LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Functional Specification of a High-Performance Network Interface Unit on a Peripheral Bus

Computation Structures Group Memo 389

September 30, 1996

James C. Hoe

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310 and Ft. Huachuca contract DABT63-95-C-0150.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Functional Specification of a High-Performance Network Interface Unit on a Peripheral Bus

James C. Hoe

September 30, 1996

Contents

List of Figures

List of Tables

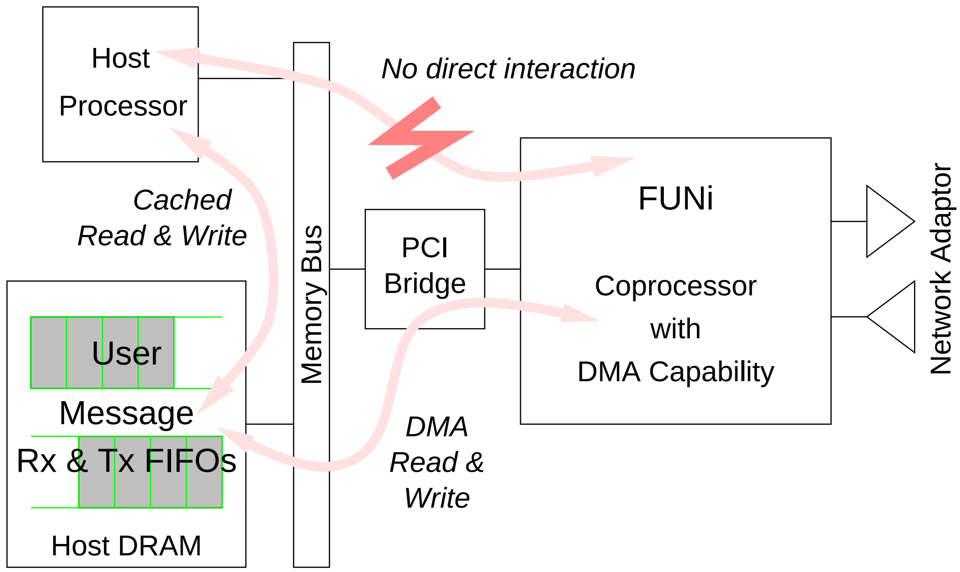

Figure 1: A Message Interface based on Software Queues in the Host DRAM

1 Introduction

This document describes a higher performance reincarnation of START-JR's FUNI[3], retargeted for a larger, more capable class of parallel machines interconnected by MIT's Arctic [2] network. This new NIU (network interface unit) continues to reside on the peripheral I/O bus for portability and ease of implementation. Like FUNI, this NIU will also implement its message-passing interface in the host DRAM to minimize communication overhead on the host processor (see Figure 1 for a depiction). The most important improvement in communication bandwidth and latency will come from replacing the embedded processing aspects of the current FUNI with direct logic implementation in FPGAs (Field Programmable Gate Arrays). Some of FUNI's features are modified or simplified to accommodate for the new implementation and operation environment. Salient features of the newly proposed NIU are:

- Message-passing interface in the host DRAM for Arctic-native-sized messages
- DRAM-base interface queues serve as large and inexpensive network buffers
- High-bandwidth remote DMA with automatic packetization for large data blocks (2 KBytes)
- Two-priority, reliable, ordered network abstraction with back-pressure flow control
- Accessibility to internal NIU states for debugging and context switching

This specification does not assume a specific bus architecture. However, the NIU design depends on the bus to meet the following requirements:

- Bus devices have direct "master" access to the primary host memory, preferably supporting cacheline-size burst at 64-bit data width.
- A combination of bus-level cache consistency, and/or processor cache control and memory barrier instructions must give the host processor efficient control over the order of memory updates as seen by the bus device, and vice versa. The ordering requirement is described further in Section 3.1.

- The bus must be capable of operating at a frequency within the range of a FPGA-based interface implementation (33 MHz or 25 MHz).
- Each network endpoint in the parallel system, whether a SMP or an uniprocessor, must have at least two free bus slots.
- A virtually-addressed bus with memory management is preferred since it eliminates: 1) the responsibility of memory-protection and 2) the requirement for contiguous physical memory allocation.

The remainder of this document presents the NIU specification. Section 2 defines the user-visible communication interface abstraction without going into operational details. Section 3 explains the usage of the interface with examples in C. Section 4 describes the privileged control interface for NIU initialization and configuration by the device driver. Section 5 drafts a NIU block diagram and identifies hardware implementation issues. Section 6 explains the NIU datapath operations under the different interprocessor communication modes. Section 7 briefly discusses the issues regarding interconnecting SMPs as network endpoints. Section 8 concludes this document with a tentative plan for implementation.

Design Constants	Estimated Value	Definition
D	2048	Number of bytes moved per remote DMA operation
N	256	Number of entries per packet queue. Must be a power of 2.
S	256	Maximum system size

Table 1: Summary of Design Constants

2 User-level Network Interface Abstraction

Three interprocessor communication mechanisms are proposed: two for message passing and one for remote DMA block transfer. This section defines the communication interface abstraction visible to an user-level process. This section covers software constructs and memory-mapped registers. Interface usage examples are given in Section 3. Throughout this document, program variables and constructs will be typed in `Courier` font. San Serif font is used to refer to memory-mapped hardware registers on NIU. A single capital letter set in bold font (**A, B, C,** etc.) represents design constants, some of which are not yet determined due to the lack of accurate estimates of hardware resources. Table 1 summarizes these constants and gives the best estimated values.

2.1 Memory-based Message-Passing Interface (MPI) for Low Overhead

This interface is intended to be the primary mechanism for interprocessor communication. The MPI abstraction is based on send and receive FIFO queues at each node. A message enqueued into a send queue eventually appears at the receive queue of the node addressed by that message. Two sets of send and receive queues are implemented to support two priorities of communication, with the guarantee that high priority communications are never blocked by low priority traffic. The choice of network ordering is currently unresolved. Implementing a point-to-point FIFO network implies that we must give up automatic network load balancing using random uproute in the underlying Fat-tree network.

These MPI transmit (Tx) and receive (Rx) queues are implemented using circular buffers in the host DRAM. To minimize communication overhead, the host processor composes and receives messages in these cacheable memory locations and thus avoids long latency accesses to the peripheral NIU device. Communication overhead is transfered to NIU which must retrieve and deliver network packets to these queues using DMA. Although lower overhead is achieved through this memory-based interface, some unwanted communication latency is introduced since intermediate steps are required in this scheme.

2.1.1 MPI Packet Queues

The MPI interface is based on four packet queues (circular buffers) in the host DRAM to support sending and receiving over two network priorities. In this document, the base pointers to these buffers are referred to as `HiTx`, `LoTx`, `HiRx`, and `LoRx` respectively. The virtual addresses of these buffers are acquired by requesting the NIU device driver to mmap() 32[1] 4-Kbyte page from the

[1] Assuming **N**=256.

0th-byte offset of the NIU device file. `HiTx` is assigned the virtual address returned by mmap(). Subsequently, `LoTx=(unsigned long)HiTx+`$\mathbf{N}$`·128`, `HiRx=(unsigned long)HiTx+2·`$\mathbf{N}$`·128`, and `LoRx=(unsigned long)HiTx+3·`$\mathbf{N}$`·128`.

The mmap() device driver routine should allocate the requested memory in pinned DRAM pages. Each of the 256-entry buffers occupies eight 4-Kbyte pages. On systems where address translation (DVMA) is supported on the I/O bus, the pages need not be physically contiguous. However, in any case, the "effective" addresses of the two Tx buffer regions, as seen by NIU on the peripheral bus, must be contiguous. The same requirement holds for the two Rx buffer regions. To simplify implementation, the starting effective address of the Tx buffer region must be ($4\cdot\mathbf{N}\cdot 128$)-byte aligned, and the Rx buffer region must be ($2\cdot\mathbf{N}\cdot 128$)-byte aligned. These device driver and system-level issues will be discussed in Section 4.

2.1.2 Mpi Packet Slots

Each circular buffer is subdivided into $\mathbf{N}$ 128-byte packet slots – each slot is large enough to hold the maximum-sized Mpi packet. Using C notation, the packet slot template is defined as:

```
typedef struct {
    unsigned long header;        /* Mpi message header */
    unsigned long command[2];    /* user message headers */
    unsigned long reserved;      /* padding for alignment */

    unsigned long payload[20];   /* data payload */

    unsigned long pad[8];        /* padding for an 128-byte aligned structure */
} MPIPacket;
```

Given this definition, the base pointer to the circular buffers can be declared as type `(MPIPacket *)`. Thus, indexing the base pointer of a buffer conveniently dereferences the corresponding packet slot. For convenience of discussion, let's also introduce four index variables for the four buffers: `HiTxHd`, `LoTxHd`, `HiRxTl`, and `LoTxHd`. During normal usage, these indices will be incremented monotonically, modulo $\mathbf{N}$. `TxHd`s point to the head (push end) of the packet queues, while `RxTl`s point to the tail (pop end) of the packet queues.

The same Mpi packet template is used in both the Tx and the Rx queues. The `command` and up to 20 words of the `payload` is transfered, in verbatim, from a Tx queue to a Rx queue during message passing. The Mpi message `header` specifies the details of the transfer. The subfields of the Mpi `header` are given below:

Bit Position	Name	Definition	width
31	valid	1 in this field indicates this packet slot currently holds a valid message. Used in producer-consumer handshaking. See Section 3.1	1
30:24	reserved		7
23:16	dest	Virtual node ID of the destination	8
15:13	reserved	For multiple virtual endpoints extension. See Appendix A for explanation.	3
12:6	type	User assigned packet type	7
5	mode	Transfer mode. This field should be 0 for MPI	1
4:0	length	Payload length	5

2.2 DMA Interface (DMA) for High Bandwidth

DMA is the recommended mechanism for transferring a large, contiguous data block. Each DMA request will cause a fixed **D**-byte[2] data block to be moved from the local memory to the remote memory location without interrupting the host processors. This interface is designed to be simultaneously operable with MPI.

2.2.1 DMA Request Queues

A node-to-node DMA transfer is initiated by enqueuing a DMA request into a fifth buffer also in the host DRAM. This buffer is allocated by calling mmap() to map **N**·128 bytes starting from the (4·**N**·128)-byte offset of the NIU device file. The base pointer to this buffer is referred to as `DMATx` in this document. Only a single buffer is used for DMA (whereas the equivalent counterparts for `LoTx`, `HiRx`, and `LoRx` do not exist) because:

- The payload of DMA is delivered directly to the memory of the remote node rather than a receive queue. A notice is generated on the receiver side following each completed DMA. However, this notice is enqueued into the `HiRx` queue instead. Thus, the counterparts of `Hi/LoRx` are not required.

- DMA payloads are always transfered as high-priority traffic because they are always *sinkable* according to the standard request/reply protocol.[3] Hence, DMA requests only have one priority (high), and the equivalent of `LoTX` is not necessary.

2.2.2 DMA Request Slots

The `DMATx` circular buffer is also subdivided into **N** 128-byte slots. `DMATxHd` is the program variable indexing into `DMATx`. Using C notation, the packet slot template is defined as:

[2] **D**=2048

[3] The two network priorities allow the user to implement a request(low)/reply(high) software protocol to avoid deadlock in the user code. Under this simple protocol, the handling of a request message can cause the receiver to respond with any number of reply messages; whereas, reply messages may not generate additional network traffic. Deadlock can be avoided with bounded-buffering resource if each node always attempts to receive inbound messages of equal or lower priority when the node is blocked from sending on a given priority due to network blockage.

```
typedef struct {
    unsigned long header;           /* DMA message header */
    unsigned long command;          /* notification message header */
    void *target;                   /* remote target virtual address of transfer */
    void *source;                   /* local source virtual address of transfer */

    unsigned long payload[20];      /* notification message payload */

    unsigned long pad[8];           /* padding for an 128-byte aligned structure */
} DMARequest;
```

Each enqueued request causes a **D**-byte data block to be transfered from the `source` virtual address on the local node to the `target` virtual address at the remote node.[4] The source and target address must be **D**-byte aligned. The DMA header contains the same subfields as in a `MPIPacket`, except the *mode* field now contains 0x1. After a DMA transfer has completed, a delivery notice is sent to `HiRx` of the receive node. The notice is in the form of a MPI packet composed from the `header`, `command`, `target` and `payload` of the DMA request.

A DMA request can also be enqueued into `HiTx` if the DMA transfer is to take priority over all other activities. If a DMA request is enqueued into `HiTx`, all subsequently enqueued high-priority transmissions will not occur until DMA has been fully transmitted. The purpose of supporting a DMA request queue separate from the MPI Tx queues is to allow MPI transmission (both low and high priority) to bypass long latency DMA operations. If implementation resource becomes scarce, this feature (separate `DMATx` buffer) is not required.

2.3 Direct Message-Passing Interface (dMPI) for Low Latency

To achieve minimum latency (at the expense of host processor overhead), the user may choose to bypass the memory-based MPI interface, and interact with the network hardware directly in dMPI mode. On any one node, dMPI mode is not inter-operable with either MPI or DMA modes. However, if a node can correctly emulate the actions normally carried out by the NIU hardware, it is conceivable that a node operating in dMPI mode can communicate with other NIUs regardless of their operating mode. dMPI mode can be selected independently for sending and receiving since they occur on independent hardware. The mode selection is decided by the control interface discussed in Section 4.1.

2.3.1 NIU Internal Network FIFO Access Points

In dMPI mode, the four internal network FIFO access points – normally accessed by NIU on behalf of the user – is exposed and mapped into the user's address space as memory-mapped registers. HiTxFIFO, LoTxFIFO, HiRxFIFO, and LoRxFIFO are 32-bit memory-mapped registers for sending/receiving high/low-priority network packets. Sending and receiving is accomplished by pushing and popping from these FIFO access points. The last word of each outbound packet must be pushed into HiTxFIFOLast or LoTxFIFOLast respectively to give an implicit "end-of-packet"

[4] These DMA accessible source and target regions may have to be specially allocated or at least specially registered with the help of the device driver so NIU has a convenient way of translating from the user virtual address to the effective address on the I/O bus. See Section 4.3 for further discussion.

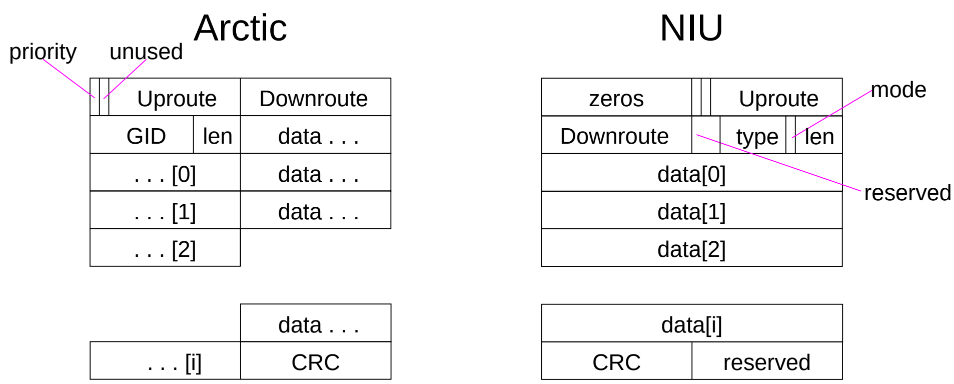

Figure 2: Comparison between Arctic Network Packet Format vs. NIU Interface Format for dMPI

command. This interface is "raw" – meaning to send a packet, the user must enqueue properly-formated Arctic packet according to the Arctic User's Manual[1]. To identify the destination, full Arctic route bits need to be specified instead of virtual node IDs. However, NIU still generates the CRC portion of the Arctic packet in this mode. The receive interface produces exactly what is delivered by the Arctic network, including the Arctic route header and the CRC trailer.

For the purpose of interfacing, the Arctic packet format is skewed sixteen bits by inserting 16 0s at the beginning of the packet. This allows the payload part of Arctic packet to be 32-bit aligned at the FIFO interface. See Figure 2 for clarification.

2.3.2 Network FIFO Status Register

Two auxiliary memory-mapped registers RxStatus and TxStatus indicate when inbound packets are waiting in the RxFIFOs and when the user must stop transmission because a TxFIFO is full.

RxStatus has the following fields:

Bit Position	Definition
31:2	reserved
1	HiRxFIFO is empty
0	LoRxFIFO is empty

TxStatus has the following fields:

Bit Position	Definition
31:4	reserved
3	HiTxFIFO is full
2	LoTxFIFO is full
1:0	reserved

2.4 Summary

This section defined the user-visible interface abstraction for MPI, DMA and dMPI. Table 2 and Table 3 summarize the relevent program variables and memory-mapped registers encountered in this section.

Variables	Type	Definition	Mode
`HiTx`	`(MPIPacket *)`	Base pointer to the high priority transmit queue. Its value is assigned by calling mmap() to the NIU device driver.	MPI,DMA
`HiTxHd`	`(unsigned long)`	Packet slot index of the high priority transmit queue. $0 \leq$`HiTxHd`$< \mathbf{N}$.	MPI,DMA
`LoTx`	`(MPIPacket *)`	Base pointer to the low priority transmit queue. `LoTx=(unsigned long)HiTx+`$\mathbf{N} \cdot$`128`.	MPI
`LoTxHd`	`(unsigned long)`	Packet slot index of the low priority transmit queue. $0 \leq$`LoTxHd`$< \mathbf{N}$.	MPI
`HiRx`	`(MPIPacket *)`	Base pointer to the high priority receive queue. `HiRx=(unsigned long)HiTx+2`$\cdot \mathbf{N} \cdot$`128`.	MPI,DMA
`HiRxTl`	`(unsigned long)`	Packet slot index of the high priority receive queue. $0 \leq$`HiRxTl`$< \mathbf{N}$.	MPI,DMA
`LoRx`	`(MPIPacket *)`	Base pointer to the low priority receive queue. `LoRx=(unsigned long)HiTx+3`$\cdot \mathbf{N} \cdot$`128`.	MPI
`LoRxTl`	`(unsigned long)`	Packet slot index of the low priority receive queue. $0 \leq$`LoRxTl`$< \mathbf{N}$.	MPI
`DMATx`	`(DMARequest *)`	Base pointer to the DMA request queue. Its value is assigned by calling mmap() to the NIU device driver	DMA
`DMATxHd`	`(unsigned long)`	Packet slot index of the DMA request queue. $0 \leq$`DMATxHd`$< \mathbf{N}$.	DMA

Table 2: Summary of Interface Variables and Pointers

Registers	R/W	Definition	Mode
HiTxFIFO	Write-only	NIU hardware FIFO access point for pushing high-priority outbound packets	dMPI
HiTxFIFOLast	Write-only	Equivalent to HiTxFIFO but implicitly indicates that this is the last word of a packet	dMPI
LoTxFIFO	Write-only	NIU hardware FIFO access point for pushing low-priority outbound packets	dMPI
LoTxFIFOLast	Write-only	Equivalent to HiTxFIFO but implicitly indicates that this is the last word of a packet	dMPI
HiRxFIFO	Read-only	NIU hardware FIFO access point for popping high-priority outbound packets	dMPI
LoRxFIFO	Read-only	NIU hardware FIFO access point for popping low-priority outbound packets	dMPI
RxStatus	Read-only	NIU hardware receive FIFO status register	dMPI
TxStatus	Read-only	NIU hardware send FIFO status register	dMPI

Table 3: Summary of Memory Mapped Registers

3 User-level Interface Usage

The previous section presented the NIU interface abstractions. This section gives explanations and examples of using the different communication mechanisms.

3.1 General Circular Buffer Usage

Both the Mpi and Dma interfaces are based on FIFO queues implemented with circular buffers in the host DRAM. The queue abstraction of the circular buffers is jointly maintained by the host processor and NIU. For each buffer, the host processor and NIU split the task of producer and consumer, or vice versa. The handshake between the consumer and the producer is signaled through a *valid* bit attached to every slot in the buffer. The producer maintains a index to the head slot (push end of the queue), and the consumer maintains a index to the tail slot (pop end of the queue) in the buffer. Following initialization, both the consumer and the producer indices are zeroed, and thus the queue is empty.

The producer enqueues by writing into the slot corresponding to its head index. Afterwards, the producer sets the slot's *valid* bit to 1 and increments its head index to the logically-next slot. If at this point, the producer encounters a slot whose *valid* bit is already set, then the queue is full, and the producer must wait for the consumer to dequeue before proceeding with additional entries.

The consumer polls on a queue by checking the *valid* bit of the slot corresponding to its tail index. When the *valid* bit becomes set, the entry is valid, and the consumer can process the content of the slot. When the slot can be reused by the producer, the consumer resets the *valid* bit to 0 and proceeds to poll on the logically-next slot. While polling on an empty Rx queue, as long as the queue remains empty, the header word containing the *valid* bit is unchanged. Thus, each failed poll from the host processor only incurs the overhead of a cache hit. This continues until the producer (NIU) flushes and updates the header word to reflect a new message arrival.

For this handshake to work, the cache coherence and consistency – either automatic or using explicit commands – must guarantee that once the consumer observes the 0 to 1 transition at the *valid* bit, all prior[5] updates from the producer are visible to the consumer.

3.1.1 Alternatives

An alternative scheme for handshaking is to use mailbox registers to pass the head and tail indices between the producer and consumer. This is the scheme employed by StarT-jr. The consideration in StarT-jr is that reading from the Tx queues in the host memory is much more costly than reading from a mailbox register[6]. Since the FUNi coprocessor is time-shared among many tasks, better overall performance is achieved when the mailbox register scheme is used. This problem does not arise in the operation of the Rx queues.

The disadvantage of the mailbox register scheme is that the host processor must perform writes to memory-mapped registers as an integral part of message passing. Fortunately, the host overhead of writing to the index registers can be partially hidden by the write-buffer of the host processor. Furthermore, the index registers do not need to be updated immediately after each message for correct operations. Thus, the overhead of updating the index registers can be amortized over many messages. Depending on the trade-offs, we might resort to the mailbox register scheme for the Tx queues.

[5] Prior in "program"-order or "bus"-order depending on whether the producer is the host processor or NIU.

[6] In StarT-jr, the mailbox registers are included in the bus bridge of the embedded system.

3.2 Mpi Interface Usage

3.2.1 Transmission

To send a message, the user process directly composes the message in the next open packet slot in the Tx queue. The user-level interface uses an abstract message format (simpler than the actual network packet). The following C function sends a high priority message. The contents of the message are passed in as arguments to the function. The example does not check for valid arguments.

```
void TransmitHi(unsigned long dest, unsigned long type,
                unsigned long command0, unsigned long command1,
                unsigned long *payload, int length) {
    int i;
    unsigned long header;

    while ( HiTx[HiTxHd].header & 0x80000000 );          /* step 1 */

    HiTx[HiTxHd].command[0]=command0;                     /* step 2 */
    HiTx[HiTxHd].command[1]=command1;

    for(i=0;i<length;i++) {
        HiTx[HiTxHd].payload[i]=payload[i];
    }

    header=( 0x80000000 |                                 /* step 3 */
             (dest << 16 ) |
             (type << 6 )  |
             length );

    STORE_BARRIER;                                        /* step 4 */

    HiTx[HiTxHd].header=header;                           /* step 5 */

    HiTxHd++;                                             /* step 6 */
    HiTxHd%=N;
}
```

The basic steps involved in sending a message are:

Step 1. Make sure the current slot (`HiTx[HiTxHd]`) is free. In this simplified example we simply spin-wait until ready.

Step 2. Compose and write the body of the message in the Mpi packet slot.

Step 3. Compose the Mpi header using bit-field operations.

Step 4. Perform any necessary barrier operations to insure the previous updates are visible to NIU.

Step 5. Update the Mpi header together with the *valid* bit set to complete the enqueue process.

Step 6. Increment `HiTxHd` modulo **N** to prepare for the next message.

3.2.2 Reception

Inbound messages from the network are delivered to one of the two Rx queues according to their network priorities. The following function polls and receives from `HiRx`. This example is blocking. The inbound message is received into the space allocated by the caller to the function.

```
void ReceiveHi(unsigned long *mode, unsigned long *type,
               unsigned long *command0, unsigned long *command1,
               unsigned long *payload, int *length) {
    int i;
    unsigned long header;

    while (!( HiRx[HiRxTl].header & 0x80000000 ));       /* step 1 */

    READ_BARRIER;                                        /* step 2 */

    header=HiRx[HiRxTl].header;                          /* step 3 */
    *mode=(header&0x00000020)>>5;
    *type=(header&0x00001fc0)>>6;
    *length=(header&0x0000001f);

    *command0=HiRx[HiRxTl].command[0];                   /* step 4 */
    *command1=HiRx[HiRxTl].command[1];

    for(i=0;i<*length;i++) {
        payload[i]=HiRx[HiRxTl].payload[i];
    }

    HiRx[HiRxTl].header=0x0;                             /* step 5 */

    HiRxTl++;                                            /* step 6 */
    HiRxTl%=N;
}
```

The basic steps involved in receiving a message are:

Step 1. Poll the *valid* bit of the current slot (`HiRx[HiRxTl]`) until it becomes valid.

Step 2. Perform any necessary barrier operations to insure the previous updates by NIU are now visible from the host processor.

Step 3. Parse MPI header words to determine the message *type* and *length*. (Note: the *mode* and *dest* fields of the header word is not used in the receive queue.)

Step 4. Receive the message by reading the `command` words and *length*-number of `payload` words from the slot.

Step 5. Reset the *valid* bit to release the slot back to NIU for reuse.

Step 6. Increment `HiRxTl` modulo **N** to prepare for the next message.

3.3 DMA Interface Usage

The C function below initiates a DMA transfer to move **D** bytes of data from the local `source` address to the remote `target` address on the node `dest`. The example does not check for valid arguments.

```
void TransmitDMA(unsigned long dest, unsigned long type,
                 unsigned long command,
                 void *target, void *source) {
                 unsigned long *payload, int length) {
    int i;
    unsigned long header;

    while ( HiDMA[HiDMAHd].header & 0x80000000 );          /* step 1 */

    HiDMA[HiDMAHd].command=command;                        /* step 2 */
    HiDMA[HiDMAHd].target=target;
    HiDMA[HiDMAHd].source=source;

    for(i=0;i<length;i++) {
        HiDMA[HiDMAHd].payload[i]=payload[i];
    }

    header=( 0x80000020 |                                 /* step 3 */
             (dest << 16 ) |
             (type << 6 )  |
             length );

    STORE_BARRIER;                                        /* step 4 */

    HiDMA[HiDMAHd].header=header;                         /* step 5 */

    HiDMAHd++;                                            /* step 6 */
    HiDMAHd%=N;
}
```

The basic steps involved in registering a DMA request are:

Step 1. Make sure the current slot (`DMATx[DMATxHd]`) is free. In this example we simply spin-wait until ready.

Step 2. Compose the body of the notification packet in the slot.

Step 3. Compose the DMA header using bit-field operations.

Step 4. Perform any necessary barrier operations to insure the previous updates are visible to NIU.

Step 5. Update the DMA header word together with the *valid* bit set to complete the enqueue process.

Step 6. Increment `DMATxHd` modulo **N** to prepare for the next DMA request.

3.4 Direct Message Passing Interface

The code segments below demonstrate the usage of the dMPI interface. The transmit and receive functions perform identical tasks as those given in Section 3.2. The code assumes the existence of an `UpRoute[S]` and a `DownRoute[S]` table. Notice the Arctic packet format is skewed by 16 bits as described in Section 2.3.1.

3.4.1 Transmission

```
void TransmitHiDirect(unsigned long dest, unsigned long type,
                      unsigned long command0, unsigned long command1,
                      unsigned long *payload, int length) {
    int i;

    /* wait if FIFO is full */
    while ( TxStatus & 0x00000008 );

    /* push Arctic header */
    HiTxFIFO= 0x00008000 | (UpRoute[dest]);
    HiTxFIFO= (DownRoute[dest]<<16) | ( (type << 6 ) |
                                        length + 4 );
    /* push Arctic body except last */
    HiTxFIFO=command0;
    HiTxFIFO=command1;

    for(i=0;i<(length-1);i++) {
        HiTxFIFO=payload[i];
    }

    /* push last word */
    HiTxFIFOLast=payload[i];
}
```

3.4.2 Reception

```
void ReceiveHi(unsigned long *mode, unsigned long *type,
                unsigned long *command0, unsigned long *command1,
                unsigned long *payload, int *length) {
    int i;
    unsigned long header;

    /* wait until FIFO is not empty 8/
    while ( RxStatus & 0x00000002 );

    /* Retrieve and through Artic Route bits */
    header=HiRxFIFO;
    header=HiRxFIFO;

    /* Parse header for type and length */
```

```
    header=HiRxFIFO;
    *type=(header&0x00007fe0)>>6;
    *length=(header&0x0000001f)-4;

    /* Retrieve packet body */
    *command0=HiRxFIFO;
    *command1=HiRxFIFO;

    for(i=0;i<*length;i++) {
        payload[i]=HiRxFIFO;
    }
}
```

4 Privileged Control Interface

This section describes the privileged memory-mapped control registers used in initialization and configuration of the NIU hardware. These registers are to be protected from user accesses through memory management.

4.1 Operation Control

Two memory-mapped registers, TxCntrl and RxCntrl, independently control the sending and receiving NIU datapath. Within each register, only two bits are active. The subfields for both control registers are:

Bit Position	read/write	Definition
31:2	reserved	
1	write only	Reset
0	read/write	MPI+DMA On/off

4.1.1 Reset

Setting the Reset bit causes a complete low-level reset of the NIU hardware. The hardware will restart in a disabled state following a reset. All NIU control states must be reinitialized after a reset.

4.1.2 MPI and DMA On/off

Writing OFF to TxCntrl causes the transmit NIU's MPI and DMA FSM to stop retrieving additional packets from the host DRAM after the completion of any retrieval currently in progress. However, the internal datapath continues to be active until all previously retrieve packets have entered the network FIFOs. Thus, after writing OFF to TxCntrl, the processor must continue to check TxCntrl until it reads OFF to be certain that all transient states have been cleared through the transmit datapath. One should note that if a blocked network prevents transient packets in the NIU datapath from entering the network, the transmit datapath may never enter the OFF state.

Writing OFF to RxCntrl will stop MPI-mode and DMA-mode delivery of additional inbound packets to the host DRAM after the completion of any delivery currently in progress. Additional inbound packets will pile up in the internal NIU buffer space and eventually back up into the network itself.

The sending or receiving datapath must be in the OFF state before the remaining control registers corresponding to that part of the datapath are modified. When NIU is OFF, the interface is in dMPI mode. The user can send and receive by accessing the hardware FIFO access points.

4.2 Buffer Control

A TxQBase register in the transmit datapath contains the base pointer to the ($4 \cdot \mathbf{N} \cdot 128$)-byte aligned Tx buffer region (`HiTx`, `LoTx`, and `DMATx`). A RxQBase register in the receive datapath contains the base pointer to the ($2 \cdot \mathbf{N} \cdot 128$)-byte aligned Rx buffer region (`HiRx` and `LoRx`). The alignment requirement allows the NIU hardware to generate addresses to the packet queues using concatenation instead of addition.

Only the upper ($23 - \log \mathbf{N}$) bits of TxQBase and the upper ($24 - \log \mathbf{N}$) bits of RxQBase memory-mapped registers are active.

As described in Section 3.1, NIU, acting either as the producer or consumer, maintains its own set of indices into each of the five interface queues. These internal state registers: HiTxTl, LoTxTl, HiRxHd, LoRxHd and DMATxTl are exposed to system-level software for reading and writing to allow user-transparent context switching and to assist in hardware debugging. Only the lower (log **N**) bits of these memory-mapped registers are active.

4.3 DMA Space

.

Given a peripheral bus with DVMA capability (address translation and access protection), NIU can directly process the virtual address supplied by the user process in DMA requests. No additional control is associated with NIU. (Instead, the complexity has been transfered to the address translation facility at the I/O bus bridge.)

However, if memory-management is not, or only partially, available, then NIU must carry out the necessary translation and protection when processing DMA transactions. DMABasePhy, DMABaseVir and DMARangeMask enable NIU to perform a rudimentary (base & bound) linear address translation and protection. Prior to utilizing DMA, the user process must request the device driver to allocate a contiguous virtual memory region where all DMA transactions must fall within. The device driver, in turn, should allocate contiguous physical memory to back up the region. The starting addresses (virtual and physical) and a mask reflecting the size of the region is loaded to the corresponding registers. To simplify the NIU implementation, the starting addresses of both the physical and virtual region must be 4-MByte aligned. The region size is also required to be a power of 2 but not less than 4 MBytes. The range mask is the region size minus one byte. Given these constraints, only the upper 10 bits of the registers need to be active.

A set of these three registers exist separately for the transmit and receive datapath on each node. However, we logically consider them to be the same registers since during normal operation, the corresponding contents should be identical.

4.4 Network Route Table

The transmit datapath contains an additional two-by-**S**-entry RouteTable[2][**S**]. This memory-mapped region is backed by SRAMs on-board NIU instead of hardware registers. Two entries (to be used for the two priorities) exist for each possible network endpoints in the system. The entries hold the source-base Arctic up and down route headers for the Fat-tree network. Each entry is 4-byte and have the following format:

```
typedef struct {
   unsigned downroute:16;          /* Arctic Downroute */
   unsigned priority:1;            /* 0x1 if hi priority else 0x0 */
   unsigned reserved:1;            /* unused bit */
   unsigned uproute:14;            /* Arctic Uproute */
} RouteEntry;
```

If we choose to allow unordered network abstraction, the NIU hardware will generate part of the up-route field randomly to take advantage of the load-balancing characteristics of the underlying Fat-tree network.

4.5 Summary

Table 4 summarize the memory-mapped registers encountered in this section.

Registers	R/W	Definition
TxCntrl	read/write	Control register for the transmit datapath
RxCntrl	read/write	Control register for the receive datapath
TxQBase	read/write	Base pointer to the $4 \cdot \mathbf{N} \cdot 128$-byte-aligned Tx buffer region
HiTxTl	read/write	Packet slot index of the high priority transmit queue. $0 \leq \mathsf{HiTxTl} < \mathbf{N}$.
LoTxTl	read/write	Packet slot index of the low priority transmit queue. $0 \leq \mathsf{LoTxTl} < \mathbf{N}$.
DMATxTl	read/write	Packet slot index of the DMA request queue. $0 \leq \mathsf{DMATxTl} < \mathbf{N}$.
RxQBase	read/write	Base pointer to the $2 \cdot \mathbf{N} \cdot 128$-byte-aligned Rx buffer region
HiRxHd	read/write	Packet slot index of the high priority receive queue. $0 \leq \mathsf{HiRxHd} < \mathbf{N}$.
LoRxHd	read/write	Packet slot index of the low priority receive queue. $0 \leq \mathsf{LoRxHd} < \mathbf{N}$.
DMABasePhy	read/write	Physical starting address of the DMA region
DMABaseVir	read/write	Virtual starting address of the DMA region
DMARangeMask	read/write	Specifies the size of the DMA region

Table 4: Complete Memory Mapped Priviledged Control Registers

5 Hardware Datapath and Implementation

The NIU hardware is composed of two PCI adaptor cards, one dedicated to the transmit datapath and another dedicated to the receive datapath. Although the two cards perform different functions, they will have identical hardware. Figure 3 gives a block diagram of the PCI card. The difference between the transmit and receive cards is only in the configuration of the NIU Transaction Core FPGA.

The two-card design decision is made under the assumption that combining the full transmit and receive datapath onto a single card is difficult. Since we most likely want to use more than one network adaptor card per node to increase network access bandwidth anyways, having separate Tx and Rx cards per node is more effective – both in implementation and operation – than using two full-featured cards per node. In any case, both the Tx and Rx cards are bi-directional in dMPI mode.

We started to investigate the possibility of using an existing commericial PCI protoboard populated with FPGAs. So far we have come across the DEC Pamette board and the Giga Operations PCI protoboard. Neither promises 64-bit operation currently. However, the largest obstacle is that these board are in 12-inch form-factors, which may not physically fit into some of the host machines. Nevertheless, they are useful for prototyping in the early stages of development.

5.1 PCI Adaptor Card Overview

5.1.1 PCI Interface Core

The PCI bus interface logic will be implemented using FPGAs (Xilinx??). We hope this part of the logic can be acquired as a synthesizable Verilog module from a commercial vendor. Xilinx LogiCore PCI offers a 32-bit implementation (and whose bus master module is only in pre-release stage). Logic Innovations has a 64-bit master-capable module that occupies approximated 8K gates. We have also requested literature from: Sand Microelectronics, Eureka Technologies, Toucan Technologies, and Virtual Chips.

A preliminary online survey has revealed several vendors offering a 64-bit PCI implementation. It is not clear how suitable they are for FPGA synthesis, although most of them at least claim to target FPGAs. Most packages conveniently include a PCI bus simulation as a design test bench. The conclusion from this survey seems to indicate that a 64-bit design at 33 MHz is within reach. We do not expect to attain 66 MHz unless faster implementation technologies (LPGAs, ASICs, etc.) are considered.

5.1.2 NIU Transaction Core

The NIU Transaction Core will also be implemented using FPGAs. The NIU Transaction Core may need to be partitioned so it partially sits in the PCI interface FPGA to reduce the pin count between FPGAs. We anticipate using a total of two large FPGAs to implement the PCI and NIU cores. The majority of the NIU Transaction Core design can be based on existing C-code from START-JR's FUNI embedded processor. Stripped down C-code, as functional specification, can be adapted to synthesizable Verilog.

The NIU Transaction Core contains FSMs and software accessible control registers. The operations of the NIU Transaction Core is described in Section 6. Most of the datapath will not flow through the NIU Transaction Core FPGA. Only selected signals will be interpreted by the NIU Transaction Core. The decision to partition the NIU design into separate transmit and receive cards was in part to reduce the complexity of the NIU Transaction Core.

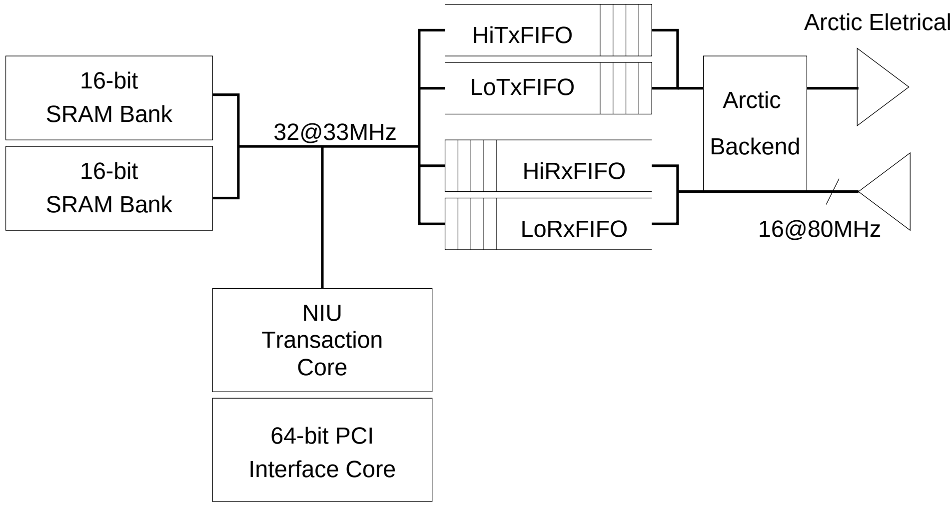

Figure 3: NIU PCI Card Datapath

5.1.3 SRAM Banks

Two 16-bit SRAM banks totaling at least 1 Kbytes of storage is used to hold the Arctic Fat-tree route table. The PCB design should be such that the SRAM banks can be left unpopulated on the Rx cards.

5.1.4 On-board Network FIFOs

Four 32-bit FIFOs bridge the 33MHz PCI bus clock domain and the 40MHz Arctic backend clock domain. Two FIFOs are provided in each direction to support the two Arctic network priorities. We may be able to conserve board space by using only two bi-directional FIFOs whose direction can be set depending on whether they are in the transmit or the receive datapath. However, having a full set of FIFOs increases the testability of the cards and allows each card to function as a stand alone interface in dMPI mode.

5.1.5 Arctic Backend and Arctic Electrical

The Arctic backend design includes 1) digital issues such as protocol handshaking and CRC-generation, and 2) analog issues such as re-timing and ECL signal conversion for transfering over cables. The Arctic backend design has been worked out as part of the StarT-jr and StarT-Voyager projects. The remaining concern is in power consumption, heat dissipation and space.

5.2 Transmit Card

Figure 4 illustrates how the transmit interface registers map onto the transmit card.

5.3 Receive NIU Card

Figure 5 illustrates how the receive interface registers map onto the receive card.

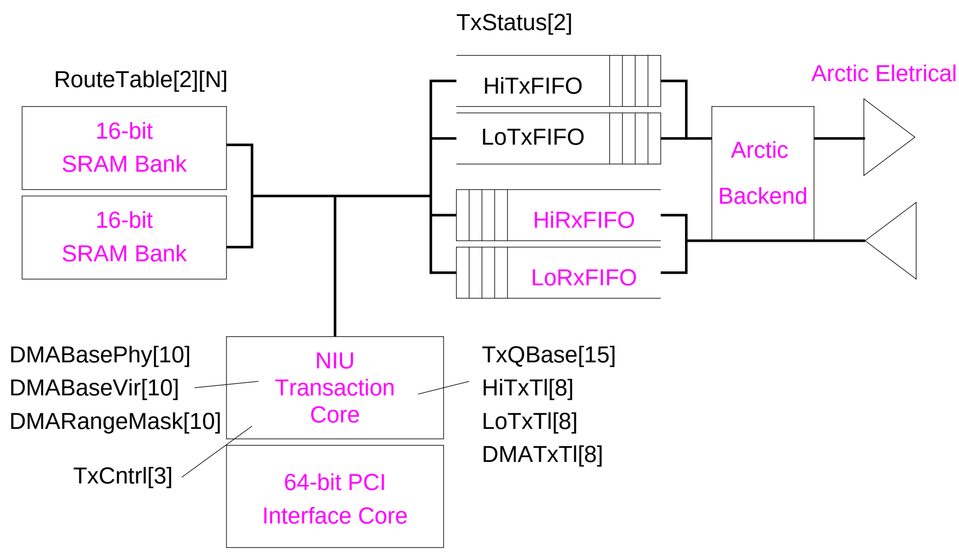

Figure 4: Transmit PCI Card with Control Registers Shown

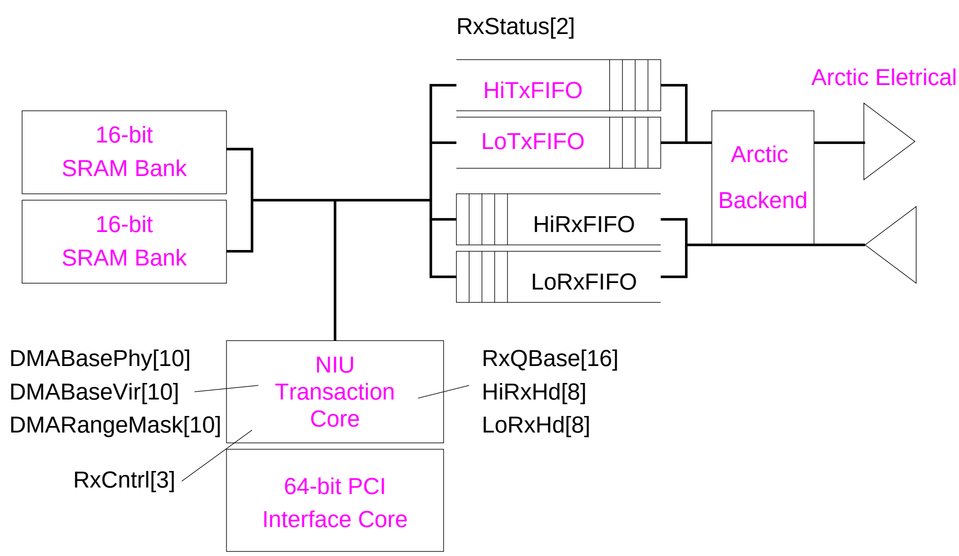

Figure 5: Receive PCI Card with Control Registers Shown

6 NIU Operations

6.1 dMPI Mode

When the NIU Transaction Core is in the OFF state, the corresponding datapath of NIU is in dMPI mode. The NIU Transaction Core is mostly inactive in this mode. The only function provided is passive handling of accesses to memory-mapped registers, SRAMs and the internal FIFO access points.

6.2 Transmit Datapath with MPI and DMAFSM Enabled

When enabled, the NIU Transaction Core's two FSMs on the Transmit NIU Card are responsible for polling the three Tx queues: `HiTx`, `LoTx` and `DMATx`.

6.2.1 MPI FSM

MPI FSM (MPFSM) is responsible for polling the `HiTx` and `LoTx` message passing queues. MPFSM repeatedly executes the following actions:

Step 1. Compose the address of the tail slot in `HiTx` by concatenation:
currAddr=TxQBase,b'00,HiTxTl,b'0000000.

Step 2. Perform a direct memory read to **currAddr** to fetch the MPI header.

Step 3. Check the *valid* bit of the MPI header. If the *valid* bit is not set, then skip to Step 9.

Step 4. Check the *mode* bit of the MPI header. If it is a DMA request, then skip to Step 11.

Step 5. Enqueue the Arctic route headers into HiTxFIFO from RouteTable[0][*dest*] where *dest* is from the MPI header . The RouteTable format is given in Section 4.4. The first Arctic header word is composed of 16 zero upper bits and lower 16 bits from the table. The second word is composed of the upper 16 bits from the table and lower 16 bits of the MPI header word[7]. Refer to the NIU-Arctic Packet format shown in Figure 2.

Step 6. Enqueue the two user MPI command words from **currAddr+4** and **currAddr+8**. Refer to Section 2.1.2 for the MPI packet format.

Step 7. Retrieve the required number of MPI payload words using 4-word-burst reads starting from **currAddr+16**. Enqueue the MPI payload words to HiTxFIFO and issue a *last word* command on the last payload word.

Step 8. Write a 0x0000000 to **currAddr** to release the slot. Increment HiTxTl.

Step 9. Compose the address of the tail slot in `LoTx` by concatenation:
currAddr=TxQBase,b'01,LoTxTl,b'0000000.

Step 10. Carry out similar steps for the low priority transmit queue, and repeat from Step 1.

Step 11. Signal DMAFSM to take over handling of this slot. Wait until DMAFSM has signaled the completion of the current slot.

Step 12. Write a 0x0000000 to **currAddr** to release the slot. Increment HiTxTl and repeat from Step 1.

[7]The *length* from the MPI header needs to be incremented by 4 since the *length* field has different interpretations in the MPI and the Arctic packet formats.

6.2.2 DMA FSM

A DMA FSM (DMAFSM) is responsible for polling `DMATx` and occasionally helping MPFSM service a DMA request in `HiTx`. DMAFSM repeatedly executes the following actions:

Step 1. Compose the address of the tail slot in `DMATx` by concatenation:
currAddr=TxQBase,b'10,DMATxTl,b'0000000.

Step 2. Perform a direct memory read to **currAddr** to fetch the DMA header.

Step 3. Check the *valid* bit of the DMA header. If the *valid* is not set, then repeat from Step 2.

Step 4. Check the *mode* bit of the packet header word. If it is a not DMA request, then turn OFF the transmit datapath to indicate an error.

Step 5. Fetch the *source* and *target* addresses from **currAddr+8** and **currAddr+12**. Refer to Section 2.2.2 for the DMA request format.

Step 6. If required, translate *source* into effective bus address.

Step 7. As part of the DMA request, the DMAFSM needs to generate 32 Arctic packets, each carrying 16 words of the total 2048-byte block transfer. The following loop is executed 32 times:

1. Enqueue the Arctic route header into HiTxFIFO from RouteTable[0][*dest*] as in MPI except the length field is fixed to 19 words.
2. Enqueue (*target* + **offset**)[8] where **offset** is initialized to 0x0 and incremented by 64 bytes after each inner loop.
3. Retrieve 16 words in bursts from (*source*+**offset**). Enqueue the DMA data words to HiTxFIFO and issue a *last word* command on the last data word of this packet.

Step 8. Signal MPFSM to generate a notification packet by interpreting the current DMA request slot as a normal packet slot[9]. Wait until MPFSM has signaled the completion of the current message.

Step 9. Write a 0x0000000 to **currAddr** to release the slot. Increment DMATxTl and repeat from Step 1.

DMAFSM operates independently from MPFSM except when they are contending for shared datapath and when they are assisting each other in the handling of DMA requests.

For DMA transmissions, DMAFSM needs to ensure that the same Arctic routes are used for all Arctic packets that are generated from the same DMA request to maintain the point-to-point FIFO property between the sender and the receiver.

6.3 Receive Datapath with MPI and DMAFSM Enabled

When enabled, the NIU Transaction Core (RCVFSM) on the Receive NIU Card is responsible for polling RxStatus and receiving the inbound Arctic packets from HiRxFIFO or LoRxFIFO accordingly. A packet is categorized as a MPI packet or a part of a DMA transfer by checking the *mode* bit of the inbound packet. The packet is then delivered to either `HiRx`, `LoRx`, or to the memory location specified in a DMA packet.

[8]This addition is actually only a concatenation due to our alignment requirement on the *source* and *target* address.

[9]In this case, MPFSM should be careful to not interpret the *mode* bit of the DMA request slot. MPFSM should also clear the *mode* field in the Arctic packet transmitted.

6.3.1 Mpi Receive

When processing a Mpi packet, the Arctic route header words are thrown away except for the 16 bits containing the *mode*, *type* and *length*.

RCVfsm executes the following steps for a Mpi packet:

Step 1. Compose the address of the head slot by concatenation:
For high-priority packets, **currAddr**=RxQBase,b'0,HiRxHd,b'0000000,
and for low-priority packets, **currAddr**=RxQBase,b'1,LoRxHd,b'0000000.

Step 2. Perform a direct memory read to **currAddr** to fetch the Mpi header.

Step 3. Check the *valid* bit of the Mpi header. If the *valid* bit is already set, then the queue is full. Repeat from Step 2.

Step 4. Perform direct memory writes of the first two Arctic data words to **currAddr+4** and **currAddr+8** as `command0` and `command1` of the Mpi packet. Refer to Section 2.1.2 for the Mpi packet format.

Step 5. Write the remaining Arctic data words using 4-word-burst writes[10] to the Mpi payload field starting at **currAddr+16**.

Step 6. Write the Mpi packet header word containing the *mode*, *type* and *length*[11] together with the *valid* bit set to **currAddr**.

Step 7. Increment HiRxHd (or LoRxHd) and repeat from the checking of the RxStatus register.

6.3.2 Dma Transfers

When processing a Dma packet, the Arctic route header words are thrown away once the *mode* bit has been examined.

RCVfsm executes the following steps for a Dma packet:

Step 1. Dequeue the *target* address from the first Arctic data word.

Step 2. If required, translate *target* into effective bus address.

Step 3. Perform burst memory writes of the next 16 Arctic data words to the host memory starting at *target*.

6.4 Summary

In this section, we attempted to give a functional description. A more precise specification in HDL is necessary in the future.

[10] Padded with dummy words if the payload size is not 4-word aligned.

[11] *length* from the Arctic header is decremented by 4 for the Mpi packet header.

7 SMP Considerations

This section discusses the issues dealing with interconnecting a cluster of SMPs (symmetric multiprocessors) with the proposed NIU design. In particular, how NIU (or NIUs) is shared by the different processors at a single network endpoint. A few scenarios are described.

7.1 NIU as a Shared System Resource

In this straight forward scenario. NIU is managed as a protected system resource, much like normal LAN hardware. User processes communicate through standard Unix socket-based interface calls. NIU details are hidden. User processes can still benefit from the increased bandwidth on block transfers. However, low-overhead, fine-grain message passing features supported by the NIU's MPI mode are lost.

7.2 Cooperating Processors with a Dedicated Coprocessor

In this model, we assume the processors are all cooperating on a single application. One processor, running user code, is assigned to handle all communication events on this "node". This scenario is similar to the previous case except the NIU's interface is exposed directly to user-level processes. In some programs, the user application may benefit from both increased bandwidth and lowered communication overhead.

7.3 Cooperating Processors sharing NIU protected by Semaphores

Again we have a scenario where the processors are all cooperating on a single application, but more tightly-coupled. For this scenario to work, processors on a node cannot be individually identified. In other words, it should not matter which processor handles any one particular inbound message. For example, we can image a system where remote procedure calls are posted to a node through NIU, and any free processor on that node can handle the remote procedure calls.

In this model, NIU is shared amongst the processors by protecting access to the Tx and Rx queues with semaphores. By carefully managing the exclusive access to the `TxHds` and `RxTls`, it is possible to share NIU in such a way that multiple processors can simultaneously enqueue or dequeue from the same queue, but working on different slots. This system has greater potential for fully utilizing the capability of the NIU design.

7.4 Independent Processes Requiring Direct Access to its Own NIU

Although the Tx queues can still be shared using semaphores, each process must have its own Rx queues. This requirement can be met in a few ways.

- If enough I/O slots can be made available, we can dedicate one set of NIU cards to each processor.
- The Tx and Rx datapath can be combined into a single PCI adaptor card so more processors can be serviced by fewer card slots.
- Instead of servicing a single set of queues, the NIU datapath can be altered to support multiple sets of Tx and Rx queues. In effect, a single NIU is servicing multiple virtual endpoints. The 3-bit reserved field in the MPI/DMA headers can be used to extended the *dest* field to identify up to eight virtual endpoints serviced by each NIU. See Appendix A for the changes required to support this extension.

8 Implementation Plans

This design attempts to leverage from the lessons learned from the StarT family of projects. The features from StarT-jr's FUNi have been trimmed to make the proposed NIU implementation feasible in the amount of time and resources. However, the performance of the design has not been sacrificed. If this proposed NIU cannot sustain at least on the neighborhood of 80 MBytes/sec (64-bit@33 MHz, or 40 MBytes/sec at 32-bit@33 MHz) in one direction in Dma mode, then we need to re-evaluate the practicality of this design. (The design should try to make full use of the available bus bandwidth.) The Mpi mode will hopefully be able to sustain greater than 20 MBytes/sec and incur less than 1 μsec of overhead on the host processor.

8.1 Hardware Development Resources

Aside from the parts for final assembly, the following resources need to be secured for hardware development:

- Synopsis and compatible FPGA development tools (Xilinx??)
- Commercial synthesizable PCI interface core capable of 64-bit bus width at 33 MHz in FPGA form
- PCI Bus simulation to verify the design against (usually included in the PCI interface core package)
- PCB design and layout expertise[12]

8.2 Software Development Resources

We need some expertise in Solaris device driver development. The minimum functions required from a bare-bone device driver are:

- Allocate pinned physical memory
- Setup DVMA translation to host memory for NIU
- Support mmap() to NIU control locations and pinned DRAM pages

For more user-friendly access, we need to develop full-featured device drivers and the IP layers.

8.3 Testing Strategy

The full NIU design will be captured in Verilog. With the PCI bus simulator, the design will be extensively simulated during the debug phase of the design cycle. Prior to committing to the PCB layout, the captured interface design will also be tested using a commercial FPGA-based PCI prototyping board. First round hardware bring-up and debugging will be carried out using inexpensive and readily-available Linux PCs or SUN workstations. The NIU hardware will be tested against matured StarT-jr NIU hardware over the Arctic network. Only a fully-debugged design will be installed in the final system.

[12] We need the most outside help in this area.

References

[1] A. Boughton, G. Papadopoulos, B. Greiner, S. Asari, S. Chamberlin, J. Costanza, R. Davis, T. Durgavich, D. Faust, E. Heit, T. Klemas, J. Kwon, E. Ogston, G. Rao, and R. Tiberio. *Arctic User's Manual*, 1993.

[2] G. A. Boughton. Arctic routing chip. In *Proceedings of Hot Interconnects II*, August 1994.

[3] J. C. Hoe and M. Ehrlich. StarT-Jr: A parallel system from commodity technology. Technical Report CSG Memo 384, MIT Laboratory for Computer Science, July 1996.

A Virtual End-point Extension

Currently, a network endpoint is composed of five queues: `HiTx`, `LoTx`, `DMATx`, `HiRx`, and `LoRx`. Each endpoint is serviced by one set of Tx and Rx NIU cards. A simple change can be made to enable one set of NIU to service a number of virtual endpoints to be used independently by different processes of the same parallel application. The aggregate bandwidth through NIU will not change significantly. However, when any one virtual endpoint user causes the network to block, all other virtual endpoints serviced by the same NIU will also block.

A.1 Tx Modifications

Three bits have been allocated in the MPI header to identify up to eight virtual endpoints per NIU. All eight sets of `Tx` queues need to be allocated contiguously and be $(4 \cdot \mathbf{N} \cdot 128)$-byte aligned. The eight different virtual endpoints can be selected by setting the three address bits above the alignment boundary.

MPFSM and DMAFSM maintain a 3-bit COUNT register that are incremented following each polling/service cycle. An additional 3-bit privileged control register, VMASK is introduced. At the beginning of each polling cycle, the COUNT is *binary-and*ed with VMASK to select one of the eight sets of Tx queues to service next.

For example, the address of the tail slot in `HiTx` of the selected endpoint is:
currAddr=TxQBase[13],(COUNT&VMASK),b'00,HiTxTl,b'00000. When VMASK=b'111, each of the eight virtual endpoints will be visited in sequence. When VMASK=b'000, only the 0th endpoint is serviced. Two and four endpoint nodes can be be configured similarly.

The remaining MPFSM and DMAFSM operations are unaffected.

A.2 Rx Modifications

Bit[15:13], just below the *destination* field, in the 2nd header word of MPI and DMA template has been reserved to specify one of the eight virtual endpoints at the destination node as the receiver. These three bits are transmitted with the corresponding Arctic packet. When delivering an inbound message, RCVFSM needs to decode these additional three bits to determine the designated virtual endpoint. Again by requiring the Rx queues to be aligned, these three bits can be directly applied to compose the delivery address by concatenation. The remainder of the RCVFSM operation is not affected by this change.

[13] TxQBase is shortened by 3-bits.