Implementation of the StarT-Voyager Bus Interface Units

by

Christopher Joseph Conley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author . . . .
Department of Electrical Engineering and Computer Science
May 27, 1997

Certified by . . . .
G. Andrew Boughton
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Implementation of the StarT-Voyager Bus Interface Units

by

Christopher Joseph Conley

Submitted to the Department of Electrical Engineering and Computer Science
on May 27, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The StarT-Voyager parallel computer is intended to be an experimental machine which will support the use of multiple communications protocols on a single parallel machine. StarT-Voyager provides hardware support for several different types of messages, cache-coherent shared memory and configurable protection domains. This flexibility allows programmers to select the communication method which is most efficient for the task at hand, which is not possible in machines which offer a single message- passing mechanism; it is also useful from an experimental viewpoint, as it will allow exploration into the usefulness of various subsets of these parallel computing support mechanisms.

The Bus Interface Units are an integral part of the StarT-Voyager hardware design. These units respond directly to processor bus transactions, implementing the basic message-passing and shared memory functionality directly to reduce latency while supporting flexible responses through configurable internal state and communication with other components, notably a secondary processor used to control communications protocols. This thesis describes the functionality and implementation of these units.

Thesis Supervisor: G. Andrew Boughton
Title: Principal Research Scientist

Acknowledgments

My sincere thanks:

To Dr. Andy Boughton, without whose assistance, support and motivation this thesis never would have been completed. I couldn't have done it without you.

To Professor Arvind, Mike Erlich, Boon Ang, Derek Chiou and the rest of the StarT-Voyager team, for a tremendous amount of encouragement and assistance throughout the past months.

To my teachers throughout the years, for instilling a love of learning in me that will never die.

To Marilyn Pierce and Peggy Carney, for putting up with me.

To Vanessa, my fellow blonde; Karen, my favorite Bud Light girl; Matt, for Liquid Squid; Laurie, for the earrings; Simon, with 15 minutes to spare; Matthew, even if he does overbid me; Brian, go Blue!; and the rest of the Pack, for keeping me at least vaguely sane for the past three years.

To Gabe, Mike, Mike, Bob and Spiff, for a lifetime of meaningless conversations and mindless fun.

To Cathy, who done good, but never forgot her bro'.

To my parents, for all the nagging, prodding, understanding, encouragement and love over the years.

And to He who makes all things possible.

Thank you all so very much!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes the implementation and verification of bus interface hardware for a scalable parallel machine based on commercial components which supports both message-passing and distributed shared memory communication methodologies. This chapter begins with a summary of previous work in the field of parallel computing. It then discusses the StarT-Voyager project in detail, and explains the motivations for this thesis within the goals of that project. At the end of the chapter, the overall organization of the thesis is summarized to direct readers to particular areas of interest.

1.1 Summary of Past Work

The past few decades have seen a rapid increase in the processing power of a single computer; over the past decade, microprocessors have increased in performance by a factor of almost 2 every year [8]. Even so, the processing power of a single computer chip has not managed to out-distance the requirements of modern software; as processing power has increased, so too has the complexity of computation required by modern algorithms. Today's microprocessors, limited by the current silicon etching technologies, power and cooling concerns and verification issues, are often insufficient to meet the demands of modern users.

Parallel computing, in essence, is an attempt to circumvent these limits on computational power by distributing the work done to several microprocessors within a single "computer." The effectiveness of this method varies, but it is capable of reducing latency for many classes of algorithms ranging from Fast Fourier Transforms to database searches.

From an architectural standpoint, the greatest challenge in the emergence of parallel computing is the development of efficient mechanisms for inter-processor communication. In order to work together on a single problem, processors must be able to communicate with each other. There are two primary mechanisms of inter-processor communication: distributed shared memory (DSM) and message-passing.

Originally, parallel architecture designs focused on one particular communication paradigm. To be certain, "distributed shared memory" machines such as the Stanford DASH [6] and MIT Alewife [4] are capable of emulating message-passing functionality while "message-passing" machines like the Cray T3D [10] can implement shared memory functionality; however, the software overhead required to do so makes program execution considerably less efficient. Hence, while supporting a single type of communication often lowered the design and hardware cost of a parallel machine by a considerable amount, it also served to restrict the classes of problems which the machine could efficiently solve.

Recently, several parallel machines have appeared with direct support for both message-passing and cache-coherent distributed shared memory. The Stanford FLASH project [7] implements communications functionality by replacing the standard memory controller with a programmable microcontroller, MAGIC, which directly handles all intra- and inter-processor functionality; other designs, including StarT- Voyager, utilize a secondary "protocol processor" (along with a smaller amount of custom logic) to provide both forms of inter- processor communication.

1.2 The StarT-Voyager Project

StarT-Voyager is a continuation of the parallel computing research begun in StarT [14] and continued in StarT-NG [3]. StarT and StarT-NG were designed with the

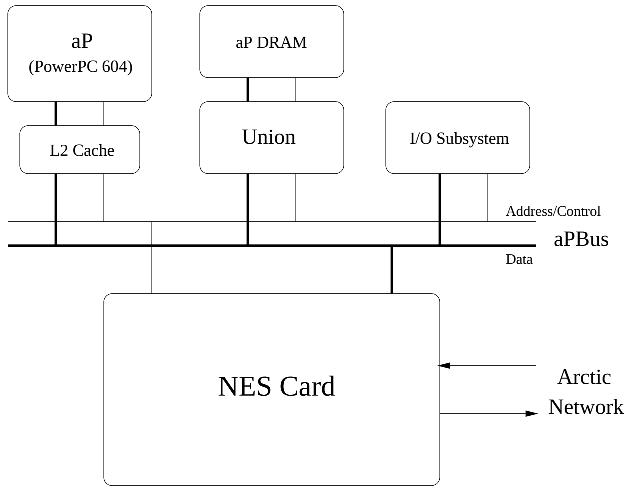

Figure 1-1: A StarT-Voyager site

dataflow programming model in general and the Id language [13] in particular; the Voyager project represents a shift to a standard programming ideology.

The primary goal of the StarT-Voyager project is to design and build a scalable parallel computer based on commercially-available components which is capable of supporting multiple communications protocols. In particular, StarT-Voyager is designed to provide efficient means of sending messages of various sizes [2]; it also provides direct hardware support for cache-coherent shared memory protocols, including (but not limited to) S-COMA [15] and CC-NUMA.

Each site in the StarT-Voyager parallel machine consists of a IBM PS/6000 Model 240 motherboard with two processor slots. One processor slot is occupied by a PowerPC 604 processor with in-line L2 cache; this processor, called the application processor or *aP*, runs a modified version of the AIX operating system with extensions for parallel job coordination. The other processor slot is occupied by the *NES card* (Network Endpoint Subsystem card), which is discussed in detail in section 1.3. The 60X memory bus [12] called the *aPBus* connects the aP, the NES, the Union memory controller (through which aP DRAM is accessed) and the I/O subsystem.

The StarT-Voyager sites are connected through the NES card to the Arctic network, a fat tree network constructed of Arctic router chips [5]. Both the Arctic

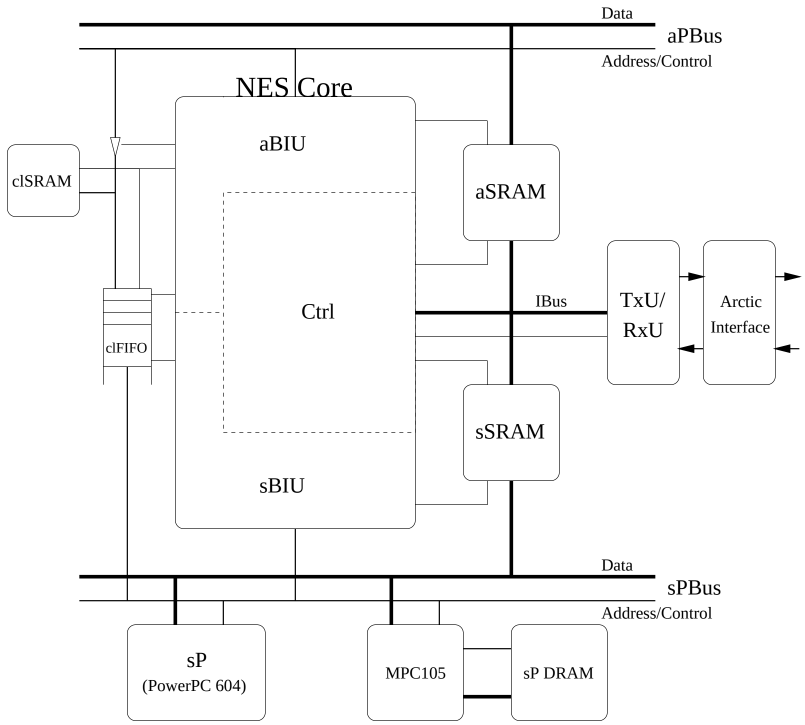

Figure 1-2: The NES card

router chips and the network boards are being designed and implemented as part of the StarT-Voyager project. The Arctic network provides a lossless packet routing system, achieving 160 MBytes/sec bandwidth on each link; as each StarT-Voyager site has two links (one incoming, one outgoing) with the network, the total available network bandwidth for a single site is 320 MBytes/sec. The fat tree topology provides a high bisection bandwidth and redundancy, two highly desirable features for this project.

1.3 NES Overview

The Network Endpoint Subsystem, or NES, is designed to manage the various communications mechanisms provided to the user by means of address space mapping. There are several components of note: the NES Core (or NES chip), the service processor, various SRAMs, and the network interface.

The *NES Core*, a small ASIC, is a custom logic design which provides hardware support for the parallel communications functions required by StarT-Voyager. To meet the goal of efficient communication, the NES Core implements many of the most common functions directly in hardware; in order to provide the flexibility required to implement various communications protocols, the Core is capable of interacting with the service processor to handle complex or unusual operations.

The NES Core is further divided into three sections: the aP Bus Interface Unit (*aBIU*), the sP Bus Interface Unit (*sBIU*) and the NES Ctrl (*Ctrl*). The aBIU and sBIU handle all interaction with the memory busses of the aP and sP; the details of these sections is contained in this thesis. The Ctrl module is responsible for interacting with the *IBus*, an internal 64b data bus which connects the NES Core to the aSRAM, sSRAM and network interface. Ctrl also contains most of the "official" NES state, including the status of all message queues; the BIUs contain their own state and shadowed versions of certain message queue pointers.

The service processor, or *sP*, is a PowerPC 604 which provides flexible and extendible support for the NES Core. The sP observes all aP memory accesses to an address region called *sP Serviced Space*; it is also able to observe and modify all NES Core state, and can initiate transactions on the aP memory bus through the aBIU (see Section 2.2.3). These features are extremely flexible, and will be used to implement coherent shared memory protocols and non-resident message queues. The sP is connected to the NES Core by the 60X bus called the *sPBus*; a MPC105 memory controller [11] also lies on the bus and allows access to the sP DRAM.

The *aSRAM* and *sSRAM* provide storage space for resident message queues of the aP and sP respectively; they are also used to pass data between the two and to read NES Core state. These are dual-ported synchronous SRAMs; one port is attached to the relevant 60X bus and controlled by the BIU, while the other port is attached to the IBus and operated by Ctrl.

The *clSRAM* is a special SRAM used to store cache line status; this allows the NES Core to handle requests for resident global memory in hardware without the overhead of sP interaction.

The network interface, including the *TxU/RxU* and the *Arctic Interface*, is used to provide CRC calculation/checking, compression/expansion and electrical conversions between the IBus and the Arctic network itself.

1.4 Thesis Outline

This chapter has presented a brief background of the history of parallel computing, provided a brief architectural overview of the StarT-Voyager project goals and basic architecture, and described at a very basic level the goals of this research.

Chapter 2 expands on these goals by describing the specific functionality required of the BIUs, including a list of all input and output signals, the support provided for various address spaces and transaction types, the access and use of the SRAMs and the aBIU's ability to serve as master of the aPBus.

Chapters 3 and 4 describe the implementation of the functionality of the BIUs; these includes a description of the functional modules within each BIU, the state machines responsible for each function, the system registers and other internal state, and the timing requirements for interaction with the MPC105 and Union memory controllers and the various SRAMs.

Chapter 5 details the methodology used to verify the functionality of the design; it will indicate the advantages and difficulties found with this methodology, and indicate areas of future concern.

Finally, Chapter 6 concludes the thesis with commentary on the contributions of this research to the author's understanding of parallel architecture. Included in this chapter are future work to be done involving the design described herein and specific insights obtained from this design.

Chapter 2

BIU Functionality

This chapter details the functionality of the BIUs. Although the functions supported by the two BIUs overlap considerably, they will be described separately here in the interest of readability.

Each BIU section begins with the definition of the interfaces between the BIU, the sP or aP Memory Bus, the SRAMs and the remainder of the NES Core. Next, the address space mapping for the Memory Bus and the appropriate responses by the BIU are given. Finally, internal commands issued by other NES Core modules and the expected responses are detailed.

The emphasis of this chapter is on the various functions which are supported, not on the mechanisms used to support them. Chapters 3 and 4 contain the detailed implementation of the sBIU and aBIU respectively.

A summary of the various hardware components is given in Sections 1.2 and 1.3. In addition, the StarT-Voyager Hardware Engineering Specification [1] and the "Message Passing Support on StarT-Voyager" memo [2] provide a higher-level perspective on the Voyager project and may help to put this thesis in context.

2.1 sBIU Functionality

The sBIU is responsible for handling bus transactions issued by the sP (including control of the sSRAM port on the sPBus); it also reacts to a small set of commands

Signal Name	Direction	Size	Comments
sPBusAddress	I	32b	Address
sPBusTransferStart	I	1b	Start of Address Phase
sPBusTransferType	I	5b	Transfer Type
sPBusTransferSize	I	3b	Transfer Size
sPBusTransferBurst	I	1b	Burst Transfer Indicator
sPBusAddressAck	O	1b	Completion of Address Phase
sPBusAddressRetry	O	1b	Address Retry
sPBusDataBusBusy	I	1b	Data Phase in Progress
sPBusTransferAck	O	1b	Completion of Data Phase
sPBusHardReset	I	1b	Hard Reset
sPBusSoftReset	I	1b	Soft Reset
sPBusInterrupt	O	1b	Interrupt sP

Table 2.1: sBIU-sPBus Interface

issued by the CTRL or aBIU modules.

2.1.1 sBIU Interface Groups

The sBIU's interfaces can be broken down into several groups: sPBus Interface, sSRAM Interface, JBus Interface, Ctrl Interface, aBIU Interface and miscellaneous signals.

All signal directions are described with respect to the sBIU, i.e. "I" means an input to the sBIU.

sPBus Interface

The sPBus Interface includes all of the signals relevant to the sBIU's ability to respond to transactions on the sPBus according to 60X bus protocol and the specifications of the MPC105 memory controller. The signals included in this group are listed in Table 2.1.

The sPBusAddress, sPBusTransferType, sPBusTransferSize and sPBusTransfer-Burst signals define the current transfer on the address portion of the sPBus. sP-BusTransferStart, sPBusAddressAck and sPBusAddressRetry provide the boundaries

Signal Name	Direction	Size	Comments
sSRAMAddress	O	12b	sSRAM sP-Port Address
sSRAMRead/Write	O	1b	sSRAM sP-Port Read/_Write
sSRAMOutputEnable	O	1b	sSRAM sP-Port Output Enable
sSRAMChipEnable	O	2b	sSRAM sP-Port Chip Enable

Table 2.2: sBIU-sSRAM Interface

for the address portion of the pipelined transaction, while sPBusDataBusBusy and sPBusTransferAck identify and terminate the data phase of the transfer.

The sPBusHardReset, sPBusSoftReset and sPBusInterrupt signals also fall under this interface; these signals may be used to reset the NES or generate interrupts on certain events (such as a message arrival).

sSRAM Interface

The sSRAM Interface consists of the signals required to control the port of the sSRAM which is connected to the data portion of the sPBus; the other sSRAM port is controlled by NES Ctrl).

The sSRAMAddress signal addresses the SRAM, while sSRAMRead/Write, sSRAMOutputEnable and sSRAMChipEnable serve as control signals.

JBus Interface

The JBus is a 32b data bus which connects the sBIU and Ctrl modules; it serves to allow the JBus to request IBus access in order to transfer data between the aSRAM and sSRAM or write data into one of the SRAMs. The JBus Interface specifies the manner in which these requests are made.

When the JBus is used to write data to the SRAMs, JBusData contains the data to be written and JBusAddress contains the SRAM address of the write. ShTxCompose indicates that the JBus is being used to write the header of a short message (see Section 2.1.2). MemQOut0Compose and MemQOut1Compose further specify that the message being composed is in one of the two special MemQOut queues, with

Signal Name	Direction	Size	Comments
JBusAddress	O	13b	JBus Compose Address
JBusData	O	32b	JBus Data
ShTxCompose	O	1b	32b ShTx Compose Request
MemQOut0Compose	O	1b	MemQOut0 Compose
MemQOut1Compose	O	1b	MemQOut1 Compose
MemQOutOp	O	2b	MemQOut Operation
ComposeFree	I	1b	ShTx Compose Completion
DataMotionValid	O	1b	Data Motion Request
DataMotionFree	I	1b	Data Motion Acknowledge
DataMotionDone	I	1b	Data Motion Complete

Table 2.3: sBIU-JBus Interface

MemQOutOp indicating the particular type of transaction. ComposeFree indicates that the JBus is available and the SRAM write has completed.

For transfers between the aSRAM and sSRAM, known as DataMotion transfers, the JBusData field encodes the aSRAM and sSRAM addresses, the transaction size and the direction of transfer. DataMotionValid indicates that the JBus is being used for a DataMotion command; DataMotionFree acknowledges the reception of the command by NES Ctrl, while DataMotionDone indicates that the transfer is complete.

Ctrl State Interface

The Ctrl State Interface group consists of the SCBus, which is used by the sBIU to access Ctrl state, and the CSBus, which is used by Ctrl to access sBIU state.

The SCBusAddress defines the Ctrl state to be accessed. SCBusOp defines the operation as a read and/or write or no-op. SCBusData provides the 7-bit data field for a state write. SCBusFree indicates that the SCBus is available for a new transaction, while SCBusDone indicates that the previous transaction has been completed.

The CSBusAddress, which is only two bits wide, defines which of the four ShRx (short message receive) producer pointers are being updated by Ctrl. This data is required to allow the sBIU to calculate the full/empty status of the ShRx queues.

Signal Name	Direction	Size	Comments
SCBusAddress	O	11b	Ctrl State Address
SCBusData	O	7b	Ctrl Update Data
SCBusOp	O	2b	SCBus Operation
SCBusFree	I	1b	SCBus Acknowledge
SCBusDone	I	1b	SCBus Operation Done
CSBusAddress	I	2b	sBIU State Address
CSBusData	I	7b	sBIU Update Data
CSBusValid	I	1b	sBIU State Update

Table 2.4: sBIU-Ctrl Interface

Signal Name	Direction	Size	Comments
SABusAddress	O	10b	aBIU State Address
SABusWData	O	7b	aBIU Update Data
SABusRData	I	7b	aBIU Read Data
SABusOp	O	2b	SABus Operation
ASBusAddress	I	10b	sBIU State Address
ASBusWData	I	7b	sBIU Update Data
ASBusRData	O	7b	sBIU Read Data
ASBusOp	I	2b	ASBus Operation

Table 2.5: sBIU-aBIU Interface

CSBusData contains the new pointer value; CSBusValid indicates a valid operation on the bus. (There is no Free or Done signal, as the update is guaranteed to complete in one cycle.)

aBIU Interface

The aBIU interface is used by the sBIU to initiate and respond to state access requests. It consists of the SABus and the ASBus; the SABus carries aBIU state requests from the sBIU, while the ASBus carries sBIU state requests from the aBIU.

The SABus is used by the sBIU to initiate aBIU state reads or writes. SABusAddress indicates the aBIU state to be accessed, SABusOp indicates the transaction type (read, write or no-op), SABusWData provides the data for a write and SABusRData

Signal Name	Direction	Size	Comments
RxEmpty	I	5b	Medium Receive Queue Status
RxLateAck	O	1b	Arctic Acknowledge
NESResetSP	O	1b	Reset Received
CLSLatch	O	1b	Latch clSRAM Update
ResetDMA	O	1b	Reset aBIU DMA Counter
ClearApproval	O	1b	Clear Approval Register

Table 2.6: sBIU Miscellaneous Signals

returns the data from a read.

The ASBus is a mirror image of the SABus which is used by the aBIU to access sBIU state. ASBusAddress indicates the aBIU state to be accessed, ASBusOp indicates the transaction type and ASBusWData and ASBusRData supply/return the write or read data respectively.

Updates or reads indicated on the ASBus or SABus are expected to be processed immediately, so no acknowledgment signals are required.

Miscellaneous Signals

There are a few other I/O signals which do not fall under any of the previous interface groups.

The RxEmpty signal, generated by Ctrl, indicates the status of the medium message receive queues; this information allows the sBIU to implement queue polling.

RxLateAck, output to Ctrl, indicates that the sP has read a message from a queue which requires that a late acknowledgment be sent to the Arctic network. This use of late acknowledgments is controlled by sBIU state (Section 3.3.3). This signal is also used in response to the Arctic Ack Immediate Command, as described in Section 2.1.2.

NESResetSP, output to Ctrl, indicates that a Hard or Soft Reset has been received on the sPBus, or that the NES Reset Immediate Command has been issued.

CLSLatch, output to the external clFIFO, causes the sPBusAddress to be latched into the FIFO. Section 4.5 describes the use of this FIFO in updating the clSRAM.

ResetDMA and ClearApproval, output to aBIU, have specified meanings related

Field	Content	Description
[0:6]	010xxxx	Special ShTx Space
	01100xx	SRAM Space
	011010x	QPtr Space
	0110110	ShTx Space
	0110111	ShRx Space
	01110xx	clSRAM Update Space
	011110x	Immediate Command Space
	011111x	Config Access Space

Table 2.7: sP Address Space

to the appropriate Immediate Commands.

2.1.2 sP Address Space

The sP has access to a full range of address spaces which are used to support direct access to the aSRAM and sSRAM, updates of the clSRAM, access to sBIU, aBIU and CTRL queue state at user or system level and access to message receive and transmit queues.

The address space of the transaction is decoded by bits [0:6] of the sPBusAddress, as shown in Table 2.7. Further decoding of the address depends on the address space. (All unlisted address spaces are ignored.)

The sBIU only supports read-like and write-like transactions issued by the sP; all other transaction types are ignored. Further, as detailed below, the size of the read/write has varying effects on the sBIU's response.

SRAM Space

The simplest functionality supported by the sBIU is sP access to the aSRAM and sSRAM. For sSRAM access, the sBIU simply acts as a 60X bus slave and SRAM controller, providing the appropriate address and control signals to the SRAM while coordinating the transfer on the sPBus. For aSRAM access, the sBIU coordinates a transfer to/from a temporary sSRAM location with a data motion request to NES

Field	Content	Description
[0:4]	01100	SRAM Space
[5:15]	X	** Unused **
[16]	0	sSRAM Access
	1	aSRAM Access
[17:28]	X	SRAM Address
[29]	X	SRAM Chip Select
[30:31]	00	** Fixed Field **

Table 2.8: sP SRAM Space

Ctrl along with a sSRAM read or write.

The sBIU supports direct access to the sSRAM through the control signals described in Section 2.1.1. These accesses are limited to 4B and 8B aligned transfers and 32B burst transfers; burst writes are assumed to be zeroth word first, while reads are returned critical word first.

In the case of an aSRAM write, the provided data is stored in a temporary sSRAM location; a DataMotion request to transfer data from the temporary sSRAM to the specified aSRAM location is then issued through the JBus Interface. An aSRAM read is handled in similar fashion: DataMotion is requested from the specified aSRAM address to a temporary sSRAM location, and the read from this location is executed when the data motion completes. Again, 4B, 8B and 32B transactions are supported with the same restrictions as sSRAM transfers.

QPtr Space

QPtr Space is designed to allow user-level access to short and medium message queue pointers in CTRL. It is intentionally similar to the system level access provided in Config Access Space (Section 2.1.2), which simplifies the hardware implementation.

QPtr Space, and several other address spaces, have NESAddress and NESData encoded within the sPBusAddress. NESAddress is an internal addressing scheme used to access BIU or Ctrl state; bit 6 of the sPBusAddress selects between the two. NESData contains the data for a state write if one is specified by the Update field.

Field	Content	Description
[0:5]	011010	QPtr Space
[6]	0	NESAddress[0] (Fixed)
[7:9]	X	NESAddress[3:5] == Comm Group
[10:11]	01	NESAddress[1:2]
[12:13]	X	** Unused **
[14:16]	100	NESAddress[6,9:10] (Fixed)
[17]	X	NESAddress[11]
[18]	X	** Unused **
[19:25]	X	NESData[0:6]
[26]	0	No Update
	1	Update
[27:28]	X	NESAddress[7:8]
[29]	X	High/Low Word (for 32b Reads)
[30:31]	00	** Fixed Field **

Table 2.9: sP QPtr Space

The use of NESAddress in the sBIU is explained in Appendix A.

Of particular note is NESAddress[3:5]; this is also called the Comm Group, since it encodes the short or medium message group which is being referenced. This encoding is shown in Table 2.10. PasT and VasR are short message transmit and receive queues, respectively; PcpT and PcpR are medium message queues.

Also of note in QPtr Space is the effect of the sPBusTransferType. A read-like transaction causes the sBIU to return data from a particular sSRAM location associated with the Comm Group; this location contains various status information about the Comm Group in question. A write-like transaction has no effect on the sSRAM; writes to NES state are implemented through the Update field, which causes the Ctrl state specified by the encoded NESAddress to be updated to the value contained in NESData if asserted.

ShTx Space

The ShTx (short message transmit) space allows short messages to be sent by a single uncached write from the sP. The message may be entirely contained in the

Group ID	Description
000	PasT-0/VasR-0 (Short Message) Group
001	PasT-1/VasR-1 (Short Message) Group
010	PasT-2/VasR-2 (Short Message) Group
011	PasT-3/VasR-3 (Short Message) Group
100	Ctrl: Pcp-0 (Medium Message) Group BIU: sBIU MemQIn/MemQOut Group
100	Ctrl: Pcp-1 (Medium Message) Group BIU: Overflow Group
100	Ctrl: Pcp-2 (Medium Message) Group BIU: aBIU MemQIn/MemQOut Group
100	Ctrl: Pcp-3 (Medium Message) Group BIU: aBIU DMA Group

Table 2.10: Comm Groups

Field	Content	Description
[0:6]	0110110	ShTx Space
[7]	0	PasT-0 Group
	1	PasT-1 Group
[8:27]	X	Message Content
[28]	0	Low Priority Queue
	1	High Priority Queue
[29]	X	High/Low Word (for 32b Writes)
[30:31]	00	** Fixed Field **

Table 2.11: sP ShTx Space

data portion of the write, or the sBIU can be used to compose a message header from the sP Address of the write. Additionally, up to 2.5 lines of “Tag-On Data” can be attached to the outgoing message by specifying this request in the message (in either format).

When a ShTx Space write is recognized, the data portion of the write is immediately placed in the sSRAM; the SRAM address used is specified in sBIU queue pointer state (see Section 3.3.3).

Additionally, if this particular message queue uses 32b Compose format (as specified in sBIU system registers; see Section 3.3.3) and the transfer is a 32b write to

Field	Content	Description
[0:6]	0110111	ShRx Space
[7:14]	X	** Unused **
[15]	0	VasR-0 Group: Do not poll
	1	VasR-0 Group: Poll
[16]	0	PcpR-0 Group: Do not poll
	1	PcpR-0 Group: Poll
[17]	0	VasR-1 Group: Do not poll
	1	VasR-1 Group: Poll
[18]	0	PcpR-1 Group: Do not poll
	1	PcpR-1 Group: Poll
[19]	0	MemQIn: Do not poll
	1	MemQIn: Poll
[20]	0	Overflow: Do not poll
	1	Overflow: Poll
[21:16]	X	** Unused **
[27]	0	High Priority Queue: Do not poll
	1	High Priority Queue: Poll
[28]	0	Low Priority Queue: Do not poll
	1	Low Priority Queue: Poll
[29]	x	High/Low Word (for 32b Reads)
[30:31]	00	** Fixed Field **

Table 2.12: sP ShRx Space

the odd word, sPBusAddress is written into the even word of the generated SRAM location through the JBus.

When the entire message is contained in sSRAM, the sBIU updates its internal queue pointers and updates NES Ctrl pointers via the SCBus.

ShRx Space

ShRx (short message receive) Space allows the user process to poll some or all of the various receive queues associated with that process. It uses a bit vector form of Comm Group to allow the user to select the queues to be polled and reads from the highest priority (according to a static ordering described in Table 2.13) non-empty polled queue, or returns data from a static location if no such queue exists.

The sBIU selects the queue to be read by ANDing the encoded polling information

Queue Name
VasR-0 High
PcpR-0 High
VasR-1 High
MemQIn
Overflow
PcpR-1 High
VasR-0 Low
PcpR-0 Low
VasR-1 Low
PcpR-1 Low

Table 2.13: sP ShRx Priorities (in descending order)

and the queue empty state (the internal short message queue state and the RxEmpty signals provided by Ctrl for medium message and overflow status) to determine the highest priority non-empty queue which is being polled.

For reads to the short message receive (VasR) queues, the sBIU returns the header of the message as indicated by internal queue pointers. At the completion of one 8B read or two 4B reads, the sBIU updates its own queue pointer state and forwards the update to NES Ctrl over the SCBus. MemQIn reads operate in a similar manner; however, MemQIn is located in the aBIU, so an update is passed along the SABus instead of the SCBus.

Reads to the medium message receive (PcpR) or overflow queues, as well as reads where there are no non-empty queues polled, result in the return of values from static sSRAM addresses and no internal state change.

Writes to this address space are ignored.

Special ShTx Space

The Special ShTx Space is very similar to the ShTx Space; both spaces are used to generate complete messages via uncached writes. There are two differences, however: Special ShTx Space has a wider message content field in the sP Address, and Special ShTx Space allows messages to be sent via MemQOut as well as the PasT (short

Field	Content	Description
[0:2]	010	Special ShTx Space
[3:4]	00	PasT-0 Group
	10	PasT-1 Group
	X1	MemQOut Group
[5:27]	X	Message Content
[28]	0	Low Priority Queue
	1	High Priority Queue
[29]	X	High/Low Word (for 32b Writes)
[30:31]	00	** Fixed Field **

Table 2.14: sP Special ShTx Space

Field	Content	Description	MemQOutOp
[3,15:16]	000	DMA Receive	01
	001	NES-Mastered Op	01
	010	DMA Read/Send	10
	011	DataMotion Command	11
	100	Message Launch	00

Table 2.15: MemQOutOp Encoding

message transmit) queues.

As in ShTx Space, Special ShTx Space allows complete message specification in data through a single 64b write or partial specification in the sPBusAddress through a 32b write. The message content change has no bearing on the sBIU, since the entire sPBusAddress is written to the SRAM through the JBus.

When one of the MemQOut queues is the target queue, the sBIU indicates this fact to NES Ctrl via the MemQOut0Compose and MemQOut1Compose signals. Further, the type of the MemQOut command is decoded as shown in Table 2.15 and passed on the MemQOutOp bus.

clSRAM Update Space

The clSRAM, which is used to determine cache line status for aP reads to global memory space (see Section 2.2.2) is only writable through clSRAM Update Space.

Field	Content	Description
[0:4]	01110	clSRAM Update Space
[5]	X	** Unused **
[6:26]	X	clSRAM Address
[27]	0	Do not acknowledge
	1	Acknowledge
[28]	0	Use MemQOut-0
	1	Use MemQOut-1
[29:31]	X	clSRAM Data

Table 2.16: sP clSRAM Update Space

The sBIU's response to this transaction is to latch the sP Address in an external FIFO and compose a MemQOut message which is directed to the aBIU; the aBIU then completes the clSRAM update when it receives the message, which keeps the update in FIFO order with other MemQOut transactions.

The Acknowledge signal is part of the clSRAM Update command; it is used to request a confirmation from the aBIU to the sP via MemQIn when the clSRAM update completes. See Section 4.5 for the details of this acknowledgment.

Config Access Space

Config Access Space is used at the system level to monitor and control NES status. This space is similar to QPtr Space; however, a wider range of states are accessible from Config Access Space, and the results of a read are different.

As with QPtr Space, NESAddress and NESData are explicitly provided in sPBusAddress; however, Config Access Space allows access to the full NESAddress space, while QPtr Space restricts access to a small region of queue state.

Reads to Config Access Space vary in result depending on the NESAddress being accessed. Reads to Ctrl End-Point State return the status pointers for the specified Comm Group, just as in QPtr Space. Reads to the Overflow Comm Group of Ctrl Queue State return overflow status from a static SRAM address.

Reads to the remainder of Ctrl state require that the state be written to a hard-

Field	Content	Description
[0:5]	011010	Config Access Space
[6]	0	NESAddress[0] = Ctrl Access
	1	NESAddress[0] = BIU Access
[7:9]	X	NESAddress[3:5] == Comm Group
[10:11]	X	NESAddress[1:2]
[12:13]	X	** Unused **
[14:17]	100	NESAddress[6,9:11]
[18]	X	** Unused **
[19:25]	X	NESData[0:6]
[26]	0	No Update
	1	Update
[27:28]	X	NESAddress[7:8]
[29]	X	High/Low Word (for 32b Reads)
[30:31]	00	** Fixed Field **

Table 2.17: sP Config Access Space

wired sSRAM location via a SCBus command; the sPBus read is delayed until this write is completed. Reads of sBIU state operate in a similar fashion; the state is written to the same SRAM location through the JBus and then read out by the sP. Finally, reads of aBIU state require a read request across the SABus, a write to SRAM on the JBus and finally a read from SRAM to the sPBus.

Write-like transaction types have no effect on the SRAM; state updates are indicated by the Update field, as in QPtr Space.

Immediate Commands

The Immediate Command address space is used to send specific commands to the Ctrl or aBIU modules. Immediate commands are decoded in the sBIU and passed via the relevant signals to Ctrl or aBIU as required.

2.1.3 Other sBIU Functions

In addition to responding to transactions issued by the sP, the sBIU also responds to state accesses issued across the CSBus and ASBus by Ctrl and aBIU.

Field	Content	Description
[0:5]	011110	Immediate Command Space
[6:14]	X	** Unused **
[15:17]	000	NES Reset
	001	** Unused **
	010	Arctic Ack
	011	Clear Ctrl DMA State
	100	Clear aBIU DMARxDataQ CPtr
	101	Clear Approval Register
	110	Interrupt aP
	111	Interrupt sP
[18:29]	X	** Unused **
[30:31]	00	** Fixed Field **

Table 2.18: sP Immediate Command Space

The CSBus is used by Ctrl to update sBIU's short message receive queue pointers; this information is needed so that the sBIU can determine if a given queue is empty when polled. The ASBus is used by the aBIU to read or write sBIU state as necessary.

2.2 aBIU Functionality

The aBIU is responsible for handling bus transactions issued by the aP (including control of the aSRAM port on the aP Data Bus); it also responds to various NES-issued commands, including commands which require it to act as master of the aPBus.

2.2.1 aBIU Interface Groups

The aBIU interfaces can be broken down into several groups: aPBus Interface, aSRAM Interface, KBus Interface, Ctrl Interface, sBIU Interface, NESBuffer Interface, clSRAM Interface and miscellaneous signals.

All signal directions are described with respect to the aBIU, i.e. "I" means an input to the aBIU.

Signal Name	Direction	Size	Comments
aPBusAddress	I/O	32b	Address
aPBusRequest	O	1b	Bus Request
aPBusGrant	I	1b	Bus Grant
aPBusTransferStart	I/O	1b	Start of Address Phase
aPBusTransferType	I/O	5b	Transfer Type
aPBusTransferCode0	I/O	1b	Transfer Code bit 0
aPBusTransferSize	I/O	3b	Transfer Size
aPBusTransferBurst	I/O	1b	Burst Transfer Indicator
aPBusGlobal	I/O	1b	Global
aPBusCacheInhibit	I/O	1b	Cache Inhibit
aPBusWriteThrough	I/O	1b	Write-Through
aPBusShared	I/O	1b	Shared
aPBusAddressBusBusy	O	1b	Address Phase in Progress
aPBusAddressAck	I	1b	Completion of Address Phase
aPBusL2Hit	O	1b	Address Claim
aPBusAddressRetry	I/O	1b	Address Retry
aPBusDataBusGrant	I	1b	Data Bus Grant
aPBusDataBusBusy	I/O	1b	Data Phase in Progress
aPBusTransferAck	O	1b	Completion of Data Phase
aPBusHardReset	I	1b	Hard Reset
aPBusSoftReset	I	1b	Soft Reset
aPBusInterrupt	O	1b	Interrupt sP

Table 2.19: aBIU-aPBus Interface

aPBus Interface

The aPBus Interface includes all of the signals relevant to the aBIU's ability to respond to transactions issued by the aP (or Union memory controller), as well as the signals needed to issue NES-Mastered transfers. The signals included in this Interface Group are listed in Table 2.19.

aPBusRequest is used when the aBIU requires bus mastery for a NES-Mastered operation; aPBusGrant indicates that the next bus cycle (indicated by negated aP-BusAddressBusBusy) is granted to the NES.

aPBusTransferStart, aPBusL2Hit and aPBusAddressRetry define the boundaries of the address phase of a non-NES-Mastered transfer, while aPBusDataBusBusy and aPBusTransferAck identify the boundaries of the data phase. NES-Mastered trans-

Signal Name	Direction	Size	Comments
aSRAMAddress	O	12b	aSRAM aP-Port Address
aSRAMRead/Write	O	1b	aSRAM aP-Port Read/_Write
aSRAMOutputEnable	O	1b	aSRAM aP-Port Output Enable
aSRAMChipEnable	O	2b	aSRAM aP-Port Chip Enable

Table 2.20: aBIU-aSRAM Interface

actions use aPBusGrant and aPBusAddressBusBusy to identify a granted address phase, drive aPBusTransferStart and aPBusAddressBusBusy to claim the bus and watch aPBusAddressAck to determine completion. aPBusDataBusGrant, aPBus-DataBusBusy and aPBusTransferAck control the corresponding data phase.

aPBusAddress, aPBusTransferType, aPBusTransferSize and aPBusTransferBurst indicate the type of transaction on the memory bus. aPBusTransferCode0, aPBus-Global, aPBusCacheInhibit, aPBusWriteThrough and aPBusShared further define the transfer. All of these signals are inputs for transfers controlled by the aP or the memory controller and outputs for NES-Mastered operations.

aPBusHardReset, aPBusSoftReset and aPBusInterrupt signals also fall under this interface; these signals may be used to reset the NES or generate interrupts on certain events (such as a message arrival).

aSRAM Interface

The aSRAM Interface consists of the signals required to control the port of the aS-RAM which is connected to the data portion of the aPBus; the IBus port of the aSRAM is controlled by NES Ctrl.

The aSRAMAddress signal addresses the SRAM, while aSRAMRead/Write, aS-RAMOutputEnable and aSRAMChipEnable serve as control signals.

KBus Interface

The KBus is a 64b data bus between aBIU and Ctrl which is used by the aBIU to request access to the aSRAM or sSRAM through the IBus (see Figure 4-1). There are

Signal Name	Direction	Size	Comments
KBusAddress	O	13b	KBus Compose Address
KBusData	O	64b	KBus Data
ShTxCompose	O	1b	32b Compose Request
MemQInComposeRead	O	1b	MemQIn Compose from Read
MemQInComposeWrite	O	1b	MemQIn Compose from Write
MemQInCtrlReq	I	1b	MemQIn Dummy Request
MemQInComposeCtrl	O	1b	MemQIn Dummy Compose
ComposeFree	I	1b	ShTx Compose Completion
DataMotionValid	O	1b	Data Motion Request
DataMotionFree	I	1b	Data Motion Acknowledge
DataMotionDone	I	1b	Data Motion Complete

Table 2.21: aBIU-KBus Interface

two distinct command types issued over the KBus: a DataMotion command enacts a transfer of data from one SRAM to the other, while a Compose command writes the provided data into a specified SRAM location. The KBus Interface controls these functions.

For short message composition, the KBusData contains the data to be written; the KBusAddress contains the aSRAM or aSRAM location of the write. ShTxCompose indicates that the KBus is being used to write the header of a short message (see Section 2.2.2); MemQInComposeRead and MemQInComposeWrite indicate that a transaction of the given type has been captured and is being written to MemQIn. (See Sections 2.2.2 and 2.2.2 for transaction captures.)

MemQInCtrlReq indicates that Ctrl wishes to place something in the MemQIn queue; MemQInCompCtrl indicates that the current MemQIn producer pointer is being passed along the KBusAddress bus and will be incremented when the compose completes.

For DataMotion instructions, KBusData encodes the aSRAM and sSRAM addresses, the transaction size and the direction of transfer. DataMotionValid indicates that the KBus is being used for a DataMotion transfer; DataMotionFree acknowledges the reception of the command by NES Ctrl, while DataMotionDone indicates

Signal Name	Direction	Size	Comments
ACBusAddress	O	11b	Ctrl State Address
ACBusData	O	7b	Ctrl Update Data
ACBusOp	O	2b	ACBus Operation
ACBusFree	I	1b	ACBus Acknowledge
ACBusDone	I	1b	ACBus Operation Done
CABusAddress	I	2b	aBIU State Address
CABusData	I	7b	aBIU Update Data
CABusValid	I	1b	aBIU State Update

Table 2.22: aBIU-Ctrl Interface

that the transfer is complete.

Ctrl State Interface

The Ctrl State Interface group consists of the ACBus, which is used by the aBIU to access Ctrl state, and the CABus, which is used by Ctrl to access aBIU state.

The ACBusAddress defines the Ctrl state to be accessed. ACBusOp defines the operation as a read and/or write or no-op. ACBusData provides the 7-bit data field for a state write. ACBusFree indicates that the ACBus is available for a new transaction, while ACBusDone indicates that the previous transaction has been completed.

The CABusAddress, which is only two bits wide, defines which of the four ShRx (short message receive) producer pointers are being updated by Ctrl. This data is required to allow the aBIU to calculate the full/empty status of the ShRx queues. CABusData contains the new pointer value; CABusValid indicates a valid operation on the bus. (There is no Free or Done signal, as the update is guaranteed to complete in one cycle.)

sBIU Interface

The aBIU interface is used by the aBIU to initiate and respond to state access requests by the sBIU. It consists of the SABus and the ASBus.

The ASBus is used by the aBIU to initiate sBIU state reads or writes. ASBusAd-

Signal Name	Direction	Size	Comments
SABusAddress	I	10b	aBIU State Address
SABusWData	I	7b	aBIU Update Data
SABusRData	O	7b	aBIU Read Data
SABusOp	I	2b	SABus Operation
ASBusAddress	O	10b	aBIU State Address
ASBusWData	O	7b	aBIU Update Data
ASBusRData	I	7b	aBIU Read Data
ASBusOp	O	2b	ASBus Operation

Table 2.23: aBIU-aBIU Interface

Signal Name	Direction	Size	Comments
NESBufferOp	I	64b	NESBuffer Command
NESBufferValid	I	1b	NESBufferOp Active
NESBufferFree	O	1b	Latched NESBufferOp
NESBufferDone	O	1b	Command Completed

Table 2.24: aBIU NESBuffer Interface

dress indicates the aBIU state to be accessed, ASBusOp indicates the transaction type (read, write or no-op), ASBusWData provides the data for a write and ASBusRData returns the data from a read.

The SABus is a mirror image of the ASBus which is used by the sBIU to access aBIU state. SABusAddress indicates the aBIU state to be accessed, SABusOp indicates the transaction type and ASBusWData and SABusRData supply/return the write or read data respectively.

NESBuffer Interface

The NESBuffer Interface is used by Ctrl to relay commands to the aBIU. These commands, as described in Section 2.2.3, are issued by the sP, sent through the MemQOut queue to Ctrl, and relayed from Ctrl to the aBIU across this interface. Table 2.24 lists the signals which comprise this interface.

NESBufferOp contains the command issued to the aBIU; NESBufferOpValid in-

Signal Name	Direction	Size	Comments
clSRAMData	I	3b	Cache Line Status
clSRAMRead/Write	O	1b	clSRAM Read/_Write
clSRAMUpdate	O	1b	Enable clFIFO Output
clSRAMDone	O	1b	Increment clFIFO

Table 2.25: aBIU clSRAM Interface

Signal Name	Direction	Size	Comments
RxEmpty	I	5b	Medium Receive Queue Status
RxLateAck	O	1b	Arctic Acknowledge
NESResetAP	O	1b	Reset Received
ResetDMA	I	1b	Reset aBIU DMA Counter
ClearApproval	I	1b	Clear Approval Register

Table 2.26: aBIU Miscellaneous Interfaces

dicates that a command is being issued. NESBufferFree indicates that the aBIU is ready to accept another command, while NESBufferDone indicates that the earliest outstanding command has been completed.

clSRAM Interface

The clSRAM interface has two functions: it is used to read cache line state for transactions to aP Snooped Space (Section 2.2.2), and it is used to complete clSRAM updates requested by a NESBuffer command (Table 2.40).

clSRAMData is the status of the current cache line being read; this is an input only, as the updates are done through an external FIFO to save pins. clSRAMRead/Write is used to control the SRAM itself; clSRAMUpdate places the current clFIFO output onto the address and data ports of the clSRAM, while clSRAMDone indicates that the clFIFO should move to the next item in its queue.

Miscellaneous Interfaces

There are a few other I/O signals which do not fall under any of the previous interface groups.

The RxEmpty signal indicates the status of the medium message receive queues; this information allows the aBIU to implement queue polling.

RxLateAck indicates that the sP has read a message from a queue which requires that a late acknowledgment be sent to the Arctic network. This use of late acknowledgments is controlled by a aBIU System Register (see Section 4.3.3). This signal is also used in response to the Arctic Ack Immediate Command. (The Immediate Command Space is described in Section 2.2.2.)

NESResetSP indicates that a Hard or Soft Reset has been received on the aPBus, or that the NES Reset Immediate Command has been issued.

CLSLatch causes the aPBusAddress to be latched into an external FIFO, which is used to allow clSRAM updates to be processed in FIFO order. ResetDMA and Clear-Approval clear the DMA counter and Approval Register, respectively. See Table 2.40 for details.

2.2.2 aP Address Space

The aP has access to a full range of address spaces which are used to support direct access to the aSRAM and sSRAM, updates of the clSRAM, access to aBIU, aBIU and Ctrl internal state at user or system level and access to message receive and transmit queues.

The address space of the transaction is encoded in bits [0:6] of the aPBusAddress, as shown in Table 2.27. Further decoding of the address depends on the address space. (All unlisted address spaces are ignored.)

The aBIU can respond to any transaction issued to Snooped Space or Serviced Space; only read-like and write-like transactions issued to other spaces are supported.

The aBIU also reacts to certain address-independent commands (like Sync) that may be issued on the aPBus.

Field	Content	Description
[0:6]	000001x	Snooped Space
	001xxxx	aP Serviced Space
	010xxxx	
	01100xx	SRAM Space
	011010x	QPtr Space
	0110110	ShTx Space
	0110111	ShRx Space
	011110x	Immediate Command Space
	011111x	Config Access Space

Table 2.27: aP Address Space

Field	Content	Description
[0:4]	01100	SRAM Space (Fixed)
[5:15]	X	** Unused **
[16]	0	aSRAM Access
	1	aSRAM Access
[17:29]	X	SRAM Address
[30:31]	00	** Fixed Field **

Table 2.28: aP SRAM Space

SRAM Space

The simplest functionality supported by the aBIU is direct access to the aSRAM and aSRAM. For aSRAM access, the aBIU simply acts as a memory controller, providing the appropriate address and control signals to the SRAM. For sSRAM access, the aBIU coordinates a transfer to/from a temporary aSRAM location with a DataMotion request on the KBus.

The aBIU supports direct access to the aSRAM through the control signals described in Section 2.2.1. These accesses are limited to 4B and 8B aligned transfers and 32B burst transfers; burst writes are assumed to be zeroth word first, while reads are returned critical word first.

In the case of an sSRAM write, the provided data is stored in a temporary aSRAM location; a DataMotion request to transfer data from the temporary aSRAM to the

Field	Content	Description
[0:5]	011010	QPtr Space
[6]	0	NESAddress[0] (Fixed)
[7:9]	X	NESAddress[3:5] == Comm Group
[10:11]	01	NESAddress[1:2] (Fixed)
[12:13]	X	** Unused **
[14:16]	100	NESAddress[6,9:10] (Fixed)
[17]	X	NESAddress[11]
[18]	x	** Unused **
[19:25]	X	NESData[0:6]
[26]	0	No Update
	1	Update
[27:28]	X	NESAddress[7:8]
[29]	X	High/Low Word (for 32b Reads)
[30:31]	00	** Fixed Field **

Table 2.29: aP QPtr Space

specified sSRAM location is then issued on the KBus. An sSRAM read is handled in similar fashion: DataMotion is requested from the specified sSRAM address to a temporary aSRAM location, and the read from this location is executed when the data motion completes. Again, 4B, 8B and 32B transactions are supported with the same restrictions as aSRAM transfers.

QPtr Space

QPtr Space is designed to allow user-level access to short and medium message queue pointers in NES Ctrl. It is intentionally similar to the system level access provided in Config Access Space (Section 2.2.2), which simplifies the hardware implementation; however, QPtr Space is considerably more restricted than Config Access Space.

QPtr Space, and several other address spaces, have NESAddress and NESData encoded within the aPBusAddress. NESAddress is an internal addressing scheme used to access BIU or Ctrl state; bit 6 of the aPBusAddress selects between the two. NESData contains the data for a state write if one is specified by the Update field. The use of NESAddress in the aBIU is explained in Appendix A.

Field	Content	Description
[0:6]	0110110	ShTx Space
[7]	0	PasT-2 Group
	1	PasT-3 Group
[8:27]	X	Message Content
[28]	0	Low Priority Queue
	1	High Priority Queue
[29]	X	High/Low Word (for 32b Writes)
[30:31]	00	** Fixed Field **

Table 2.30: aP ShTx Space

Of particular note is NESAddress[3:5]; this is also called the Comm Group, since it encodes the short or medium message group which is being referenced. This encoding is shown in Table 2.10. PasT and VasR are short message transmit and receive queues, respectively; PcpT and PcpR are medium message queues.

Also of note in QPtr Space is the effect of the aPBusTransferType. A read-like transaction causes the aBIU to return data from a particular aSRAM location associated with the Comm Group; this location contains various status information about the Comm Group in question. A write-like transaction has no effect on the aSRAM; writes to NES state are implemented through the Update, NESAddress and NESData fields. In the case of QPtr Space, the NESAddress is restricted to Ctrl state, so updates are passed to Ctrl on the ACBus.

ShTx Space

The ShTx (short message queue) space allows short messages to be sent by a single uncached write. The message may be entirely contained in the data portion of the write, or the aBIU can be used to compose a message header from the aP Address of the write. Additionally, up to 2.5 lines of "Tag-On Data" can be attached to the outgoing message by specifying this request in the message (in either format).

When a ShTx Space write is recognized, the data portion of the write is immediately placed in the aSRAM location indicated by the queue pointers resident in the

Field	Content	Description
[0:6]	0110111	ShRx Space
[7:14]	X	** Unused **
[15]	0	VasR-2 Group: Do not poll
	1	VasR-2 Group: Poll
[16]	0	PcpR-2 Group: Do not poll
	1	PcpR-2 Group: Poll
[17]	0	VasR-3 Group: Do not poll
	1	VasR-3 Group: Poll
[18]	0	PcpR-3 Group: Do not poll
	1	PcpR-3 Group: Poll
[19:16]	X	** Unused **
[27]	0	High Priority Queue: Do not poll
	1	High Priority Queue: Poll
[28]	0	Low Priority Queue: Do not poll
	1	Low Priority Queue: Poll
[29]	x	High/Low Word (for 32b Reads)
[30:31]	00	** Fixed Field **

Table 2.31: aP ShRx Space

aBIU.

Additionally, if this particular message queue uses 32b Compose format (as specified in aBIU queue state; see Section 4.3.3) and the transfer is a 32b write to the odd word, aPBusAddress is written into the even word of the 8B SRAM block through the KBus.

When the entire message is contained in aSRAM, the aBIU updates its internal queue pointers and updates NES Ctrl pointers via the ACBus.

ShRx Space

ShRx (short receive queue) Space allows the user process to poll some or all of the various receive queues associated with that process. It uses a bit vector form of Comm Group to allow the user to select the queues to be polled and reads from the highest priority (according to a static ordering described in Table 2.32) non-empty polled queue, or returns data from a static location if no such queue exists.

Queue Name
VasR-2 High
PcpR-2 High
VasR-3 High
PcpR-3 High
VasR-2 Low
PcpR-2 Low
VasR-3 Low
PcpR-3 Low

Table 2.32: aP ShRx Priorities (in descending order)

The aBIU selects the queue to be read by ANDing the encoded polling information and the queue empty state (the internal short message queue status and the RxEmpty signals provided by Ctrl for medium message queue status) to determine the highest priority non-empty queue which is being polled.

For reads to the short message receive (VasR) queues, the aBIU returns the header of the message as indicated by internal queue pointers. At the completion of one 8B read or two 4B reads, the aBIU updates its own queue pointer state and forwards the update to NES Ctrl over the ACBus.

Reads to the medium message receive (PcpR) queues, as well as reads where there are no non-empty queues polled, result in the return of values from static aSRAM addresses and no internal state change.

Writes to this address space are ignored.

Config Access Space

Config Access Space is used at the system level to monitor and control NES status. This space is similar to QPtr Space; however, a wider range of states are accessible from Config Access Space, and the results of a read are different. This space is identical to sP Config Access Space, shown in Table 2.17.

As with QPtr Space, NESAddress and NESData are explicitly provided in aPBusAddress; however, Config Access Space allows access to the full NESAddress space,

Field	Content	Description
[0:5]	011110	Immediate Command Space
[6:14]	X	** Unused **
[15:17]	000	NES Reset
	001	** Unused **
	010	Arctic Ack
	011	Clear Ctrl DMA State
	10X	** Unused **
	110	Interrupt aP
	111	Interrupt sP
[18:29]	X	** Unused **
[30:31]	00	** Fixed Field **

Table 2.33: aP Immediate Command Space

while QPtr Space restricts access to a small region of queue state.

Reads to Config Access Space access vary in function depending on the state being accessed. Reads to Ctrl End-Point State return the status pointers for the specified Comm Group, just as in QPtr Space. Reads to the remainder of Ctrl state require that the state be written to a hardwired aSRAM location via a ACBus command; the aPBus read is delayed until this write is completed.

Reads of aBIU state operate in a similar fashion; the state is written to the same SRAM location via a KBus transaction and then read out by the sP. Finally, reads of sBIU state require a read request across the ASBus, a write to aSRAM on the KBus and finally a read from aSRAM on the aPBus.

Write-like transaction types have no effect on the SRAM; state updates are indicated by the Update field, as in QPtr Space, and may be forwarded to Ctrl or aBIU by the ACBus or ASBus respectively.

Immediate Commands

The Immediate Command address space is used to send specific commands to the Ctrl module. These commands are decoded in the aBIU and used to assert the relevant signals to Ctrl.

Field	Content	Description
[0:5]	000001	Snooped Space
[6:25]	X	clSRAM Address
[26:31]	X	Interpreted by sP

Table 2.34: aP Snooped Space

Snooped Space

Snooped Space is used to implement shared local global memory. The response of the aBIU to this space is determined by the aPBusTransferType and cache-line status received from the clSRAM. Sections 4.2 and 4.3 describe the implementation of this functionality.

When a transfer to Snooped Space is attempted, the aBIU does a table lookup indexed by the aPBusTransferType and the clSRAMData corresponding to the cache line of the transfer. There are four possible outcomes: ignore, notify, approve or retry indefinitely.

Ignored transactions elicit no further action by the aBIU.

Transactions which require notification are placed in MemQIn over the KBus Interface. If the transfer is a write, the associated data is placed in MemQDataIn. In either case, the aBIU does *not* assert any control signals, so that the read or write can be completed to aP DRAM.

If a transaction requires approval, the state of the Approval Register is checked. If the Approval Register is free, the transaction is placed in the register, an approval request is sent over MemQIn, and the aPBus transfer is retried. If the Approval Register is in pending or locked state, the aPBus transfer is retried and no further action is taken by aBIU; this is also the case if the Approval Register is ready but the associated address does not match this transaction. If the Approval Register is ready and the address does match, the transaction is allowed to complete, with data written to or read from the SRAM address supplied by the Approval Register.

If the transaction indicates indefinite retry, it is retried with no further aBIU action.

Field	Content	Description
[0:2]	001	Serviced Space
	010	
[3:31]	X	Interpreted by sP

Table 2.35: aP Serviced Space

See Sections 2.2.3 and 4.3 for more details on the use and implementation of the Approval Register mechanism.

Serviced Space

Serviced Space is used to support global shared memory and other communication to the sP. As with Snooped Space, the aBIU can ignore, capture, retry or service sP Serviced Space transactions; the response to these transactions is dictated entirely by aPBusTransferType, however.

Once the response – ignore, notify, approve or retry – is determined from the Transfer Type, Serviced Space acts exactly as described above for Snooped Space with two exceptions: the aBIU always asserts the control signals appropriate for the slave of the current transaction, and data is read from/ written to a default aSRAM location for transactions with ignore or notify responses.

2.2.3 NESBuffer Commands

The NESBuffer is used to pass a variety of commands to the aBIU, including operations to be executed on the aPBus, clSRAM updates, Approval Register state changes and DMA commands.

The various commands are distinguished by bits 14, 16 and 18 of the NESBufferOp, as detailed in Table 2.36.

The exact encoding of the various operations, as well as the actions required for each one, are described in the following sections. Two common functions exist: any NESBuffer command can explicitly increment the DMARxDataQ CPtr, as indicated

Field	Content	Description
[14,16]	00	DMA Receive Command
	01	NES-Mastered Operation
	10	DMARx-Mastered Operation
	11	Miscellaneous Commands

Table 2.36: NESBuffer Operations

Field	Content	Description
[0]	X	DMAIncrement
[1:3]	X	** Unused **
[4:13]	X	aSRAM Address
[14:16]	100	** Fixed Field **
[17]	0	Do not acknowledge
	1	Acknowledge
[18:31]	X	aPBus control signals
[32:61]	X	aPBusAddress
[62:63]	X	aPBus transfer size

Table 2.37: NES-Mastered Operation

by the DMAIncrement field; and any NESBuffer command can request that the command be acknowledged by placing the exact command, with bit [28] set to 1, into MemQIn.

NES-Mastered Operations

NES-Mastered Operations require that the aBIU execute a specified transaction on the aPBus. The aSRAM address and all aPBus signals are fully specified. When the transaction completes, the aBIU increments the DMARxDataQ consumer pointer and/or returns an acknowledgment through the MemQIn queue as requested.

DMA Rx Operations

DMARx-Mastered Operations, like NES-Mastered Operations, require that the aBIU execute a specified transaction on the aPBus.

Field	Content	Description
[0]	X	DMAIncrement
[1:10]	X	** Unused **
[11:13]	X	DMAChannel
[14:16]	001	** Fixed Field **
[17]	0	Do not acknowledge
	1	Acknowledge
[18:31]	X	aPBus control signals
[32:61]	X	aPBusAddress
[62:63]	X	aPBus transfer size

Table 2.38: DMARx-Mastered Operation

In the case of a DMARx-Mastered Operation, data is provided from the DMARx-DataQ queue in aSRAM. If requested, the Approval Register is cleared upon completion of the transaction. The DMARxDataQ consumer pointer is incremented and/or an acknowledgment is returned through MemQIn if specified. Finally, the DMAPending counter for the indicated DMAChannel is decremented when a successful transfer is executed; if the counter reaches zero, a special acknowledgment is returned to the sP through MemQIn.

DMA Receive

DMA Receive commands initialize the DMAPending counter for a given DMAChannel; like other NESBuffer Operations, an acknowledgment is sent via MemQIn if requested.

Miscellaneous Commands

In addition to the above, the NESBuffer is also used to update the Approval Register and Notification status and to initiate a clSRAM update.

The NESBuffer is one mechanism used to change the Approval Register contents or related state. See Sections 2.2.4 and 4.1.2 for more details.

The NESBuffer can also be used to lock and unlock the aBIU's Notification resource; any transfers requesting notify while this resource is locked are retried.

Field	Content	Description
[0]	X	DMAIncrement
[1:10]	X	Unused
[11:13]	X	DMAChannel
[14:16]	000	** Fixed Field **
[17]	0	Do not acknowledge
	1	Acknowledge
[18:47]	X	** Unused **
[48:63]	X	DMAPending[DMAChannel]

Table 2.39: DMA Receive

Field	Content	Description
[0]	X	DMAIncrement
[1:13]	X	** Unused **
[14:16]	101	** Fixed Field **
[17]	0	Do not acknowledge
	1	Acknowledge
[18]	0	No clSRAM Update
	1	clSRAM Update
[19:46]	X	** Unused **
[47:49]	X	Approval Command
[50:59]	X	New ApprSRAMAddress
[60:63]	X	** Unused **

Table 2.40: Miscellaneous NESBuffer Commands

Finally, if the clSRAMUpdate field is set, the aBIU performs a write to clSRAM using the address and data provided by the external clFIFO (see Section 2.1.2).

2.2.4 Approval Register

The Approval Register is a mechanism used by the aBIU to support carefully controlled aPBus transfers to the Serviced Space and Snooped Space address spaces. The general concept is to force some transfers to these spaces to be "approved" by the sP before completion, which allows the sP to fetch remote data or perform any other required tasks without completely stalling the aP. When the sP completes the required tasks, approval can be granted and the transfer can be allowed to complete.

ApprRegState is the current status of the approval register; it can be FREE, PENDING, READY or LOCKED. ApprRegAddress contains the aPBus Address of the transaction which requires approval; ApprSRAMAddress contains the SRAM location of data to be provided to any approved reads.

When a transaction requires approval, the state of the Approval Register is checked. If ApprRegState == FREE, the transaction is retried, the address is stored in ApprRegAddress, ApprRegState transitions to PENDING and an approval request is sent through MemQIn.

If ApprRegState == LOCKED or PENDING, or if ApprRegState == READY and the aPBusAddress does not match ApprRegAddress, the transaction is retried with no further action

If ApprRegState == READY and the addresses do match, the transaction is allowed to complete; read data is provided from the SRAM location specified in ApprSRAMAddress, while write data is placed in MemQDataIn.

2.2.5 Other aBIU Functions

In addition to responding to transactions issued by the sP, the aBIU also responds to state accesses issued across the CABus and SABus by Ctrl and sBIU.

The CABus is used by Ctrl to update aBIU's short message receive queue pointers;

this information is needed so that the aBIU can determine if a given queue is empty when polled. The SABus is used by the sBIU to read or write aBIU state as necessary.

2.3 Summary

The sBIU and aBIU are designed to observe and response to commands issued on the sPBus or aPBus respectively; these commands are encoded within the address of a bus transfer. In order to carry out these commands, the BIUs must interact with other NES components; the interface groups described here allow that interaction.

Chapter 3

sBIU Implementation

This chapter describes the functionality of the sBIU. First, an overview of the sBIU is given, describing the assignment of functionality to the various submodules within the sBIU and the interaction between these submodules. The submodules are then described in detail, including all finite state machines (FSMs), registers and logic used to provide the required functionality.

3.1 Overview

The sBIU is broken into three distinct submodules: the sBIU Bus Interface, or sBI; the sBIU Queue/State, or sQS; and the sBIU Ctrl Interface, or sCI. Figure 3-1 shows these submodules and the primary interfaces of each.

One further general comment bears mentioning: all signals leading into or out of the sBIU are latched. Hence, input signals are "received" one cycle after they appear on the port, and output signals are "asserted" one cycle before they are actually seen by the other end of the connection.

3.1.1 Submodule Descriptions

The sBI submodule is responsible for interacting with the sPBus; it asserts the control signals to respond to a sPBus transaction as specified by the 60X bus protocol,

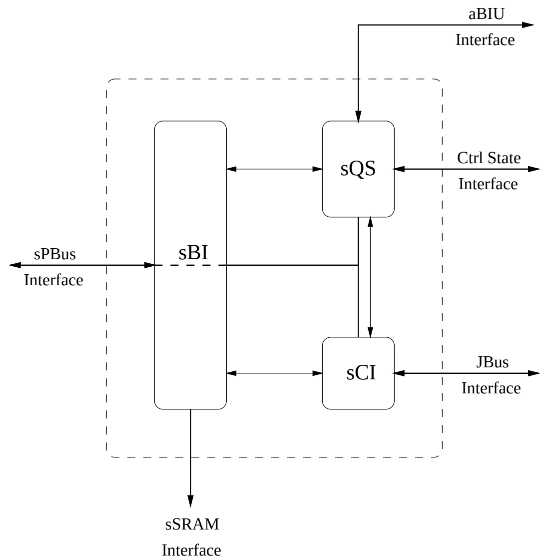

Figure 3-1: sBIU Overview

coordinates the timing of events within the sBIU, and controls the sSRAM. It has complete control of the sPBus Interface (see Section 2.1.1) and sSRAM Interface (Section 2.1.1); it is also responsible for decoding and forwarding Immediate Commands (Section 2.1.2) to the aBIU or Ctrl modules.

The sQS submodule is responsible for maintaining and manipulating the sBIU's queue pointers, system registers and other global state; it also uses this state to generate the appropriate sSRAM address for each transaction. It operates the Ctrl State Interface (Section 2.1.1) and aBIU State Interface (Section 2.1.1) as well as the RxEmpty status bus.

The sCI submodule is responsible for the operation of the JBus, the internal data bus by which the sBIU can request access to aSRAM and sSRAM data. The JBus Interface (Section 2.1.1) is used to handle these requests.

3.1.2 Internal Signals

There are several signals internal to the sBIU which are used to coordinate activity between the submodules.

The sBI generates several internal sPBus signals: sPBusAddressI, sPBusTrans-

Signal Name	Size	Source	Destination	Comments
sPBusAddressI	32b	sBI	sQS, sCI	Transfer Address
sPBusTransferSizeI	3b	sBI	sQS, sCI	Transfer Size
sPBusTransferStartI	1b	sBI	sQS, sCI	Transfer Start
sPBusAddressRetryI	1b	sBI	sQS, sCI	Transfer Retry
sPBusTransferAckI	1b	sBI	sQS, sCI	Transfer Data Done
AddressRead	1b	sBI	sQS, sCI	Transfer = Write
AddressWrite	1b	sBI	sQS, sCI	Transfer = Read
AddressConfirm	1b	sBI	sQS, sCI	Transfer Address Done
DataMotionReq	1b	sBI	sCI	DataMotion Request
SRAMAddress	12b	sQS	sBI, sCI	sSRAM Address
ComposeReq	1b	sQS	sCI	Compose Request
MemQOut0Req	1b	sQS	sCI	MemQOut0 Compose
MemQOut1Req	1b	sQS	sCI	MemQOut1 Compose
clsReq	1b	sQS	sCI	clSRAM Update Compose
sQSRetry	1b	sQS	sBI	Retry Required
ComposeAvail	1b	sCI	sQS	Compose Available
DataMotionAvail	1b	sCI	sBI	DataMotion Available
DataMotionDone	1b	sCI	sBI	DataMotion Done

Table 3.1: sBIU Internal Signals

ferSizeI, sPBusTransferStartI, sPBusAddressRetryI and sPBusTransferAckI. These signals are directly latched from the corresponding signals described in Section 2.1.1. The AddressRead and AddressWrite signals indicate the results of the transfer type decoding; AddressConfirm indicates that the most recent address has been completed without retry, allowing sQS and sCI to proceed with irreversible state changes. DataMotionReq indicates that the current address is an aSRAM access and thus requires that sCI send a DataMotion command on the JBus at the appropriate time.

A primary function of sQS is the generation of SRAMAddress, the sSRAM address corresponding to the current sPBusAddressI value. ComposeReq, MemQOut0Req, MemQOut1Req and clsReq are used to request the specified Compose command to sCI. sQSRetry indicates a resource conflict within sQS, and causes the sBI to retry the current address transfer.

sCI generates ComposeAvail and DataMotionAvail, which indicate the current status of the Compose and DataMotion buffers. It also relays DataMotionDone to the sBI so that transfers waiting for a DataMotion command may proceed.

3.2 sBI Implementation

The sBI submodule, as described above, is responsible for observing and responding to transactions on the sPBus. In doing so, it must conform to the 60X bus protocol [12] and meet the requirements of the MPC105 memory controller [11]. The finite state machine and associated logic which generate the sPBus and sSRAM signals compose the bulk of this logic.

3.2.1 Bus Interface Finite State Machine

The sPBus is a 60X split phase bus, with address and data phases of a given transfer occurring separately and governed by different signals. The two phases can overlap, or the data phase can occur after the corresponding address phase has completed.

The MPC105 restricts the pipelining of address and data phases. In principle, the 60X bus places no restriction on this pipelining; any number of address phases may complete while a single data phase is in progress or pending, with the corresponding data phases waiting in order for bus availability. The MPC105 controller does not allow such pipelining, however; it does not allow any address phase to complete until the preceding data phase is completed.

The pipelining restriction of the MPC105 and the standard 60X bus protocol provide the motivation for the primary sBI finite state machine, as demonstrated in in Figure 3-2. This is essentially a two-dimensional FSM; a transition along the vertical direction indicates an address phase change, while a transition in the horizontal direction indicates a data phase event. The events which occur at each state, as well as the conditions for the possible transitions from each state, are described below.

One point of 60X bus protocol is particularly relevant to the FSM: the Address-Retry window occurs after the “acknowledgment” of the address phase. Due to the split phase nature of the 60X bus, the retry window can occur after the data phase has been initiated, and cancels such a data transfer if it exists. In order to allow this,

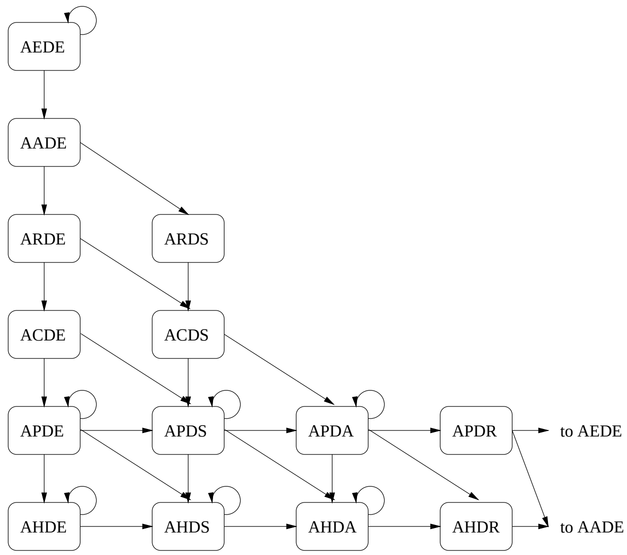

Figure 3-2: sBI Finite State Machine

State	Address Phase Status	Data Phase Status
AEDE	AddressEmpty	DataEmpty
AADE	AddressActive	DataEmpty
ARDE	AddressRelease	DataEmpty
ACDE	AddressConfirm	DataEmpty
APDE	AddressPending	DataEmpty
AHDE	AddressHold	DataEmpty
ARDS	AddressRelease	DataSetup
ACDS	AddressConfirm	DataSetup
APDS	AddressPending	DataSetup
AHDS	AddressHold	DataSetup
APDA	AddressPending	DataActive
AHDA	AddressHold	DataActive
APDR	AddressPending	DataRelease
AHDR	AddressHold	DataRelease

Table 3.2: sBI Finite State Machine States

the data phase is always stalled until the retry window passes, and may be cancelled if a retry is received.

As stated above, the MPC105 memory controller restricts pipelining of address phases; hence, the state machine stalls any address phase until the preceding data phase has completed.

With these exceptions, it is clear that the address and data portions of the FSM are independent, as are the address and data phases of a 60X bus transfer. Each state of the FSM is composed of an address and data phase state; the bus signals for the address or data phase are generated independently of the opposite state. The only use of the complete state is to determine state transitions.

3.2.2 Address Phase Events

During the address phase, the sBI reads, decodes and latches the address, type and size of the current transaction; assuming this transaction is to a sBIU-serviced address space, the sBI generates the appropriate control signals to complete the address phase of the transfer. The coordination of these events is handled by one part of the state of the sBI FSM (Figure 3-2).

The *AddressEmpty* state indicates that there is no address phase currently in progress. No address-related signals are asserted during this state; transition to AddressActive occurs when a new address phase is detected on the sPBus.

AddressActive is the state in which the sPBusAddress and related signals are decoded. The sPBusAddressAck signal is generated during this phase (if the transfer is to sBIU-serviced space). If a DataMotion transfer between the aSRAM and SSRAM is required, a request is sent to the sCI submodule. Also, this phase is used to detect and register any resource conflicts within the sBIU; if the command cannot be completed, a register is set indicating that a retry is necessary. Transition to AddressRelease is automatic.

AddressRelease is used to place a retry request on the sPBus when required and to deassert all other 60X control signals. Also, sSRAM address and control signals, and any ordering constraints on the data phase of the transfer, are generated (received from sQS for sSRAM address) and latched during this phase. Transition to AddressConfirm is automatic.

The *AddressConfirm* state is used to indicate to other sBIU units that an address phase has completed without being retried, which allows those units to finalize state updates. Immediate commands are issued to the aBIU or Ctrl in this state; no other signals external to the sBIU are generated. At the end of a cycle, transition to AddressPending (if the transfer has a data component) or AddressEmpty (if not) occurs.

The *AddressPending* state indicates that the address phase has been confirmed and the data phase is in progress. State transition occurs when the data phase completes and/or a new address phase is detected. If a new address phase is detected before the data phase completes, transition is to AddressHold. If a new address phase is detected and the data phase completes on the same cycle, transition is to AddressActive. If the data phase completes without a new address phase detection, transition is to AddressEmpty. No control signals are asserted in this state.

AddressHold indicates that a new address phase has begun while a data phase is in progress; this state is used to stall the address portion of the FSM until the current completes, at which point the state transitions to AddressActive. Again, no control signals are generated in this state.

3.2.3 Data Phase Events

The primary task of the data phase portion of the sBI FSM is coordination of the transfer of data between the sPBus and the sSRAM.

DataEmpty indicates that no data phase is in progress. No signals are asserted in this state, and transition to DataSetup (if the data phase is a read or must be stalled) or DataActive (no stall, write) occurs when the beginning of a data phase is indicated.

DataSetup is used to drive sSRAM address and control signals to prepare for a read, and to stall the data phase until the address phase is “confirmed” and any required JBus or SCBus commands are completed. Transition to DataActive occurs when no wait conditions exist.

DataActive is the state in which data is read from or written to the sSRAM. The sSRAM control signals and sPBus signals are used to process the transfer of a single beat of data; if the current transfer is a burst, the latched sSRAM address is incremented as well. This state transitions after one (non-burst transfer) or four (burst transfer) cycles to DataRelease.

DataRelease is used to deassert all data phase sPBus and sSRAM signals. It transitions immediately to DataEmpty.

3.3 sQS Implementation

The sQS submodule is designed to generate sSRAM addresses based on the address and type of the current sPBus transfer, and to maintain the sBIU queue pointer and system register state. These two functions are related, and combined in a single submodule, because the sSRAM address generation is heavily dependent on the queue pointer values.

3.3.1 sSRAM Address Generation

The sSRAM address which is used for sPBus transfers is generated in sQS based on the address and type of the transfer. Additionally, read or update requests to sBIU, aBIU and Ctrl state are generated after the sPBusAddress is decoded in the sQS. Read requests are issued when the transfer begins, while update requests are delayed until the address phase of the transfer is confirmed by sBI. Further details are given in Section 3.3.2.

Address generation for the various sPBus address spaces is done in parallel; the appropriate address for the actual sPBusAddress is selected by a large mux.

Readers are directed to Section 2.1.2 and the StarT-Voyager Hardware Engineering Spec [1] for further details about the encodings of the various address spaces.

SRAM Space

Generation of the SRAM address for SRAM Space is straightforward. If bit [16] of the sPBusAddress is asserted, the actual access is to the aSRAM; a static address named aSRAMTmp is thus provided as the sSRAM address. If bit [16] is deasserted, the fully-specified sSRAM address in the sPBusAddress is provided to sBI.

QPtr Space

The SRAM address for QPtr Space consists of a fixed base address named QPtrTmp and an offset which consists of bits [7:9] (the Comm Group) of the sPBusAddress; NES Ctrl maintains status information for the Comm Groups at these locations.

Additionally, if the Update field in sPBusAddress is asserted, an update request is placed in the SCBus buffer for processing once the address phase is confirmed. If this buffer is occupied, sBI is notified so that the current transfer can be retried.

ShTx Space

The sSRAM address for ShTx (Short Transmit) Space address is composed from the concatenation of the Base and PPtr (Producer Pointer) state for the given PasT (short transmit) queue. If the 32bCompose register is set for this queue, and the current transfer is a 4B write, sQS also requests that the sPBusAddress be written to the even word of the sSRAM address via the JBus Interface. The sBIU PPtr is incremented (or zeroed if it matches the Bound register for the queue) at the conclusion of the address phase; the command to update Ctrl PPtr is issued on the SCBus when the data portion of the transfer completes (the message must be present in the sSRAM before the Ctrl pointer is updated).

See Section 3.3.3 for a description of the state associated with the short transmit queues.

ShRx Space

The ShRx (Short Receive) Space format uses a bit vector (with static priorities) to determine which queue is to be polled; hence, the first step in generating the SRAM address and related actions is determining which queue is actually to be read. This is done by decoding the address to form a full prioritized bit vector of the queues to be polled, ANDing this vector with a vector of the non- empty queues (as determined by internal queue state and the RxEmpty signal from Ctrl) and then using a priority encoder to determine the actual queue being accessed.

If the queue being read is a VasR (short receive) queue, the address is generated by concatenating the Base and CPtr (Consumer Pointer) registers for the selected queue. Also, the sBIU and Ctrl CPtr values are incremented (or zeroed if the CPtr matches the Bound value for the queue) at the conclusion of the address phase of the transfer. Finally, if the LateAck register for the queue is set, a late acknowledgment is sent to the Arctic network via Ctrl.

On the other hand, if the queue indicated is a PcpR (medium receive) queue or the overflow queue, or if none of the polled queues is non-empty, the SRAM address generated is a static Continuation Address corresponding to the poll result. No state updates occur for this case.

One exception to the above process exists: if the transfer is a 4B read, a special OnePoll? register is tested. If the register returns a positive value, the SRAM address is taken from the OnePollAddress register, the OnePoll? register is cleared and no state update occurs; if the OnePoll? register returns a negative value, the SRAM address is generated as above and placed into the OnePollAddress register (as well as being provided to the sBI as usual) and the OnePoll? register is set. This special mechanism is used to support interruptible 4B queue reads, since the OnePoll? and OnePollAddress registers are sP accessible system registers.

Again, Section 3.3.3 describes the queue status registers in more detail.

Special ShTx Space

The SRAM address generation for Special ShTx Space transfers which indicate one of the PasT queues executes exactly as in ShTx Space. Transfers which involve one of the MemQOut queues are processed in a similar manner; however, instead of using the SCBus to update Ctrl state, MemQOut0Comp / MemQOut1Comp are used to indicate a composition to the indicated MemQOut and MemQOutOp is used to indicate the type of the transfer as encoded in the sPBusAddress (see Section 2.1.2).

Config Access Space

The response to transfers in the Config Access Space depends on the state referenced by the NESAddress encoded in the instruction. The encoding can be found in Section 2.1.2; NESAddress details are described in Appendix A.

sSRAM address generation depends only on the state being accessed. If the access is to sBIU state, aBIU state or Ctrl QRAM or SysReg state, the sSRAM address is a special static location called QConfigTmp. If the access is to Ctrl EPRAM state, the sSRAM address is generated from QPtrBase and the Comm Group of the access as in QPtr Space; and if the access is to the Overflow group, the sSRAM address is a static OverflowStatus location.

Additional actions depend on the transfer type of the sPBus transaction and the Update field in the sPBusAddress. If the transfer type indicates a read and the state referenced is in the sBIU, sQS arranges for the state to be written to the QConfigTmp location in sSRAM via the JBus when the address phase completes; if a read to aBIU state is indicated, sQS uses the SABus to read the indicated state and then passes the information to the JBus. A read to Ctrl state is passed to Ctrl on the SCBus.

A state update, as indicated by the Update field in the address, is executed when the address phase completes. Updates to sBIU state are handled internally; updates to Ctrl or aBIU are passed along the SCBus or SABus respectively.

clSRAM Update Space

The clSRAM Update space generates an undefined sSRAM address, as no SRAM transfer actually occurs. However, when a clSRAM Update access is seen, sQS does generate a special request to sCI so that the update will be forwarded via the specified MemQOut queue to aBIU.

Immediate Command Space

The Immediate Command space has no bearing on the sSRAM or any of the interfaces provided by sQS; the generated sSRAM address is undefined.

3.3.2 State Access

All of the state within the sQS module is accessible for sPBus and aBIU Interface reads and updates; the addressing format used to access the states, called NESAddress, is described in Appendix A. The Ctrl State Interface access to sQS state is limited to write access to the short receive PPtr values, which are described in Section 3.3.3.

State Updates

The visible state in the sQS module can be updated by the sBIU (as a direct update from a QPtr Space or Config Access Space transaction, or as an indirect result of a ShRx or ShTx transfer), the aBIU (via the ASBus) or Ctrl (via the CSBus).

Updates by the aBIU and sBIU use the same access path to system state, since both units can access the full range of states specified by NESAddress (Appendix A). Since the aBIU expects an immediate response, it is given priority; the sBIU state update is buffered if a conflict arises. Each of these units can issue at most one sBIU state update every four cycles, which eliminates the possibility of buffer overflow.

Updates issued by Ctrl are always executed immediately, and are issued along a different data path than the updates described above. No resource conflict exists between Ctrl and BIU updates to different states; simultaneous updates to the same state are not expected, and the value written by Ctrl will be seen if such an event occurs.

State Reads

All sBIU state is available immediately to both the aBIU and the sBIU (for composing sSRAM addresses). Responses to aBIU read requests are generated by using the request address as the input to a large mux; state used by the sBIU is directly wired to the relevant address generation units, and can be muxed to the JBus interface for status reads by the sP.

3.3.3 sBIU Internal State

Aside from the registers which are exclusively used by the sBI and sCI state machines, all sBIU state resides in the sQS. This state is divided into three types: short receive queue state, short transmit queue state, and system register state.

Short Receive Queue State

There are five short receive queues accessible by the sBIU: VasR-0L, VasR-0H, VasR-1L, VasR-1H and MemQIn. Each of these queues is specified with CPtr, PPtr, Base, Bound and LateAck registers.

The CPtr indicates the position of the queue's consumer pointer; the PPtr indicates the producer pointer position. Together, these two pointers are used to determine if the queue is empty; also, CPtr is used to generate the sSRAM address for a ShRx read.

Base specifies the base sSRAM address for the given queue, and is used with CPtr to generate the full sSRAM address.

Bound limits the size of the queue; since these queues are circular, the CPtr is set to zero instead of incremented if it would pass Bound. Bound is only tested for internal incrementation of the pointers; direct updates are not tested against this value.

Finally, LateAck indicates that ShRx reads to the queue should generate an Arctic acknowledgment.

Short Transmit Queue State

There are six short transmit queues associated with the sBIU: PasT-0L, PasT-0H, PasT-1L, PasT-1H, MemQOut0 and MemQOut1. Each of these queues has a PPtr, Base, Bound and 32bCompose register associated with it.

PPtr associated with a given queue specifies the producer pointer; this value is used with Base to determine the sSRAM location where the next message should be written, and is incremented when the write is complete.

Base indicates the beginning of the given queue's sSRAM space, and is used with PPtr to generate the sSRAM address.

Bound indicates the size of the queue; PPtr is zeroed rather than incremented when it reaches Bound.

Finally, 32bCompose indicates that the queue in question supports 32-bit compose mode. This mode allows an 8B message to be specified via a single 4B write, with the address of the compose command sent via the JBus to form the other 4B. (This does not prohibit complete specification of messages via 8B write, however.)

System Register State

There are only two system registers used by the sBIU: OnePoll? and OnePollAddress. Section 3.3.1 describes the use of these registers.

Static Values

Several address spaces use static sSRAM locations rather than configuring sBIU state or encoding the sSRAM address within the sPBus address phase. aSRAMTmp is used as the temporary sSRAM location for data transferred to or from the aSRAM. OverflowStatus contains the status of the overflow queue, which is maintained by Ctrl. EmptyMsgAddress indicates the sSRAM address to be read when a ShRx read does not poll any non-empty queues. QConfigTmp is used to write BIU or Ctrl state to the

Field	Value	Description
[0]	X	** Unused **
[1:13]	X	aSRAM address
[14:15]	X	** Unused **
[16]	0	Transfer to sSRAM
	1	Transfer to aSRAM
[17:29]	X	sSRAM address
[30:31]	X	Transfer Size

Table 3.3: JBus Data Motion Format

sSRAM so it can be read by the sP. Finally, QPtrBase defines the region containing Ctrl-maintained status pointers corresponding to the various Comm Groups.

3.4 sCI Implementation

The sCI unit is responsible for coordinating the JBus interface, which the sBIU uses to request action on the IBus between NES, aSRAM, sSRAM and the TxU/RxU network interface module.

The logic in this unit is fairly straightforward. There are two buffers used: one for pending aSRAM access requests generated by the sBI submodule and one for Compose requests generated by the sQS module. All requests are initially placed in these buffers; additional buffer state indicate that the command should be delayed until the next address or data phase completes.

When a DataMotion command is ready, the sCI places the command on the JBusData interface according to Table 3.3 and asserts the DataMotionValid signal; these signals are held until Ctrl respond with DataMotionFree to indicate that the command has been received. At this point, the buffer is cleared.

A Compose request requires that the SRAM address generated by sQS be placed on the JBusAddress interface and the data (sPBusAddress in most cases; a hardwired 32b field for clSRAM updates) be placed onto JBusData. Again, the signals are held until the ComposeFree response is given, when the buffer is cleared.

The sCI uses a very simple method for resolving conflicting requests for the JBus: it tracks the last use of the JBus and awards priority to the type (DataMotion or Compose) which was not serviced most recently. There is no need for more than one buffer for each type, since the sCI broadcasts the buffer status and the other units generate retries when a conflict is detected.

3.5 Summary

The sBIU implementation is driven by the functions and interfaces described in Chapter 2. To provide this functionality, the sBIU is broken into three modules: a Bus Interface module which interacts with the sPBus, a Queue/State module which generates sSRAM addresses and maintains sBIU state, and a Ctrl Interface module which handles sBIU interaction with the JBus. Together, these modules provide all of the functionality required.

Chapter 4

aBIU Implementation

This chapter describes the functionality of the aBIU. First, an overview of the aBIU is given, describing the assignment of functionality to the various submodules within the aBIU and the interaction between these submodules. The submodules are then described in detail, including all finite state machines (FSMs), registers and logic used to provide the required functionality.

As mentioned, the aBIU and sBIU share many common functions, and the aBI, aQS and aCI submodules all are similar to their sBIU counterparts. The complete descriptions are provided here nonetheless in the interest of readability.

4.1 Overview

The aBIU is broken into four distinct submodules: the aBIU Bus Interface, or aBI; the aBIU Queue/State, or aQS; the aBIU Ctrl Interface, or aCI; and the aBIU Bus Master, or aBM. Figure 4-1 shows these submodules and the primary interfaces of each.

Because the aBIU serves as both slave and master on the aPBus, several tri-state buffers are used along the aPBus Interface.

One further general comment bears mentioning: all signals leading into or out of the aBIU are latched. Hence, input signals are "received" one cycle after they appear on the port, and output signals are "asserted" one cycle before they are actually seen

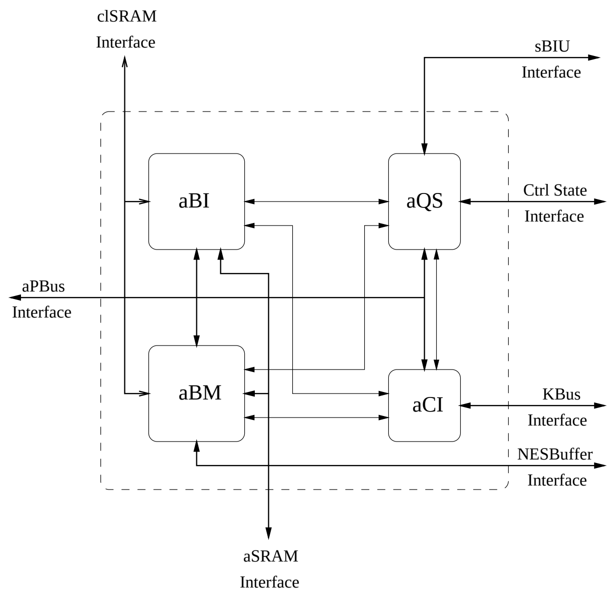

Figure 4-1: aBIU Overview

by the other end of the connection.

4.1.1 Submodule Descriptions

The aBI submodule is responsible for interacting with the aPBus; it asserts the control signals to respond to a aPBus transaction as specified by the 60X bus protocol, coordinates the timing of events within the aBIU, and controls the aSRAM. It shares the aPBus Interface (see Section 2.2.1) and aSRAM Interface (Section 2.2.1 with aBM; it is also responsible for decoding and forwarding Immediate Commands (Section 2.2.2) to Ctrl. Ctrl modules.

The aQS submodule is responsible for maintaining and manipulating the aBIU's queue pointers, Service Space and Snooped Space response tables, system registers and other global state; it also uses this state to generate the appropriate aSRAM address for each transaction. It operates the Ctrl State Interface (Section 2.2.1) and sBIU State Interface (Section 2.2.1) as well as the RxEmpty status bus. Finally, the aQS contains the Approval Register, and indicates the appropriate response (see Section 2.2.4) when a transfer requires approval.

The aCI submodule is responsible for using the KBus to allow the aBIU to reach the IBus. It operates the KBus Interface (Section 2.2.1).

The aBM submodule is used to respond to commands issued to aBIU by the NESBuffer Interface (Section 2.2.1). It can control the aPBus Interface and sSRAM Interface when directed to place a transaction on the aPBus; it also maintains DMA state, handles clSRAM updates and relays Approval Register updates to the aQS submodule.

4.1.2 Internal Signals

There are several signals internal to the aBIU which are used to coordinate activity between the submodules.

The aBIU generates several signals indicating the status of the current aPBus transfer. AddressRead and AddressWrite indicate the decoded type of the most current address phase. AddressConfirm indicates that the most recent address has been completed without retry, allowing aQS and aCI to proceed with irreversible state changes. DataMotionReq indicates that the current address is an sSRAM access and thus requires that aCI send a DataMotion command on the KBus at the appropriate time.

A primary function of aQS is the generation of SRAMAddress, the aSRAM address corresponding to the current address phase. ComposeReq, MemQInReadReq and MemQInWriteReq are used to request the specified Compose command to aCI. aQSRetry indicates a resource conflict within aQS, and causes the aBI to retry the current address transfer. LookupResponse indicates the response to a transaction in Serviced Space or Snooped Space. MemQInNext relays the next MemQIn address to aBM so that module can issue messages to that queue.

aCI generates ComposeAvail and DataMotionAvail, which indicate the current status of the Compose and DataMotion buffers. It also relays DataMotionDone to the aBI so that transfers waiting for a DataMotion command may proceed.

The aBMMasterAddress and aBMMasterData signals indicate that the aBM is mastering the address or data phase of the aPBus. clsUpdate indicates that a clSRAM

Signal Name	Size	Source	Destination	Comments
AddressRead	1b	aBI	aQS, aCI	Transfer = Write
AddressWrite	1b	aBI	aQS, aCI	Transfer = Read
AddressConfirm	1b	aBI	aQS, aCI	Transfer Address Done
DataMotionReq	1b	aBI	aCI	DataMotion Request
SRAMAddress	12b	aQS	aBI, aCI	sSRAM Address
ComposeReq	1b	aQS	aCI	Compose Request
MemQInReadReq	1b	aQS	aCI	MemQIn Read Compose
MemQInWriteReq	1b	aQS	aCI	MemQIn Write Compose
aQSRetry	1b	aQS	aBI	Retry Required
LookupResponse	2b	aQS	aBI	Serviced/Snooped Response
MemQInNext	12b	aQS	aBM	Next MemQIn Address
ComposeAvail	1b	aCI	aQS, aBM	Compose Available
DataMotionAvail	1b	aCI	aBI	DataMotion Available
DataMotionDone	1b	aCI	aBI	DataMotion Done
aBMMasterAddress	1b	aBM	aBI	aBM Mastering aPBus Address
aBMMasterData	1b	aBM	aBI	aBM Mastering aPBus Data
clsUpdate	1b	aBM	aQS	clSRAM Update In Progress
ApprUpdate	1b	aBM	aQS	Approval Register Update
ApprCommand	3b	aBM	aQS	Approval Register Command
ApprAddressNew	10b	aBM	aQS	New ApprSRAMAddress Value
aBMComposeReq	1b	aBM	aQS, aCI	aBM MemQIn Compose

Table 4.1: aBIU Internal Signals

Field	Value	Comments
[0:2]	000	ApprRegState = READY
	001	ApprRegState = LOCKED
	010	Reset DMARxDataQ CPtr
	011	ApprRegState = FREE
	100	No Operation
	101	Lock Notification
	110	Lock/Unlock aPBus
	111	Unlock Notification

Table 4.2: ApprCommand Encoding

update is in progress, which indicates that any transfer to Snooped Space must be retried. ApprUpdate indicates that a NESBuffer command related to the Approval Register; the specific command is specified in ApprCommand according to Table 4.2, and the new ApprSRAMAddress is given in ApprAddressNew. Finally, aBMComposeReq indicates that aBM needs to compose an acknowledgment in MemQIn through aCI.

4.2 aBI Implementation

The aBI submodule acts as a slave device on the aPBus according to the specifications of the 60X bus protocol [12] and the Union memory controller [9]. The finite state machine and associated logic which generate the sPBus and aSRAM signals compose the bulk of this logic.

4.2.1 Finite State Machine

The aPBus is a 60X split phase bus, with address and data transfers occurring separately; the two phases can overlap, or the data transfer can occur after the address transfer is complete.

The Union memory controller allows flexible pipelining of 60X bus transactions; the aBI implementation allows up to three outstanding address phase transactions to be pipelined. In other words, the address pipeline does not stall until there are three

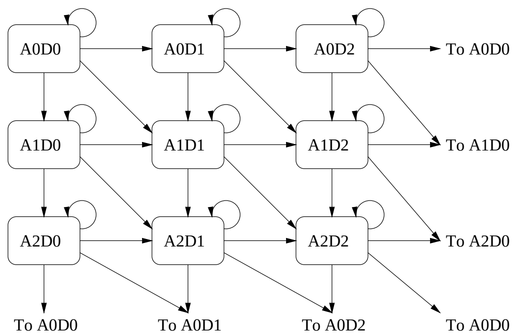

Figure 4-2: aBI Finite State Machine Overview

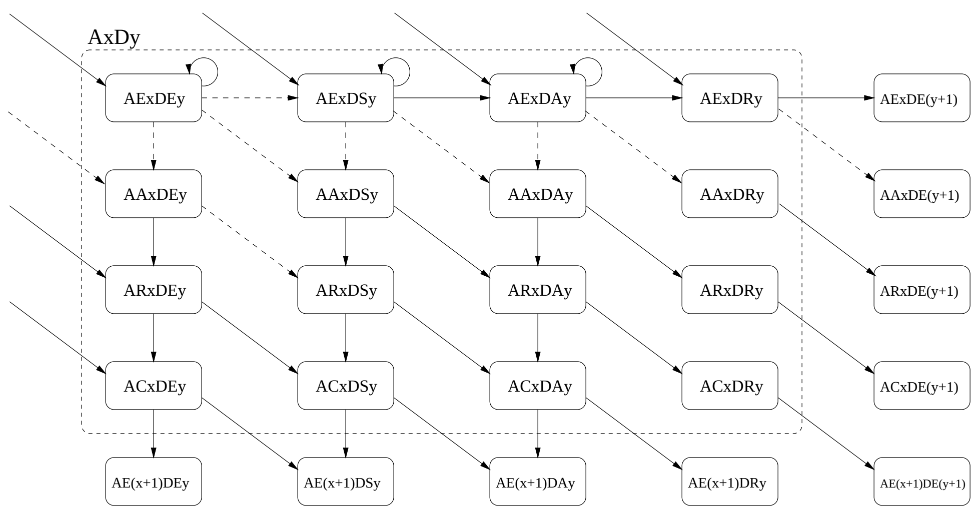

Figure 4-3: aBI Finite State Machine Detail

uncompleted data phases.

The 60X bus protocol and selected pipeline depth provide the motivation for the primary aBI finite state machine. This FSM is best described in several steps, due to the complexity involved in pipelining the address and data phases.

Figure 4-2 gives a simplified overview of the entire state machine; Figure 4-3 breaks one FSM state into several sub-states, each of which indicates a specified address and data phase as shown in Table 4.3.

Figure 4-2 suggests a circular queue implementation of the registers associated with the finite state machine; this is precisely the structure used. A pair of pointers

State	Address Phase Status	Data Phase Status
AExDEy	AddressEmpty	DataEmpty
AAxDEy	AddressActive	DataEmpty
ARxDEy	AddressRelease	DataEmpty
ACxDEy	AddressConfirm	DataEmpty
AExDSy	AddressEmpty	DataSetup
AAxDSy	AddressActive	DataSetup
ARxDSy	AddressRelease	DataSetup
ACxDSy	AddressConfirm	DataSetup
AExDAy	AddressEmpty	DataActive
AAxDAy	AddressActive	DataActive
ARxDAy	AddressRelease	DataActive
ACxDAy	AddressConfirm	DataActive
AExDRy	AddressEmpty	DataRelease
AAxDRy	AddressActive	DataRelease
ARxDRy	AddressRelease	DataRelease
ACxDRy	AddressConfirm	DataRelease

Table 4.3: aBI Finite State Machine States

is used to indicate the queue position of the current address and data phase, allowing a single data path to be used for each phase.

As mentioned above, the aBI is designed to allow up to three pipelined address phases; this restriction is seen in the dashed lines in Figure 4-3. The state AExDEx has two meanings: the pipeline is empty, or the pipeline is completely full; the next state is restricted to AAxDEx if the pipeline is empty and AExDSx if the pipeline is full. Similarly, the states AExDS(x-1), AExDA(x-1) and AExDR(x-1) indicate that the pipeline may be full or empty; transition to AAx can occur only in the empty case. Finally, the state AAxDEx can only be reached when the pipeline is empty, and thus does not allow a transition to ACxDSx.

4.2.2 Address Phase Events

During the address phase, the aBI reads, decodes and latches the address, type and size of the current transaction; assuming this transaction is to a abiu-serviced address space, the aBI generates the appropriate control signals to complete the address phase of the transfer.

AddressEmpty indicates that there is no address phase currently in progress; hence, no control signals are asserted. When an address start is detected, a sub-state transition to AddressActive occurs.

AddressActive is the state in which the aPBusAddress and related signals are decoded. The aPBusAddressAck signal is generated during this phase (if the transfer is to aBIU-serviced space). If DataMotion between the aSRAM and aSRAM is required, a request is sent to the aCI submodule; if an Immediate Command is indicated, the appropriate signal is sent to the Ctrl. Also, this phase is used to detect and latch any retry conditions within aBIU. Transition to AddressRelease is automatic.

AddressRelease is used to place a retry request on the aPBus bus when required (due to resource conflict or a specified Snooped/Serviced Space response) and to deassert all other aPBus control signals. Also, aSRAM address and control signals, and any ordering constraints on the data phase of the transfer, are generated (received from aQS for aSRAM address) and latched into the appropriate registers during this phase. Transition to AddressConfirm is automatic.

The *AddressConfirm* state is used to indicate to other aBIU units that the address phase has completed without being retried, which allows those units to finalize state updates. This is the last sub-state in a given FSM state; completion of this state increments the address counter and returns to the AddressEmpty sub-state.

4.2.3 Data Phase Events

The primary requirement of the data phase portion of the finite state machine is the transfer of data between the aPBus and the aSRAM.

DataEmpty indicates that no data phase is in progress. No signals are asserted in this state, and transition to DataSetup (if the data phase is a read and/or must be stalled) or DataActive (no stall, write) occurs when the beginning of a new data phase is detected.

DataSetup is used to drive aSRAM address and control signals to prepare for a read, and to stall the data phase until the address phase is "confirmed" and any required JBus or SCBus commands are completed. Transition to DataActive occurs when no wait conditions exist.

DataActive is the state in which data is read from or written to the aSRAM. The aSRAM control signals are used to process the transfer of a single beat of data; in addition, aPBus signals are asserted if the aBI is acting as a slave device (rather than snooping the bus). This state transitions after one (non-burst transfer) or four (burst transfer) cycles to DataRelease; when this transition occurs

DataRelease is used to deassert all data phase sPBus and aSRAM signals. This state indicates a master FSM state transition my advancing the data counter and returns to the DataEmpty sub-state.

4.3 aQS Implementation

The aQS submodule is designed to generate aSRAM addresses based on the aPBusAddress and aPBusTransferType field, to determine the response to transactions to Serviced or Snooped Space, and to maintain the aBIU queue pointer and system register state. These functions are related, and combined in a single submodule, because the aSRAM address generation and Serviced/Snooped Space responses are dependent on the queue pointer values.

4.3.1 aPBus Address Response

The aQS has several responses to an aPBus address phase. The aSRAM address which is used by aBI in responding to transfers on the aPBus is generated in aQS based on the address and type of the aPBus transfer; in the case of Snooped Space or Serviced Space transfers, the response lookup is also provided by aQS.

Additionally, read or update requests to aBIU, aBIU and Ctrl state can be generated as the aPBusAddress is decoded in the aQS. Read requests are issued upon reception of the aPBusTransferStart, while write requests are issued when the address phase is confirmed by aBI. Further details are given in Section 4.3.2.

Address generation for the various aPBus address spaces is done in parallel; the appropriate address for the actual aPBusAddress is selected by a large mux.

Readers are directed to Section 2.2.2 for more details about the encodings and uses of the various address spaces.

SRAM Space

Generation of the SRAM address for SRAM Space is straightforward. If bit [16] of the aPBusAddress is asserted, the fully-specified aSRAM address from aPBusAddress is used. If bit [16] is deasserted, the access is to the sSRAM; in this case, the aSRAM address is to a hardwired location called sSRAMTmp.

QPtr Space

The SRAM address for QPtr Space consists of a fixed base address named QPtrTmp and an offset which consists of bits [7:9] (the Comm Group) of the aPBusAddress; NES Ctrl maintains status information for the Comm Groups at these locations.

Additionally, if the Update field in aPBusAddress is asserted, an update request is placed in the ACBus buffer for processing once the address phase is confirmed. If this buffer is occupied, aBI is notified so that the current transfer can be retried.

ShTx Space

The SRAM address for ShTx (Short Transmit) Space address is composed from the concatenation of the Base and PPtr (Producer Pointer) state for the given PasT (short transmit) queue. If the 32bCompose register is set for this queue, and the current transfer is a 4B write, aQS also requests that the aPBusAddress be written to the even word of the aSRAM address via the KBus Interface. The aBIU PPtr is incremented (or zeroed if it matches the Bound register for the queue) at the conclusion of the address phase; the command to update Ctrl PPtr is issued on the ACBus when the data portion of the transfer completes (the message must be present in the SRAM before the Ctrl pointer is updated).

See Section 4.3.3 for a description of the state associated with the short transmit queues.

ShRx Space

The ShRx (Short Receive) Space format uses a bit vector (with static priorities) to determine which queue is to be polled; hence, the first step in generating the SRAM address and related actions is determining which queue is actually to be read. This is done by decoding the address to form a full prioritized bit vector of the queues to be polled, ANDing this vector with a vector of the non-empty queues (as determined by internal queue state and the RxEmpty signal from Ctrl) and then using a priority encoder to determine the actual queue being accessed.

If the queue being read is a VasR (short receive) queue, the address is generated by concatenating the Base and CPtr (Consumer Pointer) registers for the selected queue. Also, the aBIU and Ctrl CPtr values are incremented (or zeroed if the CPtr matches the Bound value for the queue) at the conclusion of the address phase of the transfer. Finally, if the LateAck register for the queue is set, a late acknowledgment is sent to the Arctic network via Ctrl.

On the other hand, if the queue indicated is a PcpR (medium receive) queue, the SRAM address generated is a static Continuation Address corresponding to the particular queue. No state updates occur for this case.

One exception to the above process exists: if the transfer is a 4B read, a special OnePoll? register is tested. If the register returns a positive value, the SRAM address is taken from the OnePollAddress register, the OnePoll? register is cleared and no state update occurs; if the OnePoll? register returns a negative value, the SRAM address is generated as above and placed into the OnePollAddress register (as well as being provided to the aBI as usual) and the OnePoll? register is set. This special mechanism is used to support interruptible 4B queue reads, since the OnePoll? and OnePollAddress registers are sP accessible system registers.

Again, Section 4.3.3 describes the queue status registers in more detail.

Config Access Space

The response to transfers in the Config Access Space depends on the state being accessed by the instruction (the encoding can be found in Section 2.1.2).

aSRAM address generation depends only on the state being accessed. If the access is to aBIU state, sBIU state or Ctrl QRAM or SysReg state, the aSRAM address is a special static location called QConfigTmp; if the access is to Ctrl EPRAM state, the aSRAM address is generated from QPtrBase and the Comm Group of the access as in QPtr Space; and if the access is to the Overflow group, the aSRAM address is a static OverflowStatus location.

Additional actions depend on the transfer type of the aPBus transaction and the Update field in the aPBusAddress. If the transfer type indicates a read and the state referenced is in the aBIU, aQS arranges for the state to be written to the QConfigTmp location in aSRAM via the KBus when the address phase completes. If a read to sBIU state is indicated, aQS uses the ASBus to read the indicated state and then passes the information to the KBus. A read to Ctrl state is passed to Ctrl on the ACBus.

A state update, as indicated by the Update field in the aPBusAddress, is executed when the address phase completes. Updates to aBIU state are handled internally; updates to Ctrl or sBIU are passed along the ACBus or ASBus respectively.

Immediate Command Space

The Immediate Command space has no bearing on the aSRAM or any of the interfaces provided by aQS; the generated aSRAM address is undefined.

Serviced Space

The aQS response to Serviced Space commands is determined by the two-bit SSResponse to the particular aPBusTransferType and the status of the Approval Register.

If SSResponse = IGNORE, the aSRAM address generated is the static MissPattern address (although no data is actually written in a write), and no other action is taken by aQS.

If SSResponse = NOTIFY, the transfer is a notify only, the aSRAM address is either MissPattern (for a read) or the MemQDataIn pointer (for a write). In either case, aQS requests a MemQIn message compose of the specified type (read/write) through the aCI, and updates the MemQIn PPtr in both the aBIU and sBIU (via the ASBus).

If SSResponse = APPROVE, the exact response depends on the contents of ApprRegState, ApprRegAddr and ApprSRAMAddr. If ApprRegState = FREE, indicating that there is no pending approval, the current transaction is retried, aPBusAddress is placed into ApprRegAddr, ApprRegState is set to PENDING, an approval request is placed into MemQIn and the MemQIn PPtr is incremented. If ApprRegState = PENDING or LOCKED, or if ApprRegState = READY and ApprRegAddr != aPBusAddress, the current transfer is retried and no further action is taken. If ApprRegState = READY and ApprRegAddr = aPBusAddress, approval has been granted: the transfer completes using the ApprSRAMAddr as the target aSRAM address and ApprRegState is reset to FREE.

If SSResponse = RETRY, the transaction is retried with no further action by aQS.

Snooped Space

aQS response to Snooped Space is very similar to the response to Serviced Space. However, the HALResponse table is indexed by the clSRAMData for the cache line of the current transfer as well as the aPBusTransferType. All other responses are identical from the aQS point of view.

4.3.2 State Access

All of the state within the aQS module is accessible for aPBus and sBIU Interface reads and updates; the addressing format used to access the states, called NESAddress, is described in Appendix A. The Ctrl State Interface access to aQS state is limited to write access to the short receive PPtr values, which are described in

Section 4.3.3.

State Updates

The visible state in the aQS module can be updated by the aBIU (as a direct update from a QPtr Space or Config Access Space transaction, or as an indirect result of a ShRx or ShTx transfer), the sBIU (via the SABus) or Ctrl (via the CABus).

Updates by the aBIU and sBIU use the same access path to system state, since both units can access the full range of states specified by NESAddress (Appendix A). Since the sBIU expects an immediate response, it is given priority; the aBIU state update is buffered if a conflict arises. Each of these units can issue at most one aBIU state update every four cycles, which eliminates the possibility of buffer overflow.

Updates issued by Ctrl are always executed immediately, and are issued along a different data path than the updates described above. No resource conflict exists between Ctrl and BIU updates to different states; simultaneous updates to the same state are not expected, and the value written by Ctrl will be seen if such an event occurs.

Finally, NotifyLock and the approval register states can be directly accessed by NESBuffer commands and/or Immediate Commands relayed by the sBIU; all of these are executed immediately and take priority over other updates.

State Reads

All aBIU state is available immediately to both the aBIU and the aBIU. Responses to sBIU read requests are generated by using the request NESAddress to select the required data; within the aBIU, state is directly wired to the relevant address generation units and can be muxed to the KBus interface for status reads by the aP.

4.3.3 aBIU Internal State

Aside from the registers which are exclusively used by the aBI and aCI state machines, all aBIU state resides in the aQS. This state is divided into four types: short receive queue state, short transmit queue state, Serviced Space and Snooped Space response tables and system register state.

Short Receive Queue State

There are four short receive queues accessible by the aBIU: VasR-2L, VasR-2H, VasR-3L and VasR-3H. Each of these queues is specified with CPtr, PPtr, Base, Bound and LateAck registers. The DMARxDataQ uses the same format, but is used to provide data for DMARx-Mastered operations; see Section 4.5.

The CPtr indicates the position of the queue's consumer pointer; the PPtr indicates the producer pointer position. Together, these two pointers are used to determine if the queue is empty; also, CPtr is used to generate the aSRAM address for a ShRx read.

Base specifies the base aSRAM address for the given queue, and is used with CPtr to generate the full aSRAM address.

Bound limits the size of the queue; since these queues are circular, the CPtr is set to zero instead of incremented if it would pass Bound. (PPtr is always explicitly set rather than incremented, so Bound is unused.)

Finally, LateAck indicates that reads to the queue should generate an Arctic acknowledgment.

Short Transmit Queue State

There are five short transmit queues associated with the aBIU: PasT-0L, PasT-0H, PasT-1L, PasT-1H and MemQIn. Each of these queues has a PPtr, Base, Bound and 32bCompose register associated with it. The MemQDataIn queue shares most of this format as well, although it is used to store data for captured transactions rather than as a message queue.

PPtr associated with a given queue specifies the producer pointer; this value is used with Base to determine the aSRAM location where the next message should be written, and is incremented when the write is complete.

Base indicates the beginning of the given queue's aSRAM space, and is used with

PPtr to generate the aSRAM address.

Bound indicates the size of the queue; PPtr is zeroed rather than incremented when it reaches Bound.

Finally, 32bCompose indicates that the queue in question supports 32-bit compose mode. This mode allows an 8B message to be specified via a single 4B write, with the address of the compose command sent via the KBus to form the other 4B. (This does not prohibit complete specification of messages via 8B write, however.)

Serviced/Snooped Space Response Tables

The Serviced Space Response table (SSResponse) is an 11x2b table; the response (IGNORE, NOTIFY, APPROVE, RETRY) associated with each of the supported transfer types for the Serviced Space address region is obtained by using the aPBus-TransferType as the table index.

The Snooped Space Response table (HALResponse) is an (11x8)x2b table containing the same information as SSResponse. The transfer type of the transaction and the 3-bit clSRAM response to the transaction address form the index to this table.

System Register State

The OnePoll? and OnePollAddress system registers are used to handle 4B reads to ShRx space; Section 4.3.1 describes the use of these registers.

The aPBusLock register indicates that the aPBus is to be locked; if this register is set, the aBIU will retry all transfers on the aPBus (not only those normally serviced by the aBIU).

The NotifyLock register is used to lock the notification resource. If this is set, any Serviced Space or Snooped Space transactions which attempt to notify the sP through MemQIn are retried instead.

ApprRegState, ApprRegAddr and ApprSRAMAddr are used to implement a simple approval mechanism for specified transactions to Serviced or Snooped Space. ApprRegState can be either FREE, indicating that the register is not in use; PENDING, indicating that an outstanding approval request exists; READY, indicating that approval has been granted; or LOCKED, indicating that the approval mechanism is not active. ApprRegAddr stores the address of a PENDING or READY approval request; this is matched against the address of future transactions to complete the approval. ApprSRAMAddr specifies the aSRAM address for an approved transaction.

Static Values

Several address spaces use static aSRAM locations rather than configuring aBIU state or encoding the aSRAM address within the aPBus address phase. aSRAMTmp is used as the temporary aSRAM location for data transferred to or from the aSRAM. OverflowStatus contains the status of the overflow queue, which is maintained by Ctrl. EmptyMsgAddress indicates the aSRAM address to be read when a ShRx read does not poll any non-empty queues. QConfigTmp is used to write BIU or Ctrl state to the aSRAM so it can be read by the sP. Finally, QPtrBase defines the region containing Ctrl-maintained status pointers corresponding to the various Comm Groups.

4.4 aCI Implementation

The aCI unit is responsible for coordinating the KBus interface, which the aBIU uses to request action on the IBus (which connects the NES, aSRAM, aSRAM and TxU/RxU network interface).

The logic in this unit is fairly straightforward. There are two buffers used: one for pending DataMotion requests generated by the aBI submodule and one for Compose requests generated by the aQS module. All requests are initially placed in these buffers; additional buffer state indicate that the command should be delayed until the next address or data phase completes.

When a DataMotion command is ready, the aCI places the command on the KBusData interface according to Table 3.3 and asserts the DataMotionValid signal; these signals are held until Ctrl responds with DataMotionFree to indicate that the command has been received. At this point, the buffer is cleared.

A general Compose request from aQS requires that the SRAM address generated by aQS be placed in KBusAddress and the aPBusAddress be placed onto KBusData. MemQIn Compose requests use the MemQIn aSRAM address instead of the general aQS-generated address. In either case, all signals are held until ComposeFree is received.

Compose requests from aBM use the same interface with the KBus as general Compose requests. Data provided by aBM is used as KBusData and MemQIn is used as KBusAddress.

The aCI uses a very simple method for resolving conflicting requests for the KBus: it tracks the last use of the KBus and awards priority to the type (DataMotion or Compose) which was not serviced most recently. There is no need for more than one buffer for each type, since the aCI broadcasts the buffer status and the other units generate retries when a conflict is detected.

4.5 aBM Implementation

The aBM module is used to respond to commands issued over the NESBuffer Interface, including mastering the aPBus, configuring DMA State, coordinating clSRAM Updates and relaying Approval Register related commands to aQS. A full description of the available commands is found in Section 2.2.3.

The state machine which controls the aBM, shown in Figure 4-4, is similar to that found in the sBI (Figure 3-2). Unlike the aBI, the aBM is not designed to take advantage of the Union's pipelining capabilities in order to simplify the design of the module.

Functions which do not require mastery of the aPBus, such as setting up the DMA (direct memory access) state and forwarding Approval Register commands to aQS, are relatively simple; these are executed immediately and serve to short-circuit the normal state machine execution.

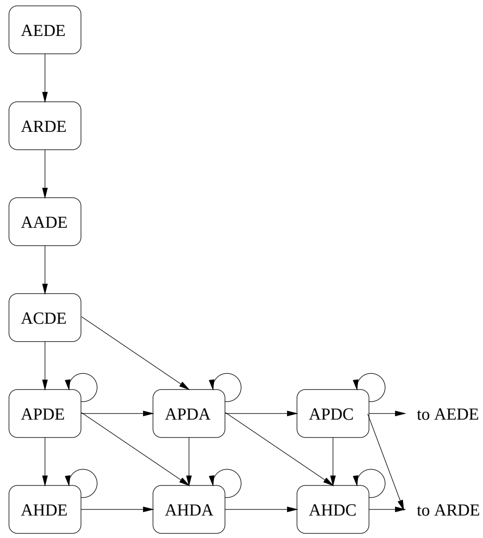

Figure 4-4: aBM Finite State Machine

4.5.1 Address Phase Events

During the address phase, the aBM obtains mastery of the aPBus and places the specified transaction on the address portion of the bus, asserting and responding to the appropriate 60X signals. Also, NESBuffer commands which do not actually involve the aPBus are executed during the "address" phase.

AddressEmpty indicates that there is no address phase currently in progress. No signals are asserted in this state; transition to AddressRequest occurs when a new command is received on the NESBuffer Interface.

AddressRequest is the state in which the aBM sends a request for the address bus to Union. It is also the state in which non-aPBus commands are executed. For bus commands, transition is to AddressActive when the bus grant is given; for non-bus commands, transition is to AddressEmpty (and DataComplete) and occurs immediately.

AddressActive is the state in which the aPBusAddress and related signals are asserted. This state, and all associated signals, are held until aPBusAddressAck is received, at which time the FSM transitions to Address Confirm.

AddressConfirm is used to deasserted the aPBusAddress and all other 60X signals, and to observe the aPBusAddressRetry signal. If a retry is observed, the address phase returns to Address Request and the data phase is cancelled; otherwise, the address phase transitions to AddressPending to indicate that the data phase is in progress.

AddressPending indicates that the data phase corresponding to the last address has not yet completed. No control signals are asserted in this state. Transition occurs when a new command is detected and/or the data phase completes, and is to AddressHold (new command), AddressEmpty (data completion) or AddressRequest (both).

AddressHold indicates that a new address phase has begun while a data phase is in progress; this state is used to stall the address phase until the current or pending data phase completes, at which point the state transitions to AddressRequest. Again, no control signals are generated at this time.

4.5.2 Data Phase Events

The data phase includes the actual transfer of data, the updating of aBM state and the completion of the current command. The aSRAM used for data transfer depends on the actual NESBuffer command type: NES-Mastered operations include the target aSRAM address within the command encoding, while DMARx-Mastered operations use a special DMARxDataQ to generate the aSRAM address.

One general point of note: the data phase immediately transitions to DataEmpty if aPBusAddressRetry is detected.

DataEmpty indicates that no data transfer is in progress. No signals are asserted, and transition occurs to DataActive if a data bus grant is seen or DataComplete if the NESBufferOp decoding indicates a non-bus command.

Data Active is the state in which the data is actually transferred. The aBM asserted the aSRAM control and address signals and monitors aPBusTransferAck; when the data transfer is complete (after one or four beat acknowledgments), the state transitions to Data Complete. This state will never be reached for a non-bus

NESBuffer command.

Data Complete is used to deassert the various 60X data control signals. It is also used to compose an acknowledgment to MemQIn, and to update MemQIn PPtr, DMARxDataQ CPtr, and the DMAPending count if necessary.

4.5.3 DMA State

Each DMA channel contains a separate count of pending operations, DMAPending. DMAPending is initialized by a DMA Receive command (see Section 2.2.3) and decremented at each successful DMARx-Mastered operation.

The DMARxDataQ, which is used to provide data for DMA operations, is described in Section 4.3.3.

4.5.4 Summary

The aBIU implementation is driven by the functions and interfaces described in Chapter 2. To provide this functionality, the aBIU is broken into four modules: a Bus Interface module which acts as a slave or snoop device on the aPBus, a Queue/State module which generates aSRAM addresses and maintains sBIU state, a Ctrl Interface module which handles aBIU interaction with the KBus; and a Bus Master module which responds to NESBuffer commands sent by Ctrl and acts as a master on the aPBus. Together, these modules provide all of the functionality required.

Chapter 5

Verification

This chapter describes the methods used to verify that the implementation detailed in Chapters 3 and 4 was properly coded in the Verilog logic language.

Two chief methods of testing have been used: manual testing of the internal BIU signals and automated testing of the complete NES response to transactions on the aPBus or sPBus. In addition, as the project advances, we expect to make use of testing facilities provided by IBM.

5.1 Manual Testing

Manual testing of the BIU submodules, and the BIUs as a whole, occurred in parallel with the Verilog coding of the implementation. The primary goal of this portion of the verification process was to detect bugs in the implementation code, and in the finite state machines in particular.

Each submodule was independently tested with a manually adjusted Verilog script controlling the input vectors; internal state and outputs were observed graphically with the Undertow utility.

5.2 Automated Testing

Automated testing of the entire NES Subsystem has been used to verify that the NES Chip accurately performs its various functions. The primary goal of this testing method is the detection and correction of design errors; additionally, the ability to easily generate multiple tests allows a second round of bug detection. In order to accomplish these goals, a complete StarT-Voyager site model has been created and a series of required tests have been generated.

5.2.1 StarT-Voyager System Model

In order to simulate the entire StarT-Voyager site in Verilog, a complete Verilog site has been created which contains functional models of the all hardware components (as seen in Figures 1-1 and 1-2).

The sSRAM, aSRAM, clSRAM and clFIFO Verilog models were composed from component libraries as required; these models should be completely accurate.

The MPC105 and Union models are simplified to meet the minimal functionalities required for testing. The MPC105 model was generated almost directly from the sBI submodule; the finite state machine was modified to accurately depict the memory controller's responses, while the SRAM interface was unchanged to allow the SRAM model to serve as a pseudo-DRAM for testing purposes.

The Union model is essentially designed with the aBI at its core, surrounded by the bus arbitration machine. Again, the finite state machine was changed to correspond to Union's interaction with the aBIU and the aPBus, while the SRAM interface is again used to emulate aP DRAM. The bus arbitration machine is not exact; however, it is expected to be sufficiently close to test the various aBIU functions with the aid of IBM's testing environment (described below).

Provided bus models of the PowerPC 604 are used for the aP and sP. These bus models are used to execute a specified list of bus transactions, which is the primary testing mechanism of the automated testing method.

To execute tests, a list of bus instructions and supporting commands is generated.

Through a sequence of utilities, this instruction sequence is compiled with the base Verilog object files into an executable which encodes the specified instructions. When the executable is run on the system model (including the aP and sP 604 bus models), the bus transactions are placed on the aPBus or sPBus at the appropriate time and allowed to complete as 60X protocol dictates.

5.2.2 Test Groups

Instruction sequences have been written to test the full range of BIU functionality. These tests are divided into several groups: SRAM Access, NES State Access, Message Passing, clSRAM Update, NESBuffer Commands, aP Serviced Space, aP Snooped Space, NES-Mastered Transactions and DMA Transactions.

SRAM Access

The SRAM Access test group includes 4B, 8B and 32B reads by the aP and sP to both aSRAM and sSRAM space. SRAM and aPBus/sPBus activity are monitored to confirm the tests.

NES State Access

This test group uses transactions to QPtr Space and Config Access Space to directly update and/or read internal NES state. The internal state is directly observed to confirm updates, while the SRAM and aPBus/sPBus activity are used to confirm reads.

Message Passing

This test group involves setting up the BIU and Ctrl queue pointer state and then sending and receiving a message through the ShTx and ShRx Spaces. The internal NES state, SRAM/bus activity and IBus activity are all monitored to confirm proper message handling.

clSRAM Updates

The use of the sP clSRAM Update Space is tested in this group to confirm that the required data is stored in the clFIFO and the update request is passed to aBIU through MemQOut and the NESBuffer Interface.

NESBuffer Commands

This address space tests the processing of non-bus NESBuffer commands: Approval Register and related manipulation, DMAChannel initialization and clSRAM updates. Internal aBIU state is used to confirm these tests.

aP Serviced Space

This group of tests confirms the proper processing of transfers to aP Serviced Space. The SSResponse and Approval Register mechanisms are initialized and the response on the aPBus to a transfer to aP Serviced Space is observed to confirm this functionality.

aP Snooped Space

The response to transfers to aP Snooped Space are tested in this group. The HAL-Response, Approval Register and clSRAM are all initialized to particular states, and then a transfer to Snooped Space is placed on the aPBus. The response to the transfer as observed on the aPBus confirms this functionality.

NES-Mastered Transactions

To test this functionality, an NES-Mastered Transaction is sent to aBIU through a sP compose to a MemQOut queue. The aBIU is expected to request control of the aPBus and issue the specified transfer, so the aPBus signals are used to confirm this functionality.

DMA Commands

To test the full DMA engine, NESBuffer commands are arranged to set up a DMAChannel and issue a DMARx-Mastered operation. The observed activity on the aPBus, the internal activity in aBM and the generation of a confirmation signal when the DMAChannel is empty are all observed.

5.3 IBM Verification

As noted above, we do not expect our Union bus model to be complete; we are also using simplified PowerPC 604 bus models as our sP and aP models, as no better or faster models are available to us.

In order to confirm that our design actually functions as expected with real Union and 604s in place, our agreement with IBM includes aid in verifying our design. The use of complete models of the Union and aP/sP will be essential in confirming the proper functionality of the NES in general and the sBI, aBI and aBM modules in particular.

5.4 Current Status

Manual testing of all BIU components has been completed; with the completion of the system model and increasing complexity of test cases, this testing method is not expected to be used even if changes to the BIUs are made.

The automated process is still in progress. Of the test cases listed in Section 5.2.2, all but NES-Mastered Transactions and DMA Commands have been verified in basic cases, and exhaustive tests of all possible cases are being prepared. Future test groups involving possible conflicts between transactions will also be created as necessary.

A basic methodology for preparing and testing StarT-Voyager Verilog code in the IBM environment has been established; actual testing within this environment will proceed over the next few months.

Chapter 6

Conclusions

This chapter concludes the thesis by presenting the work yet to done involving the BIUs and the StarT-Voyager project and by presenting the author's thoughts on the work done to produce this thesis.

6.1 Future Work

There are several steps that need to be completed before the NES Chip can be fabricated: verification, synthesis and vendor interaction.

The verification process, as described in Chapter 5, is in progress. Confirmation of the basic NES Chip functionality is nearing completion, but there is still work to be done in the development and execution of complex test cases in which resource conflict or other exceptional situations occur. Obviously, this process includes the correction of any detected errors in BIU implementation.

Synthesis is the conversion of the RTL (Register Transfer Level) Verilog implementation to gate level Verilog and evaluation of the generated gate level logic. This process, which will utilize the Synopsis synthesis and evaluation tools, has three purposes: to identify improperly formed structures in the RTL implementation, to locate critical timing paths in the implementation and estimate the delays along these paths, and to compile the RTL code into a format suitable for fabrication. In addition, the gate level code produced in this step can be run in the same verification environment as the RTL code, allowing the confirmation of accurate synthesis.

Finally, taking the NES Chip from design to hardware will require considerable interaction with an ASIC vendor. Once the verification and synthesis of the NES Chip are completed, the gate level code will be translated into vendor-specific format (if necessary) and sent the the vendor, where it will be laid out according to the vendor's design rules. This layout will be subject to further verification, and the more accurate delay estimations will be used to analyze the timing of the entire chip.

It is hoped that verification and synthesis can be completed by the end of the summer, with the final product produced near that time.

6.2 Architecture Comments

The work described in this thesis has led to a number of thoughts about the design of the StarT-Voyager parallel computer. This section describes some of the advantages and disadvantages of the actual design, compares Voyager to a radically different design (Stanford Flash), and briefly suggests a possible method of designing a hybrid design.

6.2.1 StarT-Voyager

The chief advantages of the StarT-Voyager design are the flexibility and reduced design time; the tradeoffs include some inefficiency in on-site communications.

Flexibility

StarT-Voyager is designed to provide exceptionally flexible message-passing and shared memory support. The primary source of this flexibility is the interaction between the NES Chip and the sP.

The NES Chip provides fixed hardware support of the simplest messages and configurable support for other message types. It also allows for configurable responses for responses to the Snooped Space and Serviced Space address spaces on the aP bus.

Many of these responses are executed in hardware, while others involve direct software (sP) intervention.

The service processor can be used to implement a wide variety of message-passing and shared memory protocols. With its own DRAM and memory controller, and access to the aPBus through the NES Chip, it provides an additional degree of flexibility beyond that provided by the configurable hardware support.

Design Effort

The use of commercial components has greatly reduced the design effort of the StarT-Voyager custom logic. The greatest reduction in design time comes from the use of commercial memory controllers. Further reduction is in fact obtained from the sP; obtaining even reduced flexibility without a second processor would require a considerable amount of additional hardware design.

Efficiency

While the use of commercial components and a distributed design does reduce the design time of the StarT-Voyager parallel computer, it also reduces the efficiency of the NES subsystem. In particular, the latency of commands on the 60X busses increases due to the delays in BIU interaction with the memory controllers and SRAMs; competition for the IBus can also cause delays.

6.2.2 Flash

As noted in Chapter 1, there have been recent efforts to design parallel computers which support both message-passing and distributed shared memory. These designs differ from that used in StarT-Voyager to various degrees.

Of these, the Stanford Flash project stands out. While attempting to meet the same goals as the StarT-Voyager project, Flash employs a vastly different design; the site consists solely of a commercial processor, local DRAM and a single custom logic chip called MAGIC. MAGIC responds to all traffic on the processor bus and network, and acts as a local DRAM controller as well.

The use of a single device reduces the latency of the bus and network interfaces, thus increasing the overall efficiency; however, it also reduces the flexibility and greatly increases the design effort required.

(The details of the Flash design are beyond the scope of this thesis; readers are directed to [7] for more information.)

Flexibility

Because the design is implemented entirely in hardware, the Flash design is somewhat limited in flexibility. It does provide for a variety of message-passing types and supports global shared memory, but the protocols used to support these communications methods are limited in comparison to those available in the StarT-Voyager design.

Design Effort

The wide range of functions included in the MAGIC chip dramatically increases the design effort required to produce the MAGIC chip and hence the Flash parallel computer. Incorporating DRAM control and some of the sP functionality into a single chip is a considerably more difficult task than that posed by the implementation of the StarT-Voyager NES Core.

Efficiency

The single largest advantage of the Flash design is efficiency. By incorporating the entire design, including memory controller and network interface, onto a single chip, MAGIC avoids some of the communication overheads seen in StarT-Voyager.

In addition, the use of a fully custom design allows the bus arbitration and memory controller to be optimized to meet the demands of the other functional units in the MAGIC chip.

6.2.3 Voyager/Flash Extensions

Since the advantages of the Voyager and Flash designs seem to be completely complimentary, it seems natural to consider the possibility of combining the positive aspects of each: the flexibility and low design effort of the Voyager site and the enhanced efficiency of the Flash MAGIC chip.

One potential hybrid might be a slight modification of the StarT-Voyager design: the incorporation of the aBI and aBM modules and memory controller functionality into a separate component. This combination should eliminate Voyager's inefficient use of the aPBus bus without affecting its flexibility (since the remainder of the NES subsystem could be relatively unchanged); the generation of a DRAM controller would increase the design cost of the system, however, since work would be done from scratch.

6.3 BIU Similarities

As has been frequently noted, the sBIU and aBIU are similar in many regards; hence, it was natural to attempt to reduce the design effort required by reusing much of the Verilog code used to produce the modules.

The original decision was to implement the simpler sBIU module and then use that module as the basis for the aBIU. This worked well for the aQS and aCI modules, but the aBI module demonstrated the limitation of this idea: inserting new states and signals into an existing FSM proved difficult. Implementation of the aBM from the sBI proved even more difficult; eventually, the base was scrapped and the model written from scratch.

It seems that it would have been far better to begin with the more complex module, the aBIU, and then simplify that as required to meet the sBIU functionality.

6.4 Concluding Remarks

This thesis has described the design and implementation of a substantial portion of the StarT-Voyager Network Endpoint Subsystem. The aBIU and sBIU modules interact with the two processors on a Voyager site, processing or relaying commands and data as required.

Although the verification and synthesis of these modules is not yet complete, the work completed for this thesis represents a significant step towards the fabrication of the NES Core and the ultimate goal of the project, the construction of a physical StarT-Voyager parallel machine.

Appendix A

NESAddress

The NESAddress is an internal addressing scheme used to access all available NES Core state. Each NESAddress corresponds to up to 7 bits of data. The basic encoding is indicated in Table A.1.

This section describes only the NESAddress space corresponding to BIU State; Ctrl State NESAddress encodings can be found in Section 4.1 of the Hardware Engineering Specification [1]. (Note: CtrlAddr == NESAddress[1:11].)

A.1 Queue State

Table A.2 describes the encoding of BIU queue state. Note that bit [4] of the NESAddress indicates the location of the state; a value of 0 indicates state contained in the sBIU, while a value of 1 indicates state contained in the aBIU.

Field	Value	Comments
[0]	0	Ctrl State
	1	BIU State
[1:2]	00	End-Point State (Ctrl Only)
	01	Queue State
	10	System Registers
[3:11]	X	(Meaning Depends on State)

Table A.1: NESAddress: Basic Encoding

Field	Value	Comments
[0:2]	101	BIU Queue State
[3:5]	000	PasT/VasR-0 Comm Group
	001	PasT/VasR-1 Comm Group
	010	PasT/VasR-2 Comm Group
	011	PasT/VasR-3 Comm Group
	100	MemQOut Comm Group
	101	** Reserved **
	110	MemQIn Comm Group
	111	DMARxDataQ
[5]	0	** Fixed Field **
[6]	0	Tx Queue (PasT)
	1	Rx Queue (VasR)
[7]	0	Low Priority / MemQOut0 / MemQIn
	1	High Priority / MemQOut1 / MemQDataIn
[8:10]	000	CPtr for Rx, PPtr for Tx
	001	PPtr for Rx
	010	Base[0:6]
	011	Base[7:8], Bound
	100	LateAck for Rx, 32bCompose for Tx
	101	** Reserved **
	11X	** Reserved **

Table A.2: NESAddress: Queue State

Field	Value	Comments
[0:3]	1100	BIU System Registers
[4]	0	sBIU System Register
	1	aBIU System Register
[5:7]	00	General SysReg State
[8:11]	X	SysReg Number

Table A.3: NESAddress: General SysReg State

Number	Name	Comments
0-1	OnePoll	Contains OnePoll?, OnePollAddr
2	ABIULocks	aBIU Only; contains aPBusLock, NotifyLock

Table A.4: BIU General System Registers

A.2 System Registers

The system register state of the BIUs is broken into three sections: general system registers, SSResponse and HALResponse.

General system register space is shown in Table A.3; the list of system registers is given in Table A.4.

The responses to aP Serviced Space transactions, SSResponse, are stored in a special system register array; each element of the array contains the response (IGNORE, NOTIFY, APPROVE, RETRY) for a single transaction type.

A similar encoding is used to access the responses to Snooped Space. The HALResponse (Hardware Approval Logic Response) is generated based on the transfer type and 3-bit clSRAMData response to the cache line of the current address; the

Field	Value	Comments
[0:3]	1100	BIU System Registers
[4:6]	101	SSResponse Table
[7:11]	X	TransferType

Table A.5: NESAddress: SSResponse

Field	Value	Comments
[0:3]	1101	HAL Response Table
[4:6]	X	clSRAMData
[7:11]	X	TransferType

Table A.6: NESAddress: HALResponse

corresponding NESAddress space is shown in Table A.6.

Bibliography

[1] Boon S. Ang and Derek Chiou. StarT-Voyager: Hardware Engineering Specification. CSG Memo 385, MIT Laboratory for Computer Science, February 1997.

[2] Boon S. Ang, Derek Chiou, Larry Rudolph, and Arvind. Message Passing Support on StarT-Voyager. CSG Memo 387, MIT Laboratory for Computer Science, July 1996.

[3] Derek Chiou and Boon Ang et al. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of the First International EURO-PAR Conference*, pages 101–116, August 1995.

[4] Anant Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the First International EURO-PAR Conference*, pages 2–13, 1995.

[5] Andrew Boughton et al. Arctic User's Manual. CSG Memo 353, MIT Laboratory for Computer Science, 1994.

[6] Daniel Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[7] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[8] John Hennessey and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1990.

[9] IBM. *UNION Workbook*, 1996.

[10] G. Leopold. Cray T3D couples MPP, vector technology. *Electronic Engineering Times*, 25(3):63–79, October 1993.

[11] Motorola. *MPC105 PCI Bridge/Memory Controller Hardware Specifications*, 1995.

[12] Motorola. *The 60X Bus Manual*, 1996.

[13] Rishiyur S. Nikhil and Arvind. Id: A Language with Implicit Parallelism. CSG Memo 305, MIT Laboratory for Computer Science, February 1990.

[14] Rishiyur S. Nikhil, Gregory M. Papadoupolus, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 224–235, May 1992.

[15] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An Argument for Simple COMA. In *Proceedings of the First IEEE Symposium on High Performance Computer Architectures*, pages 276–285, January 1995.