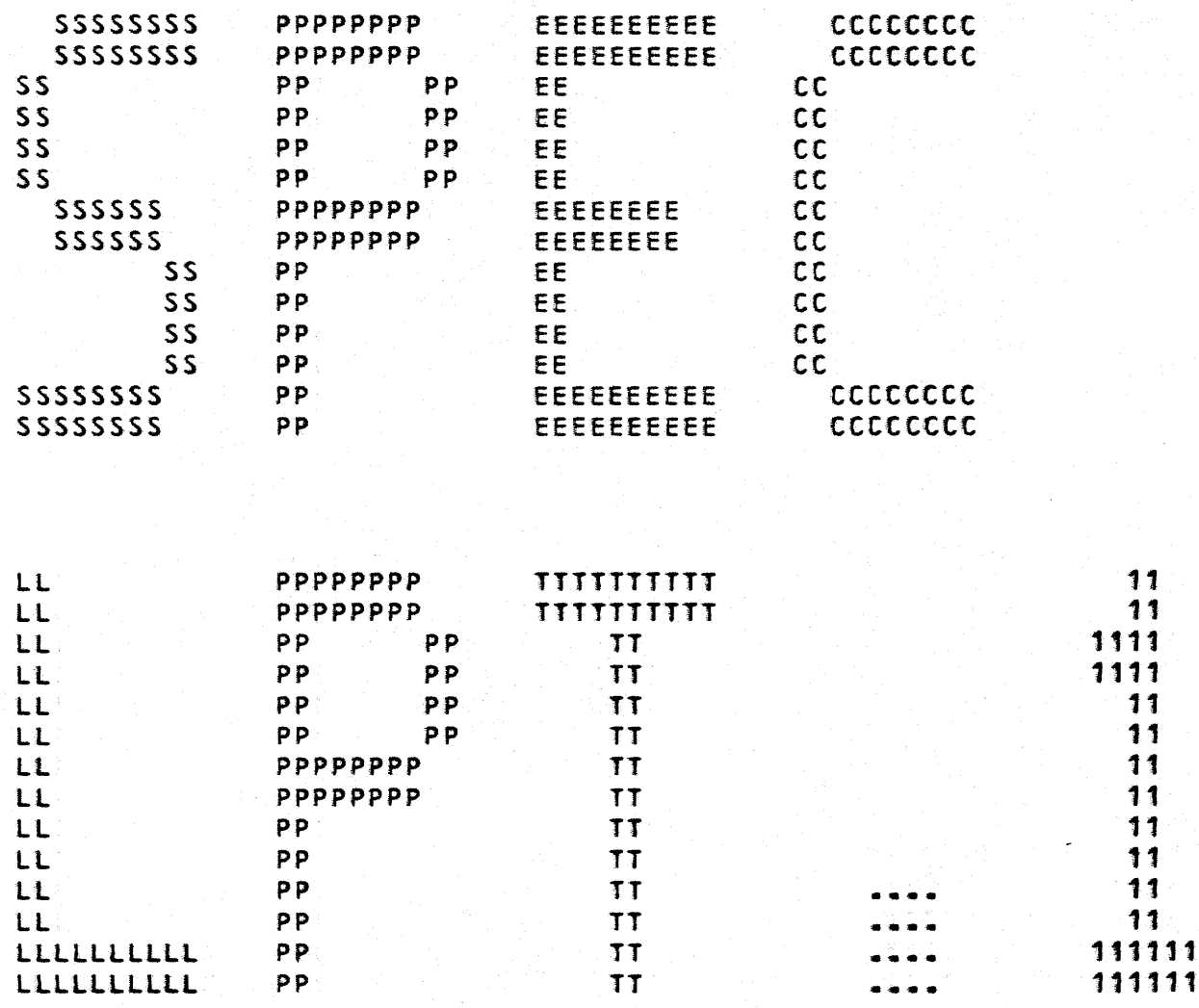

START Job SPEC Req #705 for EGB Date 3-Dec-82 1:58:40 Monitor: Rational M
File RM:<MICRO-ARCH.VAL>SPEC.LPT.1, created: 2-Aug-82 9:02:20
printed: 3-Dec-82 1:58:40
Job parameters: Request created: 3-Dec-82 1:56:48 Page limit:153 Forms:NORMA
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:A

Functional Specification of the Value Board

DRAFT 4

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

- George Bernard Shaw

1. Summary

This document completely describes the operational characteristics 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 interface and operation.

The organization of this document is as follows; Section 2 provides a detailed definition of the capabilities, 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 operational characteristics 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 temparary values that may be needed during execution of a microinstruction.

The next 16 locations in the RF contain the special purpose 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 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 REGISTER_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 2-1. 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 REGISTER_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.

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.

ZERO The constant value zero is sourced from the A Port of the Register File.

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	spare
010011	reg(loop_cntr)	reg(loop_cntr)	reg(loop_cntr)
010100	zero	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)

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 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 (through the SHIFT MUX), 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 a testable condition on any one of the boards to be sent over to the SEQUENCER to participate in a conditional branching operation (certain combinations of conditions from the VAL and TYPE board are also possible For more information about how these combination conditions, the reader is referred to the specification of the microsequencer). At the end of every cycle that the microcode selects a condition from the VAL board, the value of the selected condition gets latched on the VAL board and may be used in the following microcycle either to select the outcome of a conditional ALU operation, or be sent over to the sequencer board to be used as a branch condition. The value of this condition latch, called the VAL_PREVIOUS latch, only changes at the end of cycles when VAL board conditions are selected by the sequencer. In general, this VAL_PREVIOUS 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 result of the conditional add/subtract operation. Additional details of the divide operation are described in section 4.4.1 of this document. One note about using conditional ALU operations; Neither the VAL_PREVIOUS nor the Q_BIT gets saved when handling an event. This implies, since event handlers can select conditions also, if an operation relies on the value of one of these latched conditions for a conditional ALU operation all events should be disabled in order to guarantee that the correct value of the

condition bit gets used. A description of each conditional ALU operation is as follows:

plus_else_minus The ALU function is A_PLUS_B when the condition is TRUE, A_MINUS_B when the condition is FALSE.

minus_else_plus The ALU function is A_MINUS_B when the condition is TRUE, A_PLUS_B when the condition is FALSE.

passA_else_passB
The ALU function is PASS_A when the condition is TRUE, PASS_B when the condition is FALSE.

passB_else_passA
The ALU function is PASS_B when the condition is TRUE, PASS_A when the condition is FALSE.

passA_else_incA The ALU function is PASS_A when the condition is TRUE, INC_A when the condition is FALSE.

incA_else_passA The ALU function is INC_A when the condition is TRUE, PASS_A when the condition is FALSE.

passA_else_decA The ALU function is PASS_A when the condition is TRUE, DEC_A when the condition is FALSE.

decA_else_passA The ALU function is DEC_A when the condition is TRUE, PASS_A when the condition is FALSE.

In addition to the explicit functions that microcode can choose from with the ALU micro-orders, additional ALU operations 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. 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.

Table 2-2: ALU Operations

Microword Field	Operation Name	Result
Arithmetic Operations		
0 0000	dec_A	F = A - 1
0 0001	A_plus_B	F = A + B
0 0010	inc_A_plus_B	F = A + B + 1
0 0011	left_1_A	F = A + A
0 0100	left_1_A_inc	F = A + A + 1
0 0101	dec_A_minus_B	F = A - B - 1
0 0110	A_minus_B	F = A - B
0 0111	inc_A	F = A + 1
0 1000	plus_else_minus	CONDITIONAL
0 1001	minus_else_plus	CONDITIONAL
0 1010	passA_else_passB	CONDITIONAL
0 1011	passB_else_passA	CONDITIONAL
0 1100	passA_else_incA	CONDITIONAL
0 1101	incA_else_passA	CONDITIONAL
0 1110	passA_else_decA	CONDITIONAL
0 1111	decA_else_passA	CONDITIONAL
Logical Operations		
1 0000	not_A	F = A~
1 0001	A_nand_B	F = (A and B)~
1 0010	not_A_or_B	F = A~ or B
1 0011	ones	F = -1 (2's comp)
1 0100	A_nor_B	F = (A or B)~
1 0101	not_B	F = B~
1 0110	A_xnor_B	F = (A xor B)~
1 0111	A_or_not_B	F = A or B~
1 1000	not_A_and_B	F = A~ and B
1 1001	A_xor_B	F = A xor B
1 1010	pass_B	F = B
1 1011	A_or_B	F = A or B
1 1100	pass_A	F = A
1 1101	A_and_not_B	F = A and B~
1 1110	A_and_B	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.

In addition to selecting one of the above data paths, the microcoder can specify two other sources of data to get stored into the register file. One of these other sources, the FIU BUS, is described in section 2.7.2 of this document. The final option when sourcing data to a C address is to take half of the 64 bit data word from the FIU BUS and half of it from the SHIFT MUX. This option is exercised by selecting the SPLIT_C_SOURCE encoding in the RANDOM field of the VAL microword. When this random is selected, if the C_SOURCE field of the VAL microword selects the SHIFT MUX as the source of C data, then the lower 32 bits of data that actually get passed to the register file are the lower 32 bits of the SHIFT MUX and the upper 32 bits of C data are the upper 32 bits of the FIU BUS. Similarly, if the SPLIT_C_SOURCE random is selected and the C_SOURCE field selects the FIU BUS, then the lower 32 bits of C data come from the lower 32 bits of the FIU BUS and the upper 32 bits of C data come from the upper 32 bits of the SHIFT MUX.

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_SOURCE and MULT_B_SOURCE, 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:

PRODUCT_LEFT_32 The product is left shifted 32 bits into <0..31> of the A_INPUT. All other bits of the A_INPUT are zero filled.

PRODUCT_LEFT_16 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 RANDOM instructions 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 seven LSB's, i.e. bits <57:63>, and all of the remaining bits are zero filled by the hardware. The range of values this number can assume is 0 .. 64.

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 REGISTER_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 the loop counter with the 10 LSB's of data from either the SHIFT MUX or the FIU BUS. This operation is accomplished by selecting the loop counter as the C ADDRESS field of the microword and then selecting the appropriate source of the C data with the C_SOURCE field.

The second way to change the value of the loop counter is to use the RANDOM micro-orders INC_LOOP_COUNTER or DEC_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 TRUE

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 in one place: the FIU board. 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 the 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 CSA_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 three restrictions are imposed: the default (write disable) C ADDRESS of the RF must be specified, the C_SOURCE field must select the SHIFT MUX, and the MUX_SOURCE field must select the WDR.

* 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 CSA_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 the existence of a data path between the FIU and some other board when in fact that path does not legally 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.

4. Data coming out of the multiplier, going over the FIU bus through the FIU and getting stored in the MDR, VAR, or TAR.

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.

5. Any time a reference is made to a Control Stack address space (by any board), the data cannot go onto the VAL bus, through the FIU and then in the same cycle get stored in the VAL board RF. This restriction is imposed because of hardware timing problems when the control address hits in the CSA.

In addition to the legal data path functions described above, the CONDITION_TO_FIU random allows the microcoder to merge certain conditions on the VAL board into the LSB of the FIU bus. When this random is selected, the currently selected testable condition is "stuffed" into the LSB of the FIU bus receiver, the other bits of the FIU bus are unaffected. Several restrictions are imposed when using this operation. First, only five conditions are legal choices to get merged with the FIU. They are : VAL_ALU_ZERO, VAL_ALU_NONZERO, VAL_ALU_A_LT_B, VAL_ALU_A_LE_B, and VAL_PREVIOUS. In addition, specific ALU functions must be chosen when merging these conditions. The correct ALU function for each of the legal conditions is shown in Table 2-3. The principal use of this merging 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.

Table 2-3: ALU functions for Condition Merging

Merge Condition	ALU Function
VAL_ALU_ZERO	A_XOR_B
VAL_ALU_NONZERO	A_XOR_B
VAL_ALU_A_LT_B	A_MINUS_B
VAL_ALU_A_LE_B	DEC_A_MINUS_B
VAL_PREVIOUS	any ALU function is allowed

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 a particular encoding.

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

ENCODING		NAME
00xxxx		a_gp
010000		a_tos
010001		a_tos_plus1
010010		spare1
010011	Default	a_loop_reg
010100		zero
010101		zero_detect
010110		product
010111		a_loop_ctr
011000		a_tos_minus8
011001		a_tos_minus7
011010		a_tos_minus6
011011		a_tos_minus5
011100		a_tos_minus4
011101		a_tos_minus3
011110		a_tos_minus2
011111		a_tos_minus1
1xxxxx		a_frame_reg

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

00xxxx		b_gp
010000		b_tos
010001		b_tos_plus1
010010		spare2
010011		b_loop_reg
010100		csa_bot_minus1
010101		csa_bot
010110	Default	csa_val_bus
010111		spare3
011000		b_tos_minus8
011001		b_tos_minus7
011010		b_tos_minus6
011011		b_tos_minus5
011100		b_tos_minus4
011101		b_tos_minus3
011110		b_tos_minus2
011111		b_tos_minus1
1xxxxx		b_frame_reg

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

00xxxx		c_gp
010000		c_tos
010001		c_tos_plus1
010010		spare4
010011		c_loop_reg
010100		c_csa_bot_minus1
010101		c_csa_bot
010110	Default	write_disable
010111		c_loop_ctr
011000		c_tos_minus8
011001		c_tos_minus7
011010		c_tos_minus6
011011		c_tos_minus5
011100		c_tos_minus4
011101		c_tos_minus3
011110		c_tos_minus2
011111		c_tos_minus1
1xxxxx		c_frame_reg

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

xxxxx		frame (11111 = Default Value)

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

0		fiu
1	Default	mux

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

00		alu_left
01	Default	alu
10		alu_right
11		wdr

ALU_FUNCTION (5 bits): specify the ALU function

00000		dec_A
00001		A_plus_B
00010		inc_A_plus_B
00011		left_1_A
00100		left_1_A_inc
00101		dec_A_minus_B
00110		A_minus_B
00111		inc_A
01000		plus_else_minus
01001		minus_else_plus
01010		passA_else_passB
01011		passB_else_passA
01100		passA_else_incA
01101		incA_else_passA
01110		passA_else_decA
01111		decA_else_passA
10000		not_A
10001		A_nand_B
10010		not_A_or_B
10011		ones
10100		A_nor_B
10101		not_B
10110		A_xnor_B
10111		A_or_not_B
11000		not_A_and_B
11001		A_xor_B
11010		pass_B
11011		A_or_B
11100		pass_A
11101		A_and_not_B
11110		A_and_B
11111	Default	zeros

MULT_A_SOURCE (2 bits): specify which group of 16 bits is used as the multiplier A_INPUT

00	Default	bits_0_15
01		bits_16_31
10		bits_32_47
11		bits_48_63

MULT_B_SOURCE (2 bits): specify which group of 16 bits is used as the multiplier B_INPUT

00	Default	bits_0_15
01		bits_16_31
10		bits_32_47
11		bits_48_63

RANDOM (4 bits): specify the described random operation

0000	Default	no_op
0001		inc_loop_counter
0010		dec_loop_counter
0011		condition_to_fiu
0100		split_C_source
0101		count_zeros
0110		spare5
0111		product_left_32
1000		product_left_16
1001		pass_A_high
1010		pass_B_high
1011		divide
1100		start_multiply
1101		spare6
1110		spare7
1111		spare8

TOTAL NUMBER OF BITS IN MICROWORD = 39

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).

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. The loop counter can be accessed through an A PORT address of the RF. The value of the loop counter is packed into the least significant 10 bits of the word (bits <54:63>) with the remaining bits of the word being zero filled by the hardware. The loop counter word can immediately be driven onto the FIU bus to get further packed into a "microstate" block (or whatever) that will be saved.

4.2. Conditions

The VAL board generates 18 testable conditions. On a given cycle, any one of these conditions can be selected, using the microcode control on the SEQUENCER board, to participate in conditional sequencing operations. Table 4-1 lists all of the VAL board conditions, the SEQUENCER board encoding to select each condition, and an indication as to whether the condition is early or late.

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.

Table 4-1: Value Board Conditions

Each condition is selected by specifying its encoding in the CONDITION_SELECT field of the SEQUENCER board microword. The selected condition also gets latched on the VAL board. Early conditions to the SEQUENCER are indicated by an "E", all other conditions are late.

Encoding		Name
0000000		val_alu_zero
0000001		val_alu_nonzero
0000010		val_alu_A_lt_B or val_alu_A_le_B (see text)
0000011		val_spare1
0000100	E	val_cntr_zero
0000101		val_spare2
0000110		val_spare3
0000111		val_spare4
0001000		val_alu_carry
0001001		val_alu_overflow
0001010		val_alu_lt_zero
0001011		val_alu_le_zero
0001100		val_sign_bits_equal
0001101		val_spare5
0001110		val_spare6
0001111	E	val_previous
0010000		alu_32_zero
0010001		alu_48_zero
0010010		alu_middle_zero
0010011	E	q_bit
0010100	E	m_bit
0010101		val_spare7
0010110	E	val_zero
0010111	E	val_one

VAL_ALU_ZERO 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.

VAL_ALU_NONZERO 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.

VAL_ALU_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 A_MINUS_B instruction on both halves of the ALU (using either of the "pass high" randoms is not allowed when selecting this condition). For additional information, see the description of the condition VAL_ALU_A_LE_B below.

VAL_ALU_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 DEC_A_MINUS_B instruction on both halves of the ALU (using either of the "pass high" randoms is not allowed when selecting this condition). Note, the hardware to generate this condition is the same as that to generate the VAL_ALU_A_LT_B condition. The distinction between these conditions, from a hardware perspective, comes from the different ALU operation that must be specified for each. For this reason, the two conditions are assigned only one encoding in the hardware, however the microcode may treat them as two distinct conditions for clarity.

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

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

VAL_ALU_LT_ZERO This condition is TRUE when the MSB (sign bit) of the ALU is a logical 1. Selecting this condition is equivalent to testing whether the 64 bit ALU output < 0.

VAL_ALU_LE_ZERO This condition is true whenever the 64 bit ALU output <= 0. This condition is logical "or" of the VAL_ALU_ZERO and the VAL_ALU_LT_ZERO conditions.

VAL_SIGN_BITS_EQUAL
This condition is TRUE whenever the MSB (sign bit) of the A_INPUT to the ALU is equal to the MSB of the B_INPUT to the ALU.

ALU_32_ZERO This condition is TRUE when the upper 32 bits (i.e. bits <0..31>) of the ALU output = 0.

ALU_48_ZERO This condition is TRUE when the upper 48 bits (i.e. bits <0..47>) of the ALU output = 0.

ALU_MIDDLE_ZERO 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.

VAL_ZERO When this condition is selected a logical zero is sent to the sequencer as the VAL board condition.

VAL_ONE When this condition is selected a logical one is sent to the sequencer as the VAL board condition.

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

M_BIT This condition is TRUE when the M_BIT is a logical 1. The M_BIT is a latched copy of the VAL_ALU_LT_ZERO condition from the previous cycle and is used by the hardware in the divide operation.

VAL_CNTR_ZERO 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.

VAL_PREVIOUS 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 to select as a condition the condition that was latched on the VAL board during the previous cycle.

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 functions fit into the operation 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 M_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 M_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 = VAL_ALU_CARRY      --- if no divide is in progress.
                               This is the initial value the Q
                               bit needs to begin a divide.

Q_BIT = (Q_BIT xor VAL_ALU_CARRY xor M_BIT)~    -- 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 PLUS_ELSE_MINUS 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 M_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 START_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_SOURCE and MULT_B_SOURCE 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 START_MULTIPLY instruction and choosing the desired 16 bit multiplier inputs with the MULT_A_SOURCE and MULT_B_SOURCE 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 muliplier 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: PRODUCT_LEFT_16 and PRODUCT_LEFT_32. 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 capabilities 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 (usually 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 CSA_VAL_BUS address. Whenever a memory write is made to a Control Stack address space, three restrictions are imposed during the second cycle after the START WRITE instruction is issued: First, the only legal C

ADDRESS that may be specified for the VAL board RF is the default (write disable) address. Second, the C_SOURCE encoding must select the shift mux as the source of C data. Third, the MUX_SOURCE encoding must select the WDR as the data path selected by the shift mux. These restrictions are imposed 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 CSA_VAL_BUS address. This restriction is imposed by hardware timing.

3. One note of caution when executing Control Stack POP_DOWN_TO 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 execute a POP_DOWN_TO instruction before executing a READ RDR and therefore move 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_DOWN_TO instructions in this type of situation. This restriction only applies to POP_DOWN_TO instructions, similar constraints do not apply to the PUSH and POP operations.

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 any 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.

5. Any time a reference is made to a Control Stack address space (by any board), the data cannot go onto the VAL bus, through the FIU and then in the same cycle get stored in the VAL board RF. This restriction is imposed because of hardware timing problems when the control address hits in the CSA.

6. Only certain testable conditions are allowed to be merged into the LSB of the FIU bus. Additionally, specific ALU functions are required to be selected during the cycle that the condition is being merged. Refer to Table 2-3 of this document for a summary of the allowable conditions and their respective ALU function.

OTHERS

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

1. The COUNT_ZEROS random cannot be used to count the number of leading zeros of the VAL ALU output when either of the following two conditions occur:

   a. The B_INPUT to the ALU is coming from the CSA_VAL_BUS address of the RF.

   b. The A_INPUT to the ALU is coming from the loop counter.

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

5.3. 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

Table of Contents

List of Tables