FIELER

C38xx Scalar Processor Specification

Document No. 416-001247-000

Revision 1.1
August 28, 1991

INTERNAL USE ONLY

PROPRIETARY DISCLAIMER

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Physical Description

A Neptune processor consists of two boards, a Neptune Scalar Processor board (NSP), and a Neptune Vector Processor board (NVP). The NSP board initiates execution of all instructions. Only if an instruction is a vector instruction is the NVP dispatched. The NSP board has two primary interfaces, the cross bar, and the vector processor. All vector instruction memory accesses pass through the NSP board.

The NSP board is installed in a CPU/memory bay. There are up to four CPU/memory bays in a C38xx system, with each bay containing up to two NSP/NVP CPU board pairs.

The NSP board assembly consists of the NSP logic board and the NSP power board. The power board receives 300 volts DC and transforms it in to -2.0, -4.5, -5.0, and +5.0 volts DC.

1.2 Major Subsystems

The functionality of the NSP board is subdivided into four major subsystems: Neptune Address and Scalar (NAS) subsystem, Neptune Instruction Processor (NIP) subsystem, Neptune Return Control (NRC) subsystem, and Neptune Data Cache (NDC) subsystem. These four subsystems operate independently from one another with well defined interfaces. The basic functionality of each of the four subsystems is presented below. A detailed description of each subsystem and the interconnecting interfaces is described in the later chapters.

The NAS subsystem is a micro code controlled instruction execution unit. The subsystem is dispatched decoded instructions (from the NIP subsystem) and executes micro words stored in control store to perform the required operations. All memory operations are sent to the NDC subsystem. All vector instructions are dispatched to the NAS subsystem and then passed on to the vector processor board. The NAS subsystem is implemented with seven types of gate array/ custom devices: Neptune micro sequencer (NUS), Neptune Register File and ALU (NRFA), Neptune Processor Status Word (NPSW), Neptune Floating Point Adder (NFAD), Neptune Multiplier (NMUL), Neptune Divider (NDIV), and Neptune Miscellaneous Function Unit (NMISC). In addition to the gate arrays/custom devices, 39 rams are used for the control store and scratch ram memory, and a few ECLinPS parts are used for external data path and control.

The NIP subsystem is a cache based instruction processor. Data from memory is written unmodified into the cache. The data is read from the cache and parsed into individual instructions. The parsed instructions are transferred to the NAS subsystem for execution. The NIP subsystem has a seven deep queue for parsed instructions so the NIP subsystem can continue parsing instructions even when the NAS subsystem is unable to accept them. The NIP subsystem is implemented with two types of gate arrays: Neptune Instruction Address (NIAD), and Neptune Parser (NPAR). Twenty rams are used to implement the instruction cache and a few discrete ECLinPS parts are used to implement functionality that didn't fit in the gate arrays.

The NDC subsystem is responsible for arbitrating the priority of memory requests, accessing the data and PTE caches, and issuing requests to the cross bar for memory/communication accesses. The NAS subsystem issues memory requests which are needed to execute instructions, and the NIP subsystem issues memory requests for instruction cache look ahead data. In addition to the requests from the NAS and NIP the NDC subsystem also generates

requests for data cache block prefetch data and requests for vector address generation. The NDC subsystem contains four gate array types: Neptune Data Cache control (NDC), Neptune Data Path (NDP), Neptune Address Generation (NAG), and Neptune Physical Address (NPA). Forty-two rams are used for the data and PTE caches with ECLinPS parts for address staging and control.

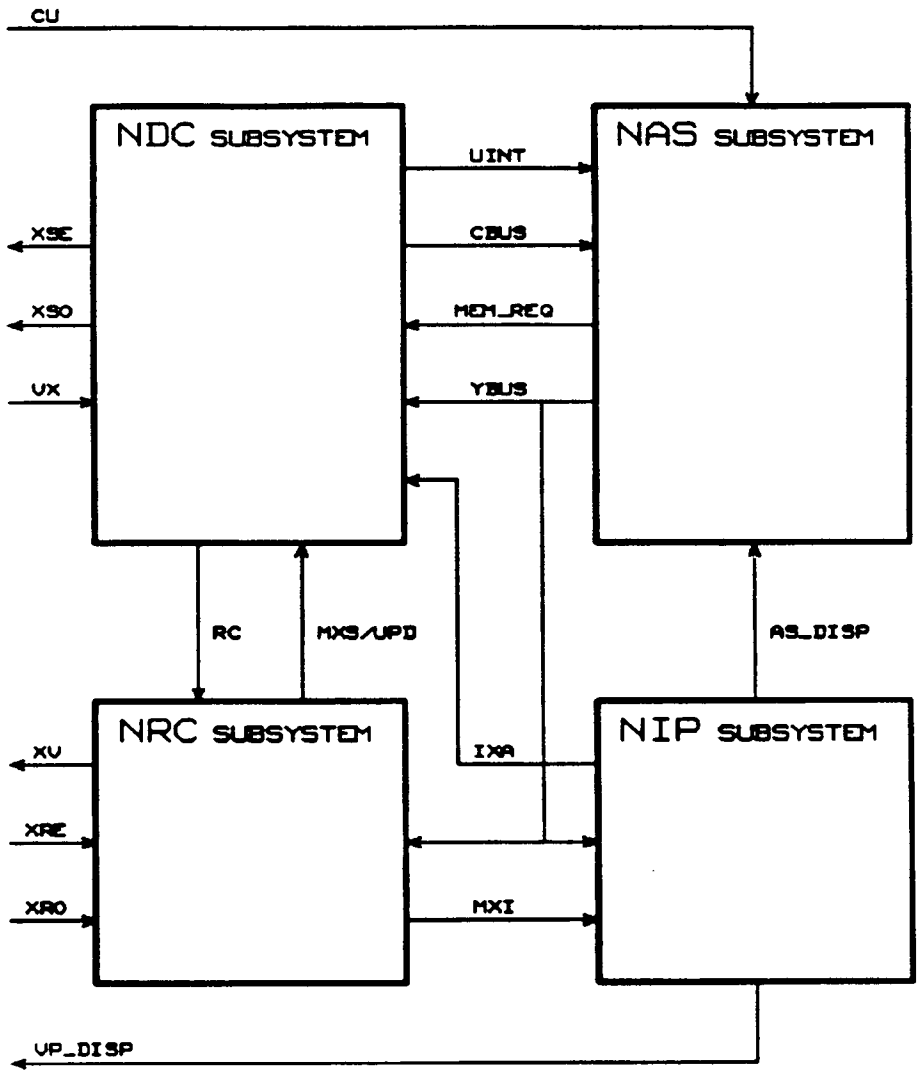
The NRC subsystem queues control information for read requests which have been sent to the cross bar for memory/communication access. When the requested data is received from the cross bar to the NSP board, it is matched up with the appropriate queued control information and transferred to the destination subsystem. The NRC subsystem is implemented with a single gate array type, Neptune Return Control gate array (NRC).

2 NSP Board Interfaces

The NSP board interfaces are presented in this chapter. The chapter starts by describing the interfaces to other boards, followed by interfaces between the major subsystems on the board.

Figure 2-1 shows the interfaces which are described below. The cold start process will be described to illustrate the use of these interfaces. The first step of the cold start process is to scan initialize all registers and rams. A register within the NAS subsystem is scan initialized with the starting address of an instruction to be executed and a starting micro accress. Once the scan control is changed to normal mode, the NAS subsystem begins executing micro code and transfers the starting address to the NIP subsystem using the YBUS.

Figure 2-1 NSP Board Interfaces



The NIP subsystem access the instruction cache and finds invalid data. To obtain the instruction data the instruction address is transferred to the NDC subsystem using the IXA interface. The

NDC subsystem accesses the PTE cache to translate the logical instruction address to a physical memory address. Again, since this is the first access to that page of memory the PTE cache is invalid. The NDC subsystem interrupts the NAS subsystem (from an idle state) using the UINT interface. The UINT interface informs the NAS that a PTE miss occurred. The NAS subsystem executes micro code to bring in the required page table entry. To do so the NAS makes a memory request to the NDC subsystem for the first level PTE using the MEM_REQ interface. The NDC subsystem processes the PTE request which results in sending a memory request to the cross bar using either the XSE or XSO interface based on whether the request is for the even/odd side of memory. Additionally, the NDC subsystem uses the RC interface to the NRC subsystem to transfer the memory request destination information. Eventually the memory request is returned on either the XRE or XRO read data return busses. The NRC subsystem uses the destination information to know to transfer the return data to the NDC subsystem using the MXS interface. The NDC subsystem forwards the data to the NAS subsystem using the CBUS interface.

The NAS subsystem continues executing micro code and issues a PTE request (using the MEM_REQ interface) to the NDC subsystem to obtain the second level PTE data. The NDC subsystem processes the request and issues either a XSE or XSO memory request as well as sending the destination information to the NRC over the RC interface. As with the first level PTE data the returned data is received over either the XRE or XRO interfaces. The NRC subsystem transfers the data to the NDC subsystem over the MXS interface and the NDC forwards the data to the NAS over the CBUS. Finally, the NAS subsystem completes the micro coded interrupt routine by sending the second level PTE data over the YBUS to the NDC subsystem to be written to the PTE cache.

Now that the PTE cache has the needed page table entry, the instruction look ahead request is retried. The NDC subsystem completes the instruction look ahead request by issuing a memory read request using both XSE and XSO interfaces. As with the PTE requests, the read request destination information is transferred to the NRC subsystem using the RC interface. When the look ahead data is received over the XRE and XRO interfaces it is transferred to the NIP subsystem using the MXI interface. The NIP subsystem writes the look ahead data into the instruction cache. The NIP reaccesses the instruction cache and finds the required location valid. The data read from the cache is parsed, decoded, and sent to the NAS using the AS_DISP interface. The NAS uses the transferred control store entry point to start executing micro code to complete the first instruction.

2.1 External Interfaces

The NSP board transfers data to/from the cross bar and the vector processor. Each of the external interfaces are described below.

2.1.1 XSE - Even Cross Bar Send Interface

The even cross bar interface is used to send even side memory/communication requests from the NSP board to the cross bar. Requests for both memory data and communication data are transferred over the interface. The signals which make up the interface are listed in Table 2-1.

Table 2-1 Even Cross Bar Interface Signals

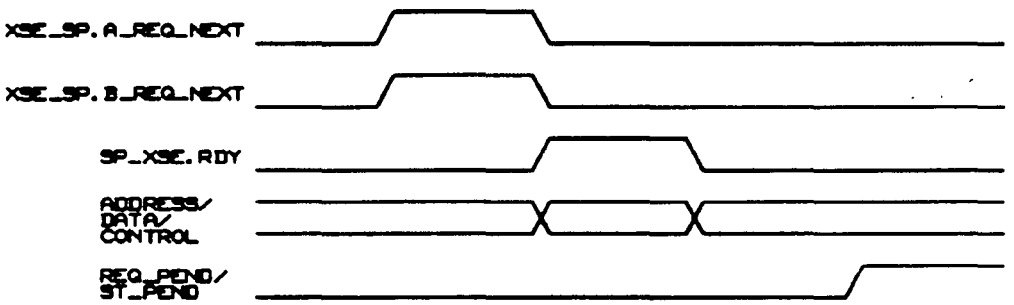
SP_XSE.ADDR<28..3>	Address for request
SP_XSE.BD_SEL<3..0>	Memory/Communication board select
SP_XSE.CTL_PAR<4..0>	Parity for address and control signals

SP_XSE.CYCLE<1..0>	Type of request
SP_XSE.RDY	Handshake signal to cross bar
SP_XSE.WR_DATA<31..0>	Write data bus
SP_XSE.WR_PAR<3..0>	Byte parity for write data bus
SP_XSE.WR_ZONE<3..0>	Byte write enables
XSE_SP.A_REQ_NEXT	Handshake signal from cross bar
XSE_SP.A_REQ_PEND	Request pending signal
XSE_SP.A_ST_PEND	Store request pending signal
XSE_SP.B_REQ_NEXT	Handshake signal from cross bar
XSE_SP.B_REQ_PEND	Request pending signal
XSE_SP.B_ST_PEND	Store request pending signal

The signal SP_XSE.CTL_PAR is parity for signals SP_XSE.ADDR, SP_XSE.CYCLE, and SP_XSE.WR_ZONE (refer to the signal list appendix for specific parity bit assignment).

Figure 2-2 illustrates the handshake mechanism used for the send interface.

Figure 2-2 Even Cross Bar Send Interface



A transfer is completed when the RDY signal is asserted and both REQ_NEXT signals were asserted on the previous cycle. Address, data, and control are valid on the cycle which the RDY signal is asserted. The REQ_PEND and ST_PEND (for writes) signals are asserted two cycles after the RDY signal and will remain asserted until the request has won arbitration through the cross bar.

Communication requests are special in that they always are to both even and odd sides of the cross bar. Furthermore the even and odd side requests must travel through the cross bar and arrive at the communication board on the same cycle. To guarantee these constraints, a communication request is held at the scalar processor until the REQ_PEND signals for both even and odd sides are deasserted. An exception is made in that after one communication request is issued additional communication requests can be issued without waiting for REQ_PEND to be deasserted.

2.1.2 XRE - Even Cross Bar Receive Interface

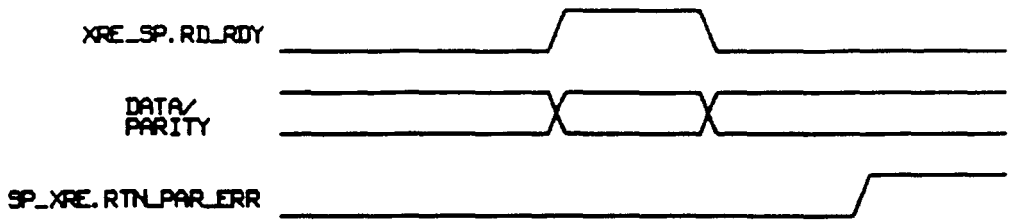
The even cross bar receive interface is used to receive memory/communication requests from the cross bar. The signals which are included in the interface are listed in Table 2-2.

Table 2-2 Even Cross Bar Receive Interface Signals

SP_XRE.RTN_PAR_ERR	Return data parity error
XRE_SP.RD_DATA<31..0>	Read data bus
XRE_SP.RD_PAR<3..0>	Byte parity for read data bus
XRE_SP.RD_RDY	Handshake signal for read data

Figure 2-3 illustrates the handshake mechanism used for the receive interface.

Figure 2-3 Even Cross Bar Receive Interface



The NSP board must be able to accept the return read data from all outstanding read requests. This requirement allows the cross bar to eliminate return data queueing. The NRC subsystem receives the returned data and has sufficient queueing to receive all outstanding data. A transfer is completed when the RD_RDY signal is asserted. On the same cycle as the RD_RDY the data and parity are valid. The NRC subsystem will assert the RTN_PAR_ERR signal two cycles after data with bad parity is received.

2.1.3 XSO - Odd Cross Bar Send Interface

The odd cross bar send interface is used to send memory/communication requests from the NSP board to the cross bar. The XSO interface is identical to the XSE interface described above, refer to that section for additional information.

2.1.4 XRO - Odd Cross Bar Receive Interface

The odd cross bar receive interface is used to receive memory/communication read data. The XRO interface is identical to the XRE interface described above, refer to that section for additional information.

2.1.5 Communication Board Interface

These signals are used to interface with the Neptune Communication Unit (NCU) board. Many of the signals work independently to transfer information between the NSP and NCU boards. The signals which are included in the interface are listed below.

Table 2-3 Communication Board Interface Signals

SP_XC.DEADLOCK	Deadlocked instruction indicator
SP_XC.TRAP_COMP	Micro trap local complete signal
XC_SP.CU_STATUS	Communication request status
XC_SP.CU_STATUS_EN	Communication request status enable
XC_SP.MT_COMP	Micro trap system wide complete signal

XC_SP.TRAP_RDY	Micro trap ready
XC_SP.TRAP_TYPE<3..0>	Micro trap type
XC_SP.TRAP_VECT<11..0>	Supplemental information for micro trap

The signal XC_SP.CU_STATUS_EN is asserted to inform the NAS subsystem that status from a communication board request is being transferred. The signal XC_SP.CU_STATUS is valid the cycle XC_SP.CU_STATUS_EN is asserted and contains the returned status. An example of a communication request in which status is returned is checking if a communication register is locked.

The signal XC_SP.TRAP_RDY is asserted for a single cycle indicating that a trap is being sent to the NSP board. On the same cycle as the TRAP_RDY signal the XC_SP.TRAP_TYPE and XC_SP.TRAP_VECT signals are valid and contain the trap type information. Once the NAS subsystem has completed processing the trap the signal SP_XC.TRAP_COMP is asserted for one cycle. When all processors executing for the same process complete the trap the signal XC_SP.MT_COMP is sent to the originating processor to inform it of the traps completion.

The signal SP_XC.DEADLOCK is asserted when a two instruction loop is being executed in which one of the instructions is a branch and the other is a deadlock detectable instruction. The deadlock detectable instructions are used to wait on a write to either memory or the communication registers from another processor (i.e. TAS, TAC, ...). When all processors executing on the same process have this signal asserted a deadlock trap is sent to each of the processors.

2.1.6 Data to the Vector Processor

This interface is used to transfer data from memory or the scalar processor to the vector processor. The NRC subsystem sources the data to be transferred. Table 2-4 lists the signals for the interface.

Table 2-4 Data to Vector Processor Interface Signals

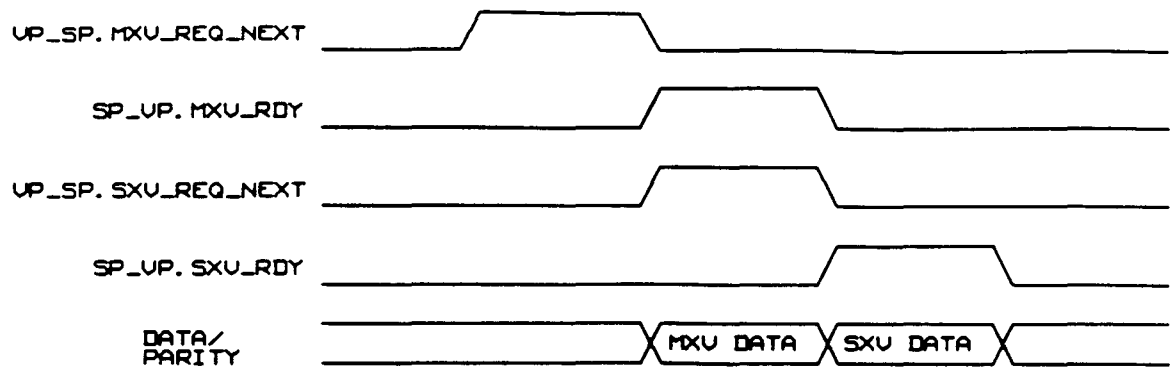
SP_VP.DATA<63..0>	Data Bus
SP_VP.PAR<7..0>	Byte parity for data bus
SP_VP.MXV_RDY	Handshake signal for memory data
SP_VP.SXV_RDY	Handshake signal for scalar data
VP_SP.MXV_REQ_NEXT	Handshake signal for memory data
VP_SP.SXV_REQ_NEXT	Handshake signal for scalar data

For memory transfers the NRC subsystem receives data from the cross bar using the XRE and/or XRO interfaces. The NRC subsystem uses queued destination information to rotate the data for proper alignment and then forwards the data to the vector processor.

Scalar transfers are used to send to the vector processor data from the NAS subsystem register file. The NAS subsystem reads the register file and transfers the data using the YBUS to the NRC subsystem. The NRC subsystem stages the YBUS and then transfers the data to the vector processor. Scalar data transfers have priority over memory data transfers.

Figure 2-4 illustrates the handshake mechanism used for the interface. The REQ_NEXT signals are used to specify that the receiving logic can accept a data transfer. Normally both REQ_NEXT signals will be asserted, although on any cycle neither, one, or both REQ_NEXT signals can be

Figure 2-4 Data to Vector Processor Interface



asserted. A transfer is completed when a RDY signal is asserted and the corresponding REQ_NEXT signal was asserted on the previous cycle.

2.1.7 Data from the Vector Processor

The interface is used to transfer data from the vector processor to either memory or the register file within the NRFA gate arrays of the NAS subsystem. The NDP gate arrays of the NDC subsystem receive the data and pass it on to either memory using XSE and XSO interfaces or transfer it to the NAS subsystem using the CBUS. Table 2-5 lists the signals for the interface.

Table 2-5 Data from Vector Processor Interface Signals

SP_VP.VX_REQ_NEXT	Handshake signal from NSP board
VP_SP.DATA<63..0>	Data Bus
VP_SP.PAR<7..0>	Byte parity for data bus
VP_SP.VXA_ADDR<31..0>	Address Bus
VP_SP.VXA_PAR<3..0>	Byte parity for address bus
VP_SP.VXA_LAST	Last transfer indicator
VP_SP.VXA_VM<1..0>	Vector Mask information
VP_SP.VXM_RDY	Vector to memory ready handshake signal
VP_SP.VXS_RDY	Vector to scalar ready handshake signal

On the NSP board both types of transfers are placed in the same queue. A common REQ_NEXT signal is used for both types of transfers indicating that there is space in the queue for data on the following cycle. The vector processor knows the destination for the data and will assert only one of the two RDY signals for each transfer.

Memory transfers consist of both data and address information. The data fields, VP_SP.DATA and VP_SP.PAR, are queued on the NDP (data path) gate arrays until the logical address has been generated and translated to a physical address. The address fields, VP_SP.VXA_ADDR and VP_SP.VXA_PAR, are queued by the NAG (address generator) gate arrays. The control fields, VP_SP.VXA_LAST and VP_SP.VXA_VM<1..0>, are queued by the NDC (data cache control) gate arrays. All vector memory accesses are treated as a series of transfers with the last transfer marked with the assertion of VP_SP.VXA_LAST.

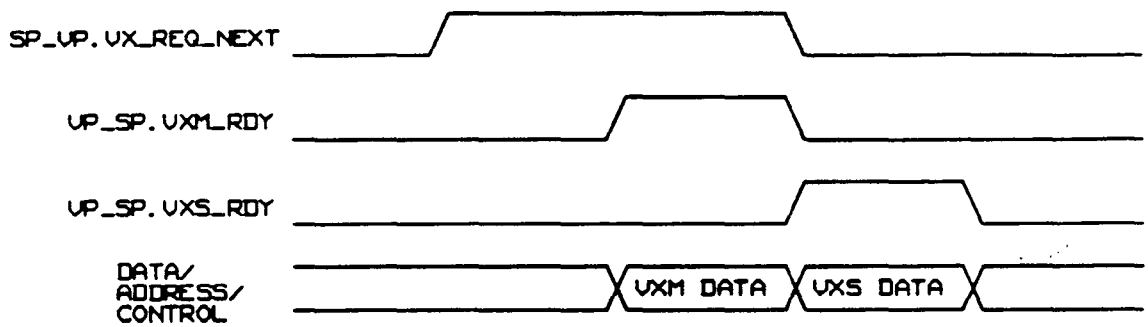
The various vector load/store requests are accomplished by using the interface fields differently.

The only vector instruction which makes requests to memory that does not use this interface is a simple vector load (not under mask). If an instruction is for long word writes or uses vector index for address generation, then a single logical address is generated per transfer. Otherwise, the interface operates at a 2x rate and two logical addresses are generated per transfer. For indexed operations the VXA_ADDR field is sourced from a vector register, non-indexed operations source the product of the register element number and the vector stride register. Under mask operations use the VXA_VM bits to specify if a request is to be issued to memory. Normal operations force the VXA_VM bits to one so that all operations go to memory.

Scalar transfers only use the DATA and PAR fields. The NDC subsystem uses queued destination information to transfer the data using the CBUS to the register file of the NAS subsystem.

Figure 2-5 shows the handshake mechanism used for the interface. Normally the

Figure 2-5 Data from Vector Process Interface



VX_REQ_NEXT signal will be asserted indicating that the queues can accept a transfer. A transfer is completed when a ready signal is asserted and the VX_REQ_NEXT signal was asserted on the previous cycle.

2.1.8 Vector Processor Instruction dispatch

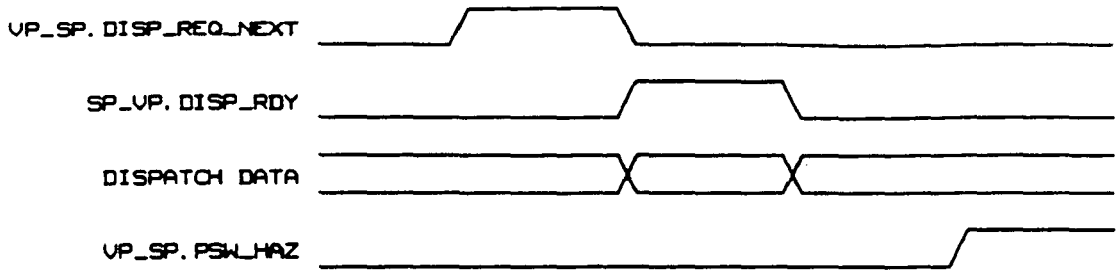
Vector instructions are transferred to the NVP board by this interface. The NIP subsystem on the NSP board parses and stages the instructions until they are transferred to the vector processor. Table 2-6 lists the signals of the interface.

Table 2-6 Vector Processor Instruction Dispatch Interface Signals

SP_VP.DISP_EP<10..0>	Vector instruction entry point
SP_VP.DISP_IREG<2..0>	Vector instruction I register field
SP_VP.DISP_JREG<2..0>	Vector instruction J register field
SP_VP.DISP_KREG<2..0>	Vector instruction K register field
SP_VP.DISP_PSW_HAZ	Vector instruction PSW hazard signal (from NAS)
SP_VP.DISP_RDY	Handshake signal from NSP board
VP_SP.DISP_REQ_NEXT	Handshake signal from vector processor
VP_SP.PSW_HAZ	Processor status word hazard signal

Figure 2-6 illustrates the handshake mechanism used for the transfer. A transfer is complete when the DISP_RDY signal is asserted and the DISP_REQ_NEXT was asserted on the previous

Figure 2-6 VP Instruction Dispatch Interface



cycle. The signal SP_VP.DISP_PSW_HAZ is used under micro code control to specify that the instruction being dispatched may modify the processor status word register. When an instruction is dispatched and the DISP_PSW_HAZ signal is asserted the vector processor will assert VP_SP.PSW_HAZ two cycles after the transfer. The signal will remain asserted until that instruction has completed execution. The NAS subsystem uses the PSW_HAZ signal to hold reading the PSW register until all hazards have been cleared.

2.2 Internal Interfaces

The following sections describe the interfaces which exist between the major subsystems within the NSP board. Figure 2-1 illustrates the interconnection of these interfaces.

2.2.1 AS_DISP - Instruction dispatch interface

This interface is used to transfer instructions that were parsed by the NIP subsystem to the NAS subsystem. The NAS subsystem is a micro code controlled, pipelined subsystem. The information transferred from the NIP to the NAS subsystem is used to start the micro sequencer to execute the instruction. The list of signals which are part of the interface are listed below.

Table 2-7 AS_DISP Instruction Dispatch Interface Signals

AS_DISP_BR_DISPL<7..0>	Branch displacement field
AS_DISP_DISPL<31..0>	Instruction displacement field
AS_DISP_DISPL_PAR<3..0>	Byte parity for instruction displacement
AS_DISP_EP<9..0>	Instruction entry point
AS_DISP_IREG<2..0>	Instruction I register field
AS_DISP_JREG<2..0>	Instruction J register field
AS_DISP_KREG<2..0>	Instruction K register field
AS_DISP_RDY	NAS subsystem handshake signal
AS_DISP_REQ	NIP subsystem handshake signal
AS_DISP_SEL_IREG	K register field control signal
AS_DISP_SIZE<1..0>	Instruction size in halfwords
AS_DISP_XTEND	Extended instruction space indicator
BR_RESTART	Branch restart signal from NAS to NIP
UPC_BR_POL	Branch instruction condition polarity
UPC_BR_SEL<1..0>	Branch instruction condition
UPC_CPC<31..0>	Instruction address

UPC_DL	Instruction deadlock indicator

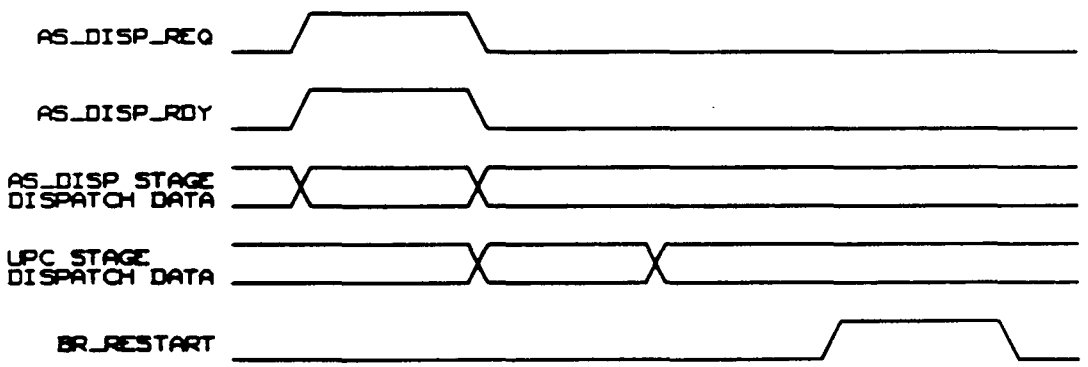
The handshake signals for the interface, AS_DISP_RDY and AS_DISP_REQ, are associated with the AS_DISP pipeline stage. Most of the information transferred is also associated with the AS_DISP pipeline stage. However some data is transferred at the next stage of the pipe (the UPC stage). The same signals which control the NAS pipeline stages also control the corresponding NIP pipeline stages. It was a matter of timing that some fields are at the AS_DISP stage and others are at the UPC stage.

Branch instructions are not transferred as instructions but rather as a tag on the target instruction of the branch. The UPC_BR_SEL field specifies the branch tag type: scalar carry, address carry, or unconditional branch. The UPC_BR_POL signal specifies the polarity for the branch condition. The NIP subsystem makes an assumption as to whether a branch instruction will cause a branch or will execute sequentially. The NIP continues to parse instructions after a branch assumption is made. The UPC_BR_SEL and UPC_BR_POL allow the NAS subsystem to check whether the branch assumption made by the NIP was correct. If the branch assumption is correct then the tagged instruction is executed. Otherwise, the NAS asserts BR_RESTART and the tagged instruction as well as all instruction parsed after the incorrect branch are marked invalid and the NIP is restarted following the correct target for the branch instruction.

The address for the dispatch, UPC_CPC, is the address of the tagged branch instruction. The address for instructions which have a 'nop' for a branch tag is the address of the dispatched instruction. The NAS subsystem requires the address for the tagged branch instruction, the dispatched instruction, and the instruction after the dispatched instruction. The AS_DISP_BR_DISPL, AS_DISP_SIZE, AS_DISP_XTEND, and UPC_CPC fields are used by the NPSW gate array of the NAS subsystem to generate all required instruction addresses.

Figure 2-7 shows the instruction dispatch mechanism. The BR_RESTART signal is associated

Figure 2-7 AS_DISP Interface



with the UIR2 stage of the NAS subsystem pipe. Figure 2-7 illustrates a transfer without having any pipe stages being held.

2.2.2 CBUS - Register File write port for NDC subsystem

The CBUS interface is used to transfer data from the NDC subsystem to the register file within the NRFA gate arrays of the NAS subsystem. The sources of data are cache read data, memory return data from the NRC subsystem, and vector to scalar transfers from the NVP board. The register file is always able to accept the data allowing a write enable signal to control the transfer.

Table 2-8 shows the signals of the interface.

Table 2-8 CBUS Interface Signals

CBUS_DATA<63..0>	Data bus
CBUS_PAR<7..0>	Byte parity for the data bus
CBUS_REG_SEL<4..0>	Register file write location
CBUS_SIZE<1..0>	Size of data being transferred
CBUS_WR_EN	Bus write enable signal

The data and destination control signals are valid the same cycle that the CBUS_WR_EN signal is asserted.

2.2.3 YBUS - NAS subsystem general use bus

The YBUS consists of YBUS_DATA<63..0> and YBUS_PAR<7..0>. The YBUS is used as a general purpose bus to transfer register file data to other subsystems. Micro code fields are used to control the register file location to source the bus as well as the destination for the bus.

2.2.4 MEM_REQ - NAS subsystem memory request interface

The interface is used by the NAS subsystem to issue memory requests from micro code. The interface was designed to minimize cache read latency. To minimize the latency the NDC subsystem interprets the NAS subsystem pipe stage control signals instead of using a ready/ request interface. Table 2-9 lists the major signals involved in the interface.

Table 2-9 MEM_REQ Interface Signals

AS_DISP_DISPL<31..0>	Instruction dispatch displacement from NIP
AS_DISP_DISPL_PAR<3..0>	Byte parity for AS_DISP_DISPL from NIP
UIR1_BDREG_SEL<4..0>	Memory request destination register select
UIR1_INST_SEG<2..0>	Memory request instruction segment
US_AG_SEL<3..0>	Address generation type select
US_BDSIZE<1..0>	Memory request size
US_FAKE_RING0	Ignore ring protection checks
US_MEM_OP_REQ	Memory operation request signal
US_MFP_OP<10..0>	Memory operation type
ZMUX_DATA<31..0>	Address generation register file read port
ZMUX_PAR<3..0>	Byte parity for ZMUX_DATA

Figure 2-7 shows the pipe stages in which information is transferred for a memory request. The AS_DISP_DISPL field is sourced by the NIP subsystem. Refer to the micro code chapter of this document for information on the US_MFP_OP and US_AG_SEL fields.

2.2.5 UINT - Micro interrupt interface

This interface is used to inform the micro sequencer of the NAS subsystem of a condition in the NDC subsystem that requires assistance to proceed. The most common condition is a PTE miss when reading the PTE cache for a memory access. The NAS subsystem responds by micro interrupting the micro sequencer and executing micro code to handle the condition. The signals which are included in the interface are list in Table 2-10.

Figure 2-7 MEM_REQ Interface

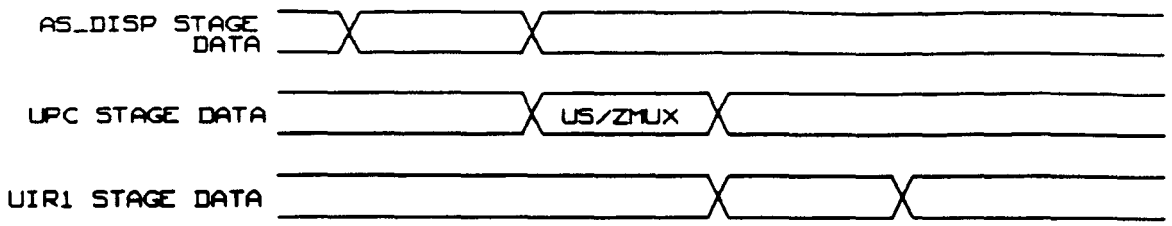
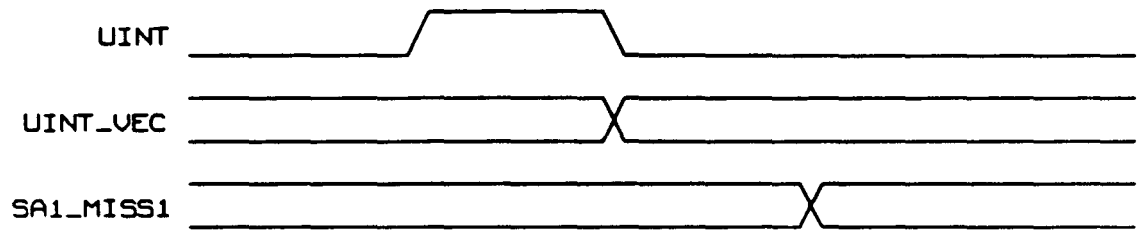


Table 2-10 Uint Interface Signals

UINT	Micro interrupt signal
UINT_VEC<3..0>	Micro interrupt type
SA1_MISS1<31..0>	Logical address causing problem

Figure 2-8 shows the timing of the signals for the interface. The micro interrupt vector is valid one

Figure 2-8 UINT Interface



cycle after UINT, and the logical address causing the problem is valid two cycles after UINT. Both UINT_VEC and SA1_MISS1 are valid until the NAS subsystem has completed processing the micro interrupt. The logical address is used for accessing SDRs and first level PTEs contained in the scratch ram. Refer to the signal list appendix for decoding details of UINT_VEC.

2.2.6 RC - Memory return destination control interface

The NDC subsystem transfers destination information to the NRC subsystem for each read request sent to the cross bar. The RC interface is used to transfer the information to the NRC subsystem. The NRC subsystem writes the destination information into a queue. When the corresponding read data is returned from the cross bar the destination information is read from the queue and used to control where the data is transferred to. Table 2-10 lists the signals included in the interface.

Table 2-11 RC Interface Signals

RC_2T_FIRST	First of two transfers signal
RC_DST<2..0>	Read data destination
RC_EVEN	Even side memory data required
RC_ODD	Odd side memory data required
RC_PTE	Data is a page table entry

RC_RDY	NDC subsystem handshake signal
RC_REG_SEL<4..0>	Register file write location
RC_REQ_NEXT	NRC subsystem handshake signal
RC_SIZE<1..0>	Data size
RC_TAG<4..0>	Data cache update tag
SA1_ADDR2<2..0>	Logical address of request (least significant 3 bits)
SA1_ADDR2_PAR<3..0>	Byte parity for logical address
SA1_MISS1<31..3>	Logical address of request (most significant 29 bits)

The signal RC_2T_FIRST indicates that two requests to the cross bar were required for a single data cache request. Examples would be an unaligned long word request, or a word vector request running at 2X rate which is not word aligned. The signal informs the NRC subsystem that the data for this RC transfer and the next RC transfer should be combined before being transferred to the destination. For the case of the unaligned long word each of the two RC transfers would contribute a word to the long word result.

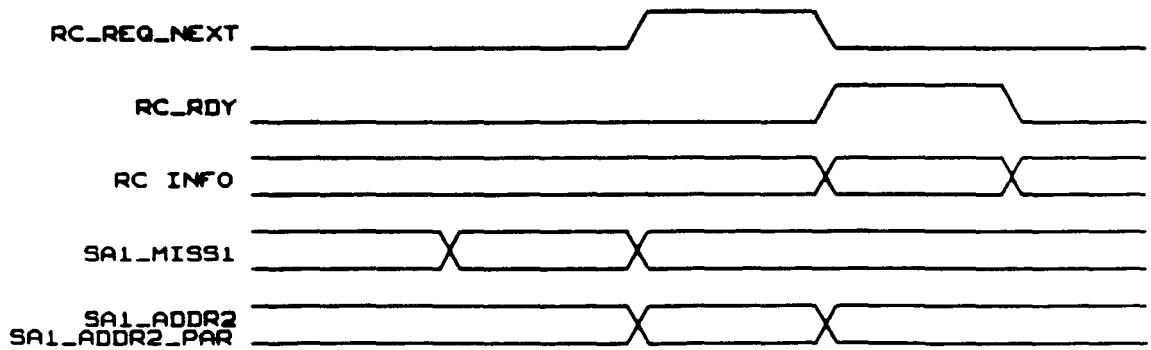
The RC_DST signals specify the destination for the returned data, refer to the signal list appendix for decode information. The RC_EVEN and RC_ODD signals specify whether a request was made to the corresponding side of the cross bar. If a request was made then the NRC subsystem waits for data to be returned from that side of the cross bar. Otherwise if the transfer requires data from a side of memory where no request was issued then the last data that was returned on that side is used. This mechanism (of being able to use the previously read data) is called read data buffering and is a performance feature. An example of an instruction which would use the feature is a vector byte load. Provided the stride is one, four consecutive vector address generator requests would be issued for the same word in memory. With the feature only the first VAG request would access memory, the three that follow would use the same return data as the first.

The RC_PTE signal marks a request as data for a micro interrupt routine. Normally returned cross bar data to the NRC subsystem is handled in a first-in-first-out (FIFO) order. A deadlock condition may occur if the FIFO order is maintained for PTE requests. The RC_PTE signal marks a request as a priority transfer and will not be held if previous requests can not be transferred.

The signals RC_REG_SEL and RC_SIZE are used when the destination is the register file of the NAS subsystem. The RC_TAG field is used for data cache update requests. The SA1_MISS1 logical address is used for data cache updates and for instruction look ahead requests.

Figure 2-9 illustrates the mechanism for the interface. For gate array I/O pin count reasons the

Figure 2-9 RC Interface



logical address, SA1_MISS1, is transferred to the NRC gate arrays two cycles earlier than the RC information. The least significant three bits of the logical address are modified by the request type and are sent out a cycle later than the most significant 29 bits. The NRC subsystem stages the SA1_MISS1 data for two cycles and the SA1_ADDR2 data for one cycle using the same hold conditions used by the NDC subsystem for the corresponding register stages.

2.2.7 MXS/UPD - NDC subsystem memory return data

This interface is used to transfer data received from the cross bar by the NRC subsystem to the NDC subsystem. The two interfaces, MXS and UPD, share the same data bus, UPD_DATA. All other handshake and control signals are not shared. Table 2-12 lists the signals of the interface.

Table 2-12 MXS/UPD Interface Signals

MXS_PTE_EVEN	Even side PTE transfer
MXS_PTE_ODD	Odd side PTE transfer
MXS_RDY	NRC subsystem handshake signal
MXS_REG_SEL<4..0>	Register file write port
MXS_REQ_NEXT	NDC subsystem handshake signal
MXS_SIZE<1..0>	Data size
UPD_ADDR<31..0>	Logical address for cache update
UPD_ADDR_PAR<3..0>	Byte parity for logical address
UPD_DATA<63..0>	Data bus
UPD_EVEN	Even side update data
UPD_ODD	Odd side update data
UPD_PAR<7..0>	Byte parity for data bus
UPD_RDY	NRC subsystem handshake signal
UPD_REQ_NEXT	NDC subsystem handshake signal
UPD_TAG<4..0>	Update tag

The MXS interface is used to transfer data to the NDC subsystem which is passed on using the CBUS to the register file within the NRFA gate arrays of the NAS subsystem. The signals MXS_PTE_EVEN and MXS_PTE_ODD are used to rotate the PTE data to the least significant 32 bits of the CBUS. If the transfer is not PTE data then UPD_ADDR<2..0> is used to control the rotation of the data.

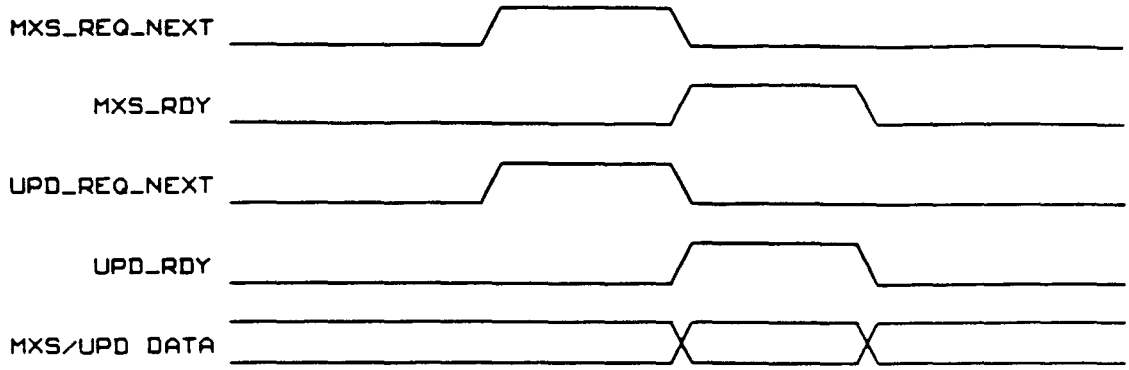
The UPD interface is used to write memory return data to the data cache. The location within the cache to write the data is obtained from UPD_ADDR. The signals UPD_EVEN and UPD_ODD specify which of the two data cache sides (or both) is to be written. UPD_TAG is used to compare the tag which was written into the data cache update tag rams to determine if the cache should still be updated.

Figure 2-10 shows the handshake mechanism for the interface. The data being transferred on the interface may be an MXS transfer, a UPD transfer or both.

2.2.8 IXA - NIP subsystem look ahead request interface

The IXA interface is used to transfer instruction look ahead requests to the NDC subsystem. The

Figure 2-10 MXS/UPD Interface



NDC subsystem processes the request and issues cross bar requests for memory data. Table 2-13 lists the signals included in the interface.

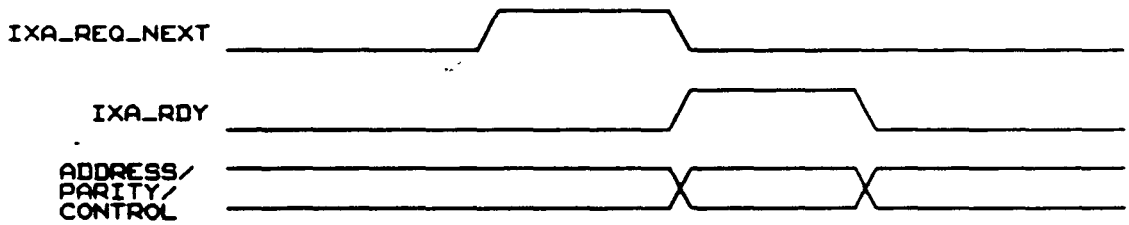
Table 2-13 IXA Interface Signals

IXA_ADDR<31..0>	Instruction look ahead address
IXA_ADDR_PAR<3..0>	Byte parity for IXA_ADDR
IXA_NOFLT	No fault request indicator
IXA_RDY	NIP subsystem handshake signal
IXA_REQ_NEXT	NDC subsystem handshake signal

The IXA_NOFLT signal is used to specify that a request should not cause a memory fault. When the signal is asserted PTE misses are resolved but access violations and invalid or non-resident PTE entries cause the IXA request to be dropped. The only time the NIP will issue an IXA request with out asserting the IXA_NOFLT signal is when that specific location of memory is required to parse the next instruction.

Figure 2-11 shows the mechanism for the interface.

Figure 2-11 IXA Interface



2.2.9 MXI - NIP subsystem memory return data

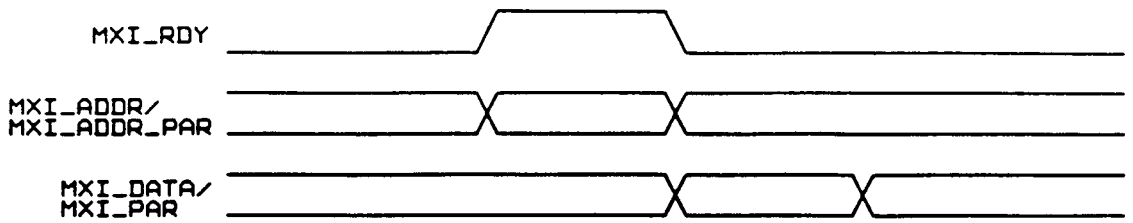
The MXI interface is used to transfer memory return data to the NIP subsystem instruction cache. The MXI_ADDR field is used to select the location of the cache to write the data, MXI_DATA. Table 2-14 lists the signals included in the interface.

Table 2-14 MXI Interface Signals

MXI_ADDR<31..0>	Logical address of data
MXI_ADDR_PAR<3..0>	Byte parity for MXI_ADDR
MXI_DATA<63..0>	Data to be written in cache
MXI_PAR<7..0>	Byte parity for MXI_DATA
MXI_RDY	NRC subsystem handshake signal

Figure 2-12 shows the mechanism used for the interface. The data is staged one additional cycle

Figure 2-12 MXI Interface



so that it can be presented directly to the instruction cache rams. The address must enter the NIAD gate arrays to be selected to the instruction cache rams. The address is staged in a register immediately after entering the NIAD gate arrays. The NIP subsystem can always accept a transfer so no handshake from the NIP subsystem is needed.

2.3 Scan Interface

All registerson the board are connected via a unidirectional scan ring. The scan ring is controlled by a set of scan control signals which are driven from the crossbar. The signals which make up the scan interface are listed in Table 2-15.

Table 2-15 Scan Interface Signals

SP_XC.SCAN_OUT	Board scan ring output
XC_SP.SCAN_CTL<2..0>	Scan mode selection signals
XC_SP.SCAN_IN	Board scan ring input

The available modes for the scan mode selection signals are listed in Table 2-16.

Table 2-16 Scan Mode Selections

0	Normal execution mode
1 thru 4	Unused
5	Last Left Shift
6	Load Mode (Cast)
7	Left Shift

The SCAN_CTL signal is only allowed to change value when the clock to the board is disabled. SCAN_CTL mode 0 enables normal board operation. This mode is used for normal execution.

The board is scanned by issuing scan ring length minus one clocks with mode 7 (left shift) followed

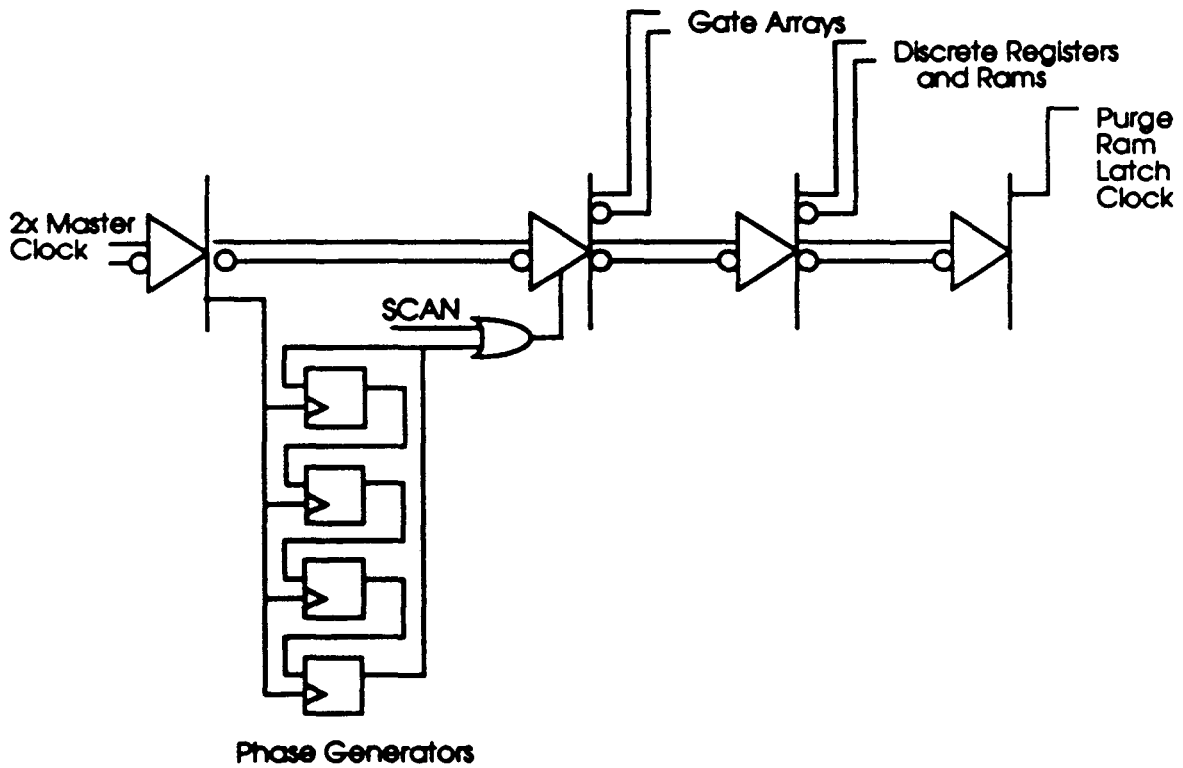
by one clock with mode 5 (last left shift). The last left shift is required to inform the self timed rams that the command scanned in should be executed. The difference between the left shift mode and the last left shift mode is that the last left shift mode does not assert the scan enable signal to the self timed rams. The self timed rams have a register which is not on the scan ring used to remember that scan was enabled on the previous cycle. On the first cycle that the scan enable input pin is deasserted the internal register forces the part to shift, but also forces the part to execute the command that was shifted in.

Mode 6 is used for CAST (Convex at speed test) and SST. This mode disables all rams and latches from writting. Additionally the mode forces all clock gating to registers to all a clock to occur.

2.4 Clock Interface

All clocks for the board are generated from a single differential clock source. The name of the input clock signal is CU_SP.CLOCK_2X. The clock signal can be generated with from one to three rising edges per major system clock period (16.67 ns). The NSP board only uses one and two rising edges per major system clock period. Figure 2-13 shows a functional diagram of the clock generation and fanout logic contained on the NSP board.

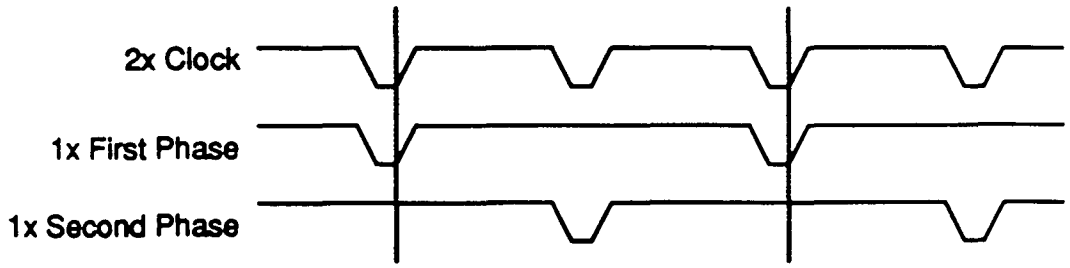
Figure 2-13 Clock Generation



When the scan mode is set to do a left shift (or last left shift) the clock is issued with one rising edge per major system clock period (1x rate). The 1x rate is distributed on the NSP board to all registers to allow scanning to occur. The phase generators are disabled from effecting the generated clocks to allow the phase generation registers to be part of the scan ring.

During normal mode the phase generators are initialized to generate 1x first phase, 1x second phase and 2x clocks. There are three phase generators implemented using discrete registers on the board to generate the three type of clocks. Additionally, the gate arrays which require 2x clocks also have clock generators on the gate array to generate 2x and 1x second phase clocks. Figure 2-14 illustrates the clock wave forms which are generated.

Figure 2-14 Clock Wave Forms



3 NAS Subsystem

The functionality of the NAS subsystem is presented in this chapter. References will be made to the Neptune Scalar Processor Block Diagram, which is separate from this document.

3.1 Overview

The Neptune Address and Scalar data path subsystem (NAS) is the microcoded engine that controls the execution of instructions by the scalar processor. The NAS receives instruction dispatches from the NIP and executes a microcode routine to perform the function required by the instruction. For vector instructions, the NAS works together with the NIP to start the vector processor. The NAS can also restart the NIP to execute non-sequential program flows, such as jumps and exceptions. Also contained within the NAS are the architecturally defined address and scalar registers, as well as the Program Status Word (PSW). For memory reference instructions, the NAS generates logical addresses for and initiates requests to the NDC subsystem. The NAS contains function unit chips which are used to execute floating point operations. Through microcode control, the NAS directs context save and restore operations for the entire NSP.

The NAS is implemented using four Vitesse 30K gate array types and three Vitesse custom parts. The gate array types used, with quantities in parenthesis, are:

- NUS - microsequencer (1)
- NRFA - register file / ALU slice (4)
- NPSW - program status word (1)
- NMISC - miscellaneous function unit (1)

There are 1 each of the following custom parts:

- NFAD - floating point addition
- NMUL - integer and floating point multiplication
- NDIV - integer division, floating point division and square root

There are also 38 self-timed RAMs - 34 for control store and 4 for scratch RAM. Part of scratch RAM is used as a first level PTE cache. A single Vitesse purgeable self-timed RAM is associated with the PTE1 cache half of scratch RAM as a validity bit. Finally, there are some ECLinPS parts used for buffering and "glue" logic.

A block diagram for the data paths within the NAS is included in the Neptune Scalar Processor Block Diagram. The microsequencer portion of the NAS is in the lower left corner. The remainder of the NAS is found in the upper center of the drawing.

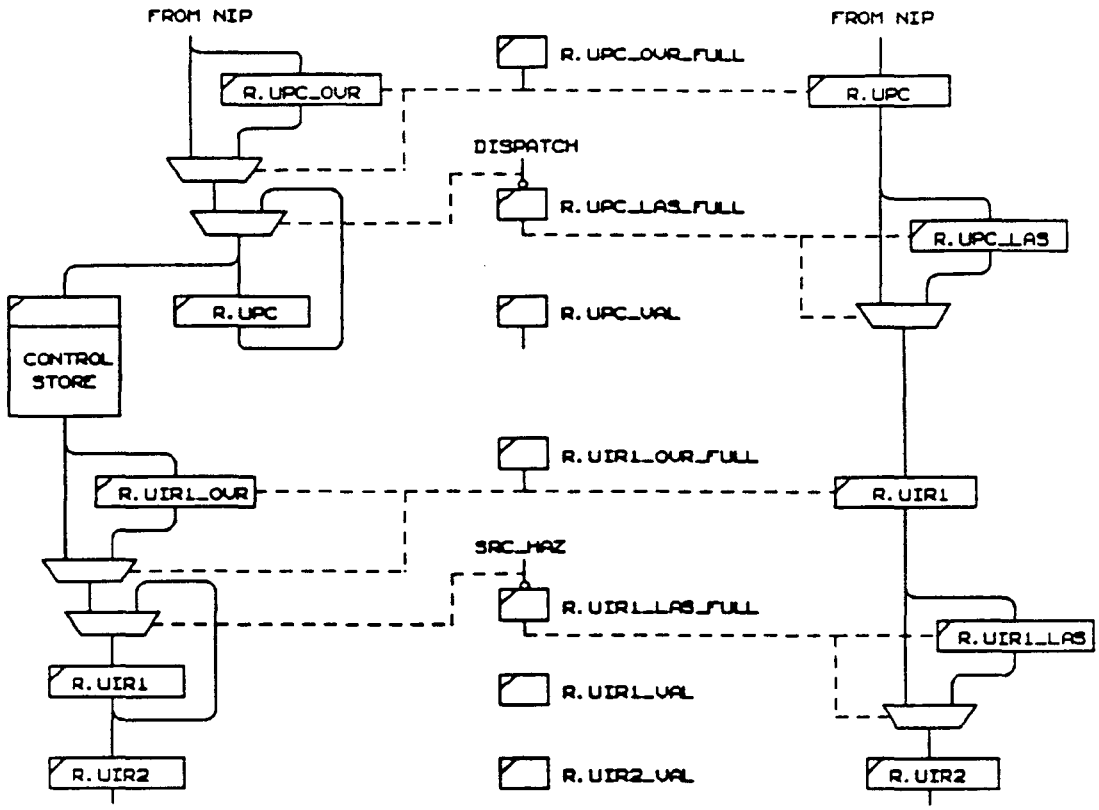
3.2 NAS Pipeline Stages

The NAS is a pipelined design. The standard pipeline stages explained here are used throughout the NAS, and since the rest of the NSP interfaces with the NAS, the NAS outputs its pipeline control signals for the rest of the NSP as part of its interfaces. Therefore, a good grasp of the NAS pipeline is crucial to understanding how the NSP works. For this reason, the basic pipeline is presented first in the NAS chapter, and the NAS chapter is first in the subsystem description chapters of this document. The functions which define the pipeline control signals will be detailed

later in this chapter, as the logic that generates them is described.

The NAS pipeline has 3 major stages - control store access, operand access, and execution. These are the stages all instructions and traps dispatched by the NIP follow. At the input of the pipeline is the NIP dispatch interface. At the output of the pipeline are the various execution units within the NAS, such as the integer ALU and function units. A block diagram of these 3 basic pipeline stages is shown in Figure 3-1.

Figure 3-1: NAS Pipeline Stages



The left side of Figure 3-1 shows the pipeline as it is implemented within the parts of the NAS that generate pipeline control signals. The right side shows how it is implemented in the rest of the NAS, where pipeline controls are received as inputs. The reason for the 2 versions will be explained shortly. There are many muxes and intermediate registers (with _OVR and _LAS suffixed names) shown in this diagram, but let's ignore them for the time being. We then see that the basic stages of the pipeline are the **UPC**, **UIR1**, and **UIR2** levels.

When the NAS accepts an instruction dispatch from the NIP, dispatch data enters the *control store access* stage of the pipeline. This level is referred to as the **UPC** level, which is an acronym for microprogram counter. The microprogram counter R.UPC mirrors the address register in the control store self-timed RAMs, which is used to access it for reads. This level of the pipeline shows what the microsequencer is currently working on.

Microinstruction fields read from control store flow into the *operand access* stage of the pipeline. This stage of the pipeline is referred to as the **UIR1** level, for microinstruction register level 1. All fields of the microinstruction flow through the pipeline on their journey from dispatch to execution. At the UIR1 stage of the pipe, data operands and other resources required for the microinstruction

to execute are accessed and hazards are checked. A hazard is simply a condition that prevents the access of a resource. For example, trying to read **S2** when load data is pending for **S2** causes a hazard. Some other types of hazards are:

- the microsequencer attempting to access a test condition that has a pending result
- the NIP attempting to branch based on a condition bit (**PSW<AC>** or **PSW<SC>**) with a pending result
- an attempt to start a function unit operation on a busy function unit (e.g. back to back divides, since divides take more than one clock)
- making a memory request to the NDC when it's already busy processing a previous request

All hazards are checked and resolved at the UIR1 level of the pipeline. This level may stall for a large number of clocks.

The final level of the pipeline is the *execute* stage. This level is referred to as the **UIR2** level, for microinstruction level 2. At the UIR2 level, operands and other resources are guaranteed to be available, and the operations in the microinstruction are executed.

All register names in the NSP conform to a naming standard to identify the level of the pipeline they are part of. For example, the object FOO staged through the basic pipeline flows from R.UPC_FOO to R.UIR1_FOO to R.UIR2_FOO. These register names are shortened by dropping their preceding R. when they are driven off chip. For example, R.UPC_VAL becomes UPC_VAL. The two names will be used interchangeably here since they refer to the same logic signal.

To enforce timing restrictions at the interfaces between subsystems, the pipeline includes *overrun* or *lookaside* registers. These registers hold data when a stage in the pipeline is stalled due to a hazard of some sort. For example, when dispatch data passes from the NIP to the NAS into the UPC level of the pipe, if the UPC level is stalled, there isn't time for the NAS to inform the NIP and make it hold the transfer, all on one clock. Therefore, the NAS has an overrun register to take one extra transfer from the NIP, permitting a register delay in the "stop, I can't take anymore" signal from the NAS to NIP. Referring to Figure 3-1, the NAS holds overrun data from the NIP in the R.UPC_OVR register. On the next clock, the NAS knows it has overrun data, so it will not load the overrun anymore and is set up to select the overrun register into the UPC level rather than the NIP data. At the same time the NIP is informed of the stall so it does not attempt another dispatch.

A lookaside register is different than an overrun register because it holds data for a pipeline stage *after* the pipeline register, rather than at the input to the pipeline register. These two different types of pipeline holding registers are required due to timing considerations. If the gate array that holds the pipeline stage also performs the hazard checks at that pipeline stage, an overrun can be used. This is because the hazard signal does not have to cross chip boundaries and can make timing to hold an input overrun register. For example, the control for the UPC level of the pipeline is maintained in the NUS microsequencer array in the NAS. Therefore, the NUS can hold its own R.UPC registers and make timing with an R.UPC_OVR arrangement. The register naming convention is continued with the _OVR suffix for an overrun register and the _LAS suffix for a lookaside - for example R.UPC_FOO_OVR and R.UPC_FOO_LAS. The UPC-level control signals won't make timing to the rest of the NSP to hold an overrun register. Therefore, the data is held after the UPC level in a lookaside register, and a registered version of the UPC stall signal is used to mux the input to the UIR1 level between R.UPC and R.UPC_LAS. The registered stall signal is also used to keep the lookaside register from loading after it has taken data. Similarly,

the control for the UIR1 level is maintained in the NRFA arrays in the NAS, so it has UIR1 overruns, while the rest of the NSP has UIR1 lookasides.

Each stage of the pipeline has a control signal associated with it, shown in the center of Figure 3-1. For the 3 major stages of the pipeline (UPC, UIR1, UIR2) there are *valid* signals (UPC_VAL, UIR1_VAL, UIR2_VAL). The assertion of these signals implies there is a valid microinstruction (or data, for non-microinstruction staging) in the associated pipeline level. These signals are used as qualifiers to insure operations are only performed on valid microinstructions and data. For example, UIR2_VAL must be asserted for the execution of an operation decoded from a UIR2 microinstruction field to actually occur.

The overruns and lookasides also have control signals associated with them:

- UPC level - UPC_OVR_FULL and UPC_LAS_FULL
- UIR1 level - UIR1_OVR_FULL and UIR1_LAS_FULL

The UIR2 level has no overrun or lookaside, because it cannot stall. The _OVR_FULL and _LAS_FULL signals are used in conjunction with the pipeline stage valid signal to show where in the pipeline level data comes from. For example, if UIR1_VAL is set and UIR1_LAS_FULL is clear, valid data is in the UIR1 level of the pipe, and it is found in the R.UIR1_FOO type registers. If UIR1_VAL is set and UIR1_LAS_FULL is also set, valid UIR1 data is found in the R.UIR1_FOO_LAS type registers. When empty, the overrun and lookaside registers free clock. This means that overruns will contain the same data as the pipe level they take overrun for, and lookasides will shadow the pipe stage they lookaside for. The _OVR_FULL and _LAS_FULL are used to hold data in the overrun or lookaside registers once they are full. The _OVR_FULL signal is used to hold _OVR registers (either through input muxing or clock gating) on the chip that generates the _OVR control. In addition, it controls muxing of the input to the pipeline stage that the overrun register is associated with. For example, the UIR1_OVR_FULL signal muxes the input to the UIR1 level. The _LAS_FULL signal is used to hold _LAS registers on chips outside the one that generates the _LAS_FULL signal. In addition, it controls muxing of the input to the pipeline stage following the stage the lookaside register is associated with. For example, UIR1_LAS_FULL muxes the input to the UIR2 level.

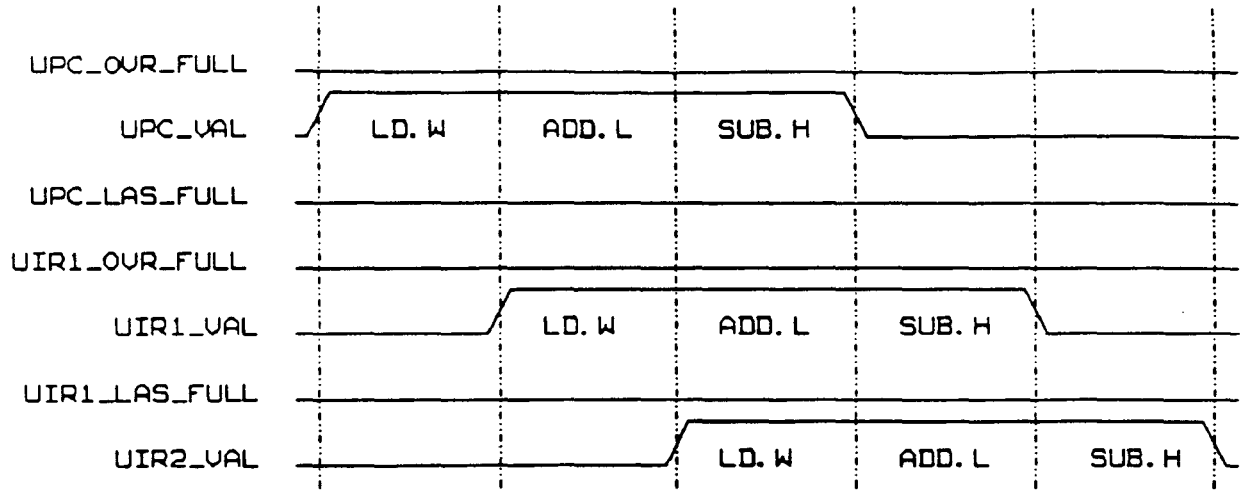
This basic pipeline is found throughout the NSP. There are a few exceptions for certain interfaces (like a UIR3 level for a small number of operations that must be delayed a clock after the UIR2 level for timing), but overall these pipeline rules apply.

To complete the discussion of the NSP pipeline stages, let's consider a few examples of instruction streams and how they flow through the pipe. The first example is a sequence of three one-clock (single microinstruction) instructions with no interdependencies:

```
ld.w    #1,a1
add.l   s0,s1
sub.h   a3,a4
```

Figure 3-2 shows the pipeline control signals for this sequence, assuming the NIP dispatches them on consecutive clocks. The vertical dashed lines denote clock boundaries. The instruction mnemonics are overlaid with the timing diagram to show the flow of instructions through the pipe. The overlaid information is for the registers in the level relevant to the control signal. For example, the microsequencer's view of UPC-level registers is shown under UPC_VAL.

Figure 3-2: Pipeline Example 1



The second example illustrates an overrun at the UPC level. Consider the following sequence:

```
ld.w    #1,a1
ld.d    #2.0,s2
add.l   s0,s1
sub.h   a3,a4
```

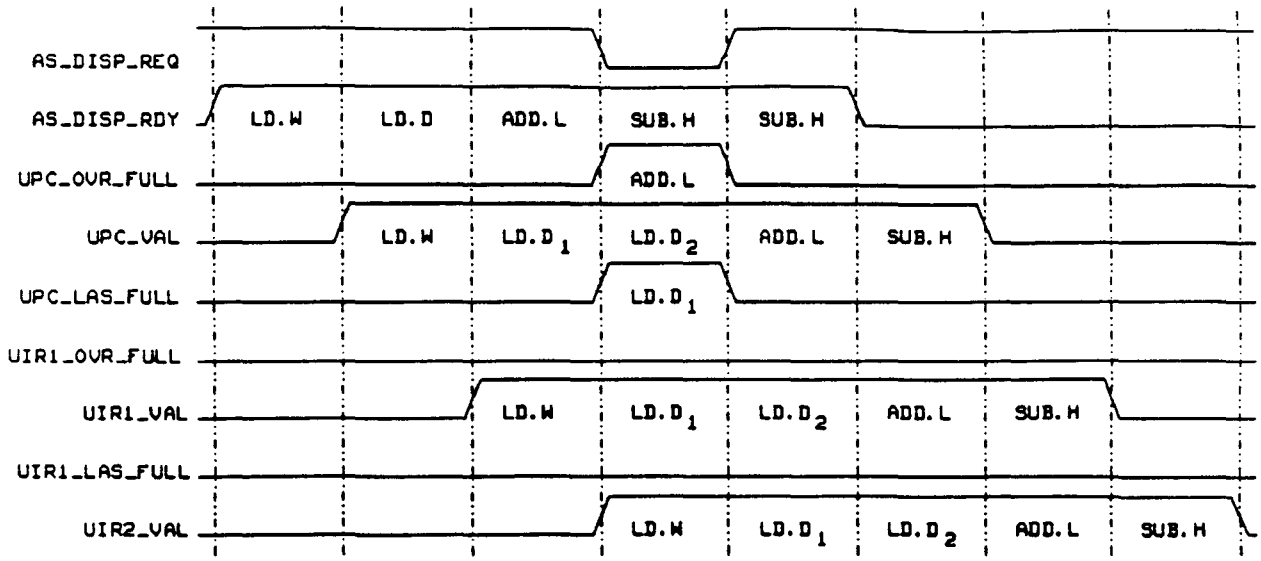
The *ld.d* instruction takes 2 cycles (i.e. 2 microinstructions) to execute, so if the NIP dispatches all 4 instructions consecutively, the *add.l* would overrun the second microinstruction of the *ld.d*. Instead, the *add.l* dispatch information goes into the UPC_OVR registers and the NIP is held off dispatching the *sub.h* for 1 clock. The timeline for this sequence is shown in Figure 3-3. In addition to the pipeline control signals, the NIP dispatch interface signals are shown. The combination of the assertion of AS_DISP_REQ and AS_DISP_RDY constitutes an instruction dispatch with the entrypoint, AS_DISP_EP, represented mnemonically under AS_DISP_RDY. THis Shows the input to the UPC level of the pipeline. Note also that UPC_LAS_FULL is asserted the same time that UPC_OVR_FULL is. The UPC_LAS level is used outside the microsequencer to hold dispatch information for the macroinstruction, rather than information for each microinstruction, which only the microsequencer cares about. Dispatch information (such as register selects) for the *ld.d* are held in the UPC_LAS level as long as the microsequencer isn't exiting the macroinstruction (called dispatching). The microsequencer informs the rest of the NAS of this condition via UPC_LAS_FULL, while it uses UPC_OVR_FULL to hold the information of the *add.l* dispatch to be used at the input of the UPC level when the *ld.d* dispatches.

The final example illustrates a hazard causing an overrun at the UIR1 level of the pipeline. Consider the following sequence, assuming *op1* and *op2* are resident in the PTE and data caches:

```
ld.w    op1,a1
ld.w    op2,a2
add.w   a1,a2
st.w    a2,op1
```

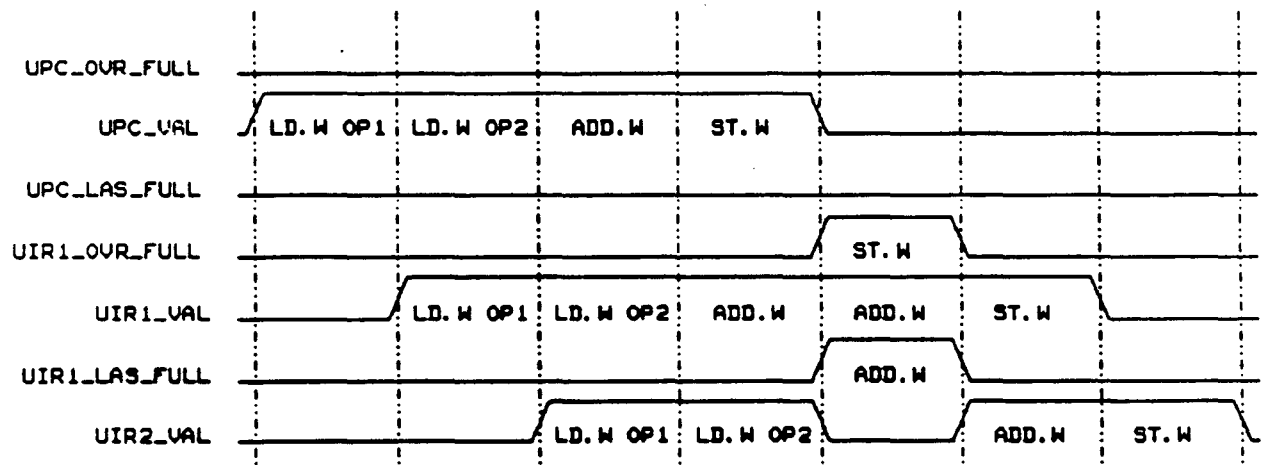
Two clocks are required from the UIR1 request until data is returned from the cache to the register

Figure 3-3: Pipeline Example 2



file, so the *add.w* will stall for one clock at the UIR1 level waiting for *op2* before it can proceed to the UIR2 level and execute. The timeline for this sequence is shown in Figure 3-4. UPC_VAL shows the UPC level registers within the NUS. UIR1_VAL, UIR1_OVR_FULL, and UIR2_VAL are shown related to the NRFAs. UIR1_LAS_FULL is shown related to parts of the NAS outside the NRFAs..

Figure 3-4: Pipeline Example 3



3.3 Instruction Dispatch

When the NIP completes parsing of a macroinstruction, it is handed off to the NAS via the instruction dispatch interface. Chapter 2 detailed this interface. In this section, we will describe what the NAS does with the interface inputs and which part of the NAS receives the interface signals.

When the microsequencer (NUS array) asserts AS_DISP_REQ, it means it can accept an instruction dispatch, either in the UPC or UPC_OVR level of the NAS pipeline. The NIP confirms

the dispatch with AS_DISP_RDY, which is used by the NUS in generation of the pipeline control signals and to control the selection of the next microaddress. The NUS takes the entrypoint address, AS_DISP_EP, appends 11_2 to the most significant end (making all entrypoints greater than or equal to $C00_{16}$), and uses it for the first address of a microroutine to execute the dispatched instruction. The overrun allows the NUS to accept a dispatch while it's still executing the microroutine for a previous dispatch. After the overrun is full, no more dispatches are accepted (AS_DISP_REQ is dropped).

The NIP also parses the register selects from the macroinstruction and gives them to the NAS. There are 3 such selects in the dispatch (AS_DISP_IREG, AS_DISP_JREG, and AS_DISP_KREG). The NAS only uses 2 of these - usually the J and K selects. For the few instructions that use the I select, AS_DISP_SEL_IREG is asserted by the NIP forcing AS_DISP_IREG to be used. This is implemented with an ECLinPS mux, which produces MUNGED_DISP_KREG at its output. AS_DISP_JREG and MUNGED_DISP_KREG are input to the NRFA slices to be staged and used as parameterized register selects for the instruction. These parameters must be available for all microinstructions in the routine, whether the macroinstruction takes 1 cycle or 20 to execute. Therefore, the NRFA recirculates its UIR1 level register selects back to the UPC level whenever UPC_OVR_FULL is asserted (this implies we're executing a multi-microinstruction routine). The NRFAs also receive branch tag information (UPC_BR_POL and UPC_BR_SEL) at the UPC level of the pipe and use it to generate branch restarts, asserting BR_RESTART back to the NIP. Branch restarts are described in more detail in section 3.14.3, beginning on page 3-39. The NRFA also receives the displacement field of the dispatched instruction (AS_DISP_DISPL) for use in the data path for immediate operand instructions.

Information about the program counter is also supplied with the instruction dispatch. The NPSW array provides multiple program counters for the NAS data path. The current program counter, UPC_CPC, is provided with the dispatch to the NPSW as the basis for all PCs it generates. Along with UPC_CPC are the branch displacement, instruction size, and extended opcode indicator. These signals (AS_DISP_BR_DISPL, AS_DISP_SIZE, and AS_DISP_XTEND) are provided at the input to the UPC level of the pipeline. UPC_CPC is transferred one stage later for timing reasons. Refer to section 3.11, beginning on page 3-33, for more details on the PC generation performed by the NPSW.

3.4 Microsequencer

Control of the NAS microengine emanates from the microsequencer, contained primarily in the NUS gate array. The NUS takes dispatches from the NIP and sequences the microinstruction stream to execute the prescribed instruction. It provides addresses to the control store, which contain the various microinstruction fields. Its sequencing logic includes conditional operations based on test inputs to the NUS. It also accepts microinterrupts from the NDC, temporarily redirecting its microinstruction sequence. The major functions of the microsequencer are described in the following sections. For more detail, refer to the NUS Gate Array Specification.

3.4.1 Control Store

The NSP control store contains 4096 microinstructions, each 151 bits in width. The format of the microinstruction word is detailed in Chapter 7. Control store parity is spread throughout the RAMs as microinstruction fields. There are 6 parity bits included in the 151 bit microinstruction, with a single parity bit assigned to each group of fields that go to a different gate array type. For example, the US_NDC_PAR bit is odd parity over all bits in the microinstruction that are driven to the NDC array. Tracing parity error sources is covered in detail in section 3.18, beginning on page 3-41.

NSP control store is implemented with 38 2K by 9 bit STRAMs (initially the National 100492), and organized as two 2K banks of 19 STRAMs each. Outputs of the two banks are wire-ORed, except in time critical situations where they are discrete-ORed with ECLinPS parts or within a gate array. The signal naming convention for control store outputs prefixes the mnemonic with US_ for wire-ORed fields. For discretely ORed fields, the lower bank (addresses 000_{16} through $7FF_{16}$) outputs are prefixed with L_US_, and the upper bank (addresses 800_{16} through FFF_{16}) with U_US_.

The STRAM addresses are provided by the NUS array. The most significant bit of the microaddress, CSBANK, is output in true and complement form and acts as the bank control ($\overline{CS}$ on the STRAM). The other 11 bits of the microaddress are output as CSADDR<10..0> (and the fanout copy CSADDR2<10..0>) and are used as the address input to the STRAM. Gate array outputs use the 25-ohm driver macro available in the ASICs.

Control store is loaded via the STRAM's scan feature. All of the RAM control registers (address, write enable, chip select, etc.) are in registers in the STRAM scan ring. These STRAM scan rings are included in the NSP ring, and require no special treatment other than the last left shift control that all NSP STRAMs require.

3.4.2 Next Address Flow

The main function of the microsequencer is to provide the address stream to control store dictated by the microprogram. This is achieved via the next address logic portion of the NUS array. The L_US_BRTYPE and U_US_BRTYPE fields of the microinstruction are input to the NUS and control the next address sequencing. The decoding of this field is detailed in Chapter 7. Typical sequencing functions such as jump, conditional jump, subroutine call and return are provided. A 4 entry microstack is included to allow nested subroutining. The branch address for both unconditional and conditional jumps is provided with the L_US_BRADDR and U_US_BRADDR inputs to the NUS. The dispatched entrypoint from the NIP (AS_DISP_EP) may also be selected as the next address upon the completion of a microroutine, which the microsequencer accomplishes with its "dispatch" branch type.

Microaddress sequencing occurs at the UPC level of the NSP pipeline. The output of the next address logic, CSADDR and CSBANK, is registered internally to the NUS in the microprogram counter, R.UPC. It is incremented by one to form the next microaddress for failed conditional operations. R.UPC is also staged to the UIR1 level of the pipeline and output from the NUS as UIR1_UPC for debug visibility. This provides the microinstruction address of both the UPC and UIR1 levels of the scalar pipeline for logic analyzer connection.

3.4.3 Test Conditions and Conditional Sequencing

The microsequencer can perform most branch operations conditionally as well as unconditionally. Most conditional operations are based on the TCOND<14..0> test condition inputs to the NUS array. There are also test conditions internal to the NUS array, some of which are scan writable only and reserved for future use as static flags if needed. Most of the external TCOND inputs are driven by the NPSW array, and include PSW bits, USW bits, CCR bits, etc. Refer to Chapter 7 for a complete definition of the test select encodings.

The test conditions are selected with the US_TSEL microinstruction field input, staged to the UIR1 level. US_TPOL is used as branch polarity, which tells the sequencer to consider the condition met if true (logic 1) or false (logic 0). Conditional operations must be pipelined since the test selects come from the UIR1 level of the pipeline. This means that a conditional operation is a 2-

step process. On the first microinstruction, the test condition select is specified. On the following microinstruction, the conditional operation (branch, call, etc.) is specified. The reason for this pipelined architecture is because there may be hazards on some of the test conditions. For example, the microcode may perform a communication register lock operation, and reserve PSW<AC> to receive the return status. If PSW<AC> is selected as a sequencer test condition before the status has returned from the NCU, we must wait for it. We saw in section 3.2, beginning on page 3-1, that all hazards are checked at the UIR1 level. Therefore, the UIR1 test select (UIR1_TSEL) is used to choose the condition needed at the UPC level in the microsequencer. This test hazard also feeds into the other UIR1 hazards and makes the entire NAS pipeline stall at the UIR1 level waiting for the pending test condition. For timing considerations, the test hazard output of the NUS is broken into 4 factored terms:

- THAZ_U1_U2VAL - test hazard if UIR1_LAS_FULL = 0 and UIR2_VAL = 1
- THAZ_U1LAS_U2VAL - test hazard if UIR1_LAS_FULL = 1 and UIR2_VAL = 1
- THAZ_U1_U2INV - test hazard if UIR1_LAS_FULL = 0 and UIR2_VAL = 0
- THAZ_U1LAS_U2INV - test hazard if UIR1_LAS_FULL = 1 and UIR2_VAL = 0

These 4 terms are combined externally to the NUS with UIR1_LAS_FULL and UIR2_VAL to form TEST_HAZ, which is input to the NRFAs for their source hazard logic. This is faster than driving UIR1_LAS_FULL and UIR2_VAL off the NRFAs, on to the NUS, and back to the NRFAs, and requires a single input pin to the NRFAs.

A test hazard at the UIR1 level also stalls the UPC level if it is performing a conditional operation, since the UPC level can't know which way to advance if the test condition it needs isn't valid. As an example of a test hazard flow, consider the following microcode sequence (expressed verbally - the real microcode syntax is presented in Chapter 7). The number on the left is the address of the microinstruction.

```
100: test PSW<SC>, unconditional jump to 101
101: conditional jump to 200

200: dispatch
```

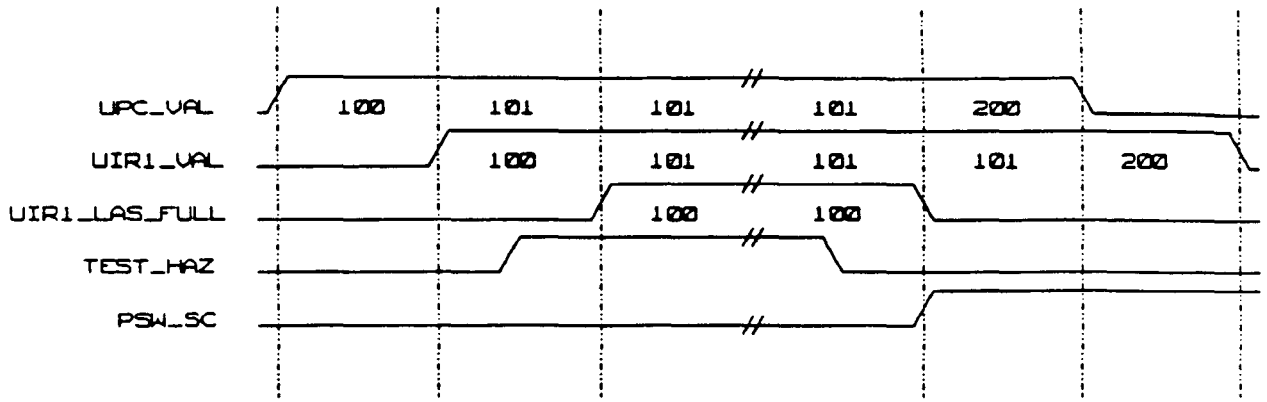
Assume that PSW<SC> is currently reserved for a status return from the NCU, and is currently 0. This implies that when valid status returns, we take the conditional branch to 200. A timeline for the UPC and UIR1 levels of this sequence is shown in Figure 3-5. The microinstruction address is placed under the pipeline control signals to show their advancement through the pipeline.

Microinstruction 100 flows from the UPC to UIR1 levels, and its test select choosing SC causes TEST_HAZ to become asserted. This keeps microinstruction 101 at the UPC level and also causes a UIR1 hazard, shown with the setting of UIR1_LAS_FULL. Microinstruction 100's test select is held in the UIR1_LAS until some clocks later, when the return status from the NCU clears TEST_HAZ. At the next clock, PSW_SC is valid and is selected, causing the UPC level to execute the branch to microinstruction 200.

3.4.4 Microinterrupts

In addition to its microprogram directed flow, the microsequencer can be directed by *microinterrupts* from the NDC subsystem. These are commands from the NDC to stop what's currently in progress and service a PTE miss or page fault. These commands are presented to the

Figure 3-5: Test Hazard Example



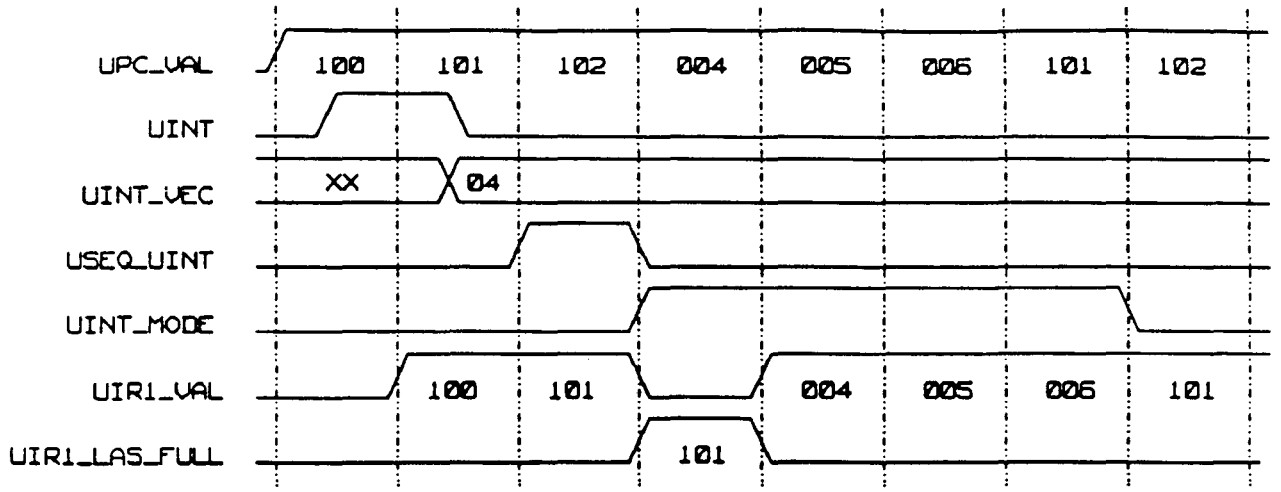
NUS via the microinterrupt interface, detailed in Chapter 2. This section describes how the microsequencer responds to these commands.

A microinterrupt is the highest priority selection in the next address logic. One clock after the NDC asserts UINT, the microinterrupt vector UINT_VEC is valid and is registered by the NUS. On the next clock this vector (zero extended on the most significant end) is forced as the next microaddress. The NAS considers the microinterrupt to have interrupted the microinstruction that is at the UIR1 level one clock after UINT is asserted by the NDC. The address of this microinstruction is pushed on the microinterrupt stack within the NUS. This stack is separate from the one used for microsubroutining. The NUS asserts UINT_MODE to the rest of the NSP, indicating a microinterrupt is being serviced. It also asserts USEQ_UINT to the NRFAs for one clock, causing the microinstruction currently at the UIR1 level to be aborted. The entire NAS pipeline is flushed by deasserting the pipeline stage valid signals. At the end of the microinterrupt routine, the NIP will be restarted to redispatch the aborted instruction, filling the NAS pipeline again. The NIP receives a special microinterrupt restart, which causes it to restart at the instruction at the UIR1_LAS level. Since USEQ_UINT forces a UIR1 level hazard, UIR1_LAS_FULL is asserted and the UIR1_LAS level holds the instruction the NAS is executing at UIR1, which is the instruction that was microinterrupted. There is one exception to this. If the NIP has been recently restarted (e.g. a jump instruction), it checks to see if the instruction it was restarted to has reached UIR1_LAS, and if not, it restarts for the microinterrupt restart at the PC it was given in the previous jump restart.

The microsequencer returns to the interrupted UIR1-level microinstruction via a special return from microinterrupt operation, which is one of the BRTYPE field encodings. A timeline example of microinstruction flow for a microinterrupt is shown in Figure 3-6. Assume the sequencer is executing an incrementing sequence of instructions beginning at address 100 before the interrupt occurs, that the microinterrupt vector is 004, and the microinterrupt routine completes at address 006 with a return from microinterrupt.

Microinterrupts may occur at any time. If they occur when the NAS is idle (not executing microcode), it returns to that state. Microinterrupts are also used to service page faults. These result in the NAS directing the NSP and NVP to save fault context and go to the macroprogram-level page fault handler. The type of microinterrupt service to be performed is specified by the microinterrupt vector. Refer to the Signal List appendix for the detailed encoding of UINT_VEC.

Figure 3-6: Microinterrupt Flow Example



3.5 Register File / ALU Data Path

At the heart of the NAS subsystem is the register file and integer ALU data path. This logic, implemented in 4 slices of NRFA gate arrays, forms the main integer execution unit for the scalar processor. The register file portion of this logic consists of a 6 port (3 write, 3 read) array of addressable latches which implement:

- 8 32-bit architecturally defined address registers A0 through A7
- 8 64-bit architecturally defined scalar registers S0 through S7
- 8 32-bit microcode temporary registers T0 through T7

The major data buses of the NSP flow through this logic, which is shown in the upper left of the Neptune Scalar Processor Block Diagram. The major busses are:

- X mux - operand bus from NRFAs to integer function units and to NRFA-internal ALU operand registers
- Y mux - operand bus from NRFAs to integer function units and to NRFA-internal ALU operand registers
- Z mux - register file port output to NDC for logical address generation
- Y bus - operand register from NRFA ALU output to entire NSP for inter-subsystem transfers such as vector stride or PSW writes
- A bus - NRFA ALU output, routed internally to the register file A write port and bypassable to the external world
- B bus - function unit result bus, input to register file B write port
- C bus - cache / memory / communication register / NVP transfer bus, input to register file C write port

3.5.1 Register Partitioning and Addressing

The register file for the NSP is contained within the 4 NRFA arrays. "Register file" is actually a misnomer, since due to cell count restrictions, it is actually implemented as a latch file. Each slice

contains a 32 entry, 8 bit wide array of master latches which hold the data. Across all 4 NRFAs this forms a 32 entry, 32-bit wide array of storage elements. To implement 32-bit A and T registers as well as 64-bit S registers, these latches are accessible in pairs. For 32-bit operations, each NRFA supplies 8 bits of the register and associated data path. For 64-bit operations, each NRFA supplies 16 bits - 8 bits of the least significant word and 8 bits of the most significant word. The 64-bit S registers are internally composed of 2 32-bit registers. Internal to the NRFA, the storage elements are named R.RF0 through R.RF1F (numbered 0_{16} through $1F_{16}$). For example, the architectural register S2 is composed of a 32-bit register called S2 by the microcode (R.RF14 in the NRFA) and a 32-bit register called S2U (U for upper - R.RF15 in the NRFAs). All registers are paired in this even/odd manner, which allows the manipulation two 32-bit registers as a longword. For example, A1 and A0 may be manipulated as a longword with A1 forming the most significant word and A0 forming the least significant word. The NRFA arrays are numbered backwards from memory byte ordering - NRFA0 contains the least significant byte and NRFA3 contains the most significant byte. Register partitioning is detailed in Figure 3-7.

Figure 3-7: Register File Partitioning

register name	NRFA name	NRFA3	NRFA2	NRFA1	NRFA0
A0	R.RF0	31..24	23..16	15..8	7..0
A1	R.RF1	31..24	23..16	15..8	7..0
A2	R.RF2	31..24	23..16	15..8	7..0
A3	R.RF3	31..24	23..16	15..8	7..0
A4	R.RF4	31..24	23..16	15..8	7..0
A5	R.RF5	31..24	23..16	15..8	7..0
A6	R.RF6	31..24	23..16	15..8	7..0
A7	R.RF7	31..24	23..16	15..8	7..0
T0	R.RF8	31..24	23..16	15..8	7..0
T1	R.RF9	31..24	23..16	15..8	7..0
T2	R.RFA	31..24	23..16	15..8	7..0
T3	R.RFB	31..24	23..16	15..8	7..0
T4	R.RFC	31..24	23..16	15..8	7..0
T5	R.RFD	31..24	23..16	15..8	7..0
T6	R.RFE	31..24	23..16	15..8	7..0
T7	R.RFF	31..24	23..16	15..8	7..0
S0	R.RF10	31..24	23..16	15..8	7..0
	R.RF11	63..56	55..48	47..40	39..32
S1	R.RF12	31..24	23..16	15..8	7..0
	R.RF13	63..56	55..48	47..40	39..32
S2	R.RF14	31..24	23..16	15..8	7..0
	R.RF15	63..56	55..48	47..40	39..32
S3	R.RF16	31..24	23..16	15..8	7..0
	R.RF17	63..56	55..48	47..40	39..32
S4	R.RF18	31..24	23..16	15..8	7..0
	R.RF19	63..56	55..48	47..40	39..32

S5	R.RF1A	31..24	23..16	15..8	7..0
	R.RF1B	63..56	55..48	47..40	39..32
S6	R.RF1C	31..24	23..16	15..8	7..0
	R.RF1D	63..56	55..48	47..40	39..32
S7	R.RF1E	31..24	23..16	15..8	7..0
	R.RF1F	63..56	55..48	47..40	39..32

The register file may be written from any one of three write ports - A, B, and C. The A port is controlled internally at the UIR2 level via pipeline-staged versions of the US_ABUS_WR_EN, US_ABUS_SIZE, and US_ABUS_FMT (which selects between US_XREG_SEL, US_YREG_SEL, and US_ZREG_SEL). The data input for the A port is the A bus output of the integer ALU. The B and C ports take writes from the B and C busses, respectively, and are described in section 3.5.5, beginning on page 3-21. There may never be 2 simultaneous writes from different ports. This is avoided because of the hazard scoreboard kept on the register file, which will stall the pipeline when a request for an operation that will cause a later B or C write encounters an A write in progress (and similarly between all port combinations). Hazards are discussed in more detail in section 3.6.1, beginning on page 3-22.

The register file is accessed through a sequence of read port muxes and slave latches, which hold the data output of the addressed master latch. Parity is generated at this level for read operations. Parity is not stored in the master latches on writes - bus parity is checked at the inputs of the NRFAs, from registered versions of the inputs. There are 3 read ports - X, Y, and Z, two of which (X and Y) have upper and lower byte portions. This means that within a single NRFA array, 5 8-bit registers can be accessed in parallel. For example, S0 could be accessed as a longword through the X port, placing R.RF10 on the lower X port and R.RF11 on the upper X port. Meanwhile, the A3/A2 pair could be accessed as a longword through the Y port, placing R.RF2 on the lower Y port and R.RF3 on the upper Y port. Finally, A6 could be accessed on the Z port, placing R.RF6 on the single 8-bit Z port output. Register addressing for the X and Y ports occurs at the UIR1 level of the pipeline, controlled by pipeline-staged versions of the US_XREG_SEL, US_XREG_FMT, US_YREG_SEL, and US_YREG_FMT microinstruction fields input to the NRFA. Register addressing for the Z port occurs at the special address generator (or advanced effective address) level of the pipeline, controlled by a pipeline-staged version of the US_ZREG_SEL microinstruction field input to the NRFA. This AG (address generator) level of the pipeline accesses the register required for the logical address used in a memory request at the UIR1 level of the pipeline. The AG level of the pipeline is discussed in greater detail in section 3.5.3, beginning on page 3-15. See chapter 7 for encodings of the (X,Y,Z)REG_SEL and (X,Y,Z)REG_FMT microinstruction fields.

The dispatched J and K register selects, AS_DISP_JREG and MUNGED_DISP_KREG, parsed from the macroinstruction by the NIP, may also be fed into the register select flow to reference either address (A) or scalar (S) registers, via encodings of the (X,Y,Z)REG_SEL fields.

3.5.2 Operand Data Path and Bypass

After readout from the register file, data is placed on various operand busses and used in the integer ALU or passed to other subsystems of the NSP. The two operand paths, X and Y, are driven by the output of the X and Y ports of the register file. This UIR1-level access supplies data to the integer ALU's UIR2-level X and Y operand registers. These operand registers are the data

input to the ALU, which is described in section 3.5.4, beginning on page 3-17. The Y operand register output is driven off the NRFA as the Y bus. This is the main inter-subsystem data transfer path for the NSP.

In addition to the register file, the X operand may be driven by sources external to the NRFA array. This logic is described in section 3.13, beginning on page 3-36. This external data feeds into the NRFA's XMUX_EXTDATA input, with odd byte parity provided on the XMUX_EXTPAR input. However, parity is only valid when the external data source is scratch RAM, so the XMUX_EXTPAR_VAL signa is asserted by the NUS and presented to the NRFA to convey this. The NRFAs maintain a lookaside register for XMUX_EXTDATA, which is selected for the X operand register if the USE_EXTDATA_LAS input is asserted when UIR1_LAS_FULL is also true. The X operand may also be driven by the displacement field of the macroinstruction. This is parsed by the NIP and given to the NRFAs on the AS_DISP_DISPL input (with AS_DISP_DISPL_PAR containing odd byte parity) as the instruction is dispatched. The displacement is staged to the UIR1 level in the NRFA in the usual manner. Finally, the NRFA may internally mux zeros into the X operand path. This X operand mux is controlled by the US_XMUX_SEL microinstruction field input to the NRFA, staged to the UIR1 level.

Somewhat separate from the X and Y operand muxes are the X and Y mux outputs of the NRFAs. These outputs provide operands to the function units at the UIR1 level. The function units register these inputs at a UIR2 level operand register, much like the integer ALU in the NRFA. Each NRFA drives 16 bits of the X mux. A byte of the lower word is output on XMUX_LDATA with odd byte parity on XMUX_LPAR. A byte of the upper word is output on XMUX_UDATA with parity on XMUX_UPAR. Together, all 4 NRFAs form the XMUX_DATA and XMUX_PAR buses driven to all function units. All outputs use the special 25-ohm drive gate array output buffers. The X mux outputs may be driven by the register file read port output, displacement, or zeros. The external X data may not be driven to the X mux outputs. Although the XMUX_SEL microinstruction field is used to control both the X operand mux and the X mux output, they are decoded differently. When the XMUX_SEL indicates external data, the operand register gets the external data but the X mux gets the register file read port. This allows the microcode to simultaneously select external data for the integer ALU and drive the register selected by XREG_SEL off-chip to the function units. This feature is used quite extensively in the intrinsic instructions (sin, cos, etc.). The Y mux outputs, YMUX_LDATA / YMUX_LPAR and YMUX_UDATA / YMUX_UPAR, are similar to their X mux counterparts. They are a little simpler in that since there is no external Y data, the only source for the Y mux is the register file Y read port.

The discussion of the operand data paths to this point has left out one major feature which complicates matters. Consider the situation in which a register in the register file is being written from the A, B, or C bus at the same time it's being read on the X or Y operand paths. It would be desirable for performance reasons to not waste time writing the data into the register file on one clock and then reading it out on the next. A technique called *bypass* is used to do just that. The X and Y internal and external operand muxes are widened to include the A, B, and C buses as possible selections. Logic is added to compare write controls from the A, B, and C buses to read controls for the X and Y operands. If this logic detects a match, data is *bypassed* to the operand muxes as it's simultaneously written to the register file. For example, if the NDC writes S0 from the C bus while the UIR1 operand access level attempts to read S0 to a function unit, data is written into the register file to S0 on the same clock it's written to the function unit's UIR2-level input operand register. The sequence of operations is the same as if there were no bypass - it just takes one less clock to complete. This is an important distinction - bypass is an optimization, not an integral function. Due to timing restrictions, all bypass combinations are not valid. Some basic

exceptions are:

- Function unit output (B bus) to input (X, Y mux) bypass occurs local to the function units. Data is **not** brought from the B bus, through the NRFA, and back to the function unit input. Instead, the NRFAs detect the bypass situation and asserts the B_X_BYPASS and/or B_Y_BYPASS control signals to the function units, which internally route input from their bidirectional B bus pins.
- To avoid parity errors caused by missing setup time requirements, X and Y mux bypasses are qualified to only occur when there is a function unit request at the UIR1 level. The NRFAs force all ones (data and parity) on the X and Y mux outputs if the UIR1-level staging of the FU_OP_REQ microinstruction field is zero.
- Some ALU operations are too slow to make setup from the A bus to the destination of the bypass. For example, the add operation will not bypass off-chip. The ALUFAST microinstruction field, staged to the UIR2 level, indicates whether the associated ALU operation will make timing to bypass off-chip. All ALU operations do bypass to the internal operand muxes, however.
- There is only partial implementation of bypassing from the upper word data path to the lower word data path. This type of bypass would occur if the A bus controls specify a write of a longword to T0 (implying the T1/T0 pair were to be written) and the UIR1 X port controls specify a read of T1. The NRFA-internal A bus is the only bypass source that does this. There are no upper C bus or upper B bus to lower X / Y port bypasses.
- For a bypass to occur, the size of the write operation must be greater than or equal to the size of the read operation. For example, a halfword write of A2 from the C bus would not bypass to a word read of A2 from the X port.

This list covers most exceptions, but probably not all. For complete details of bypass control, refer to the NRFA specification and ISP model. Bypassing is also performed to the Z mux output to the address generator, which is discussed in the next section. All previous discussion of bypass applies to the Z mux as well - it was omitted because the topic of the Z mux had not been introduced.

3.5.3 NDC Address Generation - the Z Mux

The third read port of the register file is the Z port. The bus from the NRFAs to the NDC is called the Z mux. It is used to supply base register data to the logical address generator in the NDC. For example, the Aj register of each macroinstruction is read out of the Z port and used as the base for logical address calculation. Recall that there are really only two direct address modes in the Convex architecture - absolute and indexed. For absolute mode, the instruction specifies zero in the Aj field indicating A0 is the address register. Instead of the true contents of A0, zero must be supplied so that the absolute address is obtained when the Z mux data is added to the displacement by the NDC. For indexed mode, the instruction specifies 1-7 in the Aj field indicating one of A1 through A7 is the address register. The NRFA supplies the true contents of A1 - A7 on the Z mux so that the indexed address is obtained when the Z mux data is added to the displacement by the NDC. The microcode has special encodings of the register selects (the ZREG_SEL, XREG_SEL, and YREG_SEL fields) to indicate this special mode is to be used.

The ZMUX_DATA (and associated odd byte parity ZMUX_PAR) outputs of the NRFAs may be driven by the Z read port of the register file, or by the A or C bus using the same bypass mechanism described in the discussion of the X and Y mux data paths. There is no path provided to bypass from the B bus to the Z mux.

The only difference between the Z mux and its X and Y counterparts is the level of the pipeline the access occurs. The X and Y operands are accessed at the UIR1 level of the NAS pipeline. The Z mux data is accessed at the address generator (AG) level, which is a hybrid of the UPC and UIR1 levels of the pipeline. The NDC employs a technique called *advanced effective address* generation, or *advanced effa* for short. The basic principle of advanced effa is this: the address for a memory (or communication register) request must be specified one cycle before the request. This gives the NDC a jump on the cache access which provides an overall performance boost.

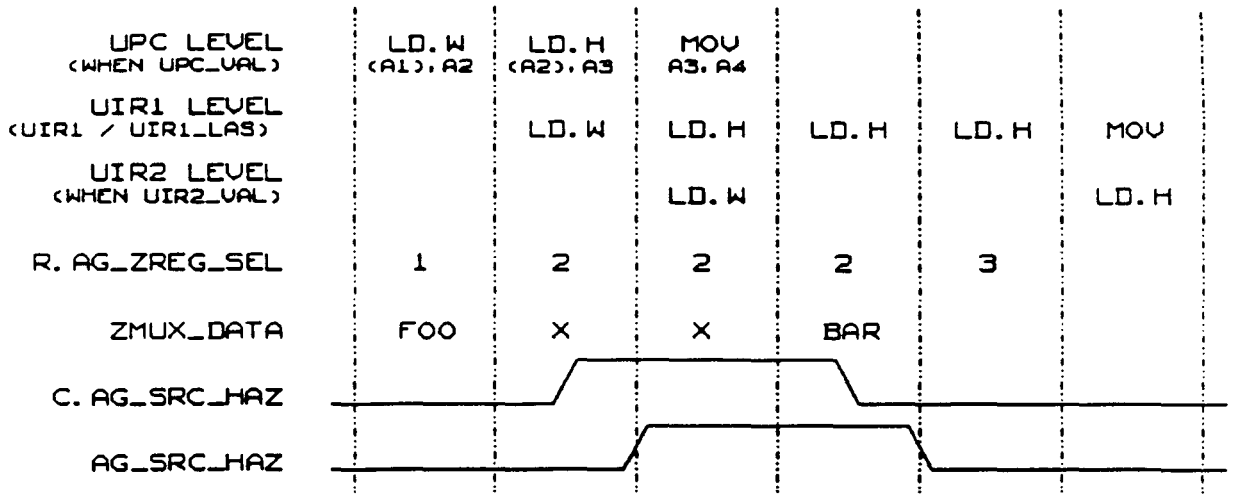
For single microinstruction macroinstructions (for example *ld.w effa,Ak*) or memory requests made on the first microinstruction of a multi-microinstruction macroinstruction, the base register is selected and read out onto the Z mux by the last microinstruction of the previous macroinstruction (or trap handler microroutine, etc.). This is implemented by requiring the microcode to select the advanced effa source in its AG_SEL field whenever a dispatch may occur. This is not actually coded in each microinstruction - the assembler forces it by default. The Z port in the register file is accessed using the next macroinstruction's dispatched register selects. The advanced effa encoding of the AG_SEL field enables the data read from the Z port to be added in the address generator's ALU to form the logical address. Of course, if the register accessed by the Z port has a pending hazard, the pipeline must stall and wait for the hazard to clear. This stalled AG-level access results in an *address generator source hazard*, which is discussed in section 3.6.2, beginning on page 3-24. The NRFA uses this hazard signal internally to hold its AG register selects, and registers it and sends it to the NDC. In the instance just discussed, the AG level of the pipeline appears to be equivalent to the UPC level in that it's one level before the UIR1 level for the single microinstruction. As an example of this kind of advanced effa access, consider the following instruction stream:

```
ld.w    (a1),a2
ld.h    (a2),a3
mov     a3,a4
```

Assume initially A1 contains "foo", A2 contains "x", and that at location "foo" the value "bar" is present. Assume also that all memory references hit the PTE and data caches. A timeline of this sequence is shown in Figure 3-8. The C.AG_SRC_HAZ signal shown is the NRFA-internal address generator hazard signal. AG_SRC_HAZ is the registered version the NRFA outputs to the NDC.

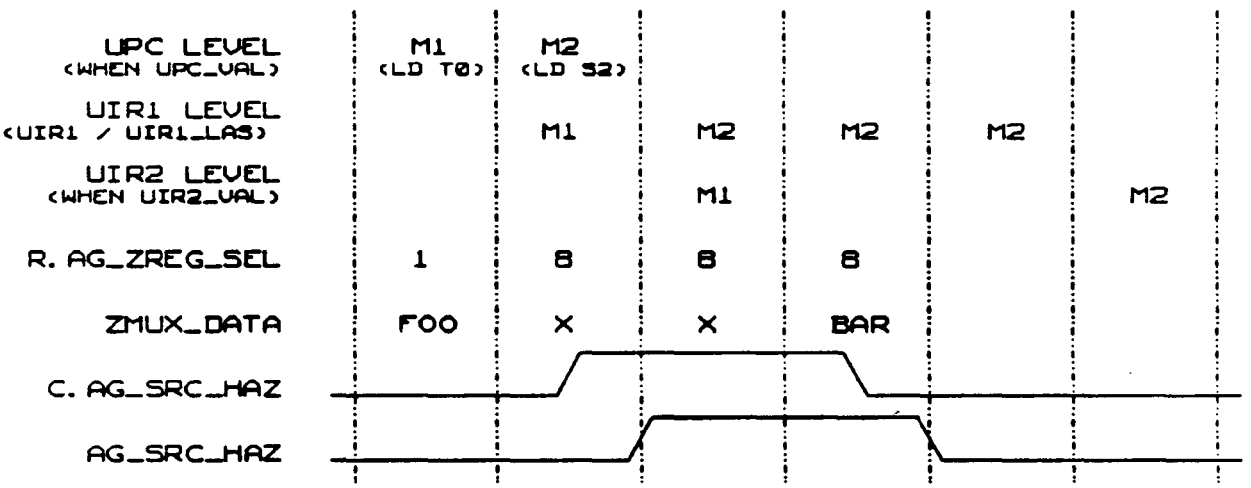
For multiple microinstruction macroinstructions, the microcode specifies the address generator source (with the AG_SEL field) on the microinstruction before it makes the memory request operation. Therefore the relative level of the AG pipeline is the UIR1 level of the microinstruction that specifies the AG_SEL value, or the UPC level of the microinstruction that specifies the memory request operation. As an example of this type of advanced effa, consider the two microinstruction sequence in the *ld.b @(a1),s2* microroutine. This code is discussed in the example section of the microcode section of this document (chapter 7). Basically, the first microinstruction uses the previous macroinstruction's selection of advanced effa to read the indirect pointer into the temporary register T0 (indicated with select value 8). It also specifies the result of this load as the AG_SEL for the second microinstruction. The second microinstruction uses the returned indirect pointer in T0, already set up on the Z mux by the first microinstruction, to read the data word into S2. This can be specified in only two microinstructions because the AG source hazard controlling the second microinstruction's memory request is asserted on the same clock as the second microinstruction reaches the UIR1 level. A three microinstruction sequence of load indirect pointer, select result as AG source, read data word would work the same, but

Figure 3-8: Advanced Effa Example 1



would waste a location in control store. Assume initially A1 contains "foo", T0 contains "x", and that at location "foo" the value "bar" is present. Assume also that all memory references hit the PTE and data caches. The timeline for the *ld.b @(a1),s2* sequence is shown in Figure 3-9. The first microinstruction is denoted M1, while the second is denoted M2.

Figure 3-9: Advanced Effa Example 2



M1's address reference uses advanced effa like the previous example. M2's reference has R.AG_ZREG_SEL equal to 8 the clock before it reaches the UIR1 level, so the AG level of the pipeline appears to be equivalent to the UIR1 level of the selecting microinstruction M1.

There is some rather complex logic in the NRFA array that maintains the AG level of the pipeline to always provide the address base register data on the Z mux one cycle before the memory request. It also generates the AG_SRC_HAZ signal that informs the NDC of a stall at the AG level of the pipeline. For more details of this advanced effa logic, refer to the NRFA array ISP model.

3.5.4 ALU

The main execution unit of the NAS is the integer ALU, located in the NRFA gate arrays. This 64-

bit ALU takes input from the X and Y operand registers at the UIR2 level, uses microinstruction fields staged to the UIR2 level as control, and produces the A bus at its outputs. Since the A bus may be bypassed off-chip on the X, Y, and Z muxes, parity is generated from the A bus and driven to the chip outputs. Each of the 4 NRFA arrays contains 16 bits of the ALU - 8 bits from the least significant word and 8 bits from the most significant word. The partitioning among the NRFAs follows the register partitioning detailed in section 3.5.1, beginning on page 3-11. The integer ALU performs the following classes of operations:

- add / subtract using a carry-lookahead adder
- logical (Boolean) operations
- type conversions
- stripmine function, i.e. reducing vector length to the range $0\text{-}128_{10}$
- ring function, i.e. returning the ring of a candidate address by mapping the segment bits (31..29) into a ring 0-4.
- "short" shift, i.e. logical shift of up to 7 bits of a 32-bit operand

The ALU is controlled by the ALUOP and ALUSIZE fields of the microinstruction. ALUOP describes the kind of operation to be performed and ALUSIZE controls the width of the ALU for the particular operation. ALUSIZE is not meaningful to all operations. The detailed encodings of these fields may be found in chapter 7.

Internally, each NRFA has a separate upper (most significant) and lower (least significant) ALU and A bus path. The stripmine, ring, and shift functions are not included on the upper ALU path since they only work on 32-bit operands. The upper word of the A bus output for these operations is forced to be zero.

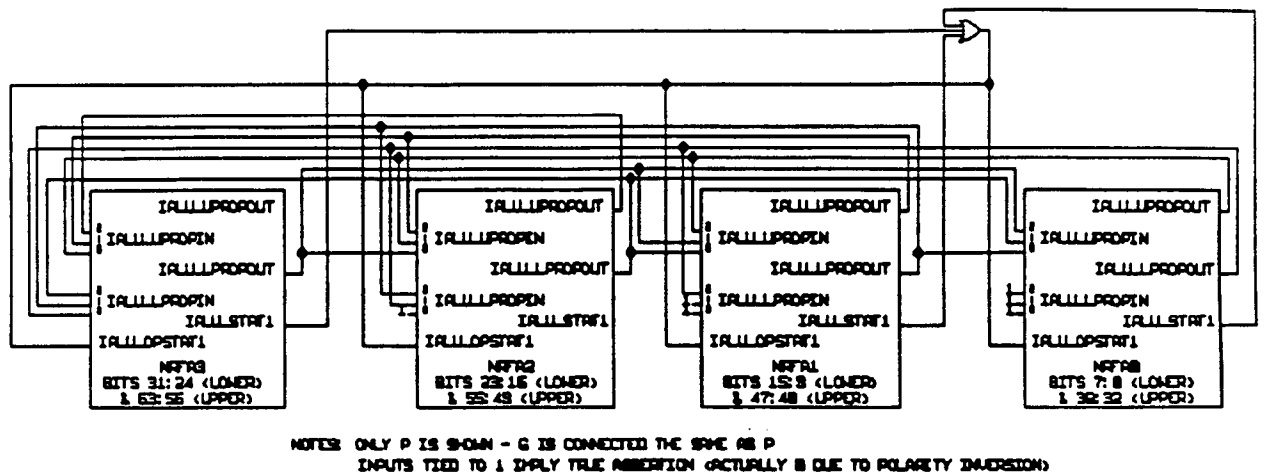
With the exception of the logical operations and the ring function, all ALU operations require communication between NRFA slices to complete their computations. For example, a full 64-bit add may generate a carry in the lower byte (bits 7..0 in NRFA0) that ripples through to the upper byte (bits 63..56 in NRFA3). A complex trade-off between timing and pincount restrictions on the NRFA resulted in the following set of multi-use inter-chip communication signals. They are named for their function in the adder operation, but are used for different purposes for convert, stripmine, etc.

- IALU_LPROPOUT - "propagate" signal from the lower byte of the ALU
- IALU_UPROPOUT - "propagate" signal from the upper byte of the ALU
- IALU_LGENOUT - "generate" signal from the lower byte of the ALU
- IALU_UGENOUT - "generate" signal from the upper byte of the ALU
- IALU_STAT1 - result sign output of the upper or lower byte of the ALU, depending on ALUSIZE
- IALU_LPROPIN<2..0> - "propagate" inputs from the other 3 bytes of the lower ALU
- IALU_UPROPIN<2..0> - "propagate" inputs from the other 3 bytes of the upper ALU
- IALU_LGENIN<2..0> - "generate" inputs from the other 3 bytes of the lower ALU
- IALU_UGENIN<2..0> - "generate" inputs from the other 3 bytes of the upper ALU
- IALU_OPSTAT1 - result sign input - ORed together IALU_STAT1 outputs from all 4 NRFAs

For shift operations, each NRFA has the IALU_SHIFTIN<6..0> input which is connected to the

YBUS_LDATA<7..1> outputs of its next least significant neighbor NRFA. To compute overall ALU status (overflow, carry, etc.) information, the PROPOUT and GENOUT outputs of all NRFAs are connected the NPSW array. In addition, each NRFA produces an IALU_STAT0 output signalling byte overflow which is also connected to the NPSW. This signal is registered on the NRFA before it is output, which adds an extra cycle of latency, effectively making the NPSW's overflow calculation a UIR3 level operation. A block diagram showing the interconnection of the NRFAs to form the ALU is given in Figure 3-10.

Figure 3-10: NRFA ALU Interconnection



Each NRFA has an internal scannable position indicator register (R.POSITION) which tells it where it is installed in the slice ordering. For example, the initialization of this register to 01_2 makes a generic NRFA part act like NRFA1 in all previous discussions. This position register is used to interpret the PROPIN, GENIN and OPSTAT1 inputs relative to the particular NRFA's position in the ALU.

3.5.4.1 Add and Subtract Operations

For add and subtract operations, the interconnections must be used by each slice to determine what the result of the operation is. Although all propagate (P) and generate (G) outputs are connected to the NPSW for overall status computation, each slice must know what the carry in to its upper and lower ALU is in order to produce the correct A bus result data. Each byte's carry in can be affected by the carry in to the entire 64-bit ALU (for add with carry or subtract operations) or by and carry generated by a less significant byte and propagated up the tree. To implement this, a full 8-byte carry-lookahead adder (CLA) tree is included in the generic NRFA design. The P and G inputs to the NRFA chips are interconnected (as shown in Figure 3-10) to make each slice appear as the upper byte of a full tree. Unused P inputs are tied high and unused G inputs are tied low to fill out the tree allowing the base ALU carry in to propagate up to the most significant byte if it is asserted. Then the byte 3 carry in calculated in the CLA tree can be used as the carry in for the lower byte ALU. Similarly, the byte 7 carry in can be used as the carry in for the upper byte ALU. This allows the CLA logic to be independent of slice position, which saves gates in the NRFA design. Note that the physical implementation inverts the polarities of the P and G signals. For example, a propagate is asserted on the PROPOUT outputs with a logic zero. The PROPIN inputs on the NRFA and NPSW interpret this negative polarity, only for add / subtract operations. Other uses of the P / G signals are positive polarity. The IALU_STAT0 output asserts byte overflow for the upper byte if ALUSIZE indicates longword, or lower byte overflow otherwise. The NPSW

receives these STAT0 outputs and decides which NRFA's output to use based on ALUSIZE. The IALU_STAT1 output value depends on position. If the NRFA is the most significant of the operand (for example NRFA1 for halfword ALUSIZE), the result sign is driven out. Otherwise, zero is driven so that when the 3 IALU_STAT1 outputs (NRFA0, NRFA1, NRFA3 - NRFA2 is never most significant for any data type) are ORed, the resultant IALU_OPSTAT1 contains the result sign.

3.5.4.2 Logical Operations

For logical operations (AND, OR, etc.) the IALU_LPROPOUT is used to assert an A bus result equal zero condition, which is used in the NPSW. All NRFAs' PROPOUTs are combined to produce an overall result zero flag. IALU_STAT1 outputs the byte sign in the same way add operations do. The IALU_LGENOUT and IALU_STAT0 outputs are not used.

3.5.4.3 Type Conversion Operations

There are two classes of integer type conversions. "Up" conversions convert a smaller data type to a larger one. An example of an up conversion would be byte to word. The NRFAs implement 3 up conversions - byte to word, halfword to word, and word to longword. "Down" conversions are the opposite - a larger data type converted to a smaller one. An example of a down conversion would be word to byte. The NRFAs implement 3 down conversions - word to byte, word to halfword, and longword to word. Both classes of conversions work on the X operand register.

Up conversions are basically sign extensions. The IALU_OPSTAT1 input tells each NRFA what the operand sign is. If the position indicator shows the NRFA to be more significant than the input size, the sign (IALU_OPSTAT1) is extended to 8 bits and placed on the A bus. If the NRFA is less significant than the input size, the input is passed to the A bus. P and G outputs are computed like down converts (described in the next paragraph) but are not used.

Down conversions are pass operations with overflow calculation. The NRFAs pass the X input to the A bus as the data result. All of the NRFA status outputs are used by the NPSW to determine if a down convert overflowed. IALU_OPSTAT1 contains result sign, which is the same as the input sign for converts. The NRFA IALU_STAT0 outputs contain the most significant bit of the result (upper ALU for result type longword, lower ALU otherwise). The P outputs indicate whether the associated byte is nonzero. The G outputs indicate whether the associated byte is not equal to FF_{16}. For example, for word to byte conversion, if the input operand is positive, there is an overflow if any of the 3 most significant bytes are nonzero (P) or if the most significant bit in the result is set (STAT0). The G outputs are used similarly for negative operands. The NPSW uses ALUOP and ALUSIZE (which identify input and result size) to decide which NRFAs' outputs are relevant to the computation.

3.5.4.4 Stripmine Operation

The stripmine function clips the input X operand to the range $0\text{-}128_{10}$ ($0\text{-}80_{16}$). The upper three slices (NRFA1-NRFA3) must communicate to the lower slice (NRFA0) whether they contain any significance for the input operand. The LPROPOUT output is asserted if the input byte is nonzero. IALU_OPSTAT1 contains the operand sign (ORed from the NRFA IALU_STAT1 outputs, which is driven with the sign by NRFA3 and with zero by NRFA2-NRFA0). NRFA0 uses the operand sign and its IALU_UPROPIN signals (which are connected to the LPROPOUT signals from NRFA3-NRFA1) to determine the A bus output. If IALU_OPSTAT1 indicates a negative operand, zeros are driven to the A bus. If the operand is positive, it is passed unless one of the IALU_UPROPIN inputs is set (indicating significance in NRFA3-NRFA1) or the MSB of NRFA0's operand input is set (indicating the input is in the range $81_{16}\text{-}FF_{16}$) in which case the A bus is forced to 80_{16}.

3.5.4.5 Ring Operation

The ring operation takes a single input from the X operand register and returns the ring number in bits 31..29 and zeros everywhere else. The position indicator is used to enable the appropriate slice-dependent logic in the NRFA chips. NRFAs 0, 1, and 2 force zeros to the A bus output. NRFA3 forces zeros on the least significant 5 bits of the A bus. If the most significant bit is set, the segment is 4-7, so the ring is 4 and the upper 3 bits of the A bus are forced to 100_2. If the MSB is clear, the upper 3 bits of the input are passed to the A bus. All NRFAs drive zeros on their upper A bus paths. Each NRFA decodes the

3.5.4.6 Shift Operation

The integer ALU includes a shift operation to implement the short immediate shifts in the Convex architecture. These shifts are left shifts with a 3-bit shift amount, making the longest shift 7 bits left. The Y operand register contains the word to be shifted. The least significant 3 bits of the X operand register contain the shift amount. Since the shift can be at most 7 bits, each NRFA requires 7 bits of shift input data from its next least significant neighbor. This is provided by connecting the most significant 7 bits of the YBUS_LDATA output of the neighbor to the IALU_SHIFTIN<6..0> input. NRFA0's IALU_SHIFTIN inputs are connected to zeros. The shift amount is communicated from NRFA0 to the other NRFAs using the IALU_STAT1, IALU_LPROPOUT, and IALU_LGENOUT outputs. The other 3 NRFAs use the position indicator to know where NRFA0's P and G outputs appear in their P and G inputs, following the connection diagram in Figure 3-10.

3.5.5 B and C Bus Writes

There are two write ports to the register file that are sourced externally to the NRFA arrays. These are the B bus write port, used to store function unit results, and the C bus write port, used to store memory and communication register operation results.

The B port is driven by the wire-ORed bidirectional outputs of the function units. Each NRFA receives a byte of the lower word on its BBUS_LDATA inputs and a byte of the upper word on its BBUS_UDATA inputs. Odd byte parity is supplied on BBUS_LPAR and BBUS_UPAR. Data is supplied directly to the register file write muxes within the NRFA, and is also registered in the NRFA for parity checking. The NRFAs generate B port write controls internally in response to assertion of the FU_RSLT_EN input with the associated function unit encoded on FU_RSLT_SEL. The register selects and size are removed from a queue in the NRFAs where they were stored when the function unit request was made.

The C port is driven by the NDP gate arrays in the NDC subsystem. Each NRFA receives a byte of the lower word in its CBUS_LDATA inputs and a byte of the upper word on its CBUS_UDATA inputs. Odd byte parity is supplied on CBUS_LPAR and CBUS_UPAR. Register writing and parity checking is implemented similar to the B port, except write controls are provided externally from the NDC subsystem on the CBUS_WR_EN, CBUS_SIZE, and CBUS_REG_SEL inputs to the NRFAs.

3.6 Hazards

When a resource is accessed that is unavailable, a condition known as a *hazard* occurs. The function of hazards in the NAS pipeline was discussed in section 3.2, beginning on page 3-1. In this section, the logic that implements these hazard checks is discussed. Almost all hazard checking is performed at the UIR1 level of the pipeline. The logic to check for these hazards

resides in the NRFA gate arrays. Each NRFA maintains a separate copy of the hazard logic and register scoreboard, which it uses to compute the pipeline control signals (UIR1_VAL, UIR1_LAS_FULL, UIR2_VAL, etc.). All 4 NRFAs should have the same value on these outputs at all times. The different NRFA outputs are used as fanout copies of the control signals. There are two major classes of hazards, which are discussed individually in separate subsections.

3.6.1 Source Hazard

The first type of hazard is the UIR1-level source hazard. The source hazard signal internal to the NRFA holds the UIR1 level registers in the NRFA and influences the UIR1_OVR_FULL, UIR1_VAL, and UIR2_VAL pipeline control signals. It is registered directly and output as UIR1_LAS_FULL, which informs the rest of the NSP of the hazard condition the clock after the NRFAs realize it. The following events cause a source hazard when they occur simultaneously with UIR1_VAL. The first list contains topics which have already been discussed, so they are mentioned briefly with a cross-reference.

- A microsequencer test hazard, which was discussed in section 3.4.3, beginning on page 3-8.
- A microinterrupt causes a hazard for one clock until the pipeline can be flushed.
- A branch restart causes a hazard for one clock until the pipeline can be flushed. Branch restarts are discussed in detail in section 3.14.3, beginning on page 3-39.
- An address generator (AG) source hazard causes a source hazard one clock after the AG hazard occurs. AG source hazards are discussed in detail in section 3.6.2, beginning on page 3-24.

The second list of source hazard contributors have not yet been mentioned, and may be discussed in more detail in following subsections:

- A branch hazard, i.e. when an instruction is tagged with a branch on a condition (AC or SC) with a pending hazard.
- An access to a register that is not currently valid.
- A wait hazard, i.e. a microcode-selected wait condition.
- A function unit hazard, i.e. an attempt to initiate a function unit operation with a busy function unit.
- An access to a carry bit (PSW or the microarchitecturally defined USW) that is not currently valid.
- A data cache hazard, i.e. an attempt to initiate a memory request when the NDC subsystem is busy. This is indicated by the NDC's assertion of the DC_HAZ input to the NRFA.

3.6.1.1 Branch Hazard and Branch Tagging

The NIP employs a branch prediction strategy that is implemented by making the execution of the instruction following the branch conditional. In essence the NIP says "I got to this instruction assuming that AC (or SC) was set (or clear). Restart me if I was wrong in my assumption." Each instruction is *tagged* with a branch condition, which may be AC, SC, or unconditional. This is specified on the UPC_BR_SEL input to the NRFAs. In addition, the condition may be qualified to be true or false with UPC_BR_POL. The NRFA detects a branch hazard when the microinstruction at the UIR1 level is tagged with a branch on AC or SC and there is a pending hazard on it, indicated with the ACPEND or SCPEND inputs from the NPSW array to the NRFA. This hazard

must remain active until the specified condition is valid, at which time the branch restart decision is made.

3.6.1.2 Register Access Hazard

Perhaps the most common source hazard is a register access hazard. An access in this sense means the desire to read or reserve for future write. The NRFAs contain a *scoreboard* which consists of a "reserved" bit for each of the 32 registers in the register file. When a backdoor operation (memory read, function unit operation, etc.) is initiated, this scoreboard is set by the UIR2 level of the pipeline, using the BDOP, BDSIZE, and BDREG_FMT fields of the microinstruction. BDREG_FMT specifies the register select from XREG_SEL, YREG_SEL, or ZREG_SEL, which controls which scoreboard bit to set. Register accesses occur at the UIR1 level of the pipeline. They are specified with the UIR1 level versions of the aforementioned reservation controls or the XREG_SEL, XREG_FMT, and XREG_HAZ fields (or their Y and Z counterparts).

When the microcode attempts either type of UIR1 level access, the UIR1 control fields (read or reserve) are used to read the scoreboard and insure there isn't already a hazard on the register. Since the scoreboard is set from the UIR2 level of the pipeline, hazard logic at the UIR1 level must also check that the UIR2 level isn't currently setting a hazard on the register. Finally, the UIR2 level of the pipeline may be writing the register from the A bus, so UIR1 control fields are compared to UIR2 level A bus controls to cause a hazard if this is the case. This is complicated slightly by the write bypass mechanism, discussed in section 3.5.2, beginning on page 3-13. When an incoming write bypasses the register file directly to the operand register, the hazard logic must detect this so that the hazard is kept consistent with the data path. All register access hazard equations therefore have terms in them to comprehend bypasses on any write port.

3.6.1.3 Wait Hazard

A *wait hazard* is a microcode selected condition that is input directly to the source hazard logic in the NRFAs. An example of a wait hazard would be waiting for the vector processor to be idle. The vector processor asserts the signal VP_SP.IDLE whenever it is idle. Rather than waste control store space and incur the latency of a microcode spin loop, this condition is waited on by feeding it into the source hazard logic. There are a number of different wait hazard conditions available - refer to the microcode documentation (chapter 7) for the complete list. Wait conditions are selected within the NPSW gate array using a range of encodings of the TSEL microinstruction field. Due to pin count restrictions in the NRFA, the selected wait condition is combined with the function unit hazard and carry hazard signals (also generated in the NPSW array) to produce the WFC_UIR1_HAZ input to the NRFAs. Due to timing restrictions, UIR1_LAS_FULL and UIR2_VAL are factored out of the NPSW outputs to produce the following 4 signals:

- WFC_U1_U2VAL - WFC hazard if UIR1_LAS_FULL = 0 and UIR2_VAL = 1
- WFC_U1LAS_U2VAL - WFC hazard if UIR1_LAS_FULL = 1 and UIR2_VAL = 1
- WFC_U1_U2INV - WFC hazard if UIR1_LAS_FULL = 0 and UIR2_VAL = 0
- WFC_U1LAS_U2INV - WFC hazard if UIR1_LAS_FULL = 1 and UIR2_VAL = 0

These signals are actually active low but are listed in true form here. These 4 terms are combined externally to the gate arrays with an ECLinPS mux selected with UIR1_LAS_FULL and UIR2_VAL, producing WFC_UIR1_HAZ.

3.6.1.4 Function Unit Hazard

The NSP's 4 function units act as asynchronous execution units for the NAS. To control flow through the function units, when an operation is initiated it must be determined that the function unit is not currently busy working on a previous operation. The logic to implement this function is contained in the NPSW array, and is discussed in more detail in section 3.7.2, beginning on page 3-25. Basically, when the UIR1 function unit request fields are active, the current state of the function units is checked. If the requested function unit is busy, WFC_UIR1_HAZ is asserted at the NRFA input, which stops the UIR1 level microinstruction until the function unit is ready to take the request. The function unit hazard signal is combined with the wait hazard signal discussed in the previous section and the carry hazard signal discussed in the next section.

3.6.1.5 Carry Hazard

The NPSW maintains a scoreboard for the "carry bits" (AC and SC in the PSW, IC, JC, and KC in the USW) much like the scoreboard for the register file. This is used to maintain hazards on the carry bits, so that when a UIR1 level microinstruction attemps an access to a carry bit that has a pending hazard, the pipeline stalls. The NPSW carry hazard logic is discussed in more detail in section 3.8.4, beginning on page 3-29. The output of this logic is combined with the wait and function unit hazards discussed in the previous two sections to form the WFC_UIR1_HAZ input to the NRFAs, which feeds directly into the source hazard logic.

3.6.2 Address Generator Source Hazard

The second major type of hazard is the address generator source hazard, which is similar in nature to the generic source hazard except it occurs at the address generator (AG) level of the NAS pipeline. The AG pipeline level was introduced in section 3.5.3, beginning on page 3-15. The AG hazard is computed in the NRFAs and registered. The register input is used within the NAS to hold AG pipeline level registers. The registered version is driven off the NRFAs (as AG_SRC_HAZ) to the NDC subsytem, to inform it of the piepine stall. The registered version also causes a UIR1 level source hazard.

Unlike generic source hazards, there is only one cause of AG source hazards - accesses to registers with pending hazards. The scoreboard is used to check an access for a hazard, much like the generic source hazard. However, the AG level register selects are used instead of the UIR1 level selects that were used for the generic source hazard. These AG register selects are used to read the scoreboard to check for an existing hazard. Since the scoreboard is set from the UIR2 level of the pipeline, hazard logic at the UIR1 level must also check that the UIR2 level isn't currently setting a hazard on the register. The AG level register selects combine with the UIR1 level BDOP and BDSIZE fields to define the access in question. Finally, the UIR2 level of the pipeline may be writing the register from the A bus, so AG register selects are compared to UIR2 level A bus controls to cause a hazard if this is the case. Bypass affects the AG source hazard in the same way it affects generic source hazards.

3.7 Function Units (NFU)

The NFU (Neptune Function Units) section of the NAS subsystem performs all arithmetic functions that the integer ALU does not. It is made up of four function unit chips, 3 of which are Vitesse custom ASICs and one of which is a 30K Vitesse gate array. These parts and their basic function are:

- NFAD (custom) - floating point addition, subtraction, and comparison
- NMUL (custom) - integer and floating point multiplication

- NDIV (custom) - integer division, floating point division and square root
- NMISC (30K array) - floating point to/from integer type conversion, leading ones position, trailing zero count, shift, population count, and integerization of float

The NFU is shown in the upper center of the NSP Block Diagram. Input operands are driven by the NRFA arrays on the 25-ohm output XMUX_DATA and YMUX_DATA busses, with associated odd parity on XMUX_PAR and YMUX_PAR. These inputs set up to UIR1 level operand registers in the function unit part, from which parity is checked. Microcode controls such as opcode and size are presented on the next clock and set up to UIR2 level opcode registers in the function unit. The function unit part executes the opcode in a variable number of clocks (depending on operation) and places the result in an internal register. The NFAD produces results in one clock. Other function units assert a "result ready next clock" signal when the operation is complete. The output register is driven on 25 ohm bidirectional pins from the function unit onto the B bus (BBUS_DATA with associated odd parity BBUS_PAR), which is connected to the NRFA arrays for writing the register file. Under control of the B_X_BYPASS and B_Y_BYPASS signals from the NRFA, the function units use the bidirectional BBUS_DATA pins as inputs to route the result back to the X and/or Y operand input registers.

Function unit operations may cause exception conditions such as divide by zero, overflow, etc. to occur. The arithmetic exception outputs from the function units are ORed together along with exception bit set signals from the vector processor in ECLinPS OR gates and driven to the NPSW array to set the appropriate PSW bit(s).

3.7.1 NFU Request Generation

Function unit requests are initiated by the microcode. The FU_OP field (an 8-bit subfield of the MFP_OP field, see chapter 7 for the complete microword definition) of the microinstruction encodes the type of operation desired. The FU_OP_REQ field qualifies this as a valid request. Part of the FU_OP field (bits 5..0) are taken by the NUS array as part of is US_MFP_OP input, staged to the UIR1 level, and driven to the NFU as opcode bits for the function units. The function units receive UIR2_VAL on the following clock to qualify the UIR1 level request they just received. Other parts of the FU_OP field are taken by the NPSW array for use in generation of function unit controls. Bits 7..6 encode the function unit desired (NFAD,NMUL, NDIV, NMISC). Bits 3..2 are used for NFAD operations to determine if the operation is a compare or not, which controls whether a status bit or full floating point result is expected. The NPSW also receives the BDSIZE field, which along with FU_OP bit 0 is used to decide if an NMUL operation is a short or long operation. Shortness refers to the number of cycles required to complete the operation. Byte, halfword, word, and single float are short operations. Longword and double float are long operations. The NPSW uses the information that the operation is short in its determination of function unit hazards.

3.7.2 NFU Hazards

After an NFU request is made, the NAS must retain information about the request to cause hazards due to incomplete operations and to control result storing when the operation completes. This section discusses how hazards are maintained. When an NFU request is made, microcode reserves a register for the return data (except for float compares, discussed later). This is controlled with the BDOP, BDREG_FMT, and BDSIZE fields of the microinstruction. The NRFA scoreboard bit associated with the selected register is set, indicating a hazard. If a later microinstruction accesses this register before the function unit operation completes, a hazard stall occurs as described in section 3.6.1.2, beginning on page 3-23. Float compare operations do not

require a register for the result. Instead, a carry bit is reserved for status returned from the function unit part. Hazards on carry bits are discussed in section 3.6.1.5, beginning on page 3-24.

The other type of NFU hazard is caused by making a request to a busy function unit. The NPSW array decodes parts of the FU_OP field, as mentioned earlier. This is used, along with the completion signals from the function units (NDIV_RDY_NEXT, NDIV_RSLT_NEXT, etc.) to determine if a function unit is busy. The NDIV asserts NDIV_RDY_NEXT if it can take an input request on the next cycle. The NPSW receives this signal and uses it to aid in determination of the busy status of the NDIV. At one time, the NMUL required this type of control, so the NPSW has an NMUL_RDY_NEXT input. Later in the design cycle it was determined the NMUL is always ready for an input request, so this input to the NPSW is tied high. If a request is made to a busy function unit, the NPSW determines this and asserts one of the WFC hazard output lines, which after external combination with UIR2_VAL and UIR1_LAS_FULL, results in the assertion of WFC_UIR1_HAZ to the NRFAs, which causes a source hazard, stalling the requesting microinstruction.

3.7.3 NFU Result Storing

When the function unit operation is complete, result data is stored to the register file via the B bus. This section describes the control used for the result transfer.

Since the 4 function units share the B result bus, it is possible to two function units to attempt to drive the bus simultaneously. To prevent this, results must be taken from them in a prioritized manner. The "winner" must be instructed to drive the bus and the "losers" must hold their result data and potentially stop taking input requests. The priority ordering given the function units is:

1. NFAD - highest priority
2. NMUL
3. NDIV
4. NMISC - lowest priority

Since the NFAD takes one clock to complete all operations and has highest priority, it always drives the B bus immediately. The other function units assert the desire to drive the B bus on the next clock by asserting a RSLT_NEXT signal (NMUL_RSLT_NEXT, NDIV_RSLT_NEXT, NMISC_RSLT_NEXT). These outputs are driven three places. The NPSW receives them to determine busy status of the function units. A PAL receives them along with a UIR2 encoding of the FU_OP field and performs the prioritization. This PAL asserts one of four RSLT_SEL signals (NFAD_RSLT_SEL, NMUL_RSLT_SEL, NDIV_RSLT_SEL, and NMISC_RSLT_SEL) to the function units, selecting the winning output onto the B bus. A second PAL receives the RSLT_NEXTs and encodes a 2-bit result selector code FU_RSLT_SEL, implementing the priority ordering listed above, as well as an enable signal, FU_RSLT_EN, to qualify it. It also generates NFAD_RSLT_OE, which is used to enable the NFAD result onto the B bus. The other function units use the RSLT_SEL input to do this. The NFAD requires 2 inputs since it performs two types of operations - compare and non-compare. The FU_RSLT_SEL and FU_RSLT_EN signals are driven to the NRFA arrays to control writing of the register file. The NRFA contains a 4 element queue for each of the function units except the NFAD. Since NFAD operations complete in one cycle and are taken immediately, the UIR2 level stagings within the NRFA of the selects and size are used to write the result. For NMUL, NDIV, and NMISC operations, when the microcode reserves the result register in the scoreboard, the register select and result size are pushed in the appropriate queue (decoded from the FU_OP field). These queues are popped when result data

is taken from the function unit, under control of the FU_RSLT_EN and FU_RSLT_SEL inputs.

The other type of function unit result is status - either the result bit from an NFAD compare, which replaces the register-written result of non-compare operations, or an exception bit, which is written to the PSW in conjunction with the register write of the data result. For compares, the PALs in the NFU assert NFAD_RSLT_SEL but not NFAD_RSLT_OE, which tells the NFAD the status result is being accepted, but does not enable the NFAD to drive the B bus. The NFAD_STATUS output of the NFAD is driven to the NPSW and included in its SC write logic. Exception outputs of the function units are ORed together externally by type (e.g. NFAD_OV, NMUL_OV, NDIV_OV, and NMISC_OV) along with exception flags from the NVP (e.g. VP_SP.SET_OV) to form a single input per exception bit to the NPSW (e.g. FU_SET_OV). These exception inputs are included in the NPSW's PSW write logic.

Timing diagrams for typical NFU operations can be found in the function unit chip specifications.

3.7.4 NFU Operation Times

This section presents the execution times for the various function unit operations. The NDIV includes a power of 2 optimization which provides much faster results when the divisor is a power of 2. This option is available for floating point division only. Also, the NDIV includes an internal pipeline stage that permits a clock of overlap in back-to-back divide operations, effectively making the second divide operations a clock shorter than the first. The execution times referred to are measured from the function unit's input registers (UIR2-level registering of X and Y operands) to output register. It takes one more clock to write the result into the register file assuming the function unit is immediately selected to drive the B bus. Refer to the microcode section of this document (chapter 7) for a complete breakdown of all function unit operations by the part that performs them.

Figure 3-11: Function Unit Execution Times

NFAD	all operations	1
NMUL	integer byte multiplication	2
	integer halfword multiplication	2
	integer word multiplication	2
	integer longword multiplication	3
	single precision multiplication	2
	double precision multiplication	3
NDIV	integer byte division	4
	integer halfword division	6
	integer word division	10
	integer longword division	18
	float single division	9
	float double division	16
	float single power of 2 division	2
	float double power of 2 division	2
	float single square root	9

	float double square root	16
NMISC	all operations	2

3.8 Program Status Word (PSW)

The architecturally defined Program Status Word, or PSW, is implemented entirely within the NAS subsystem, primarily within the NPSW gate array. The PSW contains programmer-visible exception bits to record events such as arithmetic overflows, etc. It also includes trap enable bits for the various classes of exception bits, which allow the programmer to ignore the occurrence of these exception events. Also included are the two programmer-testable status flags, AC and SC, which are used to control conditional branches. Finally, it contains mode bits which control operation of the machine. For example, the sequential (SEQ) bit causes each instruction to complete before the following instruction begins execution.

The PSW is physically implemented as a 32-bit register in the NPSW array. It is readable by microcode through the NPSW's pre-X mux data path, described in section 3.13.1, beginning on page 3-36. Many of the individual bits of the PSW (and USW, described in the next section) are driven off the NPSW directly or in a Boolean combination for use by the rest of the NSP as microsequencer test conditions, trap indicators, or branch instruction conditions.

For write operations, the PSW may be considered to have 2 parts - carry bits (AC and SC in the PSW along with IC, JC, and KC in the USW) and non-carry bits. Each of these two classes of PSW bits have different types of write operations available. Writing of non-carry bits is discussed in section 3.8.3, beginning on page 3-29. Writing of carry bits is discussed in section 3.8.4, beginning on page 3-29.

3.8.1 Microprogram Status Word (USW)

The Microprogram Status Word, or USW, is a logical extension of the PSW for the NAS microengine. The USW is a 3-bit register in the NPSW array, consisting of (from most to least significant) the IC, JC, and KC bits. These three bits are similar in function to the AC and SC bits in the PSW in that they can be used as microsequencer branch conditions and can receive status results from the NFU and NCU. They are to the PSW as the T registers are to the register file - microcode temporary storage. In addition, IC can be selected as input carry to the integer ALU for implementation of multi-precision operations.

3.8.2 Arithmetic Overflow and Carry Generation

When an operation is performed in the integer ALU, status results such as carry and overflow may be loaded into the PSW or USW. The NPSW array therefore contains logic to compute all status outputs for the integer ALU. The NPSW receives all the propagate, generate, sign, and status outputs of the NRFA slices and uses them to compute status. These signals were detailed in the description of the integer ALU in section 3.5.4, beginning on page 3-17. The NPSW contains a full carry lookahead tree and logic to support the special ALU functions, just like the NRFA. The ALUOP and ALUSIZE fields are used by the NPSW in the same manner as the NRFA to decode the operation and generate the following status results:

- ALU carry-out
- complement of ALU carry-out

- ALU overflow
- X equal Y comparison status
- X less than Y comparison status
- X less than or equal to Y comparison status

Most of these results are generated using UIR2 level controls. The NRFA uses the UIR2 level X and Y operand registers and the UIR2 stagings of the ALUOP and ALUSIZE fields to generate the P, G, and STAT outputs, which are driven to the NPSW. The NPSW uses UIR2 level ALUOP and ALUSIZE fields to interpret these inputs and generate status to be written in the PSW or USW on the next clock. The exception to this is the overflow calculation. In order to make timing, the NRFA registers its IALU_STAT0 output before driving it off-chip. For add operations, IALU_STAT0 represents the overflow of the byte add. The NPSW uses the UIR2 level ALUSIZE field staged one register level (often called UIR3) to determine which NRFA's IALU_STAT0 is the most significant for the operation and uses it as overall overflow. For converts, IALU_STAT0 contains operand size and is therefore only part of the overflow calculations. The NPSW registers the other P / G / STAT inputs to synchronize them with IALU_STAT0, and uses UIR3 level control fields to determine overflow. The next two sections describe how the various status bits are loaded into the PSW or USW.

3.8.3 Non-Carry Bit Manipulation

The first class of PSW bits are those other than the carry bits. They are manipulated directly under control of the microcode, using the UIR2 staging of the PSW_OP field. This permits the following operations:

- load the integer ALU overflow status into the AIV or SIV bit. This operation adds an extra level of staging after UIR2 to make timing, as described in the previous section.
- explicitly load the FRL bits with 01_2, 10_2, or 11_2
- clear the bits that need to be zeroed (AC, AIV, ADZ, FRL, SC, SIV, SDZ, UN, OV, RO, FDZ, FIN) for the *call* instructions
- load all 32 bits in parallel from the Y bus

The exception bits (AIV, ADZ, SIV, SDZ, UN, OV, RO, FDZ, FIN, and IEC) can be set by the NVP or NFU with the assertion of the FU_SET_AIV, etc. inputs to the NPSW array. FU_SET_FSN is asserted for square root of negative and is decoded internally to the NPSW to set FIN = 1 and IEC = 000_2.

3.8.4 Carry Manipulation, Reservation, and Hazards

The second class of PSW bits are the carry bits - AC and SC, along with IC, JC, and KC in the USW. These bits may be loaded directly from a number of sources, and also may be reserved for future write using a scoreboard and set of queues similar to the register file. Three microinstruction fields, staged to the UIR2 level of the pipeline, control this operation. CRY_OP defines whether the operation is to be a direct load or a reservation for future write. For direct loads, CRY_SRC controls the input for the load. These sources are the various carry and status outputs of the integer ALU. For reservation operations, CRY_SRC specifies the status source for the return write - either NFU (for float compares) or NCU (for communication register operations). In either case, CRY_DST specifies which PSW or USW carry bit is the destination of the operation.

For direct load operations, the UIR2 level microinstruction fields directly gate the outputs of the

carry-lookahead adder tree in the NPSW into the selected PSW or USW bit.

For reservation operations, the NPSW includes two queues - one for NCU operations and one for NFU operations. If the UIR2-level CRY_OP field specifies a reservation, the UIR2-level CRY_DST field is pushed into the queue selected by the UIR2-level CRY_SRC field. The scoreboard bit for the selected carry bit is set and the reservation is complete. The scoreboard bits are R.LAC, R.LSC, R.LIC, R.LJC, and R.LKC. When status is returned from the NCU (detected by the assertion of the XCL_SP.CU_STATUS_EN input to the NSP), the enable and returned status (XCL_SP.CU_STATUS) are registered on the NSP. On the next clock, these registered values are input to the NPSW array and used to write the PSW. The XCL_CU_STATUS_EN input to the NPSW pops an entry from the CU queue, which is used to select which carry bit to write the XCL_CU_STATUS input to the NPSW into. The FU queue is handled similarly except that the NPSW determines when status is returned internally. Since the NFAD takes 1 clock for all operations and is the highest priority function unit, the clock after an NFAD compare operation reaches the UIR2 level of the pipeline, status is returned. The NPSW-internal R.FU_STATUS_EN is used to pop an entry from the FU queue, which is used to select which carry bit to write the FU_STATUS input to the NPSW into.

Hazards on carry bits are detected in a similar manner to the way they are detected on the register file. When a UIR1 level microinstruction attempts to access a carry bit, the UIR1 level controls are used to check the appropriate scoreboard bit to see if a hazard is pending. UIR1 controls are also compared to UIR2 controls to determine if a hazard is currently being set. This also includes the combinations of reservation and writing. For example, if a UIR1 microinstruction attempts to reserve SC while a UIR2 microinstruction simultaneously attempts to load SC with ALU status, a hazard occurs, stalling the UIR1 level for one clock until the write completes. This type of carry hazard is asserted at one of the WFC hazard outputs of the NPSW (carry represents the C in WFC). The WFC output partitioning was discussed in section 3.6.1.3, beginning on page 3-23. A carry hazard is also caused whenever the CU or FU queues becomes full. However, the queues were designed overly deep (8 entries) so they would never become full. Full detection is included as a stopgap measure, in case the assumption that 8 was large enough was incorrect.

A second type of hazard is detected on the PSW in the form of a wait hazard. When the microcode desires to read or write the entire PSW, it must also select PSW_HAZ as a wait condition to insure any pending hazards on the carry bits are considered. This is necessary since the PSW hazard logic only uses CRY_OP, CRY_SRC, and CRY_DST to represent an access of the carry bits. The wait hazard gives the microcode a second way to indicate an access of the carry bits. The NPSW determines the presence of a carry hazard in this instance if there are entries in the CU or FU status queues, implying pending return status.The wait hazard directly selects the PSW hazard into the UIR1 source hazard equation. Wait hazards were discussed in more detail in section 3.6.1.3, beginning on page 3-23.

The final way in which a pending carry bit hazard can cause a pipeline stall is through a test hazard. The NPSW receives the TSEL field of the microinstruction and decodes it to select the pending hazard condition for the carry bits. If there is a hazard pending on the carry bit selected by the UIR1 level staging of TSEL, one of the THAZ hazard outputs of the NPSW is asserted. These 4 outputs, which are discussed in more detail in section 3.4.3, beginning on page 3-8, combine in external NAS logic to form the TEST_HAZ input to the NRFAs. A test hazard stalls the pipeline at the UIR1 level.

3.8.5 Other PSW Hazards

The wait hazard has already been mentioned as a way the microcode can signify a PSW access to insure the validity of its contents before a read or write. The PSW wait hazard must include much more than just carry hazards, however. Incomplete NFU operations must also place hazards on the PSW. Consider the following instruction stream:

```
div.w    s0,s1
mov      psw,a3
```

If the divide instruction overflows and the PSW access doesn't wait for the divide to complete, then A3 would contain the wrong PSW value. Logic in the NPSW handles this by asserting a PSW hazard (which is selectable as a wait hazard) whenever any of the following occurs:

- a UIR2 level microinstruction is initiating a function unit request
- whenever there are entries in any of the function unit queues, indicating one of the function units is busy. The NPSW maintains copies of the read and write pointers for the NRFA's function unit queues (discussed in section 3.7.2, beginning on page 3-25) for this purpose
- whenever a function unit is writing the register file via the B bus. This would be the last possible clock an exception bit could be set. The NPSW determines B bus write enable in the same manner the NRFA does.

If the above instruction stream is changed slightly to use vector divides, we see the need for another type of hazard detection:

```
div.w    v0,v1,v2
mov      psw,a3
```

If the vector divide instruction overflows and the PSW access doesn't wait for the divide to complete, then A3 would contain the wrong PSW value. Recall from the interface chapter of this document (chapter 2) that the vector processor is dispatched under control of the NSP microcode. One parameter of this dispatch (SP_VP.DISP_PSW_HAZ) is whether the instruction should cause a PSW hazard. For example, vector divide causes a PSW hazard while vector load does not. The vector processor then asserts VP_SP.PSW_HAZ when it receives the dispatch. The NPSW receives the VP_PSW_HAZ field of the microinstruction and stages it to generate the SP_VP.DISP_PSW_HAZ signal for the NVP. The NPSW stages this signal to match all register levels of the dispatch interface, and combines it with the VP_SP.PSW_HAZ signal to determine if a vector processor PSW hazard is active. This VP PSW hazard then becomes part of the selectable PSW wait hazard.

3.9 CPU Control Register (CCR)

There are a number of infrequently performed functions in the NSP that aren't worth wasting microinstruction bits on. Examples of these are purging the instruction cache and changing operating modes on the data cache. To implement these sorts of functions, the NPSW array contains the 32-bit CPU Control Register (CCR). This register can be broadside loaded from the Y bus using an encoding of the YZ_WR_DST microinstruction field. The CCR is readable on the X data path via the pre-X mux described in section 3.13.1, beginning on page 3-36. Some of the bits within the CCR are driven off the NPSW as control outputs to the rest of the NSP. Since the NDC already receives the Y bus and the YZ_WR_DST field for other purposes, it maintains a local copy of the parts of the CCR which control the NDC. This conserves I/O pins on the NPSW and NDC. The format of the CCR is shown in Figure 3-12.

Figure 3-12: CCR Format

Bit(s)	Name	NPSW Output	Function
31	HALT	CCR_HALT	asserts board halt line
30	BKPT	none	used for simulation only
29:14	unused		
13	IDLE	IDLE_TRAP	tells NIP processor is idle
12	HARDERR	none	asserts NPSW hard error
11	unused		
10:9	VP_CNTX_CTL	SP_VP.CNTX_CTL	context mode controls for NVP
8	FAULT	none	used by microcode to detect recursive page faults
7	PURGE_ICACHE	CCR_PURGE_ICACHE	purge control for instruction cache
6	1PG_CACHE	CCR_1PG_CACHE	reduces data cache size to one page (from four)
5	VV	CCR_VV	architectural vector valid
4	unused		
3:2	FORCE_FAULT	none	forced fault mode controls: 00 - no forced fault 01 - fault on non-IP requests 10 - fault on IP requests 11 - fault on all requests
1	FORCE_DC_HIT	none	force data cache hit
0	CACHE_OFF	none	disable data cache

3.10 Microsecond Timer

The Convex architecture provides two execution timers - the thread timer register (TTR) and CPU Execution Timers (CTRs). Refer to the *Convex Architecture Reference* for details on the architecture of these timers. This section describes the NAS hardware used to implement them.

The TTR and CTRs count with microsecond granularity and are maintained by the microcode. The NAS has a 24-bit microsecond timer within the NPSW array that is used to accrue time on the processor between microcode updates of the TTR and/or CTRs. The NCU provides a 1-clock pulse (XCL_SP.USEC_EN) which is active once per microsecond, regardless of the system clock margin. This is registered on the NSP and input to the NPSW array as XCL_USEC_EN, which causes the timer to increment. In order to maintain the TTR and CTRs, microcode needs two operations from the timer: reading and clearing. The timer is readable through the X operand path via the NPSW's pre-X mux, which is described in detail in section 3.13.1, beginning on page 3-36. The timer may be cleared using an encoding of the YZ_WR_DST microinstruction field, staged to

the UIR2 level. Since a microsecond is very long compared to the system clock rate, it is important to insure no microsecond ticks are lost by clearing the timer on the same clock a microsecond pulse occurs. This is accomplished by a slight modification to the clearing function. Instead of setting the timer to zero, the XCL_USEC_EN input is loaded into the timer on a clear operation.

3.11 PC Staging

There are many times when the microcode must read the program counter. One example is in the call instruction, where the PC of the instruction following the call must be pushed in the return block. Another example is when a vector valid trap is detected - the PC of the current instruction must be pushed on the trap frame. A third example is when the NIP dispatches a trap. In this case, the PC of the instruction that would have been dispatched had the trap not occured must be pushed in the trap frame. To facilitate all three situations, the NPSW contains PC staging logic that maintains three PCs relative to the UIR1 level of the pipeline. The PC for the first example is called the NPC (next PC), which is the address of the next instruction. The PC for the second example is called the XPC (executing PC), which is the address of the instruction being executed by the NAS. The PC for the third example is called the CPC, which is the address of the executing instruction, or the next instruction in event of a trap dispatch, since the trap becomes the "executing instruction" and has no real address. Any of these three PCs can be read on the X operand path via the NPSW's pre-X mux, discussed in section 3.13.1, beginning on page 3-36. A block diagram of this logic is included in the upper center of the NSP Block Diagram.

The NPSW develops these three PCs by combining dispatch information from the NIP in three different ways. The NIP provides the following information:

- UPC_CPC - If the dispatched instruction is not the target of a branch instruction, this is the program counter for the dispatched instruction. The *target* of a branch is either the instruction branched to for a successful branch, or the instruction immediately following a failed branch. If the dispatched instruction is the target of a branch, UPC_CPC is the PC of the branch instruction. It is passed at the UPC level of the pipeline
- AS_DISP_BR_DISPL - If the dispatched instruction is the target of a successful branch, this is the displacement (distance in halfwords) from the branch to the dispatched instruction. If it's the instruction after a failed branch, the displacement is forced to 1 (the size of the branch instruction). If the dispatched instruction isn't a branch target, the displacement is 0. This parameter is passed at the input to UPC level.
- AS_DISP_SIZE - size in halfwords of the dispatched instruction, passed at the input to the UPC level.
- AS_DISP_XTEND - extended opcode flag for the current instruction, which for PC staging purposes just implies an extra halfword in size of the instruction

These dispatched values are staged and combined to form the CPC, XPC, and NPC in the following manner:

- CPC - staged directly from UPC_CPC
- XPC - branch displacement added to CPC
- NPC - branch displacement, size, and extend added to CPC

With a little study it should be clear that these 3 PCs provide the correct value required for the

above examples in all cases. In the event a microinterrupt occurs between the dispatch and the reading of the PC, the dispatch information must be held in the pipeline. Recall that a microinterrupt causes a source hazard for one clock. This guarantees that the PC at the UIR1 level will enter the UIR1_LAS level. The clock after the microinterrupt, the NUS asserts UINT_MODE which remains active throughtout the microinterrupt. The NPSW uses UINT_MODE as a control to the input to its PC staging, recirculating the UIR1_LAS level into the top of the pipe staging. This assures that the correct PC (NPC, XPC, and CPC) remains in the pipe and in fact appears at the UIR1 level of the pipe when the microinterrupted microinstruction returns to the UIR1 level.

3.12 Scratch RAM

The NAS includes a 2K by 36-bit (32 data and 4 parity) scratch RAM used for encaching SDRs and first level PTEs, holding constants, and acting as temporary storage. It is implemented with 4 2K by 9 self-timed RAMs (initially the National 100492). This implementation dictates the organization of 1 bit of odd parity for each data byte, placing data and parity in the same RAM part. Scratch RAM is initialized by the control store loader utility using the RAM part's built-in scan mode. The initial contents are defined using a simple microlanguage defined in chapter 7 of this document.

The data output of the RAMs can be sourced to the X operand path at the UIR1 level using the external X mux, described in detail in section 3.13.2, beginning on page 3-37. Scratch RAM (SR) may be written from the Y bus using a decode of the YZ_WR_DST field, from the UIR2 level. The different levels of access (UIR1 read, UIR2 write) for SR complicates the addressing of the RAM quite a bit. SR addressing is discussed in the next section.

3.12.1 Scratch RAM Addressing

The different functions of scratch RAM are implemented using a mapping scheme which divides the address space as shown in Figure 3-13.

Figure 3-13: Scratch RAM Address Mapping

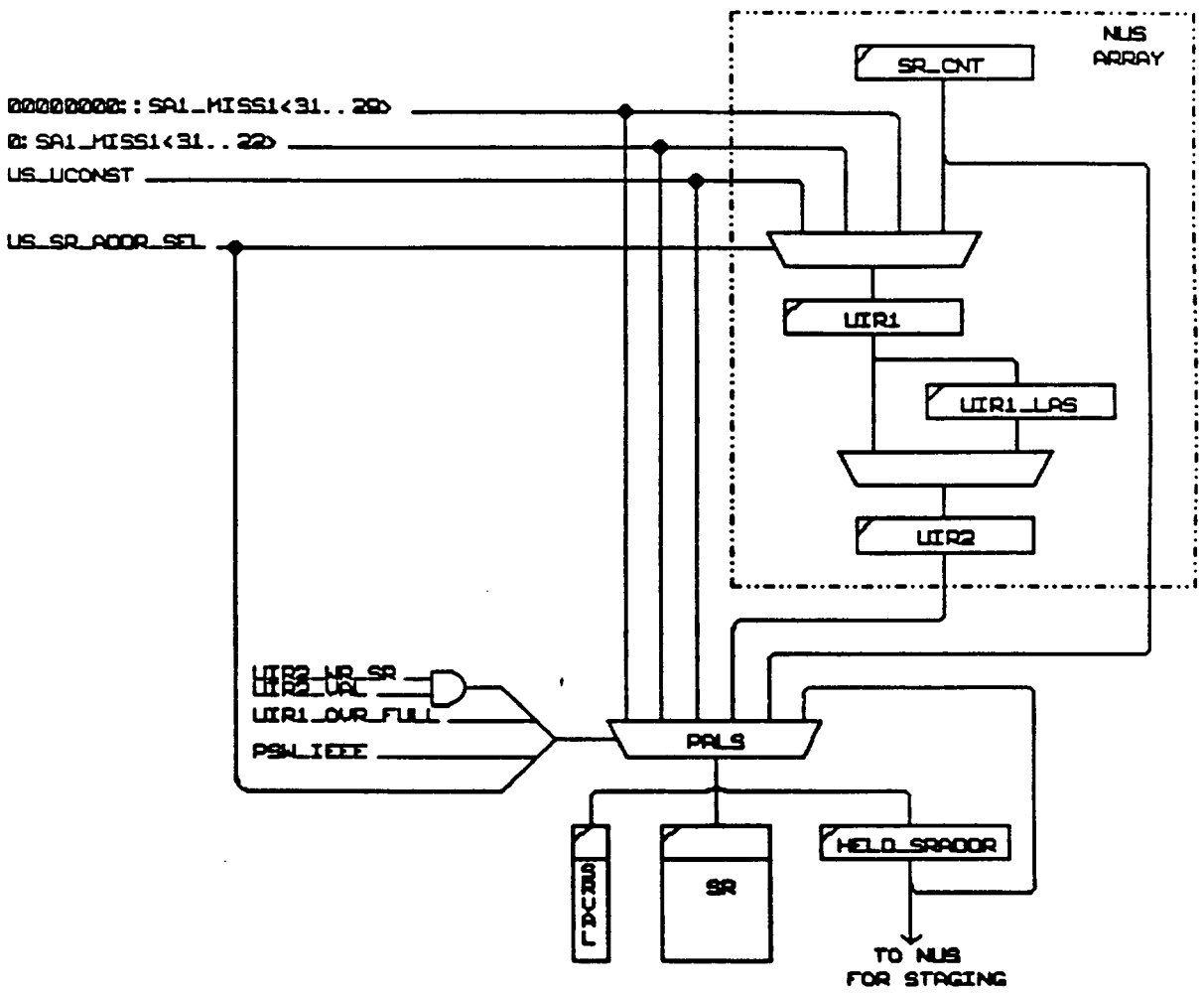
Start	End	Purpose
000	007	SDRs 0-7
008	1FF	floating point mode independent constants and temporary storage
200	2FF	Native mode floating point constants (PSW<IEEE> = 0)
300	3FF	IEEE mode floating point constants (PSW<IEEE> = 1)
400	7FF	PTE1 cache

Within this mapping are two major divisions - ranges addressed directly by the microcode (008_{16}-$3FF_{16}$) and ranges addressed based on logical memory addresses (000_{16}-007_{16} and 400_{16}-$7FF_{16}$). The first class of mapping is addressed using the UCONST field of the microinstruction (staged to the appropriate pipe level). The second class uses the SA1_MISS1 address supplied by the NDC during a PTE miss resolution. This is the logical memory address associated with the offending memory request. There is also a counter in the NUS array available to address SR.

SR is read at the UIR1 level of the pipe but written from the UIR2 level of the pipe. To make data

available for a UIR1 read, the read address is taken from the UPC level of the pipe. To write from a valid UIR2 microinstruction, the address is taken from the UIR2 level of the pipe. There may be only one access to the RAMs on each cycle, so it would be unacceptable for one microinstruction at the UPC level to be attempting a read while a second at the UIR2 level was attempting a write. This is avoided by microcode. Refer to chapter 7 for descriptions of the microprogramming rules that govern this. This also implies there is a muxing function in the address path for the RAMs. A block diagram of the SR addressing logic is shown in Figure 3-14.

Figure 3-14: SR Addressing Logic



The UIR2 level write address is developed in the NUS array and output as UIR2_SRADDR. The NUS also takes the WR_SR field of the microinstruction and stages it to UIR2, outputting UIR2_WR_SR which, when UIR2_VAL is asserted, causes the SR addressing PALs to select UIR2_SRADDR to the RAMs. Writes therefore have priority over reads (but there should be no collision due to microcode convention). If a write is not occurring, the SRADDR_SEL field of the microinstruction, directly from the control store RAM, is used to select the read address source. The pipeline overrun for the SR address is implemented with the external 100E241 HELD_SRADDR register. This register serves as a shadow copy of the STRAM's internal address register, which is not readable. HELD_SRADDR is fed back into the SR addressing PALs to be

re-selected as the address in the event UIR1_OVR_FULL is asserted, indicating the UIR1 level has stalled. HELD_SRADDR is also fed into the NUS array and staged to record the address of a read in the event of an SR parity error. It is staged through three registers which free-clock unless HALT is asserted, at which point data is held. These three registers mirror the latency from detection of an SR parity error until HALT is asserted. SR parity errors are detected in the NRFA through the XMUX_EXTDATA and XMUX_EXTPAR inputs. If the STRAM detects a parity error on a write operation, the latency is one less clock, so the RAM write address will be found one level earlier in the NUS staging. A write enable signal is also staged to handle multiple writes. Register name details for this hard error logic may be found in section 3.18, beginning on page 3-41.

The SR address PALs also apply the IEEE range modification function, for the range it is effective (008_{16} - $3FF_{16}$). If bits <10..9> of the selected SR address <10..0> are 01_2, the IEEE bit is inserted in place of bit 7.

3.12.2 Scratch RAM Validity

To assist in implementation of the PTE1 cache function of SR, there is a validity RAM associated with it, called SRVAL. This is implemented with a 2K by 2 bit purgeable self-timed RAM (currently the Vitesse VS12G478). This so-called "purge RAM" has the same interface as the data STRAMs with one exception. It also has a purge command available which clears all bits in the RAM immediately. In parallel with all SR writes the corresponding location in SRVAL is written with 11_2. Bit 0 of each RAM cell is used as the validity bit for the corresponding entry in SR. Bit 1 is used as *even* parity for the validity bit. A parity error is detected whenever the 2 bits read from the SRVAL RAM are different. Since the write data is a constant 11_2, write parity is not checked. Recall that the lower 1K of SR is used as the PTE1 cache. This implies that when a level 1 PTE is written to SR during PTE miss resolution, the corresponding validity bit in SRVAL is set. When SR is read, the SRVAL output is driven to the NUS array, which registers it for use as a test condition. PTE miss resolution microcode uses this test condition to determine if the PTE1 it read from SR was valid. The upper 1K of SRVAL is written but never used by the microcode.

3.13 X Mux Data Path

The X mux is the main data path used to bring information outside of the NRFA core of the NAS in to the main integer ALU and register file. This data path collects the pre-X mux output of the NPSW along with data external to gate arrays such as scratch RAM, combines it in the external X mux, and feeds it into the NRFAs' XMUX_EXTDATA inputs. It is shown in the upper center of the Neptune Scalar Processor Block Diagram. The two stages of the X data muxing are described in the upcoming sections.

3.13.1 NPSW Pre-X Mux

The first level of X data path muxing is the pre-X mux within the NPSW array. This mux selects from the following items:

- CPC - current program counter
- XPC - executing program counter
- NPC - next program counter
- CCR - CPU Control Register
- PSW - Processor Status Word

- TIMER - microsecond timer

The generation and maintenance of these items has been previously described. For the PCs, refer to section 3.11, beginning on page 3-33. The CCR was discussed in section 3.9, beginning on page 3-31. The PSW was discussed in section 3.8, beginning on page 3-28. The timer was discussed in section 3.10, beginning on page 3-32. The pre-X mux feeds through the PREX_DATA<31..0> outputs of the NPSW array. It accesses UIR1 level registers, so it is selected with the UIR1 level staging of the US_PREX_SEL control store field. The decodes of this field may be found in the microcode chapter (7) of this document.

The UIR1_LAS stage of the X mux path is maintained within the NRFA array. The PSW and CCR registers, which are read through the pre-X mux, do not move with the pipeline. To understand this, consider a UIR1 level read of the PSW that must wait for a function unit hazard on the PSW to clear. The resolution of this hazard will affect the contents of the PSW, so we need to insure we get the post-hazard value. While waiting for the hazard to clear, the pre-hazard value of the PSW would automatically load into a UIR1_LAS register and then hold there. We would therefore like the registered PSW value, which will be updated as the hazard clears, to be used in the read. To implement this, the NPSW decodes the PREX_SEL field and asserts USE_EXTDATA_LAS when the selected source for PREX_DATA is not the PSW or CCR. USE_EXTDATA_LAS is used by the NRFA to decide to use its internal XMUX_EXTDATA_LAS register as the X operand source.

There is no parity on the pre-X mux. This is due to pin limitations and the fact that there are no RAMs in the path and the signals do not cross connectors.

3.13.2 External X Mux

The second stage of X data path muxing is the external X (EXTX) mux. This stage of muxing is implemented using 16 100E163 discrete 2-bit 8:1 muxes. It selects between the following items:

- SA1_MISS1 - the logical address of the current request in the NDC, used to record the address of a page fault.
- UIR1_UCONST - the constant field of the microinstruction, staged to the UIR1 level within the NUS array.
- PREX_DATA - the output of the NPSW array's pre-X mux, described in the previous section
- SR_DATA - the data output of scratch RAM. Scratch RAM was discussed in more detail in section 3.12, beginning on page 3-34.
- TRAP_VECT - the 12-bit vector supplied with a trap by the NCU and registered within the NIP. This register is connected to 2 inputs on the mux, aligned in two ways. For most traps, right justification and zero extension provides the trap vector in the format desired by microcode. For the PATU (purge ATU entry) trap, the 12-bit vector is actually bits 21..12 of the logical address, so it is available on the X mux "shifted" left 12 bits for convenience in the microcode.
- NRC_CNTX - fault context from the NRC. These are the outputs of the context scan rings for the NRC arrays, concatenated to be read in parallel form
- CNTX1 - fault context set 1. These are the outputs of the context scan rings for the entire NSP excluding the NRCs.

The EXTX mux select comes from the EXTX_SEL field of the microinstruction, staged to the UIR1 and UIR1_LAS levels within the NUS array. UIR1_LAS_FULL from the NRFA selects between

these two using a discrete 100E155 2:1 mux. The decodes of the EXTX_SEL field may be found in the microcode chapter (7) of this document. The output of the EXTX mux, XMUX_EXTDATA, is fed into the NRFA arrays, along with XMUX_EXTPAR, which is just the parity output of scratch RAM. There is no parity on other XMUX_EXTDATA sources. As the NUS array stages the EXTX_SEL field, it is decoded to assert XMUX_EXTPAR_VAL whenever the selected data is from scratch RAM. This is used by the NRFAs to qualify their parity check on XMUX_EXTDATA.

3.14 NIP Control

Although the Instruction Processor (NIP) is a separate internally controlled entity from the NAS, there are times when the NAS must redirect the NIP. This section explains the logic in the NAS which controls the NIP

3.14.1 Jumps

The most obvious occasion for the NAS to redirect the NIP is when a jump instruction is executed. Similar but less obvious is when the microcode determines a program counter change is in order. Examples of this are fork acceptance, traps and the initial start-up of the machine. All of these are achieved with a *jump restart*. The JMP_RESTART field of the microinstruction, staged within the NUS array to the UIR2 level and qualified by the NIP with UIR2_VAL, does this. The decodes of the JMP_RESTART field may be found in the microcode chapter (7) of this document.

There are three types of jump restarts. The first type, called a SYSJMP (using the microcode mnemonic) is an absolute jump to a 31-bit address, specified on bits 31..0 of the Y bus. Since all instructions begin on halfword boundaries, the least significant bit of the Y bus is always ignored by the NIP. The SYSJMP may change the ring of execution (bits 31..29) of the program counter. This type of jump is used by the microcode for cross-ring jumps and traps.

The second type of jump, called a USRJMP, is similar to the SYSJMP except that the jump address constrained to remain within the current ring of execution. This means the NIP ignores bits 31..29 of the Y bus in addition to bit 0. USRJMPs are used in the jump instruction microcode.

The final type of jump is more complicated. When the NDC microinterrupts the NAS, the pipeline staging is disrupted for the microinterrupt handler. At the end of the microinterrupt handler, the pipeline staging of the PC within the NIP must be restored to maintain proper instruction flow. This is accomplished with the *microinterrupt restart* (or UINT_RESTART, using the microcode mnemonic). When a microinterrupt occurs, the NIP is informed by the assertion of UINT_MODE by the NAS. This assertion causes the NIP to internally hold the PCs it needs to begin after the microinterrupt is resolved. The NAS performs a UINT_RESTART at the completion of the microinterrupt handler, signaling the NIP to restart at the saved PC. This restart does *not* cause the NIP to restart its lookahead. A variant of the microinterrupt restart called a RTNC_RESTART is used at the completion of the *rtnc* instruction. The RTNC_RESTART, in addition to the microinterrupt restart of the PC, also causes the NIP to reset lookahead. The JMP_RESTART field of the microinstruction is 2 bits wide, while the UIR2_JMP_RESTART control output of the NAS to the NIP is 3 bits. The 2 least significant bits of the UIR2_JMP_RESTART control is formed by the UIR2 level staging of the JMP_RESTART field. The most significant bit is formed by the NUS from a decode of the USEQ_RAND field. This bit is asserted for a RTNC_RESTART and clear for a UINT_RESTART.

3.14.2 Lookahead

The NIP always attempts to keep instructions beyond the current PC prefetched so that it can continuously dispatch the NAS. There are times when the microcode knows the PC is about to change, so the extra memory bandwidth required by the NIP for its prefetch requests is being wasted. The NAS can remedy this situation using the IPINH field of the microinstruction. This field is staged within the NUS array to the UIR2 level and used to set and clear the IP_REQ_INH register. This signal is driven to the NDC. When it is asserted, the NDC ignores lookahead requests from the NIP in its arbitration. IP_REQ_INH is typically set at the beginning of a PC-changing microroutine, and cleared when the jump restart that changes the PC occurs.

3.14.3 Branch Restart

The final manner in which the NAS controls the NIP is the *branch restart*. The NIP employs a branch prediction scheme to improve performance. The implication to the NAS of this strategy is that every instruction dispatch may have followed a conditional branch that the NIP guessed whether or not to take. The NAS's role in this strategy is to inform the NIP if it guessed wrong. The NIP informs the NAS about its guess using the branch tagging method described in section 3.6.1.1, beginning on page 3-22. The essence of branch tagging is that each instruction specifies a condition (AC or SC) and polarity (0 or 1) it assumed was true when it was dispatched. Within the NRFA, the branch tag is checked for correctness at the UIR1 level of the pipeline. This check only occurs on the first microinstruction of a macroinstruction, denoted by assertion of the FIRSTU field of the microinstruction. If the check fails, the NRFA sets its BR_RESTART register, which is driven to the NIP to cause it to restart itself at the PC of the last branch instruction. The branch target is recalculated including the knowledge of the restart, and the correct instruction is dispatched.

3.15 Deadlock Detection

The Convex architecture provides for detection of software deadlocks in certain situations. For example, if all processors executing on behalf of a process are waiting on a semaphore that is locked, a deadlock condition is detected which causes the deadlocked processors to enter a software deadlock handler routine. This allows the operating system's scheduler to suspend the deadlocked process. A potential deadlock condition is detected on a processor if it takes a backward branch to a deadlockable instruction. Deadlockable instructions are listed in the *Convex Architecture Reference*. An example of a deadlockable sequence would be:

```
spin:  tas    effa
       bra.f  spin
```

If the tas fails, the branch back to try it again would be a potential deadlock. Each processor asserts SP_XCL_DEADLOCK deadlock if it is in a deadlocked state. The XCL passes these signals from all processors on to the NCU. The NCU checks all processors running in the same CIR (indicating they're executing on behalf of the same process), and if they're all in a deadlock state, a deadlock trap is issued to all processors.

The NAS plays a major role in the detection of deadlocks. The determination of whether an instruction is deadlockable is made by the microcode. Each microinstruction in the microcode routine for a deadlockable instruction must include the DL_EN construct, which encodes the USEQ_RAND field with the deadlock enable value. The NIP detects the backward branch part of the deadlock condition, and asserts UPC_DL with the instruction dispatch of the branch target. Note that this signal is passed at the UPC level of the pipeline.

The NUS array receives the USEQ_RAND field of the microinstruction as well as UPC_DL from the NIP. If both of these signals are active at the UIR2 level of the pipeline and UIR2_VAL is also asserted, the NUS sets an internal register, XCL_DEADLOCK. This is registered once more in a discrete part on the board and driven to the backplane as SP_XC.DEADLOCK. The NUS's XCL_DEADLOCK register is cleared the first time a microinstruction without the DL_EN construct reaches the UIR2 level. Assuming the process is deadlocked, this will be the first microinstruction of the deadlock handler microroutine.

3.16 Context Save and Restore

The NSP and NVP have a great deal of state that must be saved when a page fault occurs, and restored on the subsequent return from context block (rtnc). The NAS orchestrates context save and restore through microcode. The basic approach of context save is to scan context data out of all gate arrays in the NSP. Each gate array has a context ring, which is a subset of its entire scan ring. The outputs of these rings are chained together in pieces that are 80 bits in length. Some arrays contribute all 80 bits, while others are chained together to form an 80-bit string. These context subrings are bussed in parallel and made available as an input to the external X mux. This provides a path for the context data into the main ALU, so it can be written to registers and then stored in the context block in memory. To insure valid state with good parity is present in the context registers at the completion of the scan operation, the scanned out data is recirculated back to its original position. During context scan, the input to the context rings is supplied from the Y bus. The scanned out data is written in a register in the register file, which is sourced by the microcode on the Y bus on the next clock. In essence, the operand pipeline registers and the register file add a few extra bits to what becomes a circular scan ring. The NRC arrays are handled separately because they require special control to save return queue RAM state. The NRC RAMs are also directly resettable.

Microcode controls this scan operation through some bits in the MFP_OP microinstruction field. The CX_OP_REQ field is asserted to qualify the MFP_OP as a context type (as opposed to a function unit or memory operation). The NUS uses CX_OP_REQ and MFP_OP, staged to the UIR2 level and decoded, to produce separate context controls for the NRCs (NRC_CNTX_CTL) and the rest of the NSP (NSP_CNTX_CTL). The encodings of these controls are detailed in Appendix A. While the majority of the NSP is scanned, the NRC is placed in hold mode. Then the NSP is held while the NRC is scanned.

The NAS also provides context controls to the NVP. The NVP uses a similar context scan approach to produce fault state for the context block. When the fault is detected, the microcode fault handler encodes SP_VP.CNTX_CTL instructing the NVP to hold its context state. The NSP then saves its own context. When it's ready to save vector context, the NSP initiates a vector store operation through the normal microcode-driven NDC request interface. It then encodes SP_VP.CNTX_CTL instructing the NVP to save state. The vector context controls are contained within the CCR. The NPSW drives the 2 context control bits within the CCR off-chip to the NVP.

Context restore is similar to save, except in reverse order. Vector context is restored first. SP_VP.CNTX_CTL is encoded to instruct the NVP to restore state. The NSP then initiates a vector load using the normal microcode-driven NDC request interface. When vector context restoration is complete, the NVP is put in context hold mode while the NSP restores its own state. The NSP scanned context blocks are read from the fault block and stored temporarily in scratch RAM. From there, they are routed by microcode through the register file and sourced to the Y bus, while the context rings are scanned. The restore data essentially replaces the recirculation path described in the context save discussion.

This discussion obviously only scratched the surface of the very complex context save and restore operations. Much more insight can be gained by reading the fault and rtnc microcode routines. The format of the context frame, including the layout of the NSP and NRC scanned context blocks, is detailed in Appendix B.

3.17 Hung Hardware Detection

When the machine is running under the operating system and appears to have hung, it is often difficult without a logic analyzer to tell whether the hardware or the operating system is hung. A circuit is included in the NAS to aid in troubleshooting in these circumstances. The system can be considered *hardware hung* if the NAS has not accepted an instruction dispatch in a very long time. A 15-bit counter, HUNG_CNT, performs this function. This counter is incremented every clock, but loaded with zeroes each time the combined assertions of AS_DISP_REQ and AS_DISP_RDY show that a dispatch is accepted. If this counter carries out of the uppermost bit, HW_HUNG is asserted. This is registered on the NSP and sent to the backplane as SP_XC.HW_HUNG. The XCL registers this signal in the halt / halt mask register, which is readable by the SPU. In this manner

HUNG_CNT is implemented with 2 100E016 discrete 8-bit counters chained together. It is made scannable using 100E155 2:1 muxes at the input, switching between loading scan data or zeros.

3.18 NAS Hard Errors

This section lists the sources of hard errors detected by the NAS subsystem. These error descriptions are organized by the gate array or RAM that detects the hard error and asserts the error to the NSP board error detection logic. All hard errors except the NPSW array are caused by parity errors. The information listed for each source includes:

- parity error register
- list of registers or RAM and corresponding board level signals which parity is checked on
- list of registers which are held when the error occurs
- queue pointers for errors in queue management
- staged address and control for RAM-based errors

Parity is odd unless stated otherwise. Field widths are not supplied unless the part checks only a subset of the entire field. The names given signals are their scan ring definition names.

3.18.1 NUS Gate Array Parity Errors

The NUS array checks parity on the control store fields it receives, as well as the data read from the scratch RAM validity RAM (SRVAL).

Parity Error Signal:	NUS_PAR_ERR	
Enable:	NUS.PARITY_ENABLE	
Parity Error Register:	NUS.NUS_WCS_PAR_ERR	
Registers/Signals:	NUS.UIR1_NUS_PAR	US_NUS_PAR
	NUS.UIR1_USEQ_RAND	US_USEQ_RAND
	NUS.UIR1_CX_OP_REQ	US_CX_OP_REQ
	NUS.UIR1_LC_CTL	US_LC_CTL

	NUS.UIR1_SR_CNT_SEL	US_SR_CNT_SEL
	NUS.UIR1_WR_SR	US_WR_SR
	NUS.UIR1_IPINH	US_IPINH
	NUS.UIR1_SR_ADDR_SEL	US_SR_ADDR_SEL
	NUS.UIR1_MFP_OP<10..3>	US_MFP_OP<10..3>
	NUS.UIR1_EXTX_SEL	US_EXTX_SEL
	NUS.UIR1_BDSIZE	US_BDSIZE
	NUS.UIR1_JMP_RESTART	US_JMP_RESTART
	NUS.UIR1_TPOL	US_TPOL
	NUS.UIR1_TSEL	US_TSEL
	NUS.UIR1_BRTYPE	US_BRTYPE
	NUS.UIR1_UCONST	US_UCONST
	NUS.UIR1_BRADDR	US_BRADDR
Control Store Address:	NUS.UPC_LAS3	
Parity Error Register:	NUS.SRV_RD_PAR_ERR	
Registers/Signals:	NUS.SRVAL_DATA	SRVAL_DATA
	NUS.SRVAL_PAR	SRVAL_PAR
RAM Address:	NUS.SRADDR_LAS3	

3.18.2 NPSW Gate Array Hard Errors

The NPSW array detects one parity error and two hard errors. Like the NUS, it checks parity over all control store fields it receives. It asserts a hard error if a pop of an empty status queue is attempted. There are two such queues - one for NFAD compare status and one for NCU communication register lock operations. If the status queue error is asserted, it would imply that operations with the NFAD or NCU had somehow gotten confused. Examining the queue pointers would indicate which error occurred. An empty queue is indicated when the read and write pointers are equal. Unfortunately, the pop control signal (the FU_STATUS_EN register in the NPSW array or the board-level XCL_CU_STATUS_EN register) is not held on detection of the error. The second type of hard error detected by the NPSW is the microcode pulled error. When the microcode detects an error such as recursive page fault, it sets a bit in the CCR which is then registered in the NPSW as UCODE_PULLED_HARD_ERR. Isolation of this type of error requires examination of the microstack in the NUS array. A unique microaddress is associated with each kind of hard error. The read pointer is used to index in to the stack and read the microaddress.

Hard Error Signal:	NPSW_HARD_ERR	
Enable:	NPSW.PARITY_ENABLE (does NOT disable hard errors)	
Parity Error Register:	NPSW.NPSW_WCS_PAR_ERR	
Registers/Signals:	NPSW.UIR1_NPSW_PAR	US_NPSW_PAR
	NPSW.UIR1_VP_PSW_HAZ	US_VP_PSW_HAZ
	NPSW.UIR1_PREX_SEL	US_PREX_SEL
	NPSW.UIR1_FU_OP_REQ	US_FU_OP_REQ
	NPSW.UIR1_PSW_OP	US_PSW_OP
	NPSW.UIR1_CRY_OP	US_CRY_OP
	NPSW.UIR1_CRY_SRC	US_CRY_SRC
	NPSW.UIR1_CRY_DST	US_CRY_DST
	NPSW.UIR1_YZ_WR_DST	US_YZ_WR_DST
	NPSW.UIR1_FU_OP_7_6<7..6>	US_MFP_OP<10..9>
	NPSW.UIR1_FU_OP_4_3<4..3>	US_MFP_OP<7..6>

	NPSW.UIR1_FU_OP_0	US_MFP_OP<0>
	NPSW.UIR1_BDSIZE	US_BDSIZE
	NPSW.UIR1_ALUOP	US_ALUOP
	NPSW.UIR1_ALUSIZE	US_ALUSIZE
	NPSW.UIR1_TSEL	US_TSEL
Hard Error Register:	NPSW.STATQ_HARD_ERR	
Registers/Signals:	NPSW.CU_STAT_Q_0	
	NPSW.CU_STAT_Q_1	
	NPSW.CU_STAT_Q_2	
	NPSW.CU_STAT_Q_3	
	NPSW.CU_STAT_Q_4	
	NPSW.CU_STAT_Q_5	
	NPSW.CU_STAT_Q_6	
	NPSW.CU_STAT_Q_7	
	NPSW.FU_STAT_Q_0	
	NPSW.FU_STAT_Q_1	
	NPSW.FU_STAT_Q_2	
	NPSW.FU_STAT_Q_3	
	NPSW.FU_STAT_Q_4	
	NPSW.FU_STAT_Q_5	
	NPSW.FU_STAT_Q_6	
	NPSW.FU_STAT_Q_7	
Queue Pointers:	NPSW.CUSQ_WPTR	NPSW.CUSQ_RPTR
	NPSW.FUSQ_WPTR	NPSW.FUSQ_RPTR
Hard Error Register:	NPSW.UCODE_PULLED_HARD_ERR	
Microstack Pointer:	NUS.USTACK_RPTR	
Microstack Registers:	NUS.USTACK0	
	NUS.USTACK1	
	NUS.USTACK2	
	NUS.USTACK3	
Address/Error Type:	$BF4_{16}$ - Unimplemented trap entrypoint dispatched	
	$BF6_{16}$ - rtnc executed on non-context frame (FRL != 00)	
	$BF8_{16}$ - System Resource Structure underflow	
	BFA_{16} - diag instruction pull hard error subcode executed	
	BFC_{16} - invalid dispatch (escape opcode, branch, etc. was dispatched to NAS	
	BFF_{16} - recursive page fault; i.e. page fault on context block	

3.18.3 NRFA0 Gate Array Parity Errors

The NRFA array slice 0 checks parity on all control store fields it receives. In addition it checks parity on two bytes of the B and C busses, one byte of the instruction displacement from the NIP and one byte of scratch RAM. Scratch RAM parity is checked on the XMUX_EXTDATA input, whenever XMUX_EXTPAR_VAL is asserted. XMUX_EXTPAR_VAL is asserted whenever scratch RAM is selected for the X mux source by the microcode.

Parity Error Signal:	NRFA0_PAR_ERR
Enable:	NRFA0.PARITY_ENABLE
Parity Error Register:	NRFA0.NRFA_WCS_PAR_ERR

Registers/Signals:	NRFA0.UIR1_NRFA_PAR	US_NRFA_PAR
	NRFA0.UIR1_FU_OP_REQ	US_FU_OP_REQ
	NRFA0.UIR1_FIRSTU	US_FIRSTU
	NRFA0.UIR1_XREG_SEL	US_XREG_SEL
	NRFA0.UIR1_YREG_SEL	US_YREG_SEL
	NRFA0.UIR1_ZREG_SEL	US_ZREG_SEL
	NRFA0.UIR1_XREG_FMT	US_XREG_FMT
	NRFA0.UIR1_XREG_HAZ	US_XREG_HAZ
	NRFA0.UIR1_YREG_FMT	US_YREG_FMT
	NRFA0.UIR1_YREG_HAZ	US_YREG_HAZ
	NRFA0.UIR1_ZREG_HAZ	US_ZREG_HAZ
	NRFA0.UIR1_AGZ_HAZEN	US_AGZ_HAZEN
	NRFA0.UIR1_XMUX_SEL	US_XMUX_SEL
	NRFA0.UIR1_BDREG_FMT	US_BDREG_FMT
	NRFA0.UIR1_BDOP	US_BDOP
	NRFA0.UIR1_BDSIZE	US_BDSIZE
	NRFA0.UIR1_ABUS_FMT	US_ABUS_FMT
	NRFA0.UIR1_ABUS_WR_EN	US_ABUS_WR_EN
	NRFA0.UIR1_ABUS_SIZE	US_ABUS_SIZE
	NRFA0.UIR1_ALUOP	US_ALUOP
	NRFA0.UIR1_ALUFAST	US_ALUFAST
	NRFA0.UIR1_CSDISP	US_CSDISP
Parity Error Register:	NRFA0.BBUS_PAR_ERR	
Registers/Signals:	NRFA0.BBUS_UDATA	BBUS_DATA<39..32>
	NRFA0.BBUS_UPAR	BBUS_PAR <3>
	NRFA0.BBUS_LDATA	BBUS_DATA<7..0>
	NRFA0.BBUS_LPAR	BBUS_PAR<7>
Function Unit ID:	NRFA0.FU_RSLT_SEL2	
	0: NFAD 1: NMUL 2: NDIV 3: NMISC	
Parity Error Register:	NRFA0.CBUS_PAR_ERR	
Registers/Signals:	NRFA0.CBUS_UDATA	CBUS_DATA<39..32>
	NRFA0.CBUS_UPAR	CBUS_PAR <3>
	NRFA0.CBUS_LDATA	CBUS_DATA<7..0>
	NRFA0.CBUS_LPAR	CBUS_PAR<7>
Parity Error Register:	NRFA0.DISPL_PAR_ERR	
Registers/Signals:	NRFA0.UPC_DISPL	AS_DISP_DISPL<7..0>
	NRFA0.UPC_DISPL_PAR	AS_DISP_DISPL_PAR<3>
Parity Error Register:	NRFA0.EXTDATA_PAR_ERR	
Registers/Signals:	NRFA0.XMUX_EXTDATA_LAS	XMUX_EXTDATA<7..0>
	NRFA0.XMUX_EXTPAR_LAS	XMUX_EXTPAR<3>
Scratch RAM Address:	NUS.SRADDR_LAS3	
Note:	parity only checked when XMUX_EXTPAR_VAL asserted	

3.18.4 NRFA1 Gate Array Parity Errors

The NRFA array slice 1 checks parity on the same items as NRFA0, but on different bytes in the

buses.

Parity Error Signal:	NRFA1_PAR_ERR	
Enable:	NRFA1.PARITY_ENABLE	
Parity Error Register:	NRFA1.NRFA_WCS_PAR_ERR	
Registers/Signals:	NRFA1.UIR1_NRFA_PAR	US_NRFA_PAR
	NRFA1.UIR1_FU_OP_REQ	US_FU_OP_REQ
	NRFA1.UIR1_FIRSTU	US_FIRSTU
	NRFA1.UIR1_XREG_SEL	US_XREG_SEL
	NRFA1.UIR1_YREG_SEL	US_YREG_SEL
	NRFA1.UIR1_ZREG_SEL	US_ZREG_SEL
	NRFA1.UIR1_XREG_FMT	US_XREG_FMT
	NRFA1.UIR1_XREG_HAZ	US_XREG_HAZ
	NRFA1.UIR1_YREG_FMT	US_YREG_FMT
	NRFA1.UIR1_YREG_HAZ	US_YREG_HAZ
	NRFA1.UIR1_ZREG_HAZ	US_ZREG_HAZ
	NRFA1.UIR1_AGZ_HAZEN	US_AGZ_HAZEN
	NRFA1.UIR1_XMUX_SEL	US_XMUX_SEL
	NRFA1.UIR1_BDREG_FMT	US_BDREG_FMT
	NRFA1.UIR1_BDOP	US_BDOP
	NRFA1.UIR1_BDSIZE	US_BDSIZE
	NRFA1.UIR1_ABUS_FMT	US_ABUS_FMT
	NRFA1.UIR1_ABUS_WR_EN	US_ABUS_WR_EN
	NRFA1.UIR1_ABUS_SIZE	US_ABUS_SIZE
	NRFA1.UIR1_ALUOP	US_ALUOP
	NRFA1.UIR1_ALUFAST	US_ALUFAST
	NRFA1.UIR1_CSDISP	US_CSDISP
Parity Error Register:	NRFA1.BBUS_PAR_ERR	
Registers/Signals:	NRFA1.BBUS_UDATA	BBUS_DATA<47..40>
	NRFA1.BBUS_UPAR	BBUS_PAR <2>
	NRFA1.BBUS_LDATA	BBUS_DATA<15..8>
	NRFA1.BBUS_LPAR	BBUS_PAR<6>
Function Unit ID:	NRFA1.FU_RSLT_SEL2	
	0: NFAD 1: NMUL 2: NDIV 3: NMISC	
Parity Error Register:	NRFA1.CBUS_PAR_ERR	
Registers/Signals:	NRFA1.CBUS_UDATA	CBUS_DATA<47..40>
	NRFA1.CBUS_UPAR	CBUS_PAR <2>
	NRFA1.CBUS_LDATA	CBUS_DATA<15..8>
	NRFA1.CBUS_LPAR	CBUS_PAR<6>
Parity Error Register:	NRFA1.DISPL_PAR_ERR	
Registers/Signals:	NRFA1.UPC_DISPL	AS_DISP_DISPL<15..8>
	NRFA1.UPC_DISPL_PAR	AS_DISP_DISPL_PAR<2>
Parity Error Register:	NRFA1.EXTDATA_PAR_ERR	
Registers/Signals:	NRFA1.XMUX_EXTDATA_LAS	XMUX_EXTDATA<15..8>
	NRFA1.XMUX_EXTPAR_LAS	XMUX_EXTPAR<2>

Scratch RAM Address:	NUS.SRADDR_LAS3
Note:	parity only checked when XMUX_EXTPAR_VAL asserted

3.18.5 NRFA2 Gate Array Parity Errors

The NRFA array slice 2 checks parity on the same items as NRFA0, but on different bytes in the buses.

Parity Error Signal:	NRFA2_PAR_ERR	
Enable:	NRFA2.PARITY_ENABLE	
Parity Error Register:	NRFA2.NRFA_WCS_PAR_ERR	
Registers/Signals:	NRFA2.UIR1_NRFA_PAR	US_NRFA_PAR
	NRFA2.UIR1_FU_OP_REQ	US_FU_OP_REQ
	NRFA2.UIR1_FIRSTU	US_FIRSTU
	NRFA2.UIR1_XREG_SEL	US_XREG_SEL
	NRFA2.UIR1_YREG_SEL	US_YREG_SEL
	NRFA2.UIR1_ZREG_SEL	US_ZREG_SEL
	NRFA2.UIR1_XREG_FMT	US_XREG_FMT
	NRFA2.UIR1_XREG_HAZ	US_XREG_HAZ
	NRFA2.UIR1_YREG_FMT	US_YREG_FMT
	NRFA2.UIR1_YREG_HAZ	US_YREG_HAZ
	NRFA2.UIR1_ZREG_HAZ	US_ZREG_HAZ
	NRFA2.UIR1_AGZ_HAZEN	US_AGZ_HAZEN
	NRFA2.UIR1_XMUX_SEL	US_XMUX_SEL
	NRFA2.UIR1_BDREG_FMT	US_BDREG_FMT
	NRFA2.UIR1_BDOP	US_BDOP
	NRFA2.UIR1_BDSIZE	US_BDSIZE
	NRFA2.UIR1_ABUS_FMT	US_ABUS_FMT
	NRFA2.UIR1_ABUS_WR_EN	US_ABUS_WR_EN
	NRFA2.UIR1_ABUS_SIZE	US_ABUS_SIZE
	NRFA2.UIR1_ALUOP	US_ALUOP
	NRFA2.UIR1_ALUFAST	US_ALUFAST
	NRFA2.UIR1_CSDISP	US_CSDISP
Parity Error Register:	NRFA2.BBUS_PAR_ERR	
Registers/Signals:	NRFA2.BBUS_UDATA	BBUS_DATA<55..48>
	NRFA2.BBUS_UPAR	BBUS_PAR <1>
	NRFA2.BBUS_LDATA	BBUS_DATA<23..16>
	NRFA2.BBUS_LPAR	BBUS_PAR<5>
Function Unit ID:	NRFA2.FU_RSLT_SEL2	
	0: NFAD 1: NMUL 2: NDIV 3: NMISC	
Parity Error Register:	NRFA2.CBUS_PAR_ERR	
Registers/Signals:	NRFA2.CBUS_UDATA	CBUS_DATA<55..48>
	NRFA2.CBUS_UPAR	CBUS_PAR <1>
	NRFA2.CBUS_LDATA	CBUS_DATA<23..16>
	NRFA2.CBUS_LPAR	CBUS_PAR<5>

Parity Error Register:	NRFA2.DISPL_PAR_ERR	
Registers/Signals:	NRFA2.UPC_DISPL	AS_DISP_DISPL<23..16>
	NRFA2.UPC_DISPL_PAR	AS_DISP_DISPL_PAR<1>
Parity Error Register:	NRFA2.EXTDATA_PAR_ERR	
Registers/Signals:	NRFA2.XMUX_EXTDATA_LAS	XMUX_EXTDATA<23..16>
	NRFA2.XMUX_EXTPAR_LAS	XMUX_EXTPAR<1>
Scratch RAM Address:	NUS.SRADDR_LAS3	
Note:	parity only checked when XMUX_EXTPAR_VAL asserted	

3.18.6 NRFA3 Gate Array Parity Errors

The NRFA array slice 3 checks parity on the same items as NRFA0, but on different bytes in the buses.

Parity Error Signal:	NRFA3_PAR_ERR	
Enable:	NRFA3.PARITY_ENABLE	
Parity Error Register:	NRFA3.NRFA_WCS_PAR_ERR	
Registers/Signals:	NRFA3.UIR1_NRFA_PAR	US_NRFA_PAR
	NRFA3.UIR1_FU_OP_REQ	US_FU_OP_REQ
	NRFA3.UIR1_FIRSTU	US_FIRSTU
	NRFA3.UIR1_XREG_SEL	US_XREG_SEL
	NRFA3.UIR1_YREG_SEL	US_YREG_SEL
	NRFA3.UIR1_ZREG_SEL	US_ZREG_SEL
	NRFA3.UIR1_XREG_FMT	US_XREG_FMT
	NRFA3.UIR1_XREG_HAZ	US_XREG_HAZ
	NRFA3.UIR1_YREG_FMT	US_YREG_FMT
	NRFA3.UIR1_YREG_HAZ	US_YREG_HAZ
	NRFA3.UIR1_ZREG_HAZ	US_ZREG_HAZ
	NRFA3.UIR1_AGZ_HAZEN	US_AGZ_HAZEN
	NRFA3.UIR1_XMUX_SEL	US_XMUX_SEL
	NRFA3.UIR1_BDREG_FMT	US_BDREG_FMT
	NRFA3.UIR1_BDOP	US_BDOP
	NRFA3.UIR1_BDSIZE	US_BDSIZE
	NRFA3.UIR1_ABUS_FMT	US_ABUS_FMT
	NRFA3.UIR1_ABUS_WR_EN	US_ABUS_WR_EN
	NRFA3.UIR1_ABUS_SIZE	US_ABUS_SIZE
	NRFA3.UIR1_ALUOP	US_ALUOP
	NRFA3.UIR1_ALUFAST	US_ALUFAST
	NRFA3.UIR1_CSDISP	US_CSDISP
Parity Error Register:	NRFA3.BBUS_PAR_ERR	
Registers/Signals:	NRFA3.BBUS_UDATA	BBUS_DATA<63..56>
	NRFA3.BBUS_UPAR	BBUS_PAR <0>
	NRFA3.BBUS_LDATA	BBUS_DATA<31..24>
	NRFA3.BBUS_LPAR	BBUS_PAR<4>
Function Unit ID:	NRFA3.FU_RSLT_SEL2	
	0: NFAD 1: NMUL 2: NDIV 3: NMISC	
Parity Error Register:	NRFA3.CBUS_PAR_ERR	

Registers/Signals:	NRFA3.CBUS_UDATA	CBUS_DATA<63..56>
	NRFA3.CBUS_UPAR	CBUS_PAR <0>
	NRFA3.CBUS_LDATA	CBUS_DATA<31..24>
	NRFA3.CBUS_LPAR	CBUS_PAR<4>
Parity Error Register:	NRFA3.DISPL_PAR_ERR	
Registers/Signals:	NRFA3.UPC_DISPL	AS_DISP_DISPL<31..24>
	NRFA3.UPC_DISPL_PAR	AS_DISP_DISPL_PAR<0>
Parity Error Register:	NRFA3.EXTDATA_PAR_ERR	
Registers/Signals:	NRFA3.XMUX_EXTDATA_LAS	XMUX_EXTDATA<31..24>
	NRFA3.XMUX_EXTPAR_LAS	XMUX_EXTPAR<0>
Scratch RAM Address:	NUS.SRADDR_LAS3	
Note:	parity only checked when XMUX_EXTPAR_VAL asserted	

3.18.7 NFU Gate Array Parity Errors

The Neptune Function Unit gate arrays (NFAD, NMUL, NDIV, NMISC) each check parity on their X and Y input operands. Recall that these operands are bussed together. Therefore, if the NRFA drives bad parity, an error should be detected in all four function units. If less than four function units detect a parity error, a terminator, board net, or bad function unit would most likely be indicated.

Parity Error Signal:	NFAD_PAR_ERR	
Enable:	NFAD.PARITY_ENABLE	
Parity Error Register:	NFAD.XBUS_PAR_ERR	
Registers/Signals:	NFAD.XBUS_DATA	XMUX_DATA
	NFAD.XBUS_PAR	XMUX_PAR
Parity Error Register:	NFAD.YBUS_PAR_ERR	
Registers/Signals:	NFAD.YBUS_DATA	YMUX_DATA
	NFAD.YBUS_PAR	YMUX_PAR
Parity Error Signal:	NMUL_PAR_ERR	
Enable:	NMUL.PARITY_ENABLE	
Parity Error Register:	NMUL.XBUS_PAR_ERR	
Registers/Signals:	NMUL.XBUS_DATA	XMUX_DATA
	NMUL.XBUS_PAR	XMUX_PAR
Parity Error Register:	NMUL.YBUS_PAR_ERR	
Registers/Signals:	NMUL.YBUS_DATA	YMUX_DATA
	NMUL.YBUS_PAR	YMUX_PAR
Parity Error Signal:	NDIV_PAR_ERR	
Enable:	NDIV.PARITY_ENABLE	
Parity Error Register:	NDIV.XBUS_PAR_ERR	
Registers/Signals:	NDIV.XBUS_DATA	XMUX_DATA
	NDIV.XBUS_PAR	XMUX_PAR

Parity Error Register:	NDIV.YBUS_PAR_ERR	
Registers/Signals:	NDIV.YBUS_DATA	YMUX_DATA
	NDIV.YBUS_PAR	YMUX_PAR
Parity Error Signal:	NMISC_PAR_ERR	
Enable:	NMISC.PARITY_ENABLE	
Parity Error Register:	NMISC.XBUS_PAR_ERR	
Registers/Signals:	NMISC.XBUS_DATA	XMUX_DATA
	NMISC.XBUS_PAR	XMUX_PAR
Parity Error Register:	NMISC.YBUS_PAR_ERR	
Registers/Signals:	NMISC.YBUS_DATA	YMUX_DATA
	NMISC.YBUS_PAR	YMUX_PAR

3.18.8 Scratch RAM Write Parity Errors

The STRAM used for scratch RAM includes the ability to detect parity errors on write operations. This function is used on the NAS to check Y bus parity when SR is written. The address and write enables are staged in the NUS array to aid in interpreting which operation caused the error. In the event that the NRFA drives bad parity on the Y bus, it should be detected many other places in addition to SR. SR detects this error a clock later than other Y bus sources. In this instance the staged address and control would be incorrect. Therefore, when a Y bus parity error is detected, SR write should be interrogated last by the hard logger software.

Parity Error Signal:	SRD_WR_PAR_ERR	
Enable:	_STRAM_ENABLE	
Parity Error Register:	SRD_WR_PAR_ERR<3..0>	
RAMS/Signals:	SR_[0,1,2,3]	YBUS_DATA<31..0>
		YBUS_PAR<3..0>
RAM Address:	NUS.SRADDR_LAS2	
	NUS.SRADDR_LAS1 (most recent previous write for multiple writes)	
	HELD_SRADDR (least recent previous write for multiple writes)	
RAM Write Enable:	NUS._SR_WE_LAS2	
	NUS._SR_WE_LAS1 (associated with NUS.SRADDR_LAS1)	
	NUS._SR_WE_LAS0 (associated with HELD_SRADDR)	

4 NIP Subsystem

4.1 Overview

The NIP subsystem is a cache based instruction processor. Data from memory is written unmodified into the Icache, a 2kx72 cache. The data is read from the Icache and parsed into individual instructions which are later transferred to the NAS subsystem for execution. The NIP subsystem has a seven deep queue for parsed instructions so the NIP subsystem can continue parsing instructions even when the NAS subsystem is unable to accept them. The NIP subsystem is implemented with two types of gate arrays: Neptune Instruction Address (NIAD) which does all address manipulation required by the NIP, and Neptune Parser (NPAR) which parses instructions and cracks entrypoints, register fields, and other information from the raw memory data stored in the Icache. Twenty rams are used in the implementation of the NIP: eight combine to implement the 2kx72 Icache which contains the raw data from memory from which instructions will be parsed. Three rams are used in a 2kx27 configuration for the Icache address tag and three similarly for the look ahead address tag. The Icache and address tag rams are all 2kx9 rams. An additional ram is used to contain the Icache valid and another for the Look Ahead valid bit, which indicate simply that the data at the corresponding address is valid. These two and the four rams used to implement branch history are 2kx2 only, using one of the two bits as data and the other as parity. The four branch history rams contain a branch history bit and a correct/incorrect bit each corresponding to the upper and lower words of the Icache. In addition to these rams, the NIP contains a few discrete ECLiPS parts which implement functionality that did not fit in or was shared by the gate arrays. In general, address caculations used by the instruction processor are done in the NIAD gate arrays, and parsing and cracking of instructions is done in the NPAR array.

4.2 Starting the Instruction Processor

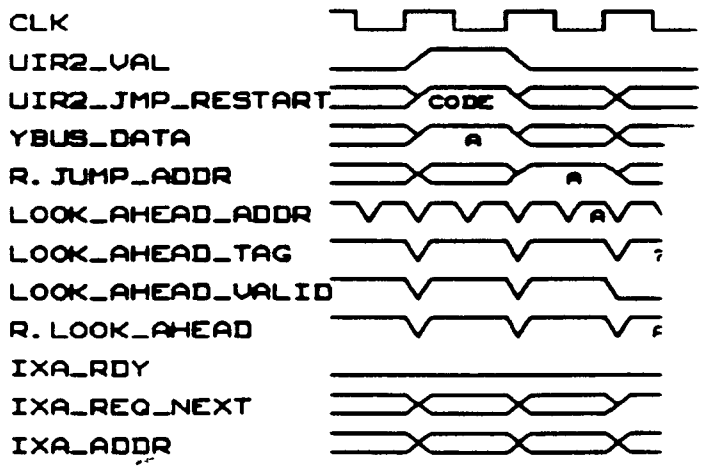
The Neptune instruction processor is designed to continuously fetch data from the Icache and parse instructions from that data. The instruction processor is started at a known address by the NAS with the assertion of UIR2_VAL, UIR2_JMP_RESTART<2..0>, and the jump address asserted on YBUS_DATA<31..0> with corresponding parity on YBUS_PAR<7..4>. UIR2_JMP_RESTART is a restart code, interpreted as shown in Table 4-1. Immediately after power up or a new process startup, the restart code should be a 2 which specifies a ring in which to begin parsing instructions as well as the address at which to start. The restart code from UIR2_JMP_RESTART is qualified with the assertion of UIR2_VAL then registered in the NIADs as R.UIR2_VAL. At the same time, YBUS_DATA and parity will also be registered as R.JUMP_ADDR and R.YBUS_PAR. Parity on the YBUS_DATA will be checked after it is registered.

Table 4-1 UIR2_JMP_RESTART

Code	Operation
0	No restart
1	Microinterrupt restart
2	Cross ring jump
3	Same ring jump
4	Microinterrupt restart
5	Not used, action und
6	Not used, action und
7	Not used, action und

The address now in R.JUMP_ADDR, shown as data A in Fi program counter, R.BPC, on the next clock cycle. The address the read address on LOOK_AHEAD_ADDR which addresse ahead address tag rams, INST_CACHE_ADDR which addresse instruction address tag rams, and BRANCH_HIST_ADDR whi rams. The rams are all double cycled to accept write addresses scalar processor clock and read addresses on the second h represented by A will be asserted on the second half cycle to LOOK_AHEAD_ADDR will select the address contained in F ahead tag and validity rams to determine if the data at this addre valid. The look ahead tag rams will contain the upper 18 bits of by address bits 13 through 3, so that the address compared is th the next clock edge, the address on LOOK_AHEAD_ADDR is During the cycle following that clock edge, the output of the look to the upper 18 bits of R.LOOK_AHEAD which will be combi determine if the data in the Icache rams at this address is the co data needed to begin parsing instructions at the restart address request must be made for the data at the address now in R.LO

Figure 4-1 Timing of a jump res



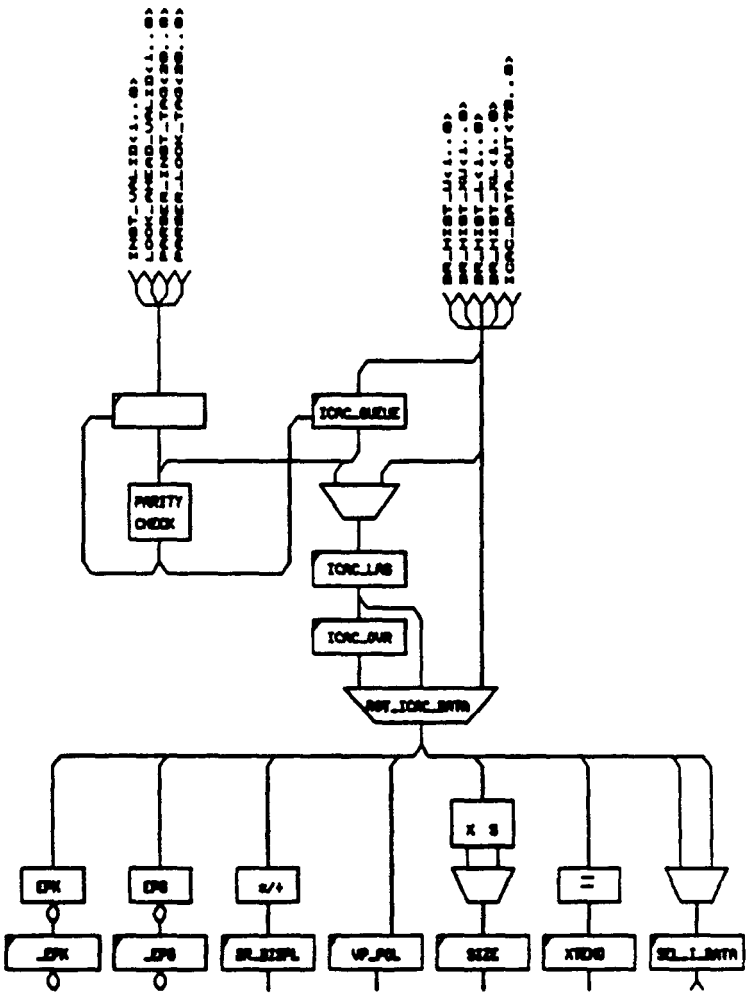
cycle, this address becomes IXA_ADDR, which is handed off using the Neptune standard RDY/REQ_NEXT protocol with described in section 2.2.8.

The data requested will eventually come back from memory and will be passed to the NIP from the NRC via the MXI bus, as described in section 2.2.9. The NIP will have continued accessing this icache location and the next consecutive location until this data is returned by the NRC and is written into, then read from the Icache, instruction address tag rams and instruction validity ram. After writing and then reading these rams, the PARSER_INST_TAG shows that the requested data is now present and the INST_VALID bit as accessed by the INST_CACHE_ADDR indicates to the instruction parsing logic that valid data was read from the Icache in the latest access. At this time, instructions will begin to be parsed and NIP will begin normal operation.

4.3 Instruction Parsing

Once valid data has been read from the Icache, that data enters the parsing logic shown in Figure

Figure 4-2 Parser input staging



4-2 as ICAC_DATA_OUT, which includes 64 bits of data plus eight bits of parity. When the processor has just been restarted, the data from the Icache will be selected and rotated immediately upon entering the parsing logic. The rotation selects which halfword of data will be first parsed as an instruction, and is initially determined simply by bits <2..1> from the address presented on YBUS_DATA at restart. An instruction may be from one to four halfwords, and may cross longword boundaries so that a read may contain incomplete instructions which will be

discussed later. For this discussion, assume that the first instruction parsed is complete in the first read of data from the Icache. The starting halfword will be cracked into a microcode entrypoint for both a standard and an extended instruction. For a complete list of op codes and entrypoints, see the chapter on microcode. The entrypoint for a standard instruction will be cracked from the first halfword of the instruction shown above as EPS and the entrypoint for an extended instruction will be cracked from the second halfword in the instruction and is shown as EPX. The entrypoints are cracked in parallel due to timing constraints and the proper entrypoint will be selected after both are registered. The rotated data is also used to generate R.XTEND, which indicates that the instruction was extended as well as to calculate R.SIZE and R.BR_DISPL. These become board signals PARSE_XTEND, PARSE_SIZE<1..0> and PARSE_BR_DISP<7..0>. The rotated data also is used to generate R.SEL_ICAC_DATA which contains much of the same data from the long word which was fetched from the Icache. The use of these signals will be discussed later.

Note that the preceding discussion applies only if the instruction was complete in the first valid long word of data read from the Icache. If that instruction was incomplete, for example the jump start address indicated that the NIP should begin parsing at halfword 3 but the size indicates that the instruction is 2 halfwords long, then the next long word must be fetched from the Icache to complete parsing the instruction. The current longword of data will be registered in R.ICAC_LAS to save the data until it can be parsed. The same longword will be saved in R.ICAC_QUEUE, simply for parity checking. When the next consecutive longword is read from the Icache and is valid, the complete instruction will be parsed from the least significant halfword of the data now in R.ICAC_LAS and the most significant halfword of the data at ICAC_DATA_OUT. This complete parsed instruction will then be registered and applied to the various fields as described above.

The information concerning where to parse the next instruction, i.e. from ICAC_DATA_OUT, R.ICAC_LAS or R.ICAC_OVR is maintained in C.INST_ROT. C.INST_ROT is a four bit field: the most significant bit indicates if instruction parsing should begin in R.ICAC_OVR, the second most significant bit indicates that parsing should begin in R.ICAC_LAS, and the least significant two bits indicate at which halfword in a given longword to begin parsing instructions. C.INST_ROT and its determination of where to begin parsing instructions is illustrated in Table 4-2. If the most significant two bits of C.INST_ROT are not set, then parsing begins with ICAC_DATA_OUT. At no

Table 4-2 Rotation and alignment of data

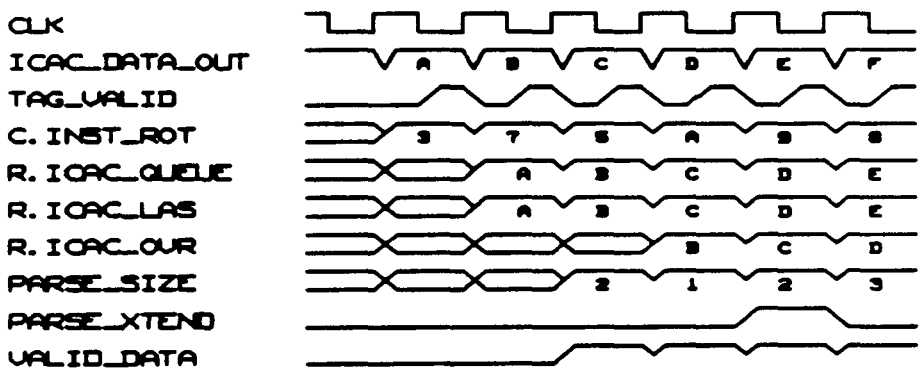
C.INST_ROT	Rotation of incoming data and registers
0	ICAC_DATA_OUT<63..0>
1	ICAC_DATA_OUT<47..0>, R.ICAC_OVR <63..48>
2	ICAC_DATA_OUT<31..0>, R.ICAC_OVR<63..32>
3	ICAC_DATA_OUT<15..0>, R.ICAC_OVR<63..16>
4	R.ICAC_LAS<63..0>
5	R.ICAC_LAS<47..0>, ICAC_DATA_OUT <63..48>
6	R.ICAC_LAS<31..0>, ICAC_DATA_OUT<63..32>
7	R.ICAC_LAS<15..0>, ICAC_DATA_OUT<63..16>
8	R.ICAC_OVR<63..0>
9	R.ICAC_OVR<47..0>, R.ICAC_LAS <63..48>
A	R.ICAC_OVR<31..0>, R.ICAC_LAS<63..32>
B	R.ICAC_OVR<15..0>, R.ICAC_LAS<63..16>

time will both of the most significant bits be set. As valid instructions are parsed and registered,

the least significant bits of C.INST_ROT will be properly incremented by the number of halfwords in the last instruction to insure that the bits point to the location of the next instruction to be parsed, but there is no carry into the upper two bits of that field. The upper two bits reflect only the location of the data to be parsed.

Figure 4-3 shows a possible scenario with an instruction which is not complete in the first longword of data read from the Icache after a restart. This example assumes a restart address with bits <2..1> equal to 3, perhaps from an address such as 0X60008006. After a restart the data in any of the input staging registers for the parsing logic, i.e. R.ICAC_QUEUE, R.ICAC_LAS, or R.ICAC_OVR, is invalid. The first valid data will enter the parsing logic on ICAC_DATA_OUT, labeled "A" in Figure 4-3. The restart address becomes C.INST_ROT which, according to Table 4-2 indicates that the least significant halfword of the ICAC_DATA_OUT is part of the new instruction, but the remainder is not available until some future read of the Icache; here it is shown as the next read. Since no complete instruction may be parsed, the ICAC_DATA_OUT or A is registered in R.ICAC_LAS and R.ICAC_QUEUE, the latter only for parity checking. There is now valid data in R.ICAC_LAS, so C.INST_ROT becomes a product of the starting address and the validity of R.ICAC_LAS and the value is now 7 (from C.INST_ROT<3..2> equal one which means use R.ICAC_LAS and C.INST_ROT<1..0> equal three means start with the least significant halfword.) Now parsing is to begin with the least significant halfword of R.ICAC_LAS and the data needed to complete the instruction will be available on ICAC_DATA_OUT. When the new data from the Icache is available on ICAC_DATA_OUT, the first instruction may be completely parsed using the least significant halfword from R.ICAC_LAS and the most significant halfword from ICAC_DATA_OUT and its entrypoint will be registered as described above. VALID_DATA will be asserted on the next cycle, indicating that a complete instruction was parsed and PARSE_SIZE and PARSE_EXTEND will have registered the size and extend information about the instruction. Also note that the data that was presented on ICAC_DATA_OUT has moved into R.ICAC_LAS, but that which was in R.ICAC_LAS has not moved to R.ICAC_OVR since it is no longer valid. R.ICAC_OVR should load data only when R.ICAC_LAS and ICAC_DATA_OUT are valid and the R.ICAC_LAS will contain valid data after parsing the current instruction. This is discussed in depth in the next section

Figure 4-3 Parse Timing



The cycle after successfully parsing the first instruction, C.INST_ROT becomes a 5 which indicates that the data in R.ICAC_LAS is valid and parsing should begin on the next most significant halfword. The example shows the next instruction parsed to be only one halfword long

and is completely contained in R.ICAC_LAS. As the second instruction is parsed, another valid longword of data is presented from the Icache. This new data is again stored in R.ICAC_QUEUE and R.ICAC_LAS, while the data which was in R.ICAC_LAS and still contains instructions to be parsed now moves to R.ICAC_OVR, since that register does not contain any valid data at this time. Since the previous C.INST_ROT was a 5 and the size of that instruction was 1, C.INST_ROT now becomes A where the most significant 2 bits indicate parsing should be from R.ICAC_OVR and the least significant bits represent the addition of the least significant two bits of 5 (0b01) and the length of the previous instruction (0b01), or 0b10. Note that this instruction is an extended instruction so that the actual length of the instruction is PARSE_SIZE plus PARSE_XTEND, or three halfwords and C.INST_ROT must change by the total instruction length to parse the next instruction. Similarly, movement of valid data through R.ICAC_LAS and R.ICAC_OVR along with the size of the instruction being parsed can be used to determine the subsequent C.INST_ROT values shown in the example.

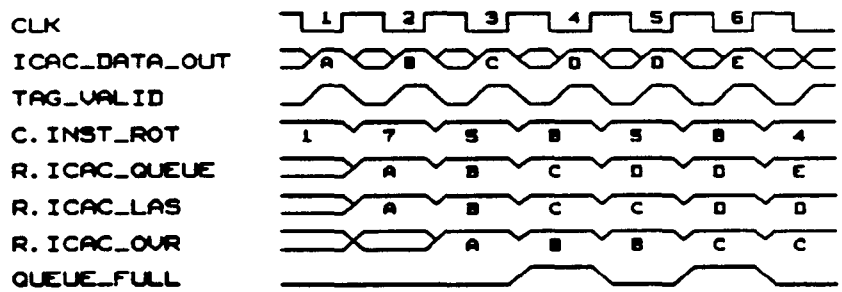
4.4 Queues and Queue Manipulation

There are two sections of logic in the NIP regularly referred to as queues: the look aside registers which hold data that has been read from the Icache until it can be parsed, and the queues which contain parsed and cracked instructions and their corresponding address and branch information. This section will discuss each of these queues, what they contain and how they are controlled as well as why they are necessary.

4.4.1 The Look Aside Queue

The registers which make up the look aside queue are R.ICAC_LAS, R.ICAC_OVR and R.ICAC_QUEUE and their flow is shown in the upper right side of Figure 4-2. These registers are the input staging registers for the NIP parsing logic and their use is illustrated in Figure 4-4. ICAC_DATA_OUT with a value of A from the Icache enters the parsing logic in cycle 1. If the longword is accompanied by TAG_VALID and it cannot be completely parsed then it is stored in R.ICAC_LAS. Unless R.ICAC_QUEUE is holding valid data, as will be discussed later, a copy of the new data from ICAC_DATA_OUT is always registered in R.ICAC_QUEUE since this is the only place that the parity is checked on data from the Icache. Also in cycle 1, C.INST_ROT is set

Figure 4-4 Typical input queue activity



as it would be for a restart address ending in 0b010 or 0b011, since the least significant bit is always ignored due to a minimum of halfword accesses. With C.INST_ROT equal to 1 and TAG_VALID set, a complete instruction would be parsed if the instruction size were 3 halfwords or less. The example assumes a valid instruction of two halfwords, which adds two to the least significant two bits of C.INST_ROT. Those bits go from 0b01 to 0b11, as shown in the change from 1 to 7 in cycle 2. As explained in 4.3, bits 2 and 3 of C.INST_ROT refer to which register the data is actually being parsed from, so a value of 7 indicates that parsing will start with halfword 3

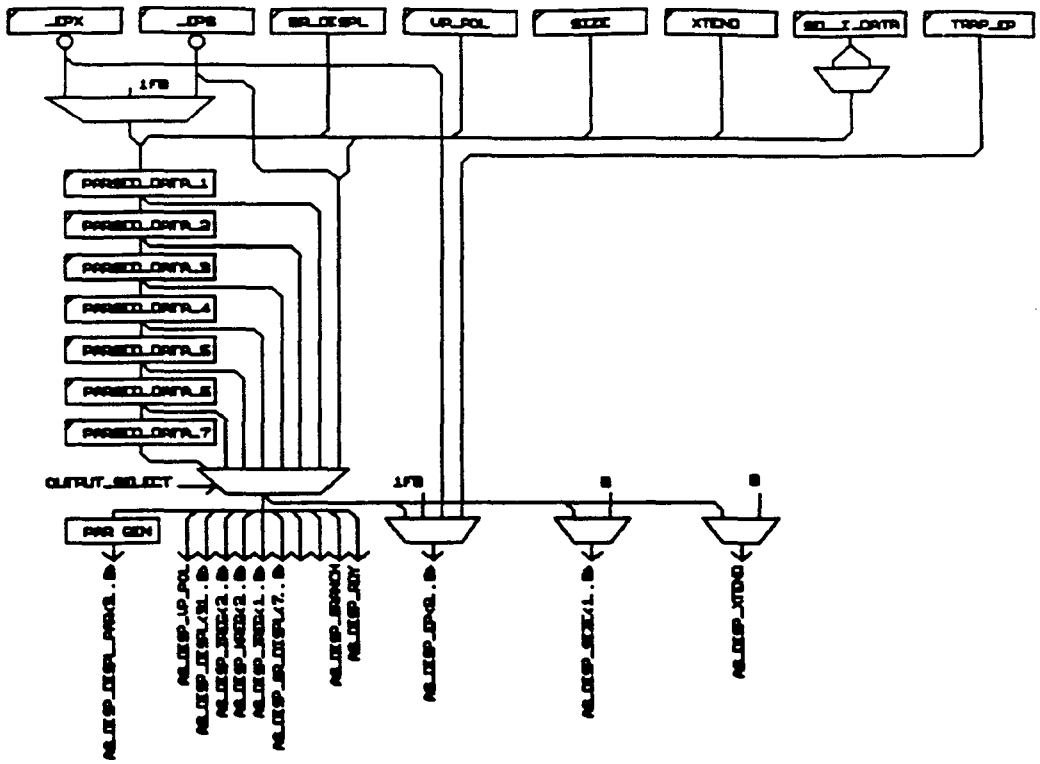
of the data contained in R.ICAC_LAS. The data from which the first instruction was parsed automatically moved into R.ICAC_LAS since all valid data in that long word had not been parsed, indicated by the resulting C.INST_ROT of 7. Note that data A has moved to R.ICAC_LAS for cycle 2 and that new valid data, B, is presented on ICAC_DATA_OUT.

On cycle 3, the original data A has moved into R.ICAC_OVR, which will happen any time R.ICAC_LAS contains valid data and R.ICAC_OVR does not. C.INST_ROT of 5 indicates that the next instruction is to be parsed from data B which is now contained in R.ICAC_LAS. From cycle 3 to cycle 4, C.INST_ROT goes from 5 to B, so it is pointing to the next instruction as data moves from R.ICAC_LAS to R.ICAC_OVR. On cycle 4, QUEUE_FULL is asserted to show the Icache addressing mechanism that no more data can be accepted from the Icache at this time, which results in the same data that was read from the Icache this cycle being presented for parsing on the next cycle as well. Also on cycle 4, new data presented on ICAC_DATA_OUT will be registered in R.ICAC_QUEUE only, for parity checking but also to retain that data for parsing as soon as the data in R.ICAC_OVR is completely parsed. This happens on cycle 5, when data is now to be parsed from R.ICAC_LAS which indicates that on the next clock edge the data in R.ICAC_QUEUE will flow into R.ICAC_LAS and the still valid data in R.ICAC_LAS will flow into R.ICAC_OVR. C.INST_ROT reflects all these changes and follow the correct data through to R.ICAC_OVR. Note from the definition of C.INST_ROT that data can never be parsed while it is contained in R.ICAC_QUEUE which is intended for parity checking and holding valid data which would otherwise be lost as R.ICAC_LAS and R.ICAC_OVR fill up but the Icache read mechanism has not yet been turned off. When R.ICAC_LAS and R.ICAC_OVR both contain valid data, QUEUE_FULL will be asserted to cause the mechanism which reads the Icache to spin but R.ICAC_QUEUE allows it to spin on its next address. Data is loaded from R.ICAC_QUEUE into R.ICAC_LAS on the cycle after QUEUE_FULL is removed and parsing continues as described above. Note that QUEUE_FULL can be asserted if the Look Aside queue or the seven-deep dispatch queue is full.

4.4.2 The Dispatch Queue

The NAS may not be able to accept instructions as they are cracked so the NIP has a seven level dispatch queue to hold parsed and cracked instructions until the NAS is able to begin processing them. This queue is at the AS level for the AS_DISP_DISPL, AS_DISP_DISP_PAR, AS_DISP_IREG, AS_DISP_JREG, AS_DISP_KREG, AS_DISP_SEL_IREG, AS_DISP_VP_POL, and AS_DISP_EP. The NIP stages the UPC signals for an additional cycle, effecting UPC_CPC, UPC_BR_SEL, UPC_BR_POL, and UPC_DL. When data is parsed and registered at the R._EPS level as shown in Figure 4-5 below, it may be immediately dispatched or it may be registered in the first level of the dispatch queue, R.PARSED_DATA_1. If the NAS is unable to accept the dispatch while the data is at the PARSED_0 level, indicated by the absence of AS_DISP_REQ, R.OUTPUT_SELECT is incremented. R.OUTPUT_SELECT is the pointer which selects the mux that feeds the AS dispatch signals. PUSH_QUEUE will also be asserted to indicate that the dispatch and all associated data must be pushed on the queue. As newly parsed data becomes available and is not dispatched, it will continue to be pushed on the queue with new data entering level 1, level 1 moving to level 2, and so on until level 6, R.PARSED_DATA_6 is full. Then the NIP will stop parsing requests if AS_DISP_REQ or AS_DISP_RDY are not asserted or a trap is pending. If those conditions do not occur and R.PARSED_DATA_7 becomes valid, the NIP will quit parsing instructions and accepting data from the Icache and QUEUE_FULL will be asserted. It will remain asserted until the NAS begins accepting dispatches or a NAS restart is issued to the NIP, which invalidates all queues. The oldest data in the queue will always be dispatched first, such as R.PARSED_DATA_6 then R.PARSED_DATA_5, and so on.

Figure 4-5 AS Dispatch data and queue

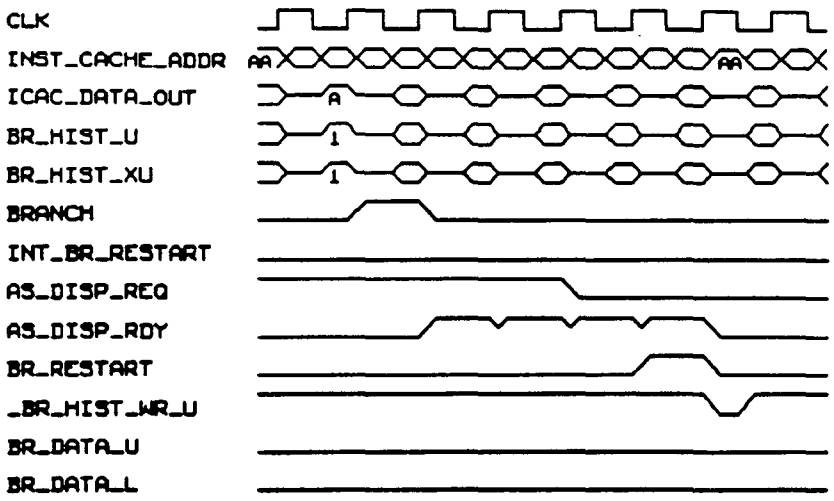


4.5 Branches

Branches are never dispatched as entrypoints but are tagged to the entrypoint of the next instruction. When a branch is parsed, related logic determines if the branch is a branch always or a conditional branch, then a target instruction is chosen which may depend on branch history, discussed in the next section. Whether it is the instruction to be branched to or the instruction immediately following the branch, any condition on which the branch depended as well as the direction taken will be tagged to the instruction actually dispatched, the target instruction. Note that in the case of back to back branches, the first branch will be dispatched tagged to a noop. The NAS will compare the direction taken and condition upon which the branch depended then determine if the action was correct. If incorrect, the NIP will be restarted to the correct address which has been maintained in its pipe staging logic, and the branch history rams at the address of the incorrectly taken branch will be updated.

Figure 4-6 shows the timing for a branch instruction parsed from the third halfword of data immediately after a restart. Data is parsed directly from ICAC_DATA_OUT, rather than data saved in a level of the queue. The branch history corresponding to the Icache data was 0b11, indicating that a branch in this word of data should not be taken but that prediction was restarted the last time a branch in this word was dispatched and executed. BRANCH is asserted on the next clock cycle to show the address logic that a branch has been parsed, but INT_BR_RESTART is not set which means that the branch will not be taken and the branch direction and conditions will be tagged to the instruction immediately following the branch. This is dispatched on the next clock with AS_DISP_RDY and AS_DISP_REQ, although the address, branch selects and direction will be dispatched at the UPC level. The branch information passes to the UPD and UIR1 levels of the NAS pipe (assuming no holds) and BR_RESTART is asserted at the UIR2 level which

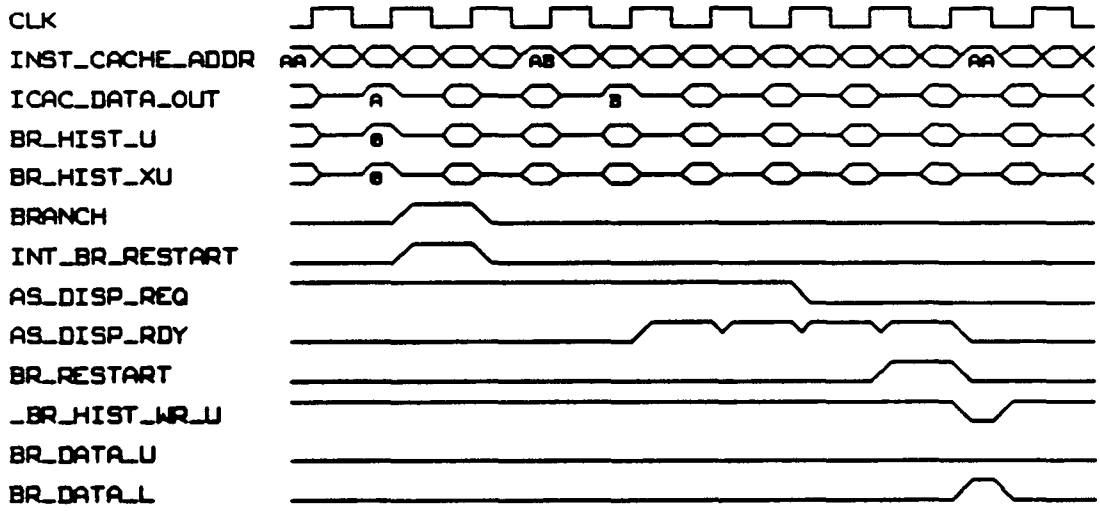
Figure 4-6 Branch restart on branch not taken



means the branch should have been taken. This second restart results in changing the direction bit in the branch history rams and clearing the history bit.

The branch restart shown in Figure 4-7 shows the timing for a branch which was taken but was restarted. This again assumes that the branch is parsed from ICAC_DATA_OUT rather than one

Figure 4-7 Branch restart on branch taken



of the look aside queue levels, and that the branch is the first instruction parsed from that incoming data. On the clock cycle following valid data read from the Icache, the parsed branch results in BRANCH and INT_BR_RESTART both being set to indicate that a branch was parsed and the branch history bits indicate that this branch is to be taken. The address of the branch target is calculated and available at the Icache on the next cycle. Valid data at the target address is assumed to be already encached in Figure 4-7, and is available on the cycle after the target address is presented to the Icache. This new data is parsed and on the next cycle the resulting instruction is available to the NAS tagged with the branch condition and direction. The data passes through the NAS pipeline without holds (an ideal case and not always true) and the branch is found to be incorrectly taken so the NAS restarts the NIP when the branch target instruction

reaches the UIR2 level. The restart resets the INST_CACHE_ADDR to allow the NIP to parse the instruction immediately following the incorrectly taken branch and the branch history and direction rams are updated to show that the last branch dispatched and executed at this address was incorrect, but the next branch parsed at this address will be taken.

4.6 Branch History

The dispatch queue for cracked-but-not-dispatched instructions allows the NIP to continue parsing and cracking instructions even when the NAS is unable to accept them. It is undesirable to have this pre-parsing mechanism halted whenever a conditional branch is parsed but traditionally the branch conditions would need to be known before the direction of the branch could be determined, i.e. whether or not the branch would be taken. To avoid this loss of potentially usable cycles, the NIP uses a two bit branch history mechanism to predict the direction of branches where the most significant bit is used to predict direction and the least significant indicates whether the last branch prediction at this address was correct or required a restart from the NAS. Table 4-3 shows the proper action for the polarity of each bit. For example, if the branch history corresponding to a parsed branch is a 00, the most significant bit will force the branch to be taken while the least significant bit indicates that the branch was taken correctly the last time a branch was parsed from this address. When the branch passes through the NAS pipe, it will

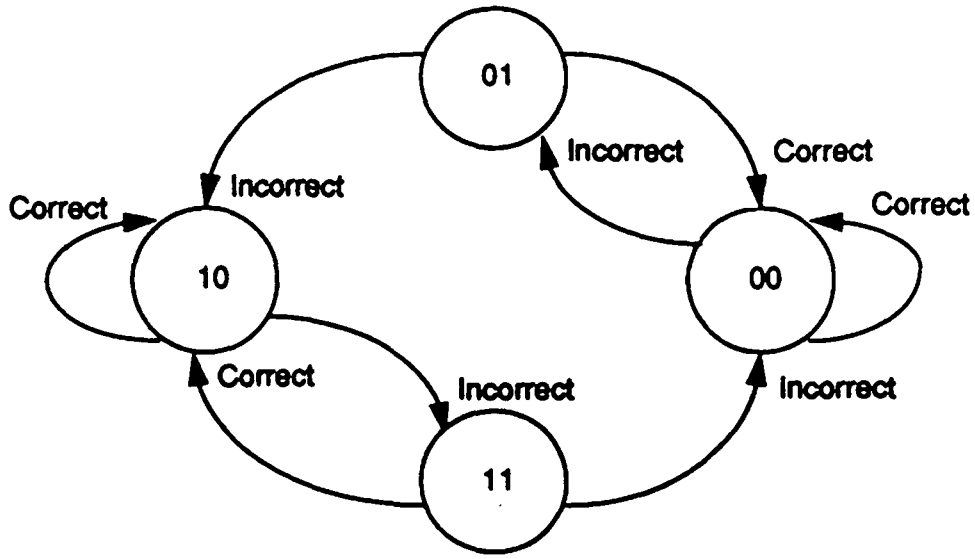
Table 4-3 Branch Direction and History

Direction + History	Action on a branch	Result of last branch
0b00	Take the branch	Taken and correct
0b01	Take the branch	Taken and incorrect
0b10	Do not take the branch	Not taken and correct
0b11	Do not take the branch	Not taken and incorrect

determine if the branch was taken correctly according to the state of the condition on which the branch depended. If the branch was predicted incorrectly, the NIP will be restarted as discussed in the previous section and the branch history bits at the address of the parsed branch will become 0b01, which means take the branch again next time, but records the fact that the branch was incorrectly taken the last time. If the NIP has incorrectly taken the branch and has been restarted on two consecutive parsings of a branch from a word address, the branch history for that address would become 0b10, so that the next time a branch was parsed from this address, the branch would not be taken and the 0 for history implies that the branch was correctly taken the last time. The complete sequencing of the branch history bits according to the correctness of their predictions is shown in Figure 4-8. A correct branch will always clear the history bit, forcing a state of 0b10 or 0b00. A single incorrect branch will always set the least significant history bit to indicate the error but maintain the most significant bit. A second, consecutive incorrect branch at the same word address will clear the history and invert the branch direction, so that the next time a conditional branch is parsed in that word, the opposite direction will be taken.

The implementation of branch history in the NIP uses four 2kx2 purge rams. The rams are organized in two pairs with a pair corresponding to branch direction and history for the upper word and a pair corresponding to the lower word of each long word of data in the Icache. The second bit in each of the rams is used for parity. The rams are addressed by INST_CACHE_ADDR, the same address bus which accesses the Icache so that the branch history data read at any given time corresponds to the data being read from the Icache at that same address. For examples and

Figure 4-8 Branch history updates



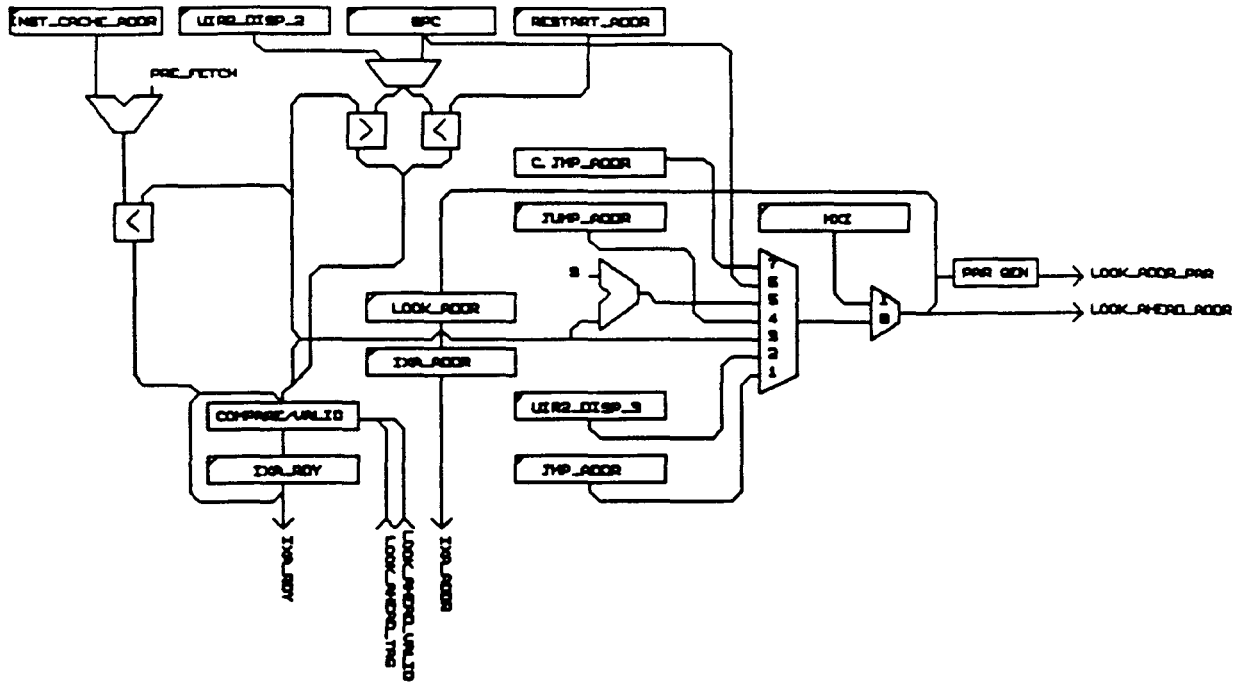
actual timing of branch parsing, execution and restart see the preceding section.

4.7 Look Ahead Mechanism

The NIP look ahead mechanism is designed to continuously request memory data from the NDC subsystem via IXA_RDY and IXA_ADDR. Normal flow for the mechanism places a read address on LOOK_AHEAD_ADDR during the second half of a cycle, and on the next cycle LOOK_AHEAD_VALID and the LOOK_AHEAD_TAG will be available from their rams. The tag will be compared to the upper 18 bits of LOOK_AHEAD_ADDR, which has been saved in R.LOOK_AHEAD. If the tags are the same and the valid bit is set then the data in the Icache at the address pointed to by LOOK_AHEAD_ADDR<13..3> is the data corresponding to the entire address. If the valid bit is not set or the tags do not compare, IXA_RDY will be set and IXA_ADDR will become the address previously at LOOK_AHEAD_ADDR and saved in the mean time at R.LOOK_ADDR. This basic flow can be seen in Figure 4-9 below. Note that the read/write address is selected in the second level of multiplexing on LOOK_AHEAD_ADDR by a phase line derived from the 2x clock which enables the address to appear to run at 2x the normal clock speed. The first level of multiplexing merely selects between various read address sources. This same method is used to generate BRANCH_HIST_ADDR and INST_CACHE_ADDR.

While standard operation is fairly straight forward, special action is required for a number of situations, including jump or branch restarts and reaching a pre-set maximum look ahead value. When UIR2_JMP_RESTART is valid and requests a jump restart as explained in section 4.2, the look ahead mechanism will request tag and valid information from the address in R.JUMP_ADDR rather than the normal R.LOOK_AHEAD + 0x8. This results in a read address mux select equal 4, as shown in Figure 4-9 above, while normal operation would use a mux select of 5. A UIR2_JMP_RESTART which is valid and equals 5 will never restart the look ahead mechanism, but a 1 will restart the mechanism from the address at C.JMP_ADDR with a mux select of 1. If BR_RESTART forces the restart of the NIP from the NAS or INT_BR_RESTART forces the restart from within the NIP, a pair of comparators are used to determine if the look ahead mechanism must also restart. On a restart of the NIP due to BR_RESTART, R.RESTART_ADDR will be compared to R.UIR2_DISP_2 to determine if the restart address in R.UIR2_DISP_2 is previous to

Figure 4-9 NIP Look Ahead logic



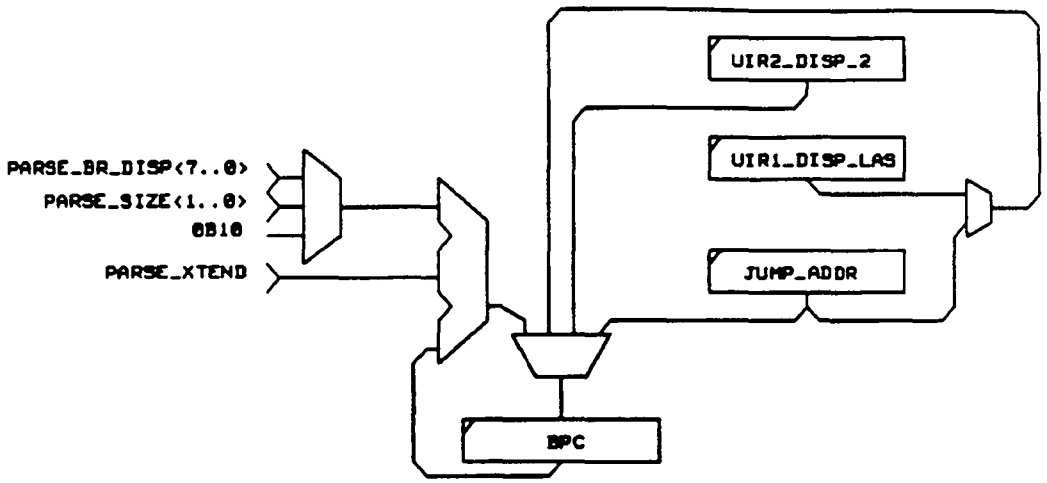
the address at which the mechanism was last started. In addition, R.UIR2_DISP_2 will be compared to the current address at R.LOOK_AHEAD to determine if the new address, which is stored in R.UIR2_DISP_2 is larger than the current address. If the new address is smaller than the previous restart address or larger than the current address, the look ahead mechanism will be restarted in order to fetch the required data for the restart address. If the restart was due to INT_BR_RESTART, the last restart and current address will be compared to R.BPC since it will contain the new address for internal branch restarts. The BR_RESTART assertion will use mux select 2 and the INT_BR_RESTART restart will use mux select 6 if a restart of the mechanism is necessary

One other special condition which involves the look ahead mechanism is reaching the prefetch maximum. R.PRE_FETCH is a six bit register which is set only by scan. This represents the maximum distance, in long words, that the look ahead mechanism may prefetch beyond where instructions are currently being parsed. R.INST_CACHE_ADDR contains the last read address of the Icache and R.PRE_FETCH will be added to this value then compared to R.LOOK_AHEAD to see if R.LOOK_AHEAD is now greater than the sum of the other two. If not, the mechanism will continue to request memory data via the NDC subsystem. If it is larger than the sum, the look ahead mechanism will hold at that address until R.INST_CACHE_ADDR increases so that R.LOOK_AHEAD is no longer larger than the sum or until a NIP restart which requires a restart of the look ahead mechanism.

4.8 BPC: Maintaining the Current Program Counter

The NIP maintains a copy of the current program counter in R.BPC. The program counter is updated each time a valid instruction is parsed to be either dispatched to the NAS or pushed on the seven deep queue, which is indicated by VALID_DATA. R.BPC can be updated from one of four sources:C.NEW_BPC, R.JUMP_ADDR, R.UIR2_DISP_2 or C.JMP_ADDR as shown in

Figure 4-10. R.JUMP_ADDR contains the last restart address used by a UIR2_JMP_RESTART
Figure 4-10 Generation of BPC



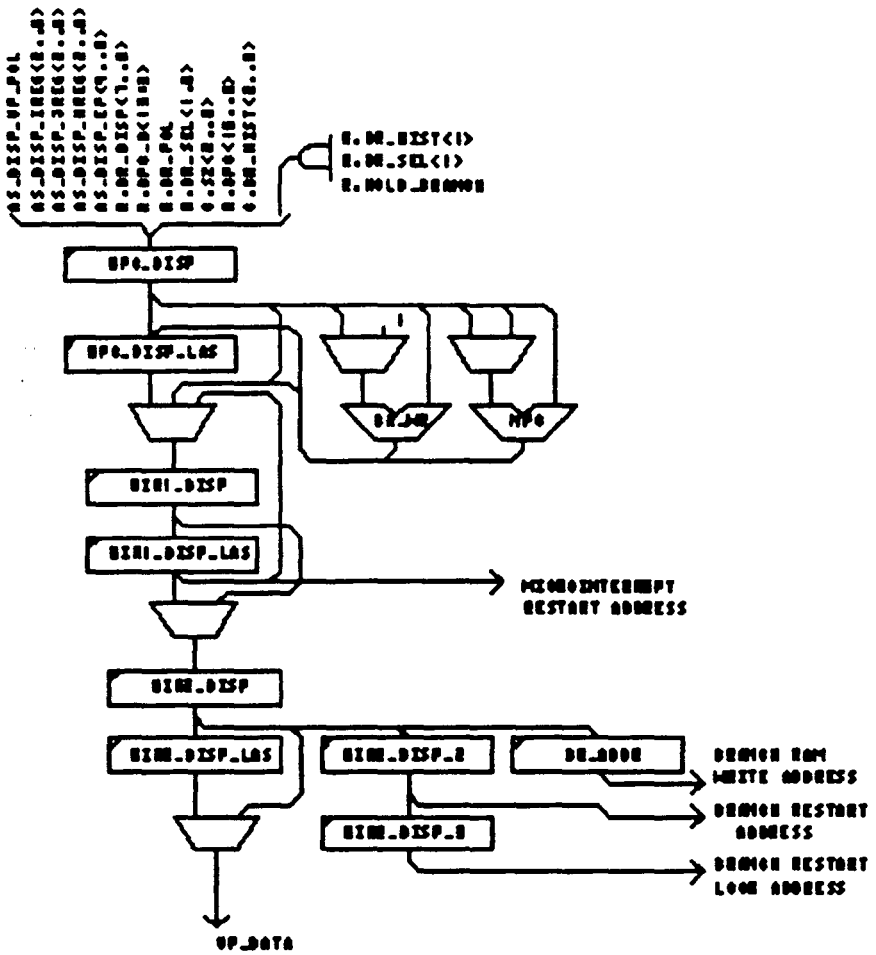
from the NAS or loads new data from YBUS_DATA when there is a valid jump restart on UIR2_JMP_RESTART. Therefore R.JUMP_ADDR will be loaded into R.BPC when R.UIR2_JMP_RESTART is a 0b010 or 0b011. C.NEW_BPC uses the current R.BPC added to the various size and branch fields to determine a new program counter value based on the current program counter plus the size of the current instruction, in the case of normal sequential instruction flow or branches not taken, or based on the current program counter plus the branch displacement in the case of branches that are taken. C.NEW_BPC will be selected except in the case of external restarts from BR_RESTART or UIR2_JMP_RESTART. R.UIR2_DISP_2 contains the address to which the NIP will be restarted if branch history provided an incorrect branch direction and a BR_RESTART is subsequently received. C.JMP_ADDR will provide the new program counter when a microinterrupt is received. C.JMP_ADDR may be either R.UIR1_DISP_LAS or R.JUMP_ADDR depending on whether R.UIR1_DISP_LAS contains valid data. If the valid data has not filtered down to that level of the pipe since the last UIR2_JMP_RESTART or BR_RESTART forced restart as will be indicated by a valid R.UINT_RESTART, C.JMP_ADDR will refer to R.JUMP_ADDR. If R.UINT_RESTART is invalid, C.JMP_RESTART will look at R.UIR1_DISP_LAS.

4.9 Pipe Staging

The NIP address logic contains staging which is a duplicate of the NAS pipe staging. The NIP staging contains data which it must keep for vector processor dispatches or for restarts at various times as indicated by the pipe stage controls. For more detail on the levels and controls of the pipe, see the appropriate section of the NAS section of this document. The NIP staging and the signals which are staged are shown in the following Figure 4-11.

The NIP stages many signals to generate two restart addresses at a later stage in the pipe. Timing at the head of the pipe did not allow the addresses to be generated before being placed in the pipe. One address which is generated is C.BR_WR, the branch restart address which is generated by adding the branch displacement or a 0b01 to the program counter of the original branch. If the branch was taken, i.e. the original branch generated a target address by adding its program counter to the branch displacement, then C.BR_WR will be generated by adding 0b01 to the program counter of the original branch which would be the address of the target instruction if

Figure 4-11 NIAD Pipe and data



the branch was not taken. And if the branch was not taken, C.BR_WR will be generated by adding the branch displacement to the original program counter so that C.BR_WR contains the address that would have been for the target address if the branch had been taken. Whether or not the branch was taken is determined by the state of R.HOLD_ADDR, R.BR_SEL<1>, and R.BR_HIST<1>. If any is not set, R.BR_DISP will be added to R.BPC_B, else 0b10 will be added to R.BPC_B so that it will point to the instruction following the branch. Note that R.BPC can be placed in the pipe in the position normally occupied by R.BPC_B but only if the instruction is not a branch or a branch with PSW_TR or PSW_SEQ set.

The second address generated within the stages of the pipe is C.NPC. This is the address of the next instruction and enables the NIP to start with the instruction following the last completed instruction on microinterrupts. This address uses all inputs shown in Figure 4-11 except the AS_DISP level data to determine the proper next address. In the normal flow of data, the next address is effected by the size and extended state of the current instruction as well as if the current instruction is a branch. If the instruction is branch, the C.SZ field should be 0, and likewise if the instruction is not a branch R.BR_DISP will equal 0x00. This enables C.NPC to add all fields except during branches not taken. If a branch is not taken, as indicated by the and of R.HOLD_ADDR, R.BR_SEL<1>, and R.BR_HIST<1>, the R.BR_DISP field at the adder is selected to be 0's so that the adder may again add all fields to get the proper next instruction

address. The resulting C.NPC proceeds through the pipe stages to UIR1_DISP_LAS, at point any microinterrupt requiring that address should occur and the address will be used to access the new instruction.

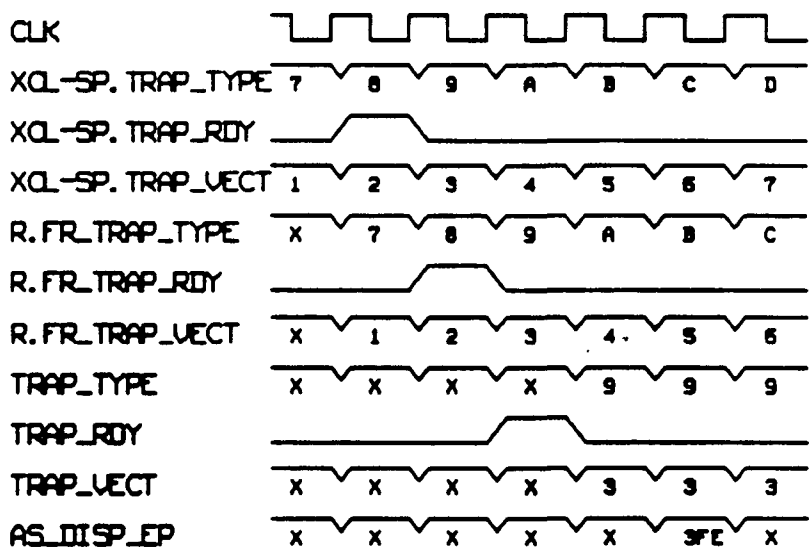
The third set of inputs to the pipe staging are the AS_DISP level inputs. These signals follow through the pipe as directed by the control signals from the NAS until they reach the UIR2_DISP level. At that point they may be used to dispatch the NVP or they may go on the UIR2_DISP_LAS and be dispatched from that level, depending on the state of R.VP_DISP_HAZ.

4.10 Traps

Occasionally, situations arise which require specific action by the Neptune Scalar Processor which are not part of any program it may be running at that time. Many of these situations have been grouped as Traps, and are handled in the NIP. When such a situation arises, the NIP evaluates and prioritizes pending traps, then will dispatch the highest priority trap entrypoint to the NAS using the standard AS level signals and handshakes as described above. There are two sources of traps: the NAS and the NCU through the XCL. Each trap type is discussed in detail below.

The source of the traps may be the NCU through the XCL, which has a four bit trap-type field that is translated to various entrypoints by the NIP and dispatched only once. These traps are generally related to system level or program activity. A ready signal, XCL_SP.TRAP_RDY may

Figure 4-12 NCU Trap Timing



be received by the NIP at any time, and it as well as XCL-SP.TRAP_TYPE<3..0> and XCL-SP.TRAP_VECT<11..0> are registered in free-running registers. That level is checked for a valid ready signal on R.FR_TRAP_RDY, which will be registered as TRAP_RDY and allows TRAP_TYPE<3..0> and TRAP_VECT<11..0> to register new data. The ready signal and trap type are used by the NIP to generate a trap entrypoint a few cycles later, as will be discussed below, but the TRAP_VECT<11..0> is sent on to the NAS and not used by the NIP.

The other source of traps is signals from the NAS, including two types of traps which require quite different action. IDLE_TRAP and AR_TRAP are from the NAS Condition Code Register and a combination of signals from the NAS Program Status Word, respectively. These two traps are

dispatched when the signals are asserted, and if they remain asserted may be dispatched several times. It may take several clocks before the corresponding trap entrypoint is sent to the NAS, particularly if a NCU trap is pending. The third trap type is represented by PSW_TR and PSW_SEQ, signals which are taken immediately from the Program Status Word. Both of these signals will results in a single instruction in the NAS pipe at any time and the NIP will give no additional dispatches until the current dispatch has completely executed. The assertion of PSW_TR will result in the sequential instruction execution with a trace trap dispatched after every instruction. The assertion of PSW_SEQ will result in the sequential instruction dispatch mode with no trace traps between instruction, since it is not a true trap but effects the flow of instructions exactly as PSW_TR does. PSW_TR and PSW_SEQ will continue to effect instruction flow, and dispatch traps in the case of PSW_TR as long as these signals are asserted. Note that trap entrypoints are always dispatched as soon as possible after they have been received, or for PSW_TR after each instruction, and trap entrypoints will never be dispatched from the seven-deep parsed instruction queue.

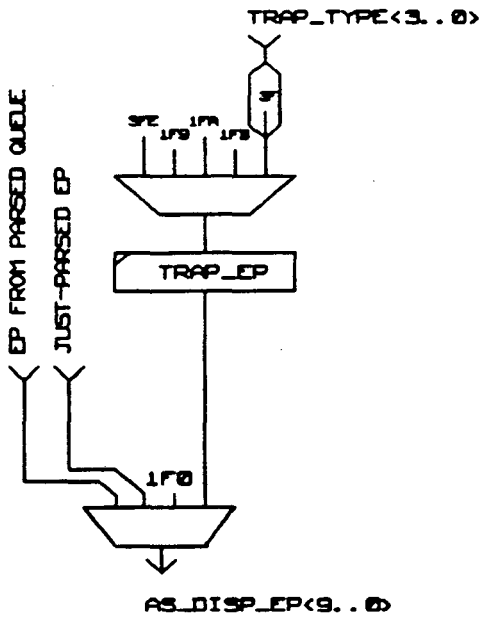
Entrypoints for the various trap types discussed above are listed in Table 4-4 below. Traps are prioritized in the following order: CU traps, AR_TRAP or IDLE_TRAP (will not occur simultaneously), and trace traps due to the assertion of PSW_TR. If PSW_TR or PSW_SEQ are asserted, any other traps received will be handled like instructions since they will complete execution before the next instruction or trap will be dispatched.

Table 4-4 Trap Entrypoints

Trap Type	CU Trap-type	Entrypoint
Xmiti	0	0X1F8
Trace	-	0X1F9
Arithmetic	-	0X1FA
Idle	-	0X1FB
Patu	4	0X1FC
Pmod	5	0X1FD
Pref	6	0X1FE
Pich	7	0X1FF
Ctrsg	8	0X3F8
Trap	9	0X3F9
Pate	B	0X3FB
Deadlock	D	0X3FD
Unimplemented	-	0X3FE
Format 8	-	0X3FF

The following illustration shows the flow and selection of the trap entrypoint in the NIP. If the trap is from the CU the bits of the trap-type, XCL_SP.TRAP_TYPE<3..0>, are combined with a 0x3f in bit locations 8..3 using trap-type bit 3 as the entrypoint bit 9 and bits 2..0 as entrypoint bits 2..0 to generate the CU trap entrypoint corresponding to trap-type. The AR_TRAP, IDLE_TRAP, and PSW_TR entrypoints are hard coded to the values shown, as is the unimplemented trap code. The unimplemented trap code is used only when handshakes from the XCL indicate a CU trap request but the type indicated by XCL_SP.TRAP_TYPE<3..0> is not a recognized code, for example a trap type of 0xE would result in an unimplemented or 0x3FE entrypoint dispatched on AS_DISP_EP.

Figure 4-13 Trap entrypoint flow



4.11 Deadlocks

Deadlocks occur in the NIP when a branch and its target instruction loop indefinitely on themselves, an example of which is shown in the small code segment in Figure 4-14 where the branch will continue to loop in L1 until the test conditions proves false. The deadlock is tested for

Figure 4-14 An example of deadlockable instructions

```
L1:  rcv.w #0x3000, a2
     bra.f L1
```

by adding the branch displacement to the target instruction size to determine if the sum is zero, and if so the instructions are deadlocked. When a deadlock is detected, UPC_DL is asserted which informs the NAS of the deadlock situation. The NAS informs the NCU of a deadlock condition resulting from a deadlockable instruction and ignores a deadlock condition from non-deadlockable instructions. A deadlock condition from deadlockable instructions may result in a restart from the NCU.

A deadlock can occur with either conditional or unconditional branches. A deadlocked unconditional branch generally indicates a process which has progressed incorrectly and may be unresolvable without terminating the process. A deadlocked conditional branch may simply be waiting at the end of a process for other processor heads to reach the same point in their processes and will eventually be restarted by a trap from the NCU. For more information following the notification of the NAS of a deadlocked process, see the appropriate NAS section.

4.12 Parity

The NSP maintains good parity throughout the various subsection interfaces and in all rams to help with error detection. The major address buses read by the NIP, i.e. MXI_ADDR and

YBUS_DATA are checked for valid parity whenever the appropriate control signals indicate that these buses contain data for use by the NIP. MXI_DATA contains the data to be written to the Icache and parity on this bus is checked by the rams when the data is written. The major buses driven by the NIP, i.e. IXA_ADDR and AS_DISP_DISPL have valid parity driven at all times. Parity checking may be disabled in the NIP gate arrays with the scannable parity enable bits.

Parity error detection is of primary importance to insure data integrity in the twenty rams used for the Icache, Look Ahead and Icache tags, Look Ahead and Icache validity, and various branch history functions. Correct parity is in all rams at all location at all times. Parity is checked on reads of data from all rams when enabled. Parity on all ram data is checked in the NPAR gate array after first registering the data from the rams. If the data has bad parity, the registers will be held and all potential parity errors combine to generate NPAR_PAR_ERR, which eventually will stop clocks on the board by generating a halt signal. The NPAR detectable parity errors are listed below with the error register, signals which are checked to generate an error held by that register, the data that will be held as a result of that detected error, the rams which sourced the bad data and the address of that data.

Two internal address buses always have valid write parity: LOOK_AHEAD_ADDR and INST_CACHE_ADDR because these addresses are written to tag rams. These are a special case, since parity for each address is generated in the NIADs but while the upper gate array generates parity on the full sixteen bits of its address, the lower NIAD generates parity only on the upper two bits of the address. This is because only the most significant 18 bits of address (16 from the upper array and 2 from the lower) are recorded in the tag rams.

npar_par_err:

nip:par:r.br_que_par_err

signals checked:br_hist_u<2>, br_hist_xu<2>, br_hist_l<2>, br_hist_xl<2>

data held:r.icac_queue<79:72>

ram:nip:br_hist_u:ram, nip:br_hist_xu:ram, nip:br_hist_l:ram, nip:br_hist_xl:ram

ram address:nip:i_addr:r.inst_cache_addr<13:3>, nip:i_addr:r.inst_las_{1,2,3}

nip:par:r.inst_tag_par_err

signals checked: inst_tag<17:0>, inst_tag<20:18>

data held:r.inst_tag<2>

ram:nip:i_tag:ram

ram address:nip:i_addr:r.inst_cache_addr<13:3>, nip:i_addr:r.inst_las_{1,2,3}

nip:par:r.inst_valid_par_err

signals checked:inst_valid<2>

data held:r.inst_valid<2>

ram:nip:i_valid:ram

ram address:nip:i_addr:r.inst_cache_addr<13:3>, nip:i_addr:r.inst_las_{1,2,3}

nip:par:r.look_tag_par_err

signals checked:parser_look_tag<17:0>, parser_look_tag<20:18>

data held:r.look_tag

ram:nip:look_tag:ram

ram address:nip:i_addr:r.look_ahead<13:3>, nip:i_addr:r.look_las_{1,2,3}

nip:par:r.look_valid_par_err

signals checked:look_ahead_valid<2>

data held:r.look_valid

ram:nip:look_valid:ram

ram address:nip:i_addr:r.look_ahead<13:3>, nip:i_addr:r.look_las_{1,2,3}

nip:par:r.que_par_err

signals checked:icac_data_out

data held:r.icac_queue<71:0>

queue valid:r.icac_queue<80>

ram:nip:icache:ram

ram address:nip:i_addr:r.inst_cache_addr<13:3>, nip:i_addr:r.inst_las{1,2,3}

The address gate arrays check parity on the YBUS_DATA as R.UPPER_YBUS<2..0> (the upper three bits or ring bits) and R.JUMP_ADDR<13..0>, which accept new data only on certain combinations of UIR2_JMP_RESTART<2..0> and UIR2_VAL. The arrays also check parity on the MXI_ADDR at R.MXI_ADDR only when R.MXI_RDY is valid. Parity errors from either of these areas will be registered and combinatorially generate NIADX_PAR_ERR. The errors, where detected and which signals are held are described below.

niad0_par_err:

nip:l_addr:r.ybus_par_err

signals checked:ybus_data<15:0>, ybus_par<3:2>

data held:r.jump_addr, r.ybus_par, r.upper_ybus

nip:l_addr:r.mxi_par_err

signals checked:mxi_addr<15:0>, mxi_addr_par<3:2>

data held:r.mxi_addr, r.mxi_par

niad1_par_err:

nip:u_addr:r.ybus_par_err

signals checked:ybus_data<31:16>, ybus_par<1:0>

data held:r.jump_addr, r.ybus_par, r.upper_ybus

nip:u_addr:r.mxi_par_err

signals checked:mxi_addr<31:16>, mxi_addr_par<1:0>

data held:r.mxi_addr, r.mxi_par

icd_wr_par_err:

signals checked:mxi_data, mxi_par

ram:nip:icache:ram

ram address:nip:l_addr:r.mxi_addr<13:3>, nip:l_addr:r.mxi_las_{1,2}

ram we:nip:misc:r.mxi_rdy, r.mxi_rdy_{1,2}

ict_wr_par_err:

signals checked:inst_addr<31:14>

ram:nip:i_tag:ram

ram address:nip:l_addr:r.mxi_addr<13:3>, nip:l_addr:r.mxi_las_{1,2}

ram we:nip:misc:r.mxi_rdy, r.mxi_rdy_{1,2}

lat_wr_par_err:

signals checked:look_addr<31:14>

ram:nip:look_tag:ram

ram address:nip:l_addr:r.mxi_addr<13:3>, nip:l_addr:r.mxi_las_{1,2}

ram we:nip:misc:r.mxi_rdy, r.mxi_rdy_{1,2}

5 NDC Subsystem

The functionality of the NDC subsystem is presented in this chapter. References will be made to the Neptune Scalar Processor Block Diagram which is separate from this document.

5.1 Overview

The NDC subsystem is responsible for arbitrating the priority of memory requests, accessing the data and PTE caches, and issuing requests to the cross bar for memory/communication accesses. The NAS subsystem issues memory requests which are needed to execute instructions, and the NIP subsystem issues memory requests for instruction cache look ahead data. In addition to the requests from the NAS and NIP the NDC subsystem also generates requests for data cache block prefetch data and requests for vector address generation. The NDC subsystem contains four gate array types: Neptune Data Cache control (NDC), Neptune Data Path (NDP), Neptune Address Generation (NAG), and Neptune Physical Address (NPA). Forty-two rams are used for the data and PTE caches with ECLinPS parts for address staging and control.

Figure 5-1 shows the NDC Subsystem control block diagram. The majority of the control is implemented in the NDC gate array. The Neptune Scalar Processor block diagram shows the address and data paths for the NDC subsystem. The data paths for the subsystem are implemented in the NDP gate arrays, the logical address generation within the NAG gate arrays, and the physical address translation in the NPA gate arrays.

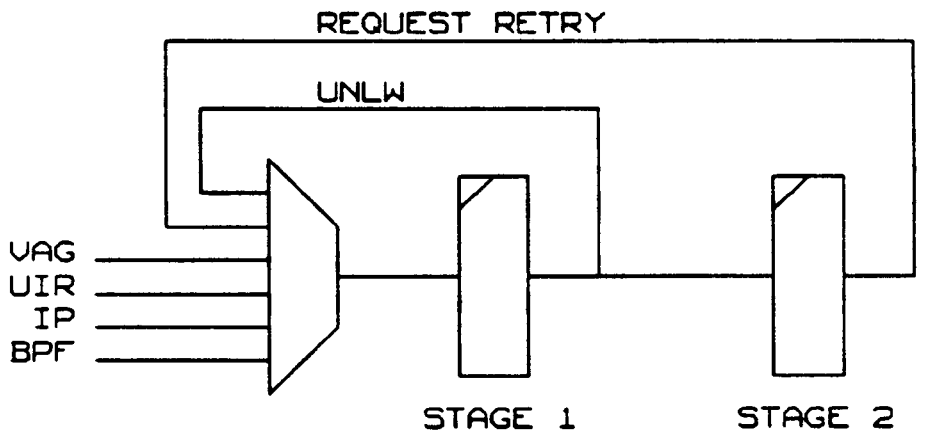
5.2 Data Cache Pipe

The data cache pipe refers to the two stages of registers which all requests use to access the data and PTE caches. The following sections describe the staging registers and paths for special operations. Later sections will cover the data and PTE caches in detail.

5.2.1 Pipe Stages

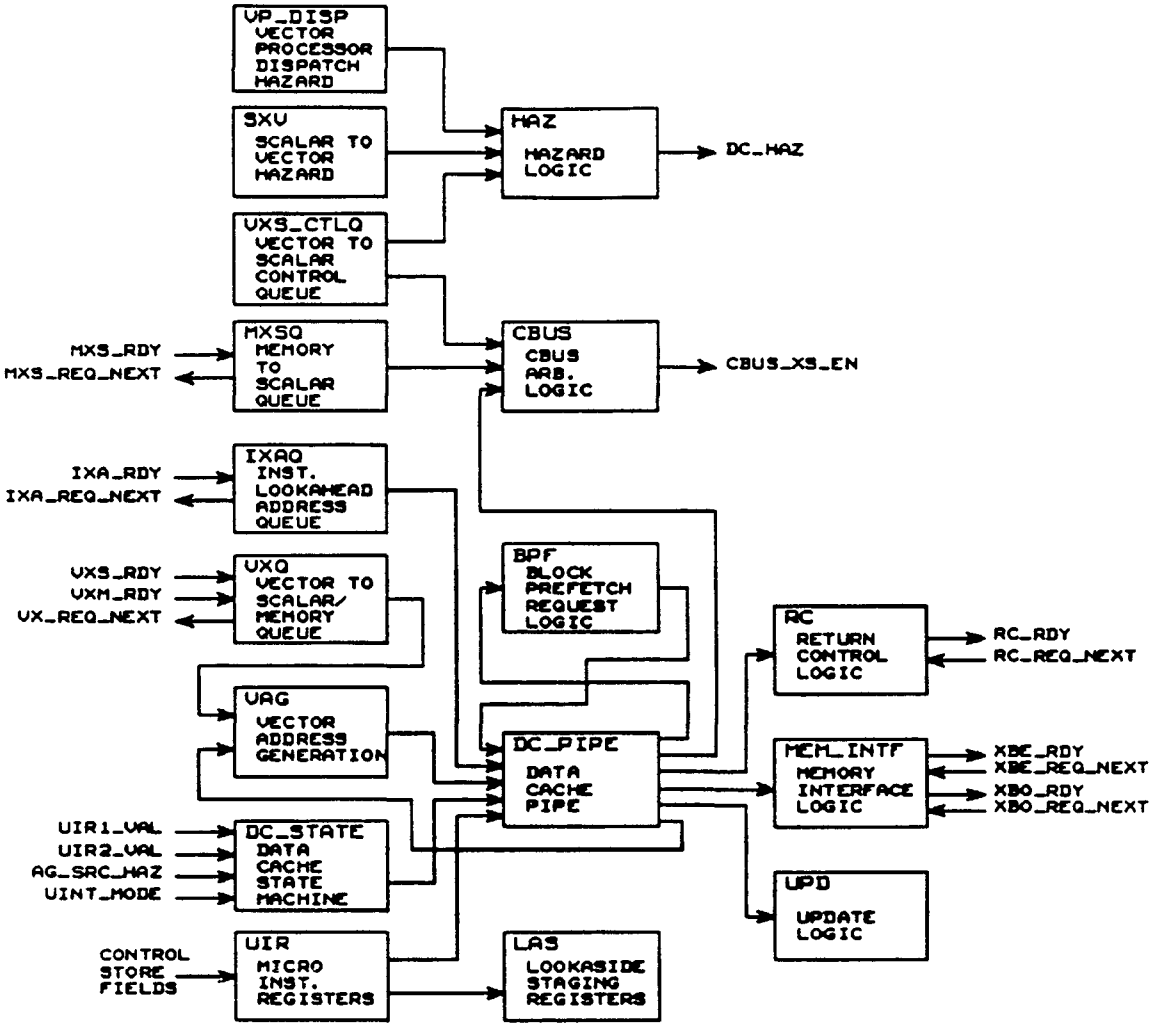
Figure 5-2 shows a block diagram for the data cache pipe staging logic. The input mux for the

Figure 5-2 Data Cache Pipe Block Diagram



staging logic selects the information from the highest priority request. Request arbitration logic is

Figure 5-1 NDC Subsystem Control Block Diagram



used to select one of the input request sources. The selected request is registered in the first stage of the pipe. The data and PTE caches are accessed between the first and second stage registers. The cache tags are compared prior to the second stage registers. Cache access status (data cache hit/miss, PTE cache hit/miss/access violations) is available from the second stage registers.

All control is implemented in the NDC gate array, data is staged in the NDP gate arrays, and addresses are staged in the NAG gate arrays. The output of the request selection mux on the NAG gate arrays is brought off the NAG gate arrays and used to access the cache rams (the cache rams have an input register which is the data cache pipe first stage).

As shown in Figure 5-2 the request at the first stage register is checked for unaligned long word (UNLW) access. If an UNLW request is detected then a request to complete the UNLW is issued.

When a PTE miss occurs the data cache pipe stages are held while the PTE caches are written. The request which missed the PTE cache is contained in the second stage register. To retry the request the input mux selects the second stage data as input to the first stage register. Two clocks are issued which leaves the retried request back in the second stage registers. The data that was in the first stage registers went to the second stage registers and back to the first again.

5.2.2 Request information

All requests selected to the data cache pipe stage consist of an eleven bit memory operation field (DC_MEM_OP), a two bit size field (DC_SIZE), a three bit segment field (DC_SEG), a 32 bit even side logical address (SA0), and a 32 bit odd side logical address (SA1). If the request is a write then data is required, and if the request is a read destined for the register file then a five bit register select is required (DC_REG_SEL). The logical addresses are staged by the NAG gate arrays, data is staged by the NDP gate arrays, and all other fields and control is in the NDC gate array.

The eleven bit memory operation field is used to specify the type of operation to be performed. The IP and BPF requests have most of the bits of the field forced to specify the request type. The UIR interface is able to specify each bit using micro code. The table below describes each bit of the field.

Table 5-1 Memory operation field

<2..0> MEM_OP_DST

000 - IP, Instruction Processor
001 - IP_NOFLT, Instruction Processor non-faulting request
010 - VP, Vector Processor
011 - DC, Data Cache
100 - RF, Register File
101 - RF_DC, Register File and Data Cache
110 - Not Used
111 - RF_DC_BPF, Register File and Data Cache
(Initiate block prefetch if data cache misses)

<5..4> MEM_OP_AT_TYPE address translation type

00 - NAT, no address translation (physical addressing mode)
01 - CRAT, communication register address translation
10 - VAT, virtual address translation
Write to data cache if hit occurs, do not read from data cache.
11 - VAT_DC, virtual address translation with data cache enabled
Write to data cache, read from data cache.

<11..6> MEM_OP_TYPE

If (MEM_OP_REQ and MEM_OP_AT_TYPE is NAT and <7..6> equal zero)
All purges reset block prefetch mechanism
0x0000 - No operation, does not reset block prefetch
x10000 - Reset block prefetch
1x0000 - Purge PTE entry
x10100 - Purge page reference ram
1x0100 - Purge page modified ram
x11000 - Purge PTE thread validity
1x1000 - Purge PTE non-thread validity
x11100 - Purge data cache thread validity

1x1100 - Purge data cache non-thread validity

If (MEM_OP_REQ and MEM_OP_AT_TYPE is not CRAT
and <7..6> does not equal zero)

xxxx01 - Read request
xxxx10 - Write request
xxxx11 - Test and modify (TAM) request
xxx1xx - Vector address generator start
xx1xxx - Store scalar extended
x1xxxx - Operate under mask
1xxxxx - Vector of indices

If (MEM_OP_REQ and MEM_OP_AT_TYPE equal CRAT)

xxxxx0 - No return data from cross bar
xxxxx1 - Return data from cross bar

If (PTE_OP_REQ)

000xxx - Write second level PTE entry to PTE cache rams (DATA, TAG, and VALID)
001xxx - Write thread level PTE entry to PTE cache rams (DATA, TAG, and VALID)
101xxx - First level PTE translation and read PTE table in memory
110xxx - Second level PTE translation and read PTE table in memory
111xxx - Thread level PTE translation and read PTE table in memory

5.3 Request Types

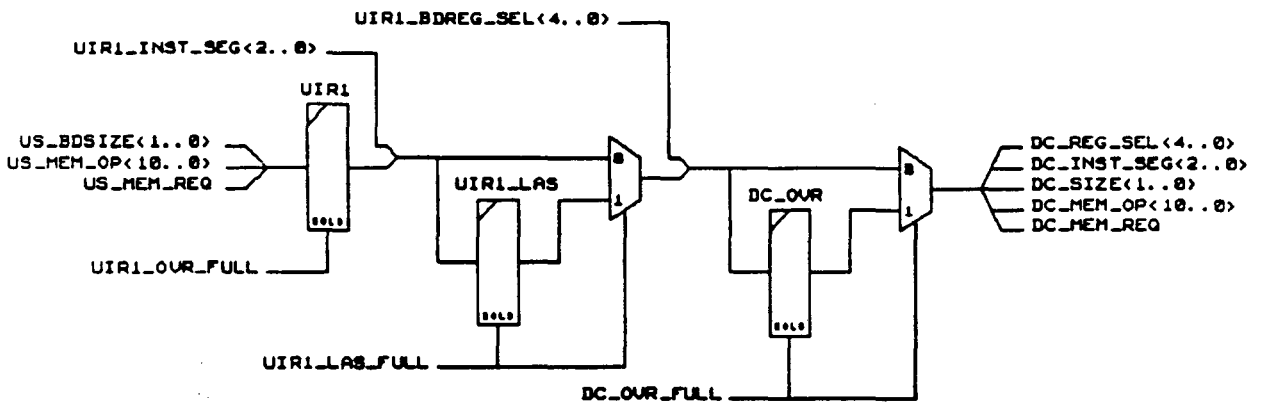
There are four sources of requests for the data cache pipe. These four are: micro instruction (UIR) requests, instruction processor (IP) look ahead requests, vector address generator (VAG) requests, and block prefetch (BPF) requests. The VAG mechanism is started by a UIR request which has the VAG start bit set. The BPF mechanism is started by a UIR request which has block prefetch enabled and misses the data cache.

A request is selected to the data cache pipe based on a fixed priority arbitration scheme. The priority of the requests is (from highest to lowest): VAG, UIR, IP, BPF. Unaligned long word (UNLW) requests require two access cycles to the data cache or memory. This occurs because the data is aligned in such a way that two accesses to either the even or odd side is required. UNLW requests are handled by detecting that a request is an UNLW and issuing a second request. The second request for an UNLW request always has higher priority than the four sources of requests.

5.3.1 UIR requests

The UIR requests are originated from NAS subsystem micro code. Figure 5-3 shows the staging used for the UIR request information. The micro code fields enter into the UIR1 register stage. The UIR1_INST_SEG field is the most significant three bits of the instruction address for the micro code. The NPSW gate array on the NAS subsystem generates the instruction address. The UIR1_BDREG_SEL field is generated by the NRFA gate arrays of the NAS subsystem. The field

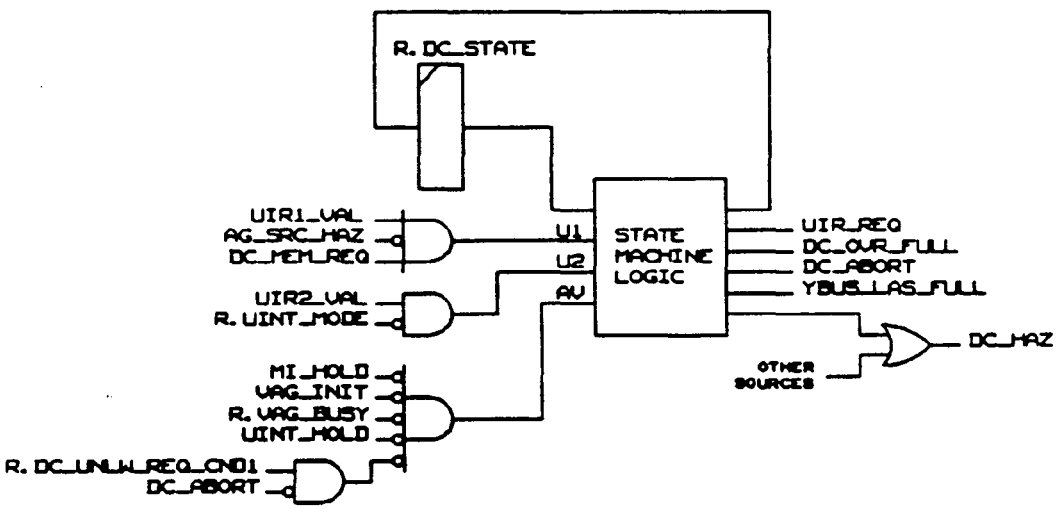
Figure 5-3 UIR Interface Staging



is generated from micro code fields as well as register select fields from the dispatched instruction. The output fields of the UIR request staging logic are feed into the data cache pipe when the UIR request wins arbitration.

A state machine is used to monitor the NAS subsystem pipe control signals (UIR1_VAL, UIR2_VAL, ...) and issue requests to the data cache pipe request arbitration logic. Figure 5-4 shows the state machine functionality. The state machine uses three inputs variables to

Figure 5-4 UIR Request State Machine Logic



determine the next state. These variables are U1, U2, and AV. The variable U1 when asserted implies that a valid request is at the UIR1 level of the NAS subsystem pipe. The signal UIR1_VAL indicates that the UIR1 level registers contain a valid instruction, AG_SRC_HAZ implies that the register from the register file which is used for address generation does not have a hazard, and DC_MEM_REQ specifies that the micro instruction is issuing a memory request. The variable U2 specifies that there is a valid instruction at the UIR2 level of the NAS subsystem pipe and that the NAS subsystem is not in micro interrupt mode. The variable AV means that the data cache pipe is available to accept a UIR request. The data cache pipe is available unless a higher priority request is ready (a VAG request or the second request for an UNLW request), or the data cache pipe is being held (UINT_HOLD or MI_HOLD).

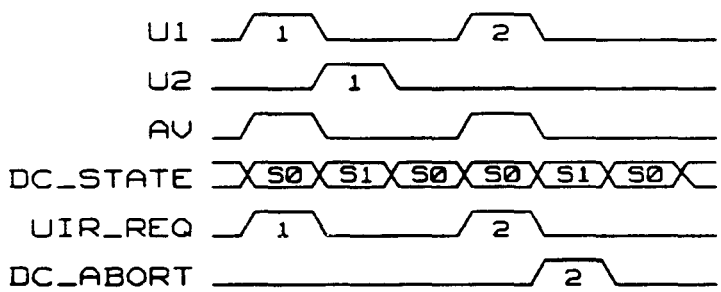
The state machine generates signals to control the UIR request interface. The signal UIR_REQ is used by the data cache pipe arbitration logic to specify that a UIR request is ready to enter the data cache pipe. The signal DC_OVR_FULL is used to hold and select an overrun register, as shown in Figure 5-3. DC_ABORT is used to abort the UIR request that entered the data cache pipe on the previous cycle. YBUS_LAS_FULL is used to hold and select a YBUS look aside register within the NDP gate arrays. The final signal generated is DC_HAZ. This signal is used to hold the NAS subsystem UIR1 level pipe stage.

The signal UINT_MODE from the NUS gate array of the NAS subsystem is used to specify that the NAS has been micro interrupted. The signal is associated with the UIR1 level of the pipe and is asserted as long as micro interrupt code is at the UIR1 pipe stage. On the NDC subsystem the signal is used to hold the DC_OVR register level and UIR request state machine.

Figure 5-5 is the state transition diagram for the UIR request state machine. The state transition diagram shows the outputs (in square brackets) for each set of input values and present state. Within each circle representing a state is the name of the state and the value assigned to the state. The discussion that follows refers to the states by the state name. The state value is included for reference when trying to follow actual hardware values.

Three example sequences will be walked through to show the flow of the UIR request state machine. The first will be a single request without any register stages being held. Figure 5-6 shows a timing diagram of the sequence. The first request starts with U1 being asserted (as

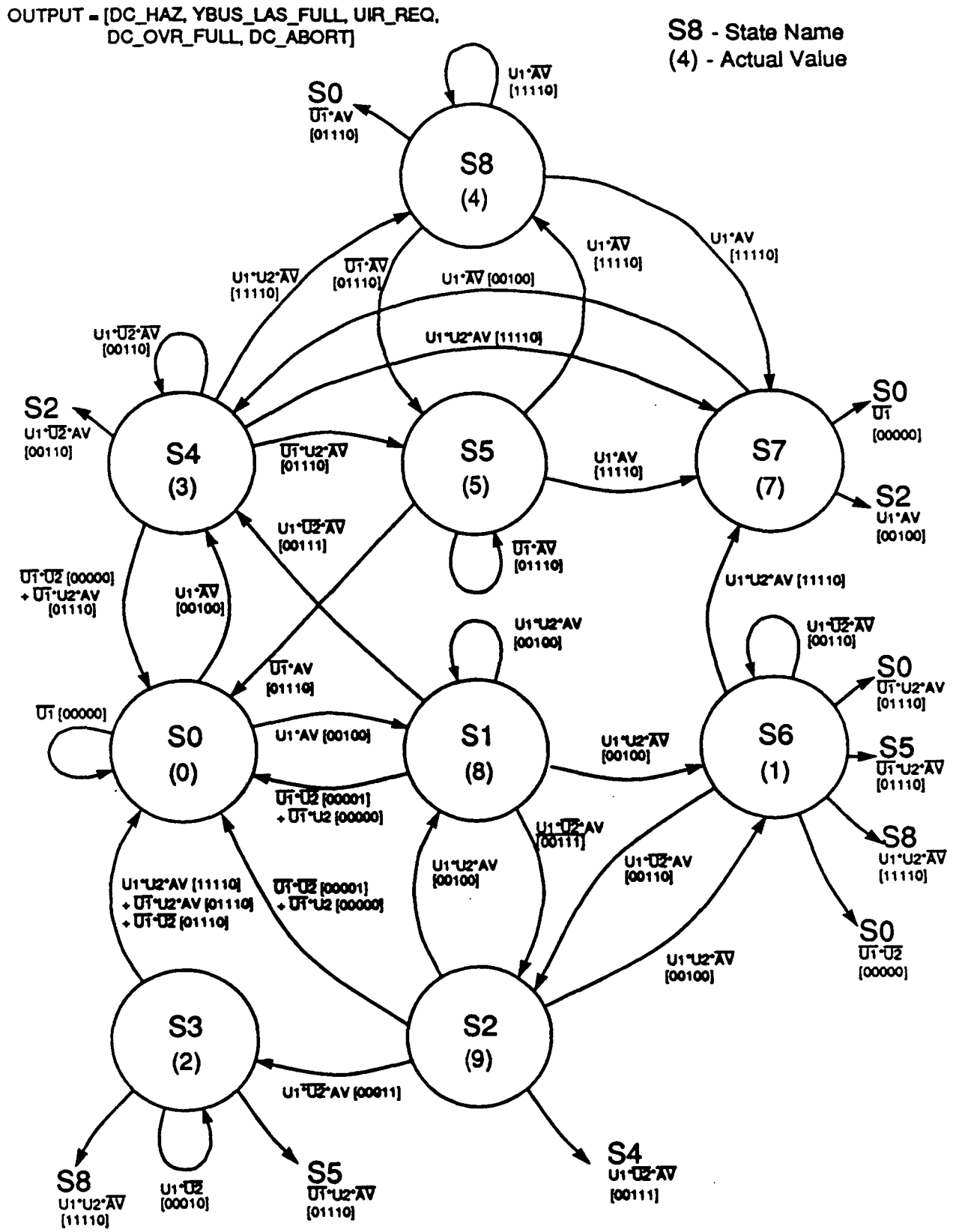
Figure 5-6 UIR request: Example one



mentioned before U1 is generated from UIR1_VAL). The idle state, S0, looks for the assertion of U1 and when asserted checks whether the data cache pipe arbitration logic will accept a UIR request. In the example AV is asserted so the state machine logic asserts UIR_REQ to inform the DC pipe arbitration logic of a request and transitions to state S1. State S1 checks for U2 indicating that the previously made request proceeded to the second level micro instruction stage of the NAS subsystem. Since U2 is asserted and U1 is not (indicating that a second request is not at the UIR1 stage) the state machine returns back to the idle state, S0.

The second request of the sequence is similar to the first except that at state S1, U2 is not asserted. This can happen when the NAS subsystem receives an instruction from the NIP subsystem that is tagged with a branch. The NAS subsystem checks to make sure the NIP followed the correct branch target at the UIR1 pipe stage. If the correct target was predicted then the micro instruction at the UIR1 stage proceeds to the UIR2 level (resulting in U2 being asserted). Otherwise, when an incorrect target was followed the UIR1 stage micro instruction is dropped (and the NIP is restarted to follow the correct target). The UIR request state machine already started the memory request into the data cache pipe, so it must be aborted. The signal DC_ABORT is used to cause the request at the first level of the data cache pipe to be marked invalid

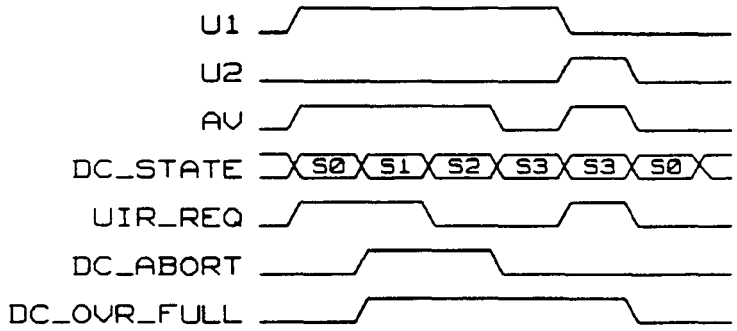
Figure 5-5 UIR request state machine transition diagram



(DC_REQ_VALID deasserted).

The second example sequence illustrates using DC_ABORT when the UIR1 level registers are held by a hazard condition. Figure 5-7 shows the second UIR request example. The sequence

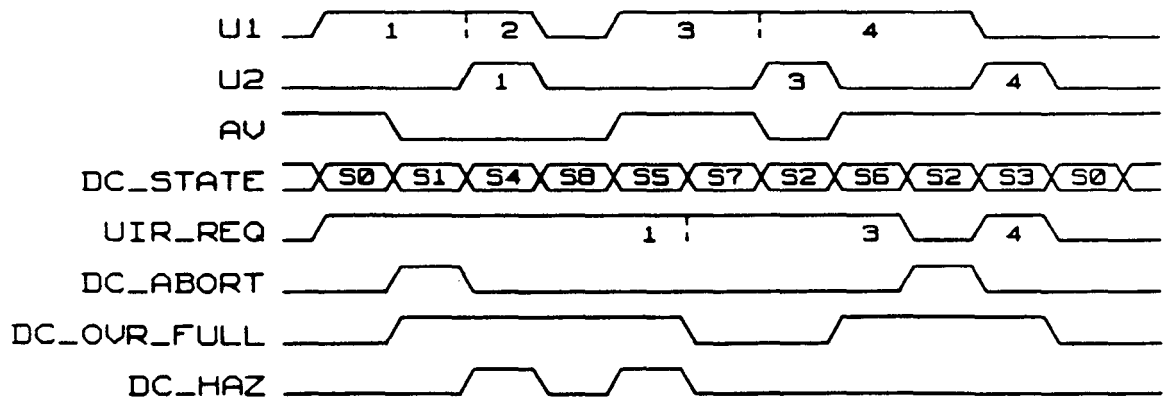
Figure 5-7 UIR request: Example two



starts with U1 being asserted. The UIR1 level pipe stage is being held so U2 does not get asserted until four cycles later. The state machine issues UIR_REQ and transitions to state S1 as in the previous example. This time since U2 is not asserted the request made last cycle is aborted (DC_ABORT) and another is issued (UIR_REQ) in hopes that U2 will be asserted on the next cycle. In the example U2 is still not asserted in state S2 resulting in another aborted request. The process of issuing a request (UIR_REQ) and aborting on the following cycle (DC_ABORT) ties up the data cache pipe. Other requests that happen in the background, instruction look ahead requests and data cache block prefetch requests, could be using the data cache pipe. In order to allow the background requests to use the data cache pipe, once a request reaches state S3 it no longer issues requests (UIR_REQ) but rather waits until the U2 signal is asserted. This causes a potential performance penalty in that the request could have been started one cycle earlier if the request/abort sequence was continued. The example finishes by issuing a request from S3 once U2 is asserted and then transitioning back to the idle state

The final example sequence shows what happens when the data cache pipe is unable to accept a request due to either the pipe being held or higher priority requests being selected. Figure 5-8 shows the final example sequence. The sequence starts with a valid request at the UIR1 level

Figure 5-8 UIR request: Example three



and the data cache pipe available, so the state machine transitions to state S1. At state S1, the

UIR2 stage of the pipe is not valid so the previous data cache pipe is aborted. Note that in state S1 UIR_REQ is asserted but since the data cache pipe is unavailable it will not win arbitration and will be ignored. At state S1 the data cache becomes unavailable so the state machine transitions to state S4. In state S4 the second level UIR2 stage becomes valid, but we were unable to start a request on the previous cycle. At state S4 the UIR1 level contains a valid memory request, so DC_HAZ signal is asserted to hold the UIR1 level registers until the request at the DC_OVR level has entered into the data cache pipe. In state S8 the request being held at UIR1 stage went invalid (due to an incorrect branch tag) so the state machine transitions to state S5. At state S5 the request being held in the DC_OVR level is issued and U1 is asserted implying that another request is ready to be made. At state S7 the request at the UIR1 level (which was held by state S5 on the previous cycle) is issued.

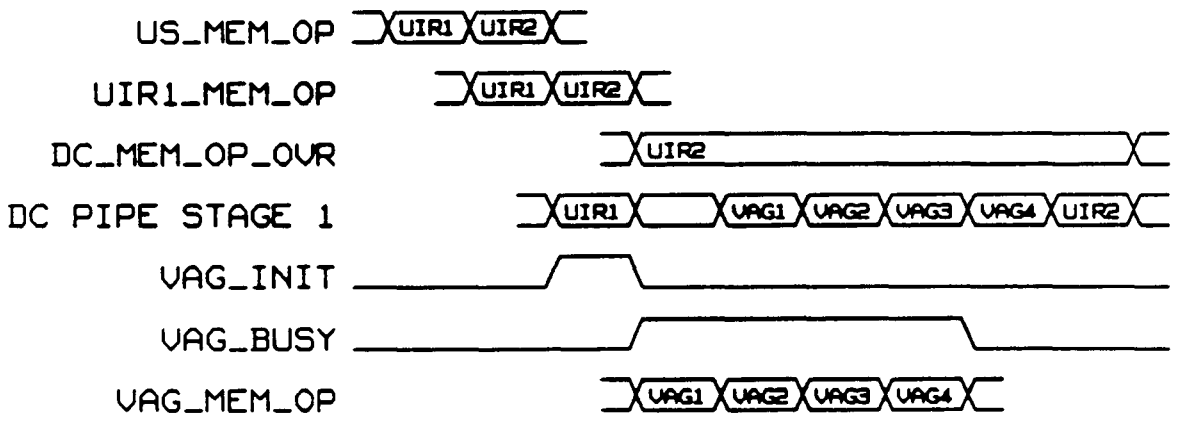
At this point we have caught up with the requests being made from the UIR interface. State S2 is entered with U2 asserted meaning that request three has been completed. At state S2, U1 is asserted but AV is not, so the state machine transitions to S6 to wait on either the cache becoming available or U2 being asserted. In state S6 the AV signal is asserted so a request is issued and the state machine transitions back to state S2. In S2 the previously made request is aborted due to the lack of U2 be asserted. Finally in state S3 U2 is asserted and a request is made for the forth UIR request, completing the sequence.

5.3.2 VAG requests

The vector address generator (VAG) is started for a vector load or store instruction from micro code with a memory operation which has the VAG start bit set. When the UIR request is at the first stage of the data cache pipe the vector address generator is initialized. On the next cycle the VAG control logic begins issuing VAG requests. Because VAG requests are the highest request source, the VAG requests will win arbitration. The cycle that the VAG is initialized is indicated by the assertion of the signal VAG_INIT. The VAG_INIT signal is used to inhibit a UIR request from entering the data cache pipe to force the proper order of instruction execution. After the VAG_INIT cycle the signal VAG_BUSY is issued informing the data cache pipe arbitration logic that VAG requests are valid.

Figure 5-9 illustrates the timing for the VAG start sequence. The sequence starts with the micro

Figure 5-9 VAG start sequence



code field US_MEM_OP. The micro code field is staged to become UIR1_MEM_OP on the next cycle. At the UIR1 stage the request wins arbitration to the data cache pipe. At the first stage of

the data cache pipe the signal VAG_INIT is asserted. The VAG_INIT signal prohibits the second UIR request from entering the data cache pipe. The VAG_INIT signal also starts the VAG control logic. The VAG_BUSY signal is asserted for four cycles (assuming four requests were needed) with four identical VAG_MEM_OP requests. Once the VAG goes idle again, the UIR requests can be accepted to the data cache pipe.

Some requests require the VAG logic to wait for data and addresses from the vector processor. When the required data or address is unavailable the VAG holds until the required information is transferred from the vector processor. The data cache pipe is issued VAG requests which are invalid while the VAG is being held. This has the effect of sending NOP requests until a valid VAG request can be made. No other requests sources (UIR, IP, or BPF) are able to make requests while the VAG is busy. When the data cache pipe is being held (due to backed up memory or a PTE cache micro interrupt) or when the second request for a vector unaligned long word is being processed, the VAG logic is held.

5.3.3 IP requests

Instruction processor requests are the simplest type of requests made. They are always long word requests which are long word aligned (same even and odd side address). The only variation is whether they are allowed to cause a fault if they are unable to complete.

IP requests are transferred from the NIP subsystem over the IXA interface. The address and NOFLT control bit is queued by the NDC subsystem. The address is queued by the NAG gate arrays, and the control bit is queued by the NDC gate array. A request is issued to the data cache pipe arbitration logic whenever the queue is not empty. IP requests have lower priority than UIR and VAG requests because IP requests are for look ahead data.

5.3.4 BPF requests

The block prefetch (BPF) mechanism is used to load a block of memory into the data cache. The mechanism is started from a UIR request with BPF enabled that misses the data cache. The size of a block is scan initialized and can be as large as 32 longwords. There are two registers which must be initialized when changing the block prefetch size. There is a six bit counter on the NDC gate array and a five bit mask register on the least significant NAG gate array. Table 5-2 shows the possible block sizes and appropriate initialization values.

Table 5-2 BPF initialization values

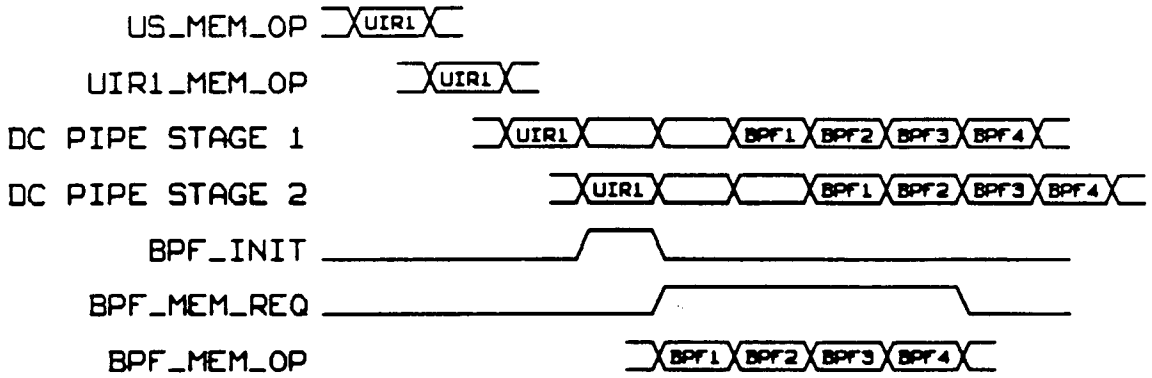
Block Size	R.BPF_MAX_CNT	R.BPF_MASK
No BPF	0	X
1 Long Word	1	0
2 LW	2	1
4 LW	4	3
8 LW	8	7
16 LW	0x10	0xF
32 LW	0x20	0x1F

The request type is a read which is destined for the data cache. The address is kept within the original block by allowing only the address bits within the block to change. The first address to be accessed is the original UIR request address plus eight. The remainder of the block is accessed by incrementing the previous address by eight. The last address to be accessed is the

original UIR address. On the final access only the portion of the long word which was not originally accessed is brought in. For each long word a request is issued to memory if the data cache does not have the data.

The block prefetch mechanism is started from the second stage of the data cache pipe. By the time the request reachs the second stage of the pipe information is available as to whether a data cache miss has occurred. The second stage cycle initializes the BPF mechanism and the first request is issued on the following cycle. Figure 5-10 shows the timing for block prefetch initialization.

Figure 5-10 Block Prefetch Timing



When the BPF mechanism is prefetching a block of memory and a UIR request misses the data cache, the BPF mechanism is restarted from the new UIR request address. The NAG gate arrays contain comparison logic to detect that the new UIR request is accessing memory from the same block that is being prefetched. When the comparison logic indicates that the new UIR request is in the same block being prefetched the BPF mechanism is not restarted.

5.3.5 Data cache update requests

A data cache update request is the process of writing data which was transferred from memory to the data cache. A data cache update operation can be performed simultaneously with any data cache read request (but not a data cache write request). An update request is initiated when a transfer of data from the NRC subsystem to the NDC subsystem using the UPD interface is made. The update data is queued in the NDP gate arrays, the update address in the NAG gate arrays, and the control in the NDC gate array. The NDC gate array monitors the type of requests being selected into the data cache pipe and allows an update request to proceed when a read request is being processed.

5.4 Logical Address Generation

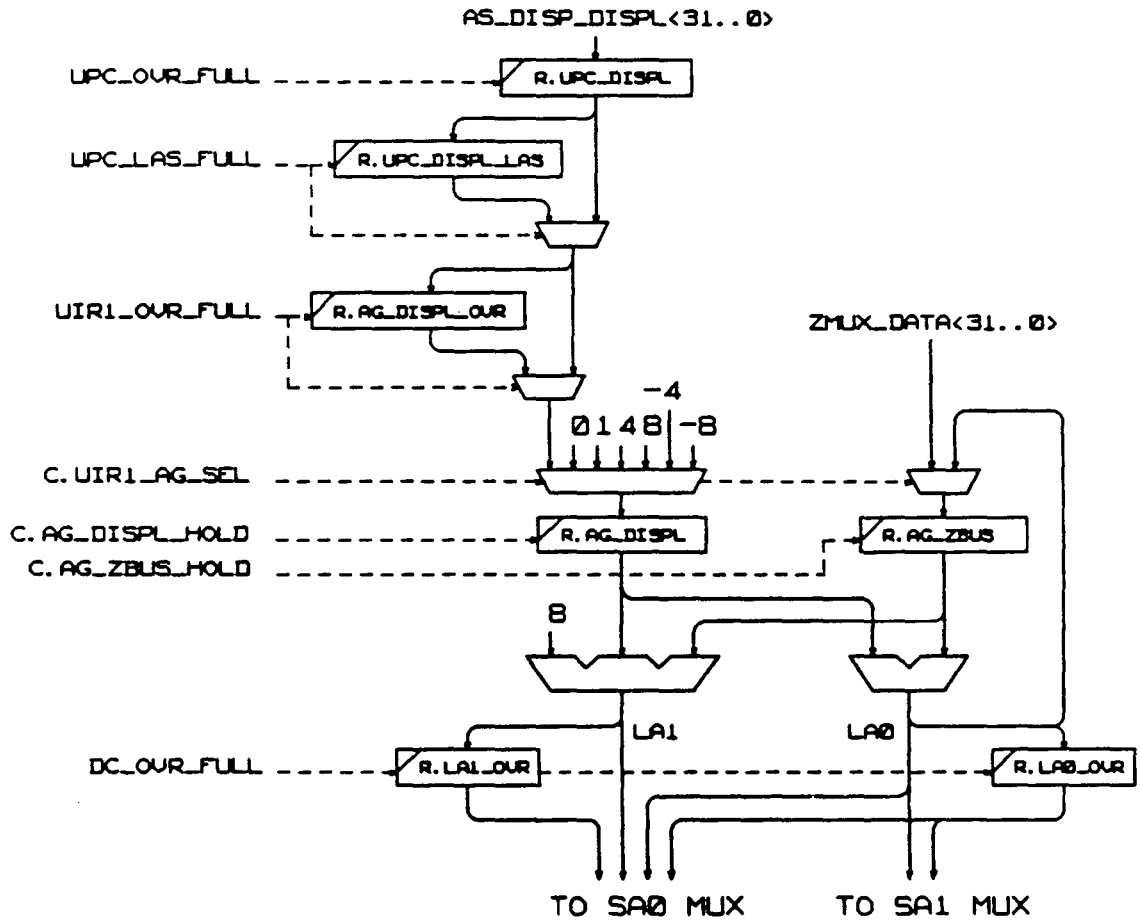
Each of the request sources described in the previous section provides information to generate a logical address. For NIP requests the logical address is provided, for other sources, UIR or VAG, the logical address must be generated with dedicated hardware. The following sections describe address generation for each of the request sources.

5.4.1 UIR interface address generation

The address generation for the UIR interface calculates the effective address for instructions. An

effective address (EFFA) is calculated by adding the displacement field of an instruction to a register file location accessed using the J register field of the instruction. Figure 5-11 shows the address generation paths of the NAG gate arrays for the UIR interface. The two NAG gate arrays are partitioned into upper and lower halfs. NAG0 contains bits <15..0> of the address paths and NAG1 contains bits <31..16>.

Figure 5-11 UIR interface address generation logic



The displacement field of an instruction is transferred from the NIP subsystem using AS_DISP interface directly to the NAG gate arrays. The field is staged through the NAS pipe stages to the R.AG_DISPL register. The J register field is staged and used by the NRFA gate arrays of the NAS subsystem to access the register file. The register file contents are transferred to the NAG gate arrays via the ZMUX bus. The R.AG_DISPL and R.AG_ZBUS registers are at the UIR1 register stage of the NAS subsystem pipe.

Address and parity for the displacement field, AS_DISP_DISPL and AS_DISP_DISPL_PAR, are loaded into register R.UPC_DISPL and the parity field is checked at the output of the register. Similarly address and parity for the register file location are checked at the output of register R.AG_ZBUS.

A signal from the NRFA gate arrays, AG_SRC_HAZ, is used to specify that the register file location being accessed by the J register field is not valid (an operation is being performed which will write to that register). The signal inhibits the request from starting and allows the register

R.AG_ZBUS to continue loading until valid data is received.

The mux select signal C.UIR1_AG_SEL is a staged micro code field. It is used to select the type of address generation which is to be performed. Table 5-3 shows the available selections.

Table 5-3 UIR interface address generation selection

C.UIR1_AG_SEL	Address Generation Selection
0	AG_SEL_AE - ZMUX address plus displacement
1	AG_SEL_LDZ - ZMUX address plus zero
2	AG_SEL_HOLD - C.LA0 plus zero
3	AG_SEL_LP4 - C.LA0 plus four
4	AG_SEL_LP8 - C.LA0 plus eight
5	AG_SEL_ZM4 - ZMUX address minus four
6	AG_SEL_ZM8 - ZMUX address minus eight
7	AG_SEL_LM4 - C.LA0 minus four
8	AG_SEL_LM8 - C.LA0 minus eight
9	AG_SEL_LP1 - C.LA0 plus one

The selection of the address generation type is made one cycle before the actual memory operation is specified. This requires that the last micro word of every instruction must specify AG_SEL_AE. This will setup the hardware to generate the effective address for all memory instructions. Some instructions must push or pop data onto a stack. The last plus/minus offset selections are available to conveniently generate stack and consecutive communication register address sequences.

Two addresses are always produced, LA0 and LA1. LA1 is LA0 plus eight. LA1 is required when a reference must access different even and odd side cache locations. As an example, the data cache uses logical address bits <13..3> to index the rams. If a word is accessed at location 0x80001006, then the bytes which are required range from 0x80001006 to 0x80001009. This corresponds to cache ram addresses 0x200 and 0x201. The LA0 address is used to access the odd side cache rams, and LA1 is used to access the even side rams. If the original address was 0x800010002, then both even and odd side rams would use address LA0 to access the rams. As shown in Figure 5-11, LA0 or LA1 can be selected as SA0 (the even side ram address). LA0 is always selected for the SA1 address.

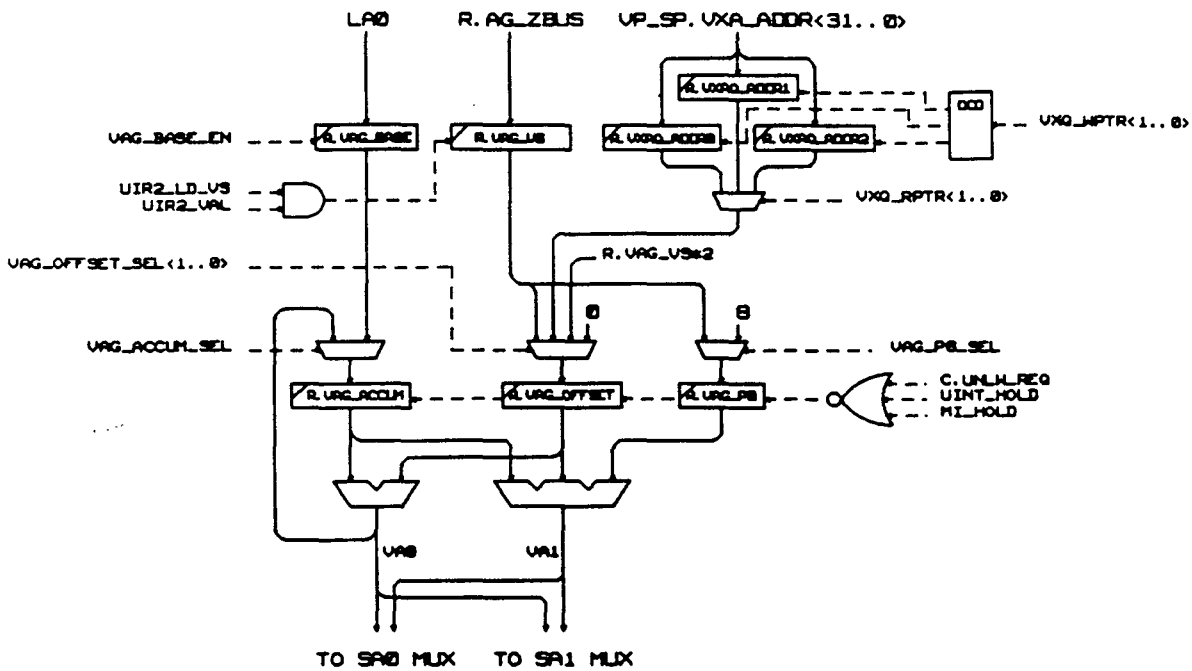
Overrun registers are used to hold LA0 and LA1 as shown in Figure 5-11. These registers correspond to the DC_OVR register stage of Figure 5-3. When the data cache pipe is being held or has higher priority requests the UIR addresses are generated and held in the R.LA0_OVR and R.LA1_OVR registers until they can be used.

5.4.2 Vector address generation

Figure 5-12 shows the logic used for vector address generation. The vector address generator (VAG) is started from a UIR interface request with the VAG start bit set. The address used for the vector base address is either register file location A5 for vector indexed loads and stores or the effective address of the vector instruction. The base address for the vector memory operation, LA0, is loaded into R.VAG_BASE every cycle the VAG is idle. The VAG control logic is started at the first stage of the data cache pipe. The signal VAG_INIT indicates that a UIR request is at the first stage of the pipe and is used to hold the contents of register R.VAG_BASE. The VAG_INIT

signal is used to force the selections of the VAG_ACCUM, VAG_OFFSET and VAG_P8 muxes to the first address of the vector instruction. During the cycle that VAG_INIT is active the base register value is transferred from R.VAG_BASE to R.VAG_ACCUM.

Figure 5-12 Vector Address Generator Logic



Other sources of information used to generate vector addresses are the vector stride register and vector address queue. The vector stride register, R.VAG_VS, is an architecturally defined register which specifies the distance in bytes from one element of a vector to the next. The register is loaded from R.AG_ZBUS when a load vector stride instruction is executed. The vector address queue is used to hold addresses transferred from the vector processor to the scalar processor.

Table 5-4 shows the selections for the VAG muxes for the different types of vector instructions.

Table 5-4 VAG mux selection

INDEX	2X	MASK	SIZE	INIT	EVEN	ACCUM	OFFSET	P8	INST
0	0	0	X	1	X	Base	0	8	ld.w
0	0	0	X	0	X	VA0	VS	8	ld.w
0	0	1	Not LW	X	1	Base	VXAQ	8	ld.w.t
0	0	1	Not LW	X	0	VA0	VS	8	ld.w.t
0	0	1	LW	X	X	Base	VXAQ	8	ld.l.t
0	1	0	X	1	X	Base	0	VS	ld.w
0	1	0	X	0	X	VA0	2xVS	VS	ld.w
0	1	1	X	X	X	Base	0	VS	ld.w.t
1	X	X	X	X	X	Base	VXAQ	8	ldvi.w.t

The columns of the table represents inputs to the selection logic, the mux selects, and an example instruction which would use the selections. The INDEX column specifies whether the INDEX bit

of the memory operation was set when the VAG was started, DC_MEM_OP<11>. Only vector of indices load and stores set this bit. The 2X column indicates whether the operation will proceed by making requests for two vector elements per cycle. Only vector loads and stores which are byte, halfword, or word with proper vector stride can proceed at rate 2X. The column labeled MASK represents the mask bit of the original memory operation field, DC_MEM_OP<10>. The SIZE field is the size of the memory operation, but only whether the value is a longword is decoded. The INIT column represents the generation of the first memory address for the vector. The EVEN column represents whether the vector element being processed is even (0, 2, 4, ...) or odd.

The mux selects are shown in columns ACCUM, OFFSET, and P8. For decodes of the selections refer to the signal definition appendix for the signals VAG_ACCUM_SEL, VAG_OFFSET_SEL, and VAG_P8_SEL. The final column is an example instruction which would use the selections.

The VA1 address is either VA0 plus eight, or the second request address for 2X rate operations. Rate 2x operations are allowed to proceed when the even elements access one side on memory and the odd elements access the other side of memory. Both VA0 and VA1 must be able to be selected to SA0 or SA1.

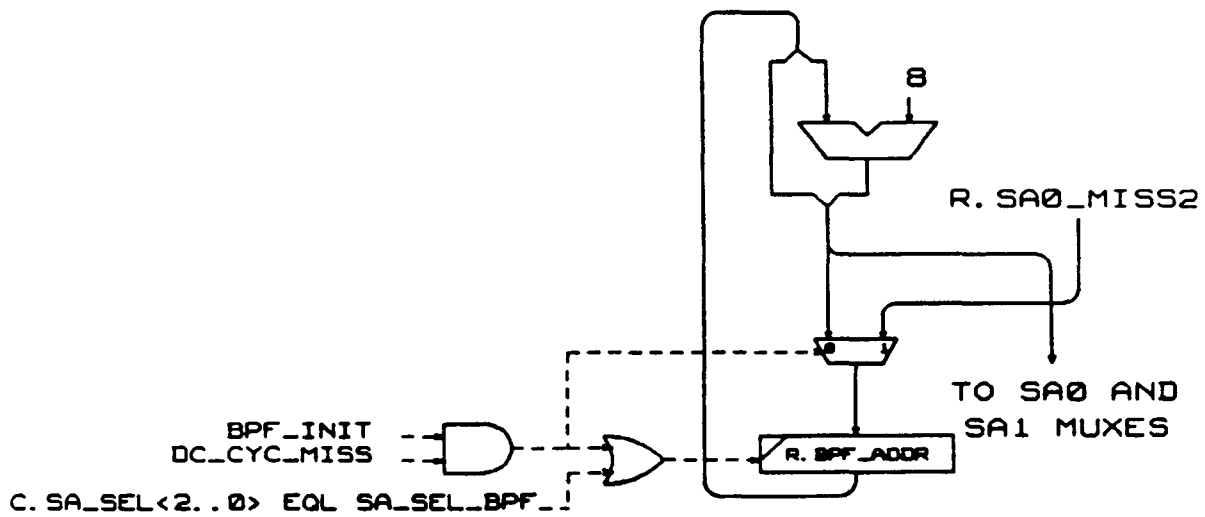
5.4.3 Instruction processor address queue

The instruction processor transfers logical addresses to the NDC subsystem using the IXA interface. The NAG gate arrays of the NDC subsystem queue the addresses until the IP request becomes the highest priority request. The logical address is read from the queue and selected unmodified to SA0 and SA1. The logical address can be used for both even and odd sides because IP request are always long word aligned.

5.4.4 Block Prefetch address generation

The block prefetch address generation logic is shown in Figure 5-13. The block prefetch

Figure 5-13 Block Prefetch Address Generation Logic



mechanism is started from the second stage of the data cache pipe. The signal BPF_INIT is used to select the second stage data cache pipe address, R.SA0_MISS2, to be loaded into the BPF address register R.BPF_ADDR. The address is incremented by eight and used as the logical

address for the first BPF request. Logic exists at the output of the adder which allows only the memory block address bits to change. The scan initialized register R.BPF_MASK on the NAG gate arrays specifies how large the block is. BPF memory requests are always long word aligned allowing the same logical address to be used for both the even and odd side of memory. A path exists from the register R.BPF_ADDR back to itself which is used whenever a block prefetch request is ready but is not selected to enter the data cache pipe.

5.4.5 Unaligned Long Word address generation

An unaligned long word occurs anytime a long word request is made where bits <1..0> of the logical address are not zero. As an example, a long word request to address 0x800010001 would result in accessing bytes from 0x800010001 to 0x800010008. The cache is addressed using bits <13..3> of the logical address, so cache locations 0x200 on the even side, 0x200 on the odd side, and 0x201 again on the even side must be accessed. The even side of the data cache must be accessed twice, using different ram addresses.

Unaligned long words are accessed in two pieces. Each piece is four bytes (a word). When an unaligned long word is detected, at the first stage data cache pipe, the request for a long word is changed to a request for a word and a request for the second word is entered into the data cache pipe the following cycle. The second piece of an unaligned long word request has higher priority than any other request source and will always follow immediately after the first piece request.

When the unaligned long word (UNLW) request is from the UIR interface (destined for the register file), then each piece of the request is written separately to the register file on the NAS subsystem. This allows for one piece of the UNLW to hit the data cache and be written to the register file immediately and the other missing the data cache and being written after the data is returned from the cross bar.

All requests from the VAG read and write memory. An UNLW read request from the VAG is processed in two pieces but is reassembled by the NRC subsystem before being transferred to the vector processor board.

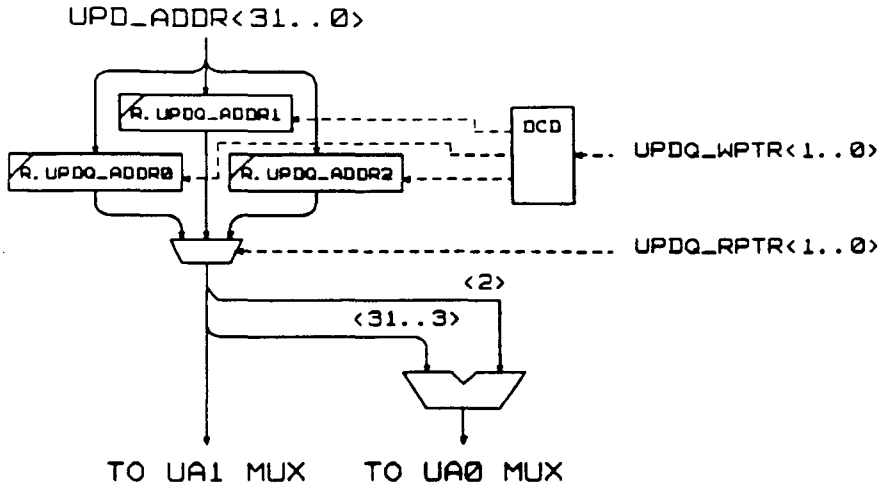
The logical address for the first piece of the UNLW request is generated from its normal source, either the UIR interface or VAG logic. The address for the second piece is generated by adding eight to the odd side first stage data cache pipe logical address (R.SA0_MISS1). The addresses selected for SA0 and SA1 are from either the UNLW adder, R.SA0_MISS1, or R.SA1_MISS1 depending on the alignment of the address for the request.

5.4.6 Data Cache update address generation

Data cache update requests come from the NRC subsystem using the UPD interface. Update requests are return data from memory which are to be written in to the data cache. The UPD interface address is queued in the UPDQ on the NAG gate arrays. When an update request is able to be processed, the address is read from the queue and used to access the data cache rams. Bits <13..3> of the update address is used to access the rams.

Figure 5-14 shows the logic used to generate the update addresses. Updates are always word aligned (but not necessarily long word aligned). If bit <2> of the queued update address is set, then the even side update address must be one larger than the odd side update address. An adder is used to conditionally add eight for the even side update address.

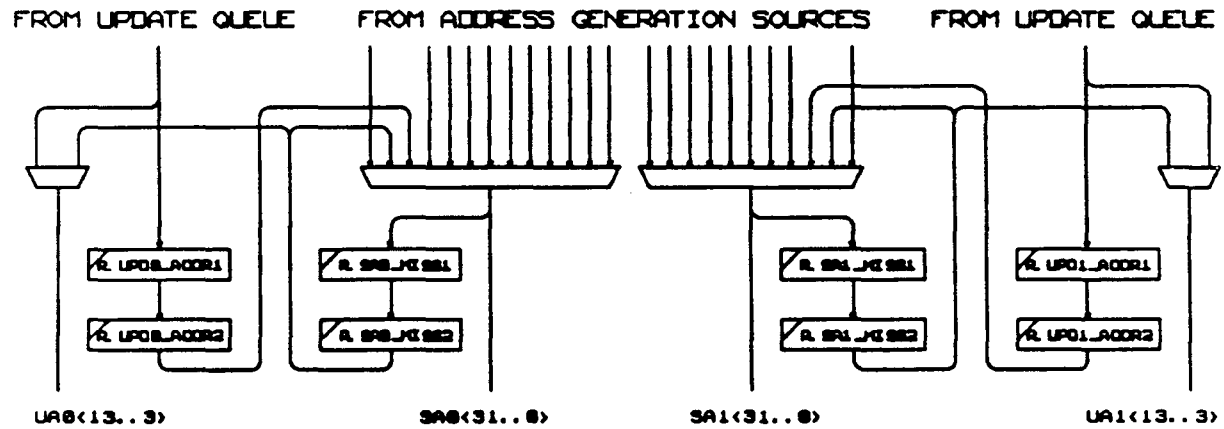
Figure 5-14 Update Address Generation Logic



5.4.7 Cache address selection muxes

Figure 5-15 shows the address selection muxes and write staging registers. Each of the four cache ram address sources, UA0, UA1, SA0, and SA1, are staged for two cycles and provided as inputs to the selection muxes. The addresses generated by the address generation sources are first used for cache ram reads, then after being staged two cycles are potentially used for cache ram writes. The selection muxes use a 2x phase signal as one of the selection control signals.

Figure 5-15 Address selection muxes



During the first half clock of each cycle a write address is selected. Write addresses are selected right out of a register for timing reasons. There are two write addresses available for SA0 and SA1, one for UIR or VAG request writes (R.SAx_MISS2) and one for update request writes (R.UPDx_ADDR2). The UA0 and UA1 addresses are used to address the data cache update tag rams and are only written from UIR or VAG requests, not update requests.

A read address is selected the second half clock of each cycle. The read address sources are directly from the address generation logic for UIR, VAG, IP and BPF requests. Figure 5-17 (located in the Data Cache Operations section) shows the 2x address selection and staging.

The muxes are also used to select the faulting address for PTE misses during a micro interrupt.

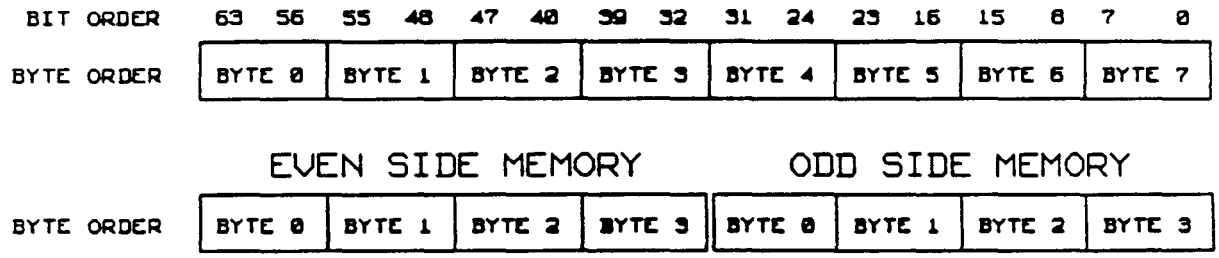
The faulting address is selected as both even and odd side addresses. The address is used to supply the NAS subsystem with the faulting address and when writing the PTE cache rams with a new page table entry.

5.5 Zone Information

Zone information is used to generate byte write enables for the data cache and for memory. The logical address, size of data, long word alignment, and rate 2x information is used to generate individual byte write enables. To understand zone data byte ordering it is necessary to describe the ordering of bytes in main memory.

Figure 5-16 shows bit and byte order for the memory system. When a load or store is performed

Figure 5-16 Memory Longword Structure



the logical address for the memory reference is the most significant byte. Therefore a longword reference to address 0x80001000 would reference bytes 0x80001000 through 0x80001007. The memory reference would be longword aligned with the most significant word coming from the even side memory and the least significant word from the odd side.

Parity and zone bits are organized by byte order (bit zero of zone information corresponds to byte zero of data). To associate data bytes to their zone bits the order must be reversed, since bits are numbered from right-to-left and bytes are ordered from left-to-right.

There are four even side and four odd side zone bits generated. The zone information is generated by taking the size of the data in bytes and rotating to align it based on the logical address for the most significant byte.. Table 5-5 shows the generated zone bits.

Table 5-5 Zone Information

Size	Address	Rate 2x	Even Zone	Odd Zone
Byte	0	0	1	0
Byte	1	0	2	0
Byte	2	0	4	0
Byte	3	0	8	0
Byte	4	0	0	1
Byte	5	0	0	2
Byte	6	0	0	4
Byte	7	0	0	8
Byte	0	1	1	1
Byte	1	1	2	2
Byte	2	1	4	4
Byte	3	1	8	8

Byte	4	1	1	1
Byte	5	1	2	2
Byte	6	1	4	4
Byte	7	1	8	8

Size	Address	Rate 2x	Even Zone	Odd Zone
Halfword	0	0	3	0
Halfword	1	0	6	0
Halfword	2	0	0xC	0
Halfword	3	0	8	1
Halfword	4	0	0	3
Halfword	5	0	0	6
Halfword	6	0	0	0xC
Halfword	7	0	1	8
Halfword	0	1	3	3
Halfword	1	1	6	6
Halfword	2	1	0xC	0xC
Halfword	4	1	3	3
Halfword	5	1	6	6
Halfword	6	1	0xC	0xC

Size	Address	Rate 2x	Even Zone	Odd Zone
Word	0	0	0xF	0
Word	1	0	0xE	1
Word	2	0	0xC	3
Word	3	0	8	7
Word	4	0	0	0xF
Word	5	0	1	0xE
Word	6	0	3	0xC
Word	7	0	7	8
Word	0	1	0xF	0xF
Word	4	1	0xF	0xF

Size	Address	Aligned	Even Zone	Odd Zone
Longword	0	Yes	0xF	0xF
Longword	1	No, Request one	0xE	0xF
Longword	1	No, Request two	0x1	0xF
Longword	2	No, Request one	0xC	0xF
Longword	2	No, Request two	0x3	0xF
Longword	3	No, Request one	0x8	0xF
Longword	3	No, Request two	0x7	0xF
Longword	4	Yes	0xF	0xF
Longword	5	No, Request one	0xF	0xE
Longword	5	No, Request two	0xF	0x1
Longword	6	No, Request one	0xF	0xC
Longword	6	No, Request two	0xF	0x3
Longword	7	No, Request one	0xF	0x8
Longword	7	No, Request two	0xF	0x7

The halfword and word rate 2x operations are only allowed when each of the two requests are contained entirely in one side of memory. This eliminates halfword logical address cases 3 and

7. Similarly, word rate 2x cases 1, 2, 3, 5, 6, 7 are all prohibited from happening.

5.6 Data Cache

This section describes the data cache. First the structure of the data cache is given, followed by the operations performed by the data cache.

5.6.1 Data cache rams

The data cache within the NDC subsystem is implemented with 2K deep by 9 bit wide self timed static rams for data and tag information, and 2K deep by 2 bit wide self timed, static, purgeable rams for validity. The rams are cycled twice per system cycle, the first access is always a read and the second is a write. The rams can be read and written from the scan chain for initialization and visibility.

An even and an odd side data cache exists to allow access using different addresses. Different addresses will occur when an access is made that is not contained in a long word aligned long word or when a vector operation is being performed at rate 2X. For rate 2X operations the two addresses are different by the value in the vector stride register.

The data cache contains four data rams (32 bits wide with four bits of parity), four tag rams (32 bits wide with four bits of parity), one update tag ram (8 bits wide with one bit of parity), one thread validity ram (one bit wide with one bit of parity), and one non-thread validity ram (one bit wide with one bit of parity). The tag rams contain many tag fields: a twenty bit logical page address (bits <31..12>), a five bit communication index register (CIR) tag, and four bits of byte zone data.

The zone bits are used with the valid bits to specify which bytes of data in the cache are valid. A bit in each PTE cache entry specifies whether a memory page is thread or non-thread memory. This bit is used to select either the thread or non-thread validity ram information. The selected validity bit is logically and'ed with the zone bits to specify which of the data bytes contain valid data.

The update tag ram contains a five bit update tag with parity, the other three bits of the ram are unused. The update tag ram is separate from the other tag rams because it requires different addresses for the various operations performed on the data cache.

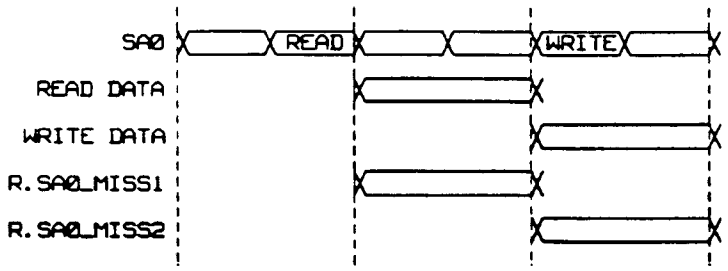
The update tag is used to keep the data cache consistent. A typical sequence of operations which demonstrates the need for the update tag starts with a read from the data cache which misses (is not contained in the cache). The read request is forwarded to memory and about twenty cycles later data will return. Before data returns from the read, a write to the same logical address and therefore to the same cache location occurs. When the read return data arrives it can not be written into the data cache because the data cache location contains data which is more current. The update tag ram is written each read and write operation. The update tag counter value is incremented for each read which is forwarded to memory after the update ram is written. In the example sequence assume the tag value written for the read request was a three, then since it missed the cache and was forwarded to memory the update tag counter would have been incremented to four. The tag value of three would have been sent to the NRC subsystem with the other request destination control information. When the return data arrived and was transferred to the NDC subsystem using the UPD interface the update tag would also be transferred. The returned tag value of three would be compared to the update ram tag value of four and the returned data would not have been written to the data cache rams.

5.6.2 Data cache operations

This section describes how the primitive read/write operations of each ram are used to perform the data cache operations.

Figure 5-17 shows the data cache address staging. Read addresses are presented to the rams on the second half of each system clock cycle. The address is registered internal to the rams and

Figure 5-17 Data Cache Address Staging



used to access the internal ram array on the following cycle. The output of the ram array is latched and held internal to the ram until the next cycle when the latch is opened for another read access. The read address is staged in R.SA0_MISS1 on the NAG gate array and then staged again by R.SA0_MISS2. These two staging registers correspond to the two data cache pipe stages. The address in R.SA0_MISS2 is selected for the SA0 address during the first half of a cycle. The data which is written into the ram is presented at the same time as the address and must meet setup at half clock to the rams.

Figure 5-18 illustrates the timing for a read operation. The data, tag and validity rams are read

Figure 5-18 Data Cache Read Operation

	CYCLE 1		CYCLE 2	
	PHASE 1	PHASE 2	PHASE 1	PHASE 2
DATA	READ			
TAG	READ			
VALID	READ			
UPDATE				WRITE

on phase one of cycle one. The update tag write occurs on the second phase of the second cycle. The tag and validity information are processed at the end of cycle one before being registered to produce the cache hit or miss signals.

Data cache write operations are always performed as a read-modify-write operation. The 32 bit data rams are always written together (no byte write capability). Figure 5-19 illustrates the timing for a data cache write operation. The data in the cache is read and merged with the new data to be written to the rams before being registered at the end of the cycle. If the previous cache location is for the same address as the new write then the tag zone bits are logically or'ed with the new zone bits. Otherwise the old content of the cache location is being discarded, requiring the old zone bits to be cleared before the new zone bits are logically or'ed in. The current request's page address and CIR are always written to the tag rams with the modified zone bits for a write cycle. Both validity bits, thread and non-thread, are set to a one.

Figure 5-19 Data Cache Write Operation

	CYCLE 1		CYCLE 2	
	PHASE 1	PHASE 2	PHASE 1	PHASE 2
DATA	READ			WRITE
TAG	READ			WRITE
VALID	READ			WRITE
UPDATE				WRITE

The final data cache operation is a data cache update. The update operation requires reading the

Figure 5-20 Data Cache Update Operation

	CYCLE 1		CYCLE 2	
	PHASE 1	PHASE 2	PHASE 1	PHASE 2
DATA				WRITE
TAG				WRITE
VALID				WRITE
UPDATE	READ			

update tag ram on the first phase of cycle one. The update tag is compared with the tag returned with the NRC subsystem transfer to determine if the cache write operation should be completed. If the tags match then on the second phase of the second cycle the data, tag and validity rams are written. Update writes always write all four bytes, resulting in all four zone bits being written to a one.

5.6.3 Data cache data paths

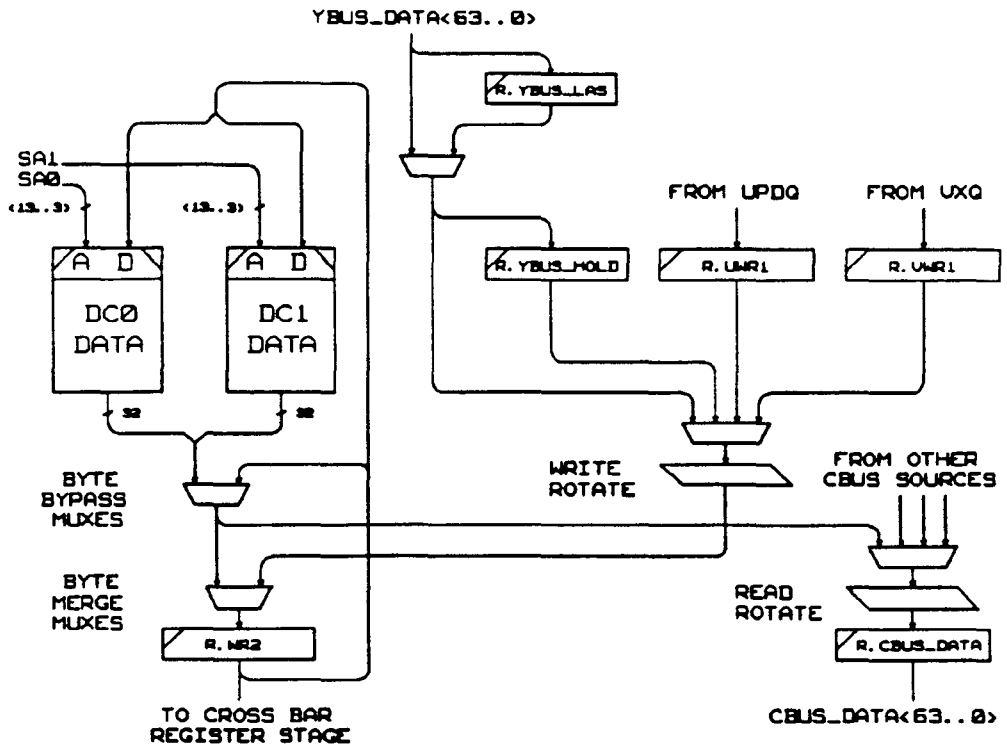
All data paths for the NDC subsystem are contained on the NDP gate arrays. Figure 5-21 shows the data paths for the NDC subsystem. The figure shows cache read data being selected by the byte bypass muxes followed by byte merge muxes and finally being registered in R.WR2. For write operations where read-modify-write occurs the merge muxes merge in the new write data to the cache's existing data based on zone information.

The byte bypass muxes are used when a read or write operation immediately follows a write operation to the same logical address. In this situation the data for the first write operation is located in R.WR2 when the ram read for the second operation is being performed. The ram has not been written with the write data when the read occurs. The second operation obtains the correct data by using the byte bypass muxes to select R.WR2.

UIR write requests select the YBUS from either the R.YBUS_HOLD register, R.YBUS_LAS register or directly from the YBUS_DATA bus. Update write requests obtain data from R.UWR1 register and VAG requests receive data from the R.VWR1 register. The data to be written must be aligned using the write rotator before being merged with the byte merge muxes.

5.7 PTE Cache

Figure 5-21 Data Cache Data Paths



The PTE cache contains the last level entries for the memory based page translation tables. The cache is accessed by bits <22..12> of the logical address (the least significant eleven bits of a page address). The most significant bit of the ram address is bit <22> logical exclusive or'ed with bit <31> (SA0_XOR and SA1_XOR). The physical page address read from the PTE cache is used to replace the logical page address of the requested address to generate the memory address.

There are two PTE caches, one for the even side and one for the odd side. The contents of both caches are always the same, both caches are always written at the same time.

5.7.1 PTE cache rams

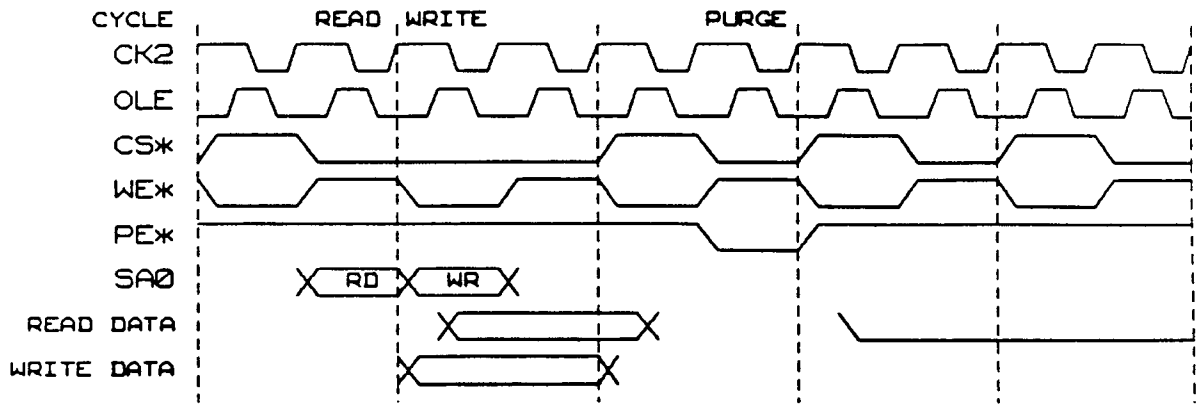
The PTE cache is implemented with 2K deep by 9 bit wide self timed static rams for data and tag information, and 2K deep by 2 bit wide self timed, static, purgeable rams for validity, referenced and modified bits.

Four rams are used for the PTE cache data rams. The rams store page table entries read from memory. The entries contain a physical page address in bits <31..12> and access and violation mode bits in <11..0>. Bits <11,9,8,6,5> are forced to be zero in the rams to reduce the pin count required to check parity on the data read from the rams. These five bits have no function for hardware. Bit seven is used to specify whether the PTE is from a second level page table or a thread level page table. The micro code specifies where the entry came from. The PTE cache data rams are written from the YBUS. Exclusive or gates external to the gate arrays modifies the YBUS parity to yield correct parity with the five bits zero'ed and bit <7> coming separately from micro code. The rams are clocked once per system clock cycle, being either read or written. Normally the rams are read, only when a PTE miss occurs for a memory request are the rams written.

Two rams are used for the PTE cache tag tams. The tag rams contain logical page address bits <31..23> and the five bit communication index register (CIR). The rams are accessed with the same address and control as the PTE cache data rams.

There are two sets of validity rams for each PTE cache, thread and non-thread. The rams are able to be purged (cleared to zero) when the encached page table entries are to be purged. Bit <7> of the PTE cache data ram is used to specify which of the two validity rams is to be used. The rams are written with a one when writing PTE data into the rams and written with a zero when a purge PTE entry operation occurs. Figure 5-22 shows the timing for the validity rams. Note that

Figure 5-22 PTE cache validity ram timing



the read data is latched internally based on the OLE input strobe signal. Also the read data is forced to be zero for the four cycles a purge operation is being performed.

The final rams to be described are the reference and modified bit rams. These rams implement a cache of the referenced and modified bits stored in memory. The referenced bit is used to tell the operating system that a page is being accessed and should not be swapped out of main memory. The modified bit is used to specify whether the page in memory has been written. These rams are cycled once per cycle similar to the data and tag rams.

5.7.2 PTE cache misses and access violations

Each time a logical to physical address translation is performed the PTE cache is accessed to supply the physical translation information. Each encached page table entry contains the physical page address and access mode bits. The mode bits are shown in Table 5-6.

Table 5-6 PTE access bits

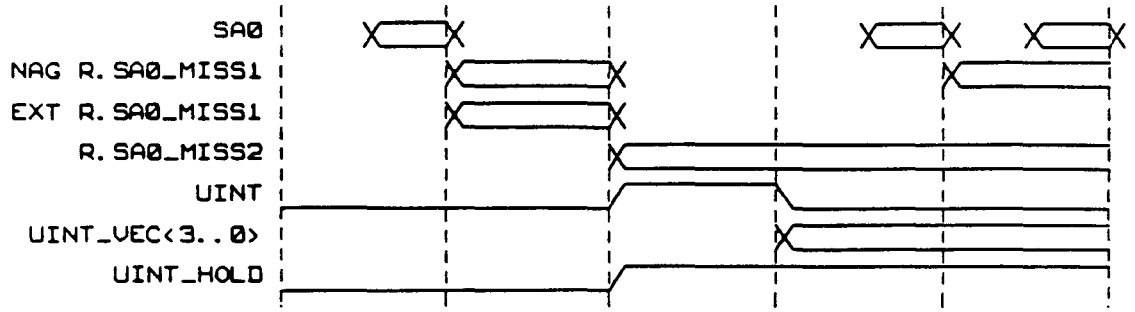
Bit<10> - Cache Load Bypass (CLB), disables read from accessing the data cache
Bit<7> - Thread Level, selects thread/non-thread validity rams
Bit<4> - Page table entry valid bit
Bit<3> - Read access permission bit
Bit<2> - Write access permission bit
Bit<1> - Execute access permission bit
Bit<0> - Resident bit

PTE misses occur when a PTE is read and the appropriate validity ram is not set. An access violation occurs when a PTE is read and the appropriate validity ram bit is set, but the type of access being performed is not allowed. The timing for a PTE miss and an access violation is

identical.

Figure 5-23 shows the timing for a miss or violation. The PTE miss/access violation sequence

Figure 5-23 PTE miss/violation timing



starts with the read address, SA0 or SA1, on the second half of a system cycle. The address accesses the PTE rams and is also staged in R.SA0_MISS1. There are two versions of this register, one is contained in the NAG gate arrays, the other is in discrete parts. The version in the NAG gate arrays is used to stage the address for writes and for request retries. The external register version is used by the NPA gate arrays to generate the physical address, and to supply the NAS subsystem with the logical address of PTE misses.

Once the PTE rams are accessed, the PTE tag is checked to see if it matches the logical page address and the validity rams are checked for a valid entry. At the same time the PTE data is checked against the type of access being made. All of these checks are performed before the PTE ram data is registered by R.SA0_MISS2. If a PTE miss or access violation exists, then UINT is asserted for one cycle informing the NAS subsystem. Also the signal UINT_HOLD is asserted and remains asserted until the problem is corrected. The NAS subsystem requires two pieces of information to process the micro interrupt initiated by UINT. The two pieces are the micro interrupt vector and the logical address of the request which caused the micro interrupt.

The logical address has been staged to R.SA0_MISS2 and is held by UINT_HOLD during the micro interrupt sequence. The NAG gate arrays register UINT_HOLD and use the registered value to select R.SA0_MISS2 (the held logical address) to both SA0 and SA1. Both PTE caches are always written with the same data, so the logical address is presented to both even and odd sides. The external R.SA0_MISS1/R.SA1_MISS1 registers are not held by UINT_HOLD (only memory interface hold, MI_HOLD, and HALT) and reloads the value of SA0/SA1 each cycle. The register R.SA1_MISS1 is used by the NAS subsystem as the logical address of the problem request.

The micro interrupt vector, UINT_VEC<3..0>, specifies the type of problem which was encountered. Table 5-7 lists the type of micro interrupts.

Table 5-7 Micro Interrupt Vector

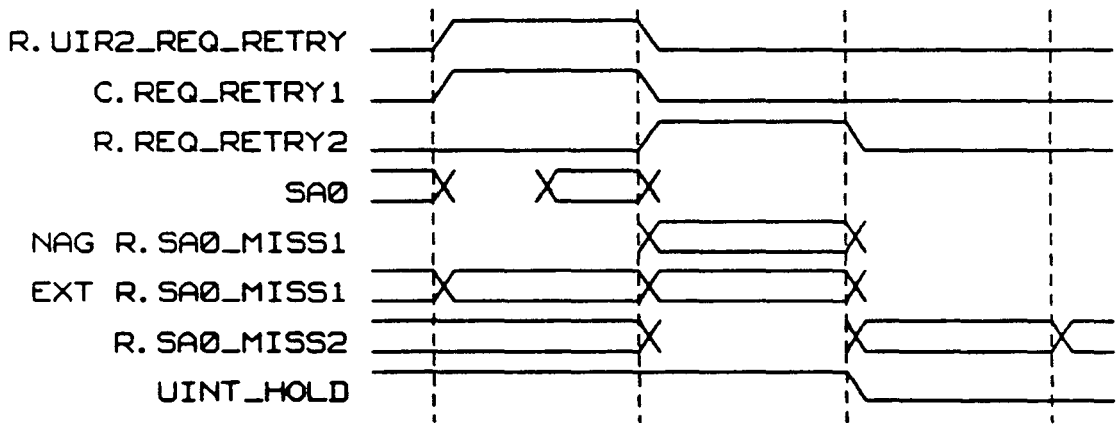
0	Inward Memory Access
1	PTE Miss
2	Invalid Thread Level PTE
3	Invalid Second Level PTE
4	Read Access Violation
5	Write Access Violation

6	Execute Access Violation
7	Non-Resident PTE Data
8	Inward Communication Access
9	IP Request Look Ahead
0xA	Invalid SDR
0xB	Invalid First Level PTE
0xC	Non-Resident Second Level PTE
0xD	Invalid Second Level PTE (from memory)
0xE	Non-Resident Thread Level PTE
0xF	Forced Fault

The micro interrupt types are listed in order of priority with zero being highest. Most of the above types are described in the Convex Architecture Reference Manual and will not be described here. The IP Request Look Ahead is a special case and will be described in the following section.

The micro code for a PTE miss micro interrupt accesses the PTE tables in main memory to read the needed page table entry. If the NAS subsystem is able to obtain the needed PTE then the PTE caches are written with the new entry and the request is retried. Figure 5-24 shows the timing for retrying a request. A retry sequence is initiated by micro code asserting US_REQ_RETRY.

Figure 5-24 Request Retry Timing



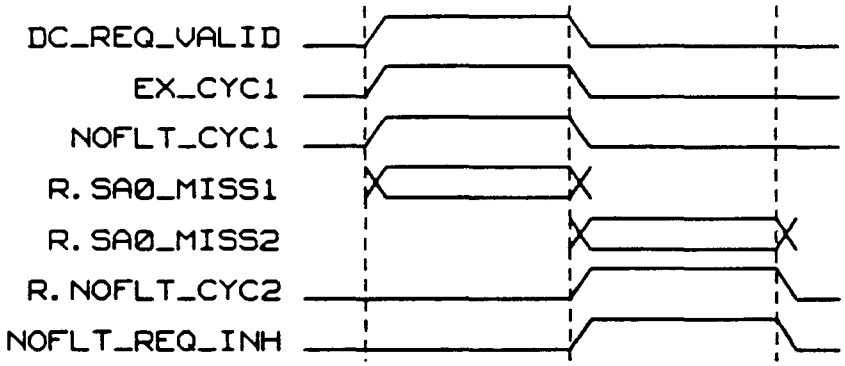
The micro code signal is staged to the second stage UIR level as UIR2_REQ_RETRY. The signal is combined with UIR2_VAL to produce C.REQ_RETRY1. The C.REQ_RETRY1 signal forces the data cache pipe mux to select the request to be retried (second stage register, R.SA0_MISS1/ R.SA1_MISS1) and allows the data cache pipe control and address registers to clock. The signal R.REQ_RETRY2 is a staged version of C.REQ_RETRY1 and allows the data cache registers to clock once again. During the R.REQ_RETRY2 stage the data cache and PTE cache rams are being read (with the newly written PTE data) and the PTE miss/access violations are rechecked. Provided that the checks are acceptable the UINT_HOLD signal is deasserted allowing the entire data cache to proceed. If a problem still exists, the signal UINT is asserted the cycle after R.REQ_RETRY2 and UINT_HOLD is not deasserted. At this point the micro interrupt sequence is repeated.

5.7.3 IP Look Ahead Non-Faulting Requests

The IP look ahead requests come in two varieties, faulting and non-faulting. Faulting requests are issued when the IP requires the memory location's contents to issue the next instruction. Faulting requests are required to be performed and an access violation or a PTE miss resulting in a context swap must be processed. Faulting requests are treated like requests from the other sources.

The other type of request is a non-faulting type look ahead request which is being issued in advance of actually knowing if it is needed. These requests should not cause an access violation or a context swap from a PTE miss. There are two sequences which are affected when a non-faulting request is made and a problem occurs. The first is if an access violation occurs. Requests with access violations are dropped, a UINT is not generated nor is a request issued to memory. Figure 5-25 shows the timing for a non-faulting request which is dropped. The request starts

Figure 5-25 Non-Faulting Request Timing



normally with the first pipe stage signals DC_REQ_VALID, EX_CYC1, NOFLT_CYC1, and R.SA0_MISS1. During the first pipe stage the PTE rams are being read and violation checks are performed. On the second pipe stage the logical address for the request is at R.SA0_MISS2, and NOFLT_CYC1 is staged to R.NOFLT_CYC2 internal to the NPA gate arrays. Assuming an access violation was detected on the previous cycle the signal NOFLT_REQ_INH is asserted. This signal is used internal to the NPA gate arrays inhibiting the assertion of UINT and UINT_HOLD and also is used by the NDC gate array to inhibit a request to main memory.

The second sequence which is modified by a non-faulting request is when a PTE miss occurs. The PTE miss proceeds normally by asserting UINT and UINT_HOLD with a UINT_VEC of PTE miss. If at any time while the NAS subsystem is making main memory PTE table requests an access violation occurs, the nested micro interrupt vector type will be type nine, IP Look Ahead Request. In this case the micro code retries the request without correcting the problem. A non-faulting IP request is dropped by asserting NOFLT_REQ_INH (as in the previous sequence) when a request retry is issued and a PTE miss or access violation still exists.

5.7.4 PTE cache referenced and modified bits

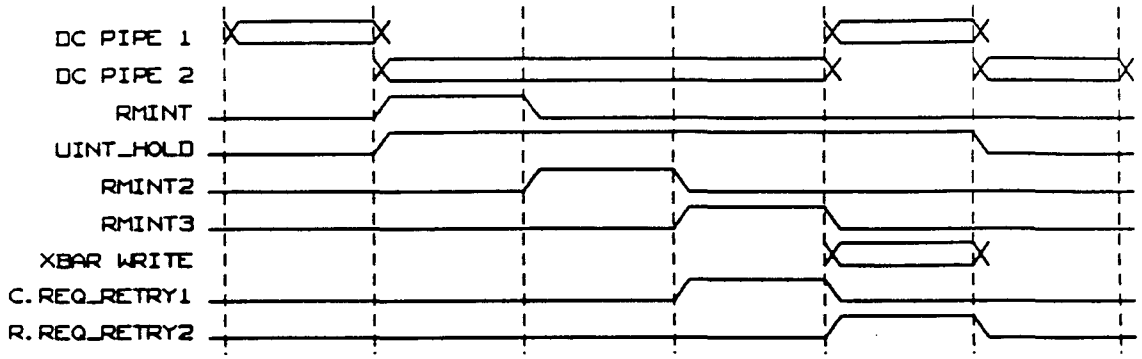
Associated with each location of the PTE cache is a referenced and a modified bit. These bits are an encached copy of the referenced and modified bits located in main memory. Each time a logical to physical address translation is performed the referenced and modified bits are checked to make sure they have been previously set. For reads only the referenced bit needs to be set. For writes both referenced and modified bits must be set. If either of these two bits need to be set but are not then a referenced/modified interrupt is processed.

A reference/modified interrupt is processed entirely within the NDC subsystem. The few steps which must be sequenced are well defined which allows the NDC subsystem to complete the

interrupt without requiring help from the NAS subsystem. Figure 5-26 shows the timing for a ref/mod interrupt. The NPA gate arrays generate the RMINT0/RMINT1 signals at the second stage of the data cache pipe. The signal RMINT is the logical or of RMINT0 and RMINT1 from the two NPA gate arrays. At the same time the signal UINT_HOLD is asserted holding the data cache pipe stages. The signal RMINT2 is a staged version of RMINT and used to enable the PTE cache referenced (and modified if a write) ram write. During the cycle that the PTE cache ref/mod rams are being enabled to write, the SA0 and SA1 address buses have the correct logical address.

The RMINT3 signal is a staged version of RMINT2 and is used to initiate a main memory write to

Figure 5-26 Referenced/Modified Interrupt Timing



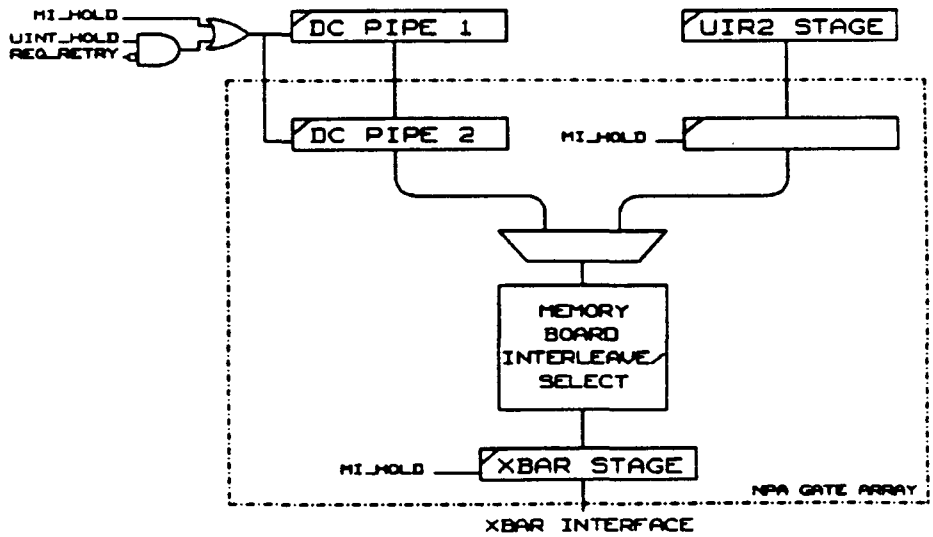
the referenced and modified bits and is also used to generate C.REQ_RETRY1. The signal C.REQ_RETRY1 initiates the request retry sequence. Refer to the previously described request retry sequence section for details.

The entire reference/modified interrupt requires four cycles.

5.8 Physical Address Generation

The NPA gate arrays implement the logic for physical address generation. There is an NPA gate array for the even side (NPA0) and one for the odd side (NPA1). Figure 5-27 shows the stages

Figure 5-27 Physical Address Staging



of registers for physical address generation. The normal path for addresses to flow is through the

data cache pipe stages, select the appropriate fields for logical to physical address translation, through the board interleave and select logic, and finally registered in the cross bar stage. The signal MI_HOLD is asserted whenever a cross bar request is at the cross bar register stage and must be held. MI_HOLD is used to hold the cross bar stage as well as the data cache pipe. The signal MI_HOLD holds the VAG request source and indicates that the cache is unavailable for the other request sources. Once a request reaches the cross bar stage, it will proceed to memory (or the communication board). When a PTE miss occurs the data cache pipe stages are held by UINT_HOLD with the offending request at the second stage data cache pipe registers. The NAS subsystem issues PTE requests to memory using the UIR2 interface. The main memory PTE table addresses are constructed using the second level data cache pipe logical address and the information provided by the UIR2 interface.

The following sections describe the various types of address translation which are performed by the NPA gate arrays. There are three sources of information which is used to generate the physical addresses. The first source of information is the second stage data cache pipe registers, logical address (R.SA0_MISS2), and PTE data (R.PRD_DATA). The second source is the UIR2 interface, R.YBUS_DATA. The last source is registers which are system, thread, and process constants. These registers include R.REFMOD_BASE, R.DC_TID, R.PID_CONF, and R.DC_CIR. R.REFMOD_BASE and R.PID_CONF are system constants which are scan initialized. R.REFMOD_BASE is the base address where the reference and modified bits in main memory are stored. R.PID_CONF is the processor identification number. The registers R.DC_TID and R.DC_CIR are loaded from the YBUS and are the thread identification number and communication index register.

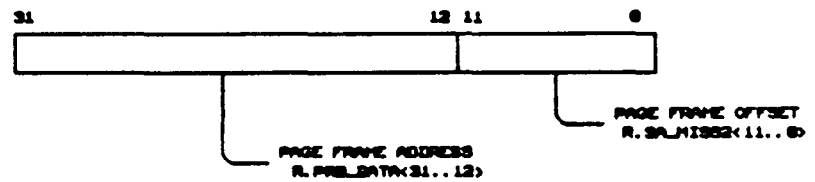
5.8.1 NAT - No address translation mode

This mode is simplest, the logical address is used as the physical address. The mode is used during the load physical address (LDPA) instruction where the micro code must access the main memory based page tables and int the diagnostic instruction (DIAG). No access or violation checks are performed in this mode.

5.8.2 VAT - Virtual address translation mode

This mode is used when a user process accesses memory. The physical address is constructed by using the page frame address which is the most significant 20 bits of the PTE data register, R.PRD_DATA, and the page frame offset which is the least significant 12 bits of R.SA0_MISS2. Figure 5-28 shows the format for virtual address translation.

Figure 5-28 Virtual Address Translation



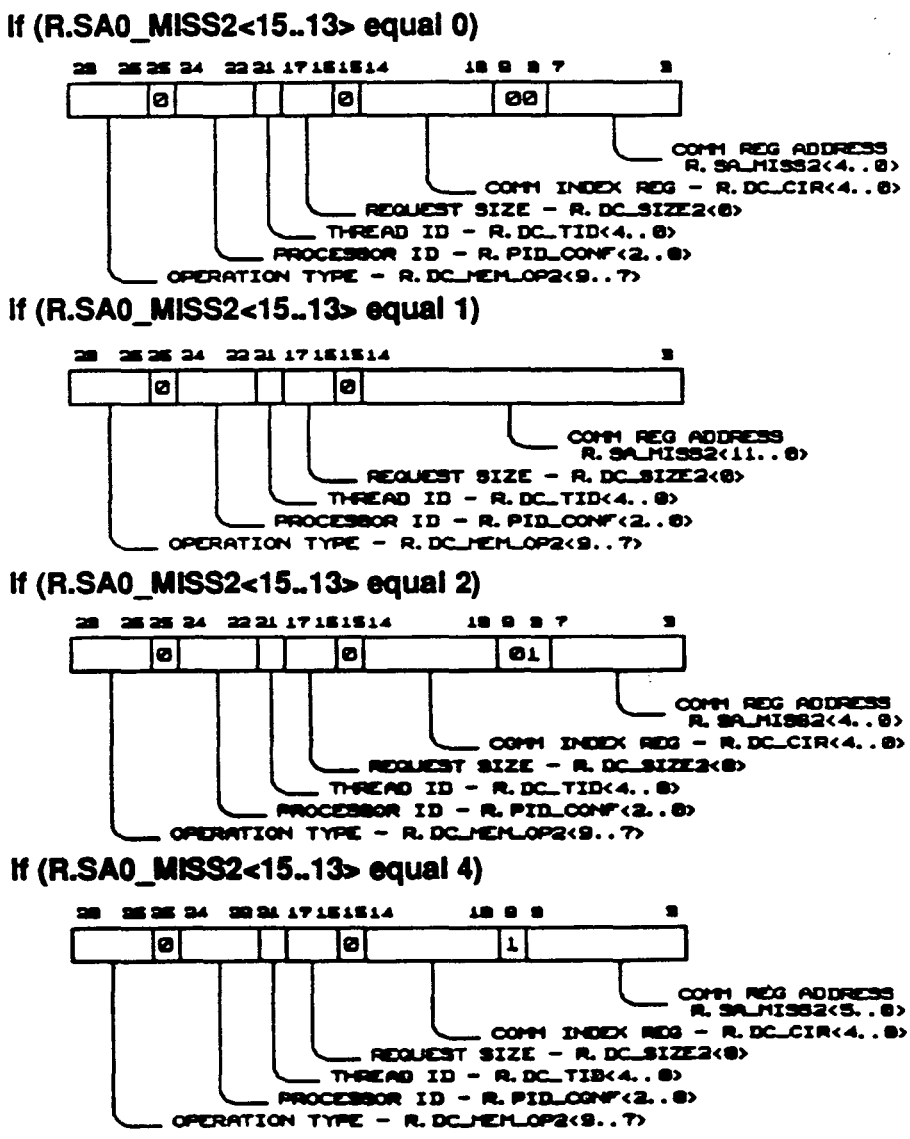
The violations which are checked in this mode include inward ring access, invalid thread level PTE, invalid second level PTE, read/write/execute access, and non-resident PTE data.

5.8.3 CRAT - Communication address translation mode

The CRAT mode is used when an access to the communication registers is made. The communication register space has 4096 locations, requiring only twelve bits of address. The other bits of the physical address consist of additional information required by communication requests. The most significant three bits of address space, <31..29>, are not used because they are decoded as memory board selects, and the least significant three bits are decoded as zone and even/odd selects. Figure 5-29 shows the format for communication register address translation.

The violations checked in this mode are inward communication access and invalid communication address. The check for an invalid communication address only looks at the least significant sixteen bits of the logical address. The valid addresses are R.SA0_MISS2<15..5> equal zero, R.SA0_MISS2<15..12> equal three, R.SA0_MISS2<15..5> equal 0x200, and R.SA0_MISS2<15..6> equal 0x200.

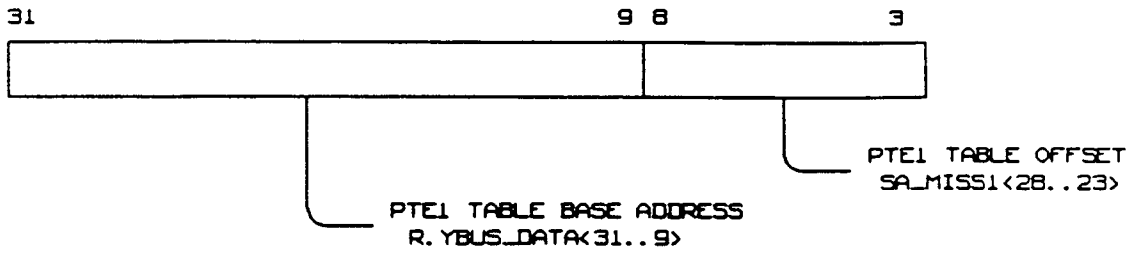
Figure 5-29 Communication Register Address Translation



5.8.4 PTE1 - First Level PTE Accesses

The first level PTE address generation mode uses the UIR2 interface shown in Figure 5-27. This type of access can only be issued when the data cache pipe is being held by UINT_HOLD. Figure 5-30 shows the format for the address translation.

Figure 5-30 First Level PTE Address Translation

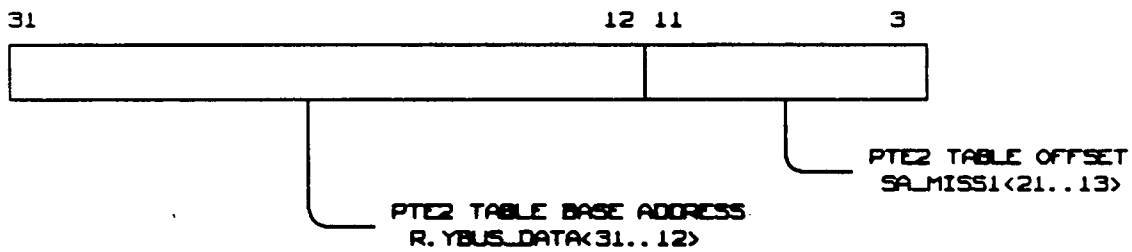


The NAS subsystem uses the three segment bits, <31..29>, of the logical address R.SA1_MISS1 to access a table of segment description registers (SDRs) contained in the scratch ram. The contents of the rams are sourced to the YBUS. The PTE1 table base address, bits <31..9> of the translated address, are obtained from the registered YBUS. The offset within the table is from the miss request's logical address bits <28..23>. Bit <22> of the logical address is used to select either the even or odd side of memory to obtain the page table entry.

5.8.5 PTE2 - Second level PTE accesses

The second level PTE address generation mode uses the UIR2 interface shown in Figure 5-27. This type of access can only be issued when the data cache pipe is being held by UINT_HOLD. Figure 5-31 shows the format for the address translation.

Figure 5-31 Second Level PTE Address Translation



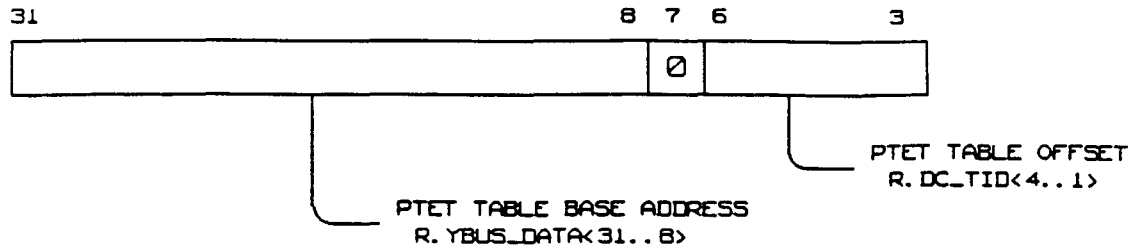
The NAS subsystem uses a first level page table entry to provide the second level table base address. The first level page table is obtained from either the PTE1 cache (scratch ram) or from a PTE1 access to main memory. The second level page table offset is obtained from the miss request's logical address bits <21..13>. Bit <12> of the logical address is used to select either the even or odd side of memory to obtain the page table entry.

5.8.6 PTET - Thread Level PTE Accesses

The thread level PTE address generation mode used the UIR2 interface shown in Figure 5-27. This page table level is only accessed when a second level page table entry is read from main memory and the thread level bit is set. Figure 5-32 shows the format for the thread level address translation.

For the thread level address the YBUS is sourced with the second level page table entry returned

Figure 5-32 Thread Level PTE Address Translation

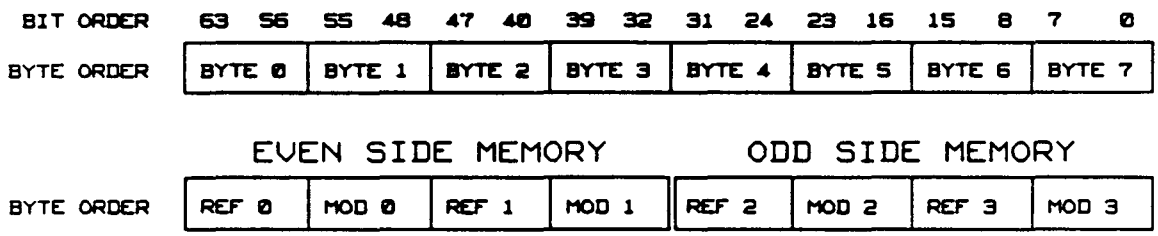


from memory from a PTE2 level access. The thread identification number (TID) is used to index into the thread level page table. Bits <4..1> of the TID is used as the least significant four bits of the physical address and bit <0> is used to direct the main memory request to either the even of odd side of memory.

5.8.7 REF/MOD - Reference/Modified bit accesses

The referenced and modified bits in main memory are interleaved in memory with one bit per byte. Organizing the bits by this means allows a processor to modify a byte to write a referenced bit and a halfword for both the referenced and modified bits of a page. Figure 5-33 illustrates the memory organization for ref/mod bits. Each byte contains the value one or zero. The data for a referenced

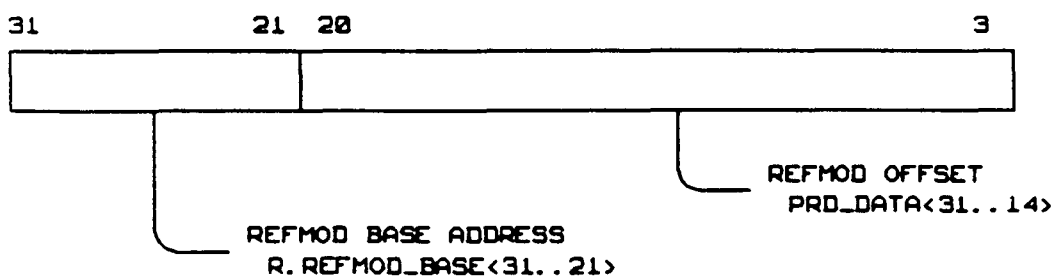
Figure 5-33 Referenced/Modified Memory Organization



or modified write is generated by the NDP gate arrays. The NDP gate arrays force the data to be a longword with each byte containing a one in the least significant bit and zeros in all other bits. The zone bits transferred with the request to memory specify which of the data bytes to actually write. Therefore, the address and data is generated the same whether the write is a referenced or referenced and modified.

Figure 5-34 shows the format used for the referenced/modified accesses. A scan initialized

Figure 5-34 Referenced/Modified Address Translation



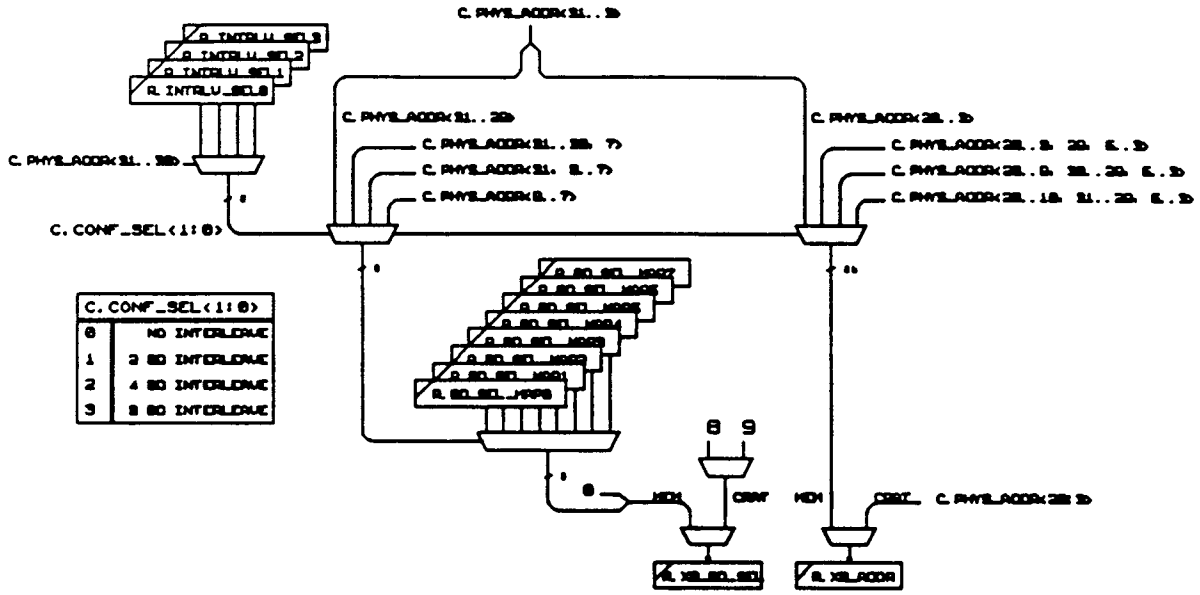
register is used to supply the referenced/modified bits base address and the physical address of the memory reference from the PTE data rams is used to supply the offset. Bit <13> of the PTE

data ram is used to specify if the write is to the even or odd side of memory and bit <12> specifies if the write is to the upper or lower halfword.

5.9 Cross Bar Address Generation

Requests transferred to the cross bar are destined to one of eight memory boards or to the communication board. The memory system can contain as much as 4 Gigabytes of actual memory, which is the full 32 bit address space. Figure 5-35 shows the logic used for memory

Figure 5-35 Memory Interleave and Board Select Logic



interleave and board select.

The most significant three bits of a physical address specifies which of the eight memory boards a request will go to. A one-to-one mapping of the three most significant physical address bits is implemented. An access to a logical memory board can be mapped to any of the physical memory boards. The mapping allows a memory board to be installed into any physical memory slot in the system and still have contiguous address space with other memory boards.

The NPA gate arrays modify the physical address to provide the maximum interleave possible for a memory system configuration.The memory system can have up to a maximum of eight memory boards. Each pair of memory boards 0/1, 2/3, 4/5, and 6/7 can have the memory interleave independently configured.

The most significant two bits of the translated address, C.PHYS_ADDR<31..30>, are used to select the interleave to be used by a pair of memory boards. The most significant three bits of the interleave address are then mapped to a physical memory board. The board select sent to the cross bar is four bits wide. Values zero through seven select memory boards, values eight and nine select the communication board.

5.10 CBUS Logic

The NDC subsystem writes to the register file within the NRFA gate arrays of the NAS subsystem using the CBUS interface. There are three sources of data which are transferred to the register

file using the CBUS: data cache read data, vector processor data, queued memory return data, and non-queued memory return data.

Figure 5-36 CBUS Functionality

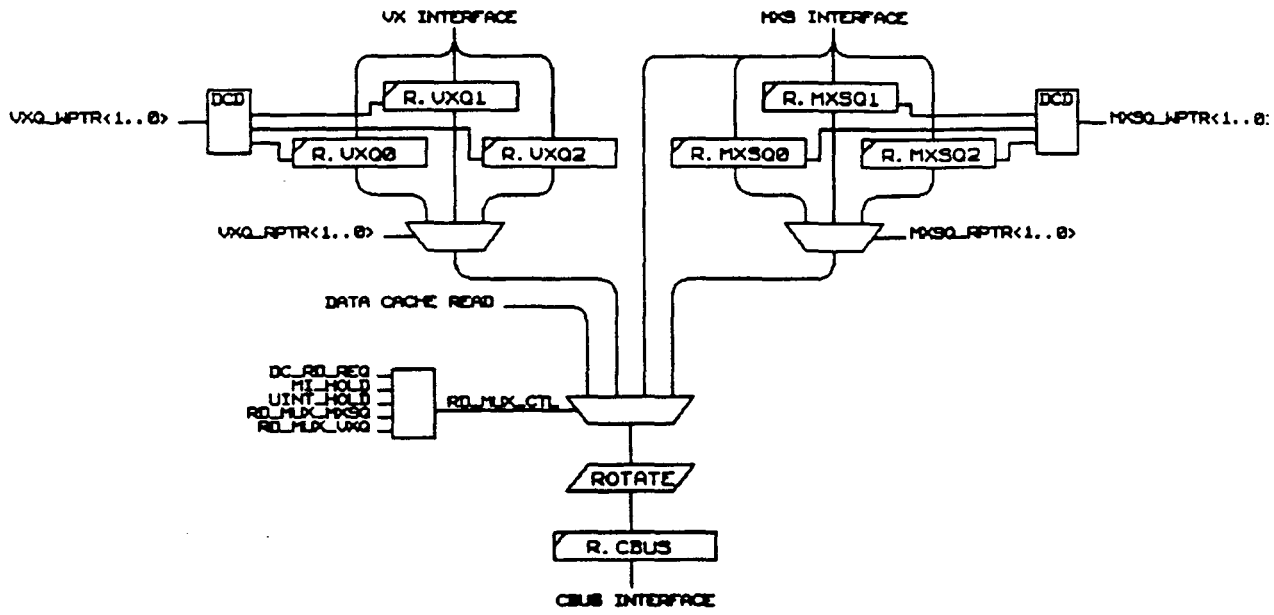


Figure 5-36 shows the generic structure used to implement the functionality. The logic is partitioned on the NDC gate array and the three NDP gate arrays. Each NDP gate array implements a three bit slice of each data byte resulting in 24 bits per gate array. Figure 5-37 shows which bits of the busses go to the three NDP gate arrays. Each NDP gate array also maintains identical copies of the rotation information for each source. Rotation control for the cache read data is obtained from SA1_MISS1<2..0>, the MXS interface uses UPD_ADDR<2..0>, and the vector data does not need rotated.

The NDC gate array contains the queue control logic, read mux selection logic, and queueing for register select and size information. The NDC gate array sends the queue read/write control and read mux control signals to the NDP gate arrays.

The logic which generates the RD_MUX_CTL signals arbitrates the selection of the four sources. The arbitration uses a fixed priority scheme. The priority of the inputs from highest to lowest is cache read data, queued vector data, queued memory data, and finally non-queued memory data.

The data cache will be selected if the first stage of the data cache pipe is performing a data cache read and the data cache pipe stages are not being held (MI_HOLD or UINT_HOLD). A data cache read operation reserves a cycle of the CBUS for data cache reads, but only if the read actually hits the data cache will data be transferred.

5.11 Return Control Interface

For each read request which is sent to memory, destination control information is transferred to the NRC subsystem. The return control (RC) interface is at the same register stage as the cross bar interface registers. A longword read request requires a transfer using both the even and odd memory sides and the RC interface. The three ready signal (one associated with each of the

interfaces) are asserted on the same cycle. If any of the three transfers are not completed then the signal MI_HOLD is asserted and the interface registers from all three interfaces are held. On the following cycle only the ready signal(s) of the interface(s) which were unable to be completed are reasserted.

Refer to the previous section describing internal NSP board interfaces for a description of the information transferred by the RC interface.

5.12 Context Save and Restore

A previous section described the process of missing the PTE cache when performing a virtual address translation (VAT). A UINT is issued to the NAS subsystem which micro interrupts its current micro code operation and executes micro code to fix the PTE cache problem. When this occurs the signal UINT_HOLD is asserted to hold the request which missed so that it can be retried when the PTE cache is written.

The micro code can sometimes resolve a PTE miss by referencing the page tables in main memory and writing the PTE cache. If the logical memory page is not resident in main memory (has never been accessed or has been swapped to disk) then the micro code can not correct the problem on its own. Access violations are another case where the micro code can not resolve the problem.

When the micro code can not resolve the problem, the context of the entire processor is saved by writing it to main memory. This operation is called a context save. Once context is saved the micro code jumps to the system exception handler to allow the operating system to correct the problem. Once the problem is corrected (if it can be) the operating system executes the instruction RTNC which initiates a context restore operation. The context restore reads the state from memory and restores it back to the original registers. Once the state is restored, the UINT_HOLD signal once again holds the data cache pipe until a request retry is issued from micro code.

Context for the NDC subsystem is organized as eighty bit context scan rings. All context for the NDC subsystem is contained in twenty of the context rings. The output of the context rings is fed into the external XMUX on the NAS subsystem. The input of the context rings is from the YBUS.

When saving state the scan ring output is selected by the external XMUX, registered by the R.XBUS registers, staged by the R.YBUS registers and presented back to the inputs of the context scan rings. The context ring is shifted 82 times, which shifts the data back into the original registers. During the save process the state is also written to the scratch rams to be transferred to memory later. After the context has been shifted, the control state is reset to invalidate all currently executing requests.

State is restored by first loading the scratch rams from memory and then reading and shifting the data back into the context rings.

Table 5-8 shows the organization of the NDC Subsystem context rings. The YBUS bit which is the input to the context rings is shown as well as the bit of NDC_CNTX output context bus. Notice on NDC_CNTX bits <9> and <15> that two gate arrays are specified. For these two bits the NAG gate array's context ring is 72 bits in length and the remainder of the ring is in the NPA gate array (8 bits).

Table 5-8 NDC Subsystem Context Rings

YBUS_DATA Bit	Gate Array	NDC_CNTX Bit
<2>	NDC Gate Array - Ring 0	<0>
<3>	NDC Gate Array - Ring 1	<1>
<4>	NDC Gate Array - Ring 2	<2>
<5>	NAG 0 Gate Array - Ring 0	<3>
<6>	NAG 0 Gate Array - Ring 1	<4>
<8>	NDP 0 Gate Array - Ring 0	<5>
<9>	NDP 0 Gate Array - Ring 1	<6>
<10>	NDP 1 Gate Array - Ring 0	<7>
<11>	NDP 1 Gate Array - Ring 1	<8>
<12>	NAG 0 Gate Array - Ring 0	<9>
<13>	NDP 2 Gate Array - Ring 0	<10>
<14>	NAG 0/NPA 0 Gate Arrays	<11>
<21>	NAG 1 Gate Array - Ring 0	<12>
<22>	NAG 1 Gate Array - Ring 1	<13>
<24>	NDP 0 Gate Array - Ring 2	<14>
<25>	NAG 1 Gate Array - Ring 0	<15>
<26>	NDP 1 Gate Array - Ring 2	<16>
<28>	NAG 1/NPA 1 Gate Arrays	<17>
<29>	NDP 2 Gate Array - Ring 1	<18>
<30>	NDP 2 Gate Array - Ring 2	<19>

The signal NSP_CNTX_CTL<1..0> is used to control the context save and restore operations. Table 5-9 lists the modes which can be specified by NSP_CNTX_CTL.

Table 5-9 Context control modes

0	Normal Mode
1	Context Hold Mode
2	Context Reset Mode
3	Context Shift Mode

Usually the mode is set to normal mode (zero). When context save micro code is entered the entire scalar processor and vector processor are put into hold mode at the same time. The micro code proceeds through the steps to save state, with one step being to enter context shift mode for 82 clocks. After shifting the mode is put back to hold until all scalar processor state is in the scratch rams. Finally the scalar processor state is reset using context reset mode. At this point the scalar processor stores context state in the scratch ram to memory in normal mode.

To restore context modes context hold and shift are used.

5.13 Modes of Operation

This section describes various modes of operation which have been implemented.

5.13.1 SQS - Sequential store mode

This mode is entered by setting the SQS bit of the processor status word (PSW) register. The mode forces the DC_HAZ signal to be asserted whenever a store request is at the second level UIR stage or in any of the data cache, and cross bar stages. Once the store request is transferred

from the cross bar to a memory board the hazard is released.

5.13.2 One page cache mode

The data cache is 16K bytes in size, four pages. This mode reduces the usable size of the data cache to one page (4K bytes). The mode is controlled by the 1PG_CACHE bit of the CCR register within the NAS subsystem. When the CCR bit is asserted the most significant two bits of the data cache address bus are forced to ones. The specific address lines are SA0_CPG<1..0>, SA1_CPG<1..0>, UA0<13..12>, and UA1<13..12>. Note that separate address lines are used for the SA0 and SA1 address busses since the logical address is not being changed, only the cache size.

5.13.3 Forced fault mode

This mode is used for diagnostic purposes to force context swaps. When the mode is enabled then all virtual address translation accesses will have forced fault type micro interrupt (UINT_VEC type 0xF) provided that no other micro interrupt condition is present. Upon returning from a forced fault, the request will be retried and allowed to complete without another forced fault.

The forced fault mode is entered by setting bits in the CCR register within the NAS subsystem. Separate bits exist to enable forced faults of IP prefetch requests and non-IP prefetch requests.

5.13.4 Forced hit mode

The mode allows the data cache to be read without checking the normal data and PTE cache tags and validity. The mode is used for diagnostic purposes. Forced hit mode is entered by setting a bit in the CCR register of the NAS subsystem.

5.13.5 Data cache off mode

This mode disables the data cache for reads and writes. The mode is enabled by setting a bit in the CCR register within the NAS subsystem. The data cache is disabled by forcing the DC_HIT signals to zero, and forcing the DC_SEL and DC_USEL signals to one (which generate the write enables to the rams).

5.14 Parity Error Sources

This section lists the sources of parity errors which are generated by the NDC subsystem. The list is organized by board parity error signal. Gate array parity error signals can represent multiple parity error sources. The information listed for each source include: parity error register, list of registers and corresponding board level signals which parity is checked, list of registers which are held, queue indices for parity checked at output of a queue, ram being checked, and staged ram address and control.

5.14.1 NDC gate array parity errors

The NDC gate array checks parity for the control store fields it receives. The signals are registered and parity is checked on the output of the register. The registers are held when a parity error is detected.

Parity Error Signal: NDC_PAR_ERR

Parity Error Register: NDC.UIR1_PAR_ERR

Registers/Signals:	NDC.UIR1_NDC_PAR	US_NDC_PAR
	NDC.UIR1_VP_DISP	US_VP_DISP
	NDC.UIR1_SXV_REQ	US_SXV_REQ
	NDC.UIR1_YZ_WR_DST<2..0>	US_YZ_WR_DST<2..0>
	NDC.UIR1_VXS_REQ	US_VXS_REQ
	NDC.UIR1_REQ_RETRY	US_REQ_RETRY
	NDC.UIR1_FAKE_RING0	US_FAKE_RING0
	NDC.UIR1_MEM_OP_REQ	US_MFP_OP<0>
	NDC.UIR1_PTE_OP_REQ	US_MFP_OP<12>
	NDC.UIR1_MEM_OP<10..0>	US_MFP_OP<11..1>
	NDC.UIR1_SIZE<1..0>	US_BDSIZE<1..0>

5.14.2 NAG0 gate array parity errors

The NAG slice zero gate array checks parity for all addresses and control store fields it receives. The parity for addresses which are queued is checked after the read mux for the queue. When a parity error is detected both the queue data registers and queue read pointer register is held.

Parity error Signal:	NAG0_PAR_ERR	
Parity Error Register:	NAG0.VXAQ_PAR_ERR	
Registers/Signals:	NAG0.VXAQ_ADDR{0,1,2}<15..0>	VXA_ADDR<15..0>
	NAG0.VXAQ_ADDR{0,1,2}_PAR<1..0>	VXA_ADDR_PAR<3..2>
Queue Read Pointer:	NAG0.VXAQ_RPTR_LAS<1..0>	
Parity Error Register:	NAG0.IXAQ_PAR_ERR	
Registers/Signals:	NAG0.IXAQ_ADDR{0,1,2}<15..0>	IXA_ADDR<15..0>
	NAG0.IXAQ_ADDR{0,1,2}_PAR<1..0>	IXA_ADDR_PAR<3..2>
Queue Read Pointer:	NAG0.IXAQ_RPTR_LAS<1..0>	
Parity Error Register:	NAG0.UPDQ_PAR_ERR	
Registers/Signals:	NAG0.UPDQ_ADDR{0,1,2}<15..0>	UPD_ADDR<15..0>
	NAG0.UPDQ_ADDR{0,1,2}_PAR<1..0>	UPD_ADDR_PAR<3..2>
Queue Read Pointer:	NAG0.UPDQ_RPTR_LAS<1..0>	
Parity Error Register:	NAG0.ZBUS_PAR_ERR	
Registers/Signals:	NAG0.AG_ZBUS<15..0>	ZMUX_DATA<15..0>
	NAG0.AG_ZBUS_PAR<1..0>	ZMUX_PAR<3..2>
Parity Error Register:	NAG0.DISPL_PAR_ERR	
Registers/Signals:	NAG0.UPC_DISPL<15..0>	AS_DISP_DISPL<15..0>
	NAG0.UPC_DISPL_PAR<3..2>	AS_DISP_DISPL_PAR<3..2>
Parity Error Register:	NAG0.UIR1_PAR_ERR	
Registers/Signals:	NAG0.UIR1_NAG_PAR	US_NAG0_PAR
	NAG0.UIR1_AG_SEL<3..0>	US_AG_SEL<3..0>

5.14.3 NAG1 gate array parity errors

The NAG slice one gate array checks parity for all addresses and control store fields it receives. The parity for addresses which are queued is checked after the read mux for the queue. When a

parity error is detected both the queue data registers and queue read pointer register is held.

Parity error Signal:	NAG1_PAR_ERR	
Parity Error Register:	NAG1.VXAQ_PAR_ERR	
Registers/Signals:	NAG1.VXAQ_ADDR{0,1,2}<15..0>	VXA_ADDR<31..16>
	NAG1.VXAQ_ADDR{0,1,2}_PAR<1..0>	VXA_ADDR_PAR<1..0>
Queue Read Pointer:	NAG1.VXAQ_RPTR_LAS<1..0>	
Parity Error Register:	NAG1.IXAQ_PAR_ERR	
Registers/Signals:	NAG1.IXAQ_ADDR{0,1,2}<15..0>	IXA_ADDR<31..16>
	NAG1.IXAQ_ADDR{0,1,2}_PAR<1..0>	IXA_ADDR_PAR<1..0>
Queue Read Pointer:	NAG1.IXAQ_RPTR_LAS<1..0>	
Parity Error Register:	NAG1.UPDQ_PAR_ERR	
Registers/Signals:	NAG1.UPDQ_ADDR{0,1,2}<15..0>	UPD_ADDR<31..16>
	NAG1.UPDQ_ADDR{0,1,2}_PAR<1..0>	UPD_ADDR_PAR<1..0>
Queue Read Pointer:	NAG1.UPDQ_RPTR_LAS<1..0>	
Parity Error Register:	NAG1.ZBUS_PAR_ERR	
Registers/Signals:	NAG1.AG_ZBUS<15..0>	ZMUX_DATA<31..16>
	NAG1.AG_ZBUS_PAR<1..0>	ZMUX_PAR<1..0>
Parity Error Register:	NAG1.DISPL_PAR_ERR	
Registers/Signals:	NAG1.UPC_DISPL<15..0>	AS_DISP_DISPL<31..16>
	NAG1.UPC_DISPL_PAR<3..2>	AS_DISP_DISPL_PAR<1..0>
Parity Error Register:	NAG1.UIR1_PAR_ERR	
Registers/Signals:	NAG1.UIR1_NAG_PAR	US_NAG0_PAR
	NAG1.UIR1_AG_SEL<3..0>	US_AG_SEL<3..0>
	NAG1.UIR1_PTE_OP_REQ	US_PTE_OP_REQ
	NAG1.UIR1_PTE_OP<2..0>	US_MFP_OP<10..8>
	NAG1.UIR1_REQ_RETRY	US_REQ_RETRY

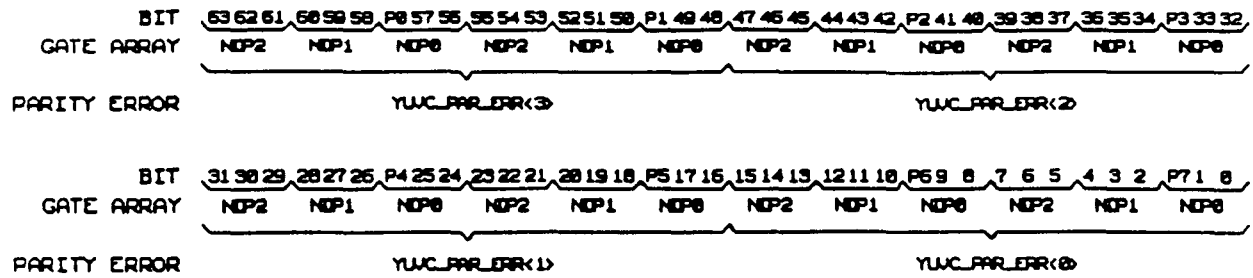
5.14.4 NDP{0,1,2} gate array common parity errors

This section describes parity errors that are generated from data on all three NDP gate arrays. Each gate array generates partial parity information and sends the signals to external PALs which generate the final parity error signals. The final parity error signals are returned back to all three NDP gate arrays. When a parity error is detected, all three NDP gate arrays will indicate the error.

The NDP gate arrays are partitioned to allow byte rotation without chip crossings. This partitioning is accomplished by having each of the three NDP gate arrays handle three bits for each byte (one gate arrays has two data bits and one bit of parity). Figure 5-37 shows the partitioning of the data and parity busses for the NDP gate arrays.

There are four data busses which parity must be checked with the split gate array partitioning, YBUS_DATA, UPD_DATA, VP_SP.DATA, and CRD_DATA (YUVC). The number of pins on the gate array was limited so multiple bytes of the previously mentioned busses are checked together. Parity error signal YUVC_PAR_ERR<0> indicates that a parity error was detected on one (or more) of bytes six and seven of the four busses. Each gate array generates four partial parity

Figure 5-37 NDP gate array common parity errors



signals, one for each of the parity error signals. The partial parity signals are the exclusive-or of all of the bits of the bytes being checked.

When a parity error is detected all of the data registers are held as well as the staged cache ram address and control.

Parity Error Signal:	NDP{0,1,2}_PAR_ERR	
Parity Error Register:	NDP{0,1,2}.YUVC_PAR_ERR<3..0>	
Registers/Signals:	NDP{0,1,2}.YBUS<23..0>	YBUS_DATA<63..0>
		YBUS_PAR<7..0>
	NDP{0,1,2}.UPD<23..0>	UPD_DATA<63..0>
		UPD_PAR<7..0>
	NDP{0,1,2}.VX<23..0>	VP_SP.DATA<63..0>
		VP_SP.PAR<7..0>
	NDP{0,1,2}.CRD<23..0>	CRD_DATA0<35..0>
		CRD_DATA1<35..0>
Data Cache Address:	NPA0.SA_RD_LAS4<13..3>	SA0<13..3>
	NPA1.SA_RD_LAS4<13..3>	SA1<13..3>
Data Cache Control:	NPA0.DC_WE_LAS3	
	NPA1.DC_WE_LAS3	

5.14.5 NDP0 Gate Array Parity Errors

The NDP slice zero gate array uses the GP_IN<9..0> inputs to check parity for ram outputs that are not checked elsewhere.

Parity Error Signal:	NDP0_PAR_ERR	
Parity Error Register:	NDP0.CUPD_PAR_ERR	
Registers/Signals:	NDP0.CRD_UPD<5..0>	CRD_UPD0<5..0>
Update Cache Address:	NAG0.UA_RD_LAS4<13..3>	UA0<13..3>
Update Cache Control:	NPA0.UPD_WE_LAS3	
Parity Error Register:	NDP0.PREF_PAR_ERR	
Registers/Signals:	NDP0.PRD_REF<1..0>	PRD_REF0<1..0>
	NPA0.PRD_REF_LAS3	PRD_REF0<0>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.REF_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	

Parity Error Register:	NDP0.PMOD_PAR_ERR	
Registers/Signals:	NDP0.PMOD_PAR_ERR	PRD_MOD0<1..0>
	NPA0.PRD_MOD_LAS3	PRD_MOD0<0>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.MOD_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	

5.14.6 NDP1 Gate Array Parity Errors

The NDP slice one gate array uses the GP_IN<9..0> inputs to check parity for ram outputs that are not checked elsewhere.

Parity Error Signal: NDP1_PAR_ERR

Parity Error Register:	NDP1.CUPD_PAR_ERR	
Registers/Signals:	NDP1.CRD_UPD<5..0>	CRD_UPD1<5..0>
Update Cache Address:	NAG1.UA_RD_LAS4<13..3>	UA1<13..3>
Update Cache Control:	NPA1.UPD_WE_LAS3	
Parity Error Register:	NDP1.PREF_PAR_ERR	
Registers/Signals:	NDP1.PRD_REF<1..0>	PRD_REF1<1..0>
	NPA1.PRD_REF_LAS3	PRD_REF1<0>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.REF_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
Parity Error Register:	NDP1.PMOD_PAR_ERR	
Registers/Signals:	NDP1.PMOD_PAR_ERR	PRD_MOD1<1..0>
	NPA1.PRD_MOD_LAS3	PRD_MOD1<0>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.MOD_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	

5.14.7 NPA0 Gate Array Parity Errors

All errors checked by the NPA side zero gate array is from the even side cache rams.

Parity Error Signal: NPA0_PAR_ERR

Parity Error Register:	NPA0.PTAG_PAR_ERR	
Registers/Signals:	NPA0.PRD_TAG<33..17>	PRD_TAG0<33..17>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Register:	NPA0.PDATA_PAR_ERR	
Registers/Signals:	NPA0.PRD_DATA<35..0>	PRD_DATA0<35..0>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Register:	NPA0.CTAG_PAR_ERR	

Registers/Signals:	NPA0.CRD_TAG<35..0>	CRD_TAG0<35..0>
Ram Address:	NPA0.SA_RD_LAS4<13..3>	SA0<13..3>
Ram Write:	NPA0.DC_WE_LAS3	
Parity Error Register:	NPA0.PVAL_PAR_ERR	
Registers/Signals:	NPA0.PRD_VAL<1..0>	PRD_VAL0<1..0>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
Parity Error Register:	NPA0.PTVAL_PAR_ERR	
Registers/Signals:	NPA0.PRD_TVAL<1..0>	PRD_TVAL0<1..0>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
Parity Error Register:	NPA0.CVAL_PAR_ERR	
Registers/Signals:	NPA0.CRD_VAL<1..0>	CRD_VAL0<1..0>
Ram Address:	NPA0.SA_RD_LAS4<13..3>	SA0<13..3>
Ram Write:	NPA0.DC_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
→ Parity Error Register:	NPA0.CTVAL_PAR_ERR	
Registers/Signals:	NPA0.CRD_TVAL<1..0>	CRD_TVAL0<1..0>
Ram Address:	NPA0.SA_RD_LAS4<13..3>	SA0<13..3>
Ram Write:	NPA0.DC_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	

5.14.8 NPA1 Gate Array Parity Errors

All errors checked by the NPA side one gate array are from the odd side cache rams.

Parity Error Signal:	NPA1_PAR_ERR	
Parity Error Register:	NPA1.PTAG_PAR_ERR	
Registers/Signals:	NPA1.PRD_TAG<33..17>	PRD_TAG1<33..17>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Register:	NPA1.PDATA_PAR_ERR	
Registers/Signals:	NPA1.PRD_DATA<35..0>	PRD_DATA1<35..0>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Register:	NPA1.CTAG_PAR_ERR	
Registers/Signals:	NPA1.CRD_TAG<35..0>	CRD_TAG1<35..0>
Ram Address:	NPA1.SA_RD_LAS4<13..3>	SA1<13..3>
Ram Write:	NPA1.DC_WE_LAS3	
Parity Error Register:	NPA1.PVAL_PAR_ERR	

Registers/Signals:	NPA1.PRD_VAL<1..0>	PRD_VAL1<1..0>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
Parity Error Register:	NPA1.PTVAL_PAR_ERR	
Registers/Signals:	NPA1.PRD_TVAL<1..0>	PRD_TVAL1<1..0>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
Parity Error Register:	NPA1.CVAL_PAR_ERR	
Registers/Signals:	NPA1.CRD_VAL<1..0>	CRD_VAL1<1..0>
Ram Address:	NPA1.SA_RD_LAS4<13..3>	SA1<13..3>
Ram Write:	NPA1.DC_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	
Parity Error Register:	NPA1.CTVAL_PAR_ERR	
Registers/Signals:	NPA1.CRD_TVAL<1..0>	CRD_TVAL1<1..0>
Ram Address:	NPA1.SA_RD_LAS4<13..3>	SA1<13..3>
Ram Write:	NPA1.DC_WE_LAS3	
Ram Purge:	NDC.PURGE_OP2_LAS3<3..0>	

5.14.9 Cache Ram Write Parity Errors

The self-timed rams used to implement the data and PTE caches check data for parity errors when a write to the ram is performed. The data with the parity error is written into the ram, and therefore the ram must be read to recover the bad data.

Parity Error Signal:	DCD0_WR_PAR_ERR<3..0>	
Parity Error Register:	NSP_CLK.DCD0_WR_PAR_ERR<3..0>	
Rams/Signals:	DC_DATA0[0,1,2,3]	CWR_DATA0<35..0>
Ram Address:	NAG0.SA0_WR_LAS3<13..3>	SA0<13..3>
Ram Write:	NPA0.DC_WE_LAS3	
Parity Error Signal:	DCT0_WR_PAR_ERR<3..0>	
Parity Error Register:	NSP_CLK.DCT0_WR_PAR_ERR<3..0>	
Rams/Signals:	DC_TAG0[0,1,2,3]	SA0<23..12> DC_CIR<4..0> CWR_ZONE0<3..0>
Ram Address:	NAG0.SA0_WR_LAS3<13..3>	SA0<13..3>
Ram Write:	NPA0.DC_WE_LAS3	
Parity Error Signal:	DCU0_WR_PAR_ERR	
Parity Error Register:	NSP_CLK.DCU0_WR_PAR_ERR	
Rams/Signals:	DC_UPD0	CWR_UPD0<5..0>
Ram Address:	NAG0.UA0_WR_LAS3<13..3>	UA0<13..3>
Ram Write:	NPA0.UPD_WE_LAS3	

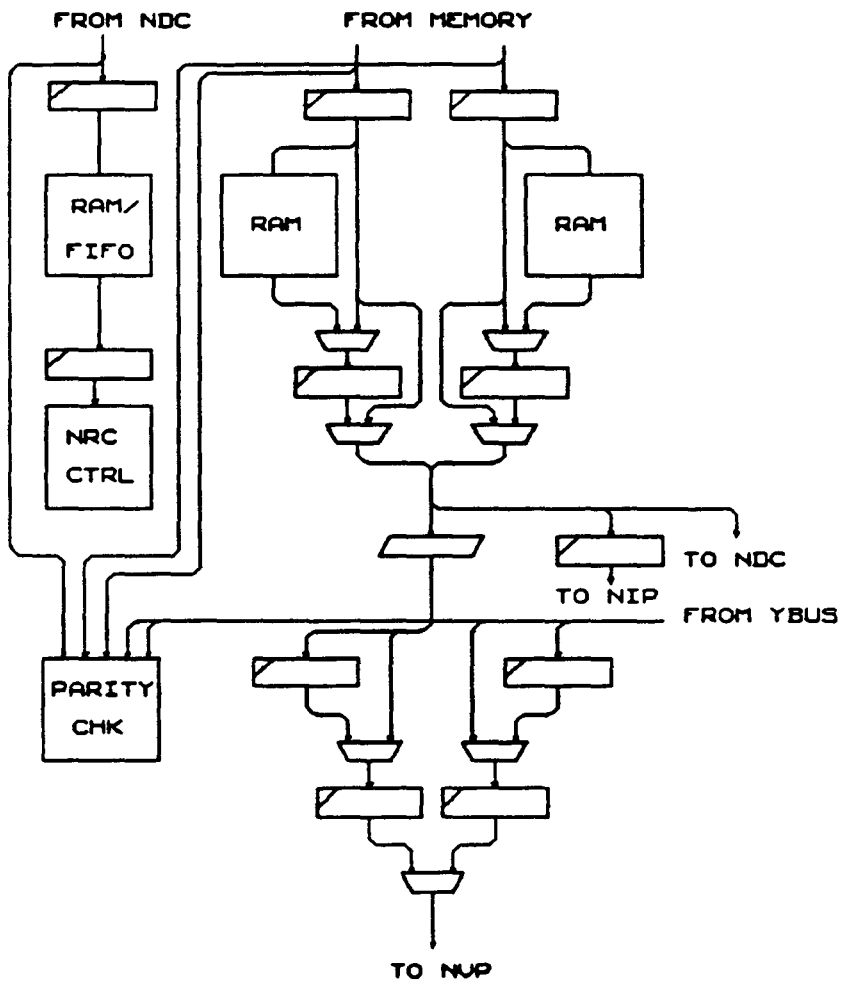
Parity Error Signal:	PTED0_WR_PAR_ERR<3..0>	
Parity Error Register:	NSP_CLK.PTED0_WR_PAR_ERR<3..0>	
Rams/Signals:	PTE_DATA0[0,1,2,3]	YBUS_DATA<31..0> YBUS_PAR<7..4>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Signal:	PTET0_WR_PAR_ERR<1..0>	
Parity Error Register:	NSP_CLK.PTET0_WR_PAR_ERR<1..0>	
Rams/Signals:	PTE_TAG0[0,1]	SA0<31..23> DC_CIR<4..0>
Ram Address:	NPA0.SA_RD_LAS4<22..12>	SA0<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Signal:	DCD1_WR_PAR_ERR<3..0>	
Parity Error Register:	NSP_CLK.DCD1_WR_PAR_ERR<3..0>	
Rams/Signals:	DC_DATA1[0,1,2,3]	CWR_DATA1<35..0>
Ram Address:	NAG0.SA1_WR_LAS3<13..3>	SA1<13..3>
Ram Write:	NPA1.DC_WE_LAS3	
Parity Error Signal:	DCT1_WR_PAR_ERR<3..0>	
Parity Error Register:	NSP_CLK.DCT1_WR_PAR_ERR<3..0>	
Rams/Signals:	DC_TAG1[0,1,2,3]	SA1<23..12> DC_CIR<4..0> CWR_ZONE1<3..0>
Ram Address:	NAG0.SA1_WR_LAS3<13..3>	SA1<13..3>
Ram Write:	NPA1.DC_WE_LAS3	
Parity Error Signal:	DCU1_WR_PAR_ERR	
Parity Error Register:	NSP_CLK.DCU1_WR_PAR_ERR	
Rams/Signals:	DC_UPD1	CWR_UPD1<5..0>
Ram Address:	NAG0.UA1_WR_LAS3<13..3>	UA1<13..3>
Ram Write:	NPA1.UPD_WE_LAS3	
Parity Error Signal:	PTED1.WR_PAR_ERR<3..0>	
Parity Error Register:	NSP_CLK.PTED1_WR_PAR_ERR<3..0>	
Rams/Signals:	PTE_DATA1[0,1,2,3]	YBUS_DATA<31..0> YBUS_PAR<7..4>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.PTE_WE_LAS3	
Parity Error Signal:	PTET1_WR_PAR_ERR<1..0>	
Parity Error Register:	NSP_CLK.PTET1_WR_PAR_ERR<1..0>	
Rams/Signals:	PTE_TAG1[0,1]	SA1<31..23> DC_CIR<4..0>
Ram Address:	NPA1.SA_RD_LAS4<22..12>	SA1<22..12>
Ram Write:	NDC.PTE_WE_LAS3	

6 NRC Subsystem

6.1 Overview

The NRC subsystem queues control information for read requests which have been sent to the cross bar for memory/communication access. When the requested data is received from the cross bar to the NSP board, it is matched up with the appropriate queued control information and transferred to the destination subsystem. The NRC subsystem is implemented with three copies of a single gate array type, Neptune Return Control gate array. NRC will refer to the NRC subsystem in this specification.

Figure 6-1 NRC Flow Diagram



The NRC receives information on the data being requested from memory by the NDC usually as the request is being made. The information includes the address and control information which is necessary to return memory data to the proper destination. A memory request may be made to the even and/or odd sides of memory, and the data will return on the same side and in the order that the requests were made, so the control information is registered as shown in Figure 6-1, and then enters the control FIFO. The control FIFO is a queue which holds the control information until the corresponding data is returned from memory. The data returns from memory in even and/or

odd 32 bit words plus parity which are registered, then may enter the data RAM or if the rams are empty or the request is a Page Table Entry (PTE), the data may pass straight through to be handed off to the Instruction Processor (NIP), the Data Cash (NDC) or may be rotated and registered again before being passed off to the Vector Processor (NVP.) Each aspect of the control and data paths is discussed in detail below.

6.2 Implementation

The NRC is implemented completely in three copies of the NRC array, which is a Vitesse 20k arrays with RAM. This implementation forced some peculiarities on the design which will be discussed here. The three arrays are bit sliced across most data buses as shown in Table 6-1, which allows each part to contain the bits necessary from each byte of data to enable byte rotates. For example, gate array 1 could rotate bits 4,3 and 2 in byte 3 into the same bit positions in byte 2 which are 12, 11, and 10.

Table 6-1 Bit positions of SA1_MISS1 among 3 NRC arrays

Byte No.	Gate Array 0	Gate Array 1	Gate Array 2
Byte 0	25,24, Parity 0	28,27, 26	31,30,29
Byte 1	17,16, Parity 1	20,19,18	23,22,21
Byte 2	9, 8, Parity 2	12,11, 10	15,14,13
Byte 3	1, 0, Parity 3	4, 3, 2	7, 6, 5

Gate Array 0 is responsible for generating certain control signals which are used by all three arrays. For example, each of the three arrays has the ability to save RC_EVEN, RC_ODD, RC_PTE, RC_2T_FIRST, two bits of RC_TAG, and 2 bits of RC_REG_SEL. The information received by each array in these locations is shown in Table 6-1. Each array requires certain fields for its own internal control, but the byte rotation is determined by Gate Array 0 and sent to the other array using RC_SIZE, RC_DST, RC_2T_FIRST and the least significant 3 bits of the address presented in the SA1 fields, the timing of which is described in section 2.2.6 above. The details of the rotation calculation are described in the control section below.

Gate Array 0 is also the only array which receives the handshakes from the memory which correspond to the memory data as described in section 2.1.2 above. It registers the handshakes then sends the handshake off-chip to the other two arrays and returns the signal to itself so that all three may deal with the registered handshake identically.

Table 6-2 RC signals saved by the 3 NRC arrays

Field	Gate Array 0	Gate Array 1	Gate Array 2
RC_EVEN	RC_EVEN	RC_EVEN	RC_EVEN
RC_ODD	RC_ODD	RC_ODD	RC_ODD
RC_PTE	RC_PTE	RC_PTE	RC_PTE
RC_2T_FIRST	RC_2T_FIRST	RC_2T_FIRST	RC_2T_FIRST
RC_TAG	RC_TAG<0>: ZERO	RC_TAG<2:1>	RC_TAG<4:3>
RC_REG_SEL	RC_REG_SEL<0>: RC_SIZE<1>	RC_REG_SEL<2:1>	RC_REG_SEL<4:3>
RC_DST	RC_DST<0>	RC_DST<1>	RC_DST<2>
RC_SIZE	RC_SIZE<0>	ZERO	ZERO
SA1_MISS1	SA1_MISS1<26:24, 18:16,10:8>	SA1_MISS1<29:27, 21:19,13:11,5:3>	SA1_MISS1<31:30, 23:22,15:14,7:6>
SA1_ADDR2	SA1_ADDR2<2:0>	N/U	N/U
SA1_ADDR2_PAR	N/U	N/U	SA1_ADDR2_PAR<3:0>

The Vitesse 20k array with RAM contains eight ram sections which are 256x5. This allows each ram section to contain 4 bits of data plus parity on that data to verify the RAM section's integrity. The NRC function places 12 bits of data from each of the memory buses plus 20 bits of control information in the queue of each of the three arrays, for a total of 44 bits of data to be queued but only 32 bits of RAM are available (5x8 minus parity for each section.) The 24 bits of memory data with 8 bits of the control use all available RAM, so the remainder of the control data to be queued is placed in a register array which has been designed to read and write nearly the same as the RAM sections. Which information is stored in the RAMs and which is stored in the register array should be invisible during normal operation but requires somewhat different handling during context saves and restores, as will be discussed below. Note that it is necessary to run the RAM array at 2x the normal clock to enable reads to the queues during the first half cycle of the normal clock and writes during the second half cycle. The register array is simply written at the end of the normal clock cycle and is read through a mux during the entire clock cycle.

Yet another difficulty imposed by implementation is parity checking across the three arrays. When R.PAR_ERR_DISABLE is cleared and the system is in normal mode, parity is checked on 5 buses: R.OYBUS_DATA, R.EYBUS_DATA, R.XRO_DATA, R.XRE_DATA and R.SA1_MISS3. The bits of the buses to be checked are XORed to generate parity data as shown in Table 6-1 below. Each array generates 8 bits of parity data, PAR_ERR_STOP_OUT<7..0> from these buses, which is actually just two copies of four bits of parity data. Each of the three arrays receives either bits 7..4 or 3..0 of the PAR_ERR_STOP_OUT from each of the other two arrays and XORs that data with its own PAR_ERR_STOP_OUT<3..0> to check for parity errors. If a parity error exists, this method assures that each array detects it at the same time and each will hold all 5 buses used to generate the PAR_ERR_STOP_OUT signals, including whichever one contains the parity error.

Table 6-3 Bits used in generation of PAR_ERR_STOP_OUT

BIT No.	OYBUS DATA	EYBUS DATA	XRO DATA	XRE DATA	SA1 MISS3
0	2, 1, 0	2, 1, 0	2, 1, 0	2, 1, 0	2, 1, 0
1	5, 4, 3	5, 4, 3	5, 4, 3	5, 4, 3	5, 4, 3
2	8, 7, 6	8, 7, 6	8, 7, 6	8, 7, 6	8, 7, 6
3	11,10,9	11,10,9	11,10,9	11,10,9	11,10,9
4	2, 1, 0	2, 1, 0	2, 1, 0	2, 1, 0	2, 1, 0
5	5, 4, 3	5, 4, 3	5, 4, 3	5, 4, 3	5, 4, 3
6	8, 7, 6	8, 7, 6	8, 7, 6	8, 7, 6	8, 7, 6
7	11,10,9	11,10,9	11,10,9	11,10,9	11,10,9

6.3 NRC Control

The NRC control consists of all information necessary for the NRC to return memory data to the proper destination. This information includes RC_SIZE<1..0>, RC_PTE, RC_ODD, RC_EVEN, RC_REG_SEL<4..0>, RC_TAG<4..0>, RC_DST<2..0>, SA1_MISS1<31..3>, SA1_ADDR2<2..0>, and SA1_ADDR2_PAR<3..0> and the transfer handshake signals RC_RDY and RC_REQ_NEXT. The description of each of these signals and timing of the handshakes is described in sections 2.2.6 RC - Memory return destination control interface, as well as in the signal appendix. Briefly, RC_2T_FIRST indicates that two requests to the cross bar were required for a single data cache request due to an unaligned request. RC_DST specifies the destination for the returned data, such as the NIP, NVP, etc. The RC_EVEN and RC_ODD signals specify which side of the cross bar the data was requested from. The RC_PTE signal marks a request for a Page Table Entry (PTE) which requires priority handling in the NRC and avoids the normal queues. RC_REG_SEL and RC_SIZE are used when the destination is the register file of the NAS subsystem. The RC_TAG field is used for NDC update requests. RC_PTE, RC_EVEN, RC_ODD, RC_2T_FIRST and RC_SIZE and SA1_ADDR2 are used by the NRC to determine data flow, rotation and destination. The other RC fields mentioned above simply pass through the NRC to be used by the data destination.

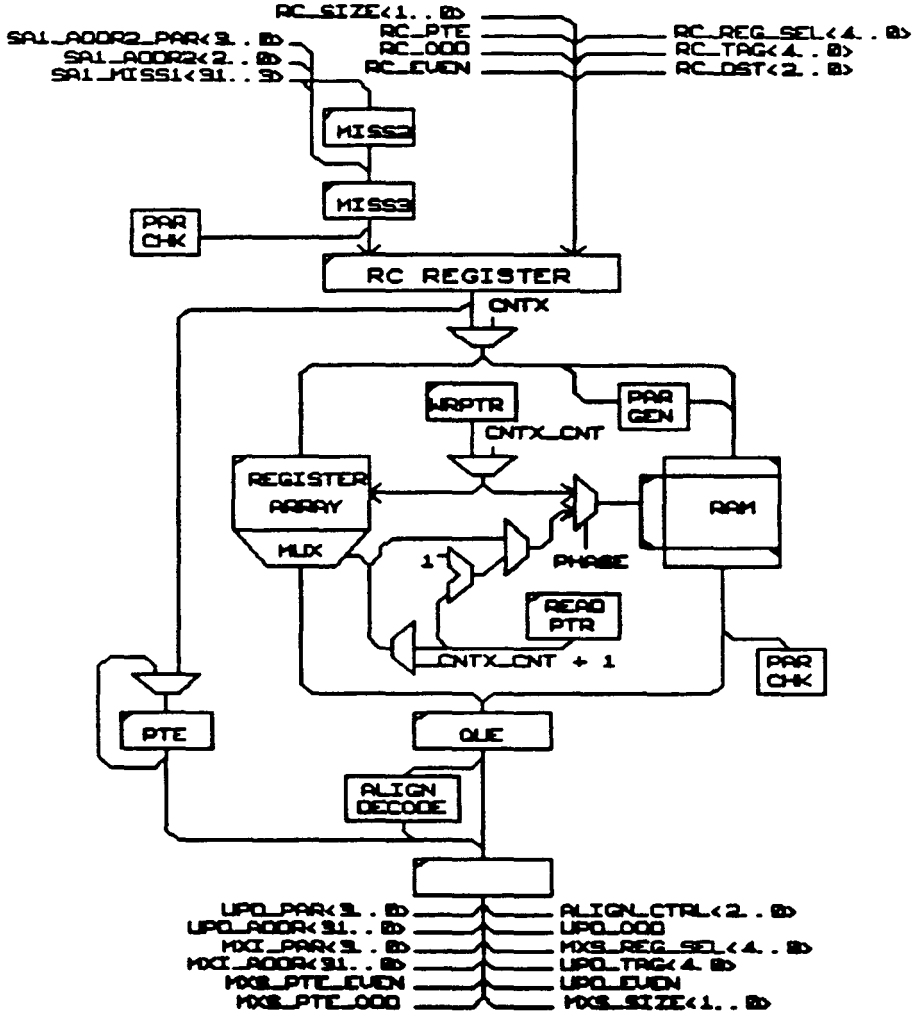
The SA1_MISS1 logical address is used for data cache updates and for instruction look ahead requests. As mentioned in 2.2.6, SA1_MISS1 is transferred to the NRC gate arrays two cycles earlier than the RC information, and the munged least significant three bits of that address plus parity are transferred one cycle early. The NRC stages the SA1 data as shown in Figure 6-2 to enter the normal control flow of the system at the same time as the RC fields with the normal RC_RDY and RC_REQ_NEXT handshakes. Note that parity is checked on the SA1 address at the SA1_MISS3 level in the manner discussed above.

For a non-PTE requests in normal mode, valid request information at the R.RC level will be queued and saved until the corresponding data is returned from memory. Of the control information, only the lower eight bits of the SA1 address field will be stored in the RAM; the remainder will be saved in the register array as mentioned above. The information is saved at the location pointed to by R.WRPTR in both the RAM and Register arrays and R.WRPTR will be incremented to prepare for the next write.

If a request at the RC level is a PCE request, no additional memory requests will be made until the NDC has received the PCE data. The control information for a PCE request is not placed on

the queue, but flows through to set up the proper data paths to insure that when the PCE returns from memory it may be set straight to the NDC. PCEs require no alignment and always are directed immediately to the NDC, requiring a minimum of control setup and data manipulation.

Figure 6-2 NRC Control Flow



Requested data returns from memory via XRE_SP.RD_DATA<31..0> and XRO_SP.RD_DATA<31..0> with corresponding parity (XRE_SP.RD_PAR<3..0> XRO_SP.RD_PAR<3..0>) and handshakes to indicate that the data is returning on this cycle, XRE_SP.RD_RDY and XRO_SP.RD_RDY as described in section 2.1.2. If the NRC was waiting for that data, i.e. the data RAM queues are empty, the control to return the data to the proper destination will be currently read from the control queues. On the next clock, the control will be progress to a series of registers, R.QUE_ADDR, R.QUE_UPD_EVEN, R.QUE_UPD_ODD, R.QUE_SIZE, R.QUE_UPD_TAG, R.QUE_REG_SEL, R.QUE_DEST and R.QUE_2T_FIRST. Each array will send R.QUE_DEST off chip, and each array accepts the output of all three arrays to put together the destination code for the data, DEST_IN<2..0> which is detailed in Table 6-1. Gate Array 0 uses R.QUE_ADDR<2..0>, R.QUE_2T_FIRST, and R.QUE_REG_SEL<0> (which contains the most significant bit of the size) to determine the proper alignment of the data. Note that the data uses the alignment control only if the destination is the NVP, but the alignment will always be calculated. The three bits of alignment control translate to the rotation indicated in

Table 6-1.

Table 6-4 Destination code between NRC arrays

DEST_IN	Destination
000	Instruction Processor (NIP)
001	Vector Processor (NVP)
010	Data Cache (NDC)
011	Scalar Unit (NAS)
1xx	Scalar Unit and Data Cache

When the destination code is complete, the NRC sets up the correct handshakes to registers to prepare for the transfer of data to a particular destination. The UPD handshakes to the NAS and NDC are described in section 2.2.7. The transfers to the NIP include only MXI_RDY, but no handshake from the NIP since it must always be ready to receive data. Transfers to the NVP are somewhat more complex. There are two sources of data to be transferred to the NVP: YBUS_DATA and data from the memories. The former are referred to as SXV requests and have priority, the latter as MXV requests and both handshakes are described in 2.6.1.

Table 6-5 Rotation as specified by ALIGN_CTRL

ALIGN_CTRL	Byte Rotation
0	0, 1, 2, 3, 4, 5, 6, 7
1	1, 2, 3, 4, 5, 6, 7, 0
2	2, 3, 4, 5, 6, 7, 0, 1
3	3, 4, 5, 6, 7, 0, 1, 2
4	4, 5, 6, 7, 0, 1, 2, 3
5	5, 6, 7, 0, 1, 2, 3, 4
6	6, 7, 0, 1, 2, 3, 4, 5
7	7, 0, 1, 2, 3, 4, 5, 6

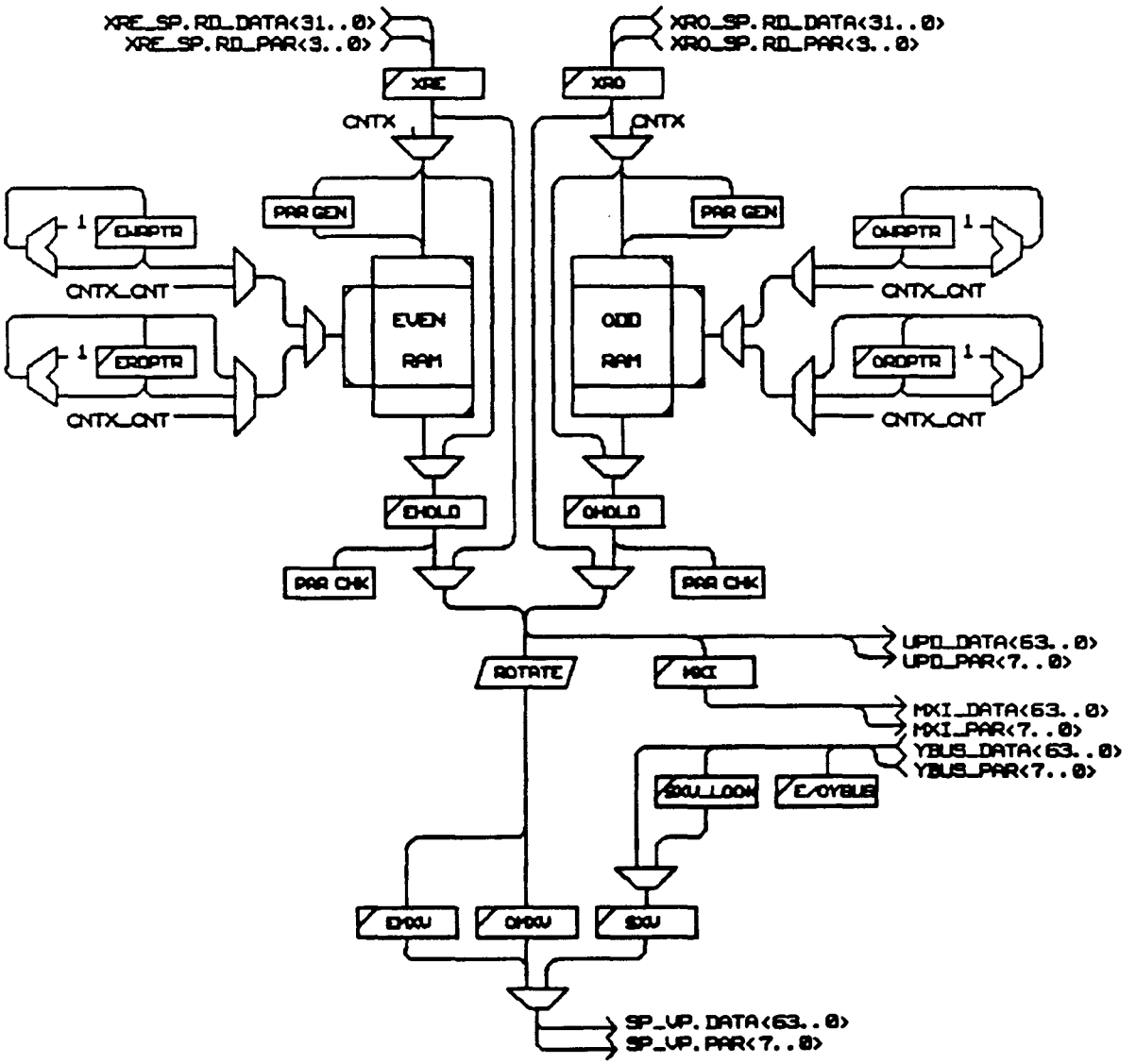
If the request has neither RC_EVEN nor RC_ODD asserted, it will be using data from the previous request which will be held in R.OHOLD or R.EHOLD. This feature may be used if, for example the NVP is doing byte operations for which it could receive four data transfers for only one word request actually made to memory, using the alignment control to select the proper byte to align for transfer. Also note that an unaligned long word vector request will require more than two words of data, as will be indicated by the assertion of RC_2T_FIRST which shows that two consecutive memory requests will be used to generate one single long word of data to be transferred to the NVP. In this case, the first long word of data will be accessed and rotated to select the data from it which can be used and this data will be saved. Then the unused most significant part of that data will be held in R.OHOLD or R.EHOLD while the next consecutive word is accessed from the queue and the proper alignment controls will enable the NRC to use these two words to generate the final, properly aligned word. The location of R.EHOLD and R.OHOLD in the data path is shown in Figure 6-3 below.

6.4 NRC Data

The NRC receives all data from the system memory for the scalar and vector processors. The

data is received in 32-bit words plus four bits of parity from the even and/or odd sides of memory, with separate handshakes and data paths for each side. Due to timing constraints, the data enters the NRC block and is immediately registered, as shown in Figure 6-3 below. Then during normal operation, R.XRE_DATA and R.XRO_DATA enter the parity checking mechanism described above and will either be written into the RAM queues, passed through to the holding registers, or be passed immediately to the NDC. If the data returning from memory is not a Page Table Entry (PTE) request and there is valid data in the queue, the data will be written into the queue and wait to be dispatched in the normal First In First Out order. If the data returning is not a PTE and the queues are empty, the data will both bypass and be written into the queue. The queue is bypassed to eliminate an idle cycle while the data is written then read from the queue, but it is still written into the queue simply for ease in implementation. This data will go straight from R.XRE_DATA or R.XRO_DATA to R.EHOLD or R.OHOLD. If the data is a Page Table Entry, when the data returns it is accelerated past any data in the queue, going from R.XRE_DATA or R.XRO_DATA through a single level of mux and out of the NRC as UPD_DATA, avoiding the R.EHOLD level of registering altogether.

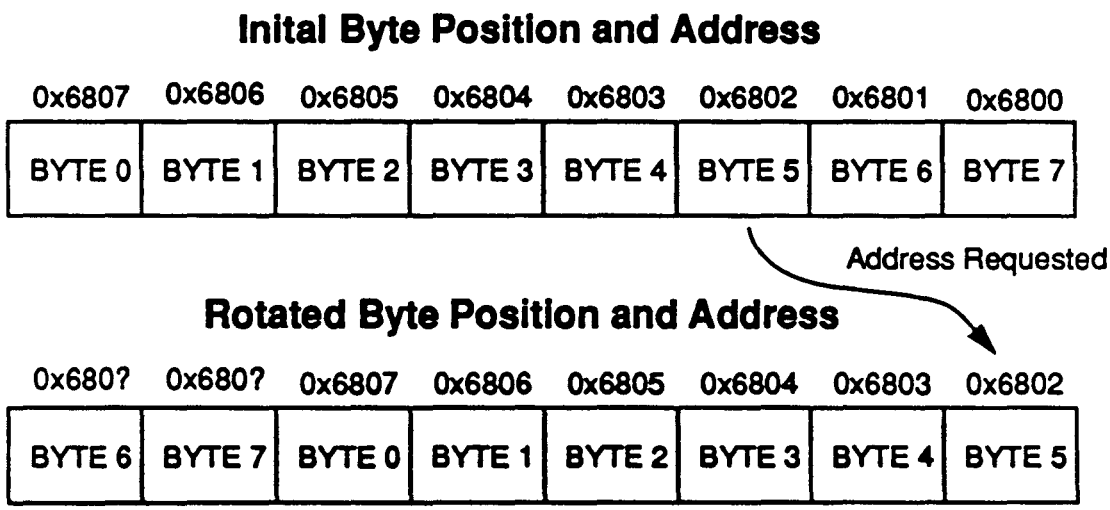
Figure 6-3 NRC Data Flow



Once the data is at the R.EHOLD or R.OHOLD level, it may be passed off as normal rather than accelerated UPD_DATA, may be registered again and sent to the NIP as MXI_DATA, or may go through further staging and manipulation before being passed off to the NVP. UPD data has the standard RDY/REQ_NEXT handshakes, but the NIP must always be able to accept requested data so an MXI request is simply announced by an MXI_RDY on the preceding cycle.

Requests for data for the NVP may be at any alignment and may be longword, word, halfword or byte lengths. Data to be returned to the NVP must be aligned so that the least significant byte requested is returned as the least significant byte, but data will be returned from system memory in aligned words. Aligned words are such that the even data always contains bytes 0 through 3 and the odd data contains bytes 4 through 7 where bytes 3 and 7 are the least significant. For example, if even and odd requests were made for address 0x6800, the odd data would contain data from addresses 0x6807 through 0x6804 and the even data would contain data from 0x6803 through 0x6800. The NVP may request data at 0x6802 to be the least significant byte of data it receives, which requires that the NRC align the data so that the data from 0x6802 is now the least significant byte, 0x6803 is next least significant, and so on, as shown in Figure 6-4. Note that the bits rotated in to fill the upper bits after the rotation are shown with address 0x680?. This is because the NVP may make one or many requests for the data with this alignment, and if the request is for many consecutive words of data with this alignment the addresses would be 0x6808 and 0x6809.

Figure 6-4 An example of rotation of data for the NVP



Once the data has been rotated according to the size and address(es) requested by the NVP, the data is registered in either R.EMXV_DATA or R.OMXV_DATA, depending on if the request and alignment provided even, odd, or both words of data. When the appropriate combination of R.EMXV_DATA or R.OMVX_DATA is available, it may be handed off to the NVP via SP_VP.DATA and SP_VP.PAR if there is no SXV data transfer pending. SXV requests transfer data from within the NSP to the NVP and have priority over transfers of memory data. Data for the SXV requests is taken from YBUS_DATA and YBUS_PAR when UIR2_VAL and UIR2_SXV_REQ are both set, which indicates that the NSP has internal data to be transferred to the NVP. The YBUS data will be registered in R.EYBUS_DATA and R.OBYUS_DATA to check parity and R.ESXV_LOOK_DATA and R.OSXV_LOOK_DATA, the look aside registers or R.ESXV_DATA_OUT and R.OSXV_DATA_OUT, the output stage registers. The data only enters

the look aside registers if the output stage registers are full and the NVP is unable to accept SXV transfers, indicated by the deassertion of SXV_REQ_NEXT. Note the SXV requests as well as MXV requests, the transfer of memory data to the NVP, both use the Neptune standard RDY/ REQ_NEXT handshakes and are described in section 2.1.6. From this last stage of SXV and MXV registers, the data passes through a single level of muxing which selects SXV or MXV data and is transferred to the NVP

6.5 Context Save and Restore

Much of the information held by the NRC must be saved and replaced during context saves and restores. All information held in the RAM and register arrays, as well as the read and write addresses and internal state and control must be saved to assure a proper restart after a context restore. Data is saved on 30 bits from the NRC_CNTX bus, ten from each array, and restored via YBUS_DATA.

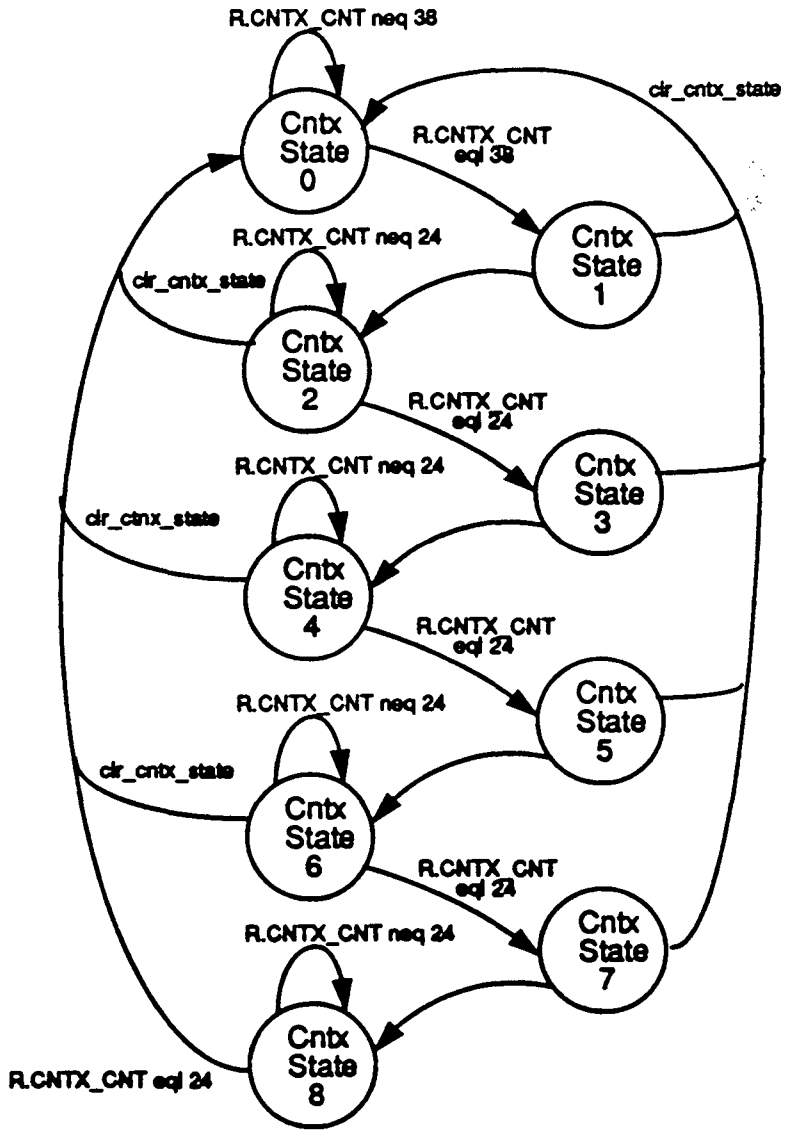
NRC context saves and restores are both controlled by the same state machine, shown in Figure 6-5. If R.CNTX_CTRL is a 2, the machine will execute a context save, and a 3 will provide a context restore. During normal operation, the state machine should be in context state 0 or R.CNTX_STATE containing a value of zero. NRC_CNTX_CTRL is the reset and context control for the NRC, and is registered as R.CNTX_CTRL which is a 2 for a context save or a 3 which is for a context load. R.CNTX_CTRL equal to a 2 or 3 will cause R.CNTX_CNT to increment, but as long as R.CNTX_CNT is less than 38, the state machine will stay in state 0 which causes the NRC to place only scan ring data on the NRC_CNTX bus. During context saves and restores, the parts of the scan ring which are context are reconfigured on each array into ten short scan rings of 39 bits which includes the registers which make up the register array of 25 rows of 13 registers each. The reconfigured scan rings are circular and connect on one end to a bit of the NRC_CNTX bus, which allows each ring to retain data while dumping it onto the bus, although the data is likely to be replaced during the next context restore. The 39 bits of the rings will be dumped to the NRC_CNTX bus during context State 0. All other states involve context save or restore from the RAM arrays.

For normal operation, the eight RAM blocks are divided into two sets of three for the even and odd data RAMs and a set of the remaining two which is part of the control storage. During context save and restore operations, these RAM blocks are regrouped into four sets of two, taking the most significant block from each of the sets of three for data and combining those two to get the fourth set of two. Context save from the ram section actually saves only the four bits of data that were written into the RAM, since to restore the context will require writing the data back into the same section, which requires generating the parity for that RAM block again. Thus only eight bits of data will be saved while ten bits of NRC_CNTX are available to each array. The extra two bits are padded with register rings which function as explained in the preceding paragraph. Context saves or restores of the RAM blocks save data from only the 25 levels of the RAMs which are actually used, even though 256 levels are available.

After saving or restoring context from the register array, State 0 flows into State 1 followed by State 2 as shown in Figure 6-5, which begins the context save or restore of the RAM arrays starting with the pair used to save part of the control information. To save the 25 levels of the RAM which are used, State 2 is valid until R.CNTX_CNT equals 24 so that the saves or restores from State 1 and State 2 allow a total of 25 reads or writes. Then R.CNTX_CNT is reset and the state machine moves into States 3 and 4, which saves or restores the special pair of RAM blocks made up of the most significant block from each of the even and odd memory banks. When that section has been saved or restored, States 5 and 6 will do the same to the lower two RAM blocks of the even data

bank, then States 7 and 8 will repeat it for the lower two RAM blocks of the odd data bank. When R.CNTX_CNT equals 24 and the state machine is in State 8, the state machine will move back into State 0. If the least significant two context control lines return to a value of 0 or 3 or the most significant bit is set it will set C.CLR_CNTX_STATE which causes the state machine to return to State 0 and R.CNTX_CNT will be reset regardless of what is being done at the time.

Figure 6-5 NRC Context State Machine



7 Microcode

There are two microcode files used within the NSP. These are the US microcode, which contains the control instructions, and the SR microcode, which contains initial values for the scratch RAM. The purpose of this chapter is to provide enough explanation so that the microcode listings can be read and understood. A definition of the microword is presented, giving the microinstruction field names and defining their encodings. Constructs of the microlanguage are presented along with descriptions of how they control the hardware. A list of microprogramming rules not covered in the language explanations is also provided.

7.1 The US Microinstruction

The NAS subsystem of the NSP is directed by a stream of instruction dispatches from the NIP subsystem. Part of this dispatch information is the entrypoint, which is used to form the address in US control store of the first microinstruction to be executed for the dispatched macroinstruction. The complete address is formed by adding the entrypoint and 0xC00. The fields within the US microinstruction are shown below, along with the meaning of each encoding. Any values not listed are reserved for future use. In some cases (such as register selects), the same encodings are used for multiple fields - in this case the encodings are only listed once. Some microlanguage constructs map directly to a single field encoding. For these fields, the mnemonic for the encodings are given with the description. Other field encodings arise due to the presence of multiple microlanguage constructs. In this case, there is no mnemonic corresponding to the field encoding, so only the numerical field encoding can be supplied. The default encodings (i.e. what value the field holds if no microlanguage construct explicitly sets it to another value) for each field may be found in the microcode listing.

The US microinstruction fields are listed below, in alphabetical order.

Field Name	Width	Pos.	Description
ABUS_FMT	2	57-56	A bus write format select. Multiplexes between read port selects to use as A port write register select. 0 - XREG_SEL 1 - YREG_SEL 2 - ZREG_SEL
ABUS_SIZE	2	133-132	A bus write size 0 - byte 1 - halfword 2 - word 3 - longword
ABUS_WR_EN	1	134	A port write enable. When set to 1, the register selected by ABUS_FMT is written with size ABUS_SIZE.
AG_SEL	4	140-137	Address generator address source selection

0 - advanced effa, i.e. use next instruction's dispatched register select
1 - Z port data from register file
2 - hold, i.e. use registered last address
3 - last registered address plus 4
4 - last registered address plus 8
5 - Z port data from register file minus 4
6 - Z port data from register file minus 8
7 - last registered address minus 4
8 - last registered address minus 8
9 - last registered address plus 1

AGZ_HAZEN	1	43	Address generator register file access hazard check enable. Implies that the microinstruction intends to use the last cycles' Z port access for an address calculation.
ALUFAST	1	63	Qualifier for ALUOP - when set to 1, denotes that the operation makes timing to bypass off-chip from the NRFA A bus to the X, Y, or Z read port. When 0, no bypass is possible so the result on the A bus must be written to the operand registers before going off-chip.
ALUOP	5	113-109	Integer ALU operation code - *cvtx.y* operations follow the convert instruction descriptions in the Convex Architecture Reference.

00 - add; A = X + Y
01 - add with carry; A = X + Y + IC
02 - convert byte to word; A = cvtb.w(X)
03 - convert halfword to word; A = cvth.w(X)
04 - convert word to longword; A = cvtw.l(X)
05 - increment by 4; A = X + 4
06 - increment by 8; A = X + 8
07 - increment by 16; A = X + 16
08 - reverse subtract; A = Y - X
09 - reverse subtract with carry; A = Y - X - IC
0A - subtract; A = X - Y
0B - subtract with carry; A = X - Y - IC
0D - decrement by 4; A = X - 4
0E - decrement by 8; A = X - 8
0F - decrement by 16; A = X - 16
10 - pass zeros; A = 00000000
11 - convert word to byte; A = cvtw.b(X)
12 - convert word to halfword; A = cvtw.h(X)
13 - convert longword to word; A = cvtl.w(X)
14 - logical AND-complement; A = $\overline{X}$ AND Y
15 - logical complement; A = $\overline{X}$

16 - logical exclusive-OR; A = X XOR Y
17 - logical NAND; A = $\overline{(X \text{ AND } Y)}$
18 - logical AND; A = X AND Y
19 - ring wrap; if X<31> = 1 then A = 100 concatenated with X<28:0> otherwise A = X
1A - pass X; A = X
1B - stripmine; if X < 0, A = 0, if X > 0x80 A= 0x80, otherwise A = X
1C - pass Y; A = Y
1D - short shift; A = Y shifted left by X<2..0>
1E - logical OR; A = X OR Y
1F - pass ones; A = all ones

ALUSIZE	2	115-114	Size control supplied with ALUOP 0 - byte 1 - halfword 2 - word 3 - longword
BDOP	2	136-135	Backdoor reservation operation. Used to set hazards in conjunction with BDREG_FMT and BDSIZE. 0 - no operation 1 - set hazard 3 - clear hazard
BDREG_FMT	2	59-58	Backdoor register format select. Multiplexes between read port selects to use as backdoor register select for hazard setting and destination select for B and C bus write operations 1 - XREG_SEL 2 - YREG_SEL 3 - ZREG_SEL
BDSIZE	2	108-107	Back door operation size, for hazard setting, B bus writes, and C bus writes 0 - byte 1 - halfword 2 - word 3 - longword
BRADDR	12	11-0	Microsequencer branch address, used for jumps, calls, etc.
BRTYPE	4	15-12	Microsequencer branch type, i.e. command for sequencer to determine next address 0 - unconditional jump: next address = BRADDR

1 - unconditional call: push next sequential address, next address = BRADDR
2 - unconditional return: next address = top of microstack
3 - accept dispatch: next address = dispatched entrypoint + 0xC00
4 - pop microinterrupt stack: next address = next sequential address, throw away top of microinterrupt stack
6 - return from microinterrupt: next address = top of microinterrupt stack
8 - conditional jump: if test true jump (0), else next address = next sequential address
9 - conditional call: if test true call (1), else next address = next sequential address
A - conditional return: if test true return (2), else next address = next sequential address
B - conditional dispatch: if test true dispatch (3), else next address = next sequential address

CRY_DST 3 82-80 PSW (AC,SC) and USW (IC,JC,KC) carry bit load destination and hazard set select

0 - no destination
1 - PSW<AC>
2 - PSW<SC>
3 - USW<IC>
4 - USW<JC>
5 - USW<KC> including hazard detection
6 - USW<KC> ignoring hazards

CRY_OP 2 84-83 PSW (AC,SC) and USW (IC,JC,KC) carry bit control opcode

0 - No operation
1 - set hazard
2 - load from source specified by CRY_SRC

CRY_SRC 3 87-85 PSW (AC,SC) and USW (IC,JC,KC) carry bit load source select

0 - integer ALU carry out
1 - logical complement of integer ALU carry out
2 - integer ALU compare for equal status flag output
3 - integer ALU compare for less than status flag output
4 - integer ALU compare for less than or equal to status output

			5 - float add function unit return status 6 - communication register operation return status
CSDISP	1	42	Control Store dispatch - set to 1 if this microinstuction is the last of the macroinstruction - essentially a fast decode of the DISP and CDISP decodes of the BRTYPE field.
CX_OP_REQ	1	65	Qualifies MFP_OP as a context scan operation (as opposed to memory, PTE, or function unit operation)
EXTX_SEL	3	68-66	External X mux select 0 - pre-X mux output from NPSW array 1 - scratch RAM 2 - context set 1 3 - zero-extended return control (NRC) context 4 - PTE miss address (SA1_MISS1) 5 - zero-extended, right justified trap vector from NCU 6 - UIR1 level microconstant 7 - trap vector from NCU with 12 zeros appended on the right to format as a PTE purge address
FAKE_RING0	1	150	When set, allows memory or communication register access in the microinstruction to receive ring 0 access privileges even if the current ring is 1-4.
FIRSTU	1	131	First microinstruction - set to 1 if this microinstruction is the first of the macroinstruction
FU_OP_REQ	1	116	Qualifies MFP_OP as a function unit operation (as opposed to a memory, PTE, or context scan operation)
IPINH	2	70-69	IP lookahead request inhibit control 0 - no operation 1 - clear 2 - clear 3 - set
JMP_RESTART	2	72-71	IP jump restart control. 0 - no operation 1 - microinterrupt restart 2 - cross-ring restart 3 - intra-ring restart (i.e. ring bits don't change)
LC_CTL	2	74-73	Microsequencer loop counter control

0 - hold; i.e. no operation
1 - load from UIR2 level microconstant
2 - decrement by 1

MEM_OP_REQ	1	149	Qualfies that MFP_OP is a memory request (as opposed to function unit, PTE, or scan operation)
MFP_OP	11	106-96	Memory, function unit, PTE, and context scan control opcode. Function qualified by MEM_OP_REQ, FU_OP_REQ, PTE_OP_REQ, or CX_OP_REQ. The 11-bit field is subdivided in different ways depending on the type of operation.

If MEM_OP_REQ = 1 or PTE_OP_REQ = 1

MEM_OP_TYPE	6	106-101	Operation type (binary, x = don't care bit)

If MEM_OP_REQ = 1 & MEM_OP_AT_TYPE = 0:
- 0x0000 - no operation
- x10000 - reset block prefetch
- x10100 - purge referenced bit RAM
- x11000 - purge PTE thread (unshared)validity
- x11100 - purge data cache thread (unshared) validity
- 1x0000 - purge PTE entry
- 1x0100 - purge modified bit RAM
- 1x1000 - purge PTE non-thread (shared) validity
- 1x1100 - purge data cache non-thread (shared) validity

Note: all purges reset block prefetch

If MEM_OP_REQ = 1 & MEM_OP_AT_TYPE != 1:
- xxxx01 - read
- xxxx10 - write
- xxxx11 - test and modify
- xxx1xx - start vector address generator
- xx11xx - start vector address generator for store scalar extended
- x1x1xx - start vector address generator for operation under mask
- 1xx1xx - start vector address generator for vector indexed operation

If MEM_OP_REQ = 1 & MEM_OP_AT_TYPE = 1:
- xxxxx0 - no return data
- xxxxx1 - return data

If PTE_OP_REQ = 1
- 000xxx - write PTE data, tag, and validity RAMs for PTE2 (level T bit clear)

001xxx - write PTE data, tag, and validity RAMs for PTET (level T bit set)
101xxx - PTE1 translation and fetch
110xxx - PTE2 translation and fetch
111xxx - PTET translation and fetch

MEM_OP_AT_TYPE	2	100-99	Address translation type

0 - no address translation (physical addressing)
1 - communication register address translation
2 - virtual address translation, write to data cache if hit occurs
3 - virtual address translation, read and write data cache

MEM_OP_DST	3	98-96	Request destination

0 - instruction processor , allow faults
1 - instruction processor, don't allow faults
2 - vector processor
3 - data cache
4 - scalar register file
5 - scalar register file and data cache, do not initiate block prefetch if data cache miss
7 - scalar register file and data cache, initiate block prefetch if data cache miss

If CX_OP_REQ = 1:

MEM_OP_AT_TYPE	2	100-99	NAS context operation (yes, poorly named field)

1 - hold: NAS context rings hold
2 - reset: NAS context resets
3 - left: NAS context rings shift left

MEM_OP_DST	3	98-96	NRC context operation (yes, poorly named field)

1 - hold: NRC context rings hold
2 - save: NRC context rings shift out for save
3 - restore: NRC context rings shift in for restore
4 - reset: NRC context resets

If FU_OP_REQ = 1:

FU_OP	8	106-99	Function unit (NFAD, NMUL, NDIV, NMISC) opcode - operand/result size specfied by BDSIZE. Some source operand sizes specified by BDSIZE as noted.

00 - float add: B = X + Y (NFAD)
08 - float subtract: B = Y - X (NFAD)
10 - float subtract: B = X - Y (NFAD)

1C - float compare: Y > X (NFAD)
19 - float compare: Y = X (NFAD)
1D - float compare: Y >= X (NFAD)
1A - float compare: Y < X (NFAD)
1B - float compare: Y <= X (NFAD)
60 - integer address multiply - overflow sets AIV (NMUL)
40 - integer scalar multiply - overflow sets SIV (NMUL)
41 - float multiply (NMUL)
A0 - integer address divide: B = Y / X - overflow sets AIV (NDIV)
80 - integer scalar divide: B = Y / X - overflow sets SIV (NDIV)
81 - float divide: B = Y / X (NDIV)
83 - float square root: B = sqrt(Y) (NDIV)
C0 - leading ones position: B = zero-based index of leftmost 1 bit in Y - size of Y controlled by BDSIZE (NMISC)
C1 - trailing zero count: B = number of trailing zeros in Y - size of Y controlled by BDSIZE (NMISC)
C3 - logical shift: B = Y shifted left by X bits (NMISC)
C4 - population count: B = number of ones in 64 bits of Y (NMISC)
C5 - population count: B = number of ones in X bits of Y (NMISC)
C6 - population count: B = number of zeros in X bits of Y (NMISC)
D7 - integerize float: B = Y with fractional part of mantissa removed (NMISC)
D8 - convert byte to float: B = cvt(Y) (NMISC)
C8 - convert byte to integert: B = cvt(Y) (NMISC)
D9 - convert halfword to float: B = cvt(Y) (NMISC)
C9 - convert halfword to integer: B = cvt(Y) (NMISC)
DA - convert word to float: B = cvt(Y) (NMISC)
CA - convert word to integer: B = cvt(Y) (NMISC)
DB - convert longword to float: B = cvt(Y) (NMISC)
CB - convert longword to integer: B = cvt(Y) (NMISC)
DE - convert single float to float: B = cvt(Y) (NMISC)
CE - convert single float to integer: B = cvt(Y) (NMISC)
DF - convert double float to float: B = cvt(Y)

(NMISC)
CF - convert double float to integer: B = cvt(Y) (NMISC)

NAG0_PAR	1	142	Odd parity over bits sent to NAG array in position 0
NAG1_PAR	1	141	Odd parity over bits sent to NAG array in position 1
NDC_PAR	1	148	Odd parity over bits sent to NDC array
NPSW_PAR	1	88	Odd parity over bits sent to NPSW array
NRFA_PAR	1	64	Odd parity over bits sent to each NRFA array
NUS_PAR	1	75	Odd parity over bits sent to NUS array
PREX_SEL	3	91-89	Mux select for pre-X mux in NPSW array 0 - 0xCACACACA (garbage) 1 - timer - 24-bit microsecond timer zero-extended to 32 bits 2 - PSW 3 - NPC - next program counter at UIR1 level (CPC with branch displacement, size, and extend added in) 4 - CPC - current program counter at UIR1 level (from NIP, without branch displacement) 5 - XPC - execute program counter at UIR1 level (CPC with branch displacement added in) 6 - CCR - CPU control register
PSW_OP	3	94-92	PSW load/clear/set opcode 0 - No operation 1 - load PSW<SIV> with ALU overflow 2 - load PSW<SIV> with ALU overflow 3 - clears AC, AIV, ADZ, FRL, SC, SIV, SDZ, UN, OV, RO, FDZ, and FIN. Other PSW bits left unmodified 4 - PSW<FRL> = 01. Other PSW bits left unmodified. 5 - PSW<FRL> = 10. Other PSW bits left unmodified. 6 - PSW<FRL> = 11. Other PSW bits left unmodified. 7 - PSW loaded from Y bus data
PTE_OP_REQ	1	144	Qualifies that MFP_OP is a PTE read request (as opposed to function unit, memory, or context scan operation)

REQ_RETRY	1	143	Request retry control used to restart request after a PTE miss or fault has been fixed
SR_ADDR_SEL	2	29-28	Scratch RAM address source mux control 0 - zero-extended UPC level microconstant 1 - scratch RAM address counter 2 - zero-extended segment bits of PTE miss address 3 - zero-extended PTE1 offset bits of PTE miss address
SR_CNT_SEL	2	27-26	Scratch RAM address counter control 0 - hold; i.e. no operation 1 - load from UPC level microconstant 2 - increment by 1
SXV_REQ	1	147	Scalar to vector transfer request control
TPOL	1	35	Microsequencer test condition polarity 0 - branch if false (selected test = 0) 1 - branch if true (selected test = 1)
TSEL	5	34-30	Microsequencer test condition select (all values) and wait hazard selection (10-17 only) Test selection: 00 - always (1) 01 - PSW<AC> 02 - PSW<SC> 03 - USW<IC> 04 - USW<JC> 05 - USW<KC> with test hazard checking 06 - PSW<FRL> = 00 07 - PSW<FRL> = 10 08 - PSW<FRL> = 11 0A - vector valid, i.e. CCR<VV> 0B - executing in ring 0 0C - PSW<TR> OR PSW<SEQ> = 1 0D - PSW<TTC> 0E - PSW<TIT> 10 - 0 (used as wait hazard select) 11 - 0 (used as wait hazard select) 12 - 0 (used as wait hazard select) 13 - 0 (used as wait hazard select) 14 - 0 (used as wait hazard select) 15 - 0 (used as wait hazard select)

16 - 0 (used as wait hazard select)
17 - 0 (used as wait hazard select)
0F - USW<KC> with no test hazard checking
18 - R.SPARE_TEST in NUS
19 - R.SCANNED_TCOND<2> in NUS
1A - R.SCANNED_TCOND<1> in NUS
1B - R.SCANNED_TCOND<0> in NUS
1C - microtrap complete
1D - level T bit in read PTE (R.CBUS_DATA<7>)
1E - scratch RAM valid
1F - loop count negative (R.LOOP_CNT<9>)

Wait hazard selection:

10 - PSW hazard
11 - store pending
12 - data cache not idle
13 - memory request pending
14 - memory system not idle
15 - vector processor not idle
16 - function units busy
17 - request pending in crossbar

UCONST	10	25-16	General purpose microconstant
USEQ_RAND	3	79-77	Miscellaneous "random function" control opcode

0 - no operation
1 - assert SP_XCL_TRAP_COMP, issuing trap complete to the NCU
2 - enable assertion of SP_XCL_DEADLOCK instruction at UIR2 level is a deadlock detection instruction
3 - purge scratch RAM validity
4 - clear microsequencer's microtrap complete flag
5 - qualify IP jump restart with prevention of lookahead restart

VP_DISP	1	146	Dispatches vector processor to entrypoint supplied by NIP
VP_PSW_HAZ	1	95	Vector processor PSW hazard. Asserted with vector processor dispatch to claim a hazard on the PSW for the dispatched instruction
VXS_REQ	1	145	Vector to scalar transfer request control
WR_SR	1	76	Scratch RAM write enable

XMUX_SEL 2 130-129 X mux data path select

- 0 - register file
- 1 - displacement (immediate) from NIP
- 2 - external XMUX
- 3 - zero

XREG_FMT 2 128-127 X read port format control. Determines how selected data is presented to X read port output, if at all.

- 0 - register file
- 1 - effective address, i.e. zeros if Rj = 0, else register file
- 2 - Rj field as immediate, for short shift, etc.

XREG_HAZ 3 125-123 X read port hazard control. Determines what hazard checking is done for register selected by XREG_SEL.

- 0 - byte access
- 2 - halfword access
- 3 - word access
- 4 - effective address access, i.e. word if Rj nonzero, no hazard if Rj = 0
- 5 - longword access
- 7 - no hazard check

XREG_SEL 6 49-44 Register select for X port of register file

00: RJA	08: RKA	10: RJS	14: RJSU
18: RKS	1C: RKSU		
20: A0 (SP)	21: A1	22: A2	23: A3
24: A4	25: A5	26: A6 (AP)	27: A7(FP)
28: T0	29: T1	2A: T2	2B: T3
2C:T4	2D: T5	2E: T6	2F: T7
30: S0	31: S0U	32: S1	33: S1U
34: S2	35: S2U	36: S3	37: S3U
38: S4	39: S4U	3A: S5	3B: S5U
3C: S6	3D: S6U	3E: S7	3F: S7U

The RJ/RK notation implies the use of the register selects from the NIP dispatch. The U means the upper word of a 64-bit register. The RKA (as opposed to RKS) means address as opposed to scalar register - A registers use 1 32-bit location while S registers use 2 32-bit locations.

YREG_FMT 1 126 Y read port format control. Determines how selected data is presented to Y read port output, if at all.

- 0 - register file

			1 - effective address, i.e. zeros if Rj = 0, else register file
YREG_HAZ	3	122-120	Y read port hazard control. Determines what hazard checking is done for register selected by YREG_SEL Encoded same as XREG_HAZ.
YREG_SEL	6	55-50	Register select for Y port of register file - encoded same as XREG_SEL
YZ_WR_DST	3	62-60	Write destination control for Y bus and Z mux receivers 0 - no destination 2 - load CCR from Y bus <31:0> 3 - load microsecond timer from Y bus <23:0> 4 - load CIR (NDC copy) from Y bus <4:0> 5 - load TID (NDC copy) from Y bus <4:0> 6 - load VL (NDC copy) from Y bus <7:0> 7 - load VS (NDC copy) from Z mux <31:0>
ZREG_HAZ	3	119-117	Z read port hazard control. Determines what hazard checking is done for register selected by ZREG_SEL. Encoded same as XREG_HAZ
ZREG_SEL	6	41-36	Register select for Z port of register file - encoded same as XREG_SEL

7.2 The US Microlanguage

This section describes the constructs of the US microlanguage, detailing which microinstruction fields are coded for each construct, with a brief description of how the construct controls NSP hardware. Many constructs are simply a mnemonic for a single encoding of a single field. For example, the construct **WR_VL** simply encodes the **YZ_WR_DST** field with the value 6, which causes the NDC's copy of vector length to be written. More complex constructs may define the encoding of many different fields simultaneously. The microlanguage was designed to allow easy coding of the microinstruction fields and provide a readable microprogram, while still maintaining a close relationship to the microinstruction fields. Readers familiar with the C2 ASP microlanguage will find the US microlanguage very similar in basic structure.

For each construct, the field values coded are listed to the right of the construct. Note that in some cases, the actual encoding of the microinstruction field is put off until the entire microinstruction is parsed. For example, encoding hazard selects is done in this way to specify the largest size hazard implied by the microinstruction. If one microp specifies a byte hazard check (for example a byte read access from the register file) and a second microp in the same microinstruction specifies a word hazard check (for example a function unit result reservation), the microinstruction will contain a word hazard select. In the construct definitions following, this is not noted - the byte register file access construct states that it encodes a byte hazard check, while the word function unit reservation construct states that it encodes a word hazard check. The microassembler performs consistency checks at the completion of each line parse. In some cases this results in a compatible field assignment, while in other cases the conflict cannot be resolved and an error is reported.

In most cases, a description of the action of the construct is given with it. Constructs with multiple parenthesized listings indicate optional parameters for the micro-operation.

7.2.1 General Notational Conventions

There are some rules and conventions which apply to all microlanguage constructs. A language construct may be referred to as a *microp*, short for micro-operation. This is the name the CONVEX microassembler gives to the initial definition of the syntax of microlanguage construct. A microp consists of a name optionally followed by a list of parameters enclosed in parentheses. Parameter identification is by position in the parameter list. For example, a construct with three parameters

```
FOO(X,Y,Z)
```

in which the first two are to be unspecified must show the third parameter preceded by two commas to indicate the two missing parameters:

```
FOO(,,Z)
```

Commas need not be shown for unspecified parameters following the rightmost specified parameter. For example, a four parameter construct with only the third parameter specified could be written in either of these forms:

```
FOO(,,Z,)
FOO(,,Z)
```

In the following subsections, all language constructs appear in upper case. Parameters for the construct are lower case.

A shorthand notation is used in the figures which detail the microlanguage constructs. If a construct has multiple parameters (e.g. FOO(X,Y)), rather than listing all possible combinations of all parameters, each parameter is enumerated with a dummy variable for the other parameters. This indicates that all other values for the other parameters may be substituted. For example, if the FOO(X,Y) construct has potential values of X1 and X2 for the X parameter and Y1, Y2, and Y3 for the Y parameter, the constructs would be listed like this:

FOO(X1,yval)
FOO(X2,yval)
FOO(xval,Y1)
FOO(xval,Y2)
FOO(xval,Y3)

where xval and yval are dummy variables indicating the presence of another parameter.

7.2.2 Register Selection Construct

The **REG** construct specifies which registers are selected to be read from or written to the register file. These three register selects control the three read ports X, Y, and Z and the A write port directly. They also control the other two write ports B and C indirectly. Indirect control over these two "backdoor" ports is gained by using one of the three register selects as an entry in the return queues that control writing of the register file through the B and C ports. The **REG** construct may optionally specify whether hazard checking is performed on the selected register(s). The format of the **REG** construct is given in Figure 7-1.

Figure 7-1 Register Selection Construct

construct	encodings
REG(xsel,ysel,zsel)	XREG_SEL = xsel
	YREG_SEL = ysel
	ZREG_SEL = zsel
REG(xsel,ysel,zsel,NHAZ)	XREG_SEL = xsel
	YREG_SEL = ysel
	ZREG_SEL = zsel
	XREG_HAZ = 0
	YREG_HAZ = 0
	ZREG_HAZ = 0

The second form is used to operate on registers without hazards considered, effectively eliminating any encodings of **XREG_HAZ**, **YREG_HAZ**, and **ZREG_HAZ** implied by the **XY** construct defined next.

7.2.3 Operand Selection Constructs

There are two constructs that control operand selection in the main scalar data path. The **XY** construct specifies the sources for the X and Y operands in the scalar data path, as well as how the operands are formatted when read.

For the X operand, the **XY** construct controls the internal X mux in the NRFA arrays as well as the external X mux that brings in scratch RAM and the NPSW's external X mux output. Recall from the functional description section that there are separate X operand paths off-chip from the NRFAs to the function units and to the internal operand registers of the integer ALU. For this reason, there is also an **XMUX** construct that can specify the X operand to the function unit. For example, it is possible to read a scratch RAM constant and use it as the X operand in the integer ALU while simultaneously reading the X port of the register file and sending it to a function unit as its X operand. The microassembler performs consistency checks to insure that the **XY** and **XMUX** constructs are encoded consistently. For example, the only source for function unit operations is the register file. The final encodings of the microinstruction due to the **XMUX** construct therefore depend on the **XY** construct, if any, in the same microinstruction.

Control of the Y operand is much simpler. The **XY** construct controls how data from the Y read port of the scalar register file is formatted on the Y mux output to the function units and the internal Y operand path in the NRFA.

The **XY** and **XMUX** constructs are detailed in Figure 7-2.

Figure 7-2 Operand Selection Constructs

construct	encodings		
XMUX(RFB)	XMUX_SEL = 0 or 2	XREG_FMT = 0	XREG_HAZ = 0
XMUX(RFH)	XMUX_SEL = 0 or 2	XREG_FMT = 0	XREG_HAZ = 2
XMUX(RFW)	XMUX_SEL = 0 or 2	XREG_FMT = 0	XREG_HAZ = 3
XMUX(RFS)	same as XMUX(RFW)		
XMUX(RFL)	XMUX_SEL = 0 or 2	XREG_FMT = 0	XREG_HAZ = 5
XMUX(RFD)	same as XMUX(RFL)		
XMUX(RF0)	XMUX_SEL = 0 or 2	XREG_FMT = 1	XREG_HAZ = 4
XMUX(RJIMM)	XMUX_SEL = 0 or 2	XREG_FMT = 2	
XMUX(IMM)	XMUX_SEL = 1		
XMUX(ZERO)	XMUX_SEL = 3		
XY(RFB,ysrc)	XMUX_SEL = 0	XREG_FMT = 0	XREG_HAZ = 0
XY(RFH,ysrc)	XMUX_SEL = 0	XREG_FMT = 0	XREG_HAZ = 2
XY(RFW,ysrc)	XMUX_SEL = 0	XREG_FMT = 0	XREG_HAZ = 3
XY(RFS,ysrc)	same as XMUX(RFW,ysrc)		
XY(RFL,ysrc)	XMUX_SEL = 0	XREG_FMT = 0	XREG_HAZ = 5
XY(RFD,ysrc)	same as XMUX(RFD,ysrc)		
XY(RF0,ysrc)	XMUX_SEL = 0	XREG_FMT = 1	XREG_HAZ = 4
XY(RJIMM,.ysrc)	XMUX_SEL = 0	XREG_FMT = 2	
XY(IMM,ysrc)	XMUX_SEL = 1		

XY(ZERO,ysrc)	XMUX_SEL = 3		
XY(TIMER,ysrc)	XMUX_SEL = 2	EXTX_SEL = 0	PREX_SEL = 1
XY(PSW,ysrc)	XMUX_SEL = 2	EXTX_SEL = 0	PREX_SEL = 2
XY(NPC,ysrc)	XMUX_SEL = 2	EXTX_SEL = 0	PREX_SEL = 3
XY(CPC,ysrc)	XMUX_SEL = 2	EXTX_SEL = 0	PREX_SEL = 4
XY(XPC,ysrc)	XMUX_SEL = 2	EXTX_SEL = 0	PREX_SEL = 5
XY(CCR,ysrc)	XMUX_SEL = 2	EXTX_SEL = 0	PREX_SEL = 6
XY(RAM(addr),ysrc)	XMUX_SEL = 2	EXTX_SEL = 1	
	UCONST = addr	SR_ADDR_SEL = 0	
XY(RAM(CNT),ysrc)	XMUX_SEL = 2	EXTX_SEL = 1	
	SR_ADDR_SEL = 1		
XY(SDR,ysrc)	XMUX_SEL = 2	EXTX_SEL = 1	
	SR_ADDR_SEL = 2		
XY(PTE1,ysrc)	XMUX_SEL = 2	EXTX_SEL = 1	
	SR_ADDR_SEL = 3		
XY(CNTX1,ysrc)	XMUX_SEL = 2	EXTX_SEL = 2	
XY(NRC_CNTX,ysrc)	XMUX_SEL = 2	EXTX_SEL = 3	
XY(SA1_MISS1,ysrc)	XMUX_SEL = 2	EXTX_SEL = 4	
XY(TRAP_VEC,ysrc)	XMUX_SEL = 2	EXTX_SEL = 5	
XY(TRAP_PATE_ADDR,ysrc)	XMUX_SEL = 2	EXTX_SEL = 7	
XY(xsrc,RFB)	YREG_FMT = 0	YREG_HAZ = 0	
XY(xsrc,RFH)	YREG_FMT = 0	YREG_HAZ = 2	
XY(xsrc,RFW)	YREG_FMT = 0	YREG_HAZ = 3	
XY(xsrc,RFL)	YREG_FMT = 0	YREG_HAZ = 5	
XY(xsrc,RF0)	YREG_FMT = 1	YREG_HAZ = 4	

7.2.4 ALU Operation Constructs

This section describes the constructs used to control the operation performed by the integer ALU. These constructs are mnemonics for encodings of the **ALUOP** and **ALUFAST** fields with a size parameter added in some cases which encodes the **ALUSIZE** field. The **ALUFAST** field controls whether the specified operation makes timing for bypass from the A result bus to the X, Y, and Z operand paths of the microinstruction upstream in the pipeline. This means that if the UIR2 level **ALUFAST** field is set, a UIR1 level access to a register being written from the A port at the UIR2 level may bypass the register file write. The **ALUSIZE** field controls the width of the ALU and determines which bits are used to generate ALU status flag output. For type conversion operations, there are two parameters - the first is source data type, the second is destination data type. The **ALUSIZE** field specifies the data type of the source operand. The data type of the destination operand is encoded in the **ALUOP** field.

The ALU operation control constructs are listed in Figure 7-3. The *cvtx.y* operations follow the descriptions in the Convex Architecture Reference.

Figure 7-3 ALU Operation Constructs

construct	description	encodings		
ADD(size)	A = X + Y	ALUOP = 00	ALUFAST = 0	ALUSIZE = size
ADDC(size)	A = X + Y + IC	ALUOP = 01	ALUFAST = 0	ALUSIZE = size
XADD4(size)	A = X + 4	ALUOP = 05	ALUFAST = 0	ALUSIZE = size
XADD8(size)	A = X + 8	ALUOP = 06	ALUFAST = 0	ALUSIZE = size
XSUB4(size)	A = X - 4	ALUOP = 0D	ALUFAST = 0	ALUSIZE = size
XSUB8(size)	A = X - 8	ALUOP = 0E	ALUFAST = 0	ALUSIZE = size
XSUB16(size)	$A = X + 16_{10}$	ALUOP = 0F	ALUFAST = 0	ALUSIZE = size
YSUBX(size)	A = Y - X	ALUOP = 08	ALUFAST = 0	ALUSIZE = size
XSUBY(size)	A = X - Y	ALUOP = 0A	ALUFAST = 0	ALUSIZE = size
YSUBXC(size)	A = Y - X - IC	ALUOP = 09	ALUFAST = 0	ALUSIZE = size
XSUBYC(size)	A = X - Y - IC	ALUOP = 0B	ALUFAST = 0	ALUSIZE = size
STRIPX	VL stripmine X if X < 0: A = 0 if X > 128_{10}: A = 128_{10} otherwise A = X	ALUOP = 1B	ALUFAST = 0	ALUSIZE = size
SSHFY	A = Y shifted left by X bits	ALUOP = 1D	ALUFAST = 0	ALUSIZE = size
PASSX(size)	A = X	ALUOP = 1A	ALUFAST = 1	ALUSIZE = size
PASSY(size)	A = Y	ALUOP = 1C	ALUFAST = 1	ALUSIZE = size
PASS1(size)	A = all ones	ALUOP = 1F	ALUFAST = 1	ALUSIZE = size
PASS0(size)	A = 0	ALUOP = 10	ALUFAST = 1	ALUSIZE = size
AND1(size)	A = X & Y	ALUOP = 18	ALUFAST = 1	ALUSIZE = size
NAND(size)	$A = \overline{(X \& Y)}$	ALUOP = 17	ALUFAST = 1	ALUSIZE = size
YANDXC(size)	$A = Y \& \overline{X}$	ALUOP = 14	ALUFAST = 1	ALUSIZE = size
OR1(size)	A = X OR Y	ALUOP = 1E	ALUFAST = 1	ALUSIZE = size
EXOR(size)	A = X XOR Y	ALUOP = 16	ALUFAST = 1	ALUSIZE = size
NOTX(size)	$A = \overline{X}$	ALUOP = 15	ALUFAST = 1	ALUSIZE = size
RINGX	ring wrap X if X<31> = 1: A = 100_2::X<28:0> else A = X	ALUOP = 19	ALUFAST = 1	ALUSIZE = size
CVTX(W,B)	A = cvtw.b(X)	ALUOP = 11	ALUFAST = 1	ALUSIZE = 2
CVTX(W,H)	A = cvtw.h(X)	ALUOP = 12	ALUFAST = 1	ALUSIZE = 2
CVTX(L,W)	A = cvtl.w(X)	ALUOP = 13	ALUFAST = 1	ALUSIZE = 3
CVTX(B,W)	A = cvtb.w(X)	ALUOP = 02	ALUFAST = 0	ALUSIZE = 0
CVTX(H,W)	A = cvth.w(X)	ALUOP = 03	ALUFAST = 0	ALUSIZE = 1
CVTX(W,L)	A = cvtw.l(X)	ALUOP = 04	ALUFAST = 0	ALUSIZE = 2

7.2.5 Register File Write Control Constructs

The next set of constructs to be described controls the writing of the scalar register file from the A bus. The register to be written may be specified by the X, Y, or Z port select. In addition, the size of the result operand, and thus the width of register to be written, is specified. The standard B (byte), H (halfword), W (word), L (longword), S (single float, same as word), and D (double float, same as longword) notation is used. Also, a hazard check is encoded for the written register, as it may have been reserved for a function unit or memory operation destination. If the hazard is caught by a register write only, it implies that the destination register was being written without being read. This is unlikely, but since all operations must complete in the order they were initiated, the register file write construct must encode a hazard check. The register file write constructs are shown in Figure 7-4.

Figure 7-4 Register File Write Control Constructs

construct	encodings			
WRFX(B)	ABUS_FMT = 0	ABUS_WR_EN = 1	ABUS_SIZE = 0	XREG_HAZ = 0
WRFX(H)	ABUS_FMT = 0	ABUS_WR_EN = 1	ABUS_SIZE = 1	XREG_HAZ = 2
WRFX(W)	ABUS_FMT = 0	ABUS_WR_EN = 1	ABUS_SIZE = 2	XREG_HAZ = 3
WRFX(S)	ABUS_FMT = 0	ABUS_WR_EN = 1	ABUS_SIZE = 2	XREG_HAZ = 3
WRFX(L)	ABUS_FMT = 0	ABUS_WR_EN = 1	ABUS_SIZE = 3	XREG_HAZ = 5
WRFX(D)	ABUS_FMT = 0	ABUS_WR_EN = 1	ABUS_SIZE = 3	XREG_HAZ = 5
WRFY(B)	ABUS_FMT = 1	ABUS_WR_EN = 1	ABUS_SIZE = 0	YREG_HAZ = 0
WRFY(H)	ABUS_FMT = 1	ABUS_WR_EN = 1	ABUS_SIZE = 1	YREG_HAZ = 2
WRFY(W)	ABUS_FMT = 1	ABUS_WR_EN = 1	ABUS_SIZE = 2	YREG_HAZ = 3
WRFY(S)	ABUS_FMT = 1	ABUS_WR_EN = 1	ABUS_SIZE = 2	YREG_HAZ = 3
WRFY(L)	ABUS_FMT = 1	ABUS_WR_EN = 1	ABUS_SIZE = 3	YREG_HAZ = 5
WRFY(D)	ABUS_FMT = 1	ABUS_WR_EN = 1	ABUS_SIZE = 3	YREG_HAZ = 5
WRFZ(B)	ABUS_FMT = 2	ABUS_WR_EN = 1	ABUS_SIZE = 0	ZREG_HAZ = 0
WRFZ(H)	ABUS_FMT = 2	ABUS_WR_EN = 1	ABUS_SIZE = 1	ZREG_HAZ = 2
WRFZ(W)	ABUS_FMT = 2	ABUS_WR_EN = 1	ABUS_SIZE = 2	ZREG_HAZ = 3
WRFZ(S)	ABUS_FMT = 2	ABUS_WR_EN = 1	ABUS_SIZE = 2	ZREG_HAZ = 3
WRFZ(L)	ABUS_FMT = 2	ABUS_WR_EN = 1	ABUS_SIZE = 3	ZREG_HAZ = 5
WRFZ(D)	ABUS_FMT = 2	ABUS_WR_EN = 1	ABUS_SIZE = 3	ZREG_HAZ = 5

7.2.6 PSW and USW Control Constructs

Status flag outputs from the integer ALU may be loaded into the Program Status Word (PSW) or Microsequencer Status Word (USW) using the CLD construct. This construct identifies a source for the status bit and a destination bit location in the PSW or USW.

The destination may be any of the "carry" bits in the PSW (AC, SC) or any of the bits of the USW (IC, JC, KC). In addition, for context restore, the KC bit in the USW may be loaded with a pending hazard ignored. Finally, it is possible (though not particularly useful) to specify no destination for

the selected status flag. The source specifies which status flag output of the ALU is to be loaded - ALU carry out, complement of ALU carry out, or the compare X to Y flags. The **CLD** construct is detailed in Figure 7-5.

Figure 7-5 PSW and USW Loading Construct

construct	encodings
CLD(NO,source)	CRY_OP = 2 CRY_DST = 0
CLD(AC,source)	CRY_OP = 2 CRY_DST = 1
CLD(SC,source)	CRY_OP = 2 CRY_DST = 2
CLD(IC,source)	CRY_OP = 2 CRY_DST = 3
CLD(JC,source)	CRY_OP = 2 CRY_DST = 4
CLD(KC,source)	CRY_OP = 2 CRY_DST = 5
CLD(KC_NH,source)	CRY_OP = 2 CRY_DST = 6
CLD(dest,CRY)	CRY_OP = 2 CRY_SRC = 0
CLD(dest,CRY@)	CRY_OP = 2 CRY_SRC = 1
CLD(dest,EQL)	CRY_OP = 2 CRY_SRC = 2
CLD(dest,LSS)	CRY_OP = 2 CRY_SRC = 3
CLD(dest,LEQ)	CRY_OP = 2 CRY_SRC = 4

The same PSW and USW bits may also be loaded from the scalar function units or communication registers with the **CRSV** construct. This construct causes a reservation to be placed in the status bit return queues. There are two such queues - one for function unit return status and one for communication register return status. The **CRSV** construct specifies a destination bit in the PSW or USW for the returned status and a source which identifies which queue to enter the reservation in. The **CRSV** construct is detailed in Figure 7-6

Figure 7-6 PSW and USW Reservation Construct

construct	encodings
CRSV(NO,source)	CRY_OP = 1 CRY_DST = 0
CRSV(AC,source)	CRY_OP = 1 CRY_DST = 1
CRSV(SC,source)	CRY_OP = 1 CRY_DST = 2
CRSV(IC,source)	CRY_OP = 1 CRY_DST = 3
CRSV(JC,source)	CRY_OP = 1 CRY_DST = 4
CRSV(KC,source)	CRY_OP = 1 CRY_DST = 5
CRSV(dest,FU)	CRY_OP = 1 CRY_SRC = 5
CRSV(dest,CU)	CRY_OP = 1 CRY_SRC = 6

Additionally, there are a set of constructs to control the loading of the PSW bits other than the "carries", such as overflow and frame length. There is also a construct to zero the bits in the PSW that need to be cleared by the call instructions. Finally, there's a construct to enable the loading of the PSW from the main scalar data path via the Y bus. These miscellaneous PSW constructs are detailed in Figure 7-7

Figure 7-7 Miscellaneous PSW Control Constructs

construct	description	encoding
LDAV	PSW<AIV> = ALU overflow	PSW_OP = 1
LDSV	PSW<SIV> = ALU overflow	PSW_OP = 2
CLR_PSW	PSW<AC,AIV,ADZ,FRL,SC,SIV, SDZ,UN,OV,RO,FDZ,FIN> = 0	PSW_OP = 3
LDFRL01	PSW<FRL> = 01_2	PSW_OP = 4
LDFRL10	PSW<FRL> = 10_2	PSW_OP = 5
LDFRL11	PSW<FRL> = 11_2	PSW_OP = 6
WR_PSW	PSW = Y bus	PSW_OP = 7

7.2.7 Y Bus and Z Mux Write Control Constructs

The 32-bit Y bus, which is generated by the NRFA arrays from the registered Y operand in the main scalar data path, is distributed to the remainder of the NSP for a number of purposes. The NDC is given vector length (VL), Communication Index Register (CIR), and Thread Identifier (TID) information via this path. The NPSW array is given data for its CPU Control Register (CCR) as well as commanded to clear its microsecond timer via these constructs. The NIP is given jump address data via the Y bus, however the constructs to control the NIP are detailed in section 7.2.13 on page 7-31.

The 32-bit Z read port output of the register file is used to distribute vector stride (VS) to the NDC. The write control for the VS destination register in the NDC is controlled by an encoding of the **YZ_WR_DST** field, specified by the **WR_VS** construct. This additionally enables hazard checking for the address generator logic in the NDC.

The Y and Z write control constructs, which are essentially mnemonics encoding the **YZ_WR_DST** field of the microinstruction, are detailed in Figure 7-8.

Figure 7-8 Y and Z Write Control Constructs

construct	description	encodings
WR_CCR	CCR = Y bus	YZ_WR_DST = 2
CLR_TIMER	clear microsecond timer	YZ_WR_DST = 3
WR_CIR	NDC's CIR = Y<4..0>	YZ_WR_DST = 4
WR_TID	NDC's TID = Y<4..0>	YZ_WR_DST = 5
WR_VL	NDC's VL = Y<7..0>	YZ_WR_DST = 6

WR_VS	NDC's VS = Z	YZ_WR_DST = 7	AGZ_HAZEN = 1

7.2.8 Address Generator Source Control Constructs

The cycle before a memory or communication register request is made to the data cache, the source for the address generator is specified with the **AG** construct. The Z mux output of the register file is the simplest source for this address. In addition, the Z mux with 4 or 8 subtracted may be selected for stack push operations. The address generator also registers the address from the last request. This saved address or its value modified by an addition or subtraction may also be selected. Finally, advanced effective address mode may be selected, which uses the Z register select of the next microinstruction in the pipeline to read out a register on the Z mux. This mode is most often used on the last microinstruction of a macroinstruction, and in fact is the default encoding whenever the microsequencer either conditionally or unconditionally accepts the next instruction dispatch. The **AG** construct is detailed in Figure 7-9.

Figure 7-9 Address Generator Source Control Construct

construct	selected AG source	encoding
AG(AE)	advanced effective address (effa)	AG_SEL = 0
AG(Z)	Z mux	AG_SEL = 1
AG(HOLD)	hold last address	AG_SEL = 2
AG(LP4)	last address plus 4	AG_SEL = 3
AG(LP8)	last address plus 8	AG_SEL = 4
AG(ZM4)	Z mux minus 4	AG_SEL = 5
AG(ZM8)	Z mux minus 8	AG_SEL = 6
AG(LM4)	last address minus 4	AG_SEL = 7
AG(LM8)	last address minus 8	AG_SEL = 8
AG(LP1)	last address plus 1 (for comm reg)	AG_SEL = 9

7.2.9 Memory Control Constructs

One of the functions of the NAS, through US microcode, is the generation of address for and initiation of memory operations. Section 7.2.8 described the instructions used for address generation. This section describes the memory operation constructs.

There are many types of memory operations. For scalar loads and stores, the NAS makes a request to the data cache, supplies data for stores, or sets up control for the destination register to be loaded when data returns for loads. For vector loads and stores, the NAS generates the initial address and first request and turns the remainder of the VL requests over to the Vector Address Generator (VAG) in the NDC subsystem. Many operations may be under mask. These operations require the NVP to supply bits from the VM register to enable or disable each element of the operation. There are also special memory operations for indexed vector operations, which require the NVP to supply vector elements to be used as indices by the VAG. Other memory

operations are used to request PTEs and purge the data, PTE, and referenced / modified caches. The basic memory control constructs are detailed in Figure 7-10.

Figure 7-10 Basic Memory Control Constructs

construct	description	encodings	
MEM_NOP	no operation	MEM_OP_REQ = 0	
READ	load	MEM_OP_REQ = 1	MEM_OP_DST = 7
		MEM_OP_TYPE = 01	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 3	
PHYS_READ	physical address based load	MEM_OP_REQ = 1	MEM_OP_DST = 4
		MEM_OP_TYPE = 01	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 0	
READ_BYPASS	load bypassing data cache	MEM_OP_REQ = 1	MEM_OP_DST = 4
		MEM_OP_TYPE = 01	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 2	
IREAD	read longword to instruction cache	MEM_OP_REQ = 1	MEM_OP_DST = 1
		MEM_OP_TYPE = 01	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 3	
VREAD(size)	vector load	MEM_OP_REQ = 1	MEM_OP_DST = 2
		MEM_OP_TYPE = 05	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 2	BDSIZE = size
VREAD_IX(size)	indexed vector load	MEM_OP_REQ = 1	MEM_OP_DST = 2
		MEM_OP_TYPE = 25	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 2	BDSIZE = size
VREADM(size)	vector load under mask	MEM_OP_REQ = 1	MEM_OP_DST = 2
		MEM_OP_TYPE = 35	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 2	BDSIZE = size
VREADM_IX(size)	indexed vector load under mask	MEM_OP_REQ = 1	MEM_OP_DST = 2
		MEM_OP_TYPE = 35	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 2	BDSIZE = size
WRITE(size)	store	MEM_OP_REQ = 1	MEM_OP_DST = 5
		MEM_OP_TYPE = 02	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 3	BDSIZE = size
PHYS_WRITE(size)	physical address based store	MEM_OP_REQ = 1	MEM_OP_DST = 5
		MEM_OP_TYPE = 02	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 0	BDSIZE = size
VWRITE(size)	vector store	MEM_OP_REQ = 1	MEM_OP_DST = 2
		MEM_OP_TYPE = 06	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 2	BDSIZE = size
VWRITE_IX(size)	indexed vector	MEM_OP_REQ = 1	MEM_OP_DST = 2

	store	MEM_OP_TYPE = 26 MEM_OP_AT_TYPE = 2	AGZ_HAZEN = 1 BDSIZE = size
VWRITEM(size)	vector store under mask	MEM_OP_REQ = 1 MEM_OP_TYPE = 16 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 2 AGZ_HAZEN = 1 BDSIZE = size
VWRITEM_IX(size)	indexed vector store under mask	MEM_OP_REQ = 1 MEM_OP_TYPE = 36 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 2 AGZ_HAZEN = 1 BDSIZE = size
WRITE_IX(size)	indexed scalar store	MEM_OP_REQ = 1 MEM_OP_TYPE = 2E MEM_OP_AT_TYPE = 2	MEM_OP_DST = 2 AGZ_HAZEN = 1 BDSIZE = size
WRITEM_IX(size)	indexed scalar store under mask	MEM_OP_REQ = 1 MEM_OP_TYPE = 3E MEM_OP_AT_TYPE = 2	MEM_OP_DST = 2 AGZ_HAZEN = 1 BDSIZE = size
STE(size)	extended scalar store	MEM_OP_REQ = 1 MEM_OP_TYPE = 0E MEM_OP_AT_TYPE = 2	MEM_OP_DST = 2 AGZ_HAZEN = 1 BDSIZE = size
STEMsize)	extended scalar store under mask	MEM_OP_REQ = 1 MEM_OP_TYPE = 1E MEM_OP_AT_TYPE = 2	MEM_OP_DST = 2 AGZ_HAZEN = 1 BDSIZE = size
TAM(size)	test and modify	MEM_OP_REQ = 1 MEM_OP_TYPE = 03 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4 AGZ_HAZEN = 1 BDSIZE = size
PATE	purge PTE cache entry	MEM_OP_REQ = 1 MEM_OP_TYPE = 20 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4 AGZ_HAZEN = 1
PATU	purge PTE cache	MEM_OP_REQ = 1 MEM_OP_TYPE = 38 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4
PREF	purge referenced bit cache	MEM_OP_REQ = 1 MEM_OP_TYPE = 14 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4
PMOD	purge modified bit cache	MEM_OP_REQ = 1 MEM_OP_TYPE = 24 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4
PURGE_PTE_TVAL	purge PTE cache thread validity	MEM_OP_REQ = 1 MEM_OP_TYPE = 18 MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4
PURGE_DC_TVAL	purge data cache thread validity	MEM_OP_REQ = 1 MEM_OP_TYPE = 1C	MEM_OP_DST = 4

		MEM_OP_AT_TYPE = 2	
PURGE_DC_VAL	purge data cache non-thread validity	MEM_OP_REQ = 1 MEM_OP_TYPE = 2C MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4
PURGE_DC_ALL	purge data cache thread and non-thread validity	MEM_OP_REQ = 1 MEM_OP_TYPE = 3C MEM_OP_AT_TYPE = 2	MEM_OP_DST = 4
READ_PTE1	read level 1 PTE	PTE_OP_REQ = 1 MEM_OP_TYPE = 28	AGZ_HAZEN = 1
READ_PTE2	read level 2 PTE	PTE_OP_REQ = 1 MEM_OP_TYPE = 30	AGZ_HAZEN = 1
READ_PTET	read thread level PTE	PTE_OP_REQ = 1 MEM_OP_TYPE = 38	AGZ_HAZEN = 1
WRITE_PTE2_CACHE	write PTE cache with nonthread validity	PTE_OP_REQ = 1 MEM_OP_TYPE = 00	AGZ_HAZEN = 1
WRITE_PTET_CACHE	write PTE cache with thread validity	PTE_OP_REQ = 1 MEM_OP_TYPE = 08	AGZ_HAZEN = 1
RETRY_REQ	re-initiate request from PTE miss	REQ_RETRY = 1	
RESET_DC_PREFETCH	reset block prefetch	MEM_OP_REQ = 1 MEM_OP_TYPE = 10	
ZHAZEN	enable AG hazard check	AGZ_HAZEN = 1	

There are two common modifiers for most memory operations. The RA_ prefix is used to denote a registered address operation - i.e. the address selected by the AG construct in the previous microinstruction is registered from a previous request. The difference in the RA_ prefixed version is that there is no need for hazard checking by the address generator since the address is not coming from the register file. The RA_ prefixed memory control constructs are listed in Figure 7-11, with their definition referring to the basic construct they are built on. There are more constructs that use the RA_ prefix, but they also use a suffix modifier, so their introduction is postponed until the suffix modifier is defined.

Figure 7-11 Registered Address Memory Constructs

RA_READ	READ with AGZ_HAZEN = 0
RA_IREAD(size)	IREAD(size) with AGZ_HAZEN = 0
RA_WRITE(size)	WRITE(size) with AGZ_HAZEN = 0

RA_VWRITEM_IX(size)	VWRITEM_IX(size) with AGZ_HAZEN = 0
RA_STE(size)	STE(size) with AGZ_HAZEN = 0
RA_TAM	TAM with AGZ_HAZEN = 0

The **_R0** suffix is used to denote that the request is made with the **FAKE_RING0** field set, denoting that ring maximization checking is disabled for the request. An example of its use is in the system call microcode to reference ring 0 memory addresses before the PC is changed to ring 0. Normally this would result in an inward ring reference fault, but the **_R0** suffix on the memory operation bypasses that fault check. The **_R0** suffixed memory control constructs are listed in Figure 7-12, with their definition referring to the basic construct they are built on. There are some constructs here which also use a registered address and therefore also have the **RA_** prefix.

Figure 7-12 Fake Ring 0 Memory Constructs

READ_R0	READ with FAKE_RING0 = 1
READ_BYPASS_R0	READ_BYPASS with FAKE_RING0 = 1
WRITE_R0(size)	WRITE(size) with FAKE_RING0 = 1
RA_WRITE_R0(size)	WRITE(size) with AGZ_HAZEN = 0 and FAKE_RING0 = 1
RA_VWRITEM_IX_R0(size)	VWRITEM_IX(size) with AGZ_HAZEN = 0 and FAKE_RING0 = 1

Note that in general it is possible to apply the **RA_** prefix or **_R0** suffix to most basic memory operations. The ones defined here are those actually defined in the released microlanguage definition, i.e. are actually used in the instruction set microcode.

7.2.10 Communication Register Control Constructs

In the C38xx system, communication registers reside across the crossbar subsystem from the NSP, on the NCU. In this respect, to the NSP they appear very similar to memory. Virtual address translation for communication register requests is performed by the NDC. The microcode constructs to initiate communication register operations are therefore very similar in nature to the memory operation constructs detailed in section 7.2.9. The method for generating a communication register operation is also similar to memory operations. The cycle before the communication register request, the AG construct is used to specify the address. An address must be supplied for all operations, since the NDC attempts a translation even for address-less operations like **DSI** and **ENI**. In this case a bogus yet valid address is supplied. Next, one of the constructs in Figure 7-13 is used to specify the type of communication register operation. For write operations (e.g. PUT) the NAS supplies data on the Y bus for the write. For read or status operations, other constructs are used to reserve a register (**RSV**, section 7.2.12 on page 7-31) and/or status bit (CRSV, section 7.2.6 on page 7-19) for return data/status from the NCU. Recall that the communication registers include a partition known as control, or X, space. These "hardware" communication register addresses require a special mapping. This is achieved with a

separate set of constructs with the _X suffix, which encode different memory operation type field values than their non-_X counterparts.

Figure 7-13 Basic Communication Register Control Constructs

construct	description	encodings	
ENI	set ION, return	MEM_OP_REQ = 1	MEM_OP_DST = 4
	old value of ION	MEM_OP_TYPE = 08	AGZ_HAZEN = 1
	as status	MEM_OP_AT_TYPE = 1	
DSI	clear ION, return	MEM_OP_REQ = 1	MEM_OP_DST = 4
	old value of ION	MEM_OP_TYPE = 0C	AGZ_HAZEN = 1
	as status	MEM_OP_AT_TYPE = 1	
LCK	set lock, return	MEM_OP_REQ = 1	MEM_OP_DST = 4
	old value of lock	MEM_OP_TYPE = 00	AGZ_HAZEN = 1
	as status	MEM_OP_AT_TYPE = 1	
ULK	clear lock, return	MEM_OP_REQ = 1	MEM_OP_DST = 4
	old value of lock	MEM_OP_TYPE = 04	AGZ_HAZEN = 1
	as status	MEM_OP_AT_TYPE = 1	
TST	return lock bit	MEM_OP_REQ = 1	MEM_OP_DST = 4
	as status	MEM_OP_TYPE = 10	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 1	
RCV	return lock, if it	MEM_OP_REQ = 1	MEM_OP_DST = 4
	was set clear it	MEM_OP_TYPE = 01	AGZ_HAZEN = 1
	and return data	MEM_OP_AT_TYPE = 1	
GET	return data	MEM_OP_REQ = 1	MEM_OP_DST = 4
		MEM_OP_TYPE = 11	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 1	
SND(size)	return lock, if it	MEM_OP_REQ = 1	MEM_OP_DST = 4
	was clear set it	MEM_OP_TYPE = 02	AGZ_HAZEN = 1
	and return data	MEM_OP_AT_TYPE = 1	BDSIZE = size
PUT(size)	write data	MEM_OP_REQ = 1	MEM_OP_DST = 4
		MEM_OP_TYPE = 12	AGZ_HAZEN = 1
		MEM_OP_AT_TYPE = 1	BDSIZE = size
PUTS(size)	write data, return	MEM_OP_REQ = 1	MEM_OP_DST = 4
	status when NCU	MEM_OP_TYPE = 0E	AGZ_HAZEN = 1
	gets request	MEM_OP_AT_TYPE = 1	BDSIZE = size
MAT(size)	return status of	MEM_OP_REQ = 1	MEM_OP_DST = 4
	sent data = comm	MEM_OP_TYPE = 1E	AGZ_HAZEN = 1
	reg contents	MEM_OP_AT_TYPE = 1	BDSIZE = size

INC(size)	return lock, if it was set, add sent data to comm reg	MEM_OP_REQ = 1 MEM_OP_TYPE = 03 MEM_OP_AT_TYPE = 1	MEM_OP_DST = 4 AGZ_HAZEN = 1 BDSIZE = size
RCV_X	RCV from control (X) address space	MEM_OP_REQ = 1 MEM_OP_TYPE = 05 MEM_OP_AT_TYPE = 1	MEM_OP_DST = 4 AGZ_HAZEN = 1 FAKE_RING0 = 1
PUT_X(size)	PUT to control (X) address space	MEM_OP_REQ = 1 MEM_OP_TYPE = 16 MEM_OP_AT_TYPE = 1 FAKE_RING0 = 1	MEM_OP_DST = 4 AGZ_HAZEN = 1 BDSIZE = size
GET_X	RCV from control (X) address space	MEM_OP_REQ = 1 MEM_OP_TYPE = 15 MEM_OP_AT_TYPE = 1	MEM_OP_DST = 4 AGZ_HAZEN = 1 FAKE_RING0 = 1
RDCMR	return data, shift lock into lock bit accumulator	MEM_OP_REQ = 1 MEM_OP_TYPE = 19 MEM_OP_AT_TYPE = 1	MEM_OP_DST = 4 AGZ_HAZEN = 1 FAKE_RING0 = 1
WRCMR(size)	write data, write lock from lock accumulator	MEM_OP_REQ = 1 MEM_OP_TYPE = 15 MEM_OP_AT_TYPE = 1 FAKE_RING0 = 1	MEM_OP_DST = 4 AGZ_HAZEN = 1 BDSIZE = size

The same type of registered address prefixing (RA_) and fake ring 0 suffixing (_R0) is used for these constructs. The RA_ prefixed constructs are listed in Figure 7-14. The _R0 suffixed constructs are listed in Figure 7-15. Constructs with both the RA_ prefix and _R0 suffix are also included in Figure 7-15.

Figure 7-14 Registered Address Communication Register Constructs

RA_LCK	LCK with AGZ_HAZEN = 0
RA_ULK	ULK with AGZ_HAZEN = 0
RA_RCV	RCV with AGZ_HAZEN = 0
RA_GET	GET with AGZ_HAZEN = 0
RA_SND(size)	SND(size) with AGZ_HAZEN = 0
RA_PUT(size)	PUT(size) with AGZ_HAZEN = 0
RA_PUT_X(size)	PUT_X(size) with AGZ_HAZEN = 0
RA_GET_X	GET_X with AGZ_HAZEN = 0
RA_RDCMR	RDCMR with AGZ_HAZEN = 0
RA_WRCMR(size)	WRCMR(size) with AGZ_HAZEN = 0

Figure 7-15 Fake Ring 0 Communication Register Constructs

RA_LCK_R0	LCK with FAKE_RING0 = 1 and AGZ_HAZEN = 0
LCK_R0	LCK with FAKE_RING0 = 1
ULK_R0	ULK with FAKE_RING0 = 1
RA_ULK_R0	ULK with FAKE_RING0 = 1 and AGZ_HAZEN = 0
RCV_R0	RCV with FAKE_RING0 = 1
RA_RCV_R0	RCV with FAKE_RING0 = 1 and AGZ_HAZEN = 0
GET_R0	GET with FAKE_RING0 = 1
RA_GET_R0	GET with FAKE_RING0 = 1 and AGZ_HAZEN = 0
SND_R0(size)	SND(size) with FAKE_RING0 = 1
RA_SND_R0(size)	SND(size) with FAKE_RING0 = 1 and AGZ_HAZEN = 0
PUT_R0(size)	PUT(size) with FAKE_RING0 = 1
RA_PUT_R0(size)	PUT(size) with FAKE_RING0 = 1 and AGZ_HAZEN = 0
RA_PUTS_R0(size)	PUTS(size) with FAKE_RING0 = 1 and AGZ_HAZEN = 0

7.2.11 Function Unit Control Constructs

The next set of microcode constructs to be described control the 4 scalar function units on the NSP. These are the NFAD, NMUL, and NDIV custom parts and the NMISC gate array. These function units perform all scalar floating point operations as well as many integer operations such as multiply, divide, and shift that are not executed in the NRFA data path slices. The US microcode initiates these function unit operations by placing operands on the X and Y mux outputs from the NRFA data path (recall the mux level means they're at the UIR1 operand access stage of the pipe) and starting the function unit with and encoding of the **MFP_OP** field. A register is reserved for the result (or carry bit for compare operations) using the RSV construct described in section 7.2.12 on page 7-31. The size of register reserved for the result also is sent to the function unit to control the size of operation it performs. In some cases, a size parameter is added to the function unit control construct to denote the other operation's data type in a dyadic operation. The function unit control constructs are listed in Figure 7-16. The *cvtx.y* operations follow the descriptions in the Convex Architecture Reference.

Figure 7-16 Function Unit Control Constructs

NFAD construct	operation	encodings		
F_ADD	X + Y	FU_OP_REQ = 1	FU_OP = 00	
F_YSUBX	Y - X	FU_OP_REQ = 1	FU_OP = 08	
F_XSUBY	Y - X	FU_OP_REQ = 1	FU_OP = 10	
F_YGTRX(size)	Y > X ?	FU_OP_REQ = 1	FU_OP = 1C	BDSIZE = size
F_YEQLX(size)	Y = X ?	FU_OP_REQ = 1	FU_OP = 19	BDSIZE = size
F_YGEQX(size)	Y >= X ?	FU_OP_REQ = 1	FU_OP = 1D	BDSIZE = size
F_YLSSX(size)	Y < X ?	FU_OP_REQ = 1	FU_OP = 1A	BDSIZE = size
F_YLEQX(size)	Y <= X ?	FU_OP_REQ = 1	FU_OP = 1B	BDSIZE = size

NMUL construct	operation		encodings
A_MUL	int X * Y, set AIV	FU_OP_REQ = 1	FU_OP = 60
S_MUL	int X * Y, set SIV	FU_OP_REQ = 1	FU_OP = 40
F_MUL	float X * Y	FU_OP_REQ = 1	FU_OP = 41

NDIV construct	operation		encodings
A_YDIVX	int Y/X, set AIV	FU_OP_REQ = 1	FU_OP = A0
S_YDIVX	int Y/X set SIV	FU_OP_REQ = 1	FU_OP = 80
F_YDIVX	float Y/X	FU_OP_REQ = 1	FU_OP = 81
F_SQRTY	float square root of Y	FU_OP_REQ = 1	FU_OP = 83

NMISC construct	operation		encodings	
LOPY	leading 1's pos of Y	FU_OP_REQ = 1	FU_OP = C0	BDSIZE = 3
TZCY	trailing 0 count of Y	FU_OP_REQ = 1	FU_OP = C1	BDSIZE = 3
XSHFY	Y shifted X	FU_OP_REQ = 1	FU_OP = C3	
PLCT64Y	# of 1s in Y	FU_OP_REQ = 1	FU_OP = C4	BDSIZE = 3
PLCTXY	# of 1s in X bits of Y	FU_OP_REQ = 1	FU_OP = C5	BDSIZE = 3
PLCFXY	# of 0s in X bits of Y	FU_OP_REQ = 1	FU_OP = C6	BDSIZE = 3
FRINTY	integerize Y	FU_OP_REQ = 1	FU_OP = D7	
CVTY(B,dest)	cvtb.dest(Y)	FU_OP_REQ = 1 if (dest = S or D) FU_OP = D8 else FU_OP = C8		
CVTY(H,dest)	cvth.dest(Y)	FU_OP_REQ = 1 if (dest = S or D) FU_OP = D9 else FU_OP = C9		
CVTY(W,dest)	cvtw.dest(Y)	FU_OP_REQ = 1 if (dest = S or D) FU_OP = DA else FU_OP = CA		
CVTY(L,dest)	cvtl.dest(Y)	FU_OP_REQ = 1 if (dest = S or D) FU_OP = DB else FU_OP = CB		
CVTY(S,dest)	cvts.dest(Y)	FU_OP_REQ = 1 if (dest = S or D) FU_OP = DE else FU_OP = CE		
CVTY(D,dest)	cvts.dest(Y)	FU_OP_REQ = 1 if (dest = S or D) FU_OP = DF else FU_OP = CF		

7.2.12 Register Reservation Constructs

The results returned from memory, communication register, and function unit operations are written to registers under control of the RSV register reservation construct. When these so-called "backdoor" operations are initiated, the **RSV** construct, by encoding the **BDOP** and **BDREG_FMT** fields, specifies which register select (**XREG_SEL**, **YREG_SEL**, or **ZREG_SEL**) denotes the destination register. The size of result is also specified through encoding the BDSIZE field. This information is pushed into queues that are popped when data is returned with a B bus (function unit) or C bus (memory or communication register) write. Information popped from the queue controls the width and address of the register to be written. **RSV** also sets a hazard in scoreboard for the specified register. For context save and restore, **RSV** can encode the **BDOP** field to clear scoreboard bits. For vector to scalar transfers, RSV also encodes the **VXS_REQ** control field. The **RSV** register reservation construct is detailed in Figure 7-17.

Figure 7-17 Register Reservation Construct

construct	encodings			
RSV(X,size)	BDREG_FMT = 1	BDOP = 1	XREG_HAZ = size	BDSIZE = size
RSV(Y,size)	BDREG_FMT = 2	BDOP = 1	YREG_HAZ = size	BDSIZE = size
RSV(Z,size)	BDREG_FMT = 3	BDOP = 1	ZREG_HAZ = size	BDSIZE = size
RSV(XVP,size)	BDREG_FMT = 1 VXS_REQ = 1	BDOP = 1	XREG_HAZ = size	BDSIZE = size
RSV(YVP,size)	BDREG_FMT = 2 VXS_REQ = 1	BDOP = 1	YREG_HAZ = size	BDSIZE = size
RSV(ZVP,size)	BDREG_FMT = 3 VXS_REQ = 1	BDOP = 1	ZREG_HAZ = size	BDSIZE = size
RSV(XCLR,size)	BDREG_FMT = 1	BDOP = 3	XREG_HAZ = size	BDSIZE = size
RSV(YCLR,size)	BDREG_FMT = 2	BDOP = 3	YREG_HAZ = size	BDSIZE = size
RSV(ZCLR,size)	BDREG_FMT = 3	BDOP = 3	ZREG_HAZ = size	BDSIZE = size

7.2.13 Instruction Processor Control Constructs

The next set of constructs to be described control the NIP, initiating restarts and limiting the extent of its lookahead prefetch. There are 4 types of NIP restarts. A **SYSJMP** is used to give the NIP a jump address and allow it to cross rings. A **USRJMP** gives a jump address but requires that ring wrapping take effect. A **UINT_RESTART** is used to restart the NIP after a microinterrupt, jumping it to an internally saved PC but not resetting its lookahead. Finally, a **RTNC_RESTART** is used at the end of restoring context to jump the NIP to the restored PC and reset its lookahead address. At times it is necessary to keep the NIP from making lookahead requests. This is a mode that is begun with the **SETINH** construct, and ends when a **CLRINH** is coded. This actually works by making the NDC ignore NIP requests while this mode is active - the NIP continues making requests to no avail. Note that the **SYSJMP** and **USRJMP** constructs implicitly clear this mode. There is also a variant of **USRJMP** called **USRJMP_SETINH** which sets NIP inhibit mode rather than clears it. The NIP control constructs are listed in Figure 7-18.

Figure 7-18 Instruction Processor Control Constructs

construct	encodings	
SYSJMP	JMP_RESTART = 2	IPINH = 1
USRJMP	JMP_RESTART = 3	IPINH = 1
USRJMP_SETINH	JMP_RESTART = 3	IPINH = 3
UINT_RESTART	JMP_RESTART = 1	USEQ_RAND = 5
RTNC_RESTART	JMP_RESTART = 1	
SETINH	IPINH = 3	
CLRINH	IPINH = 1	

7.2.14 Vector Processor Control Constructs

The NSP has a few controls over the NVP regarding dispatching and data transfer. This section describes the US microcode constants that perform these functions. For all instructions (vector and scalar), the NIP cracks the opcode and produces an entrypoint to the US microcode on the NSP and vector dispatch logic on NVP. For vector instructions, the vector processor must be told to execute microcode corresponding to the dispatched entrypoint. For non-vector instructions, the NVP does nothing. The control over this is via the DISPATCH_VP construct. Note that unlike previous Convex machines, the scalar processor is dispatched *every* instruction, including vector only instructions. This is so the NSP can dispatch the NVP.

There is one optional parameter to the **DISPATCH_VP** construct, **PSW**, which tells the NVP to assert a PSW hazard on the instruction. For example, vector loads do not need to assert a PSW hazard, and therefore are not dispatched with the **PSW** parameter. Vector add instructions, however, do cause PSW hazard and therefore the US microcode includes the **PSW** option with the dispatch construct.

The NSP also controls scalar-to-vector and vector-to-scalar data transfers. The **RSV** construct, described in section 7.2.12 on page 7-31, controls vector-to-scalar (VXS) transfers like reading a single vector element. The **SXV** construct initiates a scalar-to-vector transfer request for operations such as Vector Merge register writes. Data for an **SXV** transfer is placed on the Y bus on the same microinstruction the **SXV** is coded.

The NVP control constructs are detailed in Figure 7-19

Figure 7-19 Vector Processor Control Constructs

construct	encodings	
DISPATCH_VP	VP_DISP = 1	VP_PSW_HAZ = 0
DISPATCH_VP(PSW)	VP_DISP = 1	VP_PSW_HAZ = 1
SXV	SXV_REQ = 1	

7.2.15 Scratch RAM Control Constructs

The NAS includes a 2K by 36-bit (32 data and 4 parity) scratch RAM used for encaching SDRs and first level PTEs, holding constants, and acting as temporary storage. The RAM address may come from a number of sources. The most common is from the **UCONST** field, specified in the **XY** construct described in section 7.2.3 on page 7-16. The PTE miss address, SA1_MISS1, may be used as the address for SDR and PTE1 encachements, also specified with the XY construct. During fault or rtnc microcode, a counter may be loaded and incremented to be used as an indexed address (the counter may only be used in fault/rtnc because it is not saved in the context block). The output of the RAM is readable on the X mux in the main data path, using the XY construct. Scratch RAM is written from the Y bus under control of the **WR_RAM** construct, which also specifies the write address source. Finally, the validity RAM associated with scratch RAM (used for the PTE1 cache) may be purged using the **PURGE_SRVAL** construct. The scratch RAM control constructs are detailed in Figure 7-20.

Figure 7-20 Scratch RAM Control Constructs

construct	encodings	
SRCNT(INCR)	SR_CNT_SEL = 2	
SRCNT(addr)	SR_CNT_SEL = 1	UCONST = addr
WR_RAM(CNT)	SR_ADDR_SEL = 1	
WR_RAM(PTE1)	SR_ADDR_SEL = 3	
WR_RAM(addr)	SR_ADDR_SEL = 0	UCONST = addr
PURGE_SRVAL	USEQ_RAND = 3	

7.2.16 Loop Counter Control Constructs

The NAS microsequencer contains a 10-bit counter that may be tested by the branch logic and used to implement microcode loops. The test condition available is simply the most significant bit of the counter, i.e. trip when the counter goes negative. Refer to section 7.2.20 on page 7-35 for a description of how to select the loop counter test construct. The counter is used by loading a positive value from the **UCONST** field and decrementing it until it becomes negative. This section describes the microlanguage constructs used to control this counter, which are basically mnemonics for the **LC_CTL** field, with the additional encoding of the microconstant for the load operation. These constructs are detailed in Figure 7-21.

Figure 7-21 Loop Counter Control Constructs

construct	encodings	
LD_LCNT(addr)	LC_CTL = 1	UCONST = addr
DEC_LCNT	LC_CTL = 2	

7.2.17 Context Scan Control Constructs

During fault (context save) and rtnc (context restore), the US microcode directs the NSP and NVP in context save and restore. Most of the NSP context is scanned out of gate arrays, merged in parallel on the X mux, and stored to memory. The CXSCAN construct, detailed in Figure 7-22, is used to control these operations. The NSP has two major context blocks, controlled with separate microinstruction fields. One block comes from the NRC, which requires special control to scan its RAMs. The other block contains everything else in the NSP which scans.

Figure 7-22 Context Scan Control Constructs

construct	description	encodings	
CXSCAN(HOLD,nrc)	hold non-NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 1
CXSCAN(RESET,nrc)	reset non-NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 2
CXSCAN(LEFT,nrc)	shift non-NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 3
CXSCAN(other,HOLD)	hold NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 1
CXSCAN(other,SAVE)	shift out NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 2
CXSCAN(other,RESTORE)	shift into NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 3
CXSCAN(other,RESET)	reset NRC	CX_OP_REQ = 1	MEM_OP_AT_TYPE = 4

7.2.18 Miscellaneous Control Constructs

There are a few encodings of the **USEQ_RAND** field that don't fit nicely in any other category of microlanguage construct. These miscellaneous constructs are described in this section.

When the NIP dispatches the NAS a trap under direction of the NCU (e.g. a patu microtrap), the US microcode acknowledges this trap with the **TRAP_COMPLETE** construct. In addition, for a ctrsg microtrap (global timer synchronization), microtrap complete must be cleared after it is detected (implying all CPUs have completed the timer update). This is achieved via the **CLR_MT_COMP** construct.

The US microcode uses the **DL_EN** construct on each cycle of a deadlock detect instruction (e.g. tas). The assertion of deadlock enable is combined with the deadlock signal from the NIP denoting that it's in a 1-instruction branch loop. The combination of deadlock and deadlock enable causes the NIP to dispatch a deadlock trap.

The miscellaneous control constructs are detailed in Figure 7-23.

Figure 7-23 Miscellaneous Control Constructs

construct	encoding
TRAP_COMPLETE	USEQ_RAND = 1
CLR_MT_COMP	USEQ_RAND = 4

DL_EN USEQ_RAND = 2

7.2.19 Wait Hazard Constructs

US microcode can directly assert a hazard into the operand access stage of the pipeline with the WAIT construct. This construct selects a condition such as **VP_IDLE** (vector processor idle) and feeds it directly into the UIR1 source hazard equations. This was designed in to avoid cumbersome microcode spin loops for commonly occurring spin conditions. The mnemonics for these wait hazard conditions may have an @ suffix, implying the complement of the condition is waited for. This makes all conditions read the same way, i.e. **WAIT(VP_IDLE)** means "wait until the vector processor is idle" and **WAIT(ST_PEND@)** means "wait until there are no stores pending". These constructs, detailed in Figure 7-24, are simple encodings of the **TSEL** field. A range of this field is used for wait hazards, while the other encodings are used for branch condition selects, described in section 7.2.20 on page 7-35.

Figure 7-24 Wait Hazard Constructs

construct	wait until..	encoding
WAIT(ST_PEND@)	there's no pending stores in memory or the data cache	TSEL = 11
WAIT(DC_IDLE)	the data cache is idle, i.e. has no pending requests	TSEL = 12
WAIT(REQ_PEND@)	there's no request pending in the NDC or memory and xbar	TSEL = 13
WAIT(XB_REQ_PEND@)	there's no request pending in the memory / xbar	TSEL = 17
WAIT(MM_IDLE)	memory is idle, i.e. there's no request in the NDC or memory / xbar and the return control queue is empty	TSEL = 13
WAIT(VP_IDLE)	the vector processor is idle	TSEL = 15
WAIT(FU_BUSY@)	the function units aren't busy	TSEL = 16
WAIT(PSW_HAZ@)	there's no PSW hazard	TSEL = 10

7.2.20 Microsequencer Test Condition Constructs

Due to timing constraints, the NAS microsequencer executes conditional branches through a two cycle sequence. The test condition is selected on one cycle, and the following cycle branches on the value of the selected test condition. This section describes the **TEST** construct, which is used to select the microsequencer branch condition. This construct encodes two fields - TSEL, the test

select, and **TPOL**, the test polarity. Every condition may selected in true or complement form. The complement form is obtained by adding an @ to the test condition mnemonic. For example, to branch if PSW<AC> is 1, use **TEST(AC)**. To branch if PSW<AC> is 0, use **TEST<AC@>**. The **TEST** construct is detailed in Figure 7-25. For brevity, only the positive polarity is listed. The only difference of the negative polarity is that **TPOL** is encoded 0 instead of 1.

Figure 7-25 Microsequencer Test Condition Construct

construct	test selected	encodings	
TEST(ALWAYS)	1, i.e. always true	TSEL = 00	TPOL = 1
TEST(AC)	PSW<AC>	TSEL = 01	TPOL = 1
TEST(SC)	PSW<SC>	TSEL = 02	TPOL = 1
TEST(IC)	USW<IC>	TSEL = 03	TPOL = 1
TEST(JC)	USW<JC>	TSEL = 04	TPOL = 1
TEST(KC)	USW<KC>, hazard check	TSEL = 05	TPOL = 1
TEST(FRL00)	PSW<FRL> = 00_2	TSEL = 06	TPOL = 1
TEST(FRL10)	PSW<FRL> = 10_2	TSEL = 07	TPOL = 1
TEST(FRL11)	PSW<FRL> = 11_2	TSEL = 08	TPOL = 1
TEST(VV)	CCR<VV>, vector valid	TSEL = 0A	TPOL = 1
TEST(RING0)	CPC<31:28> = 000_2	TSEL = 0B	TPOL = 1
TEST(TR_OR_SEQ)	PSW<TR> OR PSW<SEQ>	TSEL = 0C	TPOL = 1
TEST(TTC)	PSW<TTC>	TSEL = 0D	TPOL = 1
TEST(TIT)	PSW<TIT>	TSEL = 0E	TPOL = 1
TEST(KC_NH)	USW<KC>, no hazard check	TSEL = 0F	TPOL = 1
TEST(SPARE)	spare NUS test input	TSEL = 18	TPOL = 1
TEST(SCAN_SPARE2)	spare NUS scanned test bit 2	TSEL = 19	TPOL = 1
TEST(SCAN_SPARE1)	spare NUS scanned test bit 1	TSEL = 1A	TPOL = 1
TEST(SCAN_SPARE0)	spare NUS scanned test bit 0	TSEL = 1B	TPOL = 1
TEST(MT_COMP)	microtrap complete from NCU	TSEL = 1C	TPOL = 1
TEST(PTE_LVL_T)	CBUS<7>, i.e. level T bit in returning PTE2	TSEL = 1D	TPOL = 1
TEST(PTE1_VAL)	scratch RAM validity bit for last read from PTE1 cache	TSEL = 1E	TPOL = 1
TEST(LCNT_LT_0)	loop count negative	TSEL = 1F	TPOL = 1

7.2.21 Microsequencer Branch Control Constructs

The next set of constructs to be described are the microsequencer branch constructs, which control the next microaddress selection. These constructs encode the **BRTYPE** and in some cases **BRADDR** field. In addition to the next microaddress, these constructs also control operation of the two 4-entry deep microsequencer stacks - the microstack and microinterrupt stack.

Conditional operations are based on the test selected in the previous microinstruction with the **TEST** construct, described in section 7.2.20 on page 7-35. One possible operation of the microsequencer is to dispatch, i.e. accept the next instruction dispatch and begin execution at the appropriate entrypoint. There are both conditional and unconditional dispatch operations. Dispatch operations additionally encode the **CSDISP** field. These operations also encode the AG_SEL field to specify advanced effective address mode for the first microinstruction of the next macroinstruction (for more information see section 7.2.8 on page 7-22).

The microsequencer branch constructs are detailed in Figure 7-26. The *upc* referred to is the microprogram counter. The *ustack* and *uistack* denote the microstack and microinterrupt stack, respectively. The *test* notation refers to the test condition selected on the previous cycle. The $ symbol is used here (and is recognized by the microassembler) as the current microinstruction address.

Figure 7-26 Microsequencer Branch Constructs

construct	description	encodings
JMP(addr)	unconditional jump: upc = addr	BRTYPE = 0 BRADDR = addr
NOP	no-operation: upc = upc + 1	coded as JMP($+1)
CJMP(addr)	conditional jump: if test upc = addr else upc = upc + 1	BRTYPE = 8 BRADDR = addr
CALL(addr)	unconditional subroutine call: push(ustack) = upc + 1 upc = addr	BRTYPE = 1 BRADDR = addr
CCALL(addr)	conditional subroutine call: if test push(ustack) = upc + 1 upc = addr else upc = upc + 1	BRTYPE = 9 BRADDR = addr
RTN	unconditional return from subroutine: upc = pop(ustack)	BRTYPE = 2
CRTN	conditional return from subroutine if test upc = pop(ustack) else upc = upc + 1	BRTYPE = A
DISP	unconditional dispatch: upc = dispatch entrypoint + $C00_{16}$	BRTYPE = 3 CSDISP = 1 AG_SEL = 0
CDISP	conditional dispatch:	BRTYPE = B CSDISP = 1

```
                    if test                                  AG_SEL = 0
                      upc = entrypoint + C00
                    else
                      upc = upc + 1
RTNI                return from microinterrupt:              BRTYPE = 6
                      upc = pop(uistack)
POPI(addr)          pop microinterrupt stack and branch:     BRTYPE = 4  BRADDR = addr
                      bit bucket = pop(uistack)
                      upc = addr
```

7.3 Building US Microcode

This section briefly describes how the microcode is assembled and describes some conventions used in the source code. The intent here is not to make the reader become an expert with the CONVEX microassembler; rather to illuminate some non-straightforward aspects of it which may obscure understanding of the microcode. The microcode is built with the following steps:

- the standard Unix macro-preprocessor m4 (actually a local version by the diagnostic group called **diagm4**) is used to include all sources from a root file (us.m4list) and expand macros
- the Unix **expand** utility is used to transform all tabs to blanks
- a home-grown preprocessor **uproc** expands single source lines to multiple for different data types and inserts entrypoint / non-entrypoint address information
- **nuasm.new**, the CONVEX microassembler, assembles the source into .h format
- **hexlst** creates a hex listing (.hex) of the object
- **hsplitb** (a slight variation of htob) creates two b.out format files (.b), one for each RAM bank
- **mkmem** creates two memory image files (one for each RAM bank) used by the N2 simulator
- **csgen** uses the b.out format files (.b) and creates the loadable image (us.wcs) used by the control store loader on the service processor

Of these, only **diagm4** and **uproc** affect the appearance of the source file beyond the rules of the microlanguage defined here (except that the microassembler maintains a built in symbol $ which contains the current microinstruction address). The rules of the **m4** preprocessor can be obtained from Unix documentation. The **uproc** step bears some explanation here. The first feature is the @@1 symbol used as a branch address. Each entrypoint microinstruction (first microinstruction of a macroinstruction) must branch to exit entrypoint space. Rather than require the microprogrammer to insert a bunch of bogus branch labels like this:

```
        JMP(FOO);                                   "entrypoint instruction"
FOO:    REG(,T0,) XY(1,RFW) ADD(W) WRFX(W);         "1st non-ep inst"
```

uproc maintains the @@1 label that always is the next available non-entrypoint microaddress, allowing this notation:

```
        JMP(@@1);                                   "entrypoint instruction"
```

```
REG(,T0,) XY(1,RFW) ADD(W) WRFX(W);     "1st non-ep inst"
```

this requires uproc to insert some ORG and **SET** microassembler commands to maintain the @@1 symbol.

The second feature of uproc is the **%** symbol combined with the EP directive. Many macroinstructions differ only in data type, which would require multiple near-identical source lines to be maintained, like these two for the *ld.h effa,Ak* and *ld.w effa,Ak* instructions:

```
    ORG @LD.H_A;
REG(RKA,,) READ RSV(X,H) DISP;

    ORG @LD.W_A;
REG(RKA,,) READ RSV(X,W) DISP;
```

The @LD.H_A symbol is defined in a list of entrypoint addresses called ep.s, which was carried over directly from C2 with a few additions for the new privileged instructions. By convention all entrypoint symbols are the instruction name preceded with an @. The function of the microinstructions is not important here - note that the only differences are the H and W sizes. The EP directive may be used to define a list of n entrypoints of this type with a similar microinstruction that will generate n lines of source code. A **%** symbol in the line will be replaced with the distinguishing letter from the entrypoint name. This allows the above 2 source lines to be replaced with a single line:

```
    EP @LD.H_A, @LD.W_A;
REG(RKA,,) READ RSV(X,%) DISP;
```

The features of the uproc preprocessor are essentially transparent in the listing file (except for the repeated **ORG** and **SET** commands associated with the **@@1** label). They will be encountered when reading source code, however.

7.4 US Microcode Examples

This section presents some examples of US microcode accompanied with explanation. The examples were chosen to be representative of the basic operation of the NSP. They were also chosen to be short due to the changing nature of microcode. A long example in this document would be more likely to become inconsistent with the actual code and would probably cause more confusion than enlightenment. A deeper understanding of the NSP can be gained by reading through the complete microcode listing.

7.4.1 Basic Loads

The first example illustrates address generation and a basic memory operation. The *ld.b effa,Sk* instruction reads the contents of memory at *effa* into scalar register *Sk*. The microcode follows:

```
    EP @LD.B_S;
REG(RKS,,) READ RSV(X,B) DISP;
```

Recall that memory operations are initiated with a two cycle sequence. First, the address is

specified. On the following cycle the memory operation is specified. The *ld.b effa,Sk* instruction relies on an advanced effective address (effa) calculation based on Aj (the address register pointed to by the J register select in the opcode) to be performed by the previous microinstruction, which is automatic whenever a dispatch operation is performed. In fact, the microinstruction shown performs advanced effa for the next microinstruction. The **READ** requests a byte from the data cache (or memory in event of a miss). The result is to be loaded into the Rk-selected scalar register by reserving the backdoor for an X register selected write, which identifies **RKS** as the destination. The term *backdoor* is commonly used in referring to asynchronous operations such as memory and function unit requests - the backdoor interface as opposed to the "front door" path the integer ALU takes to the register file. The **DISP** completes the microinstruction, making the NAS available for another dispatch from the NIP.

7.4.2 Indirect Loads and Advanced Effective Address

Indirect loads are a bit trickier. Recall that an indirect load first uses the instruction's Aj-based effa to read the address of the operand rather than the operand itself. This requires two address generations, so there are two separate microroutines for direct and indirect loads. The indirect load microcode follows:

```
    EP @LD.B_S@
REG(T0,,T0) READ RSV(X,W) AG(Z) JMP(@@1);
REG(RKS,,)  READ RSV(X,B) DISP;
```

The first microinstruction uses the advanced effa and initiates a word load for the indirect address, much like the direct version's load of the byte data. The return data is destined for temporary register **T0**. This is then used as the address for the final byte read. Note that the first microinstruction reserves **T0** using the X select and also accesses it on the Z port for the address generator via the **AG(Z)**. This actually uses the result of the first read as the address for the second read. This is taking advantage of the fact that the address generator source hazard (**AG_SRC_HAZ** signal) for the first microinstruction is controlled by the **AG_SEL** field of the previous microinstruction (i.e. the last microinstruction of whatever macroinstruction preceded the ld.b effa,Sk). The **AG(Z)** in the first microinstruction will not cause an **AG_SRC_HAZ** until the second microinstruction reaches the UIR1 level. At that time, the hazard logic will recognize that the UIR2 level of the first microinstruction (specifically the **RSV(X,W)** with **XREG_SEL = T0**) is setting a hazard on **T0**, so the second microinstruction will stall until **T0** is written.

When the indirect address returns via the C bus (destined for **T0**), it will bypass from the C bus to the Z port of the register file and be used as the address for the second **READ**, at the same time it's actually written to **T0**.

Continuing to look at the first microinstruction, a branch out of entrypoint space to the second microinstruction is executed with the **JMP(@@1)**. The second microinstruction initiates the **T0**-based read of the byte data, reserves *Sk* for the return data, and dispatches.

The trick of combining the **AG(Z)** on a microinstruction that also modifies the register selected from the Z port is used frequently throughout the microcode. This is done to save control store space. The above sequence could have been written:

```
REG(T0,,) READ RSV(X,W) JMP(@@1);
REG(,,T0) AG(Z);
REG(RKS,,) READ RSV(X,B) DISP;
```

There are two reasons why this version would take the same amount of execution time as the first version. Recall that it takes a minimum of 2 clocks for the first load to return (if it's in the data cache). This means there are 2 clocks to waste before the memory operation can start, so wasting one by separating the **AG(Z)** doesn't hurt. Even ignoring this, both versions still take the same amount of time. This is because the **AG_SEL** for the second read must come from the microinstruction before the read, which could be the entrypoint instruction, as in the first version, or on the extra microinstruction in the second version. If the first read came back in 1 clock (which it can't, but assume so for purposes of illustration), there would still be a clock of address generation before the memory request could be started.

Whenever vector stride (VS) is changed, the new value is written to the NDC using the **WR_VS** construct. The data path for VS is the Z mux, so the advanced effa logic comes into play. The **WR_VS** can be considered as the memory operation (like **READ**) that follows the advanced effa. The **WR_VS** operations specifies the address generator hazard. Therefore, the correct sequence to transfer the contents ot T0 to VS would be:

```
REG(,,T0) AG(Z);
WR_VS;
```

7.4.3 Microsequencer Test and Branch

The next example illustrates the pipelined operation of the microsequencer branching logic. A conditional branch is a three step process:

- load the test condition (in some cases this is a pre-existing value so it doesn't have to be explicitly generated)
- select the test condition
- execute a conditional sequencing operation

The microcode for the inc.l Ceffa,Sk instruction illustrates this:

```
        EP @INC.L_S;
REG(T2,RKS,) XY(,RFL) INC RSV(X,L) CRSV(SC,CU) DL_EN JMP(@@1);
TEST(SC@) DL_EN;
DL_EN CDISP
REG(RKS,T2,) XY(RFL,RFL) ADD(L) LDSV WRFX(L) DL_EN DISP;
```

The first microinstruction places Sk (selected with **RKS** on the Y port) on the Y bus and initiates an **INC** communication register operation through the NDC subsystem. This relies on the Aj-based advanced effa to be used as the communication register virtual address. The **INC** operation will return the previous value of the lock bit, which is set up to be written to the **SC** bit of the PSW with the **CRSV(SC,CU)** construct. This also places a hazard on **SC** until the status information returns. The Y bus data (Sk) is sent via the crossbar to the NCU and used to increment the contents of the communication register at the physical address the NDC translated *Ceffa* to (using CIR and current execution ring). The NCU will return the old value of the target communication register, which is destined for **T2** due to the **RSV(X,L)** construct with the X register select pointing to **T2**. This will also set a hazard on **T2** until the data returns. The inc.l instruction is a deadlock detection instruction, so the DL_EN construct must appear on each microinstruction, starting with the first. If the NIP detects a deadlock condition, this will enable it and allow the deadlock condition of the current processor to be broadcast to the NCU.

At this point, as far as the NCU is concerned, the *inc.l* instruction is complete. If the target communication register is locked (i.e. full), it will use the Sk data to increment it. If it's unlocked (i.e. empty), nothing will be done to the communication register. The NSP must also make this decision and either increment Sk with the data returned to **T2**, or leave it alone. The second microinstruction begins this process by selecting the complement of **SC** as a test condition. Since the first microinstruction set a hazard on **SC**, the second will stall with a hazard until the **INC** status returns.

The third microinstruction conditionally dispatches if the value returned to SC is 0, i.e. if the target communication register was empty.

The fourth microinstruction, if executed, selects Sk and the returned communication register value in T2 as X and Y operands. It performs a longword add (**ADD(L)**), placing the result in Sk (**WRFX(L)**). Finally, the **DISP** ends the macroinstruction.

7.4.4 Vector Operations

The next example shows how the vector processor is controlled by the NSP. The interfaces here are quite simple and powerful - there really isn't a whole lot to this microcode. For example, consider *stvi.h.t Vk,Vj* (store vector of indices under mask true):

```
        EP @STVI.H_V;
REG(,,A5) AG(Z) DISPATCH_VP TEST(VV) JMP(@@1);
VWRITEM_IX(H) CDISP;
JMP(VVTRAP);
```

Due to its indexed nature, this instruction can't use advanced effa, so it starts by supplying the base address from **A5** on the Z port and using it as the address generator source (**AG(Z)**). The vector processor is conditionally dispatched with **DISPATCH_VP**. This dispatch is conditional because the staging logic that turns the microinstruction field into the dispatch handshake to the vector processor kills off the handshake if vector valid is zero. The first microinstruction also begins the test for vector valid with the **TEST(VV)** construct. If vector valid is clear, we must take a vector valid trap. Entrypoint space is exited with the **JMP(@@1)**.

The second microinstruction requests the indexed store under mask with the **VWRITEM_IX(H)** construct, where H specifies the halfword data type. The NDC takes over and controls the entire store operation. This memory request is also killed off if vector valid is clear, allowing us to optimize performance for the more likely vector valid set case, and get things started. We conditionally dispatch completing the instruction if vector valid is set.

The third microinstruction is only executed if vector valid is clear (i.e. the conditional dispatch failed). This simply jumps to the microcode vector valid trap handler, not detailed here.

7.4.5 PTE Miss Handler

The final example is the PTE miss microinterrupt routine. When a memory request to the NDC misses the PTE cache, it microinterrupts the NAS, forcing the microsequencer to begin executing a routine that begins at an address specified by the NDC. This address, called a microinterrupt vector, varies depending on the kind of memory fault the NDC has recognized. These vectors begin at address 000_{16} and form a "jump table" into a group of microinterrupt handlers. These service the various kinds of page faults as well as a simple PTE miss. For the PTE miss vector,

the microcode must fetch the correct PTE from main memory and write it in the PTE cache. The missing request is then retried, hitting the just-written PTE cache and continuing. Microcode for the PTE miss routine follows. The first microinstruction is located at address 001_{16} corresponding to the PTE miss microinterrupt vector. The second microinstruction shown continues the handler at the address label **PTE_MISS**.

```
        REG(,T7,) XY(,RFW) WR_RAM(MISS_T7) UINT_RESTART JMP(PTE_MISS);     1
PTE_MISS:
        REG(T7,,) XY(PTE1,) PASSX(W) WRFX(W);                              2
        TEST(PTE1_VAL@);                                                   3
        CJMP(LONG_MISS);                                                   4
        REG(,T7,) XY(,RFW) READ_PTE2 RSV(Y,W);                             5
MISS_END:
        REG(,T7,) XY(,RFW) WRITE_PTE2_CACHE;                               6
        TEST(PTE_LVL_T@);                                                  7
        CJMP(MISS_RETRY);                                                  8
        REG(,T7,) XY(,RFW) READ_PTET RSV(Y,W);                             9
        REG(,T7,) XY(,RFW) WRITE_PTET_CACHE;                              10
MISS_RETRY:
        REG(T7,,) XY(RAM(MISS_T7),) PASSX(W) WRFX(W) RETRY_REQ RTNI;      11

LONG_MISS:
        REG(T7,,) XY(SDR,) PASSX(W) WRFX(W);                              12
        REG(,T7,) XY(,RFW) READ_PTE1 RSV(Y,W);                            13
        REG(,T7,) XY(,RFW) WR_RAM(PTE1) READ_PTE2 RSV(Y,W) JMP(MISS_END);14
```

For convenience the microinstructions have been labelled on the right edge from 1-14. The purpose of this microroutine is as follows:

- fetch the first level PTE from the PTE1 cache in scratch RAM
- if the PTE1 isn't in the cache, fetch it from memory and write it in the PTE1 cache
- use the PTE1 to fetch the second level PTE from memory
- write the PTE cache with the fetched PTE2
- if the LT bit is set in the PTE2, indicating a thread level PTE exists, fetch the PTET and write it in the PTE cache
- retry the original request
- return from the microinterrupt

Microinstruction 1 starts by saving temporary register T7 in scratch RAM at the symbolically addressed location **MISS_T7**. This routine needs a temporary register, so by convention T7 is never used as the destination of a memory, function unit, or communication register operation, so there will never be a hazard on it. T7 is placed on the Y bus and written to RAM with the **WR_RAM(MISS_T7)** construct. The NIP is given a microinterrupt restart with the **UINT_RESTART** construct. This invalidates all instructions in the pipeline since if the NAS has accepted a dispatch, the microinterrupt caused it to be dropped. The vector table is exited with the **JMP(PTE_MISS)** construct.

Microinstruction 2 reads the scratch RAM PTE1 cache using the PTE1 offset bits of SA1_MISS1 (the address the NDC missed with) as the address. RAM data is placed on the X mux (**XY(PTE1,)**),

passed through the ALU and written to **T7**.

Microinstruction 3 selects as the test condition the complement of the validity bit from the purge RAM associated with the PTE1 cache read on the last cycle. Microinstruction 4 conditionally branches to **LONG_MISS** in the event the PTE1 was invalid, in order to fetch the correct PTE1.

If the PTE1 was valid, microinstruction 5 places the PTE1 in T7 on the Y bus and requests a PTE2 fetch from the NDC with the **READ_PTE2** construct. The NDC expects the PTE1 on the Y bus to supply the base address of the appropriate PTE2 table. It uses SA1_MISS1 internally to supply the PTE2 offset bits. The read is set up to return to **T7** with the **RSV(Y,W)** construct with **YREG_SEL = T7**.

If the PTE1 was invalid, we continue at microinstruction 12 by reading the SDR from scratch RAM. The segment bits from SA1_MISS1 are used to index into locations 0-7 in scratch RAM and read the SDR onto the X mux (**XY(SDR,)**). The SDRs are accelerated into these scratch RAM locations whenever the CIR changes or the SDRs are explicitly loaded (*ldsdr*). The SDR is passed through the ALU and written to **T7**. Microinstruction 13 places the SDR in **T7** on the Y bus and requests a PTE1 from the NDC. The SDR must be on the Y bus to supply the PTE1 table base address. The NDC uses the PTE1 offset bits from SA1_MISS1 in conjunction with this to access the correct PTE1. The PTE1 is set up to return to **T7**. Microinstruction 14 uses the PTE1 in **T7** to request the PTE2 in the same manner as if the PTE1 came from the scratch RAM cache (microinstruction 5). It is also written into the scratch RAM PTE1 cache by placing it on the Y bus and using the **WR_RAM(PTE1)** construct, which again uses SA1_MISS1 bits as part of the address. We then join with the shorter miss routine with the **JMP(MISS_END)** construct.

By the time we've reached microinstruction 6 (**MISS_END**), the PTE2 has been requested and reserved for **T7**. It is then placed on the Y bus and written to the PTE cache with the **WRITE_PTE2_CACHE** construct, setting non-thread validity. Microinstruction 7 selects as the test condition the complement of bit 7 of the registered C bus from the previous cycle, i.e. the returning PTE2 data. This is the **LT** bit of the PTE2, which tells us if the translation is complete. Microinstruction 8 conditionally jumps (**CJMP(MISS_RETRY)**) if **LT** is zero, meaning the translation is complete.

If **LT** was set, we don't take the branch and the translation continues at microinstruction 9. The PTE2 fetched into **T7** is placed on the Y bus and the thread level PTE is requested with the **READ_PTET**, returning it to **T7**. Microinstruction 10 takes the fetched PTET, places it on the Y bus and writes the PTE cache, using the **WRITE_PTET_CACHE** construct, setting thread validity.

At this point (MISS_RETRY), the translation is complete and correctly encached in both the scratch RAM PTE1 cache and PTE2 cache. Microinstruction 11 completes everything by restoring the pre-microinterrupt value of **T7** from scratch RAM, retrying the request with the NDC (RETRY_REQ) and returning from the microinterrupt (**RTNI**).

7.5 Microprogramming Rules

There are a few features of the NSP hardware controlled by microcode that are not straightforward. Most of these involve pipeline latencies that must be observed by not putting a read of a register too close to a write of the same register. In the descriptions, it's important to make the distinction of what a *cycle* is. This implies a complete microinstruction, not just an extra clock of execution time due to a hazard stall in the pipeline. There are no doubt more than are listed here - more insight can be gained by reading the microcode.

- Jump restarts of the NIP (**USRJMP**, SYSJMP, **UINT_RESTART**, **RTNC_RESTART**) must wait 3 cycles before dispatching to make sure the jump gets to the UIR2 level of the pipe before exiting the current macroinstruction. The following sequence is acceptable, where the comments show the pipe stage each microinstruction has reached when the dispatch reaches the UPC level:

```
USRJMP;        "at UIR2 level"
NOP;       "at UIR1_LAS level"
NOP;           "at UIR1 level"
DISP;          "at UPC level"
```

- Register references (**RSV**, WRFX, etc.) for longwords should use the lower register designator (i.e. **T0** not **T1**). Hazard checking logic in the NRFA will not catch a UIR1 access to T0 as a word as a hazard if UIR2 is reserving **T1** as a longword (e.g. with an A bus write)
- There must be 1 cycle between writing and reading the CCR. This is because it is written at the UIR2 level and read at the UIR1 level. This only applies if you want to read the value you just wrote - a read immediately following a write will read the old value of the register. This rule also implies the CCR may not be read on the first microinstruction of a macroinstruction, since the last microinstruction of the previous macroinstruction may have written the CCR.
- The PSW may not be read or written on the first microinstruction of a macroinstruction (actually the first 2 microinstructions, the second is covered in a later rule). In addition to the fact that, like the CCR, the PSW is written at UIR2 and read at UIR1, this is because there's a 1 clock delay between the initiation of a function unit operation and when it can affect the PSW hazard. If the last microinstruction of the previous macroinstruction started a function unit operation, we would miss any exception bit the function unit operation caused.
- **AIV** and **SIV** are calculated one stage after UIR2 ("UIR3") so you can't read the PSW correctly after an LDAV or **LDSV** operation. Two cycles must be between the LDxV and PSW read, to cover UIR3 write through UIR1 read. This means the PSW can't be read on the first 2 microinstructions of a macroinstruction.
- There must be 1 cycle between setting PSW<FRL> (**LDFRLxx**) and testing FRL (**TEST(FRLxx)**). Similarly, there must be 1 cycle between **LDFRLxx** and reading the PSW. This is because the **LDFRLxx** operations occur at the UIR2 level and test selection and PSW reads occur at the UIR1 level.
- There must be 2 cycles between writing and reading scratch RAM. This is because scratch RAM is written at the UIR2 level and addressing begins at the UPC level. This even applies if the read and write reference different addresses - UIR2 writing RAM has higher priority at the external RAM address mux, so the UIR2 write address is selected over the UPC-level read address. Actually, it is possible to follow a write immediately with a read (if the addresses are different), since the write will only have reached UIR1 when the read reaches UPC. The sequence write, other cycle, read is what must be avoided. To observe this rule between macroinstructions, it is convention that you must wait 2 cycles after a scratch RAM write before dispatching.
- The microsequencer loop counter may only be used in faults since it is not part of context state.
- The scratch RAM address counter may only be used in faults since it is not part of

context state. Additionally, it may generally only be used on writes. It doesn't work on reads if there's a microinterrupt, since it counts off the UIR2 level and scratch RAM is read at the UIR1 level. Therefore if the scratch RAM read is interrupted at the UIR1 level, the counter will increment and the wrong count will be used when the read is resumed at the UIR1 level after the microinterrupt.

- The delta timer in the NPSW array must be cleared and read on the same clock to insure accuracy. To do this code **CLR_TIMER** on 1 cycle followed by **XY(TIMER,)** on the next. This is because the timer is cleared from the UIR2 level and read at the UIR1 level.
- Data cache and PTE purges are considered writes by the NDC. They may not be followed immediately by a read, since they don't have address comparison checks performed on them in the NDC which would make the read see the purged value.case. Therefore, a MEM_NOP request must be made after each purge before the next load is seen. It is important to note this doesn't just mean the next cycle, rather the next load that my be many cycles later. This means if a purge is not followed by a write before the completion of a macroinstruction, a **MEM_NOP** must be added. Communication register operations count as the no-op, since they don't read the data cache.
- Test conditions that can cause a hazard between macroinstructions, namely AC and SC, may not be selected on the first microinstruction of a macroinstruction. This is a work around for a bug in the NUS array.
- The carry flag test conditions (**AC, SC, IC, JC, KC**) may not be loaded using the **CLD** microp and selected using the **TEST** microp on the same microinstruction. This is due to the way microinterrupts affect the sequencer pipeline. If a microinterrupt occurs when the microinstruction following the **TEST** select (i.e. the one doing the conditional branch based on the test), the **CLD** value will be used after the microinterrupt, which is not the case if no microinterrupt occurs. The microassembler has a check for this condition built into it, so it is impossible to violate this rule unknowingly.
- After receiving an NCU trap dispatch, at least six cycles must pass before **TRAP_COMPLETE** may be issued, to avoid a race condition in the NCU.
- The NCU may not receive a write to the local or global interrupt enable registers followed immediately by an **ENI**. If there is a chance of this occurring, the **ENI** operation must wait for **ST_PEND** to become false (to insure no communication register requests are in the pipeline), then wait one clock. Because of the semaphoring rules implicit in the use of the enable registers (i.e. you must **DSI** before modifying the local or global enable), the dangerous sequence (e.g. *enal - enl*) can only come from a single processor.
- The vector processor should not be dispatched for an instruction with a memory operation until hazards have cleared on any address registers that are required for the memory operation. Examples of these are indirect and indexed load/store.
- The Z mux port (for AG accesses) in the register file does not consistently implement longword to word bypasses and associated hazards. Therefore, the source register for an AG operation may not come from a longword write. For example, the sequence

```
REG(T0,A0,) XY(RFL,) PASSX(L) WRFY(L);
REG(,,A1) AG(Z);
```

would not result in the value copied from T1 being used as the AG source. This also implies that an address register pair may never be written as a longword on the last

microinstruction of a macroinstruction (since an odd address register may be used as an address source on the first microinstruction of another macroinstruction).

- Cache updates can interfere with single entry purge operations (**PATE**) on the PTE cache. Therefore, microcode must wait for **MM_IDLE** before issuing a **PATE** command.

7.6 The SR Microinstruction

The second, much simpler control store on the NSP is the scratch RAM, or SR. This is a 2K entry read/write RAM that is initialized via the control store loader to contain some constants used by the microcode. Scratch RAM is 32 bits wide, plus 4 bits of parity. It is implemented in 4 2K X 9 self-timed RAMs. This implementation leads to an organization of 1 odd parity bit for each 8 data bits.

The SR microinstruction fields are listed below, in bitwise order. They are not listed alphabeticall as the US microinstruction was, because on the board they are never referred to with these field names - just as SR_DATA<31..0> and SR_PAR<3..0>.

Field Name	Width	Pos.	Description
PAR_BYTE0	1	35	Odd parity for BYTE0 (data bits 31-24)
BYTE0	8	34-27	Data bits 31-24
PAR_BYTE1	1	26	Odd parity for BYTE0 (data bits 23-16)
BYTE1	8	25-18	Data bits 23-16
PAR_BYTE2	1	17	Odd parity for BYTE0 (data bits 15-8)
BYTE2	8	16-9	Data bits 15-8
PAR_BYTE3	1	8	Odd parity for BYTE3 (data bits 7-0)
BYTE3	8	7-0	Data bits 7-0

7.7 The SR Microlanguage

This section describes the constructs of the SR microlanguage, detailing which microinstruction fields are coded for each construct. This is a trivial microlanguage, actually just a single construct used to define the initial value for a scratch RAM location. The **RAM** construct takes a single parameter which is the 32-bit value to be assigned to the current "control store" location. The microassembler automatically calculates parity and assigns the parity bits in the "microinstruction". This construct is detailed in Figure 7-27. The notation x >> y means x shifted right y bits.

Figure 7-27 Scratch RAM Construct

RAM(value)	BYTE0 = (value >> 24) AND FF_{16}
	BYTE1 = (value >> 16) AND FF_{16}
	BYTE2 = (value >> 8) AND FF_{16}
	BYTE3 = value AND FF_{16}
	PAR_BYTE0 = odd_parity(BYTE0)
	PAR_BYTE1 = odd_parity(BYTE1)
	PAR_BYTE2 = odd_parity(BYTE2)
	PAR_BYTE3 = odd_parity(BYTE3)

7.8 Building SR Microcode

This section briefly describes how the SR microcode is assembled. There is one linkage with the US microcode - the predefined constant addresses. This is accomplished by using the same sr_def.h file in both Makefiles. This file defines a hex address for the symbolic addresses used in scratch RAM address references in US microcode and the assembly origination directives (ORGs) in the SR microcode. The SR microcode is built with the following steps:

- the standard Unix macro-preprocessor m4 (actually a local version by the diagnostic group called **diagm4**) is used to include all sources from a root file (sr.m4list) and expand macros
- the Unix **expand** utility is used to transform all tabs to blanks
- **nuasm.new**, the CONVEX microassembler, assembles the source into .h format
- **hexlst** creates a hex listing (.hex) of the object
- **htob** creates a b.out format file (.b)
- **mkmem** creates a memory image file used by the N2 simulator
- **csgen** uses the b.out format file (.b) and creates the loadable image (sr.wcs) used by the control store loader on the service processor

Of these, only **diagm4** affects the appearance of the source file beyond the rules of the microlanguage defined here (except that the microassembler maintains a built in symbol $ which contains the current microinstruction address). The rules of the **m4** preprocessor can be obtained from Unix documentation.

7.9 SR Microcode Examples

This section presents a few examples of SR microcode. The single parameter passed to the **RAM** construct is a 32-bit hex value (hex by convention - the H suffix on the constant identifies this to the microassembler). This hex value must be preceded with a 0 if the first character is non-numeric decimal (A-F). The first example initializes the location at symbolic location M16 with a mask for bit 16 in a 32-bit word:

```
ORG M16;
RAM(000010000H);
```

The final example illustrates one of the **m4** macros defined to specify intrinsic constants. This doesn't show up in the listing, since it's expanded out by then, but it should be useful to readers attempting to modify the microcode source. Recall that there is a special mapping function performed in the address range 200_{16}-$3FF_{16}$. The IEEE bit in the PSW is used as bit 8 of the address in this range, making addresses 200_{16}-$2FF_{16}$ contain Convex native mode values and 300_{16}-$3FF_{16}$ contain IEEE mode. This allows a single US microroutine to perform the major part of both IEEE and native mode intrinsic instructions, since the major difference is the format of the constants. This means when a single intrinsic constant address is referenced, there needs to be two values defined. The m4 macro called **lls** allows the definition of two constants simultaneously. There are three parameters to this macro. The first is the symbol address, defined for the native constant (in the range 200_{16}-$2FF_{16}$). The second parameter is the native mode format constant, and the third is the IEEE mode format constant. In the source sr.s, you will see something like this:

Appendix A Signal List

ACPEND

Signal from NPSW gate array indicating that a write to the AC bit of the PSW register is pending. Branches which rely on the AC bit of the PSW will be held off until this signal is deasserted.

AG_SRC_HAZ.0

Signal from NRFA gate array indicating that the location being read from the register file has a pending write hazard. The memory request which uses this address will be held off until the hazard has cleared.

ALIGN_CTRL_OUT<2..0>

Control signals driven from the master NRC gate array (slice zero) to the other two NRC gate arrays which controls the rotation of memory return data destined for the vector processor.

AR_TRAP

Signal from NPSW gate array to the NPAR gate array indicating that the PSW register has an arithmetic trap condition pending.

AS_DISP_BR_DISPL<7..0>

These signals are part of an instruction being passed from the NIP subsystem to the NAS subsystem. The signals are the branch displacement field for branch instructions.

AS_DISP_DISPL<31..0>

These signals are part of an instruction being passed from the NIP subsystem to the NAS subsystem. The signals contain an address offset for instructions requiring an effective address calculation or a constant for immediate instructions.

AS_DISP_DISPL_PAR<3..0>

Byte parity for AS_DISP_DISPL

AS_DISP_EP<9..0>

These signals are part of an instruction being passed from the NIP subsystem to the NAS subsystem. The signals are used as the initial location in control store to be executed for the instruction.

AS_DISP_IREG<2..0>

These signals are part of an instruction being passed from the NIP subsystem to the NAS subsystem. The signals contain the I register field.

AS_DISP_JREG<2..0>

These signals are part of an instruction being passed from the NIP subsystem to the NAS subsystem. The signals contain the J register field.

AS_DISP_KREG<2..0>

These signals are part of an instruction being passed from the NIP subsystem to the NAS subsystem. The signals contain the K register field.

AS_DISP_RDY

Handshake signal which indicates that a valid instruction is being passed from the NIP subsystem to the NAS subsystem.

AS_DISP_REQ

Handshake signal indicating that the NAS subsystem can accept an instruction from the NIP subsystem.

AS_DISP_SEL_IREG

Control signal from the NPAR gate array used to select whether AS_DISP_IREG or AS_DISP_KREG should be selected for MUNGED_DISP_KREG. The signal is decoded from AS_DISP_EP on the NPAR gate array.

AS_DISP_SIZE<1..0>

Signals from the NIP subsystem indicating the size in half words of the instruction being passed to the NAS subsystem.

AS_DISP_XTEND

This signal is part of an instruction being passed from the NIP subsystem to the NAS subsystem. It indicates that the instruction is an extended instruction.

AS_IDLE

Signal from the NAS subsystem indicating that it does not have an instruction executing in any stage of its pipe including the funciton units.

BBUS_DATA<63..0>

This bus is used to write results from the function units to the register file contained within the NRFA gate arrays.

BBUS_PAR<7..0>

Byte parity for BBUS_DATA.

BPC_CARRY_IN

Branch PC carry from the lower to upper NIAD gate arrays.

BPF_CYC1

Signal indicating that a block prefetch request is in the first stage of the data cache pipe.

BPF_EQL

Signal indicating that the request at the first stage of the data cache pipe has the same block address as the block prefetch address register. The block prefetch mechanism is not restarted when the block address is the same.

BPF_EQL0

BPF_EQL1

Signals from the lower (BPF_EQL0) and upper (BPF_EQL1) NAG gate arrays. The signals are combined to generate BPF_EQL.

BPF_INIT

The block prefetch mechanism is restarted when this signal is asserted and a read request misses the data cache.

BPF_ODD

Signal used to indicate the the current block prefetch request entering the first stage of the data cache pipe will be to the odd side of memory.

BP_SP.LBD_INTL_IN

This signal is used as a logic board interlock signal. The power pallet uses the signal to make sure the board is plugged into the backplane before power is applied.

BP_SP.PORTID<3..0>

These signals allow the power pallet controller to determine which port of the cross bar the board is connected.

BP_SP.SLOTID<3..0>

These signals allow the power pallet controller to determine which slot on the backplane the board is plugged into.

BRANCH

Signal from the NPAR gate array to the NIAD gate arrays indicating that a branch instruction was parsed.

BRANCH_ADDR

This address bit indicates which word of the branch history cache at the current long word address should be updated.

BRANCH_DATA_L

The lower bit of the branch history cache is written with this data.

BRANCH_DATA_H

The upper bit of the branch history cache is written with this data.

BRANCH_HIST_ADDR<10..0>

These signals are used as the address for the branch history cache. The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

BRANCH_HIST_DATA<1..0>

Branch history information for the instruction currently being rotated and parsed. The signals are driven by the NPAR gate array to the NIAD gate arrays.

BR_CARRY_IN

Carry bit from NIAD0 to NIAD1 when determining the next program counter address withing the NIADs.

BR_HIST_L<1..0>

Branch history cache read data for the lower word.

BR_HIST_U<1..0>

Branch history cache read data for the upper word

BR_HIST_WR_L*

Branch history cache write enable for the lower word rams.

BR_HIST_WR_U*

Branch history cache write enable for the upper word rams.

BR_HIST_XL<1..0>

Branch history cache read data for the lower word.

BR_HIST_XU<1..0>

Branch history cache read data for the upper word.

BR_RESTART

Signal from the NAS subsystem to the NIP subsystem indicating that the NIP incorrectly guested which way to follow a branch instruction. The signal causes the

NIP to flush all parsed instructions and restart the pipe following the opposite direction for the branch.

BR_SEL<1..0>

Branch select information for the instruction currently being rotated and parsed. The signals are driven from the NPAR gate array to the NIAD gate arrays.

0	Unconditional branch
1	Branch on ION (not supported)
2	Branch on AC of PSW register
3	Branch on SC of PSW register

BR_WR_CARRY_IN

Carry from NIAD0 to NIAD1 used to determine the correct address for restarting from an incorrect branch. This operation takes place at the UPC_DISP level.

B_X_BYPASS

Control signal used to instruct the function units to load the XMUX register from the function unit result bus (BBUS).

B_Y_BYPASS

Control signal used to instruct the function units to load the YMUX register from the function unit result bus (BBUS).

C.PTE_WE

This signal is UIR2_PTE_DATA_WE qualified with UIR2_VAL. The signal is fanned out to the PTE rams as the write enable signal.

CACHE_CARRY_IN

Carry from the lower NIAD to the upper NIAD used when adding one long word to the read address of the instruction cache, as in normal sequential instruction parsing.

CBUS_DATA<63..0>

Bus used to write data cache, memory, and vector processor data to the register file contained on the NRFA gate arrays.

CBUS_PAR<7..0>

Byte parity for CBUS_DATA.

CBUS_REG_SEL<4..0>

These signals are used to select which register within the register file should be written from the CBUS.

CBUS_SIZE<1..0>

Signals indicating the size of the data on the CBUS.

0	Byte
1	Half Word
2	Word
3	Long Word

CBUS_WR_EN.0

Signal indicating that valid data is on the CBUS and will be written to the register file this cycle.

CBUS_WR_EN0*

CBUS_WR_EN1*

Signals from NPA0 (CBUS_WR_EN0*) and NPA1 (CBUS_WR_EN1*) gate arrays indicating that either data wasn't needed or was present in the data cache.

CBUS_XS_EN*

Signal from NDC gate array indicating that data from either memory or the vector processor is on the CBUS and is to be written to the register file.

CCR_1PG_CACHE

Signal from the CCR register in the NPSW gate array indicating that the data cache is to be reduced in usable size to one page (4096 bytes).

CCR_HALT

Signal from the CCR register in the NPSW gate array used to send a signal to the NCU board to stop the clock cycle counter.

CCR_PURGE_ICACHE

Signal from the CCR register in the NPSW gate array used to instruct the NIP subsystem to purge the instruction cache.

CCR_VV

Signal from the CCR register in the NPSW gate array which is used to allow the execution of vector instructions.

CK_PHASE.1

Phase signal used to control the double cycled data cache rams.

CK_PHASE.3.0

Phase signal used to control the double cycled instruction cache rams.

CLKLEDCLR

Clear signal from power pallet which resets the scan control and clock test logic.

CLKLEDON

Output from the scan control and clock test logic.

CNTX_NORMAL

Control signal for NIP subsystem logic which inhibits writes and purges of cache rams when deasserted.

CPC_RING0

Signal from the NPSW gate array indicating that the current PC is in ring zero. The signal is used by the NUS gate array as a test condition.

CRAT_CYC1

Signal from the NDC gate array indicating that the request in the first stage of the data cache pipe requires communication register address translation (CRAT).

CRD_DATA0<35..0>

Even memory data cache read data and parity. Bits <35..32> are byte parity for bits <31..0>.

CRD_DATA1<35..0>

Odd memory data cache read data and parity. Bits <35..32> are byte parity for bits <31..0>.

CRD_TAG0<35..0>

Even memory data cache tag read data and parity. Bits<35..32> are byte parity for bits <31..0>. The tag is composed of a page address (bits <31..12>), the CIR (bits <10,7..4>), and zone validity (bits <3..0>).

CRD_TAG1<35..0>

Odd memory data cache tag read data and parity. Bits<35..32> are byte parity for bits <31..0>. The tag is composed of a page address (bits <31..12>), the CIR (bits <10,7..4>), and zone validity (bits <3..0>).

CRD_TVAL0<1..0>

Even memory data cache thread validity read data and parity. Due to the purge feature of the ram even parity is used.

CRD_TVAL1<1..0>

Odd memory data cache thread validity read data and parity. Due to the purge feature of the ram even parity is used.

CRD_UPD0<5..0>

Even memory data cache update tag read data and parity. Bit <5> is the parity for bits <4..0>.

CRD_UPD1<5..0>

Odd memory data cache update tag read data and parity. Bit <5> is the parity for bits <4..0>.

CRD_VAL0

Even memory data cache non-thread validity read data and parity. Due to the purge feature of the ram even parity is used.

CRD_VAL1

Odd memory data cache non-thread validity read data and parity. Due to the purge feature of the ram even parity is used.

CSADDR<10..0>

CSADDR2<10..0>

Address used for accessing the control store rams.

CSBANK

CSBANK*

Control signal for enabling either the upper or lower bank of control store rams.

CURRENT_COMP

Communication bit from NIAD1 to NIAD0 used during restarts of the instruction processor to indicate that the restart address does not compare to the current look ahead address in that array. Each array combines its compare signal with the one from the other array to determine proper action based on the complete address compare.

CURRENT_COMP_L

Communication bit from NIAD0 to NIAD1. See CURRENT_COMP.

CU_SP.CLOCK_2X

CU_SP.CLOCK_2X*

Differential signals which supply the clock to the NSP board.

CWR_DATA0<35..0>

Even memory data cache write data and parity. Bits <35..32> are byte parity for bits <31..0>.

CWR_DATA1<35..0>

Odd memory data cache write data and parity. Bits <35..32> are byte parity for bits <31..0>.

CWR_UPD<5..0>

Even and odd memory data cache update tag write data and parity. Bit <5> is parity for bits <4..0>.

CWR_ZONE0<4..0>

Even memory data cache zone write data and parity. The zone bits are written as part of the data cache tag information. Bit <4> is parity for bits <3..0>.

CWR_ZONE1<4..0>

Odd memory data cache zone write data and parity. The zone bits are written as part of the data cache tag information. Bit <4> is parity for bits <3..0>.

DCD0_CKE*

Even memory data cache data ram clock enable (active low).

DCD0_CS*

Even memory data cache data ram chip select (active low).

DCD0_WE*

Even memory data cache data ram write enable (active low).

DCD0_WR_PAR_ERR<3..0>

Even memory data cache data ram write parity error.

DCD1_CKE*

Odd memory data cache data ram clock enable (active low).

DCD1_CS*

Odd memory data cache data ram chip select (active low).

DCD1_WE*

Odd memory data cache data ram write enable (active low).

DCD1_WR_PAR_ERR<3..0>

Odd memory data cache data ram write parity error.

DCT0_CKE*

Even memory data cache tag ram clock enable (active low).

DCT0_CS*

Even memory data cache tag ram chip select (active low).

DCT0_WE*

Even memory data cache tag ram write enable (active low).

DCT0_WR_PAR_ERR<3..0>

Even memory data cache tag ram write parity error.

DCT1_CKE*

Odd memory data cache tag ram clock enable (active low).

DCT1_CS*

Odd memory data cache tag ram chip select (active low).

DCT1_WE*

Odd memory data cache tag ram write enable (active low).

DCT1_WR_PAR_ERR<3..0>

Odd memory data cache tag ram write parity error.

DCU0_CS*

Even memory data cache update tag chip select (active low).

DCU0_WR_PAR_ERR<3..0>

Even memory data cache update tag write parity error.

DCU1_CS*

Odd memory data cache update tag chip select (active low).

DCU1_WR_PAR_ERR<3..0>

Odd memory data cache update tag write parity error.

DC_BPF_REQ

Signal indicating the block prefetch mechanism has a memory request ready. The signal is used in arbitrating which request will be selected to the first stage of the data cache pipe.

DC_CIR<4..0>

These signals are the communication index register which is part of the data and PTE cache tag information.

DC_CIR_PAR

Parity for DC_CIR<4..0>

DC_CYC_MISS

Signal indicating that a data cache read resulted in a miss. The signal is from the second stage of the data cache pipe.

DC_HAZ

Signal from the NDC gate array indicating that a condidtion exists which requires holding instructions at the UIR1 stage in the NAS. Conditions which will cause DC_HAZ to be asserted are unable to dispatch the vector processor, unable to load the VL or VS registers due to a VAG request, a sequential store hazard, unable to complete a scalar to vector data transfer, or unable to accept an additional memory request.

DC_HIT0

DC_HIT1

Signal indicating that a data cache read was succuessful in obtaining the needed data from the data cache on the even side (DC_HIT0) and on the odd side (DC_HIT1).

DC_IDLE

DC_IDLE_HAZ

Signals indicating that the NDC does not have any active requests in its pipe

stages. The DC_IDLE_HAZ signal is used by the NUS gate array as a hazard condition. The signal DC_IDLE is a registered version of DC_IDLE_HAZ and is used by the NIP when deciding to restart instruction look ahead prefetches.

DC_IP_REQ

Signal indicating that an instruction prefetch request is ready. The signal is used in arbitrating which request will be selected to the first stage of the data cache pipe.

DC_MEM_OP1<9..3>

Signals used to specify the type of memory request at the first stage of the data cache pipe. Refer to field MFP_OP of the Micro Code section for encodings.

DC_RD_REQ

Signal from the first stage of the data cache pipe indicating a VAT (virtual address translation) read request is to be performed.

DC_REQ_VALID

Signal from the first stage of the data cache pipe indicating a valid request.

DC_REQ_VALID_CND

Signal from the first stage of the data cache pipe indicating that the request is conditionally valid depending on the state of the UIR interface. DC_REQ_VALID_CND and DC_REQ_VALID_UNC are used to generate a request valid signal internal to gate arrays.

DC_REQ_VALID_UNC

Signal from the first stage of the data cache pipe indicating that the request is unconditionally valid. DC_REQ_VALID_CND and DC_REQ_VALID_UNC are used to generate a request valid signal internal to gate arrays.

DC_SEG1<2..0>

Signals from the first stage of the data cache pipe specifing the segment the request is being made in. The segment is used for ring violation checking.

DC_SEL0*

DC_SEL1*

Signals generated by the NPA0 (DC_SEL0*) and NPA1 (DC_SEL1*) gate arrays used to disable writes to the data cache data and tag rams. Writes are disabled due to the cache being turned off, scan control not in normal mode, context control in context hold mode, or the normal operation of the data cache does not require a write.

DC_SIZE1<1..0>

Signals from the first stage of the data cache pipe specifing the size of the request being made. The normal size encoding is used (0-byte, 1-half word, 2-word, 3-long word).

DC_STATE<3..0>

These signals represent the state of the UIR interface state machine. The state machine handles the flow of requests from the NAS subsystem interface (UIR1_VAL, UIR2_VAL, AG_SRC_HAZ, BR_RESTART) to the arbitration logic for the data cache pipe.

DC_TVAL_PE*

Signal used to enable purging the data cache thread valid rams (active low).

DC_UNLW_REQ_CND

Signal from the first stage of the data cache pipe indicating that an unaligned long word request is in the pipe. The request is conditional on DC_REQ_VALID. Provided the request is valid the next request for the data cache pipe will be the second request of the long word access.

DC_USEL0*

DC_USEL1*

Signal generated by the NPA0 (DC_USEL0*) and NPA1 (DC_USEL1*) gate arrays used to disable writes to the data cache update rams. Writes are disabled due to the cache being turned off, scan control not in normal mode, context control in context hold mode, or the normal operation of the data cache does not require a write.

DC_VAL_PE*

Signal used to enable purging the data cache non-thread valid rams (active low).

DEST_IN<2..0>

Control signals driven from the master NRC gate array (slice zero) to the other two NRC gate arrays which specifies the destination for the data corresponding to the next entry in the memory return control queue.

EX_CYC1

Signal from the first stage of the data cache pipe indicating that the request is an instruction look ahead request

FAR_ENOUGH_AHEAD_L

FAR_ENOUGH_AHEAD_H

Signals from NIAD0 (L) and NIAD1 (H) gate arrays used to stop look ahead memory requests when the look ahead mechanism has prefetched to the look ahead limit.

FC_CARRY_IN

Carry bit from NIAD0 to NIAD1 used when adding the current instruction cache read address to the programmed prefetch limit which will be compared to the look ahead address to determine if the look ahead mechanism has reached the look ahead limit.

FRL_EQ_00

FRL_EQ_10

FRL_EQ_11

Decode from the PSW register within the NPSW gate array of the frame length bits. The signals are used as test conditions on the NUS gate array.

FU_BUSY

Signal from NPSW gate array indicating that one of the function units is busy. The signal is generated by using the function unit result control queue empty signals.

FU_RSLT_EN

Signal used to enable a write to the register file on the NRFA gate arrays from the BBUS.

FU_RSLT_SEL<1..0>

These signals specify which function unit is driving a result on the BBUS. The NRFA gate arrays use the signals to select which function unit result control queue to obtain the register to write the data to.

FU_SET_ADZ

FU_SET_AIV

FU_SET_FDZ

FU_SET_FSN

FU_SET_OV

FU_SET_RO

FU_SET_SDZ

FU_SET_SIV

FU_SET_UN

These signals are used to set the arithmetic exception bits in the PSW register within the NPSW gate array. The signals are generated by or'ing the exception bits from the scalar and vector processor function units.

FU_STATUS

Result of floating point compare operations performed on the NFAD function unit.

GA0_PAR_ERR<7..0>

GA1_PAR_ERR<7..0>

GA2_PAR_ERR<7..0>

These signals are NRC gate array partial parity. Each of the three NRC gate arrays generates partial parity for its data and drives it to the other two NRC gate arrays. The parity from each NRC gate array is four bits wide and is driven out twice (bits <7..4>, and <3..0>).

GA_SCAN_CTL<1..0>

Scan control signals for all gate arrays on the NSP board.

0	Normal mode
1	Not used
2	Cast mode
3	Left shift mode

HALT.0

Signal from the NSP_CLK logic indicating an error was detected on the board. The errors which can set HALT are gate array parity errors and ram write parity errors.

HELD_SRADDR<10..0>

These signals are the registered version of the address which was previously used to access the scratch rams. This address is used to reaccess the ram when the UIR1 level of the pipe is being held.

HOLD_QUEUE

Used in NIADs to hold the address to instruction cache if the current access is not valid, and used in NPAR as an indicator of data validity. See also QUEUE_INVALID.

HUNG_CNT<14..0>

The hang counter is used to count the number of cycles since the last instruction transfer from the NIP subsystem to the NAS subsystem (AS_DISP_RDY and AS_DISP_REQ).

HUNG_TC<1..0>*

These signals are the hang counter termination count indicators. When the hang counter chips reach their maximum count value the TC outputs are asserted (active low).

HW_HUNG

Signal indicating that the 16 bit hang counter has reached its maximum value.

IALU_LGENOUT<3..0>

IALU_LPROPOUT<3..0>

The NRFA gate arrays implement the integer alu as eight 8-bit slices. IALU_LGENOUT represents the carry generate values and IALU_LPROPOUT are the carry propagate values for the lower four byte slices.

IALU_OPSTAT1.0

?????

IALU_STAT0<2..0>

?????

IALU_STAT1<3,1..0>

?????

IALU_UGENOUT<3..0>

IALU_UPROPOUT<3..0>

The NRFA gate arrays implement the integer alu as eight 8-bit slices. IALU_UGENOUT represents the carry generate values and IALU_UPROPOUT are the carry propagate values for the upper four byte slices.

ICAC_DATA_OUT<71..0>

Instruction cache data ram data and parity. Bits <71..64> are byte parity for bits <63..0>.

ICD_WR_PAR_ERR<7..0>

Instruction cache data ram write parity errors.

ICT_WR_PAR_ERR<2..0>

Instruction cache tag ram write parity errors.

IDLE_TRAP

Signal from the CCR register of the NPSW gate array used to inform the NIP subsystem to dispatch an idle trap.

INST_ADDR_PAR<2..0>

Parity for INST_CACHE_ADDR<31..14>. The parity is written with INST_CACHE_ADDR bits <31..14> to the instruction cache tag rams.

INST_CACHE_ADDR<31..1>

Instruction cache address which is used for cache read and write accesses (bits <13..3>), comparing the instruction cache tag (bits <31..14>). The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

INST_VALID<1..0>

Instruction cache validity ram read data and parity.

INT_BR_RESTART

Signal from the NPAR gate array to the NIAD gate arrays informing them to restart the accesses to the instruction cache rams to follow the branch path of a branch instruction.

IP_REQ_INH

Signal from the NUS gate array to the NDC gate array inhibiting instruction look ahead requests. Refer to field IPINH of the Micro Code section for more information.

IXAQ_RPTR<1..0>

These signals are the read pointer for the instruction look ahead queue of the memory request arbitration mechanism

0	Read queue location zero
1	Read queue location one
2	Read queue location two
3	Unused decode

IXAQ_WPTR<1..0>

These signals are the write pointer for the instruction look ahead queue of the memory request arbitration mechanism.

0	Write queue location zero
1	Write queue location one
2	Write queue location two
3	Write inhibit (queue full)

IXA_ADDR<31..0>

Instruction look ahead address from NIAD gate arrays of the NIP subsystem to the NAG gate arrays. The address is enqueued in the IXA queue until the request wins arbitration to enter the data cache pipe.

IXA_ADDR_PAR<3..0>

Byte parity for IXA_ADDR.

IXA_NOFLT

Signal passed from NIP subsystem to NDC subsystem in an IXA transfer indicating that neither a memory or access violation fault should result from this instruction look ahead request.

IXA_RDY

IXA_REQ_NEXT

Handshake signals used for IXA transfers from the NIP subsystem to the NDC subsystem.

LA0<2..0>

The least significant three bits of the logical address addr, generated by the least significant NAG gate array. The NAG gate arrays use bit <2> for the SA address selection muxes and the NDC gate array uses bits <1..0> for unaligned long word checking.

LA0_CRY

This signal is the carry from the least to the most significant NAG gate arrays for the logical address adder.

LA0_OVR<2..0>

The least significant three bits of the logical address overrun register from the least significant NAG gate array. The NAG gate arrays use bit <2> for the SA address selection muxes and the NDC gate array uses bits <1..0> for unaligned long word checking.

LA1_CRY<1..0>

These signals are the carrys from the least to the most significant NAG gate arrays for the logical address plus eight adders.

LAT_WR_PAR_ERR<2..0>

Look ahead address cache tag write parity errors.

LBD_OVT1

LBD_OVT1_GND

LBD_OVT2

LBD_OVT2_GND

LBD_OVT3

LBD_OVT3_GND

Power pallet signals used to supply ground and monitor the output voltage of the temperature sensors on the board.

LOOK_ADDR_PAR<2..0>

Byte parity for LOOK_AHEAD_ADDR.

LOOK_AHEAD_ADDR<31..3>

Look ahead cache address which is used for cache read and write accesses (bits <13..3>), and writing the look ahead cache tag (bits <31..14>).

LOOK_AHEAD_VALID<1..0>

Look ahead cache validity ram data and parity.

LOOK_CARRY_IN

Look ahead adder carry from least to most significant NIAD gate arrays.

L_IXA_RDY

This signal is from the least significant NIAD gate array and is combined with U_IXA_RDY from the most significant NIAD gate array to generate IXA_RDY.

L_US_ABUS_FMT<1..0>

L_US_AGZ_HAZEN

L_US_BDOP<1..0>

L_US_BDREG_FMT<1..0>

L_US_BRADDR<11..0>

L_US_BRTYPE<3..0>

L_US_CSDISP

L_US_XREG_SEL<5..0>

L_US_YREG_SEL<5..0>

L_US_ZREG_SEL<5..0>

Control store field signals from the lower bank of control store rams. These signals are or'ed with similarly named fields from the upper bank and are then distributed to various gate arrays on the board. Refer to the Micro Code section for encoding details.

MI_HOLD

Memory interface hold signal. The signal is used to hold the data cache pipe stages when the memory system or return control stops accepting requests.

MUNGED_DISP_KREG<2..0>

These signals are either the I or K register select fields of an instruction and are used by the NRFA gate arrays to access the register file.

MXI_ADDR<31..0>

These signals are the logical address of the data being transferred from the NRC subsystem to the NIP subsystem.processor.

MXI_ADDR_PAR<3..0>

Byte parity for MXI_ADDR.

MXI_DATA<63..0>

Memory return data being transferred from the NRC gate arrays to the NIP subsystem instruction cache rams.

MXI_PAR<7..0>

Byte parity for MXI_DATA.

MXI_RDY

Signal from the master NRC gate array (slice zero) to the NIP subsystem indicating a transfer of memory data to the instruction cache rams is ready.

MXSQ_RPTR<1..0>

Memory to scalar processor queue read pointer. Transfers of memory data from the NRC gate arrays are queued in the NDP gate arrays when the data can not be transferred directly to the NRFA gate arrays register file.

0	Read queue location zero
1	Read queue location one
2	Read queue location two
3	Unused decode

MXSQ_WPTR<1..0>

These signals are the write pointer for the memory to scalar processor queue.

0	Write queue location zero
1	Write queue location one
2	Write queue location two
3	Write inhibit (queue full)

MXS_PTE_EVEN

Signal indicating that a memory to scalar processor transfer is for an even memory side page table entry.

MXS_PTE_ODD

Signal indicating that a memory to scalar processor transfer is for an odd memory side page table entry.

MXS_RDY

Handshake signal from the master NRC gate array (slice zero) to the NDC gate array for a memory to scalar processor transfer.

MXS_REG_SEL<4..0>

Memory to scalar processor control signals specifing which register file location within the NRFA gate arrays to write.

MXS_REQ_NEXT

Handshake signal from the NDC gate array to the master NRC gate array (slice zero) for a memory to scalar processor transfer.

MXS_SIZE<1..0>

Memory to scalar processor control signals specifing the size of data being transferred. The normal size encoding is used (0-Byte, 1-Halfword, 2-Word, 3-Longword).

NAG0_PAR_ERR

NAG1_PAR_ERR

NAG gate array lower and upper parity error signals.

NAS_CNTX<6..0>

Context from the NAS subsystem gate arrays.

NDC_CNTX<19..0>

Context from the NDC subsystem gate arrays.

NDC_PAR_ERR

Parity error signal from the NDC gate array.

NDIV_AIV

NDIV_FDZ

NDIV_FSN

NDIV_OV

These signals are the exception conditions generated by the NDIV function unit. The signals are used to set bits of the PSW register within the NPSW gate array.

NDIV_PAR_ERR

Parity error signal from the NDIV function unit.

NDIV_RDY_NEXT

Signal from the NDIV function unit indicating that it will be able to accept a new operation on the next cycle.

NDIV_RO

This signal is an exception condition generated by the NDIV funciton unit. The signal is used to set the RO bit of the PSW register within the NPSW gate array.

NDIV_RSLT_NEXT

Signal from the NDIV function unit indicating that it will be able to drive a result on the BBUS on the next cycle.

NDIV_RSLT_SEL

Signal from the BBUS arbitration PALs selecting the NDIV function unit to drive the

BBUS on the next cycle.

NDIV_SDZ

NDIV_SIV

NDIV_UN

These signals are the exception conditions generated by the NDIV function unit. The signals are used to set bits of the PSW register within the NPSW gate array.

NDP0_PAR_ERR

NDP1_PAR_ERR

NDP2_PAR_ERR

Parity error signals from the NDP gate arrays.

NEED0*

NEED1*

Signals from the NPA0 and NPA1 gate arrays indicating that data was required from the even/odd side of the data cache.

NFAD_OV

This signal is an exception condition generated by the NFAD function unit. The signal is used to set the OV bit of the PSW register within the NPSW gate array.

NFAD_PAR_ERR

Parity error signal from the NFAD function unit.

NFAD_RO

This signal is an exception condition generated by the NFAD function unit. The signal is used to set the RO bit of the PSW register within the NPSW gate array.

NFAD_RSLT_OE

Signal from the BBUS arbitration PALs selecting the NFAD function unit to drive the BBUS on the next cycle. The NFAD function unit is selected to drive the BBUS by default so that good parity is always present.

NFAD_RSLT_SEL

This signal selects the NFAD to drive a result to the BBUS on the next cycle.

NFAD_UN

This signal is an exception condition generated by the NFAD function unit. The signal is used to set the UN bit of the PSW register within the NPSW gate array.

NIAD0_PAR_ERR

NAID1_PAR_ERR

Parity error signals for the NIAD gate arrays.

NIP_CNTX<3..0>

Context from the NIP subsystem gate arrays.

NMISC_OV

This signal is an exception condition generated by the NMISC function unit. The signal is used to set the OV bit of the PSW register within the NPSW gate array.

NMISC_PAR_ERR

Parity error signal from the NMISC function unit.

NMISC_RO

This signal is an exception condition generated by the NMISC function unit. The signal is used to set the RO bit of the PSW register within the NPSW gate array.

NMISC_RSLT_NEXT

Signal from the NMISC function unit indicating that it will be able to drive a result onto the BBUS on the next cycle.

NMISC_RSLT_SEL

Signal from the BBUS arbitration PALs selecting the NMISC function unit to drive the BBUS on the next cycle.

NMISC_SIV

NMISC_UN

These signals are exception conditions generated by the NMISC function unit. The signals are used to set bits of the PSW register within the NPSW gate array.

NMUL_AIV

NMUL_OV

These signals are exception conditions generated by the NMUL function unit. The signals are used to set bits of the PSW register within the NPSW gate array.

NMUL_PAR_ERR

Parity error signal from the NMUL function unit.

NMUL_RO

This signal is an exception condition generated by the NMUL function unit. The signal is used to set the RO bit of the PSW register within the NPSW gate array.

NMUL_RSLT_NEXT

Signal from the NMUL function unit indicating that it will be able to drive a result onto the BBUS on the next cycle.

NMUL_RSLT_SEL

Signal from the BBUS arbitration PALs selecting the NMUL function unit to drive the BBUS on the next cycle.

NMUL_SIV

NMUL_UN

These signals are exception conditions generated by the NMUL function unit. The signals are used to set bits of the PSW register within the NPSW gate array.

NOFLT_CYC1

Signal from the first stage of the data cache pipe which specifies that the request is a non-faulting instruction look ahead request.

NOFLT_REQ_INH

NOFLT_REQ_INH0

NOFLT_REQ_INH1

Each NPA gate arrays generates a signal specifing that a non-faulting instruction look ahead request which was being retried had a PTE miss or access violation. The two signals are or'ed together (NOFLT_REQ_INH) and used to tell the NDC

gate array to drop the request.

NPA0_PAR_ERR

NPA1_PAR_ERR

Parity error signals from the NPA gate arrays.

NPAR_ADDR<1..0>

Instruction cache read address used to inform the NPAR gate array of the instruction allignment for the instruction cache read data.

NPAR_PAR_ERR

Parity error signal from the NPAR gate array.

NPC_CARRY_IN

Carry from NIAD0 to NIAD1 used in determining the restart program counter for microinterrupted instructions. This operation takes place at the UPC_DISP level.

NPSW_HARD_ERR

Hard error signal from the NPSW gate array.

NRC0_PAR_ERR

NRC1_PAR_ERR

NRC2_PAR_ERR

Parity error signals from the NRC gate arrays.

NRC_CNTX<29..0>

Context data from the NRC gate arrays

NRC_CNTX_CTL<2..0>

Context control signals for the NRC gate arrays. The NRC gate arrays require special modes to save and restore the rams in the gate arrays.

0	Context normal mode.
1	Context hold mode.
2	Context save mode.
3	Context load mode.
4,5,6,7	Context reset mode.

NRC_RAM_SCAN_CTL

Scan control signal for ram macros contained in the NRC gate array. The rams shift and execute the cycle after this scan control signal is deasserted (similar to the discrete self timed rams).

NRFA0_PAR_ERR

NRFA1_PAR_ERR

NRFA2_PAR_ERR

NRFA3_PAR_ERR

Parity error signals for the NRFA gate arrays.

NSP_CNTX_CTL.0<1..0>

Context control signals for all gate arrays except the NRC gate arrays.

0	Context normal mode.
1	Context hold mode.
2	Context reset mode.
3	Context left mode.

NUS_PAR_ERR

Parity error signal for the NUS gate array.

OUTPUT_SELECT<2..0>

From the NPAR to the NIADs, used to select the proper level of the pre-parsed instruction output queue to be asserted at the AS_DISP level from the NPAR and the UPC_DISP level from the NIADs.

PARSE_INST_TAG<34..14>

Instruction cache tag ram read data. Bits <34..32> are byte parity for bits <31..14>.

PARSE_LOOK_TAG<34..14>

Look ahead cache tag ram read data. Bits <34..32> are byte parity for bits <31..14>.

PARSE_BR_DISP<7..0>

The branch displacement from the most recently parsed instruction from the NPAR to the NIADs to be used in determining the next program counter address.

PARSE_BR_POL

From the NPAR to the NIADs, bit <8> from the actual parsed instruction used in a conditional branch to indicate whether the branch should take place on condition true(1) or false(0).

PARSE_SIZE<1..0>

The size of the most recently parsed instruction from the NPAR to the NIADs to be used in determining the next program counter address.

PARSE_XTEND

Indicates whether the most recently parsed instruction in the NPAR was an extended instruction. See also PARSE_SIZE.

PHASE_GEN_1X1P<3..0>

Signals from the phase generation registers for the first phase single cycle clocks.

PHASE_GEN_1X2P<3..0>

Signals from the phase generation registers for the second phase single cycle clocks.

PHASE_GEN_2X<3..0>

Signals from the phase generation registers for the double cycle clocks.

PPC_SPI_CLK

Clock signal from the power pallet controller for the serial eeprom.

PPC_SPI_IN

Scan out signal from the serial eeprom.

PPC_SPI_OUT

Scan in signal to the serial eeprom.

PPC_SPI_SELECT

Chip select signal from the power pallet controller for the serial eeprom.

PRD_DATA0<35..0>

Even memory PTE cache read data and parity. Bits <35..32> are byte parity for bits <31..0>.

PRD_DATA1<35..0>

Odd memory PTE cache read data and parity. Bits <35..32> are byte parity for bits <31..0>.

PRD_MOD0<1..0>

Even memory PTE cache modified bit read data and parity. Due to the purge feature of the ram even parity is used.

PRD_MOD1<1..0>

Odd memory PTE cache modified bit read data and parity. Due to the purge feature of the ram even parity is used.

PRD_REF0<1..0>

Even memory PTE cache referenced bit read data and parity. Due to the purge feature of the ram even parity is used.

PRD_REF1<1..0>

Odd memory PTE cache referenced bit read data and parity. Due to the purge feature of the ram even parity is used.

PRD_TAG0<33..17>

Even memory PTE cache tag read data and parity. Bits <33..32> are byte parity for bits <31..17>.

PRD_TAG1<33..17>

Odd memory PTE cache tag read data and parity. Bits <33..32> are byte parity for bits <31..17>.

PRD_TVAL0<1..0>

Even memory PTE cache thread validity read data and parity. Due to the purge feature of the ram even parity is used.

PRD_TVAL1<1..0>

Odd memory PTE cache thread validity read data and parity. Due to the purge feature of the ram even parity is used.

PRD_VAL0<1..0>

Even memory PTE cache non-thread validity read data and parity. Due to the purge feature of the ram even parity is used.

PRD_VAL1<1..0>

Odd memory PTE cache non-thread validity read data and parity. Due to the purge feature of the ram even parity is used.

PREX_DATA<31..0>

The NPSW gate array drives this bus. External muxes are used to select what drives the XMUX_EXTDATA bus, the PREX_DATA bus is one of the available selections.

PSW_AC

PSW_IEEE

PSW_SC
PSW_SEQ
PSW_SQS
PSW_TIT
PSW_TR
PSW_TRACE_TRAP_PEND
PSW_TTC

These signals are bits of the PSW register within the NPSW gate array.

PTED0_CKE*

Even memory PTE cache data ram clock enable (active low).

PTED0_WE*

Even memory PTE cache data ram write enable (active low).

PTED0_WR_PAR_ERR<3..0>

Even memory PTE cache data ram write parity errors.

PTED1_CKE*

Odd memory PTE cache data ram clock enable (active low).

PTED1_WE*

Odd memory PTE cache data ram write enable (active low).

PTED1_WR_PAR_ERR<3..0>

Odd memory PTE cache data ram write parity errors.

PTET0_CKE*

Even memory PTE cache tag ram clock enable (active low).

PTET0_WE*

Even memory PTE cache tag ram write enable (active low).

PTET0_WR_PAR_ERR<1..0>

Even memory PTE cache tag ram write parity errors.

PTEtT1_CKE*

Odd memory PTE cache tag ram clock enable (active low).

PTET1_WE*

Odd memory PTE cache tag ram write enable (active low).

PTET1_WR_PAR_ERR<1..0>

Odd memory PTE cache tag ram write parity errors.

PTE_CLB

This signal is the logical or of the even and odd PTE cache read data bit <10>. The PTE_CLB (page table entry - cache load bypass) signal is used to inhibit a read from accessing the data cache. The signal also inhibits the block prefetch mechanism for that page of memory.

PTE_MOD_PE*

Signal used to enable purging the PTE cache modified bit rams (active low).

PTE_MOD_WE*

Write enable signal for both even and odd PTE cache modified bit rams (active low).

PTE_PURGE_ENTRY

Signal from the NDC gate array which enables writing a zero to the PTE validity rams (even and odd, thread and non-thread).

PTE_REF_PE*

Signal used to enable purging the PTE cache reference bit rams (active low).

PTE_REF_WE*

Write enable signal for both even and odd PTE cache reference bit rams (active low).

PTE_TVAL_PE*

Signal used to enable purging the PTE cache thread validity rams (active low).

PTE_VAL_CS*

Even and odd PTE cache validity ram chip select (active low).

PTE_VAL_PE*

Signal used to enable purging the PTE cache non-thread validity rams (active low).

PURGE*

Signal used to enable purging the look ahead cache validity rams, instruction cache validity rams, and branch history rams.

PURGE_OP2<3..0>

Signals which are decoded by PALs to generate the data cache and PTE cache purge enable signals. The signals are forced to zero when a purge operation is not being requested. Refer to the MFP_OP field in the Micro Code section for additional details.

PURGE_SRVAL*

Signals which enable purging the scratch ram validity. This invalidates the first level PTE entries encached in the scratch ram.

PUSH_QUEUE

From the NPAR to the NIADs indicating that a valid instruction has been parsed, is complete, and the NIADs should push the address and other information onto the output queue, as the NPAR will be doing with its data.

PWR_DATA<35..34>

PTE cache data ram write parity for the least significant two bytes.

PWR_REFMOD

PTE cache reference and modified bit rams write data.

PWR_VAL

PTE cache thread and non-thread validity rams write data.

QUEUE_FULL

Signal from the NPAR to the NIADs indicating that either the input staging for raw instruction data or the output queue is full, and no data from the instruction cache can be accepted.

QUEUE_INVALID

From the NIADs to the NPAR indicating that the data from the instruction cache on this cycle is not valid.

R.CACHE_ADDR_OUT<31..14>

Registered version of INST_CACHE_ADDR<31..14>. The signals are compared to the value read from the instruction cache tag rams to determine if the data read from the instruction cache is from the correct memory page.

R.CNTX_NORMAL

Signal which enables purges and writes to the instruction processor cache rams. The enable is needed to inhibit writes and purges in CAST scan mode and during context save and restore modes.

R.FR_TRAP_TYPE<3..0>

These signals are from a free running register which is loaded with the trap type information from the cross bar.

R.FR_TRAP_VECT<11..0>

Signals from a free running register which loads the trap vector information from the cross bar.

R.MXI_RDY

Registered version of the MXI_RDY signal. It is used to generate chip select signals to write the MXI data to the instruction processor cache rams.

R.PURGE

Registered version of the CCR_ICACHE_PURGE signal. It is used to generate the purge enable signals for the instruction processor validity and branch prediction rams.

R.TRAP_RDY*

Registered and inverted version of the trap ready signal from the cross bar.

RC_2T_FIRST

Signal from NDC subsystem to the NRC gate arrays specifing that this is the first of two memory transfers needed for a processor request. The two type of requests which cause this to happen are unaligned long word requests and unaligned 2x vector requests.

RC_DST<2..0>

Signals from the NDC subsystem to the NRC gate arrays specifing the destination of the memory request.

0	Destined for instruction processor
1	Destined for vector processor
2	Destined for data cache
3	Destined for register file
4	Destined for data cache and register file
5,6,7	Unused

RC_EVEN

Signal from the NDC subsystem to the NRC gate arrays indicating that an even cross bar memory request was required.

RC_ODD

Signal from the NDC sybsystem to the NRC gate arrays indicating that an odd cross bar memory request was required.

RC_PTE

Signal from the NDC subsystem to the NRC gate arrays indicating that the memory request is for a PTE.

RC_RDY

Handshake signal from the NDC gate array to the master NRC gate array (slice zero).

RC_REG_SEL<4..0>

Signal from the NDC subsystem to the NRC gate arrays specifing the location in the register file on the NRFA gate arrays that the return memory data should be written.

RC_REQ_NEXT

Handshake signal from the master NRC gate array (slice zero) to the NDC gate array impling that the NRC gate arrays have sufficient space in the memory return control queue for a transfer the following cycle.

RC_SIZE<1..0>

Signals from the NDC subsystem to the NRC gate arrays specifing the size of the memory request data. The normal encoding is used (0-Byte, 1-Halfword, 2-Word, 3-Longword).

RC_TAG<4..0>

Five bit value used to compare against the data cache update tag once memory data has returned from the cross bar destined for the data cache.

RD_CYC1

Signal from the first stage data cache pipe indicating that the request is for a read.

RD_MUX_MXSQ

Signal used by the CBUS arbitration logic indicating that the memory to scalar queue is not empty.

RD_MUX_VXQ

Signal used by the CBUS arbitration logic which is asserted when the vector to scalar queue is not empty.

REQ_PEND

REQ_PEND_HAZ

Signals indicating that the NDC subsystem has an active request either in its pipe stages or in the cross bar. The REQ_PEND_HAZ signal is used by the NUS gate array as a hazard condition. The signal REQ_PEND is a registered version of REQ_PEND_HAZ and is used by the NIP.

RMINT

Signal used to initiate an NDC subsystem referenced or modified interrupt sequence.

RMINT0

RMINT1

These signals are generated by the NPA0 and NPA1 gate arrays. They are asserted when a page table entry was accessed and the referenced or modified bit

was invalid. PTE misses and access violations have higher priority than ref/mod interrupts.

RMINT2

Registered version of the RMINT signal. The signal is used for reference/modified interrupt sequencing.

RMUINT

The signal is the logical or of the UINT0, UINT1, RMINT0, and RMINT1 signals. RMUINT is used by the NDP gate arrays to control the writing of the micro interrupt vector register.

RMUINT_SIDE

This signal indicates which side (even or odd) of the data cache pipe has the highest priority interrupt. The signal is used to select the even or odd micro interrupt vector for the NAS subsystem micro controller.

RMUINT_SIDE2

A registered version of RMUINT_SIDE used by the NAG gate arrays to select either the even or odd side for the faulting address.

RTN_CTLQ_EMPTY

The signal is asserted by an NRC gate array when the return control queue is empty. The signal is used by the NIP subsystem to determine when to restart the look ahead mechanism and by the NPSW gate array as part of a wait hazard condition (memory idle).

SA0<31..3>

Even side data and PTE cache logical address. The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

SA0_CPG<1..0>

Even side data cache address bits <13..12>. These two address bits are forced to one when the CCR_1PG_CACHE signal is asserted to decrease the size of the data cache to one page. The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

SA0_EQL.0

SA0_EQL0

SA0_EQL1

The upper and lower gate arrays generate signals SA0_EQL0 and SA0_EQL1 indicating that the even side addresses in the first and second stages of the data cache pipe are equal. The two signals are logically anded to generate SA0_EQL.0 and used to control merging reads and writes to the same data cache cell.

SA0_MISS1<31..3>

Registered version of SA0<31..3>. This is the first stage even side address for the data cache pipe. The signals are used by the NPA0 gate array for comparing data and PTE cache tags and for generating the even side physical address.

SA0_PAR<2..0>

Byte parity for writing the even side data cache tags and PTE cache tags.

SA0_VM1

This signal is the VM bit associated with the first stage of the even side data cache

pipe. The VM bit is used to enable or disable vector memory requests which are operating under mask.

SA0_XOR

This is the exclusive or of SA0 bits <31> and <22>. The signals is used as the most significant address bit for the even side PTE cache rams. The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

SA1<31..0>

Odd side data and PTE cache address. The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

SA1_ADDR2<2..0>

Least significant three bits of the second stage data cache pipe logical address. The signals are used by the NRC gate arrays to control rotation for read memory data.

SA1_ADDR2_PAR<3..0>

Byte parity for the second stage data cache pipe logical address.

SA1_CPG<1..0>

Odd side data cache address bits<13..12>. These two address bits are forced to one when the CCR_1PG_CACHE signal is asserted to decrease the size of the data cache to one page. The signals are double cycled with a write address on the first half cycle and a read address on the second half cycle.

SA1_EQL.0

SA1_EQL0

SA1_EQL1

The upper and lower gate arrays generate signals SA1_EQL0 and SA1_EQL1 indicating that the odd side addresses in the first and second stages of the data cache pipe are equal. The two signals are logically anded to generate SA1_EQL.0 and used to control merging reads and writes to the same data cache cell.

SA1_MISS1<31..0>

Registered version of SA1<31..0>. This is the first stage odd side address for the data cache pipe. The signals are used by the NPA1 gate array for comparing data and PTE cache tags and for generating the odd side physical address.

SA1_MISS1_2

Signal generated by the lower NAG gate array for the upper NAG gate array to control selection of the SA0 and SA1 logical address for unaligned long word requests.

SA1_PAR<2..0>

Byte parity for writing the odd side data and PTE cache tags.

SA1_VM1

The signal is the VM bit associated with the first stage of the odd side data cache pipe. The VM bit is used to enable or disable vector memory requests which are operating under mask.

SA1_XOR

This is the exclusive or of SA1 bits <31> and <22>. The signal is used as the most significant bit of the address for the odd side PTE cache rams.

SCPEND

Signal from the NPSW gate array indicating that a write to the SC bit of the PSW register is pending. Branches which rely on the SC bit of the PSW will be held off until this signal is deasserted.

SENSE_VEE

SENSE_VEE10K

SENSE_VEE10K_RET

SENSE_VEE_RET

SENSE_VTT

SENSE_VTTGA

SENSE_VTTGA_RET

SENSE_VTT_RET

These signals are connected to the power planes near the center of the logic area on the board. They are used to sense the voltages on the board for the power pallet regulators.

SP_BP.LBD_INTL_OUT

The signal is used as a logic board interlock signal. The power pallet uses the signal to make sure the board is plugged into the backplane before power is applied.

SP_VP.CNTX_CTL<1..0>

Scalar processor to vector processor context control signals. The signals are driven from bits <10..9> of the CCR register within the NPSW gate array.

SP_VP.DATA<63..0>

Data bus from the scalar processor to vector processor used for scalar processor and memory transfers.

SP_VP.DISP_EP<10..0>

Vector processor instruction dispatch entry point.

SP_VP.DISP_IREG<2..0>

Vector processor instruction dispatch I register field.

SP_VP.DISP_JREG<2..0>

Vector processor instruction dispatch J register field.

SP_VP.DISP_KREG<2..0>

Vector processor instruction dispatch K register field.

SP_VP.DISP_PSW_HAZ

Vector processor instruction dispatch signal indicating that the instruction updates the PSW register with exception conditions. The signal VP_SP.PSW_HAZ must be asserted until the instruction can no longer update the PSW register.

SP_VP.DISP_RDY

Handshake signal from the scalar processor to the vector processor indicating that a dispatch instruction is ready.

SP_VP.IEEE

Signal from the IEEE bit of the PSW register within the NPSW gate array setting the mode for floating point operations to either IEEE mode or native mode.

SP_VP.MXV_RDY

Handshake signal used for the transfer of memory data to the vector processor.

SP_VP.PAR<7..0>

Byte parity signals for SP_VP.DATA.

SP_VP.SXV_RDY

Handshake signal used for the transfer of scalar processor data to the vector processor.

SP_VP.VX_REQ_NEXT

Handshake signal used for the transfer of vector data to either memory or the scalar processor.

SP_XC.DEADLOCK

Signal from the NUS gate array indicating that a two instruction loop is being executed with one of the instruction being a branch and the other being a deadlockable instruction (TAS, TAC, ...).

SP_XC.HARD_ERROR

Signal indicating that a parity error was detected on the board. The signal is driven to the cross bar to halt the entire system.

SP_XC.HW_HUNG

Signal from the hardware hang counter indicating that an instruction has not been transferred from the NIP subsystem to the NAS subsystem for 65536 cycles.

SP_XC.STOP_CNTR

Signal indicating that either a parity error was detected on the board or the micro code set the CCR_HALT bit of the CCR register. The signal is used to stop the system clock cycle counter.

SP_XC.TRAP_COMP

Signal from the NUS gate array indicating that the scalar processor has completed processing a trap.

SP_XRE.RTN_PAR_ERR

Signal to the cross bar even return board indicating that a parity error was detected from the XRE_SP.RD_DATA bus.

SP_XRO.RTN_PAR_ERR

Signal to the cross bar odd return board indicating that a parity error was detected from the XRO_SP.RD_DATA bus.

SP_XSE.ADDR<28..3>

Even side cross bar physical memory/communication address.

SP_XSE.BD_SEL<3..0>

Even side cross bar board select. Values zero through seven select memory boards, value eight and nine selects the communication board.

SP_XSE.CTL_PAR<4..0>

Parity for the even side cross bar control signals.

Parity bit	Signals parity generated from
4	SP_XSE.WR_ZONE<3..0>
3	SP_XSE.ADDR<7..3>, SP_XSE.CYCLE<1..0>
2	SP_XSE.ADDR<14..8>
1	SP_XSE.ADDR<21..15>
0	SP_XSE.ADDR<28..22>

SP_XSE.CYCLE<1..0>

Signals indicating the type of request being made.

0	Communication board status return request
1	Memory/Communication board read request
2	Memory/Communication board write request
3	Memory/Communication board read modify write request

SP_XSE.RDY

Signal from the scalar processor indicating that a valid request is ready to be transferred.

SP_XSE.WR_DATA<31..0>

Scalar processor to cross bar even side write data bus.

SP_XSE.WR_PAR<3..0>

Byte parity for SP_XSE.WR_DATA.

SP_XSE.WR_ZONE<3..0>

Scalar processor to cross bar even side write zone information.

SP_XSO.ADDR<28..3>

Odd side cross bar physical memory/communication address.

SP_XSO.BD_SEL<3..0>

Odd side cross bar board select. Values zero through seven select memory boards, value eight and nine selects the communication board.

SP_XSO.CTL_PAR<4..0>

Parity for the odd side cross bar control signals.

Parity bit	Signals parity generated from
4	SP_XSO.WR_ZONE<3..0>
3	SP_XSO.ADDR<7..3>, SP_XSO.CYCLE<1..0>
2	SP_XSO.ADDR<14..8>
1	SP_XSO.ADDR<21..15>
0	SP_XSO.ADDR<28..22>

SP_XSO.CYCLE<1..0>

Signals indicating the type of request being made.

0	Communication board status return request
1	Memory/Communication board read request
2	Memory/Communication board write request
3	Memory/Communication board read modify write request

SP_XSO.RDY

Signal from the scalar processor indicating that a valid request is ready to be transferred.

SP_XSO.WR_DATA<31..0>

Scalar processor to cross bar odd side write data bus.

SP_XSO.WR_PAR<3..0>

Byte parity for SP_XSO.WR_DATA.

SP_XSO.WR_ZONE<3..0>

Scalar processor to cross bar odd side write zone information.

SRADDR.0<10..0>

Scratch ram address generated by external pals.

SRD_WR_PAR_ERR<3..0>

Scratch ram write parity errors.

SRVAL_DATA

Scratch ram validity read data.

SRVAL_PAR

Scratch ram validity read parity. Even parity is used due to the purge feature of the ram.

SR_CNT<10..0>

Scratch ram counter value which can be selected as an address to access the scratch rams.

SR_DATA<31..0>

Scratch ram read data bus.

SR_PAR<3..0>

Byte parity for SR_DATA.

SR_WE*

Scratch ram write enable (active low).

START_COMP

Communication from NIAD1 to NIAD0 during branch restarts, used with its own results of the compare of the restart address with the starting program counter to determine if the restart will force a look ahead restart.

START_COMP_L

Communication from NIAD0 to NIAD1. See START_COMP.

STRAM_CS.0

Chip select signal for all instruction processor cache rams (active low).

STRAM_ENABLE*

Signal used to enable self timed ram write parity checking for all rams on the board (active low).

STRAM_SE.0

Self timed ram scan enable signal. The signal is fanned out and used for all rams on the board.

STRAM_WR_PAR_ERR*

Logical or of all self timed ram write parity signals (active low).

ST_PEND_HAZ

Hazard condition used by the NUS gate array indicating that a store memory request is active in the data cache pipe stages or the cross bar.

TAG_VALID.0

Result of the instruction cache tag compare and valid bit.

TEST_HAZ

Test hazard signal indicating that the selected micro code test condition is not ready.

THAZ_U1LAS_U2INV

THAZ_U1LAS_U2VAL

THAZ_U1_U2INV

THAZ_U1_U2VAL

These four signals are test hazard conditions assuming the four possible logic values for the two signals UIR2_VAL and UIR1_LAS_FULL. The appropriate signal is selected by a mux to drive the signal TEST_HAZ.

TRACE_TRAP_PEND

Signal from the NIP subsystem to the NPSW gate array indicating that a trace trap is pending. The signal is loaded into the PSW register when a ring crossing system call is made.

TRAP_PEND

Signal which indicates a trap is pending. The signal is used to inhibit look ahead instruction prefetch requests.

TRAP_RDY

Registered twice version of XC_SP.TRAP_RDY.

TRAP_TYPE<3..0>

Registered version of R.FR_TRAP_TYPE. The register is only loaded when a new trap is ready.

TRAP_VECT<11..0>

Registered version of R.FR_TRAP_VECT<11..0>. The register is only loaded when a new trap is ready.

TR_OR_SEQ

This signals is generated by the NPSW gate array by oring the TR and SEQ bits. It is used as a test condition by the NUS gate array.

UA0<13..3>

Even side data cache update tag address. The most significant two bits are forced to one when the cache is in one page mode.

UA1<13..3>

Odd side data cache update tag address. The most significant two bits are forced to one when the cache is in one page mode.

UINT

UINT0

UINT1

The NPA0 and NPA1 gate arrays generate UINT0 and UINT1 when a PTE miss or data cache access violation is detected. These two signals are logically ored together to generate UINT. The signal is used to interrupt the NUS gate array micro controller.

UINT_HOLD.0

UINT_HOLD0

UINT_HOLD1

The NPA0 and NPA1 gate arrays generate UINT_HOLD0 and UINT_HOLD1 when a PTE miss or data cache access violation is detected. These two signals are logically or'ed together to generate UINT_HOLD. The signal is used to hold the data cache pipe stages while the micro code resolves the problem.

UINT_MODE

Signal from the NUS gate array indicating that it is executing interrupt micro code. The signal causes the UIR interface from the NAS subsystem to the NDC subsystem to be held.

UINT_VEC<3..0>

UINT_VEC0<3..0>*

UINT_VEC1<3..0>*

The NPA0 and NPA1 gate arrays generate UINT_VEC0 and UINT_VEC1 signals. The signals represent the type of PTE miss or access violation which exists. The higher priority micro interrupt vector is registered within an NDP gate array and driven off as UINT_VEC. Zero is highest priority, and F being lowest.

0	Inward ring access violation
1	PTE miss
2	Invalid thread level PTE (from cache)
3	Invalid second level PTE (from cache)
4	Read access violation
5	Write access violation
6	Execute access violation
7	Non-resident data
8	Invalid communication address
9	Non-faulting IP access PTE miss
A	Invalid segment descriptor register
B	Invalid first level PTE (from memory)
C	Non-resident second level PTE
D	Invalid second level PTE (from memory)
E	Non-resident thread level PTE
F	Forced fault

UIR1_BDREG_SEL<4..0>

First stage micro instruction register field from NAS subsystem to NDC subsystem. The signals are the register select for register file writes from the data cache, memory, or vector to scalar transfers.

UIR1_BDSIZE<1..0>

First stage micro instruction register field. The signals are the size for memory requests, communication board requests, and vector to scalar transfers.

UIR1_EXTX_SEL<2..0>

UIR1_EXTX_SEL_LAS<2..0>

First stage micro instruction register field used to select the external xmux. When UIR1_LAS_FULL is asserted then UIR1_EXTX_SEL_LAS is used else UIR1_EXTX_SEL is used. Refer to the micro code section for additional details.

UIR1_FU_OP<5..0>

First stage micro instruction register field used for function unit opcode, driven by the NUS gate array. Refer to the micro code section for additional details.

UIR1_INST_SEG<2..0>

First stage micro instruction register field which specifies the most significant three bits of the current instructions PC. It is used by the NDC subsystem to check for inward ring access violations.

UIR1_LAS_FULL.0

First stage micro instruction register pipe control signal. The signal indicates that the UIR1 level of the pipe was held on the previous cycle and that the UIR1 look aside (LAS) registers sould be selected.

UIR1_MEM_REQ

First stage micro instruction register signal specifing that a memory operation is present at the UIR1 stage of the pipe.

UIR1_NDIV_REQ

First stage micro instruction register signal specifing that an operation for the NDIV function unit is present at the UIR1 stage of the pipe.

UIR1_NFAD_REQ

First stage micro instruction register signal specifing that an operation for the NFAD funtion unit is present at the UIR1 stage of the pipe.

UIR1_NMISC_REQ

First stage micro instruction register signal specifing that an operation for the NMISC function unit is present at the UIR1 stage of the pipe.

UIR1_NMUL_REQ

First stage micro instruction register signal specifing that an operation for the NMUL function unit is present at the UIR1 stage of the pipe.

UIR1_OVR_FULL.0

First stage micro instruction register pipe control signal. The signal indicates that the UIR1 level of the pipe was held on the previous cycle and that the UIR1 overrun registers should be used to load into the UIR1 registers when the hold is released.

UIR1_UCONST<9..0>

First stage micro instruction register field used as a micro constant. Refer to the micro code section for details.

UIR1_UPC<11..0>

First stage micro instruction register signals which contain the micro program counter which was used to access the control store rams.

UIR1_VAL.0

First stage micro instruction register pipe control signal. The signal indicates that a valid instruction is present at the UIR1 level of the pipe.

UIR2V_LD_CIR

Second stage micro instruction register signal used to enable loading of the NDC subsystem copy of the CIR (communication index register). The signal has already been qualified with UIR2_VAL.

UIR2V_PTE_OP_REQ

Second stage micro instruction register signal used to specify that a PTE operation is present at the UIR2 level of the pipe. The signal has already been qualified with UIR2_VAL.

UIR2V_REQ_RETRY.0

Second stage micro instruction register signal used to retry a data cache pipe request. The original memory request was aborted due to a PTE miss, or forced fault. The signal has already been qualified with UIR2_VAL.

UIR2V_Y_WR<1..0>

Second stage micro instruction register field used to specify the NDC subsystem destination register for the YBUS data. The signals have already been qualified with UIR2_VAL.

0	No write
1	CCR (CPU control register)
2	TID (Thread index register)
3	CIR (Communication index register)

UIR2_JMP_RESTART<2..0>

Second stage micro instruction register field used to inform the NIP subsystem of the type of jump restart to perform. Refer to the micro code section for encoding details.

UIR2_LD_CIR

Second stage micro instruction register signal used to enable loading of the NDC subsystem copy of the CIR (communication index register).

UIR2_LD_VS

Second stage micro instruction register signal used to enable loading of the NDC subsystem copy of the VS register (vector stride).

UIR2_NFAD_CMP

Second stage micro instruction register signal used to specify that the NFAD function unit operation at the UIR2 level will be generating a status bit indicating the result of the compare. The signals is also used to inhibit writing the register file from this operation.

UIR2_PTE_DATA_WE

Second stage micro instruction register signal used to enable loading the PTE cache rams with a page table entry.

UIR2_PTE_OP<2..0>

Second stage micro instruction register field specifing the type of PTE operation to perform.

0	Write second level PTE to PTE cache
1	Write second level thread PTE to PTE cache
2	No operation
3	Write referenced or modified bit PTE rams

4	No operation
5	Request first level PTE from memory
6	Request second level PTE from memory
7	Request thread level PTE from memory

UIR2_PTE_OP_REQ

Second stage micro instruction register signal indicating that a PTE operation is at the UIR2 level.

UIR2_REQ_RETRY.0

Second stage micro instruction register signal used to retry a data cache pipe request. The original request was aborted due to a PTE miss or forced fault.

UIR2_SRADDR<10..0>

Second stage micro instruction register field which can be selected as the address to write the scratch rams.

UIR2_SXV_REQ

Second stage micro instruction register indicating that a scalar to vector transfer operation is at the UIR2 level.

UIR2_VAL.0

Second stage micro instruction register control signal. The signal specifies that the UIR2 level is valid.

UIR2_WR_SR

Second stage micro instruction register used to enable a write operation for the scratch rams.

UIR2_Y_WR<1..0>

Second stage micro instruction register field used to specify the destination register for the YBUS data. Refer to the signal UIR2V_Y_WR for a table of the encoded values.

UIR3_VAL

Registered version of UIR2_VAL. Used to determine when the NAS is idle.

UIR_HAZ

Signal from the NDC gate array indicating that a memory request can not be accepted due to the data cache pipe being held (MI_HOLD or UINT_HOLD) or because a higher priority request is being processed (vector or unaligned long word).

UNLW2

First stage data cache pipe signal which is asserted when the second half of an unaligned long word request is be processed.

UNLW_CRY

Carry from the lower to upper NAG gate arrays for the unaligned long word address adder.

UPC_BR_POL

Branch polarity signal at the micro program counter pipe stage. The signal is part of an instruction being dispatched from the NIP subsystem to the NAS subsystem.

UPC_BR_SEL<1..0>

Branch select signals at the micro program counter pipe stage. The signals are

part of an instruction being dispatched from the NIP subsystem to the NAS subsystem.

0	Unconditional branch
1	Branch on ION (not supported)
2	Branch on AC of PSW register
3	Branch on SC of PSW register

UPC_CPC<31..1>

Current program counter at the micro program counter pipe stage. The signals are the address of the instruction being dispatched from the NIP subsystem to the NAS subsystem.

UPC_DL

Deadlock signal at the micro program counter pipe stage. The signal is used to identify the dispatched instruction as a deadlockable instruction.

UPC_LAS_FULL

Micro program counter stage pipe control signal. The signal indicates that the UPC level of the pipe was held on the previous cycle and that the UPC look aside (LAS) registers should be selected.

UPC_OVR_FULL

Micro program counter stage pipe control signal. The signal indicates that the UPC level of the pipe was held on the previous cycle and that the UPC overrun registers should be used to load into the UPC registers when the hold is released.

UPC_VAL

Micro program counter stage pipe control signal. The signal is used to specify that the instruction at the UPC level is valid.

UPD0_CYC1*

Signal from the first stage of the data cache pipe indicating an even side data cache update write is to be performed (active low).

UPD0_TAG_EQL

Signal generated by discrete logic which is the result of comparing the even update tag ram read data with the update tag received from the NRC gate arrays.

UPD0_WR_EQL0

UPD0_WR_EQL1

Signals from the lower and upper NAG gate arrays which are the result of comparing the second stage even side data cache pipe logical address with the first stage update address. The signals are used to inhibit an update to the data cache when a processor write to the same cell just occurred.

UPD1_CYC1*

Signal from the first stage of the data cache pipe indicating an odd side data cache update write is to be performed (active low).

UPD1_TAG_EQL

Signal generated by discrete logic which is the result of comparing the odd update tag ram data with the update tag received from the NRC gate arrays.

UPD1_WR_EQL0

UPD1_WR_EQL1

Signals from the lower and upper NAG gate arrays which are the result of comparing the second stage odd side data cache pipe logical address with the first stage update address. The signals are used to inhibit an update to the data cache when a processor write to the same cell just occurred.

UPDQ_RPTR<1..0>

These signals are the read pointer for the data cache update queue.

0	Read queue location zero
1	Read queue location one
2	Read queue location two
3	Unused decode

UPDQ_TAG<4..0>

These signals are the update tag from the data cache update queue. The tag is compared against the update tag ram data to determine if an update write should occur.

UPDQ_WPTR<1..0>

These signals are the write pointer for the data cache update queue.

0	Write queue location zero
1	Write queue location one
2	Write queue location two
3	Write inhibit (queue full)

UPD_ADDR<31..0>

Logical address for the data cache update data being transferred from the NRC subsystem to the NDC subsystem.

UPD_ADDR_PAR<3..0>

Byte parity for UPD_ADDR.

UPD_CRY

Carry from the lower to upper NAG gate arrays for the update address adder.

UPD_CYC1.0

Signal generated from UPD0_CYC1* and UPD1_CYC1* indicating a cache update write cycle is to be performed.

UPD_DATA<63..0>

Data path for update and memory to scalar processor transfers from the NRC subsystem to the NDC subsystem.

UPD_EVEN

Signal from the NRC subsystem to the NDC subsystem indicating that UPD_DATA<63..32> contains even side data cache update data.

UPD_ODD

Signal from the NRC subsystem to the NDC subsystem indicating that UPD_DATA<31..0> contains odd side data cache update data.

UPD_PAR<7..0>

Byte parity for UPD_DATA.

UPD_RDY

UPD_REQ_NEXT

Handshake signals from the NRC subsystem to the NDC subsystem for data cache updates as well as memory to scalar processor transfers.

UPD_TAG<4..0>

Update tag for the update data being transferred from the NRC subsystem to the NDC subsystem.

USEQ_UINT

?????

USE_EXTDATA_LAS

Signal indicating that the external XMUX look aside register within the NRFA should be selected as the source for the XMUX data.

USW_IC

USW_JC

USW_KC

These signals are bits of the USW (user status word) register within the NPSW gate array and are used as test conditions for the NUS gate array.

US_ABUS_FMT<1..0>

US_ABUS_SIZE<1..0>

US_ABUS_WR_EN

US_AGZ_HAZEN

US_AG_SEL<3..0>

US_ALUFAST

US_ALUOP<4..0>

US_ALUSIZE<1..0>

US_CRY_DST<2..0>

US_CRY_OP<1..0>

US_CRY_SRC<2..0>

US_CSDISP

US_CX_OP_REQ

US_EXTX_SEL<2..0>

US_FAKE_RING0

US_FIRSTU

US_FU_OP_REQ

US_IPINH<1..0>

US_JMP_RESTART<1..0>

US_LC_CTL<1..0>

US_MEM_OP_REQ

US_MFP_OP<10..0>

US_NAG0_PAR
US_NAG1_PAR
US_NDC_PAR
US_NPSW_PAR
US_NRFA_PAR
US_NUS_PAR
US_PREX_SEL<2..0>
US_PSW_OP<2..0>
US_PTE_OP_REQ
US_REQ_RETRY
US_SR_ADDR_SEL<1..0>
US_SR_CNT_SEL<1..0>
US_SXV_REQ
US_TPOL
US_TSEL<4..0>
US_UCONST<9..0>
US_USEQ_RAND<2..0>
US_VP_DISP
US_VP_PSW_HAZ
US_VXS_REQ
US_WR_SR
US_XMUX_SEL<1..0>
US_XREG_FMT<1..0>
US_XREG_HAZ<2..0>
US_XREG_SEL<5..0>
US_YREG_FMT
US_YREG_HAZ<2..0>
US_YREG_SEL<5..0>
US_YZ_WR_DST<2..0>
US_ZREG_HAZ<2..0>
US_ZREG_SEL<5..0>

Control store field signals which are the logical or of the upper and lower control store ram banks. The logical or function is implemented in either discrete or gates or as wire or'ed signals. The fields are distributed to various gate arrays on the board. Refer to the Micro Code section for encoding details.

U_IXA_RDY

This signal is from the most significant NIAD gate array and is combined with L_IXA_RDY from the least significant NIAD gate array to generate the signal

IXA_RDY.

U_US_ABUS_FMT<1..0>

U_US_AGZ_HAZEN

U_US_BDOP<1..0>

U_US_BDREG_FMT<1..0>

U_US_BRADDR<11..0>

U_US_BRTYPE<3..0>

U_US_CSDISP

U_US_XREG_SEL<5..0>

U_US_YREG_SEL<5..0>

U_US_ZREG_SEL<5..0>

Control store signals from the lower bank of control store rams. These signals are or'ed with similarly named fields from the upper bank and are then distributed to various gate arrays on the board. Refer to the Micro Code section for encoding details.

VA0<2..0>

The least significant three bits of the logical address generated by the vector address generator within the least significant NAG gate array. The NAG gate arrays use bit <2> for the SA address selection muxes and the NDC gate array uses bits <1..0> for unaligned long word checking.

VA0_CRY

Carry from the lower to upper NAG gate arrays for the vector address generator adder.

VA0_ODD1

Signal from lower NAG gate array indicating that a 2x vector operation is being performed which requires swapping the upper and lower 32 bits of the data bus in the NDP gate arrays.

VA1_CRY<1..0>

Carrys from the least to most significant NAG gate arrays for the vector address generator plus eight adder.

VAG_2T_FIRST1

Signal from the NDC gate array specifing that the request at the first stage of the data cache pipe is the first of two requests required.

VAG_2X0

Signal from the NDC gate array specifing that a vector address generator request is being processed at 2x rate. The signal is used by the NAG gate arrays to control selection of the SA address muxes.

VAG_2X1

Registered version of VAG_2x0. The signal is for the first stage of the data cache pipe.

VAG_ACCUM_SEL

Signal generated by the vector address generation logic on the NDC gate array controlling the selection of the accumulation mux on the NAG gate arrays.

VAG_BASE_EN

Signal generated by the vector address generation (VAG) logic on the NDC gate array which enables loading the VAG base address register.

VAG_BUSY

Signal from the NDC gate array indicating that the VAG logic is processing a vector request.

VAG_OFFSET_SEL<1..0>

Signals generated by the vector address generation logic on the NDC gate array controlling the selection of the offset mux on the NAG gate arrays.

VAG_P8_SEL

Signal generated by the vector address generation logic on the NDC gate array controlling the selection of the plus eight mux on the NAG gate arrays.

VAG_VM<1..0>

The VM bits are driven by the NDC gate array. The NAG gate arrays process the VM bit information to keep the VM bits flowing with the appropriate vector address throuch the SA muxes.

VAG_VS_15

The lower NAG gate array drives bit <15> of the VS (vector stride) register to the upper NAG gate array. The upper NAG gate array uses the signal when the VAG must add two times VS to generate an address.

VALID_DATA

From the NPAR to the NIADs indictating that the last instruction parsed was complete, so the data at PARSE_BR_DISP, PARSE_BR_POL, PARSE_SIZE and PARSE_XTEND is valid.

VAT_CYC1

First stage data cache pipe signal specifing that the request requires virtual to physical address translation.

VP_DISP_HAZ

Signal from the NDC gate array which is asserted when a vector instruction is ready to be dispatched but the vector processor can not accept it.

VP_DISP_U2INV

VP_DISP_U2VAL

Signals from the NDC gate array which are used to generate SP_VP.DISP_RDY. The signal UIR2_VAL is used to select which of the two signals to use.

VP_SP.DATA<63..0>

Vector processor data for transfers to the memory system and scalar processor.

VP_SP.DISP_REQ_NEXT

Handshake signal from the vector processor for vector instruction dispatch transfers.

VP_SP.IDLE

Signal from the vector processor indicating that it has completed all instructions.

VP_SP.MXV_REQ_NEXT

Handshake signal from the vector processor for memory to vector processor data

transfers.

VP_SP.PAR<7..0>

Byte parity for VP_SP.DATA.

VP_SP.PSW_HAZ

Signal from the vector processor indicating that it has an instruction which may have exception conditions which would modify the PSW regsiter.

VP_SP.SET_FDZ

VP_SP.SET_FSN

VP_SP.SET_OV

VP_SP.SET_RO

VP_SP.SET_SDZ

VP_SP.SET_SIV

VP_SP.SET_UN

Signals from the vector processor which set exception condition bits in the PSW register within the NPSW gate array.

VP_SP.SXV_REQ_NEXT

Handshake signal from the vector processor for scalar processor to vector processor transfers.

VP_SP.VXA_ADDR<31..0>

Vector processor address used by the vector address generation logic of the NDC subsystem. The address bus is transferred under control of handshake signals VP_SP.VXM_RDY and SP_VP.VX_REQ_NEXT.

VP_SP.VXA_ADDR_PAR<3..0>

Byte parity for VP_SP.VXA_ADDR.

VP_SP.VXA_LAST

Signal from the vector processor used to indicate the last transfer for a sequence of vector to memory data transfers. The signal is used by the vector address generator to know when to stop making requests.

VP_SP.VXA_VM<1..0>

VM bits from the vector processor which are used to mask read and write memory operations.

VP_SP.VXM_RDY

Handshake signal from the vector processor for vector to memory transfers.

VP_SP.VXS_RDY

Handshake signal from the vector processor for vector to scalar transfers.

VS_ODD_WORD

VS_ODD_WORD0

VS_ODD_WORD1

The lower and upper NAG gate arrays generate the signals VS_ODD_WORD0 and VS_ODD_WORD1 which are combined together to produce VS_ODD_WORD. The signal is used by the NDC gate array to enable 2x data transfer rate when the

signal is asserted.

VXQ_RPTR<1..0>

These signals are the read pointer for the vector processor transfer queue. Both vector to scalar processor and vector to memory transfers are put into the queue.

0	Read queue location zero
1	Read queue location one
2	Read queue location two
3	Unused decode

VXQ_WPTR<1..0>

These signals are the write pointer for the vector processor transfer queue. Both vector to scalar processor and vector to memory transfers are put into the queue.

0	Write queue location zero
1	Write queue location one
2	Write queue location two
3	Write inhibit (queue full)

WFC_U1LAS_U2INV

WFC_U1LAS_U2VAL

WFC_U1_U2INV

WFC_U1_U2VAL

?????

WFC_UIR1_HAZ

?????

WR_CYC1

Signal from the first stage of the data cache pipe specifing that the request is a write.

WR_CYC2

Registered version of WR_CYC1.

WR_MUX_CTL

Signal generated by the NDC gate array which controls the write muxes on the NDP gate arrays.

WR_SR.0

Signal used to select either a read or write address for scratch ram accesses.

XB_REQ_PEND

The signal is the logical or'ing of the four registered cross bar request pend signals. The signal is asserted whenever the cross bar has a request waiting arbitration to be issued to a memory or communiction board.

XB_ST_PEND

The signal is the logical or'ing of the four registered cross bar store request pend signals. The signal is asserted whenever the cross bar has a store request waiting arbitration to be issued to a memory or communication board.

XC_CU_STATUS

Registered version of XC_SP.CU_STATUS.

XC_CU_STATUS_EN

Registered version of XC_SP.CU_STATUS_EN

XC_DEADLOCK

Signal from the NUS gate array indicating that a two instruction loop is being executed with one of the instructions being a branch and the other being a deadlockable instruction (TAS, TAC, ...).

XC_MT_COMP

Registered version of XC_SP.MT_COMP.

XC_SP.CU_STATUS

Status result from an operation performed on the communication board.

XC_SP.CU_STATUS_EN

Signal used to enable loading of the XC_SP.CU_STATUS signal. The signal is asserted only when the status signal is valid.

XC_SP.MT_COMP

Signal from the communication board specifing that a micro trap operation which was originated by this scalar processor has completed on all required scalar processors.

XC_SP.SCAN_CTL<2..0>

Scan control signals for the board from the cross bar control board.

0	Normal mode
1	Undefined
2	Undefined
3	Undefined
4	Undefined
5	Last left shift mode
6	Load mode (CAST)
7	Left shift mode

XC_SP.SCAN_IN

Scan input to the serial scan ring on the board.

XC_SP.TRAP_RDY

Signal from the communication board specifing that a trap is being sent to this scalar processor. The trap is sent using XC_SP.TRAP_TYPE and XC_SP.TRAP_VECT.

XC_SP.TRAP_TYPE<3..0>

Signals from the communication board specifing the type of trap which is being issued.

XC_SP.TRAP_VECT<11..0>

Signals from the communication board used to specify additional information for a trap. As an example, a purge page table entry trap specifies the ram address which is to be cleared.

XC_SP.USEC_EN

Increment enable for the micro second counter implemented within the NPSW gate

array.

XC_TRAP_COMP

Signal from the NUS gate array which is used to inform the communiction board that a micro trap has been completed by this scalar processor.

XC_TRAP_RDY

Registered version of XC_SP.TRAP_RDY.

XC_USEC_EN

Registered version of XC_SP.USEC_EN.

XMUX_DATA<63..0>

Bus which supplies the X data input for the four function units. The bus is at the first level micro instruction register level.

XMUX_EXTDATA<31..0>

Data bus from the external X muxes which is received by the NRFA gate arrays.

XMUX_EXTPAR_VAL

Signal from the NUS gate array indicating that the XMUX_EXTDATA is from the scratch rams and that SR_PAR should be checked for correct byte parity.

XMUX_PAR<7..0>

Byte parity for XMUX_DATA.

XRE_RDY

Registered version of XRE_SP.RD_RDY. The master NRC gate array (slice zero) receives XRE_SP.RD_RDY and informes the other two NRC gate arrays on the following cycle that data was received.

XRE_SP.RD_DATA<31..0>

Even cross bar side memory return data.

XRE_SP.RD_PAR<3..0>

Byte parity for XRE_SP.RD_DATA.

XRE_SP.RD_RDY

Signal form the even side cross bar informing the scalar processor that even side return data is being transferred.

XRO_RDY

Registered version of XRO_SP.RD_RDY. The master NRC gate array (slice zero) receives XRE_SP.RD_RDY and informes the other two NRC gate arrays on the following cycle that data was received.

XRO_SP.RD_DATA<31..0>

Odd cross bar side memory return data.

XRO_SP.RD_PAR<3..0>

Byte parity for XRO_SP.RD_DATA.

XRO_SP.RD_RDY

Signal from the odd side cross bar informing the scalar processor that odd side return data is being transferred.

XSE_SP.A_REQ_NEXT

Handshake signal from the A - XSE send cross bar board for memory/ communication requests being transferred from the scalar processor to the cross bar.

XSE_SP.A_REQ_PEND

Signal from the A - XSE send cross bar board indicating that an active memory request is waiting to be sent to a memory/communication board.

XSE_SP.A_ST_PEND

Signal from the A - XSE send cross bar board indicating that an active store memory request is waiting to be sent to a memory/communication board.

XSE_SP.B_REQ_NEXT

Handshake signal from the B - XSE send cross bar board for memory/ communication requests being transferred from the scalar processor to the cross bar.

XSE_SP.B_REQ_PEND

Signal from the B - XSE send cross bar board indicating that an active memory request is waiting to be sent to a memory/communication board.

XSE_SP.B_ST_PEND

Signal from the B - XSE send cross bar board indicating that an active store memory request is waiting to be sent to a memory/communication board.

XSO_SP.A_REQ_NEXT

Handshake signal from the A - XSO send cross bar board for memory/ communication requests being transferred from the scalar processor to the cross bar.

XSO_SP.A_REQ_PEND

Signal from the A - XSO send cross bar board indicating that an active memory request is waiting to be sent to a memory/communication board.

XSO_SP.A_ST_PEND

Signal from the A - XSO send cross bar board indicating that an active store memory request is waiting to be sent to a memory/communication board.

XSO_SP.B_REQ_NEXT

Handshake signal from the B - XSO send cross bar board for memory/ communication requests being transferred from the scalar processor to the cross bar.

XSO_SP.B_REQ_PEND

Signal from the B - XSO send cross bar board indicating that an active memory request is waiting to be sent to a memory/communication board.

XSO_SP.B_ST_PEND

Signal from the B - XSO send cross bar board indicating that an active store memory request is waiting to be sent to a memory/communication board.

YBUS_DATA<63..0>

Main external data bus for the scalar processor. The bus is at the second stage micro instruction register level.

YBUS_HOLD_FULL

Signal specifing that the YBUS_HOLD register in the NDP gate arrays contains

valid data which must be held for use on subsequent cycles.

YBUS_LAS_FULL

Signal specifing that the YBUS_LAS register in the NDP gate arrays contains valid data which must be held for use on subsequent cycles.

YBUS_PAR<7..0>

Byte parity for YBUS_DATA.

YMUX_DATA<63..0>

Bus which supplies the Y data input to the four function units. The bus is at the first stage micro instruction register level.

YMUX_PAR<7..0>

Byte parity for YMUX_DATA.

YUVC_PAR0<7..0>

YUVC_PAR1<7..0>

YUVC_PAR2<7..0>

Signals from the NDP gate arrays which are the partial parity for the byte parity checkers.

YUVC_PAR_ERR.0<3..0>

Parity error sigals from external PALs which combine the partial parity signals YUVC_PAR0, YUVC_PAR1, and YUVC_PAR2.

ZMUX_DATA<31..0>

Address generation data from the NRFA gate arrays to the NAG gate arrays.

ZMUX_PAR<3..0>

Byte parity for ZMUX_DATA.

Appendix B Context Block Format

This appendix shows the format of the context frame for the C38xx system. Offsets are given from the "top", i.e. program counter, end of the frame in hexadecimal. If the machine is stopped at the entry to the page fault handler, these offsets can be added to the frame pointer to provide the byte address of a particular item of processor context. The items such as RAM(SAVE_A0) are scratch RAM locations used as temporary storage to save user registers by microcode routines that need more than T0-T7 to operate. Since there may be a fault between when the user registers have been copied into scratch RAM and restored for the next instruction, these RAM locations must be fault context.

The format of the NSP and NRC scanned context blocks are given following the list of context frame offsets. The format of the NVP scanned context block may be found in the *C38xx Vector Processor Specification.*

Offset	Contents
000	PC
004	PSW
008	A7
00C	A6
010	A5
014	A4
018	A3
01C	A2
020	A1
024	A0
028	S7 (longword)
030	S6 (longword)
038	S5 (longword)
040	S4 (longword)
048	S3 (longword)
050	S2 (longword)
058	S1 (longword)
060	S0 (longword)
068	TTR (Thread Timer Register - longword)
070	T7
074	T6
078	T5
07C	T4
080	T3
084	T2
088	T1

08C	T0
090	FRL - user frame length bits (before set to 00 to indicate context frame)
094	CCR
098	XPC
09C	NSP scanned context - 80_{10} words
1DC	NRC scanned context - 145_{10} words
420	RAM(SAVE_S0U)
424	RAM(SAVE_S0), also referred to as RAM(XTRA_REG)
428	RAM(SAVE_S1U)
42C	RAM(SAVE_S1)
430	RAM(SAVE_S2U)
434	RAM(SAVE_S2)
438	RAM(SAVE_S3U)
43C	RAM(SAVE_S3)
440	RAM(SAVE_S4U)
444	RAM(SAVE_S4)
448	RAM(SAVE_S5U)
44C	RAM(SAVE_S5)
450	RAM(SAVE_S6U)
454	RAM(SAVE_S6)
458	RAM(SAVE_S7U)
45C	RAM(SAVE_S7)
460	RAM(SAVE_A7)
464	RAM(SAVE_A6)
468	RAM(SAVE_A5)
46C	RAM(SAVE_A4)
470	RAM(SAVE_A3)
474	RAM(SAVE_A2)
478	RAM(SAVE_A1)
47C	RAM(SAVE_A0)
480	NVP scanned context - 692_{10} words
F50	first location after end of context block

The format of the NSP and NRC scan blocks are shown in Figure B-1 and Figure B-2, respectively. Since the scan data is written into memory in the order it scans out and the machine scans unidirectionally left, the most significant bit of the scan ring will be found at the lower memory address, i.e. closer to the top of the context frame. Memory offsets from the top of the context frame are given on the left edge of the figure. Scan ring bit indices are given on the right. The context bus bit index is given across the top, giving a "column number" to the block to contrast with the "row number" of the scan ring bit index.

Figure B-1 NSP Scanned Context

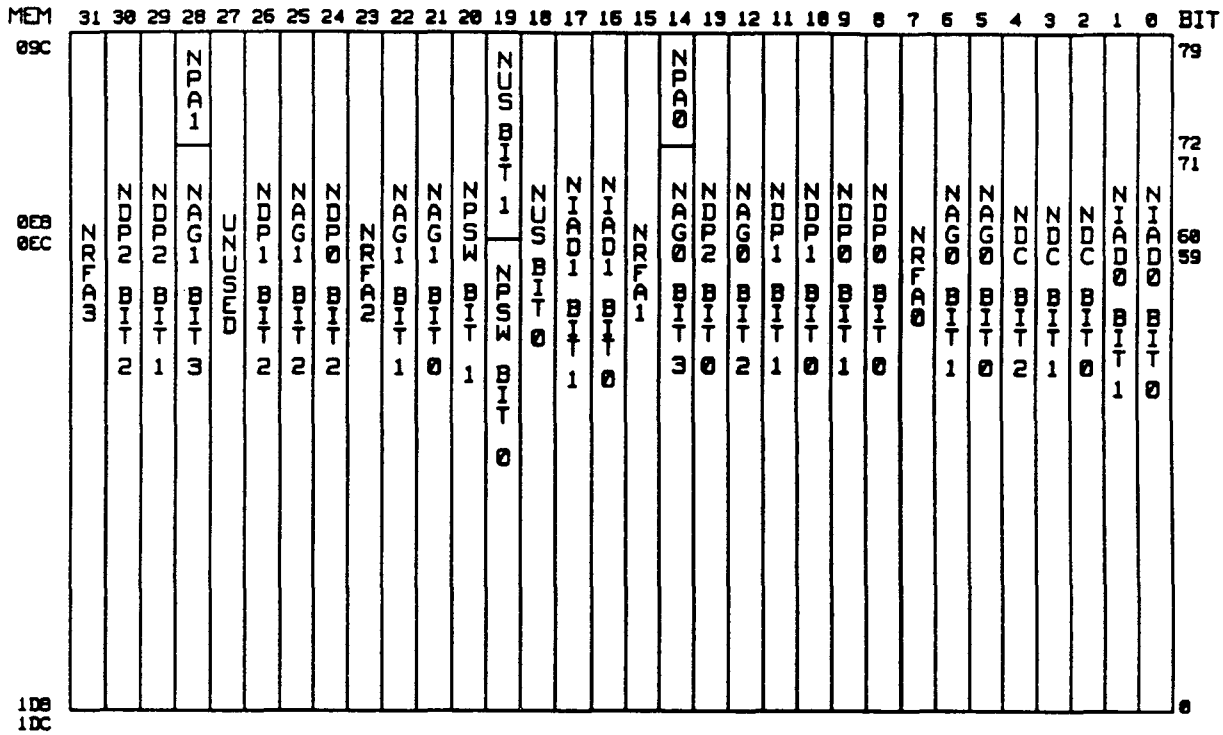
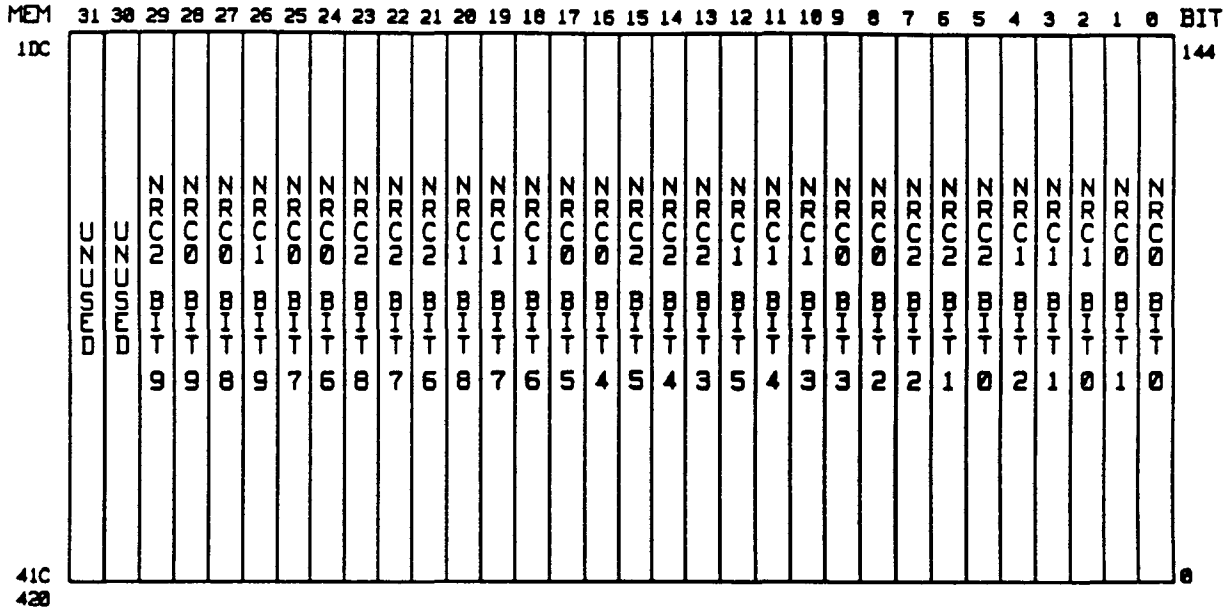


Figure B-2 NRC Scanned Context



Appendix C

Class Foils