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



LL            PPPPPPPP      TTTTTTTTTT                  44    44
LL            PPPPPPPP      TTTTTTTTTT                  44    44
LL            PP      PP        TT                      44    44
LL            PP      PP        TT                      44    44
LL            PP      PP        TT                      44    44
LL            PP      PP        TT                      44    44
LL            PPPPPPPP          TT                      4444444444
LL            PPPPPPPP          TT                      4444444444
LL            PP                TT                          44
LL            PP                TT                          44
LL            PP                TT            ....          44
LL            PP                TT            ....          44
LLLLLLLLLL    PP                TT            ....          44
LLLLLLLLLL    PP                TT            ....          44



*START* Job SPEC Req #697 for EGB    Date  3-Dec-82  2:02:54 Monitor: Rational M
File RM:<MICRO-ARCH.SEQUENCER>SPEC.LPT.4, created: 21-Oct-82 23:45:44
       printed:  3-Dec-82  2:02:54
Job parameters: Request created: 3-Dec-82  1:53:38   Page limit:171   Forms:NORM
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:
```

R1000 Microsequencer Specifications

Draft 3

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
casef           --jump to the branch address plus the lsb 14 bits
                --of the FIU_DATA from the last cycle, if the
                --the condition is false
case_call *     --same as the case, except PC + 1 is pushed onto
                --the stack,and the call is unconditional.
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 neither readable nor writable. It is therefore necessary to avoid taking 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.

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
casef	PC + 1	branch_addr + FIU_data(50:63)
case_call	branch_adr + FIU_data(50:63)	" *
push	PC + 1	PC + 1

(* -- For both cases true and false)

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

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 timing some conditions 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 of the tested condition's value (true or false). And the actual condition is latched at the end of the cycle and tested during the next cycle. 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_timing		
0	0	branch on the early condition
0	1	branch on the latched condition
1	0	branch on the late condition, hint is true
1	1	branch on the late condition, hint is false

Table 2-2: Branch Timing

To ease the complexity of the hardware a conditional dispatch true and a hint false or a conditional dispatch false and a hint true are not allowed! (NO RARELY DISPATCHES.)

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. Early abort memory if the dispatching instruction does not require a memory read. (The translator must start a memory read during every dispatch or conditional dispatch.)
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:

011 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 (Zero extended if the lex level is 0 or 1, and sign extended is the lex level is 2-16.).

110 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).

001 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).

100 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).

111 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.)

010 CONTROL_READ_CONTROL_PRED
Start a control read. The stack name is the current_name register and the offset is the control_ pred register.

000 CONTROL_READ_(INNER-PARAMS)
Start a control read. 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. The macro events are cleared by some action that is taken during the handler. (All of the early macros must be cleared at least two cycles before the handler executes a return).
The macro events and some of the characteristics are:

MEMORY	Early /Late	priority	memory address	micro address
refresh memory	E	0		0138
SYSBUS				
sysbus_status	E	2		0128
sysbus_packet	E	3		0120
slice_timer	E	5		0110
gp_timers	E	6		0108
SPARES				
spare	E	1		0130
spare	E	4		0118
spare	E	7		0100
SEQUENCER				
CSA_underflow	L	8		0178
CSA_overflow	L	9		0170
spare	L	10		0168
resolve_ref	L	11		0160
TOS_optimization_err	L	12		0158
spare	L	13		0150
break_class	L	14		0148
IBUF_empty	L	15	PC+1	0140

(0 is the highest priority event)

Table 2-3: Macro Events

The macro events generated by the sequencer are:

BREAK_CLASS The sequencer contains a 16 bit register which specifies 15 break classes. 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 15 break classes or it is the "no_break_class" class.) If the break class of the current instruction is set in the break class register the break_class macro will occur. (NOTE: To break on every instruction (except an instruction of the "no_break_class") every bit of the break register must be set. An instruction of the "no_break_class" will never cause a break class macro, under any circumstances.)

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 complement of 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 valid 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), (the hardware will disable the IBUFF macro event automatically during any cycle where the IBUFF is loaded). (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 EARLY EVENT 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. (Persistent Events are cleared either when tested or when cleared by a micro-order. See the individual specs. for details.)

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.)

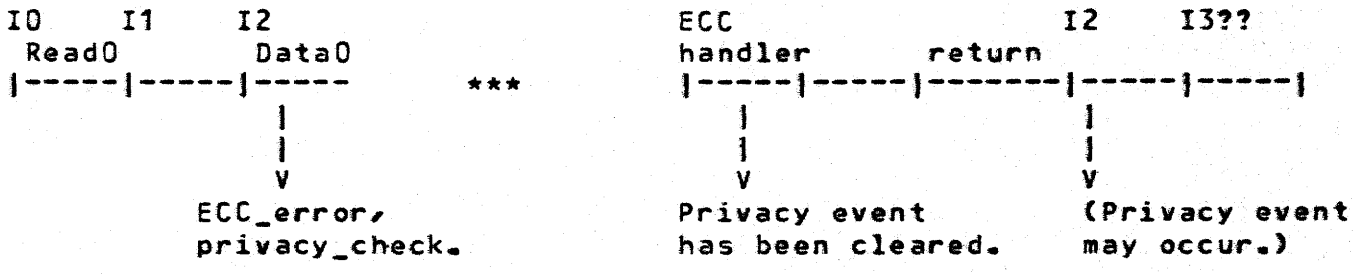
Some of the events are specifiable. These events are normally disabled and are only enabled when specifically selected by the microcode. (If a specifiable event is not selected it is not remembered, but it is testable.)

	cond	E/L	mask	specify	priority	persistant	micro addr
MEMORY MONITOR							
cache_miss	X	E			1	X	188
ECC error		E			2		190
page_crossing	X	E			12	X	1D8
TYPE CHECK ERRORS							
class error	X	E		E	5		1A8
binary_eq_privacy_check	X	E		E	6		1B0
binary_op_privacy_check	X	E		E	7		1B8
[tos]_op_privacy_check	X	E		E	8		1C0
[tos-1]_op_privacy_check	X	E		E	9		1C8
SEQUENCER							
field_number_error	X	E		E	4		1A0
SYSBUS							
new_packet	X	L	X		12	X	1E0
new_status	X	L	X		13	X	1E8
break	X	E	X		0	X	180
SPARES							
		E	?	?	3	?	198
		L	?	?	14	?	1D0
		L	?	?	11	?	1F0

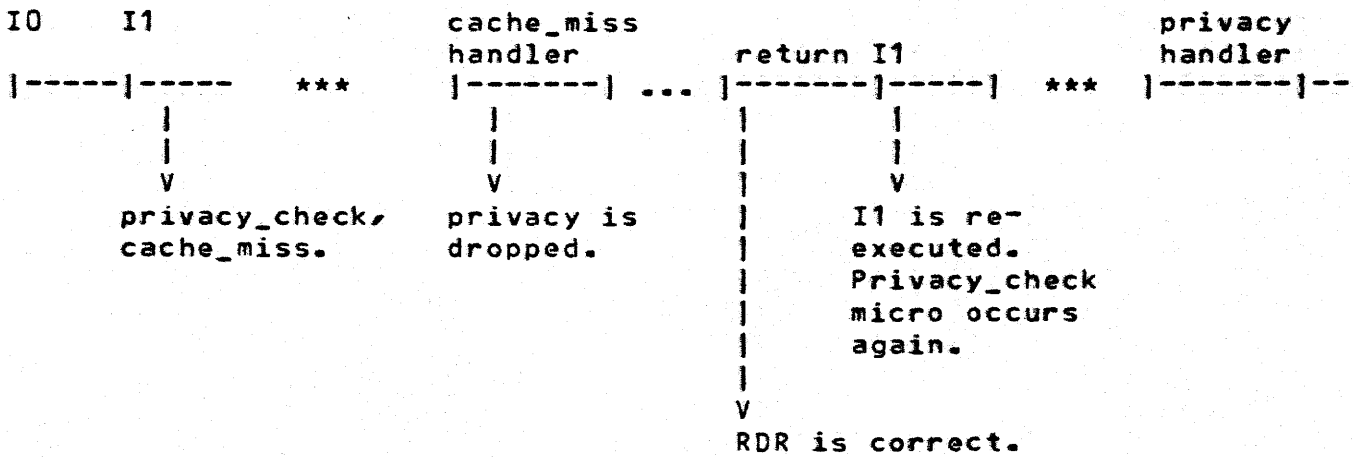
(Highest priority is zero)

Table 2-4: Micro Events

Some examples of micro events and how they are handled.



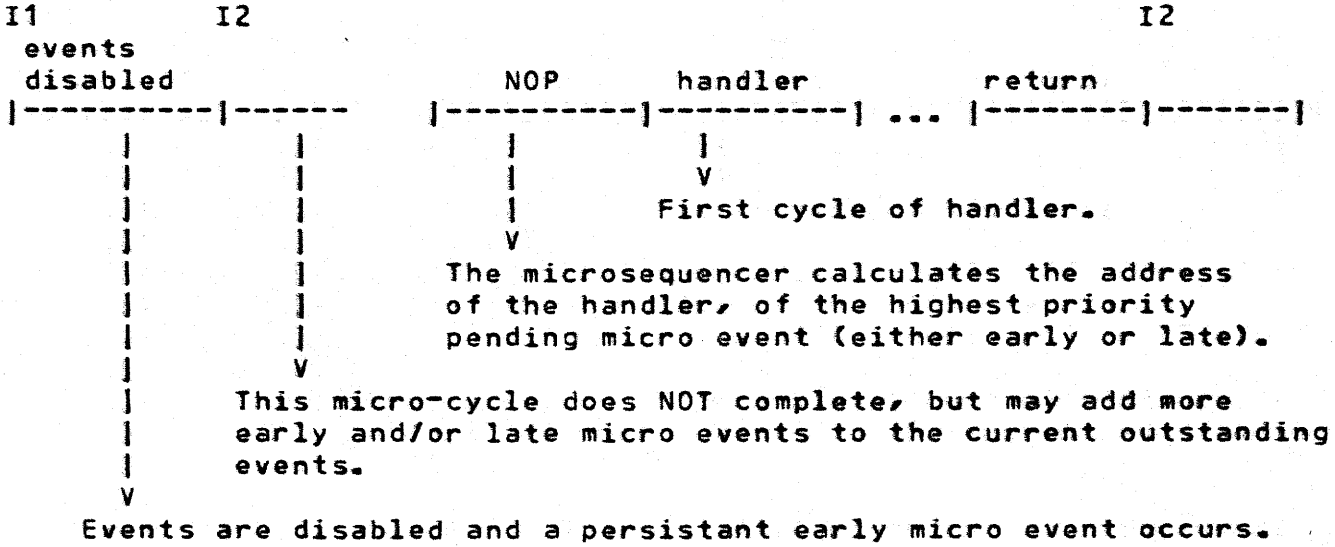
(*** A microcode cycle occurs, but the state change clocks are inhibited.)



The microsequencer has a one bit microcode field which specifies during each instruction if micro events are enabled or disabled. The disabling will turn off all events, except for ECC error, which can never disabled from the micro-sequencer (only from the memory monitor). If micro events are disabled during a micro-instruction, no early micro event will occur during that cycle (except for ECC error). 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 persistant 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.

An example of timing for disabled micro events.



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 (if 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 if the lex level is greater than one).

Microcode has the capabilities to both read and write registers in the resolve circuit (but can NOT do both during the same cycle). 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), 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, all the validity bits at a greater lex level can be cleared, or all the validity bits can be cleared. A three bit lex level validity command control field in the random field determines the control of the validity bits during each micro-cycle.

The resolve circuit is also used to calculate the control or type addresses that the sequencer starts during some dispatches (see the dispatch section). Consequently the lex level address must be set to the current lex level during dispatches, and the microcycle can NOT be writing into the resolve rams.

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). (During a bad hint cycle the validity bit associated with the tos_latch will be restored to it's previous value.)

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 indicates if the macro-instruction is restartable or not restartable. If the instruction is restartable, the address 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.) The random field contains a two bit field

which allows microcode to set the restartable state to; not restartable, restartable @(macro_pc), restartable @(macro_pc - 1), or nop.

(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.) (NOTE: The fiu_data that is pushed onto or read the stack is negative logic. All other data pushed onto or read from the stack is positive logic. These means that the stack will only work properly if the data that is pushed onto the stack from the fiu_bus, was read off of the stack using the fiu bus, or the data is complemented.)

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 a comparator for checking field numbers during the execution of the field ops. If enabled the checker will cause the field number error micro event. The variant field check compares bits (79:88) of the sequencer bus to bits (6:15) of the current instruction. An unequal comparison will generate 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 and their encodings 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.) (This field of the decode rams is encoded by complementing the number of free locations!)

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.) (This field is also complemented in the decode rams.)

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 (one of which is no break_class). (If the instruction belongs to a break_class which is currently 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 one 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.

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

Table 2-5: R1000 Condition Partioning

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 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.

2.12. Memory Aborts

The memory monitor and the micro-sequencer have two abort controls of memory, early and late abort. The early aborts will prevent a new memory operation from starting during the current micro-cycle (without affecting any memory operations already in progress). The late abort will stop any memory operation in cycle 1, but will allow cycle 0 memory operations (see the memory monitor spec. for the definition of memory cycles).

The micro-sequencer will set the early abort during bad conditional memory references, dispatches that don't start memory, micro events hiccups, late macro event hiccups, and all bad hints (a bad_hint occurs the cycle after and incorrect hint is specified). The memory monitor provides two signals to the micro-sequencer to identify conditional memory references; memory_ref_is_conditional and condition_polarity. If the memory reference is conditional and the current condition doesn't match the condition_polarity the micro-sequence detects a bad condition memory reference.

The micro-sequencer sets the late abort during bad hints that were caused by hint dispatches and during micro event hiccups.

2.13. Miscellaneous Registers and other Junk

There are a few other register and function units that deserve honorary mention.

1. CURR_NAME reg. -- This register holds the name of the currently running task (32 bits; vpid and segment). The register is used when reading the new_offset, control_pred, or contorl_top registers. The register can be loaded from bits 0:31 of the type half of the sequencer bus, when specified in the random field.

2. SAVE_OFFSET reg. -- This 20 bit register input is the offset output of the resolve circuit. The register is clocked when specified in the random field. The register should be clocked

during every dispatch, and is used to save the conrol offset that is used by dispatching instructions that automatically start a control read. (The register saves offsets during call and exit instructions.)

3. CONTROL_PRED reg. -- This 20 bit register is used to hold the offset of the previous mark words (top of the previous frame). This register is changed during context switch and all frame changing instructions (call, exit, etc.). The clocking of this register is specified in the random field.

4. CONTROL_TOP reg. -- This 20 bit register is used to hold the offset of the top of the current task's control stack. The clocking of the register is specified in the random field. (At macro instruction boundaries this register should contain the same value the control top register on the memory monitor board contains.) The regester is increment and decremented every cycle that a push_control_stack or pop_control_stack respectively.

5. CONTROL_MUX -- This mux selects between the type bits 37:56 of the sequencer bus, and bit 37:56 of the fiu bus. The output of the mux is the input to both the CONTROL_PRED and the CONTROL_TOP. The select is a field in the random.

6. MACRO_PC reg. -- This 39 bit register (24 bits of segment, 12 bits of offset, and 3 bits of index) contains the current macro pc. The clocking of the segment portion of the macro_pc is controlled by a random field. The segment portion of the macro_pc can only be loaded from the val have of the sequencer bus. The offset and index portion of the macro_pc register are countable. The random field contains a two bit mode field which allows four control operations; hold, increment, decrement, and load. When loaded the data comes from the output of the MACRO_MUX (see below). The random field can also specify that the macro_pc (offset and index) and the ibuff operations are done conditionally (on the current EARLY condition). If the condition is FALSE the macro_pc mode is changed to hold.

7. RETURN_PC reg. -- This 39 bit register (24 bits of segment, 12 bits of offset and 3 bits of index) contains the new macro pc, to use after an exit macro-instruction is used. The clocking of the register is controlled by a random field, and the data is always the current macro_pc.

8. BRANCH_ADDER -- This adder is used to calculate the new macro pc, if the dispatching or current macro-instruction is a jump. (During ibuff macro events the adder will also calculate the address of the new ibuff.) During dispatches the adder assumes the dispatching instruction is a jump, and adds the offset and index of the macro_pc to bits 5:15 of the dispatching

instruction (sign extended). During non-dispatching micro-instructions the adder adds the macro_pc to bits 5:15 of the current instruction (sign extended).

9. MACRO_MUX -- This mux selects either the val half of the sequencer bus or the output of the branch added as input to the offset and index half of the MACRO_PC. The select is a field in the random.

10. CODE_MUX -- The code mux selects between the macro_pc, the return_pc, and the output of the branch_adder. The output of the code_mux may by read on the sequencer internal bus, or used as a code address on the address bus. The select is a field in the random. (During all dispatchs the random field must select the output of the branch adder.)

11. MEMORY_ADDRESS_LOGIC -- This logic determines which address to drive onto the address bus; either from the CODE_MUX or from the NAME and OFFSET busses. During dispatches that start a read operation from memory, the select is chosen based on the output of the decode rams. During any other micro-instruction the select is based on the value of a field in the random.

12. IBUFF reg. -- The instruction buffer holds 8 macro-instructions (128 bits) and is loadable from the type and val busses only. The load control is from load_ibuff field of the random. The cond_load field of the random, when specified simultaneously with the load_ibuff random, will only load the ibuff if the current (early) condition is TRUE.

13. INSTR_MUX -- The instr_mux selects between the currently selected macro-instruction in the ibuff and bits (48:63) of the val half of the sequencer bus. The select is determined by the load_curr_instr field of the random. Loading the current instruction forces the mux to select the val half of the sequencer bus as input to the current instruction register.

14. CURR_INSTR -- The current instruction register is automatically loaded during every successful dispatch. The register can also be loaded with data from val bits (48:63) of the sequencer bus by specifying load_curr_instr field of the random. During ALL bad hints the curr_instr is restored to it"s previous value.

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.

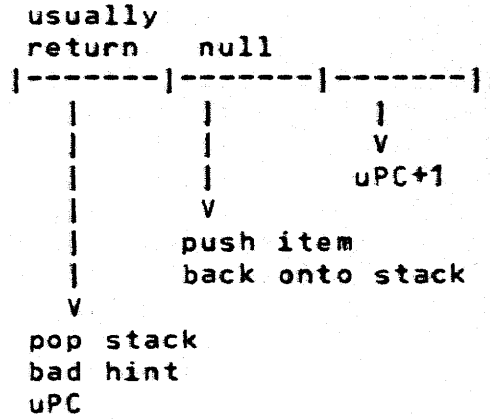
3. If there are any early micro events, stop the instruction from completing, and push the current micro-address onto the stack. During the next cycle execute a null and calculate the micro address of the micro event handler. The next micro-instruction is the first instruction of the handler.

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

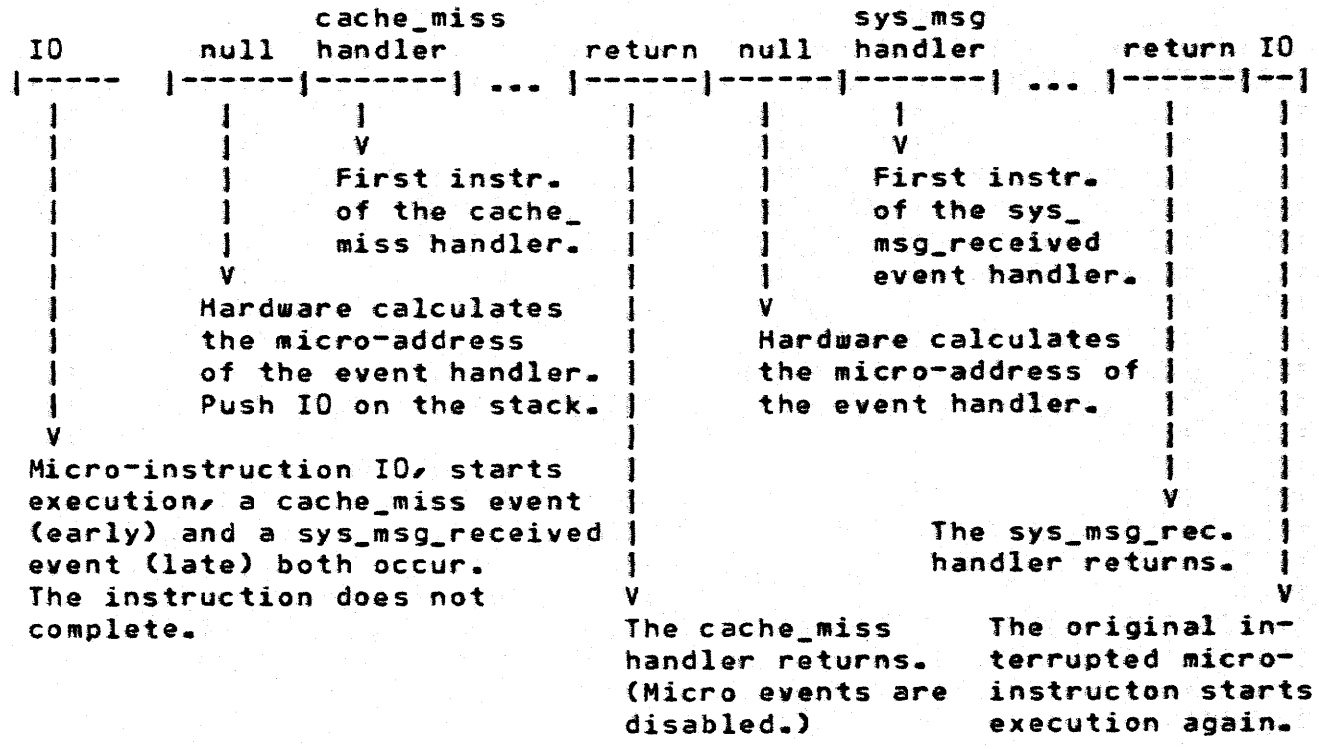
5. If the instruction is a dispatch and there are only late macro events, the next instruction is a null. The following micro-instruction will be the first instruction of the macro event handler.

6. 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: "Two persistant micro events occur at once"



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

BRANCH TYPE (4 bits)

0001	brt	conditional branch (branch if true)
0000	brf	conditional branch (branch if false)
0011	br	unconditional branch
0110	cont	continue (PC + 1)
0101	callt	conditional call (call if true)
0100	callf	conditional call (call if false)
0111	call	unconditional call
1000	returnt	conditional return (return if true)
1001	returnf	conditional return (return if false)
1010	return	unconditional return
1100	dispt	conditional dispatch (dispatch if true)
1101	dispf	conditional dispatch (dispatch if false)
1110	disp	unconditional dispatch
1011	casef	jump to the branch address plus the 14 lsb bits of the FIU_DATA from the last cycle, if the condition is false
1111	case_call	same as the case, except PC + 1 is pushed onto the stack, and the branch is unconditional
0010	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.)

00 EARLY CONDITION -- Test the currently selected early condition.

01 LATCHED CONDITION -- Use the output of the latch.

10 HINT TRUE -- 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.

11 HINT FALSE -- 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.

(NOTE: The translator must also set this field to "hint true" during unconditional branch types.)

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 ADDRESS (2 bits)

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

00 CURRENT_LEX -- Use the current lex level.

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

10 1 -- Address the outer frame.

11 0 -- Address the import frame.

(NOTE: This field must be set to "CURRENT_LEX" during dispatches (conditional or otherwise).)

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.)

0 DISABLE_ALL_MICROS

1 NOP

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.)

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

		start	end	#_bits
111	RESOLVE_OUTPUT			
	resolve_frame.number	0	23	24
	resolve_frame.proc	24	31	8
	ZEROS	32	36	5
	resolve_offset	37	56	20
	ZEROS	57	59	3
	number_in_the_csa**	60	63	4
	ZEROS	64	71	8
	code.segment	72	95	24
	ZEROS	96	108	13
	code.offset	109	120	12
	code.index	121	123	3
	current_lex_level	124	127	4

101	CONTROL_PRED			
	current_name.number	0	23	24
	current_name.proc	24	31	8
	ZEROS	32	36	5
	control_pred	37	56	20
	ZEROS	57	59	3
	number_in_the_csa**	60	63	4
	ZEROS	64	71	8
	code.segment	72	95	24
	ZEROS	96	108	13
	code.offset	109	120	12
	code.index	121	123	3
	current_lex_level	124	127	4
100	CONTROL_TOP			
	current_name.number	0	23	24
	current_name.proc	24	31	8
	ZEROS	32	36	5
	control_top	37	56	20
	ZEROS	57	59	3
	number_in_the_csa**	60	63	4
	ZEROS	64	71	8
	code.segment	72	95	24
	ZEROS	96	108	13
	code.offset	109	120	12
	code.index	121	123	3
	current_lex_level	124	127	4
110	SAVE_OFFSET			
	current_name.number	0	23	24
	current_name.proc	24	31	8
	ZEROS	32	36	5
	save_offset	37	56	20
	ZEROS	57	59	3
	number_in_the_csa**	60	63	4
	ZEROS	64	71	8
	code.segment	72	95	24
	ZEROS	96	108	13
	code.offset	109	120	12
	code.index	121	123	3
	current_lex_level	124	127	4

001	CURRENT_INSTRUCTION				
		current_name.number	0	23	24
		current_name.proc	24	31	8
		ZEROS	32	36	5
		control_pred	37	56	20
		ZEROS	57	59	3
		number_in_the_csa**	60	63	4
		ZEROS	64	71	8
		code.segment	72	95	24
		ZEROS	96	108	13
		code.offset (3 msb's)	109	111	3
		current_instruction	112	127	16
010	DECODING_INSTRUCTION				
		current_name.number	0	23	24
		current_name.proc	24	31	8
		ZEROS	32	36	5
		save_offset	37	56	20
		ZEROS	57	59	3
		number_in_the_csa**	60	63	4
		ZEROS	64	71	8
		code.segment	72	95	24
		ZEROS	96	108	13
		code.offset (3 msb's)	109	111	3
		decoding_instruction	112	127	16
011	TOP_OF_MICRO_STACK				
		resolve_frame.number	0	23	24
		resolve_frame.proc	24	31	8
		ZEROS	32	36	5
		resolve_offset	37	56	20
		ZEROS	57	59	3
		number_in_the_csa**	60	63	4
		code.segment	72	95	24
		ZEROS	96	108	13
		code.offset (3 msb's)	109	111	3
		top_of_micro_stack	112	127	16

Where code.* is the output of the code_mux. This output is either the macro_pc, return_pc, or the output of the branch adder. The mux selects are determined by the random field.

** The number_in_the_csa is the number valid when the current cycle started, and does not reflect any csa operations that happen during the current cycle. This value is not valid during the two micro-cycles that follow a micro-instruction that does a pop_down_to.

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. The least significant two bits of the random address are used to address the code_mux in the following manner:

```
00  macro_pc.segment, macro_pc_address plus current_instr(5..15)
01  macro_pc.segment, macro_pc_address
10  return_pc.segment, macro_pc_address plus current_instr(5..15)
11  return_pc.segment, return_pc_address
```

The operations controlled by the random are:

Addr_control (1)

-- This random selects between control or import memory addresses from the name, offset bus and a code address from the code bus. (During dispatches the hardware will automatically override this field if the dispatching instruction needs to start a memory operation.)

Load_macro_pc_h (1)

-- This random unconditionally loads the segment portion of the macro pc from the val half bits (8:31) of the sequencer bus.

Macro_pc_low (2)

-- This random controls the offset and index portion of the macro pc. The mode bits work as follows:

```
0  1     load
1  1     increment
0  0     decrement
1  0     hold (nop)
```

During a cycle that may dispatch, this random must be set to nop and the hardware will automatically perform the increment if the dispatch is done. If the cond_load random is set and the current condition is false, the macro pc will hold, regardless of the value of this random. (The macro_pc is not changed until the end of the cycle.)

Macro_mux (1) -- This random selects between the val half of the sequencer bus and the output of the branch adder, as input to the low half of the macro pc. If the bit is set the ouput of the branch adder is selected.

Load_return_pc (1)
-- If set the return_pc register will be loaded from the macro_pc during this cycle. (This random is active low.)

Load_ibuff (1) -- If set the ibuff will be loaded with bits (0:63) of the typ half of the sequencer bus and (0:63) of the val half of the sequencer bus. If the Cond_load random is set the load will only occur if the current condition is true. (If the load_random is not set the Cond_load random has no affect on the ibuff.) (This random is active low.)

Cond_load (1) -- This random affects the operation of the low half of the macro pc and the loading of the ibuff. If this random is set and load_ibuff is set, the load will only occur if the current condition if true. If this random is set and the macro_pc_low is set to anything except hold, the operation will only occur if the current condition is true. This random can only be used in conjuction with a early condition. (This random is active low.)

Load_break_mask (1)
-- If this random is set the break mask will be loaded from bits (32:47) of the val half of the sequencer bus. (This random is active low.)

Control_mux (1) -- This random selects between the bits (37:56) of the type half of the sequencer bus and bit (37:56) of the fiu bus, as input to the control_top and control_pred registers. If the random is set the type bits will be selected.

Load_control_top (1)
-- If this random is set the control_top register will be loaded from the output of the control_mux. (This random is active low.)

Load_control_pred (1)
-- If this random is set the control_pred register will be loaded from the output of the control_mux. (This random is active low.)

Load_save_offset (1)
-- If this random is set the save_offset register will be loaded from the offset portion of the resolve circuit. (This random is active low.)

Load_curr_name (1)
-- If this random is set the current_name (curr_name reg in the block diagram) register will be loaded from

bits (0:31) of the type half of the sequencer bus. (This random is active low.)

Load_current_instr (1)
-- If this random is set the current_instr register will be loaded from bits (48:63) of the val half of the sequencer bus. (This random is active low.)

Push_stack (1) -- If this random is set the complement of the data on bits (48:63) of the fiu bus will be pushed onto the micro stack. (This random is active low.)

Pop_stack (1) -- If this random is set the micro stack will pop one item off of the stack.

Clear_stack (1) -- If this random is set the micro stack will be cleared. (This random is active low.)

Restartable_contol (2)
-- This random controls the state of the two restartable bits as follows:

0	0	not restartable
0	1	nop
1	0	restartable at macro pc
1	1	restartable at (macro pc - 1)

Load_current_lex (1)
-- If this random is set the current lex register will be loaded with the value of bits (60:63) of the val half of the sequencer bus. (This random is active low.)

Load_resolve_name (1)
-- If this random is set the name half of the resolve ram will be loaded with the value of bits (0:31) of the type half of the sequencer bus. (This random is active low.)

Load_resolve_offset (1)
-- If this random is set the offset half of the resolve ram will be loaded with the value of bits (37:56) of the type half of the sequencer bus. (This random is active low.)

Lex_validity (3)
-- This random controls the 16 lex level validity bits associated with the resolve circuit. The commands are encoded as follows:

0	0	0	Clear Lex Level
0	0	1	Set Lex Level
0	1	0	Illegal
0	1	1	Nop
1	0	0	Clear Greater Than Lex Level
1	0	1	Illegal
1	1	0	Illegal
1	1	1	Clear all Lex Levels

The lex level used is determined by the lex level address field of the microword.

Validate_tos_optimizer (1)
-- If set the Tos optimizing latch will be valid during the current cycle. If the cycle doesn't perform a dispatch the valid bit will also stay set in the subsequent cycle.

Check_field_number (1)
-- If set this the field check micro event is enabled on the micro sequencer.

Disable_macro_events (1)
-- If set macro events are disabled during the current cycle.

Halt (1) -- If set the current cycle will not complete (no state will be updated) and control will be transfered to the diagnostic processor subsystem.

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	lex_level_is_import	(L)
1000011	valid_lex~	(L)
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_break_class~	(L)
1001101	ME_ibuff_empty~	(L)
1001110	spare	
1001111	spare	
1010000	ME_refresh_memory	(E)
1010001	ME_sysbus_status	(E)
1010010	ME_sysbus_packet	(E)
1010011	ME_slice_timer	(E)
1010100	ME_gp_timer	(E)
1010101	Any_early_macro	(E)
1010110	spare	
1010111	Field_#_error	(L)

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.)

lex_level_is_import
: This conditon is false if the currently selected address to the resolve rams is zero.

TOS_LATCH_valid This condition is false 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 false if the dispatching of the decoding instruction would cause a CSA_underflow macro event.

ME_CSA_overflow~
: This condition is false if the dispatching of the decoding instruction would cause a CSA_overflow macro event.

ME_resolve_ref~ This condition is false if the dispatching of the decoding instruction would cause a resolve_ref macro event.

ME_TOS_opt_error~
: This condition is false if the dispatching of the decoding instruction would cause a TOS_opt_error macro event.

ME_break_class~ This condition is false if the dispatching of the decoding instruction would cause a break_class macro event.

ME_ibuff_empty~ This condition is false if the dispatching of the decoding instruction would cause a ibuff_empty macro event.

Any_early_macro This condition is true if the dispatching of the decoding instruction would cause an early macro event.

uE_field_number_error
: This condition is true if bits (6:15) of the current_instruction are equal to bits (15:24) of the val half of the sequencer bus.

ME_refresh_memory, ME_sysbus_status, ME_sysbus_packet, ME_slice_timer, and ME_gp_timer are generated on other boards and should be explained in the appropriate spec.

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.

Therefore NO conditional memory references may be started during a micro-cycle where the branch type field is a conditional or unconditional dispatch.

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.

Conditional dispatches may not be issued with the following conditions: lex_level_is_import or valid_lex~.

The casef 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.

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

1. Summary 1
2. Functional Description 1
  2.1. Branches 1
  2.2. Dispatch 4
  2.3. Macro Events 6
  2.4. Micro Events 9
  2.5. Resolve Circuit 13
  2.6. Tos_Latch 14
  2.7. Restartable State 14
  2.8. Micro Stack 15
  2.9. Field Number Checker 16
  2.10. Instruction Decoder 16
  2.11. R1000 Processor conditions 17
  2.12. Memory Aborts 18
  2.13. Miscellaneous Registers and other Junk 18
3. Some Timing Examples 20
4. Microword Specifications 22
5. Microcode Considerations 31
  5.1. Conditions 31
  5.2. Context Switch 34
  5.3. Microcode Restrictions 34
    5.3.1. Branches 34
    5.3.2. Sequencer Address Enables 34
    5.3.3. CYA 35

List of Tables

Table 2-1: Micro-Address Selection for Branch Types 3
Table 2-2: Branch Timing 4
Table 2-3: Macro Events 6
Table 2-4: Micro Events 11
Table 2-5: R1000 Condition Partioning 17