C38xx NIA Functional Specification

Document No. 416-001244-000

Revision 1.2
September 18, 1991

INTERNAL USE ONLY

PROPRIETARY DISCLAIMER

Table of Contents

List of Figures

List of Tables

1 Introduction

The C38xx Interface Adaptor (NIA) provides Pbus resident devices access to the C38xx system memory. The NIA supports four separate Pbus interfaces with one or two Channel Control Units (CCUs) per Pbus. The NIA also supports an Expansion Pbus called the NXP. The NXP is a high speed ECL version of the Convex Pbus. Soon to be designed CCUs (called XIOPs for eXpansion I/O Processors) will plug into the NXP yielding maximum transfers rates of up to three times the current Pbus. The NIA can support only one XIOP, however C38xx systems can be configured with multiple NIAs and thus Multiple XIOPs.

1.1 Physical Description

The NIA is implemented on a tightly packed, doublesided, C38xx size board. The NIA consists of discrete PALs, ECLiPS components, gate arrays, self-timed RAMs, ECL-TTL translators, and a variety of terminators. The bulk of the NIA is implemented in ECL logic (100K levels). The gate arrays have ECL I/O with GAS internals and require a -2V supply. The self-timed RAMs have ECL I/O, but are powered by -5.2V supply. The rest of the NIA's ECL logic is powered by -4.5V supply. The translators require a +5V supply as well as a -4.5V supply. The NIA uses 799 out of 800 available signal I/O pins on the board. Roughly half of these I/O pins are Pbus related TTL level signals

Like all C38xx system boards, the NIA is powered from a power pallet attached to the front of the board. The power pallet provides +5V, -5V, -4.5V and -2V power to the board. The power pallet provides these different supply voltages with DC-DC conversion from a 300V input supply from the Bay power system. The total power dissipation of the NIA is just under 1000W.

A C38xx system can be configured with one to eight NIAs. There will always be one NIA configured in the I/O Bay of a system. Optional expansion bays may contain one or two NIAs. Within the I/O Bay, the NIA plugs into the I/O bay backplane. Other boards that plug into the I/O Bay backplane include the CPU Utilities board (NCU) and an optional XIOP. The XIOP is a yet to be designed CCU that attaches to the NIA's NXP interface. The NXP is routed in the I/O Bay backplane. The NCU also attaches to the NXP, giving the SPU access to system memory. See Figure 1-1

The Pbuses that interface to the NIA are routed to connectors at each end of the I/O Bay backplane (two Pbuses to one end and two to the other end). Controlled impedance ribbon cables are used to attach the Pbuses to CCU backplanes located on each side of the I/O Bay card cage. Each CCU backplane holds four CCUs, with two CCUs per Pbus. This configuration allows for up to eight CCUs attached to four Pbuses in one I/O Bay cabinet.

Expansion Bays are configured slightly different from the I/O Bay. In an expansion Bay, the NIA supports four Pbuses with only one CCU per Pbus. However, up to two NIAs may be configured in each Expansion I/O Bay. Also, one XIOP may be configured with each NIA. Thus an NIA/XIOP pair can replace an NSP/NVP (processor) pair for mix and match CPU-I/O system configurations. See Figure 1-2.

Figure 1-1 Base I/O

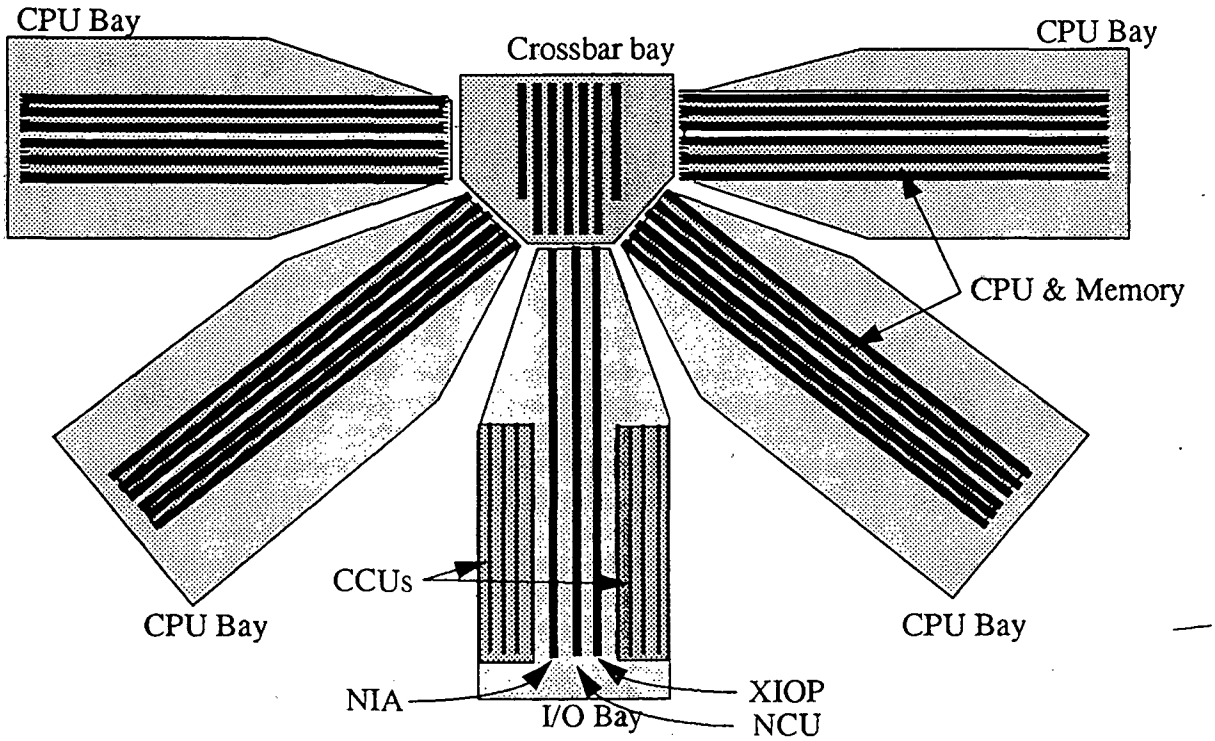
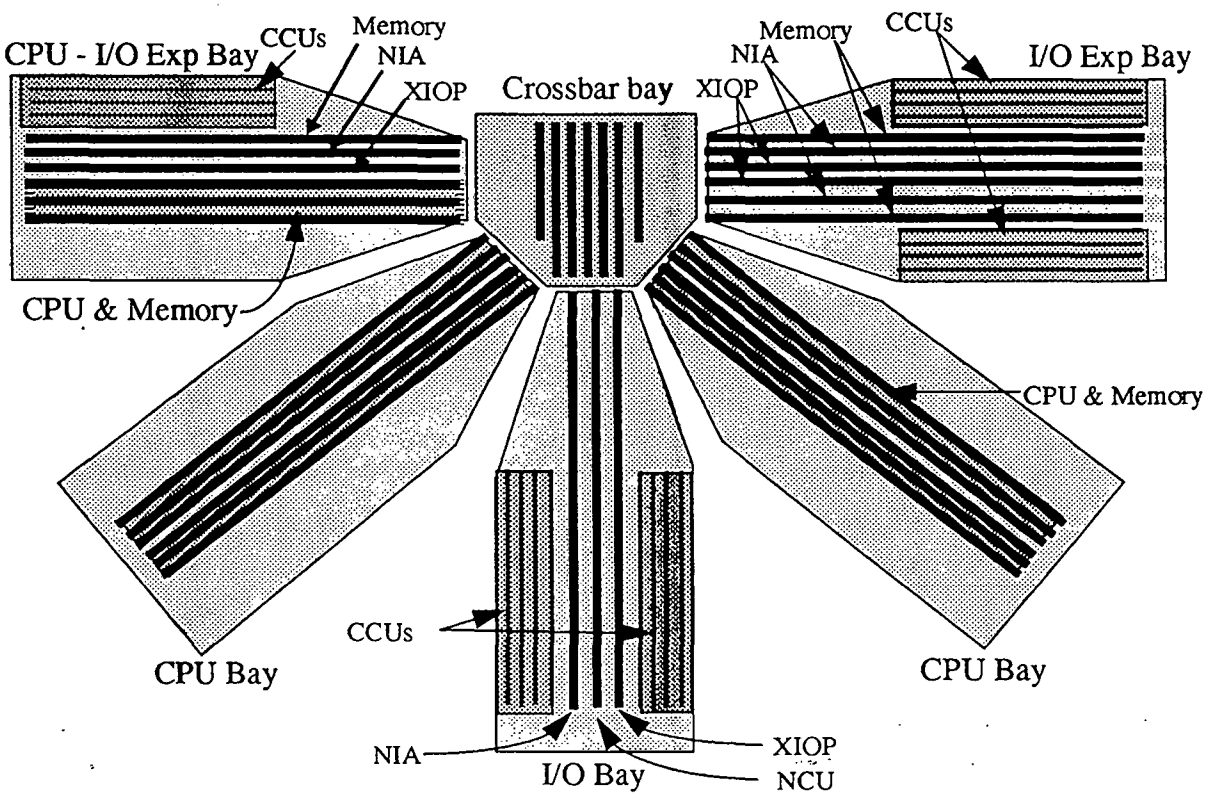


Figure 1-2 Expansion I/O



1.2 Major Subsystems

The NIA has six major subsystems. Each subsystem contains one or more smaller "sub-blocks" which split data path and control functions. In all, there are 15 of these sub-blocks on the NIA. Some of the sub-blocks are replicated as in the case of the Pbus control interfaces. All 15 sub-blocks are defined in the NIA schematics. A brief description of each subsystem is given below. A detailed description of each subsystem is described in the following chapters

1.2.1 I/O Channel Interface

The I/O Channel Interface contains the logic that interfaces to the four Pbus channels and the NXP channel. The I/O Channel interface contains the four Pbus interface blocks (PBI0-PBI3), the NXP interface block (NXI), the Pbus Interrupt interface (PI), and the eight CDS gate arrays (CDS_ARRAYS). Each independent PBIx and NXI block controls it's own I/O channel interface and contains bus arbitration logic, a protocol state machine, header decode, error detection and error logging. The PI block contains the Pbus interrupt bus state machine, staging registers, and interrupt bus arbitration. The CDS_ARRAYS block contains the eight CDS arrays. Each CDS array is a eight bit slice of each I/O Channel interface. The CDS arrays provide read and write data staging, data rate matching, write data multiplexing, header multiplexing and header update logic.

1.2.2 Write Data Queue and Arbitration Logic

The Write Data Queue (WDQ) buffers write data between the I/O channels and system memory. The WDQ is logically split into five smaller queues, one for each I/O channel interface. Physically, the WDQ is implemented using 2kx9 self-timed RAMS (STRAMS). These STRAMs are duty cycled, meaning that a read and a write access occur in one system clock. This allows the NIA to write new data from the I/O channels without interfering with the read access of previously written data on its way to memory.

Access to the WDQ is controlled by the Write Queue Arbitration logic (WQA). The WQA schedules write accesses to the WDQ based upon demand and predefined priorities. In general, the Pbus I/O channels are given the highest priority to the WDQ. The NXP gets all of the remaining available write cycles to the WDQ. Read accesses to the WDQ are controlled in part by the Crossbar Interface control logic (XBI). Reads are performed in preparation for write transfers to system memory by the XBI. No arbitration of read access is necessary since the XBI is the only resource that reads the WDQ. All 72 bits of the WDQ are accessed at the same time for both read and write operations.

1.2.3 Read Data Queue and Arbitration Logic

The Read Data Queue (RDQ) buffers read data between system memory and the I/O channels. The RDQ is organized and functions very similar to the WDQ described above. Each I/O channel has it's own smaller queue implemented in the RDQ. The RDQ is also duty cycled like the WDQ to allow one read and one write access to the RDQ each system clock cycle.

Access to the RDQ is controlled by the Read Queue Arbitration logic (RQA). The RQA schedules read accesses to the RDQ based upon demand and predefined priorities. Like the WQA, Pbus I/O channels are given the highest priority to the

RDQ. Write access to the RDQ is governed by return data from main memory. However unlike the WDQ, the RDQ can be written in 36 bits at a time. This is because the memory and crossbar system is split into independent even and odd words. Meaning, the even and odd words of a long word container can return to the NIA on different cycles. Read accesses to the RDQ occur as 72 bit operations.

1.2.4 Port Arbitration

I/O channel access to the memory interface is controlled by the Port Arbitration (PA) logic. The PA logic arbitrates I/O channel access to the memory interface and tracks I/O channel requests through the arbitration pipeline. The PA also contains logic that splits large block transfers into smaller 128 or 256 byte memory requests.

1.2.5 Crossbar Interface

The Crossbar Interface contains the logic that talks to the Even and Odd crossbars. The Crossbar interface data path is implemented in the XDS arrays. The XDS gate arrays are contained in the XDS_ARRAYS block. The XDS array performs address generation, board select, cycle and write zone generation, and a read data return FIFO. The Crossbar Interface control logic is contained in the XBI block. The XBI logic controls the Crossbar interface pipeline, generates "request ready" signals, and contains the Crossbar interrupt (Trap) interface.

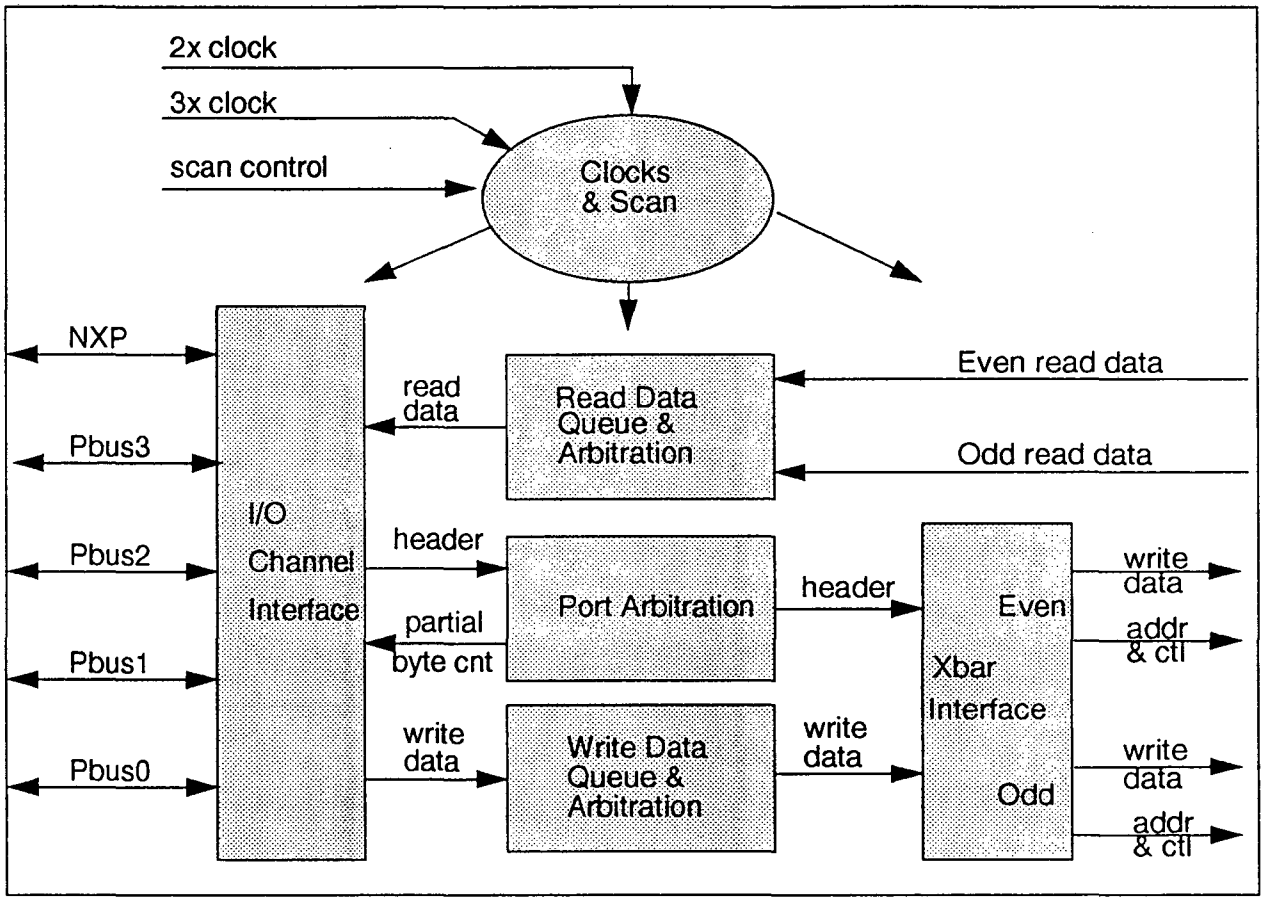
1.2.6 Clock Generation and Scan

The NIA generates over 300 clocks for devices resident on the NIA. The NIA receives a 2x and a 3x clock from the NCU. The 3x clock is generally referred to as the "free running" clock and is used mainly to generate clocks to the CCUs that are connected to the Pbus I/O channels. The NIA contains four different scan rings and uses all eight possible scan modes. The clock generation block (CLK) is the largest single logic block on the NIA.

2 NIA Interfaces

This chapter describes the NIA interfaces. The external interfaces are described first, followed by a description of the major block interfaces internal to the board.

Figure 2-1 Major Subsystems



2.1 External Interfaces

The NIA uses 799 out of 800 signal I/O pins. All 799 signals exit/enter the NIA though the Augat connector located on the backplane edge of the board. The major signal interfaces to the NIA are the even and odd Xbar interfaces, the four Pbus interfaces, and the NXP interface.

The Xbar interface signals and half of the NXP interface signals exit/enter the board in the top two Augat modules. The bottom three Augat modules are used for the Pbus interfaces and the other half of the NXP interface. The I/O signals are arranged in this fashion for two reasons. First, the Xbar interface signals must be located near the top of the board in order to reduce the amount of wire between the NIA and the Xbar boards. Second, the number of ECL to TTL signal boarders is kept to a minimum. Keeping ECL and TTL signals separated helps reduce the chance of crosstalk between the two logic families.

The NIA also has two 30-pin card edge connectors located on the power pallet

edge of the board. These two J-connectors are used for signal I/O to the Power Pallet Controller located on the NIA Power Pallet. These signals include the port and slot IDs from the backplane, a hard coded board ID, voltage and temperature sense lines, and access to the Serial EEPROM.

Table 2-1 NIA I/O Signals

Pbus Interface0	90	NXP Interface	84
Pbus Interface1	90	NXP Interrupt Interface	13
Pbus Interface2	90	Even Xbar Interface	122
Pbus Interface3	90	Odd Xbar Interface	122
Pbus Interrupt Interface	36	Xbar Interrupt Interface	16
Shared Pbus Diag Ctl	12	Clocks, Scan, and Misc.	34
TTL signal Total	408	ECL signal Total	391
Total signal I/O	799		

2.1.1 Pbus Interfaces

The NIA contains four independent Pbus interfaces. Each Pbus interface supports two CCUs. The Pbus is a bidirectional, 64 bit wide (plus 8 parity bits) data bus. Even parity is expected on the Pbus. The Pbus interface is implemented in TTL logic and cycles at 100ns (80 MBytes/sec). Data transfers across the Pbus occur in block mode fashion. A header transfer precedes each block of transfers. The header specifies the transfer type, a starting address, and a byte count for the total number of bytes requested. A detailed description of the Pbus can be found in the Convex Input/Output Bus (PBUS) Functional Specification, document No 081-000109-000. The following paragraphs describe the Pbus interface as it is used during transfers to and from the NIA.

Table 2-2 Pbus Interface Signals

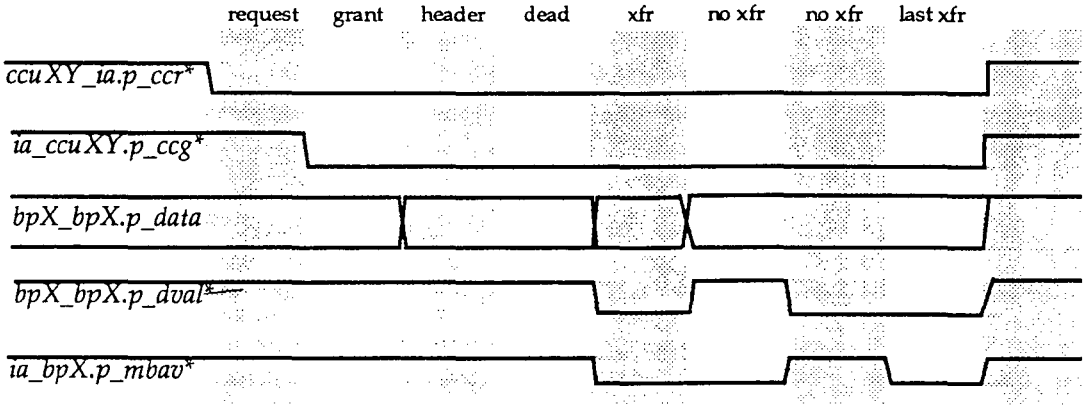
BIDIRECTIONAL	
bpX_bpX.p_data<63..0>	Pbus X data bus
bpX_bpX.p_par<7..0>	Pbus X parity
bpX_bpX.p_dval*	Pbus X data valid
INPUTS	
bpX_ia.p_cbav*	Pbus X channel buffer available
ccuX0_ia.err*	Pbus X ccu0 hard error
ccuX1_ia.err*	Pbus X ccu1 hard error
ccuX0_ia.p_ccr*	Pbus X ccu0 bus request
ccuX1_ia.p_ccr*	Pbus X ccu1 bus request
OUTPUTS	
ia_bpX.p_buserr*	Pbus X bus error
ia_bpX.p_mbav*	Pbus X memory buffer available
ia_ccuX0.p_ccg*	Pbus X ccu0 bus grant
ia_ccuX1.p_ccg*	Pbus X ccu1 bus grant

Transfers on the Pbus are always initiated by a CCU. Each Pbus interface on the NIA supports two CCUs. A CCU can request the use of the Pbus by asserting its *ccuXY_ia.p_ccr** bus request signal. The NIA arbitrates access to the Pbus and will eventually grant the Pbus to the CCU by asserting *ia_ccuXY.p_ccg**. Once the bus grant reaches the CCU, the CCU will drive a transfer header on the Pbus on the

following Pbus cycle. The NIA uses the cycle following the header transfer to decode the header. If all is good, the NIA or the CCU may begin driving data on the Pbus the cycle following the decode cycle. For write transfers the CCU drives the Pbus with write data. For read transfers, "test and modify" type transfers, and I/O space reads, the NIA drives the Pbus with read return data.

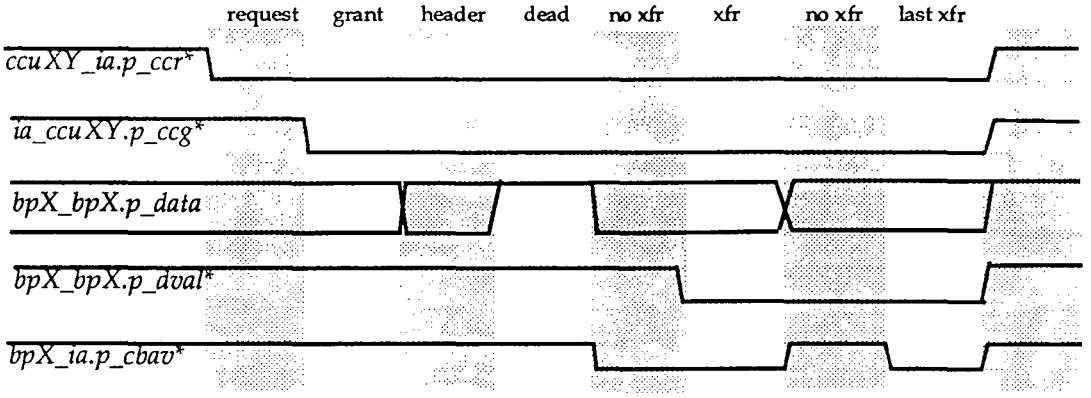
During write transfers, the CCU will assert the data valid signal, *bpX_bpX.p_dval**, when the CCU drives the Pbus with valid write data. The NIA can hold off the CCU's write data transfer by not asserting the *ia_bpX.p_mbav** memory buffer available signal. A write data transfers is completed when the bus request, bus grant, data valid and memory buffer available signals are all true. See Figure 2-2.

Figure 2-2 Pbus Write Transfers



During read transfers, the NIA will assert the data valid signal, *bpX_bpX.p_dval**, when it drives the Pbus with valid read data. The CCU can hold off the NIA's read data transfer by not asserting the *bpX_ia.p_cbav** channel buffer available signal. A read data transfers is completed when the bus request, bus grant, data valid and channel buffer available signals are all true. See Figure 2-3.

Figure 2-3 Pbus Read Transfers



If the NIA detects an error with the header or a data transfer, the NIA will assert the bus error signal *ia_bpX.p_buserr** to terminate the transfer. The following conditions will cause a Pbus error: header parity error, illegal header, write data parity error, write PCM violation, and read PCM violation. If a CCU detects an error on the Pbus or has a hard error internal to itself, it can alert the NIA by asserting its hard error signal, *ccuXY_ia.err**. The CCU hard error signals and all of the bus error conditions are recorded into a soft error log that can be read by the service processor (SPU).

2.1.2 Pbus Interrupt Interface

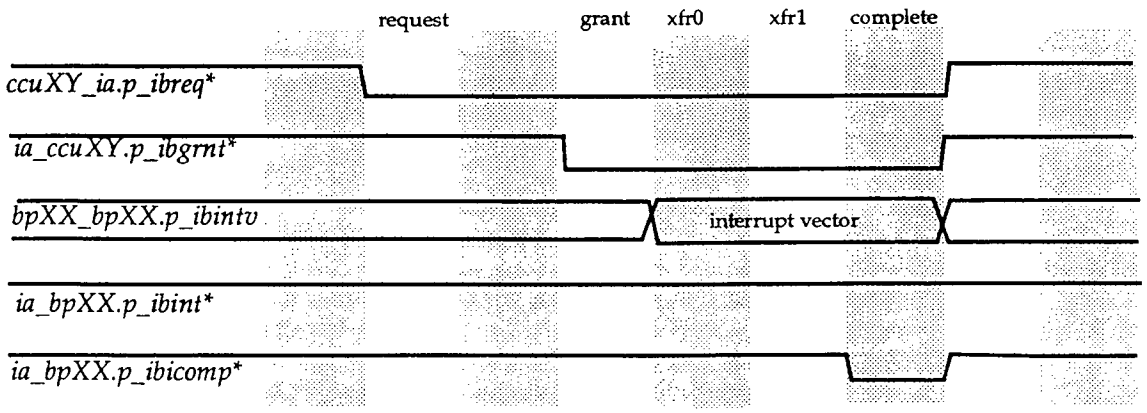
The Pbus interrupt interface provides interrupts to and from the CCUs. The NIA supports four Pbus data channels, but only one (logical) Pbus interrupt channel. All eight CCUs are connected to the Pbus interrupt bus. The NIA supports two physical Pbus interrupt busses: one for the CCUs connected to the left side CCU backplane and one for the CCUs connected to the right. Only one of the eight CCUs can source an interrupt to the NIA at one time. Like the Pbus data interface, the interrupt interface is implemented in TTL logic and runs at a 100ns bus cycle time.

Table 2-3 Pbus Interrupt Interface Signals

bp01_bp01.p_ibintv<7..0>	interrupt bus for channels 0 & 1
bp23_bp23.p_ibintv<7..0>	interrupt bus for channels 2 & 3
ia_bp01.p_ibint*	interrupt bus valid for channels 01
ia_bp23.p_ibint*	interrupt bus valid for channels 2 3
ia_bp01.p_ibicomp*	interrupt bus complete for channels 01
ia_bp23.p_ibicomp*	interrupt bus complete for channels 2 3
ia_ccuXY.p_ibgrnt*	grant for ccu xy = 00,01,10,11,20,21,30 & 31
ccuXY_ia.p_ibreq*	request for ccu xy= 00,01,10,11,20,21,30 & 31

Interrupt transfers on the Pbus occur in two directions. Interrupt transfers can be initiated by either a CCU or the NIA. CCU initiated interrupts are sent to the NIA and then forwarded to the NCU. To initiate a transfer, the CCU must request the use of the interrupt bus from the NIA by asserting *ccuXY_ia.p_ibreq**. The NIA arbitrates access to the interrupt bus and will grant the interrupt bus to a requesting CCU by asserting *ia_ccuXY.p_ibgrnt**. When a CCU has been granted the bus, it drives the bus with an interrupt vector (data). The vector is driven by the CCU for three consecutive Pbus cycles. The interrupt vector is registered on the NIA at the end of the second cycle. The NIA then asserts the complete signal, *ia_bpXX.p_ibicomp**, during the third cycle. At the end of the third cycle, the NIA will remove the CCU's grant from the bus. See Figure 2-4. The interrupt valid signal, *ia_bpXX.p_ibint**, is not asserted by the NIA during CCU initiated transfers. The NIA asserts interrupt valid only during NIA initiated transfers.

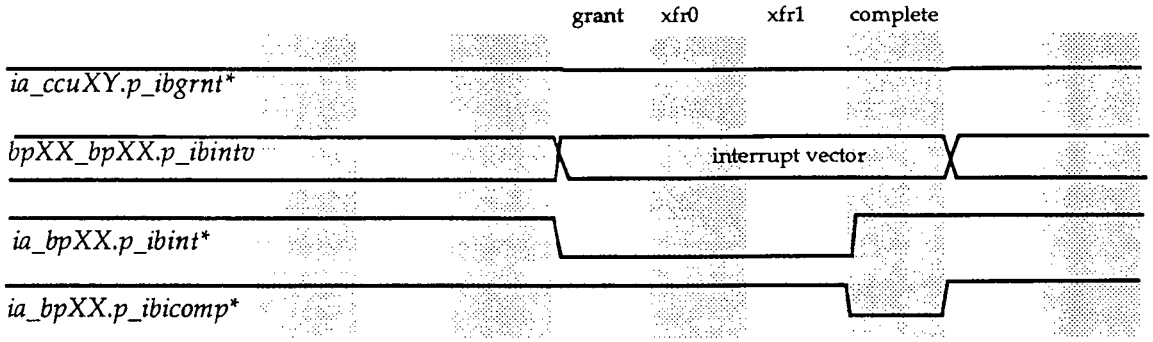
Figure 2-4 CCU Initiated Pbus Interrupt Cycle



NIA initiated interrupt transfers follow the same basic flow as just described, except there is no request and grant action. The NIA will, in essence, grant itself

the interrupt bus when it has received an interrupt from the NCU provided the interrupt bus is not currently granted to a CCU. The NIA will drive the interrupt vector for four consecutive cycles. The interrupt valid signal, *ia_bpXX.p_ibint**, is asserted for the first three cycles. The interrupt complete signal, *ia_bpXX.p_ibicomp**, is asserted on the fourth cycle. See Figure 2-5.

Figure 2-5 NIA Initiated Pbus Interrupt Cycle



2.1.3 Pbus Clocks and Scan Control

The NIA sources clocks and scan control signals to Pbus resident CCUs. Each CCU receives three clock signals from the NIA: one 20 Mhz clock and two 10 Mhz "phase" clocks. One phase clock, *ia_ccuXY.gphase*, provides a free running clock to the CCU while the other phase clock, *ia_ccuXY.mphase*, is stoppable. Each CCU clock is generated from the NIA's 3x clock input.

The NIA also provides scan control and scan data signals to the CCUs. Scan control and scan data signals are provided separately to each CCU backplane (left and right). However, only one CCU can be scanned at a time. The CCU scan control signals are asserted on the NIA by scanning the NIA's CCU scan control ring.

Table 2-4 Pbus Clocks and Scan Control

ia_ccuXY.20mhz	20 Mhz clock to CCU XY
ia_ccuXY.gphase	Gphase clock to CCU XY (free running)
ia_ccuXY.mphase	Mphase clock to CCU XY (stoppable)
ia_ccuXY.odena*	CCU XY output data enable
ia_bp01.ccdmode*	CCU diagnostic mode to Pbuses 0 & 1
ia_bp23.ccdmode*	CCU diagnostic mode to Pbuses 2 & 3
ia_bp01.ccldram*	CCU load RAM to Pbuses 0 & 1
ia_bp23.ccldram*	CCU load RAM to Pbuses 2 & 3
ia_bp01.ccclear*	CCU clear to Pbuses 0 & 1
ia_bp23.ccclear*	CCU clear to Pbuses 2 & 3
ia_bp01.scandat	Scan data from NIA to Pbuses 0 & 1
ia_bp23.scandat	Scan data from NIA to Pbuses 2 & 3
bp01_ia.scandat	Scan data from Pbuses 0 & 1 to NIA
bp23_ia.scandat	Scan data from Pbuses 2 & 3 to NIA
ia_bp03.ccsctl<1..0>	Scan control to Pbuses 0,1,2, & 3

2.1.4 Expansion Pbus (NXP) Interface

The Expansion Pbus (NXP) Interface is basically an ECL version of the Pbus interface. The NXP interface bus cycle time is 33.3ns (2 x 16.67) yielding 240 MBytes/sec throughput. Odd parity is expected on the NXP. The NXP interface signals are the same as the Pbus interface, except for the data valid signal. The Pbus interface has a bidirectional data valid, where as the NXP has two data valid signals, *ia_xiop.rd_val* and *xiop_ia.wrt_val*. The NXP interrupt interface is also similar to the Pbus interrupt interface, except the NXP interrupt interface does not include an "interrupt cycle complete" signal.

Table 2-5 NXP Interface Signals

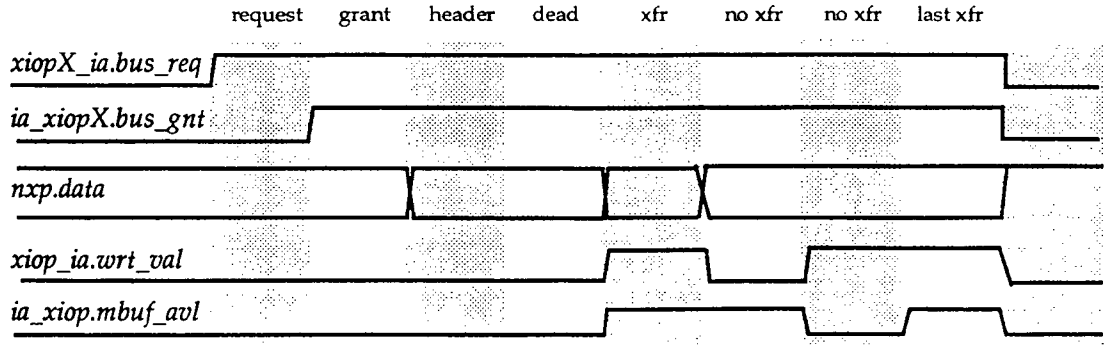
BIDIRECTIONAL	
nxp.data<63..0>	NXP data bus
nxp.par<7..0>	NXP data parity
nxp.ibintv<7..0>	NXP interrupt bus
INPUTS	
xiop_ia.wrt_val	write data valid
xiop_ia.cbuf_avl	channel buffer available
xiopX_ia.err	XIOP X hard error
xiopX_ia.bus_req	XIOP X data bus request
xiopX_ia.ibreq	XIOP X interrupt bus request
OUTPUTS	
ia_xiop.bus_err	bus error
ia_xiop.mbuf_avl	memory buffer available
ia_xiop.rd_val	read data valid
ia_xiopX.bus_gnt	XIOP X data bus grant
ia_xiopX.ibgrnt	XIOP X interrupt bus grant
ia_xiop.ibint	interrupt bus valid

2.1.4.1 NXP Data Transfers

The NXP data interface has the same transfer states as the Pbus interface. The NXP bus is a block mode transfer channel supporting two XIOPs (eXpansion I/O Processors). The only instance where two XIOPs would share an NXP bus would be in the Base I/O bay. In the Base I/O bay, the NCU shares the NXP bus with an optional XIOP. The NCU uses the NXP as a system memory interface for the SPU.

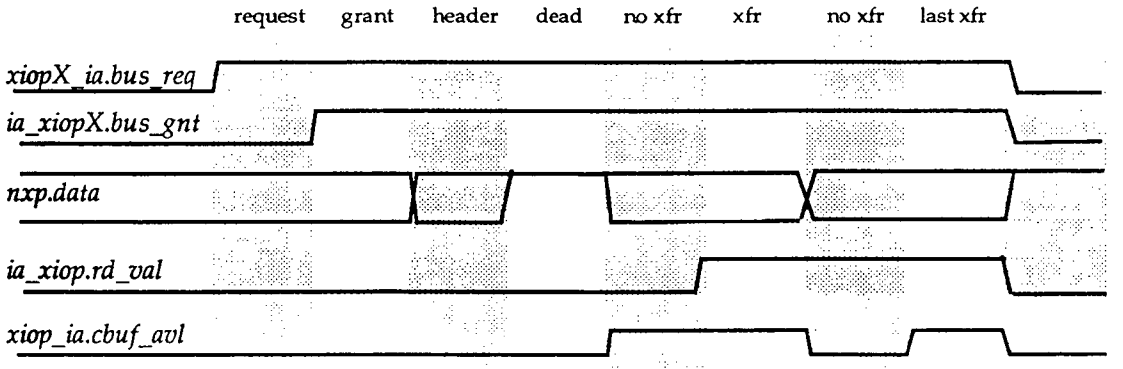
The XIOP (or SPU) requests the NXP bus by asserting its *xiopX_ia.bus_req* signal. The NIA grants the XIOP the bus by asserting the *ia_xiop.bus_gnt* signal back to the requesting XIOP. The XIOP then drives the NXP bus with header information, followed by a dead cycle, followed by one or more transfer cycles. If the transfer is a memory write or memory scrub operation, the XIOP drives the NXP bus with data. If the transfer is a memory read, I/O space read, or a "test and modify" type operation, the XIOP gets off the NXP after the header transfer and the NIA drives the NXP when the requested data is available.

Figure 2-6 NXP Write Transfers



The bus request and bus grant signals must remain asserted during all header and data transfers. During write transfers, the XIOP asserts *xiop_ia.wrt_val* when the NXP contains valid write data. The NIA asserts *ia_xiop.mbuf_avl* when it can receive write data. Both write data valid and memory buffer available must be asserted together for a write data transfer to occur. See Figure 2-6. During read transfers, the NIA asserts *ia_xiop.rd_val* when the NXP contains valid read data. The XIOP asserts *xiop_ia.cbuf_avl* when it can receive read data. Both read data valid and channel buffer available must be asserted together for a read data transfer to occur. See Figure 2-7.

Figure 2-7 NXP Read Transfers



The NIA asserts the bus error signal, *ia_xiop.bus_err* when ever an error condition is detected on the NXP. Error conditions include, header parity error, illegal header, write data parity error, write PCM violation, and read PCM violation. The XIOP can assert its hard error signal, *xiopX_ia.err*, to the NIA whenever conditions on the XIOP warrant. The bus error conditions detected by the NIA and the hard error indications from the XIOPs are registered in a soft error log on the NIA. The soft error log is available to the SPU by scanning the NIA's soft error log ring.

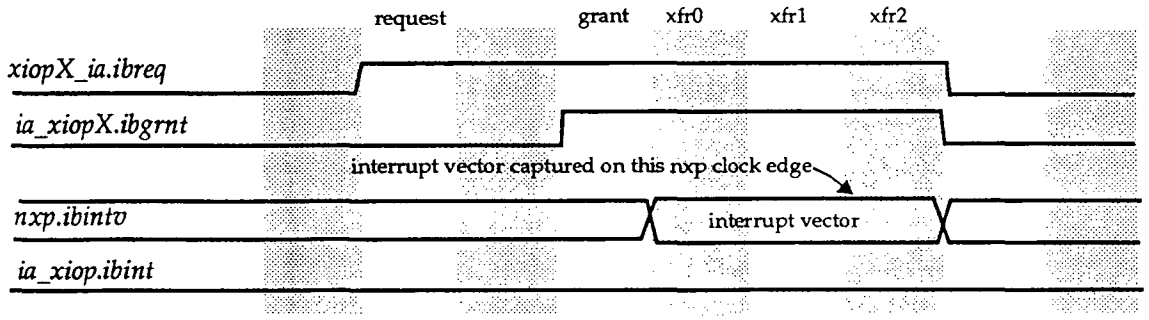
2.1.4.2 NXP Interrupt Transfers

The NXP interrupt bus is a bidirectional, ECL bus. The NXP interrupt bus supports interrupts in two directions: interrupts sourced by an XIOP and interrupts sourced by the NIA. The NIA sources the NXP interrupt bus on behalf of the NCU. The NXP interrupt bus cycle is similar to the Pbus interrupt cycle.

An XIOP must request to use the NXP interrupt bus by asserting its *xiopX_ia.ibreq* signal. The NIA will grant the NXP the interrupt bus by asserting *ia_xiopX.ibgnt* back to the requesting XIOP. Once the XIOP receives the interrupt bus grant, it

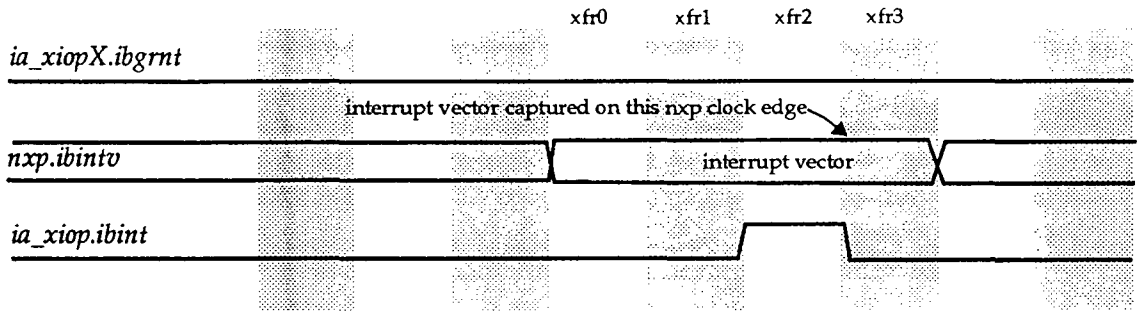
drives the interrupt bus with its interrupt vector for three consecutive NXP cycles. The NIA captures the interrupt vector at the end of the second cycle. At the end of the third cycle, the NIA removes the XIOP grant. The NIA does not assert interrupt bus valid, *ia_xiop.ibint*, during XIOP initiated transfers.

Figure 2-8 XIOP Initiated NXP Interrupt Transfer



NIA initiated interrupt transfers have no request or grant states to speak of. Once the NIA receives an interrupt from the NCU, it waits until the NXP interrupt bus is free (no XIOP has the interrupt bus grant) and then drives the interrupt bus for four consecutive NXP cycles. The NIA asserts the interrupt valid signal during the third cycle. The XIOPs capture the interrupt vector at the end of the third cycle. The NIA then removes its drivers from the interrupt bus at the end of the fourth cycle.

Figure 2-9 NIA Initiated NXP Interrupt Transfer



2.1.5 Even Crossbar Send Interface

The crossbar interface is split into even and odd sides. The even side interface accesses the even word of a long word container, bits 63..32. The even side send interface is used by the NIA to access control registers on the NCU as well as memory.

Table 2-6 Even Crossbar Send Interface

ia_xse.addr<28..3>	memory address
ia_xse.cycle<1..0>	transfer cycle type
ia_xse.bd_sel<3..0>	memory board select
ia_xse.wr_zone<3..0>	write zone (byte) enables
ia_xse.ctl_par<4..0>	parity bits (odd parity) for the signals above
ia_xse.wr_data<31..0>	write data to memory
ia_xse.wr_par<3..0>	write data parity (odd)
ia_xse.rdy	transfer valid (ready) to crossbar
xse_ia.a_req_next	A side request next

Table 2-6 Even Crossbar Send Interface continued

xse_ia.b_req_next	B side request next
xse_ia.a_st_pend	A side store pending
xse_ia.b_st_pend	B side store pending
xse_ia.a_req_pend	A side request pending
xse_ia.b_req_pend	B side request pending

The even crossbar interface has parity protection for the address, cycle, board select, write zones and write data fields. Odd parity is generated. The parity bits *ia_xse.wr_par<3..0>* cover the write data field and *ia_xse.ctl_par<4..0>* covers the address and control fields. See Figure 2-10.

Figure 2-10 Even Crossbar Send Interface Parity

Write Data Parity

ia_xse.wr_par<3>	ia_xse.wr_par<2>	ia_xse.wr_par<1>	ia_xse.wr_par<0>
ia_xse.wr_data<7..0>	ia_xse.wr_data<15..8>	ia.xse.wr_data<23..16>	ia_xse.wr_data<31..24>

Control Parity

ia_xse.ctl_par<4>	ia_xse.ctl_par<3>	ia_xse.ctl_par<2>	ia_xse.ctl_par<1>	ia_xse.ctl_par<0>
ia_xse.wr_zone<3..0>	ia_xse.addr<7..3> ia_xse.cycle<1..0>	ia_xse.addr<14..8>	ia_xse.addr<21..15>	ia_xse.addr<28..22>

The transfer type is defined by the cycle field, *ia_xse.cycle<1..0>*. The NIA can send a NOP, memory read, memory write, or a test and modify transfer to the crossbar. See Figure 2-11.

Figure 2-11 Even Crossbar Cycle Field Encoding

ia_xse.cycle<1..0>	operation
0	NOP
1	Read
2	Write
3	Test and Modify

Memory write transfers are controlled by the write zones, *ia_xse.wr_zone<3..0>*. Each bit of the write zone enables a different byte in the write data field *ia_xse.wr_data<31..0>*. See Figure 2-12.

Figure 2-12 Even Crossbar Write Zones

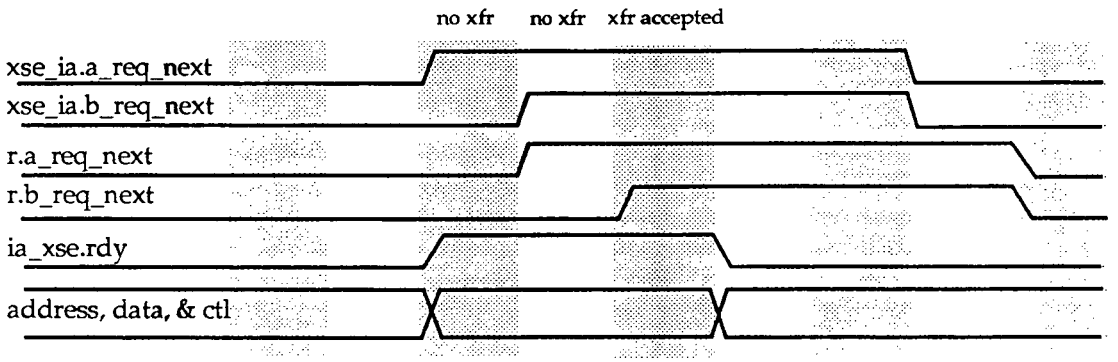
Write Zones

ia_xse.wr_zoner<3>	ia_xse.wr_zone<2>	ia_xse.wr_zone<1>	ia_xse.wr_zone<0>
ia_xse.wr_data<7..0>	ia_xse.wr_data<15..8>	ia.xse.wr_data<23..16>	ia_xse.wr_data<31..24>

The NIA initiates a transfer by asserting the ready signal, *ia_xse.rdy*, with the proper address, write data, and control for the desired operation. The even side

crossbar supplies request next signals, *xse_ia.a_req_next* and *xse_ia.b_req_next*, to the NIA. These signals tell the NIA if the crossbar can accept the current transfer on the even crossbar interface. Both request next signals must be true on the cycle before the NIA's ready is asserted before the transfer is completed. The store pending signals, *xse_ia.a_st_pend* and *xse_ia.b_st_pend* are checked by the NIA during "test and modify" transfers. Both store pending signals must be false one cycle prior to the NIA sending a test and modify transfer. Likewise, the request pending signals, *xse_ia.a_req_pend* and *xse_ia.b_req_pend*, are checked by the NIA prior to sending an NCU transfer. An NCU transfer is a crossbar transfer with a board select value of 8 or 9. The only NIA transfers to the NCU are Time of Century (TOC) reads, Memory Base Pointer (MBP) reads, and Trap register (interrupt) writes. Both request pending signals must be false one cycle prior to the NIA sending a transfer to the NCU.

Figure 2-13 Even Side Crossbar Transfer



2.1.6 Even Crossbar Receive Interface

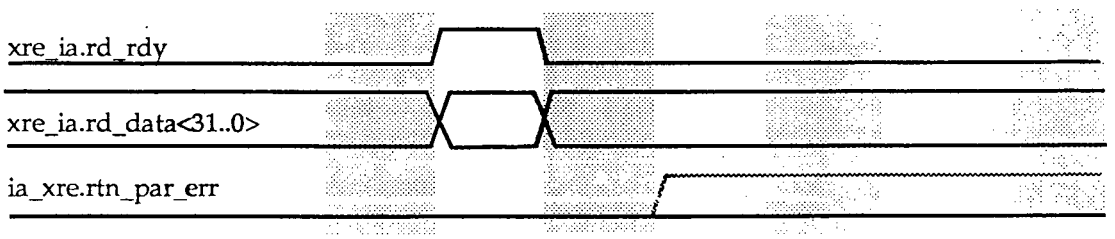
The even side crossbar receive interface is used to receive memory read data or data requested from the NCU. The even side interface supplies the even word of a long word container, bits 63..32.

Table 2-7 Even Crossbar Receive Interface Signals

xre_ia.rd_data<31..0>	read data bus
xre_ia.rd_par<3..0>	read data parity
xre_ia.rd_rdy	read data bus ready (valid)
ia_xre.rtn_par_err	return data parity error

The even side crossbar asserts the read ready signal, *xre_ia.rd_rdy*, when the read data bus has valid data. The NIA must accept the read data from the crossbar. Therefore, the NIA will request read data only when it can accept read data. The NIA checks for odd parity on the read return data and asserts the return data parity error signal, *ia_xre.rtn_par_err*, if the parity is bad.

Figure 2-14 Even Side Crossbar Return Transfer



2.1.7 Odd Side Crossbar Send Interface

The odd side interface accesses the odd word of a long word container, bits 31..0. The odd side send interface is used by the NIA to access control registers on the NCU as well as memory. The odd side send interface is identical to the even side send interface. The odd side interface signal names are prefixed with either *ia_xso* or *xso_ia* depending upon the direction of the signal. Refer toEven Crossbar Send Interface on page 12 for details.

2.1.8 Odd Crossbar Receive Interface

The odd side crossbar receive interface is used to receive memory read data or data requested from the NCU. The odd side interface supplies the odd word of a long word container, bits 31..0. The odd side receive interface is identical to the even side receive interface. The odd side interface signal names are prefixed with either *ia_xro* or *xro_ia* depending upon the direction of the signal. Refer to Even Crossbar Receive Interface on page 14 for details.

2.1.9 Trap Vector Bus Interface

The trap vector bus is used by the NIA to receive interrupt transfers from the NCU. The trap vector bus is 12 bits wide, however the NIA only needs the lower 8 bits of the bus. The trap vector bus is a general purpose trap bus, however the NIA only uses it to receive interrupts. When an interrupt transfer is received on the trap vector bus, the interrupt is forwarded to the Pbus and NXP channels by the Pbus Interrupt and the NXP interrupt interfaces respectively.

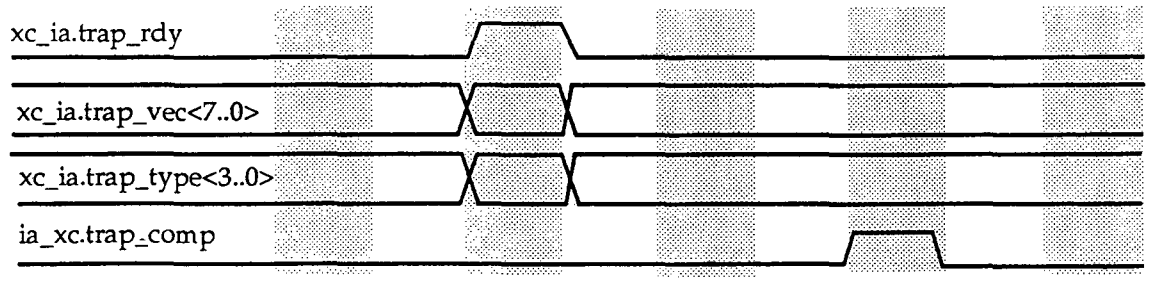
Table 2-8 Trap Vector Bus Interface Signals

xc_ia.trap_vec<7..0>	Trap vector bus
xc_ia.trap_type<3..0>	Trap type
xc_ia.trap_rdy	Trap vector bus ready (valid)
ia_xc.trap_comp	Trap complete
xc_ia.status_en	Trap send status enable
xc_ia.status	Trap send status

The crossbar asserts the trap ready signal, *xc_ia.trap_rdy*, when the trap vector bus has valid data. The NIA examines the trap type, *xc_ia.trap_type*<3..0>,to determine if the trap vector bus contains an interrupt or a general purpose trap. If the trap vector bus contains an interrupt, the NIA forwards the vector to the Pbus and NXP interrupt interfaces. Otherwise, the NIA ignores the trap vector data and continues looking for interrupts. Once an interrupt has been received by the NIA and forwarded to the Pbus and NXP interrupt interfaces, the NIA acknowledges the interrupt transfer by asserting the trap complete signal, *ia_xc.trap_comp*, back to the NCU.

Interrupts are sent to the NCU via the crossbar interface. When the NIA sends an interrupt to the NCU, it must wait for status from the NCU that tells the NIA whether or not the interrupt transfer was successful. The trap send status enable, *xc_ia.status_en*, and trap send status, *xc_ia.status*, provide this information. The NIA waits for the status enable to be asserted. Once asserted, the NIA examines the status bit. If the status bit is asserted, the interrupt transfer was successful. If not, the interrupt transfer must be retried. The NIA will continue to retry the interrupt transfer until the transfer is successful.

Figure 2-15 Trap Vector Bus Transfer



2.1.10 Clocks, Scan Control and Miscellaneous Signals

The NIA receives and provides an assortment of clocks, scan controls and miscellaneous signals. Table 2-9 lists and defines these signals.

Table 2-9 Clocks, Scan Control and Misc. Signals

cu_ia.clock_2x *cu_ia.clock_2x**	Differential 2x clock inputs. Used to provide the majority of the clocks used on the NIA.
cu_ia.clock_3x *cu_ia.clock_3x**	Differential 3x clock inputs. Mainly used to generate clocks for the CCUs.
xc_ia.scan_ctl<3..0>	Scan control inputs. Defines the ring and type of scan operation to be performed.
xc_ia.scan_in	Scan data input to the NIA.
ia_xc.scan_out	Scan data output from the NIA.
ia_xc.hard_error	Hard error indication to the SPU. Asserted when the NIA detects a catastrophic error, such as a data parity error.
ia_xc.soft_error	Soft error indication to the SPU. Asserted when the NIA has detected a soft error condition and has logged the error in the soft log scan ring.
xc_ia.slog_ena	Soft error log scan enable. Used in conjunction with the scan control to dynamically scan the soft error log.
xc_ia.reset	NIA reset. Used to perform a non-scan based reset of the NIA. Forces the NIA into a known, quiescent state.
xc_ia.clock_sync	Clock synchronization from the NCU. Used to force synchronization of the NIA and scan engine clock cycle counters.
ia_xiop.clk_sync	Clock synchronization to the XIOPs. Used to force synchronization between the NIA and XIOP NXP clocks.
xc_ia.ccu_clear	CCU clear. Buffered and sent to the CCUs as part of the CCU diagnostic control signals.
xc_ia.ccu_rbe<7..0>	CCU run bit enables 7..0. Used to control the CCU clocks, specifically the Mphase clocks.

Table 2-9 Clocks, Scan Control and Misc. Signals continued

xc_ia.ccu_cntrl_ena	CCU scan control ring scan enable. Used in conjunction with the scan control to dynamically scan the CCU scan control ring.
bp_ia.port_id<3..0>	Backplane port ID. Identifies the backplane's crossbar port.
bp_ia slot_id<3..0>	Backplane slot ID. Identifies the slot number in the given backplane.
bp_ia.lbd_intl_in	Logic board interlock in. Backplane interlock used to determine if the NIA is plugged into the backplane. Connected to *ia_bp.lbd_intl_out* in the backplane.
ia_bp.lbd_intl_out	Logic board interlock out. Backplane interlock used to determine if the NIA is plugged into the backplane. Connected to *bp_ia.lbd_intl_in* in the backplane.

2.2 Internal Interfaces

The following sections describe the internal interfaces between blocks in the NIA. The block diagram shown in Figure 2-1 on page 5 shows the major subsystems and external interfaces of the NIA. In reality, the NIA is composed of several smaller blocks as defined in the schematics and shown in Figure 2-16.

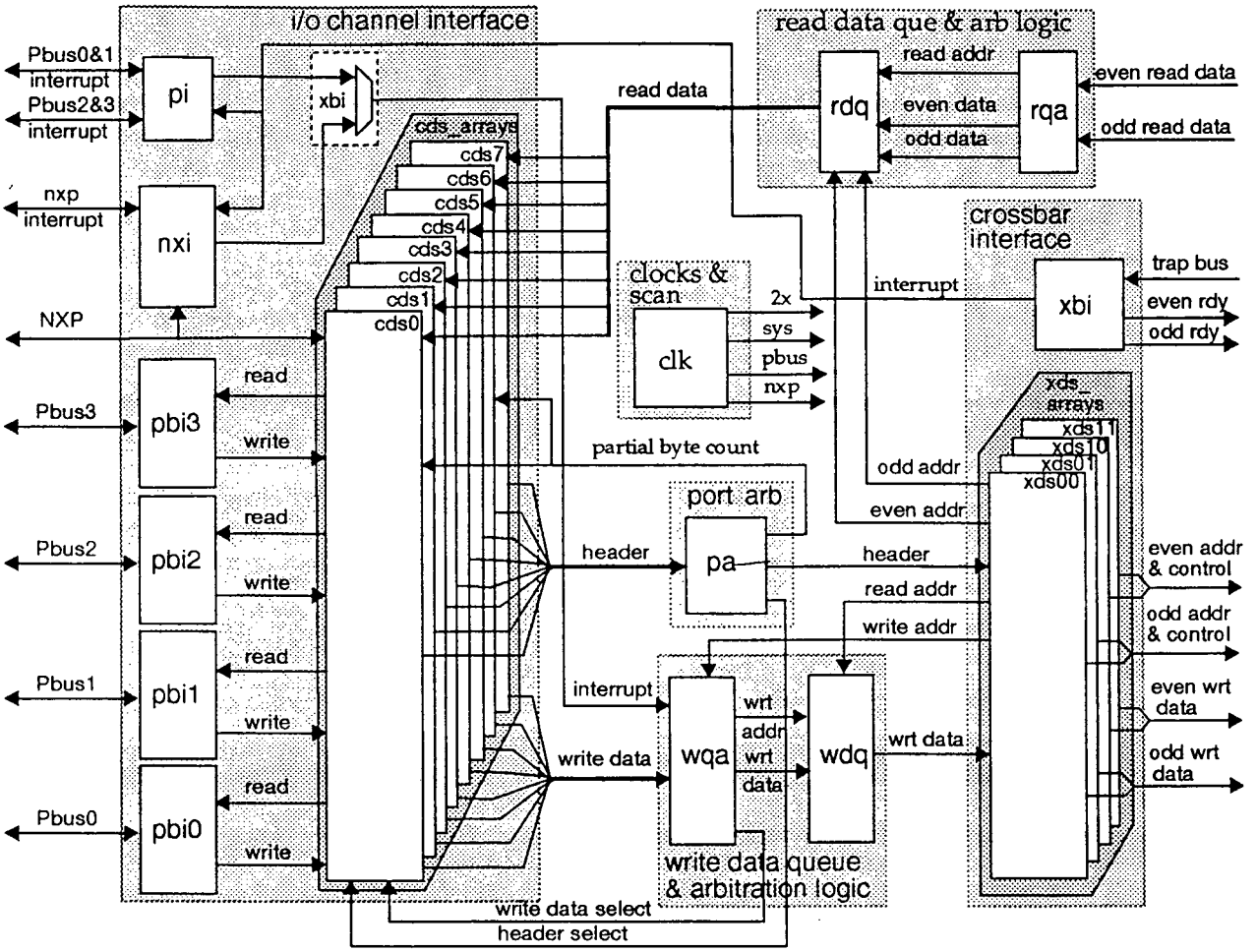
There are fifteen logic blocks that comprise the major subsystems in the NIA. Each logic block is identified with its major subsystem in Figure 2-16 and defined in Table 2-10.

Table 2-10 NIA Logic Blocks

I/O CHANNEL INTERFACE	
pbi0	Pbus 0 Interface
pbi1	Pbus 1 Interface
pbi2	Pbus 2 Interface
pbi3	Pbus 3 Interface
pi	Pbus Interrupt interface (all channels)
nxi	NXP Interface (includes interrupt interface)
cds_arrays	Channel Data Slice (CDS) gate arrays
WRITE DATA QUEUE and ARBITRATION LOGIC	
wdq	Write Data Queue
wqa	Write Queue Arbitration logic
PORT ARBITRATION	
pa	Port Arbitration logic
READ DATA QUEUE and ARBITRATION LOGIC	
rdq	Read Data Queue
rqa	Read Queue Arbitration logic
XBAR INTERFACE	
xbi	Crossbar (xbar) Interface control logic
xds_arrays	Xbar Data Slice (XDS) gate arrays
CLOCKS and SCAN	
clk	Clock generation and scan control logic

The internal interface signals are defined by describing each logic block's outputs. The outputs are grouped with respect to the destination of the interface signals.

Figure 2-16 Detailed Block Diagram of the NIA



2.2.1 Pbus Interfaces - pbiX

Each of the four Pbus channels has an interface block. The interface blocks contain data path and control state. The Pbus interface blocks are numbered 0 through 3. Thus PBIx refers to the pbi0, pbi1, pbi2 and pbi3 Pbus interface blocks. The following sections describe the internal interface signals for all of the Pbus interfaces by using the PBIx designation. Note that the designation pX is also used to refer to each of the four Pbus channels.

2.2.1.1 pbiX to cds_arrays

Table 2-11 pbiX signals to cds_arrays

pbiX.data<63..0>	Pbus X data input bits 63..0
pbiX.par<7..0>	Pbus X data input parity bits 7..0
pbiX.load_hdr[0,1]	Load Pbus X header register
pbiX.rd_hold[0,1]	Read Data register hold (clock enable)

The PBIx logic block provides Pbus data input to the cds_arrays block. The *pbiX.data<63..0>* signals are the outputs from the ttl-to-ecl latch translators. The parity bits, *pbiX.par<7..0>*, are also included. The Pbus data inputs are used to carry write data and header data to the CDS arrays. The CDS arrays provide parity checking of the Pbus data. The PBIx block also provides data path control signals to the CDS arrays. The load header signals, *pbiX.load_hdr0* and *pbiX.load_hdr1*, control the loading of a new header into the partial header register located in the CDS arrays. The read data hold signals, *pbiX.rd_hold0* and *pbiX.rd_hold1*, are clock enables for the Pbus read data registers. The hold signals are asserted when a CCU can not accept read data from the NIA. The hold signals preserve the read data register contents until the CCU can accept the read data return transfer.

2.2.1.2 pbiX to pa

These signals interface to the Port Arbitration logic. They are mostly control signals from the PBIx state machine.

Table 2-12 pbiX signals to pa

pbiX.rd_xfr_abort	Read transfer abort
pbiX.wrt_xfr_abort	Write transfer abort
pbiX.last_wrt_xfr	Last write transfer
pbiX.new_rd_hdr	New read transfer header
pbiX.mbp_hdr	Memory Base Pointer (MBP) header
pbiX.toc_hdr1	Time of Century (TOC) header #1
pbiX.ccu1_gnt	CCU #1 Pbus grant

The transfer abort signals, *pbiX.rd_xfr_abort* and *pbiX.wrt_xfr_abort*, are asserted when a read or write transfer is aborted prior to completion. The read transfer abort signal is used to set a "read flush" state to clear any previously requested read data from the NIA. The write transfer abort is used to force write data from the Write Data Queue (WDQ). The last write signal,*pbiX.last_wrt_xfr*, is also used to force write data from the WDQ to main memory.

The new read header signal, *pbiX.new_rd_hdr*, informs the PA logic to set its read transfer state accordingly. If the header is a read request for the MBP, the *pbiX.mbp_hdr* signal will also be set. This signal informs the PA logic to select the MBP header instead of the PBIx header. The *pbiX.toc_hdr1* signal is used the same way except the PA is to select the TOC header in the CDS arrays. The CCU #1 bus grant signal, *pbiX.ccu1_gnt*, is passed on to the XDS_ARRAYS from the PA logic. This signal is used as the CCU ID that is stored into the read return FIFOs in the XDS arrays.

2.2.1.3 pbiX to rqa and wqa

These signals interface to either the Read Queue Arbitration or Write Queue Arbitration blocks as defined in Table 2-13. Like the previous signals, they are control signals from the PBIx state machine.

Table 2-13 pbiX signals to rqa, wqa

pbiX.rd_hold0	Pbus X Read hold (to rqa)
pbiX.toc_hdr	Time of Century (TOC) header (rqa)
pbiX.new_toc_hdr2	New TOC header #2 (rqa)

Table 2-14 pbiX signals to rqa, wqa continued

pbiX.ccu1_gnt	CCU #1 Pbus grant (rqa)
pbiX.val_wr_data	Valid Pbus write data (wqa)
pbiX.wrt_xfr_abort	Write transfer abort (wqa)

The read hold signal, *pbiX.rd_hold*, is used by the RQA logic to hold off reading the Read Data Queue (RDQ). The signal is asserted when a CCU stops accepting read return data from the NIA. The TOC header, *pbiX.toc_hdr*, and new TOC header2, *pbiX.new_toc_hdr2*, signals are used to read a TOC value from the RDQ. The RQA uses the TOC header signal and the *pbiX.ccu1_gnt* signal to address the appropriate TOC entry in the RDQ. The new TOC header 2 signal is asserted when a TOC header is received for the 2nd byte and beyond. This signal forces the RDQ to increment the RDQ head pointer in order to force a TOC read from the RDQ.

The WQA uses the valid write data signal, *pbiX.val_wrt_data*, to schedule a write access to Pbus X's WDQ. The write transfer abort signal, *pbiX.wrt_xfr_abort*, is also used to schedule a write access and to set the write abort flag in the WDQ.

2.2.1.4 pbiX to clk

Each PBIx logic block sources a soft error, *pbiX.soft_error*, indication to the Clocks and Scan block. The soft errors from each I/O channel interface are ORed together and then sent to the SPU.

2.2.2 Expansion Pbus Interface - nxi

The NXP interface is similar to the Pbus interface just described. However, the NXP includes extra buffering for read and write data paths which must be controlled. The NXI logic block also includes the NXP interrupt bus and interface.

2.2.2.1 nxi to cds_arrays

The NXI logic block provides control signals for the NXP data path implemented in the CDS gate arrays. Unlike the Pbus interface blocks, the NXI does not latch, deskew or translate the NXP bus for the CDS arrays. The CDS arrays drive and receive the NXP bus directly.

Table 2-15 nxi signals to cds_arrays

nxi.load_hdr[0,1]	Load NXP header register
nxi.rd_hold[0,1]	Read Data register hold (clock enable)
nxi.rdxfer[0,1]	Read transfer state (bus enable)
nxi.sel_rd1[a,b]	Select Read Data register 1

The load header signals, *nxi.load_hdr0* and *nxi.load_hdr1*, control the loading of a new header into the partial header register located in the CDS arrays. The read data hold signals, *nxi.rd_hold0* and *nxi.rd_hold1*, are clock enables for the NXP read data output register. The hold signals are asserted when an XIOP can not accept read data from the NIA. The hold signals preserve the read data output register contents until the XIOP can accept the read data return transfer. The read transfer state signals, *nxi.rdxfer0* and *nxi.rdxfer1*, enable the CDS arrays' bus drivers onto the NXP bus. The select read data register 1 signals, *nxi.sel_rd1a* and *nxi.rd_sel1b*, are used to select read data register 1 as input to the read data output register. When these signals are not asserted, read data register 0 is selected.

2.2.2.2 nxi to pa

The NXI interface to the Port Arbitration logic is mostly control signals from the NXP state machine.

Table 2-16 nxi signals to pa

nxi.rd_xfr_abort	Read transfer abort
nxi.wrt_xfr_abort	Write transfer abort
nxi.last_wrt_xfr	Last write transfer
nxi.new_rd_hdr	New read transfer header
nxi.mbp_hdr	Memory Base Pointer (MBP) header
nxi.toc_hdr1	Time of Century (TOC) header #1
nxi.xiop1_gnt	XIOP #1 bus grant

The transfer abort signals, *nxi.rd_xfr_abort* and *nxi.wrt_xfr_abort*, are asserted when a read or write transfer is aborted prior to completion. The read transfer abort signal is used to set a "read flush" state to clear any previously request read data from the NIA. The write transfer abort is used to force write data from the Write Data Queue (WDQ). The last write signal,*nxi.last_wrt_xfr*, is also used to force write data from the WDQ to main memory.

The new read header signal, *nxi.new_rd_hdr*, informs the PA logic to set its NXP read transfer state accordingly. If the header is a read request for the MBP, the *nxi.mbp_hdr* signal will also be set. This signal informs the PA logic to select the MBP header instead of the NXP header. The *nxi.toc_hdr1* signal is used the same way except the PA is to select the TOC header in the CDS arrays. The XIOP #1 bus grant signal, *nxi.xiop1_gnt*, is passed on to the XDS_ARRAYS from the PA logic. This signal is used as the CCU ID that is stored into the read return FIFOs in the XDS arrays.

2.2.2.3 nxi to rqa

The Read Queue Arbitration block tracks the fullness of the NXP read data registers. Therefore, the RQA needs information about when the contents of each read data register will advance to the NXP output stage. The NXI to RQA interface signals provides this information.

Table 2-17 nxi signals to rqa

nxi.rd_hold0	NXP Read hold
nxi.toc_hdr	Time of Century (TOC) header
nxi.new_toc_hdr2	New TOC header #2
nxi.xiop1_gnt	XIOP #1 NXP bus grant
nxi.sel_rd1b	Select Read Data register 1
nxi.rd_val2_rqa	Read Data valid to RQA
nxi.rs_cbuf_avl	Channel Buffer Available

The read hold signal,*nxi.rd_hold0*, is used by the RQA logic to hold off reading NXP data from the Read Data Queue (RDQ). The signal is asserted when an XIOP stops accepting read return data from the NIA. The TOC header signal, *nxi.toc_hdr*, and new TOC header2 signal, *nxi.new_toc_hdr2*, are used to read a TOC value from the RDQ. The RQA uses the TOC header signal and the *nxi.xiop1_gnt* signal to address the appropriate TOC entry in the RDQ. The new TOC header 2 signal is asserted

when a TOC header is received for the 2nd byte and beyond. This signal forces the RDQ to increment the RDQ head pointer in order to force a TOC read from the RDQ.

The select read data register 1 signal, *nxi.sel_rd1b*, is used by the RQA logic to help track the fullness of the NXP read data registers. The control signals, *nxi.rd_val2_rqa* and *nxi.cbuf_avl* together give the RQA an early indication of a read data return transfer on the NXP. The RQA uses this information so that it can schedule a Read Data Queue read access for the NXP port at the earliest possible moment.

2.2.2.4 nxi to wqa

Like the RQA logic, the Write Queue Arbitration logic tracks the fullness of the NXP's write data registers. The WQA logic also schedules WDQ write accesses for the NXP port.

Table 2-18 nxi signals to wqa

nxi.val_wr_data	Valid NXP write data
nxi.wrt_xfr_berr	Write transfer bus error
nxi.wrt_xfr_perr*	Write transfer parity error
nia.load_wd1	Load Write Data register 1
nxi.arisa*	Active request is still around
nxi.xfer_q	Transfer state, qualified
nxi.nxp_intr_rdy	NXP interrupt ready

The WQA uses the valid write data signal, *nxi.val_wrt_data*, to schedule a write access to the NXP's WDQ. The write transfer bus error signal, *nxi.xfr_berr*, and the write transfer parity error signal, *nxi.wrt_xfr_perr**, to set the error states for the NXP write data registers. The load signal, *nxi.load_wd1*, tells the WQA which write data register to mark with full or error status. *nxi.arisa** and *nxi.xfer_q* are used to help qualify the bus error and parity error signals. The interrupt ready signal, *nxi.nxp_intr_rdy*, informs the WQA to schedule an interrupt write access to the WDQ.

2.2.2.5 nxi to xbi

The NXI block contains the NXP interrupt interface and therefore provides an internal interface to the Crossbar Interface which sequences interrupts to the NCU.

Table 2-19 nxi signals to xbi

nxi.intvec<7..0>	NXP interrupt vector bits 7..0
nxi.nxp_intr_busy	NXP interrupt interface busy
nxi.nxp_intr_rdy	NXP interrupt ready

The NXP interrupt vector, *nxi.intvec<7..0>*, is sent to the XBI logic where it is multiplexed with the Pbus interrupt vector before being written into the WDQ. The XBI logic selects the NXP or Pbus interrupt vector for processing to the NCU. The interrupt busy signal, *nxi.nxp_intr_busy*, tell the XBI logic that the NXP interrupt interface is currently busy and can not accept another interrupt at the moment. The XBI logic must wait for the busy signal to be false before it can transfer an interrupt to the NXI. The *nxi.nxp_intr_rdy* signal informs the XBI that

an interrupt has been received by the NXI and to begin processing the interrupt to the NCU.

2.2.2.6 nxi to clk

The NXI block sources two signals to the Clock generation and Scan Control block: *nxi.hard_error* and *nxi.soft_error*. The hard error signal is used to inform the CLK block and the rest of the system that the NIA has detected a catastrophic error. The hard error signal is set if an XIOP removes its NXP interrupt bus request during an interrupt bus cycle. This could result in an erroneous interrupt being sent to through the system.

The soft error is set when the NXI detects a soft error indication over the NXP bus interface. Soft errors include illegal header, header parity error, write data parity error, write transfer PCM violation, read transfer PCM violation, or an XIOP hard error. The soft error indication is forwarded by the CLK block to the SPU. The SPU can then initiate a soft error log scan of the NIA.

2.2.3 Pbus Interrupt Interface - pi

The Pbus Interrupt Interface logic controls the interrupt bus for the four Pbus channels. The PI logic sequences through Pbus interrupt bus cycles and delivers interrupts from a CCU to the Crossbar Interface logic on the NIA.

2.2.3.1 pi to xbi

The PI logic informs the Crossbar Interface logic when an interrupt has been received from a CCU. The XBI logic controls the flow of the interrupt transfer through the NIA and on to the NCU.

Table 2-20 pi signals to xbi

pi.pb_busy	Pbus interrupt interface is busy
pi.pb_intr_rdy	Pbus interrupt ready
pi.intvec<7..0>	Pbus interface interrupt vector

The busy signal, *pi.pb_busy*, informs the XBI logic that the PI logic is busy transferring an interrupt to the CCUs and can not accept another interrupt vector from the XBI. When this occurs, the XBI must wait until the busy signal goes away before the XBI can transfer the next interrupt to the PI. The busy signal goes false once the PI state machine returns to the idle state.

The interrupt ready signal, *pi.pb_intr_rdy*, is asserted when the PI logic receives an interrupt from a CCU. It's used to inform the XBI logic to begin processing the interrupt on to the NCU. The *pi.pb_intvec<7..0>* bus is used to transfer the Pbus interrupt vector to the Write Data Queue. The XBI logic selects either the Pbus or the NXP interrupt vector to process to the NCU.

2.2.3.2 pi to wqa, clk

The PI logic sources the interrupt ready signal, *pi.pb_intr_rdy*, to the Write Queue Arbitration logic. The WQA uses the ready signal to prepare for writing the interrupt vector into the Write Data Queue. The PI also sources a hard error signal to the Clock generation block. The *pi.hard_error* signal is asserted if a CCU removes its interrupt bus request prior to the completion of the interrupt bus cycle. This could result in an erroneous interrupt being sent to through the system.

2.2.4 Channel Data Slice arrays - cds_arrays

The CDS arrays implement the data path for the Pbus and NXP channels. The CDS arrays provide buffering for read data, write data and headers. The arrays also provide multiplexing for headers and write data.

2.2.4.1 cds_arrays to pbiX & nxi

The CDS_ARRAYS block sources read return data to the Pbus Interface blocks. The PBIx blocks latch and translate (ecl to ttl) the data for the Pbus.

Table 2-21 cds_arrays signals to pbiX & nxi

cds.pX_rd_data<63..0>	Pbus X Read Data bits 63..0 (pbiX)
cds.pX_rd_par<7..0>	Pbus X Read Parity bits 7..0 (pbiX)
cdsY.pbX_perr	Pbus X parity error from CDS array Y (pbiX)
cdsY.nxp_perr	NXP bus parity error from CDS array Y (nxi)

Each CDS arrays provides eight data plus 1 parity bit of the each Pbus data path. The bus *cds.pX_rd_data<63..0>* provides read return data and is supplied by a combination of all eight CDS arrays (eight bits per array). The read data parity bits, *cds.px_par<7..0>* are also supplied by the CDS arrays (one bit per array). Each Pbus interface also receives a parity error signal, *cdsY.pbX_perr*, from each of the CDS arrays (array "Y"). The parity error indications are combined and qualified by the individual Pbus Interface blocks.

The CDS arrays drive the bidirectional NXP bus directly. However, the NXI logic receives a parity error indication, *cdsY.nxp_perr*, from each CDS array. The parity error indications are combined and qualified by the NXI block.

2.2.4.2 cds_arrays to wqa & pa

The CDS arrays provide write data to the Write Queue Arbitration logic and header data to the Port Arbitration logic

Table 2-22 cds_arrays signals to wqa & pa

cds.wrt_data<63..0>	Write data bits 63..0 (wqa)
cds.wrt_par<7..0>	Write data parity bits 7..0 (wqa)
cds.header<63..0>	Header data bits 63..0 (pa)
cds.header_par<7..0>	Header data parity bits 7..0 (pa)
cds[1,2,6].prop	Carry propagate from CDS arrays 1,2 & 6 (pa)
cds[0,1,2,6].gen*	Carry generate from CDS arrays 0,1,2 &6 (pa)

Write data from one of the Pbus or NXP write data buffers is selected with in the CDS arrays and sourced to the Write Queue Arbitration logic. The *cds.wrt_data<63..0>* bus provides the write data. The parity bits *cds.wrt_par<7..0>* are selected with the write data.

The CDS arrays also provide header data multiplexing. The Port Arbitration logic selects the desired I/O channel header. The *cds.header<63..0>* bus sources the header data to the PA logic where it is staged and a partial byte count is calculated. The CDS arrays also provide the header parity bits, *cds.header_par<7..0>* to the PA logic to verify parity.

Select CDS arrays source carry propagate signals, *cds[1,2,6].prop*, and carry generate signals, *cds[0,1,2,6].gen**, to the PA logic. The PA block contains the carry look-ahead logic used during the header update process in the CDS arrays. The CDS arrays update the address portion and the byte count portion of a header after it has been selected by the PA logic. The carry look-ahead logic helps speed up this update process.

2.2.5 Read Data Queue - rdq

The Read Data Queue provides read data buffering between system memory and the I/O channels. Read data from system memory is queued into the RDQ at the system clock rate and is de-queued at the Pbus or NXP channel clock rate.

2.2.5.1 rdq to cds_arrays

Table 2-23 rdq signals to cds_arrays

rdq.data_out<63..0>	Read Data Queue data out bits 63..0
rdq.par_out<7..0>	Read Data Queue data parity bits 7..0

Read data is delivered to the CDS arrays on the *rdq.data_out<63..0>* bus. Read data is buffered by the CDS arrays in the appropriate I/O channel staging register. The CDS arrays also check read data parity. The RDQ supplies parity bits, *rdq.par_out<7..0>*, to the CDS arrays for this purpose.

2.2.5.2 rdq to rqa

Table 2-24 rdq signals to rqa

rdq.read_error	RDQ read error flag
rdq.flag_par_out	RDQ read error flag parity
rdq.wrt_pe	RDQ write parity error

The read error flag, *rdq.read_error*, is used by the Read Queue Arbitration logic to track read PCM violations. The RQA logic marks the appropriate I/O channel read data staging register as having an error. When this error advances to the I/O channel output stage, the Pbus or NXP state machine sources a bus error and terminates the transfer. The error flag is parity protected by the *rdq.flag_par_out* signal. Error flag parity (odd) is checked in the RQA block.

Data is parity checked when it is written into the RDQ. Each self-timed RAM outputs a parity error indication during writes. The error indications are combined in the RDQ block and sourced to the RQA logic as *rdq.wrt_pe*. A write parity error eventually results in a hard error.

2.2.6 Write Data Queue - wdq

The Write Data Queue provides write data buffering between the I/O channels and system memory. Write data is queued into the WDQ at the Pbus and NXP clock rates and is de-queued at the system clock rate.

2.2.6.1 wdq to xds_arrays & xbi

Write data from the WDQ is captured in the XDS arrays block. The XDS arrays source the write data to the crossbar.

Table 2-25 wdq signals to xds_arrays & xbi

wdq.data_out<63..0>	Write Data Queue data out bits 63..0 (xds)
wdq.par_out<7..0>	Write Data Queue data parity bits 7..0 (xds)
wdq.wrt_abort	Write Data Queue write abort flag (xbi)

The WDQ sources write data, *wdq.data_out<63..0>* and parity, *wdq.par_out<7..0>*, to the XDS arrays. The XDS arrays check write data parity as data is sent to the crossbar. The WDQ sources a general write abort signal, *wdq.wrt_abort*, to the Crossbar Interface logic. The XBI logic uses the write abort flag to terminate the current write data transfer and to clear the current header from the XBI pipeline.

2.2.6.2 wdq to pa

Table 2-26 wdq signals to pa

wdq.nxp_flush_flag	NXP channel flush flag
wdq.pX_flush_flag	Pbus X channel flush flag
wdq.wrt_abort	WDQ write abort flag
wdq.fflag_par	Flush flag parity bit

The WDQ sources several write abort flags. The general purpose flag, *wdq.wrt_abort*, is set when any of the individual flush flags are set. The individual flush flags, *wdq.nxp_flush_flag* and *wdq.pX.flush_flag*, are set when a write transfer is aborted at the NXP or Pbus X interface. They are used by the Port Arbitration logic to clear individual I/O channel "write complete" states. The PA logic also checks the WDQ flush flag parity using the *wdq.fflag_par* bit. Expected parity is odd.

2.2.7 Crossbar Data Slice Arrays - xds_arrays

The Crossbar Data Slice arrays drive the NIA's crossbar interface signals. On board, the XDS arrays interface with the Crossbar Interface control logic and the Read Queue Arbitration logic. The XDS arrays also contain the Read and Write data queue pointers and fill levels.

2.2.7.1 xds_arrays to rqa

The XDS arrays provide read data "tag" information and RDQ fill level status to the Read Queue Arbitration logic.

Table 2-27 xds_arrays signals to rqa

TAGS	
xds.rdata_id_e<2..0>	Even Read Data ID bits 2..0
xds.ccu_id_e	Even side CCU ID
xds.toc_id_e	Even side TOC ID
xds.pcm_flag_e	Even side PCM flag
xds.rdata_id_o<2..0>	Odd Read Data ID bits 2..0
xds.ccu_id_o	Odd side CCU ID
xds.toc_id_o	Odd side TOC ID
xds.pcm_flag_o	Odd side PCM flag
FILL LEVELS AND POINTERS	
xds.nxp_rdqe_rdy	Even side NXP RDQ ready

Table 2-27 xds_arrays signals to rqa continued

xds.pX_rdqe_rdy	Even side Pbus X RDQ even
xds.rdqe_wrt_addr<7..0>	Even side RDQ write address pointer
xds.nxp_rdqo_rdy	Odd side NXP RDQ ready
xds.pX_rdqo_rdy	Odd side Pbus X RDQ even
xds.rdqo_wrt_addr<7..0>	Odd side RDQ write address pointer

The XDS arrays contain FIFOs which track read data return transfers from memory. There are two read data FIFOs: an even side and an odd side. Both FIFOs contain the same tags for their respective sides. These tags are described in the paragraph below.

Even side read data returns are identified by the channel ID bits, *xds.rdata_id_e<2..0>*. The read data channel ID tells the RQA logic which I/O channel the read data is destine for. This information is used to select the appropriate RDQ address. The XDS arrays also provide two additional even read data tags: *xds.toc_id_e* and *xds.ccu_id_e*. The TOC ID is set for Time of Century (TOC) data returns. This tells the RQA to write the TOC data in one of the reserved locations in the RDQ. The exact reserved location is a function of the channel ID and the CCU ID. The CCU ID identifies the CCU (0 or 1) as the requestor for the TOC data.

The odd side tags, *xds.rdata_id_o*, *xds.ccu_id_o* and *xds.toc_id_o*, are sourced by the odd side FIFOs in the XDS arrays block. They have the same function as the even side tags described above, except they are used in conjunction with the odd side data and RDQ.

The XDS arrays also provide fill level monitors and address pointers to the RQA logic. The even side ready signals, *xds.nxp_rdqe_rdy* and *xds.pX_rdqe_rdy*, are set when the respective even side NXP and Pbus X RDQs have one or more read data entries. An I/O channel's RDQ is read when both the even and odd side ready signals are set. The ready signals tell the RQA logic to schedule a read access for the respective I/O channel. The write address, *xds.rdqe_wrt_addr<7..0>*, is used to form the lower eight bits of the even RDQ write address. This address is selected in the XDS arrays from one of the five even side I/O channel RDQ head pointers.

The odd side monitors and RDQ write address signals, *xds.nxp_rdqo_rdy*, *xds.pX_rdqo_rdy* and *xds.rdqo_wrt_addr<7..0>*, provide the same functionality to the odd RDQ as the even side monitor and write address described above.

2.2.7.2 xds_arrays to xbi

The Crossbar Interface logic controls the header transfer and data flow through the XDS arrays. The XDS arrays supply FIFO fill level and error status to the XBI logic for control purposes.

Table 2-28 xds_arrays signals to xbi

xds.fifo_fl_e<4..0>	Even side FIFO fill level bits 4..0
xds.pcm_flag_e	Even side PCM error flag
xds.fifo_fl_o<4..0>	Odd side FIFO fill level bits 4..0
xds.pcm_flag_o	Odd side PCM error flag
xds.pcm_error1	PCM error detected

The FIFO fill levels, *xds.fifo_fl_e<4..0>* and *xds.fifo_fl_o<4..0>*, track the fullness of the even and odd FIFOs respectively. The FIFOs are 24 locations deep. If the FIFOs become full, the XBI logic will hold off making transfers until the fill level drops. The FIFO fill level will decrease when read data returns from memory.

The PCM error flags, *xds.pcm_flag_e* and *xds.pcm_flag_o* are used by the XBI logic to force a FIFO read. The XBI logic controls the reading and writing to the read return FIFOs. When a PCM error flag reaches the top of a FIFO, the XBI logic forces another FIFO read to pop the error flag off the top. The error flag is written into the RDQ to be later used by the appropriate I/O channel interface.

The PCM error indication, *xds.pcm_error1*, is different from the error flags described in the previous paragraph. The PCM error indication is asserted at the time a read PCM error is detected. It tells the XBI logic to abort the current transfer and to flush the header from the transfer stage. This PCM error indication is also written into the even and odd side return FIFOs. After all of the previously requested read data has been returned, the error "indication" reaches the FIFO output and is then called the PCM error "flag" mentioned above.

2.2.7.3 xds_arrays to pa

The XDS arrays provide several Read and Write Data Queue fill level monitors to the Port Arbitration logic.

Table 2-29 xds_arrays signals to pa

xds.nxp_rdq_full	NXP channel RDQ full indication
xds.pX_rdq_full	Pbus channel X RDQ full indication
xds.nxp_wdq_rdy	NXP channel WDQ ready indication
xds.pX_wdq_rdy	Pbus channel X WDQ ready indication
xds.pcm_error0	PCM error detected

The RDQ full indications, *xds.nxp_rdq_full* and *xds.pX_rdq_full*, are used by the PA logic to suspend selection of the NXP and Pbus X channels respectively. If an I/O channel's RDQ becomes full, the PA logic must not select that channel for additional memory accesses or else risk over flowing the RDQ. The WDQ ready indications, *xds.nxp_wdq_rdy* and *xds.pX_wdq_rdy*, have the opposite effect on the PA logic. The ready indications tell the PA logic to consider that I/O channel in the next selection of headers. The ready indication means that the WDQ contains "enough" write data entries to warrant a memory access. "Enough" transfers is a scan programmable value, but generally means 17 or more for a Pbus channel and 33 or more for the NXP channel

The PCM error indication, *xds.pcm_error0*, is used to clear the read transfer state for the appropriate I/O channel. It's a different version of the same signal defined in the previous section.

2.2.7.4 xds_arrays to wqa & wdq

The XDS arrays contain the I/O channel head and tail pointers for the Write Data Queue. The Write Queue Arbitration logic selects the desired head pointer for writing data into the WDQ. The address bus, *xds.wdq_wrt_addr<7..0>* is the selected head pointer. The write address is staged in the WQA logic prior to WDQ write access.

Reading the WDQ is under control from the Crossbar Interface logic. The Port Arbitration logic provides the "active" I/O channel ID to the XDS arrays. The active ID selects the desired WDQ read address, *xds.wdq_rd_addr<7..0>*, which is sent directly to the WDQ.

2.2.7.5 xds_arrays to pbiX & nxi

The XDS arrays provide WDQ full indications, *xds.pX_wdq_full* and *xds.nxp_wdq_full* to the Pbus X and NXP channel interfaces respectively. The WDQ full indications are used by the I/O channel interfaces to suspend write data transfers from CCUs. Suspending CCU write data transfers prevents the WDQ from over flowing.

2.2.7.6 xds_arrays to clk

The XDS arrays provide parity error indications to the Clock and Scan control logic. The XDS arrays parity check header data from the PA logic and write data from the WDQ. The parity error indications, *xds00.par_err*, *xds01.par_err*, *xds10.par_err* and *xds11.par_err* result in a hard error on the NIA.

2.2.8 Crossbar Interface - xbi

The Crossbar interface logic controls the XDS arrays' data path and the flow of transfers to the crossbar. The XBI logic also contains the interrupt logic and the trap vector bus interface.

2.2.8.1 xbi to xds_arrays

The XBI logic provides clock enables, FIFO control and status to the XDS arrays for controlling the transfer stages.

Table 2-30 xbi signals to xds_arrays

xbi.inc_pX_wdq_tp	Increment Pbus X WDQ tail pointer
xbi.inc_nxp_wdq_tp	Increment NXP WDQ tail pointer
xbi.fifo_read_e	Even side FIFO read control
xbi.fifo_read_o	Odd side FIFO read control
xbi.fifo_wrt[0,1]	FIFO write controls (both even & odd)
xbi.l0_clk_ena[0,2]	Level 0 stage clock enables
xbi.l0_full	Level 0 stage full indication
xbi.l1_clk_ena[0,1]	Level 1 stage clock enables
xbi.l1_count[0,1]	Level 1 address counter enable
xbi.l1_full	Level 1 stage full indication
xbi.l2_clk_ena	Level 2 stage clock enable
xbi.last_xfr	Last transfer indication
xbi.ovr_flo_full	Over flow staging register full

The XBI logic controls the flow of transfers to the Xbar and in turn controls the read access of the Write Data Queue. The appropriate WDQ tail pointer increment control, *xbi.inc_pX_wdq_tp* or *xbi.inc_nxp_wdq_tp*, is set each time write data is read from the Pbus X or NXP WDQ respectively.

The read FIFO controls *xbi.read_fifo_e* and *xbi.read_fifo_o* are set each time even and odd read data returns from the Xbar to the NIA or when a PCM error flag reaches

the top of the FIFO. The FIFO read control forces the read return FIFOs to pop the top entry from the FIFO queue. The XBI logic also controls writes to the FIFO with the *xbi.fifo_wrt0* and *xbi.fifo_wrt1* control signals. Both even and odd side FIFOs are written at the same time. The FIFO write controls are asserted for each memory read transfer to the Xbar. The FIFOs track all outstanding read requests and identify read data as it returns from memory.

The XBI logic provides clock enables for the level 0 stage (*xbi.l0_clk_ena0, xbi.l0_clk_ena2*), the level 1 stage (*xbi.l1_clk_ena0, xbi.l1_clk_ena1*) and the level 2 stage (*xbi.l2_clk_ena*). When the clock enables are asserted, the stage will load a new header. The XBI logic also provides level0 full, *xbi.l0_full,* and level1 full, *xbi.l1_full,* indications to the XDS arrays. The level 0 full indication is used to qualify parity checking on header data input to the l0 stage. The level 1 stage full indication is used to qualify PCM checking of read transfers. The level 1 stage also has an address counter controls, *xbi.l1_count0* and *xbi.l1_count1*, which tell the XDS array when to increment it's level 1 address. The level 1 address field is incremented by eight (long word offset) after each successful memory transfer.

The XDS arrays use the last transfer control, *xbi.last_xfr,* for special treatment of the cycle and write zone outputs to the Xbar. The last transfer control is set during the last transfer of the header in the level 1 stage. The over flow full indication, *xbi.ovr_flo_full,* is used to hold the write data over flow register contents. The over flow indication also forces the over flow register as the next input to the write data output stage (level 2). The over flow full indication gets set if memory becomes busy and the NIA has prefetched write data from the WDQ before the busy indication could stop the read access.

2.2.8.2 xbi to pa

The XBI logic provides the following control signals to the Port Arbitration logic.

Table 2-31 xbi signals to pa

xbi.l0_clk_ena[1,2]	Level 0 clock enable
xbi.l0_clk_ena1*	Level 0 clock enable, active low
xbi.l1_clk_ena*	Level 1 clock enable
xbi.dec_fetch_cnt	Decrement fetch counter
xbi.read_wrt_data	Read request for write data in WDQ
xbi.xdr_pnd_e	Even crossbar data return pending
xbi.xdr_pnd_o	Odd crossbar data return pending
xbi.pa_int_req	Interrupt transfer request to PA logic

The level 0 and level 1 clock enables, *xbi.l0_clk_ena1, xbi.l0_clk2,xbi.l0_clk_ena** and *xbil1_clk_ena**, are used in the PA logic to stage I/O channel valid flags in the level 0 and 1 stages. The level 0 clock enable is also used to load the fetch counter. The PA logic uses the decrement fetch counter signal, *xbi.dec_fetch_cntr,* to decrement the fetch counter. The decrement control is asserted when as transfers are sent to the crossbar. The fetch counter tracks the number of transfers left in the level 0 stage. It also represents the number of WDQ read accesses or "fetches" that are required to complete the header in the level 0 stage. The read write data signal, *xbi.read_wrt_data,* is asserted when the XBI logic wants to read the next WDQ entry for the header in the level 0 stage. The PA logic stages this signal and uses it to qualify parity checking logic for the WDQ flush flags.

The even and odd side "crossbar data return pending" signals, *xbi.xdr_pend_e* and *xbi.xdr_pend_o*, are used to hold the read flush states in the Port Arbitration logic. The read flush states are set when a CCU aborts a read transfer and are cleared when the data return pending signals go false. The data return pending signals are asserted as long as the NIA expects read data from memory. When all outstanding read data has been returned to the NIA, the data return pending signals are false.

The XBI logic requests an interrupt transfer from the PA logic when an interrupt has been received from either the Pbus or NXP interrupt bus. The XBI logic requests this action by asserting the *xbi.pa_int_req* signal to the PA logic. The PA logic services this request by selecting the interrupt header from the CDS arrays.

2.2.8.3 xbi to pi and nxi

The XBI logic interfaces to the trap vector bus from the crossbar. The XBI logic screens trap vector bus transfers for interrupts. When an interrupt transfer is received, the XBI logic forwards the interrupt vector to the Pbus Interrupt interface and the NXP Interface.

Table 2-32 xbi signals to pi & nxi

xbi.ncu_intr_rdy_p	NCU interrupt ready to PI (pi)
xbi.ncu_intr_rdy_x	NCU interrupt ready to NXP (nxi)
xbi.ncu_intvec<7..0>	NCU interrupt vector bits 7..0 (pi & nxi)
xbi.pb_intr_ack	Pbus interrupt acknowledge (pi)
xbi.nxp_intr_ack	NXP interrupt acknowledge (nxi)

The interrupt ready signals, *xbi.ncu_intr_rdy_p* and *xbi.ncu_intr_rdy_x*, are asserted when the XBI logic receives an interrupt from the NCU. The Pbus and NXP interrupt interfaces each get their own ready signal due to their different clock rates. The XBI logic provides the interrupt vector on the *xbi.ncu_intvec<7..0>* bus. Once the Pbus and NXP interrupt interfaces get the NCU interrupt vector, they initiate interrupt bus cycles to deliver the interrupt to the CCUs.

The XBI also sequences interrupt transfers from the NIA to the NCU. When an interrupt is received by the Pbus or NXP interface, the XBI logic is notified and initiates an interrupt transfer sequence to the NCU. When the Pbus or NXP interrupt vector is written into the WDQ, the XBI logic acknowledges this fact by asserting *xbi.pb_intr_ack* and *xbi.nxp_intr_ack* to the PI or NXI logic respectively. This acknowledge allows the interrupt interface to begin another interrupt bus cycle if needed.

2.2.8.4 xbi to wdq and wqa

The XBI interfaces to the Write Data Queue and Write Queue Arbitration logic for the purposes of reading data from the WDQ and writing interrupt vectors into the WDQ.

Table 2-33 xbi signals to wdq and wqa

xbi.read_wrt_data	Read request for write data in WDQ (wdq)
xbi.snd_intvec<7..0>	Send interrupt vector bus bits 7..0 (wqa)
xbi.snd_state0	Send interrupt state machine is in state 0 (wqa)
xbi.wdq_ena	WDQ write enable (wqa)

The XBI sources *xbi.read_wrt_data* when it wants to read data from the WDQ. The WDQ logic uses this read request to enable the flush flag outputs from the WDQ. The flush flag outputs are only enabled when read data from the WDQ is requested by the XBI logic. Otherwise the XBI logic could receive a write abort (flush flag) prior to when it should have been received.

2.2.9 Read Queue Arbitration - rqa

The Read Queue Arbitration logic controls the read access to the Read Data Queue. The RQA logic contains the arbitration logic, RDQ tail pointers, write data staging, write address staging, full flags, and error flags for Pbus and NXP channel read data buffers.

2.2.9.1 rqa to rdq

The RQA logic provides write data and addresses to the even and odd side RDQs.

Table 2-34 rqa signals to rdq

rqa.rdqe_wadr<10..0>	Even RDQ write address bits 10..0
rqa.rdqe_wdata<31..0>	Even RDQ write data bits 31..0
rqa.rdqe_wpar<3..0>	Even RDQ write parity bits 3..0
rqa.rdqe_wvalid	Even RDQ write valid
rqa.rdq_rd_err	RDQ read error flag
rqa.rdq_rd_err_par	RDQ read error flag parity bit
rqa.rdqo_wadr<10..0>	Odd RDQ write address bits 10..0
rqa.rdqo_wdata<31..0>	Odd RDQ write data bits 31..0
rqa.rdqo_wpar<3..0>	Odd RDQ write parity bits 3..0
rqa.rdqo_wvalid	Odd RDQ write valid

The even and odd side RDQs are written to independently as read data returns from the crossbar. The RQA logic captures read data from the crossbar, parity checks the data and then stages it to the WDQ. Write address information is provided by the *rqa.rdqe_wadr<10..0>* and *rqa.rdqo_wadr<10..0>* interface buses to the even and odd RDQs respectively. Write data is provided by *rqa.rdqe_wdata<31..0>* and *rqa.rdqo_wdata<31..0>*. Write data parity is supplied by *rqa.rdqe_wpar<3..0>* and *rqa.rdqo_par<3..0>*. Parity is odd. An error flag, *rqa.rdq_rd_err*, and parity bit, *rqa.rdq_rd_err_par*, are also written into the RDQ. The error flag and parity bit are written with the even side RDQ. The error flag is set when a read PCM error reaches the top of the read data return FIFO in the XDS arrays. Again, error flag parity is odd.

2.2.9.2 rqa to xds_arrays

The RQA logic controls the reading and writing of the RDQ. The XDS arrays contain write addresses (head pointers) for the even and odd side RDQs. The XDS arrays also contain RDQ fill level counters used to track the fullness of each I/O channel's RDQ.

Table 2-35 rqa signals to xds_arrays and xbi

rqa.inc_pX_e_hp	Increment Pbus X RDQ even head pointer (xds)
rqa.inc_pX_o_hp	Increment Pbus X RDQ odd head pointer (xds)
rqa.inc_pX_tp	Increment Pbus X RDQ tail pointer (xds)

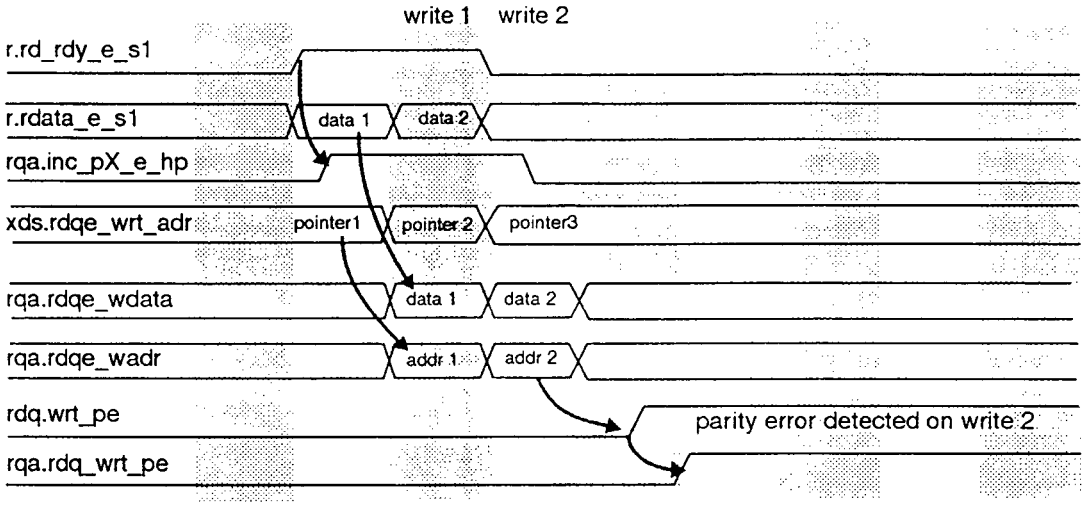
Table 2-35 rqa signals to xds_arrays and xbi continued

rqa.inc_nxp_e_hp	Increment NXP RDQ even head pointer (xds)
rqa.inc_nxp_o_hp	Increment NXP RDQ odd head pointer (xds)
rqa.inc_nxp_tp	Increment NXP RDQ tail pointer (xds)
rqa.rdq_wrt_pe	RDQ write parity error (xds)
rqa.rd_rdy_e_s1	Read Ready Even side stage 1 (xbi)
rqa.rd_rdy_o_s1	Read Ready Odd side stage 1 (xbi)

The RDQ write addresses (head pointers) are controlled by the RQA. The RQA asserts the increment controls *rqa.inc_pX_e_hp* and *rqa.inc_nxp_e_hp* to increment the even side Pbus X or NXP channel tail pointer respectively. The tail pointer is incremented each time data is written into the RDQ. The odd side tail pointers are incremented with the *rqa.inc_pX_o_hp* and *rqa.inc_nxp_o_hp* control signals. The tail pointer increment controls *rqa.inc_pX_tp* and *rqa.inc_nxp_tp* are used to help control the RDQ fill level counters. The tail pointer increments are used to decrement the RDQ fill levels. The head pointer increment controls are used to increment the fill level counters as well as the head pointer addresses.

Notice that the head pointer increment controls have two flavors: even and odd. Yet the tail pointer controls do not. The reason for this is the RDQ is written 32 bits at a time but is read 64 bits at a time. Thus an even and odd write address (head pointer) are needed but only one read address (tail pointer) is required.

Figure 2-17 rdq write sequence



The RQA logic provides a write parity error indication, *rqa.rdq_wrt_pe*, to the XDS arrays. This signal is used to freeze the RDQ head pointer shadow registers which track the last three head pointer addresses. When a write parity error is detected, the shadow registers will be frozen to preserve the address from which the write parity error was detected. Parity is checked by the RDQ self-timed RAMs. The RDQ logic combines the individual RAM parity error signals and sends a combined write parity error indication to the RQA (called *rdq.wrt_pe*). The RQA stages this parity error signal to provide *rqa.rdq_wrt_pe* to the XDS arrays. See Figure 2-2.

The read ready stage 1 signals, *rqa.rd_rdy_e_s1* and *rqa.rd_rdy_o_s1*, are the registered read ready interface signals from the crossbar read return interface.

These signals are set when the NIA receives read return data from the even and odd side crossbar respectively. The RQA logic uses these signals internally to prepare for an RDQ write cycle. The RQA also sends these signals to the XBI logic where they are used to control the read return FIFOs in the XDS arrays.

2.2.9.3 rqa to pbiX and nxi

Table 2-36 rqa signals to pbiX, nxi and cds_arrays

rqa.pX_rd1_full	Pbus X read data register full (pbiX)
rqa.pX_rd1_error	Pbus X read data error (pbiX)
rqa.nxp_rd0_full	NXP read data register 0 full (pbiX)
rqa.nxp_rd1_full	NXP read data register 1 full (pbiX)
rqa.nxp_rd0_error	NXP read data register 0 error (pbiX)
rqa.nxp_rd1_error	NXP read data register1 error (pbiX)
rqa.pX_rd1_ce[0,1]*	Pbus X read data register clock enables (cds)
rqa.nxp_rd0_ce[0,1]*	NXP read data register 0 clock enables (cds)
rqa.nxp_rd1_ce[0,1]*	NXP read data register1 clock enables (cds)

The RQA logic tracks full and error states for the Pbus X and NXP interfaces. Each Pbus interface receives a full and error state signal, *rqa.pX_rd1_full* and *rqa.pX_rd_error*, respectively. The full state is set when the read data staging register is loaded with data from the RDQ. The read data staging register feeds the read data output register that drives read data onto the Pbus. The error state is set for read PCM violations. The read error flag from the RDQ is used to set the read error state. The Pbus X interface uses the full state to assert the data valid control on the Pbus. The error state is used to assert the bus error on the Pbus and to terminate the read transfer.

The NXI logic receives two full states, *rqa.nxp_rd0_full* and *rqa.nxp_rd1_full*, and two error states, *rqa.nxp_rd0_error* and *rqa.nxp_rd1_error*, from the RQA. Unlike the Pbus interfaces, the NXP interface has two read data registers that can feed the NXI's read data output stage. The RQA tracks the full and error states for both of the read data buffers. The NXI has two read data buffers because of it's high bandwidth and low arbitration priority.

In addition to the full and error states mentioned above, the RQA logic provides clock enables for the Pbus X and NXP read data buffers. These read data registers are located in the CDS arrays. The Pbus X read data registers use the *rqa.pX_rd1_ce0** and *rqa.pX_rd1_ce1**, clock enables. Version "0" goes to arrays 0-3 and version "1" goes to arrays 4-7. The NXP read data registers are enabled by *rqa.nxp_rd0_ce0**, *rqa.nxp_rd0_ce1** and *rqa.nxp_rd1_ce0**, *rqa.nxp_rd1_ce1**. All of the read data registers are free running until they are loaded with valid read data from the RDQ. At that point the clock enable goes false until the data is forwarded into the read data output register.

2.2.9.4 rqa to clk

The RQA logic checks parity on the read data received from the even and odd side crossbar interfaces. The error signals, *rqa.rd_par_err_e* and *rqa.rd_par_err_o* are asserted if bad parity is detected on the even and odd sides respectively. The RQA logic also checks parity for the RDQ error flag and asserts the *rqa.rdq_flag_pe* signal if bad parity is detected. Finally, the RQA stages the RDQ write parity error from the RDQ logic. The RDQ logic sources *rdq.wrt_pe* to the RQA where it is registered

and sent to the CLK logic as *rqa.rdq_wrt_pe*. In all cases, expected parity is odd. The error signals sent to the CLK logic result in a hard error from the NIA.

2.2.10 Write Queue Arbitration - wqa

The Write Queue Arbitration logic controls the write access to the Write Data Queue. The WQA logic contains the arbitration logic, write data, flush flag data and address staging.

2.2.10.1 wqa to cds_arrays

The arbitration function is the mainstay of the WQA logic. The arbitration function generates write data selects to the CDS arrays. Other WQA interface signals to the CDS arrays include a write data select for the two NXP write data registers plus clock enables for each NXP write data register.

Table 2-37 wqa signals to cds_arrays

wqa.wr_data_sel[a,b]<2..0>	Write data selects (versions a & b)
wqa.nxp_wd1_sel[a,b]	NXP write data register 1 select (vers a & b)
wqa.nxp_wd0_ce[0,1]*	NXP write data register 0 clock enables
wqa.nxp_wd1_ce[0,1]*	NXP write data register 1 clock enables

Each of the above signals have two versions. Some are labeled "a" and "b" and others are labeled "0" and "1". The two versions of the same signal are logically the same but physically separated for loading reasons. Versions "a" and "0" drive CDS arrays 0-3, versions "b" and "1" drive CDS arrays 4-7.

The write data selects, *wqa.wr_data_sela*<2..0> and *wqa.wr_data_selb*<2..0>, are used by the CDS arrays to select the desired I/O channel's write data. The WQA can also select an "interrupt" data pattern that's used when writing interrupt vectors into the WDQ. The interrupt vector itself is multiplexed into the least significant eight data bits of write data output from the CDS arrays. This multiplexing function is performed in the WQA logic.

Figure 2-18 WQA write data selects to the CDS arrays

select	write data
0	Pbus 0
1	Pbus 1
2	Pbus 2
3	Pbus 3
4	NXP
5	not used
6	not used
7	interrupt

The NXP write data selects, *wqa.nxp_wd1_sela* and *wqa.nxp_wd1_selb*, are used to select between NXP write data register 0 and NXP write data register 1. When asserted, NXP write data register 1 is selected. The NXP interface has two write data registers, whereas each Pbus interface only has one write data register.The

NXP needs two write data registers because of is high transfer rate and because is has the lowest priority to the WDQ.

The WQA also provides NXP write data clock enables to the CDS arrays. The write data clock enables *wqa.nxp_wd0_ce0** and *wqa.nxp_wd0_ce1** control NXP write data register 0. The signals *wqa.nxp_wd1_ce0** and *wqa.nxp_wd1_ce1** control NXP write data register 1. The NXP write data clock enables are necessary because the NXP interface may need to hold its write data for several cycles before it is written into the WDQ. The clock enables provide this holding function. In contrast, the Pbus interfaces do not need write data clock enables because the WQA will provide each Pbus interface a write access to the WDQ each Pbus cycle.

2.2.10.2 wqa to wdq

The WQA provides write data, flush flags and write address to the Write Data Queue.

Table 2-38 wqa signals to wdq

wqa.wdq_wrt_addr<10..0>	WDQ write address bits 10..0
wqa.wdq_wrt_data<63..0>	WDQ write data bits 63..0
wqa.wdq_wrt_par<7..0>	WDQ write parity bits 7..0
wqa.pX_flush flag	Pbus X WDQ flush flag
wqa.nxp_flush_flag	NXP WDQ flush flag
wqa.flush_flag	"General" flush flag
wqa.fflag_par	Flush flag parity bit
wqa.wdq_wdata_valid	WDQ write data valid
wqa.wdq_wflag_valid	WDQ write flags valid

The WQA logic stages the WDQ write address, *wqa.wdq_wrt_addr<10..0>* and write data, *wqa.wdq_wrt_data<63..0>* to the WDQ logic. The write address and data are selected by the arbitration logic. The write address is selected from the XDS arrays and the write data is selected from the CDS arrays. Write data parity, *wqa.wdq_wrt_par<7..0>* is also selected from the CDS arrays and staged with the write data.

The flush flags, *wqa.pX_flush_flag* and *wqa.nxp_flush_flag*, are sourced to the WDQ. The flush flags are registered write abort states from the respective I/O channel interfaces. The flush flags are set when an I/O channel interface detects an aborted write transfer. The flush flags are stored into the WDQ in place of write data and are used by the Port Arbitration logic to clear the "write complete" state. The "general" flush flag, *wqa.flush_flag*, is the logical OR of all of the I/O channel flush flags. The general flag is used by the Crossbar interface logic to terminate the memory write transfer at the appropriate time. The flush flags are parity protected by the *wqa.fflag_par* bit. Parity is odd.

The WQA also sources data valid control signals to the WDQ. The *wqa.wdq_wdata_valid* control is asserted when the WQA has valid write data for the WDQ. The write data valid control is used to enable a write access to the data RAMs. The write flag valid control, *wqa.wdq_wflag_valid*, is asserted when the WDQ flush flags are written. The flush flag RAM is written for all valid data writes and for write aborts. The data RAMs are written only when valid write data is available.

2.2.10.3 wqa to nxi

The WQA logic the full states of the NXP write data registers. The NXI logic needs this information plus additional states in order to know when to hold off an XIOP from sending write data to the NIA.

Table 2-39 wqa signals to nxi

wqq.hld_wd0_full	Holding term of write data register 0 full
wqq.hld_wd1_full	Holding term of write data register1 full
wqa.nxp_wd_full	NXP write data registers full
wqa.nxp_wd_haz	NXP write data hazard

The signals, *wqa.hld_wd0_full* and *wqa.hld_wd1_full*, are holding terms for the full states of the NXP write data registers. The holding term is true when the write data register is going to remain full during the next clock cycle. The write data registers full signal, *wqa.nxp_wd_full* is the logical AND of the individual full states for registers 0 and 1. The hazard signal, *wqa.nxp_wd_haz*, is asserted when the NXP port will not get a write access for the next two clock cycles or when an interrupt vector is going to be written into the WDQ. All of these signals are used in the NXI logic to create the channel buffer available signal, *ia_xiop.cbuf_avl*. The channel buffer available signal tells the XIOP if the NIA can accept write data on that cycle.

2.2.10.4 wqa to xds_arrays

The XDS arrays contain the WDQ head pointers and fill levels for the I/O channels. The WQA logic sources head pointer selects, head pointer controls and fill level controls to the XDS arrays.

Table 2-40 wqa signals to xds_arrays

wqa.hp_select<2..0>	WDQ head pointer select
wqa.inc_pX_hp	Increment Pbus X WDQ head pointer
wqa.inc_nxp_hp	Increment NXP WDQ head pointer
wqa.wdq_wrt_pe	WDQ write parity error

The WQA logic generates write data selects, described earlier, for selecting the desired I/O channel's write data. The WQA logic also generates WDQ head pointer selects, *wqa.hp_select<2..0>*, for selecting the desired write address. The head pointers are implemented in the XDS arrays were the head pointer selects are used to select the desired address. The WQA also sources head pointer increment controls, *wqa.inc_pX_hp* and *wqa.inc_nxp_hp*, for incrementing the head pointers. The head pointer increment controls are also used to increment the WDQ fill levels which are also implemented in the XDS arrays.

The WDQ RAMs will parity check write data as it is written into the RAMs. If a parity error is detected on a write access, the WDQ logic sources the parity error signal *wdq.wrt_pe* to the WQA logic. The WQA logic stages this signal and sends it to the XDS arrays as *wqa.wdq_wrt_pe*. The XDS arrays use this write parity error signal to freeze the WDQ head pointer shadow registers in order to preserve the address that was used when the parity error was detected.

2.2.10.5 wqa to xbi, pa and clk

The WQA logic has single bit interfaces to the XBI, PA and CLK logic. The WQA

sources *wqa.nxp_prio* to the XBI logic. The signal is asserted when the NXP port has write access priority to the WDQ (clock cycles 3,5,8,11,14, & 17). The XBI logic uses the NXP priority signal to know when an interrupt vector has been written to the WDQ. Interrupts received from the Pbus or NXP interrupt buses are written into the WDQ during NXP priority clock cycles. The XBI logic posts a request to write an interrupt into the WDQ, then waits for the next NXP priority cycle (defined by the assertion of *wqa.nxp_prio*), and then continues with the interrupt transfer process.

The WQA sources *wqa.nxp_wd_empty* to the Port Arbitration logic. The NXP write data empty control signal is asserted when both NXP write data buffers are empty (not full). The PA logic uses this control signal to know when to process an NXP header to the crossbar interface.

The *wqa.wrt_pe* signal is asserted when a parity error is detected on a write to the WDQ. The CLK logic uses this signal to assert the NIA's hard error signal to the NCU. This signal is also sent to the XDS arrays as described in the previous section.

2.2.11 Port Arbitration logic - pa

The Port Arbitration logic is the central control logic for the NIA. It governs I/O channel access to the crossbar interface. The PA logic performs three main functions: arbitrates I/O channel access to memory, generates a partial byte count used to update headers in the CDS arrays, and generates a fetch count used by the XBI logic for fetching WDQ data and completing memory transfers.

2.2.11.1 pa to cds_arrays

The PA logic provides header selects, partial byte count and pipeline control to the CDS arrays.

Table 2-41 pa signals to cds_arrays

pa.header_sel[a,b]<2..0>	Header select bits 2..0 (versions a and b)
pa.partial_bc<8..0>	Partial byte count bits 8..0
pa.update_pX_hdr	Update Pbus X header
pa.update_nxp_hdr	Update NXP header
pa.carryin_cds[1,2,3,7]	Carry in control to CDS arrays 1,2,3 and 7
pa.stage1_ce[0,1]	Stage 1 clock enable (versions 0 and 1)
pa.stage2_ce[0,1]	Stage 2 clock enable (versions 0 and 1)

The header selects, *pa.header_sela*<2..0> and *pa.header_selb*<2..0>,are the outputs from the arbitration logic. The header selects define the next I/O channel header to enter the pipeline to the crossbar interface. The selected header may be one of the I/O channel headers or a predefined header which is scan loaded into the CDS arrays at initialization time. There are three predefined headers: an MBP header for Memory Base Pointer reads, a TOC header for Time of Century reads, and an interrupt header for interrupt transfers to the NCU. The MBP and TOC headers are selected when one of these requests are received by an I/O channel interface. The Pbus address used to access the MBP and TOC is different from the crossbar address used in C3. Therefore, these predefined headers are selected when one of these requests are received. The interrupt header works the same way and is used to send interrupt transfers to the NCU. Signal version "a" goes to CDS arrays 0-3, and version "b" goes to arrays 4-7.

Figure 2-19 PA header selects to the CDS arrays

select	header
0	Pbus 0
1	Pbus 1
2	Pbus 2
3	Pbus 3
4	NXP
5	TOC
6	MBP
7	interrupt

The partial byte count, *pa.partial_bc<8..0>* defines the number of bytes that are going to be written or read for the selected I/O channel interface. The byte count is used by the CDS arrays to "update" the I/O channel's header. The partial byte count is added to the header address and subtracted from the header byte count. This arithmetic function keeps the I/O channel header ready for the next block of transfers to memory.

The addition and subtraction functions in the CDS arrays are aided by carry look-ahead logic in the PA. The CDS arrays output group propagate and group generate signals to the PA. The PA logic uses these signal to create the carry in controls *pa.carryin_cds1*, *pa.carryin_cds2*, *pa.carryin_cds3* and *pa.carryin_cds7*. Only CDS arrays 1,2,3 and 7 require carry in controls. Arrays 0-3 perform the address addition function while arrays 6 & 7 perform the byte count subtraction.

The update header controls, *pa.update_pX_hdr* and *pa.update_nxp_hdr*, are used to load the "updated" header into the desired Pbus X or NXP header register. The PA logic tracks the header update pipeline and issues the update control at the appropriate time.

The stage 1 clock enables, *pa.stage1_ce0* and *pa.stage1_ce1*, qualify stage 1 registers in the CDS arrays. The stage 1 registers hold the CDS header selects. The stage 2 clock enables, *pa.stage2_ce0* and *pa.stage2_ce1*, qualify stage 2 registers in the CDS arrays. The stage 2 registers hold an internal copy of the selected header. The "0" versions go to arrays 0-3 and the "1" versions go the arrays 4-7.

2.2.11.2 pa to xbi

The PA logic provides several control signals to the Crossbar interface control logic. The PA logic monitors and controls the transfer pipeline up to the crossbar interface. The XBI logic controls the transfers once they reach the crossbar interface.

Table 2-42 pa signals to xbi

pa.fetch_cntr<5..0>	Fetch counter bits 5..0
pa.fetch_cntr_co	Fetch counter carry out
pa.fetch_cntr_rem	Fetch counter remainder
pa.active_port<2..0>	Active port ID bits 2..0

Table 2-42 pa signals to xbi continued

pa.hdr_cycle<3..0>	Header transfer cycle bits 3..0
pa.hdr_valid	Header valid
pa.clear_s2	Clear stage 2 header
pa.clear_xs0_[a,b]	Clear crossbar stage 0 header (vers a & b)
pa.clear_xs1	Clear crossbar stage 1 header
pa.hold_reads	Hold read transfers
pa.int_val_s1	Interrupt transfer valid in stage 1

The fetch counter, *pa.fetch_cntr<5..0>*, represents the number of crossbar transfers that are required to service the transfer in the crossbar interface. The counter is loaded when the header advances into the XS0 stage (crossbar stage 0). The carry out flag, *pa.fetch_cntr_co*, and the remainder flag, *pa.fetch_cntr_rem*, each represent one transfer in addition to the 6 bit fetch counter value. Together, these signals are used by the XBI logic to know when the transfer has been completed. The XBI asserts a decrement command to the PA logic for each successful transfer. When the count reaches zero, the transfer is complete.

The active port ID, *pa.active_port<2..0>*, defines the I/O channel for the header in the XS0 stage. This port ID is used by the XBI logic to assert the proper I/O channel's WDQ tail pointer increment control as write data is read from the WDQ. The XBI logic controls the reading of write data from the WDQ.

The header cycle bits, *pa.hdr_cycle<3..0>*, define header transfer type. The transfer type is decoded by the XBI logic and used to control the transfer flow. The header valid signal, *pa.hdr_valid*, is asserted when the PA logic has a valid header transfer for the crossbar interface.

Occasionally, a header must be cleared from the crossbar interface pipeline. The PA logic issues clear signals for the S2 stage, *pa.clear_s2*, the XS0 stage, *pa.clear_xs0_a* and *pa.clear_xs0_b*, and the XS1 stage, *pa.clear_xs1*. A header can be cleared from the pipeline if a read transfer is aborted at the I/O channel interface or if a read PCM error is detected. If a read transfer is aborted by a CCU on any I/O channel interface, the PA logic will assert the hold reads control, *pa.hold_reads*. The hold reads control is used by the XBI logic to hold off read transfers at the crossbar interface. Read transfers are held until all previous reads have returned from memory and the aborting I/O channel's Read Data Queue has been flushed.

The interrupt valid in stage 1 signal, *pa.int_val_s1*, is asserted when an interrupt enters stage 1 of the transfer pipeline. The XBI logic uses this signal to advance its send interrupt state machine to the next state. The XBI logic sequences interrupt transfers through the NIA and this control signal is feedback to the send interrupt state machine.

2.2.11.3 pa to xds_arrays & wdq

The PA logic provides the XDS arrays status signals that describe the header at the crossbar interface. The PA logic also provides WDQ tail pointer selects to the XDS arrays and the Write Data Queue.

Table 2-43 pa signals to xds_arrays and wdq

pa.channel_id<2..0>	Channel ID to XDS array read return FIFOs (xds)
pa.ccu_id)	CCU ID to XDS array read return FIFOs (xds)
pa.toc_id	TOC ID to XDS array read return FIFOs (xds)
pa.active_port<2..0>	Active port bits 2..0 (xds & wdq)

The channel ID bits, *pa.channel_id<2..0>*, define the I/O channel for the header in the XS1 stage. The channel ID is written into the XDS array FIFOs during read operations to identify read data return transfers from memory. The CCU ID, *pa.ccu_id*, and the TOC ID, *pa.toc_id*, are used the same way. The CCU ID identifies the CCU that sourced the header and the TOC ID identifies the read request as TOC data. Both bits are written into the XDS FIFOs. TOC (Time of Century) read data is written into unique locations in the Read Data Queue. Each CCU has its own TOC location in the RDQ. The CCU ID and the TOC ID are used to address these unique locations.

The active port signals, *pa.active_port<2..0>*, define the I/O channel for the header in the XS0 stage. The active port signals are used to select the correct I/O channel tail pointer in the XDS arrays. The WDQ logic also uses the active port signals as the most significant three bits of the WDQ address.

2.2.11.4 pa to pbiX, nxi, & rqa

The PA logic sources several control signals to the PBIx, NXI and RQA logic.

Table 2-44 pa signals to pbiX, nxi and rqa

pa.rdy_4_pX_hdr	Ready for Pbus X header (pbiX)
pa.rdy_r_nxi_hdr	Ready for NXP header (nxi)
pa.pX_rflush	Pbus X read flush state (rqa)
pa.nxp_rflush	Pbus X read flush state (rqa)

The Pbus X and NXP interfaces receive "ready for header" control signals, *pa.rdy_4_pX_hdr* and *pa.rdy_4_nxp_hdr*, respectively from the PA logic. An I/O channel's "ready for header" signal is asserted when that I/O channel's previous header transfer (if any) has been selected and has cleared the pipeline to the crossbar interface. The Pbus X and NXP interfaces use this signal to hold off granting the interface bus to a CCU until the PA logic can accept another header. This prevents an I/O channel from having two different headers in the NIA at the same time.

The read flush states, *pa.pX_rflush* and *pa.nxp_rflush*, are set when a Pbus X or NXP CCU aborts a read transfer. The state remains set until all previously requested read data has returned from memory. The Read Queue Arbitration logic uses the read flush states to prevent RDQ accesses and to clear the RDQ head and tail pointers for that I/O channel.

2.2.11.5 pa to clk

The PA logic sources the WDQ write parity error signal, *pa.wdq_wrt_pe*,to the CLK logic where it is used to assert a hard error from the NIA. This signal is the registered version of the write parity error signal, *wdq.wrt_pe*, from the WDQ logic, which is asserted when a parity error is detected during a write access to the WDQ.

2.2.12 Clocks and Scan control - clk

The Clock Generation and Scan Control logic provides clocks, clock enables, scan control, scan data, scan enables and PCM mapping to the rest of the NIA. The CLK logic also contains logic for reporting hard and soft errors to the NCU. The CLK logic provides over 300 clocks to the rest of the board. Instead of defining each clock, the next section will describe the clock naming convention used on the NIA. After that, the following section will describe individual signals sourced from the CLK logic.

2.2.12.1 Clock Naming Convention

The NIA receives two clocks from the NCU: a 2x clock used to generate clocks to the majority of the NIA, and a 3x clock used mainly to generate clocks to the CCUs. Both the 2x and the 3x clocks are divided down using a method that is common to all C3800 system boards. This clock generation method is fully defined in section 8.1 on page 109.

Table 2-45 Clock Name Prefixes

clk	normal 1x clock. 60Mhz
clk2	2x clock. Used by STRAMs. 120 Mhz
clkp	Pbus clock. Used by Pbus Interfaces. 10 Mhz
clkx	NXP clock. Used by NXP Interface. 30Hhz
clkf	Free running 1x clock. 60 Mhz
clkf3	Free running 3x clock. 180 Mhz

All clocks on the NIA begin with one of the six prefixes defined in Table 2-45. The most common clock prefix is *clk*. Clock signals that begin with *clk* are normal 1x system clocks (60Mhz). Clock signals that begin with *clk2* run at 120 Mhz or 2x the normal system clock. These clocks are used to clock the Self-timed RAMs (STRAMs) that are used to implement the Read and Write Data Queues. Clock signals that begin with *clkp* are used with the Pbus interfaces on the NIA. These clocks run at the 10 Mhz rate and are used to clock Pbus data in and out of the NIA as well as the Pbus Interface state machines. The NXP Interface uses *clkx* clock signals. These clocks run at the NXP clock rate of 30 Mhz.

All of the previously described clock types are derived from the 2x clock input to the NIA. The next two types, *clkf* and *clkf3*, are derived from the 3x clock input to the NIA. The 3x clock input is considered the free running clock of the NIA. Clock signals that begin with *clkf* are 1x (60 Mhz) free running clocks. Clock signals that begin with *clkf3* are 3x (180 Mhz) free running clocks.

Clock signals on the NIA follow the general naming convention of "prefix.x.y.z", where prefix is one of the previously described prefixes and x,y and z are clock

level designators. The clock level designator x defines the first level buffer output for that clock. The y designator defines the second level buffer output for that clock, and z defines the third level buffer output from which the clock is sourced. Take the clock signal *clk.3.2.0* as an example: *clk* defines it as a normal 1x clock. The *3.2.0* means that the clock is sourced from the "0" output from the third level buffer which is driven by the "2nd" output from the second level buffer, which is driven by the "3rd" output from the first level buffer.

This naming convention makes it easy to determine the expected skew between two clocks. For example *clk.3.2.0* and *clk.3.2.7* differ by only the 3rd level buffer output. This means that the expected skew between these two clocks is the 3rd level buffer pin to pin skew (100ns). The clocks *clk.3.2.5* and *clk.3.4.6* have different second levels which means that the two clocks have 100 ns of skew (q to q) contributed from the second level plus 150 ps of skew (d to q) from the third level for a total of 250 ps. Clocks that differ at the first level have 100ns (1st level) + 150ns (2nd level) + 150ns (3rd level) = 400ns total clock skew.

The free running clocks (*clkf* and *clkf3* prefixes) have only two levels of clock buffering on the NIA. This is because the first level of buffering occurs on the NCU. Each NIA in the system receives one of the 3x clock outputs from this first level buffer on the NCU. Clock skew between the free running and non-free running clocks is the same as board to board clock skew which is defined in the NCU board specification.

2.2.12.2 CLK Interface Signals

The CLK logic sources signals to every block of logic on the NIA. A lot of these signals go to multiple blocks and provide the same functionality to each block. Therefore, the CLK interface signals will be defined in one table with a reference to the destination block or blocks.

Table 2-46 clk signals

ck.1x_clk_rst	rdq, wdq	1x clock reset. Resets the "toggle" state used to control the RDQ and WDQ address multiplexers.
ck.cast_mode	rqa, wqa	Cast Mode. Asserted during CAST scan control mode. Prevents writing to the RDQ and WDQ.
ck.clk_count<4..0>	rqa, wqa	Clock cycle Counter. Defines the 18 clock cycles used to determine RDQ and WDQ access priorities. Also defines the Pbus and NXP clock edges.
ck.force_rdq_rd	rdq	Force RDQ read. Scan loadable control bit that forces read operations to the RDQ (no writes).
ck.force_rdq_wrt	rdq	Force RDQ write. Scan loadable control bit that forces write operations to the RDQ (no reads).
ck.force_wdq_rd	wdq	Force WDQ read. Scan loadable control bit that forces read operations to the WDQ (no writes).
ck.force_wdq_wrt	wdq	Force WDQ write. Scan loadable control bit that forces write operations to the WDQ (no reads).
ck.hard_error	xds_arrays, rqa	Hard Error. Internal copy of hard error indication sent the NCU. Used to hold internal state that may be used to help determine the cause of the error.

Table 2-46 clk signals continued

ck.nxp_sync[0-6]	nxi, xbi, rqa, cds_arrays, wqa,pa	NXP Sync. Asserted during the system clock cycle just prior to an NXP clock edge. Used to qualify registers and control signals that interface to the NXI.
ck.par_ck_ena *ck.par_ck_ena**	nxi, pbiX rqa, pa	Parity Check Enable. Scan loaded control used to enable parity checking at different points on the NIA.
ck.pbus_sync[0-6]	pbiX, pi, xbi, cds_arrays, rqa, pa	Pbus Sync. Asserted during the system clock cycle just prior to a Pbus clock edge. Used to qualify registers and control signals that interface to the PBIx.
ck.pcm<31..0>	pbiX, nxi	PCM mapping bits 31..0. Used to check memory write addresses at the Pbus X and NXP interfaces. Read addresses are checked in the XDS arrays.
ck.soft_scan_en *ck.soft_scan_en[0,1]*	pbiX, nxi cds_arrays	Soft log Scan Enable. Used to enable soft error log scan operations.
ck.soft_sdo	pbi0	Soft error log Scan Data Out. Soft error log scan data from the CLK logic to the PBI0 logic.
ck.rdq_wpar_cken	rdq	RDQ Write Parity Check Enable. Scan loaded control used to enable parity checking on data written into the RDQ. Check is performed by RDQ RAMs.
ck.wdq_wpar_cken	wdq	WDQ Write Parity Check Enable. Scan loaded control used to enable parity checking on data written into the WDQ. Check is performed by WDQ RAMs.

3 I/O Channel Interface

The I/O Interface is the largest section of logic in terms of both space and the number of gates on the NIA. The I/O Interface contains seven independent subsections or blocks: PBI0, PBI1, PBI2, PBI3, NXI, PI, and CDS_ARRAYS. Each Pbus interface and NXP interface operates independent of each other. Each Pbus interface is identical in design and function. The Channel Data Slice (CDS) gate arrays are controlled in part by each Pbus/NXP interface and by arbitration logic. This chapter will describe the operation of the Pbus Interface (only one is described since all four Pbus interfaces are the same), the NXP Interface (which is only slight different from the Pbus interface), the Pbus Interrupt interface, and an overview of the CDS arrays.

3.1 Pbus Interface

The Pbus Interface should be the best understood portion of the NIA since the Pbus as been around since the beginnings of Convex. In fact, the Pbus state machine was borrowed from the PIA design used in the C2 series. Only the implementation of the Pbus state machine and control signals have changed. The Pbus interface also generates signals used by the Port Arbitration and queue arbitration logic.Table 3-1 lists the I/O signals for the Pbus Interface block.

3.1.1 PBI Data Path

The Pbus Interface (PBI) is built around the Pbus data path. The PBI drives and receives the Pbus with latch translators. All TTL signals on the NIA are immediately translated to or from ECL at the Augat connector. Write data and header data transfers received by the NIA are translated and then sent to the CDS gate arrays. Header data is also sent to decode logic, an address counter and a transfer counter within the Pbus Interface. Read return data comes from the CDS gate arrays and is translated at the Pbus Interface. The output enable of the TTL-to-ECL translators for read data returns is controlled by the Pbus interface state *r.rdxfer* or read transfer state. See Figure 3-1.

The PBI logic decodes header information and loads an eight bit decode register at the end of the header transfer cycle. The PBI state machine uses decode information to determine if the header is valid and the type of transfer state to enter (read vs. write). Also, the PBI sends specific decode bits to the Port Arbitration logic and the Read Queue Arbitration logic. The Port Arbitration logic uses *pbiX.mbp_hdr* and *pbiX.toc_hdr1* to force selection of the Memory Base Pointer (MBP) or the Time of Century (TOC) header registers in the CDS_ARRAYS. The Read Queue Arbitration logic uses *pbiX.toc_hdr* to force a read access the Read Data Queue for TOC reads.

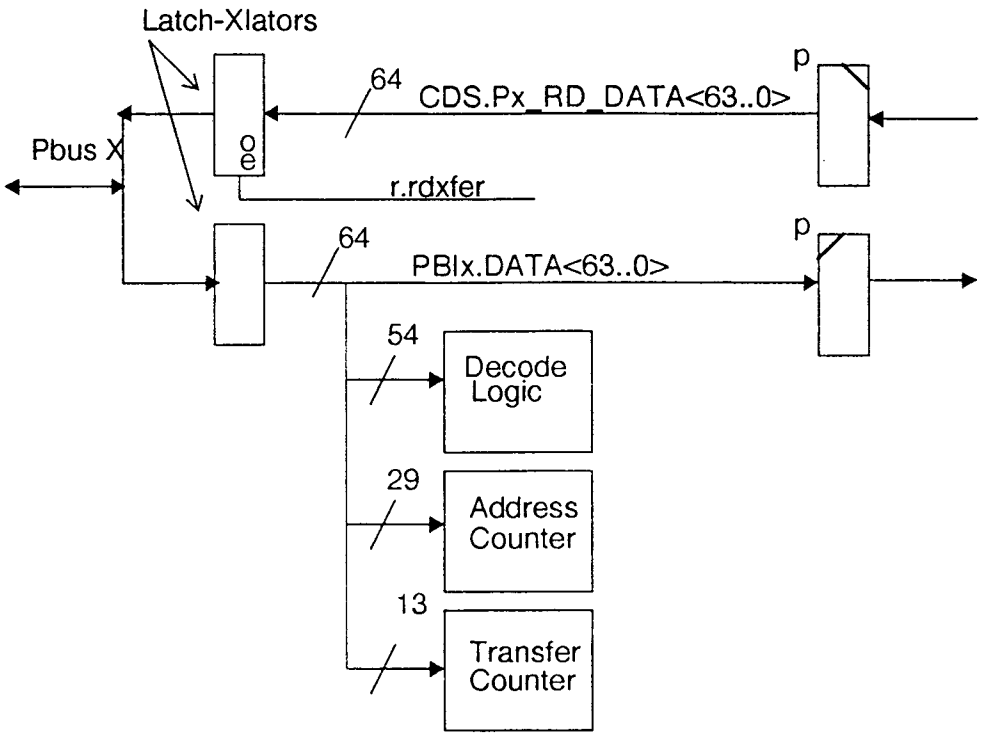
The address counter is loaded at the end of the header transfer cycle with bits 31..3 of the header address field. The address counter is implemented in four 8-bit counters. The counter itself (100e016) is not scannable, however pals are used to multiplex header data or scan data depending upon the desired operation. The address counter is incremented at bit position 3 for each write data transfer received by the NIA. Incrementing at bit position 3 effectively adds a value of eight to the 32-bit byte address. The address counter value is used for PCM checking. If write data is sent to a non-existent memory address, the PBI will generate a bus

Table 3-1 PBIx I/O Signals

BIDIRECTIONAL	
BPx_BPx.P_DATA<63<0>	Pbus Data bits 63..0,bidirectional
BPx_BPx.P_PAR<7..0>	Pbus Parity bits 7..0,bidirectional
BPx_BPx.P_DVAL*	Pbus Data Valid, bidirectional
INPUTS	
BPx_IA.P_CBAV*	Pbus Channel Buffer Available from CCUs
CCUxy_IA.ERR*	Hard error from CCUxy
CCUxy_IA.P_CCR*	Pbus CCU bus Request from CCUxy
CDS.Px_RD_DATA<63..0>	Pbus x Read Data bits 63..0 from CDS_ARRAYS
CDS.Px_RD_PAR<7..0>	Pus x Read Data Parity bits7..0 from CDS_ARRAYS
CDSy.PBx_PERR	Pbus x Data Parity Error from CDS gate array y
CK.PAR_CK_ENA	Parity Check Enable from CLK block
CK.PBUS_SYNCy	Pbus Sync clock enable from CLK clock
CK.PCM<31..0>	PCM bits 31..0
CK.SOFT_SCAN_EN	Soft error log scan enable
PA.RDY_4_Px_HDR*	Ready for Pbus x Header
RQA.Px_RD1_FULL	Pbus x Read Data register 1 is Full
RQA.Px_RD_ERROR	Pbus x Read Data Error
XDS.Px_WDQ_FULL	Pbus x Write Data Queue is Full
PBIx.R_WRT_PCM_ERR	Pbus x registered write PCM Error
PBIy.SOFT_SDO	Pbus Interface y Soft error log Scan Data Out
Z.PBIx_SDI	Pbus Interface x Scan Data In
OUTPUTS	
IA_BPx.P_BUSERR	Pbus Bus Error
IA_BPx.P_MBAV*	Pbus Memory Buffer Available to CCUs
IA_CCUxy.P_CCG*	Pbus CCU bus Grant to CCUxy
PBIx.DATA<63..0>	Pbus x Data In to CDS_ARRAYS
PBIx.PAR<7..0>	Pbus x Data Parity In to CDS_ARRAYS
PBIx.LAST_WR_XFR	Last Write Transfer to Port Arbitration logic
PBIx.LOAD_HDRy	Load Header register version y to CDS_ARRAYS
PBIx.RD_HOLDy	Read Data Hold version y to CDS_ARRAYS
PBIx.RD_XFR_ABORT	Read Transfer Abort to Port Arbitration logic
PBIx.VAL_WR_DATA	Valid Write Data to Write Queue Arbiter
PBIx.WRT_XFR_ABORT	Write Transfer Abort to Write Queue Arbiter
PBIx.MBP_HDR	Memory Base Pointer Header to Port Arbiter
PBIx.NEW_RD_HDR	New Read Header to Port Arbiter
PBIx.NEW_TOC_HDR2	New Time Of Century Header to Port Arbiter
PBIx.TOC_HDR	Time of Century Header to Read Queue Arbiter
PBIx.TOC_HDR1	Time of Century Header to Port Arbiter
PBIx.CCU1_GNT	CCU1 of Pbus x has the Pbus Grant
PBIx.SOFT_ERROR	Pbus Interface x Soft Error
PBIx.BUS_ERROR	Pbus x Bus Error
PBIx.C_WRT_PCM_ERROR	Pbus x combinational Write PCM Error
PBIX.SOFT_SDO	Pbus Interface x Soft error log Scan Data Out
Z.PBIx_SDO	Pbus Interface x Scan Data Out

error on the following cycle and then abort the current transfer. PCM checking for read transfers is performed at the Crossbar Interface.

Figure 3-1 PBI Data Path



The transfer counter (also called long word counter or LWC) is used to count the number of Pbus transfers during a block mode operation. The LWC is implemented using the same eight-bit counters as the address counter. The LWC is loaded with the upper 13-bits of the Byte Count field of the Header. These 13 bits are inverted prior to loading the LWC. Incrementing the inverted value yields an effective decrement operation. When the counter value reaches x1FFF (all ones), the transfer is complete and the Pbus state machine returns to the IDLE state. Actually, the Pbus state machine looks for an LWC value of one (x1FFE) and returns to the IDLE state following a successful Pbus transfer when *c.1_xfr_left* is true.

The LWC value alone is not sufficient to count Pbus transfers. Two additional states are needed: *r.lwc_cflag* and *r.lwc_rflag*. These two states are loaded along with the LWC at the end of the header cycle. The LWC is a modulo 8 of the header's byte count field. The LWC is sufficient only when the block transfer is long word aligned (i.e. the block transfer starts and ends on a long word boundary). However, long word alignment is not a requirement for Pbus transfers. The *r.lwc_cflag* (carry flag) and the *r.lwc_rflag* (remainder flag) are used with misaligned transfers. The carry flag and the remainder flag are conditionally set as a result from adding the lower three bits of the header address field (called the long word offset) to the lower three bits of the header byte count. If the three bit addition results in a carry out, the carry flag is set. If the addition results in the sum bits 2..0 being non-zero, then the remainder flag is set. The carry flag and the remainder flag represent one or two Pbus transfers in addition to the LWC.

3.1.2 PBI Control Logic

The PBI control logic has three main parts: the Pbus arbitration logic, the Pbus state machine, and control signal outputs. Control signal outputs are generated for the Pbus channel and for port and queue arbitration logic on the NIA. These control signals are based upon the current state of the PBI and the Pbus channel control inputs.

3.1.2.1 Pbus Arbitration Logic

The Pbus arbitration logic controls the granting or "mastership" of the Pbus to CCUs. CCUs must request for the right to become master of the Pbus. The NIA grants mastership of the Pbus to a CCU by asserting the CCU's grant control. Pbus protocol dictates that a CCU must assert its request for the duration of the Pbus transfer. If the CCU removes its bus request, the NIA will remove the CCU's grant on the following Pbus cycle.

Two CCUs per Pbus are supported. The Pbus arbiter is a "fair" arbiter in that if both CCUs request the Pbus at the same time, then the arbitration logic will grant the Pbus to the least recently granted CCU. Granting a particular CCU is determined by the state of *r.ccu0_mrg* (ccu 0 is the Most Recently Granted device). This state is set when CCU0 is granted the Pbus and remains set until CCU1 is granted the Pbus.

Granting mastership of the Pbus to either CCU is a function of two states: *c.set_gnt* and *c.enabgnt*. The set_grant state controls the initial granting of the Pbus. Set_grant means that a CCU is requesting the Pbus and all the right conditions exist in order to grant the Pbus. "All the right conditions" means that the Pbus state machine is in the IDLE state (explained later) and the state machine has not just returned to the IDLE state, and the Port Arbiter is ready for another header from the PBI. Enable bus grant, *c.enabgnt*, is the holding term for Pbus grants after the initial setting. Enable bus grant is true when the Pbus state machine is not in the IDLE state and no error conditions exist and the block transfer has not yet completed. When the block transfer completes or if an error occurs, enable bus grant goes false and the Pbus grant is removed.

3.1.2.2 Pbus State Machine

The Pbus state machine sequences through the standard fives state of the Pbus protocol: IDLE, GRANT, HEADER, DECODE, and XFER. The Pbus state machine starts out in the IDLE state, waiting for a CCU request. The state machine can be scan initialized to the IDLE state or can be forced to the IDLE state by resetting the NIA.

When a CCU bus request is received by the NIA and the Pbus arbiter is ready to grant the Pbus to a CCU, the Pbus state machine enters the GRANT state. The GRANT state gives the bus grant indication one Pbus cycle to reach the targeted CCU. The HEADER state is unconditionally entered on the Pbus cycle following the GRANT state. During the HEADER state the CCU drives the Pbus with its transfer header. At the end of the HEADER cycle, the NIA registers the header internal to the CDS gate arrays. Also at the end of the HEADER cycle, the PBI captures decoded bits and loads the address and long word counters.

If the CCU is still requesting the Pbus, the state machine next enters the DECODE state following the HEADER state. During the DECODE state, the Pbus state

machine examines the decode information and looks for header parity errors. If there are no parity errors and the header requires Pbus transfers, the state machine enters the XFER state on the cycle following the DECODE state. The Pbus state machine remains in the XFER state until the transfer is complete, or the CCU aborts the transfer prior to completion, or an error condition is detected. The CCU can abort a transfer by removing its bus request prior to completing the transfer. Error conditions which terminate a Pbus transfer include write data parity errors, write data PCM errors or read data PCM errors. The Pbus state machine is diagramed in Figure 3-2.

3.1.2.3 Control Signal Outputs

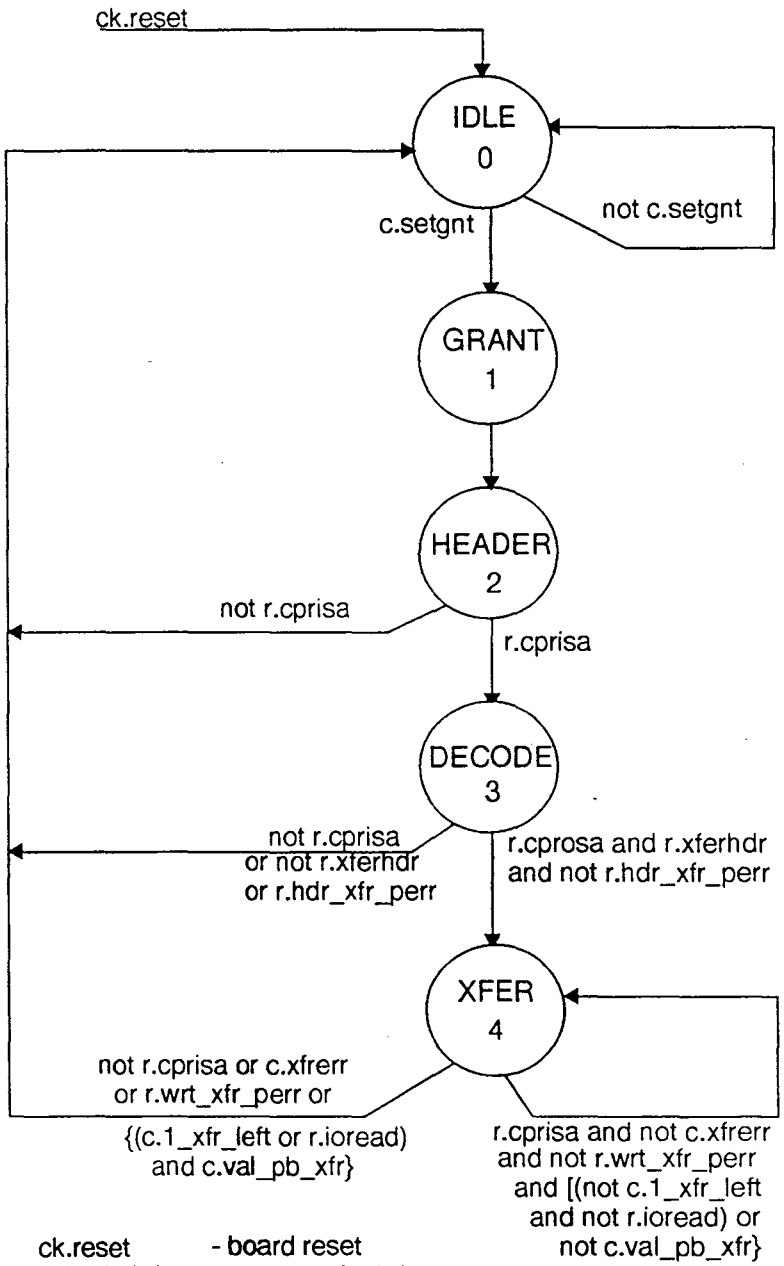
The PBI sources control signals to the Pbus channel and to different parts of the NIA. Pbus channel control signals regulate and control Pbus operations. Pbus data validity is defined by the *bpX_bpX.p_dval* Pbus channel control signal. This bidirectional control signal is driven by the PBI during read data returns and is driven by the granted CCU during write data transfers to the NIA. The PBI drives this signal when the NIA is in a read transfer state and the read data register is full and does not contain a read data error. The PBI can regulate write data from the CCU by de-asserting the *ia_bpX.p_mbav* control signal. When mbav is false, the PBI will not accept write data from the CCU. Therefore a successful write data transfer needs to have good data parity, the dval control from the CCU is true and the mbav control from the PBI is true. The PBI will terminate a Pbus transfer if an error is detected by asserting the *ia_bpX.p_buserr* control signal.

The PBI sources control signals to the Port Arbitration logic, the Read Queue Arbitration logic and the Write Queue Arbitration logic. The Port Arbitration logic receives the bulk of these control signals including: *pbiX.last_wr_xfr*, *pbiX.wrt_xfr_abort*, *pbiX.rd_xfr_abort*, *pbiX.new_rd_hdr*, *pbiX.toc_hdr1*, and *pbiX.mbp_hdr*. The *pbiX.last_wr_xfr* control tells the Port Arbiter that the last write transfer has been received by the PBI and to send this data to the Crossbar Interface. The *pbiX.wrt_xfr_abort* signal tells the Port Arbiter that the CCU has aborted its block transfer of write data early and to send write data in the WDQ to the Crossbar Interface. The *pbiX.rd_xfr_abort* signal means that the CCU has aborted a read operation and that the Port Arbiter needs to flush any previously requested read data and any pending read data requests.

The signals *pbiX.new_rd_hdr*, *pbiX.new_toc_hdr1*, and *pbiX.mbp_hdr* mean that the PBI has received a new read header, or a new TOC header, or an MBP header respectfully. The PBI uses these control signals to notify the Port Arbiter that a memory request has been received and to service these requests as soon as possible.

The PBI sends two control signals to the Read Queue Arbiter: *pbiX.toc_hdr* and *pbiX.new_toc_hdr2*. The Read Queue Arbiter uses *pbiX.toc_hdr* to modify the Read Data Queue tail pointer address. The *pbiX.new_toc_hdr2* signal is used to force a read access of the TOC from the Read Data Que. The Write Queue Arbiter also receives two control signals from the PBI: *pbiX.val_wr_data*, and *pbiX.wrt_xfr_abort*. The Write Queue Arbiter uses *pbiX.val_wr_data* to select the PBI's write data register from the CDS gate arrays and store this write data into the Write Data Que. The *pbiX.wrt_xfr_abort* is used to store an abort flag into the Write Data Que. This flag is used by the Crossbar Interface to know that the write transfer was aborted prior to completion.

Figure 3-2 Pbus State Machine



Other control signals worthy of noting are *pbiX.rd_hold* and *pbiX.load_hdr*. These signals are sent to the CDS gate arrays for data path control. The *pbiX.rd_hold* signal controls the clock enable of the Pbus clocked read data register in the CDS gate arrays. This register holds the current read return data sourced by the PBI during read transfers. The *pbiX.load_hdr* forces the loading of a new header into the header holding register internal to the CDS arrays.

3.1.3 Error Detection and the Soft Error Log

The PBI detects and reports several errors related to the Pbus Channel. All of these errors are soft errors which are reported to the SPU for soft error log scanning. Upon detection of any of these transfer related errors, the PBI will terminate the transfer by asserting the Pbus error control signal, *ia_bpX.p_buserr*, and by removing the CCU's grant signal. Two of the errors are hard errors from the CCUs. These CCU hard errors are not transfer related errors. CCU hard errors are reported to the SPU as soft errors. Table 3-2 defines the soft errors detected by the PBI as they appear in the Soft Error Log. The LSB of the Soft Error Log is a CCU identifier for tagging soft errors to specific CCUs on the Pbus Channel.

Table 3-2 PBI Soft Errors

R.ILL_HDR	- Illegal header	— MSB
R.WRT_PCM	- Write transfer PCM error	
R.RD_PCM	- Read transfer PCM error	
R.WRT_PE	- Write transfer parity error	
R.HDR_PE	- Header transfer parity error	
R.CCU0_ERR	- CCU0 hard error	
R.CCU1_ERR	- CCU1 hard error	
R.CCU0_MRG_SL	- CCU0 most recent grant for the soft log	— LSB

Illegal headers are detected during the DECODE state and result in a bus error on the following cycle. Write transfer parity errors and header transfer parity errors are detected by the PBI with help from the CDS gate arrays. Each CDS array checks byte parity of the Pbus data bus and reports parity errors to the PBI. Even parity is expected on the Pbus. The PBI gathers all eight of these parity error indications from the CDS arrays and qualifies these indications with Pbus state machine outputs.

The PBI logic checks for PCM violations on write transfers only. Read transfers are checked by the Crossbar Interface logic. PCM checking verifies that memory is resident at the specified address. Installed memory may or may not reside in a contiguous address space. "Holes" in the memory space may exist depending upon the number of memory boards installed and the size of each memory board. The PCM (Physical Configuration Map) defines what memory addresses are valid based upon the number and size of memory boards installed in the system. The PCM information is encoded and scan loaded into the NIA, *ck.pcm<31..0>*. This function is provided by the memory initialization program "mminit".

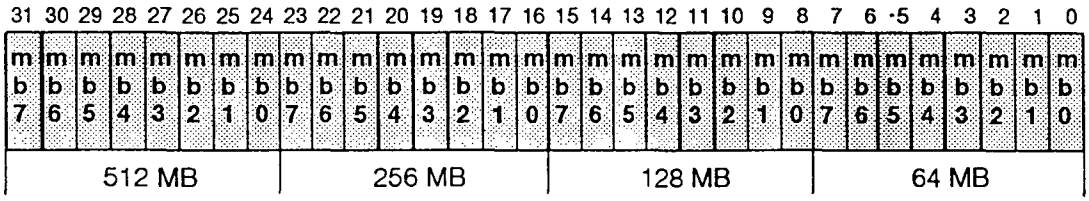
Up to eight memory boards can be installed in a system. The PCM state, *ck.pcm<31..0>*, describes which memory boards are installed and the size of each memory board installed See Figure 3-3. Valid memory board sizes are 64MB, 128MB, 256MB and 512MB. Bits 31..24 of *ck.pcm<31..0>* define the 512 MB memory boards that are installed in the system. Bits 23..16 define 256 MB boards, 15..8 define the 128 MB boards and 7..0 define 64 MB boards.

Pbus protocol dictates that all write data sent prior to the PCM violation must be written out to memory. Likewise, all read data requested prior to the PCM violation must be returned to the CCU. For write operations, the transfer is terminated on the cycle following the Pbus cycle in which write data destined to non-existent memory was sent to the PBI. For read operations, the transfer is terminated on the cycle in which the read data, which caused the PCM violation,

would have been returned had the memory been resident in the system. In both cases the transfer is terminated by the PBI asserting the bus error indication to the CCU and then removing the CCU grant on the following cycle.

Figure 3-3 PCM State

ck.pcm<31..0>



3.2 Pbus Interrupt Interface

The Pbus Interrupt Interface (PI) contains the logic necessary to arbitrate and control the Pbus Interrupt Bus. Logically, all four Pbus channels tie into one "Pbus" interrupt bus. However, two Pbus channels are located to the right side of the I/O Bay and two channels are located to the left. This makes having a single interrupt bus an electrically undesirable thing to do. Therefore, there are two Pbus interrupt busses: *bp01_bp01.p_ibintv* and *bp23_bp23.p_ibintv*. CCU00-11 connect to *bp01_bp01.p_ibintv* and CCU20-31 connect to *bp23_bp23.p_ibintv*. Only one device (CCU or NIA) will be driving the interrupt bus(es) at one time.

Table 3-3 Pbus Interrupt Interface I/O Signals

BIDIRECTIONAL	
bp01_bp01.p_ibintv<7..0>	- Pbus interrupt bus bits 7..0 for channels 0 & 1
bp23_bp23.p_ibintv<7..0	- Pbus interrupt bus bits 7..0 for channels 2 & 3
OUTPUTS	
ia_bpxy.p_ibintv*	- Pbus interrupt bus valid for channels xy = 01 and 23
ia_bpxy.p_ibicomp*	- Pbus interrupt bus complete for channels xy = 01 and 23
iia_ccuxy.p_ibgrnt*	- Pbus interrupt bus grant for ccu xy = 00,01,10,11,20,21,30,31
pi.hard_error	- Hard error
pi.pb_busy	- Pbus interrupt interface is busy
pi.pb_intr_ready	- Pbus interrupt ready to XBI
pi.pb_intvec<7..0>	- Pbus interrupt interface vector bits 7..0 to XBI
z.pi_sdo	- scan data out from PI
INPUTS	
ccuxy_ia.p_ibreq*	- Interrupt bus request from ccu xy = 00,01,10,11,20,21,30,31
ck.pbus_sync2	- Pbus clock sync version 2
xbi.ncu_intr_rdy_p	- NCU interrupt ready for Pbus interrupt interface
xbi.ncu_intvec<7..0>	- NCU interrupt vector bits 7..0
xbi.pb_intr_ack	- pbus interrupt acknowledge from XBI
z.pi_sdi	- scan data in to PI

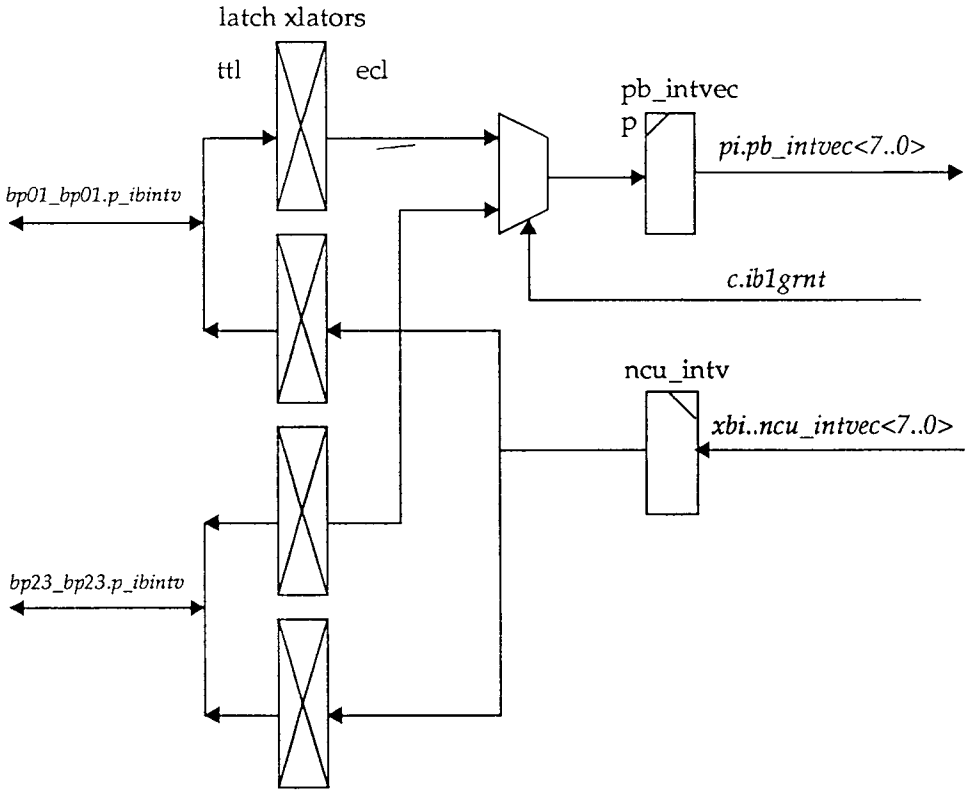
3.2.1 Pbus Interrupt Interface Data Path

The Pbus Interrupt Interface (PI) has two data path components: the receive path and the send path. The PI receive path contains an eight bit register to hold interrupts received from CCUs. The two Pbus interrupt busses are translated to

ECL logic levels at the Augat connector and then multiplexed into the pb_intvec register. See Figure 3-4. The multiplexor selects between *bp01_bp01.p_ibintv* and *bp23_bp23.p_ibintv* as the input to the pb_intvec register. The pb_intvec register is loaded at the end of the PI transfer state and is held until the XBI acknowledges that it has written the interrupt vector into the Write Data Que.

The send path contains an eight bit register to hold the interrupt vector from the XBI. This register (called the ncu_intv) is loaded with an interrupt vector from the NCU. The XBI asserts the *xbi.ncu_intr_rdy_p* signal to the PI when it receives an NCU interrupt. If *pi.pb_busy* is false, the PI will load the ncu_intv register with the *xbi.ncu_intvec<7..0>* input bus. If *pi.pb_busy* is true, then the PI already has an NCU interrupt vector in the ncu_intv register and the XBI must wait until the PI has processed the current vector. Once loaded, the PI will use the ncu_intv register to drive ECL to TTL translators which in turn drive both Pbus interrupt buses with the ncu interrupt vector. SeeFigure 3-4.

Figure 3-4 PI Data Path



3.2.2 Pbus Interrupt Interface Arbiter

The PI bus arbiter is a polling round robin arbiter. Each CCU is polled is a round robin fashion checking for interrupt bus requests. Round robin polling is achieved by incrementing a four bit counter and using the counter value to point to a CCU's interrupt bus request. All eight CCUs which interface to the NIA across the four Pbus channels are polled in this manner. Only one out of the eight CCUs will be selected at any one time.

The four bit counter yields 16 different count values. Eight of these values point to the eight CCUs. The other eight values are used to check the ncu_intvec register to

see if an interrupt has been received from the NCU via the XBI. All odd count values are used to check the ncu_intvec register for an NCU interrupt. The arbitration counter increments each pbus cycle until an interrupt bus request is found. Once a request is found, the PI state machine sequences through an interrupt bus cycle and the arbiter count value is held. Once the interrupt bus cycle is complete, the arbiter count value is incremented and the next request is checked.

3.2.3 Pbus Interrupt Interface State Machine

The PI state machine sequences through a Pbus interrupt bus cycle when an interrupt request is found. The PI state machine has six states and starts out in the IDLE state. The PI state machine can be reset or scan initialized to the IDLE state. In the IDLE state, the PI state machine waits for the interrupt bus arbiter to grant the bus to either a CCU or to the PI itself.

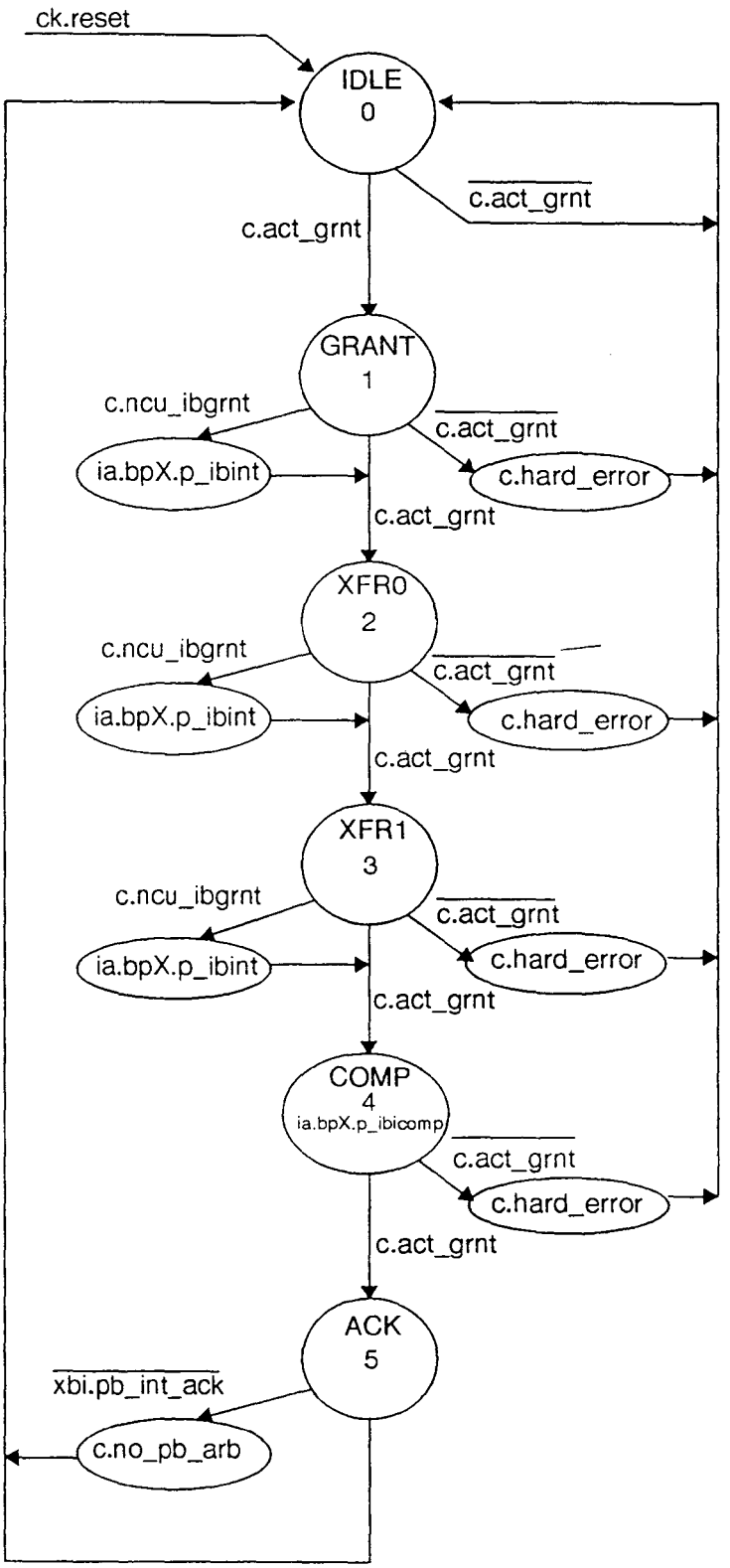
Once the arbiter has granted the interrupt bus, the state machine enters the GRANT state. The GRANT state allows the grant indication time to reach the selected CCU. When the CCU receives the bus grant, it will enable its output drivers onto the interrupt bus. If the PI has the bus grant, its drivers are enabled onto the interrupt bus and the PI asserts the interrupt valid control signal, *ia_bpXY.p_ibint*. If the selected device is still requesting the interrupt bus, the PI state machine enters the XFR0 state on the following Pbus cycle. If the granted CCU removes its bus request, the PI state machine returns to the IDLE state and a hard error is set.

The XFR0 state and the next two states, XFR1 and COMP, allow the interrupt bus time to settle out. The PI state machine sequences through the XFR1 state and then the COMP state as long as the granted CCU is still asserting its request. If a CCU is sourcing the interrupt, the PI captures the interrupt vector at the end of the XFR1 state and sets the interrupt ready state, *pi.pb_intr_rdy*, to alert the XBI. During the COMP state the PI asserts the interrupt complete control signal, *ia_bpxy.p_ibicomp*, to inform CCUs that the interrupt bus cycle is complete.

The ACK state is entered following the COMP state. The ACK state is used by the PI to remove its drivers from the interrupt bus and to check the status of the interrupt acknowledge from the XBI if the interrupt ready state is set. If the XBI acknowledges the interrupt, the interrupt ready state is reset. However, if the XBI has not acknowledged the interrupt, the interrupt ready state remains set and the no pbus arbitration, *r.no_pb_arb*, state is set. In either case, the PI state machine returns to the IDLE state and the arbitration counter is incremented to check the next request.

The r.no_pb_arb state is used to control the arbitration counter. As long as this state is set, the arbiter counter will hold its current count value. This value will always be odd (lsb is set) because *r.no_pb_arb* can only be set following a CCU sourced interrupt transfer. The arbiter counter is unconditionally incremented in the ACK state and is conditionally held in the during the following cycles based upon *r.no_pb_arb*. The *r.no_pb_arb* state is reset when the XBI acknowledges the PI interrupt.

Figure 3-5 PI State Machine



3.3 Expansion Pbus (NXP) Interface

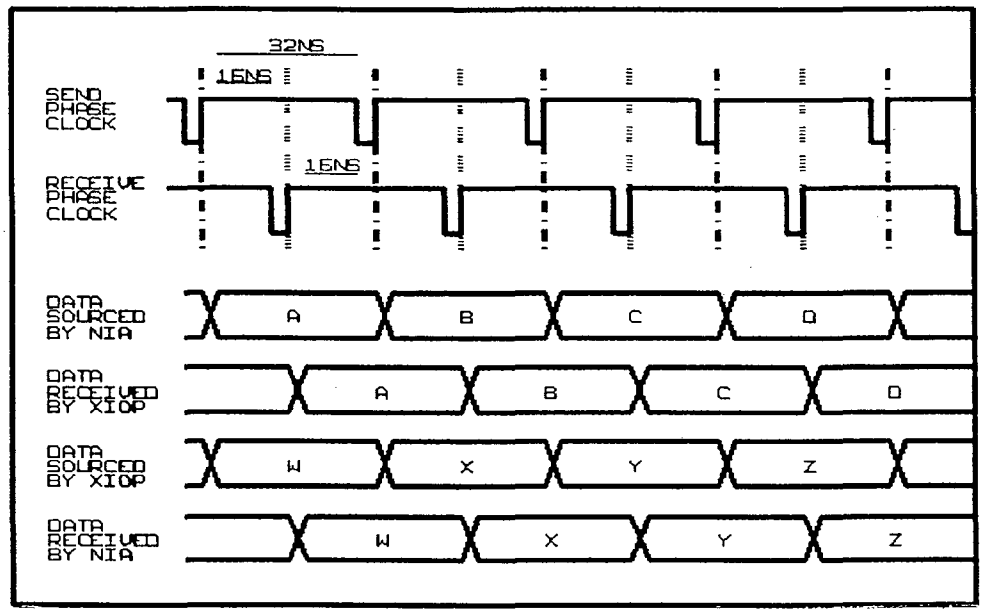
The Expansion Pbus (NXP) is basically a 30Mhz, ECL version of a normal Pbus. The NXP is a higher performance alternative to the Pbus, the general purpose I/O interface to the C38xx system. The NXP is a block mode transfer bus to an Input/Output Processor (IOP). This IOP which connects to the NXP (called an XIOP for Expansion port IOP) is a special purpose I/O device for applications with any high performance data hose. In the base I/O bay, the SPU accesses system memory through the NXP via the CPU Utilities (NCU) board.

The NXP Interface (NXI) on the NIA contains the NXP state machine for controlling NXP protocol, NXP bus arbitration logic, header decode logic, and counters for tracking address and transfer counts. The NXI also contains the NXP interrupt state machine and control logic. The NXI generates control signals for the NXP data bus and interrupt bus, and generates control signals for the port and queue arbitration logic. The NXI also generates signals to control the NXP data path implemented in the CDS gate arrays. NXI I/O signals are listed in Table 3-4. The NXI data path and control logic are discussed in the next two sections followed by a discussion of the NXP interrupt logic.

3.3.1 NXI Data Path

NXP data and control signals are deskewed at the receiving side of a transfer. These deskewing registers are clocked at the mid-way point of an NXP bus cycle. Therefore, data and control transfer from one device to another across the NXP in 1/2 the NXP cycle time. See Figure 3-6. The NXI provides two buffer registers for read data and two buffer registers for write data. This is twice the buffering of the PBI. The additional buffering is needed because Pbus channels have a higher priority to accessing the Read Data Queue and the Write Data Que.

Figure 3-6 NXP Transfer Timing



External ECLiPS 100e241 registers are used to deskew header transfers received by the NIA. Deskewing registers are also implemented internal to the CDS gate arrays to deskew write data as well as header transfers. The output of the external deskewing registers is sent to header decode logic, and is used to load an address and a transfer counter (similar to the PBI).

The NXI decodes header information and loads an eight bit decode register at the end of the header transfer cycle. The NXI state machine uses decode information to determine if the header is valid and the type of transfer state to enter (read vs. write). The NXI sends specific decode bits to the Port Arbitration logic and the Read Queue Arbitration logic. The Port Arbitration logic uses *nxi.mbp_hdr* and *nxi.toc_hdr1* to force selection of the Memory Base Pointer (MBP) or the Time of Century (TOC) header registers in the CDS_ARRAYS. The Read Queue Arbitration logic uses *nxi.toc_hdr* to force a read access the Read Data Queue for subsequent TOC reads.

Table 3-4 NXI Input/Output Signals

BIDIRECTIONAL	
NXP.IBINTV<7..0>	NXP Interrupt Bus bits 7..0I
INPUTS	
NXP.DATA<63..0>	NXP Data bits 63..0
XIOP_IA.WRT_VAL	NXP Write data Valid from XIOPs
XIOP_IA.CBUF_AVL	NXP Channel Buffer Available from XIOPs
XIOPx_IA.ERR	Hard error from XIOPx
XIOPx_IA.BUS_REQ	NXP bus Request from XIOPx
XIOPx_IA.IBREQ	NXP Interrupt Bus Request from XIOPx
CDSy.NXP_PERR	NXP x Data Parity Error from CDS gate array y
CK.PAR_CK_ENA	Parity Check Enable from CLK block
CK.NXP_SYNCy	NXP Sync clock enable from CLK clock
CK.PCM<31..0>	PCM bits 31..0
CK.SOFT_SCAN_EN	Soft error log scan enable
PA.RDY_4_NXP_HDR*	Ready for NXP x Header
RQA.NXP_RDx_FULL	NXP Read Data register x is Full
RQA.NXP_RDx_ERROR	NXP Read Data Error in register x
WQA.HLD_WDx_FULL	Hold NXP Write Data register x Full
WQA.NXP_WD_FULL	NXP Write Data registers are both Full
WQA.NXP_WD_HAZ	NXP Write Data Hazard
XBI.NCU_INTR_RDY_X	NCU Interrupt Ready to NXP
XBI.NCU_INTVEC<7..0>	NCU Interrupt Vector bits 7..0
XBI.NXP_INTR_ACK	NXP Interrupt Acknowledge from Crossbar Interface
XDS.NXP_WDQ_FULL	NXP Write Data Queue is Full
PBI3.SOFT_SDO	Pbus Interface 3 Soft error log Scan Data Out
Z.NXI_SDI	NXP Interface Scan Data In
OUTPUTS	
IA_XIOP.BUS_ERR	NXP Bus Error
IA_XIOP.MBUF_AVL	NXP Memory Buffer Available to XIOPs
IA_XIOPx.BUS_GNT	NXP CCU bus Grant to XIOPx
IA_XIOP.RD_VAL	NXP Read Data Valid to XIOPs
IA_XIOPx.IBGRNT	NXP Interrupt Bus Grant to XIOPx
IA_XIOP.IBINT	NXP Interrupt Bus Valid to XIOP
NXI.RS_CBUF_AVL	Deskew register Channel Buffer Available

Table 3-4 NXI Input/Output Signals continued

NXI.LAST_WR_XFR	Last Write Transfer to Port Arbiter
NXI.LOAD_HDRy	Load Header register version y to CDS_ARRAYS
NXI.RD_HOLDy	Read Data Hold version y to CDS_ARRAYS
NXI.RD_XFR_ABORT	Read Transfer Abort to Port Arbiter
NXI.VAL_WR_DATA	Valid Write Data to Write Queue Arbiter
NXI.WRT_XFR_ABORT	Write Transfer Abort to Write Queue Arbiter
NXI.WRT_XFR_BERR	Write Transfer Bus Error to Write Queue Arbiter
NXI.WRT_XFR_PERR*	Write Transfer Parity Error
NXI.RD_VAL2_RQA	Read Data Valid to Read Queue Arbiter
NXI.LOAD_WD1	Load Write Data register 1
NXI.SEL_RD1y	Select Read Data register 1 version y
NXI.MBP_HDR	Memory Base Pointer Header to Port Arbiter
NXI.NEW_RD_HDR	New Read Header to Port Arbiter
NXI.NEW_TOC_HDR2	New Time Of Century Header to Port Arbiter
NXI.TOC_HDR	Time of Century Header to Read Queue Arbiter
NXI.TOC_HDR1	Time of Century Header to Port Arbiter
NXI.XIOP1_GNT	XIOP1 of NXP has the NXP Grant
NXI.HARD_ERROR	NXP Interface Hard Error
NXI.SOFT_ERROR	NXP Interface Soft Error
NXI.BUS_ERROR	NXP Bus Error
NXI.ARISA*	NXP Active Request Is Still Around
NXI.XFER_Q	Transfer State, Qualified w/ CK.PAR_CK_ENA
NXI.SOFT_SDO	NXP Interface Soft error log Scan Data Out
Z.NXI_SDO	NXP Interface Scan Data Out

The address counter is loaded at the end of the header transfer cycle with bits 31..3 of the header address field. The address counter is implemented in four 8-bit counters. The counter itself (100e016) is not scannable, however pals are used to multiplex header load data or scan data depending upon the desired operation. The address counter is incremented at bit position 3 for each write data transfer received by the NIA. Incrementing at bit position 3 effectively adds a value of eight to the 32-bit byte address. The address counter value is used for write transfer PCM checking. If write data is sent to a non-existent memory address, the NXI will generate a bus error on the following cycle and then abort the current transfer. PCM checking for read transfers is performed at the Crossbar Interface.

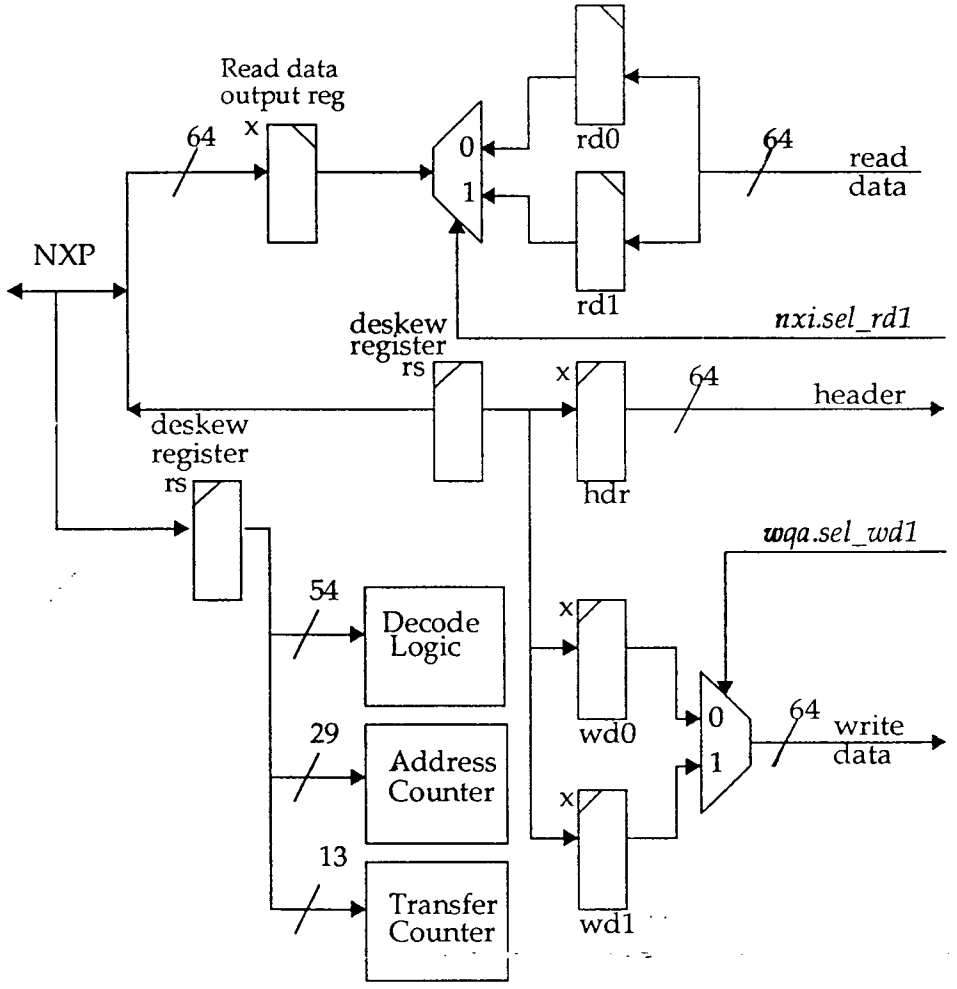
The transfer counter (also called long word counter or LWC) is used to count the number of NXP transfers during a block mode operation. The LWC is implemented using the same type of eight-bit counters as the Address Counter. The LWC is loaded with the upper 13-bits of the byte count field of the header. These 13 bits are inverted prior to loading the LWC. Incrementing the inverted value yields an effective decrement operation. When the counter value reaches x1FFF (all ones), the transfer is complete and the NXP state machine returns to the IDLE state. Actually, the NXP state machine looks for an LWC value of one (x1FFE) and returns to the IDLE state following a successful NXP transfer when *c.1_xfr_left* is true.

The LWC value alone is not sufficient to count NXP transfers. Two additional states are needed: *r.lwc_cflag* and *r.lwc_rflag*. These two states are loaded along with the LWC at the end of the header cycle. The LWC is a modulo 8 of the header's

byte count field. The LWC is sufficient only when the block transfer is long word aligned (i.e. the block transfer starts and ends on a long word boundary). However, long word alignment is not a requirement for NXP transfers. The *r.lwc_cflag* (carry flag) and the *r.lwc_rflag* (remainder flag) signals are used with misaligned transfers. The carry flag and the remainder flag are conditionally set as a result from adding the lower three bits of the header address field (called the long word offset) to the lower three bits of the header byte count. If the three bit addition results in a carry out, the carry flag is set. If the addition results in the sum bits 2..0 being non-zero, then the remainder flag is set. The carry flag and the remainder flag represent one or two NXP transfers in addition to the LWC.

Two write data registers are implemented in the CDS gate arrays. These registers are labeled wd0 and wd1 in Figure 3-7. Write data received from the NXP is deskewed in the CDS gate arrays and then captured in one of the write data registers (wd0 or wd1). The Write Queue Arbitration logic selects which write data register is used to source the Write Data Queue with the *wqa.sel_wd1* signal. The CDS gate arrays also contain two read data registers to buffer read return data back to the XIOP. These registers are labeled rd0 and rd1 in Figure 3-7. Read data from the Read Data Queue is captured in one of the read data registers. The NXI selects which read data register will source the read data output register with the *nxi.sel_rd1* signal.

Figure 3-7 NXI Data Path



3.3.2 NXI Control Logic

The NXI control logic has three main sections: the NXP arbitration logic, the NXP state machine, and control output signals. Control output signals are generated for the NXP channel and for port and queue arbitration logic. These control signals are based upon the current state of the NXP state machine and the NXP channel control inputs.

3.3.2.1NXI Arbitration Logic

The NXP arbitration logic controls the granting or "mastership" of the NXP to either the XIOP or the SPU. Although the arbitration logic is always present on the NIA, it is only needed when the NIA is used in the I/O Bay. I/O expansion bays support only one XIOP per NXP port. In the I/O Bay, the XIOP shares the NXP with the SPU.

The NXI grants mastership of the NXP to the XIOP or SPU by asserting the appropriate bus grant. The grantee must assert its request for the entire duration of the transfer. If the grantee removes its request prior to the completion of the transfer, the NXI will remove the bus grant on the following NXP cycle.

The NXI arbiter is a fair arbiter in that the least recently granted device (XIOP or SPU) wins the bus grant if both devices request the NXP at the same time. The NXI tracks bus grant history with the state of *r.xiop0_mrg* (xiop0 is the most recently granted device). This state is set when XIOP0 (SPU) is granted the Pbus and remains set until XIOP1 (XIOP) is granted the bus.

Granting mastership of the NXP to either XIOP is a function of two states: *c.set_gnt* and *c.enabgnt*. The set grant state controls the initial granting of the NXP. Set_grant means that an XIOP is requesting the NXP and all the right conditions exist in order to grant the NXP. "All the right conditions" means that the NXP state machine is in the IDLE state (explained later) and the state machine has not just returned to the IDLE state, and the Port Arbiter is ready for another header from the NXI. Enable bus grant, *c.enabgnt*, is the holding term for NXP grants after the initial setting. Enable bus grant is true when the NXP state machine is not in the IDLE state and no error condition exists and the block transfer has not yet completed. When the block transfer completes or if an error occurs, enable bus grant goes false and the NXP grant is removed.

3.3.2.2NXI State Machine

The NXP state machine sequences through the standard fives state of the NXP protocol: IDLE, GRANT, HEADER, DECODE, and XFER. The NXP state machine starts out in the IDLE state, waiting for an XIOP request. The state machine can be scan initialized to the IDLE state or can be forced to the IDLE state by resetting the NIA.

When an XIOP bus request is received by the NIA and the NXP arbiter is ready to grant the NXP to an XIOP, the NXP state machine enters the GRANT state. The GRANT state gives the bus grant indication one NXP cycle to reach the targeted XIOP. The HEADER state is unconditionally entered on the NXP cycle following the GRANT state. During the HEADER state, the XIOP drives the NXP with its transfer header. At the end of the HEADER cycle, the NIA registers the header internal to the CDS gate arrays. Also at the end of the HEADER cycle, the NXI captures decoded bits and loads the address and long word counters.

If the XIOP is still requesting the NXP, the state machine next enters the DECODE state following the HEADER state. During the DECODE state, the NXP state machine examines the decode information and looks for header parity errors. If there are no parity errors and the header requires NXP transfers, the state machine enters the XFER state on the cycle following the DECODE state. The NXP state machine remains in the XFER state until the transfer is complete, or the XIOP aborts the transfer prior to completion, or a error condition is detected. The XIOP can abort a transfer by removing its bus request prior to completing the transfer. Error conditions which terminate a NXP transfer include write data parity errors, write data PCM errors or read data PCM errors. The NXP state machine is diagramed in Figure 3-8.

3.3.2.3Control Signal Outputs

The NXI sources control signals to the NXP channel and to different parts of the NIA. NXP channel control signals regulate and control NXP operations. NXP data validity is defined by two NXP channel control signals: *ia_xiop.rd_val* and *xiop_ia.wrt_val*. The NXI asserts *ia_xiop.rd_val* when the NXI is in a read transfer state and the read data register is full and does not contain a read data error. The XIOP asserts *xiop_ia.wrt_val* during write data transfers to the NXI. The NXI can regulate write data from the XIOP by de-asserting the *ia_xiop.mbuf_avl* control signal. When *ia_xiop.mbuf_avl* is false, the NXI will not accept write data from the XIOP. Therefore a successful write data transfer needs to have good data parity, the dval control from the XIOP is true and the mbav control from the NXI is true. See Figure 2-6 on page 11. The NXI will terminate an NXP transfer if an error is detected by asserting the *ia_xiop.bus_err* control signal.

The NXI sources control signals to the Port Arbitration logic, the Read Queue Arbitration logic and the Write Queue Arbitration logic. The Port Arbitration logic receives the bulk of these control signals including: *nxi.last_wr_xfr*, *nxi.wrt_xfr_abort*, *nxi.rd_xfr_abort*, *nxi.new_rd_hdr*, *nxi.toc_hdr1*, and *nxi.mbp_hdr*. The *nxi.last_wr_xfr* control tells the Port Arbitration logic that the last write transfer has been received by the NXI and to send this data to the Crossbar Interface. The *nxi.wrt_xfr_abort* signal tell the Port Arbitration logic that the XIOP has aborted its block transfer of write data early and to send write data in the WDQ to the Crossbar Interface. The *nxi.rd_xfr_abort* signal means that the XIOP has aborted a read operation and that the Port Arbitration logic needs to flush any previously requested read data and any pending read data requests.

The *nxi.new_rd_hdr*, *nxi.new_toc_hdr1*, and *nxi.mbp_hdr* signals mean that the NXI has received a new read header, or a new TOC header, or an MBP header respectfully. The NXI uses these control signals to notify the Port Arbitration logic that a memory request has been received and to service these requests as soon as possible.The NXI sends two control signals to the Read Queue Arbitration logic: *nxi.toc_hdr* and *nxi.new_toc_hdr2*. The Read Queue Arbitration logic uses *nxi.toc_hdr* to modify the Read Data Queue tail pointer address. The *nxi.new_toc_hdr2* signal is used to force a read access of the TOC from the Read Data Que. The Write Queue Arbitration logic also receives two control signals from the NXI: *nxi.val_wr_data*, and *nxi.wrt_xfr_abort*. The Write Queue Arbitration logic uses *nxi.val_wr_data* to select the NXI's write data register from the CDS gate arrays and store this write data into the Write Data Que. The *nxi.wrt_xfr_abort* is used to store an abort flag into the Write Data Que. This flag is used by the Crossbar Interface to know that the write transfer was aborted prior to completion.

Other control signals worthy of noting are *nxi.rd_hold* and *nxi.load_hdr*. These

signals are sent to the CDS gate arrays for data path control. The *nxi.rd_hold* signal controls the clock enable of the Pbus clocked read data register in the CDS gate arrays. This register holds the current read return data sourced by the NXI during read transfers. The *nxi.load_hdr* signal forces the loading of a new header into the header holding register internal to the CDS arrays.

3.3.3 Error Detection and the Soft Error Log

The NXI detects and reports several errors related to the NXP Channel. All of these errors are soft errors which are reported to the SPU for soft error log scanning. Upon detection of any of these transfer related errors, the NXI will terminate the transfer by asserting the bus error control signal, *ia_xiop.bus_err*, and by removing the XIOP's grant signal. Two of the errors are hard errors from the XIOPs. These XIOP hard errors are not transfer related errors. XIOP hard errors are reported to the SPU as soft errors. Table 3-2 defines the soft errors detected by the NXI as they appear in the Soft Error Log. The LSB of the Soft Error Log is an XIOP identifier for tagging soft errors to specific XIOPs on the NXP.

Table 3-5 NXI Soft Errors

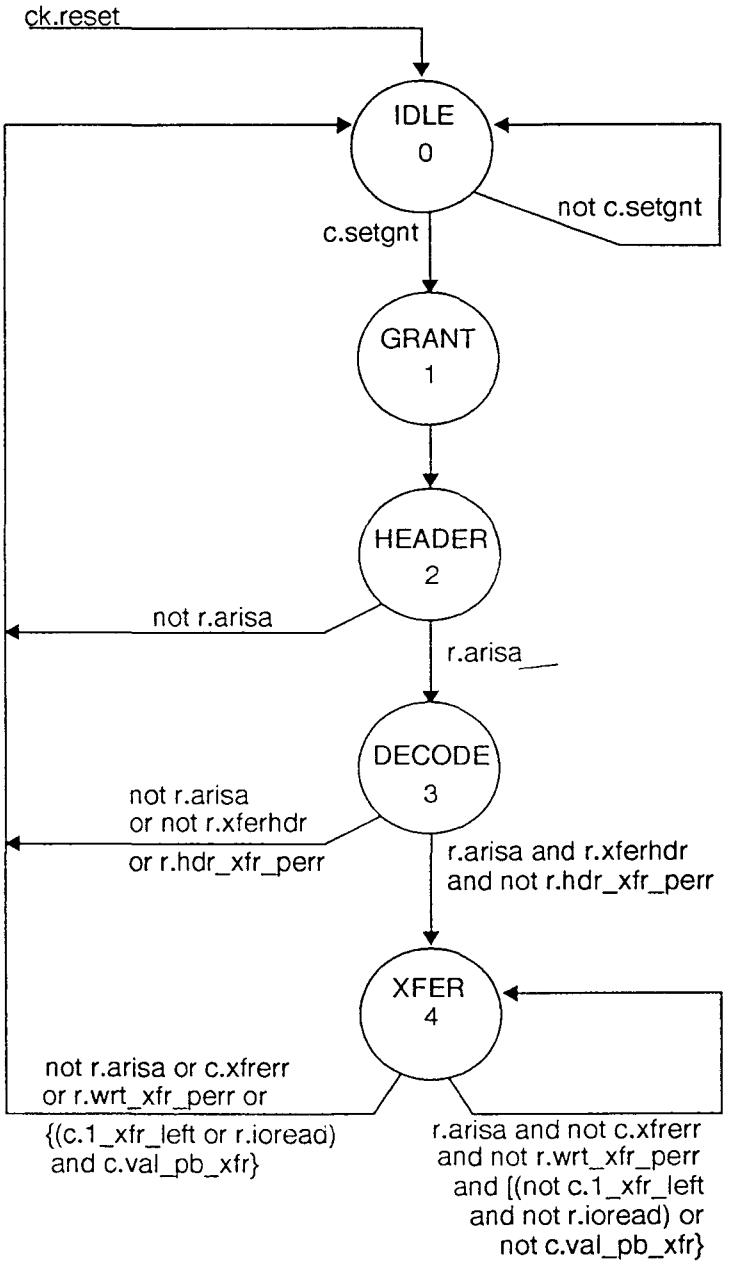
R.ILL_HDR	- Illegal header	— MSB
R.WRT_PCM	- Write transfer PCM error	
R.RD_PCM	- Read transfer PCM error	
R.WRT_PE	- Write transfer parity error	
R.HDR_PE	- Header transfer parity error	
R.XIOP0_ERR	- XIOP0 hard error	
R.XIOP1_ERR	- XIOP1 hard error	
R.XIOP0_MRG_SL	- XIOP0 most recent grant for the soft log	— LSB

Illegal headers are detected during the DECODE state and result in a bus error on the following cycle. Write transfer parity errors and header transfer parity errors are detected by the NXI with help from the CDS gate arrays. Each CDS array checks byte parity of the NXP data bus and reports parity errors to the NXI. Odd parity is expected. The NXI gathers all eight of these parity error indications from the CDS arrays and qualifies these indications with NXP state machine outputs.

PCM (physical configuration map) errors are detected at the NXI for write transfers and at the Crossbar Interface for read transfers. PCM checking verifies that memory is resident at the specified address. Installed memory may or may not reside in a contiguous address space. "Holes" in the memory space may exist depending upon the number of memory boards installed and the size of each memory board. The PCM defines what memory addresses are valid based upon the number and size of memory boards installed in the system. The PCM information is encoded and scan loaded into the NIA, *ck.pcm<31..0>*. This function is provided by the memory initialization program "mminit".

Up to eight memory boards can be installed in a system. The PCM state, *ck.pcm<31..0>*, describes which memory boards are installed and the size of each memory board installed See Figure 3-3. Valid memory board sizes are 64MB, 128MB, 256MB and 512MB. Bits 31..24 of *ck.pcm<31..0>* define the 512 MB memory boards that are installed in the system. Bits 23..16 define 256 MB boards, 15..8 define the 128 MB boards and 7..0 define 64 MB boards.

Figure 3-8 NXP State Machine



ck.reset	- board reset
c.1_xfr_left	- one transfer left
c.setgnt	- set bus grant
c.val_pb_xfr	- valid pbus transfer
r.arisa	- active request is still around
r.hdr_xfr_perr	- header transfer parity error
r.ioread	- io read header
r.wrt_xfr_perr	- write transfer parity error
r.xfrerr	- transfer error
r.xferhdr	- transfer header

NXP protocol dictates that all write data sent prior to the PCM violation must be written out to memory. Likewise, all read data requested prior to a PCM violation must be returned to the XIOP. For write operations, the transfer is terminated on the cycle following the NXP cycle in which write data destined to non-existent memory was sent to the NXI. For read operations, the transfer is terminated on the cycle in which the read data, which caused the PCM violation, would have been returned had the memory been resident in the system. In both cases the transfer is terminated by the NXI asserting the bus error indication to the XIOP and then removing the CCU grant on the following cycle.

3.3.4 NXI Interrupt Logic

The NXI also contains the NXP interrupt interface. The NXP interrupt bus is an eight bit wide bidirectional bus used to send and receive interrupts from the XIOP and SPU. The NIA forwards interrupts from the NXP interrupt bus to the NCU via the Crossbar Interface. Likewise, the NXI forwards interrupts from the NCU (via the Crossbar Interface) to the SPU and XIOP on the NXP interrupt bus.

3.3.4.1 NXI Interrupt Data Path Flow

Interrupts received from the NXP interrupt bus are captured at the NXI. The NXI then informs the Crossbar interface that an interrupt is ready and needs to be serviced, *nxi.nxp_intr_rdy*. The Crossbar interface (XBI) will then acknowledge the NXP interrupt ready, *xbi.nxp_intr_ack*, after then NXP interrupt vector has been written into the Write Data Que. The XBI then informs the Port Arbitration logic to queue an interrupt header in the port arbitration pipeline. From there the interrupt is sent as a Crossbar transfer to the NCU.

Interrupts can also be received by the XBI and sent to the NXI to be forwarded onto the NXP interrupt bus. Once the XBI receives an interrupt vector from the NCU, the XBI asserts *xbi.ncu_intr_rdy_x* to the NXI. The XBI then monitors the *nxi.nxp_intr_busy* signal to determine if the NXI has accepted the interrupt.If *nxi.nxp_intr_busy* is false, the XBI will remove its interrupt ready signal to the NXI and return to an idle state.

When the NXI sees *xbi.ncu_intr_rdy_x*, it captures the NCU interrupt vector in the NCU interrupt register. Then the NXI signals that it is busy to the XBI until the current NCU interrupt has been forwarded onto the NXP interrupt bus. The NXI then waits until it can become master of the NXP interrupt bus, and then drives the interrupt bus with the contents of the NCU interrupt register.

3.3.4.2 NXI Interrupt Bus Arbitration

Like the data bus, access to the interrupt bus is controlled by the NXI. The interrupt bus request and grant function is just like the data bus function described earlier. The XIOP and the SPU request access to the NXP interrupt bus by asserting their interrupt bus request signals. The NXI grants mastership of the interrupt bus by asserting the XIOP or SPU interrupt bus grant signal. The XIOP or SPU must assert its request for the complete duration of the interrupt transfer. If the request is removed early, the NXI will report this as a hard error.

The NXI becomes master of the interrupt bus when an interrupt is received from the NCU. The XBI asserts *xbi.ncu_intr_rdy_x* to the NXI after receiving an interrupt from the NCU. If neither XIOP is requesting the interrupt bus or the NXI has a higher priority than the requesting XIOPs, then the NXI grants itself the NXP

interrupt bus and proceeds to transfer the NCU's interrupt to the XIOPs.

The interrupt bus arbitration logic tracks who the most recently granted device was on the NXP interrupt bus (XIOP, SPU or the NXI itself). The state of *r.mrg_code<1..0>* defines the most recently granted device. If *r.mrg_code<0..1>* is equal to 00, then XIOP0 is most recent, 01 means XIOP1 is most recent, and 1x means the NXI is the most recently granted device. The state of *r.mrg_code* determines the priority of access to the NXP interrupt bus. For example, if *r.mrg_code* equals 00, (XIOP 0 is most recently granted device), then XIOP1 has the highest priority, NXI is next and XIOP0 is last. If *r.mrg_code* equals 01, then the NXI has the highest priority, followed by XIOP0, and then XIOP1. The *r.mrg_code* state is updated when the NXI grants the NXP interrupt bus to itself or to an XIOP.

The interrupt arbiter will override the priority logic if it is more efficient to do so. This situation can occur just after the NXI receives an interrupt from XIOP0. If XIOP1 is requesting the interrupt bus and the NXI has an interrupt from the NCU, then XIOP1 would normally receive the next interrupt bus grant. However, if the interrupt vector that was just received from XIOP0 has not been processed by the XBI, then the interrupt arbiter will grant the interrupt bus to itself instead of waiting to grant the bus to XIOP1. In this situation, the mrg code is not updated with the NXI code (1x) because the priority logic was over ridden. XIOP1 should retain the highest priority device even though it was not immediately granted the interrupt bus. XIOP1 will be granted the interrupt bus as soon as the XBI processes the correct interrupt vector from XIOP0.

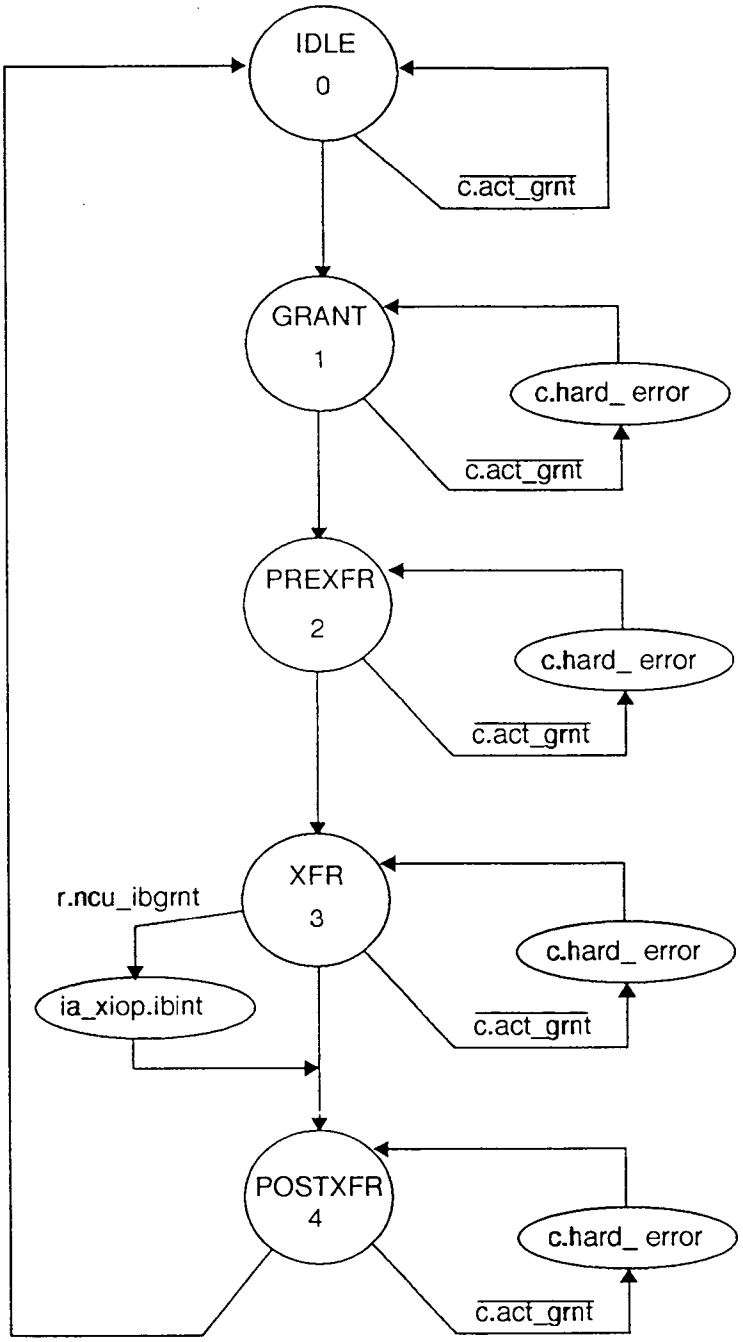
3.3.4.3NXP Interrupt State Machine

The NXP interrupt state machine is a five state machine. The interrupt state machine basically counts NXP cycles, beginning when the interrupt bus is granted and ending five NXP cycles later. The interrupt state machine starts out in the IDLE state. The start machine can be forced to the IDLE state by either scan initialization or by resetting the NIA. The state machine remains in the IDLE state until the arbitration logic grants the interrupt bus to either an XIOP or to the NXI. When the arbiter grants the interrupt bus, the interrupt state machine enters the GRANT state.

The GRANT state allows the grant indication to reach the targeted device (XIOP). If an XIOP has the bus grant, the state machine checks to make sure that the granted XIOP is still requesting the interrupt bus. If not, the state machine remains in the current state and a hard error is generated.

If the granted XIOP is still requesting the interrupt bus, *c.act_grnt*, the state machine enters the PREXFR state. This state allows the granted device one full NXP cycle to enable its bus drivers onto the interrupt bus and to prepare for the interrupt transfer on the following cycle. Again, the state machine checks to make sure the granted device is still requesting the interrupt bus. The state machine enters the XFR state on the following cycle. During this state, the targeted device is driving the interrupt bus with its interrupt vector. If the NXI has the bus grant, the interrupt interface will also assert the interrupt valid signal, *ia_xiop.ibint*. This signal is only asserted during the XFR state when the NXI has been granted the interrupt bus.

Figure 3-9 NXP Interrupt State Machine



c.act_grnt - active intrrupt bus grant
c.hard_error - hard error
r.ncu_ibgrnt - NCU interrrupt bus grant
ia_xiop.ibint - NXP interrupt bus valid

The state machine then enters the PSTXFR state on the following cycle. The interrupt vector is captured by the receiving devices at the beginning of this cycle. The granted device remains on the bus during this state and continues to drive its interrupt vector until the end of the cycle. The state machine checks to make sure the granted device is still asserting its request and then returns to the IDLE state on the following cycle. See Figure 3-9.

3.4 Channel Data Slice Gate Arrays

The Channel Data Slice (CDS) gate arrays contain the data path portion of the I/O Channel Interface. A total of eight CDS arrays are required to implement the complete data path. Each CDS array is an eight bit slice (plus one parity bit) of the 64 bit data path from each of the five I/O channels. The CDS arrays provide temporary buffering for read and write data, buffering for header data, multiplexing for write data and header data. The CDS arrays also perform arithmetic operations for header updates. The following two sections briefly describe the data path and arithmetic operations provided by the CDS gate arrays.

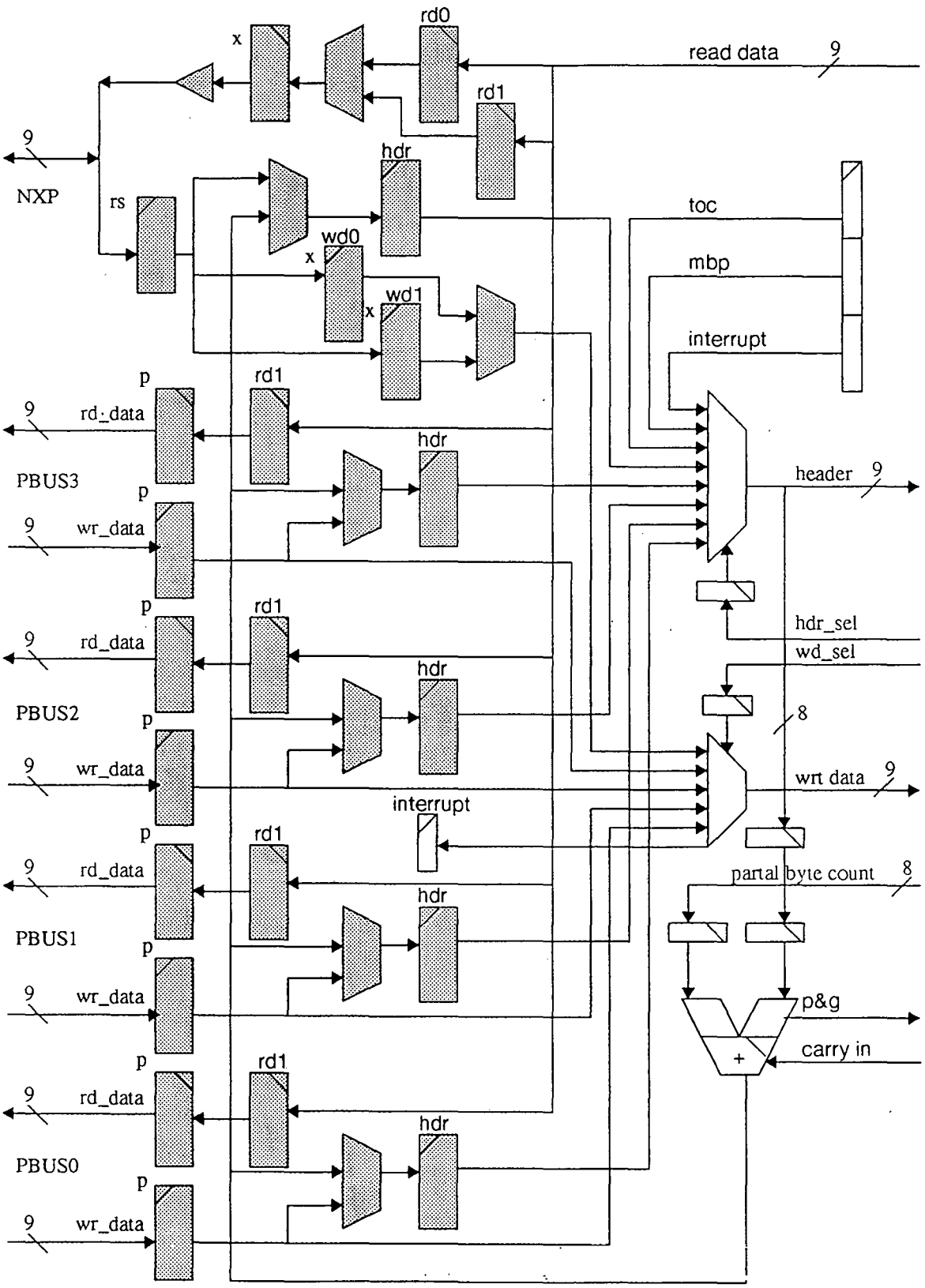
3.4.1 CDS Data Path

The data of each I/O channel interface consists of three sections: write data buffering, read data buffering, and a partial header holding register. Write data and header data multiplexing is also performed by the CDS array. The CDS array provides the data path interface between each I/O channel interface and the NIA Crossbar interface. A block diagram of the CDS array is shown in Figure 3-10. Registers which directly interface to a Pbus or NXP bus are clocked at the Pbus and NXP bus clock rates respectively (noted with a "P" or an "X" in the upper left hand corner). Therefore, three different clock rates are required internal to the CDS array: system clock, Pbus clock, and NXP clock. The slower clock rates (Pbus and NXP) are created internal to the CDS array by gating the system clock with externally supplied clock enables called clock "syncs". A complete description of how these clocks are aligned is presented in the Clocks and Scan chapter (section 8 on page 109) of this specification.

Write data buffering is provided at each I/O channel interface. The Pbus interfaces have one level of write data buffering. As write data is received from Pbus CCUs, it is immediately written into the Write Data Queue (WDQ). Each Pbus interface is guaranteed one write access to the WDQ during each Pbus cycle. The NXP interface has two write data buffers. The NXP interface accesses the WDQ during cycles in which there is no Pbus write data. Therefore, two NXP write data buffers give the NXP interface added buffering until a write cycle access to the WDQ becomes available. All write data transfers into the CDS array are checked for correct parity inside the CDS array. A 6:1 multiplexer function is implemented in the CDS array to select one of five I/O channel interface write data buffers or a scan loaded "interrupt" data register which is selected for interrupt vector writes to the WDQ.

Read data buffering is also provided at each I/O channel interface. Each Pbus interface has two staging registers to buffer read data. The first register is used to capture read data which is sourced from the Read Data Queue. This register receives a qualified system clock in which captured read data is held until the next Pbus clock edge. The second register receives a Pbus clock and sources read data to the Pbus interface. Once read data has been transferred from the first register (system clocked) to the second (Pbus clocked), the first register becomes available

Figure 3-10 CDS Array Data Path



to capture the next read data return. The NXP interface has two system clocked read data staging registers and one NXP clocked staging register. Like the Write Data Queue, the Read Data Queue is accessed for NXP read data during cycles in which no Pbus read data is available. Providing two system clocked staging registers allows for back-to-back Read Data Queue accesses for the NXP channel.

The partial header register holds the I/O channel's memory request (header) until the request can be forwarded to the NIA Crossbar Interface (XBI). For small requests, read or writes of 128 bytes or less, the whole request is forwarded to the XBI at one time. For memory requests greater than 128 bytes, the request is split into multiple 128 byte requests (256 bytes for NXP transfers). The partial header register holds the transfer type, the starting address and the remaining byte count of the memory request as it is sent to the XBI in multiple requests. After the last request is forwarded to the XBI, the partial header register becomes available to receive a new header from the I/O channel interface.

An 8:1 multiplexer function selects one of the five header buffers or one of three "special" registers. The special registers contain scan loaded headers for I/O transfers such as reading the Memory Base Pointer (MBP) or reading the Time of Century (TOC) counter. The third "special" register contains an interrupt header. The special registers are selected when an MBP or TOC read request is received or when the NIA receives an interrupt from a CCU.

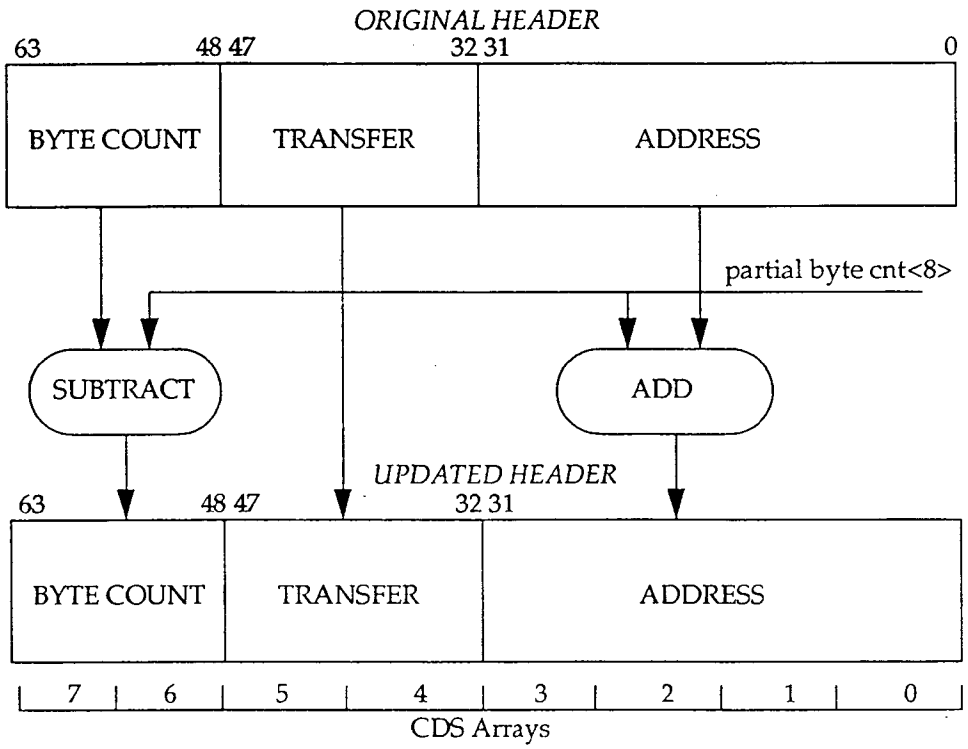
3.4.2 CDS Header Update Logic

The CDS arrays perform header updates as memory requests are sent to the Crossbar Interface. This function keeps the partial header register up to date and ready to source subsequent transfers to the XBI. Each array contains an eight bit adder to update headers. CDS Arrays 3-0 perform addition functions, arrays 7 & 6 provide subtraction functions and arrays 5 & 4 provide a pass through or "no adjustment" function. Each array uses a scan loaded slice ID to determine which operation to perform. As a header is sent off the CDS gate arrays, it is registered externally in the Port Arbitration logic and internally in the arrays. The Port Arbitration logic generates a nine bit partial byte count that is sent back to the CDS arrays. The partial byte count is registered internal to the CDS arrays and the header is staged into a second register. Then the staged header and the registered partial byte count are applied to the adder logic in each array.

CDS arrays 3-0 contain the address portion of the header. In these arrays, the partial byte count is added to the address. CDS arrays 7 and 6 contain the byte count of the header. Here, the partial byte count is subtracted from the original byte count. And finally, CDS arrays 5 and 4 contain the transfer field of the header which is passed unmodified through the adder logic.

The addition (or subtraction) is a two cycle event. In the first stage, group P and G terms are generated and sent off of each array to external carry look ahead logic. Internal to the arrays, the addition (or subtraction) is performed twice: once assuming a carry into the slice and once assuming no carry into the slice. The results are registered and sent to a 2-1 mux where the real carry in (from external look ahead) will select the appropriate result. From there, the result sent to a 2:1 mux in front of each header register where it can be selected and loaded into the header register on the following clock. An "update header" signal is generated by the Port Arbitration logic for each I/O channel. The CDS arrays use the update header signals to update the header registers at the appropriate time.

Figure 3-11 Header Updates



4 Write Data Queue and Arbitration

The Write Data Queue (WDQ) provides temporary buffering of write data between the I/O channels and system memory. The term "write data" is defined from the CCU's point of view. Therefore, write data is sourced by a CCU and is written into system memory. The Write Queue Arbitration Logic (WQA) controls write accesses to the WDQ.

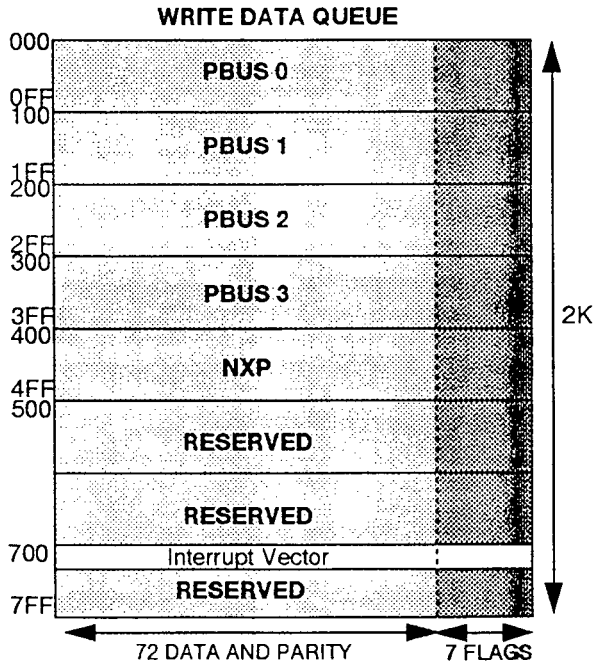
4.1 Write Data Queue

The Write Data Queue is implemented in 2Kx9 self-timed RAMS (called STRAMs). The WDQ contains 64 bits of write data, 8 data parity bits (odd parity), 6 flush/abort flags, and 1 flag parity bit (odd parity). Associated with the WDQ are various pointers and fill levels. The pointers define the current read and write address into the WDQ for each I/O channel interface. The fill level count the number of entries in the WDQ for each I/O channel interface. Reading or writing to the WDQ involves a two stage pipe. WDQ partitioning, pointers and fill levels, and read/write accesses are detailed in the following sections.

4.1.1 Partitioning

The WDQ is physically 2K entries deep. However, the WDQ is logically split into eight sections. Each I/O channel interface is assigned a section. Each WDQ section is 256 entries deep. Three of the WDQ sections are unused, except for one entry (address 700) which is reserved for interrupt vectors. Each WDQ entry has 64 data bits, 8 data parity bits, 6 flush flags and 1 flag parity bit. See Figure 4-1.

Figure 4-1 WDQ Partitioning



Each I/O channel has a WDQ flush flag bit. The flush flag bit is set when the NIA

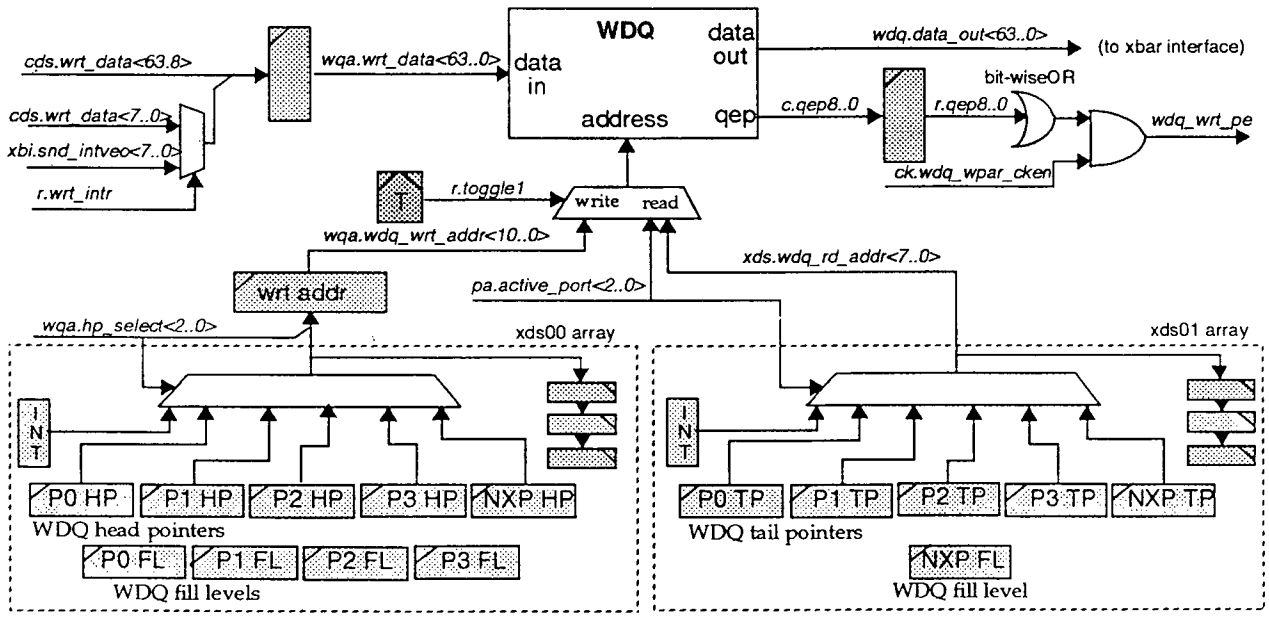
detects a write data parity error at the I/O channel interface, or when a CCU aborts a write transfer prior to completion. In either case, the appropriate flush flag is written into the WDQ. The data portion of the WDQ is not written under these conditions.One of the flags is the logical OR of all of the individual I/O channel flush flags. This "write abort" flag is used by the Crossbar interface logic to terminate the transfer at the crossbar interface. A flush flag parity bit is generated and written into the WDQ to parity protect flag data. Like the write data, the flag data has odd parity.

4.1.2 Pointers and Fill Levels

Each I/O channel has an eight bit read address and an eight bit write address to the WDQ. These addresses are actually "pointers" into the 2K deep WDQ STRAMs. The write addresses are called the head pointers and the read addresses are called the tail pointers. Each I/O channel also has a fill level counter to track the fullness of the channel's WDQ. The fill level counters are up/down counters. At system initialization time, the head pointers, tail pointers and fill level counters are set to zero.

The pointer and fill level definitions are as follows: The head pointer always "points" to the next location to write to in the WDQ. The tail pointer always "points" to the next location to read from in the WDQ. The fill level always tracks the number of un-read entries in the WDQ. The WDQ is a circular queue. When the end of the queue has been reached, the pointers will wrap around to the beginning of the queue.

Figure 4-2 WDQ Address and Data Path



The pointers and fill levels are controlled as follows: When write data is received by the NIA, the I/O channel informs the Write Queue Arbitration Logic (WQA). The WQA then schedules a write access to the WDQ. The current head pointer address is used as the write address. When the data is written, the I/O channel's head pointer is incremented to the next location. At the same time, the fill level counter is incremented to indicate that a new entry has been placed into the WDQ. Each time data is written into the WDQ, the head pointer and fill level are

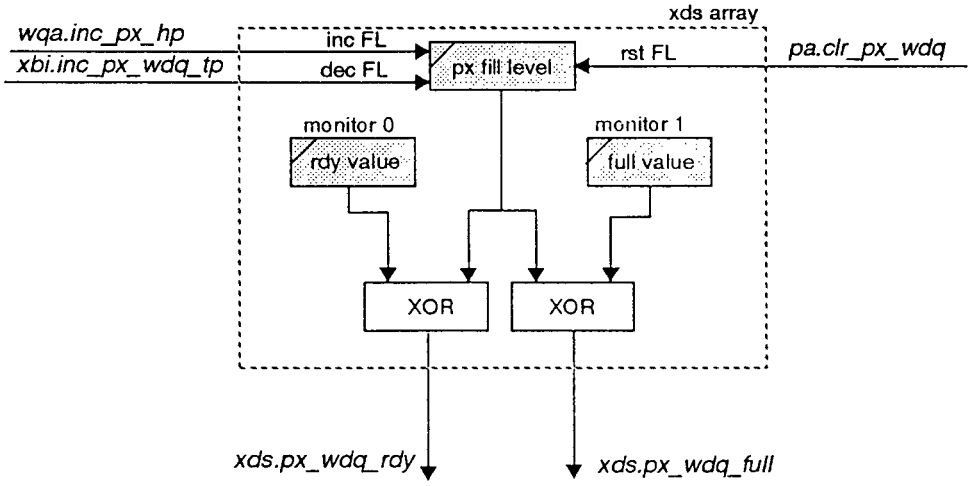
incremented.

At some point, the WDQ will be emptied as write data is forwarded to system memory. Data is read from the WDQ using the current tail pointer address. When data is read from the WDQ, the I/O channel's tail pointer is incremented to the next entry/location. At the same time, the fill level counter is decremented to indicate that an entry has been read from the WDQ. Each time data is read, the tail pointer is incremented and the fill level is decremented.

The fill level counters track the fullness of each I/O channel's WDQ. Occasionally, the fill level counters need to be reset to zero. This occurs whenever a flush flag bit is read from the WDQ. Reading the WDQ involves a two stage process (described later). Because of this two stage pipeline, the Xbar interface logic can request more reads than it should whenever the flush flag bit is set. The flush flag bits are sent to the Xbar Interface and the Port Arbitration logic. The Port Arbitration logic captures these bits in a register, checks the parity and returns a "clear wdq" signal, *pa.clr_px_wdq*, back to the fill level logic. The head and tail pointers are also reset to zero when the clear wdq is asserted.

The fill level counters are implemented in the XDS arrays. The count value itself is not directly available outside of the arrays (available through scan). However the rest of the NIA is not concerned with the exact value, but only certain "fill levels". For the WDQ, the NIA is uses two such ~~fill~~level indicators or "monitors" per I/O channel: a wdq ready monitor and a wdq full monitor. The ready monitor, *xds.px_wdq_rdy*, is used by the Port Arbitration logic and means that enough write data has been queued into WDQ to warrant sending a partial header to memory. "Enough data" is defined as 17 or more entries for the Pbus WDQs and 33 or more entries for the NXP WDQ. The full monitor, *xds.px_wdq_full*, is used by the I/O channel interfaces to hold off write data from a CCU. The full indication goes true when there are 223 or more entries in the WDQ (Pbus or NXP). See Figure 4-3.

Figure 4-3 WDQ Fill Level Monitors



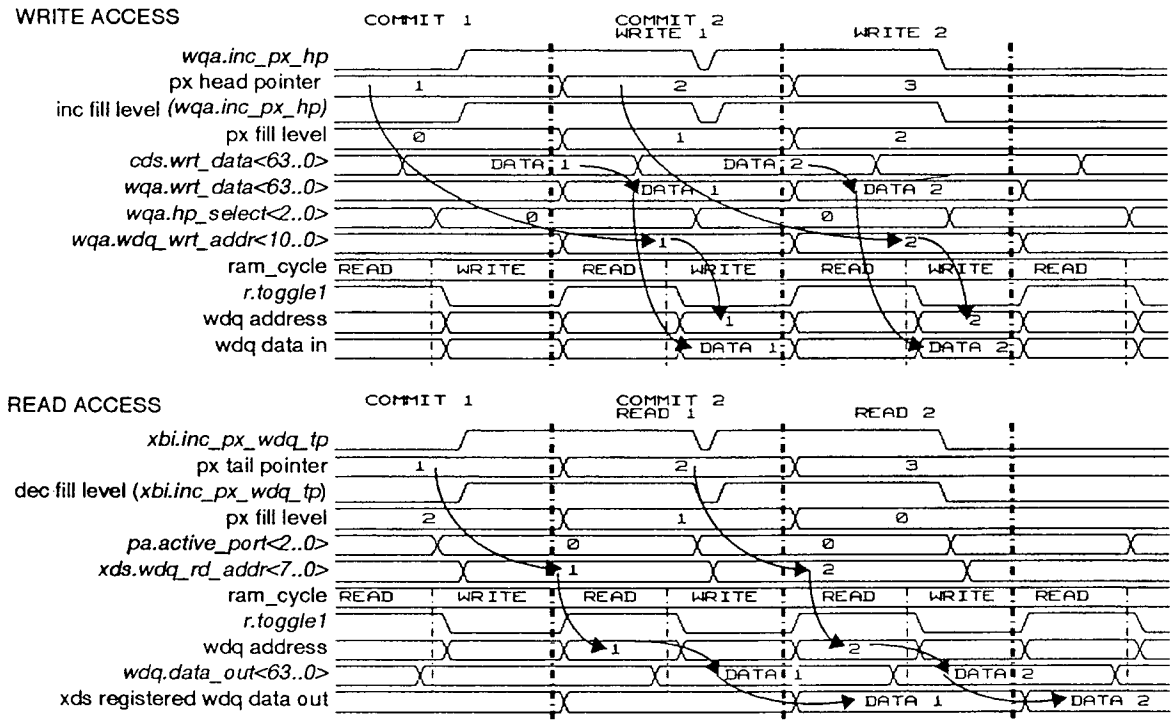
4.1.3 Read & Write Cycles

The STRAMs that make up the WDQ receive a 2x clock. This allows two WDQ accesses per system clock cycle: a read access and a write access. Reads occur in the

first half of the cycle and writes occur in the second half. A 2:1 multiplexer selects the appropriate address for each access.

The process of reading the WDQ involves a two stage pipe. The selection of the appropriate tail pointer is made the cycle prior to clocking the read address into the STRAMs. The three bit tail pointer select, *pa.active port<2..0>*, and the eight bit tail pointer, *xds.wdq_rd_addr<7..0>*, make up the eleven bit read address. The control signals to increment the tail pointer and fill level, *xbi.inc_px_wdq_tp*, are active during the first stage. The WDQ read access occurs during the first half of the second stage. Data accessed from the WDQ is stable midway in the system clock period and is captured in a system clocked register at the next system clock edge. The tail pointer and fill level counters are updated at the beginning of the second stage as a result of the control signals asserted during the previous cycle. See Figure 4-4.

Figure 4-4 WDQ Read and Write Timing



Writing into the WDQ also involves a two stage pipe. Write data from the I/O channel interface is captured in the CDS arrays. The WQA logic selects the appropriate head pointer and data for a WDQ write operation. The selection occurs in the first stage. The controls signals to increment the head pointer and fill level are also asserted in the first stage. Write data, *wqa.wrt_data<63..0>*, and address, *wqa.wdq_wrt_addr<10..0>*, are sourced from system clocked registers which mark the beginning of the second stage. Head pointers and fill levels are also updated at the beginning of the second stage. At the midway point of the second stage, the write data and address are clocked into the STRAMs where the data is written during the second half of the system clock period. Write data is parity checked during the write cycle. The STRAMs assert the QEP output if odd parity is not detected. When this occurs, a hard error, *wdq.wrt_pe*, is sourced. See Figure 4-2. Reads and writes to the same I/O channel interface can occur in the

same system clock. When this occurs, the head and tail pointers will both get incremented but the fill level will be left unchanged.

Interrupt vectors are also written into the WDQ during interrupt processing. When this occurs, the WQA selects the interrupt write data from the CDS arrays. The data from the CDS arrays contains an ID that says the interrupt is from the NIA. The interrupt vector itself is multiplexed into the least significant eight bits of the write data from the CDS arrays. Again see Figure 4-2.

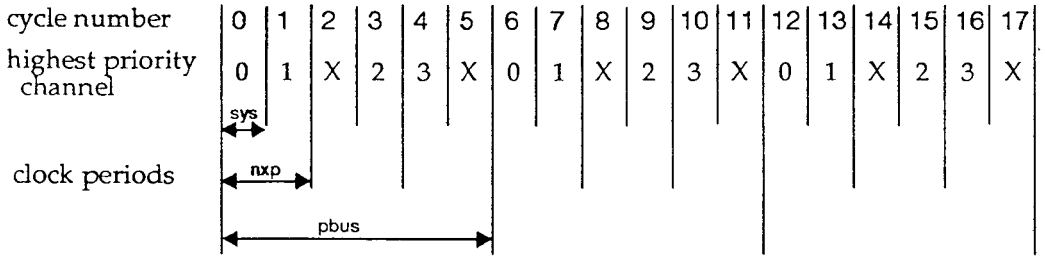
4.2 Arbitration

The Write Queue Arbitration logic (WQA) controls write accesses to the WDQ. Write data can be received by one or more I/O channel interfaces at the same time. The WQA schedules write accesses for each I/O channel interface according to predefined priorities. On the other hand, read access to the WDQ is controlled by the Xbar interface logic (XBI). Write data remains in the WDQ until the NIA is ready to forward it to system memory. At that time, the XBI controls which I/O channel's WDQ is accessed according to which header is being serviced. Also, the XBI controls the rate at which data is read from the WDQ. The rate depends on how often the Xbar/memory goes busy.

4.2.1 Priority Scheme

Write access to the WDQ is limited to 480 MB/sec (one write per system clock). Each Pbus can generate 80MB/sec (peak) of write data, and the NXP can supply write data at a peak rate of 240MB/sec. Obviously, if all I/O channels were delivering write data at their peak rate, the total write data bandwidth is greater than what the WDQ can sustain. If this situation occurs (believed to be rare), then the NIA will give priority to the Pbuses and hold off the NXP. This priority scheme is embedded in the WDQ arbitration logic.

Figure 4-5 WQA Access Priorities



The arbitration process takes place the cycle before the commit stage of WDQ access pipeline. On the NIA, each system clock cycle is identified by a number. This "cycle number" is used by the WQA as well as the Read Queue Arbitration logic and the Clock Generation logic. The cycle numbers start at 0 and go to 17 before returning to zero. The WQA uses the cycle number to determine which I/O channel has the highest priority to the WDQ. As seen in Figure 4-5, each Pbus channel is allocated one write access for each Pbus clock period. The NXP (X) channel is allocated the remaining cycles. Notice the NXP is only guaranteed a write cycle two out of every three NXP clock periods.

During the arbitration process, the WQA examines the priority as determined by

the cycle number and then looks to see if the highest priority channel has valid write data for the WDQ. If it does, the WQA selects that channel to have it's data written into the WDQ. If the highest priority channel is a Pbus interface and it does not have valid write data, the arbitration logic looks to see if the NXP channel has valid write data. If so, the WQA selects the NXP channel for a WDQ write cycle. The arbitration logic will give a Pbus channel's WDQ access to the NXP channel only when that Pbus channel does not have valid write data. The arbitration logic will not give a Pbus channel's access to another Pbus channel since each Pbus channel is guaranteed at least one WDQ access for each Pbus clock period.

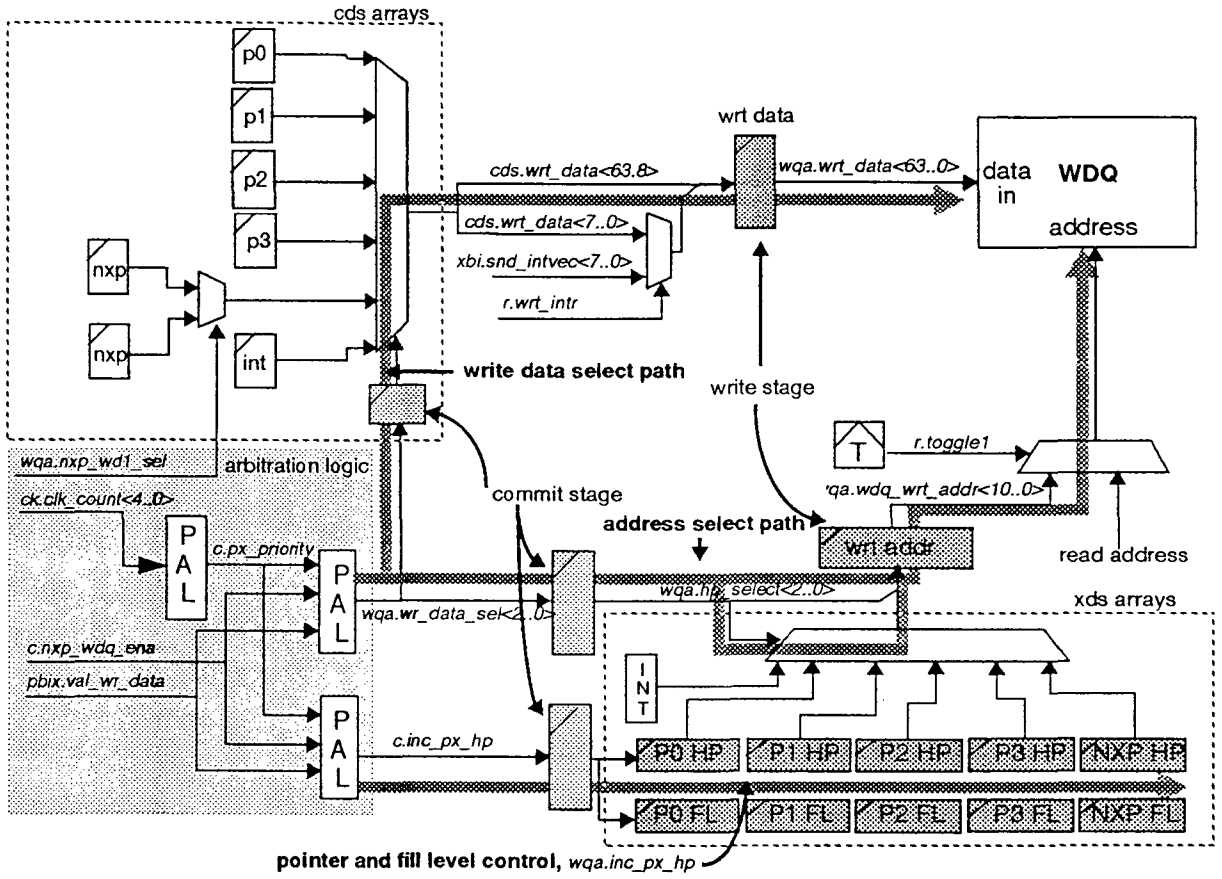
4.2.2 Arbitration Logic

The arbitration logic examines the cycle number, *ck.clk_count<4..0>*, and generates priority signals for each I/O channel, *c.pX_priority*. Only Pbus channel priority signals are used is the arbitration process. Valid write data flags, *pbiX.val_wr_data* and *c.nxp_wdq_ena*, are then examined and compared to the priority signals. If there is a match between a Pbus priority signal and a Pbus valid write data signal (meaning both are set), then the arbitration logic selects that Pbus channel. If none of the Pbus priority signals are set or if the valid write data flag of the highest priority Pbus is not set, then the arbitration logic looks to service the NXP. If the NXP valid write data flag is set under these conditions, the arbitration logic selects the NXP. The selection is encoded into write data selects, *wqa.wr_data_sela<2..0>* and *wqa.wr_data_selb<2..0>*, and sent to the CDS arrays. The arbitration logic also generates head pointer increment control signals, *c.inc_pX_hp*, for each channel. The write data selects are staged for one cycle and sent to the XDS00 array as WDQ head pointer selects, *wqa.hp_select<2..0>*. The head pointer increment controls are also staged for 1 clock cycle and sent to the XDS00 and XDS01 arrays as WDQ head pointer and fill level increment controls, *wqa.inc_pX_hp* and *wqa.inc_nxp_hp*. See Figure 4-6.

4.2.3 Miscellaneous Logic

The WQA block also contains some miscellaneous logic to control the write data path associated with the NXP channel. The NXP channel interface has two registers for buffering write data from the NXP bus. Pbus channel interfaces have only one write data buffer. The extra buffering in the NXP channel helps offset the NXP's lower priority, with respect to the Pbus channels, for writing to the WDQ. Logic in the WQA tracks the full state of the write data registers. Write errors are also kept for each write data register. The WQA generates clock enables, *wqa.nxp_wd0_ce** and *wqa.nxp_wd1_ce**, for the write data registers. The WQA also generates a write data select, *wqa.nxp_wd1_sela*, to select the appropriate write data register when writing to the WDQ.

Figure 4-6 WDQ Write Access Control



5 Read Data Queue and Arbitration

The Read Data Queue (RDQ) provides temporary buffering of read data between system memory and the I/O channels. The term "read data" is defined from the CCU's point of view. Therefore, read data is data which is requested by the CCU and is sourced by system memory. The Read Queue Arbitration logic (RQA) controls read access to the RDQ and stages write data and address to the RDQ.

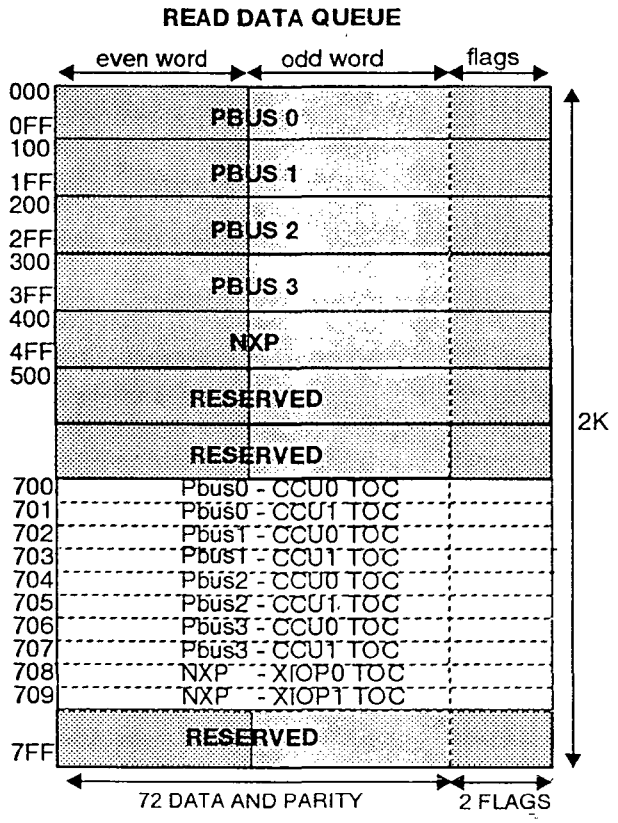
5.1 Read Data Queue

The Read Data Queue is implemented in 2Kx9 self-timed RAMs (called STRAMS). The RDQ contains 64 bits of read data, 8 data parity bits (odd parity), 1 read error bit and 1 read error parity bit (odd parity). Associated with the RDQ are various pointers and fill levels. The pointers define the current read and write address into the RDQ, and the fill levels count the number of entries in the RDQ. Reading or writing to the RDQ involves a two stage pipe.

5.1.1 Partitioning

The RDQ is physically 2K entries deep. However, the RDQ is logically split into eight sections. Each I/O channel interface is assigned a section. Each RDQ section is 256 entries deep. Three of the RDQ sections are unused, except for 10 entries which are reserved for Time of Century (TOC) reads. See Figure 5-1.

Figure 5-1 RDQ Partitioning



The RDQ is accesses two different ways for read versus write transfers. The RDQ is written 36 bits (32 data + 4 parity) at a time. The RDQ is split into even and odd words. Splitting the RDQ is this manner is necessary because memory may not always return even and odd words on the same cycle. In fact, the crossbar and memory are split into independent even and odd words. Therefore, the NIA must be capable of queueing even and odd words separately.

The RDQ is read 72 bits at a time since each I/O channel is 72 bits wide. Even in the case of partial long word requests, the NIA will read a long word from memory. Reading long words reduces logic on the NIA since good parity must be provided across all 64 data bits of the channel for each data return.

A portion of the RDQ is reserved for Time of Century (TOC) values. Each CCU has a dedicated TOC location in the RDQ. The TOC is a 64 bit value. A CCU can read the TOC, but may want to read it 8,16 or 32 bits at a time. The CCU can address the TOC with byte, half word, or long word addressing. When the first byte of the TOC is requested, the NIA will make a read request for the TOC and then load the requesting CCU's TOC location in the RDQ with the value return from the NCU. The NIA will return all 8 bytes of the TOC to the requesting CCU, regardless of the number of bytes the CCU requested. All subsequent TOC reads that do not include the first byte will cause the NIA to return the current TOC value stored in the RDQ. If the NIA read the TOC from the NCU for each partial request, the TOC value could change between CCU TOC reads and would give the CCU an inconsistent value.

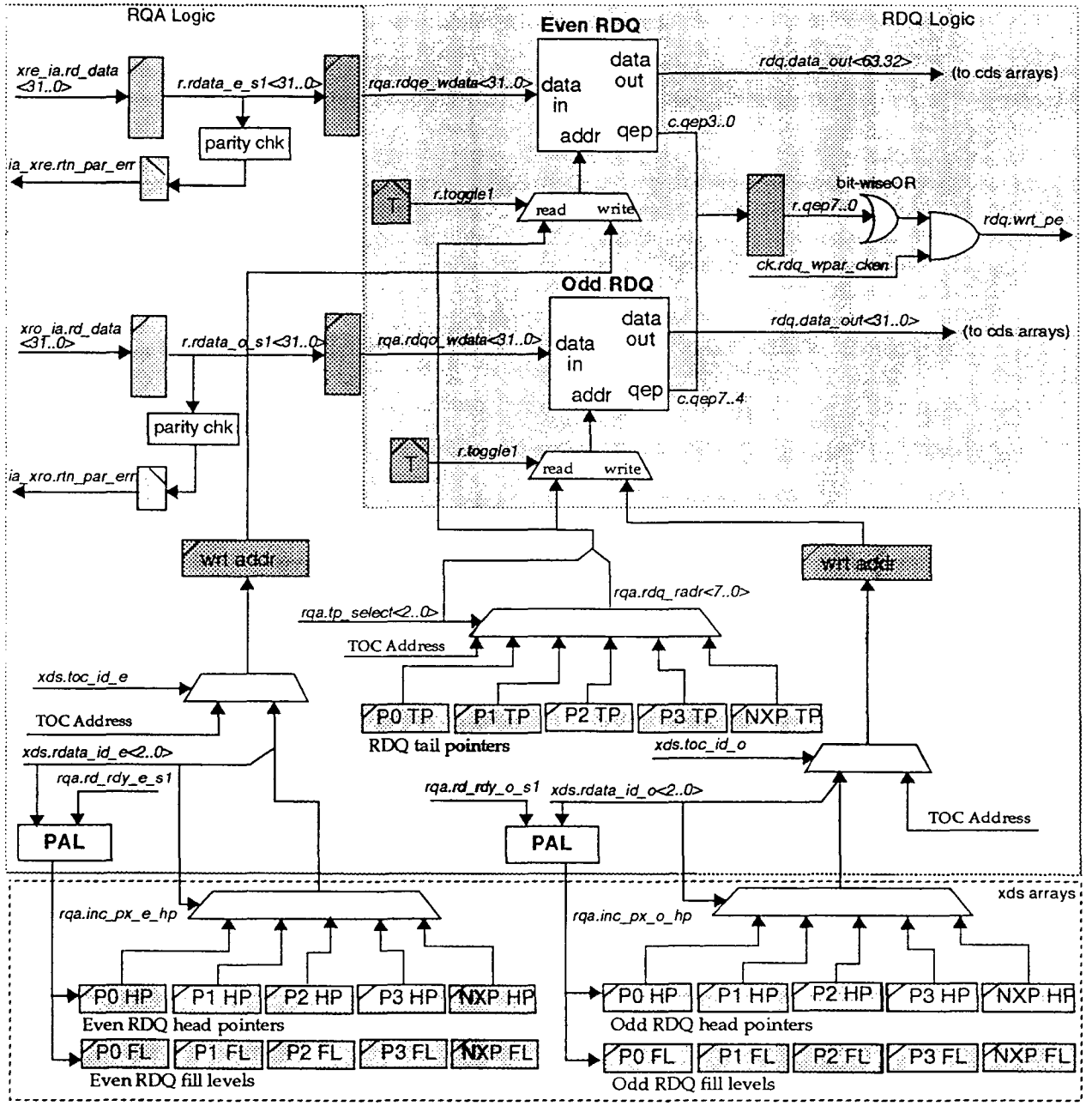
A read error flag is included in the RDQ. The read error flag is set when a read request results in a PCM error. The NIA's Crossbar Interface checks for PCM errors during read requests. If a PCM error is detected, the XDS arrays queue an error indication into the read return FIFO. When the error indication reaches the tail of the FIFO (output stage), the PCM error is written into the RDQ as an error flag. The flag bits (error bit plus a parity bit) are written into the even side of the RDQ. When the error flag is read from the RDQ, it is stored as a read error for that particular I/O channel. When the read error reaches what would normally be the return transfer stage, the I/O channel sources a bus error to the CCU instead of read data.

5.1.2 Pointers and Fill Levels

Each I/O channel has one eight bit read address and two eight bit write addresses to the RDQ. These addresses are actually "pointers" into the 2K deep RDQ STRAMs. The write addresses are called head pointers and the read addresses are called tail pointers. Each I/O channel has two RDQ head pointers: even and odd. However, only one RDQ tail pointer per I/O channel is required since even and odd data are read at the same time. Each channel also has two fill level counters to track the fullness of a channel's even and odd RDQ. The fill level counters are up and down counters. At system initialization time, the head pointers, tail pointers and fill level counters are set to zero.

The pointer and fill level counters are defined as follows: The head pointer always "points" to the next location to write to in the RDQ. The tail pointer always "points" to the next location to read from in the RDQ. The fill level always tracks the number of un-read entries in the RDQ. The RDQ is a circular queue. When the end of the queue has been reached, the pointers will wrap around to the beginning of the queue.

Figure 5-2 RDQ Address and Data Path



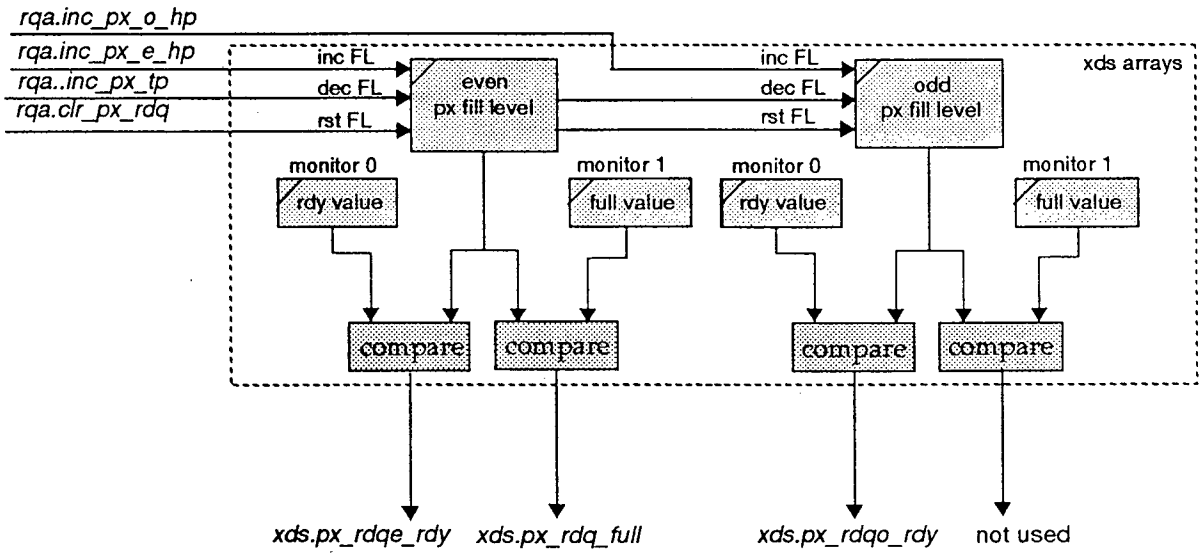
The pointers and fill levels are controlled as follows: When read data returns from the crossbar/memory, the NIA checks it for good parity, *r.rdata_e/o_s1<31..0>*. Expected parity is odd. The read data is staged an additional cycle before being written into the RDQ. The read return FIFO in the XDS arrays supplies two channel IDs: one for the even side read data, *xds.rdata_id_e<2..0*, and one for the odd side read data, *xds.rdata_id_o_<2..0>*. The channel IDs identify which channel the read data word belongs to. The RQA logic uses the channel IDs to select and increment the appropriate RDQ head pointers and fill levels. The even side channel ID selects the appropriate even side head pointer and the odd side channel ID selects the odd side head pointer. Incrementing the head pointers moves the pointers to the next location to write to in the RDQ. The fill levels (even and odd side) are incremented

to indicate that read data has been queued into the RDQ. Note that no arbitration takes place for writing data into the RDQ since only one even and one odd word of read data can return to the NIA each cycle. Also note that even and odd read data may return to the NIA on the same cycle but destined for different I/O channels. Both even and odd words can be written into the RDQ at the same time even though they are written to different addresses and destine for different I/O channels.

Once a channel has data written into both even and odd sides of the RDQ, the RQA logic schedules a read access. Both even and odd sides are read at the same time using the I/O channel's current tail pointer address. Note that each I/O channel needs only one tail pointer address since both even and odd sides are read at the same time. When the RDQ is read, the I/O channel's tail pointer is incremented to the next location in preparation for a subsequent read. At the same time, the channel's even and odd side fill level counters are decremented to indicate that a data entry has been read from the queue. Each time data is read from the RDQ, the channel's tail pointer is incremented and its fill levels are decremented.

The fill level counters track the fullness of each I/O channel's RDQ. Occasionally, the fill level counters need to be reset to zero. This occurs whenever a CCU aborts a read transfer (by removing it's request) prior to receiving all of the read data it requested. The NIA must "flush" all of the previously requested read data that is queued in the RDQ. When this flush occurs, *rqa.clr_px_rdq*, the RDQ head pointer, tail pointers and fill levels for that channel are reset to zero.

Figure 5-3 RDQ Fill Level Monitors



The fill level counters are implemented in the XDS arrays, The count value itself is not directly available outside of the arrays (available through scan). However, the rest of the NIA is not concerned with the exact value, but only certain "fill levels". For the RDQ, the NIA uses three such fill level indicators or "monitors" per I/O channel: an RDQ even ready monitor, an RDQ odd ready monitor, and an RDQ full monitor. The ready monitors, *xds.px_rdqe_rdy* and *xds.px_rdqo_rdy*, are used by the RQA logic to schedule read accesses to the RDQ. The even side monitor is

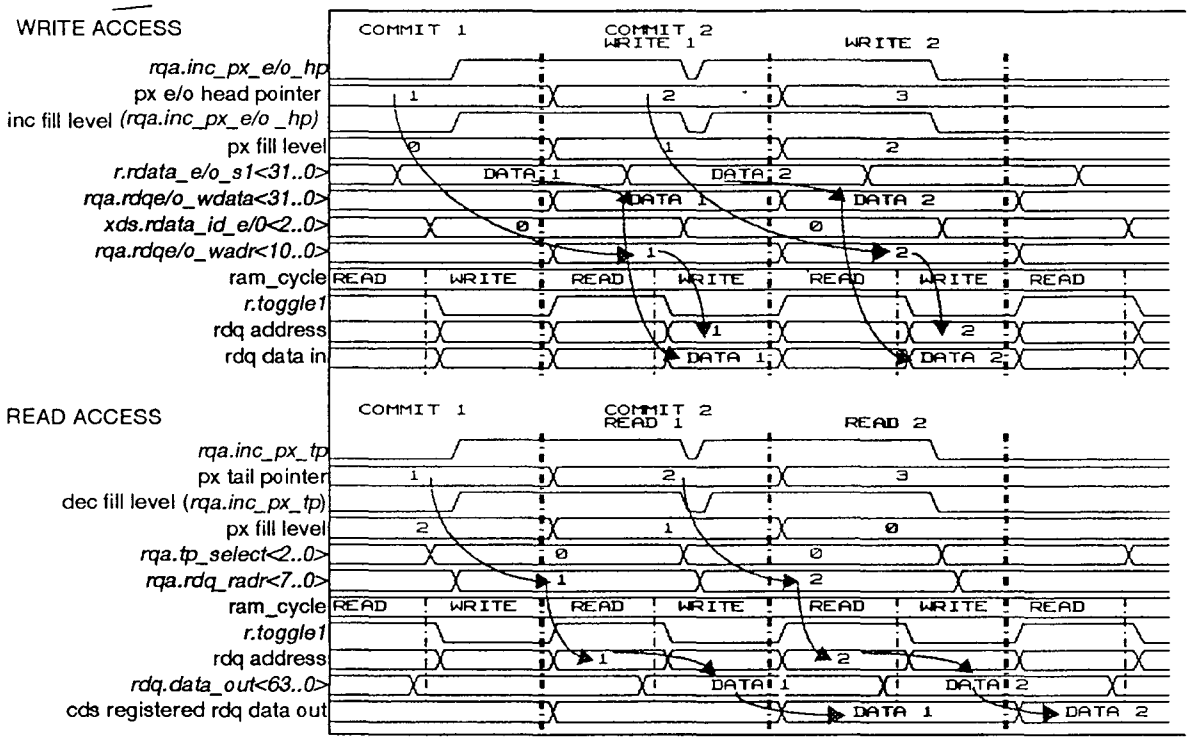
active when one or more entries are written into the even side RDQ, likewise for the odd side monitor. Both even and odd side monitors must be active before the RQA logic will schedule a read from the RDQ.

The full monitor, *xds.px_rdq_full*, is used by the Port Arbitration logic to prevent overflow of the RDQ. When the RDQ fill level (even side) reaches 223 or more, the full monitor goes true. When this occurs, the Port Arbitration logic will cease to send read data requests to the NIA's crossbar interface for that particular I/O channel. This condition holds until the full monitor goes false. Only the even side fill level counter is used to determine the fullness of a channel's RDQ. Since both even and odd words are requested for all memory reads, the NIA knows that both even and odd sides of the RDQ will eventually reach the same fill level. Therefore only one full monitor is required to prevent overflow of the RDQ.

5.1.3 Read And Write Cycles

The STRAMs that make up the RDQ receive a 2x clock. This allows two RDQ accesses per system clock cycle: a read access and a write access. Reads occur in the first half of the cycle and writes occur in the second half. A 2:1 multiplexer selects the appropriate address for each access.

Figure 5-4 RDQ Read and Write Timing



The process of reading the RDQ involves a two stage pipe. The selection of the appropriate tail pointer is made by the RQA logic prior to clocking the read address into the STRAMs. The three bit tail pointer select, *rqa.tp_select<2..0>*, and the eight bit tail pointer, *rqa.rdq_radr<7..0>*, make up the eleven bit read address. The control signals to increment the tail pointer and fill level, *xbi.inc_px_rdq_tp*, are active during the first (commit) stage. The RDQ read access occurs during the first half of the second (read) stage. Data accessed from the RDQ is stable midway in the system clock period and is captured by the CDS gate arrays in a system clocked

register at the next system clock edge. The tail pointer and fill level counters are updated at the beginning of the second stage as a result of the control signals asserted during the previous cycle. See Figure 5-4

Writing into the RDQ also involves a two stage pipe. Read data from the crossbar/memory is captured by the RQA block. The head of the read return FIFO in the XDS arrays defines the appropriate head pointer for a RDQ write operation. Head pointer selection occurs in the first (commit) stage. The controls signals to increment the head pointer and fill level are also asserted in the first stage. Write data, *rqa.rdqe_wdata<31..0>* and *rqa.rdqo_wdata<31..0>*, and address, *rqa.rdqe_wadr<10..0>* and *rqa.rdqo_wadr<10..0>*, are sourced from system clocked registers which mark the beginning of the second (write) stage. Head pointers and fill levels are also updated at the beginning of the second stage. At the midway point of the second stage, the write data and address are clocked into the STRAMs where the data is written during the second half of the system clock period. Write data is parity checked during the write cycle. The STRAMs assert the QEP output if odd parity is not detected. When this occurs, a hard error, *rdq.wrt_pe*, is sourced. See Figure 5-2. Reads and writes to the same I/O channel interface can occur in the same system clock. When this occurs, the head and tail pointers will both get incremented but the fill level will be left unchanged.

5.2 Read Queue Arbitration

The Read Queue Arbitration logic (RQA) controls the reading of the RDQ. The RDQ may contain valid read data for all five I/O channels at one time. The RQA schedules read access for each I/O channel interface according to predefined priorities. Write accesses to the RDQ are also controlled by the RQA, but there is no arbitration process to schedule a write. Writes to the RDQ occur when the NIA receives data from the crossbar/memory subsystem. The crossbar/memory subsystem can only return one even word and one odd word of data per cycle. Since the NIA is capable of writing one even and one odd word of data into the RDQ each cycle, there is no need to arbitrate writes.

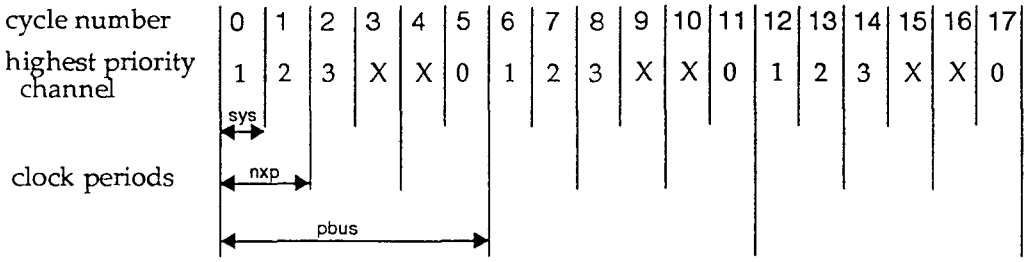
5.2.1 Priority Scheme

Read access to the RDQ is limited to 480 MB/sec, or one read per system clock. Each Pbus is capable of receiving 80 MB/sec (peak) of read return data, and the NXP can accept read data at the peak rate of 240 MB/sec. Obviously, if all I/O channels were waiting for read data to be returned to the CCUs, the total read data bandwidth out of the NIA is greater than what the RDQ (and memory port) can support. If this situation occurs, then the NIA will give priority to the Pbus channels and will starve the NXP channel. The goal of the arbitration logic is to keep all of the Pbus channels filled with read data and then give the NXP channel all of the remaining RDQ bandwidth. This priority scheme is embedded in the RDQ arbitration logic.

The arbitration process takes place the cycle before the commit stage of the RDQ access pipeline. On the NIA, each system clock cycle is identified by a number. This "cycle number" is used by the RQA as well as the Write Queue Arbitration Logic and the Clock Generation logic. The cycle numbers start at 0 and go to 17 before returning to zero. The RQA uses the cycle number to determine which I/O channel has the highest priority to the RDQ. As seen in Figure 5-5, each Pbus channel is allocated one read access for each Pbus clock period. The NXP (X) channel is allocated the remaining cycles. Notice the NXP is only guaranteed a

read cycle two out of every three NXP clock periods.

Figure 5-5 RQA Access Priorities



During the arbitration process, the RQA examines the priority as determined by the cycle number and then looks to see if the highest priority channel has valid read data in the RDQ. A channel has valid read data when both the even and odd side RDQ ready monitors are asserted, *xds.px_rdqe_rdy* and *xds.px_rdqo_rdy*. If the highest priority channel has valid read data, the RQA selects that channel for a read access to the RDQ. If the highest priority does not have valid read data in the RDQ and that channel is a Pbus channel, the arbitration logic looks to see if the NXP channel has valid read data in the RDQ. If so, the RQA selects the NXP channel for a read access to the RDQ. The RQA will not give a Pbus channel access to another Pbus channel since each Pbus channel is guaranteed one read access per Pbus cycle.

5.2.2 Arbitration Logic

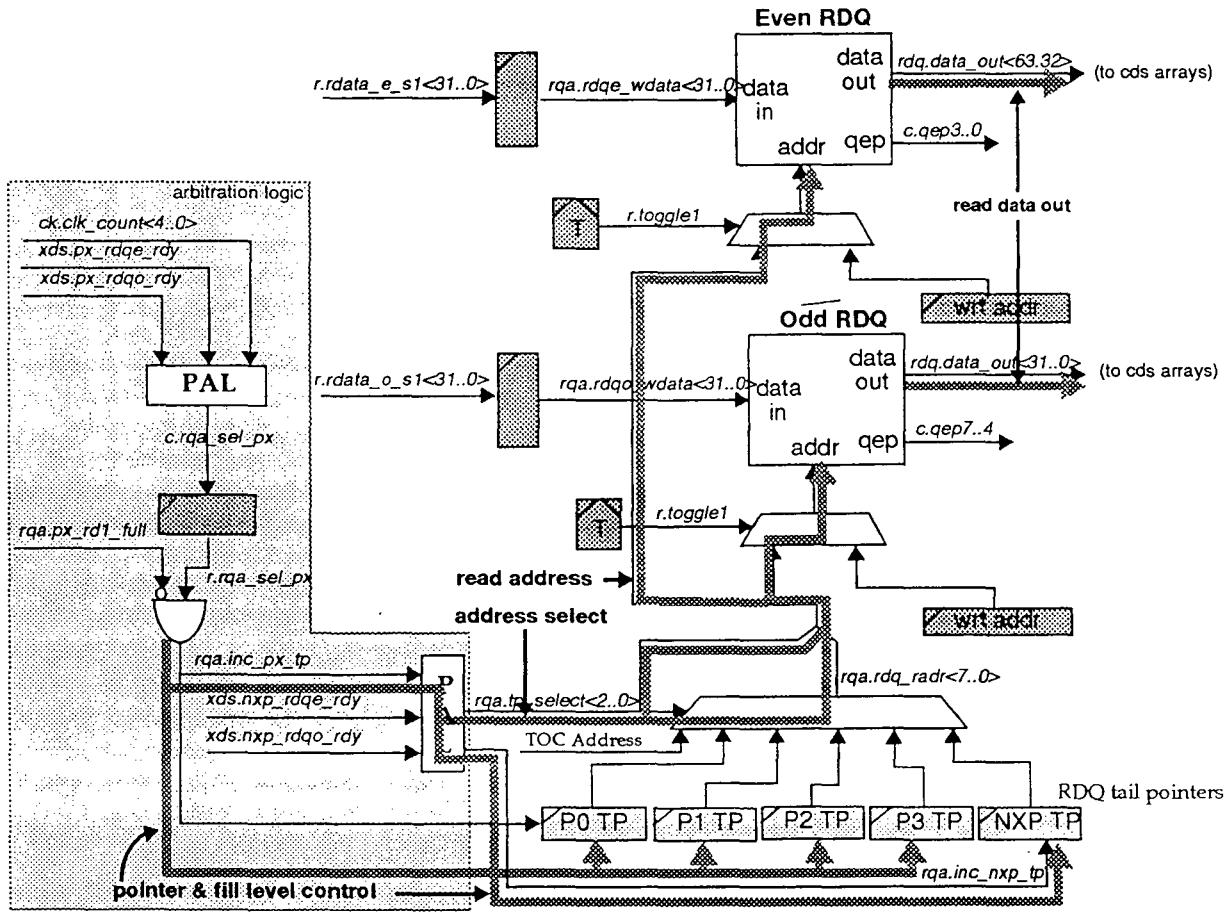
The arbitration logic begins by examining the clock cycle number, *ck.clk_count<4..0>*, and RDQ ready signals, *xds.px_rdqe_rdy* and *xds.px_rdqo_rdy*. The clock cycle number defines which channel with the highest priority to the RDQ. The RDQ ready signals are set when a channel has valid read data in the RDQ. The ready signals and the clock count are compared in PALs to generate select signals,*c.rqa_sel_px*. Select signals are generated for Pbus channels only and are captured in a register which marks the beginning of the commit stage. The registered selects, *r.rqa_sel_px*, are qualified by "read data register full" signals, *rqa.px_rd1_full*. These "full" signals tell the arbitration logic that the I/O channel interface currently has read data in it's staging register and can not accept more read data from the RDQ. See Figure 5-6. This condition would occur if the CCU could not accept read data and did not assert it's channel buffer available signal during the previous Pbus cycle.

If the selected Pbus channel's staging register is not full, then the arbitration logic asserts *rqa.inc_px_tp* to increment the channel's tail pointer and decrement the channel's fill level. The increment tail pointer signals are also encoded to generate a tail pointer select, *rqa.tp_select<2..0>*, which is used to select the appropriate tail pointer address. The tail pointer select and the selected tail pointer address are combined to form a read address to both even and odd RDQs. The RDQ address is then clocked into the even and odd side STRAMs to mark the beginning of the read stage. Midway into the read stage, the read data is valid from the STRAMs and is sent to the CDS arrays to be captured in a read data staging register.

Accessing the RDQ for the NXP channel is slightly different than that described for Pbus channels above. If the clock cycle defines the NXP to have the highest priority, or if a Pbus channel has priority but does not have valid data in the RDQ

then none of the *c.rqa_sel_px* signals will be asserted. Thus, none of the *rqa.inc_px_tp* signals will be asserted. When this condition occurs, the PAL that encodes the tail pointer selects examines the RDQ ready signals for the NXP channel. If the NXP channel has valid read data in the RDQ, the arbitration logic will select the NXP channel's tail pointer and assert the tail pointer increment control, *rqa.inc_nxp_tp*. If the highest priority channel is a Pbus channel and it has valid data in the RDQ, then that channel's *r.req_sel_px* signal will be asserted. However, if the selected channel's read data staging register is full, then it's increment tail pointer signal will not be asserted. Therefore the arbitration logic will again look to see if it can service an NXP channel read.

Figure 5-6 RDQ Read Access Control



6 Port Arbitration

The Port Arbitration (PA) logic is a large control logic block that arbitrates I/O channel access to memory. The arbitration process involves selecting the appropriate I/O channel header in the CDS Arrays, and staging this header and appropriate states through the arbitration pipeline. Partial byte count generation is performed in the PA block. The partial byte count logic generates the byte count that is used by the crossbar interface on the NIA. This allows the NIA to split up large block transfers from I/O channels into groups of smaller transfers. Several miscellaneous functions such as a fetch counter and CDS Array carry look-ahead logic are also included in the PA block

6.1 Arbitration Process

The basic theory behind the Port Arbitration logic is to allocate the NIA's memory bandwidth to all I/O channels in a manner that keeps them all busy. This is accomplished by giving each I/O channel small bursts of memory access at a time. For example, if all I/O channels have open read requests to memory, the PA logic will give each channel 16 (for Pbuses) or 32 (for NXP) memory transfers in a round robin fashion. When PA logic comes back to the first channel in the round robin, the channel will be just finishing up with the previous reads and ready to accept more. This approach reduces the average latency each I/O channel will see and prevents transfer "bubbles" on the I/O channel bus. However, it is possible for I/O channel bandwidth demand to exceed the NIA's memory bandwidth supply. When this occurs, the Port Arbitration logic gives the Pbus I/O channels higher priority over the NXP channel. This priority scheme will maintain each Pbus channel's peak transfer rate of 80MB/sec and will reduce the NXP's bandwidth from 240MB/sec peak to a sustained 160 MB/sec.

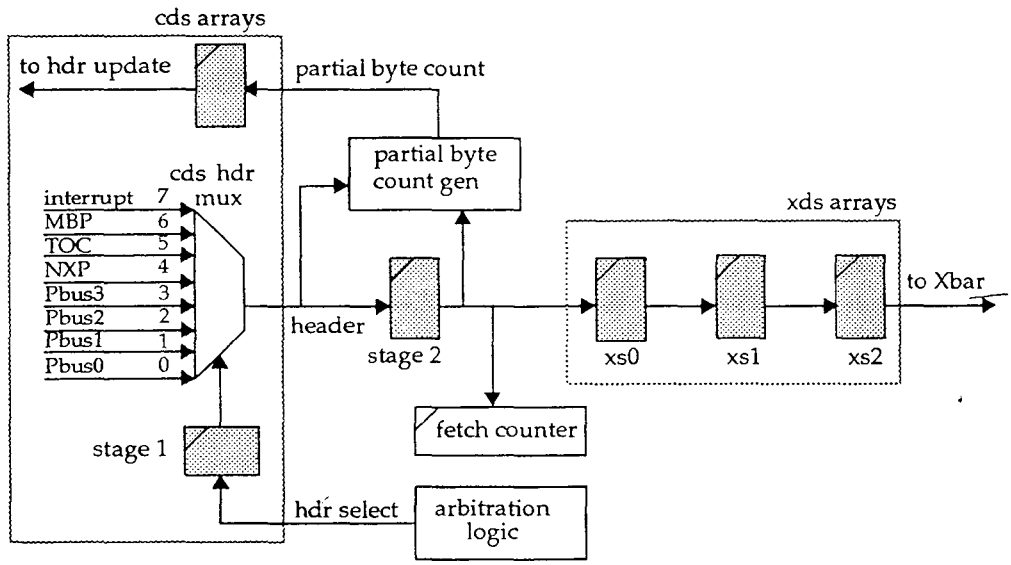
Header selection and arbitration pipeline control are the main functions provided by the PA block. Headers are selected from the CDS Arrays and are pipelined to the crossbar interface. Headers from different I/O channels can reside in different stages of the port arbitration pipeline at the same time. The PA block tracks the movement of these headers through the PA stages and through the crossbar interface stages as well.

6.1.1 Pipeline Stages and Control

The arbitration process is pipelined in order to make timing and to maximize the usage of the NIA's memory bandwidth. The process of selecting an I/O channel header, getting it off of the CDS arrays, and generating a partial byte count and a fetch count on it's way to the XDS arrays takes too long to perform in one cycle. Therefore, there are two stages in the PA block plus three stages in the XDS arrays for a total of five stages. All of these stages have qualified clocks which allow multiple headers from different I/O channels to reside in each of the five stages at one time. This allows the NIA to begin transfers to memory from one I/O channel header the cycle following the completion of another I/O channel header. Committing header transfers to the arbitration pipeline in this manner makes better use of the memory bandwidth. However, there are a couple of tradeoffs: first, the design is more complicated and second, interrupts much wait for previously committed (to the arbitration pipeline) header transfers to finish before they are sent to the NCU.

The PA pipeline begins with the arbitration logic. This function takes all of one system clock cycle to generate the header selects and have them registered on the CDS arrays. The header select register in the CDS arrays represents the first stage (stage 1) of the PA pipeline. From the header select register, a header is selected out of the CDS arrays and registered into the second stage (stage 2). The stage 2 header register is used to generate a partial byte count and to load a fetch count register in the PA block. The header register output is also sent to the XDS arrays where it is captured in an XS0 staging register. XS0, XS1 and XS2 are stages internal to XDS arrays and are described in the Crossbar Interface chapter (section 7 on page 97). See Figure 6-1.

Figure 6-1 Port Arbitration Pipeline Stages



Each I/O channel has valid flags that are staged with the header transfer through the arbitration pipeline. The valid flags are used to track and identify header transfers on their way to the crossbar interface. The validity flags are encoded at two different points in the pipeline: at XS0 and at XS1. At the XS0 stage, the valid flags are encoded into an "active port" ID, *pa.active_port<2..0>*. This port ID is used to select the appropriate Write Data Queue tail pointer located in the XDS01 gate array. At the XS1 stage, the valid flags are again encoded into signals named *pa.channel_id<2..0>*. These signals are used by the XDS arrays as the I/O channel ID that is written into the read data return FIFO for all memory read transfers. The FIFO contents are used to track outstanding read requests and to identify returning read data from memory.

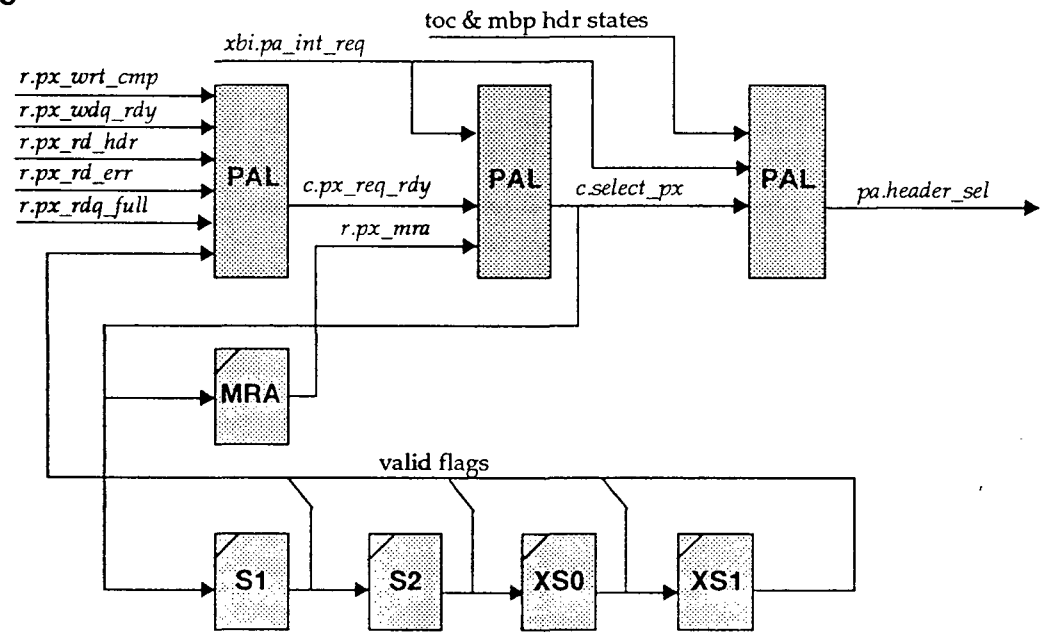
The valid flags are used to identify and clear header transfers in the pipeline when transfers are aborted or terminated at the I/O channel. When such a condition occurs, the appropriate staging register (stage 1,2, XS0,1,2) is cleared. The valid flags are also used to tell the I/O channel interfaces that it is OK to grant the channel to a requesting device. The valid flags are NANDed together to create "ready for header" control signals, *pa.rdy_4_p3..0_hdr* and *pa.rdy_4_nxp_hdr*. These control signals tell each I/O channel interface that any previous transfer from the channel has cleared the arbitration pipeline, that there are no pending transfers waiting to be pipelined, and there are no error conditions waiting to be cleared.

The stage 1 and stage 2 clock enables are generated by the PA logic. XS0, 1 and 2 clock enables are generated by the Crossbar Interface block (XBI). Stage 1 and 2 registers freerun by default and are held when the stage is full (valid) and the XS0 clock enable from the XBI block is false.

6.1.2 Arbitration and Header Selection

The arbitration and header selection process is a time critical function that takes place in three PAL gate levels. Basically, the PA logic contains several states for each I/O channel and uses these states to create "request ready" signals, *c.px_req_rdy*, at the first PAL level. These request ready signals are then used by arbitration logic to generate individual "select this channel" signals, *c.select_px*, at the second PAL level. The select signals are then encoded to generate the header selects, *pa.header_sel<2..0>*, used by the CDS arrays. The header selects are generated at the third PAL gate level. The three PAL levels are shown in Figure 6-2 and are described in more detail below.

Figure 6-2 Three Levels of Header Selection



An I/O channel is considered for arbitration to memory when it's request ready signal is true. Setting the request ready signal is based upon the occurrence of one or more conditions: the channel's write complete state is true, the channel's read header state is true, or the channel's Write Data Queue ready signal is true. The write complete state, *r.px_wrt_cmp*,[1] means that the last write data transfer has been received by the I/O channel interface or that the CCU has aborted it's write transfer early. In either case, the I/O channel needs to have it's Write Data Queue (WDQ) emptied by forwarding the write transfer to the XBI. A write transfer can also be forwarded to the XBI when the WDQ ready signal is true, *r.px_wdq_rdy*. This signal is sourced by the one of the XDS arrays and means that the WDQ has received at least 17 or more long word transfers.[2] The read header state,

1. note: px stands for port X and represents pbus0,1,2, & 3 and nxp ports. There are five wrt compiete states, one for each i/o channel. This notation is used throughout this chapter.
2. 17 is used for pbus channels. 33 is used for the nxp channel.

r.px_rd_hdr, is set when the I/O channel interface receives a read header transfer. Read transfers are processed as soon as they are received by the I/O channel interface.

Given that at least one of the conditions described above is true, then up to three other conditions must be true in order for an I/O channel's request ready, *c.px_req_rdy*, signal to asserted. First the I/O channel must not already have a transfer pending in the arbitration pipeline. Valid flags for each stage are examined to check this condition. Given that no other transfers for an I/O channel are pending in the arbitration pipeline, then either *r.px_wrt_cmp* or *r.px_wdq_rdy* will set the request ready state for that channel. Two additional states are checked for read transfers. The Read Data Queue full, *r.px_rdq_full*, state must not be true, and the read error, *r.px_rd_err*, state must also not be true. The Read Data Queue full state indicates that the I/O channel's Read Data Queue is full and can not accept any more read data until the CCU begins accepting read data from the NIA. The read error state is set when a PCM error has been detected for a particular I/O channel read header.

When a request ready signal is set, the arbitration phase (second PAL level) is entered. The output of this phase is a "select this channel" signal, *c.select_px*, for each I/O channel. Only one of these channel selects will be true at any one time. The channel selects are registered into stage 1 and stage 2 registers and are used as the channel valid flags for each pipeline stage.

Channel selection involves examining the request ready signals and the "most recently arbitrated" states, *r.px_mra*<2..0>. The *r.px_mra*<2..0> states define the last I/O channel that had a transfer sent to stage 1 of the arbitration pipeline. The channel with it's "MRA" state set has the lowest priority in the arbitration process. The MRA states also imply a priority level for all of the other channels. See Table 6-1.

Table 6-1 Channel Priority Schedule

MRA / priority	0	1	2	3	4
	Pbus0	Pbus1	Pbus2	Pbus3	NXP
highest	Pbus1	Pbus2	Pbus3	NXP	Pbus0
	Pbus2	Pbus3	NXP	Pbus0	Pbus1
	Pbus3	NXP	Pbus0	Pbus1	Pbus2
	NXP	Pbus0	Pbus1	Pbus2	Pbus3
lowest	Pbus0	Pbus1	Pbus2	Pbus3	NXP

If only one request ready signal is set, then the MRA based priority schedule does not come into play in selecting that channel. If two or more request readys are set, then Table 6-1 defines which *c.select_px* signal gets asserted. However, if an interrupt request is received, *xbi.pa_int_req*, the arbitration logic overrides the priority schedule and selects the interrupt. In this situation, none of the channel selects will be true and the header select phase (third level PAL) will select the header interrupt.

The header select phase encodes the channel selects into header selects. The header selects, *pa.header_sela*<2..0> and *pa.header_selb*<2..0, are sent to the eight CDS arrays where they are registered into stage 1 of the arbitration pipeline. Version "a" is sent to CDS arrays 0-3 and "b" is sent to CDS arrays 4-7. Both versions are the same.

The header selects point to one of the five I/O channel headers in the CDS arrays or to one of three "special" registers. These "special" registers contain the addresses for the Memory Base Pointer (MBP), the Time of Century Counter (TOC), and the Trap register. The MBP and TOC were previously implemented in I/O address space for C1 and C2 series machines. In the C3800 series, these two entities were moved into NCU control registers. In order to access these control registers, the NIA must translate the I/O space addresses used in C2 and C1 into NCU control register addresses. This address translation is achieved by selecting the scan loaded MBP or TOC header registers in the CDS arrays. The header select phase examines TOC header, *pbix.toc_hdr* and *nxi.toc_hdr*, and MBP header, *pbix.mbp_hdr* and *nxi.mbp_hdr*, states from the I/O channel interfaces in order to know when to select these "special" header registers. For example, if *c.select_p2* was set and *pbi2.toc_hdr* was also set, then the header select would point to the TOC header instead of the Pbus2 header.

Table 6-2 Header Select Mapping

select	header
0	Pbus 0
1	Pbus 1
2	Pbus 2
3	Pbus 3
4	NXP
5	TOC
6	MBP
7	interrupt

The trap register is also located in the NCU and is accessed by the NIA in order to forward interrupts from the I/O channels to the rest of the system. The trap register address is scan loaded into the interrupt header register located in the CDS arrays.

6.2 Partial Byte Count Generation

The partial byte count generation logic splits large block transfers (greater than 128 bytes) into smaller transfers. Splitting up large block transfers has two main benefits. First, splitting large transfers reduces the average read data latency incurred by a CCU. This occurs when the NIA is servicing large block transfers from two or more I/O channels at the same time. A CCU does not have to wait for the completion of another CCU's transfer before getting serviced. The CCU will only wait for the amount of time required to service a portion of another CCU's transfer. Second, splitting large transfers makes efficient use of the NIA's memory bandwidth. The crossbar is designed in such a way as to give a processor port (IA or SP) priority for 16 consecutive requests before giving priority to another port. This "burst mode" feature of the crossbar is exploited by the NIA when transfers are split into blocks of 16 long words (128 bytes).

6.2.1 Theory of Operation

The partial byte count (PBC) generation logic examines the byte count field and the long word offset of the stage 2 header register in calculating the partial byte count. The long word offset (LWO) is the least significant three bits of the header address, *pa.hdr_address<2..0>*, The PBC generation logic also examines two different "maximum transfer size" values: 128 bytes for Pbus transfers and 256 bytes for NXP transfers. These values are scan initialized at power up. The goal of the PBC generation logic is to generate as large a byte count as possible up to the predefined maximum transfer size. The PBC generation logic will generate a maximum partial byte count when the header byte count is greater than the maximum transfer size and the LWO is zero. If the maximum partial byte count is not generated, then one of two values is generated. The PBC could be set to the header byte count which can could happen when the header byte count is less than the maximum transfer size. Or, the PBC could be set to the maximum transfer size minus the LWO. See Figure 6-3. This second case occurs when a large, unaligned (on a long word boundary) transfer is processed. The theory here is to split large block transfers at long word boundaries in order to reduce the number of partial word accesses. Partial word accesses will tie up (make busy) the entire word aligned bank in memory. Reducing the number of partial word accesses results in efficient use of memory bandwidth.

Figure 6-3 Pseudo Code for PBC Generation

```
If ((header byte count + long word offset) > max transfer size)
then
     partial byte count = max transfer size - long word offset
else
     partial byte count = header byte count;
```

6.2.2 Implementation

The PBC generation logic implements the pseudo code shown in Figure 6-3. PBC generation is a time critical function that spreads across 1 1/2 system clock cycles. PBC generation is split into three different arithmetic parts. First, the header byte count is added to the long word offset. Second, the long word offset is subtracted from the max transfer size. And third, the max transfer size is compared to the sum of the header byte count plus the long word offset.The time critical path is getting the result from the addition function to the comparison function in order to select the desired partial byte count value. Details are given below and in Figure 6-4.

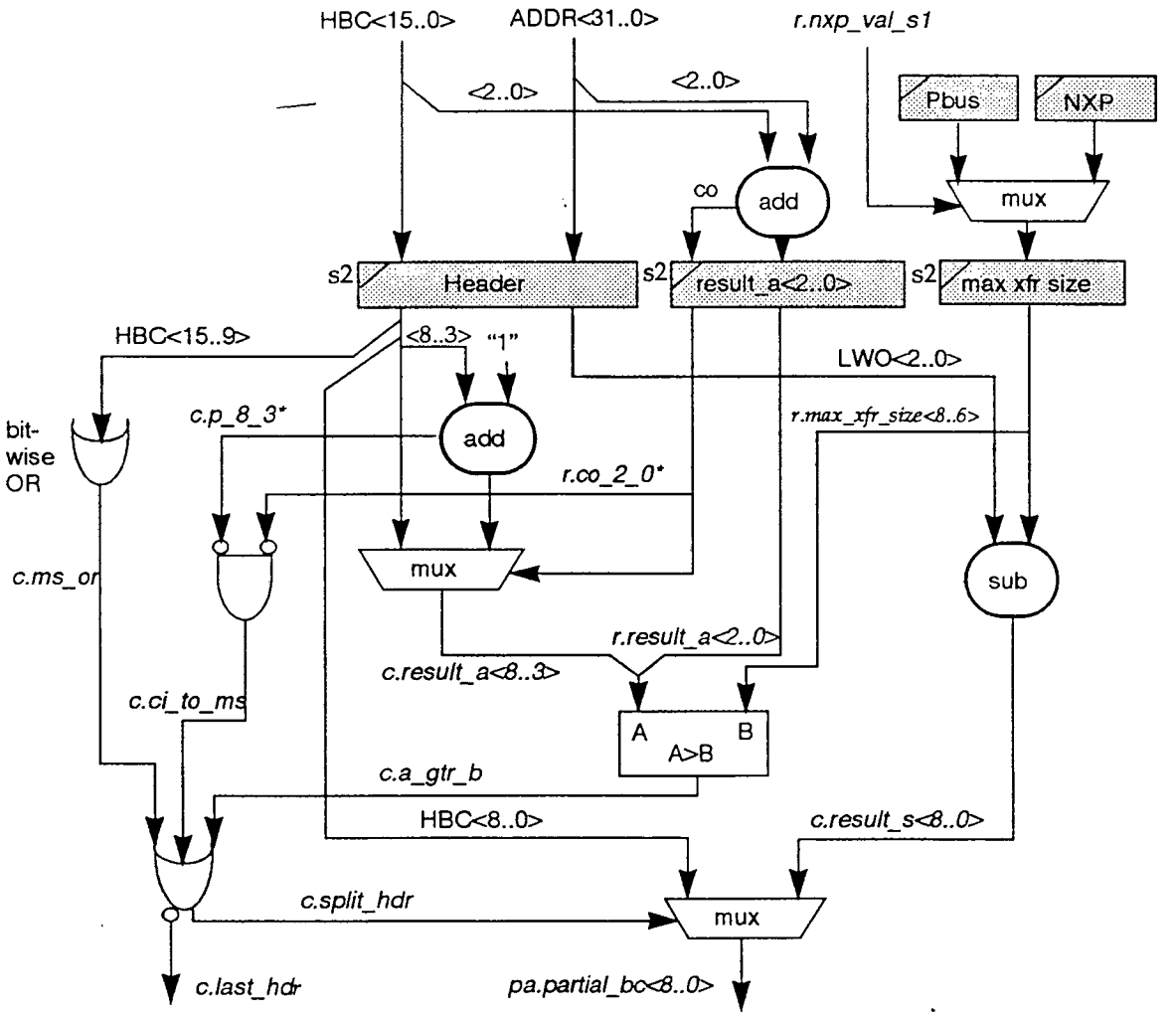
The addition function involves adding a 3 bit value, long word offset, to a 16 bit value, header byte count (HBC). Implementing a 16 bit adder is too costly in terms of time, so the addition function is split into three smaller sub-functions geared for use with the comparison function: a bit-wise ORing of the HBC bits 15..9, incrementing HBC bits 8..3 by one, and adding HBC<2..0> to LWO<2..0>. Looking ahead at the comparison function, we want to know if the result from the addition function is greater than the max transfer size, a 9 bit value. Therefore, only the least significant nine bits from the addition function are really needed. If any of the upper 7 bits from the addition function are set, then we know right away that the HBC is greater than the max transfer size. With the comparison function in mind, the addition function was reduced to 9 bits instead of 16 bits.

The addition function is split into two parts. The LWO<2..0> and the HBC<2..0>

are added together ahead of the stage 2 header register with the 3 bit result, *r.result_a<2..0>*, and the carryout, *r.co_2_0**, being captured in a stage 2 register. Then HBC<8..3> is incremented in bit position 3 using the stage 2 header register output. The *r.co_2_0** signal is used to select either HBC<8..3> or the incremented HBC<8..3> as the addition result for bits 8..3, *c.result_a<8..3>*. The concatenation of *c.result_a<8..3>* and *r.result_a<2..0>* is used in the comparison function with the max transfer size. See Figure 6-4.

Comparing result_a<8..0> to the max transfer size is only part of the overall comparison function. HBC<15..9> are bit-wise ORed to determine if the HBC by itself is greater than the max transfer size. Also, the 9 bit addition function could produce a carry out and thus result in the addition being greater than the 9 bit max transfer size. This condition is checked by ANDing the carry out from the lower 3 bit addition with a group propagate term, *c.p_8_3**, from the HBC<8..3> increment function. This "carry in to most significant" bits signal, *c.ci_to_ms*, is in turn used to determine if the addition function result is greater than the max transfer size.

Figure 6-4 PBC Generation



The 9 bit result from the addition function is compared to the max transfer size. Only the most significant three bits of the max transfer size, *r.max_xfr_size<8..6>*,

are kept in hardware. This allows values from 64 to 448 bytes for the max transfer size. The NIA is scan initialized with a 128 byte max transfer size for Pbus transfers and a 256 byte max transfer size for NXP transfers. The comparison is implemented with an ECLiPS magnitude comparator. The output from this comparator, *c.a_gtr_b*, is true when the result from the addition function is larger than the max transfer size. This output is ORed with the *c.ci_to_ms* and the *c.ms_or* signals to create the *c.split_hdr* signal. The split header signal is means that the addition function result is larger than then maximum allowed transfer size and when asserted, selects the result from the subtraction function, *c.result_s<8..0>*, as the partial byte count. If *c.split_hdr* is not asserted, then the remaining HBC<8..0> is selected as the partial byte count. When this occurs, the last "partial" of the block transfer has been reached. The PA asserts a "last header" signal, *c.last_hdr*, which is registered and later used to clear the *r.px_rd_hdr* and *r.px_wrt_cmp* states in the arbitration logic.

6.3 Miscellaneous Functions

The Port Arbitration block contains several miscellaneous functions. These functions include a fetch counter, Write Data Queue flush flag parity checking, and carry look-ahead logic for the header updates performed by the CDS gate arrays. Some of these functions logically belong in the PA block and others, well, really don't!

6.3.1 Fetch Counter

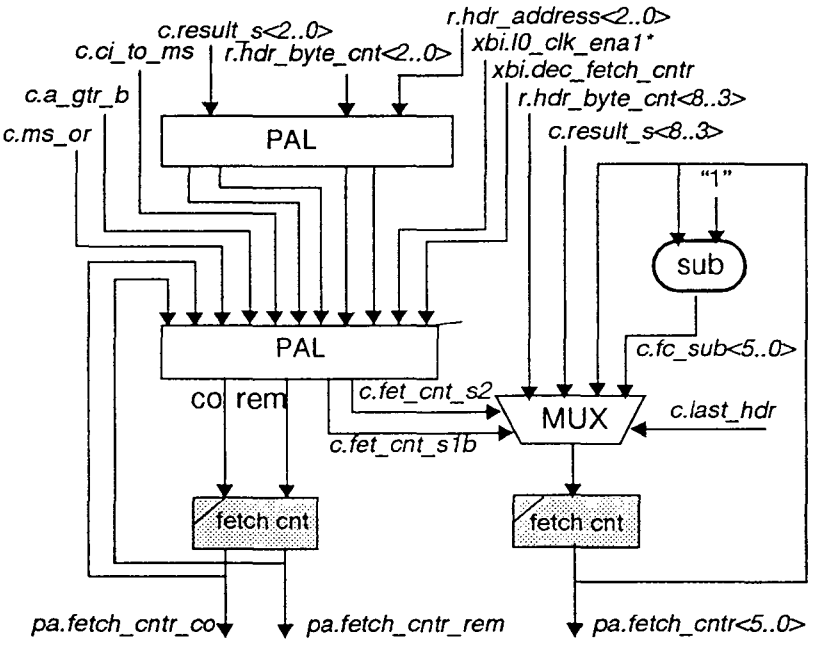
The fetch counter is included in the PA block because it is closely integrated with the PBC generation logic. Logically, this function should reside in the Crossbar Interface block since that is were the fetch count value is used. However, timing constraints prevented using the PBC output to generate a fetch count value. The fetch count value is calculated in parallel with the PBC generation in order to satisfy timing requirements and is therefore included in the PA block.

The fetch counter is initially loaded when a partial header transfer advances into the XS0 stage of the crossbar interface (XBI). The fetch counter is loaded with the number of crossbar transfers associated with a partial header. The fetch count is used by the XBI logic to prefetch data from the Write Data Queue during write transfers and to know when the partial header transfer has completed. The fetch counter itself can perform the following functions: load a new value, hold the current value, or decrement the current value by one. The XBI block controls the load and decrement functions with two control signals: *xbi.l0_clk_ena** and *xbi.dec_fetch_cntr* respectfully. If neither control signal is asserted, the fetch counter holds the current value. Logic in the PA block determines the load value and decrement value when needed.

In its simplest form, the fetch counter is loaded with a mod 8 value of the PBC. In fact, *pa.fetch_cntr<5..0>* is just that, a mod 8 PBC. However, unaligned transfers require the need for two additional states: *pa.fetch_cntr_co* (carryout flag) and *pa.fetch_cntr_rem* (remainder flag). The carryout flag and the remainder flag are conditionally set as a result from adding the least significant three bits of the header address to the least significant three bits of the PBC. If the sum of the addition is a non-zero value then the remainder flag is set. If the sum of the addition results in a carry out, the carryout flag is set. The carryout flag and the remainder flag each represent one transfer in addition to the fetch counter value.

Timing constraints prevented the use of the PBC generator output to calculate the fetch counter load value. Therefore, the fetch counter logic calculates two fetch count values: one assuming that the transfer will be split, and another assuming that it will not. The fetch counter logic uses the *c.last_hdr* control signal from the PBC generation logic to determine what to load for the *pa.fetch_cntr<5..0>* value. If *c.last_hdr* is true, then the header byte count, *r.hdr_byte_cnt<8..3>* is used. Otherwise, *c.result_s<8..3>* (from the PBC generation logic) is loaded into the fetch counter. The fetch counter logic uses the components of *c.last_hdr* (*c.ci_to_ms*, *c.ms_or*, and *c.a_gtr_b*) to determine the carryout and remainder flag values.

Figure 6-5 Fetch Counter Logic



After the fetch counter is loaded with a new value, the value can be held or decremented under control of the XBI block. When a decrement command is issued, *xbi.dec_fetch_cntr*, the fetch counter logic tries to clear the remainder flag first. If the remainder flag is already clear, then the carryout flag is cleared. If both the remainder and carryout flags are cleared, the fetch count value, *pa.fetch_count<5..0>*, is decremented. The decremented fetch count value, *c.fc_sub<5..0>*, is calculated by a PAL.

6.3.2 WDQ Flush Flag Parity Check

The PA logic uses the Write Data Queue (WDQ) flush flags, *wdq.px_flush_flag*, to clear the *r.px_wrt_cmp* state used in the arbitration logic. The WDQ flush flags are set when a CCU aborts a write transfer prior to completion or when the NIA detects bad parity on a write data transfer. The flush flag is also used by the XBI block to know when a transfer needs to be terminated early.

Like the write data, the flush flags are parity protected in the WDQ. The WDQ flush flag parity is checked in the PA block. The PA logic checks for odd parity and reports a hard error, *pa.wdq_fflag_per* if even parity is found. The PA block also

sources registered versions of the flush flags, *pa.clr_px_wdq*, to the rest of the NIA. These "clear wdq" signals are used to reset the head pointer, tail pointer, and fill level for the I/O channel's WDQ.

6.3.3 CDS Carry Look-ahead

This is one of those functions that really belongs in the CDS Arrays block, but appears here. The CDS arrays perform header updates after the PA logic commits an I/O channel header to the arbitration pipeline. The update is performed to the address and byte count fields of the header. The address field is incremented by the PBC and the byte count field is decremented by the PBC. The address field spans across four CDS arrays and the byte count field across two arrays. Carry look ahead logic is used to speed up this update function across arrays.

Each CDS array outputs a group propagate, *cdsX.prop*, and a group generate, *cdsX.gen**, term. The propagate and generate terms are combined in a PAL to create carry in terms, *pa.carryin_cdsX*. Carry in terms are generated only for CDS arrays 1,2,3 and 7.

7 Crossbar Interface

The NIA's Crossbar Interface logic translates I/O channel memory requests into equivalent requests to the C3800 system memory. The Crossbar Interface logic has two main parts: a memory interface and an interrupt interface. The data path of the memory interface is implemented in the Crossbar Data Slice (XDS) arrays. The data path control and interrupt interface is implemented in discrete parts.

7.1 Memory Interface

The Crossbar (Xbar) memory interface handles transfers to system memory. The interface is really two interfaces: an even side interface and an odd side interface. The interfaces are separate, but are clocked together on the NIA. An address pipeline handles the headers from the Port Arbitration logic and generates address information to the Xbar. A write data pipeline handles the write data from the Write Data Queue. Memory reads are tracked by a read return FIFO. Control logic manges the flow of headers and transfers through both pipelines and the read return FIFO. The address and write data pipelines, the read return FIFO, and the control logic are all described below.

7.1.1 Address Pipeline

The Xbar interface pipeline has three stages: XS0, XS1 and XS2 (see Figure 7-1). A single header transfer can and will often occupy all three Xbar interface stages at the same time. It will advance into each stage one cycle at a time and will advance out of each stage one cycle at a time. The three stages represent the pipeline involved for a write transfer. XS0 represents the cycle to commit an access the Write Data Queue (WDQ). XS1 represents the cycle to access the WDQ and to generate the proper board selects, write zones and address. Finally, XS2 represents the cycle to send the transfer to the Xbar.

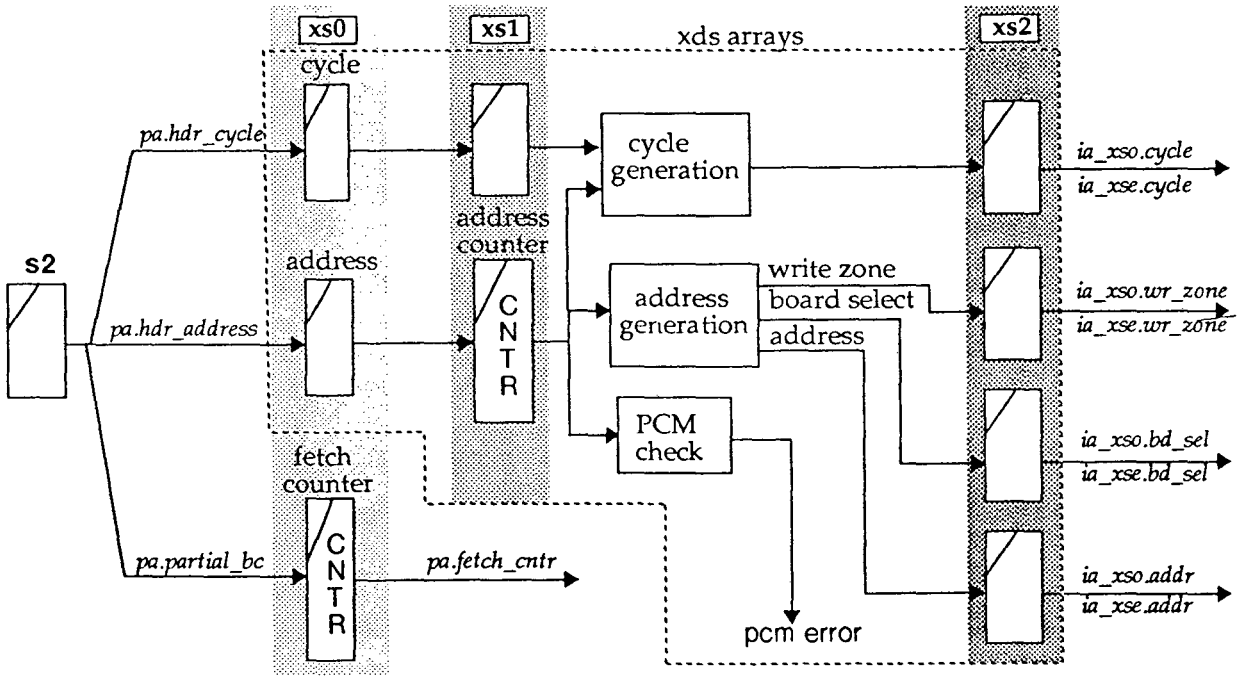
Header transfers are loaded into the XS0 stage from the Port Arbitration logic. When a header advances into the XS0 stage, a fetch counter is loaded in the Port Arbitration logic. The fetch count represents the number of Xbar transfers that are required to service the header. It also represents the number of WDQ read accesses that are required to service a memory write transfer. The fetch counter is loaded for all transfers, however the WDQ is accessed only for write transfers. The fetch counter is decremented each cycle until the count reaches zero. However, if the memory system becomes busy and can not service the requests from the NIA, the fetch counter will hold its current value until the memory system can once again accept transfers from the NIA. The decrementing, holding and loading function of the fetch counter is under control of the Xbar Interface logic (XBI) block.

Once the XS0 stage is loaded with a transfer, the XS0 will hold the transfer until the fetch count reaches zero. However, the transfer will also advance to the XS1 stage on the following cycle occupying both XS0 and XS1 stages at the same time. The XS1 contains a counter that is loaded with the header address. The counter then increments the initial address by 8 (long word offset) each subsequent cycle for the number of cycles specified by the fetch counter. The XS1 address counter will hold its current value if the memory system becomes busy during the block of transfers. The output of the XS1 address counter is decoded into the proper board select and write zones. Board select generation is a function of memory interleave and a

physical board remap. Write zones are a function of the starting address and byte count of the transfer. If the transfer is a read operation, the address is checked against the PCM mapping. The header cycle field is encoded into the desired crossbar cycle transfer.

The boards selects, write zones, address and cycle from the XS1 stage are loaded into the XS2 stage. The XS2 stage drives the Xbar interface signals directly to the Xbar. The XS2 stage is generally free running until the memory system becomes busy. At that point, the XS2 stage will hold the current transfer until it is accepted by the Xbar.

Figure 7-1 Xbar Interface Address Pipeline



7.1.1.1 Address and Cycle Generation

The address generation function in the XDS arrays takes the 32 bit header address and generates the proper write zones, memory board selects and a 26 bit address. The generated address and board selects are a function of the memory interleave and the logical to physical memory board remap. The memory interleave function is performed first followed by memory board remap.

In its purest form, the most significant three address bits, 31..29, are used directly as the memory board selects. This is true for a non-interleaved memory system. Interleaving involves exchanging lower order address bits (specifically bits 9..7) with address bits 31..29 (board selects). Address interleaving begins at address bit 7 since each memory board has an internal 16 way interleave. Memory boards are always interleaved in pairs or groups of pairs. The XDS arrays contain scan initialized state, *xdsXX.swap<6..0>*, that controls interleaved address generation. See Figure 7-2. There are three levels of address interleaving. The first level involves interleaving memory boards into pairs: 0&1, 2&3, 4&5, and 6&7. When this occurs address bits 7 and 29 are exchanged. The second level involves interleaving the board pairs: 0-1 & 2-3, and 4-5 & 6-7. Address bits 8 and 30 are exchanged at this level. The third level involves interleaving the two groups of

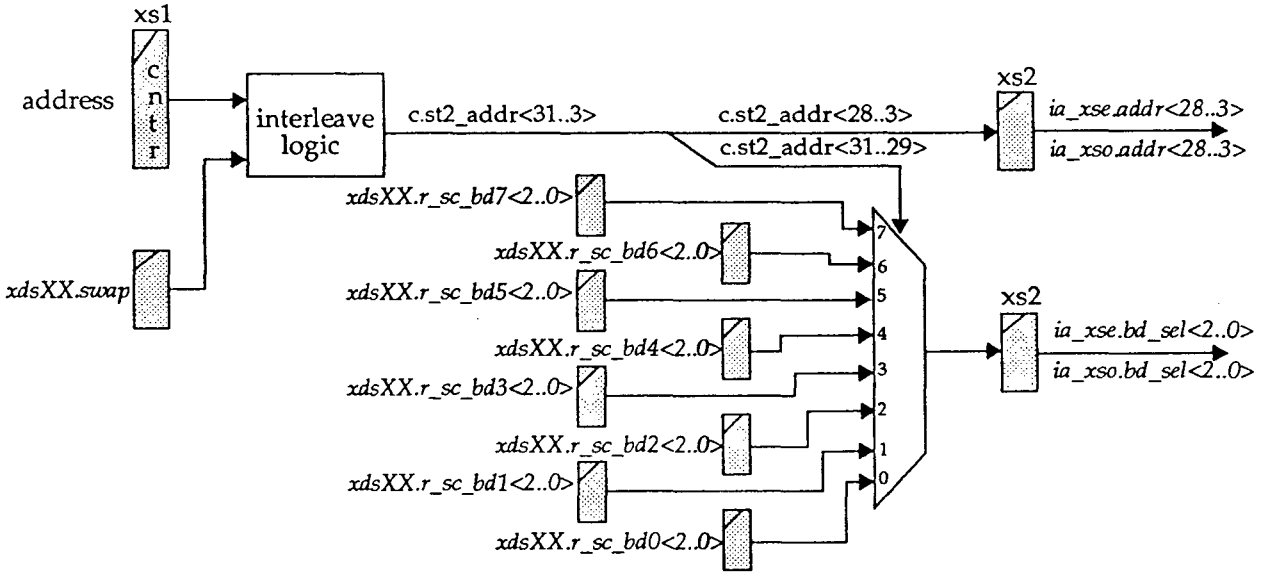
four boards: 0-3 & 4-7. Address bits 9 and 31 are exchanged at this level.

Figure 7-2 Memory Board Interleave Control

Interleave level	Interleave control	Memory Boards 7 6 5 4 3 2 1 0	Address change
1st level	*xdsXX.swap<3..0>*	<3> <2> <1> <0>	exchange bits 29 and 7
2nd level	*xdsXX.swap<5..4>*	<5> <4>	exchange bits 30 and 8
3rd level	*xdsXX.swap<6>*	<6>	exchange bits 31 and 9

After the interleaving function, the board selects go through a logical to physical board select remap. This remap function allows a "logical" memory board select to address a memory board in any "physical" port. For example: memory board 0 can reside in memory port 7 by remapping the 0 select to 7. The logical to physical remap is controlled by scan initialized state, *xdsXX.r_sc_bd7..0<2..0>*, in the XDS arrays. See Figure 7-3. The logical selects (c.st2_addr<31..29>) drive an 8:1 mutiplexor which selects the desired remapping. The *xdsXX.r_sc_bd7..0<2..0>* registers must be initialized, even if remapping is not desired.

Figure 7-3 Logical to Physical Board Select Remap.



The write zones are used only during write operations. Write zone and cycle generation work together to write only the desired byte or bytes within a long word container. The NIA's even and odd side interfaces are locked together, meaning both sides make transfers together. The even and odd side write zones are used to selectively write the bytes within a word. The even and odd side cycle fields are used to control writes to the even or odd words. Take, for example, a partial long word write of say three bytes starting at byte address 0. The even side write zones will select the first three bytes (*ia_xse.wr_zone<3..0>* = 7). The even side cycle will specify a write operation. The odd side word is not written. However, the even and odd sides are locked together, so the odd side cycle field is changed to specify a NOP operation. The odd side write zones are a "don't care" value since the cycle specifies a NOP. All partial long word write operations involve altering the write zones and cycle field to affect the desired byte or bytes.

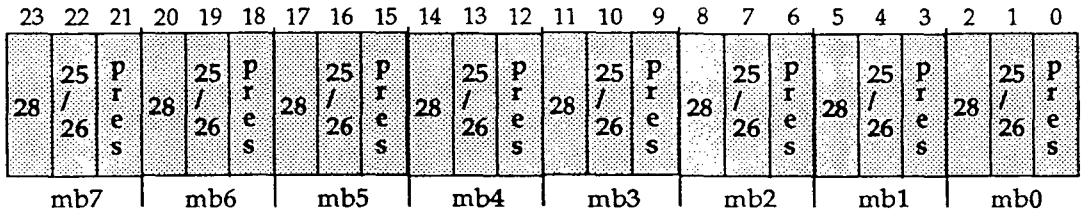
7.1.1.2PCM Check

PCM checking verifies that memory is resident at the specified address. Installed memory may or may not reside in a contiguous address space. "Holes" in the memory space may exist depending upon the number of memory boards installed and the size of each memory board. The PCM (Physical Configuration Map) defines what memory addresses are valid based upon the number and size of memory boards installed in the system. The PCM information is encoded and scan loaded into the XDS arrays, *xdsXX.r_sc_bd<32..0>*. This function is provided by the memory initialization program "mminit". The XDS arrays provide PCM checking on read transfers only. Write transfers are PCM checked at the I/O channel interfaces.

Up to eight memory boards can be installed in a system. The PCM state, *xdsXX.r_sc_bd<32..0>*, in the XDS arrays describes which memory boards are installed and the valid addresses for each board. Valid memory board sizes are 64MB, 128MB, 256MB and 512MB. Only address bits 28, 25 and 26 are needed in the PCM check. See Figure 7-4. The "pres" bits indicate if the memory board is present. The "28" bits indicate if address bit 28 is valid (can be set) for that memory board. The "25/26" bits indicate if address bits 25 and 26 are valid for that memory board.If a PCM error is detected, the transfer is aborted and the error indication is written into the read return FIFO.

Figure 7-4 XDS Array PCM State

xdsXX.r_sc_bd<23..0>



	28	25/26	pres
no board	X	X	0
64MB	0	0	1
128MB	1	0	1
256MB	0	1	1
512MB	1	1	1

For address bits 28, 26 & 25:
X = don't care
0 = address bit can not be set
1 = address bit can be set

7.1.2 Write Data Pipeline

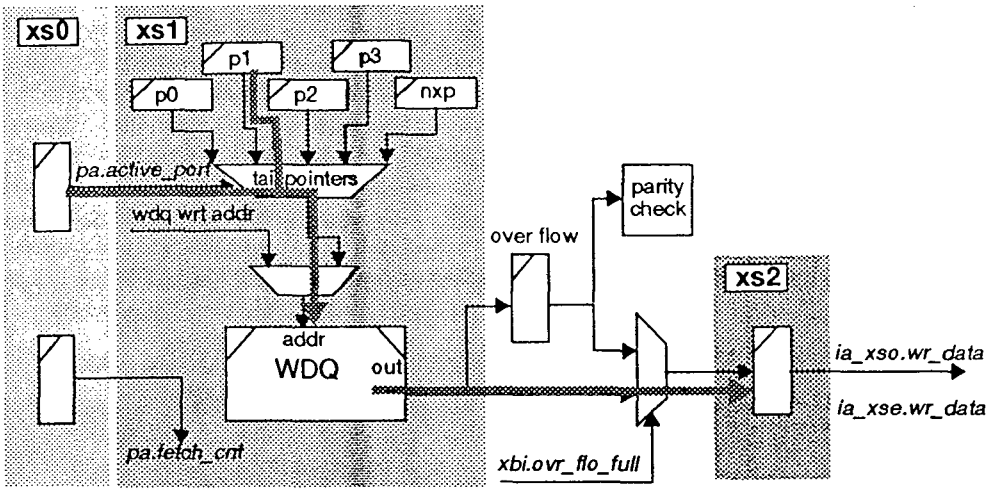
The write data pipeline is similar to the address pipe described above. During the XS0 stage, the fetch count and transfer type are used to decide whether or not to commit a read access to the WDQ. If a read is committed, the *pa.active_port<2..0>* field selects the appropriate WDQ tail pointer address, and the appropriate tail pointer increment control is asserted. The WDQ read address is clocked into the WDQ and the appropriate tail pointer is updated to mark the beginning of the XS1 stage. Read data from the WDQ is valid midway into the XS1 stage and is captured in an XS2 stage register. This XS2 register sources write data to the Xbar/memory system. See Figure 7-5.

Data from the WDQ is also captured in an over flow register. The over flow register holds read data from the WDQ when memory becomes busy and the Xbar stops

accepting transfers from the NIA. The over flow register is necessary because reads to the WDQ must be committed to prior to knowing if the previous transfer will be accepted. Once the NIA sees that memory is busy, it stops committing reads to the WDQ. However a read access could be active at the moment and therefore needs a place to be held until memory is no longer busy. The over flow register provides this functionality. The over flow register is also used to check write data parity. All write data transfers are parity checked as data is sent to the Xbar.

Once the overflow register is full, the XS2 register input mux will select the over flow register to be loaded into the XS2 register. The XS2 register will hold the current transfer until memory is no longer busy. When this occurs, the NIA commits a read access to the WDQ and the XS2 register gets loaded with the over flow register contents.

Figure 7-5 Xbar Interface Write Data Pipeline



7.1.3 Read Return FIFO

Memory read requests are "tracked" by a read return FIFO. The contents of the read return FIFO are used to identify read data returned to the NIA from system memory. Read data is always returned to the NIA in the order that it was requested. When a memory read request is made, a port ID,*pa.channel_id<2..0>*,is written into the FIFO along with a CCU ID, *pa.ccu_id*, a TOC ID, *pa.toc_hdr*,and potentially a PCM error flag. When read data is received by the NIA from system memory, the top of the FIFO is read. The port ID from the FIFO tells the NIA which I/O channel the read data is destine for and where to write the data in the Read Data Queue (RDQ).

The FIFO is only 24 locations deep. The NIA should never be able to over flow or even fill up the FIFO. As the NIA makes memory read requests and thus fills the FIFO, one of two things will occur: read data will start returning from memory and the FIFO will begin to empty, or the memory system will become busy and the NIA will have to stop sending read requests to the Xbar/memory. However, the NIA will also detect a "FIFO full" condition, *r.fifo_full*,and will stop making requests to the Xbar until read data begins returning from memory.

The TOC IDs, *xds.toc_id_e* and *xds.toc_id_o*, and CCU IDs, *xds.ccu_id_e* and *xds.ccu_id_o*, from the FIFO are used for Time of Century (TOC) reads. When TOC data is returned to the NIA, it is written into a specific location in the RDQ. The

TOC ID identifies the returning data as TOC data. The location that is written in the RDQ is based upon the port ID, and the CCU ID. Each CCU has a unique location in the RDQ for TOC reads.

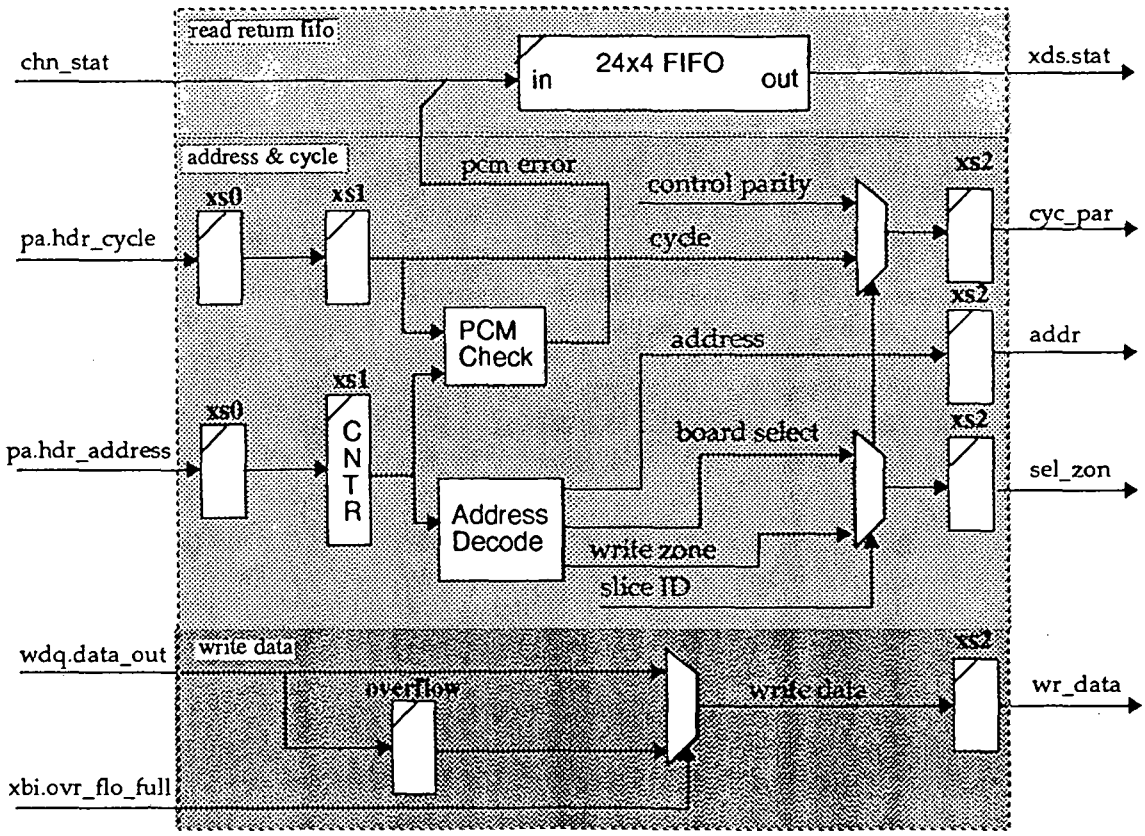
The PCM flag is set in the read return FIFO when a PCM error is detected on a read request. Pbus protocol requires the NIA to return all read data that was requested up to the point of where the read address crossed into non-existent memory. When a PCM error is detected on a read request, the request is aborted and the PCM error flag is set in the read return FIFO. When the PCM error flag, *xds.pcm_flag_e* or *xds.pcm_flag_o*, reaches the top of the FIFO, the XBI logic forces a read of the FIFO. Normally, the FIFO is read only read when data is returned to the NIA. The even side PCM error flag is written into the RDQ and is latter used by the I/O channel interface to send a bus error to the appropriate CCU.

7.1.4 XDS Arrays

The XDS arrays contain the address pipe, write data pipe and the read return FIFO previously described. There are four XDS arrays on the NIA. XDS arrays 00 and 01 interface to the odd side Xbar/memory. Arrays 10 and 11 interface to the even side. Each array supplies 16 bits of write data and 13 bits of address per even or odd side interface. A scan initialized slice ID selects which array sources board selects or write zones and control parity or cycle.

The XDS arrays also contain pointers and fill levels for the Read Data Queue and Write Data Queue. The operation and control of these pointers are detailed in the Read Data Queue and the Write Data Queue Chapters.

Figure 7-6 XDS Array



7.1.5 Xbar Interface Control Logic

The address and write data pipelines are controlled by the Crossbar Interface (XBI) logic. The XBI control logic provides the following functions:

- Clock enables for the XS0, XS1 and XS2 pipeline stages.
- Request "ready" signals to the Xbar interface.
- Fetch counter, address counter, and read return FIFO control.
- Write Data Queue tail pointer increment controls.
- Request Pending and Store Pending counters

7.1.5.1 Clock enables

It is important to note that the even and odd side Xbar interfaces are clocked together. A new header will be allowed to advance into the Xbar interface pipeline only when the previous header has cleared both even and odd sides. For example, suppose the even side Xbar becomes busy and can not accept transfers, and yet the odd side Xbar does not go busy. The XBI control logic will hold off sending transfers on the odd side until the even side is no longer busy and can begin accepting transfers.

Each pipeline stage will free run if it is not full. When a header or transfer is loaded into a stage, that stage's clock enable will go false until the header or transfer has satisfied that stage's requirements. The XS2 stage clock enable, *xbi.l2_clk_ena*, requirement is relatively simple: clock unless the current transfer will not be accepted by the Xbar. The NIA receives "request next" signals, *xse_ia.a_req_next*, *xse_ia.b_req_next*, *xso_ia.a_req_next*, and *xso_ia.b_req_next*, from the Xbar which when true, tell the NIA that the current transfer (if any) in the XS2 stage will be accepted. If one of these request next signals goes false, the current transfer in the XS2 stage must be held.

The XS1 stage enable, *xbi.l1_clk_ena*, is true when the XS1 stage is empty or when then last transfer for the current header is advancing to the XS2 stage. The signal *xbi.last_xfr* defines the last transfer from the XS1 stage. However, there are several "hold" conditions that can prevent a transfer from advancing to the XS2 stage. These hold conditions are the heart of the XBI control logic. The following is a list of the conditions that can hold up a transfer in the XS1 stage.

- The XS1 transfer is a Test and Modify type instruction and one of the store pending signals is true or the store pending count is not zero.
- The XS1 transfer is a request to the NCU (MBP, TOC or interrupt) and one of the request pending signals is true or the request pending count is not zero.
- The read return FIFO is full.
- A PCM error was detected on a memory read request,
- The XS2 stage has a valid transfer and memory is busy.
- A "read flush" state is set in the Port Arbitration logic, meaning that one or more of the I/O channels had an aborted read operation.

The XS0 stage enable, *xbi.l0_clk_ena*, is true when the XS0 stage is empty or when the fetch count equals one and none of the previously defined hold conditions are true. The "fetch count equals one" state, *c.last_fetch*, is used to set the *xbi.last_xfr* state described above.

7.1.5.2 Request readys

The request ready signals tell the Xbar that the NIA is making a memory request. There are two request ready signals, one for the even side Xbar and one for the odd side Xbar, *ia_xse.rdy* and *ia_xso.rdy*.The request ready signals are asserted when the XS1 stage is full, none of the hold conditions are true, and the WDQ write abort signal is not true. The write abort signal is asserted from the WDQ whenever a write transfer was aborted prior to completion. When this occurs, the NIA must write all of the data sent prior to the abort. The write abort flag aborts the current transfer and terminates the header in the XS1 stage.

The request ready signals are also asserted whenever the XS2 stage has a transfer and memory becomes busy. The request ready signals will remain asserted along with the transfer in the XS2 stage until memory can accept the transfer.

7.1.5.3 Counter and return FIFO control

The fetch counter in the Port Arbitration logic is controlled by the XBI logic. The fetch counter performs three operations: load a new count, hold the current count, or decrement the current count. The fetch counter is loaded with a new value when the XS0 clock enable, *xbi.l0_clk_ena*, is true. If the clock enable is not true, then the current count is either held or decremented. The "decrement fetch counter" signal, *xbi.dec_fetch_cntr*, is asserted when the count should be decremented. This occurs when the XS0 stage is full and none of the hold conditions are true. If *xbi.dec_fetch_cntr* is not true and the XS0 clock enable is not true, then the fetch counter holds it's current value.

The address counters in the XDS arrays provide the following functions: load a new address, hold the current address, or increment the current address. The XS1 clock enable, *xbi.l1_clk_ena*, when true, tells the XDS arrays when to load their address counters. If the XS1 clock enable is not true, then the count control, *xbi.l1_count*, tells the address counters to increment. If both XS1 clock enable and the count control are false, the address counters will hold their current value. The *xbi.l1_count* signal is true when the XS1 stage is full and none of the hold conditions are true.

The read return FIFO in the XDS arrays is written and read under control from the XBI logic. Like the rest of the interface, the FIFO is split into even and odd sides. Both sides are written at the same time, but are read separately. Each even and odd side FIFO is written with the port ID and CCU ID of the header in the XS1 stage, a TOC header flag, and a PCM error flag. The *xbi.wrt_fifo* control signal causes the even and odd sides of the FIFO to be written. This signal is true when the XS1 stage is full and the XS1 header is a memory read transfer and none of the hold conditions are true.

The FIFO is read when read data returns from memory or when the PCM error flag reaches the top of the FIFO. Two read controls are used: *xbi.fifo_read_e* for the even side FIFO and *xbi.fifo_read_o* for the odd side. The even and odd side interfaces have read ready control signals, *xre_ia.rd_rdy* and *xro_ia.rd_rdy*, to indicate when the Xbar has returned read data to the NIA. The registered versions of these control signals cause the FIFO read control signals to be asserted. The PCM error flag will also cause a FIFO read. Read data is not returned to the NIA when a PCM error occurs. However, the NIA needs to write the PCM error flag in the RDQ after all of the previously requested read data has returned to the NIA. By the time the PCM error flag reaches the top of the FIFO, all the previously requested read data

has been returned to the NIA. At that point the PCM error flag is written into the RDQ.

7.1.5.4 WDQ tail pointer control

The WDQ tail pointers are incremented under control from the XBI. The increment signals, *xbi.inc_px_wdq_tp*, are asserted when data is read from the WDQ. Only one I/O channel's WDQ tail pointer is incremented at a time. The active port signals, *pa.active_port<2..0>*, tell the XBI logic which tail pointer to increment. An I/O channel's tail pointer is incremented when the XS0 stage is full and the header is a memory write transfer and none of the hold conditions are true.

7.1.5.5 Request and store pending counters

The store pending counter, *r.st_pnd_cnt<1..0>*, and the request pending counter, *r.req_pnd_cnt<1..0>*, are used to help determine two of the hold conditions. The store pending counter is used in conjunction with the store pending signals, *xse_ia.a/b_st_pend* and *xso_ia.a/b_st_pend*, from the Xbar. When the NIA sends a store transfer (memory write), it takes three cycles for the Xbar to indicate back to the NIA that the store is pending. The store pending counter is used to count off the three cycles following a store transfer. In essence, the store counter is a local store pending indication on the NIA. This same situation holds for the request pending counter, except the request counter is used to count off the three cycles following any memory read or write request.

The store pending counter is set to a value of three each time the NIA sends a memory write transfer. If the NIA continues to send write transfers, the counter will continue to stay at three. When the NIA stops sending stores, the store pending counter will begin to count down each cycle until it reaches zero. The request pending counter operates the same way except that it's value is set to three for any memory transfer from the NIA.

7.2 Interrupt Interface

The Xbar interrupt interface has two parts: send interrupt logic for sending interrupts from the NIA to the NCU, and receive interrupt logic that receives interrupts from the NCU. The Xbar interrupt logic also interfaces with the Pbus and NXP interrupt state machines.

7.2.1 Send Interrupt Logic

The send interrupt logic interfaces with several blocks of logic on the NIA. Since interrupts are sent to the NCU using the Xbar memory interface, the send logic does not have many interface signals to the Xbar. The send logic basically sequences all of the events that occur in the process of sending an interrupt. The following paragraphs describe the send interrupt flow.

The send interrupt flow begins at the I/O channel interrupt interfaces. There are two I/O channel interrupt interfaces: the NXP interrupt interface and the Pbus interrupt interface. Each I/O channel interface controls and sequences through interrupt bus cycles to send or receive interrupts from the CCUs. When an interrupt is received from a CCU, the I/O channel interrupt interface informs the XBI send logic. If both the NXP and Pbus interfaces receive an interrupt at the same time, the XBI send logic handles them one at a time based upon a round robin

priority scheme. The send state machine starts out in state 0 and advances to state 1 upon seeing an interrupt ready, *r.nxp_intr_rdy* or *r.pb_intr_rdy*, from one of the I/O channel interfaces. The XBI logic then schedules a write access, *xbi.wdq_ena*, with the Write Queue Arbitration logic in order to write the interrupt vector into the WDQ. Once the vector has been written into the WDQ, the XBI send logic acknowledges the appropriate I/O channel interface, *xbi.nxp_intr_ack* or *xbi.pb_intr_ack*, and then advances to state 2.

In state 2, the XBI logic schedules a memory access with the Port Arbitration (PA) logic. The PA logic gives the interrupt highest priority to the Xbar interface and will select the interrupt header from the CDS gate arrays. Once the interrupt header reaches the S1 stage in the PA pipeline, *pa.int_val_s1*, the send state machine advances to state 3 and waits for the interrupt transfer to be sent to the NCU. When the interrupt header reaches the Xbar interface, the interrupt vector is read from the WDQ, as in a normal write transfer. However, the NCU board is selected and the trap register address is specified for a "sendx" type operation to the NCU.

The trap register in the NCU is semaphore protected.Therefore, the NIA must wait for status from the NCU that says if the interrupt transfer was successful. The NCU sends two status signals to the NIA, *xc_ia.cu_status_en* and *xc_ia.cu_status*. The status enable tells the NIA that the status is valid. If the status bit is true, the send interrupt transfer is complete and the send logic advances to state 0. If the status bit is false, the interrupt transfer must be sent again because the trap register was already full during the previous sendx operation. When this occurs, the send logic returns to state 2 and schedules another memory access with the PA logic. This cycle is repeated until the NCU status returns true and the send logic advances to state 0.

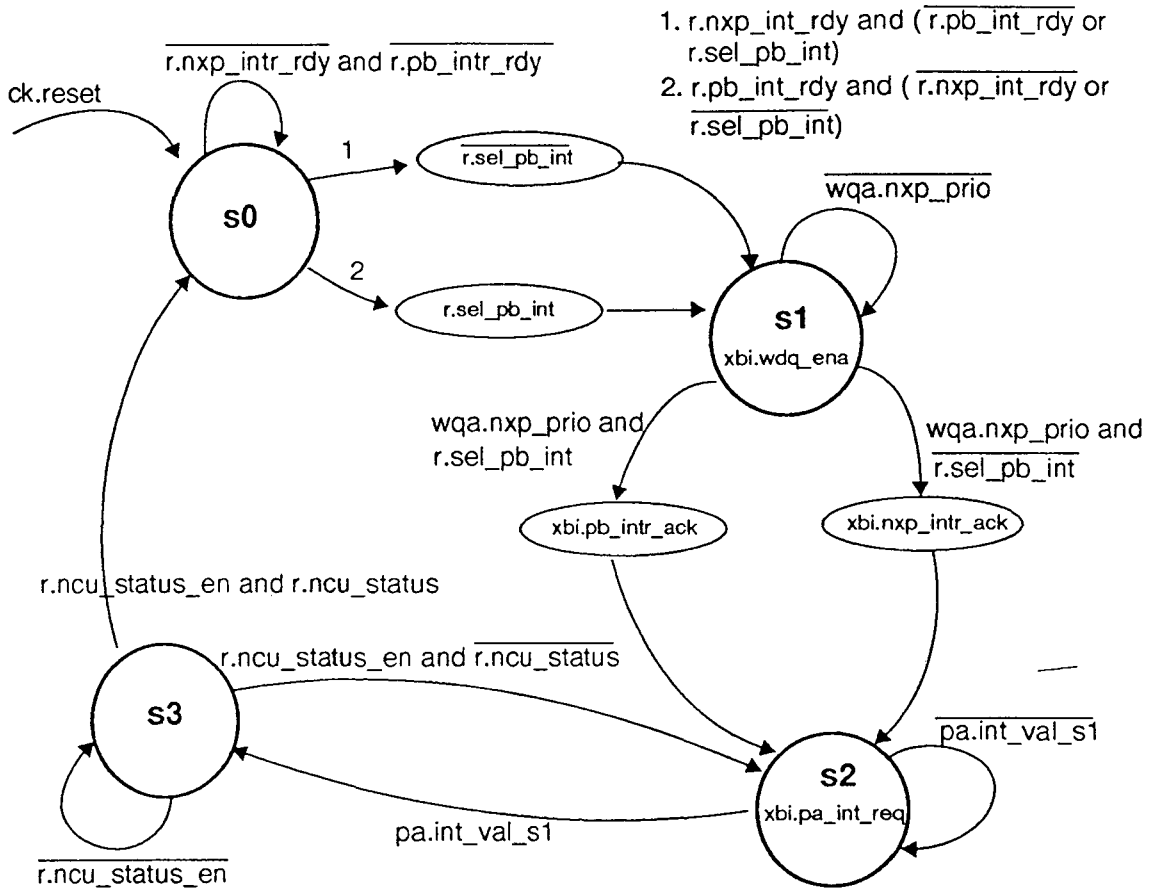
7.2.2 Receive Interrupt Logic

The receive interrupt logic receives interrupts from the NCU and forwards them to the Pbus and NXP interrupt interfaces. The receive logic registers the lower 8 bits of the 12 bit trap vector bus from the NCU, *xc_ia.trap_vec<7..0>*, The receive logic also captures the 4 bit trap type field, *xc_ia.trap_type<3..0>*, and sources a trap complete signal, *ia_xc.trap_comp*, back to the NCU. The following paragraphs describe the receive interrupt flow.

The NCU asserts the trap ready signal, *xc_ia.trap_rdy*, to the NIA when the trap vector bus is valid. When valid, the receive interrupt logic captures and holds the trap vector and then examines the trap type field. If the trap type is zero, it asserts interrupt ready to the Pbus and NXP interrupt interfaces, *xbi.ncu_intr_rdy_p* and *xbi.ncu_intr_rdy_x*. If the trap type is not zero, the receive logic remains idle and releases the trap vector register to receive another transfer.

The Pbus and NXP interfaces each get their own version of interrupt ready because these interfaces operate at different clock speeds from each other and from the receive interrupt logic. The Pbus and NXP interfaces each supply the receive logic a "busy" signal, *pi.pb_busy* and *nxi.nxp_intr_busy*, to indicate when it can not accept another interrupt vector. If the interface is busy, the receive logic continues to assert the interrupt ready signal to that interface. Once the busy signal is false the receive logic removes the interrupt ready signal, knowing that the interface has received the new interrupt vector. When both Pbus and NXP interfaces have received the new interrupt vector, the receive logic asserts the trap complete signal back to the NCU.

Figure 7-7 Xbar Send Interrupt State Machine



The trap complete does not mean that the interrupt has been delivered to the CCUs. It's only meaning is the NIA can accept another interrupt from the NCU. If the NCU sends a second interrupt soon after the first, chances are the Pbus and NXP interrupt interfaces will be busy. If this occurs, the NCU must wait for the first interrupt to be delivered to the CCUs before the receive logic can forward the second interrupt to the Pbus and NXP interfaces.

8 Clocks and Scan

The NIA receives a 2x and a 3x clock from the clock generator on the NCU. In turn, the NIA generates 3x, 2x and system (1x) clocks as well as NXP and Pbus clocks for use on the NIA. Over 300 clocks are generated on the NIA with only one load per clock net.The NIA also generates clocks for Channel Control Units (CCUs) which interface to the NIA. Each CCU receives a 20Mhz clock, a free running 10Mhz phase clock and a stoppable 10Mhz phase clock from the NIA. All interfaces to the NIA are synchronous.

The NIA also implements four different scan rings. Some scan operations are performed dynamically with clocks running, while others are performed as normal scan operations. The Clock logic block also contains the hard error reporting logic.

8.1 Clock Generation Logic

The NIA contains two independent clock generation trees. The 2x clock input to the NIA is used to create 2x, 1x, NXP and Pbus clocks for use on the NIA. These clocks generally have three levels of buffering before reaching a clocked device. The 3x clock input is used to generate 3x and 1x clocks on the NIA. The 3x clocks are used to create CCU clocks. The 3x and 1x clocks generated from the 3x clock input are referred to as "free running" clocks. These "free running" clocks are controllable (i.e. stoppable) if need be, but will usually free run during scan and diagnostic operations on the NIA. Clocks sourced from the 3x clock input go through two levels of buffering before reaching a clocked device on the NIA.

8.1.1 2x Clock Tree

The 2x clock input is the primary clock for the NIA. The 2x clock is divided down into 1x, NXP and Pbus clocks. The divide down logic is similar to clock logic used on other C38xx boards. Two 4-bit shift registers circulate clock buffer enables to generate 2x and 1x clocks. The 2x clock shift register has all of its bits active and is implemented as such for use with the Convex At Speed Test (CAST) system. The 1x clock shift ring has every other bit active to perform a divide by two operation. Both shift rings are initialized by scan at board power up. See Figure 8-1.

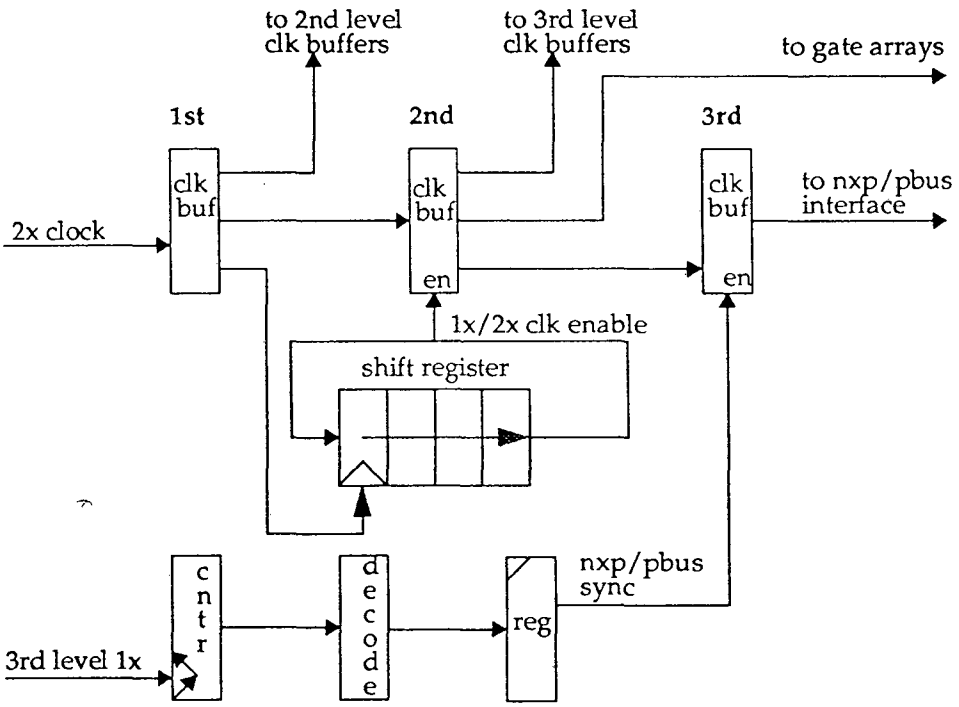
All "non-free running" devices, except gate arrays, receive a third level clock from the 2x clock generation logic. As the 2x clock input enters the NIA it is sent immediately into an ECLiPS™ 100E111 differential buffer. This is the first level buffer. Two of the differential outputs from the first level are used to clock the divide down shift rings. The reset of the differential outputs (7 pairs) are sent to second level differential buffers. The clock buffer enables from the divide down shift registers are used to enable the second level buffers. In the case of the 1x clock generator, every other 2x clock is gated off by disabling the second level buffer. Thus, the 2x clock is divided down into a 1x clock at the second level buffer. In the case of the 2x clock generator, the second level buffer is always enabled and thus the 2x clock from the first level passes through the second level buffer unchanged.

Most of the second level clocks (2x and 1x) are sent to third level buffers. However, twelve of the second level 1x clocks bypass this third level and are sent directly to the twelve gate arrays on the NIA. Gate arrays receive second level clocks in order

to reduce the amount of clock skew between gate arrays and board level discrete components. Third level buffers for 1x and 2x clocks are always enabled. However, another level of clock enabling is performed at the third level buffer for NXP and Pbus clocks. NXP and Pbus "sync" clock enables, *r.nxp_sync** and *r.pbus_sync**, are generated by centralized clock cycle counters. The "sync" signals disable the third level buffers until the "sync" signal is true. The NXP sync signal is true every other 1x clock cycle and the Pbus sync signal is true every six 1x clock cycles. NXP clocks are used in the NXP interface logic and Pbus clocks are used in the Pbus interfaces.

Twox clocks are used by the Read Data Que and Write Data Que logic. The queues are accessed twice every 1x clock period. Thus the self-timed RAMs receive a 2x clock.

Figure 8-1 Three Level Clock Buffering



8.1.2 3x Clock Tree

The 3x clock tree is very similar to the 2x clock tree except there are two levels of clock buffering instead of three. The first level of buffering occurs on the NCU and the next two levels occur on the NIA. Both 3x and 1x clocks are generated from the 3x clock input. The 3x clocks are used to generate CCU clocks. The 1x clocks are used with registers that interface with the NCU during CCU scan operations.

The 3x clock input enters the board and is sent to the 1st level buffer on the NIA. Two differential clock pairs are used to clock the divide down shift rings. Each shift ring is six bits long. The divide down rings create 2nd level clock buffer enables. The 2nd level clocks are sent to free running clocked devices on the NIA.

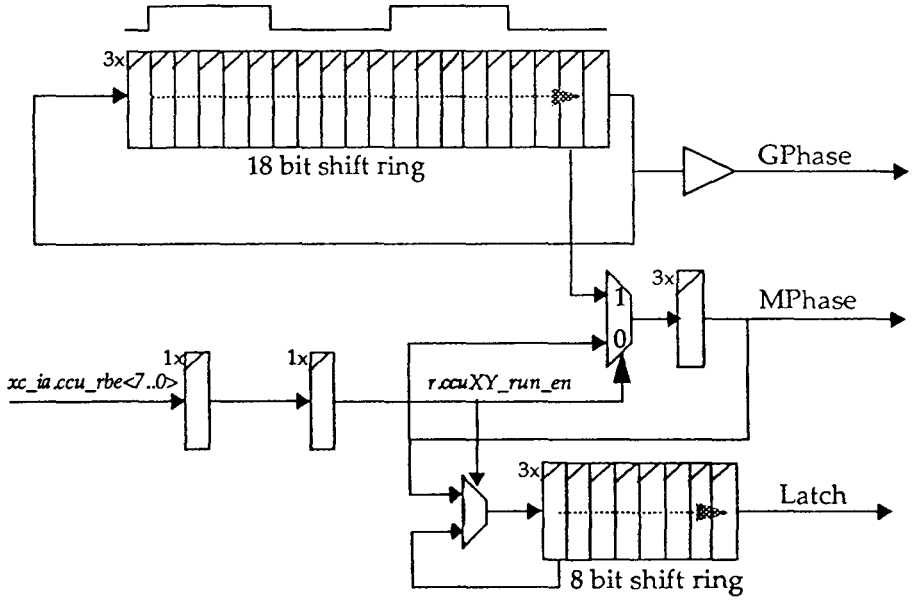
8.1.3 CCU Clock Logic

The NIA generates clocks to the CCUs that interface with the NIA. The NIA

supplies three clock signals to each CCU: a 20Mhz clock, a 10 Mhz "MPhase" and a 10 MHZ "GPhase". The MPhase clock is stoppable and the GPhase clock is free running. Each CCU uses these three clock signals to generate its own free running and stoppable 10Mhz clocks.

The NIA generates the 20 Mhz clock by circulating a predefined pattern of ones and zeros in an 18 bit shift ring. The 18 bit shift ring is clocked at the 3x clock rate. The ring is forced loaded with a predefined value once every 18 1x clock periods. The force loading of the shift ring is controlled by the NCLK array on the NCU. The NCU sources a signal called *xc_ia.clock_sync* to the NIA. When true, this signal forces the 20 Mhz shift ring to force load and resets the NIA's clock cycle counter. The clock_sync signal provides a "warm fuzzy" about keeping the state of the CCU clocks and the NIA clocks in sync. The 20Mhz shift ring is scan loadable and can be initialized with the correct pattern of ones and zeros at power up.

Figure 8-2 MPhase and GPhase Generation



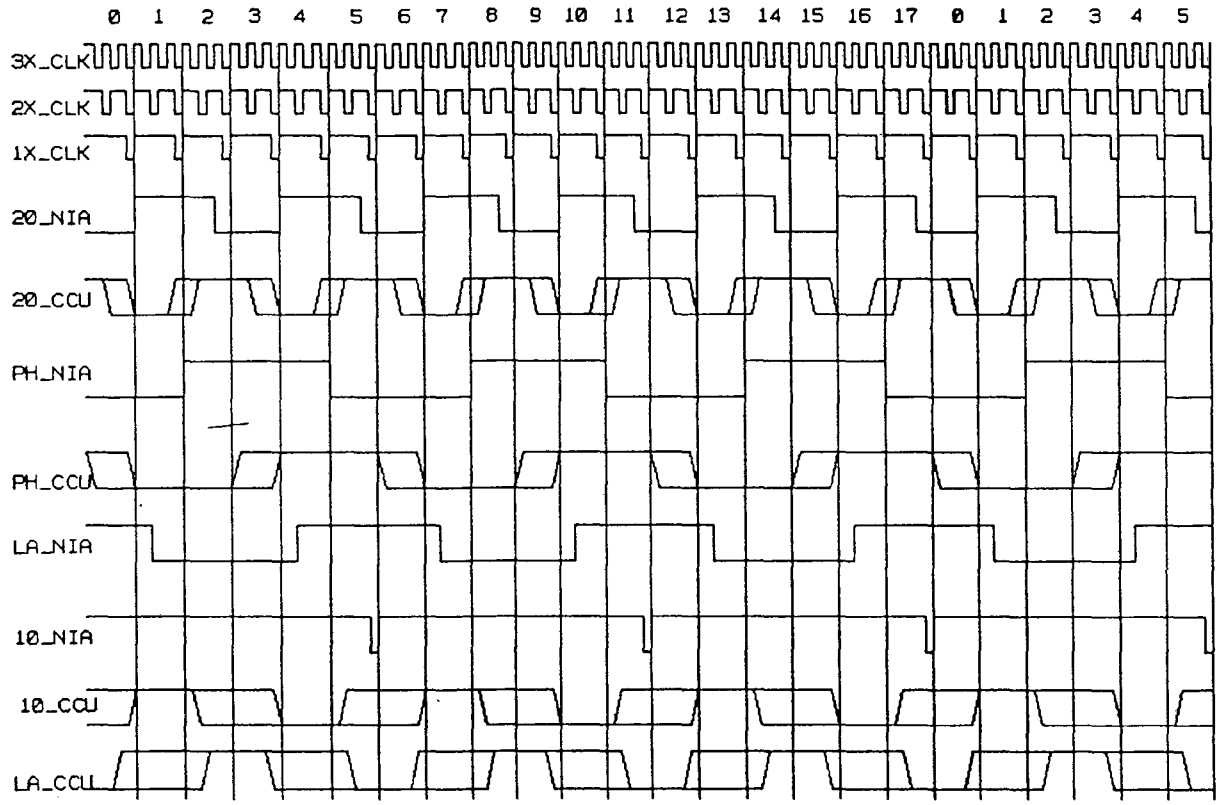
The MPhase and GPhase clocks are generated in a similar fashion by circulating a pattern of ones and zeros in an18 bit shift ring. This ring is also clocked at the 3x clock rate. Like the 20 Mhz ring, the phase ring is force loaded under control of the clock_sync signal from the NCU. The MSB of the phase ring is buffered and sent to the CCUs as the GPhase or free running phase clock. The Mphase clocks are qualified by "run bit enables" from the NCU. Each CCU has a run bit enable that is set or reset by the SPU. When a CCU's run bit enable is true, the MPhase is running and looks just like the GPhase. When the run bit enable is false, the MPhase is stopped. The GPhase is unaffected by the run bit enable.

Latch clocks are also generated on the NIA. These clocks control latch-translators used to deskew Pbus data transfers. Each Pbus interface has its own latch clocks. Latch clocks are generated by staging the MPhase clock for eight 3x clock periods. Like the MPhase clocks, the latch clocks are controlled by the ccu run bit enables. The latch is held when the ccu run bit enable is false and runs when its true.

8.1.4 Clock Cycle Counter and Clock Alignment

There is a fixed relationship between the system clocks on the NIA, the Pbus clocks and the NXP clocks. The system clock to NXP clock ratio is 2:1. The system clock to Pbus clock ratio is 18:3 or 6:1. A clock cycle counter is used to label each of the 18 system clocks in the 18:3 ratio. The clock cycle counter outputs are decoded into Pbus clock syncs and NXP clock syncs. The count values range from 0 to 17. The counter is reset to 0 when it reaches a count of 17. See Figure 8-3.

Figure 8-3 Pbus to System Clock Alignment



NXP clock syncs are active on odd numbered cycles. Pbus clock syncs are active on cycles 5, 11, and 17. Pbus clocks on the NIA occur every 6 system clock cycles. Pbus clocks on the CCUs occur every 100ns (or every 18 3x clock cycles). The "fuzz" on the Pbus clocks in Figure 8-3 accounts for the wire delays between the closest and most distant CCUs from the NIA. This represents the worst case clock skew between any two CCUs that are local to the NIA.

8.1.5 Board Level Clock Skew

Clock skew on the NIA is held to a minimum by matching clock net delays across the board. This is accomplished by adding wire to the clock routes were needed. All clocks sourced from the 2x clock input have a 10ns delay from the 2x clock input through the clock tree buffers to the clocked device. There is only one load per clock net. The only skew introduced by the NIA is the part to part skew introduced by the three levels of clock buffers. The clocks to gate arrays are handled in a slightly different manner. Gate arrays receive 2nd level clocks. These clocks are tuned such that the delay from the 2x clock input to the array plus the nominal clock tree delay internal to the array equals 10ns. This means that registers internal to the arrays are clocked at the same time (nominally) as registers external to the gate arrays. Clock skew introduced internal to the arrays is larger than the

clock skew introduced by the clock buffers. Therefore, gate array interface signals have more skew than signals which do not interface to an array.

Clocks sourced from the 3x clock input are also delay matched. However, these clocks have a 9.47ns delay from the 3x clock input to the clocked device. Again, each clock net has only one load. The base delay is smaller for the 3x clock because the NCU buffers the 3x clock prior to sending it to the NIA. The one level on the NCU plus the 2 clock buffer levels on the NIA result in the three levels of clock buffering that is used across the C38xx system.

8.2 Scan

The NIA supports four scan modes: two normal modes for scanning the board, and two "dynamic" modes for scanning the soft error log and for scanning the CCU Scan Ctl register. Table 8-1 lists the different scan modes on the NIA.

8.2.1 Scan Ring Topology

The NIA contains four different scan rings. The Board Left scan ring contains all scannable logic on the NIA. The System Left scan ring contains most of the NIA scannable registers except the logic that gets a clock from the 3x clock tree. The Soft Log scan ring contains the soft error log registers. The CCU Scan Ctl scan ring contains the CCU scan control signals. The scan control signals, *ia_xc.scan_ctl<2..0>*, select the different scan operations on the NIA.

The Board Left and System Left scan rings are used at different times during NIA initialization and debug. The Board Left is used to scan everything on the NIA including the CCU clock generation logic. This type of scan operation destroys the CCU clocks and therefore destroys volatile CCU memory and data. Board Left scan operations are used for power up initialization, for SST, CAST and possibly GENRAD testing. The System Left scan ring is used to scan the NIA during system level debug. The System Left scan mode does not affect the 3x clock input nor the CCU clock generation logic. The System Left scan mode is used to scan the NIA without loosing volatile CCU memory. This would allow one to scan the NIA and then step system clocks (NIA and I/O subsystem) and then scan the NIA again.

Table 8-1 NIA Scan Modes

SCAN CTL	SCAN MODE
000	NORMAL
001	SOFT EROR LOG
010	LAST SYSTEM LEFT
011	SYSTEM LEFT
100	CCU SCAN CTL
101	LAST BOARD LEFT
110	CAST
111	BOARD LEFT

Both the Board left and System Left scan rings include scan ring "signature" bits. The signature bits are implemented as part of the 100492 Self-Timed RAM scan rings and as part of the CDS and XDS gate array scan rings. The signatures from each of these components should always be present at the same bit locations for Board Left and System Left scan operations. Each signature is 10 bits and all signatures have a value of 0b1001010110. The intent of the signatures is to help identify breaks in the scan ring. If any of the signatures are missing from their

dedicated bit positions in the scan ring, then the scan ring or operation is not correct. Scan software checks the signatures in each Board or System Left scan operation.

Also the Board Left and System Left scan modes have a "Last Left Shift" counterpart for exiting scan modes. The Board Left and System Left scan rings contain self-timed RAM components. The last left shift modes are used on the very last scan clock to allow the self-timed RAMs to exit scan mode.

8.2.2 Scan Initialization

At power up time, the NIA must be scan initialized. The CDS and XDS gate arrays are scan initialized with slice IDs. The clock generation logic must be initialized with the proper divide down pattern. The Physical Configuration Map (PCM) must be initialized with the memory configuration. Table 8-2 lists all of the initialized registers and fields on the NIA.

8.2.3 Soft Error Log

The NIA contains a soft error log for logging errors detected by the Pbus and NXP interfaces. The NIA's Soft Error Log is made up of the individual soft error logs from each I/O Channel interface. See the I/O Channel Interface chapter for a list of contents of the Soft Error Log (Table 3-2 on page 51). Each I/O Channel Interface has an eight bit register as part of the Soft Error Log. Each CDS gate array has a soft error log internal to the array and is scanned as part of the Soft Error Log. The CDS arrays contain parity error information to identify parity errors down to the byte level.

The Soft Error Log is scanned dynamically, meaning the clocks are not stopped prior to scanning. The Soft Error Log scan mode is selected while clocks are running. Then the NCU asserts the *xc_ia.slog_ena* signal to the NIA. The NIA registers this signal and uses it to enable the scanning operation of the Soft Error Log. The NCU asserts this signal for exactly 82 clocks in order to completely scan the Soft Error Log. The Soft Error Log should be cleared during the scanning operation.

8.2.4 CCU Scan

The NIA contains a 16 bit scan ring that is used to scan CCUs. The CCU Scan Ctl ring is scanned in preparation for scanning a CCU. The CCU Scan Ctl ring contains the CCU output data enables, dmode and CCU scan control signals used to scan CCUs. The CCU Scan Ctl ring is defined in Table 8-3

In order to scan a CCU the following events must occur. First the CCUs' clocks must be turned off. This is accomplished by resetting the CCU's run bit enable. The NIA receives eight CCU run bit enables, *xc_ia.ccu_rbe<7..0>*, to control the CCU's clocks. During system level debug, all of these bits would normally be reset at the same time along with the NIA's clocks in order to stop the NIA and I/O subsystem at the same time. The Crossbar and memory clocks would also be stopped if one was debugging a system level problem. Note that only the 2x clock is stopped, the 3x clock should continue to run in order to preserve volatile CCU data.

Once clocks are stopped, the CCU Scan Ctl ring is scanned with the desired scan control. Only one CCU can be scanned at a time. The NIA examines the CCU scan control to determine which CCU is being scanned. This allows the NIA to select the

Table 8-2 NIA Scan Initialization

cds7..0:slice_id	0x7..0x0	CDS array slice IDs
cds7..0:toc	0x0008feff16010008	Time of Century header
cds7..0:toc_par	0b10010010	TOC header parity
cds7..0:mbp	0x0008feff16010190	Memory Base Pointer header
cds7..0:mbp_par	0b10010001	MBP header parity
cds7..0:int	0x0008fdff06000040	Interrupt header
cds7..0:int_hdr_par	0b10011110	Interrupt header parity
cds7..0:int_dat	0x0000000000000800	Interrupt data
cds7..0:int_par??	0b11111101	Interrupt data parity
cds7..0:par_msk	0x11111111	Read Data Parity Mask
cds7..0:rd_shadow	0x0000000000000000	Read Data Shadow register
cds7..0:rd_shadow_par	0b11111111	Read Data Shadow parity
xds11..00:slice_id	0x3..0x0	XDS array slice IDs
xds11..00:b7..0†	(config dependent)	log to phy board select map
xds11..00:swap†	(config dependent)	memory interleave
xds11..00:sc_bd†	(config dependent)	physical configuration map
xds11..00:int_ptr	0x0000	interrupt pointer que addr
xds00:fl3..0_cmp0	0x2f	Pbus3..0 wdq ready compare
xds00:fl3..0_cmp1	0x04	Pbus3..0 wdq full compare
xds10:fl3..0_cmp0	0x3f	Pbus3..0 rdqe ready compare
xds10:fl3..0_cmp1	0x04	Pbus3..0 rdqe full compare
xds11:fl3..0_cmp0	0x3f	Pbus3..0 rdqo ready compare
xds11:fl3..0_cmp1	0x04	Pbus3..0 rdqo full compare
xds01:fl1_cmp0	0x1f	NXP wdq ready compare
xds01:fl1_cmp1	0x04	NXP wdq full compare
xds01:fl2_cmp0	0x3f	NXP rdqe ready compare
xds01:fl2_cmp1	0x04	NXP rdq full compare
xds01:fl3_cmp0	0x3f	NXP rdqo ready compare
xds11..00:msk_dat_par	0b1111	data parity error mask
xds11..00:msk_hdr_par	0b1111	header parity error mask
xds11..00:par_err	0b0000	XDS parity error state
ck_pcm_31_0†	(config dependent)	physical configuration map
pxfr_limit	0x080	Pbus transfer limit
nxfr_limit	0x100	NXP transfer limit

† - initialized by mminit

appropriate CCU scan return data to the NCU. After the CCU Scan Ctl ring is scanned, the Scan Engine will assert the run bit enable of the CCU to be scanned. The CCU's run bit enable is asserted for the amount of time required to scan the CCU. The CCU is scanned at the Pbus clock rate (10 Mhz). Scan data to or from the CCU is captured on the NIA in Pbus clock qualified registers. The Scan Engine uses an internal cycle counter to know when to sample return scan data. Once the

CCU has been scanned, and no other CCU scan operations are desired, the Scan Engine must scan the CCU Scan Ctl ring to the "inactive" state in order to return the CCUs to their normal mode of operation.

After the scan operation is complete, the CCU run bit enables are re-asserted along with the NIA's 2x clocks. This action starts the NIA's clocks and the I/O subsystem's clocks at the same time

Table 8-3 CCU Scan Ctl Ring

R.CCU00_ODENA*	msb
R.CCU01_ODENA*	
R.CCU10_ODENA*	
R.CCU11_ODENA*	
R.PBUS01_DMODE*	
R.PBUS03_CCSCTL<1>*	
R.PBUS03_CCSCTL<0>*	
R.PBUS01_LDRAM*	
R.CCU20_ODENA*	
R.CCU21_ODENA*	
R.CCU30_ODENA*	
R.CCU31_ODENA*	
R.PBUS23_DMODE*	
R.PBUS23_LDRAM*	
Not Used	
Not Used	lsb

8.3 Hard Errors

The NIA detects several hard error conditions across the board. The individual hard error indications are group (ORed) together in the Clock logic block and are reported to the NCU as an NIA hard error, *ia_xc.hard_error*. Table 8-4 lists the different hard error indications on the NIA. Once a hard error condition is detected, the NIA will freeze all state associated with the hard error and then report the hard error condition to the NCU. The NCU will then shut off clocks to the entire system (if enabled) and report the error condition to the SPU. At that point, the SPU can interrogate the system with the hard logger to determine the exact cause of the hard error condition.

Table 8-4 NIA Hard Errors

xdsXY.par_err -	XDS arrays detected a header parity error or a data parity error on data from WDQ
cdsX.rdq_perr -	CDS arrays detected read data parity error on data from the RDQ.
rqa.rdq_wrt_pe -	Parity error detected on write data during write access to the RDQ.
wqa.wdq_wrt_pe -	Parity error detected on write data during write access to the WDQ.
pi.hard_error -	Pbus interrupt state machine hard error. CCU unexpectedly dropped its interrupt request.
nxi.hard_error -	NXI interrupt state machine hard error. XIOP unexpectedly dropped its interrupt request.

Table 8-4 NIA Hard Errors continued

pa.wdq_fflag_pe -	WDQ flush flag parity error.
rqa.rdq_flag_pe -	RDQ read error flag parity error.
rqa.rd_par_err_e -	Parity error on read data from even side crossbar.
rqa.rd_par_err_o -	Parity error on read data from odd side crossbar.

9 Data Flows

The intent of this chapter is to describe the various data flows through the NIA. This chapter is organized into three sections: write data transfers, read data transfers, and interrupt transfers. This chapter can be used to tie together the functional description of the NIA's major subsystems or as an introduction to the NIA's data paths.

9.1 Write Transfers

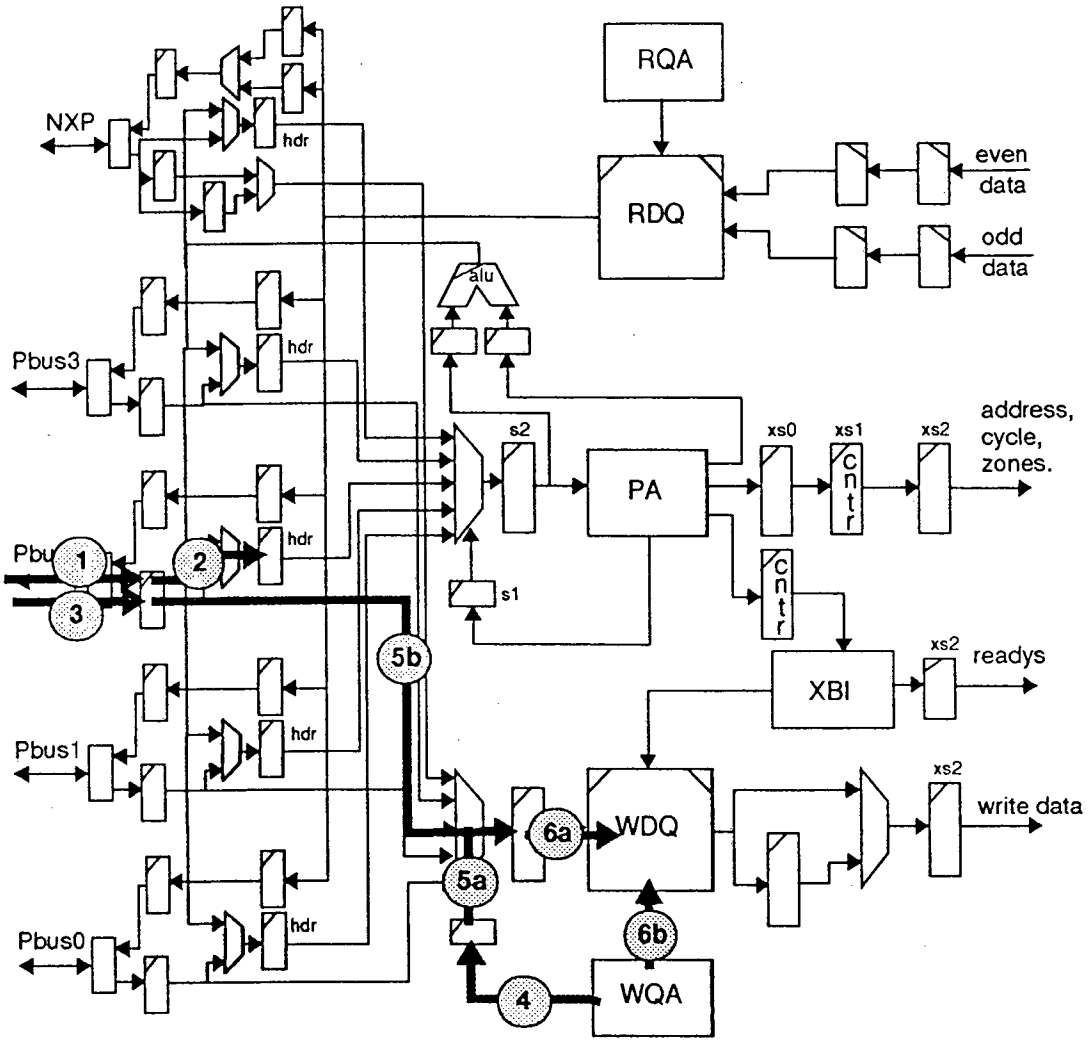
Write transfers describe the movement of data from a CCU into main memory. Thus the CCU is "writing" to memory. As with all transfers, read or write, a CCU initiates the process by making a request to the NIA. The request is made in the form of a header transfer to the NIA. See 1 in Figure 9-1.The header transfer defines the type of transfer (read or write), the starting address of the transfer and the total number of bytes involved in the transfer. For a write transfer header, the address defines where to begin writing in main memory and the byte count defines the number of bytes being written.

The header transfer (data) is stored in a header register implemented in the Channel Data Slice (CDS) arrays. The header register is initially loaded under control from the respective Pbus X or NXP interface logic. See 2 in Figure 9-1. Of course prior to loading the header register, the PBIx or NXI logic has arbitrated access to the respective Pbus or NXP bus and has sequenced through the proper states to arrive at the header transfer state. This is action is described in chapter 3, I/O Channel Interface, on page 45.

After the header has been received and verified (no errors), the CCU will send write data to the NIA (see 3 in Figure 9-1). The number of write data transfers sent to the NIA is a function of the starting address and the number of bytes specified in the header transfer. As the NIA receives write data from the CCU, the Write Queue Arbitration logic (WQA) schedules write accesses to the Write Data Queue (WDQ). The WDQ provides temporary storage for the CCU's write data until it is written to memory. Access priorities are determined by the WQA logic. The WQA selects the I/O channel write data register (see 4 through 6 in Figure 9-1). The arbitration and access to the WDQ is described in chapter 4, Write Data Queue and Arbitration, on page 71.

All of the CCU's write data must pass through the WDQ on it's way to memory. Each write data transfer is checked for good parity as it is received from the CCU. At some point, one of two things will occur: 1) the CCU will complete it's transfer by either sending all of it's write data or by aborting early, or 2) the CCU will send enough write data to the NIA to trigger a memory access to the crossbar. Seventeen longword transfers are required to trigger this event for a Pbus channel, and 33 longword transfers are required for the NXP. The Port Arbitration (PA) logic is notified by the I/O channel's WDQ fill level monitor (implemented in the Crossbar Data Slice or XDS arrays). The fill level monitor is scan initialized to be asserted when the I/O channel's WDQ reaches a certain fill level. Once the PA logic is notified, it will consider that I/O channel in the next round of arbitration to the crossbar interface.

Figure 9-1 Header and Write Data Transfers



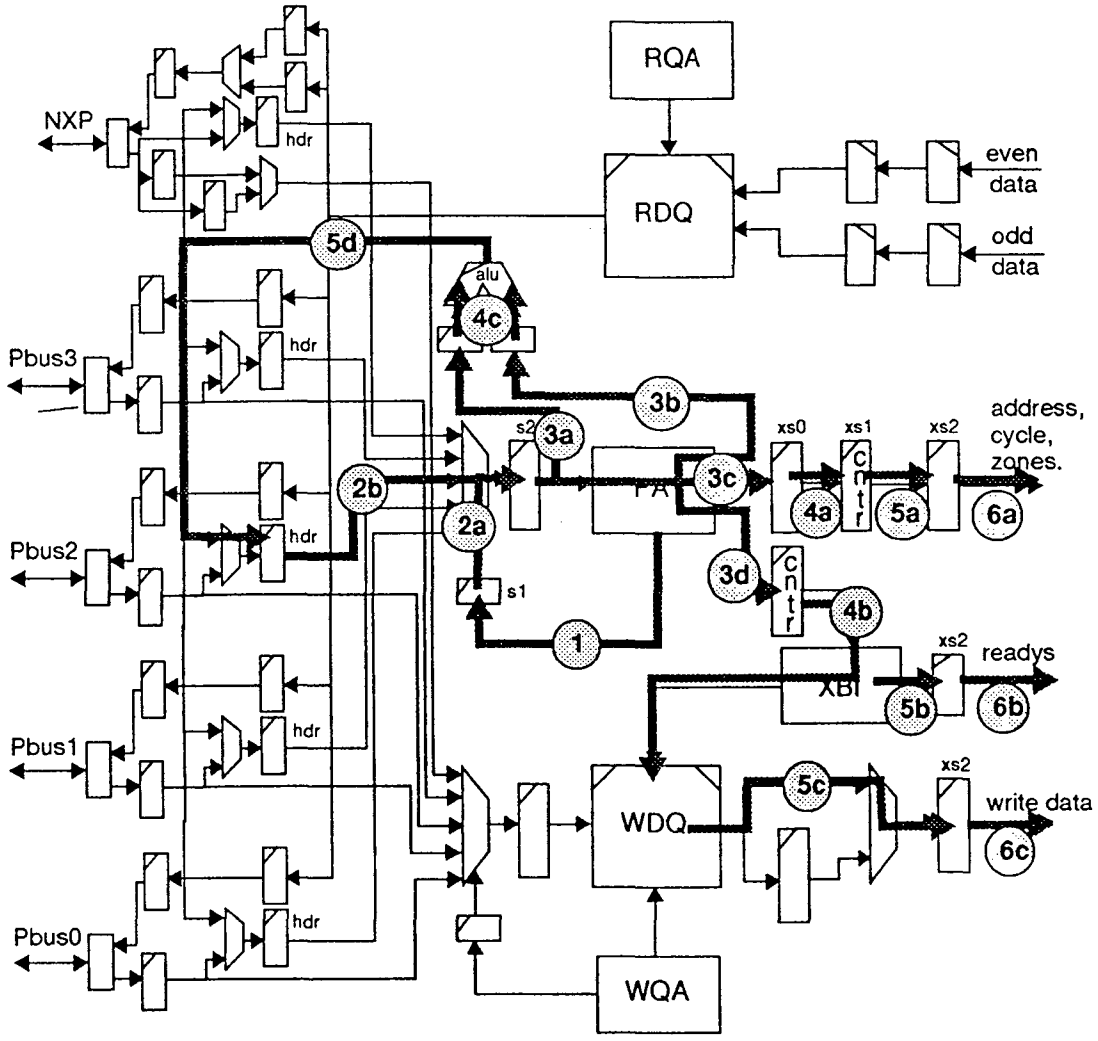
If the CCU aborts early or sends fewer than the 17 or 33 long words required to assert the fill level monitor, the PBIx or NXI interface logic will notify the PA logic directly. The PA logic will in turn set a "write complete" state for that I/O channel and will at that time consider the I/O channel in the next round of arbitration. The PA arbitration process is described in chapter 6, Port Arbitration, on page 87 .

When the I/O channel "wins" the arbitration process, the PA logic will select that I/O channel's header from the CDS arrays. The header selects sent to the CDS arrays and are registered into "stage 1"(S1) of the arbitration pipeline (see 1 in Figure 9-2). The selects drive an 8:1 multiplexor in the CDS arrays which is used to select the desired I/O channel header. The selected header is sourced off the CDS arrays and into a "stage 2" (S2) register in the PA logic (2a & 2b in Figure 9-2). Stage 2 of the arbitration pipeline is used to calculate a partial byte count and to generate a fetch count.

The partial byte count defines the number of bytes that will be sent to memory during the header's time slice at the crossbar interface. Time slice may be a slight misnomer in that each channel gets to transfer up to X number of byes to memory at a time. There is no "time limit" on how long the transfer of X bytes may take. X

is equal to 128 bytes for a Pbus channel and 256 bytes for the NXP. The partial byte count is used to update the selected header in the CDS arrays (see 3b).The fetch count is related to the partial byte count. It represents the number of Crossbar transfers that are required to service the header transfer. It also represents the number of read operations that are needed to fetch write data from the WDQ. The fetch count value in loaded into a counter when the header advances into the next stage (see 3d).

Figure 9-2 Header Selection and Write Data Transfer Pipeline



From the S2 stage the header will advance into the crossbar interface stage 0 (XS0, see 3c). The XS0 stage (and the next two stages, XS1 and XS2) are implemented in the XDS arrays. The XS0 stage buffers the header address and transfer field. The fetch counter in the PA logic is loaded with the fetch count and, in the case of memory write transfers, a read request is kicked off to the WDQ (4b). The read request is controlled by the Crossbar Interface logic (XBI). The header then advances to the XS1 stage and also remains in the XS0 stage (4a). In fact, the header transfer will at some point occupy all three crossbar interface stages: XS0, XS1 and XS2. Meanwhile, the CDS arrays are updating the selected header (4c).

In the XS1 stage the header address is loaded into an address counter. The transfer

type is staged and encoded to the appropriate cycle bits (5a). If there are no hold conditions, the XBI logic will generate the request ready signals (5b). The read request that was kicked off in the XS0 stage is clocked into the WDQ at the beginning of the XS1 stage and the read is performed (5c). The output from the CDS header update is now valid and ready to be clocked into the selected I/O channel header register (5d).

Next, the header advances to the XS2 stage where the first transfer address is presented to the crossbar (6a) along with the request ready signals (6b). The write data from the WDQ is registered and presented with the write address (6c). The write data from the WDQ is registered and parity checked in the XDS arrays.

The XS0, XS1 and XS2 stages work together to service the header transfer at the crossbar interface. The fetch counter in the XS0 stage is decremented as write data is read from the WDQ and the transfer is made to the crossbar. The XS1 stage contains the address counter where the address is incremented with each crossbar transfer. The output from the address counter and the WDQ are then captured in the XS2 stage where address and data are presented to the crossbar.

The crossbar stages continue to operate in this fashion until the fetch count reaches zero. At that point the last read request has been made to the WDQ. On the next clock cycle, the address count will have incremented to the last write address and on the next cycle, the last address and write data are presented to the crossbar. The XS0, XS1 and XS2 stages can be filled with another header transfer as soon as they are finished with the current transfer, thus allowing the end of one header transfer and the beginning of another header transfer to reside in the crossbar stages at one time.

As soon as the header transfer has cleared the XS1 stage, the PA logic can then reconsider that I/O channel for arbitration of another "slice" of transfers to the crossbar. If there are no more transfers left to be performed, then the I/O channel interface logic is allowed to grant the channel to another requesting CCU.

9.2 Read Transfers

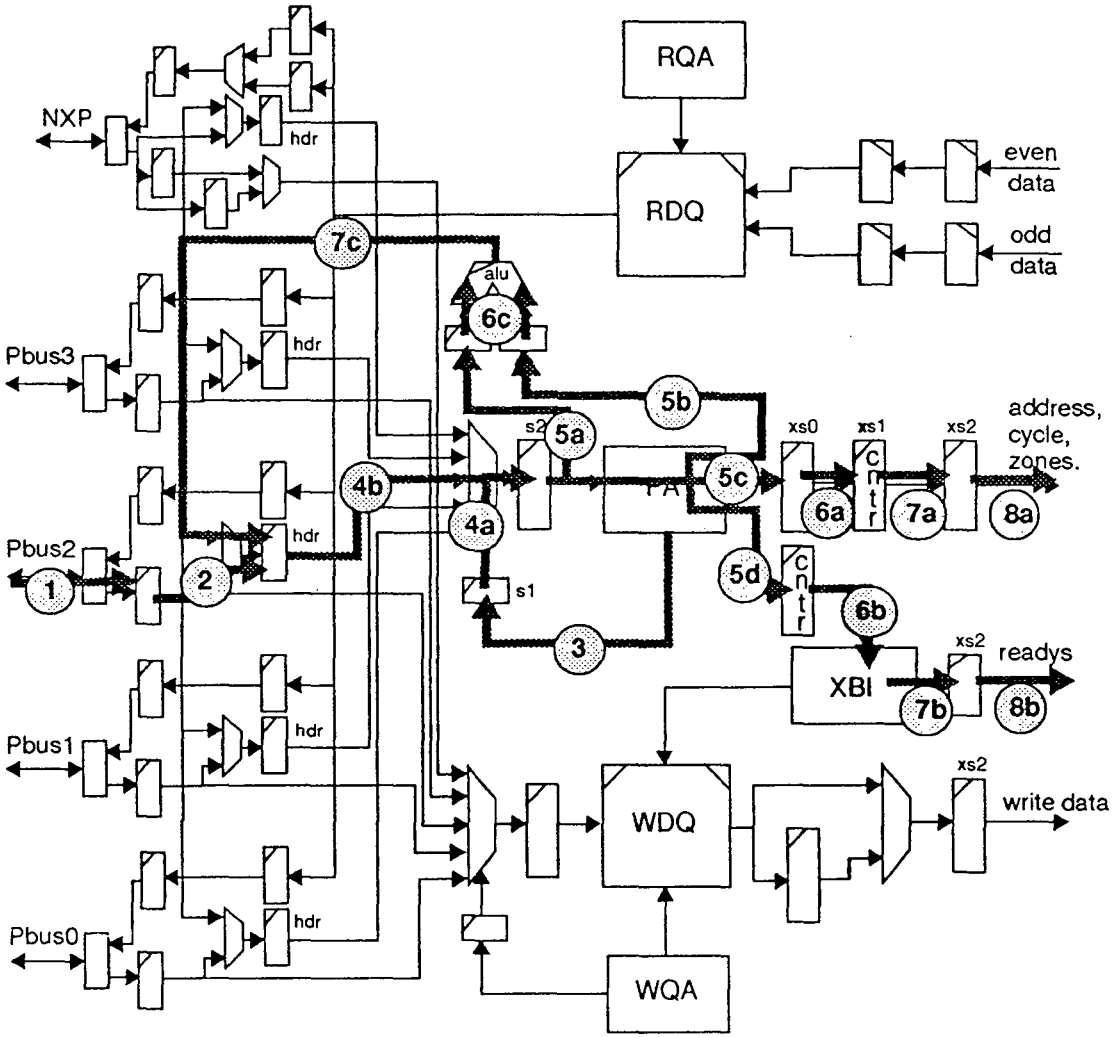
Read transfers describe the movement of data from main memory to a CCU. Thus the CCU is "reading" memory. The read process in initiated by the CCU making a request to the NIA. The request is made in the form of a header transfer to the NIA. See 1 in Figure 9-3. For a read transfer header, the address defines where to begin reading in main memory and the byte count defines the number of bytes to be read. The NIA does not align bytes for a CCU. If a fraction of a long word is requested, the NIA returns the complete long word of data and the CCU must select the desired byte(s) within the returned long word.

The header transfer (data) is stored in a header register implemented in the CDS arrays. The header register is initially loaded under control from the respective Pbus X or NXP interface logic. See 2 in Figure 9-3. Prior to loading the header register, the PBIx or NXI logic has arbitrated access to the respective Pbus or NXP bus and has sequenced through the proper states to arrive at the header transfer state.

Once the read header transfer has been received and verified (no errors), the I/O channel interface informs the Port Arbitration (PA) logic that a read request has been received. The PA logic will at that time consider the I/O header in the next

round of arbitration. When the I/O channel "wins" the arbitration, the PA logic sources the appropriate header select to the CDS arrays (3 in Figure 9-3). The selects are staged into the S1 register and then drive the selects to the 8:1 header multiplexor which drives the header off the CDS arrays (4a and 4b). The header is registered by the PA logic in an S2 register.

Figure 9-3 Read Header Transfer and Header Selection



Several things occur during the S2 cycle. First the selected header is internally staged by the CDS arrays (5a). Second, the PA logic calculates a partial byte count value and sources this value back to the CDS arrays (5b). Third, the PA logic calculates a fetch count value to load into the fetch counter (5d). Finally, the header address and transfer type is transferred to the XS0 stage (5c).

Next, the header advances to the XS0 stage. The header address and transfer type are captured into an XS0 register in the XDS arrays and are transferred to the input of the XS1 stage (6a). The fetch counter in the PA logic is loaded and the fetch count is made available to the XBI logic (6b). Since this is a read transfer header, no requests are made to the WDQ. Over in the CDS arrays, the selected header is updated by an arithmetic unit in the arrays (6c).

The header transfer will then advance to the XS1 stage where an address counter is loaded in the XDS arrays. The output of the address counter is encoded into the proper crossbar address and checked against the PCM mapping(7a). The transfer field is encoded into the proper crossbar cycle. If there are no transfer hold conditions, the XBI logic sources request ready signals to the input of the XS2 stage register(7b). The CDS arrays are completing the header update function and the updated header is sourced as an input the header register.

Finally, the transfer advances into the XS2 stage where the address and cycle are presented to the crossbar (8a) along with the transfer ready controls (8b). The register is updated in the CDS arrays. At this point, the XS0, XS1 and XS2 stages work together to process the header into transfers across the crossbar interface. When the fetch counter reaches zero, the last transfer will have been loaded into the XS1 stage and will then advance into the XS2 stage on the next cycle (assuming no hold conditions).

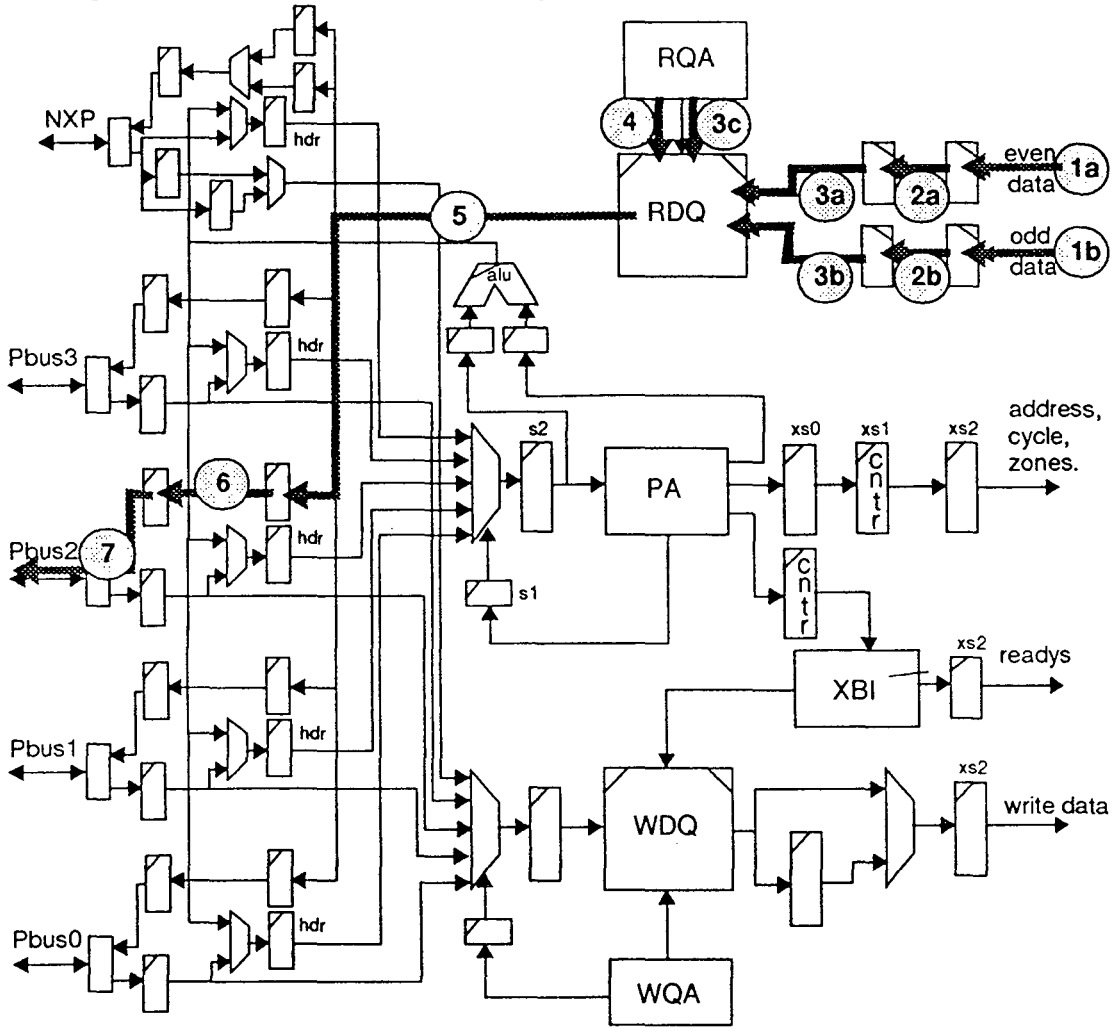
It takes approximately 19 clock cycles from the time a read request is made at the crossbar interface to the time that the data is returned to the NIA (assuming no memory contention). Memory is split into even and odd words. Depending upon memory contention, the even and odd words of a long word address may or may not return to the NIA on the same cycle. The NIA is designed to accept even and odd words on different cycles. Read data returns to the NIA are registered at the crossbar interface. See 1a and 1b in Figure 9-4.

Once the read data is registered from the crossbar, the NIA checks it's parity and stages the read data to be written into the Read Data Queue (2a and 2b). The Read Queue Arbitration (RQA) logic provides the write address and data to the Read Data Queue (RDQ) where the data is written on the cycle following the parity check (3a,3b,and 3c).

When both an even and an odd words are returned to the NIA, the RQA logic schedules a read access to the RDQ. The scheduled read access will cause the RQA logic to provide a read address to the RDQ at the appropriate time (4). The read address is clocked into the RDQ at the beginning of the next cycle and allowing read data to become valid at the mid point of the clock period (5). Both even and odd words are read at the same time. The read data is supplied to the CDS arrays where it is captured into a staging register of the selected I/O channel interface. The read data is also registered in a shadow register in the CDS arrays in order to check the read data parity.

The read data is held in the staging register until the next I/O channel interface clock edge where it will advance into the read data output register (6). From there, the read data will be driven back to the requesting CCU on the I/O channel interface bus (7). As read data returns to the NIA from the crossbar interface, this sequence of writing the data to the RDQ and then reading the RDQ and driving the data back to the CCU continues until all of the requested data has been returned. The NIA will return read data to the CCU in long word transfers only. Thus the NIA must wait for at least one even and one odd word to be written into the RDQ prior initiating an RDQ read and data return sequence.

Figure 9-4 Read Data Return Pipeline



9.3 Interrupt Transfers

Interrupt transfers are processed in two directions on the NIA. The NIA receives interrupts from CCUs and sends them to the NCU via the crossbar memory interface. The NIA also receives interrupts from the NCU and then sends them on to the CCUs. In both situations, the NIA plays a middle-man role by acknowledging the interrupt from the sender prior to forwarding the interrupt to the receiver.

9.3.1 CCU to NCU

Interrupts sourced from a CCU start out at one of the NIA's I/O channel interrupt busses. The NIA has two Pbus type interrupt busses and one NXP type interrupt bus. One Pbus interrupt bus is used with Pbus channels 0 and 1 in the left side CCU backplane, while the other Pbus interrupt bus is used with Pbus channels 2 & 3 in the right side CCU backplane. The NXP interrupt bus services the NCU and an optional XIOP.

The interrupt transfer begins by having a CCU request to use the interrupt bus. The NIA will grant the use of the interrupt bus to the requesting CCU when that

CCU has highest priority. Once granted, the CCU will drive its interrupt on the interrupt bus to the NIA. The NIA will register the interrupt vector from the CCU and complete the interrupt bus cycle. See 1 in Figure 9-5.

Next the I/O channel interrupt interface logic (PI or NXI) will assert interrupt ready to the Crossbar interface logic (XBI). The XBI logic then makes a request to the Write Queue Arbitration (WQA) logic to write the interrupt into the Write Data Queue (WDQ). The WQA logic schedules the interrupt vector write cycle and selects the interrupt data pattern from the CDS arrays (see 2 in Figure 9-5). The interrupt data pattern in the CDS arrays has the NIA's "processor" ID that is sent with the interrupt vector to the NCU (3a). The interrupt vector itself is multiplexed into the lower eight bits of the write data output from the CDS arrays (3b). The interrupt vector is staged and then written into the WDQ on the following cycle (4a & 4b).

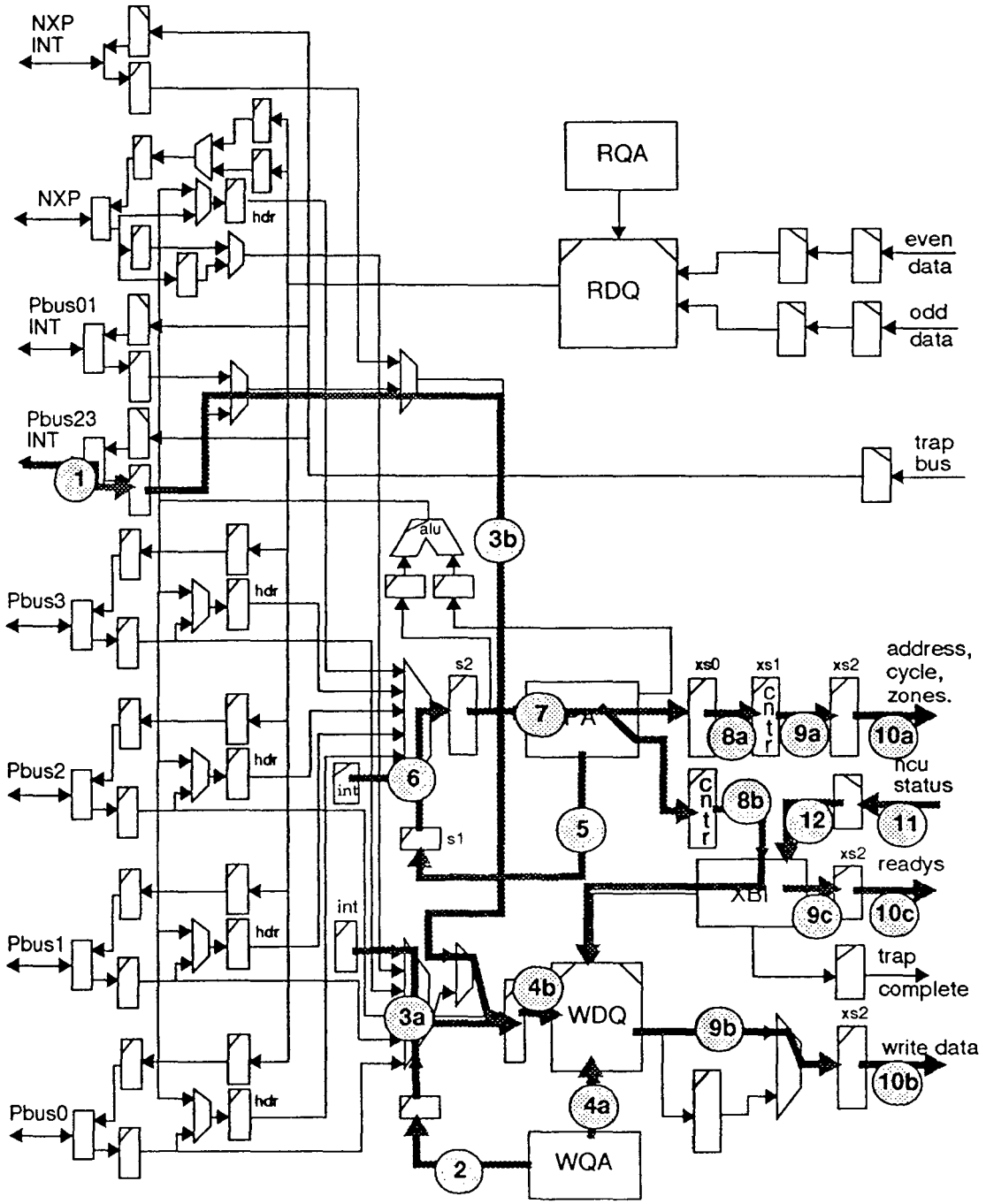
Once the interrupt vector is written into the WDQ, the XBI acknowledges the interrupt to the I/O channel interrupt interface. Once acknowledged, the I/O channel interrupt interface can grant the interrupt bus to anther requesting CCU. At this point the XBI logic makes a request to the Port Arbitration (PA) logic to send the interrupt to the NCU. The PA logic will select the interrupt header from the CDS arrays on the next available cycle (5). Interrupt requests to the PA logic have the highest priority. The header selects are registered in the S1 stage register in the CDS arrays. From there, the selects drive an 8:1 multiplexor where a scan initialized "interrupt" header is selected by the CDS arrays (6). The interrupt header is driven off the CDS arrays and is captured in the S2 header register located in the PA logic.

From the S2 register, the interrupt header will advance to the XS0 stage (7). When the interrupt reaches the XS0 stage, a read request is made to the WDQ similar to the case of a memory write transfer (8b). When the header advances to the XS1 stage (8a), the read request is clocked into the WDQ and the interrupt vector will be read (9b). Next the interrupt transfer advances to the XS2 (9a) stage along with the interrupt vector from the WDQ. The interrupt transfer looks a lot like a normal memory write transfer except the NCU board is selected and the trap register address is specified in the address field (10a & 10b).

The interrupt transfer lasts only once cycle at the crossbar interface. Once the interrupt transfer is presented, the XBI pipeline advances allowing a normal memory transfer to take place. The interrupt transfer may or may not be successful, depending upon whether or not the trap register on the NCU is currently busy or not. The interrupt transfer is acknowledged by the NCU in the form of two status control signals sent to the NIA (11 & 12). If the trap register was not busy, the NCU will send a "completed" status and the interrupt transfer was successful. If the trap register was busy, the NCU will send a "busy" status and the interrupt transfer must be retried. In the latter case, the XBI logic makes another request to the PA logic to send another interrupt transfer. The interrupt header will advance through the pipeline and eventually result in another interrupt transfer to the NCU (5 through 10). This cycle continues until the NCU sends a "completed" status to the NIA.

Once the interrupt is received, the NCU begins to process the interrupt. The NCU examines who (CPU, SPU or I/O) the interrupt is for and if necessary checks the interrupt mask level for the CPUs. At some point the NCU forwards the interrupt to the appropriate target(s) via a dedicated trap bus interface to all CPUs and NIAs in the system.

Figure 9-5 CCU to NCU Interrupt Transfer Flow



9.3.2 NCU to CCU

Interrupts from the NCU are originated by either a processor or an I/O device (CCU or SPU). Generally, a CCU will not sent an interrupt to another CCU. However the SPU, which looks like a CCU to the NIA, and the processors often send interrupts to the CCUs. All interrupts in the system are sent to the NCU first, and then from the NCU to the targeted device(s). When the NIA receives an interrupt from the NCU, it does not know who (CPU or SPU) sourced the interrupt. The NIA just forwards the interrupt from the NCU to the Pbus and NXP interrupt interfaces.

An NCU to CCU interrupt process starts out at the NIA's crossbar interface. The XBI logic receives the lower 8 bits (out of 12) of the trap vector bus from the XCL (crossbar board). A 4 bit trap type field tells the NIA that the trap bus contains an interrupt vector. The XBI logic registers the vector and trap type field. See 1 in Figure 9-6. If the trap vector is an interrupt, then the XBI logic notifies the Pbus interrupt interface (PI) and the NXP interrupt interface (NXI). See 2 in Figure 9-6. Once the PI and NXI blocks accept the interrupt, the XBI logic acknowledges the interrupt transfer back to the NCU by asserting a "trap complete" signal (3 & 4).

Once the interrupt has been received, the PI and NXI blocks will begin interrupt bus cycles on their respective interrupt busses(5, 6a & 6b). Once the interrupt has been delivered to the CCUs, the PI and NXI blocks can accept another interrupt from the XBI logic. Note: there is no hardware retry mechanism for an interrupt delivered to the CCUs. This includes cases where there is no CCU acknowledge from the interrupt bus cycle. Because multiple NIAs can exist in the system, the NIA must assume that an unacknowledged interrupt was destine for a CCU attached to another NIA. System Software must not send an interrupt to a CCU if it is busy and can not accept it.

Figure 9-6 NCU to CCU Interrupt Transfer Flow