C38xx Crossbar Specification

Document No. 500-001252-000

Revision 0.8
May 16, 1991

INTERNAL USE ONLY

PRELIMINARY

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Physical Description

The Neptune crossbar allows simultaneous transfers on the memory busses between up to nine processor-I/O ports and up to eight memory boards and the NCU. There are two independent crossbars in the Neptune system, an even and an odd, each handles 32 bit transfers between processors and memory. Together, the two independent crossbars can handle 64 bits of data per clock per processor-I/O port in both the store and return directions. Each crossbar consists of three boards, the crossbar return board (XRT), the crossbar send zero board (XS0) and the crossbar send one board (XS1). The Neptune crossbar consists of three even and three odd board sets for a total of six crossbar boards. The seventh board, the XCL, will be dealt with in depth in another document.

The XS0 board arbitrates between the processors and contains part of the send data path crossbar. The rest of the send data path crossbar is on the XS1 board. The XRT contains the return data path as well as control to insure data returns in the order it was sent.

The crossbar boards are installed in the central cabinet. Two XRTs, two XS0s, two XS1s and a XCL are needed in the central cabinet. Two power supplies (even and odd) provide the boards with -2.0 and -4.5 volts DC.

1.2 Major Subsystems

Since even and odd crossbars are identical and share no signals between them, most of the references to the crossbar in this document will refer to only one side. The functionality of the crossbar boards is subdivided into five major subsystems: arbitration, send crossbar, return control, return crossbar, and error capture. The first four subsystems are implemented by specific gate array types, the fifth by a combination of gate arrays and ECLiPS parts. The basic functionality of each is presented below with a detailed description to follow in later chapters.

Arbitration is performed by two XARB gate arrays which reside on the XS0 board. The XARB determines which processor gets access to a particular memory board if there is contention for a board. It also controls the input queues on the send crossbar. Each XARB can control access to four memory boards, and one XARB also controls access to the NCU.

The send crossbar consists of eleven SXBR gate arrays, four on the XS0 and seven on the XS1. Data from the processors is queued in the SXBR if there is contention. Multiplexors in the SXBR select data from these queues to each of the memory boards. The arbitration controls the selects for the multiplexors and the queues.

The return control is performed by one RCTL gate array on the XRT. The RCTL directs the return data path for read data returning from the memory and insures that each processor receives data in the order that it was sent. The RCTL will queue a communication register request if it might return before a slower memory operation.

The return crossbar consists of five RXBR gate arrays which reside on the XRT. Data from the NCU may be queued in the RXBR. A multiplexor selects data from the memory boards, the NCU, or the queue to each processor. The RCTL controls the selects for the multiplexors and queue.

The error capture logic resides on all three boards and insures that the crossbar is halted and data is captured if any processor or memory detects a parity error in the crossbar data. The crossbar does no parity checking and relies on keeping copies of the data sent to the other boards for comparison if a parity error is detected by the other boards. These copies are kept in the SXBR and RXBR gate arrays with the originating processor or memory board number stored in the RCTL or RXBR. This subsystem also contains the rest of the logic for the boards including the scan interface and the clock distribution.

1.3 Conventions and Abbreviations

All boards in the C3 (Neptune) system have three letter abbreviations which are refered to often in this document. These are listed in Table 1-1. In addition, signals going between boards specify a source board abbreviation, a destination board abbreviation, and a signal name. These signal name abbreviations are listed in Table 1-1.

Table 1-1 Board Abbreviations

Abbreviation	Signal Name Abbreviation	Board
XRT	XR	Crossbar Return Board
XS0	XS0	Crossbar Send 0 Board
XS1	XS1	Crossbar Send 1 Board
XCL	XC	Crossbar Control Logic Board
NCU	CU	Neptune Communications register and Utility Board (MB8)
NMB	MBx	Neptune Memory board. x=0 to 7
NSP	SPx	Neptune Scalar Processor Board. x=0 to 7
NIA	IA	Neptune Interface Adapter Board. (Treated as SP8)

For example, a 32 bit signal named WR_DATA going from scalar processor 2 to the XS1 board would be named SP2_XS1.WR_DATA<31:0> on the XS1 board.

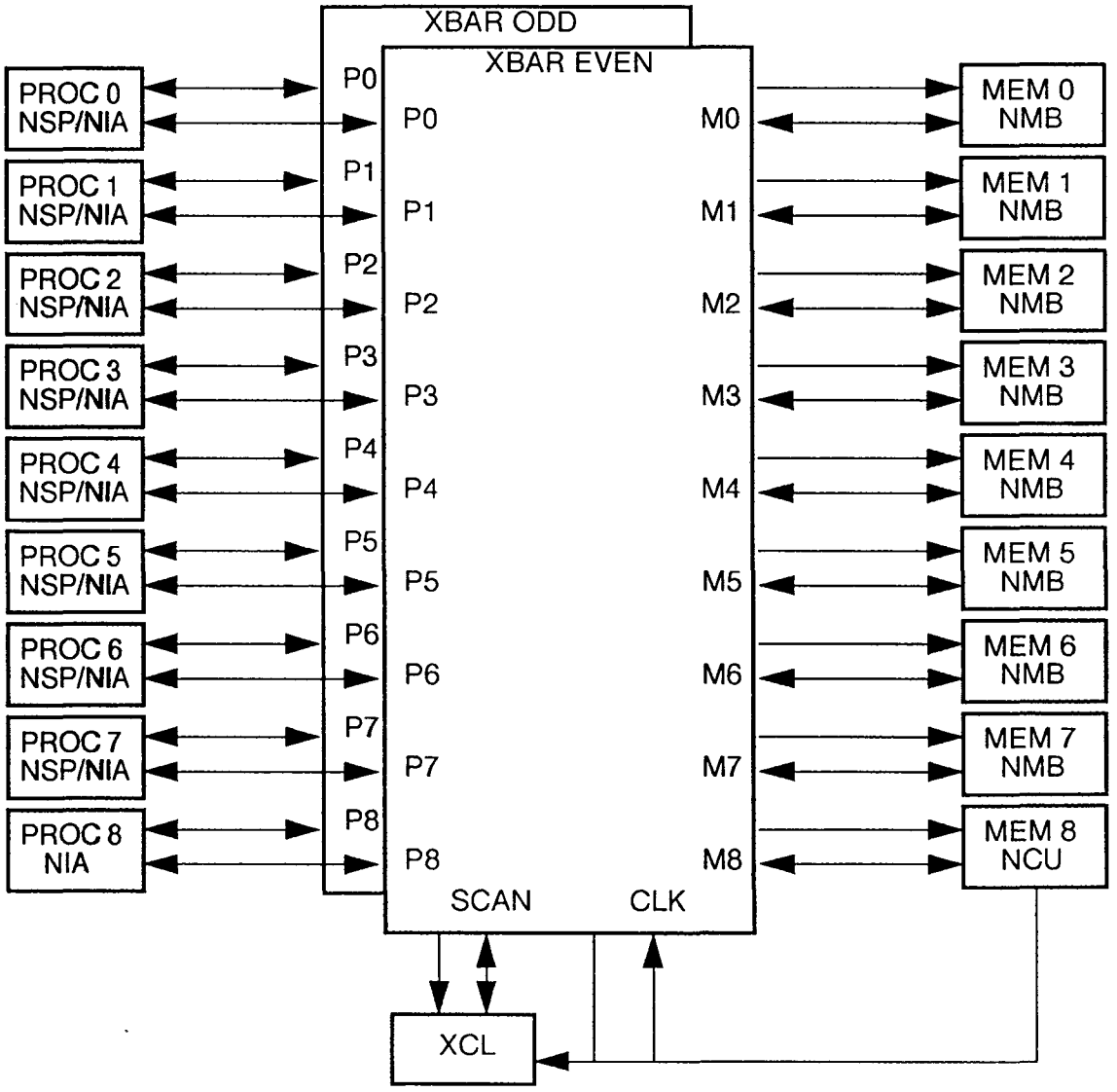
Note that <31:0> refers to bits 31 through 0. The notation <31..0> is equivalent. Note also that signals will necessarily be named differently on the backplane and other boards due to multiple occurrences of the same type of boards. Several boards use the abbreviation XSE and XSO to refer to signals going to the even and odd send crossbar (boards XS0 and XS1).

2 Crossbar Interfaces

The crossbar interfaces are presented in this chapter. The chapter starts by describing the interfaces of the crossbar to the rest of the system, followed by the interfaces between boards in the crossbar.

Figure 2-1 shows the external interfaces described in this chapter. The crossbar links up to nine processor-I/O ports with up to eight memory boards and the NCU which contains the communication registers. There are two independent crossbars, an even and an odd, each of which controls word requests from the processors to the memory. Hard errors and scan are staged through the XCL board, and clocks come directly from the NCU.

Figure 2-1 - Crossbar Interfaces



2.1 External Interfaces

The crossbar boards transfer requests from the processors to the memory and then data from the memory back to the processor. Each of the external interfaces are described below.

2.1.1 Processor Send Path Interface

The processor interface is used to receive memory/communication requests from NSP or NIA boards. Requests for both memory data and communication data are transferred over the interface. The signals which make up the interface are listed in Table 2-9. Note that for brevity only processor zero (SP0) is shown. There are nine processor interfaces total, SP0 through SP7 and the dedicated NIA interface signal named IA.

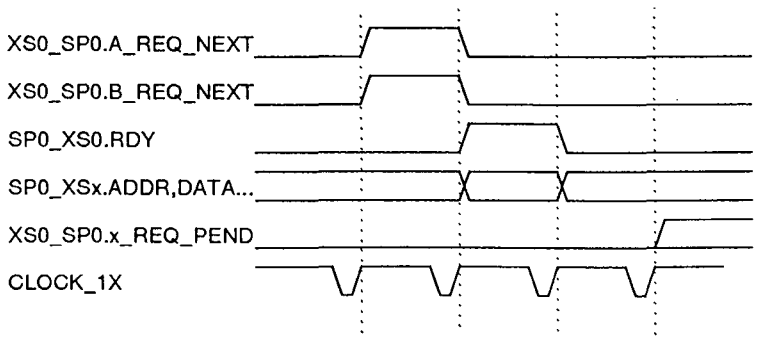
Table 2-1 Processor Send Interface Signals

SP0_XS0.ADDR<28:3>	Address for request
SP0_XS0.BD_SEL<3:0>	Memory/Communication board select
SP0_XS1.CTL_PAR<4:0>	Parity for address and control signals
SP0_XS0.CYCLE<1:0>	Type of request
SP0_XS0.RDY	Handshake signal from processor
SP0_XS1.WR_DATA<31:0>	Write data bus
SP0_XS1.WR_PAR<3:0>	Byte parity for write data bus
SP0_XS1.WR_ZONE<3:0>	Byte write enables
XS0_SP0.A_REQ_NEXT	Handshake signal to processor from XARB A (0)
XS0_SP0.A_REQ_PEND	Request pending signal
XS0_SP0.A_ST_PEND	Store request pending signal
XS0_SP0.B_REQ_NEXT	Handshake signal to processor form XARB B (1)
XS0_SP0.B_REQ_PEND	Request pending signal
XS0_SP0.B_ST_PEND	Store request pending signal

The signal SP0_XS1.CTL_PAR is parity for signals SP0_XS0.ADDR, SP0_XS0.CYCLE, and SP0_XS1.WR_ZONE (refer to the signal list appendix for specific parity bit assignment).

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

Figure 2-2 Processor Send Interface



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

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

The cycle field tells what type of memory operation is to be performed. These operations are listed in Table 2-2 below.

Table 2-2 Memory Cycle Codes

CYCLE<1:0>	Operation
0	NOP
1	READ
2	WRITE
3	TAM (Test and Modify)

The WR_ZONE field tells which bytes of the thirty two bit word are to be written to in a write or TAM memory request. The byte mapping is shown in the table Table 2-3 below

Table 2-3 WR_ZONE to Data Byte Mapping

WR_ZONE<0>	WR_DATA<31:24>
WR_ZONE<1>	WR_DATA<23:16>
WR_ZONE<2>	WR_DATA<15:8>
WR_ZONE<3>	WR_DATA<7:0>

The BD_SEL field selects which memory board to send the request to. A value of 0-7 selects memory boards 0-7. A value of 8 or 9 selects the NCU (communications registers). A value of 9 signals the arbitration that this request to the NCU is an operation that, once sent to the NCU, cannot be followed immediately by another request. A value of 9 is sent on a TAM request.

The mapping of parity bits to the data is shown in below. An even number of ones in data should have a one in the parity bit. Example: WR_DATA of 0xFFFFFFFF and WR_PAR of 0xF is good parity.

Table 2-4 Parity Mapping

Data	Corresponding parity
WR_DATA<31:24>	WR_PAR<0>
WR_DATA<23:16>	WR_PAR<1>
WR_DATA<15:8>	WR_PAR<2>
WR_DATA<7:0>	WR_PAR<3>
ADDR<28:22>	CTL_PAR<0>
ADDR<21:15>	CTL_PAR<1>
ADDR<14:8>	CTL_PAR<2>
ADDR<7:3>, CYCLE<1:0>	CTL_PAR<3>
WR_ZONE<3:0>	CTL_PAR<4>

The translation from the physical address word generated by the NSP or NIA to the XBAR address as used by the crossbar for memory boards is shown in Figure 2-3 below. ADDR bits are numbered <28:3> to reflect their original positions in the physical address. Bits <28:10> and <6:3> keep their original positions. Bits <9:7> in the physical address are used as board selects when the number of boards and thus the interleaving factor increases. Bits <6:3> are the memory board bank select. BD_SEL<3> is always zero for memory boards.

Note that physical address bits 2 through 0 are not used in the ADDR field. Bit 2 selects whether the address is to an even or odd word and the processor will send it to the even or odd crossbar. Bits 1 and 0 together with operand size are used by the processor to specify WR_ZONE.

Figure 2-3 Physical to XBAR Address Translation

Physical Address Bit					
31:29	28:10	9:7	6:3	2	1:0
BD_SEL or ADDR	ADDR	ADDR or BD_SEL	ADDR bank sel	Even or Odd word	Byte position

XBAR Address			
# Boards	BD_SEL<2:0>	ADDR<28:7>	ADDR<6:3>
1	<31,30,29>	<28:7>	<6:3>
2	<31,30,7>	<28:8,29>	<6:3>
4	<31,8,7>	<28:9,30:29>	<6:3>
8	<9,8,7>	<28:10,31:29>	<6:3>

In addition, the NMB decodes the XBAR address (ADDR<28:3>) field into a bank address, a RAM address, extra address bits for addressing 4 Meg DRAMs, and a row select for addressing multiple rows of DRAMs (the NMCs on the NMB can have two or four rows). The decoding table is shown in Figure 2-4 below. If the NMCs contain 1 Meg DRAMS then the 4 Meg Address must be zero or the NMB will report a hard error. If the NMCs are not fully populated (contain two rows not four) then bit 28 must be zero else the NMB will report a hard error.

Figure 2-4 XBAR to NMB Address Translation

ADDR<28:3>			
28:27	26:25	24:7	6:3
Row Select	4 Meg Address	RAM Address	Bank Address

2.1.2 Processor Return Path Interface

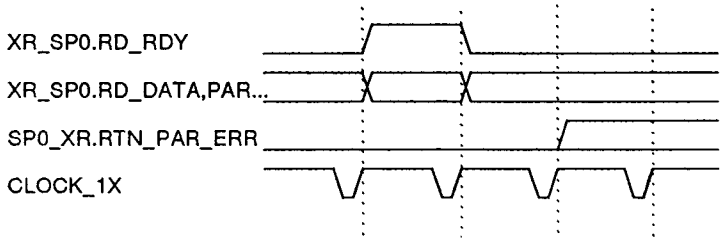
The processors receive memory/communication requests from the cross bar through the processor return path interface. The signals which are included in the interface are listed in Table 2-4. Note again that for brevity only processor zero (SP0) is shown instead of SP0 through SP7 and IA (processor eight).

Table 2-4 Processor Return Interface Signals

SP0_XR.RTN_PAR_ERR	Return data parity error
XR_SP0.RD_DATA<31:0>	Read data bus
XR_SP0.RD_PAR<3:0>	Byte parity for read data bus
XR_SP0.RD_RDY	Handshake signal for read data

Figure 2-5 illustrates the handshake mechanism used for the receive interface.

Figure 2-5 Processor Return Interface



The processor will always be able to accept the return read data as the processor never makes a read request without allocating a place to put the data. A transfer is completed when the RD_RDY signal is asserted. On the same cycle as the RD_RDY the data and parity are valid. The processor will assert the RTN_PAR_ERR signal two cycles after data with bad parity is received, and the return crossbar (XRT) will hold all registers one cycle after RTN_PAR_ERR is received. This will be just in time for the XRT to hold copies of the data and parity that was sent to the processor.

2.1.3 Memory Send Path Interface

The memory interface is used to send memory requests to the Neptune Memory Boards (NMB). The signals which make up the interface are listed in Table 2-5. Note that for brevity only memory zero (MB0) is shown. There are eight memory interfaces total, MB0 through MB7. Requests to the communication registers on the NCU are listed in the Communication Register Send Path Interface.

At the memory send path interface a request for a memory has been arbitrated from competing processors and the crossbar knows the state of the memory board and knows it can accept the request. A transfer is completed when the RDY signal is asserted. There is no return handshake from the NMB; the request is simply accepted. On the same cycle as the RDY the address, cycle, data, parity, zone, and ctl_parity signals are valid. The memory will assert the SEND_PAR_ERR signal two cycles after data with bad parity is received. The send crossbar (XS0 and XS1) will hold the last staged data registers one cycle after SEND_PAR_ERR is received. These registers contain the information that was sent to the memory so one can determine where data was corrupted. Note that no parity checking is done in the XBAR.

Table 2-5 Memory Send Interface Signals

XS0_MB0.RDY	Request Ready
XS0_MB0.ADDR<28:3>	Request Address
XS0_MB0.CYCLE<1:0>	Request Type
XS1_MB0.WR_ZONE<3:0>	Bytes to Write for WRITE or TAM cycle
XS1_MB0.CTL_PAR<4:0>	Parity for address, cycle, write zone
XS1_MB0.WR_DATA<31:0>	Data for WRITES and TAMs
XS1_MB0.WR_PAR<3:0>	Parity for WR_DATA
XS0_MB0.REF_REQ	Refresh request from crossbar
MB0_XS0.BANK_DONE<15:0>	Indicates a bank is finished with a request
MB0_XS0.SEND_PAR_ERR	Parity error on data sent to memory
MB0_XS1.SEND_PAR_ERR	Parity error on data sent to memory

The crossbar will assert the REF_REQ signal for one cycle when it wants the memory board to refresh its banks. See section 3.3.4 on page 3-7 for more details on refresh. MB*m*_XS0.BANK_DONE bit *b* is asserted by NMB *m* when its bank *b* will be finished with a request. See section 3.3.2 on page 3-6 for more details on bank done.

2.1.4 Memory Return Path Interface

The return data interface from the memory board is shown in Table 2-4. RD_DATA is the return data resulting from a READ or TAM request, the full thirty two bit word as specified by the ADDRESS passed with the RDY for this request regardless of WR_ZONE bits. That is, even if WR_ZONE was set to something other than all ones, a full thirty two bit word is returned. A single

byte can not be requested from the NMB. RD_PAR is mapped similarly to WR_PAR for WR_DATA. Note again that for brevity only MB0 is shown in Table 2-4.

Table 2-4 Memory Return Interface Signals

MB0_XR.RD_DATA<31:0>	READ Return Data Bus
MB0_XR.RD_PAR<3:0>	READ Return Parity
MB0_XR.RD_RDY	READ Return Data Ready

The RD_RDY signal is asserted when valid return data is present on RD_DATA. Since the NMB always returns RD_DATA a fixed number of clocks after RDY (thirteen clocks as currently programmed), the crossbar expects the return data at that time and uses the RD_RDY signal only for error checking.

2.1.5 Communications Register Send Path Interface

The communications register interface is used to send memory requests to the Neptune CPU Utilities (NCU) board which contains the communications registers. The signals which make up the interface are listed in Table 2-5. At this interface a request for a memory has been arbitrated from competing processors and the crossbar knows the state of the NCU and knows it can accept the request. A transfer is completed when the RDY signal is asserted. There is no return handshake from the NCU; the request is simply accepted. On the same cycle as the RDY the address, cycle, data, parity, and ctl_parity signals are valid. The NCU will assert the SEND_PAR_ERR signal two cycles after data with bad parity is received. The send crossbar (XS0 and XS1) will hold the last staged data registers one cycle after SEND_PAR_ERR is received.

Table 2-5 Communication Board Interface Signals

XS0_CU.RDY	Request Ready
XS0_CU.ADDR<28:3>	Request Address
XS0_CU.CYCLE<1:0>	Request Type
XS1_CU.CTL_PAR<4:0>	Parity for address and cycle
XS1_CU.WR_DATA<31:0>	Data for WRITES and TAMs
XS1_CU.WR_PAR<3:0>	Parity for WR_DATA
CU_XS0.SEND_PAR_ERR	Parity error on data sent to NCU
CU_XS1.SEND_PAR_ERR	Parity error on data sent to NCU

2.1.6 Communications Register Return Path Interface

The return data interface from the NCU is shown in Table 2-6. RD_DATA is the return data resulting from a READ or TAM request. RD_PAR is for parity checking on RD_DATA which is done the same way as the WR_DATA bus.

Table 2-6 Communications Register Return Path Interface

CU_XR.RD_DATA<31:0>	READ Return Data Bus
CU_XR.RD_PAR<3:0>	READ Return Parity
CU_XR.RD_RDY	READ Return Data Ready

The RD_RDY signal is asserted when valid return data is present on RD_DATA. Since the NCU always returns RD_DATA a fixed number of clocks after RDY (three clocks as currently programmed for both READ and TAM requests), the crossbar expects the return data at that time and uses the RD_RDY signal only for error checking.

2.1.7 Scan, Hard Error, Clock, and Refresh Interface

This interface connects the crossbar boards to the diagnostic engine on the NCU. The clock signals come directly from the NCU, all others are staged through the XCL. Table 2-7 lists the signals for the interface.

Table 2-7 Scan, Hard Error, and Clock Interface Signals

CU_XR.CLOCK_1X	62.5 MHz (nominal) clock for XRT board
CU_XR.CLOCK_1X*	62.5 MHz (nominal) clock negative
XC_XR.CONFIG_LOAD	Dynamic Configuration Load mode
XC_XR.SCAN_CTL<2:0>	Scan Control (only bits 1,0 used)
XC_XR.SCAN_IN	Scan Data input
XR_XC.SCAN_OUT	Scan Data output
XR_XC.HARD_ERROR	Hardware Error detected
CU_XS0.CLOCK_1X	62.5 MHz (nominal) clock for XS0 board
CU_XS0.CLOCK_1X*	62.5 MHz (nominal) clock negative
XC_XS0.CONFIG_LOAD	Dynamic Configuration Load mode
XC_XS0.SCAN_CTL<2:0>	Scan Control (only bits 1,0 used)
XC_XS0.SCAN_IN	Scan Data input
XS0_XC.SCAN_OUT	Scan Data output
XS0_XC.HARD_ERROR	Hardware Error detected
XC_XS0.REF_REQ	Refresh Request
CU_XS1.CLOCK_1X	62.5 MHz (nominal) clock for XS1 board
CU_XS1.CLOCK_1X*	62.5 MHz (nominal) clock negative
XC_XS1.CONFIG_LOAD	Dynamic Configuration Load mode
XC_XS1.SCAN_CTL<2:0>	Scan Control (only bits 1,0 used)
XC_XS1.SCAN_IN	Scan Data input
XS1_XC.SCAN_OUT	Scan Data output
XS1_XC.HARD_ERROR	Hardware Error detected

The XC_XS0.REF_REQ signal tells the XS0 board when to force refresh requests. While the signal is high, the XS0 will tell an NMB to refresh if its banks are idle. When the signal goes low, the XS0 will force a refresh on any outstanding NMB. XC_XS0.REF_REQ is free running and has a cycle of 15.36 microseconds or 960 clocks at 16ns. It is high for 860 clocks and low for 100 clocks.

Each board receives a differential clock. Each board outputs a hard error signal that will stop the system when a hardware error is detected by the board. Each board also has a scan interface to the diagnostic engine for looking at the internal state of the board. The interface consists of a data input SCAN_IN, a data output SCAN_OUT, and control fields. The control fields are listed inTable 2-8 below.

Table 2-8 Scan Control Decode

CONFIG_LOAD	SCAN_CTL<2:0>	Scan Operation
0	X00	Normal Mode
1	X00	Dynamic Configuration Load Mode
X	X10	CAST Load Mode
X	X11	Board Shift Left Mode

Scan modes are discussed in more depth in section 3.8 on page 3-11.

2.2 Internal Interfaces - Board to Board

2.2.1 XS0 to XS1 Interface

Signals from the arbitration are sent to the send crossbar gate arrays on XS1 by this interface. The signals on this interface are simply buffered copies of the signals sent to the send crossbar gate arrays (SXBRs) on the XS0. Requests from the processor go directly to the input staging registers of the SXBR, and then to register level 0 in the SXBR. The arbitration (XARBs) will assert PUSH*p* if it received a valid request from processor *p* that cannot immediately proceed, and the SXBRs will mark that request as valid and push it into the queue.

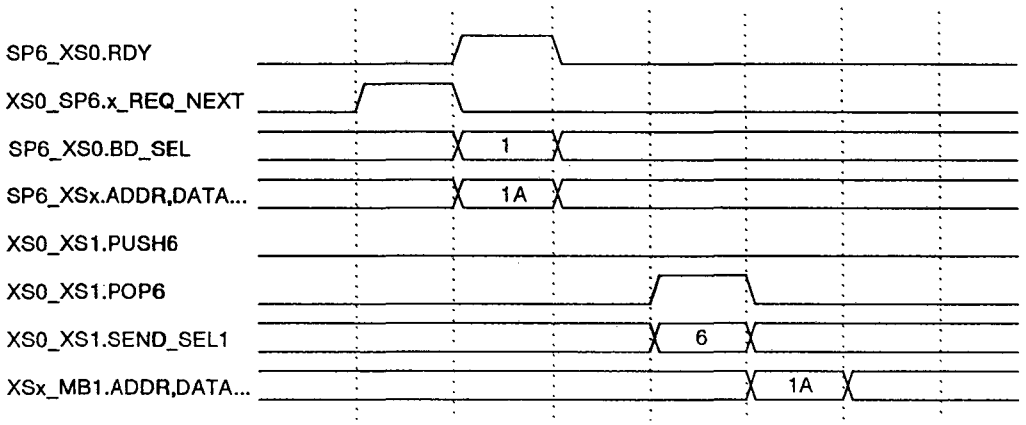
When the arbitration has determined a request from processor *p* can proceed to memory *m*, it asserts POP*p* and sets SEND_SEL*m* equal to *p*. The SXBR selects the head of the processor *p* queue into the memory *m* output staging register and pops the processor *p* queue.

Table 2-9 XS0 to XS1 Interface

XS0_XS1.PUSH0	Push Processor 0 request into SXBR queue
XS0_XS1.POP0	Pop Processor 0 request off SXBR queue
. . .	. . .
XS0_XS1.POP8	Pop Processor 8 request off SXBR queue
XS0_XS1.PUSH8	Push Processor 8 request into SXBR queue
XS0_XS1.SEND_SEL0<3:0>	Select processor request to memory board 0
. . .	. . .
XS0_XS1.SEND_SEL8<3:0>	Select processor request to memory board 8 (NCU)

Figure 2-5 shows the XS0 to XS1 Interface timing diagram for one request from processor 6 to memory board 1 which can proceed immediately.

Figure 2-5 XS0 to XS1 Interface Timing



2.2.2 XS0 to XRT Interface

Information about every request sent to memory is sent to the return crossbar by this interface. The signals on this interface are simply buffered copies of signals sent to the memory boards and a buffered copy of signals sent to the send crossbar gate arrays (SXBRs) on the XS0. When the arbitration has determined a request from processor *p* can proceed to memory *m*, it sets SEND_SEL*m* equal to *p*. Then next cycle the RDY signal to memory is set and the CYCLE bits are valid.

This information is buffered and sent to the XRT so it can determine when a read operation occurred. The XRT will then send the proper RD_DATA back to the processor that requested it. SEND_SEL tells which processor requested a memory board, RDY tells when the operation happened, and the bottom bit of the CYCLE field is set if it was a read request. Both bits are used

for requests to the NCU since the XRT can handle different return times for read and test and modify requests.

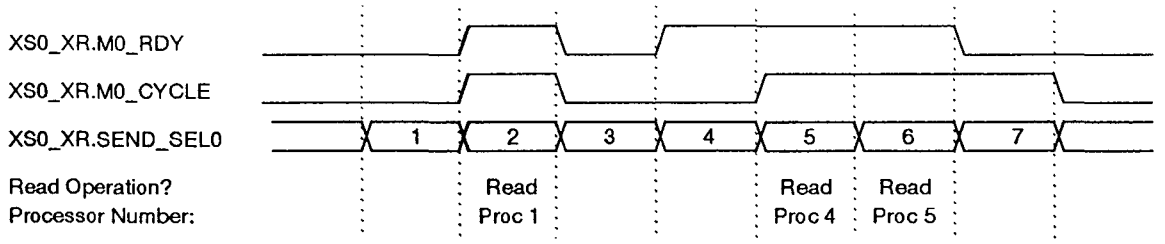
Table 2-10 XS0 to XRT Interface

XS0_XR.M0_RDY	Request went to memory board 0
XS0_XR.M0_CYCLE	Set if request to memory board 0 was a read
. . .	. . .
XS0_XR.M7_RDY	Request went to memory board 7
XS0_XR.M7_CYCLE	Set if request to memory board 7 was a read
XS0_XR.CR_RDY	Request went to memory board 8 (NCU)
XS0_XR.CR_CYCLE<1:0>	Cycle field of request sent to NCU
XS0_XR.SEND_SEL0<3:0>	Select processor request to memory board 0
. . .	. . .
XS0_XR.SEND_SEL7<3:0>	Select processor request to memory board 7
XS0_XR.CR_SEND_SEL<3:0>	Select processor request to memory board 8 (NCU)
XS0_XR.HALT_XRT	Halt the XRT. XS0 detected a hardware error.

The signal XS0_XR.HALT_XRT is used by the XS0 to halt the XRT which contains registered copies of the SEND_SEL fields. If a send parity error occurs, a memory board will halt the XS0 and XS1, and the XS0 will halt the XRT on the next cycle. The XRT will have halted in time to preserve a copy of the SEND_SEL field which contains which processor sent the request that had the parity error. See section 3.7 on page 3-9 for more details on error capture.

Figure 2-5 below contains a timing diagram for RDY, CYCLE, and SEND_SEL. NMB 0 is given as an example but the rest are similar. For memory boards, the XRT is only interested if a read type operation occurred.

Figure 2-5 XS0 to XRT Interface Timing



2.2.3 RCTL to RXBR Interface

This is an interface internal to the XRT board. The return controller (RCTL) uses this interface to control the return crossbar (RXBR) similar to the XS0 to XS1 interface. The RCTL directs the RXBR to select data to processor *p* from memory board *m* by setting RTN_SEL*p* to *m*. Data coming from the NCU may need to be queued to wait for previously requested data from a memory board. CR_PUSH pushes data from the NCU onto a queue, CR_POP pops if off the queue. RTN_SEL*p* set equal to 8 select the contents of the queue to processor *p*. With no requests in the system, all RTN_SEL signals will be set to 8. A code of 0xF selects the NCU and bypasses

the queue. The interface is shown in Table 2-11 below.

Table 2-11 RCTL to RXBR Interface

CR_PUSH	Push NCU Data and Par into queue
CR_POP	Pop Data and Par off queue
RTN_SEL0<3:0>	Select memory data to return to processor 0
RTN_SEL1<3:0>	Select memory data to return to processor 1
RTN_SEL2<3:0>	Select memory data to return to processor 2
RTN_SEL3<3:0>	Select memory data to return to processor 3
RTN_SEL4<3:0>	Select memory data to return to processor 4
RTN_SEL5<3:0>	Select memory data to return to processor 5
RTN_SEL6<3:0>	Select memory data to return to processor 6
RTN_SEL7<3:0>	Select memory data to return to processor 7
RTN_SEL8<3:0>	Select memory data to return to processor 8 (NIA)

3 Functional Description

The functional details for the crossbar boards XRT, XS0, and XS1 are presented in this chapter. The chapter starts by describing the crossbar dataflow in general, followed by detailing the functionality of each subsystem in the crossbar. Partitioning of functions between boards in the crossbar is mentioned and finally each board is described in detail.

The crossbar links up to nine processor-I/O ports with up to eight memory boards and the NCU. There are two independent crossbars, an even and an odd, each of which controls word requests from the processors to the memory. Since these are independent, all references to the crossbar in this chapter will refer to one side only.

3.1 Basic Crossbar Dataflow

3.1.1 Crossbar to NMB

Figure 3-1 shows the data paths used when a processor requests a read of a memory board with no conflicts. This example shows a transfer from processor 0 (SP0) to memory 1 (MB1). The registers in the figure are marked with the time that data is stored inside them. At time 0, the processor is ready to make a request. Assume the REQ_NEXT handshakes were set the previous cycle. At the next clock the request is stored in the XARB and SXBR registers (time 1).

At time 1, the memory request has made it to the crossbar and the processor 0 interface logic in the XARB looks at the request to determine if the bank is free. If so the request is sent to the memory 1 interface for arbitration. If processor 0 has the highest priority of all processors requesting memory board 1 then it will win arbitration and that result will be clocked into registers at time 2.

At time 2, the request has won arbitration and the destination bank has been marked as busy. The signals PUSH0 and POP0 will be set to one which informs the SXBR's state machine to not queue the request but send it on. SEND_SEL1 will be set to 0 by the arbitration logic which informs the SXBRs to select the request from processor 0 into the memory 1 output staging register next clock. The arbitration also sets a register that will set RDY to memory board 1 next clock.

At time 3, RDY is asserted to memory board 1, the request is sent to memory board 1, and the return control gate array (RCTL) has registered SEND_SEL1.

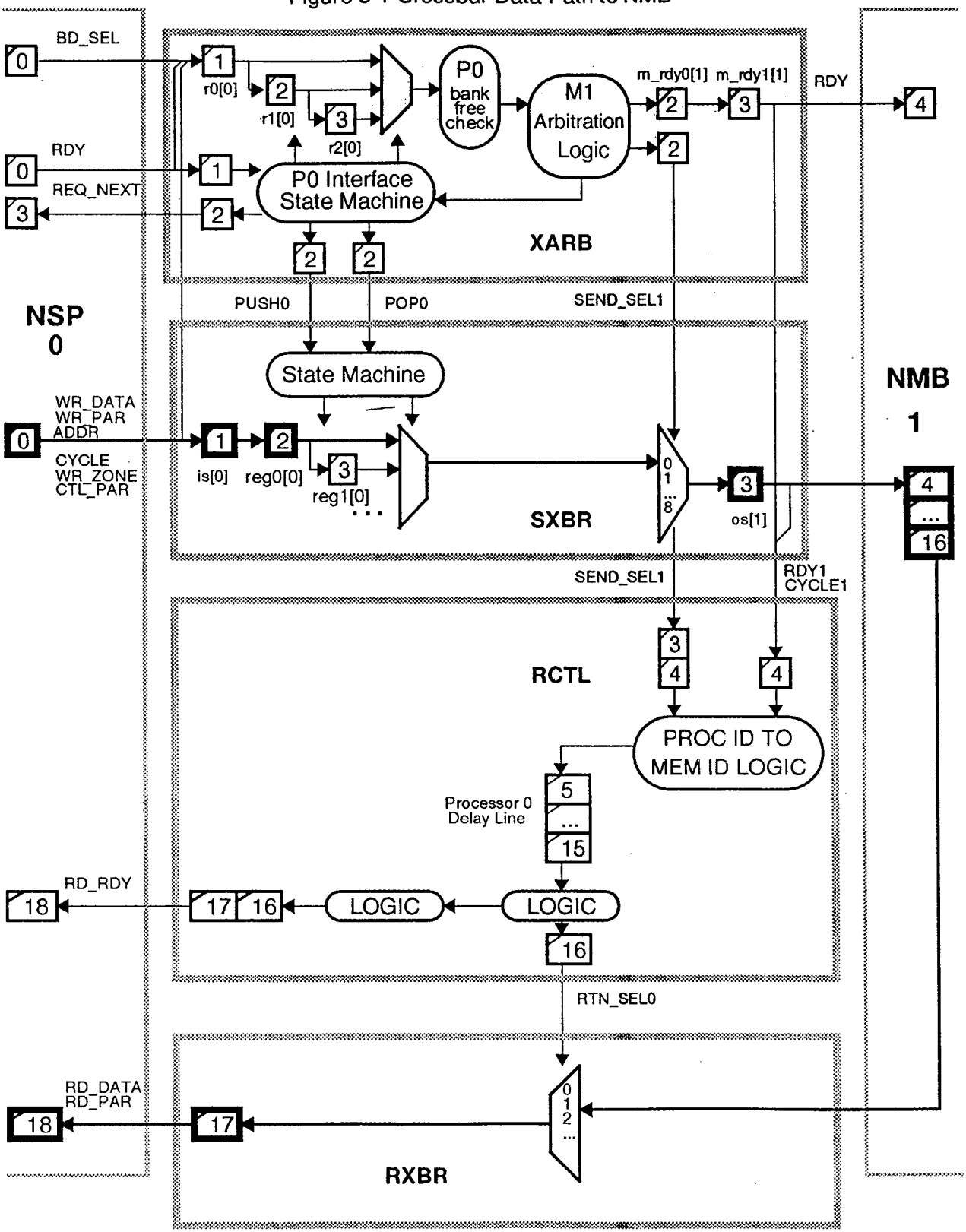
At time 4, the request is registered in memory board 1. The RCTL has registered the RDY to memory board 1 also. The RCTL decodes the registered SEND_SEL1 field and determines that processor 0 has made a read request to memory board 1.

At time 5, the delay line for processor 0 is set to expect read data from memory board 1 in a certain number of clocks.

At time 14, given the current DRAM speeds in the NMB, the BANK_DONE signal from the memory board to the arbitration will be asserted. The bank that was marked busy will be cleared on the next clock (time 15).

At time 16, the read data is on the bus from the memory board to the return crossbar (RXBR). Also, the RCTL's delay line for processor 0 now expects the data from memory board 1 and it selects the data from memory board 1 to go to processor 0.

Figure 3-1 Crossbar Data Path to NMB



At time 17, the read data is in a register in the RXBR and is sent on a bus to processor 0. The RCTL has also asserted RD_RDY to processor 0 which will accept the data next clock (time 18).

3.1.2 Crossbar to NCU

Figure 3-2 shows the dataflow of a processor read request to the NCU's communication registers with no contention in the arbitration and no queueing in the return path. The data path registers are marked in heavy outline to follow the data flow. This example is similar to the previous memory board example with the following differences:

Since there are no banks on the NCU, the arbitration at time 1 will always see that the bank is free. The memory arbitration takes place in memory interface 4 of XARB0, see section 3.10.1 on page 13 for more details on the arbitration partitioning.

The RCTL, at time 4, determines that the communication register request can bypass the queue and sets a bypass delay line (one register long with current NCU) at the next clock (time 5).

At time 5, the bypass delay line overrides the processor 0 delay line. Note that the bypass delay line register is marked with a heavy outline.

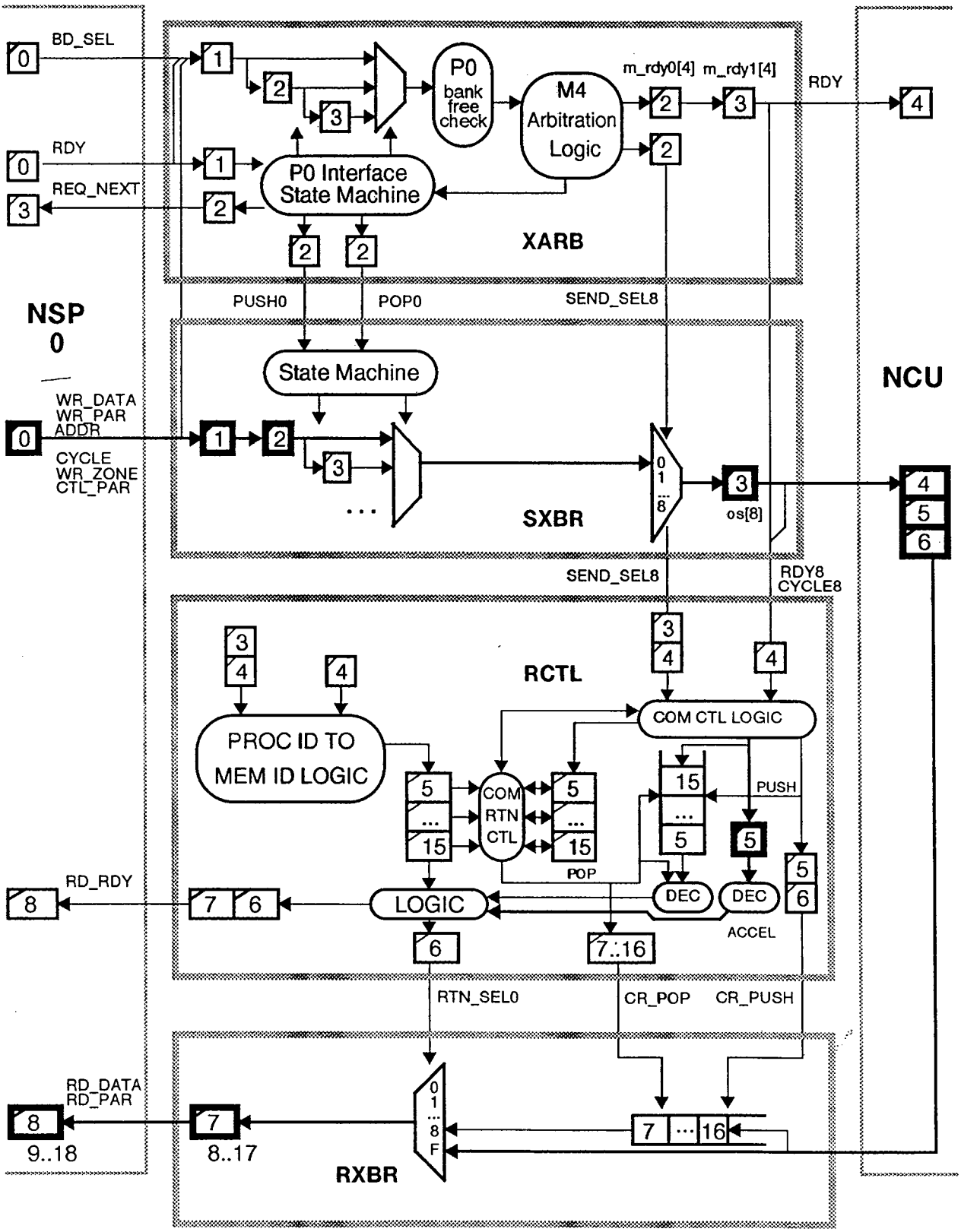
At time 6, the RTN_SEL0 signal selects the read data directly from the NCU (bypassing the queue) into the register destined for processor 0.

3.1.3 NCU Data Queueing

Note that a request to the NCU can return data to the processor in eight clocks whereas a request to a memory board takes at least eighteen clocks. NCU requests may have to be queued to guarentee that they do not return before the same processor's memory board request. When a request is queued, a push delay line is set instead of the bypass delay line at time 5. The CR_PUSH signal is set at time 6 and the read data is pushed into the queue in the RXBR at time 7.

The CR_POP signal is asserted when the data is ready to go. That could be time 7 through 16. The RTN_SEL signal always selects the queue, so the data will be in the outgoing register at time 8 through 17 and at the processor at time 9 through 18. Note that in the worst case an NCU request takes as long as a memory board request.

Figure 3-2 Crossbar Data Path to NCU



3.2 Crossbar Functionality

The crossbar is designed to allow nine processors to send memory requests to eight memory boards and the NCU. It is designed to handle large bandwidth from neptune processors which often request memory sequentially, arbitrate between them, send the requests on to the memory, and return read data to the processors with minimal latency and in the same order as the individual processor requested it. The throughput clock rate is 16 ns and the crossbar allows for independent even and odd word (32 bit) requests. In addition, the design allows for the use of faster DRAMs on the memory boards in the future.

As the clock rate is very fast, all the busses from the crossbar to the rest of the system are point to point and register to register, with one exception in the return crossbar. There are provisions in the arbitration to take let processors take advantage of the independent sequential memory banks. To simplify the design, the memory board always returns read data a fixed time after being sent a request, regardless of the type of read (read or test and modify) or whether it generated an ECC error. That fixed time is programmable in the hardware on the memory board and the return crossbar to take advantage of faster DRAMs in the future.

The crossbar functionality can be split into five subsystems: arbitration, send crossbar, return control, return crossbar, and error capture/diagnostics. These subsystems are described in detail below. The next paragraphs will describe how the subsystems work together.

Memory requests are initiated by the processor. The processor will never initiate a request for read data unless it has a place to put the data. The processor uses the RDY/REQ_NEXT handshake with the arbitration to send the request to the arbitration and the send crossbar using the processor send path interface (see section 2.1.1 on page 2-2 for more details). The arbitration determines if the request can proceed and where it will go. It directs the send crossbar to queue the request or send it to a memory using the XS0 to XS1 interface detailed in section 2.2.1 on page 2-9. A request is sent from the arbitration and send crossbar to memory using the memory send path interface detailed in section 2.1.3 on page 2-6. Transfers to the NCU use a similar interface (see section 2.1.5 on page 2-7). The error capture subsection is also involved in all of these interfaces except the XS0 to XS1.

Read requests (read and TAM) to memory are tracked by the return control. It receives its information from the arbitration and by snooping on the transfer to memory. The information is sent by the arbitration and send crossbar in the XS0 to XRT interface detailed in section 2.2.2 on page 2-10. The return data is received by the return control and the return crossbar using the memory return path interface detailed in section 2.1.4 on page 2-6. Return data from the NCU uses a similar interface (see section 2.1.6 on page 2-7). The return control directs the return crossbar to send data back to the correct processors and also to queue NCU data using the RCTL to RXBR interface detailed in section 2.2.3 on page 2-11. The return control and return crossbar use the processor return path interface (section 2.1.2 on page 2-5) to return read data to the processors. The error capture subsection is involved in all of these interfaces.

3.3 Arbitration

The arbitration subsystem consists of the processor interface (PINT), the memory interface (MINT), bank free tables, refresh, PCM error checking, and DPC logic.

3.3.1 Processor Interface

There is a PINT for each processor in the system. The PINT performs the handshake with the processor to receive memory requests. There is a three deep queue of requests inside the PINT to handle overruns from the processor. Overruns will happen when the arbitration has no requests and leaves REQ_NEXT high, the processor starts sending continous requests, and then the arbitration determines that a request cannot proceed. It will take one clock to turn off REQ_NEXT, and one clock before REQ_NEXT is received by the processor which then stops sending requests. Durring those two clocks two additional requests will be received. Figure 3-1 shows the overrun registers r0[0] through r2[0] for PINT 0 which store BD_SEL, RDY, CYCLE and BANK (ADDR<6:3>).

This handshake scheme is due to the register to register restriction due to the speed of the clock. There is no time to put logic that can receive a request, turn off REQ_NEXT that cycle and send it back to the processor, nor is there time to have the processor get the raw REQ_NEXT and turn off its request.

As the PINT receives and processes requests it directs the send crossbar to queue or send requests via the PUSH and POP signals. PUSH is asserted the cycle after a request arrives that cannot be immediately proceed. POP is asserted the cycle after a request has won arbitration. Note that with no contention the arbitration will send multiple POPs (no PUSH) which will be ignored by the send crossbar.

3.3.2 Memory Interface

The PINT sends the valid request at the head of its queue to its bank free check logic. If the request is for a bank that is free, then the request is sent on to the memory interface (MINT) for arbitration between PINTs. Arbitration is a modified round robin strategy. Each MINT has a rotating priority list (pselx[ ] where *x* is the MINT number) which gives each processor its turn for the highest priority. For example, if processor three has highest priority then processor four would be second and processor two would be last (priority would be 3,4,5,6,7,8,0,1,2). The priority list rotates forward every cycle to the next installed processor (a 0 in pselx[3] would move to pselx[4] if 3 and 4 were enabled) with four exceptions.

The first exception is burst mode when a processor can burst up to sixteen requests back-to-back to a memory board without being interrupted if each request is to a free bank. In this case the priority list will not rotate. The second exception is when the processor with the highest priority is blocked by a busy bank. The priority will not move, otherwise the processor might never be given access. The third exception is the interface to the NCU board (CINT). There are no banks to check on the NCU, but TAM requests to the NCU must be followed by a dead cycle. During this dead cycle the priority list will not rotate. The fourth exception occurs at the end of dynamic processor configuration mode (DPC) when all priority lists are reset giving highest priority to processor eight (the NIA). The priority list reset is done so a processor which was configured out during DPC will not end up with the highest priority.

When a request from PINT *p* has won arbitration in MINT *mx* for memory *m*, the next cycle m_rdy0[*mx*] is set to 1 which will set RDY the next cycle. Also, POP*p* will be set to one to tell the send crossbar to release the request if queued, and SEND_SEL*m* will be set to *p* to inform the send crossbar to send the request from processor *p* to the os[*m*] register for memory board *m*. In addition, the appropriate bank will be marked as busy in the bank free tables. The bank will be cleared by BANK_DONE signals from the appropriate memory board. An examples of these registers (except bank free) are shown in Figure 3-1 and Figure 3-2.

3.3.3 PCM Error

If m_rdy0[ ] is set for a memory board that is not installed (according to the valid memory board registers vmb[ ]) and error checking is enabled, then the arbitration will signal a physical configuration map (PCM) error. If a processor is disabled (p_config bit *p* set to 0 either from scan or DPC mode) then the r0[*p*] register for processor *p* will be held and cause the arbitration to see no requests. Note that if DPC mode is used to disable a processor while the system is running then there may not be any requests comming from that processor else a request could get held in r0[*p*] and be forever making requests.

3.3.4 Memory Board Refreshing

The arbitration also controls memory board refreshing. When the free running REF_REQ signal goes high, the arbitration will tell an NMB to refresh if that memory board is idle (no PINTs are sending requests to that MINT). When the REF_REQ signal goes low the arbitration will force a refresh on all outstanding NMBs. It does this by holding off any requests from the PINTs to that NMB, waiting untill the NMBs banks are free, and then telling the NMB to refresh. Requests from the PINTs continue after the refresh.

The arbitration tells an NMB to refresh by setting the REF_REQ signal to the NMB high for one cycle and setting all the banks of that NMB busy for the next seven cycles. After that, the banks will be cleared by BANK_DONE signals from the NMB. These BANK_DONE signals are guarenteed to come from the refresh request. BANK_DONE signals from earlier requests will come back during the seven cycles and will be ignored. For more intrigueing insights into the arbitration read the XARB Design Specification.

3.4 Send Crossbar

The send crossbar subsystem consists of nine processor interfaces (PINTs) and nine memory interfaces (MINTs) and DPC logic. Each cycle all the PINTs register WR_DATA, WR_PAR, ADDR, CYCLE, WR_ZONE, and CTL_PAR from the processors into the input staging registers (is[*p*] where *p* is the processor). If a processor is disabled via scan or DPC mode (config bit *p* set to 0 disables processor *p*) then the input staging register is held to prevent bad parity from going into the crossbar. The next stage is reg0[*p*] which loads from is[*p*]. By the time a valid request is in reg0[*p*], the arbitration has determined whether the request can proceed or whether it must be queued. By default a request proceeds and if there are no requests queued in the PINT then the state machine will set the mux select to send reg0[*p*] out to all the MINTs (see Figure 3-1).

If the request needs to be queued then the arbitration will set PUSH to 1 and the next cycle valid1[*p*] will be set to 1 and the request will be stored in reg1[*p*]. The request and associated valid bit will continue to be passed on in the queue until it hits the end (reg4[*p*]) in which case it stops, or until it stops behind a request with a valid bit set. Note that the only way valid0[*p*] will be set is when valid bits 1 through 4 are set and PUSH is set. In this case the PINT's state machine will hold reg0[*p*] and set valid0[*p*] the next cycle. When a POP signal is received, the highest numbered valid bit is cleared the next cycle. The PINT's state machine will set the mux select to send the highest numbered valid request out to all the MINTs.

MINT *m* receives SEND_SEL*m* which has been set to *p* by the arbitration. The next cycle, MINT *m* will select the output of PINT *p* to be put into the output staging register (os[*m*]) to be sent to memory board *m*. If memory board *m* detects a partity error then it will send a signal to the error capture logic which will hold registers in the send crossbar and a copy of the data sent to memory

board *m* will be in register lsd2[*m*].

3.5 Return Control

The return control subsystem consists of the processor id to memory id (PID2MID), the processor delay lines, error checking, and the communication register control with its assorted queue and delay lines.

3.5.1 Processor ID to Memory ID

The return control monitors the RDY, CYCLE and SEND_SEL signals for each memory board. The PID2MID logic takes valid memory read requests (RDY and CYCLE<0> both set) for the NMBs (memory boards 0 through 7) and the respective processor identification (SEND_SEL) and generates memory IDs to the processor delay lines for each valid request. Note that a processor will only generate one memory request per cycle and therefore only one memory ID per cycle (max) will be sent to a processor delay line.

3.5.2 Processor Delay Lines

The processor delay lines are of programmable length in anticipation of faster DRAMs in the NMBs. The examples used in Figure 3-1 assume a total of 13 register stages on the NMB; the return control can handle 6 to 17 register stages in the NMB. For a fascinating discussion of programmability in the return control read the RCTL Design Specification. When sent a valid memory ID *m*, processor delay line *p* will delay it for a fixed number of clocks and then instruct the return crossbar to send the data from memory board *m* to processor *p*. This is done by setting RTN_SEL*p* to *m*. In addition, a RD_RDY will be sent to instruct the processor to accept the data. Note that this works because memory boards all have a fixed read access time.

The memory boards send a RD_RDY to the return control but this is only used for error checking. If RD_RDY was set and the return control did not expect data or if RD_RDY was not set when the return control expected data then the return control will signal an RCTL hard error. All registers in the return control will halt when this signal is asserted.

3.5.3 Comm Reg control

Returns from the NCU are more complicated and are handled by the communications register control logic. When a valid request that returns read data is sent to the NCU, it is registered and then a decision is made whether or not to queue the return data. If there are returns in the queue already then the NCU data will be queued. If there currently is a memory request by the same processor to an NMB that will return data on or after the NCU returns data then the NCU data will be queued and sent to the processor after the NMB data so it arrives in the order the processor sent it.

If a decision is made not to queue the data then the read data is accelerated through the return crossbar by bypassing the queue. The bypass delay line (shown as the heavy outline register at time 5 in Figure 3-2) is enabled and the processor ID is set. A fixed number of clocks later (two as currently programmed) RTN_SEL for the processor that sent the request will be forced to value 0xf which will direct the return crossbar to send the NCUs data the the correct processor. In addition, a RD_RDY will be sent to the processor one clock after RTN_SEL is sent.

If a decision is made to queue the data then the processor ID is pushed onto the pid queue, a bit is set in a clearable delay line for the request type (read or TAM) and a bit is set in the push delay

line. Both of these delay lines are programmable and allow for different return times for reads or TAMs. The push delay line asserts CR_PUSH at the same time the data returns from the NCU and the next cycle the return crossbar will push the data onto a queue.

The request at the head of the pid queue is examined to determine when the return data can be popped from the queue in the return crossbar and sent to the processor. This is done by looking at the first entry in the clearable delay line (cr_rd[ ] or cr_tam[ ]) vs the processor's delay line (p*p*m[] and p*p*id[ ] where *p* is the processor) as they are equivalent in time. A pop signal will be sent when the data is returning or has returned from the NCU and any NMB requests ahead of that from the same processor are returning or have returned. The next cycle CR_POP will be asserted, the pid queue will be popped, and the first entry in the delay line cleared. RTN_SEL will not have a valid memory ID and will default to 0x8 which directs the return crossbar to select the NCU queue to the appropriate processor.

Note that CR_POP will always be asserted after CR_PUSH. The time spreads for queued data returning are also shown in Figure 3-2. Pardon the mess. The return control also detects the following errors and signals an RCTL hard error if they occur: illegal memory error (arbitration sending a request to an illegal memory board), bubble error (arbitration sent a request to the NCU immediately following a TAM request to the NCU), and accel error (return control attempted to push NCU request on queue and bypass queue; return control is very confused).

3.6 Return Crossbar

The return crossbar subsystem consists of the actual crossbar made of muxes and registers and a queue for NCU data. The crossbar section is a processor interface. Each processor interface receives RTN_SEL*p* set to *m* by the return control. The return data from memory board *m* is selected and sent to a staging register (data_p[*p*] and par_p[*p*]) destined for processor *p*. Note that the return data signals from the memory boards are not register to register; they go through a mux first. This is the one exception in the system. If processor *p* detects a parity error in the return data then it will send a signal to the error capture logic which will hold all registers in the return crossbar and a copy of the data sent to the processor will be in registers r3data_p[*p*] and r3data_p[*p*].

Returns from the NCU are handled slightly differently. A RTN_SEL code of 0xf selects data directly from the NCU and a code of 8 selects data from the NCU queue. The queue is implemented in hardware as a circular queue with head and tail pointers. The CR_PUSH signal from the return control pushes data onto the queue (next cycle the tail pointer increments) and the CR_POP signal pops data off the queue (next cycle the head pointer increments). Return data is loaded into the location pointed to by the tail pointer every clock. The output of the queue is the data at the location pointed to by the head pointer.

If error checking is enabled and the queue overflows (CR_PUSH asserted last cycle, CR_POP not asserted last cycle, and tail equals head this cycle) then all registers will be held on the return crossbar and a queue error will be signaled. The same applies for a queue underflow (CR_POP asserted last cycle and head equaled tail last cycle). Note that head equal tail means the queue is empty.

3.7 Error Capture

This subsystem provides various utilites for the operation and diagnosis of the crossbar system. Various hardware errors are detected or captured by the crossbar using the error capture logic.

The scan interface allows diagnosis and configuration of the boards.

3.7.1 Send Parity Error

A send parity error is detected by an NMB or NCU which informs the crossbar error capture (in XS0 and XS1). Parts of the send crossbar are halted and the next cycle the return control and return crosssbar are halted. Memory board *m* has detected an error if send_par_err bit *m* is set and errors are enabled with m_err_en bit *m* set. A copy of the data or control that was sent is in the send crossbar in lsd2[*m*] and the processor's number that sent the request is in the return control in r5sel*m* (or r5_cr_sel if *m*=8).

3.7.2 Return Parity Error

A return parity error is detected by an NSP or NIA which informs the crossbar error capture on the XRT. The return control and return crossbar are halted. Processor *p* has detected an error if rtn_par_err bit *p* is set and p_err_en is enabled, a copy of the data is in the return crossbar in r3data_p[*p*], and it was sent by memory board r3sel_p[*p*].

3.7.3 PCM Error

A physical configuration map (PCM) error is detected by the arbitration which halts and informs the error capture logic which stops part of the send crossbar (XS0 only) the next cycle. XARB 0 detected an invalid reference to NMB 0 through 3 or NCU if r_xarb_a_pcm_err is set, XARB 1 if r_xarb_b_pcm_err is set. The XARB should have pcm_err set and pcm_chk_en_n zero (enabled). The invalid request was to memory interface *mx* if vmb[*mx*] is zero (not valid) and m_rdy0[*mx*] is set. Processor send_sel[*mx*] sent the request which is in the send crossbar: addr_os[*m*] and cycle_os[*m*].

3.7.4 Return Control Errors

If the return control detects an error then it halts and informs the error capture logic which halts the return crossbar the next cycle. The field r_rctl_hard_error on the XRT will be set. A RD_RDY error has occured for NMB *m* when it is enabled (disable_m[*m*] is zero) and r_mx_rd_rdy[*m*] (what RD_RDY was last cycle) differs from r_check_m[*m*] (what the return control expected it to be). A RD_RDY error has occured for NCU when disable_cr is zero and r_cr_rd_rdy (what RD_RDY was last cycle) differs from the OR of r_cr_push and r2_force_accel (NCU data was pushed last cycle or queue was bypassed last cycle).

An illegal memory error has occured in the return control for NMB *m* when r_check_m[*m*] and disable_m[*m*] are both set. If the NCU is ever disabled, the error occurs when disable_cr is set and either r_cr_push or r2_force_accel is set. A bubble error occurs when rcr_rdy is set and rcr_cycle bit 0 is set (NCU received a request this cycle) and r_cr_tam is set (NCU received a TAM request last cycle). The return control is double checking the arbitration for both of these errors. The return control checks itself with the cr_accel error which occurs if both r_cr_push and r2_force_accel are set (return control is pushing and bypassing queue at the same time).

3.7.5 Return Crossbar Errors

Queue overflow and underflow are detected by the return crossbar which is double checking the return control. Both of these errors should leave r_rxbr*x*_hard_err set for all RXBRs (*x* is the RXBR number). An overflow has occured if queue_overflow is set in the RXBRs, an underflow has occured when queue_underflow is set. Truly amazing. Interesting state saved in the RXBRs

includes r_cr_push and r_cr_pop (what push and pop were last cycle which caused the error), and head and tail (state of the pointers after the error).

Note that all these error checks can be disabled. The error capture logic which halts other subsystems is disabled when halt_disable is set on that board. Return control errors are separately enabled by rd_rdy_err_en, illegal_mem_err_en, bubble_err_en and cr_accel_err_en. Return crossbar errors are enabled by setting queue_err_en.

3.8 Scan Modes

All the crossbar boards can be in one of four modes. Most of this document describes the normal mode of operation. The boards can also be in board shift left mode where all registers on the board are made into a shift register and the scan engine in the NCU can read or set the state of the boards. Processors can be taken offline and online while the system is running by putting the boards in dynamic configuration load mode. This mode is identical to normal mode except the processor configuration fields are read or set by the NCU while the system is running. The fields will be set when the board returns to normal mode; the fields are held via latches or register copies durring dynamic configuration load mode. The last mode is CAST load mode which is similar to normal mode except all clock gating is disabled, i.e. all registers will clock. This mode is for diagnostic purposes.

3.9 Clock and Misc Logic

One other system of note is the clock distribution system. The crossbar boards receive a 1X clock with a 16 ns cycle or 62.5 MHz freq. This clock is buffered once and copies sent to the gate arrays, and one copy is buffered again and copies sent to the ECLiPS registers. Note that there is no phase generator and only one or two levels of buffering.

3.10 Crossbar Partitioning

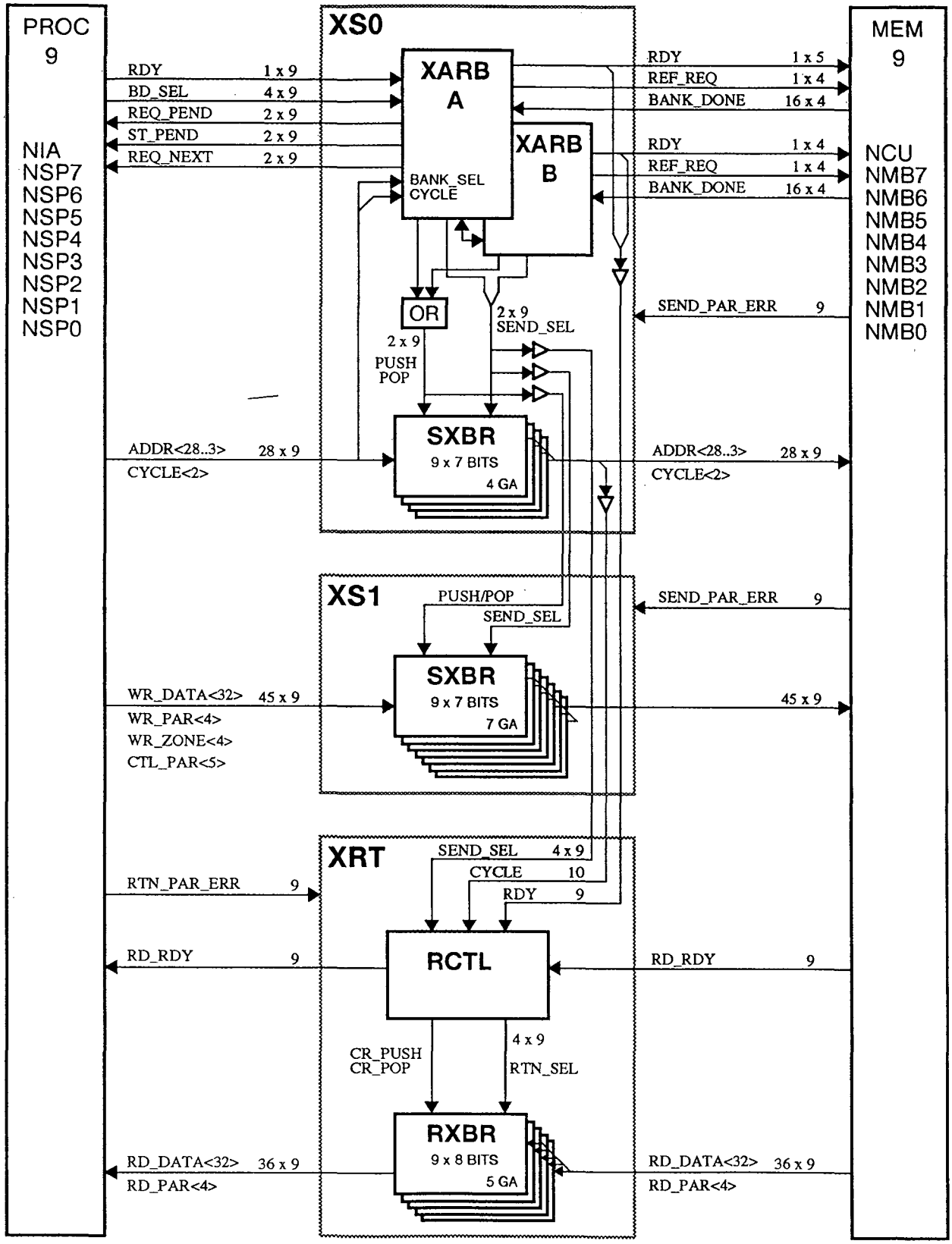
The crossbar has to send and receive a large number of signals (over 2000 per side) in a fast clock cycle (16 ns). For this reason the crossbar is partitioned into three boards per side which are centrally located. Each board has a large number of connector signals going to the horizontally mounted crossbar backplane but is fairly light on gate count.

The partitioning of the even crossbar into boards and gate arrays is shown in Figure 3-3 . The odd side of the crossbar is identical. Table 3-1 below lists the partitioning of the crossbar, showing which gate arrays and subsystems are on which boards.

Table 3-1 Crossbar Partitioning

Board	Gate Arrays		Subsystems
XS0	XARB (2)	SXBR (4)	Arbitration, Send Crossbar, Error Capture
XS1		SXBR (7)	Send Crossbar, Error Capture
XRT	RCTL (1)	RXBR (5)	Return Control, Return Crossbar, Error Capture

Figure 3-3 Even Crossbar Block Diagram



3.10.1 Arbitration Partitioning

The arbitration function is performed by two XARB gate arrays and eighteen ECLiPS OR gates on the XS0 board. Each XARB handles the arbitration for four memory boards and one handles the NCU. XARB 0 (A) handles memory boards 0-3 and the NCU, and XARB 1 (B) handles memory boards 4-7. Each XARB arbitrates between all nine processors. The PUSH and POP signals from both XARBs are ORed together on the XS0 board so that a processor's queue in the SXBR will push or pop if either XARB requests it.

The two XARB gate arrays coordinate their activities by looking at each other's POP and REQ_NEXT signals. Each XARB contains a queue for each processor of all valid requests being arbitrated so that all requests will be processed in order. A POP signal from the other XARB will clear requests stored in the current XARB's queue but being handled by the other XARB.

If an XARB has a blocked request but can queue another then it will deassert REQ_NEXT for two cycles and assert it for one until it is full of requests (maximum is three). Since a request can be sent only when REQ_NEXTs from both XARBs are asserted, the XARBs look at each other's REQ_NEXT signals to synchronize asserting them at the same time.

3.10.2 Send Crossbar Partitioning

The send crossbar function is perfomed by eleven SXBR gate arrays, four on the XS0 and seven on the XS1. Each SXBR is a seven bit slice of the total 73 signals that each processor sends to each memory. Parts of the ADDR and CYCLE fields are also used by the arbitration, so these fields were put on the XS0, and the XS1 handles all the rest.

3.10.3 Return Partitioning

The return control function is performed by the RCTL gate array which resides on the XRT. The return crossbar function is performed by five RXBR gate arrays which live on the XRT also. Each RXBR is an eight bit slice of the total 36 signals that each memory board returns to the procesors.

3.10.4 Error Capture Partitioning

The error capture function is performed by ECLiPS logic on each of the three boards. Return control, return crossbar, and return parity errors are handled by logic on the XRT. PCM errors are handled by logic on the XS0. Send parity errors are handled by logic on XS0, XS1 and XRT. Note that there is a signal (not shown in Figure 3-3, see section 2.2.2 on page 2-10) send by the XS0 which informs the XRT error capture logic to halt any time the XS0 halts.

3.11 XS0 Board

3.11.1 XS0 Timing

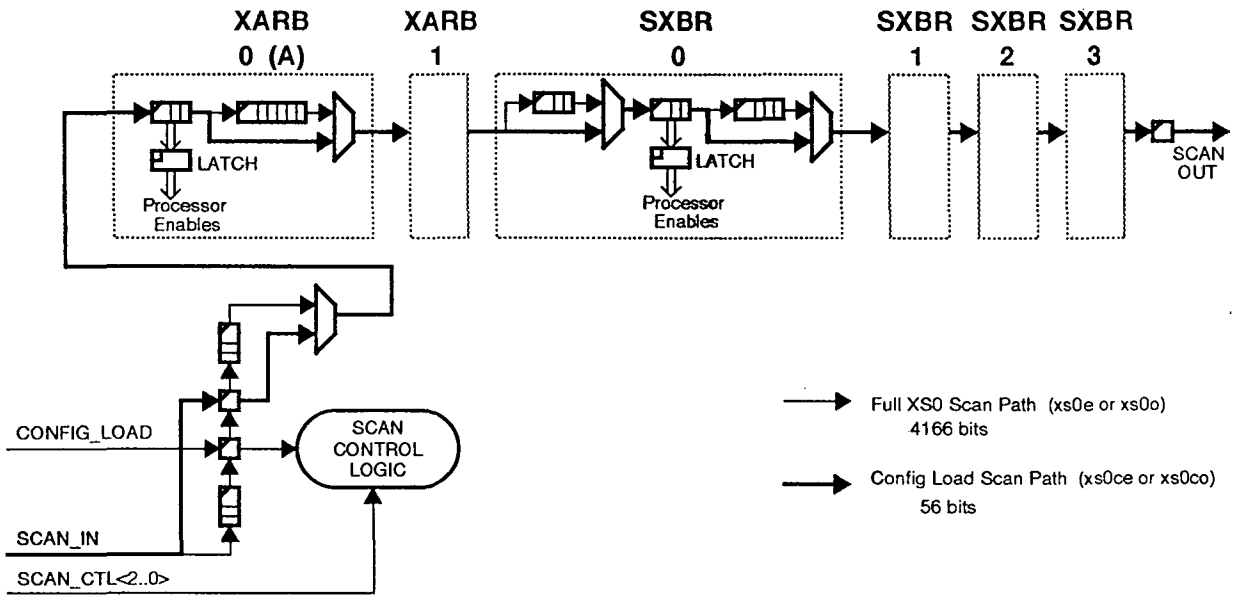
The XS0 times internally to 14.98 ns, the critical path being in the XARB. The XS0 board critical path is B_XPOP5 (13.98 ns) followed closely by A_XPOP5 (13.94 ns) and a host of other XPOP signals.

3.11.2 XS0 Scan Rings

Figure 3-4 shows the two scan rings used in the XS0. The full scan path includes every register on the board while the config load scan path bypasses most ot the registers (which are running in

normal mode) and scans the processor configuration bits (p_config[*x*]) while the system is running. The processor configuration bits have two levels; while the registers are being scanned during dynamic configuration load mode the second level bits which enable the processors are held in a latch.

Figure 3-4 XS0 Scan Rings



3.11.3 XS0 Bit Slicing

Table 3-2 shows a map of the bit slicing done on the send crossbar on XS0. The signals ADDR<6..4> are used by the XARB as bank_sel and CYCLE<1> is used by the XARB for st_pend generation. Those signals had to be on the XS0 for the XARB so to make things simple all ADDR and CYCLE signals were put on the XS0. There were no clear timing differences so all ADDR signals were bit sliced in order.

Table 3-2 XS0 Bit Slicing

SXBR	Bit	Signal Name	SXBR	Bit	Signal Name
3	6	ADDR<28>	1	6	ADDR<14>
3	5	ADDR<27>	1	5	ADDR<13>
3	4	ADDR<26>	1	4	ADDR<12>
3	3	ADDR<25>	1	3	ADDR<11>
3	2	ADDR<24>	1	2	ADDR<10>
3	1	ADDR<23>	1	1	ADDR<9>
3	0	ADDR<22>	1	0	ADDR<8>
2	6	ADDR<21>	0	6	ADDR<7>
2	5	ADDR<20>	0	5	ADDR<6>
2	4	ADDR<19>	0	4	ADDR<5>
2	3	ADDR<18>	0	3	ADDR<4>
2	2	ADDR<17>	0	2	ADDR<3>
2	1	ADDR<16>	0	1	CYCLE<1>
2	0	ADDR<15>	0	0	CYCLE<0>

3.12 XS1 Board

3.12.1 XS1 Timing

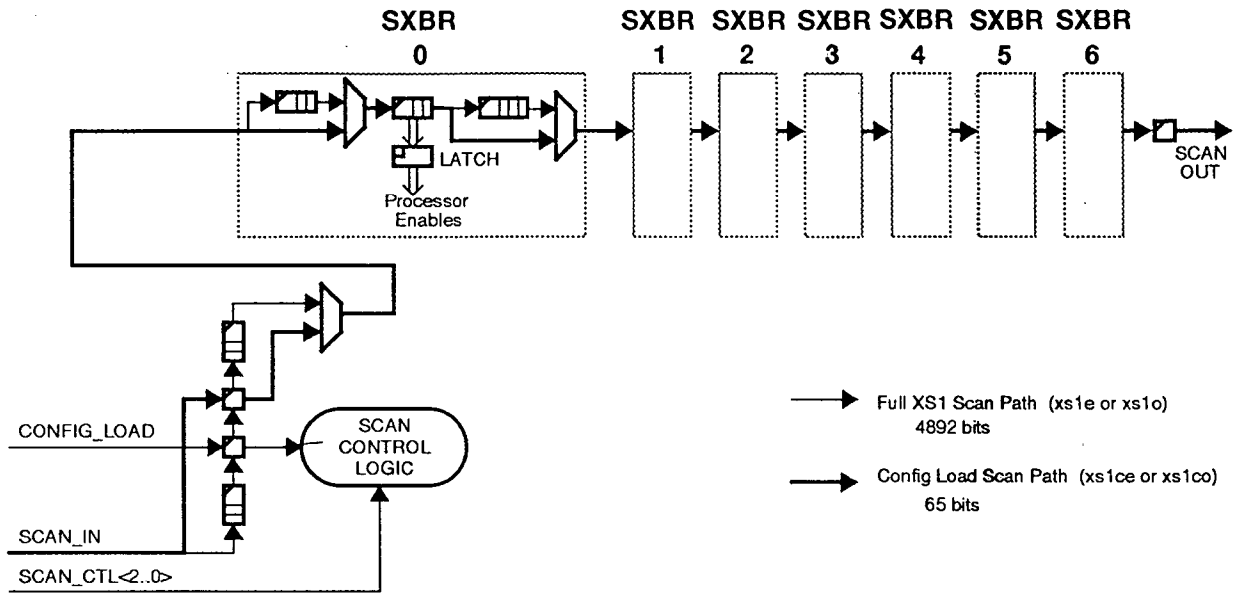
Internal timing on the XS1 works at 14 ns or faster. SXBR internal paths time to 12 ns.

The XS0 to XS1 path times to 14.22 ns with the critical path being XS0_XS1.SEND_SELx<2>. XS0_XS1.SEND_SELx<3> times to 14.12 ns and XS0_XS1.SEND_SEL<1> times to 14.06 ns.

3.12.2 XS1 Scan Rings

Figure 3-5 shows the two scan rings used in the XS1. The full scan path includes every register on the board while the config load scan path bypasses most ot the registers (which are running in normal mode) and scans the processor configuration bits (config[*x*] in SXBR *x*) while the system is running. The processor configuration bits have two levels; while the registers are being scanned during dynamic configuration load mode the second level bits which enable the processors are held in a latch.

Figure 3-5 XS1 Scan Rings



3.12.3 XS1 Bit Slicing

Table 3-3 shows a map of the bit slicing done on the send crossbar on XS1. WR_DATA, WR_PAR, WR_ZONE, and CTL_PAR signals go through the XS1's crossbar since it is larger and can handle more bits and the other signals were needed by the XS0. The NSP's bit slicing is done such that it can rotate bytes easily so, for example, one gate array will contain bits 7,6,5 of every byte, one has 4,3,2 and one has 0,1,and parity. Due to the placement of the gate arrays, some of the bits come out faster (and have better setups) than others.

The XS1 was bit sliced to optimize the timing from the NSP by putting the slowest signals in the middle of the board which is closest to the center of the crossbar. The shortest path considering every possible location of processor is the center of the crossbar and thus the slowest signals should be on the shortest paths.

Table 3-3 XS1 Bit Slicing

SXBR	Bit	Signal Name	Scalar Speed
6	6		
6	5		
6	4	WR_ZONE<3>	FAST
6	3	WR_ZONE<2>	FAST
6	2	WR_PAR<3>	FAST
6	1	WR_PAR<2>	FAST
6	0	CTL_PAR<4>	FAST
5	6	WR_DATA<25>	FAST
5	5	WR_DATA<24>	FAST
5	4	WR_DATA<17>	FAST
5	3	WR_DATA<16>	FAST
5	2	WR_DATA<28>	MED
5	1	WR_DATA<27>	MED
5	0	WR_DATA<26>	MED
4	6	WR_DATA<20>	MED
4	5	WR_DATA<19>	MED
4	4	WR_DATA<18>	MED
4	3	CTL_PAR<3>	MED
4	2	CTL_PAR<2>	MED
4	1	WR_DATA<31>	SLOW
4	0	WR_DATA<30>	SLOW
3	6	WR_DATA<29>	SLOW
3	5	WR_DATA<23>	SLOW
3	4	WR_DATA<22>	SLOW
3	3	WR_DATA<21>	SLOW
3	2	WR_DATA<15>	SLOW
3	1	WR_DATA<14>	SLOW
3	0	WR_DATA<13>	SLOW
2	6	WR_DATA<7>	SLOW
2	5	WR_DATA<6>	SLOW
2	4	WR_DATA<5>	SLOW
2	3	CTL_PAR<1>	MED
2	2	CTL_PAR<0>	MED
2	1	WR_DATA<12>	MED
2	0	WR_DATA<11>	MED
1	6	WR_DATA<10>	MED
1	5	WR_DATA<4>	MED
1	4	WR_DATA<3>	MED
1	3	WR_DATA<2>	MED
1	2	WR_DATA<9>	FAST
1	1	WR_DATA<8>	FAST
1	0	WR_DATA<1>	FAST
0	6	WR_DATA<0>	FAST
0	5	WR_PAR<1>	FAST
0	4	WR_PAR<0>	FAST
0	3	WR_ZONE<1>	FAST
0	2	WR_ZONE<0>	FAST
0	1		
0	0		

3.13 XRT Board

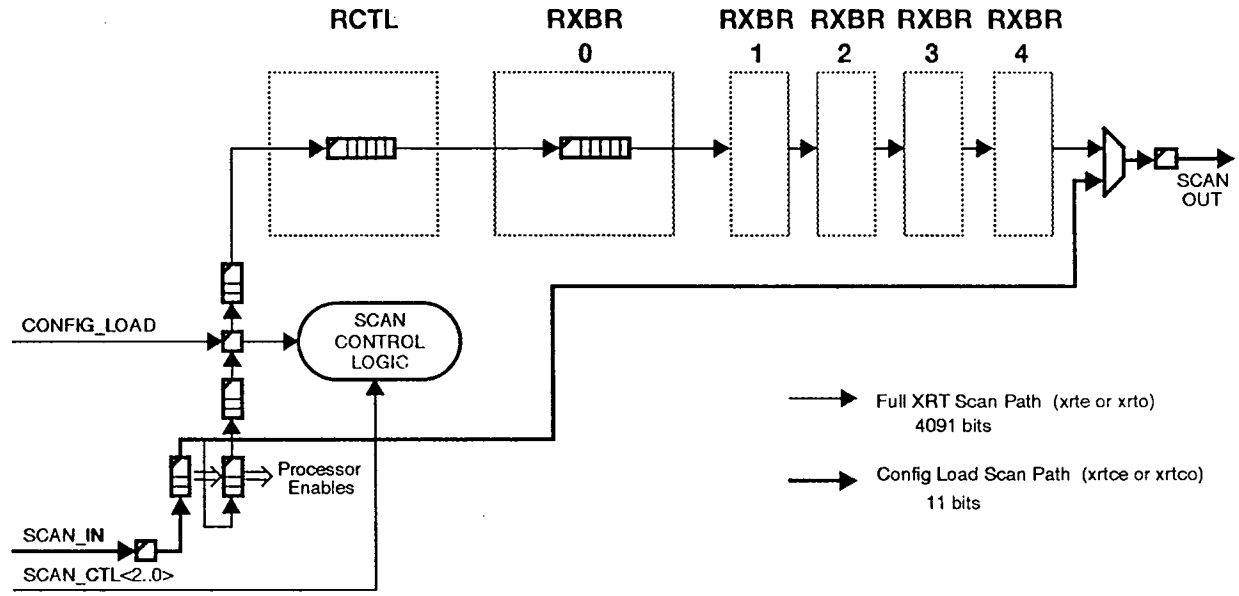
3.13.1 XRT Timing

Internally, the XRT times to 12.67 ns, the critical path being internal to the RCTL gate array. All board paths time to 12 ns with no errors.

3.13.2 XRT Scan Rings

Figure 3-6 shows the two scan rings used in the XRT. The full scan path includes every register on the board while the config load scan path bypasses most ot the registers (which are running in normal mode) and scans the processor configuration bits (p_config) while the system is running. The processor configuration bits have two levels; while the registers are being scanned during dynamic configuration load mode the second level bits which enable the processors (p_err_en) are held in another register.

Figure 3-6 XRT Scan Rings



3.13.3 XRT Bit Slicing

Table 3-4 shows a map of the bit slicing done on the return crossbar on the XRT. The NSP's bit slicing is done such that it can rotate bytes easily so, for example, one gate array will contain bits 7,6,5 of every byte, one has 4,3,2 and one has 0,1,and parity. Due to the placement of the gate arrays, some of the bits come out faster (and have better setups) than others.

The XRT was bit sliced to optimize the timing to the NSP by putting the slowest signals (the ones with the worst setup) in the middle of the board which is closest to the center of the crossbar. The

shortest path considering every possible location of processor is the center of the crossbar and thus the slowest signals should be on the shortest paths.

Table 3-4 XRT Bit Slicing

RXBR	Bit	Signal Name	Scalar Speed
4	7		
4	6		
4	5	RD_PAR<3>	FAST
4	4	RD_PAR<2>	FAST
4	3	RD_DATA<25>	FAST
4	2	RD_DATA<24>	FAST
4	1	RD_DATA<17>	FAST
4	0	RD_DATA<16>	FAST
3	7	RD_DATA<28>	MED
3	6	RD_DATA<27>	MED
3	5	RD_DATA<26>	MED
3	4	RD_DATA<20>	MED
3	3	RD_DATA<19>	MED
3	2	RD_DATA<18>	MED
3	1	RD_DATA<31>	SLOW
3	0	RD_DATA<30>	SLOW
2	7	RD_DATA<29>	SLOW
2	6	RD_DATA<23>	SLOW
2	5	RD_DATA<22>	SLOW
2	4	RD_DATA<21>	SLOW
2	3	RD_DATA<15>	SLOW
2	2	RD_DATA<14>	SLOW
2	1	RD_DATA<13>	SLOW
2	0	RD_DATA<7>	SLOW
1	7	RD_DATA<6>	SLOW
1	6	RD_DATA<5>	SLOW
1	5	RD_DATA<12>	MED
1	4	RD_DATA<11>	MED
1	3	RD_DATA<10>	MED
1	2	RD_DATA<4>	MED
1	1	RD_DATA<3>	MED
1	0	RD_DATA<2>	MED
0	7	RD_DATA<9>	FAST
0	6	RD_DATA<8>	FAST
0	5	RD_DATA<1>	FAST
0	4	RD_DATA<0>	FAST
0	3	RD_PAR<1>	FAST
0	2	RD_PAR<0>	FAST
0	1		
0	0		

Appendix A XRT Signal List

BP_XPC.SPI_IN

Serial output from EEPROM cop chip to XBAR power controller.

CK<8:0>
CK<8:0>*

First level differential clocks. These drive gate arrays clock inputs. CK<0> and CK<0>* drive the buffers for the second level clocks. CK<7,8> and CK<7,8>* are unused. All are terminated.

CK_0<8:0>
CK_0<8:0>*

Second level clocks which drive ECLiPS registers. These are single ended only so all inverted copies are unused. All are terminated.

CR_POP

Signal from RCTL to all RXBRs to pop the communications register queue.

CR_PUSH

Signal from RCTL to all RXBRs to push the NCU's read data and read parity outputs onto the communications register queue.

C_SCAN_OUT

Combinational scan output. This signal goes to a register whose output is the scan output of the XRT. C_SCAN_OUT is the output of a MUX which selects P_CONFIG<8> in dynamic configuration load mode or SCAN_DATA_RXBR4 in scan mode.

CU_XR.CLOCK_1X
CU_XR.CLOCK_1X*

Differential clock inputs into the XRT. Comes from the NCU and drives buffers which generate the first level differential clocks.

CU_XR.RD_DATA<31:0>

Read data bus from NCU's communications registers to RXBR gate arrays.

CU_XR.RD_PAR<3:0>

Read parity for the above bus. Bit 3 is the parity for data<7:0>. An odd number of bits (out of those 9) set to 1 is correct parity. Goes to RXBR gate arrays but parity checking will be done at the receiving processor.

CU_XR.RD_RDY

When this signal is asserted, the NCU is returning read data. This signal is used in error checking by the RCTL.

CU_XR.SPARE0
CU_XR.SPARE1

These wires were put in as spares in case we need extra signals between the NCU and the XRT. They connect to the via farm only.

GA_SCAN<1:0>

Gate array scan control. This signal puts all the gate array's register controls into the following modes:

00	Normal Mode
10	CAST Load Mode
11	Left Shift Mode

HALT_DISABLE

This signal is set by scan. When set to 1 it disables the XRT's error capture by preventing the XRT from halting all registers. When set it also prevents the XR_XC.HARD_ERROR signal from being set.

IA_XR.RTN_PAR_ERR

Signal from NIA 8 that it received bad parity on data from the XRT.

MB0_XR.RD_DATA<31:0>

Read data bus from NMB 0 to RXBR gate arrays.

MB0_XR.RD_PAR<3:0>

Parity check on above bus. Bit 3 is the parity for data<7:0>. An odd number of bits (out of those 9) set to 1 is correct parity. Goes to RXBR gate arrays but parity checking will be done at the receiving processor.

MB0_XR.SPARE0

MB0_XR.SPARE1

These wires were put in as spares in case we need extra signals between the NMB0 and the XRT. They connect to the via farm only.

MB1_XR.RD_DATA<31:0>

MB1_XR.RD_PAR<3:0>

MB1_XR.RD_RDY

MB1_XR.SPARE0

MB1_XR.SPARE1

MB2_XR.RD_DATA<31:0>

MB2_XR.RD_PAR<3:0>

MB2_XR.RD_RDY

MB2_XR.SPARE0

MB2_XR.SPARE1

MB3_XR.RD_DATA<31:0>

MB3_XR.RD_PAR<3:0>

MB3_XR.RD_RDY

MB3_XR.SPARE0

MB3_XR.SPARE1

MB4_XR.RD_DATA<31:0>

MB4_XR.RD_PAR<3:0>

MB4_XR.RD_RDY

MB4_XR.SPARE0

MB4_XR.SPARE1

MB5_XR.RD_DATA<31:0>

MB5_XR.RD_PAR<3:0>

MB5_XR.RD_RDY

MB5_XR.SPARE0

MB5_XR.SPARE1

MB6_XR.RD_DATA<31:0>

MB6_XR.RD_PAR<3:0>

MB6_XR.RD_RDY

MB6_XR.SPARE0

MB6_XR.SPARE1

MB7_XR.RD_DATA<31:0>
MB7_XR.RD_PAR<3:0>
MB7_XR.RD_RDY
MB7_XR.SPARE0
MB7_XR.SPARE1

The above signals come from NMBs 1 through 7 and carry the same meanings as the signals from NMB 0 which are described above.

UN10VREF-ADJ$18P$ADJUST<0>

This signal is an unnamed analog wire from the LM385BM Voltage reference. The voltage reference should be holding this line to -1.24 Volts.

P0_4_ERR

Partial result in generating XRT_HARD_ERR. Asserted if return parity errors have been enabled and detected from processors 0 through 4.

P5_8_ERR

Ditto above except processors 5 through 8.

P_CONFIG<8:0>

These signals come directly from a register which is loaded via scan or dynamic configuration load. Asserting bit *p* will enable processor *p* and the return parity error checking thereof after one clock in normal mode.

P_ERR_EN<8:0>

Registered copies of the P_CONFIG<8:0> signals. This bus is held during a dynamic configuration load but is loaded continously when back in normal mode. Bit *p* directly enables return parity error checking on processor *p*.

R_CONFIG_LOAD

Registered copy of XC_XR.CONFIG_LOAD. If asserted when XC_XR.SCAN_CTL<1,0> are zero (normal mode) then XRT is in dynamic configuration mode.

RCTL_HARD_ERR

Signal from RCTL asserting that it has taken a hardware error and is halted.

RCTL_RXBR10_ERR

Partial result in generating XRT_HARD_ERR. Asserted if RCTL, RXBR 0, or RXBR 1 has taken a hardware error (last cycle).

R_HALT_XRT

Registered XS0_XR.HALT_XRT signal. Used to generate XRT_HARD_ERR.

R_RCTL_HARD_ERR

Registered RCTL_HARD_ERR. RCTL took a hard error last cycle or earlier.

R_RXBR0_HARD_ERR
R_RXBR1_HARD_ERR
R_RXBR2_HARD_ERR
R_RXBR3_HARD_ERR
R_RXBR4_HARD_ERR

Registered RXBR*x*_HARD_ERR signals. RXBR *x* took a hard error last cycle or earlier.

R_SCAN_IN

Registered XC_XR.SCAN_IN. First bit of scan chain.

RTN_PAR_ERR<8:0>

If bit *p* is asserted then processor *p* has detected a return parity error. That bit is a

registered copy of SP*p*_XR.RTN_PAR_ERR or IA_XR.RTN_PAR_ERR if *p*=8.

RTN_SEL0<3:0>
RTN_SEL1<3:0>
RTN_SEL2<3:0>
RTN_SEL3<3:0>
RTN_SEL4<3:0>
RTN_SEL5<3:0>
RTN_SEL6<3:0>
RTN_SEL7<3:0>
RTN_SEL8<3:0>

These signals come from the RCTL and control MUXes in the RXBR gate arrays. RTN_SEL*p* controls the source of read data returned to processor *p*. Example:

RTN_SEL0	Source of read data for processor 0
0	NMB 0
1	NMB 1
2	NMB 2
3	NMB 3
4	NMB 4
5	NMB 5
6	NMB 6
7	NMB 7
8	NCU queue in RXBR
F	NCU

RXBR0_HARD_ERR

Signal from RXBR 0 asserting that it has taken a hardware error and is halted.

RXBR1_HARD_ERR

Signal from RXBR 1 asserting that it has taken a hardware error and is halted.

RXBR2_HARD_ERR

Signal from RXBR 2 asserting that it has taken a hardware error and is halted.

RXBR3_HARD_ERR

Signal from RXBR 3 asserting that it has taken a hardware error and is halted.

RXBR4_HARD_ERR

Signal from RXBR 4 asserting that it has taken a hardware error and is halted.

RXBR42_HALT_ERR

Partial result in generating XRT_HARD_ERR. Asserted if RXBR 2, 3, or 4 has taken a hardware error (last cycle) or R_HALT_XRT is asserted.

SCAN_CONFIG

Asserted when the XRT is in dynamic configuration load mode. Generated from the scan decoder PAL, it holds the P_ERR_EN bus during configuration load mode.

SCAN_CONFIG_SHIFT

Asserted when the XRT is in dynamic configuration load mode or left shift mode. Generated from the scan decoder PAL, it shifts the P_CONFIG bus when asserted.

SCAN_DATA_CLK

Full scan ring data output from clock subsystem to RCTL scan data input.

SCAN_DATA_CONFIG

One of the many scan data path signals in the clock, error capture, and scan

subsystem.

SCAN_DATA_RCTL
: Scan data out of RCTL to RXBR 0.

SCAN_DATA_RXBR0
: Scan data out of RXBR 0 to RXBR 1.

SCAN_DATA_RXBR1
: Scan data out of RXBR 1 to RXBR 2.

SCAN_DATA_RXBR2
: Scan data out of RXBR 2 to RXBR 3.

SCAN_DATA_RXBR3
: Scan data out of RXBR 3 to RXBR 4.

SCAN_DATA_RXBR4
: Scan data out of RXBR 4 to scan logic in error capture subsystem.

SCAN_EQ_0
: Used in the heartbeat logic. Asserted when all three bits of SCAN_CTL are zero.

SCAN_SHIFT
: Asserted when the XRT is in left shift mode. Generated from the scan decoder PAL, it shifts most of the ECLiPS registers when asserted.

SP0_XR.RTN_PAR_ERR
SP1_XR.RTN_PAR_ERR
SP2_XR.RTN_PAR_ERR
SP3_XR.RTN_PAR_ERR
SP4_XR.RTN_PAR_ERR
SP5_XR.RTN_PAR_ERR
SP6_XR.RTN_PAR_ERR
SP7_XR.RTN_PAR_ERR
: SP*p*_XR.RTN_PAR_ERR is asserted when processor *p* has detected a return parity error on data sent to it from the XRT.

STOP_CLOCK<3:0>
: When asserted, all registers on the XRT will hold except the register generating XR_XC.HARD_ERROR which will be set on the next clock. All of these signals will be asserted if XRT_HARD_ERR is asserted and HALT_DISABLE is not asserted. NOTE: Registers are only held when the XRT is in normal or dynamic configuration load mode. These signals are ignored otherwise.

STOP_HOLD
: Asserted when XRT is in normal or dynamic configuration load mode and STOP_CLOCK is asserted. This signal is generated from the scan decoder PAL and it holds the ECLiPS registers containing the hardware error which caused the STOP_CLOCK to be asserted.

UN11CAP$27P$PIN2<0>
: Unnamed signal between a capacitor and resistor used as a low pass filter for the temperature sensor.

VREF
: Voltage reference for the RCTL and RXBR gate arrays. Should be -1.32 Volts.

XC_XR.CONFIG_LOAD
: Signals XRT to go into dynamic configuration load mode as soon as this signal is registered. This only applies if the XRT was in normal mode.

XC_XR.SCAN_CTL<2:0>

Scan control for the XRT. This signal puts all the XRT's register controls into the following modes:

X00	Normal Mode or Dynamic Configuration Load Mode
X10	CAST Load Mode
X11	Left Shift Mode

XC_XR.SCAN_IN

Scan data input into the XRT. Comes from the scan engine and carries serial data to be stored in registers on the XRT.

XPC_BP.SPI_OUT

Serial input into the EEPROM cop chip from the XBAR power controller.

XPC_XR.BDTYPE_GND

Ground signal applied to the XRT from the XBAR power controller. This is used to ground certain board type bits on the bus XR_XPC.BD_TYPE<7:0> which forces them to zero. Bits 7,6,5,4, and 2 are forced to zero, the rest float high, so the XBAR power controller reads a code of 0x0b when an XRT is plugged in.

XPC_XR.CLKLEDCLR

Clock LED clear. Power people have not find the underscore key yet. This signal will asynchronously reset the register driving the XR_XPC.CLKLEDON signal. Brought to you by the XBAR power controller, goes to the heartbeat logic.

XPC_XR.OVERTEMP_GND

Ground signal from the XBAR power controller which supplies ground for the temperature sensor.

XPC_XR.OVERTEMP_VCC

Comming from the XBAR power controller, it supplies VCC (+5 Volts) for the temperature sensor.

XPC_XR.SPI_CLK

This signal clocks data into the serial EEPROM. From the XBAR power controller.

XPC_XR.SPI_GND

Ground for the EEPROM.

XPC_XR.SPI_SELECT

Chip select for the EEPROM.

XPC_XR.SPI_VCC

VCC (+5 Volts) for the EEPROM.

XR_IA.RD_DATA<31:0>

Read data bus from RXBR gate arrays to NIA 8.

XR_IA.RD_PAR<3:0>

Parity bits for read data bus above.

XR_IA.RD_RDY

When this signal is asserted, the XRT is returning read data to NIA 8. From the RCTL to the NIA 8.

XR_IA.SPARE0

XR_IA.SPARE1

These wires were put in as spares in case we need extra signals between the XRT and NIA 8. They connect to the via farm only.

XR_SP0.RD_DATA<31:0>

Read data bus from RXBR gate arrays to NSP 0. Note that this could be a scalar

processor or an NIA even though the name implies a scalar processor.

XR_SP0.RD_PAR<3:0>

Parity bits for read data bus above.

XR_SP0.RD_RDY

When this signal is asserted, the XRT is returning data to NSP 0. This signal comes from the RCTL.

XR_SP0.SPARE0

XR_SP0.SPARE1

These wires were put in as spares in case we need extra signals between the XRT and NSP 0. They connect to the via farm only.

XR_SP1.RD_DATA<31:0>
XR_SP1.RD_PAR<3:0>
XR_SP1.RD_RDY
XR_SP1.SPARE0
XR_SP1.SPARE1
XR_SP2.RD_DATA<31:0>
XR_SP2.RD_PAR<3:0>
XR_SP2.RD_RDY
XR_SP2.SPARE0
XR_SP2.SPARE1
XR_SP3.RD_DATA<31:0>
XR_SP3.RD_PAR<3:0>
XR_SP3.RD_RDY
XR_SP3.SPARE0
XR_SP3.SPARE1
XR_SP4.RD_DATA<31:0>
XR_SP4.RD_PAR<3:0>
XR_SP4.RD_RDY
XR_SP4.SPARE0
XR_SP4.SPARE1
XR_SP5.RD_DATA<31:0>
XR_SP5.RD_PAR<3:0>
XR_SP5.RD_RDY
XR_SP5.SPARE0
XR_SP5.SPARE1
XR_SP6.RD_DATA<31:0>
XR_SP6.RD_PAR<3:0>
XR_SP6.RD_RDY
XR_SP6.SPARE0
XR_SP6.SPARE1
XR_SP7.RD_DATA<31:0>
XR_SP7.RD_PAR<3:0>
XR_SP7.RD_RDY
XR_SP7.SPARE0
XR_SP7.SPARE1

All the above signals are identical to the XR_SP0 signals except they go to NSP 1

through 7. See XR_SP0 signals above for signal descriptions. Note again that the signal destinations can be NSPs or NIAs.

XRT_HARD_ERR
XRT_HARD_ERR*

This differential signal is asserted when the XRT has detected a hardware error. XRT clocks will be halted unless HALT_DISABLE is asserted. This is internal to the error capture logic.

XR_XC.HARD_ERROR

This signal tells the XCL that the XRT has detected a hardware error and has halted. If not masked by the diagnostics, this will eventually halt the entire Neptune system. This is a registered version of STOP_CLOCK.

XR_XC.SCAN_OUT

Scan data output from the XRT. Comes from an XRT register (in the scan logic) and carries serial data from the XRT's registers to the scan engine.

XR_XPC.BDTYPE<3,1,0>

This bus goes to the XBAR power controller from the XRT to identify board type. Bits 3,1, and 0 are not attached to XPC_XR.BDTYPE_GND so they will float high; the others will be low. This sends a board type code of 1011 (binary) when the XRT is plugged in. The XBAR power controller will complain if the wrong board is in an XRT slot.

XR_XPC.CLKLEDON

Clock LED on. As the name implies, when this signal is asserted the LED associated with the XRT is on. This signal is asserted when the XRT is in normal mode, clocks are running, and CLKLEDCLR is not asserted. Drives to the XBAR power controller from the heartbeat logic on the XRT. When Neptune was still young, Clkledons roamed the simulations wreaking terror in designers hearts.

XR_XPC.OVERTEMP

This is the output of the XRT's temperature sensor which drives out an analog voltage proportional to the temperature of the board. This signal is presumably read by an A/D converter in the XBAR power controller.

XS0_XR.CR_CYCLE<1:0>

Copy of the cycle bits sent to the NCU from the XS0. This bus is received by the RCTL gate array which will direct returning read data, if any, to the appropriate processor. Upon receiving a valid request, the cycle bits direct the NCU to perform the following memory cycles:

CYCLE<1:0>	Memory Operation	
00	NOP	No operation, ignored by RCTL
01	Read	Returns read data from NCU
10	Write	Ignored by RCTL
11	Test and Modify	Generally an increment instruction, returns read data from NCU, can't be immediately followed by a valid request

XS0_XR.CR_RDY

Copy of the RDY signal sent to the NCU from the XS0. This is received by the RCTL gate array. When asserted, a valid request is being made to the NCU and the RCTL will direct returning read data, if any, to the appropriate processor.

XS0_XR.CR_SEND_SEL<3:0>

This bus contains the number of the processor whose request will be sent to the NCU next cycle. The RCTL receives this signal, and on the next clock receives RDY and CYCLE for the NCU. If RDY is asserted and CYCLE<0> is one, then a request for read data is being made to the NCU. The value of SEND_SEL on the previous clock was the processor that is sendng the request. See the XS0 to XRT Interface for more details. This signal is a copy of SEND_SEL8 sent by XARB 0 in the XS0 to the SXBR gate arrays.

XS0_XR.HALT_XRT

Signal from the XS0 to the XRT that the XS0 had taken a hardware error and has halted. Once registered, this signal halts the XRT clocks if HALT_DISABLE is not asserted. The purpose of this signal is to hold registers inside the RCTL which contain the processor's number that sent data that might have caused the send parity error which halted the XS0.

XS0_XR.M0_CYCLE

Copy of CYCLE<0> sent to NMB 0 from the XS0. Note that if this bit is one then a read cycle is specified. This bus is received by the RCTL gate array. If this bit is one and XS0_XR.M0_RDY is asserted then a read request is being sent to NMB 0.

XS0_XR.M0_RDY

Copy of the RDY signal sent to NMB 0 from the XS0. This is received by the RCTL gate array. When asserted, a valid request is being made to NMB 0 and if XS0_XR.M0_CYCLE is one then that request will return read data. XS0_XR.SEND_SEL0<3:0> from the previous clock contains the number of the processor that is making the request.

XS0_XR.M1_CYCLE
XS0_XR.M1_RDY
XS0_XR.M2_CYCLE
XS0_XR.M2_RDY
XS0_XR.M3_CYCLE
XS0_XR.M3_RDY
XS0_XR.M4_CYCLE
XS0_XR.M4_RDY
XS0_XR.M5_CYCLE
XS0_XR.M5_RDY
XS0_XR.M6_CYCLE
XS0_XR.M6_RDY
XS0_XR.M7_CYCLE
XS0_XR.M7_RDY

The above signals are copies of signal sent to NMBs 1 through 7. Other than that they are identical to the NMB 0 signals which are described above.

XS0_XR.SEND_SEL0<3:0>
XS0_XR.SEND_SEL1<3:0>
XS0_XR.SEND_SEL2<3:0>
XS0_XR.SEND_SEL3<3:0>
XS0_XR.SEND_SEL4<3:0>
XS0_XR.SEND_SEL5<3:0>
XS0_XR.SEND_SEL6<3:0>
XS0_XR.SEND_SEL7<3:0>

These busses are copies of SEND_SEL busses sent by the XARBs in the XS0 to the SXBR gate arrays. XS0_XR.SEND_SEL*m* contains the number of the processor whose request will be sent to NMB *m* next cycle. The RCTL receives this signal, and on the next clock receives RDY and CYCLE for NMB *m*. If RDY is asserted and CYCLE<0> is one, then a request for read data is being made to NMB *m*. The value of XS0_XR.SEND_SEL*m* on the previous clock is the processor that is sending the request. See the XS0 to XRT Interface for more details.

XS0_XR.SPARE0

XS0_XR.SPARE1

These wires were put in as spares in case we need extra signals between the XS0 and the XRT. They connect to the via farm only.

ZERO<19:0>

As the name implies, these signals should always be logic zero. They are attached to terminators and pull several inputs to zero. Some of these pull gate array inputs (which should not float) down to zero. Others are attached to ECLiPS inputs (which can float due to internal pull downs) and are there for GENRAD testing so the terminator via can be probed by GENRAD and the net forced to a one for testing.

Appendix B XS0 Signal List

NOTE: Signals prefaced with "A_" come from XARB A, signals prefaced with "B_" come from XARB B. XARB A (0) arbitrates all processors for NMB 0 through 3 and the NCU. XARB B (1) arbitrates all processors for access to NMB 4 through 7.

A_POP0
A_POP1
A_POP2
A_POP3
A_POP4
A_POP5
A_POP6
A_POP7
A_POP8

Signal A_POP*p* pops processor *p*'s request off the processor interface queue in the SXBRs. This comes from XARB A and signifies that the request at the head of processor *p*'s queue won arbitration last cycle for memory boards 0,1,2,3, or NCU. These signals go to OR gates which combine with XARB B's B_POP*p* signals into a POP*p* signal which goes to the SXBRs with information on all memory boards. These signals also go to buffers and then, as A_XPOP*p*, to XARB B so it can pop the request at the head of its processor interface *p* queue.

A_PUSH0
A_PUSH1
A_PUSH2
A_PUSH3
A_PUSH4
A_PUSH5
A_PUSH6
A_PUSH7
A_PUSH8

Signal A_PUSH*p* pushes processor *p*'s last request onto the processor interface queue in the SXBRs. This comes from XARB A and signifies that processor *p*'s request for memory boards 0,1,2,3, or NCU cannot immediately proceed and needs to be queued. These signals go to OR gates which combine XARB B's signals into a PUSH*p* signal which goes to the SXBRs with information on all memory boards.

A_REQ_NEXT0
A_REQ_NEXT1
A_REQ_NEXT2
A_REQ_NEXT3
A_REQ_NEXT4
A_REQ_NEXT5
A_REQ_NEXT6

A_REQ_NEXT7
A_REQ_NEXT8

Signal A_REQ_NEXT*p* is a buffered copy of the XS0_SP*p*.A_REQ_NEXT (or IA if *p*=8) signal which goes to processor *p* as a handshake. When set, these signals mean that XARB A is ready to accept a request (RDY set) from processor *p* next cycle. A_REQ_NEXT*p* goes to test points on the XS0 for easy logic analyzer access, and also to XARB B so it can synchronize asserting its REQ_NEXT signals since the handshake is only valid if both XARB's REQ_NEXT signals are asserted.

A_XPOP0
A_XPOP1
A_XPOP2
A_XPOP3
A_XPOP4
A_XPOP5
A_XPOP6
A_XPOP7
A_XPOP8

Signal A_XPOP*p* is a buffered copy (for timing purposes) of A_POP*p*. These go to XARB B to tell it to pop the request at the head of its processor interface *p* queue.

B_POP0
B_POP1
B_POP2
B_POP3
B_POP4
B_POP5
B_POP6
B_POP7
B_POP8

Signal B_POP*p* pops processor *p*'s request off the processor interface queue in the SXBRs. This comes from XARB B and signifies that the request at the head of processor *p*'s queue won arbitration last cycle for memory boards 4,5,6, or 7. These signals go to OR gates which combine with XARB A's A_POP*p* signals into a POP*p* signal which goes to the SXBRs with information on all memory boards. These signals also go to buffers (B_POP2 goes direct) and then, as B_XPOP*p*, to XARB A so it can pop the request at the head of its processor interface *p* queue.

B_PUSH0
B_PUSH1
B_PUSH2
B_PUSH3
B_PUSH4
B_PUSH5
B_PUSH6
B_PUSH7
B_PUSH8

Signal B_PUSH*p* pushes processor *p*'s last request onto the processor interface queue in the SXBRs. This comes from XARB B and signifies that processor *p*'s request for memory boards 4,5,6, or 7 cannot immediately proceed and needs to

be queued. These signals go to OR gates which combine with XARB A's signals into a PUSH*p* signal which goes to the SXBRs with information on all memory boards.

BP_XPC.SPI_IN

Serial output from EEPROM cop chip to XBAR power controller.

BP_XS0.SPARE0
BP_XS0.SPARE1

These wires were put in as spares in case we need extra signals between the XS0E and the XS0O. They connect to the via farm only.

B_REQ_NEXT0
B_REQ_NEXT1
B_REQ_NEXT2
B_REQ_NEXT3
B_REQ_NEXT4
B_REQ_NEXT5
B_REQ_NEXT6
B_REQ_NEXT7
B_REQ_NEXT8

Signal B_REQ_NEXT*p* is a buffered copy of the XS0_SP*p*.B_REQ_NEXT (or IA if *p*=8) signal which goes to processor *p* as a handshake. When set, these signals mean that XARB B is ready to accept a request (RDY set) from processor *p* next cycle. B_REQ_NEXT*p* goes to test points on the XS0 for easy logic analyzer access, and also to XARB A so it can synchronize asserting its REQ_NEXT signals since the handshake is only valid if both XARB's REQ_NEXT signals are asserted.

B_XPOP0
B_XPOP1
B_XPOP3
B_XPOP4
B_XPOP5
B_XPOP6
B_XPOP7
B_XPOP8

Signal B_XPOP*p* is a buffered copy (for timing purposes) of B_POP*p*. These go to XARB A to tell it to pop the request at the head of its processor interface *p* queue. Note that B_XPOP2 is "missing"; B_POP2 goes directly to XARB A without buffering (also for timing reasons).

CK<8:0>
CK<8:0>*

First level differential clocks. These drive gate arrays clock inputs. CK<0> and CK<0>* drive the buffers for the second level clocks. CK<7,8> and CK<7,8>* are unused. All are terminated.

CK_0<8:0>
CK_0<8:0>*

Second level clocks which drive ECLiPS registers. These are single ended only so all inverted copies are unused. All are terminated.

CU_XS0.CLOCK_1X
CU_XS0.CLOCK_1X*

Differential clock inputs into the XS0. Comes from the NCU and drives buffers which generate the first level differential clocks.

CU_XS0.SEND_PAR_ERR

Signal from NCU that it received bad parity on data from XS0 or XS1. This signal is registered in the error capture logic. If set, then once it is registered, if M_ERR_EN<8> is set and HALT_DISABLE is not set, then registers on the XS0 will be halted.

GA_SCAN_A<1:0>

GA_SCAN_B<1:0>

Gate array scan control. A and B should be identical buffered copies. The A copy goes to SXBR 0, the B copy goes to SXBR 1,2, and 3. These signals do not go to the XARBs; they use the scan control signals that come on board directly. These signals put all the SXBR gate array's register controls into the following modes:

00	Normal Mode
01	Dynamic Configuration Load Mode
10	CAST Load Mode
11	Left Shift Mode

$\overline{\text{HALT}}$_DISABLE

This signal is set by scan. When set to 1 it disables the XS0's error capture by preventing the XS0 from halting registers. When set it also prevents the XS0_XC.HARD_ERROR and XS0_XR.HALT_XRT signals from being set.

IA_XS0.ADDR<28:3>

Address bus from NIA 8 to the SXBRs on XS0. Bits 3 through 6 are the bank select portion of the address and these go to both XARBs as well as the SXBRs. Note that this is a word address. Bits 1 through 0 would be byte selects and these are encoded in the WR_ZONE bus to the XS1. Bit 2 would be even or odd word select and this is done by the processor's sending the request to the even or odd crossbar.

IA_XS0.BD_SEL<3:0>

Board select bus from NIA 8 to both XARBs. This bus determines to which memory board (or NCU) the NIA's request will be sent:

BD_SEL	Destination board
0-7	NMB 0-7
8	NCU
9	NCU which will be busied for 2 cycles (TAM).

IA_XS0.CYCLE<1:0>

Memory cycle to be performed by NIA 8's memory request. These signals go to SXBR 0. Bit also 1 goes to both XARBs so the XARBs will know if the request is a write and therefore to set ST_PEND if the request is pending. When sent to the memory board or the NCU, the following operations are performed:

CYCLE	Memory Operation
00	NOP
01	Read
10	Write
11	Test and Modify (TAM).

IA_XS0.RDY

This signal comes from the NIA 8 to both XARBs. When it is asserted and

XS0_IA.A_REQ_NEXT and XS0_IA.B_REQ_NEXT were set last cycle then a valid request is being sent from NIA 8 to the crossbar.

IA_XS0.SPARE0
IA_XS0.SPARE1

These wires were put in in case we need extra signals between the XS0 and NIA 8. They connect to the via farm only.

M0_4_ERR

This is a partial result in generating XS0_HARD_ERR. Asserted if send parity errors have been enabled and detected by any processor 0 through 4.

M5_8_ERR

Ditto above except processors 5 through 7 and NIA 8.

MB0_XS0.BANK_DONE<15:0>
MB1_XS0.BANK_DONE<15:0>
MB2_XS0.BANK_DONE<15:0>
MB3_XS0.BANK_DONE<15:0>
MB4_XS0.BANK_DONE<15:0>
MB5_XS0.BANK_DONE<15:0>
MB6_XS0.BANK_DONE<15:0>
MB7_XS0.BANK_DONE<15:0>

MB*m*_XS0.BANK_DONE bit *b* is set by NMB *m* when its bank *b* is done with its memory operation. These signals go to the appropriate XARB (A for *m*=0-3, B for *m*=4-7). The XARB sets a bank busy when it sends the NMB a memory request, and these signals tell the XARB that the bank will be free and it can send another request to that bank.

MB0_XS0.SEND_PAR_ERR
MB1_XS0.SEND_PAR_ERR
MB2_XS0.SEND_PAR_ERR
MB3_XS0.SEND_PAR_ERR
MB4_XS0.SEND_PAR_ERR
MB5_XS0.SEND_PAR_ERR
MB6_XS0.SEND_PAR_ERR
MB7_XS0.SEND_PAR_ERR

MB*m*_XS0.SEND_PAR_ERR is asserted when NMB *m* has detected a send parity error on data or control sent to it from the XS0 or XS1. These signals are registered in the error capture logic. Once registered, registers on the XS0 will halt and save a copy of the request sent to NMB *m* if MB*m*_XS0.SEND_PAR_ERR was set, M_ERR_EN<*m*> is set and HALT_DISABLE is not set.

M_ERR_EN<8:0>

Bit *m* of this bus, when set, enables send parity error checking on memory board *m*. These signals are set by scan.

UN12VREF-ADJ$14P$ADJUST<0>

This signal is an unnamed analog wire from the LM385BM voltage reference. The voltage reference should be holding this line to -1.24 Volts.

POP0
POP0*
POP1
POP1*

POP2
POP2*
POP3
POP3*
POP4
POP4*
POP5
POP5*
POP6
POP6*
POP7
POP7*
POP8
POP8*

The signal POP*p* goes to the SXBRs and pops processor *p*'s request off the processor interface queue. The signal is an OR of A_POP*p* and B_POP*p* from XARB A and XARB B. POP*p** comes from the inverted output of the OR gate, goes to an inverter whose output is POP*p*. The positive output of the OR gate is XS0_XS1.POP*p* which goes to the SXBRs on the XS1.

PUSH0
PUSH0*
PUSH1
PUSH1*
PUSH2
PUSH2*
PUSH3
PUSH3*
PUSH4
PUSH4*
PUSH5
PUSH5*
PUSH6
PUSH6*
PUSH7
PUSH7*
PUSH8
PUSH8*

The signal PUSH*p* goes to the SXBRs and pushes processor *p*'s last request onto the processor interface queue in the SXBRs. The signal is an OR of A_PUSH*p* and B_PUSH*p* from XARB A and XARB B. PUSH*p** comes from the inverted output of the OR gate, goes to an inverter whose output is PUSH*p*. The positive output of the OR gate is XS0_XS1.POP*p* which goes to the SXBRs on the XS1.

R_CONFIG_LOAD

Registered copy of XC_XS0.CONFIG_LOAD. If asserted when XC_XS0.SCAN_CTL<1,0> are zero (normal mode) then the XS0 will be put into dynamic configuration load mode.

R_SCAN_IN

Registered XC_XS0.SCAN_IN. In dynamic configuration load mode it is the first bit of scan chain.

R_XARB_A_PCM_ERR

Registered XARB_A_PCM_ERR signal. When asserted, XARB A took a physical configuration map error last cycle and the XS0 will halt registers if HALT_DISABLE is not set.

R_XARB_B_PCM_ERR

Registered XARB_B_PCM_ERR signal. When asserted, XARB B took a physical configuration map error last cycle and the XS0 will halt registers if HALT_DISABLE is not set.

SCAN_241<1:0>

These signals control the ECLiPS 241 registers containing the error state of the XS0. These drive to the SEL1 and SEL0 inputs of the 241s. For those of you without your ECLiPS manuals:

SEL1, SEL0	ECLiPS 241 function
00	Load
01	Hold
1X	Shift Left

SCAN_241 comes from the scan decoder PAL with the following function:

SCAN_CTL	STOP_CLOCK	SCAN_241	function	Board mode
00	0	00	Load	Normal mode
00	1	10	Hold	Hardware error
10	X	00	Load	CAST load
11	X	01	Left Shift	Scan Left

SCAN_DATA_CLK

This is the scan data path out of the error capture and clock logic to the scan input of XARB A. It comes from a MUX on the scan decoder PAL. In scan left mode this comes from SCAN_DATA_FREE, in dynamic configuration mode it comes from R_SCAN_IN.

SCAN_DATA_FREE

In scan left mode this is the scan data path out of the error capture and clock logic, through a MUX on the scan decoder PAL, and to the scan input of XARB A.

SCAN_DATA_SXBR0

Scan data out of SXBR 0 to SXBR 1.

SCAN_DATA_SXBR1

Scan data out of SXBR 1 to SXBR 2.

SCAN_DATA_SXBR2

Scan data out of SXBR 2 to SXBR 3. Sense a pattern here?

SCAN_DATA_SXBR3

Scan data out of SXBR 3 to a register in the error capture and clock logic.

SCAN_DATA_XARBA

Scan data out of XARB A to XARB B.

SCAN_DATA_XARBB

Scan data out of XARB B to SXBR 0.

SCAN_EQ_0

Used in the heartbeat logic to set the register that sets the LED. Asserted when all three bits of SCAN_CTL are zero (like in normal mode).

SEND_PAR_ERR<8:0>

If bit *m* is asserted then memory board *m* has detected a send parity error. If that memory board is enabled (M_ERR_EN<*m*>=1) and HALT_DISABLE is zero then the registers will be halted on the XS0. That bit is a registered copy of MB*m*_XS0.SEND_PAR_ERR or CU_XS0.SEND_PAR_ERR if *m*=8.

SEND_SEL0<3:0>
SEND_SEL1<3:0>
SEND_SEL2<3:0>
SEND_SEL3<3:0>
SEND_SEL4<3:0>
SEND_SEL5<3:0>
SEND_SEL6<3:0>
SEND_SEL7<3:0>
SEND_SEL8<3:0>

SEND_SEL*m* contains the number of the processor whose request will be sent to memory board *m* next cycle. These signals come from the XARBs and go to buffers that drive the SXBRs on the XS0 and buffers that drive the SXBRs on the XS1. These signals do not drive gate arrays due to critical timing and reflections.

SEND_SEL0_B<3:0>
SEND_SEL1_B<3:0>
SEND_SEL2_B<3:0>
SEND_SEL3_B<3:0>
SEND_SEL4_B<3:0>
SEND_SEL5_B<3:0>
SEND_SEL6_B<3:0>
SEND_SEL7_B<3:0>
SEND_SEL8_B<3:0>

These signals come from buffers and drive the SXBRs on the XS0. See above.

SP0_XS0.ADDR<28:3>
SP1_XS0.ADDR<28:3>
SP2_XS0.ADDR<28:3>
SP3_XS0.ADDR<28:3>
SP4_XS0.ADDR<28:3>
SP5_XS0.ADDR<28:3>
SP6_XS0.ADDR<28:3>
SP7_XS0.ADDR<28:3>

SP*p*_XS0.ADDR is the address of the memory request that processor *p* is making. Bits 3 through 6 are the bank select portion of the address and these go to both XARBs as well as the SXBRs. Note that this is a word address. Bits 1 through 0 would be byte selects and these are encoded in the WR_ZONE bus to the XS1. Bit 2 would be even or odd word select and this is done by the processor's sending the request to the even or odd crossbar.

SP0_XS0.BD_SEL<3:0>
SP1_XS0.BD_SEL<3:0>
SP2_XS0.BD_SEL<3:0>
SP3_XS0.BD_SEL<3:0>
SP4_XS0.BD_SEL<3:0>
SP5_XS0.BD_SEL<3:0>

SP6_XS0.BD_SEL<3:0>
SP7_XS0.BD_SEL<3:0>

SP*p*_XS0.BD_SEL is the memory board selection to which processor *p* is making the request. This bus determines to which memory board (or NCU) the processor's request will be sent:

BD_SEL	Destination board
0-7	NMB 0-7
8	NCU
9	NCU which will be busied for 2 cycles (TAM).

SP0_XS0.CYCLE<1:0>
SP1_XS0.CYCLE<1:0>
SP2_XS0.CYCLE<1:0>
SP3_XS0.CYCLE<1:0>
SP4_XS0.CYCLE<1:0>
SP5_XS0.CYCLE<1:0>
SP6_XS0.CYCLE<1:0>
SP7_XS0.CYCLE<1:0>

SP*p*_XS0.CYCLE is the memory cycle to be performed by processor *p*'s memory request. These signals go to SXBR 0. Bit also 1 goes to both XARBs so the XARBs will know if the request is a write and therefore to set ST_PEND if the request is pending. When sent to the memory board or the NCU, the following operations are performed:

CYCLE	Memory Operation
00	NOP
01	Read
10	Write
11	Test and Modify (TAM).

SP0_XS0.RDY
SP1_XS0.RDY
SP2_XS0.RDY
SP3_XS0.RDY
SP4_XS0.RDY
SP5_XS0.RDY
SP6_XS0.RDY
SP7_XS0.RDY

SP*p*_XS0.RDY comes from the processor *p* to both XARBs. When it is asserted and XS0_SP*p*.A_REQ_NEXT and XS0_SP*p*.B_REQ_NEXT were set last cycle then a valid request is being sent from processor p to the crossbar.

SP0_XS0.SPARE0
SP0_XS0.SPARE1
SP1_XS0.SPARE0
SP1_XS0.SPARE1
SP2_XS0.SPARE0
SP2_XS0.SPARE1
SP3_XS0.SPARE0
SP3_XS0.SPARE1

SP4_XS0.SPARE0
SP4_XS0.SPARE1
SP5_XS0.SPARE0
SP5_XS0.SPARE1
SP6_XS0.SPARE0
SP6_XS0.SPARE1
SP7_XS0.SPARE0
SP7_XS0.SPARE1

These wires were put in as spares in case we need extra signals between the XS0 and the processors.

STOP_CLOCK<3:0>

When asserted, registers on the XS0 will hold and save state. All these signals are asserted together and this happens when XS0_HARD_ERR is asserted and HALT_DISABLE is not set. In addition, XS0_XR.HALT_XRT will be set and XS0_XC.HARD_ERROR will be set in one clock. The registers that are held are the SXBR's output staging and last staged data register, and registers in the error capture logic that contain the error state of the XS0. NOTE: registers are only held when the XS0 is in normal mode or dynamic configuration load mode. These signals are ignored otherwise.

UN13CAP$27P$PIN2<0>

Unnamed signal between a capacitor and resistor used as a low pass filter for the temperature sensor.

VREF

Voltage reference for the XARB and SXBR gate arrays. Should be -1.32 Volts.

XARB_A_PCM_ERR

When asserted, XARB A just took a physical configuration map error and has halted relevant registers. The XS0 will halt registers if HALT_DISABLE is not set.

XARB_B_PCM_ERR

When asserted, XARB B just took a physical configuration map error and has halted relevant registers. The XS0 will halt registers if HALT_DISABLE is not set.

XC_XS0.CONFIG_LOAD

When set it signals the XS0 to go into dynamic configuration load mode as soon as this signal is registered. This only applies if the XS0 was in normal mode.

XC_XS0.REF_REQ

Refresh request staged through the XCL. This is a free running signal, high for 860 clocks and low for 100 clocks. While the signal is high the XS0 will tell NMBs to refresh if its banks are idle. When the signal goes low, the XS0 will force a refresh on any outstanding NMBs.

XC_XS0.SCAN_CTL<2:0>

Scan control for the XS0. This signal puts the XS0's register controls into the following modes:

X00	Normal Mode or Dynamic Configuration Load Mode
X10	CAST Load Mode
X11	Left Shift Mode

XC_XS0.SCAN_IN

Scan data input into the XS0. Comes from the scan engine and carries serial data to be stored in registers on the XS0.

XPC_BP.SPI_OUT

Serial input into the EEPROM cop chip from the XBAR power controller.

XPC_XS0.BDTYPE_GND

Ground signal applied to the XS0 from the XBAR power controller. This is used to ground certain board type bits on the bus XR_XPC.BD_TYPE<7:0> which forces them to zero. Bits 7,6,5,4,2, and 1 are forced to zero, the rest float high, so the XBAR power controller reads a code of 0x09 when an XS0 is plugged in.

XPC_XS0.CLKLEDCLR

Clock LED clear. This signal will asynchronously reset the register driving the XR_XPC.CLKLEDON signal. Brought to you by the XBAR power controller, goes to the heartbeat logic.

XPC_XS0.OVERTEMP_GND

Ground signal from the XBAR power controller which supplies ground for the temperature sensor.

XPC_XS0.OVERTEMP_VCC

Coming from the XBAR power controller, it supplies VCC (+5 Volts) for the temperature sensor.

XPC_XS0.SPI_CLK

This signal clocks data into the serial EEPROM. From the XBAR power controller.

XPC_XS0.SPI_GND

Ground for the EEPROM.

XPC_XS0.SPI_SELECT

Chip select for the EEPROM.

XPC_XS0.SPI_VCC

VCC (+5 Volts) for the EEPROM.

XS0_BP.SPARE0

XS0_BP.SPARE1

These wires were put in as spares in case we need extra signals between the XS0E and the XS0O. They connect to the via farm only.

XS0_HARD_ERR

XS0_HARD_ERR*

This differential signal is asserted when the XS0 has detected a hardware error. XS0 clocks will be halted in normal mode unless HALT_DISABLE is asserted. This is internal to the error capture logic.

XS0_MB0.ADDR<28:3>

XS0_MB1.ADDR<28:3>

XS0_MB2.ADDR<28:3>

XS0_MB3.ADDR<28:3>

XS0_MB4.ADDR<28:3>

XS0_MB5.ADDR<28:3>

XS0_MB6.ADDR<28:3>

XS0_MB7.ADDR<28:3>

XS0_CU.ADDR<28:3>

XS0_MB*m*.ADDR (CU for *m*=8) is the address in memory on which to perform the memory request for memory board *m*. Note that this is a word address. Bits 1 through 0 would be byte selects and these are encoded in the WR_ZONE bus to the XS1. Bit 2 would be even or odd word select and this is done by the processor's

having sent the request to the even or odd crossbar.

XS0_MB0.CYCLE<1:0>
XS0_MB1.CYCLE<1:0>
XS0_MB2.CYCLE<1:0>
XS0_MB3.CYCLE<1:0>
XS0_MB4.CYCLE<1:0>
XS0_MB5.CYCLE<1:0>
XS0_MB6.CYCLE<1:0>
XS0_MB7.CYCLE<1:0>
XS0_CU.CYCLE<1:0>

XS0_MB*m*.CYCLE (CU for *m*=8) is the memory cycle to be performed on the memory request for memory board *m*. These signals come from SXBR 0. When sent to the memory board or the NCU, the following operations are performed:

CYCLE	Memory Operation
00	NOP
01	Read
10	Write
11	Test and Modify (TAM).

XS0_MB0.RDY
XS0_MB1.RDY
XS0_MB2.RDY
XS0_MB3.RDY
XS0_MB4.RDY
XS0_MB5.RDY
XS0_MB6.RDY
XS0_MB7.RDY
XS0_CU.RDY

XS0_MB*m*.RDY (CU for *m*=8) goes to memory board *m* from the XARBs. When it is asserted then a valid request is being sent from the crossbar to memory board *m*.

XS0_MB0.REF_REQ
XS0_MB1.REF_REQ
XS0_MB2.REF_REQ
XS0_MB3.REF_REQ
XS0_MB4.REF_REQ
XS0_MB5.REF_REQ
XS0_MB6.REF_REQ
XS0_MB7.REF_REQ

XS0_MB*m*.REF_REQ is asserted for one cycle when the crossbar wants memory board *m* to perform a refresh request. Goes from XARB A to memory boards 0-3 (*m*=0,1,2,3) and from XARB B to memory boards 4-7 (*m*=4,5,6,7).

XS0_MB0.SPARE0
XS0_MB0.SPARE1
XS0_MB1.SPARE0
XS0_MB1.SPARE1
XS0_MB2.SPARE0
XS0_MB2.SPARE1
XS0_MB3.SPARE0

XS0_MB3.SPARE1
XS0_MB4.SPARE0
XS0_MB4.SPARE1
XS0_MB5.SPARE0
XS0_MB5.SPARE1
XS0_MB6.SPARE0
XS0_MB6.SPARE1
XS0_MB7.SPARE0
XS0_MB7.SPARE1
XS0_CU.SPARE0
XS0_CU.SPARE1

These wires were put in as spares in case we need extra signals between the XS0 and the memory boards. The connect to the via farms only.

XS0_SP0.A_REQ_NEXT
XS0_SP1.A_REQ_NEXT
XS0_SP2.A_REQ_NEXT
XS0_SP3.A_REQ_NEXT
XS0_SP4.A_REQ_NEXT
XS0_SP5.A_REQ_NEXT
XS0_SP6.A_REQ_NEXT
XS0_SP7.A_REQ_NEXT
XS0_IA.A_REQ_NEXT
XS0_SP0.B_REQ_NEXT
XS0_SP1.B_REQ_NEXT
XS0_SP2.B_REQ_NEXT
XS0_SP3.B_REQ_NEXT
XS0_SP4.B_REQ_NEXT
XS0_SP5.B_REQ_NEXT
XS0_SP6.B_REQ_NEXT
XS0_SP7.B_REQ_NEXT
XS0_IA.B_REQ_NEXT

XS0_SP*p.x*_REQ_NEXT goes to processor *p* (IA for p=8) as a handshake on accepting memory requests. When both A and B signals are set for processor *p* then a request will be accepted next cycle if the processor asserts SP*p*_XS0.RDY. The signals come from XARB A and XARB B. I think you can guess which ones are which.

XS0_SP0.A_REQ_PEND
XS0_SP1.A_REQ_PEND
XS0_SP2.A_REQ_PEND
XS0_SP3.A_REQ_PEND
XS0_SP4.A_REQ_PEND
XS0_SP5.A_REQ_PEND
XS0_SP6.A_REQ_PEND
XS0_SP7.A_REQ_PEND
XS0_IA.A_REQ_PEND
XS0_SP0.B_REQ_PEND

XS0_SP1.B_REQ_PEND
XS0_SP2.B_REQ_PEND
XS0_SP3.B_REQ_PEND
XS0_SP4.B_REQ_PEND
XS0_SP5.B_REQ_PEND
XS0_SP6.B_REQ_PEND
XS0_SP7.B_REQ_PEND
XS0_IA.B_REQ_PEND

XS0_SP*p.x*_REQ_PEND goes to processor *p* (IA for *p*=8) from XARB *x* to inform the processor that there is a request that is pending. That is, a request that is waiting in the arbitration. These signals will not be set if the request can proceed immediately.

XS0_SP0.A_ST_PEND
XS0_SP1.A_ST_PEND
XS0_SP2.A_ST_PEND
XS0_SP3.A_ST_PEND
XS0_SP4.A_ST_PEND
XS0_SP5.A_ST_PEND
XS0_SP6.A_ST_PEND
XS0_SP7.A_ST_PEND
XS0_IA.A_ST_PEND
XS0_SP0.B_ST_PEND
XS0_SP1.B_ST_PEND
XS0_SP2.B_ST_PEND
XS0_SP3.B_ST_PEND
XS0_SP4.B_ST_PEND
XS0_SP5.B_ST_PEND
XS0_SP6.B_ST_PEND
XS0_SP7.B_ST_PEND
XS0_IA.B_ST_PEND

XS0_SP*p.x*_ST_PEND goes to processor *p* (IA for *p*=8) from XARB *x* to inform the processor that there is a write request (store) that is pending. That is, a write cycle request that is waiting in the arbitration. These signals will not be set if the write request can proceed immediately.

XS0_XC.HARD_ERROR

This signal tells the XCL that the XS0 has detected a hardware error and has halted. If not masked by the diagnostics, this will eventually halt the entire Neptune system. This is a registered version of STOP_CLOCK.

XS0_XC.SCAN_OUT

Scan data output from the XS0. Comes from an XS0 register (in the scan logic) and carries serial data from the XS0's registers to the scan engine.

XS0_XPC.BDTYPE<3,0>

This bus goes to the XBAR power controller from the XS0 to identify board type. Bits 3 and 0 are not attached to XPC_XR.BDTYPE_GND so they will float high; the others will be low. This sends a board type code of 1001 (binary) when the XS0 is plugged in. The XBAR power controller will complain if the wrong board is in an XS0 slot.

XS0_XPC.CLKLEDON

Clock LED on. As the name implies, when this signal is asserted the LED associated with the XS0 is on. This signal is asserted when the XS0 is in normal mode, clocks are running, and CLKLEDCLR is not asserted. Drives to the XBAR power controller from the heartbeat logic on the XS0. Clockledon's are not nasty monsters.

XS0_XPC.OVERTEMP

This is the output of the XS0's temperature sensor which drives out an analog voltage proportional to the temperature of the board. This signal is presumably read by an A/D converter in the XBAR power controller.

XS0_XR.HALT_XRT

The XRT will halt the cycle after this signal is asserted. This signal is a copy of STOP_CLOCK and is asserted when the XS0 had detected a hardware error and has halted. This signal is asserted to save send select state on the XRT in case the XS0's error was a send parity error.

XS0_XR.M0_CYCLE
XS0_XR.M1_CYCLE
XS0_XR.M2_CYCLE
XS0_XR.M3_CYCLE
XS0_XR.M4_CYCLE
XS0_XR.M5_CYCLE
XS0_XR.M6_CYCLE
XS0_XR.M7_CYCLE
XS0_XR.CR_CYCLE<1:0>

These signals are buffered copies of XS0_MB*m*.CYCLE<0> and XS0_CU.CYCLE. The XRT snoops on those signals to find out if the memory request was a read.

XS0_XR.M0_RDY
XS0_XR.M1_RDY
XS0_XR.M2_RDY
XS0_XR.M3_RDY
XS0_XR.M4_RDY
XS0_XR.M5_RDY
XS0_XR.M6_RDY
XS0_XR.M7_RDY
XS0_XR.CR_RDY

These signals are buffered copies of XS0_MB*m*.RDY and XS0_CU.RDY. The XRT snoops on those signals to find out if a valid memory request was made.

XS0_XR.SEND_SEL0<3:0>
XS0_XR.SEND_SEL1<3:0>
XS0_XR.SEND_SEL2<3:0>
XS0_XR.SEND_SEL3<3:0>
XS0_XR.SEND_SEL4<3:0>
XS0_XR.SEND_SEL5<3:0>
XS0_XR.SEND_SEL6<3:0>
XS0_XR.SEND_SEL7<3:0>
XS0_XR.CR_SEND_SEL<3:0>

These signals are buffered copies of the SEND_SEL*m* signals. The XRT looks at

these signals to determine which processor requested which memory board. If XS0_XR.M*m*_RDY is set, XS0_XR.M*m*_CYCLE is set, and XS0_XR.SEND_SEL*m* = p, then processor *p* is making a request to memory board *m* that will return data. The NCU is similar: XS0_XR.CR_RDY set, XS0_XR.CR_CYCLE<0> set, and XS0_XR.CR_SEND_SEL = *p*.

XS0_XR.SPARE0
XS0_XR.SPARE1

Spare wires between the XS0 and XRT in case we need them later.

XS0_XS1.POP0
XS0_XS1.POP1
XS0_XS1.POP2
XS0_XS1.POP3
XS0_XS1.POP4
XS0_XS1.POP5
XS0_XS1.POP6
XS0_XS1.POP7
XS0_XS1.POP8

Buffered copies of POP*p*.

XS0_XS1.PUSH0
XS0_XS1.PUSH1
XS0_XS1.PUSH2
XS0_XS1.PUSH3
XS0_XS1.PUSH4
XS0_XS1.PUSH5
XS0_XS1.PUSH6
XS0_XS1.PUSH7
XS0_XS1.PUSH8

Buffered copies of PUSH*p*.

XS0_XS1.SEND_SEL0<3:0>
XS0_XS1.SEND_SEL1<3:0>
XS0_XS1.SEND_SEL2<3:0>
XS0_XS1.SEND_SEL3<3:0>
XS0_XS1.SEND_SEL4<3:0>
XS0_XS1.SEND_SEL5<3:0>
XS0_XS1.SEND_SEL6<3:0>
XS0_XS1.SEND_SEL7<3:0>
XS0_XS1.SEND_SEL8<3:0>

Buffered copies of SEND_SEL*m*. Go look there for more details.

XS0_XS1.SPARE0
XS0_XS1.SPARE1

You never know, spares may be needed between XS0 and XS1. Stranger things have happened.

ZERO<12:0>

As the name implies, these signals should always be zero. Don't set them to one or you will void the warranty unless you are a trained GENRAD professional. These signals are attached to terminators and pull inputs to zero. Some were included so nets could be forced to one for GENRAD testing.

Appendix C XS1 Signal List

BP_XPC.SPI_IN

Serial output from EEPROM cop chip to XBAR power controller.

CK<8:0>
CK<8:0>*

First level differential clocks. These drive gate arrays clock inputs. CK<0> and CK<0>* drive the buffers for the second level clocks. CK<8> and CK<8>* are unused. All are terminated.

CK_0<8:0>
CK_0<8:0>*

Second level clocks which drive ECLiPS registers. These are single ended only so all inverted copies are unused. All are terminated.

CU_XS1.CLOCK_1X
CU_XS1.CLOCK_1X*

Differential clock inputs into the XS1. Comes from the NCU and drives buffers which generate the first level differential clocks.

CU_XS1.SEND_PAR_ERR

Signal from NCU that it received bad parity on data from XS0 or XS1. This signal is registered in the error capture logic. If set, then once it is registered, if M_ERR_EN<8> is set and HALT_DISABLE is not set, then registers on the XS1 will be halted.

GA_SCAN_A<1:0>
GA_SCAN_B<1:0>

Gate array scan control. A and B should be identical buffered copies. The A copy goes to SXBR 0,1,2,and 3, the B copy goes to SXBR 4,5, and 6. These signals put all the SXBR gate array's register controls into the following modes:

00	Normal Mode
01	Dynamic Configuration Load Mode
10	CAST Load Mode
11	Left Shift Mode

HALT_DISABLE

This signal is set by scan. When set to 1 it disables the XS1's error capture by preventing the XS1 from halting registers. When set it also prevents the XS1_XC.HARD_ERROR signal from being set.

M0_4_ERR

This is a partial result in generating XS1_HARD_ERR. Asserted if send parity errors have been enabled and detected by any processor 0 through 4.

M5_8_ERR

Ditto above except processors 5 through 7 and NIA 8.

MB0_XS1.SEND_PAR_ERR
MB1_XS1.SEND_PAR_ERR
MB2_XS1.SEND_PAR_ERR
MB3_XS1.SEND_PAR_ERR
MB4_XS1.SEND_PAR_ERR
MB5_XS1.SEND_PAR_ERR

MB6_XS1.SEND_PAR_ERR
MB7_XS1.SEND_PAR_ERR

MB*m*_XS1.SEND_PAR_ERR is asserted when NMB *m* has detected a send parity error on data or control sent to it from the XS0 or XS1. These signals are registered in the error capture logic. Once registered, registers on the XS1 will halt and save a copy of the request sent to NMB *m* if MB*m*_XS1.SEND_PAR_ERR was set, M_ERR_EN<*m*> is set and HALT_DISABLE is not set. See STOP_CLOCK.

M_ERR_EN<8:0>

Bit *m* of this bus, when set, enables send parity error checking on memory board *m*. These signals are set by scan.

UN12VREF-ADJ$2P$ADJUST<0>

This signal is an unnamed analog wire from the LM385BM voltage reference. The voltage reference should be holding this line to -1.24 Volts.

R_CONFIG_LOAD

Registered copy of XC_XS1.CONFIG_LOAD. If asserted when XC_XS1.SCAN_CTL<1,0> are zero (normal mode) then the XS1 will be put into dynamic configuration load mode.

R_SCAN_IN

Registered XC_XS1.SCAN_IN. In dynamic configuration load mode it is the first bit of scan chain.

SCAN_241<1:0>

These signals control the ECLiPS 241 registers containing the error state of the XS1. These drive to the SEL1 and SEL0 inputs of the 241s. For those of you without your ECLiPS manuals:

SEL1, SEL0	ECLiPS 241 function
00	Load
01	Hold
1X	Shift Left

SCAN_241 comes from the scan decoder PAL with the following function:

SCAN_CTL	STOP_CLOCK	SCAN_241	function	Board mode
00	0	00	Load	Normal mode
00	1	10	Hold	Hardware error
10	X	00	Load	CAST load
11	X	01	Left Shift	Scan Left

SCAN_DATA_CLK

This is the scan data path out of the error capture and clock logic to the scan input of SXBR 0. It comes from a MUX on the scan decoder PAL. In scan left mode this comes from SCAN_DATA_FREE, in dynamic configuration mode it comes from R_SCAN_IN.

SCAN_DATA_FREE

In scan left mode this is the scan data path out of the error capture and clock logic, through a MUX on the scan decoder PAL, and to the scan input of SXBR 0.

SCAN_DATA_SXBR0

Scan data out of SXBR 0 to SXBR 1.

SCAN_DATA_SXBR1

Scan data out of SXBR 1 to SXBR 2.

SCAN_DATA_SXBR2

Scan data out of SXBR 2 to SXBR 3.

SCAN_DATA_SXBR3

Scan data out of SXBR 3 to SXBR 4.

SCAN_DATA_SXBR4

Scan data out of SXBR 4 to SXBR 5.

SCAN_DATA_SXBR5

Scan data out of SXBR 5 to SXBR 6.

SCAN_DATA_SXBR6

Scan data out of SXBR 6 to one register in the clock logic then off the board.

SCAN_EQ_0

Used in the heartbeat logic to set the register that sets the LED. Asserted when all three bits of SCAN_CTL are zero (like in normal mode).

SEND_PAR_ERR<8:0>

If bit *m* is asserted then memory board *m* has detected a send parity error. If that memory board is enabled (M_ERR_EN<*m*>=1) and HALT_DISABLE is zero then the registers will be halted on the XS0. That bit is a registered copy of MB*m*_XS0.SEND_PAR_ERR or CU_XS0.SEND_PAR_ERR if *m*=8.

SP0_XS1.CTL_PAR<4:0>
SP1_XS1.CTL_PAR<4:0>
SP2_XS1.CTL_PAR<4:0>
SP3_XS1.CTL_PAR<4:0>
SP4_XS1.CTL_PAR<4:0>
SP5_XS1.CTL_PAR<4:0>
SP6_XS1.CTL_PAR<4:0>
SP7_XS1.CTL_PAR<4:0>
IA_XS1.CTL_PAR<4:0>

SP*p*_XS1.CTL_PAR is the parity check for the control signals (ADDR, CYCLE, and ZONE) from processor *p* (NIA 8 is *p*=8). Good parity should be asserted on these signals at all times, however, parity is only checked by the destination memory board. See the Parity Mapping table in the Processor Send Path Interface for details on good parity.

SP0_XS1.WR_DATA<31:0>
SP1_XS1.WR_DATA<31:0>
SP2_XS1.WR_DATA<31:0>
SP3_XS1.WR_DATA<31:0>
SP4_XS1.WR_DATA<31:0>
SP5_XS1.WR_DATA<31:0>
SP6_XS1.WR_DATA<31:0>
SP7_XS1.WR_DATA<31:0>
IA_XS1.WR_DATA<31:0>

SP*p*_XS1.WR_DATA is the write data bus from processor *p* (NIA 8 is *p*=8) containing data to be written to the memory board that processor *p* is requesting.

SP0_XS1.WR_PAR<3:0>
SP1_XS1.WR_PAR<3:0>
SP2_XS1.WR_PAR<3:0>
SP3_XS1.WR_PAR<3:0>
SP4_XS1.WR_PAR<3:0>

SP5_XS1.WR_PAR<3:0>
SP6_XS1.WR_PAR<3:0>
SP7_XS1.WR_PAR<3:0>
IA_XS1.WR_PAR<3:0>

SP*p*_XS1.WR_PAR is the write parity bus from processor *p* (NIA 8 is *p*=8) containing the parity bits to check the WR_DATA bus. Good parity should be asserted on these signals at all times, however, parity is only checked by the destination memory board. See the Parity Mapping table in the Processor Send Path Interface for details on good parity.

SP0_XS1.WR_ZONE<3:0>
SP1_XS1.WR_ZONE<3:0>
SP2_XS1.WR_ZONE<3:0>
SP3_XS1.WR_ZONE<3:0>
SP4_XS1.WR_ZONE<3:0>
SP5_XS1.WR_ZONE<3:0>
SP6_XS1.WR_ZONE<3:0>
SP7_XS1.WR_ZONE<3:0>
IA_XS1.WR_ZONE<3:0>

SP*p*_XS1.WR_ZONE is the write zone bus from processor *p* (NIA 8 is *p*=8) which tells which bytes of the WR_DATA bus to write during a write or TAM memory request. See the WR_ZONE to Data Byte Mapping table in the Processor Send Path Interface for more details.

SP0_XS1.SPARE0
SP0_XS1.SPARE1
SP1_XS1.SPARE0
SP1_XS1.SPARE1
SP2_XS1.SPARE0
SP2_XS1.SPARE1
SP3_XS1.SPARE0
SP3_XS1.SPARE1
SP4_XS1.SPARE0
SP4_XS1.SPARE1
SP5_XS1.SPARE0
SP5_XS1.SPARE1
SP6_XS1.SPARE0
SP6_XS1.SPARE1
SP7_XS1.SPARE0
SP7_XS1.SPARE1
IA_XS1.SPARE0
IA_XS1.SPARE1

These wires were put in a spares in case we need extra signals between the XS1 and the processors. They connect to the via farm only.

STOP_CLOCK<2:0>

When asserted, registers on the XS1 will hold and save state. All these signals are asserted together and this happens when XS1_HARD_ERR is asserted and HALT_DISABLE is not set. XS0_XC.HARD_ERROR will be set in one clock. The registers that are held are the SXBR's output staging and last staged data register,

and registers in the error capture logic that contain the error state of the XS1. NOTE: registers are only held when the XS1 is in normal mode or dynamic configuration load mode. These signals are ignored otherwise.

UN13CAP$27P$PIN2<0>

Unnamed signal between a capacitor and resistor used as a low pass filter for the temperature sensor.

VREF

Voltage reference for the SXBR gate arrays. Should be -1.32 Volts.

XC_XS1.CONFIG_LOAD

When set it signals the XS1 to go into dynamic configuration load mode as soon as this signal is registered. This only applies if the XS1 was in normal mode.

XC_XS1.SCAN_CTL<2:0>

Scan control for the XS1. This signal puts the XS1's register controls into the following modes:

X00	Normal Mode or Dynamic Configuration Load Mode
X10	CAST Load Mode
X11	Left Shift Mode

XC_XS1.SCAN_IN

Scan data input into the XS1. Comes from the scan engine and carries serial data to be stored in registers on the XS1.

XPC_BP.SPI_OUT

Serial input into the EEPROM cop chip from the XBAR power controller.

XPC_XS1.BDTYPE_GND

Ground signal applied to the XS1 from the XBAR power controller. This is used to ground certain board type bits on the bus XR_XPC.BD_TYPE<7:0> which forces them to zero. Bits 7,6,5,4,2, and 0 are forced to zero, the rest float high, so the XBAR power controller reads a code of 0x0a when an XS1 is plugged in.

XPC_XS1.CLKLEDCLR

Clock LED clear. This signal will asynchronously reset the register driving the XR_XPC.CLKLEDON signal. Brought to you by the XBAR power controller, goes to the heartbeat logic.

XPC_XS1.OVERTEMP_GND

Ground signal from the XBAR power controller which supplies ground for the temperature sensor.

XPC_XS1.OVERTEMP_VCC

Coming from the XBAR power controller, it supplies VCC (+5 Volts) for the temperature sensor.

XPC_XS1.SPI_CLK

This signal clocks data into the serial EEPROM. From the XBAR power controller.

XPC_XS1.SPI_GND

Ground for the EEPROM.

XPC_XS1.SPI_SELECT

Chip select for the EEPROM.

XPC_XS1.SPI_VCC

VCC (+5 Volts) for the EEPROM.

XS0_XS1.POP0

XS0_XS1.POP1
XS0_XS1.POP2
XS0_XS1.POP3
XS0_XS1.POP4
XS0_XS1.POP5
XS0_XS1.POP6
XS0_XS1.POP7
XS0_XS1.POP8

The signal POP*p* goes to the SXBRs and pops processor *p*'s request off the processor interface queue. This comes from the XARBs (and an OR gate) on the XS0 and signifies that the request at the head of processor *p*'s queue won arbitration last cycle.

XS0_XS1.PUSH0
XS0_XS1.PUSH1
XS0_XS1.PUSH2
XS0_XS1.PUSH3
XS0_XS1.PUSH4
XS0_XS1.PUSH5
XS0_XS1.PUSH6
XS0_XS1.PUSH7
XS0_XS1.PUSH8

The signal PUSH*p* goes to the SXBRs and pushes processor *p*'s last request onto the processor interface queue in the SXBRs. This comes from the XARBs (and an OR gate) on the XS0 and signifies that processor *p*'s request for a memory board cannot immediately proceed and needs to be queued.

XS0_XS1.SEND_SEL0<3:0>
XS0_XS1.SEND_SEL1<3:0>
XS0_XS1.SEND_SEL2<3:0>
XS0_XS1.SEND_SEL3<3:0>
XS0_XS1.SEND_SEL4<3:0>
XS0_XS1.SEND_SEL5<3:0>
XS0_XS1.SEND_SEL6<3:0>
XS0_XS1.SEND_SEL7<3:0>
XS0_XS1.SEND_SEL8<3:0>

XS0_XS1.SEND_SEL*m* contains the number of the processor whose request will be sent to memory board *m* next cycle. These signals come from the XARBs (from buffers) on XS0.

XS0_XS1.SPARE0
XS0_XS1.SPARE1

These wires were put in as spares in case we need extra signals between the XS0 and the XS1.

XS1_HARD_ERR
XS1_HARD_ERR*

This differential signal is asserted when the XS1 has detected a hardware error. XS1 registers (see STOP_CLOCK) will be halted in normal mode unless HALT_DISABLE is asserted. This is internal to the error capture logic.

XS1_MB0.CTL_PAR<4:0>

XS1_MB1.CTL_PAR<4:0>
XS1_MB2.CTL_PAR<4:0>
XS1_MB3.CTL_PAR<4:0>
XS1_MB4.CTL_PAR<4:0>
XS1_MB5.CTL_PAR<4:0>
XS1_MB6.CTL_PAR<4:0>
XS1_MB7.CTL_PAR<4:0>
XS1_CU.CTL_PAR<4:0>

XS1_MB*m*.CTL_PAR is the parity check for the control signals (ADDR, CYCLE, and ZONE) to memory board *m* (NCU is *m*=8). Good parity should be present on these signals at all times. See the Parity Mapping table in the Processor Send Path Interface for details on good parity.

XS1_MB0.WR_DATA<31:0>
XS1_MB1.WR_DATA<31:0>
XS1_MB2.WR_DATA<31:0>
XS1_MB3.WR_DATA<31:0>
XS1_MB4.WR_DATA<31:0>
XS1_MB5.WR_DATA<31:0>
XS1_MB6.WR_DATA<31:0>
XS1_MB7.WR_DATA<31:0>
XS1_CU.WR_DATA<31:0>

XS1_MB*m*.WR_DATA is the write data bus to memory board *m* (NCU is *m*=8) containing data to be written to the memory board if the CYCLE is write or TAM.

XS1_MB0.WR_PAR<3:0>
XS1_MB1.WR_PAR<3:0>
XS1_MB2.WR_PAR<3:0>
XS1_MB3.WR_PAR<3:0>
XS1_MB4.WR_PAR<3:0>
XS1_MB5.WR_PAR<3:0>
XS1_MB6.WR_PAR<3:0>
XS1_MB7.WR_PAR<3:0>
XS1_CU.WR_PAR<3:0>

XS1_MB*m*.WR_PAR is the write parity bus to memory board *m* (NCU is *m*=8) containing the parity bits to check the WR_DATA bus. Good parity should be present on these signals at all times. See the Parity Mapping table in the Processor Send Path Interface for details on good parity.

XS1_MB0.WR_ZONE<3:0>
XS1_MB1.WR_ZONE<3:0>
XS1_MB2.WR_ZONE<3:0>
XS1_MB3.WR_ZONE<3:0>
XS1_MB4.WR_ZONE<3:0>
XS1_MB5.WR_ZONE<3:0>
XS1_MB6.WR_ZONE<3:0>
XS1_MB7.WR_ZONE<3:0>
XS1_CU.WR_ZONE<3:0>

XS1_MB*m*.WR_ZONE is the write zone bus to memory board *m* (NCU is *p*=8) which tells which bytes of the WR_DATA bus to write during a write or TAM

memory request. See the WR_ZONE to Data Byte Mapping table in the Processor Send Path Interface for more details.

XS1_MB0.SPARE0
XS1_MB0.SPARE1
XS1_MB1.SPARE0
XS1_MB1.SPARE1
XS1_MB2.SPARE0
XS1_MB2.SPARE1
XS1_MB3.SPARE0
XS1_MB3.SPARE1
XS1_MB4.SPARE0
XS1_MB4.SPARE1
XS1_MB5.SPARE0
XS1_MB5.SPARE1
XS1_MB6.SPARE0
XS1_MB6.SPARE1
XS1_MB7.SPARE0
XS1_MB7.SPARE1
XS1_CU.SPARE0
XS1_CU.SPARE1

These wires were put in as spares in case we need extra signals between the XS1 and the memory boards. They connect to the via farm only.

XS1_XC.HARD_ERROR

This signal tells the XCL that the XS1 has detected a hardware error and has halted. If not masked by the diagnostics, this will eventually halt the entire Neptune system. This is a registered version of STOP_CLOCK.

XS1_XC.SCAN_OUT

Scan data output from the XS1. Comes from an XS1 register (in the scan logic) and carries serial data from the XS1's registers to the scan engine.

XS1_XPC.BDTYPE<3,1>

This bus goes to the XBAR power controller from the XS1 to identify board type. Bits 3 and 1 are not attached to XPC_XR.BDTYPE_GND so they will float high; the others will be low. This sends a board type code of 1010 (binary) when the XS1 is plugged in. The XBAR power controller will complain if the wrong board is in an XS1 slot.

XS1_XPC.CLKLEDON

Clock LED on. As the name implies, when this signal is asserted the LED associated with the XS0 is on. This signal is asserted when the XS1 is in normal mode, clocks are running, and CLKLEDCLR is not asserted. Drives to the XBAR power controller from the heartbeat logic on the XS1. Clkledon's roamed Neptune during the Mesozoic era.

XS1_XPC.OVERTEMP

This is the output of the XS1's temperature sensor which drives out an analog voltage proportional to the temperature of the board. This signal is presumably read by an A/D converter in the XBAR power controller.

ZERO<20:0>

As the name implies, these signals should always be zero. Don't set them to one

or you will void the warranty unless you are a trained GENRAD professional. These signals are attached to terminators and pull inputs to zero. Some were included so nets could be forced to one for GENRAD testing.