```
  SSSSSSSS    PPPPPPPP      EEEEEEEEEE      CCCCCCCC
  SSSSSSSS    PPPPPPPP      EEEEEEEEEE      CCCCCCCC
SS            PP      PP    EE            CC
SS            PP      PP    EE            CC
SS            PP      PP    EE            CC
SS            PP      PP    EE            CC
  SSSSSS      PPPPPPPP      EEEEEEEE      CC
  SSSSSS      PPPPPPPP      EEEEEEEE      CC
        SS    PP            EE            CC
        SS    PP            EE            CC
        SS    PP            EE            CC
        SS    PP            EE            CC
SSSSSSSS      PP            EEEEEEEEEE      CCCCCCCC
SSSSSSSS      PP            EEEEEEEEEE      CCCCCCCC



LL            PPPPPPPP      TTTTTTTTTT                  333333
LL            PPPPPPPP      TTTTTTTTTT                  333333
LL            PP      PP        TT                  33      33
LL            PP      PP        TT                  33      33
LL            PP      PP        TT                          33
LL            PP      PP        TT                          33
LL            PPPPPPPP          TT                      33
LL            PPPPPPPP          TT                      33
LL            PP                TT                        33
LL            PP                TT                        33
LL            PP                TT            ....  33      33
LL            PP                TT            ....  33      33
LLLLLLLLLL    PP                TT            ....    333333
LLLLLLLLLL    PP                TT            ....    333333


*START* Job SPEC Req #695 for EGB    Date  3-Dec-82  2:00:44 Monitor: Rational M
File RM:<MICRO-ARCH.MEMORY>SPEC.LPT.3, created: 23-Nov-82 21:02:48
       printed:  3-Dec-82  2:00:44
Job parameters: Request created: 3-Dec-82  1:53:01   Page limit:171   Forms:NORM
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:
```

Functional Specification of the Memory Monitor

DRAFT 3

1. Summary

The R1000 memory system consists of from two to four memory boards and centralized control logic called the memory monitor. Each memory board has a capacity of two megabytes, implemented as four associative "sets" of 512 pages. Each board consists of a set associative tag store portion (where associative address translation and access control information is stored) and a parallel data array (where data is stored).

(4 boards * 4 sets * 512 pages * 1k Bytes = 8 Mbyte maximum storage)

The memory boards contain all the necessary logic to access, update, and maintain up to sixteen associative sets in parallel. The control logic which need not be duplicated on each memory board is implemented by the memory monitor. This logic resides on the FIU board except for the ERCC checker/generator and the "dummy" Read Data Register which are implemented on the Sysbus Interface board.

The memory monitor contains the microcode rams for the memory control fields, copies of various memory state registers (eliminating the need for each memory board to drive them out during state save), and the memory system control logic. The memory monitor also contains circuitry which tests all Control Stack Addresses to determine whether they point into the Control Stack Accelerator. If such a "CSA hit" occurs, the memory operation is redirected to the CSA (on the Value and Type boards). Finally, there is another address monitoring mechanism called the "scavenger monitor", which tests all collection addresses and traps if the addressed segment is potentially being garbage collected.

2. Functional Description

The functional description of the memory monitor begins by describing its role in managing the three registers defined in the memory interface: the Memory Address Register, the Read Data Register, and the Write Data Register. This section then describes the memory system's basic operations, followed by an overview of the memory mangement operations. This functional description concludes with a discussion of the address monitoring circuits: the Control Stack Accelerator Monitor and the Scavenger Monitor.

2.1. Address Bus

The ADDRESS BUS is a unidirectional bus for routing address information. It is split into two portions, the least significant 64 bits transfer the logical bit address while the most significant 3 bits transfer the space specification. The two portions are controlled separately.

The logical bit address portion is identical to the least significant 64 bits of the MAR (see the desriptions of those fields in the next section).

The driver of the address bus is determined by the memory monitor using the ADDRESS_BUS_SOURCE microorder of the FIU board, and the MAR_CONTROL microorder of the memory monitor. When the ADDRESS_BUS_SOURCE specifies SEQUENCER, both portions of the ADDRESS BUS are driven by the sequencer board. Otherwise, the space portion is driven by the memory monitor, while the address portion is driven by the selected source.

When the MAR_CONTROL microorder specifies RESTORE_MAR or RESTORE_MAR_WITH_REFRESH, the the space portion is sourced from the least significant 3 bits of the TYPE bus (along with the state flags). The SEQUENCER must not be specified as ADDRESS BUS source while RESTORE_MAR or RESTORE_MAR_WITH_REFRESH is specified. Also, in general, the TYPE BUS must be specified as TI BUS source on the FIU board.

For LOAD_MAR xxx CONTROL miroorders, the space portion is sourced from the space literal of the memory monitor. For INC_MAR, the space portion is driven from the current contents of MAR. For other MAR_CONTROL microorders, the source of the space portion of the ADDRESS BUS is undefined.

2.2. Memory Address Register

The memory monitor contains the only complete copy of the MAR. This copy of the MAR is the source of the Value and Type busses during a READ_MAR operation.

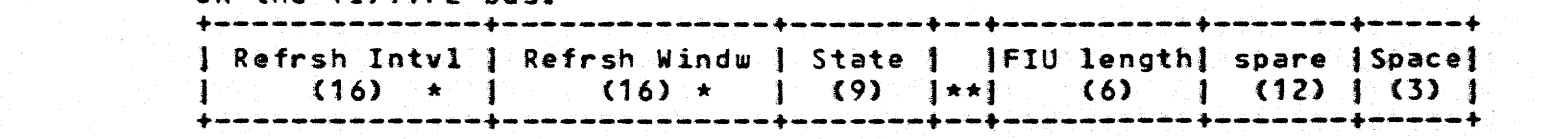
Each memory board contains a copy of the word address portion of the MAR, but these can only be read by the diagnostic processor. The fields of the complete MAR are enumerated below. The least significant 67 bits comprise the actual logical address, and are always loaded from the Address bus. The individual fields on the Address bus have separate parity bits. The most significant 61 bits contain several fields, admittably thrown together for the sake of state save efficiency. These bits are loaded from the TI_BUS on a RESTORE_MAR micro order (The MAR.SPACE field of the ADDRESS bus is driven from TYPE_BUS(61:63) during a RESTORE_MAR by the monitor).

The low order 67 bits of the MAR are always loaded from the address bus. When the FIU is selected as source for the address bus, the low order 67 bits of the MAR are driven over the address bus to the memory boards.

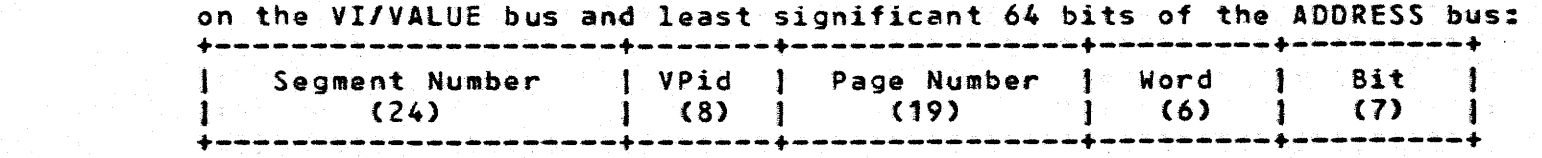
If a microevent aborts the cycle in which the MAR is loaded, the MARs on the memory boards are loaded, but not the MAR on the monitor. This inconsistency must be resolved by loading the MAR before memory is started.

MEMORY ADDRESS REGISTER
(format for READ_MAR and RESTORE_MAR microorders)

on the TI/TYPE bus:



on the VI/VALUE bus and least significant 64 bits of the ADDRESS bus:



* - Specified only for RESTORE_MAR_WITH_REFRESH.
returned by READ_MAR, ignored by RESTORE_MAR.
** - spare, must be zero.

2.2.1. Memory State Field (State)

The MAR in the memory monitor contains a nine-bit STATE field. These bits are saved and restored using TI_BUS(32:40); some of these bits are set by hardware and cleared as a side-effect of testing them. All are loaded from the TI bus by the RESTORE_MAR and RESTORE_MAR_WITH_REFRESH microorders. Briefly, the state flags are:

SCAVENGER_TRAP (TI_BUS<32>)
An address has been referenced which is specified to be trapped by the scavenger monitor. SCAVENGER_TRAP is testable in the second cycle following loading of the MAR (until then, the old value remains is returned). This bit is set in the MAR during cycle 2 of a memory reference when the test condition is true, and will cause a memory exception microevent. The MAR bit (and the memory exception) is cleared by testing this bit, but the test condition is only cleared by reloading the MAR or the scavenger ram contents. NOTE: if scavenger trap occurs during a write, data is written to memory. The RDR contains the original contents of the location; the handler may undo the write using the contents of the RDR.

CONTROL_ADDRESS_OUTOF_RANGE (TI_BUS<33>)
A Control Stack Address that is greater than the current Top of Control Stack was referenced. This bit is available as a medium late test condition during the second cycle following the loading of MAR or

CONTROL TOP (until then, the old value is returned). The value of the outof range condition is stored in the corresponding MAR flag bit during cycle 2 of a memory operation. It will generate a microevent during cycle 2 of a memory operation if the test condition is true, or the MAR flag bit was already set. If the start was a write, data are written unless CACHE MISS is also set. The MAR bit (and the microevent) is cleared by testing, but the test condition is cleared only when MAR or CONTROL TOP is loaded with an address that is in range.

PAGE_CROSSING (TI_BUS<34>)

Indicates that an INCREMENT_MAR operation incremented the word offset portion across a half-page boundary. This will cause a microevent, whose handler must add 4096 to the MAR (4096 is 32 words times 128 bits per word). This bit is set in the cycle following the INC_MAR, and cleared when tested.

CACHE_MISS (TI_BUS<35>)

The tag portion of a logical address did not match during a logical query, no invalid pages exist during an available query, no pages match the specified stack name during a name query or a logical write was attempted to a READ_ONLY page or any reference was attempted to a LOADING page. The CACHE_MISS condition is derived combinatorially from the last completed memory reference. During cycle 2 of a memory operation the condition is updated and latched in the MAR (until then, the result of the last memory operation is returned). Latching a true value into the CACHE_MISS MAR flag will cause a memory exception microevent to occur as soon as microevents are enabled. Testing the CACHE_MISS condition clears the corresponding MAR flag, but the condition is only changed by completing another memory operation (CACHE_MISS will only cause a memory exception event when the MAR flag is set).

FILL_MODE (TI_BUS<36>)

The FIU selected fill mode value is returned in this bit position. The latched fill mode value is always returned by READ_MAR. When RESTORE_MAR is specified, the microcode must explicitly latch the fill mode bit from the TI bus (see the FIU spec).

PHYSICAL_LAST (TI_BUS<37>)

Saves whether the last memory start was a physical reference or a logical reference. This is used by the ERCC error event handler for error logging and determining which type of reference to use when

writing back the corrected data. Set during cycle 2 of any START microorder which expects a frame address in the MAR; cleared during cycle 2 of all other START microorders.

WRITE_LAST (TI_BUS<38>)
Saves whether the last memory start was a read or a write; this bit turns the START_LAST_COMMAND and START_IF_INCOMPLETE microorders into START_READ or START_WRITE for logical or physical memory query (see PHYSICAL_LAST and INCOMPLETE_MEMORY_CYCLE). Set following a START microorder for logical or physical write, physical tag write, name query and LRU query. Cleared by a START microorder for logical or physical read, physical tag read, available query and tag query. Unmodified by IDLE, CONTINUE, START_LAST_COMMAND or START_IF_INCOMPLETE microorder. WRITE_LAST is set or cleared during cycle 1 of a memory start, and is testable as a conditions and readable in the MAR during cycle 2 of the memory start.

MAR_MODIFIED (TI_BUS<39>)
Indicates that a microevent occurred the cycle following an INCREMENT_MAR operation. Note the MAR will be modified during a conditional continue even if the continue does not occur. This bit must be queried by any event handler which needs to determine the address which caused the event (such as ERCC or page fault). This bit is set only on a microevent, and cleared when tested.

INCOMPLETE_MEMORY_CYCLE (TI_BUS<40>)
Indicates that a microevent has aborted cycle 1 of a memory cycle; if this bit is set, it turns the microevent return micro order (START_IF_INCOMPLETE) into a START_READ or a START_WRITE for logical or physical query, depending on the WRITE_LAST and PHYSICAL_LAST bits. If INCOMPLETE_MEMORY_CYCLE is set, and a memory cycle is in progress, a START_IF_INCOMPLETE is turned into a CONTINUE (this combination will occur when a page fault event occurs during a CONTINUE). The INCREMENT_MAR_IF_INCOMPLETE microorder must be specified in the same microinstruction for the CONTINUE to be properly initiated. This bit is cleared when a START_IF_INCOMPLETE micro order is specified.

spare (TI_BUS<41..42>,<49..60>) these bit is currently not used.

MEMORY_EXCEPTION
An exception occurred during a memory operation. This

will cause a microevent if not masked. This event is caused by SCAVENGER_TRAP, CONTROL_ADDRESS_OUTOF_RANGE or CACHE_MISS MAR flags being set or becoming set. The MEMORY_EXCEPTION event handler will query these bits to distinguish the type of fault. This bit does not exist separately and is not returned by the memory monitor; it is simply the OR of these MAR flags. It is testable by the microcode, but is reset only when all component MAR flags are not true. Note that, if the MAR is restored such that one or more of the memory exception components becomes set, a memory exception microevent will result, even though the testable conditions corresponding to these flags are not true. As always, testing the condition will clear the MAR flag.

MEMORY_EXCEPTION and its components are testable as medium late conditions. MEMORY_EXCEPTION generates a microevent in cycle 2 of the memory operation which caused the condition (see the discussion on Memory Operations), or in the cyle following the RESTORE_MAR which caused the MAR flag bit(s) to become set. The component flags are only set in the MAR when MEMORY EXCEPTION condition is true, and are visible in the MAR during the cycle following that in which MEMORY EXCEPTION becomes true (i.e., the third cycle following the memory start or the third cycle following the INC_MAR which caused the CONTROL_ADDRESS_OUTOF_RANGE).

Testing a MEMORY_EXCEPTION component during cycle 2 of a memory operation prevents that condition from being latched in the MAR, and prevents the MEMORY_EXCEPTION microevent.

2.2.2. Fill Mode (FM) and Length (FIU length) Fields

These fields are used to save and restore the fill mode (FM) and operand length state of the FIU using TI_BUS(36),(43:48), respectively. The FIU can also load thse fields from micro literals or from type descriptors. The only memory functions which will change these field are the RESTORE_MAR microorders (see FIU spec for details of field use). The fill mode and length registers returned by READ_MAR regardless of what might be selected by the current microinstruction. To properly restore FIU state from a saved MAR, the FIU must be instructed to latch fill mode and length from the TYP BUS (TI BUS) in the same microinstruction that specifies RESTORE_MAR or RESTORE_MAR_WITH_REFRESH.

2.2.3. Refresh Counts

The dynamic RAM's in the R1000's main memory are refreshed by microcode. This is accomplished by using two counters: REFRESH_INTERVAL and REFRESH_WINDOW, which are read and written using TI_BUS(0:15),(16:31), respectively. REFRESH_WINDOW is set to be greater than the longest macro event latency for the current rev of the machine. REFRESH_INTERVAL is set to be the required 2 millisecond refresh period minus the REFRESH_WINDOW.

REFRESH_INTERVAL counts by one every machine cycle. When REFRESH_INTERVAL equals the REFRESH_INTERVAL preset by microcode (by the RESTORE_MAR_WITH_REFRESH microorder from TI_BUS<0:15>), the REFRESH macro event is posted and REFRESH_WINDOW starts counting by one each machine cycle. If the ACK_REFRESH microorder is issued (by the refresh macro event handler) before REFRESH_WINDOW equals the REFRESH_WINDOW preset by microcode (in the last RESTORE_MAR_WITH_REFRESH microorder from TI_BUS <16:31>), the REFRESH_INTERVAL and REFRESH_WINDOW counters are reset and the FORCE_REFRESH machine check event is avoided. The REFRESH_INTERVAL counter is restarted by an ACK_REFRESH.

If the ACK_REFRESH microorder is not issued before REFRESH_WINDOW reaches the preset value, a FORCE_REFRESH machine check occurs, the machine is frozen by the diagnostic system, and the memory boards refresh themselves at the maximum clock rate.

REFRESH_INTERVAL and REFRESH_WINDOW are specified only in the RESTORE_MAR_WITH_REFRESH microorder. TI_BUS(0:31) are ignored for RESTORE_MAR. The preset REFRESH_INTERVAL and REFRESH_WINDOW values are always returned by READ_MAR. READ_MAR must be specified when the ACK_REFRESH MEM_START microorder is issued.

2.2.4. Memory Space Field

The memory space is restored using the TYPE_BUS<61:63>, and selects one from the following list:

0 - Reserved_for_Future_Use SPACE -- must never be used

1 - CONTROL SPACE 20 bit word displacement

2 - TYPE SPACE 20 bit word displacement

3 - QUEUE SPACE 20 bit word displacement

4 - DATA SPACE 25 bit word displacement

5 - IMPORT SPACE 20 bit word displacement

6 - CODE SPACE 20 bit word displacement

7 - SYSTEM SPACE 20 bit word displacement

The memory space field is usually loaded from a microliteral during the LOAD_MAR micro order. The microsequencer drives this field directly from the dispatch RAM's during a dispatch. During one of the RESTORE_MAR microorders, the memory monitor drives TYPE_BUS(61:63) onto the space portion of the ADDRESS bus (see the ADDRESS_BUS description).

The memory monitor places no restrictions of the size of displacements in addresses: it is the responsibility of the source of the address to drive the proper number of significant displacement bits, zero filled in the high order bits, onto the address bus.

Restrictions on the sizes of code and import spaces are now enforced by microcode and software policy, rather than hardware. Note that architectural data structures limit the range of location addressable using instruction fields or stack descriptors. In such cases, the proper number of leading zeros must be driven on the ADDRESS_BUS to fill the word displacement to 25 bits.

The least significant 2 bits of the space field participate in the cache hash function. These must be set to zero for physical memory references.

2.2.5. Stack Name Field (Segment Number, VPid)

The stack name field is actually two fields: the 24-bit Segment Number and the 8-bit Virtual Processor ID (VPid). These bits are saved using VI_BUS(0:31) and are always loaded from the ADDRESS bus. The least significant eleven bits of the Segment Number participate in the hash function.

These nine bits are also used to select the LINE_NUMBER during Physical Tag Store or Physical Memory operations. The most significant four bits of the VIRTUAL_PROCESSOR_ID select the SET_NUMBER during these Physical operations.

2.2.6. Word displacement field (Page Number, Word)

Since 1K byte pages are used, and a word is 128 bits, this field is split by the memory manager into two fields: Page Number (VI_BUS<32:50> and Word (VI_BUS<51:56>). The least significant nine bits of Page Number participate in the hash function, and must be zero during a Physical operation. These bits are saved using VAL_BUS(32:56) and are always loaded from the ADDRESS bus.

2.2.7. Bit Offset Field

The bit offset field is maintained by the FIU as the latched offset field, rather than by the memory monitor (the memory system always deals with word addresses). During READ_MAR operations, the last offset field latched by the FIU is returned as the bit offset portion of the MAR (least significant 7 bits of the logical address. During RESTORE_MAR operations, the least significant 7 bits of the address bus are latched into the FIU offset latch.

The 7-bit Bit Offset Field is not used by the memory boards. The most significant three bits of this field in the Program Counter (CODE space only) select one of the eight, 16-bit macroinstructions stored in the IBUFF.

2.2.8. Address Arithmetic

The memory monitor makes no provision for detecting arithmetic exception conditions when arithmetic is performed on displacements. In general, the operations allow displacements to wrap around some number of sigificant bits (see the space encoding definitions for the number of significant displacement bits in each of the memory spaces). This wrap around is affected by driving the appropriate number of leading zeros in place of the extraneous high order result bits.

2.2.9. Event Handler Considerations

MEMORY_EXCEPTION event handlers and the CORRECTABLE_ERROR handler must determine the address that caused the event. The PAGE_CROSSING and MAR_MODIFIED conditions indicate if an INCREMENT_MAR occurred before the event. If PAGE_CROSSING is set, then 4096 must be added to the MAR to compensate for the wraparound and RESTORE_MAR issued, thereby effectively handling the PAGE_CROSSING event. (The addition should not allow carries to propagate into the stack name field.) The event-causing condition is cleared as a side-effect of testing it.

If MAR_MODIFIED is true (always true on PAGE_CROSSING) then 128 must be subtracted from the MAR to determine the faulting address. If both MAR_MODIFIED and PAGE_CROSSING are true, both conditions must be handled in order to properly compute the exceptional memory address.

2.3. Read Data Register

There are actually ten potential sources of data when a READ_RDR micro order is specified! There are two Read Data Registers on each of four memory boards (one for each plane). The memory monitor remembers which plane hit last and normally selects the corresponding RDR. Since each memory board does not contain a path to load its RDR's from a bus, (it loads them from its RAM's) a "dummy" RDR is

provided (on the Sysbus Interface board) for state restoring. Whenever a RESTORE_RDR micro order is executed, the data is loaded into the dummy, and the monitor sources subsequent READ_RDR operations (until the next LOAD_MAR) from the dummy. The tenth source is the Control Stack Accelerator.

Read data is specified as the source for the VAL and TYP buses by a combination of bus source microorders on the FIU board (see the FIU Functional Specification).

The memory monitor determines which source of read data is valid (memory board, dummy RDR on the SYSBUS board or CSA), and causes the appropriate source to drive the VAL and TYP buses. The memory boards and dummy RDR are driven directly onto the VAL and TYP buses, while the CSA drives these buses via the B-port of the VAL and TYP register file (respectively). Data from a memory board includes ERCC bits, which are checked by the SYSBUS board and may cause an ERCC event. Data from the dummy RDR or CSA are parity checked only. See the discussion of CSA in a later section.

When RDR is specified as the TYP_VAL bus source, if the dummy RDR is valid, it is driven regardless of what other possible sources may be valid. If the dummy is not valid, but the CSA is hitting, the CSA is driven onto the TYP and VAL buses. If there is no CSA hit, which ever memory board is hitting is chosen to drive the buses. If no memory board is hitting, then noone drives the TYP and VAL buses, and a parity error machine check or spurious ERCC event may occur unless the CACHE MISS microevent is enabled. If microevents are disabled, and this situation arises, the CACHE MISS condition must be tested in the cycle in which RDR is being read, and, if CACHE MISS is true, the data read must be discarded. Testing CACHE MISS in this situation prevents TYP and VAL bus parity error machine checks and spurious ERCC events.

The validity of read data is maintained from cycle 2 of the last memory read until the MAR is reloaded (except for page mode memory operations, where the the MAR is reloaded in a reversible way). If the MAR is reloaded before read data is accessed, error correction is impossible (since the source address is lost) and results are unpredictable. For page mode operations, the memory monitor maintains state (MAR_MODIFIED, PAGE_CROSSING and INCOMPLETE_MEMORY_CYCLE flags) for the ERCC and MEMORY_EXCEPTION trap handlers to reconstruct the erroneous memory address.

2.3.1. Error Checking

In the usual case, when data is sourced by the memory boards, the 9-bit CHECK_BIT field is also driven. The ERCC checker on the sysbus interface board checks for errors. If a multiple bit error is detected, the MULTI_BIT_ERROR machine check event occurs. If a single bit error is detected, an ERCC micro event is posted. The event

handler corrects the data, restores it into the Dummy RDR (with byte parity), writes the corrected data back, then logs the error.

The correction is done based on the 7-bit BAD_BIT_ID field and the CHECK_BIT_ERROR condition generated by the ERCC checker. First CHECK_BIT_ERROR is tested to determine if the error is in the check bits. If true, no data correction is necessary since the check bits will be regenerated during the write back. If false, then BAD_BIT_ID0 is tested to determine if the error is in the VALUE half or the TYPE half. A constant "1" is then rotated by the FIU, selected by BAD_BIT_ID(1:6) and is XOR'ed with the data on the selected board.

In all cases of correctable errors, the corrected data is written back to memory by microcode. The PHYSICAL_LAST monitor state bit must be tested to determine if the address in the MAR is logical or physical. The error is logged by storing the MAR on the error log list in the scratchpad and incrementing the error count. If this count exceeds a threshold, a microcode initiated machine check occurs. The refresh event handler maintains a count which causes periodic flushes of this error log list to the diagnostic processor, using the SYSBUS. As a side effect of checking for correctable errors, the Sysbus Interface board generates byte parity on both the VALUE and TYPE halves of the read data, and drives it on the parity lines. Therefore, all users of the read data can check parity.

2.4. Write Data Register

A LOAD_WDR microorder loads data from the VAL and TYP buses into the WRITE DATA REGISTER of all memory boards and the copy of WDR on the VAL and TYP boards (VAL and TYP boards maintain their respective halves of the WDR). Since the VALUE and TYPE boards contain a copy of the Write Data Register, a dummy WDR is not necessary on the memory monitor. A READ_WDR state-saving operation is performed locally on those boards. (The VALUE and TYPE board copies of the WDR are saved in the register file by selecting the register file C-mux source appropriately.) (see the VALUE and TYP board Functional Specifications).

On a LOAD_WDR, the ERCC circuit generates the check bits and drives them to the memory boards. As a side effect of generating these check bits, it checks parity on the VALUE and TYPE busses. The local VALUE and TYPE board copies of the WDR contain byte parity, not check bits.

If a microevent aborts the cycle in which LOAD_WDR is specified, the WDRs on the memory boards are loaded, but not the copy on the VAL and TYP boards. This inconsistency must be resolved by loading WDR before any start write memory operations are issued.

2.5. Memory Operations

All memory operations involve three microcycles. Cycle 0 of a memory operation is defined to be the microcycle which issued a memory start micro order. The MAR must be loaded no later than cycle 0.

Cycle 1 follows, and is the cycle in which the memory operation actually takes place. If a micro event aborts cycle 1, the INCOMPLETE_MEMORY_CYCLE condition is set, and the event return micro order START_IF_INCOMPLETE will restart the operation. During a memory write operation, the LOAD_WDR must occur no later than cycle 1. The Tag Store query implied by the particular memory start is performed in cycle 1. If memory is interrupted during cycle 1, the operation is terminated and LRU is not updated. No data is transferred to or from memory. RDR is destroyed.

Cycle 2 is the final cycle. At this time the read data is available from the RDR and can be used by issuing a READ_RDR micro order. If required, the LRU bits are updated during cycle 2.

If a MEMORY_EXCEPION is raised during the operation, it causes an early micro event in cycle 2, if enabled. No tag store state is updated and no data is written to memory, although the contents of the RDR are lost. The event is persistant and highest priority. Therefore it will occur on the first cycle it is enabled until the event is taken or the causing conditions are tested.

The RDR remains valid until the next LOAD_MAR. After a LOAD_MAR is executed, the MEMORY_EXCEPTION and CORRECTABLE_ERROR handler will not be able to determine which address caused the event.

Overlapping memory operations are allowed. Cycle 2 of one memory operation can be cycle 0 of the next. In the CONTINUE operation the pipelining is doubly overlapped: cycle 2 of the first operation coincides with cycle 1 of the second operation and cycle 0 of the third. This can only be done to consecutive address (refer below to a more detailed explanation of CONTINUE).

This section discusses the standard memory operations: Read Logical, Write Logical, and Continue. It also describes the conditional memory reference mechanism.

2.5.1. Read Logical

Data whose address is in MAR are accessed. During cycle 2 of the memory operation, data is available via the READ_DATA microorder. Normally, the memory monitor transfers data from the proper read data register. If the accessed data resides in the CSA, data is accessed there, and the memory data is ignored (the CSA always contains the most up to date copy of data, since the CSA is not a write-through acceleration mechanism). Choice of RDR or CSA data is made by the memory monitor transparently to requesting microcode.

If the specified logical address is not encached by the memory system, a MEMORY_EXCEPTION micro event is posted, and the CACHE_MISS condition is set. The RDR is destroyed. If the specified logical address is encached, but the PAGE_STATE is set to LOADING, a MEMORY_EXCEPTION micro event is posted, and the CACHE_MISS condition is set. The RDR is destroyed.

2.5.2. Write Logical

The contents of the WDR are written into the memory word whose logical address is in the MAR. If the specified logical address is not encached by the memory system, a MEMORY_EXCEPTION micro event is posted, and the CACHE_MISS condition is set. The RDR is destroyed, but no cache location is written.

If the cache page being written is in either the LOADING or READ-ONLY state, a CACHE_MISS condition is generated. The MEMORY_EXCEPTION event is raised as a cycle 2 event. The handler must query the tag store and check page state to differentiate these states from true cache miss.

It is important to note that, although the previous contents of the written location is placed in RDR during a LOGICAL_WRITE, the microcode should not Read_RDR. This could cause an ERCC error which will write back the corrected, original contents and therefore undo the write. This feature is implemented for diagnostic purposes.

2.5.3. Continue

For extremely high speed transfers (~ 80 Megabytes per second !) to or from consecutive words in memory, the page mode feature of the dynamic RAM's is exploited. This feature is enabled by specifying CONTINUE in the cycle immediately following a memory start. The INCREMENT_MAR micro order must also be issued. A CONTINUE can immediately follow another CONTINUE, thereby allowing entire blocks of data to be transferred at this clip.

INCREMENT_MAR microorder increments the MAR by 128. If the MAR displacement mod 4096 becomes zero as a result of this increment, the PAGE_CROSSING event is posted. The PAGE_CROSSING microevent occurs after the MAR has been incremented (i.e., the MAR contains an address which is 4096 less than the proper address). PAGE_CROSSING is an early microevent in the cycle following the INCREMENT_MAR microorder.

A CONTINUE microorder (with its INCREMENT_MAR) must be issued during cycle 1 of a preceeding START or CONTINUE memory operation (there must be no idle memory cycles between the START of a page mode transfer, and any of the CONTINUE operations).

The reason that a half-page boundary crossing triggers the event

is dicatated by a low-level constraint from the dynamic RAM's themselves. They must be "precharged" every 10 microseconds. By trapping at least every 32 cycles, this constraint is met. The time penalty for the event is only 4 cycles (two dead cycles because it's an early event plus the one microinstruction handler). Another consideration is that, since pages are 64 words in length, each time a 64 word page boundary is crossed by a page mode access, the tag store must be queried again to obtain a new logical to physical association.

When INCREMENT_MAR is issued, the MAR is incremented and driven over the address bus. The incremented version of MAR is loaded back into the MAR from the address bus. (See the description of MAR_MODIFIED state bit). Note that, when INCREMENT_MAR is selected as the MAR_CONTROL microorder, the FIU board must be selected as source for the address bus.

2.5.4. Conditional Memory References

Conditional memory starts are supported. These can be based on either polarity of an early or a "medium-late" condition. If the condition is not taken, another memory start may immediately follow. If the condition is taken, only a CONTINUE or NO_MEMORY_OPERATION may follow. The INCREMENT_MAR is also required during conditional CONTINUE's, but is unconditional. If the condition is not taken, the MAR is still modified, and the PAGE_CROSSING event can still occur.

The other two conditional memory starts are used by event handlers to reconstruct the memory cycle pipeline.

All events handlers must issue a START_IF_INCOMPLETE in the cycle it returns. (Conditional returns are not allowed from event handlers.) If the event aborted cycle 1 of a memory operation, that operation is restarted by this command. If the event aborted cycle 0 of an operation, then the handler will return there and the microcode will start the memory command. (By definition, cycle 0 is the cycle which issues the start command.) If the event aborted cycle 2 of an operation, the operation is not restarted since the operation was completed before the event. (Cycle 2 is defined as the first cycle that READ_DATA can be used following a START_READ, and the first cycle that a MEMORY_EXCEPTION will occur. The operation is considered complete after cycle 1.)

MEMORY_EXCEPTION handlers must issue a START_LAST_COMMAND in the microcycle immediately before it returns. This will restart the command that caused the exception. The last microcycle of these handlers (as required of all micro event handlers) will issue the START_IF_INCOMPLETE command. If there was a CONTINUE in cycle 1 of the memory operation that caused the exception, this event return action will restart the CONTINUE. This is the only circumstance in which a START_IF_INCOMPLETE will resolve to a CONTINUE. The MAR must be incremented to correctly force a continue. Therefore the

INC_MAR_IF_INCOMPLETE micro order must be issued on MEMORY_EXCEPTION handler returns. Whenever INC_MAR_IF_INCOMPLETE is issued, INCOMPLETE_MEMORY_CYCLE must be selected as test condition.

2.6. Memory Management Operations

Each memory board manages four sets of Tag Stores used for associative comparisons and memory management. Instead of four parallel sets of RAM"s and comparators, the Tag Store is implemented with two banks of 1K X 4 Static RAM"s and can be clocked at the double frequency rate. Sets 0 and 1 are always referenced in the first half of a cycle, sets 2 and 4 are referenced during the second half.

2.6.1. Tag Value format

The contents of the Tag Store RAM"s can be read and written over the VALUE_BUS by a combination of bus control microorders (see the SYSBUS specification) and memory commands.

A tag value is latched as the result of any memory operation. The START_PHYSICAL_TAG_READ operation latches the tag value associated with a particular set wihout otherwise accessing memory. The tag value may be read during cycle two or later (it must not be read unless memory is idle). A SETUP_TAG_READ microorder must be issued one cycle prior to reading the tag value, and must not be issued earlier than cycle 1 of the operation which latches the tag value. The tag value is returned over the VAL bus.

The tag store is written using the START_PHYSICAL_TAG_WRITE microorder. START_PHYSICAL_TAG_WRITE is a three cycle operation (cycle0:cycle2). The new tag value to be written must be loaded in the value side of the WDR no later than cycle 1. Memory must remain idle during cycle 2 (i.e., no memory start may be issued until after cycle 2).

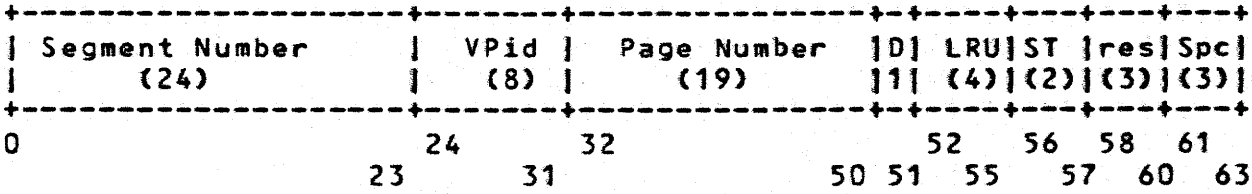
When a tag value is written, all fields are written, including LRU. Therefore, care must be taken to make sure all sets on a particular line have unique LRU values between zero and the MRU set number. Note also that, following power up, all tags on each line of the cache must be properly initialized before any memory operations are issued, or parity errors or unpredictable behavior may result.

2.6.2. Tag Store Addressing

The tag Store is addressed with a frame address which is composed of a nine-bit LINE_NUMBER field and a four-bit SET_NUMBER field.

The LINE_NUMBER is determined by a hash function operating on the two least significant space bits, the eleven least significant segment

TAG_VALUE:



name bits and the nine least significant page bits, producing an elevent bit line address. On the memory monitor, line address bits 0, 9 and 10 are computed combinatorially, while line address bits 1..8 are obtained from a RAM:

```
line address  0 := page address <18> x-or segment name <13>
line address  9 := space <1> x-or segment name <22>
line address 10 := space <2> x-or segment name <23>

line address <1..4> := RAM addressed by
                       page address<14..18> and segment name <14..18>

line address <5..8> := RAM addressed by
                       page address<10..15> and segment name <18..21>
```

The RAM is programmed to produce the same hash function as the 2 MB memory board, namely the bit-wise exclusive-or of Segment<15:23> (least significant 9 segment bits) as one component, and Page<18:12> concatenated on the right with Space<1:2> (least significant 7 page displacement bits, with the bit significance reversed, concatenated with the least significant two space bits) as the other component. This pairs Segment<15> with Page<18>, Segment<16> with Page<17>, and so forth, until Segment<23> is paired with Space<2>. The most significant 2 bits of the line address are set to zero.

When the hash function needs to be bypassed because a particular LINE_NUMBER wants to be addressed (such as in a START_PHYSICAL_TAG_WRITE), the desired LINE_NUMBER is placed in the l. s. nine bits of the SEGMENT_NUMBER, and 0's are placed in the rest of the bits participating in the hash function.

Tag comparisons are implemented on two portions of the tag value: the stack name and the full page logical address. The stack name consists of the segment number (bits 0:23) and the VPid (bits 24:31). The page logical address consists of the stack name, plus the page number (bits 32:50) and the Space (Spc, bits 61:63).

The SET_NUMBER is determined by the particular query mode implied

by the memory start. On physical Tag references the SET_NUMBER is specified by the most significant 4 bits of the VIRTUAL_PROCESSOR_ID. On Logical Tag queries, the set that contains the matching logical page address is the selected set. The Least_Recently_Used set is selected by a START_LRU_QUERY. The first available (invalid) set is selected by a START_AVAILABLE_QUERY. In a START_NAME_QUERY, only the stack name portion of the tag is compared. In this query, and in the available query, multiple set could hit. This is resolved arbitrarily by the memory hardware. (Actually, the lowest set number will win out, but no code should be written that relies on this.)

The remaining tag value bits describe the state of the page. The D bit is set whenever the page is written to via a logical query (it is not set for physical queries or maintenance or random operations). The LRU is described later in this document. The reserved bits are available for use by microcode and are not interpreted by hardware.

The page state (ST) field controls the kinds of access to each page of the memory data array:

Loading (00) This page is not yet ready to be accessed (data is being transferred to or from this page). Logical and name queries will match this entry, but the LOADING_FAULT state bit will be set, and a MEMORY_EXCEPTION event will occur (if enabled).

Read-Only (01) This page may be read, but not written. Logical and name queries will match this entry, but logical writes will set the WRITE_FAULT state bit, and a MEMORY_EXCEPTION event will occur (if enabled). Note that data is written even though the page is Read-Only. See the description of Logical Write.

Read-Write (10) This page may be read or written. Logical and name queries will match this entry. If a match occurs, no tag store related state bits will be set (CACHE_MISS, LOADING_FAULT, WRITE_FAULT).

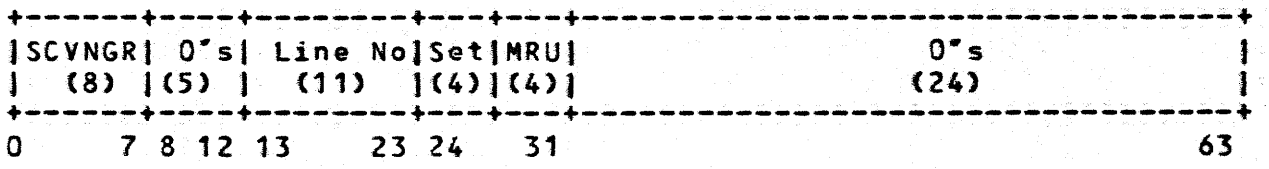
Invalid (11) This page is not assigned any logical page; no logical or name query will match this entry.

2.6.3. Frame Address

A frame address is latched by the memory late in cycle 2 of each memory operation. This may be read after cycle two. Unlike the tag value, frame addresses may be read without any preceeding SETUP. The frame address can be read by a combination of VI bus control microorders (see the FIU specification).

All physical memory and tag store accesses require a frame address in the least significant 64 bits of the MAR. Further, all fields except line number and set number must be cleared to zero prior to loading into MAR. When read, the Frame Address returns the line number to which the logical address in the MAR hashes, and the last set number which hit (if no memory board is presently hitting, all ones are returned as set number: the frame address may be read to convert a logical address to a line number without cycling memory).

FRAME_ADDRESS (read over VI bus):



* SCVNGR - bits <0..7> the contents of the scavenger ram are returned. must be zero when loading a frame address into the MAR.

* LINE NO - bits <13..23> the line number to which the current contents of MAR hashes is returned; the physical line number to be accessed is loaded prior to starting a physical memory query.

* SET - bits <24..27> the set number which hit last is returned, or all ones of no set is hitting; contains the physical set number to be accessed by the next physical memory query. When loading a physical frame address, the set number must be less than or equal to MRU (the highest valid set number), or results will be unpredictable.

* MRU - bits <28 31> the highest valid set number is computed and latched during INITIALIZE_MRU, and is returned as part of the frame address. These bits must be cleared to zero prior when loading a frame address into the MAR.

The line number portion of frame address is computed combinatorially, using the hash function logic. The set number portion is derived from the result of the specific query mode. The MRU set number is latched at memory initialization time, and always returned with the frame address. Finally, the scavenger ram contents are read using the address in the MAR, and returned as part of the frame address. The scavenger ram contents appear combinatorial in that they are derived based on the current contents of the MAR, and do not depend upon the last query mode. Note that scavenger ram contents must be initialized after power up before any frame addresses are read, or a scavenger parity error may result.

2.6.4. LRU Management

Each tag value contains a four-bit LRU field. All the sets in a line contain a unique value in these four bits.

The INITIALIZE_MRU microorder determines the highest implemented SET_NUMBER, which is defined as the Most_Recently_Used (MRU) value. MRU is returned as part of FRAME_ADDRESS. Note that INITIALIZE_MRU does not initialize any tag fields; these must be initialized explicitly by microcode.

When a particular set hits, and the query defines UPDATE_LRU as a side-effect, that set's original LRU value is broadcast, and replaced with the MRU value. All sets whose LRU value is greater than this broadcasted value will decrement their LRU value. All sets whose LRU value is less than this value will remain unchanged. Therefore the Least_Recently_Used set will have a value = 0. Note that the LRU value is updated on all sets including invalid sets, also the LRU value is unchanged by a tag store write operation.

2.7. Contol Stack Accelerator Monitor

The VALUE and TYPE boards can encache as many as fifteen locations on the top of the currently executing Control Stack. Microcode that references the CSA directly uses the the Register File Address fields to specify locations relative to TOS (the top of the Control Stack Accelerator) or to CSA_BOTTOM (the bottom of the Control Stack Accelerator). These locations are guaranteed to be in the CSA by the CSA underflow and overflow macro events.

Any other address to the Control Stack must be monitored to determine if the most recent version of that address resides in the CSA. The address is also monitored for illegal references beyond CONTROL_TOP.

During exit operations, several locations are wiped off the stack in one POP_DOWN_TO operation. After a POP_DOWN_TO, the CSA monitor must determine how many valid entries remain in the CSA, and communicate this to the VALUE and TYPE boards.

2.7.1. Control Stack Accelerator Hits

When MAR is loaded, the new MAR stack name is compared with the stack name of the current control stack. If they match, the 20 bit MAR displacement is subtracted from the displacement of the top of the current control stack (CONTROL_TOP). If this result is negative, the reference is beyond the top of the control stack, and the CONTROL_ADDRESS_OUTOF_RANGE condition becomes true in the second cycle following loading of the MAR. Under these circumstances, if a START microorder is issued, CONTROL_ADDRESS_OUTOF_RANGE is set in cycle 2,

posting the MEMORY_EXCEPTION event and aborting memory. If memory is not started, this condition is not set and MEMORY_EXCEPTION is not posted. Since this is a cycle 2 event, if the memory operation was a write, data is actually writen beyond the top of stack.

If the MAR is incremented beyond the current CONTROL_TOP, the out of range condition is set in the second cycle following the INC_MAR, memory is aborted (late abort) and MEMORY_EXCEPTION is posted. MAR_MODIFIED will be set. If other memory exceptions are also asserted (e.g., CACHE_MISS), the address in the MAR must be unwound to determine the address responsible.

If the result of the subtraction is not negative, then the difference plus one is subtracted from the number of valid entries currently in the CSA. If this result is negative, the reference falls below the CSA, and is directed to memory. If the result is not negative, the reference falls within the current contents of the CSA on the VAL and TYP boards, and the CSA_HIT condition exists. This resulting difference is called the HIT_OFFSET, and is relative to CSA_BOT. The memory monitor broadcasts both CSA_HIT and CSA_OFFSET to the VAL and TYP boards each cycle. VAL and TYP latch them for accessing the CSA during the next cycle. CSA_HIT AND HIT_OFFSET are latched every cycle, whether or not that cycle is being aborted due to a bad hint or an event. When cycle 1 of a write operation is not being aborted, the memory monitor broadcasts a write signal to the VAL and TYP boards, which is also always latched, and used in the next cycle (cycle 2) to gate data from their copies of the WDR into the CSA. When RDR is specified as the source of the TYP and VAL buses, the sysbus board asserts a read_RDR signal to the TYP and VAL boards in the same cycle that the RDR is being read. These boards use this signal, along with the CSA_HIT and HIT_OFFSET values latched the previous cycle, to read the data from the CSA.

While CSA_HIT is TRUE, memory starts are handled as usual: read operations access memory and load the RDR while writes write data to memory. Cache hits are suppressed when CSA_HIT is true. entering cycle 1 (effectively, memory is not started, memory state is Note that the timing of CSA hits is identical to the timing of any other memory operation.

During cycle 1 of a write operation, data are loaded into to the VAL and TYP board copy of the WDR from the VAL and TYP buses (as always). The memory monitor informs the VAL and TYP boards that a memory write cycle 2 is occuring. VAL and TYP use the CSA_OFFSET latched in the previous cycle to gate their local copy of WDR into the register file via the C-mux and C-port (see the VAL board spec for required C-mux and C-port microorders during cycle 2 of a write operation).

2.7.2. LOAD CONTROL TOP, PUSH/POP and INC/DEC BOT

The memory monitor maintains the number of valid CSA entries (NUM_VALID). All CSA references from the memory monitor to the VAL and TYP boards are relative to CSA_BOT (which is maintained by the VAL and TYP boards concurrently with the memory monitor and sequencer copies). The monitor informs the other boards of changes in the status of the CSA via the HIT_OFFSET and three additional wires: POP_DOWN, LOAD_TOS and LOAD_BOT. Normally, these latter three wires are not asserted. When they are asserted, the CSA is being modified, and the monitor suppresses CSA_HIT. None of the CSA modification microorders should be specified during cycle 1 of a write which should be directed to the CSA, nor in the cycle prior to reading the RDR if the CSA may contain the valid copy of data (these restrictions really apply to any CONTROL access, or any memory access whose memory SPACE is not known). In these cases, the RDR must not be read until a full memory read cycle has completed.

A LOAD_CONTROL_TOP microorder loads the address on the ADDRESS BUS into CONTROL TOP, and clears NUM_VALID to zero. A minus one is broadcast as HIT_OFFSET to the TYP and VAL boards, along with the POP_DOWN signal. HIT_OFFSET is added to BOT by the TYP and VAL boards, and stored in TOS, invalidating the CSA.

PUSH and POP microorders broadcast plus one and minus one, respectively, as HIT_OFFSET, along with LOAD_TOS. The TYP and VAL boards add HIT_OFFSET to BOT, storing the result in TOS. These microorders add or subtract an entry from the top of the current CSA.

INC and DEC BOT microorders shrink or grows the CSA from the bottom (respectively) by adding one to or subtracting one from the BOT register on the TYP and VAL boards. Plus or minus one is broadcast as HIT_OFFSET by the monitor, along with LOAD_BOT.

2.7.3. START_POP_DOWN and FINISH_POP_DOWN

POP_DOWN_TO is a two cycle operation. START POP DOWN latches the offset portion of the new CONTROL_TOP from the ADDRESS_BUS into CONTROL TOP, and saves the old CONTROL TOP offset. The NUM VALID is not cleared.

During FINISH POP DOWN, the new NUMBER_VALID is computed by subtracting the CONTROL_TOP new offset from the saved offset. This result is then subtracted from the number of valid entries in the CSA, and a negative result is set to zero. A NUMBER_VALID of zero indicates that the entire CSA has been invalidated. NUMBER_VALID minus one is broadcast to the VALUE and TYPE boards during cycle 1 in place of HIT_OFFSET, along with POP_DOWN. (If NUMBER_VALID is negative, a minus 1 is broadcast to the VAL and TYP boards, which invalidates the entire CSA contents).

Events must be disabled between issuing START POP DOWN and completing FINISH POP DOWN. The ERCC event handler must take case not to modify the state of the CSA, since if CONTROL TOP is modified, the saved top offset required by FINISH POP DOWN may be lost. Since FINISH POP DOWN uses HIT_OFFSET and suppresses CSA_HIT, memory writes must not be started in the cycle which issues FINISH POP DOWN if there's any chance that the data should go to the CSA. Similarly, the RDR must not be read in the cycle following a FINISH POP DOWN if the valid copy of data is in the CSA. These memory restrictions apply to continues as well as starts.

2.8. Scavenger Monitor

The memory monitor contains a circuit that monitors all logical memory references and can trap on certain patterns. As a bare minimum this circuit must support the current approach to garbage collection which dictates that each collection (collections are identified by the MSB of the SEGMENT_NUMBER = 1) be split into eight parts. At any time, three of these collection octants can be in the midst of garbage collection and could contain forwarding addresses instead of actual data. References to these active garbage collection octants must be trapped.

If maximum flexibility is to be maintained for garbage collection, and other potential requirements to trap on certain address patterns, this circuit can be implemented to perform a much more general pattern match than the eight bits required for the current approach to garbage collection.

The scavenger is addressed using the most significant nine bits of the MAR segment number and a bit derived similarly to the WRITE_LAST memory monitor state flag. These address bits are derived combinatorially using the state of the memory monitor at the time a START or READ_FRAME_ADDRESS microorder is issued. The 8 bit contents of the addressed scavenger location is returned as part of the frame address.

When a START microorder is issued for a logical read or write, a scavenger location is read, and the space specification of the MAR address is used to select a bit within that location. If the selected bit is one, the SCAVENGER_TRAP monitor flag is set, and a MEMORY_EXCEPTION event is posted during cycle 2 of the memory operation. Memory is cycled and writes complete even when SCAVENGER memory exception is taken. The original contents of a written location are saved in the RDR. The SCAVENGER TRAP handler must undo writes using the data in the RDR.

The scavenger ram is not accessed during physical data accesses, or during any tag store maintenance or random operations.

Only the first 7 scavenger bits are actually used to trap

references. The eighth bit, which would have traced references to system spaces, is used to store byte parity. This precludes using the scavenger to trace references to system address spaces.

The scavenger monitor ram is written over the VAL bus (least significant 8 bits) using the WRITE_SCAVENGER_MONITOR microorder. The MAR must have been loaded with a logical address prior to issuing this microorder, and the proper Scavenger monitor address must be computed by performing a NAME QUERY (write access trapping) or an AVAILABLE QUERY (read accessing trapping). The results of these operations may be ignored: they are only used to set the internal state of the scavenger monitor address data path.

The 8 bits transferred over the VAL bus must include parity in bit 7, and a one bit in each of the preceeding 7 bit positions corresponding to the space which should be trapped (i.e., one in bit 1 will trap accesses to control segments whose high order 9 segment bits correspond to those currently in the MAR). The microcode must compute correct parity.

3. Microword Specification

3.1. Field Specifications

3.1.1. MEMORY_START field - 5 bits

All of the following microorders expect a logical address to be loaded into MAR, except those whose names specify "PHYSICAL". PHYSICAL starts require a frame address in the least significant 64 bits of the MAR, with all fields cleared to zero except the LINE_NUMBER and SET_NUMBER.

The following are Data Query microorders, which access data in the memory data array:

* NO_MEMORY_OPERATION (NOP)
* START_READ
* START_WRITE
* CONTINUE
* START_READ_IF_TRUE
* START_READ_IF_FALSE
* START_WRITE_IF_TRUE

* START_WRITE_IF_FALSE

* START_CONTINUE_IF_TRUE

* START_CONTINUE_IF_FALSE

* START_LAST_COMMAND

* START_IF_INCOMPLETE

* START_PHYSICAL_READ

* START_PHYSICAL_WRITE

The following are tag store maintenance and random microorders. These manipulate the tag store and control structures of the memory monitor, and set up data paths for state transfers:

* START_PHYSICAL_TAG_READ

* START_PHYSICAL_TAG_WRITE

* START_TAG_QUERY

* START_LRU_QUERY

* START_AVAILABLE_QUERY

* START_NAME_QUERY

* SETUP_TAG_READ

* INITIALIZE_MRU

* WRITE_SCAVENGER_MONITOR

* ACK_REFRESH

* IDLE

3.1.2. MAR_CONTROL field - 4 bits

* RESTORE_MAR

* RESTORE_MAR_WITH_REFRESH

* INCREMENT_MAR

* INC_MAR_IF_INCOMPLETE

* LOAD_MAR xxx (space micro literal driven on ADDRESS.SPACE)

* RESTORE_RDR
* NO_MAR_CONTROL (NOP)

3.1.3. LOAD_WDR - 1 bit

3.1.4. CSA_CONTROL field - 3 bits

* LOAD_CONTROL_TOP
* START_POP_DOWN
* FINISH_POP_DOWN
* PUSH_CSA
* POP_CSA
* INCREMENT_CSA_BOTTOM
* DECREMENT_CSA_BOTTOM
* NO_CSA_CONTROL (NOP)

3.1.5. ADDRESS_SOURCE field - 3 bits

The centralized control of the source of the Least Significant 64 bits of the Logical address is contained in the memory monitor. The individual sources are responsible for monitoring ADDRESS.SPACE and zeroing out the appropriate Most Significant bits of ADDRESS.PAGE. This control has moved to the sysbus board.

4. Conditions

Refer to the following table for an enumeration of the memory monitor conditions.

5. Memory Control Codes

The following table describes the control modes directed by the memory monitor to the memory boards, and their side affects:

Condition	When set	Event	When cleared	active hi/lo
MAR_NEAR_TOPOFPAGE	ML, LOAD_MAR C1	no	LOAD_MAR	L
REFRESH	E, C1	early macro	ACK_REFRESH	H
WRITE_LAST	E, mem start C2	no	mem start	H
PHYSICAL_LAST	E, mem start C2	no	mem start	L
INCOMPLETE_MEMORY_CYCLE				
	E, event C1	no	START_IF_INC	H
MAR_MODIFIED	E, event C1	no	by testing	H
PAGE_CROSSING	E, INC_MAR C1	early micro	by testing	L
MEMORY_EXCEPTION	ML	early micro	by component	L
CACHE_MISS *				
(test condition)	ML, mem start C2	no	mem start	L
(MAR state bit)	ML, mem start C2	component	by testing	L
SCAVENGER TRAP *				
(test condition)	ML, LOAD_MAR C2	no	load MAR	L
(MAR state bit)	ML, mem start C2	component	by testing	L
CONTROL_ADDRESS_OUTOF_RANGE *				
(test condition)	ML, LOAD_MAR C2	no	load MAR	L
(MAR state bit)	ML, mem start C2	component	by testing	L
CSA_HIT	ML, LOAD_MAR C1	no	LOAD_MAR	H
CORRECTABLE_ERROR#	ML, READ_RDR C0	early micro	by testing	H
CHECKBIT_ERROR #	E, READ_RDR C1	no	READ_RDR	H
BAD_BIT_IDO #	E, READ_RDR C1	no	READ_RDR	H

These refer to the cycle during which the set value of the condition is available for testing or event posting:

E = Early condition, ML = medium late condition, L - Late condition

C0 = the cycle of the microorder causing the condition,
C1 = the cycle following the one in which the condition was caused
C2 = the second cycle following the one in which the condition was caused.

Active Hi/Lo: H = a True value indicates the condition is asserted.
L = a False value indicates the condition is asserted.

* These conditions are components of the MEMORY_EXCEPTION condition
# - on Sysbus interface Board

6. Microcode Restrictions

OPERATION	CODE	QUERY	LRU	MODIFIED	WRITE PHYS	LAST/ LAST
PHYSICAL MEMORY WRITE	0	PHYSICAL	PASS	PASS	1	1
PHYSICAL MEMORY READ	1	PHYSICAL	PASS	PASS	0	1
LOGICAL MEMORY WRITE	2	LOGICAL	UPDATE	SET	1	0
LOGICAL MEMORY READ	3	LOGICAL	UPDATE	PASS	0	0
COPY_0_TO_1	4	DIAG	PASS	PASS	1	0
SCAN_0	5	DIAG	**	**	0	0
COPY_1_TO_0	6	DIAG	PASS	PASS	1	0
SCAN_1	7	DIAG	**	**	0	0
PHYSICAL_TAG_WRITE	8	PHYSICAL	WRITE	WRITE	1	1
PHYSICAL_TAG_READ	9	PHYSICAL	PASS	PASS	0	1
INIT_MRU	A	CLEAR	-- undefined --		1	0
TAG_QUERY	B	LOGICAL	PASS	PASS	0	0
NAME_QUERY	C	NAME	PASS	PASS	1	0
AVAILABLE_QUERY	D	AVAIL	PASS	PASS	0	0
LRU_QUERY	E	LRU	UPDATE	PASS	1	0
IDLE	F	-- hold previous state --				

** during scan operations, Tag Store 1 is used to save the read data, therefore the LRU and modified bit fields of tag store 1 are written with the corresponding data.

1. The MAR, RDR, and WDR must be saved by the memory monitor.

2. LOAD_MAR for next reference can't precede READ_RDR for last.

3. LOAD_WDR must be no later than one cycle after START_WRITE (or anything that resolves to a START_WRITE).

4. Both the LOAD_MAR and START_READ microorders must be specified whenever a DISPATCH or USUALLY_DISPATCH is specified. The sequencer must be specified as source of the ADDRESS BUS. The sequencer will abort the start, if it isn't needed, but MAR is destroyed.

5. FIU must be specified as source of ADDRESS bus during INCREMENT_MAR.

6. PHYSICAL_TAG_WRITEs for entire line must follow INITIALIZE_MRU.

7. The reserved_for_future_use Memory space should never be referenced.

8. Micro events should be disabled when playing with the Tag Store.

9. A READ_RDR should not be issued following a START_WRITE

10. START_LAST_COMMAND and INC_MAR_IF_INCOMPLETE should only be used by MEMORY_EXCEPTION handlers. INCOMPLETE_MEMORY_CYCLE must be tested in the cycle that issues the START_IF_INCOMPLETE or INC_MAR_IF_INCOMPLETE in order to select that condition. Testing also clears the condition.

11. No Memory Start commands can be issued when a RESTORE_MAR is done. Events must be disabled when doing a RESTORE_MAR; memory must be idle. The space portion of the ADDRESS BUS is driven from TYPE BUS <60..63>. If the new MAR Random bits are originating on the TYP board, they must be routed over the TYPE bus to the TI bus (specify TYP ad TI source on the FIU board). When reading the MAR, the TI and VI buses must be routed to the TYP and VAL buses (respectively). The random bits may be both read and loaded at the same time by specifying MAR_MAR as TI_VI source, and RESTORE_MAR, which drives the random bits onto the TI, and loads them from there.

12. An Event Handler should never return using a conditional return.

13. START_PHYSICAL_TAG_WRITE is a three cycle operation. The new tag value must be written to the WDR no later than cycle 1. Cycle 2 must be an idle memory cycle. The cycle following cycle 2 is the first one in which a memory operation may be specified. Events must be disabled during this operation.

14. A conditional start or conditional continue that fails must not be followed by a continue or a conditional continue that succeeds.

15. Control references to memory must not be started during START POP DOWN. The RDR must not be read in the cycle following FINISH POP DOWN if there's any chance the valid data is in the CSA until the next full memory read completes (i.e., cycle 2 of the next start read). Memory may be started during FINISH POP DOWN. No CSA microorder other than NO CSA CONTROL (NOP) may be issued between START POP DOWN and FINISH POP DOWN.

16. LOAD_CONTROL_TOP must only be specified when memory is either idle or in cycle 2. Following a LOAD_CONTROL_TOP, RDR must not be read until a full memory read cycle completes (i.e., cycle 2 of the next start read).

17. Microevents which abort loading of MAR or WDR leave these registers in an inconsistent state (i.e., memory board copy is loaded, while the processor copy is not). These must be made consistent by successfully loading the MAR or WDR before memory is started. This is easily done by handlers always loading MAR before issuing memory starts or reading RDR (which might result in an ERCC event). As long as the proper timing of START_IF_INCOMPLETE and RETURN are observed, an interrupted LOAD_WDR should be reexecuted correctly.

18. The memory monitor conditions table indicates which memory monitor conditions are positive asserted, and which are negative asserted.

19. If a scavenger trap or out of range memory exception occurs while writing to memory, data are written even though the memory exception event is taken. In the latter case (out of range), the write may be ignored, since no valid data exists beyond the top of the control stack for a running task. In the former case (scavenger trap), the handler must undo the write, if it chooses to, by reading the RDR on the memory board (which contains the old contents of memory), and writing it to the offending location. Since the DUMMY_RDR is enabled following a WRITE, the handler must issue DISABLE_DUMMY_NEXT MAR_CONTROL random in the cycle prior to reading RDR, to get the old contents from the memory board. This may cause ERCC events. The RDR is not loaded during page mode writes, so the RDR will maintain the contents of the first location written during page mode writes.

20. OUT OF RANGE condition is testable in the second cycle following loading of the MAR or the CONTROL TOP.

21. If you want the FRAME ADDRESS on the VAL bus you must specify the FIU as VAL bus source. FRAME ADDRESS can't be read until CYCLE3 or later of a memory cycle. It must never be read during cycle 2 of a memory cycle.

22. The Scavenger Ram is accessed using a bit derived similarly to WRITE_LAST. In order to read or write the scavenger ram, this bit must be set properly, using NAME_QUERY to set it to one (WRITE_LAST) or AVAILABLE_QUERY to set it to zero (READ_LAST). Neither of these microorders will modify LRU or any other TAG state. Note: the scavenger ram parity cannot be initialized under microcode control without disabling parity checking for both the memory board tag stores and the scavenger rams themselves.

23. Testing a MEMORY EXCEPTION component or PAGE_CROSSING condition during CYCLE2 of a memory reference will cause the corresponding MAR state bit to get cleared. If events are enabled, an event will occur in CYCLE2, but the MAR state bit will be cleared, destroying evidence of why the event occurred. Thus, events must be disabled when testing MEMORY EXCEPTION component conditions or PAGE CROSSING. Note that MEMORY_EXCEPTION may be tested without side affects.

24. MEMORY EXCEPTION components are cleared in the MAR when tested, which clears the event, but the test condition is not cleared until the proper registers are reloaded, or (for CACHE_MISS) a full memory operation completes.

25. A MEMORY_EXCEPTION is posted during cycle 2 of a memory operation in which one of these components is becoming set (CACHE_MISS) or is already set (possibly OUTOF_RANGE or SCAVENGER_TRAP), or in the cycle following the one in which a RESTORE_MAR sets one of these MAR flag bits.

    In the latter case, the MAR flag bit may be set while the test condition is not true. In such cases, testing the condition will yield a false, and clear the MAR flag bit, which may not be what you want. The three memory exception MAR flags, and the page crossing flag, are testable from the TYPE_BUS (they fall in bits 32..35) independent of the current value of their respective memory monitor test conditions.

26. READ_MAR must be specified (MAR_MAR in TI_VI_SRC) when the ACK_REFRESH MEMORY_START microorder is issued.

7. Event Timing and Aborted Operations

The memory board pipelines operation directives, meaning that the memory board does not act on a directive (such as a start or an abort) until the cycle after the microinstruction in which the directive is issued. Thus, when a start microorder is issued in cycle zero, it is latched by the memory board at the end of cycle zero and examined and executed during cycle one. Memory state changes are committed during cycle 2. Thus, an operation must be aborted in cycle zero in order for the memory state machine to be stopped. Later than cycle zero, the state machine must run for its entire cycle before it is available to accept new commands. An early abort suppresses the memory finite state machine such that a new operation may be started immediately. An early abort is issued to the memory board during cycle 0 and latched. During cycle 1, the board examines both early abort and the memory control code and, if the operation is not aborted, initiates the requested operation. Note that the memory control code must be held stable during both cycle 0 and cycle 1, or the memory board will get confused.

Even though the state machine's timing can't be altered, state changes dur to a memory operation may be suppressed during cycle 1. Such an operation is called a late abort, and turns the current operation into a read, suppresses writing data to the ram array, and suppresses updating the tag store. Late abort is issued to the memory board during cycle 1 and latched. The memory board only commits state changes if the latched late abort value allows, although the RDR is lost even when the operation is late aborted. RDR is preserved if the an operation is early aborted.

Since memory is pipelined, up to three operations may be active at once: an operation may be completing (in cycle 2), another may be in progress (in cycle 1) and a third may be starting (in cycle 0).

Early and late abort apply to the operation in the appropriate stage of the pipeline. For example, asserting early abort will abort the operation in cycle 0, but not affect operations in cycle 1 or cycle 2. Similarly, late abort affects the operation in cycle 1, but not operations in cycle 0 or cycle 2. There is no way to abort an operation in cycle 2.

When a conditional memory start is issued, the memory monitor issues the memory start as it would for an unconditional start, and asserts early abort is the condition proves false:

```
microcode:                      resulting action:
---------                       ----------------
if FOO then                   START_MEMORY_READ;
  START_READ      ==>          if not FOO then
end if;                           EARLY_ABORT;
                            end if;
```

When a microevent occurs, both early and late abort are asserted: in general, memory can't be started until the second cycle of the handler if the event was an early event, since the memory may still be busy.

When a DISPATCH is issued, the microcode must specify START_READ, LOAD_MAR, and the sequencer must be the ADDRESS_BUS_SOURCE for both logical address and space portions. The sequencer supplies the memory address from the dispatch ram. If no memory operation is required, the sequencer asserts EARLY_ABORT in the cycle in which the dispatch was issued.

A USUALLY_DISPATCH is handled similarly, except that, when the hint proves false, the sequencer asserts LATE_ABORT and stops the clock for a cycle. The next sequential microinstruction may start memory, since it is delayed in time one clock, allowing the memory finite state machine to run its full (aborted) cycle.

Table of Contents

1. Summary 1
2. Functional Description 1
2.1. Address Bus 1
2.2. Memory Address Register 2
2.2.1. Memory State Field (State) 3
2.2.2. Fill Mode (FM) and Length (FIU length) Fields 6
2.2.3. Refresh Counts 7
2.2.4. Memory Space Field 7
2.2.5. Stack Name Field (Segment Number, VPid) 8
2.2.6. Word displacement field (Page Number, Word) 8
2.2.7. Bit Offset Field 9
2.2.8. Address Arithmetic 9
2.2.9. Event Handler Considerations 9
2.3. Read Data Register 9
2.3.1. Error Checking 10
2.4. Write Data Register 11
2.5. Memory Operations 12
2.5.1. Read Logical 12
2.5.2. Write Logical 13
2.5.3. Continue 13
2.5.4. Conditional Memory References 14
2.6. Memory Management Operations 15
2.6.1. Tag Value format 15
2.6.2. Tag Store Addressing 15
2.6.3. Frame Address 17
2.6.4. LRU Management 19
2.7. Contol Stack Accelerator Monitor 19
2.7.1. Control Stack Accelerator Hits 19
2.7.2. LOAD CONTROL TOP, PUSH/POP and INC/DEC BOT 21
2.7.3. START_POP_DOWN and FINISH_POP_DOWN 21
2.8. Scavenger Monitor 22
3. Microword Specification 23
3.1. Field Specifications 23
3.1.1. MEMORY_START field - 5 bits 23
3.1.2. MAR_CONTROL field - 4 bits 24
3.1.3. LOAD_WDR - 1 bit 25
3.1.4. CSA_CONTROL field - 3 bits 25
3.1.5. ADDRESS_SOURCE field - 3 bits 25
4. Conditions 25
5. Memory Control Codes 25
6. Microcode Restrictions 26
7. Event Timing and Aborted Operations 30