# Neptune System Overview

**By Russell L. Roan**

## 1.0 Introduction

The purpose of this document is to serve as a basic guide to debugging the Neptune (C3) system. While each of the major sub-systems will be touched upon, this document will be centered about the scalar processor as it relates to the rest of the C3 system.

There will be four major areas of interest:

- The nsp ( neptune scalar processor/crossbar memory interface
- The nsp/nvp ( neptune vector processor ) interface
- ASAP, interrupts, traps, and parallel processing
- Faults

Each of these sections will have a similar format. They will begin with a brief glossary of signal names, with a short description of how they interact. Following the glossary, a detailed description of how each instruction ( or instruction class ) that affects the interface is implemented.

## 2.0 The Memory System Interface

### 2.1 List of Components

The memory system consists of the following components:

- The crossbar ( xbar ) system. - The crossbar system is split into three boards, namely, the xrt, the xs0, and the xs1. However, from the perspective of the scalar processor, there are four blocks of the crossbar that accept/return memory requests. These blocks are called the xse ( even side send crossbar ) xso ( odd side send crossbar ), xre ( even side return crossbar ) and xro ( odd side return crossbar ). The send crossbar accepts read and write requests from processors for each side, and the return crossbar blocks return read data from each side of memory to the processors.
- m0-m7 - Up to 8 memory boards with an even & odd side on each.
- ncu - contains the communication registers
- xcl - some comm register status information is routed across this board.

The basic idea of the memory interface is that all memory and comm register operations from all processors are handled by a central crossbar. An illustration of this concept is de-

picted in Figure 1. Of course, any processor can still make requests to any memory board, it's just that they must go through the crossbar to get there.

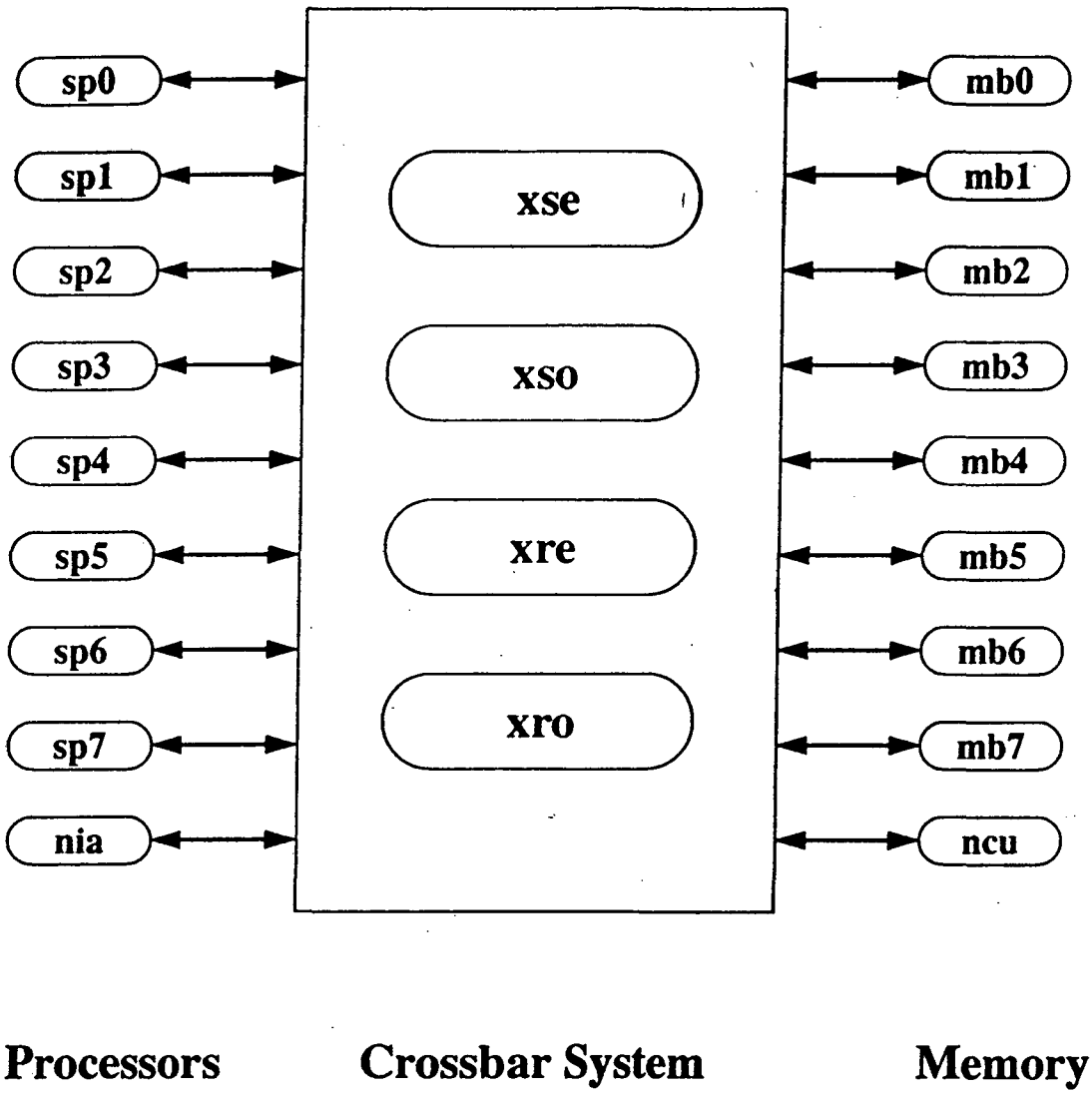



**FIGURE 1. Processor to Memory System Interface - System View**

## 2.2 Signal Glossary

For the purposes of this glossary, I will only present a detailed description of one set of signals that are replicated across identical interfaces. For example, I will only define

sp0_xse signals, while reminding the reader that the same set of signals exist for all eight processors and the nia. (sp1_xse.*, sp2_xse.*, etc. ).

### 2.2.1 Send crossbar (xse,xso) to processor (sp0-sp7,ia) signals

**xse_sp0.a_req_next** - Indicates that the a arbiter can take a request on the next clock.

**xse_sp0.a_req_pend** - Indicates that a request is pending in the a arbiter.

**xse_sp0.a_st_pend** - Indicates a write is pending in the a arbiter.

**xse_sp0.b_req_next** - Indicates that the b arbiter can take a request on the next clock.

**xse_sp0.b.req_pend** - Indicates a request is pending in the b arbiter.

**xse_sp0.b_st_pend** - Indicates a write is pending in the b arbiter.

Note that there are corresponding signals for xso_sp0,xse_sp1,xso_sp1, etc.

### 2.2.2 Processor ( sp0 - sp7, nia) to send crossbar (xse, xso) signals

The following is a list of signal definitions that are outputs from processor 0 and inputs to the even send crossbar. Note that corresponding signals exist for sp0_xso ( processor 0 to odd send crossbar ) and other processors to both even and odd send crossbars ( e.g., sp1_xse.*, sp1_xso.*, sp2_xse.*, sp2_xso.* ).

**sp0_xse.addr<28..3>** - Physical address for the even side longword. Note that the bits are numbered <28..3>. (i.e., there are 26 bits, with the MSB being bit 28 and the LSB being bit 3).

**sp0_xse.cycle<1..0>** - Indicates what operation is being done for this request, under the following scheme:

| Cycle | Operation |
|---|---|
| 0 | No Op |
| 1 | Read |
| 2 | Write |
| 3 | Test-and-Modify |

**sp0_xse.bd_sel<3..0>** - Memory board select. Indicates which memory board [0-7] is being requested. Note that if this value is >= 8, then the ncu is the board being requested.

**sp0_xse.ctl_par<4..0>** - Parity for all non-data bits going from processor to send crossbar. The scheme is as follows:

| Bit | Odd parity for: |
|---|---|
| 4 | sp0_xse.wr_zone |
| 3 | sp0_xse.addr<7..3> and sp0_xse.cycle |

| | |
|---|---|
| 2 | sp0_xse.addr<14..8> |
| 1 | sp0_xse.addr<21..15> |
| 0 | sp0_xse.addr<28..22> |

**sp0_xse.rdy** - Indicates that a memory request is being made when asserted ( ==1).

**sp0_xse.wr_data<31..0>** - Data word that is being written.

**sp0_xse.wr_par<3..0>** - Parity for wr_data. Note that MSB of the parity corresponds to least significant byte of data.

**sp0_xse.wr_zone<3..0>** - Indicates which bytes of a word of memory are being written.. Note that the order of the zone bits is reversed with respect to the corresponding data bytes. ( e.g., the MS bit of zone refers to the LS byte of data. The scheme is reproduced below:

| Zone bit | Corresponding data bits |
|---|---|
| <3> | <7..0> |
| <2> | <15..8> |
| <1> | <23..16> |
| <0> | <31..24> |

### 2.2.3 Receive crossbar (xro,xre) to processor (sp0-sp7,nia) signals

This is a list of signals that return memory requests from the crossbar to the scalar processors. Again, I will explicitly list xre_sp0 signals, and the reader can extend these definitions to include the other processors/xbars.

**xre_sp0.rd_data<31..0>** - Data word read from even side of memory.

**xre_sp0.rd_rdy** - Indicates when there is a data word ready from the even side of memory.

**xre.sp0.rd_par<3..0>** - Parity for xre_sp0.rd_data. MSB refers to LS byte of data.

### 2.2.4 Memory to crossbar signals

This section will describe the signals that connect a memory board ( mb[0-7],cu), and the send and receive crossbars ( xs{o,e}, xr{o,e} ).

The following is a list of signals from the send crossbar to memory boards. Note that an analogous set of signals goes to the ncu; thus, communication register reads have the same interface signals as a memory read. The signals

**xse_mb0.addr<28:3>**

**xse_mb0.ctl_par<5>**

**xse_mb0.cycle<2>**

**xse_mb0.rdy**

**xse_mb0.wr_data<32>**

**xse_mb0.wr_par<4>**

**xse_mb0.wr_zone<4>**

are analogous to the processor_send.crossbar signals discussed previously.

**xso_mb0.ref_req** - Indicates a refresh of DRAM is occurring this cycle.

**mb0_xso.bank_done<16>** - Each bit of this signal indicates when a bank of memory on the board is done with its request. For example, when bank<0> of mb0 has completed its request, then **mb0_xso.bank_done<0>** will be asserted.

Please note that the preceding two signals only exist for memory boards; there is no ncu copy of these signals.

Finally, there are signals that connect the memory boards/ncu with the return crossbar. Again, note that similar signals connect the cu to the return crossbars. Each of the following signals perform the same function as the similarly-named signals described previously that connect the return crossbars to the scalar processors.

**mb0_xre.rd_data<32>** - Read word that is given to the return crossbar from memory.

**mb0_xre.rd_par<4>** - Read parity from memory.

**mb0_xre.rd_rdy** - Indicates data is ready for the return crossbar.

### 2.2.5 Signals involving the xcl board

Finally, some information for communication register reads is routed across the xcl to the scalar processors. These signals are:

**cu_xc.status** - Indicates the verity of the last communications register read. For example, a 1 here on a receive command to the comm registers would indicate success (i.e., the lock bit was 1 ).

**cu_xc.status_en** - A 1 here indicates that cu_xc.status is valid on this cycle.

**cu_xc.status_psel<4>** - Indicates the processor for which the comm register status information being transmitted this cycle is intended. For example, if **cu_xc.status_psel** is 0x4, then this information is intended for sp4.

The following 2 signals from the xcl exist for all 8 processors.

**xc_sp0.status** - Indicates verity of the comm register operation last requested.

**xc_sp0.status_en** - A 1 here indicates that the .status is valid this cycle.

## 2.3 Memory Operations

Here, I discuss the series of events that occur whenever a request is made to memory. All memory requests look the same to the crossbar, so I won't delve into the various instructions that could produce memory requests; rather, I will focus upon how the various memory/ncu requests are implemented once they get to the crossbar stage.

### 2.3.1 Physical Addresses

When discussing the crossbar interface, I will refer to "physical address" frequently. This term refers to the full 32 bit physical address to which a logical address is mapped.

#### 2.3.1.1 Converting a physical address to crossbar interface

After a 32-bit logical address is mapped to a 32 bit physical address, some further bit stripping and manipulation is done before this address is presented to the send crossbars.

Since memory can only be accessed a word at a time (a word of data, rather than a byte ), then the lowest two bits ( <1> and <0> ) of the calculated physical address are not necessary for the memory board. These are, however, used to calculate the **sp0_xse.wr_zone** value. ( see the section on memory writes to see how this is done.)

Also, bit <2> of the calculated physical address is used to determine whether the even or odd side of memory is being accessed.( i.e., whether the request goes to the even send crossbar ( sp0_xse. signals ) or the odd send crossbar ( sp0_xso.signals ). )

Finally, bits <31:29> are either used to calculate the **sp0_xse.bd_sel,** or swapped with some other address bits which form the board select. ( Again, the details of this will be explained shortly. )

It is for this reason that only bits <28...3> make up the physical address that is sent to the appropriate crossbar.

Frequently, a single logical request will generate two separate physical requests, one to each side of memory ( even and odd ). For example, suppose we want to read the word at physical 0x3002. This would entail reading bytes 0x3002, 0x3003, 0x3004, and 0x3005. Note that two of these bytes ( 0x3002, 0x3003 ) are part of an even word ( bit <2> == 0 ), and the other two bytes are part of an odd word ( bit <2> == 1 ). Thus, for this request, we would have one request go to the even send crossbar ( xse, for the word at 0x3000), and one would go to the odd send crossbar ( xso, for the word at 0x3004).

With no interleave, bits <31..29> of the physical address form the **sp0_xse.bd_sel.** However, if memory interleave is being done by the processor, then some bit swapping is done. The method of interleaving depends upon the number of memory boards in the system. Table 1 shows the interleave method used for all of the possible numbers of memory boards.

TABLE 1. Memory Interleave Scheme - Bits of Physical address for bd_sel and addr

| # of Boards | bd_sel | addr |
|---|---|---|
| 1 | <31..29> | <28..3> |
| 2,3 | <31..30>,<7> | <28..8>,<29>,<6..3> |
| 4-7 | <31>,<8..7> | <28..9>,<30:29>,<6..3> |
| 8 | <9..7> | <28..10>,<31..29>,<6..3> |

The last of the crossbar interface signals that we should mention are the **wr_zone** signals. These are used to indicate which bytes of a memory word will be written during a memory write . Recall from the signal glossary that the least significant bit of the **wr_zone** signal refers to the most significant bit of data ( i.e., the bits are in reverse order with respect to the bytes to which they refer). For example, if we wanted to write a halfword at physical address 0x3000, we would issue a write cycle request for the word at 0x3000 ( note that this is the even side, as bit <2> is clear ) and set the wr_zone signal to 0x3. This will cause the bytes at 0x3000 and 0x3001 to be written, but would not change the bytes at 0x3002 and 0x3003.

### 2.3.1.2 Example

Suppose that processor four ( sp4 ) writes the word 0x11223344 at physical address 0x000035C6, with a memory system with four boards. So, our interface signals would be:

**sp4_xse.addr**= 0x0000689
**sp4_xse.cycle** = 0x2
**sp4_xse.bd_sel** = 0x3
**sp4_xse.ctl_par** = 0x13
**sp4_xse.rdy** = 1
**sp4_xse.wr_data** = 0x3344FFFF
**sp4_xse.wr_par** = 0xF
**sp4_xse.wr_zone** = 0x3

**sp4_xso.addr** = 0x0000688
**sp4_xso.cycle** = 0x2
**sp4_xso.bd_sel** = 0x3
**sp4_xso.ctl_par** = 0x1B
**sp4_xso.rdy** = 1
**sp4_xso.wr_data** = 0xFFFF1122
**sp4_xso.wr_par** = 0xF
**sp4_xso.wr_zone** = 0xC

Note that the even and odd addresses are different. This is because our address begins in the middle of the memory word beginning at 0x35C4, and ends in the middle of the memory word beginning at 0x35C8. Also, the "F"'s on the wr_data are in positions that are indeterminate since the wr_zone has that part of the word masked out. What actually would be there is unimportant.

### 2.3.2 Memory reads

This section describes the series of handshakes that occur whenever memory is read.

#### 2.3.2.1 Processor to send crossbar interface

First, the crossbar must indicate to the processor that is accepting memory requests. This is indicated to the processor by asserting both copies of the "request next" signals, namely, **xse_sp0.a_req_next** and **xse_sp0.b_req_next**. Each of these must be asserted in order to get a memory request done.

Next, the processor must assert its ready line ( **xse_sp0.rdy** ). In order for the request to proceed, the two request next signals must have been asserted on the previous clock. If the processor is ready, but one of the crossbar "request next" signals was 0 on the previous clock, then the "rdy" signal from the processor will remain 1 until one clock after both req_next signals from the crossbar are 1.

In short, if we have both req_next signals as 1 at time X, and a rdy asserted by a processor at time ( X + 1 ), then time ( X + 1 ) is when the request is being made, and this is when all of the processor interface information is valid.

#### 2.3.2.2 Send crossbar to memory interface

At some time following the request to the send crossbar by the processor, the send processor will make the request to the appropriate memory board ( determined by the processor to send_crossbar **bd_sel** value). The amount of time before the send crossbar makes this request depends upon how busy it is with other requests.

The request to memory board is made by asserting ( in the case of even send crossbar to memory board 0 ) the **xse_mb0.rdy** signal. At this time all of the send_crossbar to memory board signals have valid data.

A quick note about crossbar to memory board address field. The least significant nibble of this field will indicate the bank of memory being addressed. The the rest of the address ( with the LS nibble chopped off ) indicates the address within the bank of the word that is being accessed. So, for example, suppose we had the following signals during a read of memory board 4 on the odd side.

**xso_mb4.rdy** = 1

**xso_mb4.addr** = 0x0000227

Then we could examine the word in question by looking at the memory board 4, odd side, bank 7, bank address 0x000022.

### 2.3.2.3 Memory board to return crossbar

The crossbar to memory board reads are fixed-length in time. This is adjustable for future use. However, at this time, all memory reads occur in 13 cycles. So, exactly 13 cycles after the send crossbar asserts **rdy** to a memory board, then the memory board will assert the **rd_rdy** signal to the return crossbar on the appropriate side. At this time, the **rd_rdy** and **rd_data** signals will have their appropriate values.

### 2.3.2.4 Return Crossbar to processor

After the read is presented to the return crossbar, the return crossbar asserts its **rd_rdy**, **rd_data**, and **rd_par** signals to the processor that made the request, and the cycle is complete.

### 2.3.2.5 Example - Processor 6 reads word at physical 0x5560, 8-bd interleave

Suppose processor 6 wants to read a word at physical address 0x5560, which contains the data word 0x11223344. Assume that we have 8 memory boards in the system. Note that this is contained within an even word, so we only have to make a request to the even crossbar.

Suppose that the processor indicates that it is ready to make a request by asserting **sp6_xse.rdy** at time X. If both **xse_sp6.a_req_next** and **xse_sp6.b_req_next** were asserted at time ( X - 1 ), then the request proceeds normally. Note that if either of the **req_next** signals was not 1, then the processor would have to wait ( keeping **rdy** at 1 all the while ) until one clock after both **req_next** signals were 1.

In any case, on the cycle of the request, we get the following values:

**sp6_xse.addr**= 0x0000A8C

**sp6_xse.cycle** = 0x1

**sp6_xse.bd_sel** = 0x2

**sp6_xse.ctl_par** = 0x1B

**sp6_xse.rdy** = 1

**sp6_xse.wr_data** = 0x00000000

**sp6_xse.wr_par** = 0xF

**sp6_xse.wr_zone** = 0xF

Note that the write signals are included for completeness, but are not really relevant for our read example.

After the send crossbar receives this request, it passes it on to memory board 2 with the following values.

**xse_mb2.addr**= 0x0000A8C

**xse_mb2.cycle** = 0x1

**xse_mb2.ctl_par** = 0x1B

**xse_mb2.rdy** = 1

**xse_mb2.wr_data** = 0x00000000

**xse_mb2.wr_par** = 0xF

**xse_mb2.wr_zone** = 0xF

Note that this address indicates that we are reading board 2, even side, bank 0xC, address 0x0000A8.

Exactly 13 clocks after the rdy goes to the memory boards, we get the following signals to the return crossbar:

**mb2_xre.rd_rdy** = 1

**mb2_xre.rd_data** = 0x11223344

**mb2_xre.rd_par** = 0xF

Finally, the corresponding signals are asserted to the processor that made the request.

**xre_sp6.rd_rdy** = 1

**xre_sp6.rd_data** = 0x11223344

**xre_sp6.rd_par** = 0xF

This completes the read cycle from the scalar processor.

### 2.3.3 Memory Writes

#### 2.3.3.1 Handshakes

Memory writes begin the same way as reads, except that the **wr_par**, **wr_data**, and **wr_zone** signals are meaningful.

Again, a memory write must have both **req_next** signals from the crossbar asserted one clock before the request. On the cycle of the request, rdy is 1 and all of the processor to send crossbar signals are valid.

Following the processor to send crossbar request, the send crossbar passes the request information on to the appropriate memory board in the same manner as the read. Note that the cycle is complete after the request goes to the appropriate memory board as no information is returned to the processor on writes.

#### 2.3.3.2 Example - write halfword 0x5a5b at physical 0x3456 from sp7, 8 mb's

This halfword resides in the lower 2 bytes of the word at 0x3454, so we will be addressing the odd side of memory with a **zone** of 0xC. At the time of the request, we have

**sp7_xso.addr**= 0x000068A

**sp7_xso.cycle** = 0x2

**sp7_xso.bd_sel** = 0x0

**sp7_xso.ctl_par** = 0x1B

**sp7_xso.rdy** = 1

**sp7_xso.wr_data** = 0xFFFF5A5B

**sp7_xso.wr_par** = 0x7

**sp7_xso.wr_zone** = 0xC

Similarly, the **xso_mb0** signals are set by the send crossbar.

### 2.3.4 Test and Modifies

#### 2.3.4.1 The tam interface

The test and modify operation is essentially a read followed by a write operation. The original data word is read and returned to the processor, after which the data word in memory is replaced with the word sent by the processor. The **cycle** code for a tam is 0x3.

Since we are replacing the contents of the memory word regardless of its value, the interface to the send crossbar is exactly as for a write. Similarly, the return crossbar handshakes are exactly as for a read as we expect the original data word to be returned to the processor.

#### 2.3.4.2 Example - do a TAC of the byte at 0x3001 by processor 0

Let's assume that we have four memory boards and the word at 0x3000 is 0x00FF0000. When the request is made, we have:

**sp0_xse.addr**= 0x0000600

**sp0_xse.cycle** = 0x3

**sp0_xse.bd_sel** = 0x0

**sp0_xse.ctl_par** = 0x0F

**sp0_xse.rdy** = 1

**sp0_xse.wr_data** = 0x00000000

**sp0_xse.wr_par** = 0xF

**sp0_xse.wr_zone** = 0x2

When this gets to the memory boards, we get:

**xse_mb2.addr**= 0x0000600
**xse_mb2.cycle** = 0x3
**xse_mb2.ctl_par** = 0x0F
**xse_mb2.rdy** = 1
**xse_mb2.wr_data** = 0x00000000
**xse_mb2.wr_par** = 0xF
**xse_mb2.wr_zone** = 0x2

Finally, after exactly 13 clocks, we get:

**mb2_xre.rd_rdy** = **1**
**mb2_xre.rd_data** = 0x00FF0000
**mb2_xre.rd_par** = 0xF

Finally, the corresponding signals are asserted to the processor that made the request.

**xre_sp6.rd_rdy** = 1
**xre_sp6.rd_data** = 0x00FF0000
**xre_sp6.rd_par** = 0xF

Note that at the completion of this instruction the data word at 0x3000 is 0x00000000.

## 2.4 NCU/Communication register operations

NCU operations are identical to memory operations from the perspective of the crossbar. The differences are only from the perspective of the processor. The crossbar treats the ncu ( which contains the communication registers ) as simply another memory board. In addition, some status information is routed across the XCL, which returns the NCU's status bits to the processor on operations that require them. Finally, all of the control information is passed to the NCU on the odd ( xso ) side. The even side is only used for the upper word of data ( and parity ) for longword operations.

The operations on the NCU affect one of two data structures, the communication registers and the X-space registers.

### 2.4.1 Communication Registers

The most familiar type of NCU instruction is the communication register operation. These should be discussed in the Architecture Specification; however, in the interests of review, and to describe the implementation on the NCU, I'll present a brief description here.

The communication registers are simply longword registers that have an associated lock bit. Some comm register operations are specified as word operations; in such case, only the lower word of the longword is affected.

The comm registers are grouped into sets of 128 longwords. Each set of 128 longwords is associated with a CIR ( Communications Index Register ) value. Briefly, the CIR is a 5 bit value that all CPUs of a particular process share in common. Thus, we have128 ( registers) X 32 ( CIR values ) =4096 total comm registers ( 0xFFF hex ).

#### 2.4.1.1 Comm register access - Normal Operation

For normal operation, CPU access to comm registers is restricted to the set of comm registers associated with that CPU's CIR. To ensure this restriction, comm registers are accessed via a 16 bit logical address that is mapped according to the CPU's CIR value. This range of addresses is restricted ( for normal operation ) to include only128 possible values ( to correspond to the128 registers within the CPU's CIR value ). In addition, some of the 128 registers associated with a CIR value are readable only if the CPU is operating in ring 0. Table 2 gives the range of values allowed for "normal" comm register access, along with the further restrictions imposed.

**TABLE 2. Comm register logical addresses - Normal operation**

| Range | Description | Access |
|---|---|---|
| 0x0000-0x001F | Ring 0 HW | Cpu operating in Ring 0 |
| 0x4000-0x401F | Ring 0 SW | Cpu operating in Ring 0 |
| 0x8000-0x802F | Ring 4 SW | No restrictions |

The comm registers are accessed by constructing a 12-bit physical address from the CIR of the processor and the logical address. ( Recall that there are 4096 total registers, so that 12 bits of physical address are necessary to distinguish them.) The mapping scheme is presented in Table 3. Note that in this table, "CCCCC" refers to the 5 bit CIR, and "AAAAA" or "AAAAAA" refers to the lowest five or six bits of the logical address ( five for Ring 0, 6 for Ring 4 ).

**TABLE 3. Comm register logical to physical address mapping - Normal operation**

| Range | Logical Address | Physical address |
|---|---|---|
| 0x0000-0x001F | 0000 0000 000A AAAA | CCCC C00A AAAA |
| 0x4000-0x401F | 0100 0000 000A AAAA | CCCC C01A AAAA |
| 0x8000-0x802F | 1000 0000 00AA AAAA | CCCC C1AA AAAA |

For example, if we were in CIR 5, and we wanted to read logical address 0x4010, the physical address would be 00101 (cir) 01 (ring 0 sw indicator)10000 (address offset bits, shown as AAAAA in chart ), or 0x2B0.

### 2.4.1.2 Comm register access - "backdoor" operation

In addition to the "normal" operation, which restricts CPU access to the 128 comm registers associated with its CIR value, there is a method by which CPUs may access any of the 4096 comm registers, regardless of the value of the CPU's CIR. This method is known as "backdoor" or physical addressing. It is called "physical" addressing because it the actual physical address of the comm register is embedded in the logical address, so that the normal CIR mapping is not used. It is important to note that the CPU may only use "backdoor" addressing if it is in Ring 0. Table 4 shows the range of values and the way the logical address is stripped of some bits to form the physical address.

**TABLE 4. Comm register logical to physical mapping - "backdoor" mode**

| Range | Logical Address | Physical Address |
|---|---|---|
| 0x3000-3FFF | 0011 AAAA AAAA AAAA | AAAA AAAA AAAA |

As can be seen from the Table, backdoor addressing is in use when the leading nibble of the 16 bit logical address is 0x3. In these cases, the trailing 12 bits of logical address form the physical address of the comm register. For example, a read of 0x3FED would produce a read of physical address 0xFED.

Note that to read this same address in normal mode, the CPU would have to be in CIR 31, and read address 0x802D.

### 2.4.1.3 Comm Register Physical Implementation

The comm registers are implemented on the NCU in rams. SInce the standard ram used in the neptune project is 2K deep, then two banks of them are needed to implement the 4K longwords of comm registers and lock bits.

This is done by using bit <11> of the physical address of ram to select between the two banks of ram. So, all physical address from 0x000 through 0x7FF may be read by addressing the "msb0" bank with the lower 11 bits of address. Similarly, addresses 0x800 through 0xFFF would use the lower 11 bits to address the "msb1" bank of ram.

The lock bit rams are split up in a similar fashion.

### 2.4.2 The X-space registers

The X-space registers are special control registers on the NCU that are accessible only by the microsequencer on a processor and the NIA. Thus, these registers can not be accessed by the user.

The interface for the control registers is the same as for comm register operations. The NCU distinguishes between comm register operations and X-space operations by the command that is transmitted as part of the **addr** field from the crossbar ( see below ).

In short, each X-space register is matched to a fixed physical address, so that when an X-space command is requested, then the physical address given to the NCU during the request determines which X-space register is being requested. In reality, the X-space registers are simply hardware latches residing within the nadr and ndat gate arrays on the NCU.

For a complete list of X-space registers, consult the NCU functional specification.

### 2.4.3 Physical addresses & Crossbar Interface signals

The **addr** value that is presented to the crossbar has the following fields:

**TABLE 5. Address fields for NCU operations**

| Bits of addr | Meaning |
|---|---|
| <28..26> | op |
| <25> | 0=CPU, 1 = NIA |
| <24..22> | processor i.d. |
| <21..17> | thread i.d. |
| <16> | size ( 1 =lw, 0=w ) |
| <15> | 0 |
| <14..3> | physical address of comm register |

In order to determine what operation is being requested, the cycle signal must be appended to the op bits <28..26> of the address. Table 3 shows a chart of the possible operations. Note that the "Code" in the table is actually bits <28..26> of **addr** concatenated with **cycle.**

**TABLE 6. NCU Operations**

| Operation | Code | Meaning |
|---|---|---|
| LCK | 0x00 | Lock comm register |
| ULK | 0x04 | Unlock comm register |
| ENI | 0x08 | Enable interrupts |
| DSI | 0x0C | Disable interrupts |
| TST | 0x10 | Test comm register |
| RCV | 0x01 | Receive comm register |
| RCV_X | 0x05 | Receive X-space register |
| GET | 0X11 | Get comm register |
| GET_X | 0x15 | Get X-space register |

| | | |
|---|---|---|
| RDCMR | 0x19 | Read communication register set |
| SND | 0x02 | Send to comm register |
| SND_X | 0x06 | Send to X-space register |
| PUT_S | 0x0E | Put, with returned status |
| PUT | 0x12 | Put to comm register |
| PUT_X | 0x16 | Put to X-space register |
| WRCMR | 0x1A | Write communication register set |
| INC | 0x03 | Increment comm register |

The **wr_zone** bits re meaningless for ncu operations, as all register writes must be either word or longword, and this is determined by size field of **addr**. Furthermore, even and odd sides are not meaningful for the ncu. It is for this reason that all control information for ncu operations is contained on the odd ( xso ) side of the crossbar, and even control information can be ignored. The only data on the even side that is important is the **wr_data** signal, which contains the upper half of longword write data. The odd side, then, contains the data for word writes and forms the lower word for longword writes.

The rest of the crossbar signals have the same meaning as for memory operations.

### 2.4.4 Summary of Operations

Discussions of the various and sundry comm register operations are grouped based upon instruction type.

#### 2.4.4.1 Single Bit Operations: LCK, ULK, DSI, ENI, TST

There are a number of NCU operations that do not affect any of the register operations; rather, they simply affect a register's lock bits or a single system-wide status bit. These are what I call the "single bit operations", and consist of the LCK, ULK, ENI, and DSI.. All of these operations return status bits, and the interface is identical for all of them. A brief synopsis of each instruction follows:

LCK - Set lock bit of comm register. If register was already locked, return status of 0. Otherwise, return status of 1.

ULK- Clear lock bit of comm register. If register was already unlocked, return status of 0. Otherwise, return status of 1.

TST - Returns the lock bit of a comm register.

DSI - Disable interrupts by clearing ION bit. This bit resides in the nadr gate array as r.l2_ion. Also, it may be read at the pinout as "l2_ion." Returns a status bit equal to the value of the ION bit before the operation.

ENI - Enable interrupts by setting the ION bit. Returns status equal to the value of the ION bit before the operation.

The first thing required is to have **spx_xso.rdy** asserted one clock after both crossbar to processor **req_next** signals are 1. At this time, the **addr**, **bd_sel**, **ctl_par**, and **cycle** should

have the correct information for the request. The **wr_zone**, **wr_data**, and **wr_par** signals are unused for these ops. Note that **bd_sel** will be 0x8 for the NCU, and **cycle** will be 0x0 indicating that no register will be affected by this operation ( i.e., only status bits.)

After a short amount of time, the request should have filtered through the crossbar logic to appear on the xso_cu.* signals, namely **addr**, **ctl_par**, and **cycle**.

Exactly three clocks after this request is presented to the NCU, the NCU will return the status bits of the operation to the XCL by setting

**cu_xc.status_en** = 1 ( indicating status is being returned this cycle )

**cu_xc.status** = ( whatever is appropriate for this operation )

**cu_xc.psel** = ( the CPU that made this request ( e.g., 0x0 if from CPU 0 ) )

Exactly two clocks after this handshake, the status information is passed on to the appropriate processor ( as determined by the psel field given to the XCL ). At this time

**xc_sp?.status_en** = 1

**xc_sp?.status** = whatever was given to the XCL from the NCU.

This completes the cycle for operations that return status bits only.

**Example - LCK address 0x040 from sp1, cir 0, tid 1, register already locked**

First we make the request to the odd crossbar ( xso ). Assuming we had both **xso_sp1.req_next** signals asserted on the previous clock, we would have

**sp1_xso.addr** = 0x0084040

**sp1_xso.bd_sel** = 0x8 ( indicates NCU )

**sp1_xso.ctl_par** = 0x18

**sp1_xso.cycle** = 0x0

**sp1_xso.rdy** = 1

After some time, the send crossbar passes this information on to the NCU.

**xso_cu.addr** = 0x0084040

**xso_cu.ctl_par** = 0x18

**xso_cu.cycle** = 0x0

**xso_cu.rdy** = 1

On the third clock following this cycle, the status information is given to the XCL.

**cu_xc.status_en** = 1

**cu_xc.status** = 0 ( indicates register was already locked )

**cu_xc.status_psel** = 0x1 ( indicates processor 1 made this request )

Finally, on the second clock following the **cu_xc.status_en** assertion, the XCL passes the status information on to the appropriate processor.

**xc_sp1.cu_status_en** = 1

**xc_sp1.cu_status** = 0

This completes the LCK operation.

**Example - DSI interrupts, which are currently enabled.**

For this example, let's assume we are sp7, we are in CIR 15, thread 10.

We begin with the standard send crossbar request:

**sp7_xso.addr** = 0x1BA8000

**sp7_xso.bd_sel** = 0x8 ( indicates NCU )

**sp7_xso.ctl_par** = 0x1E ( assumes **wr_zone** is 0x0 )

**sp7_xso.cycle** = 0x0

**sp7_xso.rdy** = 1

In practice, bits <16..3> of the **addr** are irrelevant for the DSI and ENI operations, so they are filled with zeroes. Similarly, the wr_zone and wr_data are zeroes as well. After filtering through the crossbar, we have:

**xso_cu.addr** = 0x1BA8000

**xso_cu.ctl_par** = 0x1E

**xso_cu.cycle** = 0x0

**xso_cu.rdy** = 1

Again, on the third following clock, the status is returned to the XCL from the NCU:

**cu_xc.status_en** =1

**cu_xc.status** = 1 ( Indicates interrupts were enabled before instruction. ( ION was 1 ) )

**cu_xc.status_psel** = 0x7 ( indicates processor 7 made this request )

Finally, on the second following clock:

**xc_sp7.cu_status_en** = 1

**xc_sp7.cu_status** = 1

### 2.4.4.2 Read register operations - no status - GET, GET_X

There are several instructions involving the NCU that are simple reads of registers without regard to that register's lock bit. These operations include:

GET - read contents of comm register. May be word or longword in size.

GET_X - read contents of X-space register. This will always be for longwords.

These two operations are implemented in exactly the same fashion. Recall that the NCU knows that an X-space register is being requested ( rather than a comm register ) by the op field of the **addr.**

We begin, as always, with the **spx_xso.rdy** signal being asserted one clock after both **req_next** signals from the xso are 1. After a short amount of time, the xso_cu signals should have this information ready for the ncu with the assertion of **xso_cu.rdy**. On the 3rd following clock, **cu_xro.rd_rdy** is asserted, indicating that read information is ready. Note that if a longword is being read, **cu_xre.rd_rdy** is asserted as well. Finally, at some time later, this information is passed on to the appropriate processor when **xro_sp0.rd_rdy** is asserted.

**Example - get.l at 0x8000 from processor 0, cir 0, tid 0, data=0xfedca9876543210**

When the processor makes the request, we have the following values at the crossbar interface:

**sp0_xso.addr** = 0x2002040

**sp0_xso.bd_sel** = 0x8 ( indicates NCU )

**sp0_xso.ctl_par** = 0x10

**sp0_xso.cycle** = 0x1

**sp0_xso.rdy** = 1

After some time, the send crossbar passes this information on to the NCU.

**xso_cu.addr** = 0x2002040

**xso_cu.ctl_par** = 0x10

**xso_cu.cycle** = 0x1

**xso_cu.rdy** = 1

Exactly three clocks later, the ncu answers with:

**cu_xro.rd_data** = 0x76543210

**cu_xro.rd_par** = 0x0

**cu_xro.rd_rdy** = 1

**cu_xre.rd_data** = 0xFEDCBA98

**cu_xre.rd_par** = 0x0

**cu_xre.rd_rdy** = 1

Finally, the crossbar returns its data to the processor:

**xro_sp0.rd_data** = 0x76543210

**xro_sp0.rd_par** = 0x0

**xro_sp0.rd_rdy** = 1

**xre_sp0.rd_data** = 0xFEDCBA98

**xre_sp0.rd_par** = 0x0

**xre_sp0.rd_rdy** = 1

### 2.4.4.3 Reads with status - RCV, RCV_X

Some operations expect both data and status back from the ncu. These ops are

RCV - Receive data status from comm register.

RCV_X - Receive data and status from X-space register.

The request for these operations begin with the assertion of the processor to crossbar **rdy** signal. This is followed shortly thereafter by the request made to the ncu by the send crossbar. On the third following clock, the ncu asserts its **status_en** and **rd_rdy** signals indicating that the operation is complete ( from the perspective of the ncu ). On the second following clock, the XCL passes on the status information to the appropriate processor by asserting its **cu_status_en** signal. Finally, the return crossbar completes the cycle with the assertion of its **rd_rdy** signal to the appropriate processor.

**Example - sp0 to rcv locked register 0x8020, tid = 15, cir=10, data= f..0**

At the time of the request,

**sp0_xso.addr** = 0x003E560

**sp0_xso.bd_sel** = 0x8 ( indicates NCU )

**sp0_xso.ctl_par** = 0x15

**sp0_xso.cycle** = 0x1

**sp0_xso.rdy** = 1

After some time, the send crossbar passes this information on to the NCU.

**xso_cu.addr** = 0x003E560

**xso_cu.ctl_par** = 0x15

**xso_cu.cycle** = 0x1

**xso_cu.rdy** = 1

Exactly three clocks later, the ncu answers with:

**cu_xro.rd_data** = 0x76543210

**cu_xro.rd_par** = 0x0

**cu_xro.rd_rdy** = 1

**cu_xre.rd_data** = 0xFEDCBA98

**cu_xre.rd_par** = 0x0

**cu_xre.rd_rdy** = 1

**cu_xc.status_en** = 1

**cu_xc.status** = 1

**cu_xc.status_psel** = 0x0

Two clocks after this, the XCL passes this information on to the appropriate processor:

**xc_sp0.cu_status_en** = 1

**xc_sp0.cu_status** = 1

Finally, the crossbar returns its data to the processor:

**xro_sp0.rd_data** = 0x76543210

**xro_sp0.rd_par** = 0x0

**xro_sp0.rd_rdy** = 1

**xre_sp0.rd_data** = 0xFEDCBA98

**xre_sp0.rd_par** = 0x0

**xre_sp0.rd_rdy** = 1

#### 2.4.4.4 NCU writes without status - PUT, PUT_X

A number of operations on the NCU involve writing registers without concern for the status bits. These operations include:

PUT - Write a communications register

PUT_X - Write an X-space register

The interface for PUTs is simple. It begins, as always, with the assertion of the processor to send crossbar **rdy** signal. Following this, the send crossbar passes this information on to the ncu. No information is returned to the processor.

**Example - Put word 0x00112233 to 0x8001, cir 8, tid 7, sp0**

Request:

**sp0_xso.addr** = 0x201C441

**sp0_xso.bd_sel** = 0x8 ( indicates NCU )

**sp0_xso.ctl_par** = 0x0C

**sp0_xso.cycle** = 0x2

**sp0_xso.rdy** = 1

**sp0_xso.wr_data** = 0x00112233

**sp0_xso.wr_par** = 0xF

**sp0_xso.wr_zone** = 1

After some time, the send crossbar passes this information on to the NCU.

**xso_cu.addr** = 0x201C441

**xso_cu.ctl_par** = 0x15

**xso_cu.cycle** = 0x2

**xso_cu.rdy** = 1

**xso_cu.wr_data** = 0x00112233

**xso_cu.wr_par** = 0xF

This completes the write without status cycle.

### 2.4.4.5 Writes with status - PUT_S, SND, SND_X

Some operations return status bits as part of the write operation. These include:

SND - if lockbit of comm register is 0, then write it, set its lock bit, and return status of 1. Otherwise don't write the register and return status of 0.

SND_X - send to X-space register.

PUT_S - Write to a comm register, return the value of its status bit.

The interface is the same for as a put, except that status bits are returned as in a lock instruction.

**Example - Snd lw 0xFFFFFFFE FFFF0001 to 0x7 unlocked, tid 0, cir 5**

Request:

**sp0_xso.addr** = 0x0002407

**sp0_xso.bd_sel** = 0x8 ( indicates NCU )

**sp0_xso.ctl_par** = 0x09

**sp0_xso.cycle** = 0x2

**sp0_xso.rdy** = 1

**sp0_xso.wr_data** = 0xFFFF0001

**sp0_xso.wr_par** = 0x7

**sp0_xso.wr_zone** = 7

**sp0_xse.wr_data** = 0xFFFFFFFE

**sp0_xse.wr_par** = 0x7

**sp0_xse.wr_zone** = 0xF

After some time, the send crossbar passes this information on to the NCU.

**xso_cu.addr** = 0x0002407

**xso_cu.ctl_par** = 0x09

**xso_cu.cycle** = 0x2

**xso_cu.rdy** = 1

**xso_cu.wr_data** = 0xFFFFFFFE

**xso_cu.wr_par** = 0x7

**xse_cu.wr_data** = 0xFFFF0001

**xse_cu.wr_par** = 0x7

Exactly three clocks later, we get the status signals from the NCU delivered to the XCL.

**cu_xc.status** = 1 ( indicates successful send )

**cu_xc.status_en** = 1

**cu_xc.status_psel** = 0 ( indicates cpu 0 )

Finally, in the second following clock, status information is passed on to the processor.

**xc_sp0.cu_status** = 1

**xc_sp0.cu_status_en** = 1

This completes the cycle for a write with return status.

### 2.4.4.6 Writes with returned status & data - INC

The INC operation can return both data and status following the modification of a comm register. If the register is unlocked, then the INC fails, and neither the comm register nor the scalar register is modified, and a status of 0 is returned. If the register is locked, then the sreg contents is added to the comm register, and this sum replaces both the comm register and the scalar register. A status of 1 is returned in this case.

A special note should be observed regarding INC operations. For these operations, the **bd_sel** field is a 0x9 instead of 0x8. This is true only for INC operations.

The INC interface consists of several. pieces that have already been discussed, namely, status bits, read data, write data, etc.

**Example - INC locked register 0x8020, register = 0xAAAAAAAA AAAAAAAA, sreg contains 0x11111111 11111111, cir 0, tid 0 , sp0**

At the time of the request,

**sp0_xso.addr** = 0x0002060

**sp0_xso.bd_sel** = 0x9 ( indicates NCU, INC operation)

**sp0_xso.ctl_par** = 0x1D

**sp0_xso.cycle** = 0x3

**sp0_xso.rdy** = 1

**sp0_xso.wr_data** = 0x11111111

**sp0_xso.wr_par** = 0xF

**sp0_xso.wr_zone** = 0xF

**sp0_xse.wr_data** = 0x11111111

**sp0_xse.wr_par** = 0xF

**sp0_xse.wr_zone** = 0xF

After some time, the send crossbar passes this information on to the NCU.

**xso_cu.addr** = 0x0002060

**xso_cu.ctl_par** = 0x1D

**xso_cu.cycle** = 0x3

**xso_cu.rdy** = 1

**xso_cu.wr_data** = 0x11111111

**xso_cu.wr_par** = 0xF

**xse_cu.wr_data** = 0x11111111

**xse_cu.wr_par** = 0xF

Exactly three clocks later, we get the status signals from the NCU delivered to the XCL, and read information returned to the NCU.

**cu_xc.status** = 1 ( indicates success )

**cu_xc.status_en** = 1

**cu_xc.status_psel** = 0 ( indicates cpu 0 )

**cu_xro.rd_rdy** = 1

**cu_xro.rd_data** = 0xAAAAAAAA

**cu_xro.rd_par** = 0xF

**cu_xre.rd_rdy** = 1

**cu_xro.rd_par** = 0xF

**cu_xro.rd_data** = 0xAAAAAAAA

Note that this data is the contents of the comm register before the INC takes place. On the second following clock, we get the status returned to the processor:

**xc_sp0.cu_status_en** = 1

**xc_sp0.cu_status** = 1

Finally, the data is returned via the crossbar interface.

**xro_sp0.rd_rdy** = 1

**xro_sp0.rd_data** = 0xAAAAAAAA

**xro_sp0.rd_par** = 0xF

**xre_sp0.rd_rdy** = 1

**xre_sp0.rd_data** = 0xAAAAAAAA

**xre_sp0.rd_par** = 0xF

This completes the cycle for the INC.

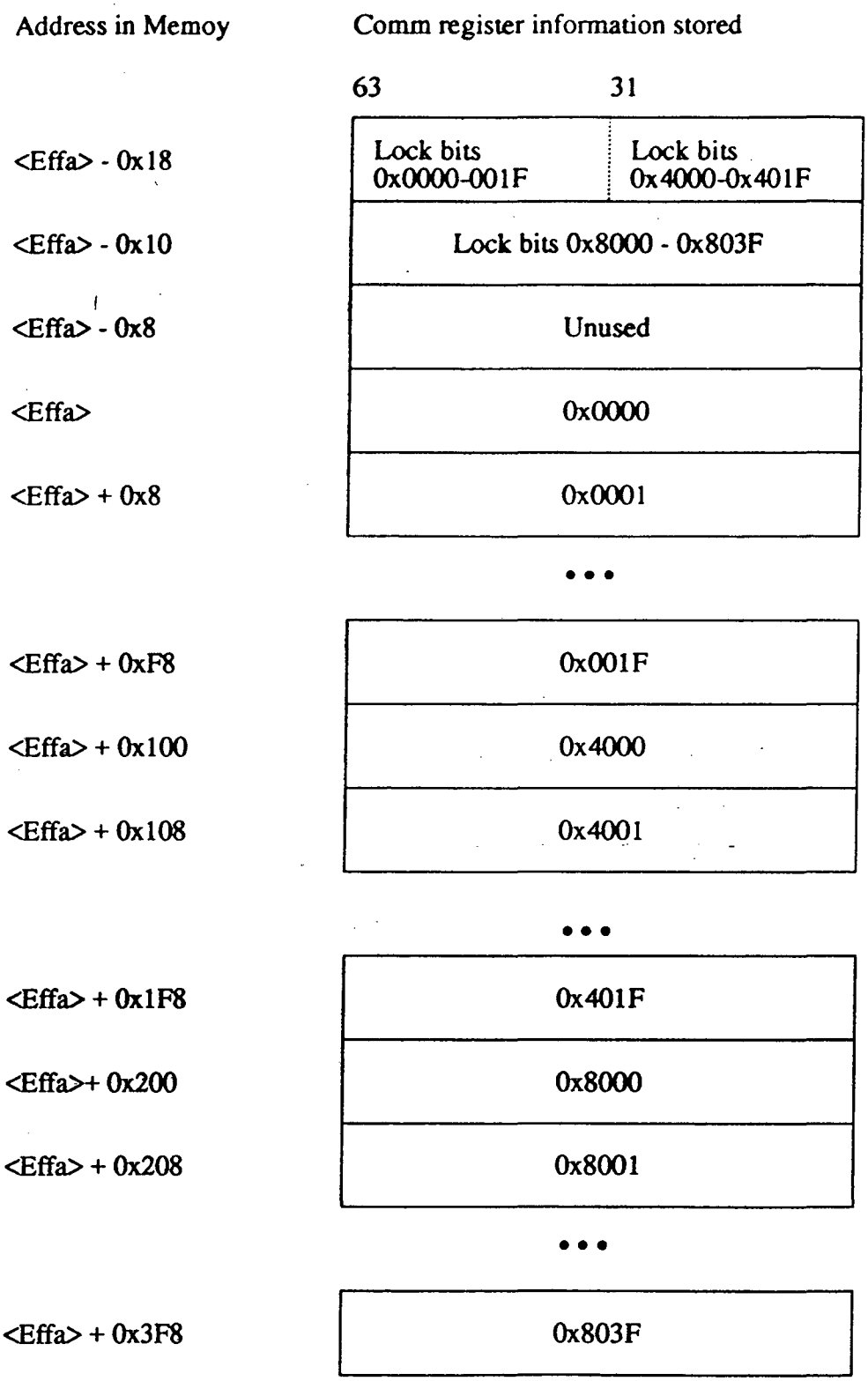
#### 2.4.4.7 Write Communication register set - WRCMR

The WRCMR operation is used during a ldcmr instruction, which writes the complete comm register set of a CIR. Each ldcmr instruction is implemented in microcode by doing a series of WRCMR instructions to each of the comm registers in a particular CIR.The WRCMR takes the MSB of the NCU comm register. The ldcmr instruction has the following sytax:

```
ldcmr           <Effa>,ak
```

In which ak holds the CIR for which the comm registers are loaded, and <Effa> is the address of the memory data base that holds all of the lock bits & data to be loaded for the instruction. Figure 2 shows the memory map for the ldcmr instruction.

FIGURE 2. Memory Database scheme for ldcmr/stcmr



As seen in the figure, the <Effa> begins the longword locations of each of the comm registers in the CIR, beginning with the comm register at 0x0000. The lock bits for the ring 4 registers ( 0x8000 - 0x803F ) are stored at <Effa> - 0x10, and the lock bits of the Ring 0 registers ( 0c0000 - 0x001F, 0x4000 - 0x401F ) are stored at <Effa> - 0x18.

A ldcmr instruction is actually implemented as a series of WRCMR instructions, one for each comm register in the set. This is more easily seen with a look at the C3 microcode for the ldcmr instruction. The data for each comm register is sent via a WRCMR instruction. The lock bits are determined by the MSB of the X-space register known as the RDCMR/WRCMR lock bit shift register. This register has X-space address 0x000. This shift register is initialized at the beginning of the ldcmr instruction with the set of all Ring 0 lock bits. As each WRCMR instruction is done for each ring 0 register, the MS bit is shifted out of this register and becomes the lock bit for the register that is affected by the WRCMR on that cycle. When all of the Ring 0 registers have been restored, the Ring 4 lock bits are sent to the lock bits shift register, and then a WRCMR is done for the Ring 4 registers, restoring all of its registers with the lock bits shifted out of the shift register.

For example, suppose we do the instruction:

```
ldcmr        0x3000,a2
```

where a2 contains 0. Let's suppose we have the following values in memory:

```
0x2FE8       0xAAAAAAAAAAAAAAAA
0x2FF0       0x5555555555555555
0X2FF8       0X0
0X3000       0X1111111111111111
0X3008       0X2222222222222222
0X3010       0X3333333333333333
```

For this instruction, the first thing that would be done is that the longword at 0x2FE8 would be sent to the RDCMR/WRCMR lock bits shift register. Then, a series of WRCMR operations would follow for each of the Ring 0 comm registers, each time accepting the MSB of the lock bits shift register as the lock bit for the data, and then shifting the shift register to the left by one bit. So, in this case, comm register 0x0000 would be written with 0x1111111111111111 with a lock bit of 1. Similarly, comm register 0x0001 would be written with 0x2222222222222222 with a lock bit of 0. After all of the RIng 0 registers are restored, then the lock longword for the Ring 4 registers is put in the lock bit shift register on the NCU. Then, each of the ring 4 registers are written with WRCMR commands.

A special note: For C3 operations, the comm register at address 0x0000 is not really affected by ldcmr and stcmr instructions. This is because the comm register at 0x0000 is "reserved", and not affected by these instructions. As such, the microcode is written to do two WRCMRs to 0x0001 so that the lock bits are shifted appropriately and register 0x0000 is still not affected. Again, see the LDCMR microcode for the details

The WRCMR operation has the same interface to the processor as a write without status ( such as a PUT ). For this reason, I will not go into specific examples of operations

#### 2.4.4.8 Read communication register set - RDCMR

The RDCMR is the converse of the WRCMR operation. The RDCMR operation is used in the stcmr instruction, which writes a communication register set of a particular CIR to memory in the form of the database depicted in Figure 2. This instruction has the following syntax:

```
stcmr        ak,<Effa>
```

in which ak has the CIR of the register set that is to be stored, and <Effa> is the address of the database in memory where the comm register set ( and associated lock bits ) are to be stored.

Each RDCMR operation not only reads the comm register address specified, but also left shifts the lock bit of that comm register into the RDCMR/WRCMR lock bit shift register.

So, in order to implement the stcmr, the microcode first begins doing RDCMRs for the Ring 0 registers until all of them have been read and written out to memory database. Then, the lock longword ( which has been filled up with the appropriate lock bits during the series of RDCMR ops ) is read and written to the appropriate place in memory. Next, all 64 Ring 4 registers are read with a RDCMR command, which simultaneously shifts in the lock bits into the lock bit shift register. Each of the registers are stored to memory as they are read. Finally, after all 64 registers have been written, the lock bit longword is read and then stored to memory.

The RDCMR interface to the crossbar and the NCU behaves exactly as a read without status ( e.g., a GET ). See that section for handshake information for the RDCMR operation.

# 3.0 The Vector Processor Interface

This section will go over the vector processor ( nvp ) / scalar processor ( nsp ) interface. Both the nsp and the nvp are self-contained ( i.e., all of the logic is contained on a single board, so there are only two components involved ion the interface. It should be notes that each scalar processor is matched with exactly one vector processor. So, sp0 uses vp0 only, sp1 uses vp1 only, etc.

## 3.1 Signal Glossary

### 3.1.1 Scalar ( nsp ) to vector ( nvp ) signals

**sp0_vp0.cntx_ctl<1..0>** - Controls the context mode that the VP is in according to the following chart:

| Value | Meaning |
|---|---|
| 0x0 | Normal |

| | |
|---|---|
| 0x1 | Hold |
| 0x2 | Reset |
| 0x3 | Left |

**sp0_vp0.data<63..0>** - 64 bit data bus used for scalar to vector ( SXV ) data transfers.

**sp0_vp0.disp_ep<10..0>** - Entry point of instruction being dispatched to the nvp. Bit <10> actually indicates the polarity of an instruction under mask. (i.e., if bits <9..0> are the EP of an instruction under mask, then bit <10> is 1 if it is a .t instruction, and 0 if it is .f instruction.

**sp0_vp0.disp_ireg<2..0>** - Register indicated by the ireg field of the instruction being dispatched. ( i.e., Vi ).

**sp0_vp0.disp_jreg<2..0>** - Register indicated by the ireg field of the instruction being dispatched. ( i.e., Vj).

**sp0_vp0.disp_kreg<2..0>** - Register indicated by the ireg field of the instruction being dispatched. ( i.e., Vk ).

**sp0_vp0.disp_psw_haz** - Indicates whether the instruction being dispatched affects the psw, in which the nvp must assert **psw_haz** until the instruction completes on the nvp.

**sp0_vp0.disp_rdy** - Indicates when a dispatch to the nvp is occurring.

**sp0_vp0.ieee** - the IEEE bit is set in the psw in the scalar processor.

**sp0_vp0.mxv_rdy** - Indicates that memory read data is ready for the nvp. Note that this memory read data is routed through the nsp to the nvp.

**sp0_vp0.par<7..0>** - Parity for the 64 bit **sp0_vp0.data** bus. Parity bits are reversed with respect to the bytes they represent.

**sp0_vp0.sxv_rdy** - Indicates that a scalar to vector transfer is occurring on this cycle.

**sp0_vp0.vx_req_next** - Indicates that the scalar processor can accept a vector to scalar transfer on the next cycle.

### 3.1.2 Vector ( nvp ) to scalar ( nsp ) signals

**vp0_sp0.data<63..0>** - 64 bit data bus for vector to scalar data transfers ( VXS ).

**vp0_sp0.disp_req_next** - Indicates that the vector processor will accept the dispatch of an instruction on the next cycle.

**vp0_sp0.idle** - Indicates that no instructions are currently being done on the vector processor ( i.e., the nvp is idle ).

**vp0_sp0.mxv_req_next** - Indicates that the vector processor will accept memory data on the next cycle.

**vp0_sp0.par<7..0>** - Parity for the 64 bit vector to scalar data bus. MSB of parity refers to LS byte of data. ( i.e., the parity bits are reversed with respect to the data bytes they represent.

**vp0_sp0.psw_haz** - Indicates to the nsp that an instruction that may affect the psw is executing on the nvp.

**vp0_sp0.set_fdz** - Tells the nsp that a vector instruction has encountered a floating divide by 0, and that the nsp should set its fdz bit in the psw.

**vp0_sp0.set_fsn** - Indicates that a floating square root of a negative operand occurred in a vector instruction, and the nsp needs to appropriately set the IEC information in the psw.

**vp0_sp0.set_ov** - Indicates that a floating point overflow occurred during a vector instruction.

**vp0_sp0.set_ro** - Indicates that a floating point reserved operand occurred during a vector operation.

**vp0_sp0.set_sdz** - Indicates an integer divide by zero occurred during a vector operation.

**vp0_sp0.set_siv** - Indicates integer overflow occurred during a vector instruction.

**vp0_sp0.set_un** - Indicates floating underflow occurred during a vector operation.

**vp0_sp0.sxv_req_next** - Indicates that the nvp will accept a scalar to vector transfer on the next cycle.

**vp0_sp0.vxa_addr<31..0>** - 32 bit address that is given to the nsp for use in calculating vector memory request addresses for some memory operations. The value of **vxa_addr** + vl is sometimes implied as a valid address as well.

**vp0_sp0.vxa_addr_par<3..0>** - Parity for **vxa_addr**. Note that parity bits are reversed with respect to the address bits they represent

**vp0_sp0.vxa_last** - Indicates that address being given to the nsp on this cycle is the last one for the instruction.

**vp0_sp0.vxa_vm<1..0>** - Shows whether the 2 vector addresses given in a vxa transfer are valid. Bit <1> is for **vxa_addr** + vl, and bit <0> is for **vxa_addr.**

**vp0_sp0.vxm_rdy** - Asserted on cycles in which vector memory addresses are being given to the nsp

**vp0_sp0.vxs_rdy** - Indicates that a vector to scalar transfer is being done.

## 3.2 Scalar to vector handshakes

There are three basic types of scalar to vector interfaces, namely, dispatches and scalar-to-vector data transfers ( SXV, MXV).

### 3.2.1 Dispatches

Unlike C2, on C3 vector instructions are dispatched to the nsp, who then dispatches the instruction to the nvp under control of the microsequencer. Thus, the scalar micro-op DISPATCH_VP must be explicitly included in the microcode for vector instructions.

Whenever this micro-op is executed, then the scalar processor will attempt to pass on the following information to the nvp: **disp_ep, disp_ireg, disp_jreg, disp_kreg, disp_psw_haz and disp_rdy.**

The interface itself is very simple. If the nvp can accept a dispatch ( meaning that its queue is not full and can accept another instruction for it ), then it will assert **vp0_sp0.disp_rdy_next,** indicating that a dispatch can be accepted on the next clock. On that next clock, the processor's assertion of **sp0_vp0.rdy** will be accepted, along with the information that it transmits Note that if the processor asserts its **rdy** signal, and the **rdy_next** was not asserted on the previous clock, then the nsp will hold **rdy** to 1 until the clock after the assertion of **rdy_next.**

**Example - Dispatch of ld.b.t <Effa>,v0**

The ld.b.t vector instruction uses the Vk field to indicate the register being loaded, so this is the field that has the pertinent register information.

At the time of the dispatch, the nvp must have asserted **vp0_sp0.disp_req_next**. Assuming that this was the case, we have:

**sp0_vp0.disp_ep** = 0x670

**sp0_vp0.disp_ireg** = 0x1

**sp0_vp0.disp_jreg** =0x0

**sp0_vp0.disp_kreg** = 0x0

**sp0_vp0.disp_psw_haz** = 0x0

**sp0_vp0.disp_rdy** = 0x1

Note that the **disp_kreg** field is the only meaningful register field, as ireg and jreg are unused for this instruction. Note also that bit <10> of the ep is 1 as this is an instruction under mask with positive polarity. The rest of the ep ( bits <9..0> ) indicates the entry point ( 0x270 ) of ld.b to vector register under mask. Finally, **disp_psw_haz** is 0 because loads do not affect the nsp's psw.

### 3.2.2 Scalar to Vector Data Transfers ( SXV )

The SXV operation is a micro-op that allows the nsp to pass data that is required by the vector processor to execute its instruction. For example, if we wanted to do a

```
mov             s0,vl
```

instruction, then the nsp must pass the value of a4 to the vector processor. The microcode for this instruction would use an SXV to pass over this information.

The interface is very simple. First, **vp0_sp0.sxv_req_next** must be a 1. On the following clock, the scalar processor may assert **sp0_vp0.sxv_rdy** to indicate that an SXV transfer is occurring this cycle. At this time, **sp0_vp0.sxv_data** and **sp0_vp0.sxv_par** contain the appropriate information for the nvp.

**Example - Load vl with 0x10**

Assuming that we had a 1 on **vp0_sp0**.sxv_req_next on the previous clock, the cycle of the transfer would have

**sp0_vp0.sxv_rdy** = 1

**sp0_vp0.data** = 0x0000000000000010

**sp0_vp0.par** = 0x7F

### 3.2.3 Memory to vector transfers ( MXV )

Memory data that is returned to the vector processor is done via MXV transfers, in which the memory data is routed to the nvp via the nsp. The MXV interface is exactly the same as for an SXV; the only difference is that **sp0_vp0.mxv_rdy** and **vp0_sp0.mxv_req_next** are the handshake signals.

## 3.3 Vector to scalar handshakes

This section deals with the various information that the nvp will pass to the nsp during the course of executing instructions.

### 3.3.1 Vector to scalar data transfers ( VXS )

Occasionally the nvp will need to pass information back to the nsp. This done via the VXS interface. The handshakes are identical to the most C3 interfaces; namely, we begin with the assertion of **sp0_vp0.vx_req_next**. On the following clock, all vector information is presented with the assertion of **vp0_sp0.vxs_rdy**. At that time the data and parity on the **vp0_sp0.data** and **vp0_sp0.par** buses are valid.

### 3.3.2 Vector to Memory transfers

On some vector memory operations, some information must be passed to the nsp. For example, for vector loads that require the passing of addresses to the VAG ( vector address generator ) in the NDC subsection of the nsp. Also, vector writes must pass write data over to the nsp. This is done using vxm transfers. The basic interface is as follows:

Following the assertion of **sp0_vp0.vx_req_next**, the nvp will activate **vp0_sp0.vxm_rdy**. At this time, the appropriate vector address will appear on the **vp0_sp0.vxa_addr** bus. If the operation is a write, then **vp0_sp0.data** will contain the write data at this time. This is done for the appropriate number of times for a given instruction. ( For example, if vl is 0x10, then 16 address transfers are necessary if it is not occurring at rate 2X. )

**vp0_sp0.vx_last** is asserted on the last transfer for a given vxm operation.

Keep in mind that not all vector memory operations require addresses from the vector processor to be explicitly provided.

### 3.3.3 Vector to scalar status information

Finally, the vector processor relays status information regarding the instructions that are executing in the various nvp function pipes. These are the **vp0_sp0.set_<psw bit>** signals. Whenever any executing instruction encounters a situation which should set a psw bit on the nsp, then these signals are asserted asynchronously ( i.e., no handshake is required.)

The last vector to scalar signal of interest is **vp0_sp0.psw_haz**, which is asserted as long as an operation is occurring on the nvp that may affect the psw of the nsp.

Finally, **vp0_sp0.idle** indicates that there is no instruction currently executing on any of the function pipes, nor is there any waiting in the queue to be dispatched to a function pipe.

## 3.4 Vector/Scalar Instruction interaction

I will attempt in this section to briefly discuss the interaction between the processors for each class of instruction. Because of the wide variety of permutations for some of these instruction classes, I will not attempt to cover in detail every possible situation.

### 3.4.1 Vector Memory Operations

In order to understand the various forms that vector memory operations that can occur, a quick review of vector memory op types is warranted. These fields are determined by the us microcode of the nsp. Recall that if the MEM_OP_REQ field is 1 and the MEM_OP_AT_TYPE is 2 or 3, then the MEM_OP_TYPE fields have the meaning depicted in Table 7.

**TABLE 7. MEM_OP _TYPE Field Definitions**

| Bits | Meaning |
|---|---|
| xxxx01 | Read |
| xxxx10 | Write |
| xxxx11 | Test and Modify |
| xxx1xx | Start vector address generator (VAG) |
| xx11xx | Start VAG for scalar store extended |
| x1x1xx | Start VAG for operation under mask |
| 1xx1xx | Start VAG for vector indexed operation |

These bits will determine the amount of vector/scalar interaction required by the VAG on the NDC section of the NSP. By looking up the vector instruction in question in the us microcode, one can determine the value of each of these bits.

#### 3.4.1.1 Simple vector loads

Simple vector operations are those that read all of the following locations of memory:

Effa, Effa+vs, Effa+2vs, Effa+3vs, ... Effa+(vl - 1)vs.

Effa is the calculated effective address in the instruction, vl is the vector length register, and vs is the vector stride register. Since the ndc of the nsp has its own copies of the vs and vl registers, no information is required from the nvp for simple operations. Examples of simple vector loads are LD.L_V, LD.W_V, etc.

By looking at the us microcode on the nsp, one can see that the VREAD micro-op is used for these instructions. Note that the VREAD op sets the us field MEM_OP_TYPE to be 00101, meaning start VAG for a read operation ( see Table 7 ).

When the simple load is at the upc level, **us_memop_req** is asserted indicating to the ndc that a memory request is at the upc level. At this time, **us_mfp_op** will contain the MEM_OP_TYPE in bits <10..5> of this field. At this point the memory request is handled as any other VAT memory request to the ndc.

When the ndc gets around to selecting the vector load for processing, it will translate the address to physical and begin issuing requests to the crossbar for reads just like any memory request. The nvp/nsp does not concern itself with the instruction until read data has made it back from memory to the nsp. At this time, the nsp begins passing this information back to the nvp via the mxv interface in the starting with the data at Effa and ending with the data from ( Effa + (vl - 1)vs ).

Examples of simple memory reads follow.

**Example 1 - ld.l_v**

This example walks through a vector load of longwords beginning at address 0x1000. Assume that the vs register is 8 and that vl is 8 also. Suppose that the contents of address 0x1000 through 0x1038 contain the following data:

| Address | Data |
|---|---|
| 0x1000 | 0x0000000088888888 |
| 0x1008 | 0x1111111199999999 |
| 0x1010 | 0x22222222AAAAAAAA |
| 0x1018 | 0x33333333BBBBBBBB |
| 0x1020 | 0x44444444CCCCCCCC |
| 0x1028 | 0x55555555DDDDDDDD |
| 0x1030 | 0x66666666EEEEEEEE |
| 0x1038 | 0x77777777FFFFFFFF |

We will now trace the course of this instruction through the nsp and nvp. When this instruction ( entrypoint == 0xC76 ) is at the UPC level, we get us_mem_op_req == 1 and us_mfp_op == 0x0B2. This decodes into a MEM_OP_TYPE of 0b000101, which is start VAG for read operation. The MEM_OP_AT_TYPE is 0b10, indicating virtual address translation (VAT). Finally, the MEM_OP_DST field is 0b010, so the destination is the nvp.

When the request has filtered through the ndc and is selected for processing, we will begin to see the logical addresses for this request appear as sa1_miss1 for each longword that we request. So, we would see 0x1000, 0x1008, 0x1010, 0x1018, 0x1020, 0x1028, 0x1030, and 0x1038 appear as sa1_miss1 on the scalar processor. Since the ndc has its own copies of the vl and vs registers, it knows that 0x1038 is the last vector address of the instruction.

The next step, assuming that the data for these addresses are not encached, is for the ndc to translate these addresses to physical addresses and submit requests to the crossbar. These are treated just as any crossbar request, which are described in the section on the processor/crossbar interface.

After reads are done to the crossbar, then the data is returned to the scalar processor from the xbar interface ( just as any read ). Returning data from memory is routed through the nrc gate arrays which then pass it on to the nvp on the sp0_vp0.data bus through the MXV handshaking process described previously. So, for each longword beginning at 0x1000, there will be an assertion of **sp0_vp0.mxv_rdy** at the time the data appears on the **sp0_vp0.data** ( and **par** ) bus. The first will look like

**sp0_vp0.mxv_rdy** = 1

**sp0_vp0.data** = 0x0000000088888888

**sp0_vp0.par** = 0xFF

Similarly, each of the longwords will be transmitted to the nvp in this manner. This information is then given to the vector register file and stored in the appropriate register.

### 3.4.1.2 Loads at rate 2X

For loads of words, halfwords, and bytes, the load occurs at rate 2X. This means that each scalar to vector transfers via the MXV interface contain two elements instead of one. This is achieved by placing the first element on the upper half of the data bus and the next element on the lower half. For example, if the first two words of a vector load were 0x55555555 for element[0] and 0xaaaaaaaa for element [1], then the MXV data would be

**sp0_vp0.data** = 0x55555555aaaaaaaa.

The bottom half would be written to element [1] and the top half to element[0].

**Example - ld.w_v**

Suppose vs=vl=8, and we are going to do a ld.w into v1 of address 0x3000. We have the following data in memory,

| | |
|---|---|
| 0x3000 | 0x8888888800000000 |
| 0x3008 | 0x9999999900000000 |
| 0x3010 | 0xaaaaaaaa00000000 |
| 0x3018 | 0xbbbbbbbb00000000 |
| 0x3020 | 0xcccccccc00000000 |
| 0x3028 | 0xdddddddd00000000 |

| | |
|---|---|
| 0x3030 | 0xeeeeeeee00000000 |
| 0x3038 | 0xffffffff00000000 |

The load would behave precisely as the ld.l described in the previous example, except that when the nsp got around to transferring the read data over to the nvp, the following sequence would occur:

| Cycle | sp0_vp0.data |
|---|---|
| 1 | 0x8888888899999999 |
| 2 | 0xaaaaaaaabbbbbbbb |
| 3 | 0xccccccccdddddddd |
| 4 | 0xeeeeeeeeffffffff |

Of course, each of these cycles must have **sp0_vp0.mxv_rdy** asserted to have valid data on them. Also, the cycles need not occur consecutively; there could be some delay between MXV transfers.

### 3.4.1.3 Vector loads under mask

A vector register load under mask will only affect those elements of the vector register being loaded if the corresponding bit in the vector merge ( vm) register is appropriately set ( or clear ).

The vm register is a 128-bit register, with each bit corresponding to an element in the vector register. The MSB of the vm register corresponds to element 127 of a vector register, and bit 0 corresponds to element 0. The vm is frequently written as a pair of longwords in the following manner:

<upper longword> <lower longword>

in which the upper longword consists of bits <127..64> and the lower longword consists of bits <63..0>.

For vector loads under mask, the status of the corresponding vm bit of an element determines whether that element is loaded. For example, if we have the instruction

ld.l.t Effa,v1

The instruction would execute a vector longword load, but only elements whose corresponding vm bit was 1 was set would be loaded. So, if the vm was

0x0000000000000000 0x0000000000005af0

and, for simplicity's sake, the vl was 16, then only elements 14, 12, 11, 9, 7, 6, 5 and 4 of register v1 would be loaded. Similarly, if the instruction was

ld.l.f Effa,v1

then only elements 15, 13, 10, 8, 3, 2, 1, 0 would be loaded. Note that the effective address for each element is still < # of element - 1 > * vs, regardless of how many elements are actually being written.

The micro-op for loads under mask is VREADM. This sets the MEM_OP_TYPE to 0b010101, with the bits set up for read, start VAG, instruction under mask.

The vector dispatches for operations under mask have a special twist. The MSB of **sp0_vp0.disp_ep** will contain the polarity of the operation mask. For example, for ld.l.t, the entrypoint dispatched to the nvp will be 0x776, but a ld.l.f will dispatch as 0x337.

Address calculation is slightly different for operations under mask because the ndc will only request addresses that have the appropriate bit of the vm register set. Because the ndc does not have a copy of the vm register, this information must be transmitted from the nvp.

Since the ndc already has the Effa of the instruction, all it need are the offsets to be added to Effa to calculate the address of a particular element. For example, if vs=8, and element ten was the location being loaded, then the nvp would have to send over an offset of 8*10 = 0x50.

The nvp sends over this information via the VXM interface. The address offsets appear on the **vp0_sp0.vxa_addr** on the same cycle as **vp0_sp0.vxm_rdy** is asserted. The nvp also sets the **vp0_sp0.vxa_vm** bits appropriately to tell the nsp whether the address being transferred is one that should be requested (i.e., that the vm bit for the element whose address is being transferred is a 0 for .f operations and a 1 for .t operations. ) The nvp tells the nsp that the last address of the instruction is occurring on a cycle by asserting **vp0_sp0.vxa_last.**

Longword reads occur at rate 1X, so only one address per VXM transfer is transmitted. For smaller sizes, an implied second offset is transmitted as ( **vp0_sp0.vxa_addr** + vs ). The vm bit for this implied is **vp0_sp0.vxa_vm**<1>. Of course, bit <0> of the vm signal refers to the vxa_addr.

One other potential twist is that the VXM addresses may occur at an accelerated rate. This means that if the nvp knows that there are number of invalid vm bits in a row for a load under mask, it may skip to the next valid one rather than issue VXMs for the invalid address with the vp0_sp0.vm bits set to 0. For example, if we were doing a longword read and the lower half of the vm was 0x0000000000000033, and vs was 0x10, then the following would be the sequence of VXMs for non-accelerated mode:

| Cycle/vm bit | vxa_addr | vxa_vm |
|---|---|---|
| 0 | 0x00 | 1 |
| 1 | 0x10 | 1 |
| 2 | 0x20 | 0 |
| 3 | 0x30 | 0 |
| 4 | 0x40 | 1 |
| 5 | 0x50 | 1 |
| 6 | 0x60 | 0 |
| 7 | 0x70 | 0 - vxa_last = 1 here |

The same instruction in accelerated mode would eliminate the cycles in which the vm bit was 0. Thus the series of VXM transfers would be:

| Cycle | vxa_addr | vxa_vm |
|---|---|---|
| 0 | 0x00 | 1 |
| 1 | 0x10 | 1 |
| 2 | 0x40 | 1 |
| 3 | 0x50 | 1 - vxa_last == 1 here |

Whether accelerated mode is being used depends upon what instructions the nvp is attempting to chain with and is non-deterministic. The important thing to note is that the nsp doesn't care whether the nvp is chaining or not, as it will only issue requests for those values with a **vp0_sp0.vxa_vm** bit of 1 anyway.

### Example - Longword loads under mask - non-accelerated mode

Suppose we want to do a ld.l.t of address 0x3000 with vl = 0x10, vs = 0x8, and the least significant halfword of vm is 0xC6F0. The memory data is:

| Address | Data |
|---|---|
| 0x3000 | 0x0000000088888888 |
| 0x3008 | 0x1111111199999999 |
| 0x3010 | 0x22222222AAAAAAAA |
| 0x3018 | 0x33333333BBBBBBBB |
| 0x3020 | 0x44444444CCCCCCCC |
| 0x3028 | 0x55555555DDDDDDDD |
| 0x3030 | 0x66666666EEEEEEEE |
| 0x3038 | 0x77777777FFFFFFFF |
| 0x3040 | 0x8888888800000000 |
| 0x3048 | 0x9999999911111111 |
| 0x3050 | 0xAAAAAAAA22222222 |
| 0x3058 | 0xBBBBBBBB33333333 |
| 0x3060 | 0xCCCCCCCC44444444 |
| 0x3068 | 0xDDDDDDDD55555555 |
| 0x3070 | 0xEEEEEEEE66666666 |
| 0x3078 | 0xFFFFFFFF77777777 |

As always, the MEM_OP_TYPE field is sent over to the ndc embedded in the **us_mfp_op** field. In this case we get a **us_mem_op_req** of 1 with us_mfp_op = 0x2B2, which has the MEM_OP_TYPE bits as 0b010101, which is the code for start VAG for read under mask.

After this, the instruction is dispatched to the nvp via the dispatch interface described previously.

The next step is for the nvp to begin sending over address offsets via the VXM bus. SInce this is a longword operation ( and therefore at rate 1x ) then only one address per VXM cycle will occur. The addresses & vm bits sent over would be of the form:

| Cycle | vxa_addr | vxa_vm |
|---|---|---|
| 0 | 0x00 | 0 |
| 1 | 0x8 | 0 |
| 2 | 0x10 | 0 |
| 3 | 0x18 | 0 |
| 4 | 0x20 | 1 |
| 5 | 0x28 | 1 |
| 6 | 0x30 | 1 |
| 7 | 0x38 | 1 |
| 8 | 0x40 | 0 |
| 9 | 0x48 | 1 |
| 10 | 0x50 | 1 |
| 11 | 0x58 | 0 |
| 12 | 0x60 | 0 |
| 13 | 0x68 | 0 |
| 14 | 0x70 | 1 |
| 15 | 0x78 | 1, vxa_last == 1 here |

After being returned from memory to the nsp, the scalar processor passes on the requested reads to the nvp via the MXV interface. The following words are the ones that are transmitted to the nvp:

| Cycle | Data |
|---|---|
| 1 | 0x44444444CCCCCCCC |
| 2 | 0x55555555DDDDDDDD |
| 3 | 0x66666666EEEEEEEE |
| 4 | 0x77777777FFFFFFFF |
| 5 | 0x9999999911111111 |
| 6 | 0xAAAAAAAAA22222222 |
| 7 | 0xEEEEEEEE11111111 |
| 8 | 0xFFFFFFFF00000000 |

The nvp would then route this information to the vector register file.

### Example - accelerated mode

The previous instruction in accelerated mode would be exactly the same, except that only vxa_addr's with a vxa_vm == 1 would be sent to the scalar processor in the VXM transfers.

### Example - ld.w.f accelerated

Suppose we want to do a ld.w.f of address 0x3000 with vs = 8, vl = 0x10, and vm = 0xC359. The data in memory is:

| Address | Data |
|---|---|
| 0x3000 | 0x0000FFFF00000000 |
| 0x3008 | 0X1111EEEE00000000 |
| 0x3010 | 0X2222DDDD00000000 |
| 0x3018 | 0X3333CCCC00000000 |
| 0x3020 | 0X4444BBBB00000000 |
| 0x3028 | 0X5555AAAA00000000 |
| 0x3030 | 0x6666999900000000 |
| 0x3038 | 0x7777888800000000 |
| 0x3040 | 0x8888777700000000 |
| 0x3048 | 0x9999666600000000 |
| 0x3050 | 0xAAAA555500000000 |
| 0x3058 | 0xBBBB444400000000 |
| 0x3060 | 0xCCCC333300000000 |
| 0x3068 | 0xDDDD222200000000 |
| 0x3070 | 0xEEEE111100000000 |
| 0x3078 | 0xFFFF000000000000 |

The instruction proceeds exactly like the longword example until it is time for the nvp to begin sending address offsets to the nsp via the VXM interface. Recall that operations smaller than longword in size proceed at rate 2x, which means that two address offsets ( and thus two corresponding vxa_vm bits ) are passed to the nsp on each VXM cycle. Bit <1> of **vp0_sp0.vxa_vm** corresponds to the offset **vxa_addr** + vs, and bit <0> refers to **vxa_addr**. So, for this instruction we would have ( in accelerated mode ) the following information transmitted via the VXM interface:

| vxa_addr | vxa_vm | valid offsets |
|---|---|---|
| 0x00000000 | 2 | 0x00000008 |
| 0x00000010 | 1 | 0x00000010 |
| 0x00000020 | 2 | 0x00000028 |
| 0x00000030 | 2 | 0x00000038 |
| 0x00000050 | 3 | 0x00000050, 0x00000058 |
| 0x00000060 | 3 | 0x00000060, 0x00000068 |
| 0x00000080 | 0 | none, vxa_last = 1 here |

Note that only offsets corresponding to elements 1, 2, 5, 7, 10, 11, 12, and 13 are transmitted with a vxa_vm bit of 1. These bits correspond to the 0 bits in the vm, which verifies our expectations as this was a ld.w.f.

When data is returned from memory ( via the nsp ) to the nvp, each MXV transfer will contain either two words of data or one, depending upon whether the VXM address offset transfer had two valid addresses or not. So, for example, the return data for element 1

would only contain that word, but elements ten and eleven would be transferred on the same MXV transfer, with the word for element 10 on the top half of the data and element 11 on the bottom half. For single word MXVs, the word always appears on the lower half of the bus. Therefore, the MXV sequence for this sequence ( with "X" == don't care ) is

| Cycle | Data |
|---|---|
| 1 | 0xXXXXXXXX1111EEEE |
| 2 | 0xXXXXXXXX2222DDDD |
| 3 | 0xXXXXXXXX5555AAAA |
| 4 | 0xXXXXXXXX77778888 |
| 5 | 0xAAAA5555BBBB4444 |
| 6 | 0xCCCC3333DDDD2222 |

### 3.4.1.4 Vector load with indices

The final vector load type is the load with vector indices. This uses the address contained in address register a5 as a base address, and each vector element's address is calculated by using the index vector register specified in the instruction as an offset to the base address. That is, if the instruction is

```
ldvi        vj,vk
```

then the pseudocode for the instruction is

$$vk[i] = c( a5 + vj[i] ) \qquad \text{(EQ 1)}$$

where c(Effa) means " the content in memory of address Effa."

The ldvi ( both normal and under mask ) uses the VREADM_IX construct, which sets the MEM_OP_TYPE to be 0b110101. Note that this encodes "start VAG for index read under mask." Although not all ldvi instructions are under mask, the nsp us microcode is set up this way for simplicity. The nvp knows to always set the vxa_vm bit for all offsets sent to the nsp for instructions that are not under mask. This way, all address offsets are used, which is the case for instructions not under mask.

One thing to note about ldvi instructions is that rate 2X operations are not possible. This is because the vxa_addr bus to the nsp is only 32 bits wide, so only one index per transfer is possible.

As always, the process for a ldvi begins with the dispatch of the instruction to the nvp via the dispatch interface. When the instruction is ready to proceed, the nvp begins dispatching address offsets for each element to the nsp via the VXM interface. This is exactly the same as for loads under mask, except that rate 2X operations are not possible.

**Example - ldvi.l**

Suppose we want to do the following instruction

```
ldvi        v1,v2
```

Let's suppose scalar register a5 contains the value 0x00003060, vl = 8, and v1 contains the following values:

v1[000]=0000000000000000
v1[001]=fffffffffffff8 ( note this is -8 )
v1[002]=0000000000000008
v1[003]=fffffffffffff0 ( note this is -16 )
v1[004]=0000000000000010
v1[005]=ffffffffffffe8 ( note this is -24 )
v1[006]=0000000000000018
v1[007]=ffffffffffffe0 ( note this is -32 )

Further, suppose we have the following data in Memory:

| Address | Data |
|---|---|
| 0x80003040 | 0x7777777788888888 |
| 0x80003048 | 0x55555555aaaaaaaa |
| 0x80003050 | 0x33333333cccccccc |
| 0x80003058 | 0x1111111eeeeeeee |
| 0x80003060 | 0x00000000ffffffff |
| 0x80003068 | 0x22222222dddddddd |
| 0x80003070 | 0x4444444bbbbbbbb |
| 0x80003078 | 0x6666666699999999 |

The instruction would proceed precisely as a ld.l.x would, and when we got to the point of transferring address offsets to the nsp via the VXM interface, we would have the following sequence of offsets transmitted in this order via the VXM interface: 0x00000000, 0xFFFFFFF8, 0x00000008, 0xFFFFFFF0, 0x00000010, 0xFFFFFFE8, 0x00000018, 0xFFFFFFE0. This last value would have vxa_last asserted as well. vxa_vm would be 1 for each of these values since this operation is not under mask; had this been a ldvi.l.x, then the vxa_vm bit would be set only if the vm bit corresponding to the index was appropriately set or clear, depending in the polarity of the .x.

When the data is returned to the nvp via the VXM interface, the following data words would be received by the nvp in this order:

0x00000000FFFFFFFF
0x11111111EEEEEEEE
0x22222222DDDDDDDD
0x33333333CCCCCCCC
0x44444444BBBBBBBB
0x55555555AAAAAAAA
0x6666666699999999
0x7777777788888888

This would complete the ldvi.l instruction's interface between the two processors.

### 3.4.2 Vector stores

Unlike vector loads, which has three classes of microinstructions ( straight loads, loads under mask, and load with vector indices ), Only two are necessary for vector stores because straight vector stores ( e.g. ST.L_V ) is handled as if were under mask ( e.g. ST.L.X_V ),, with the nvp ensuring that all vxa_vm bits are set for each VXM transfer of address offsets. These two macro instruction types are both implemented my the VWRITEM micro-op, which sets the MEM_OP_TYPE to be 0b010110. The other instruction type is store with vector indices( both normal and under mask ) which is implemented with the us micro-op VWRITEM_IX, which sets the MEM_OP_TYPE field to 0b110110

The differences between these instruction types and the ones discussed in the section on vector loads are twofold. First, the nvp must provide write data on the **vp0_sp0.data**with each VXM transfer. Secondly, no data is returned to the nvp after the VXM transfers ( i.e., no read data is returned as these are stores. )

Note that for VWRITEM operations, an word, halfword, or byte operation can occur at rate 2X, which means that an operand will appear on each half of the data bus during VXM transfers, with the upper half corresponding to the offset of vxa_addr , and the lower half to vxa_addr. Rate 2X operations are not possible for operations under mask.

**Example - st.w v1,0x3000**

Suppose we have v1 equal to 8, vs = 8, and the following data in v1:

| Element | data |
|---|---|
| V1[0] | 0x00000000 |
| V1[1] | 0x11111111 |
| V1[2] | 0x22222222 |
| V1[3] | 0x33333333 |
| V1[4] | 0x44444444 |
| V1[5] | 0x55555555 |
| V1[6] | 0x66666666 |
| V1[7] | 0x77777777 |

After the dispatch, the following data will be transmitted to the scalar processor via the VXM interface:

| Offset | Data | vxa_vm |
|---|---|---|
| 0x00000000 | 0x0000000011111111 | 3 |
| 0x00000010 | 0x22222222333333333 | 3 |
| 0x00000020 | 0x444444455555555 | 3 |
| 0x00000030 | 0x6666666677777777 | 3, vxa_last == 1 here |

Note that the upper half of the data bus corresponds to the offset represented by vxa_addr and vxa_vm<0>, and the lower half of the data bus corresponds to vxa_addr + vs and vxa_vm<1>.

### 3.4.3 Non-memory vector functions

Vector functions that do not involve memory have little or no interaction with the scalar processor. If the nvp requires some data from the nsp, then this is usually sent over shortly after dispatch in the form of an SXV operation. If data is returned to the nsp after the instruction has been executed, it is done via the VXS interface.

Because of the myriad of intra-nvp instructions that can be done, I will not discus each type as I did for the memory operations. However, I will try to present a feel for how the "typical" vector instruction is processed on the nvp. A good thing to have handy for this section would be the neptune nvp board specification and block diagram.

The nvp consists of the following subsystems:

is - input staging - Used to stage MXV data, SXV data, and context data.

vd - vector dispatch - accepts dispatches from the nsp and dispatches the appropriate functional pipe.

a_vm, l_vm, m_vm - vm copy for each of the three function pipes. Also houses the level 1 registers for each of the functional pipes. Also drives the register selection fields that are given to the nvrf for element selection for register file reads ( a_x_cnt, a_y_cnt, etc ) and the active signals for each pipe ( a_active, m_active, l_active).

nvrf - neptune vector register file, implemented in 8 identical gate arrays. This system houses the vector registers and provides operands to the functional pipes and data and address offsets to the nsp.

nq - Houses level 2 and 3 registers for the functional pipe control. Also calculates the "q" level signals.

bd ctl - control for "backdoor" writes to the nvrf.

afp - add function pipe - consists of six ndivs ( neptune dividers ), one nmisc ( neptune miscellaneous functional unit ) and one nfad ( neptune floating point adder ).

mfp - multiply function pipe - consists of four nmuls ( neptune multiplier ), one nfad and one nmisc.

afc - add function pipe controller

mfc - multiply function pipe controller.

Note that the "afp" and "mfp" logical blocks only contain the gate arrays the implement the functions required for the instructions. I will use "afp" and "mfp" in a larger context as well, including all of the control and staging registers that implement an instruction on either of the pipes. Thus, the term "mfp" will frequently be used to refer to parts of the vd, nvrf, m_vm, nq, mfp, and mfc, which are defined in the preceding glossary.

Each of the function pipes is a five stage pipelined subsystem. with4 staging levels ( the signals associated with each level are named m_*, m1_*, m2_*, and m3_* ) before data gets to the appropriate function units, and a fifth stage ( called "backdoor" or "q" stage denoted as mq_* for signals ) that takes result data from the pipes and routes it back to the nvrf. Each microinstruction from the ua ( or um ) rams will proceed down each level of the pipe. Barring a clock extend ( meaning that clocking is suspended for some reason ), steps one through four of the pipe follow directly ( i.e., an instruction is at level 1 exactly one clock after it is at level 0 (m), etc. ). The amount of time between levels three and the "backdoor" level depends upon the PIPE_LENGTH field that is read from the VD rams.

For the purpose of this discussion, I will present this information as if it were an instruction destined for the afp. Note that the logic is duplicated for the mfp, with all signals containing the prefix m* instead of a*, which is presented here.

Any vector instruction begins with a dispatch from the nsp to the nvp via the dispatch interface. If any scalar data is needed for the instruction, then in is sent over shortly after the dispatch with an SXV transfer. The dispatch information for the instruction will then sit in a three-deep queue of instructions to be dispatched in the vd subsystem until it is the instruction's turn to be executed ( i.e., it is at the front of the queue.) Note that the instructions will always be executed on the nvp in the order in which they are dispatched.

When the instruction reaches the front of the queue, bits <9..0> of the entrypoint that was dispatched to the nvp are applied to the vd microcode rams. The fields read from the dispatch rams contain information about the dispatch of the instruction. If there are reasons why the instruction can not be dispatched immediately,( e.g., the necessary function pipe is already busy or there is a data hazard ), the instruction will stay at the front of the queue until enough clocks have elapsed so that the hazards are resolved. When the instruction is able to proceed, the vd logic will assert **a_dispatch** ( assuming that the afp is the appropriate pipe, as we are doing in this discussion.

At this time, the ep ( entrypoint ) for the um rams ( which comes from the vd ram ) is appended with 0b000 to form the field **a_uaddr**. This field is used as an address for the ua microcode rams. The ua rams are similar to the us microcode rams on the nsp. Each microinstruction in the ua rams contains the necessary information for the afp to perform a step of the instruction on the pipe. These fields show up as the **af_* signals**, which are the direct outputs of the ua ram, on the nvp.

On the next clock, the first microinstruction for the macroinstruction is at level 0, or the a_* level. We now have **a_active** asserted, and **a_upc** contains the address that was **a_uaddr** on the previous cycle. Meanwhile, the address of the next microinstruction is being applied to the ua rams on **a_uaddr**.

Next, the first microinstruction reaches level1 ( m2_active = 1), the second microinstruction reaches level 0, and the third one is addressing the ua rams.

The first microinstruction then proceeds to level 2 ( n2_active = 1 ), and then to level 3, with the subsequent microinstructions following it down the pipe.

The operation on the data occurs at level three of the pipe. At that time, the appropriate gate array is started on the pipe to perform the operation. Note that in the cases of multiply or divides, there are more than one gate array that can be dispatched. The gate array that begins calculation with the first element is arbitrarily selected based upon a counter in the function pipe controller. After the first element's operation is dispatched to a gate array, the rest of the gate arrays of that type are selected in sequential sequence. ( For example, if we were doing a divide and the counter pointed to gate array three, then the sequence of gate array operations would occur as ndiv3, ndiv4, ndiv5, ndiv0, ndiv1, ndiv2, ndiv3, ndiv4, etc.

The pipe_length field of the vd microcode indicates the number of clocks between the time a microinstruction is at the m3 level and the time it gets to the mq ("backdoor") level, where the result is produced for writing to the nvrf ( if such a write is specified for the microinstruction.)

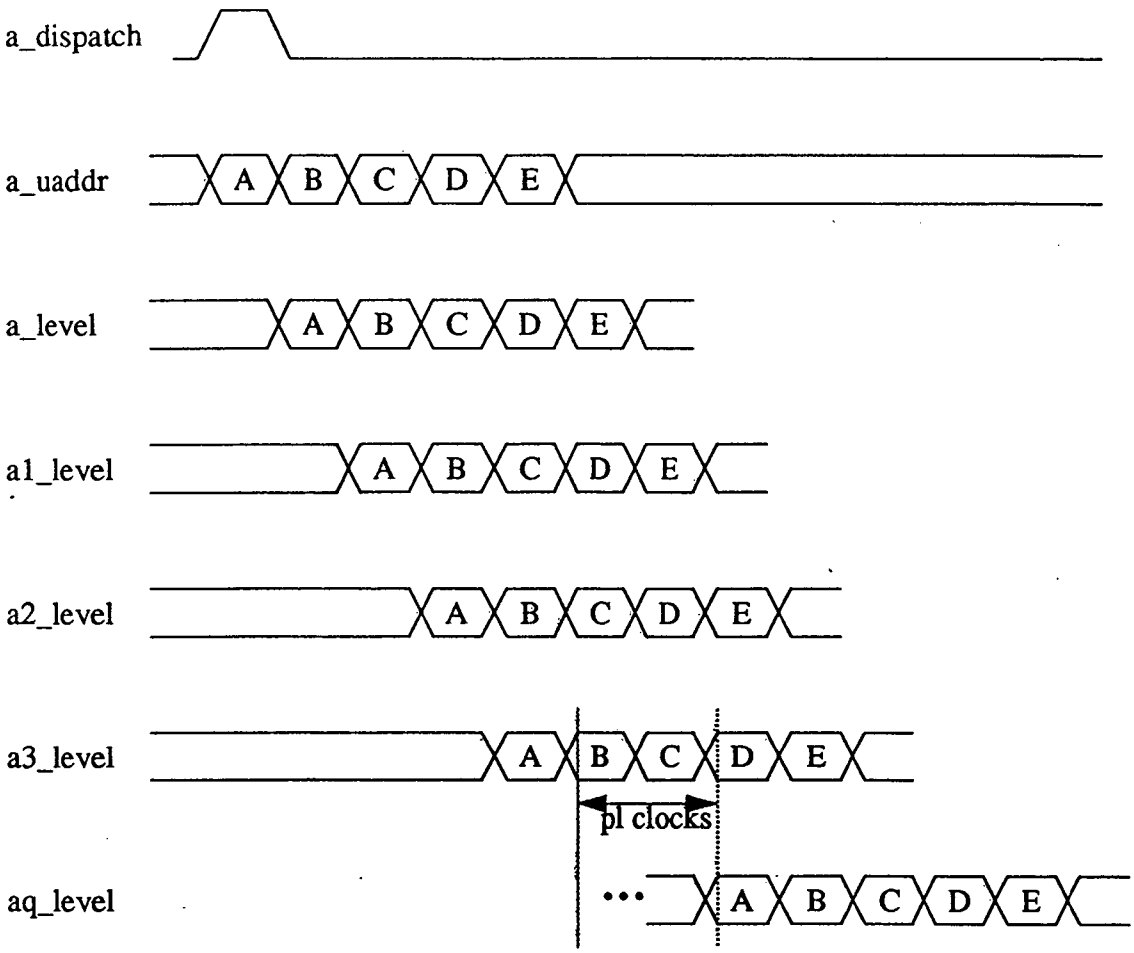
There is now a delay of length PIPE_LENGTH ( which is determined by the field from the dispatch rams ) before the data is ready for the "backdoor" level. This level is the one denoted by the prefix aq_*. When a result reaches this level then it is ready to be written into the nvrf ( if this is desired ).

Figure 3 shows the progression down the afp of a vector instruction.In this example, the vector instruction has five ua microcode steps, labeled A,B,C,D, and E.

Keep in mind that the function pipe doesn't necessarily advance each clock; if a clock extend occurs, the pipe stalls ( i.e., does not advance ) for as long as the clock extend is occurring.

When the instruction is completing, then each stage is "de-activated" in the order in which they are in the pipe. That is, m_active is the first signal de-asserted, followed by levels 1, 2, and 3. When the last microinstruction at level 3 occurs, then pipe_lentgth clocks later mq_active is de-asserted and the instruction is complete.

FIGURE 3. AFP pipe progression




A word should be inserted here about vector clock extends. These are situation in which the pipe stages are held to allow needed data to process before the stages are allowed to advance. The most common example is when a divide instruction is done and the latency of the operation takes longer than the pipe length. In this case, **ck_xtnd** is asserted, which holds all registers in the pipe but allows divide data and vector interfaces to other parts ( e.g. the nsp ) to continue processing. An example of a clock extend will be given in the divide example.

Now, I will present a couple of examples to illustrate the functional pipes in action.

### Example - MUL.L_SV

Suppose we want to execute the instruction

```
mul.l        v1,s7,v2
```

Let's suppose that in this case, we have

```
S7 =         0x0000000000000002
```

| | |
|---|---|
| V1[0] = | 0x00000000FFFFFFFF |
| V1[1] = | 0x00001111EEEEEEEE |
| V1[2] = | 0x00002222DDDDDDDD |
| V1[3] = | 0x00003333CCCCCCCC |
| V1[4] = | 0x00004444BBBBBBBB |
| V1[5] = | 0x00005555AAAAAAAA |
| V1[6] = | 0x0000666699999999 |
| V1[7] | 0x0000777788888888 |
| V1[8] | 0x0000888877777777 |
| V1[9] | 0x0000999966666666 |
| V1[10] | 0x0000AAAA55555555 |
| V1[11] | 0x0000BBBB44444444 |
| V1[12] | 0x0000CCCC33333333 |
| V1[13] | 0x0000DDDD22222222 |
| V1[14] | 0x0000EEEE11111111 |
| V1[15] | 0x0000FFFF00000000 |

According to the architecture specification this instruction has the following register field assignment:

mul.l vi,sj,vk

So, when the dispatch is made to the nvp from the nsp to start the instruction, we have the following data fields:

**disp_ep** - 0x137

**disp_ireg** - 0x1 ( refers to source operand vector register )

**disp_jreg** - 0x7 ( Scalar register for this operand is indicated by the rj field )

**disp_kreg** - 0x2 - ( result vector register - see Architecture reference )

**disp_spw_haz** - 1 - ( Indicates that this instruction may affect the psw )

All of this information is placed into the instruction queue in the vd subsection of logic where it awaits dispatch.

The next step is for the nsp to pass over the scalar data used in the instruction via an SXV transfer:

**data** - 0x0000000000000002

Now the sxv data is in the SXV queue in the nis block, and the dispatch information is in the dispatch queue in the vd block. When the instruction is at the front of the dispatch queue, the entrypoint ( in this case 0x137 ), is applied as an address to the vd dispatch rams. The vd microcode for this entrypoint is:

MUL(LW) VD_SXV REG_IK NOMASK PL4 OP=003H EP=UA_VOPS

This is broken into the following fields:

MUL(LW) - This sets F@M_DISPATCH == 1 ( indicates mfp dispatch ), FU ==2 ( indicates nmul function unit will be used ), F@DST_SIZE == F@SRC_SIZE == longword.

VD_SXV - Indicates that this instruction requires SXV data, and to wait for it if it is not in the queue.

REG_IK - Indicates that the instruction requires the use of vector registers I and K.

NOMAKS - Instruction is not under mask.

PL4 - Pipe length for this instruction is four clocks.

OP=003H - The opcode given to the nmul function units is 0x003

EP=UA_VOPS - Indicates that the entrypoint for the um microcode is the "vector op scalar" entrypoint, which is 0x04.

This information read from the vd rams is checked for hazards, and if none exist the instruction proceeds with the dispatch of the mfp, which is indicated with the assertion of **m_dispatch.** At this time, the um microcode rams are being addressed with the EP ( from the vd rams ) with 0b000 appended to the end. Thus, **m_uaddr** = 0x020.

Reads of the um rams produce the **mf_*** signals, which are directly generated from the um microcode ram outputs.

The um microinstructions for UA_VOPS, which is the um entrypoint for this instruction as determined by the vd microcode, are

0020: CNT(CLR) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

0021: VRF(X,SC) CNT(INC) PIPE(WR_VRF) LAST(RD,BD) TEST(DF_LE_2) DISP_IF;

The first microinstruction consists of the following constructs:

CNT(CLR) - Initializes the X,Y,Z and VM counts in the a_vm logic to 0. These are counters that keep track of which vector element we are working on ( Z is for results, X and Y are for input data ).

PIPE(LD_BD) - This sets up a "load backdoor" operation on the m_pipe_ctl field. This is done at the start of an operation to initialize the bd for data. This will cause mq_ld_bd to be asserted when it reaches the mq level; in addition, this will cause mq_first_write to be asserted on the following clock when actual result data is ready to be written.

TEST(VL_LE_1 ) - This tests the vl to see if is less than or equal to 1. If so, then the instruction will complete on the next microinstruction ( because of the DISP_IF construct.)

NEXT - This simply advances the micro-program counter to the next sequential microinstruction.

The second microinstruction is the one in which the actual operation is done. It consists of the following constructs:

VRF(X,SC) - Indicates that the data sources for the op0 will be the X data from the nvrf, and the source data for op1 will be scalar data.

CNT(INC) - Indicates that we will increment the element counter by one on this step.

PIPE(WR_VRF) - Results are written back into the nvrf when they are finished. ( i.e. at the backdoor level ).

LAST(RD,BD) - This will cause the M_LAST_RD ( indicating the last read of the register file ) and M_LAST_BD (indicating the last write occurring to the backdoor ) signals on the cycle following a successful test. The test condition is explained next. The rd_last signal is m_last_rd and occurs at level 0, while the bd last signal is mq_last_wr and occurs at level q.

TEST(DF_LE_2) - This will be a successful test if the difference between the counter and vl is less than or equal to 2. This would mean that the next microinstruction would be the last one.

Note that the default condition for microcode advancement in the function pipes is to stay at the current location until it is explicitly stated to do otherwise. Thus, in this case, the same microinstruction is repeated until the counter increases enough so that the test is successful and we dispatch the next macroinstruction.

On the clock following the dispatch, the first microinstruction ( 0x020 ) reaches level 0 The microinstruction at level 0 is shown as **m_upc**. Since this is the first microinstruction, **m_active** is asserted at this time and will remain so until all um microinstructions that make up the macroinstruction has passed through level one of the pipe. Also, at this time, the um ram is being read with the next microinstruction ( 0x021 ), and the fields for the microinstruction appear on the **mf_*** signals.

The next clock brings the first microinstruction ( 0x020 ) to level 1 and m1_active becomes 1. The second microinstruction ( 0x021, with counter == 0 ) reaches level 0. The signal **m_x_cnt**, which indicates what the element counter is at, is at 0x00.

Next, 0x020 reaches level 2, and ( 0x021, iteration 0 ) reaches level 1, and ( 0x021, iteration 1 ) is at level 0, as seen by the m_x_cnt == 1.

The next clock brings ( 0x020 ) to level 3, and ( 0x020, iteration 0 ) to 2, etc.

Finally, the first iteration of 0x021 ( with counter == 0 ) reaches level three. This is where the first multiply is begun. The operand data is on the a3 level data buses, and have the following values:

**m3_op0_dat** = 0x00000000FFFFFFFF

**m3_op1_dat** = 0x0000000000000002

**nmul_start_0**= 1

Since there are four nmul units on the mfp, then one is arbitrarily selected based upon a counter that resides in the mfc section of logic. Let's suppose that in our example that this counter is 0x0 when it is time to begin calculation of the first multiply. Then, on this clock

we would see **nmul_start_0** asserted, indicating that a multiply is beginning on the nmul_0 gate array.

The next clock brings ( 0x021, element 1 ) to level three, with

**m3_op0_dat** = 0x00001111EEEEEEEE

**m3_op1_dat** = 0x0000000000000002

**nmul_start_1** = 1

Then ( 0x021, element 2 ) reaches level 3, with

**m3_op0_dat** = 0x00002222DDDDDDDD

**m3_op1_dat** = 0x0000000000000002

**nmul_start_2** = 1

The next clock brings the first microinstruction ( 0x020 ) to the bd (q) level. Here, we get **mq_ld_bd** asserted as a result of the LD_BD in the pipe control construct in the microinstruction. Meanwhile, ( 0x21, element 3 ) has reached level 3, with the data

**m3_op0_dat** = 0x00003333CCCCCCCC

**m3_op1_dat** = 0x0000000000000002

**nmul_start_3** = 1

Also on this clock, nmul_0, which is processing element 0 of the vector, indicates that it will have a result ready on the next clock by asserting **nmul_rslt_next_0.** Because of the way that the multiply function pipe is designed, the pipe is always ready for multiply result data whenever the data is availiable. It is for this reason that the nmul pins **nmul_rslt_next** and **nmul_rslt_sel** are tied together for all nmul arrays on the mfp.

On the next clock, ( 0x021, element 0 ) reaches the bd level with the result data

**mq_rslt_dat** = 0x00000001FFFFFFFE

which comes from nmul0. This value is presented to the nvrf, where it is written into the vector register file. Meanwhile, ( 0x021, element 4 ) has reached level 3. Since nmul_0 finished with element 0 this cycle, it can now begin on element 4, which we see beginning with the assertion of **nmul_start_0.** Also, element 1 will be ready the next clock, as seen by the assertion of **nmul_rslt_next_1.**

One clock later, we have ( 0x021, element 1 ) at the q level, with our result data of

**mq_rslt_dat** = 0x00000001FFFFFFFE

Also, ( 0x21, element 5 ) is at the q level and begins calculation on nmul_2.

This process continues until we get to element 14. In this case, the test condition is true because the counter is 14, and vl ( 16 ) - counter (14) is two. Therefore, the next clock all test-dependent constructs will be valid on the following clock.

This means that when element 15 reaches level 0, we assert **m_clr_active** and **m_last_rd**, which indicate that the mfp level 0 will be inactive next cycle and that the current cycle is the last read of the register file. In addition, **m_rdy** will be asserted, indicating that a new instruction may be dispatched to the mfp on the next clock.

Finally, when ( 0x021, element 15 ) reaches level q, then **mq_last_wr** and **mq_last_elem** are asserted, indicating that this is the last write to the nvrf and that the instruction is complete.

**Example 2 - div.w.t operation**

This example will attempt to show the workings of an instruction under mask and at rate 2X. Rather than continue in the same level of detail as the previous example, I will here emphasize the differences that these two instruction features cause.

Suppose that the instruction that we want to execute is actually:

```
div.w.t         v1,v2,v7
```

Let's further suppose that vl = 0x10, and that vm is 0xF0C6. This would require that elements { 1, 2, 6, 7, 12, 13, 14, and 15 } would be affected by this operation. Assume register v1 has the following data:

```
V1[0] =     0x00000000
V1[2] =     0x00001111
V1[1] =     0x00004444
V1[3] =     0x00009999
V1[4] =     0x00011110
V1[5] =     0x0001aaa9
V1[6] =     0x00026664
V1[7] =     0x00034441
V1[8] =     0x00044440
V1[9] =     0x00056661
V1[10] =    0x0006aaa4
V1[11] =    0x00081109
V1[12] =    0x00099990
V1[13] =    0x000b4439
V1[14] =    0x000d1104
V1[15] =    0x000efff1
```

Also assume that each element of v2 has the value of its element number. For example, v2[0] = 0, v2[1] = 1, v2[2] = 2, etc.

The instruction begins ( from the perspective of the nvp ) with a dispatch from the nsp. For this instruction we have:

**disp_ep** - 0x7FA ( note MSB == 1, indicating a .t operation )

**disp_ireg** - 0x1 ( refers to source operand vector register v1)

**disp_jreg** - 0x2 ( Refers to the divisor in the operation - v2 )

**disp_kreg** - 0x7 - ( result vector register - v7 )

**disp_spw_haz** - 1 - ( Indicates that this instruction may affect the psw )

This instruction information is then inserted into the instruction queue in the vd section of logic, where it remains until it has moved to the front of the queue. At this time, the vd rams are read at the entrypoint address. The vd microcode for this instruction consists of the following constructs:

DIV(WD) REG_IJK MASK VM_ACC R2X PL4 OP=008H EP=UA_VOPV

These fields are decoded as follows:

DIV(WD) - Sets up dispatch to afp only, sets destination and source sizes to be word, and selects the ndivs as the appropriate FU to use.

REG_IJK - All three vector registers are requested ( REQ_VRI, REQ_VRJ, REQ_VRK ).

MASK - Indicates the instruction is under mask ( REQ_VM_RD == 1).

VM_ACC - Try to accelerate VM reads if possible ( REQ_VM_ACC == 1 ).

R2X - Operation occurs at rate 2X ( RATE_2X == 1 )

PL4 - Pipe length is four cycles.

OP=008H - Sets the function unit op to be 0x08.

EP=UA_VOPV - Sets up entrypoint to the ua microcode as 0x02. ( Note that this will be appended with 0b000 before used as an address to the ua rams ).

In addition to deciding when we can dispatch, the vd logic also determines whether we can "accelerate" the VM reads. This means that we skip over sections of the VM that do not indicate a meaningful operation. For example, if bits ( 0,1, 4, 5, 7, 0xA ) were the only set bits for a .t operation, then only element counts of 0x0 ( _vm == 3 ), 0x4 ( vm == 3 ), 0x6 ( vm == 1 ), and 0xA ( vm == 2) would be sent to the functional pipes for processing. Recall that this is a rate 2X operation, so our element counts are incremented by two and the two bit vm field indicates which of the two operands on the bus are valid.

If it is determined that we can proceed in accelerated mode, then a "1" is added to the entrypoint before it is appended with 0b000 to form the address to the ua rams. So, in our example, acceleration is possible, so our ua address will be ( 0x02 + 1 ) : 0b000 = 0x18. By examining the ua microcode listing, we see that this corresponds to the ua entrypoint UA_VOPV_ACC, which confirms our expectations.

The ua microcode for this instruction consists of the following microinstructions:

0018: CNT(CLR) NEXT;

0019: CNT(SCAN) NEXT;

001A: CNT(SCAN) PIPE(LD_BD) TEST(VL_LE_1) NEXT;

001B: VRF(X,Y) CNT(SCAN) PIPE(WR_VRF) LAST(RD,BD) TEST(DF_LE_2) DISP_IF;

The first instruction, 0x18, simply clears the counter that is used to keep track of which element we are currently processing.

The second microinstruction, 0x19, sets the element counters to the next set VM bit. This implements the VM acceleration. Note that this construct must precede the microinstruction on which the corresponding element is read by two cycles. Thus, the first scan is done at microinstruction 0x19, and the element is not actually used until instruction 0x1B.

The next microinstruction ( 0x1A ) initializes the backdoor controller, tests for ( vl <= 1 ) and does the next scan for the cycle occurring two clocks from now. These constructs are discussed in detail previously.

Finally, the last microinstruction, which is repeatedly done until the test condition terminates the instruction, is the same as the terminating instruction for the UA_VOPS microcode discussed previously, except that both operands come from the nvrf and the counter is scanned for the next set VM bit rather than simply incremented.

Recall that at time of dispatch, **a_dispatch** is asserted and the **a_uaddr** is applied to the ua rams. In this case the address is 0x18 ( the entrypoint ).

The next clock, **a_active** is asserted and **a_upc** contains the entrypoint 0x18. Meanwhile, 0x19 is at the **a_uaddr** stage.

This process proceeds in this manner until **a_upc** contains the first instance of 0x1B. Note that two cycles previous the initial SCAN of the counter was done, which should have produced a 0x0 since bit1 is set in the VM, and this is rate 2X operation, so the counter advances by 2 each time. Thus, for rate 2X operations, a counter value of 0x0 will actually refer to elements 0 and 1, with vm<1> corresponding to element 1 and vm<0> corresponding to element <0>.

Therefore, as a result of the first scan we have at level 0 a_x_cnt == a_y_cnt == 0x0. The value vm that corresponds to this register set is not visible until it reaches level 1.

On the next clock the first instance of microinstruction 0x1B has reached level 1, and we see **a1_vm** = 0x2, which indicates that element <1> is a valid operation while element <0> is invalid. Meanwhile, the counter for level 0 has advanced to 0x2.

The next clock has ( 0x1B, counter == 0x0 ) advanced to level 2, ( 0x1B, counter == 0x2 ) advanced to level 1 with **a1_vm** == 0x1, which indicates that element 2 should be operated upon, but not element 3. Meanwhile, ( 0x1B, counter == 6 ) has advanced to level 0.

The next clock brings advances information to the following state:

Level 0 : ( 0x1B, counter == 0xC )

Level 1: ( 0x1B, counter == 0x6), **a1_vm** == 0x3 ( elements 6 and 7 are valid )

Level 2: ( 0X1B, counter == 0x2), vm == 0x1 ( element 2 only is valid )

Level 3: ( 0x1B counter == 0x0 ), vm == 0x2 ( element 1 only is valid ), with the data buses **a3_op0_dat ==** 0x0000000000001111 and **a3_op0_dat ==** 0x0000000000000001.In general for rate 2X operations, the words for the even element ( element 0 in this case ) appear on the upper half of the buses and the odd element ( element 1 in this case ) appear on the lower half. However, since element 0 is not an element that will be affected by the instruction ( since its vm bit was not set ), the data on the upper half of the bus is meaningless. In this case, it matches what is in the registers only by coincidence. It would be 0's regardless of the value in the registers.

Let's assume for our example that the ndiv counter ( the one that selects which of the ndiv gate arrays on which to begin processing ) is at 2, so that the first two ndivs to get started are ndivs 2 and 3 ( recall that this is a rate 2X operation ). Thus we have **ndiv_start_2 == ndiv_start_3 ==** 1 for this clock. Note that the even numbered gate arrays are fed by the fu_vm<1> from the afc body of logic and the odd numbered ndivs are fed the fu_vm<0> bit from the afc. Thus, the odd-numbered ndivs perform operations on the even elements, and vice-versa. In this case, ndiv3 performs the division on the upper half of the bus ( element0 ) and ndiv2 performs the operation on the lower half of the bus ( element 1 ). So, strictly speaking, only ndiv2 performs an operation this clock as only its vm bit is set.

The next clock produces

Level 0 : ( 0x1B, counter == 0xE) ( elements 14 and 15 are valid )

Level 1: ( 0x1B, counter == 0xC), **a1_vm ==** 0x3 ( elements12 and13 are valid )

Level 2: ( 0X1B, counter == 0x6), vm == 0x3 ( elements 6 and 7 valid )

Level 3: ( 0x1B counter == 0x2 ), vm == 0x1 ( element 2 only is valid ), with the data buses **a3_op0_dat ==** 0x0000444400000000 and **a3_op0_dat ==** 0x0000000200000000. Again, note that for "invalid" elements there are zeroes on the half of the bus corresponding to it, and the "valid" elements' data appears appropriately, in this case on the upper half since the valid element number is even ( 0x2 ). For these elements, we have **ndiv_start_4 == ndiv_start_5 ==** 1. Note that only ndiv5 will perform a useful operation as its vm bit is set whereas ndiv4 has its vm bit set to 0.

Level q: 0x18 - No effect. **aq_active ==** 1.

Another clock:

Level 0 : Inactive

Level 1: ( 0x1B, counter == 0xE), **a1_vm ==** 0x3 ( elements14 and15 are valid ) **a1_last_elem ==** 1.

Level 2: ( 0X1B, counter == 0xC), vm == 0x3 ( elements 12 and13 valid )

Level 3: ( 0x1B counter == 0x6 ), vm == 0x3 ( element 6 and 7 are valid ), with the data buses **a3_op0_dat ==** 0x0002666400034441 and **a3_op1_dat ==** 0x0000000600000007. Since both elements are affected, each of the vm bits for this stage are set. The next ndivs to be used are numbers 0 and 1, so **ndiv_start_0 == ndiv_start_1 ==** 1.

Level q: 0x19 - No action

Clock:

Level 0 : Inactive

Level 1 : Inactive

Level 2: ( 0X1B, counter == 0xE), vm == 0x3 ( elements 14 and15 valid )

Level 3: ( 0x1B counter == 0xC), vm == 0x3 ( elements 12 and 13 are valid ), with the data buses **a3_op0_dat ==** 0x00099990000B4439 and **a3_op1_dat ==** 0x0000000C0000000D. Since both elements are affected, each of the vm bits for this stage are set. Ndivs 2 and 3 are finishing up their first operations this cycle ( see the level q discussion for this clock ), so they will be started next clock as indicated by **ndiv_start_2 == ndiv_start_3 ==** 1.

Level q : 0x1A. This microinstruction does the initialization of the backdoor controller, as indicated by **aq_ld_bd ==** 1. Since the next clock will be 4 clocks since the first operation reached level 3, then we should expect a result on the next clock. ( The problems with this assumption will be explained shortly. ) As such, the signals **ndiv_rslt_sel_0** and **ndiv_rslt_sel_3** are asserted in anticipation of having data ready for level q next cycle. If, however, the corresponding signals **ndiv_rslt_next_2** and **ndiv_rslt_next_3** do not indicate that the result will not be ready for the next cycle, then all stages must be held until the these signals are asserted. This is done by setting **ck_xtnd** to 1, which will hold all of the stages of the pipe but will allow the ndivs to continue to calculate. When we have stalled long enough for the ndivs' **rslt_next** outputs to be 1, then the ck_xtnd is de-asserted on the next cycle, and then the stages progress normally ( the stages move on the clock following the de-assertion of **ck_xtnd.)**

Clock:

Levels 0-2: Inactive. At this point **a_rdy ==** 1 which indicates that the afp may accept the dispatch of a new instruction at this time.

Level 3: (0x1B, counter == 0xE), vm ==3 ( 14 and 15 valid ). **ndiv_start_4 == ndiv_start_5 ==** 1. **a3_op0_dat ==** 0x000D1104000EFFF1 and **a3_op1_dat ==** 0x0000000E0000000F. **ndiv_start_4 == ndiv_start_5 ==** 1.

Level q : ( 0x1B, counter == 0x0 ). Since this is four cycles ( the pipe length ) since ( 0x1B, counter == 0x0 ) was at level 3 ( ignoring clocks that occurred during a clock extend ), we should expect this instruction to reach level q on this clock. As such, we see that **aq_rslt_dat =** 0x0000000000001111, **aq_vm ==** 0x2 and **aq_z_cnt ==** 0x0. This shows that element 1 will be written with the lower half of the bus and element 0 will not be affected. Meanwhile, the selection for the next clock's results is being done with **ndiv_rslt_sel4 == ndiv_rslt_sel5 ==** 1.

Clock:

Level 0-3: Inactive

Level q: ( 0x1B counter == 0x2 ). **aq_rslt_dat ==** 0x0000222200000000, **aq_vm ==** 1, **aq_z_cnt ==** 0x2. This indicates that the even element only ( element 2 ) is written with the upper half of the bus.

Clock:

Level q: ( 0x1B counter == 0x6 ). vm == 3, cnt == 0x6, rslt_dat = 0x0000666600007777. Element 6 is written with the upper half, element 7 is written with the lower half.

Clock:

Level q: ( 0x1B counter == 0xC ). vm == 3, cnt == 0xC, rslt_dat = 0x0000C-CCC0000DDDD. Element 12 is written with the upper half, element 13 is written with the lower half.

Clock:

Level q: ( 0x1B counter == 0xE ). vm == 3, cnt == 0xE, rslt_dat = 0x0000E-EEE0000DDDD. Element 14 is written with the upper half, element 15 is written with the lower half. Also, since this is the last microinstruction of the sequence, **aq_last_elem == aq_last_wr ==** 1.

This completes the execution of the instruction.

# 4.0 Scalar Processor Primer

This section of the overview will be a ( very ) brief introduction of some basic instruction types on the scalar processor. The topics discussed here are done in a very broad manner; for a more detailed analysis of the nsp instruction implementation, consult the nsp board specification.

It would be helpful to have the nsp block diagram and the us microcode specification handy as references while reading this section. In particular, the nomenclature in this document for the various interfaces between the sub-blocks conforms to the standard that is set in the introductory section of the nsp board spec.

## 4.1 Basic blocks

The nsp is further sub-divided into several blocks of logic.

### 4.1.1 The NDC - Neptune Data Cache

The ndc is actually the complete memory interface in addition to housing the data cache. All crossbar interface logic is contained within the ndc. The parts that define the ndc are:

ndc_ga - Gate array that is mostly control of the crossbar interface.

nag_ga (2) - (Neptune address generation ) - These two gate arrays generate logical addresses for memory operations.

ndp_ga (2) - ( Neptune Data Path ) - These gate arrays route data to and from the vector processor and crossbar system.

npa_ga (2) - (Neptune Physical Address ) - Generate physical addresses for the crossbar system.

dc rams - Data cache data, tag, and status rams.

pte rams -PTE cache data, tag, and status rams.

In addition, there are some discrete parts in the ndc block.

### 4.1.2 The NAS - Neptune Address and Scalar block

The nas contains the central processing engine of the nsp. It contains the following parts:

wcs rams - Writable control store rams - contains the microsequencer microcode (us).

nus_ga - microsequencer gate array - Contains and controls the various stages of the microsequencer.

nrfa complex - 4 nrfa gate arrays that make up the register file and central ALU.

npsw_ga - Neptune processor status gate array - Houses and manages the various psw bits.

sr rams - Rams that contain scratch ram data and valid bits.

In addition the, the NAS contains some discrete ECLIPS parts not included in this list.

### 4.1.3 NRC - Neptune Return Controller

The NRC controls and routes return data from memory to the appropriate place on the nsp by means of a queue of destination information that is stored whenever a memory read request is made. It consists entirely of three nrc_gas ( Neptune return controller gate arrays ), each of which handle one slice of the returning memory data.

### 4.1.4 NIP - Neptune Instruction Processor

The NIP fetches instructions from memory and maintains them in the instruction cache. It parses information from the cache and dispatches entrypoints to the NAS. The following components make up the NIP:

npar_ga - Neptune Parser gate array - Parses instruction information from the cache and produces dispatch information for the NAS.

niad_ga (2) - These two gate arrays generate logical addresses for instruction fetches from memory. Dispatch information for the nvp is also routed through these gate arrays. Each gate array contains a slice of the full data.

branch history cache - This is a set of rams that maintain the branch history cache.

The NIP also contains some discrete ECLIPS parts.

## 4.2 Instruction sequence

This section will briefly go over the sequence of events that occur when an instruction is executed on the scalar processor.

SPELL CHECKED TO HERE

### 4.2.1 Instruction Fetch/Dispatch

The first thing that occurs in the life of an instruction execution is the fetching, parsing, and dispatching of the instruction. All of this is done by the NIP block of logic.

The NIP attempts to get the instruction at location PC from the instruction cache. If it can not, it must make a request to the ndc memory interface via an IXA request for that address. The instruction is returned to the cache via an MXI handshake. The mechanics of a memory request to the NDC subsystem will be covered later in this document; for now, just assume some latency between the IXA handshake and the corresponding MXI return.

When the memory data at address PC has been encached, then this instruction word is read and parsed by the NPAR gate arrays. The parsed instruction information is then placed into a queue of dispatch information in the NPAR gate array, where it waits for its turn to be dispatched to the NAS.

When this information reaches the front of the queue, it is dispatched to the NAS via an AS_DISP handshake.

### 4.2.2 NAS activity

When instruction information is dispatched to the NAS, it enters into the microsequencer pipe. A brief discussion of the architecture of this pipe is in order before I continue tracing the path of the instruction.

The NAS microsequncer is a three stage pipe. The first stage, known as the UPC ( Micro-Program Counter ) level, is the stageat which the us microcode rams are read. The UPC level should always match the internal registers of the self-timed rams that make up the us microcode. Note that this level may be fed by branch addresses in the current microinstruction, the micro-interrupt vector, the currently dispatching instruction, the micro-address in the UPC overrun register, the top of the micro-pc stack, the current UPC plus 1, or the current UPC.

Recall that only the entrypoint ( the initial microinstruction for a macroinstruction ) is dispatched from the NIP. If more than one microinstruction make up the macroinstruction, then a source other than the dispatching EP is selected for the next UPC. The only time a

new ep is taken for the UPC is when the previous microinstruction has reached a dispatch condition. See the NAS section of the nsp board specification for more details.

The next stage following the UPC stage is the UIR1 ( Micro-Instruction Register - level 1 ) stage. If there are no hazards at the UIR1 level on cycle X, then on cycle X + 1 the UPC of cycle X will be at the UIR1 level.

The UIR1 level is used to check for hazards and wait for cancellation via branch restart. If a hazard exists, then the microinstruction will sit at the UIR1 stage until the hazard is resolved. Also, if a branch restart occurs ( i.e., the NIP dispatched the wring path for a branch ) it occurs while the microinstruction is at this level.

Whenever the instruction at UIR1 has no hazards and will progress to the next level on the next clock, then the data for any ALU or functional unit instruction should be present on the appropriate buses.

The final level of the pipe is the UIR2 level. Any instruction that makes it to this level is guaranteed to run to completion. This level begins execution of the instruction.

#### 4.2.2.1 Intra-Nrfa ALU operations

The simplest integer instructions are executed by the on-board ALU of the nrfa gate arrays. These include