A PCI Express to PCIX Bridge Optimized for Performance and Area

by

Margaret J. Chong

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

March 17, 2004

Author_____
Department of Electrical Engineering and Computer Science, May 2003

Certified by_____
Peter J. Jenkins, VI-A Company Thesis Supervisor

Certified by_____
Jeffrey LaFramboise, VI-A Company Thesis Supervisor

Certified by_____
Christopher J. Terman, M.I.T. Thesis Supervisor

Accepted by_____
Arthur C. Smith, Chairman, Department Committee on Graduate Theses

A PCI Express to PCIX Bridge Optimized for Performance and Area

by
Margaret Chong

Submitted to the
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

March 18, 2004

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Electrical Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis project involves the architecture, implementation, and verification of a high bandwidth, low cost ASIC digital logic core that is compliant with the PCI Express to PCIX Bridge Specification. The core supports PCI Express and PCIX transactions, x16 PCI Express link widths, 32 and 64-bit PCIX link widths, all PCI Express and PCIX packet sizes, transaction ordering and queuing, relaxed ordering, flow control, and buffer management. Performance and area are optimized at the architectural and logic levels. The core is approximately 27K gate count, runs at a maximum of 250 MHz, and is synthesized to a current standard technology. This thesis explores PCI Express, PCIX, and PCI technologies, architectural design, development of Verilog and Vera models, thorough module-level verification, the development of a PCI Express/PCIX system verification environment, synthesis, static timing analysis, and performance and area evaluations. The work has been completed in IBM Microelectronics in Burlington, Vermont as part of the MIT VI-A Program.

VI-A Company Thesis Supervisors: Peter Jenkins and Jeffrey LaFramboise
MIT Thesis Supervisor: Christopher Terman

Acknowledgements

This project would not have been possible without the guidance of my mentor Peter Jenkins, whose knowledge of hardware design, and IBM methodology and tools is unparalleled within IBM ASICs.

To my friend and advisor Jeff Laframboise, thank you for your sense of humor and zest for teaching. When I leave Vermont, I will miss you most of all.

There are so many people I would like to thank for making this thesis a success, but unfortunately I am running out of ways to word them. Thank you to Ben Drerup – an IBM expert on I/O bus protocols and a really great guy! Thank you to my manager Dave Sobczak for trusting me to a challenging project and giving me the opportunity to establish myself in the world of hardware design. Thank you to my thesis advisor Chris Terman for inspiring me to pursue architecture and circuit design when I was a sophomore taking 6.004. Also, many thanks go to verification gurus Frank Kampf and Bruce Ditmyer, my Vermont Dad Bob Fiorenza, and all the other individuals at IBM who have supported me during my time spent in Burlington.

Finally, I would like to thank Joseph Wong for his understanding and support throughout all five years at MIT. Your silly and sometimes very odd sense of humor never fails to lift my spirits, and your positive outlook on life is something we should all aspire to.

Table of Contents

List of Figures and Tables

1 Introduction

The conventional PCI technology bandwidth of 133 MBps has become a performance bottleneck due to significant improvements in processors and host systems. Multiple bus technologies have emerged to alleviate this bottleneck, including PCIX and PCI Express.

The PCIX parallel bus architecture was developed to increase the maximum theoretical bandwidth of up to 1 GBps by increasing timing constraints to support clock speeds of up to 133 MHz (from 66 MHz max freq for PCI). The concept of a Split transaction was also added to bring the realized performance closer to the theoretical BW.

The PCI Express protocol has also been developed as the next generation after PCIX, further increasing the maximum theoretical bandwidth to 8 GBps for a x16 (16 byte) link. Unlike the PCI and PCIX multi-drop bus architectures, PCI Express is a serial point-to-point interconnect. An advantage of PCI Express is that it has more bandwidth per pin, which results in lower cost and higher peak bandwidth.

Both PCI Express and PCIX are being widely adopted in industry, therefore it is desirable to bridge between the two protocols and allow both to coexist in the same system. This thesis outlines the development of a PCI Express to PCIX Bridge ASIC digital logic core optimized for performance and area.

2 Background

2.1 PCI

2.1.1 Overview of a Typical PCI System

Figure 2.1.1 shows a typical PCI system consisting of a Processor, a North Bridge, a PCI bus, a South Bridge, and various other components. The North Bridge interfaces the Processor to the graphics (AGP) bus, system memory, and the PCI bus. The PCI bus is also connected to the South Bridge and various high performance IO devices such as an Ethernet card. The South Bridge interfaces the PCI Bus to the ISA bus that connects to lower performance IO devices.

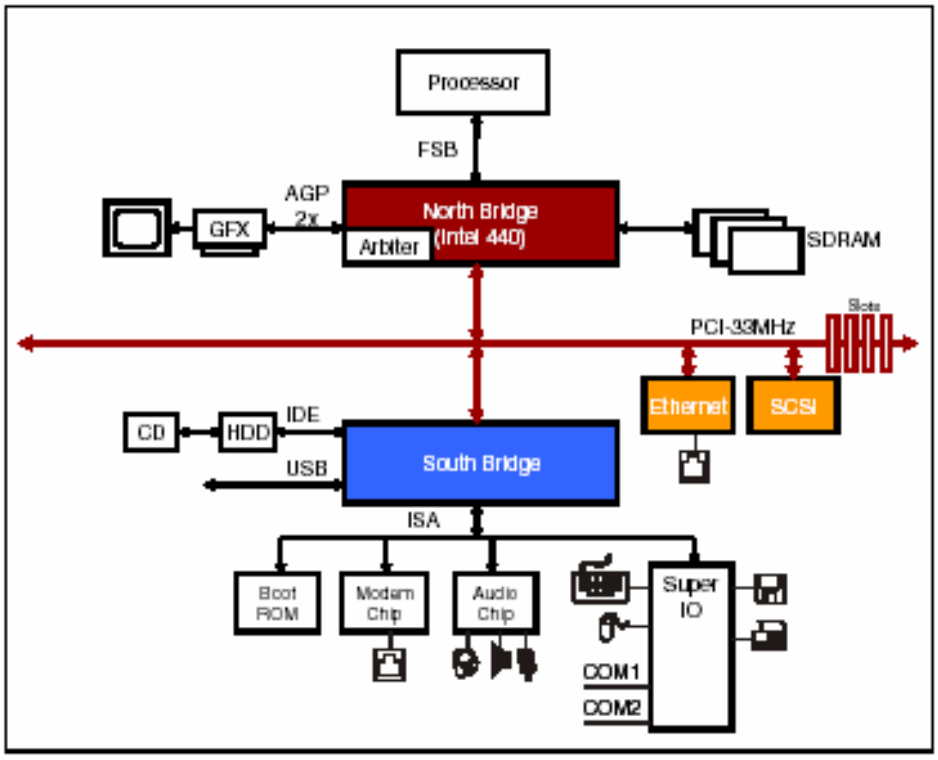


Figure 2.1.1 – A Typical PCI/PCIX System[1]

[1] Source: PCI Express System Architecture, Mindshare, Inc.

PCI supports twelve commands that allow the processor and various devices to communicate. Transactions include variations of memory, IO, and configuration reads and writes. Since PCI is a multi-drop bus (meaning that many devices might be connected to the bus at a time), all devices must win ownership of the bus from the North Bridge Arbiter before initiating a transaction.

2.1.2 PCI Correspondence Example

This section explores what happens when the processor issues an IO read to the Ethernet device. First, the processor issues an IO read cycle to the North Bridge. As illustrated in Figure 2.1.2, the North Bridge will then arbitrate to get control of the bus, and then issue an IO read on the PCI bus. The Ethernet device will claim the transaction, and if the data is ready and available the Ethernet device will drive the requested IO data on the bus.

If the data is *not* ready, however, the Ethernet device might respond with a Retry, turning the IO read into a *Delayed Transaction* and forcing the North Bridge to retry the IO read a few cycles later. The North Bridge unfortunately does not know *when* to retry, so the system may encounter situations where the North Bridge takes up valuable bus time unsuccessfully retrying the IO read while other PCI devices need the bus.

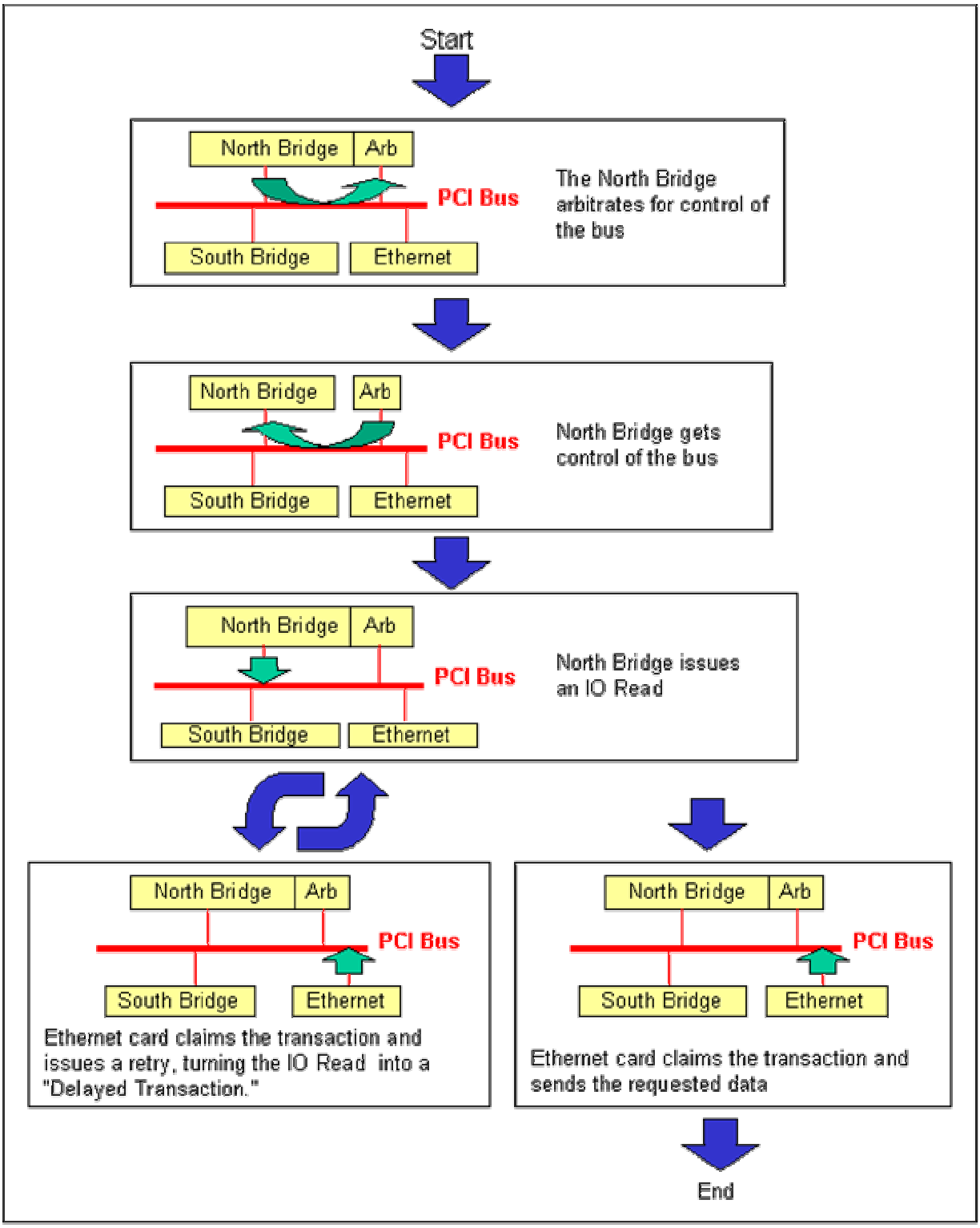


Figure 2.1.2 – PCI Correspondence Example

2.2 PCIX

PCIX builds on the PCI architecture by adding features to improve performance and bus efficiency. A significant difference between PCI and PCIX is that PCI Delayed transactions are replaced by PCIX Split transactions. In the PCI example discussed in Section 2.1.2, the North Bridge ties up the bus by repeatedly retrying the delayed IO read. If we take the same example from Section 2.1.2 but replace PCI with PCIX, as illustrated in Figure 2.2, the Ethernet device will memorize the transaction and signal a Split – telling the North Bridge not to retry the IO Read. When the data is ready, the Ethernet device will send the North Bridge a Split Completion containing the data. The addition of the PCIX Split Completion frees up the bus for other transactions, making PCIX more efficient than PCI.

PCIX utilizes clock speeds that range from 66 MHz to133 MHz, thus improving the data rate and performance over PCI. As the clock rate increases, it becomes more difficult to meet timing constraints with multiple devices connected to the bus. PCIX supports eight to ten devices at 66 MHz, and three to four devices at 133 MHz. For performance reasons, clock frequencies were increased to 266 or 533 MHz in PCIX 2.0 – Double Data Rate, sacrificing the multi-drop nature of the bus for a point-to-point connection that uses bridges to connect multiple devices. Unfortunately, PCIX bridges are not ideal for a point-to-point connection because of large pin count and area. Therefore a new bus technology emerged – PCI Express.

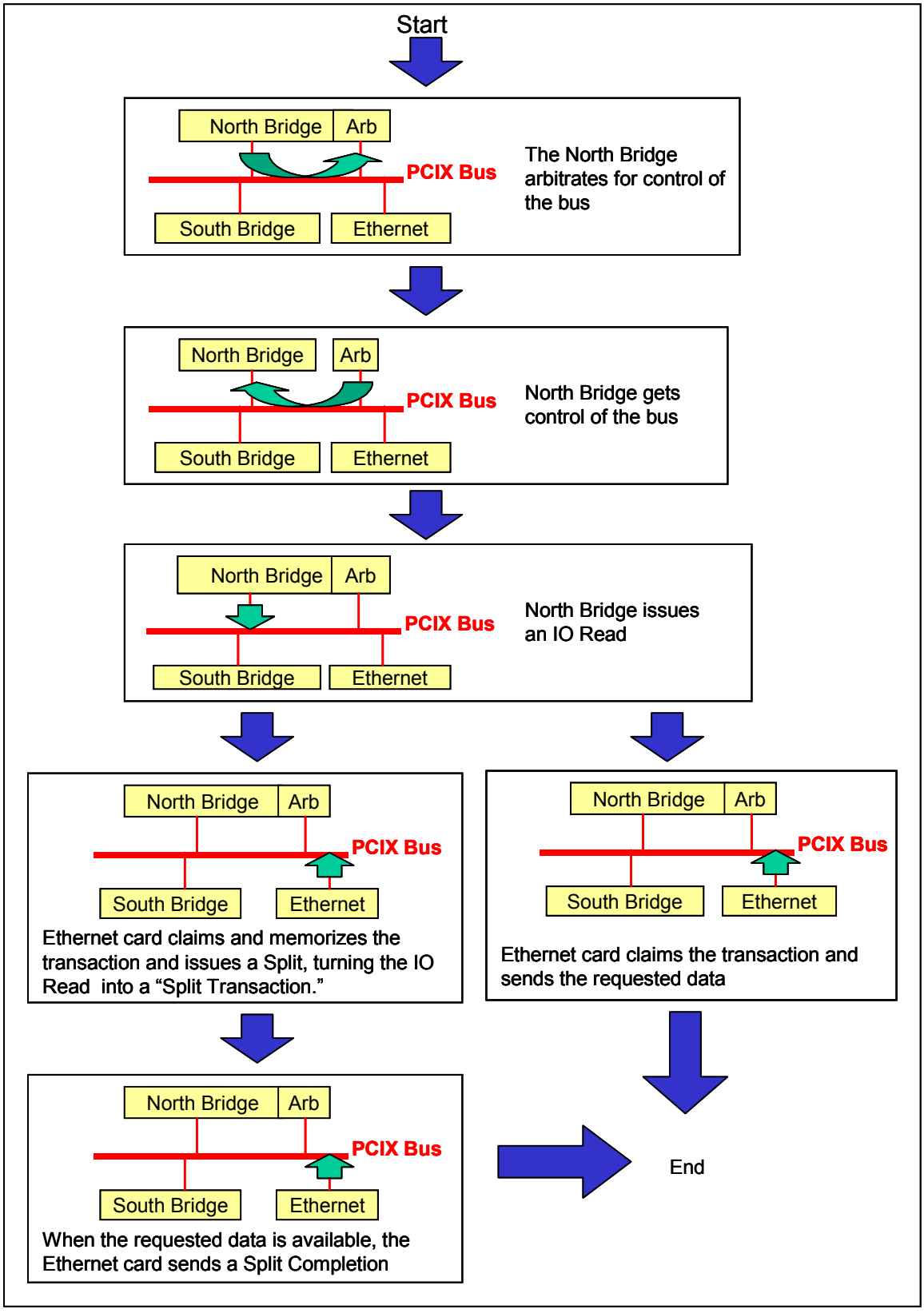


Figure 2.2 – PCIX Correspondence Example

2.3 PCI Express

PCI Express is a point-to-point link with a transmitter and receiver on both sides of the link. A PCI Express device can transmit and receive packets simultaneously. The link can be 1, 2, 4, 8, 12, 16, or 32 lanes wide in both directions with symmetric connections between the transmitting and receiving sides. PCI Express transactions include Memory, IO, and Configuration reads and writes, Completions, and various Message requests.

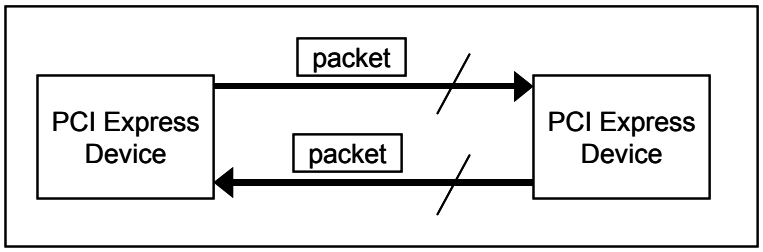


Figure 2.3.1 – PCI Express Link

A typical PCI Express system is shown in Figure 2.3.2. The bridge implemented in this thesis will operate in systems where the PCI Express link is upstream (closer to the CPU) as the primary interface, and the PCIX bus is downstream (farther away from the CPU) as the secondary interface. The root complex is the root of the PCI Express hierarchy. It allows connection of PCI Express devices, PCI Express Switches that route a PCI Express link to multiple PCI Express links, and PCI Express to PCI/PCIX Bridges.

A comparison of the two bus technologies used in this thesis, namely PCI Express and PCIX, is located in Table 2.3.

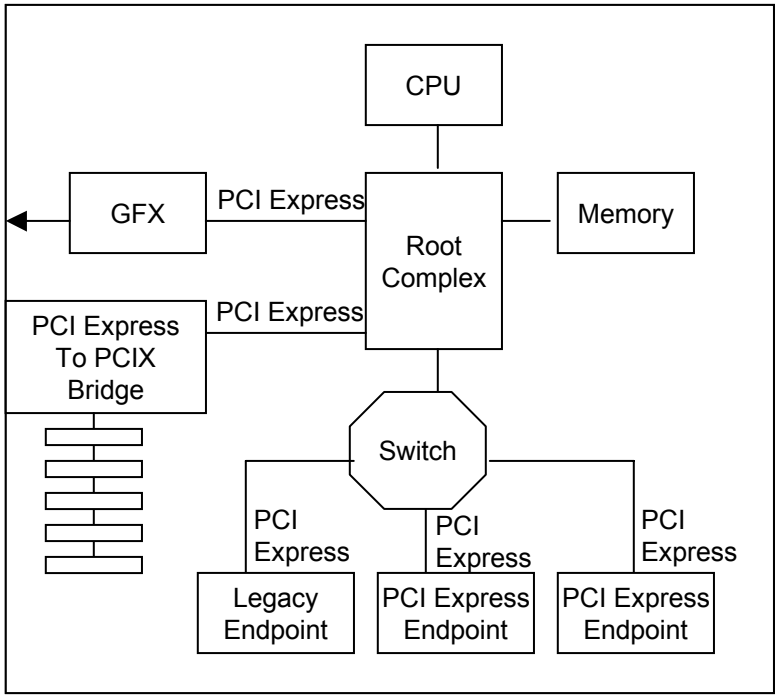


Figure 2.3.2 – Typical PCI Express System

	PCI Express (x16)	PCIX 133 MHz (64-bit bus)
Error Detection	Baseline and Advanced Error Reporting Capability, Link Layer LCRC, Transaction Layer ECRC, Poison bit in TLP	SERR# - System Error PERR# - Parity Error
Encoding	8b/10b encoding ECCC attached to packet	1 parity bit for every 32 data bits
Signaling Rate, Clock Frequency	2.5 GHz (x1, x2, x4, x8, x12, x16, x32)	133 MHz 32 bit and 64 bit lanes
# pins	64 pins (x16 64-bit bus)	90 pins (64-bit bus)
Peak Theoretical BW	8 GB/sec (x16 64-bit bus)	1 GB/sec
Performance Per Pin	125 MB/sec	11 MB/sec
Average Bandwidth	~ 40-60% peak theoretical 3.2 – 4.8 GB/sec	~ 50-70% peak theoretical 0.5 – 0.7 GB/sec
Arbitration Mechanism	Virtual Channels Arbitration, Port Arbitration, Quality of Service	Must arbitrate for sole use of the bus
Max. Physical Length	~ 10 yards	~ 1 foot
Transaction Acknowledgements and Flow Control	Non-posted – acknowledgement Posted – no acknowledgement Flow Control for non-posted, posted, and completion transactions	Master and target assert ready signals, then transmit entire transaction No Flow Control
Block Transactions	Present	Present
Split Transactions	Present	Present
Protocol for snoopy caches	No Snoop bit can eliminate snooping and improve performance during accesses to non-cacheable memory, Relaxed Ordering enable bit	No Snoop bit can eliminate snooping and improve performance during accesses to non-cacheable memory, Relaxed Ordering enable bit

Table 2.3 - A Comparison of PCI Express and PCIX[9], [16]

3 Project Requirements

3.1 IBM Soft Core Requirements

The design of the PCI Express to PCIX Bridge had to meet all IBM Methodology requirements for Soft Cores. All RTL code must be synthesize-able by standard EDA tools in order to be mapped to elements in the IBM ASIC Library. The elements in the standard libraries are static CMOS, which constrained this thesis from exploring various other technologies that cater to high performance such as Domino Logic.

Methodology requirements also constrained the physical aspects of the core to use the vendor's standard values, including wire load models, capacitance values, maximum and minimum delay between latches, maximum and minimum delay between the PCI Express Bridge to vendor ASIC PCI Express and PCIX cores, and the technology standard voltage and temperature ranges. Please see *Section 7 – Static Timing Analysis* for more detail.

3.2 PCI Express Bridge Requirements

The following contains the highlights of relevant key requirements compiled from Section 1.3.1 of the *PCI Express Bridge Specification.* Please see *Section 9.1 Future Work: Functionality* for a list of PCI Express to PCIX Bridge capabilities that should be explored in the future.

3.2.1 Supported Requirements

- The bridge includes one PCI Express primary interface and one or more PCIX secondary interfaces
- The bridge is compliant with the electrical specifications described in PCI Express Base 1.0a and PCIX 1.0a for its respective interfaces.
- Memory mapped I/O address space for transaction forwarding
- 64-bit addressing on both primary and secondary interfaces. The bridge must prevent address aliasing by fully decoding the address fields.
- The bridge must complete all DWORD and burst memory read transactions that originate from the secondary interface as Split Transactions if the transaction crosses the bridge and the originating interface is in a PCIX mode.
- Transactions that originate from PCI Express and address locations internal to the bridge have the same requirements as described for PCI Express Endpoints.

- PCI Express to PCI/PCIX bridges must not propagate exclusive accesses from the secondary interface to the primary and are never allowed to initiate an exclusive access of their own
- The PCI Express interface must comply with the definition of the flow control mechanism described in PCI Express Base 1.0a.

3.2.1 Unsupported Requirements

Configuration requirements from Section 1.3.1 of the *PCI Express Bridge Specification* are not supported because a Bridge configuration space is not included in this thesis.

- The bridge includes configuration registers accessible through the PCI-compatible configuration mechanism.
- As with PCI bridges and PCIX bridges, PCI Express to PCI/PCIX bridges us a Type 01h Configuration Space header.

3.3 PCI Express to PCIX Bridge Features

- The PCI Express to PCIX Bridge supports the following PCI Express and PCIX transactions in both upstream and downstream directions: Memory Writes, Memory Reads, I/O Writes, I/O Reads, Type 1 Configuration Writes, Type 1 Configuration Reads, Completions with Data, Completions without Data, and Split Transactions.

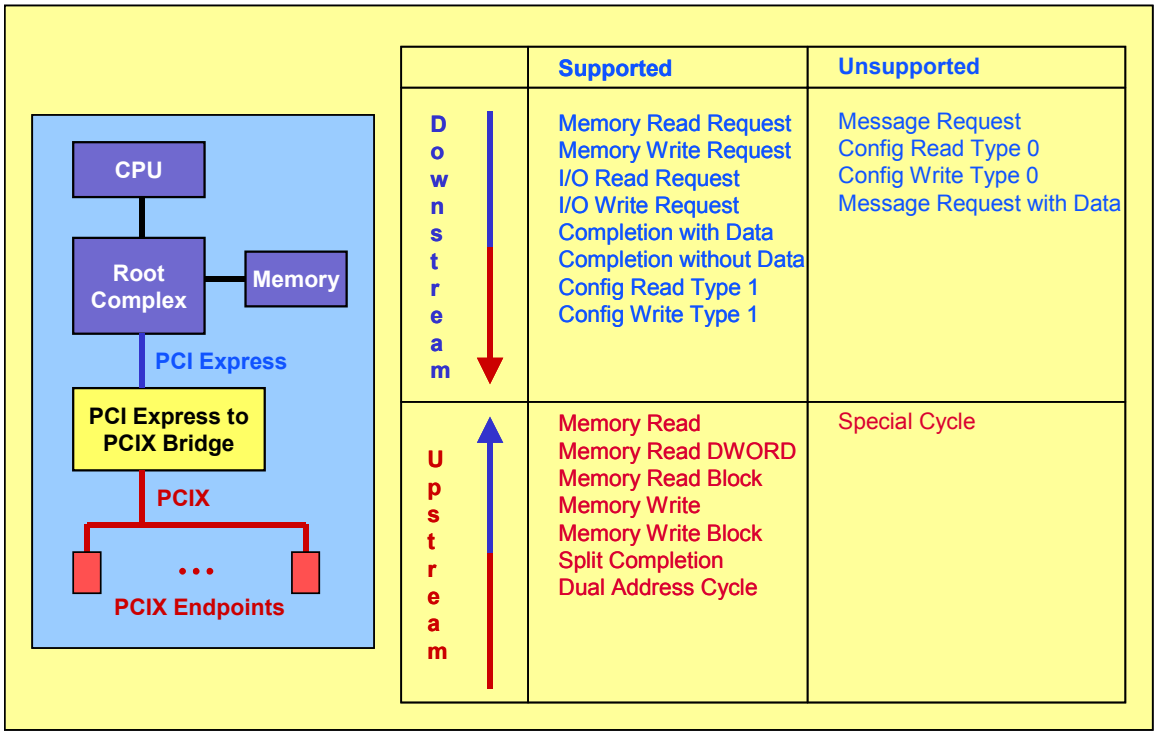


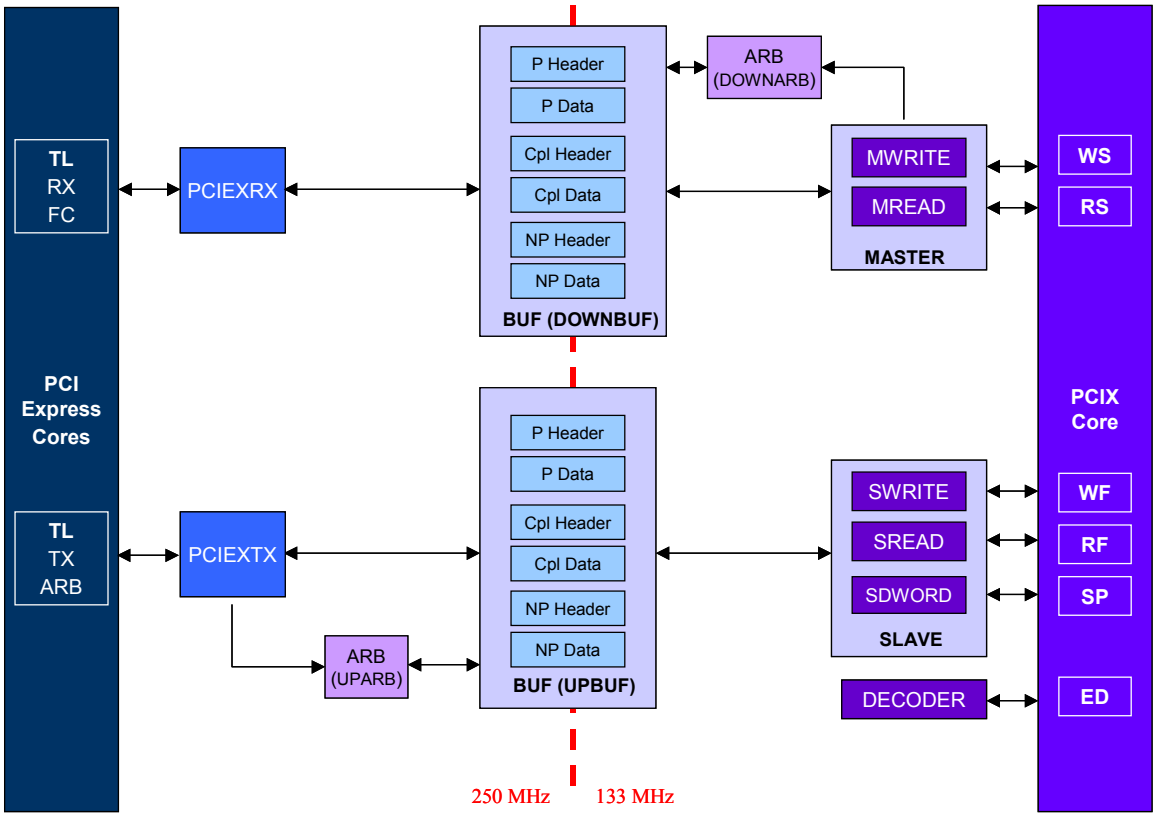
Figure 3.3 – PCI Express and PCIX Transactions

- The PCI Express to PCIX Bridge is compliant with the following specifications: PCI Express to PCI/PCI-X Bridge 1.0, PCI Express Base 1.0a, PCI-X 1.0a, PCI 2.3.
- Supports x16 link widths providing 2.5 Gbps data rate per lane per direction
- Supports the following Transaction Layer Packet (TLP) sizes:
    - Max payload size of 4KB or less for posted transactions and completions
    - Max read request size of 4KB or less for non-posted requests

- Supports Virtual Channel 0 (VC0)
- Supports PCI Express Transaction Layer functions including:
    - Transaction Layer Packet Interface transmit and receive
    - Transaction ordering and queuing
    - PCIX and PCI Express relaxed ordering model
    - PCI Express Flow Control and Buffer Management
- Provides internal buffering for up to three outstanding downstream transactions: one non-posted transaction, one posted transaction, and one completion
- Provides internal buffering for up to three outstanding upstream transactions: one non-posted transaction, one posted transaction, and one completion
- Supports one PCI Express primary interface and one PCIX secondary interface
- Memory mapped I/O address space for transaction forwarding
- 64-bit addressing on both primary and secondary interfaces

4 Architecture and Implementation

4.1 High Level Overview



TL RX	PCI Express Transaction Layer Packet Interface Receive Interface
TL FC	PCI Express Transaction Layer Packet Interface Flow Control Interface
PCIEXRX	PCI Express Transaction Layer Packet Receiver
DOWNBUF	Downstream Buffers
DOWNARB	Downstream Arbiter
MWRITE	PCIX Write Master
MREAD	PCIX Read Master
WS	PCIX Write Transmitter Interface
RS	PCIX Read Transmitter Interface
SWRITE	PCIX Write Slave
SREAD	PCIX Read Slave
SDWORD	PCIX Dword Slave
WF	PCIX Write Receiver Interface
RF	PCIX Read Receiver Interface
ED	PCIX External Decoder Interface
DECODER	PCIX Address and Command Decoder
UPBUF	Upstream Buffers
UPARB	Upstream Arbiter
PCIEXTX	PCI Express Transaction Layer Packet Transmitter
TL TX	PCI Express Transaction Layer Packet Interface Transmit Interface
TL ARB	PCI Express Transaction Layer Packet Interface Arbitration Interface

Figure 4.1 – High Level Overview of Bridge Architecture

	Module	Description
Downstream – PCI Express to PCIX	PCIEXRX	Receive PCI Express Transaction Layer Packets (1) Receive PCI Express Header and Data from PCI Express Transaction Layer Packet Interface (2) Translate PCI Express Header into PCIX Control Signals (3) Store PCIX control signals and data in a downstream buffer (4) Initialize and update Flow Control Credits
	BUF (DOWNBUF)	Downstream Buffers include: Three header buffers of equal size (128 bits) Three data buffers: One 4KB data buffer for posted data One 4KB data buffer for completions One 1DW (4byte) buffer for non-posted data
	ARB (DOWNARB)	(1) Decide which transaction to send next according to ordering rules (2) Select type (P-posted, CPL-completion, NP-nonposted) to transmit
	MASTER (MWRITE)	Master a PCIX write (1) Initiate a PCIX Write (2) Push data straight from DOWNBUF to PCIX Interface (3) Indicate if transaction was successful or needs to be retried
	MASTER (MREAD)	Master a PCIX read (1) Initiate a PCIX Read (2) Assume that the PCIX target will always split the transaction (3) Indicate if transaction was successful or needs to be retried
Upstream – PCIX to PCI Express	DECODER	When the Bridge receives a PCIX transaction: (1) Determine if the Bridge should claim the transaction (2) Determine which port should handle the transaction. (WF/RF/SP)
	SLAVE (SWRITE)	Receive PCIX writes from the PCIX Interface WF interface (1) Receive a PCIX write from the PCIX Interface (2) Translate PCIX control signals into a PCI Express Header (3) Store PCI Express Header and Data in an upstream buffer
	SLAVE (SREAD)	Receive PCIX reads from the PCIX RF interface (1) Receive a PCIX read from PCIX Interface (2) Translate PCIX control signals into a PCI Express Header (3) Split the transaction (4) Store PCI Express Header and Data in an upstream buffer
	SLAVE (SDWORD)	Receive PCIX DWORD transactions from the PCIX SP interface (1) Receive a PCIX read or write from PCIX Interface (2) Translate PCIX control signals into a PCI Express Header (3) If the transaction is non-posted, then split the transaction (4) Store PCI Express Header and Data in an upstream buffer
	BUF (UPBUF)	Identical to BUF (DOWNBUF) described above
	ARB (UPARB)	Identical to ARB (DOWNARB) described above
	PCIEXTX	Transmit PCI Express Transaction Layer Packets (1) Obtain an arbitration grant from the PCI Express Transaction Layer Packet Interface (2) Transmit the PCI Express Header and Data (3) Indicate when a transaction has submitted successfully

Table 4.1 - Description of the Bridge Architecture

4.1.1 Downstream Transaction

The following steps illustrate what happens when the Bridge handles a downstream transaction, traveling from PCI Express to PCIX.

1) Transaction Layer Packet Interface sends the Bridge a PCI Express Transaction Layer Packet containing a PCI Express Header (all transactions), and a data payload (writes and completions)
2) PCIEXRX translates the Transaction Layer Packet header into PCIX control signals
3) PCIEXRX sends the PCIX control signals and data to DOWNBUF, where the transaction is stored in the appropriate downstream buffer. Memory writes are stored in the posted buffer. Completions are stored in the completion buffer. IO reads and writes, Configuration reads and writes, and Memory reads are stored in the non-posted buffer.
4) DOWNBUF tells DOWNARB that there is a pending transaction and indicates if it is a posted transaction, a non-posted transaction, or a completion.
5) When the PCIX side is idle, DOWNARB tells DOWNBUF to transmit the transaction
6) DOWNBUF sends the PCIX control signals and data to the MASTER
7) If the transaction is a Completion or a Memory/IO/Configuration write, then MWRITE will initiate a PCIX write on the WS interface. If the transaction is a Memory/IO/Configuration read, then MREAD will initiate a PCIX read on the RS interface.
8) WS/RS ends the transaction

9) MWRITE/MREAD tells DOWNARB and DOWNBUF that the transaction has completed

11a) If the transaction was successful, DOWNBUF frees the buffer and PCIEXRX updates the flow control credits

11b) If the transaction was unsuccessful, the buffer is not freed. Go to Step 5.

4.1.2 Upstream Transaction

The following steps illustrate what happens when the Bridge handles an upstream transaction, traveling from PCIX to PCI Express.

1) ED port asks DECODER whether or not it should claim the PCIX transaction
2) DECODER instructs the ED to claim the transaction on either the WF, the RF, or the SP port
3) WF/RF/SP sends the PCIX transaction to the SLAVE. Memory writes and Completions are handled by the SWRITE module. Memory reads are handled by the SREAD module. IO reads and writes are handled by the SDWORD module.
4) SWRITE/SREAD/SDWORD translates the PCIX transaction into a PCI Express Transaction Layer Packet and sends it to UPBUF where it is stored in the appropriate buffer. Memory writes are stored in the posted buffer. Completions are stored in the completion buffer. IO reads and writes and Memory reads are stored in the non-posted buffer.

5) UPBUF tells UPARB that there is a pending transaction and indicates if it is a posted transaction, a non-posted transaction, or a completion
6) When the PCI Express side is idle, UPARB tells UPBUF to transmit the transaction
7) UPBUF starts to send the PCI Express Header to PCIEXTX
8) PCIEXTX obtains a grant from the ARB port
9) PCIEXTX transmits the PCI Express Transaction Layer Packet on the TX port
10) PCIEXTX tells UPARB and UPBUF that the transaction has completed
11) UPBUF tells SLAVE that the buffer has been freed

4.2 PCIEXRX

The PCIEXRX module receives PCI Express Transaction Layer Packets, translates the Transaction Layer Packet header into PCIX control signals, and sends the PCIX control signals and data payload to the DOWNBUF module. PCIEXRX also initiates flow control credits after a system reset and updates flow control credits whenever a buffer is freed.

The timing diagram located in Figure 4.2 illustrates the behavior of PCIEXRX interfaces when receiving a PCI Express posted transaction with a 4DW data payload.

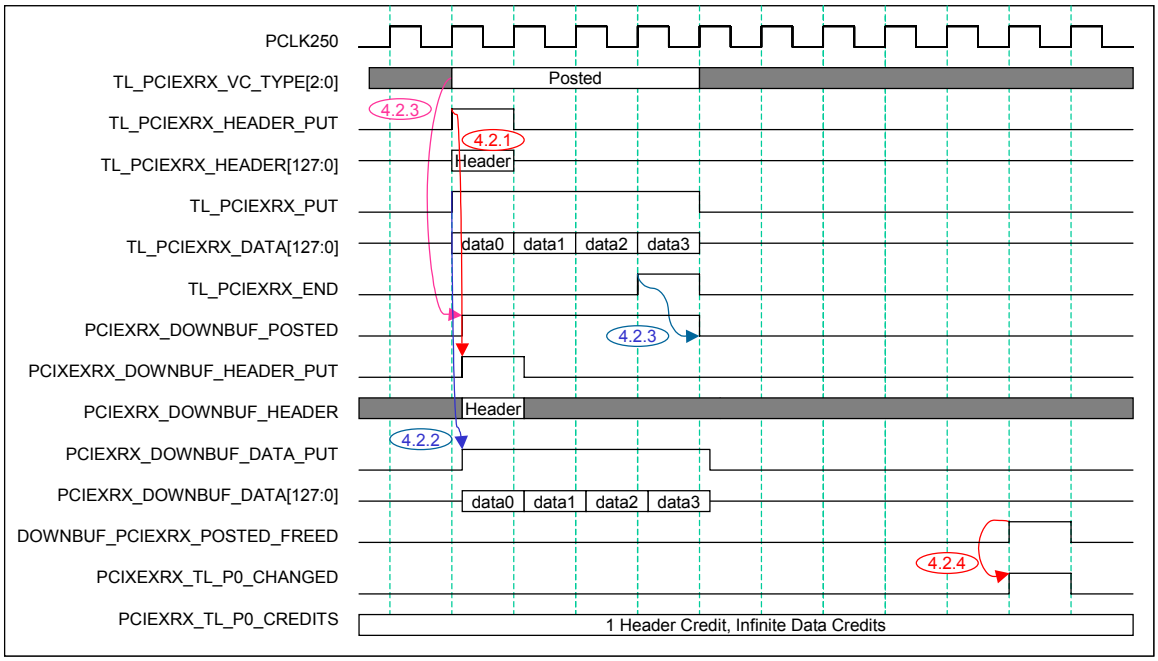


Figure 4.2 – PCIEXRX Timing Diagram

4.2.1 Header

When TL_PCIEXRX_HEADER_PUT is high, PCIEXRX will (1) translate the PCI Express header into a PCIX header containing PCIX control signals, and (2) write that PCIX header to a buffer by asserting PCIEXRX_DOWNBUF_HEADER_PUT.

4.2.1.1 PCI Express Header

As illustrated in Figure 4.2.1.1, there are four general categories of PCI Express Transaction Layer Packet Headers: 4DW Memory, 3DW Memory and IO Headers, Type 1 Configuration Headers, and Completion Headers. The Header fields are explained in Table 4.2.1.1.1.

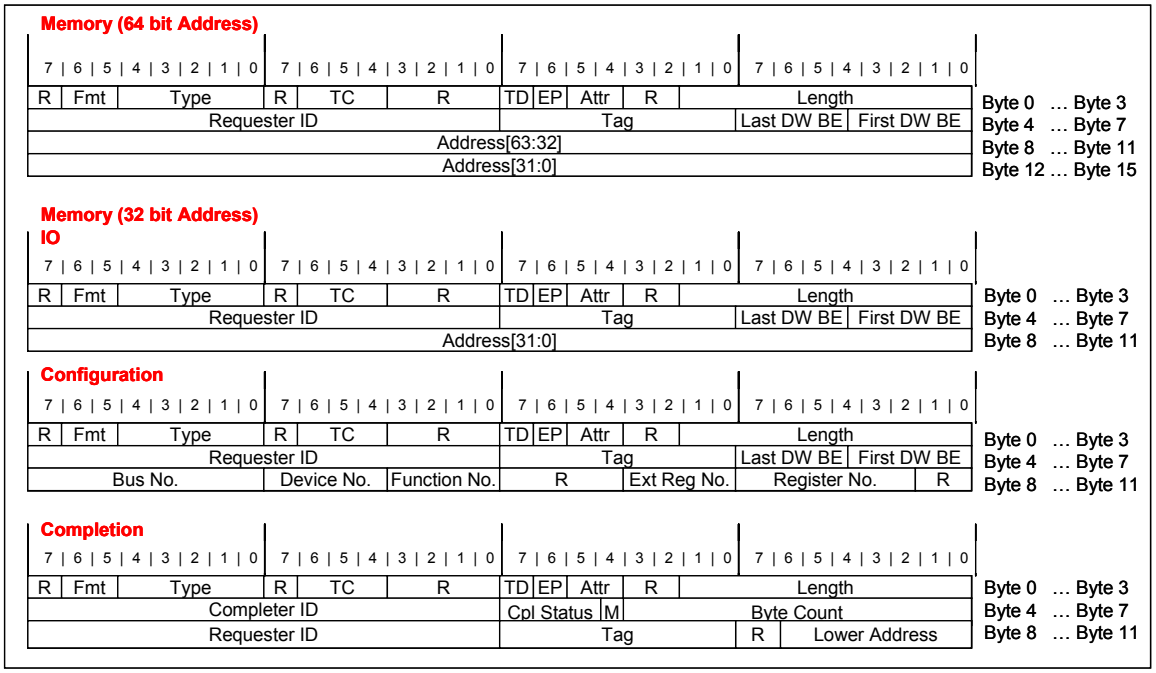


Figure 4.2.1.1 – PCI Express Headers

Header Field	# bits	Description
Address	32/64	32-bit or 64-bit Address
Attr[2]	2	Attribute[1:0] = {High bit = Relaxed Ordering, Low bit = No Snoop} Attribute[1] = 1, PCIX Relaxed Ordering Model Attribute[1] = 0, PCI Strongly Ordered Model Attribute[0] = 1, No snoop required (cache coherency not required) Attribute[0] = 0, Snoop required (cache coherency required)
Bus No.	8	Bus Number
Byte Count[4]	11	Byte Count
Completer ID	16	{Bus Number, Device Number, Function Number} of the Completer
Cpl Status[3]	3	Completion Status Code 000b = Successful Completion 001b = Unsupported Request 010b = Configuration Request Retry Status 100b = Completer Abort Others = Reserved
Device No.	4	Device Number
EP	1	1 if the Transaction Layer Packet data is poisoned and invalid
Ext. Reg. No.	4	External Register Number
First DW BE	4	First DW Byte Enable
Fmt	2	Format 00b = 3DW header, no data 01b = 4DW header, no data 10b = 3DW header, with data 11b = 4DW header, with data
Function No.	4	Function Number
Last DW BE	4	Last DW Byte Enable
Length	10	Transfer Length in DW 0000000001b = 1 DW … 1111111111b = 1023 DW 0000000000b = 1024 DW
Lower Address	7	In memory read completions, the Lower Address field contains the byte address for the first enabled byte of data returned with the completion. The field is cleared for all other types of completions.
M[4]	1	Byte Count Modified – Set for the first completion in a multiple completion sequence when the Byte Count field has been modified and contains the count for this completion only, not the total remaining
R	N/A	Reserved
Register No.	6	Register Number
Requester ID	16	{Bus Number, Device Number, Function Number} of the Requester
Tag	8	Used by a requester to uniquely identify its outstanding transactions
TC[5]	3	Traffic Class to indicate Quality of Service
TD[6]	1	1 if there is a digest field included in the Transaction Layer Packet
Type	5	When combined with Fmt, indicates the transaction type

Table 4.2.1.1.1 – PCI Express Headers

[2] This thesis assumes that the system is Strongly Ordered and Cache Coherent, Attr[1:0] = 0b
[3] This thesis assumes that Completions are always successful
[4] This thesis assumes that a Completion will contain all data, M=0b and Byte Count = 0b
[5] This thesis assumes that Traffic Class is always a default 0b
[6] This thesis assumes that there is no digest and TD = 0b

TLP	Fmt[1:0]	Type[4:0]	Description
MRd	00 01	00000	Memory Read
MWr	10 11	00000	Memory Write
IORd	00	00010	IO Read
IOWr	10	00010	IO Write
CfgRd1	00	00101	Type 1 Configuration Read
CfgWr1	10	00101	Type 1 Configuration Write
CfgRd0	00	00100	Type 0 Configuration Read
CfgWr0	10	00100	Type 0 Configuration Write
Cpl	00	01010	Completion
CplD	10	01010	Completion with Data
Msg	01	10rrr	Message Request, No Data
MsgD	11	10rrr	Message Request, With Data

Table 4.2.1.1.2 – PCI Express Transaction Type

4.2.1.1 PCIX Header

The PCIEXRX module translates a PCI Express Transaction Layer Packet Header into a PCIX Header pictured in Figure 4.2.1.1. There is no concept of a "Header" in the PCIX architecture. In this thesis, "PCIX Header," refers to a collection of PCIX control signals described in Table 4.2.1.1.1.

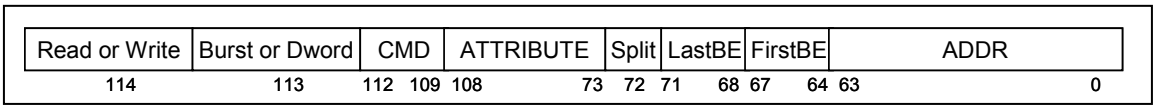


Figure 4.2.1.1.1 – PCIX Header

Field	# of bits	Description
Read or Write	1	0 for a Read, 1 for a Write/Completion
Burst or Dword	1	0 for Burst, 1 for Dword
CMD	4	PCIX CMD field
ATTRIBUTE	36	PCIX Attribute field
Split	1	1 for a Completion only
LastBE	4	Byte Enable for the last DW of data
FirstBE	4	Byte Enable for the first DW of data
ADDR	64	Address

Table 4.2.1.1.1 – PCIX Header

The Read or Write field is determined by the PCI Express Transaction Layer Packet Header Fmt and Type fields – MRd, IORd, and CfgRd1 are classified as "Reads," while MWr, IOWr, CfgWr1, Cpl, and CplD are classified as "Writes."

The PCI Express Transaction Layer Packet Header Length field determines the Burst or Dword field. Dword transactions have a 1 DW Length, otherwise the transaction is classified as a Burst transaction (more than one DW).

The PCI Express Transaction Layer Packet Header Fmt and Type fields determine the CMD field. The translation is summarized in Table 4.2.1.1.2.

PCI Express Command	PCI Express Fmt, Type	PCIX CMD	PCIX Command
Memory Read Request	00 00000 01 00000	0110	Memory Read DWORD Memory Read Block
Memory Write Request	10 00000 11 00000	0111	Memory Write Memory Write Block
IO Read Request	00 00010	0010	IO Read
IO Write Request	10 00010	0011	IO Write
Completion	00 01010	1100	Split Completion
Completion with Data	10 01010	1100	Split Completion
Type 1 Configuration Read	00 00101	1010	Configuration Read
Type 1 Configuration Write	10 00101	1011	Configuration Write

Table 4.2.1.1.2 – Translating Commands from PCI Express to PCIX

As illustrated in Figure 4.2.1.1.2, the PCIX ATTRIBUTE field can be formatted in four different ways: Burst transaction, DWORD transaction, Configuration transaction, and Completion transaction. The Byte Enables are taken straight from the PCI Express First DW BE field. The No Snoop (NS) and Relaxed (RO) Ordering bits are taken from the PCI Express Attr field. The Tag field is mapped straight from the PCI Express Tag field.

The Requester Bus Number, Device Number, and Function Number are either mapped straight from the PCI Express Header or set to the appropriate values from the Bridge's Configuration space. The Upper and Lower Byte Counts are calculated by shifting the PCI Express Header Length field left by two bits and then altering the value depending on the first and last DW byte enables. The M field, Completer Bus Number, Device Number, and Function Numbers are mapped straight from the PCI Express Header.

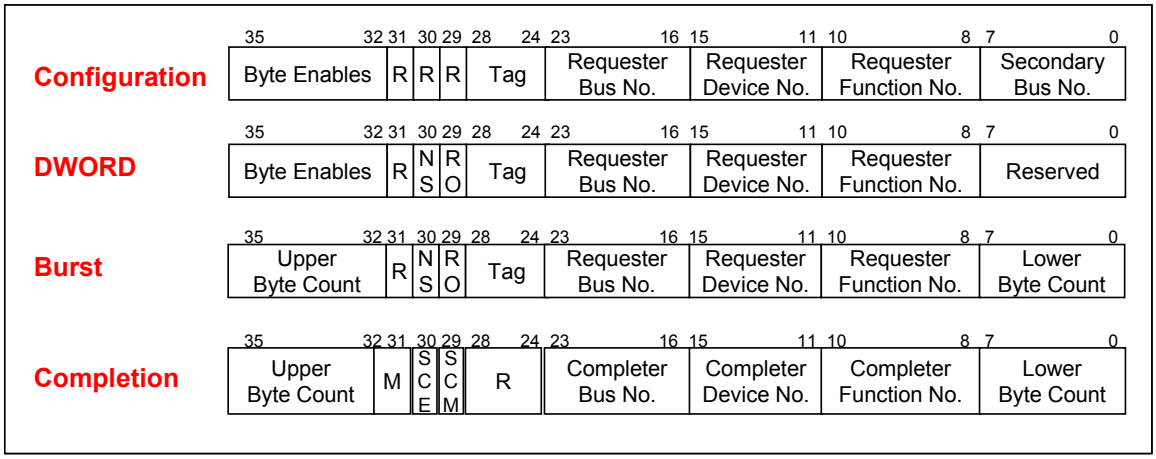


Figure 4.2.1.1.2 – PCIX Attribute

The Split field of the PCIX Header is set only if the PCI Express Transaction Layer Packet Header Fmt and Type fields indicate that the transaction is a Cpl or a CplD. This field differentiates a Split Completion from a PCIX Write.

The LastBE and FirstBE fields are the negative enabled version of the PCI Express Transaction Layer Packet Header Last DW BE and First DW BE fields respectively.

The PCIX ADDR field contains a 64-bit version of the PCI Express Transaction Layer Packet Header Address field for Memory, IO, and Configuration transactions. However, the PCIX ADDR field takes on a different form for Split Completions, as pictured in Figure 4.2.1.1.3. All PCIX ADDR fields are mapped straight from the PCI Express Transaction Layer Packet Header: Relaxed Ordering bit (RO), Requester Bus Number, Requester Device Number, Requester Function Number, and the Lower Address.

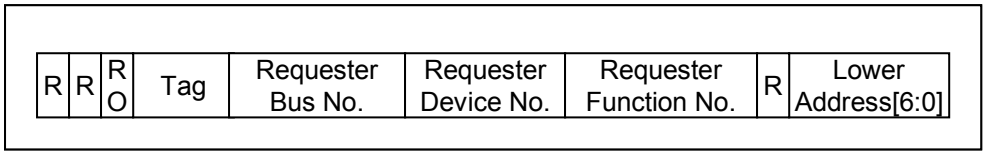


Figure 4.2.1.1.3 – PCIX ADDR field for Split Completions

4.2.2 Receiving the PCI Express Data Payload

When TL_PCIEXRX_PUT is high, PCIEXRX will map the data from TL_PCIEXRX_DATA to PCIEXRX_DOWNBUF_DATA and write the data to a buffer by asserting PCIEXRX_DOWNBUF_DATA_PUT.

4.2.3 Select the buffer for packet storage

In the PCI Express transaction being received in Figure 4.2, TL_PCIEXRX_VC_TYPE indicates that it is a posted transaction, therefore PCIEXRX will write the translated PCIX control signals and data to the *posted* buffer by asserting PCIEXRX_DOWNBUF_POSTED until the Transaction Layer Packet Interface signifies the end of the Transaction Layer Packet by asserting TL_PCIEXRX_END for one cycle.

Handling completions and non-posted transactions is nearly identical to handling posted transactions.

TL_PCIEXRX_VC_TYPE[2:0]	Selected Buffer
001	Posted
010	Non-posted
100	Completion

Table 4.2.3 – PCIEXRX Selection of Buffer

4.2.4 Initialize and Update Flow Control Credits

PCI Express includes the concept of flow control to ensure that PCI Express Receivers will always have buffer space to store incoming transactions. Most of the flow control functionality is implemented in the Transaction Layer Packet Interface. The details concerning the flow control implementation of the Transaction Layer Packet Interface will not be presented in this thesis. The flow control logic in the PCIEXRX module performs two functions: (1) initialize the flow control credits after system reset, and (2) update the flow control credits every time a buffer is *freed*.

When the system has just been reset, the PCI Express Transaction Layer Packet Interface is initialized with the number of Header and Data flow control Credits that the Bridge can support for non-posted, posted, and completion packets. There is one flow control header credit value and one flow control data credit value for each transaction type – posted, non-posted, and completion. Since the Bridge only has one buffer for each type, all three flow control header credits are equal to one. The Bridge supports the maximum data payloads for each type – 1024 DW for posted and completion transactions, and 1 DW for

non-posted transactions. One data flow control credit is equivalent to 4 DW, or 16 Bytes. Since it is a performance advantage to advertise as many flow control credits as possible, the Bridge advertises infinite data flow control credits for posted, completion, and non-posted transactions

The PCI Express Transaction Layer Packet Interface indicates that the PCIEXRX module needs to initialize flow control credits by asserting TL_AL_NEED_CREDITS_VC0 for one cycle. The PCIEXRX module will immediately assert PCIEXRX_TL_*_CHANGED for one cycle with the appropriate flow control values on PCIEXRX_TL_*_CREDITS.

In addition to initializing the flow control credits, the PCIEXRX Flow Control logic must also update the flow control credits whenever buffer space is freed. In the example presented in the timing diagram in Figure 4.2, a downstream posted transaction has been successfully transmitted on the PCIX side of the Bridge. DOWNBUF_PCIEXRX_POSTED_FREED is asserted for one cycle to indicate that a posted buffer has been freed, triggering PCIEXRX to update the posted PCI Express flow control credits by asserting PCIEXRX_TL_P0_CHANGED for one cycle with on header flow control credit, and infinite data flow control credits on PCIEXRX_TL_P0_CREDITS.

4.3 BUF

The BUF module provides the clock boundary between the PCI Express 250 MHz clock and the PCIX 133 MHz clock, as well as the interface between the Bridge logic and

physical memory. Incorporating a buffer interface rather than forcing other modules to directly access physical memory allows for a simpler interface to the buffers and provides flexibility in the choice of memory. Adding new buffers, changing buffer sizes, or changing the implementation from an SRAM to a register file will not affect the BUF interface.

The BUF module pictured in Figure 4.3 contains three submodules: HEADERBUFFERS, DATABUFFERS, and BUFFERSTATUS. HEADERBUFFERS contains all three header buffers – one posted, one completion, and one non-posted. DATABUFFERS contains all three data buffers – one posted, one non-posted, and one completion. BUFFERSTATUS keeps track of which buffers are full and which are empty.

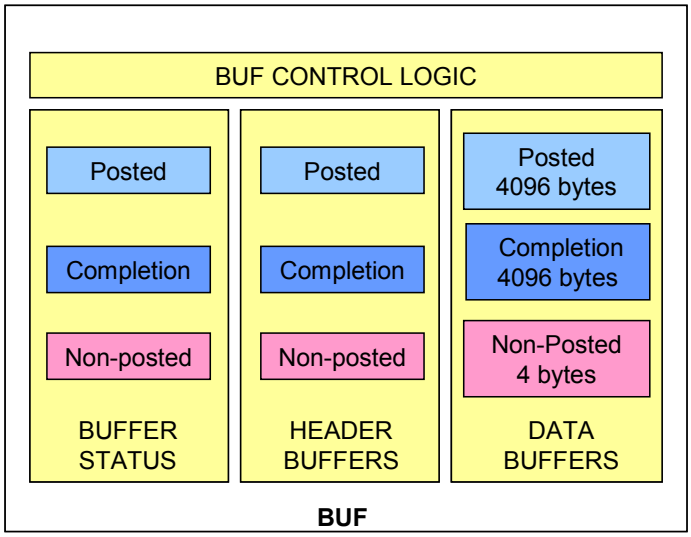


Figure 4.3 – BUF Architecture

4.3.1 HEADERBUFFERS

There are three identical header buffers, one for each type – posted, completion, and non-posted. All header buffers are 16 bytes wide and are implemented with latches because,

in this particular case, a latch implementation is more efficient in size and speed when compared to an SRAM. Data is written into a header buffer synchronously and read asynchronously, as illustrated in Figure 4.3.1.

After a buffer is written to, it is guaranteed that the corresponding read enable (RE) will not assert until at least two cycles after the data is written and the write enable (WE) has fallen.

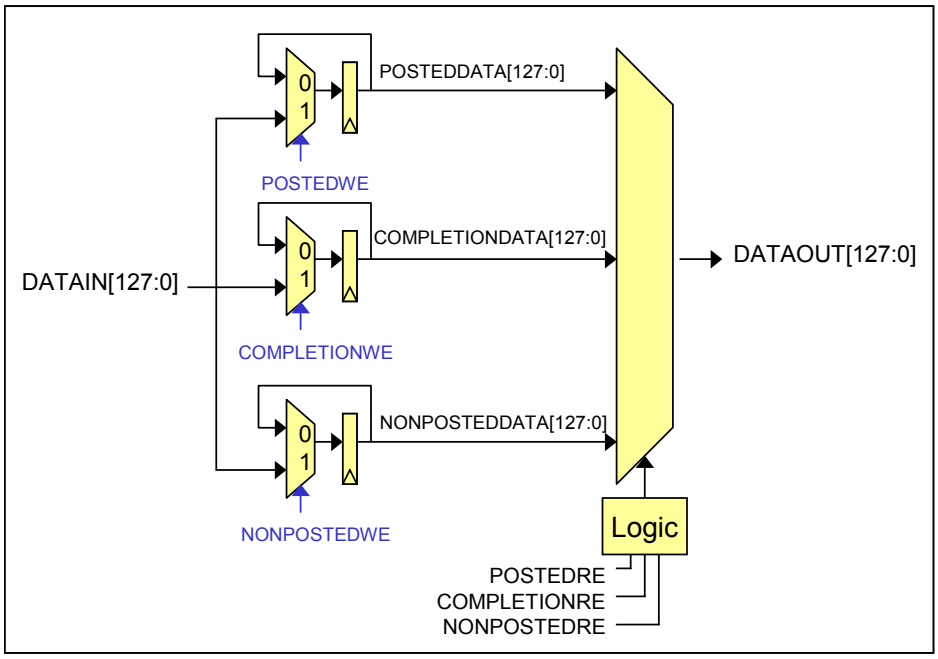


Figure 4.3.1 – Header Buffer Implementation

4.3.2 DATABUFFERS

As illustrated in Figure 4.3.2, there are three data buffers, one for each type – posted, completion, and non-posted. For simplicity, the size of the data buffers reflects the maximum data payload size for the corresponding transaction type. The posted data

buffer is 4 KB (256 rows that are 16 bytes wide), the completion data buffer is 4 KB (256 rows that are 16 bytes wide), and the non-posted data buffer is 4 bytes. The posted and completion data buffers are large and most efficiently implemented with two-port SRAMs. One port of the SRAM is used as the write port and writes the data on DATAIN to address WRITEADDR when the WE write enable is asserted on a WRITECLK clock edge. The other port of the SRAM is used as the read port and reads the data at address READADDR. The non-posted data buffer is 4 bytes so it is implemented with latches. The final DATAOUT will contain the data from whichever buffer is read-enabled.

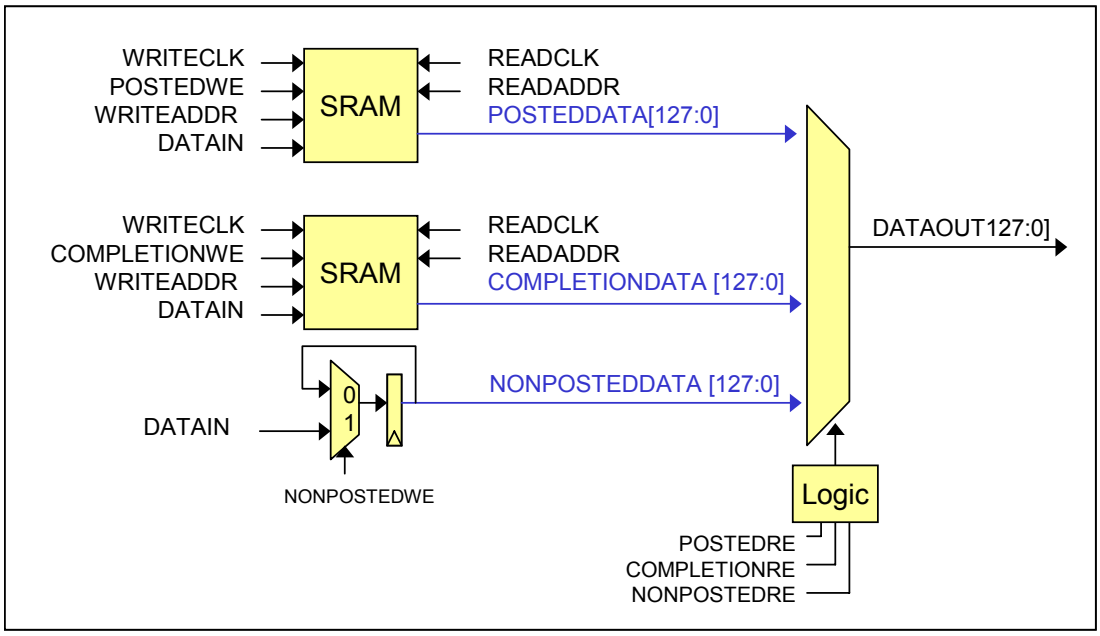


Figure 4.3.2 – Data Buffer Implementation

4.3.3 BUFFERSTATUS

The BUFFERSTATUS module indicates whether a buffer is (1) full and there is a transaction pending, or (2) has been freed and the buffer is empty with no transaction pending. There is one set of status signals for each of the three buffers. As displayed in

Figure 4.3.3.1, transactions flow from the ACLK clock domain to the BCLK clock domain. The buffer status signals pass through the clock boundary using a HANDSHAKE or a CYCLEHANDSHAKE module. Refer to Table 4.3.3 for a description of the posted buffer status signals, which are similar to the completion and non-posted buffer status signals.

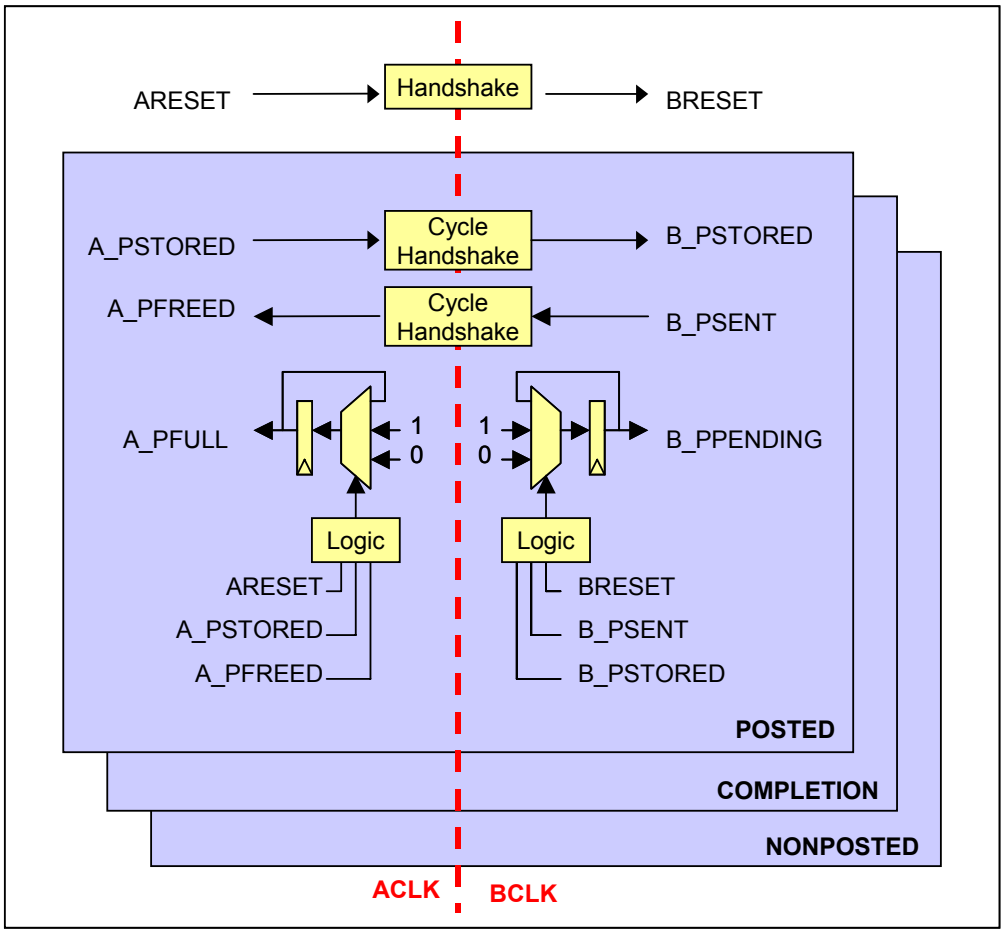


Figure 4.3.3.1 – BUFFERSTATUS Implementation

Status Signal	Description
A_PSTORED, B_PSTORED	Asserted for one cycle after a posted transaction has been completely stored in the posted buffer
A_PFREED, B_PSENT	Asserted for one cycle after a posted transaction has been transmitted
A_PFULL	Asserted when the posted buffer is full
B_PPENDING	Asserted when there is a pending posted transaction

Table 4.3.3 – Posted Buffer Status Signals

The HANDSHAKE module illustrated in Figure 4.3.3.2 transfers data from the TX clock boundary to the RX clock boundary. When TXDATA is first asserted, DATA will stay asserted until the data has crossed the clock boundary, signified by RXDATA being high. Signals that cross the clock boundary are latched twice at the destination to exponentially reduce a chance of metastability. The timing of the signals in the Handshake Implementation can be found in Figure 4.3.3.3.

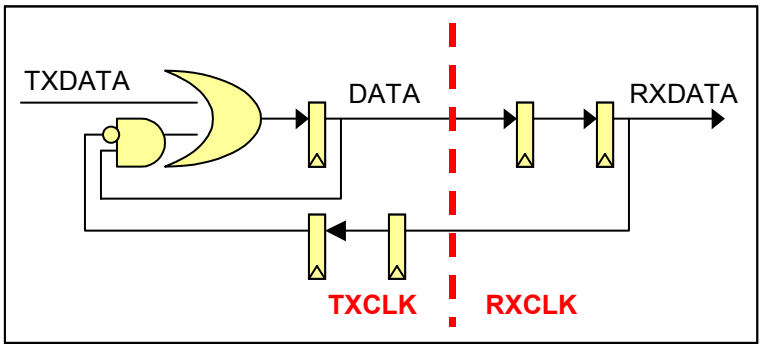


Figure 4.3.3.2 – Handshake Implementation

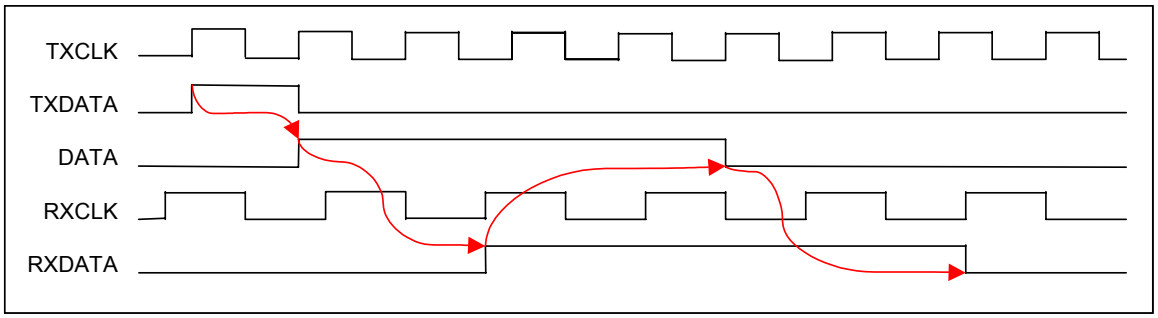


Figure 4.3.3.3 – Handshake Timing Diagram

The CYCLEHANDSHAKE module illustrated in Figure 4.3.3.4 transfers data from the TX clock boundary to the RX clock boundary and turns the data into a pulsed RXDATA that is asserted for one RXCLK cycle.

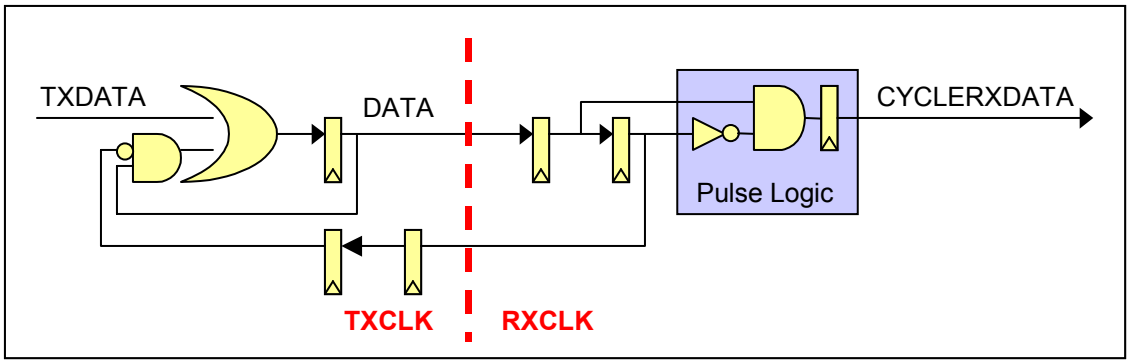


Figure 4.3.3.4 – Cycle Handshake Implementation

4.3.4 BUF Control Logic

The BUF Control Logic takes in signals from the BUF interface and creates control signals for BUFFERSTATUS, HEADERBUFFERS, and DATABUFFERS. Implementing the control logic in the BUF module affords flexibility in that the BUF interface remains the same regardless of whether or not there are changes in the BUF module implementation. For example, in the future the Bridge might need more buffer space to be expanded to support more transactions.

The BUF control logic sets and updates the Write Address, Read Address, Write Enables, and Read Enables for HEADERBUFFERS and DATABUFFERS every time a new transaction and/or 16 bytes of data is written or read. The control logic also prefetches data from the SRAMs due to certain timing constraints when accessing the SRAMs.

4.4 ARB

The ARB Arbiter module tells the BUF module which transaction to transmit next according to the PCI / PCIX / PCI Express Ordering Rules presented in Table 4.4. The columns represent the first transaction received and the rows represent the second transaction received.

Row Pass Column?		Posted Request	Non-Posted Request		Completion	
		Memory Write or Message Request (Col 2)	Read Request (Col 3)	I/O or Configuration Write Request (Col 4)	Read Completion (Col 5)	I/O or Configuration Write Completion (Col 6)
Posted Request	Memory Write or Message Request (Row A)	a) No b) Y/N	Yes	Yes	a) Y/N b) Yes	a) Y/N b) Yes
Non-Posted Request	Read Request (Row B)	No	Y/N	Y/N	Y/N	Y/N
	I/O or Configuration Write Request (Row C)	No	Y/N	Y/N	Y/N	Y/N
Completion	Read Completion (Row D)	a) No b)Y/N	Yes	Yes	a) Y/N b) No	Y/N
	I/O or Configuration Write Completion (Row E)	Y/N	Yes	Yes	Y/N	Y/N

Table 4.4 – PCI Express, PCI, PCIX Ordering Rules

The table entries in Table 4.4 indicate whether or not the second transaction (row) should be able to pass the first transaction (column). A “Yes” entry means that the second transaction must be allowed to pass the first in order to avoid deadlock, a “No” means that the second transaction must never pass the first transaction in order to support the

producer-consumer strong ordering model, and a "Y/N" indicates that it doesn't matter whether or not the second transaction passes the first. The following is an explanation of select entries from Table 4.4:

Row A, Column 2

(a) A Memory Write or Message Request with the Relaxed Ordering Attribute bit clear must not pass any other Memory Write or Message Request

(b) If the Relaxed Ordering Attribute bit is set there are no ordering requirements.

Row A, Columns 5 and 6

(a) In the upstream direction, it does not matter whether or not Memory Writes and Message Requests can pass Completions.

(b) In the downstream direction, Memory Writes and Message Requests must pass Completions to avoid deadlock.

Row D, Column 2

(a) If the Relaxed Ordering Attribute bit is clear, a Read Completion cannot pass a Memory Write or Message Request.

(b) If the Relaxed Ordering Attribute bit is set, a Read completion can pass a Memory Write or Message Request.

Row D, Column 5

(a) Read Completions associated with different Read Requests have no ordering requirements.

(b) Read Completions for one request (same Transaction ID) must return in address order.

4.4.1 Ordering Rules

The original PCI / PCIX / PCI Express Ordering Rules table is pictured in Table 4.4. To reduce the design complexity of the Arbiter, the PCI Express and PCIX Ordering Rules have been simplified in this thesis. The simplified Ordering Rules are presented in Table 4.4.1 and show what is implemented in the PCI Express to PCIX Bridge.

Row Pass Column?	Posted	Completion	Non-Posted
Posted	No	Yes	Yes
Completion	No	No	Yes
Non-Posted	No	No	No

Table 4.4.1 – Simplified PCI Express, PCI, PCIX Ordering Rules

PCI Express Flow Control requires that the Bridge differentiate between posted transactions, completions, and non-posted transactions. If there are multiple pending downstream transactions, posted transactions will have the highest priority, followed by completions, and finally by non-posted requests.

Consider the following downstream path scenario: (1) PCI Express Completion **C** is received, (2) PCIX Master Write attempts to submit **C** but the target issues a retry. Simultaneously, PCI Express Posted Transaction **P** is received. (3) Now there are two pending transactions **P** and **C**. According to the Ordering Rules, **P** must pass **C** in order to prevent deadlock. Therefore, the Arbiter will assert **P** as the next transaction.

4.4.2 Functionality

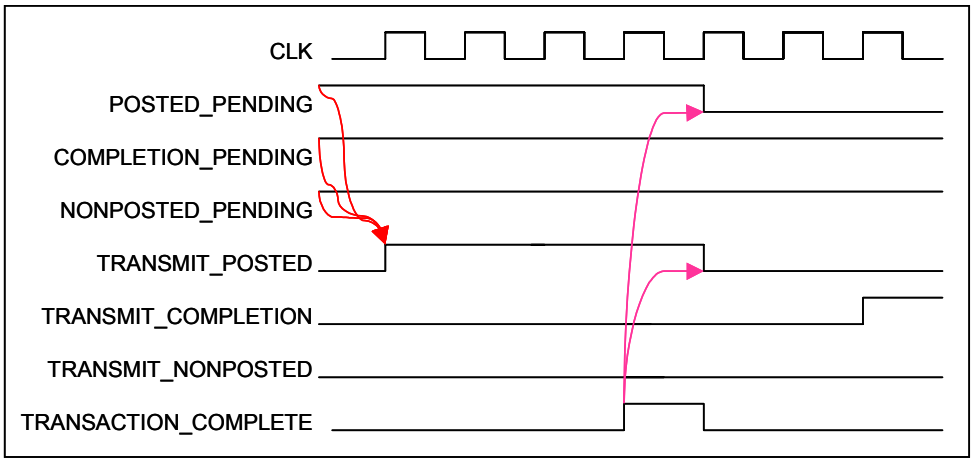


Figure 4.4.2 – ARB Timing Diagram

In the example diagrammed in Figure 4.4.2, there are three pending transactions, indicated by the assertion of POSTED_PENDING, COMPLETION_PENDING, and NONPOSTED_PENDING. The ARB modules signals the highest priority transaction, the posted transaction, should be transmitted next by asserting TRANSMIT_POSTED until a successful transmission is indicated by the assertion of TRANSACTION_COMPLETE.

4.5 MWRITE

The MWRITE module masters a write on PCIX by driving PCIX control signals to start a write, sending the data and data-get signals to/from the PCIX Interface and DOWNBUF module, and informing DOWNBUF and DOWNARB if the transaction was successful or needs to be retried. In Figure 4.5, MWRITE is mastering a burst write.

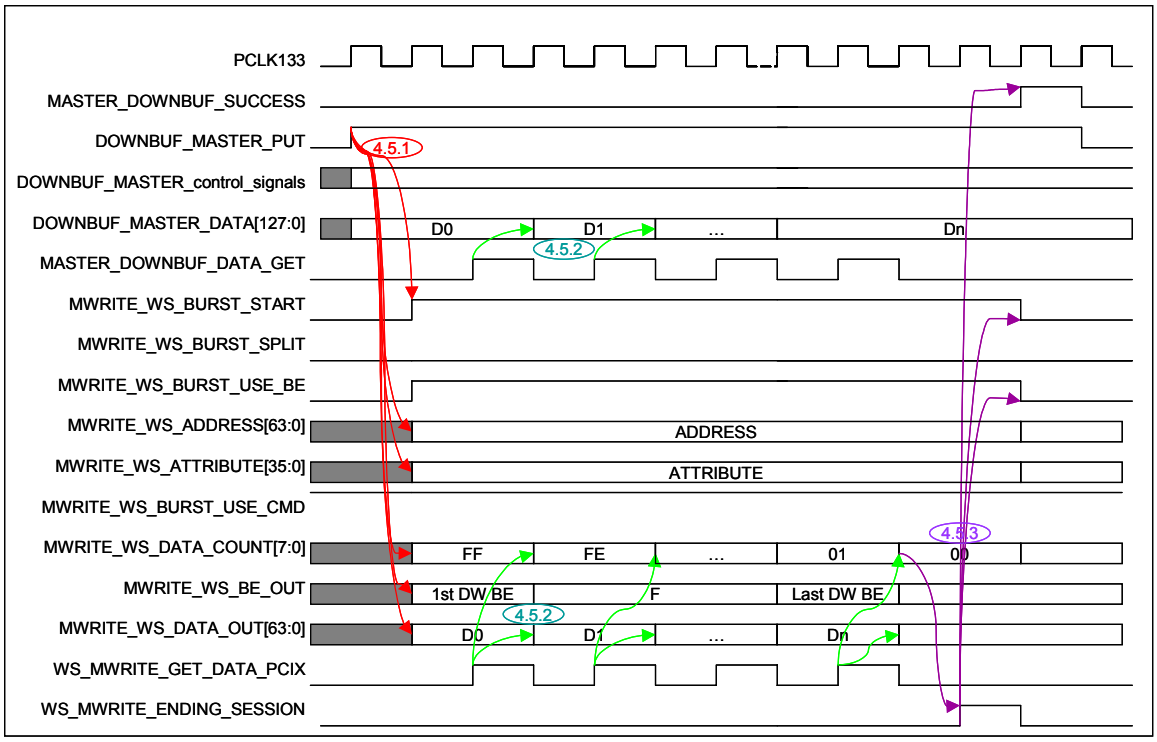


Figure 4.5 – MWRITE Timing Diagram

4.5.1 Initiation of a PCIX Master Write

If DOWNBUF_MASTER_PUT is asserted and DOWNBUF_MASTER_READORWRITE indicates that the transaction is a write, MWRITE will initiate a PCIX write by asserting either MWRITE_WS_BURST_START or MWRITE_WS_DWORD_START based on the value of DOWNBUF_MASTER_BURSTORDWORD. In Figure 4.5, MWRITE is initiating a burst transaction. MWRITE_WS_BURST_START is high and will remain high until the PCIX Write Server (WS) ends the transaction by asserting WS_MWRITE_ENDING_SESSION.

MWRITE will also ensure that all PCIX control signals are valid while MWRITE_WS_BURST_START or MWRITE_WS_DWORD_START is asserted. If the

transaction is a Split Completion, MWRITE will differentiate it from a write by asserting MWRITE_WS_BURST_SPLIT with either MWRITE_WS_BURST_START or MWRITE_WS_DWORD_START.

4.5.2 Transmit Data

In Figure 4.5, MWRITE_WS_DATA_OUT will be valid while MWRITE_WS_BURST_START or MWRITE_WS_DWORD_START is asserted. MWRITE_WS_DATA_OUT will initially contain the first Qword (8 bytes) of data. If WS_MWRITE_GET_DATA_PCIX is high on the rising edge of the PCLK133 clock, then MWRITE_WS_DATA_OUT will immediately be updated to the next Qword and MWRITE_WS_DATA_COUNT will be updated to reflect the number of Qwords left to transmit.

If MWRITE is mastering a burst transaction, it will also transmit the corresponding byte enables for the first and last four bytes. Byte enables for the intermediate Dwords in-between will always be enabled. If MWRITE is mastering a Dword transaction, however, the byte enable will be embedded in the PCIX Attribute MWRITE_WS_ATTRIBUTE.

4.5.3 End Transmission

The PCIX Write Server (WS) ends transmission by asserting WS_MWRITE_ENDING_SESSION for one cycle, causing MWRITE to deassert MWRITE_WS_BURST_START or MWRITE_WS_DWORD_START. MWRITE will

immediately inform DOWNARB that the transaction has finished by asserting MWRITE_DOWNARB_SUCCESS or MWRITE_DOWNARB_RETRY for one cycle.

If WS_MWRITE_RETRY is asserted with WS_MWRITE_ENDING_SESSION, MWRITE tells DOWNBUF that the transaction needs to be retried by asserting MWRITE_DOWNBUF_RETRY for one cycle. However, if WS_MWRITE_RETRY not asserted with WS_MWRITE_ENDING_SESSION, MWRITE tells DOWNBUF that the transaction was successful by asserting MWRITE_DOWNBUF_SUCCESS for one cycle.

4.6 MREAD

The MREAD module initiates a PCIX read and informs the DOWNBUF and DOWNARB modules whether the transaction was successful or needs to be retried. MREAD assumes that it will never receive PCIX immediate read data because the PCIX target will always split the transaction. In Figure 4.6, MREAD is mastering a burst read.

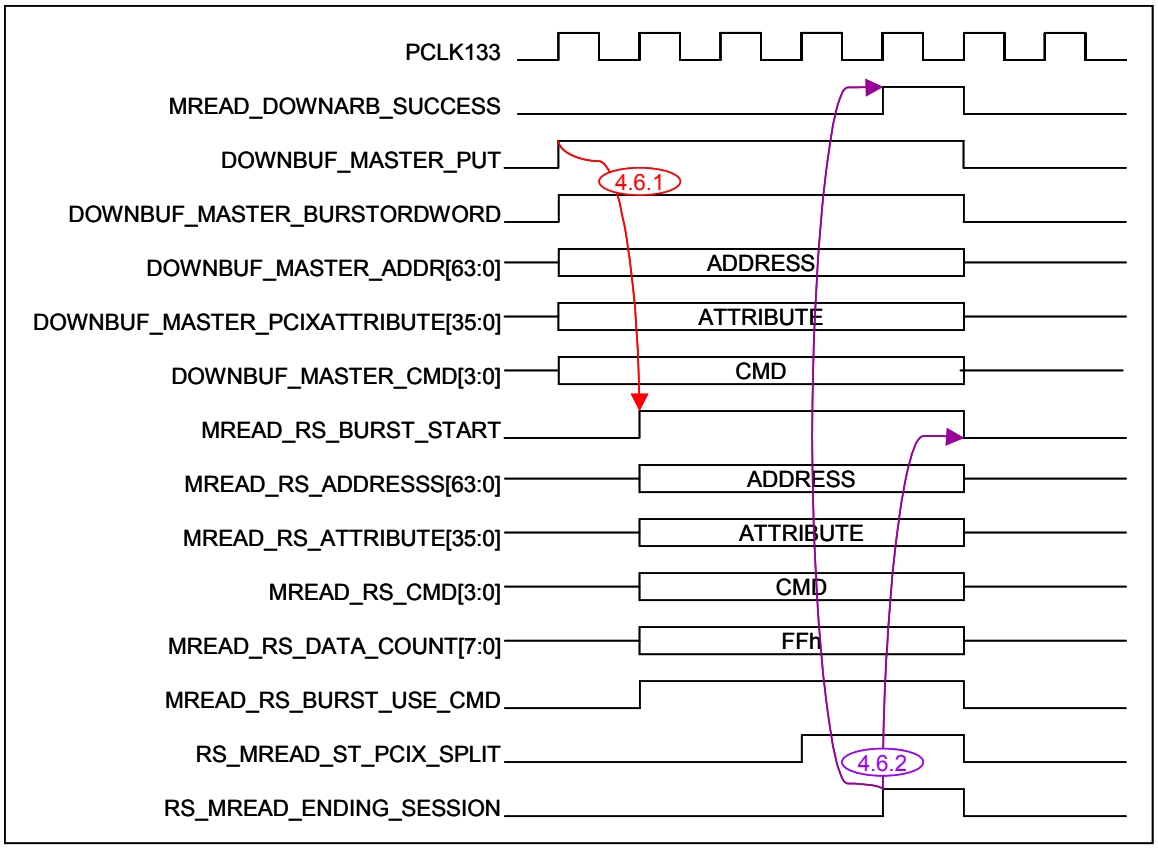


Figure 4.6 – MREAD Timing Diagram

4.6.1 Initiation of a PCIX Master Read

If DOWNBUF_MASTER_PUT is asserted and DOWNBUF_MASTER_READORWRITE indicates that the transaction is a read, MREAD will initiate a read by asserting either MREAD_RS_BURST_START or MREAD_RS_DWORD_START based on the value of DOWNBUF_MASTER_BURSTORDWORD. In Figure 4.6, MREAD is initiating a burst transaction, therefore MREAD_RS_BURST_START will remain high until the PCIX Read Server (RS) ends the transaction by asserting RS_MREAD_ENDING_SESSION. MREAD will ensure that the PCIX control signals are valid while MREAD_RS_BURST_START or MREAD_RS_DWORD_START is asserted.

4.6.2 End Transmission

The PCIX Read Server ends the transaction by asserting RS_MWRITE_ENDING_SESSION for one cycle, causing MREAD to deassert MREAD_RS_BURST_START or MREAD_RS_DWORD_START and tell DOWNARB that the transaction has finished by asserting MREAD_DOWNARB_DONE for one cycle.

If RS_MREAD_RETRY is asserted with RS_MREAD_ENDING_SESSION, MREAD tells DOWNBUF that the transaction needs to be retried by asserting MREAD_DOWNBUF_RETRY for one cycle. However, if RS_MREAD_RETRY is not asserted with RS_MREAD_ENDING_SESSION, MREAD tells DOWNBUF that the transaction was successful by asserting MREAD_DOWNBUF_SUCCESS for one cycle.

4.7 DECODER

Every time there is a new transaction on the PCIX bus, the DECODER module examines the CMD and ADDR to determine if the Bridge should claim the transaction, and which PCIX port should handle the transaction.

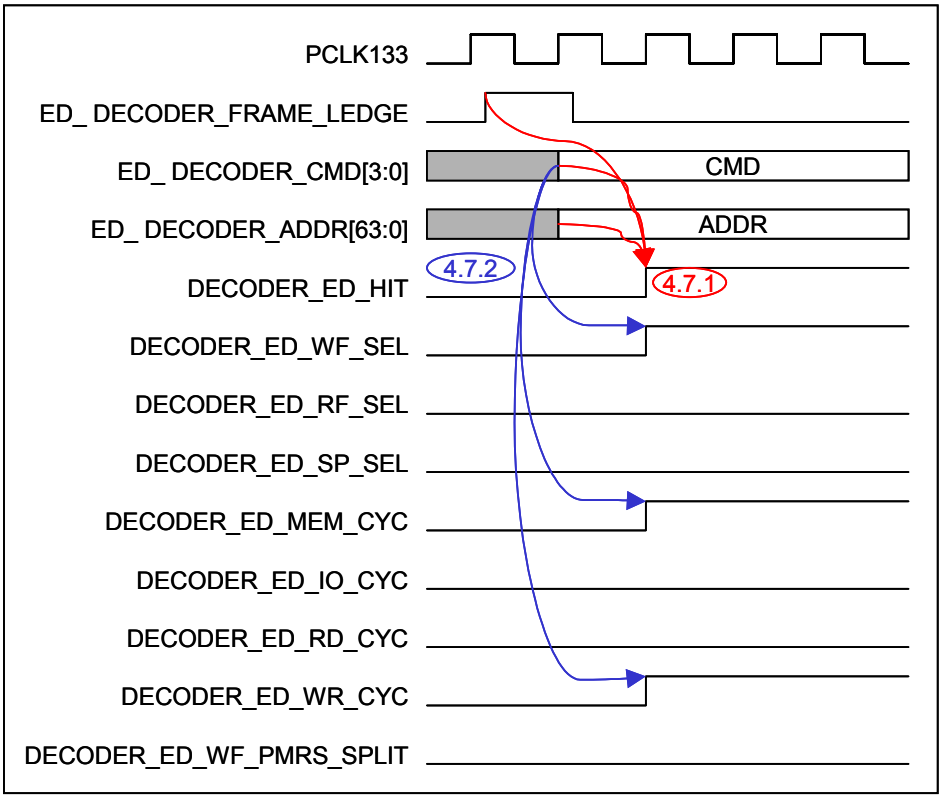


Figure 4.7 – Decoder Timing Diagram

4.7.1 Hit or miss

ED_DECODER_FRAME_LEDGE asserts for one cycle to indicate that the PCIX Interface has received a new PCIX transaction. The DECODER claims the transaction by asserting DECODER_ED_HIT if (1) PS_ED_CMD is an I/O transaction and ED_DECODER_ADDR is

outside of the IO address space designated by IOSTART and IOEND, or (2) ED_DECODER_CMD is a memory transaction and ED_DECODER_ADDR is outside of the Memory address space designated by MEMSTART and MEMEND. Configuration transactions are not considered because they do not travel upstream.

Registers in the Bridge's Configuration space usually define the I/O address space and Memory address space. Configuration space is not supported in this thesis, therefore the address windows for the I/O and Memory Address spaces are hard-wired with IOSTART, IOEND, MEMSTART, and MEMEND. IOSTART indicates the lower bound address of the Bridge's I/O address space whereas IOEND indicates the upper bound address. MEMSTART indicates the lower bound address of the Bridge's Memory address space whereas MEMEND indicates the upper bound address.

4.7.2 Select Port and Transaction Type

The DECODER has three ports to choose from based on the value of ED_DECODER_CMD – the Slave Read (RF), the Slave Write (WF), and the Synchronous Port (SP). On a transaction hit, the DECODER will select a port by asserting DECODER_ED_WF_SEL, DECODER_ED_RF_SEL, or DECODER_ED_SP_SEL.

Based on the value of ED_DECODER_CMD, the DECODER differentiates a memory from an I/O transaction, and a read from a write by asserting DECODER_ED_MEM_CYC or

DECODER_ED_IO_CYC, and DECODER_ED_RD_CYC or DECODER_ED_WR_CYC. The Bridge splits all non-posted transactions by asserting DECODER_ED_WF_PMRS_SPLIT.

4.8 SWRITE

The SWRITE module receives and translates PCIX writes into a PCIX Express Transaction Layer Packet, and sends the Transaction Layer Packet to the UPBUF upstream buffers. In Figure 4.8, SWRITE receives a PCIX write with 8 DWs of data.

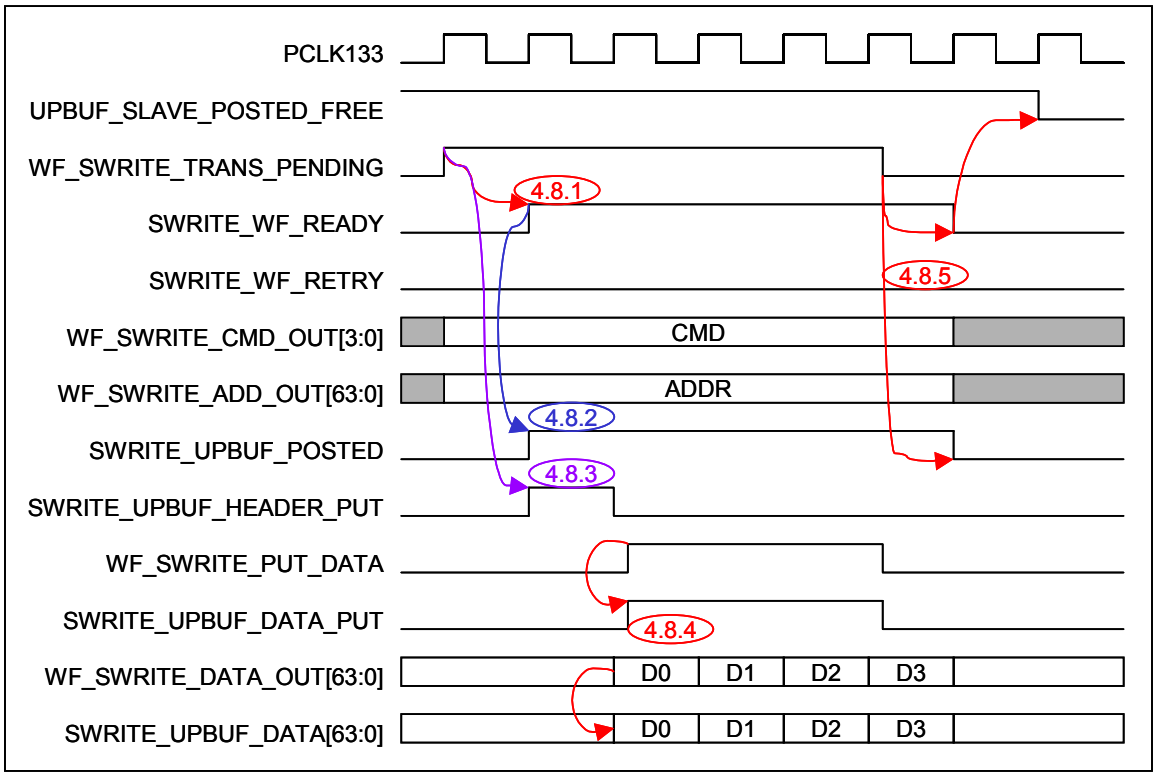


Figure 4.8 – SWRITE Timing Diagram

4.8.1 Reception of a PCIX transaction

Figure 4.8 displays the reception of a pending posted transaction, indicated by the assertion of WF_SWRITE_TRANS_PENDING. Since UPBUF_SLAVE_POSTED_FREE is high to indicate that the posted buffer is free, SWRITE will accept the transaction by asserting SWRITE_WF_READY. However, if the buffer is not free, SWRITE will tell the PCIX Interface to retry the transaction by asserting SWRITE_WF_RETRY.

4.8.2 Buffer Selection

SWRITE will tell UPBUF which buffer to store the transaction in by asserting SWRITE_UPBUF_POSTED, SWRITE_UPBUF_COMPLETION, or SWRITE_UPBUF_NONPOSTED. Once asserted, these signals will remain high until WF_SWRITE_TRANS_PENDING is deasserted.

4.8.3 PCIX / PCI Express Header

When SWRITE accepts the PCIX transaction, it will form a PCI Express header from the PCIX Command, Address, Byte Enables, and Attribute taken from WF_SWRITE_CMD_OUT, WF_SWRITE_ADD_OUT, WF_SWRITE_BE_N_OUT, and PS_ATTRIBUTE_STATE. SWRITE will then drive the PCI Express header on SWRITE_UPBUF_HEADER and assert SWRITE_UPBUF_HEADER_PUT for one cycle.

The PCIX to PCI Express translation is a backwards translation of the PCI Express to PCIX translation outlined in Section 4.2.1. Some PCI Express Header fields do not exist in PCIX, however, and are hard-wired to the values specified in Table 4.8.3.

Traffic Class (TC)	Hardwired to 0
Digest (TD)	Hardwired to 0
Poisoned (EP)	Hardwired to 0
Completion Status Code (Cpl Status)	Hardwired to 0
Byte Count Modified (M)	Hardwired to 0

Table 4.8.3 – Hard-wired PCI Express Fields in Upstream Translation

4.8.4 PCIX / PCI Express Data

SWRITE maps WF_SWRITE_PUT_DATA to SWRITE_UPBUF_DATA_PUT, and WF_SWRITE_DATA_OUT to SWRITE_UPBUF_DATA. A new QWORD is sent on SWRITE_UPBUF_DATA_OUT with every assertion of SWRITE_UPBUF_DATA_PUT.

4.8.5 End of Reception

Transmission ends when WF_SWRITE_TRANS_PENDING deasserts, causing SWRITE to deassert SWRITE_WF_READY. In the example in Figure 4.8, SWRITE will stop writing to the buffer by deasserting SWRITE_UPBUF_POSTED, and signify that the posted buffer is no longer empty by deasserting UPBUF_SLAVE_POSTED_FREE.

4.9 SREAD

The SREAD module receives and translates PCIX reads into a PCIX Express Transaction Layer Packet, and sends the Transaction Layer Packet to the UPBUF upstream buffers.

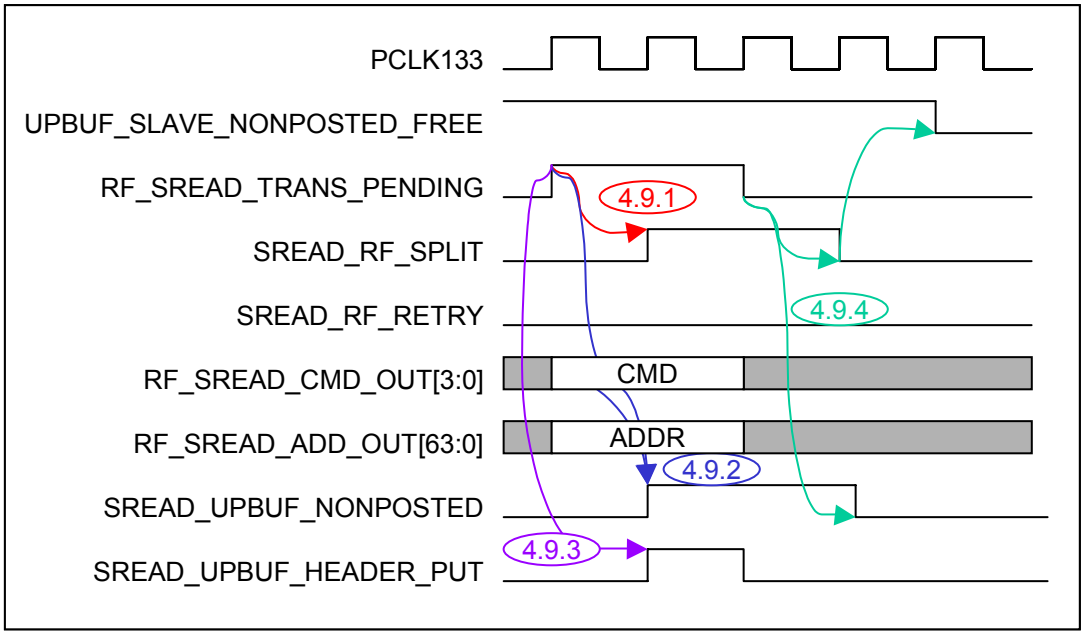


Figure 4.9 – SREAD Timing Diagram

4.9.1 Reception of a PCIX transaction

In Figure 4.9, RF_SREAD_TRANS_PENDING is asserted to indicate a pending read. Since UPBUF_SLAVE_NONPOSTED_FREE is high to indicate that the non-posted buffer is free, SREAD will translate and store the Transaction Layer Packet header and split the transaction by asserting SREAD_RF_SPLIT. However, if the buffer is not free, SREAD will tell the PCIX Interface to retry the transaction by asserting SREAD_RF_RETRY.

4.9.2 Buffer Selection

SREAD must tell UPBUF which buffer to store the transaction by asserting SREAD_UPBUF_POSTED, SREAD_UPBUF_NONPOSTED, or SREAD_UPBUF_COMPLETION until RF_SREAD_TRANS_PENDING falls.

4.9.3 PCIX / PCI Express Header

When SREAD accepts a PCIX transaction, it will form a PCI Express header from RF_SREAD_CMD_OUT and RF_SREAD_ADD_OUT. SREAD will then drive the PCI Express header on SREAD_UPBUF_HEADER and assert SREAD_UPBUF_HEADER_PUT for one cycle.

4.9.4 End of Reception

Transmission ends when RF_SREAD_TRANS_PENDING deasserts, causing SREAD to deassert SREAD_RF_SPLIT. In the example in Figure 4.9, SREAD will stop writing to the buffer by deasserting SREAD_UPBUF_NONPOSTED, and signify that the non-posted buffer is no longer empty by deasserting UPBUF_SLAVE_NONPOSTED_FREE.

4.10 SDWORD

The SDWORD module receives and translates PCIX DWORD transactions into a PCIX Express Transaction Layer Packet, and sends the Transaction Layer Packet to the UPBUF upstream buffers.

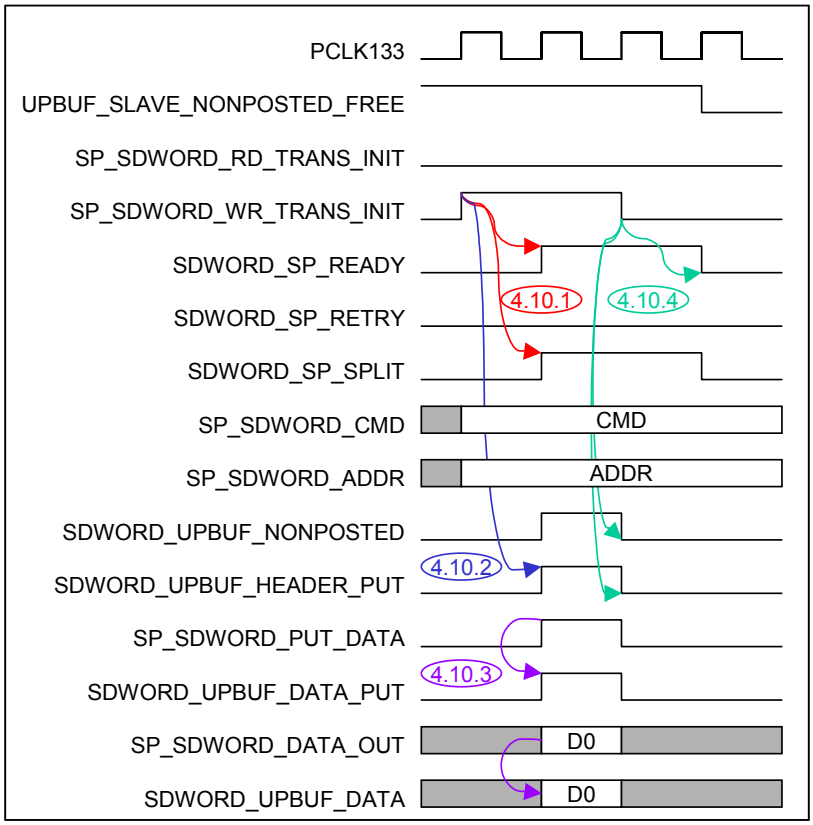


Figure 4.10 – SDWORD Timing Diagram

4.10.1 Reception of a Transaction

Receiving a Write - In Figure 4.10, SP_SDWORD_WR_TRANS_INIT is asserted to signal a new transaction. Since UPBUF_SLAVE_NONPOSTED_FREE is high to indicate that the non-posted buffer is free, SDWORD will store the transaction header in the non-posted

buffer by asserting SDWORD_UPBUF_NONPOSTED and split the transaction by asserting SDWORD_SP_SPLIT with SDWORD_SP_READY. However, if the non-posted buffer is not free, SDWORD will tell the PCIX Interface to retry the transaction by asserting SDWORD_SP_RETRY.

Receiving a Read – Only PCIX IO writes and reads are received by the SDWORD module. Therefore, receiving a read is almost identical to receiving a write except that SP_SDWORD_RD_TRANS_INIT is used in place of SP_SDWORD_WR_TRANS_INIT.

4.10.2 PCIX / PCI Express Header

SDWORD will form a PCI Express header from SP_SDWORD_CMD_OUT and SP_SDWORD_ADDR. SDWORD will then drive the PCI Express header on SDWORD_UPBUF_HEADER and assert SDWORD_UPBUF_HEADER_PUT for one cycle.

4.10.3 PCIX / PCI Express Data

When receiving a DWORD write, SDWORD maps SP_SDWORD_PUT_DATA to SDWORD_UPBUF_DATA_PUT, and SP_SDWORD_DATA_OUT to SDWORD_UPBUF_DATA.

4.10.4 End of Reception

Transmission ends when SP_SDWORD_WR_TRANS_INIT or SP_SDWORD_RD_TRANS_INIT is deasserted, causing SDWORD to deassert SDWORD_SP_READ and SDWORD_SP_SPLIT. In the example in Figure 4.10, SDWORD will stop writing to the buffer by deasserting SREAD_UPBUF_NONPOSTED, and signify that the non-posted buffer is no longer empty by deasserting UPBUF_SLAVE_NONPOSTED_FREE.

4.11 PCIEXTX

The PCIEXTX module transmits PCI Express Transaction Layer Packet s by obtaining a grant from the Transaction Layer Packet Interface, transferring data, and telling UPBUF and UPARB when the transaction is finished. In Figure 4.11, PCIEXTX is transmitting a posted transaction with a 4DW data payload.

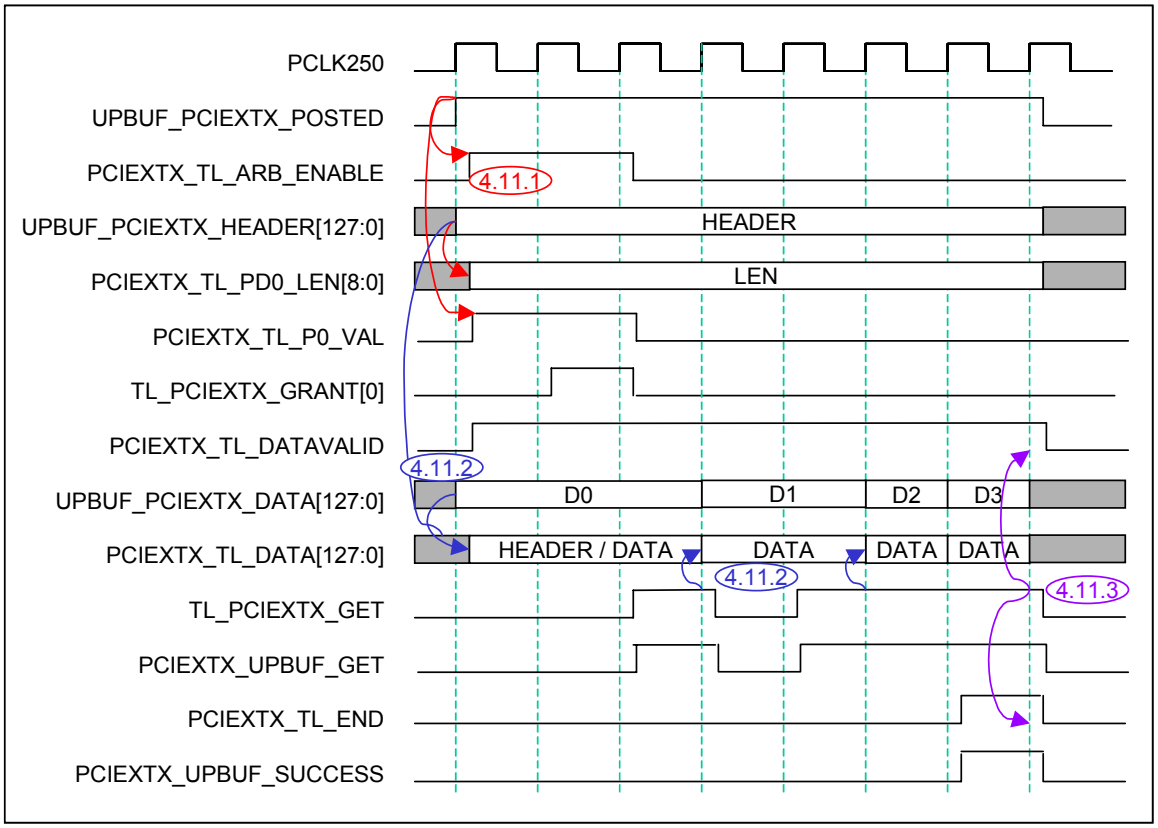


Figure 4.11 – PCIEXTX Timing Diagram

4.13.1 PCI Express Transaction Layer Packet Interface Grant

If UPBUF_PCIEXTX_POSTED and TL_PCIEXTX_TX_READY are both high, PCIEXTX will ask the Transaction Layer Packet Interface to send a posted transaction by asserting PCIEXTX_TL_ARB_ENABLE and PCIEXTX_TL_P0_VAL, and driving the length of the packet in multiples of 16 on PCIEXTX_TL_PD0_LEN. PCIEXTX obtains the grant when TL_PCIEXTX_GRANT[0] is asserted for one cycle, and responds by deasserting PCIEXTX_TL_ARB_ENABLE and PCIEXTX_TL_P0_VAL.

4.11.2 Transmitting PCI Express Header and Data

PCIEXTX_TL_DATAVALID must stay asserted while PCIEXTX is transmitting. The PCI Express header and data are both transferred on PCIEX_TL_DATA. If the header is 4 DW long, then PCIEXTX_TL_DATAVALID will be asserted with the header on PCIEX_TL_DATA. However, if the header is 3 DW long, then PCIEXTX_TL_DATAVALID will be asserted with the header and the first 4 bytes of data.

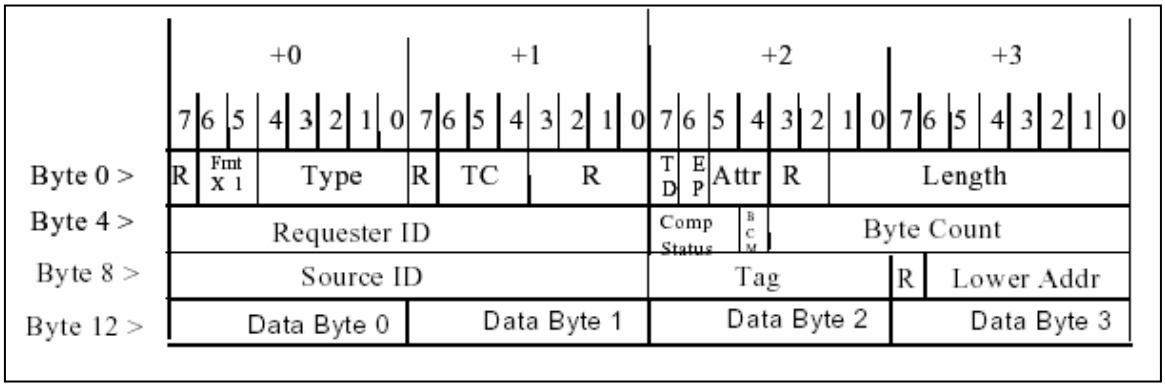


Figure 4.13.2 – Data format for a 3DW PCI Express Header

If TL_PCIEXTX_GET is asserted, PCIEXTX provides the subsequent 16 bytes of data. TL_PCIEXTX_GET is mapped straight to PCIEXTX_UPBUF_GET.

4.11.3 Ending the Transaction

PCIEXTX asserts PCIEXTX_TL_END coincident with the final 16 bytes of data. On the cycle after TL_PCIEXTX_GET is asserted, PCIEXTX will deassert PCIEXTX_TL_DATAVALID and PCIEXTX_TL_END. The PCIEXTX will then inform UPBUF that the transmission was successful by asserting PCIEXTX_UPBUF_SUCCESS for one cycle.

5 Verification

The verification portion of this thesis posed quite a challenge. The Bridge implementation ended up with 1150 input ports and 1061 output ports due to the interfaces of the vendor's PCI Express and PCIX Interfaces. With only one person to create a verification environment from scratch, a hierarchical approach was taken to find errors early since bugs are difficult to find and fix at the system level. Figure 5 outlines the continuous verification efforts taken in this thesis.

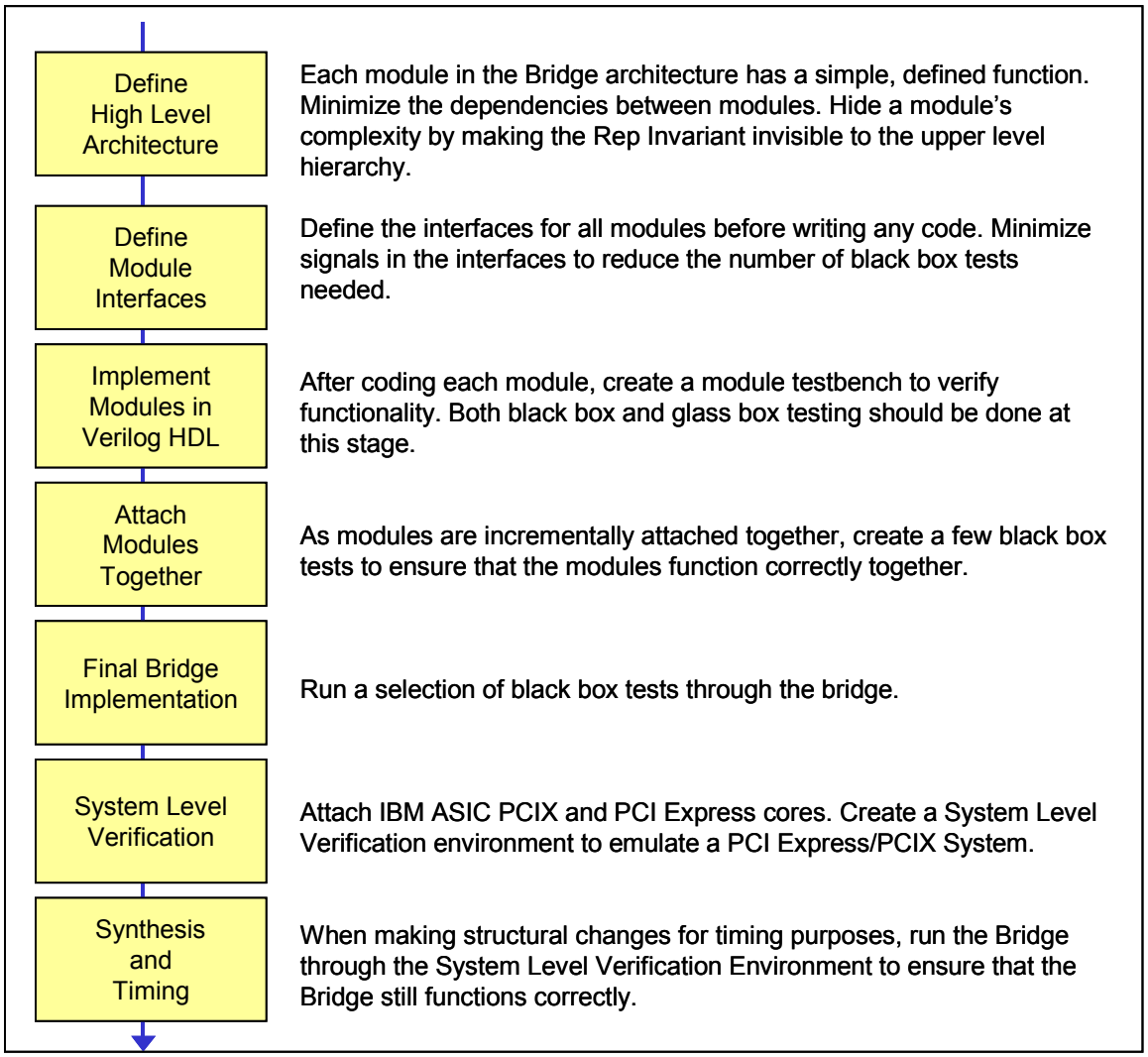


Figure 5 – Verification Development Timeline

5.1 Verification Basics

5.1.1 Types of Tests

Both black box and glass box testing was used in this thesis. A black box test monitors the IOs of the DUT (Device Under Test). For a given set of inputs, a black box test will ensure that the correct output is driven by the DUT. Black box tests were used throughout the thesis.

A glass box test monitors the internal signals as well as the IOs. Glass box tests are more thorough and more time-consuming to write and simulate than black box tests. Therefore, automated glass box tests were only used to verify individual modules. The final Bridge was too complex to automate monitoring the Bridge's many internal signals.

5.1.2 Test Selection - Equivalence Classes and Boundary Cases

Careful consideration must be taken when selecting tests. It is impossible to test every possible thing that can happen, even at the module level. Verification engineers typically have limited time and resources to verify a design, therefore it is desirable to select tests that will maximize the chance of finding a bug.

Similar tests can be grouped into equivalence classes. When testing a device, one typically runs a few tests from each equivalence class. Consider a scenario where the

DUT (Device Under Test) is a module that determines whether or not a triangle is equilateral by taking in three integer measurements of the angles, and output a true or a false. Some equivalence classes for testing such a module might include Equilateral, Not Equilateral, Not a Triangle, etc. Once the equivalence classes have been identified, our next step is to determine which test(s) should represent an equivalence class.

Tests that contain the boundary cases of an equivalence class are the most likely to find a bug. Referring back to the triangle example, some boundary cases for the Not Equilateral equivalence class might include a triangle that is almost equilateral (i.e. 60^{o}, 59^{o}, 61^{o}), and a triangle with angles that border around the valid ranges of inputs and outputs (i.e. 178^{o}, 1^{o}, 1^{o}).

5.2 Module Tests

Glass box and black box module level tests are implemented with Verilog drivers and examined through waveform simulation. The correct functionality of the module is documented in the test file.

Table 5.2 - Module-level Tests

Module	Equivalence Class	Tests
PCIEXRX	Memory Read	32-bit Address 64-bit Address 0 DW requested (min size) 1 DW requested (min boundary) 4 DW requested (PCI Express Transaction Layer Packet Interface boundary) 5 DW requested (PCI Express Transaction Layer Packet Interface boundary) 1024 DW requested (max size)
	Memory Write	32-bit Address 64-bit Address 0 DW written (min size) 1 DW written (min boundary) 4 DW written (PCI Express Transaction Layer Packet Interface boundary) 5 DW written (PCI Express Transaction Layer Packet Interface boundary) 1024 DW written (max size)
	Configuration Read	Configuration Type 1 Read
	Configuration Write	Configuration Type 1 Write
	IO Read	IO Read – 1 DW requested
	IO Write	IO Write – 1 DW written
	Completion with Data	1 DW sent (min size) 4 DW sent (PCI Express Transaction Layer Packet Interface boundary) 5 DW sent (PCI Express Transaction Layer Packet Interface boundary)
	Completion without Data	Completion without Data
	Flow Control	Posted Buffer freed Non-posted Buffer freed Completion Buffer freed
BUF	Write to and Read from Header Buffer	Non-posted – 3 DW Header Non-posted – 4 DW Header

	Write to and Read from Data Buffer	Posted – 1 DW Data (min size) Posted – 3 DW Data (PCIX Interface boundary) Posted – 4 DW Data (PCI Express Transaction Layer Packet Interface boundary) Posted – 5 DW Data (PCI Express Transaction Layer Packet Interface boundary) Completion - 1024 DW data (max data allowed)
ARB	Single Pending Transactions	Posted Completion Non-posted
	Multiple Pending Transactions	2 pending transactions 3 pending transactions 3 pending transactions + new ones coming in as old ones are being transmitted
MWRITE	Successful Burst Write	64-bit Address 32-bit Address 2 DW Data (PCIX Interface boundary) 4DW Data (PCIX Interface boundary) 512 DW Data (PCIX Interface boundary) > 512 DW Data (PCIX Interface boundary)
	Successful Dword Write	Dword Write
	Successful Split Completion	Split Completion with 512 DW of Data
	Retry	PCIX issues a retry
MREAD	Successful Burst Read	64-bit Address 32-bit Address
	Successful Dword Read	32-bit DW Read
	Retry	Retry
DECODER	Target != Bridge	Memory Address Space IO Address Space Split Completion
	Target = Bridge and there is a free buffer	Memory Read or Write IO Read or Write Split Completion
	Target = Bridge and there is no free buffer	Memory Read or Write IO Read or Write Split Completion
SWRITE	Memory Write	2 DW Data (PCIX Interface boundary) 4DW Data (PCIX Interface boundary) 512 DW Data (PCIX Interface boundary) > 512 DW Data (PCIX Interface boundary)
	Split Completion	2 DW Data (PCIX Interface boundary) 4DW Data (PCIX Interface boundary)
SREAD	Memory Read	Memory Read
SDWORD	IO	IO Read IO Write

PCIEXTX	Posted	3DW Header 4DW Header 0 DW Data (min size) 1 DW Data (PCI Express Transaction Layer Packet Interface boundary) 2 DW Data (PCI Express Transaction Layer Packet Interface boundary) 1024 DW data (max size)
	Non-Posted	4DW header 3DW header 0 DW data 1 DW data
	Completion with Data	1 DW data (PCI Express Transaction Layer Packet Interface boundary) 2 DW data (PCI Express Transaction Layer Packet Interface boundary)
	Completion without Data	0 DW data

5.3 Bridge Architecture Tests

After all the modules in the architecture were implemented and tested at the module level, modules were incrementally combined and black box tested. The final result was two separate blocks, a downstream block translating from PCI Express to PCIX, and an upstream block translating from PCIX to PCI Express. Black box Bridge Architecture tests are written as Verilog drivers. The correct functionality of the Downstream and Upstream modules is documented in the test files.

Module	Equivalence Class	Tests
DOWNSTREAM	Memory Read	32-bit Address 64-bit Address
	Memory Write	32-bit Address 64-bit Address 4 DW data 5 DW data
	Configuration Read	Configuration Type 1 Read
	Configuration Write	Configuration Type 1 Write
	IO Read	IO Read
	IO Write	IO Write
	Completion	0 DW data 1024 DW data
UPSTREAM	Memory Read	32-bit Address
	Memory Write	1 DW data 2 DW data 12 DW data
	Completion	1 DW data 2 DW data 12 DW data

Table 5.3 - Bridge Architecture Tests

5.4 System Level Verification

After the Bridge Architecture had been verified, a verification environment was created to test the Bridge at the system level. The complete verification environment, pictured in Figure 5.4, emulates how the bridge will be utilized in a PCI Express / PCIX System.

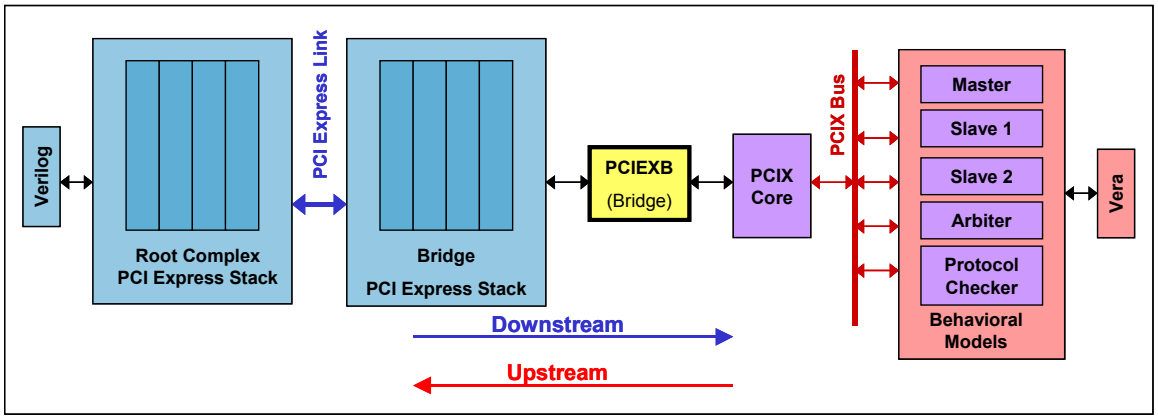


Figure 5.4 – The Complete System Level Verification Environment

On the PCI Express (upstream) side of the Bridge, the environment contains a Verilog driver that drives and checks the PCI Express Transaction Layer interface of the Root Complex PCI Express Stack. The Root Complex PCI Express Stack is connected to the Bridge PCI Express Stack through a PCI Express Link. The Bridge PCI Express Stack also interfaces to the Bridge through the Transaction Layer interfaces.

On the PCIX (downstream) side of the bridge, the environment contains a PCIX core that interfaces the Bridge to the PCI/PCIX bus. There are two PCIX IO slots on the PCIX bus represented by the Master, Slave 1, and Slave 2 models. A Vera module tells the Master to initiate transactions and monitors the state of both Slaves.

Building the System Level Verification Environment took approximately five weeks. It was built incrementally in four stages: (1) PCIX core, (2) PCIX Models and Vera Environment, (3) PCI Express Core Stack, and (4) PCI Express Verilog Driver and Monitor.

5.4.1 The PCIX Core

Piecing the PCIX core together and combining it with the Bridge was the first stage in creating the System Level Verification Environment. The PCIX core consists of three components: (1) the PCIX_O containing the logic to interface to the PCIX bus, (2) the tri containing many tri-state buffers that interface to the bi-directional PCIX bus signals, and (3) the pull-ups containing a few weak pull-ups that drive the bus high when no device is active on the bus.

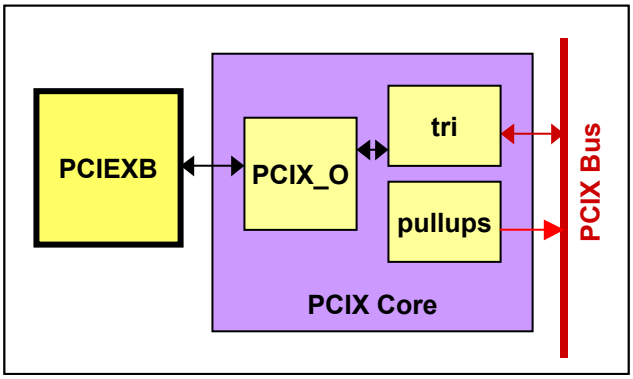


Figure 5.4.1 – Stage One of the System Level Verification Environment Development

After connecting the Bridge to the PCIX core, a piece of Verilog was developed to drive downstream transactions through the Bridge, through the PCIX core, and onto the PCIX bus where correct behavior of the PCIX bus was verified using a protocol checker.

5.4.2 The PCIX Models and Vera Environment

Stage 2 involved building and attaching a PCIX Vera environment to the PCIX bus. The existence of massive and unfamiliar files in the PCIX verification environment increased the difficulty of Stage Two.

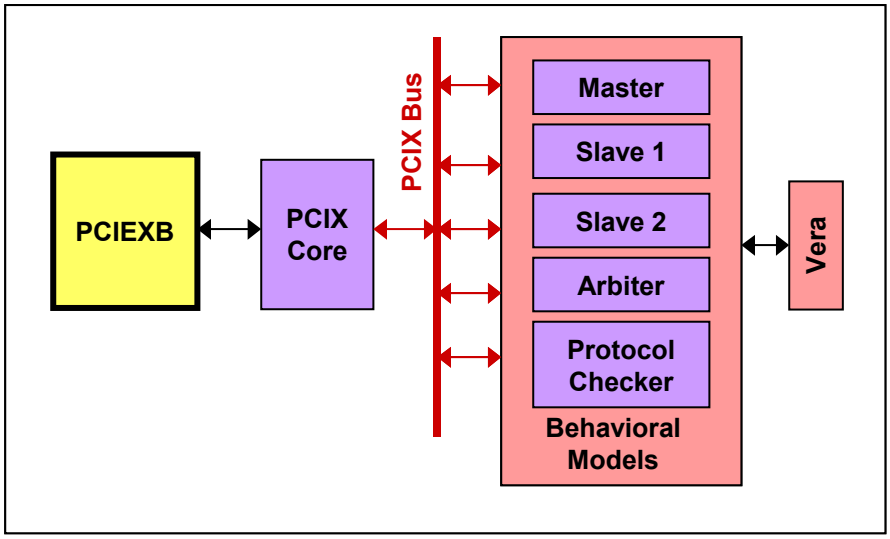


Figure 5.4.2 – Stage Two of the System Level Verification Environment Development

There are three behavior models on the PCIX bus – a Master and two Slaves. These three models are controlled and monitored by a piece of Vera code. The Vera can tell the Master to initiate transactions of various types and sizes. The Vera can also configure, set, and monitor the address spaces of both slaves to ensure that writes are successfully

received. When Slave 1 or Slave 2 receives a read, they automatically issue the Split Completion without consulting the Vera. An independent Behavioral Protocol Checker and PCIX Arbiter ensure that the PCIX bus protocol is correctly driven on the bus by only one device at a time.

5.4.3 Building the PCI Express Stack

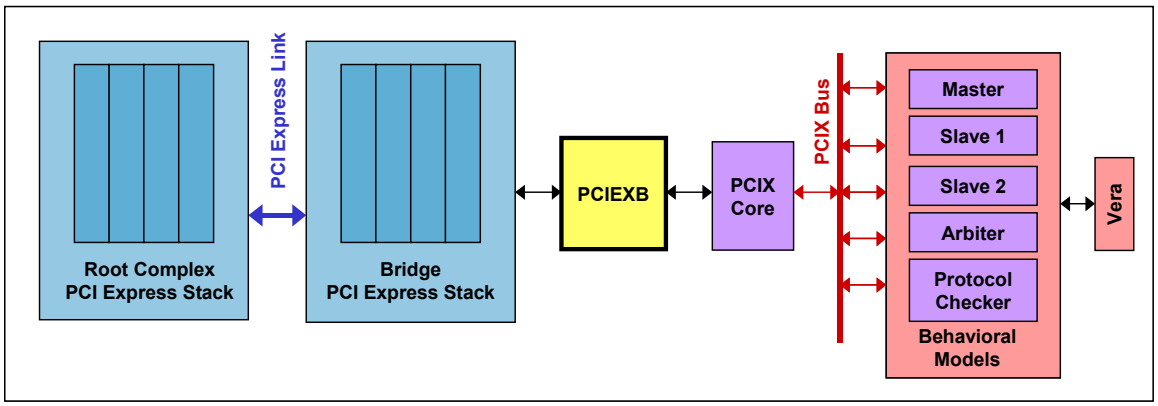


Figure 5.4.3 – Stage Three of the System Level Verification Environment Development

A custom PCI Express Stack was not available that met the needs for this verification environment. Therefore multiple PCI Express cores were assembled into a PCI Express Stack from scratch. The RTL directories for each component needed for the PCI Express Stack was obtained. *Almost* all the IOs of the components matched up directly. Unfortunately, this stage required extra time and work due to the few IOs that did not match up, the ongoing development of the PCI Express cores, and unfamiliarity with the internal interfaces.

Two copies of the PCI Express Stack are used in the verification environment. One Stack is located upstream to the PCI Express link and is configured as a Root Complex. The

other stack is located downstream of the PCI Express link and configured as a bridge port.

5.4.4 The PCI Express Verilog Driver and Monitor

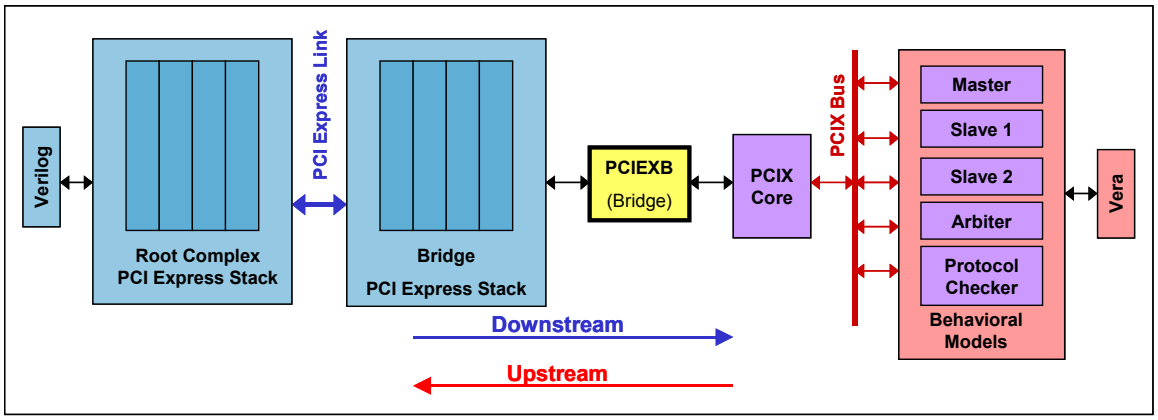


Figure 5.4.4 – Stage Four of the System Level Verification Environment Development

Creating a piece of code to interface to the Root Complex PCI Express Stack was the final step taken in the development of the verification environment. The original plan was to incorporate existing Vera from an existing PCI Express Verification Environment. Due to time constraints and differences between the PCI Express and PCIX Vera environments, the idea of combining the two environments was determined to be too risky. Instead, a piece of Verilog code was developed to stimulate and observe the Root Complex PCI Express Stack.

To minimize the complications caused by the multiple clock domains, verification was limited to one active transaction at a time. Therefore, the PCI Express Verilog driver waits until it has successfully received all planned upstream transactions before it sends any downstream transactions.

5.4.5 System Level Tests

The PCIX Vera and PCI Express Verilog modules initiate the transactions listed in Table 5.4.5. The list is in order of execution.

Transaction	Indication that Transaction is successful
(1) Upstream Memory Write	Received by the RC PCI Express Stack Transaction Layer Packet Interface
(2) Upstream Memory Read	Received by the RC PCI Express Stack Transaction Layer Packet Interface
(3) Downstream Completion	Monitor the PCIX bus with a waveform viewer
(4) Downstream Memory Write	Check the Memory space of PCIX Slave 1
(5) Downstream Memory Read	An (6) Upstream Completion is received by the RC PCI Express Stack Transaction Layer Packet Interface
(6) Upstream Completion	Received by the RC PCI Express Stack Transaction Layer Packet Interface
(7) Downstream IO Write	Check the IO space of PCIX Slave 1
(8) Downstream Configuration Write	Monitor the PCIX bus with a waveform viewer

Table 5.4.5 – System Level Tests

6 Synthesis

6.1 Specifications

A Design Compiler was used to synthesize the PCIEXB Verilog code into a gate-level netlist targeting a current IBM ASIC technology. The Bridge was synthesized with voltages and temperatures specified by IBM Methodology.

Synthesizing the Bridge involved making TCL scripts that were read by the Design Compiler. The scripts set up the wire load model, ideal networks on the two clocks, a maximum fanout of 20, and timing constraints on the IOs. The scripts also analyzed, formatted, compiled, uniquified, and flattened all of the Bridge RTL from the lowest level module and moving up in hierarchy. The scripts instruct the Design Compiler to optimize for area and then issue a timing report. The timing report presents the results of static timing analysis by the Design Compiler and indicates the success or failure of synthesis against the given timing assertions. A Verilog netlist representation of the Bridge is the final result of the synthesis.

6.2 Tools

Some difficulty was encountered when using the Design Compiler. Figure 6.2.1 shows the hardware that was represented by the RTL. A signal is outputted from the BUF module and inputted to both the MWRITE and the MREAD. The MWRITE and the

MREAD rename the signal and send it their corresponding PCIX port. In essence, wire a connection.

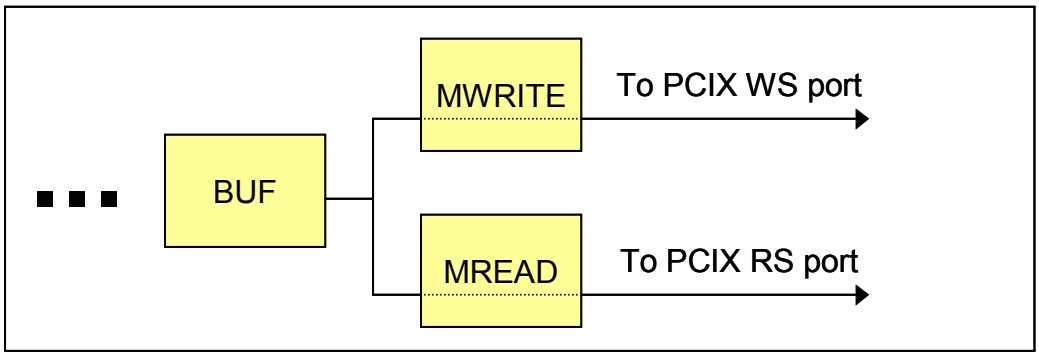


Figure 6.2.1 – Architecture Represented by the RTL

Due to a bug in the tool, the netlist that resulted from synthesis had a combinational feedback loop as pictured in Figure 6.2.2. This appeared to be due to incrementally mapping and compiling the modules in order of hierarchy. Unfortunately, this defect in the tool forced a change in the synthesis TCL script to flatten the entire hierarchy and synthesize the entire Bridge in a single compile. The optimized area reported in Section 8 of this thesis could have been further optimized if it were not for this bug in the tool.

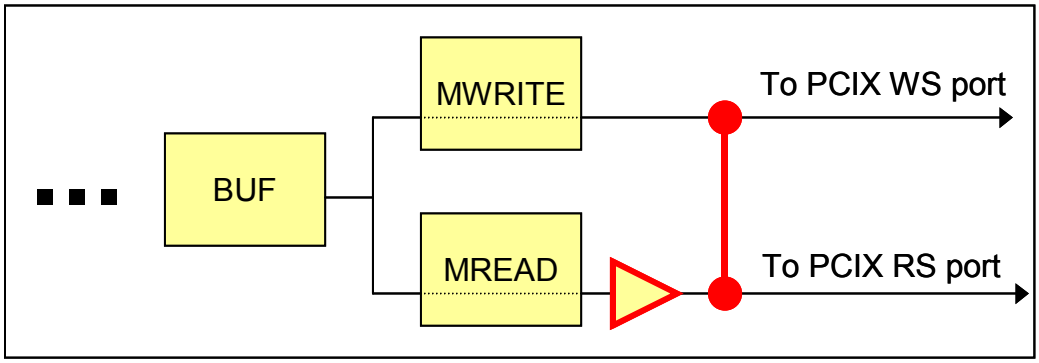


Figure 6.2.2 –Synthesis Result

6.3 Techniques to Avoid Synthesis Headaches

Many inexperienced engineers often run into problems with synthesis during their first design project. Knowing this is the case, efforts were made to fix these mistakes early on in the development process.

Synthesis Headache Prevention Tip #1 – Coding Style

Coding style was kept simple and consistent. All modules were organized, formatted, and documented in the same fashion. Combinational logic was coded in assign statements rather than in an always block for multiple reasons. First of all, it prevented any inferred latches in my code. Second of all, using assign statements is less verbose than an always block and results in code that is easier to read and is therefore easier to debug. Verilog always blocks were only used to instantiate latches.

Synthesis Headache Prevention Tip #2 – Synthesize Early and Often

Synthesis was utilized after completing the first module – PCIEXRX. This fixed any undesirable coding habits that would result in un-synthesize-able code and prevented the same mistakes from being made on the rest of the modules.

7 Static Timing Analysis

7.1 Background

Section 7.1 of this thesis is an excerpt from Section 8.1 of *Hardware Implementation of a Low-Power Two-Dimensional Discrete Cosine Transform* by Rajul Shah.

Static timing analysis, performed by the IBM static timing tool, ensured that the designed hardware functioned properly with the timing and electrical constraints of the system. Four different path types were analyzed by the timing tool: primary inputs to primary outputs, primary inputs to a register, register to register, and register to primary outputs. For each of these path types, the tool checked that data arrived at its destination in time (setup time) and that it stayed steady for the required time (hold time). This was determined by slack measurements, or relations between the Required Arrival Times (RAT) and the Actual Arrival Time (AT). Both the RAT and AT values differ for early mode and late mode tests. Negative slacks indicated static timing failures, while positive slacks indicated the hardware would function properly for that path. Of course the results of these tests were dependent on the assertions provided.

The setup tests were checked in late mode or long path analysis. The slack, in late mode analysis, is calculated as RAT-AT. In this mode, the latest arrival times are propagated to find the longest path delays. If this slowest path is too long, then the AT will be larger than the required time of arrival. In this case, data may not reach its destination in time, thereby inhibiting the hardware from running at the specified clock frequency.

Early mode, or short, fast path analysis, identified hold time violations. Slack times, in early mode analysis, were equal to AT-RAT. The AT was calculated by propagating the earliest cumulative arrival times for a path. In a fast path, the new signal may arrive too quickly, or before the RAT. The RAT for the early mode case is earliest time that a signal can change after the clock edge. These problematic paths create negative slack time and could cause incorrect hardware operation. In these cases, a race condition could occur, where data would be stored from the next clock cycle rather than from the current clock cycle.

The static timing tool also conducted electrical violation tests. For each element instantiated from the standard library, the tool compared its minimum and maximum specified load capacitances with its load capacitance in the design. The tool did the same comparisons for minimum and maximum slew values as well.

7.2 Assertions

A TCL script was created containing all timing assertions and constraints to be applied to the Bridge. The values were dependent on the timing constraints of the PCI Express Cores provided by Peter Jenkins and Scott Vento, as well as the PCIX core in PCIX only mode provided by Louis Stermole. These requirements are made to reduce timing violations in a System On Chip environment.

The clock jitter was set to a conservative default value of 0.4 ns. There are two clocks in the Bridge: PCLK250 running at 250 MHz to correspond to the PCI Express Transaction Layer Packet Interface, and PCLK133 running at speeds of up to 133 MHz to correspond to the PCIX clock frequency capabilities. The clock transitions were set to 0.3 ns. Any signals that cross the clock boundary were specified as false paths since the timing paths on those signals will change with the variance of the two clocks.

The delay for all Bridge inputs was specified in the TCL script to indicate how long it will take for that input to be valid after that domain's rising clock edge. For the PCLK250 inputs from the PCI Express interface, input delays ranged between 0.6ns to 2.0ns and input transitions were set to 0.7 ns. For the PCLK133 inputs from the PCIX interface, input delays ranged between 0.75ns to 6.70 ns and input transitions were set to 0.1 ns.

The maximum delay for all outputs was also specified in the TCL script. The PCIEXB must meet these timing requirements to ensure that customer hardware will be able to meet their timing requirements. For the PCLK250 outputs to the PCI Express interface, maximums output delays ranged from 0.6ns to 2ns. For the PCLK133 outputs to the PCIX interface, maximum output delays ranged from 2ns to 5 ns. For the outputs to the SRAM, maximum output delays ranged from 1ns to 2ns.

The maximum capacitances for all inputs were set to a default value of 0.2 pF. The load capacitances on the outputs were set to a default value of 0.3 pF. The maximums fanout was set to a default value of 20. Defining the maximum capacitive load values allows the

IBM timing tool to ensure that the PCI Express and PCIX cores can drive the Bridge inputs, and that the Bridge could drive the gates of the PCI Express and PCIX cores.

7.3 Modifications

Seven modifications were made to the RTL in order to meet static timing. The modifications made were caused by timing failures that can be classified into three categories: (1) a combinational path where the input delay inherently violated the maximum delay constrained on the output, (2) slow combinational logic, and (3) too much load on a wire.

Four modifications to the RTL resulted from category (1) situations where the input delay of a combinational path inherently violated the maximum delay allowed on the output. For example, one of the initial timing violations occurred on a path from a signal driven from the PCIX core (WF_SWRITE_TRANS_PENDING) to the Write Enable port of the SRAM (SWRITE_UPBUF_POSTED). The WF_SWRITE_TRANS_PENDING had a 3.7 ns delay whereas the maximum setup allowed for SWRITE_UPBUF_POSTED was 1 ns. Adding latches allowed the path to pass timing with a cost in area and latency.

Unfortunately, not every category (1) timing violation was correctable by adding latches. The TL_PCIEXTX_GET (2ns delay) to READADDR (maximum 1 ns setup) path required the BUF module to perform an extra stage in the pre-fetch of the SRAM data, causing many changes in the BUF module RTL. When TL_PCIEXTX_GET is high on a PCLK250 clock

cycle, the subsequent 16 bytes of data *must* be driven at the next PCLK250 cycle to meet the specification of the PCI Express Transaction Layer Packet Interface. Every time the Bridge sees that TL_PCIEXTX_GET is high, it must increment the READADDR to get the subsequent 16 bytes of data. TL_PCIEXTX_GET had to be latched in order to meet timing in the TL_PCIEXTX_GET (2ns delay) to READADDR (maximum 1 ns delay) path. This meant that the READADDR would be incremented one cycle *after* the Transaction Layer Packet Interface asserts the TL_PCIEXTX_GET, and that the subsequent 16 bytes of data would come after *two* PCLK250 cycles.

There were two category (2) timing violations where the combinational logic was too slow. These problems were caused by muxes with complex logic on the selectors and solved by either making the muxes smaller by eliminating redundancies, or by replacing the muxes with faster combinational logic.

There was one category (3) timing violation where the signal drove so many devices that the buffering added in synthesis caused the path to violate timing. The SWRITE_UPBUF_DATA_PUT signal is driven by a single latch and inputted to 64 muxes and other combinational logic. Dividing the load by three and using two additional latches to help drive the load solved the problem. This reduced the latency from buffering and allowed timing to be met.

8 Performance

The PCI Express to PCIX Bridge outlined in this thesis is optimized for area and performance. Most efforts were devoted to designing for simplicity in verification, small physical area, and minimal latency for a store and forward architecture. Possible methods to further optimize for performance are outlined in Section 9 of this thesis.

The final performance measurements are summarized in Table 8. This Bridge cannot be compared to other products because currently there are no PCI Express to PCI/PCIX bridges on the market. Latency is measured from the cycle after the last data of the transaction is received to the cycle when the transaction is first transmitted. The downstream latency is measured from when TL_PCIEXRX_HEADER_PUT falls to when MWRITE_WS_*_START is asserted. The upstream latency is measured from when WF_SWRITE_TRANS_PENDING falls to when PCIEXTX_TL_ARB_ENABLE is asserted.

Input Ports	1150
Output Ports	1061
Gates	6127
Nets	7693
Connections	15118
Latch Area	13956 gate count
Combinational Area	13478 gate count
Total Area	27434 gate count
Latency[7] Upstream Downstream	 2 PCLK250 cycles + T_H 1 PCLK133 cycles + T_H

Table 8 – Performance and Area Measurements of the Bridge

[7] T_H approximately equals 20ns – 24ns in simulation with a 4 ns PCLK250 clock cycle and a 10 ns PCLK133 clock cycle

9 Future Work

The sheer size of the functionality described in the *PCI Express Bridge Specification* was too much to handle for one person in a nine-month time frame. Some functionality still needs to be implemented in order to be able to sell this product in industry. The verification is quite thorough and complete on the module level. However, there is some room for improvement in terms of system level verification. Additional tests can be added, especially stress tests. In addition to functionality and verification, more optimizations for performance and power efficiency are suggested.

9.1 Functionality

9.1.1 Key Requirements

The Bridge implemented in this thesis can be expanded to also support PCI as its secondary interface. Additionally, it would be desirable to add Configuration Space to the Bridge so that the Bridge can be configured during system setup.

As seen in Table 9.1.1, certain assumptions were made to simplify the Bridge implementation in areas of special case handling that is specific to PCI Express and PCIX. A marketable PCI Express to PCIX Bridge would need to handle these cases.

Scenario	Appropriate Action	Assumption
The three MSBs of the PCI Express Tag are non-zero	Bridge Takes Ownership of the transaction	The three MSBs of the PCI Express Tags are always zero
Discontinuous byte enable(s)	Bridge turns the transaction into two separate transactions with contiguous byte enables, and takes ownership of both transactions	The byte enables are never discontinuous
A PCIX Memory Read is completed with multiple PCI Express CplD packets.	Bridge incrementally saves data in a buffer until the entire Completion is received, then transmits the corresponding PCIX Split Transaction.	PCIX Memory Reads is completed with *one* PCI Express CplD packet.
Maximum data payload size or read request size for PCIX target is smaller than the Transaction Layer Packet received.	Bridge turns the transaction into multiple transactions that fit the PCIX Target's max sizes. If the transaction is non-posted, the Bridge must also take ownership of each new transaction.	Assume all PCIX Targets allow 4 KB for data payloads and 4 KB for read requests, thus eliminating the need to turn the transaction into multiple transactions.
Address and length combination may not cross a 4KB boundary	Turn the transaction into two transactions that are separated by the 4KB boundary	Address and length combinations do not cross a 4KB boundary
PCIX data is ready upon request	Ensure that there is upstream buffer space for the data before mastering a downstream read. Store Immediate Read Data in the upstream buffer.	Assume PCIX targets will split the read

Table 9.1.1 – Assumptions Made to Simplify the Bridge

9.1.2 Optional Capabilities

Listed below are certain capabilities are not required by the *PCI Express Bridge Specification*, but are desirable to customers.

- PCI Express 32-bit ECRC generation and checking
- Advanced Error Reporting
- Hot Plug Support for the PCI Express primary interface
- Prefetchable memory address range
- VGA Addressing
- PCI Express Message Requests with and without Data Payload
- Expansion ROM
- PCIX Mode or Mode 2 support
- PCIX Device ID Messages

9.2 Optimizations

There are some optimizations that can be done to improve the performance and power efficiency of the PCI Express to PCIX Bridge implemented in this thesis. Two future performance optimizations are explored in this section: cut-through and multiple transactions. Clock gating can also be used to reduce power but is not explored in this thesis because performance and cost is by far the most important factors to PCI Express customers.

9.2.1 Cut-Through to Replace Store and Forward

Cut-through is a technique that is often used in bridges to improve the latency of a single transaction.

The Bridge is currently Store and Forward, meaning that when it receives a transaction on side A, it stores the *entire* transaction before it forwards the transaction to side B. Store and Forward is simple, therefore requiring less logic and taking up less area. If the transactions that typically going through the PCI Express Bridge have little or no data, then Store and Forward is acceptable in performance.

The worst-case transaction has a maximum size data payload of 1024 DW. This transaction in the downstream direction has a 256 PCLK250 cycle wait for the entire

transaction to be stored in the BUF buffers before it can be forwarded, and another 512 PCLK133 cycles to retrieve the transaction from the BUF. This transaction in the upstream direction has a 512 PCLK133 cycle wait for the entire transaction to be stored, and another 256 PCLK250 cycles to retrieve the transaction.

Adding cut-through to this Bridge would greatly improve the worst-case latency. For example, if transaction T is coming in on side A, *and* that transaction has the highest priority out of all pending transactions, *and* side B is currently idle, transaction T can *cut through* from side A to side B and begin transmission on side B before the entire transaction is stored. Cut-through reduces the worst-case downstream transaction latency to 256 PCLK250 cycles and 256 PCLK133 cycles, and the worst-case upstream transaction latency to 512 PCLK133 cycles.

There are two ways to implement cut-through in this Bridge. The first option is a simultaneous write and read from a buffer. This option works well in cases where side A and B run at separate clock frequencies. The second option, proposed in Tsang-Ling Sheu's *ATM LAN interconnections with a cut-through nonblocking switch*, bypasses the buffers with a dedicated cut-through link. This option, however, does not work well for bridges with asynchronous boundaries.

9.2.2 Multiple Transactions

The Bridge would probably fare better in performance if it handled multiple transactions of a single type. In this thesis, the decision was made to handle only one transaction of each type at a time for two reasons: (1) it requires less area, (2) the lack of a customer market requirement regarding the number of transactions supported by the Bridge. Therefore, Bridge architecture was designed to easily facilitate a change to multiple transactions. The following discussion explains how multiple transactions can improve performance and explains how easy it would be to change the Bridge to support multiple transactions.

Currently, the upstream and downstream paths of the Bridge only have one buffer of each transaction type: one posted buffer, one completion buffer, and one non-posted buffer. The Bridge performs well under conditions where there are not a lot of transactions of the same type constantly going through. However, in a scenario where the root complex tries to transmit four consecutive posted transactions T1, T2, T3, and T4, the root complex must wait until T1 is forwarded before it can transmit T2. If the Bridge had *four* posted buffers rather than one, then the root complex would be able to transmit T1, T2, T3, and T4 without delay.

The Bridge architecture was designed in anticipation of expanding the Bridge to support multiple transactions. The BUF module would be the only module to change, all other modules are untouched. The change would involve adding more physical memory and adding a queue for each transaction type. All BUF interfaces remain unchanged and no changes in flow control logic are necessary.

10 References

[1] PCI Express to PCI/PCI-X Bridge Specification Revision 1.0, PCI-SIG, February 2003.

[2] PCI Express Base Specification Revision 1.0a, PCI-SIG, p. 1-118, April 2003.

[3] PCI-X Protocol Addendum to the PCI Local Bus Specification Revision 2.0, PCI-SIG, July 2002.

[4] PCI Local Bus Specification Revision 2.3, PCI-SIG, March 2002.

[5] T. Shanley, D. Anderson, PCI System Architecture, Mindshare Inc, Addison-Wesley, 1999.

[6] J. Ajanovic, C. Jackson, "Scalable System Expansion with PCI Express Bridge Architecture," Intel Developer Forum, February 2003.

[7] Tsang-Ling Sheu, "ATM LAN interconnections with a cut-through nonblocking switch," IEEE Fourteenth Annual International Phoenix Conference, March 1995, p. 578-584.

[8] IBM Microelectronics, ASIC Databook, International Business Machines Corporation.

[9] PCI Express System Architecture, Mindshare, Inc. 2003.

[10] IBM Microelectronics, PCI Express x16 Data Link and Logical Physical, International Business Machines Corporation.

[11] IBM Microelectronics, PCI Express x16 Transaction Layer, International Business Machines Corporation.

[12] PCI/PCIX Model User's Manual (PI 2.3, PCI-X 1.0, PCI-X 2.0), August 2003.

[13] IBM Microelectronics, PCI/PCI-X Bus Interface 32/64 Bits, Revision 2.

[14] Rajul Shah, "Hardware Implementation of a Low-Power Two-Dimensional Discrete Cosine Transform," Section 8.1 Static Timing Analysis – Background, May 2002.

[15] Kaner, Falk, Nguyen, Testing Computer Software, John Wiley & Sons, Inc, New York. 1999. pp. 1-141.

[16] Drerup, Ben, IBM Austin.