Functional Specification of the Value Board

DRAFT 2

People exaggerate the things they've never had,
they admire values because they have no experience
with them.

- George Bernard Shaw

Rational Machines proprietary document.

1. Summary

This document describes the complete functionality of the Value board for the R1000. The purpose of this specification is to formally define the operation of the Value board to a level of detail that allows microcode, hardware, and packaging designers to interface with this board correctly. The reader is presumed to be reasonably familiar with the R1000 architecture and to have access to the specifications of the other boards for explanations of their functionality.

The organization of this document is as follows; Section 2 provides a detailed definition of the functionality, on a block by block basis, of each block on the attached block diagram. Section 3 defines the Value board microword along with its encodings. Section 4, along with the previous section, defines the microcode interface to the Value board by specifying what hardware resources are available to the microcoder and the restrictions that are placed on these resources. Section 5 discusses the diagnostic strategies that are employed to debug the board at both the hardware and microcode levels and what hardware support is available to support these strategies. Finally, section 6 details the issues that concern the hardware and packaging designers when interfacing to the Value board. These issues include timing considerations, chip count and power estimates, and board layout details.

2. Block Diagram Functional Definition

This section references the block diagram of the Value board attached to this document. The functionality of each block in the diagram is discussed in detail in the following sections.

2.1. Register File

The principal resource for storing and retrieving data on the Value board is the register file (RF). The RF is a "three address" structure, i.e. two locations (designated A and B corresponding to the A and B inputs of the ALU) can be independently addressed and used either as operands to the ALU or multiplier or as sources to the VAL or FIU busses. On the same cycle as A and B are addressed, a third location, named C, can be written into either to store the result of an ALU operation or to store the data coming over the FIU bus. All of the data contained in the RF is 64 bits wide.

The Register File memory is partitioned into three areas. The bottom 16 locations contain the general purpose registers (GP's). These registers, in general, should be used to store temporary values that may be needed during execution of a microinstruction.

The next 16 locations in the RF contain the special purpose

Table of Contents

List of Tables

addresses. These addresses give the microcode access to the following resources:

- The control stack accelerator (CSA). The CSA is a buffer that can contain the top 15 elements of the currently executing control stack. To minimize the number of control bits, not all of these addresses are immediately visible, however they are kept in the RF for efficency of binary operations.
- The current value stored in the loop counter.
- The current contents of the zero detector circuit.
- The output of the multiplier.
- The random micro-state present on the board.

The remaining 992 locations contain the scratch pad registers. In general, these registers should be used to store constants, templates and masks, and temparary variables that are needed for longer than a single microinstruction. Further discussion about when to use GP's and when to use scratch registers is contained in section 4.1 of this document.

2.1.1 Register File Addressing

Since there are 1024 RF locations, a minimum of 10 bits each is necessary in order for the A, B, and C address fields to access the entire RF (10+10+10 = 30 bits of microword control). To reduce the number of microcode bits controlling RF addresses, a 5 bit field called the "frame pointer" (FRAME) was introduced and each address field was reduced to 6 bits (6+6+6+5 = 23 bits of control). This addressing scheme breaks the 1024 RF locations into 32 frames of 32 locations each. Frame 0 contains the 16 GP registers and all of the special addresses, frames 1 through 31 contain the scratch pad registers. The encodings of the three address fields are shown in Table 1. The notation used in the table is as follows.

GP XXXX
: Addresses the general purpose register specified by the 4 least significant bits of the microcode address field. The upper 6 bits of the address that specify the registers frame are set to zero by the hardware.

REG(FRAME, XXXXX)
: Addresses the register specified by the 5 offset bits given in the microword. The upper 5 bits that specify the registers frame are read from the FRAME field of the microword.

TOP+/-N	Addresses the element at offset N from the top of the current control stack.
REG(LOOP_CNTR)	Addresses the register pointed at by the loop counter.
LOOP COUNTER	Addresses the contents of the 10 bit loop counter.
ZERO DETECTOR	Addresses the contents of the zero detector.
PRODUCT	Addresses the output value of the multiplier.
BOT	Addresses the bottom valid element in the control stack accelerator.
BOT-1	Addresses the element one below the bottom valid element of the control stack accelerator.
RANDOM STATE	Addresses all of the single bits of micro-state contained on the board and assembles them all into one location for easy access. See section 4.1 for more details.
VAL BUS (CSA)	Addresses either the data that is on the VAL bus this cycle, or the location in the control stack accelerator that corresponds to the control stack address being requested. This mechanism, and the operation of the control stack accelerator in general is discussed in the next section.

2.1.2. Control Stack Accelerator

The control stack accelerator (CSA) is an area in the RF that contains some number (up to 15) of the top elements of the currently executing control stack. The VAL board hardware maintains two pointers into the CSA. The TOP register, which points to the location in the CSA that holds the current top of stack. And the BOT register, which points to the bottom valid element that is in the CSA. When the machine first starts running, the CSA is initialized such that TOP points to the location one below BOT (so that when the first element gets pushed onto the CSA TOP and BOT will point to the same location) and all locations in the CSA are marked as invalid.

There are two methods of accessing the control stack accelerator. One way is to explicitly address a CSA location under microcode control. As indicated in the previous section, not all 15 elements in the CSA are directly addressable by the microcode. The locations available for direct reading (via an A or B address) or direct writing (via a C address) are:

- +1 through -8 relative to the current top of the control stack (The remaining elements are not explicitly addressable

Table 2-1: Register File Addressing

Microword Field	A Address Field	B Address Field	C Address Field
00xxxx	gp xxxx	gp xxxx	gp xxxx
010000	TOP+0	TOP+0	TOP+0
010001	TOP+1	TOP+1	TOP+1
010010	spare	spare	random state
010011	reg(loop_cntr)	reg(loop_cntr)	reg(loop_cntr)
010100	random state	BOT-1	BOT-1
010101	zero detector	BOT	BOT
010110	product	VAL bus (or CSA)	write disable
010111	loop counter	spare	loop counter
011000	TOP-8	TOP-8	TOP-8
011001	TOP-7	TOP-7	TOP-7
011010	TOP-6	TOP-6	TOP-6
011011	TOP-5	TOP-5	TOP-5
011100	TOP-4	TOP-4	TOP-4
011101	TOP-3	TOP-3	TOP-3
011110	TOP-2	TOP-2	TOP-2
011111	TOP-1	TOP-1	TOP-1
1xxxxx	reg(frame,xxxxx)	reg(frame,xxxxx)	reg(frame,xxxxx)

by the microcode but can be accessed when the CSA gets "hit" on a memory reference).

- The bottom valid entry in the CSA.

- The entry one below the bottom valid entry in the CSA.

The other method of accessing locations in the CSA is not directly under microcode control and occurs whenever a control stack location that is being referenced (as though it were in memory) happens to reside in the CSA.

When the microcode issues a "LOAD MAR" command, the memory monitor examines the address on the bus to see if it refers to the current control stack, and compares it to the current contents of the CSA. If the addressed location does reside in the CSA, then the hardware flags that the pending memory read has "hit" in the CSA. This HIT flag persists until another LOAD MAR command is given. If another LOAD MAR command is issued before the first location is accessed, the memory monitor simply resets the HIT flag and repeats the comparison procedure described above on the new memory address.

If a READ RDR command is issued, the hardware inhibits the (invalid) memory data from being placed on the VAL and TYPE busses and instead

drives the value in the CSA out onto the busses. The timing of this operation, i.e. when the data is placed on the bus or is available as an operand to the ALU, is exactly the same (from a microcode point of view) as if the data had come from memory. Similarly during a START WRITE command, if the addressed location is in the CSA the contents of the WDR are written into the CSA location during the second cycle after the START WRITE command instead of being written out to memory.

Since every time there is a reference made to memory (actually control stack space) there is a possibility that the data will come from the CSA, a restriction is placed on the microcode that nothing can be sourced from the B address of the RF during a READ RDR cycle. Further discussion of this and other microcode restrictions is given in section 4.5 of this document.

The following operations on the locations in the CSA are available to the microcode in the CSA micro-order of the FIU control word:

PUSH STACK — The value of the top of stack pointer (TOP) gets incremented by one.

POP STACK — The value of TOP gets decremented by one.

INC BOT — The pointer to the bottom valid location of the CSA (BOT) gets incremented by one.

DEC BOT — BOT gets decremented by one.

POP DOWN TO — This operation loads the top of stack pointer with a new address that is some number of locations below the current top of stack. The sequence of events for this operation are:

1. In "Cycle 0", the address of the new top of stack is driven out onto the address bus. During this cycle the POP_DOWN_TO command is given by microcode to the memory monitor.

2. During Cycle 1, the CSA control logic in the memory monitor computes the correct offset to adjust the TOP register on the CSA and at the end of this cycle the new value is loaded into this register. If the operation popped the stack down by more than the number of valid entries that were in the CSA, then the CSA will be put into its initialized state (i.e. TOP = BOT-1 and all entries are invalid).

3. At the beginning of Cycle 2, the new value

of top of stack is ready to be used for any calculation.

2.2. ALU

The principal resource for manipulating data on the VAL board is the 64 bit ALU. The ALU has two inputs designated A_INPUT and B_INPUT. The folllowing sources can be A_INPUT operands:

- The register file location pointed to by the A address field of the microword.
- The output (product) of the multiplier.
- The value stored in the Zero Detector.
- The value stored in the Loop Counter.

The following sources can be B_INPUT operands:

- The register file location pointed to by the B address field of the microword.
- The value on the VAL data bus.

The output of the ALU can either be driven onto the address bus, loaded into the Loop Counter, or loaded into a Register File C Address (through the Shift Mux).

The operations that the ALU can perform are specified by a 5 bit field in the VAL microword. The most significant bit of this field breaks the operations into two groups: logical (MSB = 1) and arithmetic (MSB = 0). Table 2-2 shows the microword encodings, names, and results of all of the ALU operations.

Of the 16 arithmetic ALU operations listed in the table, the last 8 are conditional operations. During each microcycle, the microcode can select one of the testable conditions on the VAL board to be sent over to the SEQUENCER to participate in a conditional branching operation. At the end of every cycle, the condition that was selected gets latched on the VAL board and may be used in the following microcycle to select the outcome of a conditional ALU operation. In general, this ,LAST_VAL condition is the only condition that can participate in the conditional ALU op, the only exception to this rule is when the DIVIDE random is selected. In this case, the Q_BIT condition is used to determine the the result of the conditional add/subtract operation. Additional details of the divide operation are described in section 4.4.1 of this document. A description of each conditional ALU operation is as follows:

C_ADD/SUB_T	The ALU function is PLUS when the condition is TRUE, MINUS when the condition is FALSE.
C_ADD/SUB_F	The ALU function is PLUS when the condition is FALSE, MINUS when the condition is TRUE.
C_ADD/INC_T	The ALU function is PLUS when the condition is TRUE, PLUS_INC when the condition is FALSE.
C_ADD/INC_F	The ALU function is PLUS when the condition is FALSE, PLUS_INC when the condition is TRUE.
C_SUB/DEC_T	The ALU function is MINUS when the condition is TRUE, MINUS_DEC when the condition is FALSE.
C_SUB/DEC_F	The ALU function is MINUS when the condition is FALSE, MINUS_DEC when the condition is TRUE.
C_ADD/PASS_B_T	The ALU function is PLUS when the condition is TRUE, PASS_B when the condition is FALSE.
C_ADD/PASS_B_F	The ALU function is PLUS when the condition is FALSE, PASS_B when the condition is TRUE.

In addition to the explicit operations that microcode can specify with the ALU micro-orders, additional ALU functionality can be specified by some of the encodings in the RANDOM field of the microword. In particular, the PASS_A_HIGH AND PASS_B_HIGH RANDOM's cause the 64 bit ALU to perform as though it were two 32 bit ALU's sitting side by side with the following functionality. The "least significant" alu (i.e. the portion of the ALU operating on the 32 LSB's of the A_INPUT and B_INPUT) will perform the function specified by the ALU field of the microword, just as the normal 64 bit ALU would. The "most significant" alu, however, will perform the function PASS_A or PASS_B (depending on the RANDOM that is specified) on the most significant 32 bits of the A_INPUT and B_INPUT. An example of using this capability is in address generation. The upper 32 bits of the address (i.e. the module) can be passed through the ALU while the lower 32 bits (the offset of the address) can be appropriately manipulated.

One note about split ALU operation, when selecting a condition on the VAL board that is a function of the ALU output (e.g. A < B, MSB = 1 etc.), the entire 64 bits of the ALU participate in the generation of the condition. This means for instance that if you PASS_A when generating an address, you cannot test whether the lower 32 bits (the address offset), by themselves, equal zero.

Four other encodings in the RANDOM field may specify ALU functionality although currently none are used. Further discussion of ALU functionality is given in section 4.4 of this document.

Table 2-2: ALU Operations

Microword Field	Operation Name	Result
Arithmetic Operations		
0 0000	dec_a	F = A - 1
0 0001	plus	F = A + B
0 0010	plus_inc	F = A + B + 1
0 0011	left_1_a	F = A + A
0 0100	left_1_inc_a	F = A + A + 1
0 0101	minus_dec	F = A - B - 1
0 0110	minus	F = A - B
0 0111	inc_a	F = A + 1
0 1000	c_add/sub_t	CONDITIONAL
0 1001	c_add/sub_f	CONDITIONAL
0 1010	c_add/inc_t	CONDITIONAL
0 1011	c_add/inc_f	CONDITIONAL
0 1100	c_dec/sub_t	CONDITIONAL
0 1101	c_dec/sub_f	CONDITIONAL
0 1110	c_add/pass_b_t	CONDITIONAL
0 1111	c_add/pass_b_f	CONDITIONAL
Logical Operations		
1 0000	not_a	F = A~
1 0001	nand	F = (A and B)~
1 0010	not_a_or_b	F = A~ or B
1 0011	ones	F = -1 (2's comp)
1 0100	nor	F = (A or B)~
1 0101	not_b	F = B~
1 0110	xnor	F = (A xor B)~
1 0111	or_not	F = A or B~
1 1000	not_a_and_b	F = A~ and B
1 1001	xor	F = A xor B
1 1010	pass_b	F = B
1 1011	or	F = A or B
1 1100	pass_a	F = A
1 1101	and_not	F = A and B~
1 1110	and	F = A and B
1 1111	zeros	F = 0

2.3. Shift Mux

The Shift Mux is a device that selects one of four sources of data for storage into the C address of the register file. The four data paths that the Mux can select from are:

1. The unmodified output of the ALU.

2. The output of the ALU left shifted by one bit. In this case the MSB of the ALU output is shifted out (and therefore lost) and the least significant bit of the shifted result is zero filled.

3. The output of the ALU right shifted by 16 bits. In this case the least significant 16 bits of the ALU output are shifted out (and therefore lost) and the most significant 16 bits of the shifted result are zero filled.

4. The Write Data Register (WDR). This option is selected by the hardware when a START WRITE command has been issued and the location being written to resides in the Control Stack Accelerator (see section 2.1.2). The microcode should only select this option when the WDR needs to be saved in the RF as a piece of microstate.

2.4. Multiplier

The multiplier logic operates on two, unsigned 16 bit quantities (one from the A PORT of the RF the other from the B PORT) and produces a 32 bit unsigned product that can be used as an A INPUT to the ALU. Internally, the multiplier contains three registers: two to latch the 64 bit values from the A and B ports of the RF and one to latch the 32 bit product. These three registers provide the microcoder flexibility in selecting exactly which bits are to be multiplied and how to align the product.

The two values that are driven onto the A and B ports of the register file will be latched into the two multiplier input registers simultaneously when the RANDOM micro-order START MULTIPLY is invoked. Once two values get into these input registers they remain there until new values are loaded in. Each of the two 64 bit input registers is divided into four 16 bit quarters. Two microword fields, MULT A IN and MULT B IN, allow the microcoder to independantly decide for each register which 16 bit quarter-register should be used as the operands to the multiplier. The encodings of these fields is given in Section 3 of this document.

The cycle after a multiply is begun, the product is available either to be driven out to the FIU or to be used as an operand on the A INPUT of the ALU. Of course to access the multiplier product, the correct register file A ADDRESS encoding must be selected. When the 32 bit

multiplier output is selected to be used, several options exist as to how the output should be aligned within the 64 bit A PORT bus. The normal (default) mode is to have the multiplier product in the 32 LSB's of the bus with the upper 32 bits zero filled. Two other alignments of the product can occur by selecting one of the following VAL board RANDOMS:

LEFT_32_PRODUCT The product is left shifted 32 bits into <0..31> of the A INPUT. All other bits of the A INPUT are zero filled.

LEFT_16_PRODUCT The product is left shifted 16 bits into <16..47> of the A INPUT. All other bits of the A INPUT are zero filled.

The encodings of these two functions are given in Section 3 of this document. Choosing either of these two special alignments does not incur any time penalty and the shifted product can be used just as any normal ALU A INPUT. If one of these RANDOMS is specified but the multiplier product is not selected as the A INPUT to the ALU, then the RANDOM has no affect, operation of the VAL board logic proceeds as if it were not specified. Selecting one of these two RANDOMS is the only way to align the multiplier product in a non-standard format.

Additional discussion of the multiplier and its use in extended multiplications is given in section 4.4.2 of this document.

2.5. Zero Detector

The Zero Detector logic monitors the output of the ALU, generating testable conditions that indicate whether certain ranges of the ALU output are equal to zero. The conditions available for testing are:

1. All 64 bits of the ALU output = 0.
2. Most significant 32 bits of the ALU output = 0.
3. Most significant 48 bits of the ALU output = 0.
4. Bits <32:47>, i.e. the third most significant quarter of the ALU output = 0.

Each of the above conditions is available as a late condition in the cycle that it gets selected.

In addition to generating testable conditions, the COUNT ZEROS encoding in the RANDOM field tells the hardware to count and latch the number of leading zeros on the output of the ALU. The value of this number is available as an operand on the A INPUT of the ALU on the

next cycle following the COUNT ZEROS instruction and remains available until another one of these instructions is given. The format of the number of leading zeros that is driven onto the A INPUT is as follows. The numbers value is driven onto the six LSB's, i.e. bits <58:63>, and all of the remaining bits are zero filled by the hardware.

2.6. Loop Counter

The Loop Counter is a general purpose, 10 bit counter that can be used two different ways. First, the value in the loop counter can be used by the register file addressing logic to address any A, B, or C location in the RF. This allows the microcode to get around the restriction of only being able to address one of the 32 scratch registers that reside in the frame currently pointed to by the FRAME field of the microword. The second application of the loop counter value is as an operand to the A INPUT of the ALU. The 10 bit value is read out of the counter onto the 10 LSB's of the A INPUT while all of the remaining bits are zero filled by the hardware.

The value contained in the loop counter can be changed two ways. The first way is to directly parallel load a 10 bit value. This value may come from one of two places:

1. The 10 LSB's of the ALU can be directly loaded into the loop counter by selecting it with the C ADDRESS FIELD of the microword.

2. The value of the BOT register in the CSA addressing logic can be directly loaded into the 4 LSB's of the loop counter by specifying a RANDOM. In this case the upper 6 bits of the loop counter are set to zero.

The second way to change the value of the loop counter is to use the RANDOM micro-orders that specify INCREMENT or DECREMENT the loop counter (NOTE: The DIVIDE micro-order of the RANDOM field will also decrement the loop counter but this is implicit to that instruction and not under direct microcode control. Further explanation of the divide instruction is in section 4.4.1 of this document).

Finally, a testable condition generated by the hardware is set to 1 whenever the value of the loop counter equals zero. This is an early condition to the sequencer board. In the case where this condition is tested, and in the same cycle the instruction to increment (or decrement or load) the loop counter is issued, the test condition will be TRUE only if the pre-incremented value of the loop counter was zero.

2.7. Bus Interfaces

The VAL board interfaces with three of the five major processor busses: the VAL bus, the FIU bus, and the ADDRESS bus. The microcode control for determining when a particular board should drive data onto a bus resides on one of two boards. The SEQUENCER controls which board drives the address bus, and the FIU controls which boards drive the VAL and FIU busses. The interactions between the VAL board logic and each of these busses is described in the following sections.

2.7.1. VAL Data Bus

The principal point of access to the VAL data bus is the B PORT of the register file. Any piece of data on the VAL board that can be used as an operand on the B INPUT of the ALU can as well source data onto the VAL bus. Similarly, in any cycle, the data that is currently on the VAL bus can be used as a B INPUT to the ALU (or multiplier). The only other access to data on the VAL bus is through the copy of the WDR that resides on the board. In general, this register is present only for hardware timing reasons (involving memory writes that hit in the CSA) and should only be accessed by the microcode when storing the WDR as microstate.

There are not many restrictions to follow when interfacing with the VAL bus. The only ones currently are the following:

- The board cannot read data from the VAL bus and drive data to the bus in the same cycle.

- Whenever a memory read is made, by any board, to a control stack address space, in the same cycle of the read as READ RDR is specified the VAL bus MUST be specified as the B address of the register file. Whenever a write is made, by any board, to a control stack address space, in the second cycle after the START WRITE instruction is given the default (write disable) C ADDRESS of the RF must be specified.

- When executing a POP DOWN TO instruction on the CSA, in the cycle immediately after the one when the pop down address is put on the address bus the VAL bus MUST be specified as the B address of the register file.

2.7.2. FIU Bus

Data is driven onto the FIU bus from the A PORT of the register file and data received from the FIU bus can be stored into any location that can be addressed as a C address. The primary use of the FIU is to extract and align data that flows between processor memory and local storage on the VAL and TYPE boards. The previous statement implies the principal source of data that the VAL board receives over the FIU

bus is data that has come from main memory via the rotator and merger on the FIU. However, since the FIU bus appears to provide a very flexible data path between almost all of the boards in the processor, there is a posibility of assuming functionality in the FIU data path that does not exist. The following are the legal and illegal data paths to the VAL board over the FIU data bus:

LEGAL PATHS

1. Data coming from main memory over the VAL bus, going through the FIU to the FIU bus and getting stored directly in the VAL RF.

2. Data coming out of the A PORT of the TYPE board RF, over the FIU bus and getting stored directly in the VAL RF.

3. Data coming out of an isolated (i.e. non register file) processor register, going through the FIU to the FIU bus and getting stored directly in the VAL RF. Examples of isolated registers are Timer values on the SYSBUS board, SYSBUS status registers, MAR, and RDR.

ILLEGAL PATHS

1. Data coming out of a RF (either VAL or TYPE), going through the FIU to the FIU bus and getting stored back into the VAL RF.

2. Data, from <u>any</u> source, going through the FIU to the FIU bus, then going through the VAL ALU and getting stored into the VAL RF. (There is currently no way to generate this path under microcode control. It is included here for information purposes only).

3. Data coming from the TYPE RF across the FIU bus through the VAL ALU and getting stored in the VAL RF. (There is currently no way to generate this path under microcode control. It is included here for information purposes only).

4. Data coming out of the VAL RF, going over the FIU bus and through the FIU, then getting written into the WDR.

In addition to the legal data path functions described above, the receive path of the FIU has one other use: merging. When the appropriate RANDOM encoding is selected on the VAL board, the currently selected testable condition is "stuffed" into the LSB of the FIU bus receiver, the other bits of the FIU bus are unaffected. The principal use of this feature is when all zeros are driven on the FIU bus. This zero extends the selected condition and thus allows the microcode to store the boolean value of the selected condition in one cycle.

2.7.3. Address Bus

The address bus is driven by the output of the VAL board ALU. All addresses that are generated on this board are bit addresses, i.e. when an ALU output is an address, the seven least significant bits of that output specify which bit the data object of interest begins at within the the 128 bit word that is accessed. Since the main memory system only looks at word addresses, the seven bits of bit address are fed directly into the FIU to be used for extracting fields out of memory words.

For each address space defined in the R1000 architecture, the maximum offset into that address space is, in general, different from any other space. The microcode does not need to explicitly generate the correct number of leading zeros to drive onto the address bus for each different space. This detail is automatically done by the hardware by truncating the output of the ALU at the correct bit position for the particular address space and then zero filling.

3. Microword Specification

The following section summarizes the complete microword that controls the operation of the VAL board. The organization of this section specifies each field in the microword, the encoding and name of each micro-order within a field, and, when needed, a brief description of the function the micro-order performs. Since almost all of the micro-orders are referenced in Section 2, the reader should refer to the appropriate place in that section for a more detailed description of an encodings functionality.

Note, the name of each encoding in the microword is prefixed by a 'v'. This is used to distinguish between the VAL board and TYPE board microwords which to a large extent are identical.

V_RF_A (6 bits): specify the A address of the register file

ENCODING	NAME	FUNCTION
00xxxx	v_gp	select GP register xxxx
010000	v_tos+0	select current top of control stack
010001	v_tos+1	
010010	spare	
010011	v_reg(loop_counter)	select reg. pointed to by loop counter
010100	random state	
010101	zero_dtect	select output of zero_dtect
010110	product	select output of multiplier
010111	loop_counter	select output of loop counter
011000	v_tos-8	
011001	v_tos-7	
011010	v_tos-6	
011011	v_tos-5	
011100	v_tos-4	
011101	v_tos-3	
011110	v_tos-2	
011111	v_tos-1	
1xxxxx	v_reg(v_frame, xxxxx)	select register xxxxx in the frame pointed to by v_frame field

V_RF_B (6 bits): specify the B address of the register file.

00xxxx	v_gp
010000	v_tos+0
010001	v_tos+1
010010	spare
010011	v_reg(loop_counter)
010100	v_bot-1
010101	v_bot
010110	val_bus (CSA)
010111	spare
011000	v_tos-8
011001	v_tos-7
011010	v_tos-6
011011	v_tos-5
011100	v_tos-4
011101	v_tos-3
011110	v_tos-2
011111	v_tos-1
1xxxxx	v_reg(v_frame, xxxxx)

V_RF_C (6 bits): specify the C address of the register file

00xxxx	v_gp
010000	v_tos+0
010001	v_tos+1
010010	random state (write disable to RF)
010011	v_reg(loop_counter)
010100	v_bot-1
010101	v_bot
010110	write disable
010111	loop_counter (write disable to RF)
011000	v_tos-8
011001	v_tos-7
011010	v_tos-6
011011	v_tos-5
011100	v_tos-4
011101	v_tos-3
011110	v_tos-2
011111	v_tos-1
1xxxxx	v_reg(v_frame,xxxxx)

V_FRAME (5 bits): specify one of the 32 possible frames in the RF

xxxxx	frame

V_C_SRC (1 bit): specify which data source gets passed to the C PORT of the RF

0	v_c_fiu	FIU -> C address
1	v_c_mux	MUX -> C address

V_MUX (2 bits): specify the data source that the SHIFT MUX will pass to the C address

00	v_alu_left	ALU left shifted 1
01	v_alu	ALU unshifted
10	v_alu_right	ALU right shifted 16
11	v_wdr	WDR register

V_ALU (5 bits): specify the ALU function

00000	dec_a	F = A - 1
00001	plus	F = A + B
00010	plus_inc	F = A + B + 1
00011	left_1_a	F = A + A
00100	left_1_inc_a	F = A + A + 1
00101	minus_dec	F = A - B - 1
00110	minus	F = A - B
00111	inc_a	F = A + 1
01000	c_add/sub_t	conditional plus or minus
01001	c_add/sub_f	conditional minus or plus
01010	c_add/inc_t	conditional plus or plus_inc
01011	c_add/inc_f	conditional plus_inc or plus
01100	c_dec/sub_t	conditional minus or minus_dec
01101	c_dec/sub_f	conditional minus_dec or minus
01110	c_add/pass_b_t	conditional plus or pass_b
01111	c_add/pass_b_f	conditional pass_b or plus
10000	not_a	F = A~
10001	nand	F = (A and B)~
10010	not_a_or_b	F = A~ or B
10011	ones	F = -1 (2's comp)
10100	nor	F = (A or B)~
10101	not_b	F = B~
10110	xnor	F = (A xor B)~
10111	or_not	F = A or B~
11000	not_a_and_b	F = A~ and B
11001	xor	F = A xor B
11010	pass_b	F = B
11011	or	F = A or B
11100	pass_a	F = A
11101	and_not	F = A and B~
11110	and	F = A and B
11111	zeros	F = 0

MULT_A_IN (2 bits): specify which group of 16 bits is used as the multiplier A INPUT

00	v_multa_0	bits <0..15>
01	v_multa_16	bits <16..31>
10	v_multa_32	bits <32..47>
11	v_multa_48	bits <48..63>

MULT_B_IN (2 bits): specify which group of 16 bits is used as the multiplier B INPUT

00	v_multb_0	bits <0..15>
01	v_multb_16	bits <16..31>
10	v_multb_32	bits <32..47>
11	v_multb_48	bits <48..63>

V_RAND (4 bits): specify the described random operation

0000	no_op	
0001	inc_loop_cntr	
0010	dec_loop_cntr	
0011	st_mult	start multiply op
0100	bot_to_loop	BOT -> Loop counter
0101	count_zeros	count # leading zeros
0110	cond_to_fiu	selected condition -> LSB of FIU input
0111	left_32_product	product gets left shifted 32 bits
1000	left_16_product	product gets left shifted 16 bits
1001	pass_A_high	pass upper 32 bits of A INPUT to ALU
1010	pass_B_high	pass upper 32 bits of B INPUT to ALU
1011	divide	
1100	spare	
1101	spare	
1110	spare	
1111	spare	

V_CONDS (5 bits): specify the selected condition to be sent to the sequencer for processing. Selected condition also gets latched on the VAL board. The condition bit is set TRUE if the equation below is satisfied.

00000	zero	condition bit = 0
00001	alu_eq_z	64 bit ALU output = 0
00010	alu_ne_z	64 bit ALU output /= 0
00011	a_lt_b	a_alu < b_alu (signed)
00100	a_le_b	a_alu <= b_alu (signed)
00101	alu_co	64 bit alu carry out
00110	alu_of	64 bit alu overflow
00111	alu_lt_z	MSB of alu = 1 (ALU out < 0)
01000	alu_32_z	upper 32 bits of ALU out = 0
01001	alu_48_z	upper 48 bits of ALU out = 0
01010	alu_mid_z	bits <32:47> of ALU out = 0
01011	q_bit	
01100	loop_cntr_z	loop counter = 0
01101	v_last	last cycles VAL condition
01110	alu_le_z	64 bit ALU output <= 0
01111	spare	
10000	spare	
10001	spare	
10010	spare	
10011	spare	
10100	spare	
10101	spare	
10110	spare	
10111	spare	
11000	spare	
11001	spare	
11010	spare	
11011	spare	
11100	spare	
11101	spare	
11110	spare	
11111	one	condition bit = 1

TOTAL NUMBER OF BITS IN MICROWORD = 44

4. Microcode Considerations

The following section describes in more detail some aspects of the microcode interface to the VAL board. In general, the discussion in this section is directed toward three areas: complex microcode

processes (arithmetic operations, microstate saving and restoring), condition and event handling, and microcode restrictions that are imposed by the hardware.

4.1. Context Switch Microstate

The sum total of microstate that exists on the VAL board consists of:

- The Register File. The Control Stack Accelerator (CSA) and general purpose (GP) registers, in general, will need to be saved on every context switch along with some (small?) number of scratch pad registers. There is no hardware checking of which RF locations need to be saved as microstate. This must be totally kept track of by microcode. (N bits)

- The value contained in the loop counter (10 bits).

- The value of the number of leading zeros contained in the zero detector (6 bits).

- The previous cycles selected condition that gets latched on the VAL board (1 bit).

Access to all of this microstate for saving and restoring on context switch is very straightforward. All of the CSA, GP, and scratch registers that need to be saved can be addressed on the B port of the RF and immediately be saved or restored on the VAL data bus. All of the remaining bits of random state can be accessed through an A PORT address on the RF. This random state is packed into the least significant 17 bits of the word with the remaining bits zero filled by the hardware. The word can immediately be driven onto the FIU bus to get further packed into a "microstate" block (or whatever) that will be saved. When restoring state, these bits simply get written into the RANDOM STATE location of the RF and the hardware will unpack it appropriately.

4.2. Conditions

The VAL board generates 16 testable conditions, any one of which may be selected on a given cycle to be sent over to the microsequencer board for use in conditional sequencing operations. The conditions on this board can be divided into two types: alu conditions and non-alu conditions.

All alu conditions are designated as late conditions by the microsequencer and so can only be used either as hints or latched on the sequencer and tested in the next cycle. The following is a list of the alu conditions and a brief description of what the condition means.

ALU_EQ_Z — This condition is TRUE whenever the 64 bit ALU output equals zero. The ALU carry out and ALU overflow bits are not taken into consideration when generating this condition.

ALU_NE_Z — This condition is TRUE whenever the 64 bit ALU output does not equal zero. The ALU carry out and ALU overflow bits are not taken into consideration when generating this condition.

A_LT_B — This condition is TRUE when the A input of the ALU is less than the B INPUT. The comparison treats A and B as signed numbers (i.e. negative A is always less than positive B). To generate this condition the ALU must be executing the SUBTRACT instruction.

A_LE_B — This condition is TRUE when the A input of the ALU is less than or equal to the B INPUT. The comparison treats A and B as signed numbers (i.e. negative A is always less than positive B). To generate this condition the ALU must be executing the SUBTRACT instruction.

ALU_CO — This condition is TRUE when there is a carry out of the most significant bit of the ALU.

ALU_OF — This condition is TRUE when the result of an ALU operation overflows a 64 bit representation.

ALU_LT_Z — This condition is TRUE when the MSB (sign bit) of the ALU = 1. This is equivalent to testing whether the 64 bit ALU output < 0.

ALU_LE_Z — This condition is true whenever the 64 bit ALU output <= 0. This condition is logical or of the ALU_EQ_Z and the ALU_LT_Z conditions.

ALU_32_Z — This condition is TRUE when the upper 32 bits (i.e. bits <0..31>) of the ALU output = 0.

ALU_48_Z — This condition is TRUE when the upper 48 bits (i.e. bits <0..47>) of the ALU output = 0.

ALU_MID_Z — This condition is TRUE when bits <32..47> of the ALU output = 0.

All non-alu conditions are designated as early conditions by the microsequencer and so can be used either as branch conditions in the current microcycle or latched on the sequencer and tested in the next cycle. The following is a list of the non-alu conditions and a brief description of what the condition means.

ZERO — When this condition is selected a zero is driven out on the VAL board condition wire.

ONE — When this condition is selected a one is driven out on the VAL board condition wire.

Q_BIT — This condition is TRUE when the Q bit on the VAL board = 1. The Q bit is a condition used by microcode during a divide operation that determines the outcome of the conditional add/subtract that needs to be done by the ALU. For a more detailed discussion of the Q bit, and the divide operation in general, see Section 4.4.1 of this document.

LOOP_CNTR_Z — This condition is TRUE when the value of the loop counter = 0. It is possible that in the same cycle that the value of the loop counter is being tested, the instruction to increment (or decrement or load) the loop counter is issued. In this case, the test condition will be TRUE only if the pre-incremented value of the loop counter was zero

V_LAST — At the end of every microcycle, the condition that was selected on the VAL board gets latched on the VAL board (this is different from the condition latch on the microsequencer board). During any microcycle, then, it is possible using the v_last instruction to select as a condition the value of the condition that was latched on the VAL board during the previous cycle. This selection is primarily provided for hardware diagnostics, however, it is made available to the microcode as a selectable condition also.

4.3. Events

There are no micro-events or macro-events generated by the VAL board.

4.4. Special Arithmetic Operations

Each of the following sections describes the details of the hardware support of the more complicated arithmetic functions that are handled by the VAL board. For each section, the reader is refered to Section 2 of this document for a description of how these operations fit into the functionality of the hardware as a whole.

4.4.1. Divide

The divide operation is implemented in microcode with some specific hardware support built into the VAL board. The goal of dedicating hardware support is to allow a standard non-restoring algorithm that executes a divide in approximately the same number of cycles as the number of significant bits of quotient.

Three pieces of hardware logic are provided as hardware support: the Q bit, the ALU_LT_Z bit, and the leading zero counter of the ZERO DETECTOR. The leading zero counter has been described in section 2.5 of this document but a brief description will also be given here. Essentially, the number of leading zeros of the ALU output can be counted at any time (subject to the restrictions of section 4.5 of this document) by selecting the RANDOM micro-order COUNT ZEROS on the VAL board. On the next cycle after counting, the number of leading zeros can be accessed by an A PORT address of the RF and may be used in an ALU operation or sent to the FIU. For the divide instruction, the number of leading zeros of both the dividend (numerator) and the divisor (denominator) are counted to determine the number of significant bits of quotient that will be produced for the current division (Quotient Bits = Leading zeros of Denom. minus Leading zeros of Num.). As mentioned before, since the number of quotient bits determines the number of iterations in the divide loop, once this number is computed it can be loaded into the loop counter and be used to determine when to end the divide operation.

The ALU_LT_Z bit is simply a copy of the MSB of the ALU output that gets latched for use in determining the value of the Q bit in the next cycle of the divide.

The Q bit is a condition whose value is determined by the following:

```
Q BIT = ALU carry out          --- if no divide is in progress.
                                   This is the initial value the Q
                                   bit needs to begin a divide.

Q BIT = (Q BIT xor ALU carry out xor ALU_LT_Z)~ -- when divide
                                                   in progress
```

On the VAL board, for each iteration in the non-restoring divide algorithm, the denominator conditionally gets either added to or subtracted from the numerator, depending on the value of the Q bit. The numerator then gets shifted left by one, the loop counter gets decremented by one, and the Q bit gets sent over to the TYPE board and

gets shifted in as the least significant bit of the quotient to conclude the iteration. This process is repeated until the loop counter reaches zero, at which point the quotient is complete and resides in a register on the TYPE board.

To simplify the microcode interface , some of the above operations are performed automatically by the hardware when the microcode selects the DIVIDE instruction in the VAL boards RANDOM field. During each cycle of the principal loop in the divide algorithm the DIVIDE random and the C_ADD/SUB_F conditional ALU instruction should be specified. For each of these cycles the hardware will:

1. Decrement the loop counter.

2. Use the current Q bit to select either the add or subtract function for the VAL board ALU (Q bit equal zero => add).

3. Provide a data path between the Q bit (generated on the VAL board) and the carry in input of the TYPE board ALU so that the quotient bit can get shifted into the LSB of that ALU on each cycle.

All of the other details of the divide algorithm are left to the specific microcode algorithm that gets chosen. One final note about divide: the values of the Q bit and the ALU_LT_Z bit are not currently packed into the bits of random state that get saved and restored on context switch. It is possible to include these bits later, however it is not thought to be necessary at this time. This does imply that care should be taken by the microcode to make sure that these values never need to be saved.

4.4.2. Multiply

In general, there are two types of multiplies that need to be done: those necessary for array index calculations and those that are just regular multiplications. In terms of speed, array indexing operations are a much higher priority and so the hardware is optimized for this case.

It is anticipated that a very high percentage of all array indexing multiplications will have operands whose values are less than 16 bits (It is possible for these operands to be up to 32 bits long and so they must be explicitly checked to see that they fall in the 16 bit range). The VAL board multiplier is optimized for the case of two 16 bit operands, in this case the entire multiply and accumulate operation needed for array indexing can be done in only two cycles. In the first cycle, the two 64 bit operands are latched into the multiplier input registers by selecting the RANDOM instruction ST_MULTIPLY. At the same time, the correct 16 bits of each register can be passed to the multiplier logic by choosing the appropriate micro-orders of the MULT_A_IN and MULT_B_IN microword fields. In the

second cycle, the 32 bit product is available to be added with the array's base offset to complete the array index operation. The product can be accessed by specifying the PRODUCT address of the A PORT of the RF.

When doing an extended multiply (i.e. the input values are between 16 and 64 bits), the multiplier is used to produce 32 bit partial products (at a rate of one partial product per cycle) which must all be accumulated together to form the final product. This type of multiply is begun in the same way as a 16 by 16 multiply, by latching the two 64 bit operands in the multiplier latches with the ST_MULTIPLY instruction and choosing the desired 16 bit multiplier inputs with the MULT_A_IN and MULT_B_IN fields. In the next cycle, the first partial product is available at the output of the multiplier. This partial product is accessed through the A ports PRODUCT address and can either be passed through the ALU, or combined with some other offset and then be stored in some scratch location. The partial product does not have to be accessed immediately, it will remain in the output latch of the multiplier until two new inputs are chosen to be multiplied together. In the same cycle that the first partial product first becomes available, the next 16 bit operands can be selected from the 64 bit input latches to generate the second partial product. On the following cycle the second partial product is available at the output of the multiplier and the third set of 16 bits can be chosen for generating the third partial product. This loop of generating partial products is repeated until the full multiplication is completed. The full details of the algorithm are left up to the microcode.

When each partial product becomes available, it must be accumulated with the previous partial products to obtain the final result. Also, each partial product must be shifted left by the proper amount before it can be added in. This left shift operation is built into the hardware and is used by selecting one of two RANDOM operations: LEFT_16_PRODUCT and LEFT_32_PRODUCT. When either of these two RANDOM instructions is chosen, the LSB of the multiplier output is shifted up to bit 47 (left shift 16) or bit 31 (left shift 32) before it is made available to the ALU input. All other bits are zero filled by the hardware on both of these shifts to allow immediate addition with previous partial products.

Two final notes on multiplication. First, as previously mentioned, the multiplier has been optimized to perform a 16 by 16 multiply as quickly as possible for array indexing. Since array index calculations are always done on unsigned numbers, this is the only capability built into the multiplier. If two signed numbers need to be multiplied, either some pre-processing or post-processing (or both) must be done by the microcode such that only an unsigned multiply is necessary.

Second, none of the three registers in the multiplier are available to be saved or restored as microstate. This implies that some care must be taken by the microcode to make sure that these values never need to be saved.

4.4.3. Floating Point Operations

There is no dedicated hardware support for floating point operations on the VAL board. All floating point operations will be implemented either directly by microcode using the existing functionality of the VAL, TYPE and FIU boards, or by software.

4.5. Microcode Restrictions

This section summarizes all of the known restrictions that the hardware imposes (mainly for timing reasons) on the microcode. Most of these restrictions were discussed in previous sections of this document and therefore the reader is refered, mainly to Section 2, for further details of each of these restrictions.

CSA RESTRICTIONS

The following restrictions are imposed by the CSA.

1. Whenever a memory read is made to a Control Stack address space, during the cycle when the READ RDR instruction is issued the only legal B ADDRESS that may be specified for the VAL board RF is the VAL_BUS (CSA) address. Whenever a memory write is made to a Control Stack address space, in the second cycle after the START WRITE instruction is issued the only legal C ADDRESS that may be specified for the VAL board RF is the default (write disable) address. This is because it is impossible to determine a priori whether a given control reference will "hit" in the CSA and therefore necessitate the VAL and TYPE boards accessing the CSA instead of memory.

2. When executing a POP DOWN instruction, the cycle immediately after the address being popped down to is driven onto the ADDRESS bus, the only legal B ADDRESS that may be specified for the VAL board RF is the VAL_BUS (CSA) address. This restriction is imposed by hardware timing.

3. One note of caution when executing Control Stack POP or POP_DOWN instructions. The hardware calculates whether a Control Stack address hits the CSA in the same cycle that the MAR is loaded with the address. Let us say in this example that a particular control read hits the CSA at the top of

stack minus two (TOP-2) location. Since the data from this read is not driven onto the VAL bus until the READ RDR command is issued (possibly many cycles later) it is possible for the microcode to POP or POP DOWN the top of the stack to a point below where the address hit in the CSA. In this case when the READ RDR command is issued, valid data cannot be guaranteed since the CSA will be reading a location above the current top of stack. The hardware does not protect against this scenario. It is up to the microcoder to exercise restraint in using POP and POP DOWN instructions in this type of situation.

FIU RESTRICTIONS

The following restrictions on driving data onto the FIU bus are imposed by hardware timing.

1. Data coming from the VAL board RF cannot go across the FIU bus, through the FIU and then get stored back into the VAL RF.

2. Data, from <u>any</u> VAL board source, cannot go through the FIU, come back across the FIU bus, then go through the VAL ALU and get stored into the VAL RF. (There is currently no way to generate this path under microcode control. It is included here for information purposes only).

3. Data cannot come from the TYPE RF across the FIU bus, through the VAL ALU and get stored in the VAL RF. (Similarly VAL data cannot get stored in the TYPE RF in this manner) (There is currently no way to generate this path under microcode control. It is included here for information purposes only).

4. Data cannot come out of the VAL RF, go over the FIU bus and through the FIU, then get written into the WDR.

OTHERS

The following are all of the other restrictions imposed by the VAL board hardware.

1. The COUNT ZEROS instruction cannot be used to count the number of leading zeros of the VAL ALU output when the B INPUT to the ALU is coming from the VAL BUS (CSA) address of the RF. This is because of hardware timing.

5. Diagnostics

This section, and all of the subsections that it contains, will not be a part of the initial specification of this board. Rather they will be added later as more of the specific details become known. The outline of this section is included at this point for the sake of completeness, and to elicit suggestions as to what the content and format of each section should be.

5.1. Philosophy

5.2. Hardware Support

5.2. Stand Alone Testing

5.4. System Integration Testing

5.5. Micro-Diagnostics

6. Hardware Considerations

This section, and all of the subsections that it contains, will not be a part of the initial specification of this board. Rather they will be added later as more of the specific details become known. The outline of this section is included at this point for the sake of completeness, and to elicit suggestions as to what the content and format of each section should be.

6.1. Timing Issues

6.1.1. Data Path Timing

6.1.2. Clocking Issues

6.1.3. Potential Problems and Restrictions

6.2. Chip Count and Power Estimates

6.3. System Interconnections

6.3.1. Foreplane

6.3.2. Backplane

6.4. Layout

Functional Specification of the Type Board

DRAFT 1

People exaggerate the things they've never had, they admire types because they have no experience with them.

- George Bernard Shaw

Rational Machines proprietary document.

1. Introduction

This document describes the functionality of the Type board of the R1000. The specification defines in detail the microcode and hardware interfaces to the board. The reader is assumed to be familiar with both the R1000 architecture and the specifications of the other boards in the R1000 processor.

The type board is extremely similar to the value board. The similarities are designed in to allow the microcode to have the resources of two 64 bit CPU's operating in parallel. This point can not be stressed enough! TWO 64 BIT CPU'S OPERATING IN PARALLEL. Because of this fact the functionality, hardware, and microcode, for the two boards are extremely similar.

In general the differences are the value board has a zero detector, a multiplier, and a shift mux and the type board doesn't. While the type board has some checking circuitry (privacy and class) and the value board doesn't.

This spec will only explain the sections of the type board that are drastically different from the value board. In most cases these differences are obvious from the differences in the microword.

2. Block Diagram Functional Definition

This section refereces the block diagram of the Type board attached to this document. The functionality of each block in the diagram is discussed in detail in the following sections.

2.1. Register File

Same as the value board except there is no zero detector or multiplier.

2.1.1. Register File Addressing

Same as the last comment.

2.1.2. Control Stack Accelerator

Identical to the value board.

2.2. ALU

The ALU control and operation is exactly the same as the value ALU, except there are no conditional ALU operations. (See the microword specification for the exact ALU control available.)

The random field of the type microword allows the selection of the Q bit (a bit supplied by the value board over the backplane) as the carry-in to the ALU. (See the random field of the microword.) When this random micro-order is not selected the T_ALU field of the type microword determines the carry-in to the ALU.

2.3. Mux

The type Mux determines the source of data for storage into the C address of the register file. The two data paths that the Mux can select are:

1. The unmodified output of the ALU.

2. The Write Data Register (WDR). This option is selected by the hardware when a START WRITE command has been issued and the location being written to resides in the Control Stack Accelerator (see the previous CSA section). The microcode should only select this option when the WDR needs to be saved in the RF as a piece of microstate.

(This mux operates differently than the corresponding mux on the value board.)

2.4. Checker

The checker circuitry on the type board can be divided into three function units.

1. Privacy Checker -- does a first level privacy check on one or two operands.

2. Class Check -- checks class compatability of one or two operands.

3. Of_Kind conditions -- detects special type conditions.

The privacy and class check can cause micro events, and are testable as conditions.

2.4.1. Privacy Checker

The privacy checker is used to check if the operand(s) under test is(are) in the scope of privacy. The check facilities are mostly used for scalars, but additional testable conditions are available to test for structures. The privacy checker has a 32 bit outer_frame_name register that is loadable from bits (0:31) of the B_bus. This register must be reloaded during every context switch (and during some instructions like call and exit). To use the privacy checker the type-links of the control_stack operands(s) must be on the A_bus and/or B_bus. The check is selected from the privacy check field of the microcode.

The privacy checker can perform the following five checks:

1. Bin_eq -- Privacy check for equality and assignment. The operands on both the A and B bus are checked.

2. Bin_op -- Privacy check for a binary operation. The operands on both the A and B bus are checked.

3. A_op -- Privacy check for a unary operation. The operand on the A_bus is checked.

4. B_op -- Privacy check for a unary operation. The operand on the B_bus is checked.

5. Names_equal -- Binary check for the same path names on both the A and B bus.

Using the following identifiers the privacy checks can be accurately expressed as boolean equations.

```
          o_f = outer_frame_name register(0:31)
       A_name = A_bus(0:31)
       B_name = B_bus(0:31)
   A_is_priv = A_bus(34)
   B_is_priv = B_bus(34)
  A_drv_priv = A_bus(35)
  B_drv_priv = B_bus(35)

Names_same := (A_name = B_name)

A_op := (A_drv_priv) V (A_is_priv)(o_f = A_name)~

B_op := (B_drv_priv) V (B_is_priv)(o_f = B_name)~

Bin_op := (A_op) V (B_op)

Bin_eq := (Bin_op)(Names_same)~
```

(The last four equations indicate that the first level check fails if the equation is true.)
The privacy checker control logic can enable one of six possible privacy checks during a micro-instruction. These six different privacy checks can generate one of four possible micro events. The following table indicates the correspondence between the privacy check enables and the micro events generated. (See the privacy_check field of the microword for the details on enabling a privacy micro event.)

Privacy Check	Micro Event Generated
Bin_eq	Bin_eq
Bin_op	Bin_op
A_[TOS]_op	[TOS]_op
A_[TOS-1]_op	[TOS-1]_op
B_[TOS]_op	[TOS]_op
B_[TOS-1]_op	[TOS-1]_op

Table 2-1: Privacy Micro Event Generation

(The hardware doesn't check if a operand is [TOS] or [TOS-1]. Therefore there is no guarantee that the correct micro-event is taken if the microcode is incorrect.)

All of the privacy events are early and non-persistent. Since they are early events the instruction that caused the event will be re-executed if the micro event handler returns. To prevent the privacy event from re-occuring, when the privacy event handler returns, a "pass privacy state" exists in the logic. If the micro handler

decides that the operand(s) passes the additional privacy check in the handler, the handler should set the "pass privacy check" state. This state is set by specifying the "pass privacy check" order of the privacy check micro-field. If a privacy micro event is enabled and the pass privacy state is set, the micro event doesn't occur and the pass privacy state is cleared. The pass privacy state can also be cleared from the random field. (NOTE: The pass privacy state has NO effect on the privacy test conditions.) The pass privacy state must be cleared during context switches, but does not need to be restored (the event will occur again). (NOTE: The pass privacy state will not change when the privacy check micro order is a "nop", unless the random field clears the state.)

2.4.2. Class Check

The Class checker is used to compare the of_kind bits of a type_link on the A_bus or B_bus to the other bus and/or to a literal. This check provides a parallel mechanism for ensuring that operand(s) are of the correct or same type for a specific instruction.

The class check hardware is capable of 3 different 7 bit compares. The conditions and events that can be generated are shown in the following table.

Hardware Test	Micro event	Condition
A_bus(57:63)=class_lit	X	X
B_bus(57:63)=class_lit	X	X
A_bus(57:63)=B_bus(57:63)	X	X
A_bus(57:63)=B_bus(57:63)=class_lit	X	X

Table 2-2: Class Checks

Only one class check micro event can be enabled during any particular micro-instruction. The random field of the type microword selects which class micro event is enabled (see the microword specification section). All of the class micro events are early events which prevent the current instruction from completing. (All of the class micro checks cause the SAME class event and branch to the same micro event handler.) Any of the above conditions can also be selected as the currently tested processor condition (see the microword specification section).

(Implementation Note: In an effort to reduce the width of the

microword the frame bits of the type microword have been overloaded with five bits of the class lit. The class lit is seven bits. The most significant two bits are a field in the microword. The least significant five bits are overload with the frame microbits. If a micro-instruction uses both a frame address and a class check with a class lit, the frame address must be the same as the least significant five bits of the class lit. (For more details see the microword specification section.))

2.4.3. Of Kind condition

The checker circuitry also provides a special test condition that can be used for "subrange" detection on the "of_kind" encodings. The hardware has 64 patterns programmed into two proms. (The class lit is overloaded to choose the patterns.) The first prom contains a seven bit pattern which is compared to bits (57:63) of the B_bus. The second prom contains a seven bit mask which indicates which bits are to be compared. The output of the comparator is one of the selectable conditions on the type board. The following table lists some of the patterns that are currently included in the test conditions.

PATTERN NUMBER (hex)	PATTERN NAME	BIT PATTERN (57:63)
00	TYPED	XXXXX00
01	IMPORT	XXXX0X0
02	VALUE	XXXX000
03	SCALAR	000X000
04	INDIRECT	XXXX100
05	VALUE REF	0XX0100
06	STRUCTURE	1XX0100
07	SUBINDEXED	11XX100
08	REFERENCE	XXXX010
09 - 1F	unused	
20	STATE_WORD	XXXX001
21	CONTROL_KEY	XXXX101
22	MARK_WORD	XXXX111
23 - 3F	unused	

(The "X" bits are not compared.)

Table 2-3: Of_Kind condition

(NOTE: The checker circuit also provides tests for some of the

individual bits on the B_bus. These test conditions are very useful for some type checking and are enumerated in the microword specification section under the condition field.)

2.5. Loop Counter

Identical to the value board.

2.6. Bus Interfaces

The type board bus interfaces are exactly the same as the value board, except the type board connects to the TYPE bus where the value board connects to the VALUE bus. (This section and the following three sections, are identical to the value board spec if "TYPE" is substituted for "VALUE".)

2.6.1. VAL Data Bus

2.6.2. FIU Bus

2.6.3. Address Bus

3. Microword Specification

T_RF_A (6 bits): specify the A address of the register file

ENCODING	NAME	FUNCTION
00xxxx	t_gp	select GP register xxxx
010000	t_tos+0	select current top of control stack
010001	t_tos+1	
010010	spare	
010011	t_reg(loop_counter)	select reg. pointed to by loop counter
010100	random state	outer_frame_name, pass_privacy bit, loop_counter
010101	spare	
010110	spare	
010111	loop_counter	select output of loop counter
011000	t_tos-8	
011001	t_tos-7	
011010	t_tos-6	
011011	t_tos-5	
011100	t_tos-4	
011101	t_tos-3	
011110	t_tos-2	
011111	t_tos-1	
1xxxxx	t_reg(t_frame, xxxxx)	select register xxxxx in the frame pointed to by t_frame field

T_RF_B (6 bits): specify the B address of the register file.

00xxxx	t_gp
010000	t_tos+0
010001	t_tos+1
010010	spare
010011	t_reg(loop_counter)
010100	t_bot-1
010101	t_bot
010110	type_bus (CSA)
010111	spare
011000	t_tos-8
011001	t_tos-7
011010	t_tos-6
011011	t_tos-5
011100	t_tos-4
011101	t_tos-3
011110	t_tos-2
011111	t_tos-1
1xxxxx	t_reg(t_frame, xxxxx)

T_RF_C (6 bits): specify the C address of the register file

00xxxx	t_gp
010000	t_tos+0
010001	t_tos+1
010010	random state (write disable to RF) outer_frame_name, pass privacy bit, loop_counter
010011	t_reg(loop_counter)
010100	t_bot-1
010101	t_bot
010110	write disable
010111	loop_counter (write disable to RF)
011000	t_tos-8
011001	t_tos-7
011010	t_tos-6
011011	t_tos-5
011100	t_tos-4
011101	t_tos-3
011110	t_tos-2
011111	t_tos-1
1xxxxx	t_reg(t_frame, xxxxx)

T_FRAME (5 bits): specify one of the 32 possible frames in the RF
(This field is overloaded with five bits of the class literal and five bits of the type Of_Kind condition number.)

xxxxx	frame, class literal (2:6), Of_Kind condition_number(2:6)

T_C_SRC (1 bit): specify which data source gets passed to the C PORT of the RF

0	t_c_fiu	FIU -> C address
1	t_c_mux	MUX -> C address

T_MUX (1 bit): specify the data source that the MUX will pass to the C address

0	t_alu	ALU
1	t_wdr	WDR register

T_ALU (5 bits): specify the ALU function

00000	dec_a	F = A - 1
00001	plus	F = A + B
00010	plus_inc	F = A + B + 1
00011	left_1_a	F = A + A
00100	left_1_inc_a	F = A + A + 1
00101	minus_dec	F = A - B - 1
00110	minus	F = A - B
00111	inc_a	F = A + 1
01000	spare	
01001	spare	
01010	spare	
01011	spare	
01100	spare	
01101	spare	
01110	spare	
01111	spare	
10000	not_a	F = A~
10001	nand	F = (A and B)~
10010	not_a_or_b	F = A~ or B
10011	ones	F = -1 (2's comp)
10100	nor	F = (A or B)~
10101	not_b	F = B~
10110	xnor	F = (A xor B)~
10111	or_not	F = A or B~
11000	not_a_and_b	F = A~ and B
11001	xor	F = A xor B
11010	pass_b	F = B
11011	or	F = A or B
11100	pass_a	F = A
11101	and_not	F = A and B~
11110	and	F = A and B
11111	zeros	F = 0

T_RAND (4 bits): specify the described random operation

0000	no_op	
0001	inc_loop_cntr	
0010	dec_loop_cntr	
0011	carry_in_Q	Carry_in = Q bit from val
0100	spare	
0101	write outer_frame_name	
0110	clear pass privacy state	
0111	spare	
1000	spare	
1001	pass_A_high	pass upper 32 bits of A INPUT to ALU
1010	pass_B_high	pass upper 32 bits of B INPUT to ALU
1011	class_check(A_bus, lit)	
1100	class_check(B_bus, lit)	
1101	class_check(A_bus, B_bus)	
1110	class_check(A_bus, B_bus, lit)	
1111	spare	

T_CLASS_LIT (2 bits): specify the literal that is compared against during most of the class checks (the other five bits of the class literal are overloaded with the frame). (This field is over loaded with the 2 msb's of the Of_Kind condition number.)

XX The 2 msb's of the class literal or the 2 msb's of the Of_Kind condition number.

PRIVACY_CHECK (3 bits): This field enables one of the four privacy privacy micro events.

000	check privacy for equality
001	check privacy for A_bus and B_bus
010	check privacy for [TOS] on the A_bus
011	check privacy for [TOS-1] on the A_bus
100	check privacy for [TOS] on the B_bus
101	check privacy for [TOS-1] on the B_bus
110	nop
111	set "pass privacy state"

T_CONDS: specify the selected condition to be sent to the sequencer for processing. Selected condition also gets latched on the TYPE board. The condition bit is set TRUE if the equation below is satisfied.

0011000	alu_eq_z	64 bit ALU output = 0
0011001	alu_ne_z	64 bit ALU output /= 0
0011010	a_gt_b	a_alu > b_alu (signed)
0011011	a_ge_b	a_alu >= b_alu (signed)
0011100	loop_cntr_z	loop counter = 0
0011101	spare	
0011110	spare	
0011111	spare	
0100000	alu_co	64 bit alu carry out
0100001	alu_of	64 bit alu overflow
0100010	alu_lt_z	MSB of alu = 1 (ALU out < 0)
0100011	alu_le_z	64 bit ALU output <= 0
0100100	t_last	last cycle's TYPE condition
0100101	spare	
0100110	one	condition bit = 1
0100111	zero	condition bit = 0
0101000	Of_Kind(##)	Of_Kind condition
0101001	spare	
0101010	class(A,1)	class_check(A_bus, lit)
0101011	class(B,1)	class_check(B_bus, lit)
0101100	class(A,B)	class_check(A_bus, B_bus)
0101101	class(A,B,1)	class_check(A_bus, B_bus, lit)
0101110	privacy(A)	privacy_check(A_bus)
0101111	privacy(B)	privacy_check(B_bus)
0110000	privacy(equal)	privacy_check(equality)
0110001	privacy(A,B)	privacy_check(bin_op)
0110010	privacy(names)	A_bus.path_name = B_bus.path_name
0110011	privacy(struc.)	both privacy(A,B) and privacy(names)
0110100	B_bus(32)	bit 32 of the B_bus
0110101	B_bus(33)	bit 33 of the B_bus
0110110	B_bus(34)	bit 34 of the B_bus
0110111	B_bus(35)	bit 35 of the B_bus

0111000	B_bus(36)	bit 36 of the B_bus
0111001	B_bus(34 or 36)	B_bus(34) OR B_bus(36)
0111010	spare	
0111011	spare	
0111100	spare	
0111101	spare	
0111110	spare	
0111111	spare	

TOTAL NUMBER OF BITS IN THE MICROWORD = 39

4. Microcode Considerations

The following subsections detail microcode constraints, conditions, and events.

4.1. Context Switch Microstate

The microstate that exists on the TYPE board consists of:

1. The Register File. The Control Stack Accelerator (CSA) and general purpose (GP) registers, in general, will need to be saved on every context switch along with some number of scratch pad registers. There is no hardware checking of which RF locations need to be saved as microstate.

2. The Outer_frame_name (in the checker). The outer_frame_name register only needs to be loaded for the incoming tasks. (The outer_frame must be saved on every context switch, but presumably the full outer_frame from the sequencer is written out during the context switch.)

3. The loop counter.

4. The pass privacy bit. Since privacy is an early event, it should be o.k. to NOT save the pass privacy bit during a context switch. Then the bit only needs to be CLEARED during each context switch.

4.2. Conditions

The TYPE board generates 32 testable conditions. The conditions can be divided into three groups L - late, ML - meduim late, and E - early. The early conditions can be used as conditions for conditional branch types, and don't require a hint. The meduim late and late conditions require hints if used with conditional branch types. Only the early or medium late conditions can be used as conditions for

conditional memory references. And believe it or not, every condition can be latched in the microsequencer's latch.

ALU_EQ_Z (L)
This condition is TRUE whenever the 64 bit ALU output equals zero. The ALU carry out and ALU overflow bits are not taken into consideration when generating this condition.

ALU_NE_Z (L)
This condition is TRUE whenever the 64 bit ALU output does not equal zero. The ALU carry out and ALU overflow bits are not taken into consideration when generating this condition.

A_GT_B (L)
This condition is TRUE when the A input of the ALU is greater than the B INPUT. The comparison treats A and B as signed numbers (i.e. negative A is always less than positive B). To generate this condition the ALU must be executing the SUBTRACT instruction.

A_GE_B (L)
This condition is TRUE when the A input of the ALU is greater than or equal to the B INPUT. The comparison treats A and B as signed numbers (i.e. negative A is always less than positive B). To generate this condition the ALU must be executing the SUBTRACT instruction.

ALU_CO (L)
This condition is TRUE when there is a carry out of the most significant bit of the ALU.

ALU_OF (L)
This condition is TRUE when the result of an ALU operation overflows a 64 bit representation.

ALU_LT_Z (L)
This condition is TRUE when the MSB (sign bit) of the ALU = 1. This is equivalent to testing whether the 64 bit ALU output < 0.

ALU_LE_Z (L)
This condition is true whenever the 64 bit ALU output <= 0. This condition is logical or of the ALU_EQ_Z and the ALU_LT_Z conditions.

T_LAST(L) (L)
At the end of every microcycle, the condition that was selected on the TYPE board gets latched on the TYPE

board. During any microcycle this latched condition can be selected as a testable condition. (This condition is available mostly as a diagnostic feature rather than a useful microcode feature. No provisions are made to keep the value of this latch consistent, across events or context switches.)

TRUE (E)
This condition is always true.

FLASE (E)
This condition is always false.

OF_KIND(#) (ML)
This condition is the result of the Of_Kind condition test. The number specified is used to select the pattern to match.

CLASS(A,LIT) (ML)
This condition is true if the class literal is equal to bit (57:63) of the A_bus.

CLASS(B,LIT) (ML)
This condition is true if the class literal is equal to bits (57:63) of the B_bus.

CLASS(A,B) (ML)
This condition is true if bits (57:63) of the A_bus are equal to bits (57:63) of the B_bus.

CLASS(A,B,LIT) (ML)
This condition is true if bits (57:63) of the A_bus are equal to bits (57:63) of the B_bus and equal to the class literal.

PRIVACY_A (ML)
This condition is true if the first order privacy check on the A_bus passes.

PRIVACY_B (ML)
This condition is true if the first order privacy check on the A_bus passes.

PRIVACY_EQ (ML)
This condition is true if the first order privacy for equality check passes.

PRIVACY_A&B (ML)
This condition is true if the first order privacy check for both A and B busses pass.

PRIVACY_NAMES (ML)

This condition is true if bits (0:31) of the A_bus equal bits (0:31) of the B_bus.

PRIVACY_STRUCT. (ML)
This condition is true both PRIVACY_A&B and PRIVACY__NAMES are true.

B_BUS(32) (ML)
This condition is true if and only if bit 32 of the B_bus is a one.

B_BUS(33) (ML)
This condition is true if and only if bit 33 of the B_bus is a one.

B_BUS(34) (ML)
This condition is true if and only if bit 34 of the B_bus is a one.

B_BUS(35) (ML)
This condition is true if and only if bit 35 of the B_bus is a one.

B_BUS(36) (ML)
This condition is true if and only if bit 36 of the B_bus is a one.

B_BUS(32_OR_36) (ML)
This condition is true if either bit 32 or bit 36 of the B_bus are one.

4.3. Events

The type board can generate five micro events and no macro events. All of the micro events are early and non-persistent, therefore there are no mask bits for the events (they are each enabled by specific micro-orders). The micro events are class_check, Bin_eq, Bin_op, [TOS]_op, and [TOS-1]_op. The micro events are explained in detail in previous sections.

4.4. Microcode Restrictions

The microcode restrictions for the TYPE board are exactly the same as the CSA RESTRICTIONS and FIU RESTRICTIONS from the value spec. Please consult the value spec for the details.

Table of Contents

List of Tables

R1000 Microsequencer Specifications

Draft 2

Rational Machines proprietary document.

1. Summary

This document describes the functionality of the Microsequencer board of the R1000. The specification defines in detail the microcode and hardware interfaces to the board. The reader is assumed to be familiar with both the R1000 architecture and the specifications of the other boards in the R1000 processor.

2. Functional Description

The microcode controlling the operation of the R1000 is physically separated on different boards in the processor, but all of the boards operate in a lock-step fashion. The order of execution of the micro-instructions in the R1000 is determined by the microsequencer.

2.1. Branches

The (BRANCH_TYPE) field of the microword determines how the next micro- address is selected. The (BRANCH_TYPE) field also determines if the next micro-address selection is conditional or unconditional. (The condition under test is selected by the (CONDITION) field of the microsequencer microword.)

The (BRANCH_ADDRESS) field in the microword is an absolute branch address, which is used as the next address if the branch is taken. (PC+1 is pushed onto the micro-stack during a successful call.) The (BRANCH_ADDRESS) is also selected during unsuccessful conditional returns and conditional dispatches.

The microsequencer also maintains a 15 word micro-stack that is used during micro-calls and returns. (The stack also maintains addresses for micro event handler returns.) The (BRANCH_TYPE) field can specify conditional and unconditional calls and returns, for both selected condition true and selected condition false.

The 16 branch types are:

```
brt             --conditional branch (branch if true)
brf             --conditional branch (branch if false)
br              --unconditional branch
cont            --continue (PC + 1)
callt           --conditional call (call if true)
callf           --conditional call (call if false)
call            --unconditional call
returnt         --conditional return (return if true)
returnf         --conditional return (return if false)
return          --unconditional return
dispt           --conditional dispatch (dispatch if true)
dispf           --conditional dispatch (dispatch if false)
disp            --unconditional dispatch
case *          --jump to the branch address plus the lsb 14 bits
                --of the FIU_DATA from the last cycle
case_call *     --same as the case, except PC + 1 is pushed onto
                --the stack
push            --push the branch address onto the micro_stack
```

(NOTE: The case and case_call branch to an address which is the sum of the branch address and the 14 lsb's from the last value on the FIU_DATA bus. This "last value on the FIU_DATA bus" is latched in a register on the microsequencer. This register is not readable neither writable. It is therefore necessary not to take micro-events before a micro-instruction that uses these branch types!)

The next micro-address for each combination of condition value and branch type is shown in the following table.

Since it is useful to remember a condition for several micro-cycles, the microsequencer provides a latch that can store the currently selected condition. The (LATCH) field of the microword specifies for each micro-instruction to either remember the previously latched condition or to latch the currently selected condition. (This feature is also useful for branching on late conditions, see below.) The contents of the latch will be saved and restored on context switches.

Because of the timing for the conditions some occur early in a micro-cycle and some occur late. Early conditions may always be selected for branches. Late conditions, if selected for a branch condition, must be followed by a hint. The hint informs the micro-sequencer that usually the branch will fail (rarely branch) or usually the branch will succeed (usually branch). If the hint is not correct the micro-sequencer will not execute the selected micro-instruction, but will take one micro-cycle to calculate the correct micro-address. (If both a bad hint and late micro event or a late macro event occur the hardware will take two micro-cycles to calculate the correct next micro-address!) The (BRANCH_TIMING) field for the early/late/hint conditions is interpreted as follows:

branch_type	condition value true	false
brt	branch_addr	PC + 1
brf	PC + 1	branch_addr
br	branch_addr	branch_addr
cont	PC + 1	PC + 1
callt	branch_addr	PC + 1
callf	PC + 1	branch_addr
call	branch_addr	branch_addr
returnt	micro_stack	branch
returnf	branch	micro_stack
return	micro_stack	micro_stack
dispt	decode_ram	branch
dispf	branch_addr	decode_ram
disp	decode_ram	decode_ram
case	branch_addr + FIU_data(50:63) *	
case_call	"	" *
push	PC + 1	PC + 1

(* -- For both cases true and false)

Table 2-1: Micro-Address Selection for Branch Types

branch_timing		
0	0	branch on the early condition
0	1	branch on the latched condition
1	0	branch on the late condition, hint is usually
1	1	branch on the late condition, hint is rarely

Table 2-2: Branch Timing

During a usually dispatch, memory might be started (depending on the decoding instruction). If the hint is bad, memory is aborted during the next cycle. When memory is aborted a memory read will finish and a write command will change into a read. Therefore after a usually dispatch that is bad the RDR and the MAR may no longer contain valid information.

2.2. Dispatch

During a successful dispatch, the microsequencer will do three things in hardware:

1. Increment the Macro_pc.
2. Start a memory read or write, if the decoding instruction requires a memory reference.
3. Select the next micro-address, based on the current decoding macro-instruction.

The decode rams have a 3 bit field for each macro instruction that selects one of eight possible memory references. (Because a dispatch can auto- matically start memory, microcode can not allow memory operations to extend across macro-instruction boundaries.) If the decoded memory start field, in the decode rams, is not a NOP it is one of the following memory operations:

CONTROL_READ_LL_DELTA
: Start a control or import read (if bits (3:6) of the decoding instruction are 0 the read is a import read, otherwise it's a control read). Bits (3:6) of the decoding instruction are used as the address to the resolve ram. The stack name portion of the address is read directly from the resolve ram. The offset portion of the address is the output of the resolve rams plus bits (7:15) from the decoding instruction (sign extended).

PROGRAM_READ_PC_PLUS_OFFSET
: Start a program read. The program address is equal to the current macro pc plus bits (5:15) from the decoding instruction (sign extended).

TYPE_READ_TOS_PLUS_FIELD_NUMBER
: Start a type read. The stack name portion of the address is read from bits (0:31) of the TOS_LATCH. The offset portion of the address is the sum of bits (37:56) in the TOS_LATCH and bits (8:15) in the decoding instruction (zero extended).

TYPE_READ_TOS_TYPE_LINK
: Start a type read. Both the name and offset are read from the TOS_LATCH (bits (0:31) and (37:56), respectively).

CONTROL_READ_VALUE_ITEM. NAME_AND_FIELD_NUMBER
: Start a control read. The name portion of the address is read from bits (64:95) of the TOS_LATCH. The

offset portion of the address is bits (8:15) of the decoding instruction (zero extended). (Useful on module field_reads, module field_exes, etc.)

CONTROL_READ_CONTROL_PRED
Start a control read. The stack name is the current_name register and the offset is the control_pred register.

CONTROL_WRITE_(INNER-PARAMS)
Start a control write. The stack name is the current_frame name and the offset is the current_frame offset minus bits (8:15) of the decoding instruction (zero extended).

2.3. Macro Events

If any macro events are pending during a dispatch, the dispatching instruction will complete entirely, but the dispatch will not occur. If the highest priority macro event pending is an early macro the next micro instruction will be the first micro-instruction of the corresponding macro event handler. If the highest priority macro event pending is a late macro event the next micro-instruction will be a NOP, followed by the first micro-instruction of the macro event handler. If the macro event is IBUFF_EMPTY the hardware will automatically start a program read at (macro pc + 1).

All of the macro events are testable as conditions and are maskable. Some of the macro events can be disabled during a particular micro-instruction (specified as disabled (D)). The macro events are cleared by some action that is executed during the handler. The macro events and some of the characteristics are:

Macro event definitions:

DISPATCH The dispatch macro event occurs every time a micro-instruction executes a successful dispatch. The sequencer microword allows this macro event to be disabled if specified as such during an instruction.

BREAK_CLASS The sequencer contains a 16 bit register which specifies 15 break classes and a "break any macro" bit. During each dispatch the break class of the "current instruction" is decoded. (There are decode rams on the output of the current instruction register. The rams output a 4 bit field for every instruction. This 4 bit field is either one of the break classes or it is the "no_break_class" class.) If the break class of the current instruction matches one of the 15 break classes (or break any is set) the break_class macro will occur.

MEMORY	Early /Late	priority	specif.	memory address	micro address
refresh memory	E	0			0100
SYSBUS					
sysbus_status	E	2			0110
sysbus_packet	E	3			0118
slice_timer	E	5			0128
gp_timers	E	6			0130
SEQUENCER					
CSA_underflow	L	8			0140
CSA_overflow	L	9			0148
resolve_ref	L	10			0150
TOS_optimization_err	L	11			0158
dispatch	L	13	D		0168
break_class	L	14	D		0170
IBUF_empty	L	15	D	PC+1	0178

(0 is the highest priority event)

Table 2-3: Macro Events

The conditions necessary for this macro event to occur are tested during each dispatch. If the macro event occurs, but the handler for a higher priority macro event is executed, this macro event is not latched (not remembered). The event will reoccur during the next dispatch.

CSA_UNDERFLOW Each instruction may requires some number of operands, from 0 to 7, to exist in the control stack accelerator, before the instruction can execute. If the dispatching instruction requires more operands in the CSA, than currently exist, this macro event occurs. The handler for this macro event will then read some number of entries (probably four), from the current top of the control stack not reflected in the CSA, and write them into the bottom of the CSA. (The CSA is located on both the type and value boards.) (The decode rams contains a 3 bit field for each macro instruction, which specifies the number of operands that the instruction requires in the CSA.)

The conditions necessary for this macro event to occur are tested during each dispatch. If the macro event occurs, but the handler for a higher priority macro event is executed, this macro event is not latched (not remembered). The event will reoccur during the next dispatch.

Once the handler has filled the CSA appropriately the macro event will not occur again. (NOTE: It is legitimate to change the number of entries in the CSA during the same micro-instruction that a dispatch is occuring.)

CSA_OVERFLOW

Each instruction may also requires some number of invalid locations, from 0 to 3, to exist in the control stack accelerator, before the instruction can execute. If the dispatching instruction requires more invalid locations in the CSA, than currently exist, this macro event occurs. The handler for this macro event will then write into memory some number of entries (probably two), from the bottom of the CSA, into the corresponding addresses in the control stack. (The decode ram contains a 2 bit field for each macro instruction, which specifies the number of holes that the instruction requires in the CSA.)

The conditions necessary for this macro event to occur are tested during each dispatch. If the macro event occurs, but the handler for a higher priority macro event is executed, this macro event is not latched (not remembered). The event will reoccur during the next dispatch.

Once the handler has emptied the CSA appropriately the macro event will not occur again. (NOTE: It is legitimate to change the number of entries in the CSA during the same micro-instruction that a dispatch is occuring.)

RESOLVE_REF

Any instruction that specifies a lex level, delta position in the control stack, requires that the current resolve ram registers must contain the offset of that specific lex level. If the dispatching instruction requires a resolve and the specified lex level offset is not in the current resolve ram registers this macro event occurs. The event handler for this macro will chase activation states in the control stack until the offset for the specified lex level is found.

The conditions necessary for this macro event to occur are tested during each dispatch. If the macro event occurs, but the handler for a higher priority macro event is executed, this macro event is not latched (not remembered). The event will reoccur during the next dispatch.

As soon as the handler validates the lex level, in the resolve ram, corresponding to the dispatching lex level, the macro event will not occur again.

TOS_OPTIMIZATION_ERROR

To optimize the execution speed of some instructions the sequencer hardware attempts to keep a copy of the current top of the control stack. During the dispatch cycle of some macro instructions that require a memory read, based on the address in the TOS, the microsequencer will start the memory read. If the micro- sequencer does not have a copy of the current TOS, and the dispatching instruction requires this optimization, this macro event will occur. The handler for this event will copy the current TOS from the CSA and write it into the TOS_LATCH on the microsequencer. Once the handler validates the TOS_LATCH the macro event will not reoccur. (NOTE: If the TOS_LATCH is validated during a dispatching instruction this macro-event will NOT occur.)

The conditions necessary for this macro event to occur are tested during each dispatch. If the macro event occurs, but the handler for a higher priority macro event is executed, this macro event is not latched (not remembered). The event will reoccur during the next dispatch.

IBUFF_EMPTY The microsequencer keeps a copy of the currently dispatching word from program segment memory. If the dispatching instruction is the eighth instruction in the buffer, and the instruction is not a call, exit, case, or any unconditional branch, this macro event will occur (The instructions which do not cause an ibuff_empty macro event, when they are the eighth in the buffer, are marked by the IBUFF_FILL bit out of the instruction decode. See the Instruction Decoder section.) During the same cycle the hardware will automatically start a memory read at address (PC + 1) in program memory. The handler for the event should be one instruction which conditionally loads the read data from memory into the IBUFF (instruction buffer), with the IBUFF_EMPTY macro event disabled. **(The condition is that no other macro event is occuring. This condition is necessary since the ibuff load will write over a macro-instruction that will not be dispatched if a macro event occurs.)**

2.4. Micro Events

Micro events which are early cause the execution of the current micro-instruction to be stopped. The next micro-instruction executed is a NOP, and the following micro-instruction is the first micro-instruction of the appropriate event handler (Each event maps to a unique address). Events which are late allow the current micro-

instruction to complete and inhibit the completion of the next micro-instruction. The instruction following the inhibited micro-instruction is the first micro-instruction of the event handler. (In either case the micro-PC that is pushed onto the stack is the PC of the micro-instruction that was inhibited or stopped.) If both early and late micro events happen during a micro cycle, the micro instruction is not completed. And the event (early or late) of the highest priority determines which handler is executed.

When an event is taken the handler address is the address corresponding to the highest priority event that is currently pending. The event bit is cleared for the event that is taken, and ALL other non-persistent events are cleared. (An event is also cleared if the event is tested.)

The events can be cleared for one of two possible reasons, either A) the event will occur again because the micro-instruction that caused the event to occur will execute again or B) A higher priority event detects an error that makes the other micro events insignificant (such as class error). (Since privacy_check is an early event, the privacy check will be performed again when the micro-code returns from the handler. The type board allows the microcode to disable this check for "one check cycle" in the handler.)

During a context switch only the persistent memory events must be saved. These events, page_crossing and page_fault, are part of the MAR and will be saved during the context switch. The other persistent events, the sysbus events, are independent of the currently running task and do not have to be saved.

Most of the events are testable as conditions and are maskable. The masks for a micro event can be one of two types; A)the mask bit is kept in a register and is readable and writeable by microcode (marked with a "X" in the table below), or B) the mask bit is specified (somehow, see the spec for the specific board in question) by the microcode during every micro-instruction (marked with a "M" in the table below). If an action occurs that causes a micro event which is masked off, the micro event will not occur until the mask is changed. (If the micro event is non-persistant it will clear if another unmasked event occurs first. The micro event will also clear if it is tested before the mask is changed.)

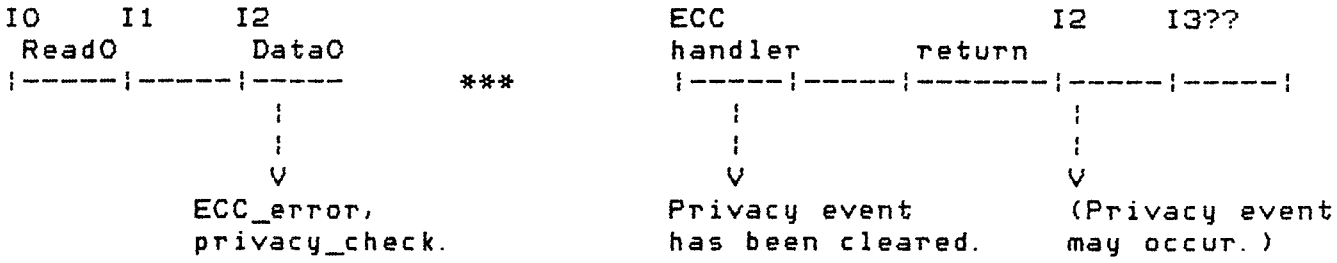
Some of the events are specifiable. These events are normally disabled and are only enabled when specifically selected by the microcode. (If a specifiable event is not selected it is not remembered, but it is testable.)

	cond	E/L	mask	specify	priority	persistant	micro addr
MEMORY MONITOR							
cache_miss	X	E	M		1	X	
ECC error		E	X		2		
page_crossing	X	E	X		9	X	
TYPE CHECK ERRORS							
class error	X	E		E	4		
binary_eq_privacy_check	X	E		E	5		
binary_op_privacy_check	X	E		E	6		
[tos]_op_privacy_check	X	E		E	7		
[tos-1]_op_privacy_check	X	E		E	8		
VALUE							
none							
SEQUENCER							
field_number_error	X	E		E	3		
FIU							
none							
CSA_CONTROL							
none							
OTHERS							
micro_interrupt (diag)	X	L	X		10	X	
SYSBUS							
new_packet	X	L	X		11	X	
new_status	X	L	X		12	X	

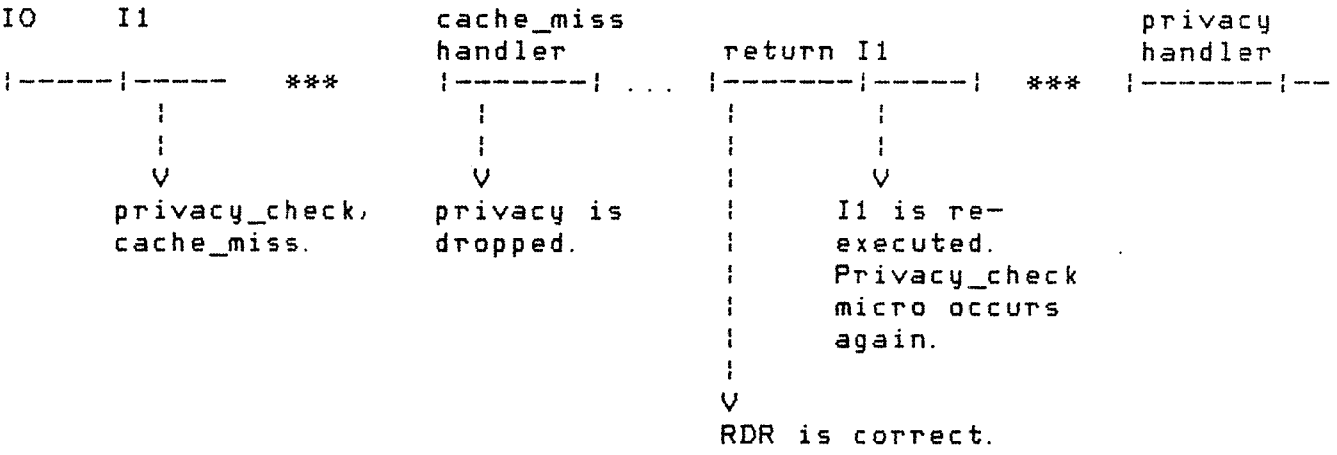
(Highest priority is one)

Table 2-4: Micro Events

Some examples of micro events and how they are handled.



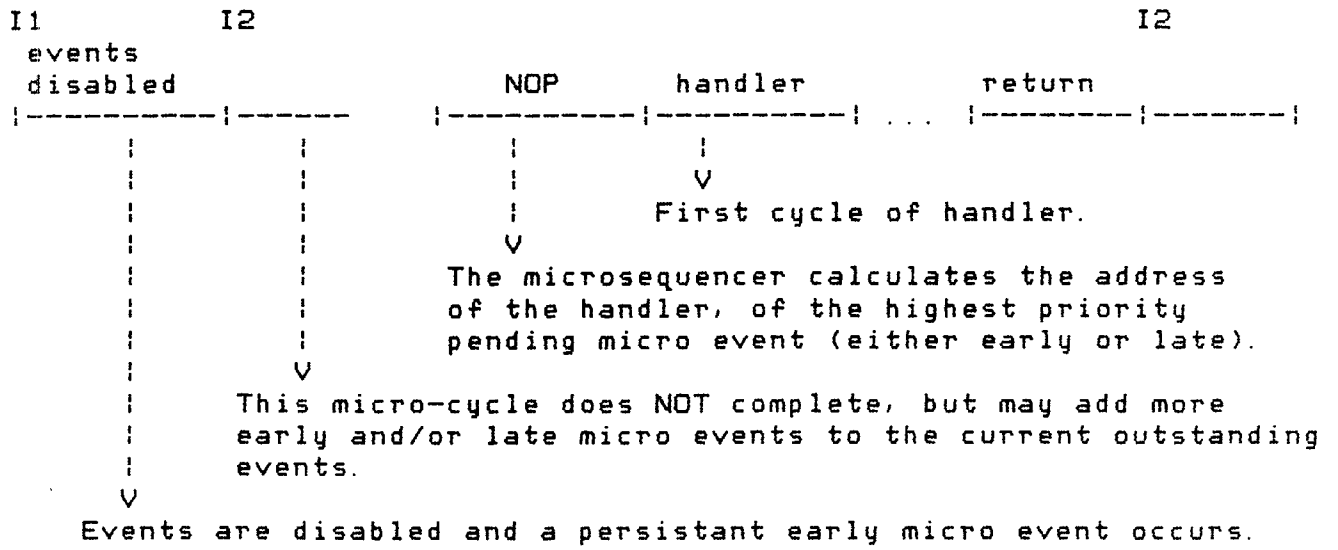
(*** A microcode invisible one cycle NOP occurs.)

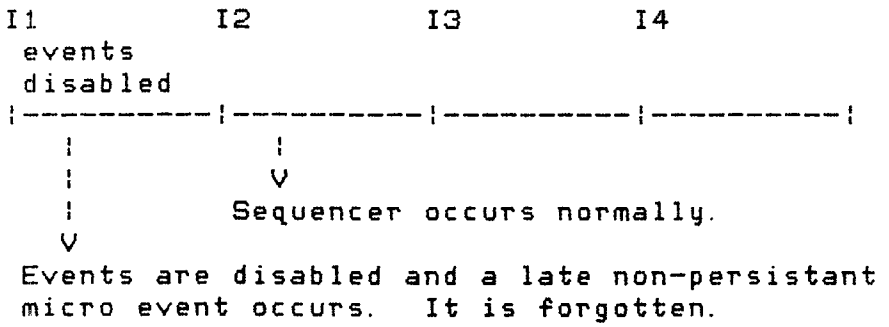


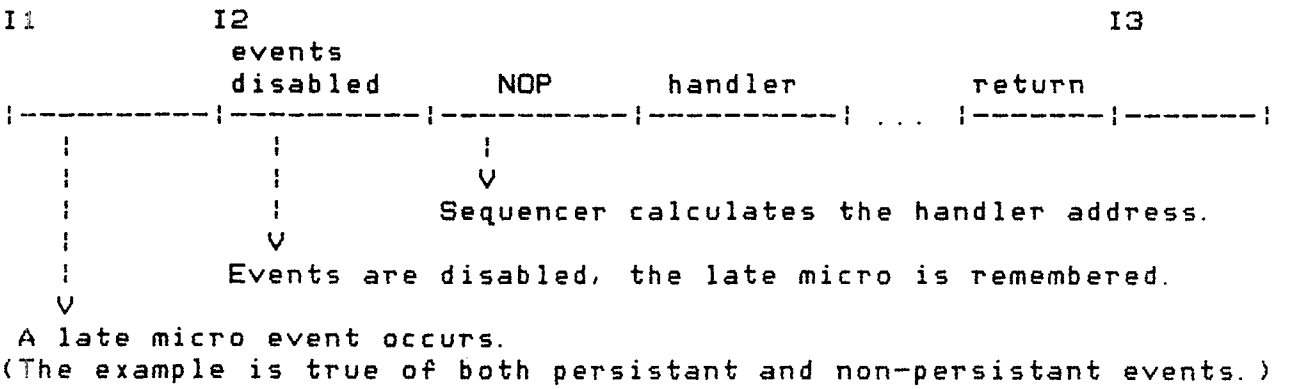
The microsequencer has a one bit microcode field which specifies during each instruction if micro events are enabled or disabled. If micro events are disabled during a micro-instruction, no early micro event will occur. If events are enabled during the following instruction, and the micro event is persistent, the early event will occur during that micro-cycle. (If this is undesirable, the micro event can be cleared, by testing it, before the micro events are enabled again.) If a non-persistent micro event occurs while interrupts are disabled, it is NOT remembered.

If a late micro event occurs and the following micro-instruction has events disabled, the micro event will be remembered and occur as soon as micro events are enabled again. If a late micro event occurs during a micro-cycle when events are disabled, it will be forgotten if it is non-persistant.

Some timing examples for disabled micro events.







do we need to put stack name here - or are all lex levels the same.

2.5. Resolve Circuit

The resolve circuit has sixteen 52 bit registers, corresponding to each of the 16 lex levels. 32 bits of the register are stack name bits (segment number and virtual processor ID) and 20 bits are an offset. There are also 16 validity bits, one corresponding to each lex level, which indicates if the contents of each register is valid. The resolve circuit also contains a current lex level register. (The architecture and some documents, including this one, refer to some of the lex level ram registers by specific names. The lex level zero register is imports, the lex level one register is the outer_frame. The register pointed to by the current lex level is the inner_frame. The register at the current lex minus one is the enclosing frame (unless the lex level is one, then the inner_frame and enclosing frame are the same).)

During the dispatch of a macro-instruction that requires the resolution of a lex level, delta, the resolve circuit will calculate the control stack address if the lex level is valid. If the lex level is invalid a macro event will be generated. The control stack address name is the name portion of the register specified by the lex level ((bits 3:6) of the decoding instruction). The offset is the sum of the offset portion of the specified register and bits (7:15) of the decoding instruction (sign extended).

Microcode has the capabilities to both read and write registers in the resolve circuit. The (LEX LEVEL ADDRESS) field of the microcode specifies how the resolve registers are addressed. The sources for the address are current lex register, incoming lex level (bits (124:127) of the sequencer bus minus 1), loop counter, zero, one. (The addresses can be used for either reads or writes.)

Microcode can also change the validity bits. In general validity bits are addressed at the same time the resolve registers are. During any cycle the addressed validity bit can be set, cleared, remain unchanged, or all the validity bits at a greater lex level can be cleared. (See the microword specification) The validity bits can also be cleared all at once, independent of the lex level address (see the random field of the microword specification).

The resolve circuit is also used to calculate the control or type addresses that the sequencer starts during some dispatches (see the dispatch section).

2.6. Tos_Latch

The (TOS_LATCH) on the microsequencer is used to latch 84 bits of the sequencer bus. If during the execution of a macro-instruction the new TOS (the control stack) is on the VAL and TYPE busses, the micro code should read the value onto the sequencer bus and latch it into the (TOS_LATCH). The (TOS_LATCH) also has an associated validity bit.

During each successful dispatch the bit is cleared. The bit is set when the latch is loaded. Some instructions will cause a macro event if the validity bit is not set. The contents of the TOS_LATCH is used during the calculation of some of the memory operations that the sequencer starts during the dispatch of some macro-instructions. (see the dispatch section)

2.7. Restartable State

For each executing macro-instruction the microsequencer remembers if the instruction is restartable and if restartable, the correct macro_pc to use. If a micro_event handler checks the restartable state before a context switch, the amount of state that needs to be saved can be minimized. (The restartable state is testable as conditions on the sequencer.) There are two bits of restartable state. The restartable bit (first bit) indicates if the macro-instruction is restartable or not restartable. If the instruction is restartable, the address bit (second bit) indicates if the instruction should be restarted at the current macro_pc or at the current macro_pc minus one. During the dispatch of each macro-instruction the restartable bit is set restartable. During a dispatch that causes a macro event the second bit is set to at macro pc. (During a bad hint both bits are restored to their previous value.) During any micro-instruction the microcode can set or reset each state bit independently.

(Example: If the cache_miss handler checks the state of these bits it can detect the case where a cache_miss is taken during a macro event. The bits would be set to restartable, at current macro_pc. By detecting this case, the saving of unnecessary micro- state is avoided.)

2.8. Micro Stack

The microsequencer maintains a 15 word deep LIFO stack of micro addresses. Micro addresses are automatically pushed and popped as a result of some of the branches (call, return), and during events. The microcode can also push FIU_DATA(48:63) onto the stack, clear the stack, read the top item, or pop an item off of the stack. (see the random field) (The micro stack hardware has no capabilities for overflow or underflow detection. The microcode must manage the stack usage to ensure that neither microcode action, or event actions will cause a underflow or overflow of the stack.)

Every time any item is pushed onto the stack the latched condition is also pushed onto the stack. This bit of the micro stack is selectable as a condition. This facility can be useful in the following circumstances:

1. If a micro event handler uses the condition latch, it

doesn't need to execute any microcode to save the previously latched condition. The condition is saved on the micro-stack. To restore the condition the return instruction should latch the condition "saved bit from the micro-stack". (Notice the save and restore take no extra micro- cycles.)

2. During a context switch, the latched condition can be saved on the micro-stack and restored from the micro-stack, just as in the above example.

3. A subprogram call that uses the condition latch, but shouldn't destroy its value can also restore the condition.

NOTE: Many subprograms will not want to restore the condition latch upon return. If a subprogram latches a condition (and doesn't restore the latch), it actually returns a boolean to the caller.

2.9. Field Number Checker

The microsequencer has two comparators for checking field numbers during the execution of the field ops. Each cause the same micro event; field number error. The variant field check compares bits (80:88) of the sequencer bus to bits (7:15) of the current instruction. The fixed field check compares bits (81:88) of the sequencer bus to bits (8:15) of the current instruction. In either case an unequal comparison generates the micro event.

2.10. Instruction Decoder

The instruction decode unit on the microsequencer outputs 23 bits of information about the instruction in the IBUFF (instruction buffer), pointed to by the macro pc. This information is divided into the following five fields:

1. MEMORY_REF A 3 bit field that indicates the dispatch of this instruction may need to start one of seven possible memory references. (The memory references that may be started are enumerated in dispatch section.)

2. CSA_VALID A 3 bit field that indicates the number of entries, from 0 to 7, that must be present in the CSA before the instruction can successfully execute. (If the CSA does not have at least that many entries valid a macro event will occur. See the macro_event section.)

3. CSA_FREE A 2 bit field that indicates the number of locations in the CSA, from 0 to 3, that must be free before the instruction can successfully execute. (If the CSA does not have at least that many free locations a macro event will occur. See the macro_event section.)

4. MICRO_ADDR A 14 bit field which is the starting micro-address for the microcode that executes the decoding macro-instruction.

5. IBUFF_FILL A 1 bit field which indicates if current instruction does not need a IBUFF_empty macro event to occur if the macro_pc mod 8, is 7. (For example: call, exit, unconditional jump, etc.) (A minor optimization used by the IBUFF_empty macro event hardware.)

HARDWARE NOTE: The decode rams are organized into two banks. The top bank of rams (1K x 23) address from the top ten bits of the decoding instruction. The bottom bank of rams (1K x 23) address from the bottom ten bits of the decoding instruction. If the top six bits of the instruction are zero the bottom bank's output is enabled otherwise the top bank is enabled.

Information is also decoded about the currently executing instruction. Another set of decode rams examines the currently executing instruction and decodes its break class. The output is 4 bits of break_class information. The instruction can belong to one (and only one) of 15 break_classes or it can belong to no break_class. (If the instruction belongs to a break_class and the break class is currently enabled, or the all_break is enabled, a break_class macro will occur during a dispatch.)

2.11. R1000 Processor conditions

The R1000 hardware has 128 testable conditions on the processor. The conditions come from all of the boards in the processor, except for the memory boards. During each cycle, the hardware selects on of the 128 conditions for testing. This condition can be latched on the microsequencer and/or used to resolve a conditional branch or conditional memory start. (If some conditions are selected they will also clear the corresponding micro event. This is true of only a few conditions. See the condition section of each spec for details.) The microsequencer contains a 7 bit microcode field which selects the condition during each cycle.

Each board in the R1000 produces some multiple of 8 conditions. The 128 conditions are divided between the hardware as shown in the following table.

The Combo conditions are special combinations between the value board and the type board. Combo condition XXX is equal to the logical NAND of condition 0000XXX and 0011XXX.

The conditions can be divided into three groups; L - late, ML - medium late, and E - early. The early conditons can be used as conditions for conditional branch types, and don't require a hint. The medium late and late conditions require hints if used with conditional branch

BOARD	CONDITION NUMBER
Value	0000XXX
	0001XXX
	0010XXX
Type	0011XXX
	0100XXX
	0101XXX
	0110XXX
	0111XXX
Microsequencer	1000XXX
	1001XXX
	1010XXX
Fiu (&Mem_M)	1011XXX
	1100XXX
Sysbus	1101XXX
	1110XXX
Combos	1111XXX

Table 2-5: R1000 Condition Partioning

types. Only the early or medium late conditions can be used as conditions for conditional memory references. And believe it or not, every condition can be latched in the microsequencer's condition latch.

3. Some Timing Examples

This section illustrates some timing examples for branch types and events. (Instructions that do not complete are equivalent to a null micro-instruction. Machine cycles that the hardware inserts, but no micro-instruction is executed are indicated as nulls.)

Each microcycle the micro sequencer decides the flow of control based on the following priorities (highest to lowest):

1. If the last instruction was a hint (and there were no macro or micro events), check for correctness. If the hint was wrong stop actions started by the wrong hint (such as dispatch memory starts), stop the current instruction from continuing, and calculate the new micro-address. (NOTE: Bad hints only stop memory operations if the branch type was a dispatch.)

2. If the last instruction was a bad hint and there were macro events, micro events, or both, stop actions started by the

hint (such as memory starts). Execute a null micro-instruction and calculate the correct address (forgetting the events). Then follow the appropriate set of rules that follow for the combination of events that occurred.

3. If there are any early micro events, stop the instruction from completing, and push the current micro-address onto the stack. During the next cycle execute a null and calculate the micro address of the micro event handler. The next micro-instruction is the first instruction of the handler.

4. If the instruction is a dispatch and there are both late micro events and early macro events, complete the instruction without starting the dispatch. During the next cycle execute a null, push the macro handler address onto the stack and calculate the micro address of the micro event handler. The next micro-instruction is the first instruction of the micro event handler.

5. If the instruction is a dispatch and there are both late micro events and only late macro events, complete the instruction without starting the dispatch. If the macro event needs a memory operation, start it (only on the IBUFF_empty macro event). During the next cycle execute a null micro-instruction and calculate the address of the macro event handler. During the following cycle execute a null, push the address of the macro event handler onto the stack, and calculate the address of the micro event handler. The following micro-instruction is the first micro-instruction of the micro event handler.

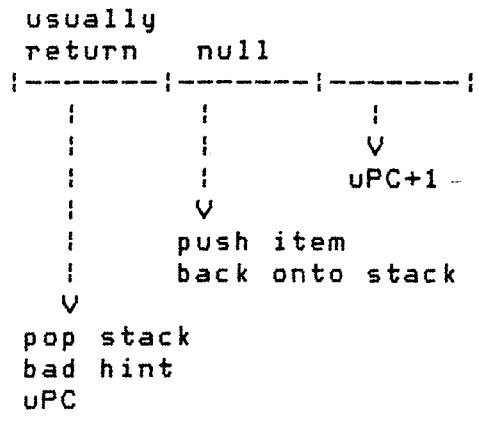
6. If there are only late micro events, complete the current micro-instruction (including a dispatch if part of the instruction). During the next cycle execute a null and push the "current micro-address" (the micro-address that would be executing if no event had occured) onto the stack. The next micro-instruction is the first instruction of the micro event handler.

7. If the instruction is a dispatch and there are any early macro events, the next micro-instruction will be the first micro-instruction of the macro event handler.

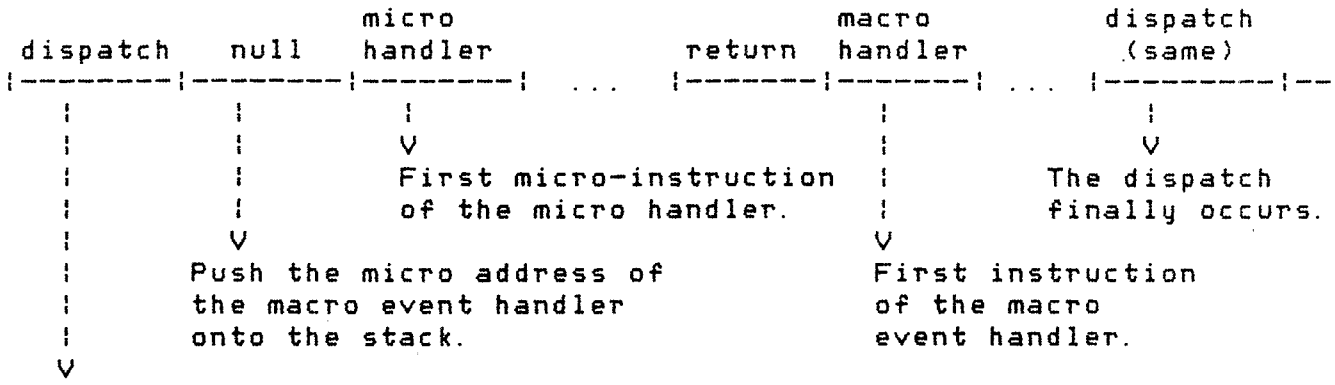
8. If the instruction is a dispatch and there are only late macro events, the next instruction is a null. The following micro-instruction will be the first instruction of the macro event handler.

9. If the instruction does not fall into one of the above categories it has the best chance of working properly, and probably does just what you expect.

Example 1: "A bad hint on a usually return"

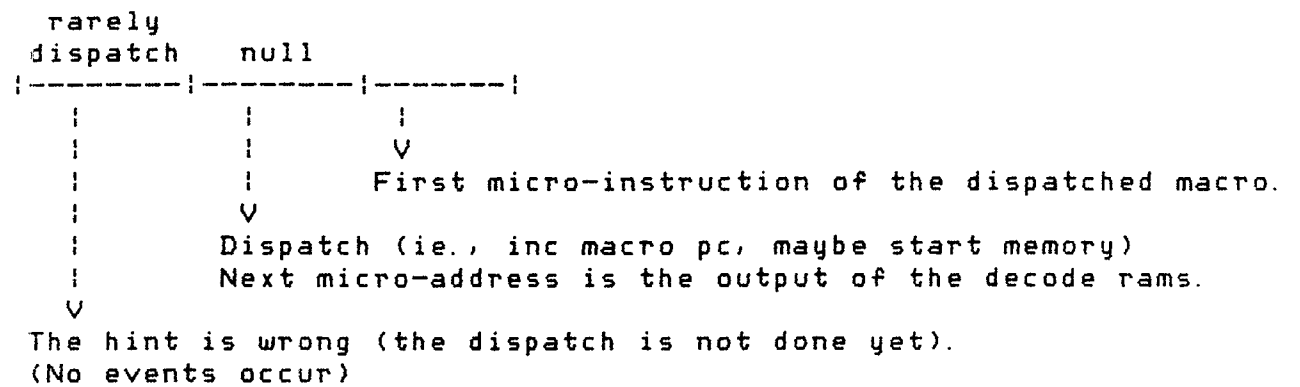


Example 2: "A late micro and a early macro"

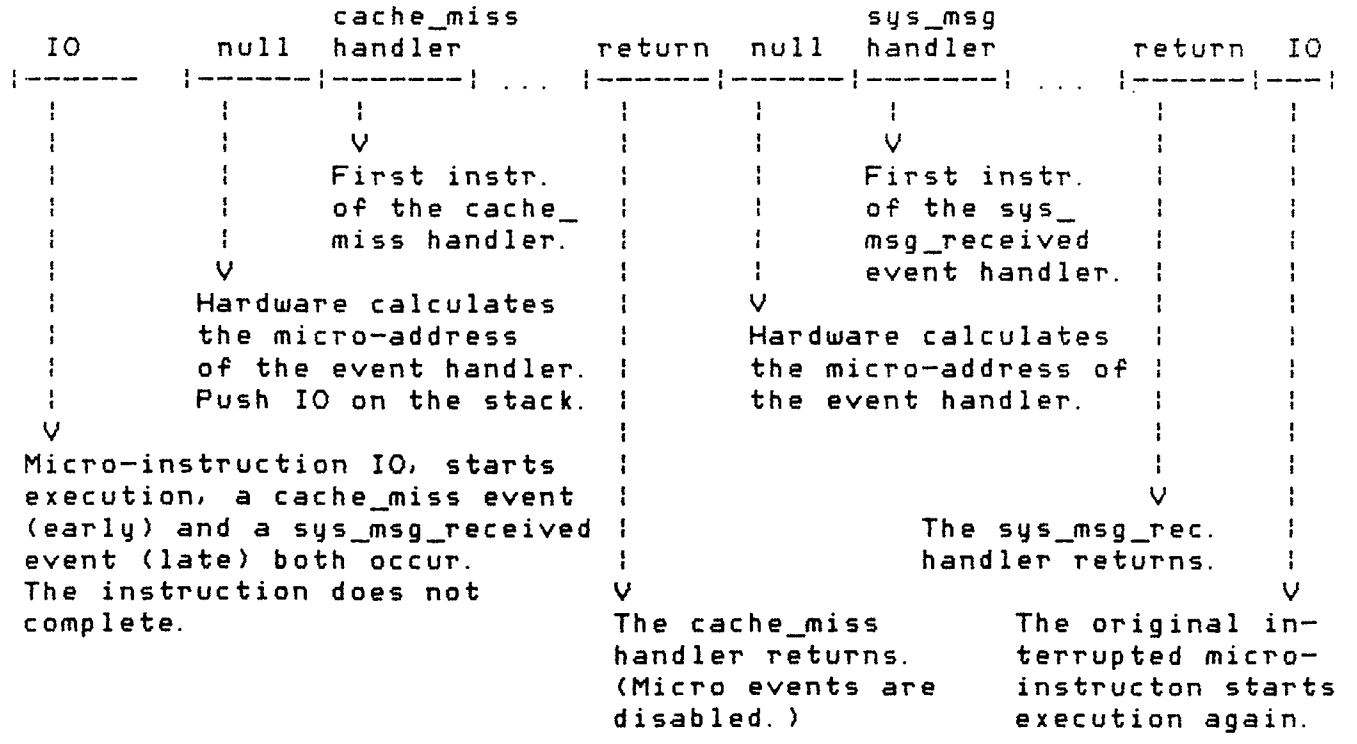


A late micro and an early macro event occur.
The dispatch doesn't start because of the late macro event.
(This means the macro pc doesn't change.)
If the macro event requires a memory operation it will start.

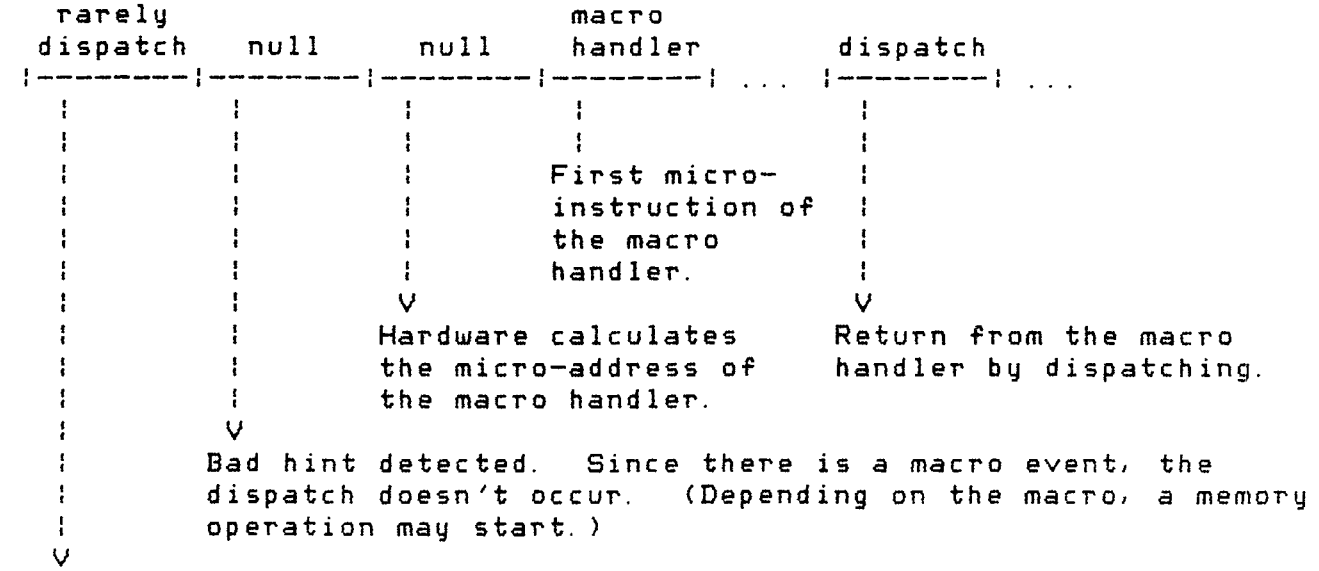
Example 3: "A incorrect hint of rarely dispatch (no events)"



Example 4: "Two persistant micro events occur at once"



Example 5:
"A incorrect hint of rarely dispatch with a late macro event"



The rarely dispatch is bad.
(The microcode branches.)

4. Microword Specifications

BRANCH ADDRESS (14 bits)

14 bits	The value of this field is the absolute branch address.

LATCH (1 bit)

The microsequencer contains a one bit latch whose input is the currently selected condition. During each micro-instruction a new value can be latched or the currently latched condition can be remembered.

0	Latch the selected condition.
1	Don't change the value of the condition latch.

BRANCH TYPE (4 bits)

brt	conditional branch (branch if true)
brf	conditional branch (branch if false)
br	unconditional branch
cont	continue (PC + 1)
callt	conditional call (call if true)
callf	conditional call (call if false)
call	unconditional call
returnt	conditional return (return if true)
returnf	conditional return (return if false)
return	unconditional return
dispt	conditional dispatch (dispatch if true)
dispf	conditional dispatch (dispatch if false)
disp	unconditional dispatch
case	jump to the branch address plus the 14 lsb bits of the FIU_DATA from the last cycle
case_call	same as the case, except PC + 1 is pushed onto the stack
push	push the branch address onto the stack

BRANCH TIMING (2 bits)

If a conditional branch type is selected, this field indicates which condition is used as test condition. (The translator default should be early condition.)

EARLY CONDITION -- Test the currently selected early condition.

LATCHED CONDITION -- Use the output of the latch.

HINT USUALLY -- Take the requested conditional branch. During the next cycle the hardware will test the outcome of the previous test condition and "undo" the branch type if incorrect.

HINT RARELY -- Do not take the requested conditional branch. During the next cycle the hardware will test the outcome of the previous test condition and take the branch type if incorrect.

PROCESSOR_CONDITIONS (7 bits)

XXXXXXX This field selects the currently tested processor condition.

(See the function description of conditions for a detailed description of how conditions work. See the condition section in microcode considerations for the sequencer generated conditions.)

LEX LEVEL VALIDITY CONTROL (2 bits)

During any microcycle the validity bits for the resolve circuitry can be set or cleared in the following manner. (The lex_level used is specified in the LEX LEVEL ADDRESS field.)

CLEAR_LL -- Clear the specified lex level.

SET_LL -- Set the specified lex level.

CLEAR > LL -- Clear all lex levels greater than the specified lex level.

NOP -- Don't change any of the validity bits.

LEX LEVEL ADDRESS (3 bits)

This field selects the address that is used to address the resolve ram.

CURRENT_LEX -- Use the current lex level.

INCOMING_LEX -- Use the value on bits (124:127) of the sequencer bus minus one.

LOOP_COUNTER -- Use the 4 lsb of the loop counter.

0 -- Address the import frame.

1 -- Address the outer frame.

MICRO EVENT CONTROL (1 bit)

When a micro-instruction disables micro events, no micro events can occur between the previous instruction and the currently executing instruction. (This disabling includes the page_crossing event.) (See the micro event section.)

DISABLE_ALL_MICROS

NOP

MACRO EVENT CONTROL (2 bit)

When a micro-instruction disables macro events, no macro event can occur between th previous instruction and the currently executing instruction.

DISABLE_(BREAK_&_DISPATCH)

???????

???????

NOP -- Allow all macro events that aren't masked.

INTERNAL SEQUENCER READS (3 bits)

This field determines what data is driven onto the sequencer bus. (The bit format, and the number of bits per field are indicated in the right margin, for some of the internal reads.)

VAL_TYPE BUS -- Read the val and type busses. (This should be the assembler default.)

RESOLVE_OUTPUT

resolve_frame.number	0	23	24
resolve_frame.proc	24	31	8
***	32	36	5
resolve_offset	37	56	20
***	57	63	7
***	64	71	8
macro_pc.segment	72	95	24
***	96	108	13
macro_pc.offset	109	120	12
macro_pc.index	121	123	3
current_lex_level	124	127	4

CONTROL_PRED				
	current_name.number	0	23	24
	current_name.proc	24	31	8
	***	32	36	5
	control_pred	37	56	20
	***	57	63	7
	***	64	71	8
	macro_pc.segment	72	95	24
	***	96	108	13
	macro_pc.offset	109	120	12
	macro_pc.index	121	123	3
	current_lex_level	124	127	4
CONTROL_TOP				
	current_name.number	0	23	24
	current_name.proc	24	31	8
	***	32	36	5
	control_top	37	56	20
	***	57	63	7
	***	64	71	8
	macro_pc.segment	72	95	24
	***	96	108	13
	macro_pc.offset	109	120	12
	macro_pc.index	121	123	3
	current_lex_level	124	127	4
NEW_TOP				
	current_name.number	0	23	24
	current_name.proc	24	31	8
	***	32	36	5
	new_top	37	56	20
	***	57	63	7
	***	64	71	8
	macro_pc.segment	72	95	24
	***	96	108	13
	macro_pc.offset	109	120	12
	macro_pc.index	121	123	3
	current_lex_level	124	127	4
CURRENT_INSTRUCTION				
	macro_mask			16
	micro_mask			4
	break_mask			16
	number_in_CSA			4
	number_in_micro_stack			4
	current_instruction	112	127	16

```
DECODING_INSTRUCTION
                macro_mask                                  16
                micro_mask                                   4
                break_mask                                  16
                number_in_CSA                                4
                number_in_micro_stack                        4
                decoding_instruction    112      127        16

TOP_OF_MICRO_STACK
                macro_mask                                  16
                micro_mask                                   4
                break_mask                                  16
                number_in_CSA                                4
                number_in_micro_stack                        4
                top_of_micro_stack      112      127        16
```

RANDOM FIELD (7 bits)

The random field controls the following specified sequencer operations. The sequencer hardware will allow 128 combinations of these operations to be programmed into a prom.

```
ADDR := control_pred
ADDR := resolve output
ADDR := control_top
ADDR := macro_pc
ADDR := return_pc


macro_pc := value on sequencer bus
write return_pc
macro_pc := return_pc
Conditional (load IBUFF & macro_PC := ADDR )

inc macro_pc
dec macro_pc


loop_counter := fiu_bus(60:63)
clear loop_counter
inc loop_counter
dec loop_counter

write break_mask
write micro_mask
write macro_mask

control_top := resolve_output
control_top := sequencer_bus (37:56)
control_pred := resolve_output
control_pred := fiu_bus (37:56)
```

```
                write current_name
                write current_instruction

                push micro_stack  (with FIU(48:63))
                pop micro_stack
                clear micro_stack

                fiu_data := top_of_micro_stack
                fiu_data := current_instruction


                restartable @PC
                restartable @(PC-1)
                not_restartable


                write resolve circuit, name half
                write resolve circuit, offset half
                validate_TOS_optimizer
                load_IBUFF
                invalidate_all_lex_levels
                take micro event
        interrupt diagnostic processor

                check fixed field number
                check variant field number
```

TOTAL NUMBER OF MICROBITS = 47.

5. Microcode Considerations

The following subsections detail microcode constraints and restrictions that are necessary for proper hardware operation.

5.1. Conditions

The following conditions are selectable on the microsequencer:

1000000	macro_restartable	(E)
1000001	restartable_@(PC-1)	(E)
1000010	valid_lex(loop_counter)	(E)
1000011	loop_counter_zero	(E)
1000100	TOS_LATCH_valid	(L)
1000101	saved_latched_cond	(E)
1000110	previously_latched_cond	(E)
1000111	#_entries_in_stack_zero	(E)
1001000	ME_CSA_underflow	(L)
1001001	ME_CSA_overflow	(L)
1001010	ME_resolve_ref	(L)
1001011	ME_TOS_opt_error	(L)
1001100	ME_dispatch	(L)
1001101	ME_break_class	(L)
1001110	ME_ibuff_empty	(L)
1001111	uE_field_number_error	(ML)
1010000	spare	
1010001	spare	
1010010	spare	
1010011	spare	
1010100	spare	
1010101	spare	
1010110	spare	
1010111	spare	

macro_restartable
: This condition is true if the current macro instruction can be restarted (See next condition for starting PC).

restartable_@(PC-1)
: If "macro_restartable" is true, this condition is true if the task should be restarted after the macro pc has been decremented. (If false the task should be restarted without changing the macro pc.)

valid_lex(loop_counter)
: This condition is true if the lex level specified by the least significant four bits of the loop counter, is valid.

loop_counter_zero
: This condition is true if the value of the loop counter is zero.

TOS_LATCH_valid This condition is true if the tos_latch is valid, or is currently being validated.

saved_latched_cond
This condition is true if the "saved latched bit" on the micros stack is currently one.

previously_latched_cond
This condition is true if the previously latched condition is true.

#_entries_in_stack_zero
This condition is true if the micro stack is empty.

ME_CSA_underflow
This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a CSA_underflow macro event.

ME_CSA_overflow This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a CSA_overflow macro event.

ME_resolve_ref This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a resolve_ref macro event.

ME_TOS_opt_error
This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a TOS_opt_error macro event.

ME_dispatch This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a dispatch macro event.

ME_break_class This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a break_class macro event.

ME_ibuff_empty This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a ibuff_empty macro event.

uE_field_number_error
This condition is true if the dispatching of the decoding instruction, with the macro mask enabled, would cause a field_number_error macro event.

5.2. Context Switch

5.3. Microcode Restrictions

There are certain combinations of microcode fields that are illegal or at a minimum produce unexpected side effects.

5.3.1. Branches

Because a dispatch may start a memory reference (depending on the decoding macro-instruction), the following combinations of memory requests and branches are illegal:

1. An unconditional dispatch and any memory reference.
2. A conditional dispatch on any early condition and any memory reference.
3. A conditional dispatch with a "usually" hint, and any memory reference.
4. A conditional dispatch, with a "rarely" hint, and a unconditional memory reference.

The above restrictions should allow only a single combination of conditional memory reference and conditional dispatch as legal microcode:

1. A conditional dispatch, with a "rarely" hint, and a conditional memory reference.

NOTE: A usually dispatch may start memory. If the hint is wrong the memory cycle will be aborted, and the contents of the MAR and the RDR are destroyed.

The case and case_call branch types use the previous value on the FIU_DATA bus as part of the branch address. The microcoder must disable all events (macro and micro) during the micro-instruction that uses these branch types to ensure that the branch address is correct.

During returns from event handlers, events (both micro and macro) should be disabled to allow the stack to remain at a reasonable size.

Translator issue

5.3.2. Sequencer Address Enables

If the microsequencer is driving the address bus with control_pred, resolve_output, or control_top the internal sequencer read micro field should be reading the same register if any are read. (ie., the combination SEQUENCER_BUS := CONTROL_PRED and ADDR := CONTROL_TOP is illegal).

5.3.3. CYA

The are a few other combinations, that are not listed here, that produce undesired side effects. The reader is warned not to use them.

Table of Contents

List of Tables

Functional Specification of the Memory Monitor

DRAFT 3

Rational Machines proprietary document.

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 microorders, 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:

Refrsh Intvl (16) *	Refrsh Windw (16) *	State (9)	**	FIU length (6)	spare (12)	Space (3)
0 15	16 31	32 39	40 43	48	49 60	61 63

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

Segment Number (24)	VPid (3)	Page Number (19)	Word (6)	Bit (7)
0 23	24 31	32 50	51 56	57 63

* - 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<33>)
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 (~ 30 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.

2.3.4. 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 without 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).

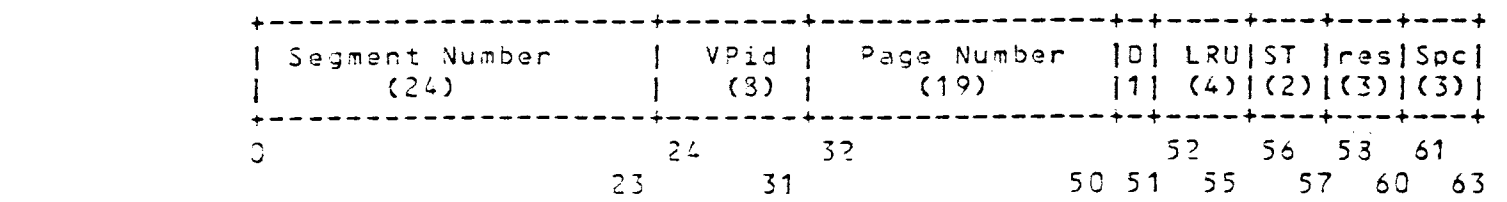
When a tag value is written, all fields are written, including LRU. Therefore, care must be taken to make sure all sets on a particular line have unique LRU values between zero and the MRU set number. Note also that, following power up, all tags on each line of the cache must be properly initialized 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..13> and segment name <14..18>

line address <5..3> := 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<13>, 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).

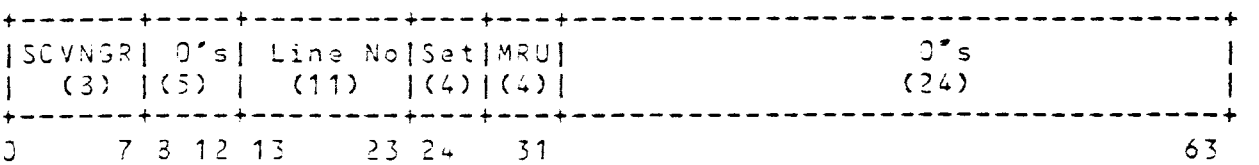
Invalid (11) This page is not assigned any logical page; no logical or name query will match this entry.

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 written 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 supporesses 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_ROR
* 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)	E, LOAD_MAR C2	no	load MAR	L
(MAR state bit)	ML, mem start C2	component	by testing	L
CONTROL_ADDRESS_OUTOF_RANGE *				
(test condition)	E, 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 LAST/ PHYS	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.

27. When the TYPE board is selected to drive the space part of the address bus (which happens whenever either the TYPE or VAL boards are selected to drive the address bus), the least significant 3 bits of B_Address data are driven onto the space part of the address bus, unless the current MAR_CONTROL microorder is LOAD_MAR_xxx, in which cases the literal xxx is driven. When RESTORE_MAR is the MAR_CONTROL microorder, whatever is selected as TYP B_Addr determines the data loaded into the MAR space portion (this is normally the same source as data driven via the TYP Bus to the TI Bus to be loaded into the RANDOM bits). When INCREMENT_MAR is issued as MAR_CONTROL microorder, but the FIU is not selected as address bus source, the memory monitor execute a LOAD_MAR on whatever is driven on the address and space portions of the address bus. Using this feature, MAR flags may be cleared and the address portion of the MAR modified in a single cycle:

```
Read_MAR,   -- handle page crossing microevent
    Typ { ALU_BUS := Pass_B( TYP_BUS ) },
    Val { ADDRESS_BUS := 4096 plus VAL_BUS },
    Address_Bus_Src := TYPE_BOARD, Increment_MAR,
    Select_Condition FIU (Page_Crossing), return ;
```

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 supporesses 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

Specification of the Field Isolation Unit

DRAFT 2

National Machines proprietary document.

1. Summary

This document is a functional and physical specification of the R1000 Field Isolation Unit (FIU). It is assumed that the reader is familiar with the R1000 architecture and has access to documentation on other parts of the hardware.

Section 2 of this document describes the functionality of the FIU and includes a description of each major block of the FIU block diagram. Section 3 describes the microword, and provides detailed information about the hardware that must be considered when writing microcode for the FIU. Section 4 includes some examples of how to use the FIU. Section 5 discusses diagnostic capabilities provided by the hardware and section 6 is a physical specification of the FIU board

2. Functional Overview

The basic operations of the FIU are inserting a field of 0 to 64 bits into any position of a 128 bit word, and extracting a field of 0 to 64 bits from a 128 bit word. The extracted data is right aligned on the value half of the FIU output and can be sign extended or zero filled. The fields being inserted or extracted can be defined by their offset and length. For insert, offset refers to the bit position where the most significant bit of the source data is to be inserted into the result. For extract, offset is the bit position of the most significant bit of the field in the source data. In both cases length is the length of the field being manipulated. The bit numbering convention is MSB=0 and LSB=127. Inserting a field of zero length does not modify the destination word, and extracting a field of zero length returns all zeroes. The FIU will also be used for Block Copy and Append operations, which can be accomplished with a series of insert and extract operations.

2.1. Rotator

The rotator is used for positioning bit fields for insert or extract operations. When extracting a field it also outputs the most significant bit of the field being extracted for sign extending. The 128 bit input to the rotator can be rotated by any amount and the appropriate 64 bit slice of the result is output. The bits to be output and the rotate amount are determined by whether you are doing an insert or extract, and by the field offset and length. If the operation is extract, the rotate amount is -(offset+length) right [(offset+length) left], and the sign bit is determined by offset. If the operation is insert, the the rotate amount is (offset+length). The hardware selecting the appropriate 64 bit slice to be output assumes, on insert operations, that the type and value halves of the rotator input are the same. If the insertion does not cross the type/value boundary or a word boundary then the data need only be

input to the side of the rotator corresponding to the side into which the data will be inserted.

2.2. Merge Data Register

The Merge Data Register (MDR) is used for storing the rotated data to be inserted into another 128 bit word, or for holding an intermediate result to be stored in the register file on either the VALUE or TYPE board. This register can be loaded and read by the diagnostic processor for testing the entire FIU data path.

2.3. Merge Vmux

The Merge Vmux selects one of four sources of data to be merged with the rotator output on the Value half of the merger. The four sources to this mux are the MDR, the rotator sign bit output, the VI bus, and the FIU bus.

2.4. Merger

The merger is a 128 bit 2 to 1 multiplexor which merges data from the output of the rotator (ROTDATA) with data coming from the TI bus on the type half, and the merge vmux on the value half. A merge mask is generated to control the select lines of the merger using the offset, length, and operation parameters. These parameters are used to calculate a start bit and end bit which mark the beginning and end of the field where the ROTDATA will be selected on the merger outputs.

operation	start bit	end bit
extract	128 -length	127
insert first word	offset	127
insert last word	0	offset+length-1
insert	offset	offset+length-1

In the examples of section 4, the merger inputs are referred to as the rotated inputs, which is the data from the ROTDATA bus, and the unrotated inputs, which is the data from the TI bus and Merge_Vmux.

2.5. Type Assembly Register and Value Assembly Register

The TAR and VAR are used for storing intermediate results, and are also used when the timing does not allow the results to be written back out into the register file. The only data path restricted by timing in this way is data being driven from the register file, through the FIU data path, and out to the register file on the FIU bus.

2.6. Bus interfaces

The FIU interfaces to the TYPE_DATA, VALUE_DATA, and FIU_DATA busses. The sources of these busses are selected by two fields in the microcode : VALUE_AND_TYPE_BUS_EN and FIU_BUS_EN, which are in the FIU part of the microword. These busses include byte parity checking and generation, and a machine check is generated if a parity error is detected on a bus from which the FIU will be receiving data.

2.7. Conditions

The FIU generates two conditions, cross_word_field, and offset_bit0. The cross_word_field condition results when a field insertion or extraction crosses a word boundary. The offset_bit0 condition is tested mainly for determining whether the BADBITID generated by the ECC hardware is in the upper or lower half of the 128 bit word.

3. Microword Specification

3.1. Microword Format

OFFSET (7 bits) - specify a literal for the offset parameter. The microcode should specify offset~ in this field.

xxxxxxx = offset~

OFFSET_REG_CONTROL (2 bits) specify the load control for the offset register

00	Load offset register with Address Bus (7 LSB's)
01	Load offset register with OFFSET literal
1x	No Load

OFFSET_SOURCE (1 bit) - specifies the source of the offset parameter

0	offset = offset register
1	offset = micro literal

LENGTH_AND_FILL_MODE (7 bits) - specify a literal for the length and fill mode parameters. The most significant bit of this field is the fill mode literal and the least significant 6 bits are a literal that specify length-1. A length of 64 is differentiated from a length of 0 by defining the fill mode bit for 64 to be 0 (which will indicate sign extend) and the fill mode bit for 0 to be 1 (which will indicate zero fill).

?llllll = fillmode(!length-1

? = 0 => sign extend
? = 1 => zero fill

LENGTH_AND_FILL_REG_CONTROL (2 bits) Specify the load control for the length and fill mode register.

	length part	fill mode part
00	Load VI (25:31) ?	Load TI (36) 4 0
01	Load literal	Load literal
10	Load TI (37:42) 41:46	Load TI (36) 4 0
11	no load	no load

FILL_MODE_SOURCE (1 bit) specifies the source of the fill mode parameter

0	fill mode = fill mode register
1	fill mode = micro literal

LENGTH_SOURCE (1 bit) specifies the source of the length parameter

0	length = length register
1	length = micro literal

VI_AND_TI_BUS_SOURCES (4 bits) specify the source of the TI and VI busses

	TI bus source	VI bus source
0000	TAR	VAR
0001	TAR	VALUE_BUS
0010	TAR	FIU_BUS
0011	FIU	VAR
0100	FIU	VALUE_BUS
0101	FIU	FIU
0110	TYPE_BUS	VAR
0111	TYPE_BUS	VALUE_BUS
1000	TYPE_BUS	FIU
1001	MAR	MAR
1010	TAR	FRAME_ADDRESS
1011	TYPE_BUS	FRAME_ADDRESS
1100	FIU	FRAME_ADDRESS
1101	spare	
1110	spare	
1111	spare	

LOAD_MDR (1 bit) specifies whether or not to load the Merge Data Register

0	don't
1	do

OPERATION_SELECT (2 bits) specify the FIU operation, mod 128 of the results for start bit, end bit, and rotate amount are used by the hardware.

op	merge mask start bit	merge mask end bit	rotate amount
00 extract	128-length	127	-(offset+length)
01 insert last	0	offset+length-1	offset+length
10 insert first	offset	127	offset+length
11 insert	offset	offset+length-1	offset+length

MERGE_INPUT (1 bit) specifies the source of data to the merger

0	rotator output
1	merge data register

MERGE_VMUX_SELECT (2 bits) specify the output of the merge_vmux

00	merge data register
01	fill value
10	VI
11	FIU BUS

LOAD_TAR (1 bit) specifies whether or not to load the TAR with TD

0	don't
1	do

LOAD_VAR (1 bit) specifies whether or not to load the VAR with VD

0	don't
1	do

FIU_BUS_SOURCE (2 bits) specify the source of data onto the FIU BUS

00	FIU board
01	VALUE board
10	TYPE board
11	MICROSEQUENCER board

VALUE_AND_TYPE_BUS_SOURCES (4 bits) specify the source of the value and type busses

	TYPE_BUS_SOURCE	VALUE_BUS_SOURCE
0000	TYPE board	VALUE board
0001	TYPE board	FIU board
0010	FIU board	VALUE board
0011	FIU board	FIU board
0100	~~MEMORY board READ DATA~~	MEMORY ~~board~~ READ_DATA ~~(READ_RDR)~~
0101	SYSBUS board	SYSBUS board
0110	MICROSEQUENCER board	MICROSEQUENCER board
0111	TYPE board	MEMORY ~~board~~ (READ_TVR)
1000	FIU board	MEMORY ~~board~~ (READ_TVR)
1001	spare	
1010	spare	
1011	spare	
1100	spare	
1101	spare	
1110	spare	
1111	all boards disabled from driving the TYPE, VALUE, and FIU busses	

3.2. Microcode Considerations

3.2.1. Offset, Length, and Fill Mode Parameters

The seven least significant bits of the ADDRESS bus are latched in the FIU board when loading or restoring the MAR, or when the SYSBUS board drives the BADBITID onto these lines as a result of an ERCC event. This register can also be loaded with the literal specified in the OFFSET field. This latched value or a literal can be selected as a source for the offset parameter. If the latched value is selected, the data must have been loaded into the register at least one cycle previous to starting any FIU operations; however, a literal can be specified and an operation using that literal can be performed in the same cycle. The source of data to the length parameter register is selected by microcode to be from the VI bus bits (25:31), from the literal specified in the LENGTH field, or from the TI bus bits (37:42). When restoring the MAR the microcode must select the TI bus source to this register. This latched value or a literal can be specified as the source for the length parameter. If the latched value is selected the data must have been loaded into the register at least one cycle previous to starting any FIU operations using this value; however, a literal can be specified and an operation started using that literal can be performed in the same cycle. The source of data to the fill mode parameter register is selected by microcode to be from the TI bus bit (36) or from the literal specified in the FILL_MODE field. If the latched value is selected, the data must have been loaded into the register at least one cycle prior to starting any FIU operations using this value; however a literal can be specified and an operation started using that literal can be performed in the same cycle. When reading the MAR for saving state, the offset register is returned on the seven LSB's of the VALUE bus, the fill_mode register is returned on bit 36 of the TYPE bus, and the length register is returned on bits 37:42 of the TYPE bus.

3.2.2. Condition timing

If the data being loaded into the offset and length registers in cycle N result in the value (offset+length) being greater than 128, then a cross_word_field condition will be generated which is an early condition and will be valid during cycle N+1. This condition will change only when new data is loaded into either of these registers. This condition only indicates that the values in the offset and length registers will result in a cross_word_field, unlatched literals will not affect this condition.

3.2.3. Saved state

When an early event stops occurs the clocks to some registers are stopped so that they can be saved and restored by the event handler. The Offset, Length, and Fill_Mode registers are the only registers in the FIU that are clock in this manner.

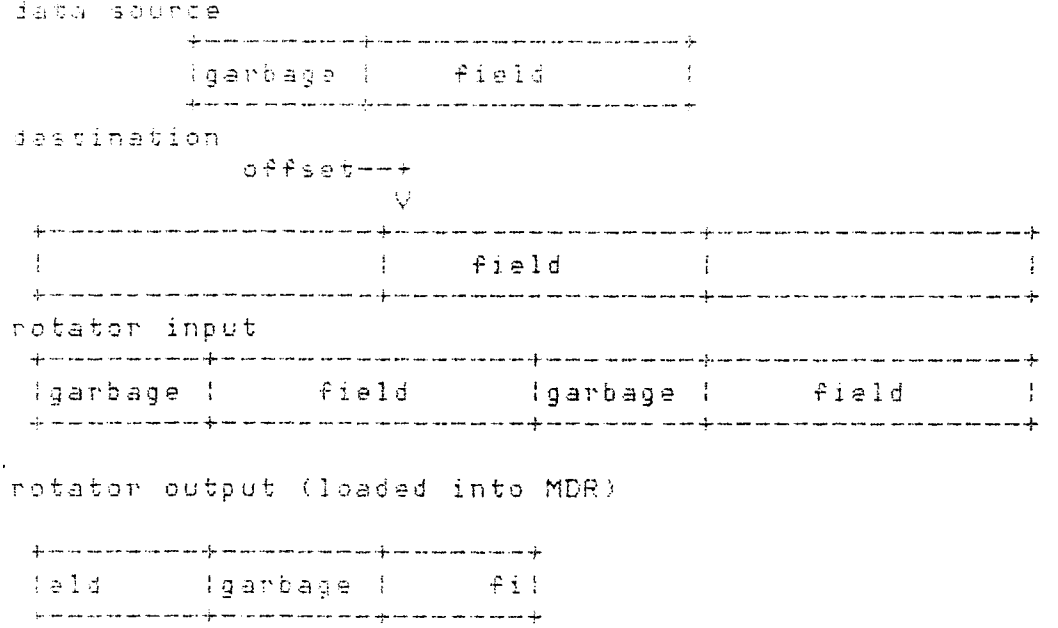
3.2.4. Miscellaneous microcode restrictions

When using the FIU bus as a source to the Merge_Vmux, the FIU bus must also be selected as a source to the TI bus so that parity can be checked.

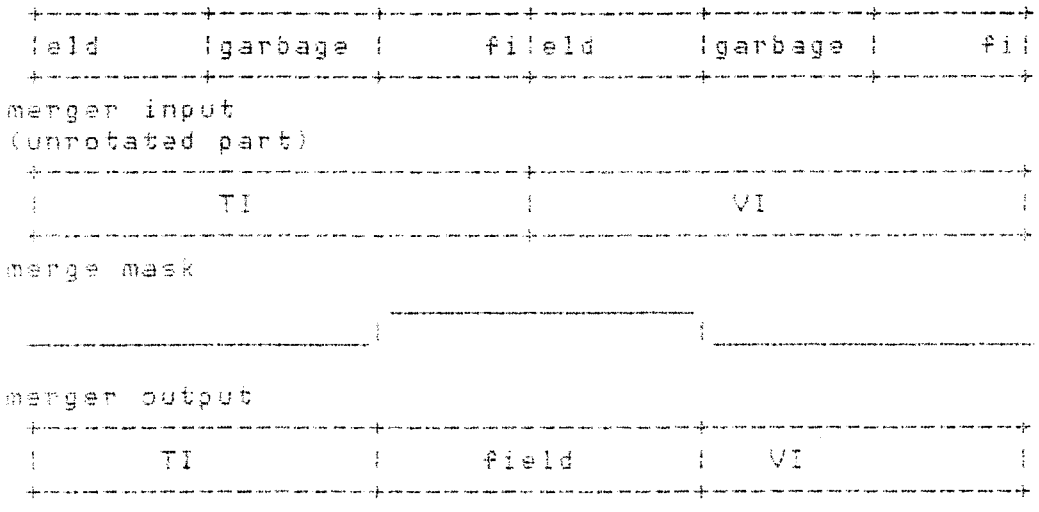
4. FIU Examples

4.1. Insert

CYCLE 1 - The offset and length parameters are determined from a literal specified in this cycle or from a value latched in some previous cycle (it is possible to specify a literal for one and a previously latched value for the other). The 1 to 64 bit field is sent from the register file on the TYPE or VALUE board, over the FIU_BUS and driven onto both the TI and VI busses. These bits are rotated right by (offset+length), and the result is stored in the MDR

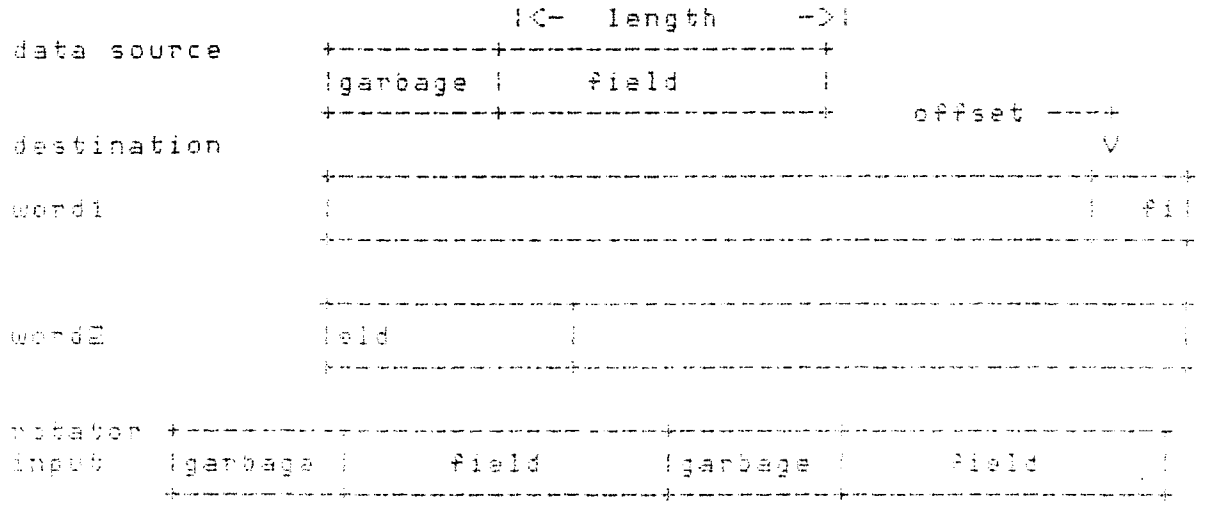


CYCLE 2 The output of the MDR is input to the rotator part of the merger on both the value and type halves. The word into which the field is inserted, is being driven onto the TI and VI busses. The TI bus drives the unrotated of the merger input on the TYPE half, and the VI bus is selected as the MERGE_VMUX output to drive the unrotated part of the merger input on the VALUE half. The merge mask start bit is (offset) and the endbit is (offset+length-1). The output of the merger is loaded into the TAR and VAR.

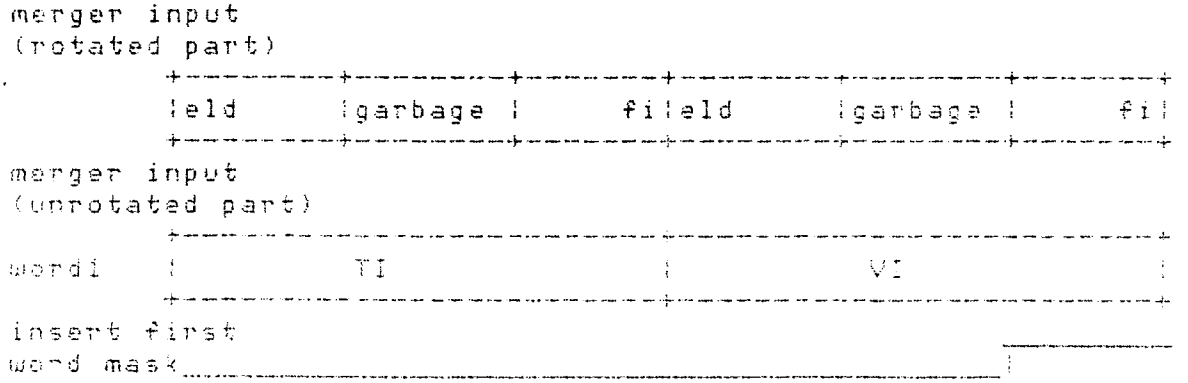


4.2. Cross Word Insert

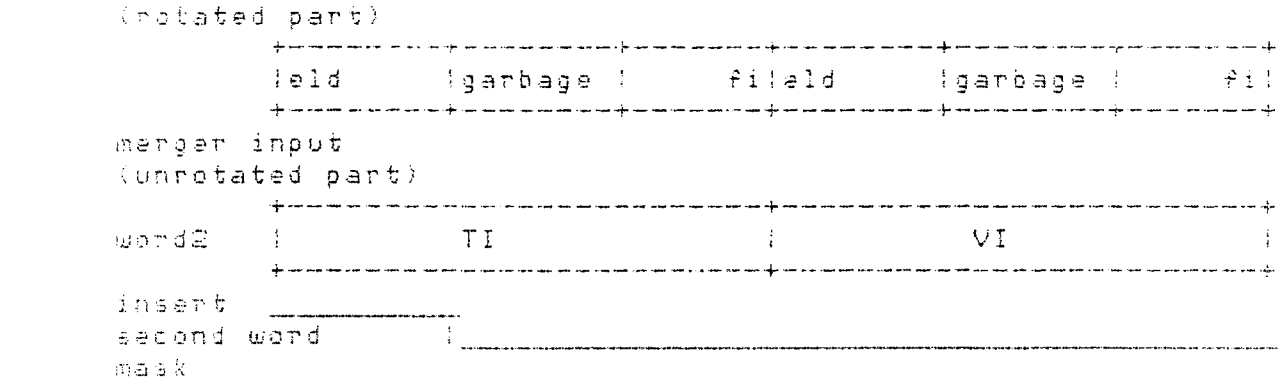
CYCLE1 - The offset and length specified have resulted in a cross word field condition. In this cycle the data to be inserted is driven over the FIU bus and onto both the TI and VI busses, then rotated left by (offset+length) and stored in the MDR.



CYCLE 2 - Word1 is driven onto the TI and VI busses and merged with the MDR. The insert first word mask is used, the type half of word1 is stored in the register file on the TYPE board, and the result of the insert is driven onto the FIU bus and stored in the register file on the VALUE board.



CYCLE 3 - Word2 is driven onto the TI and VI busses and merged with the MDR. The insert second word mask is used and the result is stored in the TAR and VAR.

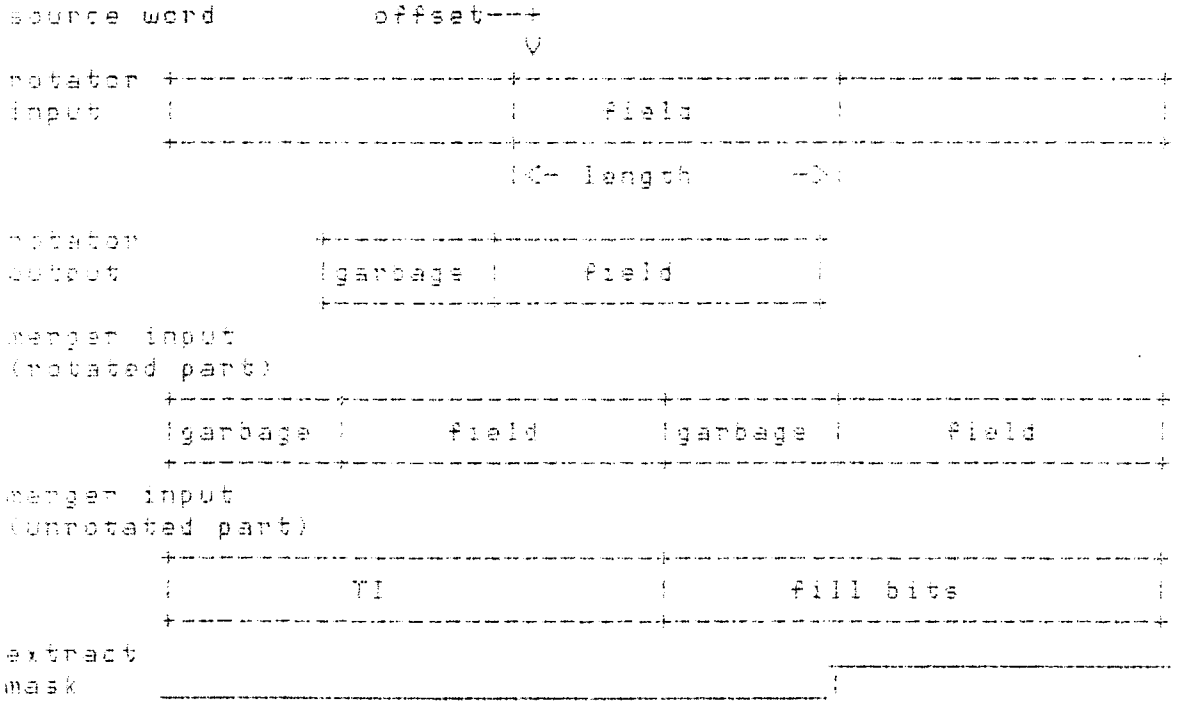


CYCLE 4 - The result of the first insert operation now stored on the register file are written back to word1.

CYCLE 5 - The result of the second insert operation now stored on the TAR and VAR are written back to word two.

4.3. Extract

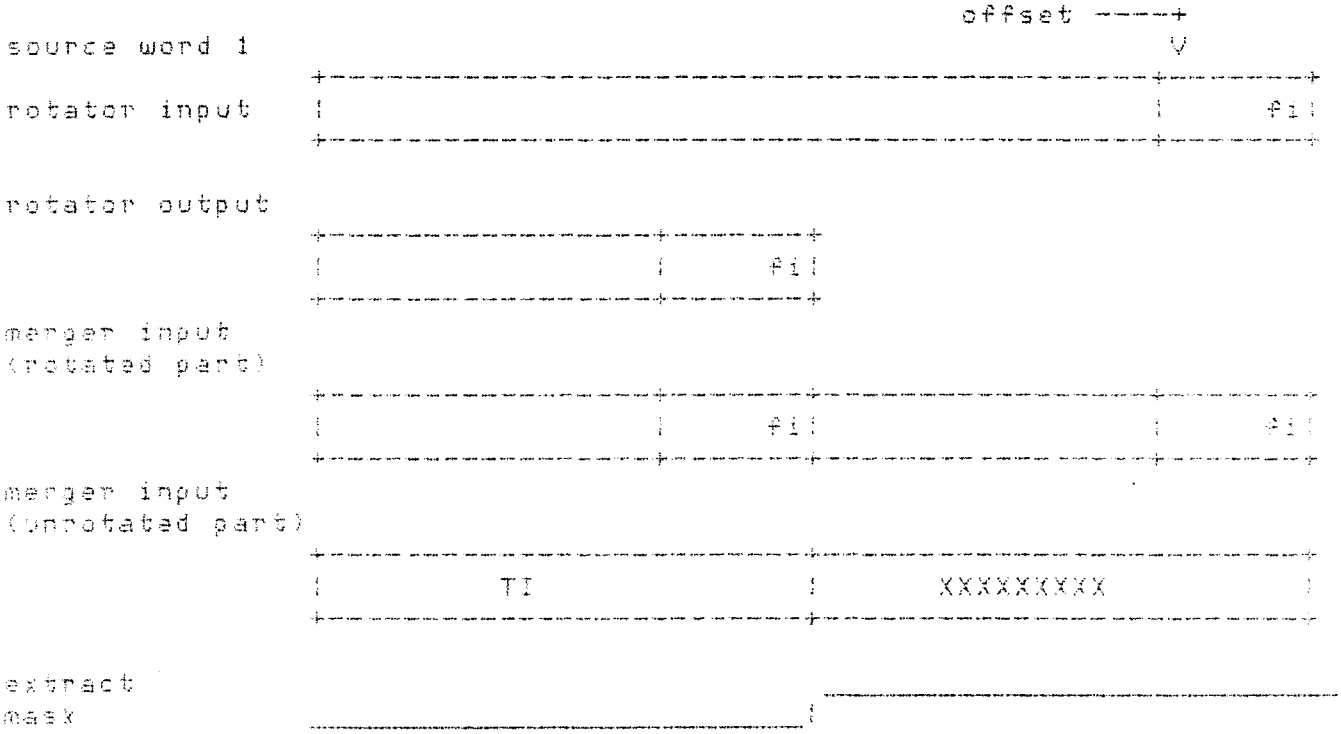
CYCLE 1 - The offset, length, and fill mode parameters are specified as literals or selected from previously latched values. The source word is driven onto the TI and VI busses and rotated left by (offset+length) to right justify the field being extracted. The sign bit is is selected to be output from the MERGE_VMUX so that the extracted field and the sign bit can be merged and either driven out the FIU bus into the register file or stored in the VAR.



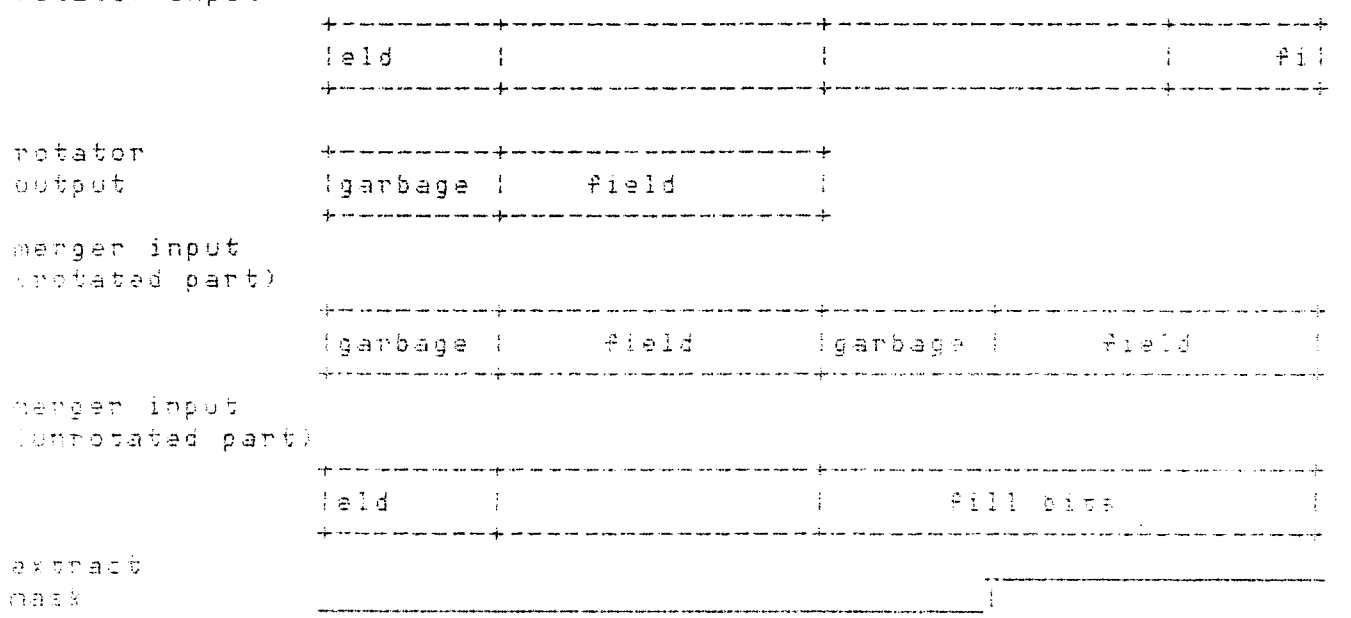
If fill mode = zero fill then the fill bits will be all zero's, and if the fill mode = sign extend then the fill bits will be the MSB of the extracted field.

4.4. Cross Word Extract

CYCLE 1 - The offset and length parameters have generated the cross word field condition. The first source word is input to the rotator on the TI and VI busses and the VALUE half of the word is extracted and stored in the VAR. This is done by specifying a literal offset of 64 and a literal length of 64.



CYCLE 2 - The type half of the second source word is driven onto the TI bus and the VAR is driven onto the VI bus. The latched offset, length, and fill mode parameters are selected and an extract is done. The result can be stored in the VAR or driven onto the FIU bus and stored in the register file.

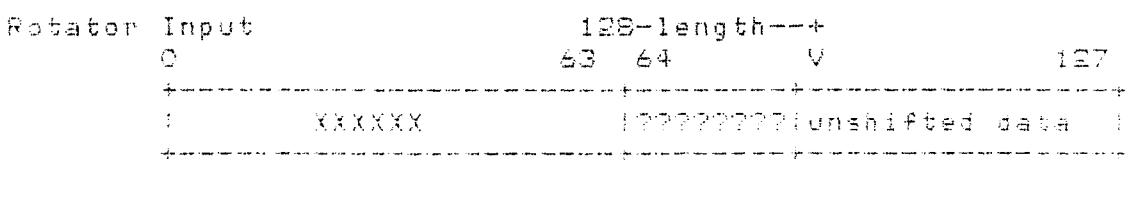


4.5. Using the FIU as a General Purpose Shifter

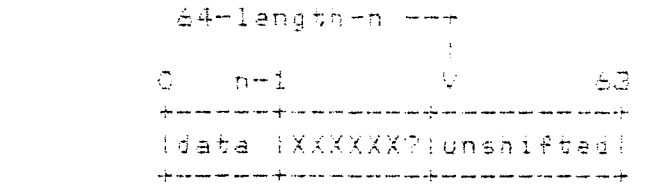
The following examples show how to use the FIU as a general purpose shifter to shift data on the Value part of the rotator input. The FIU operation and parameters selected will depend on the direction of the shift and whether the field being shifted is left or right aligned.

4.5.1. Right Aligned data, Right Shift by n bits

This is really a normal extract operation!



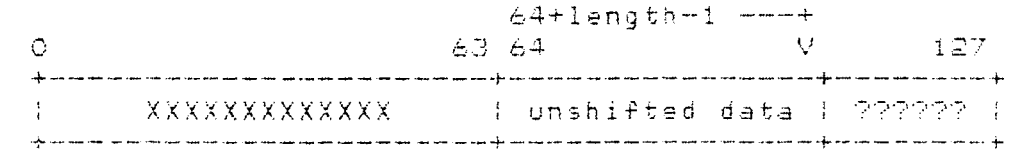
Operation = extract
Offset = 128-length
Length = length-n



Merge Mask Start Bit = 128-length+n
Merge Mask End Bit = 127

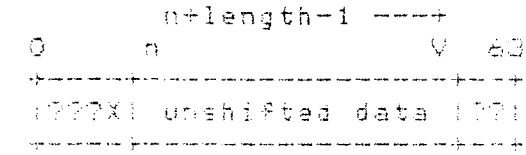
4.5.2. Left Aligned Data, Shift Right n bits

Rotator input



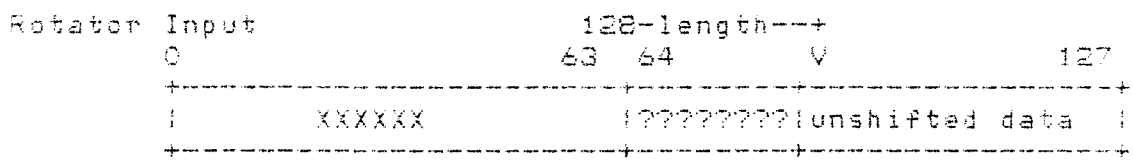
Operation = Extract
Offset = 64
Length = 64-n

Rotator Output

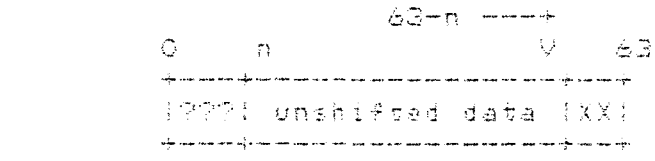


Merge Mask Start Bit = 128-(64-n) = 64+n
Merge Mask End Bit = 127

4.5.3. Right Aligned Data, Shift Left n Bits



Operation = Insert
Offset = 128-length-n *** Offset must be greater than 63 or the hardware selecting the participating bits will not work properly ***
Length = Length

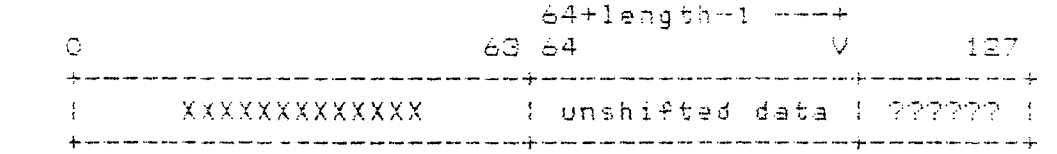


Merge Mask Start Bit = 128-length-n
Merge Mask End Bit = 127-n

None of the FIU operations will generate the correct rotate amount and merge mask to accomplish this shift in one cycle. The rotated data would have to be stored in the MDR during the first cycle, and the offset and length would have to be changed to generate the correct merge mask during the second cycle.

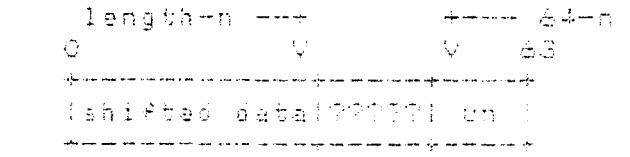
4.5.4. Left Aligned Data, Shift Left n Bits

Rotator input



Operation = Insert
Offset = 64
Length = 64-n

Rotator Output

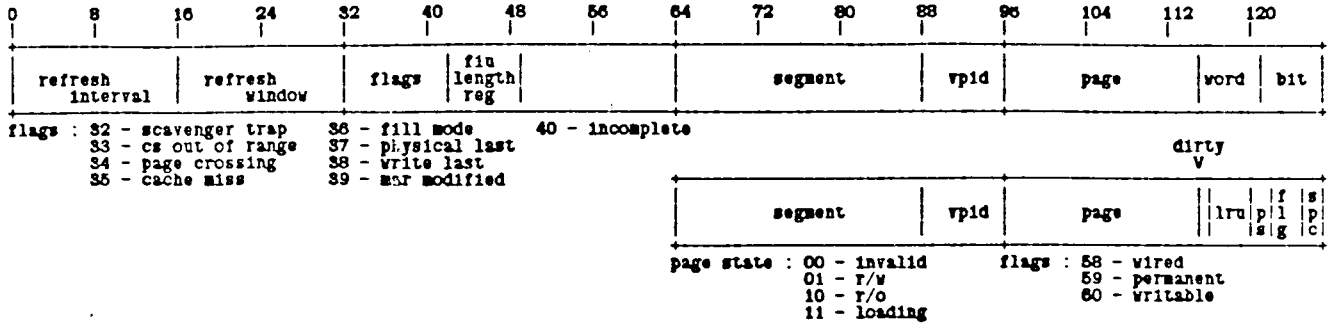


Merge Mask Start Bit = 64
Merge Mask End Bit = 127-n

5. Diagnostics

6. Physical Specification

Table of Contents



SOME USEFUL TAGS

Discrete	: 00 (80)	Record	: 44 (C4)	Subprogram	: 06 (86)	Null Subprogram	: 76	Seg Heap	: 38 (B8)		
Access	: 10 (90)	Variant Record	: 4C (CC)	(Elaborated)	: 16 (96)	Utility "	: 68 (E8)				
Task	: 18 (98)	Vector	: 6C (EC)	(Visible)	: 26 (A6)	Accept "	: 48 (C8)				
Package	: 58 (D8)	Matrix	: 74 (F4)	(Visible &		Interface "	: 58 (D6)				
Float	: 08 (88)	Array	: 7C (FC)	Elaborated)	: 36 (B6)	Exception Var	: 7E (FE)				

BLOCKED STATES

Unblocked	: 00	Terminable At End	: 07	In FS Rendezvous	: 0E	Blocking On Accept	: 18
Declaring Module	: 01	Blocking On Entry	: 08	In Wait Svc	: 0F	Blocking On Select	: 19
Awaiting Activation	: 02	Delaying On Entry	: 09	Delay In Wait Svc	: 10	Delaying On Select	: 1A
Activating Module	: 03	Attempting Entry	: 0A	Blocking On Abort	: 11	Await Children Select	: 1B
Activating Tasks	: 04	Delaying	: 0B	Deleted	: 12	Terminable In Select	: 1C
Awaiting Task Activ	: 05	Aborting Module	: 0C	Aborted While In MTS	: 13		
Awaiting Children	: 06	Terminated	: 0D	In_MTS_RendezVous	: 14