5

Design of the Control Logic for StarT-Voyager

by

Daniel L. Rosenband

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1998

[June 1998]

Author ______________________

Department of Electrical Engineering and Computer Science
May 18, 1998

Certified by ______________________ [illegible]

Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ______________________ [illegible]

Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Eng

Eng

Design of the Control Logic for StarT-Voyager

by

Daniel L. Rosenband

Submitted to the
Department of Electrical Engineering and Computer Science
May 22, 1998
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis provides a detailed description of an ASIC I have implemented for the StarT-Voyager multi-processor. The ASIC contains the control logic of an adapter we have designed to connect SMPs into a cluster of workstations. The control logic is one of the central components of the StarT-Voyager multi-processor, a system that aims to be fast, scalable, and that supports both message passing and globally cache coherent shared memory. My thesis will present the high-level architecture of this ASIC and the low-level implementation details.

Thesis Supervisor: Prof. Arvind
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to thank the entire Computation Structures Group for providing me with a pleasant and encouraging environment to work in.

I would especially like to thank Prof. Arvind for the constant support he gave the StarT-Voyager hardware team.

I also want to thank the StarT-Voyager hardware team, Boon Ang, Andy Boughton, Derek Chiou, and Mike Ehrlich. It was great working on the project with these people. I learned a tremendous amount from all of them and enjoyed working with them as a team.

Finally, I would like to thank my parents Doris and Larry Rosenband. Without their constant love and support I would never have been able to accomplish what I have.

Chapter 1 Introduction

This thesis provides a detailed description of an ASIC I have implemented for the StarT-Voyager multi-processor. The ASIC contains the control logic of an adapter we have designed to connect SMPs into a cluster of workstations. The control logic is one of the central components of the StarT-Voyager multi-processor, a system that aims to be fast, scalable, and that supports both message passing and globally cache coherent shared memory. My thesis will present the high-level architecture of this ASIC and the low-level implementation details.

In the next few sections of this chapter I will present an overview of the StarT-Voyager system. This should allow the reader to understand better how the control logic fits into the overall system. I will also discuss some of the design goals and constraints placed upon the implementation of the control logic. I will conclude this chapter with an outline of the remainder of the thesis.

1.1 StarT-Voyager Overview

StarT-Voyager[2] is a multi-processor that is designed to be a fast and scalable system that is primarily based on commodity components. It contains custom hardware components to support fast message passing and a variety of shared memory protocols. While designed to provide good performance at a relatively low implementation cost, attention was paid to providing sufficient flexibility for future research into shared memory protocols and multi-processor system design in general.

The StarT project began many years ago and has evolved from a multi-processor design that contained a custom microprocessor to a design that is built from mostly commercial products. In StarT-Voyager we chose to use IBM dual-processor RS/6000 SMPs as our base system. These systems usually come with two PowerPC 604 processor cards that connect to a shared 60X memory bus. In our system we remove one of the processor cards and insert our custom Network Endpoint Subsystem (NES). This NES provides the hardware for message passing and shared memory protocol processing. The NES also contains the interface logic to the Arctic network[1], a high-speed packet switched network.

StarT-Voyager lies between a highly customized design such as the Stanford FLASH[4] machine, and multi-processor systems such as StarT-Jr[3], which provide network interfaces through an industry standard interface (PCI in this case). The design philosophy behind StarT-Voyager is that it is not necessary to modify the memory system itself (processor, cache or memory controller). Instead, commercially available technology in these areas can be used. However, a direct connection of the NES to the memory bus, rather than an indirect connection through a PCI bus, is needed in order to establish a low latency, high bandwidth interface to the processor and memory system. As a result of the direct connection to the memory bus some modifications would be required to incorporate the NES into other architectures (for example, the Intel Pentium). As will be seen later, these changes would be rather minor.

1.2 NES Overview

Our NES provides an interface between a SMP memory bus and a high-speed network. The functionality that the NES provides to the application processor (the processor card in the SMP) includes message passing and globally cache coherent shared memory. Many hardware features have been designed into the NES to optimize message passing and shared memory performance. However, due to resource constraints and in order to allow future research into shared memory protocol processing, we have also included a commercial embedded processor.

Figure 1.1 shows the major elements of a StarT-Voyager node. The components in the dashed box are all part of the NES. In the next few paragraphs I will give a brief description of most of these components. My thesis focuses on the Ctrl ASIC that I designed, but some of the functionality included in some of the other parts will become clearer in later chapters.

On the far left of the figure is what we call the Application Processor (AP) side of the node. This is essentially the original IBM RS/6000 dual processor base system with one of the processor cards removed. The processor card that remains in the system is the Application Processor. The AP runs most of the application code. The figure differentiates between the data portion (solid) and address portion (dotted) of the AP memory bus.

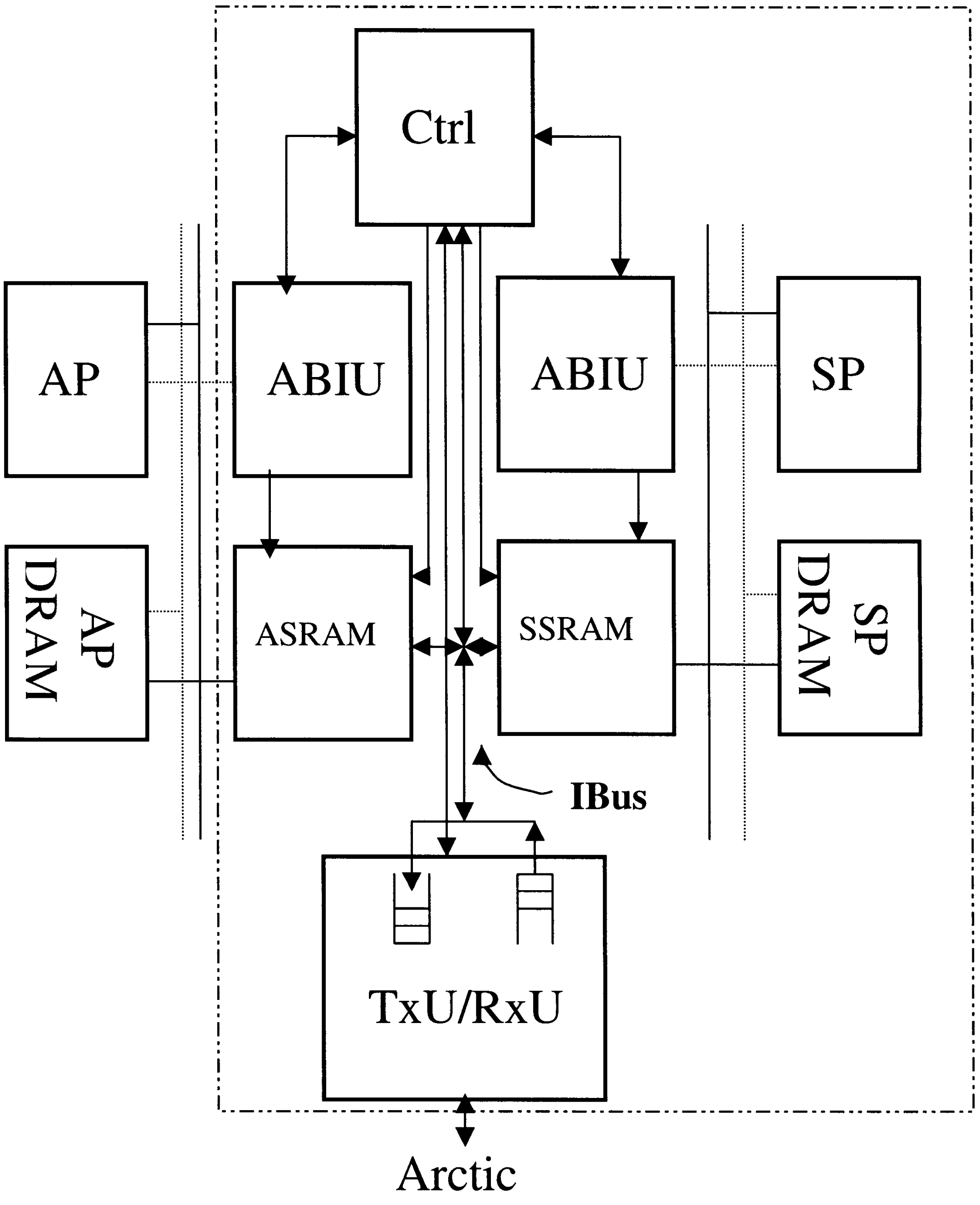

Figure 1.1 StarT-Voyager Node

All interactions between the AP and the NES occur through the memory bus. The only exception is through interrupts as will be discussed in a later chapter. The AP Bus Interface Unit (ABIU) is the NES component that communicates with the processor through the memory bus. The ABIU is able to be both bus master and bus slave. It needs to be bus master for efficient shared memory protocol processing, so that DMA requests to the AP DRAM can be issued by the NES, and so that certain message passing optimizations can be implemented in the NES. The ABIU acts as a slave when messages are being read or composed from/to the NES, when it captures a shared memory access that requires NES support, or when NES state is being accessed.

Similar to the AP side of the design is the Service Processor, or SP, subsystem on the far right of the design. The SP is the embedded processor that is used primarily for shared memory protocol processing. It is also used for part of the NES configuration, network flow control, and various other tasks.

As on the AP side, we have a bus interface unit (the SBIU), sitting on the SP memory bus. The most significant difference between the SBIU and the ABIU is that the SBIU can only be a slave on its memory bus. This means it cannot issue bus transactions to the SP memory bus. The SBIU slave interface is very similar to that of the AP. The biggest difference in the slave section is that the SBIU does not need to snoop for shared memory access as the ABIU does.

Both the ABIU and SBIU interface to the control logic (Ctrl), the ASIC I designed and that is the main focus of this thesis. Ctrl manages the message queues in the NES, is responsible for transmission and reception of messages, and contains hardware to issue DMA reads and writes to AP DRAM. Ctrl also provides special queues that are used for the shared memory protocols, and it provides various other functions which are necessary for the correct and efficient operation of the other components within the system.

Ctrl contains many features that are not necessary for a functional message passing or shared memory system. We believe it will be interesting to experiment with some of these features in order to evaluate their utility, and to see how well applications can take advantage of them.

The two SRAMS, ASRAM and SSRAM, provide buffer space for the message queues, some of the NES state, temporary storage for shared memory protocol processing and DMAs, and space for some translation tables that are needed in our message passing protocol. These are dual ported SRAMS which means that the NES can be accessing them internally, say to transmit a message, while the AP is simultaneously reading a received message out of a message queue in the same SRAM.

On the bottom of figure 1.1 is the TxU/RxU unit. This part of the NES is used to interface to the Arctic network. It contains several FIFOs for synchronization and buffering of incoming data. It also contains an FPGA that does some bit reordering of incoming and outgoing messages. TxU/RxU also computes the CRC of messages being transmitted and received in order to assure that no error occurred in the network. Because we use ECL links to the Arctic network, TxU/RxU also includes logic to convert the CMOS level signals on the NES to ECL levels.

One of the main resources in the NES is the IBus. This is the bus in the center that connects to the ASRAM, SSRAM, TxU/RxU, and Ctrl. It is a critical resource, as most functions that the NES performs require at least one use of the IBus. We have made it 64 bits wide to match the transmit and receive bandwidth of the Arctic network (combined, about 150MB/sec). Because the IBus is one of the critical resources in the NES, I have attempted to make Ctrl manage this resource efficiently.

All in all, our NES provides the functionality needed for efficient message passing and shared memory protocol processing. In addition to the basic features required, we have included some performance enhancing hardware as well as hardware that provides opportunities for future experimentation. The control logic is a central component of the NES and most of the remaining thesis will focus on what functions it performs and how it has been implemented.

1.3 Overview of Implementation

Our choice of technology has varied quite a lot from the beginning of the StarT-Voyager project to the time when we made the final parts choices. Given sufficient resources it would have been nice to have included the ABIU, SBIU, SRAMs, Ctrl, parts of TxU/RxU and possibly even the SP subsystem in a single chip. This would have allowed the components to be more tightly coupled than they now are, which would have led to significant performance improvements in some parts of the design, especially in shared memory protocol processing. Since our hardware design team has consisted of only four to eight members (four have worked on the project from its inception and are still on the team) such a single chip design would have been far too complex.

After much uncertainty over what should be included in an ASIC and what should be put into FPGAs, we finally decided to put all of Ctrl in an ASIC and all other components (ABIU, SBIU, TxU/RxU) in FPGAs. We are using relatively large FPGAs for these components and are using the highest speed grade available at these sizes. As it turns out, the FPGA speeds pretty closely match the speed we have obtained in the ASIC.

ChipExpress will fabricate the Ctrl ASIC in a 0.6um technology. The ASIC contains about 55K gates of logic and several thousand bits of RAM. Pre-layout timing has been about 55MHz, but post place and route timing has always been about 37MHz. We are targeting our NES to run at 35MHz so this should be sufficient.

The design has been completed and we are waiting to get prototypes back. The chip has been sent to fab, and the FPGA designs have been fully implemented. We are also waiting for the printed circuit boards. Hopefully we will have a prototype node running by the end of May.

1.4 Thesis Outline

In the next chapter I will describe most of the functions Ctrl provides to the application and service processor, as well as to the rest of the NES. The chapter will describe what Ctrl does to send and receive messages, how it supports shared memory operations, as well as several other features such as DMA and internal state access. Chapter 3 describes some of the functions from chapter 2 in greater detail. It outlines Ctrl's general architecture and what the functions its major components perform.

Chapter 2 Ctrl Functionality

The control logic (Ctrl) is an ASIC I have designed and implemented for the StarT-Voyager multiprocessor. It is the largest piece of custom hardware in the NES. Much of the hardware in Ctrl is dedicated to providing an efficient and flexible message passing mechanism. However, it also contains functionality that is crucial to the efficient and correct operation of the shared memory protocols. Certain portions and features of Ctrl are not necessary for correct operation, but have instead been included for research purposes as well as to optimize performance.

In the next few sections I will present a high level view of the functionality that is incorporated in the control logic. The following section illustrates what type of message passing support Ctrl provides to the application and service (protocol) processor. In section 3, I show what type of support Ctrl has for reads and writes to its internal state. Section 4 discusses some of the features Ctrl includes that are needed for shared memory operations. The last two sections describe DMA support and a special purpose queue that is used for AP to SP communication.

2.1 Message Passing Support in Ctrl

As in most machines, we have attempted to maximize throughput and to minimize latency in our message passing system. In our case, these parameters are not only important for message passing applications but are also important for shared-memory performance. The reason for this is that the SP subsystem makes heavy use of the message passing support in the NES when executing shared memory protocols.

Rather than support only a single type of message in our design, we decided to support a range of different message types. These types include express messages (short and low latency), basic messages (higher bandwidth but a little higher latency), and tagon messages (messages with a data portion that is appended to it by the NES). We also support various flags in the message header (first word in the message). These flags specify special conditions or operations that should be performed when the message is transmitted or received. Cross products of many of these options are also supported.

A commercial product would probably not include such a large range of message types but we thought they provided an interesting opportunity for future research. We also believe that significant performance improvements can be obtained by providing several, rather than a single, message passing mechanism. Depending on the application needs, the programmer or compiler can choose the message passing mechanism that is best suited for the required data transfer.

Ctrl provides very similar message passing support to the application processor and the service processor (through ABIU and SBIU). Some of the message types will be used by only one processor because it would have taken more effort to separate the functionality than to provide a uniform interface.

The following subsections describe the above-mentioned message passing mechanisms in greater detail. Several sections also detail flow control and an interesting performance optimization we have included in the control chip.

2.1.1 Message Queues

Ctrl maintains the state for all the general message queues in the NES. Ctrl differentiates between basic message queues, which contain messages ranging from 8 to 88 bytes, and express message queues, which contain only 8 bytes of data per message. The basic queues can be larger but have the drawback of using the SRAMs slightly less efficiently.

We use a standard producer/consumer pointer method to maintain all the queues in the NES. Each queue also has a base and a bound. These parameters specify the region of the SRAMs that the queues are located in. Ctrl maintains all the states associated with the queues internally. Ctrl maintains shadow copies of the pointers that it modifies (producer pointers for the receive queues and consumer pointers for transmit queues) in fixed locations in the ASRAM and SSRAM. Every time Ctrl modifies one of the pointers (after a transmit or receive) it writes it back to the SRAMs. This allows the processors to easily obtain up-to-date information about the status of the queues.

2.1.2 Express Messages

The purpose of Express Messages is to provide the application and service processors with a very low overhead mechanism of sending short (32 bit) messages to other nodes in the system. Express Messages will typically be used for synchronization with other nodes, sending acknowledgments, or when only a small amount of data needs to be sent quickly.

In the next paragraphs I will outline how an Express Message is composed, transmitted, and subsequently received by another processor. The reader should refer to figure 1.1 to understand better which NES components are communicating with each other. The example below illustrates how an express message is sent from one application processor to another application processor, but on both the transmit and receive end, a service processor could have been used interchangeably.

A short message can be sent by either the application or service processor through a single uncached 32-bit store operation. The address portion of this store operation contains some fixed fields which allow the ABIU or SBIU (depending on whether the message is being composed from the AP or SP) to recognize that a short message is being composed. However, most of the bits in the address portion of the transaction are not fixed. They are used to indicate from which message queue the message should be sent, what priority the message has, what its destination is, as well as a few other parameters that will be discussed in a later section. The 32-bit data portion of the store is general payload; that is, it does not include any routing or other header information.

Step 1 in the transmit process is the store operation. It is important to note that the 32 bit address portion of the transaction is captured by the ABIU, whereas the 32 bits from the data bus go directly to the ASRAM. By looking at the information in the 32 bit address portion, the ABIU is able to tell for which transmit queue the message is being composed. The ABIU uses this information along with some internal state to determine which ASRAM location the data portion of the transaction should be written to.

The next step is for the ABIU to forward the 32 bits it captured from the address bus to Ctrl. In the process ABIU also communicates to Ctrl for which queue the message is being composed . Ctrl acknowledges receipt of this data, and then in step 3 writes the data to the ASRAM. Since the ASRAM and SSRAM are 64 bits wide, Ctrl will write its 32 bits contiguous to the 32 bits that were written in step 1b.

An obvious question regarding this compose scheme is why we need to send the data over the IBus in step 3, rather than just write 64 bits directly into the ASRAM in step 1. The reason for this is that the ASRAM does not have access to the bits in the address portion of the AP memory bus and hence cannot capture these directly. There are several different ways in our system to avoid the use of the IBus (that is step 3), for example, by having the processor issue two separate 32 bit stores. However, we believe the additional overhead involved in such transactions generally does not warrant their use.

After compose step 3 has been completed, we are ready to transmit the message. Ctrl now needs to schedule the actual transmit. To do so, Ctrl first evaluates whether all necessary resources will be available when the message is moved onto the IBus. It then checks whether this is the highest priority operation that can be performed. If both of these conditions are met, the transmit will be scheduled. (Messages in each queue are transmitted in FIFO order, and at this point we may transmit a message that was composed much earlier than the most recent compose in steps 1-3). A later section describes the pipelining, priorities, and resource constraints used by the scheduler in greater detail.

Now that the transmit has been scheduled, the Express Message is read from the ASRAM and moved onto the IBus. Ctrl makes TxU (in Txu/RxU) capture the message on the IBus. Ctrl also captures the message because it needs to ensure that it is a valid message (for example, it needs to check that the length field is set correctly), and because it needs to extract the logical destination node number from the message. Ctrl uses this logical destination node number as the index for a translation table. The translation table is usually located in the SSRAM and among other things contains a translation from logical destination node number to physical destination node number (the actual routing path to a destination node). TxU captures the translation table entry after it appears on the IBus.

TxU now contains the message and an entry from the translation table. It combines these (reorders bits), computes the CRC of the message, and sends it to the Arctic network. An Express Message has been transmitted. Immediately after Ctrl places the translation table entry onto the IBus, Ctrl will also place the producer pointer of the queue from which the transmit originated onto the IBus. This occurs because the producer pointer needs to be written to its shadow copies in the ASRAM and SSRAM. The processors can easily access this state in the SRAMs in order to obtain information about the status of each queue.

The receive process for Express Messages is almost the reverse of the transmit process. First, the message comes from the Arctic network into RxU. Here it is buffered in a FIFO until Ctrl requests that it be placed on the IBus. At this point RxU will notify Ctrl that a message has been received. (Information that Ctrl can use to determine how large the message is and into which queue it will be placed is also passed to Ctrl over a dedicated interface).

Just as during transmit, Ctrl needs to schedule the data transfer between RxU and the SRAM. To do this, Ctrl needs to ensure that all resources will be available when needed and Ctrl has to check that this is the highest priority operation that should occur. Once these conditions have been satisfied, Ctrl will assert the signals that enable RxU to place the message onto the IBus. Ctrl also asserts the necessary control signals so that the message is written into the appropriate queue in the ASRAM.

After the message has been placed into the message queue in the ASRAM, Ctrl will update the shadow copies of the receive producer pointer in the ASRAM and SSRAM. Ctrl then notifies the ABIU that a message has been received and communicates to it which receiving queue was used.

The AP will periodically poll the ABIU to see if a message has been received. All the processor needs to do when polling is to issue an uncached load. The address specifies which queues the processor wants to poll; that is, see if they contain any new messages. If one of the polled queues contains a new message, the ABIU will assert the appropriate ASRAM control signals which will in turn place the message onto the AP memory bus.

Overall the processor overhead of composing, transmitting, and receiving an express message is quite low. Unfortunately, our NES design has some inefficiencies during the compose and transmit phases, but given the nature of our design and the SRAMS (they sometimes require a dead cycle and contain a pipeline stage themselves) this was hard to avoid. Extending the design to recover some of the wasted IBus bandwidth, and to eliminate some of the latency through additional buffering in Ctrl would have added considerable design complexity. We believe the inefficiencies will only marginally effect the available message passing bandwidth and only add a few cycles to the message latency.

2.1.3 Basic Messages

As far as Ctrl is concerned, the major difference between basic messages and express messages is that basic messages are variable length messages up to 96 bytes in length. Express messages (without tagon) contain only 8 bytes of data. Another difference is that the basic message queues are managed slightly differently than the express message queues. They can be larger but use the SRAM space less efficiently.

For the processor, basic messages are very different from express messages. A message compose occurs through cached stores. This allows large messages to be composed quickly. The cache locations that contain the message then need to be cleaned. This puts the message on the memory bus, at which point it is captured by the NES and written directly into one of the SRAMs. This means no compose through Ctrl is necessary, and no IBus bandwidth is wasted. The processor then performs a message queue pointer update to indicate to the NES that a message has been composed. The ABIU captures this pointer update and propagates it to Ctrl.

The message has been fully composed once Ctrl receives the queue pointer update. At this point the message is at a similar stage as the message was after step 2 in the express message section. The transmit process is very similar to the express message transmit. One difference is that Ctrl extracts the length of the message from the first 64 bit word that is placed on the IBus. This length tells Ctrl how many more words it should read from the SRAM and send to TxU. This additional message data is placed onto the IBus after the translation table entry has been read.

The basic message receive process is also similar to the express message receive. The major difference is that a variable number of data words need to be transferred from the RxU to the SRAM during the receive process. The length of the message determines how many words are transferred.

The BIUs do not support polling of basic messages as they do for express messages. Instead, the processor polls to see which basic message queue is not empty. The processor then reads the message from the SRAM into its cache.

Basic messages have a higher latency but much higher bandwidth. The additional latency stems from the requirement that the processor explicitly clean some cache lines on transmit and flush some of them on receive. A later section that describes the Reclaim optimization will illustrate how the processor overhead can be reduced. However, the latency will remain somewhat above that of express messages. Bandwidth will generally be higher because data can be transferred in burst mode on the memory bus, and the cost associated with the translation table can be amortized over several cycles. Also, no IBus bandwidth is wasted during the compose process.

2.1.4 Tagon Messages

We have designed a feature into our NES, which allows an express message to contain an additional 1.5 or 2.5 cache lines (48 or 80 bytes) of data. This data will not be taken from transmit message queues, but instead comes from a location in either ASRAM or SSRAM. The SRAM location is variable and Ctrl determines it based on some queue-specific internal state and some of the data in the message header (the first 32 bits).

Tagon messages will be used extensively by shared memory protocols. They allow a processor to send low overhead messages with high bandwidth. Since the data is located in the SRAMs and it can be used by multiple messages, this provides an efficient mechanism to send multiple nodes the same piece of data. An example of this type of use in a shared memory protocol would be when the home site sends an invalidate command to several other nodes.

Another example of how tagon messages can be used in a shared memory protocol is the case when the SP wants to send some data from AP DRAM to another site. First, the SP issues a command to read the data from AP DRAM into the ASRAM. It can then compose an express message with a tagon portion that contains the data that was just written into the ASRAM. This avoids the SP having to read the data out of the ASRAM, packaging it into a medium message, and only then being able to compose and transmit it.

The transmit mechanism in Ctrl that supports the tagon messages is relatively simple. It can tell from the express message header whether there should be a tagon portion, and if so, where it is located. If there is a tagon portion, then it is read from the appropriate SRAM and put into TxU after the translation table entry has been read.

The receive process is more complicated. Ctrl maintains state for separate tagon data queues for each express message receive queue. As usual, the first word of the express message goes into the header (normal) queue. All subsequent words belong to the tagon portion of the message and are placed into the data tagon queue.

I made an extension to Ctrl to allow basic messages to have tagon data. It is not clear when an application would choose this rather than an express message with tagon, but we thought it might be an interesting feature to experiment with. Another extension we made was to allow basic message queues to send messages to express message queues and vice versa. Much of the logic for these features and express messages with tagon is shared within Ctrl.

2.1.5 Raw Messages

A raw message is a message in which all translation is performed by the transmitting processor. This means that no translation table entry is used. Raw messages will typically not be allowed to be sent by user code because these messages are not protected. Unprotected messages can be sent to any destination, whereas normal messages are limited in their choice of destination nodes and queues by the translation table entries.

Raw messages are composed just like other express and medium messages. The main difference in how Ctrl deals with them is in the transmit process. Since no translation table entry is needed, Ctrl can transmit the message as a continuous stream of data. Ctrl actually speculates that all messages are raw until it sees the header (first 64 bits) of the message. This permits Ctrl to utilize the IBus more efficiently during raw message transmits, because no idle cycles between the message header and the translation table entry are wasted.

2.1.6 Miscellaneous Message Controls

Our NES also provides some additional message passing controls. The controls include information on when AP and SP should be interrupted during reception of message, whether raw messages are allowed to be transmitted from a certain queue, whether tagon messages are allowed, how big the standard tagon portion is, as well as some controls that provide additional flexibility in the receive process.

Some of these controls are handled on a per queue basis, whereas others are on a per message basis (they are encoded in the header of the message). We have tried to allow the user as much flexibility as possible in choosing the features that the NES should use and what type of protection should be enforced.

2.1.7 Flow Control

Flow control during transmits is based upon a counter that Ctrl maintains. This counter indicates how many buffers are free in the input section of the Arctic routing chip that TxU/RxU connects to. Whenever a message is transmitted, Ctrl decrements this counter. The counter is incremented whenever Arctic acknowledges (through TxU/RxU) that a buffer in its input section has been freed.

Arctic and the NES support two levels of message priorities. Ctrl will transmit a low priority message whenever the counter is greater than one; that is, more than one buffer is free in the Arctic input section. A high priority message can be sent whenever the counter is at least one. In this way we insure that at least one high priority message can always be transmitted.

Flow control on the receive end is based on the same principles as flow control for transmits. The roles of Ctrl and Arctic are just reversed. Ctrl sends a buffer free signal whenever a message has been removed from one of the receive queues.

Arctic keeps track of the number of buffers that are free inside Ctrl.

This approach does not work very well on the receive end. Let us assume that Arctic initially believes that all receive queues can contain 16 maximum-sized messages. The Arctic counter will be set to 16. Let us also assume that all the queues are initially empty. Now, four messages are sent into four different queues. The Arctic counter will decrease to 0 and Arctic will think all the buffers inside the NES are filled. Only when a processor removes a message from a receive queue will a buffer free be sent to Arctic, and only then can Arctic send the next message.

The problem here is that all the queues are only one-quarter full. However, Arctic believes that no buffers are free. We avoid this problem by allowing Ctrl to eagerly send buffer frees to Arctic. (This option can be enabled or disabled on a per queue basis.) If eager buffer free is enabled, then Ctrl will send a buffer free as soon at it receives the message into the queue. Ctrl does not wait for the message to be removed from the queue by one of the processors.

As a result of eagerly sending buffer free signals queue overflow can occur. (If messages are not being taken out of a queue, and messages are continuously being added, then the queue will eventually overflow.) Since we do not want to lose messages, Ctrl includes an overflow queue. Clearly, this overflow queue cannot eagerly send buffer frees since we do not have an overflow queue for the overflow queue.

Receiving into the overflow queues occurs in the same way as receiving into a basic message queue. The only difference is that a 64 bit header is appended by Ctrl. This header identifies to which queue this message was actually going. Software in the

SP processes the messages in the overflow queue and forwards the appropriate data to other parts of the system as required. Processing the overflow queue is a high overhead task and we hope that overflow will occur infrequently.

The logic inside Ctrl that determines whether an overflow will occur is quite complicated. It is heavily pipelined and requires pre-computing of lookup tables so that the latency of receives is not increased. However, we believe the overflow queue is a powerful feature in our design. It will be interesting to measure how much we benefit from it. The overflow queue can also be used to receive messages whose destination queues are nonresident.

2.1.8 Reclaim

One of the problems with medium messages in our design is that composing and receiving them requires significant processor resources. The reason for this is that the messages are located in the processor cache. In order for the messages to get from the cache into the NES (during transmits), clean operations must be performed on the cache. This will force the message data to be placed on the memory bus, where it can be captured by the NES. For receives as well, parts of the cache need to be flushed before a receive occurs. This prevents the processor from reading stale data in its cache.

Clean and flush are processor intensive operations (20-30 cycles), and it would be desirable not to require the processor to incur such a large penalty for each basic message transmit and receive. One solution to this problem has been implemented in Ctrl where the NES issues the cleans and flushes for the processor. Consequently, the processor performs mostly the same operations as it does during normal transmit and receive of basic messages. The two exceptions are that it does not issue the cleans and flushes, and instead of updating the normal producer or consumer pointer, it updates a reclaim pointer inside Ctrl.

As soon as Ctrl observes a reclaim pointer update it will begin a reclaim operation. This involves issuing the appropriate cleans or flushes to the memory bus (through the BIU). Depending on the size and number of messages that the processor composed, this could range from a single bus operation to close to one hundred. Once the reclaim has been completed, Ctrl can again transmit or receive from the reclaimed queue.

Since the NES can be bus master (which is required to issue cleans and flushes) only on the AP side, this reclaim optimization is only used on the AP side of the system. We believe reclaim will lead to significant performance improvements, and it will be an interesting feature to experiment with.

2.2 State Access

We have attempted to make the entire state within the NES accessible to software running on the application and service processor. Much of this state will be protected and will only be accessible by system code. Such state includes NES configuration registers, status registers (for example, interrupt vectors), and portions of the SRAMs.

Some of the state within the NES will be accessible by individual user processes as well. For example, a user process that is composing messages will need to read state from the NES in order to tell how much space is left in the queue. Similarly, a process receiving messages will want to tell how many messages are in its receive queue.

The NES provides the processor with access to the SRAMs and its internal state. The next few sections illustrates how the state can be accessed and what role Ctrl plays in these accesses.

2.2.1 SRAM State Access and Data Motion

The NES state accesses will be captured by either the ABIU or the SBIU. The ABIU will capture the AP requests and SBIU the SP requests. AP reads and writes to ASRAM. SP accesses to SSRAM can be satisfied by the BIUs without Ctrl being involved. For example, if SP wants to read data from the SSRAM, SBIU will capture the load request. After receiving the request, SBIU asserts the SSRAM signals that will return the requested data to the data portion of the memory bus.

The above scheme does not allow the SP to read the ASRAM or AP to read the SSRAM. This is because the SBIU does not have direct access to the ASRAM and the ABIU does not have access to the SSRAM. Such operations would make parts of shared memory protocols more efficient and for that reason we decided to support them. These types of state accesses also allow a process running on the application processor to have a medium message queue in the SSRAM.

As far as a processor is concerned, there is no difference whether it is accessing ASRAM or SSRAM. In either case it is issuing a load or store with slightly different addresses. (There is higher latency to the memory that is farther away.) Below I describe an example of how SP can write to the ASRAM. This example illustrates how we support the desired SRAM access.

SP issues the store operation. SBIU then captures this operation and notices that it is a request to put data into ASRAM. Since SBIU cannot access the ASRAM, it forwards the request to Ctrl. SBIU also puts the data into a temporary location in SSRAM. Ctrl then reads the data from the temporary location in the SSRAM and puts it into the desired location in the ASRAM. The write is complete.

If SP had issued a load from ASRAM, then Ctrl would have moved the data in the opposite direction as was the case in the example above. That is, Ctrl would have moved the data from the ASRAM location into a temporary SSRAM location. SBIU would have then transferred the data from the SSRAM onto the SP memory bus.

We call this type of data transfer between SRAMs data motion. Ctrl provides a relatively flexible interface for data motion commands. It supports 4 byte (either the high or low 4 bytes of the 8 byte data words in memory), 8 byte, and 32 byte data transfers between the SRAMs. Arbitrary source and destination addresses can also be specified.

2.2.2 BIU State Access

Writes to BIU state can be handled by the BIUs themselves. Surprisingly, Ctrl plays a role in BIU state reads. The reason for this is that the BIUs have no connection to the data portion of the memory bus they connect to and hence have no way of returning the data. Instead, the BIUs pass the data they want to return to Ctrl. Ctrl will then write this data into a temporary location in either ASRAM or SSRAM. Once this has been completed, the BIU can assert the necessary SRAM control signals which will in turn put the data onto the data bus.

This may seem like a rather complicated way to return a piece of data from a component that seems to be located so close to the processor. However, this type of state access will occur infrequently and this was a solution that involved a minimal amount of additional hardware. The extra hardware is minimal because the same interface to Ctrl can be used that was used for express message composes. The reader should recall that during the express message compose, Ctrl was performing exactly the same type of operation. It was accepting data from a BIU and writing it into one of the SRAMs.

2.2.3 Ctrl State Access

State inside Ctrl is divided into three separate regions. They are called QRAM, EPRAM, and SysRegs. The QRAM contains most of the queue-specific information that the processor writes. This information includes the queue base, bound, permissions, producer pointer if it is a transmit queue, and consumer pointer if it is a receive queue. It is organized in a 32 X 56 bit RAM (internal to the ASIC). Every 56-bit entry contains information pertaining to a specific queue. Transmit and receive, as well as medium and short queues, have slightly different formats for their entries since slightly different information is required.

The EPRAM contains eight entries, each containing information for a set of four queues (two transmit and two receive queues). All four queues are either basic or express message queues. They cannot be mixed. We call such a set of four queues an endpoint. The EPRAM entries contain mostly queue-specific information that Ctrl writes; that is, the consumer pointer for the transmit queues and the producer pointer for the receive queues. The EPRAM is shadowed in the SRAMs by the processors for quick access and therefore will be accessed only infrequently from outside the chip. The SysRegs consist of mostly status and configuration registers. Most of these will be initialized during bootup and will not be touched by either processor afterwards. However, they provide detailed information about what state Ctrl is in, and will be useful during machine bringup.

Ctrl provides a single interface (actually one to ABIU and one to SBIU) to access its internal state. An address is specified to indicate what state is being accessed and an opcode is specified to indicate what operation should be performed (read, write, or read-write). A data value is also supplied in the case of a write or read-write operation. The read-write feature is useful because the processor may be writing only a single bit in a SysReg, but at the same time it would like to see what other bits are in this SysReg. The read-write mechanism allows this to occur through a single request.

The first step in a read-write to state inside Ctrl is for the BIU (in this case the ABIU) to capture a request from the AP on the memory bus. It recognizes that Ctrl is being accessed and forwards the request to Ctrl. Ctrl then updates the state that should be written.

At this point the write portion is complete. Since this was a read-write request, the new value also needs to be returned to the processor. To do this, Ctrl writes the state that was accessed to the SRAM (in this case the ASRAM since it was an AP request), and notifies the BIU that Ctrl has completed the read-write request. The BIU then reads the data out of the SRAM and puts it onto the data portion of the memory bus.

2.3 Shared Memory Support in Ctrl—MemQOut

Ctrl provides a special type of queue (MemQOut) that is used by the SP for shared memory protocol processing. The SP can insert three types of operations into this queue--transmit requests, data motion requests, and requests to issue a bus operation to the application processor. Ctrl guarantees that these operations are executed in FIFO order and that no data hazards occur.

The three MemQOut operations (transmit, data motion, and bus operation) allow the SP to send data easily from AP DRAM to other nodes in the system. This occurs when the SP is moving shared data from the AP to another site in the system. They also allow the SP to quickly invalidate data in the AP or send data into AP DRAM. There are many other examples of how these three operations can be used in a shared memory protocol.

Ctrl actually supports two MemQOut queues. The only difference between the two queues is that one executes at a higher priority than the other. This support has been included in Ctrl in order to prevent deadlock. In the remainder of this section I will treat MemQOut as a single queue. Even though there are two queues, each one acts independently and operates as is described below.

The interface for SP to compose operations into the MemQOut queue is very similar to the normal express message compose mechanism. SBIU captures the compose request and forwards it to Ctrl. Ctrl accepts the compose and writes the MemQOut operation into the MemQOut queue in one of the SRAMs. However, Ctrl also stores the type of the MemQOut operation (transmit, data motion, or bus operation) in an internal

FIFO (MemQOutTypeFIFO). This FIFO is used by Ctrl to schedule the MemQOut operations. Without having to read the operation out of the SRAM, Ctrl can simply look at the MemQOutTypeFIFO to see which operations will be next. This helps Ctrl schedule the MemQOut operation more efficiently because it can prefetch the operations, and schedule them closer together. This significantly improves the MemQOut performance which in turn should improve the performance of the shared memory protocols.

The transmit operation performed by MemQOut is similar to the express transmit described earlier. Ctrl recognizes from the type FIFO that a transmit should occur. It then passes the MemQOut queue information (base, bound, consumer pointer, etc.) to the logic responsible for transmits inside Ctrl. The MemQOut queue entry provides the 32-bit header and 32-bit data portion, just like any other express message transmit queue would. Since MemQOut transmit is handled no differently than other transmits, anything that is allowed by another express queue can also be performed by MemQOut transmits (tagon messages, raw messages, etc.).

The bus operation (BusOp) support through the MemQOut facility permits the service processor to issue arbitrary bus operations to the application processor. The SP issues all shared memory bus operations that go to the AP through this interface. In order to issue one of these operations Ctrl reads the operation from the MemQOut queue (in a SRAM) onto the IBus and captures it. It then sends the bus operation to the AP memory bus (through ABIU).

Ctrl contains several stages of buffering for the bus operations. This will significantly reduce the latency of these operations because Ctrl will always have bus operations ready (assuming they are available) to issue to the AP. Rather than having to fetch a new bus operation when an earlier one completes, Ctrl will be ready to supply a new one immediately. The buffering will also significantly increase the throughput of bus operations being issued from SP to AP.

The data motion function in MemQOut is used to move data from ASRAM to SSRAM or vice versa. This can be used when SP wants to write data from SSRAM into AP DRAM. MemQOut uses the same logic to perform this data motion as was used for the SRAM to SRAM data transfer in the state access section. The only difference is that rather than getting the source, destination, and size of the data transfer from the BIUs, these parameters are extracted from the entry in the MemQOut queue.

All in all, I think MemQOut has been implemented efficiently in Ctrl. Buffering has been added for some of the functions to improve performance, and the operations are scheduled as close together as possible. The three operations are guaranteed to execute in FIFO order. Even though enforcing this efficiently increased the hardware complexity, we decided to include it because it makes the shared protocol much simpler and faster.

2.4 DMA

Ctrl contains hardware to support DMA operations to AP DRAM. There are four DMA operations that control supports: (1) reading of large regions of AP DRAM into a special queue (DMARd), (2) transmitting the data read by a DMARd operation to another node (DMATx), (3) receiving DMA messages from another node (DMARx), and (4) writing the received DMA messages to AP DRAM (DMAWt). These four operations are described in greater detail below.

A DMARd operation is issued by the SP through the MemQOut facility in Ctrl. Even though the operation goes through MemQOut, it is unrelated to the other operations (transmit, data motion, and bus op) that MemQOut performs. This means that there are no scheduling constraints between DMARd operations and the other operations. The only reason for composing DMARds into MemQOut is that we did not want to add another special DMARd queue to the system.

Ctrl contains an internal FIFO that buffers up to four DMARd commands. This prevents the DMARds from blocking other MemQOut operations. The DMARd engine inside Ctrl reads data from AP DRAM and places it into a DMATx queue in ASRAM. The AP DRAM source address and the amount of data that should be read is specified in the DMARd command.

Closely connected to the DMARd engine is the DMATx engine in Ctrl. It is responsible for transmitting the data that was read by the DMARd engine to another node. The DMATx command specifies how much data to transmit and what its destination is. A DMA tramsit message contains two cache lines (64 bytes) of data from the DMATx queue, as well as two 64-bit words that indicate what the receiving end should do with the data. I have also included an option that allows multiple DMATx operations to use the same DMARd data. This enables DMA multicasts to be issued.

Similar to the DMARds, the DMATx operations are also composed by the SP into MemQOut. DMATx has a four-entry FIFO to buffer DMATx commands. This FIFO prevents other MemQOut operations from stalling due to long DMATx operations.

Again, the MemQOut transmit, data motion, and bus op operations are not executed in FIFO order with respect to the DMATx operations.

The DMARx engine inside Ctrl is responsible for receiving messages that were sent by a DMATx. The DMARx splits the incoming message into its data portion (the two cache lines) and the two 64 bit bus operations that specify what should be done with the data. Each portion of this split message is put into its own special queue.

The DMAWt engine then processes the data that was written into the two queues by the DMARd engine. It sends the bus operations (the two 64 bit words instructing the NES what to do with the received data) to the ABIU. This in turn reformats the operations and forwards them to the AP memory bus. Usually these bus operations will simply read the data out of the ASRAM and write them to some location in AP DRAM. We have provided some flexibility that will allow the programmer to decide exactly which bus operation should be used for this write.

We have made some extensions to the receive end of the DMA engine to allow arbitrary queues to send messages to DMARx. We have not studied this carefully yet but we believe we will be able to use this extension to send data to the AP from a remote node. The SP at the destination node does not get involved in the transaction.

2.5 AP to SP Communication and Acknowledgments—MemQIn

In order to support a shared memory protocol, we need a way for a memory reference to get from the AP system bus to the SP. For example, this would be required when the AP wants to read data that is located at a remote node. The NES provides a special queue (MemQIn) to support AP to SP communication. To Ctrl, this communication looks almost the same as any other express message compose. The ABIU captures something on the AP memory bus and decides to forward it to the SP. In order to do this it uses the express message compose interface that Ctrl provides. It then informs SBIU that a message has been composed into MemQIn over a dedicated interface. When the SP polls the SBIU for any incoming messages it will then see that MemQIn contains a new request.

The only difference to Ctrl when ABIU composes these MemQIn operations is that Ctrl handles them at a higher priority than regular express message composes. This is done in order to reduce the latency of shared memory requests. (Ctrl actually differentiates between several different types of MemQIn composes. MemQIn composes that are due to a read request are handled at a higher priority than those originating from write requests because reads will stall the processor quickly if they are not satisfied.)

Ctrl also makes use of MemQIn to communicate with the SP. Each MemQOut operation has a bit which specifies whether Ctrl should acknowledge completion of that operation. If this bit is set and the operation completes, then Ctrl will send an acknowledgment via MemQIn. This acknowledgment of completion is important for the shared memory protocol. For example, when SP requests that a cache line in the AP be invalidated, the SP will want to know as soon as possible when this occurs. The reason for this is that as soon as the SP knows that the invalidate has completed, it can give exclusive rights to the cache-line to some other node.

Below I describe the process for Ctrl to send an acknowledgment to SP. First, Ctrl requests a MemQIn compose from ABIU. In step 2, ABIU replies by requesting that a MemQIn be composed by Ctrl. ABIU also raises a signal that says Ctrl should supply the data for the MemQIn compose. Ctrl then performs the compose, replacing the ABIU compose data with its own acknowledgment.

This acknowledgment will identify the MemQOut operation that Ctrl has just completed so that SP knows which operation is being acknowledged. Finally, ABIU tells SBIU that the compose has completed. Once SP gets to the just composed MemQIn message, it will know that the Ctrl operation has completed.

2.6 Scheduling

Ctrl attempts to schedule operations so that they occur as closely to each other as possible. Since much of the design is pipelined and we want operations to be scheduled efficiently, Ctrl needs to look ahead to when operations will complete. It also needs to schedule operations several cycles before they will actually begin.

The most common resources that Ctrl looks at when scheduling are the IBus, the SRAMs, and the internal memories (EPRAM and QRAM). Obviously, the IBus cannot be used simultaneously by two operations, and there are some restrictions on the SRAMs when switching from read to write operations. Other resources such as MemQOut buffering space are also important.

When scheduling an operation Ctrl ensures that the required resources will be available when they are needed, that the operation does not violate any FIFO constraints (when it is a MemQOut operation), and that the operation to be executed has the highest priority. In order to optimize performance, most of the priorities of operations have been hardwired inside Ctrl.

Chapter 3 Ctrl Implementation

In this chapter I will present a more detailed description of the Ctrl ASIC. This will allow the reader to obtain a better understanding of the lower-level implementation details of the functions that were described in the previous chapter.

In the next section I will discuss the major modules inside Ctrl and outline how these blocks are interconnected. Section 3 will describe a few facts about Ctrl's internal state, and section 4 describes the tools and technology that were used to implement Ctrl.

3.1 Major Modules Inside Ctrl

There are six major modules in Ctrl. They are Master (responsible for scheduling operations inside Ctrl), QState (contains most of the internal state), BE (handles DMA, Reclaim, and interface to ABIU bus master), EELongTx (performs transmits), EELongRx (performs receives), and DM (performs data motion). These modules along with their internal connections are shown in figure 3.1. In the next few subsections I will explain what tasks each module is responsible for and outline how some of these functions are implemented.

The reader should note that the IBus is a bidirectional interface. It can be used as either an input or output. Internally all signals go in a single direction. Except for the bidirectional IBus output driver, no signals are tristated. The IBus is also latched at the periphery. This easily separates incoming and outgoing signals.

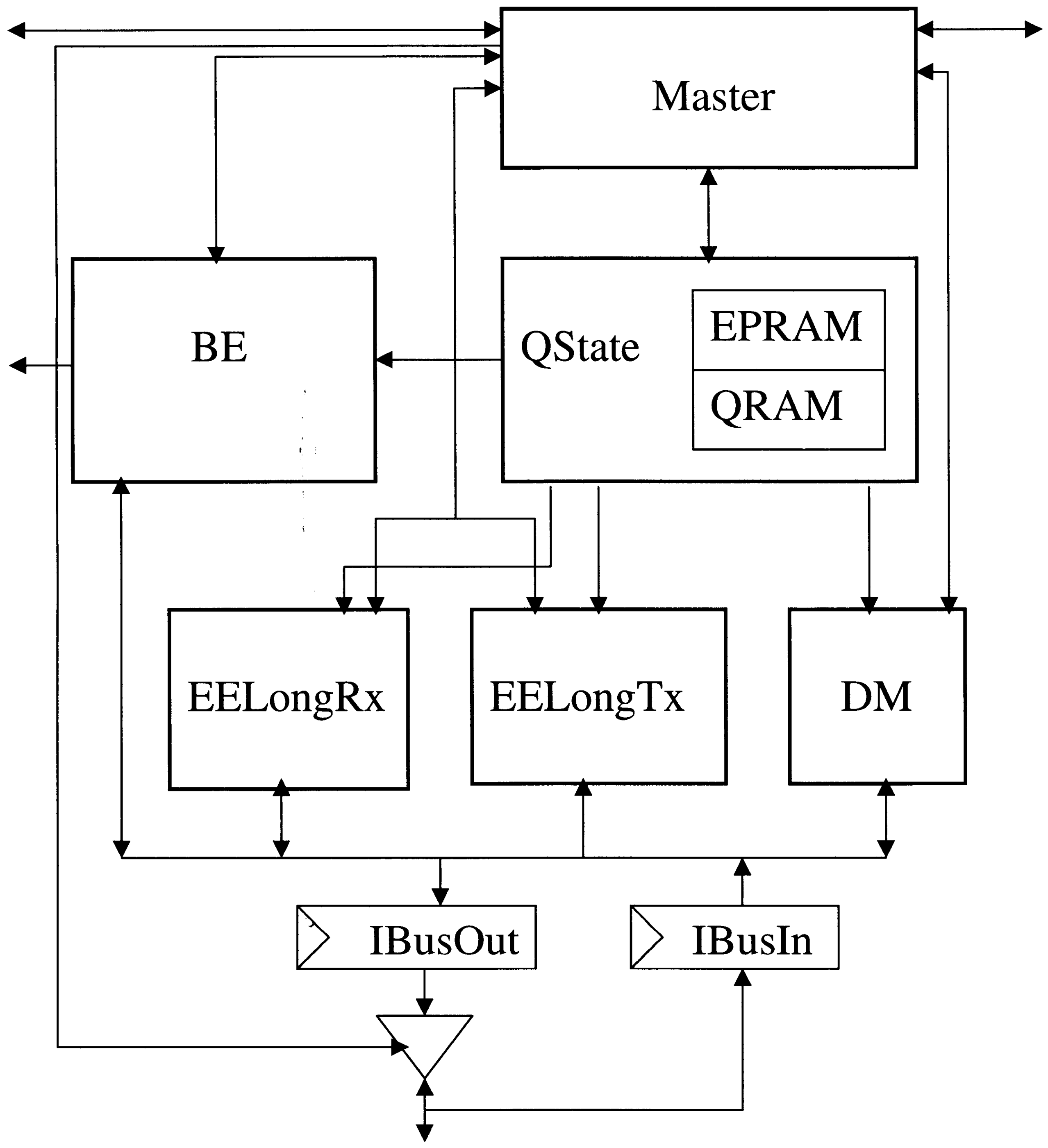
Master
BE
QState
EPRAM
QRAM
EELongRx
EELongTx
DM
IBusOut
IBusIn

Figure 3.1

3.1.1 Master

The Master is responsible for scheduling most operations that Ctrl performs. It is quite complex because it has to ensure that there are no conflicts or race conditions. It uses a priority encoder to determine the next operation that Ctrl should perform. In addition, it contains logic to keep track of which operations are active, what buffers in the various modules are full, and it determines the order in which the operations from these buffers can be issued so that FIFO ordering is enforced.

When scheduling operations, Master first determines what operations can actually be scheduled at the given time. It then picks the highest priority operation that can be scheduled. Even though in most cases Master can schedule only one operation in a single cycle, several operations can be performed simultaneously if they do not require use of the same resource.

In many cases Master is not able to schedule certain operations because it is not known when other operations that use the same resources will complete. For example, transmits take a variable amount of time to complete (based upon the length of the message). During the first few cycles of the transmit it is not known how long the message will be. This means that Master cannot schedule other operations that use the same resources as the transmit until it is known when the transmit will complete.

In some cases, operations perform very different tasks but one of the resources they require is shared. For example, a reclaim needs to access the internal queue pointers when it completes a reclaim.

Since a reclaim may take several hundred cycles to complete, it would not be efficient to have all other operations that require access to the internal state wait for the reclaim to complete before they begin. To accommodate this, some operations are scheduled even though a resource conflict may arise. Master then monitors access to the resource and if there is a conflict it will grant one of the operations access to the resource and instruct the other operation to wait.

I used the above scheduling mechanism when access to Ctrl's internal state is the only possible conflict that can arise. An alternative could have been used but the benefit would have been marginal and it would have significantly increased the scheduling complexity and the critical timing path of the design.

The two resources that are used by nearly all operations are Ctrl's internal state and the IBus. Master generally looks up to three cycles ahead in time to see whether these resources will be free. Since the design is heavily pipelined, this look ahead allows operations to be scheduled much closer together than if it had not been implemented. Consequently, we get higher utilization of the critical resources, and better performance.

Besides ensuring that no resource conflicts arise, Master also needs to ensure that the MemQOut operations occur in FIFO order. This turned out to be quite complex because several MemQOut operations may be buffered in different modules, all waiting to be executed. The MemQOut operations are scheduled using several small FIFOs inside Master. These FIFOs maintain information about what operations are queued in the various modules, in what order they need to be executed, whether they have begun execution, and whether they have completed.

Below I have listed most of the operations that Master schedules. They are listed in descending order, from highest priority to lowest priority.

SBIU Internal state write (simultaneous to others if possible)
ABIU Internal state write (simultaneous to others if possible)
MemQIn Compose (Ctrl or Read)
MemQOut1 Data Motion
MemQOut0 Data Motion
MemQOut1 60X Read (and buffer)
MemQOut1 DM Read (and buffer)
MemQOut1 Tx Read (and start Tx)
MemQOut1 DMA Read (and buffer)
MemQOut0 60X Read (and buffer)
MemQOut0 DM Read (and buffer)
MemQOut0 Tx Read (and start Tx)
MemQOut0 DMA Read (and buffer)
MemQOut Compose
SP Data Motion
AP Data Motion
SP Comopse + MemQIn Write
AP Compose
DMA Wt
Receive message
Transmit message
DMA transmit

Operations from the service processor are always scheduled at a higher priority than their application processor counterpart. The reason for this is that SP requests should be executed with as a low a latency as possible.

This is because the SP will execute the shared memory protocol and the performance of shared memory systems is very much dependent on the latency of operations in its protocol.

In the implementation phase of the project, we found that rather than have used a fixed priority scheme, it might have been beneficial to have added some variation to the priorities. For example, round robin scheduling between general transmits and DMA transmits would have balanced the bandwidth that was allocated to these operations. In simulations, we sometimes saw that the DMA transmits were starved for considerable amounts of time because there was always a general transmit that could be performed.

However, we think the priorities were assigned reasonably and that starvation will not occur in practice.

Due to timing constraints, the priority encoder inside Master and the logic that issues operations to other modules involves a two-stage process. This means that in one cycle the priority decoder will determine what the next operation should be, and only in the following cycle will the module that will perform this operation actually begin executing. In order to increase the performance of the highest priority operations, these operations begin executing immediately. I have attempted to minimize the effects of this additional stage, but in some cases it has resulted in an additional cycle of latency.

As can be seen in figure 3.1, Master connects to all other modules in the design. Signals going from Master to these modules indicate when operations should be performed, whether the modules should latch incoming data into one of their buffers, and whether or not the modules are granted access to the resources they requested. The modules return signals to Master indicating when operations have begun, when they complete, and what resources are being requested.

All in all, I think operations are scheduled efficiently by Master. Master ensures the ordering of MemQOut operations and schedules operations as closely together as possible. Some inefficiency arises in the scheduling mechanism. Alternative design approaches would have increased the critical timing path and would not have led to any performance improvements.

3.1.2 QState

QState contains most of the state inside Ctrl. It includes two internal

RAMs for queue-specific information. One of these, the QRAM, contains the information about each queue that the processor writes. The EPRAM contains the information about queues that Ctrl writes. QState also contains all the configuration registers and system registers (SysReg) that are accessible from outside the chip.

Most of the system registers are accessible through dedicated wires by other modules in the chip. This allows the state to be observed continuously by multiple modules. However, writes to most of the state are serialized. This means that only one write to state inside QState can occur in a single cycle. The writes are serialized so that only one write port is required for the registers. Master serializes the writes to QState. Since the QRAM and EPRAM are single ported SRAMs, Master needs to coordinate read accesses to them as well.

3.1.3 EELongTx

EELongTx contains the finite state machine that performs message transmits. After Master determines that a transmit should occur, it chooses the highest priority queue that is available for transmits. Master then issues a read to QRAM inside QState. The QRAM entry that contains the information about the source queue of the transmit is passed to EELongTx.

At this point EELongTx has all the information it needs to begin the transmit. The QRAM entry that it has received contains the transmit queue producer pointer, base, and bound. EELongTx uses these to construct the SRAM address (SSRAM or ASRAM) of the first word in the message. Master then instructs EELongTx to begin transmitting. Of course, Master also has to pass some information about the type of the transmit queue to EELongTx. This is because EELongTx needs to know whether it is transmitting from an express, medium, or MemQOut queue.

EELongTx starts the transmit by issuing a read to the SRAM location that contains the message header. In the following two cycles it will continue issuing reads to successive SRAM locations. This data will be wasted in most cases, but can be used during raw message transmits. After the header arrives on the IBus (because of pipelining, 3 cycles after the transmit has begun), EELongTx checks to see whether the header is formatted correctly and none of the permissions have been violated. If a violation has occurred (for example, if a raw message has been sent from a queue that does not permit raw messages to be sent), then the transmit will be aborted and an error flag raised. This error flag will either interrupt one of the processors or be part of state that is visible to SP.

After the header has been read, EELongTx can issue a read to the appropriate translation table entry. After the translation table has been read, EELongTx will perform reads to the remaining part of the message in the message queue. This could be no read if it is an express message with no tagon, or it could be eleven more reads if it is a maximum sized basic message.

As data (header, translation table, remaining message data) are placed onto the IBus, EELongTx instructs TxU to capture the data so that it can be forwarded to the Arctic network. If the message is not raw, then EELongTx also passes some additional data to TxU over dedicated wires. This data specifies the Arctic priority of the message, as well as a few additional bits that are used in conjunction with the translation table entry to decide what the receive queue should be at the receiving node.

Once all the data belonging to the message has been passed to TxU, EELongTx updates the consumer pointer of the queue from which the message was transmitted. This EPRAM update is propagated to the SRAMs so that AP and SP can easily poll the NES to see how full the message queues are.

EELongTx has been designed to be as general as possible. That means that it will allow almost any message to be sent from any message queue provided that the correct permissions are set. For example, a raw basic message with tagon can be sent from an express queue. We believe this additional flexibility will allow us to optimize some of the shared memory protocol steps.

3.1.4 EELongRx

The EELongRx engine begins in very much the same way as a transmit does. First, Master decides that a receive should occur. It then reads the information about the destination queue from the QRAM in QState and passes this information to EELongRx.

EELongRx then determines whether the queue will overflow or not. If the queue will overflow, the message is read from RxU and placed into the overflow queue (mirrored into both ASRAM and SSRAM). Otherwise, the message is read from RxU and put into the appropriate queue in one of the SRAMs. After the message has been fully received, the receive producer pointer in EPRAM is updated and copied into ASRAM and SSRAM.

It turns out to be quite tricky to determine whether a queue will overflow.

EELongRx knows the length of the message (this is passed over dedicated lines from RxU to Ctrl before the receive begins), as well as the producer and consumer pointer of the queue destination (which queue this is can again be determined from some information that is passed over dedicated lines from Ctrl to RxU). In order to process this information quickly, Ctrl precomputes a table that can be used to determine whether a queue will overflow. A table lookup is performed to see whether a queue will overflow.

EELongRx also contains the logic needed to perform DMARx operations. It shares much of the logic that is used for normal receives, but contains some additional logic that is used to split the message. It puts all but the last two 64-bit data words of the message into the data receive queue. The last two words are placed in the DMA bus operation receive queue.

The DMA receive operation is quite similar to tagon receives. Here again EELongRx needs to separate the data portion and the tagon portion. The only difference is that the split occurs at a different time. If the destination of the message is an express message queue, then the first word (the header) is placed into this queue's header queue and the tagon data is placed into the queue's data queue. Since basic message receive queues do not have separate header and tagon queues, all the data goes into a single queue, even if it is a tagon message.

3.1.5 Data Motion

The Data Motion engine (DM) is the simplest of the six major modules. It accepts data motion requests from ABIU and SBIU (these requests first pass through Master), and can also capture MemQOut data requests from the IBus. It contains an internal buffer to queue incoming MemQOut data motion requests.

As soon as Master determines that a data motion should occur, it instructs data motion to begin. If it is initiated from ABIU or SBIU, then Master will also pass the source, destination, and length of the requested data motion. If it is a MemQOut data motion, then data motion will already have captured the MemQOut request from the IBus. (Of course, Master had to read the MemQOut data motion request from the SRAM before the data motion engine could capture it.) Once data motion has begun executing, it simply moves data from one SRAM to the other. The data motion engine reads from the source SRAM and issues the write to the destination SRAM so that it can directly capture the data that was read onto the IBus.

The reader will notice in figure 3.1 that DM can also produce an IBus output. The reason for this is that MemQOut data motion requests can be acknowledged. If they need to be acknowledged, then DM will request a MemQIn compose from ABIU (through Master). Once the request has been granted, DM can place data onto the IBus that will identify what operation is being acknowledged.

3.1.6 Bus Engine

The bus engine (BE) provides CTRL and the service processor an interface to the application processor's memory bus (through the ABIU). This interface is needed to issue DMA read, DMA write, and shared memory requests to the application processor. Since the bus engine does not share many resources with the other components it is more independent than the other modules. It schedules many of its events independently of what the rest of Ctrl is doing and only in rare circumstances needs to request permission to use a shared resource.

The BE is responsible for DMARd, DMATx, DMAWt, Reclaim, and MemQOut 60X operations (MemQOut requests to issue a bus operation onto the application processor's memory bus). All of these operations except for DMATx need access to the NESBuffer interface. This is the interface that allows Ctrl to issue bus operations to the application processor through the ABIU bus master. They gain access to this resource according to the following priorities (listed from highest to lowest priority).

1) MemQOut 60X
2) DMA Wt
3) Reclaim
4) DMA Rd

Access to the NESBuffer is scheduled by BE since no other modules need access to this interface. Other operations, for example issuing of MemQOut 60X operations obviously need to be closely coordinated with Master. The MemQOut 60X operation needs to be performed in FIFO order with respect to the MemQOut operations that other modules perform.

Master will instruct BE to capture the MemQOut 60X operations whenever these are available on the IBus. BE will buffer these operations until the NESBuffer interface is free and Master grants it permission to issue the operation. If a previous MemQOut operation has not completed yet then this operation cannot be issued, since FIFO ordering needs to be enforced.

After BE receives permission to issue the MemQOut bus operation, BE will tell Master that it has done so. If the NESBuffer interface is not free, the bus operation cannot be issued, and the acknowledgement to Master that the operation has been issued is delayed. The acknowledgment that the operation has been issued indicates to Master that a buffer for a new MemQOut bus operation has been freed. Master can then read a new MemQOut bus operation if it is available. Upon completion of the MemQOut bus operation BE will inform Master that the bus operation is finished. Now the next MemQOut operation can be issued and begin executing.

DMA Wt operations are quite simple. The DMA Wt engine observes the DMA Rx consumer and producer pointer of the bus portion of the DMA Rx queue. This allows it to tell whether any bus operations have been received by DMA Rx but have not yet been processed. If this is the case, then DMA Wt will request permission from Master to read one of the bus operations from the queue. If permission is granted, then the bus operation is read and the consumer pointer is incremented. The bus operation is then passed to ABIU. ABIU will issue it to the AP memory bus.

The reclaim engine is the logic in Ctrl that allows the application processor to compose and receive cached basic messages without needing to explicitly clean or flush them. Instead, the reclaim engine performs the cleans and flushes for the processor.

QState maintains state that indicates which basic message queues need to be reclaimed. If a queue needs to be reclaimed, then the reclaim engine will send a request to Master to obtain the producer, consumer, and reclaim pointer for this particular queue. After Master has forwarded the request to QState, and QState has returned the data to the reclaim engine, reclaim can go ahead and start issuing the cleans or flushes.

The difference between the reclaim pointer and the producer pointer (transmit queues) or consumer pointer (receive queues) determines the number of cleans or flushes that need to be issued. Once all the bus operations have completed, reclaim will send a request to Master that indicates that it wants to update the producer or consumer pointer of the queue that was reclaimed. Once access to QState is available, the update occurs. The queue has been reclaimed and it can transmit or receive new messages.

DMA Rd operations are closely correlated to DMA Tx operations since they share a queue. DMA Rd issues the bus operations that transfer data from AP DRAM to the DMA transmit queue in ASRAM. DMA Tx then takes this data, appends two bus operations, and sends them as a message to the destination of the DMA Tx command.

Up to 5 DMA Rd commands can be buffered by BE. So much buffering is added inside Ctrl because DMA commands are composed through MemQOut and we do not want DMA Rds to block other MemQOut operations. Each DMA Rd command specifies the number of cache lines that should be read, as well as the AP DRAM source address. A few other parameters, such as whether the DMA Rd command should be acknowledged are also included.

DMA Tx commands are buffered for the same reason the DMA Rd commands are buffered. The DMA Tx engine transmits the data that was read from AP DRAM by the DMA Rd engine. It is able to reuse the data several times; that is, DMA Rd data can be multicast.

A DMA Tx message is a raw message that contains the data from the DMA Tx queue. It also contains two 64-bit bus operations that specify what the DMA Rx/Wt engine at the destination node should do with the received data.

A DMA Tx command specifies the destination of the DMA operation and how much of the data in the DMA Tx queue should be transmitted. It also contains some flags that allow the user to specify what type of bus operation should be appended to the DMA Tx message. DMA Tx is also able to acknowledge completion of a DMA Tx. It does so by composing an acknowledgement into the MemQIn queue.

3.1.7 Other Modules

I have described the six major modules inside Ctrl. There are a few others that have not been described. These contain mostly combinational logic such as priority encoders or mappings from queue identifiers to actual queue addresses. These were isolated from the other modules in order to provide an additional level of abstraction. Some of them are also on the critical path and putting them in separate modules allows us to optimize their performance.

3.2 Internal State Organization

Ctrl contains three sets of internal state: EPRAM, QRAM, and SysRegs. All are stored inside QState and are accessible to the ABIU and SBIU. EPRAM contains information about transmit and receive queues that Ctrl writes. QRAM contains the queue information that the application and service processors write. SysRegs contain configuration and status information. In the sections below I will describe in greater detail how this state is organized.

3.2.1 Queue Pointers

Before examining how the state is layed out inside Ctrl, I will briefly discuss how queues are managed. As mentioned throughout this thesis, we use a producer-consumer scheme to manage the queues. For most queues we have a 7-bit consumer pointer, a 7-bit producer pointer, a 9-bit base, and a 5-bit bound.

Basic message queues are managed on a per-cacheline basis. This means that consumer and producer pointers point to SRAM addresses that are multiples of four. This granularity allows queues to be larger, but has the drawback of wasting some SRAM space. Consumer and producer pointers from a basic message queue determine SRAM addresses as follows:

SRAMAddr[0:12] = {Base[0:3], Base[4:8] || Ptr[0:4], Ptr[5:6], 2-bits unspecified}

Note: If SRAMAddr[0] == 1, then ASRAM is being accessed; otherwise, SSRAM is being accessed. Each of these SRAMs contains 2048 64-bit entries.

Express message queues fully specify the SRAM addresses (64-bit granularity). The reason for this is that each express message only uses 64 bits of SRAM, whereas basic messages will usually span several cache lines. Express message queues address the SRAM as follows:

SRAMAddr[0:12] = {Base[0:5], Base[6:8] || Ptr[0:2], Ptr[3:6]}

Queues are traversed by incrementing the SRAM address that is determined from the consumer or producer pointer. The pointers wrap around to 0 whenever their first 5 bits match the bound of the queue. Queues are empty whenever the producer pointer equals the consumer pointer.

3.2.2 EPRAM

The EPRAM contains the queue specific state that Ctrl writes. For the receive queues this is the producer pointer and for the transmit queues it is the consumer pointer. Since the processor need to read this state quite frequently, Ctrl maintains a shadow copy of the EPRAM in the SRAMs. AP and SP only need to issue a read to the appropriate SRAM address to obtain the state.

The EPRAM is organized into 8 entries. Each entry contains information about four queues, two transmit and two receive queues (all either express or basic message queues). We call such a set of four queues an endpoint. Each entry also contains a base and mask that are used to determine the location of the translation table during transmits. This allows us to enforce protection at a relatively fine granularity.

If a EPRAM entry contains information about express message queues, then the entry contains the producer pointer of the tagon portion of the two receive queues. This allows the processor to quickly tell whether they have read all the tagon data.

3.2.3 QRAM

QRAM stores all the queue specific information that the processors write. It contains the producer pointer of the transmit queues and the consumer pointer of the receive queues. The QRAM also contains the base and bound of the queues, permission bits, and the consumer pointer of a tagon receive queue if it is an express message queue.

The QRAM does not need to be shadowed in the SRAMs. This is because the processor wrote the data, and hence has no reason to read the data from Ctrl (we provide a facility to do this for debug purposes).

3.2.4 SysReg

The SysRegs contain most of the configuration and status information inside Ctrl. They contain information ranging from priorities that specify what the transmit priority should be to state that specifies whether a queue should be reclaimed.

Many of these registers will be written by both the processors and Ctrl. For example, a processor may set a bit in one of the registers that indicates that a receive queue should be disabled. This would occur if the processor wants to make the current receive queue non-resident to the NES. However, Ctrl may also set this bit if an overflow occurs and no more messages should be sent to the queue unless the SP intervenes.

Other SysReg, for example a register that says which transmit queue is empty, is only set by Ctrl. This register can be reset, but other than that, the processors can only observe its behavior. We have made such registers readable so that bring-up and debugging of the hardware will be easier.

3.2.5 State Encoding

The engineering specification[2] contains the precise specification of how the state is layed out inside Ctrl. This information would not facilitate the readers understanding of how Ctrl operates and has thus not been included here.

3.3 Technology and Tools

Ctrl is implemented in a 0.6 um ASIC. Our chip vendor is ChipExpress, a company that specializes in rapid prototyping. The ASIC contains about 55K gates of logic and several thousand bits of dedicated RAM cells.

We used Cadence Verilog for RTL coding and simulation. We used Synopsys to compile the design to the gate level and to verify pre and post-layout timing. Surprisingly, pre and post-layout timing differed by nearly 50%. We believe this to be the chip vendors fault.

Test vectors were generated by hand. We considered using a full scan design but decided not to. ChipExpress claimed that a full scan design would not hurt timing, but we believe it would have used significant routing resources, and hence hurt performance.

Chapter 4 Summary

Overall, I had a lot of fun working on the control logic of the StarT-Voyager project. I learned a tremendous amount, and think I accomplished a lot. I designed the entire Ctrl ASIC and took it through all the steps so that it could be sent to the chip vendor for fabrication. I look forward to testing the chip in a real StarT-Voyager node, and hopefully it will be used for some interesting research and experiments.

Bibliography

[1] G. A. Boughton. Arctic Routing Chip. In Proceedings of Hot Interconnects II, Stanford, CA, pages 164-173, Aug. 1994.

[2] Boon S. Ang and Derek Chiou, StarT-Voyager: Hardware Engineering Specification (Version 3.2), Feb. 10, 1997.

[3] James C. Hoe and Mike Ehrlich, StarT-Jr: A Parallel System from Commodity Technology, CSG Memo 384, Jul. 1996.

[4] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chaplin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. Proceedings of the 21st Annual International Symposium on Computer Architecture, Chicago, Il. Apr. 1994.