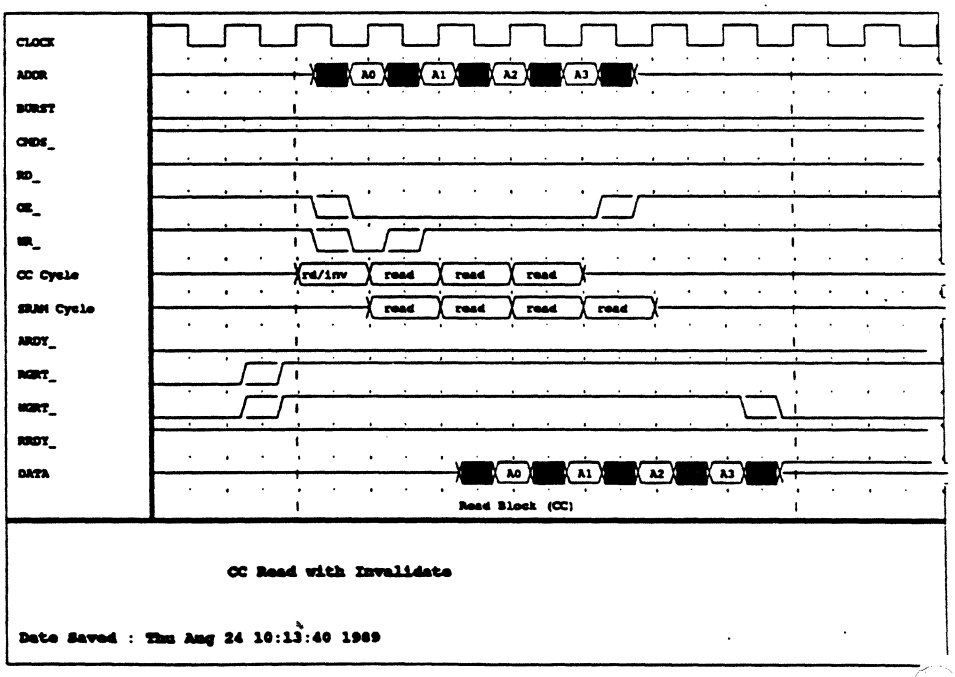
Figure 9-33 *Viking Cache Line Invalidation*



- Invalidations can be issued every other clock cycle.

In most cases, the cache controller can invalidate *Viking's* internal cache line at the same time it is updating the E-cache SRAM. For example, when the CC is copying out a modified line, it can also force *Viking* to invalidate the same line, as shown in the example below.

Figure 9-34 *Invalidation During Line Read*



Before proceeding, the cache controller obtains the bus from *Viking* by negating RGRT_ and WGRT_. Then, it reads a block from the SRAM much like *Viking*, by

driving ADDR and the SRAM output enable, OE_. The controller asserts WR_ to qualify the current ADDR value, telling *Viking* to invalidate the internal cache line that matches it, if any.

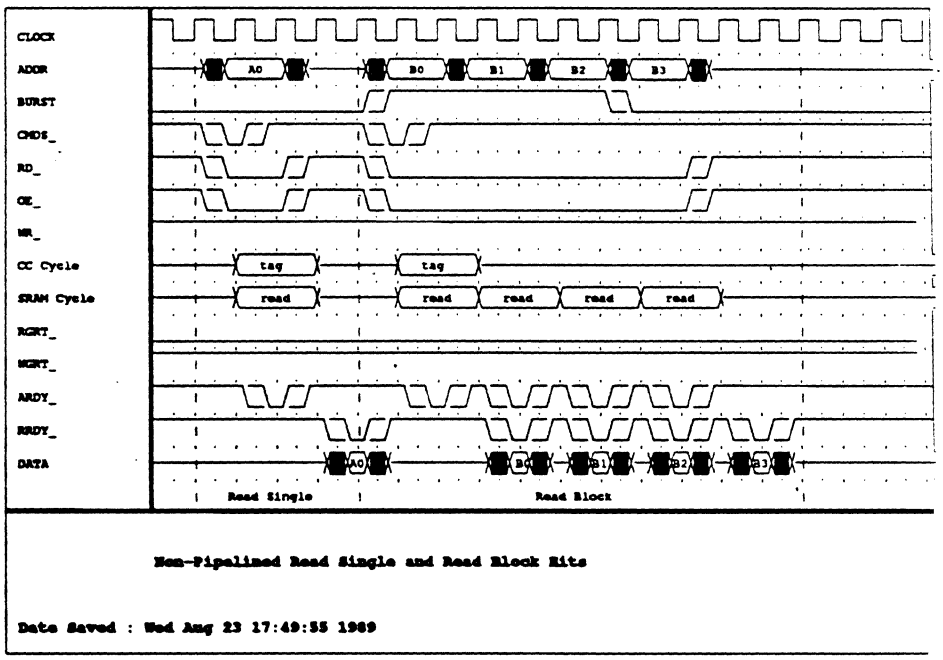
9.3.6. SRAM Timing Variations

Viking's protocol supports many variations of cache timing, in addition to the one-cycle pipelined SRAM described above. Two classes are discussed here; the first relaxes the internal SRAM access time. The second example shows how *Viking* would be interfaced to more traditional SRAM, which do not have address and data registers.

9.3.6.1 Accesses to Slower Pipelined SRAM

The most common type of pipelined SRAM performs actual memory accesses in one cycle. However, slower pipelined memory can be used. The following example increases the SRAM access time to two cycles, while maintaining the single cycle assertions of the standard interface.

Figure 9-35 *Slower Pipelined Reads*

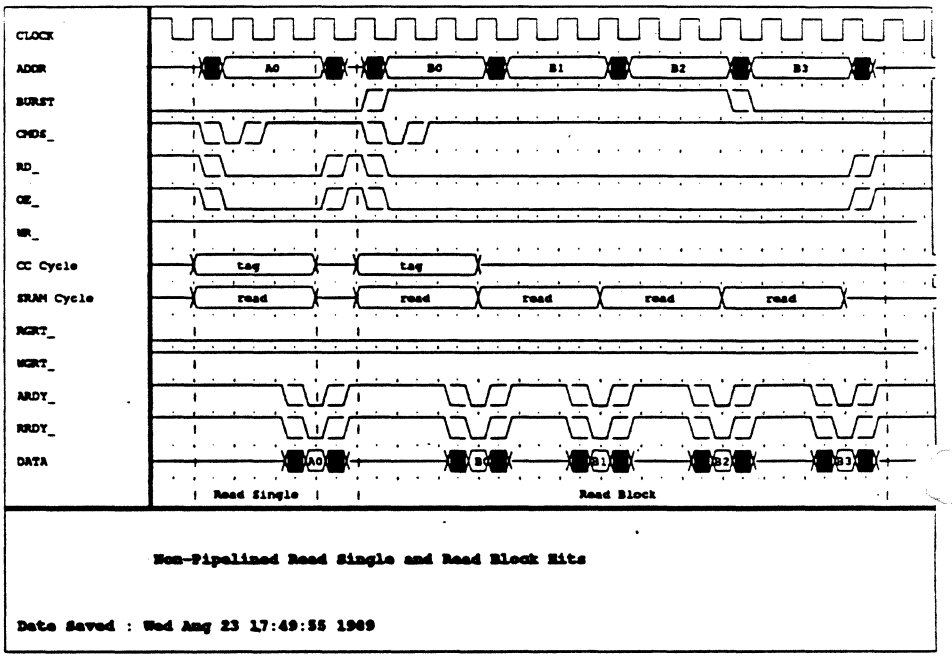


To slow down the rate at which *Viking* generates pipelined addresses, the cache controller must toggle ARDY_ rather than asserting constantly as before. Pipelined SRAMs generally have single registers for address (and data); thus, *Viking* must keep ADDR valid for two SRAM cycles. The controller accomplishes this by toggling ARDY_ as shown. *Viking* keeps driving its current address until ARDY_ is asserted. Similar timing is also needed for RRDY_, to qualify the returned data every other cycle.

9.3.6.2 Accesses to Nonpipelined SRAM

The use of non-pipelined SRAM for the cache usually implies lower performance. The SRAMs must be able to see the incoming address, read the internal memory array, and generate data, all before the next address can be asserted by *Viking*. This dramatically increases the time required to read a block from the cache. This variation is described here primarily to demonstrate the flexibility of the bus protocol.

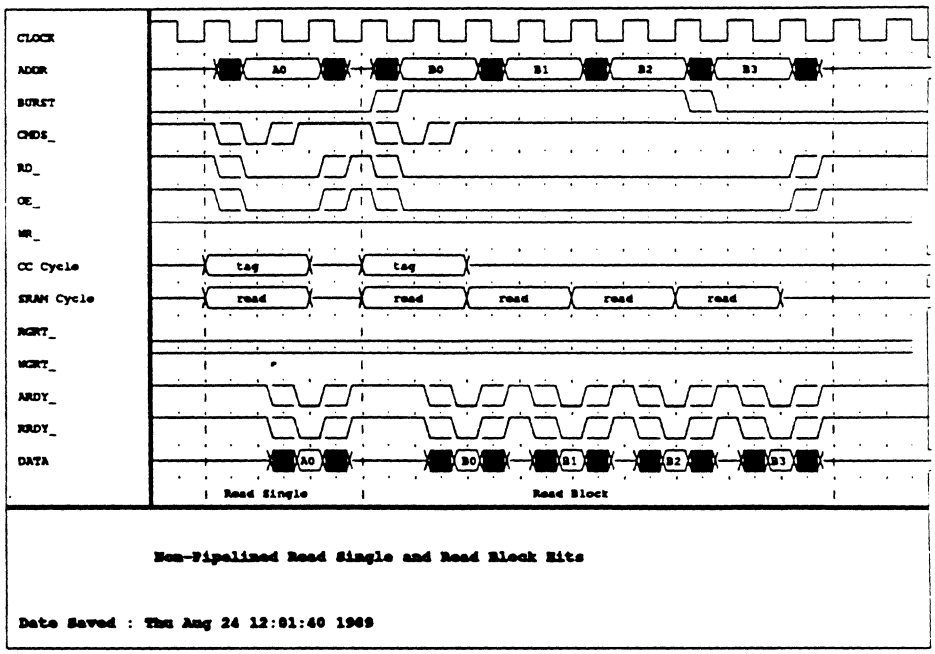
Figure 9-36 *3 Cycle Non-Pipelined Reads*



To inhibit address pipelining during reads, the cache controller must negate ARDY_ until a given read is complete. Then, ARDY_ is again asserted with RRDY_ as the SRAM data appears. Write timing is not as dramatically affected, since write addresses are generally not pipelined.

Higher speeds can be obtained with very fast SRAMs, and extremely careful system design. An example is shown below.

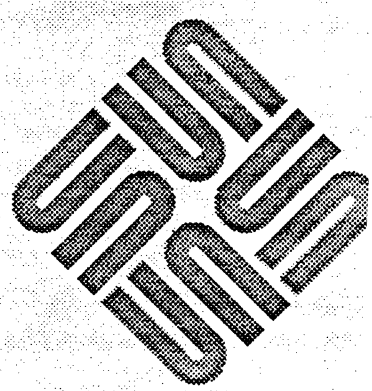
Figure 9-37 *2 Cycle Non-Pipelined Reads*



[Blank Page]

10

Signal Description



10

Signal Description

This section will describe the characteristics of *Viking's* pins.

10.1. Electrical Issues

Viking has been designed to operate in a transmission line environment. Care must be taken to carefully match the system physical design to *Viking's* I/O circuitry. I/O levels are defined in the TMS390Z50 data sheet (TI). In general, the circuitry is designed to provide controlled rise and fall times, as well as restricted voltage swing.

Viking assumes the existence of *holding drivers* on all bidirectional, or input only signals. These drivers must be provided externally, typically in an external cache controller, or in an MBUS device.

Internal pull-up resistors are provided on a number of signals, these are identified in the sections that follow.

SPICE simulations are recommended to guarantee proper operation at high frequencies.

Detailed signal timing is specified in the TMS390Z50 Data Sheet.

10.2. Pinout Descriptions

This section lists and describes all *Viking* pins, grouping them as bidirectional, input, and output signals. A summary table is provided at the end of the section.

Package pin outs are described in the TMS390Z50 Data Sheet.

All inputs are latched on rising clock edges. All outputs are sent out latched on rising edges, and are driven for a whole cycle. Setup and hold time requirements are different for each signal. Individual pin descriptions are given below.

10.2.1. Bidirectional Signals

10.2.1.1 ADDR[35:00]

As an input, this bus provides snoop addresses to *Viking*. When qualified by the proper signals (See *Viking* bus description), the address will be used to invalidate internally cached copies of data at the specified address.

As an output, *Viking* drives addresses onto this bus on all external reads and writes. ADDR[35:0] is sent out registered on rising clock edges, and is valid from one clock→output time after the clock edge, until one hold time after the next

rising clock edge. The signal is designed to meet the set up time requirements of the *MXCC* cache controller and pipelined SRAM. The bus is tristated one clock→disable time after the rising edge. One unused transfer cycle is required to change direction, and owner of the bus.

In MBUS mode, the address bus in largely unused, and should not be connected. The least significant three bits are used to sample the module identification value from the MBUS.

These pins are pulled to a logic one state with weak internal resistive pullups in MBUS mode only.

10.2.1.2 DATA[63:0]

On reads, this bus is an input, and synchronously supplies data to *Viking*. On writes, data is output synchronously to the system. DATA[63:0] is sent out registered on rising clock edges, and is valid from one clock→output time after the clock edge, until one hold time after the next rising clock edge. The signal is designed to meet the set up time requirements of the *MXCC* cache controller and pipelined SRAM. The bus is tristated one clock→disable time after the rising edge. One unused transfer cycle is required to change direction, and owner of the bus.

In MBUS mode, this pin functions as the multiplexed address and data bus, MAD[63:0].

Byte ordering is always *big-endian*, byte 0 in memory will be stored on bits DATA[63:56].

10.2.1.3 DEMAP_

This pin is an input whenever *Viking* is not the bus master. As an input, this pin is asserted along with CMDS_ signal, to indicate an incoming demap request. This signal qualifies the input value on the DATA[63:0] bus as demap request information.

This pin is pulled inactive with a weak internal resistive pullup in MBUS mode only.

10.2.1.4 DPAR[0:7]

Data bus parity. These pins carry parity in both directions. As an input, this bus is used to check data parity on *Viking* reads. As an output, *Viking* drives generated parity onto this bus whenever it drives data.

Parity checking and generation is controlled by the MCNTL.PE bit. When parity is disabled, *Viking* generates *incorrect* parity, to allow simple parity bit diagnostic testing.

Parity bit ordering corresponds to the *big-endian* convention used throughout *Viking*. Parity bit 0 corresponds to memory byte zero, which is stored on data bus bits [63:56].

10.2.1.5 OWNER_ (MIH_)

This signal is used in MBUS mode only, and corresponds to the MIH_ signal. *Viking* drives this signal when it is the owner of a cache line currently being requested on the bus with a CR or CRI transaction. It is used to inhibit memory from responding to the pending transaction.

As an input, *Viking* samples this pin while performing a CR or CRI transaction on the MBUS. It is used to qualify any MRDY_ responses that may come at the same time, or several cycles after another cache responds with MIH_.

This pin is pulled inactive with a weak internal resistive pullup in *Viking* bus mode only.

10.2.1.6 RD_

As an output, *Viking* drives RD_ to qualify all valid addresses on the bus as read cycles. It is also asserted, along with WR_ for swaps, as well as along with DEMAP_ for external demap replies.

This signal is used as an input *only* for internal SRAM test mode. It is not used in MBUS mode.

This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.1.7 SHARED_ (MSH_)

This signal is used in MBUS mode only, and corresponds to the MSH_ signal. *Viking* drives this signal when it has a copy of any cache line currently being requested on the MBUS. It is used to inform other caches that the information is shared.

Viking samples this signal when it is requesting an MBUS transaction. If the signal is active, the data received will be considered shared in the cache (which prevents writes until other copies are invalidated).

This pin is pulled inactive with a weak internal resistive pullup in *Viking* bus mode only.

10.2.1.8 WR_

As an output, *Viking* drives WR_ to qualify all valid addresses on the bus as write cycles. It is also asserted, along with RD_ for swaps, as well as along with DEMAP_ for demap requests.

WR_ is used as an input to qualify external invalidation requests. If WR_ is asserted, along with CMDS_, by another bus master, the address on the bus will be used as an invalidation request address.

WR_ is not used in MBUS mode, and is used in SRAM test mode to qualify external write requests.

This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.1.9 ARDY_ (and MAS_)

This signal is used as an input in *Viking* bus mode, to indicate that system logic is prepared to accept another address or bus cycle.

In MBUS mode, the signal is used as both an input and output, connected to the MAS_ (Address strobe) signal. When *Viking* is granted access to the MBUS, it may initiate bus cycles by asserting the MAS_ signal. When other processors own the bus, *Viking* samples the MAS_ signal to determine the beginning of bus cycles which it must snoop.

10.2.1.10 WEE_ (MBB_)

This signal is used as an input only in *Viking* bus mode. It is used to control assertion of the WE[7:0]_ signals to external cache RAM.

In MBUS mode, it is connected the the MBB_ (MBUS busy) signal, and used as both input and output. *Viking* drives this signal while it retains MBUS ownership, and samples the pin to determine when other bus masters have released their use of the bus.

10.2.1.11 WRDY_ (MRDY_)

In *Viking* bus mode, this signal is used as an input only, indicating completion of the current write transaction on the bus. It may be connected in parallel with the RRDY_ signal, if only a single global data ready signal is required.

In MBUS mode, this pin serves as the MRDY_ signal, and is both an input and output. It is an input when *Viking* has a current bus cycle pending; an output when *Viking* is responding with data that it *owns* for a CR or CRI transaction..

10.2.1.12 CMDS_

This signal is only used in *Viking* bus mode. It indicates the beginning of a *Viking* bus transaction. It is an output when *Viking* is bus master to initiate transactions.

When *Viking* is not the bus master, as indicated by *both* WGRT_ and RGRT_ being deasserted, CMDS_ is used to initiate all external snoop transactions, including invalidates and demaps.

This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.1.13 OE_

This signal is used as an I/O in *Viking* bus mode only. Generally, it is used as an output to control the pipelined output enable of external cache SRAM. As an input, it is sampled to prevent certain bus collisions. See section 9.2.2 — Arbitration for complete details.

This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.2. Input Signals

10.2.2.1 CCRDY_

This input tells *Viking* whether to operate in native *Viking* bus mode (CC mode), or to operate in MBUS mode. The signal must be statically asserted, and not changed during normal operation.

When CCRDY_ is asserted, *Viking* will use its native bus protocol, deasserted will select MBUS mode.

This signal is pulled inactive with a weak internal resistive pullup.

10.2.2.2 IRL[3:0]

Interrupt Request Level. This field tells *Viking* the level of the highest priority interrupt request that is currently pending. If IRL[3:0]=0000, no interrupts are pending. The highest priority is IRL[3:0]=1111, which is a non-maskable (except by PSR.ET) interrupt. All other priorities are maskable, within the integer unit. Since the system could be asserting multiple interrupts, an external interrupt controller must select the highest priority interrupt, and drive its IRL onto these pins.

10.2.2.3 MEXC_

This signal is used exclusively as an input in both bus modes. This signal is encoded, along with the ready and retry signals to indicate the exact type of error response. See the MBUS specification, and section 9.2.3 — Error Reporting for full details.

10.2.2.4 PEND_

This signal is an input only, used generally in *Viking* bus mode only, but may be used on the MBUS mode as well. It is sampled, depending on the *memory model* in use to determine whether certain bus cycles may start or not. See 7.5 — Memory Model Support (PEND_) for complete details of this signals operation.

This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.2.5 RETRY_ (MRTY_)

This signal is used exclusively as an input in both bus modes. This signal is encoded, along with the ready and exception signals to indicate the exact type of error response. See the MBUS specification, and section 9.2.3 — Error Reporting for full details.

10.2.2.6 RESET_

Reset. This signal is asserted when an external reset request occurs, such as on power-up. The action that *Viking* takes in response to reset is defined in section 4.3 — Reset Operation, as well as section 7.3 — Reset Operation.

10.2.2.7 RGRT_

This signal tells *Viking* that it may perform a read on the system bus. This signal may be connected to the WGRT_ signal if only a single global bus grant signal is required. See section 9.2.2 — Arbitration for more details.

RGRT_ is unconnected in MBUS mode. This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.2.8 RRDY_

Read Data Ready, *Viking* bus mode only. This signal qualifies incoming read data. When RRDY_ is asserted, *Viking* will sample incoming data, on the same clock edge as RRDY_. This signal may be directly connected to the WRDY_ signal if only a single global data ready signal is required.

This pin is pulled inactive with a weak internal resistive pullup in MBUS bus mode only.

10.2.2.9 WGRT_ (MBG_)

In both bus modes, this pin is used exclusively as an input and grants bus ac to *Viking*. In *Viking* bus mode, it is used to allow write transactions on the b only. In MBUS mode, it is the only bus grant.

In *Viking* bus mode, it may be connected to the RGRT_ signal if only a single grant signal is required.

10.2.3. Output Signals

10.2.3.1 BURST

This signal is used in *Viking* bus mode only, as an output. It indicates that tl current address on the bus is part of a burst bus cycle. If this signal is active, there will be at least one additional transfer request in the burst.

For read bursts, there are exactly four transfers in every burst. The BURST sig will be asserted for the first three of them, and deasserted for the last to indic the end.

For write bursts, the number of transfers in a burst varies from two to infinit BURST will be asserted for all but the last transfer in a burst.

10.2.3.2 BUSREQ_ (MBR_)

This signal is used as an output only in both bus modes. It has the same mea in both modes, to request the bus for use by *Viking*.

10.2.3.3 CCHBL_

This signal is used as an output, and only in *Viking* bus mode. It indicates tl the current transaction is internally cacheable. Cacheability is determined b state of several bits internal to *Viking*. (See programmers model for full det on cacheability).

10.2.3.4 CSA_

CSA_ is an output only, and is used only in *Viking* bus mode. It indicates thi current bus cycle is a *control space* transaction. It is asserted for ASI transac to ASI space 0x02.

10.2.3.5 ERROR_ (AERR_)

This signal is an output only, and is used in both bus modes. When ERROR_ asserted, it indicates that *Viking* entered an *error mode* state, and will take a *watchdog reset* trap. See section 4.3 — Reset Operation for full details on i operation.

This signal is driven (not tri-stated) while in *Viking* bus mode. In MBUS mo since the AERR_ signal is common to all processor modules, it is only drive when asserted, and tri-stated otherwise.

The signal will remain asserted until the MFSR.EM (error mode) bit has been cleared.

10.2.3.6 LDST_

This signal is an output, and is only used in *Viking* bus mode. The signal is asserted when *Viking* is performing an atomic swap operation on the bus. It is equivalent to the logical OR of the RD_ and WR_ signals.

10.2.3.7 SIZE[1:0]

This signal is used in *Viking* bus mode, only as an output. It indicates the transfer size of the current bus transaction, encoded as follows:

Table 10-1 *SIZE[1:0] Encoding*

Value	Size
00	Byte
01	Halfword
10	Word
11	Doubleword

10.2.3.8 WE_[0:7]

These signals are used in *Viking* bus mode only to directly control the write enable signals of synchronous SRAM used for an external cache. They are output only. These signals are driven active to match the valid bytes during a bus write transaction, when the WEE_ (write enable enable) signal is asserted.

The signals are tri-stated whenever WEE_ is not asserted, to allow an external cache controller to to control the cache RAM. This pin is pulled inactive with a weak internal resistive pullup.

WE_ bit ordering corresponds to the *big-endian* convention used throughout *Viking*. WE[0] corresponds to memory byte zero, which is stored on data bus bits [63:56]. WE[7] corresponds to memory byte seven, which is stored on data bus bits [07:00].

10.2.3.9 SU_

The SU_ pin is an output, used in *Viking* bus mode only. When asserted, it indicates that the current bus transaction is a supervisor transaction.

10.2.4. Clock Pins

The following pins define the clock interface to *Viking*.

10.2.4.1 VCK (CLK)

This pin provides *Viking* with its primary clock source. Full details on clock requirements are presented in section 7.2.2 — Input Clock Requirements.

In general, only the positive edge of this clock signal is significant. Duty cycle is not significant since the clock is internally multiplied, and divided based on the incoming rising edges.

10.2.4.2 VPLLRC

PLLRC Input. This pin is connected to an external 0.1μF capacitor connected to ground.

10.2.4.3 PLLBYP

This pin is an input only, used to bypass the internal PLL. When this pin is asserted, the external clock input will be routed directly to internal clock distribution, with no delay compensation. Operation in this mode is not recommended, or fully specified. It is intended primarily for debug and test purposes.

This pin is pulled inactive with a weak internal resistive pullup.

10.2.5. Test Pins

The following pins are used to simplify component and/or system testing. Many are related to JTAG-based scan testing. The PIPE signals are for observability.

Operation of the JTAG signals is described in the JTAG chapter.

10.2.5.1 TEST_

This signal is an input only, used for board level testing. When TEST_ is asserted, *Viking* de-asserts all its outputs except ESB and TDO.

This pin is pulled inactive with a weak internal resistive pullup.

10.2.5.2 TCK

JTAG test clock input. This pin is pulled to logic one with a weak internal resistive pullup.

10.2.5.3 TDI

Jtag test data input. This pin is pulled to logic one with a weak internal resistive pullup.

10.2.5.4 TDO

JTAG test data output. Also used to observe internal clock operation when the SEEPLL JTAG instruction is executed (non-standard JTAG). The signal is tri-stated under the control of the JTAG controller. The TEST_ pin will not cause TDO to tri-state.

10.2.5.5 TMS

JTAG test mode select input. This pin is pulled to logic one with a weak internal resistive pullup.

10.2.5.6 TRST_

JTAG test reset input. This pin is pulled inactive with a weak internal resistive pullup.

10.2.5.7 ESB

Execution strobe output. See section 4.14.5 — External Monitors. It is not tri-stated in response to the TEST_ signal.

10.2.5.8 PIPE[9:0]

Pipeline monitoring signals used improve internal operation observability. See section 4.14.6, and section 4.14.5 for a description of these signals.

10.2.5.9 SRMTST_

The SRMTST_ signal is an input only, used to initiate a special internal SRAM test mode for manufacturing purposes. It should be connected to VCC for normal operation. This pin is pulled inactive with a weak internal resistive pullup.

10.3. Pin Summary

Viking's pins are summarized below. All signals are latched either as inputs or outputs - latched outputs are valid for one cycle, minus a set-up time for the driver.

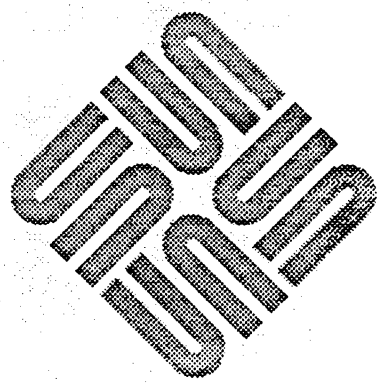
Table 10-2 *Viking Pin Summary*

Pin Type	Signal Name	Direction	Latch Edge	MBUS Pullup?	VBus Pullup?	MBUS Use	Active State
Busses [108]	ADDR[35:0]	I/O	Rising	Yes	No	[3:0] are MID	High
	DATA[63:0]	I/O	Rising	No	No	MAD[63:0]	High
	DPAR[0:7]	I/O	Rising	No	No	n/c	High
Data Size [3]	BURST	O	Rising	No	No	n/c	High
	SIZE[1:0]	O	Rising	No	No	n/c	High
Request Strobes [3]	RGRT_	I	Rising	Yes	No	n/c	Low
	WGRT_	I	Rising	No	No	MBG_	Low
	BUSREQ_	O	Rising	No	No	MBR_	Low
Access Strobes [13]	ARDY_	I/O	Rising	No	No	MAS_	Low
	CCRDY_	I	Rising	Yes	Yes	vcc	Low
	RRDY_	I	Rising	Yes	No	vcc	Low
	WRDY_	I/O	Rising	No	No	MRDY_	Low
	CCHBL_	O	Rising	No	No	n/c	Low
	CMDS_	I/O	Rising	Yes	No	n/c	Low
	CSA_	O	Rising	No	No	n/c	Low
	LDST_	O	Rising	No	No	n/c	Low
	SU_	O	Rising	No	No	n/c	Low
	DEMAP_	I/O	Rising	Yes	No	n/c	Low
	RD_	I/O	Rising	Yes	No	n/c	Low
	WEE_	I/O	Rising	No	No	MBB_	Low
	WR_	I/O	Rising	Yes	No	n/c	Low
Exception Signals [9]	IRL[3:0]	I	Rising	No	No	IRL[3:0]	High
	MEXC_	I	Rising	No	No	MERR_	Low
	PEND_	I	Rising	Yes	No	PEND_	Low
	RESET_	I	Rising	No	No	RSTIN_	Low
	RETRY_	I	Rising	No	No	MRTY_	Low
	ERROR_	O	Rising	No	No	AERR_	Low
Clock	VCK	I	-	No	No	CLK	-
	VPLLRC	I (analog)	-	No	No	VPLLRC	-
	PLLBYP_	I	Rising	Yes	Yes	PLLBYP_	High
Cache [9]	OE_	I/O	Rising	Yes	No	n/c	Low
	WE_[0:7]	O	Rising	Yes	Yes	n/c	Low
Copyback [2]	OWNER_	I/O	Rising	No	Yes	MIH_	Low
	SHARED_	I/O	Rising	No	Yes	MSH_	Low
JTAG Signals [6]	TCK	I	Rising	Yes	Yes	TCK	High
	TDI	I	Rising	Yes	Yes	TDI	High
	TMS	I	Rising	Yes	Yes	TMS	High
	TRST_	I	Rising	Yes	Yes	TRST_	High
	ESB	I	Rising	No	No	ESB	High
	TDO	O	Rising	No	No	TDO	High
Test Signals [12]	SRMTST_	I	Rising	Yes	Yes	SRMTST_	High
	TEST_	I	Rising	Yes	Yes	TEST_	High
	PIPE[9:0]	O	Rising	No	No	PIPE[9:0]	High

[Blank Page]

11

Electrical and Mechanical Specification



[Blank Page]

11

Electrical and Mechanical Specification

This information will be provided separately in data sheet form.

[Blank Page]

Index

The Viking Microprocessor Errata Document

Last update: 2/20/91

Reference: The Viking User Documentation rev 2.00 Nov 1 1990
(Sun P/N 800-4510-02)

Contents:

1.0 Summary

Date	Item	Fix?	How Fixed	Future Plans/Comments
Jan27	Doc Typos	Yes, next doc	Doc fixed	
Jan27	Doc Clarify	Yes, next doc	Doc fixed	
Jan27	Doc XRef	Yes, next doc	Doc fixed	
Feb03	Fbfcc no SeqErr	No, permanent	Doc added	Doc FPX Section 4.6.4
Feb07	Iax vs Fpx Order	No, permanent	Doc added	Doc Traps 4.13

2.0 Errata List

Reference	Description/Correction
Errata Class Page 19	Doc Typo (Section 2.3.2 The last sentence at the bottom of the page) "by its For simplicity, other instructions in the pipeline are not shown." Should be: "For simplicity, other instructions in the pipeline are not shown."
Errata Class Page 22	Doc Typo (Section 2.3.5, page 22, Table 2-2) The performance numbers for some instructions are quoted wrong. The table listed below is complete and correct, the changed rows are indicated with *. Should be: Table 2-2 Floating Point Operation Execution Time - Latency

Instruction	nn=n	nn=s	sn=n	sn=s	ns=n	ns=s	ss=n	ss=s	
fdivd	9	10	12	13	13	-	13	-	*
fdivs	6	7	9	10	10	-	10	-	
fmuld	3	4	6	7	7	8	8	-	
fmuls	3	4	6	7	7	8	8	-	
fsqrtd	12	-	15	-	-	-	-	-	*
fsqrts	8	-	11	-	-	-	-	-	
fsmuld	3	-	6	-	7	-	8	-	
idiv	18	-	-	-	-	-	-	-	*
idivcc	18	-	-	-	-	-	-	-	*
imul	4	-	-	-	-	-	-	-	
imulcc	4	-	-	-	-	-	-	-	

Errata Class Page 36	Doc Typo (Section 3.3.3, Floating point register dependency, the first example on the page, titled Bad Example, states PIPELINE STALL FOR 4 CYCLES ! Should be: PIPELINE STALL FOR 2 CYCLES !
Errata Class Page 54	Doc Clarify (Section 4.3.2 Watchdog Reset, 2nd to last paragraph, discussion on Viking behavior during error mode)

Should add:
In CC mode, when error mode is encountered, Viking asserts ERROR_ for one cycle, allowing the cache controller (or external system logic) to record the occurence of error mode. The completion of this bus cycle causes watchdog reset.
In MBUS mode, Viking asserts AERR_ until MFSR.EM is cleared. For example, a read on MFSR clears it. This is then followed by a watchdog reset.

Errata Class: Doc Clarify
Page 62
(Section 4.5.3 Load and Store Alternates)
The 2nd paragraph discusses about which ASIs do not flush the store buffer before starting execution.
Should clarify:
Nearly all ASI operations cause a store buffer copy-out (both LDA and STA) before starting execution, except:

ASI	Operation
0x20-0x2F	STA
0x40-0x4C	LDA/STA
0x8,0x9,0xA,0XB	STA
0x8,0x9,0xA,0XB	Cacheable LDA

Errata Class: Doc Clarify
Page 69
(Section 4.7.2 Instruction Cache Replacement Policy)
Whenever a memory reference hits in the cache, the history bit is written to one.
Should be:
Whenever an instruction fetch hits in the cache, the history bit is written to one.

Errata Class: Doc Clarify
Page 75
(Section 4.8.3 Data Cache Replacement Policy)
Whenever a memory reference hits in the cache, the history bit is written to one.
Should be:
Whenever a LD instruction hits in the cache, the history bit is written to one.

Errata Class: Doc Typo
Page 75
(Section 4.8.2 The 2nd sentence in the 1st paragraph after Table 4-7)
"When the cache is enabled, the C bit from the MMU ..."
Should be:
"When the cache is enabled, the C bit from the PTE ..."

Errata Class: Doc Clarify
Page 95
(Section 4.11.10, discussion about the exceptions that are not disabled by setting MCNTL.NF)
Should add:
Unassigned ASIs will trap regardless of MCNTL.NF

Errata Class: Doc Clarify
Page 97
(Section 4.11.11.1 Description of MCNTL.PE bit)
Should add:
When set, even parity is generated by Viking. When zero, odd parity is generated.

Errata Class: Doc Typo
Page 104
(Section 4.11.11.3 Table 4-12 "Access Permissions vs Access Type")
had errors.
Should be:

AT	PTE.V=0	PTE.V=1,PTE.ACC= 0	1	2	3	4	5	6	7

0	1	-	-	-	-	2	-	3	3
1	1	-	-	-	-	2	-	-	-
2	1	2	2	-	-	-	2	3	3
3	1	2	2	-	-	-	2	-	-
4	1	2	-	2	-	2	2	3	3
5	1	2	-	2	-	2	-	2	-
6	1	2	2	2	-	2	2	3	3
7	1	2	2	2	-	2	2	2	-

Errata Class Doc XRef
Page 109 (Section 4.12.1 General Operation of the Store Buffer)
Should add:
Reference to Section 9.2.4.5 Burst Writes, and index entries added.

Errata Class Doc Clarify
Page 110 (Section 4.12.5 Store Buffer Exceptions)
Should add:
In CC mode, during a store buffer exception, Viking asserts ERROR_ for one cycle. In MBUS mode, during a store buffer exception, Viking asserts AERR_ until MFSR.SB is cleared. For example, reading MFSR clears it.

Errata Class Doc Typo
Page 110 (Section 4.12.5 Store Buffer Exceptions)
(1st paragraph, last sentence)
... and the MFSR.NF bit should be deasserted
Should be:
... and the MCNTL.NF bit should be deasserted

Errata Class Doc Typo
Page 114 (Section 4.13, 1st paragraph on page, 2nd sentence)
Viking can executes ...
Should be:
Viking can execute ...

Errata Class Doc Clarify
Page 123 (Table 4-16 Breakpoints - Crontrol and Status)
Should add:
When writing into the control registers BKC, ACTION, CTRC, care must be taken to avoid overwriting unintended fields. It is best to exercise a read-modify-write programming sequence to overwrite only the intended bits and not disturb the other bits/fields.

Errata Class Doc Typo
Page 127 (Section 4.14.4.1 Description lines of register BKS)
"CBKIS Indicates that an interrupt was generated"
Should be:
"CBKIS Indicates that an interrupt was taken"

Errata Class Doc Typo
Page 127 (Section 4.14.4.1 Description lines of register BKS)
"CBKFS Indicates that an interrupt was generated "
Should be:
"CBKFS Indicates that an interrupt was taken "

Errata Class Doc Typo
Page 127 (Section 4.14.4.1 Description lines of register BKS)
"DBKIS Indicates that an interrupt was generated "
Should be:
"DBKIS Indicates that an interrupt was taken "

```
Errata Class    Doc Typo
Page 127        (Section 4.14.4.1 Description lines of register BKS)
                "DBKFS Indicates that an interrupt was generated "
                Should be:
                "DBKFS Indicates that an interrupt was taken "
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 128        (Section 4.14.4.4 Description lines of register CTRS)
                "ZICIS Records the status ... was generated ..."
                Should be:
                "ZICIS Records the status ... was taken ..."
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 128        (Section 4.14.4.4 Description lines of register CTRS)
                "ZCCIS Records the status ... was generated ..."
                Should be:
                "ZCCIS Records the status ... was taken ..."
-------------------------------------------------------------------------
Errata Class    Doc Clarify
Page 191        (Section 8.1 MBUS Compatibility)
                Should add:
                Virtual address 19 through 12 (multiplexed on MAD[53:46])
                is used to carry virtual address bits 19 through 12 of a
                read or write block transfer for virtually indexed caches,
                also known as the "superset" bits.  In compliance with the
                MBUS Specification, Viking drives these virtual address
                superset bits (VA[19:12] or MAD[53:46]) high.
-------------------------------------------------------------------------
Errata Class    Doc Clarify
Page 191        (Section 8.1 MBUS Compatibility)
                Should add:
                This bit is the supervisor access indicator bit, multiplexed
                on MAD[59].  For MBUS mode operation, during user-mode
                operation, it is recommended to operate with the store buffer
                enabled.  When the store buffer is enabled, Viking drives
                MAD.SUP high, compliant with the MBUS Specification.
                When the store buffer is disabled, however, the MAD.SUP bit
                is not driven high.  Instead, during copyback operations,
                MAD.SUP bit will be obtained from the state of the current
                transaction which caused the copyback to occur.
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 192        (Section 8.5 the last sentence the in first paragraph)
                "This protocol is described int he MBUS ..."
                Should be
                "This protocol is described in the MBUS ..."
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 193        (Figure 8-1, arc from "Clean Exclusive" to "Owned Exclusive")
                "Vik Wr. (Bus CI)"
                Should be:
                "Vik Wr."
-------------------------------------------------------------------------
Errata Class    Doc Clarify
Page 199        (Section 8.10 Port Register, the 2nd sentence in the last
                paragraph on the page)
                This indicates device 0, revision 0 for the vendor (Texas
                Instruments).
                Should be:
                This indicates device 0, revision 0, and ID 4 for the SPARC
                Licensee (Texas Instruments).
-------------------------------------------------------------------------
Errata Class    Doc Clarify
Page 199        (Section 8.10 Port Register, MBUS Interface)
```

```
                Should add:
                This MBUS port register is read-only.
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 238        (Figure 9-31 Noncacheable Read)
                label WRDY_
                Should be
                label RRDY_
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 240        (Figure 9-33 and 9-34)
                CMDS_ was not asserted
                Should be:
                CMDS_ should be asserted at the same time as WR_
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 255        (Section 10.3 Viking Pin Summary, Table 10-2)
                (Signal Name)   (MBUS pullup)   (VBus pullup)
                DPAR[0:7]       No              No
                WE_[0:7]        Yes             Yes
                The correct entries should be:
                DPAR[0:7]       Yes             No
                WE_[0:7]        No              No
-------------------------------------------------------------------------
Errata Class    Doc Typo
Page 255        (Section 10.3 Viking Pin Summary, Table 10-2)
                (ESB was listed as Input instead of Output)
                The correct entry should be:
                (Signal Name)   (Direction)
                ESB             O
-------------------------------------------------------------------------
```

3.0 Known Design Bugs

```
=========================================================================
Reference       Description
=========================================================================
Errata Class    Known Design Bug (Fbfcc in fpx does not cause Sequence Error)
Page 67         (Section 4.6.4 Floating Point Exception Details)
                The last paragraph discusses how a sequence error will be
                triggered if a new FP request is made when the FPU is in
                exception mode.
                Should add:
                There is a special case of this floating point request
                which does not trigger a sequence_error, and that is the
                FBFCC instruction.  When an FBFCC instruction is issued
                while the FPU is in exception_mode, Viking does not cause
                a sequence_error.
-------------------------------------------------------------------------
Errata Class    Known Design Bug (aix vs fpx order of traps taken)
PN2860 2/7/91   (Section 4.13 Traps, page 114, should add the following)
                Suppose we have the following instruction group:
                ------ break group ------
                fdiv    %f0,%f0,%f0     (fpop, generates fpx)
                st      %f0,[%l0]       (stf, generates iax, reports fpx)
                ------ break group ------
                where
                fpx = fp_exception, priority=11, tt=0x08
                iax = instruction_access_exception, priority=5, tt=0x01
                Scenario:
                The FPOP generates fpx
                The STF generates iax, and accepts fpx generated by FPOP
                There must exist a dependency between the fpop and stf.
```

S

T

V

W

[Blank Page]